All Slides

Nil Ozer

A&W

A&W

Exercise Session 1

Introduction

Nil Ozer

Outline

- Logistics
- A&W Overview
- Exam
- How to study for A&W
- Get to know me/you

- Warm up exercise

Logistics

- Exercise Session here and on wednesday only for today !!

- Normally : **Thursdays, 16:15 - 18:00 , HG E 33.1**

Logistics

- Programming Exercise
  - Every week, starting on 2. week
  - CodeExpert
  - 2 points, automatically graded
- Theory Exercise
  - Even weeks, starting on the 2. week
  - until 10:00 on the following Thursday
  - 2 points, TA graded
- Peer Grading Exercise
  - Odd weeks, starting on the 3. week
  - 2 points (upload + peer grading), TA graded
- Mini Quiz
  - Even weeks, starting on the 2. week
  - First ~5 min of the exercise class
  - 2 points

Logistics

W1		Warm up exercise	
W2	Mini Quiz	Theory Exercise	Programming Exercise
W3		Peer Grading Exercise	Programming Exercise
W4	Mini Quiz	Theory Exercise	Programming Exercise
W5		Peer Grading Exercise	Programming Exercise
W6	Mini Quiz	Theory Exercise	Programming Exercise
W7		Peer Grading Exercise	Programming Exercise

...

Logistics

- Bonus Point Calculation
  - >= %80 of all points -> 0.25 bonus
  - Otherwise : `bonus_grade = 0.25 * min(1, your_bonus / (0.8*max_bonus)).`

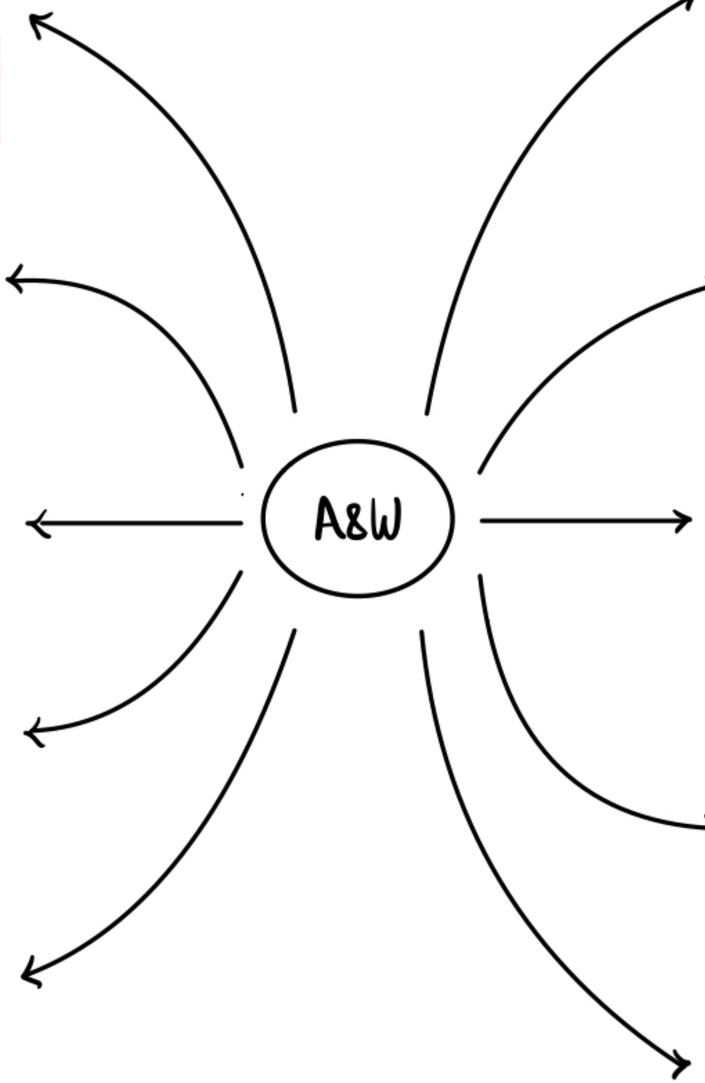
- Final Grade Calculation

```
final_grade = round(exam_grade + bonus_grade)
```

Website Introduction

www.nilozer.com

A&W Overview



Connectivity

- Articulation Points
- Menger's Theorem
- Bridges
- Block-Decomposition

Cycles

- Closed Eulerian Walk
- TSP
- Hamiltonian Cycle

Matchings

- Definition
- Hall's Theorem
- Algorithms

Colorings

- Definition
- Brooks's Theorem
- Algorithm

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
  - Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow-Mincut
- Ford-Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

A&W Standpoint at ETH CS

- Algorithms Part
  - A&D (1. Semester)
  - APC (Algorithms , Probability and Computing) (5. Semester)

- Probability Part
  - WuS (Wahrscheinlichkeit und Statistik) (4. Semester)

Exam

Exam

Quiz navigation

Algorithmen und Wahrscheinlichkeiten Klausur

i

Formelsammlung

i

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

Schriftliche Aufgaben - Part 5

i i i

Java Documentation

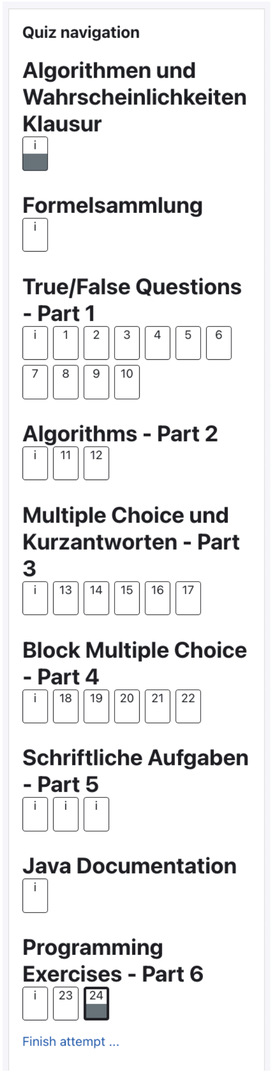
i

Programming Exercises - Part 6

i 23 24

Finish attempt ...

Exam



6 Parts

First 4 parts : each 10 points (similar to minitest)

Part 5 : written tasks, 20 points in total (similar to theory exercises)

Part 6 : 2 programming tasks, each 10 points (similar to CodeEx)

Exam

Quiz navigation

Algorithmen und Wahrscheinlichkeiten Klausur

i

Formelsammlung

i

Moodle

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

Written Theory

Schriftliche Aufgaben - Part 5

i i i

Programming

Java Documentation

i

Programming Exercises - Part 6

i 23 24

Finish attempt ...

Mock Exam 2022

Moodle

~1 points

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

Ein Matching, für das es keinen augmentierenden Pfad gibt, ist inklusionsmaximal.

Select one:

- True
- False

Drei Ereignisse A, B, C heissen unabhängig genau dann wenn $\Pr[A \cap B \cap C] = \Pr[A] \cdot \Pr[B] \cdot \Pr[C]$.

Select one:

- True
- False

Mock Exam 2022

~5 points

Moodle

True/False Questions - Part 1

i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10

Algorithms - Part 2

i | 11 | 12

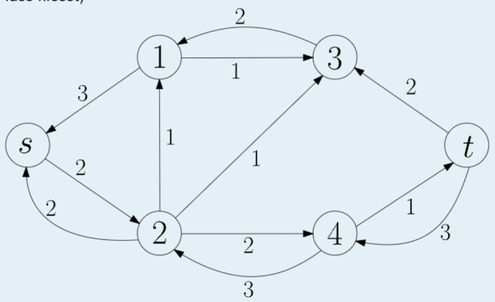
Multiple Choice und Kurzantworten - Part 3

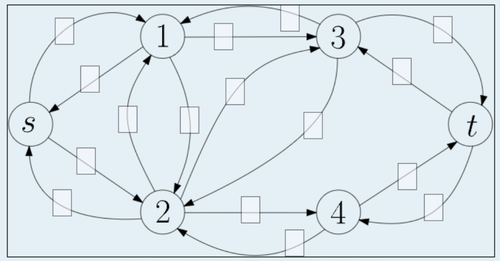
i | 13 | 14 | 15 | 16 | 17

Block Multiple Choice - Part 4

i | 18 | 19 | 20 | 21 | 22

Sei N ein Netzwerk ohne entgegengesetzte Kanten. Betrachten Sie das abgebildete Restnetzwerk R_f. Berechnen Sie den zugehörigen Fluss f und ziehen Sie die Flusswerte auf die entsprechenden Kanten (verwenden Sie die 0 für Kanten, über die kein Fluss fliesst)





1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0

Mock Exam 2022

~5 points

Moodle

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

function Metric_TSP_Approximation(G):

1. Finde ein/e (gewichts-)minimale/n/s [ ] × T in G.
2. Sei W die Menge von Knoten in T deren Grad [ ] ×
3. Finde ein/e (gewichtsminimale/n/s [ ] × M von W.
4. Finde ein/e [ ] × S im (Multi-)Graph

- $M \cup G$
- $G \setminus M$
- $M \cup T$
- $(G \setminus T) \cup M$

Mock Exam 2022

~2 points

Moodle

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

Sei $\Omega = \{-3, -2, 0, 2, 3\}$ ein Laplaceraum und sei ω ein (zufälliges) Elementarereignis in Ω. Berechnen Sie $\mathrm{E}[|\omega|]$.

Answer: ✕

Max wirft 10 faire Münzen. Leider hat er vergessen vorher das Fenster zu schliessen und jede seiner Münzen wird mit Wahrscheinlichkeit p von einer Elster gestohlen (unabhängig von den anderen Münzen).

Was ist die Wahrscheinlichkeit, dass Max wenigstens eine Münze, die Zahl zeigt, behält?

- ✓ $1 - (1 - (1 - p)/2)^{10}$
- ✕ $1 - p^{10}$
- ✕ $5 \cdot (1 - p)$
- ✕ $1 - (1 - p)^{10}/2^{10}$

Mock Exam 2022

Moodle

True/False Questions - Part 1

i 1 2 3 4 5 6 7 8 9 10

Algorithms - Part 2

i 11 12

Multiple Choice und Kurzantworten - Part 3

i 13 14 15 16 17

Block Multiple Choice - Part 4

i 18 19 20 21 22

~2 points

Welche der folgenden Probleme können -- mithilfe von Ideen aus dem Kurs -- als Fluss-Probleme modelliert und gelöst werden?

Richtig	Falsch	
☑	☐	Herausfinden, ob ein bipartiter Graph G ein perfektes Matching hat.
☐	☑	Den längsten Pfad in einem Graph G finden.
☑	☐	Herausfinden, ob ein Graph G 2-Kanten-zusammenhängend ist.
☑	☐	Herausfinden, ob ein Graph G 2-Knoten-zusammenhängend ist.

Seien A, B, C unabhängige Ereignisse. Welche der folgenden Gleichungen sind immer wahr?

Richtig	Falsch	
☑	☐	$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$
☐	☑	$\Pr[A] + \Pr[B] \leq \Pr[A \cup B]$
☑	☐	$\Pr[A \mid B \cap C] = \Pr[A \mid B \cup C]$
☐	☑	$\Pr[(A \cup B) \cap C] = (\Pr[A] + \Pr[B]) \cdot \Pr[C]$

Mock Exam 2022

Written Theory

Schriftliche Aufgaben - Part 5

i i i

Zeigen/Widerlegen Sie folgende Aussagen

a) Sei $G = (A \uplus B, E)$ ein regul"arer bipartiter Graph mit $E \neq \emptyset$. Dann ist $|A| = |B|$.

b) Seien X und Y unabh"angige Zufallsvariablen. Dann gilt $\mathbb{E}[\max(X, Y)] = \max(\mathbb{E}[X], \mathbb{E}[Y])$

c) Sei v ein Knoten, der inzident zu mindestens zwei Brücken ist. Dann ist u ein Artikulationsknoten.

jeweils 4 Punkte

on paper

Mock Exam 2022

Programming

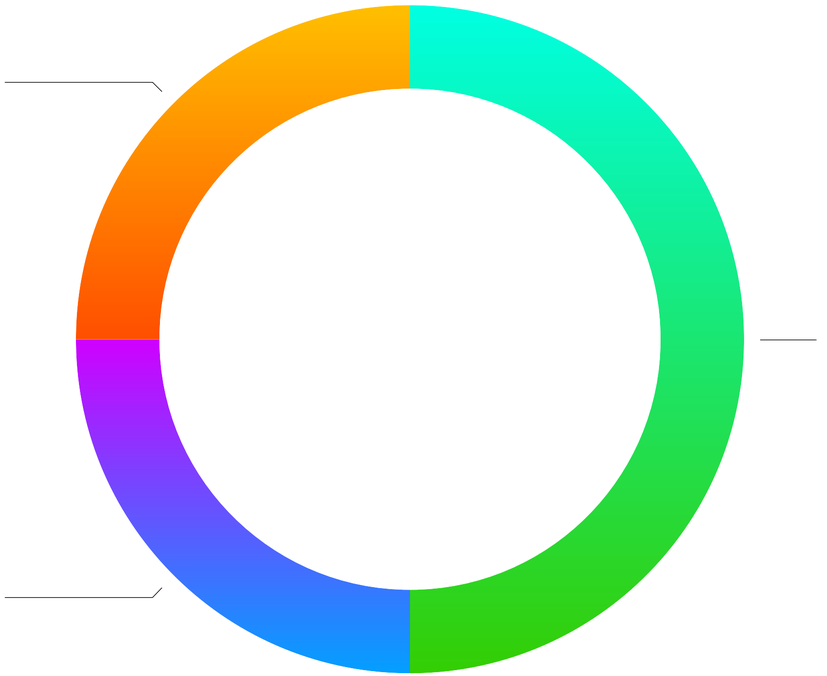
- One probability task
- One flow task

~10 points each

Point Distribution

based on mock exam

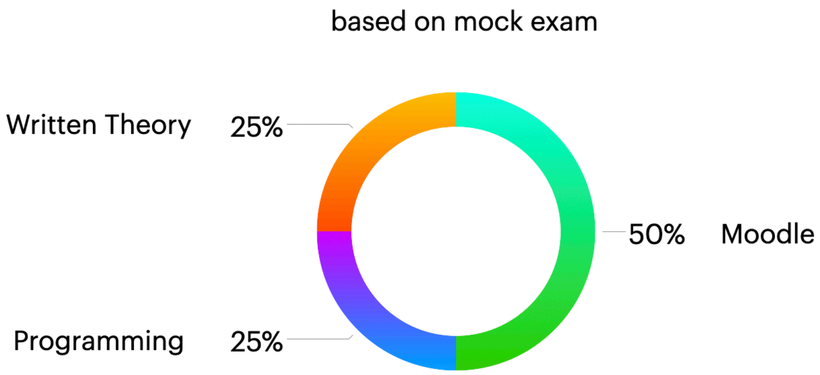
Written Theory



Moodle

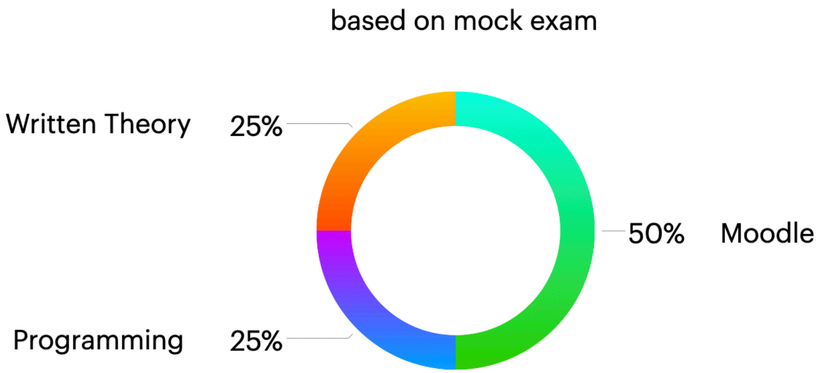
Programming

Point Distribution + Weekly Exercises



W1		Warm up exercise	
W2	Mini Quiz	Theory Exercise	Programming Exercise
W3		Peer Grading Exercise	Programming Exercise
W4	Mini Quiz	Theory Exercise	Programming Exercise
W5		Peer Grading Exercise	Programming Exercise
W6	Mini Quiz	Theory Exercise	Programming Exercise
W7		Peer Grading Exercise	Programming Exercise

Point Distribution + Weekly Exercises



	Moodle	Written + Moodle	Programming
W1		Warm up exercise	
W2	Mini Quiz	Theory Exercise	Programming Exercise
W3		Peer Grading Exercise	Programming Exercise
W4	Mini Quiz	Theory Exercise	Programming Exercise
W5		Peer Grading Exercise	Programming Exercise
W6	Mini Quiz	Theory Exercise	Programming Exercise
W7		Peer Grading Exercise	Programming Exercise

I got you !

How to study for A&W

During Semester

- Attend all lectures !
- Skript ! Some recap parts from A&D in the beginning
- Always come to the exercise session. Even if you fall back !
- Try to solve all exercises (of all types) Coding weekly !
- Ask questions ! exercise session , breaks, WhatsApp group, email , Moodle forum
- Summaries, Recaps
- Feedback Feedback pools by me or contacting me directly

Get to know me

Get to know you

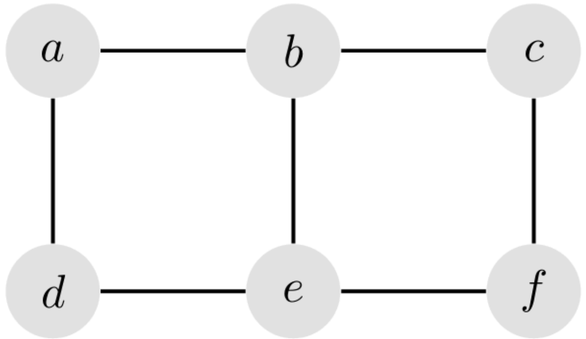
Join the whatsapp group !

Let’s take a break

Warm up Exercise Sheet

Aufgabe 1 – *Pfade, Wege, Kreise*

Betrachten Sie folgenden Graphen $G = (V, E)$.



1. Welche Pfade der Länge 4 (d.h. mit 4 Kanten) gibt es von a nach e?
2. Welche Wege der Länge 4 (d.h. mit 4 Kanten) gibt es von a nach e?
3. Welche Kreise gibt es in G?
4. Wie viele Zykeln gibt es in G?

Recap

Walk vs Path

walk

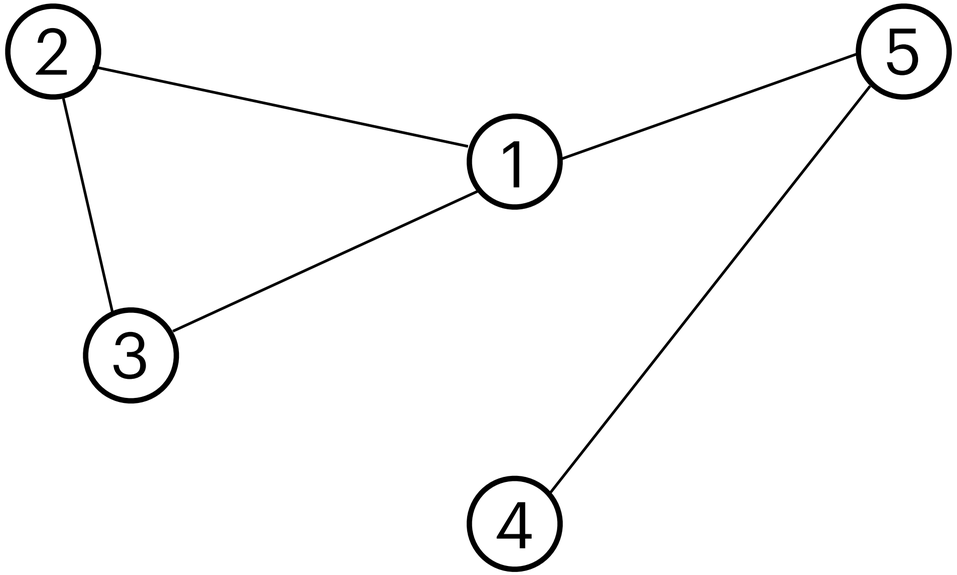
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ (with $v_i \in V$ for all i) is a **walk** (german "Weg") if $\{v_i, v_{i+1}\}$ is an edge for each $0 \leq i \leq k-1$. We say that v_0 and v_k are the **endpoints** (german "Startknoten" and "Endknoten") of the walk. The **length** of the walk $(v_0, v_1, \ldots, v_k)$ is k.

path

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **path** (german "Pfad") if it is a walk and all vertices are distinct (i.e., $v_i \neq v_j$ for $0 \leq i < j \leq k$).

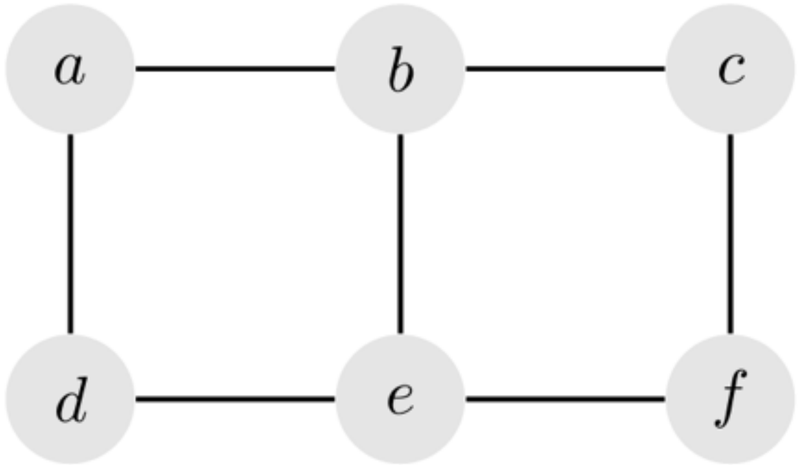
Is it a walk? Is it a path?

(5, 1, 3, 2, 1)	✅	❌
(5, 1, 3)	✅	✅



Warm up Exercise Sheet

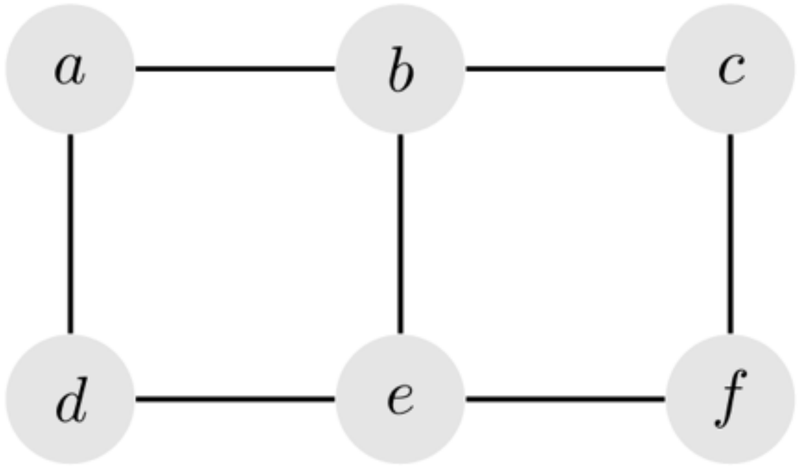
Exercise 1 : Paths, Walks, Circles



Paths of length 4 (i.e. with 4 edges) from a to e ?

Warm up Exercise Sheet

Exercise 1 : Paths, Walks, Circles

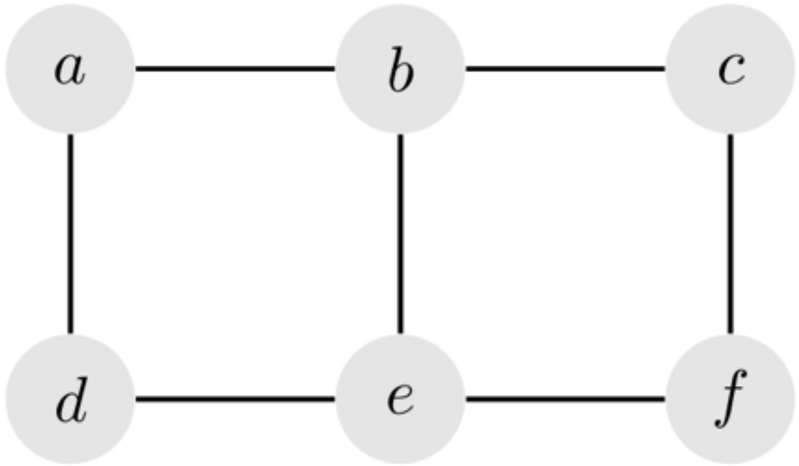


Paths of length 4 (i.e. with 4 edges) from a to e ?

⟨a,b,c,f,e⟩

Warm up Exercise Sheet

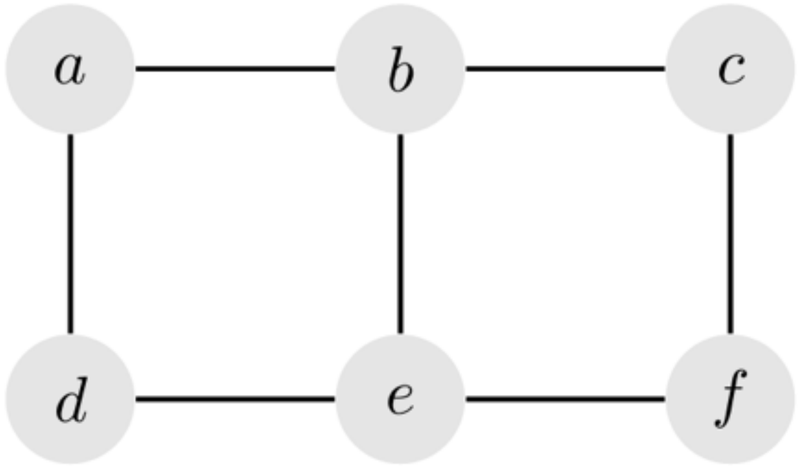
Exercise 1 : Paths, Walks, Circles



Walks of length 4 (i.e. with 4 edges) from a to e ?

Warm up Exercise Sheet

Exercise 1 : Paths, Walks, Circles



Walks of length 4 (i.e. with 4 edges) from a to e ?

⟨a,b,c,f,e⟩, ⟨a,b,c,b,e⟩, ⟨a,b,e,d,e⟩, ⟨a,b,e,f,e⟩, ⟨a,b,e,b,e⟩, ⟨a,b,a,b,e⟩, ⟨a, b, a,d,e⟩, ⟨a,d,a,d,e⟩, ⟨a,d,a,b,e⟩, ⟨a,d,e,d,e⟩, ⟨a,d,e,b,e⟩, ⟨a,d,e,f,e⟩

Recap

Closed Walk vs Cycle

Closed walk

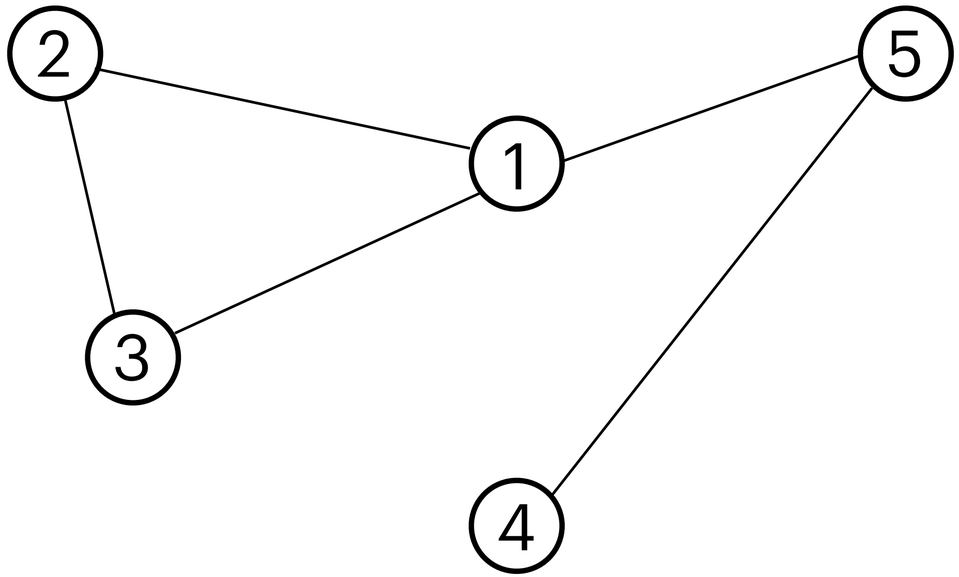
- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

Cycle

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.

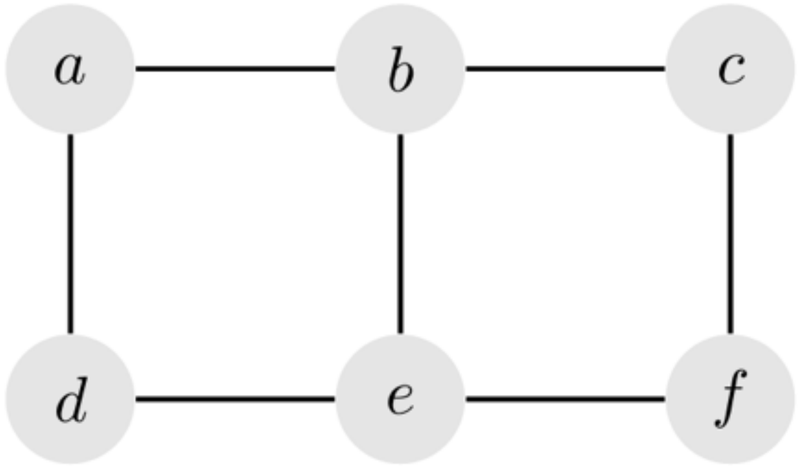
Is it a closed walk? Is it a cycle?

(5, 1, 3, 1, 5)	✅	❌
(1, 3, 2, 1)	✅	✅



Warm up Exercise Sheet

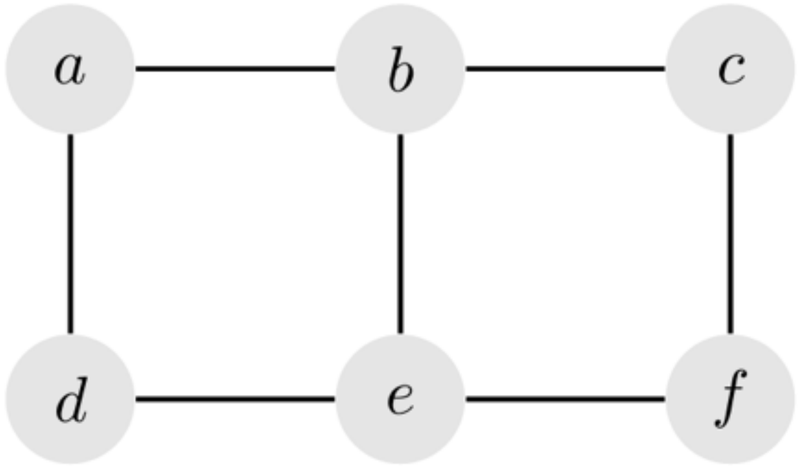
Exercise 1 : Paths, Walks, Circles



Cycles in G ?

Warm up Exercise Sheet

Exercise 1 : Paths, Walks, Circles



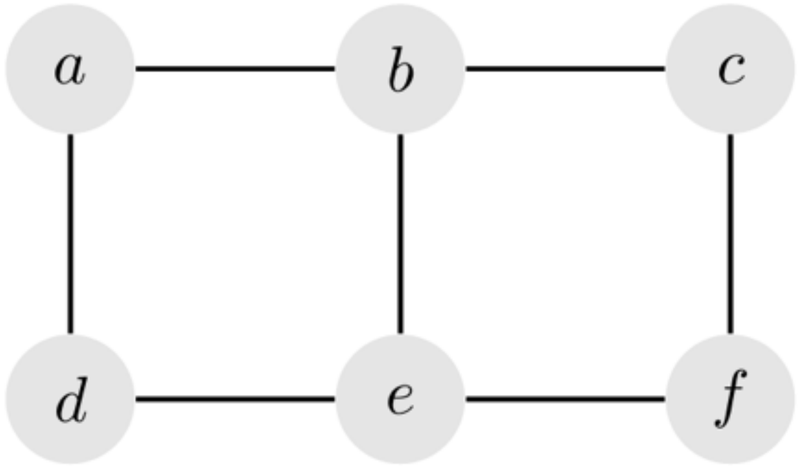
Cycles in G ?

⟨a,b,d,e,a⟩, ⟨b,c,f,e,b⟩ and ⟨a,b,c,f,e,d,a⟩

+ changing the starting points !

Warm up Exercise Sheet

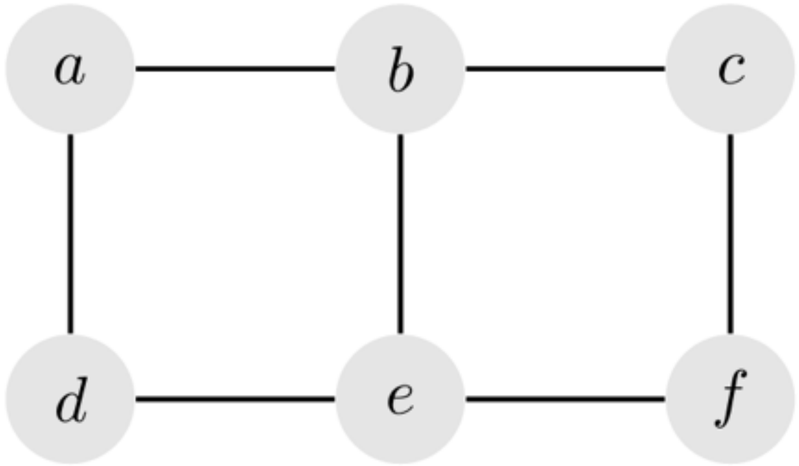
Exercise 1 : Paths, Walks, Circles



Closed Walks in G ?

Warm up Exercise Sheet

Exercise 1 : Paths, Walks, Circles



Closed Walks in G ?

Infinitely many

Warm up Exercise Sheet

Exercise 2 : Asymptotic Growth

Aufgabe 2 – *Asymptotisches Wachstum.*

(a) (Leicht.) Sortieren Sie die folgenden Funktionen asymptotisch, d.h. entsprechend der O-Notation. Dabei bezeichnet $\log n$ den Logarithmus zur Basis 2, und $\ln n$ den natürlichen Logarithmus. In welchen Fällen haben Sie asymptotische Gleichheit $\Theta(.)$?

$$n, \quad 0.01n^2, \quad e^n, \quad \log n, \quad 2^{32}, \quad 2^n, \quad n + \sqrt{n},$$

(b) (Schwerer.) Sortieren Sie zusätzlich die folgenden Funktionen in Ihre Abfolge ein.

$$\ln n, \quad \frac{n}{\log n}, \quad e^{\sqrt{\log n}}, \quad \log(n^2), \quad n^{1/4}, \quad n!$$

Recap

Mini cheat-sheet

Theorem 1 (Theorem 1.1 from the script). *Let N be an infinite subset of $\mathbb{N}$ and $f : N \to \mathbb{R}^+$ and $g : N \to \mathbb{R}^+$.*

- *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *then* $f \leq O(g)$, *but* $f \neq \Theta(g)$.
- *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, *then* $f = \Theta(g)$.
- *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *then* $f \geq \Omega(g)$, *but* $f \neq \Theta(g)$.

$\lim_{n\to\infty}$: $1 < \log(\log(n)) < \log(n) < \sqrt{n} < n < n\cdot\log(n) < n\cdot\sqrt{n} < n^2 < 2^n < n! < n^n$

$n^x < x^n$ (x being fixed)

Sums

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots n^2 = \frac{n(n+1)(2n+1)}{6}$$

Geometric series : $\sum_{k=0}^{n} q^k = \frac{q^{n+1}-1}{q-1}$

$$\sum_{k=0}^{3} 3^k = 3^0 + 3^1 + 3^2 + 3^3 = \frac{3^4-1}{3-1} = 40$$

Factorial

$$\frac{n}{2}^{\frac{n}{2}} \leq n! \leq n^n$$

From Exercise Sheet 1 :

$\sum_{i=1}^{n} i^k \leq n^{k+1}$

$\sum_{i=1}^{n} i^k \geq \frac{1}{2^{k+1}} \cdot n^{k+1}$

$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

Warm up Exercise Sheet

Exercise 3 : Induction

Aufgabe 3 – *Induktion*

(a) Zeigen Sie für alle $n \in \mathbb{N}$:

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \ldots + \frac{1}{n \cdot (n+1)} = \frac{n}{n+1}.$$

(b) Zeigen Sie die folgende *Ungleichung von Bernoulli*: Für alle natürlichen Zahlen $n \geq 1$ und alle $h \in \mathbb{R}$ mit $h \geq -1$ gilt:

$$1 + nh \leq (1+h)^n.$$

Warm up Exercise Sheet

Exercise 4 : A General Feature of Graphs

Aufgabe 4 – *Eine generelle Eigenschaft von Graphen*

Zeigen Sie, dass jeder Graph G mit $n \geq 2$ Knoten zwei Knoten $v \neq w$ enthält, sodass $deg(v) = deg(w)$.

Hinweis: Für ein gegebenes n, was ist der grösstmögliche Grad den ein Knoten haben kann?

Warm up Exercise Sheet

Exercise 5 : Algorithms

Aufgabe 5 – *Algorithmus*

Beschreiben Sie einen Algorithmus der das folgende Problem löst: Gegeben ist die Eingabe bestehend aus einen Graphen $G = (V, E)$ mit n Knoten (gehen Sie davon aus, dass der Graph als Adjazenzliste gegeben ist). Ihr Algorithmus soll "Ja" ausgeben, falls G ein Baum ist und "Nein" andernfalls.

Wie immer wenn Sie einen Algorithmus beschreiben gehöhrt zu einer vollständigen Lösung: eine klare Beschreibung des Algorithmus, ein Korrektheitsbeweis und eine Laufzeitanalyse.

Hinweis: Für diese Aufgabe dürfen Sie das Statement aus Aufgabe 6 ohne Beweis verwenden.

A&W

Exercise Session 2

Connectivity

Nil Ozer

Outline

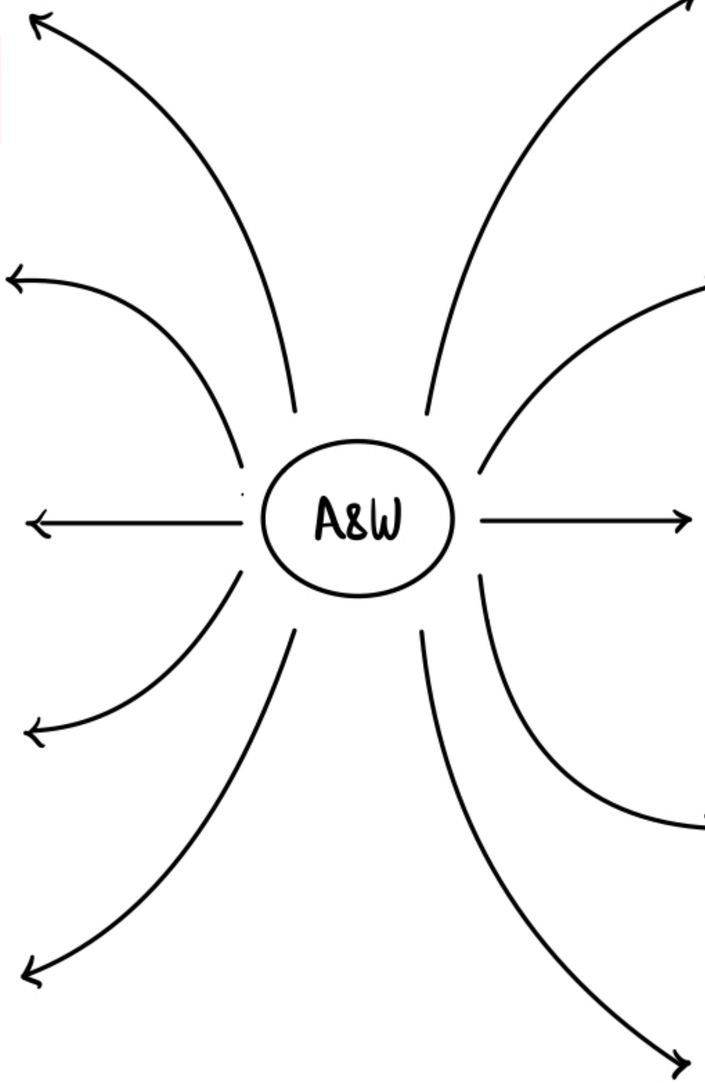
- Minitest

- Connectivity
- Articulation Points and Bridges

- (Cycles)

Minitest

A&W Overview



Connectivity

- Articulation Points
- Menger's Theorem
- Bridges
- Block-Decomposition

Cycles

- Closed Eulerian Walk
- TSP
- Hamiltonian Cycle

Matchings

- Definition
- Hall's Theorem
- Algorithms

Colorings

- Definition
- Brooks's Theorem
- Algorithm

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
  - Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow-Mincut
- Ford-Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

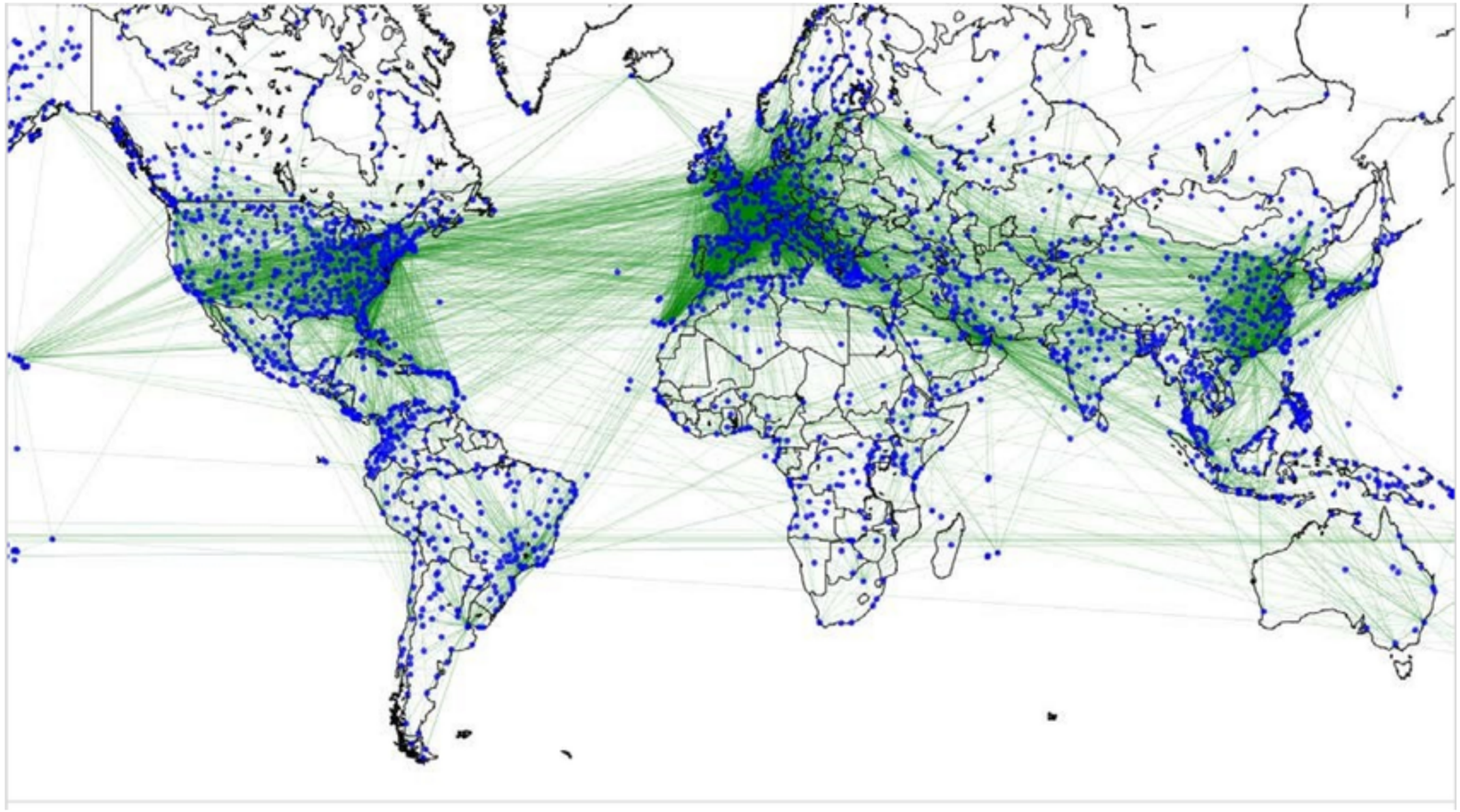
Connectivity

Connectivity

Intuition

measuring fault tolerance of a network !

How many connections can fail without cutting off the communication ?



Connectivity

Definitions

- A $G = (V, E)$ is connected if

  for $\forall u, v \in V, u \neq v$ there exists an u-v path

not connected

Connectivity

Definitions

- A $G = (V, E)$ is connected if

  for $\forall u, v \in V, u \neq v$ there exists an u-v path

connected

Connectivity

“Removing”

Remove the edge {a,b}

Connectivity

“Removing”

Remove the edge {a,b}

Connectivity

“Removing”

Remove the edge {a,b}

Connectivity

“Removing”

Remove the vertex b

Connectivity

“Removing”

Remove the vertex b

Connectivity

“Removing”

Remove the vertex b

Connectivity

“Removing”

Remove the vertex b

Connectivity

Definitions

- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"

How to find the k-edge connectivity?

Start from 1-edge connected, increase one by one !

(Every connected G is 1-edge connected)

Connectivity

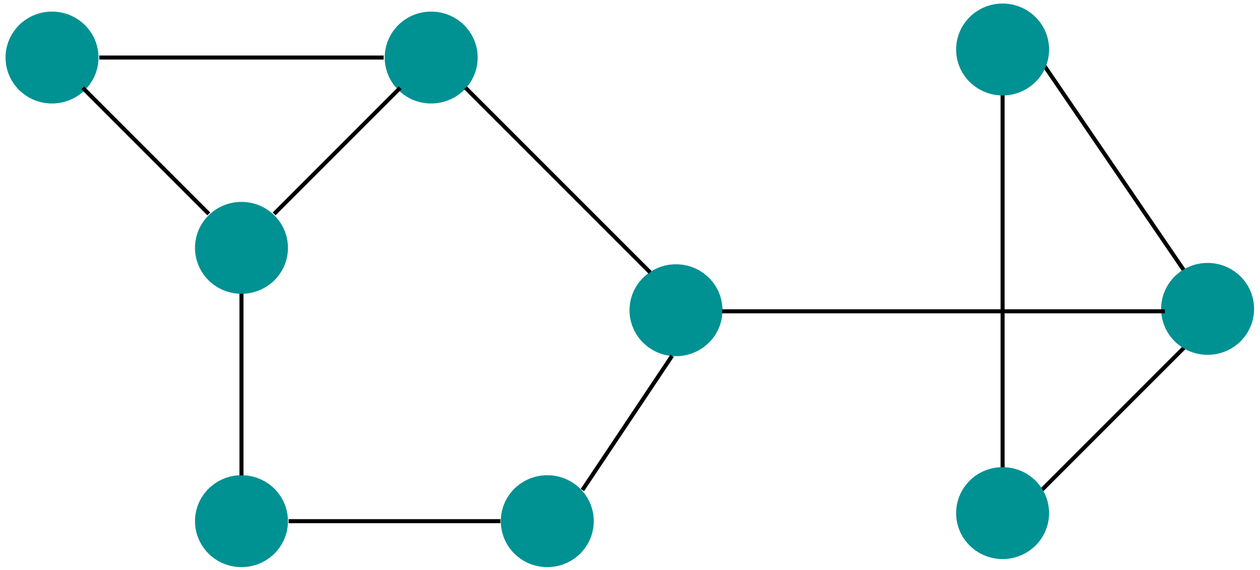
Example

- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"



Connectivity

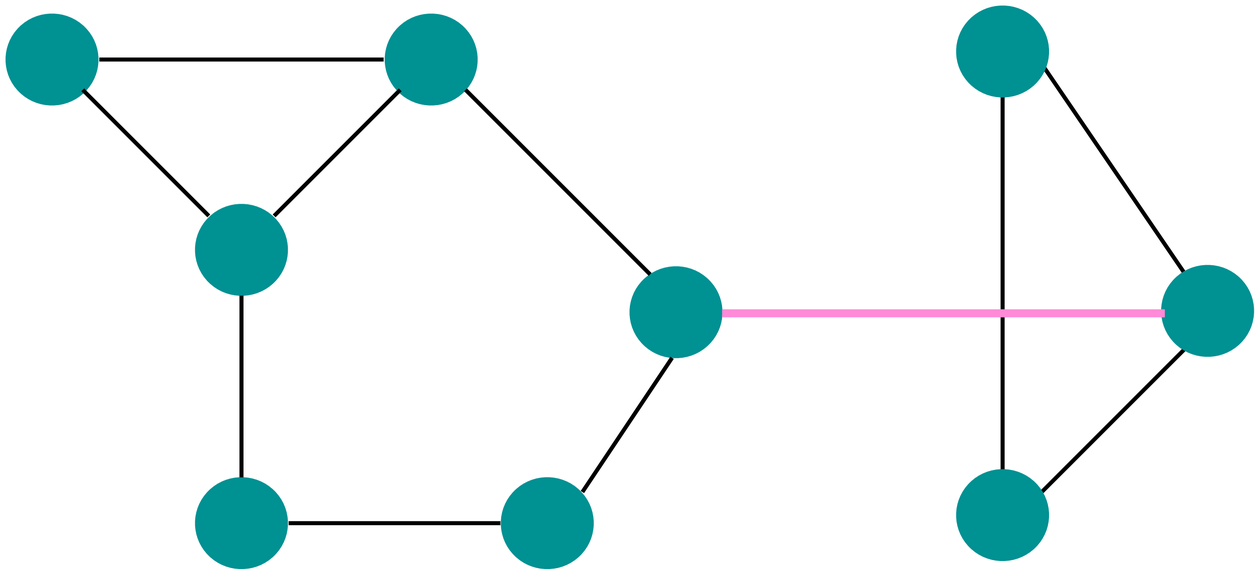
Example

- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"



1-edge connected

Connectivity

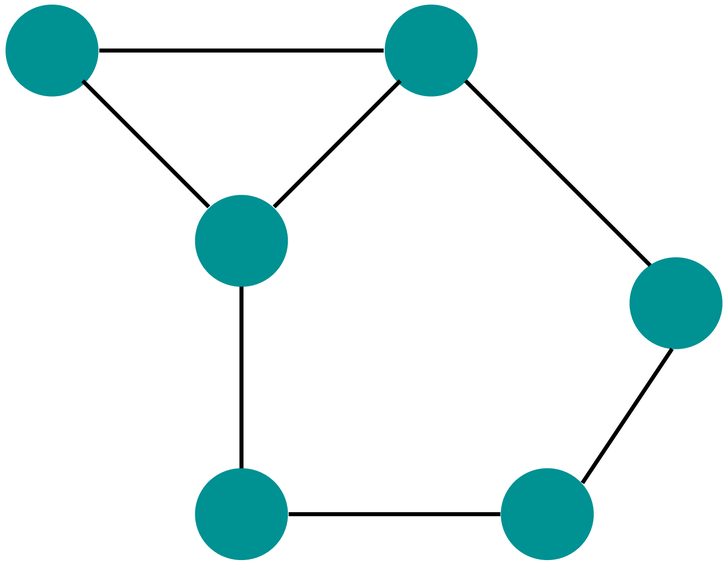
Example

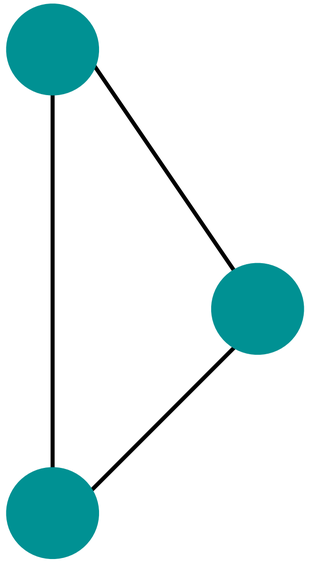
- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"





1-edge connected

becomes disconnected after removing 1 edge

Connectivity

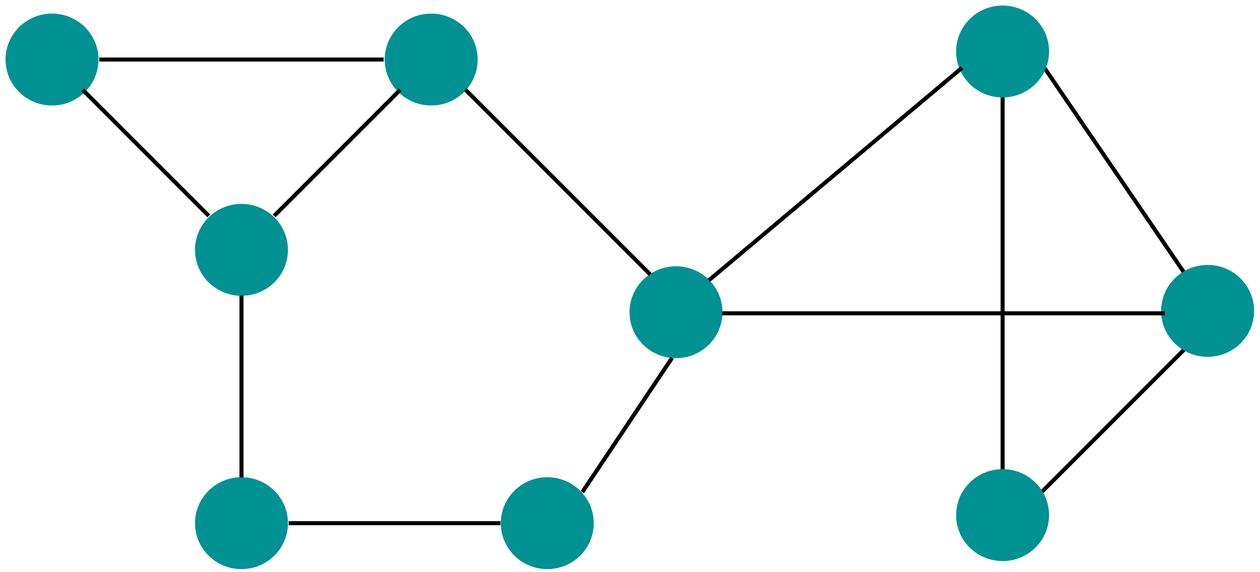
Example

- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"



Connectivity

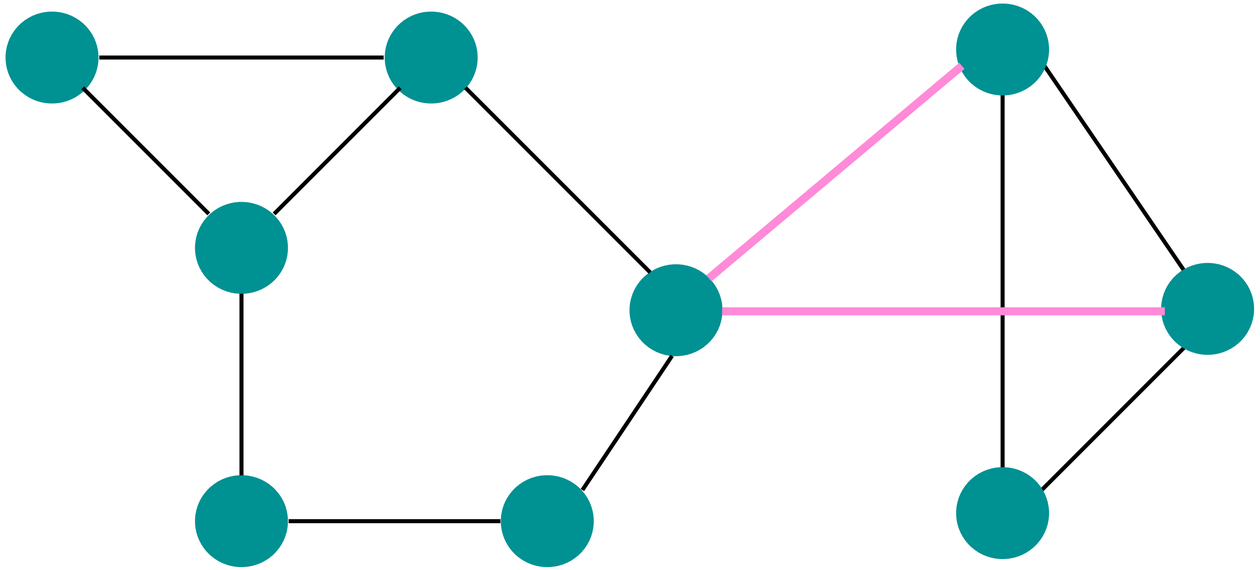
Example

- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"



2-edge connected

Connectivity

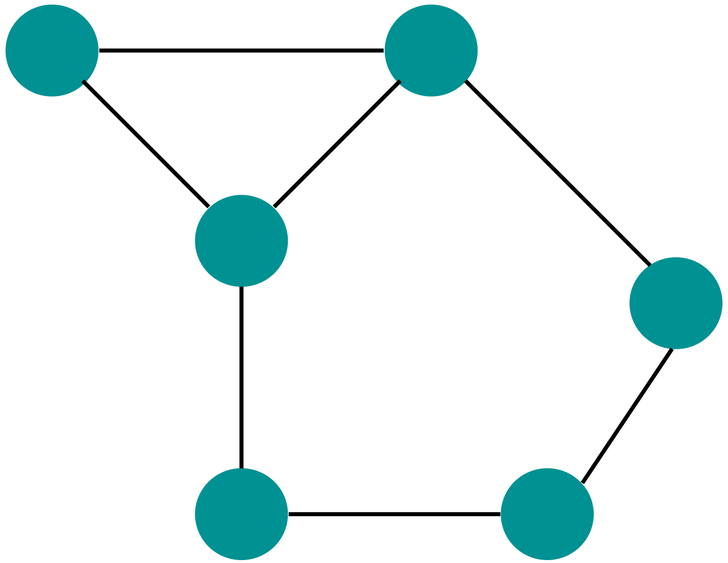
Example

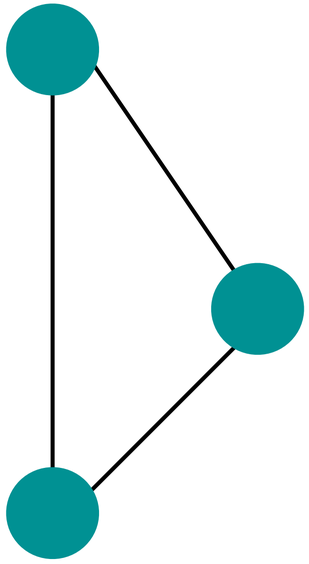
- A $G = (V, E)$ is k-edge connected if

  $\forall X \subseteq E$ with $|X| < k$, $G(V, E \setminus X)$ is connected

" The G remains connected whenever fever than k edges are removed"

" At least k edges must be removed to make the G disconnected"





2-edge connected

becomes disconnected after removing 2 edges

Connectivity

Definitions

- A $G = (V, E)$ is k - (vertex) connected if
  - $|V| \geq k + 1$
  - $\forall X \subseteq V$ with $|X| < k$, $G[V \backslash X]$ is connected

" The G remains connected whenever fever than k vertices are removed"

" At least k vertices must be removed to make the G disconnected"

minitest 8,9

Connectivity

Lemma

k-vertex connectivity $\leq$ k-edge connectivity $\leq$ minimum degree

Connectivity

Menger's Theorem

G is k-edge connected $\iff$ For $\forall u, v \in V, u \neq v$ there exists k edge-disjoint u-v paths

G is k-vertex connected $\iff$ For $\forall u, v \in V, u \neq v$ there exists k internally-vertex-disjoint u-v paths

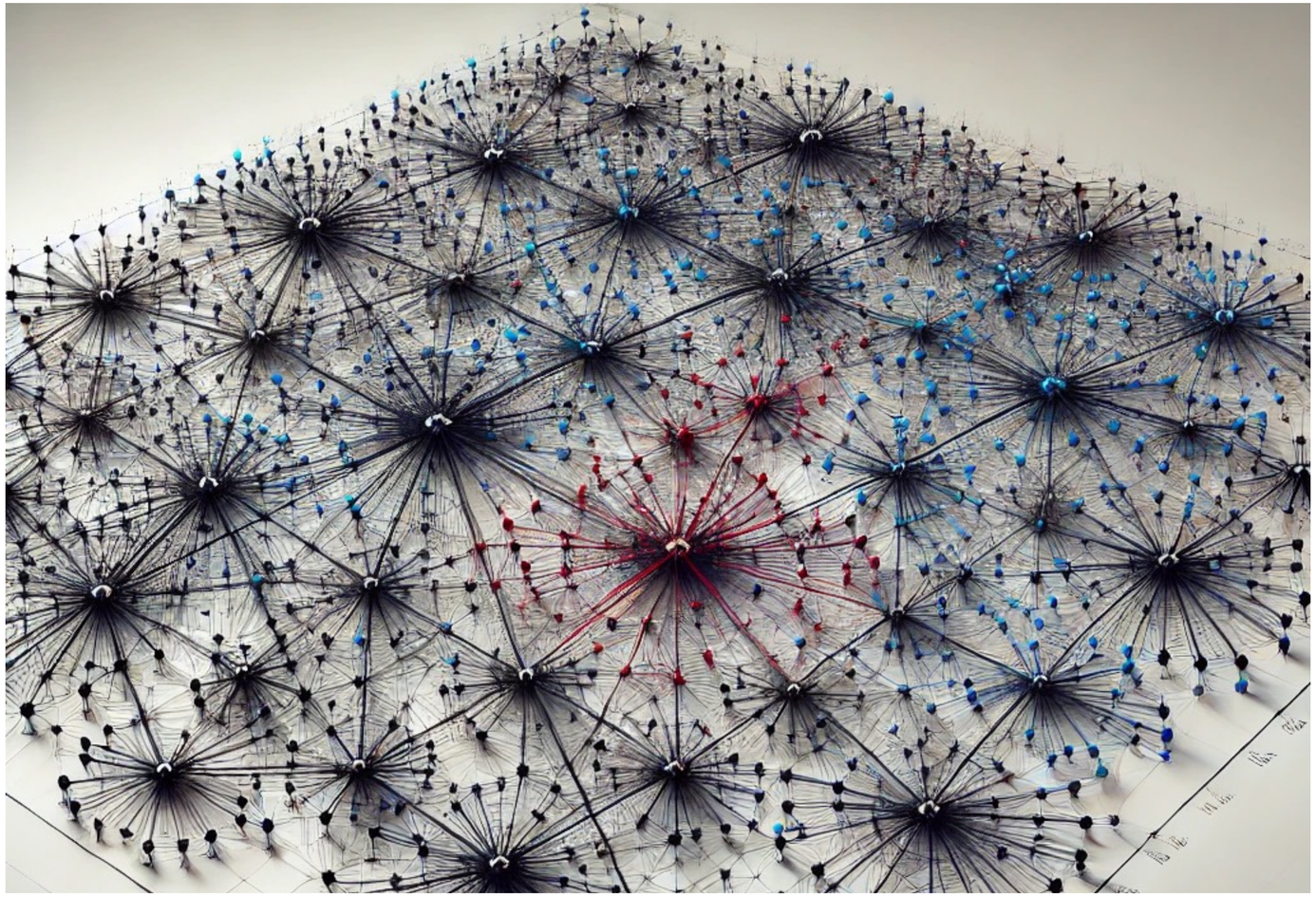
they share the starting and ending vertex

Let’s take a break

Articulation Points and Bridges

Intuition

single points whose failure would split the network into 2 or more components



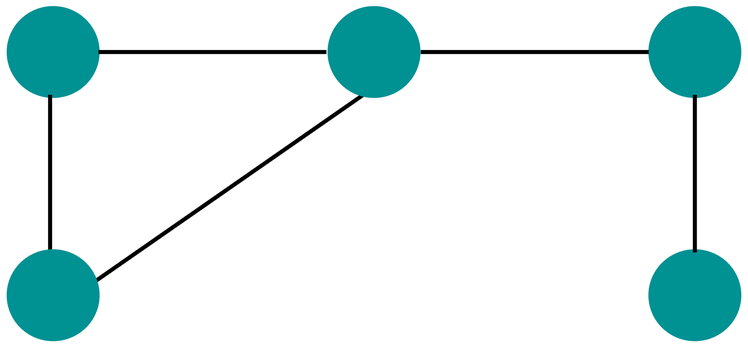
vulnerabilities in a network

Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $v \in V$ is an articulation point (cut vertex) iff $G[V \setminus \{v\}]$ is not connected

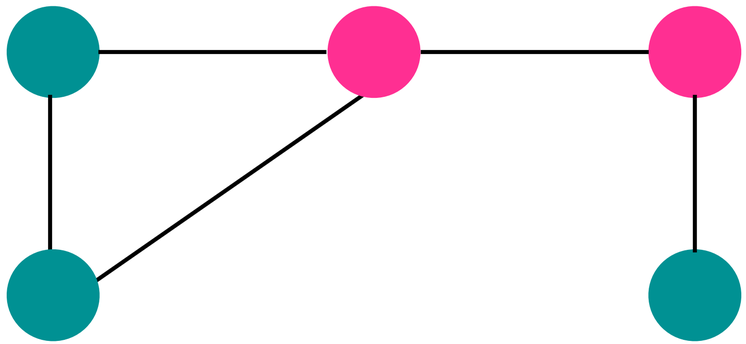


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $v \in V$ is an articulation point (cut vertex) iff $G[V \setminus \{v\}]$ is not connected



articulation points

Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

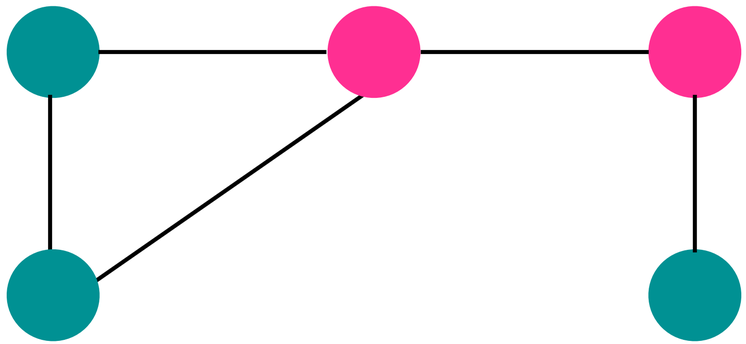
  A vertex $v \in V$ is an articulation point (cut vertex) iff $G[V \setminus \{v\}]$ is not connected

Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $v \in V$ is an articulation point (cut vertex) iff $G[V \setminus \{v\}]$ is not connected



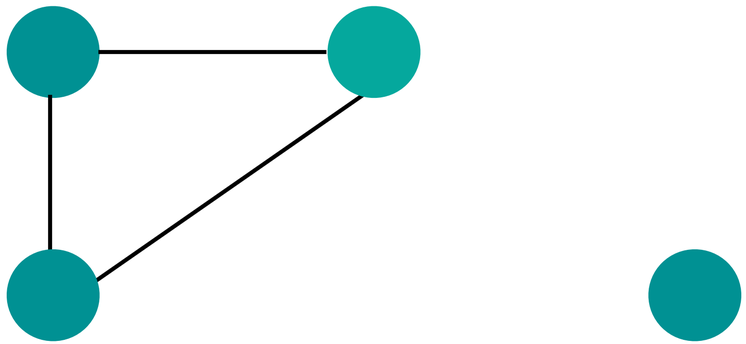
articulation points

Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $v \in V$ is an articulation point (cut vertex) iff $G[V \setminus \{v\}]$ is not connected

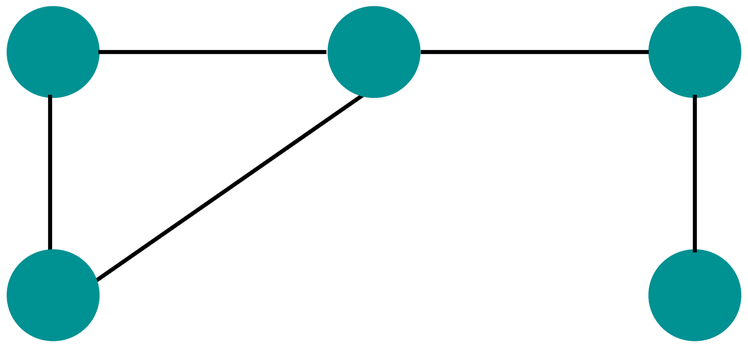


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  An edge $e \in E$ is a bridge (cut edge) iff $G - e$ is not connected

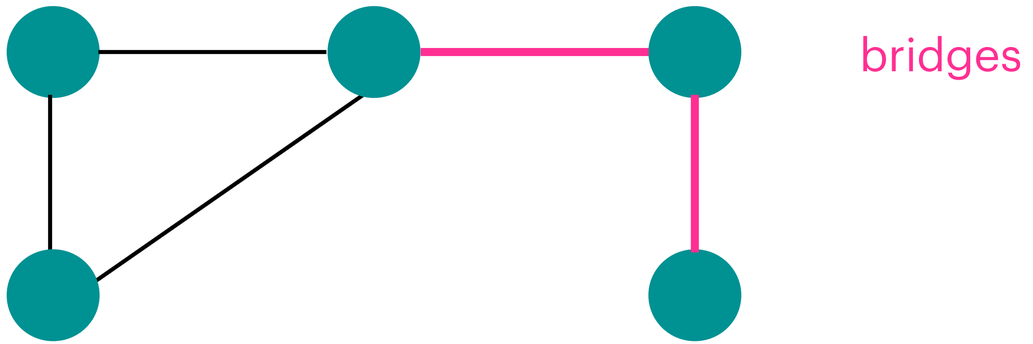


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $e \in E$ is a bridge (cut edge) iff $G - e$ is not connected

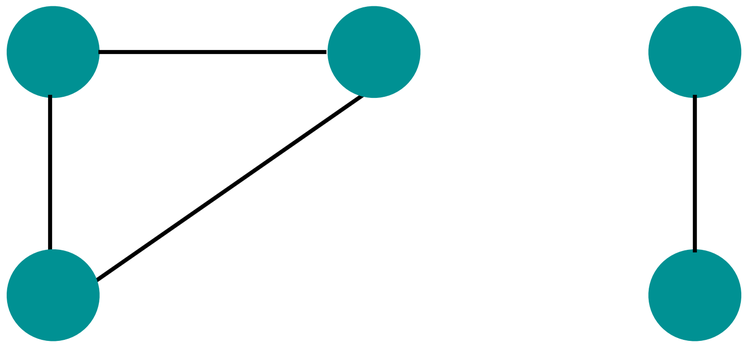


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $e \in E$ is a bridge (cut edge) iff $G - e$ is not connected

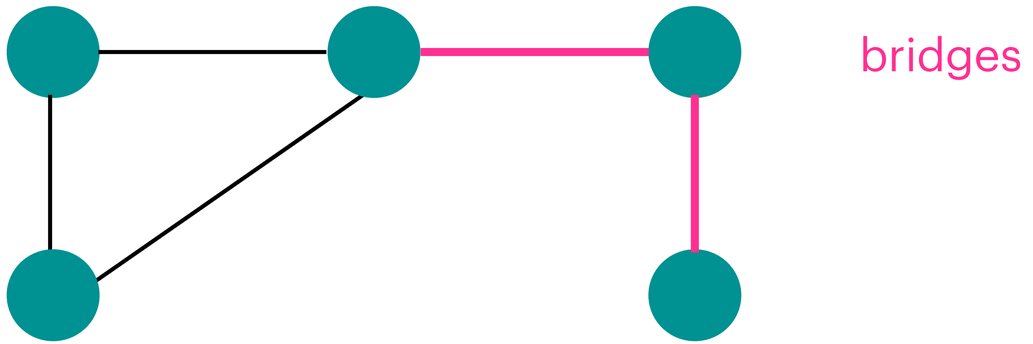


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $e \in E$ is a bridge (cut edge) iff $G - e$ is not connected

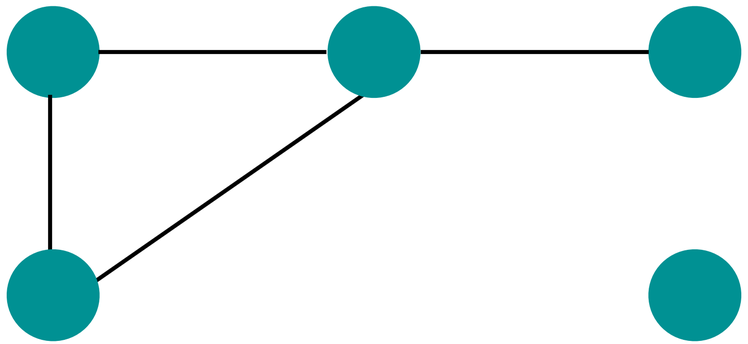


Articulation Points and Bridges

Definitions

- Let $G = (V, E)$ be connected.

  A vertex $e \in E$ is a bridge (cut edge) iff $G - e$ is not connected



Articulation Points and Bridges

Lemma

- Let $G = (V, E)$ be a connected graph.

$\{x, y\} \in E$ is a bridge $\Rightarrow$ deg(x) = 1 or x is an articulation point

minitest 2

Articulation Points and Bridges

Definition

- Let $G = (V, E)$ be a graph.

  The equivalence relation $\sim$ on E is defined as :

$$e \sim f \quad := \begin{cases} e = f & \text{or} \\ e \text{ and } f \text{ are on a common cycle} \end{cases}$$

Articulation Points and Bridges

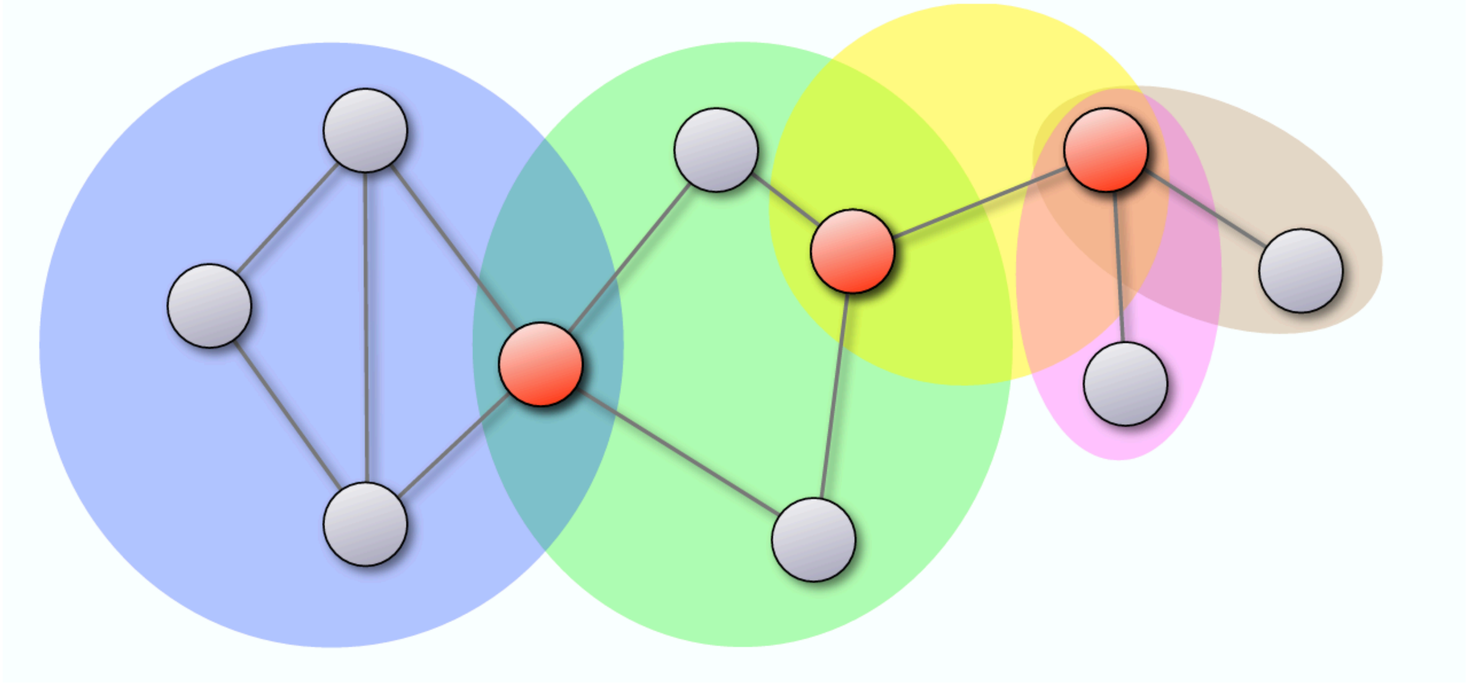
Definition

- Let $G = (V, E)$ be a graph.

  The equivalence relation $\sim$ on E is defined as :

$$e \sim f \quad := \begin{cases} e = f & \text{or} \\ e \text{ and } f \text{ are on a common cycle} \end{cases}$$

- The equivalence classes are named as Blocks



Lemma :

2 blocks always intersect at an articulation point.

Articulation point is the critical point that holds blocks together. If a graph has an articulation point, it serves as the **only connection** between two or more blocks.

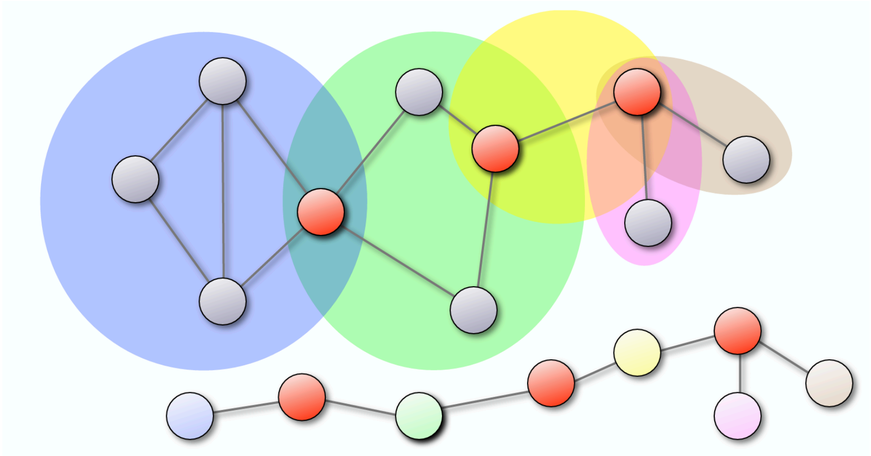
Articulation Points and Bridges

Definition

- Let $G = (V, E)$ be connected

  The Block-Graph of G is the bipartite Graph $T = \left(A \uplus B, E_T\right)$ with

  - A = {Articulation points of G}
  - B = {Block of G }
  - $\forall a \in A, b \in B : \{a, b\} \in E_T \iff a$ is incident to an edge in b



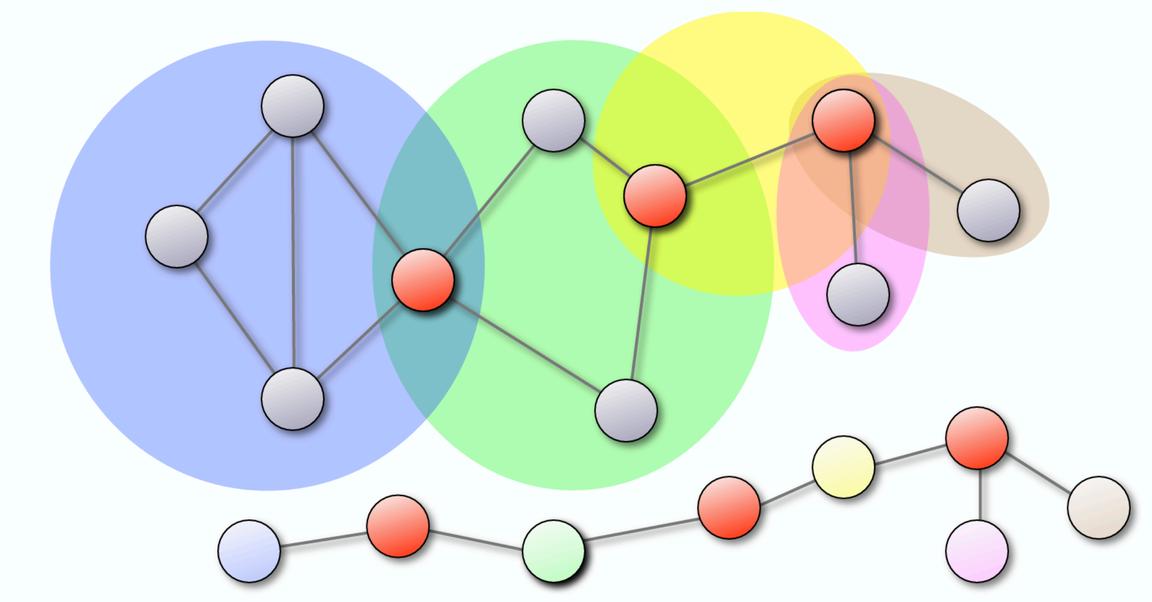
Articulation Points and Bridges

Lemma

- Let $G = (V, E)$ be connected

  The Block-Graph of G is the bipartite Graph $T = \left(A \uplus B, E_T\right)$ with

  - A = {Articulation points of G}
  - B = {Block of G }
  - $\forall a \in A, b \in B : \{a, b\} \in E_T \iff a$ is incident to an edge in b



If G is connected , then the Block-Graph of G is a tree

Articulation Points and Bridges

Finding articulation points

Queue:
First-in-first-out

Stack:
Last-in-first-out

BFS-VISIT-ITERATIVE(G, v)

```
1  Q ← ∅
2  Markiere v als aktiv
3  ENQUEUE(Q, v)
4  while Q ≠ ∅ do
5      w ← DEQUEUE(Q)
6      Markiere w als besucht
7      for each (w, x) ∈ E do
8          if x nicht aktiv und x noch nicht besucht then
9              Markiere x als aktiv
10             ENQUEUE(Q, x)
```

shortest paths

articulation points

DFS-VISIT-ITERATIVE(G, v)

```
1  S ← ∅
2  PUSH(S, v)
3  while S ≠ ∅ do
4      w ← POP(S)
5      if w noch nicht besucht then
6          Markiere w als besucht
7          for each (w, x) ∈ E in reverse order do
8              if x noch nicht besucht then
9                  PUSH(S, x)
```

Articulation Points and Bridges

Finding articulation points

DFS-VISIT-ITERATIVE(G, v)

```
1 S ← ∅
2 PUSH(S, v)
3 while S ≠ ∅ do
4     w ← POP(S)
5     if w noch nicht besucht then
6         Markiere w als besucht
7         for each (w, x) ∈ E in reverse order do
8             if x noch nicht besucht then
9                 PUSH(S, x)
```

+ Calculate low[v]

Articulation Points and Bridges

Finding articulation points

DFS-VISIT-ITERATIVE(G, v)

```
1  S ← ∅
2  PUSH(S, v)
3  while S ≠ ∅ do
4      w ← POP(S)
5      if w noch nicht besucht then
6          Markiere w als besucht
7          for each (w, x) ∈ E in reverse order do
8              if x noch nicht besucht then
9                  PUSH(S, x)
```

Calculate low[v]

\+

low[v] := the smallest dfs-number that one can reach from v with a directed path consisting of (any number of) tree edges and maximum one remaining edge .

Articulation Points and Bridges

Finding articulation points

DFS-VISIT-ITERATIVE(G, v)

```
1 S ← ∅
2 PUSH(S, v)
3 while S ≠ ∅ do
4     w ← POP(S)
5     if w noch nicht besucht then
6         Markiere w als besucht
7         for each (w, x) ∈ E in reverse order do
8             if x noch nicht besucht then
9                 PUSH(S, x)
```

Calculate low[v]

\+

low[v] := the smallest dfs-number that one can reach from v with a directed path consisting of (any number of) tree edges and maximum one remaining edge .

Articulation Points and Bridges

Finding articulation points

DFS-VISIT-ITERATIVE(G, v)

```
1 S ← ∅
2 PUSH(S, v)
3 while S ≠ ∅ do
4     w ← POP(S)
5     if w noch nicht besucht then
6         Markiere w als besucht
7         for each (w, x) ∈ E in reverse order do
8             if x noch nicht besucht then
9                 PUSH(S, x)
```

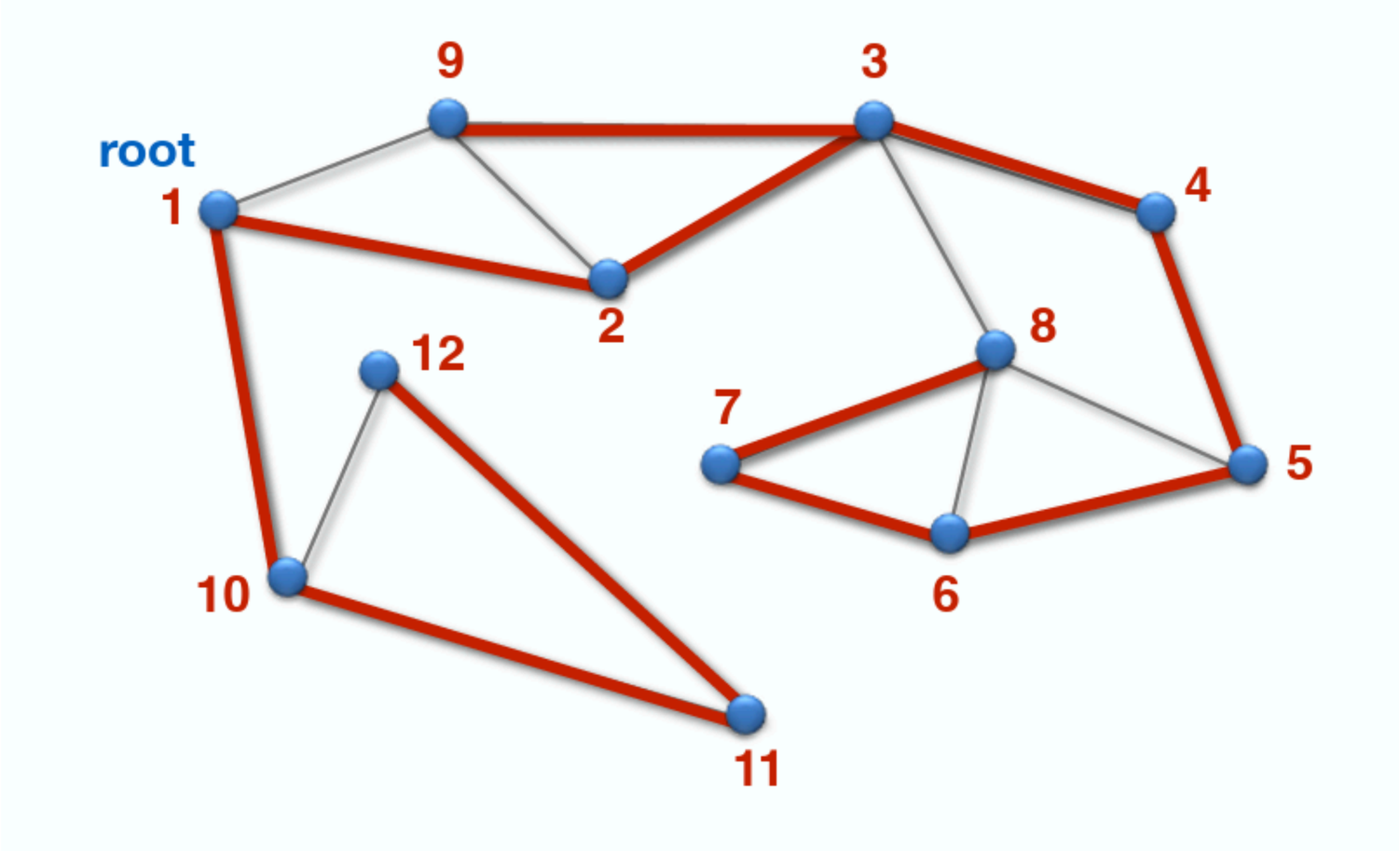
Calculate low[v]

\+

low[v] := the smallest dfs-number that one can reach from v with a directed path consisting of (any number of) tree edges and maximum one remaining edge .

Articulation Points and Bridges

Finding articulation points

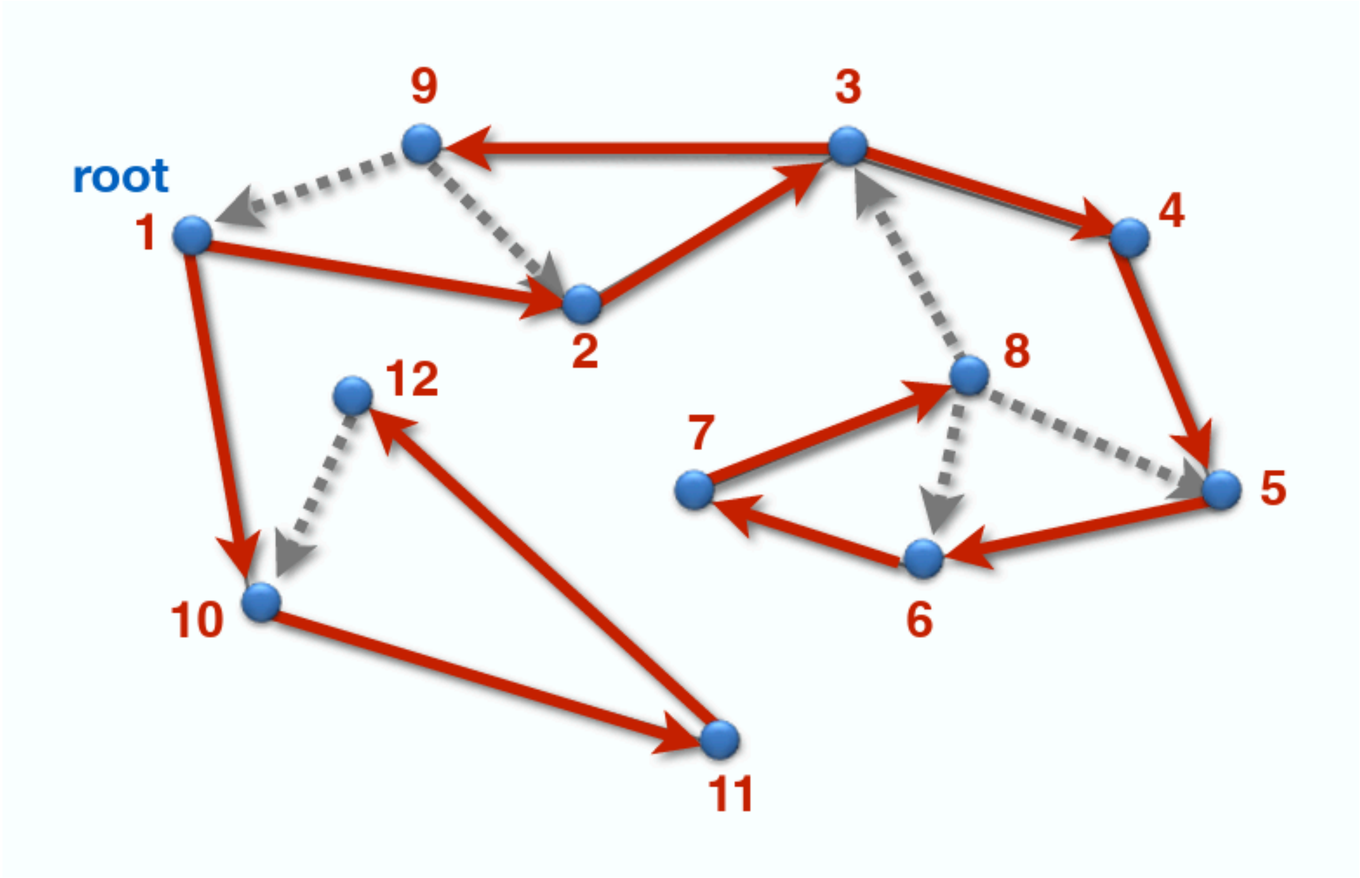


1) Find DFS numbers

- DFS from A&D
- pre-number from A&D is now the DFS number
- back edge/forward edge/ cross edge are now all remaining edges

Articulation Points and Bridges

Finding articulation points



tree edges

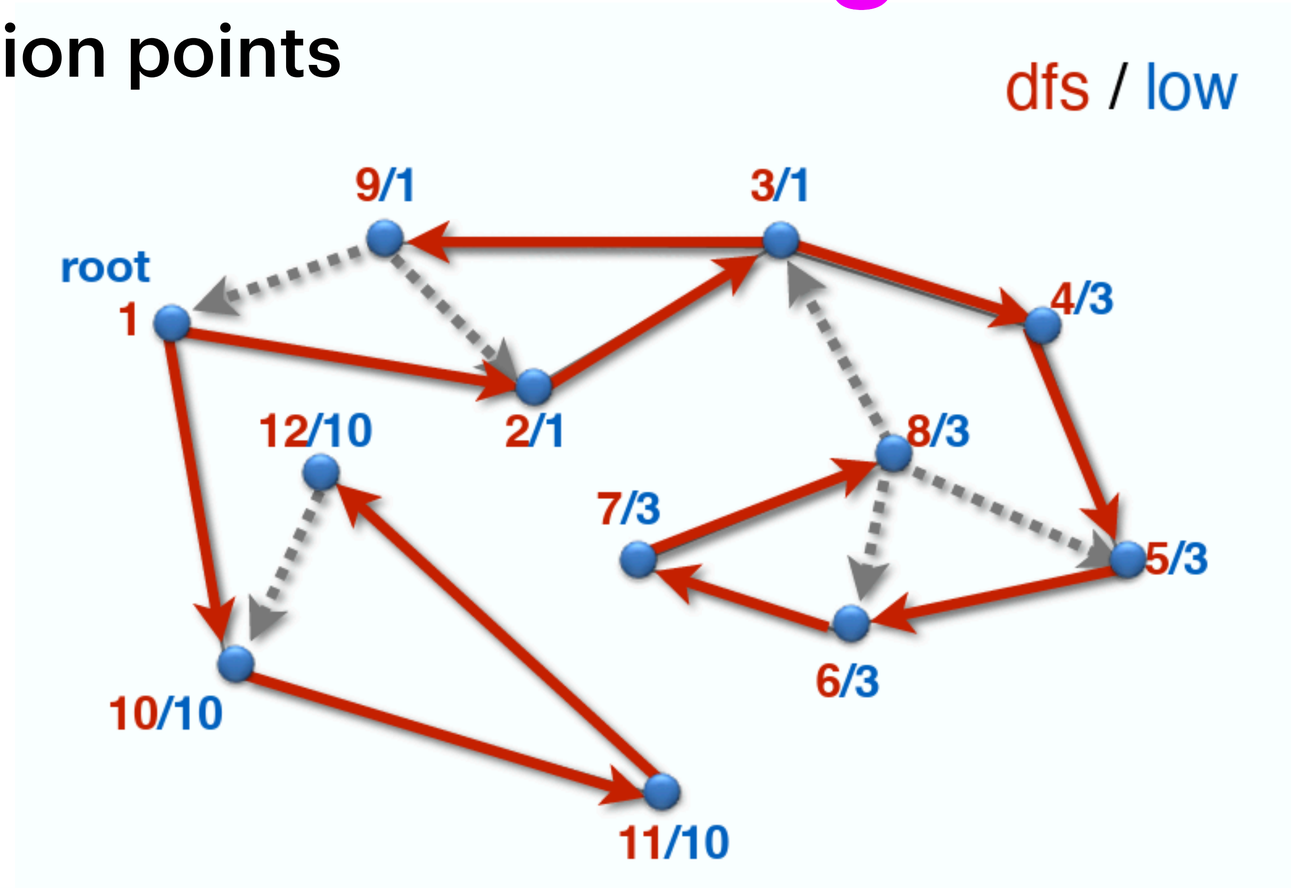
remaining edges

2) Orientation

- tree edges in the increasing dfs-number direction
- remaining edges in decreasing dfs-number direction

Articulation Points and Bridges

Finding articulation points

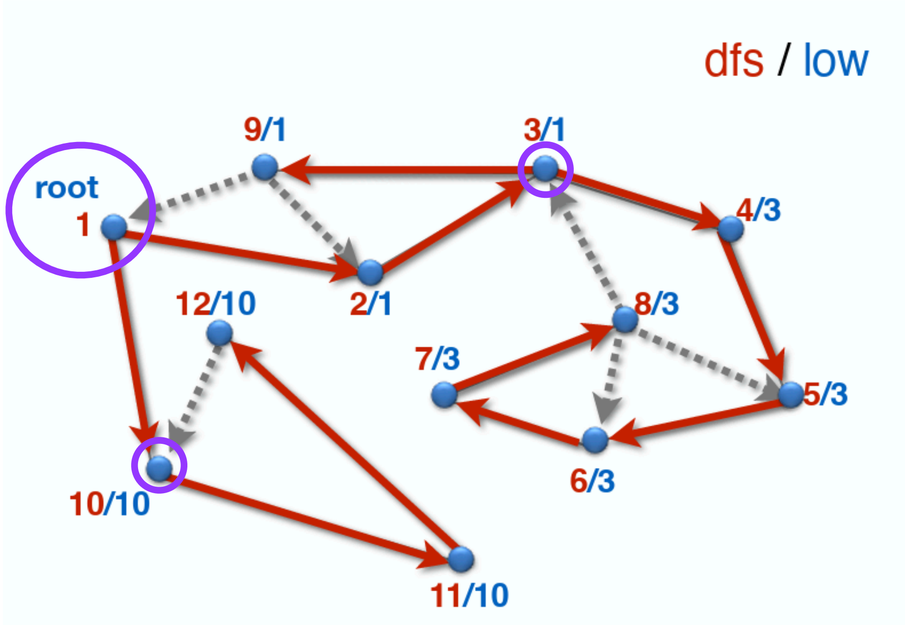


3) Calculate low[v]

$$\text{low}[v] = \min\left(\text{dfs}[v],\ \min_{(v,w)\in E}\begin{cases}\text{dfs}[w], & \text{if (v,w) is a remaining edge}\\ \text{low}[w], & \text{if (v,w) is a tree edge}\end{cases}\right)$$

Articulation Points and Bridges

Finding articulation points

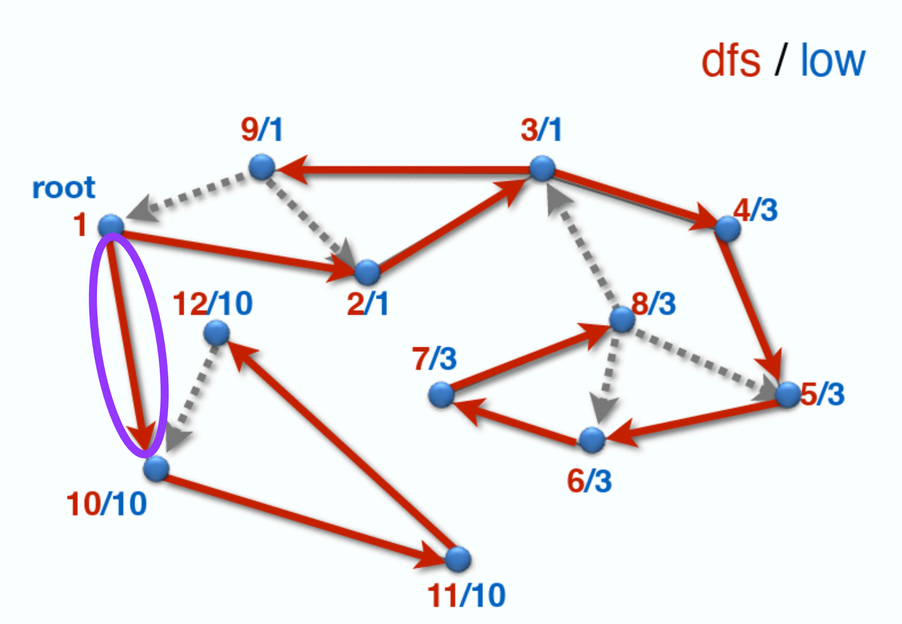


A vertex v is an **articulation point** iff

1) v ≠ root and v has a child u in DFS-Tree with low[u] ≥ dfs[v] or
2) v = root and v has at least 2 children in DFS-Tree

Articulation Points and Bridges

Finding articulation points



A tree edge e = (v,w) is a **bridge** iff

low[w] > dfs[v]

Remaining edges can never be a bridge

Cycles

Cycles

Definitions

Closed walk

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

Cycle

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

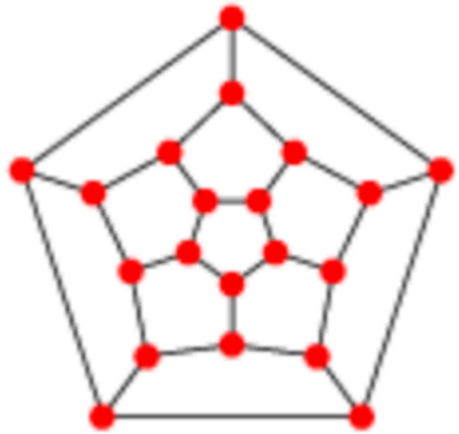
- Eulerian Cycle
  - A closed walk in G that contains every edge exactly once

minitest 4,6,7

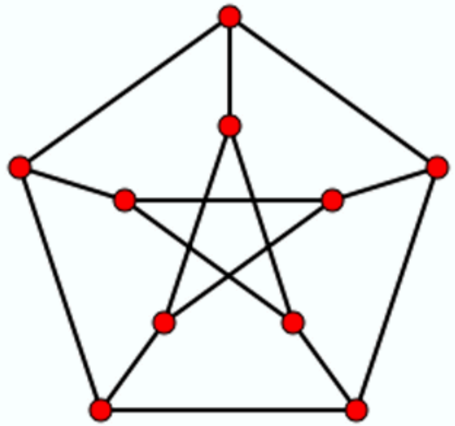
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once



Ikosaeder

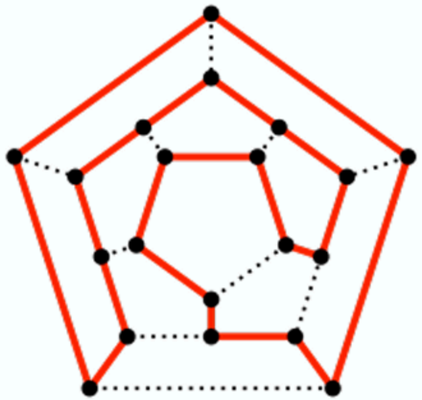


Petersengraph

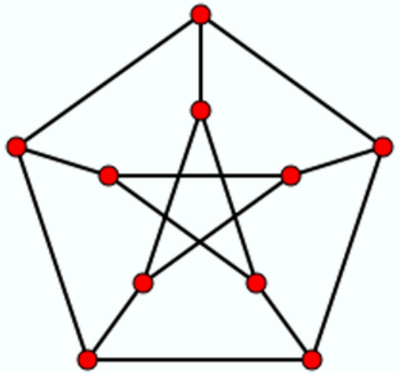
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once



Ikosaeder



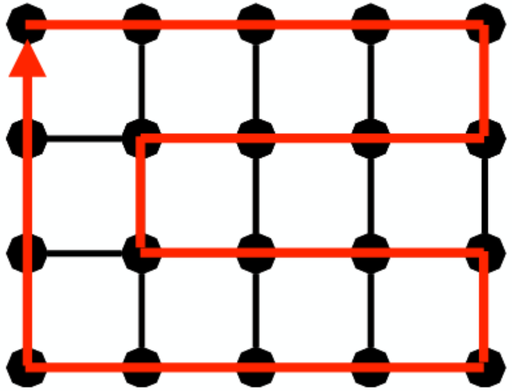
Petersengraph

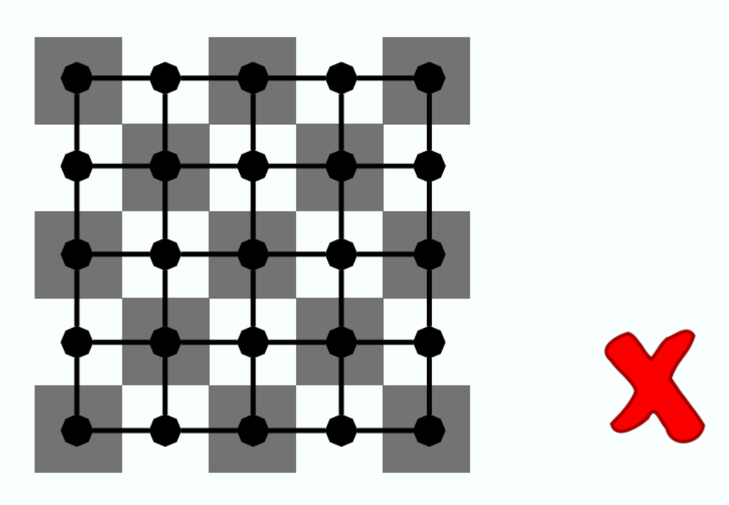
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

Grid Graph





Let $m, n \geq 2$

A n x m Grid has a hamiltonian cycle iff n x m is even

Cycles

Hamiltonian Cycle Examples

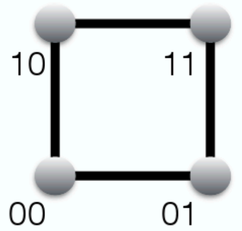
- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

d-dimensional Hypercube H_d

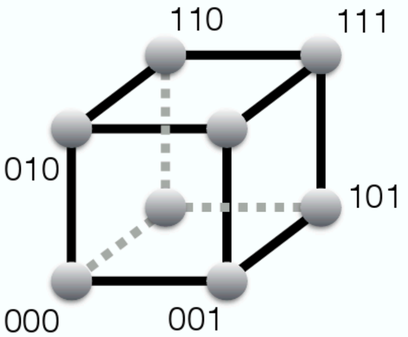
V := $\{0,1\}^d$

E := "All vertex pairs that differ in only one coordinate"

d=2:



d=3:



Has a hamiltonian cycle for all $d \geq 2$

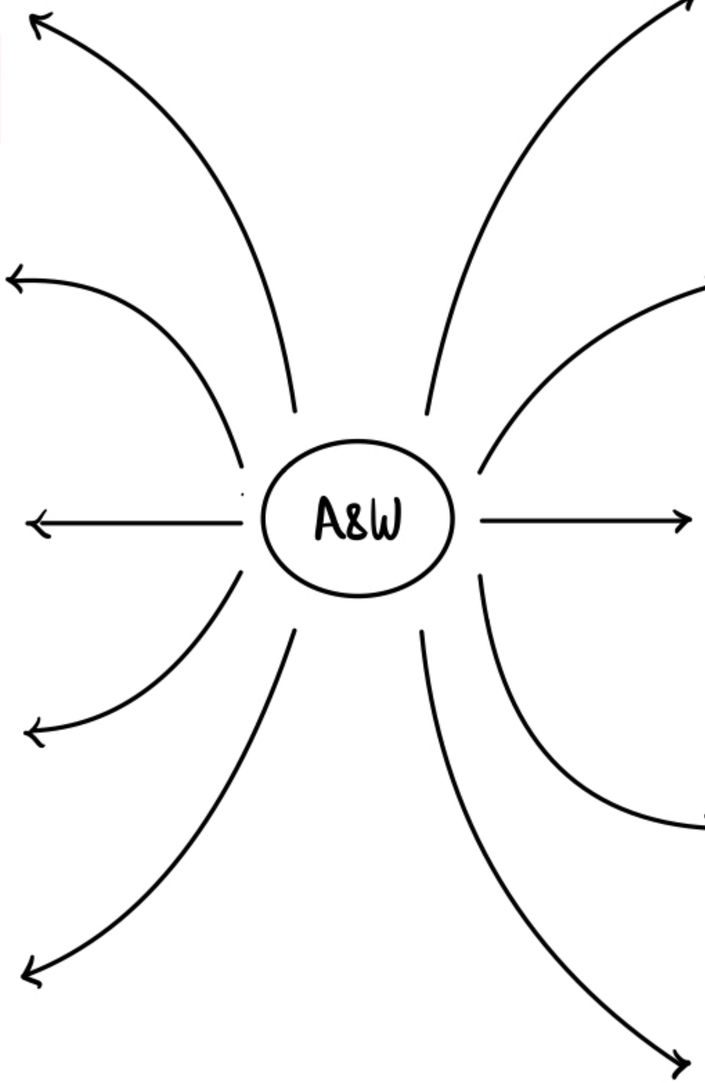
A&W

Exercise Session 3

Cycles , TSP

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Menger's Theorem
- Bridges
- Block-Decomposition

Cycles

- Closed Eulerian Walk
- TSP
- Hamiltonian Cycle

Matchings

- Definition
- Hall's Theorem
- Algorithms

Colorings

- Definition
- Brooks's Theorem
- Algorithm

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
  - Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow-Mincut
- Ford-Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- Some logistics
- Connectivity Kahoot
- Cycles
- TSP

Logistics

- For every exercise, you'll receive **feedback** from me on the exercise session next week !

- **Anki** cards
- **CodeExpert** videos

- Regular recap **kahoots** in class on the weeks without the minitest

Connectivity Kahoot

Cycles

Cycles

Definitions

Closed walk

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **closed walk** (german "Zyklus") if it is a walk, $k \geq 2$ and $v_0 = v_k$.

Cycle

- A sequence of vertices $(v_0, v_1, \ldots, v_k)$ is a **cycle** (german "Kreis") if it is a closed walk, $k \geq 3$ and all vertices (except v_0 and v_k) are distinct.

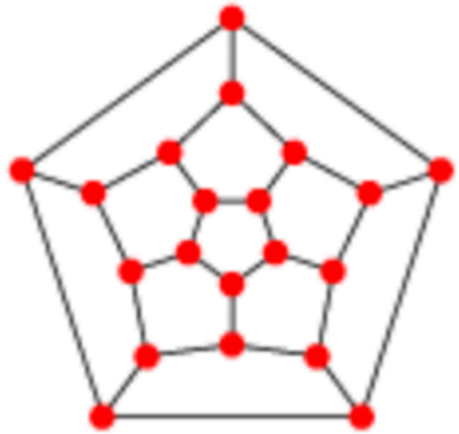
- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

- Eulerian Cycle
  - A closed walk in G that contains every edge exactly once

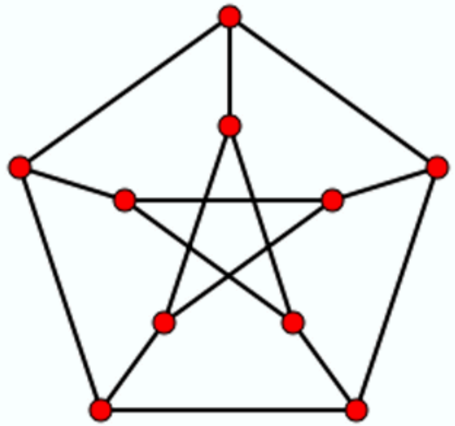
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once



Ikosaeder

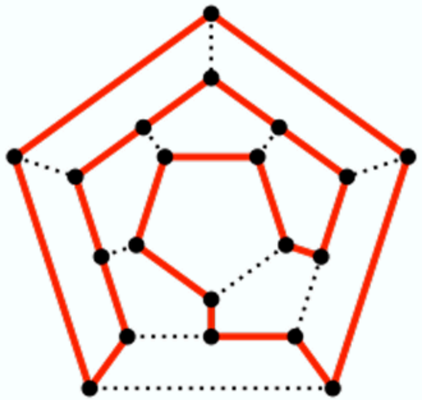


Petersengraph

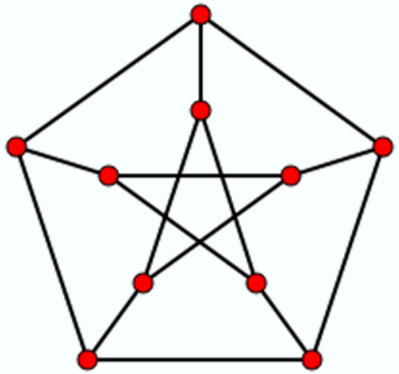
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once



Ikosaeder



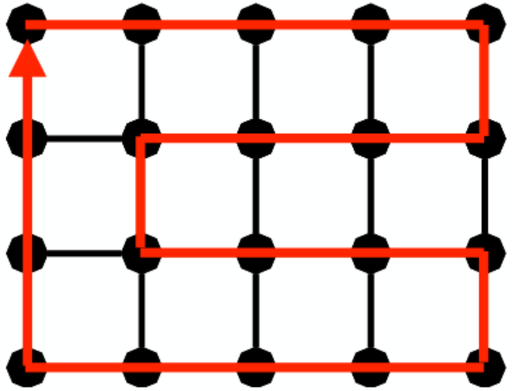
Petersengraph

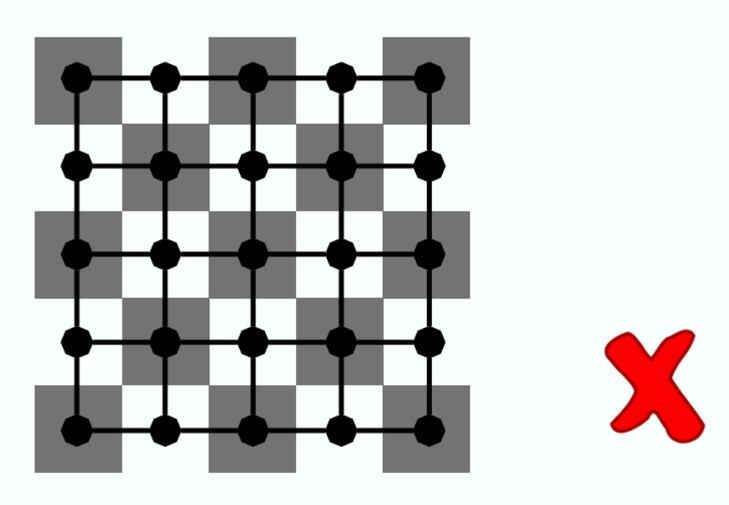
Cycles

Hamiltonian Cycle Examples

- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

Grid Graph





Let $m, n \geq 2$

A n x m Grid has a hamiltonian cycle iff n x m is even

Cycles

Hamiltonian Cycle Examples

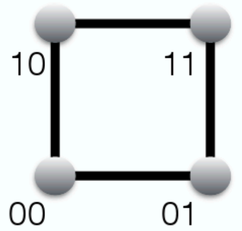
- Hamlitonian Cycle
  - A cycle in G that contains every vertex exactly once

d-dimensional Hypercube H_d

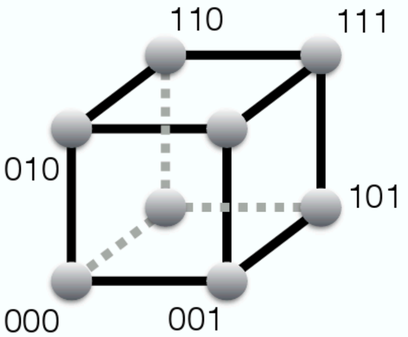
V := $\{0,1\}^d$

E := "All vertex pairs that differ in only one coordinate"

d=2:



d=3:



Has a hamiltonian cycle for all $d \geq 2$

Eulerian Cycle

Lemma

A connected G has a eulerian Cycle $\iff$ Every vertex has an even degree

Let’s take a break

Hamiltonian Cycle

Given a Graph $G = (V, E)$, does G have a hamiltonian cycle ?

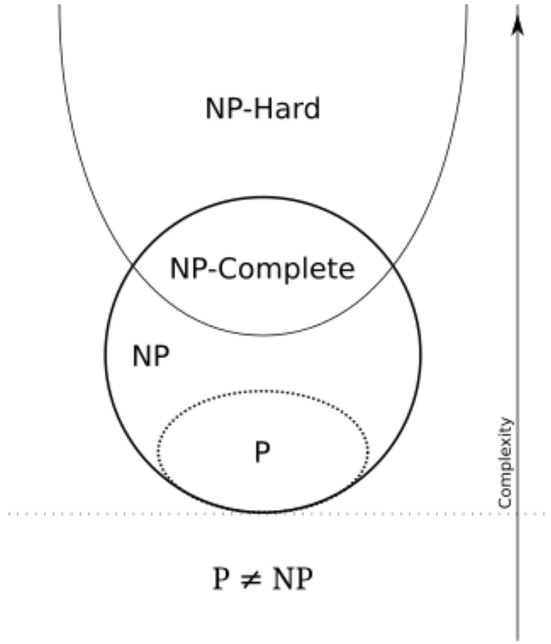
NP - Complete

NP-Complete

Given a Graph $G = (V, E)$, does G have a hamiltonian cycle ?

Complexity Theory

TI next semester



P : polynomial

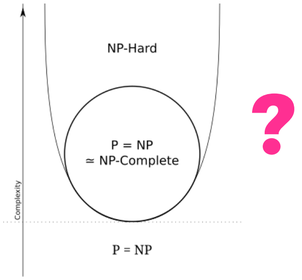
NP : non-deterministic polynomial

- NP is the set of decision problems *solvable* in polynomial time by a nondeterministic Turing machine.
- NP is the set of decision problems *verifiable* in polynomial time by a deterministic Turing machine.

NP - Complete

A problem a in NP is NP-complete if :

$$a \in P \implies P = NP$$



?

Hamiltonian Cycle

Dirac's Theorem

$|V| \geq 3$ and

the minimum degree $\delta(G) \geq |V|/2$ $\Longrightarrow$ A G has a hamiltonian cycle

Hamiltonian Cycle

DP Approach

For all $S \subseteq [n]$ with $1 \in S$ and all $x \in S$ with $x \neq 1$:

$$P[S][x] = \begin{cases} 1 & \text{, if there exists a 1-x-path that only uses vertices from S} \\ 0 & \text{, else} \end{cases}$$

Initialization : $P[\{1,x\}][x] = 1$ iff $\{1,x\} \in E$

Schleife:

for all $s = 3$ to n
for all $S \subseteq [n]$ mit $1 \in S$ und $|S| = s$:
for all $x \in S$ mit $x \neq 1$:

Rekursion

$$P_{S,x} = \max \{ P_{S\setminus\{x\},y} : y \in N(x) \cap S, y \neq 1 \}$$

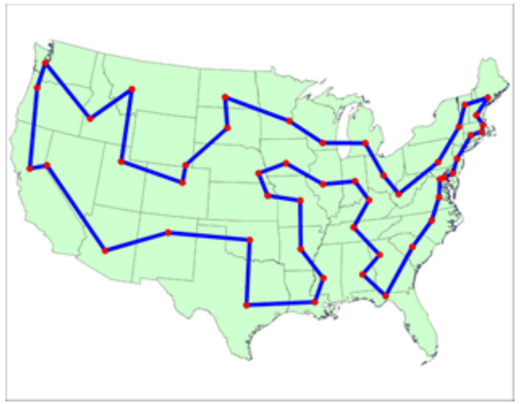
Ausgabe: G enthält Hamiltonkreis gdw $\exists x \in N(1)$ mit $P_{[n],x} = 1$

Dynamische Programmierung:
Laufzeit: $\approx n^2 2^n$
Speicherplatz: $\approx n\, 2^n$

TSP

TSP

Problem Description



Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$

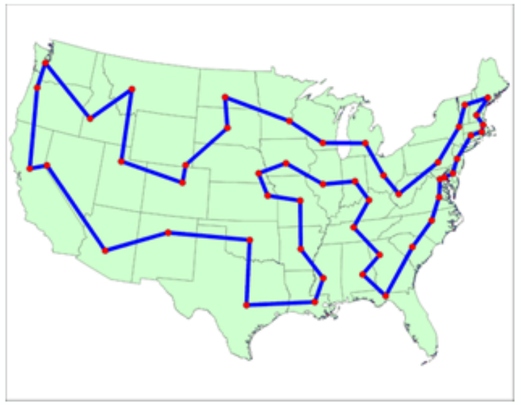
To find :
- “shortest round trip”

$$\min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$$

also NP-Complete

Metric TSP

Problem Description



Given :

- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $\quad l : \binom{[n]}{2} \to R$

To find :

- "shortest round trip"

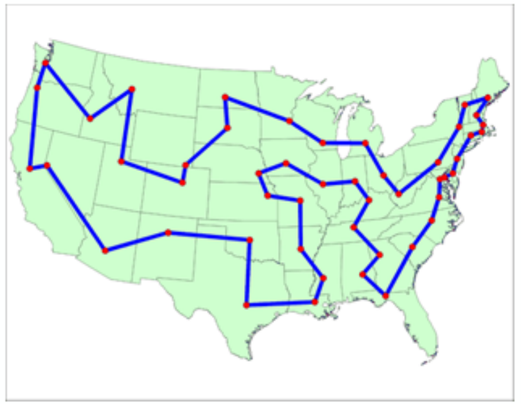
$$\min_{H\ :\ Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$$

- l satisfies the triangle inequality

$$l(x,z) \leq l(x,y) + l(y,z)$$

Metric TSP : 2-Approximation

Problem Description



Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $\quad l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t.

- l satisfies the triangle inequality

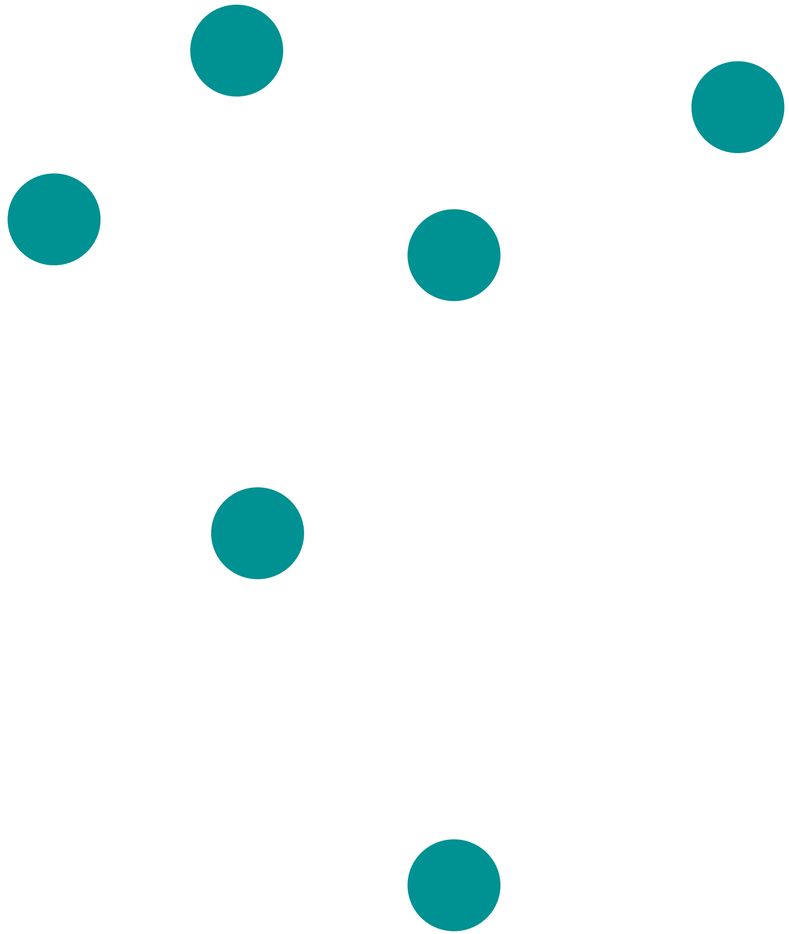
$$l(x,z) \leq l(x,y) + l(y,z)$$

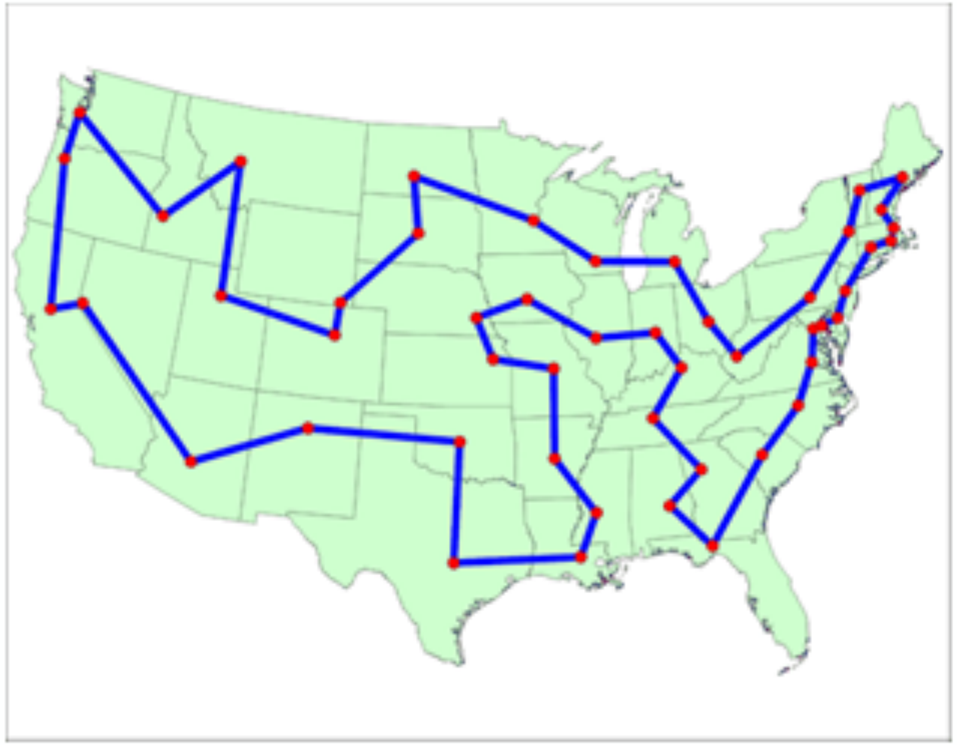
$$l(C) \leq 2\, l(OPT)$$

where $\quad OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 2-Approximation

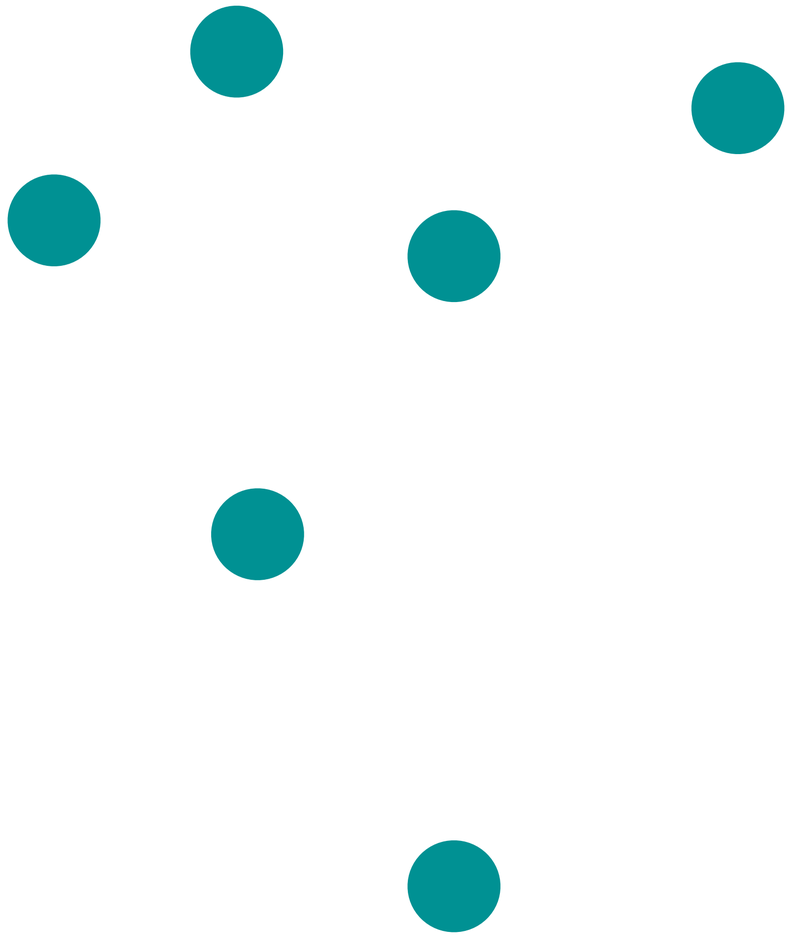
Algorithm





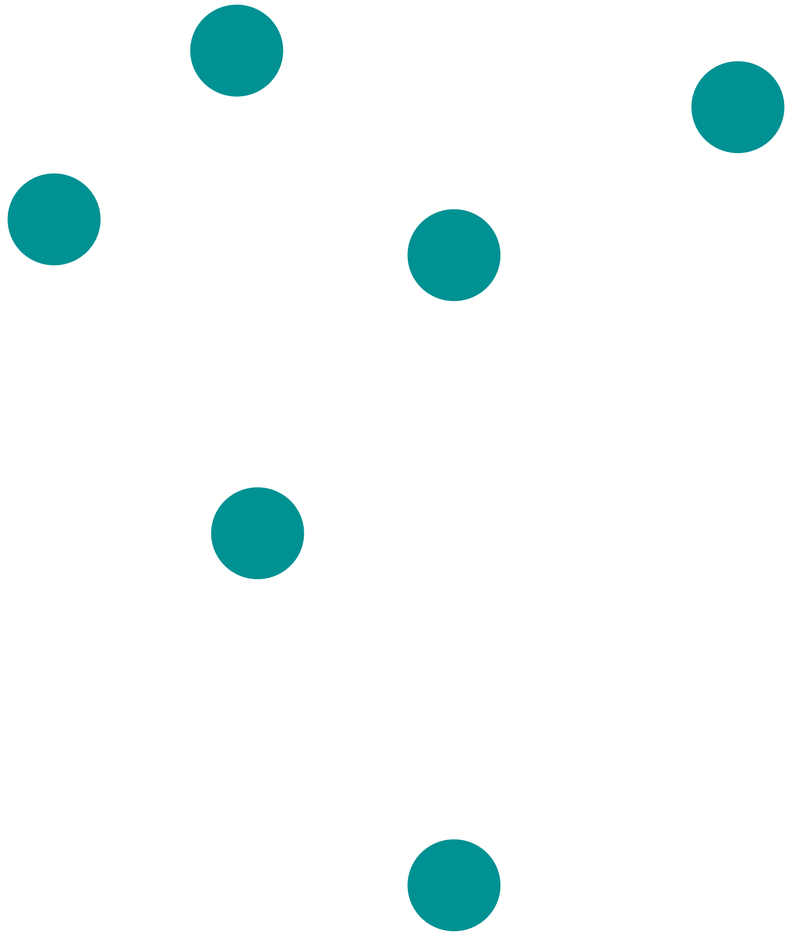
Metric TSP : 2-Approximation

Algorithm



Metric TSP : 2-Approximation

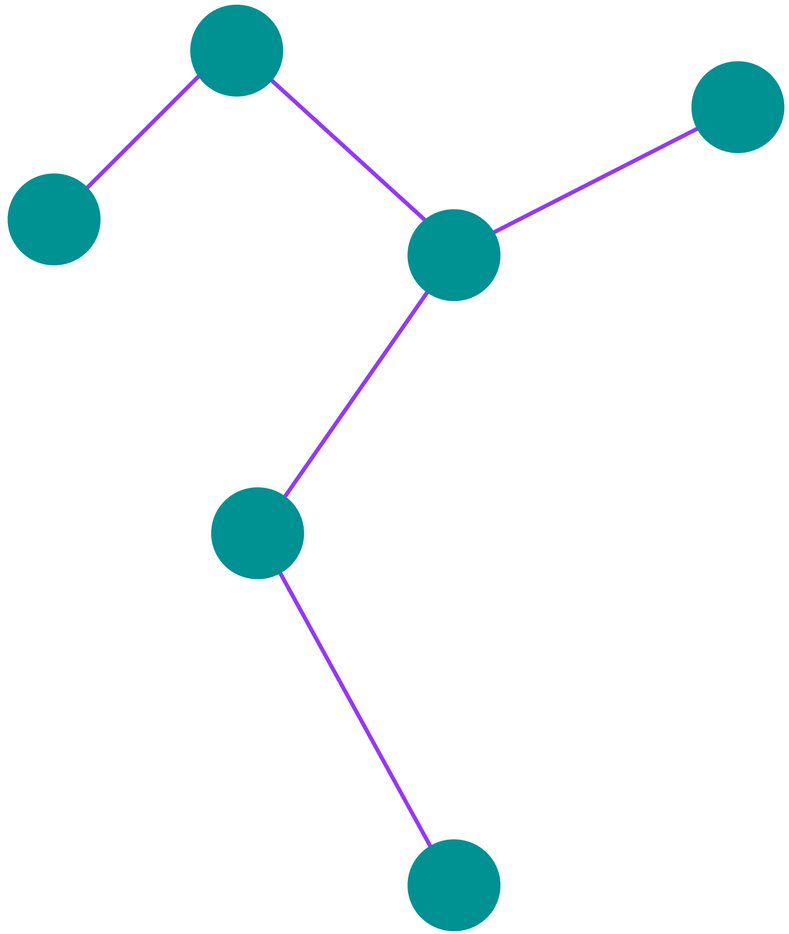
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

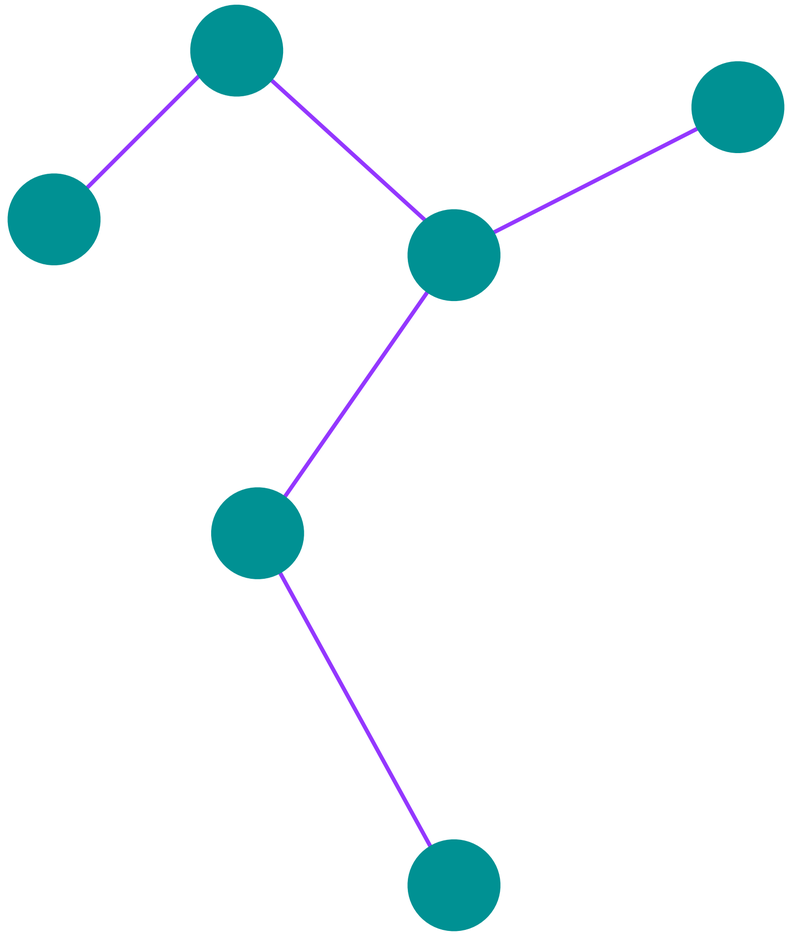
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

Algorithm

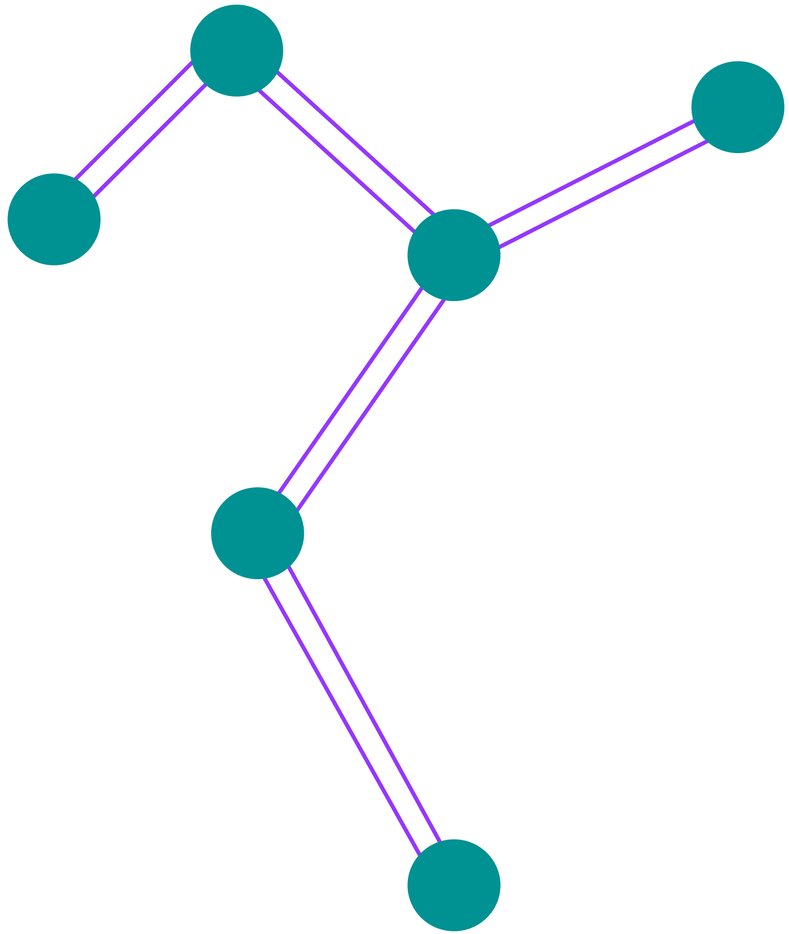


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

Algorithm

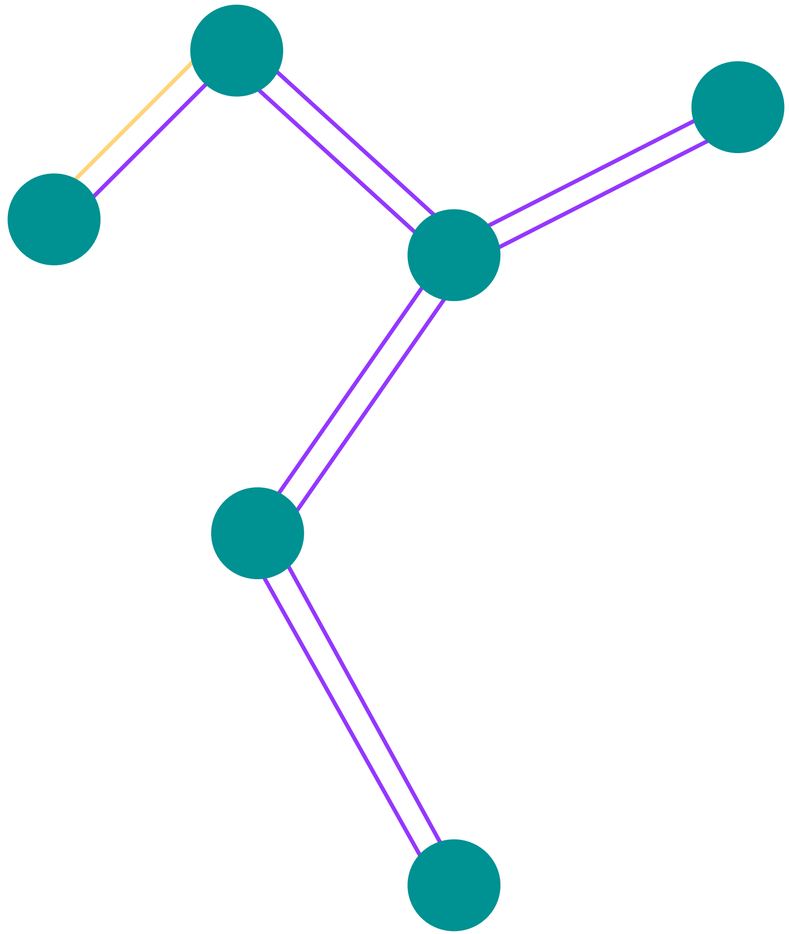


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

Algorithm



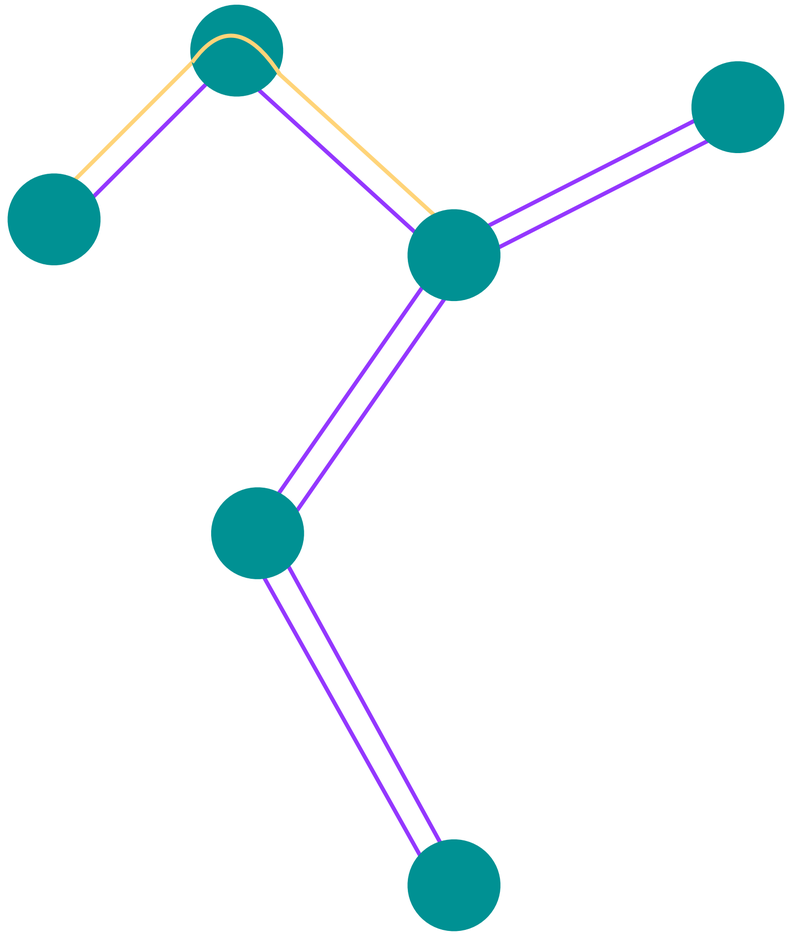
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



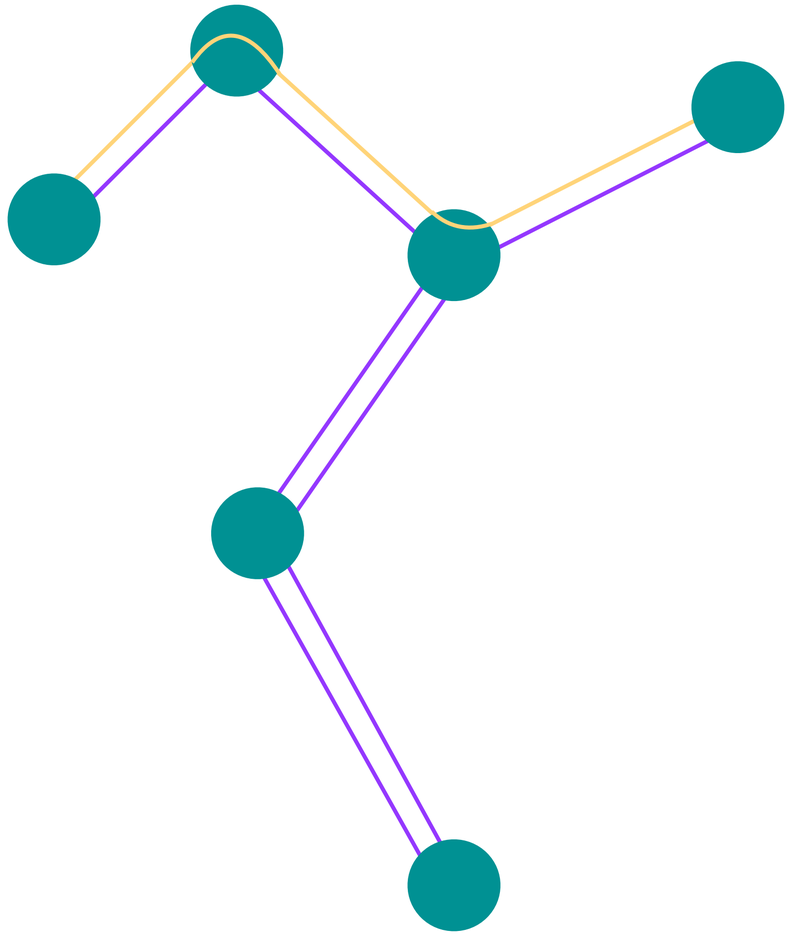
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



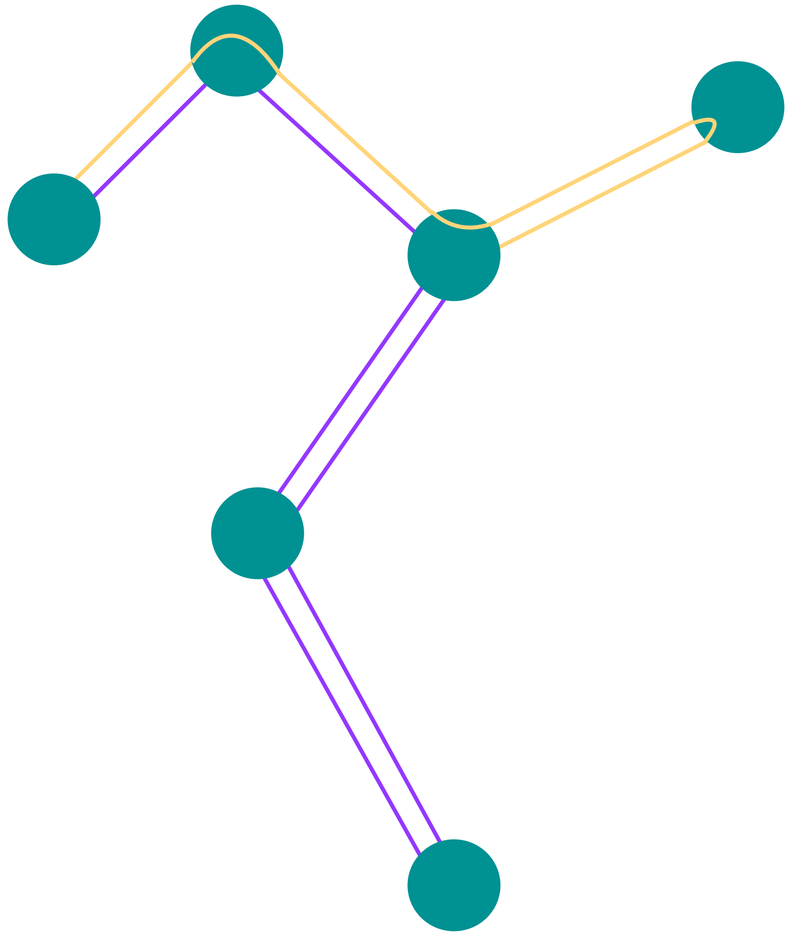
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

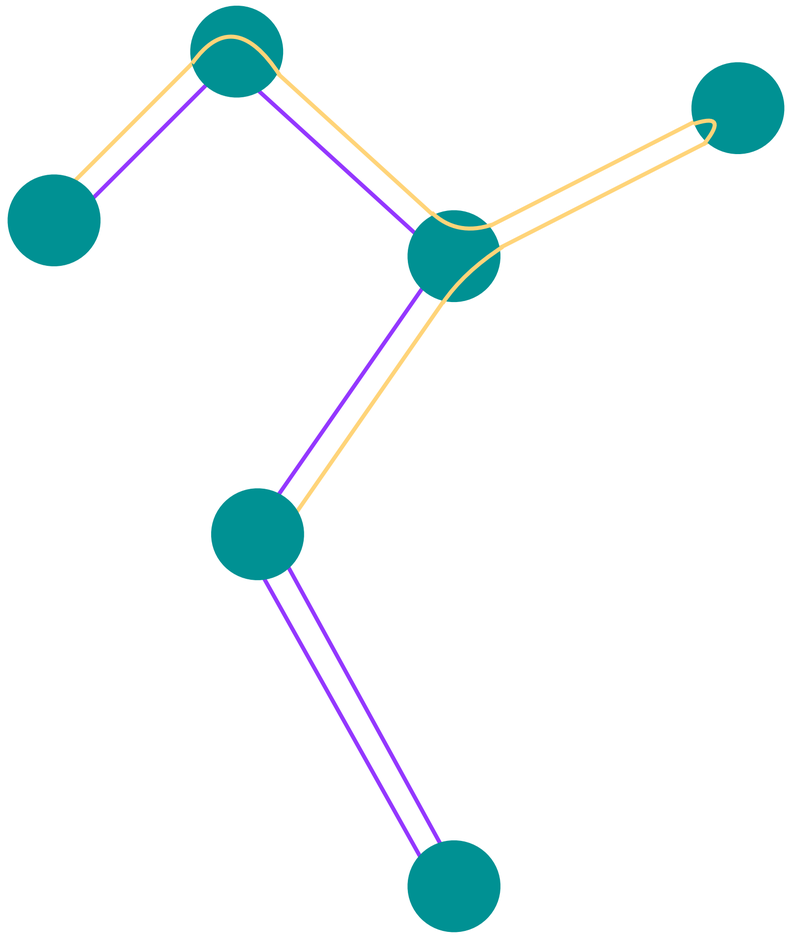
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

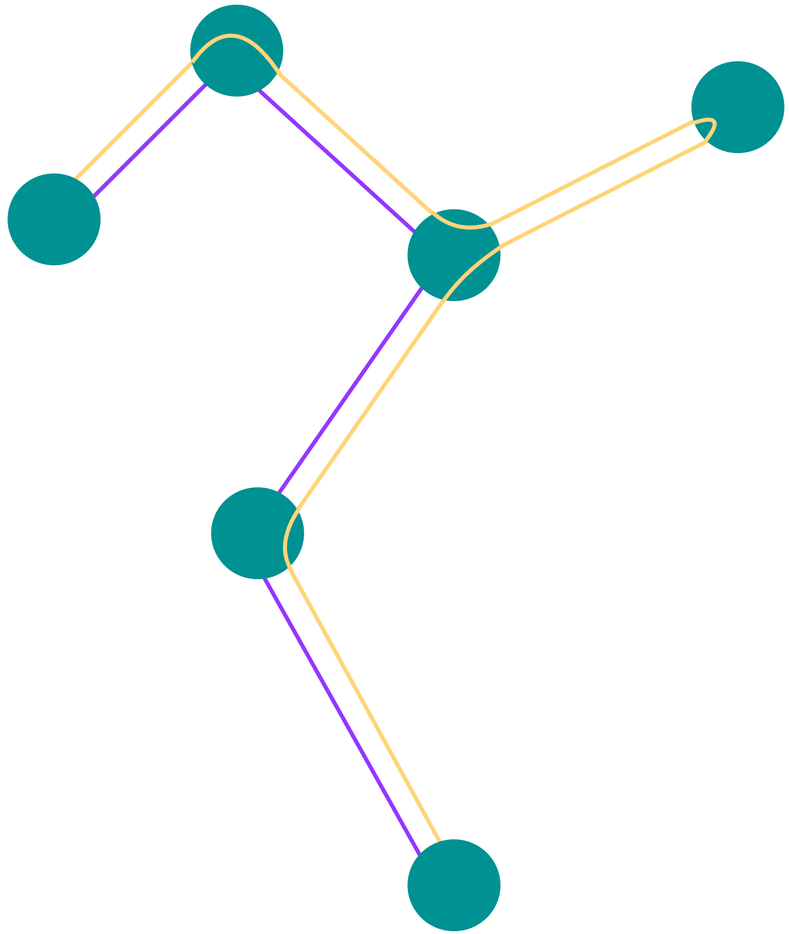
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



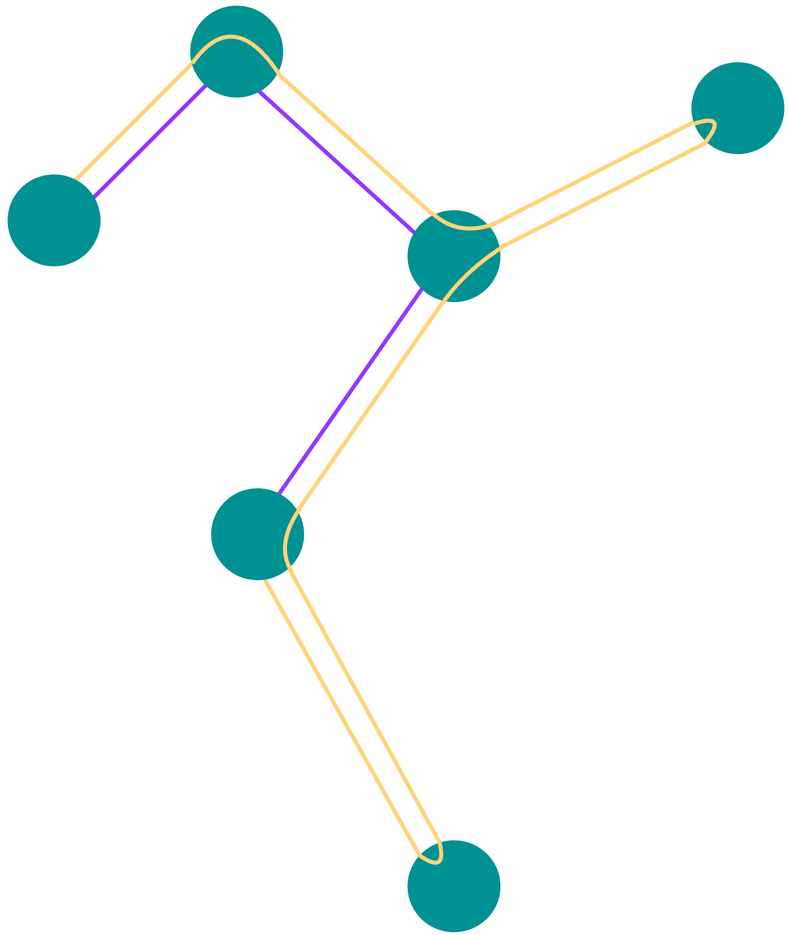
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



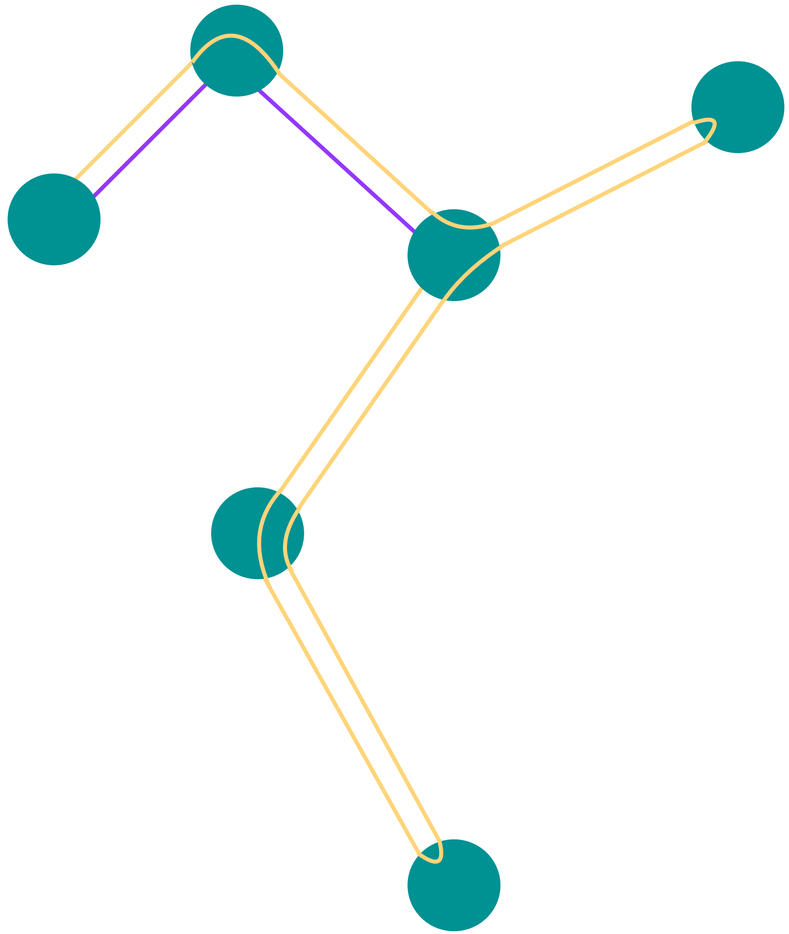
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

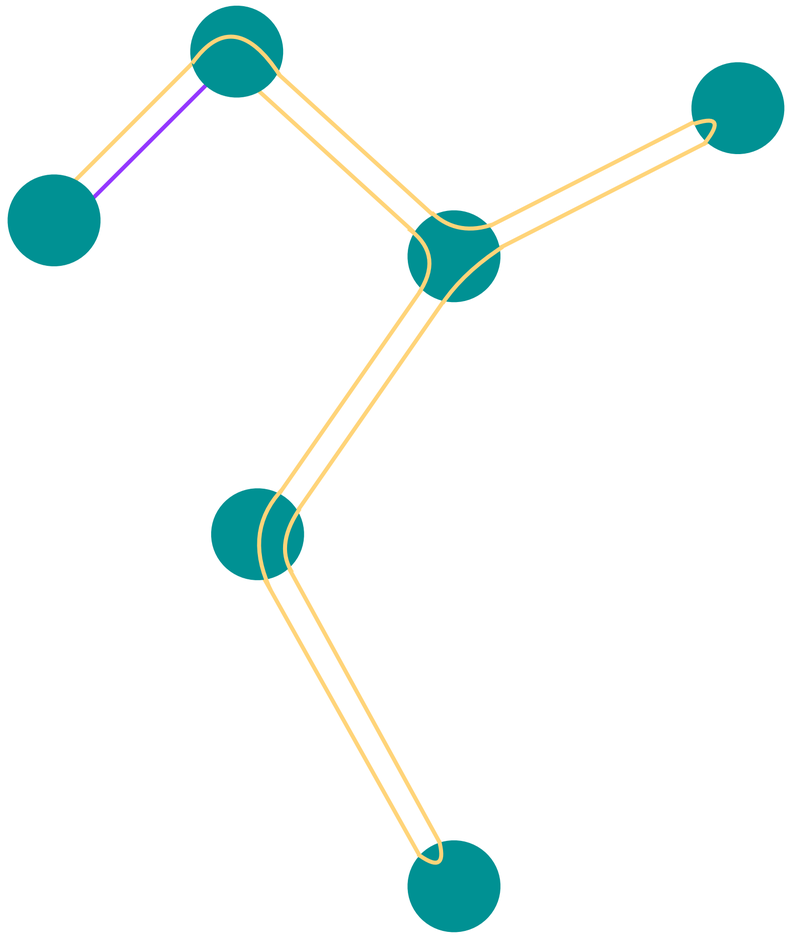
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

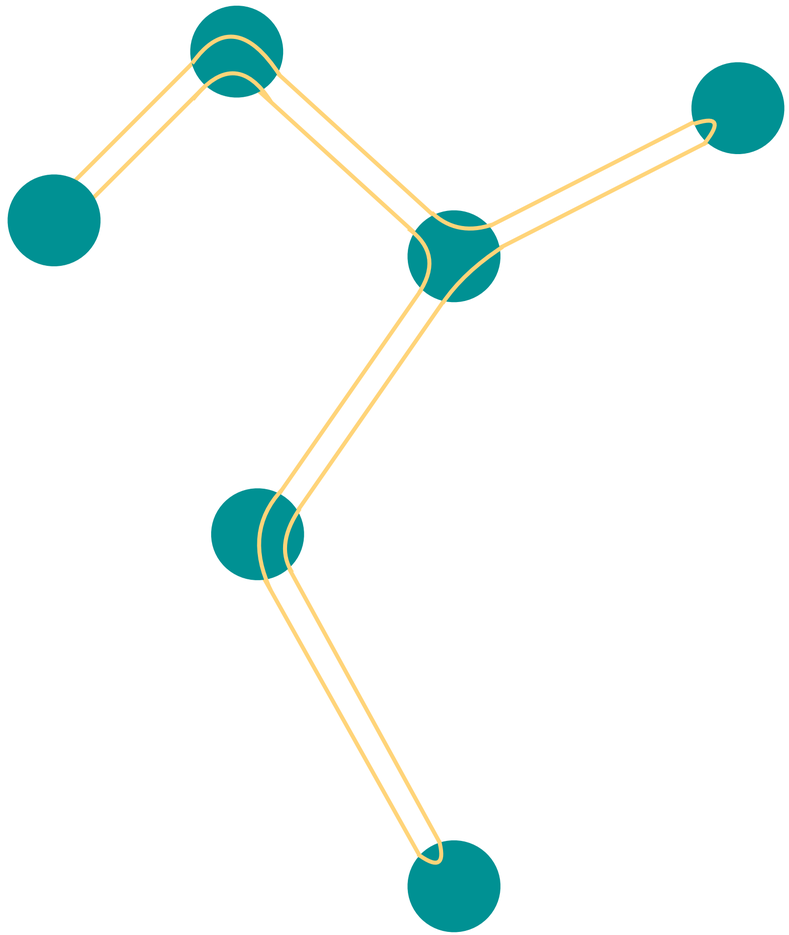
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

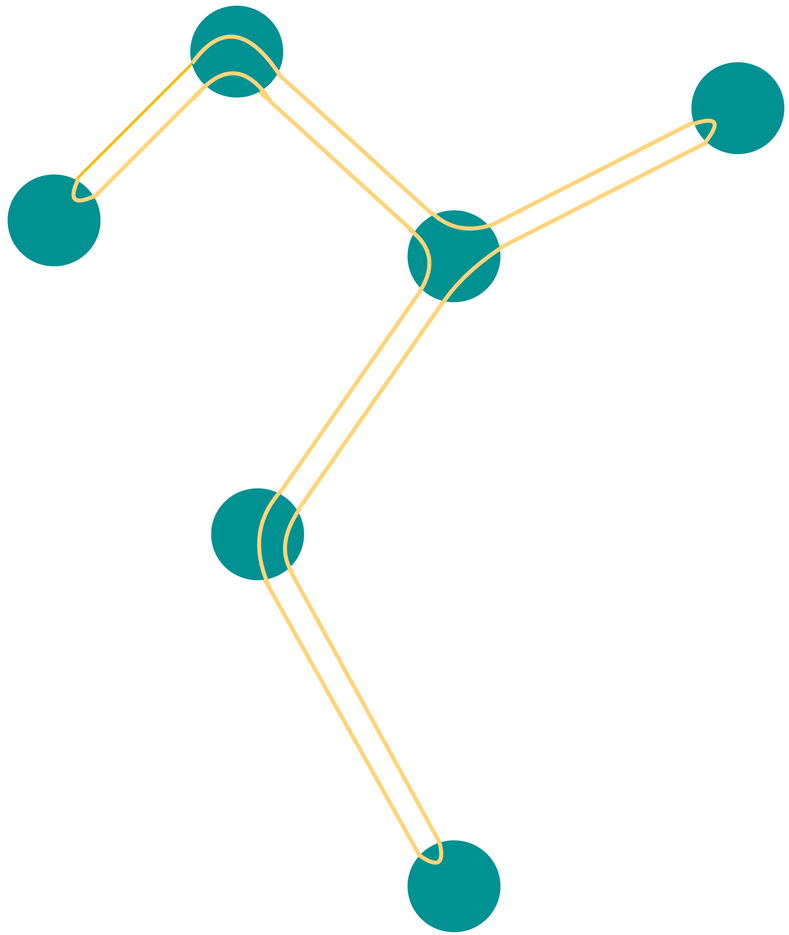
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

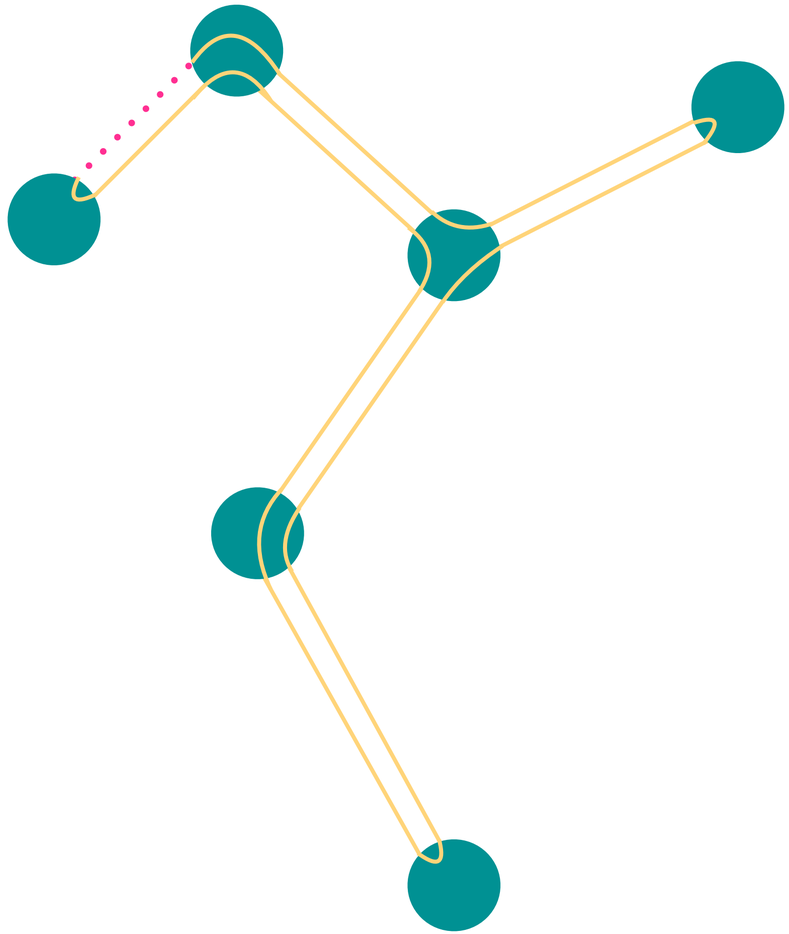
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

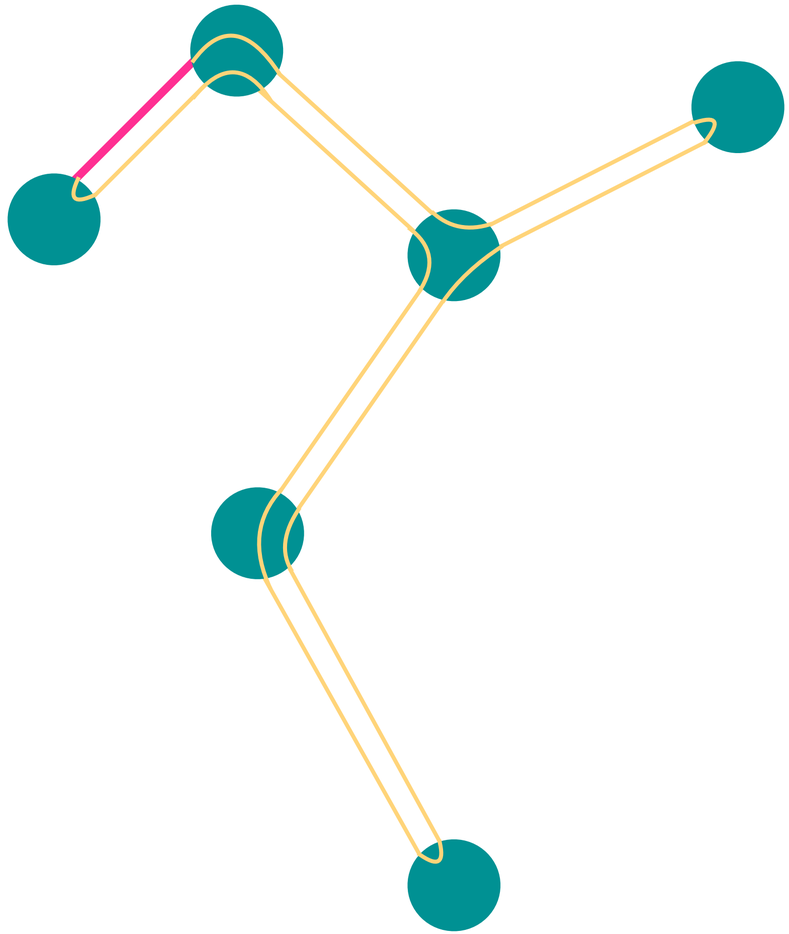
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

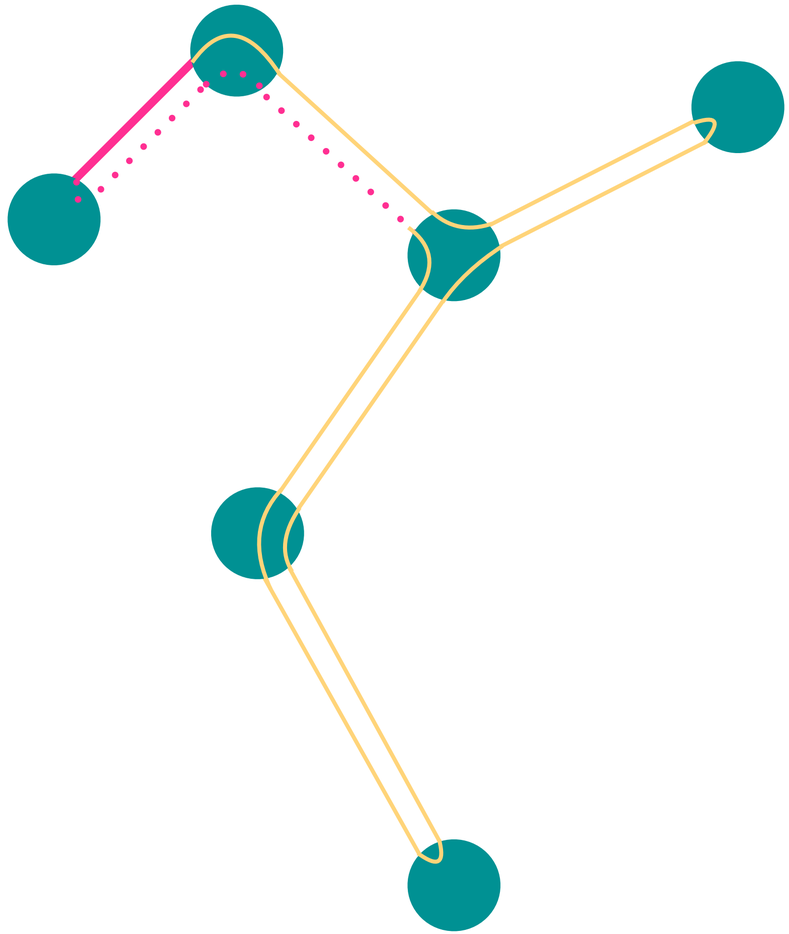
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

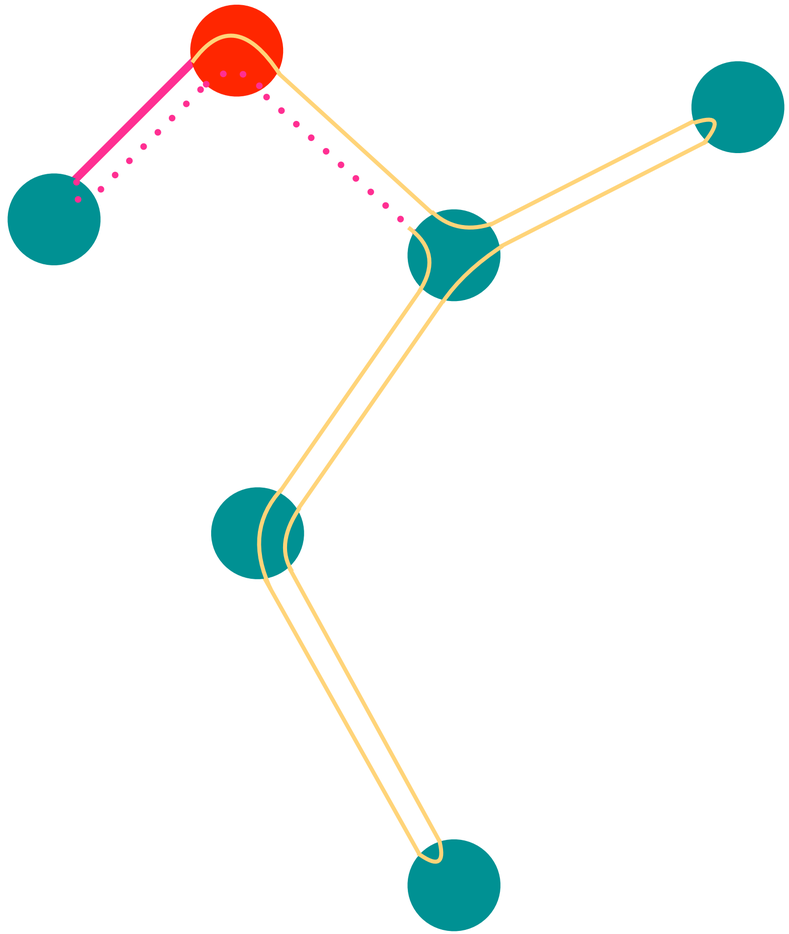
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

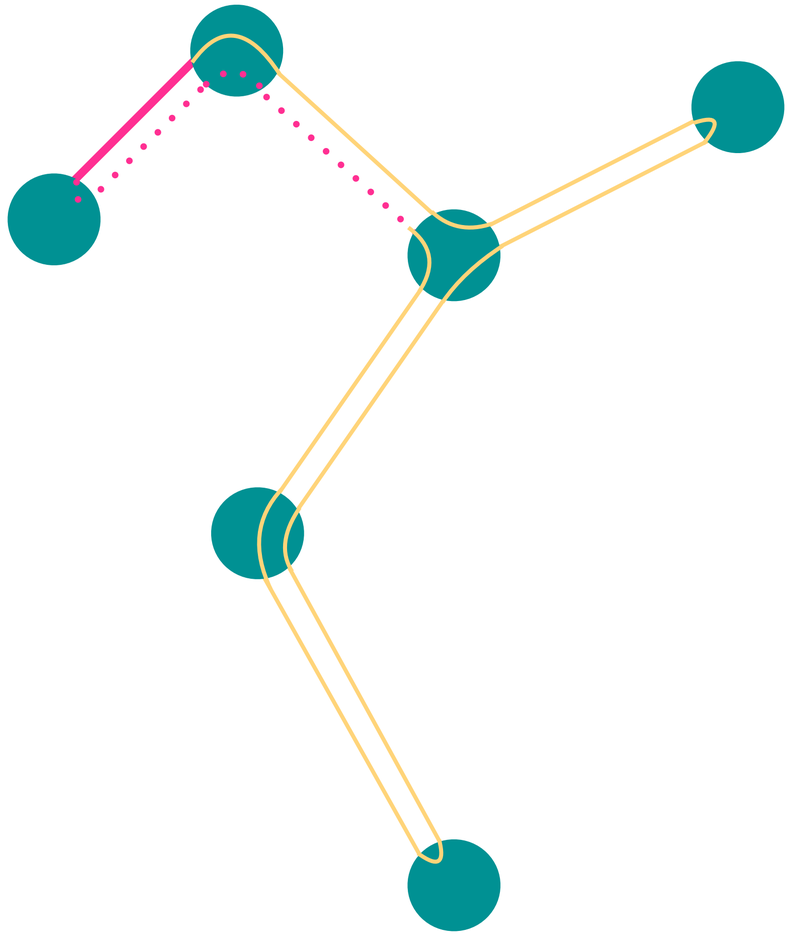
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

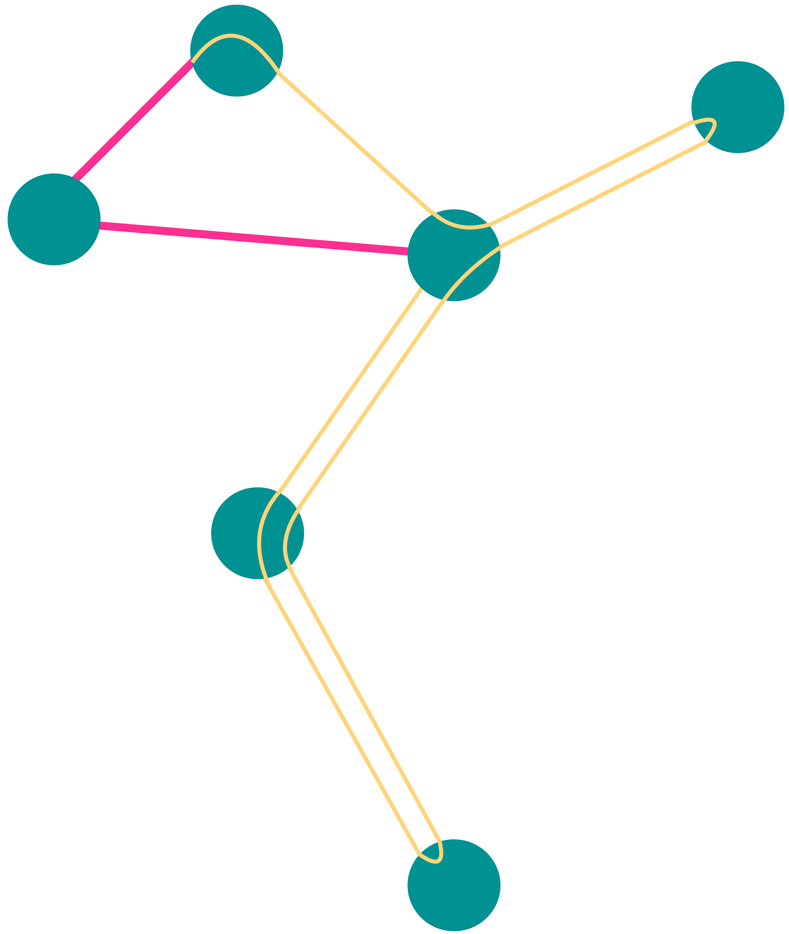
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

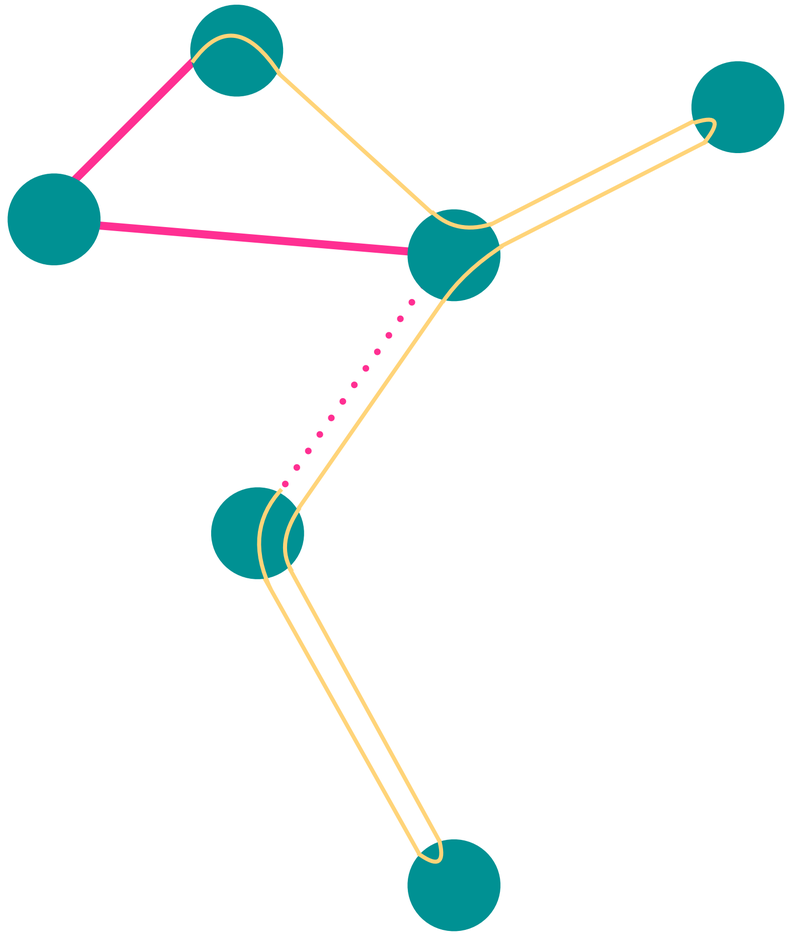
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

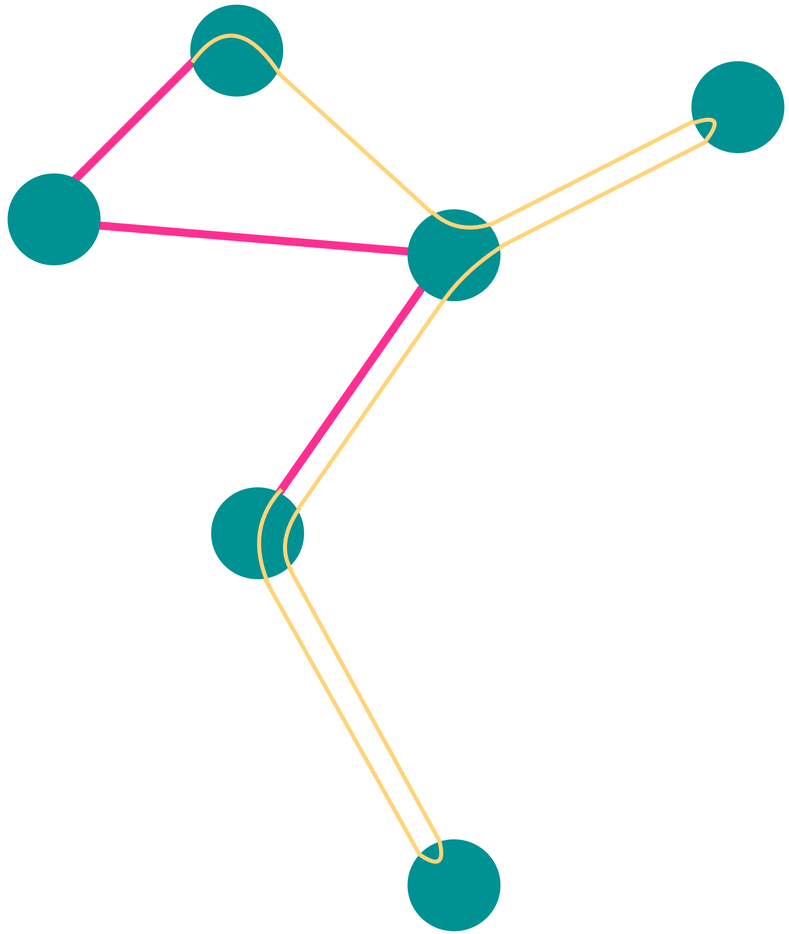
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

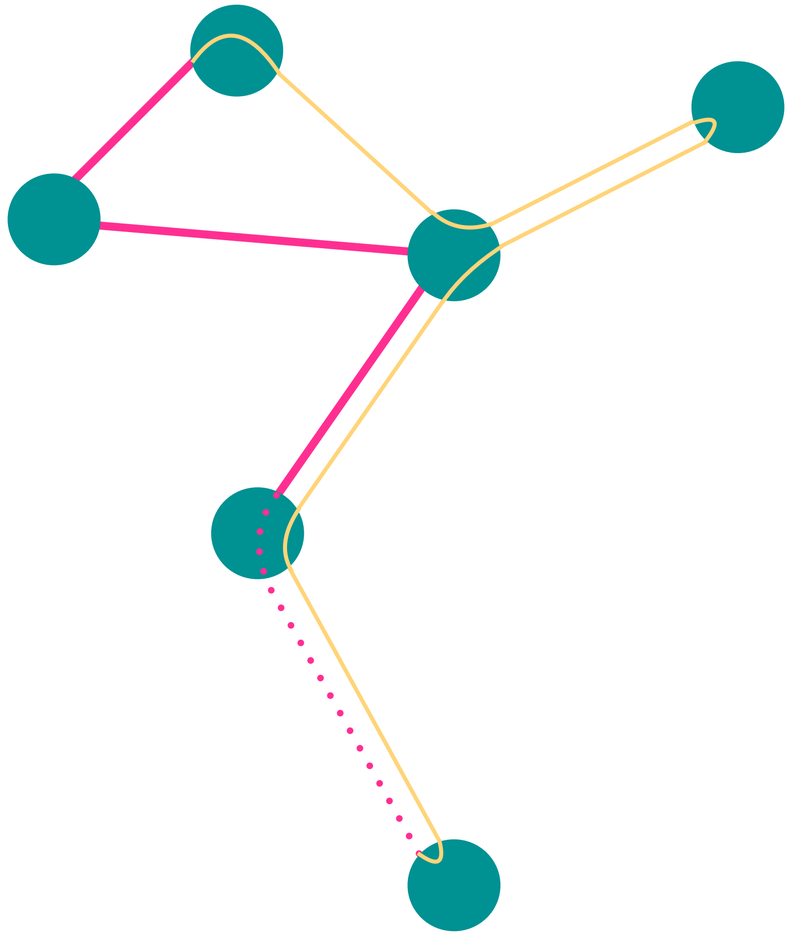
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

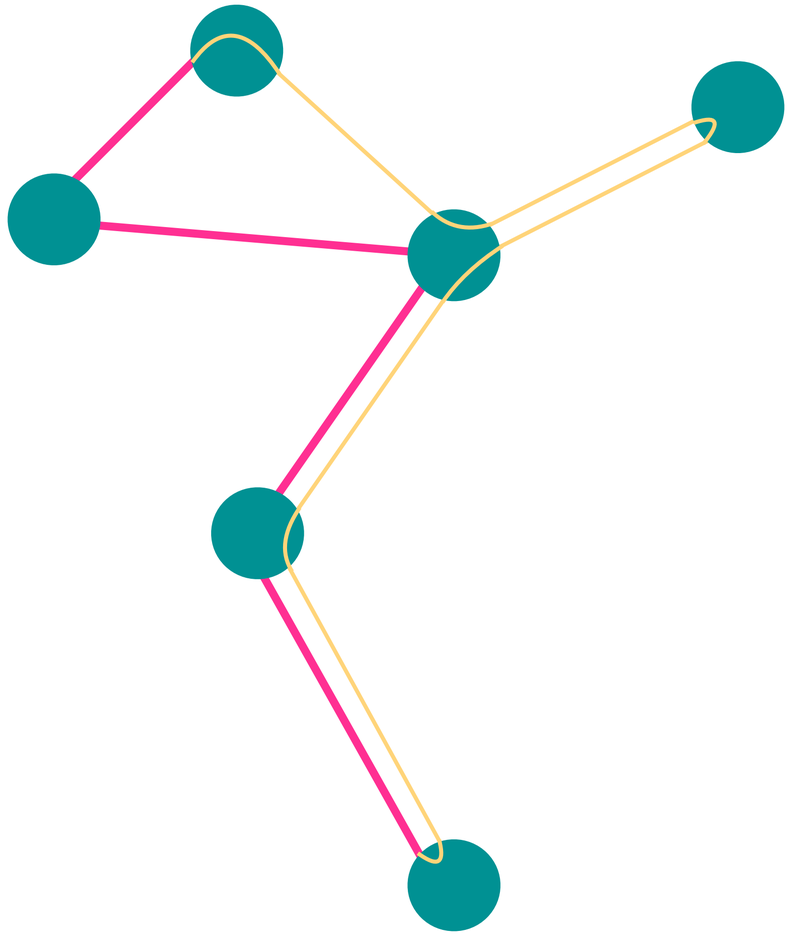
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

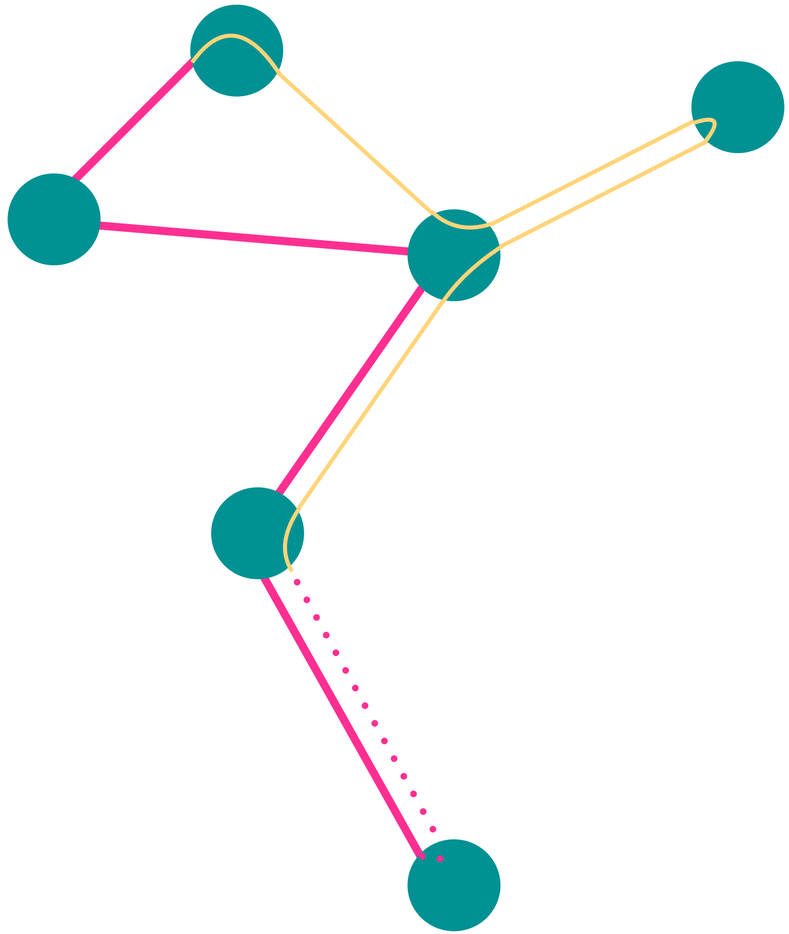
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

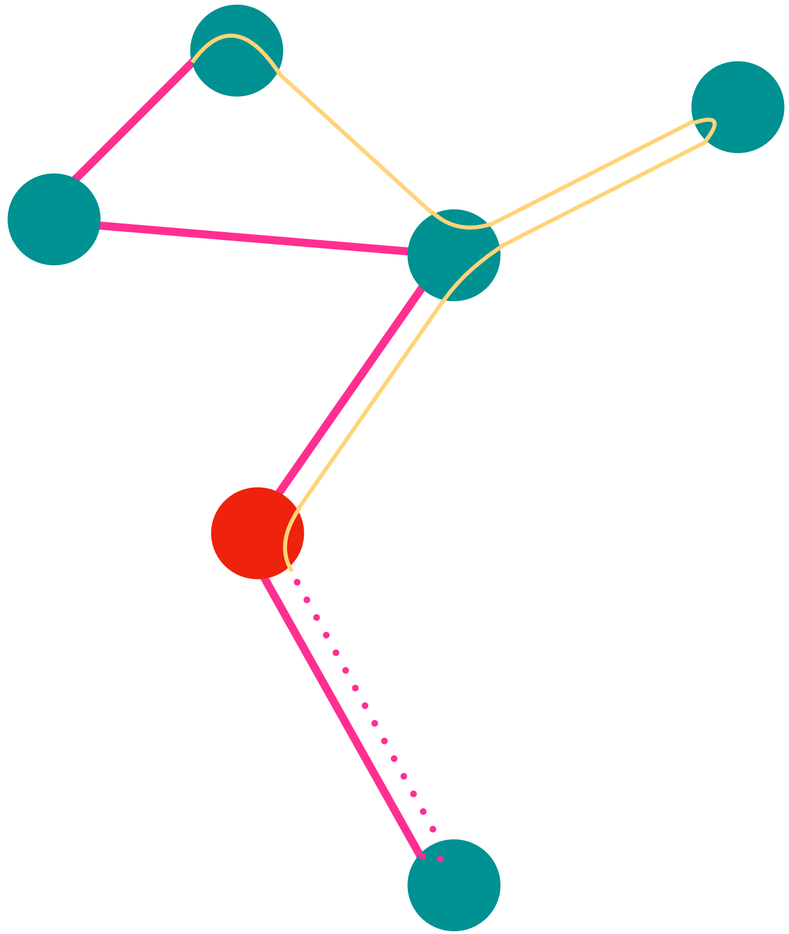
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

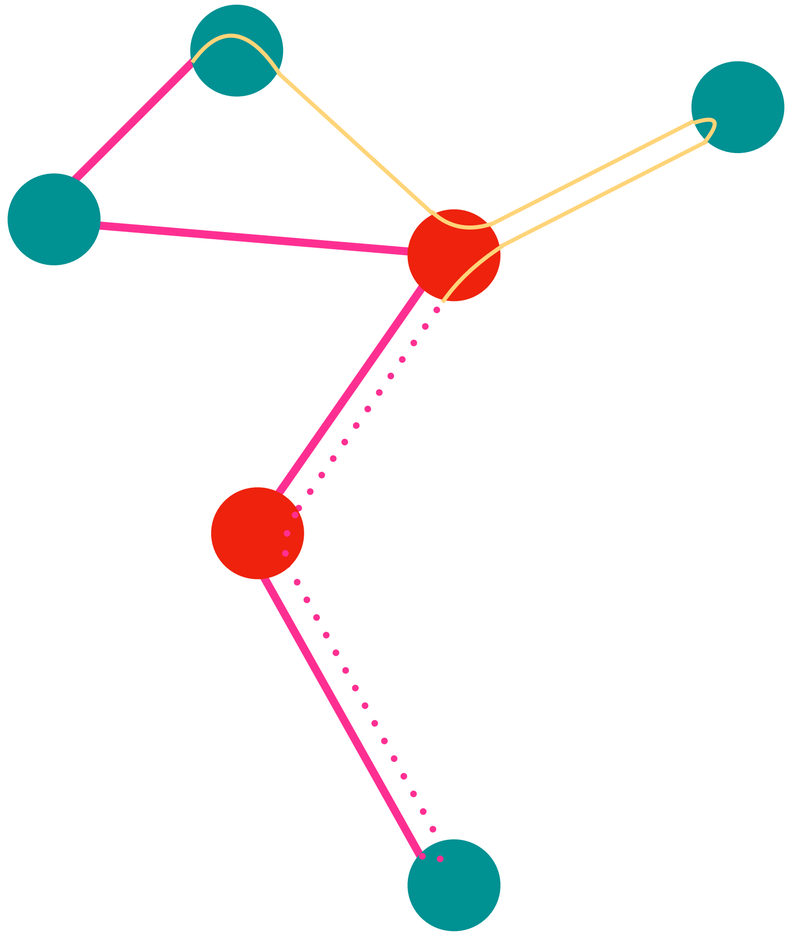
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

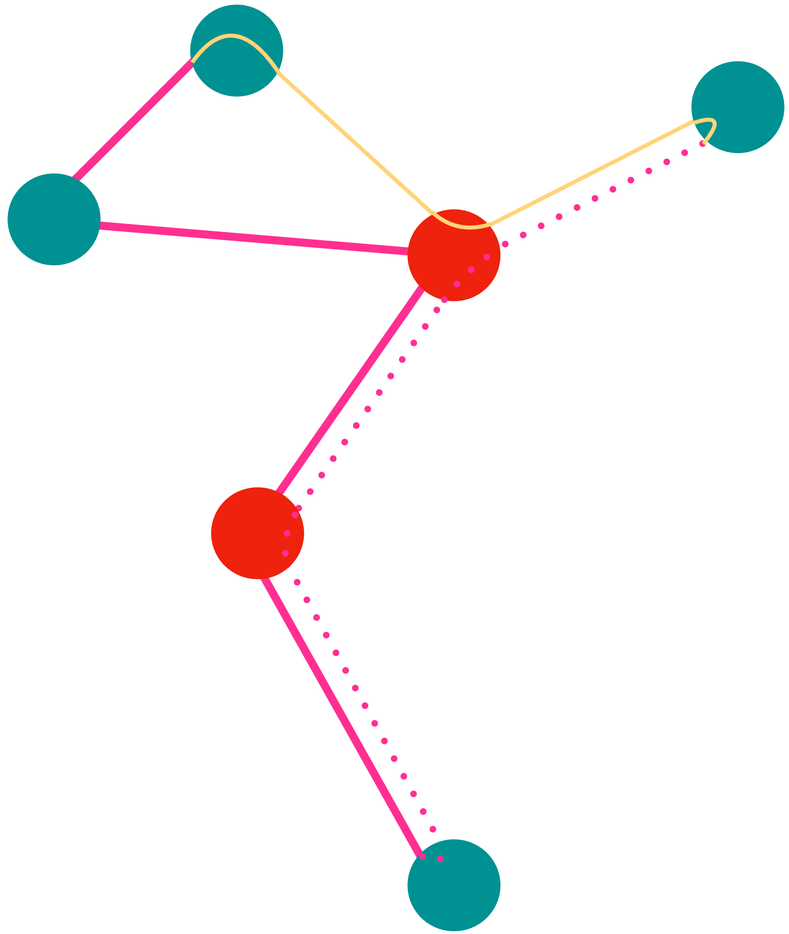
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

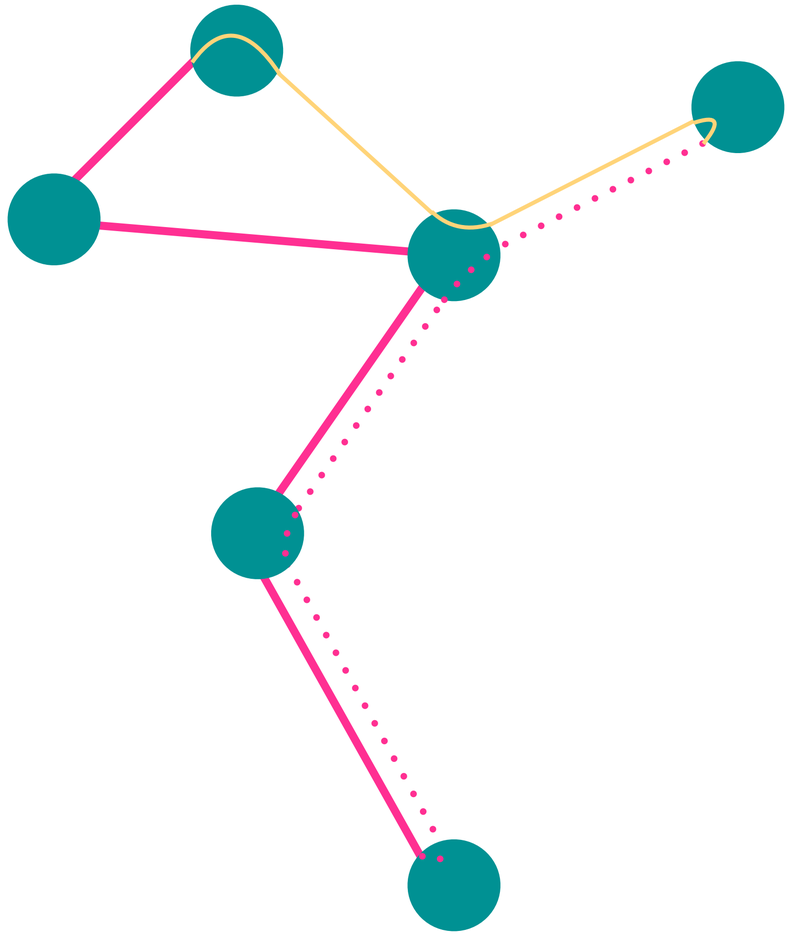
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

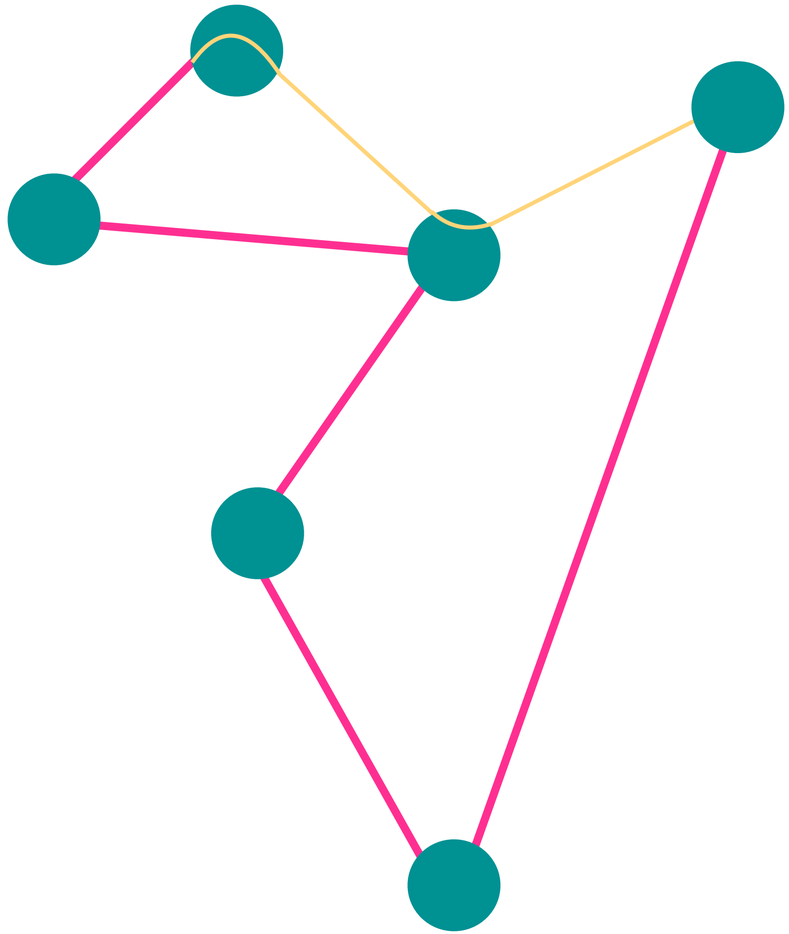
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

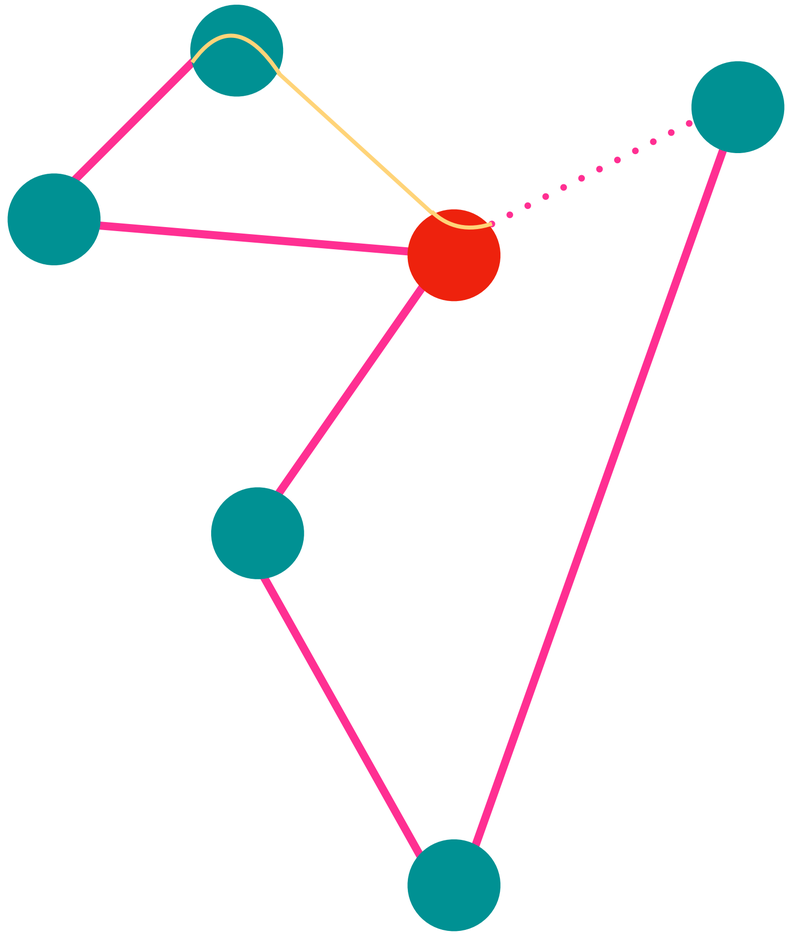
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

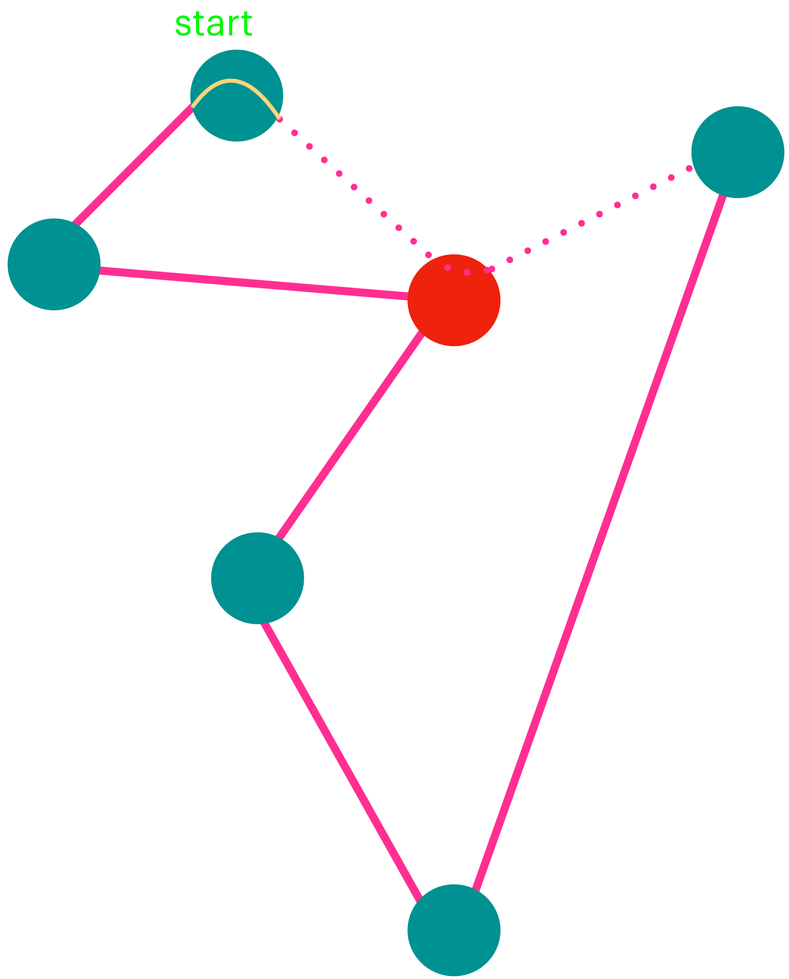
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

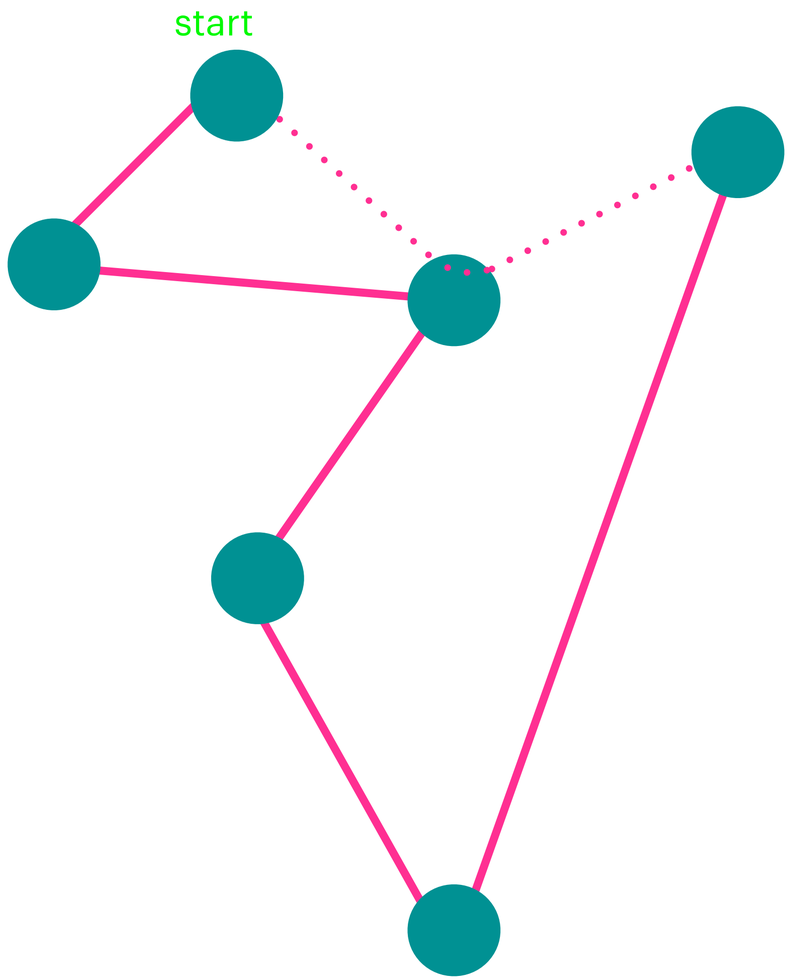
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

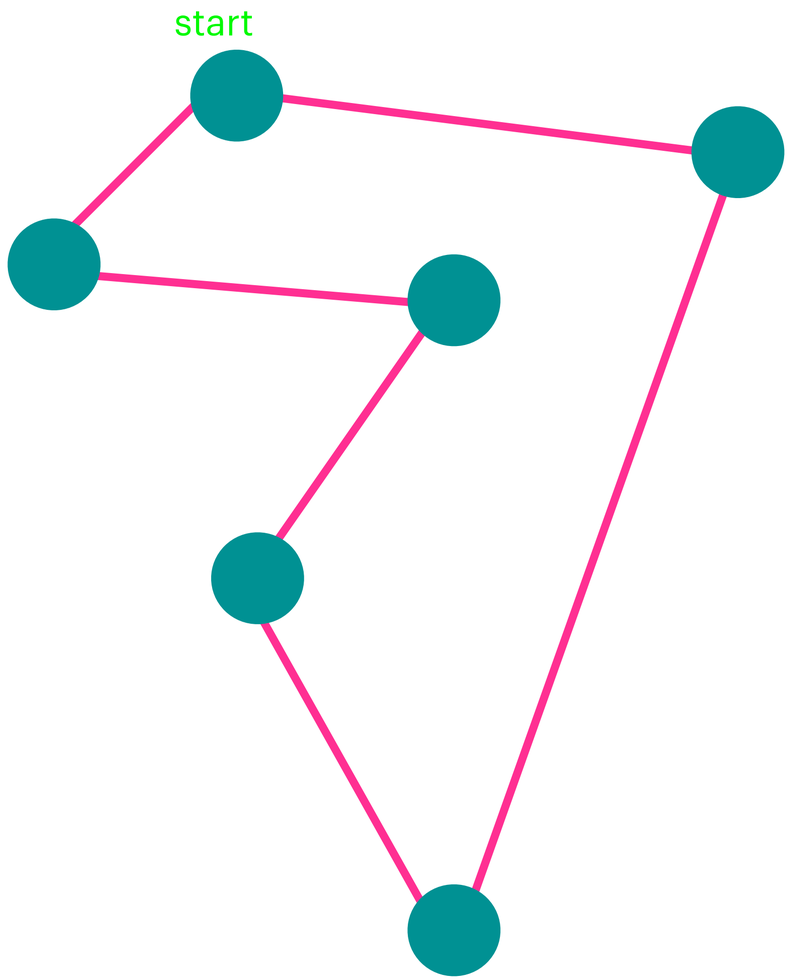
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

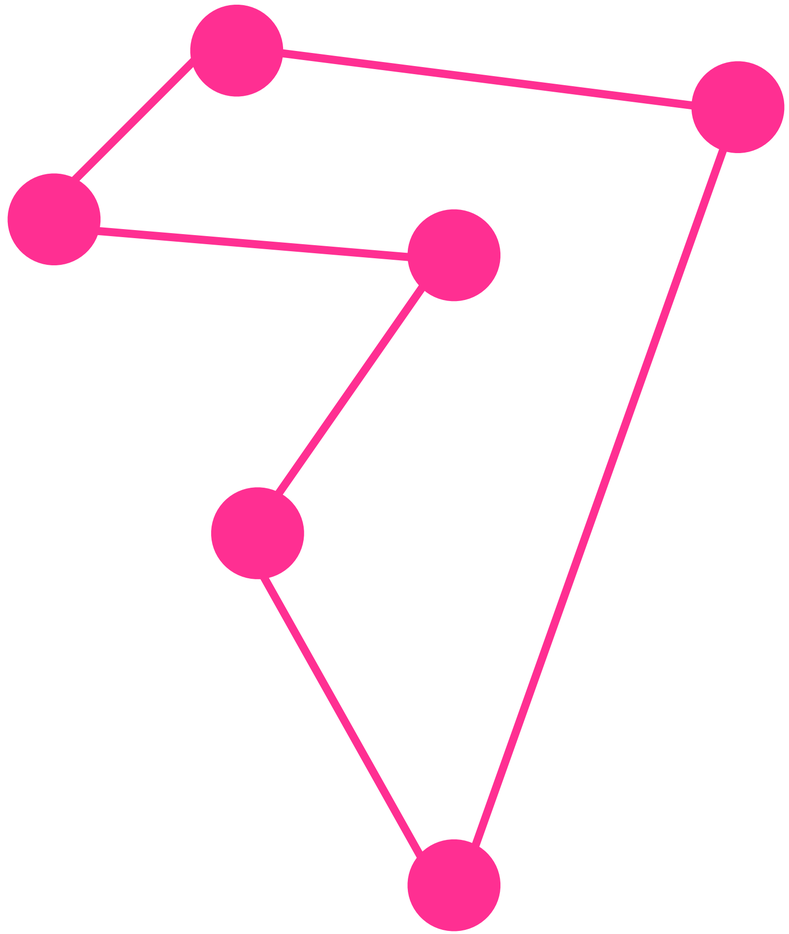
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

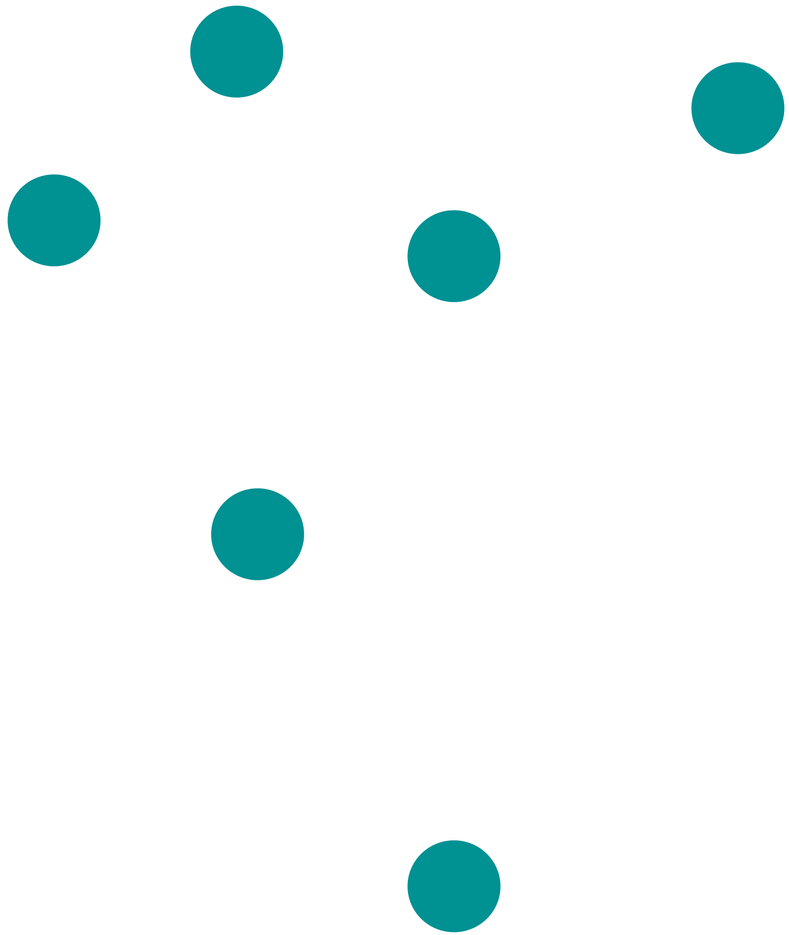
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

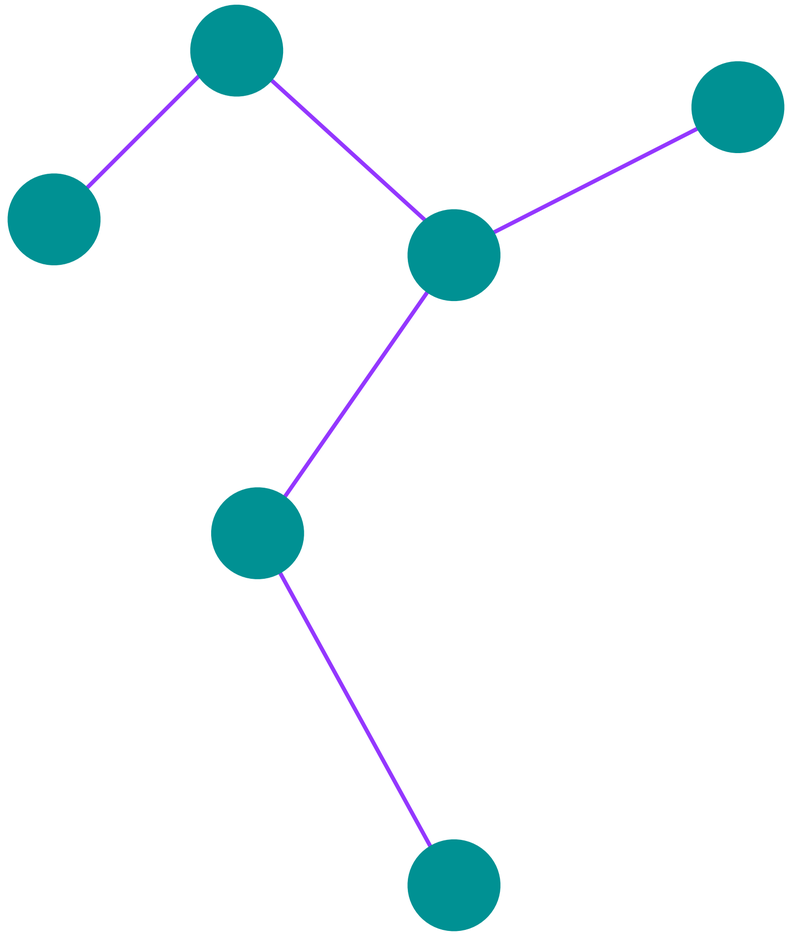
Metric TSP : 2-Approximation

Correctness



Metric TSP : 2-Approximation

Correctness

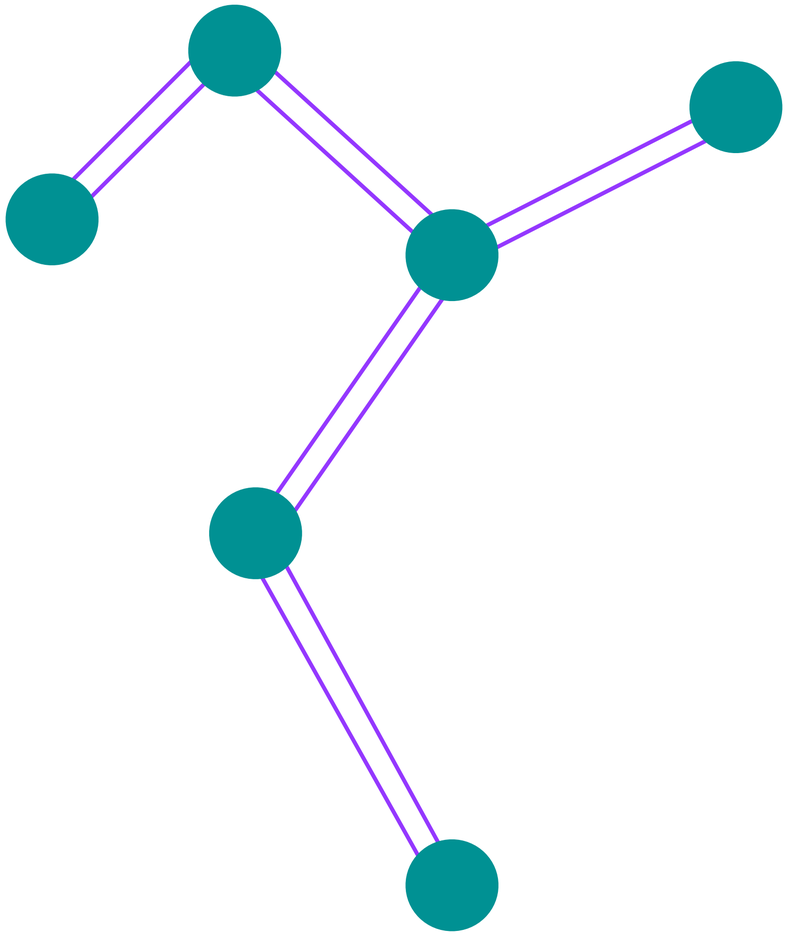


1. Find the MST T

$$l(T) \leq OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness

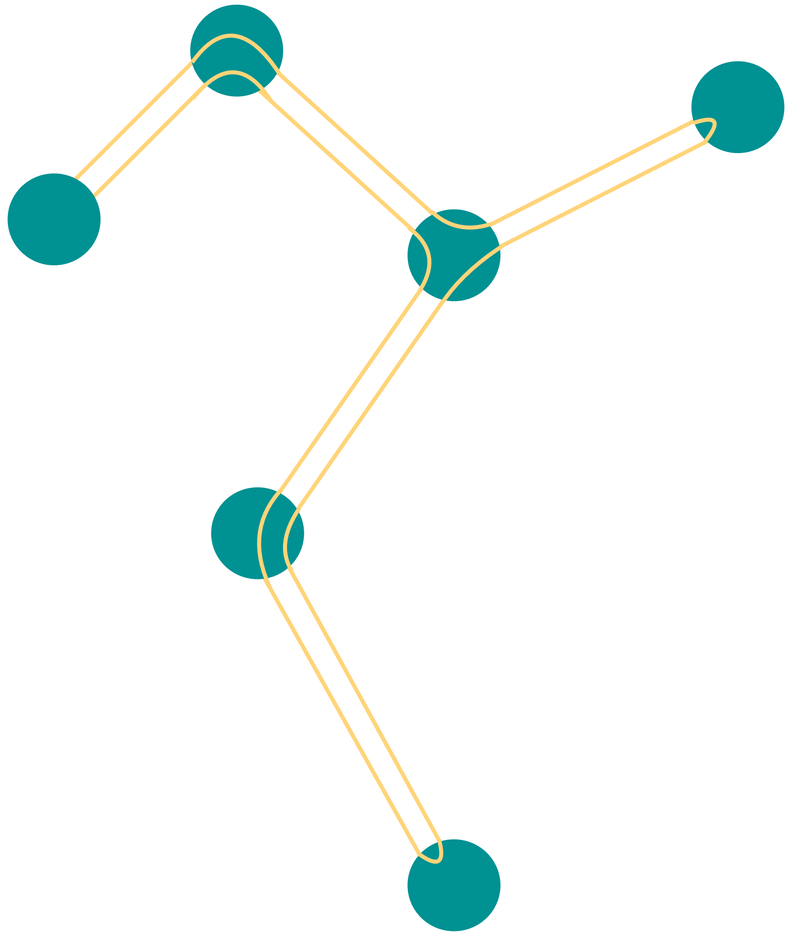


1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

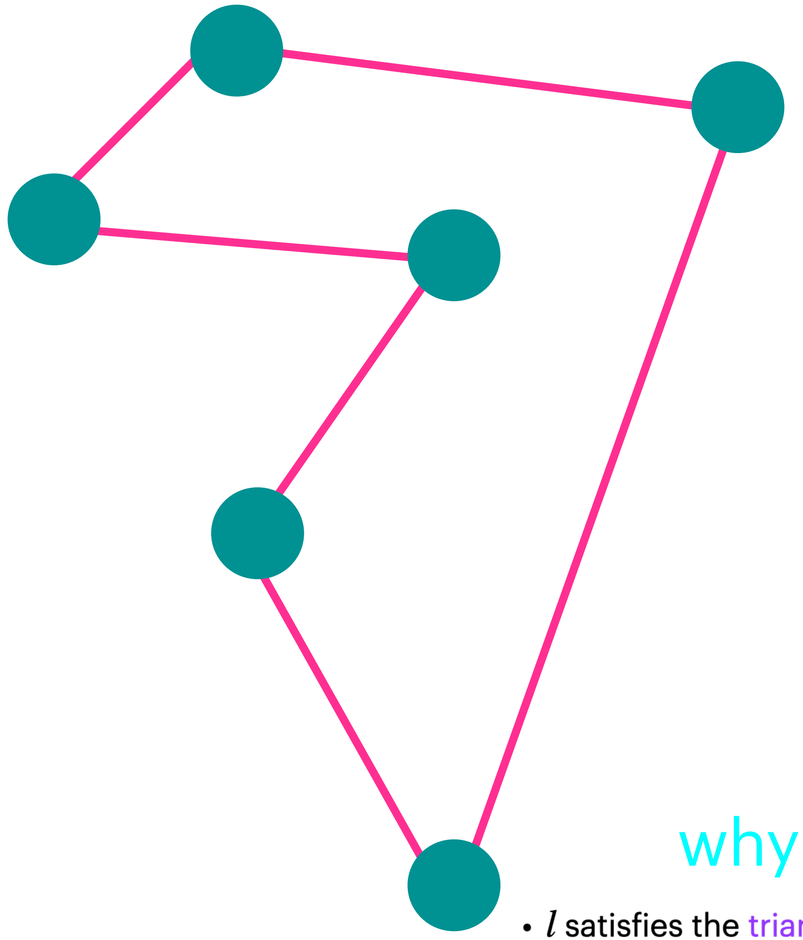
2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

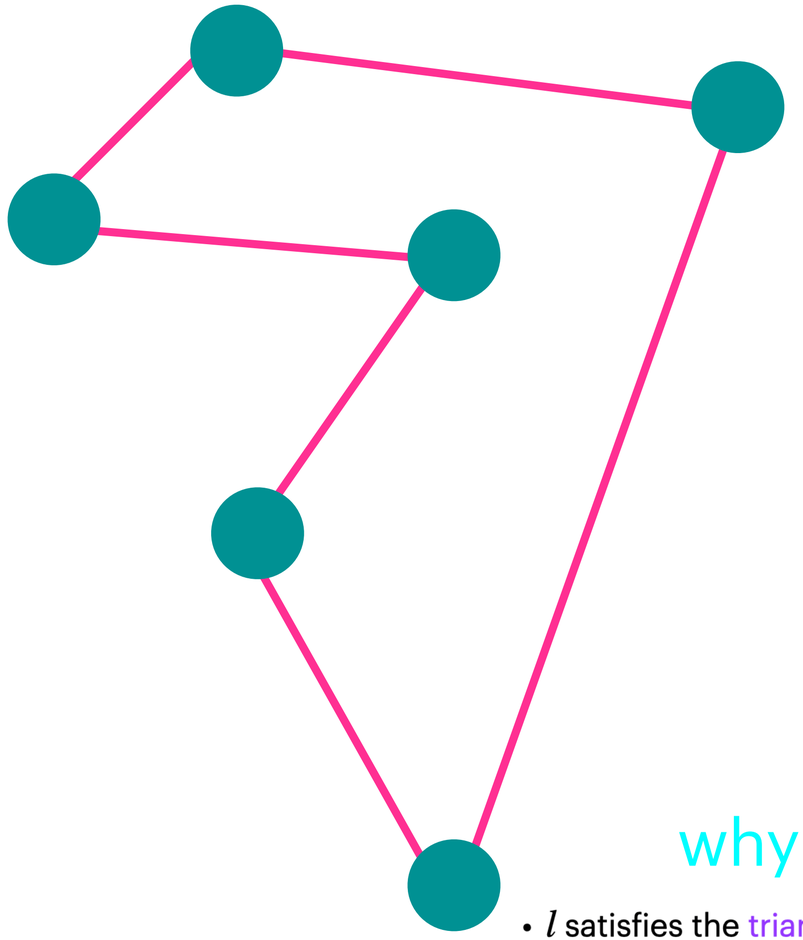
- l satisfies the triangle inequality

$$l(x, z) \leq l(x, y) + l(y, z)$$

$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality

$$l(x, z) \leq l(x, y) + l(y, z)$$

$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness

Goal : **Metric TSP : 2-Approximation**

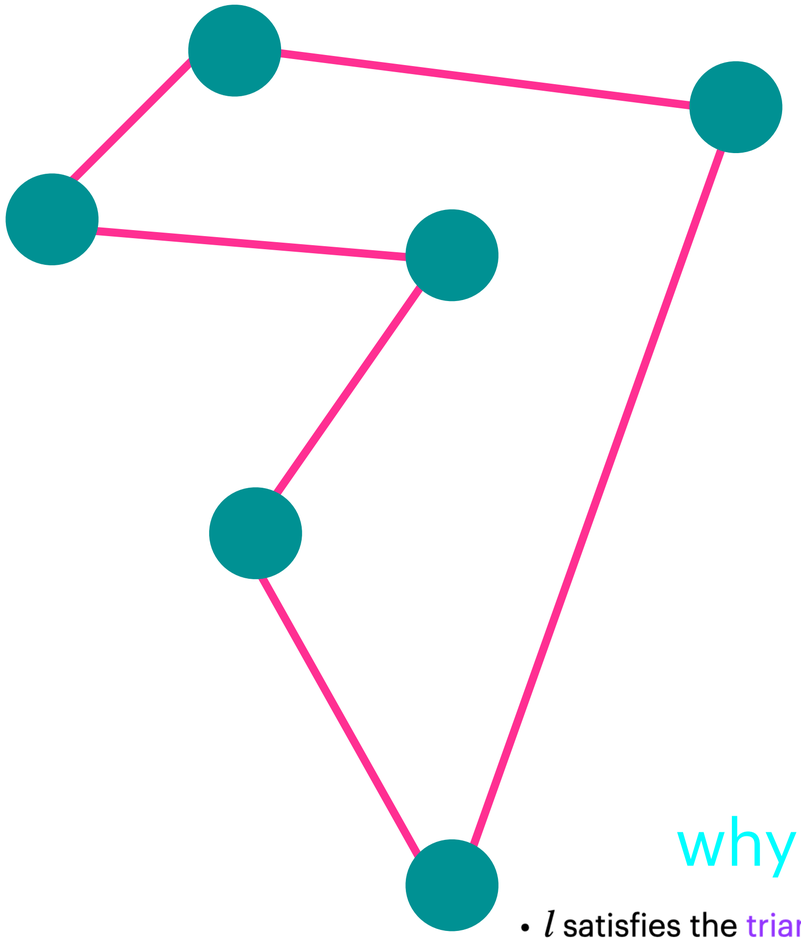
Problem Description

Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t.
  - l satisfies the triangle inequality $l(x,z) \le l(x,y) + l(y,z)$

$$l(C) \le 2\, l(OPT)$$

where $OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$



1. Find the MST T $\quad l(T) \le OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\, l(T) \le 2\, OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\, l(T) \le 2\, OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality

$$l(x,z) \le l(x,y) + l(y,z)$$

$$l(C) \le l(W) = 2\, l(T) \le 2\, OPT(K_n, l)$$

A&W

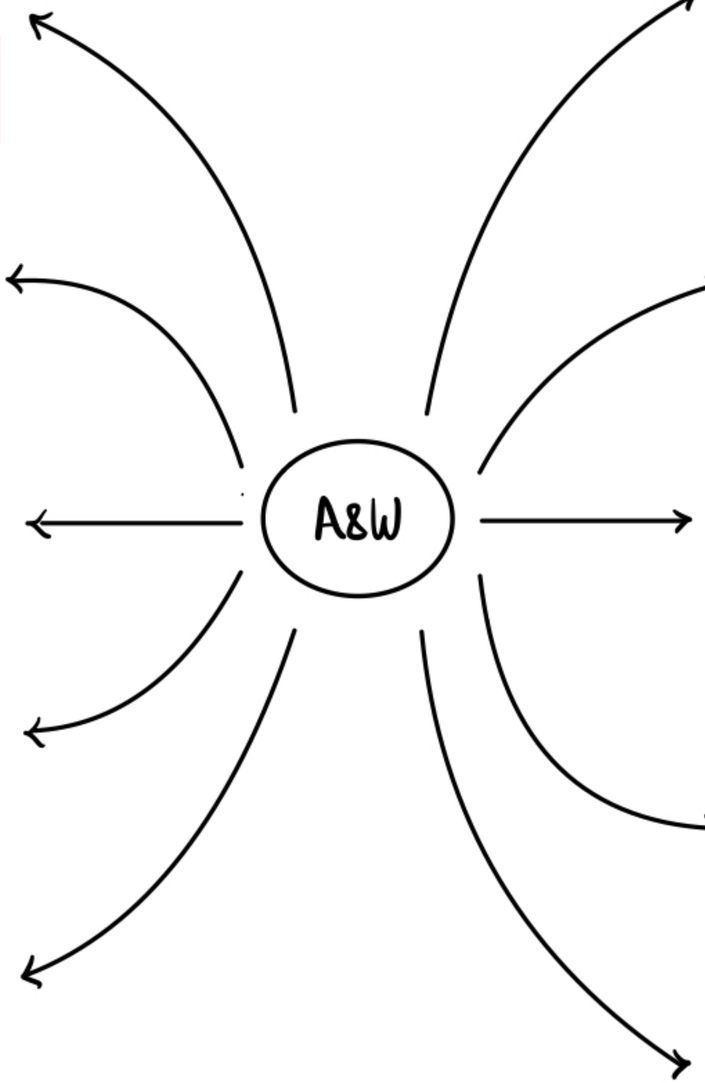
Exercise Session 4

Matching, TSP II

Nil Ozer

Minitest II

A&W Overview



Connectivity

- Articulation Points
- Menger's Theorem
- Bridges
- Block-Decomposition

Cycles

- Closed Eulerian Walk
- TSP
- Hamiltonian Cycle

Matchings

- Definition
- Hall's Theorem
- Algorithms

Colorings

- Definition
- Brooks's Theorem
- Algorithm

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
  - Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow-Mincut
- Ford-Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- Minitest II
- Minitest II Discussion

- Matching
- TSP II

Matching

Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

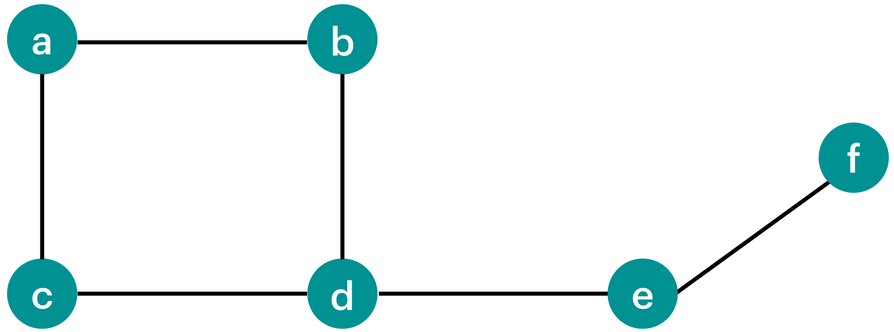
Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



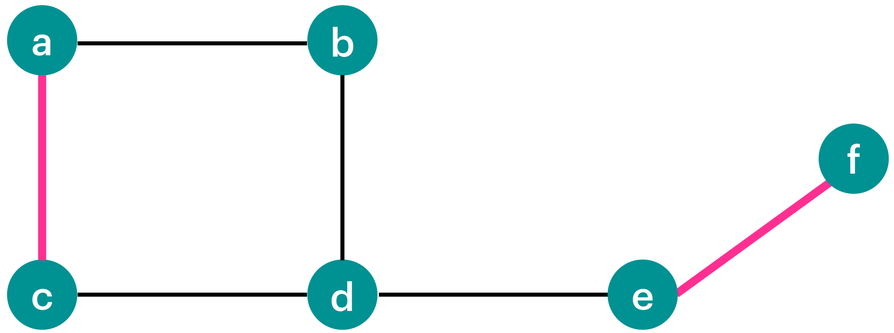
Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



M_1 = {{a,c} , {e,f}}

✅

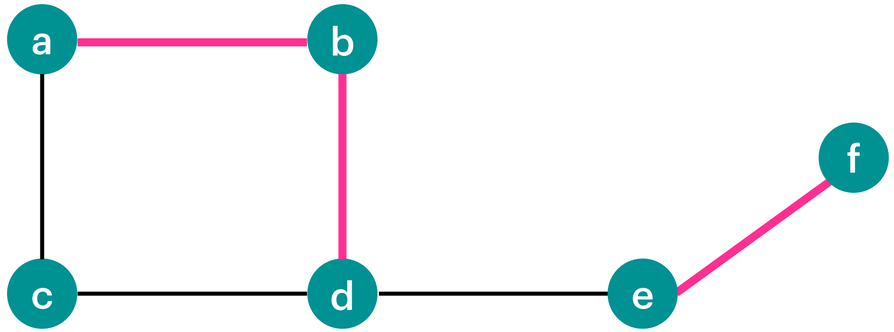
Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?

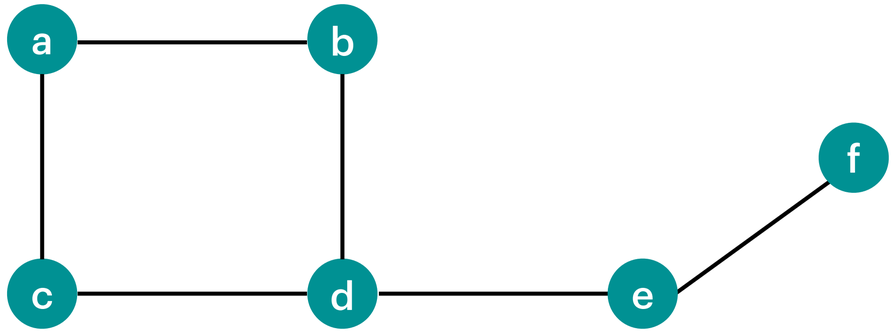


Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices



Matching

Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

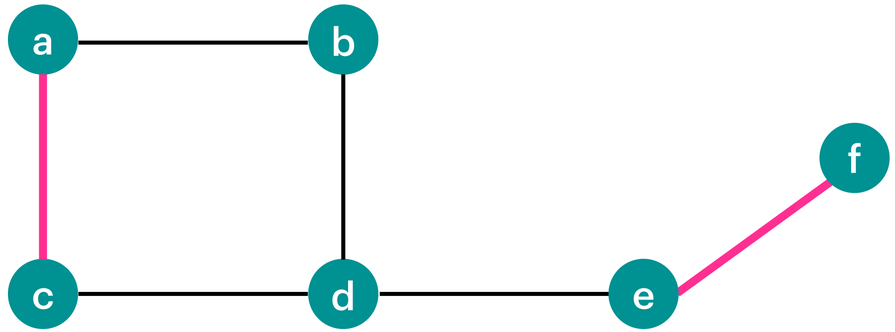
no two edges share common vertices

- covered (matched) :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

Matching

Definitions

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- covered (matched) :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v



Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching :
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M
  - equivalently, if

$$M = \frac{|V|}{2}$$

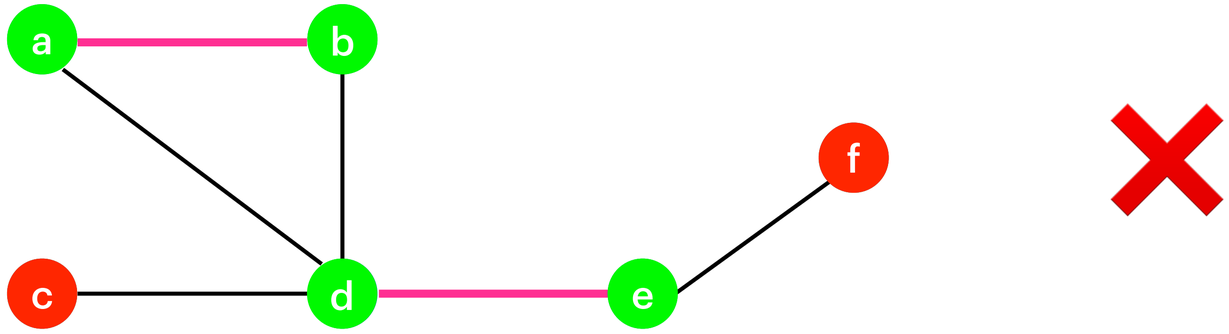
Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?



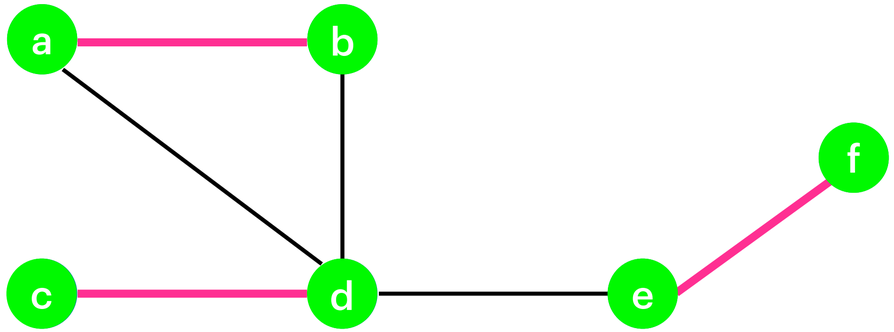
Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?



✅

Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?

Matching

Definitions

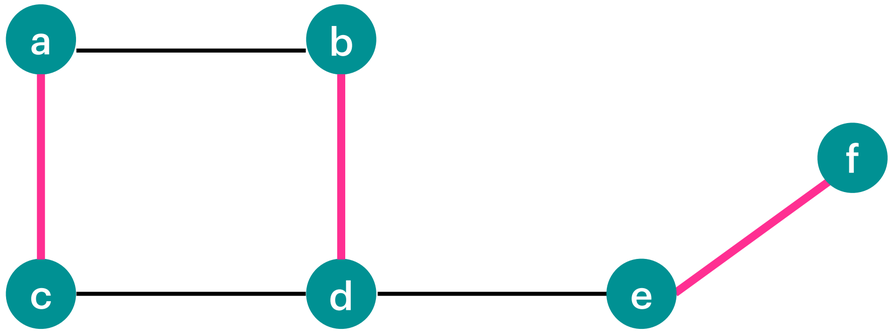
- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Matching

Definitions

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ?



Is this inclusion-maximal ?

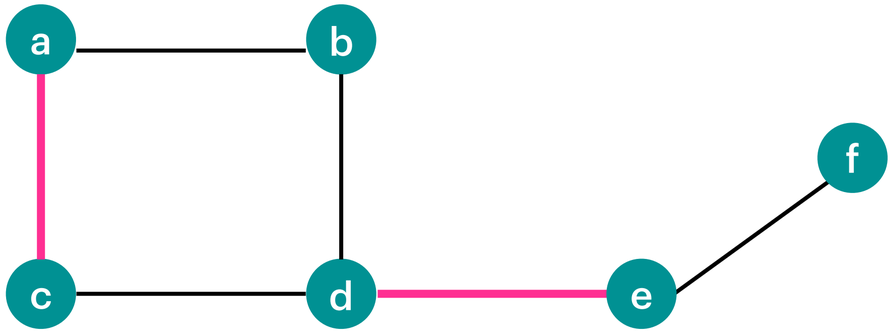
Matching

Definitions

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ?

❌



Is this inclusion-maximal ?

✅

Matching

Propositions

- M_{inc} : inclusion-maximal Matching , M_{max} : cardinality-maximum Matching

$$|M_{inc}| \leq |M_{max}|$$

$$|M_{inc}| \geq |M_{max}| \ / \ 2$$

Why ?

Every edge in M_{max} must have at least one endpoint in M_{inc}

Otherwise, that edge would be added to M_{inc}

$$|M_{max}| \leq |Endpoints\ in\ M_{inc}| = 2\,|M_{inc}|$$

- M_{inc} : inclusion-maximal Matching , M_{max} : cardinality-maximum Matching

Matching

Greedy Algorithm

$$|M_{inc}| \leq |M_{max}|$$

$$|M_{inc}| \geq |M_{max}| \,/\, 2$$

GREEDY-MATCHING (G)

```
1: M ← ∅
2: while E ≠ ∅ do
3:     pick an arbitrary edge e ∈ E
4:     M ← M ∪ {e}
5:     remove e and all incident edges in G
```

$$|M_{Greedy}| \geq |M_{max}| \,/\, 2$$

in $O(|E|)$

why ?

M_{Greedy} is inclusion-maximal

Matching

Augmenting ?

augment 1 of 2 **verb**

aug·ment ȯg-ˈment

augmented; augmenting; augments

Synonyms of *augment* >

transitive verb

1 : to make greater, more numerous, larger, or more intense

> The impact of the report was *augmented* by its timing.

Synonyms & Similar Words

increase	expand	accelerate
boost	enhance	extend
raise	multiply	reinforce
amplify	intensify	maximize
strengthen	add (to)	enlarge
aggrandize	swell	escalate
stoke	compound	build up
supplement	pump up	up
heighten	complement	supersize
develop	prolong	lengthen
hype	inflate	skyrocket
dilate	distend	elongate

Matching

M - Augmenting Path

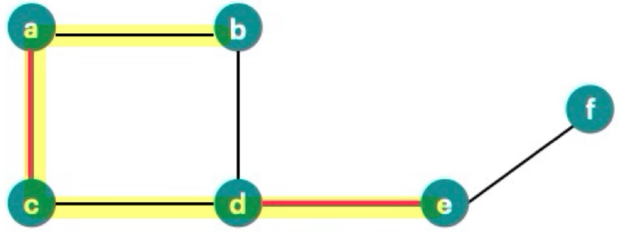
- Augmenting Path : “path with edges not in M, in M, … , not in M “
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching

Matching

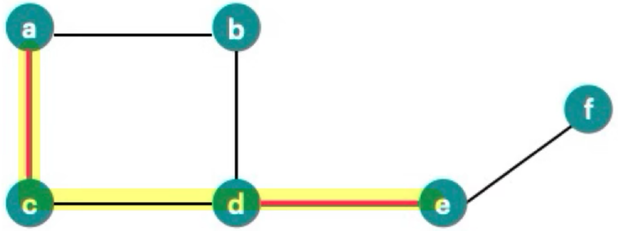
M - Augmenting Path

Is this an augmenting path ?

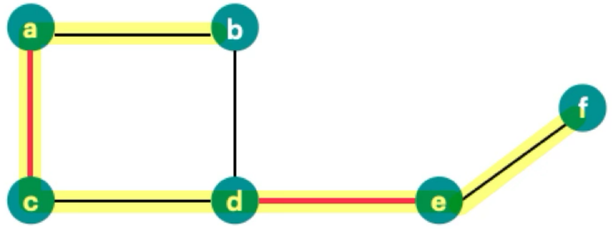
- Augmenting Path : "path with edges not in M, in M, … , not in M "
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching



— Matching M ❌

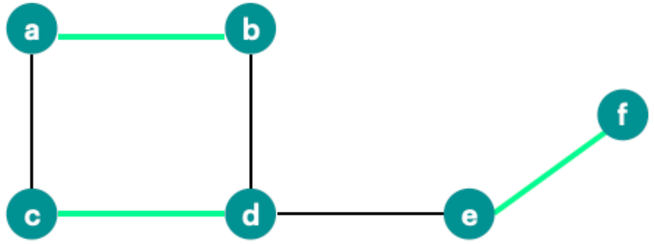


— Matching M ❌



— Matching M ✅

Idea : By swapping along M we can improve the matching



— Matching M'

Matching

Swapping ?

$$A \oplus B$$

Elements that are in A or in B but not in both

$$A = \{1,2,3\}$$

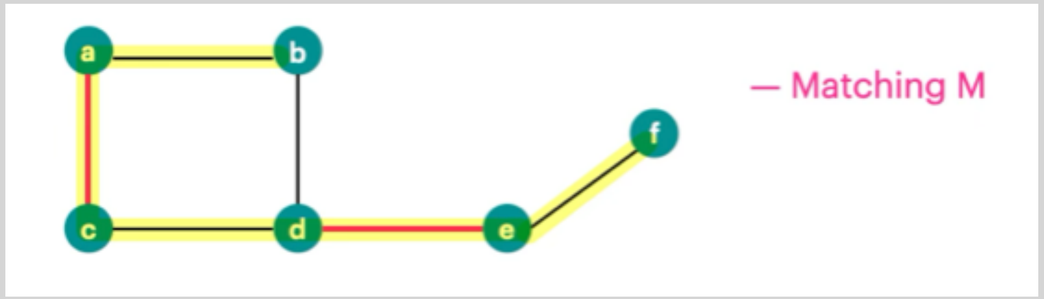
$$B = \{3,4,5\}$$

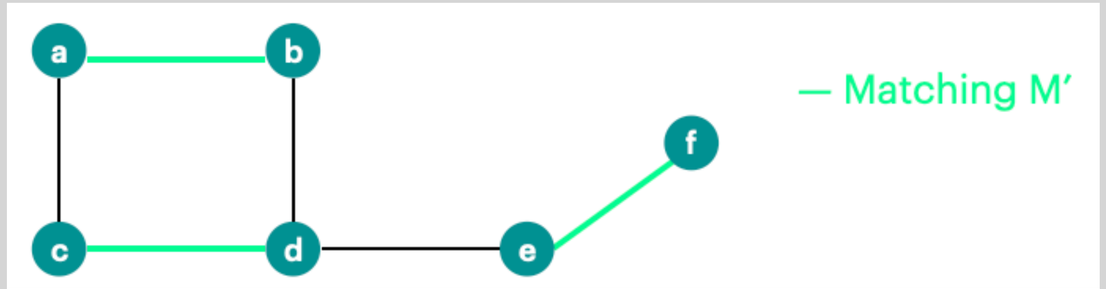
$$A \oplus B = \{1,2,4,5\}$$

Matching

Swapping ?

$$M' := M \oplus P$$





Matching

Berge's Theorem

- Augmenting Path : "path with edges not in M, in M, ... , not in M "
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching

A Matching M is (cardinality-) maximum $\iff$ There's no M-augmenting path

Idea : To find the maximum matching, update/improve the matching until there is no augmenting path left

Matching

Algorithm

Idea : Update/improve the matching until there is no augmenting path left

Input : $G = (V, E)$

Output : maximum matching M

Algorithm :

Start with $M = \emptyset$

while $\exists$ augmenting path P

$M = M \oplus P$

return M

How do we find the augmenting path P ?

bipartite Gs : with BFS

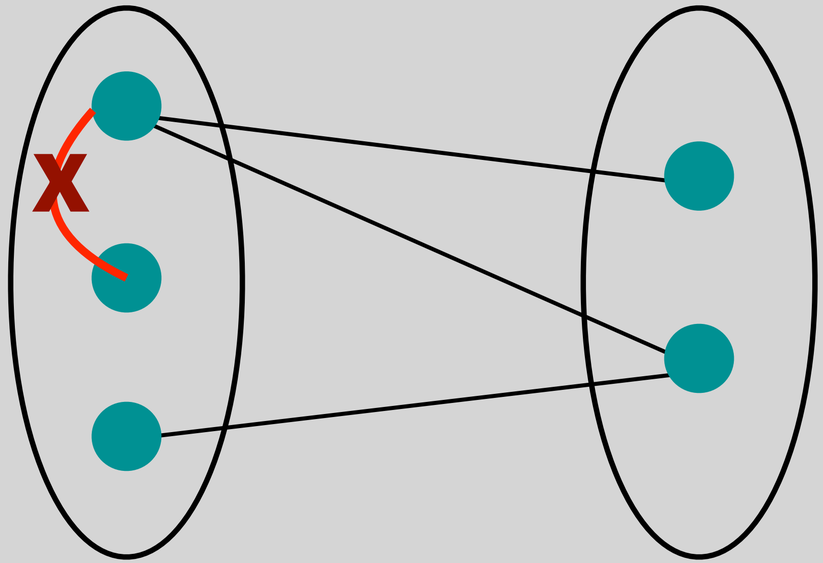
general Gs in $O(|V||E|)$

Matching

Definitions

- Bipartite Graph :
  - A graph G is bipartite , if you can split the set of vertices V into two sets U, V s.t. :

$$E \subseteq \{ \{u, v\} \ : u \in U, v \in V\}$$



Matching

Definitions

- k-regular
  - A graph G is k-regular , if every vertex has a degree of k

$$deg(v) = k \quad \forall v \in V$$

Matching

Perfect Matching finding

bipartite ?	k-regular	runtime
✅	2^K	$O(\|E\|)$
✅	k	$O(\|E\|)$
✅	-	$O(\|V\| \cdot \|E\|)$

Matching

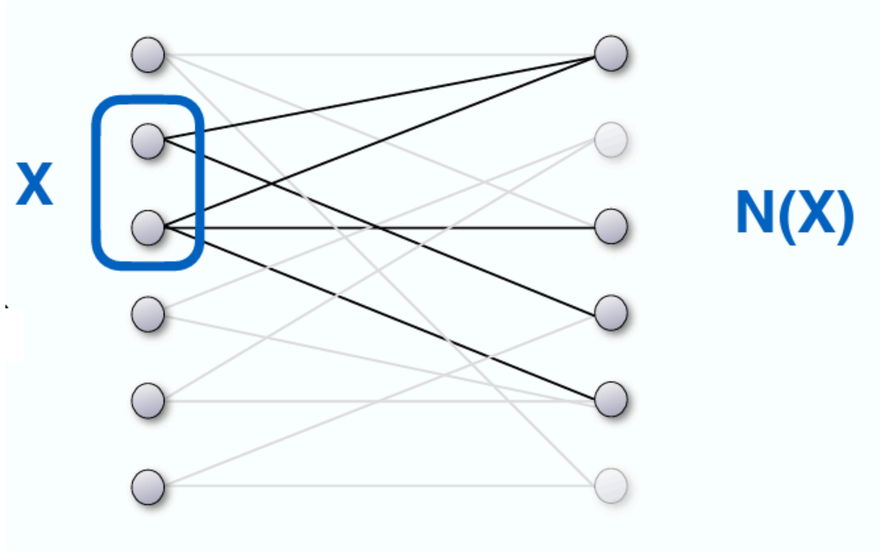
Hall's Marriage Theorem

A bipartite $G = (A \cup B, E)$

has a Matching M with cardinality $|M| = |A|$

$\Leftrightarrow$

$\forall X \subseteq A : |X| \leq |N(X)|$



N(X) := "neighbours of vertices in X"

Corollary : Every k-regular bipartite G has a perfect matching

Matching

Algorithm - revisit

Idea : Update/improve the matching until there is no augmenting path left

Input : $G = (V, E)$

Output : maximum matching M

Algorithm :

Start with $M = \emptyset$

while $\exists$ augmenting path P

$M = M \oplus P$

return M

How do we find the augmenting path P ?

bipartite Gs : with BFS

general Gs in $O(|V||E|)$

BFS + this -> O (|V| |E|)

Matching

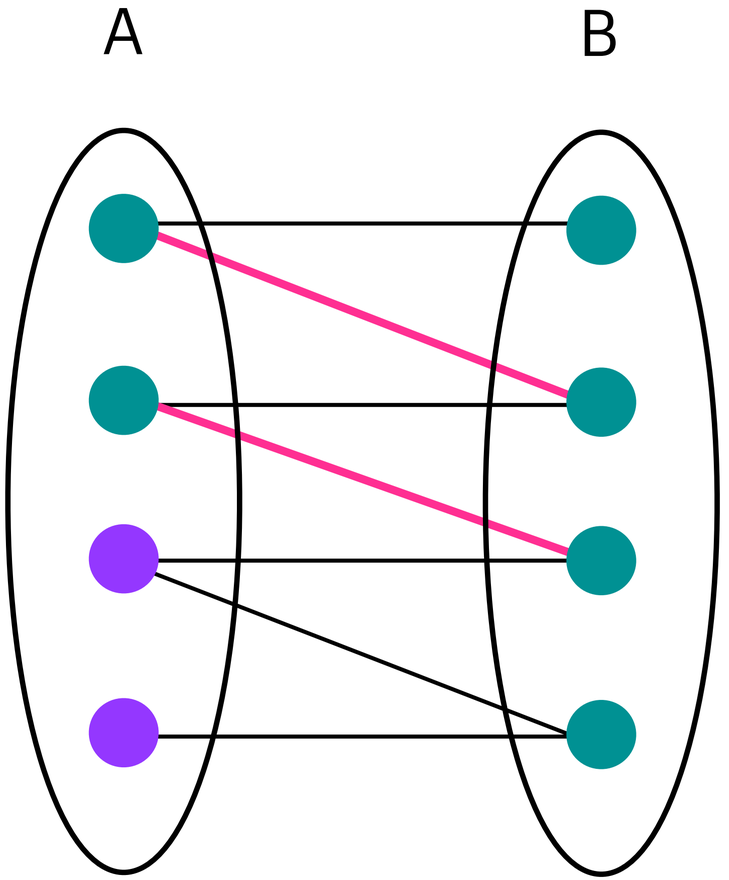
BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}



Matching

BFS for augmenting paths

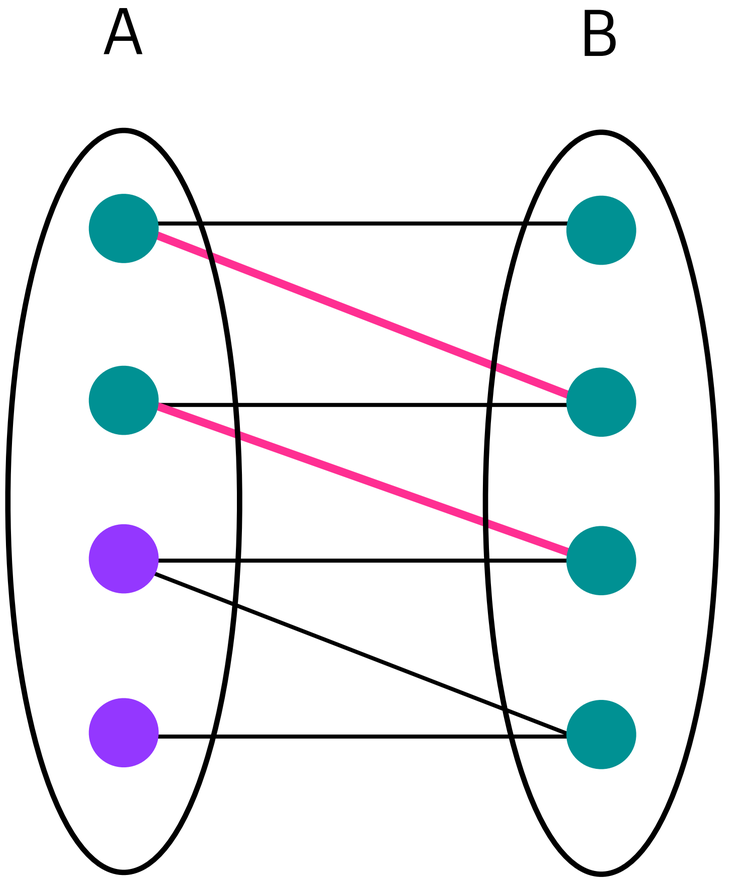
Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited



Matching

BFS for augmenting paths

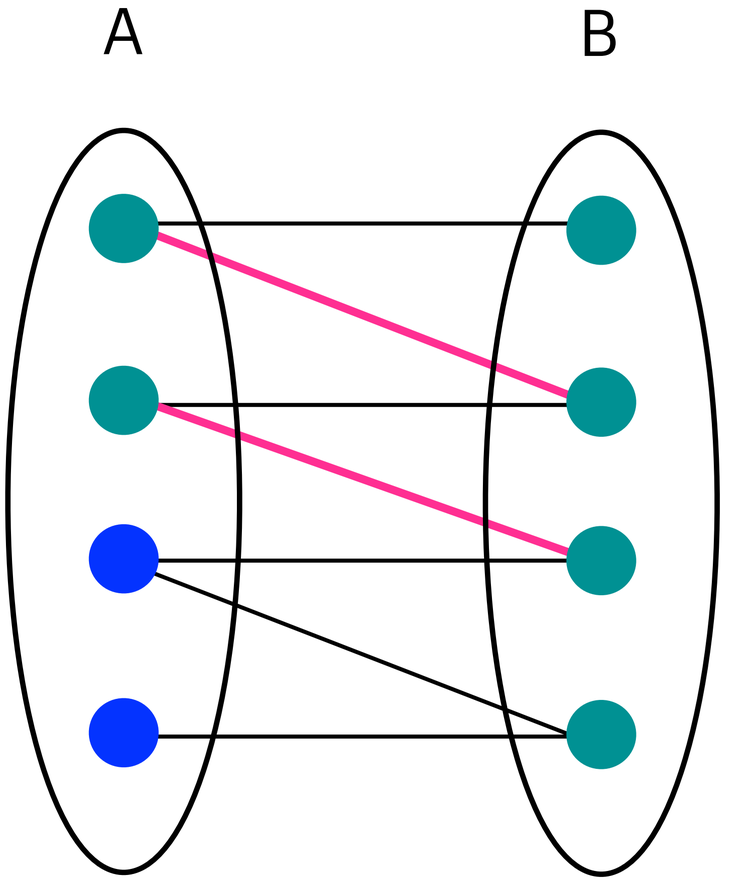
Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

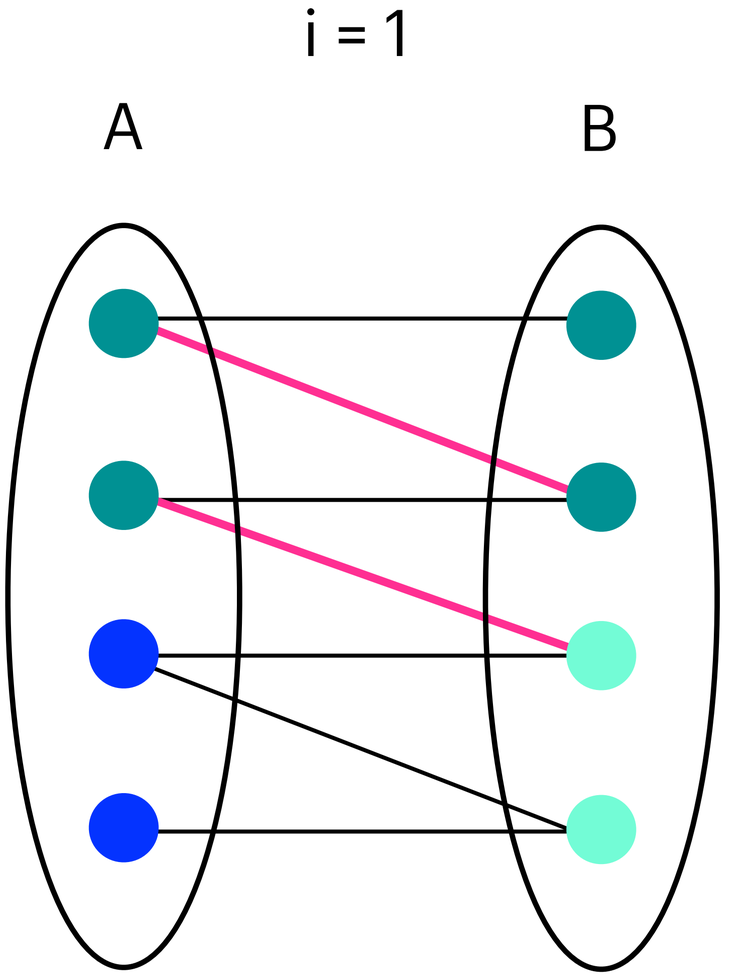
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

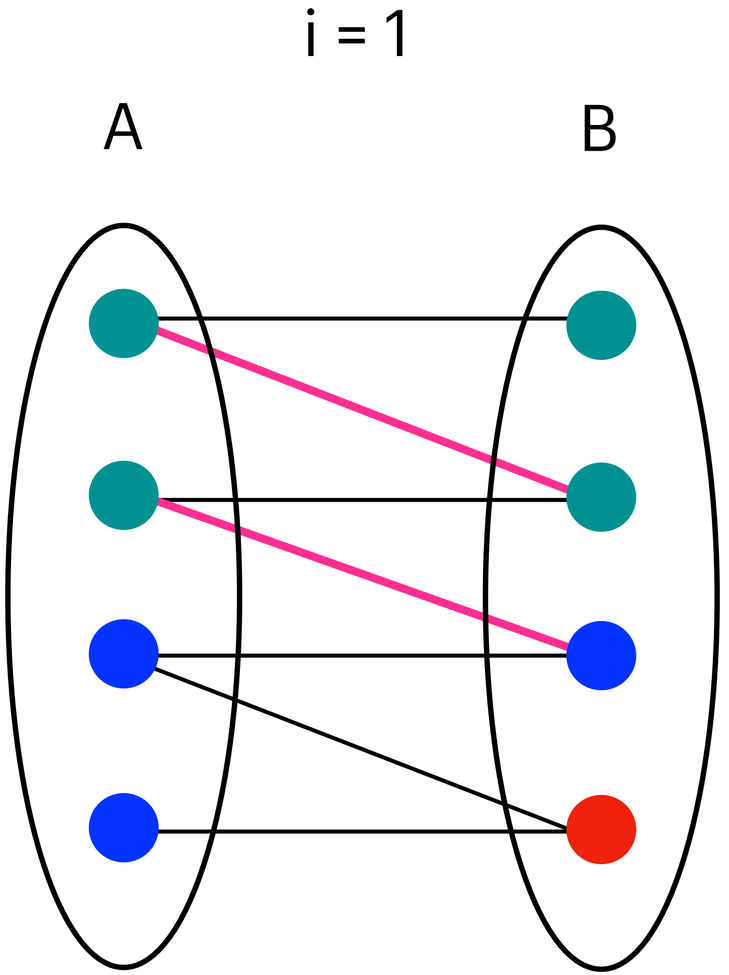
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

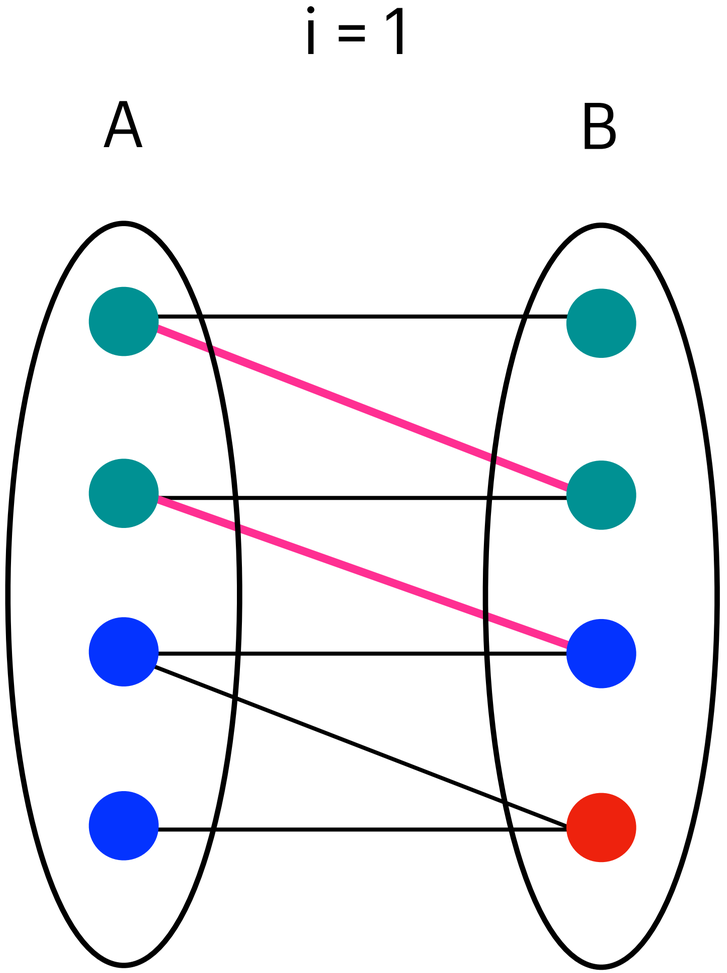
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

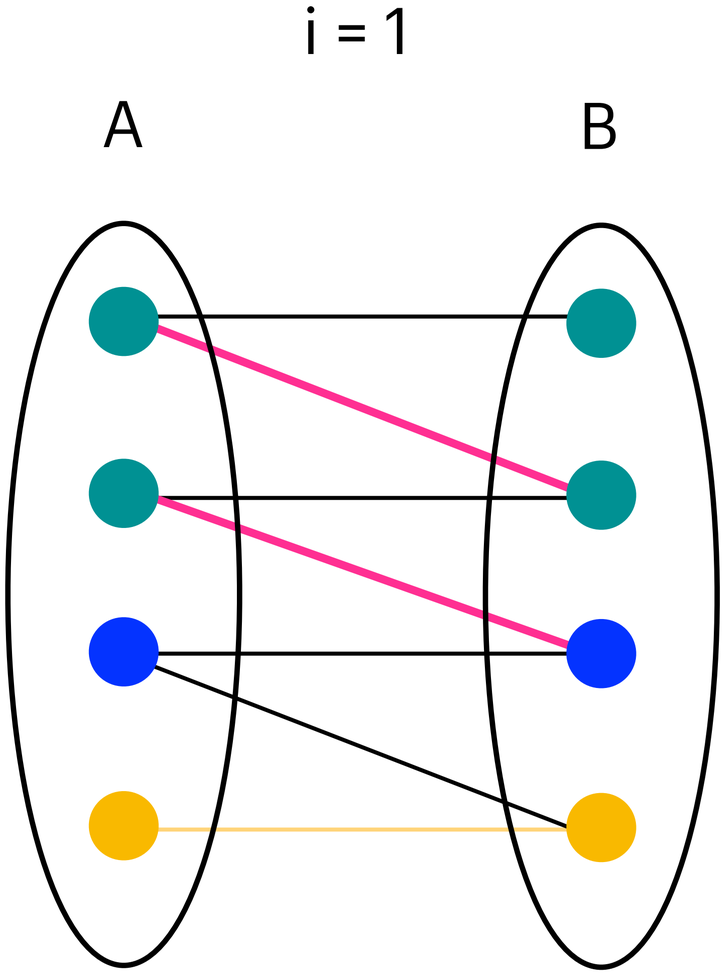
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

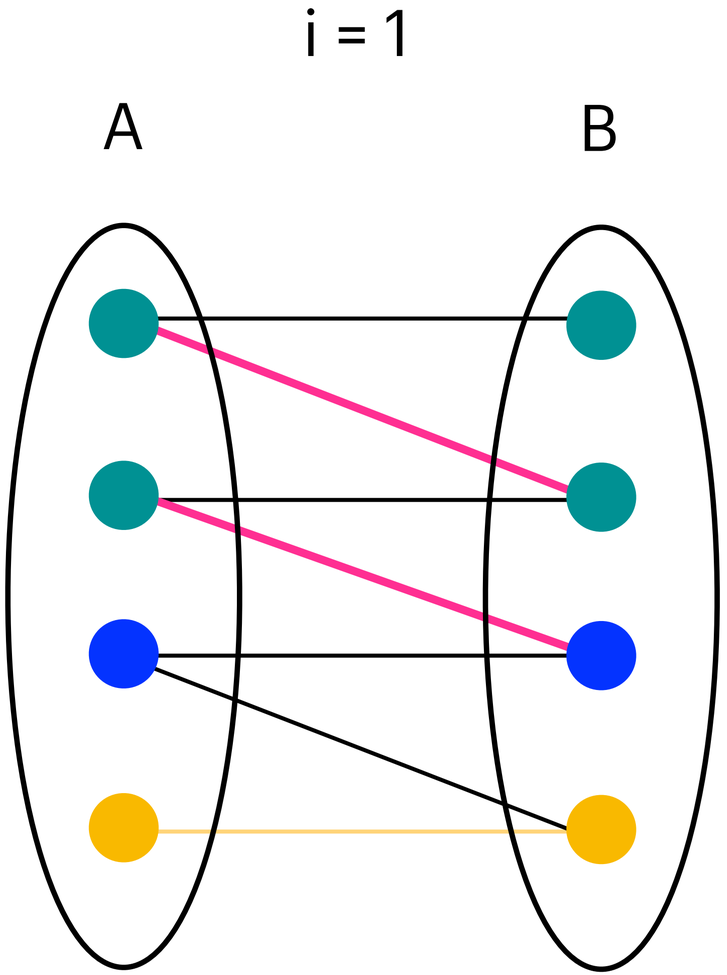
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

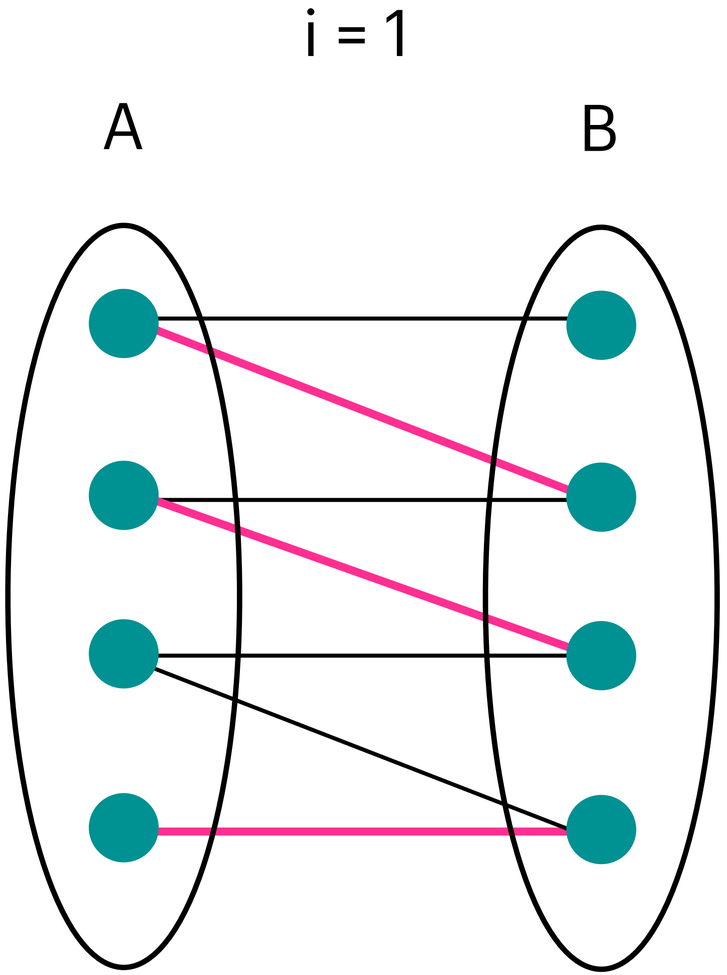
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

RESTART

if i is odd then

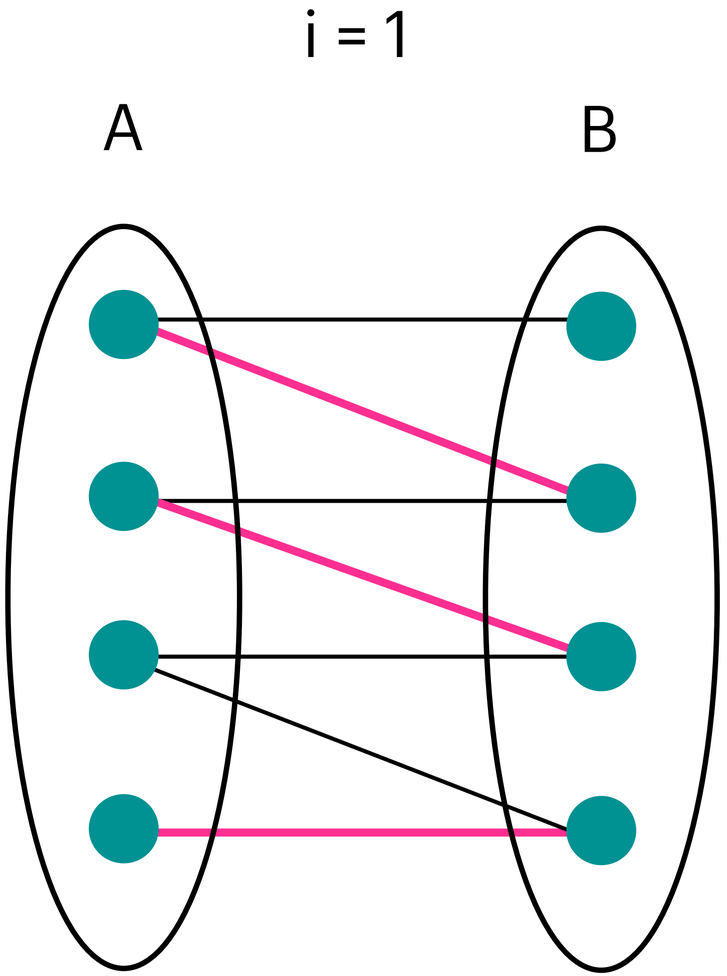
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

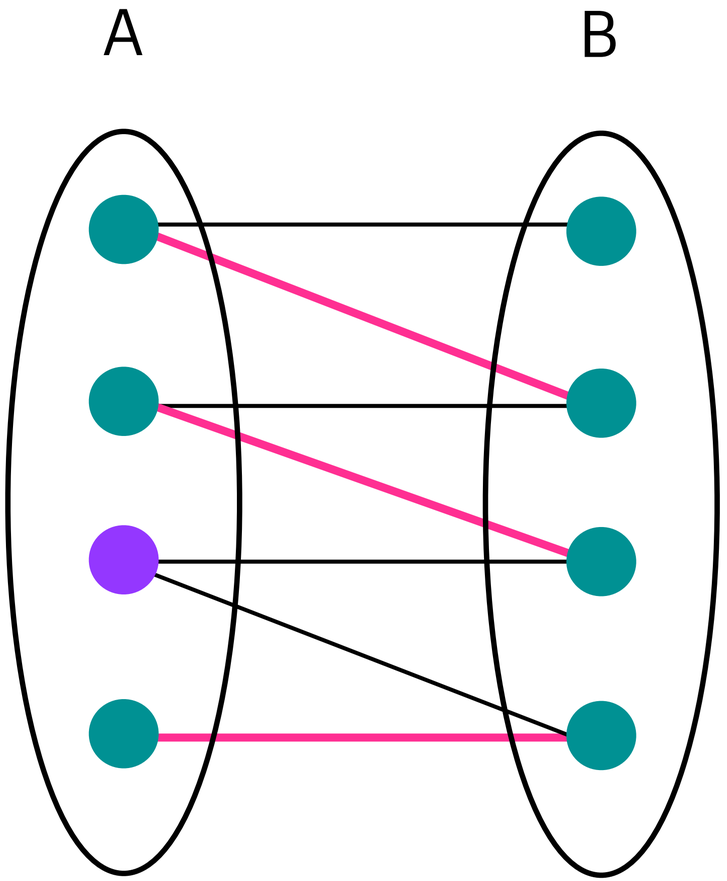
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

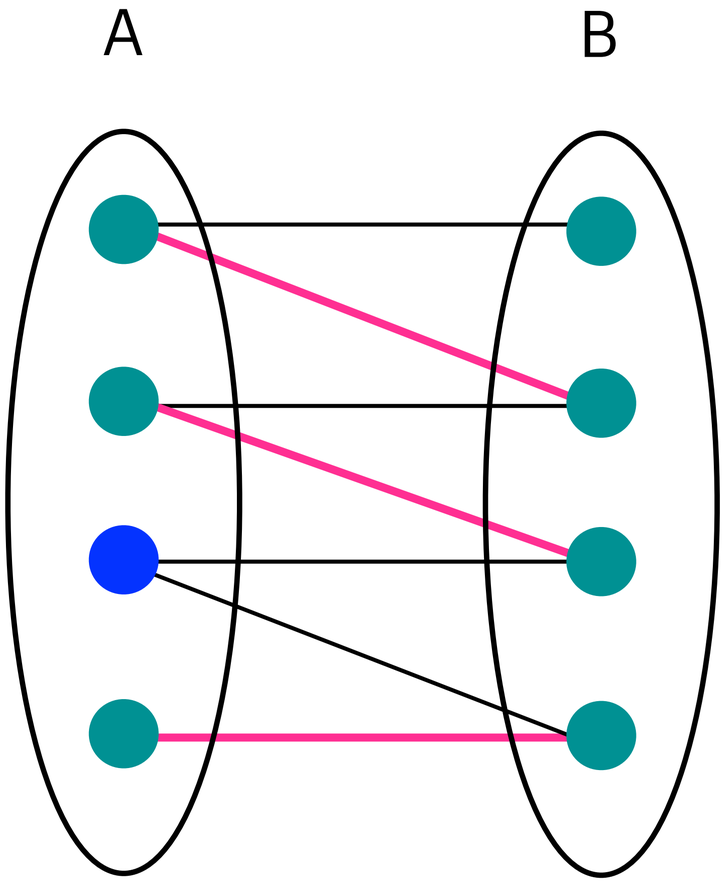
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

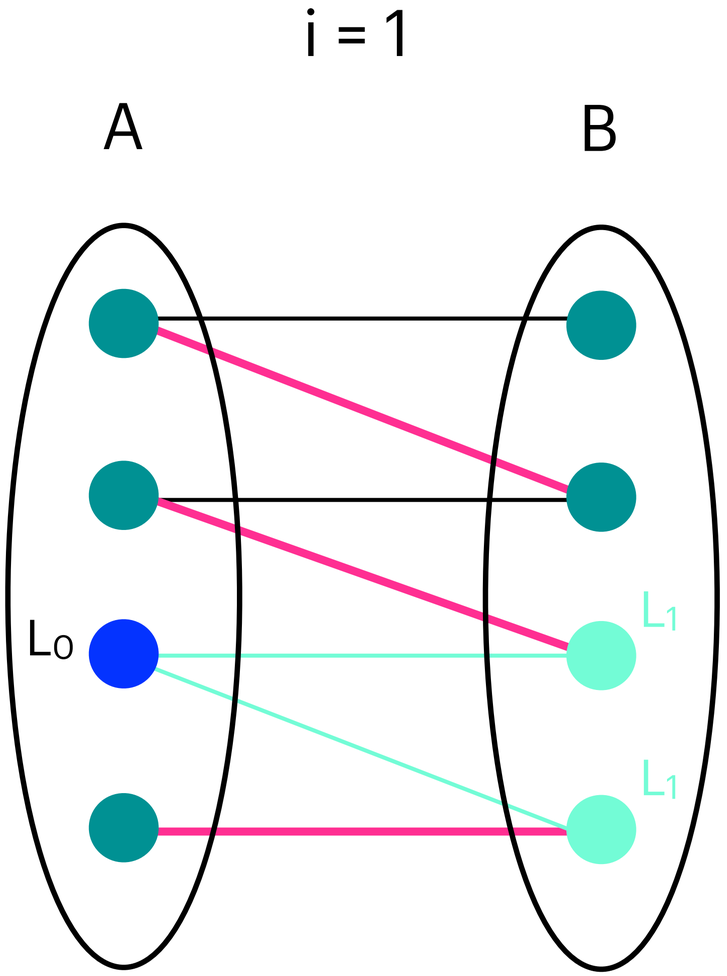
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

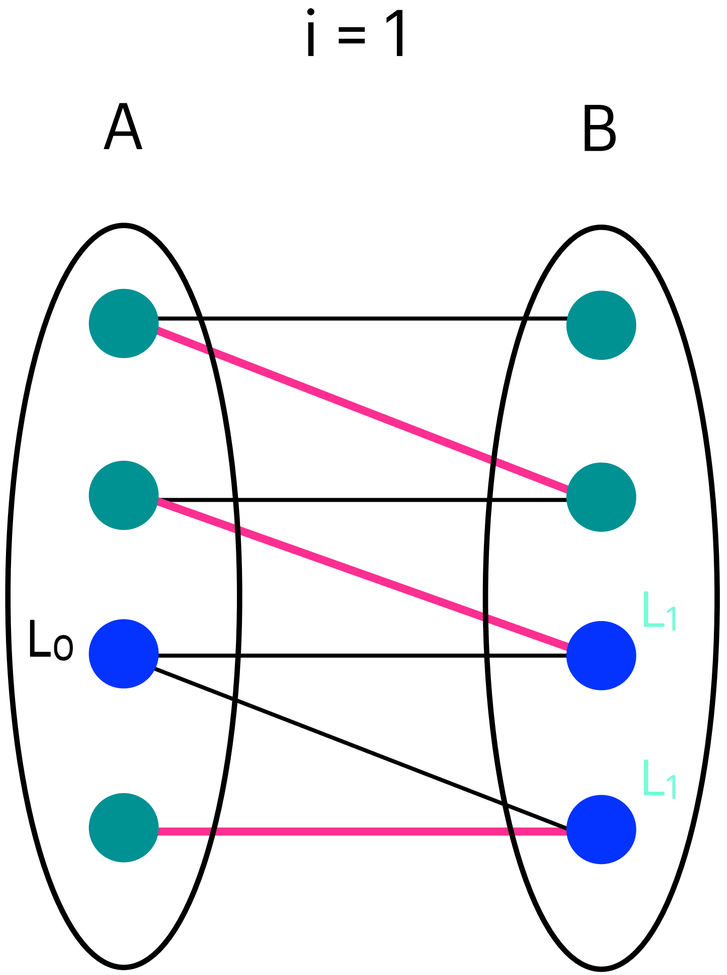
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

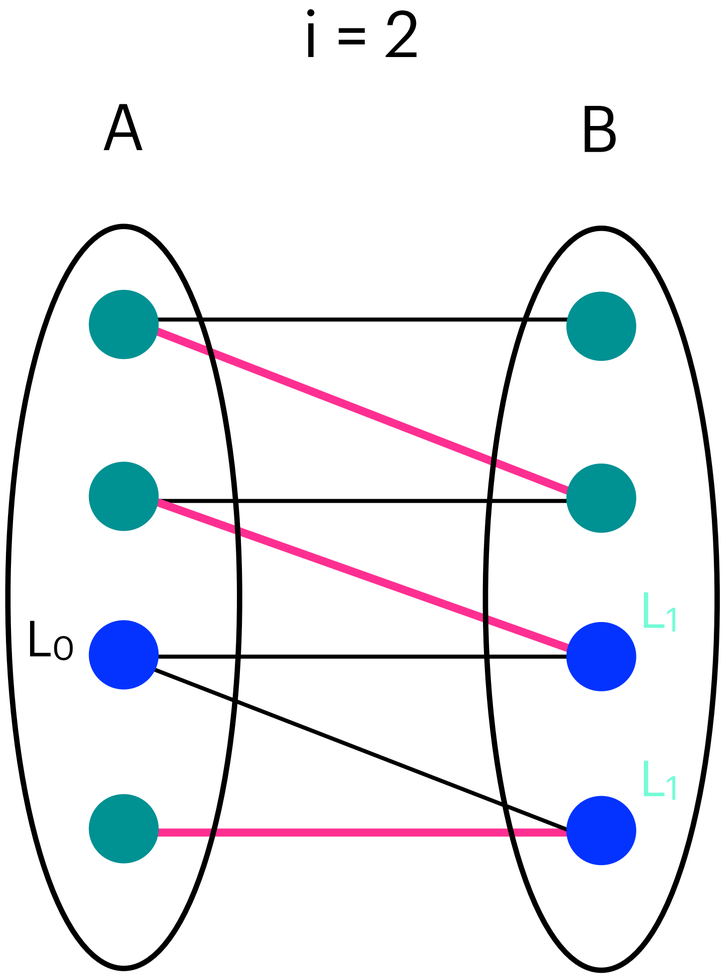
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

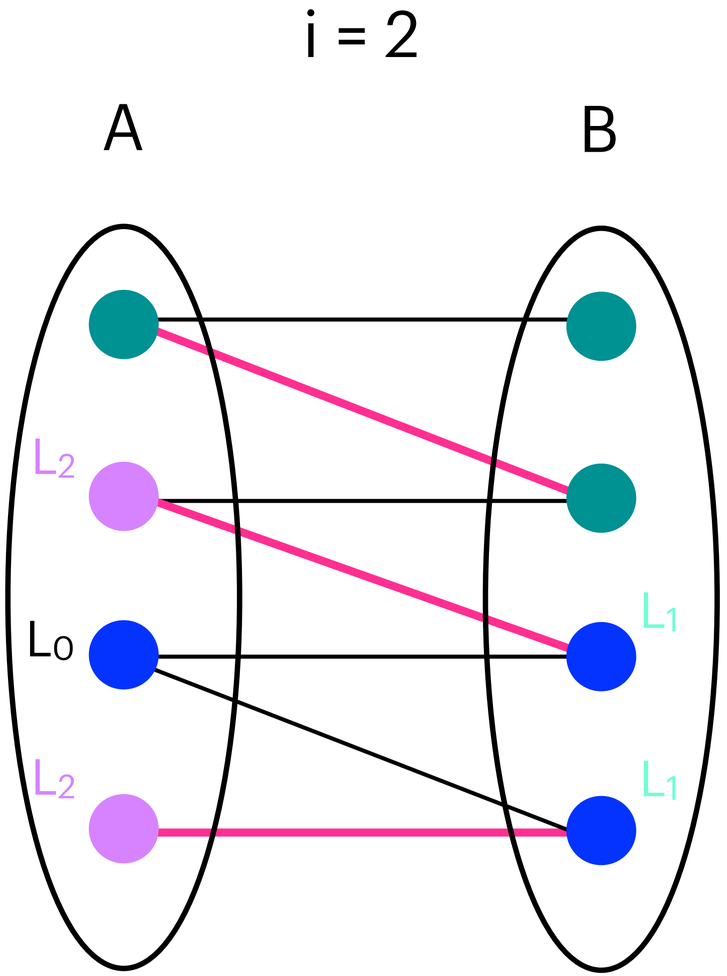
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

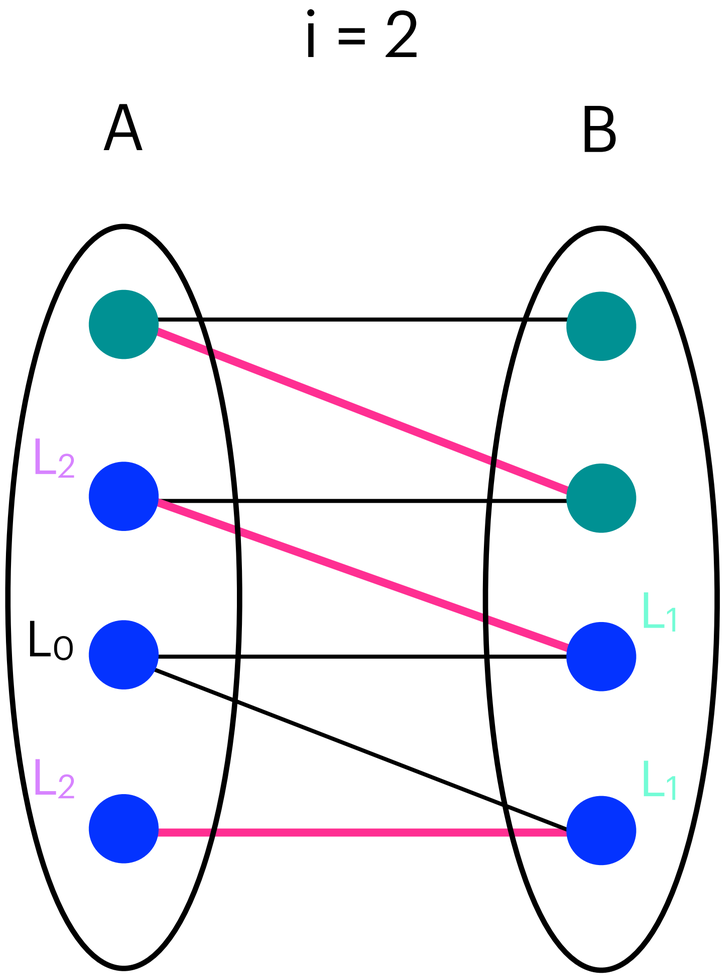
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

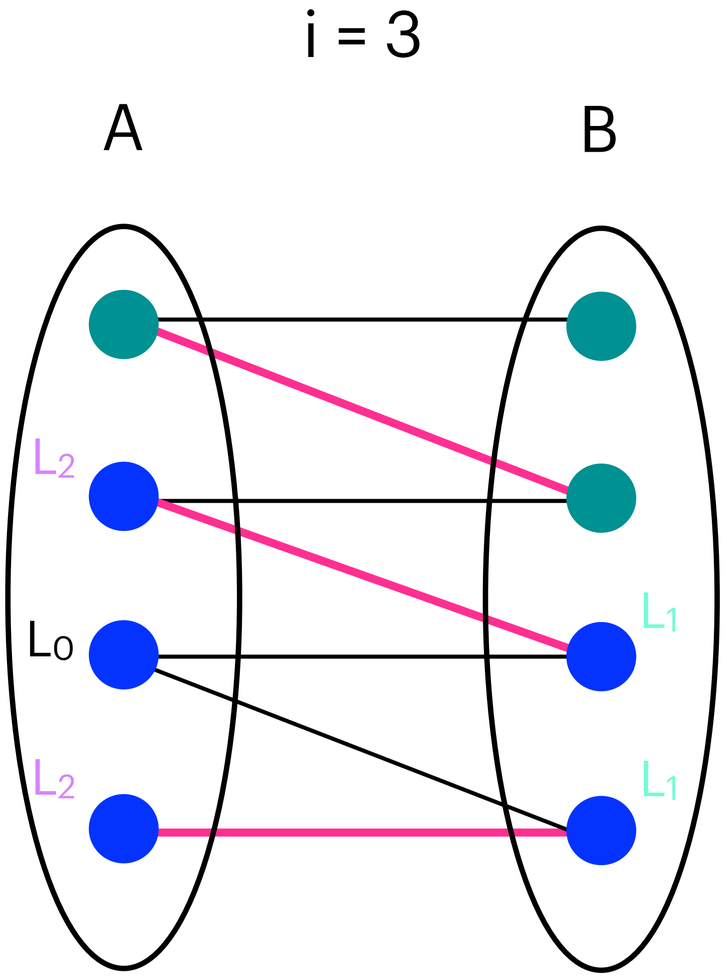
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

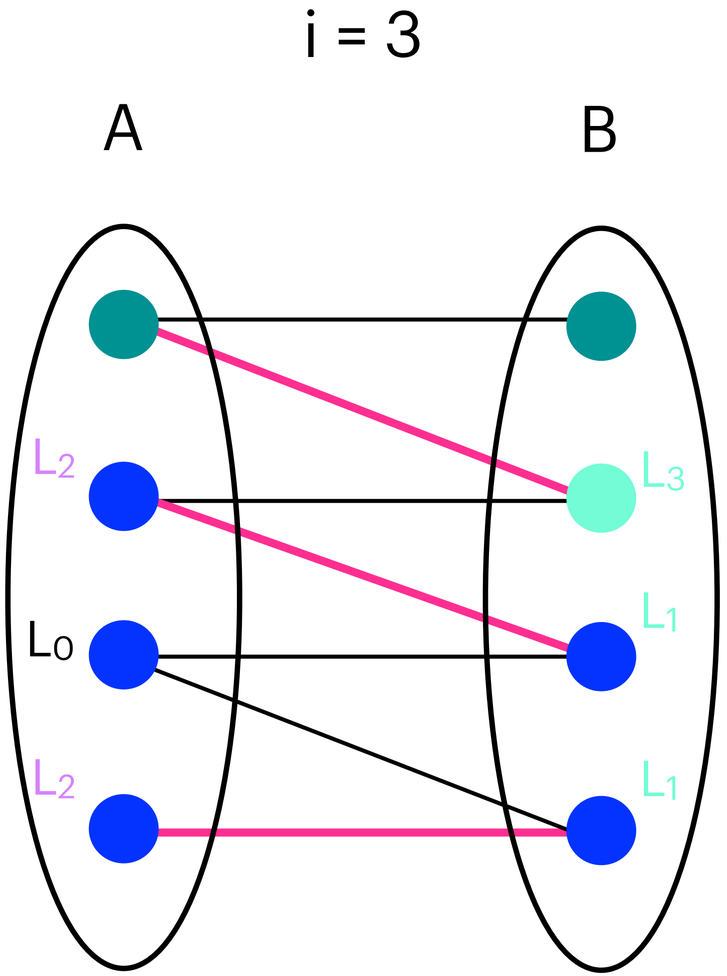
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

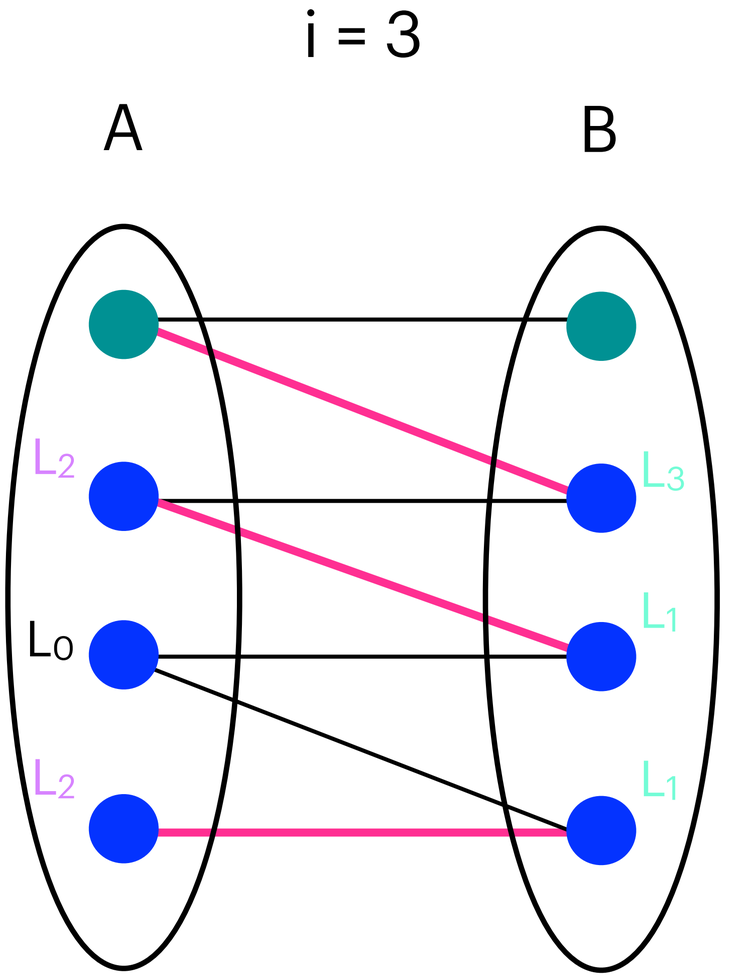
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

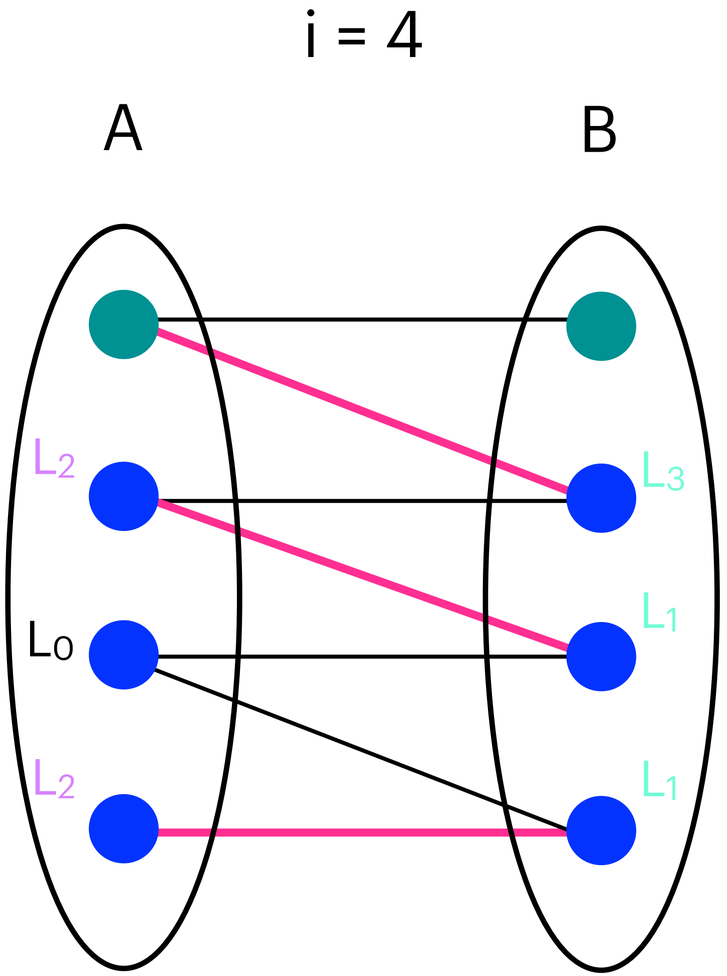
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

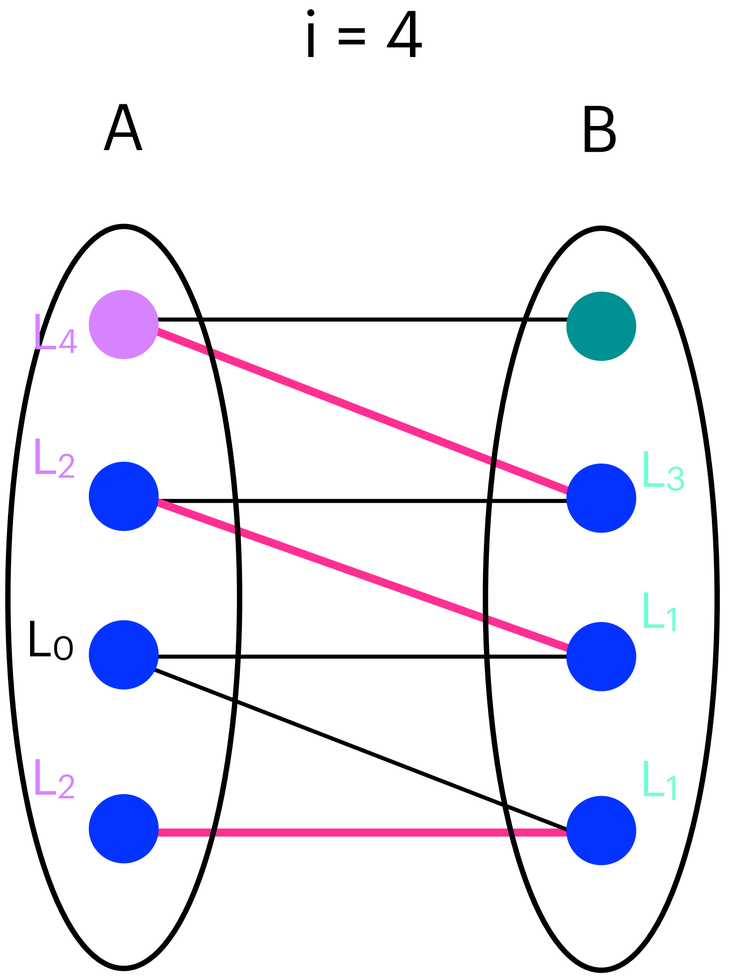
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

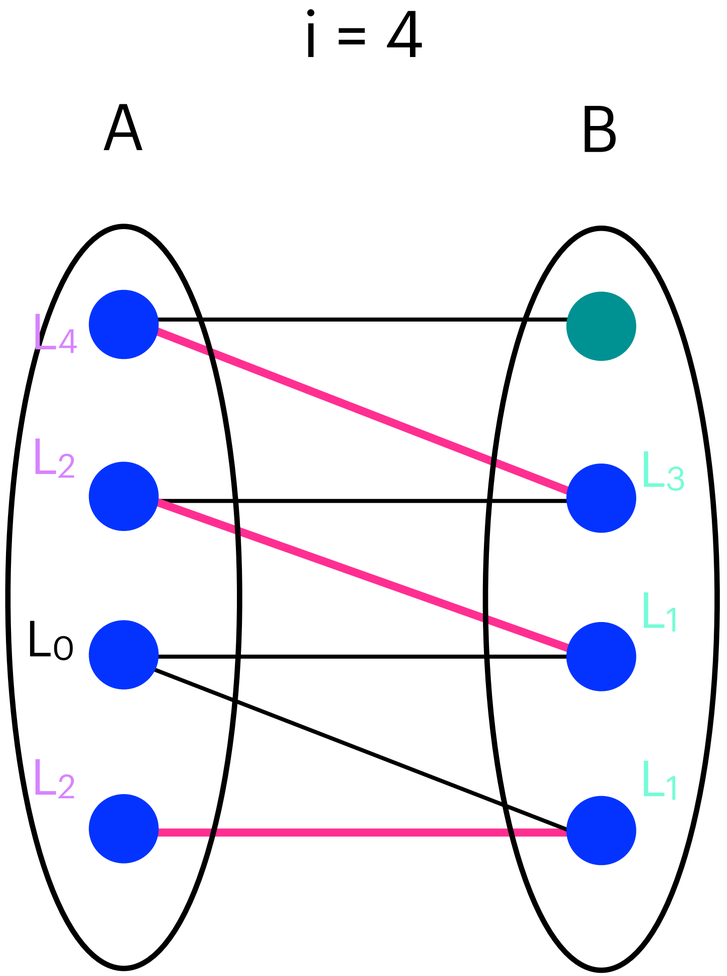
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

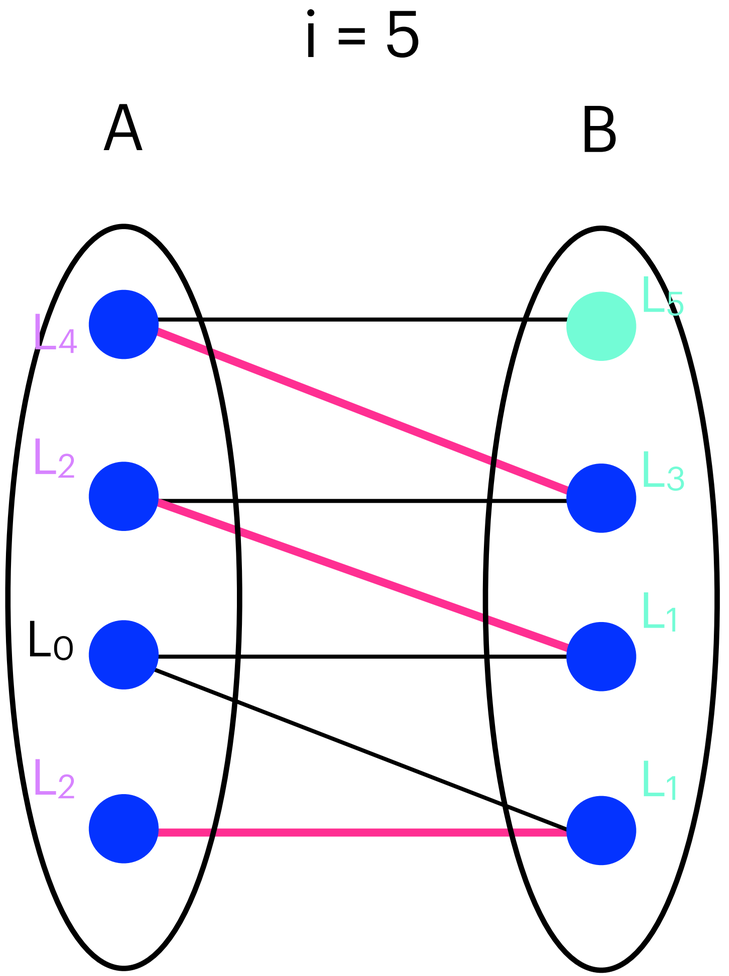
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

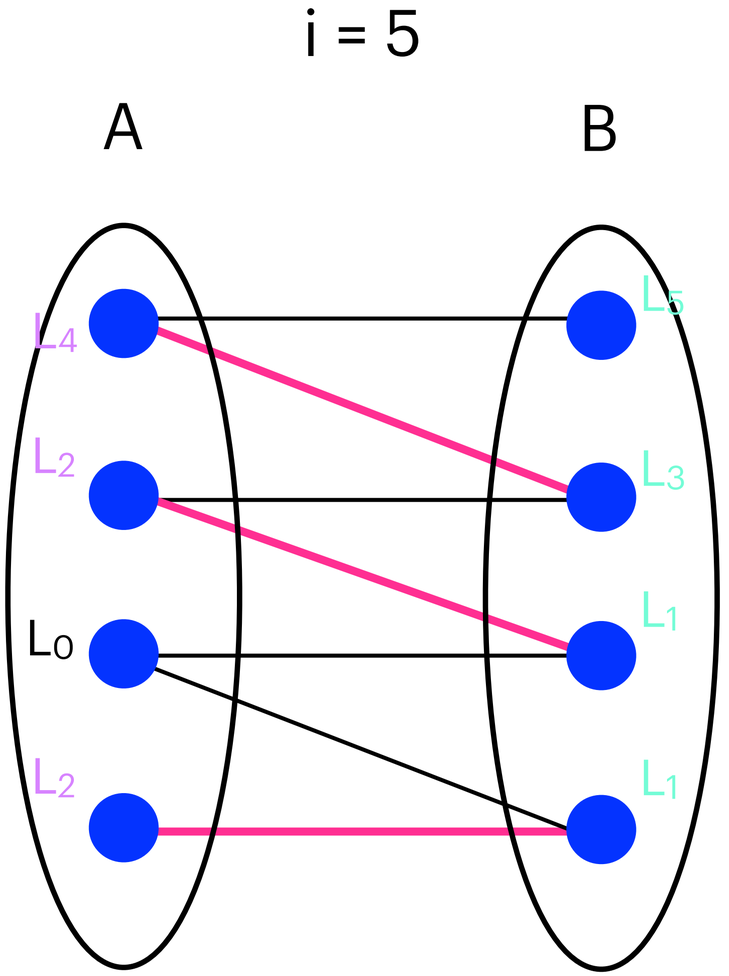
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

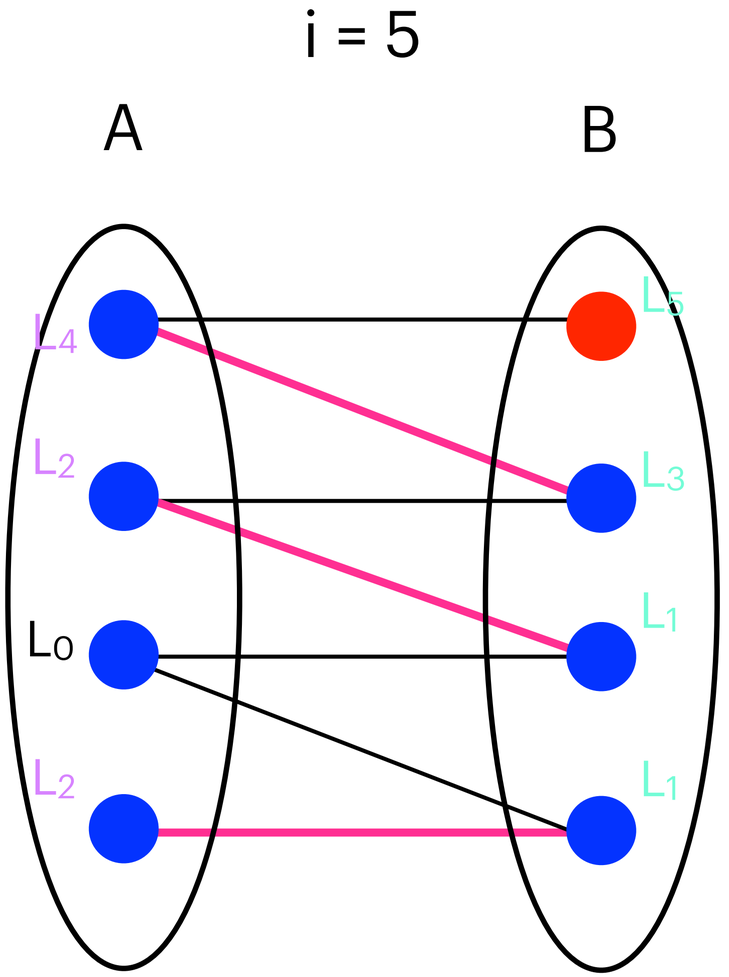
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

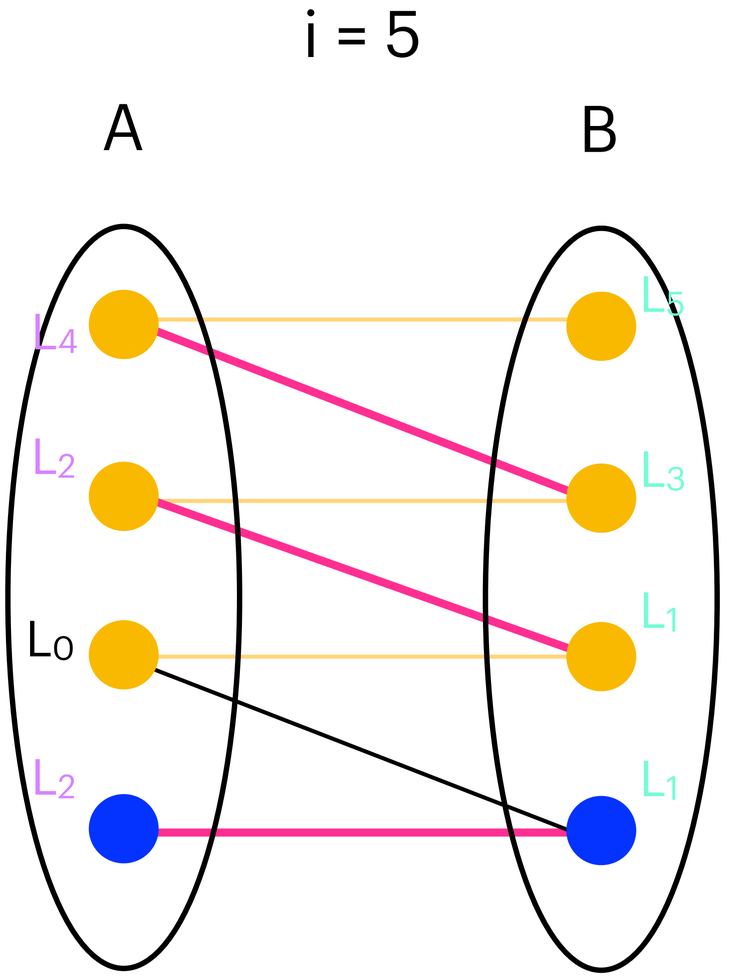
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

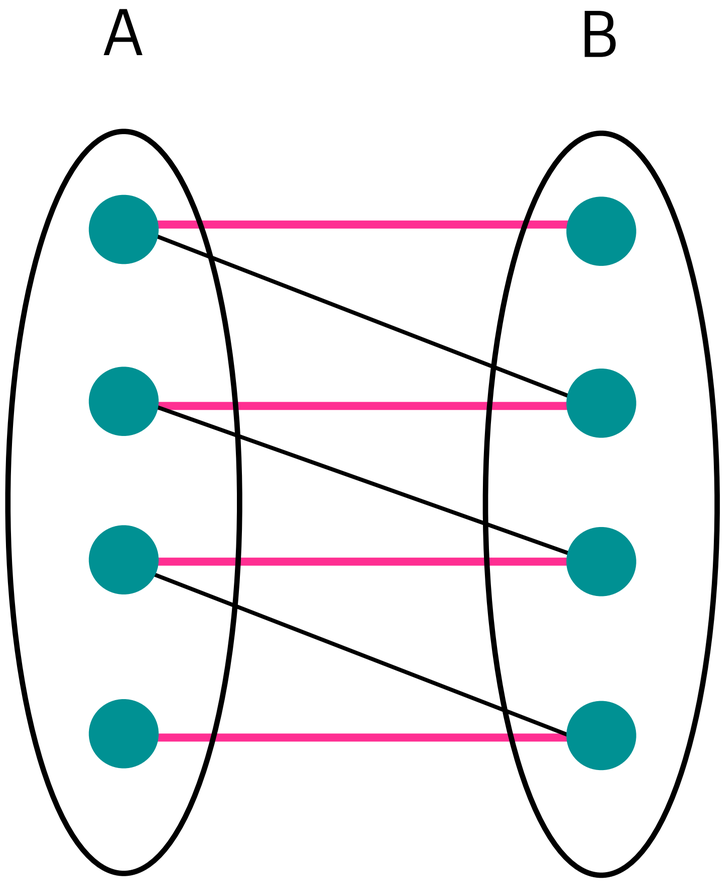
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

Input : A bipartite $G = (A \cup B, E)$

Output : Maximum Matching M

Improvement : Hopcroft Karp Algorithm

Algorithm :

Start with $M = \emptyset$
while $\exists$ augmenting path P with BFS

$M = M \oplus P$

return M

Hopcroft-Karp :

Start with $M = \emptyset$

while $\nexists$ augmenting path P

k := length of the shortest augmenting path

find more vertex disjoint augmenting paths of length k

until we have a inclusion-maximal set S of those paths

for all P in S :

$M = M \oplus P$

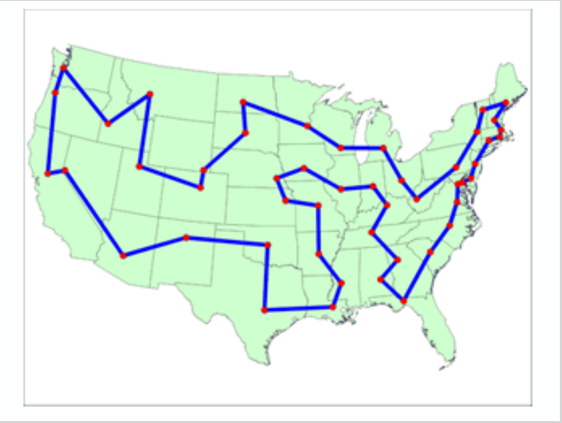
$O(|V|^{1/2} \cdot (|V|+|E|))$

Let’s take a break

TSP II

Metric TSP : 2-Approximation

Problem Description



Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $\quad l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t.

- l satisfies the triangle inequality

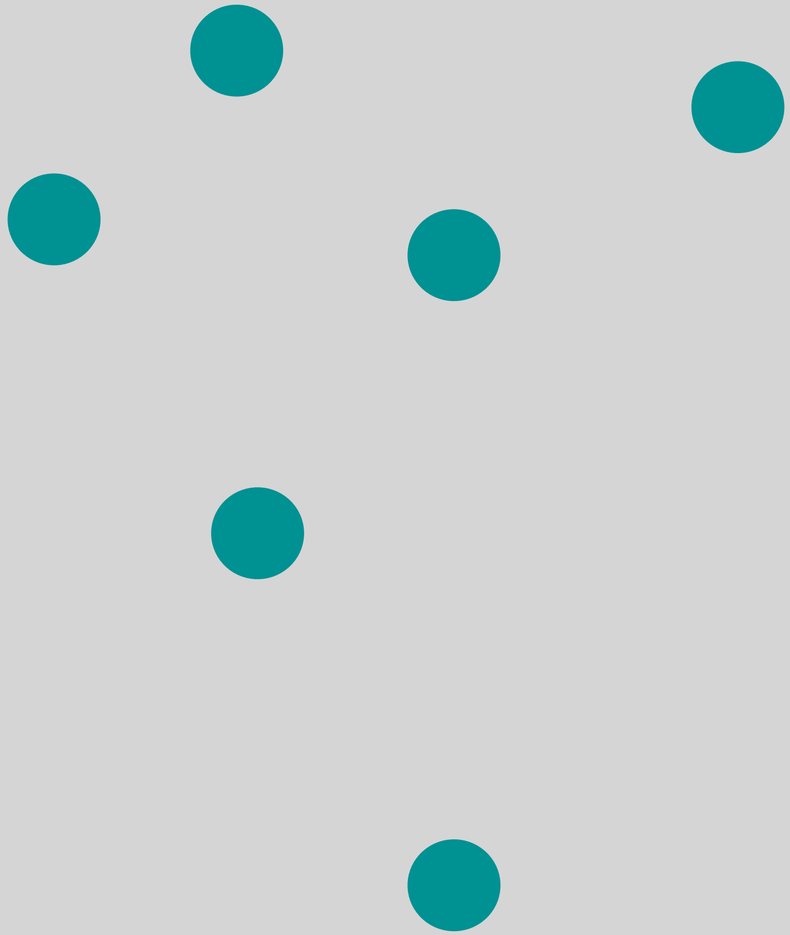
$$l(x,z) \leq l(x,y) + l(y,z)$$

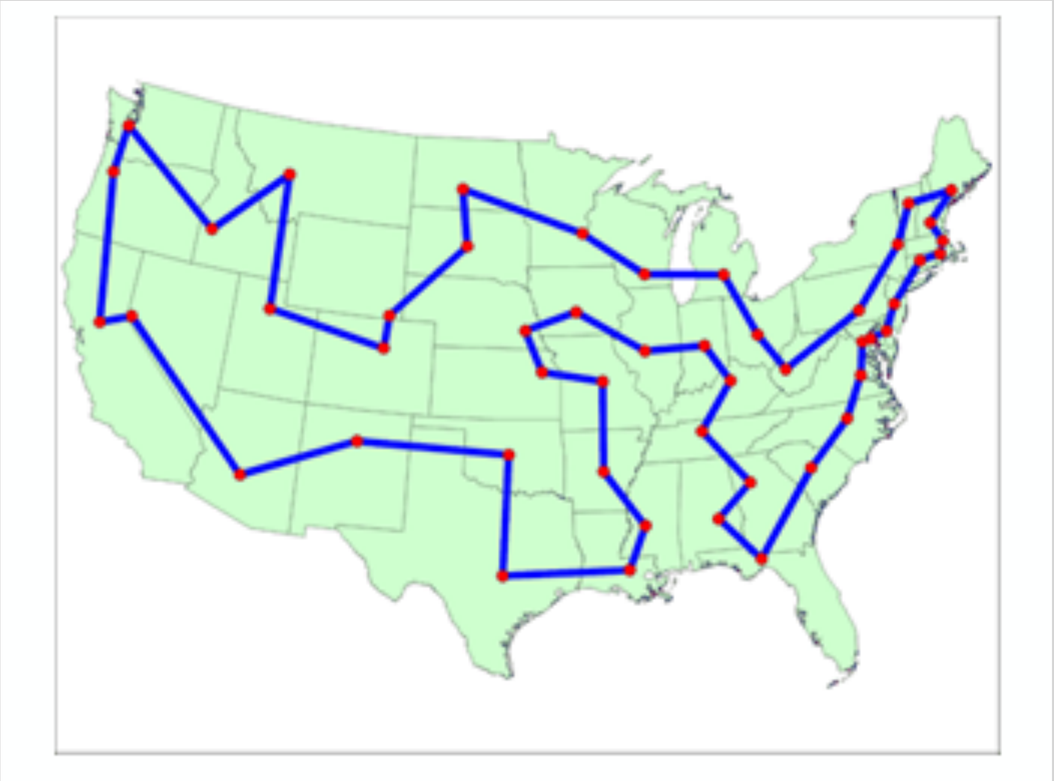
$$l(C) \leq 2\, l(OPT)$$

where $\quad OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 2-Approximation

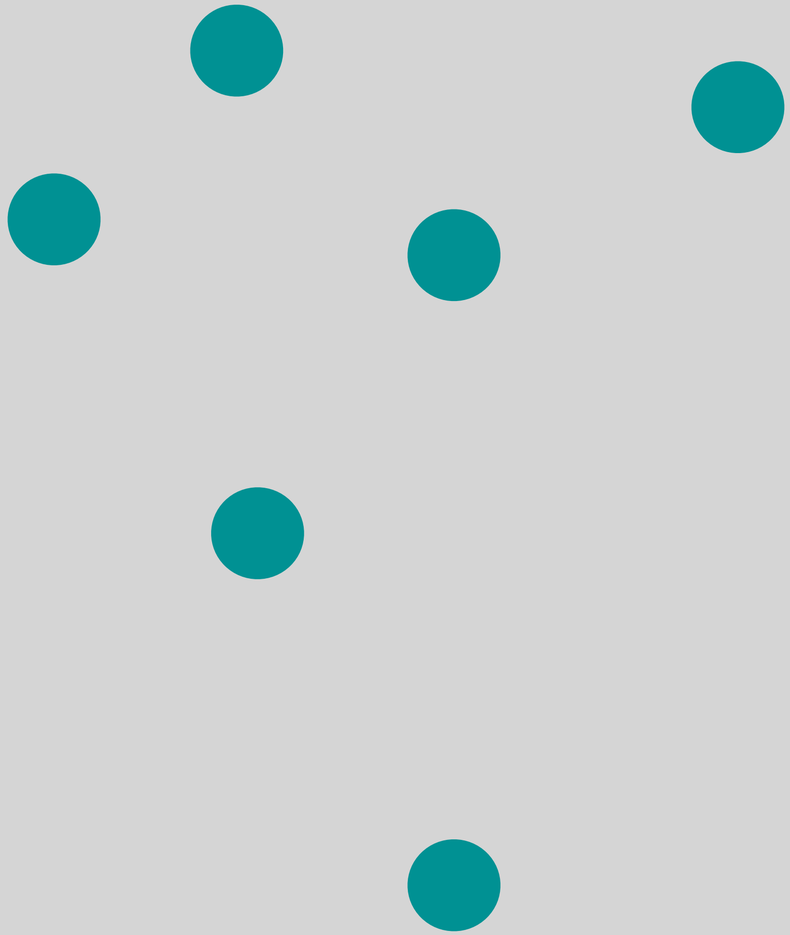
Algorithm





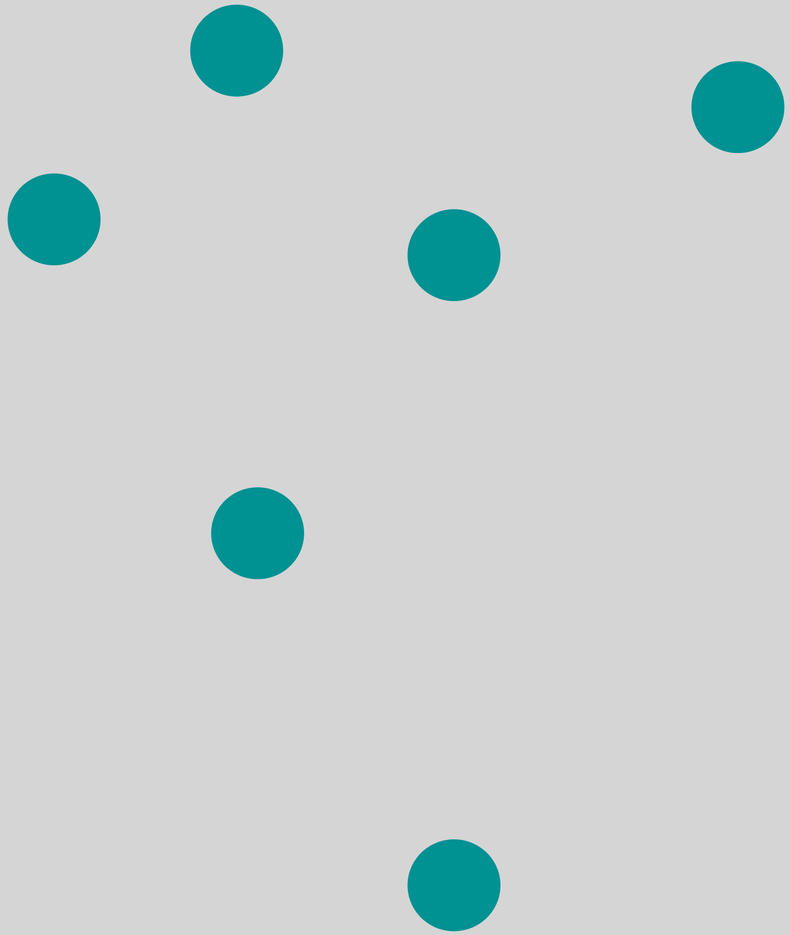
Metric TSP : 2-Approximation

Algorithm



Metric TSP : 2-Approximation

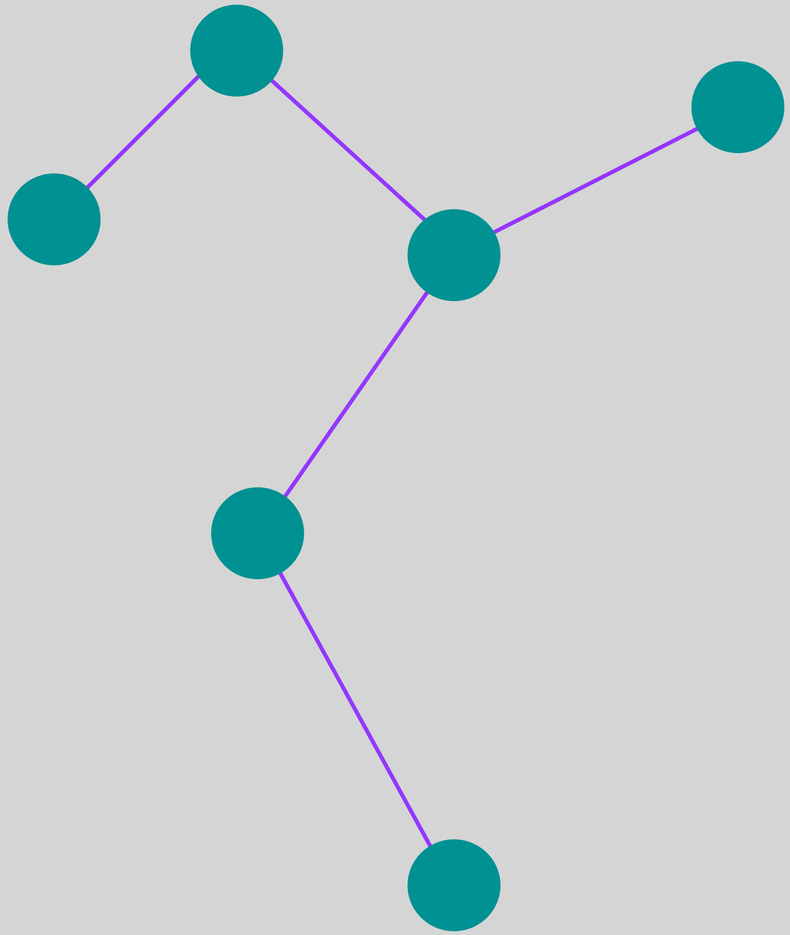
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

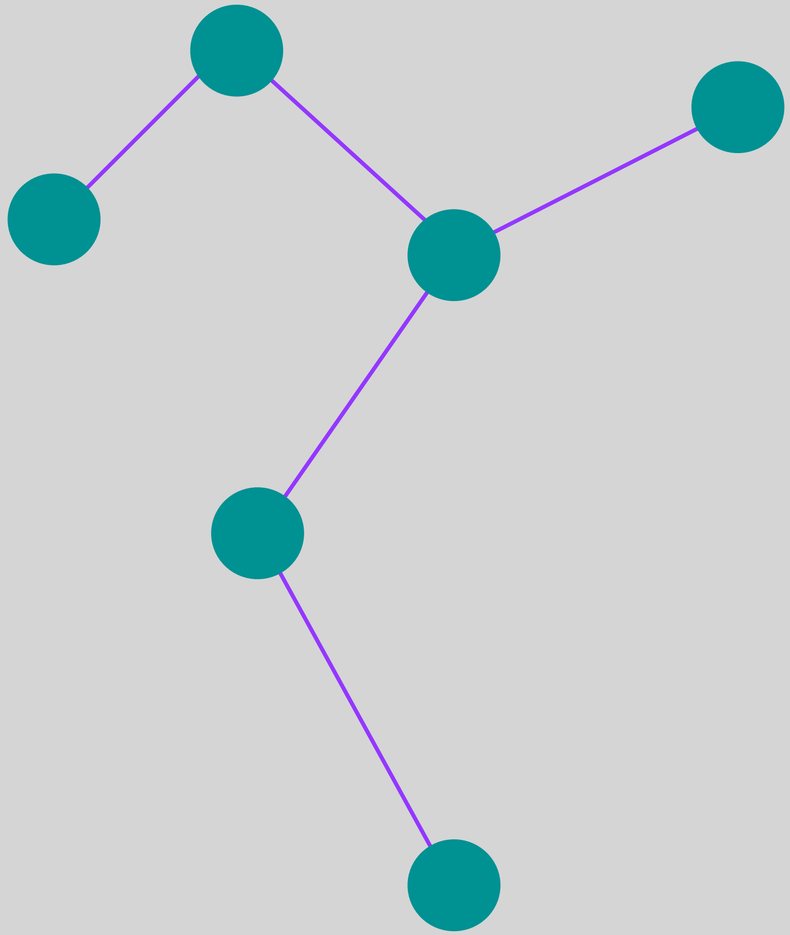
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

Algorithm

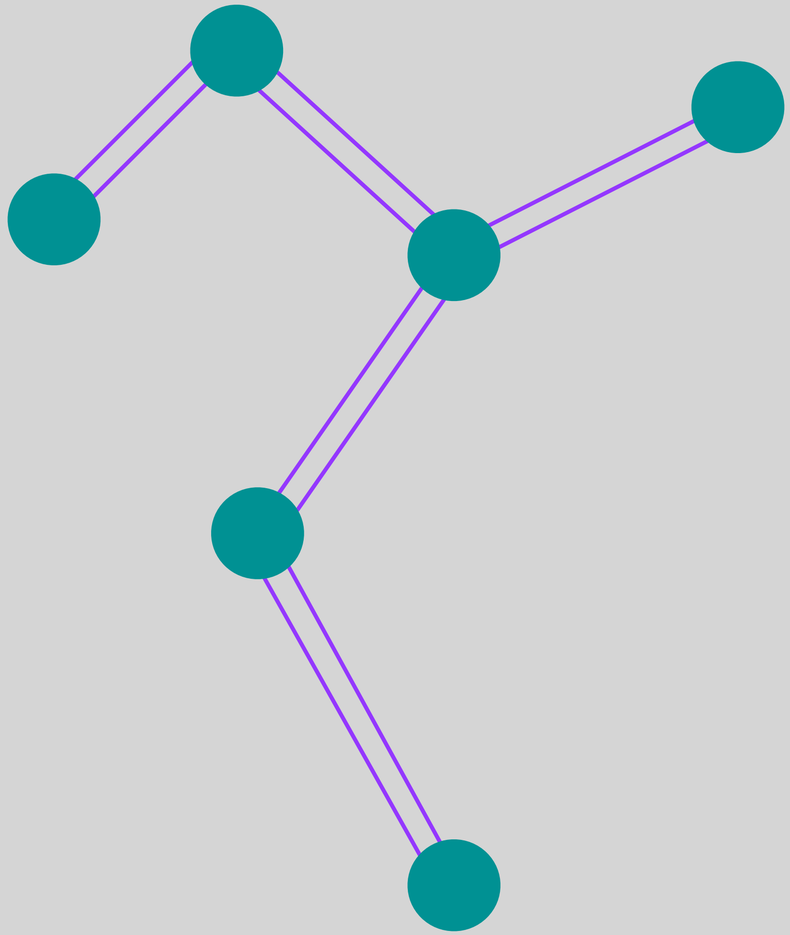


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

Algorithm

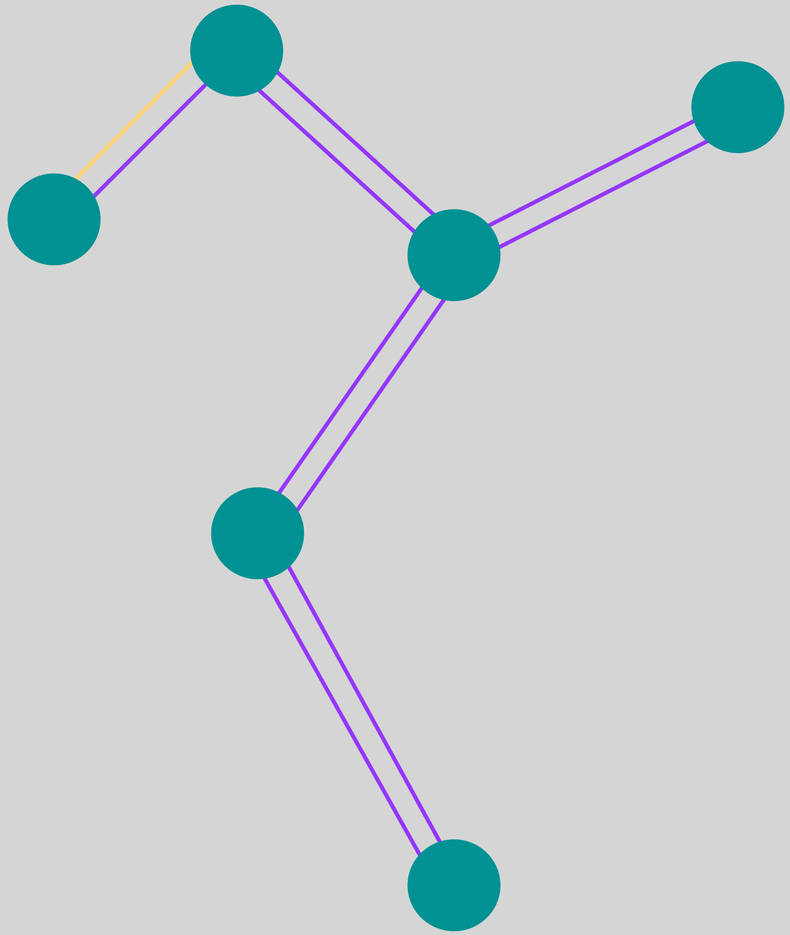


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

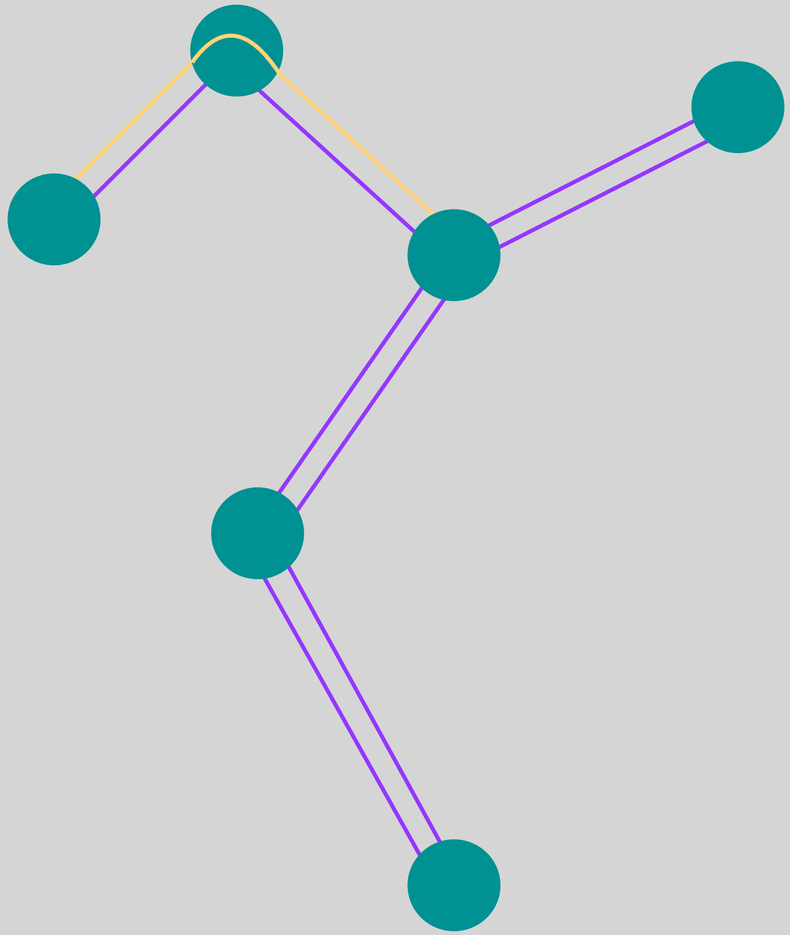
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



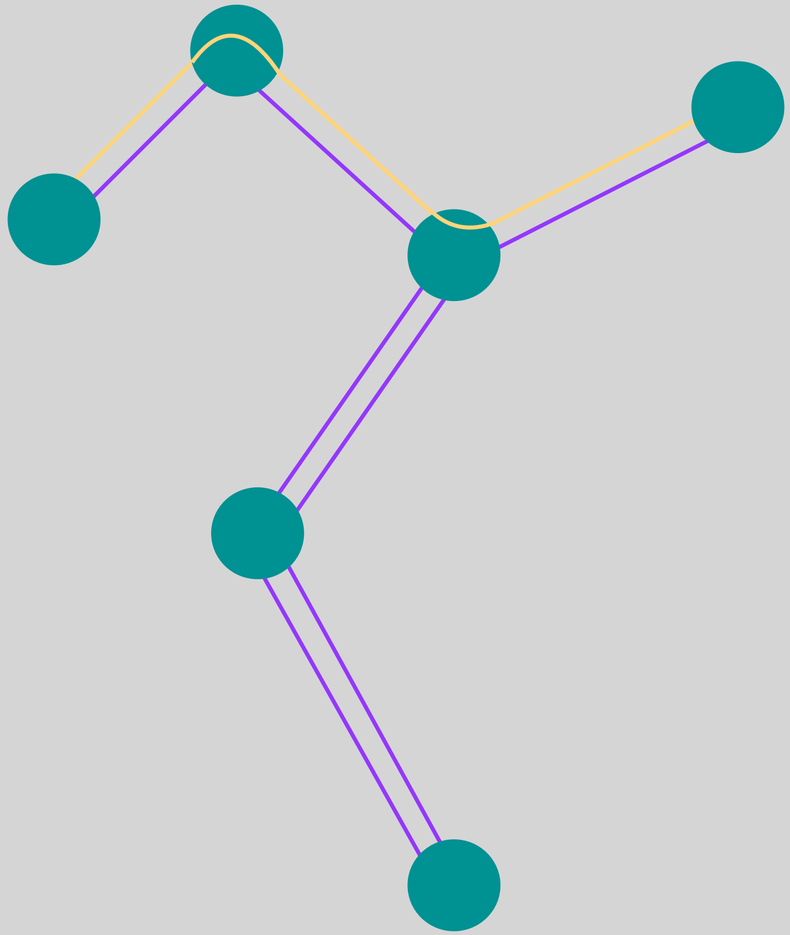
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



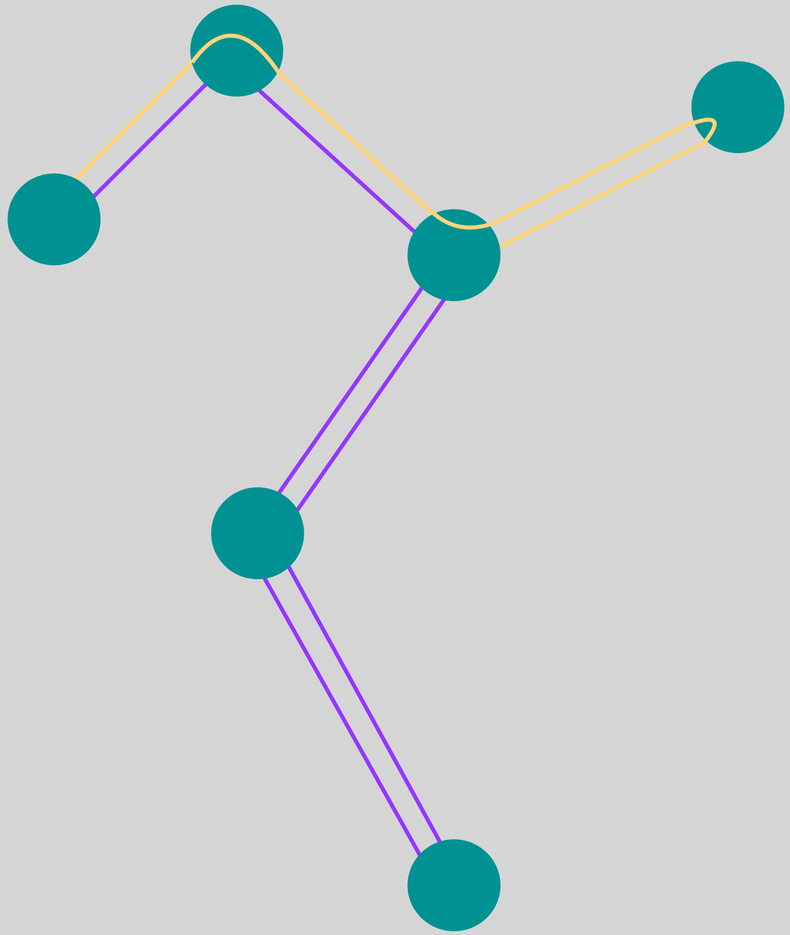
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



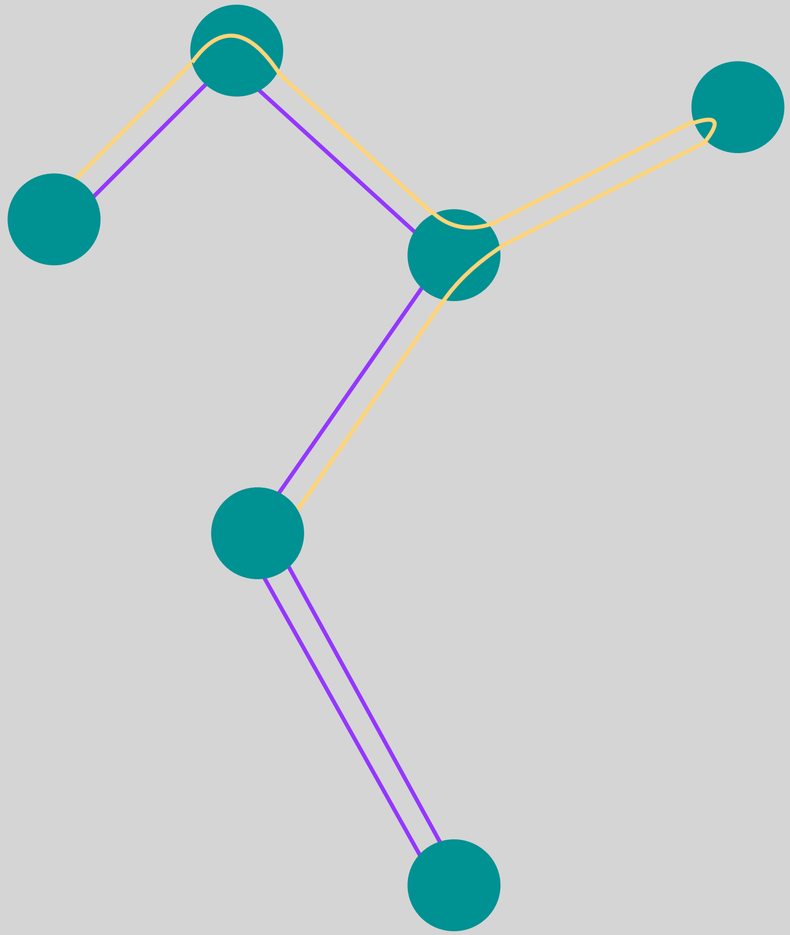
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



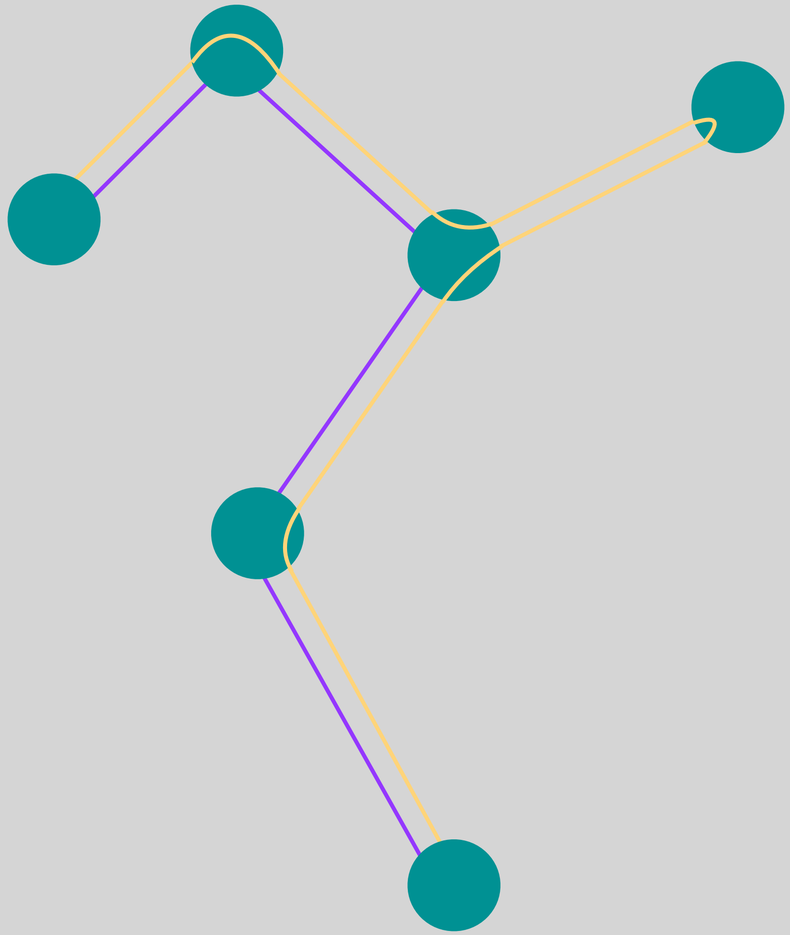
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



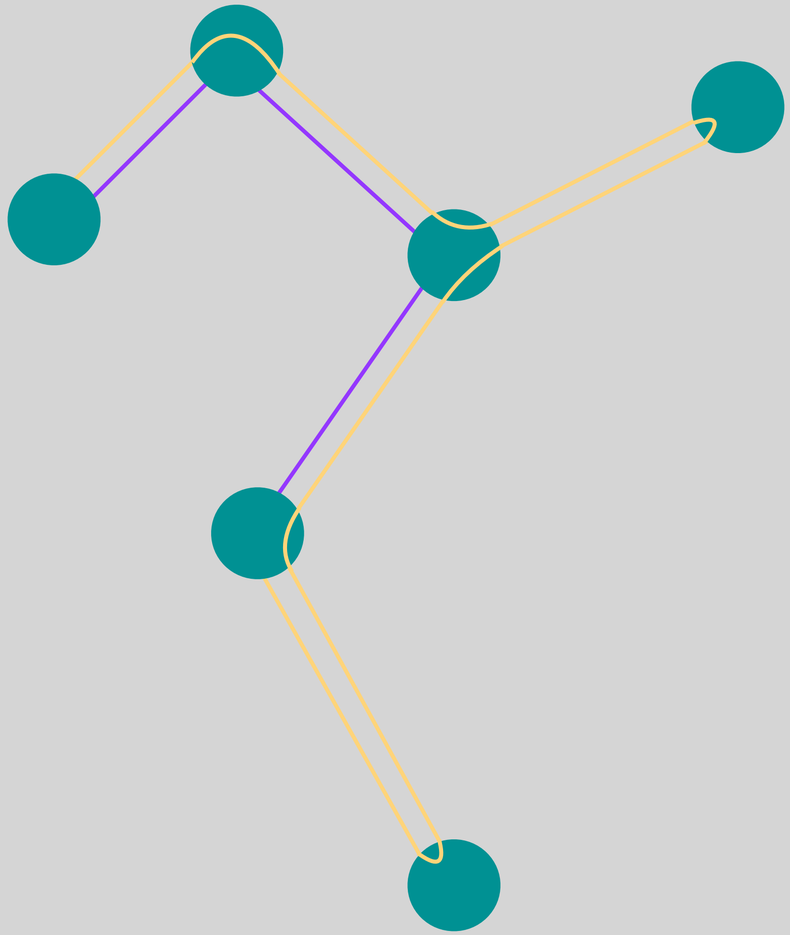
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



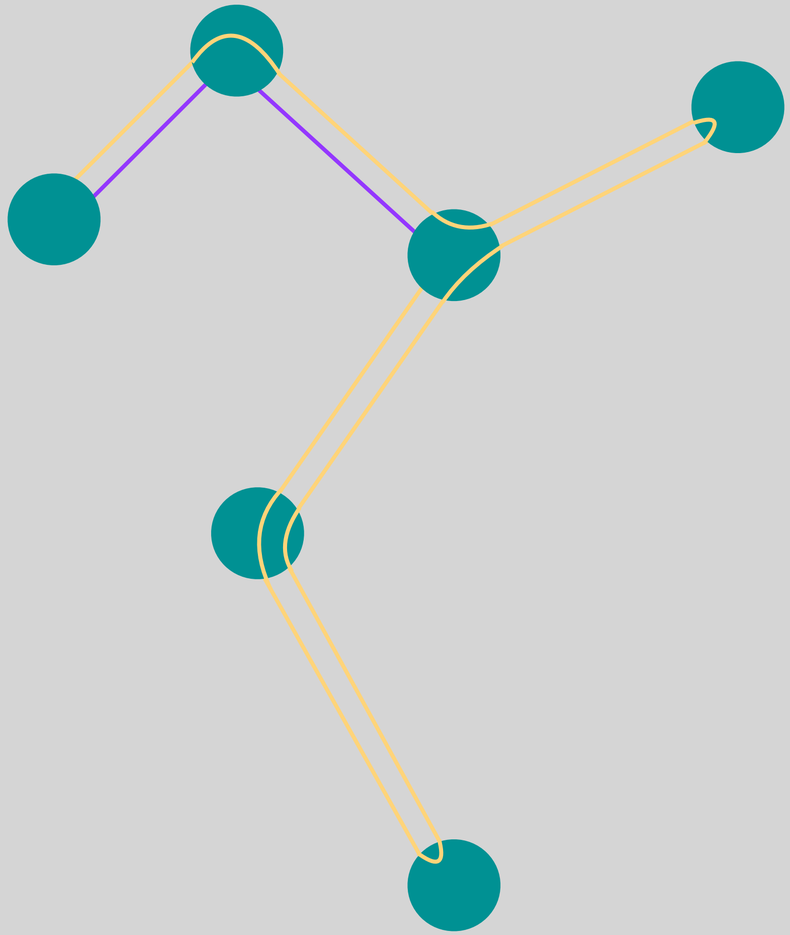
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



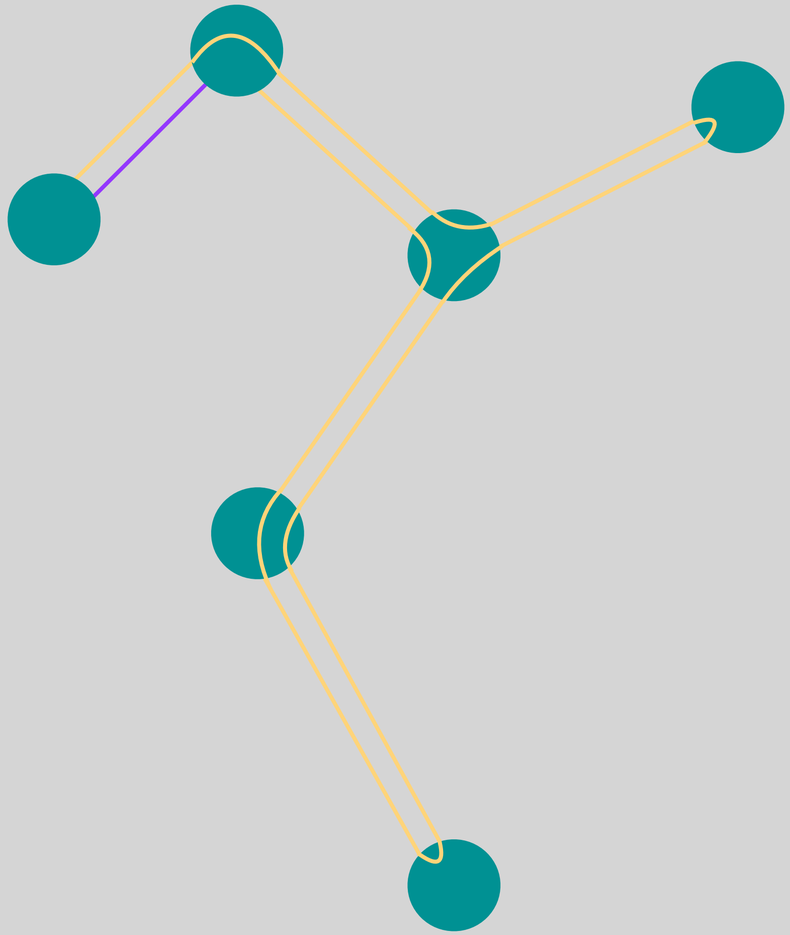
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

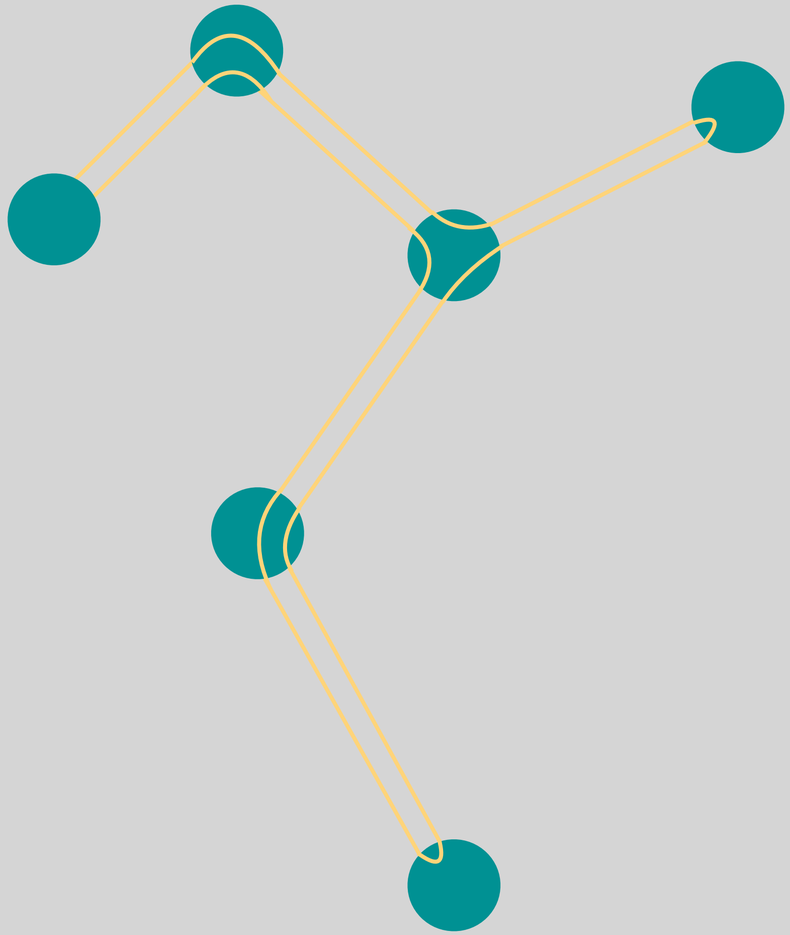
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

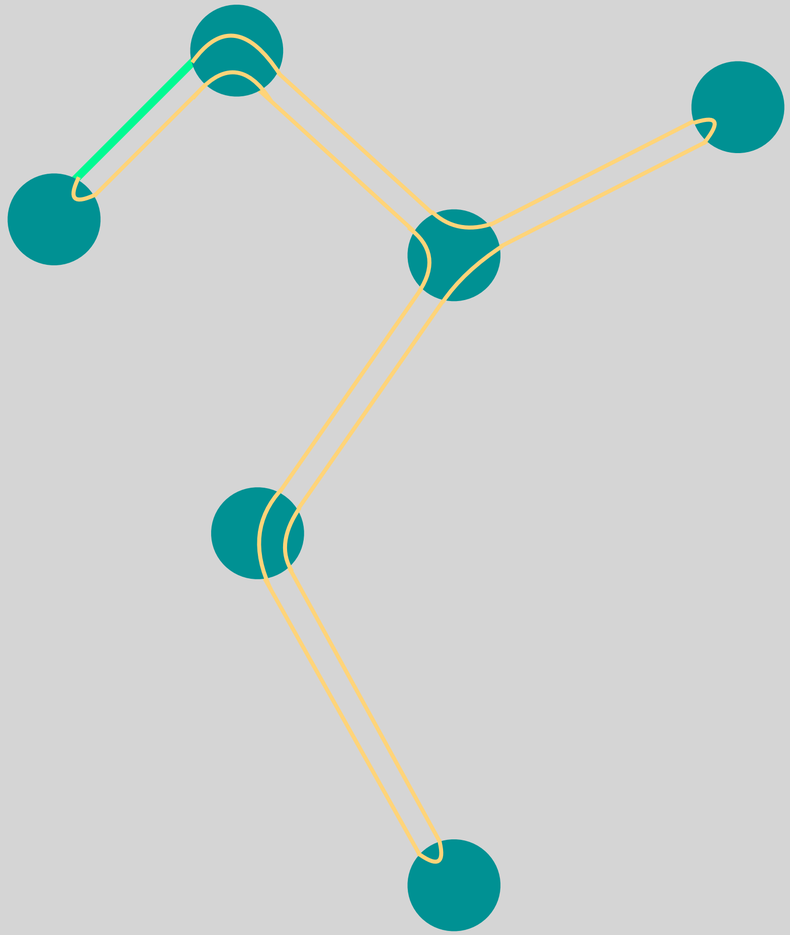
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

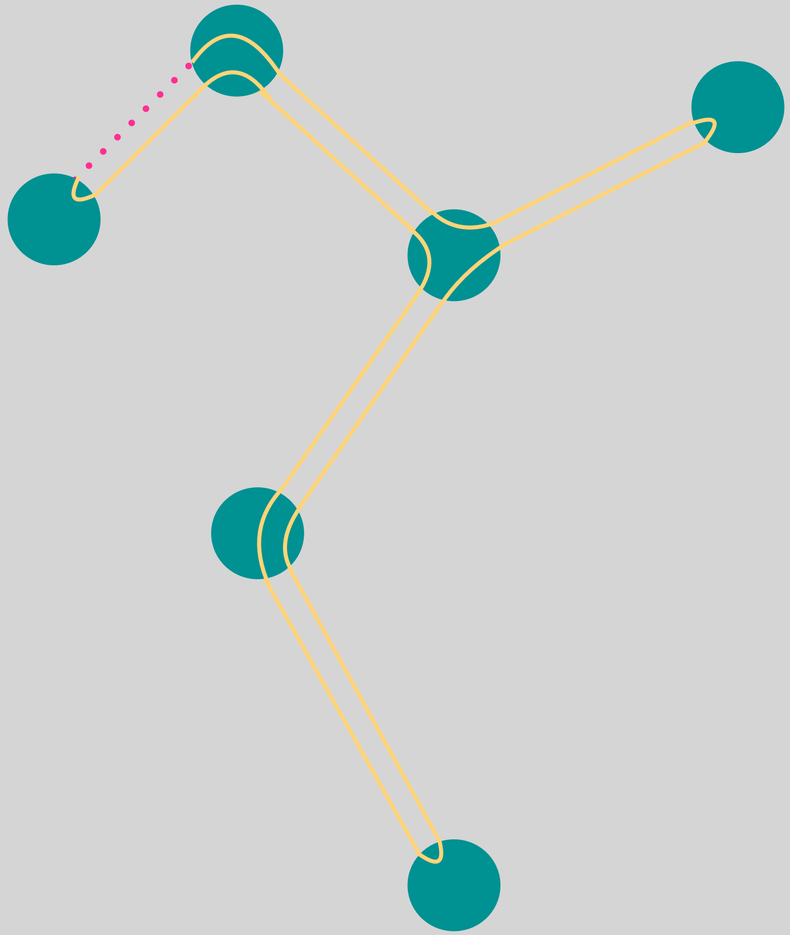
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

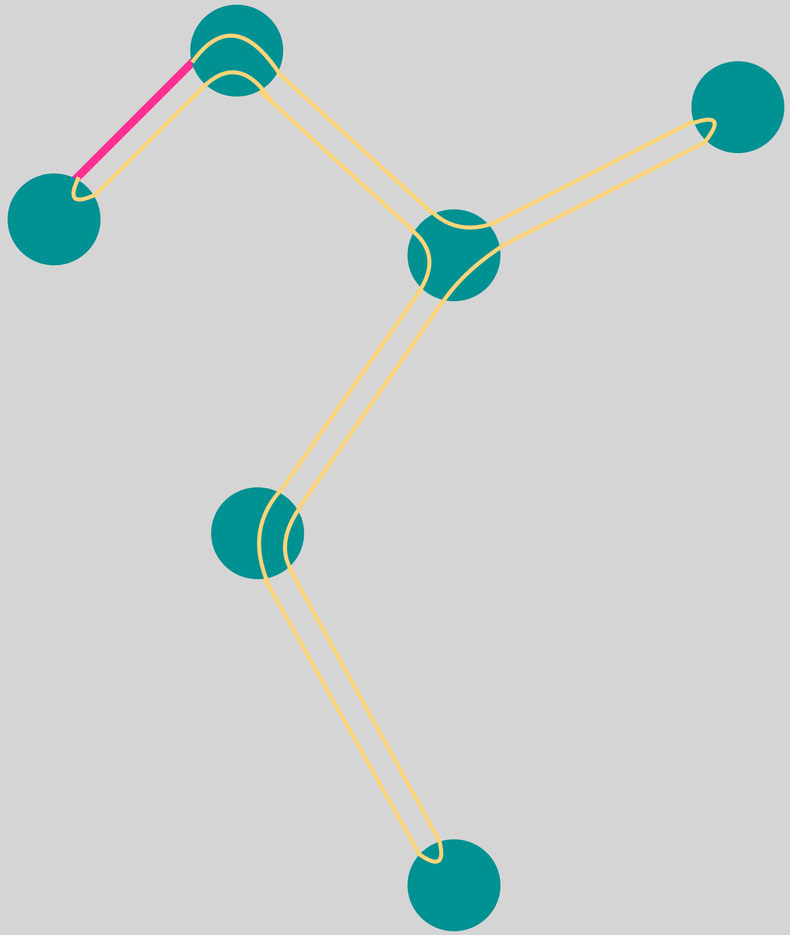
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

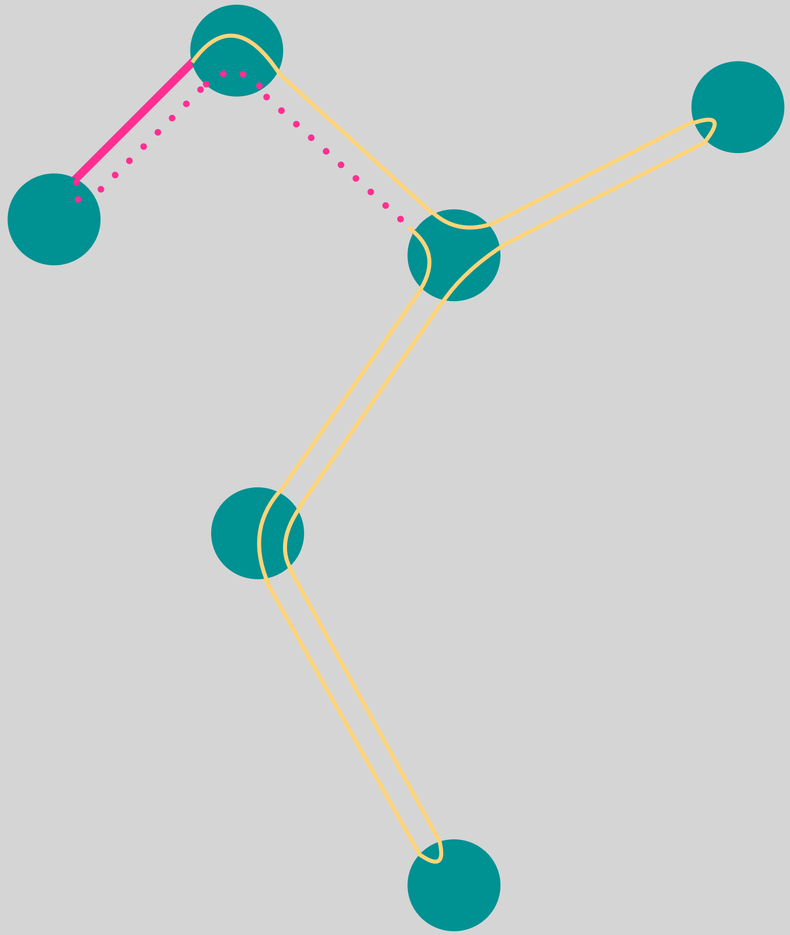
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

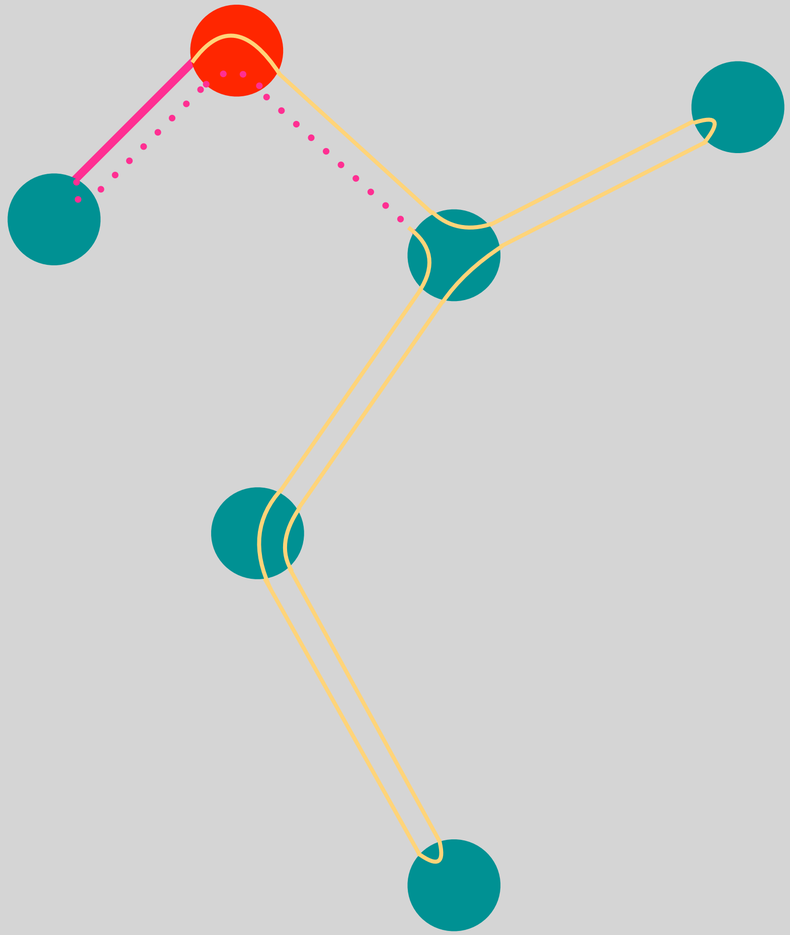
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

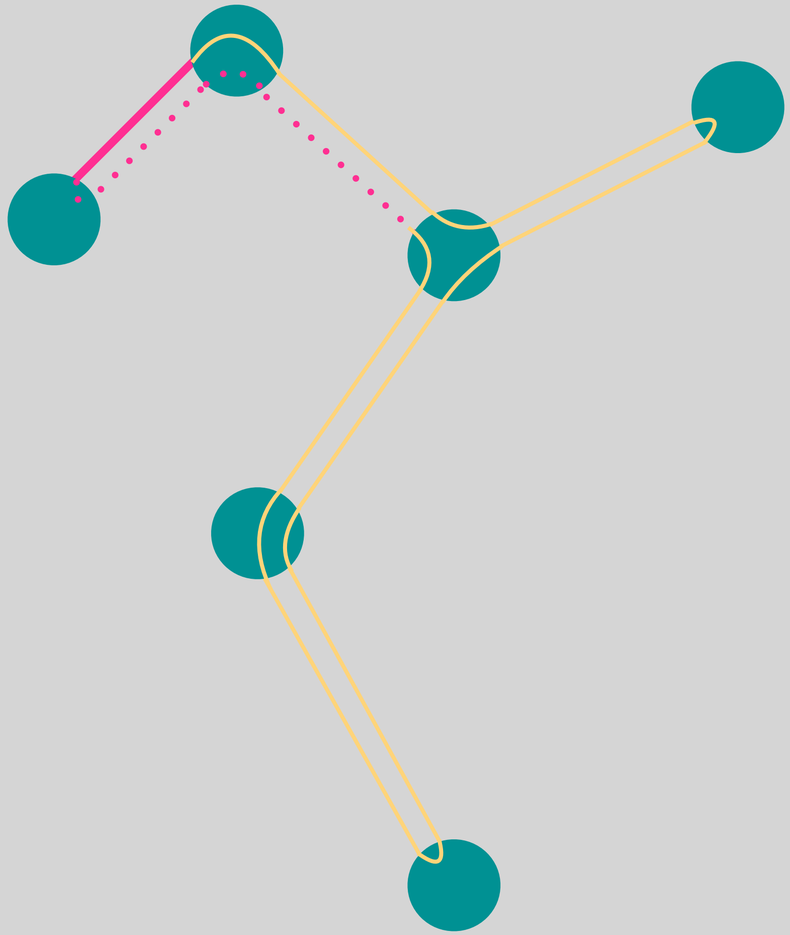
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

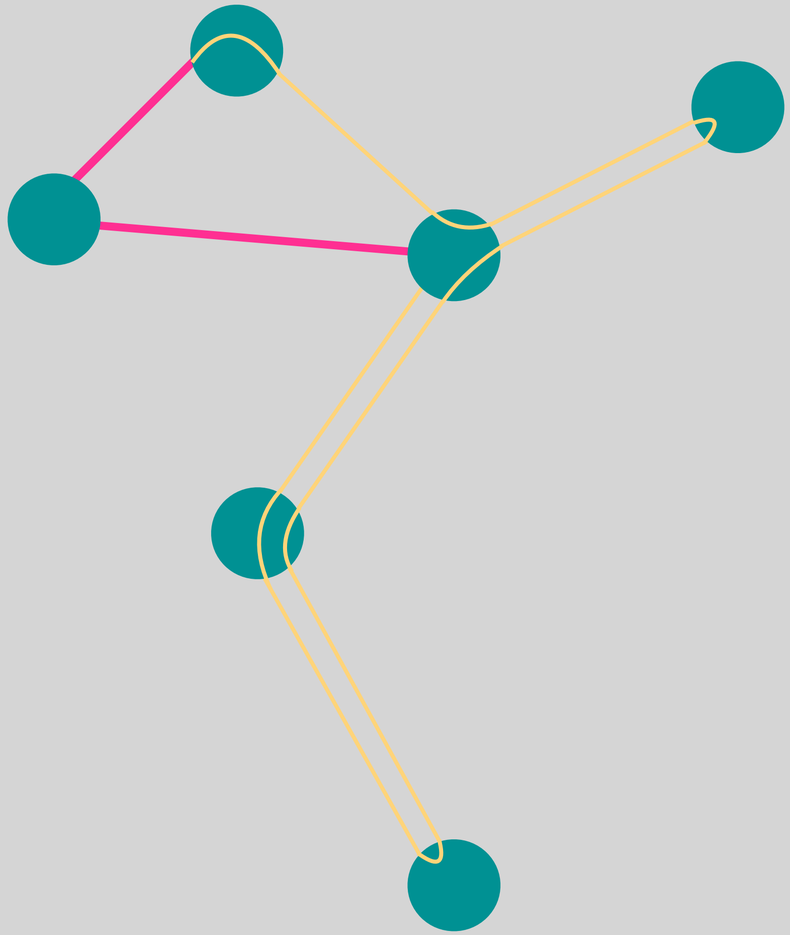
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

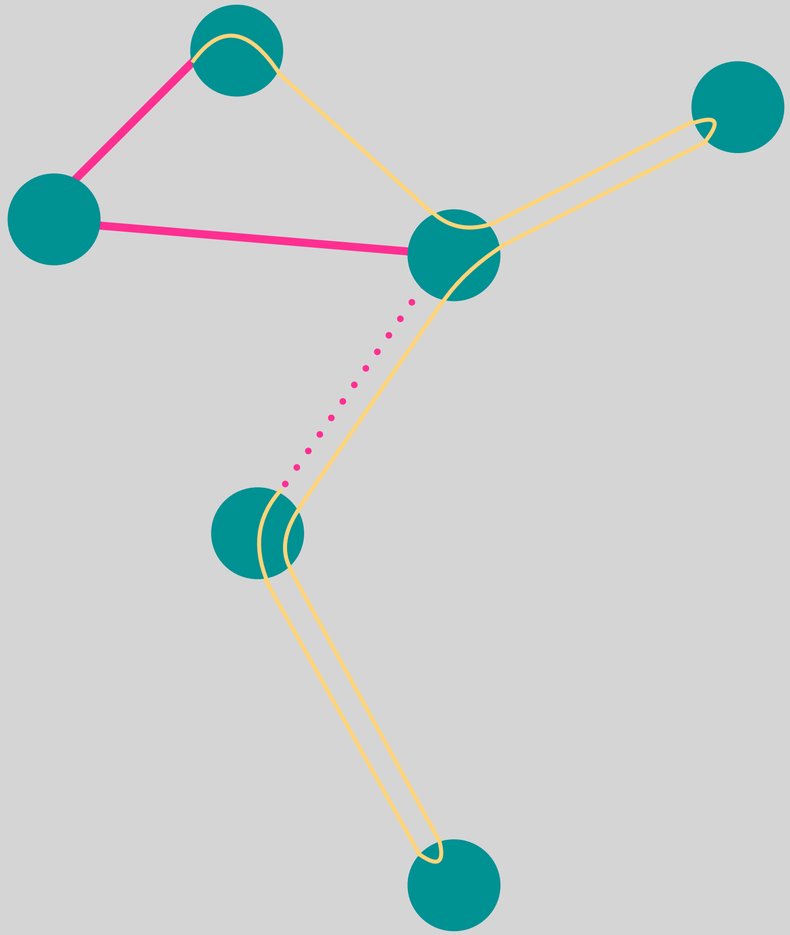
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

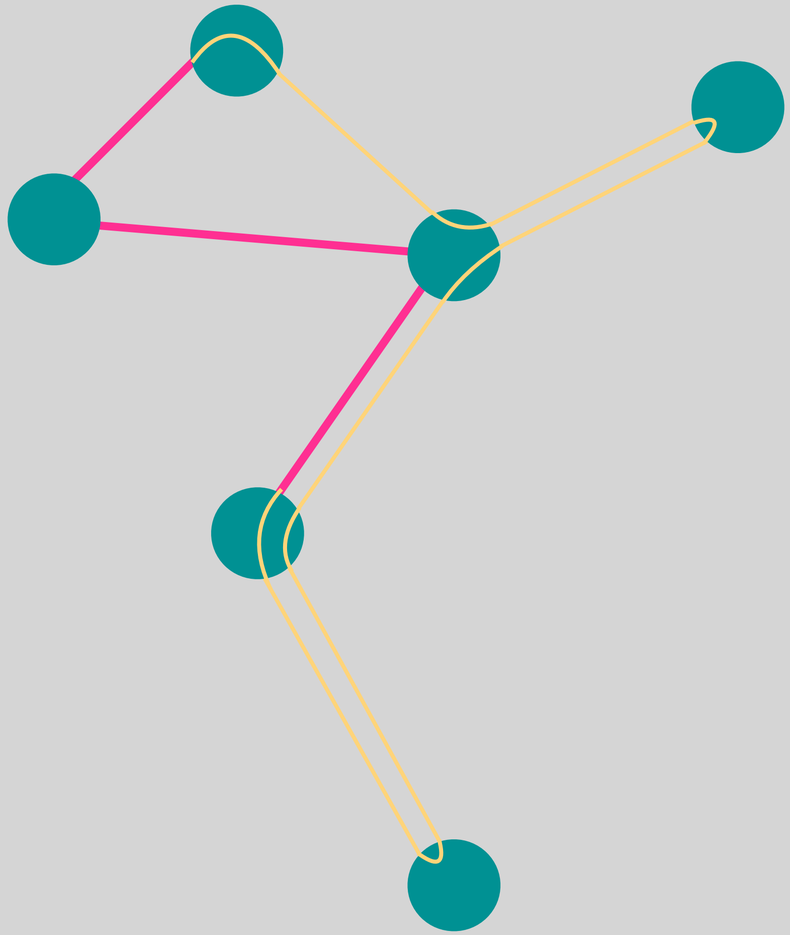
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

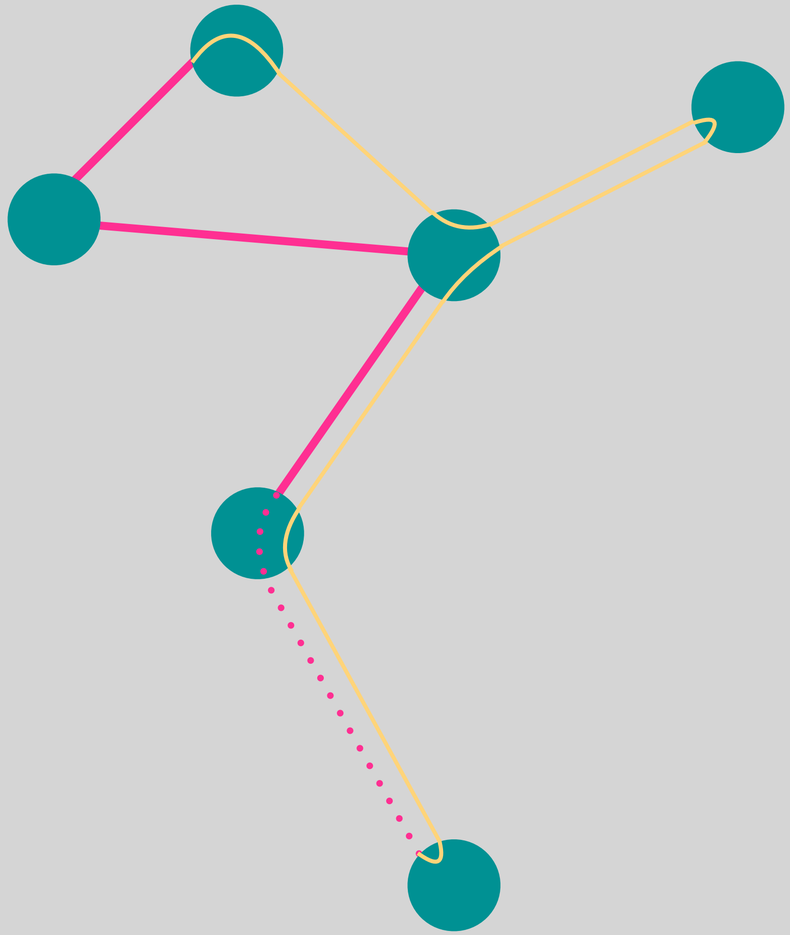
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

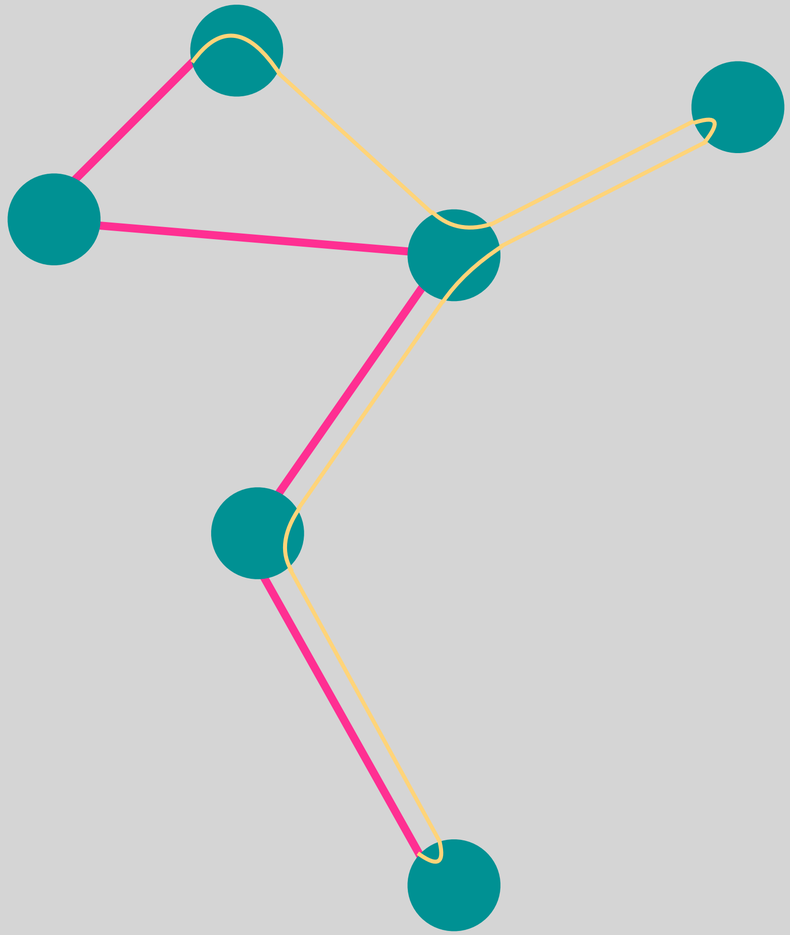
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

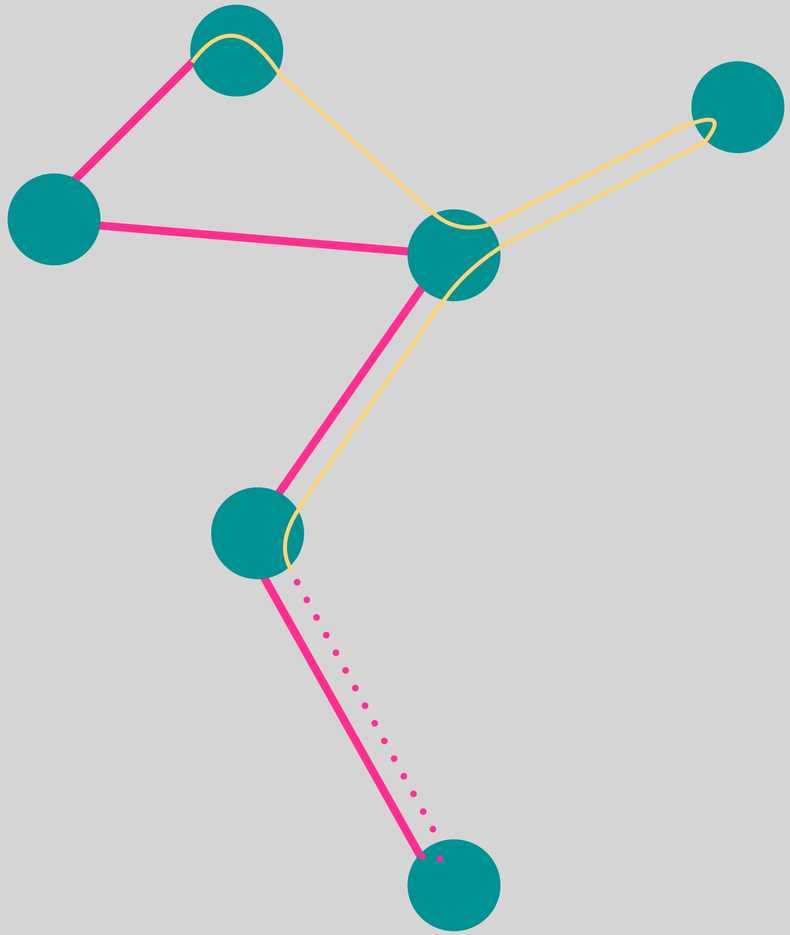
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

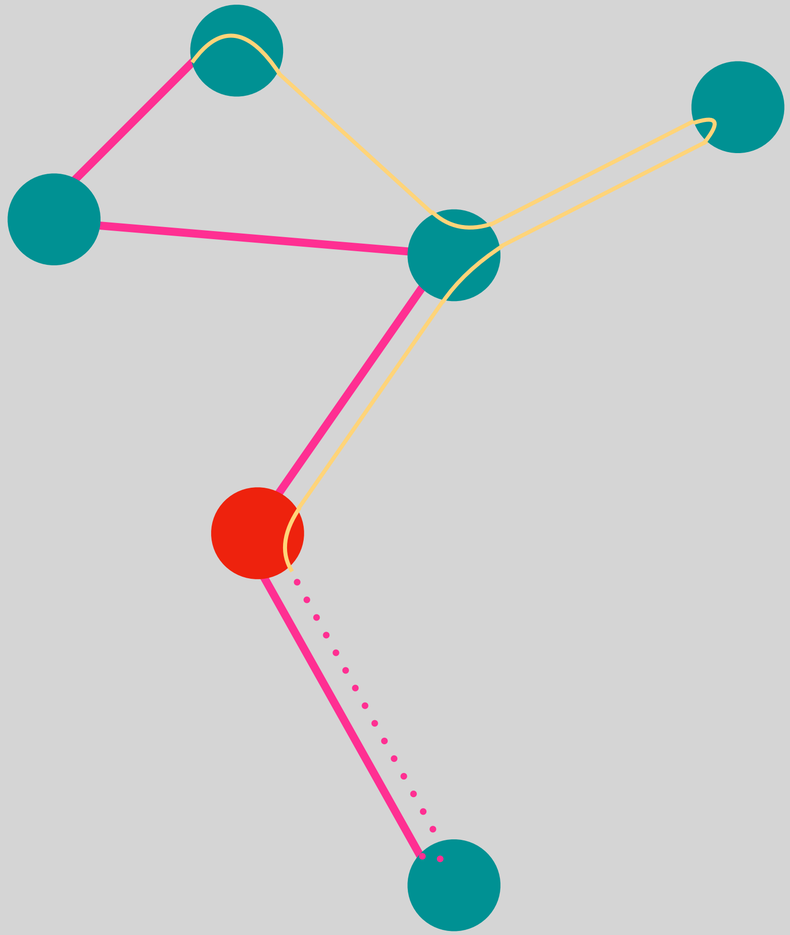
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

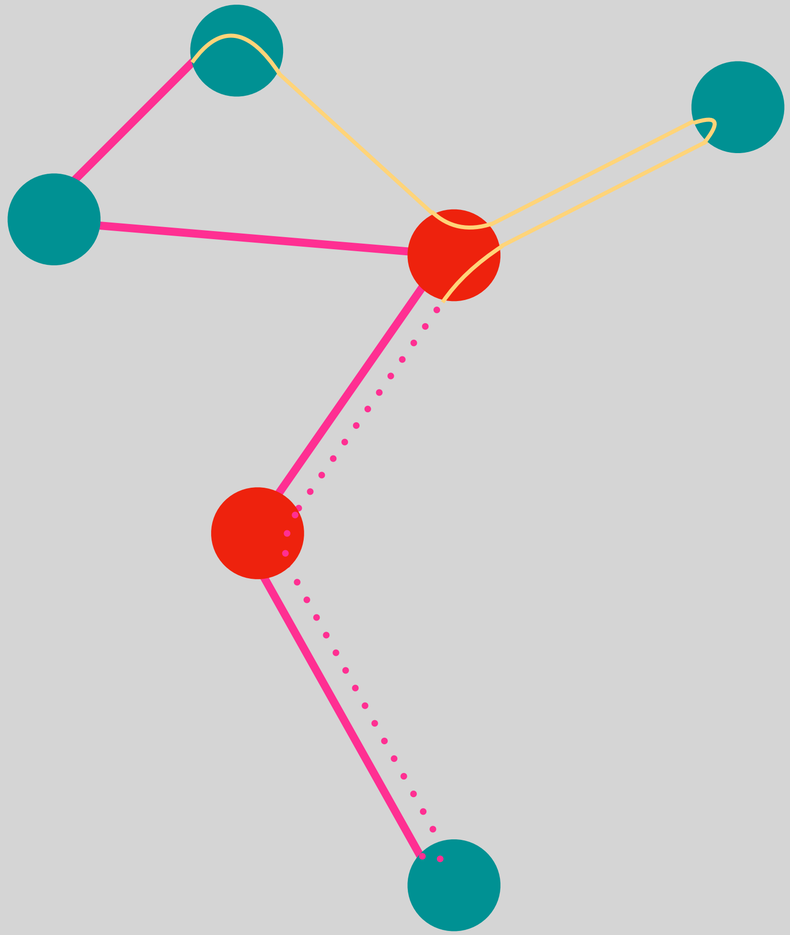
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

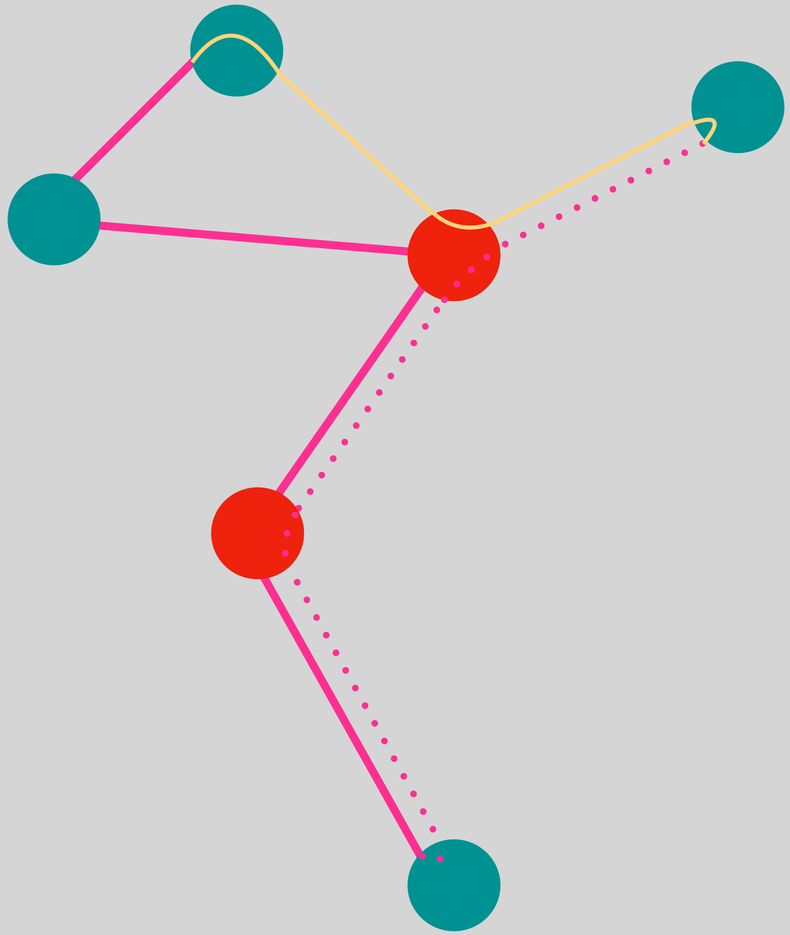
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

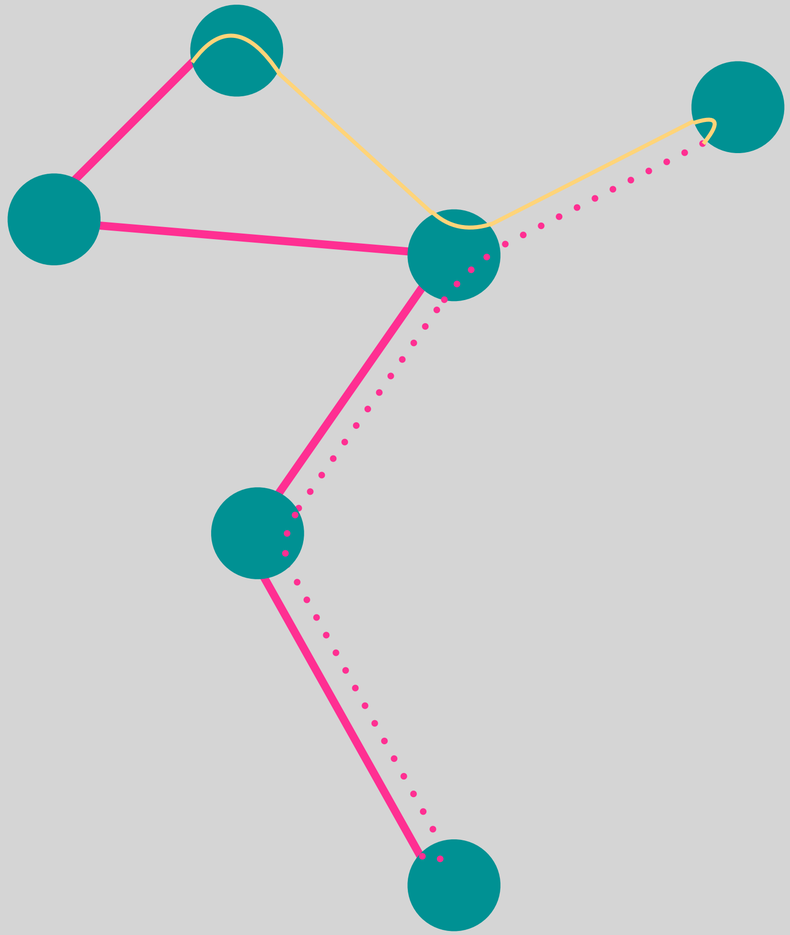
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

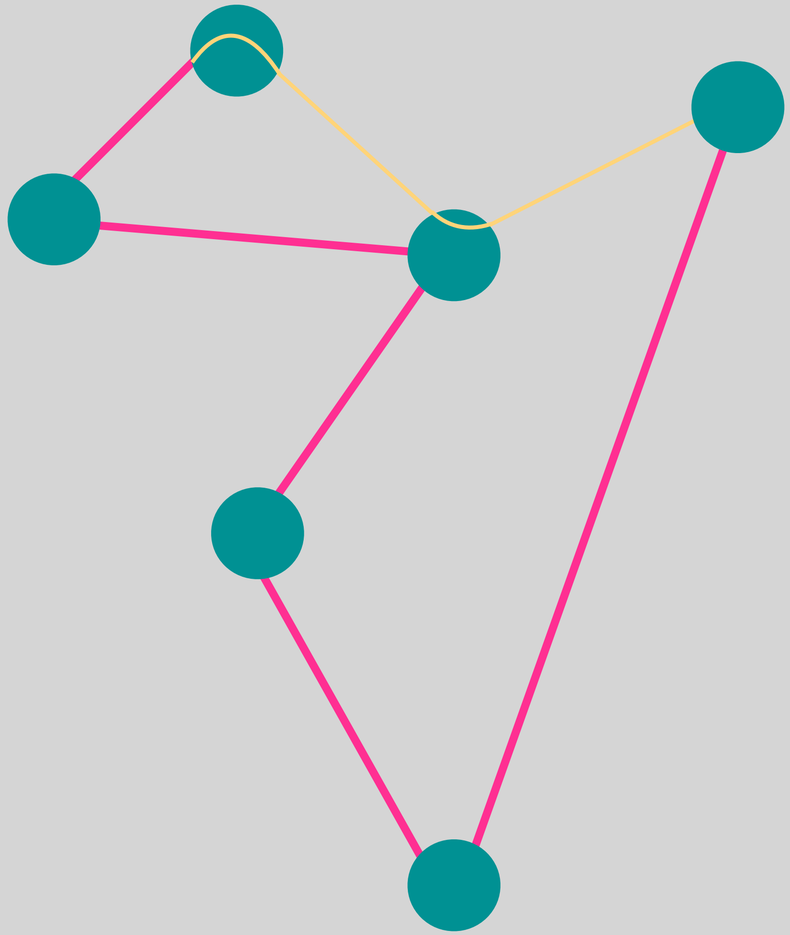
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

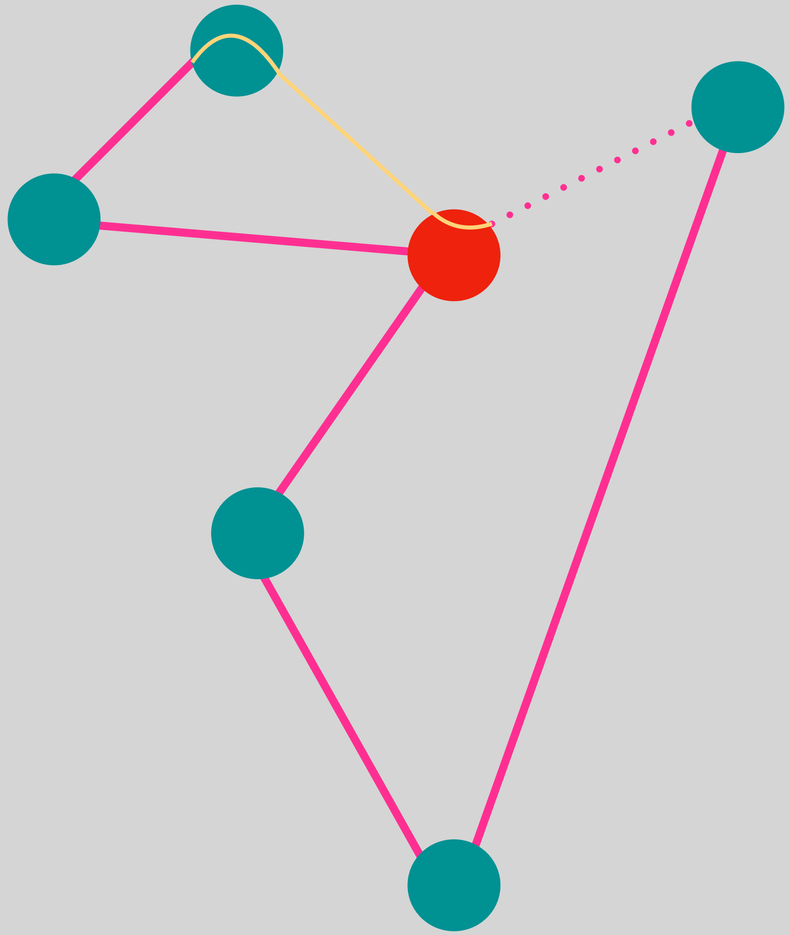
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

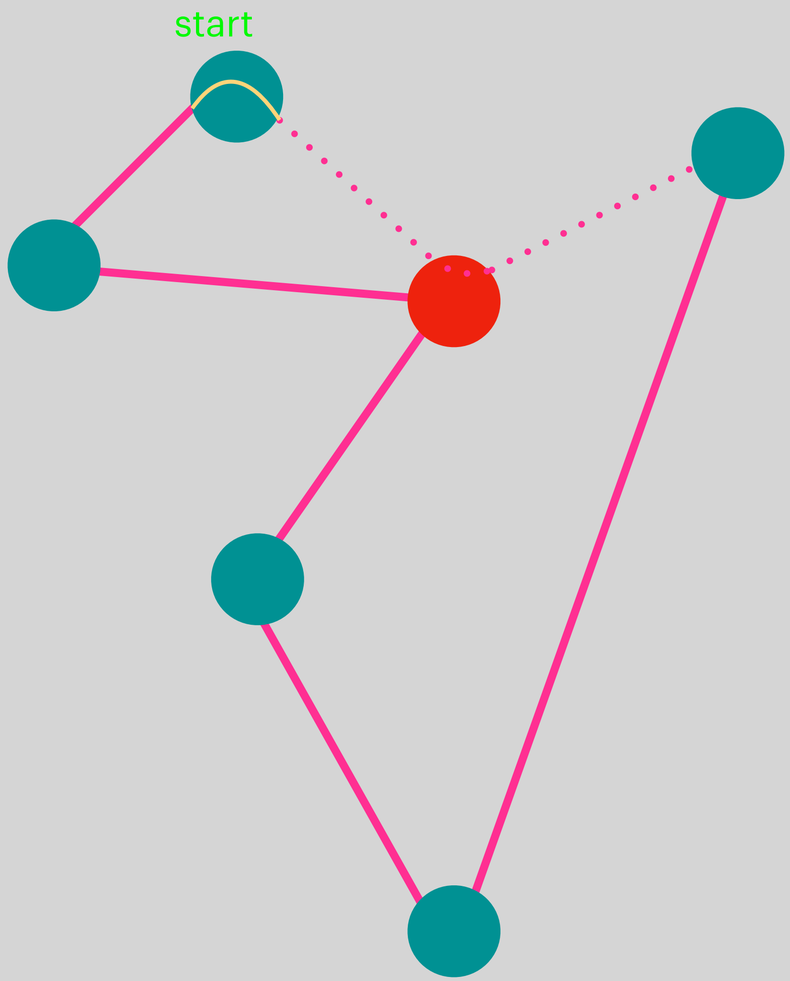
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

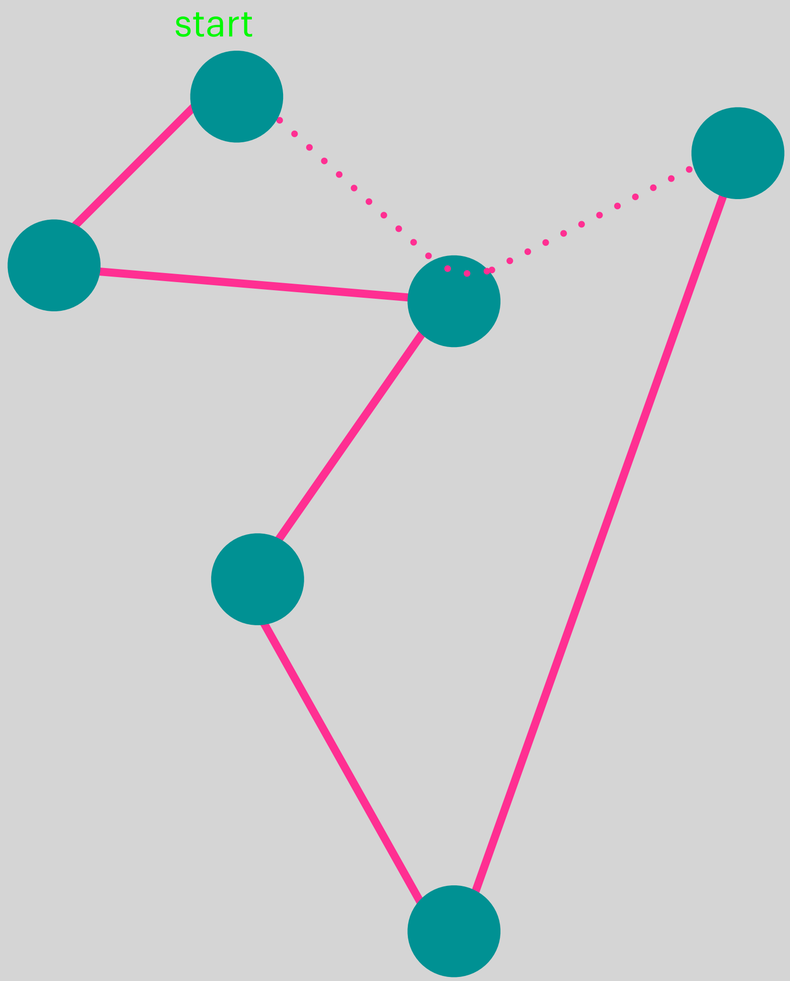
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

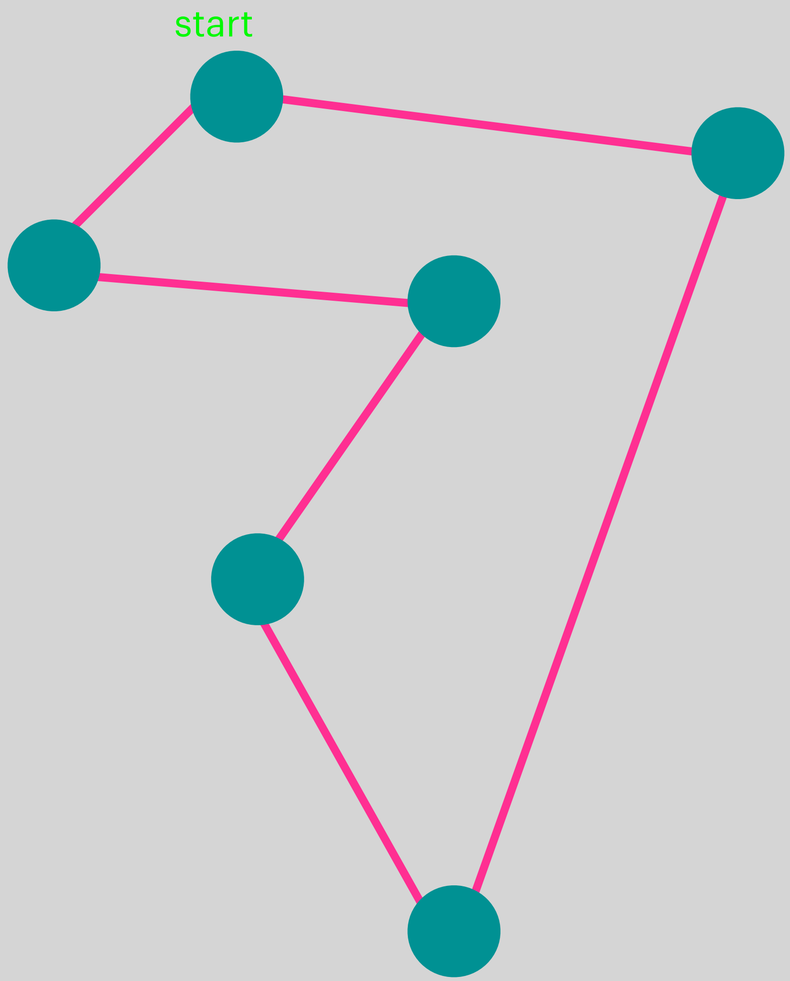
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

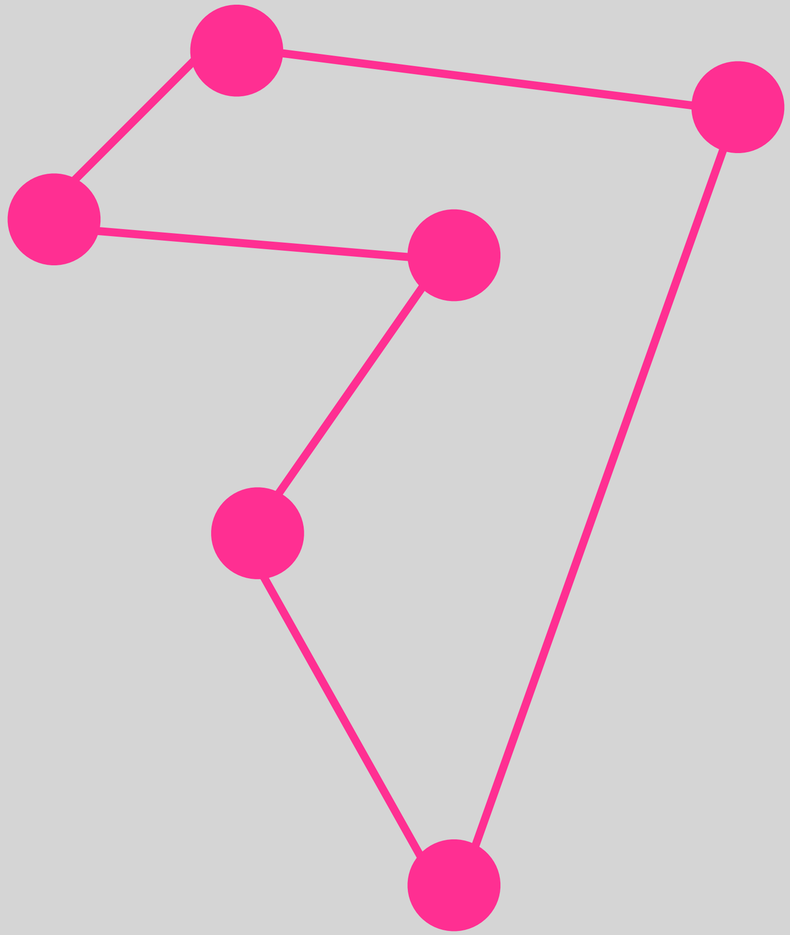
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

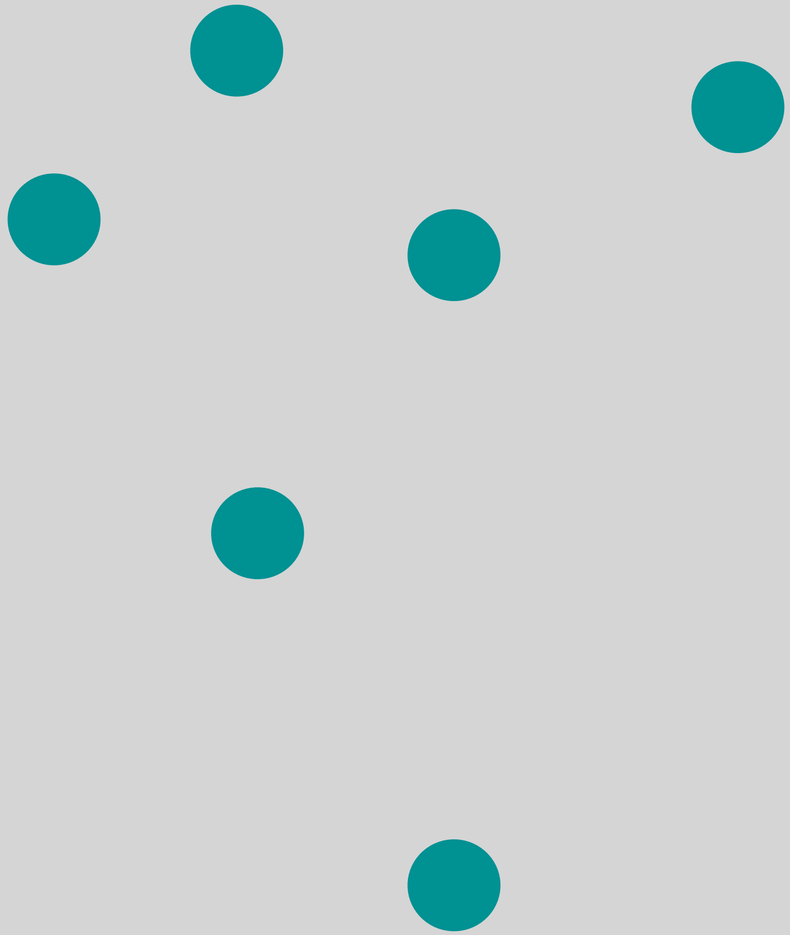
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

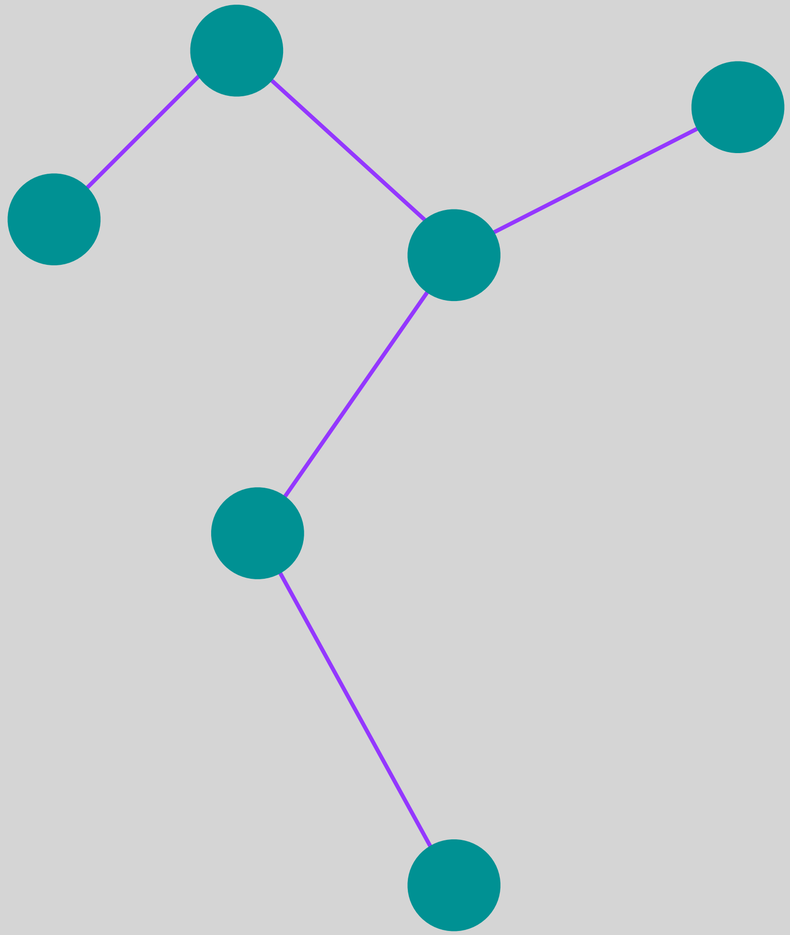
Metric TSP : 2-Approximation

Correctness



Metric TSP : 2-Approximation

Correctness

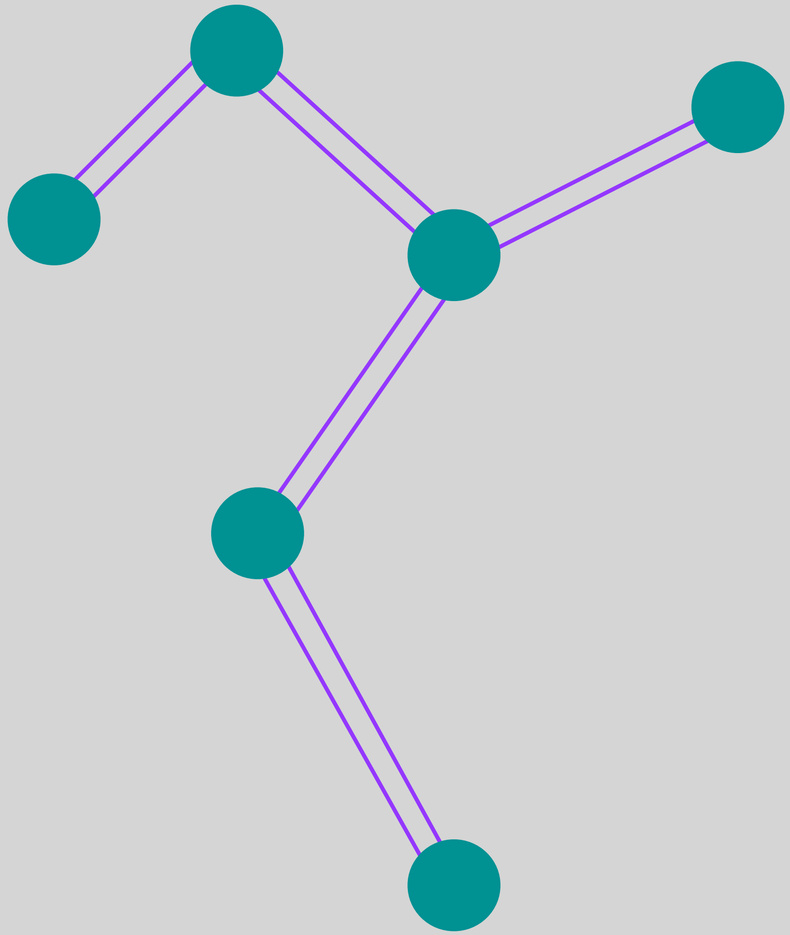


1. Find the MST T

$$l(T) \leq OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness

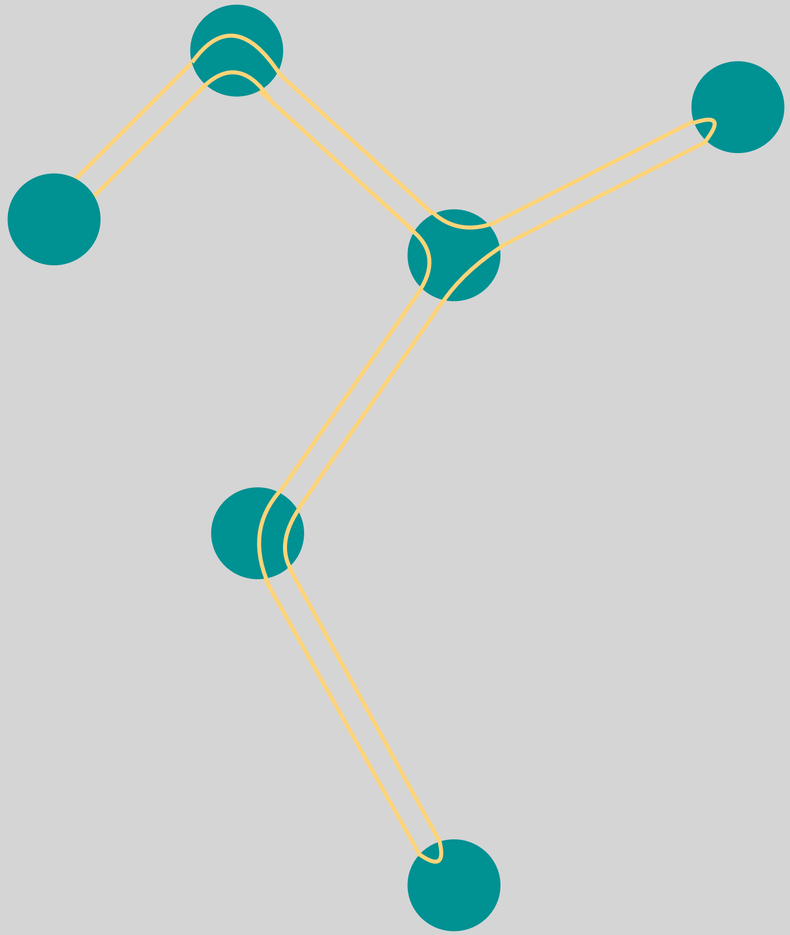


1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

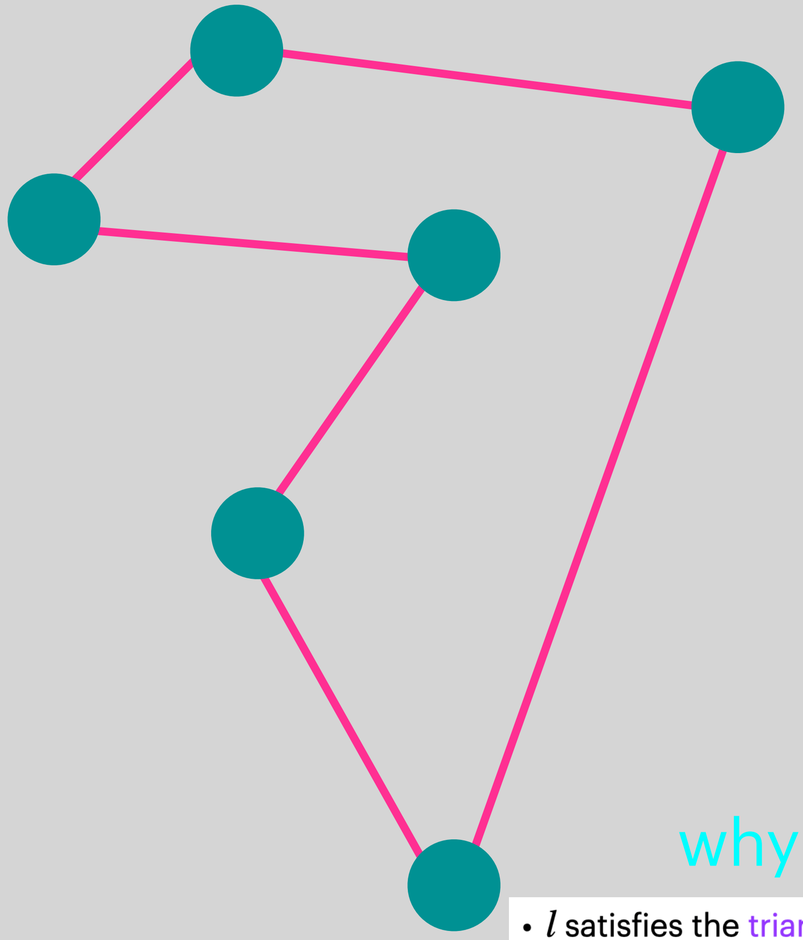
2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

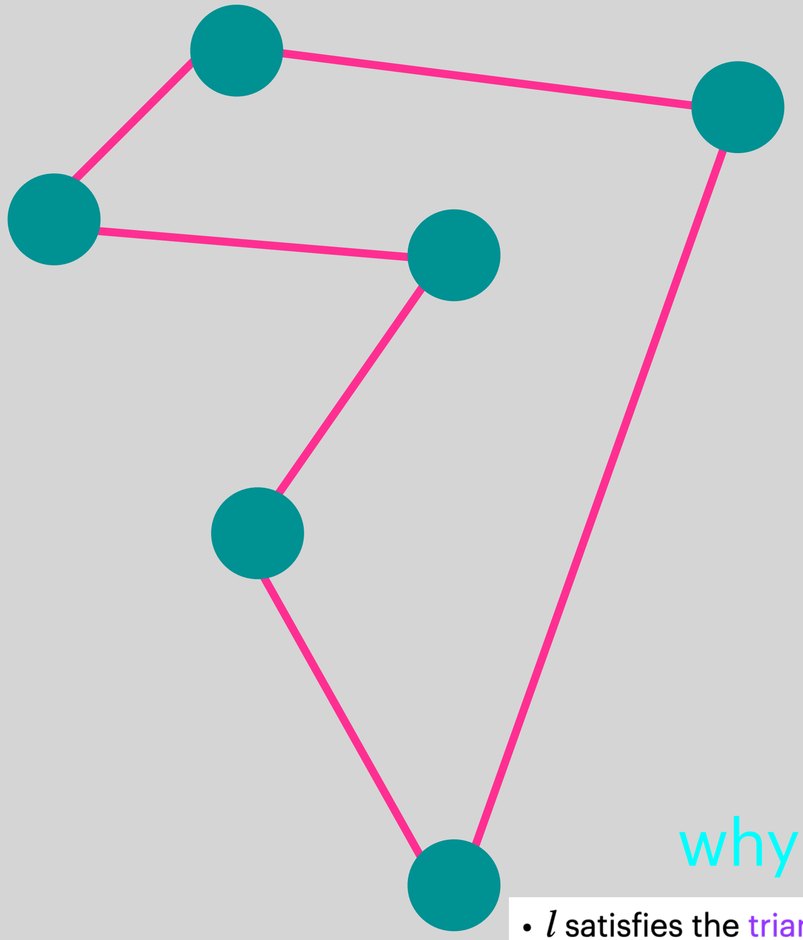
- l satisfies the triangle inequality

$$l(x, z) \leq l(x, y) + l(y, z)$$

$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality

$$l(x, z) \leq l(x, y) + l(y, z)$$

$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness

Goal :

Metric TSP : 2-Approximation

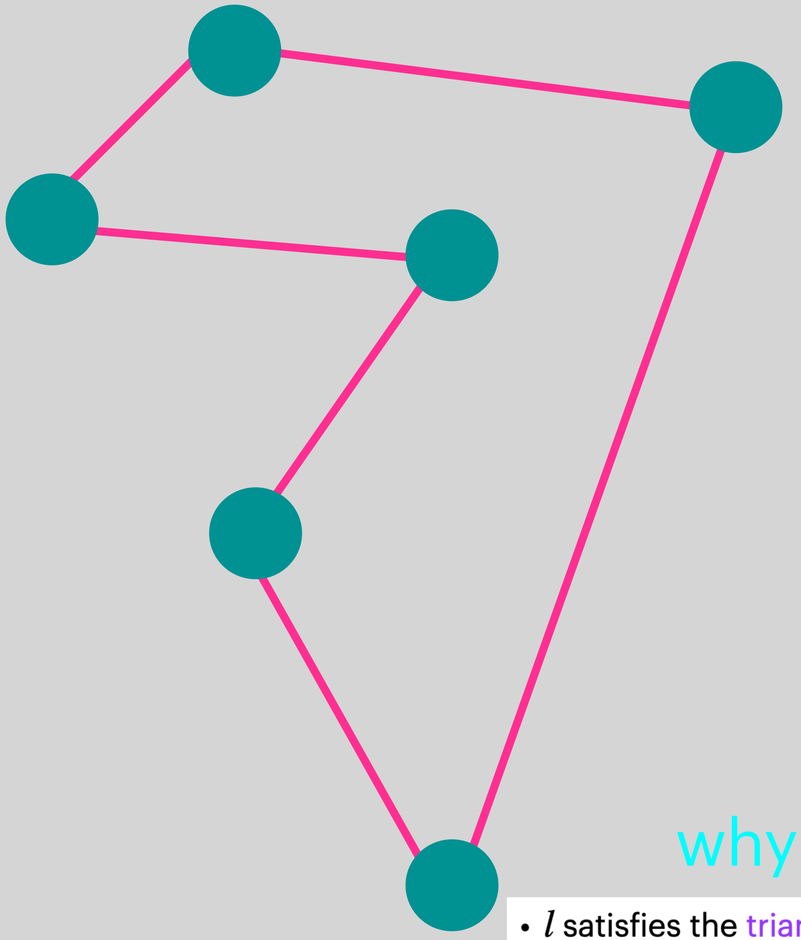
Problem Description

Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t. $l(C) \leq 2\, l(OPT)$

- l satisfies the triangle inequality $l(x,z) \leq l(x,y) + l(y,z)$

where $OPT = \min_{H\,:\,\text{Hamiltonian Cycle}} \sum_{e \in E(H)} l(e)$



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\, l(T) \leq 2\, OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\, l(T) \leq 2\, OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

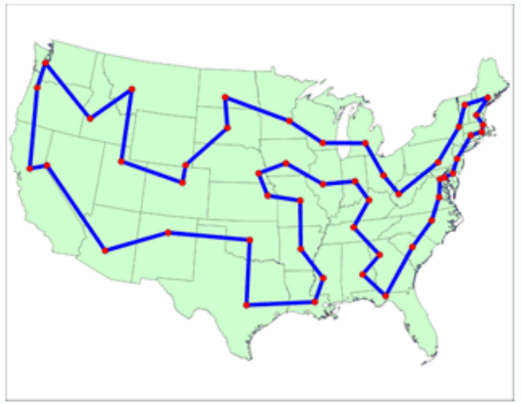
why ?

- l satisfies the triangle inequality $l(x,z) \leq l(x,y) + l(y,z)$

$$l(C) \leq l(W) = 2\, l(T) \leq 2\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Problem Description



Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $\quad l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t.

- l satisfies the triangle inequality

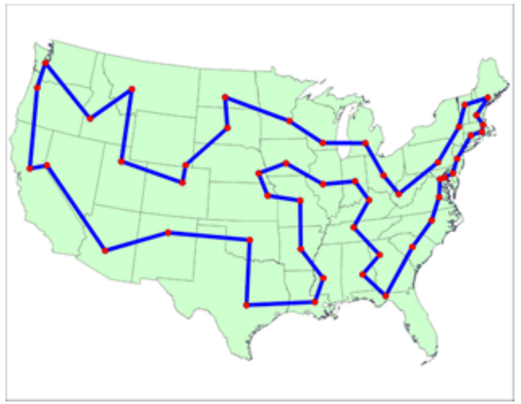
$$l(x,z) \leq l(x,y) + l(y,z)$$

$$l(C) \leq 1.5\ l(OPT)$$

where $\quad OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 1.5-Approximation

Problem Description



Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $\quad l : \binom{[n]}{2} \to R$

To find :
- Hamiltonian Cycle C s.t.

- l satisfies the triangle inequality

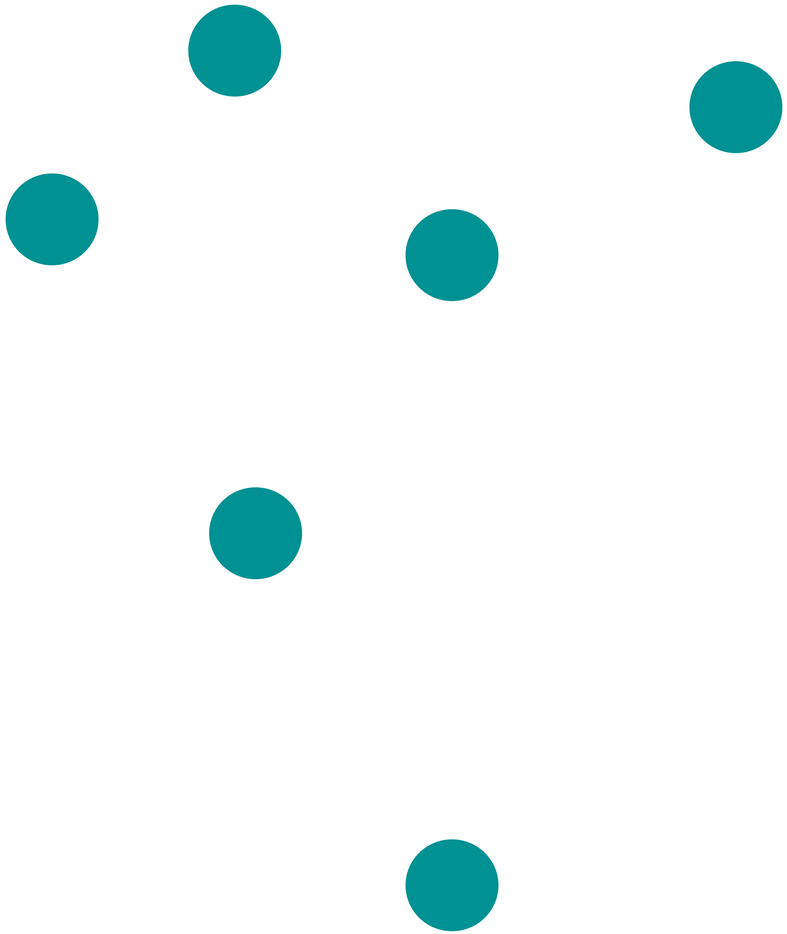
$$l(x,z) \leq l(x,y) + l(y,z)$$

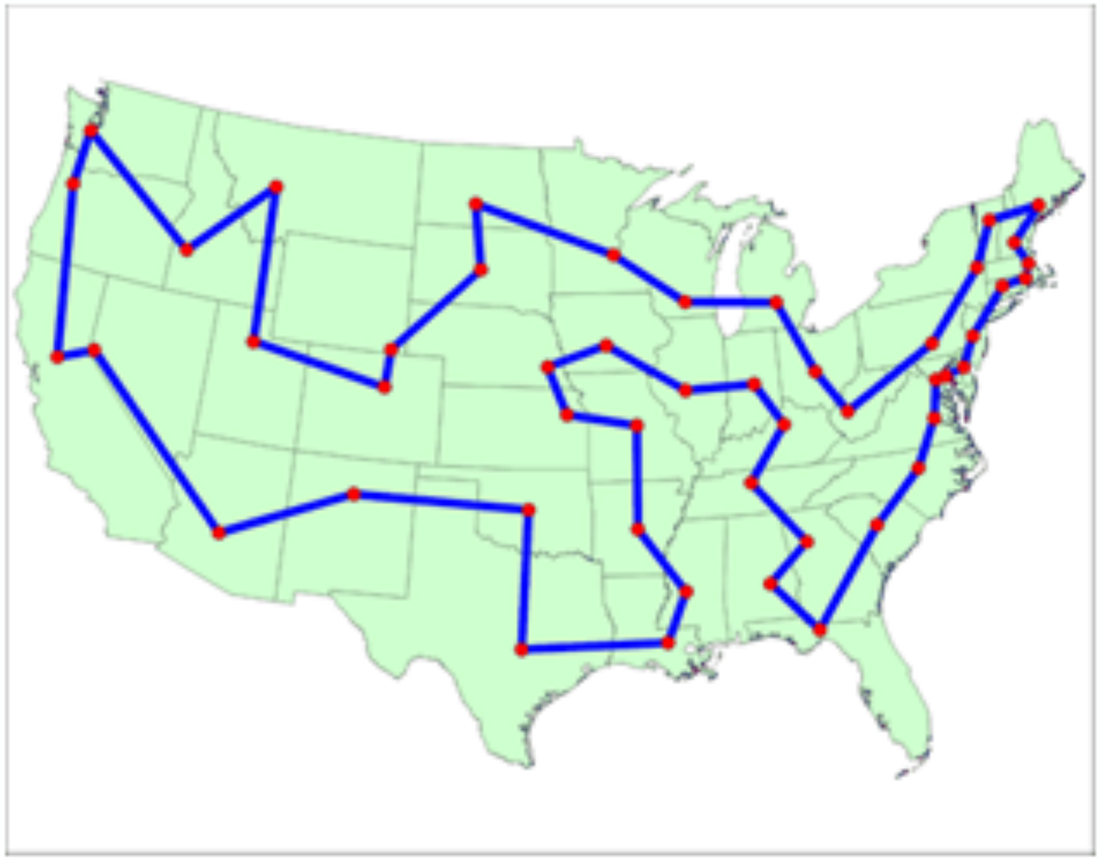
$$l(C) \leq 1.5\ l(OPT)$$

where $\quad OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 1.5-Approximation

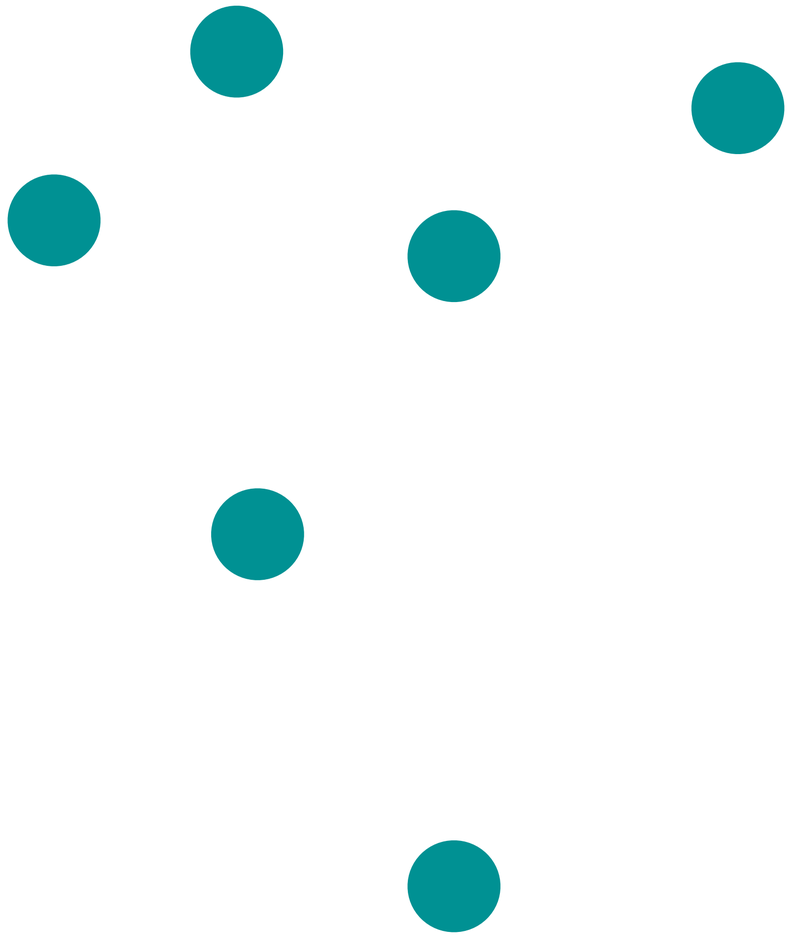
Algorithm





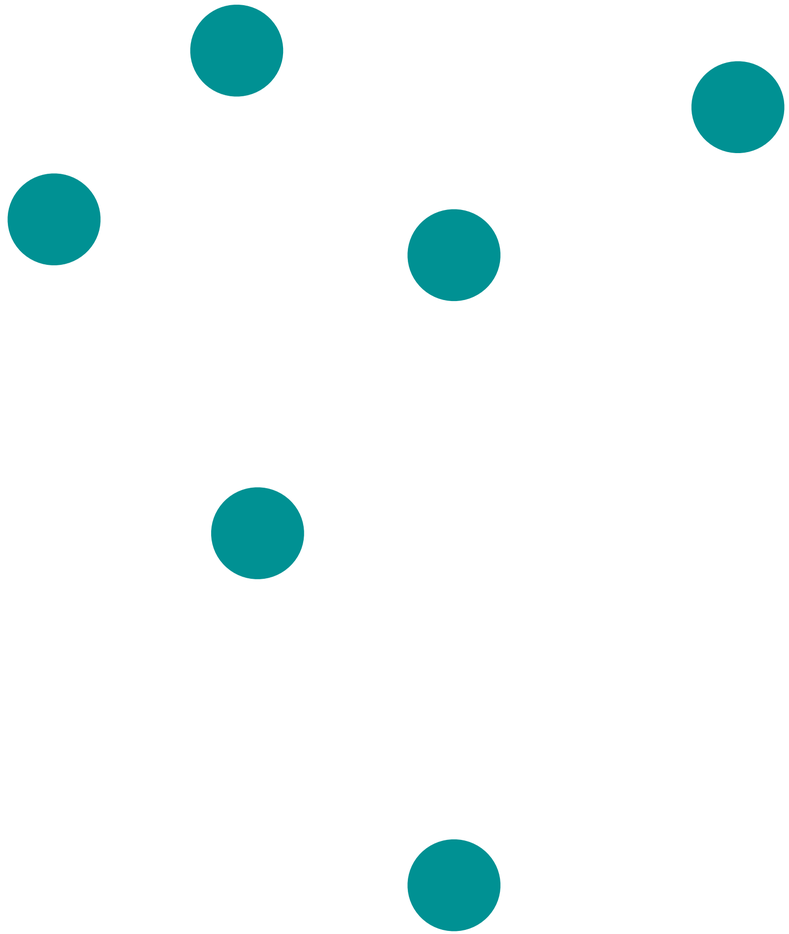
Metric TSP : 1.5-Approximation

Algorithm



Metric TSP : 1.5-Approximation

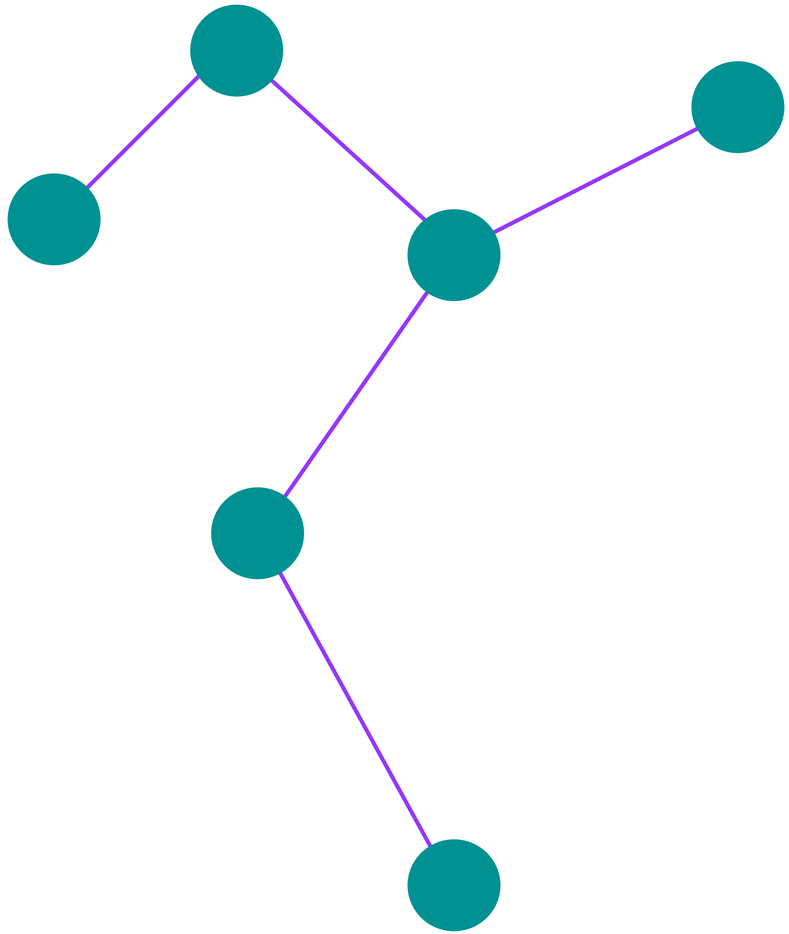
Algorithm



1. Find the MST T

Metric TSP : 1.5-Approximation

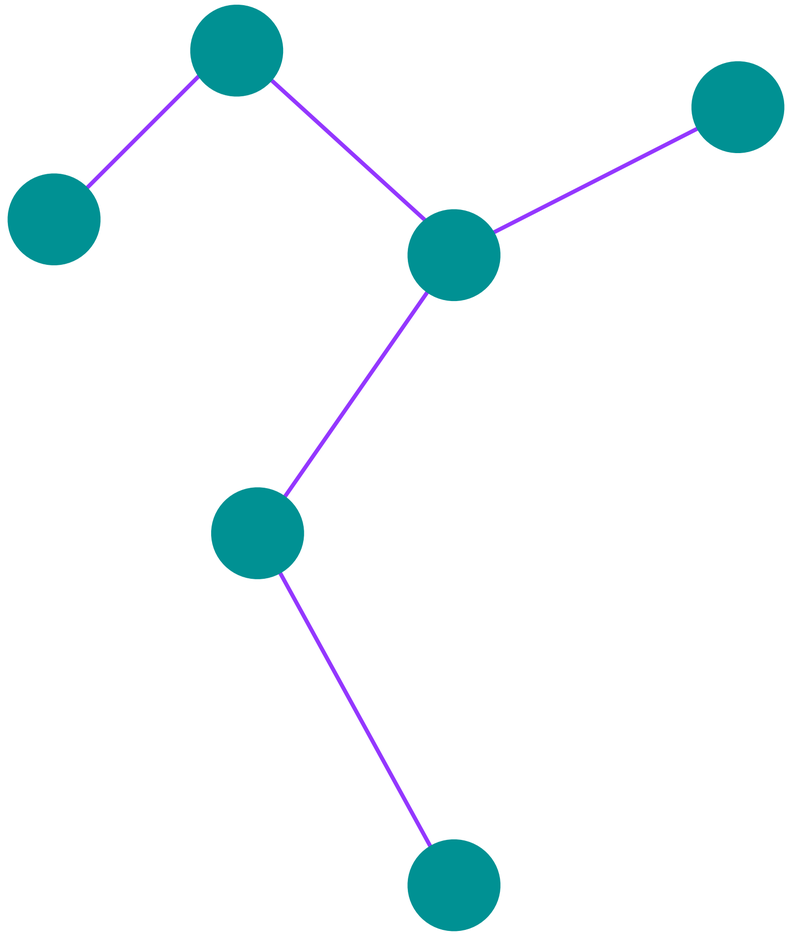
Algorithm



1. Find the MST T

Metric TSP : 1.5-Approximation

Algorithm

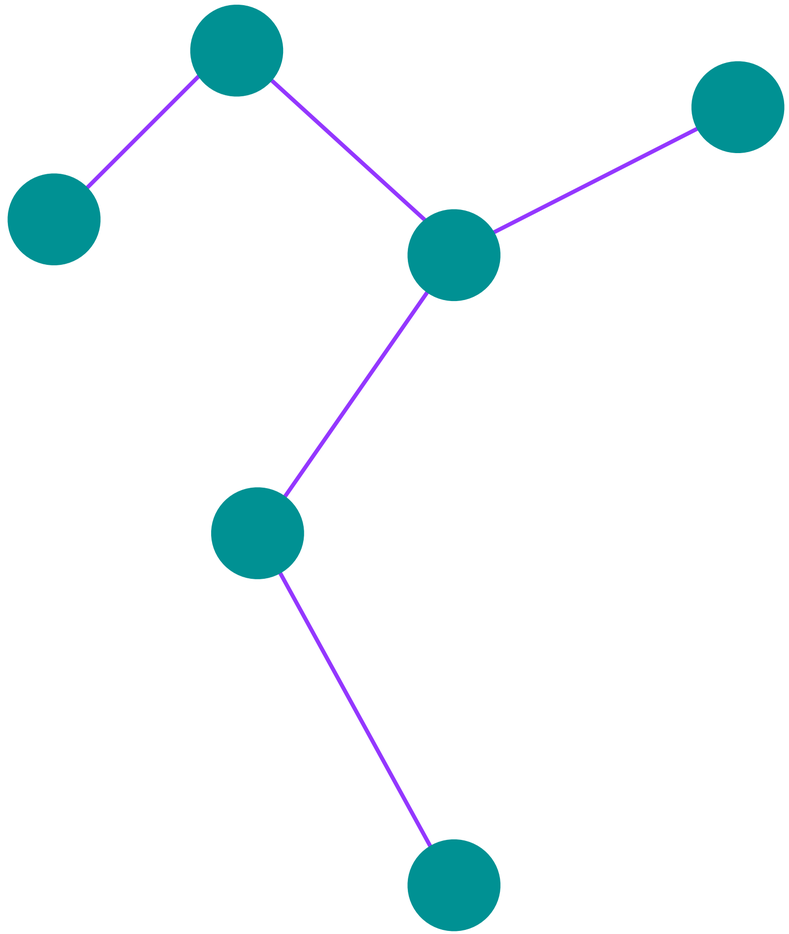


1. Find the MST T

$$l(T) \leq OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Algorithm



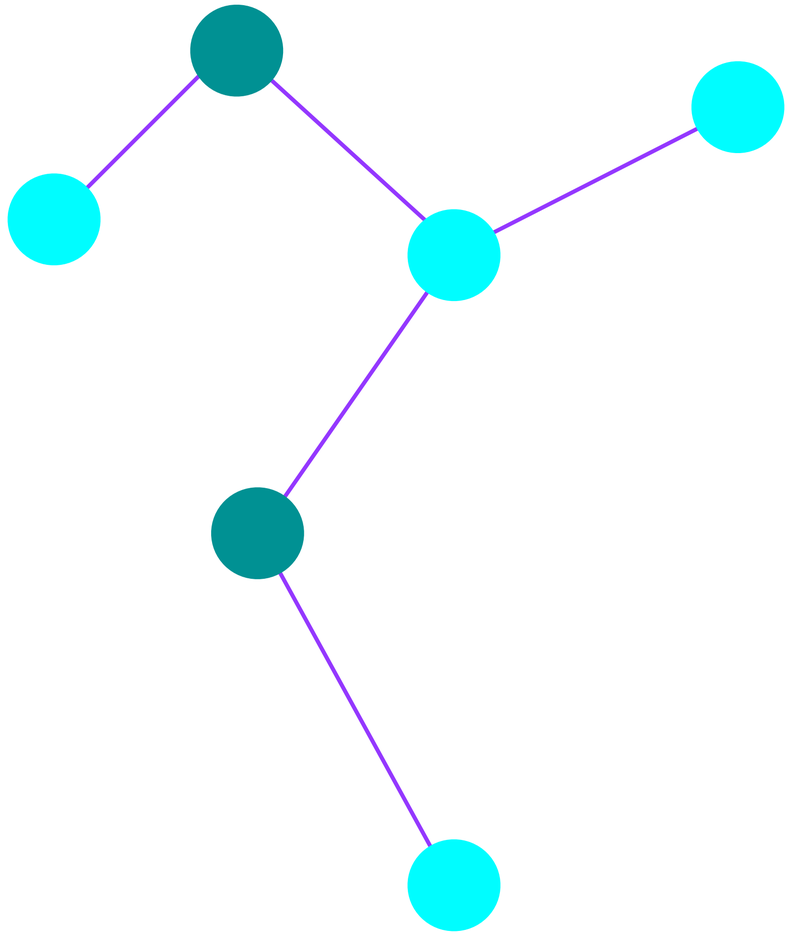
1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

Metric TSP : 1.5-Approximation

Algorithm



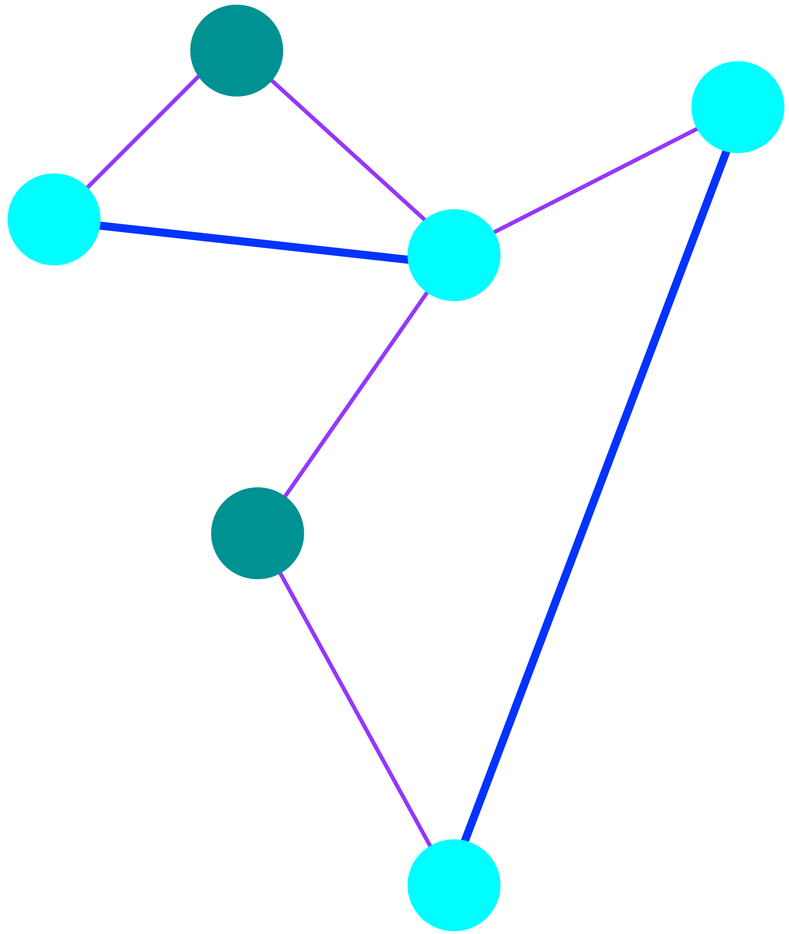
1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

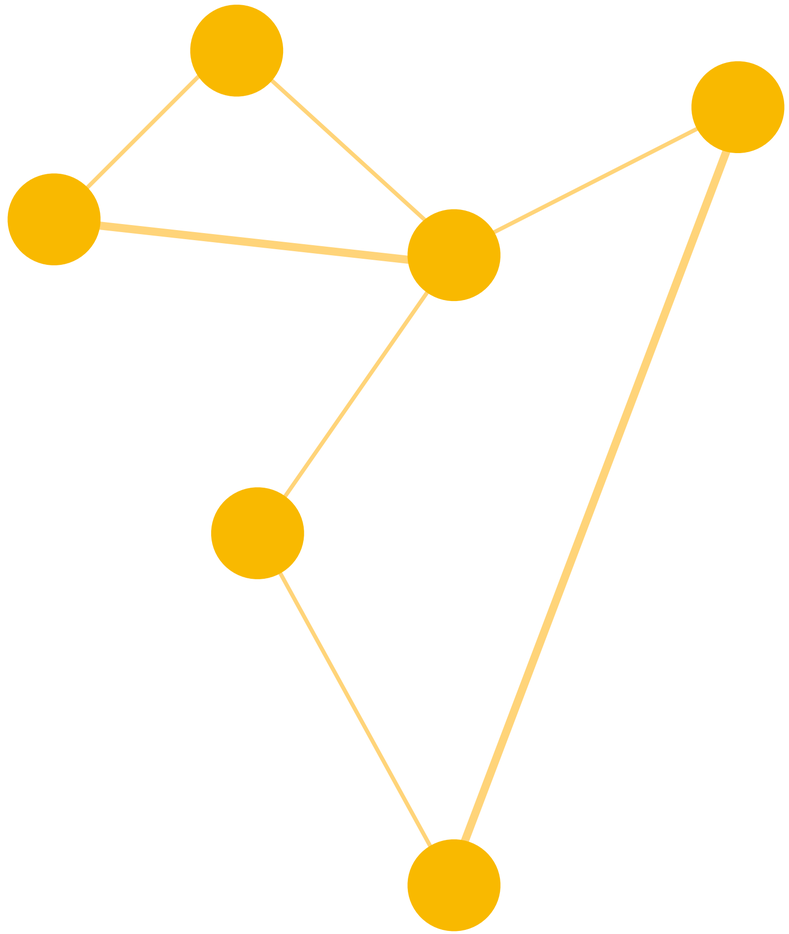
2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

$$l(M) \leq \frac{1}{2}\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2′. X:= Vertices with odd degree in T

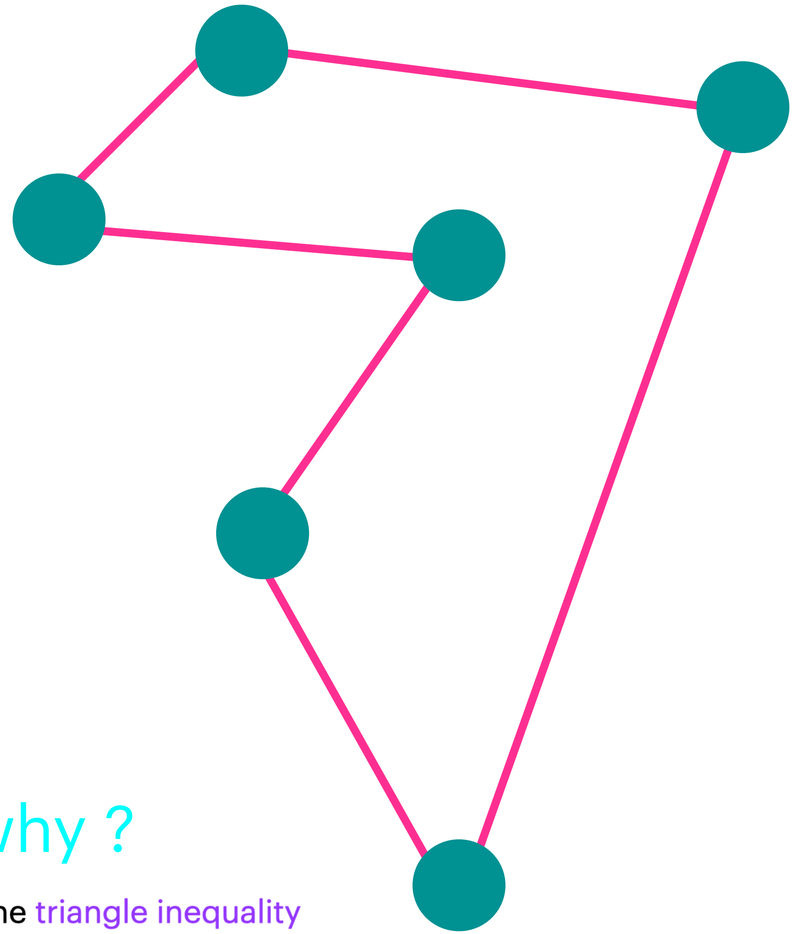
Find minimal Matching M for X

3. Find Eulerian Tour W $\quad l(M) \leq \frac{1}{2} OPT(K_n, l)$

$$l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2′. X:= Vertices with odd degree in T

Find minimal Matching M for X

$$l(M) \leq \frac{1}{2} OPT(K_n, l)$$

3. Find Eulerian Tour W

$$l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

$$l(C) \leq l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

why ?

- l satisfies the triangle inequality

$$l(x, z) \leq l(x, y) + l(y, z)$$

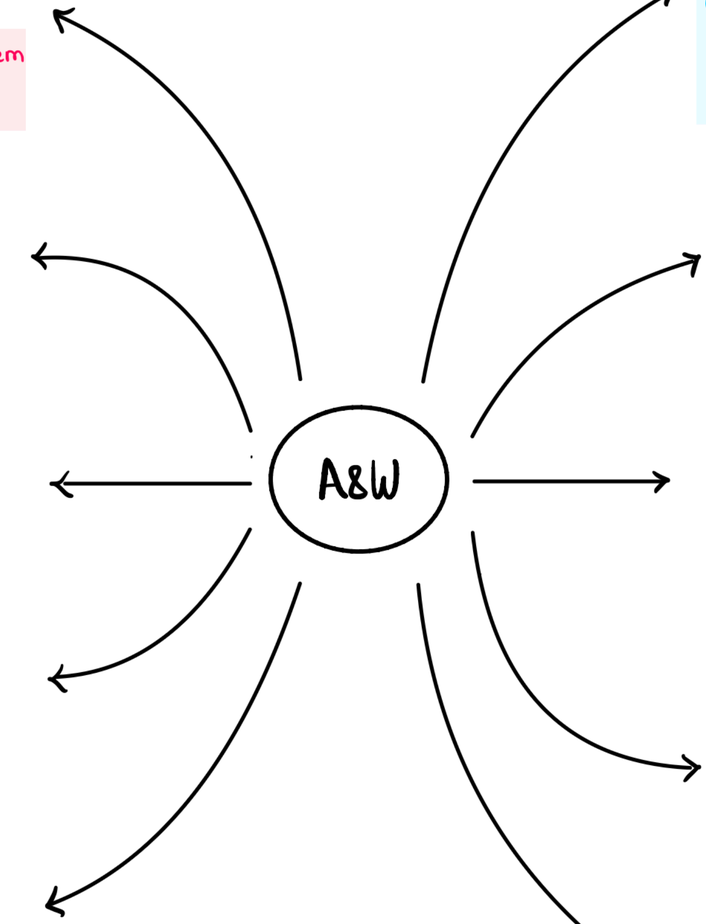
A&W

Exercise Session 5

Coloring

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- T1 Discussion
- Matching Kahoot

- Coloring

- Minitest 2 - coloring discussion

T1

Feedback + Discussion

- 1.c : Reflexivity argument accepted
- Watch out for the comments !

- Keep up the good work ! 👏👏👏

- Questions, issues ... Let me know !

Some Announcments

- Anki card approach changed (instead we have kahoots for now)
  - Matching kahoot this week
  - Cycles + TSP kahoot next week ...

- T2 (peer grading 1) ??
- Namings:
  - T1(theoretical exercise 1) , T2(peer grading 1), T3(theoretical exercise 2) ...

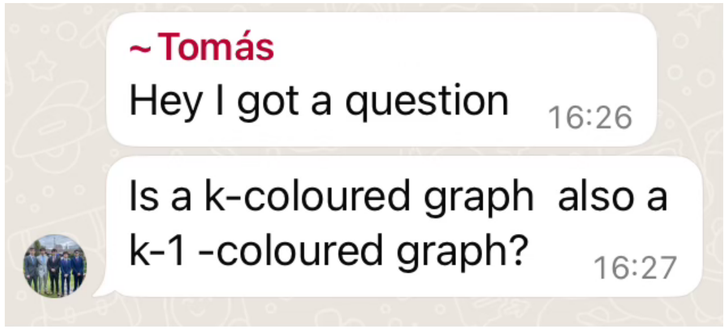
Matching Kahoot

Let’s take a break

Coloring

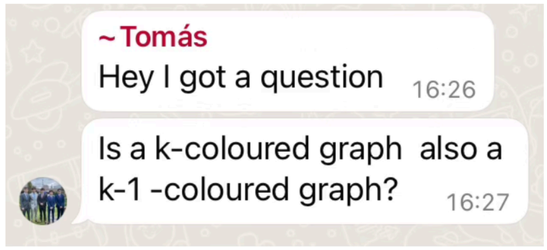
Coloring

Intuition



Coloring

Intuition



Matching

pairing adjacent vertices without conflicts

(pairing non-adjacent edges)

ensure that selected edges don’t touch the same vertex

edges are our friends

⚔️

Coloring

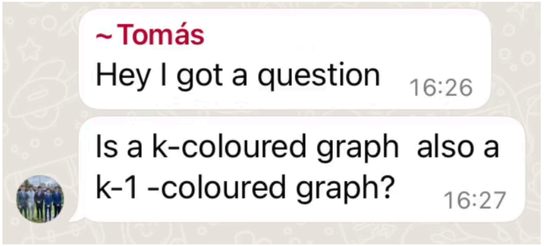
seperating adjacent vertices

ensure that the connected vertices have distinct colors

edges are our enemies

Coloring

Intuition



Matching

pairing adjacent vertices without conflicts

(pairing non-adjacent edges)

ensure that selected edges don’t touch the same vertex

edges are our friends

k-matched G is also (k-1)-matched

simply remove 1 edge

⚔️

Coloring

seperating adjacent vertices

ensure that the connected vertices have distinct colors

edges are our enemies

(k-1)-colored G is also k-colored

simply change one node’s color

Coloring

Definitions

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

Coloring

Examples

Color vertices in a way that no two vertices that share an edge are of the same color

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Is this a coloring ?

Coloring

Examples

Color vertices in a way that no two vertices that share an edge are of the same color

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Is this a coloring ?

✅

Coloring

Examples

Color vertices in a way that no two vertices that share an edge are of the same color

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Is this a coloring ?

✅

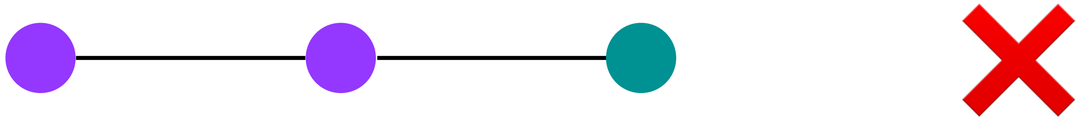
Coloring

Examples

Color vertices in a way that no two vertices that share an edge are of the same color

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Is this a coloring ?



Coloring

Definitions

Color vertices in a way that no two vertices that share an edge are of the same color

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$
- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite

Coloring

k-partite

- General version of the bipartite

- A graph $G = (V, E)$ is called k-partite if
  - the vertex set V can be divided into k disjoint sets $V = V_1 \cup V_2 \cup \ldots \cup V_k$
  - s.t. for every edge $(u, v) \in E$, u and v belong to different sets V_i and V_j where $i \neq j$

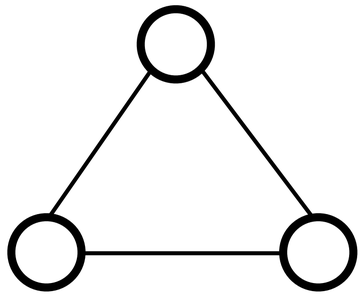
Coloring

Examples

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



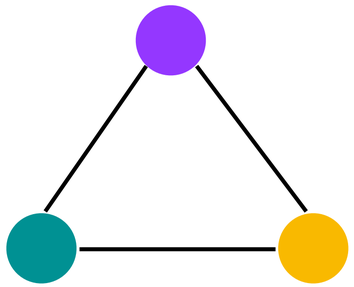
Coloring

Examples

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



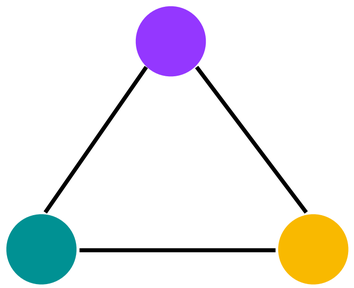
$$\chi(G_1) = 3$$

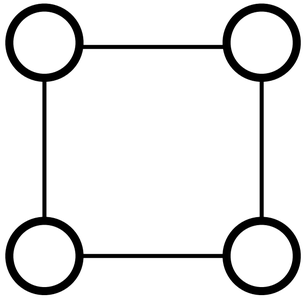
Coloring
Examples

- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite





$\chi(G_1) = 3$

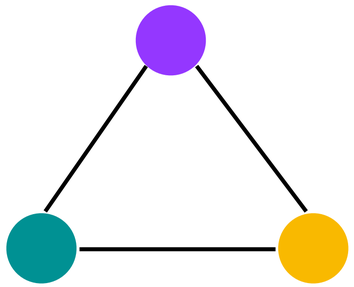
Coloring

Examples

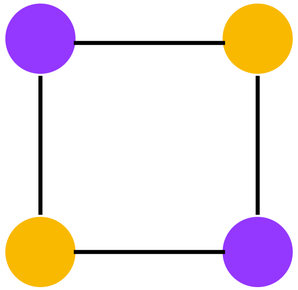
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



$\chi(G_1) = 3$



$\chi(G_1) = 2$

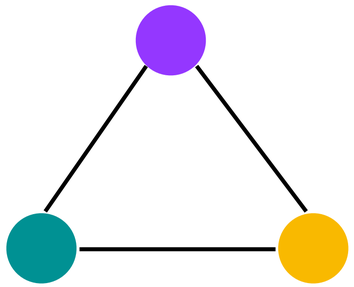
Coloring

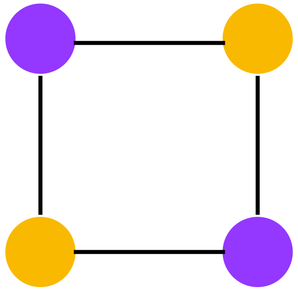
Examples

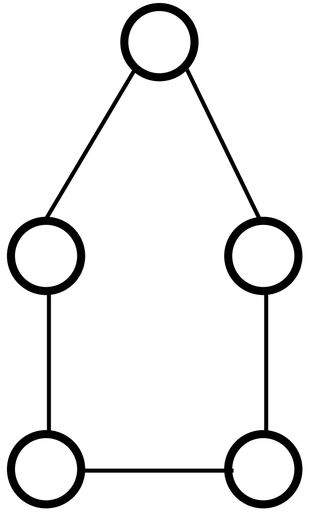
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite







$\chi(G_1) = 3$

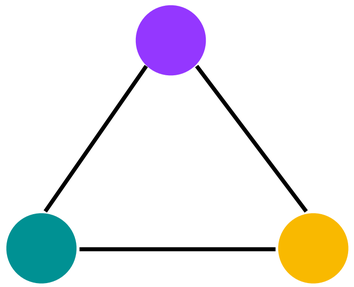
$\chi(G_1) = 2$

Coloring
Examples

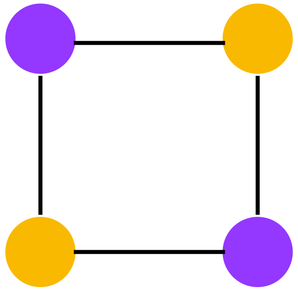
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

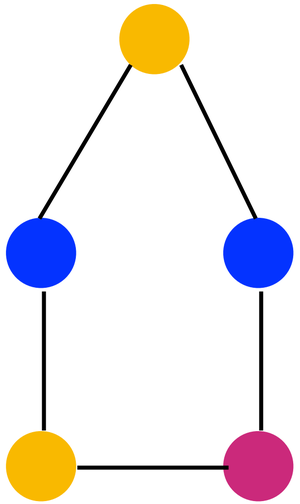
- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



$\chi(G_1) = 3$



$\chi(G_1) = 2$



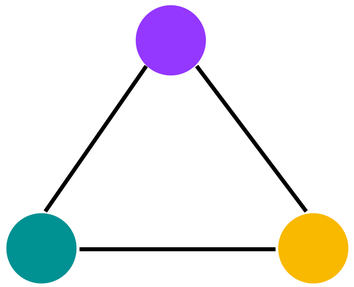
$\chi(G_1) = 3$

Coloring
Examples

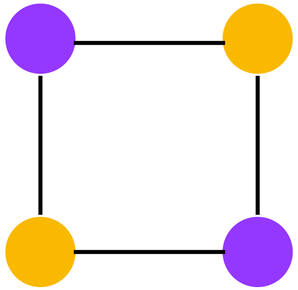
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

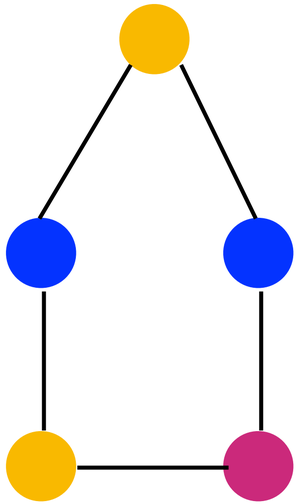
- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



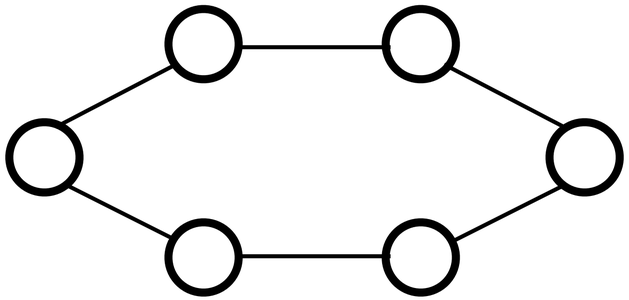
$\chi(G_1) = 3$



$\chi(G_1) = 2$



$\chi(G_1) = 3$

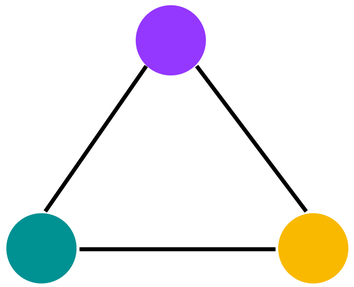


Coloring
Examples

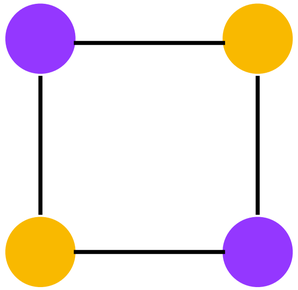
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

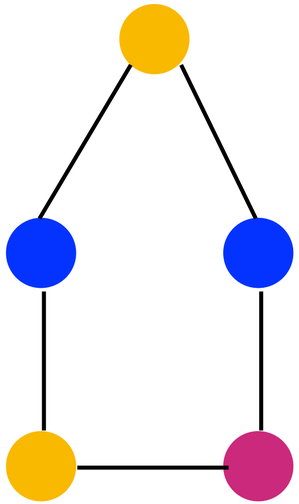
- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



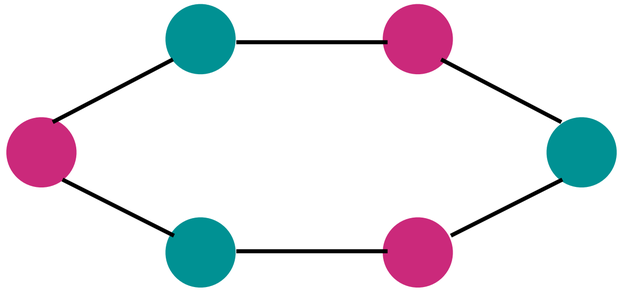
$\chi(G_1) = 3$



$\chi(G_1) = 2$



$\chi(G_1) = 3$



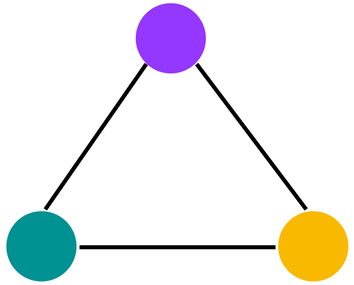
$\chi(G_1) = 2$

Coloring
Examples

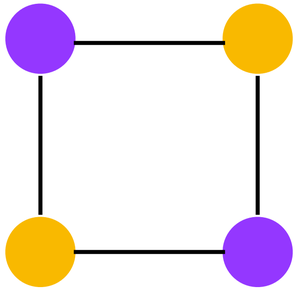
- (Vertex-) Coloring :
  - A (vertex-) coloring of $G = (V, E)$ with k colors is a mapping $c : V \to [k]$ s.t. $c(u) \neq c(v)$ for all edges $\{u, v\} \in E$

Color vertices in a way that no two vertices that share an edge are of the same color

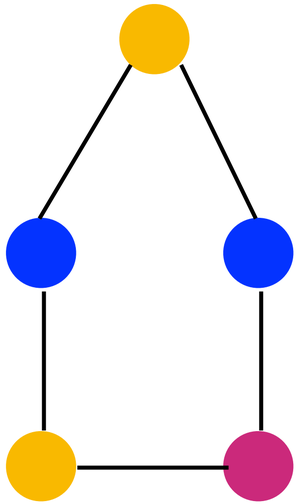
- Chromatic Number :
  - The chromatic number $\chi(G)$ is the minimum number of colors needed to color a graph
  - equivalent : $\chi(G) \leq k \iff G$ is k-partite



$\chi(G_1) = 3$

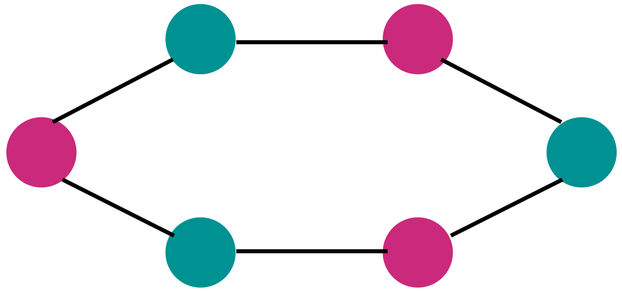


$\chi(G_1) = 2$



$\chi(G_1) = 3$

Do you notice something ?



$\chi(G_1) = 2$

Coloring
Problem

For all $k \geq 3$, given a graph $G = (V, E)$,

is $\chi(G) \leq k$?

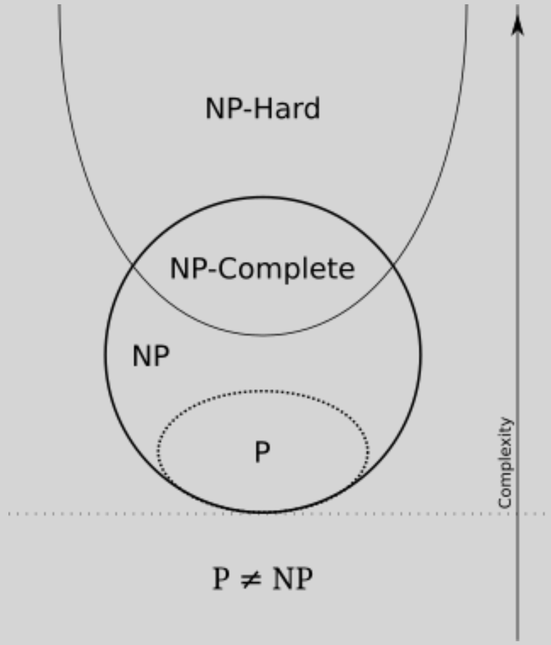
NP - Complete

NP-Complete

For all $k \geq 3$, given a graph $G = (V, E)$, is $\chi(G) \leq k$?

Complexity Theory

TI next semester



P : polynomial

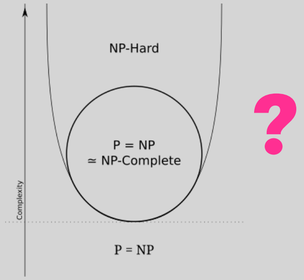
NP : non-deterministic polynomial

- NP is the set of decision problems *solvable* in polynomial time by a nondeterministic Turing machine.
- NP is the set of decision problems *verifiable* in polynomial time by a deterministic Turing machine.

NP - Complete

A problem a in NP is NP-complete if :

$$a \in P \implies P = NP$$



?

Coloring

Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ ← color the first vertex with color 1

- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

Coloring
Greedy Algorithm

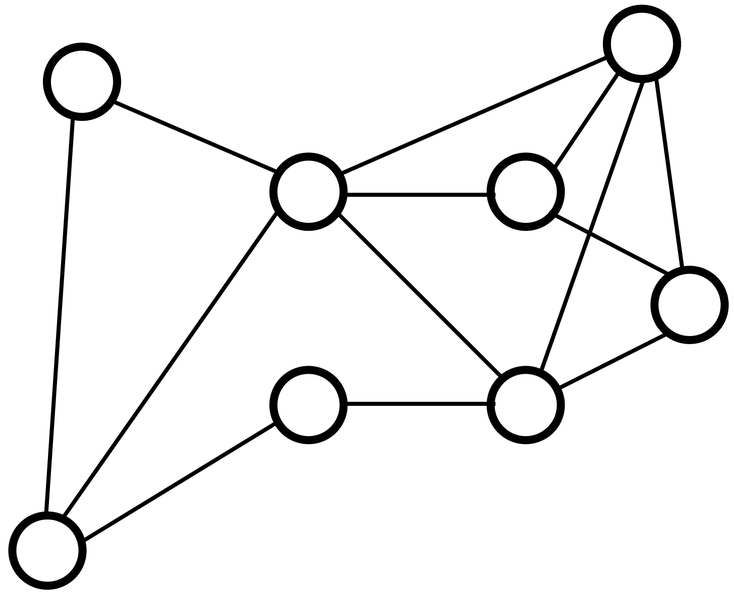
- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

C :

i	1	2	3	4	5	6	7	8
$c[v_i]$								



Coloring
Greedy Algorithm

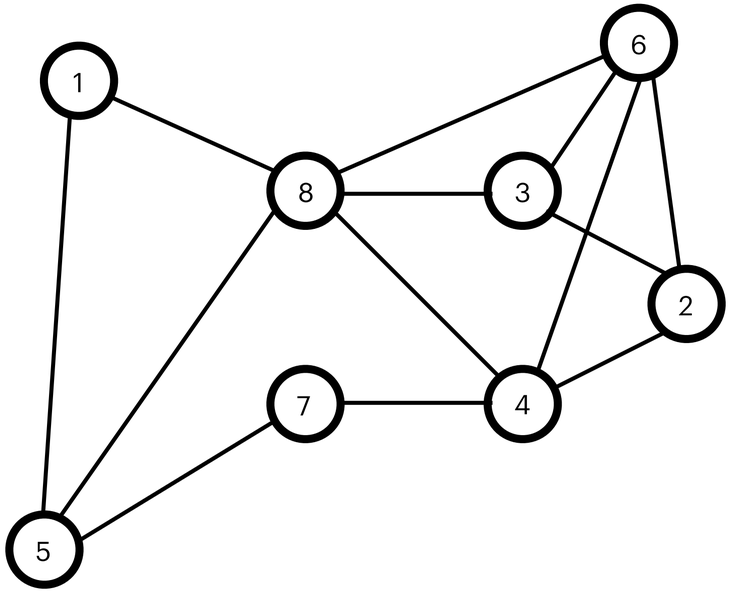
- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

C :

i	1	2	3	4	5	6	7	8
$c[v_i]$								



Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

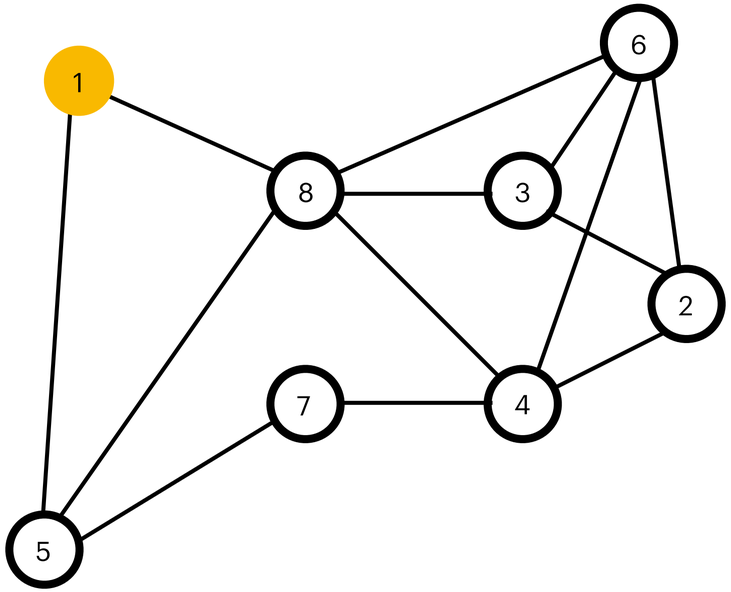
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

k = 1

C :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1							



Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

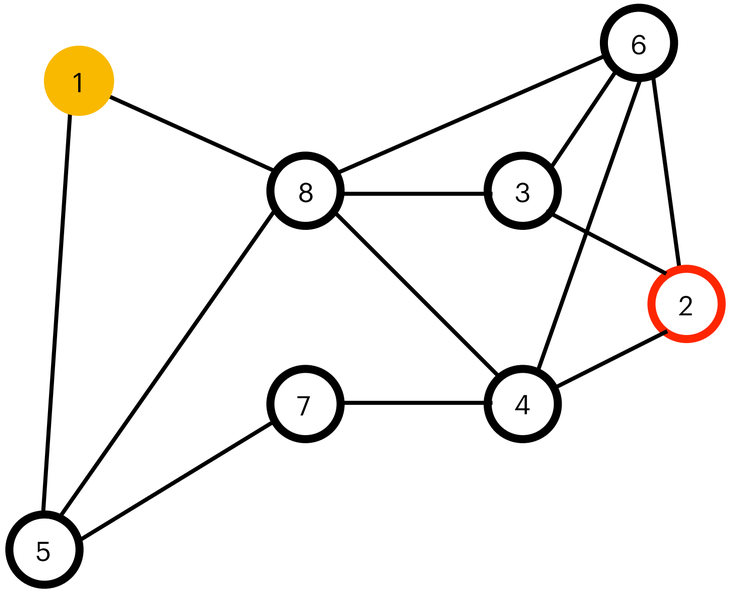
k = 1 🟡

considering ks

🟡 ✅

C :

i	1	2	3	4	5	6	7	8
c[v_i]	1							



Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

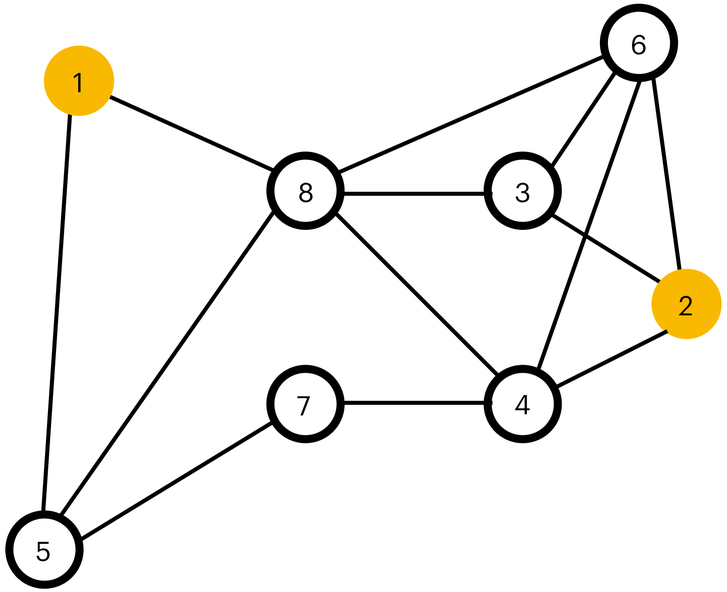
k = 1 🟡

considering ks

🟡 ✅

C :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1	1						



Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

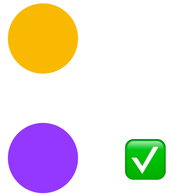
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

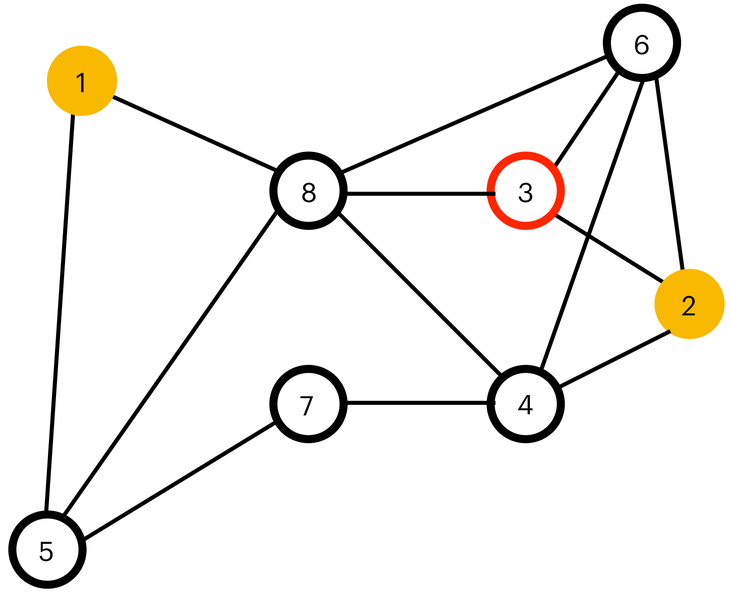
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1	1						

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

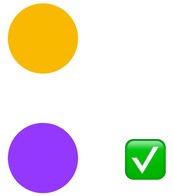
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

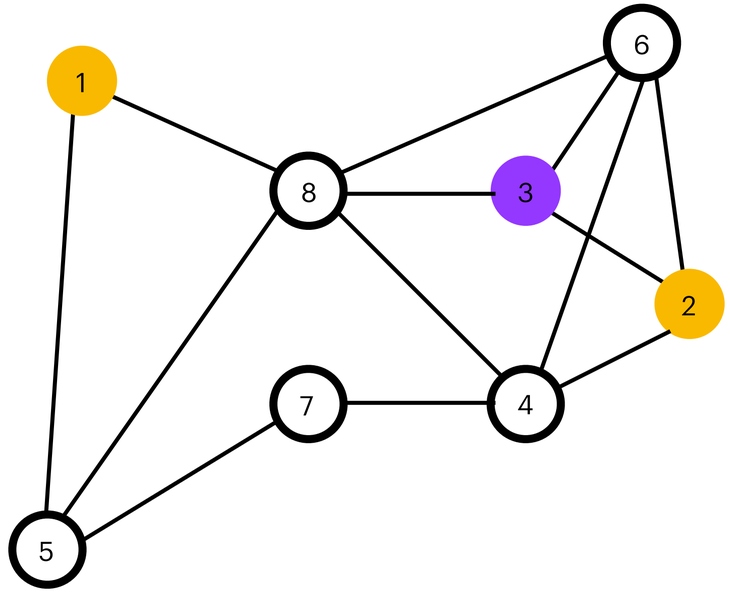
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2					

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

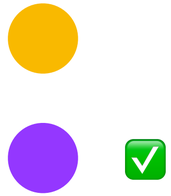
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

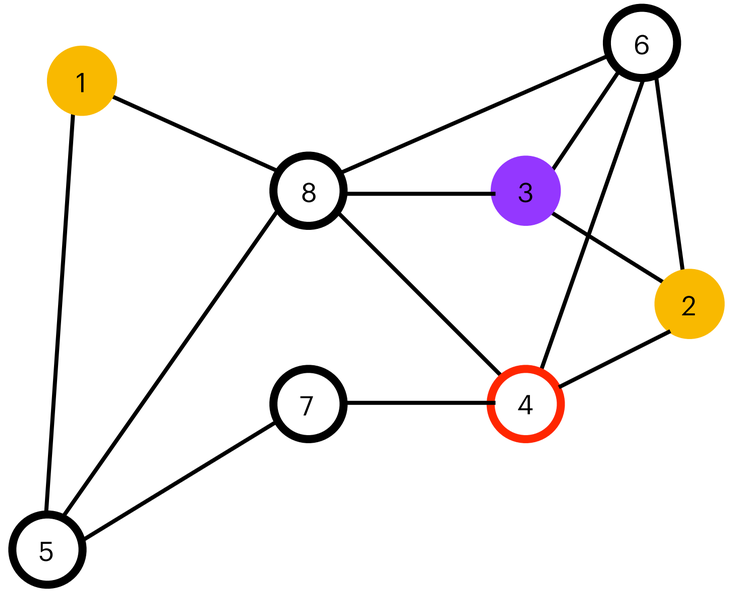
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1	1	2					

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

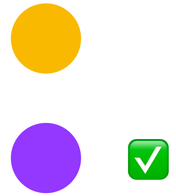
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

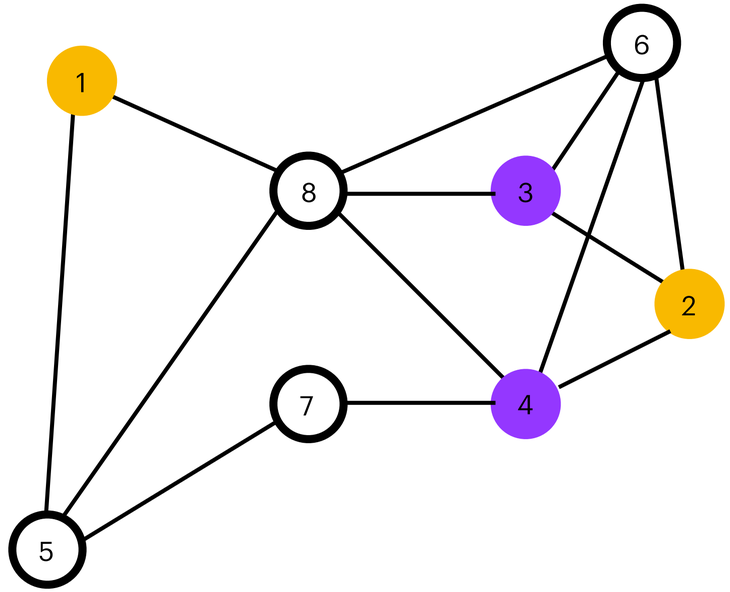
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2				

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

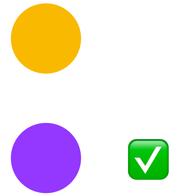
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

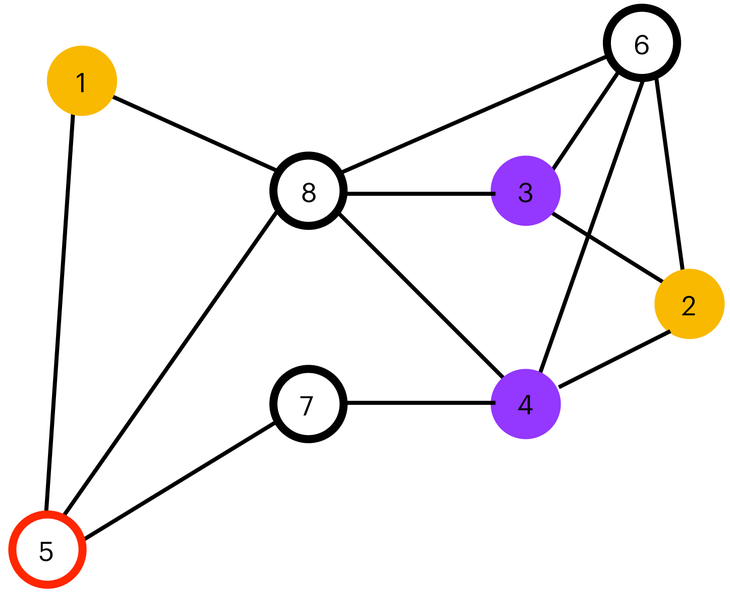
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2				

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

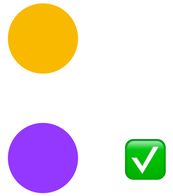
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

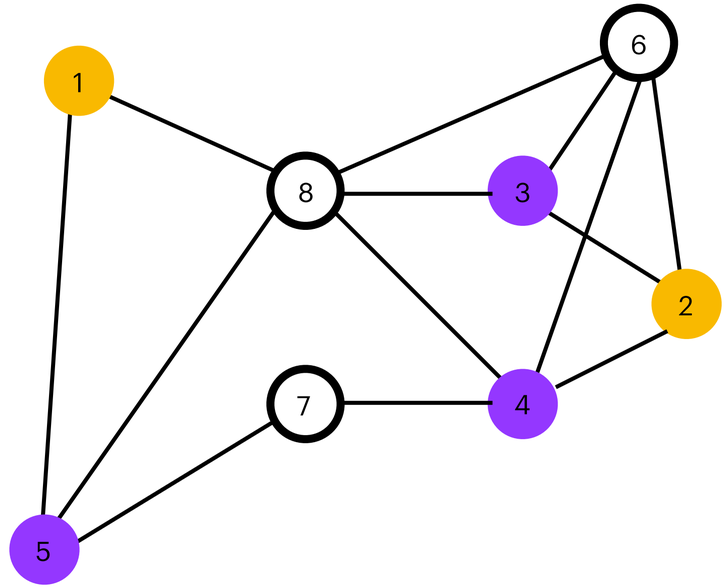
k indexing for colors :

k = 1

k = 2

considering ks





c :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2	2			

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

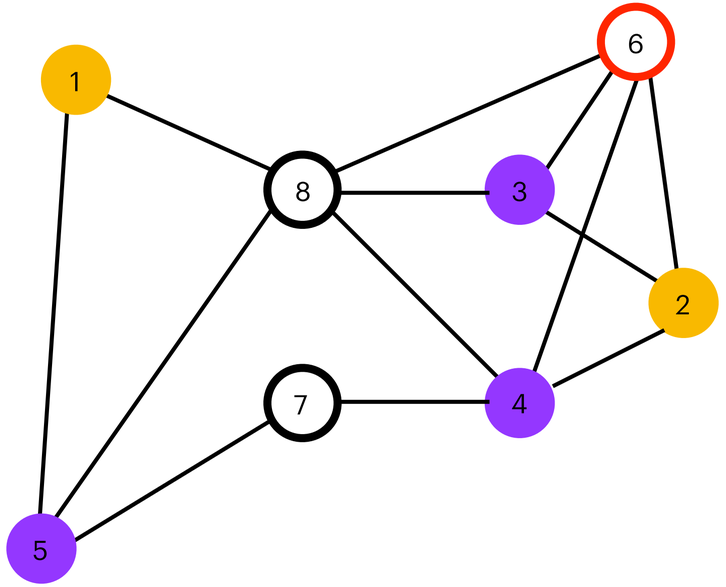
k = 1

k = 2

k = 3

considering ks

✅



C :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2	2			

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

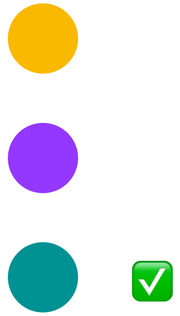
k indexing for colors :

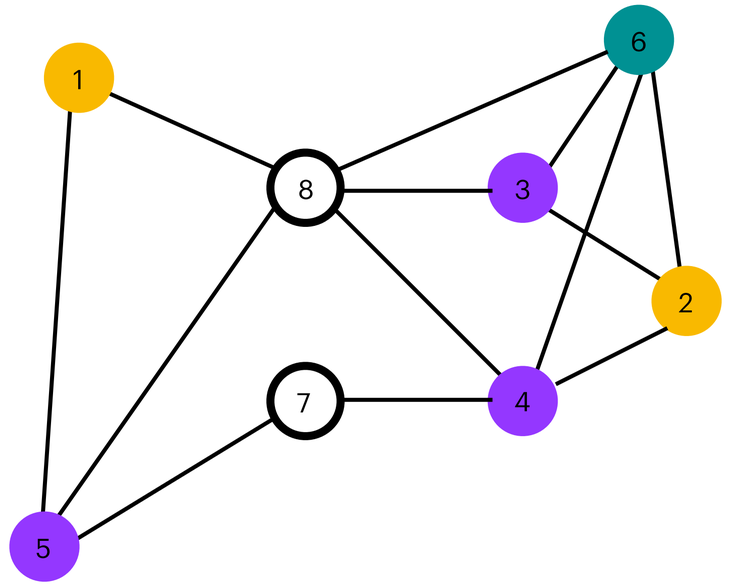
k = 1

k = 2

k = 3

considering ks





C :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1	1	2	2	2	3		

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

k = 1

k = 2

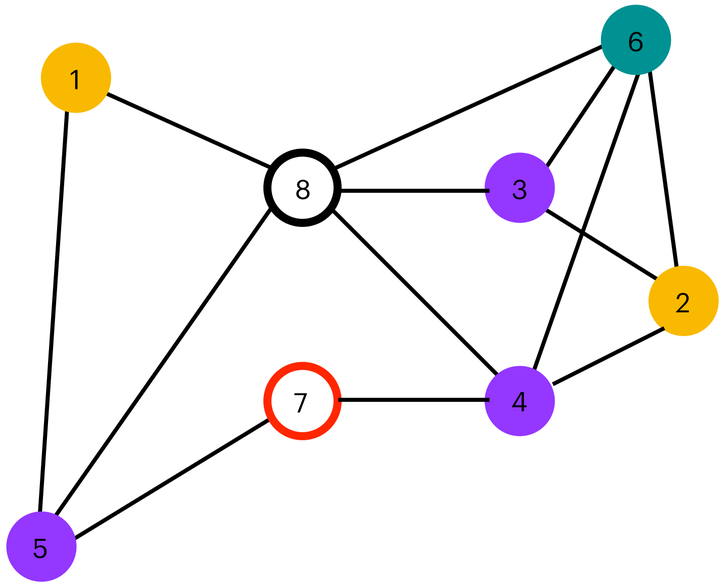
k = 3

considering ks

✅

C :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2	2	3		



Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

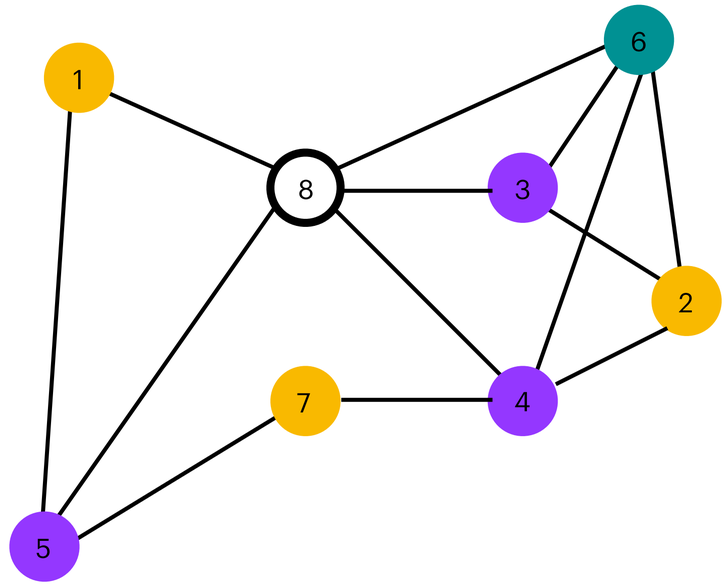
k indexing for colors :

k = 1

k = 2

k = 3

considering ks



C :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2	2	3	1	

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

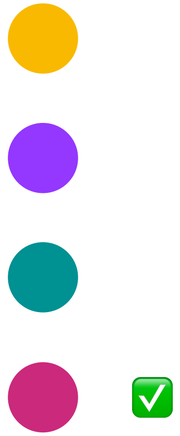
k = 1

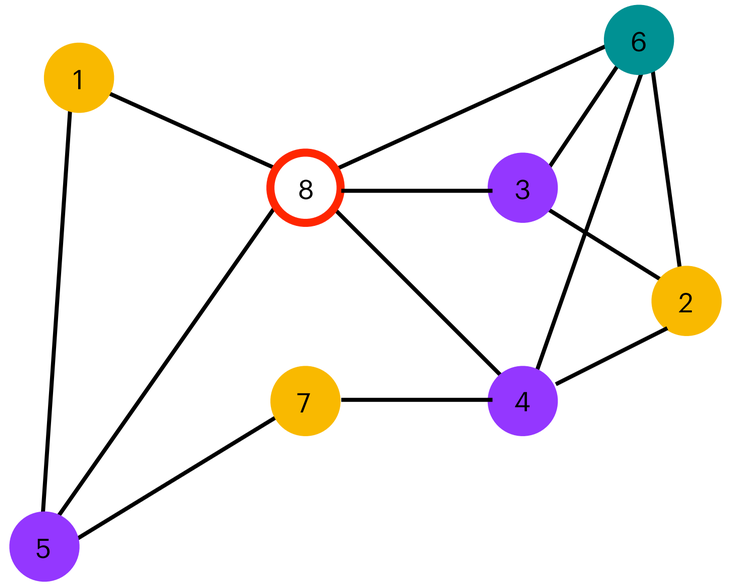
k = 2

k = 3

k = 4

considering ks





C :

i	1	2	3	4	5	6	7	8
$c[v_i]$	1	1	2	2	2	3	1	

Coloring
Greedy Algorithm

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

k indexing for colors :

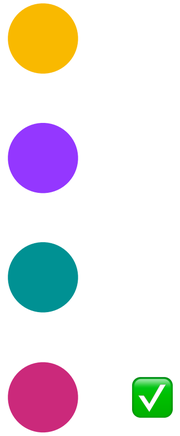
k = 1

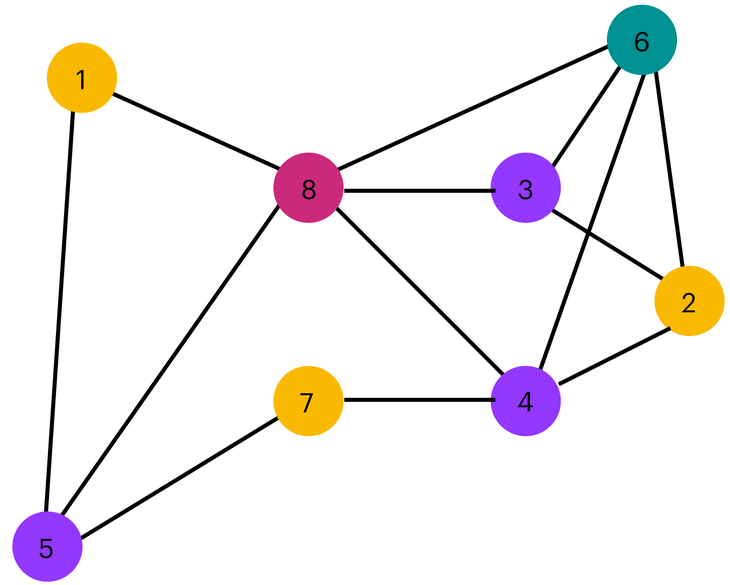
k = 2

k = 3

k = 4

considering ks





C :

i	1	2	3	4	5	6	7	8
c[v_i]	1	1	2	2	2	3	1	4

done

Coloring

Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors

$\Delta(G) :=$ maximum degree in G

Coloring

Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors $\quad \chi(G) \leq k \iff G$ is k-partite

$\Delta(G) :=$ maximum degree in G

Coloring

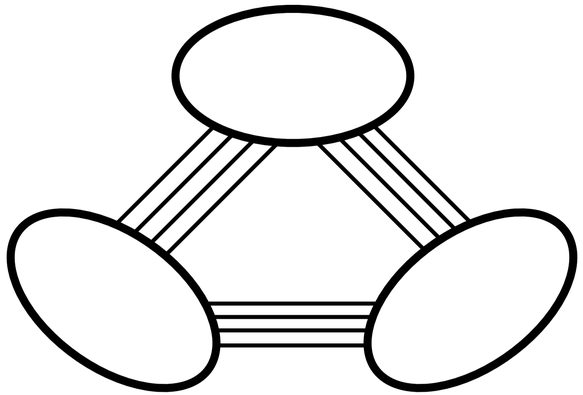
Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors

$\chi(G) \leq k \iff G$ is k-partite



$\Delta(G) :=$ maximum degree in G

Coloring

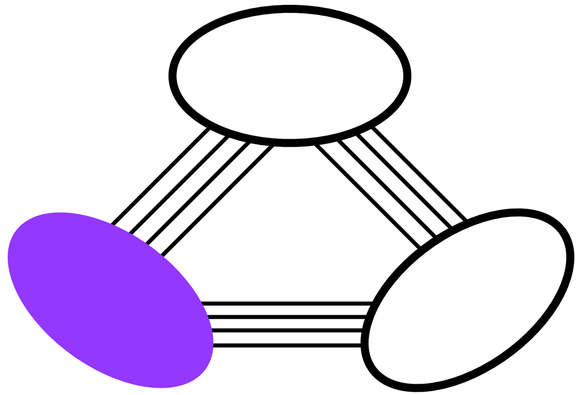
Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors

$\chi(G) \leq k \iff G$ is k-partite



$\Delta(G) :=$ maximum degree in G

Coloring

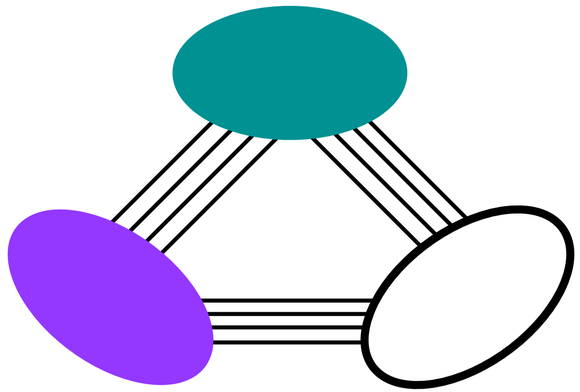
Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors

$\chi(G) \leq k \iff G$ is k-partite



$\Delta(G) :=$ maximum degree in G

Coloring

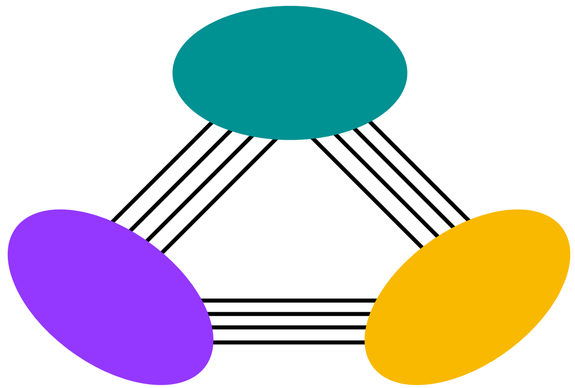
Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors

$\chi(G) \leq k \iff G$ is k-partite



$\Delta(G) :=$ maximum degree in G

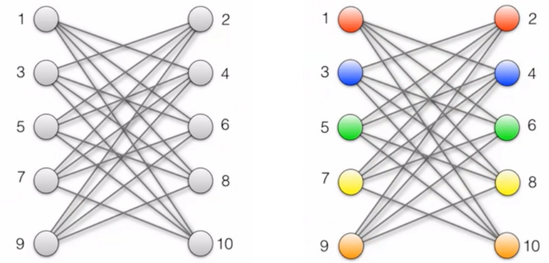
Coloring

Greedy Algorithm - Observations

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- For all orders of vertices $V = \{v_1, \ldots, v_n\}$
  - the Greedy Algorithm needs $\Delta(G)$ +1 colors
- There exists an order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $\chi(G)$ colors
- There exists bipartite Graphs and order of vertices $V = \{v_1, \ldots, v_n\}$ for which
  - the Greedy Algorithm needs $|V| \,/\, 2$ colors



Coloring

Greedy Algorithm - Observations

- For the chosen order of vertices $V = \{v_1, \ldots, v_n\}$ s.t.
  - $|N(v_i) \cap \{v_1, \ldots, v_{i-1}\}| \leq k \qquad \forall\, 2 \leq i \leq n$
  - the Greedy Algorithm needs at most k +1 colors

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
- $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

k gets increased here →

min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

Heuristic Meaning

🔎 Heuristic in Algorithms:

A **heuristic** in algorithms refers to a **problem-solving technique** that:

- Uses a **practical approach** to find a solution quickly.
- Sacrifices **accuracy** or **completeness** for **speed**.
- Aims for a **good enough** solution rather than the **perfect** one.
- Often relies on **experience, intuition, or patterns** rather than formal mathematical proofs.

Coloring

Heuristic + Greedy Algorithm

- For the chosen order of vertices $V = \{v_1, \ldots, v_n\}$ s.t.
  - $|N(v_i) \cap \{v_1, \ldots, v_{i-1}\}| \leq k \quad \forall 2 \leq i \leq n$
  - the Greedy Algorithm needs at most k +1 colors

- Pick the order of the vertices using the heuristic : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$
    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- Heuristic :
  - v_n := Vertex with the smallest degree. Delete v_n
  - v_{n-1} := Vertex with the smallest degree in the remaining G. Delete v_{n-1}
  - Iterate

Coloring

Heuristic + Greedy Algorithm - Observations

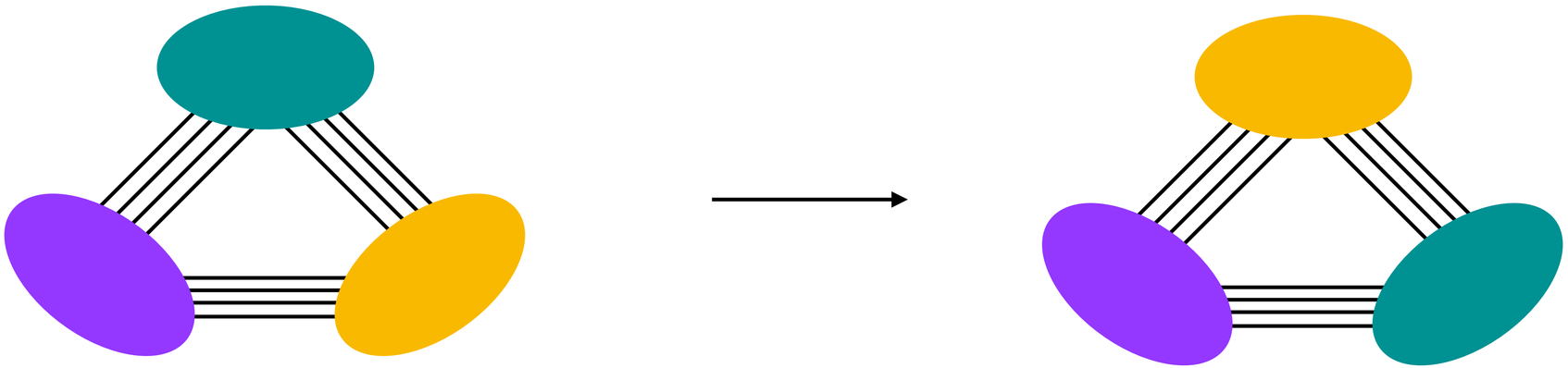
- If in every subgraph of G , there exists a vertex with degree $\leq$ k
  - heuristic provides an order $v_1, \ldots v_n$ s.t. the Greedy Algorithm needs k+1 colors

- For trees heuristic+greedy finds a coloring with 2 colors
- For planar graphs heuristic+greedy finds a coloring with $\leq$ 6 colors
- If G is connected and there exists $v \in G$ with deg(v) < $\Delta(G)$ ⟶ it doesn't hold only when the graph is regular: $\forall v \in V$ (deg(v) = $\Delta(G)$)

  heuristic (or bfs/dfs)+ greedy finds a coloring with $\leq \Delta(G)$ colors
- If the G is 3-colorable, then one can color it in O(|V| + |E|) time with O($\sqrt{|V|}$) colors

Coloring

Swapping Color Classes Trick



Articulation Points and Bridges

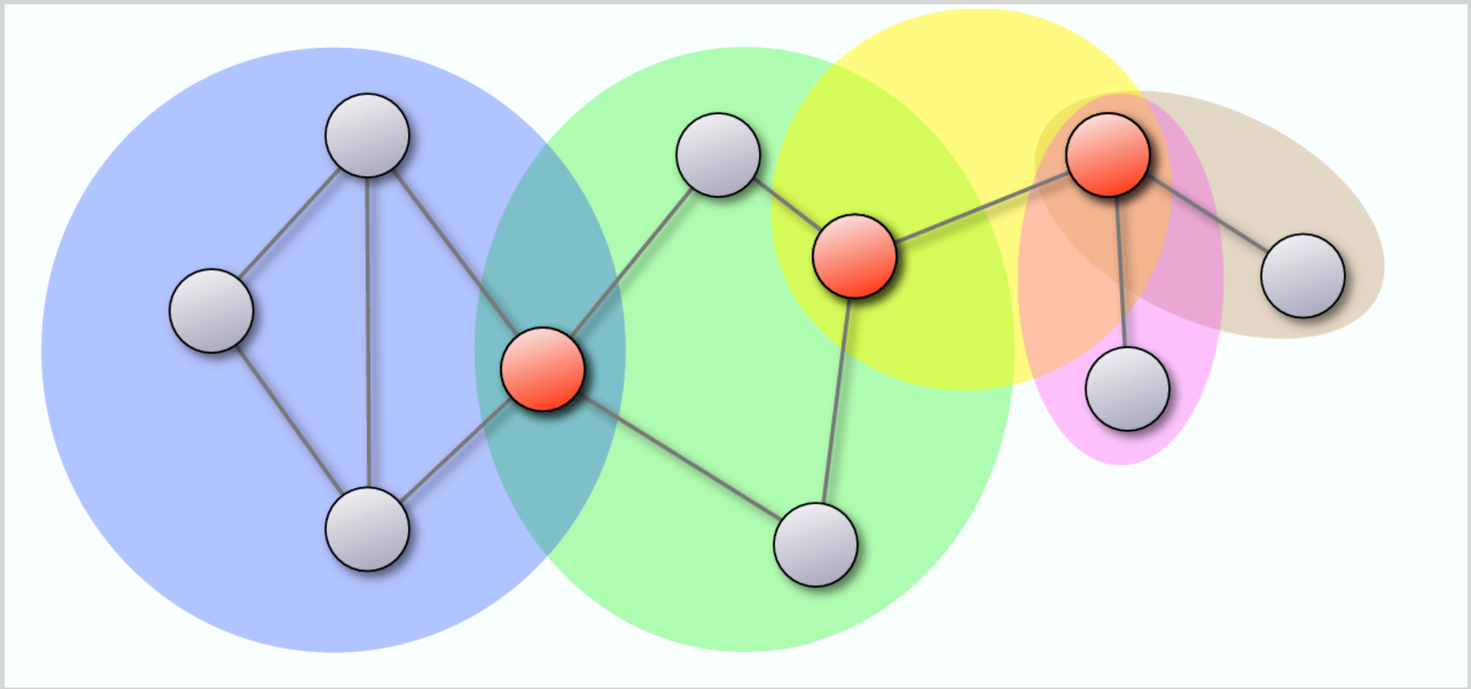
Definition

- Let $G = (V, E)$ be a graph.

  The equivalence relation $\sim$ on E is defined as :

$$e \sim f \quad := \begin{cases} e = f & \text{or} \\ e \text{ and } f \text{ are on a common cycle} \end{cases}$$

- The equivalence classes are named as Blocks



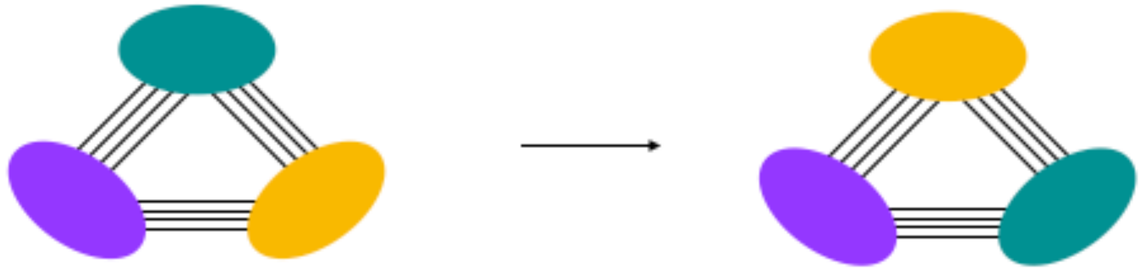
Lemma :

2 blocks always intersect at an articulation point.

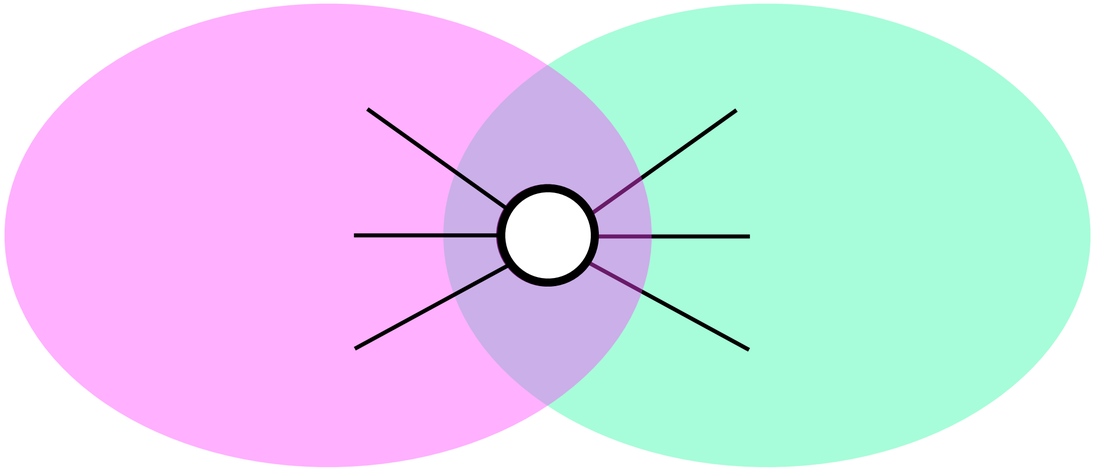
Articulation point is the critical point that holds blocks together. If a graph has an articulation point, it serves as the **only connection** between two or more blocks.

Coloring

Swapping Color Classes Trick

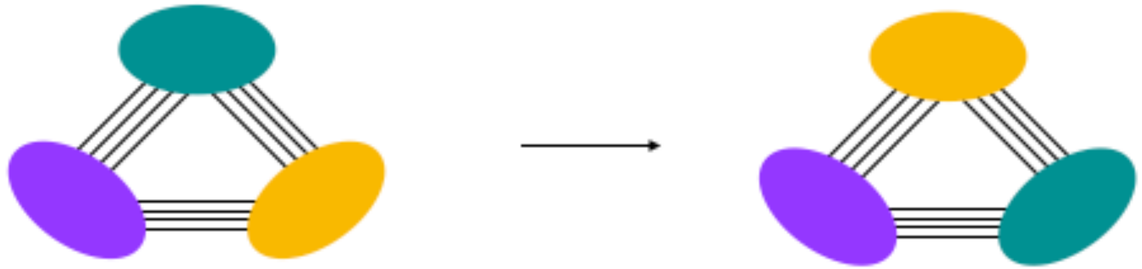


- For all Block-Graphs , if every block can be colored with k colors
  - G can be colored with k colors

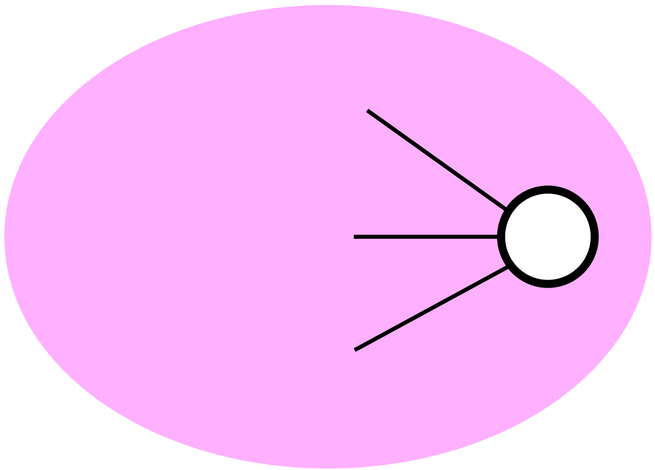


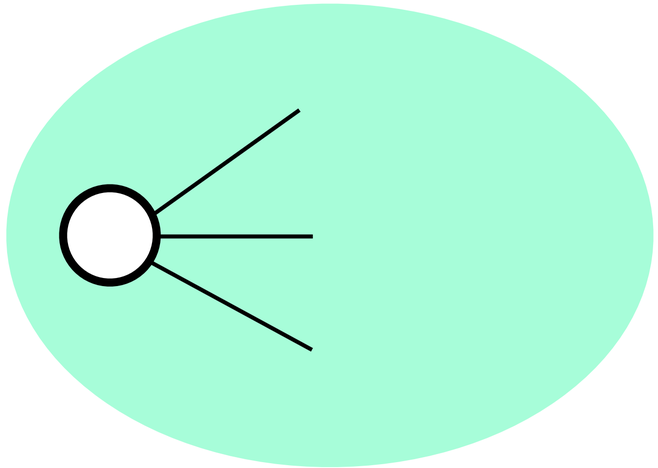
Coloring

Swapping Color Classes Trick



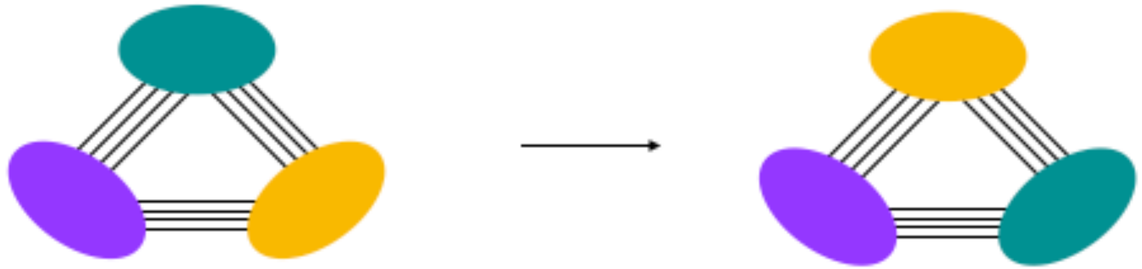
- For all Block-Graphs , if every block can be colored with k colors
  - G can be colored with k colors



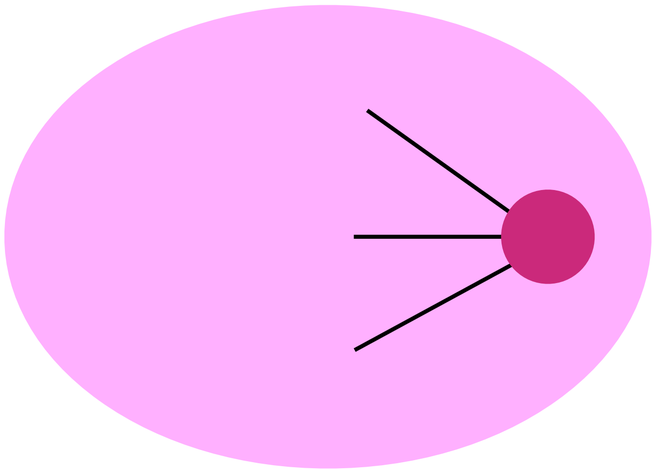


Coloring

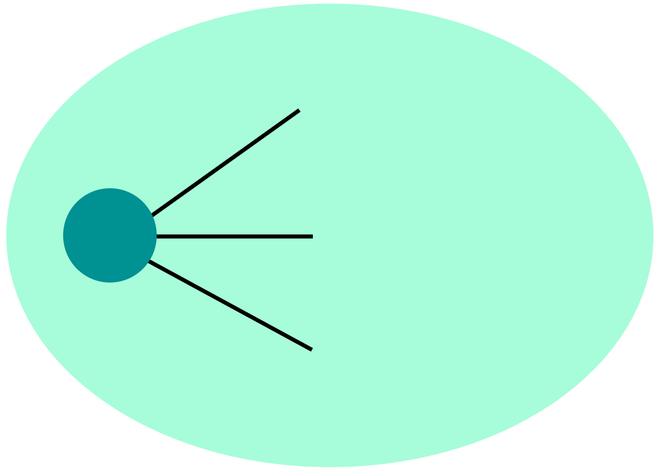
Swapping Color Classes Trick



- For all Block-Graphs , if every block can be colored with k colors
  - G can be colored with k colors

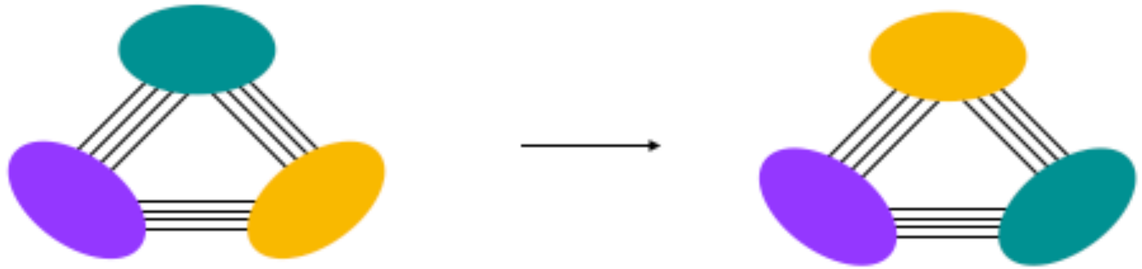


?

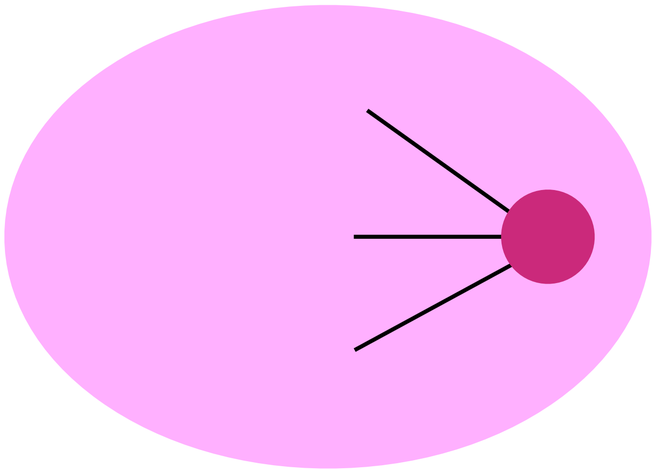


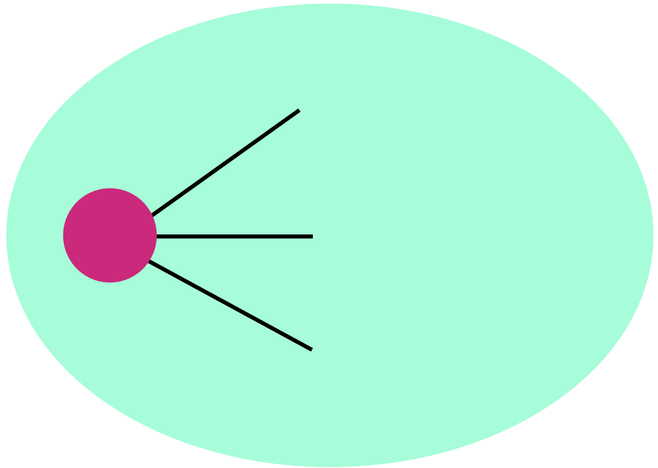
Coloring

Swapping Color Classes Trick



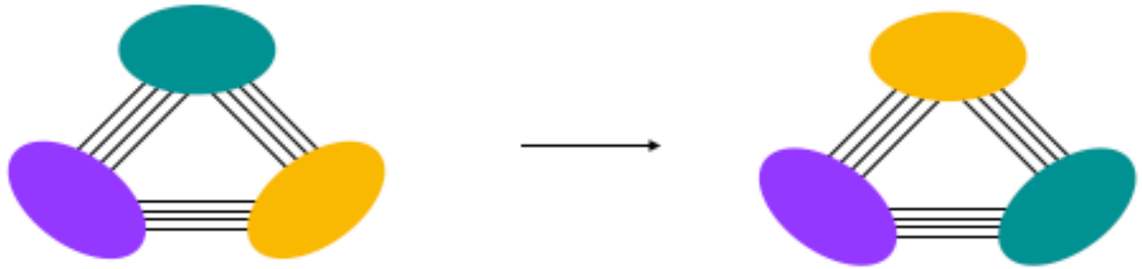
- For all Block-Graphs , if every block can be colored with k colors
  - G can be colored with k colors



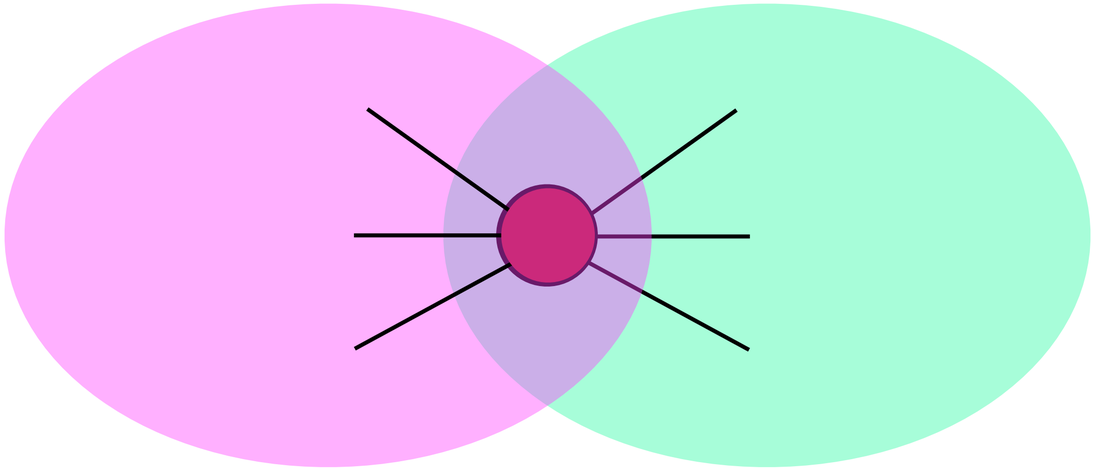


Coloring

Swapping Color Classes Trick



- For all Block-Graphs , if every block can be colored with k colors
  - G can be colored with k colors



Coloring

Brook's Theorem

- Pick an arbitrary order of the vertices : $V = \{v_1, \ldots, v_n\}$
- $c[v_1] \leftarrow 1$ color the first vertex with color 1
- **for** i = 2 **to** n **do**
  - $c[v_i] \leftarrow \min\{k \in \mathbb{N} \mid k \neq c[u] \text{ for all } u \in N(v_i) \cap \{v_1, \ldots, v_{i-1}\}\}$

    min color k s.t. it's not equal to the color of the neighbors of v_i that are already colored

- All Graphs
  - can be colored in O(|E|) time with $\Delta(G) + 1$ colors

Brook's Theorem

- $G \neq K_n$, $G \neq C_{2n+1}$, G connected
  - can be colored in O(|E|) time with $\Delta(G)$ colors

Minitest 2 - rest

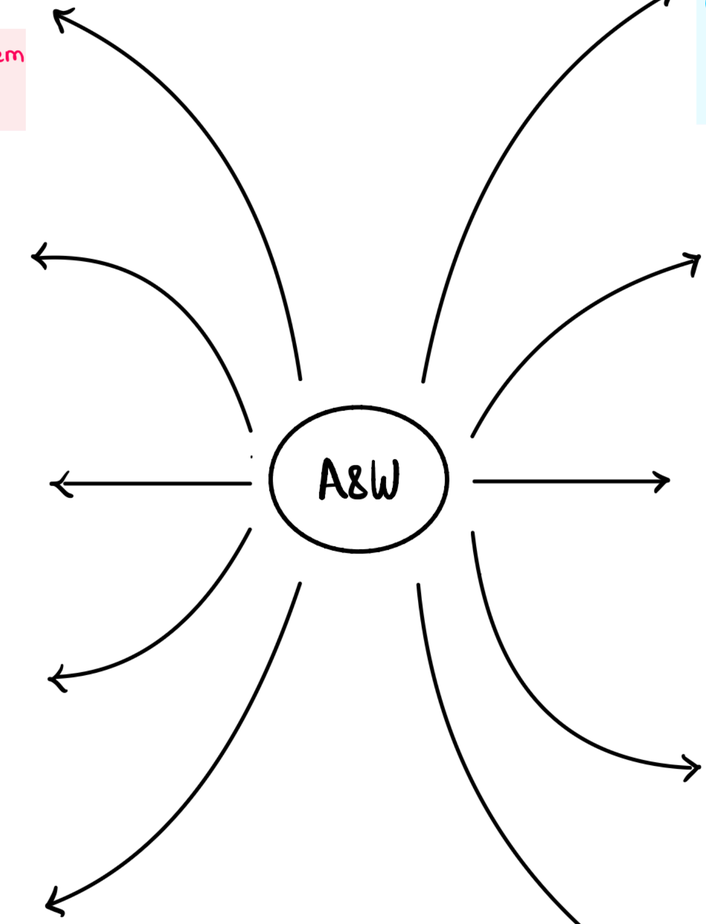
A&W

Exercise Session 6

Probability I

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- Minitest 3
- T2 Discussion
- Cycles + TSP Kahoot
- Formelsammlung
- Probability Theory

Minitest 3

T2

Feedback + Discussion

- Watch out for the comments !

- Keep up the good work ! 👏👏👏

- Questions, issues ... Let me know !
  - After the weekend

Cycles + TSP Kahoot

Let’s take a break

Probability Theory

Formelsammlung

Probability Theory

Importance for CS

- A&W
- NumCS
- WuS
- IML
- InfoSec
- Information Retrieval
- Data Science
- Anything ML related
- Anything AI related

Probability Theory

Importance for CS

LEARN IT ONCE !

Probability Theory

Basic Terms of Discrete Probability Space

- Elementary Event ω_i (Elementarereignis) : A single outcome of the experiment
- Sample Space Ω (Ergebnismenge) : The set of all possible outcomes $\Omega = \{\omega_1, \omega_2, \dots\}$
- Elementary Probability $Pr[\omega_i]$ (Elementarwahrcheinlichkeit) : Probability of an elementary event

- Event E (Ereignis) : A subset of the sample space ($E \subseteq \Omega$)
- Complementary event $\overline{E}$ (Komplementärereignis) : All outcomes not in E $\overline{E} := \Omega \setminus E$

Probability Theory

Pr[ ] Properties

- $0 \leq Pr[\omega_i] \leq 1$
- $\sum_{\omega \in \Omega} \Pr[\omega] = 1$
- Probability of an event E : $Pr[E] := \sum_{\omega \in E} \Pr[\omega]$

Probability Theory

Pr[ ] Properties

- For all events $A, B, A_1, A_2, \ldots$

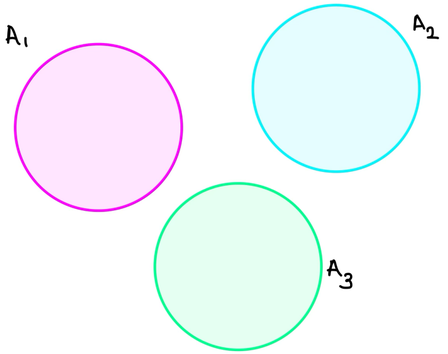
  - $Pr[\varnothing] = 0\,, Pr[\Omega] = 1$
  - $0 \leq Pr[A] \leq 1$
  - $Pr[\overline{A}] = 1 - Pr[A]$
  - $A \subseteq B \implies Pr[A] \leq Pr[B]$

- $0 \leq Pr[\omega_i] \leq 1$
- $\sum_{\omega \in \Omega} \Pr[\omega] = 1$
- Probability of an event E : $Pr[E] := \sum_{\omega \in E} \Pr[\omega]$

Examples I

Probability Theory

Addition Rule



Events $A_1, \ldots, A_n$ are pairwise disjoint

$$\Longrightarrow \quad \Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i]$$

Infinite set of events $A_1, A_2 \ldots$ are pairwise disjoint

$$\Longrightarrow \quad \Pr\left[\bigcup_{i=1}^{\infty} A_i\right] = \sum_{i=1}^{\infty} \Pr[A_i]$$

pairwise disjoint : For all pairs A_i, A_j with $i \neq j$: $A_i \cap A_j = \varnothing$

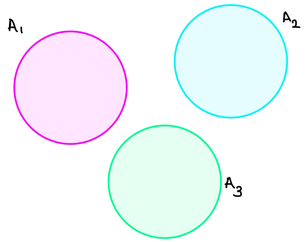
Probability Theory

Boolean Inequality

Addition Rule

Events $A_1, \ldots, A_n$ are pairwise disjoint

$$\Longrightarrow \quad \Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i]$$

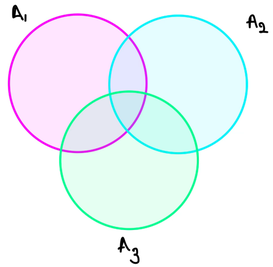


Boolean Inequality

For arbitrary events

$$A_1, \ldots, A_n$$

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr[A_i]$$



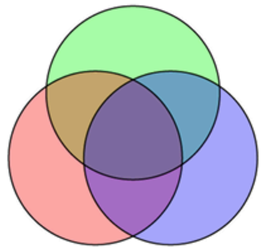
Probability Theory

Inclusion-Exclusion Principle (Siebformel)

For events $A_1, \ldots, A_n$

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i] - \sum_{1 \le i_1 < i_2 \le n} \Pr[A_{i_1} \cap A_{i_2}] + \cdots$$
$$+ (-1)^{l+1} \sum_{1 \le i_1 < \cdots < i_l \le n} \Pr[A_{i_1} \cap \cdots \cap A_{i_l}] + \cdots$$
$$+ (-1)^{n+1} \Pr[A_1 \cap \cdots \cap A_n] .$$

n=3:



Pr[A ∪ B ∪ C]
= Pr[A] + Pr[B] + Pr[C]
- Pr[A∩B] - Pr[A∩C] - Pr[B∩C] + Pr[A∩B∩C]

Probability Theory

Inclusion-Exclusion Principle (Siebformel)

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i] - \sum_{1 \le i_1 < i_2 \le n} \Pr[A_{i_1} \cap A_{i_2}] + \cdots + (-1)^{l+1} \sum_{1 \le i_1 < \cdots < i_l \le n} \Pr[A_{i_1} \cap \cdots \cap A_{i_l}] + \cdots + (-1)^{n+1} \Pr[A_1 \cap \cdots \cap A_n]\,.$$

To find the cardinality of the union of n sets:

1. Include the cardinalities of the sets.
2. Exclude the cardinalities of the pairwise intersections.
3. Include the cardinalities of the triple-wise intersections.
4. Exclude the cardinalities of the quadruple-wise intersections.
5. Include the cardinalities of the quintuple-wise intersections.
6. Continue, until the cardinality of the n-tuple-wise intersection is included (if n is odd) or excluded (n even).

Probability Theory

Inclusion-Exclusion Principle (Siebformel)

$$\Pr\left[\bigcup_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i] - \sum_{1 \le i_1 < i_2 \le n} \Pr[A_{i_1} \cap A_{i_2}] + \cdots + (-1)^{l+1} \sum_{1 \le i_1 < \cdots < i_l \le n} \Pr[A_{i_1} \cap \cdots \cap A_{i_l}] + \cdots + (-1)^{n+1} \Pr[A_1 \cap \cdots \cap A_n]\,.$$

To find the probability of the union of n sets:

1. Include the probability of the sets.
2. Exclude the probability of the pairwise intersections.
3. Include the probability of the triple-wise intersections.
4. Exclude the probability of the quadruple-wise intersections.
5. Include the probability of the quintuple-wise intersections.
6. Continue, until the probability of the n-tuple-wise intersection is included (if n is odd) or excluded (n even).

Examples II

Probability Theory

Laplace-Space

- Laplace-Space :
  - A finite probability space where all outcomes/elementary events have the same probability

$$\Omega = \{\omega_1, \omega_2, \ldots, \omega_n\}$$

$$Pr[\omega_i] = \frac{1}{n} \qquad \text{for all } i = 1, \ldots, n$$

$$Pr[A] = \frac{|A|}{|\Omega|}$$

Probability Theory

Combinatorics

different ways of
drawing k elements from n options

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Probability Theory

Combinatorics

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Q : Create a 3-digit password using digits 0-9.

Repetition : allowed

Order : matters

Formula : n^k

A :

- n = 10
  - There are 10 options
  - digits 0-9
- k = 3
  - We draw 3 elements in order
  - password length is 3

$$10^3 = 10 \cdot 10 \cdot 10 = 1000$$

Probability Theory

Combinatorics

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Q : Choose 3 students to present one after another from a group of 5

Repetition : not allowed

Order : matters

Formula : $n^{\underline{k}}$

$$n^{\underline{k}} := \underbrace{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}_{k \text{ Faktoren}}$$

A :

- n = 5
  - There are 5 options
  - student group of 5
- k = 3
  - We draw 3 elements in order
  - 3 students will present

$$5^{\underline{3}} = 5 \cdot 4 \cdot 3 = 60$$

Probability Theory

Combinatorics

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Q : You have 6 friends. Choose 2 of them to go on a trip.

Repetition : not allowed

Order : doesn't matter

Formula : $\binom{n}{k}$

A :

- n = 6
  - There are 6 options
  - you have 6 friends
- k = 2
  - We draw 2 elements without order
  - 2 friends will come

$$\binom{6}{2} = \binom{6!}{2! \cdot (6-2)!} = \binom{6!}{2! \cdot 4!} = 15$$

$$\binom{6}{2} = \binom{6 \cdot 5}{1 \cdot 2} = 15$$

Probability Theory

Combinatorics

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Q : Buy 4 scoops of ice cream from 3 flavors (vanilla, chocolate, strawberry)

Repetition : allowed

Order : doesn't matter

A :

- n = 3
  - There are 3 options
  - 3 flavors
- k = 4
  - We draw 4 elements without order
  - 4 scoops of ice cream

Formula :

$$\binom{n+k-1}{k}$$

$$\binom{3+4-1}{4} = \binom{6}{4} = 15$$

Do you notice something ?

Probability Theory

Combinatorics

	order matters	order does not matter
repetition allowed	n^k	$\binom{n+k-1}{k}$
repetition not allowed	$n^{\underline{k}}$	$\binom{n}{k}$

Q : Buy 4 scoops of ice cream from 3 flavors (vanilla, chocolate, strawberry)

Repetition : allowed

Order : doesn't matter

Formula :

$$\binom{n+k-1}{k}$$

A :

- n = 3
  - There are 3 options
  - 3 flavors
- k = 4
  - We draw 4 elements without order
  - 4 scoops of ice cream

Do you notice something ?

$$\binom{3+4-1}{4} = \binom{6}{4} = 15 = \binom{6}{2}$$

Probability Theory

Conditional Probability

the probability that event A will occur if we already know that event B has occurred

- conditional probability :
  - Let A and B be arbitrary events with $Pr[B] > 0$. The conditional probability $Pr[A \mid B]$ of A given B is

$$Pr[A \mid B] := \frac{Pr[A \cap B]}{Pr[B]}$$

$$Pr[A \cap B] = Pr[A \mid B] \cdot Pr[B]$$

$$Pr[A \cap B] = Pr[B \mid A] \cdot Pr[A]$$

Probability Theory

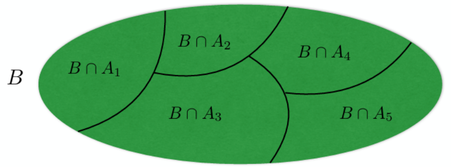
Conditional Probability - Theorems

Bedingte Wahrscheinlichkeiten

- **Definition:** Ist $\Pr[B] > 0$, so ist $\Pr[A|B] := \frac{\Pr[A \cap B]}{\Pr[B]}$.
- **Multiplikationssatz:** Ist $\Pr[A_1 \cap \ldots \cap A_n] > 0$, so ist

$$\Pr[A_1 \cap \cdots \cap A_n] = \Pr[A_1] \cdot \Pr[A_2|A_1] \cdot \Pr[A_3|A_1 \cap A_2] \cdot \ldots \cdot \Pr[A_n|A_1 \cap \cdots \cap A_{n-1}].$$

- **Satz von der totalen Wahrscheinlichkeit:**
  Ist $\Omega = A_1 \uplus \cdots \uplus A_n$ mit $\Pr[A_1], \ldots, \Pr[A_n] > 0$, so gilt $\Pr[B] = \sum_{i=1}^{n} \Pr[B|A_i] \cdot \Pr[A_i]$.



- **Satz von Bayes:**
  Ist $B \subseteq A_1 \uplus \cdots \uplus A_n$ mit $\Pr[A_1], \ldots, \Pr[A_n], \Pr[B] > 0$, so gilt

$$\Pr[A_i|B] = \frac{\Pr[A_i \cap B]}{\Pr[B]} = \frac{\Pr[B|A_i] \cdot \Pr[A_i]}{\sum_{j=1}^{n} \Pr[B|A_j] \cdot \Pr[A_j]}.$$

Examples III

Probability Theory

Conditional Independence

- conditional independence :
  - Event A and B are independent , if

$$Pr[A \cap B] = Pr[A] + Pr[B]$$

  - Events A ,B and C are independent , if

$$\Pr[A \cap B \cap C] = \Pr[A] \cdot \Pr[B] \cdot \Pr[C]$$

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$
$$\Pr[A \cap C] = \Pr[A] \cdot \Pr[C]$$
$$\Pr[B \cap C] = \Pr[B] \cdot \Pr[C]$$

Minitest 3 - Discussion

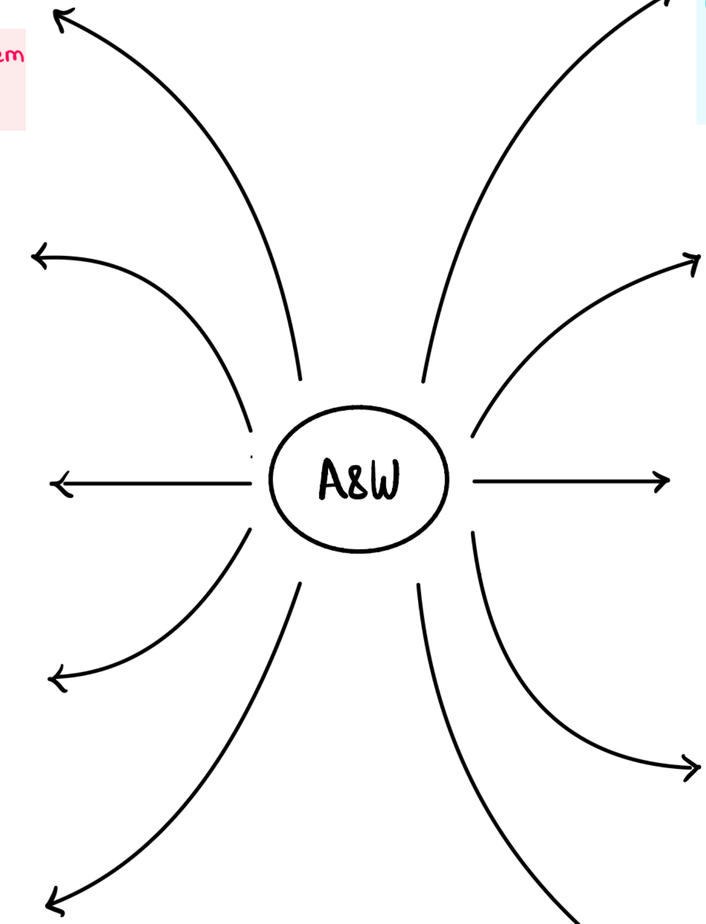
A&W

Exercise Session 7

Probability II

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- Coloring Kahoot

- Probability Theory II
  - Conditional independence
  - Random variables
  - Expected value , Variance
  - Distributions

Coloring Kahoot

Let’s take a break

Probability Theory

Probability Theory

Conditional Independence

- conditional independence :
  - Event A and B are independent , if

$$Pr[A \cap B] = Pr[A] \cdot Pr[B]$$

  - Events A ,B and C are independent , if

$$\Pr[A \cap B \cap C] = \Pr[A] \cdot \Pr[B] \cdot \Pr[C]$$

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$
$$\Pr[A \cap C] = \Pr[A] \cdot \Pr[C]$$
$$\Pr[B \cap C] = \Pr[B] \cdot \Pr[C]$$

Probability Theory

Conditional Independence - Formelsammlung

Unabhängigkeit

- **Definition:** $X_1, \ldots, X_n$ heissen genau dann unabhängig, wenn für alle $(x_1, \ldots, x_n) \in W_{X_1} \times \ldots \times W_{X_n}$ gilt: $\Pr[X_1 = x_1, \ldots, X_n = x_n] = \Pr[X_1 = x_1] \cdot \ldots \cdot \Pr[X_n = x_n]$.
- **Multiplikationsformel:** Sind $X_1, \ldots, X_n$ unabhängig und $S_i \subseteq W_{X_i}$, dann gilt: $\Pr[X_1 \in S_1, \ldots, X_n \in S_n] = \Pr[X_1 \in S_1] \cdot \ldots \cdot \Pr[X_n \in S_n]$.
- **Transformationen:** Seien $f_i : \mathbb{R} \to \mathbb{R}$. Wenn $X_1, \ldots, X_n$ unabhängig sind, dann gilt dies auch für $f(X_1), \ldots, f(X_n)$.
- **Summe:** Sind X, Y unabhängig und $Z := X + Y$, so gilt $f_Z(z) = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x)$.

Probability Theory

Random Variable

- random variable X :
  - A random variable X is a measurable function $X : \Omega \to \mathbb{R}$ from a sample space Ω as a set of possible outcomes to real numbers

Random process

Outcomes → Numbers

Flipping a coin

$$X = \begin{cases} 1 & \text{if heads} \\ 0 & \text{if tails} \end{cases}$$

Rolling 8 dices

$$Y = \text{Sum of the upward faces after rolling 8 dices}$$

Probability Theory

Random Variable

- random variable X : Random process

Outcomes → Numbers

  - A random variable X is a measurable function $X : \Omega \to \mathbb{R}$ from a sample space Ω as a set of possible outcomes to real numbers

Flipping a coin

$$X = \begin{cases} 1 & \text{if heads} \\ 0 & \text{if tails} \end{cases}$$

$$Pr[X = 1] = Pr[w \in \Omega \,|\, X(w) = 1]$$

Rolling 8 dices

$$Y = \text{Sum of the upward faces after rolling 8 dices}$$

$$Pr[Y \leq 6] = Pr[w \in \Omega \,|\, X(w) \leq 6]$$

Probability Theory

Functions

- probability density function $f_x(x)$
  - $f_X : \mathbb{R} \to [0,1], \quad x \mapsto \Pr[X = x] \quad (= \Pr[X(\omega) = x])$

- cumulative distribution function $F_x(x)$
  - $F_X : \mathbb{R} \to [0,1], \quad x \mapsto \Pr[X \leq x]$

Probability Theory

Expected Value

- expected value $\mathbb{E}[X]$
  - $$\mathbb{E}[X] := \sum_{x \in W_x} x \cdot Pr[X = x]$$
  - $$\mathbb{E}[X] = \sum_{w \in \Omega} X(w) \cdot Pr[w]$$
  - $$\mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

weighted average

Probability Theory

Expected Value Properties

- expected value
  - $\mathbb{E}[X] = \sum_{w \in \Omega} X(w) \cdot Pr[w]$

weighted average

- linearity:
  - $E[X + Y] = E[X] + E[Y]$
  - $E[aX] = aE[x]$
  - $E[a_1X_1 + a_2X_2 + \ldots + a_nX_N + b] = a_1E[X_1] + \ldots + a_nE[X_n] + b$
- monotonicity :
  - If $X \leq Y$ then $E[X] \leq E[Y]$

Probability Theory

Variance

- variance $Var[X]$

$$\mathrm{Var}[X] := \mathbb{E}[(X - E[x])^2] = \sum_{x \in W_X} (x - E[x])^2 \cdot \Pr[X = x]$$

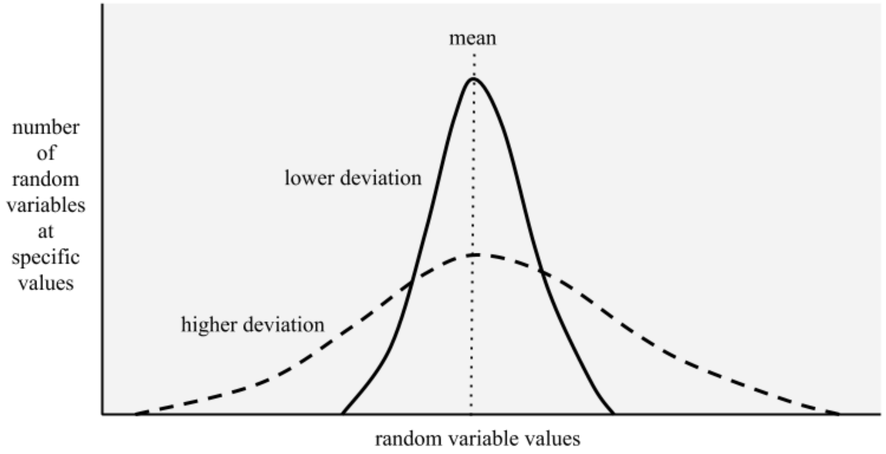
$$\mathrm{Var}[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

$$\mathrm{Var}[a \cdot X + b] = a^2 \cdot \mathrm{Var}[X]$$

- standard deviation σ

$$\sigma := \sqrt{\mathrm{Var}[X]}$$

measure of how far a set of numbers is spread out from their average value/mean



Probability Theory

Formelsammlung

Erwartungswert

- **Definition:** $\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x]$
- **Linearität:** Für $a_1, \ldots, a_n, b \in \mathbb{R}$ gilt $\mathbb{E}[a_1 X_1 + \ldots + a_n X_n + b] = a_1\,\mathbb{E}[X_1] + \ldots + a_n\,\mathbb{E}[X_n] + b$.
- **Summenformel:** Ist $W_X \subseteq \mathbb{N}_0$, dann gilt $\mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$.
- **Multiplikativität:** Für unabhängige $X_1, \ldots, X_n$ gilt $\mathbb{E}[X_1 \cdot \ldots \cdot X_n] = \mathbb{E}[X_1] \cdot \ldots \cdot \mathbb{E}[X_n]$.

Varianz

- **Definition:** $\mathrm{Var}[X] := \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$.
- **Translation:** Für $a, b \in \mathbb{R}$ gilt $\mathrm{Var}[a \cdot X + b] = a^2 \cdot \mathrm{Var}[X]$.
- **Standardabweichung:** $\sigma[X] := \sqrt{\mathrm{Var}[X]}$.
- **Additivität:** Für unabhängige $X_1, \ldots, X_n$ gilt $\mathrm{Var}[X_1 + \ldots + X_n] = \mathrm{Var}[X_1] + \ldots + \mathrm{Var}[X_n]$.

Probability Theory

Distributions

Probability Theory

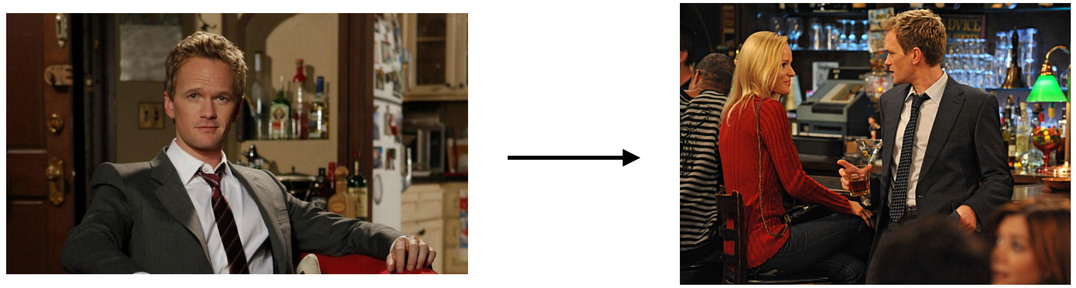
Indicator Variable

- indicator variable I_A:
  - For an event $A \subseteq \Omega$ $\quad I_A(\omega) := \begin{cases} 1, & \omega \in A \\ 0, & \omega \notin A \end{cases}$
  - $p = Pr[A] = E[I_A]$

$$f_{I_A}(x) = \begin{cases} p, & x = 1, \\ 1-p, & x = 0, \\ 0, & \text{otherwise} \end{cases}$$

Probability Theory

Bernoulli Distribution



$$X \sim \text{Bernoulli}(p)$$

yes-no question

$$f_X(x) = \begin{cases} p, & x = 1, \\ 1-p, & x = 0, \\ 0, & \text{otherwise} \end{cases}$$

Example to remember :

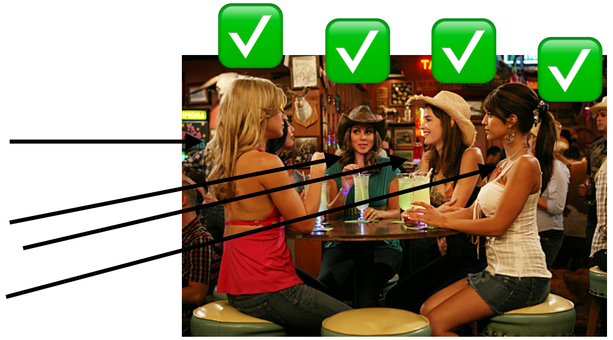
Coin toss

X = indicator for head

$$E[X] = p \qquad Var[X] = p\,(1-p)$$

Probability Theory

Binomial Distribution



$$X \sim \text{Bin}(n, p)$$

sucesses in a sequence of n yes-no questions

$$f_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x}, & x \in \{0,1,\ldots,n\} \\ 0, & \text{otherwise} \end{cases}$$

Example to remember :

Coin toss 10 times

X = #heads

$$E[X] = np \qquad Var[X] = np\,(1-p)$$

Probability Theory

Poisson-Distribution

$$X \sim \mathrm{Po}(\lambda)$$

$$f_X(i) = \begin{cases} \frac{e^{-\lambda}\lambda^i}{i!}, & \text{für } i \in \mathbb{N}_0 \\ 0, & \text{otherwise} \end{cases}$$

$\mathrm{Bin}(n, \lambda/n)$ converges to $\mathrm{Po}(\lambda)$

for $n \to \infty$

$$E[X] = \lambda \qquad Var[X] = \lambda$$

Probability Theory

Geometric Distribution

$$X \sim \text{Geo}(p)$$

$$f_X(i) = \begin{cases} p(1-p)^{i-1}, & \text{für } i \in \mathbb{N}, \\ 0, & \text{otherwise} \end{cases}$$

$$F_X(n) = 1 - (1-p)^n$$

$$E[X] = 1/p \qquad Var[X] = \frac{1-p}{p^2}$$

#yes-no questions
needed to get one yes

Example to remember :

Coin toss until a head comes

X = #tosses

Probability Theory

Geometric Distribution

$$X \sim \text{Geo}(p)$$

$$f_X(i) = \begin{cases} p(1-p)^{i-1}, & \text{für } i \in \mathbb{N}, \\ 0, & \text{otherwise} \end{cases}$$

$$F_X(n) = 1 - (1-p)^n$$

$$E[X] = 1/p \qquad Var[X] = \frac{1-p}{p^2}$$

Robin has no brain

Memorylessness

$$\Pr[X \geq s + t \mid X > s] = \Pr[X \geq t]$$

$$\Pr[X = s + t \mid X > s] = \Pr[X = t]$$

Probability Theory

Negative Binomial Distribution

$$X \sim \text{NegativeBinomial}(n)$$

#yes-no questions
needed to get n yesses

$$f_X(k) = \begin{cases} \binom{k-1}{n-1}(1-p)^{k-n}p^n, & \text{for } k = 1,2,\ldots \\ 0, & \text{otherwise} \end{cases}$$

Example to remember :

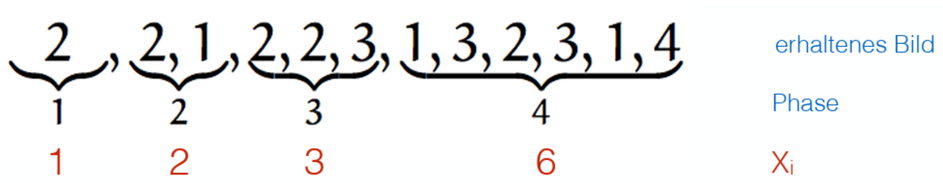
Coin toss until n-th head comes

X = #tosses

$$E[X] = n/p$$

Probability Theory

Coupon Collector



phase i := turns while we have i-1 different coupons

$X_i :=$ #turns in phase i

$$X_i \sim \text{Geo}\left(\frac{n-(i-1)}{n}\right) \quad E[X_i] = 1/p$$

$$X = \sum_{i=1}^{n} X_i$$

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \cdot \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n,$$

$$H_n = \ln n + \mathcal{O}(1)$$

collect all coupons and win

Example to remember :

n different coupons , we're getting one in each turn

X = #turns until we get all n coupons

Probability Theory

Formelsammlung

Wichtige Verteilungen

Name	Bezeichnung	Wertebereich	Dichte	Erwartungswert	Varianz
Bernoulli	Bernoulli(p)	$\{0,1\}$	$f_X(i) = \begin{cases} p & \text{für } i=1, \\ 1-p & \text{für } i=0. \end{cases}$	p	$p(1-p)$
Binomial	$\mathrm{Bin}(n,p)$	$\{0,1,\dots,n\}$	$f_X(i) = \binom{n}{i} p^i (1-p)^{n-i}$	np	$np(1-p)$
Geometrisch	$\mathrm{Geo}(p)$	$\mathbb{N}$	$f_X(i) = p(1-p)^{i-1}$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Poisson	$\mathrm{Po}(\lambda)$	$\mathbb{N}_0$	$f_X(i) = \frac{e^{-\lambda}\lambda^i}{i!}$	λ	λ

Examples

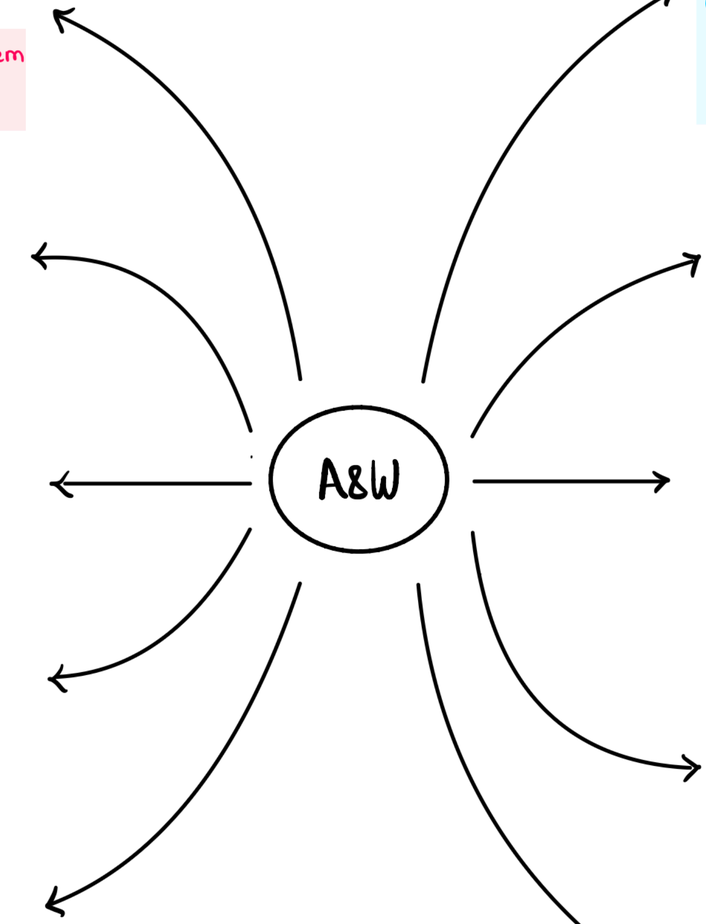
A&W

Exercise Session 8

Probability III

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Outline

- Minitest 4

- Probability Theory III
  - independence of random variables
  - wald’s identity
  - inequalities

- A Game of Skill - Probability CodeExpert

Minitest 4

Probability Theory

Probability Theory

Independence of random variables

- independent (random variables)
  - $X_1, \ldots, X_n$ are independent if for all $x_1 \in X_{X_1}, , , x_n \in W_{X_n}$ the events $X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n$ are independent.

$$\underbrace{\Pr[X_1 = x_1, \ldots, X_n = x_n]}_{=f_{X_1,\ldots,X_n}(x_1,x_2,\ldots,x_n)} = \underbrace{\Pr[X_1 = x_1]}_{=f_{X_1}(x_1)} \cdots \underbrace{\Pr[X_n = x_n]}_{=f_{X_n}(x_n)}$$

Aber: Viele der Gleichungen sind *redundant*.
Beispiel für n = 3: Falls X_3 Wertebereich {0,1} hat, dann folgt aus
$\Pr[X_1 = x_1, X_2 = x_2, X_3 = 0] = \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdot \Pr[X_3 = 0]$, und
$\Pr[X_1 = x_1, X_2 = x_2, X_3 = 1] = \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdot \Pr[X_3 = 1]$
automatisch:

$$\begin{aligned}\Pr[X_1 = x_1, X_2 = x_2] &= \Pr[X_1 = x_1, X_2 = x_2, X_3 = 0] + \Pr[X_1 = x_1, X_2 = x_2, X_3 = 1] \\ &= \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdot \Pr[X_3 = 0] + \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdot \Pr[X_3 = 1] \\ &= \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2] \cdot (\Pr[X_3 = 0] + \Pr[X_3 = 1]) \\ &= \Pr[X_1 = x_1] \cdot \Pr[X_2 = x_2].\end{aligned}$$

- independent (random variables)
  - $X_1, \ldots, X_n$ are independent if for all $x_1 \in X_{X_1}, , , x_n \in W_{X_n}$ the events $X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n$ are independent.

$$\underbrace{\Pr[X_1 = x_1, \ldots, X_n = x_n]}_{=f_{X_1,\ldots,X_n}(x_1,x_2,\ldots,x_n)} = \underbrace{\Pr[X_1 = x_1]}_{=f_{X_1}(x_1)} \cdot \ldots \cdot \underbrace{\Pr[X_n = x_n]}_{=f_{X_n}(x_n)},$$

Probability Theory

Independence of random variables

- $X_1, \ldots, X_n$ are independent random variables, $S_1, \ldots, S_n \subseteq \mathbb{R}$ arbitrary :
  - $\Pr[X_1 \in S_1, \ldots, X_n \in S_n] = \Pr[X_1 \in S_1] \cdots \Pr[X_n \in S_n]$.

- $X_1, \ldots, X_n$ are independent random variables , $f_1, \ldots, f_n$ are real-valued functions ($f_i : \mathbb{R} \to \mathbb{R}$ for $i = 1, \ldots, n$) :
  - $f_1(X_1), \ldots, f_n(X_n)$ are independent random variables

- independent (random variables)
  - $X_1, \ldots, X_n$ are independent if for all $x_1 \in X_{X_1}, , , x_n \in W_{X_n}$ the events $X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n$ are independent.

$$\underbrace{\Pr[X_1 = x_1, \ldots, X_n = x_n]}_{=f_{X_1,\ldots,X_n}(x_1,x_2,\ldots,x_n)} = \underbrace{\Pr[X_1 = x_1]}_{=f_{X_1}(x_1)} \cdot \ldots \cdot \underbrace{\Pr[X_n = x_n]}_{=f_{X_n}(x_n)},$$

Probability Theory

Independence of random variables

- X and Y are two independent random variables, $Z := X + Y$

$$f_Z(z) = \sum_{x \in W_X} f_X(x) \cdot f_Y(z - x) \, .$$

$$\text{Poisson}(\lambda_1) + \text{Poisson}(\lambda_2) = \text{Poisson}(\lambda_1 + \lambda_2)$$

$$\text{Bin}(n, p) + \text{Bin}(m, p) = \text{Bin}(n + m, p)$$

Probability Theory

Wald's Identity

- N and X are two independent random variables, $W_N \subseteq \mathbb{N}$

$$Z := \sum_{i=1}^{N} X_i \quad \text{where } X_1, X_2 \dots \text{ are independent copies of } X$$

$$\mathbb{E}[Z] = \mathbb{E}[N] \cdot \mathbb{E}[X]$$

Example

Probability Theory

Inequalities

get used to using it !

Abschätzungen

- **Boolesche Ungleichung, Union Bound:** $\Pr\left[\bigcup_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr[A_i]$.
- **Markov:** Ist $W_X \subseteq \mathbb{R}_{\geq 0}$ und $t \in \mathbb{R}_{\geq 0}$, so ist $\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$ bzw. $\Pr[X \geq t \cdot \mathbb{E}[X]] \leq \frac{1}{t}$.
- **Chebyshev:** Für $t \in \mathbb{R}_{\geq 0}$ ist $\Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{\mathrm{Var}[X]}{t^2}$ bzw. $\Pr[|X - \mathbb{E}[X]| \geq t \cdot \sigma[X]] \leq \frac{1}{t^2}$.
- **Chernoff:** Seien $X_1, \ldots, X_n$ unabhängig und Bernoulli-verteilt, $X := \sum_{i=1}^{n} X_i$ und $\delta \in [0, 1]$. Dann ist

$$\Pr[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{3}\delta^2 \mathbb{E}[X]},$$
$$\Pr[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\delta^2 \mathbb{E}[X]},$$
$$\Pr[X \geq t] \leq 2^{-t} \qquad \text{für } t \geq 2e\mathbb{E}[X].$$

Let’s take a break

A Game of Skill

CodeExpert

Questions

Feedbacks , Recommendations

Nil Ozer

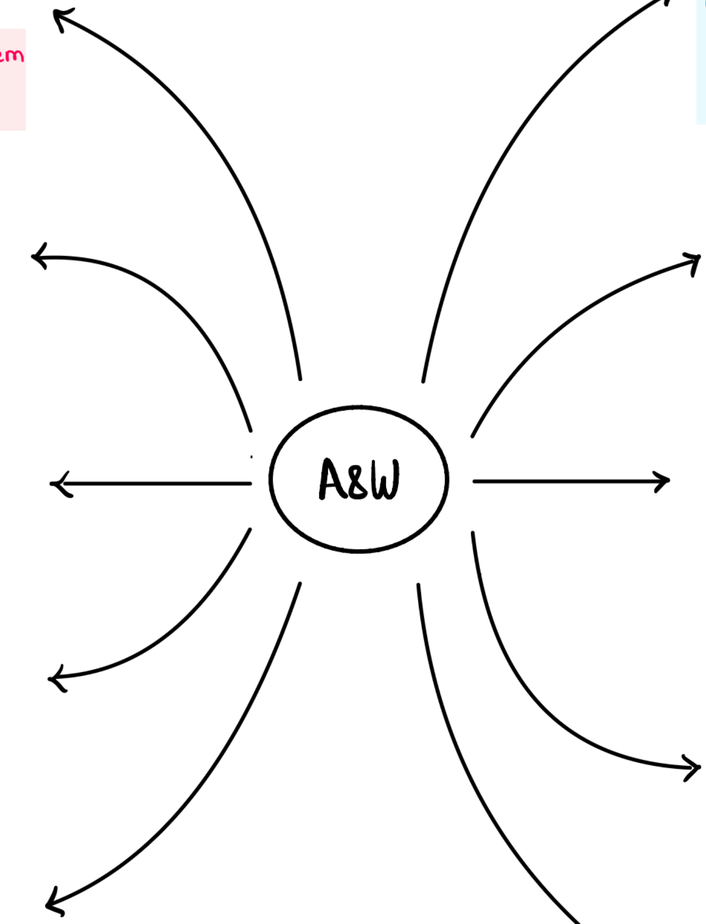
A&W

Exercise Session 10

Randomized Algorithms II

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Last Weeks ...

- 08.05 : Randomized Algorithms II
- 15.05 : Flow
- 22.05 online : Minimum Cut , Convex Hull I (shortly remaining primality tests)
- 27.05/28.05 extra session : Convex Hull II , Smallest Enclosing Cycle
- 29.05 last session : Exam Prep Session + Pizza and Drinks

Outline

- Randomized Algorithms II
  - Randomized Algorithms I recap
  - Primality Tests
  - Colorful Paths

Randomized Algorithms

Recap

Randomized Algorithms

Classic vs. Randomized

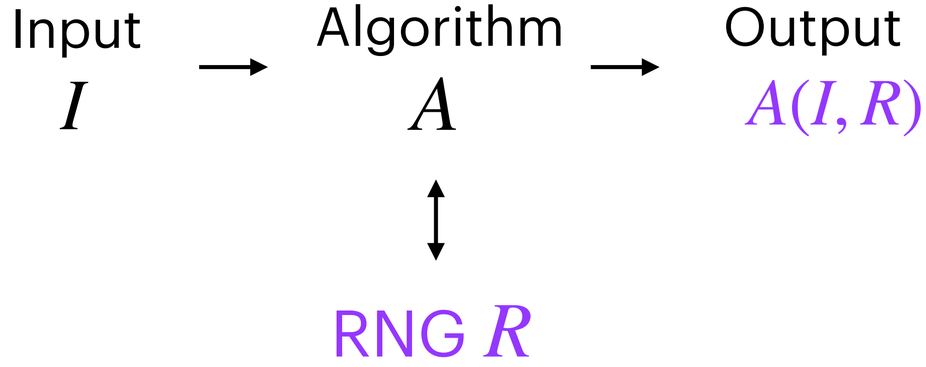
$A(I, R)$ can't be reproduced

classic

Input I → Algorithm A → Output $A(I)$

- $A(I)$ is correct and definite for all I
- The runtime is O(f(n)) for all I with $|I| = n$

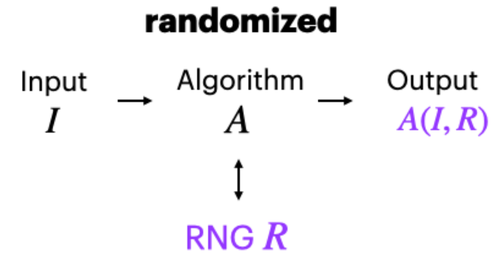
randomized



- $A(I, R)$ is correct with $Pr_R[A(I, R) \ is \ correct] \geq \ldots$ for all I
- The runtime is O(f(n)) and/or $Pr_R[\text{Runtime} \leq O(f(n))] \geq \ldots$ for all I with $|I| = n$

Randomized Algorithms

Las-Vegas vs. Monte-Carlo



Las-Vegas

Runtime is the RV

- can output true answer
- cannot output false answer
- can run forever/ can output no answer (???)

Monte-Carlo

Correctness/Quality is the RV

- can output true answer
- can output false answer
- always outputs an answer

Input: An array of $n \geq 2$ elements, in which half are '*a*'s and the other half are '*b*'s.

Output: Find an '*a*' in the array.

```
findingA_LV(array A, n)
begin
    repeat
        Randomly select one element out of n elements.
    until 'a' is found
end
```

```
findingA_MC(array A, n, k)
begin
    i := 0
    repeat
        Randomly select one element out of n elements.
        i := i + 1
    until i = k or 'a' is found
end
```

Target-Shooting

Problem Description

given : finite sets S and U with $S \subseteq U$

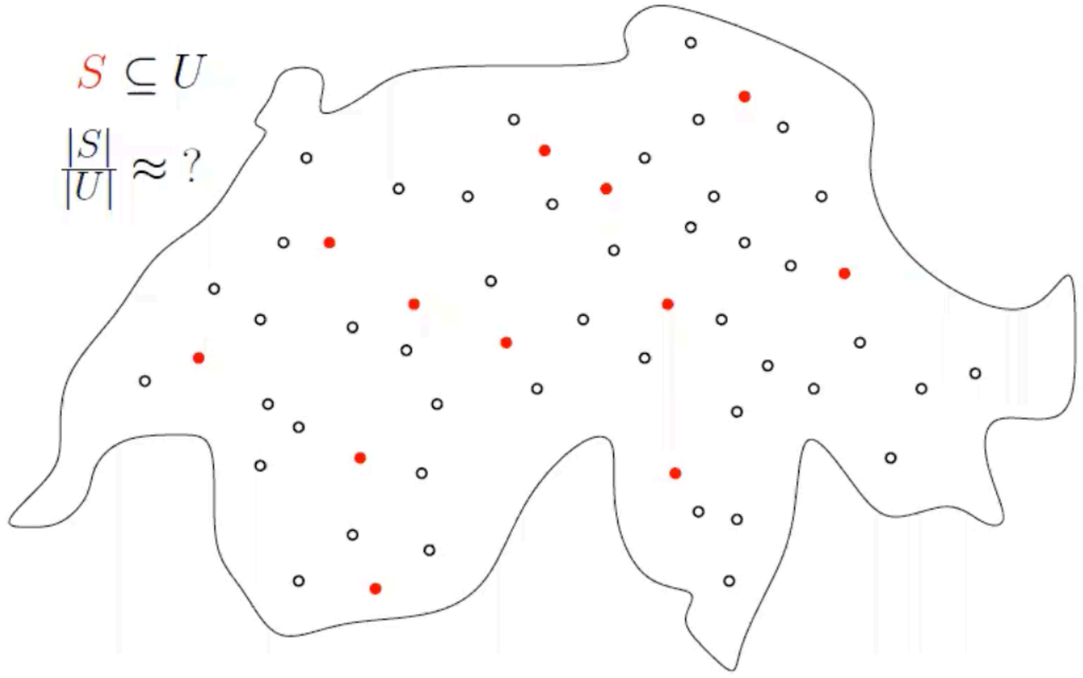
to find : $\approx \frac{|S|}{|U|}$

U is very large. We cannot afford to iterate through U

We can generate elements u in U uniformly distributed

$I_S : U \to \{0,1\}$

$I_S(u) = 1 \iff u \in S$



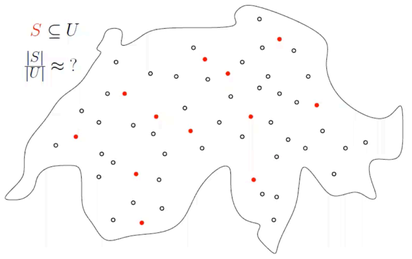
Target-Shooting

Algorithm

given : finite sets S and U with $S \subseteq U$

to find : $\approx \frac{|S|}{|U|}$

$$I_S(u) = 1 \iff u \in S$$



1 : Pick $u_1, \ldots, u_N$ from U randomly, uniformly and independently

2 : Return $\frac{1}{N} \cdot \sum_{i=1}^{N} I_S(u_i)$

Target-Shooting

Algorithm

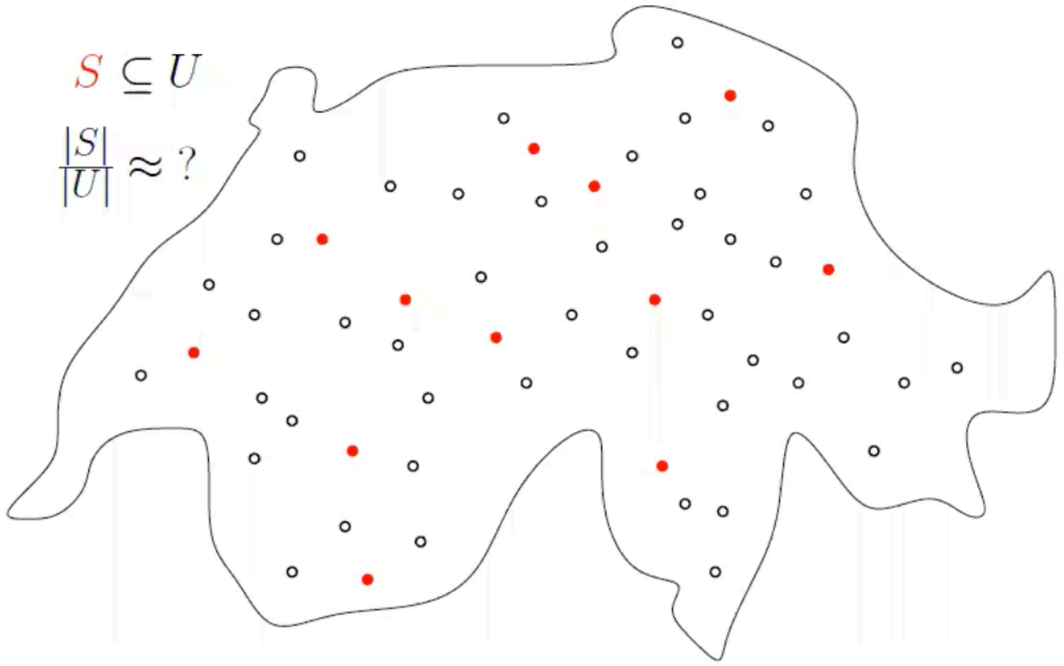
given : finite sets S and U with $S \subseteq U$

to find : $\approx \frac{|S|}{|U|}$

$$I_S(u) = 1 \iff u \in S$$

1 : Pick $u_1, \ldots, u_N$ from U randomly, uniformly and independently

2 : Return $\frac{1}{N} \cdot \sum_{i=1}^{N} I_S(u_i)$



Target-Shooting

Algorithm

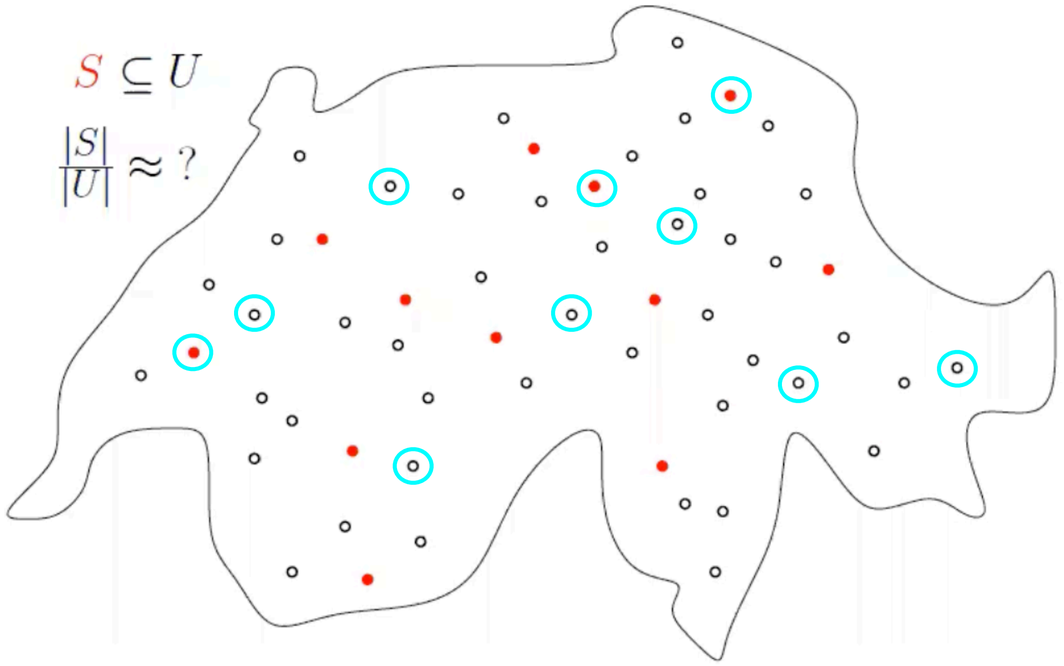
given : finite sets S and U with $S \subseteq U$

to find : $\approx \frac{|S|}{|U|}$

$$I_S(u) = 1 \iff u \in S$$

1 : Pick $u_1, \ldots, u_N$ from U randomly, uniformly and independently

2 : Return $\frac{1}{N} \cdot \sum_{i=1}^{N} I_S(u_i)$



Target-Shooting

Algorithm

given : finite sets S and U with $S \subseteq U$

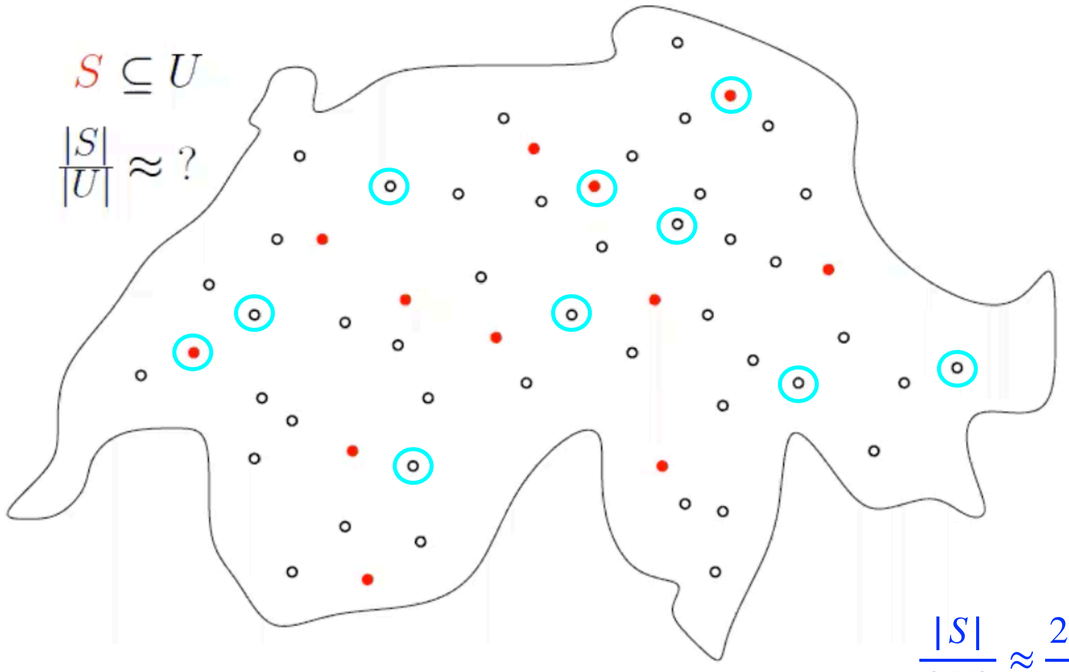
to find : $\approx \frac{|S|}{|U|}$

$$I_S(u) = 1 \iff u \in S$$

1 : Pick $u_1, \ldots, u_N$ from U randomly, uniformly and independently

2 : Return $\frac{1}{N} \cdot \sum_{i=1}^{N} I_S(u_i)$

$$\frac{1}{10} \cdot \sum_{i=1}^{10} I_S(u_i) = \frac{3}{10}$$



$$\frac{|S|}{|U|} \approx \frac{20}{64} = 0.3125$$

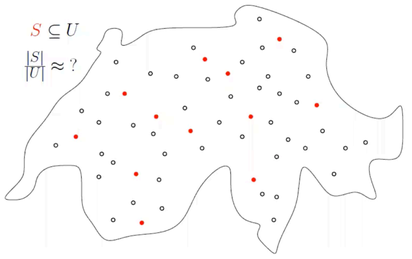
Target-Shooting

Algorithm

given : finite sets S and U with $S \subseteq U$

to find : $\approx \frac{|S|}{|U|}$

$$I_S(u) = 1 \iff u \in S$$



1 : Pick $u_1, \ldots, u_N$ from U randomly, uniformly and independently

2 : Return $\frac{1}{N} \cdot \sum_{i=1}^{N} I_S(u_i)$

Fehlerreduktionen:

- **Wiederholung MC:** Eine N-fache Wiederholung mit $N = 4\varepsilon^{-2} \ln \delta^{-1}$ steigert die Erfolgswahrscheinlichkeit eines Monte-Carlo-Algorithmus von $\frac{1}{2} + \varepsilon$ auf $\geq 1 - \delta$.
- **Wiederholung MC mit einseitigem Fehler:** Eine N-fache Wiederholung mit $N = \varepsilon^{-1} \ln \delta^{-1}$ steigert fÃ¼r einen Monte-Carlo-Algorithmus mit einseitigem Fehler die Erfolgswahrscheinlichkeit von ε auf $\geq 1 - \delta$.
- **Target Shooting:** Bestimmt der Target-Shooting-Algorithmus eine Menge $S \subseteq U$ mit $N \geq 3\frac{|U|}{|S|}\varepsilon^{-2} \ln(2/\delta)$ Versuchen, so ist die Ausgabe mit Wahrscheinlichkeit $\geq 1 - \delta$ im Intervall $\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$.

Finding Duplicates

Problem Description

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

$$\begin{array}{lccccccc} \mathcal{D} = (& A, & C, & B, & Z, & C, & B, & C \quad) \\ & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

$$\text{Dupl}(\mathcal{D}) = \{(2,5), (2,7), (5,7), (3,6)\}$$

Finding Duplicates

Problem Description

Elements in D are very large.

Storing and comparing is expensive

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \rightarrow [m]$ $[m] = \{1,2,...,m\}$

h is efficiently computable

h behaves like a random variable

$\forall u \in U \ \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$ (independent for different u)

Finding Duplicates

Problem Description

Elements in D are very large.

Storing and comparing is expensive

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$$h : U \to [m] \quad [m] = \{1,2,...,m\} \quad \forall u \in U \ \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$$s_i = s_j \implies h(s_i) = h(s_j)$$

Our m is much smaller than $|U|$ (compression)

Finding Duplicates

Algorithm

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$$h : U \to [m] \quad [m] = \{1,2,\ldots,m\} \quad \forall u \in U \; \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$$s_i = s_i \implies h(s_i) = h(s_j)$$

Our m is much smaller than $|U|$ (compression)

	A	C	B	Z	C	B	C
hashing:	$(\underbrace{31}_{h(A)}, 1)$	$(\underbrace{27}_{h(C)}, 2)$	$(\underbrace{12}_{h(B)}, 3)$	$(\underbrace{12}_{h(Z)}, 4)$	$(\underbrace{27}_{h(C)}, 5)$	$(\underbrace{12}_{h(B)}, 6)$	$(\underbrace{27}_{h(C)}, 7)$
sorting:	$(12, 3)$	$(12, 4)$	$(12, 6)$	$(27, 2)$	$(27, 5)$	$(27, 7)$	$(31, 1)$
duplicates:		$(3, 6)$,			$(2, 5), (2, 7), (5, 7)$		

Finding Duplicates

Challenge : Collisions

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \to [m]$ $[m] = \{1,2,\ldots,m\}$ $\forall u \in U \ \forall i \in [m] :$ $\Pr[h(u) = i] = \frac{1}{m}$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$$s_i = s_i \implies h(s_i) = h(s_j)$$

Our m is much smaller than $|U|$ (compression)

	A	C	B	Z	C	B	C
hashing:	$(\underbrace{31}_{h(A)}, 1)$	$(\underbrace{27}_{h(C)}, 2)$	$(\underbrace{12}_{h(B)}, 3)$	$(\underbrace{12}_{h(Z)}, 4)$	$(\underbrace{27}_{h(C)}, 5)$	$(\underbrace{12}_{h(B)}, 6)$	$(\underbrace{27}_{h(C)}, 7)$
sorting:	$(12, 3)$	$(12, 4)$	$(12, 6)$	$(27, 2)$	$(27, 5)$	$(27, 7)$	$(31, 1)$
duplicates:		$(3, 6)$,		$(2, 5), (2, 7), (5, 7)$			

collision : $h(B) = h(Z)$

Finding Duplicates

Challenge : Collisions

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \to [m] \quad [m] = \{1,2,\ldots,m\} \quad \forall u \in U \; \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$$s_i = s_i \implies h(s_i) = h(s_j)$$

Our m is much smaller than $|U|$ (compression)

Collision :

The new, undesired duplicates in the hashmap

the pairs (i, j) , $1 \leq i < j \leq n,$ with $s_i \neq s_j$ and $h(s_i) = h(s_j)$

Finding Duplicates

Challenge : Collisions

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i,j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \to [m] \quad [m] = \{1,2,\ldots,m\} \quad \forall u \in U \; \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$s_i = s_i \implies h(s_i) = h(s_j)$

Our m is much smaller than $|U|$ (compression)

Collision :

The new, undesired duplicates in the hashmap

the pairs (i,j) , $1 \leq i < j \leq n,$ with $s_i \neq s_j$ and $h(s_i) = h(s_j)$

$\mathbb{E}$[#Collisions] :

$K_{i,j}$ bernoulli RV. with : $K_{i,j} = 1 \iff (i,j)$ is a collision

$$\Pr[K_{i,j} = 1] = \begin{cases} \frac{1}{m} & \text{if } s_i \neq s_j \\ 0 & \text{otherwise} \end{cases} \qquad \mathbb{E}[K_{i,j}] \leq \frac{1}{m}$$

$$\mathbb{E}[\#\text{Collisions}] = \sum_{1 \leq i < j \leq n} \mathbb{E}[K_{i,j}] \leq \binom{n}{2} \cdot \frac{1}{m}$$

Finding Duplicates

Challenge : Collisions

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i,j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \to [m] \quad [m] = \{1,2,\ldots,m\} \quad \forall u \in U \; \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$s_i = s_i \implies h(s_i) = h(s_j)$

Our m is much smaller than $|U|$ (compression)

Collision :

The new, undesired duplicates in the hashmap

the pairs (i,j) , $1 \leq i < j \leq n,$ with $s_i \neq s_j$ and $h(s_i) = h(s_j)$

$\mathbb{E}$[#Collisions] :

$K_{i,j}$ bernoulli RV. with : $K_{i,j} = 1 \iff (i,j)$ is a collision

$$\Pr[K_{i,j} = 1] = \begin{cases} \frac{1}{m} & \text{if } s_i \neq s_j \\ 0 & \text{otherwise} \end{cases} \qquad \mathbb{E}[K_{i,j}] \leq \frac{1}{m}$$

$$\mathbb{E}[\#\text{Collisions}] \leq \binom{n}{2} \cdot \frac{1}{m} < 1 \quad \text{for } m = n^2$$

Finding Duplicates

Runtime

given : A dataset $D = (s_1, s_2, \ldots, s_n)$, is a sequence of n elements

to find : find all duplicates in D (i, j) with $1 \leq i < j \leq n$ is a duplicate in D if $s_i = s_j$

Hashfunction h :

$h : U \to [m] \quad [m] = \{1,2,\ldots,m\} \quad \forall u \in U \; \forall i \in [m] : \quad \Pr[h(u) = i] = \frac{1}{m}$

Each $h(s_i)$ is uniformly randomly distributed in $[m]$ BUT

$s_i = s_i \implies h(s_i) = h(s_j)$

Our m is much smaller than $|U|$ (compression)

Collision :

The new, undesired duplicates in the hashmap

the pairs (i, j) , $1 \leq i < j \leq n,$ with $s_i \neq s_j$ and $h(s_i) = h(s_j)$

$$\mathbb{E}[\#\text{Collisions}] \leq \binom{n}{2} \cdot \frac{1}{m} < 1 \quad \text{for } m = n^2$$

Runtime :

- n hash computations
- sorting in O(n log n)
- duplicate check comparisons (|Dupl(D)|+#Kollisionen) ≈ O(n)

additional memory

$$\underbrace{O(n \log n)}_{\text{indices}} + \underbrace{O(n \log m)}_{\text{hash values}} \overset{m=n^2}{=} O(n \log n)$$

Overall : O(n log n)

Randomized Algorithms

Primality Tests

Primality Test

Problem Description

given : A number $n \in \mathbb{N}$

to find : is n prime ? ?

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, ...

$$\pi(11) =$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, ...

$$\pi(11) = 5$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$

Primality Test

Naive Algorithm

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

1) For all $a \leq \sqrt{n}$ test if a divides n

Primality Test

Easy randomized test

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

1) Choose $a \in \{1,2, \ldots, \sqrt{n}\}$ uniformly at random

2) if a divides n then return ‘not prime’

3) else return ‘prime’

Refresher

DiskMat 🤢

gcd : greatest common divisor

$$n \text{ is prime} \Rightarrow \gcd(a,n) = 1 \quad \forall a \in [1,n-1]$$

$\mathbb{Z}_n^*$: the multiplicative group modulo n

$$\mathbb{Z}_n^* = \{a \in \{1,2,\ldots,n-1\} \mid \gcd(a,n) = 1\}$$

Primality Test

Euclidean Primality Test

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

n is prime $\Rightarrow \gcd(a,n) = 1 \quad \forall a \in [1, n-1]$

gcd := greatest common divisor

can be calculated in $O((\log nm)^3)$

1) Choose $a \in \{1, 2, \ldots, \sqrt{n}\}$ uniformly at random

2) if $gcd(a,n) > 1$ then return 'not prime'

3) else return 'prime'

- if n is a prime : always correct
- if n is not a prime : it might return a wrong answer with the probability

$$\frac{\left|\{a \in [1, n-1] : \gcd(a,n) = 1\}\right|}{n-1} = \frac{|\mathbb{Z}_n^*|}{n-1}$$

Let’s take a break

Randomized Algorithms

Colorful Paths

Helper

Mathematical Tools and Notations

$[n] := \{1, 2, \ldots, n\}$

$[n]^k :=$ the set of sequences over $[n]$ of length k $\qquad |[n]^k| = n^k$

$\binom{[n]}{k} :=$ the set of k-element subsets of $[n]$ $\qquad \left|\binom{[n]}{k}\right| = \binom{n}{k}.$

The k nodes on a path of length $k-1$ can be colored using $[k]$ in exactly k^k ways

$k!$ of these colorings use each color exactly once

Helper

Mathematical Tools and Notations

Handshaking lemma : For all graphs , it holds that $\sum_{v \in V} \deg(v) = 2|E|$.

If you repeat an experiment with success probability p until success, then the expected number of trials is $\frac{1}{p}$ $(Geo(p))$

Helper

Mathematical Tools and Notations

For $c, n \in \mathbb{R}^+$, it holds that $c^{\log n} = n^{\log c}$

$$2^{\log n} = n^{\log 2} = n \text{ and } 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)} \text{ is always polynomial in } n$$

For $n \in \mathbb{N}_0$, it holds that $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ (binomial theorem)

For $n \in \mathbb{N}_0$, it holds that $\frac{n!}{n^n} \geq e^{-n}$ (power series expansion of the exponential function)

Long-Path

Problem Description

given : A graph G and a number $B \in \mathbb{N}_0$

to find : is there a path of length B in G

Long-Path

Problem Description

given : A graph G and a number $B \in \mathbb{N}_0$

to find : is there a path of length B in G

NP-Complete

Detour !

Colorful Paths

Problem Description

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k - 1$ in a randomly colored graph ?

colorful :

A path is colorful if all of the vertices in the path have a different color

Colorful Paths

Problem Description

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k - 1$ in a randomly colored graph ?

A path is colorful if all of the vertices in the path have a different color

$P_i(v) := \{S \subseteq [k],\ |S| = i + 1 \ \mid$ There exists a colorful path of length i ending in v with colors $S\}$

$\exists$ colorful path of length $k - 1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \ \mid$ There exists a colorful path of length i ending in v with colors $S\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \ \mid$ There exists a colorful path of length i ending in v with colors $S\}$

- $P_0(v) =$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \ \mid$ There exists a colorful path of length i ending in v with colors $S\}$

- $P_0(v) = \{\{\gamma(v)\}\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \mid$ There exists a colorful path of length i ending in v with colors $S\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

- $P_0(v) = \{\{\gamma(v)\}\}$

- $P_1(v) =$

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \mid$ There exists a colorful path of length i ending in v with colors $S\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

- $P_0(v) = \{\{\gamma(v)\}\}$
- $P_1(v) = \{\{\gamma(x), \gamma(v)\} \mid x \in N(v)$ und $\gamma(x) \neq \gamma(v)\}$

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \mid$ There exists a colorful path of length i ending in v with colors $S\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

- $P_0(v) = \{\{\gamma(v)\}\}$
- $P_1(v) = \{\{\gamma(x), \gamma(v)\} \mid x \in N(v) \text{ und } \gamma(x) \neq \gamma(v)\}$
- $P_i(v) =$

Colorful Paths

$P_i(v)$

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$P_i(v) := \{S \subseteq [k],\ |S| = i+1 \mid$ There exists a colorful path of length i ending in v with colors $S\}$

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

- $P_0(v) = \{\{\gamma(v)\}\}$
- $P_1(v) = \{\{\gamma(x), \gamma(v)\} \mid x \in N(v) \text{ und } \gamma(x) \neq \gamma(v)\}$
- $P_i(v) = \bigcup_{x \in N(v)} \{R \cup \{\gamma(v)\} \mid R \in P_{i-1}(x) \text{ und } \gamma(v) \notin R\}$

Colorful Paths

Algorithm

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

Algorithm 1: COLORFUL(G, i) — G a γ-colored graph

```
1 forall v ∈ V do
2 |   P_i(v) ← ∅;
3 |   forall x ∈ N(v) do
4 |   |   forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5 |   |   |   P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

Algorithm 2: RAINBOW(G, γ) — G a graph, γ a k-coloring

```
1 forall v ∈ V do
2 |   P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4 |   COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Colorful Paths

Algorithm

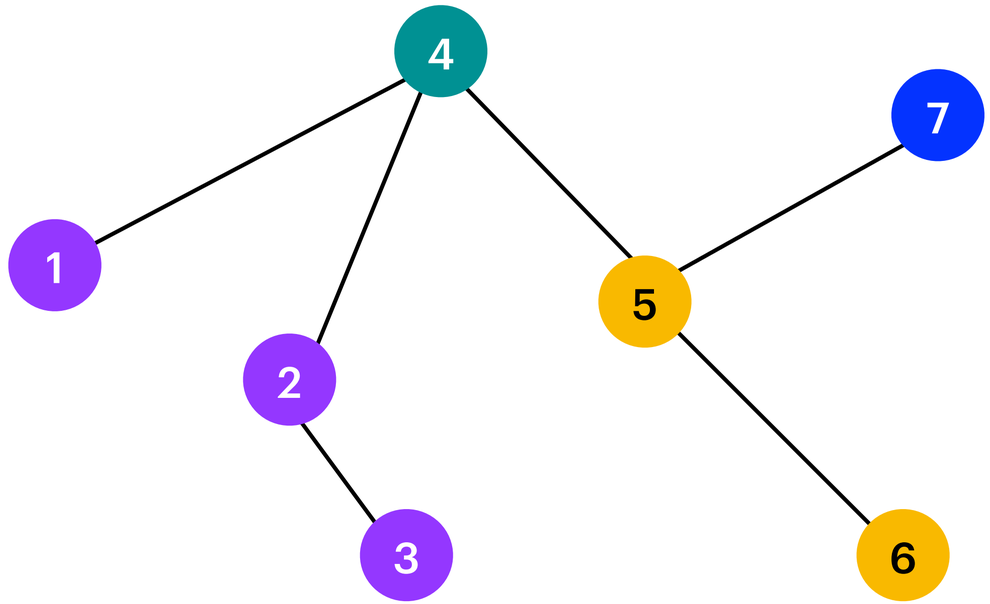
Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_0	
P_0 (1)	
P_0 (2)	
P_0 (3)	
P_0 (4)	
P_0 (5)	
P_0 (6)	
P_0 (7)	



color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

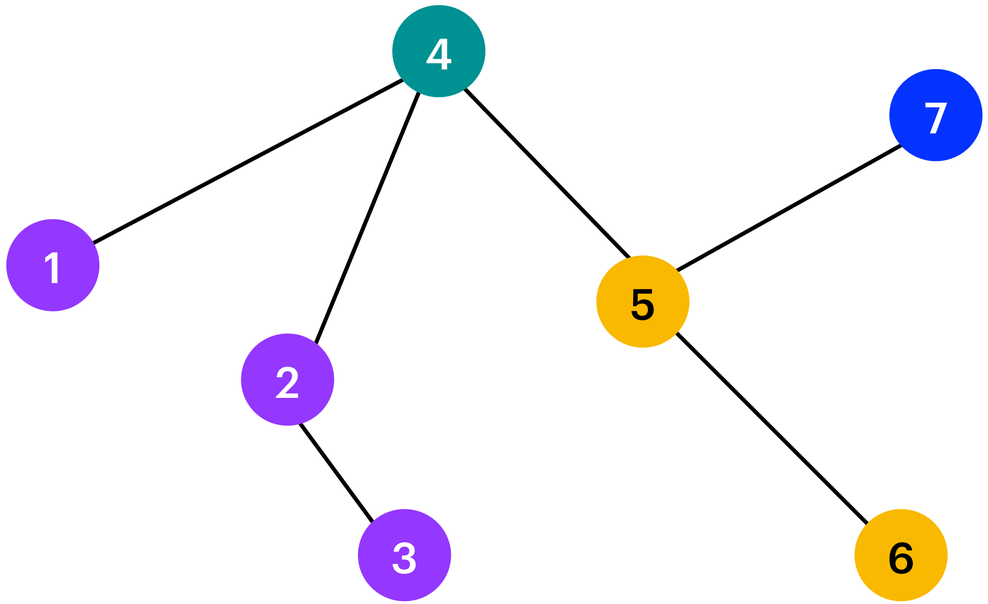
Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_0	
P_0 (1)	{{1}}
P_0 (2)	{{1}}
P_0 (3)	{{1}}
P_0 (4)	{{2}}
P_0 (5)	{{3}
P_0 (6)	{{3}}
P_0 (7)	{{4}}



color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

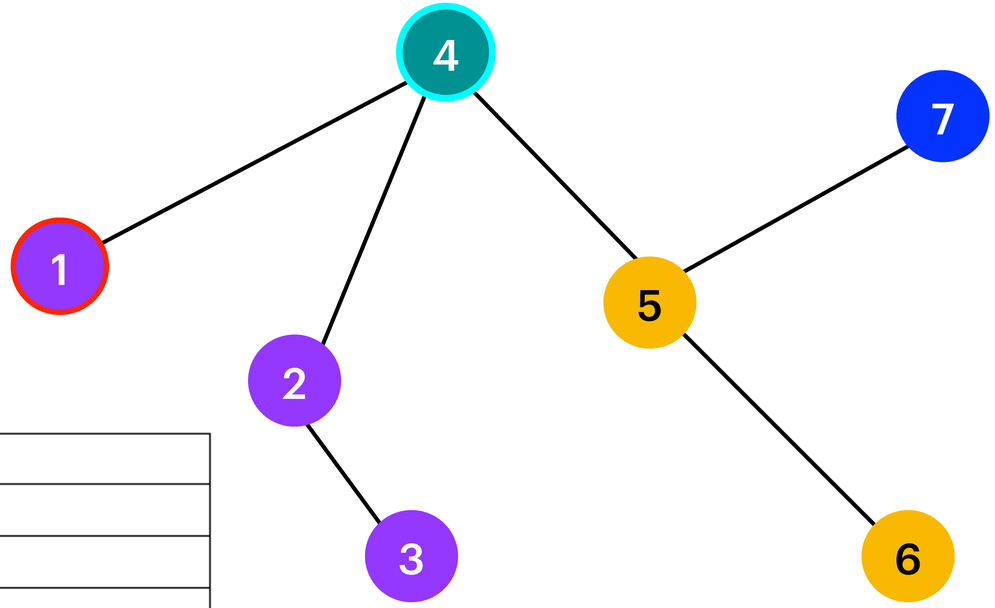
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	
$P_1(2)$	
$P_1(3)$	
$P_1(4)$	
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

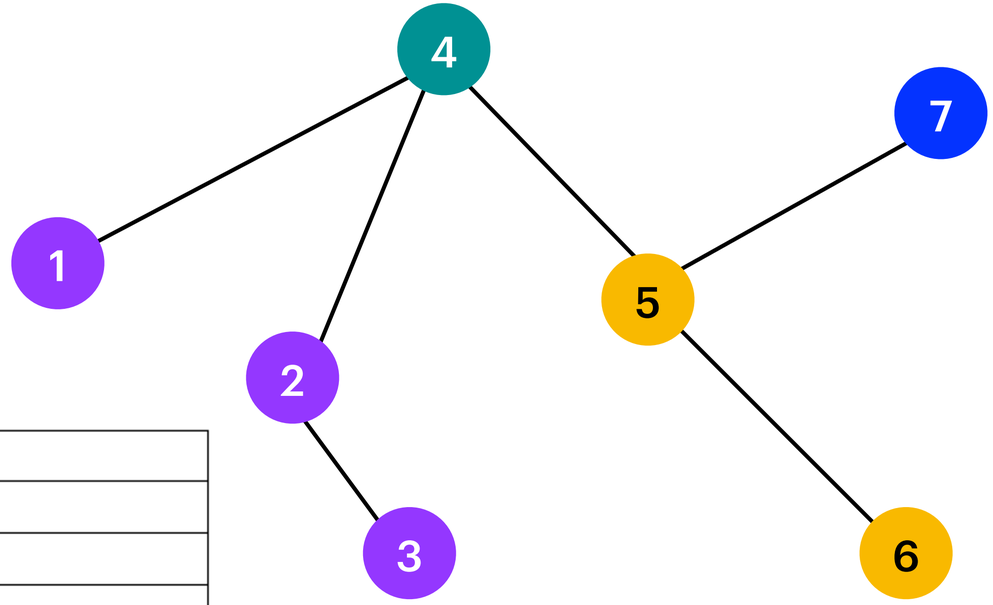
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	
$P_1(3)$	
$P_1(4)$	
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

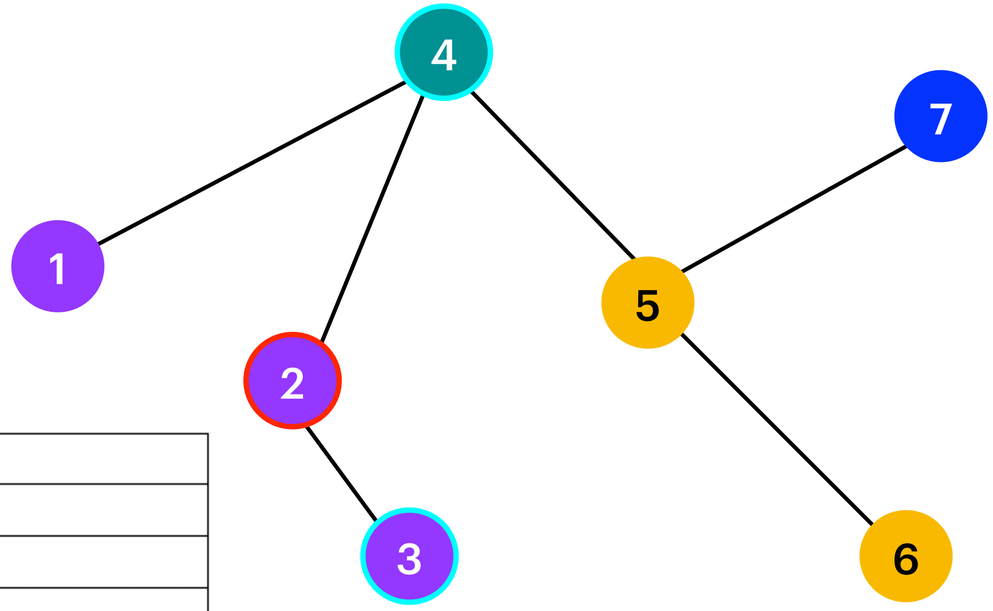
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	
$P_1(3)$	
$P_1(4)$	
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

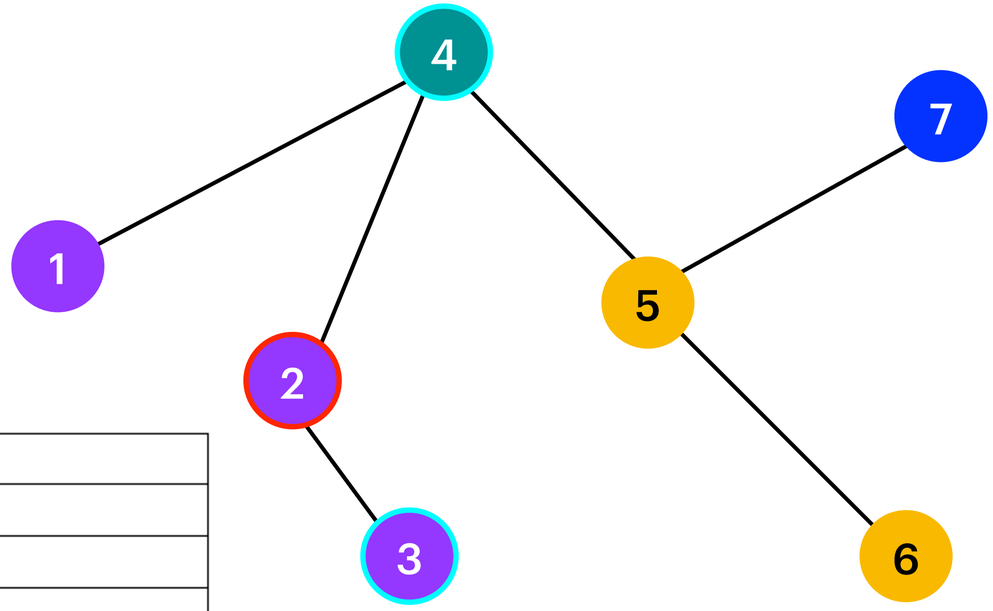
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	
$P_1(4)$	
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

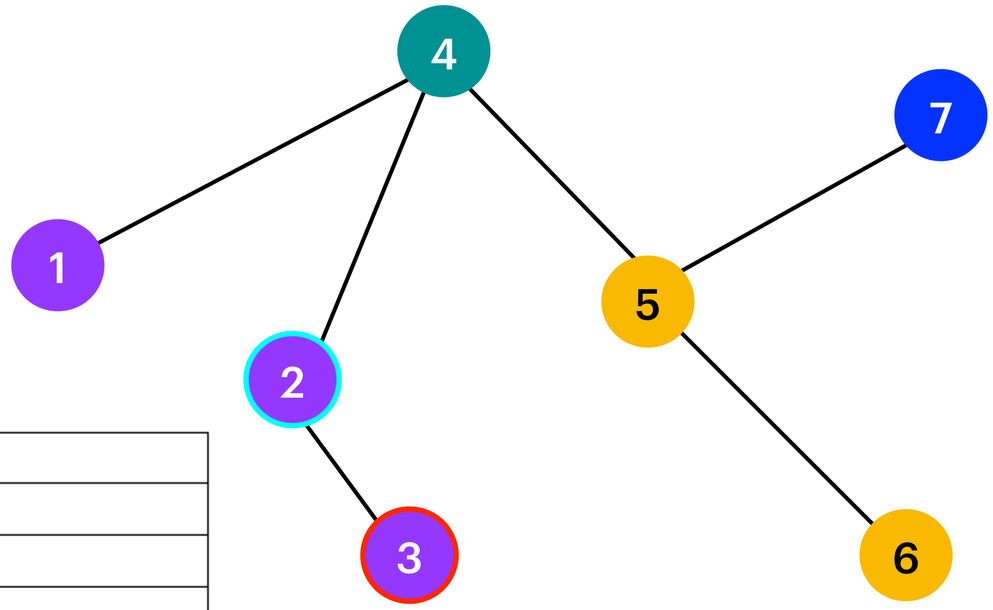
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

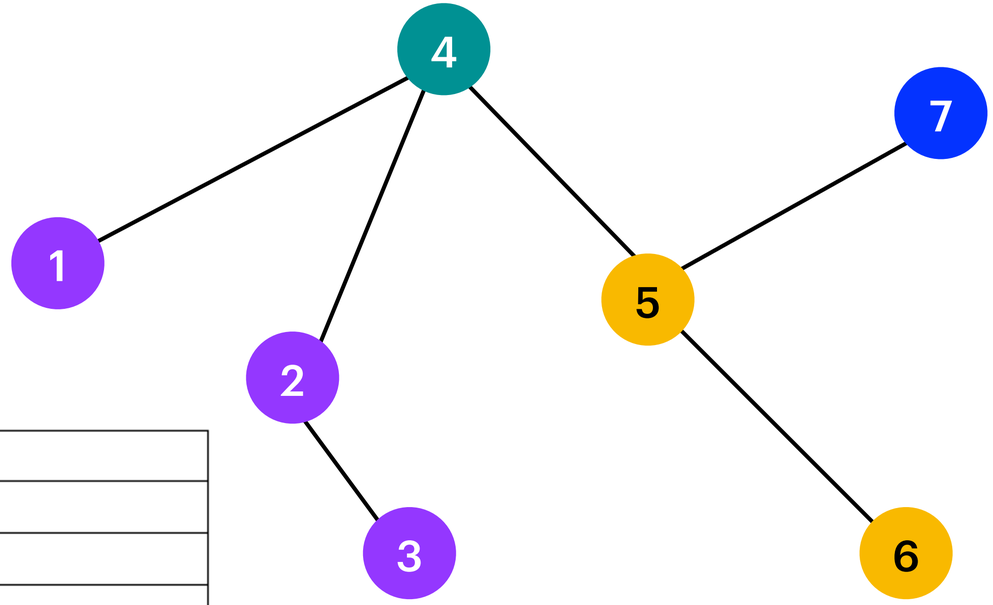
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

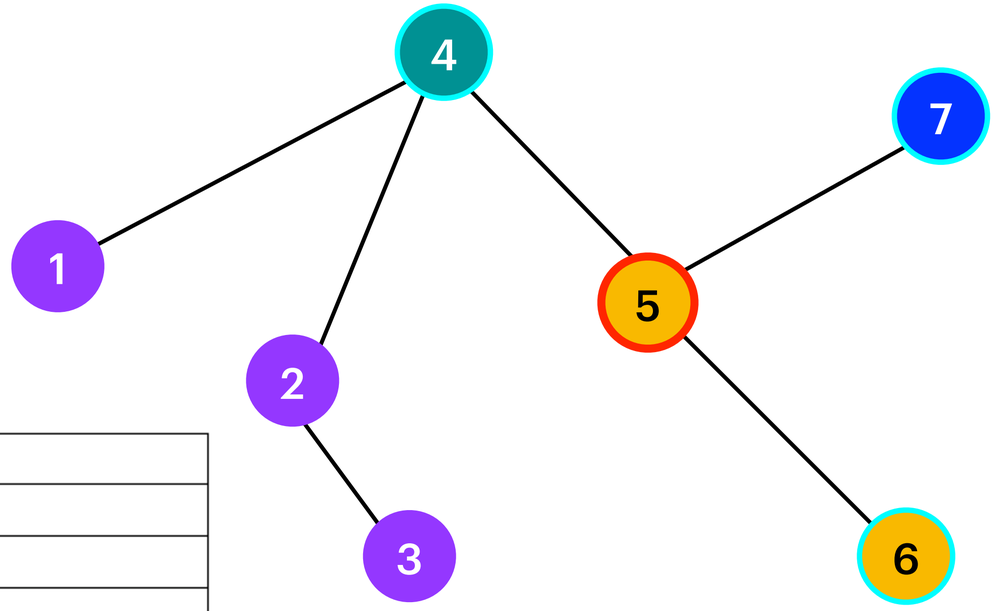
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

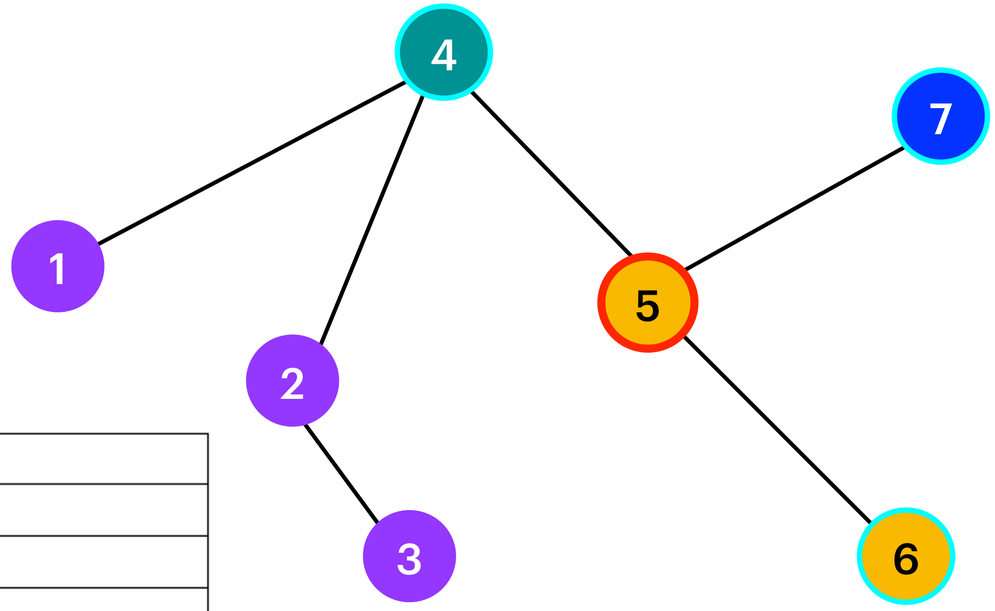
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

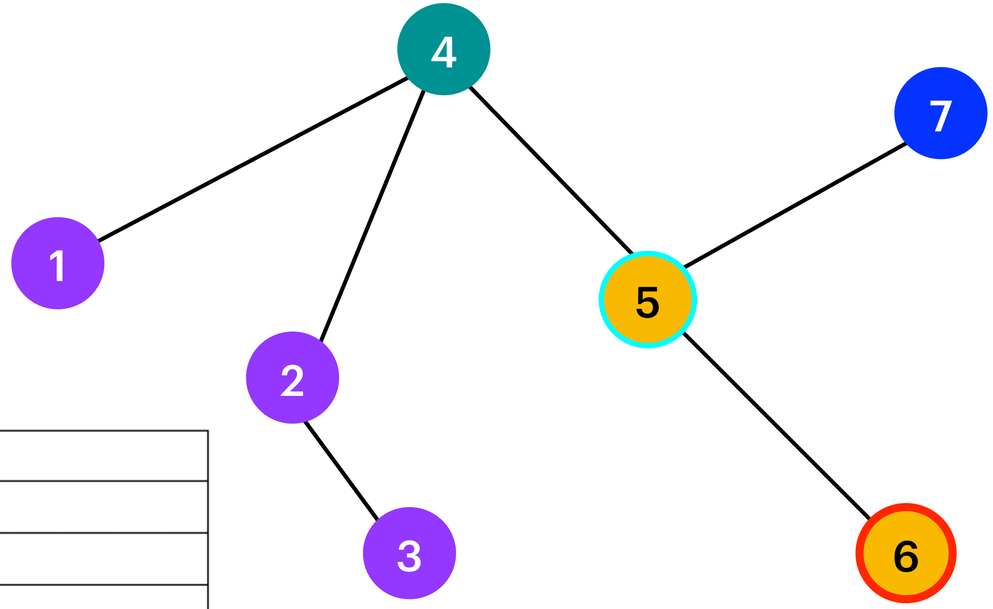
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

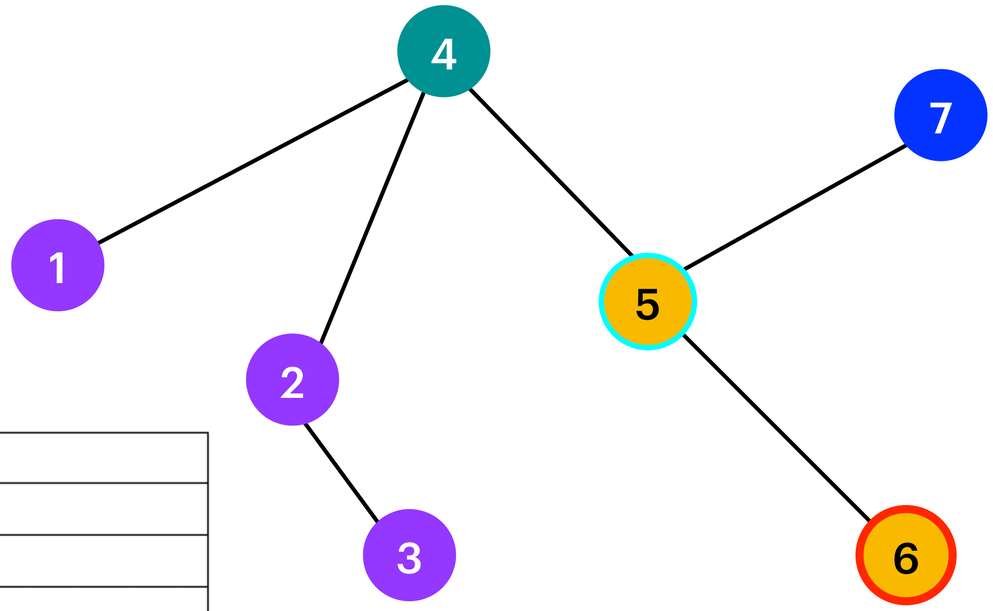
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

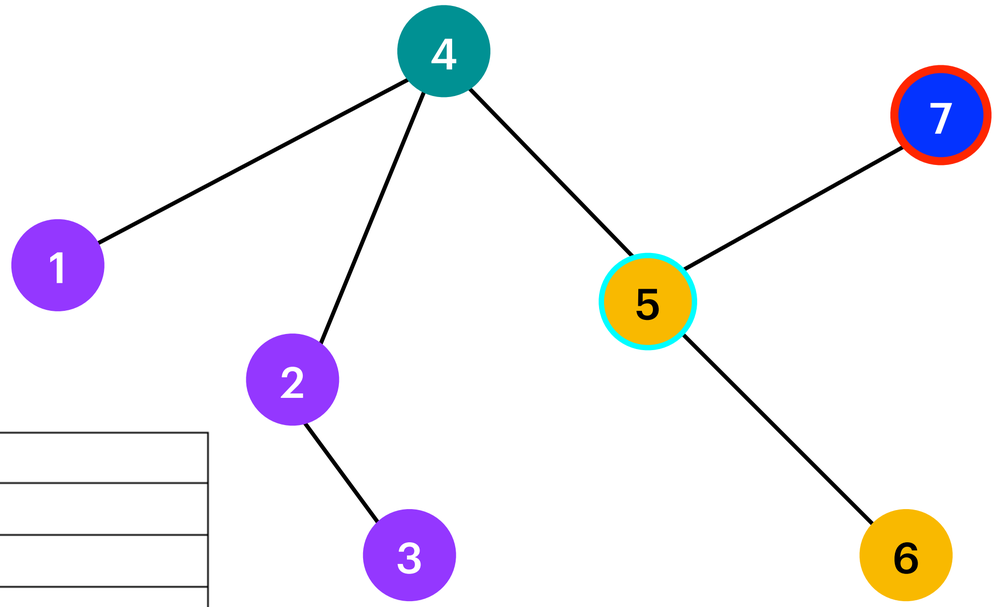
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}}

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_0	
$P_0(1)$	{{1}}
$P_0(2)$	{{1}}
$P_0(3)$	{{1}}
$P_0(4)$	{{2}}
$P_0(5)$	{{3}
$P_0(6)$	{{3}}
$P_0(7)$	{{4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

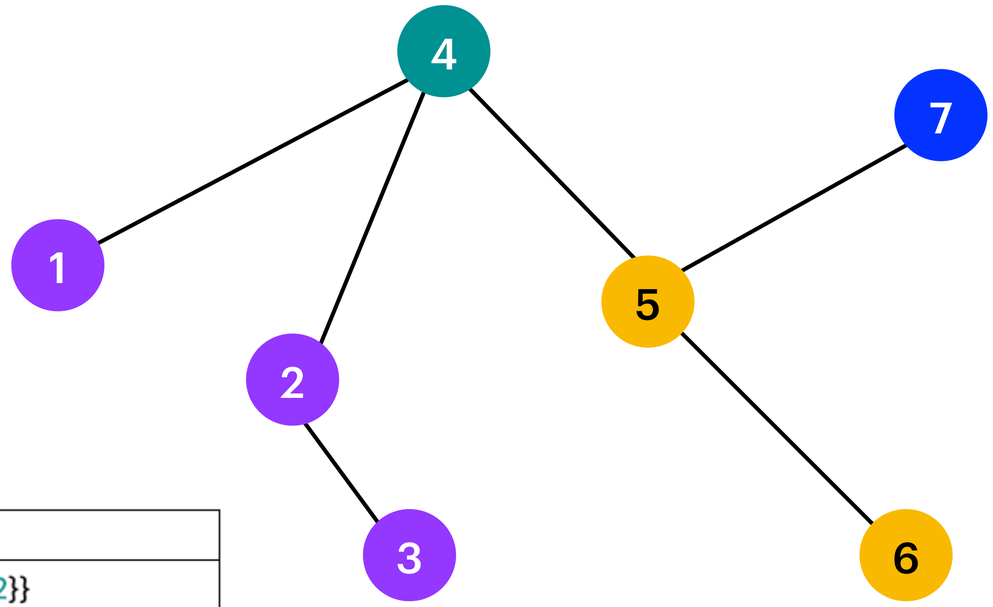
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
P_2 (1)	
P_2 (2)	
P_2 (3)	
P_2 (4)	
P_2 (5)	
P_2 (6)	
P_2 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
P_1 (1)	{{1 , 2}}
P_1 (2)	{{1 , 2}}
P_1 (3)	∅
P_1 (4)	{{1 , 2} , {2 , 3}}
P_1 (5)	{{2 , 3} , {3 , 4}}
P_1 (6)	∅
P_1 (7)	{{3 , 4}}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

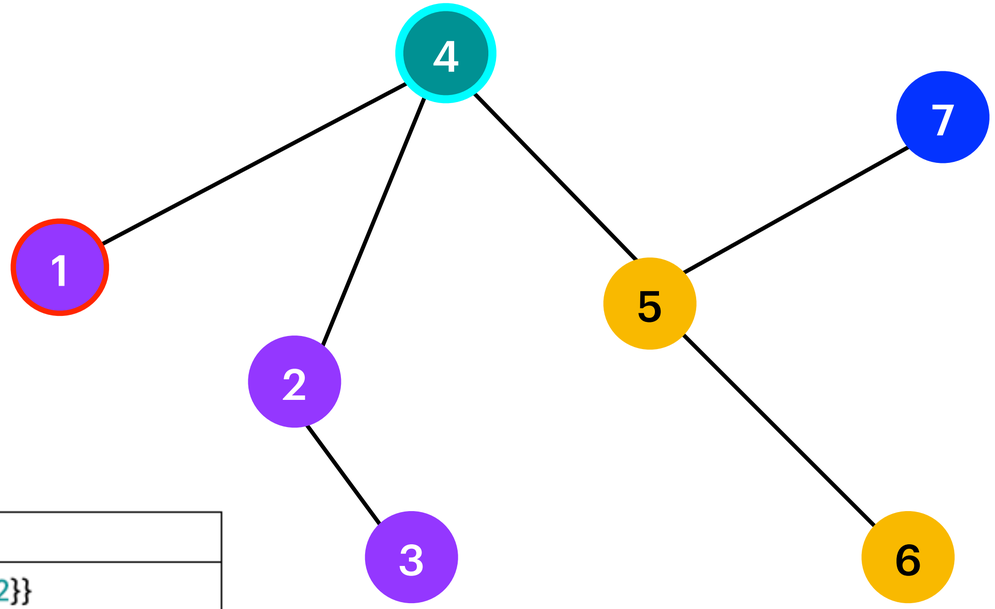
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
P_2 (1)	
P_2 (2)	
P_2 (3)	
P_2 (4)	
P_2 (5)	
P_2 (6)	
P_2 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
P_1 (1)	{{1 , 2}}
P_1 (2)	{{1 , 2}}
P_1 (3)	∅
P_1 (4)	{{1 , 2} , {2 , 3}}
P_1 (5)	{{2 , 3} , {3 , 4}}
P_1 (6)	∅
P_1 (7)	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

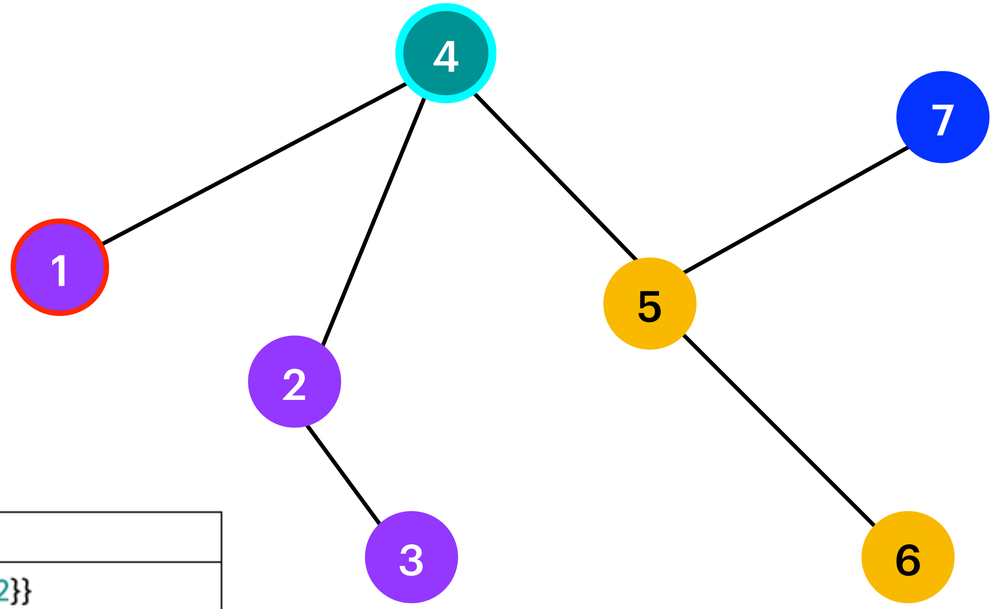
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	
$P_2(3)$	
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

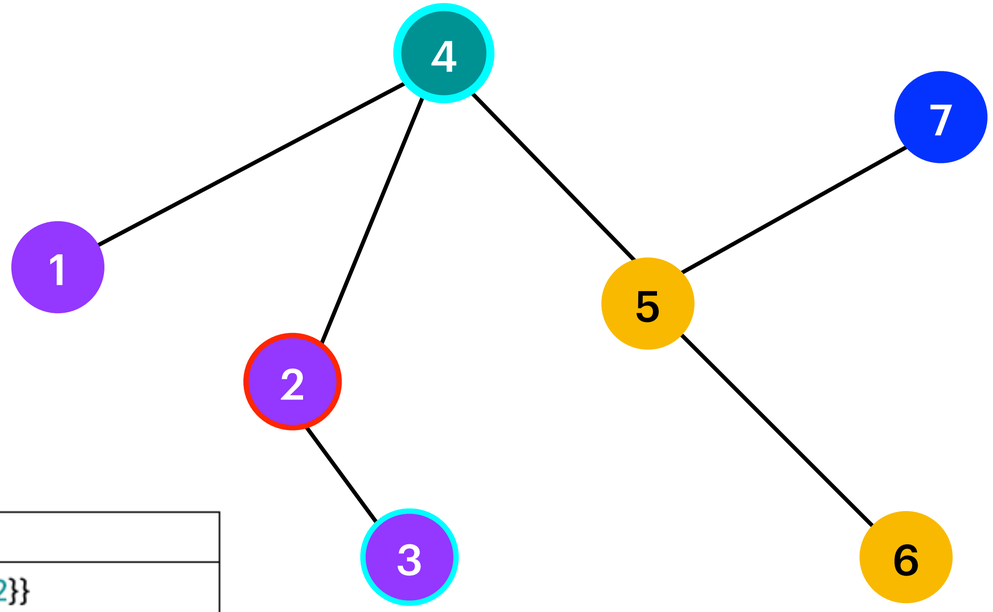
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	
$P_2(3)$	
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

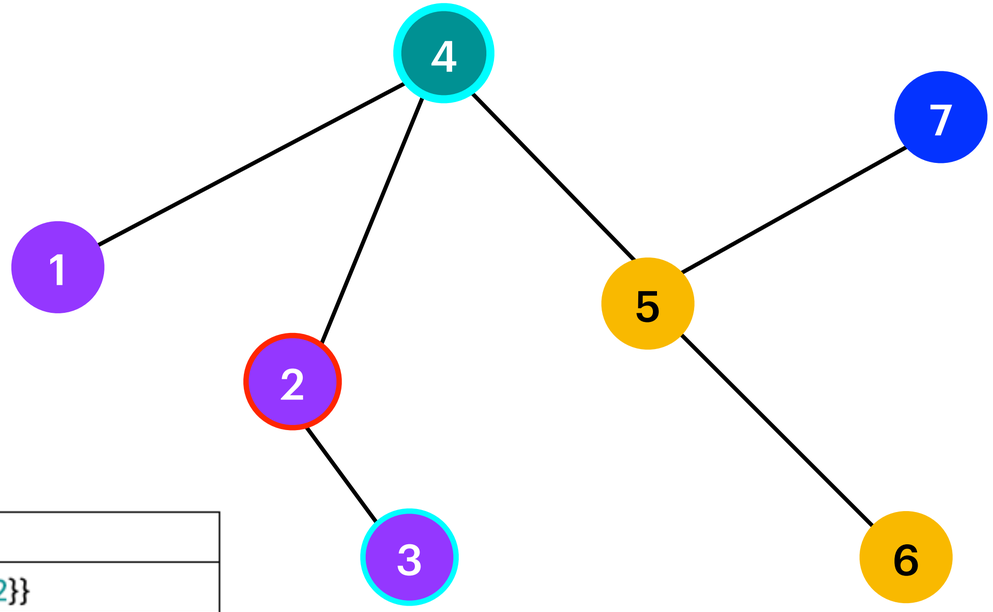
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

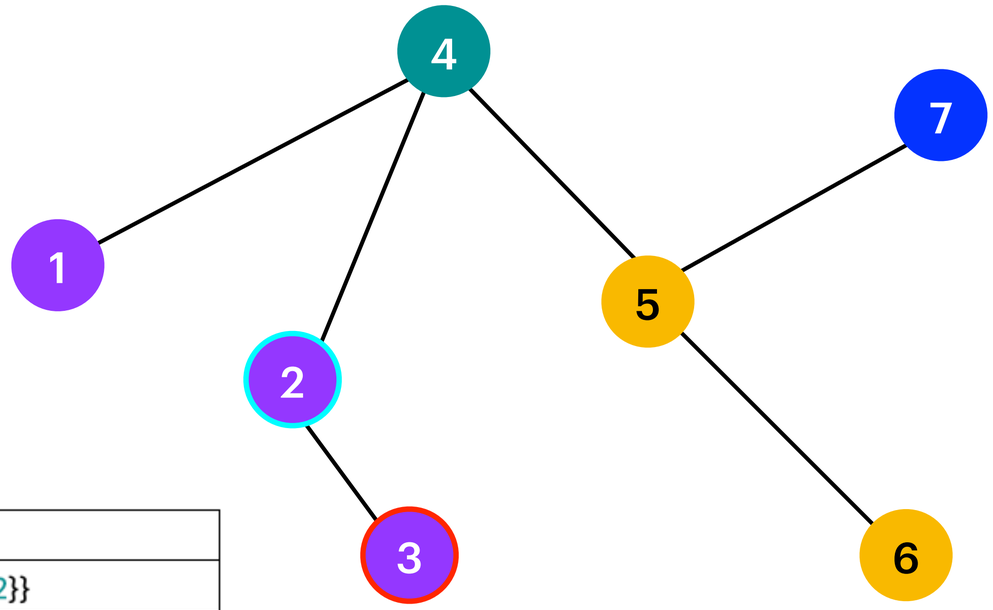
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

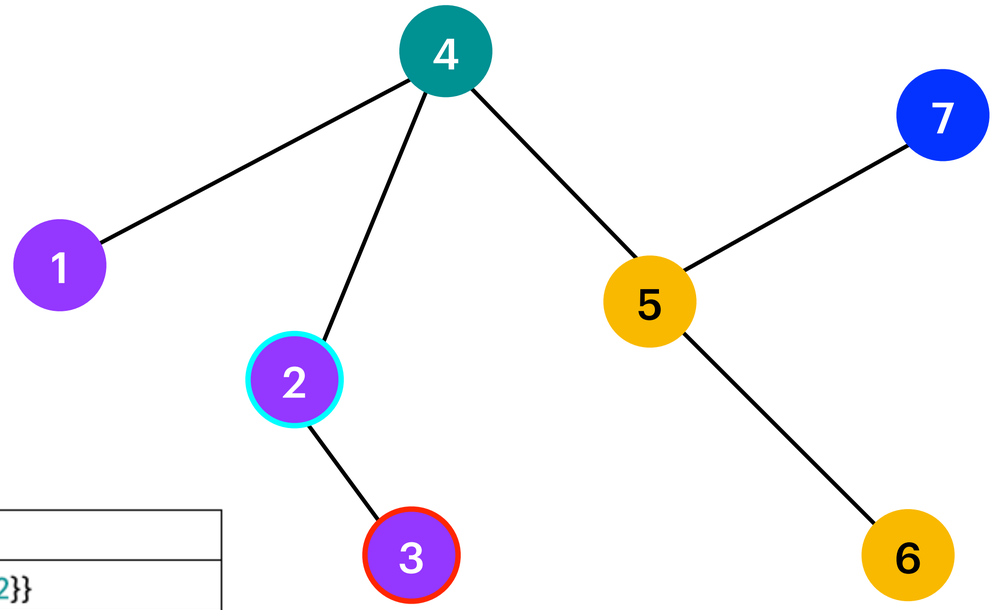
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

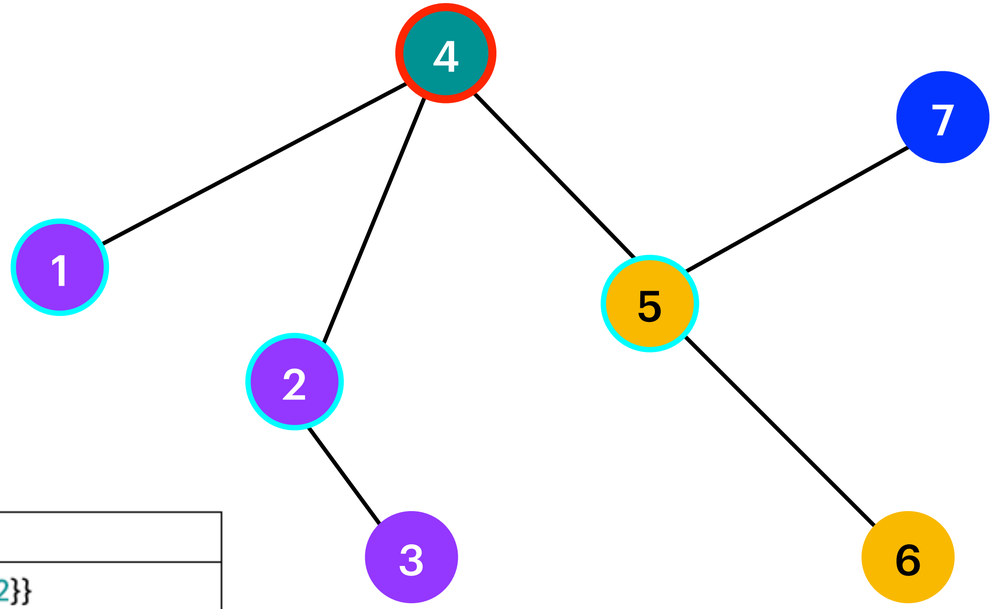
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

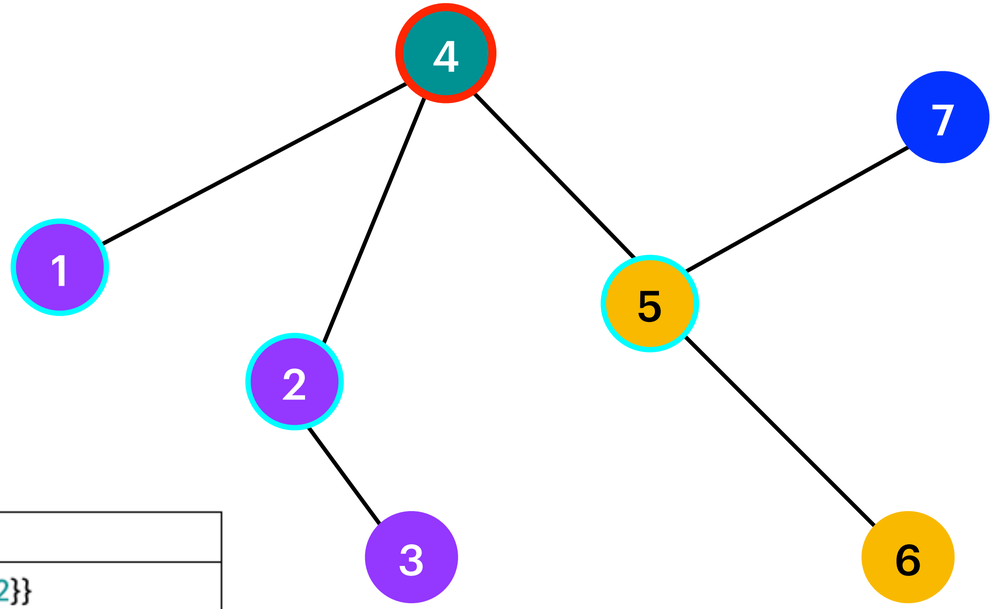
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

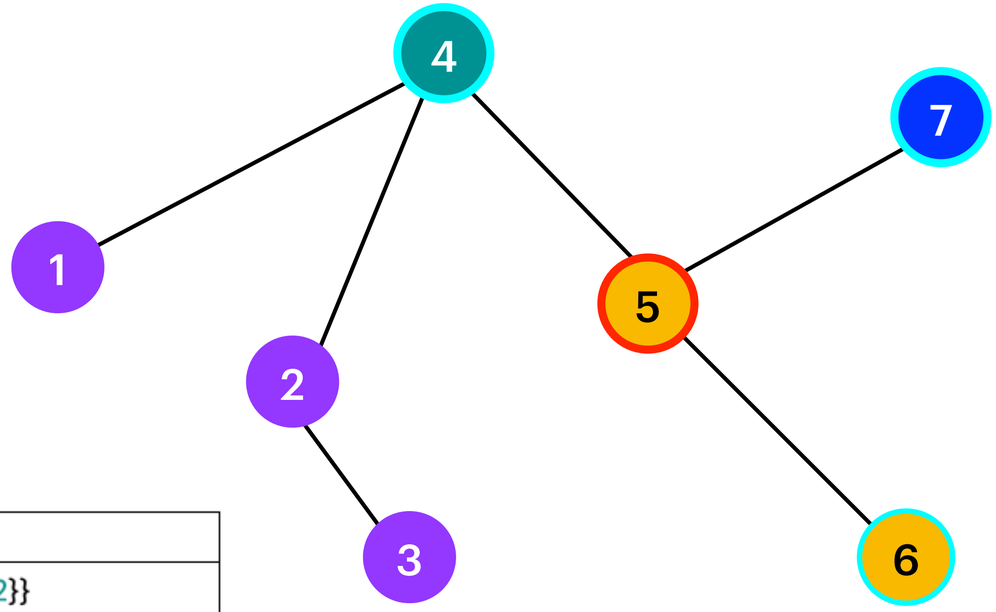
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

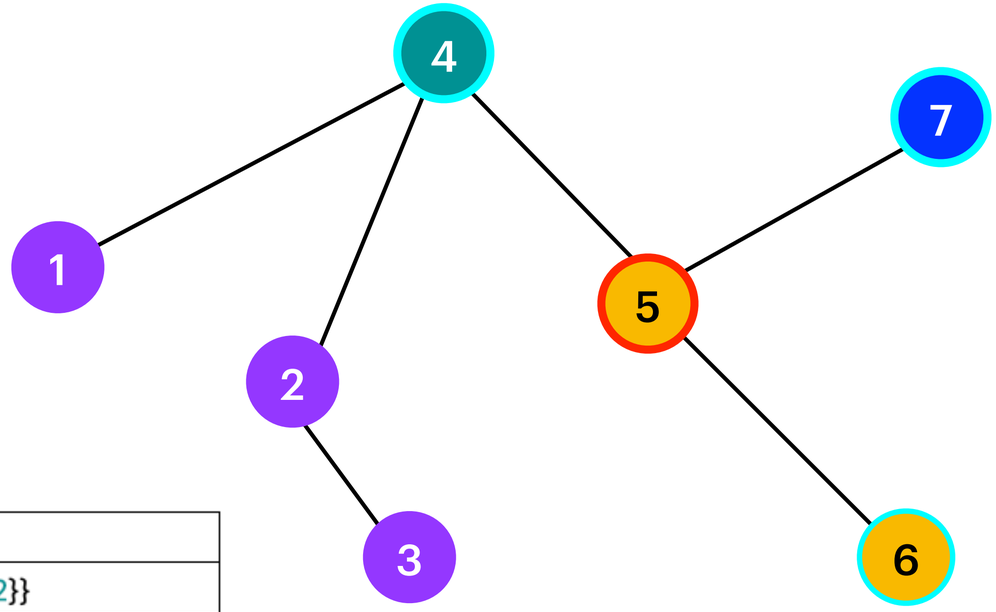
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	{{1 , 2 , 3}}
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

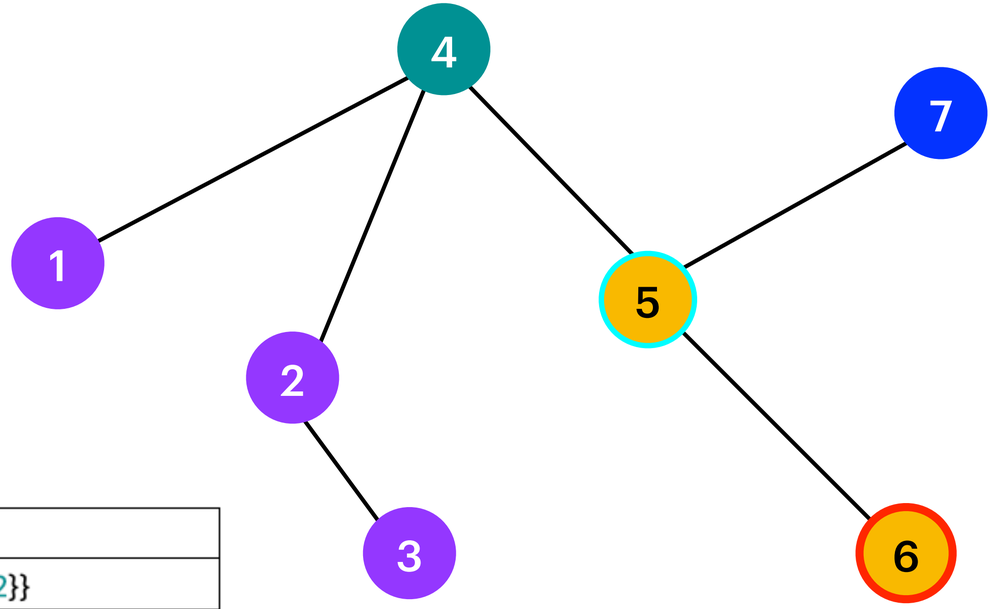
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	{{1 , 2 , 3}}
$P_2(6)$	
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

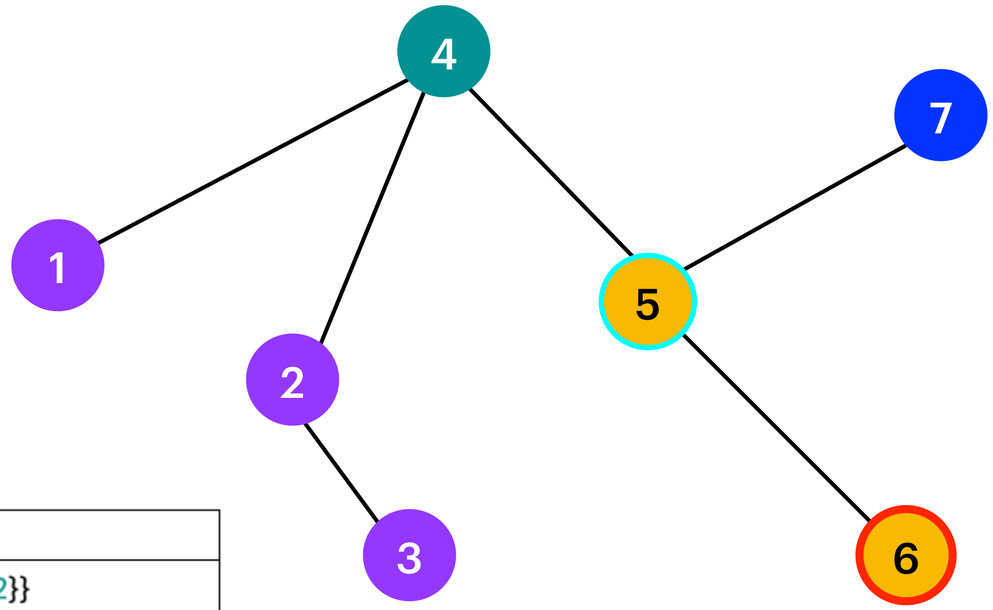
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	{{1 , 2 , 3}}
$P_2(6)$	∅
$P_2(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

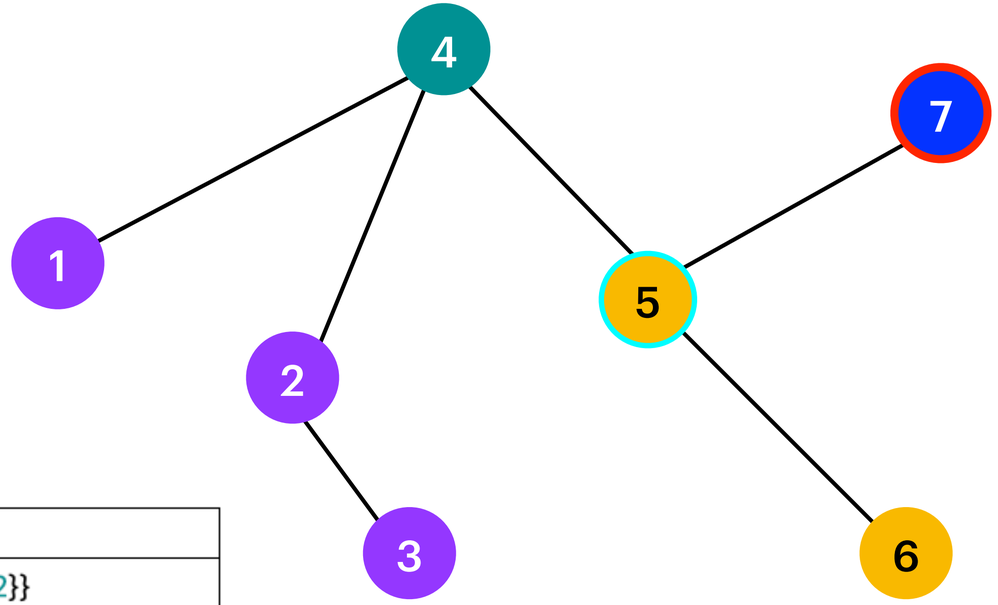
Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_2	
$P_2(1)$	{{1 , 2 , 3}}
$P_2(2)$	{{1 , 2 , 3}}
$P_2(3)$	∅
$P_2(4)$	{{2 , 3 , 4}}
$P_2(5)$	{{1 , 2 , 3}}
$P_2(6)$	∅
$P_2(7)$	{{2 , 3 , 4}}

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

P_1	
$P_1(1)$	{{1 , 2}}
$P_1(2)$	{{1 , 2}}
$P_1(3)$	∅
$P_1(4)$	{{1 , 2} , {2 , 3}}
$P_1(5)$	{{2 , 3} , {3 , 4}}
$P_1(6)$	∅
$P_1(7)$	{{3 , 4}}



γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

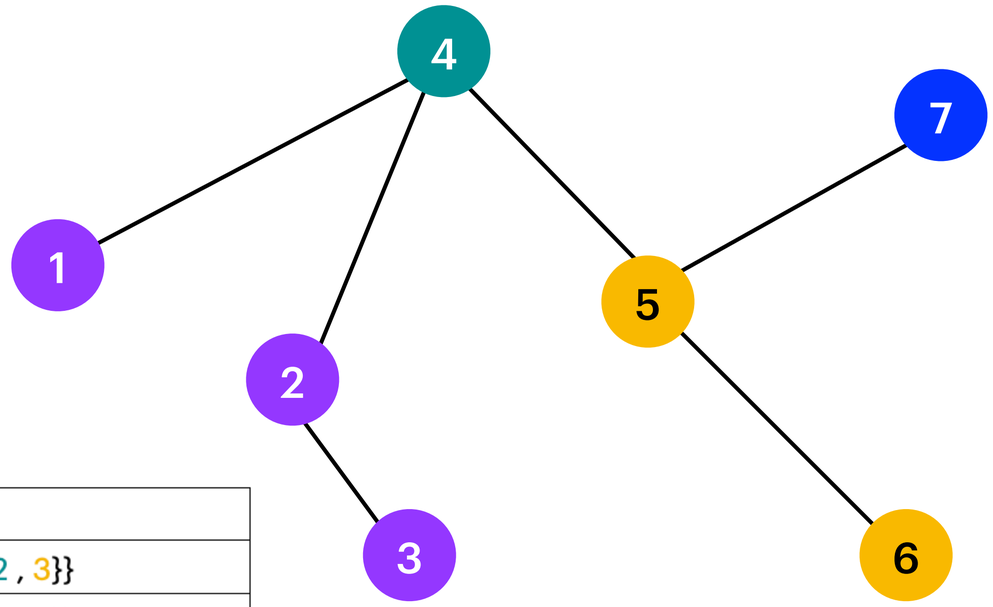
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	
P_3 (2)	
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

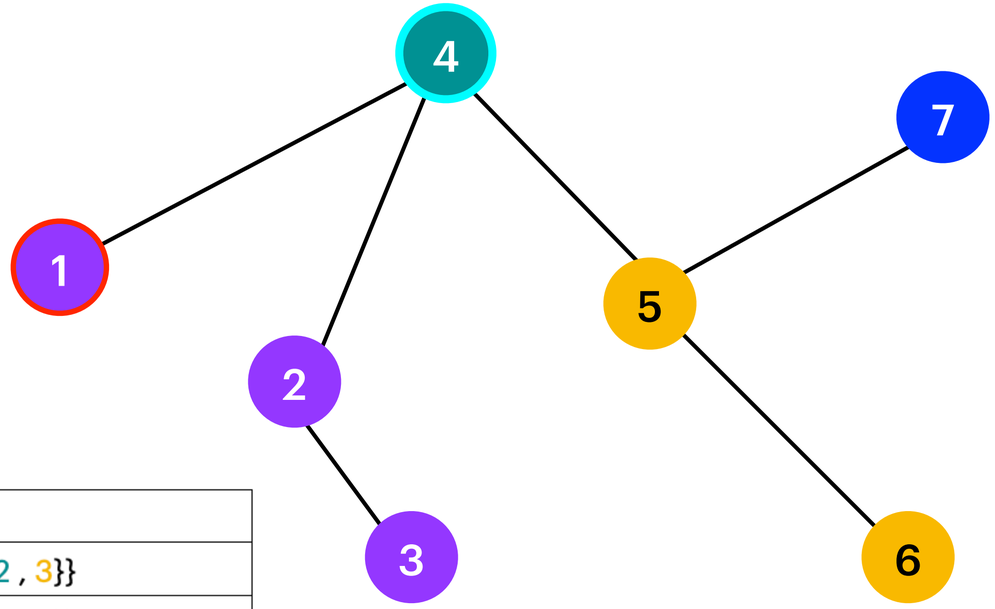
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	
P_3 (2)	
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

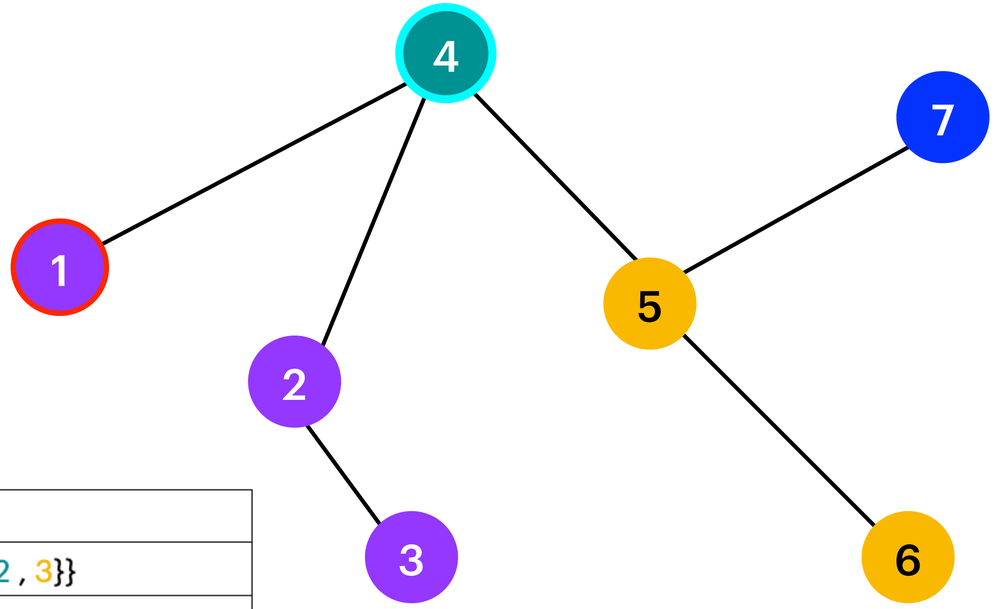
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

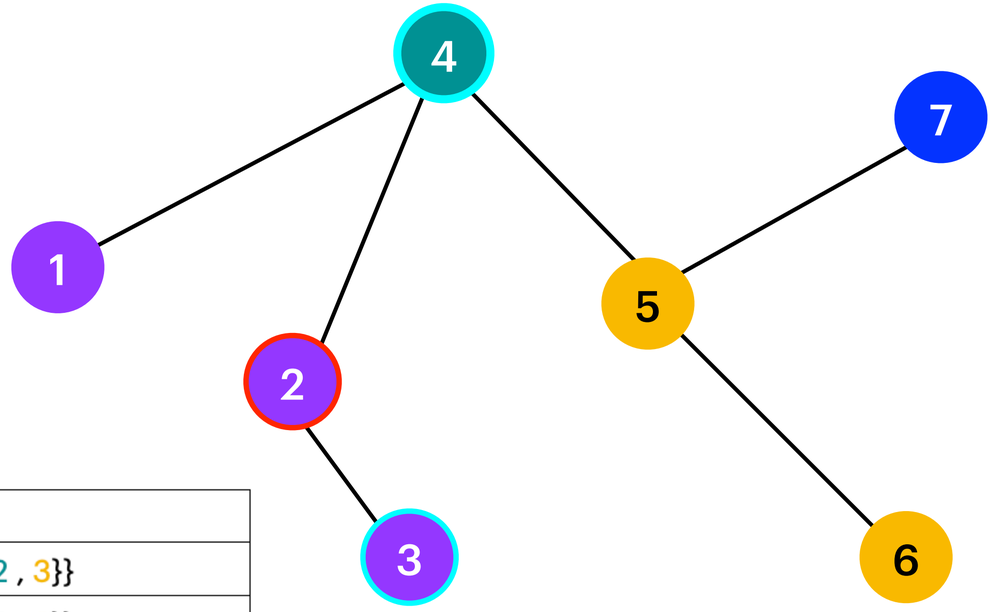
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

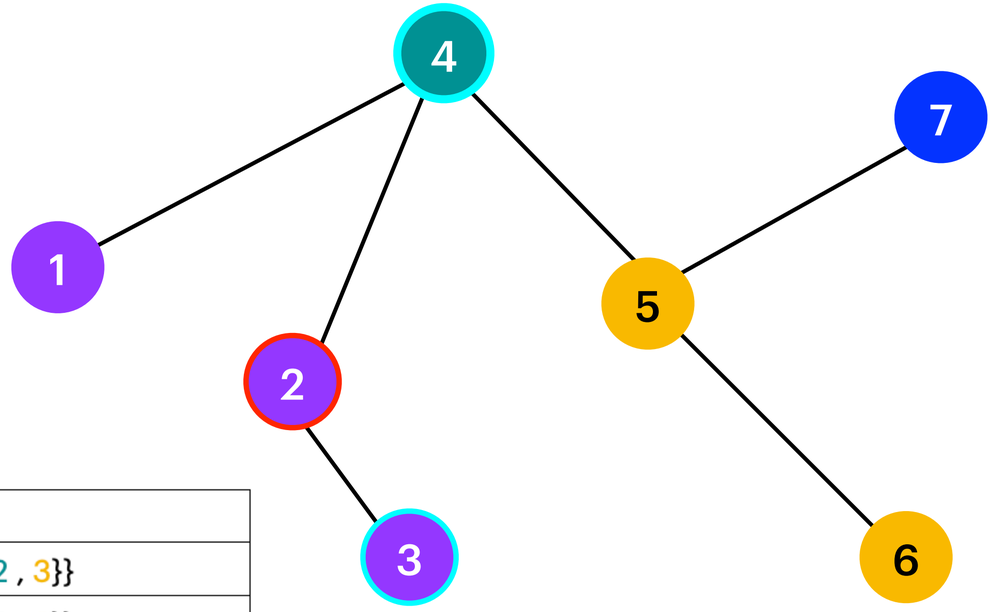
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

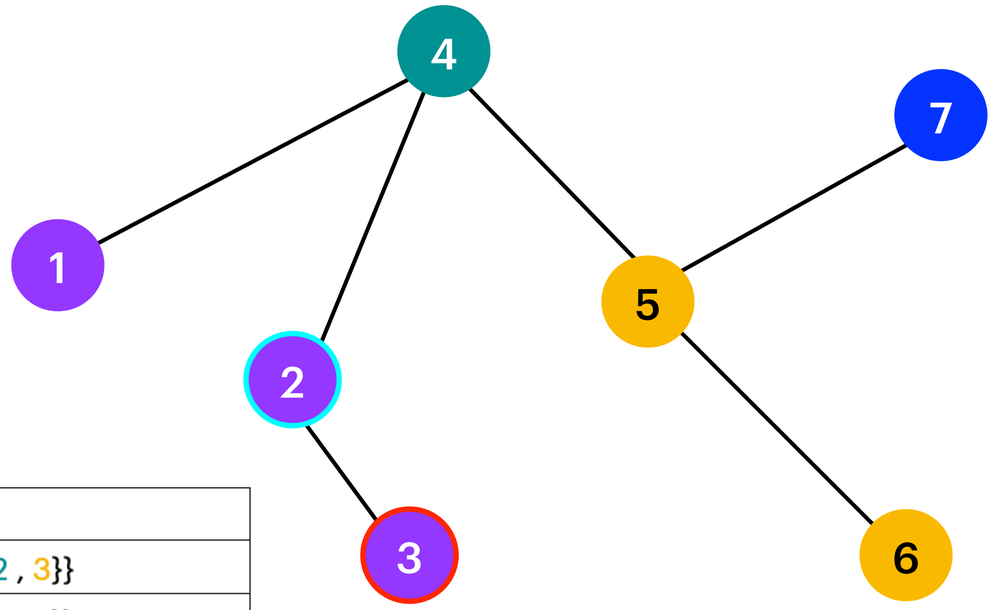
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

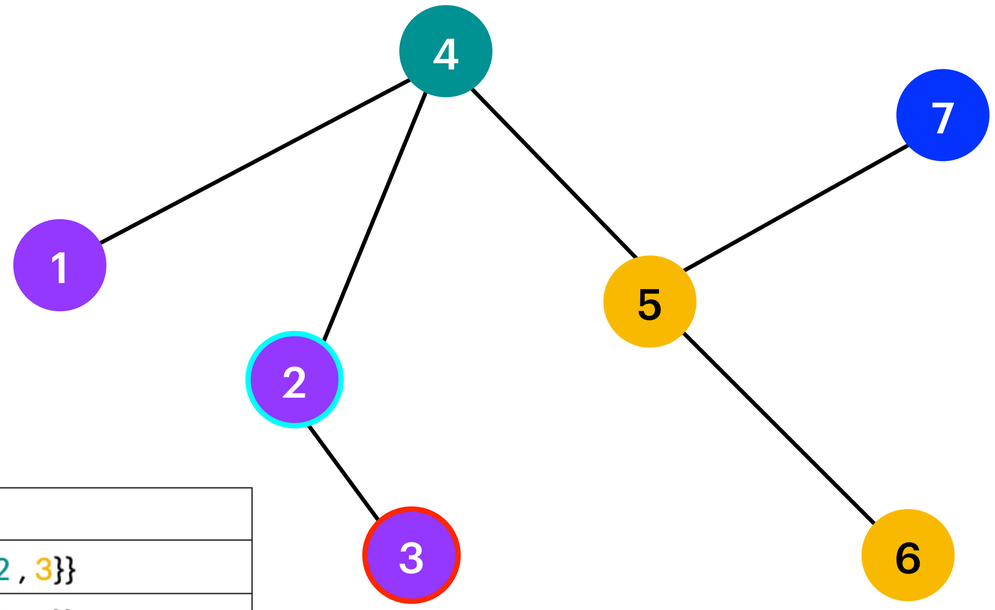
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
$P_3(1)$	$\{\{1, 2, 3, 4\}\}$
$P_3(2)$	$\{\{1, 2, 3, 4\}\}$
$P_3(3)$	$\varnothing$
$P_3(4)$	
$P_3(5)$	
$P_3(6)$	
$P_3(7)$	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
$P_2(1)$	$\{\{1, 2, 3\}\}$
$P_2(2)$	$\{\{1, 2, 3\}\}$
$P_2(3)$	$\varnothing$
$P_2(4)$	$\{\{2, 3, 4\}\}$
$P_2(5)$	$\{\{1, 2, 3\}\}$
$P_2(6)$	$\varnothing$
$P_2(7)$	$\{\{2, 3, 4\}\}$

γ 1, 2, 3, 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

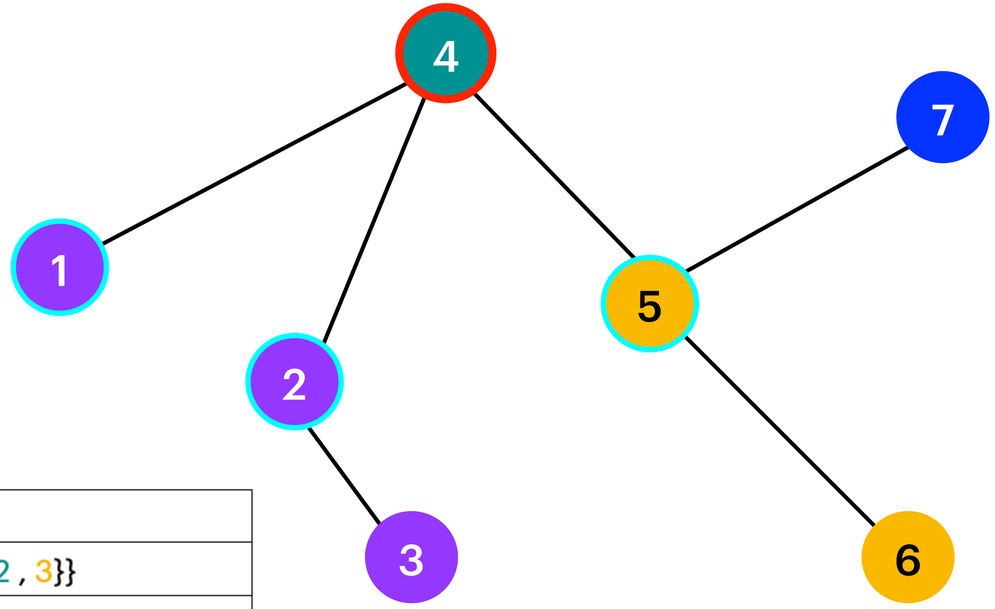
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

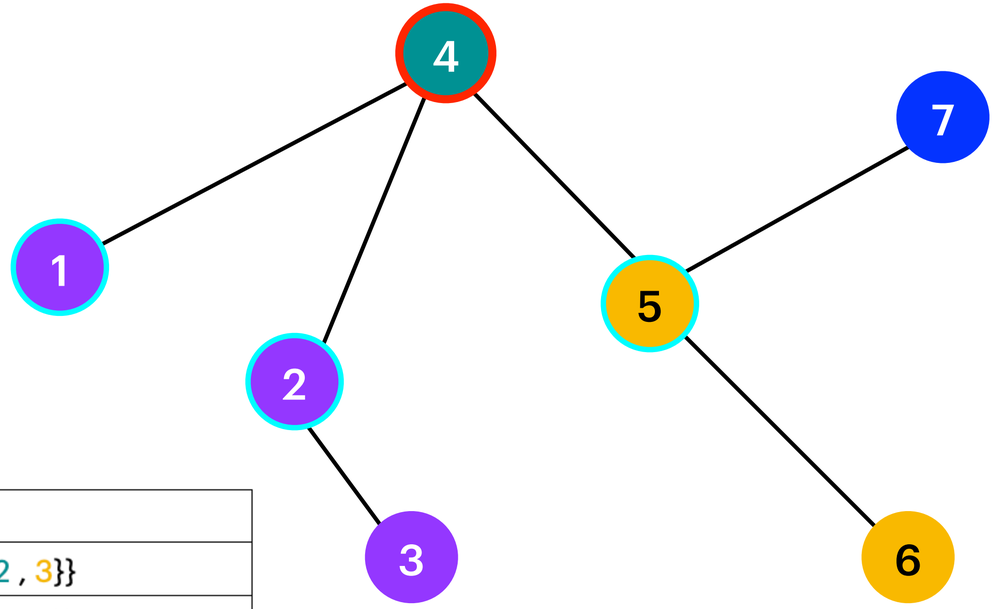
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

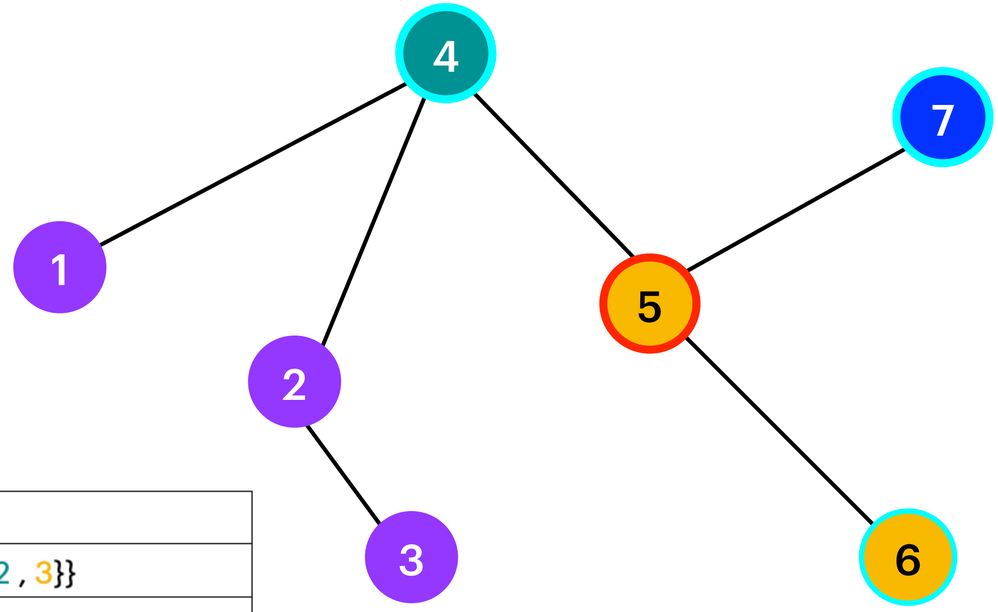
```
forall v ∈ V do
    P_0(v) ← {{γ(v)}};
for i = 1 to k − 1 do
    COLORFUL(G, i);
return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
forall v ∈ V do
    P_i(v) ← ∅;
    forall x ∈ N(v) do
        forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
            P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

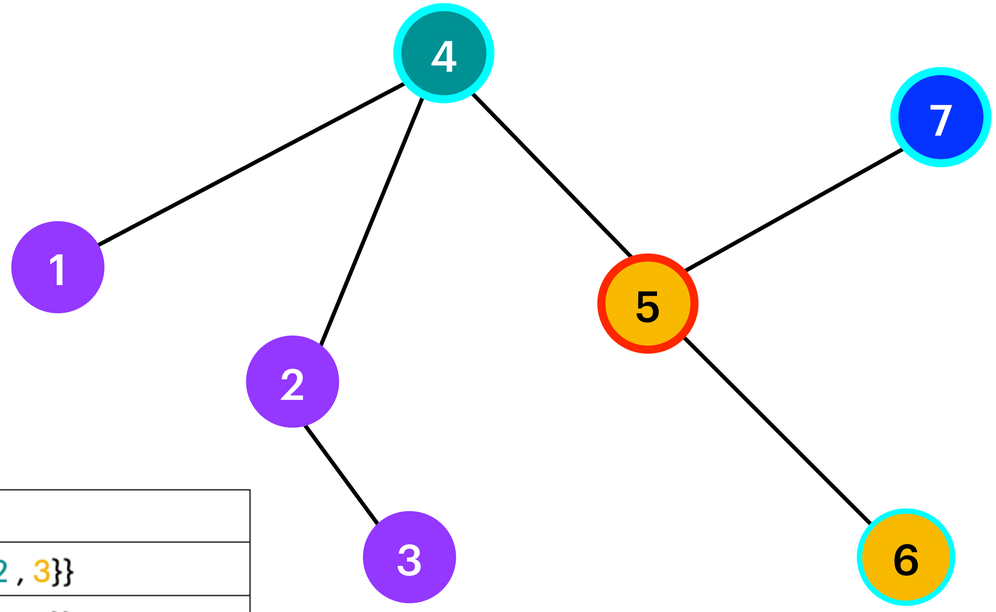
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	∅
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

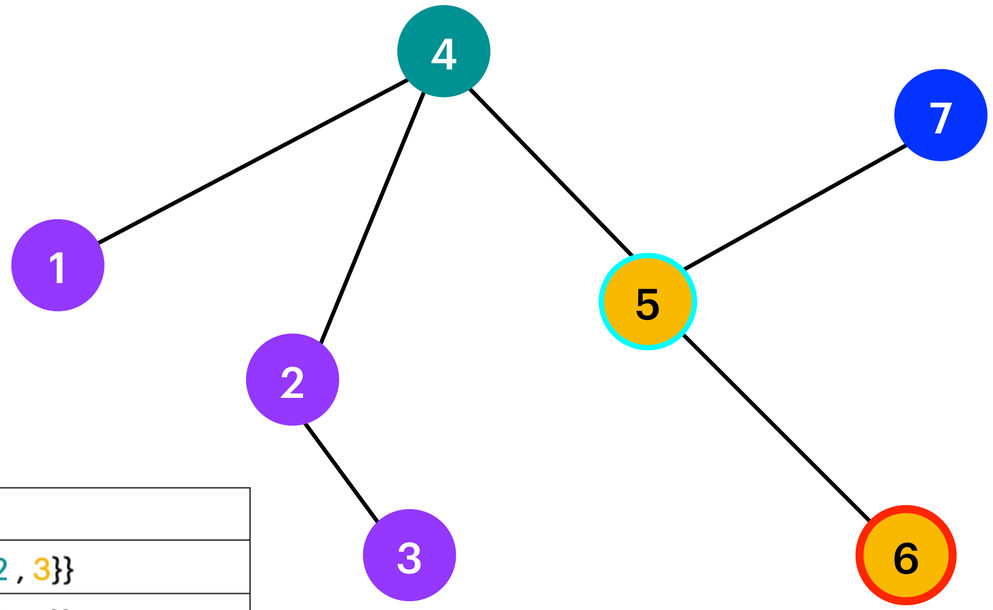
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	∅
P_3 (6)	
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

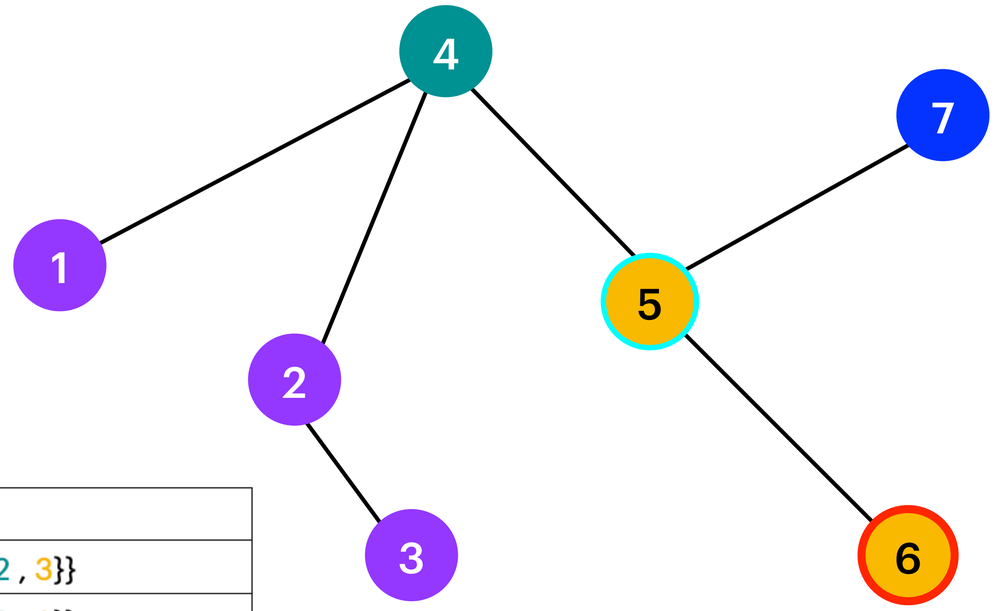
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	$\{\{1, 2, 3, 4\}\}$
P_3 (2)	$\{\{1, 2, 3, 4\}\}$
P_3 (3)	$\varnothing$
P_3 (4)	$\varnothing$
P_3 (5)	$\varnothing$
P_3 (6)	$\varnothing$
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

Algorithm 2: RAINBOW(G, γ)

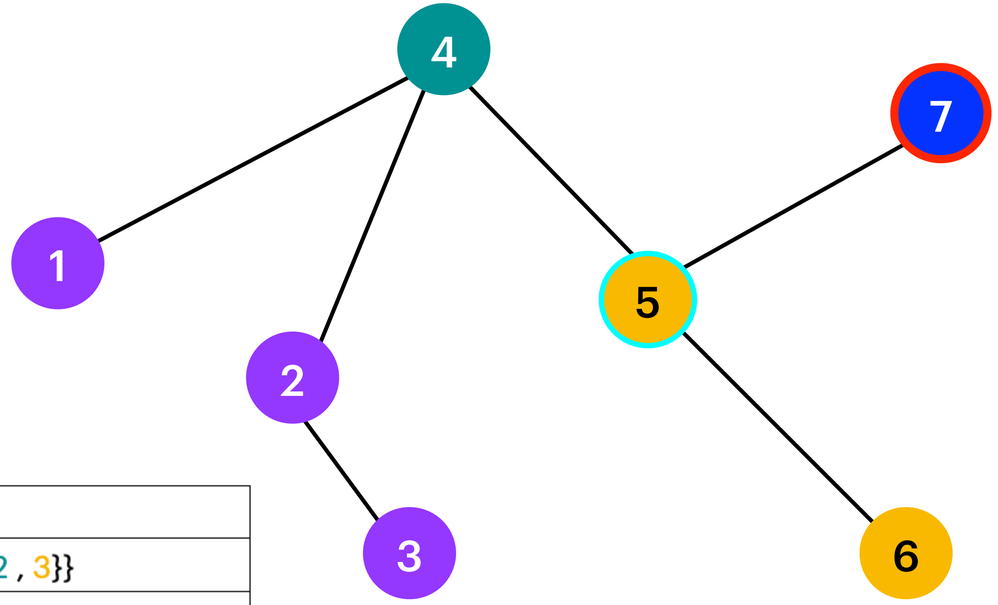
```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	∅
P_3 (6)	∅
P_3 (7)	

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S



P_2	
P_2 (1)	{{1 , 2 , 3}}
P_2 (2)	{{1 , 2 , 3}}
P_2 (3)	∅
P_2 (4)	{{2 , 3 , 4}}
P_2 (5)	{{1 , 2 , 3}}
P_2 (6)	∅
P_2 (7)	{{2 , 3 , 4}}

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

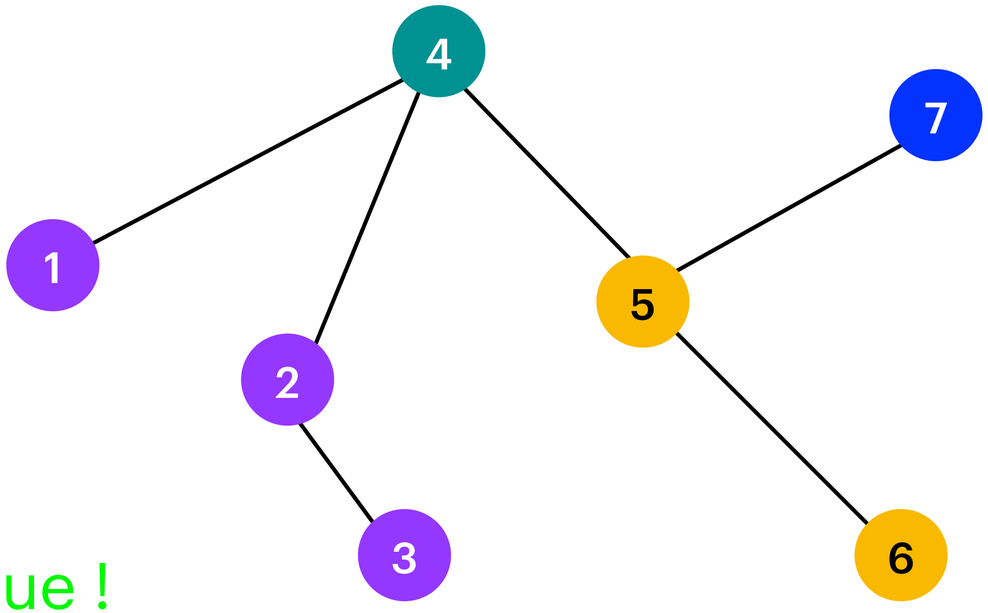
Algorithm 2: RAINBOW(G, γ)

```
1 forall v ∈ V do
2     P_0(v) ← {{γ(v)}};
3 for i = 1 to k − 1 do
4     COLORFUL(G, i);
5 return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
1 forall v ∈ V do
2     P_i(v) ← ∅;
3     forall x ∈ N(v) do
4         forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
5             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	$\{\{1, 2, 3, 4\}\}$
P_3 (2)	$\{\{1, 2, 3, 4\}\}$
P_3 (3)	$\varnothing$
P_3 (4)	$\varnothing$
P_3 (5)	$\varnothing$
P_3 (6)	$\varnothing$
P_3 (7)	$\{\{1, 2, 3, 4\}\}$



returns true !

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

γ 1 , 2 , 3 , 4

Colorful Paths

Algorithm

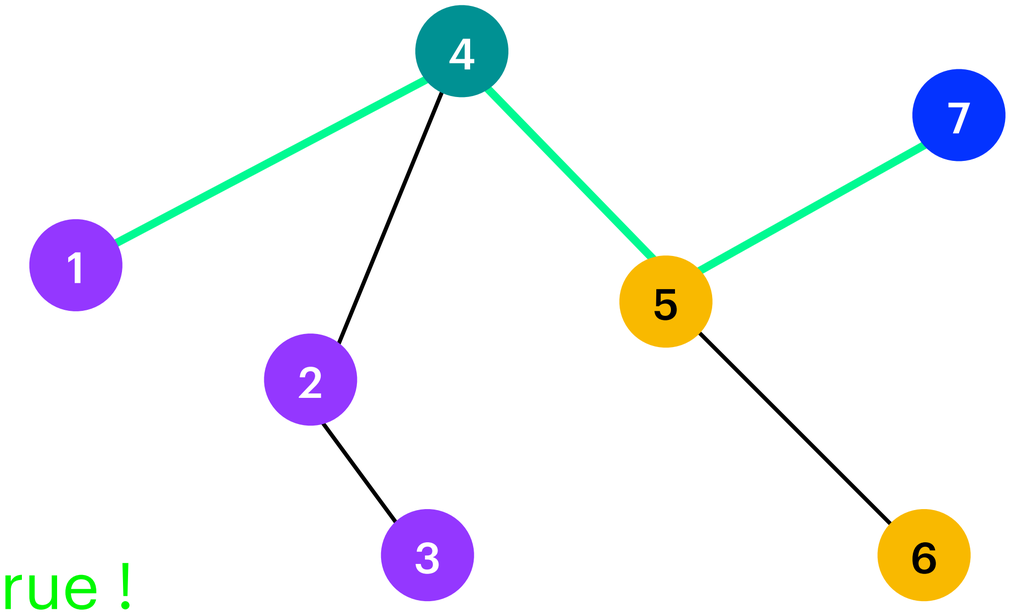
Algorithm 2: RAINBOW(G, γ)

```
forall v ∈ V do
    P_0(v) ← {{γ(v)}};
for i = 1 to k − 1 do
    COLORFUL(G, i);
return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Algorithm 1: COLORFUL(G, i)

```
forall v ∈ V do
    P_i(v) ← ∅;
    forall x ∈ N(v) do
        forall R ∈ P_{i−1}(x) such that γ(v) ∉ R do
            P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

P_3	
P_3 (1)	{{1 , 2 , 3 , 4}}
P_3 (2)	{{1 , 2 , 3 , 4}}
P_3 (3)	∅
P_3 (4)	∅
P_3 (5)	∅
P_3 (6)	∅
P_3 (7)	{{1 , 2 , 3 , 4}}



returns true !

color sets S s.t |S| = i+1 and there is a colorful path of length i ending at v only using the colors in S

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k - 1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

1 , 2 , 3 , 4

Colorful Paths

Algorithm + Probability

given : A graph $G = (V, E)$

A coloring of its vertices with k colors $\gamma : V \to [k]$

to find : Does there exist a colorful path of length $k-1$ in a randomly colored graph ? A path is colorful if all of the vertices in the path have a different color

$\exists$ colorful path of length $k-1 \iff \bigcup_{v \in V} P_{k-1}(v) \neq \emptyset$

$$\mathcal{O}(2^k \cdot k \cdot m)$$

Algorithm 1: COLORFUL(G, i) — G a γ-colored graph

```
forall v ∈ V do
    P_i(v) ← ∅;
    forall x ∈ N(v) do
        forall R ∈ P_{i-1}(x) such that γ(v) ∉ R do
            P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}};
```

Algorithm 2: RAINBOW(G, γ) — G a graph, γ a k-coloring

```
forall v ∈ V do
    P_0(v) ← {{γ(v)}};
for i = 1 to k − 1 do
    COLORFUL(G, i);
return ⋃_{v∈V} P_{k−1}(v) ≠ ∅;
```

Satz 3.2

Let G be a graph that contains a path of length $k-1$.

1. A random coloring of the graph using k colors produces a colorful path of length $k-1$ with probability at least

$$p_{\text{success}} \geq \frac{k!}{k^k} \geq e^{-k}$$

2. If we repeat the coloring process multiple times, then the **expected number of repetitions** needed until success is at most

$$\frac{1}{p_{\text{success}}} \leq e^k$$

Satz 3.3

1. The full randomized algorithm (including repetitions) has a runtime of

$$\mathcal{O}(\lambda \cdot (2e)^k \cdot km)$$

where $\lambda > 1$ is a tunable parameter (confidence level).

2. If the algorithm answers **YES**, then the graph **does contain** a path of length $k-1$.
3. If the graph does contain a path of length $k-1$, then the probability that the algorithm answers **NO** is at most

$$e^{-\lambda}$$

Helper

Mathematical Tools and Notations

$[n] := \{1, 2, \ldots, n\}$

$[n]^k :=$ the set of sequences over $[n]$ of length k $\qquad |[n]^k| = n^k$

$\binom{[n]}{k} :=$ the set of k-element subsets of $[n]$ $\qquad \left|\binom{[n]}{k}\right| = \binom{n}{k}.$

The k nodes on a path of length $k-1$ can be colored using $[k]$ in exactly k^k ways

$k!$ of these colorings use each color exactly once

Helper

Mathematical Tools and Notations

Handshaking lemma : For all graphs , it holds that $\sum_{v \in V} \deg(v) = 2|E|$.

If you repeat an experiment with success probability p until success, then the expected number of trials is $\frac{1}{p}$ $(Geo(p))$

Helper

Mathematical Tools and Notations

For $c, n \in \mathbb{R}^+$, it holds that $c^{\log n} = n^{\log c}$

$$2^{\log n} = n^{\log 2} = n \text{ and } 2^{\mathcal{O}(\log n)} = n^{\mathcal{O}(1)} \text{ is always polynomial in } n$$

For $n \in \mathbb{N}_0$, it holds that $\sum_{i=0}^{n} \binom{n}{i} = 2^n$ (binomial theorem)

For $n \in \mathbb{N}_0$, it holds that $\frac{n!}{n^n} \geq e^{-n}$ (power series expansion of the exponential function)

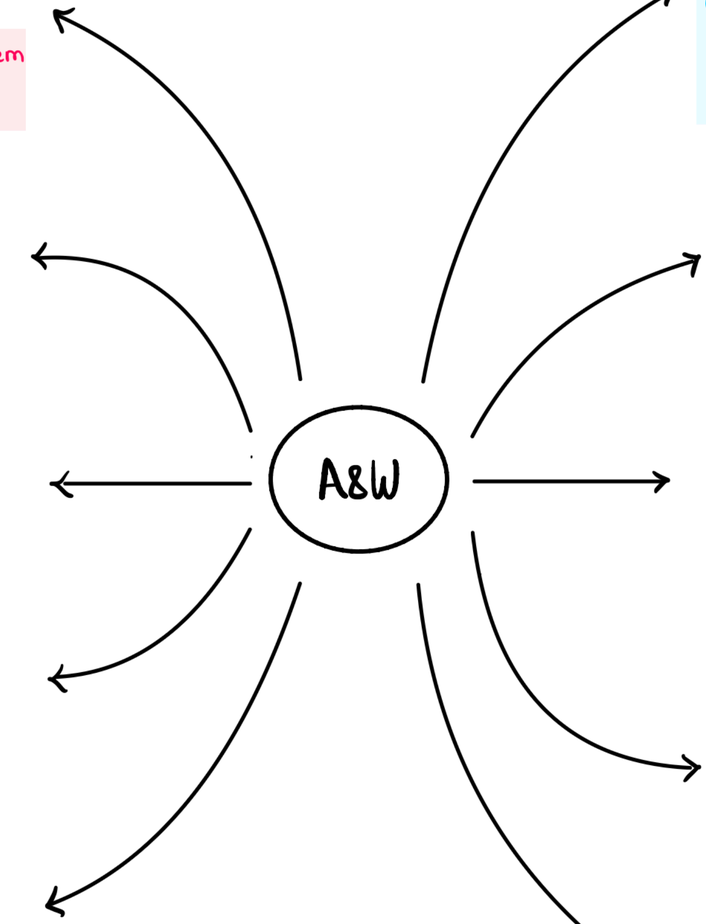
A&W

Exercise Session 11

Flow

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Last Weeks ...

- 08.05 : Randomized Algorithms II

- 15.05 : Flow

- 22.05 online : Minimum Cut , Convex Hull I (shortly remaining primality tests)

- 28.05 extra session : Convex Hull II , Smallest Enclosing Cycle

- 30.05 last extra session : Exam Prep Session + Pizza and Drinks

Outline

- Minitest 5
- Flow

Minitest 5

Flow

Flow

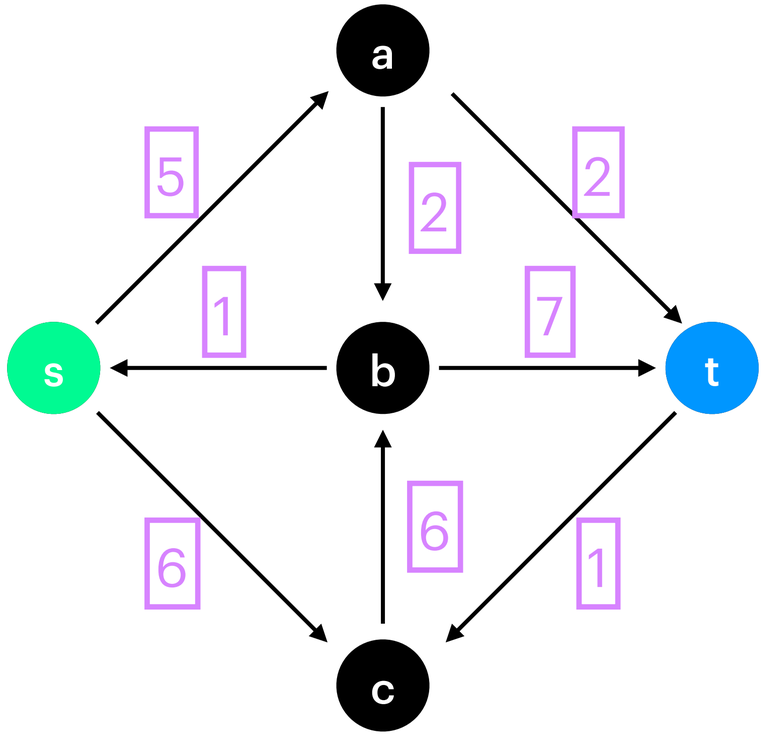
Definitions

- Network N :
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function

Flow

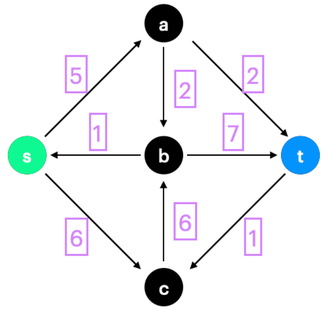
Definitions

- Network N :
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \leq f(e) \leq c(e)$ for all $e \in A$

Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

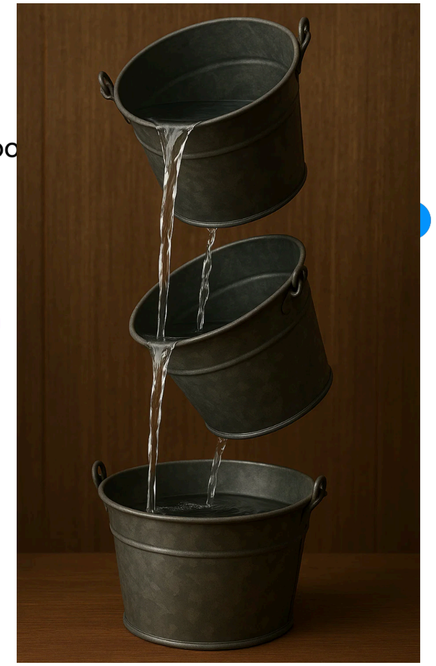
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loo
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \leq f(e) \leq c(e)$ for all $e \in A$

Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

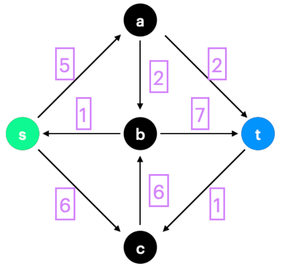
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

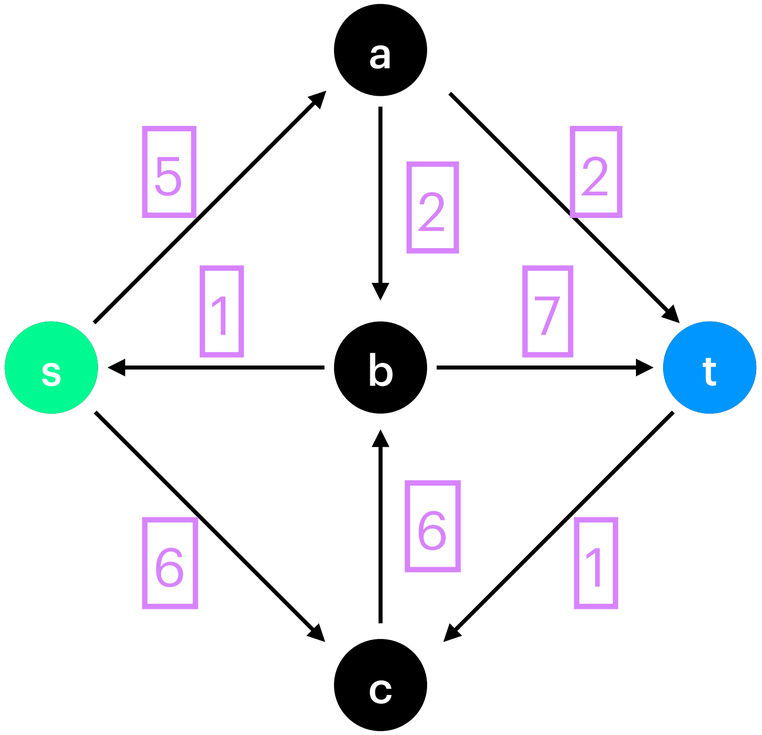
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

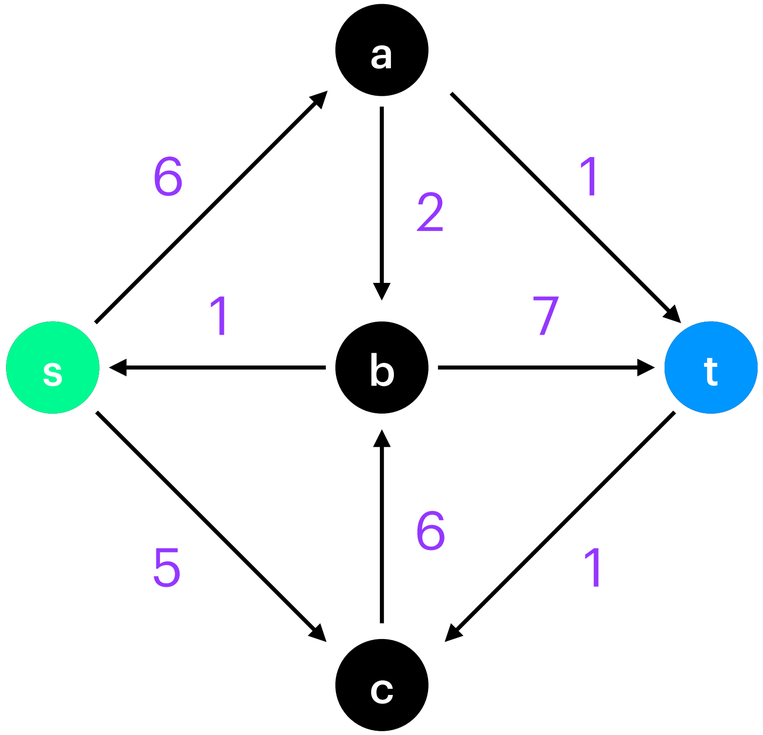
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

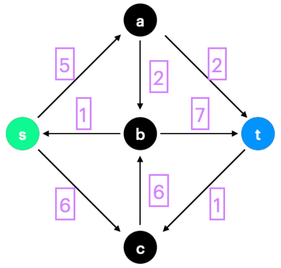
Is this a correct flow function ?





Flow
Definitions

- Network N :
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N : $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

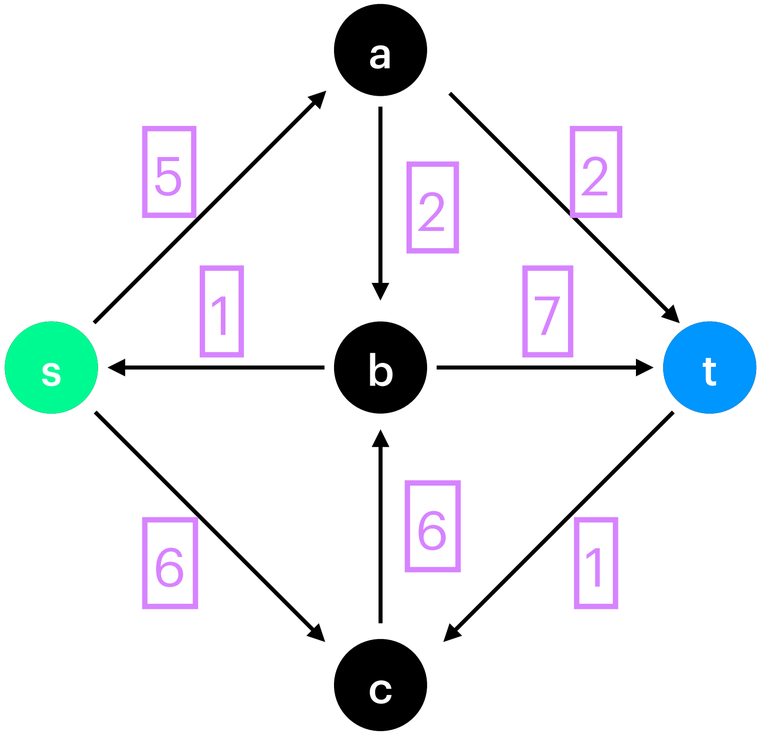
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

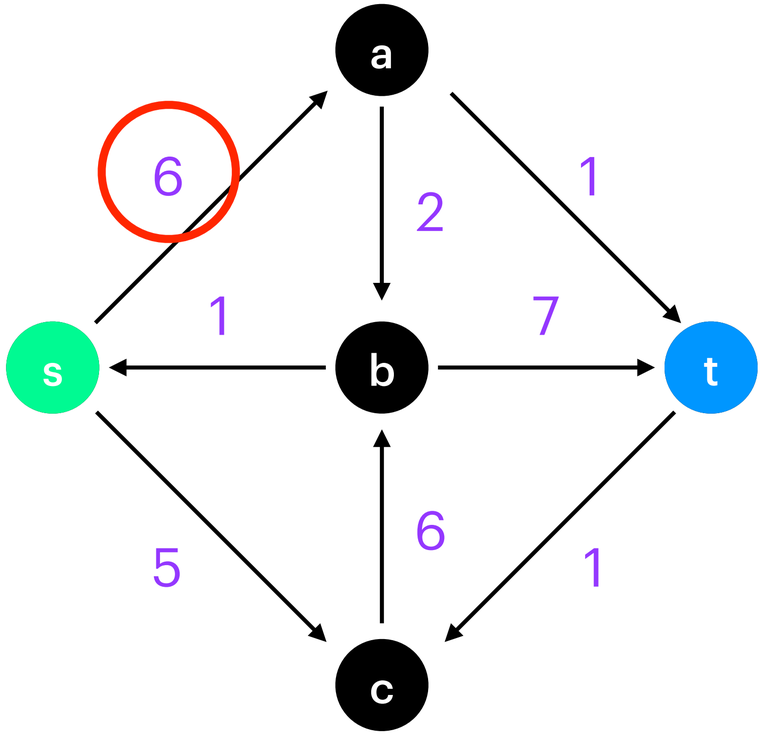
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

Is this a correct flow function ?





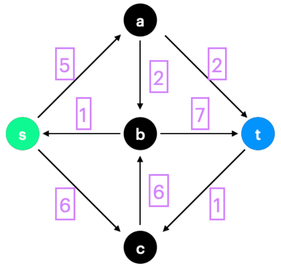
No !

Harms the capacity constraint

$0 \le f(e) \le c(e)$ for all $e \in A$

Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

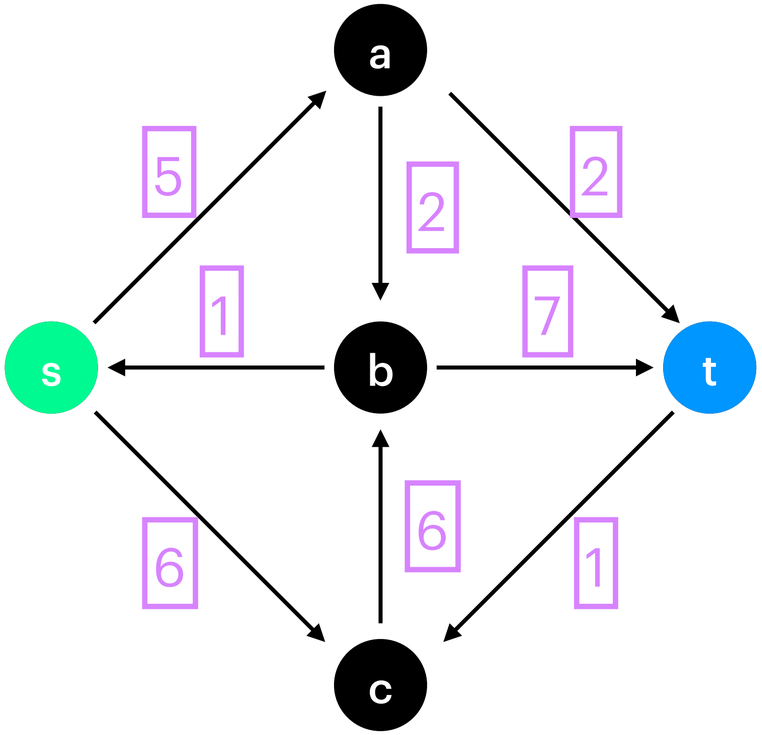
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

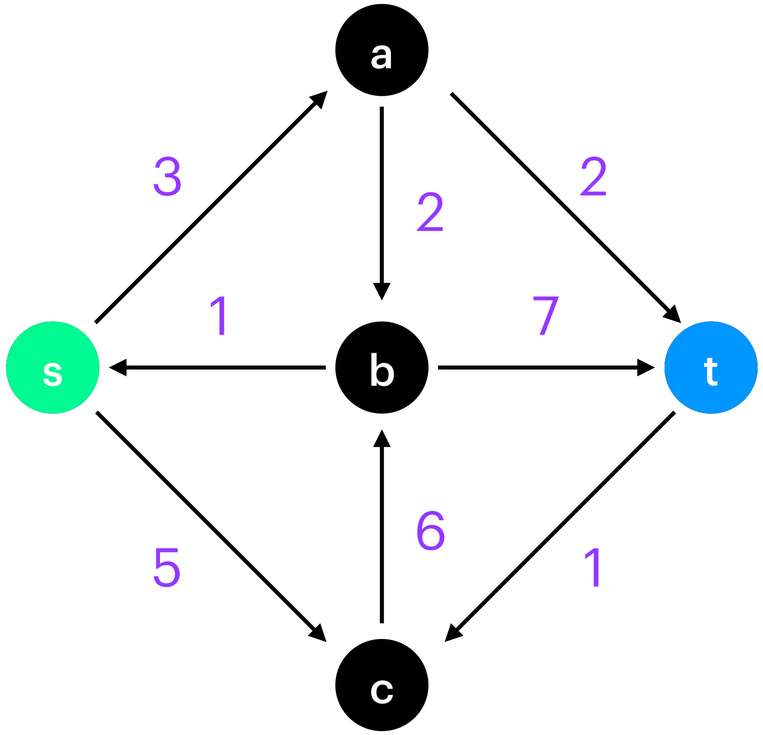
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

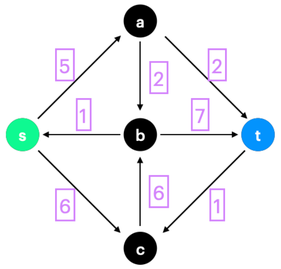
Is this a correct flow function ?





Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

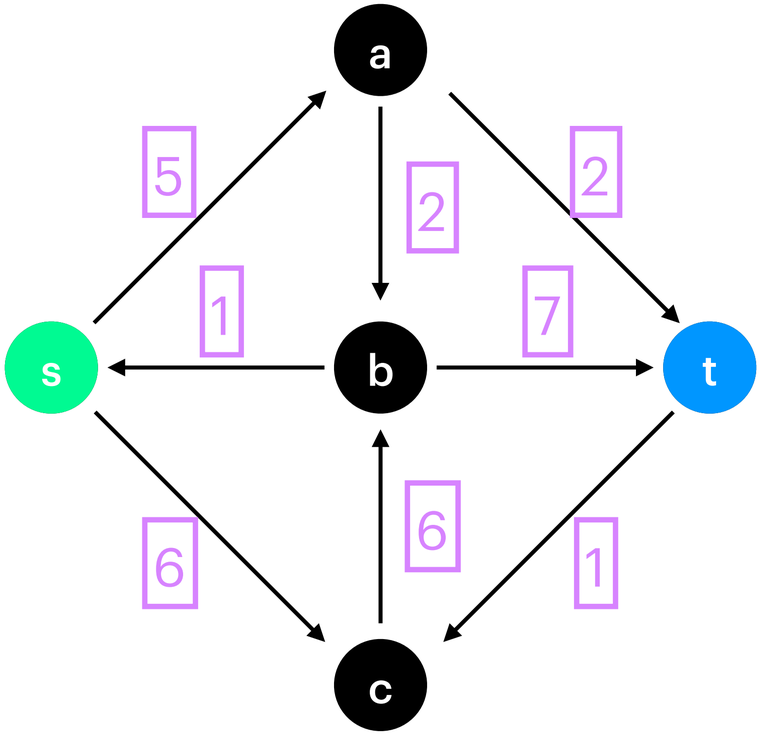
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

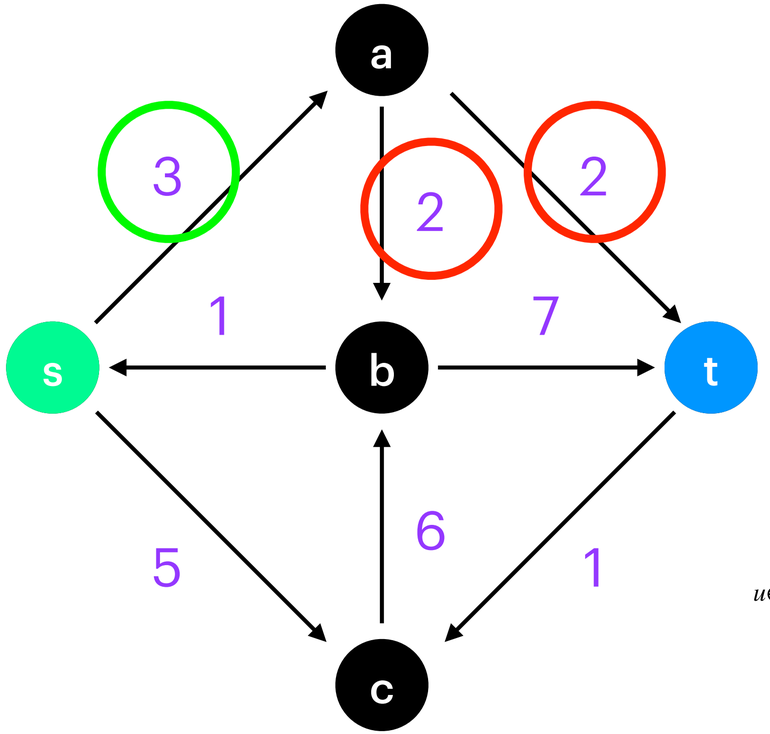
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

Is this a correct flow function ?





No !

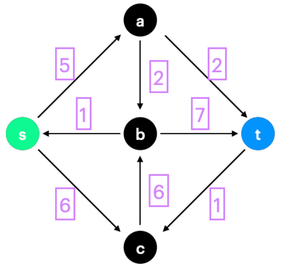
Harms the flow conservation

$\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

the total inflow equals total outflow

Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

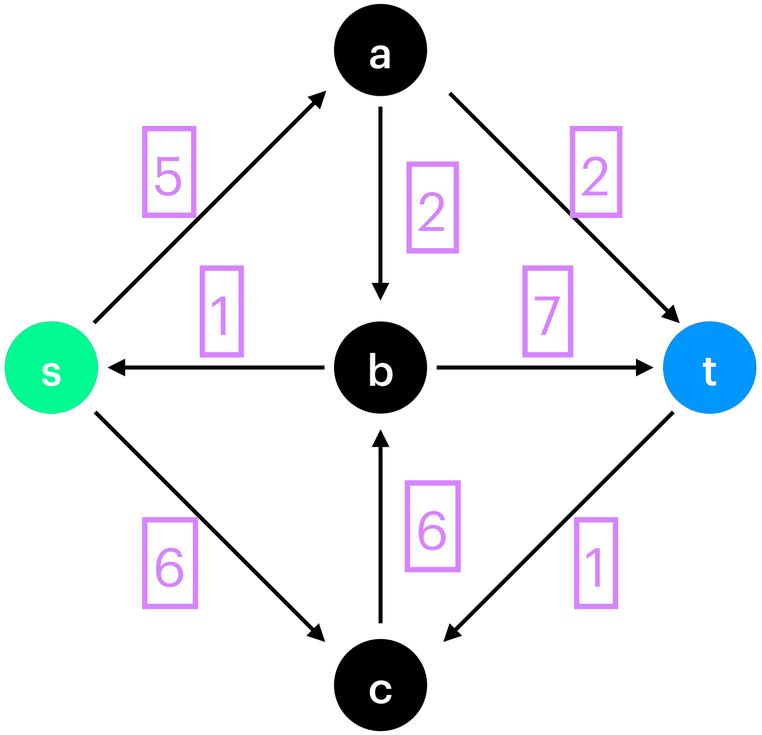
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

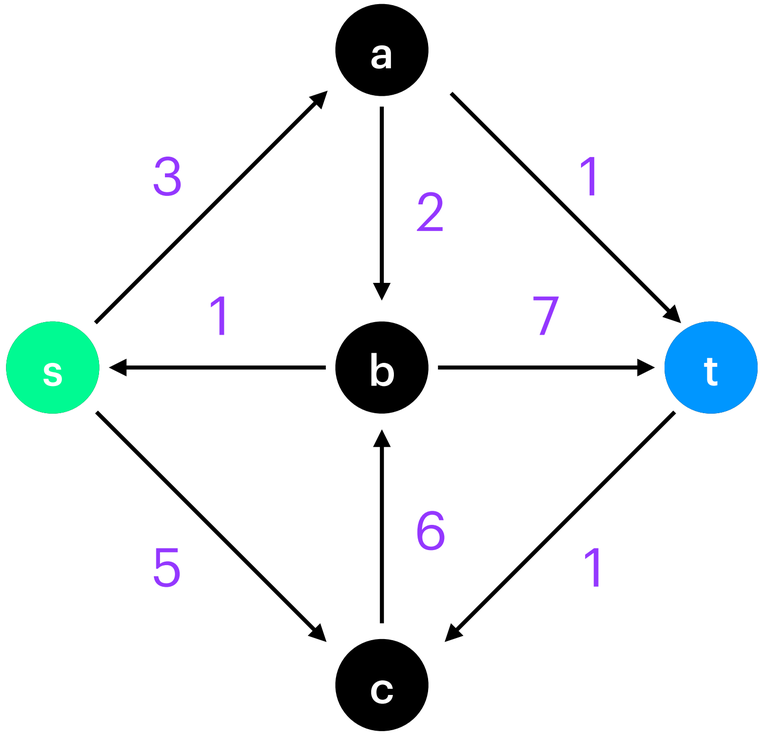
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

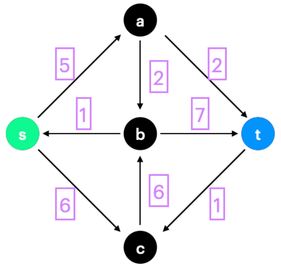
Is this a correct flow function ?





Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- Flow f in N: $f : A \to \mathbb{R}$

Capacity constraint : $0 \le f(e) \le c(e)$ for all $e \in A$

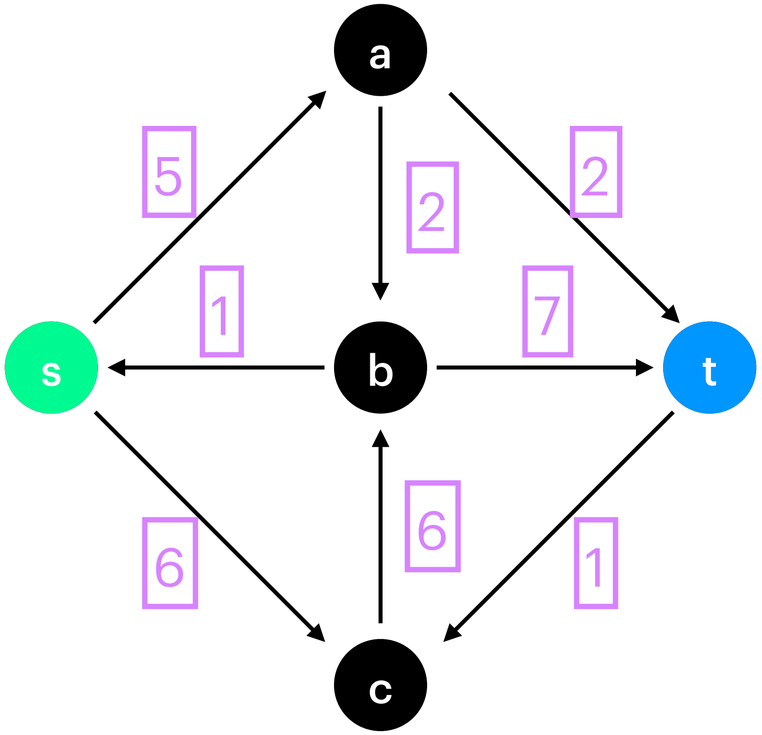
Flow conservation : $\sum_{u \in V:(u,v) \in A} f(u,v) = \sum_{u \in V:(v,u) \in A} f(v,u)$ for all $v \in V \setminus \{s,t\}$

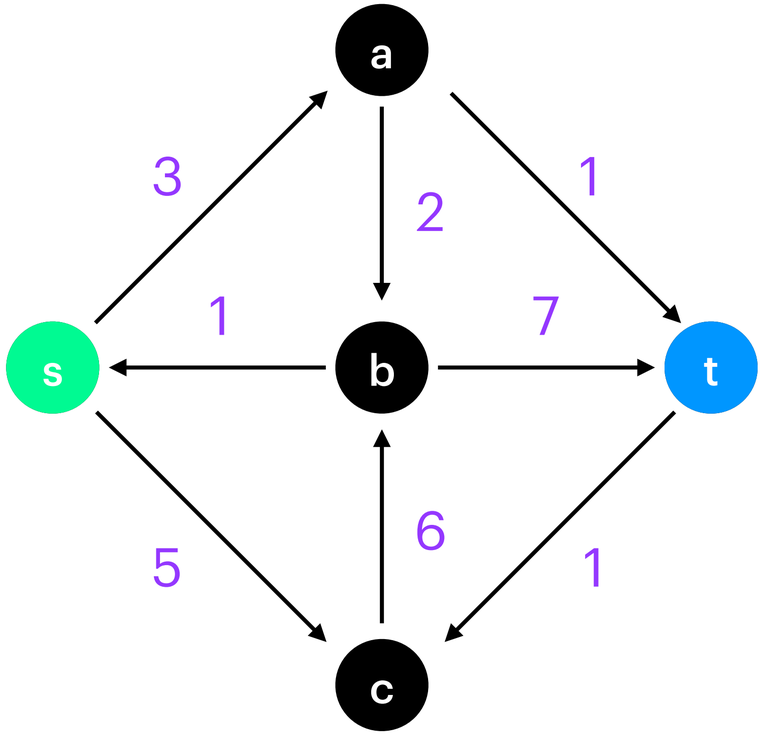
The value of a flow : $\text{val}(f) := \text{netoutflow}(s) := \sum_{u \in V:(s,u) \in A} f(s,u) - \sum_{u \in V:(u,s) \in A} f(u,s)$

what flows out of the source, must flow into the sink

$\text{val}(f) = \text{netinflow}(t) := \sum_{u \in V:(u,t) \in A} f(u,t) - \sum_{u \in V:(t,u) \in A} f(t,u)$

Is this a correct flow function ?





Yes !

What is val(f) ?

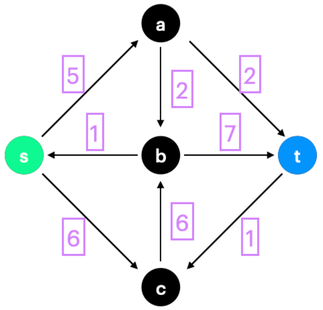
3 + 5 - 1 = 7

7 + 1 - 1 = 7

what flows out of the source, must flow into the sink

Flow
Definitions

- Network N:
  - $N = (V, A, c, s, t)$
  - (V, A) is a directed graph (without loops)
  - $s \in V$ is the source
  - $t \in V \setminus s$ is the sink
  - $c : A \to \mathbb{R}_0^+$ is the capacity function



- s-t-cut (S, T) for a network (V, A, c, s, t)
  - is a partition of V
  - (S, T) with $s \in S$ and $t \in T$

Partition (S, T): $S \cup T = V$ und $S \cap T = \emptyset$

- capacity of an s-t-cut (S, T)

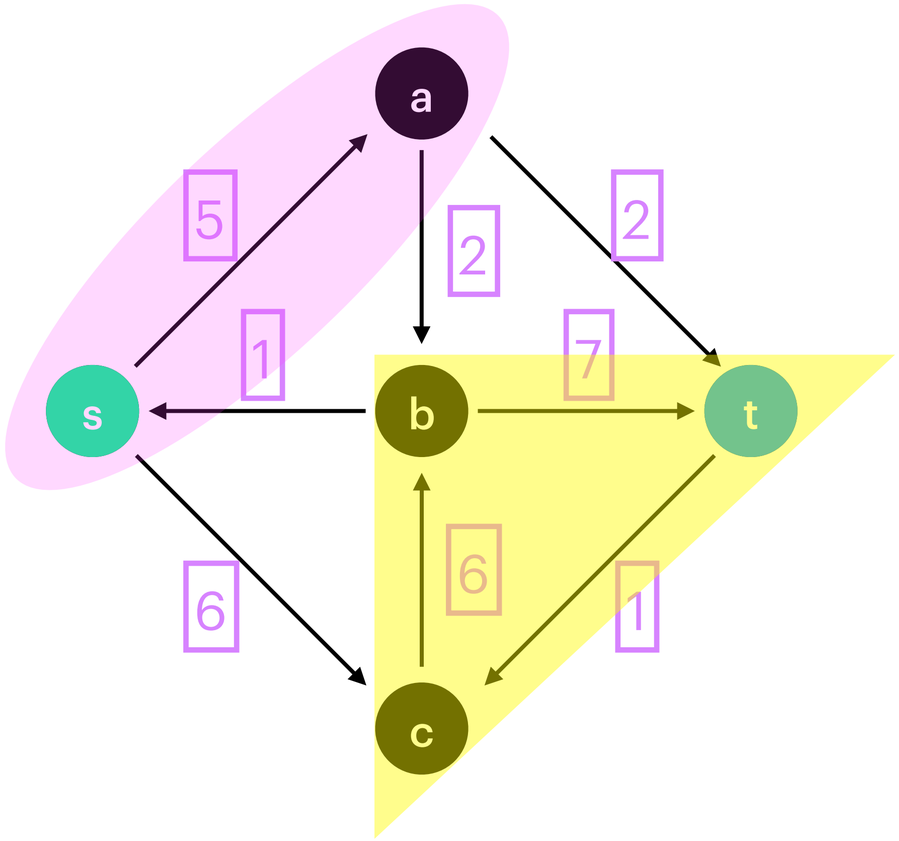
$$\text{cap}(S, T) := \sum_{(u,w) \in (S \times T) \cap A} c(u, w)$$

Flow
Definitions

- s-t-cut (S, T) for a network (V, A, c, s, t)
  - is a partition of V Partition (S, T) : $S \cup T = V$ und $S \cap T = \emptyset$
  - (S, T) with $s \in S$ and $t \in T$

- capacity of an s-t-cut (S, T)

$$\text{cap}(S, T) := \sum_{(u,w) \in (S \times T) \cap A} c(u, w)$$



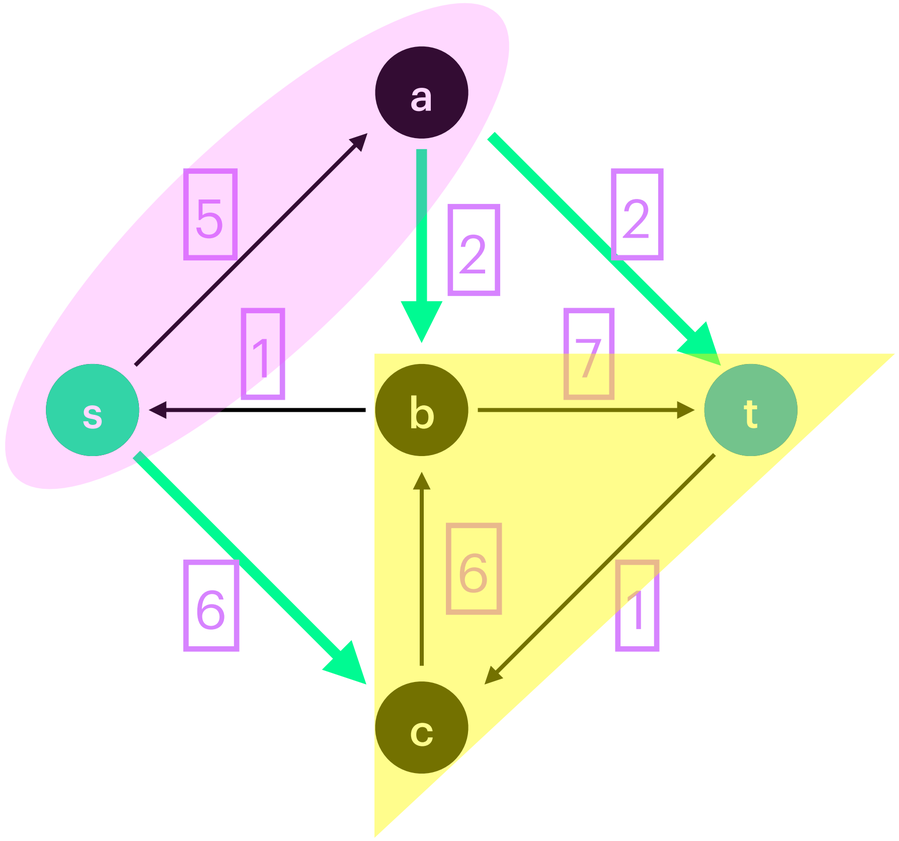
$$\text{cap}(S, T) = ?$$

Flow
Definitions

- s-t-cut (S, T) for a network (V, A, c, s, t)
  - is a partition of V Partition (S, T) : $S \cup T = V$ und $S \cap T = \emptyset$
  - (S, T) with $s \in S$ and $t \in T$

- capacity of an s-t-cut (S, T)

$$\text{cap}(S, T) := \sum_{(u,w) \in (S \times T) \cap A} c(u, w)$$



$$\text{cap}(S, T) = 6 + 2 + 2 = 10$$

Flow

Lemmas

If we can find an s-t-cut s.t. $\text{val}(f) = \text{cap}(S, T)$ then f is a maximum flow.

- Let f be a flow and (S, T) an s-t-cut in a network (V, A, c, s, t) :

$$\text{val}(f) \leq \text{cap}(S, T)$$

A flow can never exceed the capacity of an s-t-cut

- Maxflow-Mincut Theorem :

Every network satisfies

$$\max_{f} \text{val}(f) = \min_{(S,T)\ \text{s-t-cut}} \text{cap}(S, T)$$

Flow

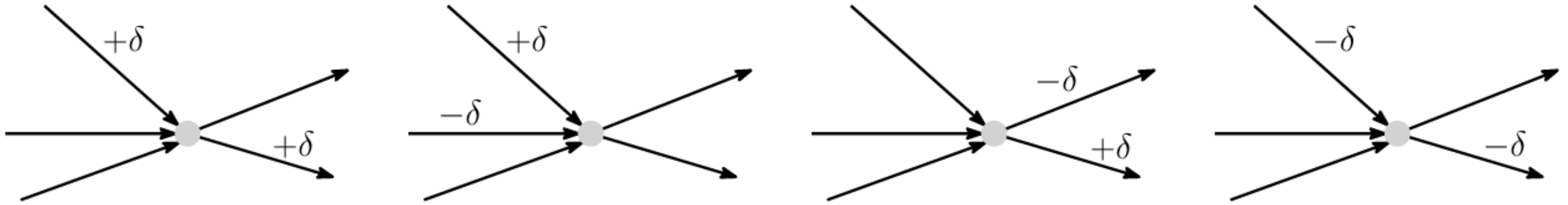
Problem Description

given : A network $N = (V, A, c, s, t)$

to find : A flow of maximum value

Flow

Local changes to the flow that preserve flow conservation

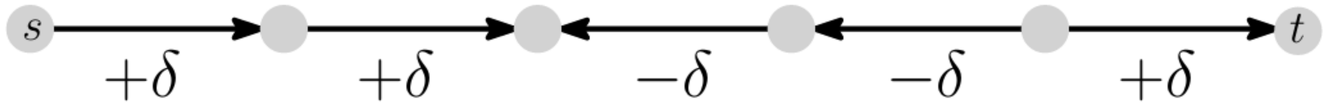


When increasing : watch out for the capacity

When decreasing : watch out for the previous flow

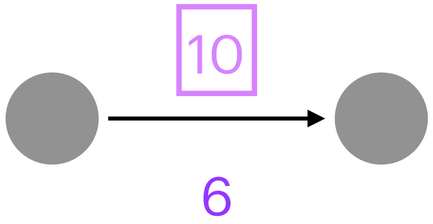
the total inflow equals
total outflow

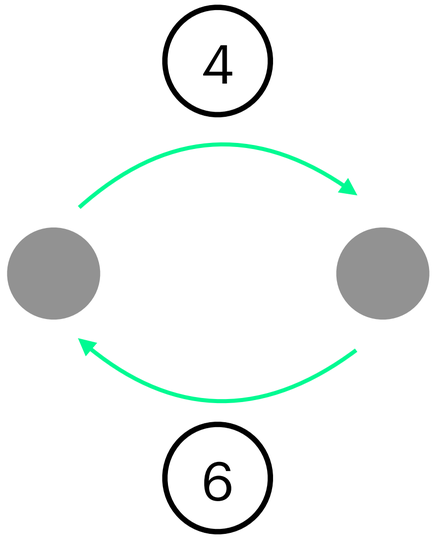
undirected augmenting path :

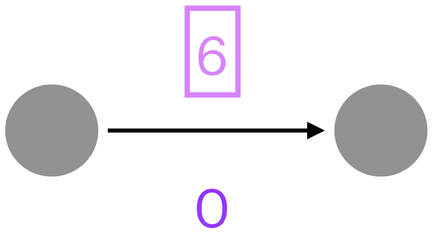


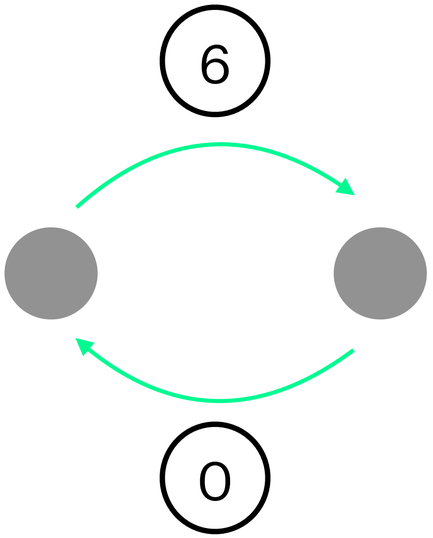
Flow

Local changes to the flow that preserve flow conservation



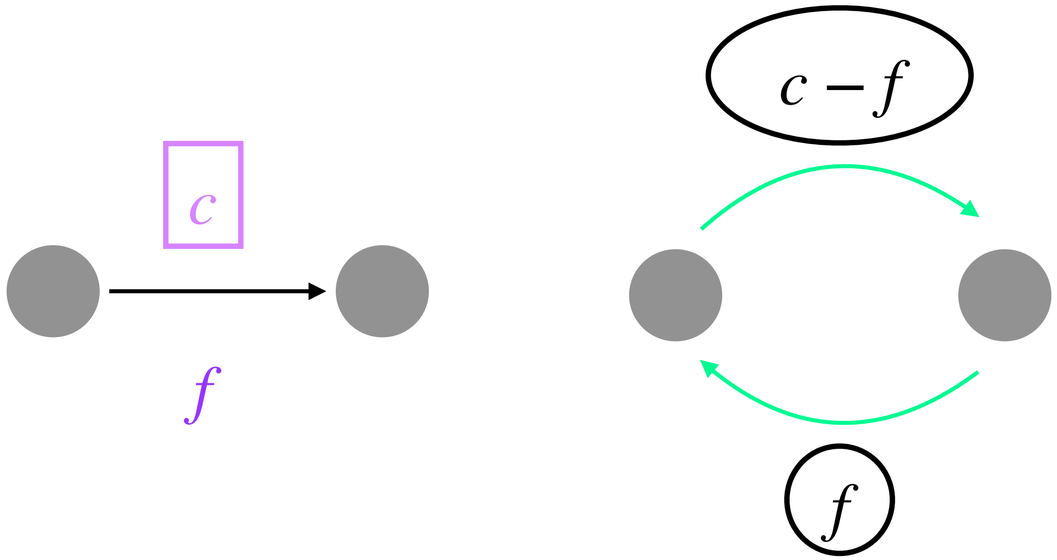






Flow

Local changes to the flow that preserve flow conservation



Flow

Residual Network

- Residual Network N_f :
  - Let $N = (V, A, c, s, t)$ be a network , f flow for N
  - $N_f = (V, A_f, r_f, s, t)$

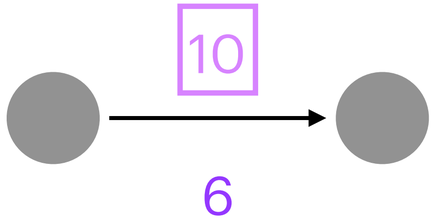
    r_f “residual capacity”
    - $e \in A$, $f(e) < c(e)$: put e in A_f , $r_f(e) := c(e) - f(e)$
    - $e \in A$, $f(e) > 0$: put e^{opp} in A_f , $r_f(e^{opp}) := f(e)$

Flow

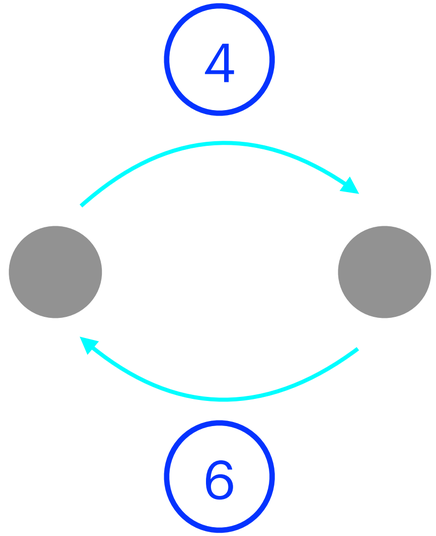
Residual Network examples

- Residual Network N_f:
  - Let $N = (V, A, c, s, t)$ be a network, f flow for N
  - $N_f = (V, A_f, r_f, s, t)$ r_f "residual capacity"
    - $e \in A$, $f(e) < c(e)$: put e in A_f, $r_f(e) := c(e) - f(e)$
    - $e \in A$, $f(e) > 0$: put e^{opp} in A_f, $r_f(e^{opp}) := f(e)$

in Network



in Residual Network

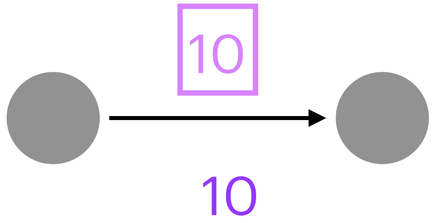


Flow

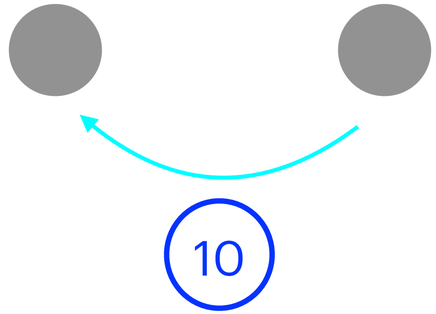
Residual Network examples

- Residual Network N_f:
  - Let $N = (V, A, c, s, t)$ be a network, f flow for N
  - $N_f = (V, A_f, r_f, s, t)$ r_f "residual capacity"
    - $e \in A$, $f(e) < c(e)$: put e in A_f, $r_f(e) := c(e) - f(e)$
    - $e \in A$, $f(e) > 0$: put e^{opp} in A_f, $r_f(e^{opp}) := f(e)$

in Network



in Residual Network

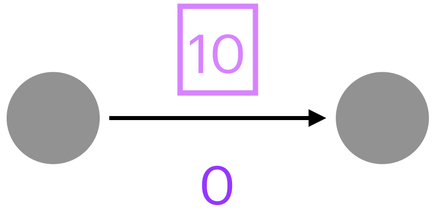


Flow

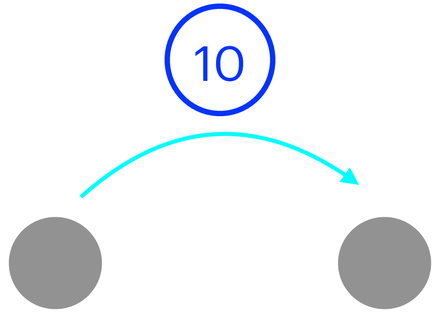
Residual Network examples

- Residual Network N_f:
  - Let $N = (V, A, c, s, t)$ be a network, f flow for N
  - $N_f = (V, A_f, r_f, s, t)$ r_f "residual capacity"
    - $e \in A$, $f(e) < c(e)$: put e in A_f, $r_f(e) := c(e) - f(e)$
    - $e \in A$, $f(e) > 0$: put e^{opp} in A_f, $r_f(e^{opp}) := f(e)$

in Network



in Residual Network



Flow

Remember

If we can find an s-t-cut s.t. $\text{val}(f) = \text{cap}(S, T)$ then f is a maximum flow.

- Let f be a flow and (S, T) an s-t-cut in a network (V, A, c, s, t) :

$$\text{val}(f) \leq \text{cap}(S, T)$$

A flow can never exceed the capacity of an s-t-cut

- Maxflow-Mincut Theorem :

Every network satisfies

$$\max_{f} \text{val}(f) = \min_{(S,T)\ \text{s-t-cut}} \text{cap}(S, T)$$

Flow

Theorem

Let N be a network without reverse edges

A flow f is a maximum flow

$\Longleftrightarrow$

There exists no **directed** s-t-path in the residual network N_f

Flow

Ford-Fulkerson Algorithm

1 : $f \leftarrow 0$

2 : while $\exists$ s-t-path P in N_f do

3 : Augment the flow along P

4 : return f

No termination guarantees in $\mathbb{R}$

Augmenting along P :

Take a look at all the residual capacities on this path

Take the minimum $r_{f,min}$

Increase the flow on this path with the minimum $r_{f,min}$

integer capacities , no reverse edges :

There exists an integer maximum flow and it can be computed in O(mnU)

U is the maximum capacity

https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

Flow

Ford-Fulkerson Algorithm

1) Start with initial flow as 0

2) While there exists an augmenting path from the source to the sink:

Find an augmenting path using any path-finding algorithm, such as breadth-first search or depth-first search.

Determine the amount of flow that can be sent along the augmenting path, which is the minimum residual capacity along the edges of the path.

Increase the flow along the augmenting path by the determined amount.

3) Return the maximum flow.

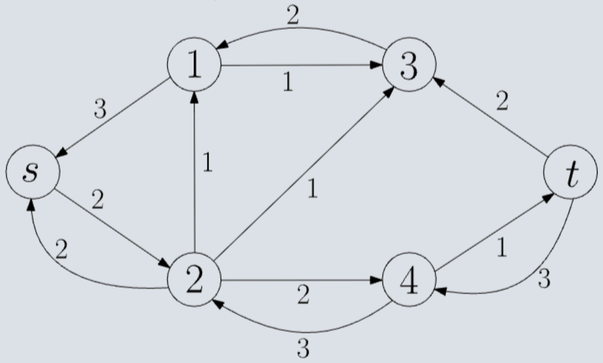
https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/

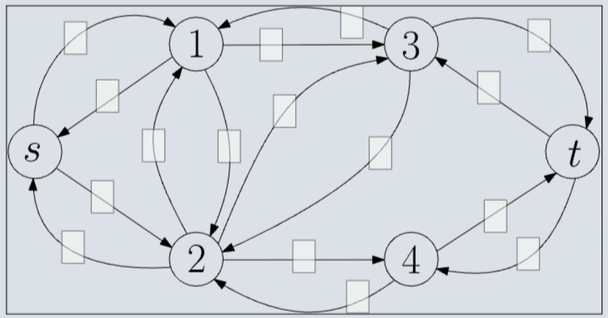
Let’s take a break

Flow

Example question

Sei N ein Netzwerk ohne entgegengesetzte Kanten. Betrachten Sie das abgebildete Restnetzwerk R_f. Berechnen Sie den zugehörigen Fluss f und ziehen Sie die Flusswerte auf die entsprechenden Kanten (verwenden Sie die 0 für Kanten, über die kein Fluss fliesst)





1 2 3 4 5 6 7 8 0

Flow

Applications Maximum Bipartite Matching Problem

given : A bipartite graph G (unweighted, undirected)

to find : Find a cardinality-maximum Matching

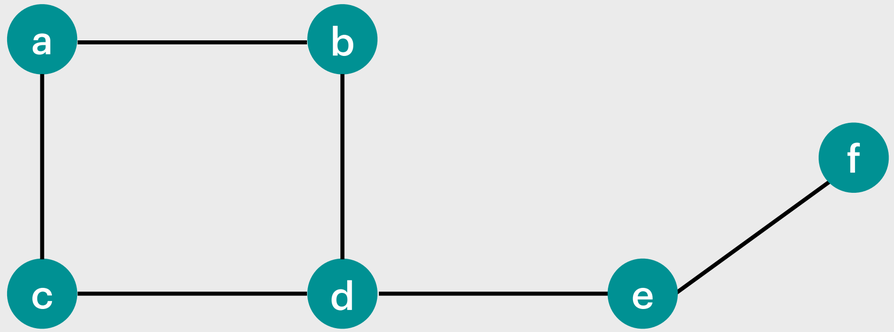
Matching

Recap

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



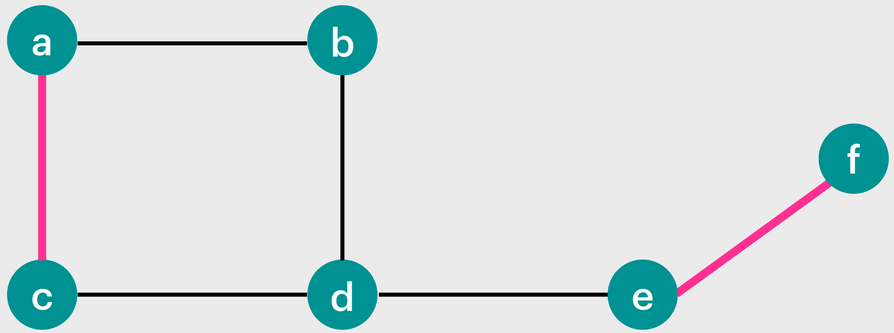
Matching

Recap

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



M_1 = {{a,c} , {e,f}}

✅

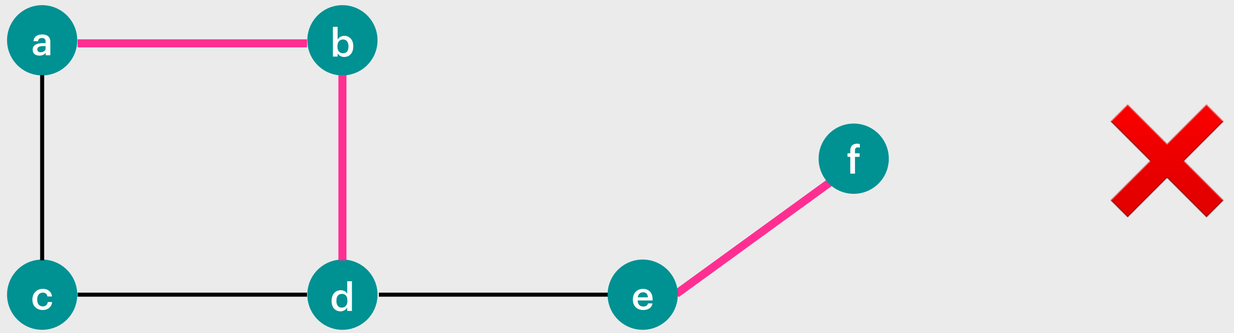
Matching

Recap

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



Matching

Recap

- inclusion-maximal : “no edge can be added to this matching”
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : “one can’t find a bigger matching”
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

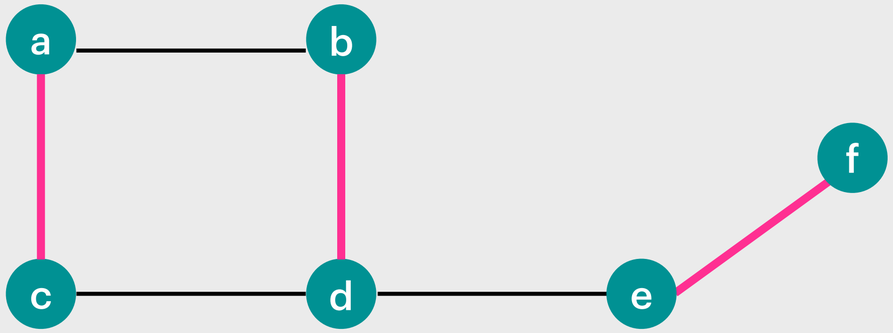
Matching

Recap

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ?

✅



Is this inclusion-maximal ?

✅

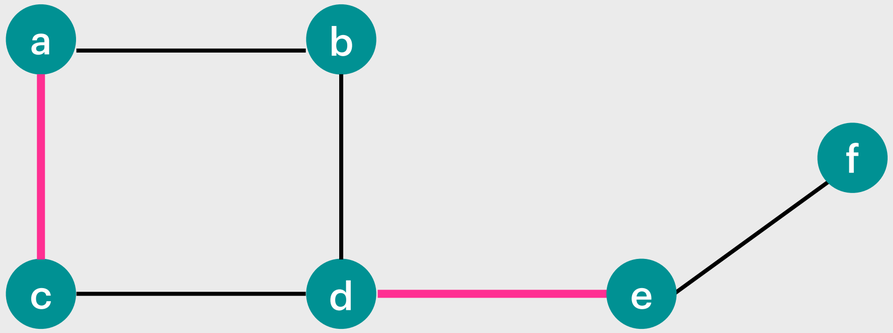
Matching

Recap

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ?

❌



Is this inclusion-maximal ?

✅

Flow

Applications Maximum Bipartite Matching Problem

Graph to Newtork : Build a Network N_G

$$\underbrace{G = (U \uplus W, E)}_{\text{bipartite G}} \quad \mapsto \quad \underbrace{N_G = \left(\underbrace{U \uplus W \uplus \{s, t\}}_{\text{Vertex Set}}, A, c, s, t \right)}_{\text{Network}}$$

add s and t , additional vertices s.t. $s \neq t$

$$A = (\{s\} \times U) \ \cup \ \{(u, w) \in U \times W \mid \{u, w\} \in E\} \ \cup \ (W \times \{t\})$$

$$c \equiv 1$$

Flow

Applications Maximum Bipartite Matching Problem

given : A bipartite graph $G = (U \cup W, E)$ (unweighted, undirected)

to find : Find a cardinality-maximum Matching

Algorithm :

Build a Network N_G

$$\underbrace{G = (U \uplus W, E)}_{\text{bipartite G}} \quad \mapsto \quad \underbrace{N_G = \left(\underbrace{U \uplus W \uplus \{s,t\}}_{\text{Vertex Set}}, A, c, s, t \right)}_{\text{Network}}$$

add s and t , additional vertices s.t. $s \neq t$

$$A = (\{s\} \times U) \ \cup \ \{(u,w) \in U \times W \mid \{u,w\} \in E\} \ \cup \ (W \times \{t\}) \qquad c \equiv 1$$

$$\max_{M \text{ Matching in } G} |M| = \max_{f \text{ Flow in } N_G} \text{val}(f)$$

Flow

Applications Edge-disjoint paths problem

given : A graph G with two vertices u, v s.t. $u \neq v$.

to find : Maximum number of edge-disjoint $u - v$ paths

Connectivity

Menger's Theorem

G is k-edge connected $\iff$ For $\forall u, v \in V, u \neq v$ there exists k edge-disjoint u-v paths

Flow

Applications Edge-disjoint paths problem

Graph to Newtork :

$$\underbrace{G = (V, E),\ u, v \in V}_{\text{Graph with 2 vertices}} \quad \Rightarrow \quad \underbrace{N_G^* = (V, A, c, u, v)}_{\text{Network}}$$

$$A := \{(x, y), (y, x) \mid \{x, y\} \in E\}$$

$$c \equiv 1$$

Flow

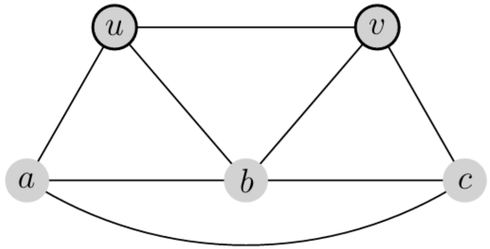
Applications Edge-disjoint paths problem

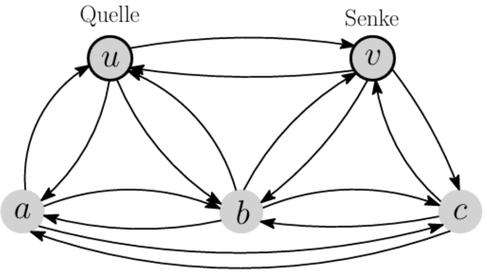
Graph to Newtork :

$$\underbrace{G = (V, E),\ u, v \in V}_{\text{Graph with 2 vertices}} \quad \Rightarrow \quad \underbrace{N_G^* = (V, A, c, u, v)}_{\text{Network}}$$

$$A := \{(x, y), (y, x) \mid \{x, y\} \in E\}$$

$$c \equiv 1$$





Flow

Applications Edge-disjoint paths problem

given : A graph G with two vertices u, v s.t. $u \neq v$.

to find : Maximum number of edge-disjoint $u - v$ paths

Algorithm :

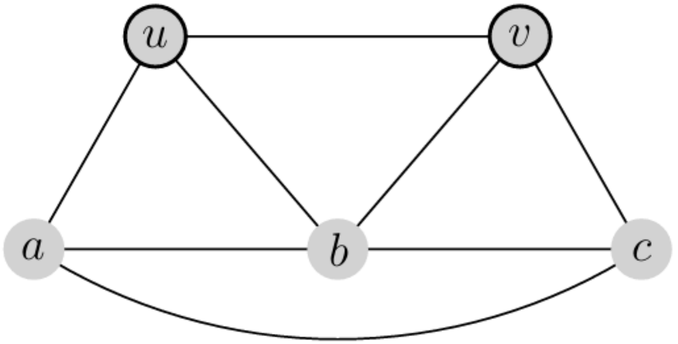
Build a Network N_G $\underbrace{G = (V, E),\ u, v \in V}_{\text{Graph with 2 vertices}} \Rightarrow \underbrace{N_G^* = (V, A, c, u, v)}_{\text{Network}}$

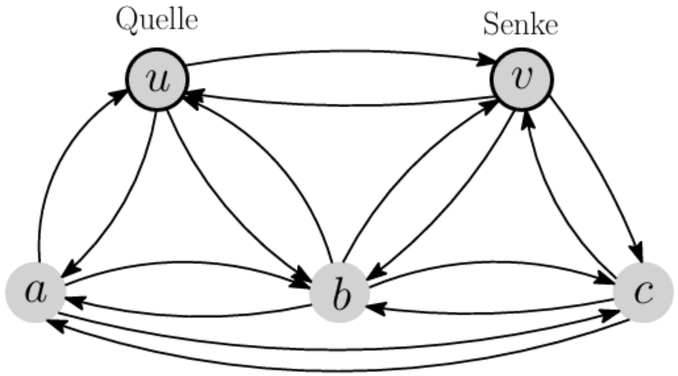
$$A := \{(x, y), (y, x) \mid \{x, y\} \in E\} \quad c \equiv 1$$

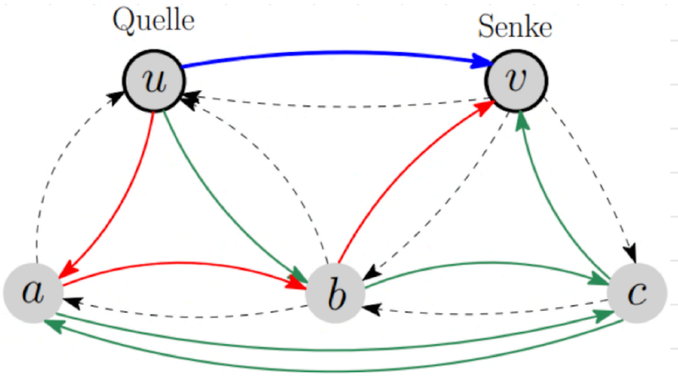
$$\#\text{ edge disjoint u-v paths } = \max_{f \text{ Flow in } N_G} \text{val}(f)$$

Flow

Applications Edge-disjoint paths problem







Flow

Applications Image Segmentation

given : An image modeled as a graph $G = (P, E)$

color information $\chi : P \rightarrow$ colors

to find : seperate foreground from background

color information $\chi : P \rightarrow$ colors

per-pixel estimates $\alpha : P \rightarrow \mathbb{R}_0^+$ $\beta : P \rightarrow \mathbb{R}_0^+$ $\gamma : E \rightarrow \mathbb{R}_0^+$

α_p is bigger $\Longrightarrow$ more likely foreground

β_p is bigger $\Longrightarrow$ more likely background

γ_e is bigger $\Longrightarrow$ more likely belong to the same region

Flow

Applications Image Segmentation

color information $\chi : P \to$ colors

per-pixel estimates $\alpha : P \to \mathbb{R}_0^+$ $\beta : P \to \mathbb{R}_0^+$ $\gamma : E \to \mathbb{R}_0^+$

α_p is bigger $\Longrightarrow$ more likely foreground

β_p is bigger $\Longrightarrow$ more likely background

γ_e is bigger $\Longrightarrow$ more likely belong to the same region

Quality function : $q(A,B) := \sum_{p \in A} \alpha_p + \sum_{p \in B} \beta_p - \sum_{\substack{e \in E, \\ |e \cap A| = 1}} \gamma_e$ maximize

$q'(A,B) := \sum_{p \in A} \beta_p + \sum_{p \in B} \alpha_p + \sum_{\substack{e \in E \\ |e \cap A| = 1}} \gamma_e$ minimize

Flow

Applications Image Segmentation

Build a Network N_G

$$N_G := \left(P \cup \{s, t\}, \overrightarrow{E}, c, s, t\right)$$

add s and t , additional vertices s.t. $s \neq t$

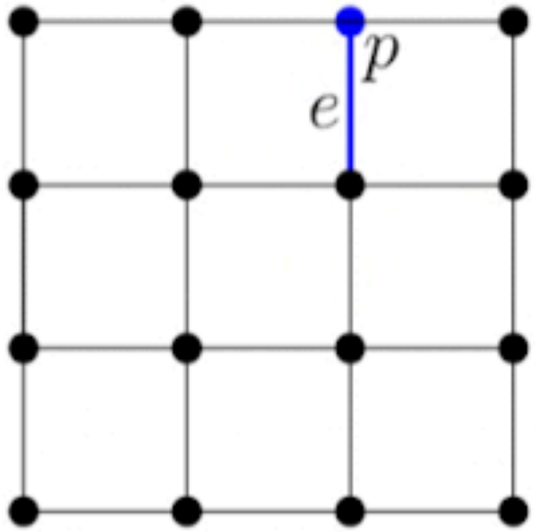
s has a directed edge to each pixel $p \in P$ with capacity α_p

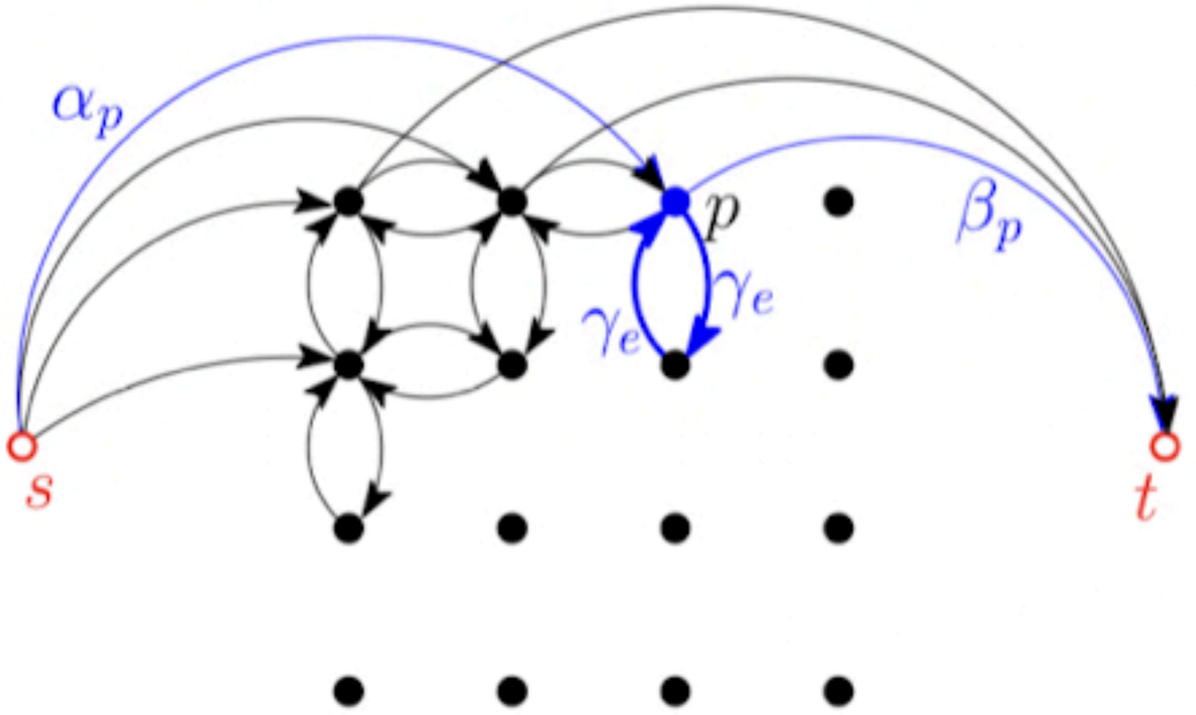
each pixel $p \in P$ has a directed edge to t with capacity β_p

for each edge $e = \{p, p'\} \in E$ there are two directed edges (p, p') and (p', p) with capactiy γ_e

Flow

Applications Image Segmentation





Flow

Applications Image Segmentation

given : An image modeled as a graph $G = (P, E)$

color information

$\chi : P \to$ colors

to find : seperate foreground from background

Algorithm :

Build a Network N_G

$$N_G := \left(P \cup \{s, t\}, \overrightarrow{E}, c, s, t\right)$$

add s and t , additional vertices s.t. $s \neq t$

s has a directed edge to each pixel $p \in P$ with capacity α_p

each pixel $p \in P$ has a directed edge to t with capacity β_p

for each edge $e = \{p, p'\} \in E$ there are two directed edges (p, p') and (p', p) with capactiy γ_e

$$q'(A, B) = \max_{f \text{ Flow in } N_G} \text{val}(f)$$

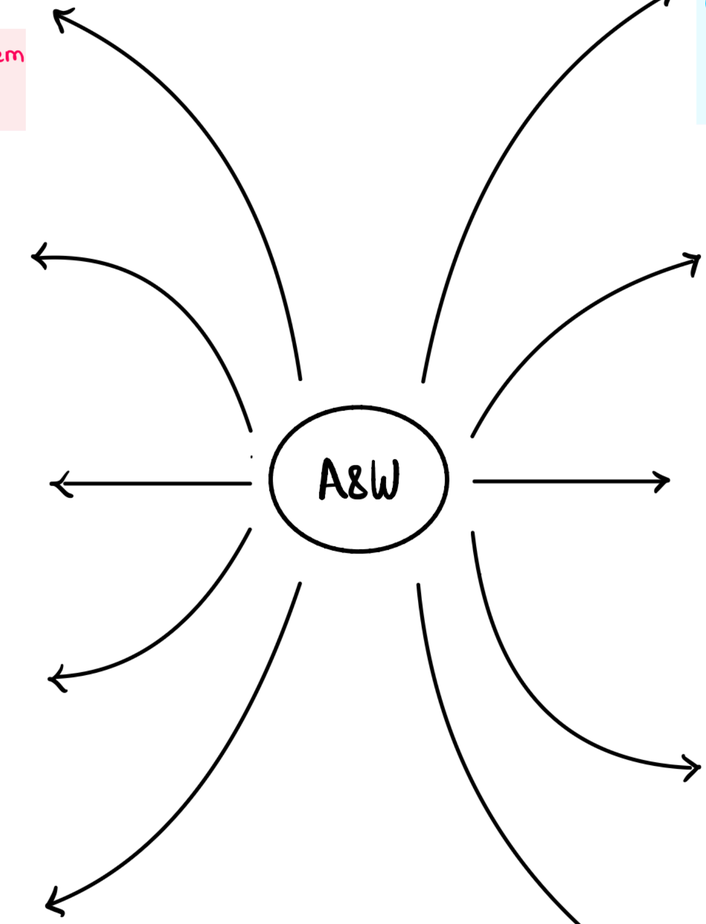
A&W

Exercise Session 12

Minimum Cut, Smallest Enclosing Cycle

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Last Weeks ...

- 08.05 : Randomized Algorithms II

- 15.05 : Flow

- 23.05 online : Minimum Cut , Smallest Enclosing Circle

- 28.05 extra session : Exam Prep Session + Pizza and Drinks

- 30.05 last extra session : Convex Hull (shortly remaining primality tests)

Outline

- Minimum Cut
- Smallest Enclosing Circle

Minimum Cut

Min-Cut

Definitions

- Multigraph :
  - undirected, unweighted, without self-loops
  - possibly with multiple edges between the same pair of nodes

Min-Cut

Definitions

- Multigraph :
  - undirected, unweighted, without self-loops
  - possibly with multiple edges between the same pair of nodes

- Edge Cut C :
  - A set of edges C s.t. $G' = (V, E \backslash C)$ is a disconnected graph.

Min-Cut

Definitions

- Multigraph :
  - undirected, unweighted, without self-loops
  - possibly with multiple edges between the same pair of nodes
- Edge Cut C :
  - A set of edges C s.t. $G' = (V, E \backslash C)$ is a disconnected graph.
- $\mu(G)$:
  - the cardinality of the smallest possible edge cut in graph G.

$$\mu(G) := \min_{C \subseteq E,\ (V, E \backslash C) \text{ disconnected}} |C|$$

Min-Cut

Problem Description

given : A multigraph G

to find : $\mu(G)$

- Multigraph :
  - undirected, unweighted, without self-loops
  - possibly with multiple edges between the same pair of nodes
- Edge Cut C :
  - A set of edges C s.t. $G' = (V, E \backslash C)$ is a disconnected graph.
- $\mu(G)$:
  - the cardinality of the smallest possible edge cut in graph G.

$$\mu(G) := \min_{\substack{C \subseteq E, \\ (V, E \backslash C) \text{ disconnected}}} |C|$$

Min-Cut

Problem Description

given : A multigraph G

to find : $\mu(G)$

Examples :

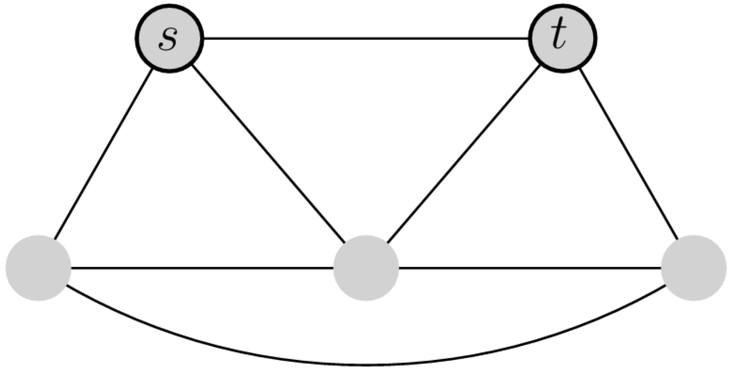
- Multigraph :
  - undirected, unweighted, without self-loops
  - possibly with multiple edges between the same pair of nodes
- Edge Cut C :
  - A set of edges C s.t. $G' = (V, E \backslash C)$ is a disconnected graph.
- $\mu(G)$:
  - the cardinality of the smallest possible edge cut in graph G.

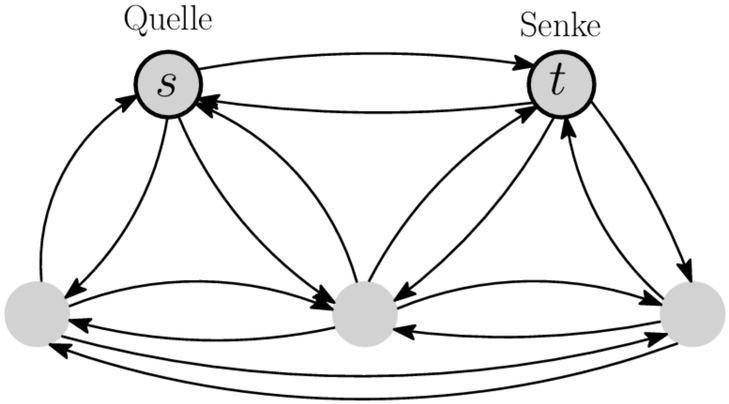
$$\mu(G) := \min_{C \subseteq E,\ (V, E\backslash C) \text{ disconnected}} |C|$$

Min-Cut

First Known Solution

minimum s-t cut in O(mn logn)





- Fix a source node s
- Then consider $t \in V \setminus \{s\}$, for each t compute the minimum s-t cut
- The global min-cut is the smallest of these s-t cuts

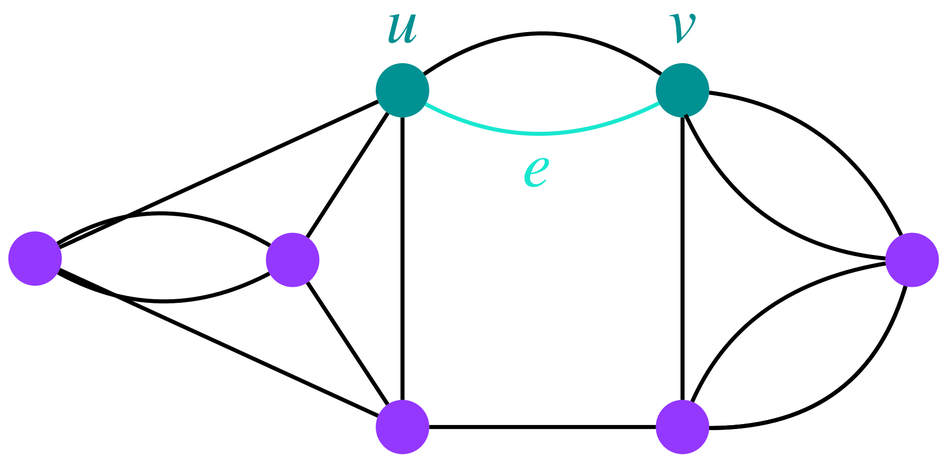
total runtime : O((n-1)mn logn) = O(n^4 logn)

Min-Cut

Edge Contraction of e

$e = \{u, v\} \in E$

G

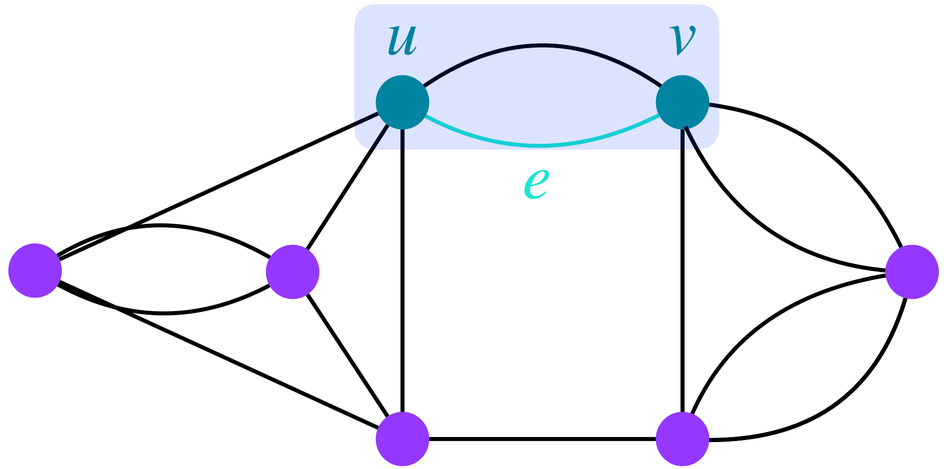


Min-Cut

Edge Contraction of e

$e = \{u, v\} \in E$

G



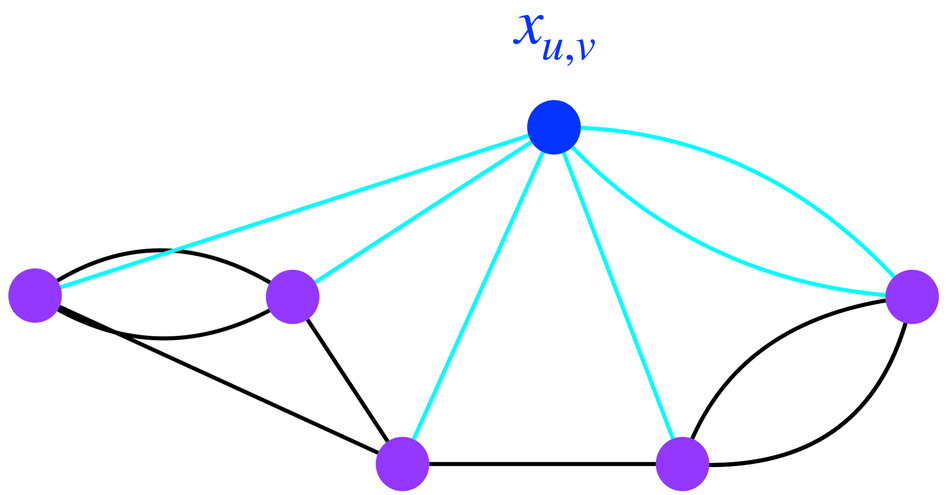
For $w, w' \in V(G) \backslash \{u, v\}$:

$$\{w, w'\} \mapsto \{w, w'\}, \quad \{w, u\} \mapsto \{w, x_{u,v}\}, \quad \{w, v\} \mapsto \{w, x_{u,v}\}$$

Min-Cut

Edge Contraction of e

$e = \{u, v\} \in E$



G / e

For $w, w' \in V(G) \backslash \{u, v\}$:

$$\{w, w'\} \mapsto \{w, w'\}, \quad \{w, u\} \mapsto \{w, x_{u,v}\}, \quad \{w, v\} \mapsto \{w, x_{u,v}\}$$

Min-Cut

- $\mu(G)$:
  - the cardinality of the smallest possible edge cut in graph G.

Lemma

Let $G = (V, E)$ be a multigraph, $e \in E$

$$\mu(G \backslash e) \geq \mu(G)$$

If G has a minimum cut C s.t. $e \notin C$

$$\mu(G \backslash e) = \mu(G)$$

Find e whose contraction preserves μ

The minimum cut value μ can never decrease when contracting an edge

μ stays unchanged if there exists a minimum cut that doesn't contain the edge being contracted

Min-Cut

Cut(G)

1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $e \leftarrow$ uniformly random edge in G'

4 : $G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'

Runtime : $O(n^2)$

Min-Cut

Cut(G)

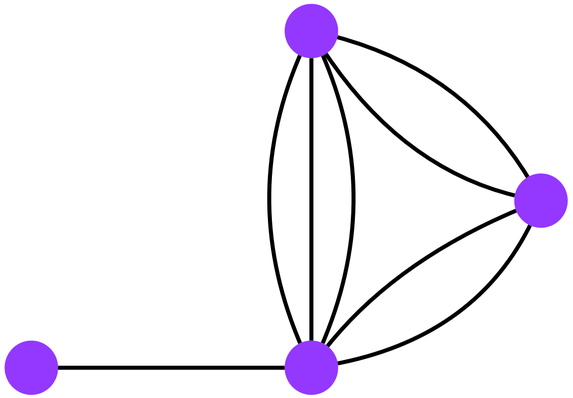
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



Min-Cut

Cut(G)

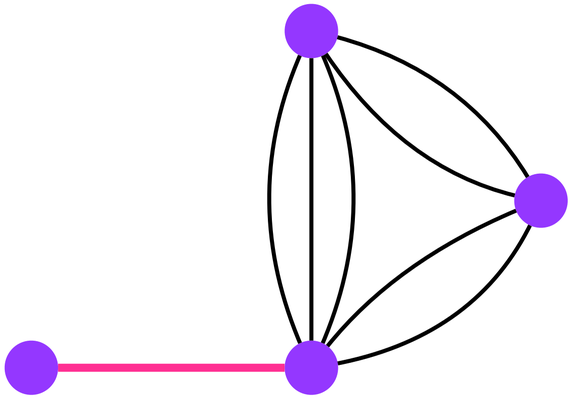
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



Min-Cut

Cut(G)

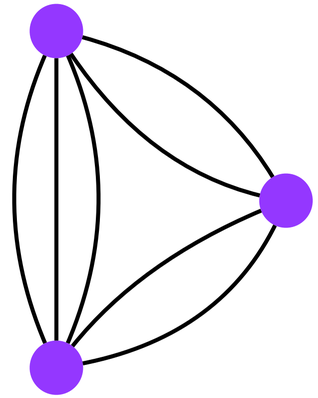
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



Min-Cut

Cut(G)

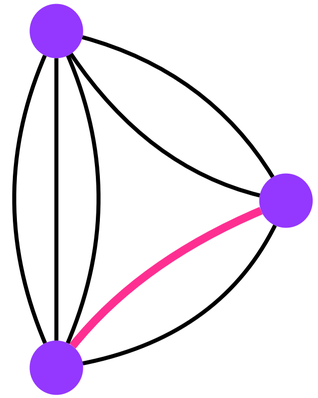
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



Min-Cut

Cut(G)

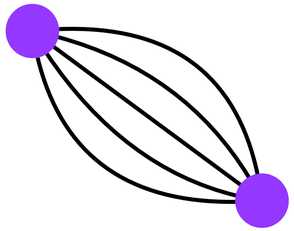
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



Min-Cut

Cut(G)

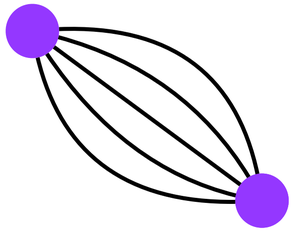
1 : $G' \leftarrow G$

2 : while $|V(G')| > 2$ do

3 : $\quad e \leftarrow$ uniformly random edge in G'

4 : $\quad G' \leftarrow G' \backslash e$

5 : return size of the unique cut in G'



5 !

Min-Cut

Cut(G)

For an edge e :

$$Pr[\mu(G) = \mu(G/e)] \geq 1 - \frac{2}{n}$$

For all G with $|V| = n$, $n \geq 3$:

$\hat{p}(G)$:= Probability that Cut(G) returns the value $\mu(G)$

$$\hat{p}(n) := \inf_{\substack{G=(V,E),\\ |V|=n}} \hat{p}(G) \qquad \hat{p}(n) \geq \left(1 - \frac{2}{n}\right) \cdot \hat{p}(n-1) \qquad \hat{p}(n) \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

Min-Cut

Cut(G)

We repeat the algorithm Cut(G) $\lambda \binom{n}{2}$ times for some $\lambda > 0$ and return the smallest value obtained.

Runtime : O(λn^4)

Success Probability : The smallest encountered value equals $\mu(G)$ with probability at least $1 - e^{-\lambda}$

$\lambda := lnn$, runtime is O($n^4 logn$) with failure probability $\leq 1/n$

we already had a deterministic solution with this runtime !

Min-Cut

Idea : Last steps are critical

Cut(G) + Strategy Switch in the Critical Region

Stop contracting when there are t vertices remaining

switch to a randomized O(t^4) algorithm with success probability $\geq 1 - e^{-1}$

Runtime : $O(\lambda(\frac{n^4}{t^2} + n^2t^2) \overset{t=\sqrt{n}}{=} O(\lambda n^3)$

Success Probability : $\geq 1 - e^{-1}$

Bootstrapping : We can use the same method to improve further. In "Limit" we have a O(n^2polylog(n)) algorithm.

Let’s take a break

Smallest Enclosing Circle

Smallest Enclosing Circle

Problem Description

given : A finite set of points $P \subseteq \mathbb{R}^2$

to find : The circle with the smallest radius that encloses all points in P

C encloses P :

C' := the closed disk bounded by C

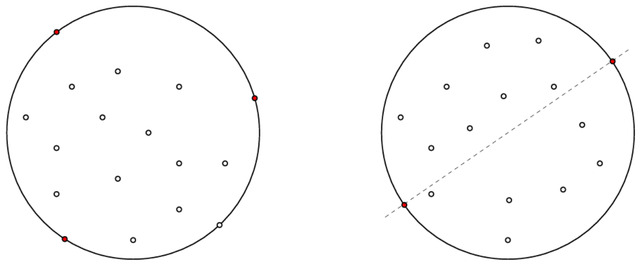
C encloses P if $P \subseteq C'$

Smallest Enclosing Circle

Lemmas

For every finite set of points $P \subseteq \mathbb{R}^2$ there exists a unique smallest enclosing cycle $C(P)$

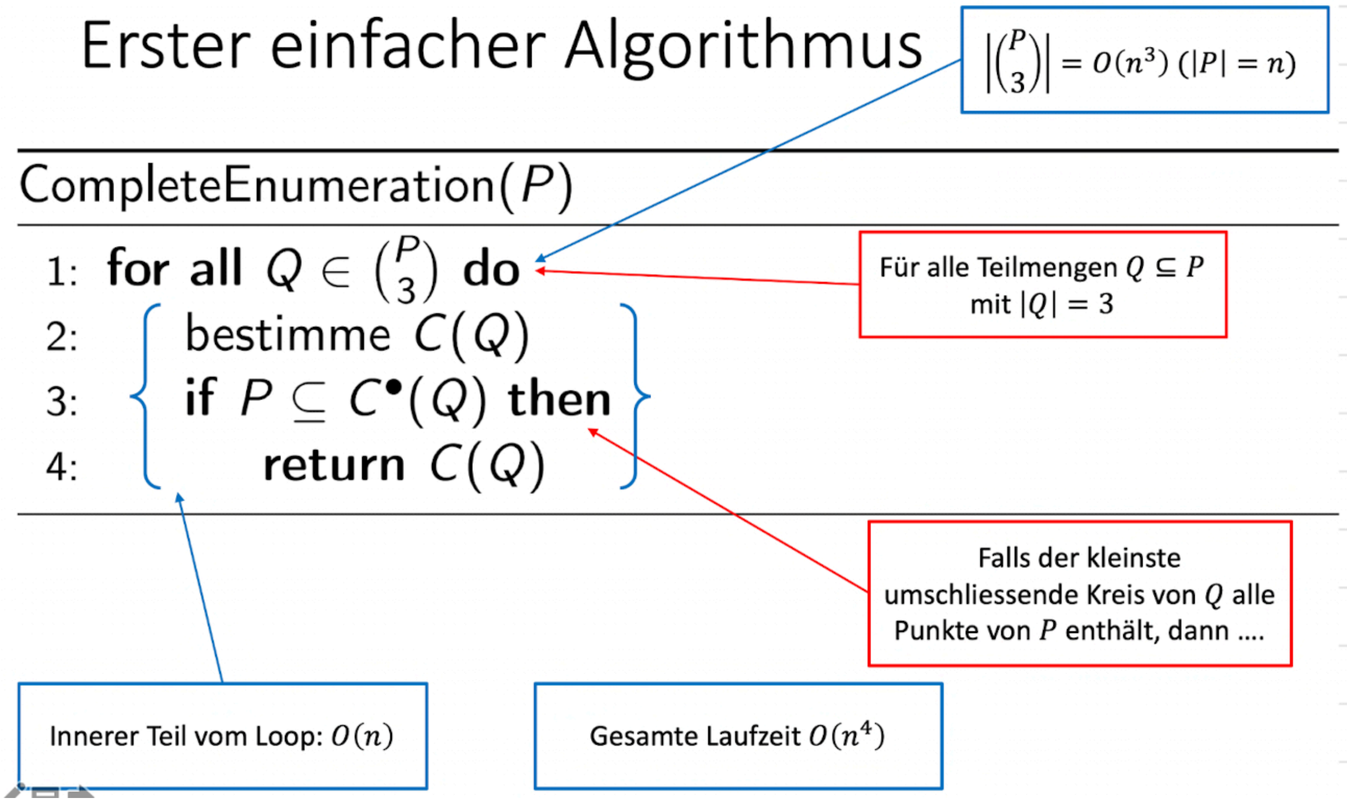
For every finite set of points $P \subseteq \mathbb{R}^2$ with $|P| \geq 3$ there exists a subset $Q \subseteq P$ with $|Q| = 3$ s.t. $C(Q) = C(P)$



Q acts as a certificate for $C(P)$

Smallest Enclosing Circle

Easy Algorithm



Smallest Enclosing Circle

Algorithm

Runtime : O(n log n)

1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

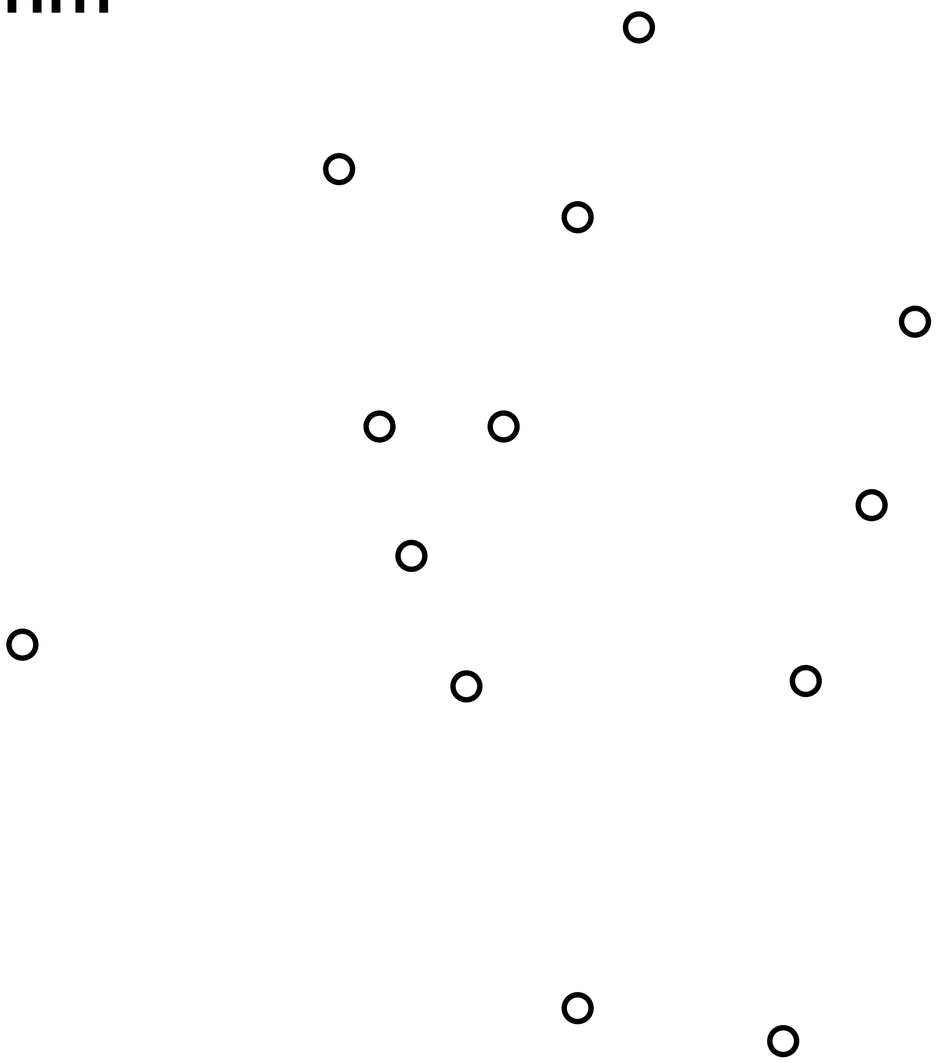
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

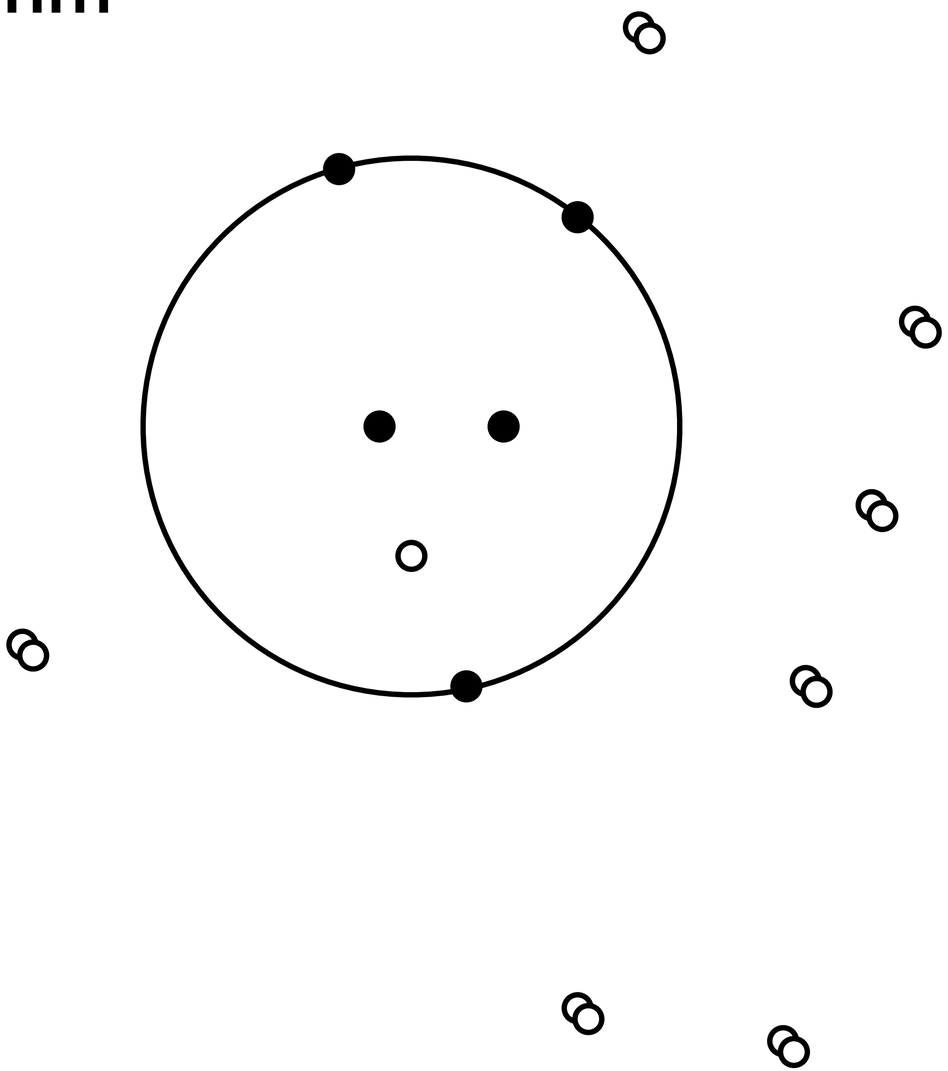
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

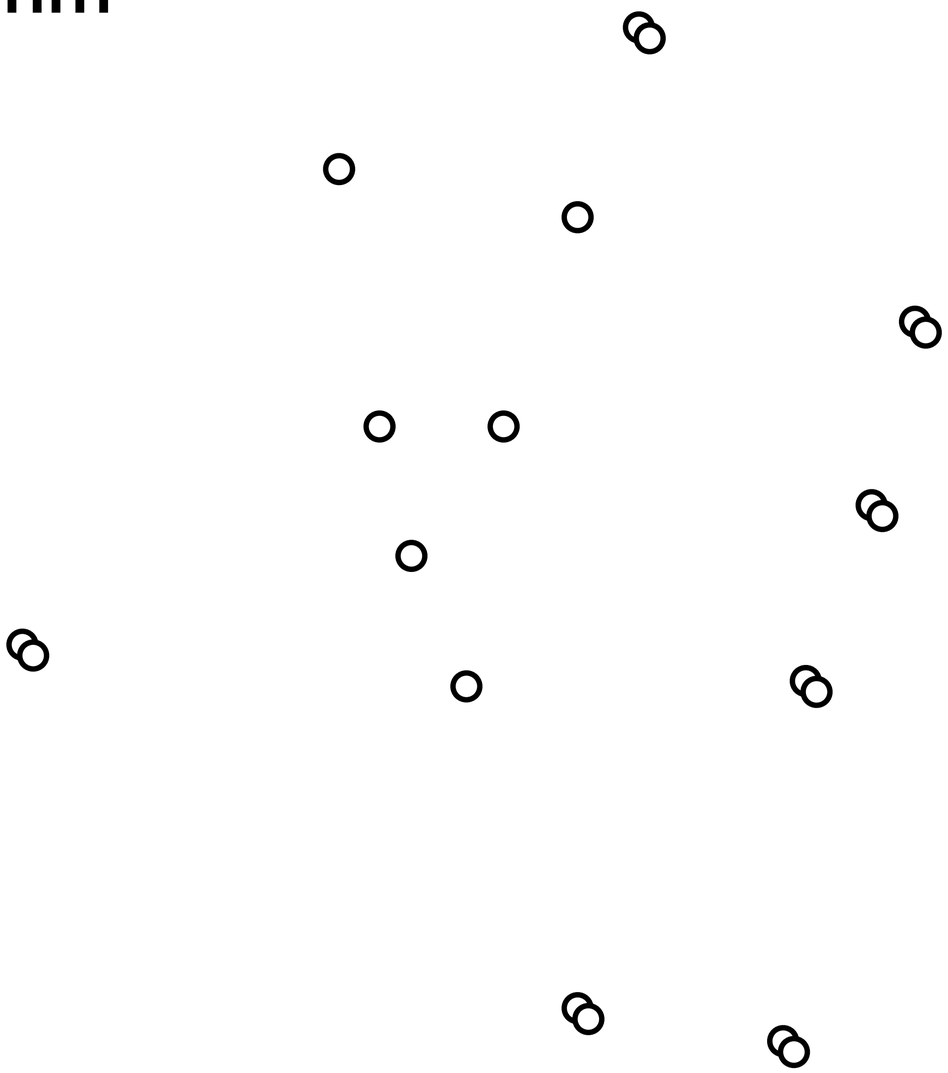
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

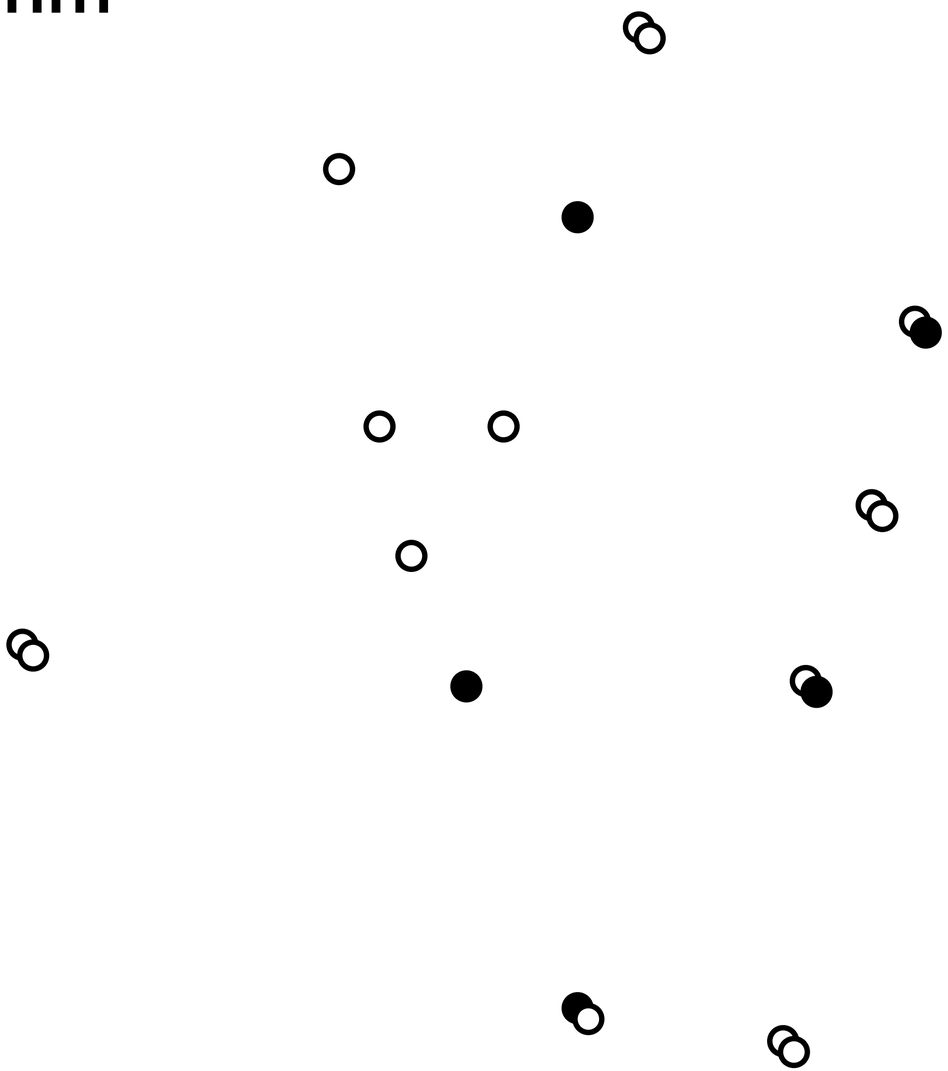
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

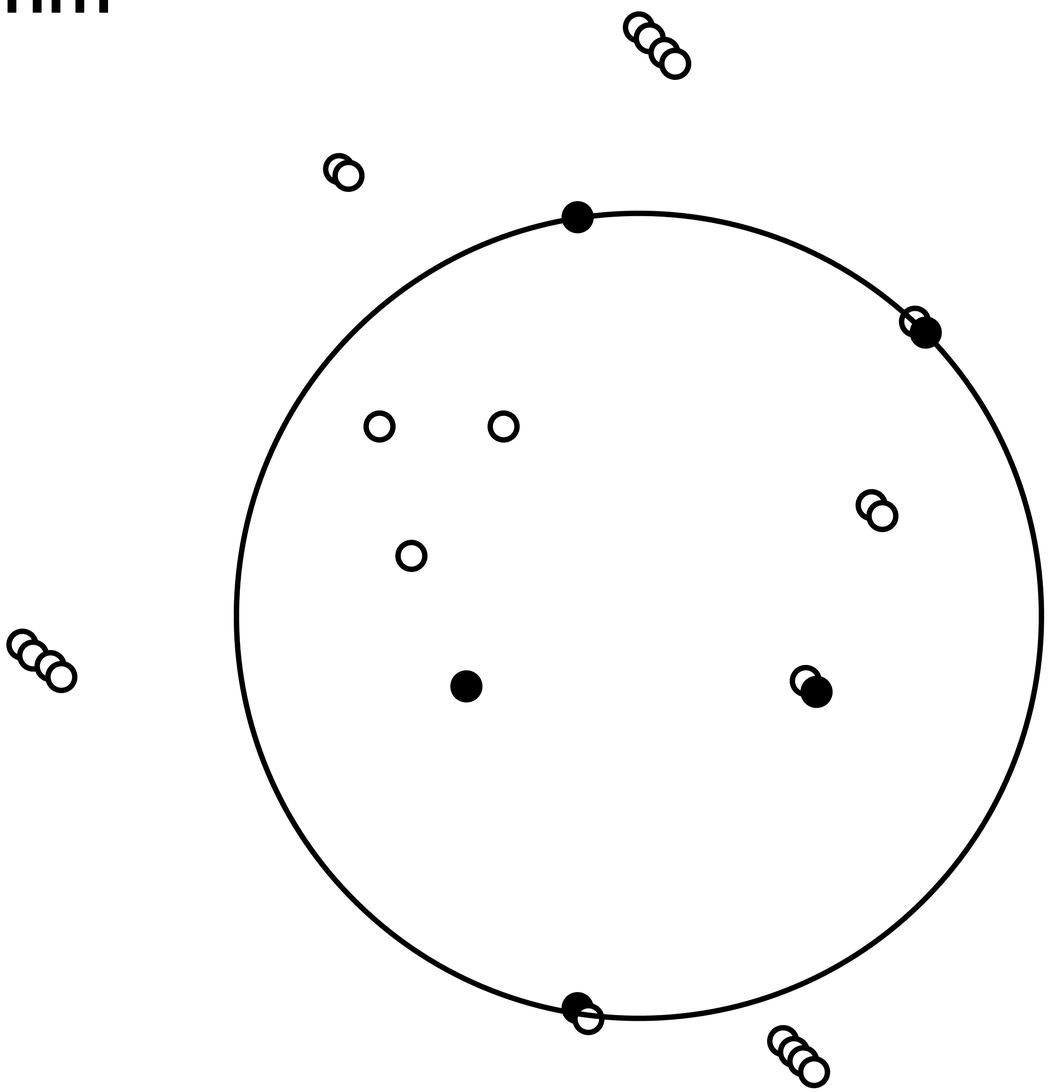
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

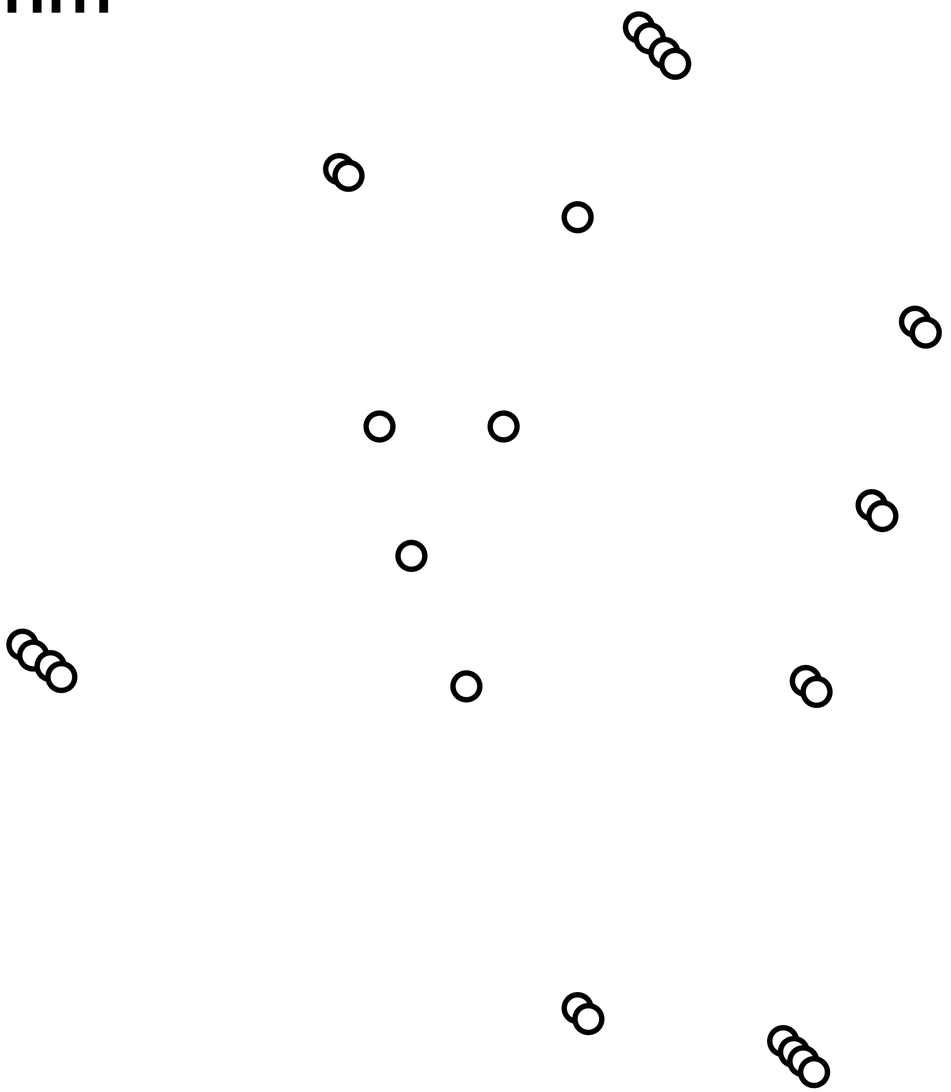
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

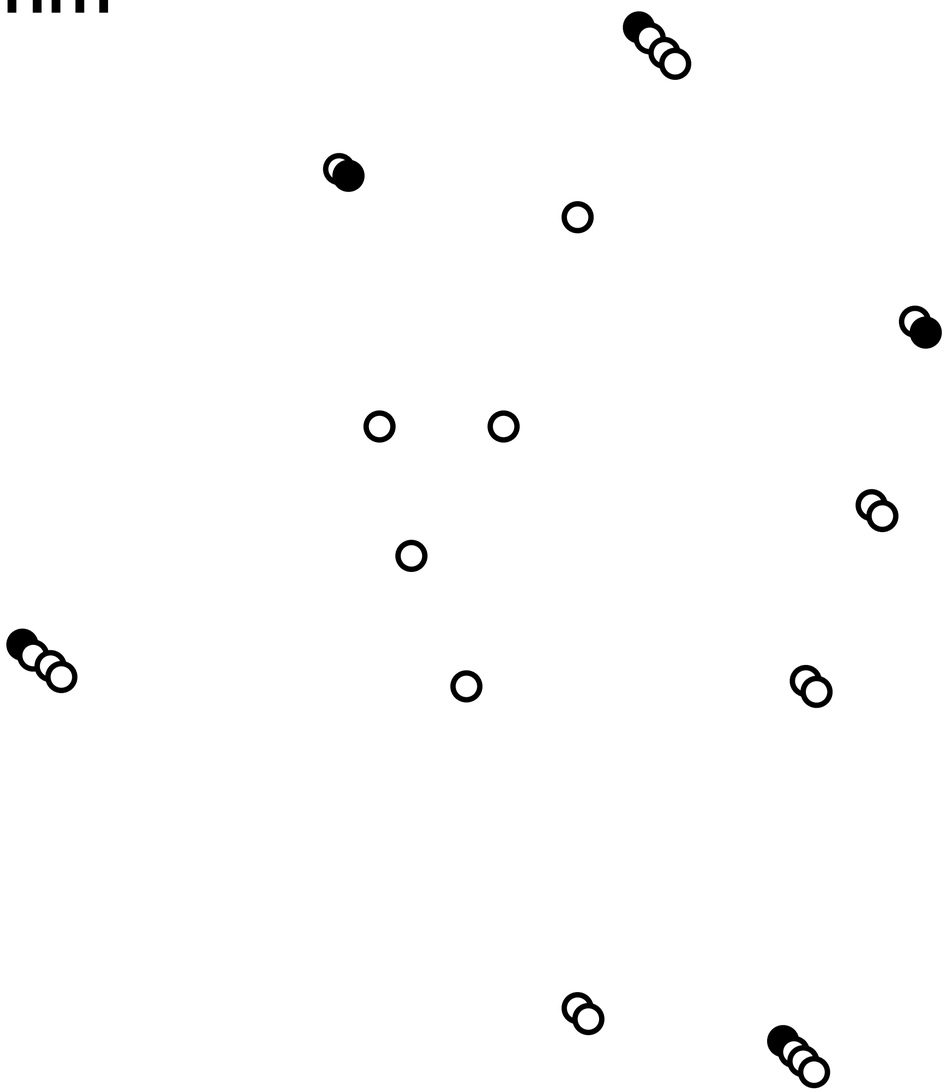
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

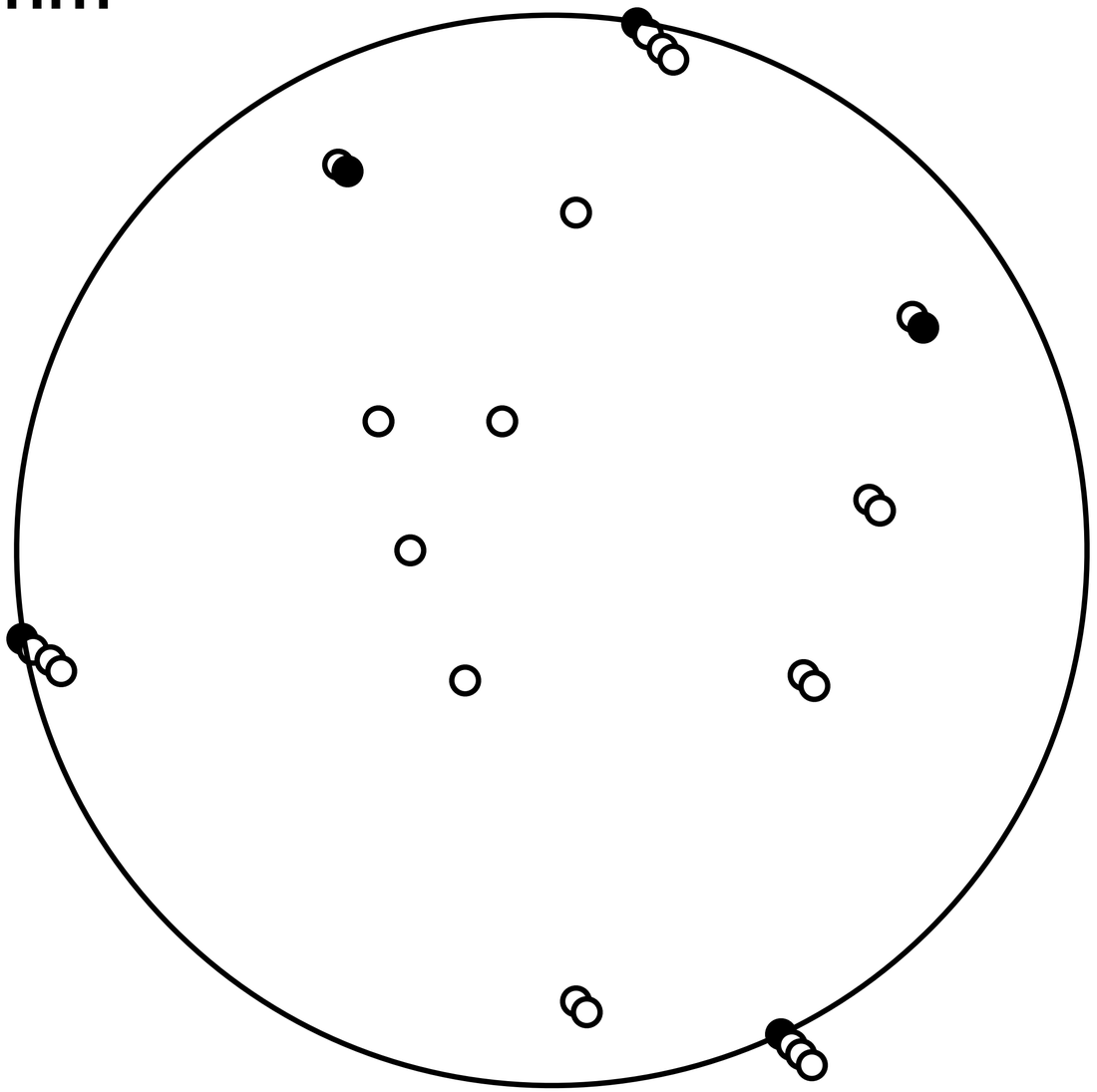
5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Algorithm



1 : $P' \leftarrow P$

2 : **repeat**

3 : randomly and uniformly choose a subset $Q \subseteq P'$ with $|Q| = 11$

4 : compute $C(Q)$

5 : if $P \subseteq C'(Q)$ then return $C(Q)$

6 : else double all points in P' that lie outside of $C(Q)$

7 : **forever**

Smallest Enclosing Circle

Sampling Lemma

Let $r, N \in \mathbb{N}$, $r \leq N$ and $P' \subseteq \mathbb{R}^2$ be a multiset with $|P'| = N$

For R chosen uniformly at random from $\binom{P'}{r}$, the folowing holds :

$$\mathbb{E}\left(\left|\underbrace{P' \setminus C'(R)}_{\text{Points in } P' \text{ outside } C(R)}\right|\right) \leq 3 \cdot \frac{N-r}{r+1} \leq 3 \cdot \frac{N}{r+1}$$

Minitest 5 solutions

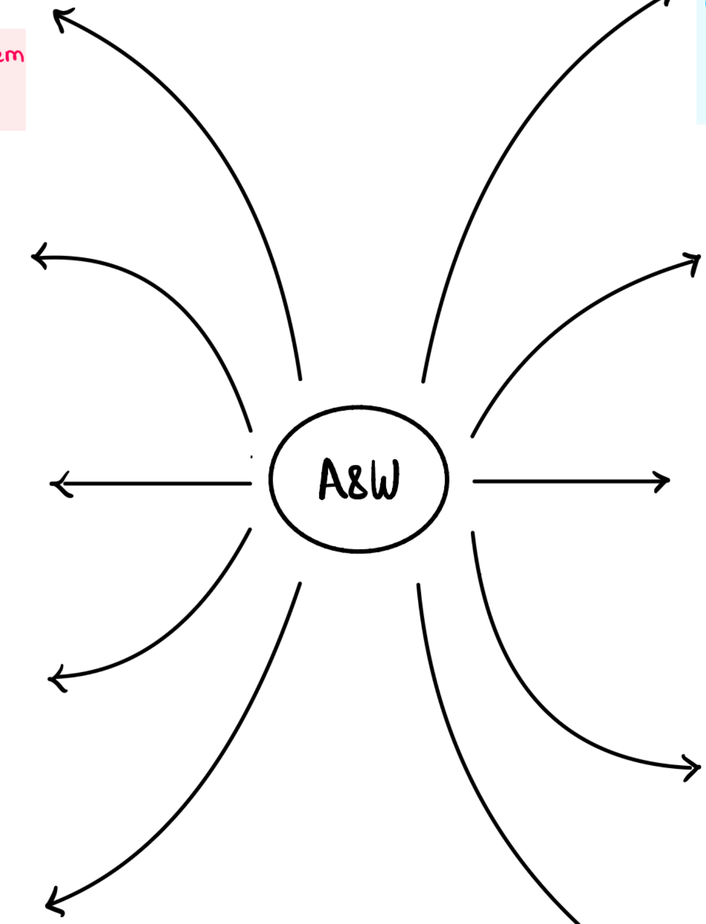
A&W

Exercise Session 13

Convex Hull , Primality Test II

Nil Ozer

A&W Overview



Connectivity

- Articulation Points
- Bridges
- Block - Decomposition
- Menger's Theorem

Cycles

- Eulerian Cycle
- Hamiltonian Cycle
- TSP

Matchings

- Definition
- Algorithms
- Hall's Theorem

Colorings

- Definition
- Algorithm
- Brooks's Theorem

Wahrscheinlichkeit

- Grundbegriffe und Notationen
- Bedingte Wahrscheinlichkeiten
- Unabhängigkeiten
- Zufallsvariablen
- Wichtige Diskrete Verteilungen
- Abschätzen von Wahrscheinlichkeiten

Randomized Algorithms

- Las-Vegas
- Monte-Carlo
- Longest Path Problem
- Primality Test
- Target-Shooting
- Finding Duplicates

Flow

- Definition
- Maxflow - Mincut
- Ford - Fulkerson
- Matching w. Flow
- Edge-disjoint paths w. Flow

Minimum Cut

- Definition
- Cut(G) Algorithm
- Bootstrapping

Graph Algorithms

Convex Hull

- Definition
- Jarvis Wrap
- Local optimization

Smallest Enclosing Circle

- Definition
- First Algorithm
- Final Algorithm

Geometric Algorithms

Last Weeks ...

- 08.05 : Randomized Algorithms II
- 15.05 : Flow
- 23.05 online : Minimum Cut , Smallest Enclosing Circle
- 28.05 extra session : Exam Prep Session + Pizza and Drinks
- 30.05 last extra session : Convex Hull (shortly remaining primality tests)

Outline

- Convex Hull
- Primality Tests II

Convex Hull

Convex Hull

Definitions

- line segment $\overline{v_0 v_1}$:

$$\text{for } v_0, v_1 \in \mathbb{R}^d \ , \ \overline{v_0 v_1} := \{(1-\lambda)v_0 + \lambda v_1 \mid \lambda \in \mathbb{R},\ 0 \leq \lambda \leq 1\}$$

Convex Hull

Definitions

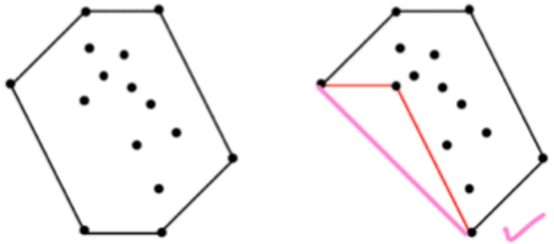
- line segment $\overline{v_0 v_1}$:

$$\text{for } v_0, v_1 \in \mathbb{R}^d \ , \ \overline{v_0 v_1} := \{(1-\lambda)v_0 + \lambda v_1 \mid \lambda \in \mathbb{R},\ 0 \leq \lambda \leq 1\}$$

- convex :

A set $C \subseteq \mathbb{R}^d$ is called convex if $\forall v_0, v_1 \in C : \ \overline{v_0 v_1} \subseteq C$

  - Convex : there is no vertex at which the polygon bends inward.
    - Convex def : If all interior angles are less than 180 degrees, meaning none of the vertices "point inward."
    - Concave def : If at least one interior angle is greater than 180 degrees, meaning at least one vertex "points inward."



Convex Hull

Definitions

- line segment $\overline{v_0 v_1}$:

  for $v_0, v_1 \in \mathbb{R}^d$, $\overline{v_0 v_1} := \{(1-\lambda)v_0 + \lambda v_1 \mid \lambda \in \mathbb{R},\ 0 \leq \lambda \leq 1\}$

- convex :

  A set $C \subseteq \mathbb{R}^d$ is called convex if $\forall v_0, v_1 \in C :\ \overline{v_0 v_1} \subseteq C$

- convex hull conv(S) :

  The convex hull of a set $S \subseteq \mathbb{R}^d$ is the intersection of all convex sets that contain S.

$$\text{conv}(S) := \bigcap_{\substack{S \subseteq C \subseteq \mathbb{R}^d \\ C \text{ konvex}}} C$$

Convex Hull

Problem Description

given : A finite set of points $P \subseteq \mathbb{R}^2$

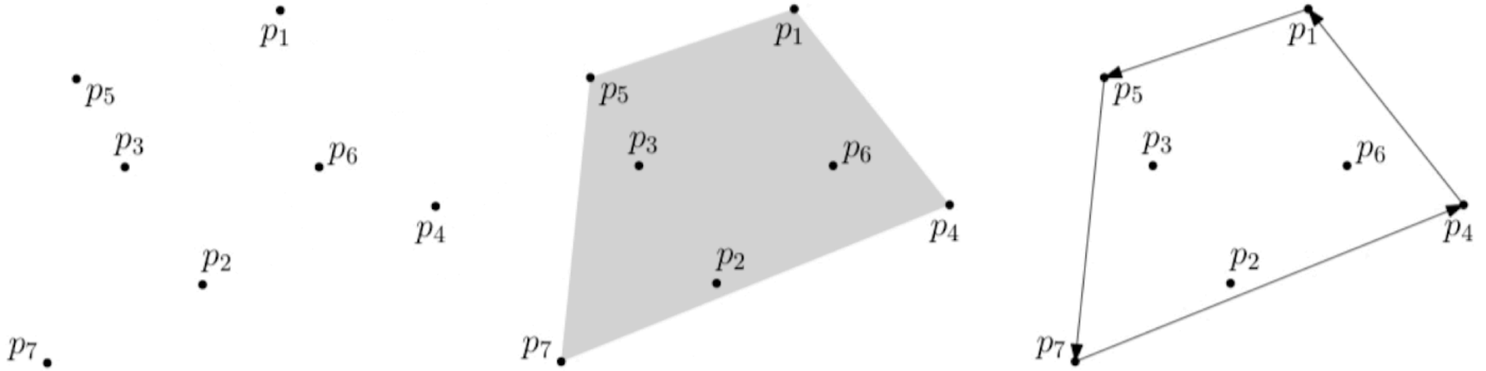
to find : The convex hull of P

- Points are in general position !
  - No three points lie on the same line
  - No two points share the same x-coordinate

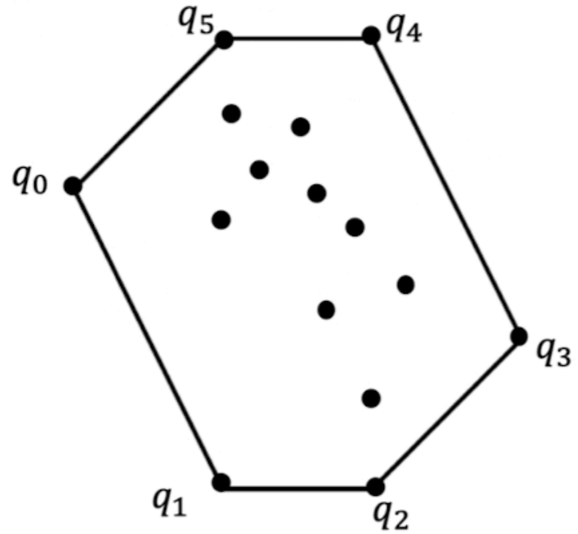
Convex Hull

Problem Description

- Output: We want to determine the vertices of the convex polygon. So, we want to determine a sequence $(q_0, q_1, \ldots, q_{h-1}), \quad h \leq n,$ that defines the vertices in a counterclockwise order.



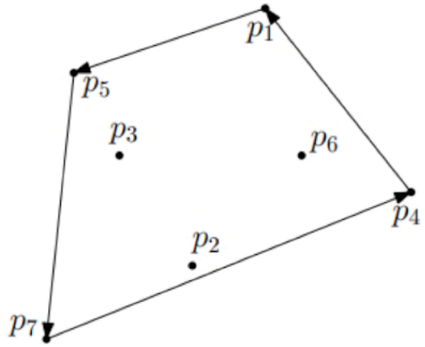
Punktemenge P, konvexe Hülle conv(P), Polygon (p_4, p_1, p_5, p_7).



Convex Hull

Boundary Edge

- A pair qr∈P, with q≠r, is called a *boundary edge* of P , if all points in $P \setminus \{q, r\}$ lie to the left of the line segment qr.



Idea :

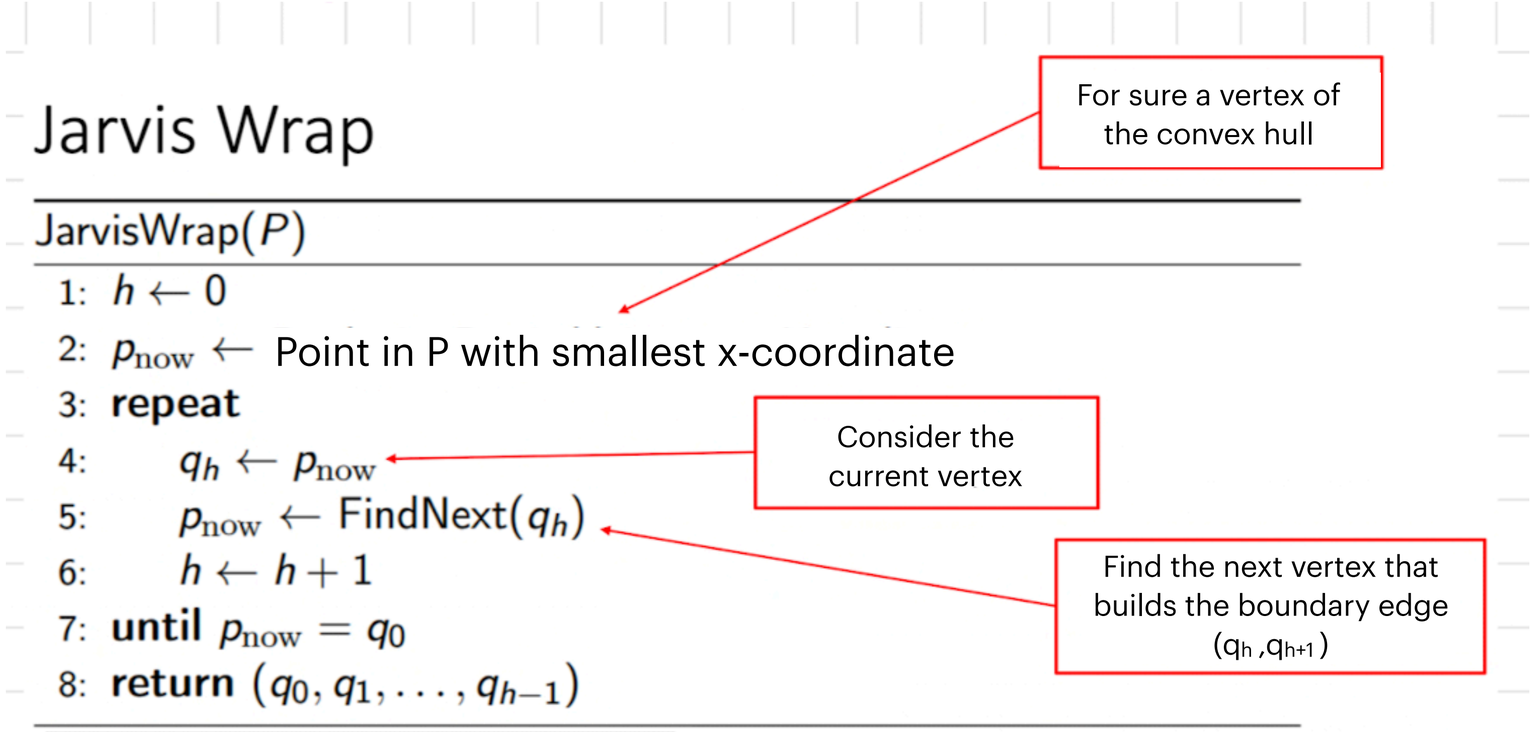
$(q_0, q_1, \ldots, q_{h-1})$ is the sequence of vertices of the polygon enclosing conv(P) in a counterclockwise order

⇔

All pairs (q_{i-1} ,q_i) are boundary edges of P

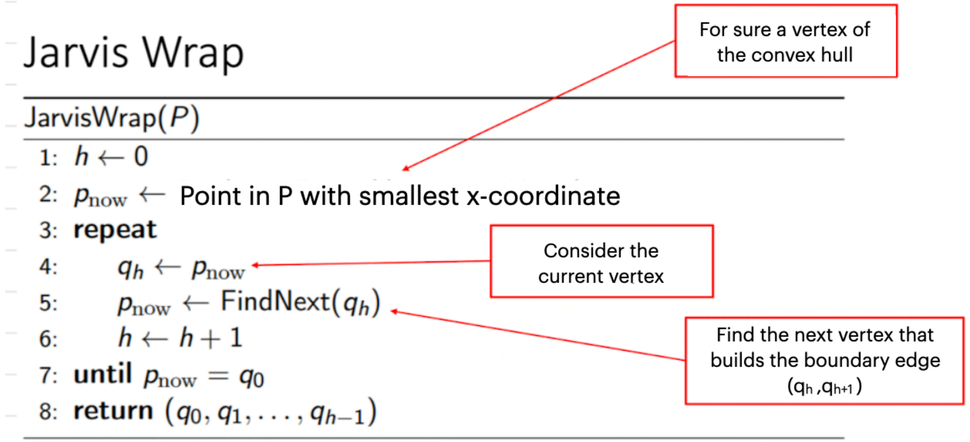
Jarvis Wrap

Algorithm



Jarvis Wrap

Algorithm



FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: **if** p is to the right of qq_{next} **then**
5: $q_{\text{next}} \leftarrow p$
6: **return** q_{next}

Jarvis Wrap
Runtime

O (nh)

h is the #vertices of the convex hull

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}

Orientierungstest

Wie entscheiden wir „p liegt links von qr“?

Lemma
Seien $p = (p_x, p_y)$, $q = (q_x, q_y)$, und $r = (r_x, r_y)$ Punkte in $\mathbb{R}^2$. Es gilt $q \neq r$ und p liegt links von qr genau dann wenn

$$\det(p,q,r) := \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = \begin{vmatrix} q_x - p_x & q_y - p_y \\ r_x - p_x & r_y - p_y \end{vmatrix} > 0$$

$$\Leftrightarrow \quad (q_x - p_x)(r_y - p_y) > (q_y - p_y)(r_x - p_x)$$

$\frac{1}{2}|\det(p,q,r)|$ = die Fläche des Dreiecks pqr

Das Vorzeichen von $\det(p,q,r)$ bestimmt, wie die Punkte p, q, r den Rand dieses Dreiecks durchlaufen.

Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

For sure a vertex of the convex hull

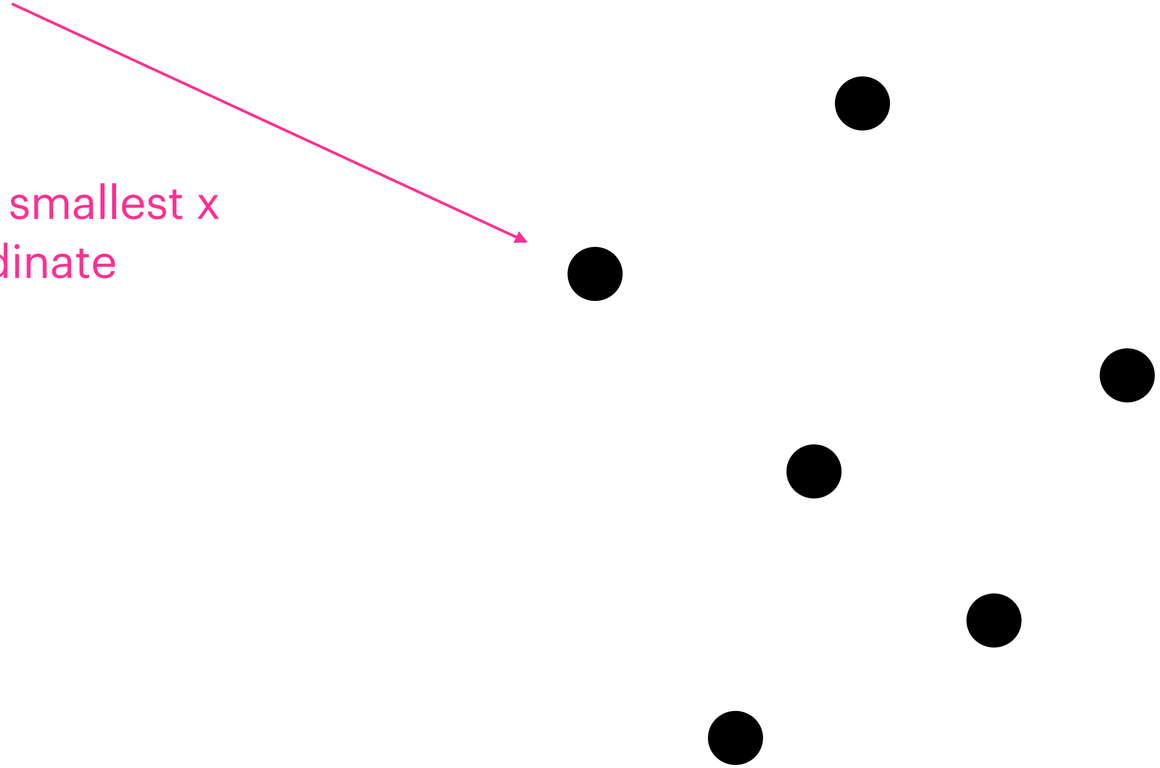
Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}

point with smallest x coordinate



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

For sure a vertex of the convex hull

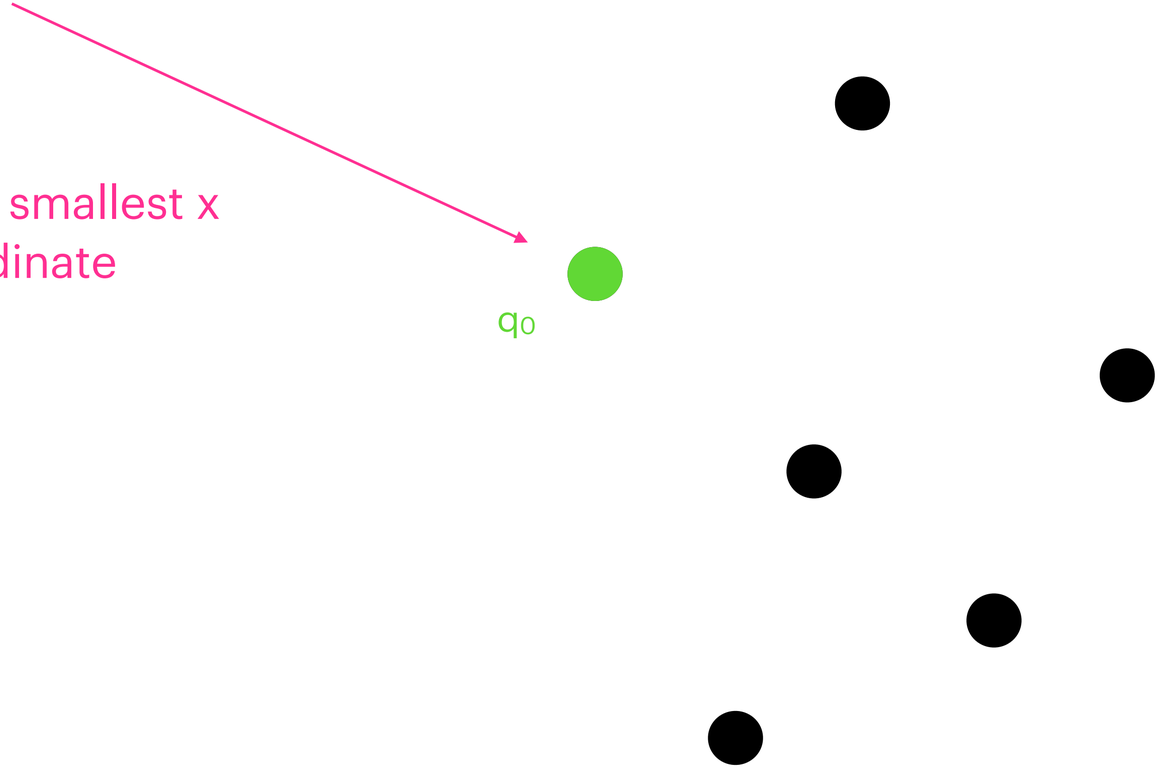
Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}

point with smallest x coordinate



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

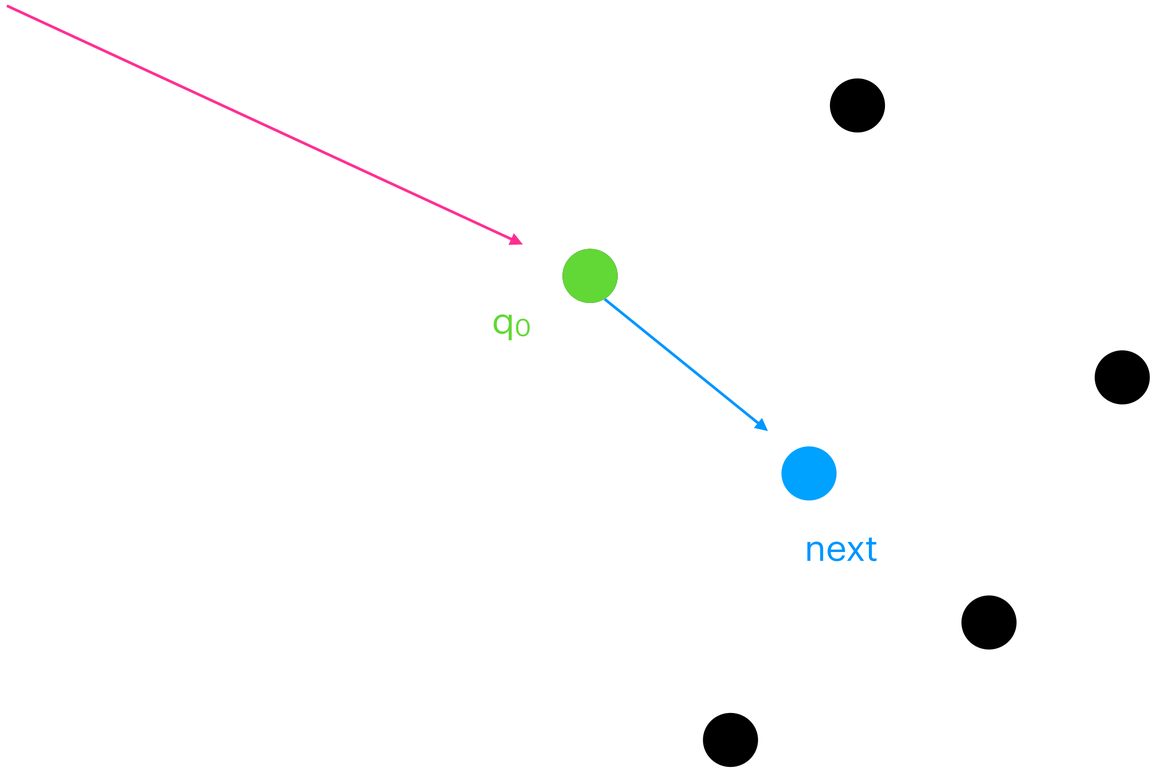
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

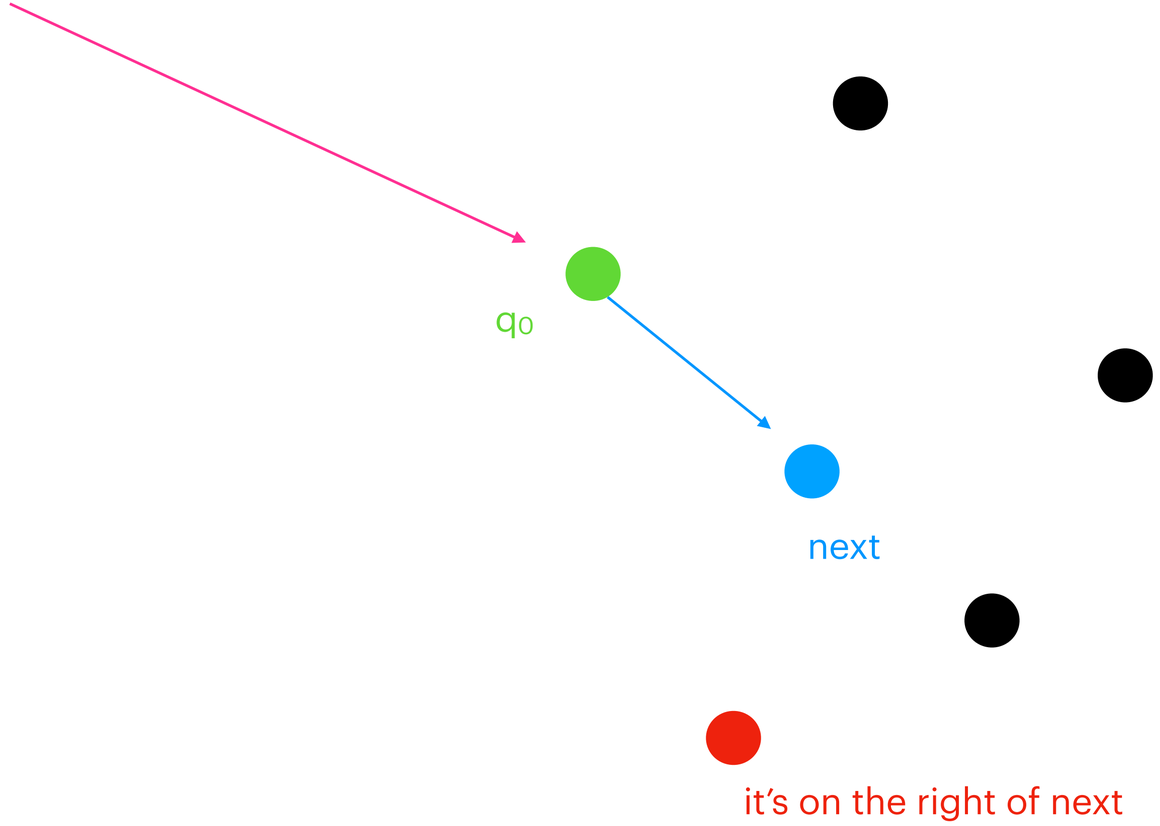
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap
Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

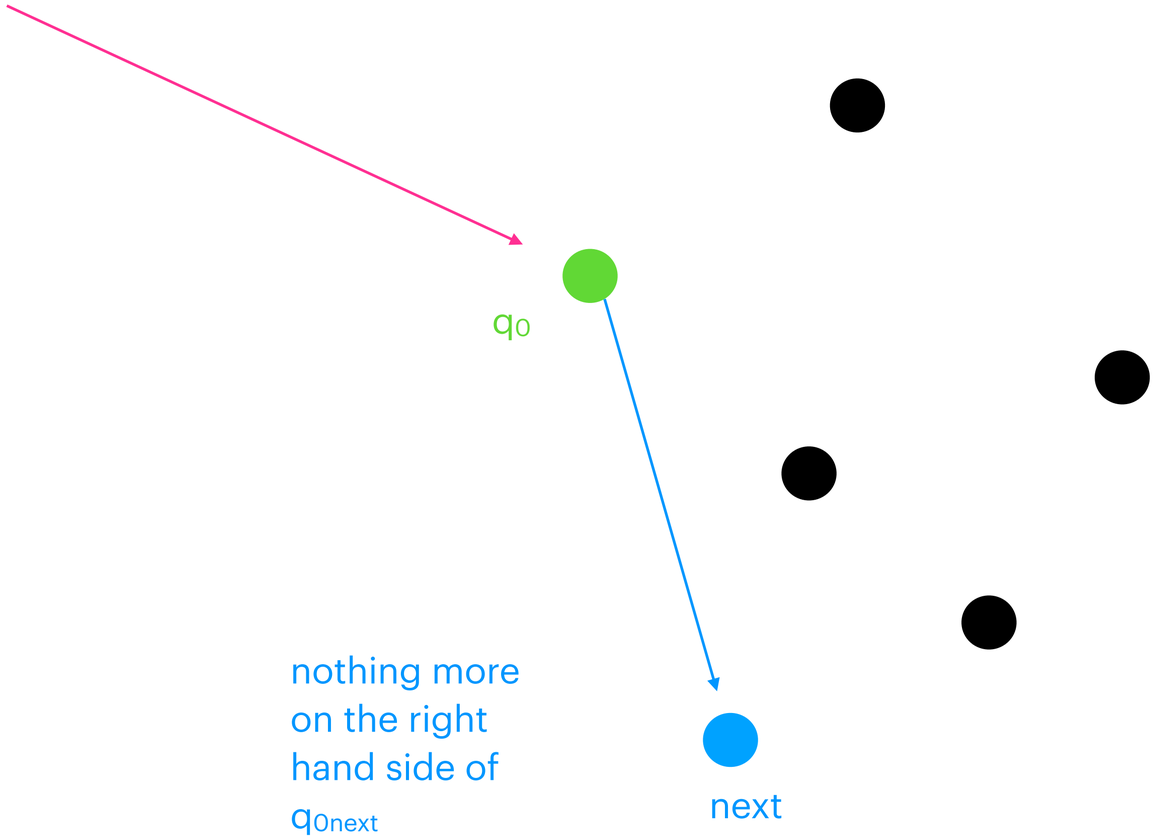
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

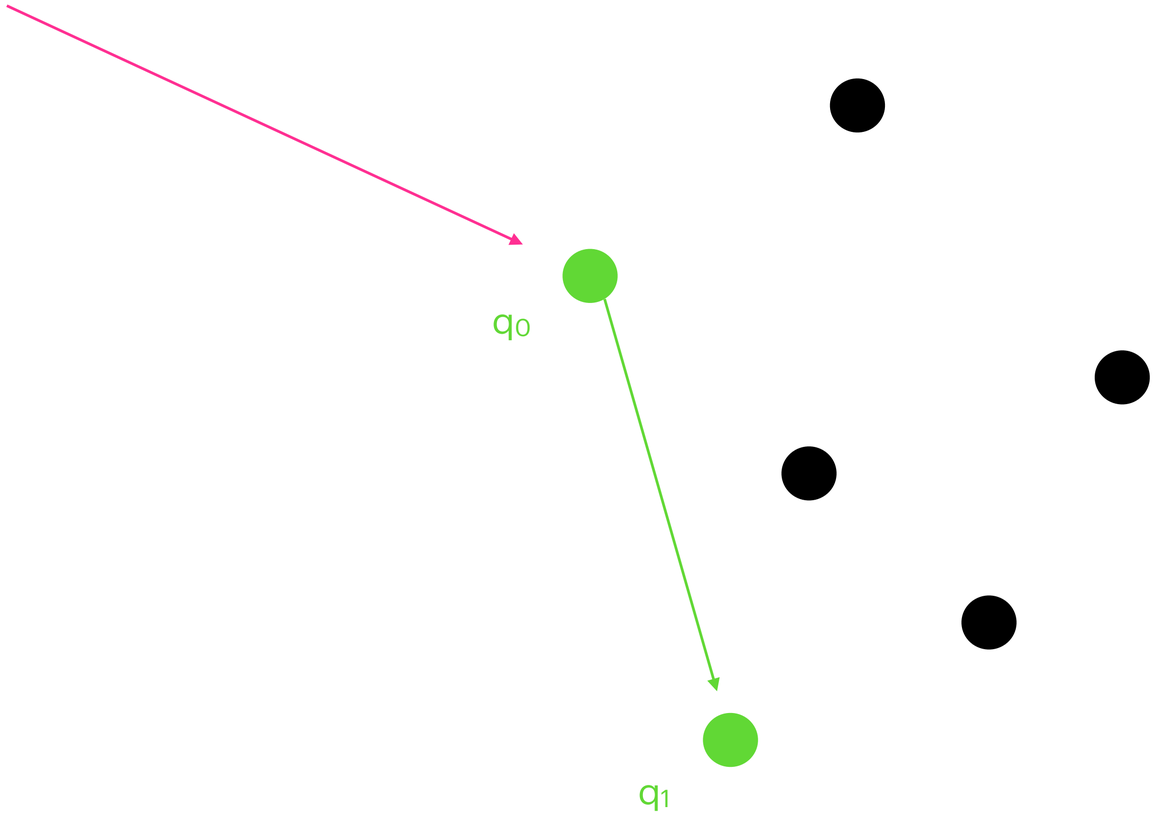
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

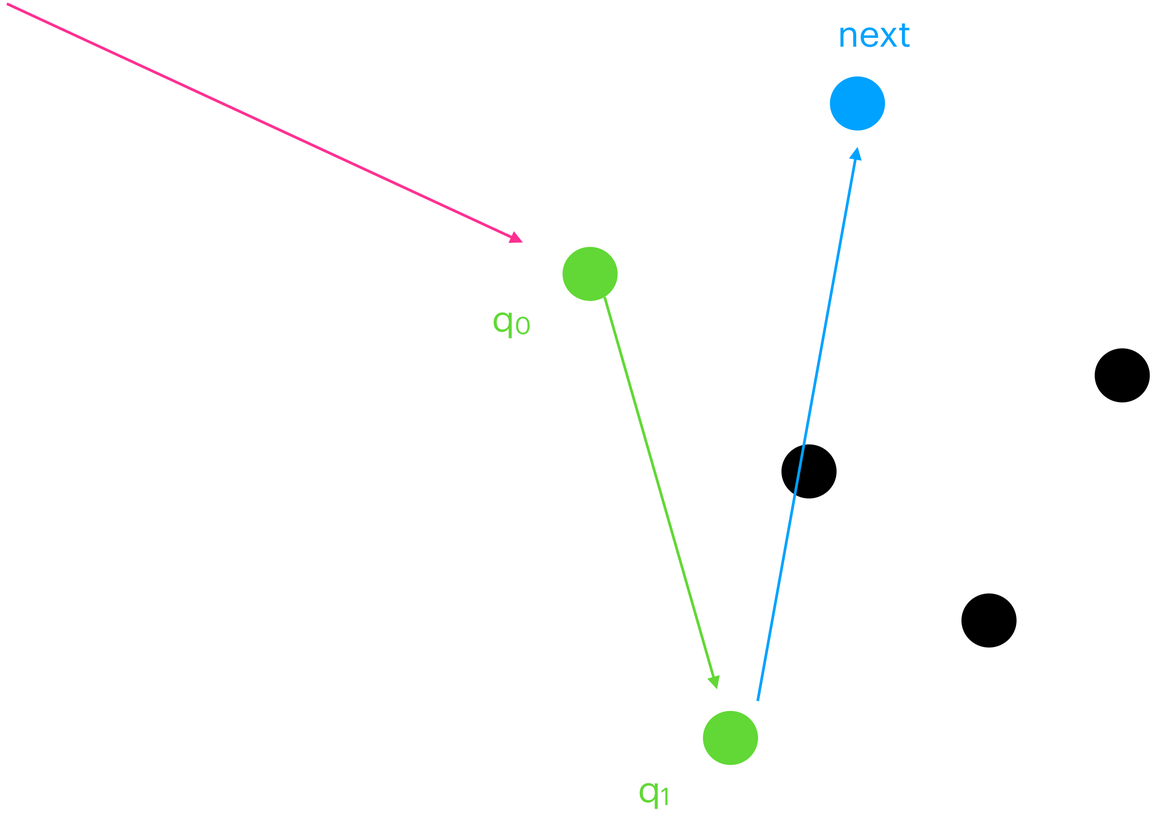
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

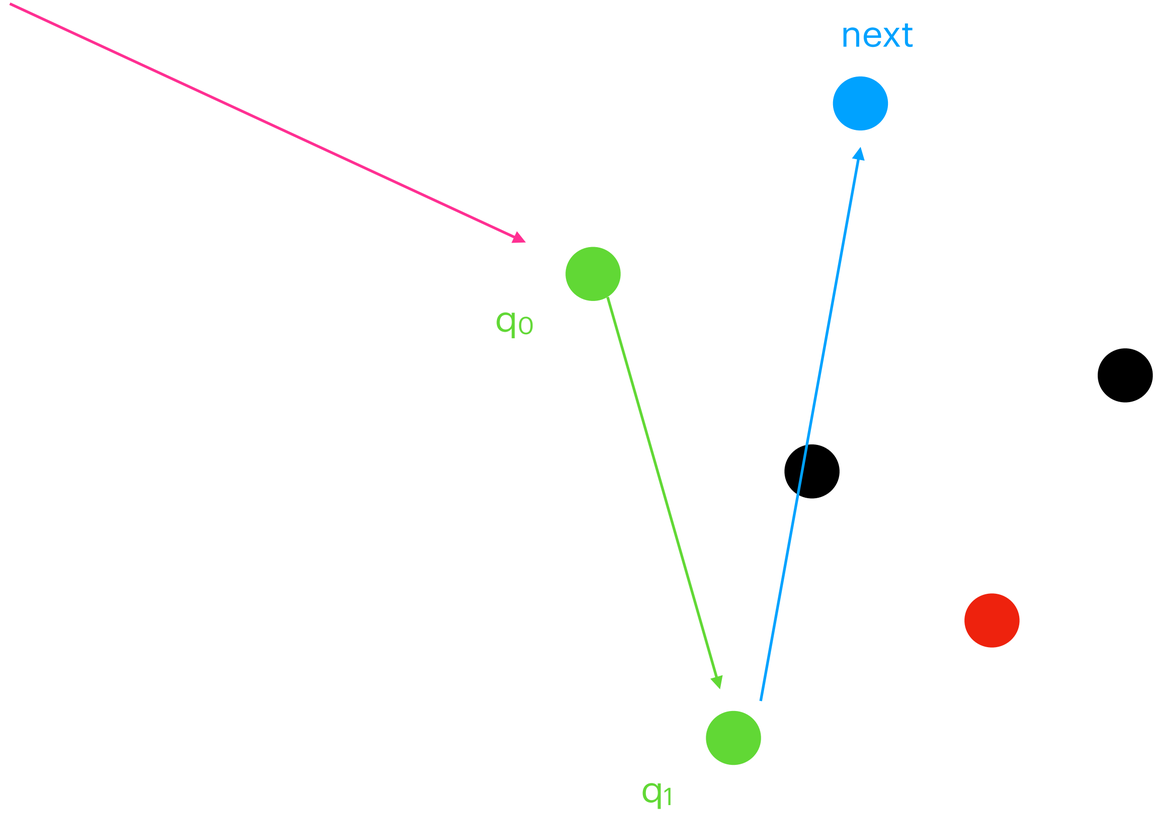
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

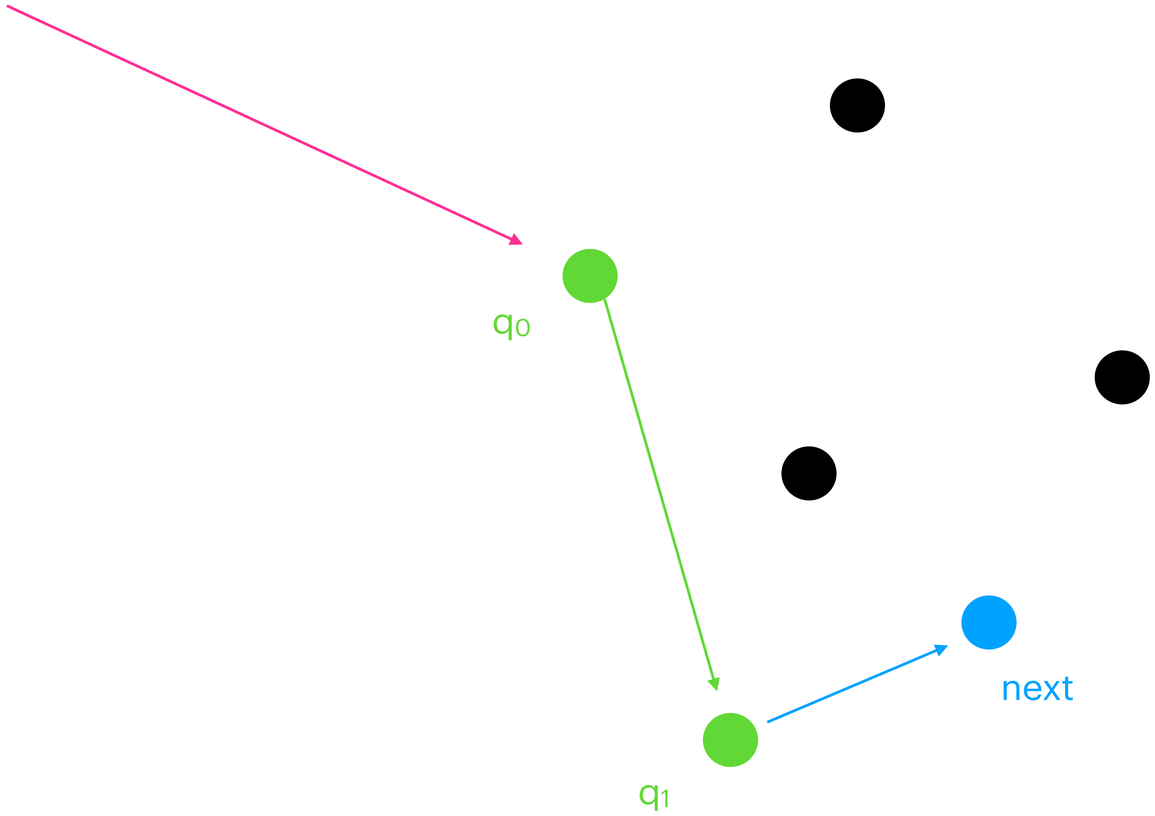
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

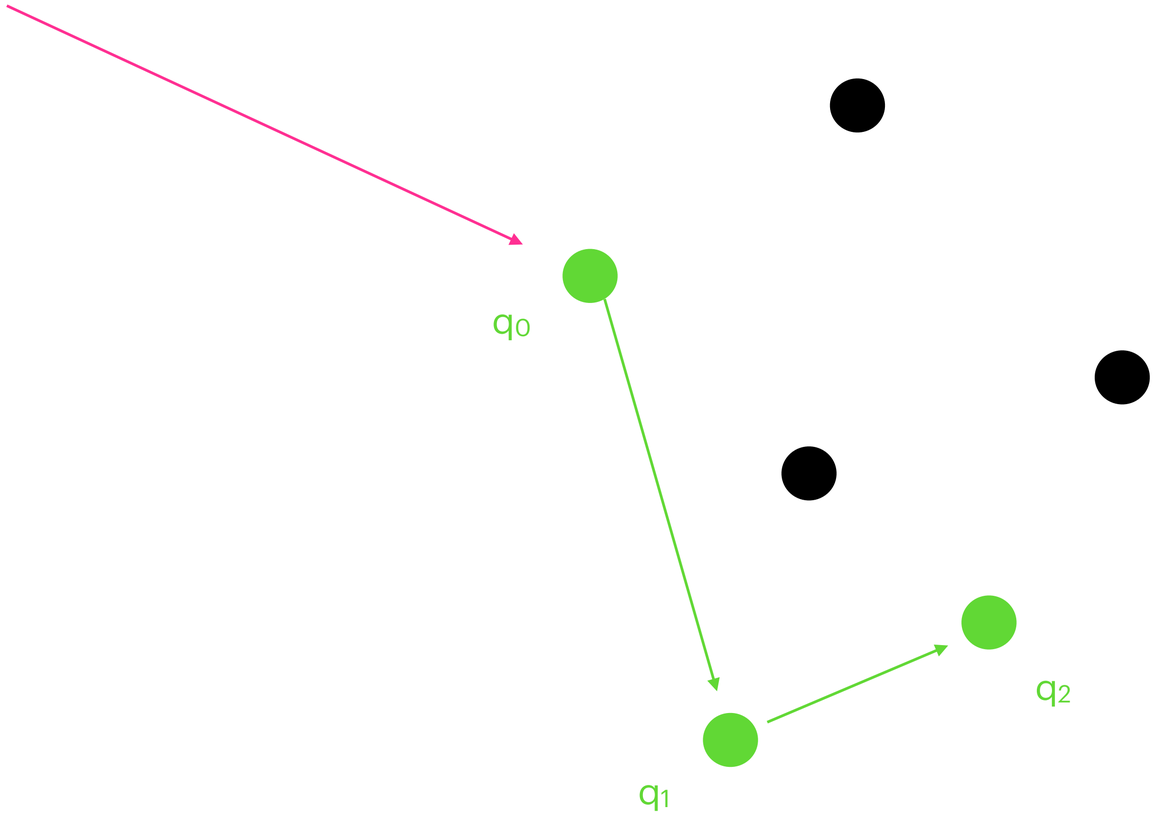
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

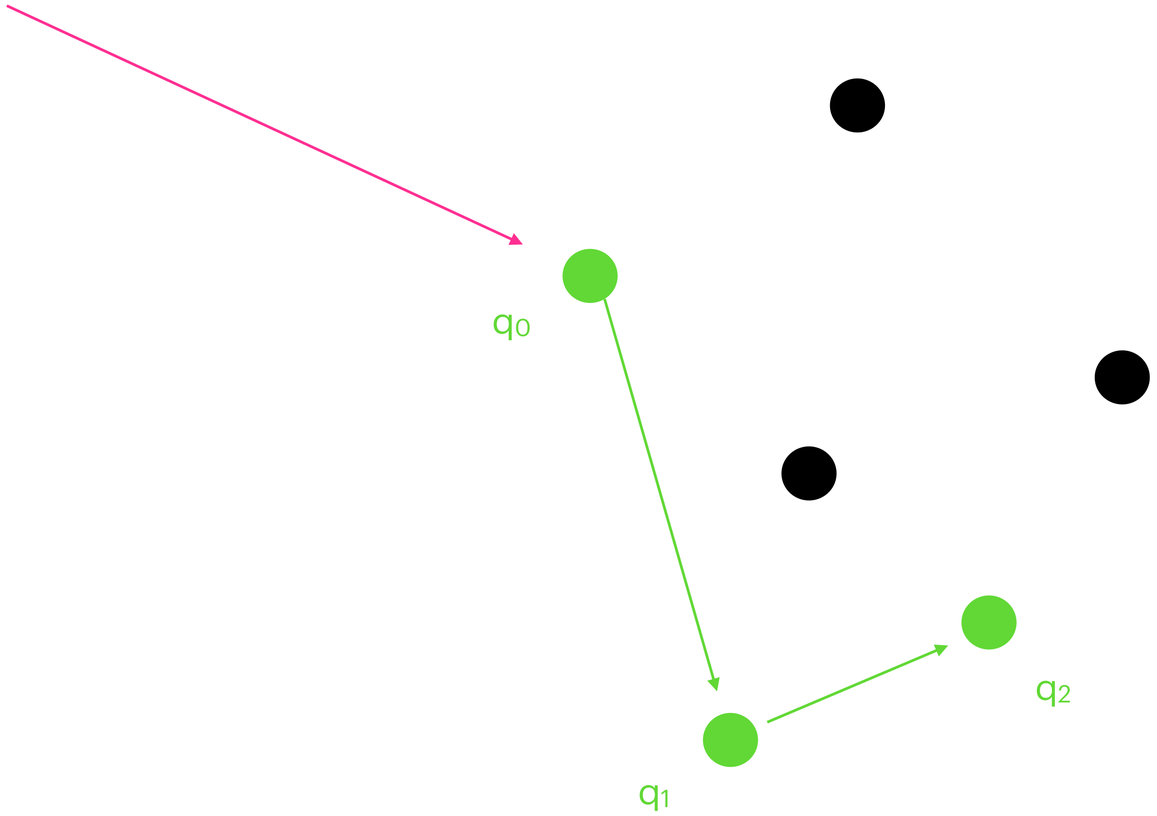
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

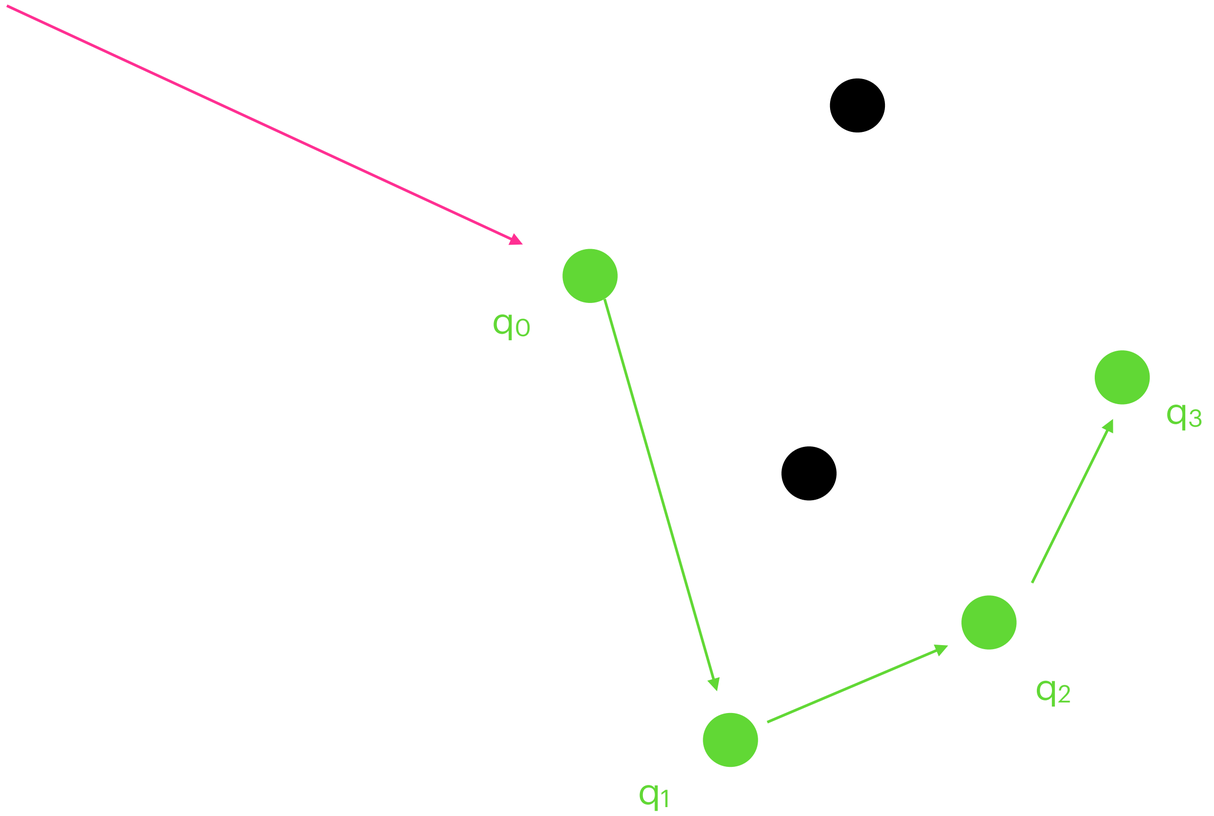
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

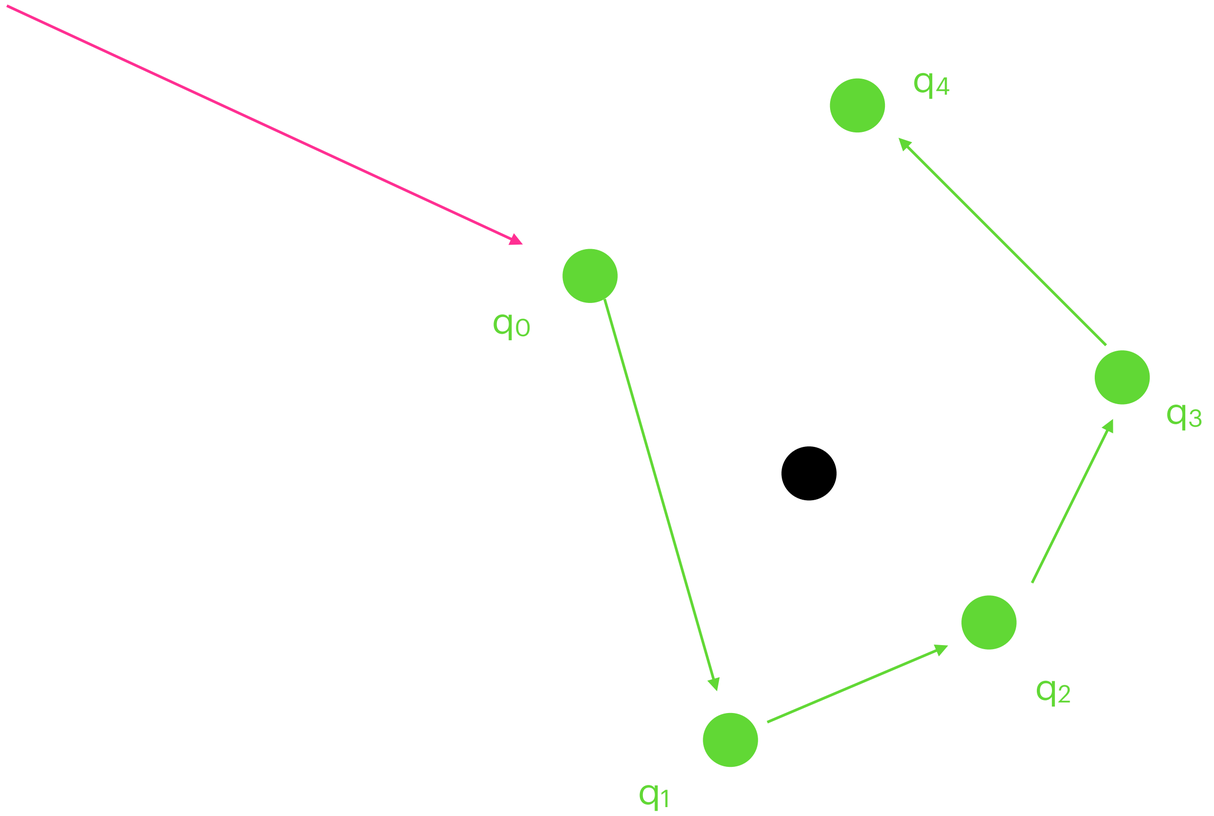
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Jarvis Wrap

Illustration

Jarvis Wrap

JarvisWrap(P)

1: $h \leftarrow 0$
2: $p_{\text{now}} \leftarrow$ Point in P with smallest x-coordinate
3: **repeat**
4: $\quad q_h \leftarrow p_{\text{now}}$
5: $\quad p_{\text{now}} \leftarrow \text{FindNext}(q_h)$
6: $\quad h \leftarrow h + 1$
7: **until** $p_{\text{now}} = q_0$
8: **return** $(q_0, q_1, \ldots, q_{h-1})$

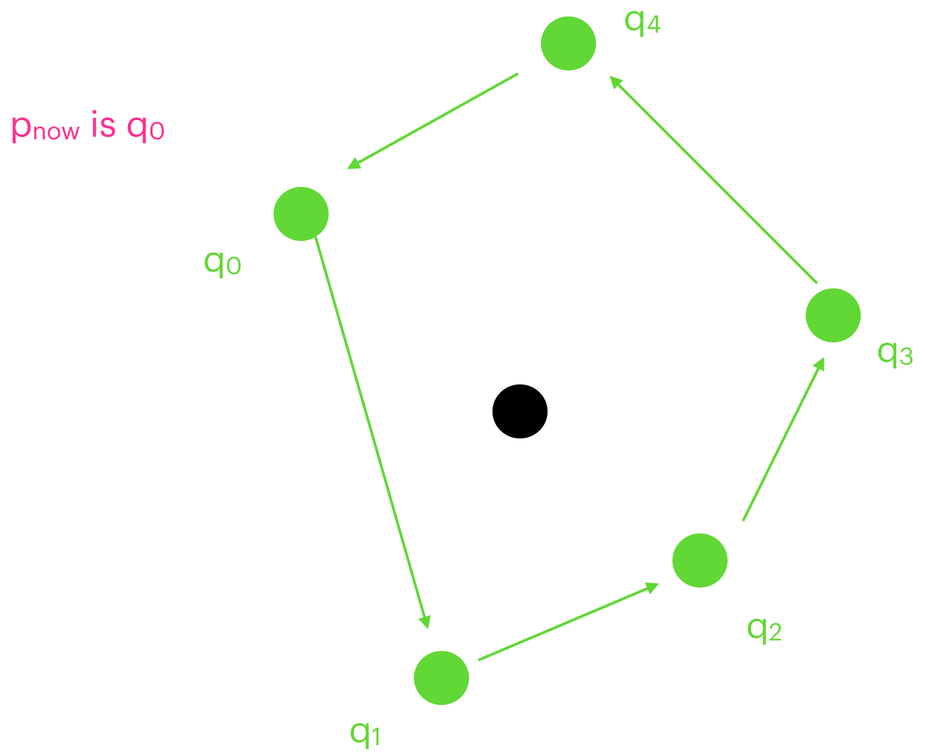
For sure a vertex of the convex hull

Consider the current vertex

Find the next vertex that builds the boundary edge (q_h, q_{h+1})

FindNext(q)

1: Choose $p_0 \in P \setminus \{q\}$ arbitrarily
2: $q_{\text{next}} \leftarrow p_0$
3: **for all** $p \in P \setminus \{q, p_0\}$ **do**
4: $\quad$ **if** p is to the right of qq_{next} **then**
5: $\quad\quad q_{\text{next}} \leftarrow p$
6: **return** q_{next}



Let’s take a break

Convex Hull

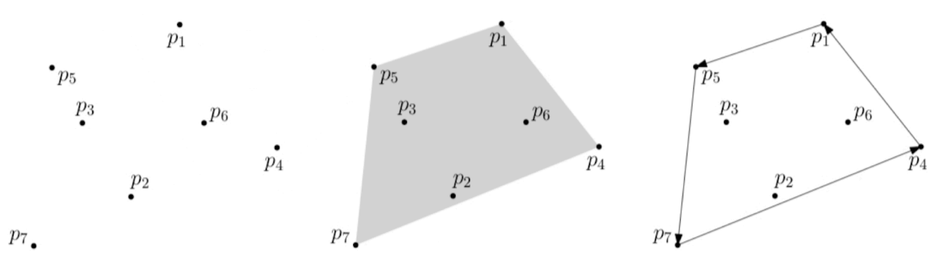
Problem Description

given : A finite set of points $P \subseteq \mathbb{R}^2$

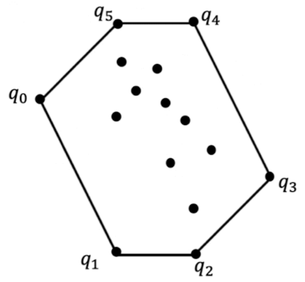
to find : The convex hull of P

- Points are in general position !
  - No three points lie on the same line
  - No two points share the same x-coordinate

- Output: We want to determine the vertices of the convex polygon. So, we want to determine a sequence $(q_0, q_1, \ldots, q_{h-1}), \quad h \leq n,$ that defines the vertices in a counterclockwise order.



Punktemenge P, konvexe Hülle conv(P), Polygon (p_4, p_1, p_5, p_7).



Convex Hull

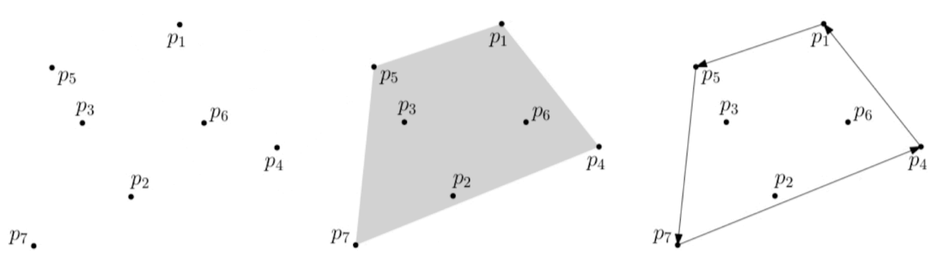
Problem Description

given : A finite set of points $P \subseteq \mathbb{R}^2$

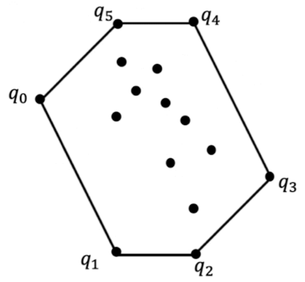
to find : The convex hull of P

- Points are in general position !
  - No three points lie on the same line
  - No two points share the same x-coordinate

- Output: We want to determine the vertices of the convex polygon. So, we want to determine a sequence $(q_0, q_1, \ldots, q_{h-1}), \quad h \leq n,$ that defines the vertices in a counterclockwise order.



Punktemenge P, konvexe Hülle conv(P), Polygon (p_4, p_1, p_5, p_7).

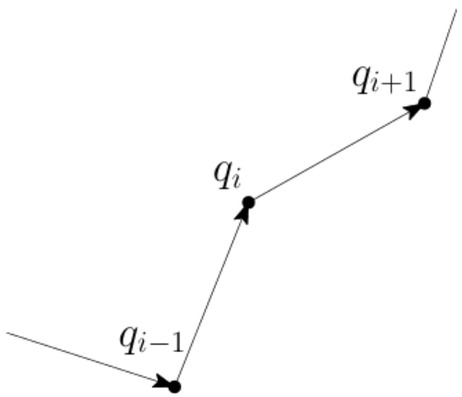


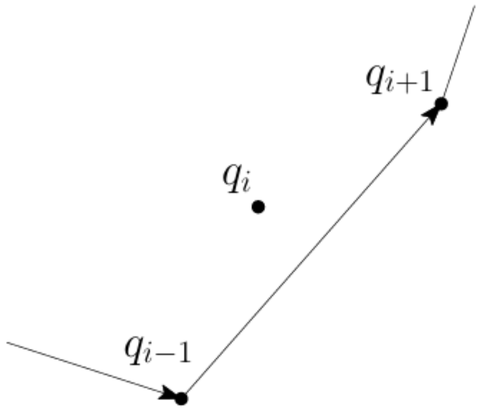
Convex Hull

Local Improvement

Idea :

Given a sequence of points$(q_0, q_1, \ldots, q_{h-1})$, if a point q_i lies to the left of the line segment from q_{i-1} to q_{i+1} , remove q_i from the sequence





Local Repair

Algorithm

O (nlogn)

O(nlogn) sorting, O(n) local repair

LocalRepair$(p_1, p_2, \ldots, p_n)$ $(p_1, p_2, \ldots, p_n)$ sorted

in increasing x-coordinate order

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

$(q_0, \ldots, q_h)$ is lower hull of $\{p_1, \ldots, p_i\}$

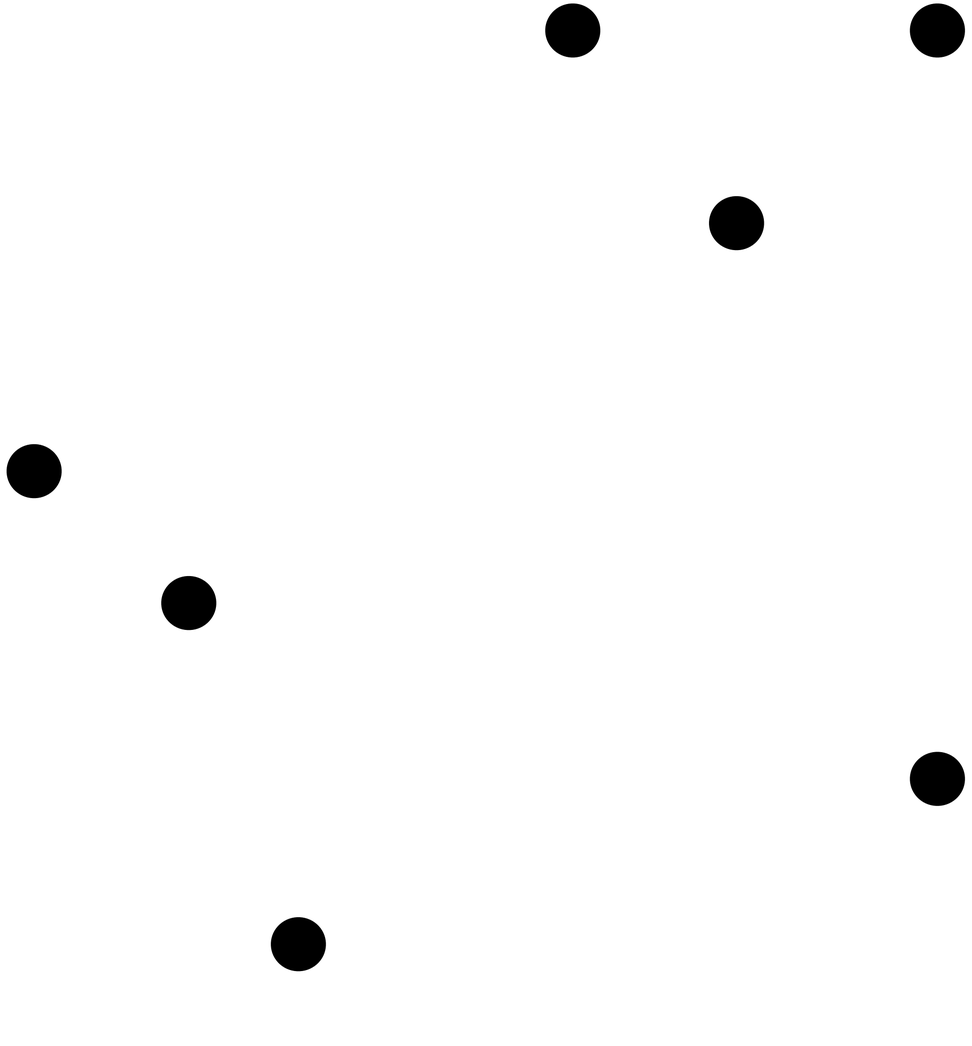
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

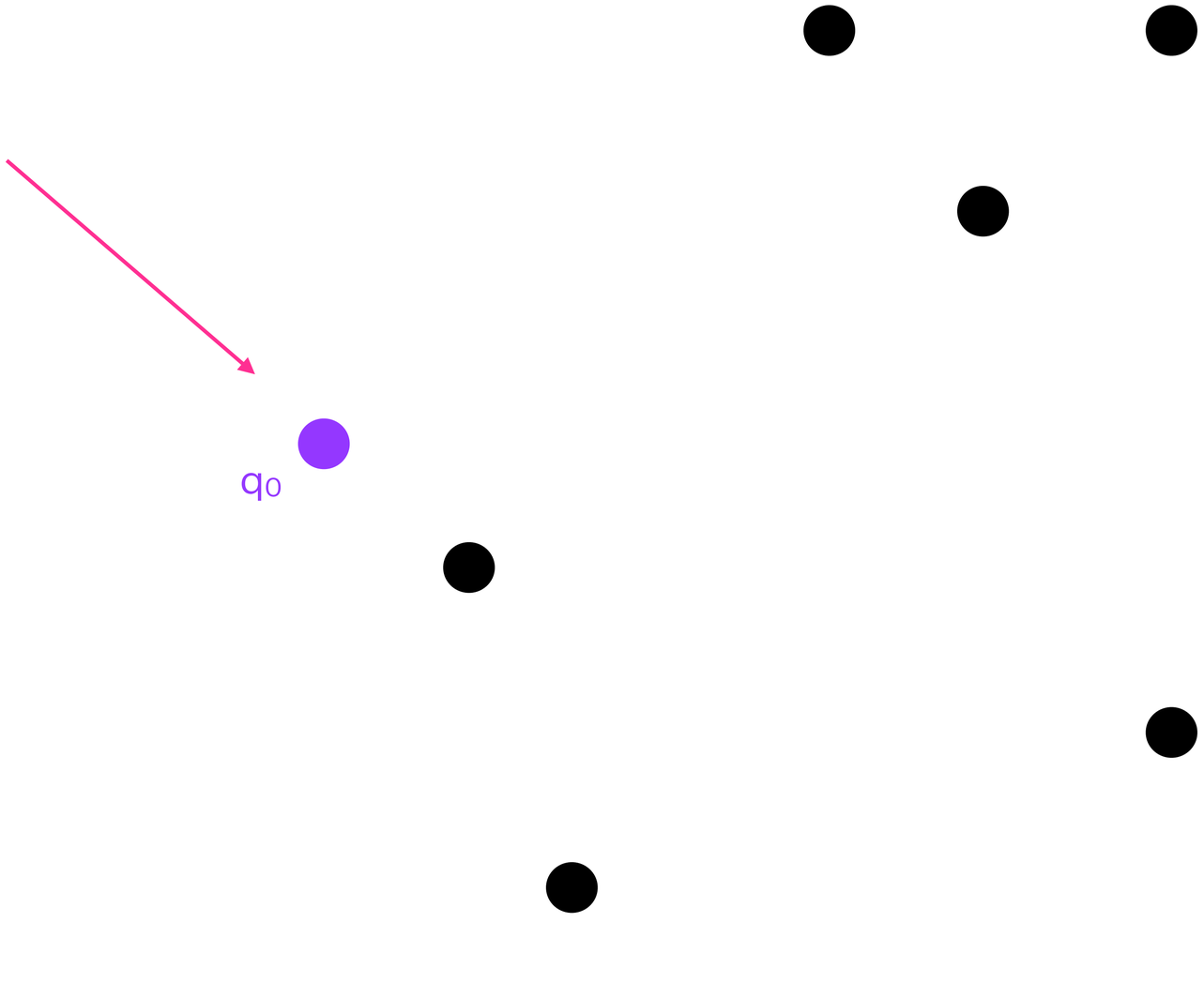
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair$(p_1, p_2, \ldots, p_n)$ $(p_1, p_2, \ldots, p_n)$ sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

setting new points

$(q_0, \ldots, q_h)$ is lower hull of $\{p_1, \ldots, p_i\}$

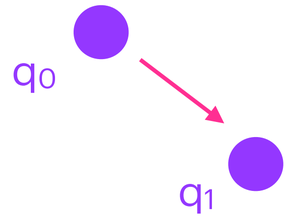
upper rand (right to left)

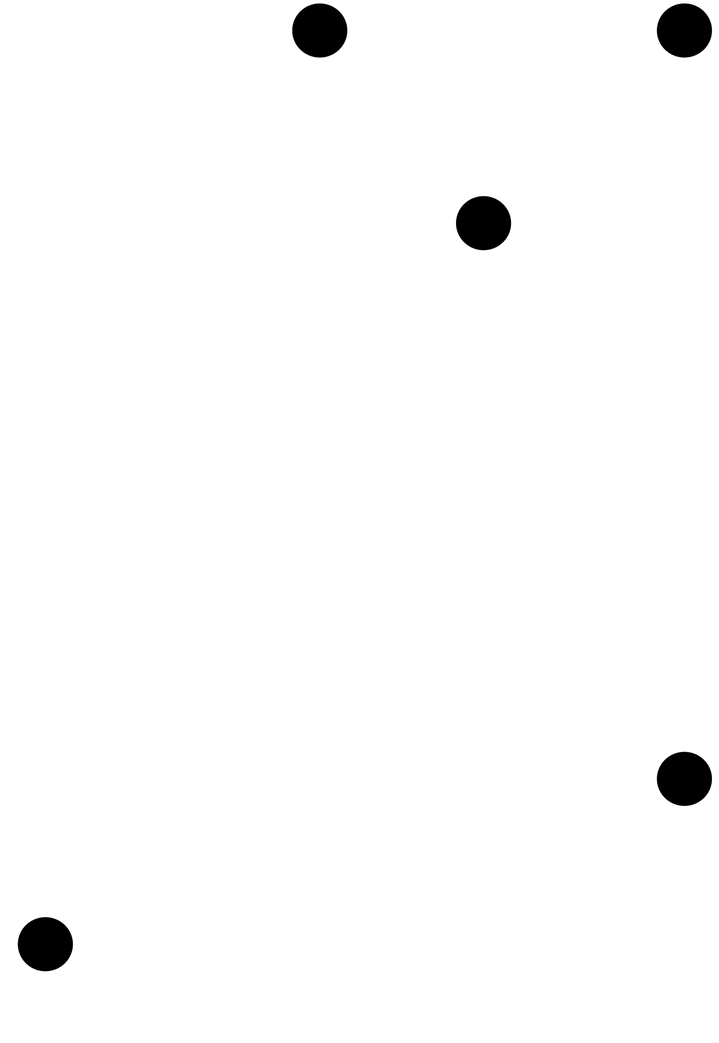
local repair

setting new points

Local Repair

Illustration





LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

setting new points

$(q_0, \ldots, q_h)$ is lower hull of $\{p_1, \ldots, p_i\}$

upper rand (right to left)

local repair

setting new points

Local Repair

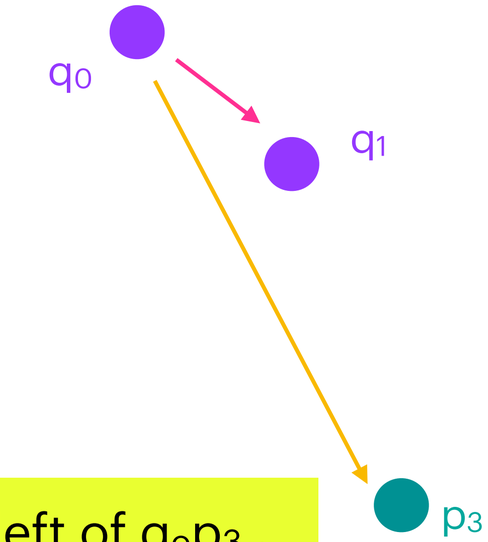
Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h                                   (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right) — local repair — setting new points

upper rand (right to left) — local repair — setting new points



q_1 on the left of q_0p_3 ,
remove q_1

Local Repair

Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

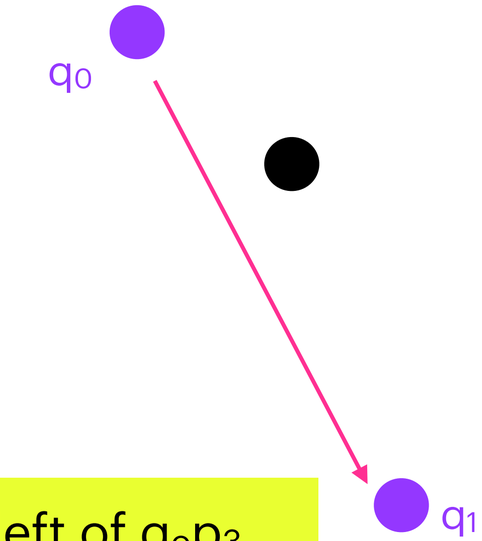
setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

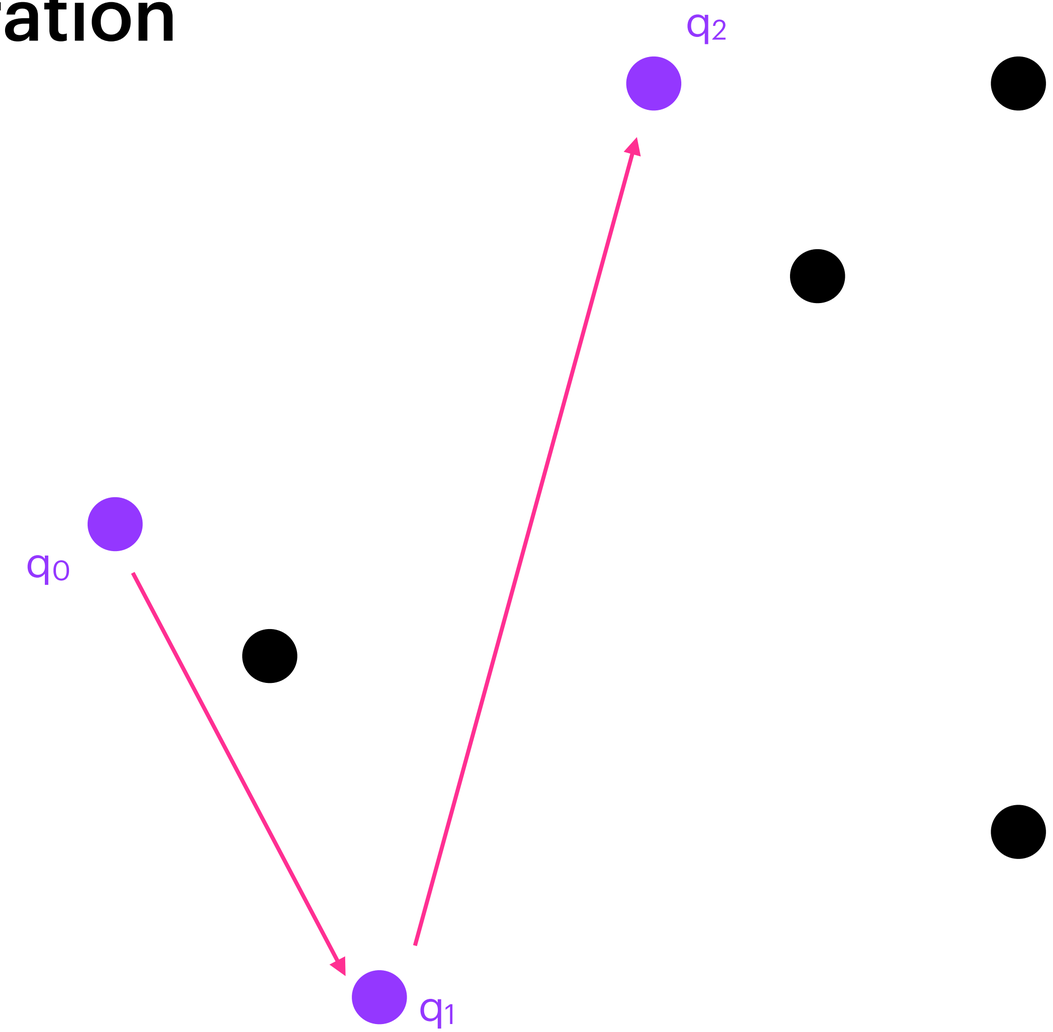
setting new points



q_1 on the left of q_0p_3 ,
remove q_1

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

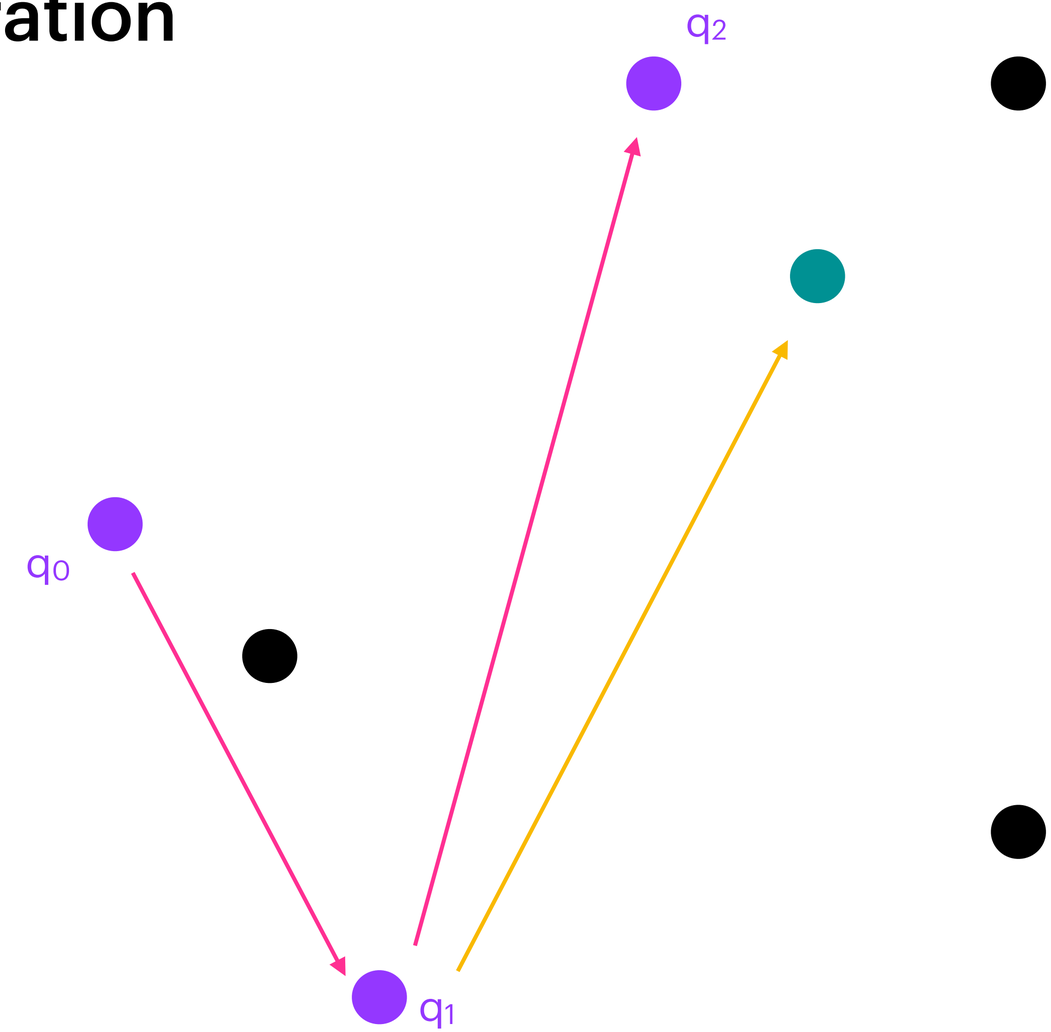
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

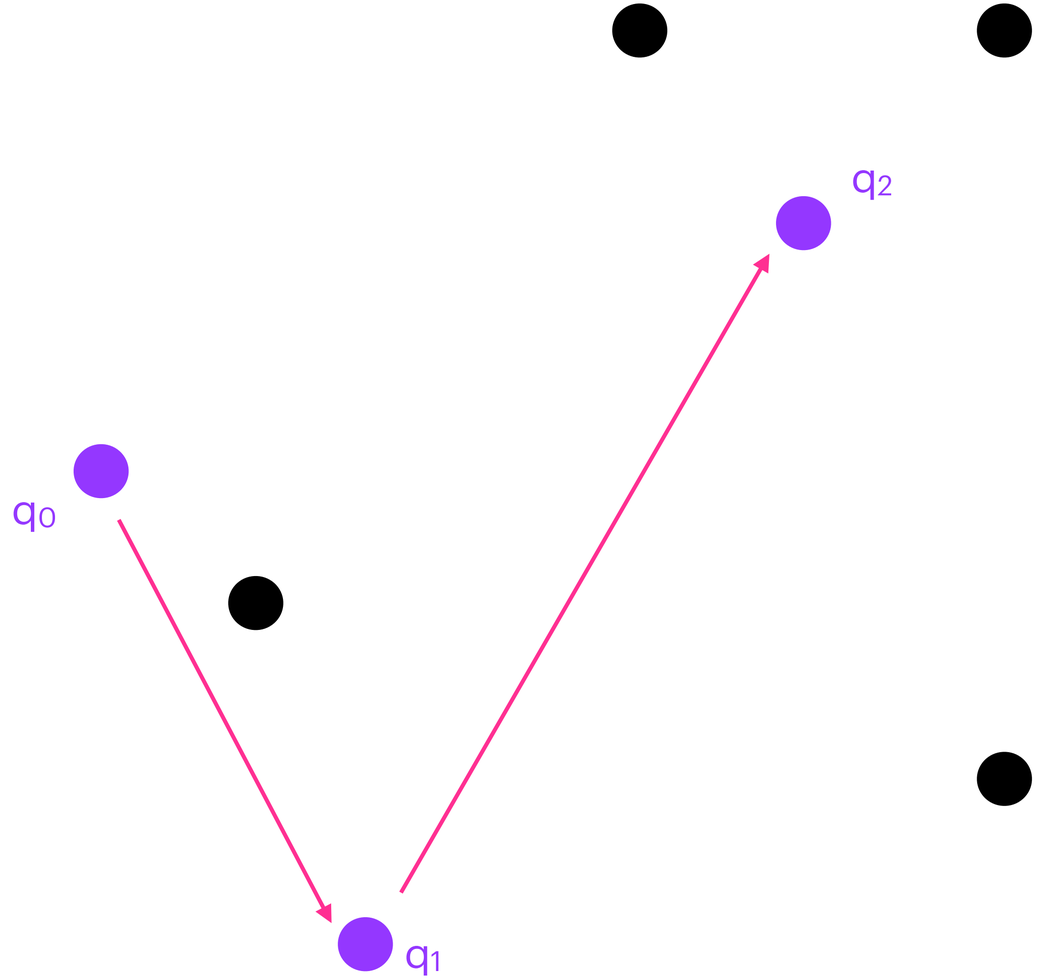
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

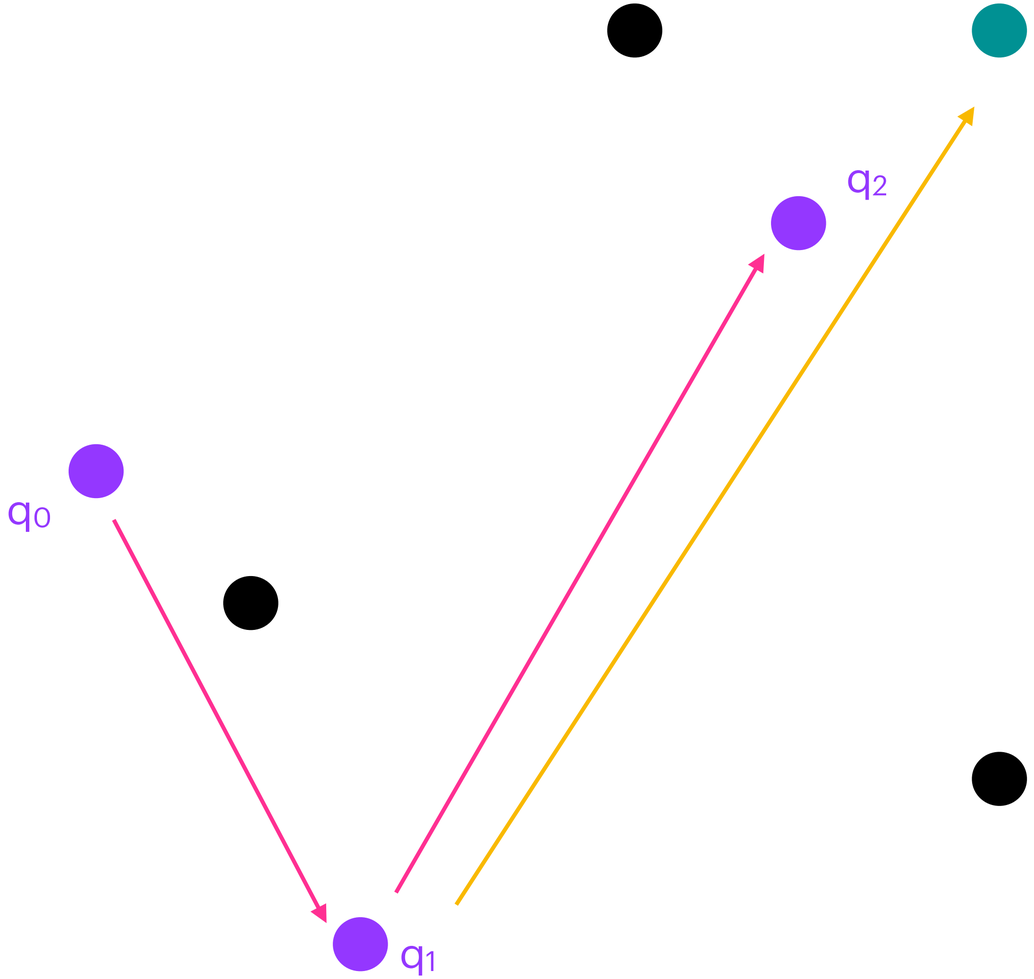
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



lower rand (left to right)

upper rand (right to left)

LocalRepair($p_1, p_2, \ldots, p_n$) — $(p_1, p_2, \ldots, p_n)$ sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

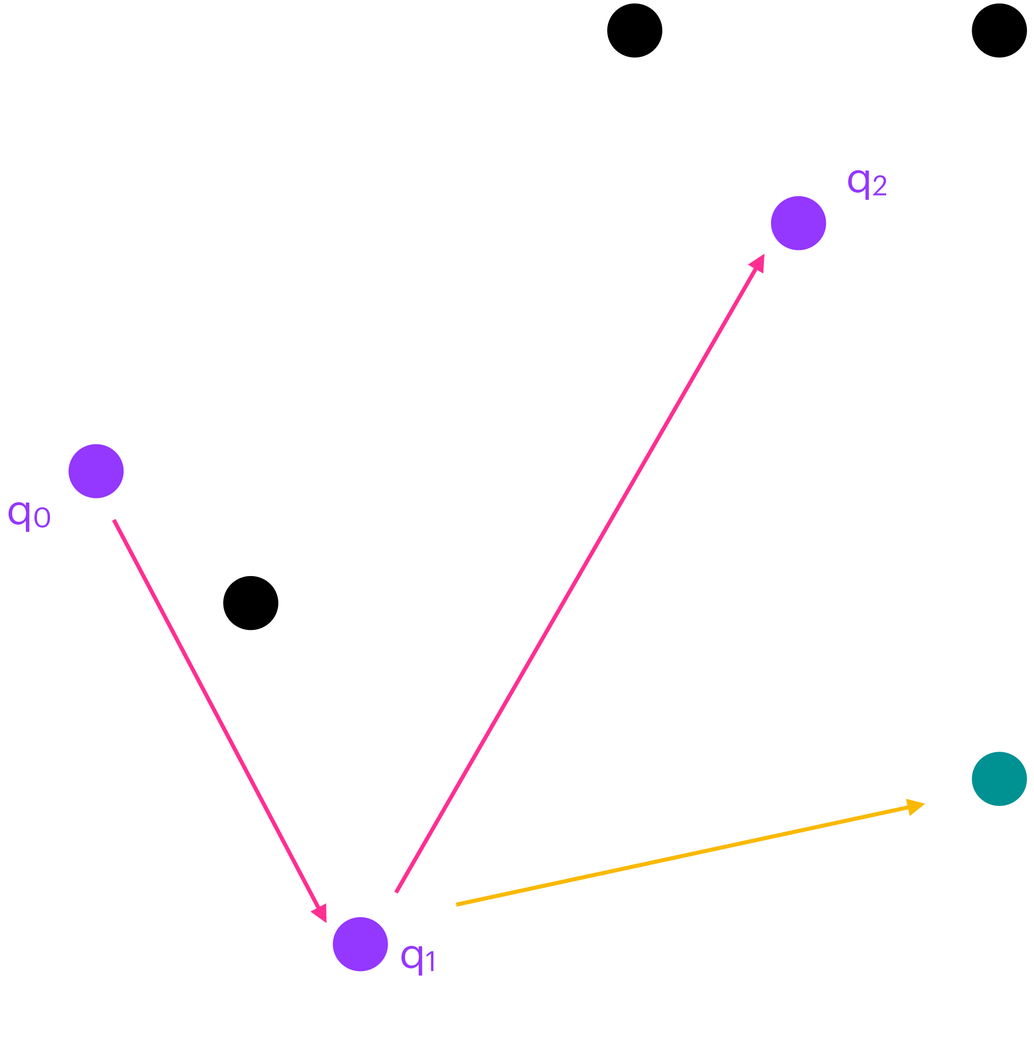
local repair
setting new points

$(q_0, \ldots, q_h)$ is lower hull of $\{p_1, \ldots, p_i\}$

local repair
setting new points

Local Repair

Illustration



lower rand (left to right)

upper rand (right to left)

LocalRepair($p_1, p_2, \ldots, p_n$) — $(p_1, p_2, \ldots, p_n)$ sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

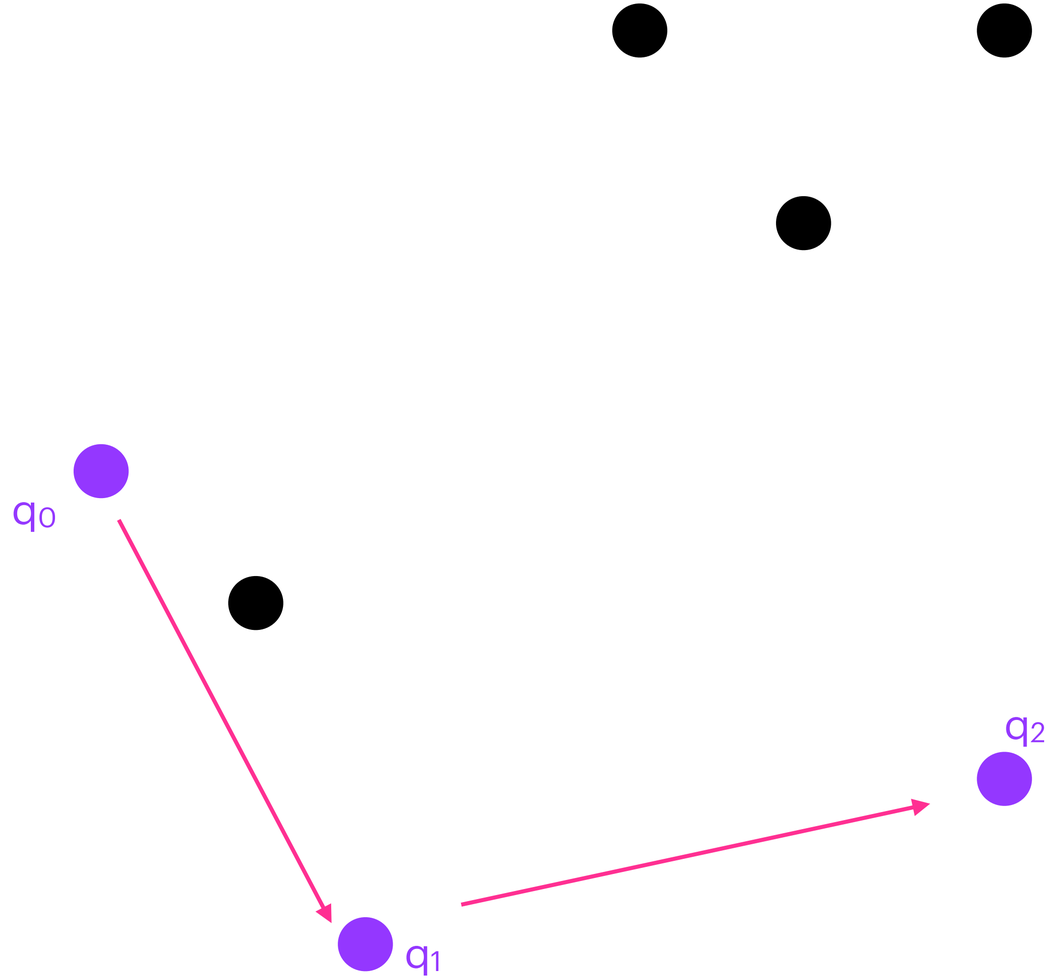
local repair
setting new points

$(q_0, \ldots, q_h)$ is lower hull of $\{p_1, \ldots, p_i\}$

local repair
setting new points

Local Repair

Illustration



lower rand (left to right)

upper rand (right to left)

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

local repair — setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

local repair — setting new points

Local Repair

Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

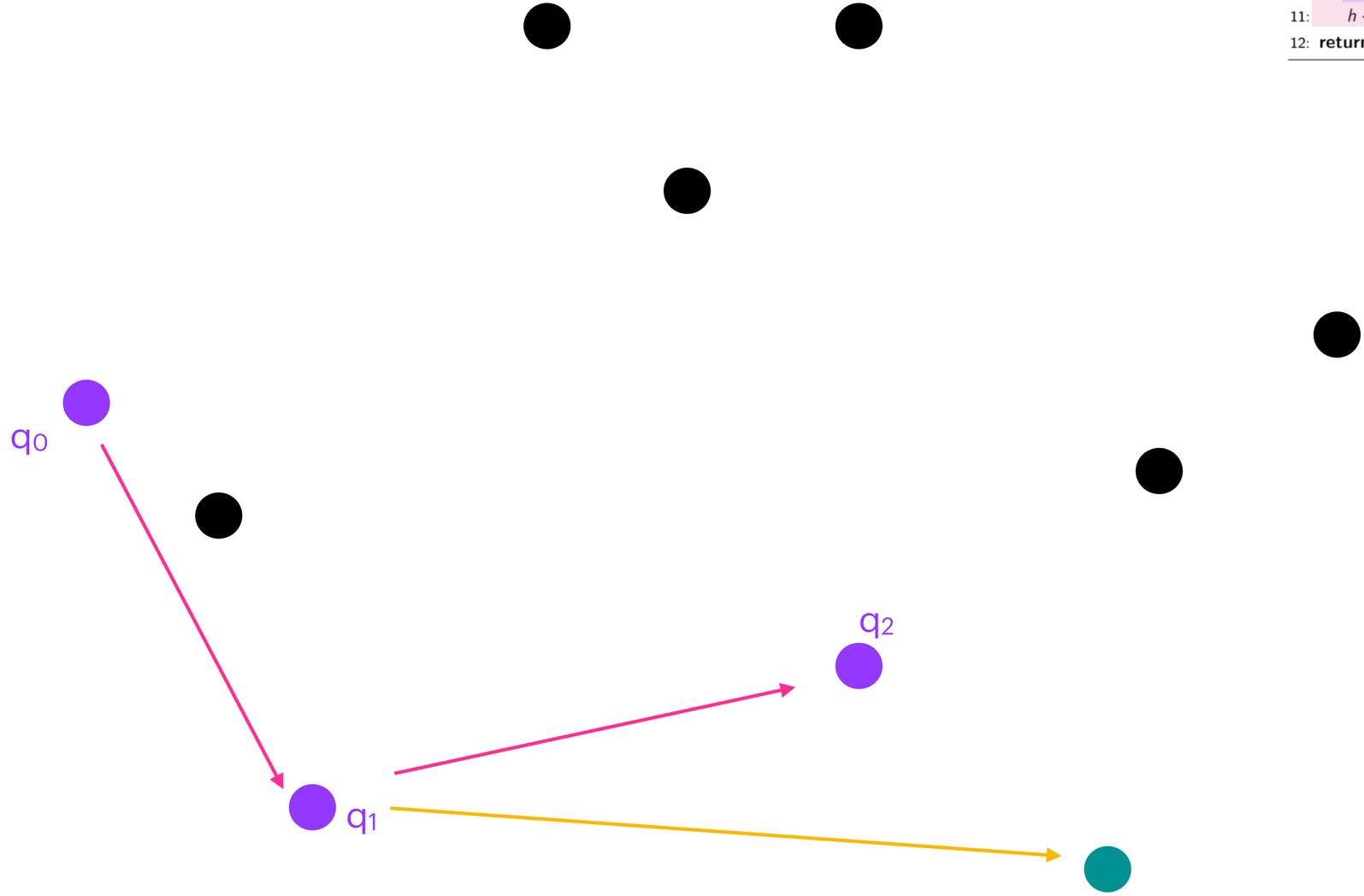
setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

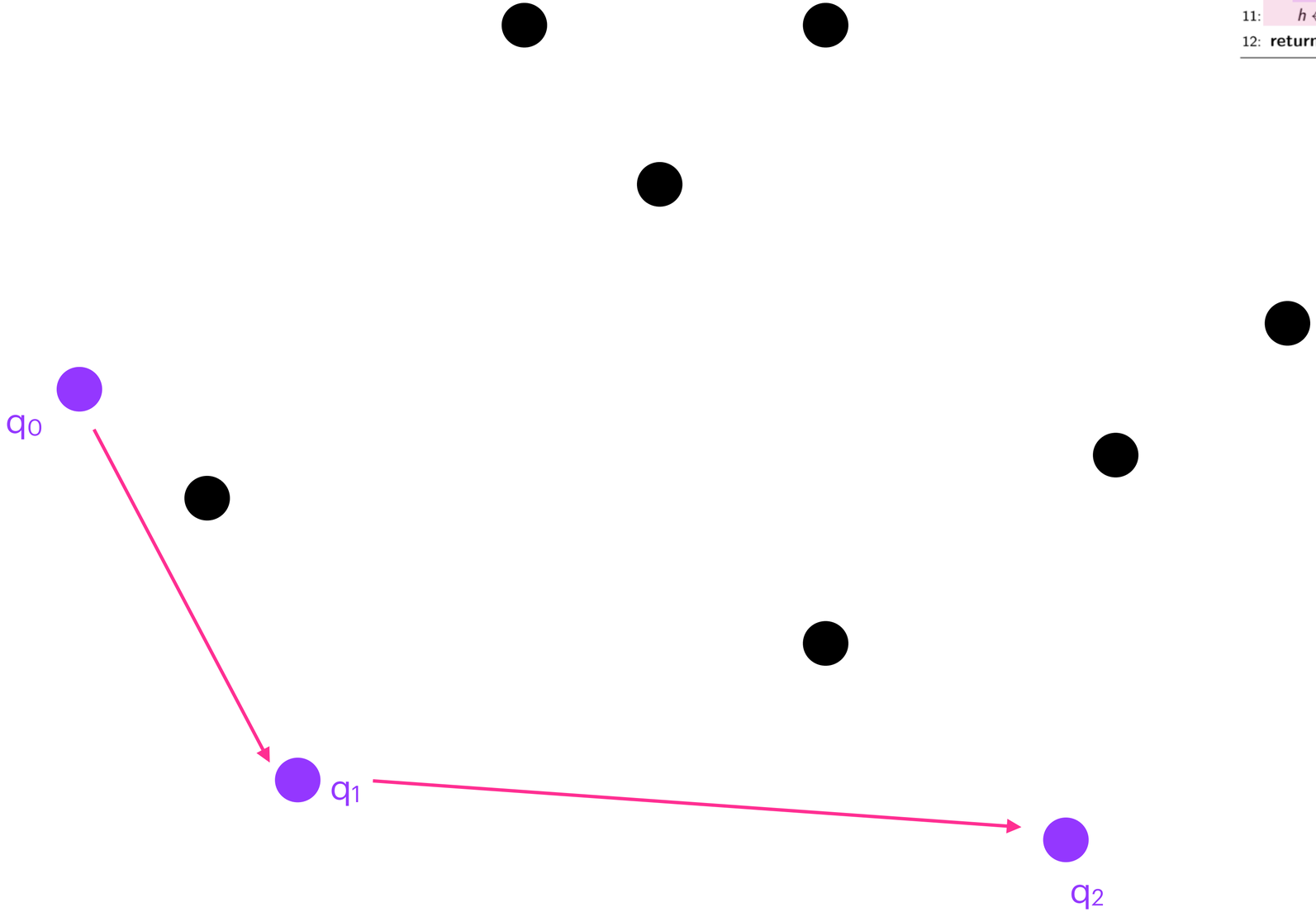
setting new points



Local Repair

Illustration

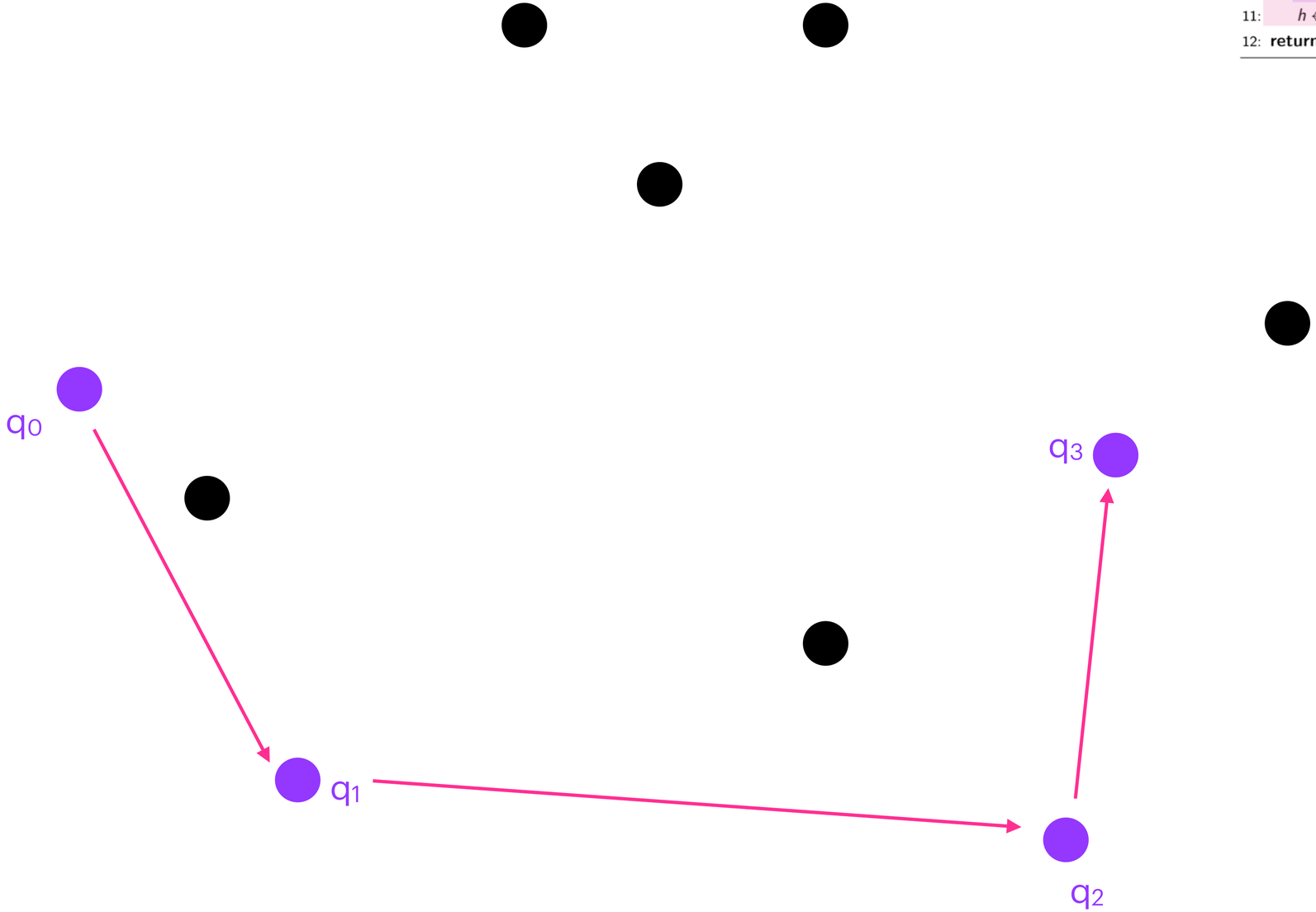
```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
 1: q_0 ← p_1; h ← 0
 2: for i ← 2 to n do                                    lower rand (left to right)
 3:     while h > 0 und q_h on the left of q_{h-1}p_i do  local repair
 4:         h ← h − 1
 5:     h ← h + 1; q_h ← p_i                             setting new points
 6:
 7: h' ← h                                               (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
 8: for i ← n − 1 downto 1 do                            upper rand (right to left)
 9:     while h > h' und q_h on the left of q_{h-1}p_i do local repair
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i                             setting new points
12: return (q_0, q_1, ..., q_{h-1})
```



Local Repair

Illustration

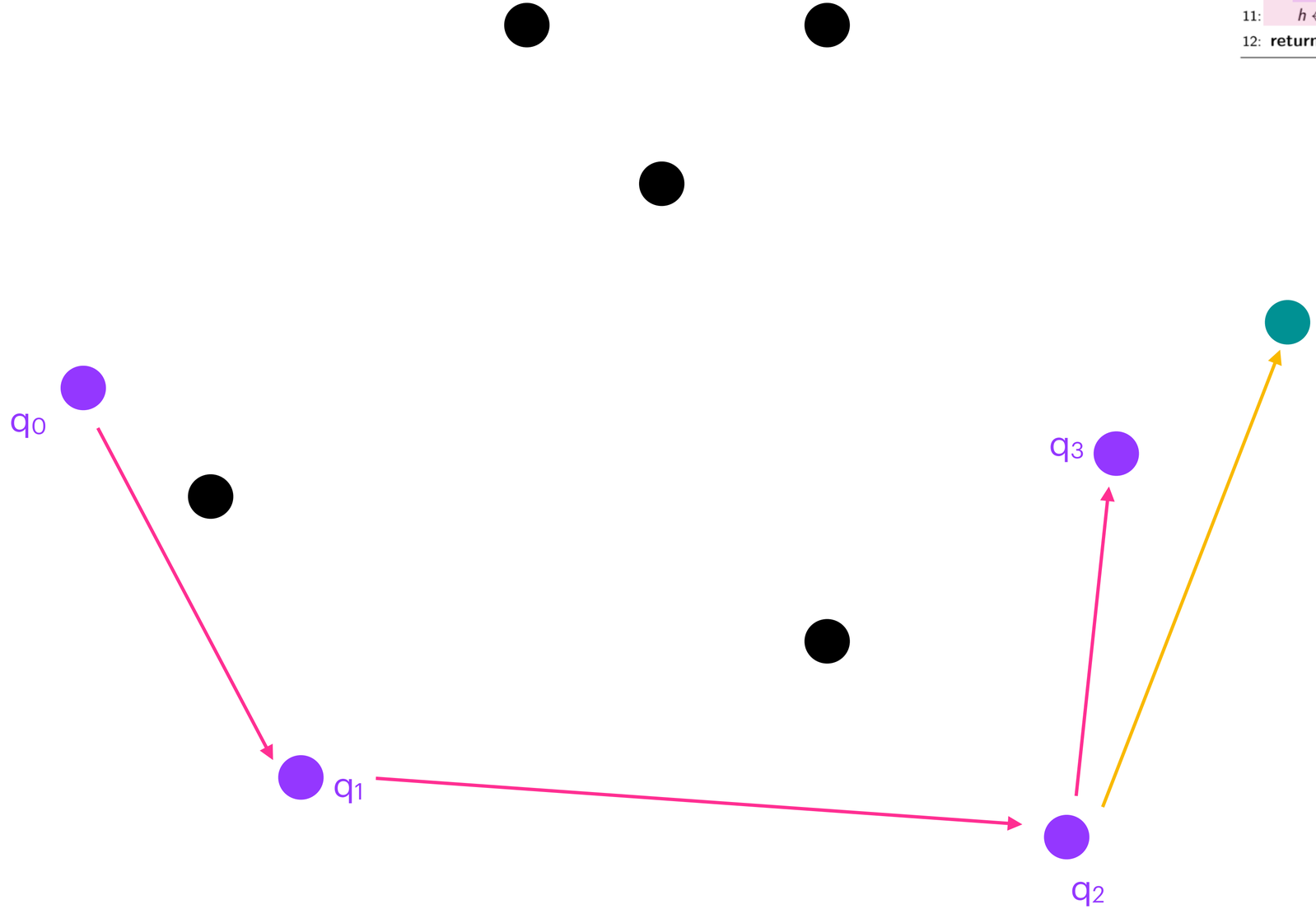
```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
 1: q_0 ← p_1; h ← 0
 2: for i ← 2 to n do                                    lower rand (left to right)
 3:     while h > 0 und q_h on the left of q_{h-1}p_i do
 4:         h ← h − 1                                    local repair
 5:     h ← h + 1; q_h ← p_i                             setting new points
 6:
 7: h' ← h                                               (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
 8: for i ← n − 1 downto 1 do                            upper rand (right to left)
 9:     while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1                                    local repair
11:     h ← h + 1; q_h ← p_i                             setting new points
12: return (q_0, q_1, ..., q_{h-1})
```



Local Repair

Illustration

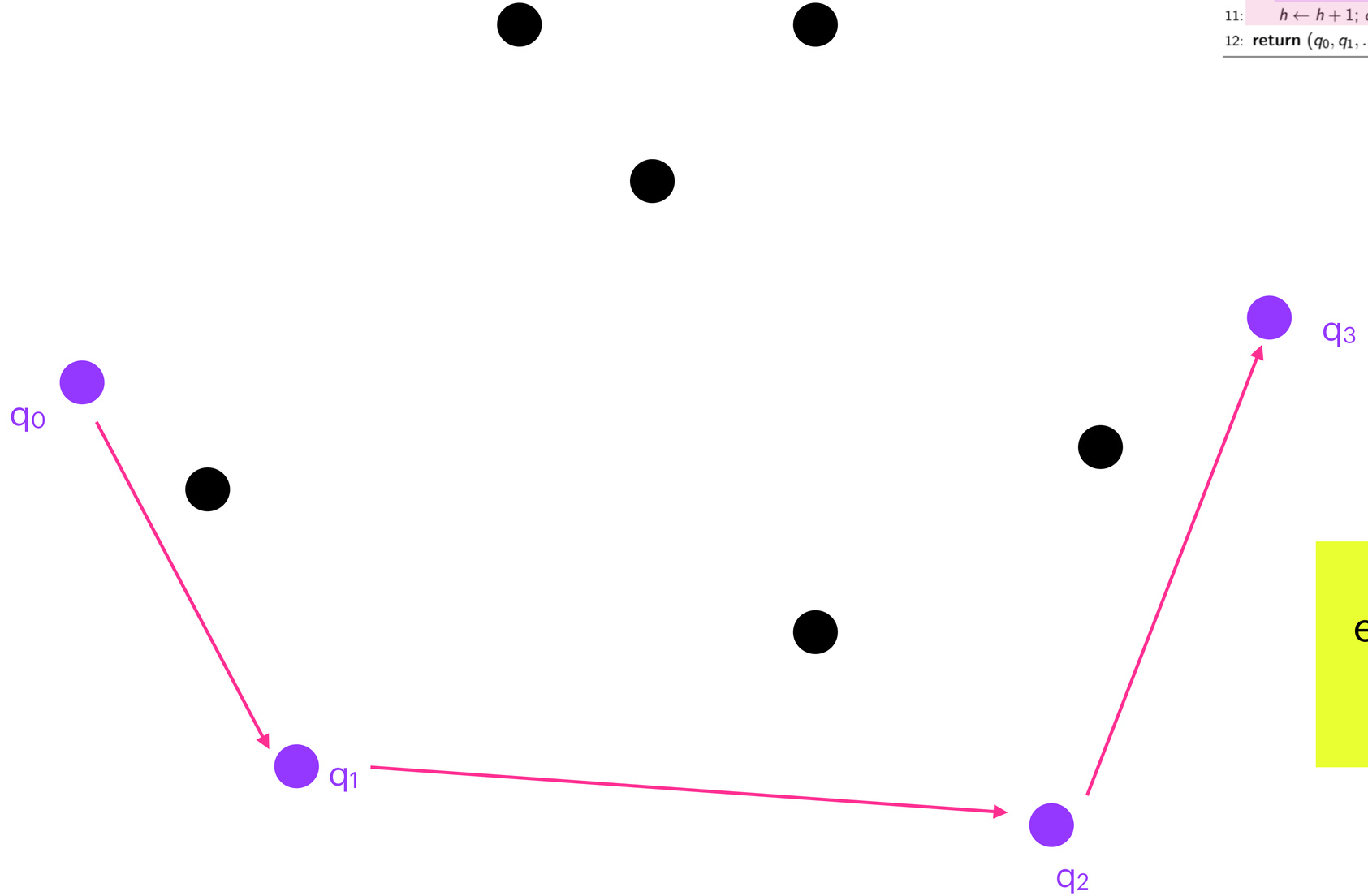
```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
1: q_0 ← p_1; h ← 0
2: for i ← 2 to n do                                                  lower rand
3:     while h > 0 und q_h on the left of q_{h-1}p_i do               (left to right)
4:         h ← h − 1                                                  local repair
5:     h ← h + 1; q_h ← p_i                                           setting new points
6:
7: h' ← h                                   (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8: for i ← n − 1 downto 1 do                                          upper rand
9:     while h > h' und q_h on the left of q_{h-1}p_i do              (right to left)
10:        h ← h − 1                                                  local repair
11:    h ← h + 1; q_h ← p_i                                           setting new points
12: return (q_0, q_1, ..., q_{h-1})
```



Local Repair

Illustration

```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do                                    lower rand (left to right)
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1                                    local repair
5:      h ← h + 1; q_h ← p_i                             setting new points
6:
7:  h' ← h                                               (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8:  for i ← n − 1 downto 1 do                            upper rand (right to left)
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1                                    local repair
11:     h ← h + 1; q_h ← p_i                             setting new points
12: return (q_0, q_1, ..., q_{h-1})
```

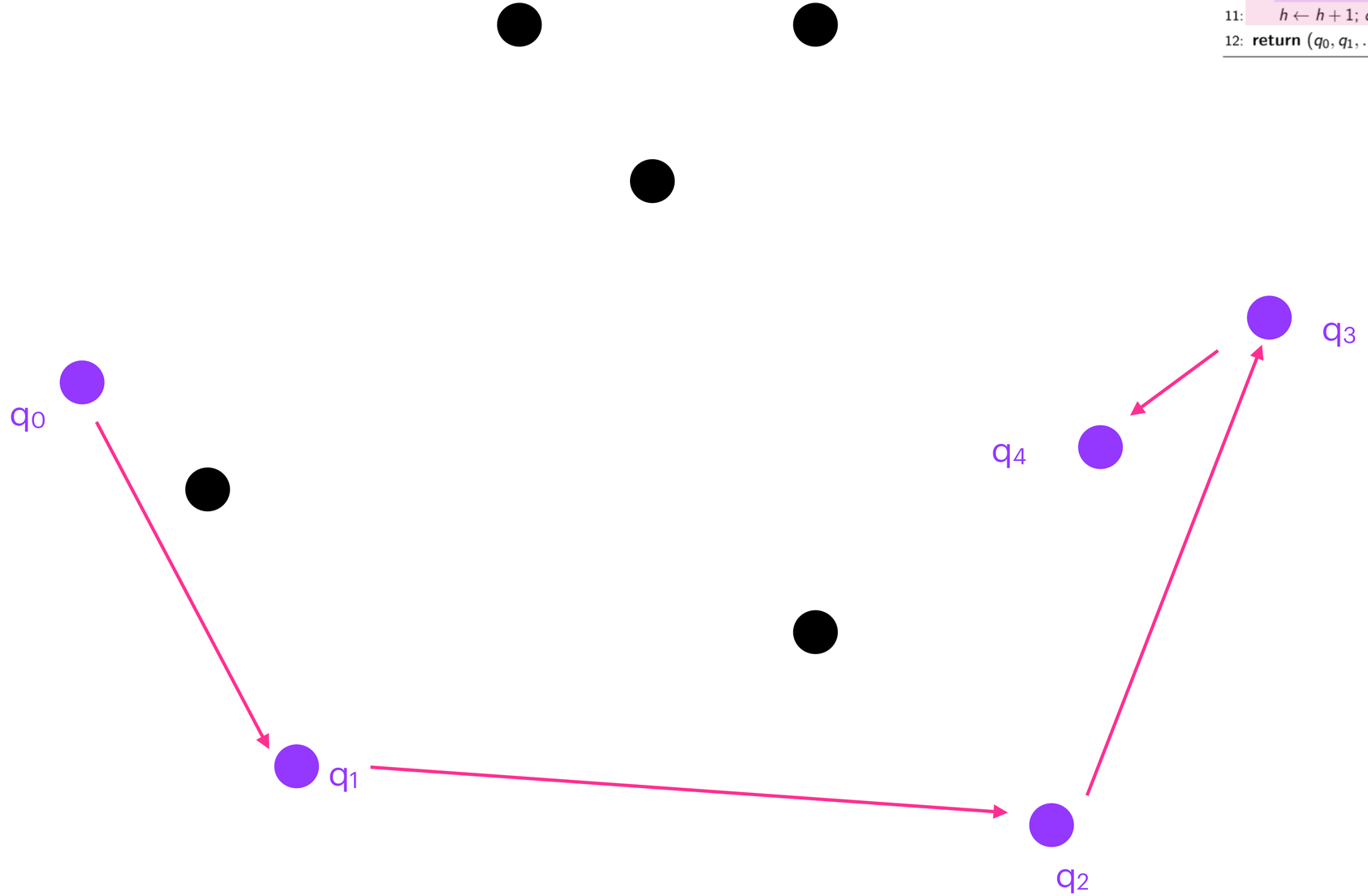


We've reached the endpoint. Let's do the same in the other direction

Local Repair

Illustration

```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do                                    lower rand (left to right)
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1                                    local repair
5:      h ← h + 1; q_h ← p_i                             setting new points
6:
7:  h' ← h                                               (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8:  for i ← n − 1 downto 1 do                            upper rand (right to left)
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1                                    local repair
11:     h ← h + 1; q_h ← p_i                             setting new points
12: return (q_0, q_1, ..., q_{h-1})
```



Local Repair

Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

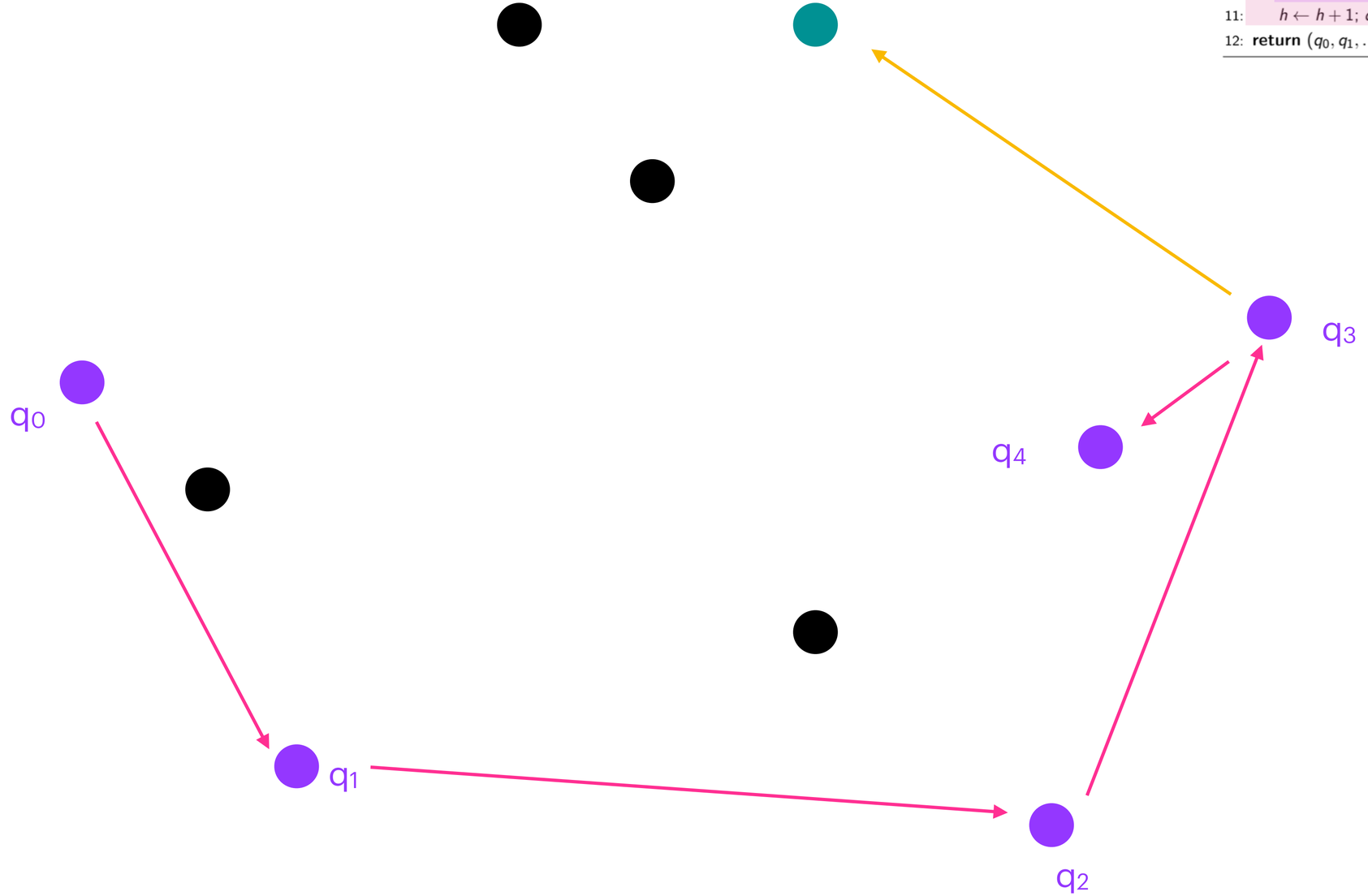
setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

setting new points



Local Repair

Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

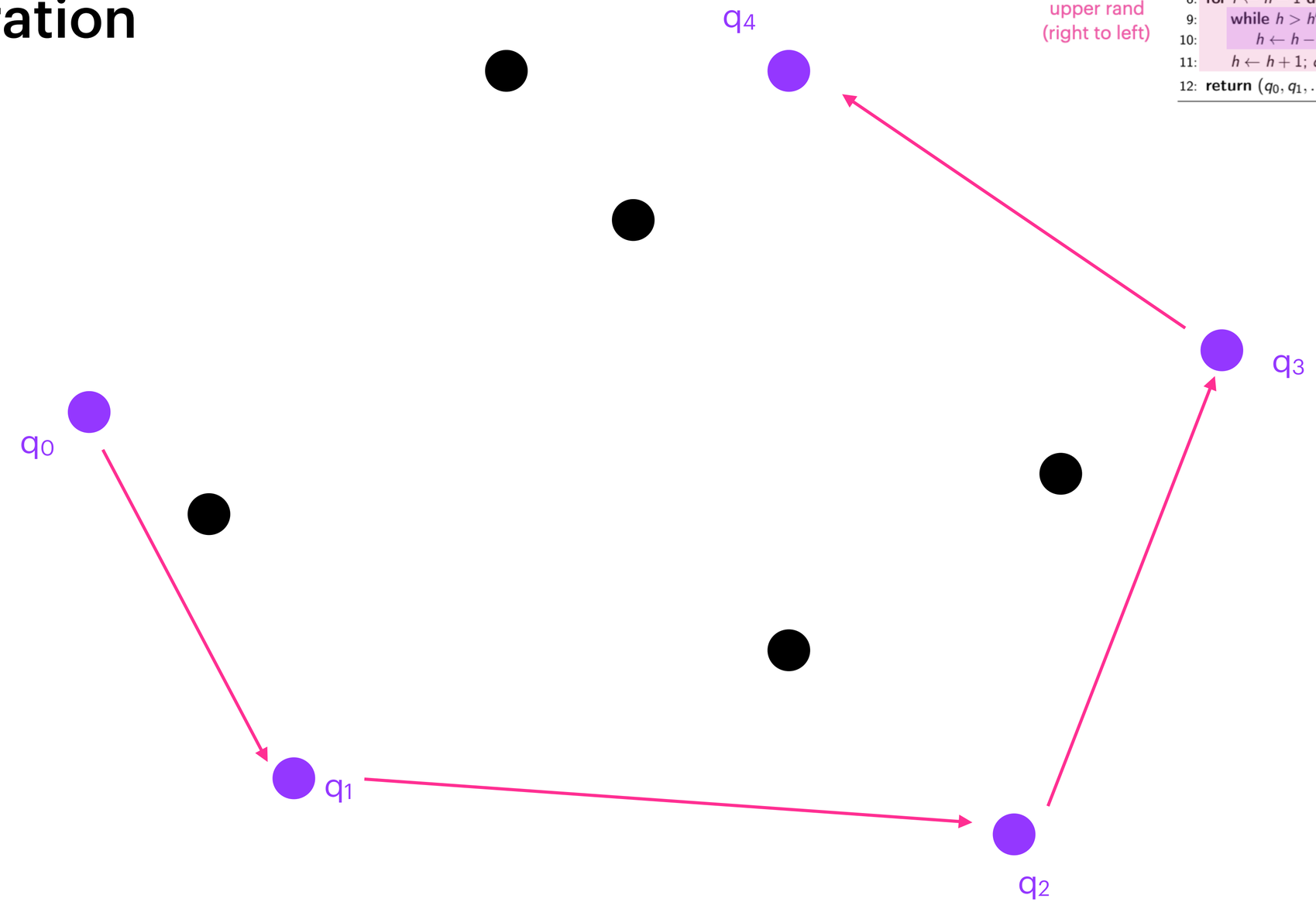
setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

setting new points



Local Repair

Illustration

LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

1: $q_0 \leftarrow p_1$; $h \leftarrow 0$
2: **for** $i \leftarrow 2$ **to** n **do**
3: **while** $h > 0$ und q_h on the left of $q_{h-1}p_i$ **do**
4: $h \leftarrow h - 1$
5: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
6:
7: $h' \leftarrow h$
8: **for** $i \leftarrow n - 1$ **downto** 1 **do**
9: **while** $h > h'$ und q_h on the left of $q_{h-1}p_i$ **do**
10: $h \leftarrow h - 1$
11: $h \leftarrow h + 1$; $q_h \leftarrow p_i$
12: **return** $(q_0, q_1, \ldots, q_{h-1})$

lower rand (left to right)

local repair

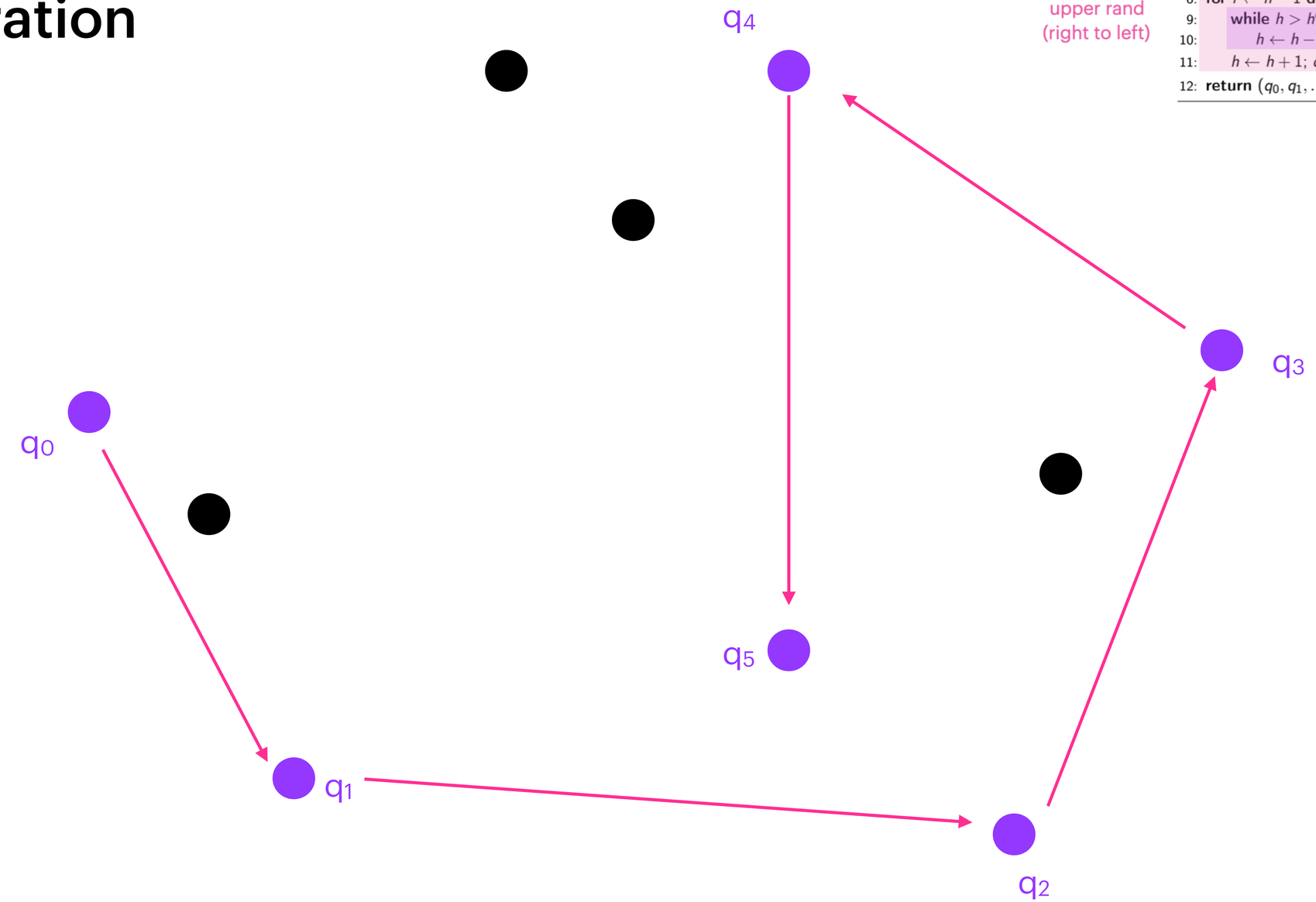
setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

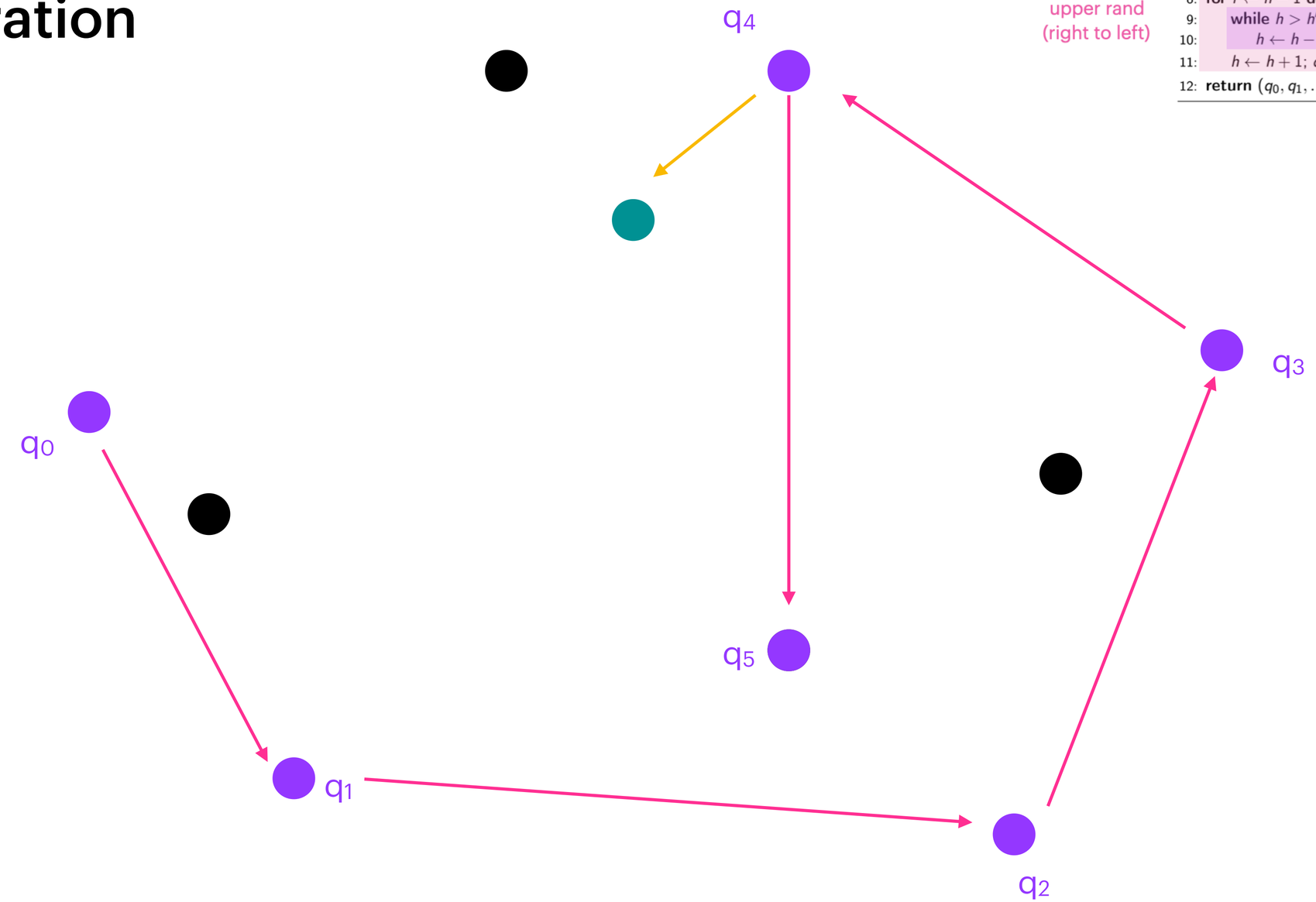
setting new points



Local Repair

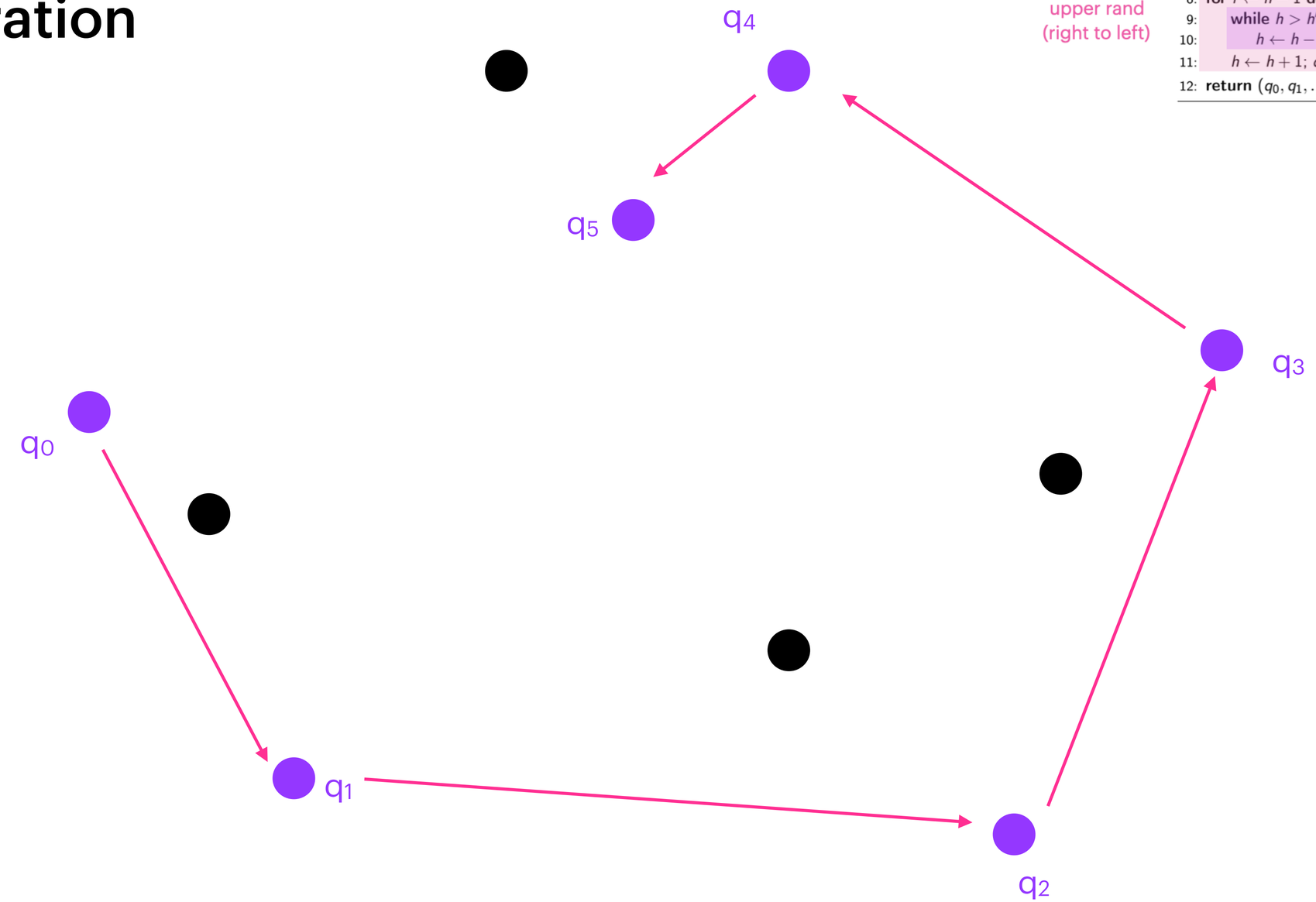
Illustration

```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do                                    lower rand (left to right)
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1                                    local repair
5:      h ← h + 1; q_h ← p_i                             setting new points
6:
7:  h' ← h                                               (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8:  for i ← n − 1 downto 1 do                            upper rand (right to left)
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1                                    local repair
11:     h ← h + 1; q_h ← p_i                             setting new points
12: return (q_0, q_1, ..., q_{h-1})
```



Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right) — local repair — setting new points

($q_0, \ldots, q_h$) is lower hull of {$p_1, \ldots, p_i$}

upper rand (right to left) — local repair — setting new points

Local Repair

Illustration

```
LocalRepair(p_1, p_2, ..., p_n)                          (p_1, p_2, ..., p_n) sorted
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h                                   (q_0, ..., q_h) is lower hull of {p_1, ..., p_i}
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

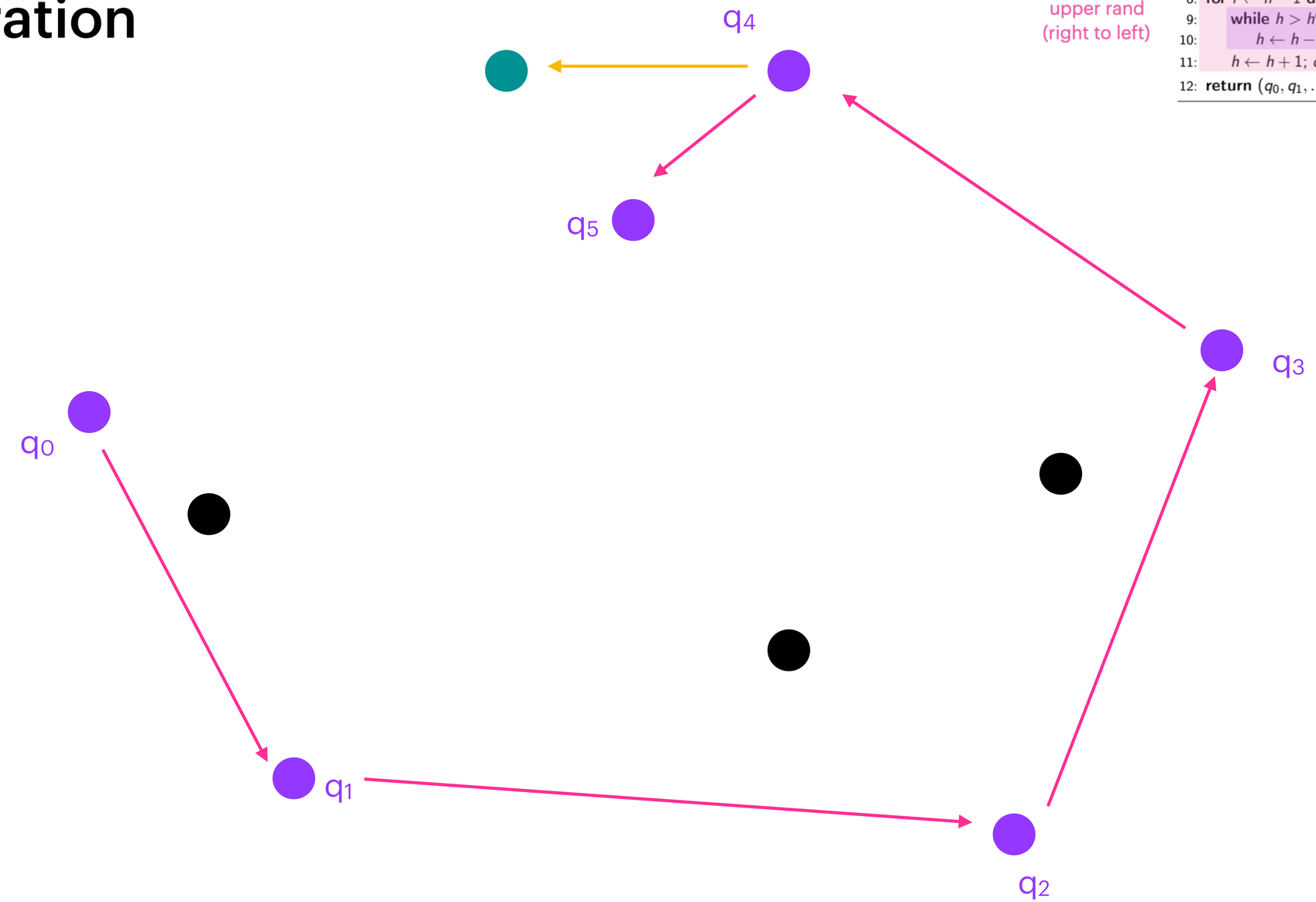
local repair

setting new points

upper rand (right to left)

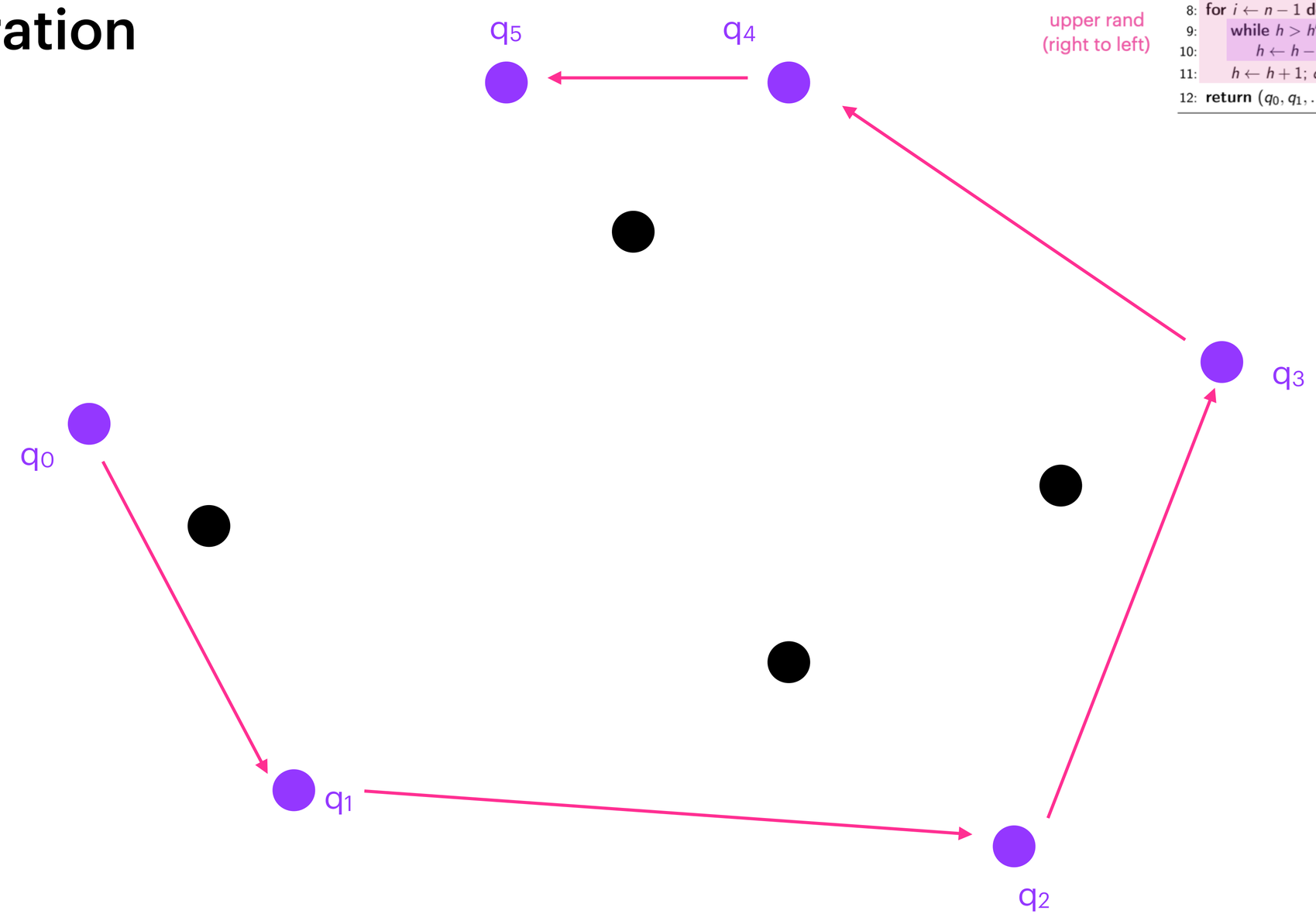
local repair

setting new points



Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

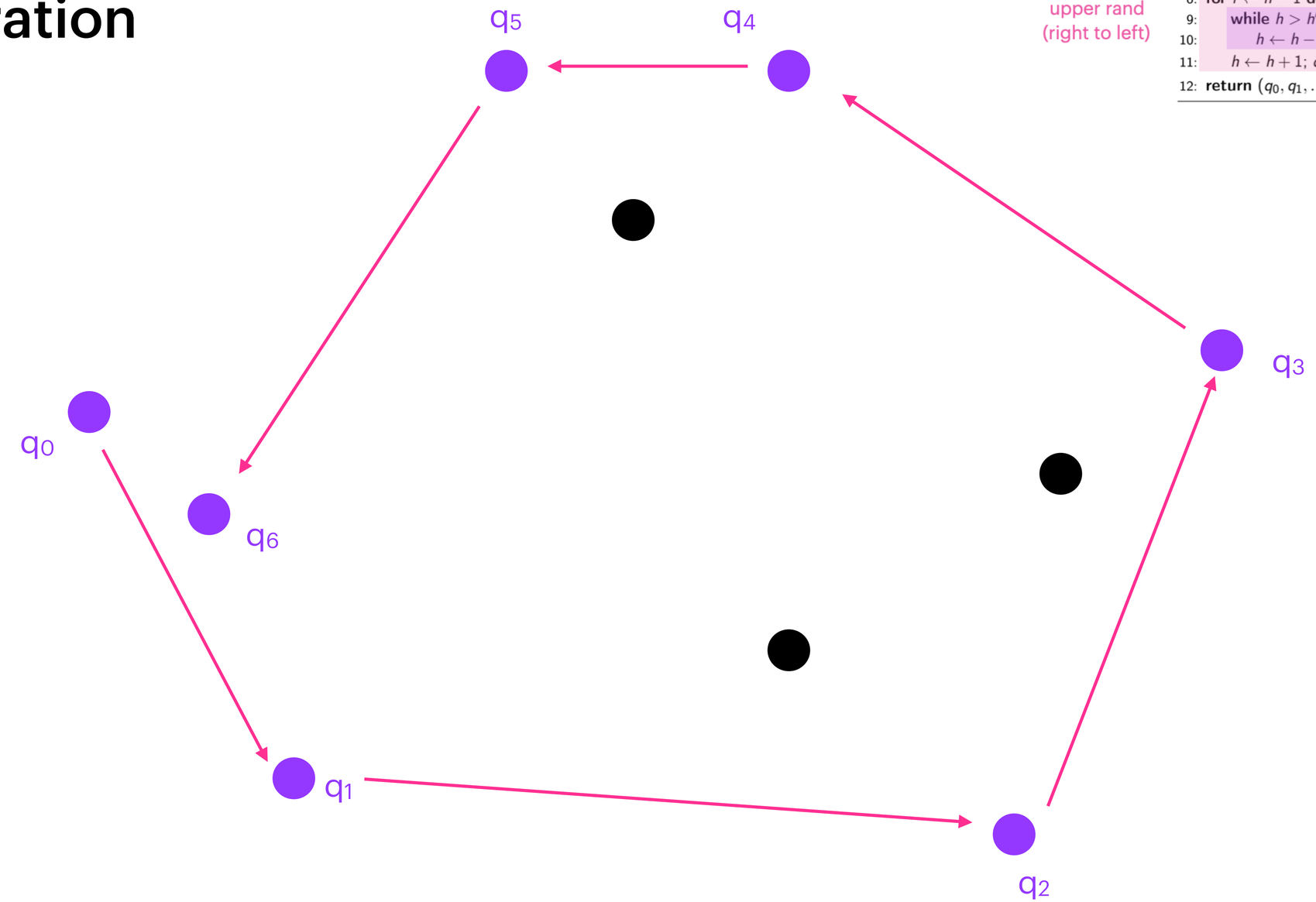
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

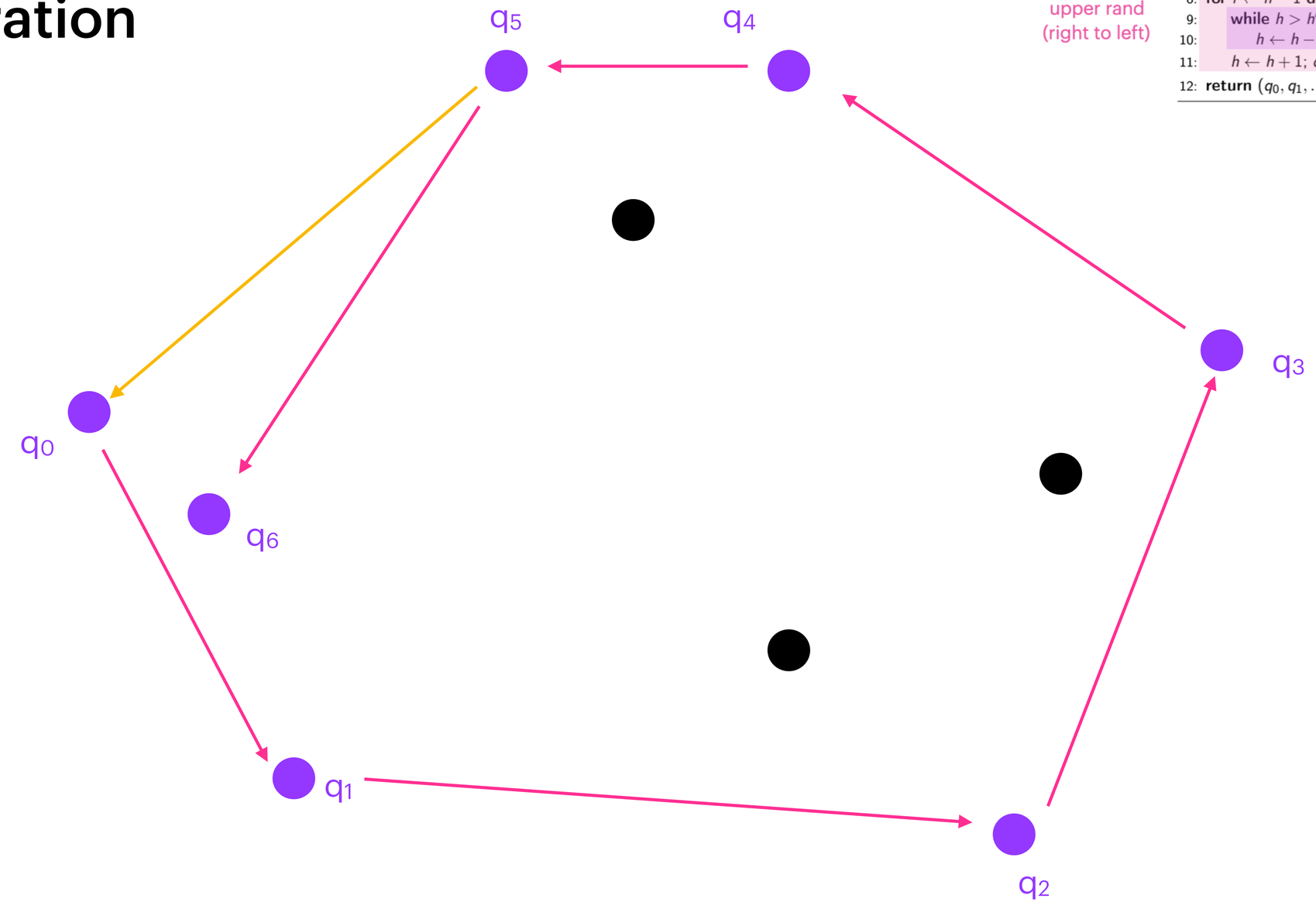
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

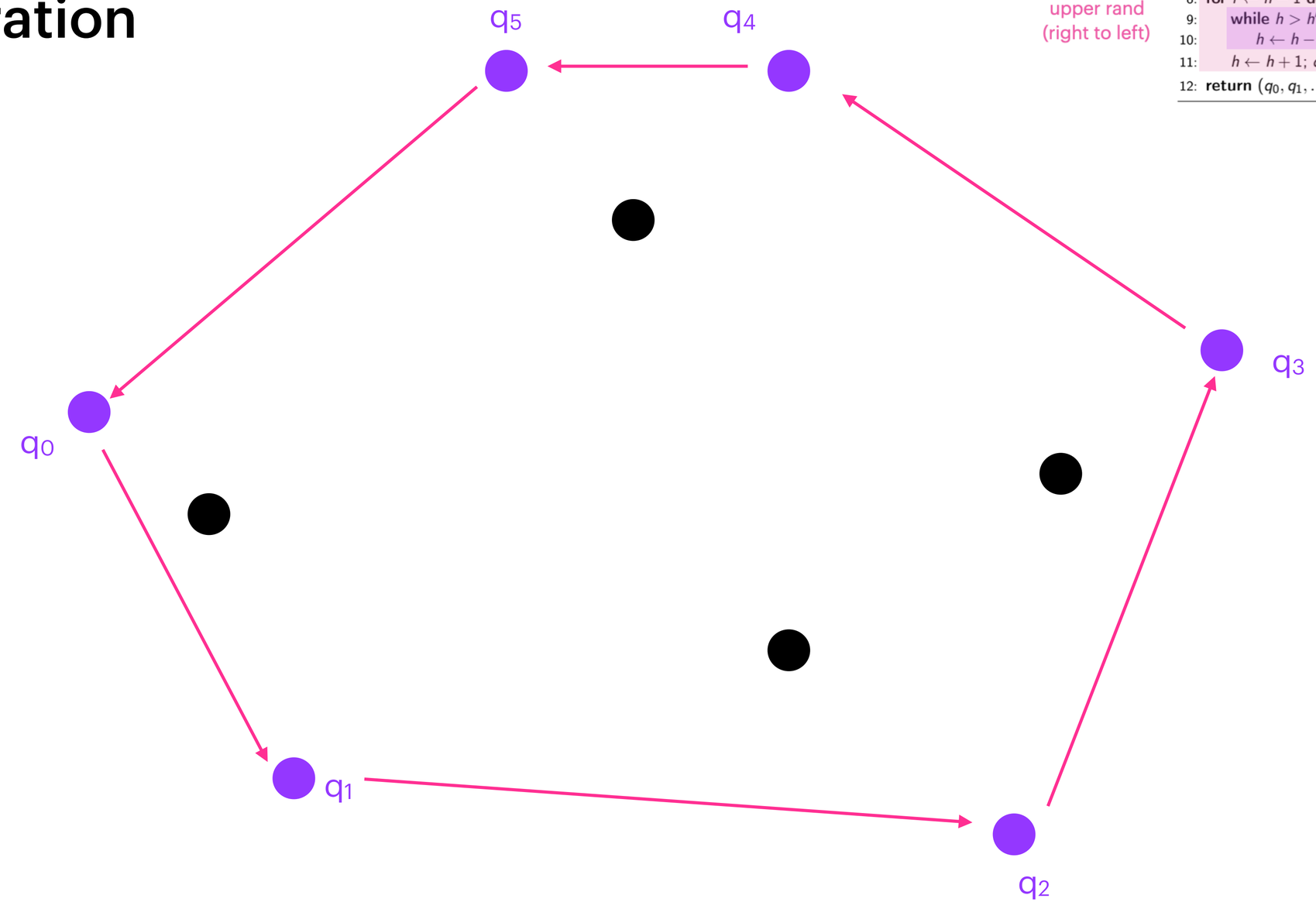
upper rand (right to left)

local repair

setting new points

Local Repair

Illustration



LocalRepair($p_1, p_2, \ldots, p_n$) — ($p_1, p_2, \ldots, p_n$) sorted

```
1:  q_0 ← p_1; h ← 0
2:  for i ← 2 to n do
3:      while h > 0 und q_h on the left of q_{h-1}p_i do
4:          h ← h − 1
5:      h ← h + 1; q_h ← p_i
6:
7:  h' ← h
8:  for i ← n − 1 downto 1 do
9:      while h > h' und q_h on the left of q_{h-1}p_i do
10:         h ← h − 1
11:     h ← h + 1; q_h ← p_i
12: return (q_0, q_1, ..., q_{h-1})
```

lower rand (left to right)

local repair

setting new points

(q_0, ..., q_h) is lower hull of {p_1, ..., p_i}

upper rand (right to left)

local repair

setting new points

Primality Tests II

Primality Test

Problem Description

given : A number $n \in \mathbb{N}$

to find : is n prime ? ?

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, ...

$$\pi(11) =$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$:

$$\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, \, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$$

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, ...

$$\pi(11) = 5$$

Primality Test

Problem Description

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	...
$f(n)$	0	1	0	1	2	1	0	4	2	1	8	1	2	4	0	1	14	1	8	4	2	1	...

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left| \{ n \in \mathbb{N} \mid n \leq x, n \text{ prime} \} \right| \sim \frac{x}{\ln x}$

Primality Test

Naive Algorithm

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

1) For all $a \leq \sqrt{n}$ test if a divides n

Primality Test

Easy randomized test

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

1) Choose $a \in \{1,2, \ldots, \sqrt{n}\}$ uniformly at random

2) if a divides n then return ‘not prime’

3) else return ‘prime’

Refresher

DiskMat 🤢

gcd : greatest common divisor

$$n \text{ is prime} \Rightarrow \gcd(a,n) = 1 \quad \forall a \in [1,n-1]$$

$\mathbb{Z}_n^*$: the multiplicative group modulo n

$$\mathbb{Z}_n^* = \{a \in \{1,2,\ldots,n-1\} \mid \gcd(a,n) = 1\}$$

Primality Test

Euclidean Primality Test

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

n is prime $\Rightarrow \gcd(a,n) = 1 \quad \forall a \in [1, n-1]$

gcd := greatest common divisor

can be calculated in $O((\log nm)^3)$

1) Choose $a \in \{1, 2, \ldots, \sqrt{n}\}$ uniformly at random

2) if $gcd(a,n) > 1$ then return 'not prime'

3) else return 'prime'

- if n is a prime : always correct
- if n is not a prime : it might return a wrong answer with the probability

$$\frac{\left|\{a \in [1, n-1] : \gcd(a,n) = 1\}\right|}{n-1} = \frac{|\mathbb{Z}_n^*|}{n-1}$$

Refresher

DiskMat 🤢

A number n is a **Carmichael number** if:

1. n is **composite**.
2. n is **square-free** (no repeated prime factors).
3. For every prime p dividing n, it holds that:

$$p - 1 \mid n - 1$$

Let's check this for $561 = 3 \cdot 11 \cdot 17$:

- $3 - 1 = 2$, and $2 \mid 560$ ✓
- $11 - 1 = 10$, and $10 \mid 560$ ✓
- $17 - 1 = 16$, and $16 \mid 560$ ✓

✅ So 561 meets all conditions → it **is** a Carmichael number.

Fermat's Little Theorem

$$n \in \mathbb{N} \text{ is prime} \implies \text{For all } a \text{ with } 0 < a < n \quad a^{n-1} \equiv 1 \pmod{n}$$

$$\text{There exists } a \text{ with } 0 < a < n \quad a^{n-1} \not\equiv 1 \pmod{n} \implies n \text{ is not prime}$$

Carmichael Numbers

it gives false positive, "probably prime"

$a^{n-1} \equiv 1 \pmod{n}$ holds for all a coprime to n for the carmichael number, FLT fails to detect

ex:

$561 = 3 \cdot 11 \cdot 17$ for every a s.t. $\gcd(a, 561) = 1$, it holds that $a^{560} \equiv 1 \pmod{561}$

Primality Test

Miller-Rabin Test

given : A number $n \in \mathbb{N}$

to find : is n prime $\iff$ n has no divider in $\{2, \ldots, n-1\}$

prime-counting function $\pi(x)$: $\pi(x) := \left|\{n \in \mathbb{N} \mid n \leq x, n \text{ prime}\}\right| \sim \frac{x}{\ln x}$

MILLER-RABIN-PRIMZAHLTEST(n)

```
1:  if n = 2 then
2:      return 'Primzahl'
3:  else if n gerade oder n = 1 then
4:      return 'keine Primzahl'
5:  Wähle a ∈ {2, 3, ..., n − 1} zufällig und
6:  berechne k, d ∈ ℤ mit n − 1 = d2^k und d ungerade.
7:  x ← a^d (mod n)
8:  if x = 1 or x = n − 1 then
9:      return 'Primzahl'
10: repeat k − 1 mal
11:     x ← x^2 (mod n)
12:     if x = 1 then
13:         return 'keine Primzahl'
14:     if x = n − 1 then
15:         return 'Primzahl'
16: return 'keine Primzahl'
```

- if n is a prime : always correct
- if n is composite : it returns “not prime” with the probability $\geq 3/4$

Questions

Feedbacks , Recommendations

Nil Ozer