Algorithms and Complexity (AC)

Fourth Lecture (2nd of October, 2023)

Paul Bouman & Tom van der Zanden (Based on slides by Gerhard Woeginger, Jesper Nederlof and Marie Schmidt)

September – November 2023

LNMB – Landelijk Netwerk Mathematische Besliskunde

Table of contents

1. Basic definitions

2. Ad-hoc approaches

   Vertex cover

   Makespan minimization

   Intermezzo: Euler tours

   Travelling Salesman

3. Linear Programming-based approaches

   Weighted vertex cover

Basic definitions

We leave decision problems, and turn to optimization problems

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\text{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\text{opt}(I)$.

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\mathrm{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\mathrm{opt}(I)$.

- approximation ratio for minimization problem always ≥ 1

small approximation ratio = good

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\text{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\text{opt}(I)$.

- approximation ratio for minimization problem always ≥ 1

small approximation ratio = good

For maximization problems

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\text{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\text{opt}(I)$.

- approximation ratio for minimization problem always ≥ 1

small approximation ratio = good

For maximization problems approximation ratio is $\inf_I A(I)/\text{opt}(I)$

always ≤ 1; large guarantee = good

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\mathrm{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\mathrm{opt}(I)$.

- approximation ratio for minimization problem always ≥ 1

small approximation ratio = good

For maximization problems approximation ratio is $\inf_I A(I)/\mathrm{opt}(I)$

always ≤ 1; large guarantee = good

We aim for polynomial time algorithms with good approximation ratios

Basic definitions

We leave decision problems, and turn to optimization problems

Definition

Let X be a minimization problem.

The optimal objective value of instance I is denoted $\text{opt}(I)$.

The objective value returned by algorithm A is denoted $A(I)$.

The *(worst-case) approximation ratio* of algorithm A is $\sup_I A(I)/\text{opt}(I)$.

- approximation ratio for minimization problem always ≥ 1

small approximation ratio = good

For maximization problems approximation ratio is $\inf_I A(I)/\text{opt}(I)$

always ≤ 1; large guarantee = good

We aim for polynomial time algorithms with good approximation ratios
This is still possible for many problems whose decision versions are NP-complete!

Questions?

Ad-hoc approaches

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$

Goal: find a vertex cover of smallest possible size

(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$
Goal: find a vertex cover of smallest possible size
(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)

Vertex Cover

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$
Goal: find a vertex cover of smallest possible size
(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)
Say M is maximal if there is no matching M' such that $M \subset M'$.

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$

Goal: find a vertex cover of smallest possible size

(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)

Say M is maximal if there is no matching M' such that $M \subset M'$.

Approximation algorithm for Vertex Cover

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$
Goal: find a vertex cover of smallest possible size
(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)
Say M is maximal if there is no matching M' such that $M \subset M'$.

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$
Goal: find a vertex cover of smallest possible size
(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)
Say M is maximal if there is no matching M' such that $M \subset M'$.

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Vertex cover (VC)

Instance: undirected graph $G = (V, E)$
Goal: find a vertex cover of smallest possible size
(vertex cover = subset of vertices that touches every edge)

How can we find a 'good' vertex cover?

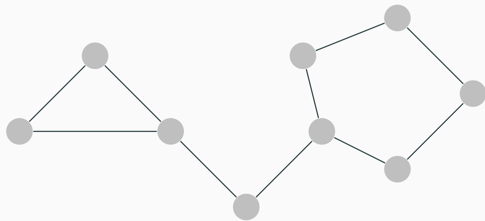
Matching

Subset $M \subseteq E$ of disjoint edges (that is $e \cap e' = \emptyset$, for distinct $e, e' \in M$)
Say M is maximal if there is no matching M' such that $M \subset M'$.

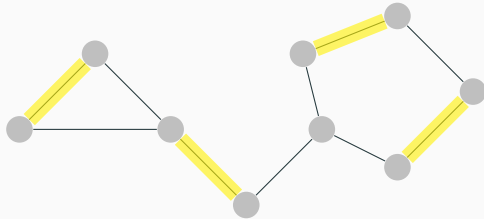
Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

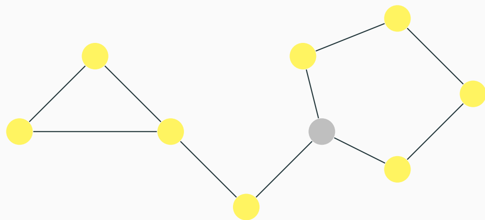
Vertex Cover



Vertex Cover



Vertex Cover



Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Why does this give a vertex cover?

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Why does this give a vertex cover?

Assume that there was an edge $\{u, v\}$ not covered by S.

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Why does this give a vertex cover?

Assume that there was an edge $\{u, v\}$ not covered by S.
That is: there was no edge adjacent to u or to v in M.

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Why does this give a vertex cover?

Assume that there was an edge $\{u, v\}$ not covered by S.
That is: there was no edge adjacent to u or to v in M.
$M \cup \{u, v\}$ is a matching which contains M.

Approximation algorithm for Vertex Cover

1. Find a maximal matching M (iteratively pick edges not yet touched)
2. Output $S = \bigcup_{\{u,v\} \in M} \{u, v\}$ (e.g. all endpoint of M)

Why does this give a vertex cover?

Assume that there was an edge $\{u, v\}$ not covered by S.
That is: there was no edge adjacent to u or to v in M.
$M \cup \{u, v\}$ is a matching which contains M.
$\Rightarrow M$ was not maximal.

Is this indeed an approximation algorithm?

I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?

What would be the approximation ratio?

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:

Approximation algorithm for Vertex Cover

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.
For each edge in M, any vertex cover S contains at least one vertex.

Approximation algorithm for Vertex Cover

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.
For each edge in M, any vertex cover S contains at least one vertex. Thus $opt(I) \geq |M|$.

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.
For each edge in M, any vertex cover S contains at least one vertex. Thus $opt(I) \geq |M|$.
The vertex cover constructed with the approximation algorithm A has size $A(I) = 2|M|$.

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.
For each edge in M, any vertex cover S contains at least one vertex. Thus $opt(I) \geq |M|$.
The vertex cover constructed with the approximation algorithm A has size $A(I) = 2|M|$.
$\Rightarrow \sup_I \frac{A(I)}{opt(I)} \leq \frac{2|M|}{|M|} = 2$

Approximation algorithm for Vertex Cover

Is this indeed an approximation algorithm?
I.e.: can we bound $\sup_I \frac{A(I)}{opt(I)}$?
What would be the approximation ratio?

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

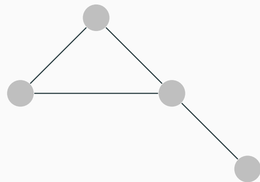
Proof: We consider an instance I of vertex cover.
Let M be a maximum matching in the considered graph G.
For each edge in M, any vertex cover S contains at least one vertex. Thus $opt(I) \geq |M|$.
The vertex cover constructed with the approximation algorithm A has size $A(I) = 2|M|$.
$\Rightarrow \sup_I \frac{A(I)}{opt(I)} \leq \frac{2|M|}{|M|} = 2$

Is this tight?

Is this analysis tight?

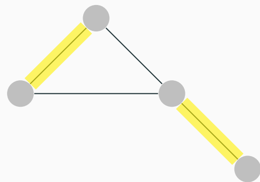
Is this analysis tight?

Approximate Solution



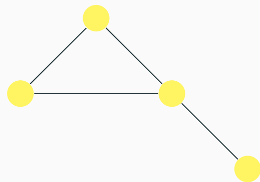
Is this analysis tight?

Approximate Solution



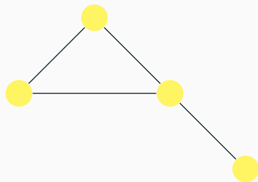
Is this analysis tight?

Approximate Solution

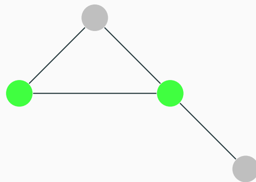


Is this analysis tight?

Approximate Solution



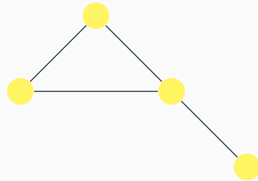
Optimal Solution



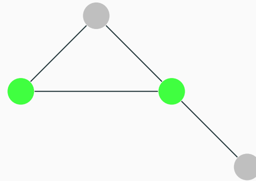
Approximation algorithm for Vertex Cover

Is this analysis tight?

Approximate Solution



Optimal Solution



Note: This is the best know approximation algorithm for vertex cover. But it is an open problem whether a better one exists.

Questions?

Questions?

Makespan minimization

Instance: m machines; n jobs with processing times $p_1, \dots, p_n$
Goal: assign jobs to machines so that the maximum workload (= makespan) is minimized

Decision variant in NP, NP-hard (thus NP-complete) already for $m = 2$:

Makespan minimization

Instance: m machines; n jobs with processing times $p_1, \ldots, p_n$
Goal: assign jobs to machines so that the maximum workload (= makespan) is minimized

Decision variant in NP, NP-hard (thus NP-complete) already for $m = 2$:
use a reduction from 2-PARTITION

Makespan minimization

Instance: m machines; n jobs with processing times $p_1, \ldots, p_n$
Goal: assign jobs to machines so that the maximum workload (= makespan) is minimized

Decision variant in NP, NP-hard (thus NP-complete) already for $m = 2$:
use a reduction from 2-PARTITION

List scheduling algorithm

Work through the job list one by one; in each step
assign current job to machine with currently smallest workload

Makespan minimization

Instance: m machines; n jobs with processing times $p_1, \ldots, p_n$
Goal: assign jobs to machines so that the maximum workload (= makespan) is minimized

Decision variant in NP, NP-hard (thus NP-complete) already for $m = 2$:
use a reduction from 2-PARTITION

List scheduling algorithm

Work through the job list one by one; in each step
assign current job to machine with currently smallest workload

Theorem

List scheduling algorithm has approximation ratio 2.

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

Makespan minimization

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$$\mathrm{opt}(I) \geq \max p_i$$
$$\mathrm{opt}(I) \geq \frac{1}{m}\sum_{i=1}^{n} p_i$$

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$$\mathrm{opt}(I) \geq \max p_i$$
$$\mathrm{opt}(I) \geq \frac{1}{m}\sum_{i=1}^{n} p_i$$

2.) Consider machine j that determines the makespan

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$$\text{opt}(I) \geq \max p_i$$
$$\text{opt}(I) \geq \frac{1}{m}\sum_{i=1}^{n} p_i$$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Makespan minimization

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$$\mathrm{opt}(I) \geq \max p_i$$
$$\mathrm{opt}(I) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

Makespan minimization

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$\text{opt}(I) \geq \max p_i$

$\text{opt}(I) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

At this moment j has workload at most $\frac{1}{m} \sum_{i'=1}^{n} p_{i'} \leq \text{opt}(I)$

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$\text{opt}(I) \geq \max p_i$

$\text{opt}(I) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

At this moment j has workload at most $\frac{1}{m} \sum_{i'=1}^{n} p_{i'} \leq \text{opt}(I)$

job i adds $p_i \leq \max_{i'} p_{i'} \leq \text{opt}(I)$ to workload of j

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$\text{opt}(I) \geq \max p_i$

$\text{opt}(I) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

At this moment j has workload at most $\frac{1}{m} \sum_{i'=1}^{n} p_{i'} \leq \text{opt}(I)$

job i adds $p_i \leq \max_{i'} p_{i'} \leq \text{opt}(I)$ to workload of j

Thus $A(I) \leq \frac{1}{m} \sum_{i'=1}^{n} p_{i'} + p_i \leq 2OPT(I)$

Makespan minimization

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$\text{opt}(I) \geq \max p_i$

$\text{opt}(I) \geq \frac{1}{m}\sum_{i=1}^{n} p_i$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

At this moment j has workload at most $\frac{1}{m}\sum_{i'=1}^{n} p_{i'} \leq \text{opt}(I)$

job i adds $p_i \leq \max_{i'} p_{i'} \leq \text{opt}(I)$ to workload of j

Thus $A(I) \leq \frac{1}{m}\sum_{i'=1}^{n} p_{i'} + p_i \leq 2OPT(I)$

Is this bound tight?

Theorem

List scheduling algorithm has approximation ratio 2.

Proof:

1.) Lower bounds:

$\text{opt}(I) \geq \max p_i$

$\text{opt}(I) \geq \frac{1}{m}\sum_{i=1}^{n} p_i$

2.) Consider machine j that determines the makespan

Consider last job i assigned to machine j

Consider the moment when i was assigned to j

At this moment j has workload at most $\frac{1}{m}\sum_{i'=1}^{n} p_{i'} \leq \text{opt}(I)$

job i adds $p_i \leq \max_{i'} p_{i'} \leq \text{opt}(I)$ to workload of j

Thus $A(I) \leq \frac{1}{m}\sum_{i'=1}^{n} p_{i'} + p_i \leq 2OPT(I)$

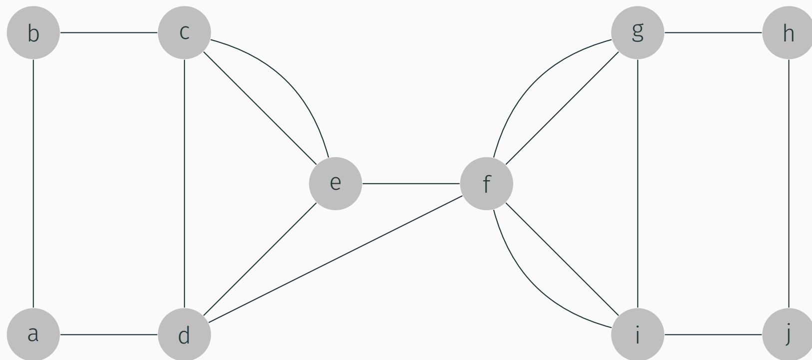
Is this bound tight?

Can sharpen above analysis to guarantee $2 - 1/m$ (exercise).

Questions?

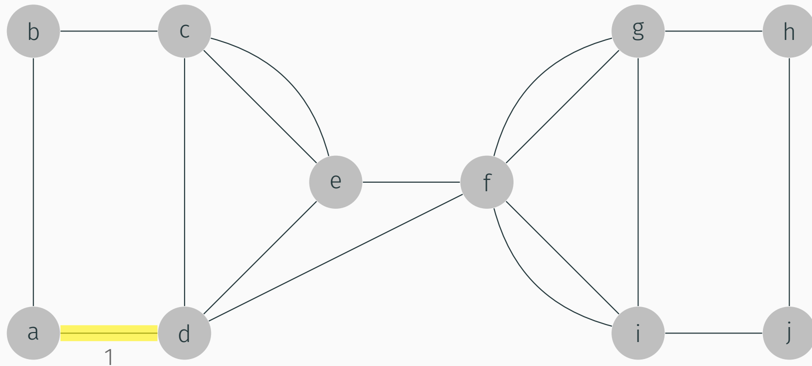
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



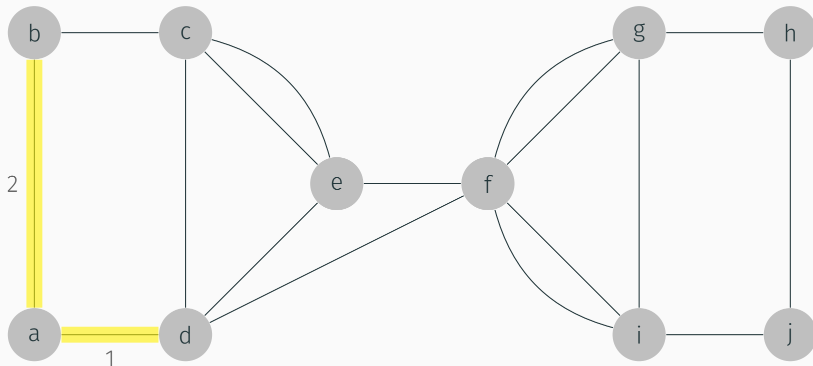
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



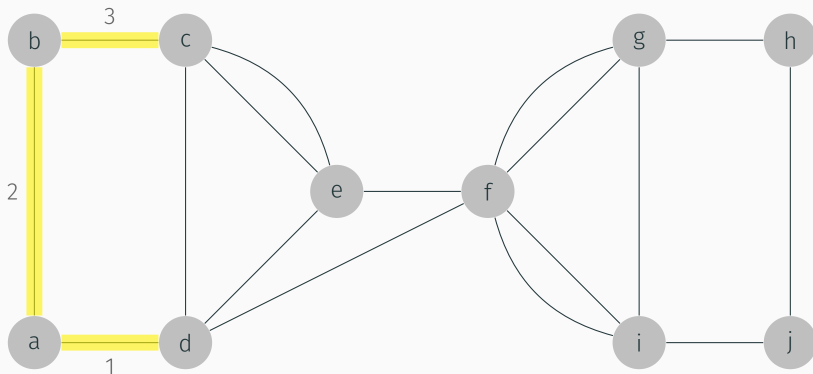
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



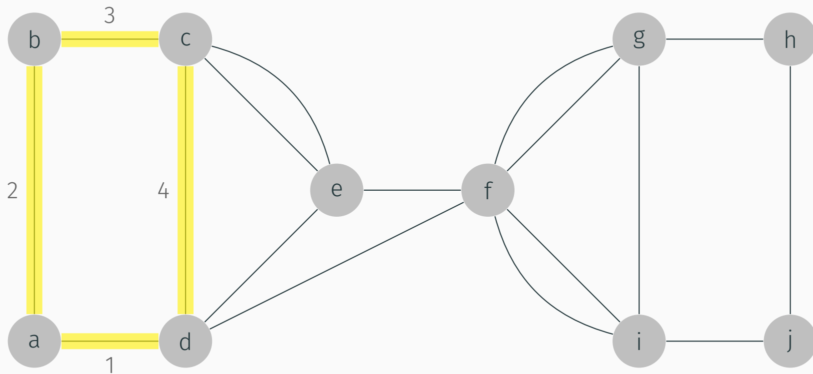
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



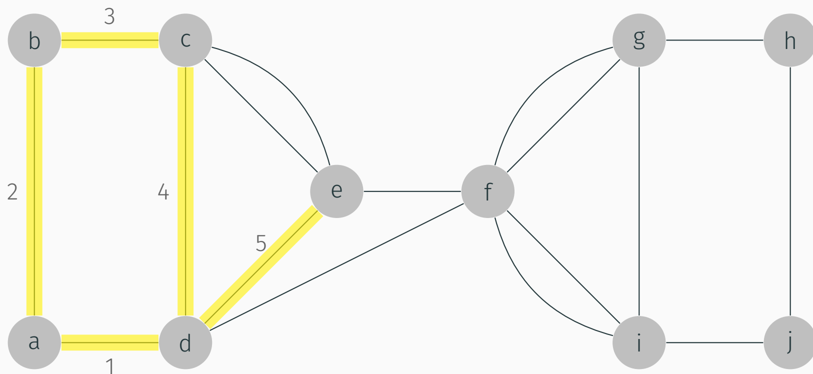
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



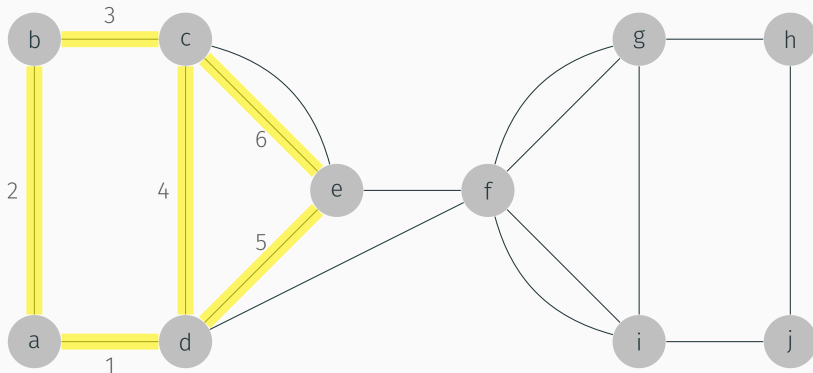
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



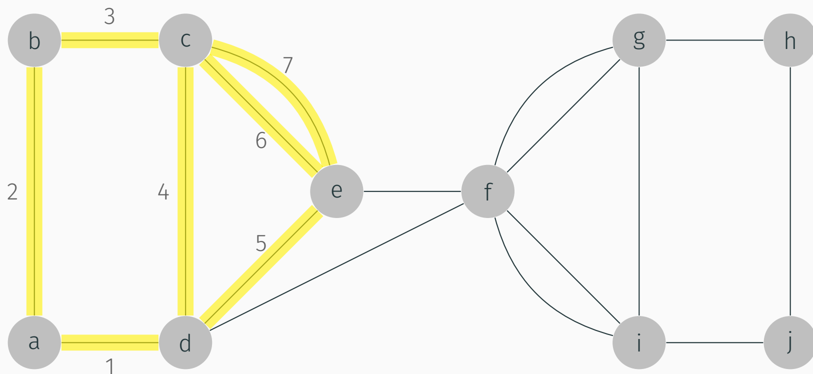
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



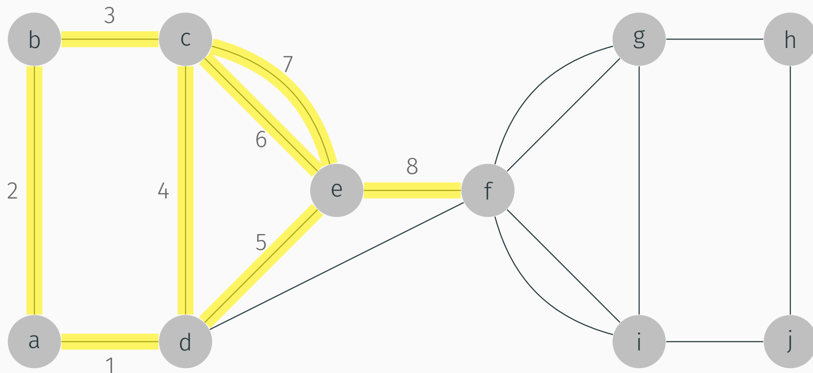
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



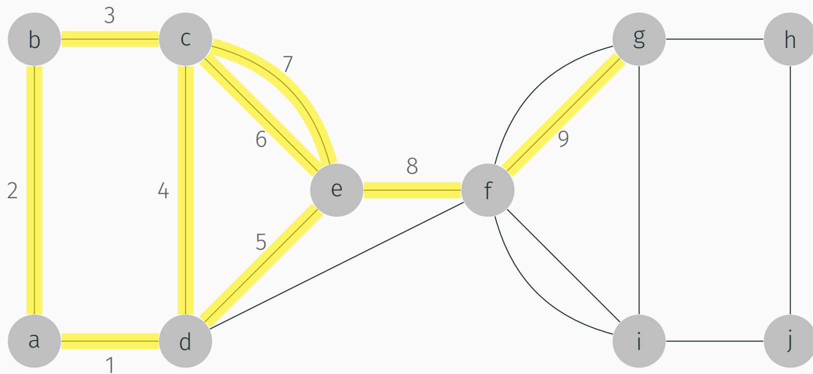
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



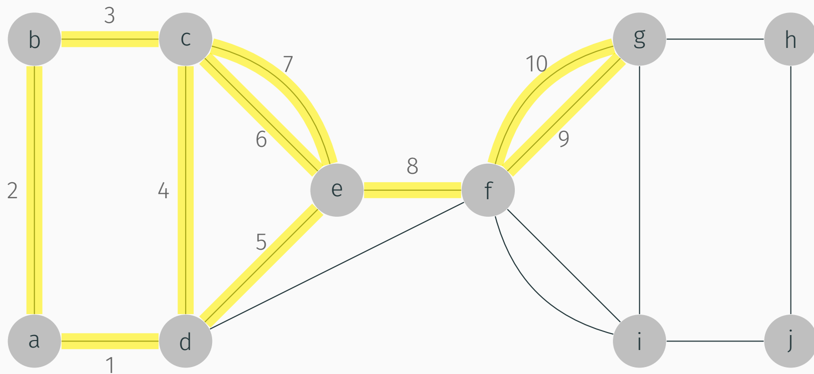
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



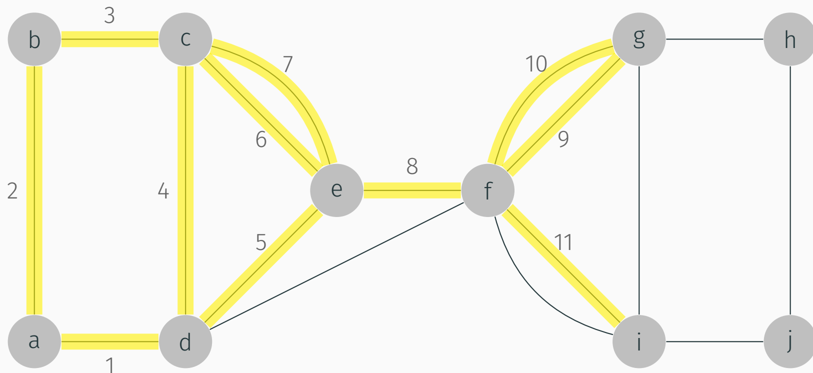
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



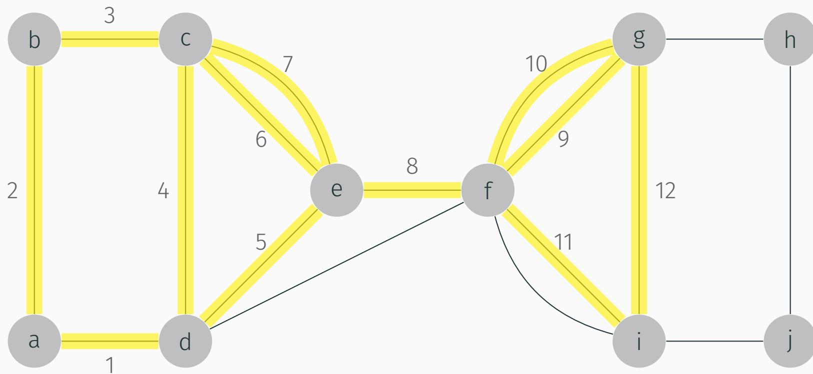
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



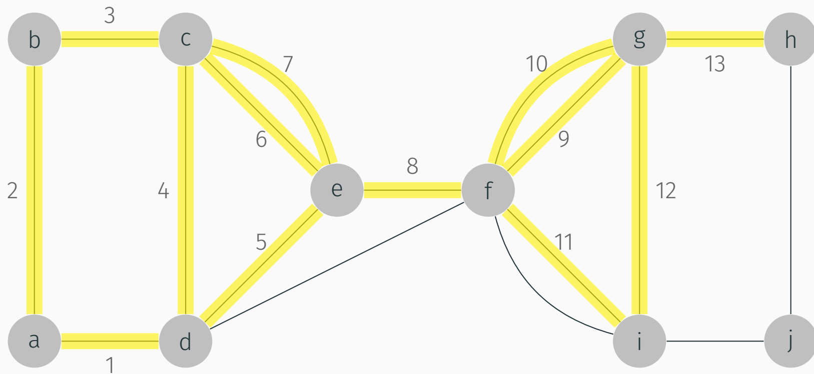
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



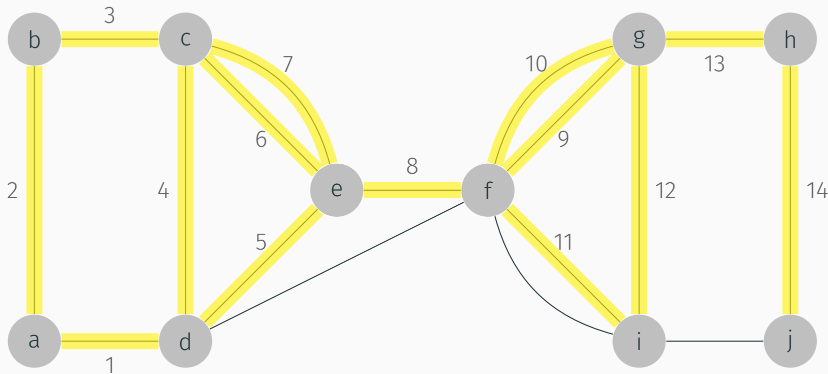
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



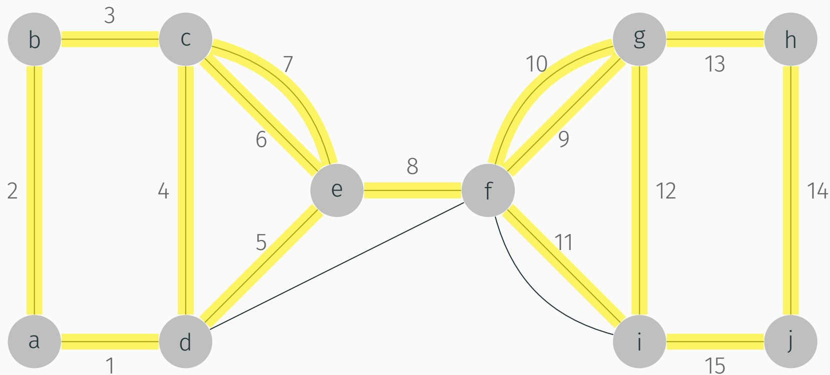
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



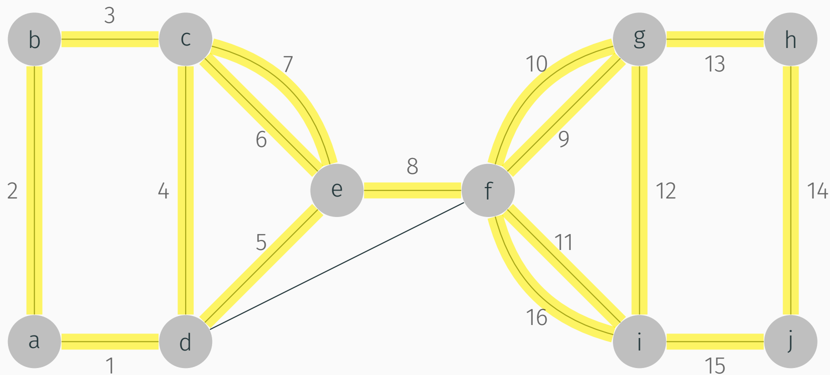
Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.

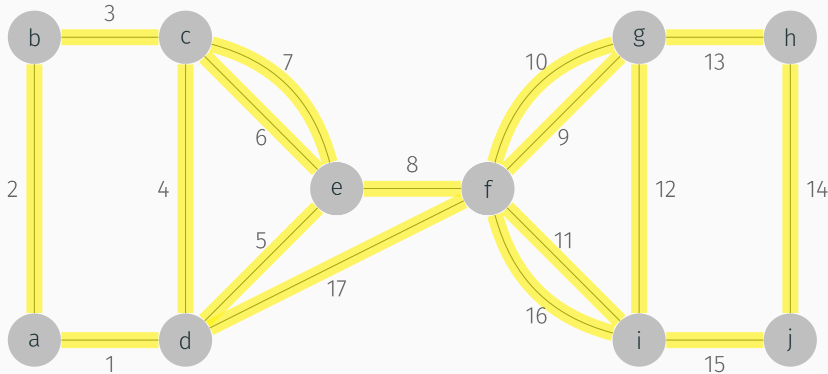


Intermezzo: Euler tours

- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



- Multi-graph: $G = (V, E)$ but now E may be a multi-set (edges may occur multiple times)
- A Eulerian tour of a graph is a tour visiting all *edges* exactly once.



Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Proof sketch:

Intermezzo: Euler tours

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Proof sketch:

($\rightarrow$) the tour enters and leaves vertices consecutively; ends at start.

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Proof sketch:
($\rightarrow$) the tour enters and leaves vertices consecutively; ends at start.
($\leftarrow$) start at any vertex v walk along unused edges as long as possible
If we end at v and edges are left incident to a visited vertex u,

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Proof sketch:
($\rightarrow$) the tour enters and leaves vertices consecutively; ends at start.
($\leftarrow$) start at any vertex v walk along unused edges as long as possible
If we end at v and edges are left incident to a visited vertex u,
walk from u and insert obtained tour in previous tour.

Euler tour

Instance: A graph $G = (V, E)$.

Question: Is there an Euler tour?

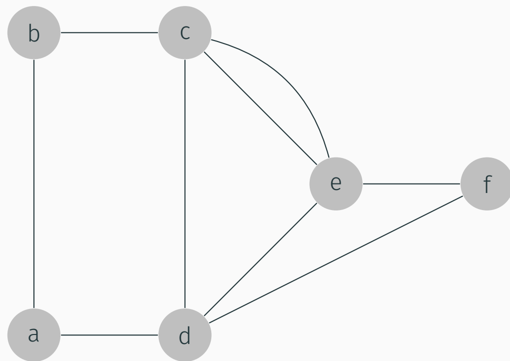
Theorem (Euler in 1736, first theorem in graph theory!!)

A connected graph has a Euler tour iff all vertices have even degree.

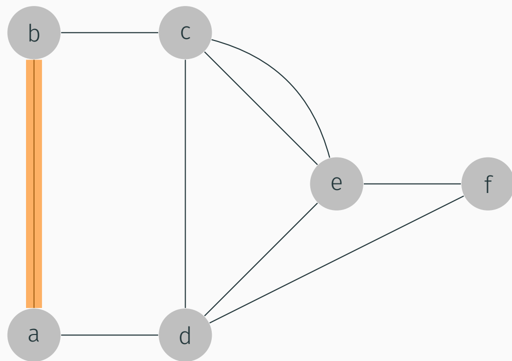
(In a multi-graph $G = (V, E)$ the degree of vertex $v \in V$ is the number of edges $\{u, v\} \in E$.)

Proof sketch:
($\rightarrow$) the tour enters and leaves vertices consecutively; ends at start.
($\leftarrow$) start at any vertex v walk along unused edges as long as possible
If we end at v and edges are left incident to a visited vertex u,
walk from u and insert obtained tour in previous tour.
(This is a constructive polynomial-time algorithm)

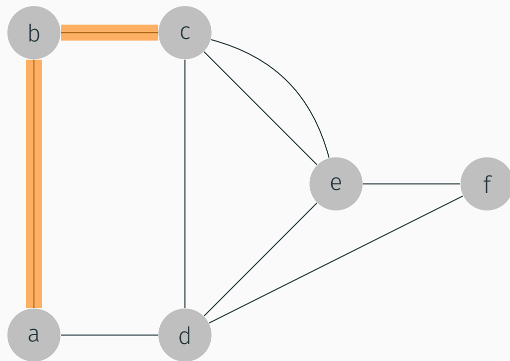
Intermezzo: Euler tours



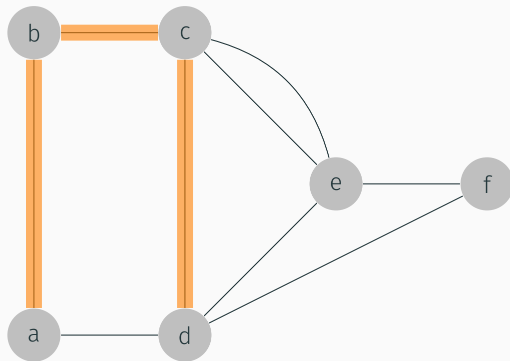
Path:



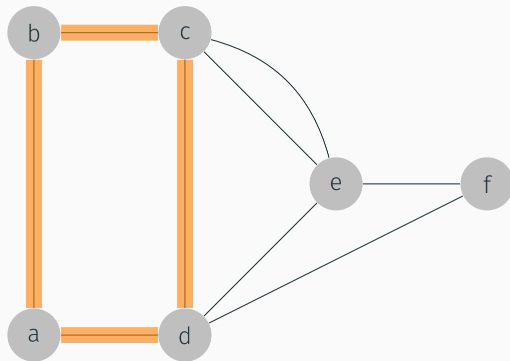
Path:



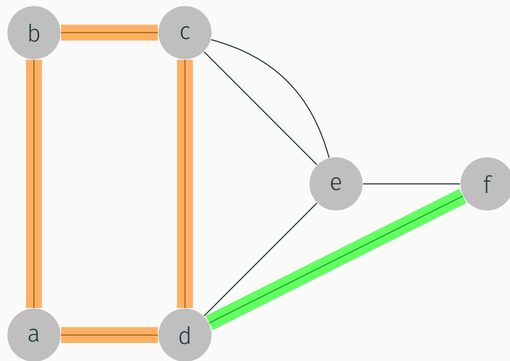
Path:



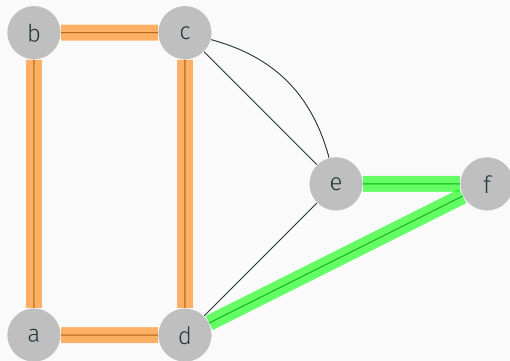
Path:



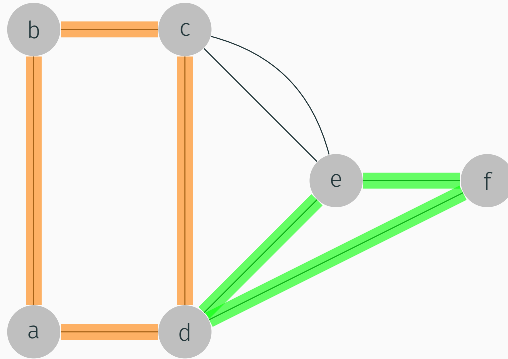
Path: abcda



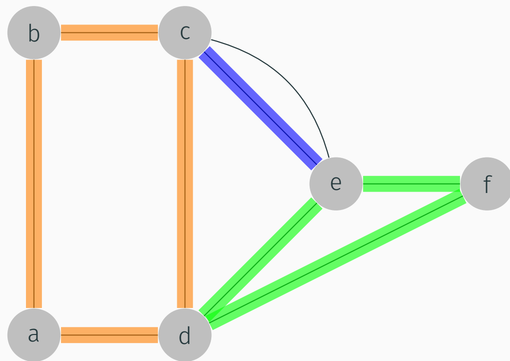
Path: abcda



Path: abcda

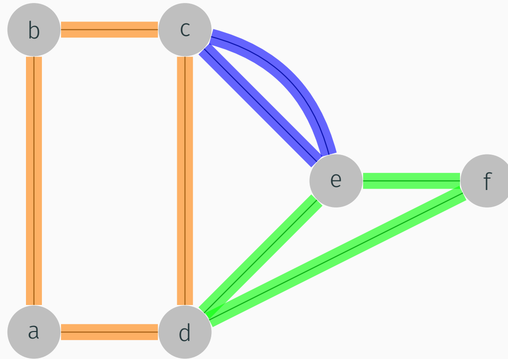


Path: abcdfeda

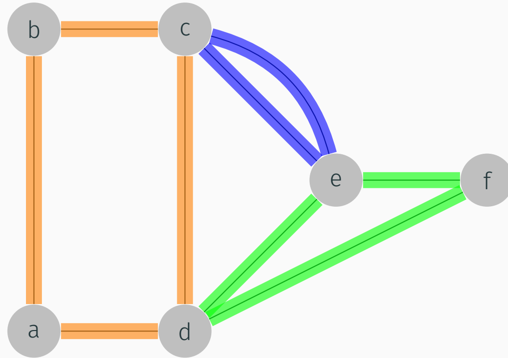


Path: abcdfeda

Intermezzo: Euler tours



Path: abcecdfeda or abcdfeceda



Path: abcecdfeda or abcdfeceda

Observe: every sub-walk can only end when it returns to the starting vertex (due to the even-degree property)

Questions?

TSP (Optimization version)

Instance: cities $1, \dots, n$; distances $d(i, j)$

Goal: find roundtrip of smallest possible length

NP-complete (reduction from Hamiltonian cycle)

Can we approximate the TSP?

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i, j)$
Goal: find roundtrip of smallest possible length

NP-complete (reduction from Hamiltonian cycle)

Can we approximate the TSP?

Lower bounds:

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i, j)$

Goal: find roundtrip of smallest possible length

NP-complete (reduction from Hamiltonian cycle)

Can we approximate the TSP?

Lower bounds:

- $\text{opt}(I) \geq$ length of minimum spanning tree MST

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i, j)$
Goal: find roundtrip of smallest possible length

NP-complete (reduction from Hamiltonian cycle)

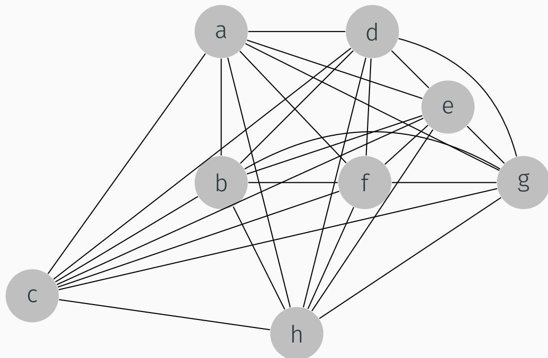
Can we approximate the TSP?

Lower bounds:

- $\text{opt}(I) \geq$ length of minimum spanning tree MST
- $\text{opt}(I) \geq$ twice the length of min. weight perfect matching, if n even

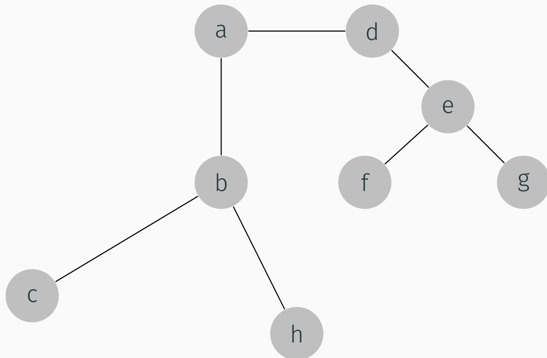
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights



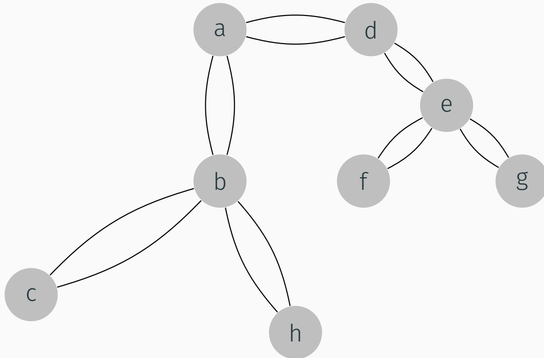
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST



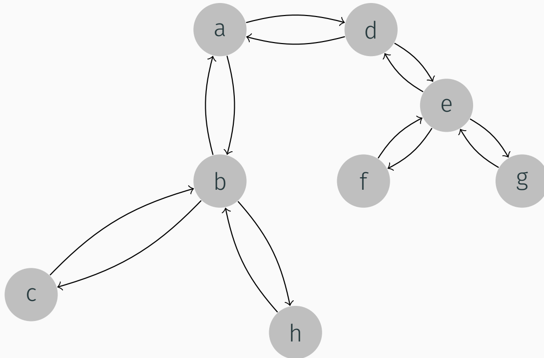
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph



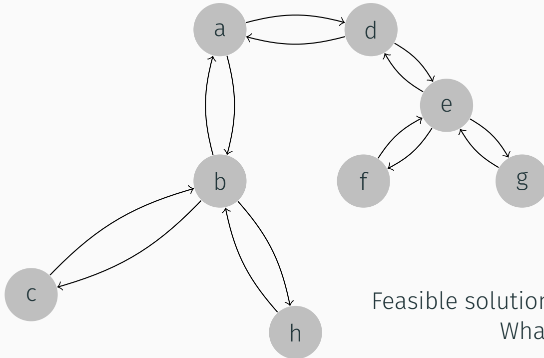
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST

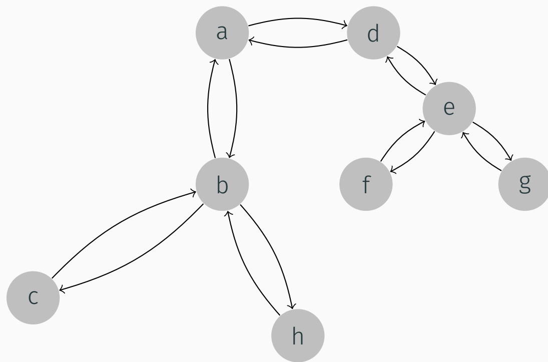


Double-tree algorithm

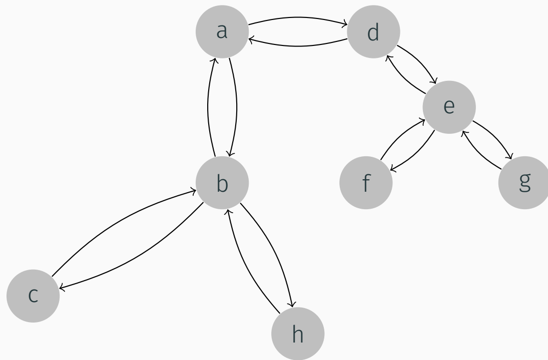
1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST



Feasible solution if cities may be visited more than once.
What is the approximation ratio?

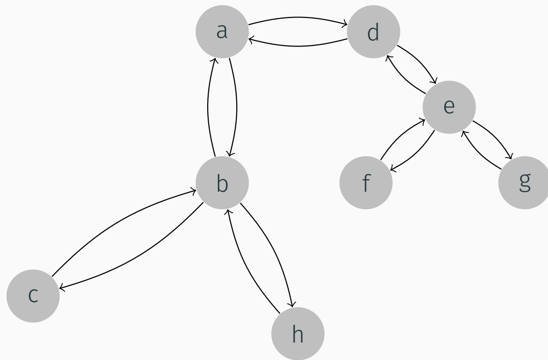


Approximation ratio of double tree algorithm



Approximation ratio of double tree algorithm

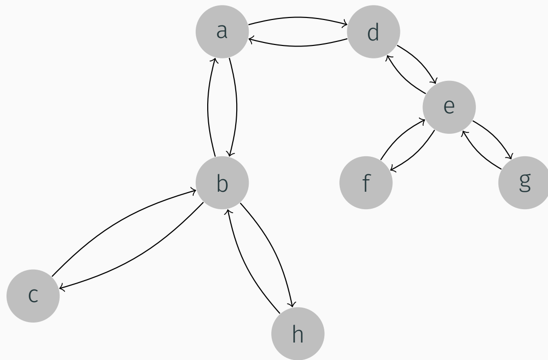
$\text{opt}(I) \geq \text{MST}(I)$



Approximation ratio of double tree algorithm

$\mathrm{opt}(I) \geq \mathrm{MST}(I)$

$A(I) = 2 \cdot \mathrm{MST}(I)$



Approximation ratio of double tree algorithm

$\text{opt}(I) \geq \text{MST}(I)$

$A(I) = 2 \cdot \text{MST}(I)$

$\Rightarrow \sup_I \frac{A(I)}{\text{opt}(I)} \leq \frac{2\text{MST}(I)}{\text{MST}(I)} = 2$

Questions?

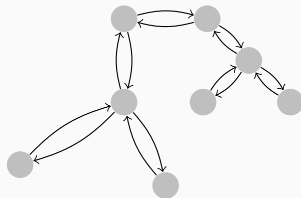
Questions?

What if we do not want to visit cities more than once?

TSP (Optimization version)

Instance: cities $1, \dots, n$; distances $d(i, j)$

Goal: find roundtrip of smallest possible length **that visits every city at most once**

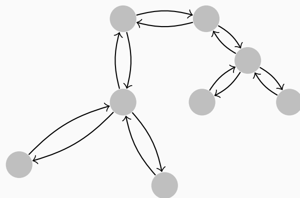


What if we do not want to visit cities more than once?

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i, j)$

Goal: find roundtrip of smallest possible length **that visits every city at most once**



Metric **TSP**

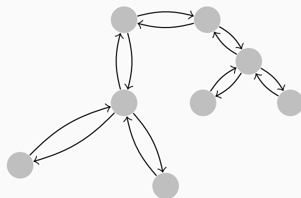
If the distances satisfy the triangle inequality $d(x, y) + d(y, z) \;\geq\; d(x, z)$ for all cities x, y, z

What if we do not want to visit cities more than once?

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i,j)$

Goal: find roundtrip of smallest possible length **that visits every city at most once**



***Metric* TSP**

If the distances satisfy the triangle inequality $d(x,y) + d(y,z) \geq d(x,z)$ for all cities x, y, z

Euclidean-TSP $\subseteq$ Metric TSP $\subseteq$ General TSP

All are NP-complete. For metric TSP reduce from HC using distances 1 and 2)

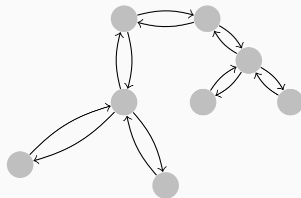
Euclidean-TSP (points in plane) requires harder reduction (beyond scope for us)

What if we do not want to visit cities more than once?

TSP (Optimization version)

Instance: cities $1, \ldots, n$; distances $d(i,j)$

Goal: find roundtrip of smallest possible length **that visits every city at most once**



Metric **TSP**

If the distances satisfy the triangle inequality $d(x,y) + d(y,z) \;\geq\; d(x,z)$ for all cities x, y, z

Euclidean-TSP $\subseteq$ Metric TSP $\subseteq$ General TSP

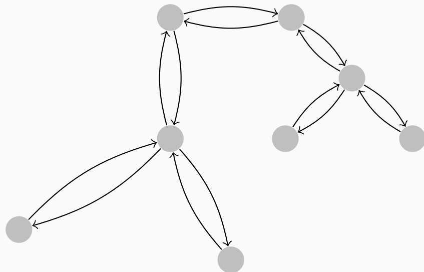
All are NP-complete. For metric TSP reduce from HC using distances 1 and 2)

Euclidean-TSP (points in plane) requires harder reduction (beyond scope for us)

In case of Metric TSP, we can “shortcut”!

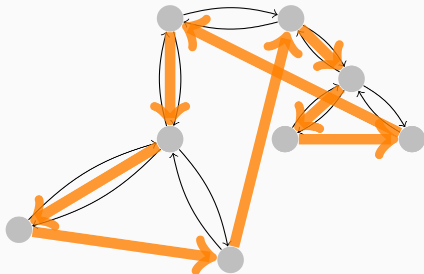
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST



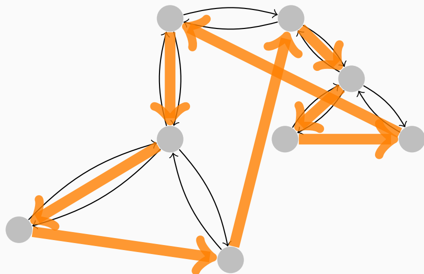
Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour



Double-tree algorithm

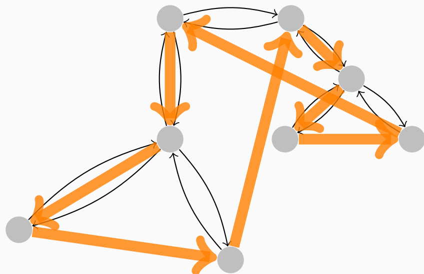
1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour



Why does this work (only) in metric TSP?

Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour

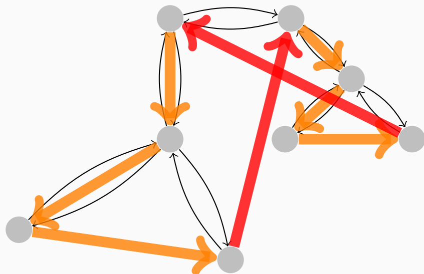


Why does this work (only) in metric TSP?

Can be further improved with a 2-opt move.

Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour

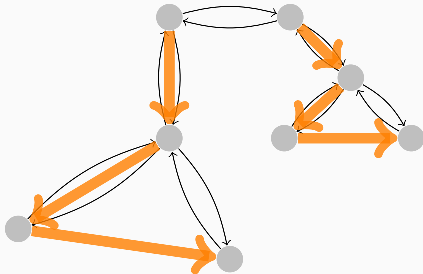


Why does this work (only) in metric TSP?

Can be further improved with a 2-opt move.

Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour

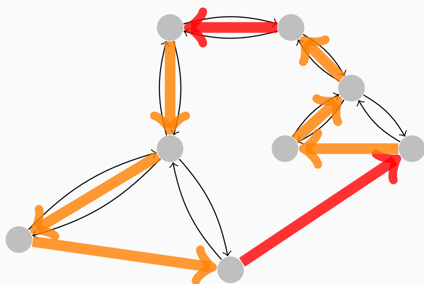


Why does this work (only) in metric TSP?

Can be further improved with a 2-opt move.

Double-tree algorithm

1. Represent cities and distances as a graph with edge weights
2. Compute a minimum spanning tree MST
3. Double every edge in MST to get a Eulerian graph
4. Compute a Euler tour in the doubled MST
5. Shortcut the Euler tour to a TSP-tour



Why does this work (only) in metric TSP?

Can be further improved with a 2-opt move.

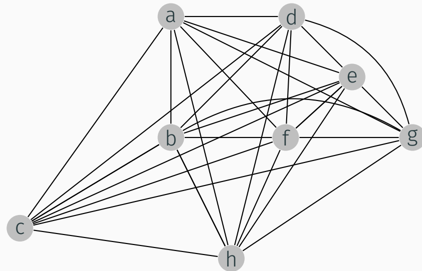
Questions?

Questions?

Christofides Algorithm

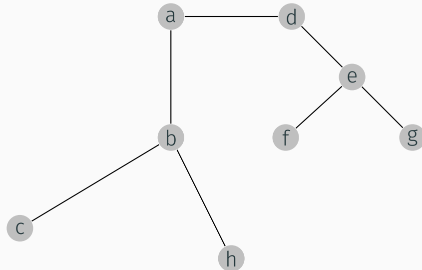
Algorithm

- Represent cities and distances as a graph with edge weights



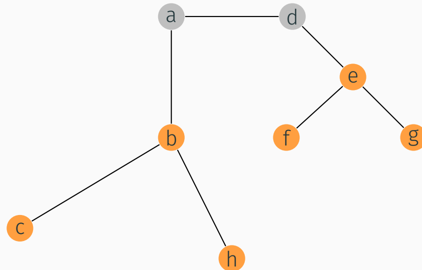
Algorithm

- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST



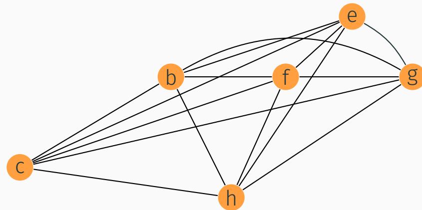
Algorithm

- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities



Algorithm

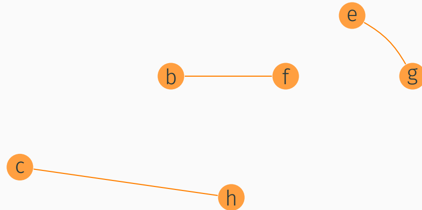
- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities



Christofides Algorithm

Algorithm

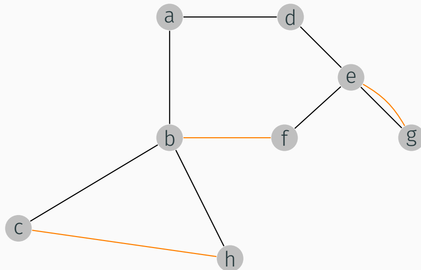
- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities



Christofides Algorithm

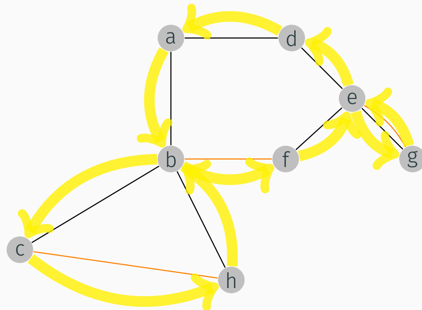
Algorithm

- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities
- Compute an Euler tour in the union of the MST and M



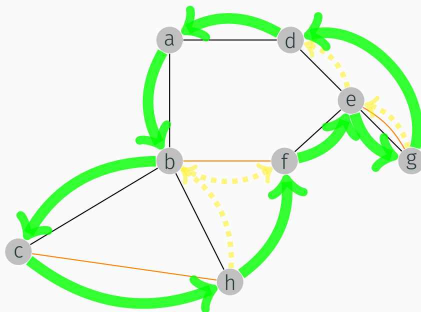
Algorithm

- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities
- Compute an Euler tour in the union of the MST and M



Algorithm

- Represent cities and distances as a graph with edge weights
- Compute a minimum spanning tree MST
- Compute a minimum perfect matching M for odd-degree cities
- Compute an Euler tour in the union of the MST and M
- Shortcut the Euler tour to a TSP tour



Two remaks:

Odd-degree vertices

The number of odd-degree vertices in the MST is even.

Two remaks:

Odd-degree vertices

The number of odd-degree vertices in the MST is even.

Why? The sum of the degrees of all nodes in a graph is $2|E|$, which is even.

Two remaks:

Odd-degree vertices

The number of odd-degree vertices in the MST is even.

Why? The sum of the degrees of all nodes in a graph is $2|E|$, which is even.

So there always exists a perfect matching in the complete graph connecting the odd-degree vertices of the MST.

Two remaks:

Odd-degree vertices

The number of odd-degree vertices in the MST is even.

Why? The sum of the degrees of all nodes in a graph is $2|E|$, which is even.

So there always exists a perfect matching in the complete graph connecting the odd-degree vertices of the MST.

Minimum-weight perfect matching

This can be done in polynomial time, but is not trivial!

Two remaks:

Odd-degree vertices

The number of odd-degree vertices in the MST is even.

Why? The sum of the degrees of all nodes in a graph is $2|E|$, which is even.

So there always exists a perfect matching in the complete graph connecting the odd-degree vertices of the MST.

Minimum-weight perfect matching

This can be done in polynomial time, but is not trivial!

Original proof due to seminal work by Jack Edmonds (1965), Maximum matching and a polyhedron with 0,1-vertices, *Journal of Research National Bureau of Standards Section B 69*.

Many later references/survey articles available. E.g. *Computing minimum-weight perfect matchings* by Cook and Rohe, INFORMS Journal on Computing 1999.

Christofides Algorithm

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

$A(I) \leq$ length of MST plus weight of matching

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

$A(I) \leq$ length of MST plus weight of matching

weight of matching $\leq 1/2$ length of a tour visiting all odd degree vertices

Christofides Algorithm

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

$A(I) \leq$ length of MST plus weight of matching

weight of matching $\leq 1/2$ length of a tour visiting all odd degree vertices

$\leq 1/2$ length of a tour visiting all vertices

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

$A(I) \leq$ length of MST plus weight of matching

weight of matching $\leq 1/2$ length of a tour visiting all odd degree vertices

$\leq 1/2$ length of a tour visiting all vertices

$A(I) \leq$ length of MST plus weight of matching $\leq \text{opt}(I) + \text{opt}(I)/2$

Theorem

The Christofides algorithm has approximation ratio 1.5.

Proof:

Consider the following lower bounds:

$\text{opt}(I) \geq$ length of minimum spanning tree MST

$\text{opt}(I) \geq$ twice the length of min. weight perfect

$A(I) \leq$ length of MST plus weight of matching

weight of matching $\leq 1/2$ length of a tour visiting all odd degree vertices

$\leq 1/2$ length of a tour visiting all vertices

$A(I) \leq$ length of MST plus weight of matching $\leq \text{opt}(I) + \text{opt}(I)/2$ ✓

Final Remark

Since recently the algorithm is sometimes referred to as Christofides–Serdyukov.

See *A historical note on the 3/2-approximation algorithm for the metric traveling salesman problem* (2020) in Historica Mathematica

Questions?

Linear Programming-based approaches

1. Find an exact ILP formulation

1. Find an exact ILP formulation
2. Relax integrality constraints (ILP $\rightarrow$ LP)

1. Find an exact ILP formulation
2. Relax integrality constraints (ILP $\rightarrow$ LP)
3. Solve the LP relaxation in polynomial time

LP-based approaches

1. Find an exact ILP formulation
2. Relax integrality constraints (ILP $\rightarrow$ LP)
3. Solve the LP relaxation in polynomial time
4. Round the optimal LP solution to approximate ILP solution (preserving feasibility!)

Weighted vertex cover (VC)

Instance: a graph $G = (V, E)$; weights $w : V \to \mathbb{R}^+$
Goal: find a vertex cover of smallest possible weight
(e.g. find a vertex cover $X \subseteq V$ minimizing $\sum_{v \in V} w(v)$)

Weighted vertex cover (VC)

Instance: a graph $G = (V, E)$; weights $w : V \to \mathbb{R}^+$
Goal: find a vertex cover of smallest possible weight
(e.g. find a vertex cover $X \subseteq V$ minimizing $\sum_{v \in V} w(v)$)

ILP formulation

$$\begin{aligned} \text{minimize} \quad & \textstyle\sum_{v \in V} w(v) \cdot x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \quad \text{for every edge } \{u, v\} \in E \\ & x_v \in \{0, 1\} \quad \text{for every vertex } v \in V \end{aligned}$$

Weighted vertex cover (VC)

Instance: a graph $G = (V, E)$; weights $w : V \to \mathbb{R}^+$
Goal: find a vertex cover of smallest possible weight
(e.g. find a vertex cover $X \subseteq V$ minimizing $\sum_{v \in V} w(v)$)

ILP formulation

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} w(v) \cdot x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every edge } \{u, v\} \in E \\ & x_v \in \{0, 1\} & \text{for every vertex } v \in V \end{array}$$

LP relaxation

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} w(v) \cdot x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for every edge } \{u, v\} \in E \\ & 0 \leq x_v \leq 1 & (\text{or simply } 0 \leq x_v) \text{ for } v \in V \end{array}$$

Approximation algorithm

1. Compute the optimal LP solution x_v^*

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
   - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
   - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
   - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
    - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
    - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
   - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
   - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

For each edge $\{u, v\} \in E$ we have

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
    - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
    - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

For each edge $\{u, v\} \in E$ we have

$$x_u + x_v \geq 1$$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
   - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
   - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

For each edge $\{u, v\} \in E$ we have

$$x_u + x_v \geq 1$$

$\Rightarrow x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$.

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
    - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
    - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

For each edge $\{u, v\} \in E$ we have

$$x_u + x_v \geq 1$$

$\Rightarrow x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$.
$\Rightarrow \tilde{x}_u = 1$ or $\tilde{x}_v^* = 1$.

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:
   - If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
   - If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Why does this give a vertex cover?

For each edge $\{u, v\} \in E$ we have

$$x_u + x_v \geq 1$$

$\Rightarrow x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$.
$\Rightarrow \tilde{x}_u = 1$ or $\tilde{x}_v^* = 1$.
$\Rightarrow u$ or v contained in vertex cover.

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:$opt(I) = opt_{ILP}$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof: $opt(I) = opt_{ILP} \geq opt_{LP}$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:$opt(I) = opt_{ILP} \geq opt_{LP}$
and $A(I) \leq 2opt_{LP}$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:$opt(I) = opt_{ILP} \geq opt_{LP}$
and $A(I) \leq 2opt_{LP}$
Thus $\frac{A(I)}{opt(I)} \leq$

Approximation algorithm

1. Compute the optimal LP solution x_v^*
2. Round the LP solution x_v^* to a feasible ILP-solution $\tilde{x}_v$:

If $x_v^* < 1/2$ then $\tilde{x}_v = 0$
If $x_v^* \geq 1/2$ then $\tilde{x}_v = 1$

Theorem

This polynomial-time approximation algorithm has approximation ratio 2.

Proof:$opt(I) = opt_{ILP} \geq opt_{LP}$
and $A(I) \leq 2opt_{LP}$
Thus $\frac{A(I)}{opt(I)} \leq \frac{2opt_{LP}}{opt_{LP}} = 2$

Questions?

Questions?