Chapter 8
Approximation Algorithms

Algorithm Theory
WS 2017/18

Fabian Kuhn

Approximation Algorithms

- Optimization appears everywhere in computer science
- We have seen many examples, e.g.:
  - scheduling jobs
  - traveling salesperson
  - maximum flow, maximum matching
  - minimum spanning tree
  - minimum vertex cover
  - …
- Many discrete optimization problems are NP-hard
- They are however still important and we need to solve them
- As algorithm designers, we prefer algorithms that produce solutions which are provably good, even if we can't compute an optimal solution.

Approximation Algorithms: Examples

We have already seen two approximation algorithms

- **Metric TSP:** If distances are positive and satisfy the triangle inequality, the greedy tour is only by a log-factor longer than an optimal tour

- **Maximum Matching and Vertex Cover:** A maximal matching gives solutions that are within a factor of 2 for both problems.

Approximation Ratio

An approximation algorithm is an algorithm that computes a solution for an optimization with an objective value that is provably within a bounded factor of the optimal objective value.

Formally:

- $\mathrm{OPT} \geq 0$: optimal objective value
  $\mathrm{ALG} \geq 0$: objective value achieved by the algorithm

- **Approximation Ratio α:**

$$\textbf{Minimization: } \alpha := \max_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

$$\frac{\mathrm{OPT}}{\mathrm{ALG}}$$

$$\textbf{Maximization: } \alpha := \min_{\text{input instances}} \frac{\mathrm{ALG}}{\mathrm{OPT}}$$

Example: Load Balancing

We are given:

- m machines $M_1, \dots, M_m$
- n jobs, processing time of job i is t_i

Goal:

- Assign each job to a machine such that the makespan is minimized

  makespan: largest total processing time of any machine

The above load balancing problem is NP-hard and we therefore want to get a good approximation for the problem.

Greedy Algorithm

There is a simple greedy algorithm:

- Go through the jobs in an arbitrary order
- When considering job i, assign the job to the machine that currently has the smallest load.

Example: 3 machines, 12 jobs

3	4	2	3	1	6	4	4	3	2	1	5

Greedy Assignment:

makespan = 16

M_1: 3 | 1 | 6 | 1 | 5

M_2: 4 | 4 | 3

M_3: 2 | 3 | 4 | 2

Optimal Assignment:

makespan = 13

M_1: 3 | 4 | 2 | 3 | 1

M_2: 6 | 4 | 3

M_3: 4 | 2 | 1 | 5

Greedy Analysis

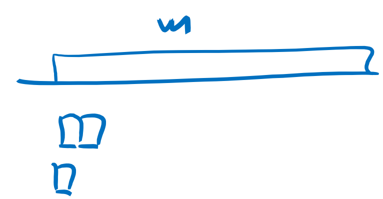
- We will show that greedy gives a 2-approximation

- To show this, we need to compare the solution of greedy with an optimal solution (that we can't compute)

- Lower bound on the optimal makespan T^*:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

- Lower bound can be far from T^*:
  - m machines, m jobs of size 1, 1 job of size m



$$T^* = m, \qquad \frac{1}{m} \cdot \sum_{i=1}^{n} t_i = 2$$

Greedy Analysis

- We will show that greedy gives a 2-approximation
- To show this, we need to compare the solution of greedy with an optimal solution (that we can't compute)
- Lower bound on the optimal makespan T^*:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

- Second lower bound on optimal makespan T^*:

$$T^* \geq \max_{1 \leq i \leq n} t_i$$

Greedy Analysis

Theorem: The greedy algorithm has approximation ratio ≤ 2, i.e., for the makespan T of the greedy solution, we have $T \leq 2T^*$.

Proof:

- For machine k, let T_k be the time used by machine k
- Consider some machine M_i for which $T_i = T$
- Assume that job j is the last one schedule on M_i:

$\boldsymbol{M_i}$:

$T - t_j$	t_j

- When job j is scheduled, M_i has the minimum load

$\hookrightarrow \forall k : T_k \geq T - t_j$

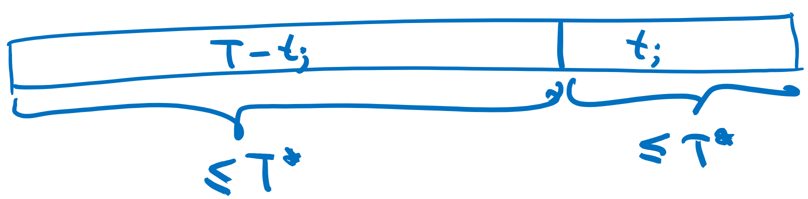
Greedy Analysis

Theorem: The greedy algorithm has approximation ratio ≤ 2, i.e., for the makespan T of the greedy solution, we have $T \leq 2T^*$.

Proof:

- For all machines M_k: load $T_k \geq T - t_j$ $\quad \sum t_i \geq m(T - t_j)$

$\hookrightarrow T^* \geq \frac{1}{m} \sum t_i \geq T - t_j$



$T = T - t_j + t_j \leq 2T^*$

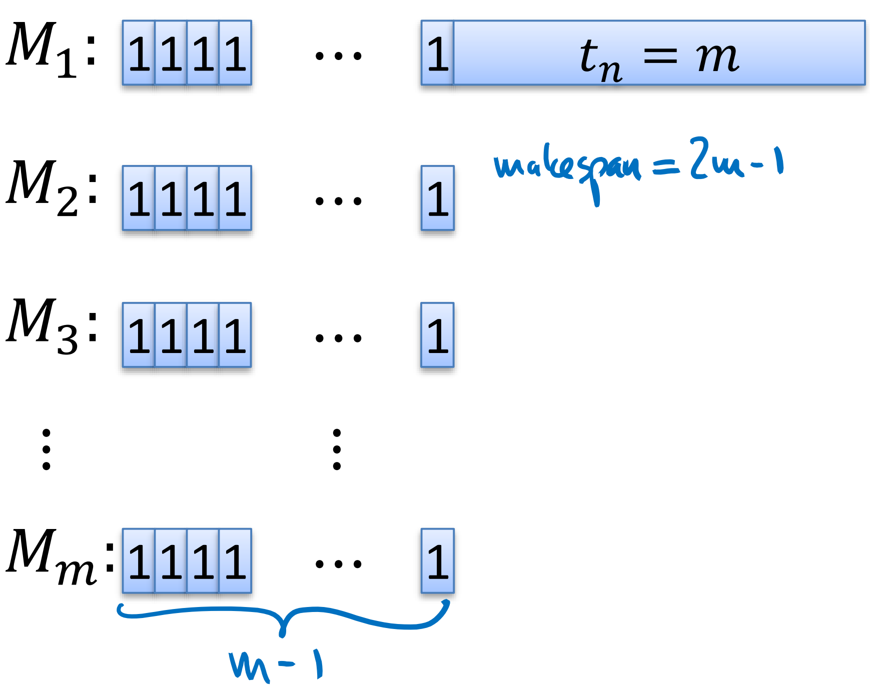
Can We Do Better?

$\frac{2m-1}{m} = 2 - \frac{1}{m}$

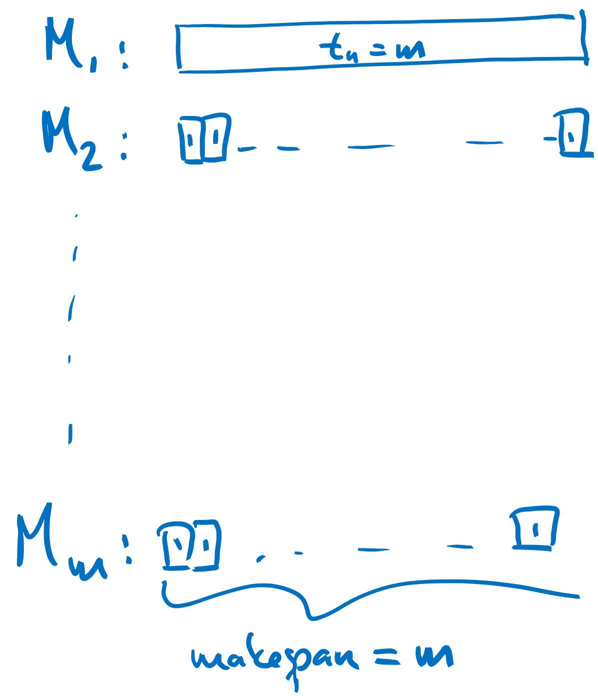
The analysis of the greedy algorithm is almost tight:

- Example with $n = m(m-1) + 1$ jobs
- Jobs $1, \dots, n-1 = m(m-1)$ have $t_i = \underline{1}$, job n has $t_n = \underline{m}$

Greedy Schedule:



OPT:



Improving Greedy

Bad case for the greedy algorithm:
One large job in the end can destroy everything

Idea: assign large jobs first

Modified Greedy Algorithm:

1. Sort jobs by decreasing length s.t. $t_1 \geq t_2 \geq \cdots \geq t_n$
2. Apply the greedy algorithm as before (in the sorted order)

Lemma: If $n > m$: $T^* \geq t_m + t_{m+1} \geq 2t_{m+1}$

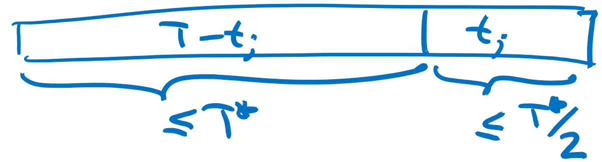
Proof:

- Two of the first $m+1$ jobs need to be scheduled on the same machine
- Jobs m and $m+1$ are the shortest of these jobs

Analysis of the Modified Greedy Alg. $\leq \frac{4}{3}$

Theorem: The modified algorithm has approximation ratio $\leq {}^3/_2$.

Proof:

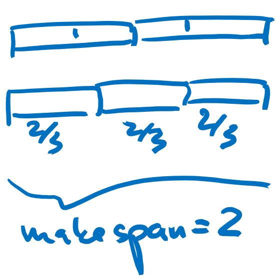


because $T^* \geq 2t_{m+1}$

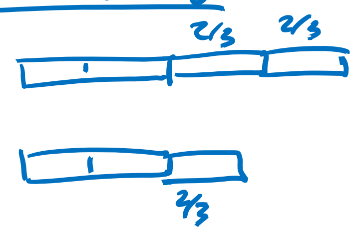
- We show that $T \leq {}^3/_2 \cdot T^*$
- As before, we consider the machine M_i with $T_i = T$
- Job j (of length t_j) is the last one scheduled on machine M_i
- If j is the only job on M_i, we have $T = T^*$
- Otherwise, we have $j \geq m + 1$
  - The first m jobs are assigned to m distinct machines

jobs: $1, 1, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}$, $m=2$

OPT



modified greedy



makespan $= 7/3$

approx. ratio $\geq \frac{7/3}{2} = \frac{7}{6}$

Metric TSP

Input:

- Set V of n nodes (points, cities, locations, sites)
- Distance function $d: V \times V \to \mathbb{R}$, i.e., $d(u, v)$ is dist from u to v
- Distances define a metric on V:

$$d(u, v) = d(v, u) \geq 0, \qquad d(u, v) = 0 \Leftrightarrow u = v$$

$$\forall u, v, w \in V : d(u, v) \leq d(u, w) + d(w, v)$$ ← Δ-inequality

Solution:

- Ordering/permutation $v_1, v_2, \dots, v_n$ of the vertices
- Length of TSP path: $\sum_{i=1}^{n-1} d(v_i, v_{i+1})$
- Length of TSP tour: $d(v_1, v_n) + \sum_{i=1}^{n-1} d(v_i, v_{i+1})$

Goal:

- Minimize length of TSP path or TSP tour

Metric TSP

- The problem is NP-hard
- We have seen that the greedy algorithm (always going to the nearest unvisited node) gives an $O(\log n)$-approximation
- Can we get a constant approximation ratio?
- We will see that we can…

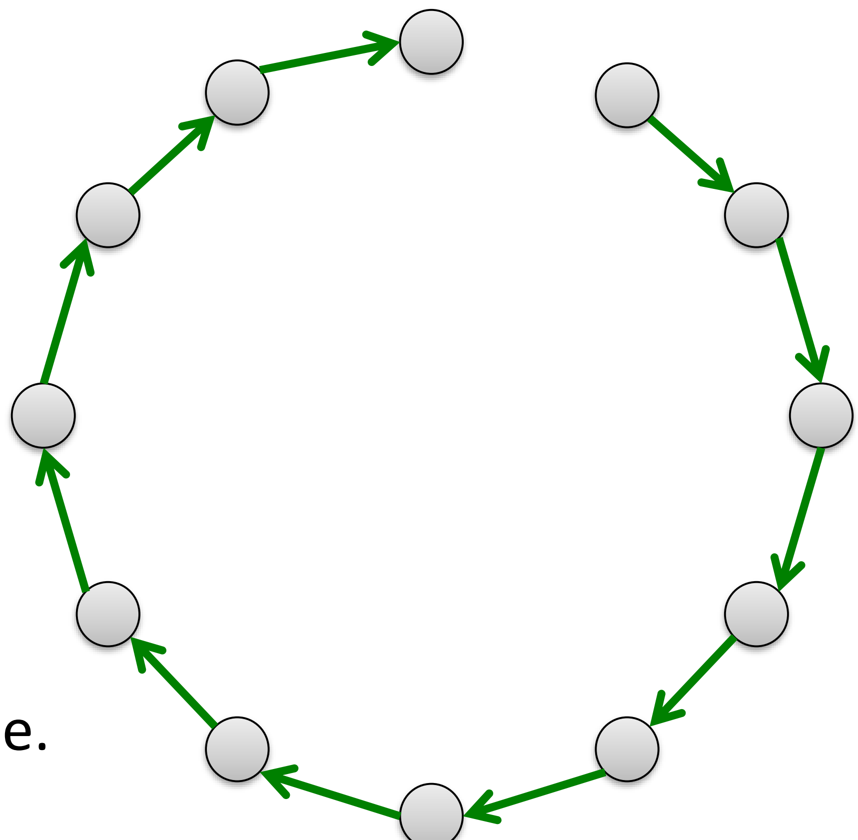
TSP and MST

Claim: The length of an optimal TSP path is lower bounded by the weight of a minimum spanning tree

Proof:

- A TSP path is a spanning tree, it's length is the weight of the tree

$w(MST) \leq TSP_{PATH} \leq TSP_{TOUR}$

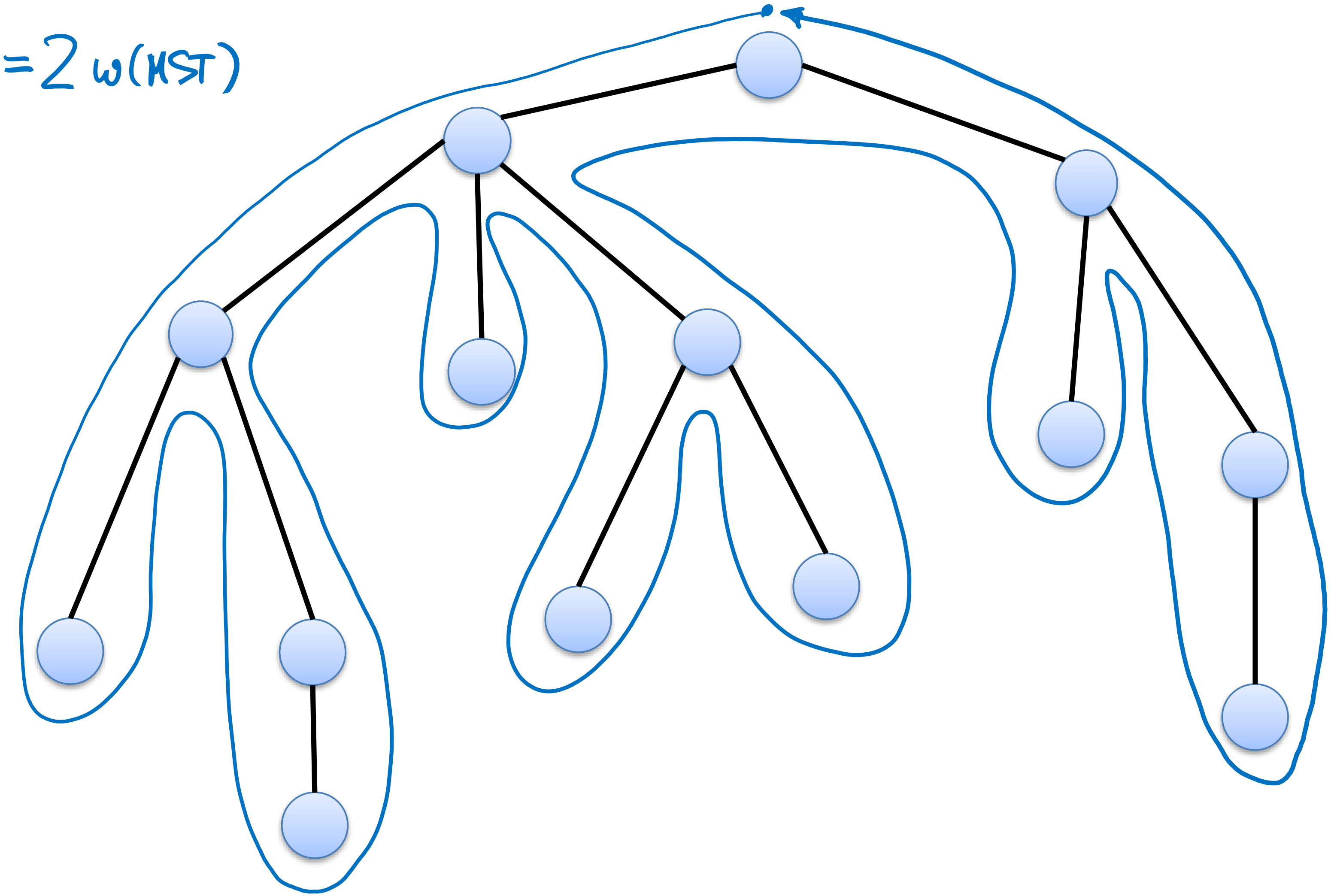


Corollary: Since an optimal TSP tour is longer than an optimal TSP path, the length of an optimal TSP tour is also lower bounded by the weight of a minimum spanning tree.

The MST Tour

Walk around the MST…

length of walk $= 2\, w(MST)$

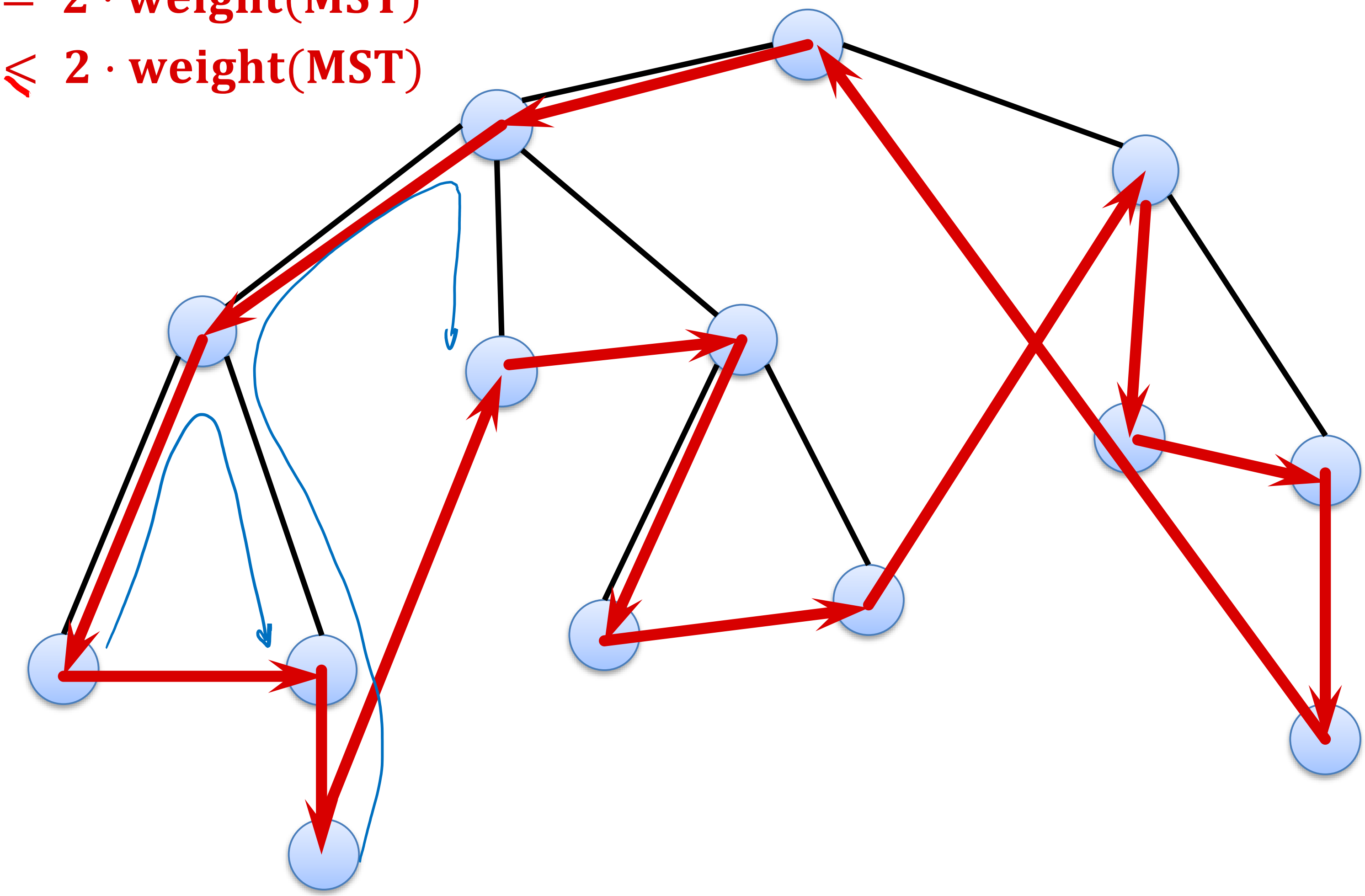


The MST Tour

Walk around the MST…

Cost (walk) $= 2 \cdot \mathbf{weight(MST)}$

Cost (tour) $\leqslant 2 \cdot \mathbf{weight(MST)}$



Approximation Ratio of MST Tour

Theorem: The MST TSP tour gives a 2-approximation for the metric TSP problem.

Proof:

- Triangle inequality → length of tour is at most $2 \cdot \text{weight(MST)}$
- We have seen that $\text{weight(MST)} < \text{opt. tour length}$

Can we do even better?

Metric TSP Subproblems

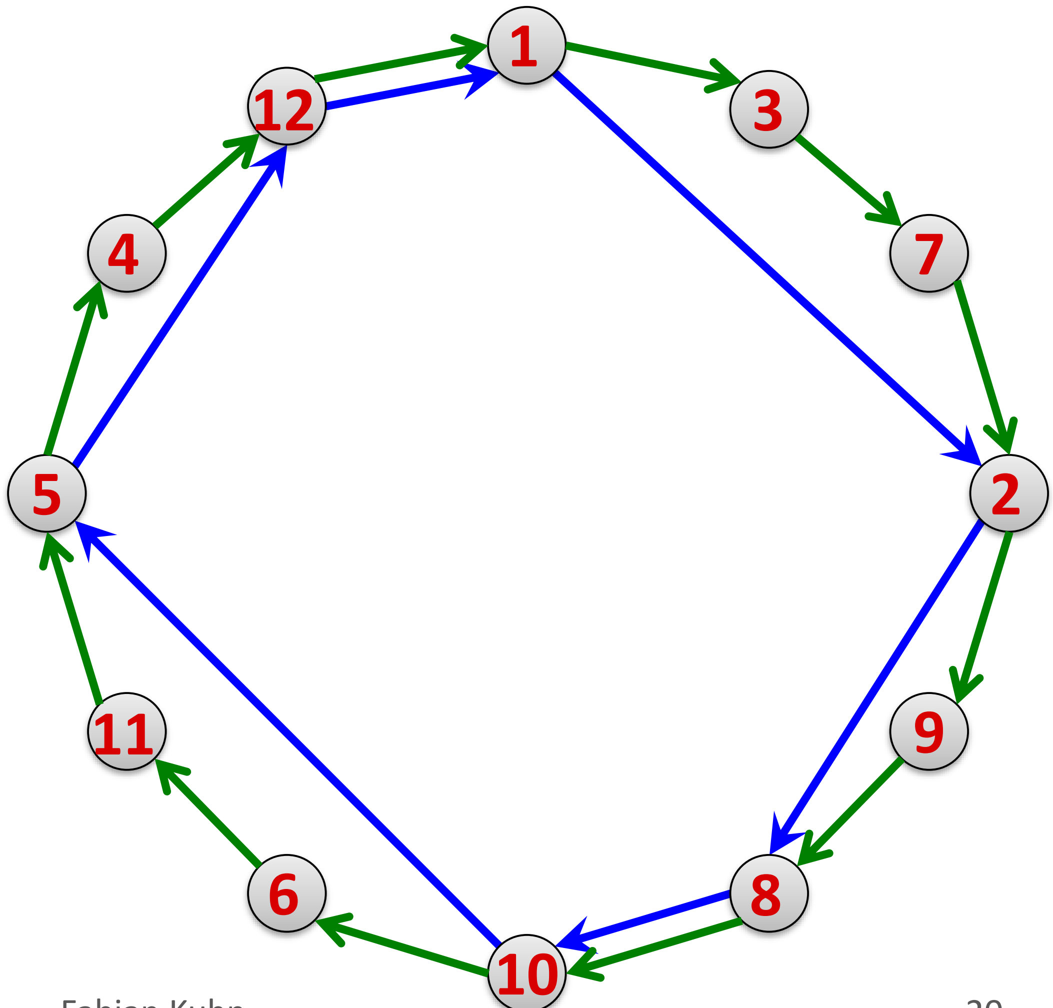
Claim: Given a metric (V, d) and (V', d) for $V' \subseteq V$, the optimal TSP path/tour of (V', d) is at most as large as the optimal TSP path/tour of (V, d).

Optimal TSP tour of nodes $1, 2, \dots, 12$

Induced TSP tour for nodes $1, 2, 5, 8, 10, 12$

blue tour $\leq$ green tour

triangle ineq.



TSP and Matching

- Consider a metric TSP instance (V, d) with an even number of nodes $|V|$

- Recall that a perfect matching is a matching $M \subseteq V \times V$ such that every node of V is incident to an edge of M.

- Because $|V|$ is even and because in a metric TSP, there is an edge between any two nodes $u, v \in V$, any partition of V into $|V|/2$ pairs is a perfect matching.

- The weight of a matching M is the sum of the distances represented by all edges in M:
$$w(M) = \sum_{\{u,v\} \in M} d(u, v)$$

TSP and Matching

Lemma: Assume we are given a TSP instance (V, d) with an even number of nodes. The length of an optimal TSP tour of (V, d) is at least twice the weight of a minimum weight perfect matching of (V, d).

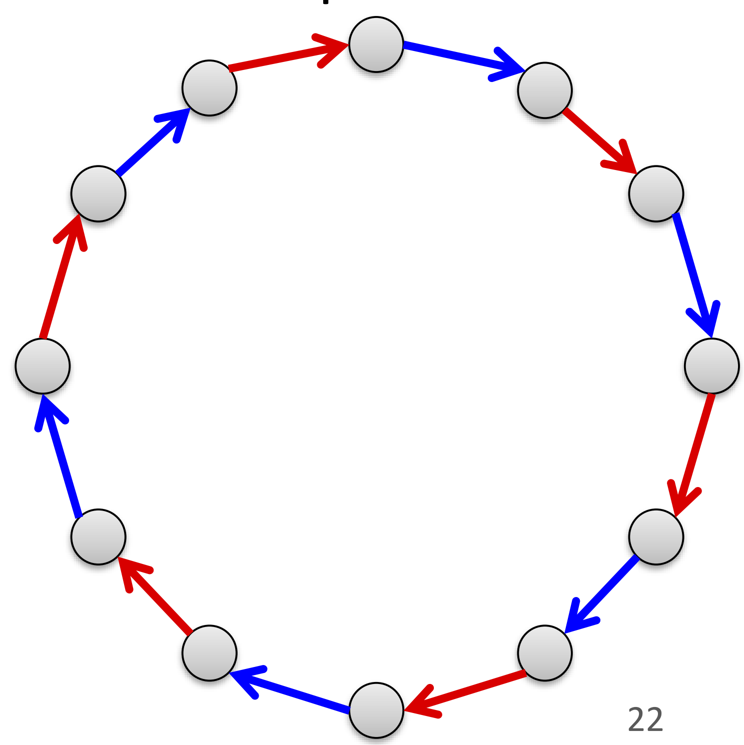
Proof:

- The edges of a TSP tour can be partitioned into 2 perfect matchings

TSP_{OPT} = red + blue

$\geq$ $\geq$

weight of a min. weight perf. matching



Minimum Weight Perfect Matching

Claim: If $|V|$ is even, a minimum weight perfect matching of (V, d) can be computed in polynomial time

Proof Sketch:

- We have seen that a minimum weight perfect matching in a complete bipartite graph can be computed in polynomial time
- With a more complicated algorithm, also a minimum weight perfect matching in complete (non-bipartite) graphs can be computed in polynomial time
- The algorithm uses similar ideas as the bipartite weighted matching algorithm and it uses the Blossom algorithm as a subroutine

Algorithm Outline

Problem of MST algorithm:

- Every edge has to be visited twice

Goal:

- Get a graph on which every edge only has to be visited once (and where still the total edge weight is small compared to an optimal TSP tour)

Euler Tours:

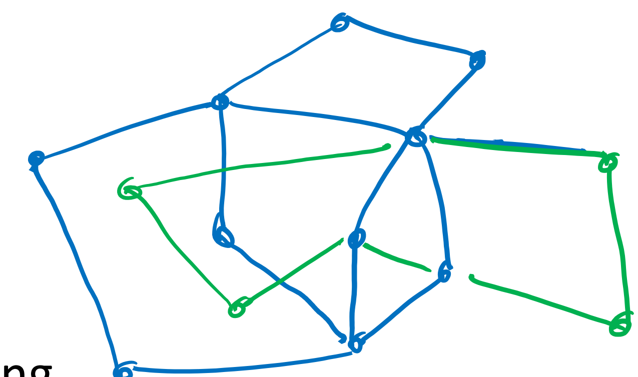
- A tour that visits each edge of a graph exactly once is called an Euler tour
- An Euler tour in a (multi-)graph exists if and only if every node of the graph has even degree
- That's definitely not true for a tree, but can we modify our MST suitably?

Euler Tour

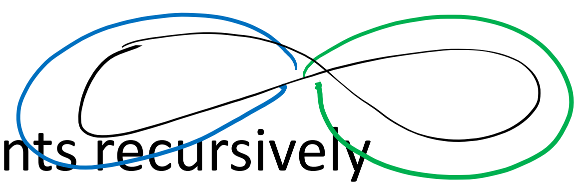
Theorem: A connected (multi-)graph G has an Euler tour if and only if every node of G has even degree.

Proof:

- If G has an odd degree node, it clearly cannot have an Euler tour
- If G has only even degree nodes, a tour can be found recursively:

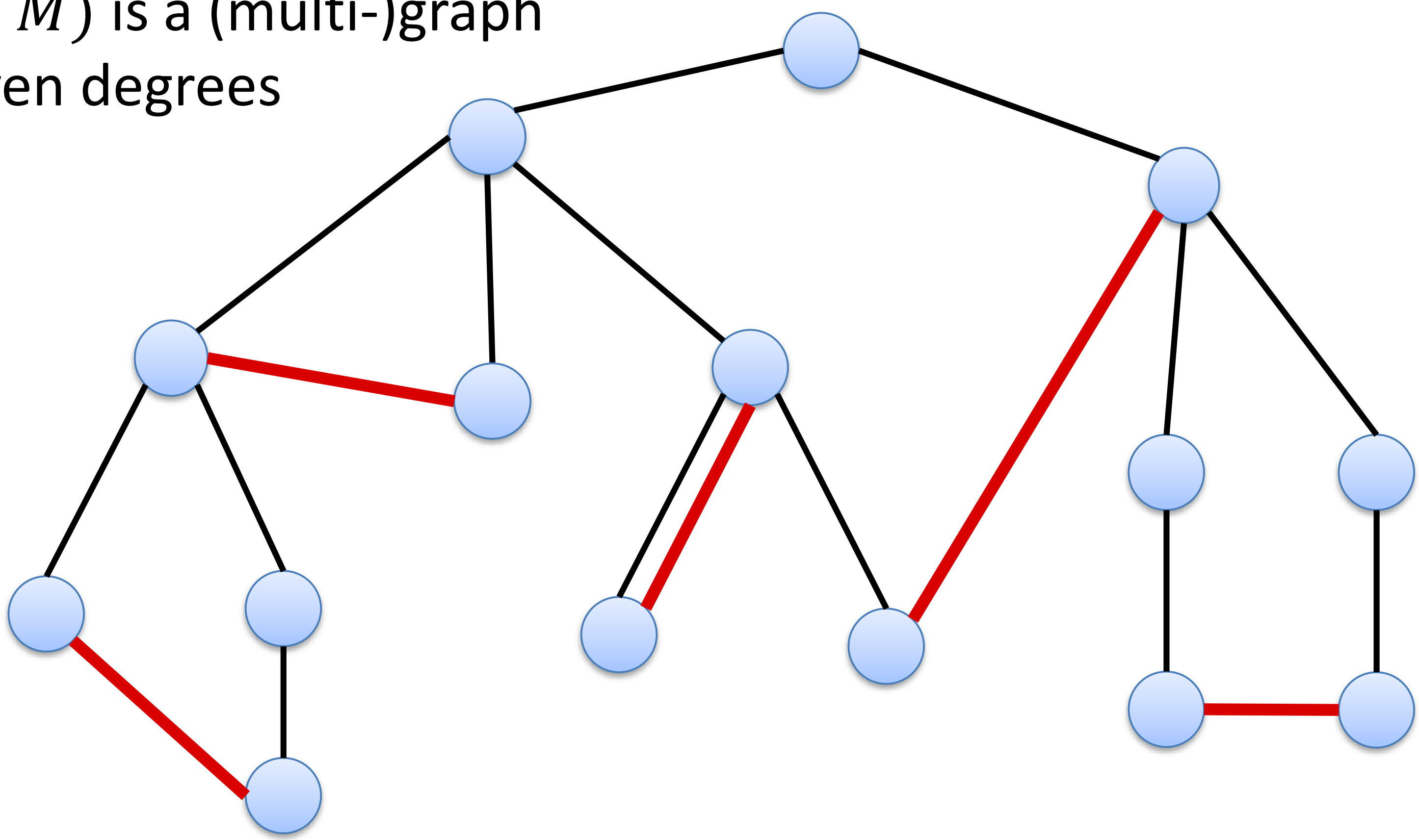


1. Start at some node
2. As long as possible, follow an unvisited edge
   - Gives a partial tour, the remaining graph still has even degree
3. Solve problem on remaining components recursively
4. Merge the obtained tours into one tour that visits all edges



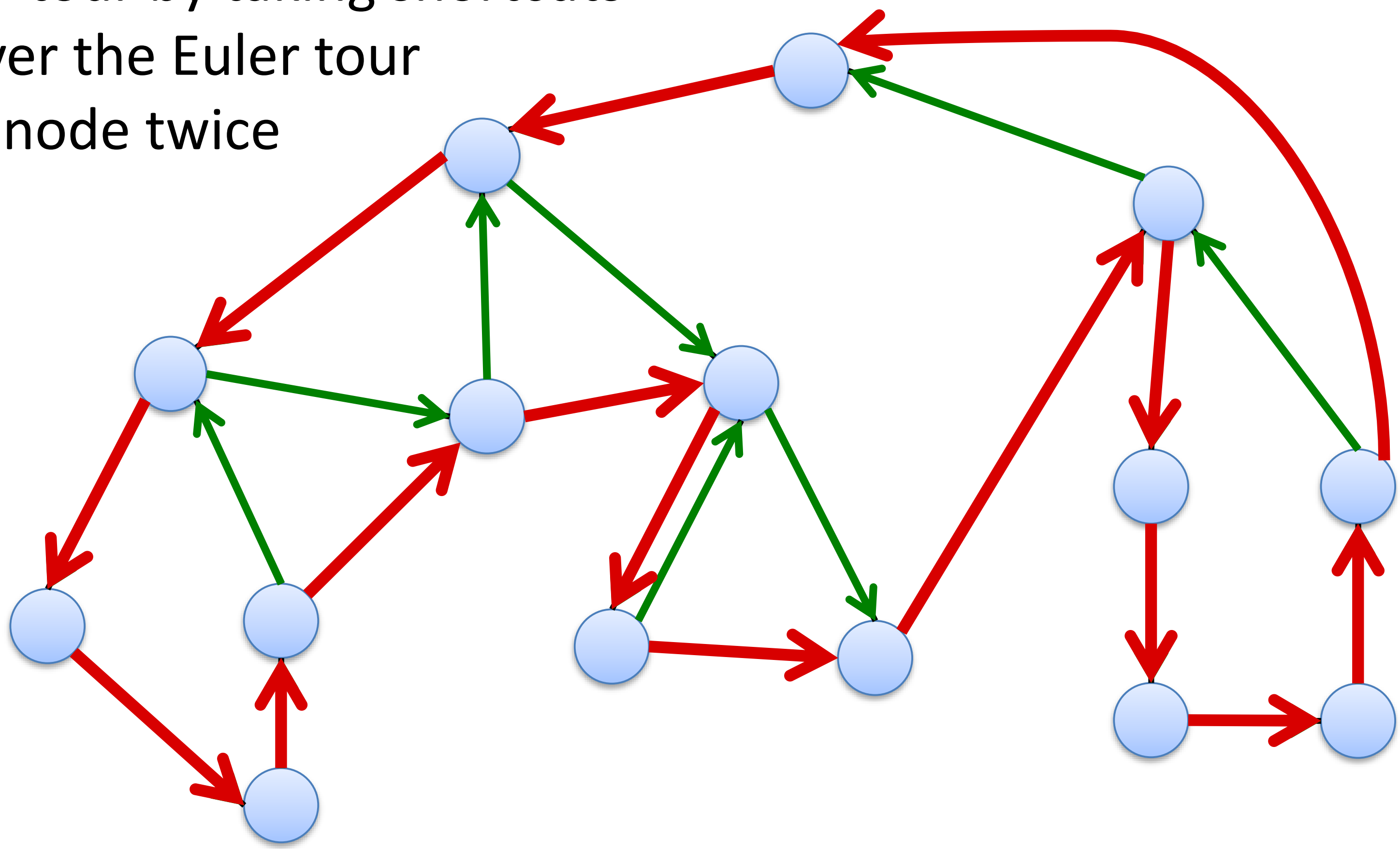
TSP Algorithm

1. Compute MST T
2. V_{odd}: nodes that have an odd degree in T ($|V_{\text{odd}}|$ is even)
3. Compute min weight perfect matching M of (V_{odd}, d)
4. $(V, T \cup M)$ is a (multi-)graph with even degrees



TSP Algorithm

5. Compute Euler tour on $(V, T \cup M)$
6. Total length of Euler tour $\leq \frac{3}{2} \cdot \mathbf{TSP_{OPT}}$
7. Get TSP tour by taking shortcuts wherever the Euler tour visits a node twice



TSP Algorithm

- The described algorithm is by Christofides

Theorem: The Christofides algorithm achieves an approximation ratio of at most $^3/_2$.

Proof:

- The length of the Euler tour is $\leq {}^3/_2 \cdot \mathrm{TSP}_{\mathrm{OPT}}$
- Because of the triangle inequality, taking shortcuts can only make the tour shorter

Set Cover

Input:

$(X, \mathcal{S})$: set system

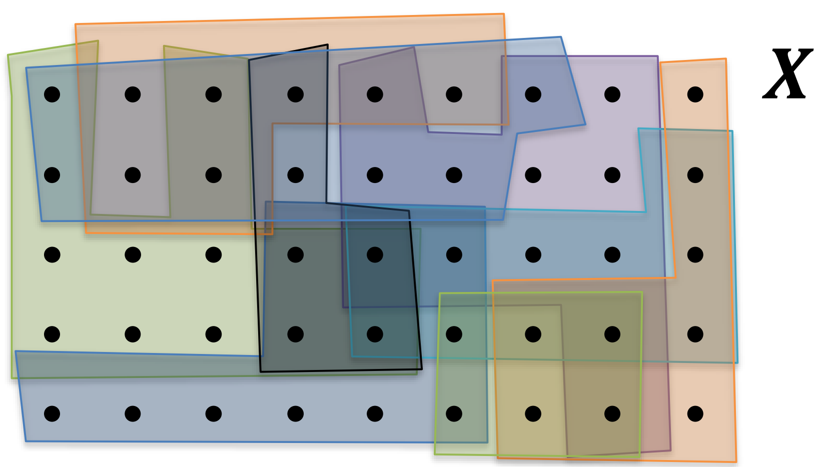
- A set of elements X and a collection $\mathcal{S}$ of subsets X, i.e., $\mathcal{S} \subseteq 2^X$
  – such that $\bigcup_{S \in \mathcal{S}} S = X$

Set Cover:

- A set cover $\mathcal{C}$ of $(X, \mathcal{S})$ is a subset of the sets $\mathcal{S}$ which covers X:

$$\bigcup_{S \in \mathcal{C}} S = X$$

Example:



Minimum (Weighted) Set Cover

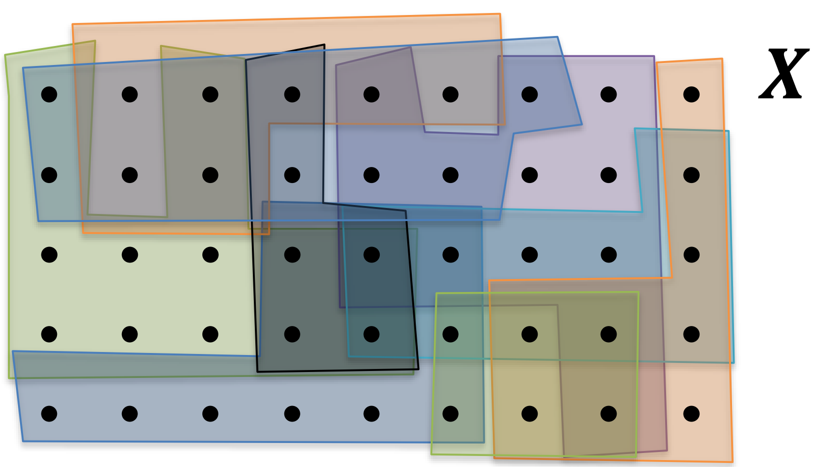
Minimum Set Cover:

- **Goal:** Find a set cover $\mathcal{C}$ of smallest possible size
  - i.e., over X with as few sets as possible

Minimum Weighted Set Cover:

- Each set $S \in \mathcal{S}$ has a weight $w_S > 0$
- **Goal:** Find a set cover $\mathcal{C}$ of minimum weight

Example:

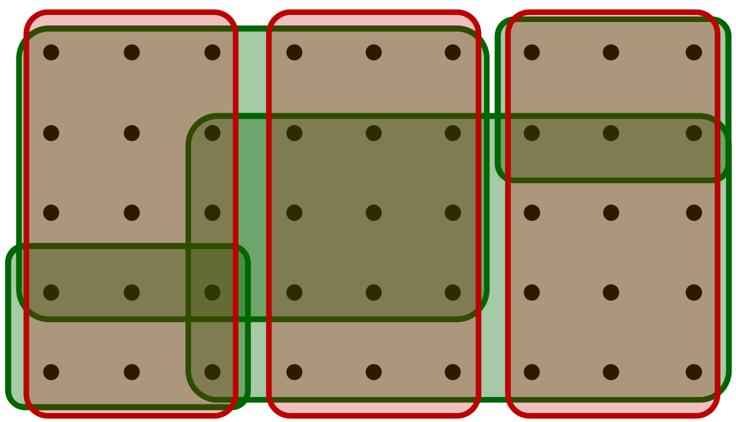


Minimum Set Cover: Greedy Algorithm

Greedy Set Cover Algorithm:

- Start with $\mathcal{C} = \emptyset$
- In each step, add set $S \in \mathcal{S} \setminus \mathcal{C}$ to $\mathcal{C}$ s.t. S covers as many uncovered elements as possible

Example:



Weighted Set Cover: Greedy Algorithm

Greedy Weighted Set Cover Algorithm:

- Start with $\mathcal{C} = \emptyset$
- In each step, add set $S \in \mathcal{S} \setminus \mathcal{C}$ with the best weight per newly covered element ratio (set with best efficiency):

$$S = \arg\min_{S \in \mathcal{S} \setminus \mathcal{C}} \frac{w_S}{\left|S \setminus \bigcup_{T \in \mathcal{C}} T\right|}$$

Analysis of Greedy Algorithm:

- Assign a price $p(x)$ to each element $x \in X$:
  The efficiency of the set when covering the element
- If covering x with set S, if partial cover is $\mathcal{C}$ before adding S:

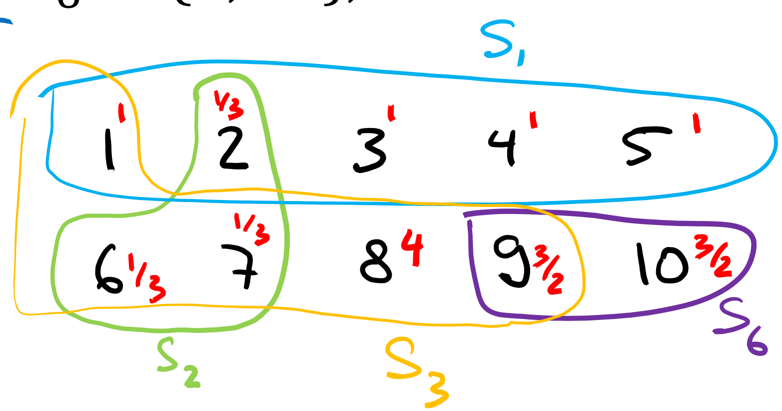
$$p(e) = \frac{w_S}{\left|S \setminus \bigcup_{T \in \mathcal{C}} T\right|}$$

Weighted Set Cover: Greedy Algorithm

Example:

- Universe $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- Sets $\mathcal{S} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$

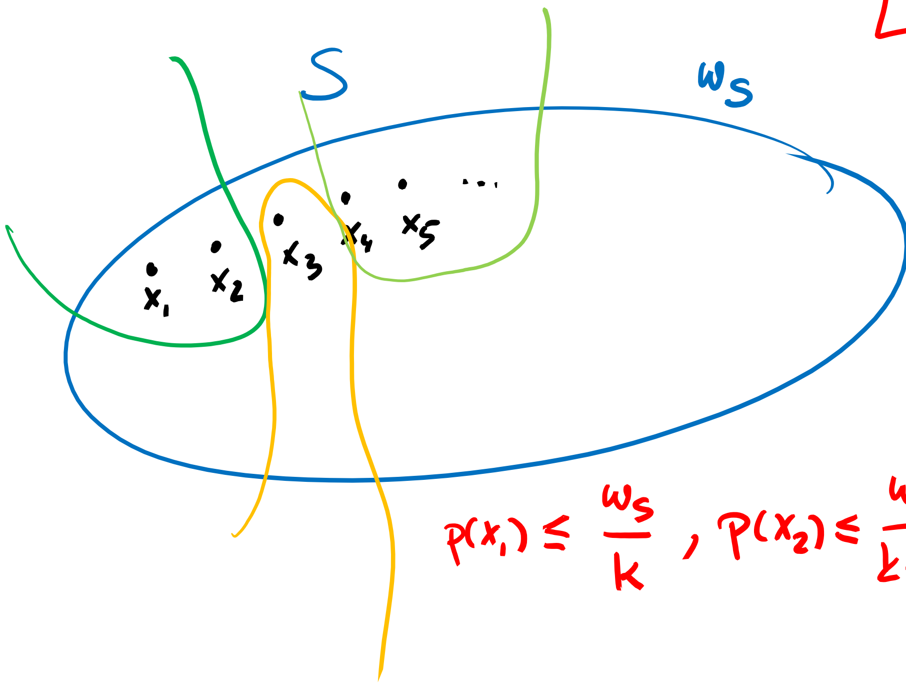
$$\cancel{S_1 = \{1, \cancel{2}, 3, 4, 5\}, \qquad w_{S_1} = 4}$$
$$\cancel{S_2 = \{2, 6, 7\}, \qquad w_{S_2} = 1}$$
$$S_3 = \{\cancel{1}, \cancel{6}, \cancel{7}, 8, \cancel{9}\}, \qquad w_{S_3} = 4$$
$$S_4 = \{\cancel{2}, \cancel{4}, \cancel{7}, \cancel{9}, \cancel{10}\}, \qquad w_{S_4} = 6$$
$$S_5 = \{\cancel{1}, \cancel{3}, \cancel{5}, \cancel{6}, \cancel{7}, 8, \cancel{9}, \cancel{10}\}, \qquad w_{S_5} = 9$$
$$\cancel{S_6 = \{9, 10\}, \qquad w_{S_6} = 3}$$



Weighted Set Cover: Greedy Algorithm

Lemma: Consider a set $S = \{x_1, x_2, \dots, x_k\} \in \mathcal{S}$ be a set and assume that the elements are covered in the order $x_1, x_2, \dots, x_k$ by the greedy algorithm (ties broken arbitrarily).

Then, the price of element x_i is at most $\boxed{p(x_i) \le \frac{w_S}{k-i+1}}$



$$\sum_{x \in S} p(x) \le w_S \underbrace{\left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \dots + 1\right)}_{H_k = \ln k + O(1)}$$

$$p(x_1) \le \frac{w_S}{k}, \quad p(x_2) \le \frac{w_S}{k-1}, \quad p(x_3) \le \frac{w_S}{k-2}$$

Weighted Set Cover: Greedy Algorithm

Lemma: Consider a set $S = \{x_1, x_2, \dots, x_k\} \in \mathcal{S}$ be a set and assume that the elements are covered in the order $x_1, x_2, \dots, x_k$ by the greedy algorithm (ties broken arbitrarily).

Then, the price of element x_i is at most $p(x_i) \leq \frac{w_S}{k-i+1}$

Corollary: The total price of a set $S \in \mathcal{S}$ of size $|S| = k$ is

$$\sum_{x \in S} p(x) \leq w_S \cdot H_k, \qquad \text{where } H_k = \sum_{i=1}^{k} \frac{1}{i} \leq 1 + \ln k$$

Weighted Set Cover: Greedy Algorithm

Corollary: The total price of a set $S \in \mathcal{S}$ of size $|S| = k$ is

$$\sum_{x \in S} p(x) \leq w_S \cdot H_k, \qquad \text{where } \; H_k = \sum_{i=1}^{k} \frac{1}{i} \leq 1 + \ln k$$

Theorem: The approximation ratio of the greedy minimum (weighted) set cover algorithm is at most $\boldsymbol{H_s \leq 1 + \ln s}$, where s is the cardinality of the largest set ($s = \max_{S \in \mathcal{S}} |S|$).

OPT :

GREEDY = total price ≤ OPT · H_s

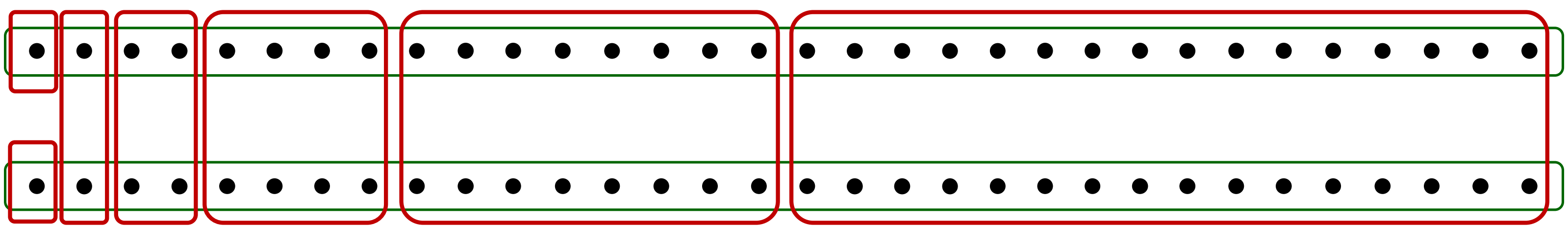
Set Cover Greedy Algorithm

Can we improve this analysis?

No! Even for the unweighted minimum set cover problem, the approximation ratio of the greedy algorithm is $\geq \big(1 - o(1)\big) \cdot \ln s$.

- if s is the size of the largest set... (s can be linear in n)

Let's show that the approximation ratio is at least $\Omega(\log n)$...



$$\mathbf{OPT = 2}$$

$$\mathbf{GREEDY \geq \log_2 n}$$

Set Cover: Better Algorithm? $(1 - f(n)) \cdot \ln n$

$\lim_{n \to \infty} f(n) = 0$

An approximation ratio of $\ln n$ seems not spectacular...

Can we improve the approximation ratio?

$2^{O(n)} = n^{O(\frac{n}{\log n})}$

No, unfortunately not, unless $\mathrm{P} \approx \mathrm{NP}$

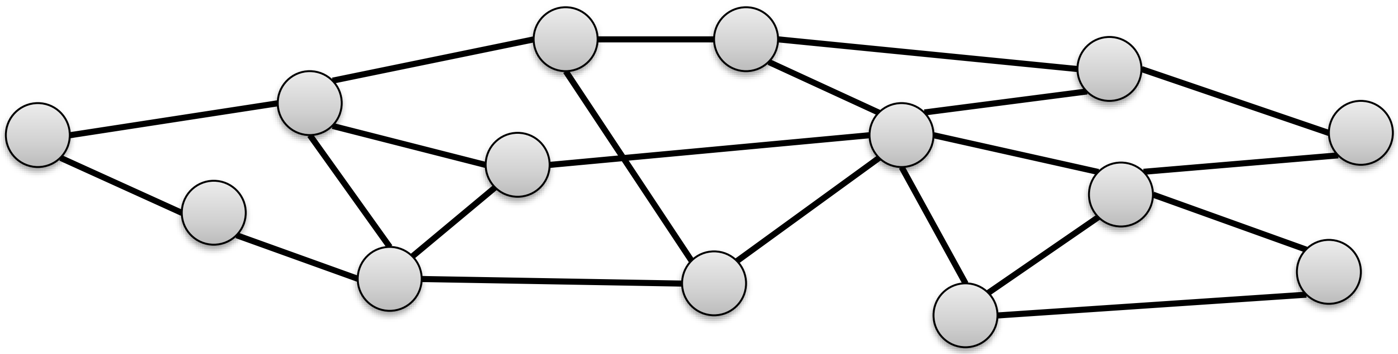
Feige showed that unless NP has deterministic $n^{O(\log \log n)}$-time algorithms, minimum set cover cannot be approximated better than by a factor $(1 - o(1)) \cdot \ln n$ in polynomial time.

- Proof is based on the so-called PCP theorem
  - PCP theorem is one of the main (relatively) recent advancements in theoretical computer science and the major tool to prove approximation hardness lower bounds
  - Shows that every language in NP has certificates of polynomial length that can be checked by a randomized algorithm by only querying a constant number of bits (for any constant error probability)

Set Cover: Special Cases

Vertex Cover: set $S \subseteq V$ of nodes of a graph $G = (V, E)$ such that

$$\forall \{u, v\} \in E, \qquad \{u, v\} \cap S \neq \emptyset.$$



Minimum Vertex Cover:

- Find a vertex cover of minimum cardinality

Minimum Weighted Vertex Cover:

- Each node has a weight
- Find a vertex cover of minimum total weight

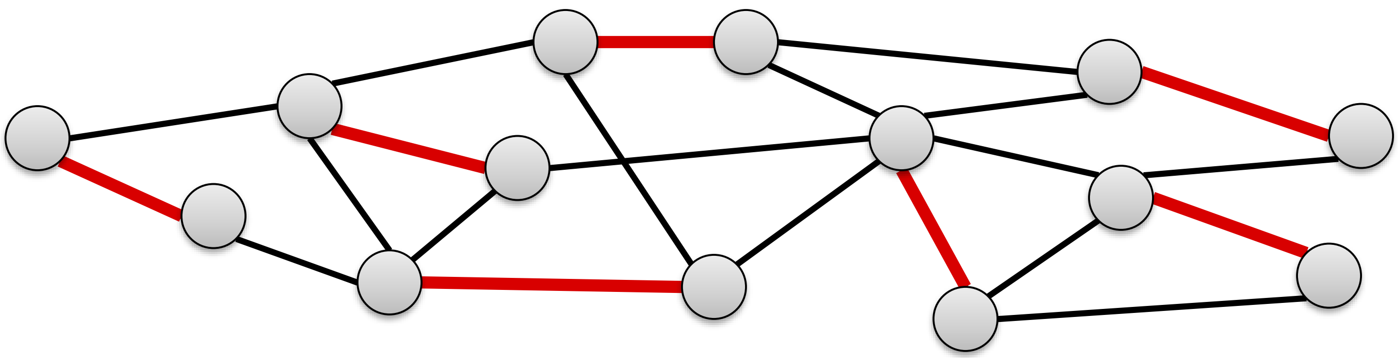
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: $|M| \leq |S|$

Proof:

- At least one node of every edge $\{u, v\} \in M$ is in S
- Needs to be a different node for different edges from M



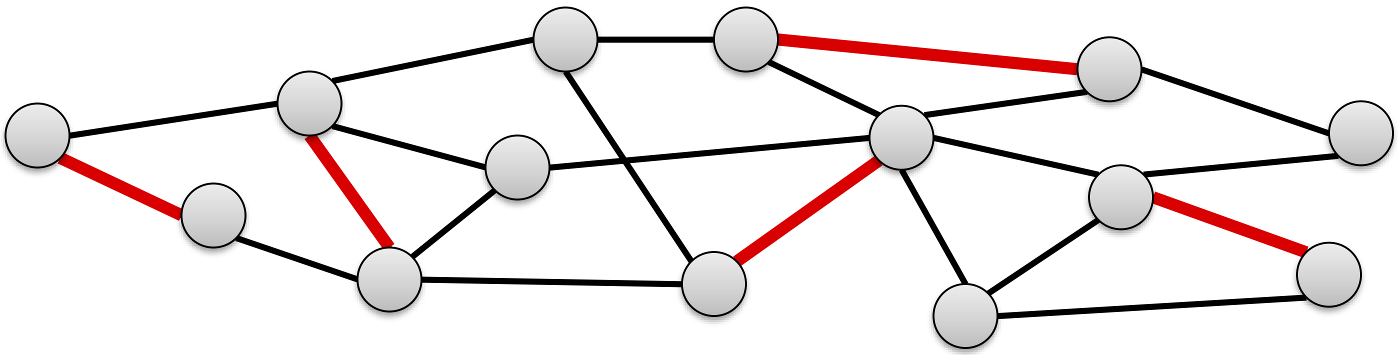
Vertex Cover vs Matching

Consider a matching M and a vertex cover S

Claim: If M is maximal and S is minimum, $|S| \leq 2|M|$

Proof:

- M is maximal: for every edge $\{u, v\} \in E$, either u or v (or both) are matched



- Every edge $e \in E$ is "covered" by at least one matching edge
- Thus, the set of the nodes of all matching edges gives a vertex cover S of size $|S| = 2|M|$.

Maximal Matching Approximation

Theorem: For any maximal matching M and any maximum matching M^*, it holds that

$$|M| \geq \frac{|M^*|}{2}.$$

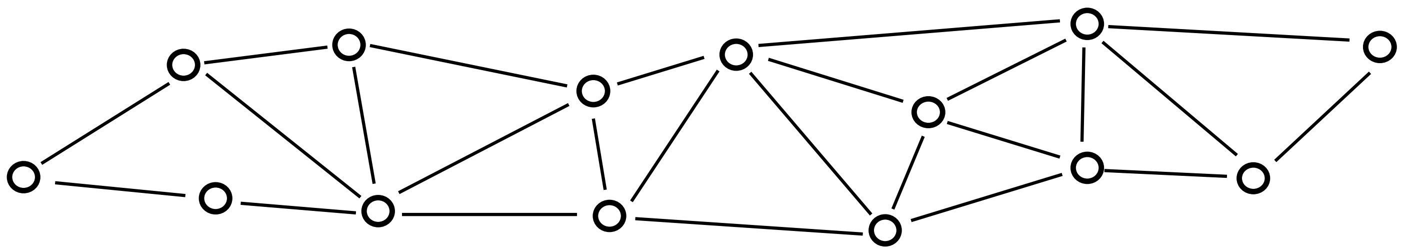
Proof:

Theorem: The set of all matched nodes of a maximal matching M is a vertex cover of size at most twice the size of a min. vertex cover.

Set Cover: Special Cases

Dominating Set:

Given a graph $G = (V, E)$, a dominating set $S \subseteq V$ is a subset of the nodes V of G such that for all nodes $u \in V \setminus S$, there is a neighbor $v \in S$.



Minimum Hitting Set

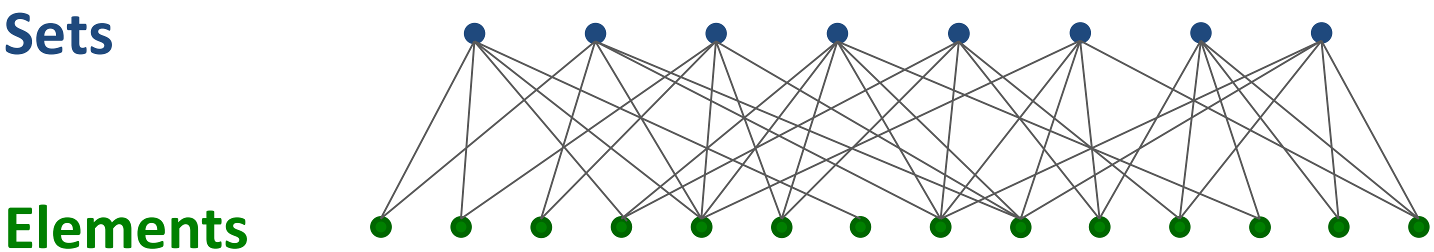
Given: Set of elements X and collection of subsets $\mathcal{S} \subseteq 2^X$

- Sets cover X: $\bigcup_{S \in \mathcal{S}} S = X$

Goal: Find a min. cardinality subset $H \subseteq X$ of elements such that

$$\forall S \in \mathcal{S} : S \cap H \neq \emptyset$$

Problem is equivalent to min. set cover with roles of sets and elements interchanged



Knapsack

- n items $1, \dots, n$, each item has weight $w_i > 0$ and value $v_i > 0$
- Knapsack (bag) of capacity W

- Goal: pack items into knapsack such that total weight is at most W and total value is maximized:

$$\max \sum_{i \in S} v_i$$

$$\text{s.t. } S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} w_i \leq W$$

- E.g.: jobs of length w_i and value v_i, server available for W time units, try to execute a set of jobs that maximizes the total value

Knapsack: Dynamic Programming Alg.

We have shown:

- If all item weights w_i are integers, using dynamic programming, the knapsack problem can be solved in time $O(nW)$
- If all values v_i are integers, there is another dynamic progr. algorithm that runs in time $O(n^2 V)$, where V is the max. value.

Knapsack: Dynamic Programming Alg.

We have shown:

- If all item weights w_i are integers, using dynamic programming, the knapsack problem can be solved in time $O(nW)$
- If all values v_i are integers, there is another dynamic progr. algorithm that runs in time $O(n^2V)$, where V is the max. value.

Problems:

- If W and V are large, the algorithms are not polynomial in n
- If the values or weights are not integers, things are even worse (and in general, the algorithms cannot even be applied at all)

Idea:

- Can we adapt one of the algorithms to at least compute an approximate solution?

Approximation Algorithm

- The algorithm has a parameter $\varepsilon > 0$
- We assume that each item alone fits into the knapsack
- We define:

$$V := \max_{1 \le i \le n} v_i, \qquad \forall i: \widehat{v_i} := \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil, \qquad \widehat{V} := \max_{1 \le i \le n} \widehat{v_i}$$

- We solve the problem with integer values $\widehat{v_i}$ and weights w_i using dynamic programming in time $O(n^2 \cdot \widehat{V})$
- If solution value $< V$, we take item with value V instead

Theorem: The described algorithm runs in time $O(n^3/\varepsilon)$.

Proof:

$$\widehat{V} = \max_{1 \le i \le n} \widehat{v_i} = \max_{1 \le i \le n} \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil = \left\lceil \frac{Vn}{\varepsilon V} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$$

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- Define the set of all feasible solutions (subsets of $[n]$)

$$\mathcal{S} := \left\{ S \subseteq \{1, \dots, n\} : \sum_{i \in S} w_i \leq W \right\}$$

- $v(S)$: value of solution S w.r.t. values $v_1, v_2, \dots$
  $\hat{v}(S)$: value of solution S w.r.t. values $\hat{v}_1, \hat{v}_2, \dots$
- S^*: an optimal solution w.r.t. values $v_1, v_2, \dots$
  $\hat{S}$: an optimal solution w.r.t. values $\hat{v}_1, \hat{v}_2, \dots$

- Weights are not changed at all, hence, $\hat{S}$ is a feasible solution

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- We have

$$v(S^*) = \sum_{i \in S^*} v_i = \max_{S \in \mathcal{S}} \sum_{i \in S} v_i \,,$$

$$\hat{v}\big(\hat{S}\big) \; = \sum_{i \in \hat{S}} \hat{v}_i = \max_{S \in \mathcal{S}} \sum_{S \in \mathcal{S}} \widehat{v_i}$$

- Because every item fits into the knapsack, we have

$$\forall i \in \{1, \dots, n\}\colon\; v_i \le V \le \sum_{j \in S^*} v_j$$

- Also: $\widehat{v_i} = \left\lceil \frac{v_i n}{\varepsilon V} \right\rceil \;\Longrightarrow\; v_i \le \frac{\varepsilon V}{n} \cdot \widehat{v_i},$ and $\widehat{v_i} \le \frac{v_i n}{\varepsilon V} + 1$

Approximation Algorithm

Theorem: The approximation algorithm computes a feasible solution with approximation ratio at least $1 - \varepsilon$.

Proof:

- We have
$$v(S^*) = \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in S^*} \widehat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \widehat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \left(1 + \frac{v_i n}{\varepsilon V}\right)$$
- Therefore
$$v(S^*) = \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot \left|\hat{S}\right| + \sum_{i \in \hat{S}} v_i \leq \varepsilon V + v(\hat{S})$$
- We have $v(S^*) \geq V$ and therefore
$$\boldsymbol{(1 - \varepsilon) \cdot v(S^*) \leq v(\widehat{S})}$$

Approximation Schemes

- For every parameter $\varepsilon > 0$, the knapsack algorithm computes a $(1+\varepsilon)$-approximation in time $O(n^3/\varepsilon)$.
- For every fixed ε, we therefore get a polynomial time approximation algorithm
- An algorithm that computes an $(1+\varepsilon)$-approximation for every $\varepsilon > 0$ is called an approximation scheme.
- If the running time is polynomial for every fixed ε, we say that the algorithm is a polynomial time approximation scheme (PTAS)
- If the running time is also polynomial in $1/\varepsilon$, the algorithm is a fully polynomial time approximation scheme (FPTAS)
- Thus, the described alg. is an FPTAS for the knapsack problem