

# Approximation Algorithms

Thomas Erlebach



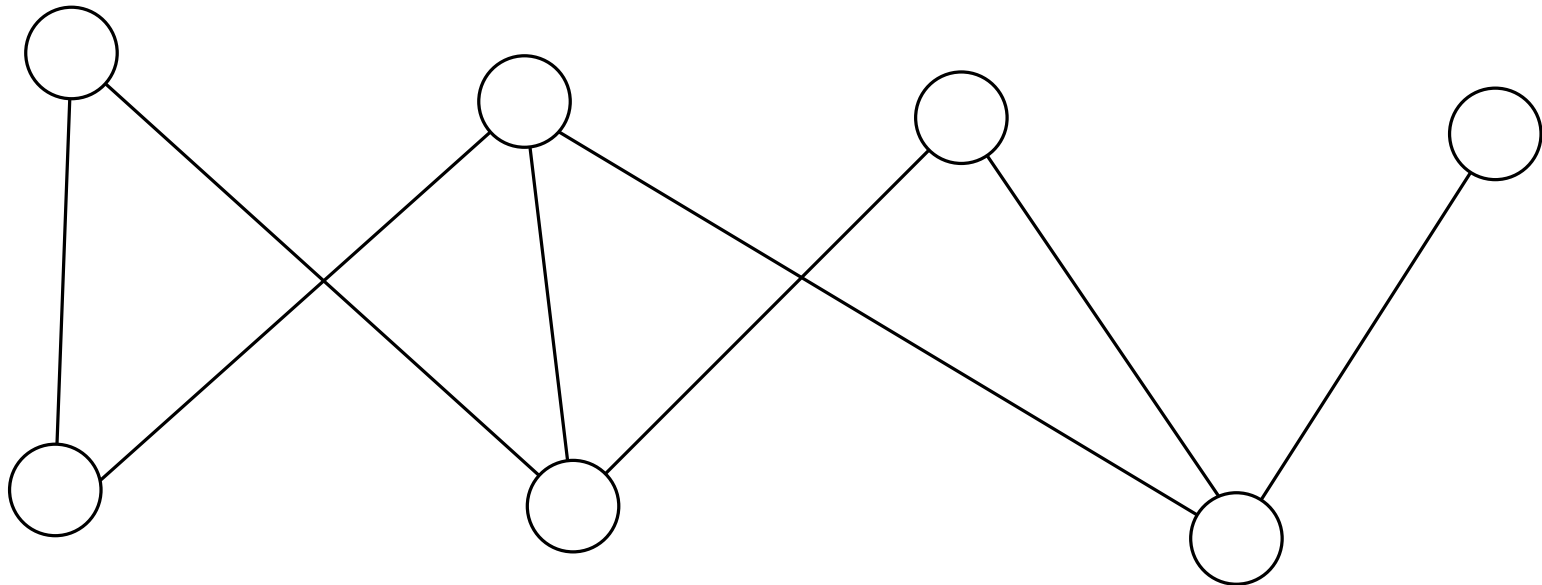
University of  
**Leicester**

# Optimisation Problems

- An **optimisation problem** is specified by:
  - A set of **instances** of the problem.
  - For each instance  $I$ , there is a set  $S_I$  of **feasible solutions**.
  - There is an **objective function** that assigns a value (objective value) to each solution  $s \in S_I$  for instance  $I$ .
  - The goal is to find a feasible solution that maximises (**maximisation problem**) or minimises (**minimisation problem**) the objective value.

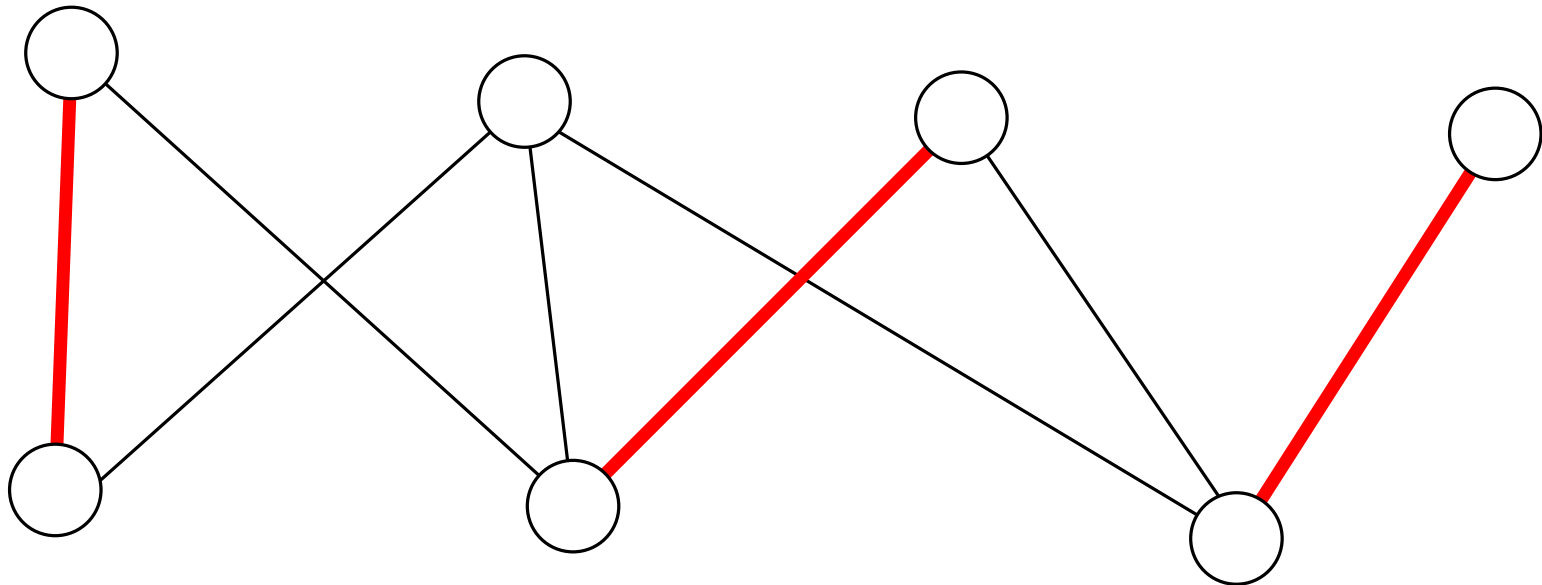
# Example: Maximum Matching

- Instance: undirected graph  $G = (V, E)$
- Feasible solution: a subset  $M \subseteq E$  such that no two edges in  $M$  share an endpoint
- Objective value: size of  $M$ , denoted  $|M|$
- Goal: Maximise  $|M|$



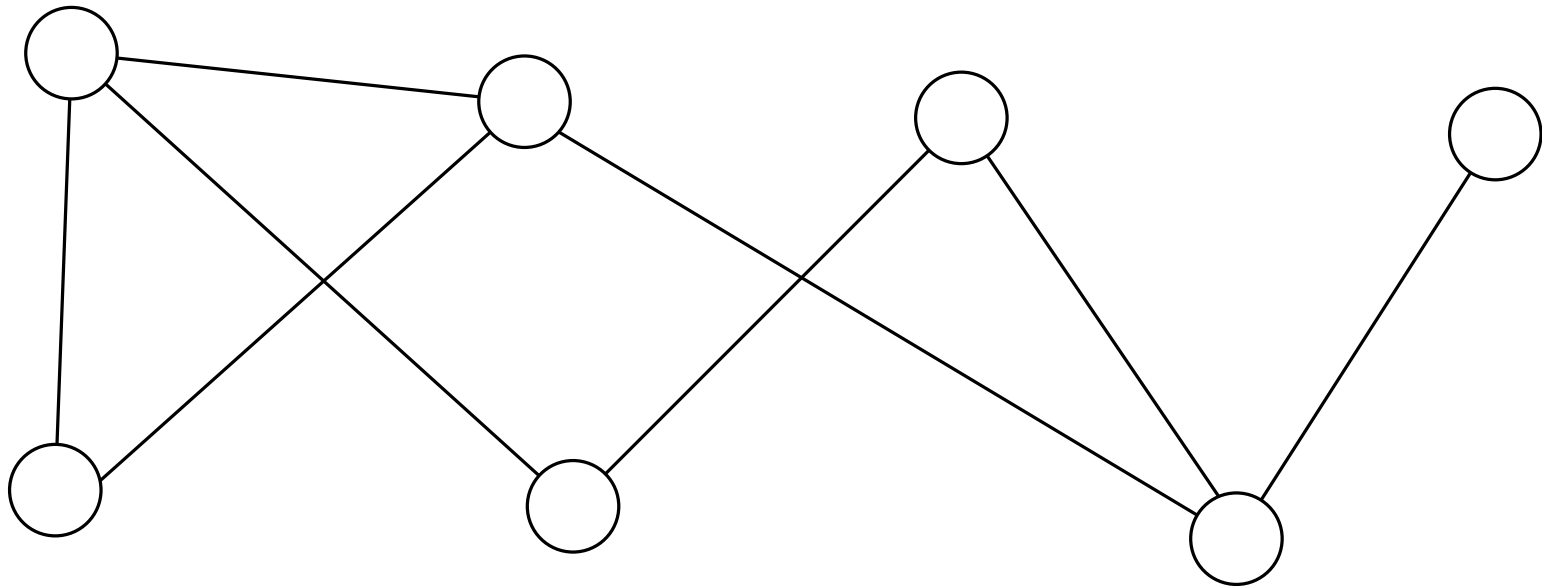
# Example: Maximum Matching

- Instance: undirected graph  $G = (V, E)$
- Feasible solution: a subset  $M \subseteq E$  such that no two edges in  $M$  share an endpoint
- Objective value: size of  $M$ , denoted  $|M|$
- Goal: Maximise  $|M|$



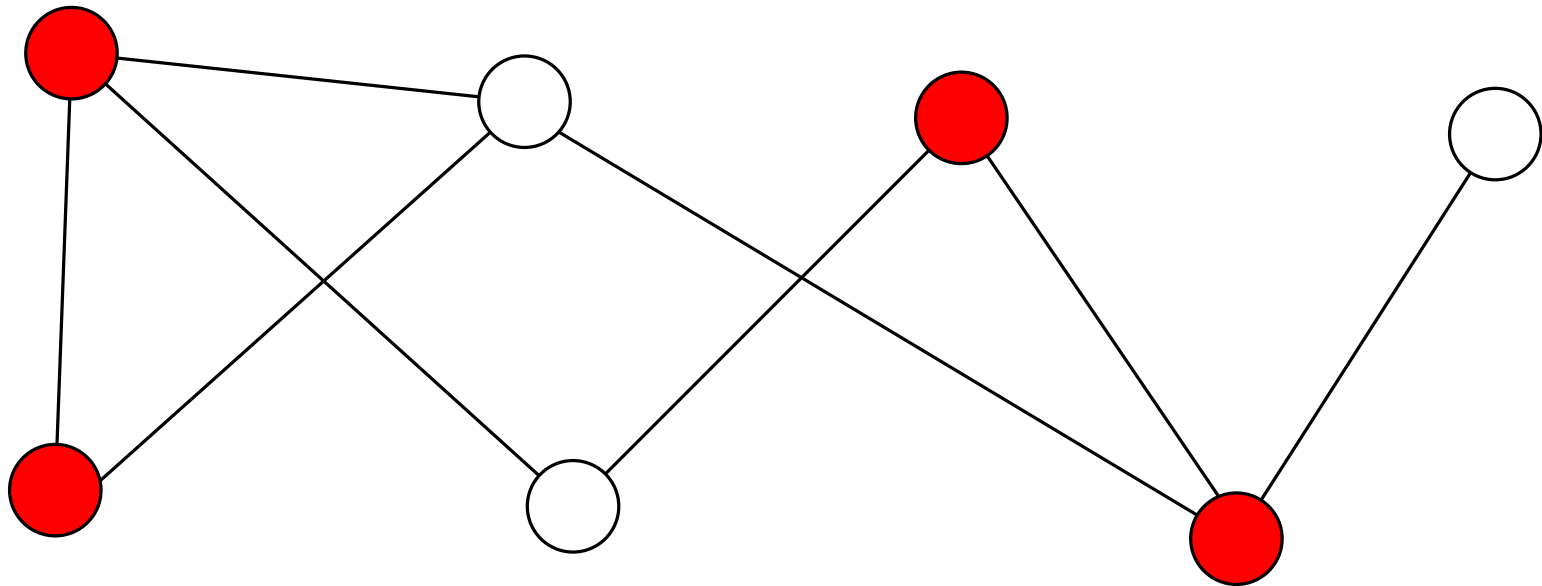
# Example: Min Vertex Cover

- Instance: undirected graph  $G = (V, E)$
- Feasible solution: a subset  $C \subseteq V$  such that every edge in  $E$  has at least one endpoint in  $C$
- Objective value:  $|C|$  (size of  $C$ )
- Goal: Minimise  $|C|$



# Example: Min Vertex Cover

- Instance: undirected graph  $G = (V, E)$
- Feasible solution: a subset  $C \subseteq V$  such that every edge in  $E$  has at least one endpoint in  $C$
- Objective value:  $|C|$  (size of  $C$ )
- Goal: Minimise  $|C|$



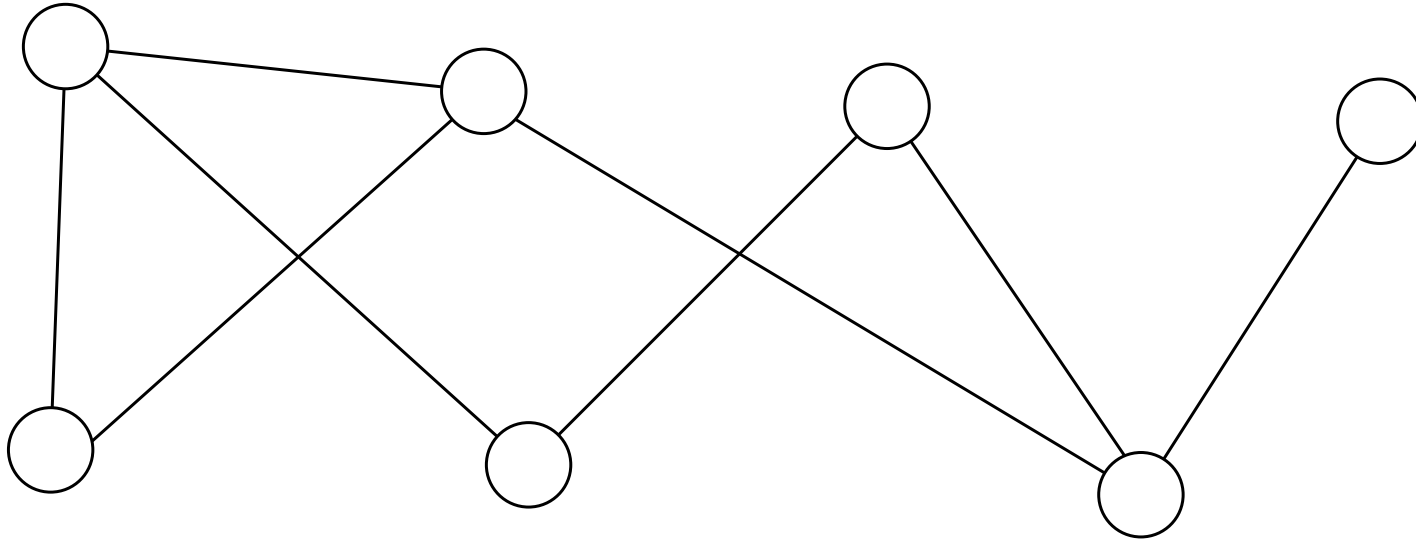
# Polynomial-Time Algorithms

- We are generally interested in algorithms whose running-time is **polynomial in the size of the input**.
- For some optimisation problems, there is a **polynomial-time algorithm** that computes an optimal solution for every given instance.
  - **Example: Maximum Matching**  
 $O(\sqrt{n} \cdot m)$  time algorithm by Micali and Vazirani (1980), where  $n = |V|$  and  $m = |E|$
- Many other optimisation problems are **NP-hard**, i.e. there cannot be a polynomial-time algorithm that computes an optimal solution for every given instance unless  $P = NP$ .
  - **Example: Min Vertex Cover**

# Approximation Algorithms

- One approach to tackle NP-hard optimisation problems is to consider **approximation algorithms**.
- An algorithm for an optimisation problem is called a  **$\rho$ -approximation algorithm** ( $\rho \geq 1$ ) if it
  - runs in **polynomial time**, and
  - always outputs a feasible solution whose objective value is **at most a factor of  $\rho$  away from the optimum objective value**.
- For a given instance  $I$ ,
  - $\text{OPT}(I)$  or  $\text{OPT}$  denotes the optimal objective value,
  - $A(I)$  or  $A$  denotes the objective value of the solution computed by algorithm  $A$ .

# Min Vertex Cover



- How can we find an approximation algorithm for Min Vertex Cover?
  - We need to design an algorithm  $A$  that takes a graph  $G$  as input and outputs a vertex cover.
  - We need to prove that the vertex cover produced by the algorithm is at most  $\rho$  times as large as the optimal vertex cover, **for any given graph  $G$ .**

# Attempt 1

- Largest-Degree-First Greedy Algorithm:

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

$v :=$  a vertex in  $V$  that covers the largest number  
    of edges that are not yet covered by  $C;$

    add  $v$  to  $C;$

**done;**

**return**  $C;$

# Attempt 2

## ● Matching-Based Algorithm

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

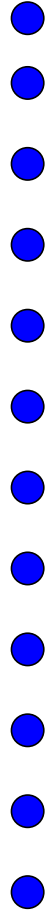
    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C;$

    Add  $u$  and  $v$  to  $C;$

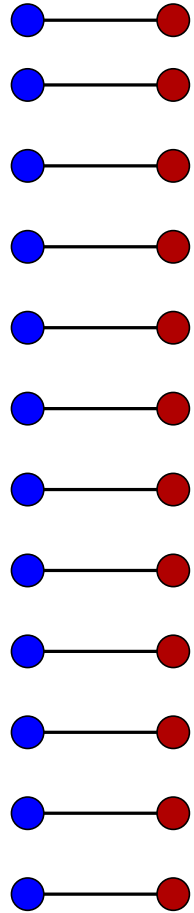
**done;**

**return**  $C;$

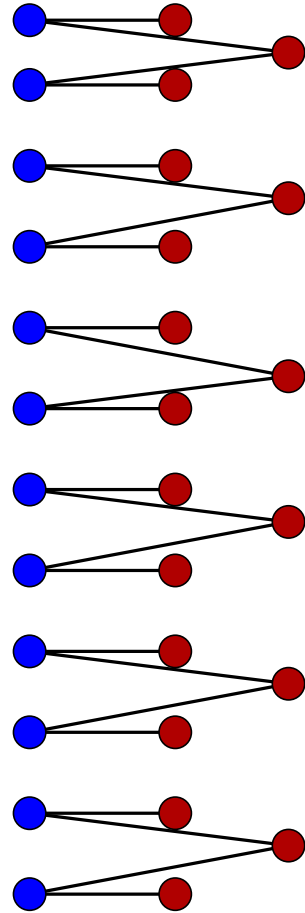
# Bad Instance for Largest-Degree-First



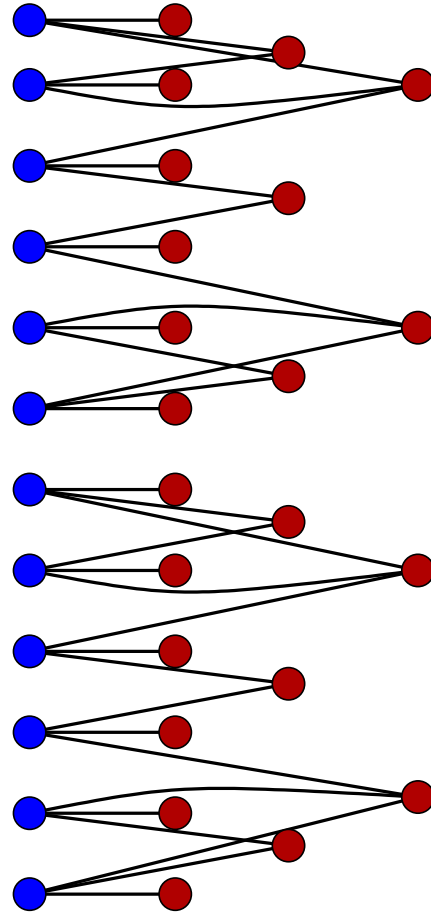
# Bad Instance for Largest-Degree-First



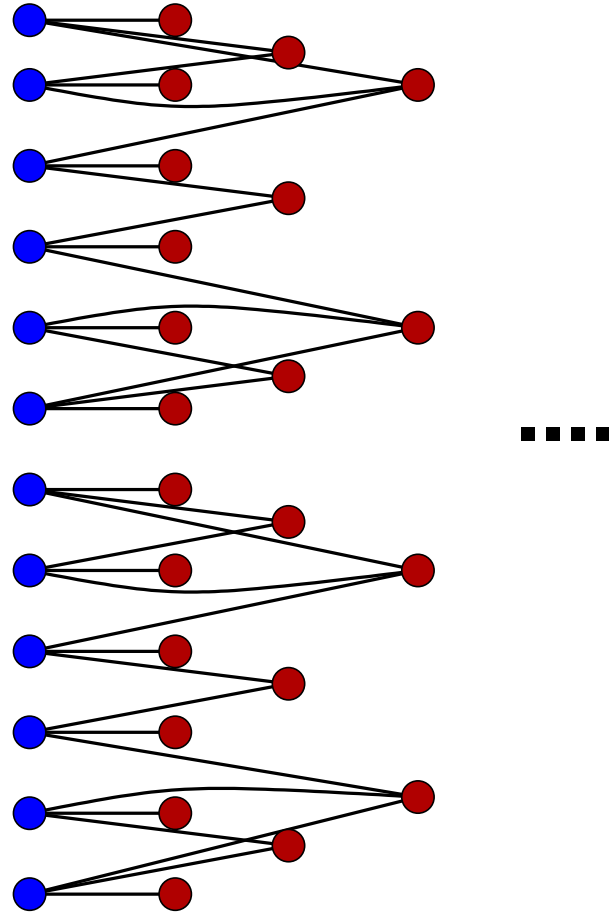
# Bad Instance for Largest-Degree-First



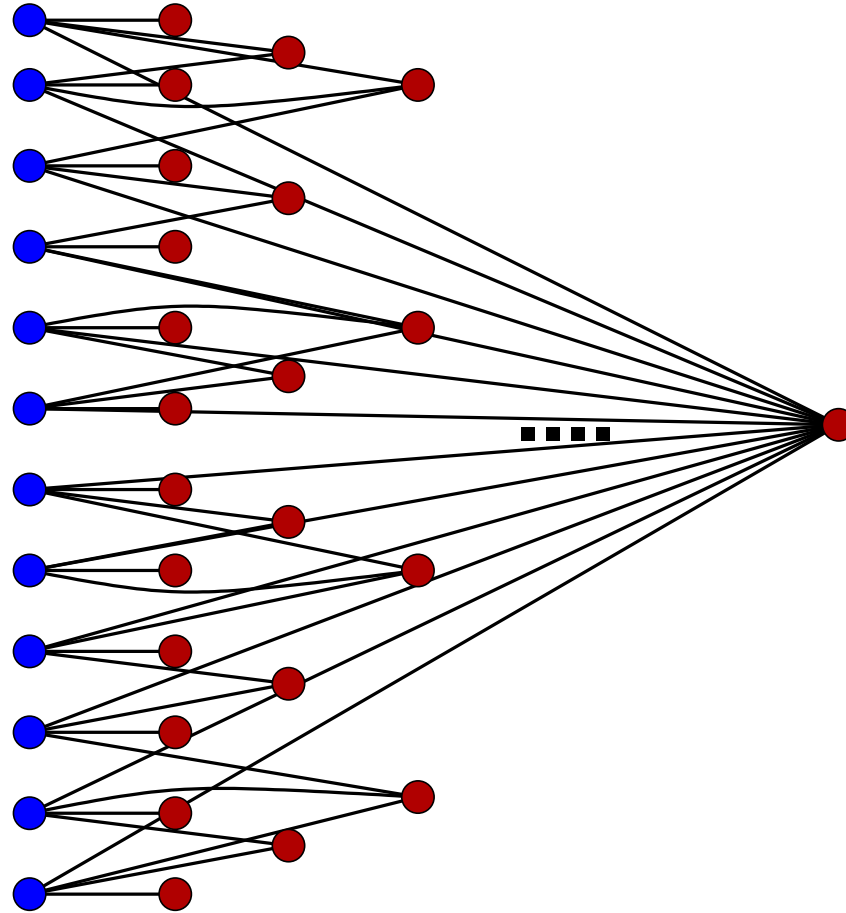
# Bad Instance for Largest-Degree-First



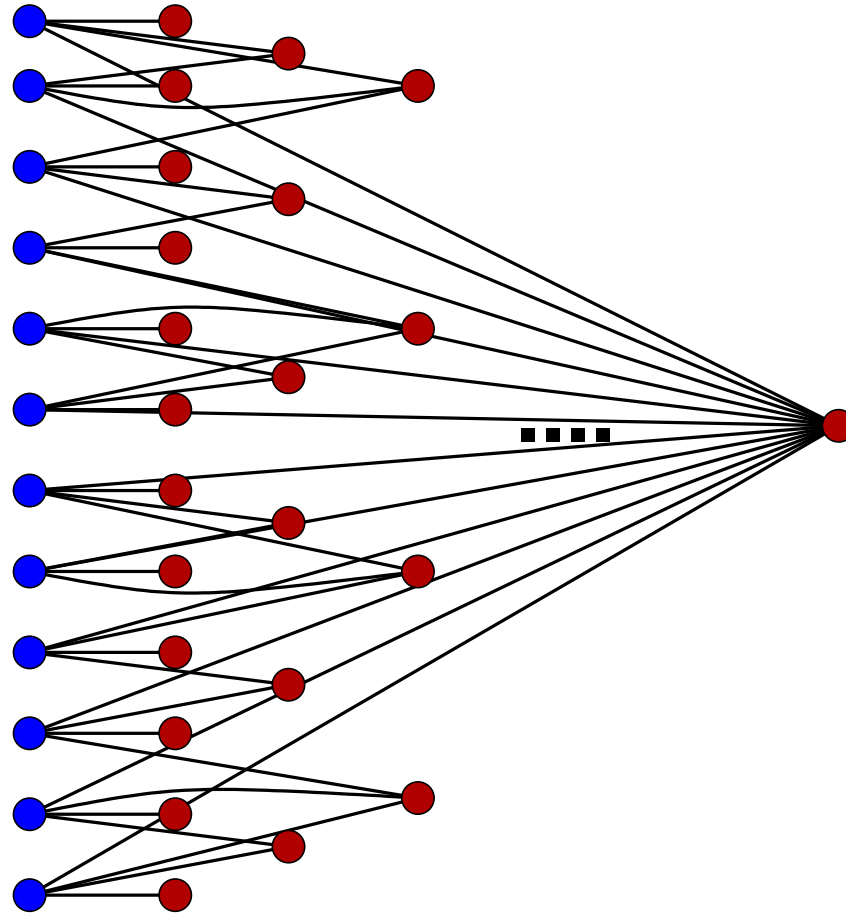
# Bad Instance for Largest-Degree-First



# Bad Instance for Largest-Degree-First



# Bad Instance for Largest-Degree-First



$\text{OPT} = N, \text{LDF} = \Theta(N \log N) \Rightarrow \text{approximation ratio } \Omega(\log |V|)$

# Anlaysia of the Matching-Based Alg.

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C$ ;

    Add  $u$  and  $v$  to  $C$ ;

**done;**

**return**  $C$ ;

- Let  $M$  be the set of edges  $e$  chosen in the algorithm.

# Anlysis of the Matching-Based Alg.

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C$ ;

    Add  $u$  and  $v$  to  $C$ ;

**done;**

**return**  $C$ ;

- Let  $M$  be the set of edges  $e$  chosen in the algorithm.
- $M$  is a matching.

# Anlysis of the Matching-Based Alg.

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C$ ;

    Add  $u$  and  $v$  to  $C$ ;

**done;**

**return**  $C$ ;

- Let  $M$  be the set of edges  $e$  chosen in the algorithm.
- $M$  is a matching.
- We have  $\text{OPT} \geq |M|$  and  $|C| = 2|M|$ , so  $|C| \leq 2\text{OPT}$ .

# Anlysis of the Matching-Based Alg.

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C$ ;

    Add  $u$  and  $v$  to  $C$ ;

**done;**

**return**  $C$ ;

- Let  $M$  be the set of edges  $e$  chosen in the algorithm.
- $M$  is a matching.
- We have  $\text{OPT} \geq |M|$  and  $|C| = 2|M|$ , so  $|C| \leq 2\text{OPT}$ .
- **The algorithm is a 2-approximation algorithm.**

# Anlaysia of the Matching-Based Alg.

**Input:**  $G = (V, E)$

$C := \emptyset;$

**while**  $C$  is not a vertex cover **do**

    Let  $e = \{u, v\}$  be an edge that is not yet covered by  $C$ ;

    Add  $u$  and  $v$  to  $C$ ;

**done;**

**return**  $C$ ;

- Let  $M$  be the set of edges  $e$  chosen in the algorithm.
- $M$  is a matching.
- We have  $\text{OPT} \geq |M|$  and  $|C| = 2|M|$ , so  $|C| \leq 2\text{OPT}$ .
- **The algorithm is a 2-approximation algorithm.**
- **No algorithm with ratio better than 2 is known!**

# Min Set Cover

- Instance:
  - Ground set  $U = \{1, 2, \dots, n\}$
  - Family  $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$  of subsets of  $U$ , such that their union equals  $U$ .
- Feasible solution: a subset  $\mathcal{S}'$  of  $\mathcal{S}$  such that the sets in  $\mathcal{S}'$  cover  $U$ .
- Objective value:  $|\mathcal{S}'|$
- Goal: Minimise  $|\mathcal{S}'|$

# Greedy Algorithm for Set Cover

**Input:**  $U$  and  $\mathcal{S}$

$\mathcal{S}' = \emptyset$ ;

**while**  $\mathcal{S}'$  is not yet a set cover **do**

    Let  $T$  be the set of not yet covered elements in  $U$ ;

    Let  $S_j$  be a set that covers a largest number of elements in  $T$ ;

    Add  $S_j$  to  $\mathcal{S}'$ ;

**done**

**return**  $\mathcal{S}'$ ;

# Analysing the Greedy Algorithm

- Let  $\text{OPT} = k$ .
- Define  $u_t =$  number of uncovered elements after  $t$  steps of the algorithm. Note that  $u_0 = n$ .
- The set  $S_j$  chosen by the algorithm in step  $t + 1$  covers at least  $u_t/k$  uncovered elements.
- Thus,  $u_{t+1} \leq u_t \cdot (1 - \frac{1}{k})$ .
- This means  $u_t \leq n \cdot (1 - \frac{1}{k})^t$  for all  $t \geq 0$ .
- If  $t > k \ln n$ , then

$$u_t \leq n \cdot (1 - \frac{1}{k})^t < n \cdot (1 - \frac{1}{k})^{k \ln n} \leq n \cdot e^{-\ln n} = 1$$

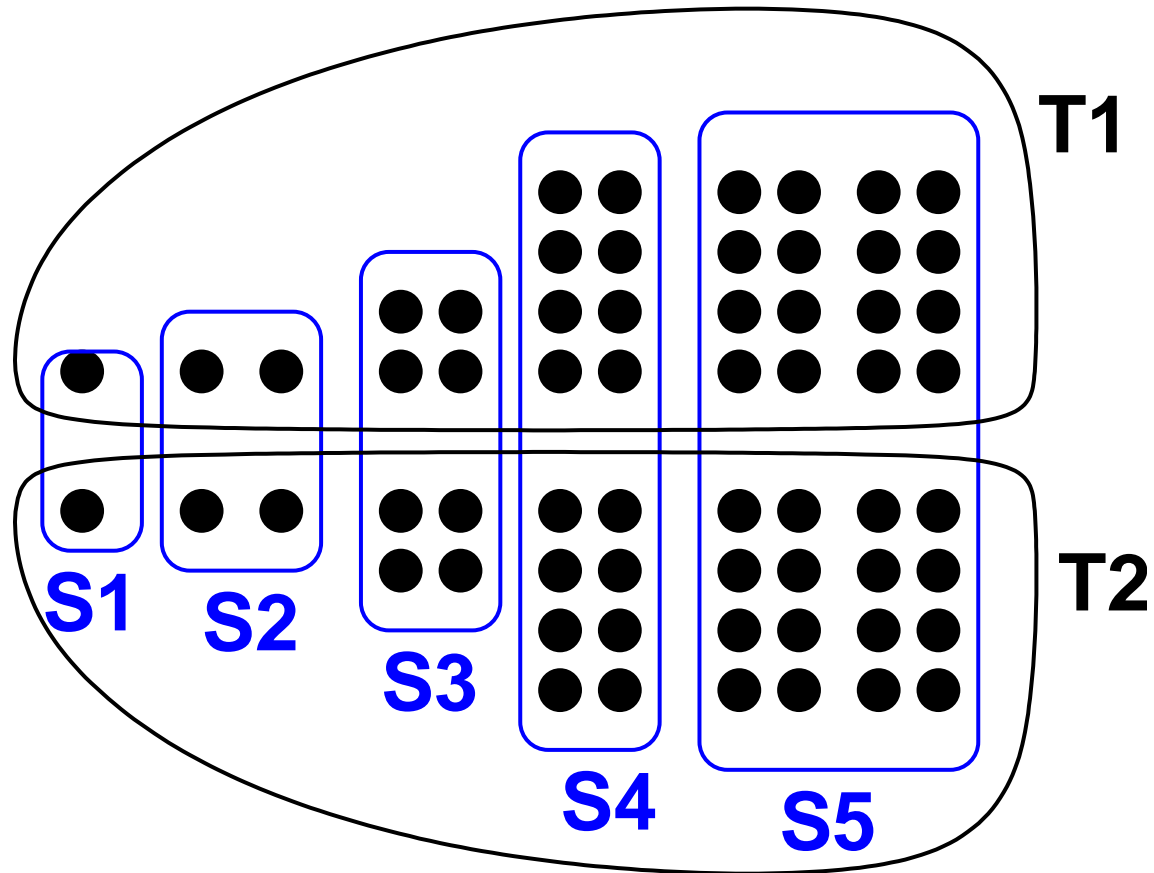
and thus the algorithm has already terminated.

# Approximation Ratio of Greedy

**Theorem.** Greedy is an  $O(\log n)$ -approximation algorithm for Min Set Cover, where  $n$  is the size of the ground set of the given instance.

**Remark.** Feige (1998) has shown that there is no  $(1 - \varepsilon) \ln n$ -approximation algorithm for Min Set Cover unless all problems in  $NP$  have quasi-polynomial time algorithms.

# Bad Instance for Greedy



# Machine Scheduling

- Instance:
  - Number  $m$  of machines
  - $n$  jobs of sizes  $p_1, p_2, \dots, p_n$
- Feasible solution: An assignment of jobs to machines.
- Objective value: Makespan, i.e., the maximum sum of the sizes of the jobs assigned to the same machine.
- Goal: Minimise the makespan.

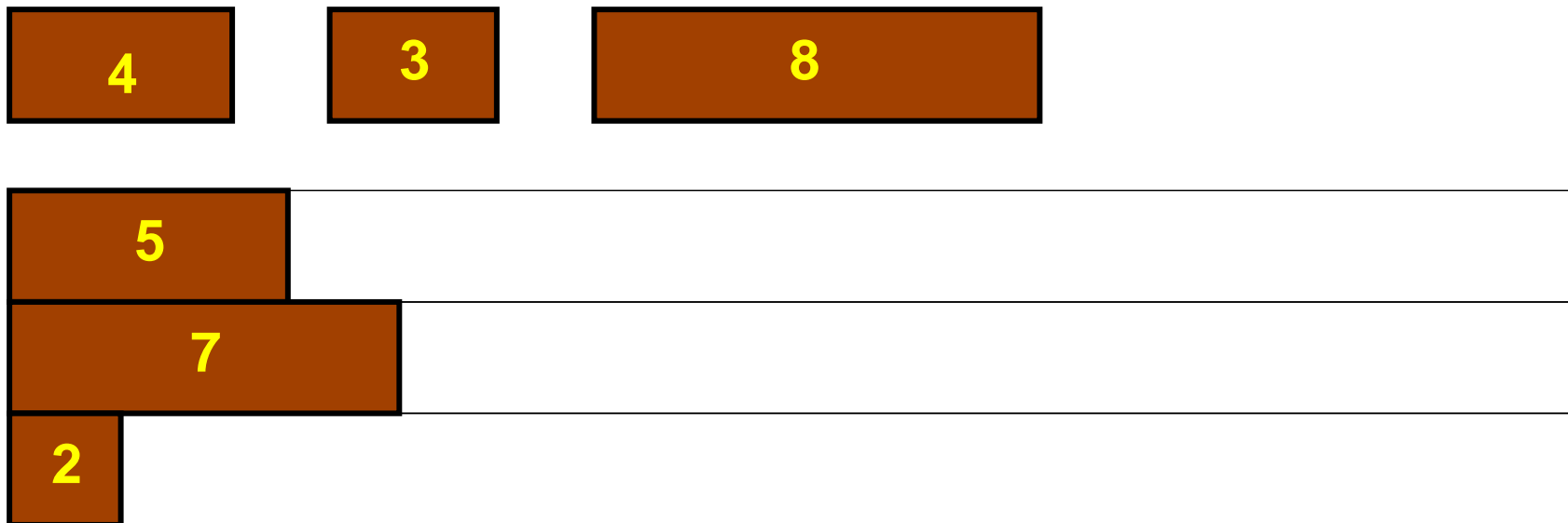
# Example

- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:



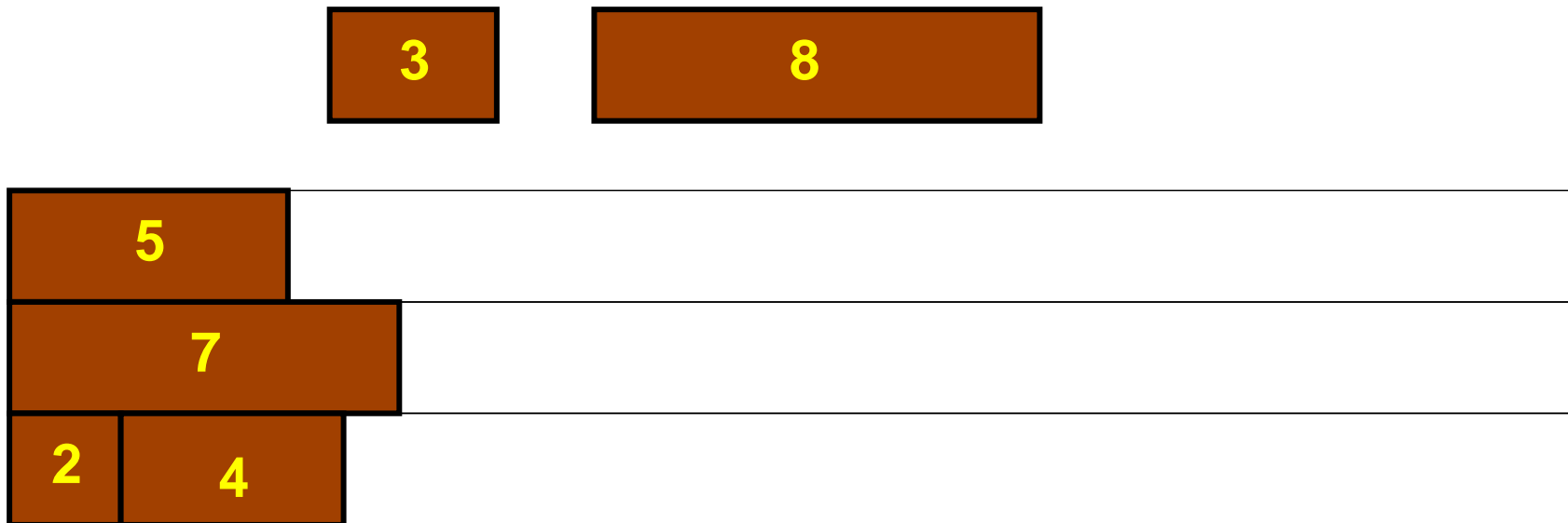

# Example

- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:



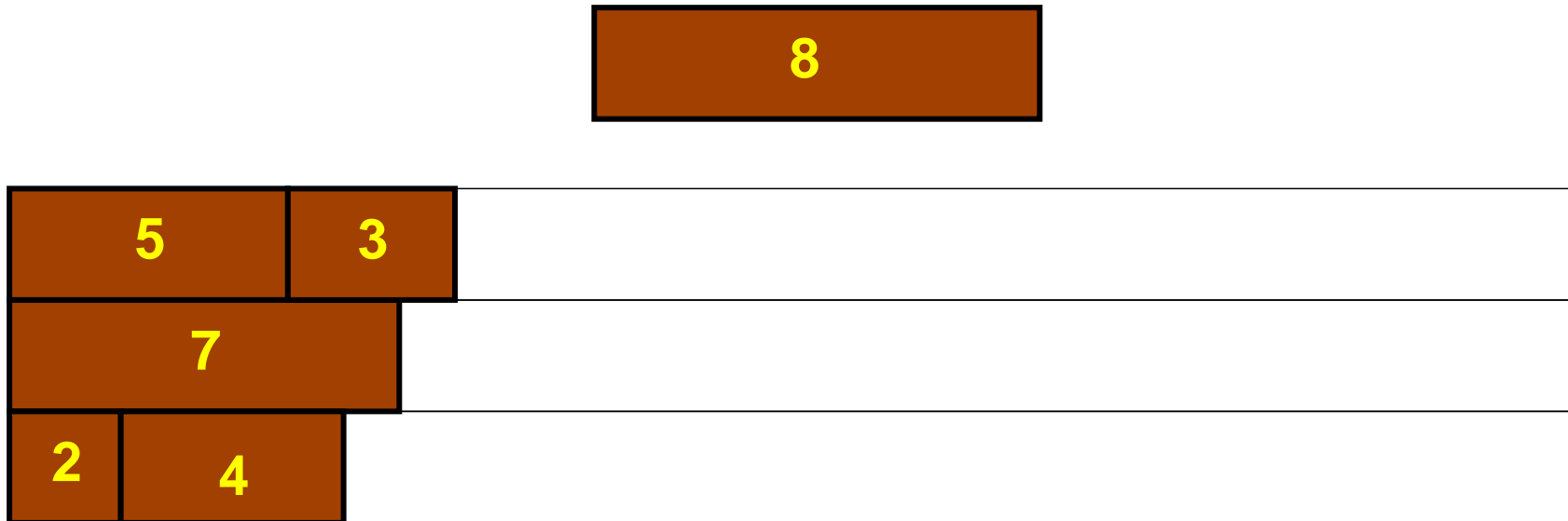
# Example

- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:



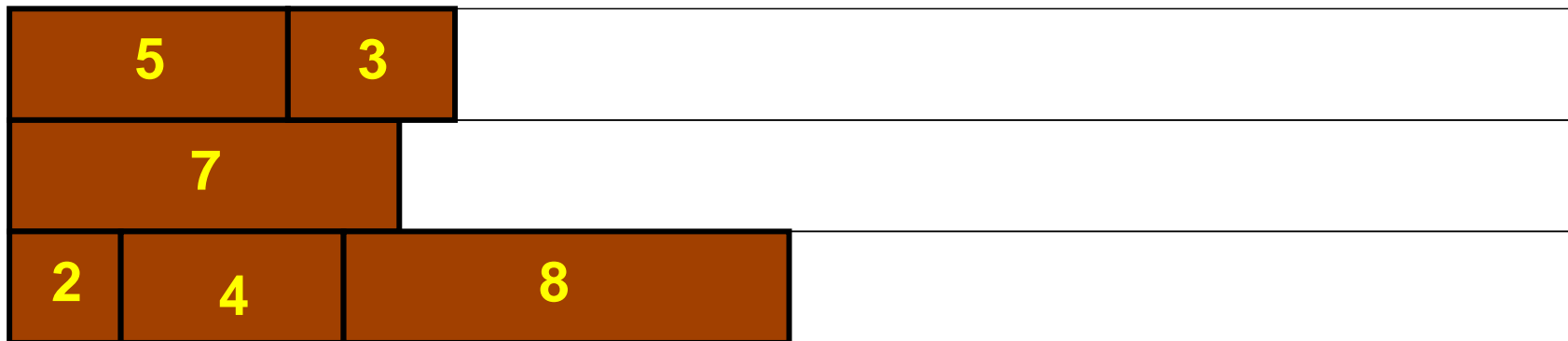
# Example

- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:



# Example

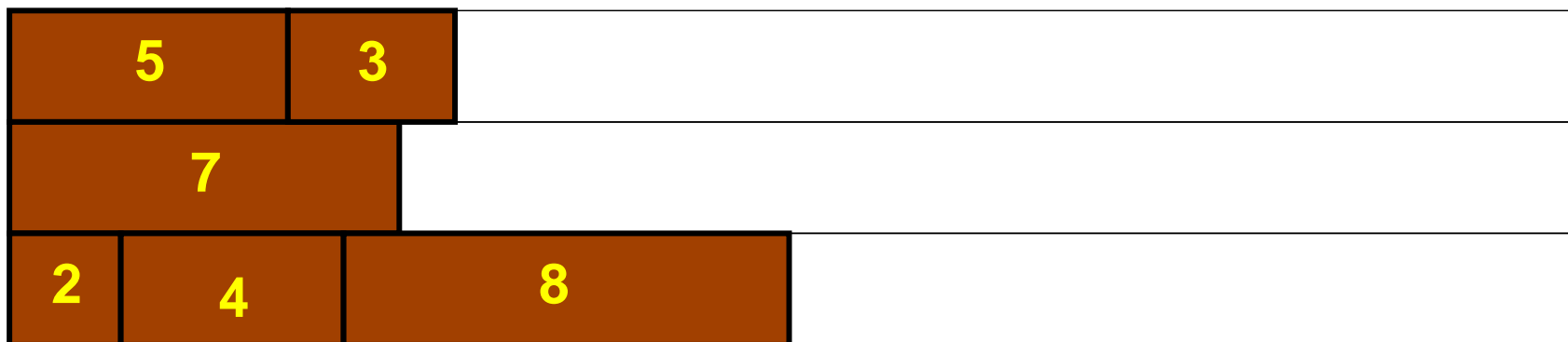
- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:



# Example

- $m = 3$  machines
- Job sizes 5, 7, 2, 4, 3, 8
- Schedule:

The makespan of this schedule is 14.



# Algorithm: List Scheduling

- Process the jobs in arbitrary order.
- Assign each job to the machine that finishes its previously assigned jobs at the earliest time.

## Analysis

- Let  $\ell$  be the last time when all machines are busy.

# Algorithm: List Scheduling

- Process the jobs in arbitrary order.
- Assign each job to the machine that finishes its previously assigned jobs at the earliest time.

## Analysis

- Let  $\ell$  be the last time when all machines are busy.
- Let  $p_{\max}$  be the largest job size.

# Algorithm: List Scheduling

- Process the jobs in arbitrary order.
- Assign each job to the machine that finishes its previously assigned jobs at the earliest time.

## Analysis

- Let  $\ell$  be the last time when all machines are busy.
- Let  $p_{\max}$  be the largest job size.
- $LS \leq \ell + p_{\max}$ ,  $OPT \geq \ell$ ,  $OPT \geq p_{\max}$

# Algorithm: List Scheduling

- Process the jobs in arbitrary order.
- Assign each job to the machine that finishes its previously assigned jobs at the earliest time.

## Analysis

- Let  $\ell$  be the last time when all machines are busy.
- Let  $p_{\max}$  be the largest job size.
- $LS \leq \ell + p_{\max}$ ,  $OPT \geq \ell$ ,  $OPT \geq p_{\max}$
- List Scheduling is a 2-approximation algorithm.

# Traveling Salesperson Problem (TSP)

- Instance:
  - $n$  cities
  - Distances  $c_{ij}$  for traveling between city  $i$  and city  $j$
- Feasible solution: a tour that visits each city exactly once (a permutation of the  $n$  cities).
- Objective value: Length of the tour, i.e., the sum of the distances between consecutive cities on the tour.
- Goal: Minimise the tour length.

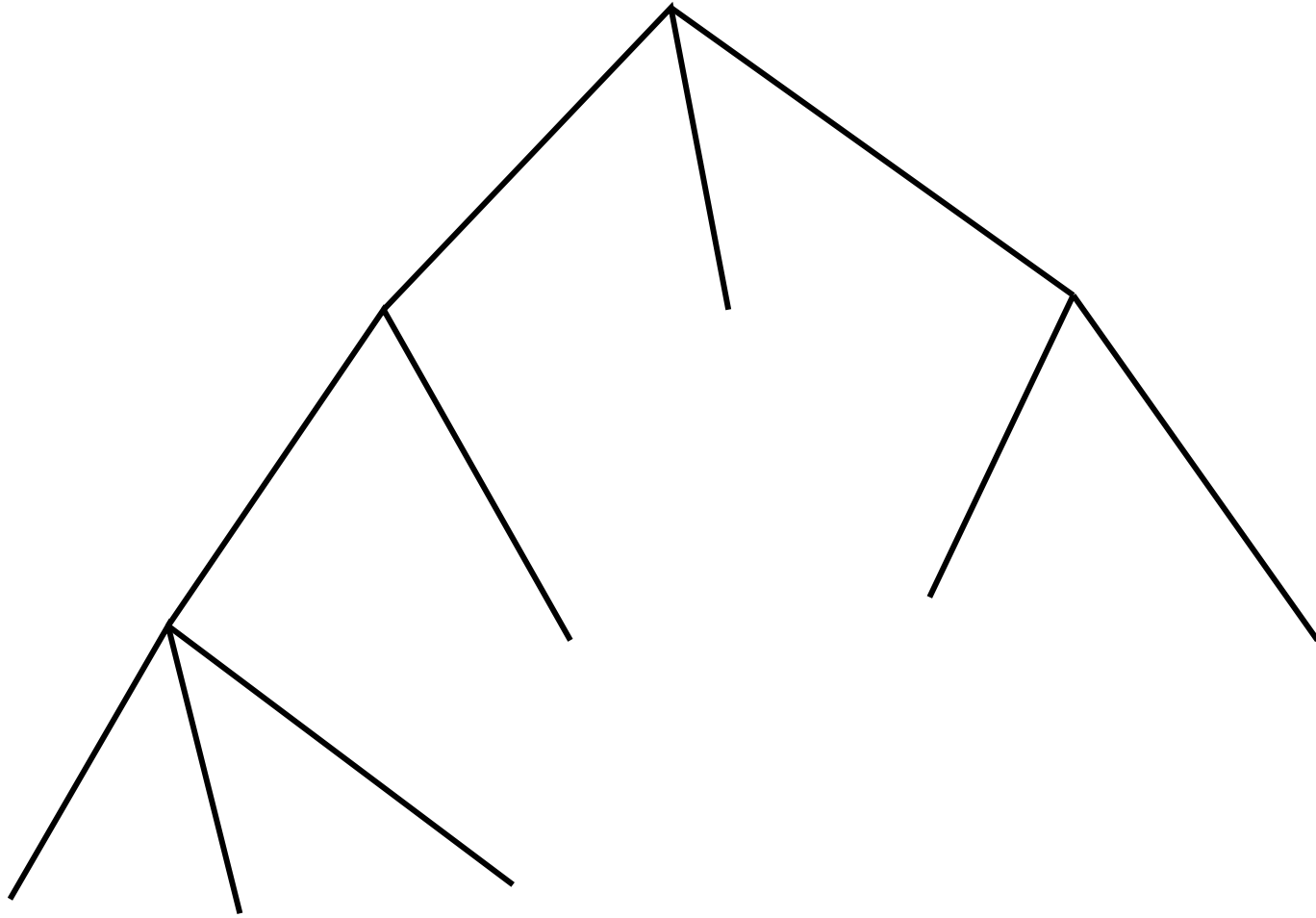
# $\Delta$ -TSP

- Assume that the triangle inequality holds for the given distances:

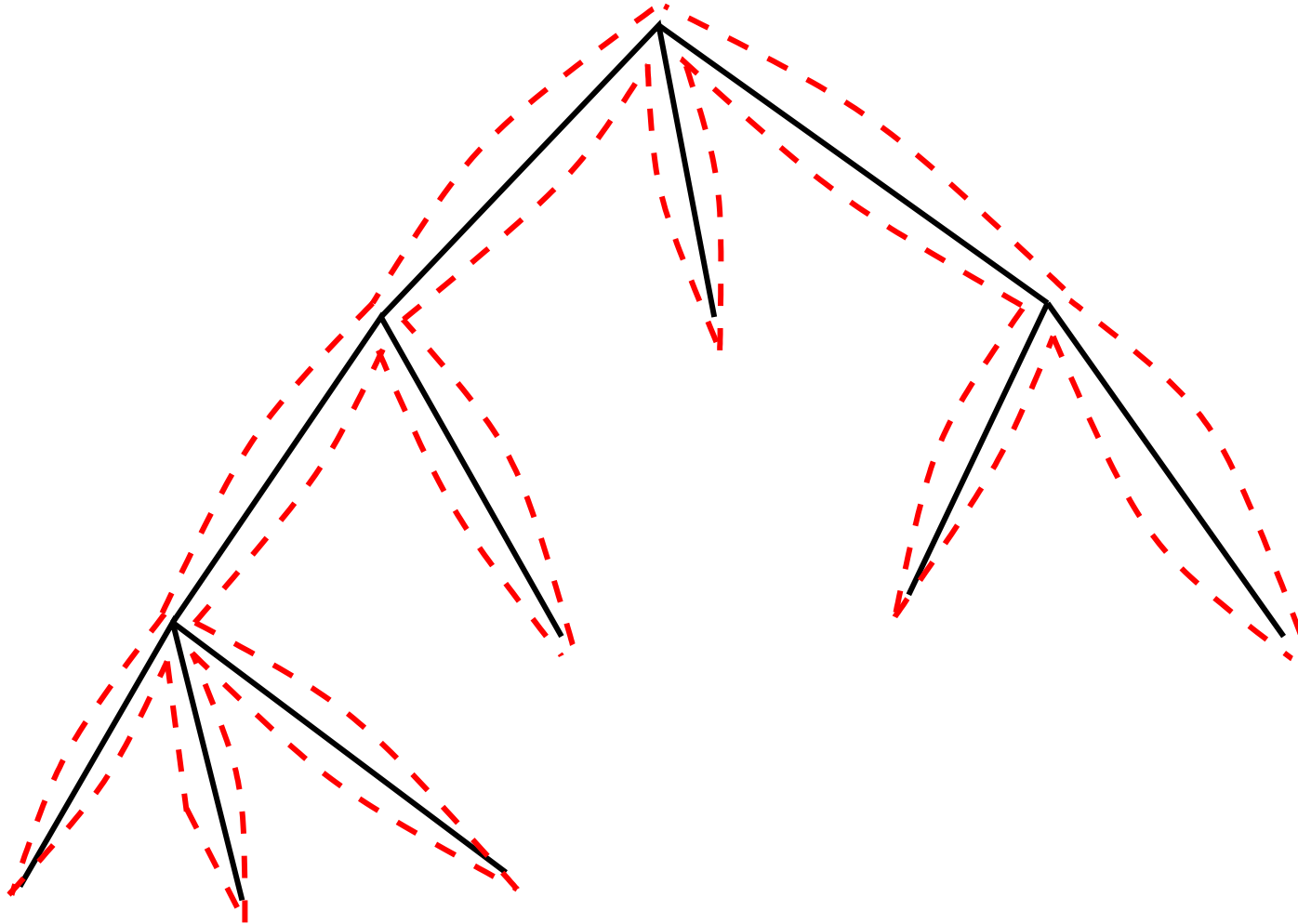
$$c_{ij} \leq c_{ik} + c_{kj}$$

- Consider the following algorithm:
  - Construct a complete graph  $G$  on  $n$  vertices with edge weights  $c_{ij}$ .
  - Compute a minimum spanning tree  $T$  in  $G$ .
  - Construct a tour by walking around the minimum spanning tree.
  - Whenever the tour visits a vertex for the second time, shortcut to the following vertex, until the tour visits every city exactly once.

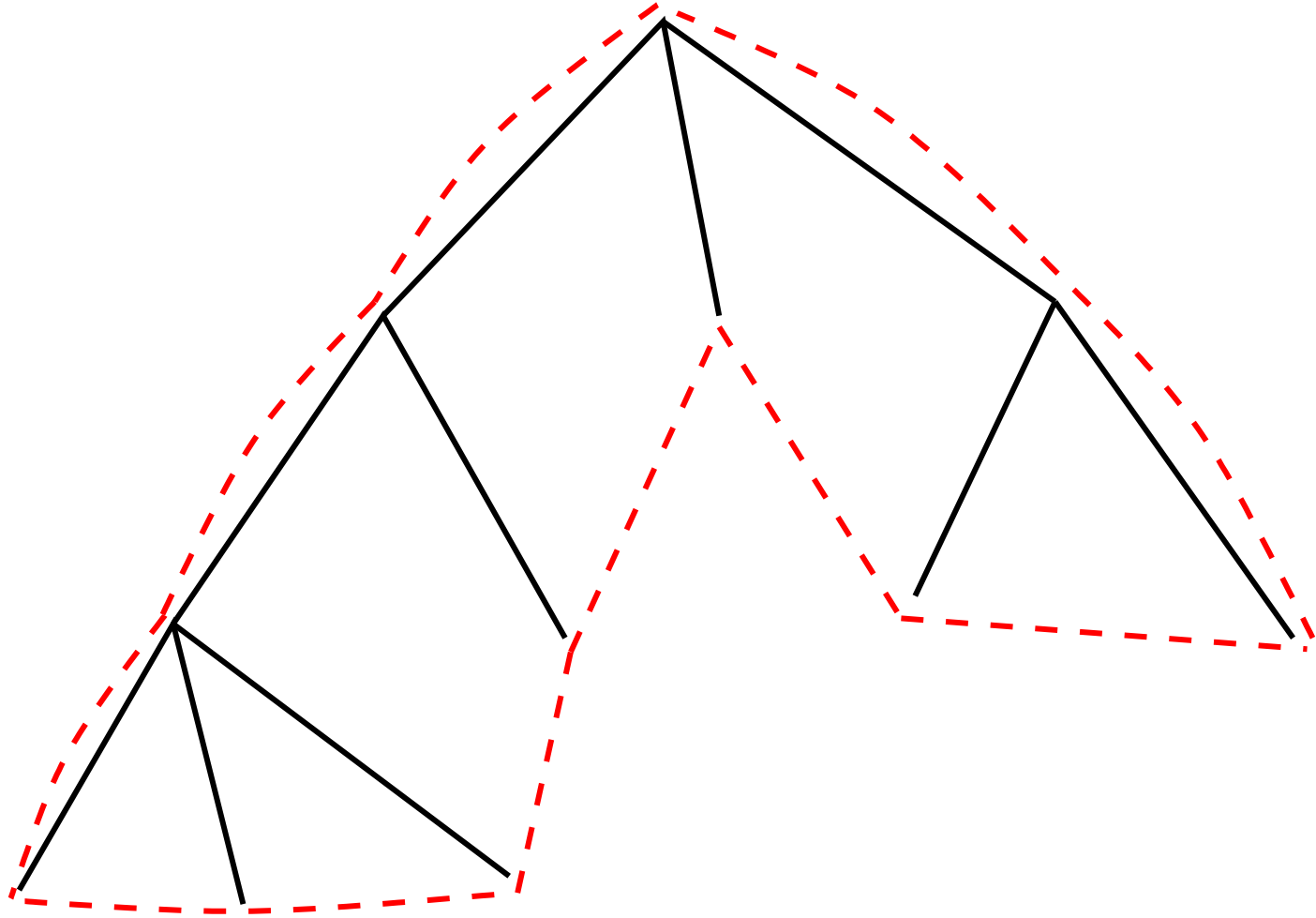
# Illustration



# Illustration



# Illustration



# Analysis

- Let  $MST$  denote the cost of the minimum spanning tree in  $G$ .
- The optimal TSP tour has length  $OPT \geq MST$ , because the optimal tour minus one edge is a spanning tree of cost at most  $OPT$ .
- The tour computed by the algorithm has cost at most  $2 \cdot MST$ , because the shortcuts do not increase the tour length (because of the triangle inequality).
- Thus, the algorithm is a 2-approximation algorithm for  $\Delta$ -TSP.
- Remark: Can improve to 1.5-approximation algorithm (Christofides, 1976; no better algorithm known for general  $\Delta$ -TSP)