

---

# *Algorithm Design Strategies V*

---

Joaquim Madeira

Version 0.3 – October 2022

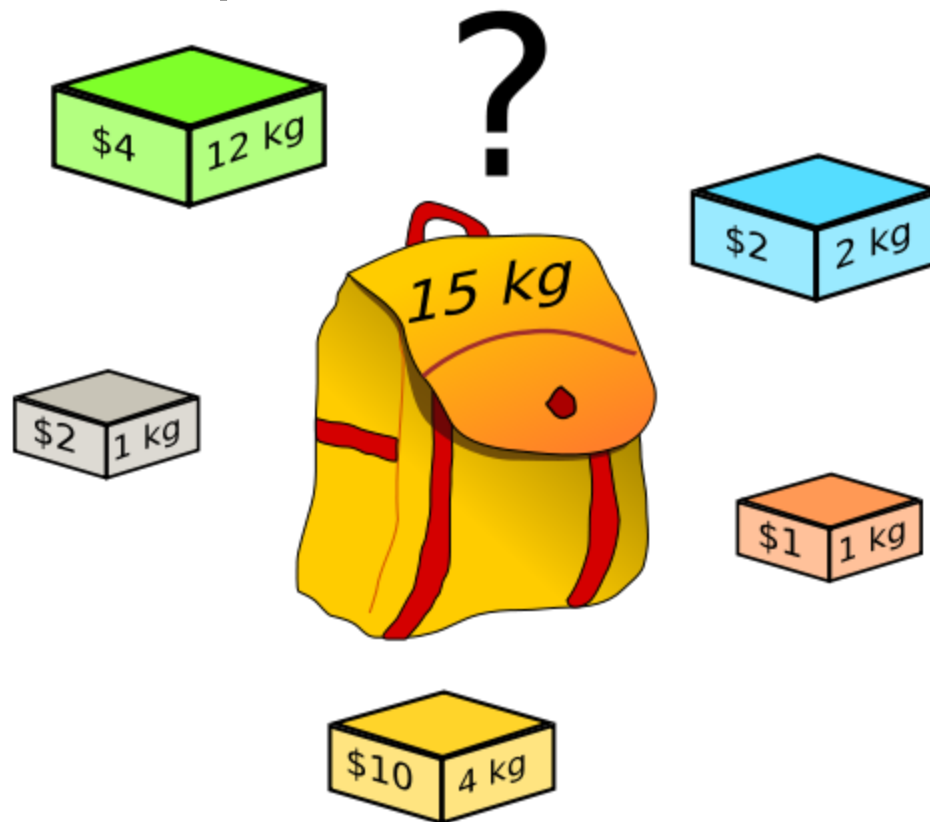
# Overview

- The 0-1 Knapsack Problem Revisited
- The Fractional Knapsack Problem
- Greedy Algorithms
- Example – Coin Changing
- Example – Activity Selection
- Some Problems on Graphs
- The Traveling Salesperson Problem

# THE 0-1 KNAPSACK PROBLEM

# The 0-1 Knapsack Problem

- Find the **most valuable subset** of items, that fit into the knapsack



[Wikipedia]

# The 0-1 Knapsack Problem

- Given  $n$  items
  - Known **weight**  $w_1, w_2, \dots, w_n$
  - Known **value**  $v_1, v_2, \dots, v_n$
- A knapsack of **capacity**  $W$
- Which one is the **most valuable subset** of items that fit into the knapsack ?
  - **More than one** solution ?

# The 0-1 Knapsack Problem

- How to formulate ?

$$\text{max } \sum x_i v_i$$

$$\text{subject to } \sum x_i w_i \leq W$$

$$\text{with } x_i \text{ in } \{0, 1\}$$

# The 0-1 Knapsack Problem

- We have seen how to solve it using
  - Exhaustive Search
  - Dynamic Programming
- BUT, it takes **too much time** for “large” **instances !!**
- Can we use a different, “**greedy**” strategy ?

# The 0-1 Knapsack Problem

- Knapsack of capacity  $W = 10$
- 4 items
  - Item 1 :  $w = 7$  ;  $v = \$42$  :  $v / w = 6$  : 2nd
  - Item 2 :  $w = 3$  ;  $v = \$12$  :  $v / w = 4$  : 4th
  - Item 3 :  $w = 4$  ;  $v = \$40$  :  $v / w = 10$  : 1st
  - Item 4 :  $w = 5$  ;  $v = \$25$  :  $v / w = 5$  : 3rd
- Solution ?
  - Item 3 + Item 4 : **\$65**
  - Optimal solution



# The 0-1 Knapsack Problem

- Greedy heuristic
  - Select items in decreasing order of their  $v / w$  ratios
- Compute the value-to-weight ratios :  $r_i = v_i / w_i$
- Sort the items in non-increasing order of their  $r_i$
- Repeat until no item is left in the sorted list
  - If the current item fits, place it in the knapsack
  - Otherwise, discard it


# The 0-1 Knapsack Problem

- It is a very **simple heuristic**...
  - There are others...
- **Can it be always optimal ?**
- What would a positive answer imply ?

# The 0-1 Knapsack Problem

- Knapsack of capacity  $W = 50$
- 3 items
  - Item 1 :  $w = 10$  ;  $v = \$60$  :  $v / w = 6$
  - Item 2 :  $w = 20$  ;  $v = \$100$  :  $v / w = 5$
  - Item 3 :  $w = 30$  ;  $v = \$120$  :  $v / w = 4$
- Result of the greedy strategy
  - Item 1 + Item 2 : \$160 ←
- Optimal solution
  - Item 2 + Item 3 : \$220 !! ←

# The 0-1 Knapsack Problem

- Knapsack of capacity  $W > 2$
- 2 items
  - Item 1 :  $w = 1$  ;  $v = \$2$  :  $v / w = 2$  : 1st !!
  - Item 2 :  $w = W$  ;  $v = \$W$  :  $v / w = 1$  : 2nd
- Solution ?
  - Item 1 !!!
  - Optimal solution : Item 2 
  - $R_A = \infty$  !!

# TASKS

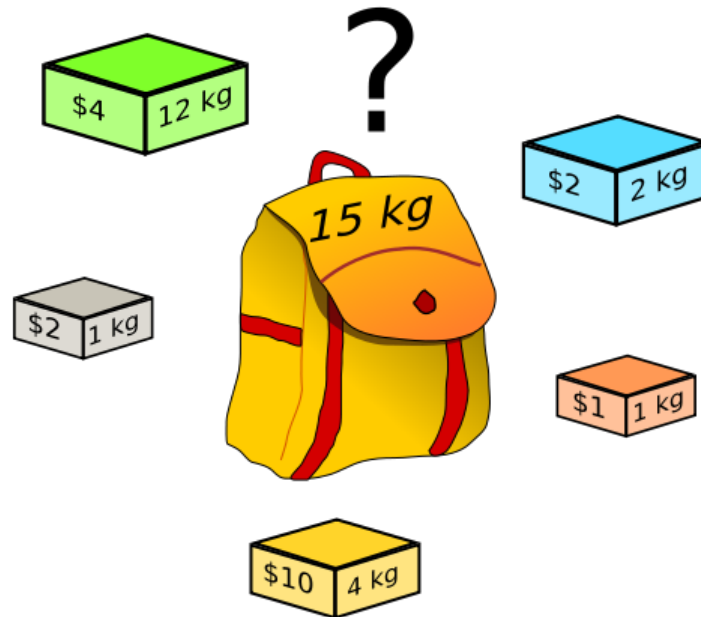
# Tasks

- Implement a **greedy function** for computing a (approx.) solution to an instance of the 0-1 Knapsack
- Run it for a set of **test instances**
- Check the **accuracy** of the obtained solutions
  - By using the **brute-force function** of last week

# THE CONTINUOUS KNAPSACK PROBLEM

# The Continuous Knapsack Problem

- Find the most valuable subset of items, that fit into the knapsack



[Wikipedia]

- BUT, now we can **take any fraction** of any given item !!



# The Continuous Knapsack Problem

- How to formulate ?

$$\max \sum x_i v_i$$

subject to  $\sum x_i w_i \leq W$

with  $0 \leq x_i \leq 1$

# The Continuous Knapsack Problem

- Compute the **value / weight ratio** for each item
- **Order items** according to  $v / w$  ratio (non-increasing)
- Scan the ordered items, while the knapsack is **not full**
  - Check if **whole item  $i$  fits** into the knapsack
  - Yes : **add it** to the solution !
  - Else, determine the **fraction** of item  $i$  that fits into the knapsack and add it to the solution

# The Continuous Knapsack Problem

- **Optimal strategy** for this problem !!
- Same example
  - Knapsack of capacity  $W = 50$
  - 3 items
    - Item 1 :  $w = 10$  ;  $v = \$60$  :  $v / w = 6$
    - Item 2 :  $w = 20$  ;  $v = \$100$  :  $v / w = 5$
    - Item 3 :  $w = 30$  ;  $v = \$120$  :  $v / w = 4$
- Result?
  - Item 1 + Item 2 +  $2 / 3$  (Item 3) ; **\$240 !!**

# TASKS

# Tasks

- Implement a **greedy function** for computing the solution to an instance of the Fractional Knapsack Problem
- Run it for a set of **test instances**

---

# GREEDY ALGORITHMS

# Greedy Algorithms

- For **Optimization Problems**
- How to construct a solution ?
- Sequence of **choices**
- Expand a **partially** constructed solution
  - Grab the “**best-looking**” alternative !!
  - Hope that it will lead to **the / a** globally optimal solution
- Reach a **complete solution**

# Greedy Algorithms

- The choice made at each step is
  - **Feasible** : satisfies constraints
  - **Locally optimal** : best choice at each step
  - **Irrevocable**
- Does it always work ?



# THE COIN-CHANGING PROBLEM

# The Coin-Changing Problem

- Make change for an **amount A**
- Available coin denominations
  - $\text{Denom}[1] > \text{Denom}[2] > \dots > \text{Denom}[n] = 1$
- Use the **fewest** number of coins !!
- Assumption
  - Enough coins of each denomination !!

# The Coin-Changing Problem

- How to formulate ?

$$\min \sum x_i$$

subject to  $\sum x_i d[i] = A$

with  $x_i = 0, 1, 2, \dots$

- Compare with the 0-1 Knapsack formulation

# The Coin-Changing Problem

$i \leftarrow 1$

while (  $A > 0$  ) do

$c \leftarrow A \text{ div } \text{denom}[i]$

output (  $c$  coins of  $\text{denom}[i]$  value )

$A \leftarrow A - c \times \text{denom}[i]$

$i \leftarrow i + 1$

- How to improve ?
  - No output when  $c = 0$  !!

# The Coin-Changing Problem

- Does the algorithm terminate ?

- Complexity ?

- Best Case ?

- ☐ Number of iterations ?
- ☐ When ?
- ☐ How many coins ?

- Worst Case ?

- ☐ Number of iterations ?
- ☐ When ?
- ☐ How many coins ?

# The Coin-Changing Problem

- Is this greedy approach **always optimal** ?
- It depends on the **set of denominations** !
- Example
  - $\text{Denom}[1] = 7$  ;  $\text{Denom}[2] = 5$  ;  $\text{Denom}[3] = 1$
  - $A = 10$
  - How many coins ?
  - Devise **other examples** !!

# The Coin-Changing Problem

- An alternative **Dynamic Programming** algorithm exists !
- It is always **optimal** !
- Check it and try to understand how it works

---

# TASKS



# Tasks

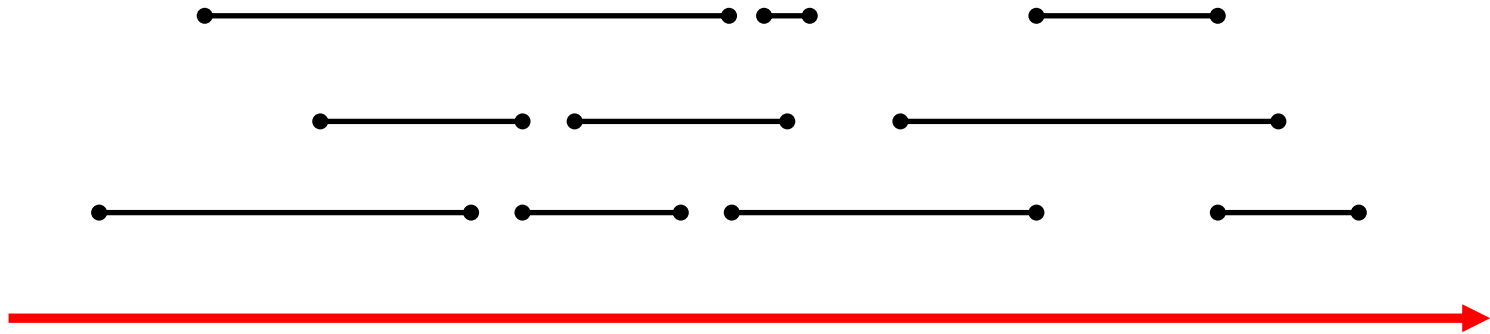
- Implement a **greedy function** for computing the solution to an instance of the Coin-Changing Problem
- Run it for a set of **test instances**

# THE ACTIVITY SELECTION PROBLEM

# The Activity Selection Problem

- Set of **jobs** to be scheduled on **one** resource
  - Classes on a classroom
  - Jobs for a supercomputer
  - ...
- **Start** time and **finish** time for each job known
  - $[s(i), f(i)]$  ←
- **Goal:** Schedule as many as possible !
  - **Max** problem !

# The Activity Selection Problem



- Which jobs / requests should be chosen ?
  - ❑ No jobs with overlapping intervals !
  - ❑ Solve the example !
- Greedy strategy:
  - ❑ Possible heuristics ?
  - ❑ Ensure optimal solutions ?

# Greedy Algorithm

$R \leftarrow$  set of all requests

$A \leftarrow \{ \}$  // Selected activities

while (  $R$  is not empty ) do

    choose  $r$  from  $R$  // How ?

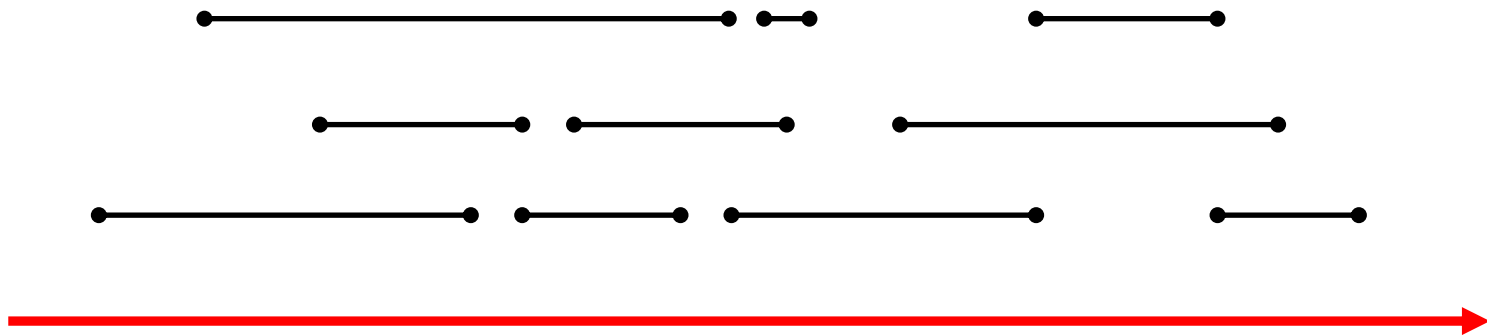
$A \leftarrow A \cup \{ r \}$

$R \leftarrow R - \{\text{pending requests overlapping } r\}$  ←

return  $A$

# Earliest Start Time

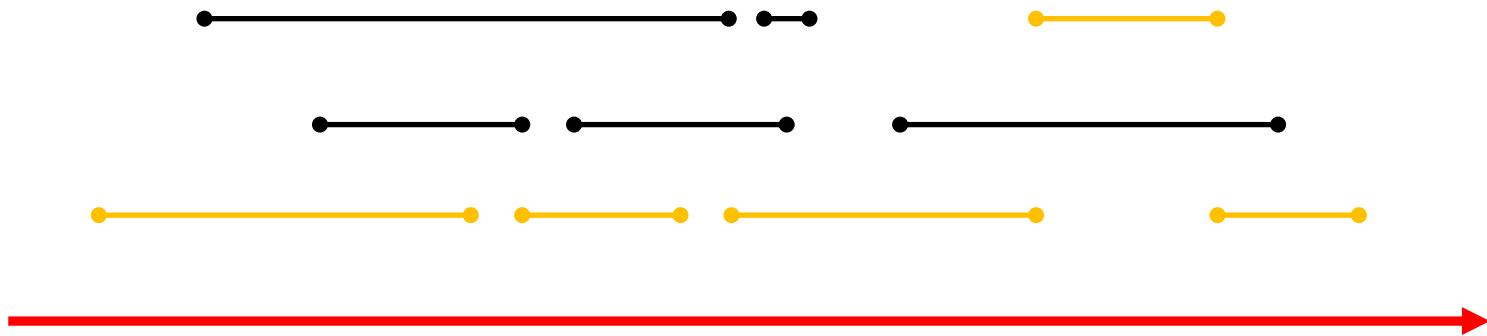
- Process requests according to **start time**
  - Begin with those that **start earliest**



- **Always** optimal ?

# Earliest Start Time

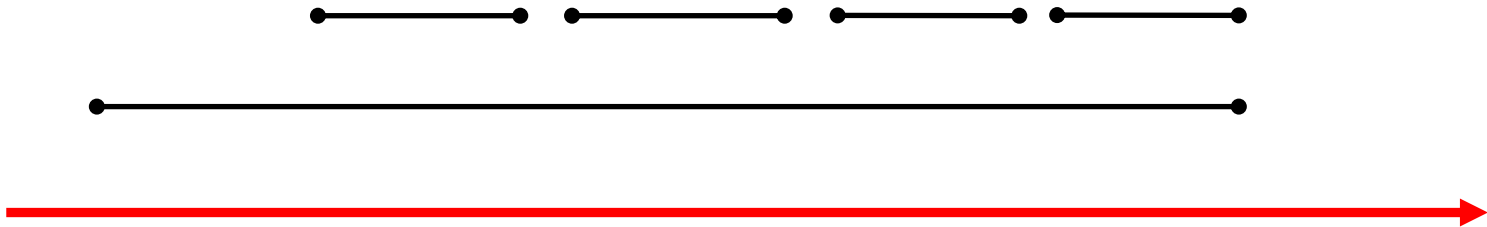
- Process requests according to **start time**
  - Begin with those that **start earliest**



- **Always** optimal ?

# Earliest Start Time

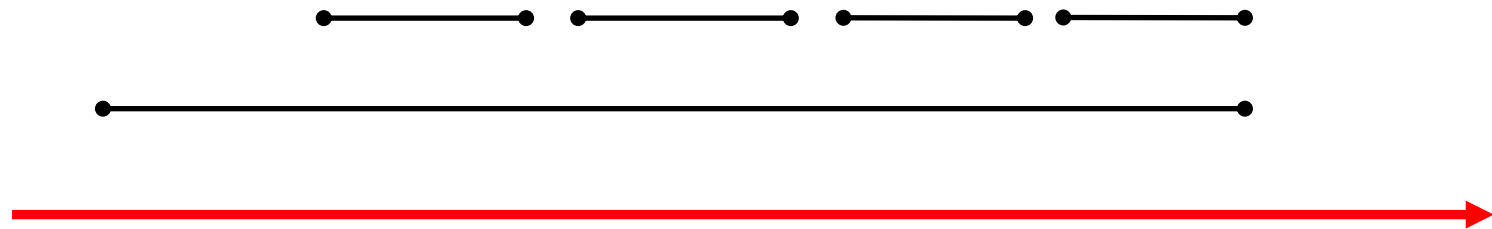
## ■ Counter example





# Smallest Processing Time

- Process requests according to **duration**
  - Begin with those requiring the **shortest processing**



- **Always** optimal ?

# Smallest Processing Time

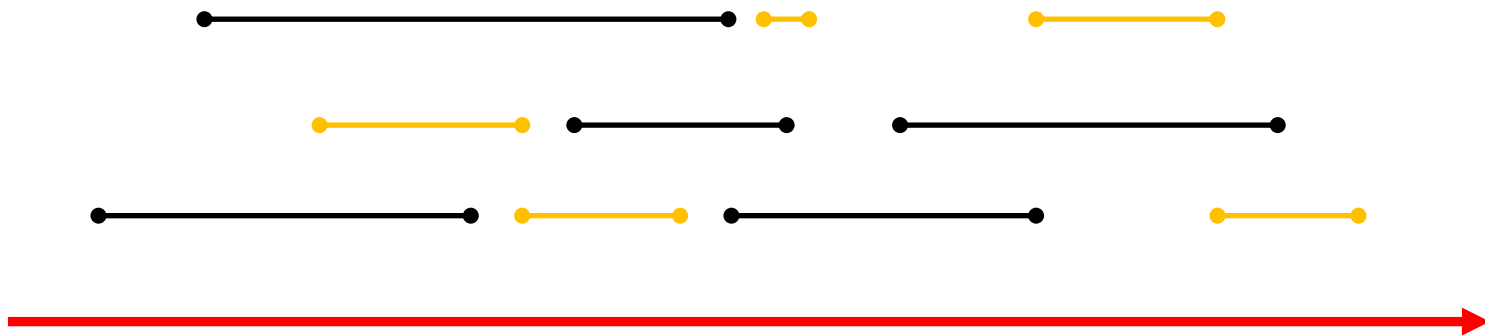
- Process requests according to **duration**
  - Begin with those requiring the **shortest processing**



- **Always** optimal ?

# Smallest Processing Time

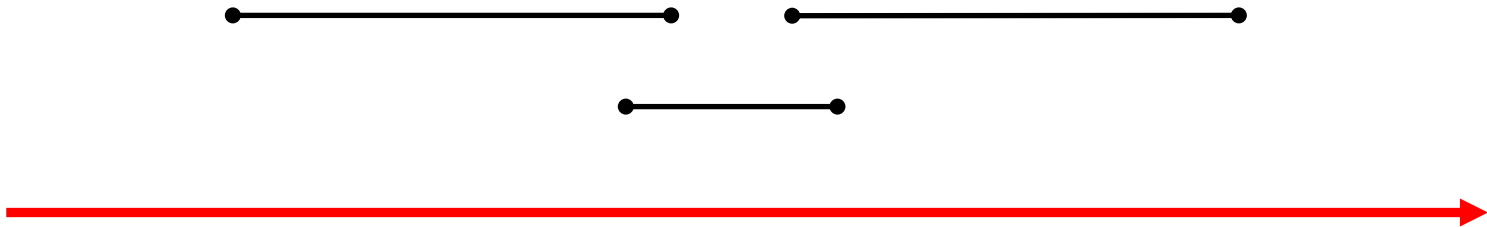
## ■ The first example



## ■ Always optimal ?

# Smallest Processing Time

## ■ Counter example



# Fewest Conflicts

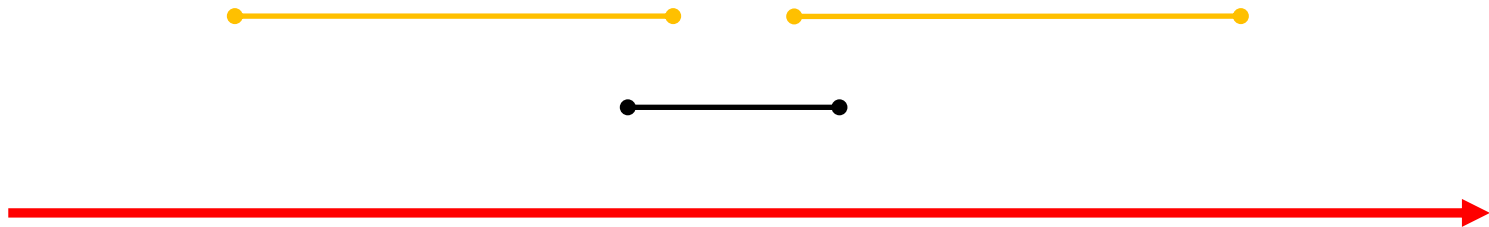
- Determine and **update** number of **conflicts**
  - Begin with those having the **fewest** conflicts



- **Always** optimal ?

# Fewest Conflicts

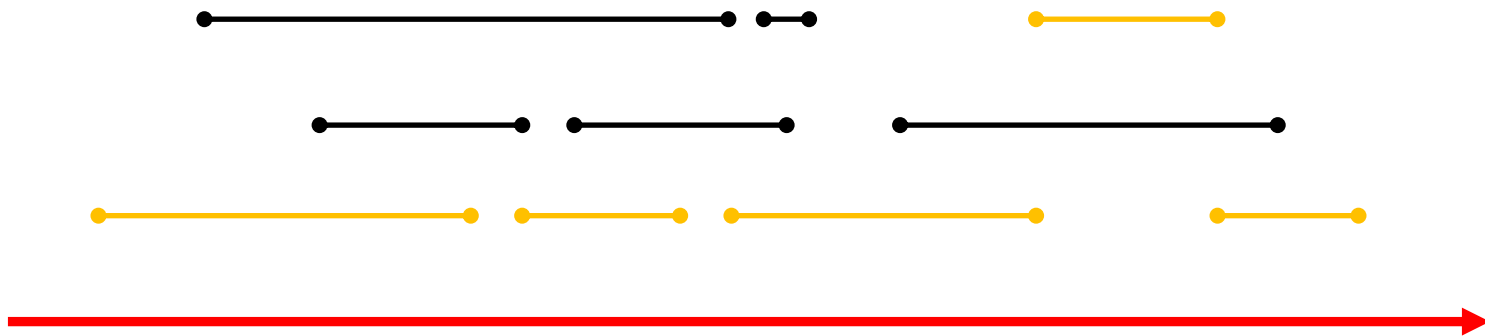
- Determine and **update** number of **conflicts**
  - Begin with those having the **fewest** conflicts



- **Always** optimal ?

# Fewest Conflicts

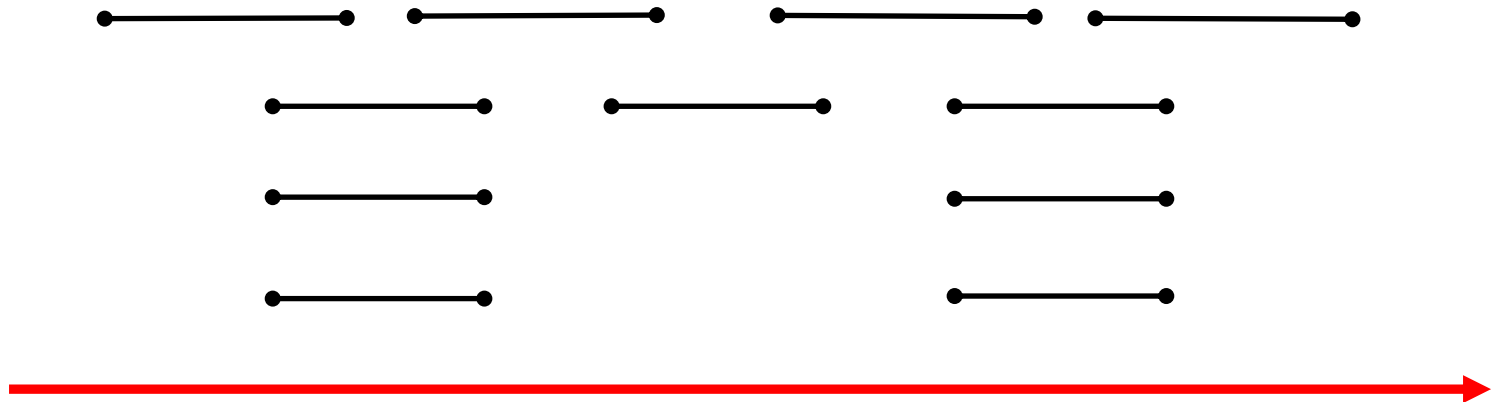
## ■ The first example



## ■ Always optimal ?

# Fewest Conflicts

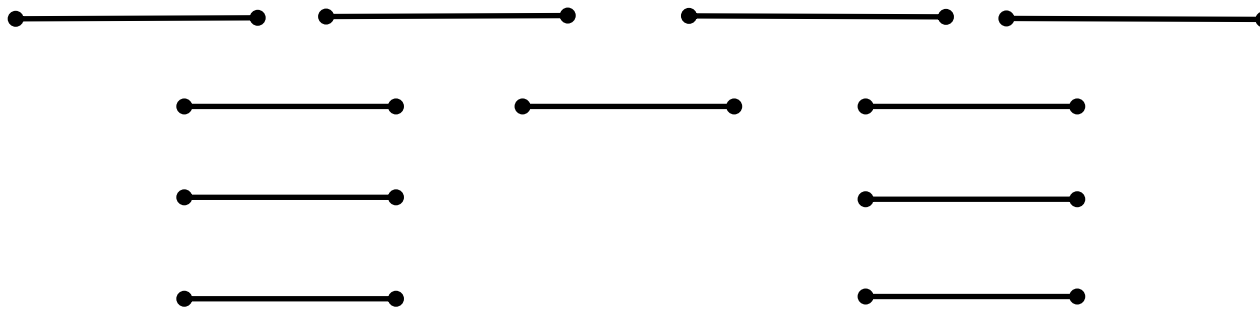
## ■ Counter example





# Earliest Finish Time

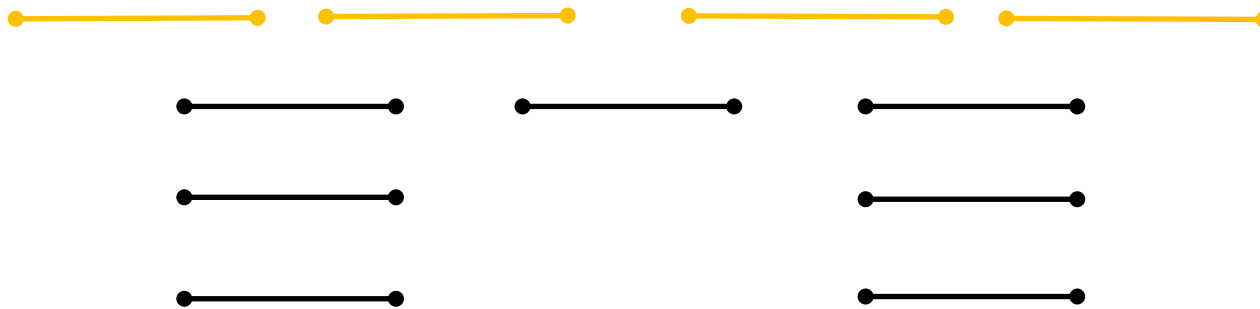
- Process request according to **finish time**
  - Begin with those **finishing earliest**



- **Always** optimal ?
- Try the previous examples !

# Earliest Finish Time

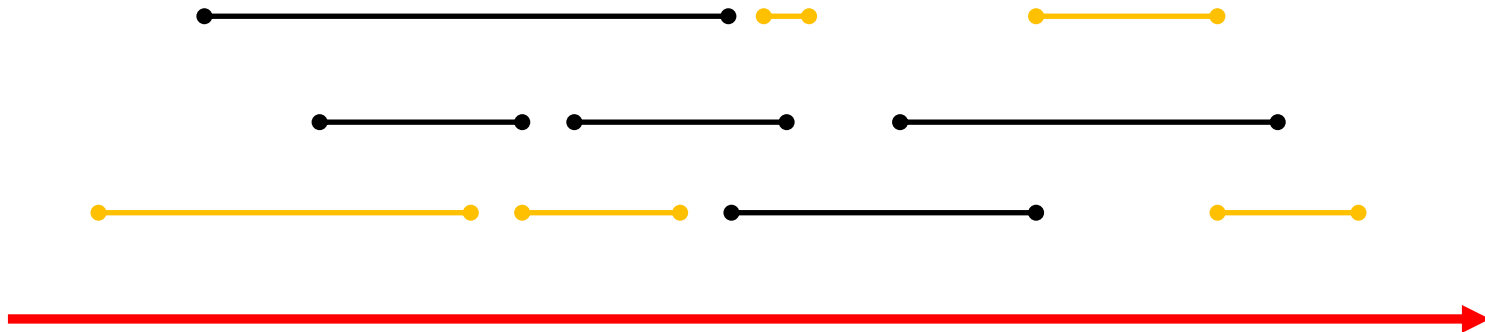
- Process request according to **finish time**
  - Begin with those **finishing earliest**



- **Always** optimal ?
- Try the previous examples !

# Earliest Finish Time

## ■ The first example



# Earliest Finish Time

- The greedy algorithm that selects requests according to their finishing time is **optimal** !
- How to implement it efficiently ?

# Greedy Algorithm

$R \leftarrow$  set of all requests

$A \leftarrow \{ \}$  // Selected activities

while (  $R$  is not empty ) do

    choose  $r$ , from  $R$ , with earliest finish time

    if(  $r$  does not overlap with any  $r$  in  $A$  )

$A \leftarrow A \cup \{ r \}$

return  $A$

# Implementation and Complexity

- **Pre-sort** all requests based on finish time
  - $O(n \log n)$
- Choosing the **next** candidate request is  $O(1)$
- Keep track of the finishing time of the **last request** added to A
- Check if the start time of the **next candidate** is later than that
  - Checking for overlapping is  $O(1)$
- $O(n \log n + n) = O(n \log n)$

# SOME PROBLEMS ON GRAPHS

# Some problems on graphs

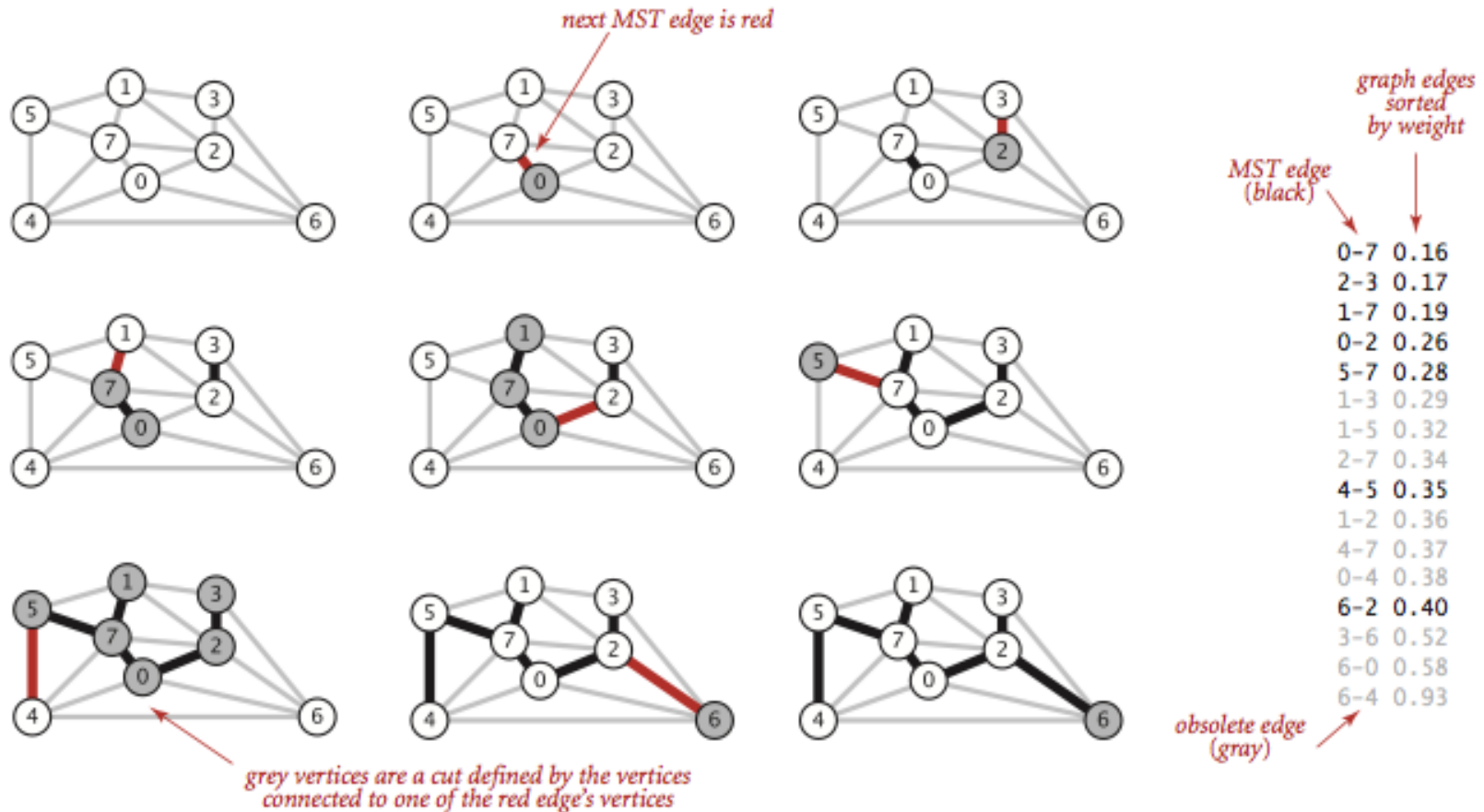
- Does a given **greedy** strategy **always** provides an **optimal** solution ?
- For **which problems** ?
- **MST** – **Minimum**-cost Spanning Tree
- **SSSP** – Single-Source **Shortest**-Paths



# MST – Minimum-cost Spanning Tree

- For ensuring connectivity with the **least cost**
- Kruskal's algorithm
  - Start with a forest of one-node trees
  - Successively add the **least-costly edge** that does not create a cycle

# Kruskal's algorithm



[Sedgewick & Wayne]

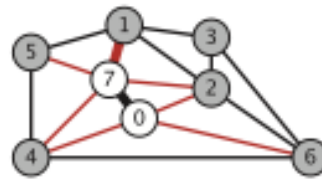
# MST – Minimum-cost Spanning Tree

- Prim's algorithm
  - Start with a one-node tree
  - Successively add the **closest node** that does not create a cycle

# Prim's algorithm

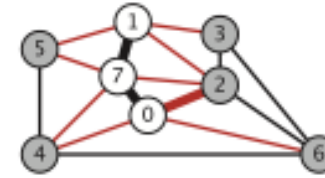


0-7 0.16  
0-2 0.26  
0-4 0.38  
6-0 0.58

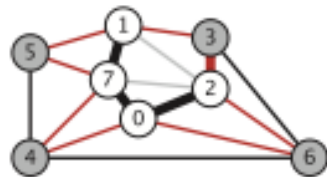


edges with exactly  
one endpoint in  $T$   
(sorted by weight)

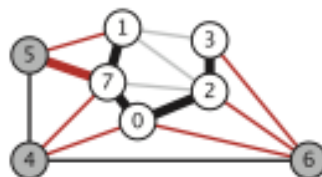
1-7 0.19  
0-2 0.26  
5-7 0.28  
2-7 0.34  
4-7 0.37  
0-4 0.38  
6-0 0.58



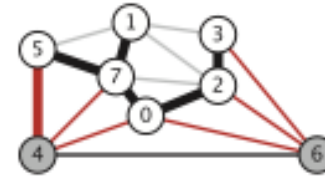
0-2 0.26  
5-7 0.28  
1-3 0.29  
1-5 0.32  
2-7 0.34  
1-2 0.36  
4-7 0.37  
0-4 0.38  
0-6 0.58



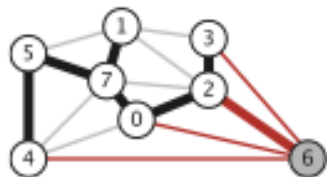
2-3 0.17  
5-7 0.28  
1-3 0.29  
1-5 0.32  
4-7 0.37  
0-4 0.38  
6-2 0.40  
6-0 0.58



5-7 0.28  
1-5 0.32  
4-7 0.37  
0-4 0.38  
6-2 0.40  
3-6 0.52  
6-0 0.58



4-5 0.35  
4-7 0.37  
0-4 0.38  
6-2 0.40  
3-6 0.52  
6-0 0.58

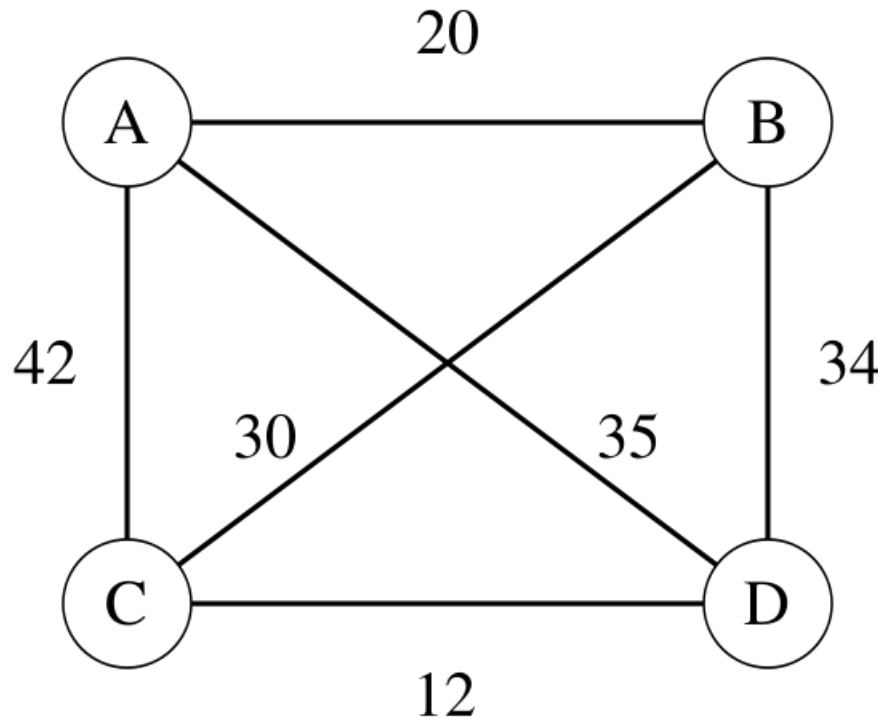


6-2 0.40  
3-6 0.52  
6-0 0.58  
6-4 0.93



[Sedgewick & Wayne]

# MST – Minimum-cost Spanning Tree

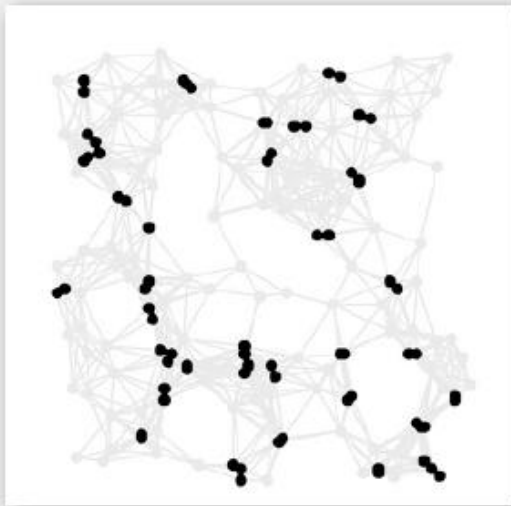


[Wikipedia]

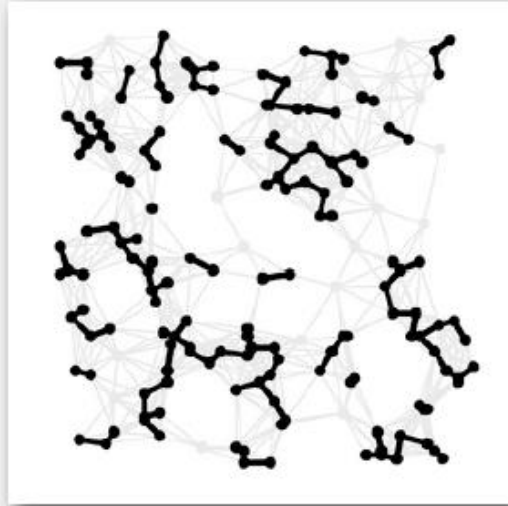
- What is the solution ?
- Apply both algorithms !!

# Prim's or Kruskal's ?

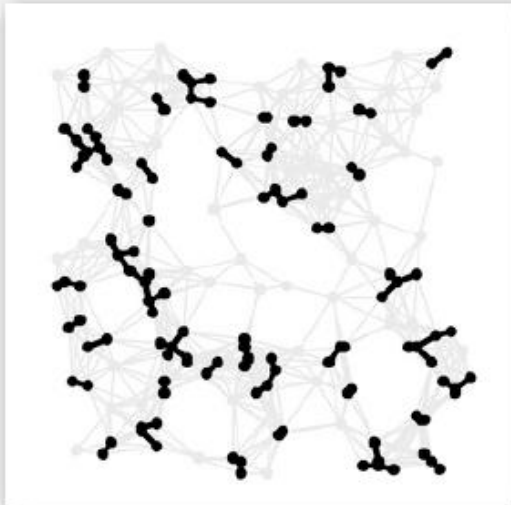
25%



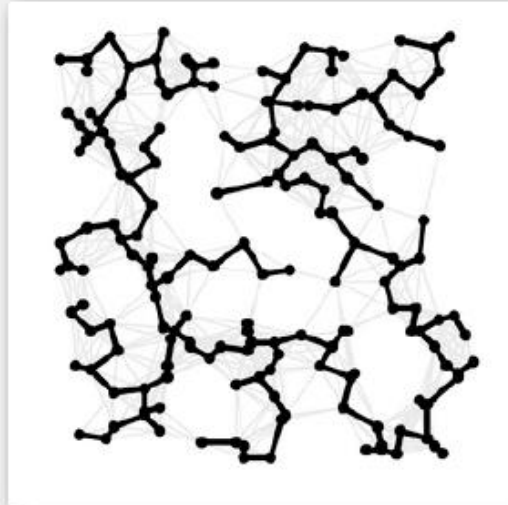
75%



50%

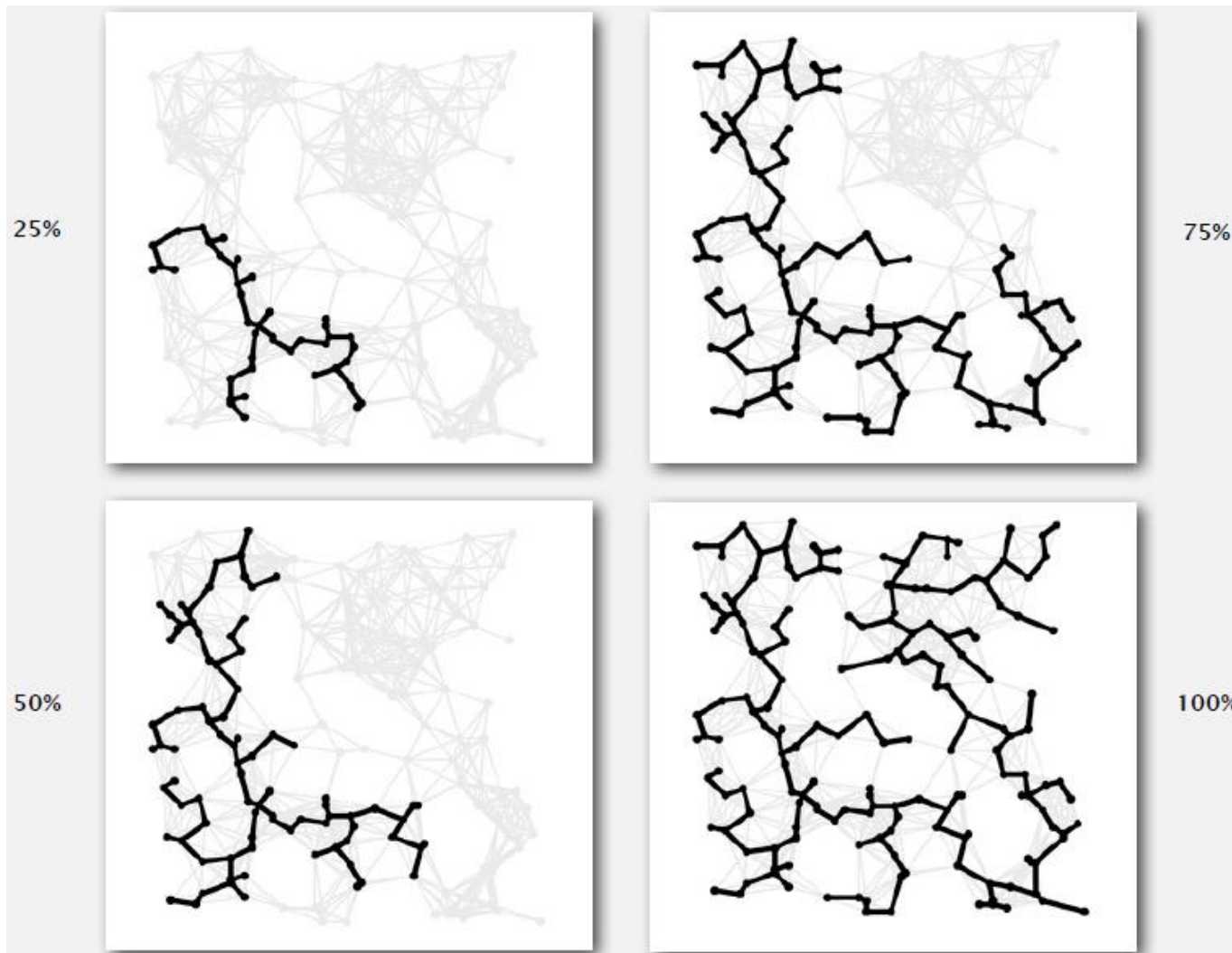


100%



[Sedgewick & Wayne]

# Prim's or Kruskal's ?



[Sedgewick & Wayne]

# The Single-Source Shortest-Paths Problem

## ■ Given

- A weighted connected graph :  $G ( V, E )$
- Edges with non-negative weights !!
- A source node :  $s$

## ■ Find

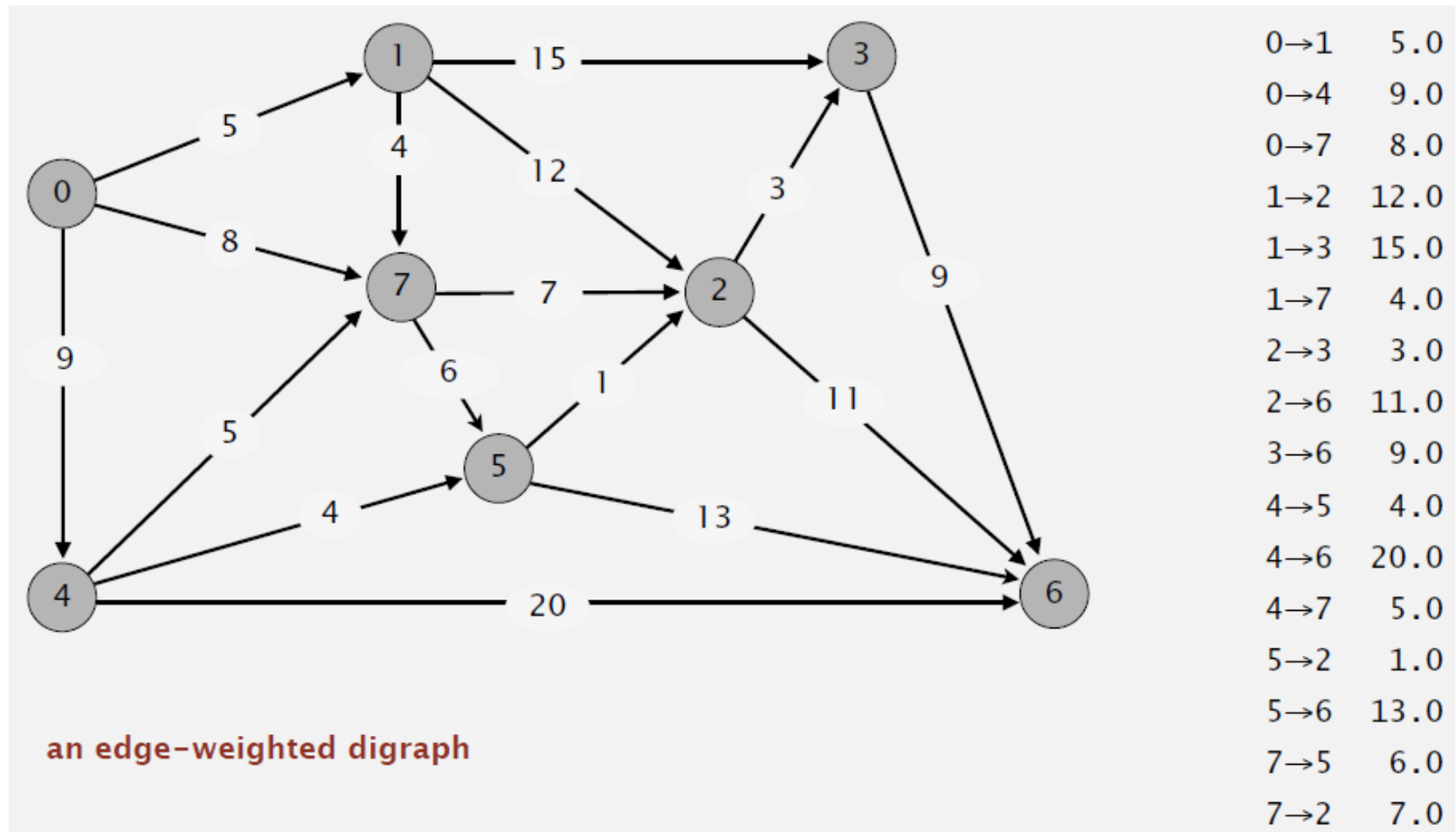
- The shortest paths from  $s$  to every other node in  $G$



# Dijkstra's algorithm

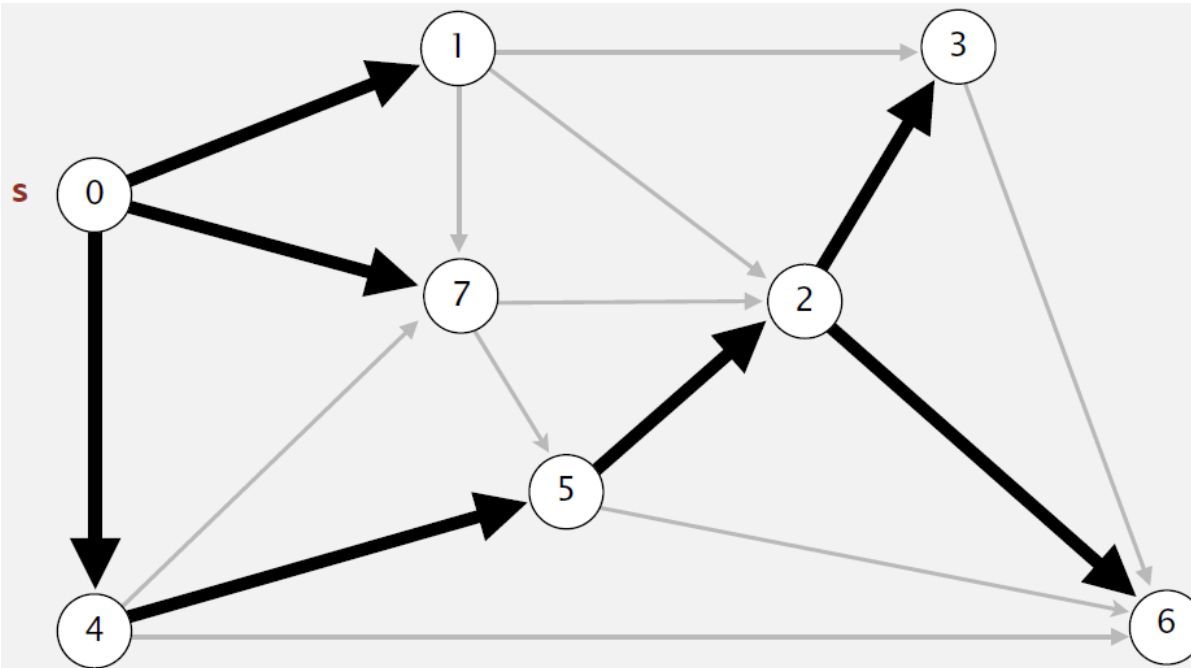
- Computes the **Shortest-Paths Tree (SPT)**, rooted in  $s$
- Finds shortest paths to graph nodes in **order of their distance** to source node  $s$
- Next node to add to the SPT ?
  - It has the **current shortest distance** to the source node
- Keep the set of **candidate nodes** not belonging to the tree !

# Dijkstra's algorithm



[Sedgewick & Wayne]

# Dijkstra's algorithm

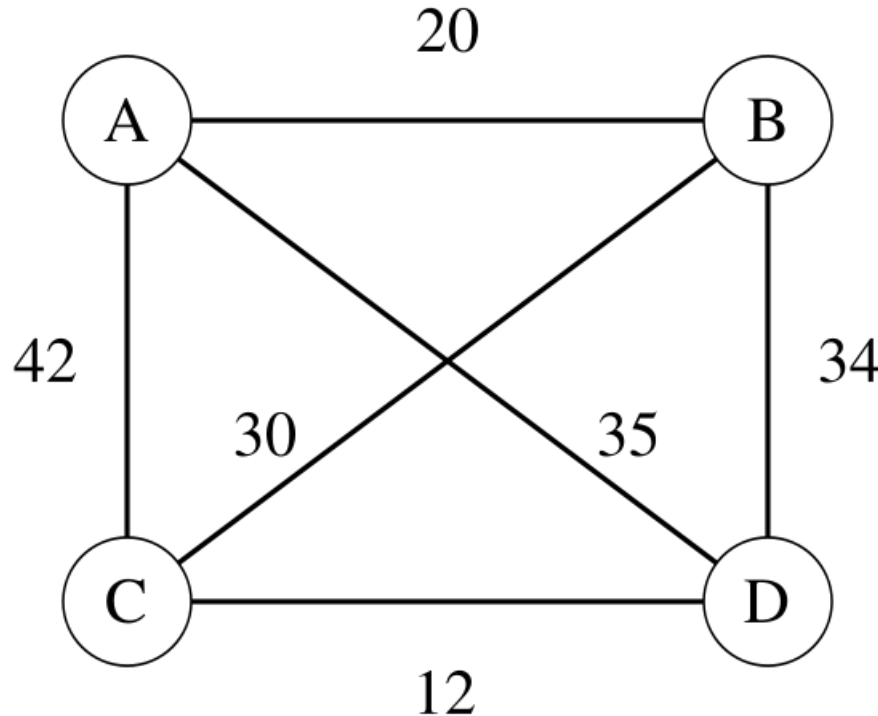


**shortest-paths tree from vertex s**

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

[Sedgewick & Wayne]

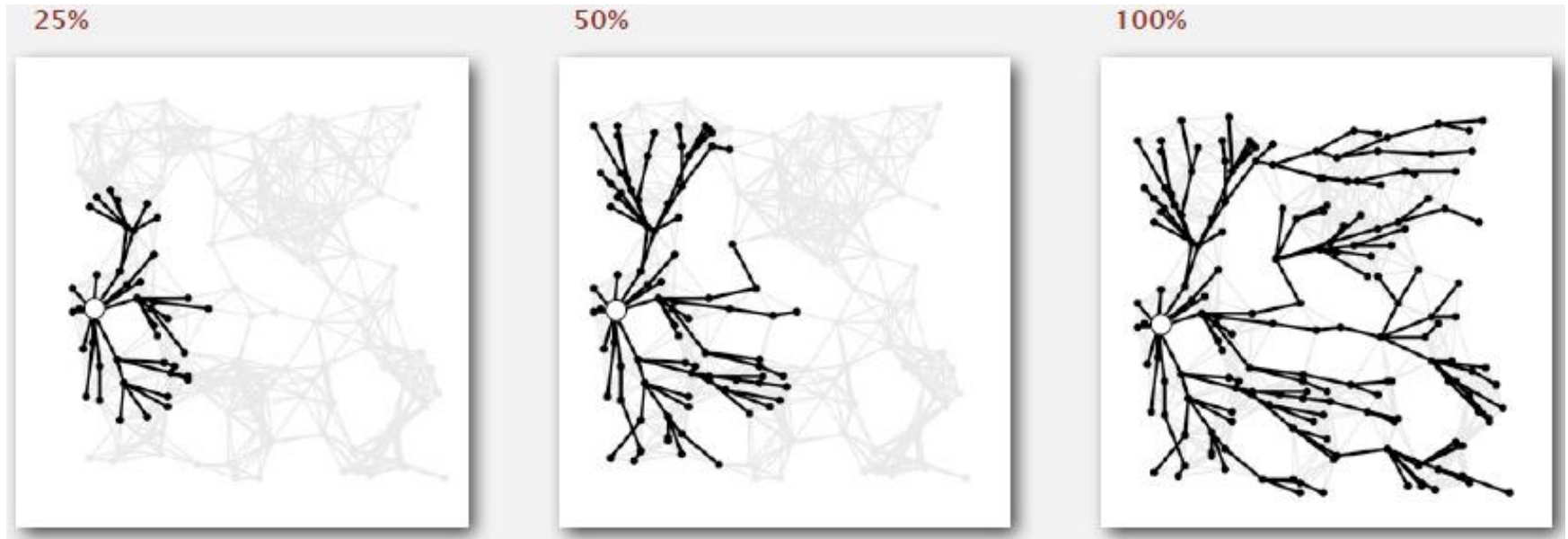
# Dijkstra's algorithm



[Wikipedia]

- What is the solution ?
- Apply Dijkstra's algorithm !!

# Dijkstra's algorithm



[Sedgewick & Wayne]

---

# THE TRAVELING SALESMAN PROBLEM

# The Traveling Salesman Problem

- Find the **shortest tour** through a given **set of  $n$  cities**
- BUT, **visiting** each city **just once !**
- AND **returning to the starting city !**



[Wikipedia]

# The Traveling Salesman Problem

- Use a weighted graph  $G$  to model the problem
- Find the **shortest Hamiltonian circuit** of  $G$ 
  - Cycle of least cost /distance
  - Passes (just once) through all vertices
- **NP-complete** problem !!



# The Traveling Salesman Problem

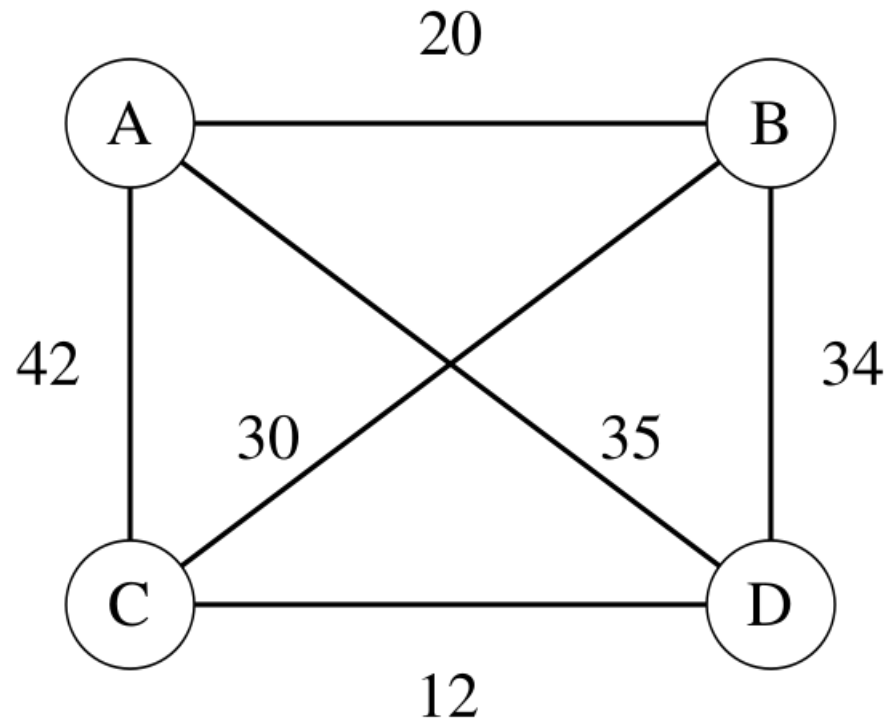
- Hamiltonian circuit

- Sequence of  $(n + 1)$  adjacent vertices
- The first vertex is the same as the last !

- How to proceed ?

- Choose **any** one vertex as the **starting point**
- Generate the  $(n - 1)!$  **possible permutations** of the intermediate vertices
- For each such cycle, compute its **cost / distance**
- And keep the **less expensive / shortest one**

# The Traveling Salesman Problem



[Wikipedia]

- What is the solution ?

# The Traveling Salesman Problem

## ■ Questions

- ❑ How do we store the graph ?
- ❑ Is it **complete** ?
- ❑ How to generate all **permutations** ?

## ■ Efficiency

- ❑  $O(n!)$
- ❑ Exhaustive search can only be applied to **very small instances** !! Alternatives ?
- ❑ Slight improvements are still possible

# TASKS

# Task – V1

- Implement a function for computing the / a solution to an instance of the TSP
  - Count the number of cycles tested
  - Count the number of times the current best solution is updated
- Use Python's **combinatoric generator**  
`permutations( )`
- Improve your function to avoid checking duplicates
  - I.e., any cycle and its reverse cycle

# COMPUTING APPROXIMATE SOLUTIONS

# Approximate Solutions

- Do not attempt to compute **exact solutions** to difficult combinatorial optimization problems
  - **It might take too long !!**
  - Real-world : **inaccurate data**
  - Approximations might suffice !!
- Compute **approximate solutions**
  - E.g., use **greedy** heuristics !!
  - Evaluate the **accuracy** of such approximations
    - Performance ratio

# Approximation Accuracy

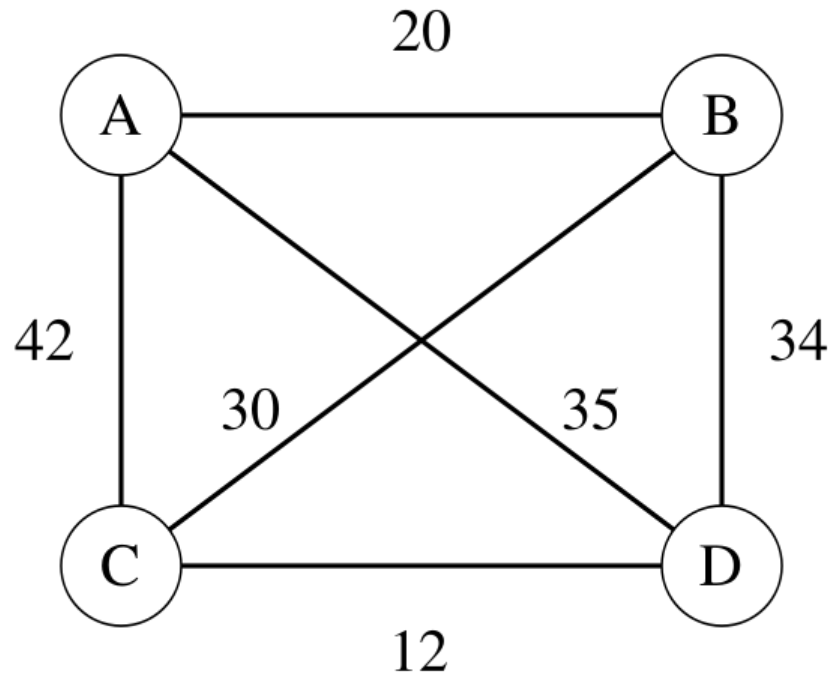
- Minimize function  $f()$
- Approximate solution :  $s_a$
- Exact solution :  $s^*$
- Relative error :  $re(s_a) = ( f(s_a) - f(s^*) ) / f(s^*)$
- Accuracy ratio :  $r(s_a) = f(s_a) / f(s^*)$
- Performance ratio :  $R_A$ 
  - The lowest upper bound of possible  $r(s_a)$  values
  - Should be as close to 1 as possible
  - Indicates the quality of the approximation algorithm



# The Traveling Salesman Problem

- **Nearest-neighbor** heuristic – **Greedy** !!
  - Always go to the nearest unvisited city
- Choose an arbitrary city as the start
- Repeat until all cities have been visited
  - Go to the unvisited city nearest to the last
- Return to the starting city
- Simple heuristic
- But  $R_A = \infty$  !!

# The Traveling Salesman Problem



- What is the optimal solution ?
- Apply the nearest-neighbor algorithm !
- Accuracy ratio ?

# The Traveling Salesman Problem

- There are other simple heuristics, for instance:
- Bidirectional-Nearest-Neighbor
- Shortest-Edge
- Apply them to the previous example !!

---

# TASKS

# Tasks – V2 + V3 + V4

- Implement functions for computing approximate solutions to an instance of the TSP
- Use the **three heuristics** presented
- Check the **accuracy** of the obtained solutions !!

---

# REFERENCES

# References

- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3<sup>rd</sup> Ed., Pearson, 2012
  - Chapter 3 + Chapter 9 + Chapter 12
- R. Johnsonbaugh and M. Schaefer, *Algorithms*, Pearson Prentice Hall, 2004
  - Chapter 7 + Chapter 11
- T. H. Cormen et al., *Introduction to Algorithms*, 3<sup>rd</sup> Ed., MIT Press, 2009
  - Chapter 16 + Chapter 23 + Chapter 24 + Chapter 35