Algorithm Design Strategies V

Joaquim Madeira

Version 0.4 – October 2024

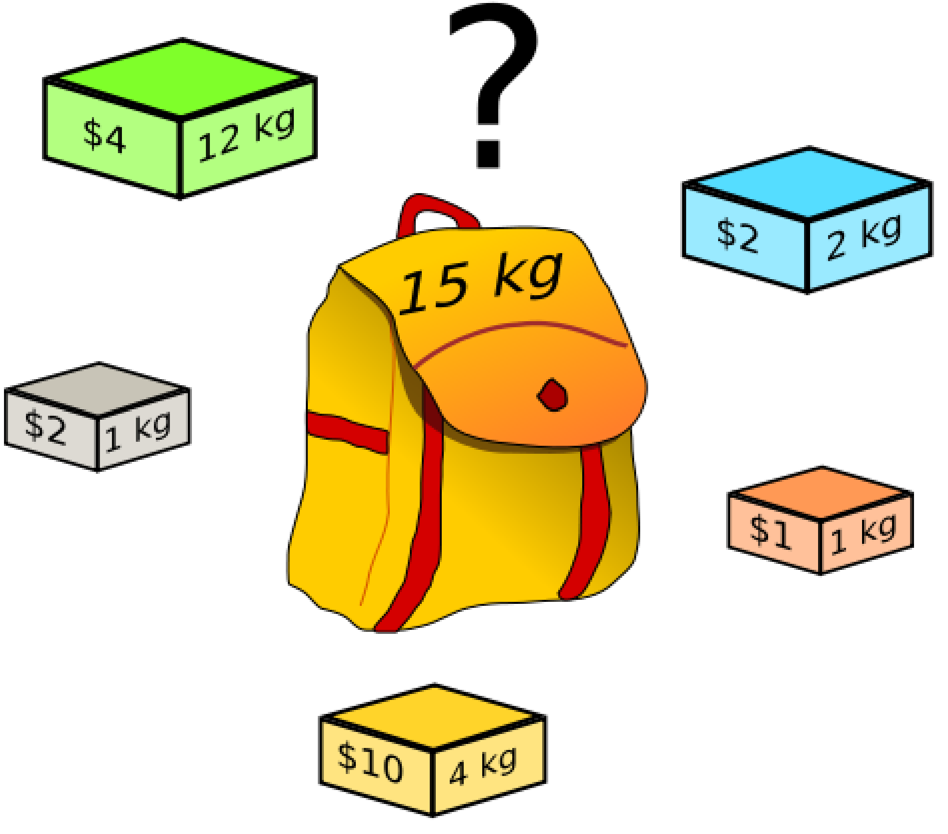
Overview

- 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 $w_1, w_2, \ldots, w_n$
  - Known value $v_1, v_2, \ldots, v_n$
- 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 ?

max $\sum x_i v_i$

subject to $\sum x_i w_i \leq W$

with x_i 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 !!

- We also applied some simple heuristics
- Can we use a better, “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

  Sorting -> O(n logn)
- Repeat until no item is left in the sorted list constructing solution O(n)
  - If the current item fits, place it in the knapsack
  - Otherwise, discard it

Overall Complexity - O(n logn)

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 ←

Not Always the Optimal Solution

- 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

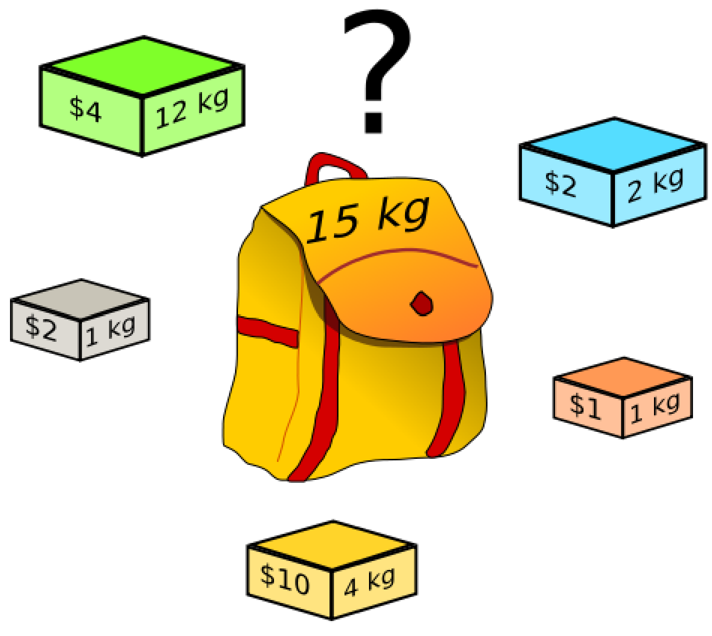
Tasks

- 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 !!
  (Always gives the optimal solution - Greedy Algorithm)

- 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 (no back tracking)

- Does it always work ?

THE
COIN-CHANGING PROBLEM

The Coin-Changing Problem

- Make change for an amount A

(tipo de moeda - 1 euro, 50,20,10,5,2,1 centimos)

- Available coin denominations
  - Denom[1] > Denom[2] > … > Denom[n] = 1

- Use the fewest number of coins !!

- Assumption
  - Enough coins of each denomination !!

The Coin-Changing Problem (amount A)

- How to formulate ?

$$\min \sum x_i$$

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

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

- Compare with the 0-1 Knapsack formulation

The Coin-Changing Problem (For european coins always optimal)

```
i ← 1
while ( A > 0 ) do
    c ← A  div  denom [ i ]
    output ( c coins of denom [ i ] value )
    A ← A – c x denom [ i ]
    i ← i + 1
```

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

The Coin-Changing Problem

- Does the algorithm terminate ?

- Complexity ?

- Best Case ?
  - Number of iterations ? 1
  - When ? Multiplo de maior moeda
  - How many coins ? c

- Worst Case ?
  - Number of iterations ? As many iterations as there are coins
  - When ? Need last coin
  - How many coins ? whatever

The Coin-Changing Problem

- Is this greedy approach always optimal ?

- It depends on the set of denominations !
  (For european coins always optimal)

- Example
  - Denom[1] = 7 ; Denom[2] = 5 ; Denom[3] = 1
  - A = 10
  - How many coins ? duas de 5

  - 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

*The idea is to fill the **DP table** based on **previous** values. For each coin, we either **include** it or **exclude** it to compute the **minimum** number of coins needed for each sum. The table is filled in an iterative manner from **i = n-1** to **i = 0** and for each sum from 1 to sum.*

The dynamic programming relation is as follows:

- *if **(sum-coins[i])** is **greater** than 0, then **dp[i][sum] = min(1+dp[i][sum-coins[i]], dp[i+1][sum])***
- *else **dp[i][sum] = dp[i+1][sum].***

Denom[1] = 7 ; Denom[2] = 5 ; Denom[3] = 1;   A=10

I DON'T FUCKING KNOW

TASKS

Tasks

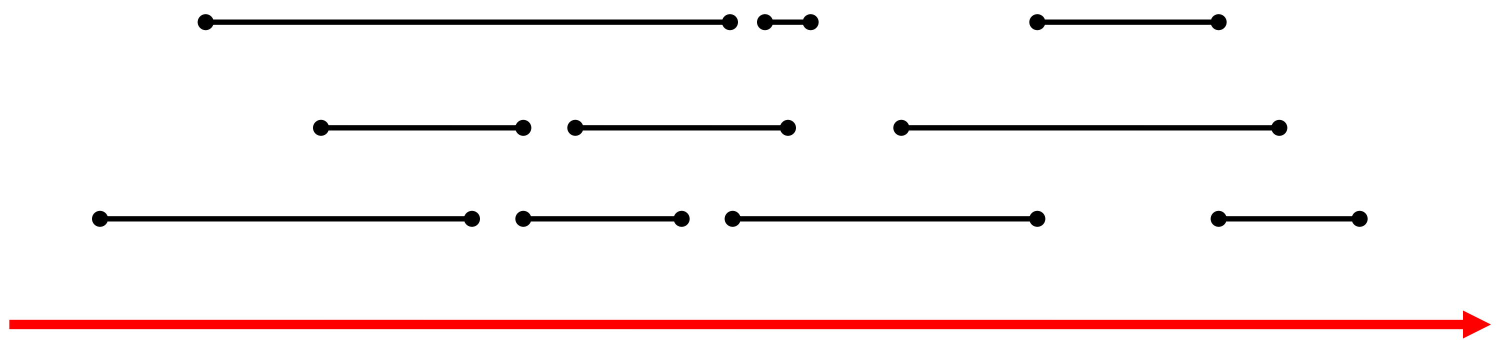
- Implement a greedy function for computing the solution to an instance of the Coin-Changing Problem
- Run it for a set of test instances

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 !

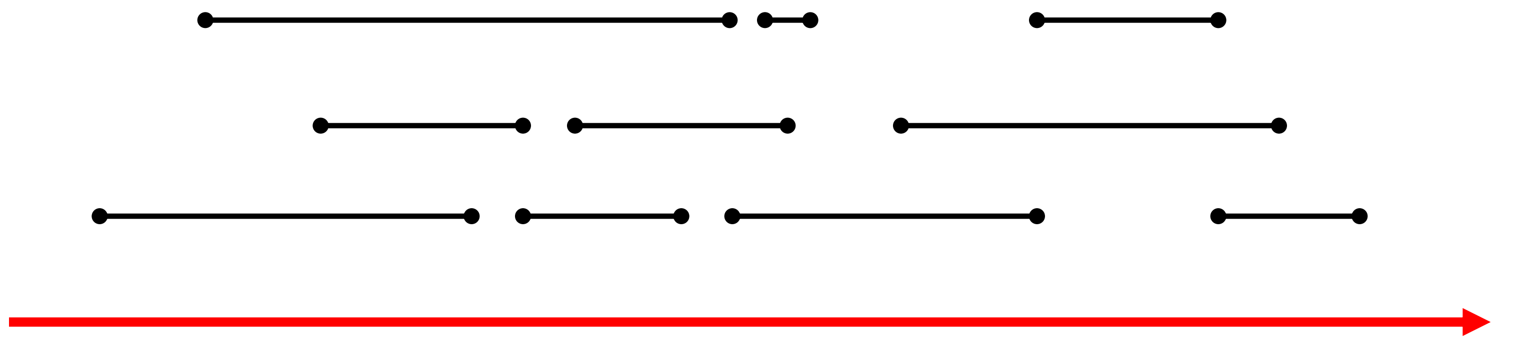
- Greedy strategy:
  - Possible heuristics ?
  - Ensure optimal solutions ?

Greedy Algorithm

```
R ← set of all requests
A ← { }                                  // Selected activities
while ( R is not empty ) do
        choose r from R                  // How ?
        A ← A U { r }
        R ← R – {pending requests overlapping r}
return A
```

Earliest Start Time

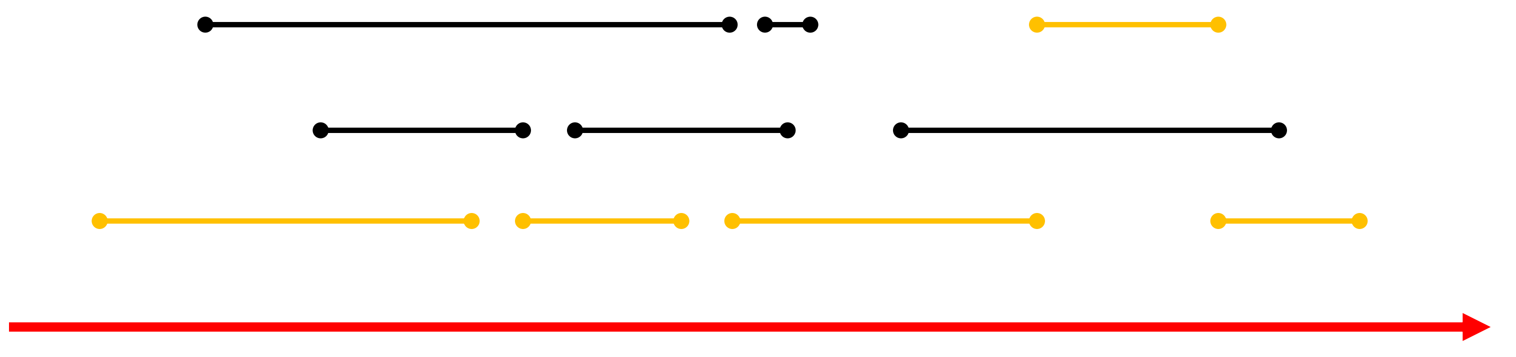
- Process requests according to start time
  - Begin with those that start earliest



- Always optimal ?

Earliest Start Time

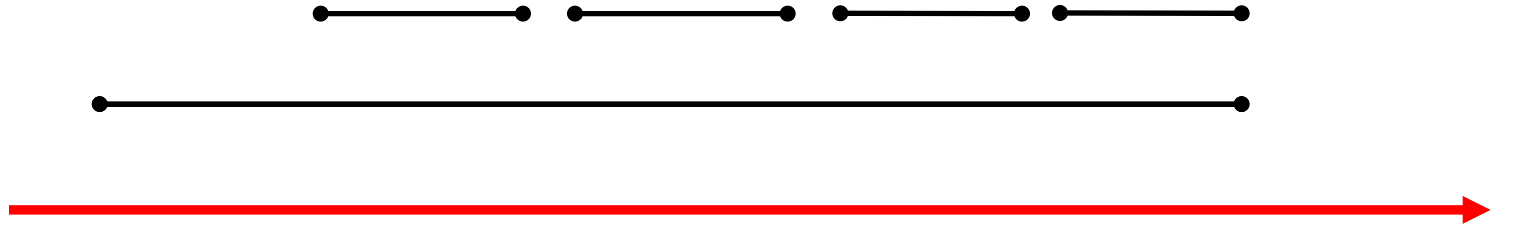
- Process requests according to start time
  - Begin with those that start earliest



- Always optimal ?

Earliest Start Time

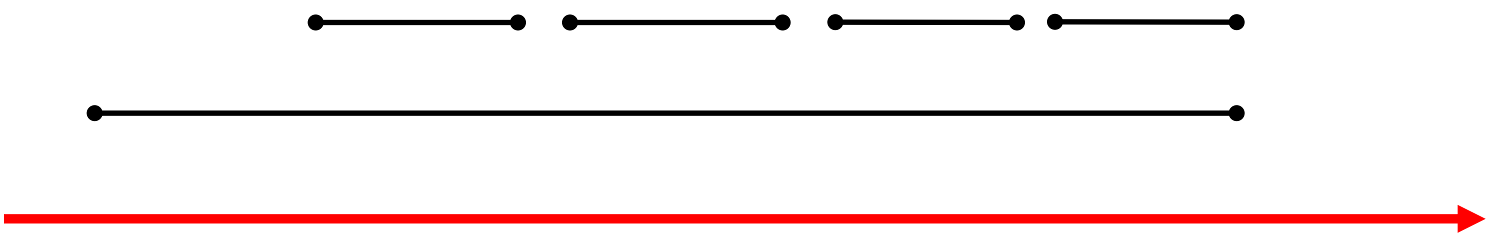
- Counter example



- ☹ NOT OPTIMAL

Smallest Processing Time

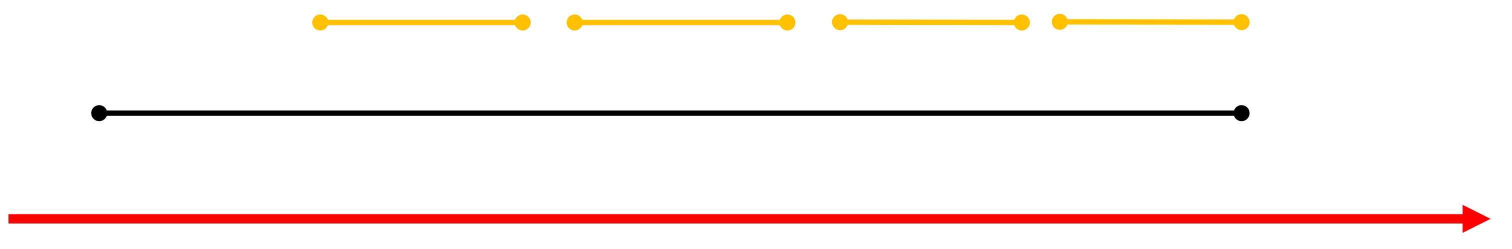
- Process requests according to duration
  - Begin with those requiring the shortest processing



- Always optimal ?

Smallest Processing Time

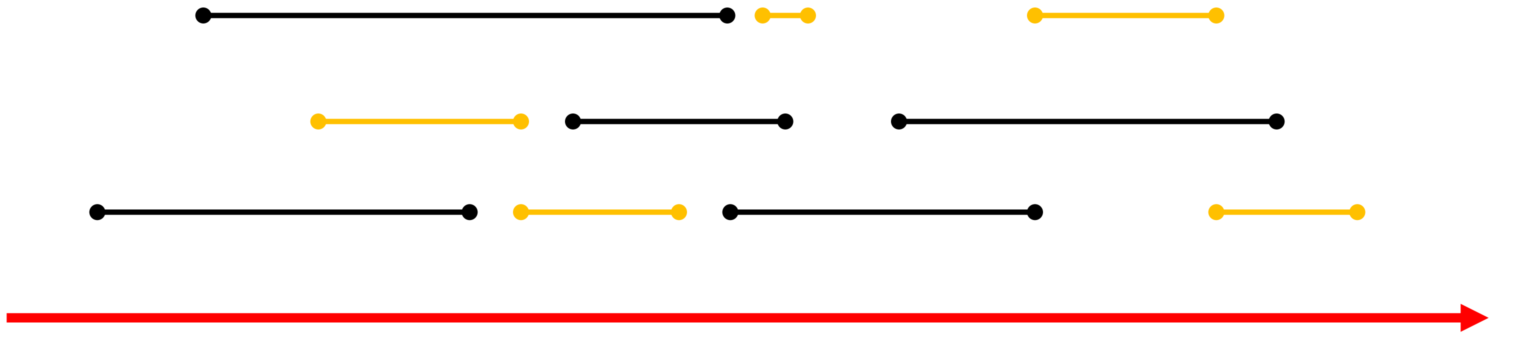
- Process requests according to duration
  - Begin with those requiring the shortest processing



- Always optimal ?

Smallest Processing Time

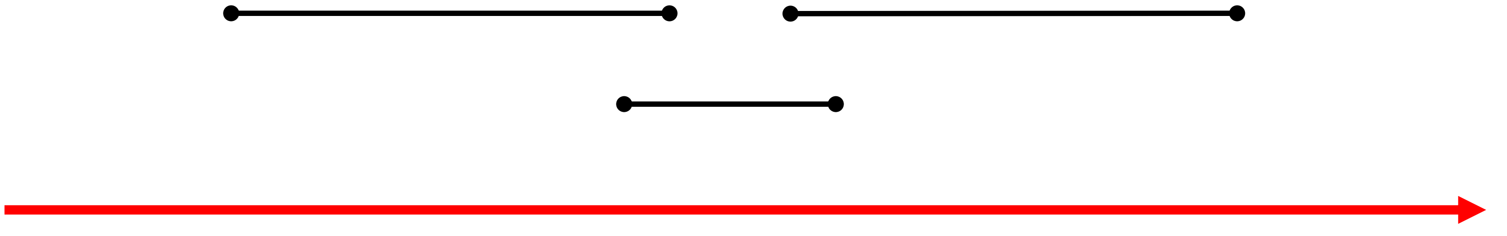
- The first example



- Always optimal ?

Smallest Processing Time

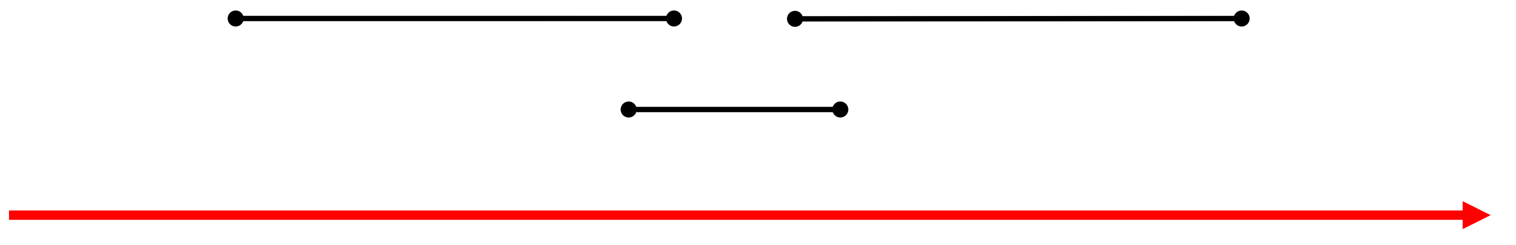
- Counter example



- ☹ Not Always the Optimal Solution

Fewest Conflicts

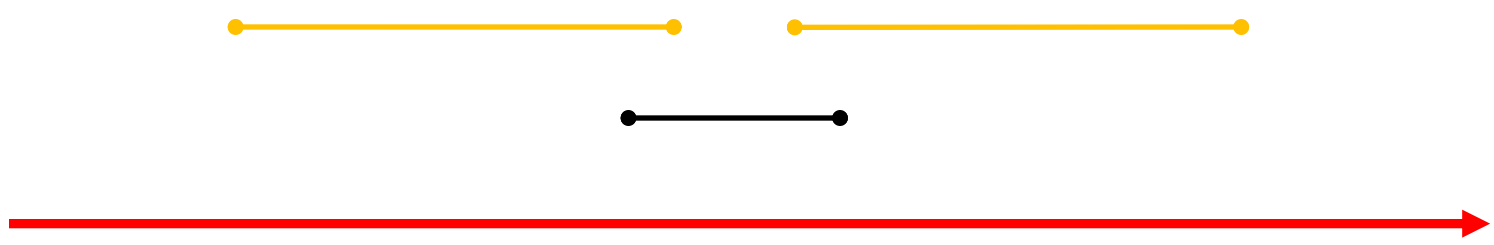
- Determine and update number of conflicts
  - Begin with those having the fewest conflicts



- Always optimal ?

Fewest Conflicts

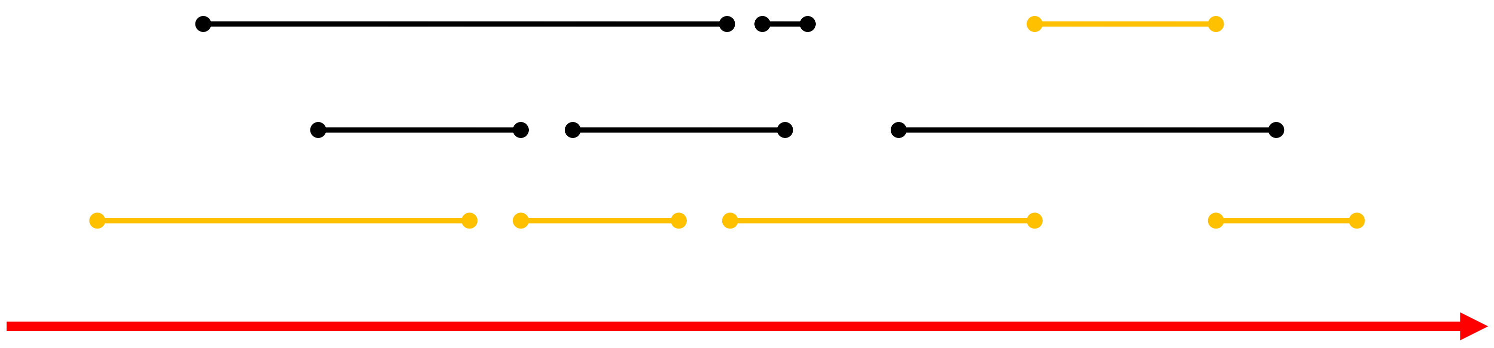
- Determine and update number of conflicts
  - Begin with those having the fewest conflicts



- Always optimal ?

Fewest Conflicts

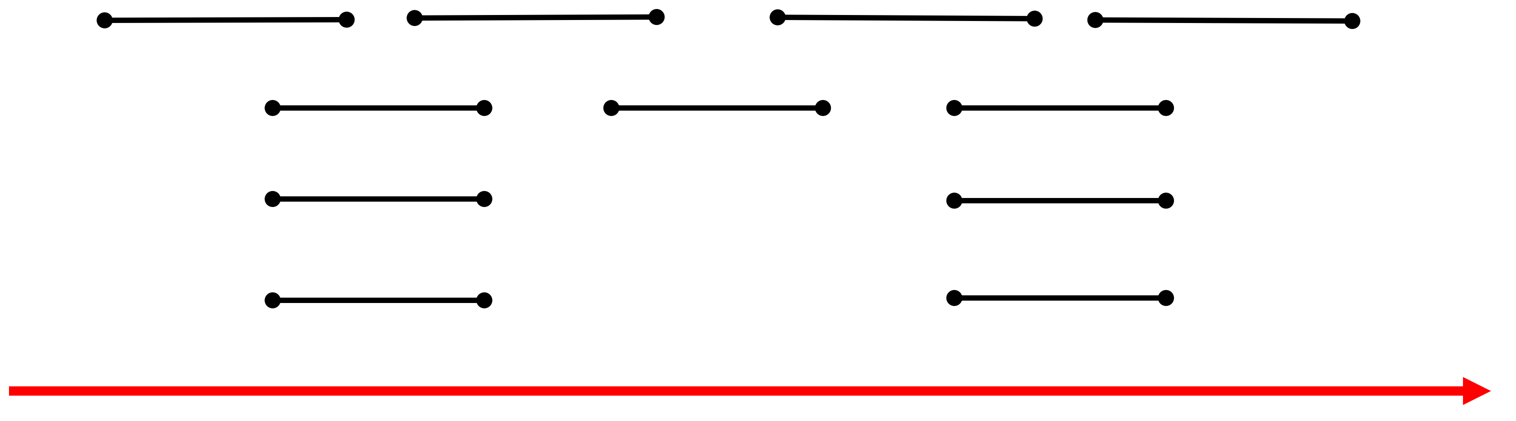
- The first example



- Always optimal ?

Fewest Conflicts

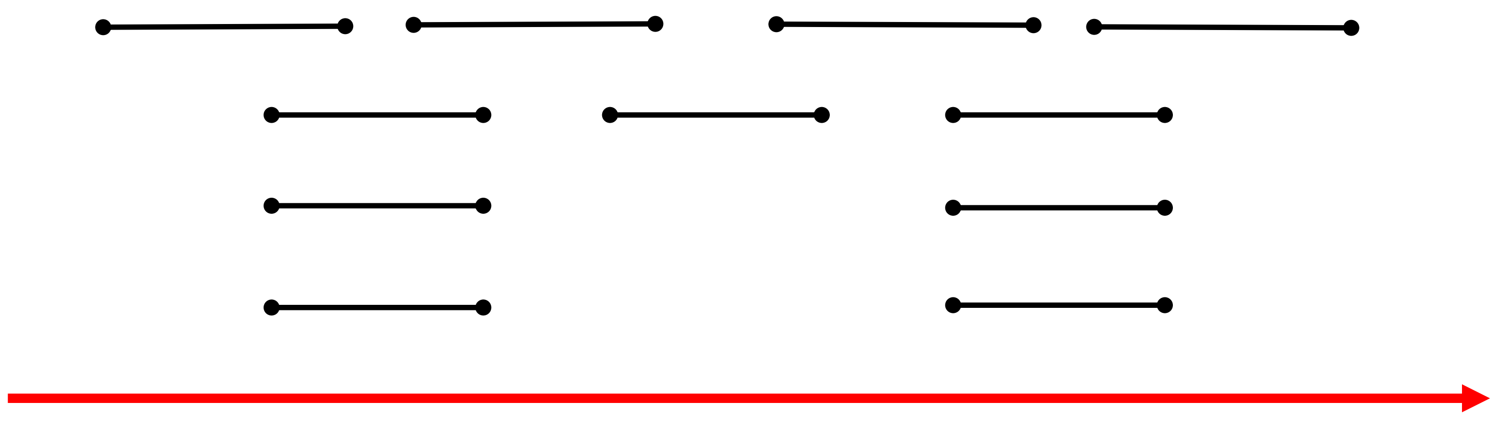
- Counter example



- ☹ Not Always the Optimal Solution

Earliest Finish Time

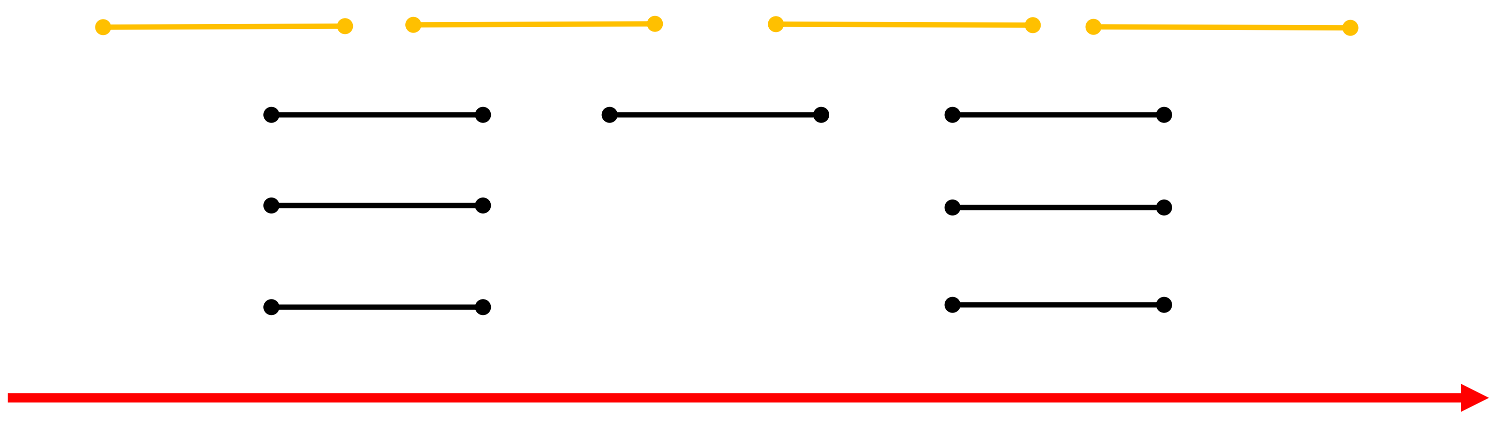
- Process request according to finish time
  - Begin with those finishing earliest



- Always optimal ?
- Try the previous examples !

Earliest Finish Time

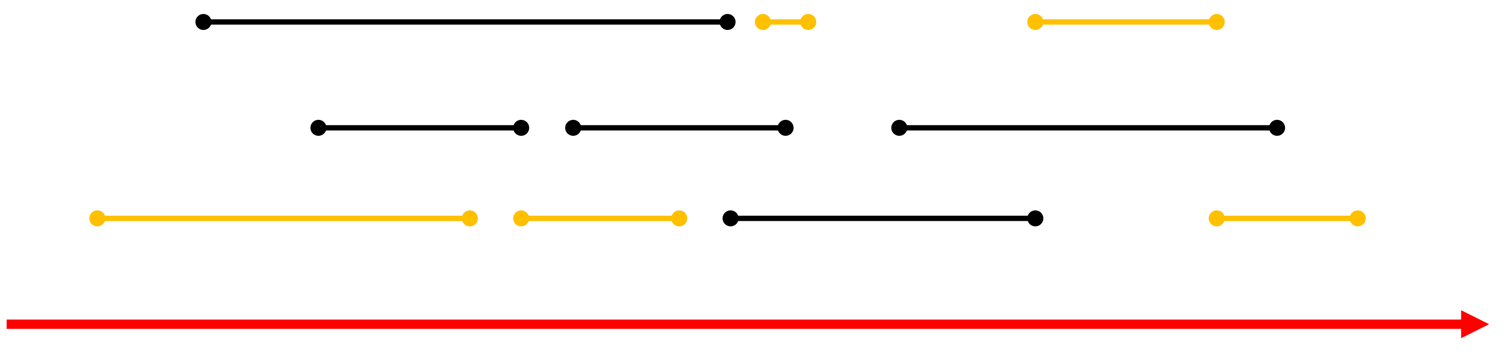
- Process request according to finish time
  - Begin with those finishing earliest



- Always optimal ?
- Try the previous examples !

Earliest Finish Time

- The first example



ALWAYS OPTIMAL

Earliest Finish Time

- The greedy algorithm that selects requests according to their finishing time is optimal !

- How to implement it efficiently ?

Greedy Algorithm

```
R ← set of all requests
A ← { }                                   // 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 ← A U { 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 ?

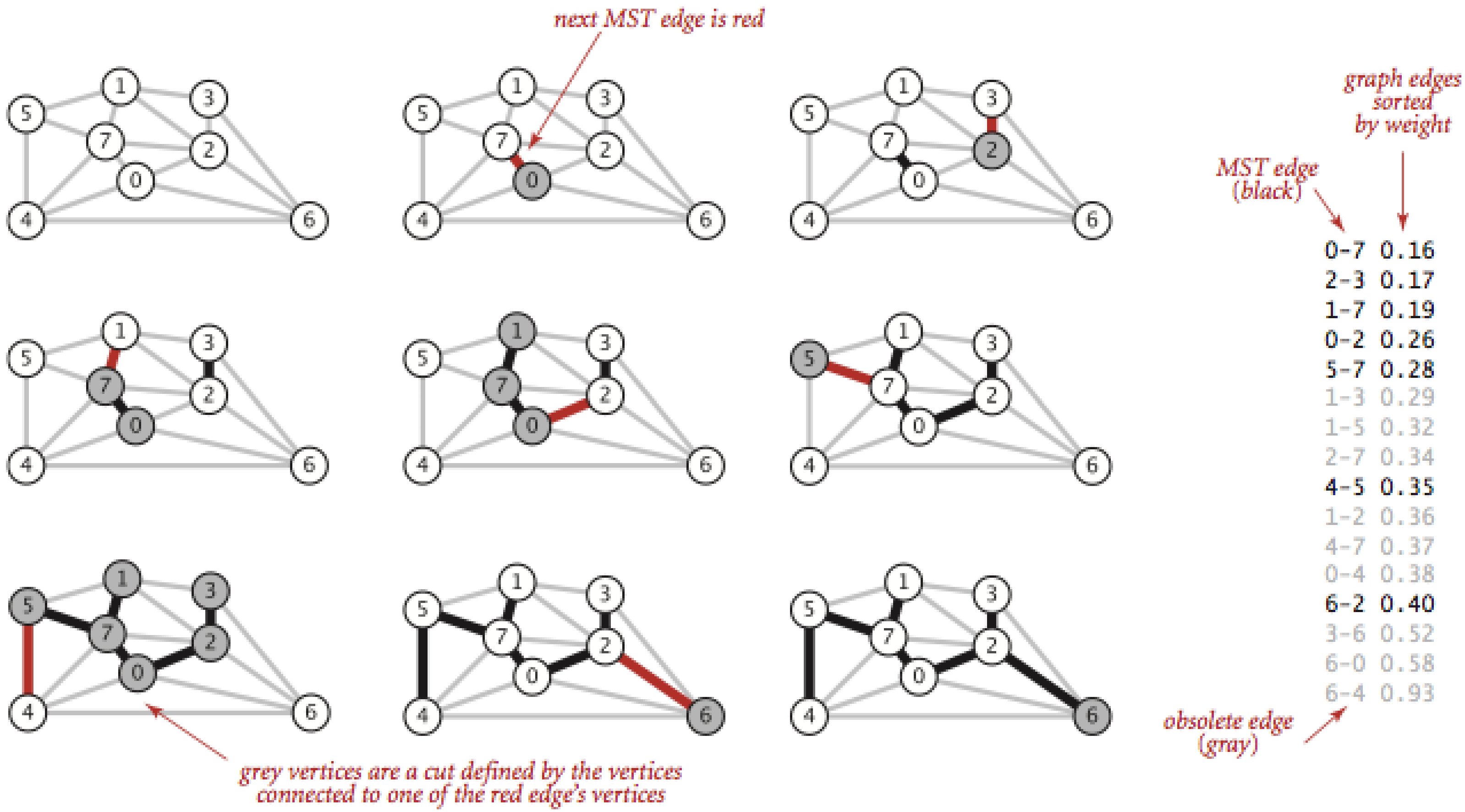
- MST – Minimum-cost Spanning Tree (Kruskal, Prim)
- SSSP – Single-Source Shortest-Paths (Dijkstra)

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

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Kruskal's algorithm



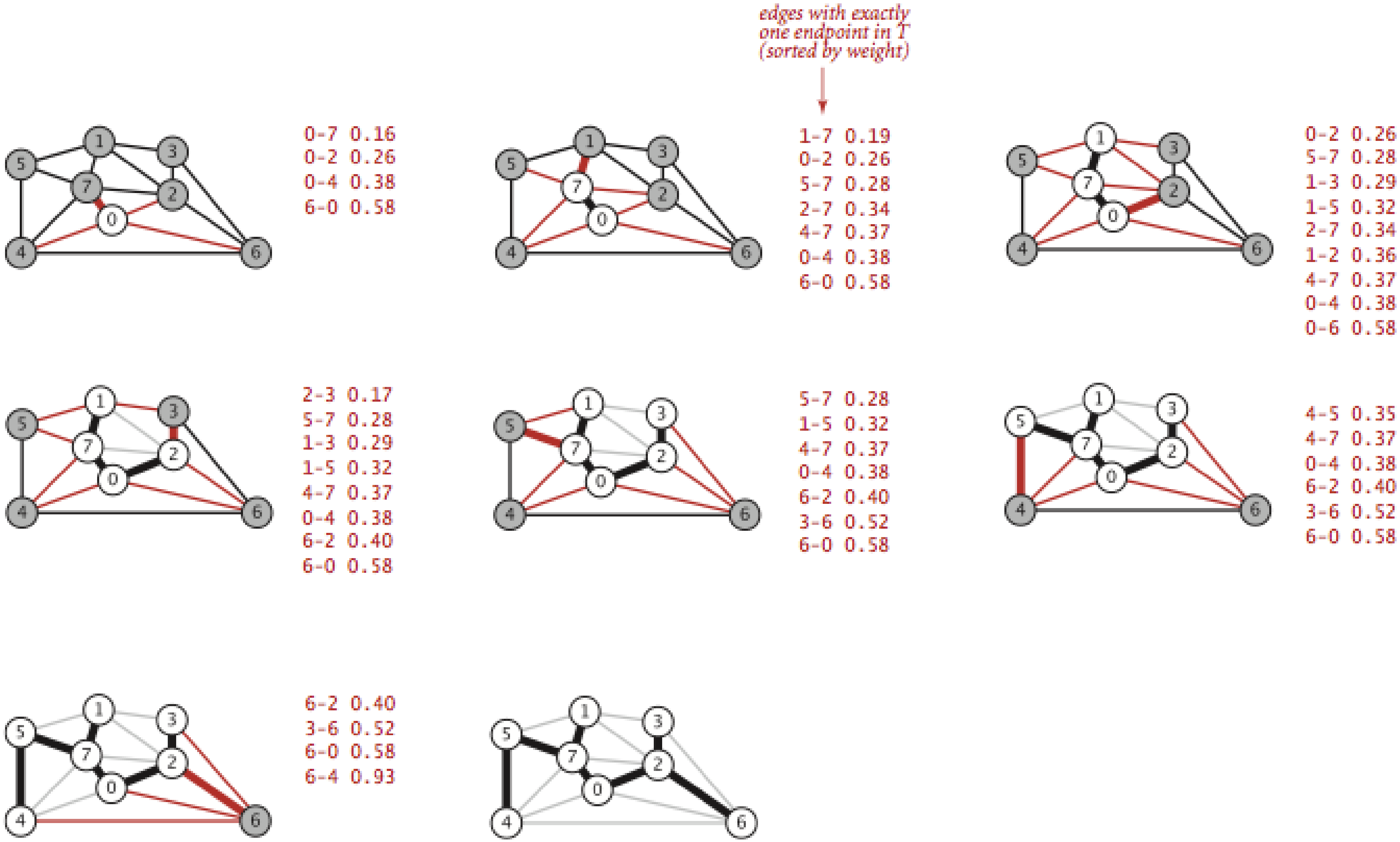
[Sedgewick & Wayne]

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

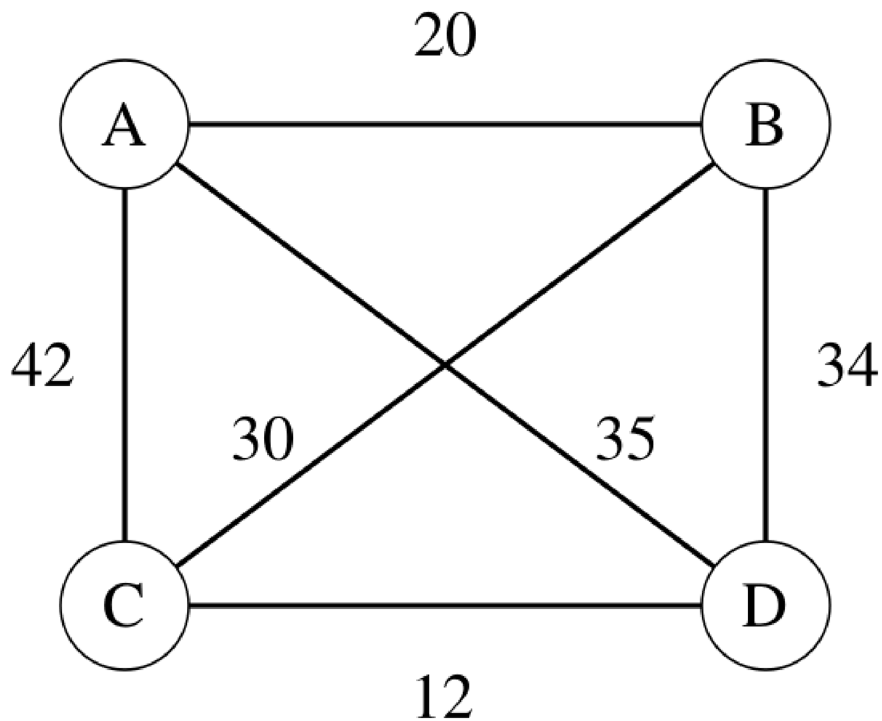
The closest node is NOT FROM the source node... Is overall to any node in the graph

Prim's algorithm



[Sedgewick & Wayne]

MST – Minimum-cost Spanning Tree

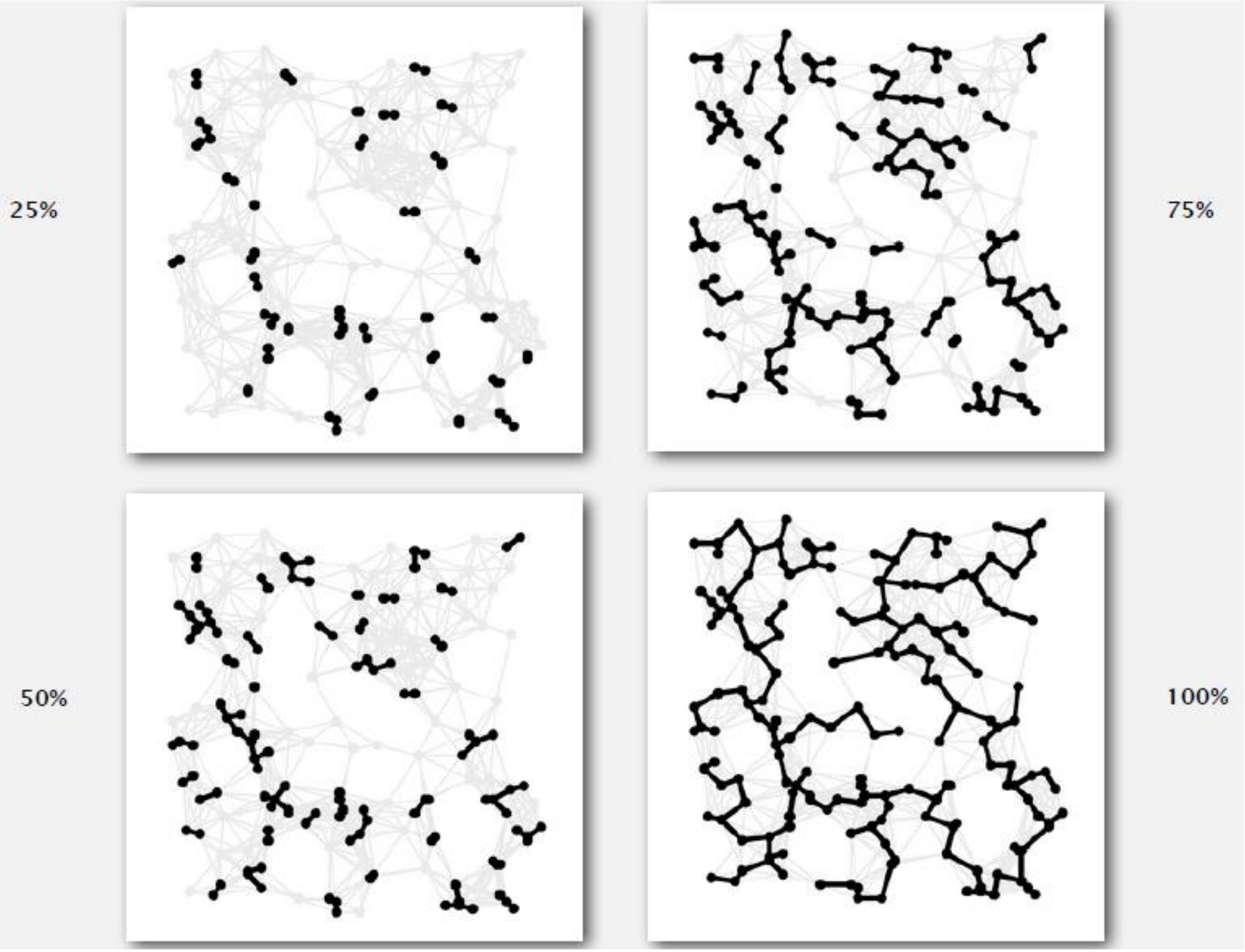


[Wikipedia]

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

Prim's or Kruskal's ?

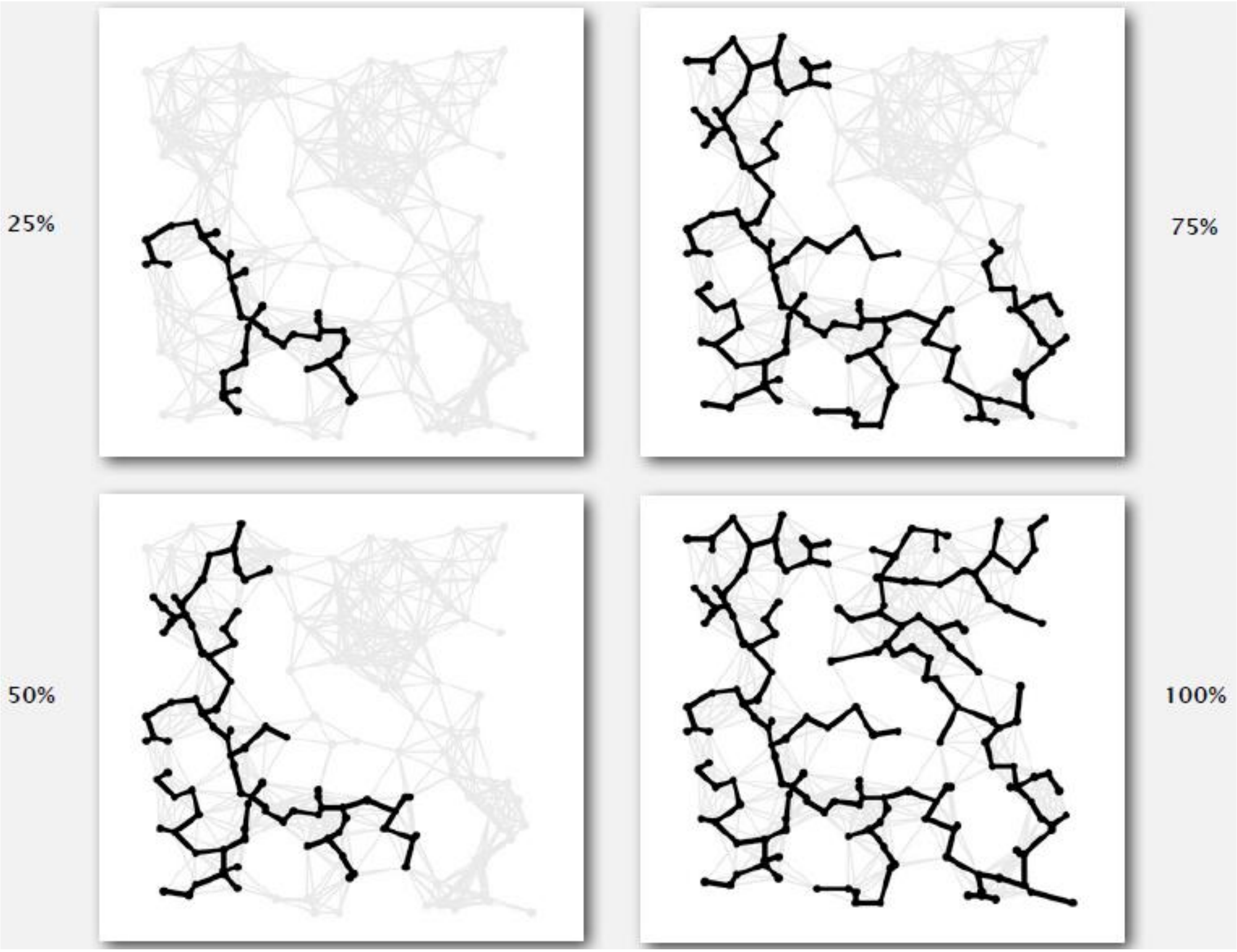
Kruskal's



[Sedgewick & Wayne]

Prim's or Kruskal's ?

Prim's



[Sedgewick & Wayne]

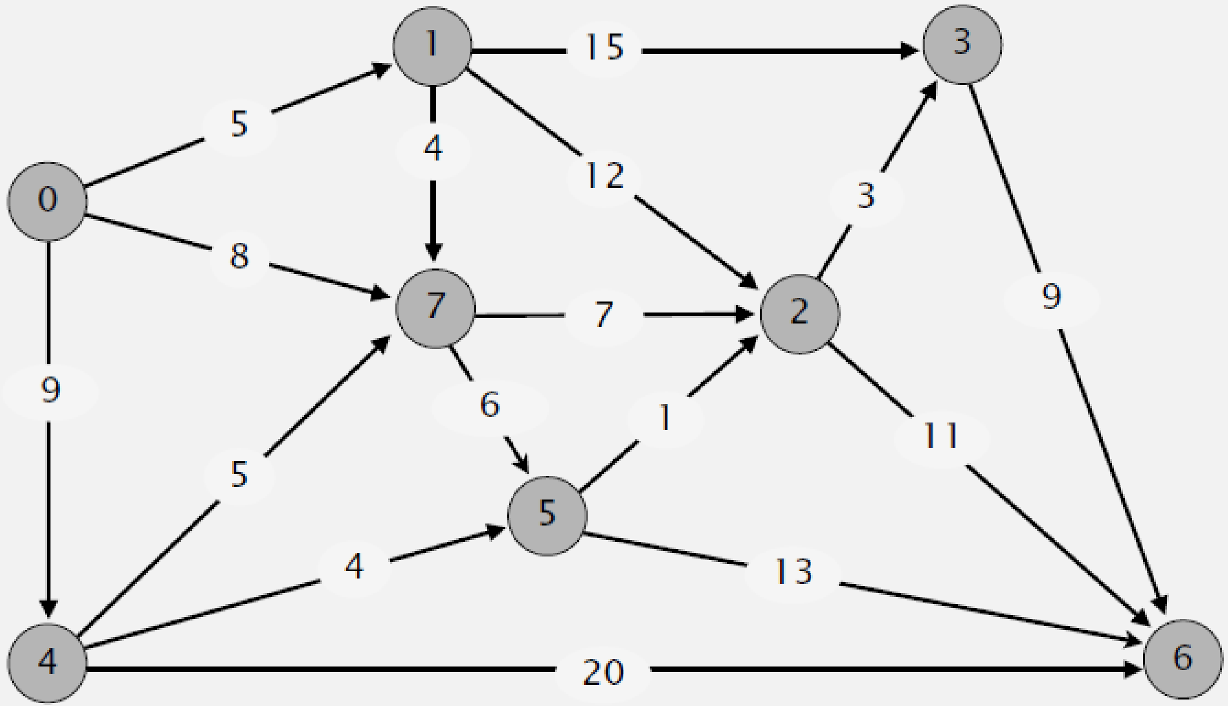
The Single-Source Shortest-Paths Problem

- 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

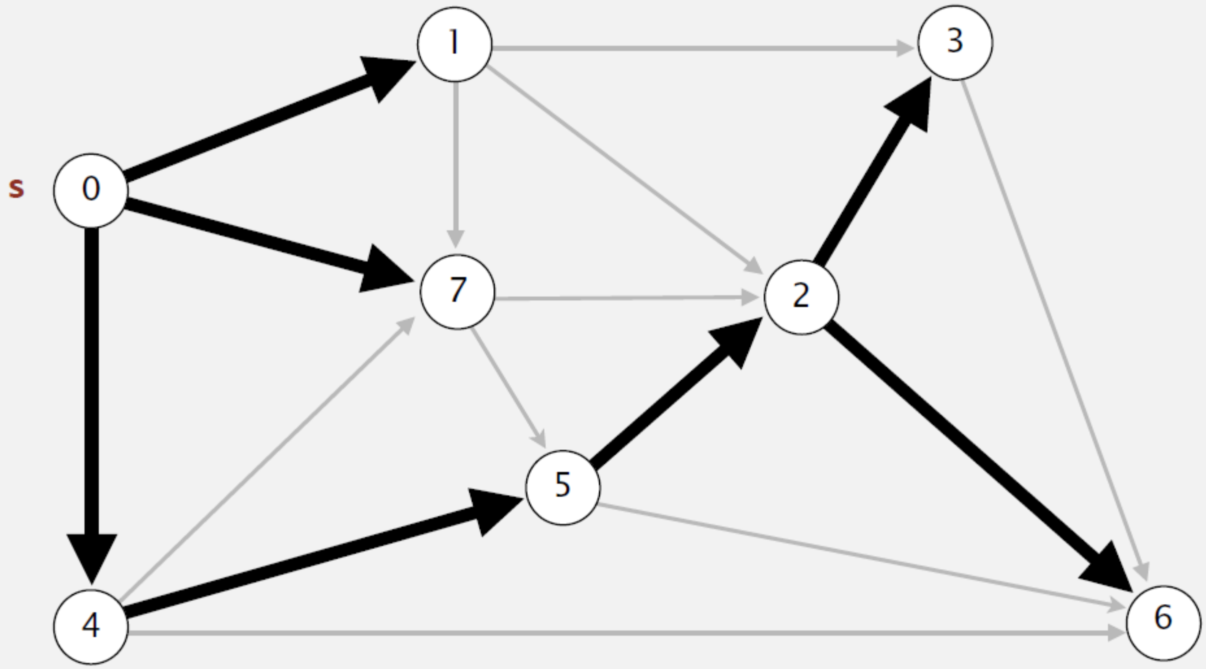


an edge-weighted digraph

0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

[Sedgewick & Wayne]

Dijkstra's algorithm

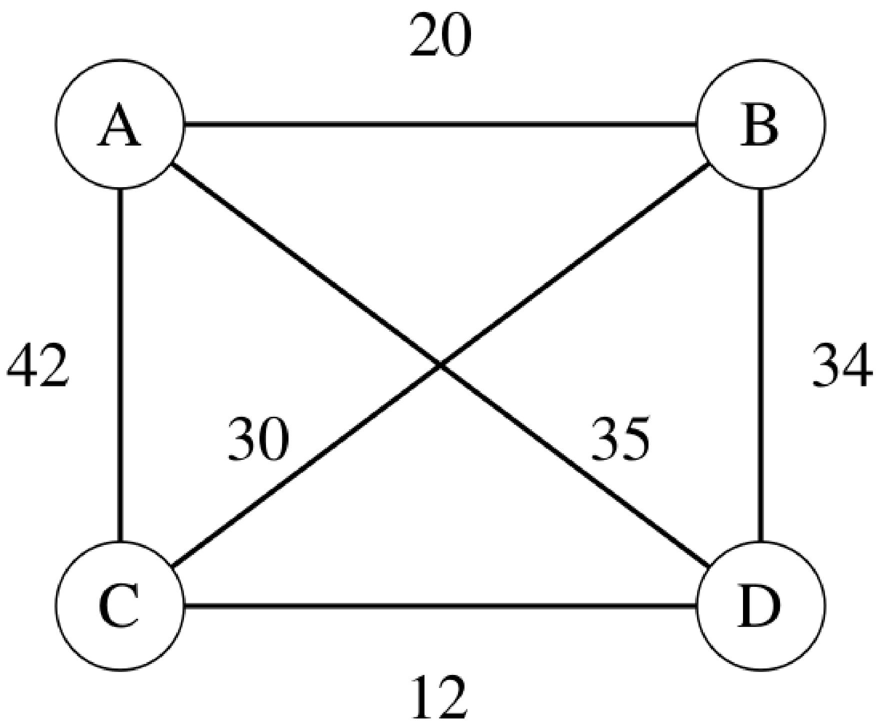


v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

[Sedgewick & Wayne]

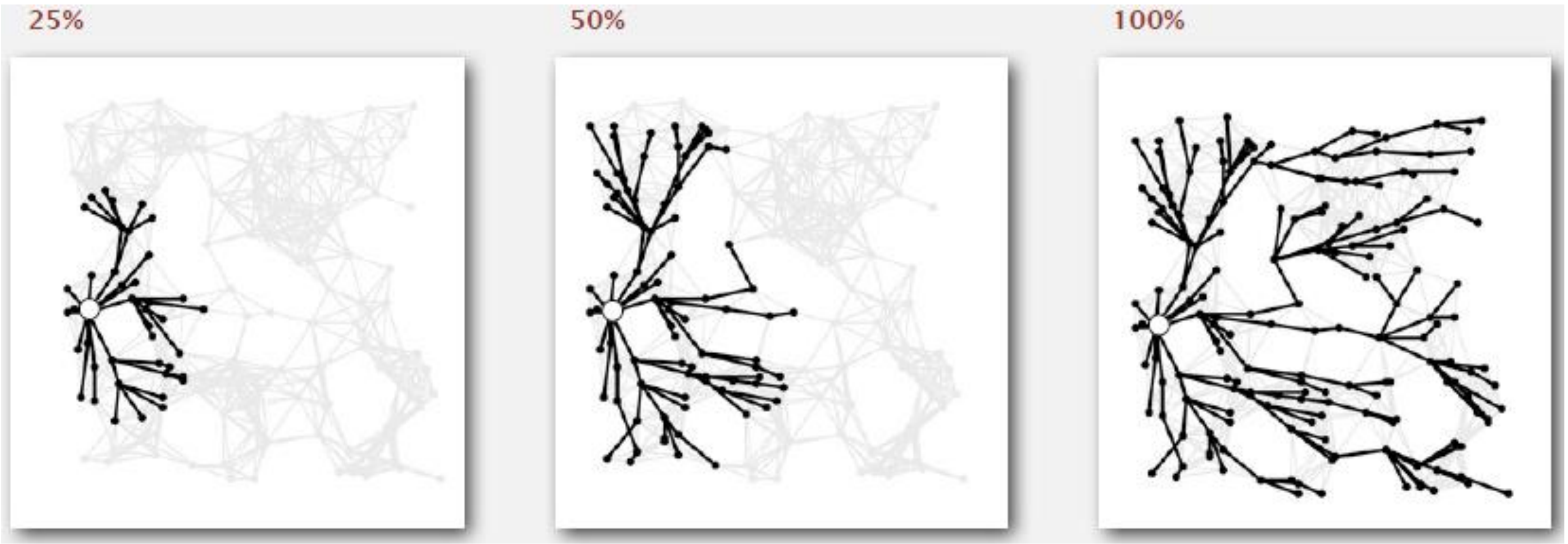
Dijkstra’s algorithm



[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

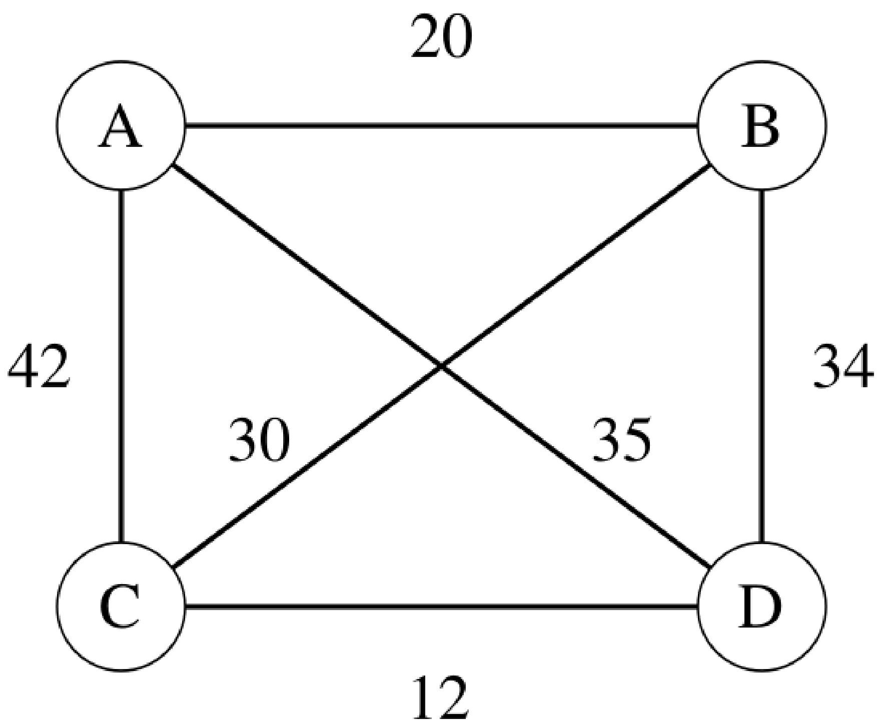
- NP-complete problem !!

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 : $r(s_a) = f(s_a) / f(s^*)$ higher than 1

- 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

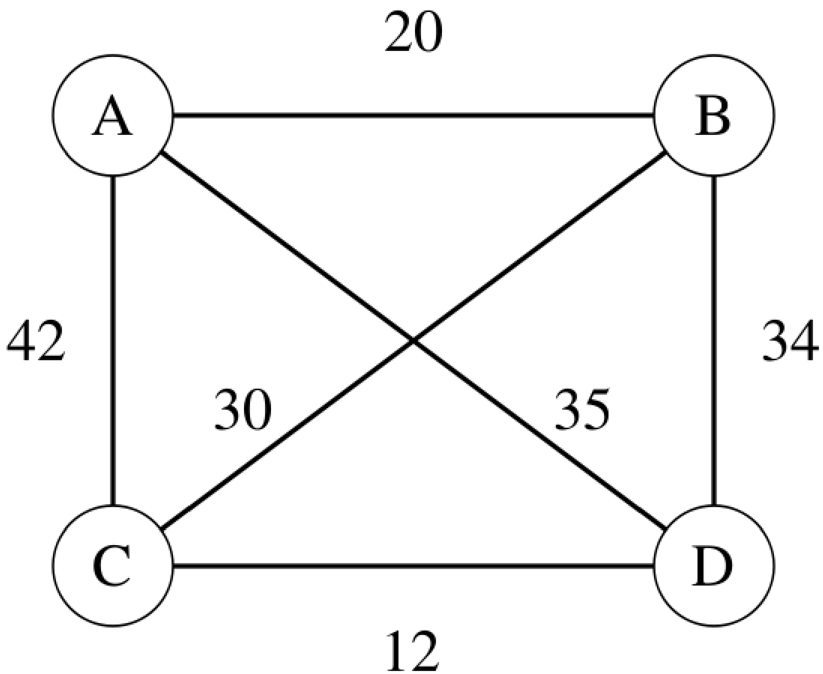
The Traveling Salesman Problem

- 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,* 3rd 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*, 3rd Ed., MIT Press, 2009
    - Chapter 16 + Chapter 23 + Chapter 24 + Chapter 35