Optimization and Software - Case Study in Transportation

CARDINAL Julien
RONDIER Lucien
JAMAL Fedoua
SOUILAH Adel

February 22, 2024

Problem presentation: introduction

- Goal: to carry out a delivery tour from a warehouse to a varying number of clients depending on the database
- fulfill the demand
- respecting the capacity limits of the three vehicles used: Truck, Bicycle, and Motorcycle
- trying to minimize the delivery cost

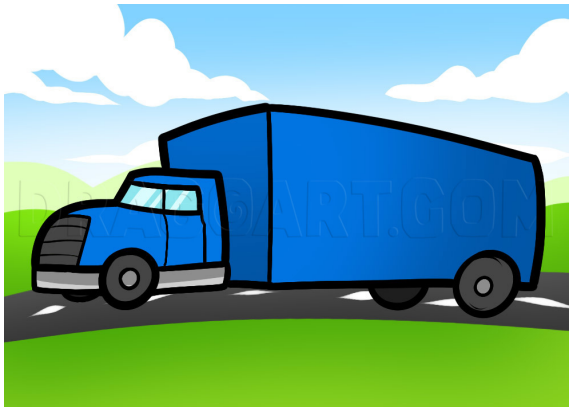


Figure: Truck

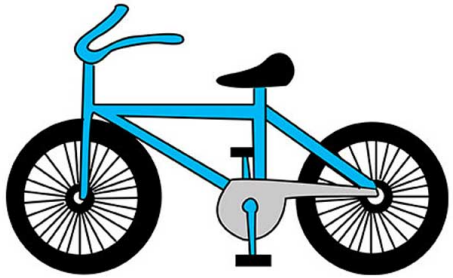


Figure: Bicycle

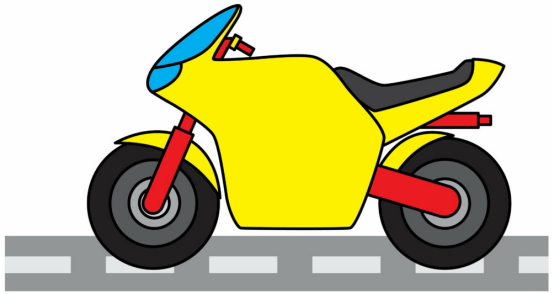


Figure: Motorcycle

Problem presentation: our data

We have decided to use the C++ language to resolve this problem To deal with the data we first transform the excel tabs into 3 different CSV files containing information about distances, characteristics of the vehicles and demand, as shown below:

parameters_name	Truck-l	Motorcycle-l	Bicycle-l	Truck-s	Motorcycle-s
Number of Vehicles	1.0	1.0	1.0	1.0	1.0
Capacity	40.0	15.0	10.0	-	-
Average Speed	40.0	50.0	10.0	-	-
Fixed Cost	180.0	100.0	60.0	350.0	250.0
Soft Time Limit	6.0	6.0	6.0	-	-
Hard Time Limit	8.0	8.0	8.0	-	-
Soft Distance Limit	300.0	300.0	150.0	-	-
Hard Distance Limit	-	-	-	75.0	
Distance Penalty Cost	0.5	0.5	0.5	-	-
Time Penalty Cost	50.0	40.0	30.0	-	-

Figure: Features of the 3 kinds of vehicles

0	0
1	10
2	7
3	5
4	1
5	9
6	4
7	6
8	10
9	1
10	5

Figure: Demand

Problem presentation: our data

Here is an extract of how we handle data loading from CSV files to a C++ class named config in our code, with parameterName being data parsed in our CSV and value a vector of floats containing lines of our tab:

```
if (parameterName == "Capacity") {
        config.Capacity = {values[0], values[1], values[2]}; //Capacity est un std::vector<float>
    } else if (parameterName == "Average Speed") {
        config.speed = {values[0], values[1], values[2]};
    } else if (parameterName == "Fixed Cost") {
        config.fixedCostVehicle = {values[0], values[1], values[2]}; //Pour les long terme
        config.fixedCostShortTermVehicle = {values[3], values[4]}; // Pour les court terme
```

Linear modeling of the problem

Here we can see the objective function, which translates our goal to reduce the cost of the delivery:

$$\min \left(\sum_{k=1}^{K_{ST}} \sum_{i=2}^{N} c_{ST,k} \cdot y_{ik} + \sum_{k=1}^{K} \left(p_{d,k} \cdot d_k + p_{t,k} \cdot t_k + \sum_{j=2}^{N} c_{V,k} \cdot x_{1jk} \right) \right)$$

With the following constraints:

- time constraints
- distance constraints
- flow constraints
- use constraints (one vehicle per demand point)
- Elimination of subtours

Solving the problem using Cplex

- For the first table, Cplex remains a good solution method because it achieves an exact solution in just 1.93252 seconds.
- This is fast, even though we will see later that we can do much better.
- However, from the second instance onwards, its solution time is almost 10 minutes. It became clear to us that we need to find a faster way to solve our problem. We initially sought to find a quicker exact method.

Bottom-up approach

As opposed to local Search methods, we first tried to **build** the best possible solution **from scratch**, this could later be used to :

- quickly solve small instances
- create a bottom-up heuristic
- initialize local search algorithms

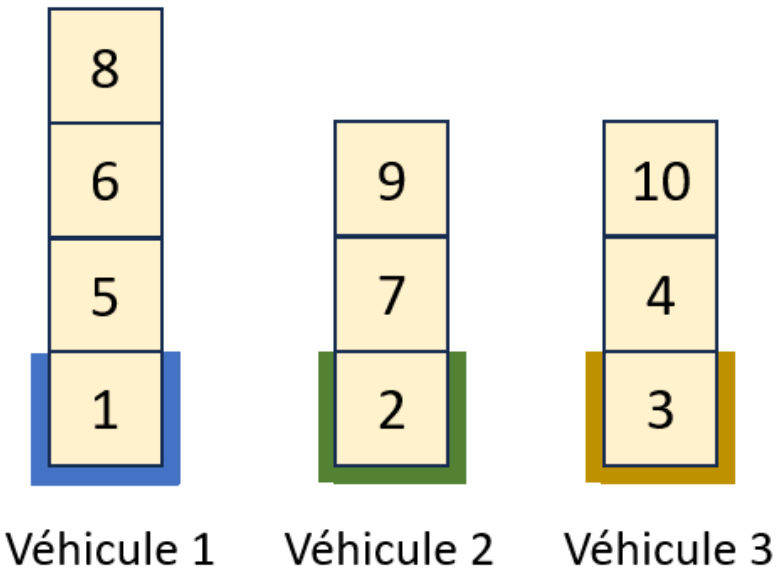
Hierarchical decomposition of the problem

The CVRP problem can be separated into two different parts :

- A K-partitioning problem (with K being the number of vehicle)
- K Travelling Salesman Problem (TSP)

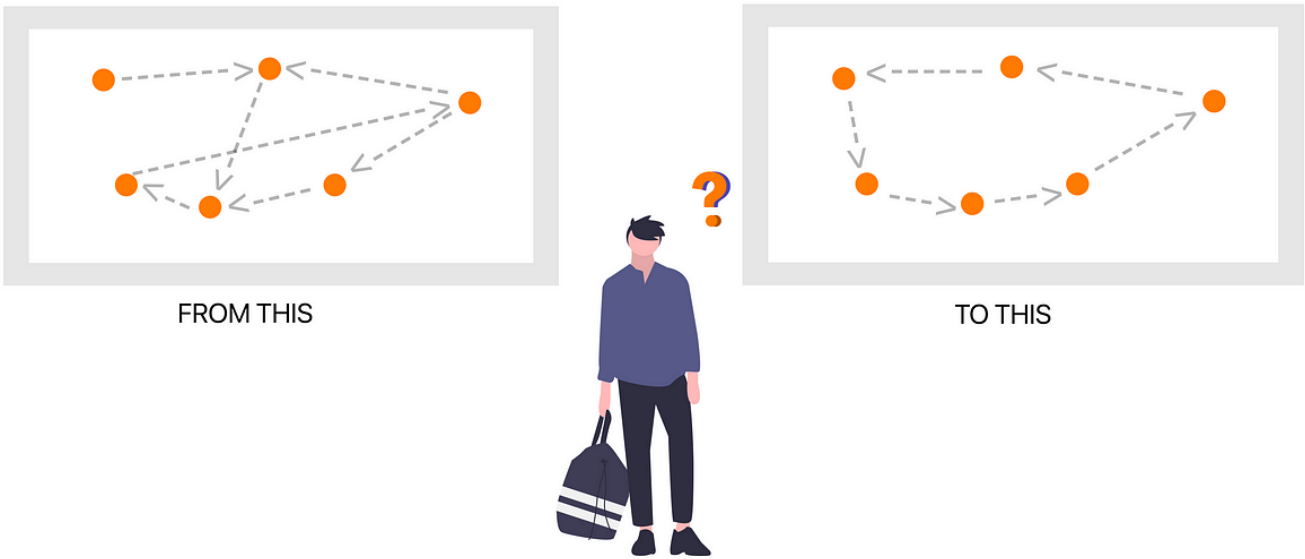
The 3-partitioning Problem

For n clients $I = \{1, 2, ..., n\}$
We aim to separate it into 3 subsets : $I_1, I_2, I_3 \subset I$ with $I_1 \sqcup I_2 \sqcup I_3 = I$
For example for n = 10 :



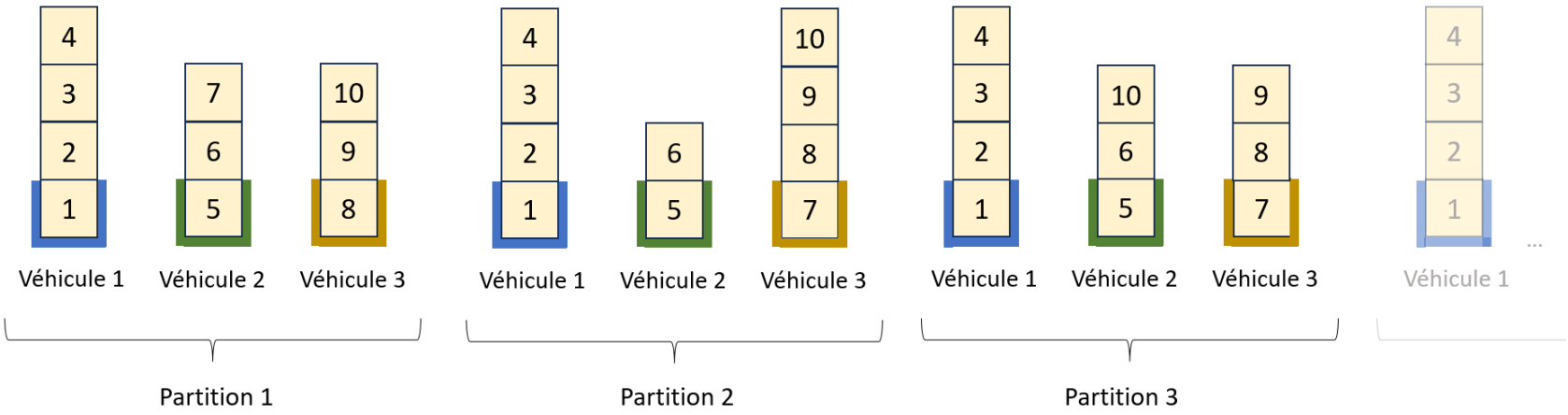
The Travelling Salesman Problem (TSP)

The Travelling salesman Problem aims at minimizing the total travelled distance while going through every cities.



Combining the two

Implementing an **exhaustive search** : We compute every K-partitioning possible solution, respecting the Capacity constraint in $O(K^n)$

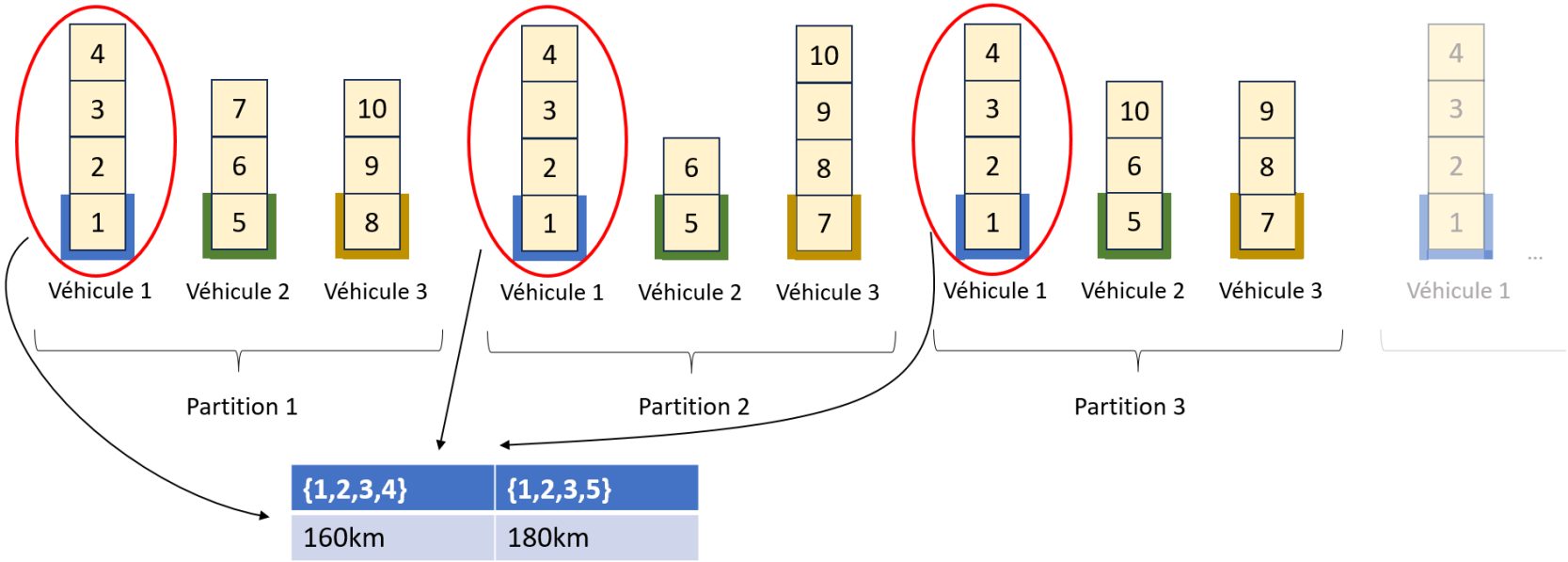


For every vehicle we compute the TSP to get the travelled distance, that way we can check the remaining constraints and compute the cost of a partition. For a total of $O(n!K^n)$

Total complexity

A problem that occurs is that we **compute too many times the same TSP**.

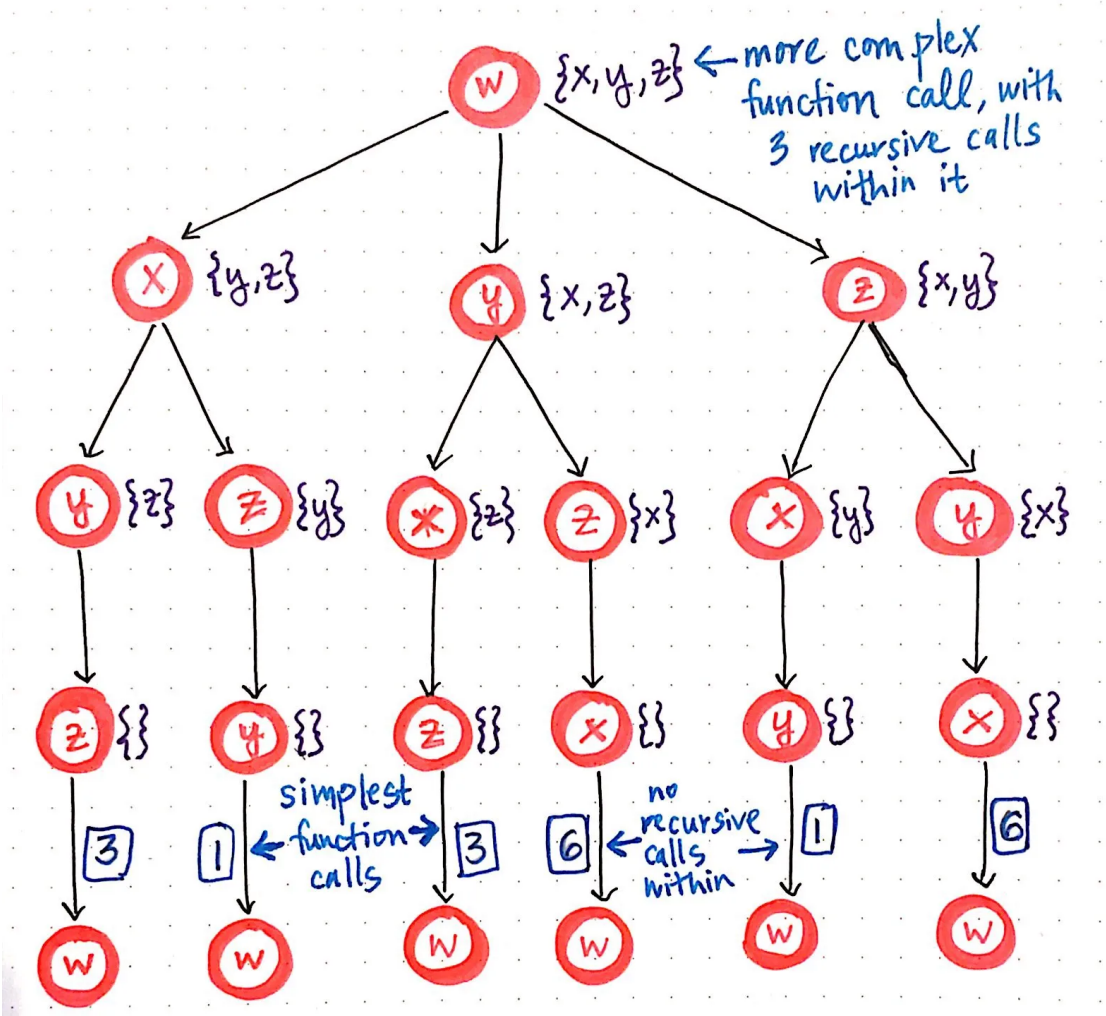
An easy improvement is to pre-compute the result of the TSP of every subset possible and **store it in a table**.



The complexity is in $O(2^n n!)$

Improvement for the TSP : The Held-Karp Method

Computing every TSP is very time consuming. An improvement would be to use the Held-Karp Method



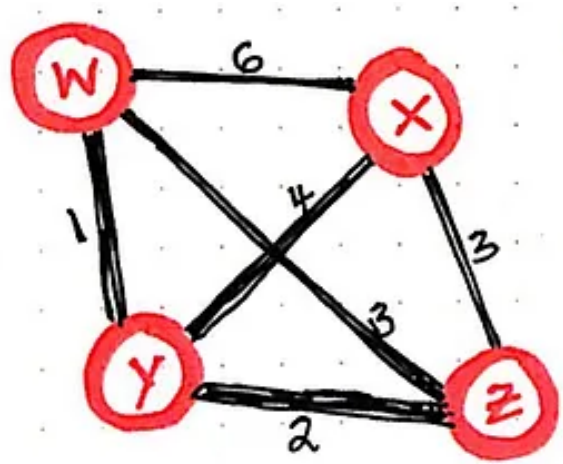


Figure: Graph for the Held-Karp dynamic Programming

Improvement for the global algorithm

The Held-Karp Method stores every minimal sub-path. With it we can easily directly **retrieve every TSP result**.

In total, filling our table is in $O(n^2 2^n)$ and after that we brute-force the K-partitioning problem, in $O(K^n)$.
With a total of $O(n^2 2^n + K^n)$

This algorithm quickly solves every instances with $n < 25$

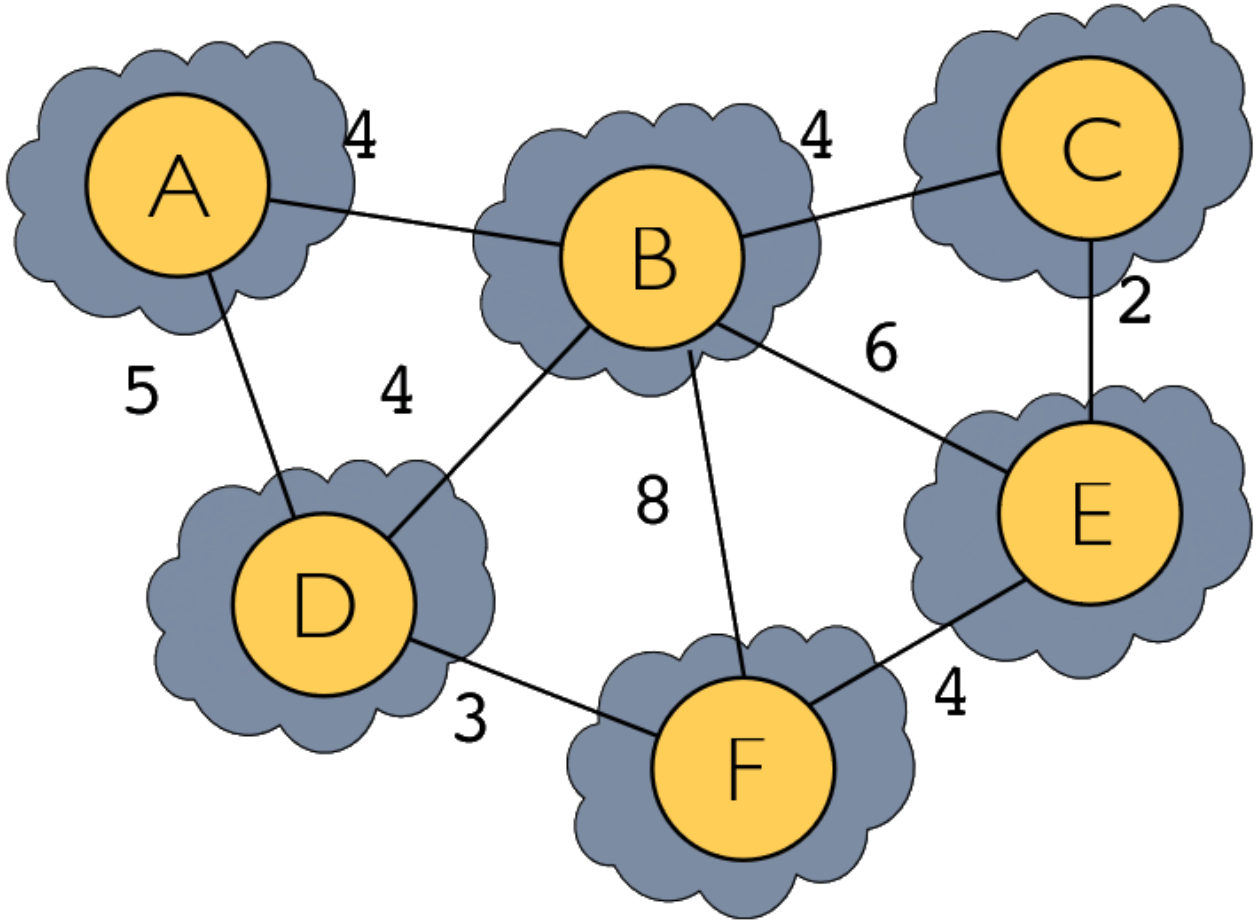
Adaptation into an heuristic

Computing the Held-Karp algorithm is not realistic for instances with $n \geq 25$ as the time and space complexity is **exponential**.

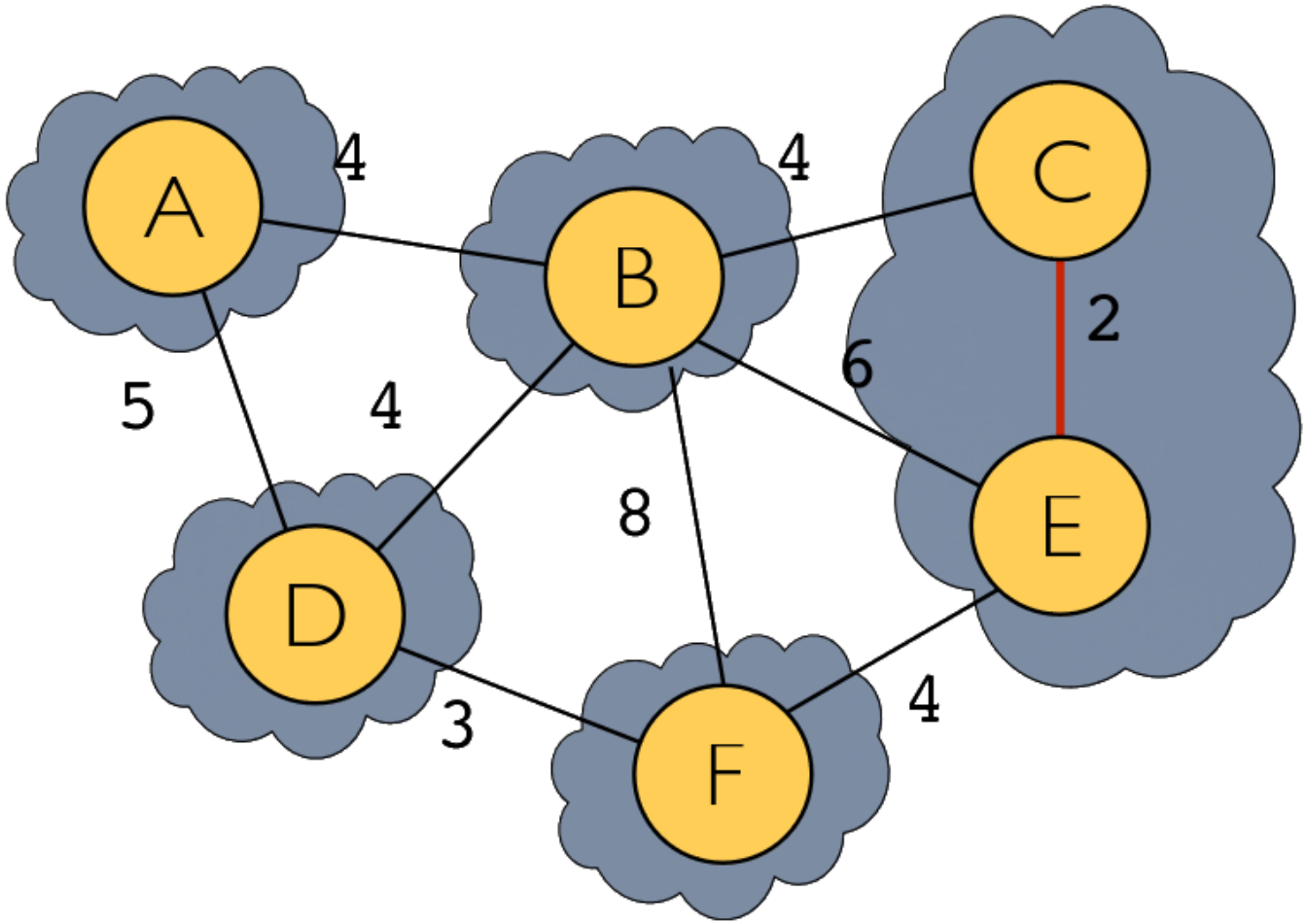
The idea is to **compute an upper-bound** of the TSP solution in **polynomial time**.

We are going to use the approach of **Minimal Spanning Trees**.

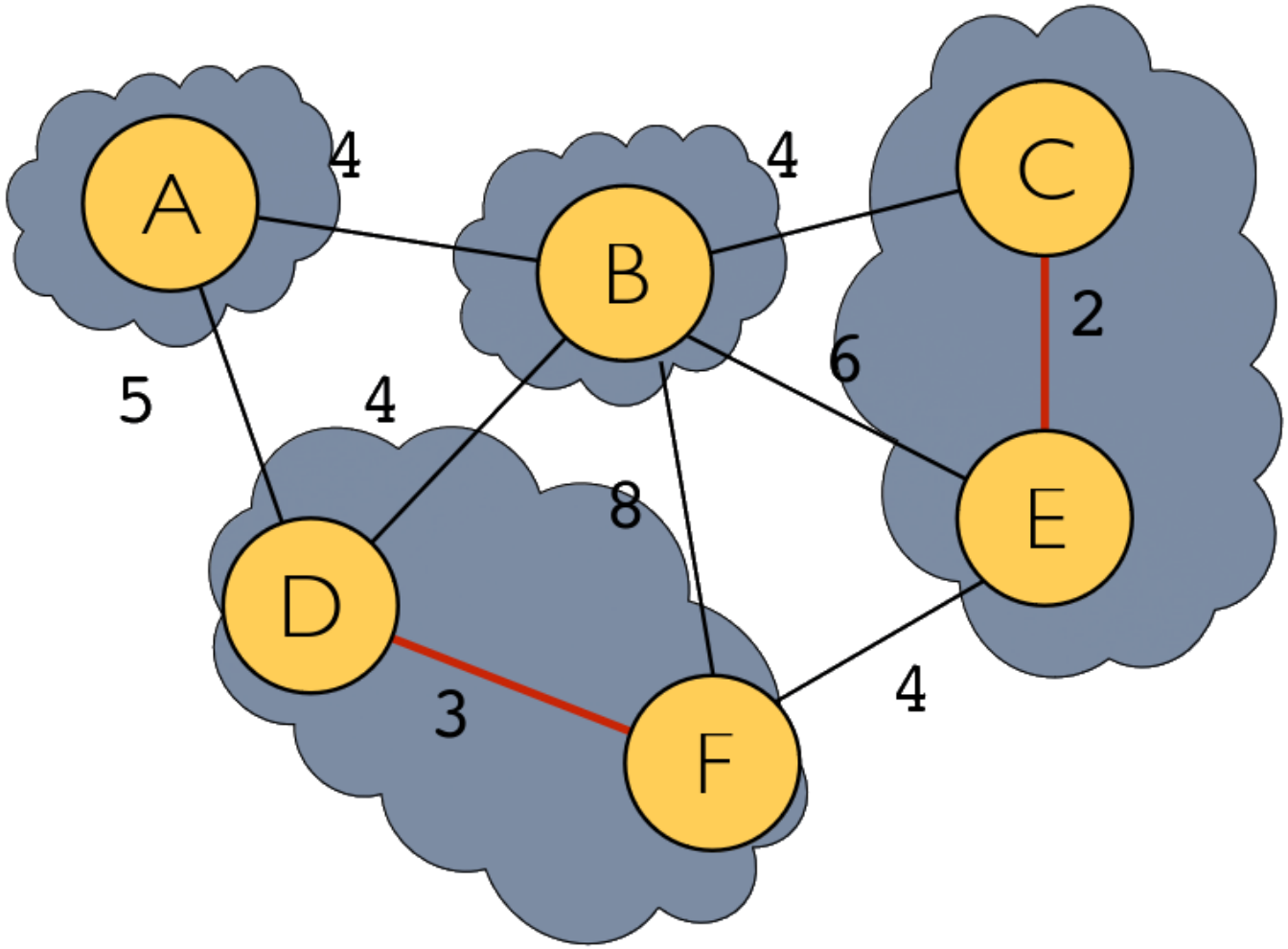
Minimal Spanning Tree : Kruskal algorithm



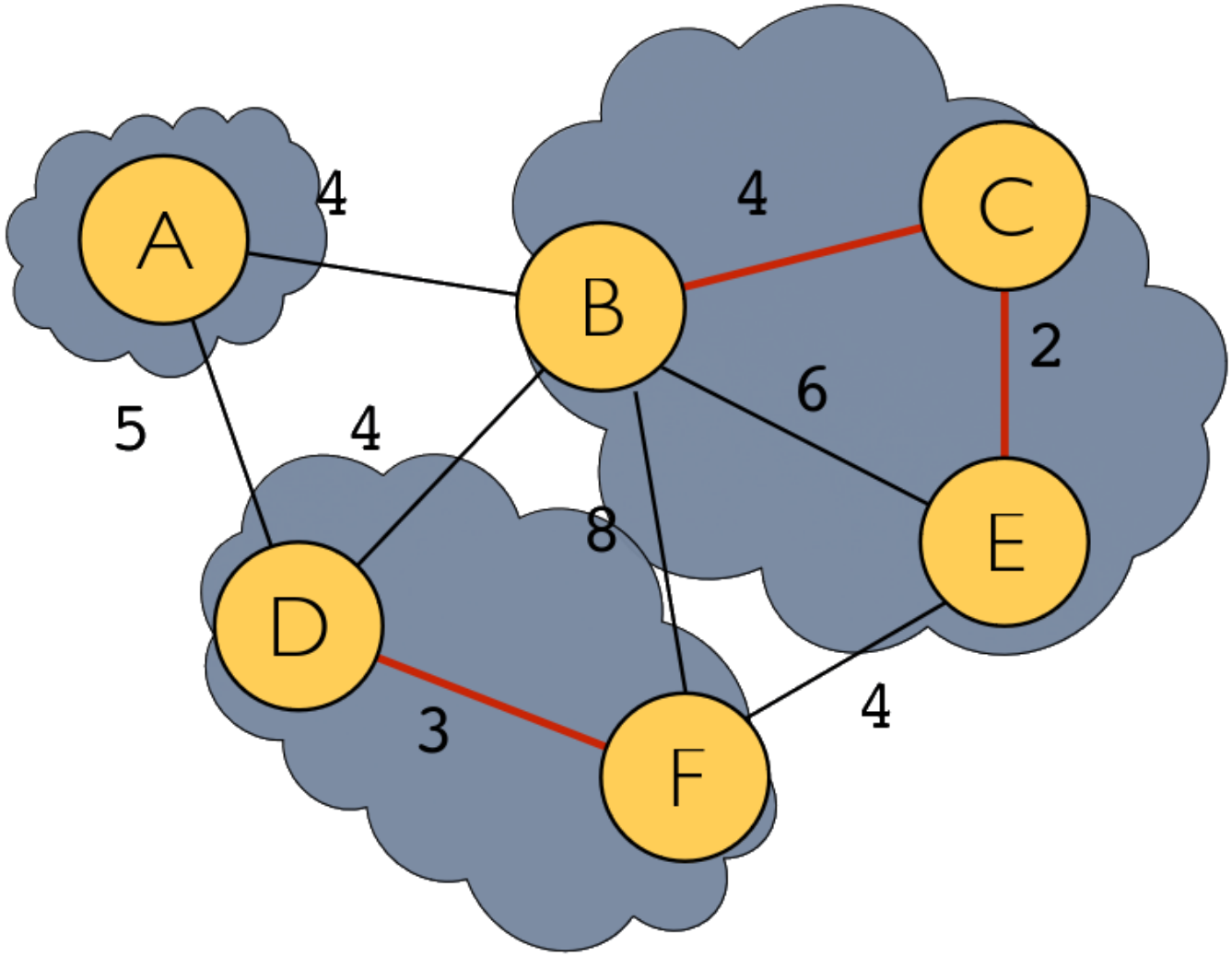
Minimal Spanning Tree : Kruskal algorithm



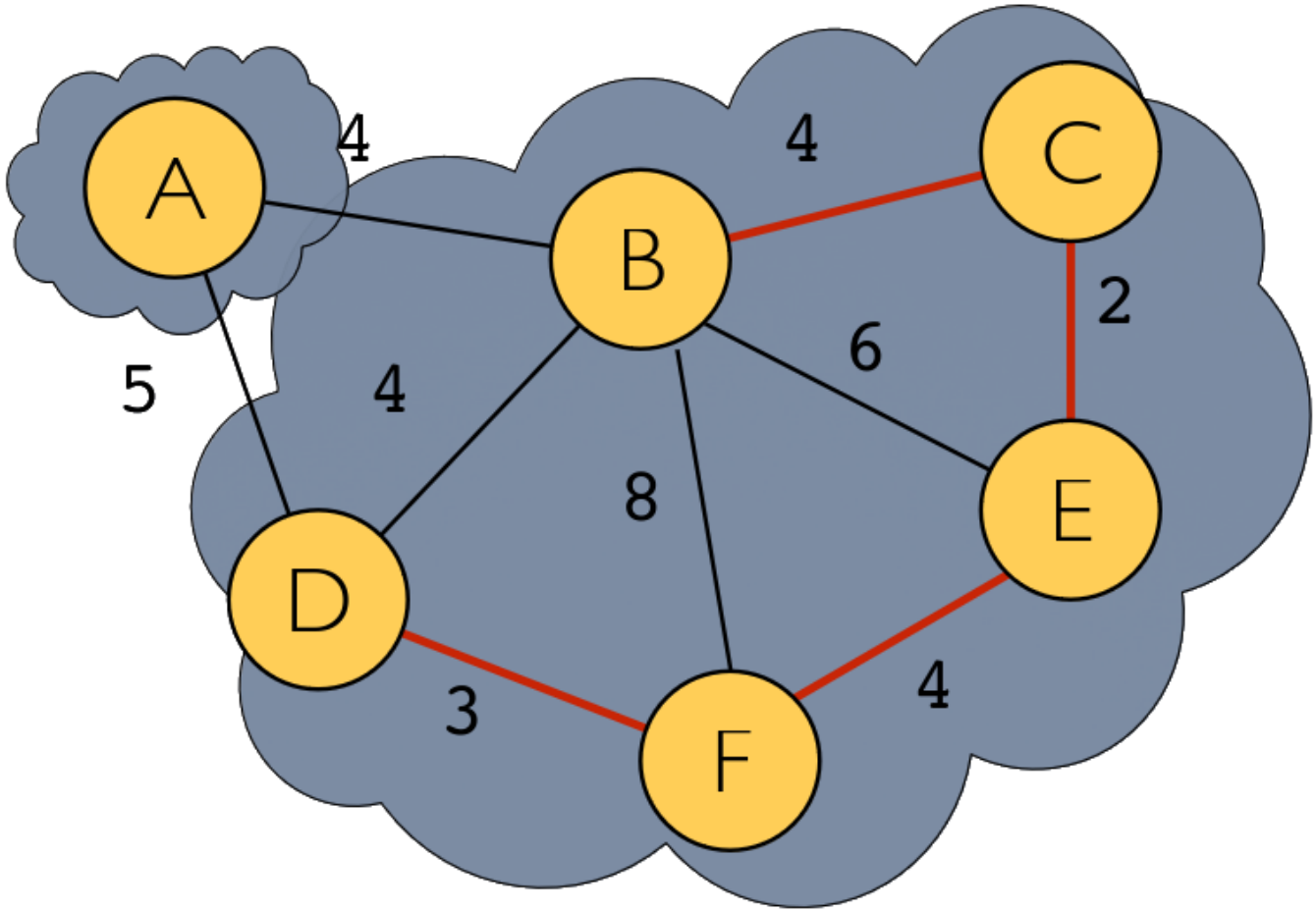
Minimal Spanning Tree : Kruskal algorithm



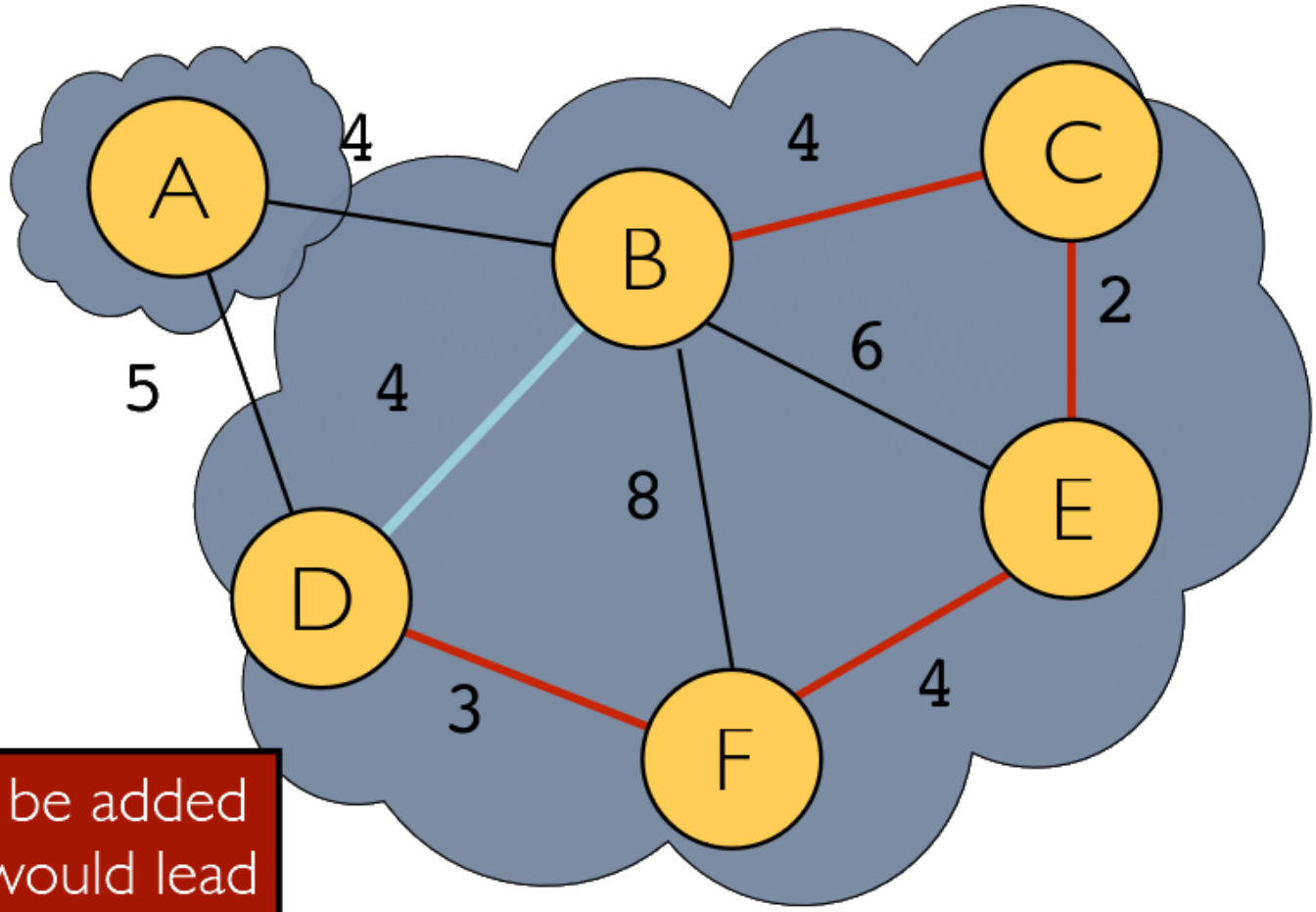
Minimal Spanning Tree : Kruskal algorithm



Minimal Spanning Tree : Kruskal algorithm

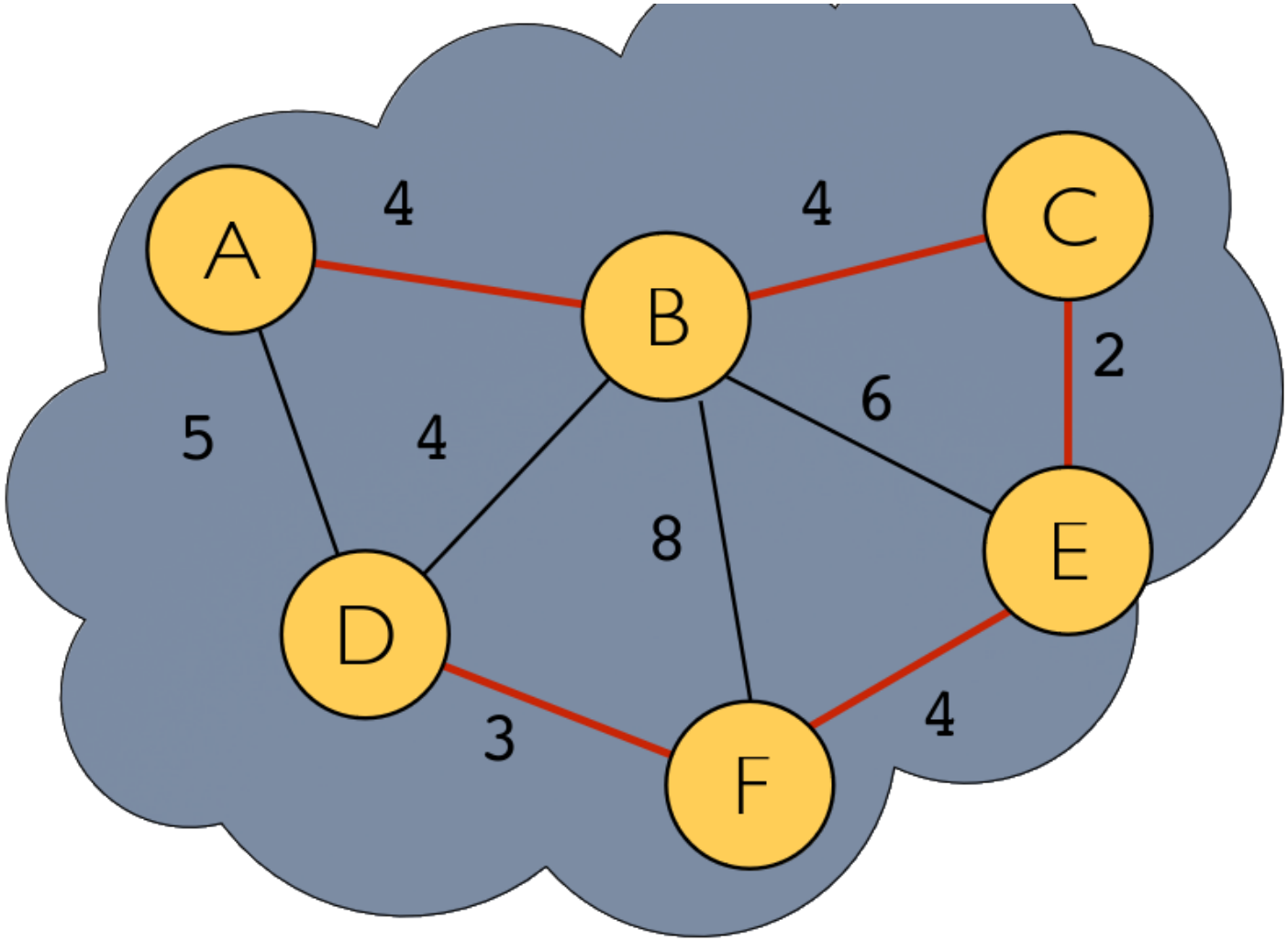


Minimal Spanning Tree : Kruskal algorithm



BD cannot be added because it would lead to a cycle

Upper bound for the TSP



Heuristics to solve the K-partitioning Problem

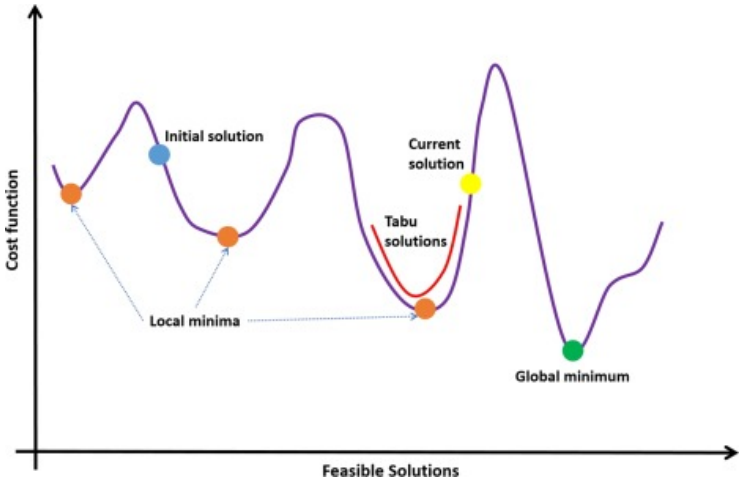
Now computing a good solution for the TSP is quick in $O(n^2 log(n))$.
Brute-Forcing the K-partitioning Problem is also not feasible for big instances. $n \geq 25$

We can try many **graph searching algorithms** to find the best approximation :
We can think of using a **Branch and Bound** algorithm or a **MCTS** algorithm.

Local search

Tabu Search

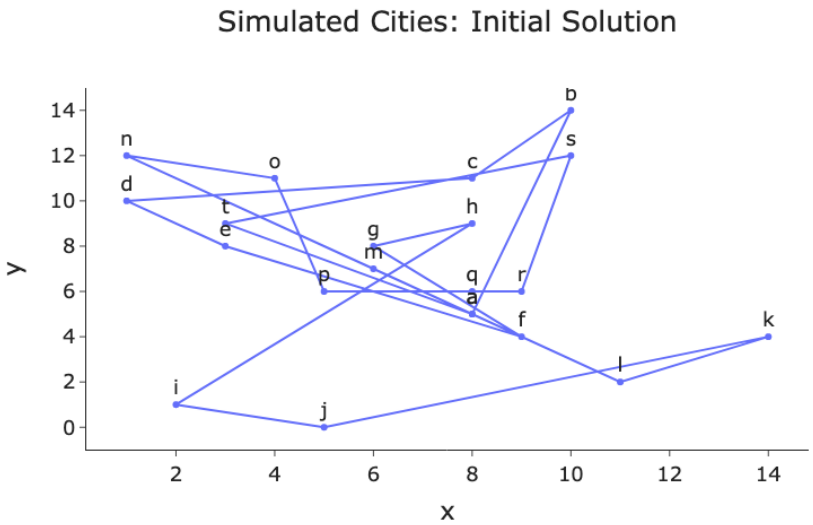
- Step 1: Initialization
- Step 2: Exploration
- Step 3: Evaluation and Selection
- Step 4: Tabu List

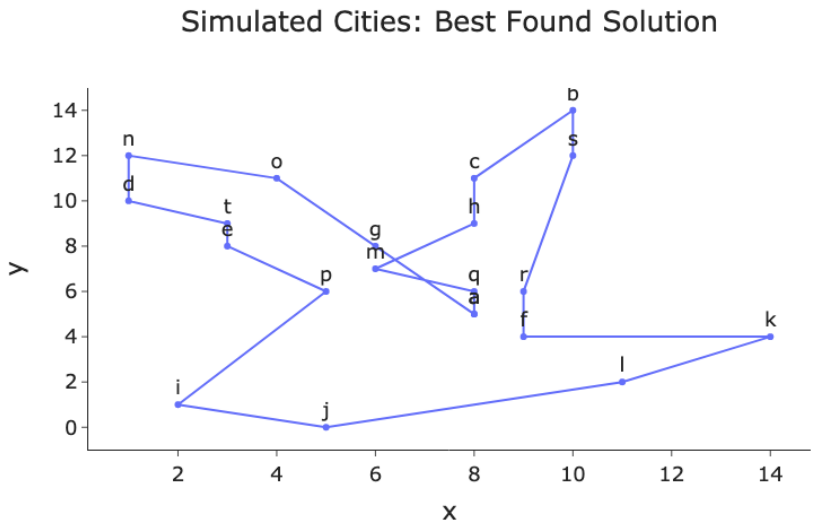


figureOvercoming local minima

The implementation

1. Start with a randomly generated initial solution, exploring neighborhoods through swapping and reassignment.
2. Use a tabu list to select moves, it avoids cycles.
3. Evaluate solutions based on the smallest cost to always improve.





Possible improvements

1. Aspiration criteria
2. Dynamically change the tabu list size and tenure
3. Expand the neighborhood definition

Conclusion et analyse des résultats

The following table shows the performance of our methods across 5 instances, with each cell containing the score achieved, the computation time required, and the error percentage relative to the exact solution:

Methods	1 (n=10)	2 (n=15)	15 (n=20)	16 (n=50)	17 (n=100)
Tabou	700.4 / 0.04s / 0%	1109.36 / 0.18s / 25.4%	1071.62 / 0.45s / 37.1%	540 / 2.8s / 80%	1080 / 12.5s / 58.8%
Heur.	700.4 / 0.08s / 0%	884.81 / 0.01s / 0%	781.36 / 23.56s / 0%	300 / 61.01s / 0%	780 / 61.02s / 14.7%
Exact	700.4 / 0.002s / 0%	884.81 / 0.065s / 0%	781.36 / 5.62s / 0%	-	-
CPLEX	700.4 / 1.39s / 0%	884.81 / 284.77s / 0%	-	-	-

Table: Comparative results of methods on datasets