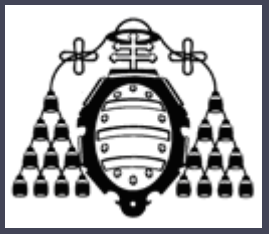


Algorithmics

Greedy algorithms

Vicente García Díaz – garciavicente@uniovi.es

University of Oviedo, 2016

Table of contents

Greedy algorithms

1. Basic concepts
2. Examples of use
    1. The problem of the change
    2. The problem of the change (with an optimal solution)
    3. The knapsack problem The knapsack problem (0/1)
    4. Maximize the number of files on a disk
    5. Minimize the free space on a disk with files
    6. The diligent plumber
    7. Some diligent plumbers
    8. The truck driver in a hurry
    9. Prim
    10. The horse jumping problem
    11. The problem of assigning tasks to agents

Basic concepts

Greedy algorithms

- The **greedy algorithms** are used in optimization problems
- Like other techniques, it is based on building the solution **step by step**
- They perform the search for the optimal solution to the problem looking for the **locally optimal solution at each step**
- To search the locally optimal solution algorithms usually employ **heuristic rules,** based on some knowledge about the problem

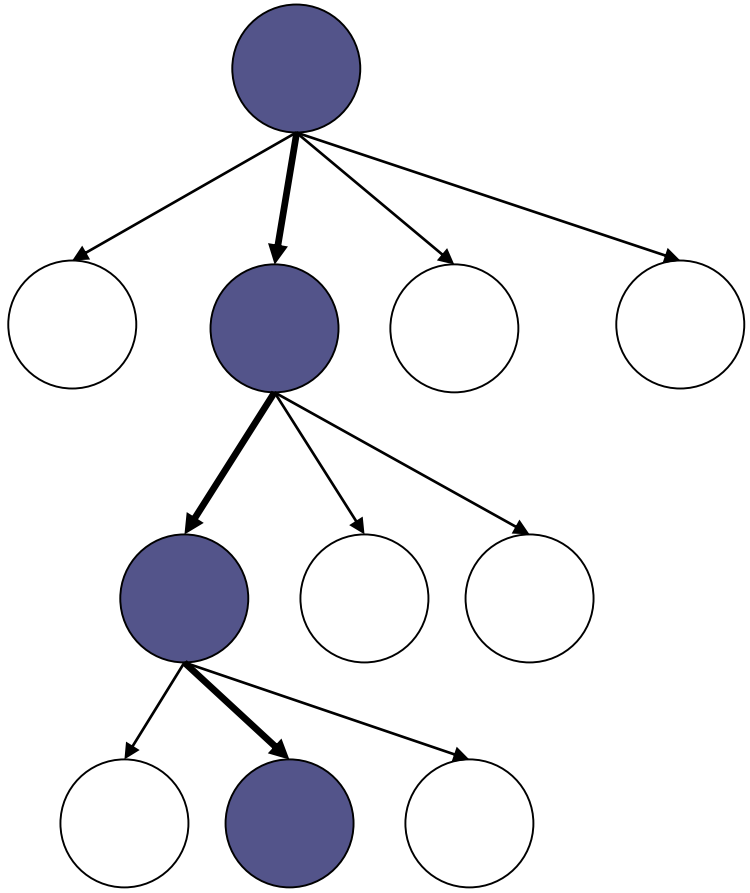
Decision making in greedy algorithms

- These algorithms make decisions based on the information they have available locally, hoping that these decisions will lead to the desired solution

- A decision is **never** reconsidered
  - There is no need to use monitoring procedures to undo the decisions already made

Tree of states

- It involves developing a node in the tree of states of the problem (virtual or fictitious tree), until finding a solution
- Nodes that are developed are indicated by the heuristic
- It is necessary that all developed states are "achievable", that is, from them it **should be possible** to reach a solution to the problem



Key Features

- The main feature of these algorithms:
  - They are quick and easy to implement
  - They depend entirely on the quality of the heuristic
    - A low-quality heuristic could not provide optimal solutions
- In the worst case it might not provide solutions, even though the problem could have solutions
- They are suitable for those cases where:
  - Because of the quality of the heuristic it finds the optimal solution
  - There is no time to apply other techniques
- Many problems cannot be solved with this approach ☹

Drawbacks (I)

- Design an algorithm to pay a certain amount of money, using the fewest possible coins

- Example:
  - We have to pay €2.89
  - Solution: 1 coin of €2, 1 coin of 50 cents, 1 coin of 20 cents, 1 coin of 10 cents, 1 coin of 5 cents, 2 coins of 2 cents
    - → Optimal solution ☺

- Heuristic: Take the coin of the biggest value without exceeding what we have left to return

Drawbacks (II)

- Depending on the monetary system and the availability of coins, the heuristic does not always provide the optimal solution

- We demonstrate it with a **counterexample**:
  - We need to pay €0.60
  - Available coins: €0.50, €0.20, €0.02
  - Solution (heuristic): €0.50 + 5· €0.02→ 6 coins
  - Optimal solution: 3· €0.20 → 3 coins

- This problem could be solved with backtracking, developing all possible solutions

Drawbacks (III)

- With the previous heuristic, if we want to optain always an optimal solution we need to meet one condition:
  - We have to assume:
    - The monetary system is composed of coins with values:
      - $1, p, p^2, p^3, p^4, \ldots, p^n$ where $p > 1$ and $n > 0$
    - We have an unlimited number of coins of each value
      - There is a mathematical demonstration of this

Differences with Backtracking (I)

```
public void assay(State currentState){
  boolean hasSolution= false;

  while (                                !hasSolution){
      State selectedState= currentState. findBestState();
      if (selectedState. meetsHeuristic()
            if (!selectedState.isSolution()){
                  currentState= selectedState;


      }
            else
                  hasSolution= true;
      }
  }

}
```

Differences with Backtracking (II)

```
public void assay(State currentState){
  boolean hasSolution= false;

  do { //loop to check all the sister states
      State selectedState= currentState.findBestState();
      if (selectedState.meetsHeuristic()
            if (! selectedState.isSolution()){
                  currentState= selectedState;


      }
            else
                  hasSolution = true;
      }
  }
  while !hasSolution);

}
```

Pseudocode. General outline

```
public void greedy(State currentState){
  while (!currentState.isSolution()){
      /* Choose the first component
      *  that satisfies a given heuristic*/
      State selectedState=
            currentState.findBestState();

      if (selectedState.meetsHeuristic())
            currentState= selectedState;
      else
            currentState= currentState.nextState();
  }
}
```

Pseudocode

```
public void calculateCoins(double amount){
  int numCoins= 0; //index of the coins (the lowest index
  corresponds to the higher value of the coin)

  while (amount > 0 && numCoins < coins.length){
      //state is valid
      if (amount - coins[numCoins] >= 0){
            //new coin in the solution set
            solution[numCoins]++;
            amount -= coins[numCoins];
      }
      else
            //check the coin of lower value
            numCoins ++;
  }
}
```

Examples of use

Goal

- **n** objects and a backpack to transport them
- Each object i= 1,2, ...n has a weight of w_i and a value of v_i
- The backpack can carry a total weight not exceeding W
- **The idea is to maximize the value of objects, while respecting the weight limitation**
- Objects **can be fragmented**
  - We can carry a fraction x_i of object i, with $0<=x_i<=1$

Data for a specific problem

- Number of objects: `n=5`
- Weight limit of the backpack: `W=100`

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Strategy

- Candidates: the `n` objects
- Status of the problem: solution vector
  $X=(x_1,x_2,x_3..x_n)$
- Function to check if a solution is feasible:
  - `meetsHeuristic()` $\sum_{i=1}^{n} w_i \leq W$
- Function for heuristic selection:
  - `findBestState()`

Heuristic (I) `findBestState()`

- Three possibilities:
  - Select the most valuable object
  - Select the less heavy object
  - Select the object which value per weight unit is as large as possible

Objects	1	2	3	4	5	Value (V_i)
Descendant v_i	0	0	1	0,5	1	146
Ascendant w_i	1	1	1	1	0	156

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Heuristic (II) `findBestState()`

- The last heuristic provides the optimal solution
- These data were the counterexample to show that the first two heuristics do not provide the optimal solution

Objects	1	2	3	4	5	Valu
Value / weight	2	1,5	2,2	1	1,2	e
Descendant *v/w*	1	1	1	0	0,8	164

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Pseudocode

```
public float[] findObjects(float maxWeight){
    int i = 0;
    float currentWeight = 0; //actual weight of the backpack
    float x[]; //fraction of the transported object 0-1

    do { //builds and initializes arrays
        i = findBestState(); //heuristic for selection
        if (currentWeight + weights[i] <= maxWeight) {
             x[i] = 1; //you take the whole object
             currentWeight += weights[i];
        }
        else {
             x[i]= (maxWeight - currentWeight)/weights[i];
             currentWeight = maxWeight;
        }
    } while (currentWeight < maxWeight);
    return x;
 }
```

Without breaking objects

- If we cannot break objects, it is possible that we don't get an optimal solution with a greedy algorithm
  - Number of objects: `n=3`
  - Weight limit of the backpack: `W=6`

Object	1	2	3
w_i	5	3	3
v_i	11	6	6
Heuristic	2,2	2	2

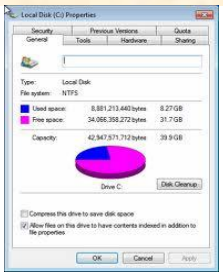
  - The previous greedy algorithm gives us this solution:
    - It takes the first element with a **value of 11**, but nothing more
  - The optimal solution would be to take the two last objects with a **value of 12** even when they have a worse relation `value/weight`

Maximize the number of files on a disc

- We have a hard disc
- We want to introduce in it as many files as possible

- For example
  - With a capacity of 880 MB
  - And the following size (in MB) for some files:
    - 100, 350, 450, 370, 5000, 500, 700, 800, 50

- What heuristic could we use?
  - Very simple, first the **smallest** files while the disc has enough space to copy them



Minimize the free space on a disk with files

- We have a hard disc
- We want to minimize its free space by introducing in it as much amount of bytes as possible

- For example
  - With a capacity of 1200 MB
  - And the following size (in MB) for some files:
    - 100, 350, 450, 370, 5000, 500, 700, 800, 60

- What heuristic could we use?
  - First the **biggest** files while the disc has enough space to copy them

`calculate()?`

The diligent plumber

- A plumber needs to make n urgent repairs
- He knows in advance how long it will take each of them
  - For the i*th* task he will take t_i minutes
- The plumber wants to minimize the average waiting time for customers to make them as happy as possible and thereby move up in his company
  - If w_i is the time the i*th* customer waits, then the plumber needs to minimize the expression $W(n) = \sum_{i=1}^{n} w_i$
- What heuristic could we use?

Heuristic

- The plumber will always take the same total time to finish all the tasks regardless of the order in which they are chosen
  - $T = t_1 + t_2 + t_3 + \ldots + t_n$
- …however, waiting times for customer will be different depending on the order of the tasks
  - $w_1 = t_1$
  - $w_2 = t_{1+} t_2$
  - …
  - $w_n = t_1 + t_2 + \ldots + t_n$
- That is, we have to find the permutation of tasks for which W(n) is minimized
- What would be the best order to address the tasks?
  - He has to meet first tasks that will last shorter time!

`getTotalTimeOfWait()?`

Some diligent plumbers

- We have the same goal as in the previous case
- …but the company has decided to hire more plumbers due to the great success of its services
- They have a number `F` of plumbers for doing `n` tasks
- We should modify the previous algorithm so that the order of execution of tasks using all the plumbers is again optimal
- What would be the best order to address the tasks?
  1. We order the requests by increasing time of the work to be done (*the same as in the previous case*)
  2. We assign the requests in that order, always to the less busy plumber

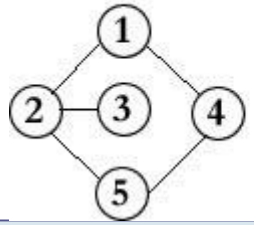
`orderInWhichTasksAreHandledBESTWAY() & assignTasksToPlumbersBESTWAY()?`

The truck driver in a hurry

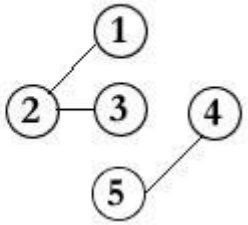
- A truck driver goes from Bilbao to Malaga
- The truck, which initially has a full fuel tank, can travel `n` kilometers without stopping
- The truck driver has a road map that shows the distances between fuel stations
- As he is in a hurry, the truck driver wish to stop to refuel as few times as possible
- What heuristic could we use?
  - *He must try to advance the largest possible number of kilometers without refueling. That is, try to go from every fuel station to the farthest possible one*

Algorithm of Prim

Connected graph



Disconnected graph



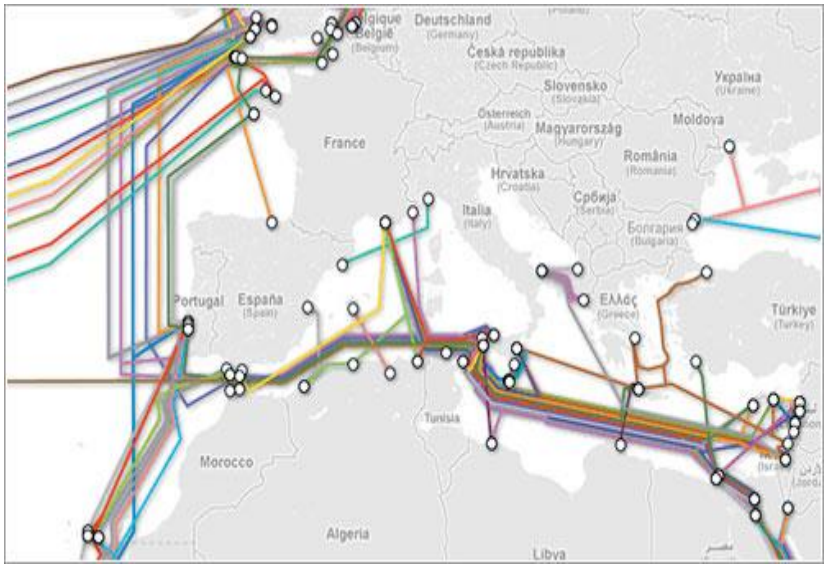
- Goal:

Find the minimum spanning tree in a connected and undirected graph

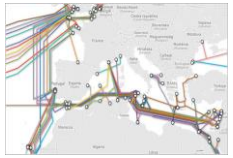
A graph is connected if every pair of vertices is connected by a path

- Example of use:

Trace the minimum possible amount of cable to meet the needs of a new urbanization



Pseudocode



method *Prim*(graph, source)

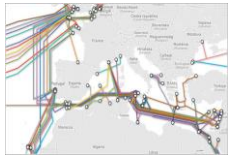
1- **mark a node** in the graph, which will be the starting node

2- **select the edge of lowest value** that leaves the marked node

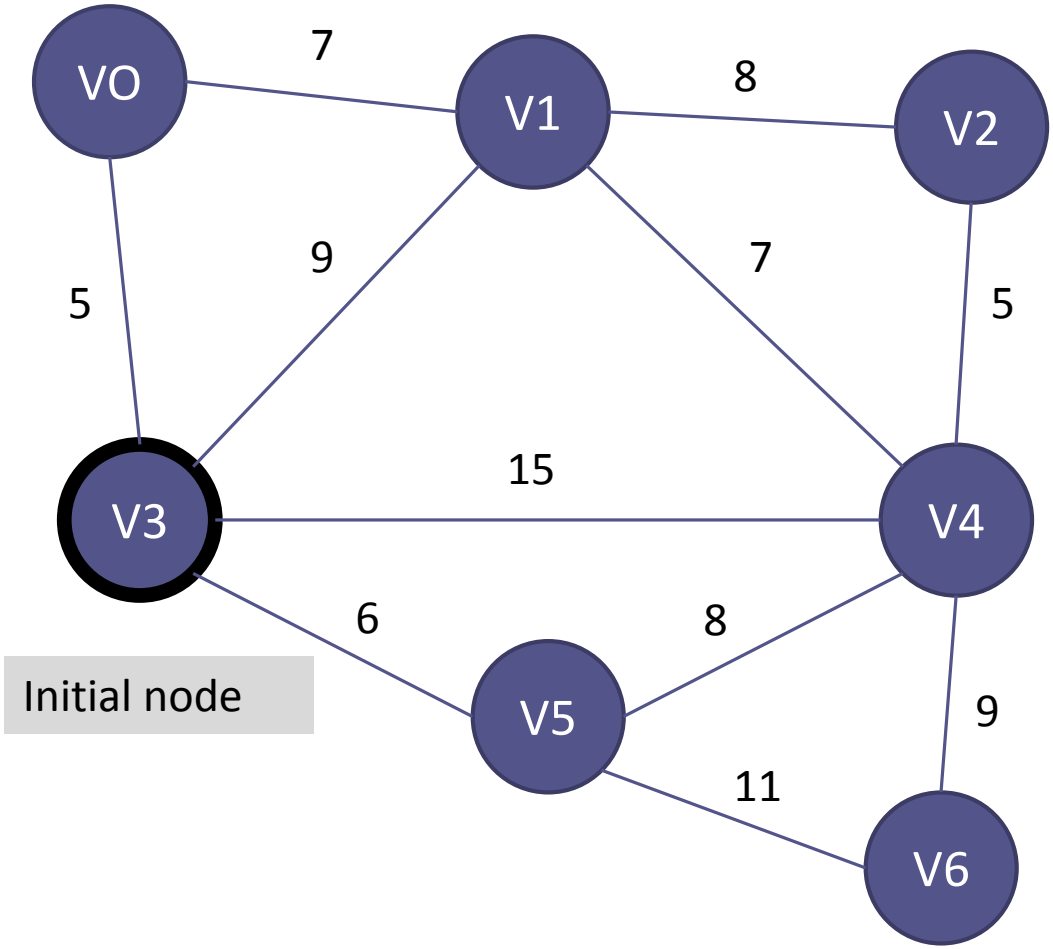
3- **mark the node that is reached by the selected edge** in the previous step

4- **repeat steps 2 and 3** getting edges between marked nodes and others which are not

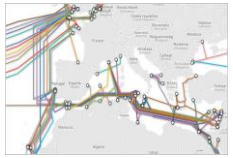
5- The algorithm **ends** when all nodes are marked



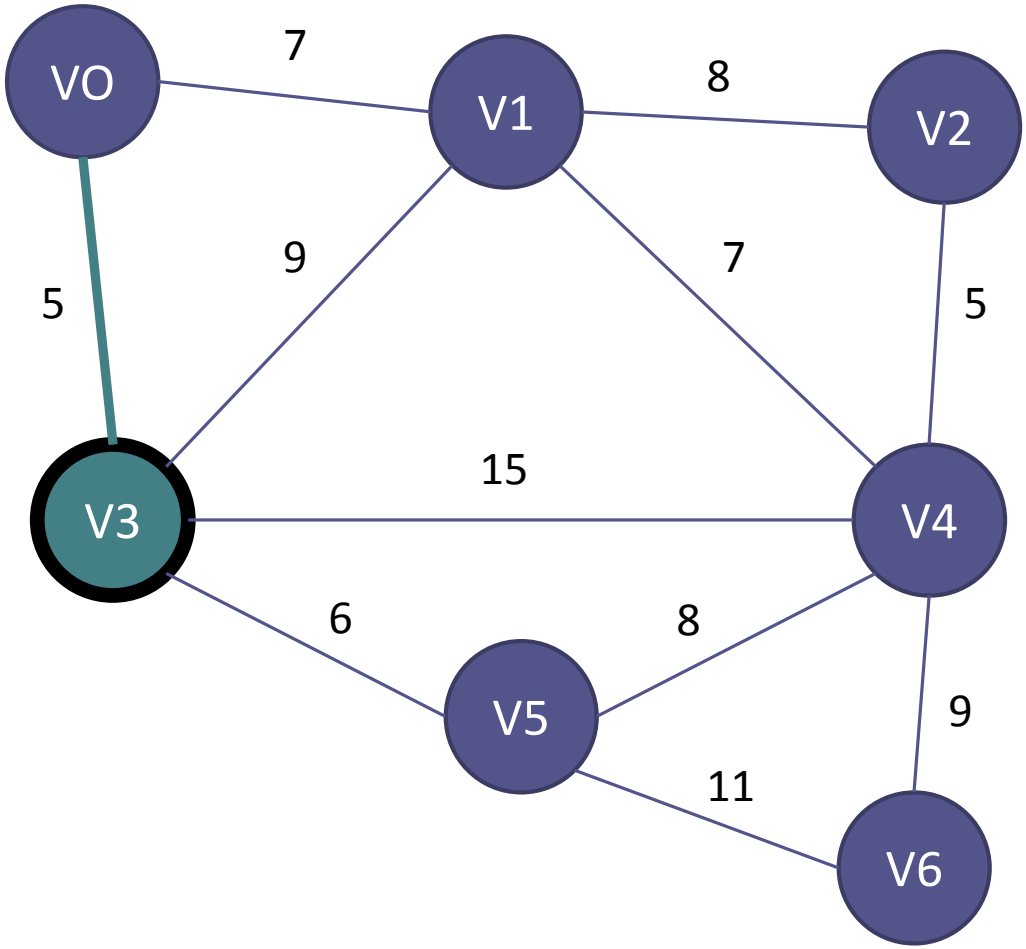
Process



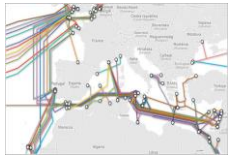
Initial node



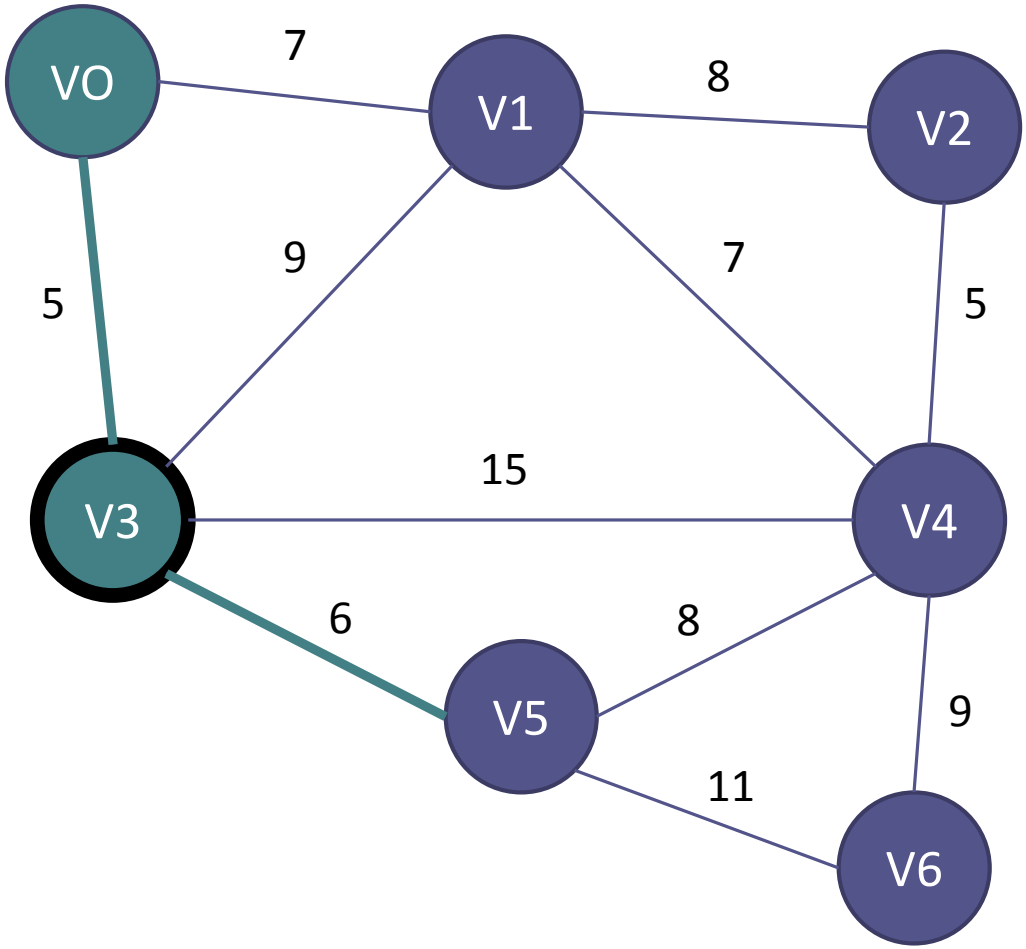
Process



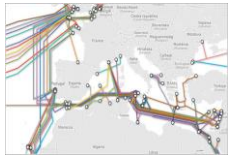
Tree	Options	Graph
{V3}	**{V3, V0} = 5** {V3, V1} = 9 {V3, V4} = 15 {V3, V5} = 6	{V0, V1, V2, V4, V5, V6}



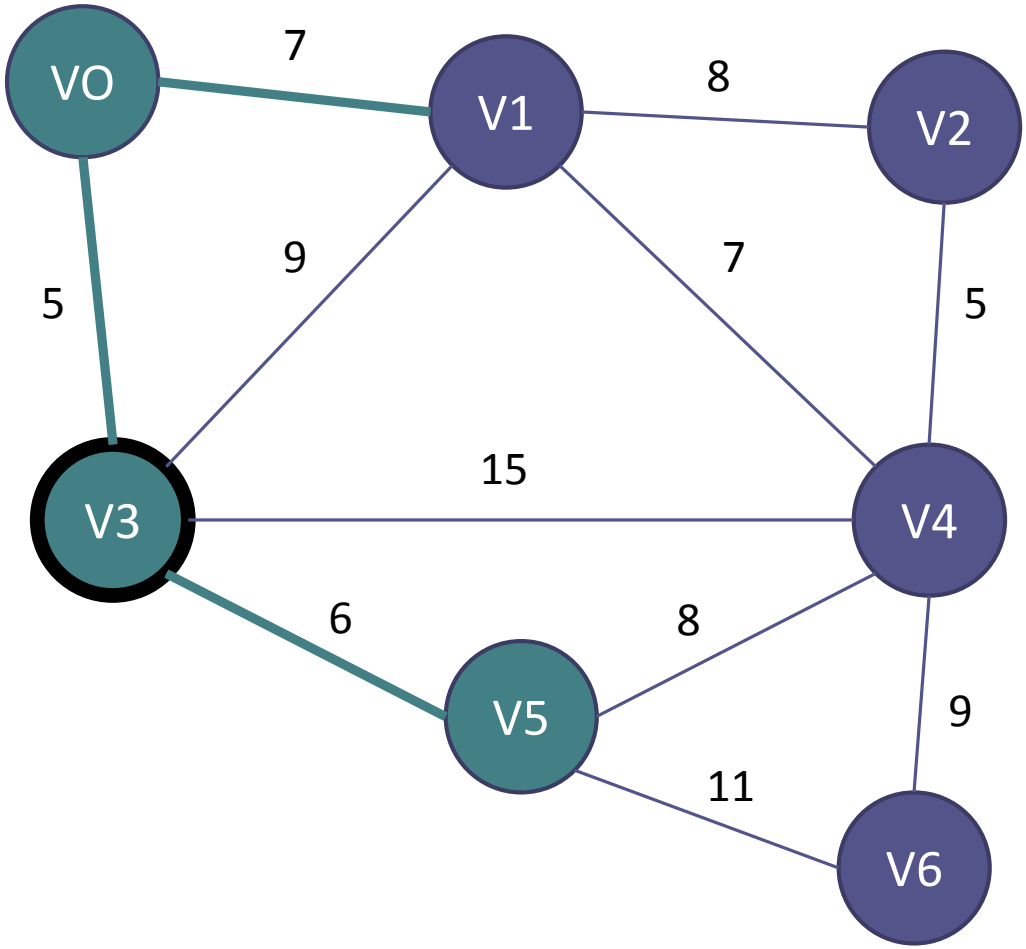
Process



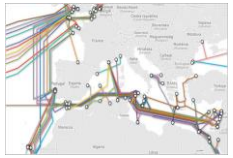
Tree	Options	Graph
{V0, V3}	~~{V3, V1} = 9~~ {V3, V4} = 15 **{V3, V5} = 6** {V0, V1} = 7	{V1, V2, V4, V5, V6}



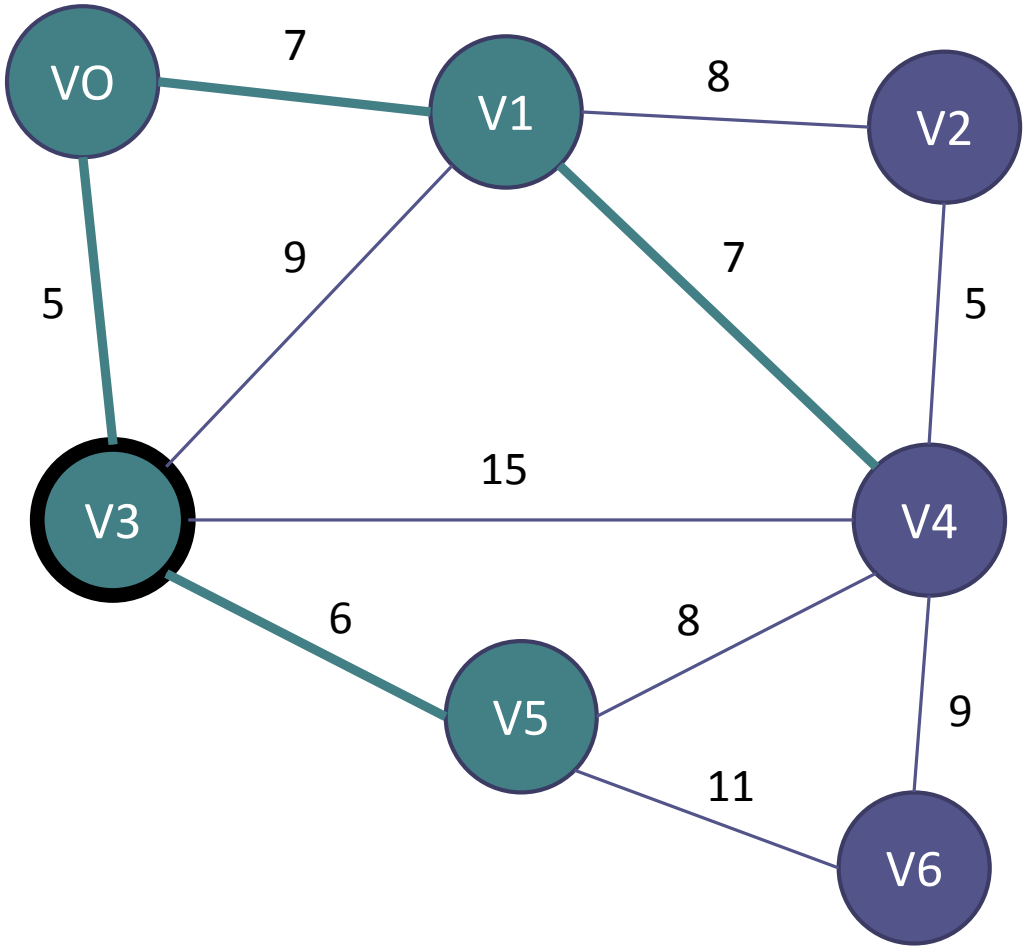
Process



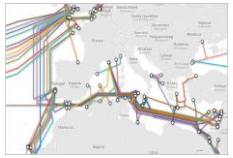
Tree	Options	Graph
{V0, V3, V5}	~~{V3, V1} = 9~~ ~~{V3, V4} = 15~~ **{V0, V1} = 7** {V5, V4} = 8 {V5, V6} = 11	{V1, V2, V4, V6}



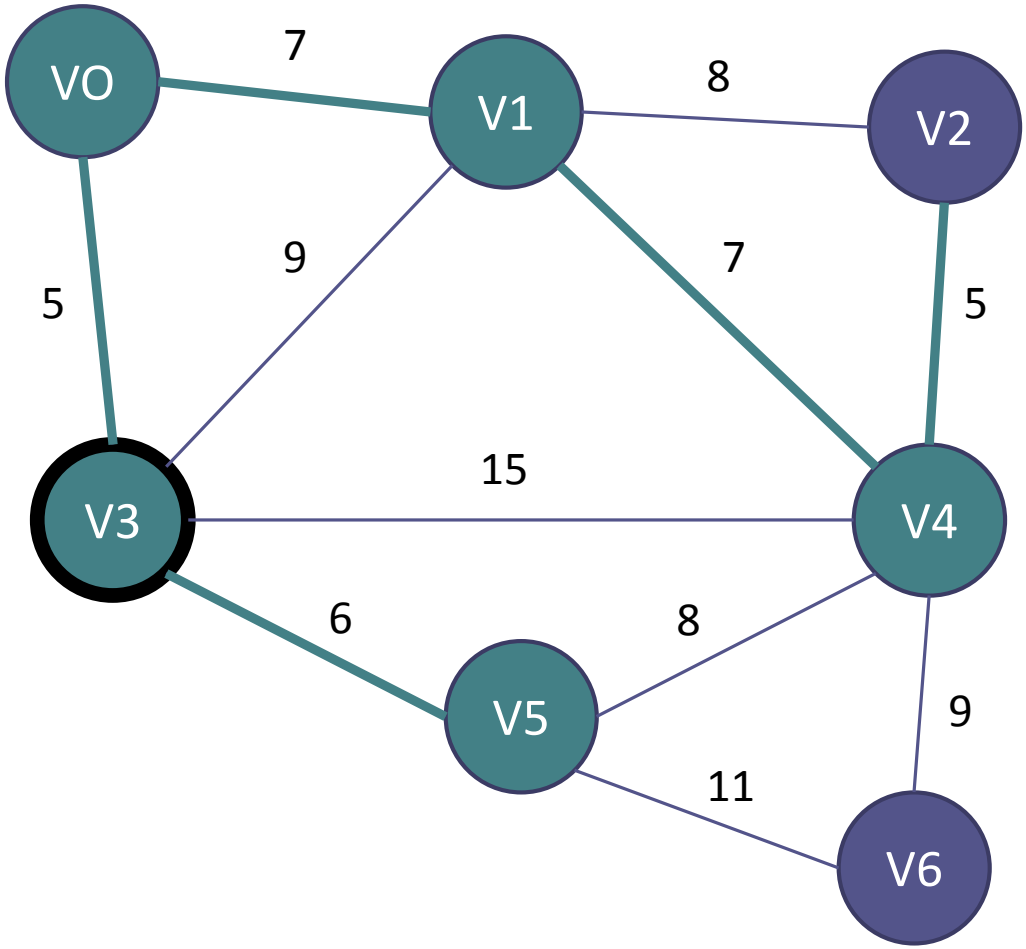
Process



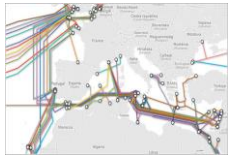
Tree	Options	Graph
{V0, V1, V3, V5}	~~{V3, V4} = 15~~ ~~{V5, V4} = 8~~ {V5, V6} = 11 {V1, V2} = 8 **{V1, V4} = 7**	{V2, V4, V6}



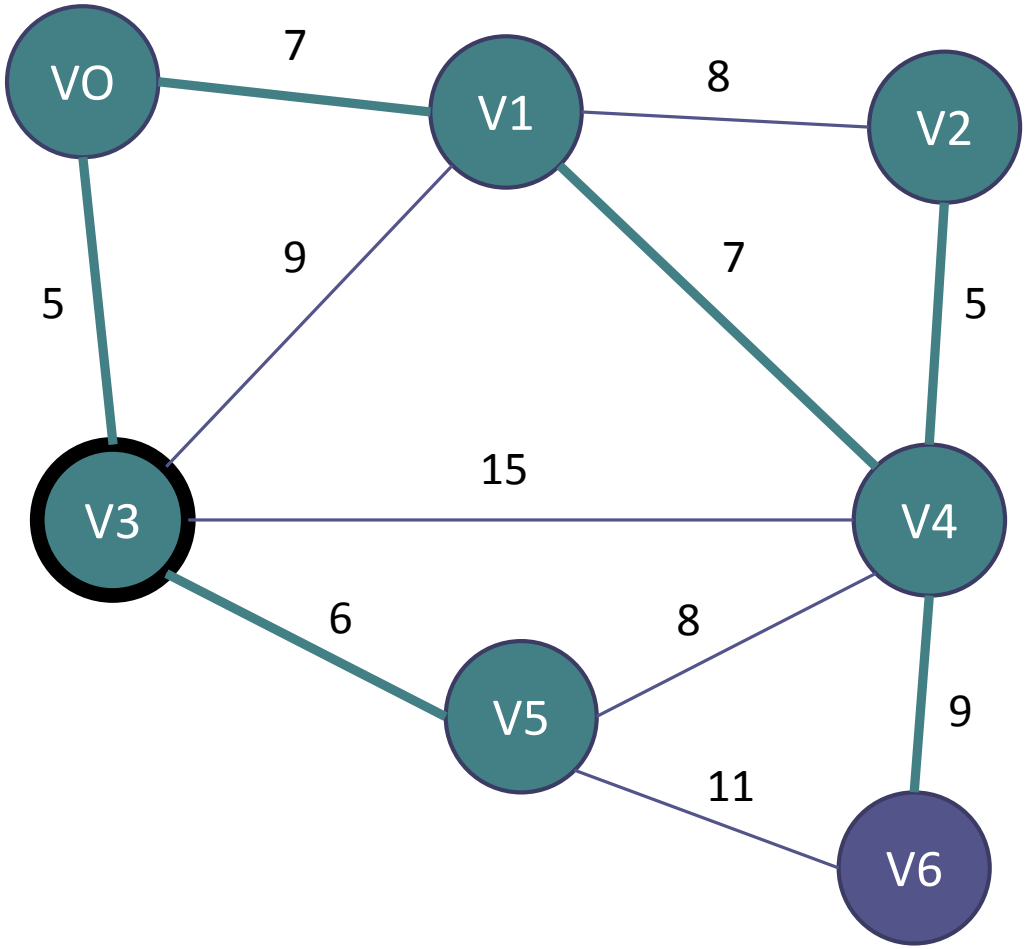
Process



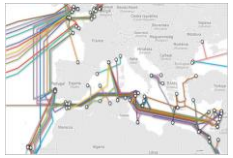
Tree	Options	Graph
{V0, V1, V3, V4, V5}	~~{V5, V6} = 11~~ ~~{V1, V2} = 8~~ **{V4, V2} = 5** {V4, V6} = 9	{V2, V6}



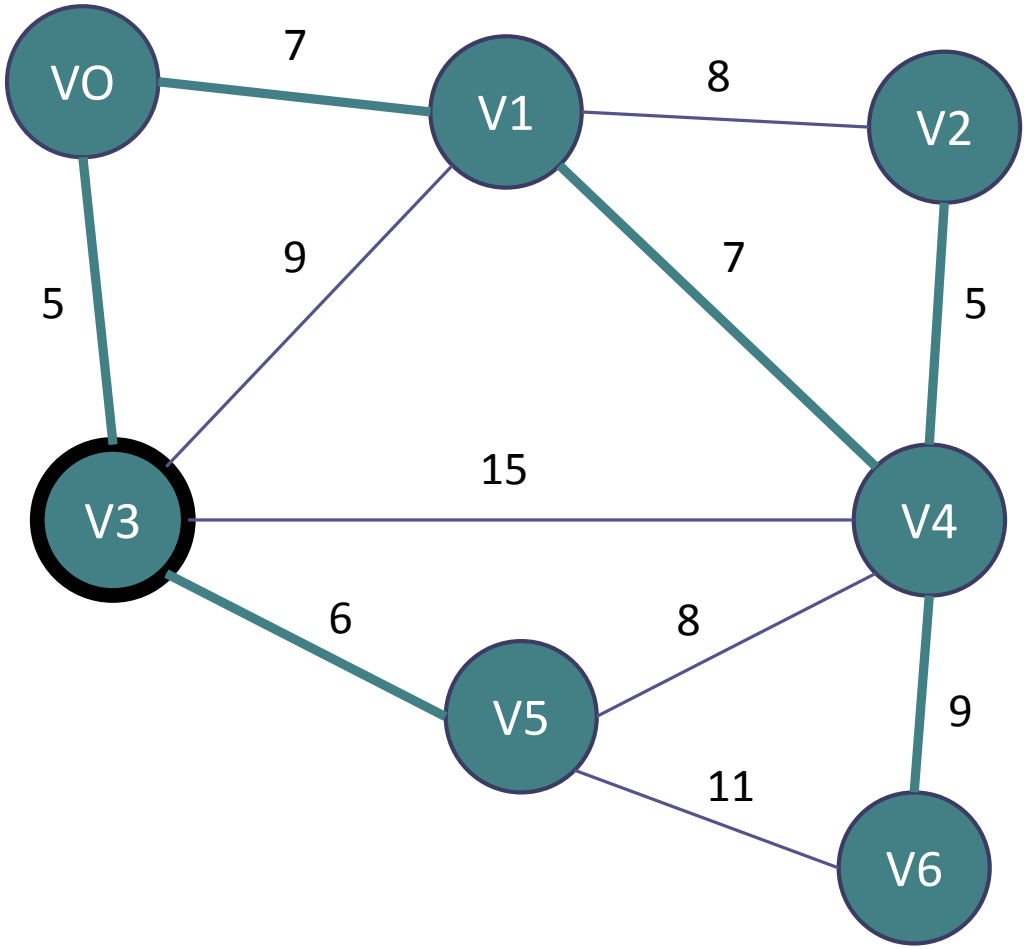
Process



Tree	Options	Graph
{V0, V1, V2, V3, V4, V5}	~~{V5, V6} = 11~~ **{V4, V6} = 9**	{V6}

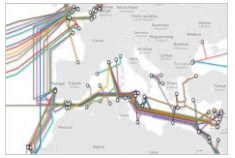


Process



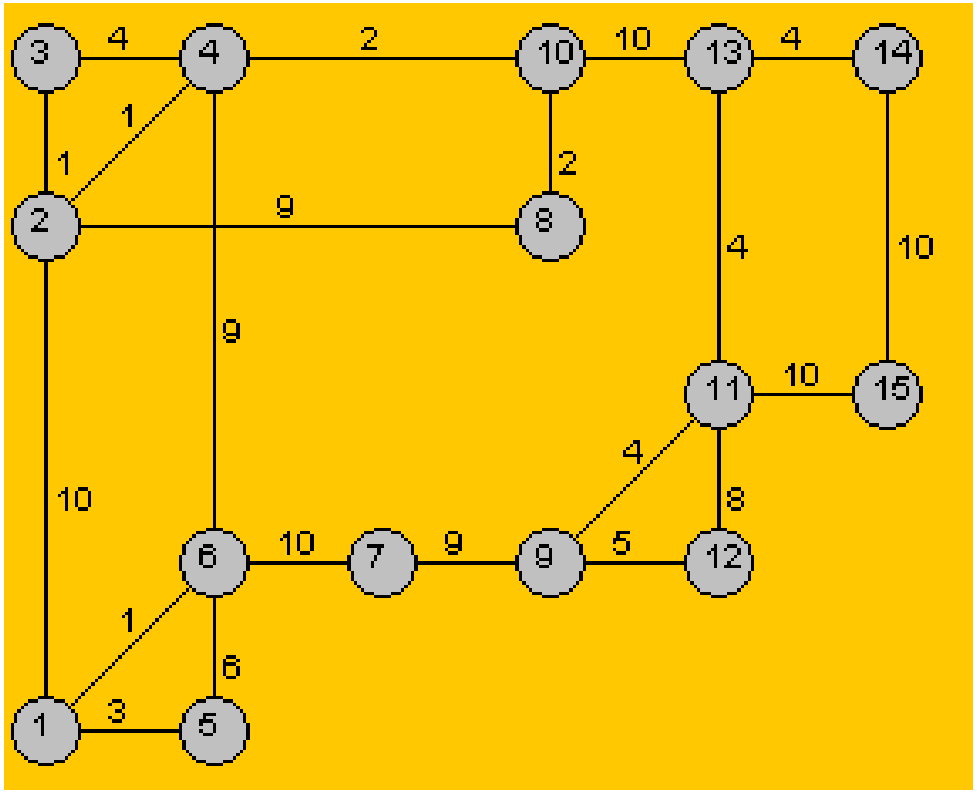
Tree	Options	Graph
{V0, V1, V2, V3, V4, V5, V6}		{}

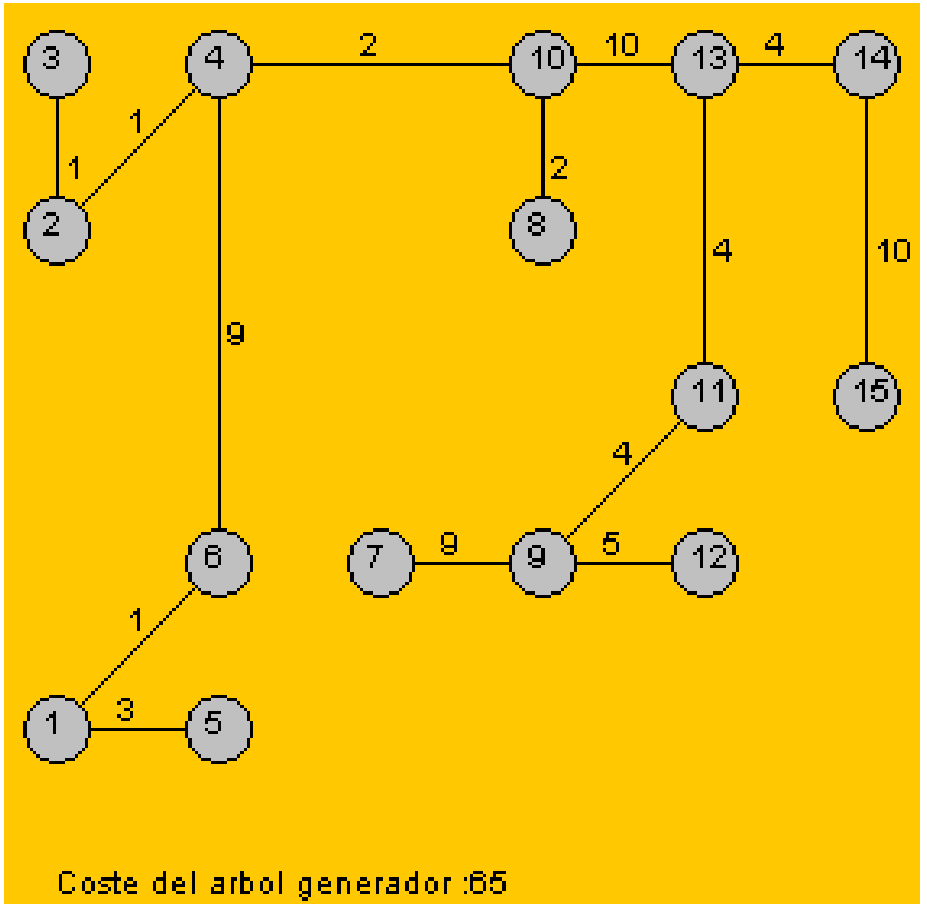
minimumSpannigTree()?



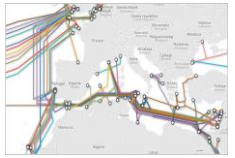
Exercise 1

- Given the following graph that represents the potential cost of wiring an installation (for example), calculate the path whose cost is minimal, and its cost



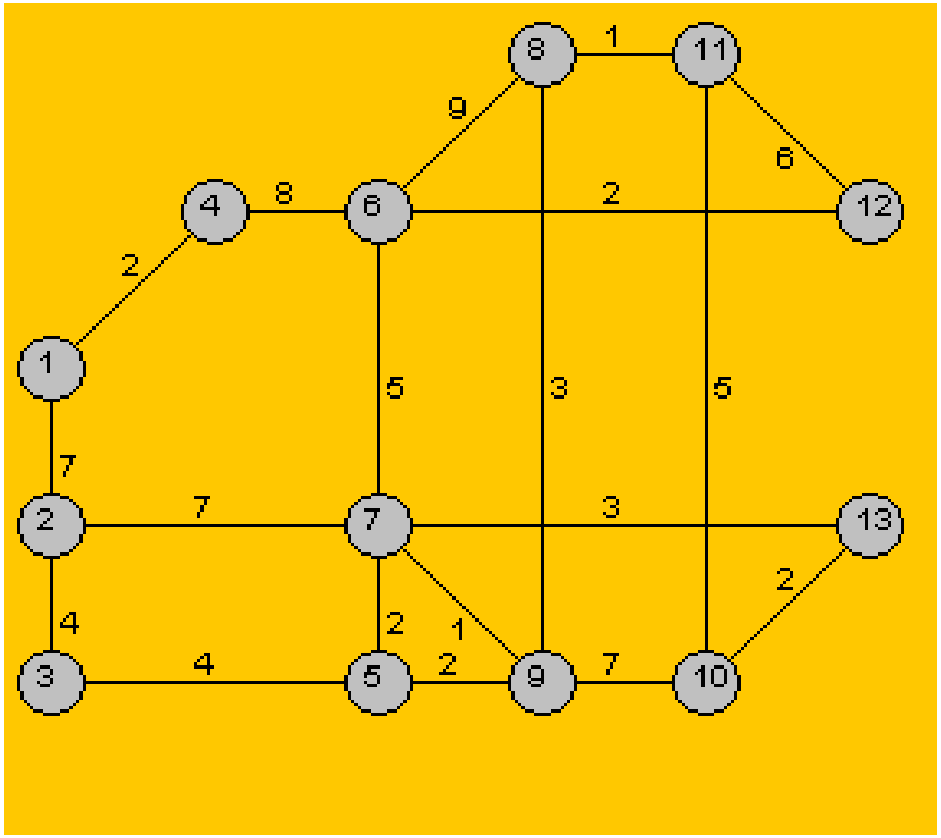


Más detalles



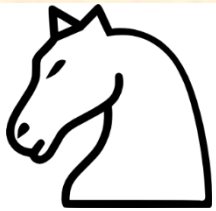
Exercise 2

- Given the following graph that represents the potential cost of wiring an installation (for example), calculate the path whose cost is minimal, and its cost



Más detalles

Goal



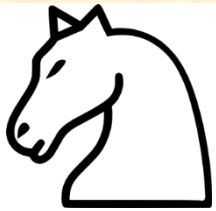
- It consists in fully going through a chessboard based on the movements performed by the horse

	5		4	
6				3
		*		
7				2
	8		1	

- The cells that a horse can access from a given cell are shown in the following tables

1		7			
8				6	
	2				
	9				5
		3		…	
		10		4	

Heuristics



1. **A simple one**: try to go to the first available position

`jump()?`

2. **A more effective one**: try to go to the new position from which we have less possible jumps

`newMovement()?`

Goal

- We must assign `j` tasks to `i` agents, so that each agent performs only one task
- We have a matrix of costs, to know the cost of performing a task `j` by an agent `i`
- Goal: Minimizing the sum of the costs for executing the `n` tasks

j (tasks) →

i (agents) ↓

	1	**2**	**...**	**n**
a	11	12	...	40
b	14	15	...	22
...				
n	17	14	...	28

Heuristics

1. **One possibility**: get the minimum from each of the rows
   - i.e., Assign to each agent the less expensive task available

2. **Another possibility**: get the minimum for each for the columns
   - i.e., Assign each task to the agent for which is less expensive

`greedy2() & getCost2()?`

Bibliography

JUAN RAMÓN PÉREZ PÉREZ; (2008) ***Introducción al diseño y análisis de algoritmos en Java***. Issue 50. ISBN: 8469105957, 9788469105955 (Spanish)