1.1

Combinatorial Optimization

Jean-Charles Régin

Master Computer Science1st Year

Remerciements

- Wikipédia
- Courses and exercises from ENS Lyon (Yves Robert, Yves Caniou and Eric Thierry)
- Roman Bartak

Goal

- I would like you to understand that computers can be used for solving complex problems
  - Web and video are not the only one usage
- Clever use of the computation power
- It is quite important to program without any bug (or with only a limited number of bugs)

Plan

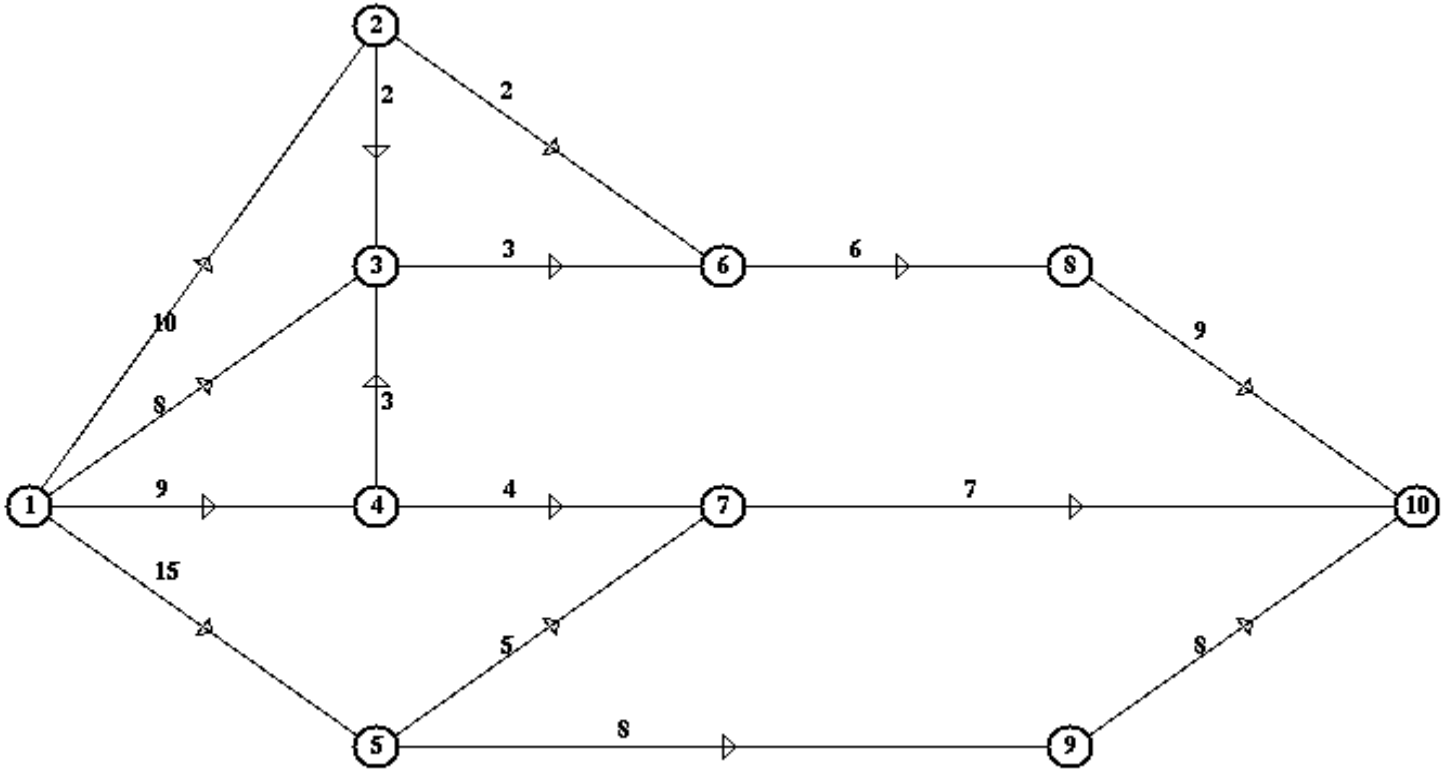
- Introduction
- Greedy algorithms
- Multi-valued Decision Diagrams
  - Data structures
  - Problem solving with operations only

1.5 Introduction

Graph : definitions

- A directed Graph G=(X,U) is defined by
  - A set of **vertices** or **nodes** X
  - A set of nodes couple U, where a couple is called an arc
- If u=(i,j) is an arc of G then i is the initial extremity of u and j the terminal extremity of u.
- The arcs have a direction. The arc u=(i,j) is from i to j.
- Arcs may have a cost, a capacity, a length, a weight etc…

Graph



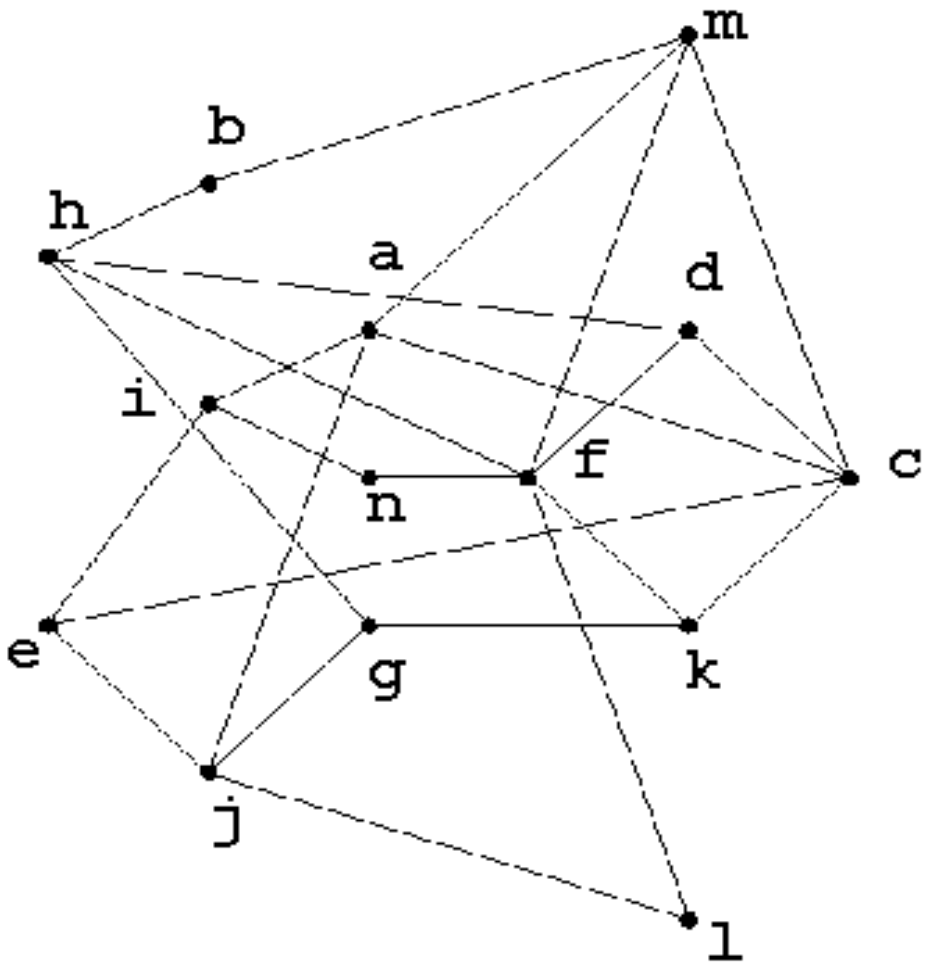
Graph

- We will denote by $\omega(i)$: the set of arcs having i as an extremity
- We will denote by $\omega^+(i)$: the set of arcs having i as an initial extremity = set of arcs outgoing i.
- We will denote by $\omega^-(i)$: the set of arcs having i as a terminal extremity = set of arcs incoming i
- N(i) : set of neighbors of i : set of nodes j such that it exists an arc from i to j

Undirected Graph

- An undirected Graph G=(X,E) is defined by
  - A set of **vertices** or **nodes** X
  - A set of pairs of nodes called edges
- The edges are not oriented

Undirected Graph



Graph : definitions

- Two nodes are neighbor if there are linked by an arc or an edge
- Incoming degree of i : number of incoming arcs of i
- Outgoing degree of i : number of outgoing arcs of i

Graph : definitions

- Directed path of length q : sequence de q arcs $\{u_1,u_2,\ldots,u_q\}$ such that
  - $u_1=(i_0,i_1)$
  - $u_2 = (i_1,i_2)$
  - $u_q=(i_{q-1},i_q)$
- Directed path : all the arcs are oriented in the same direction
- Directed cycle : directed path having the same extremities

Plan

- Short history of the TSP
- Easy and difficult problems
- Optimization and Decision Problems
- Difficult problems : limits of the resolution

Problem

- This is a general question : shortest paths between nodes, rostering…
- Define by some data and a question
- Answering to the question is solving the problem
- In computer science, we want a general answer to the problem: an algorithm working for every cases.
- **Instance** = a particular set of data. For instance, the shortest path between Nice and Roma.

Introduction

- Some problems are easy :
  - Sorting numbers
  - Searching for substrings…
- Some other are difficult
  - Example : TSP

Traveling Salesman Problem (TSP)

- **Data :** a list of city and for eahc pair of city, the distance between them.
- **Question :** find the shortest tour which visit each city exactly once.

- Mathematical Reformulation :
  - Given a complete weighted graph, find an hamiltonian cycle whose weight is minimum

TSP

Each city is visited once

Only one tour (no subtour)

TSP

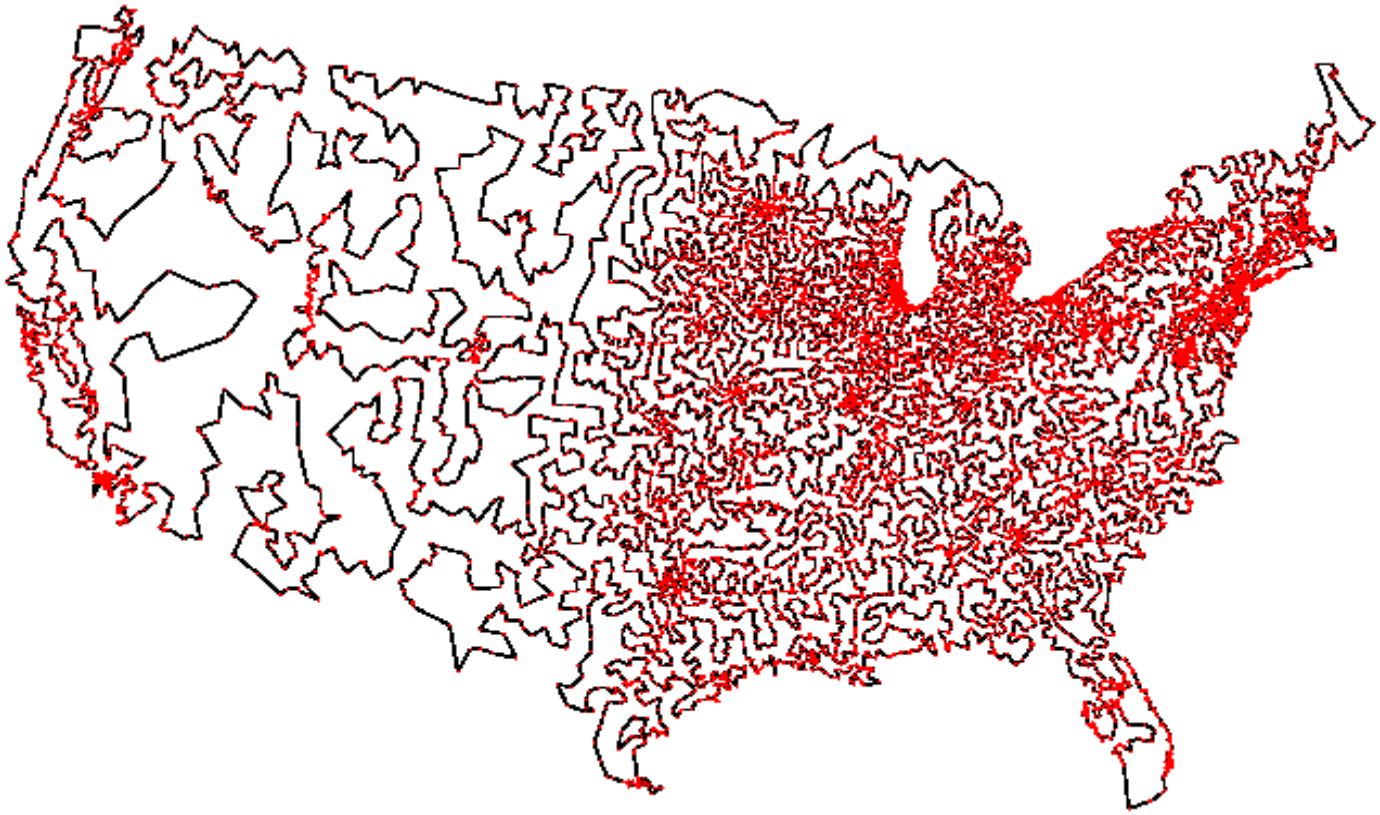
Each city is visited once

Only one tour (no subtour)

TSP

- Some problems are equivalent to the TSP
- Example : scheduling problems : find the order in which we have to build objects with hydraulic press
- The « pure » version of the TSP does not arise frequently in practice. We often meet instances that are
  - Non euclidian
  - Asymmetrical
  - Incomplete. We do not want to cover all the nodes but just a subset
- These variations do not change the difficulty of the problem
- Common problems
  - Vehicle routing (time windows, pickup and delivery…)

Procter and Gamble Contest 1962



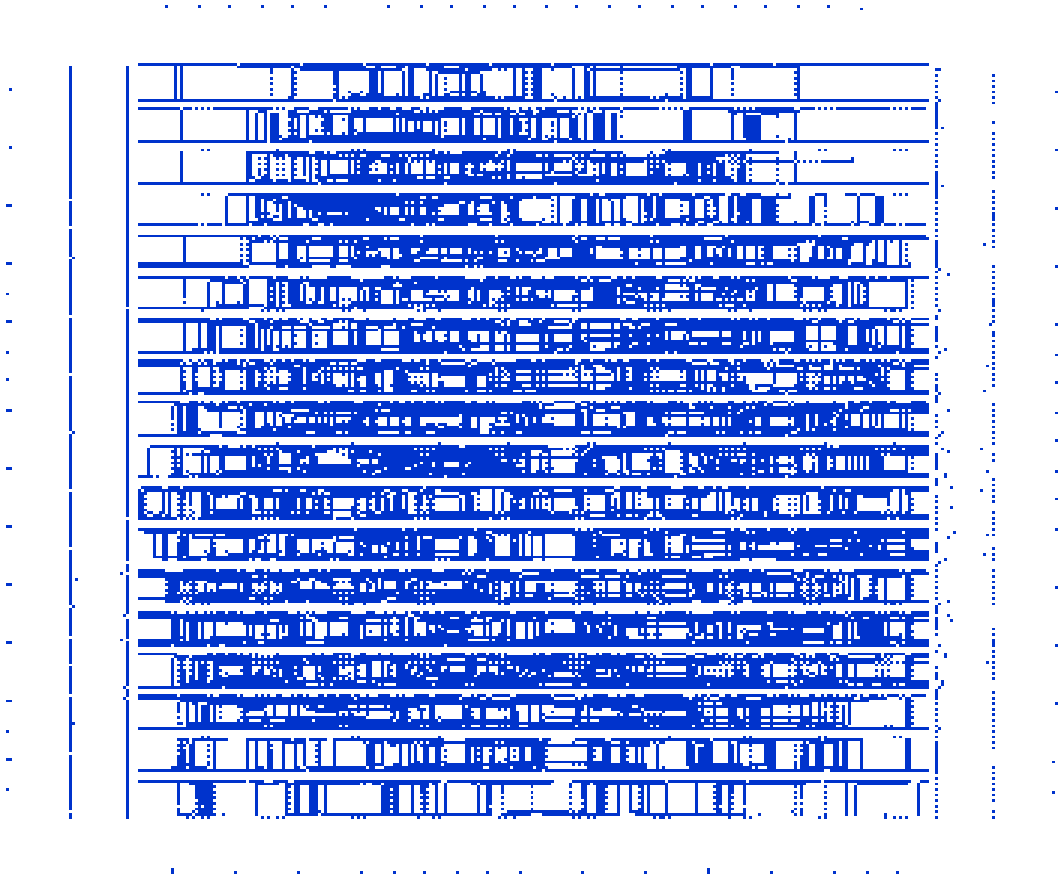
USA 13,509 cities. Solved in 1998

German Tour

- The 20 April 2001, David Applegate, Robert Bixby, Vašek Chvátal, and William Cook solved the TSP for the 15 112 cities of Germany.
- Network of 110 processors (550 Mhz) at Rice and Princeton University.
- Total time of resolution **22.6 years.**

Germany 15,112 cities. Solved in 2001

Sweden 24,978 cities. Solved in 2004



VLSI 85,900. Solved in 2006

TSP

- There exist several solvers
- The most well known is Concorde (William Cook). Concorde is free
- Lin-Kernighan's heuristic is very efficient in practice

TSP Solvers

- TSP solvers are generally dedicated to the resolution of the pure TSP
- It is alsmost impossible to use them when changing a little bit the problem (asymmetrical, side constraints …)

Plan

- Short history of the TSP
- **Easy and difficult problems**
- Optimization and Decision Problems
- Difficult problems : limits of the resolution

Algorithms

- All the algorithms are not equivalent.
- Polynomial algorithms
- NP-Complete algorithms

Computers are very fast

- Chess game :
  - A player has about 10 possible moves in average,
  - I have 10 moves, my opponent has 10 moves for each of my 10 moves.
  - If I play twice then 10(me)x10(him)x10(me) combinations to evaluate.
  - If I play 3 times then 10(me)x10(him)x10(me)x10(him)x10(me) combinations.
- If I play k times then 10^{2k-1} combinations

Chess game

- If I play k times then 10^{2k-1} combinations
- If I play 5 times then 10 000 000 = 10 millions combinations
- 20 years ago (since 1993)
  - The best program was playing like an Intl Master
  - The frequency of the computer was 100Mhz
- Today
  - The best program is the best player in the world. It costs almost nothing ($50 or $100)
  - Computer runs at 3Ghz

Chess game

- Increasing of the computer power
  - frequency : 3Ghz vs 100 Mhz
  - multicores vs mono core
  - Best generated code (compilers)
  - Best architecture of processors (CPU manufacturer)
  - Dhrystone benchmark (from 1984) : calculations made only for integers
    - Pentium (100Mhz) Dhry2 opt = 122,
    - Core i7 930 (3Ghz) Dhry 2 opt = 8684
    - Ratio : 8684/122 = 71,18. (30 for frequency)

Chess game

- 70 time faster in 20 years (30 for the frequency, 2,5 for the architecture)
  - 1 move = 10 (me) * 10 (him) = 100 combinations
  - Roughtly in 20 ans we save 1 move for a monocore
- 2 move (2 for me and 2 for) = 100*100=10000.
  - It requires 100 cores !
- **Thus we will not solve a problem by expecting only a progress of the computer**

NP-Completeness

- **The major problem** of computer science, today:
  - P vs NP

Example of difficult problem

- Hitting-set :
  - Bulbs and switches :
    - A switch is linked to some bulbs
    - If we set a switch on, then all the bulbs linked to them are ligthed up
    - Question : how many switches must be turned on to light up all the bulbs?
    - We want a general answer working for all the instances

Example of difficult problems

- Hitting-set and set cover
- Subset sum
- Knapsack
- Bin packing
- Graph coloring
- Maximal clique

Graph coloring

- Joining-contracting
- For coloring a complete graph (a clique) having n nodes we need n colors
- Select 2 nodes a and b that are not neighbor
  - Either they ahve the same color
  - Or they have a different color
- Same color = contraction
- Different color = addition of an edge

Graph coloring

- Select 2 nodes a and b that are not neighbor
  - Either they ahve the same color
  - Or they have a different color
- We obtain a clique: the number of nodes gives the number of colors
- Pb : m potential edges. 2 choices per edge (same color or different color) = 2^m

Easy vs Difficult

- The difference can be tight. Be careful with side constraints.
- Consider a 0-1 matrix. For each row and for each column, we know the number of 1. Define the value of each cell
  - The pure problem is easy
  - We add a connexity constraint: difficult
  - We add a convexity constraint: difficult
  - We add both: easy!

Easy vs Difficult

- Coloring of the edges (two edges having a common extremity must have different color):
  - Number of colors = max degree or max degree + 1
- Finding the correct value is an NP-Complete problem!

Plan

- Short history of the TSP
- Easy and difficult problems
- **Optimization and Decision Problems**
- Difficult problems : limits of the resolution

Decision Problem

- A **decision problem** is a mathematical question defined on given data and expecting a **yes or no answer.**
- Given a set of city and a distance d, is there a path traversing each city and whose length is less than d, is a decision problem
- Can we color a graph with k colors?

Optimization problem

- An optimization problem is a problem for which we must find the **best solution** of a set of feasible solutions
- An optimization problem A for an instance x is defined by
  - f(x) : the set of feasible solutions
  - c(y) : the cost function of a feasible solution y (solution cost).
  - An objective function which is usually a min, or a max
- An optimal solution is a feasible solution which respects the objective, thus such that its cost is the minimum (resp. maximum) of all feasible solutions
- If the goal is to minimize the cost function then we say that we have a minimization problem; otherwise we say that we have a maximization problem
- It is often possible to transform the minimization problem into a maximization one by changing the sign of the costs.

Optimization Problem

- DATA : A graph G
- QUESTION : What is minimum number of colors needed to color the nodes of G such that two adjacent nodes have different colors?

- Shortest path between two nodes?

Optimization Problem

- We need to differentiate two notions
  - The optimal solution
  - The proof that a solution is an optimal solution (proof of optimality)
- Be careful: do not generalize too much
  - Finding the optimality and prove it can be slow or fast.
  - One can be fast and the other slow.

Optimization and Decision

- For each optimization problem there is a decision problem asking if there is a solution having a particular cost
- Example : We find a shortest path from s to t whose cost is c.
  - Décision Problem:
    - Is there a path of cost c?
  - **Proof of optimality**
    - Is there a path whose cost is < c?

Optimization and Decision

- Optimization problems are often solved by solving a succession of decision problems.
- A feasible solution is computed. It has a cost k
- We search for a solution having a cost < k and we repeat the process
- At the end, we prove the optimality because the latest resolution is not able to find a solution.

Plan

- Short history of the TSP
- Easy and difficult problems
- Optimization and Decision Problems
- **Difficult problems : limits of the resolution**

Difficult Problems

- We do not know if difficult problems can be solved quickly
- This means that we do not have a polynomial algorithms. Therefore, an exponential appears.

The problem

- n teams and n-1 weeks and n/2 periods
- every two teams play each other exactly once
- every team plays one game in each week
- no team plays more than twice in the same period

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
Period 1	0 vs 1	0 vs 2	4 vs 7	3 vs 6	3 vs 7	1 vs 5	2 vs 4
Period 2	2 vs 3	1 vs 7	0 vs 3	5 vs 7	1 vs 4	0 vs 6	5 vs 6
Period 3	4 vs 5	3 vs 5	1 vs 6	0 vs 4	2 vs 6	2 vs 7	0 vs 7
Period 4	6 vs 7	4 vs 6	2 vs 5	1 vs 2	0 vs 5	3 vs 4	1 vs 3

First model: results

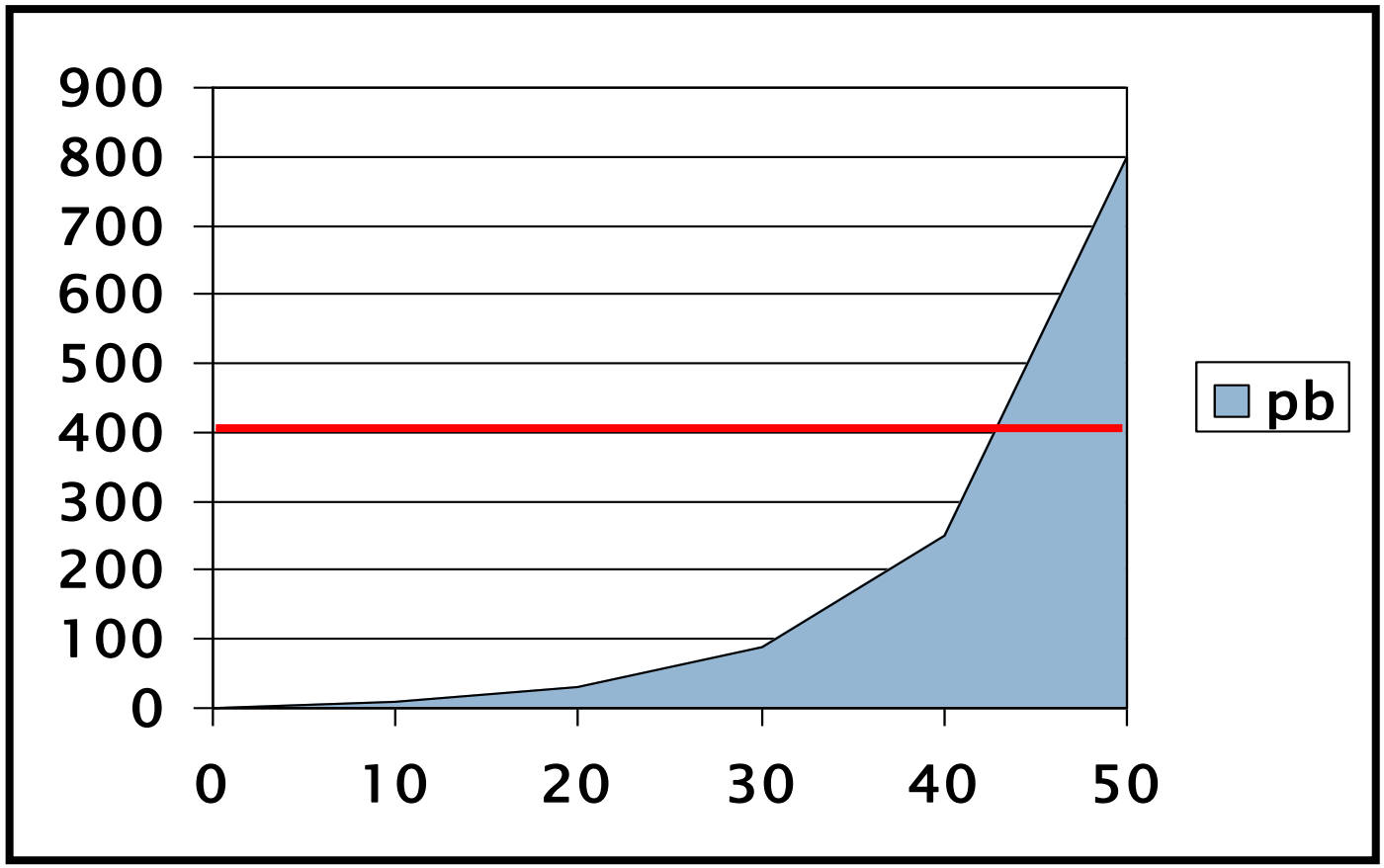
# teams	# fails	Time (in s)
4	2	0.01
6	12	0.03
8	32	0.08
10	417	0.8
12	41	0.2
14	3,514	9.2
16	1,112	4.2
18	8,756	36
20	72,095	338
22	6,172,672	10h
24	6,391,470	12h

Second model: results

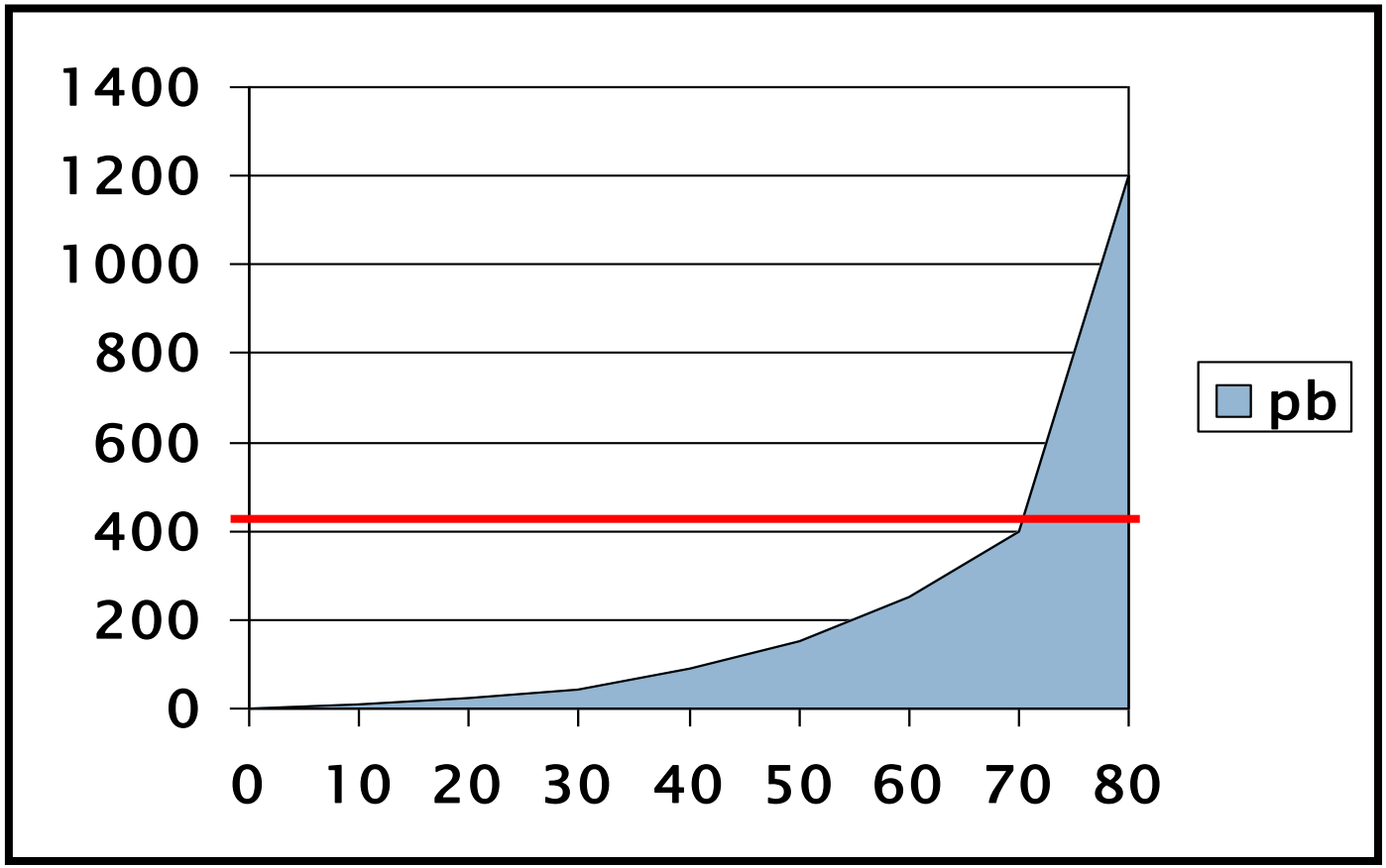
# teams	# fails	Time (in s)
8	10	0.01
10	24	0.06
12	58	0.2
14	21	0.2
16	182	0.6
18	263	0.9
20	226	1.2
24	2702	10.5
26	5,683	26.4
30	11,895	138
40	2,834,754	6h

First model limit

Shifting the exponential



Shifting the exponential



Difficult Problems

- We will see how to find some solutions for these problems
  - with heuristics (inexact but fast)
  - With a complete enumeration of possible combinations (exact but slow)

1.56

Greedy Algorithms

Plan

- Definition
- The knapsack problem
- Heuristic
- Some greedy algorithms
  - Selection of activities
  - Graph coloring
  - Covering: minimal transversal

Greedy Algorithm

- A **greedy algorithm** is an algorithm which follows a principle making at each step a choice leading to an local optimum expecting to reach a global optimal at the end.
- Dans les cas où l'algorithme ne fournit pas systématiquement la solution optimale, il est appelé une **heuristique gloutonne.**

Greedy algorithm

- The choice we make at each step is often named **greedy strategy**
- Example: give change with the minimum number of coins
  - Greedy Strategy: we use several steps. At each step we give back the largest possible coin
  - 37 cents. The largest coin less than the amount is 20, we now have 17. The largest possible coin is now 10, it remains 7. The largest possible coin is 5, it remains 2. The largest possible coin is 2. We used the coins 20+10+5+2

Greedy Algorithm

- In the european system, the greedy algorithm always returns an **optimal solution**
- In the system (1, 3, 4), the greedy algorithm is not optimal, as shown by the example: for 6 it gives 4+1+1, whereas 3+3 is optimal.

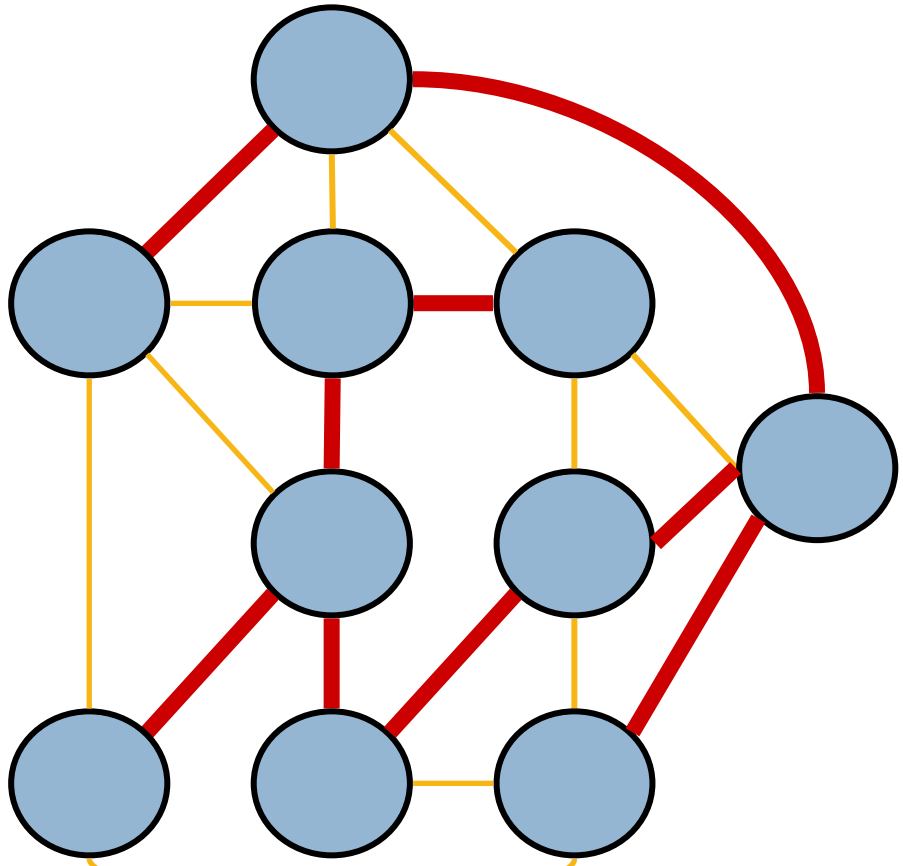
Greedy Algorithm

- Principle : be fast
- Difficulty : find a good strategy or an efficient strategy.

Greedy algorithm: proof

- We show a generic proof for greedy algorithm
- We present two proves for the spanning tree problem

Spanning Tree



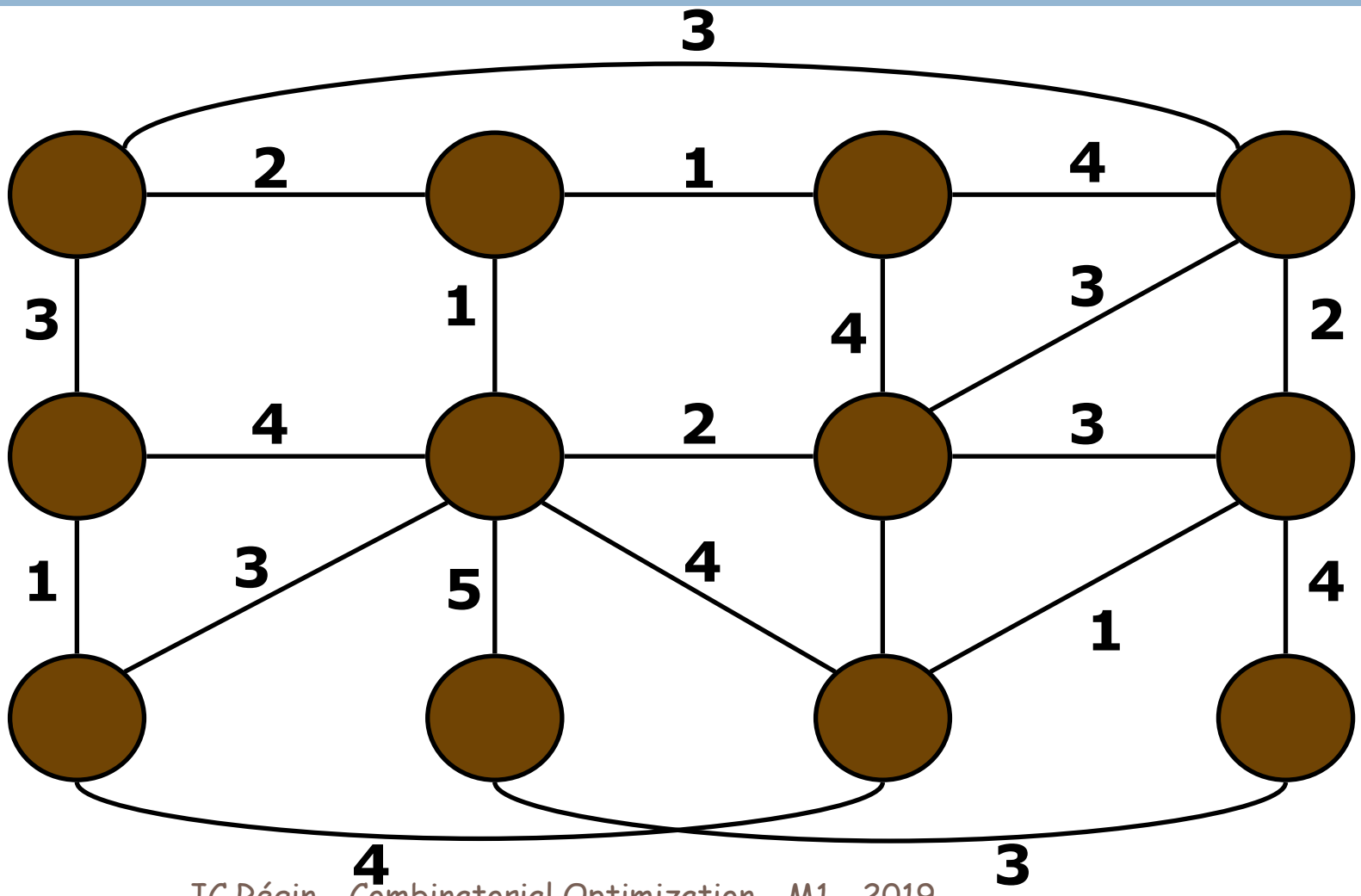
Minimum Spanning Tree

- Given a graph G, find a spanning tree where total cost is minimum.

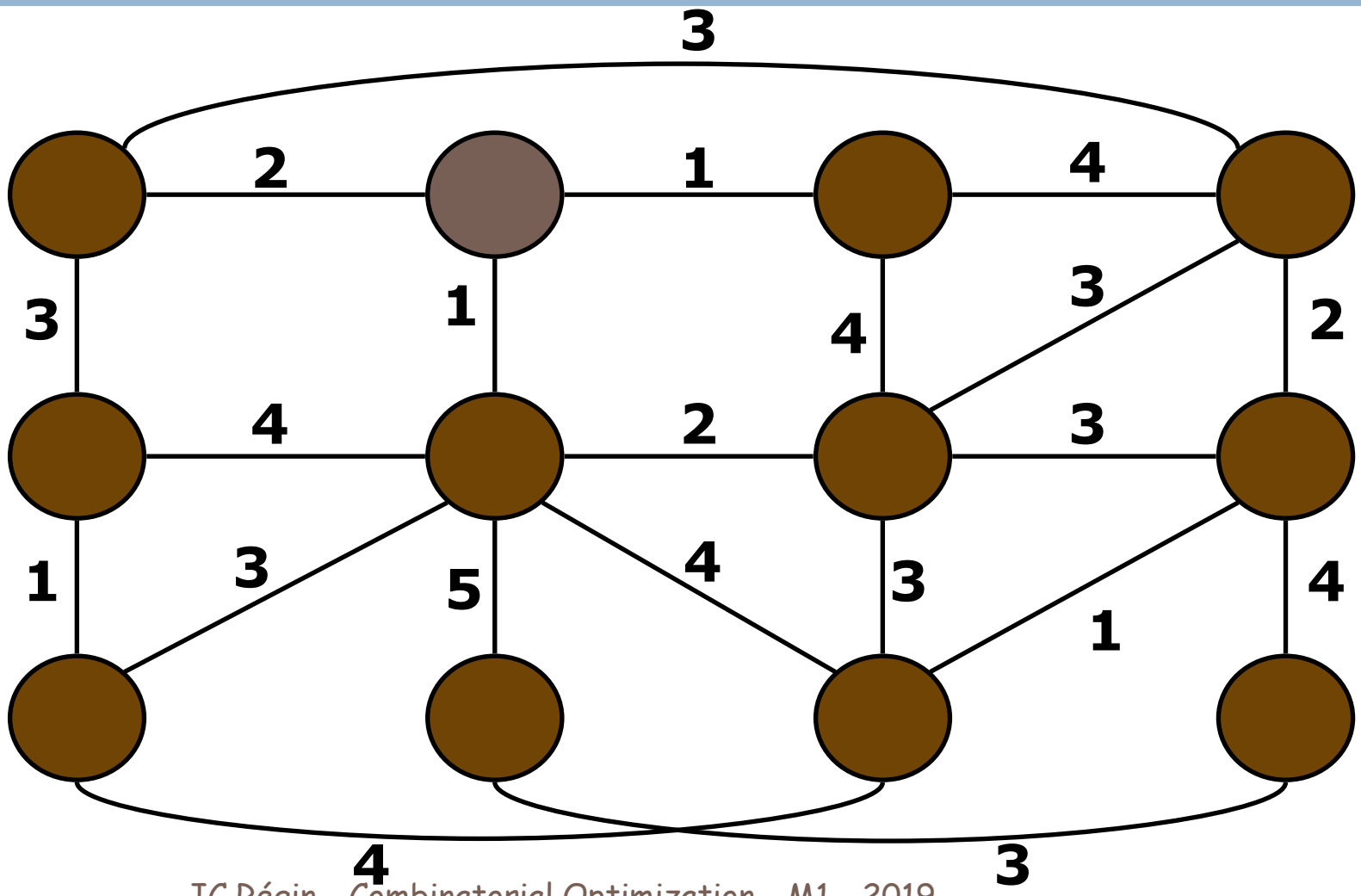
Prim's Algorithm

- We use a set S of nodes. We add one node to S
- We add successively nodes into S, one per step
- At each step, we select the edge between nodes of S and node not in S such that its cost is minimum. We add to S the extremity of this edge which is not in S

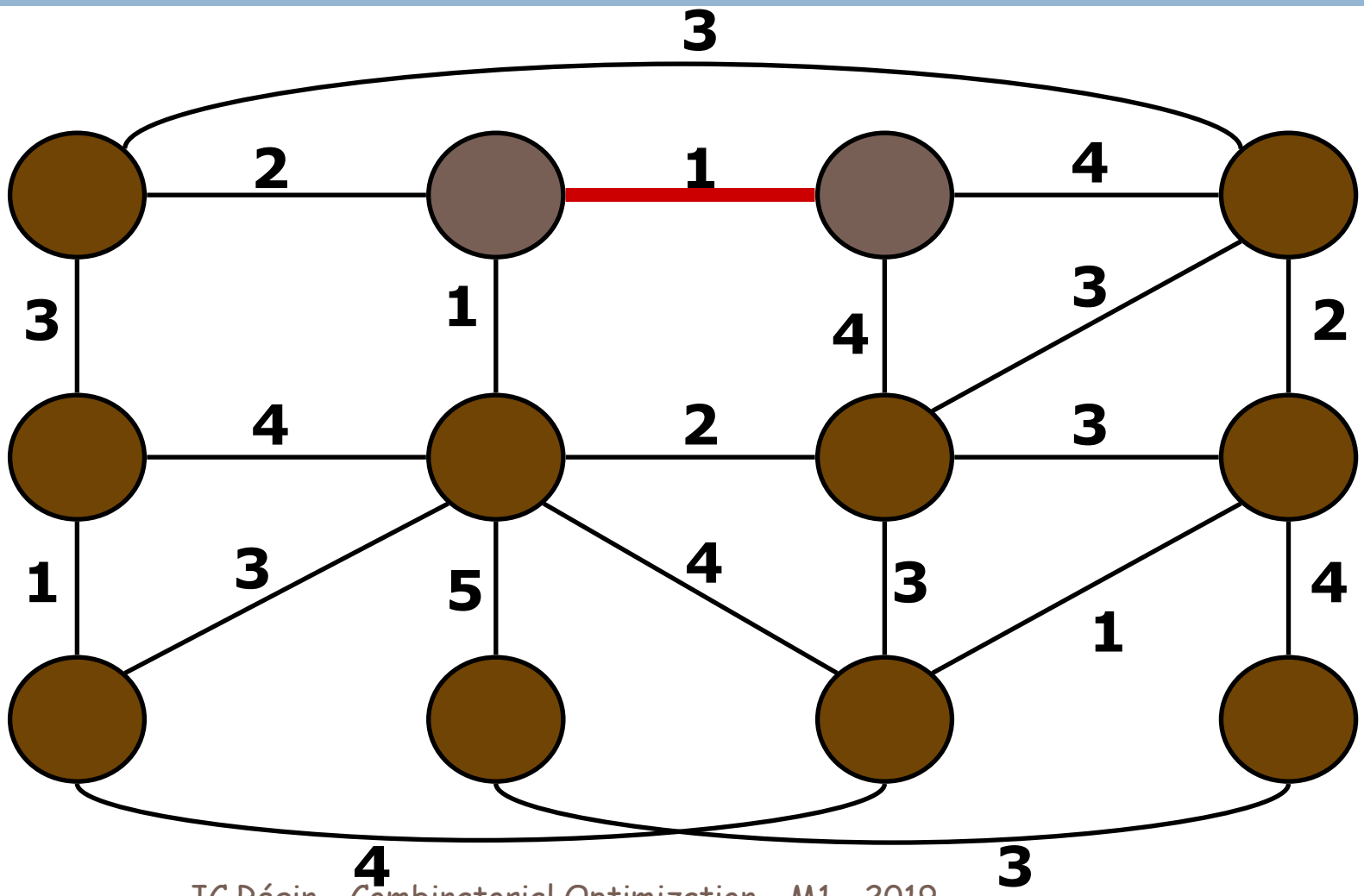
Prim's Algorithm



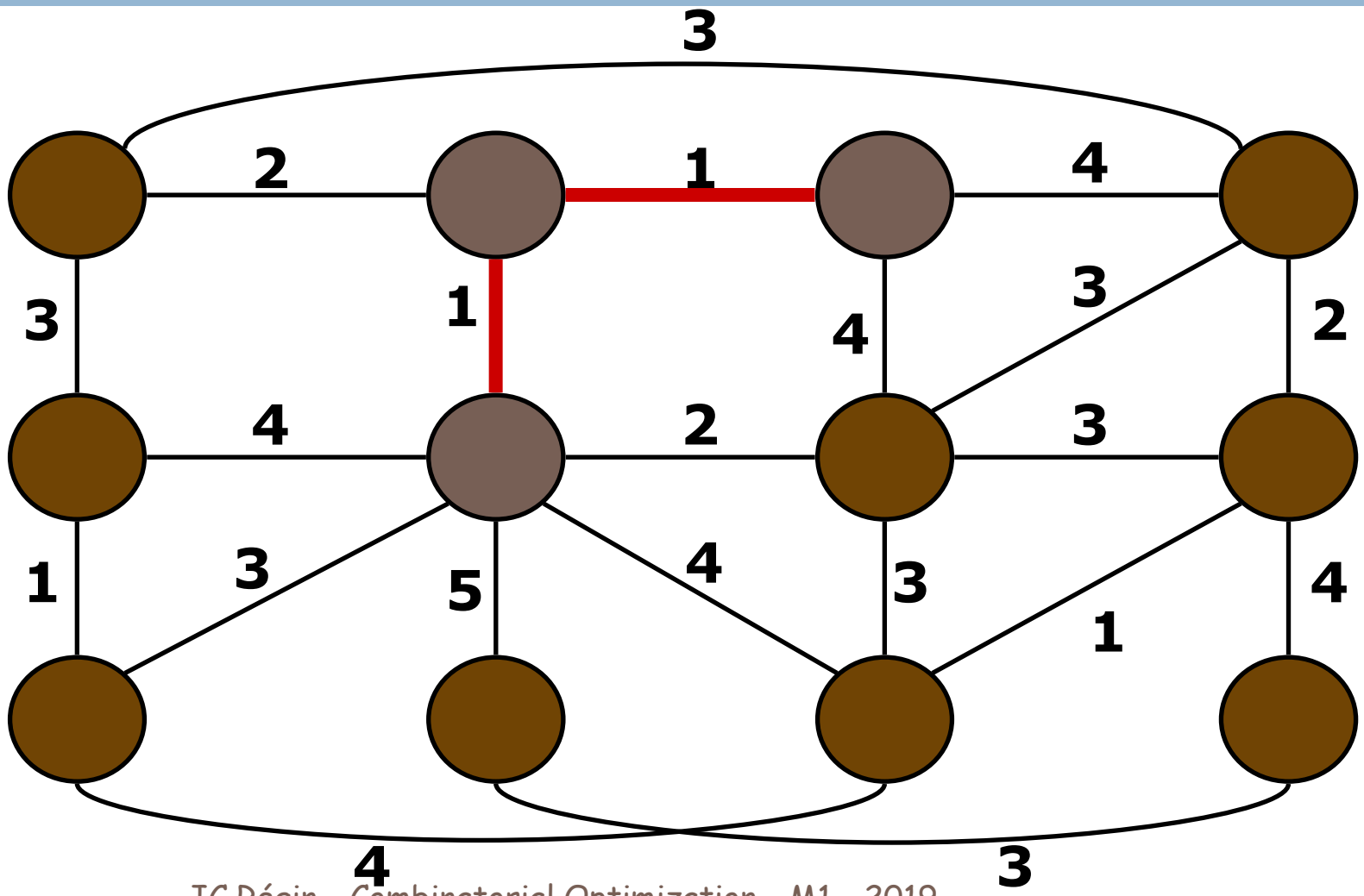
Prim's Algorithm



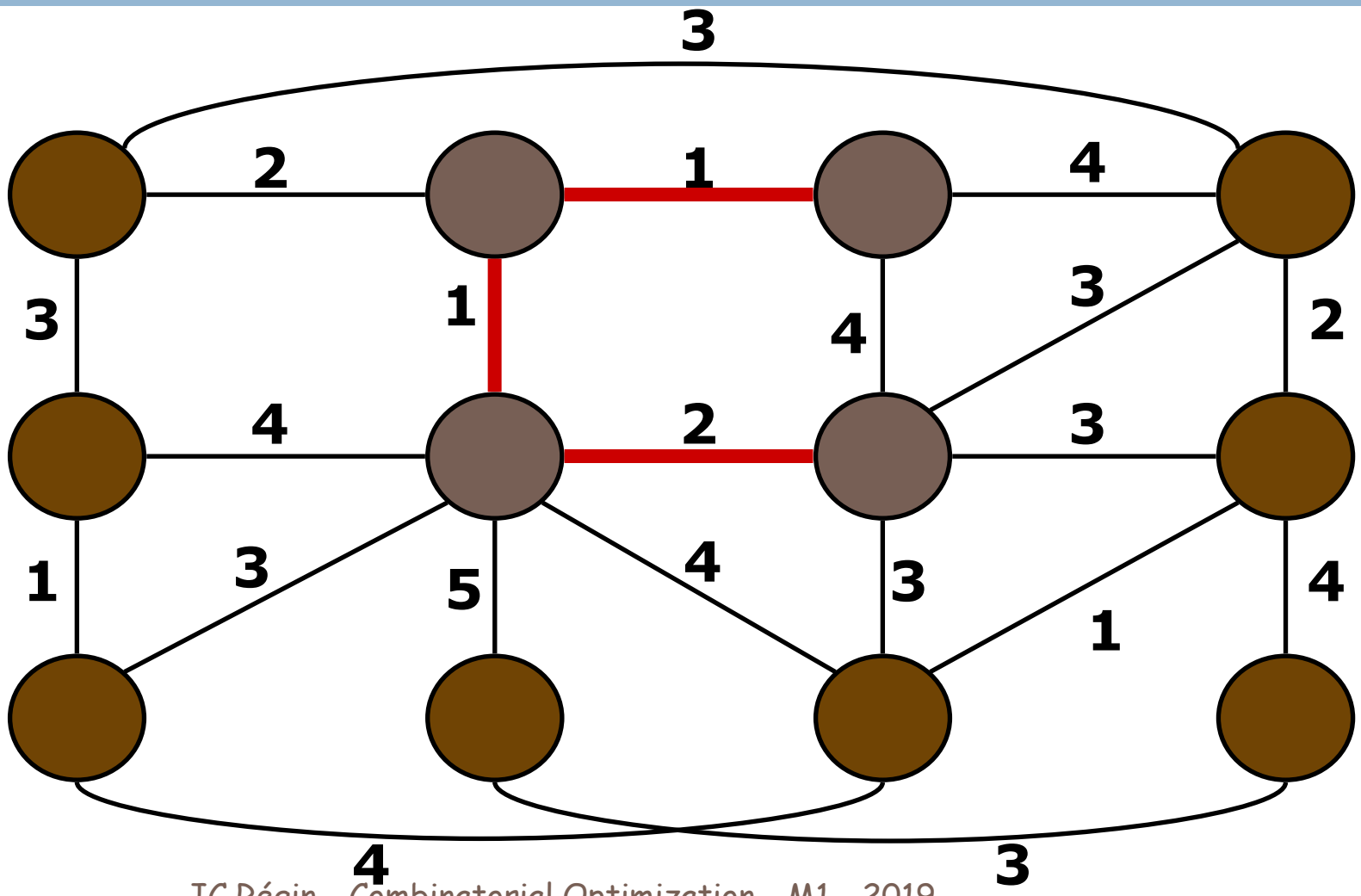
Prim's Algorithm



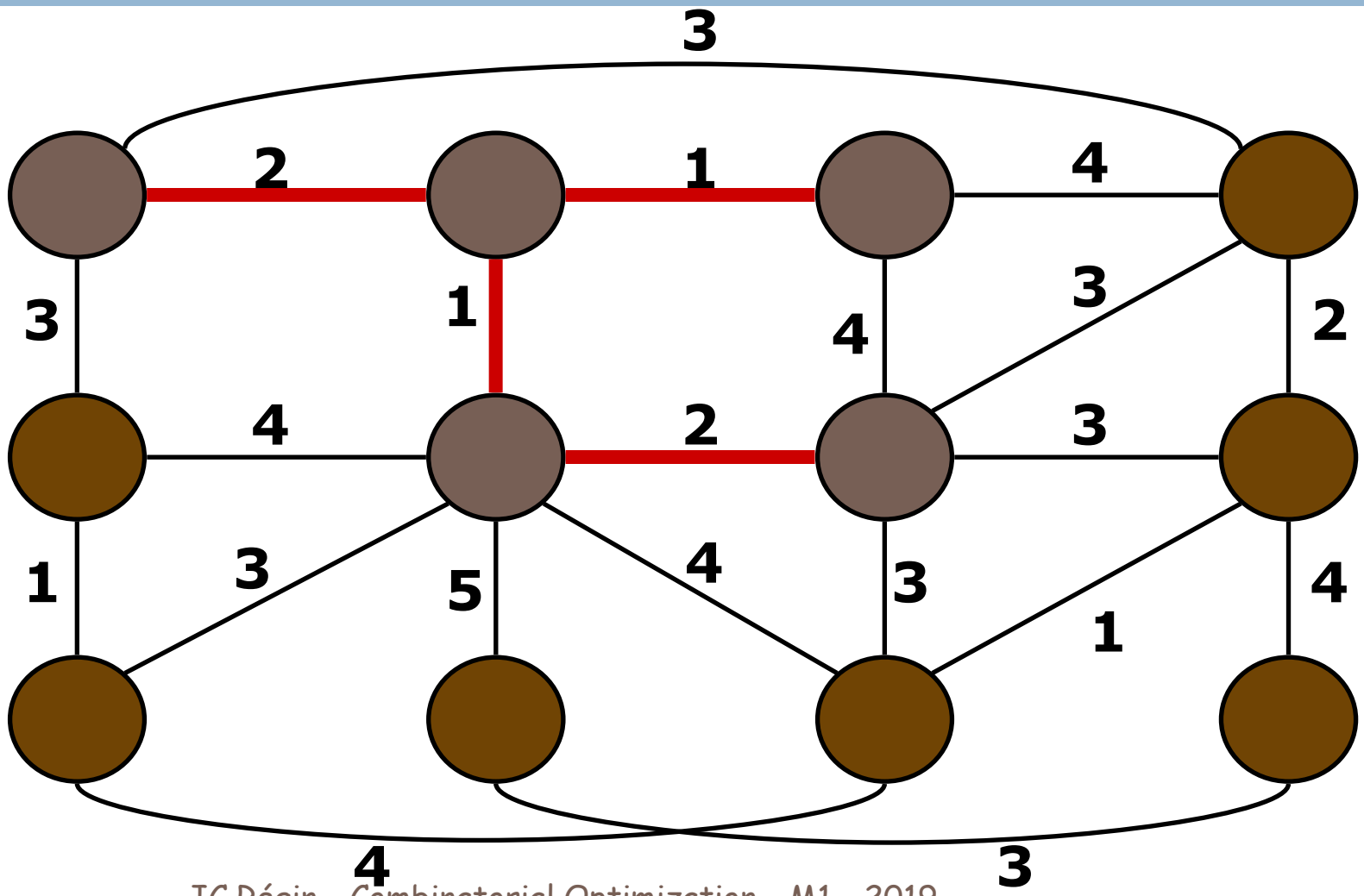
Prim's Algorithm



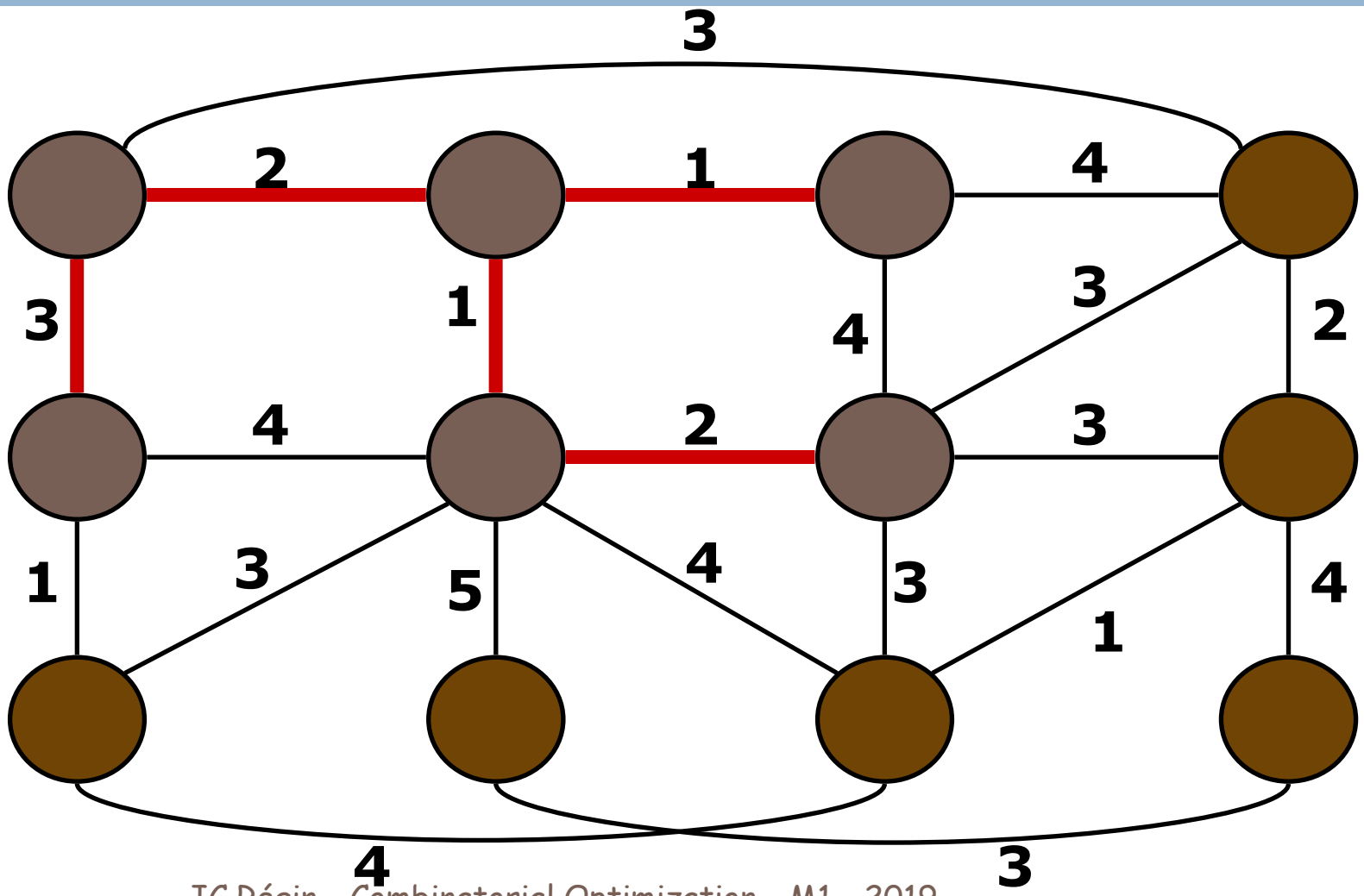
Prim's Algorithm



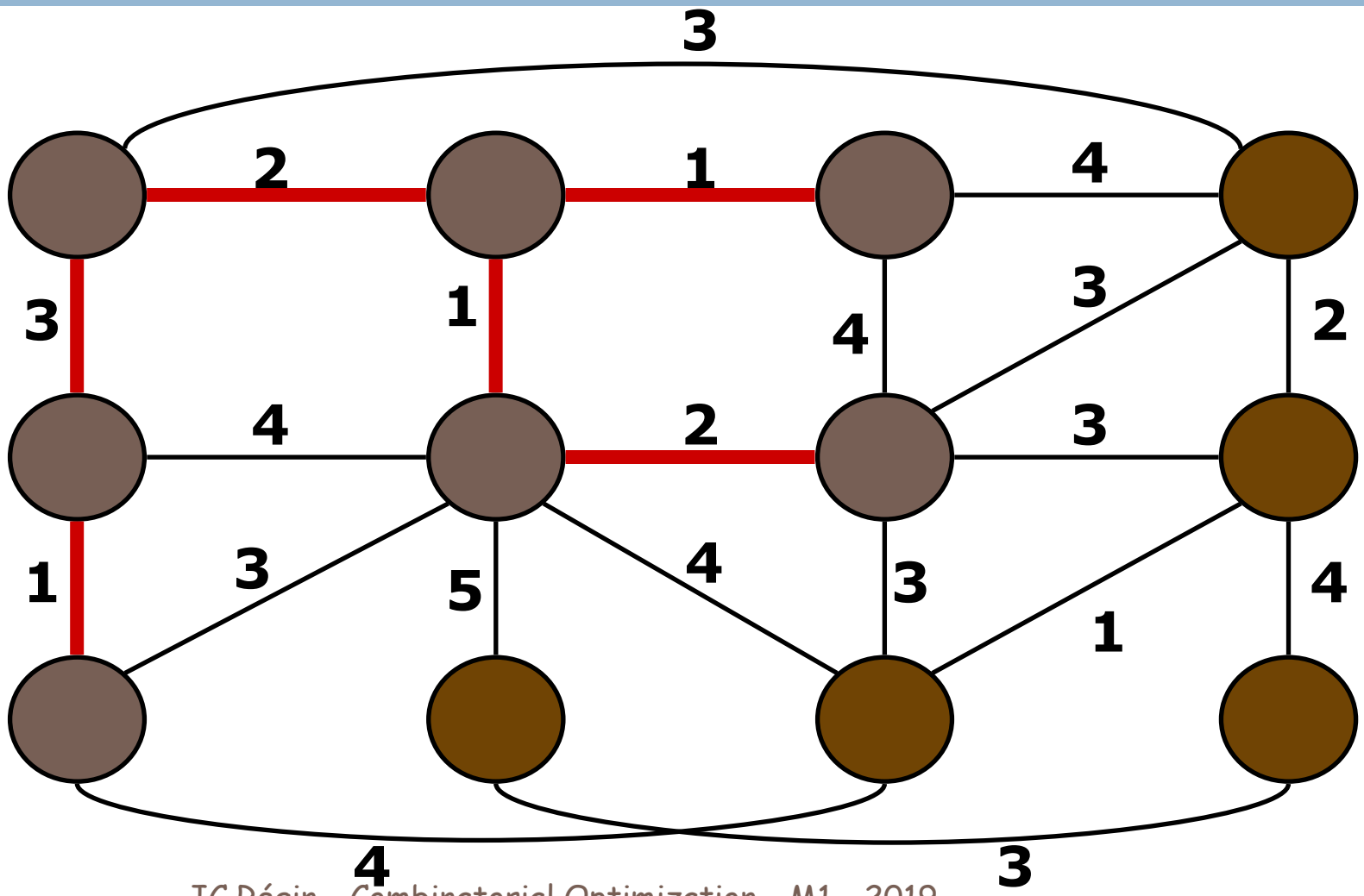
Prim's Algorithm



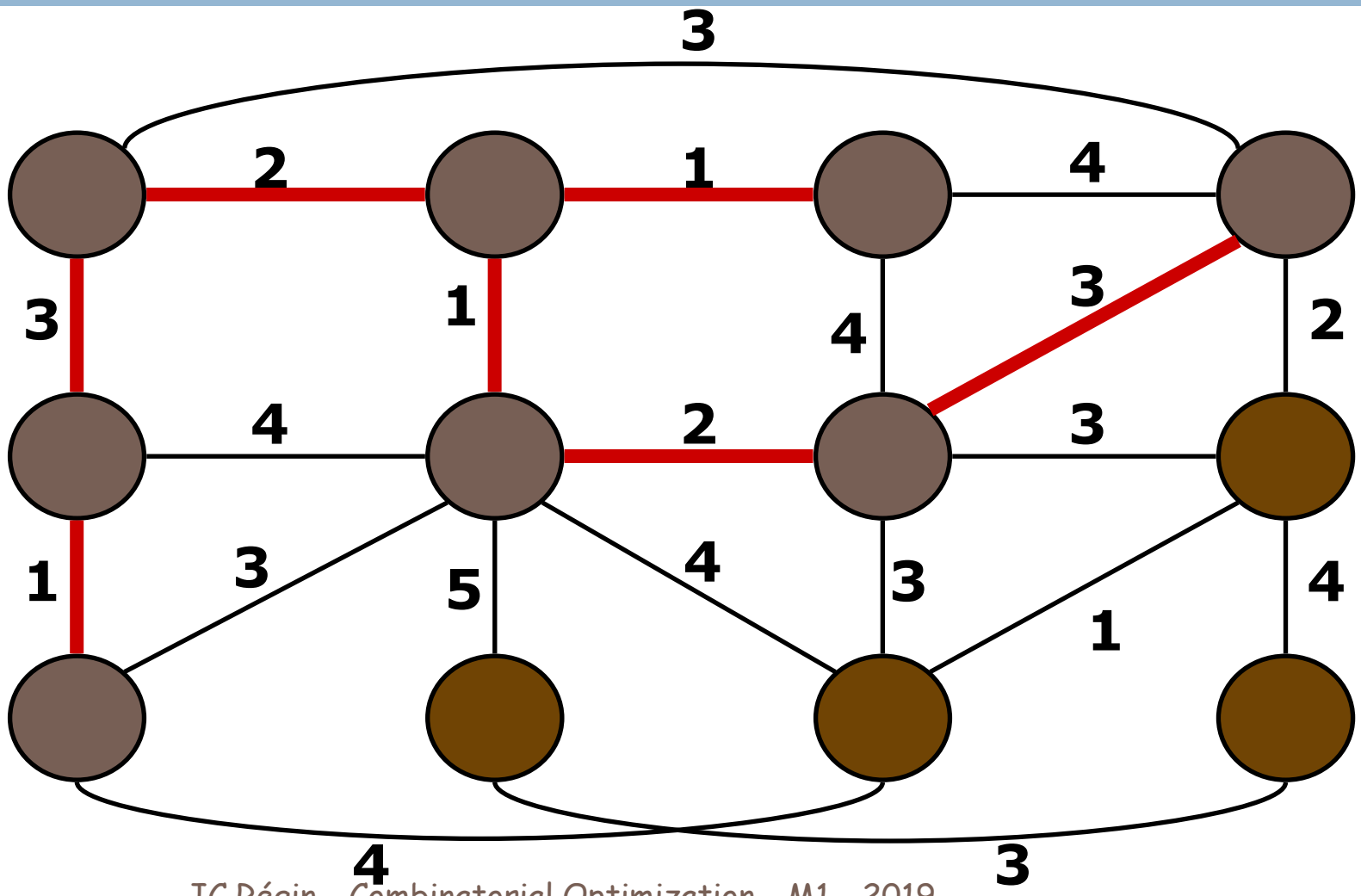
Prim's Algorithm



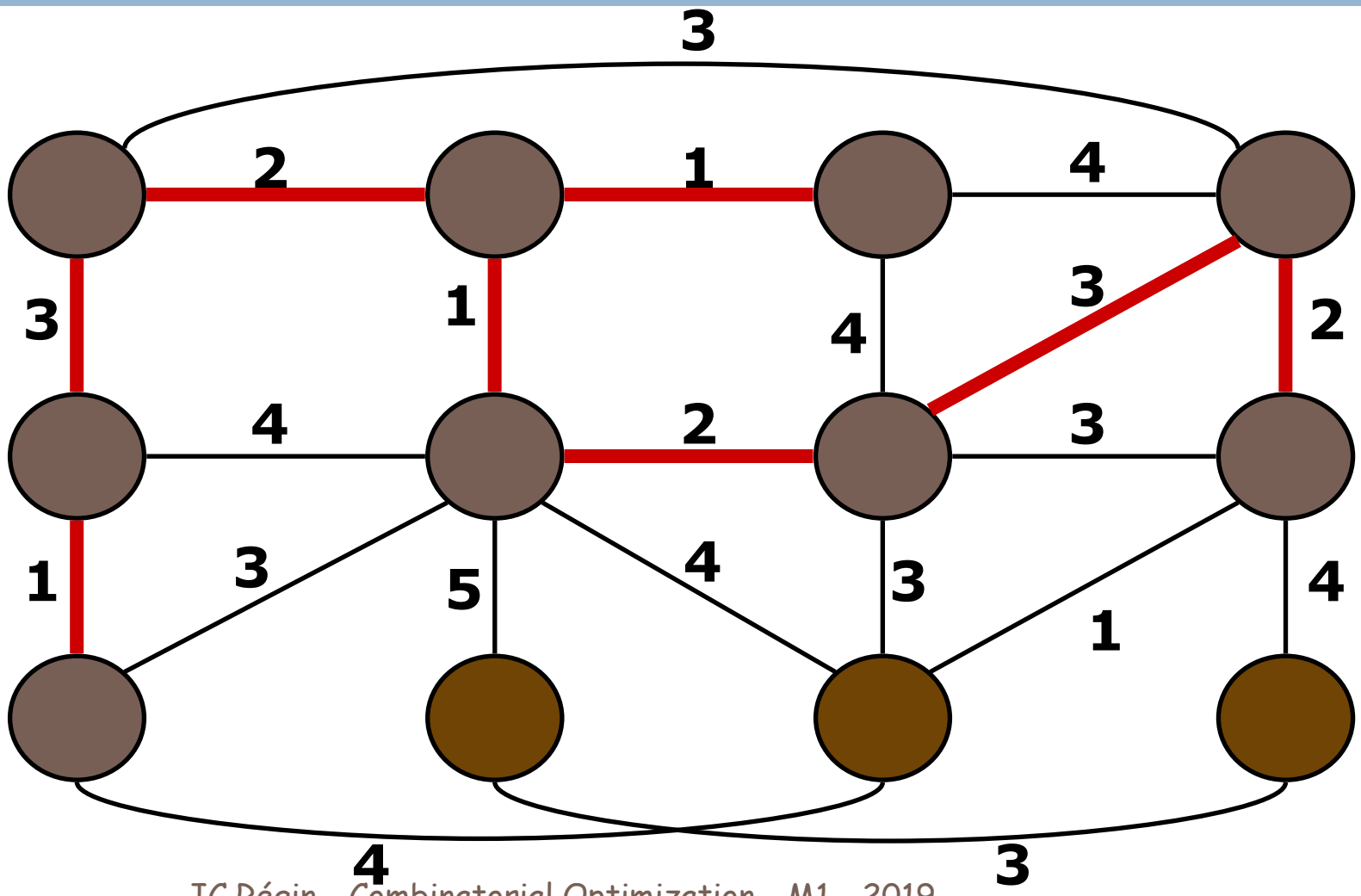
Prim's Algorithm



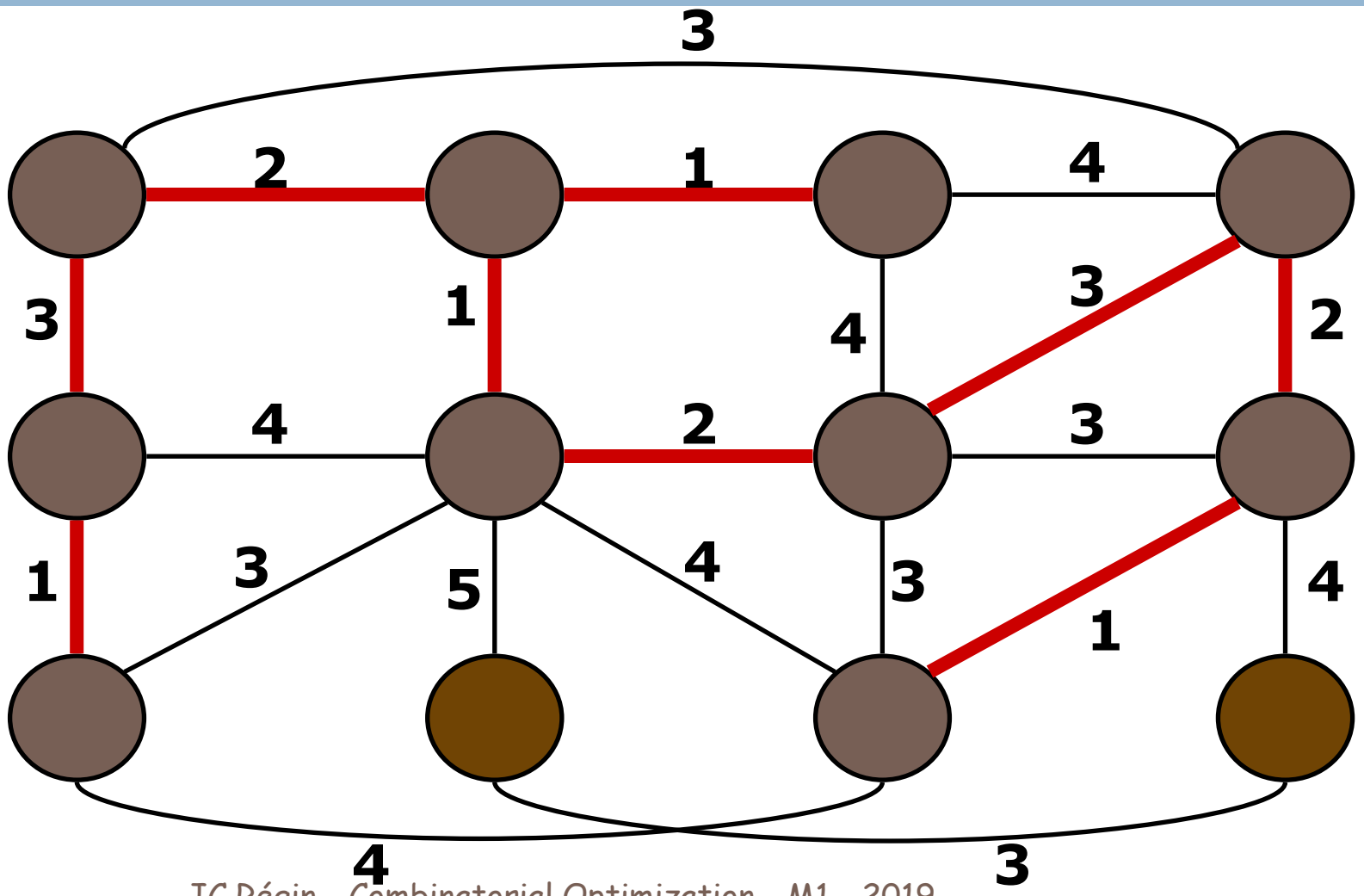
Prim's Algorithm



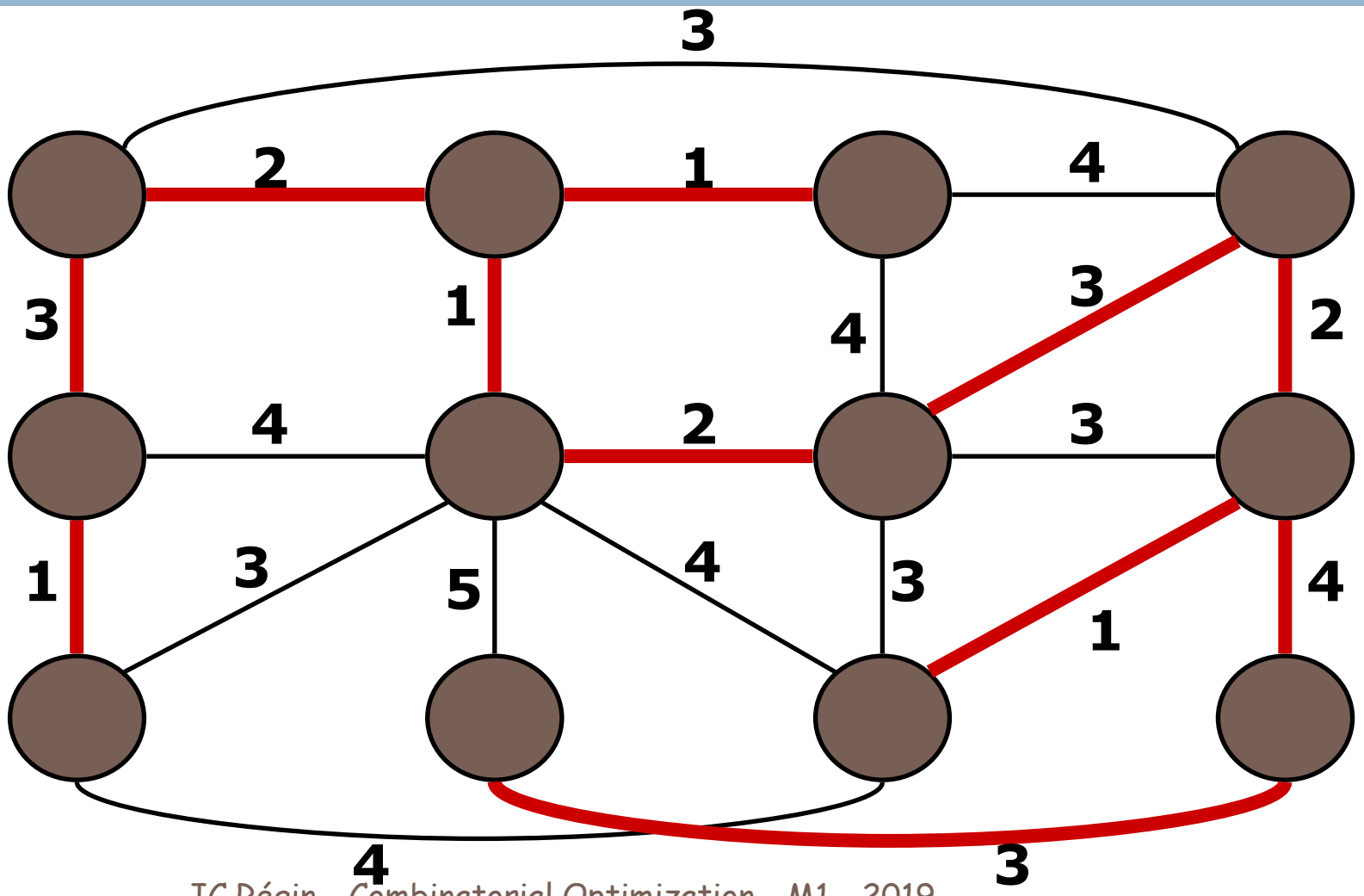
Prim's Algorithm



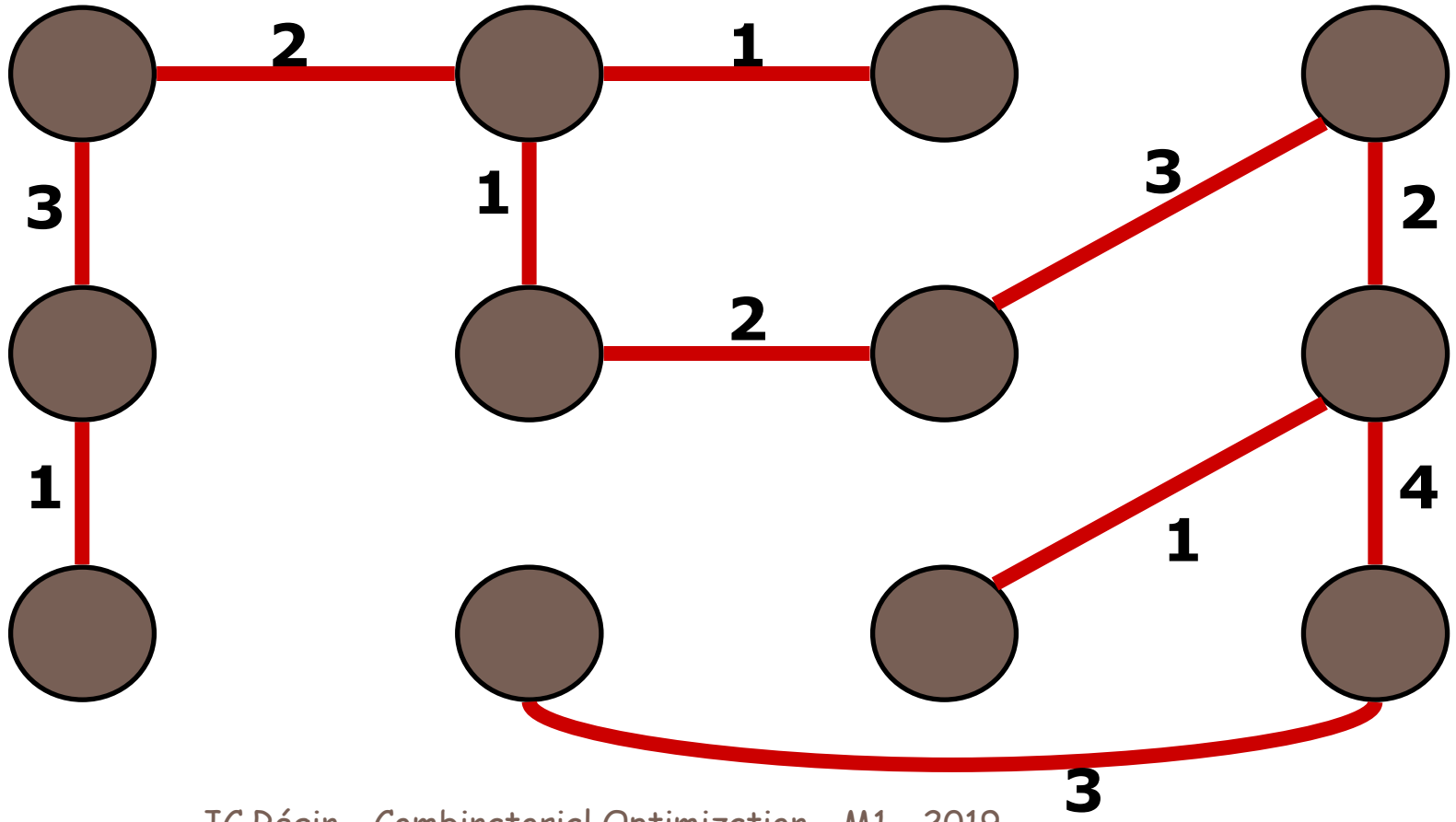
Prim's Algorithm



Prim's Algorithm



Prim's Algorithm



Prim's Greedy Algorithm

color all vertices yellow

color the root red

while there are yellow vertices

pick an edge (u,v) such that

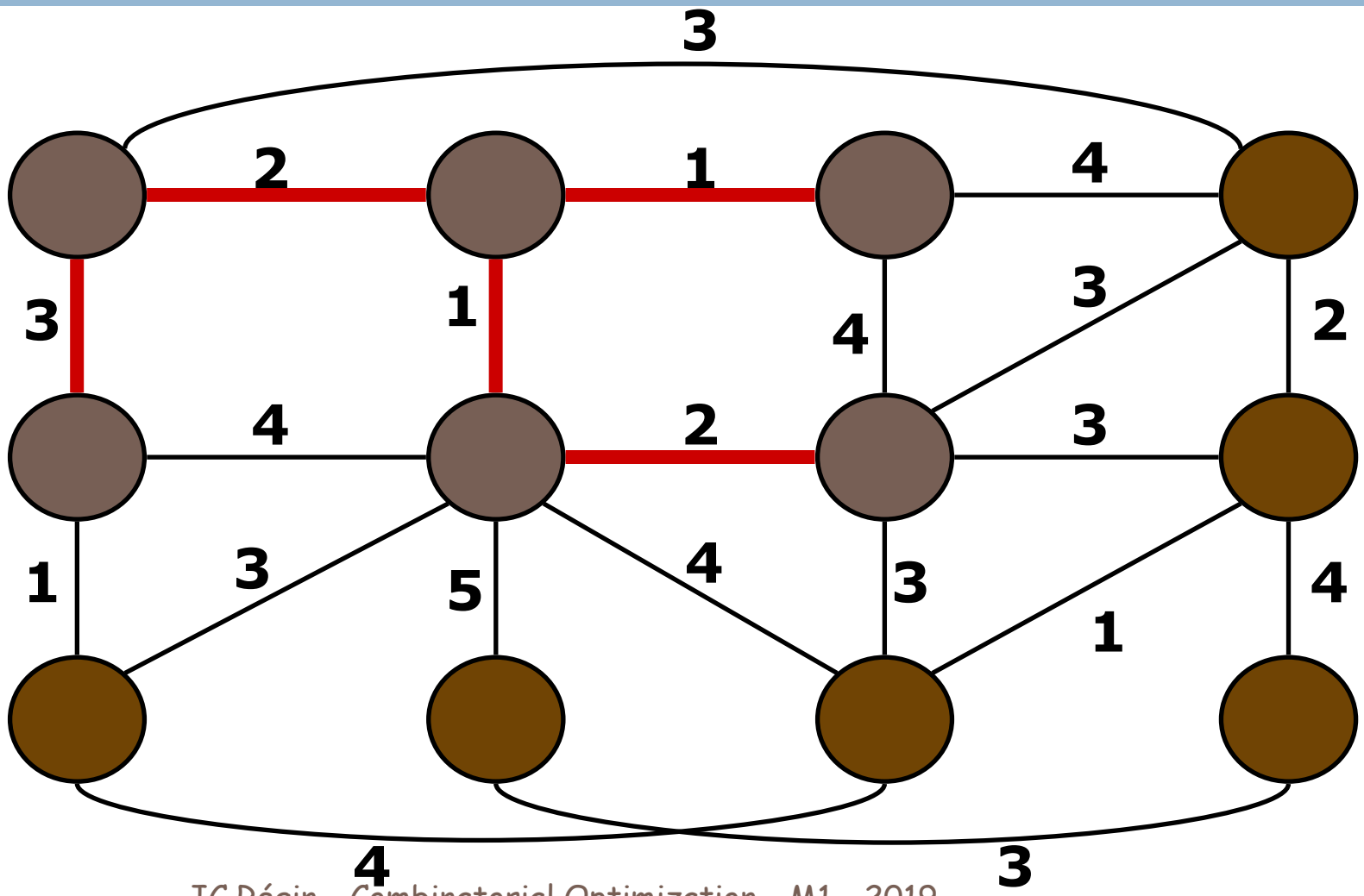
u is red, v is yellow & cost(u,v) is min

color v red

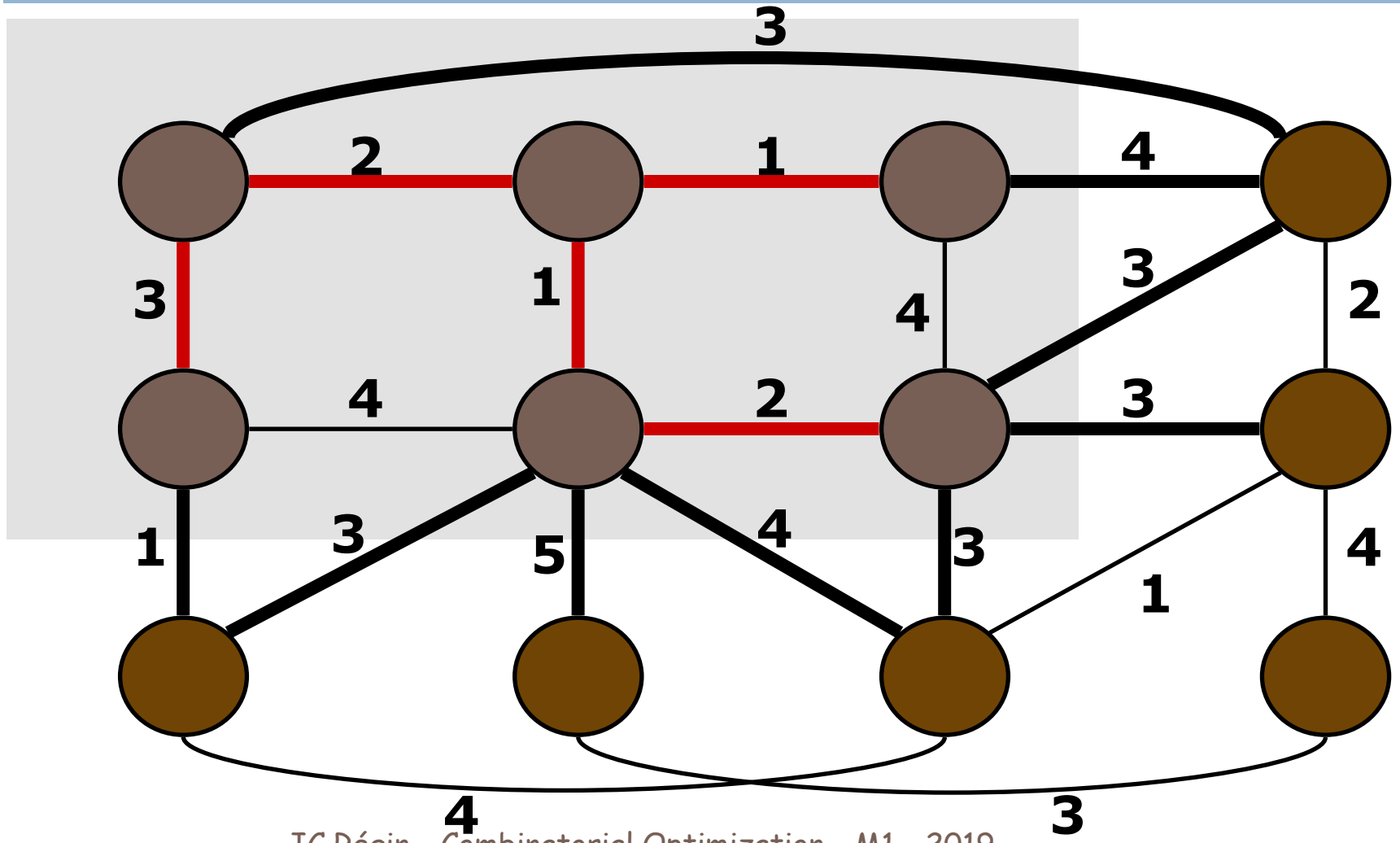
Proof

- Consider
  - SG : the solution computed by the greedy algorithm
  - Sopt : the optimal solution
- If val(SG)=val(Sopt) then ok
- Si val(SG)≠val(Sopt) then the two trees are different. We select the first edge that is different for SG and Sopt w.r.t. the greedy ordering. Let' say it is e. When this edge has been selected, it was the smallest one of the cut (V,V'). This edge link two nodes x and y. In Sopt x and y are linked by a path. On this path there is an edge f which link a node of V to a node of V'. Since the Greedy prefered e to f we have weight(e) ≤ weight(f).
- Consider the solution Sopt, and replace f by e; Sopt' is the new solution. If Sopt is optimal then we cannot improve it. So, we have val(Sopt')=val(Sopt) and so weight(e) = weight(f). Thus, we can go back to the previous point and consider a new edge with Sopt' instead of Sopt.
- At the end we will have proved that we cannot not improve Sopt and so the tree built by the Greedy algorithm has the weight and is optimal

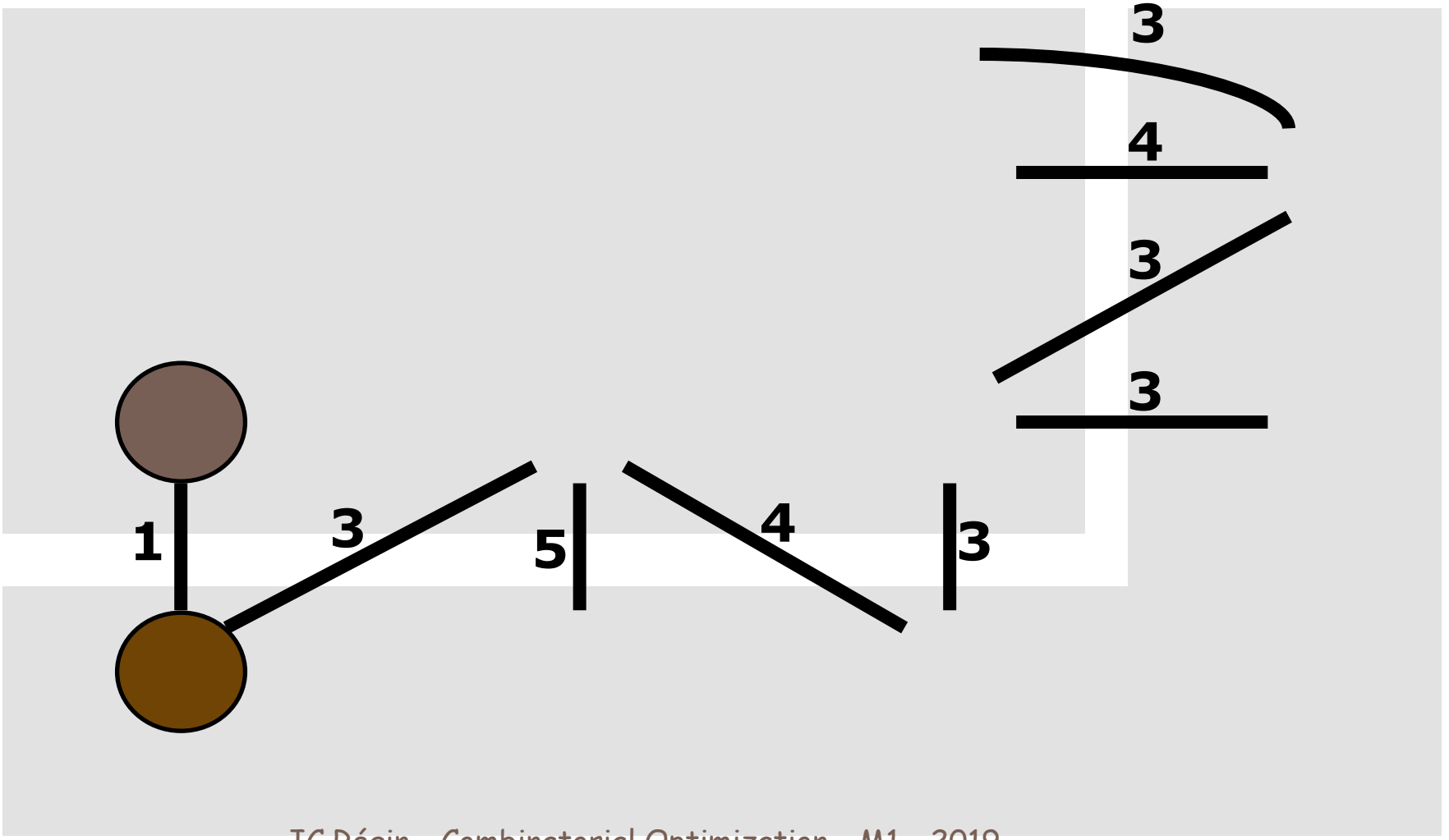
Why Greedy Works?



Why Greedy Works?



Why Greedy Works?



Prim's Algorithm


```
foreach vertex v
    v.key = ∞
root.key = 0
pq = new PriorityQueue(V) // add each vertex in the priority queue
while pq is not empty
    v = pq.deleteMin()
    foreach u in adj(v)
        if v is in pq and cost(v,u) < u.key
            pq.decreaseKey(u, cost(v,u))
```

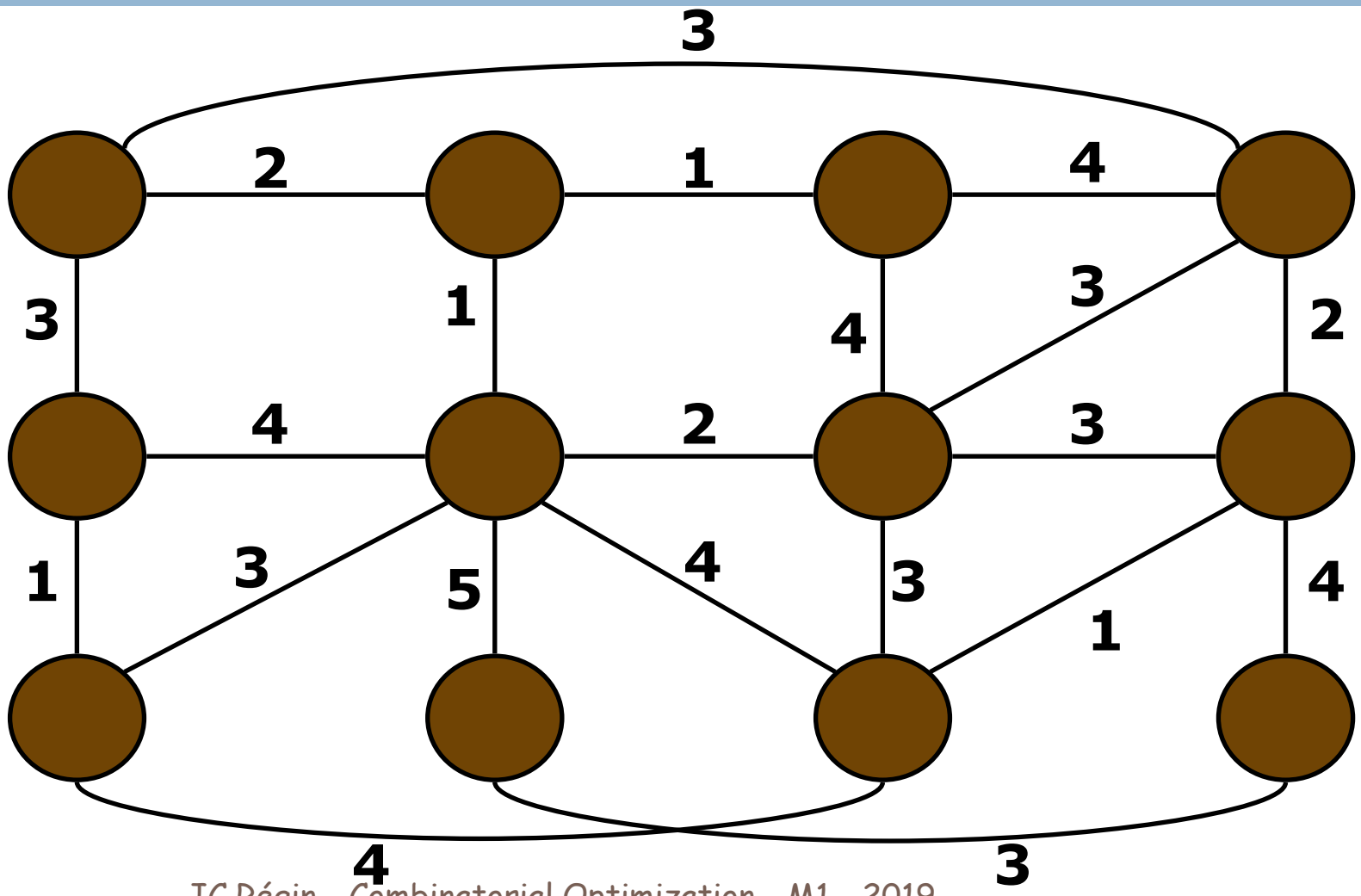
Complexity: O((V+E)log V)


```
foreach vertex v
    v.key = ∞
root.key = 0
pq = new PriorityQueue(V)
while pq is not empty
    v = pq.deleteMin()
    foreach u in adj(v)
        if u is in pq and cost(v,u) < u.key
            pq.decreaseKey(u, cost(v,u))
```

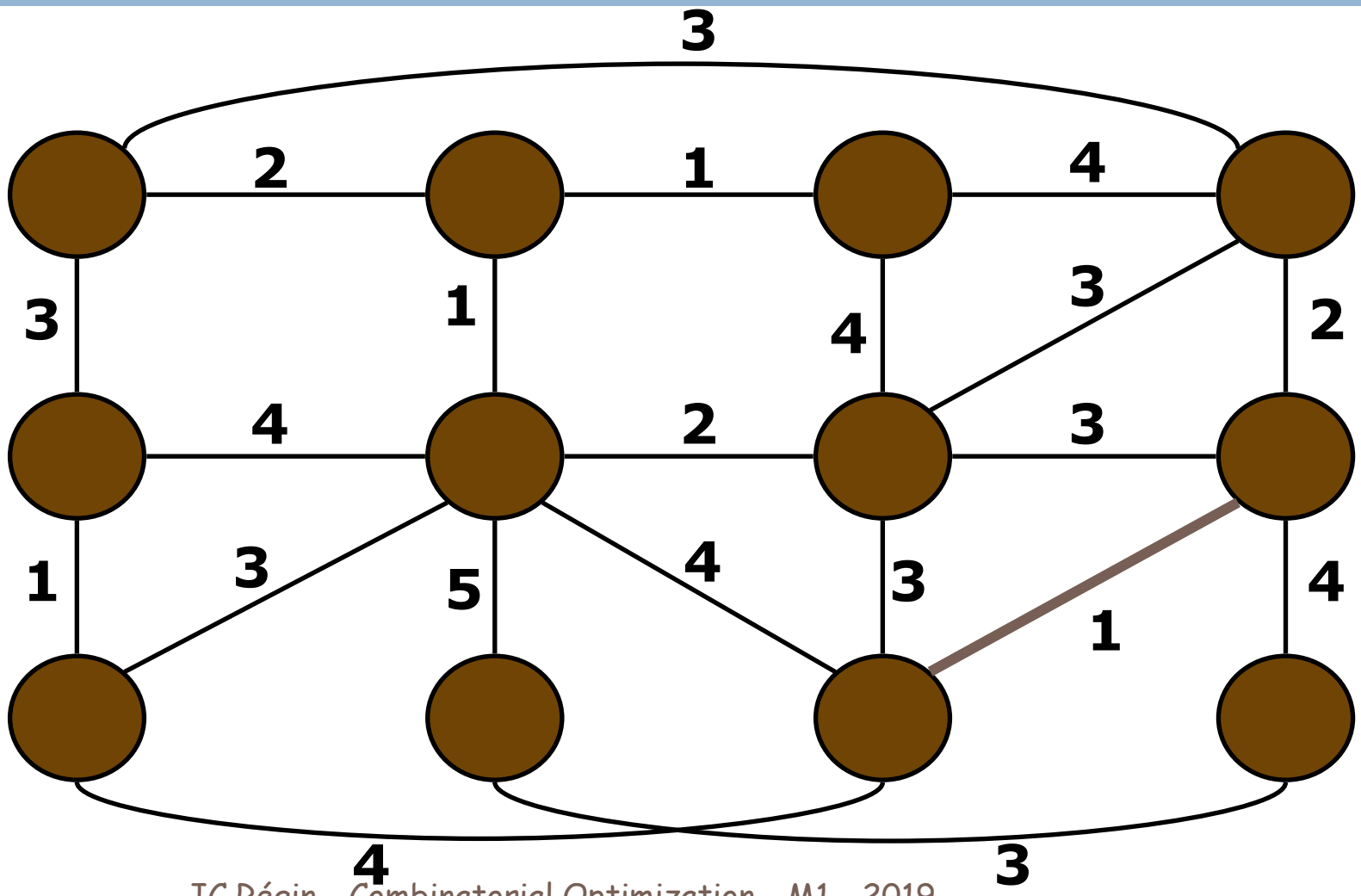
Algorithme de Kruskal

- We deal with edges.
- We traverse the edges in regards to the non decreasing order of weights
  - If an edge links two disjoints trees then we merge the tree by adding the edge to the soanning tree
  - If this is not the case we select the next edge

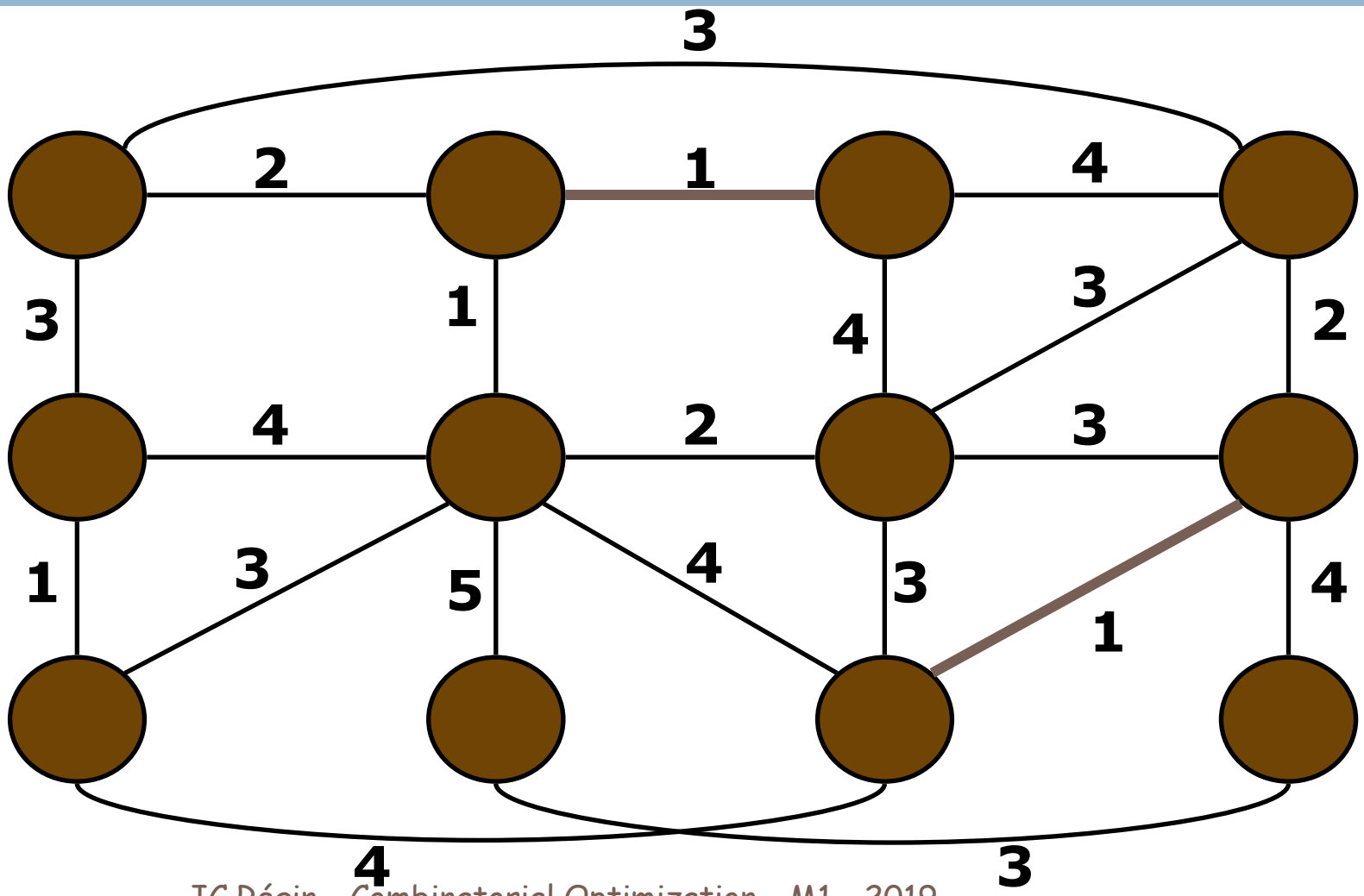
Kruskal's Algorithm



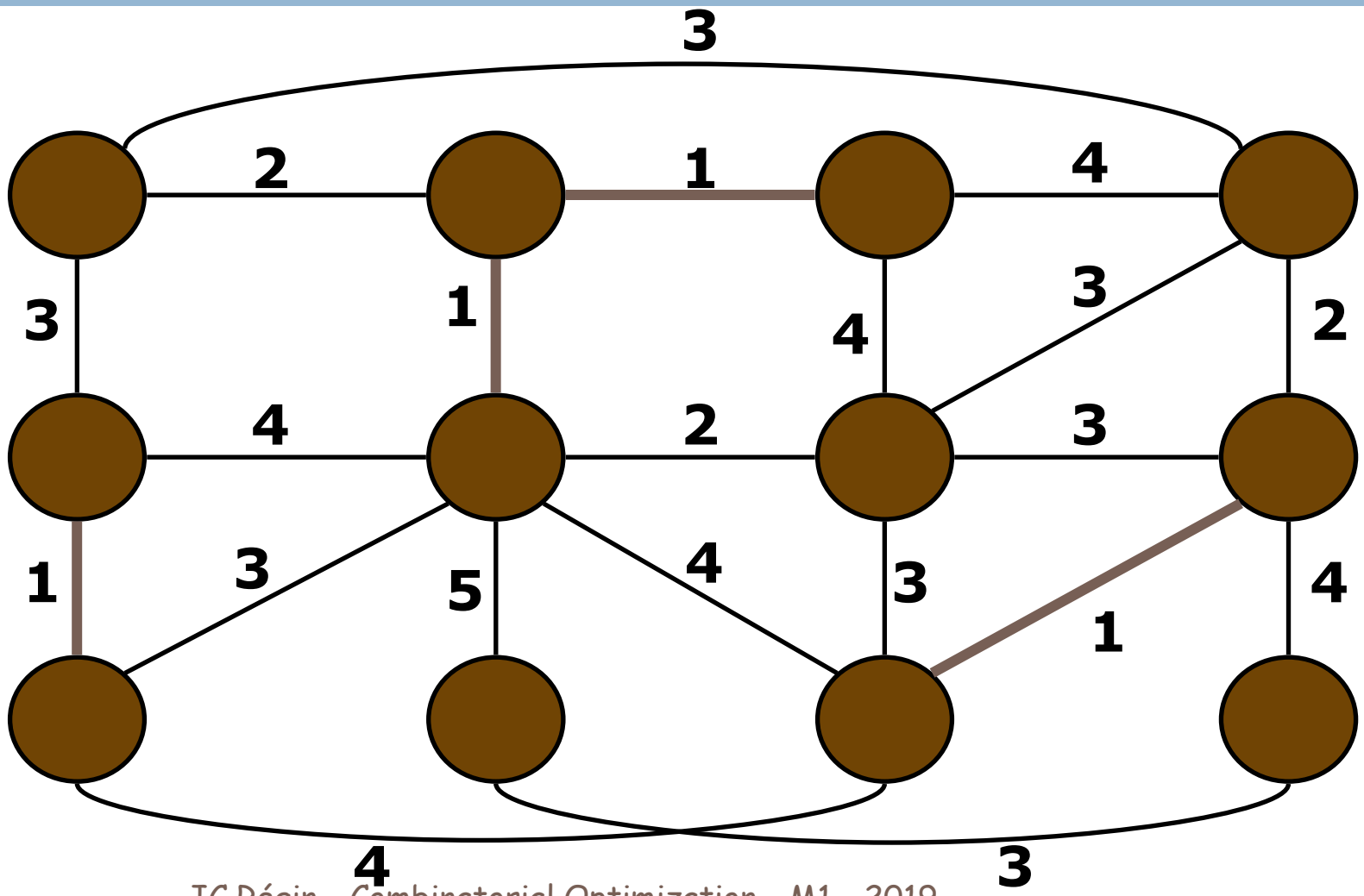
Kruskal's Algorithm



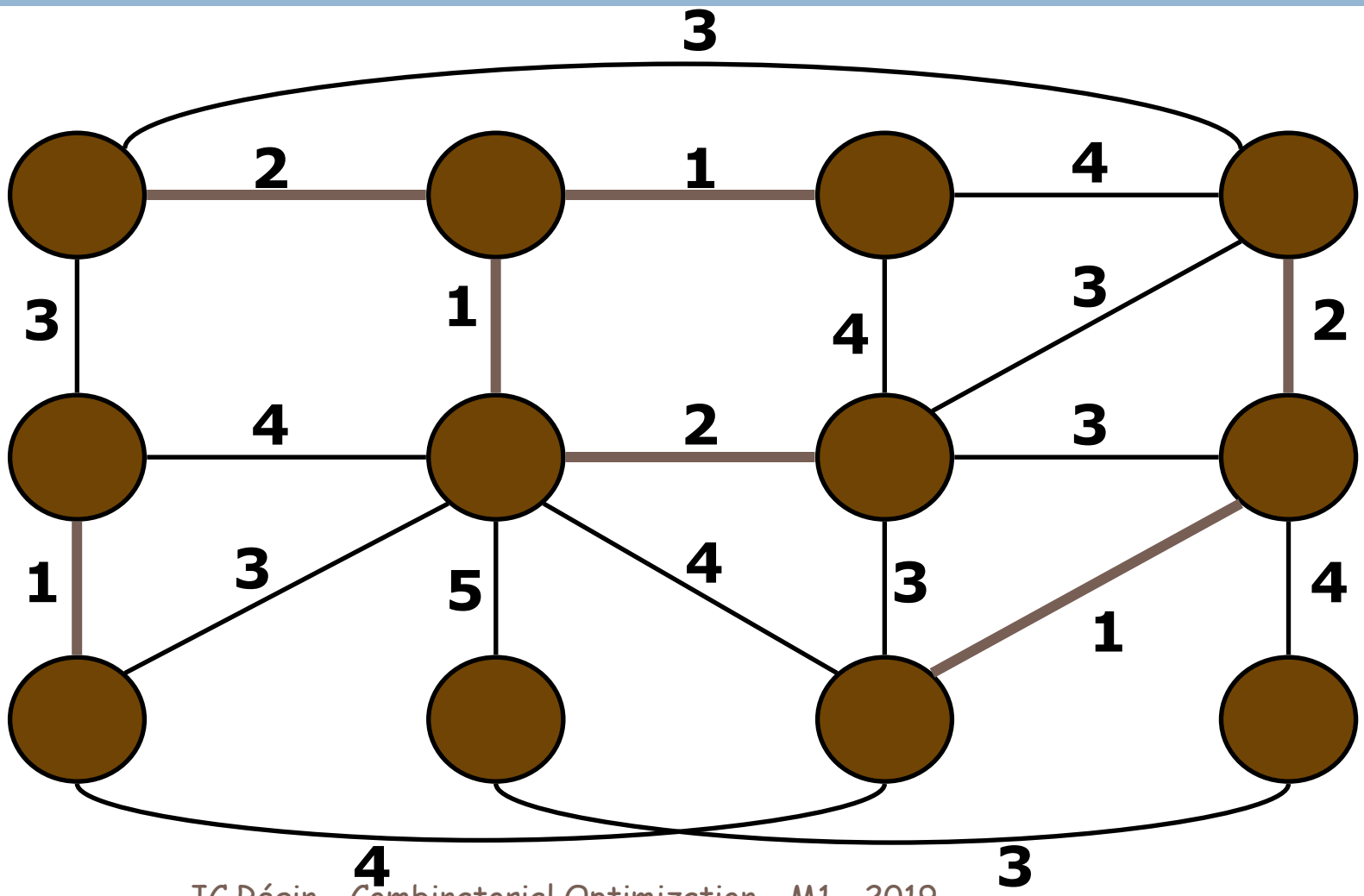
Kruskal's Algorithm



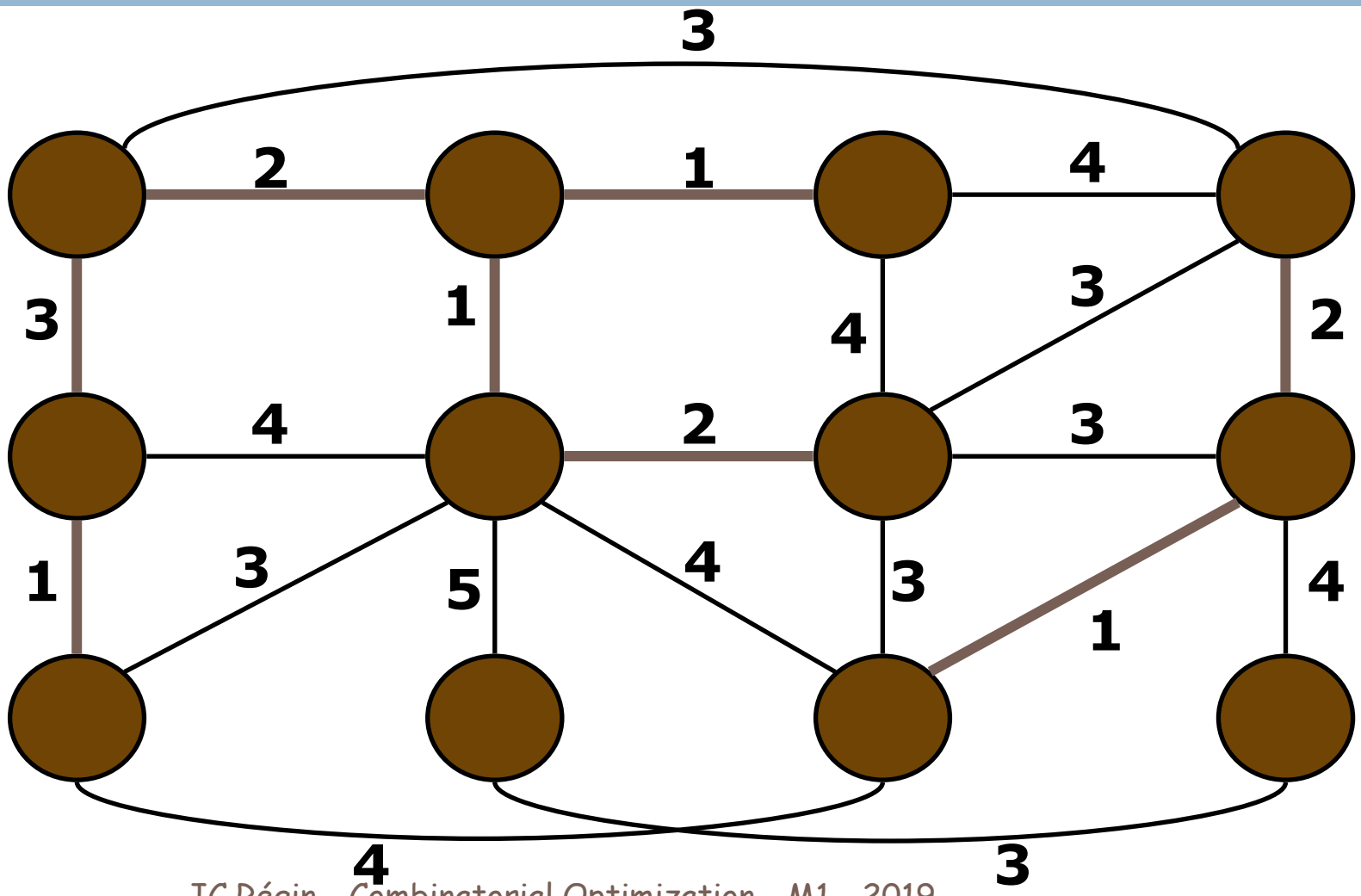
Kruskal's Algorithm



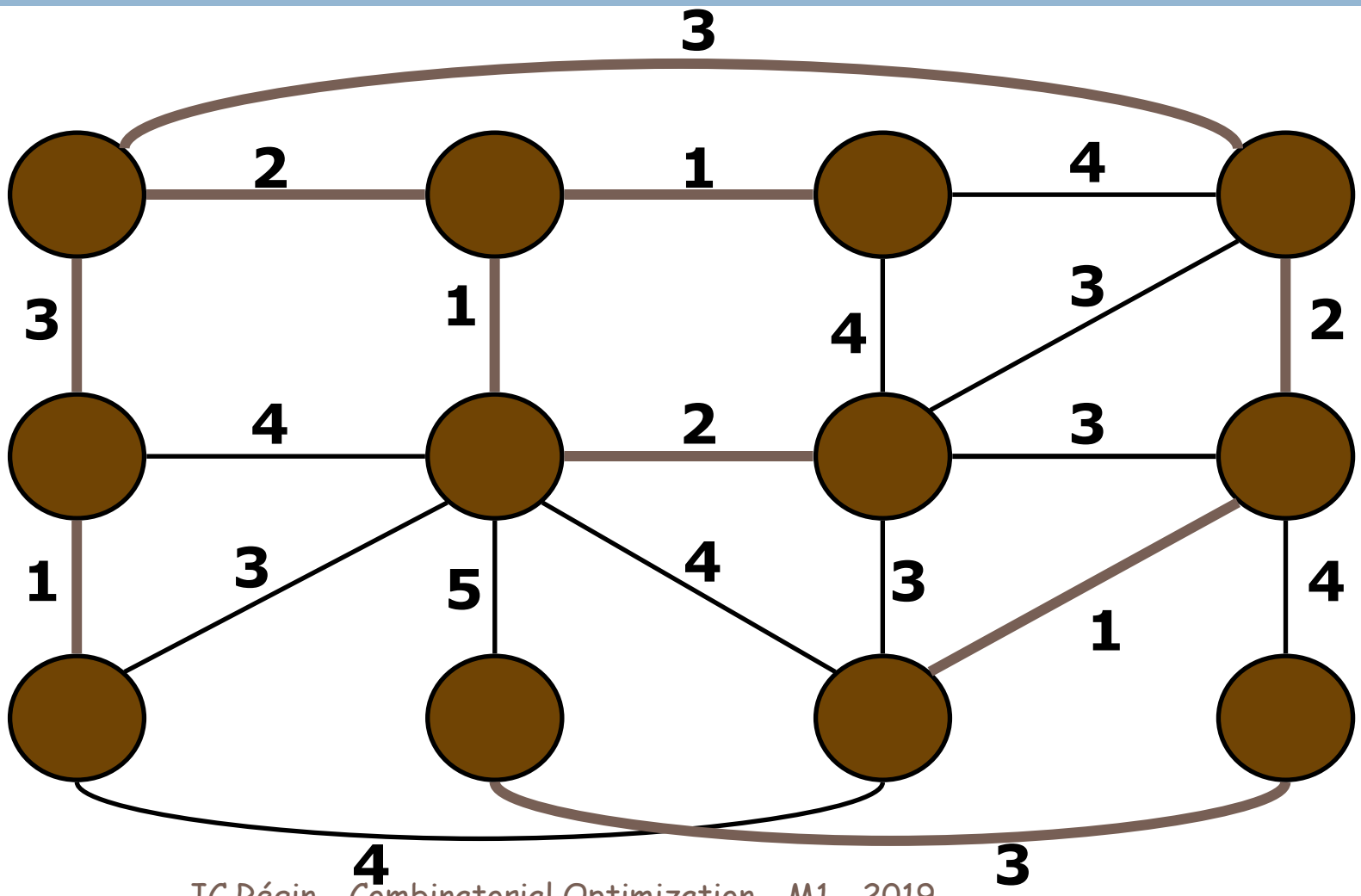
Kruskal's Algorithm



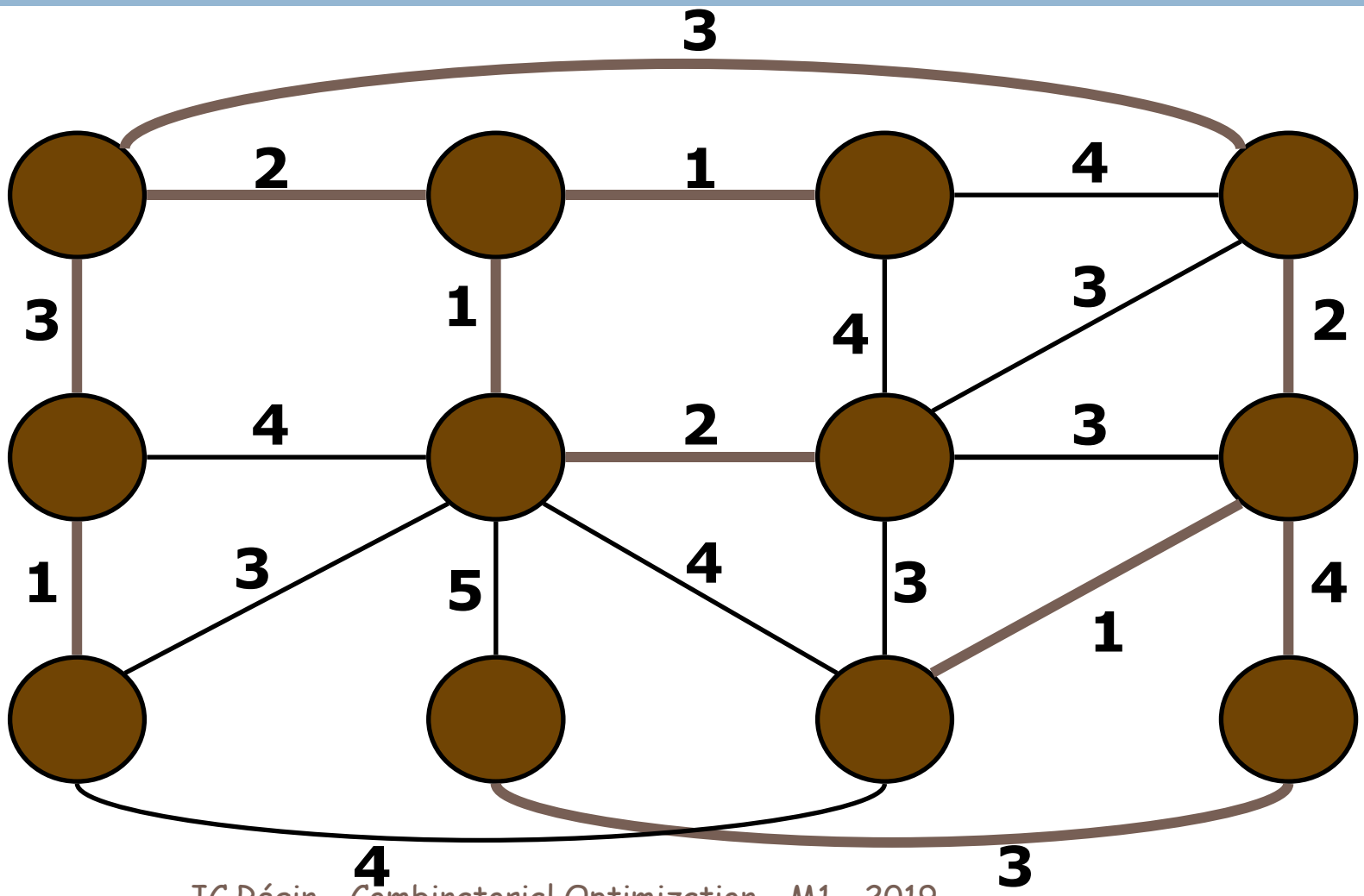
Kruskal's Algorithm



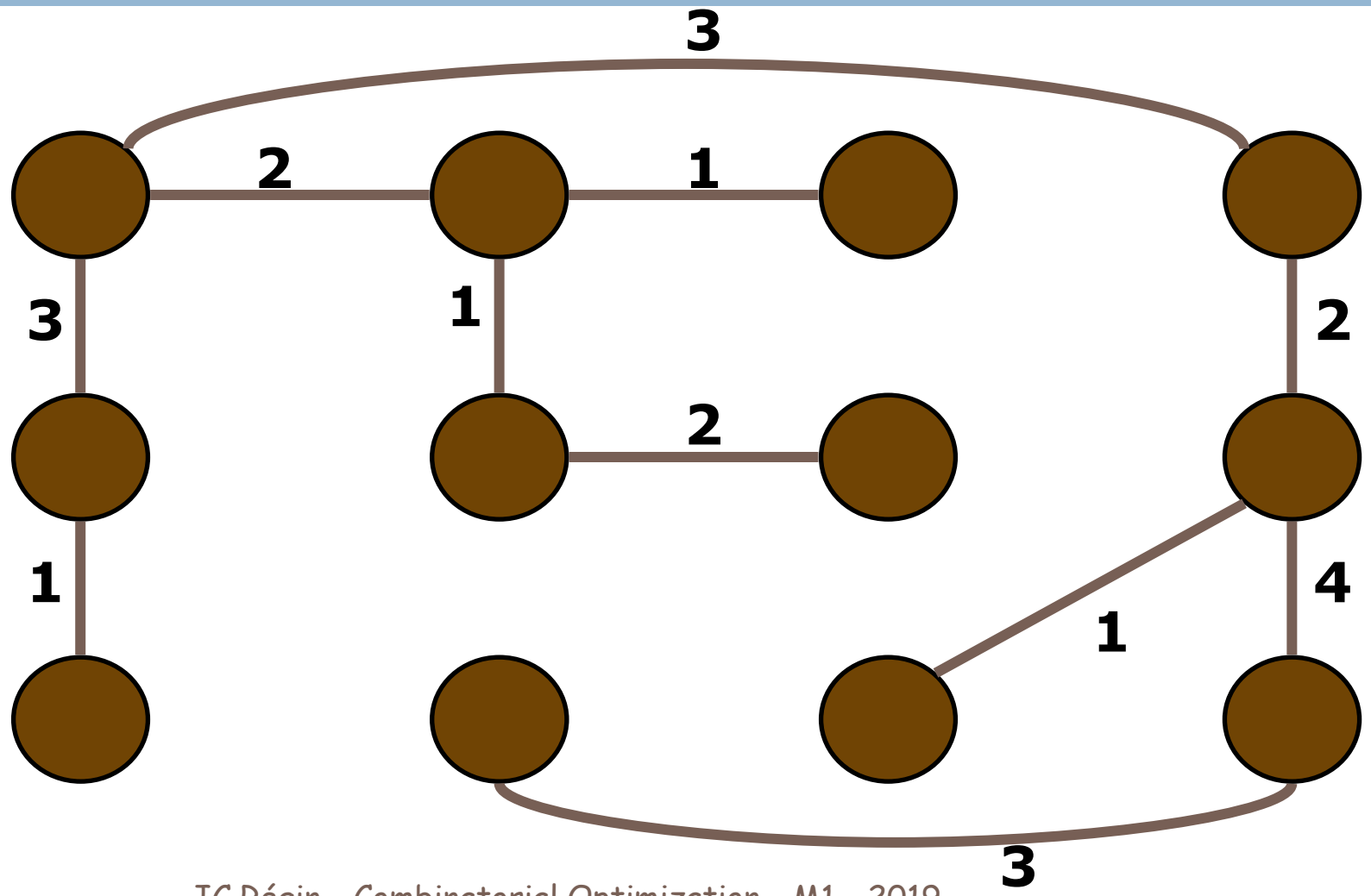
Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm



Kruskal's Algorithm

while there are unprocessed edges left

pick an edge e with minimum cost

if adding e to MST does not form a cycle

add e to MST

else

throw e away

Proof

- We effectively build a tree: clearly there is no cycle and the result is connected
- Minimality : by induction. Let's show that at each step, there is a spanning tree involving all the selected edges.
  - F set of chosen edges;
  - T spanning tree containing F;
  - e next selected edge.
- If e is in T then it is ok. Otherwise, T+e has a cycle. Consider f the edge of this cycle which is in T and not in F. T –f+e is also a spanning tree. Its weight is >= T (because T is optimal) so weight(f) <= weigh(e). Now remark that weight(f) < weight(e) is not possible because the greedy would have selectde f. So weight(f) <= weigh(e) and F+e belongs to a minimum spanning tree.

Plan

- Definition
- **The knapsack problem**
- Heuristic
- Some greedy algorithms
  - Selection of activities
  - Graph coloring
  - Covering: minimal transversal

Greedy algorithm : the knapsack

- Knapsack
  - DATA :
    - A bag, that can accept only a certain weigth
    - A set of items. Each item o_i has
      - A weight : w_i
      - A cost : c_i
  - QUESTION :
    - What are the item you should carry in order to maximize the cost of selected objects while respecting the maximum weight constraint
    - The sum of the costs of the selected items is maximal
    - The sum of the weights of the selected items is < max_weight of the bag

Knapsack : strategy

- A good strategy?
- Imagine that instead of having objects that you have to select or not, that is to take entirely or not, you could split each object into the parts you want.
  In other words, if you have speck bag whould it help you?

Knapsack: strategy

- If we have bag of dust of precious metal
- Strategy:
  - For each bag we compute the value per gram.
  - 1) We select the bag having the largest value per gram and we fill you bagpack with as many as possible of this metal
  - 2) If the weight limit is reached: we stop. Otherwise, we eliminate the bag I use and go back to 1)
- **This is an optimal strategy!**
- **Note that only one bag is not entirely selected.**

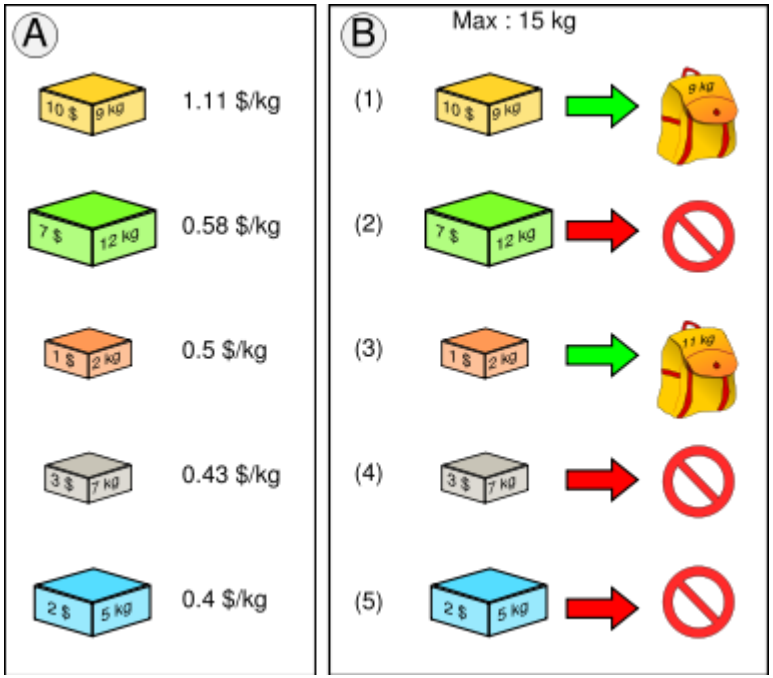
Knapsack: strategy

- Optimality proof (by contradiction):
- The efficiency of an item is its cost per gram
- Assume we have an optimal solution such that it does not take a gram of o_i but takes one gram of o_k whose efficiency is less than o_i.
- In this case: swapping 1g of o_k and 1g of o_i will improve the solution. Therefore it is not optimal.

Knapsack: optimality

- Having bag of dust is equivalent to accept to cut items
- If we cannot but items, how can we deal with that?
  - The problem becomes difficult
  - We can do « as if »
    - We compute the efficiency and select the objects in regards to their efficiency while satisfying the weight constraint.

Knapsack: strategy



Solution : 1 + 3, cost $11

Better solution 1 + 5, cost $12

Knapsack: model

- How can we represent this problem?
- What are the variables?
- How can we express constraints?

- This is modeling: mathematical representation of a problem. We define this problem

Knapsack: the variables

- We assocaite each item with a 0-1 variable (variable that can take only 0 or 1)
  - It is a « belonging » variable of the bag
  - If the item is selected then its value is 1else it is 0
- The cost of an item and its weight are data. Thus for the item o_i we have a cost c_i and a weight w_i.
- The belonging var is x_i
- The weight limit of the bag is W

Knapsack: the constraints

$$\max \sum_{i=1}^{n} c_i x_i$$

The objective

$$\sum_{i=1}^{n} w_i x_i \leq W$$

The sum of the weights of the selected items
Must be less than or equal to the maximal weigth

Plan

- Definition
- The knapsack problem
- **Heuristic**
- Some greedy algorithms
  - Selection of activities
  - Graph coloring
  - Covering: minimal transversal

Greedy heuristic

- When the greedy algorithm does not compute the optimal solution we say that it is a **greedy heuristic**

Heuristic

- Comes from the acient greek eurisko « I find »
- A **heuristic** is an algorithm which gives quickly (in polynomial time) a feasible solution, not necessary optimal, for a given NP-Complete problem
- A heuristic is an **approximative method** which does not give systematically the best solution

Heuristic

- Often we speak about a **heuristic method**
- Usually, a heuristic is dedicated to a specific problem by using its particular structure.

Heuristic Evaluation

- **Practical criteria or empirical:** we implemnt the algorithm and we evaluate the quality of its solutions according to the optimal solution (or to the best known solutions). A benchmark (set of instances) is defined.
- **Mathématical criteria:** we need to prove the performance of the heuristic. The best guarantee is usually the one coming from approximate algorithm. Probabilist guarantee are also quit interesting, mainly when the heuristic does not give the best solutions.

Heuristic Evaluation

- Empirical and mathematical criteria may be in contradiction.
- Often, the best mathematical guarantee is not interesting in practice.

Heuristic

- The exact algorithm have an exponential complexity
- It can be inetresting and worthwhile to combine them with heuristic methods for solving difficult problems.
- We can combine both methods
  - We can use the heuristic method for computing an approximate solution and speed up the whole process of resolution.

Plan

- Definition
- The knapsack problem
- Heuristic
- **Some greedy algorithms**
  - Selection of activities
  - Graph coloring
  - Covering: minimal transversal

Choice of activities

- Consider a gymnasium that is used by some activities. We would like to use it for the maximum of activitites. The constraint is that we can have only one activity at any moment.
- An event i is characterized by a start date s_i and an end date e_i.
- Two events are compatible if their time interval are disjoint

Choice of activities

- How to place the maximum number of events?
- Any idea?

Choice of activities

- Strategy 1 :
  - We sort the events by **duration**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

Choice of activities

- Strategy 1 :
  - We sort the events by **duration**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

Choice of activities

- Strategy 2 :
  - We sort the events by **start date**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

Choice of activities

- Strategy 2 :
  - We sort the events by **start date**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

Choice of activities

- Strategy 3 :
  - We sort the events by increasing **end date**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

Choice of activities

- Strategy 3 :
  - We sort the events by increasing **end date**
  - We set first the shortest one if there are compatible with the ones that have been already selected.

- This is optimal: we can prove it!

Choice of activities

- Strategy 3 :
  - We sort the events by increasing **end date**
  - We set first the shortest one if there are compatible with the ones that have been already selected.
- Let f1 the element having the smallest end date. We show that there is a solution including it.
- Consider any optimal solution O = { fi1 , fi2 , ..., fik } and k be the maximum of events that can be organized in the gymnasium.
  There are 2 possibilities either fi1 = f1 or fi1 ≠ f1.
  - If fi1 = f1 then f1 can replace fi1
  - If fi1 ≠ f1 then we replace fi1 by f1. Since f1 ends before any other event, then since fi2 did not intersect with fi1 then it does not intersect with f1. So we can find an optimal solution containing f1
- We can repeat the process for the event that do not intersect with f1 and repeat the proof by induction.

Graph Coloring

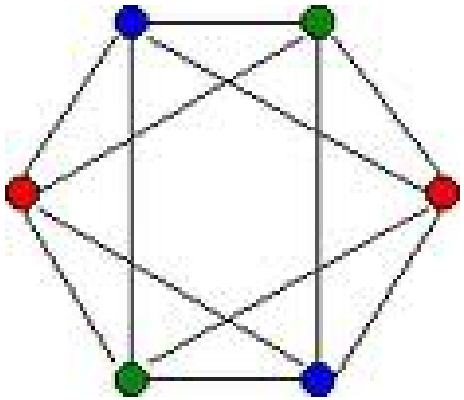
- We want to color the nodes such that two adjacent nodes have a different color.
- Find the minimum number of colors
- Difficult problem
- Arise frequently in practice. Graphs express relations between objects
- Edge = the objects must be managed in different ways
  - Register Allocation (color = register)
  - Boarding Gate Allocation (color = gate, node = aircraft, edge = at the same time)
  - Frequency Allocation (edge = interference between objets, color = frequency)

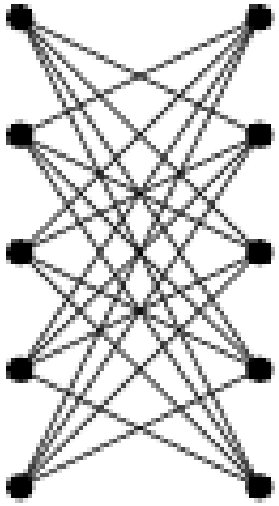
Bipartite graph coloring

- Bipartite graph
  - 2 colorable graph
  - Has no odd cycle
- Propose 2 algorithms for detecting whether a Graph is bipartite or not (and give its parts if it is bipartite)
- Adapt your algorithm to give a proof (i.e. an odd cycle) of non bipartition

Graph Coloring

- The degree of a node is the number of edges it belongs
- Δ(G) is the maximal degree
- Be careful a graph may be color with less than Δ(G) colors !





Graph Coloring

- Strategy 1 :
  - Select randomly the nodes
  - **Assign them the smallest possible value**
- Let K be the number of used colors. Then $K \leq \Delta(G) + 1$
  - Proof: when a node is considered, it has at most $\Delta(G)$ neighbors already colored, thus the algorithm will never use more than $\Delta(G) + 1$ color.
- For a clique (complete graph) this is optimal
- For a bipartite graph, this is a bad algorithm

Graph Coloring

- Strategy 2 :
  - Order the nodes by non decreasing degree
  - **Select the nodes along that order and assign them the smallest value.**
- Consider first nodes having the largest degree, so we can decrease the boundary
- We do not know when we should stop selecting nodes having large degree, so we sort the nodes
- Bad results with bipartite graph

Graph Coloring

- Strategy 2 :
  - Order the nodes by non decreasing degree
  - **Select the nodes along that order and assign them the smallest value.**
- n=|V|, di degree of node vi. We prove that
  $K \leq \max_{1 \leq i \leq n} (\min(di + 1, i))$.
  - When we color the node vi , there are at most min(di, i − 1) neighbors that have been colored, and so the color of vi is at most1 + min(di, i − 1) = min(di + 1, i). By taking the maximum for i of these value we obtain the boundary.

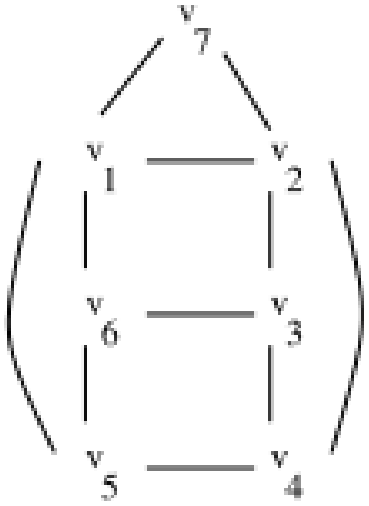
Graph Coloring

- The intuitive idea is to color first the nodes having a lot of neighbors that have already been colored
  - We define the color-degree of a node as its number of already colored neighbors.
  - The color-degree evolves during the algorithm. It is initialized at 0
- Strategy 3 (**Brelaz's algorithm: DSATUR**) :
  - **Select a node having a color-degree maximal. Break the tie by the degree. Assign the smallest possible value.**

Brelaz's Strategy

- This strategy is a general one: select the object having the minimum degree of liberty

Brelaz's Strategy



1) v1 is the first one. It has color 1

2) The color-degree of v2 , v5 , v6 and v7 is 1, v2 is selected (max degree). It has the color 2.

3) Now, v7 is th eonly one node having a color-degree equals to 2. We select it and it got the color 3.

4) All remainign nodes ahve the same color-degree (1) and the same degree (3). We select v3 (randomly) its color is 1

5) v4, color-degree 2, has color 3

6) v5 has color 2 and v6 color 3
The graph is colored with 3 colors. It is optimal.

Brelaz's Strategy

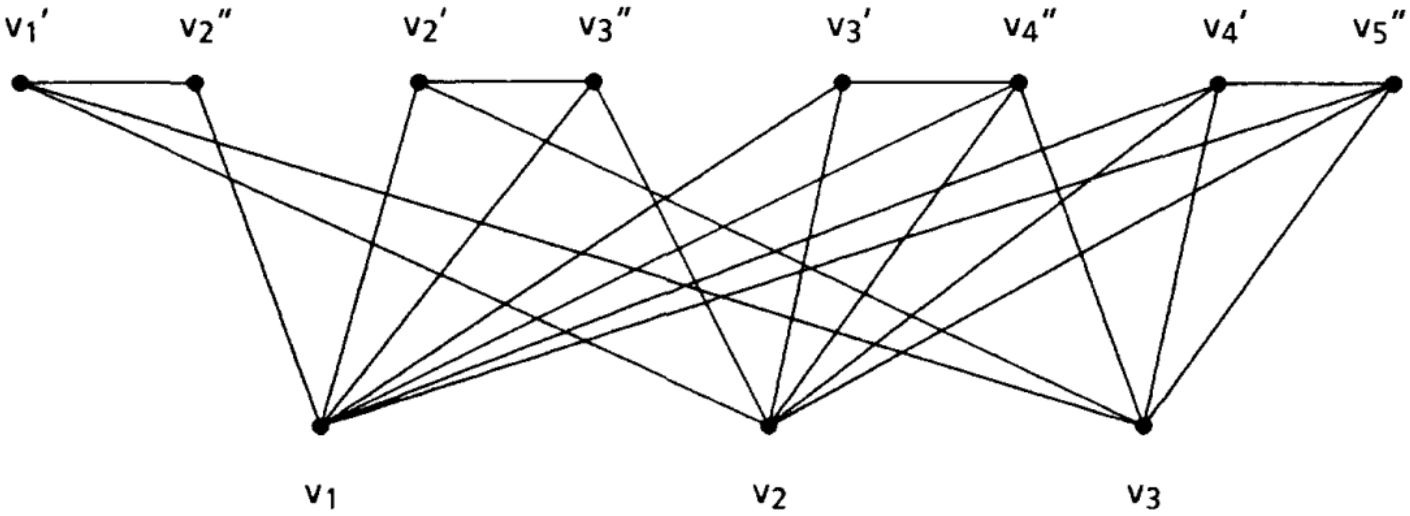


Fig. 1.

Order : v1', v2", v1, v2', v3", v2, v3', v4", v3, v4', v5"

Coloring ?

Brelaz's Strategy

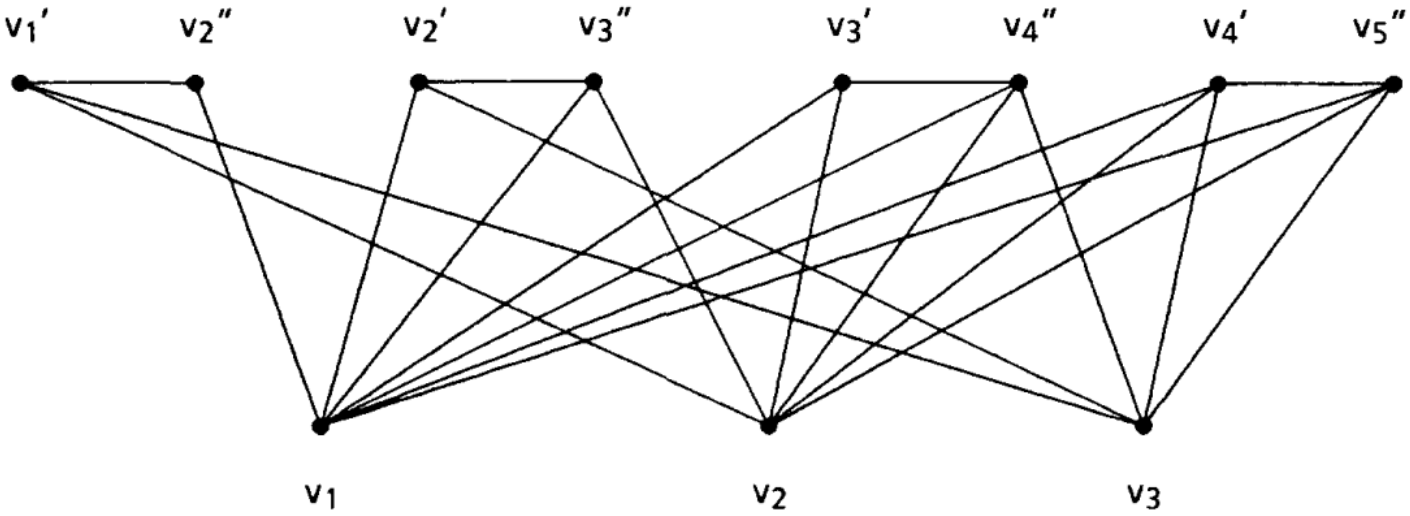


Fig. 1.

Order : v1', v2", v1, v2', v3", v2, v3', v4", v3, v4', v5"

vi, vi' and vi" have the same color

Brelaz's Strategy

- Brelaz's stratégy may color a graph with n/3 colors whereas there is a solution with only 3 colors.
- Mathematical consequence?
  - We CANNOT guarantee ANY boundary

Static or dynamic strategy

- A **strategy is static** if the order along with the objects are selected at each step is defined a priori.
  - Choice of the largest degree
- A **strategy is dynamic** if the order along with the objects are selected at each step depends on the previous choices, that is will be recomputed at each step.
  - Brelaz's strategy

Static or dynamic strategy

- With a dynamic strategy we cannot know in advance the order along with the nodes will be selected
- In general, the dynamic strategy give better results than static ones.

Vertex cover

- A **vertex cover** is a set of nodes such that each edge has an endpoint in this set.

- The **minimum vertex cover problem** is the optimization problem:
  - DATA: Graph G
  - QUESTION: The smallest number k such that G has a vertex cover of size k
- Décision problem (**vertex cover problem**) :
  - DATA: Graph G and a positive integer k
  - QUESTION: Does G contain a vertex cover of size k?

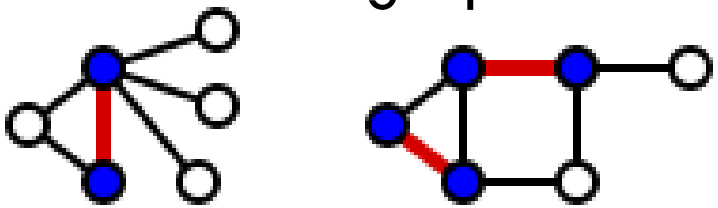
Vertex cover

- If a set of nodes S is a vertex cover then its complement is an independant set (stable set).
- A graph having n nodes admits a vertex cover of size k if and only if it has a stable set of size n-k.
- A stable set in G is a clique in the complement graph of G. The minimum stable set is equivalent to the maximum clique problem in the complement graph

- Hitting set = vertex cover in an hyper graph
- Problem arising frequently in practice: we have set of non disjoints groups; we want a representant per group and minimize the number of representants

Greedy strategy

- There exists an algorithm finding a 2-approximation of the problem (the solution is at most twice smaller). We repeat the 2 following operations while there are some edges:
  - We take an edge, we put its two endpoints in the vertex cover
  - We remove these 2 nodes from the graph



En bleu le transversal

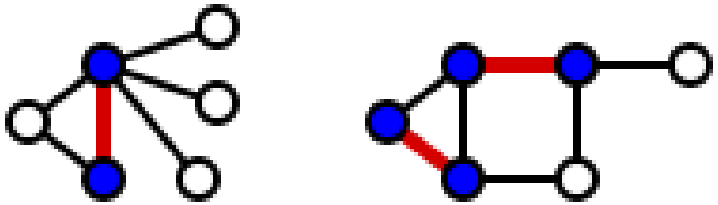
Greedy strategy

- We can prove that the error is at most a factor of 2.
- First, we show that the greedy works
  - It is clear that there is no edge at the edge, therefore all edges are covered and the computed set is a vertex cover.
- Second, we show that the optimal solution must contain at least half of the nodes.
  - When a node is selected, then at least one of its extremity must be selected ; other wise the edge would not be covered. It is true for all vertex cover, even the optimal one. Thus, we have to select at least one of the two endpoints that we selected. The best solution contain at least half of the nodes of the computed solution because the greedy algorithm take both endpoints.

Another strategy

- Heuristic of largest degrees.
  - At each iteration we select the node which covers the maximum numbers of edges.

- Usually better in practice



Another strategy

- This heuristic may give solution that are as bad as we want: for each $\rho > 1$ we can build an instance for which the heuristic gives a solution whose cost is greater than ρ times the optimal one.
- Eg: this strategy can be very bad for a bipartite graph (in this case the problem is polynomial)

A third strategy

- Savage: perform a DFS and keep only the nodes that are not leaves.
- We can show that the optimal cost is at most twice the computed cost.