EPITECH.
PARIS GRADUATE SCHOOL OF DIGITAL INNOVATION

Introduction to Algorithms

Part II. Problems.

B9 - Introduction to Algorithms

M-ALG-100

Overview of the module

- Day 1 Concept of algorithm, Cryptography, recursion, Knapsack, Shortest Path
- Day 2 Complexity, Graph problems, Theory

Organisation

- The exercices will be in python
- Please clone the following repository : `https://github.com/nlehir/ALGO1.git`
- Third party libs : **matplotlib**, **numpy**, **networkx**
- Optional but useful : **ipdb** (python debug) or another debugger

Day 2

Complexity

- Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question, as a function of the size of the input ? Is it possible to **compute** an answer with a computer ?

Complexity

- Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question, as a function of the size of the input ? Is it possible to **compute** an answer with a computer ?
- Importantly, this is called the **time complexity** of the problem. It does not take the memory usage into account.

Complexity

- Today we will **quantify** the **complexity** of several problems : how many operations are required to answer a given question, as a function of the size of the input ? Is it possible to **compute** an answer with a computer ?
- Importantly, this is called the **time complexity** of the problem. It does not take the memory usage into account.
- However, we will also discuss **space complexity** that quantifies memory usage.

Complexity

- The answer is that **it depends on the problem**. For some problems, it is very probable that there exists **no exact fast** solution (for instance the NP-hard problems)

Average and worst case complexities

- Often, for a given algorithm, the exact number of operations needed will **depend on the instance of the problem**.
- It is possible to compute several complexities given a problem size n:
  - **worst-case** the maximum number of operation needed
  - **average-case** average complexity, averaged over a **distribution** on the input. Thus this distribution is to be known, or assumed.

Measuring complexities

- Let us measure the time complexity of some simple programs. How ?

Measuring complexities

- Let us start by measuring the complexity of some simple programs.
- We can first measure the computing time.

Measuring execution times

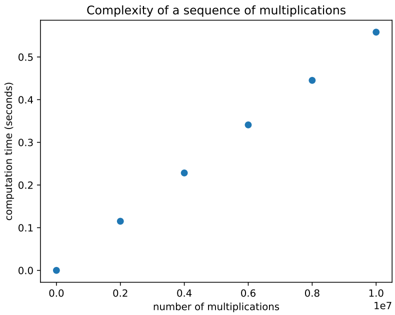
Exercice 1 : Linear complexity

- **cd complexity/** and use **linear_complexity.py** to verify that the complexity of a sequence of multiplications is proportionnal to its length.

Measuring execution times

Exercice 1 : Linear complexity

- **cd complexity/** and use **linear_complexity.py** to verify that the complexity of a sequence of multiplications is proportionnal to its length.
- It should look like this :



Measuring execution times

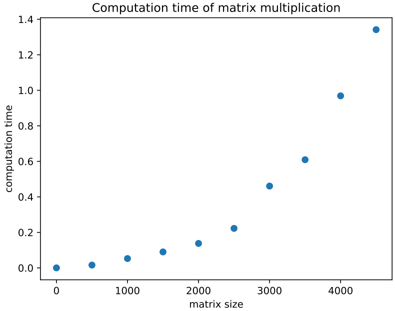
Exercice 2 : Non linear complexity

- What happens with matrix multiplication ? modify **matrix_multiplication.py** to estimate the computing time as a function of the size of the matrix.

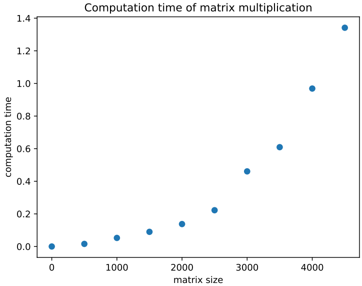
Measuring execution times

Exercice 2 : Non linear complexity

- What happens with matrix multiplication ? modify **matrix_multiplication.py** to estimate the computing time as a function of the size of the matrix.
- It should look like this :

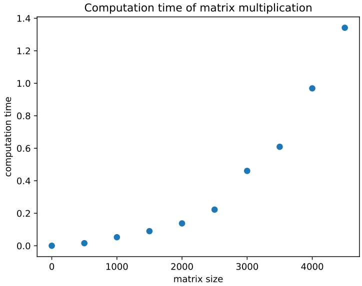


Matrix multiplication

- 

- Let's give a rough approximation of the number of operations as a function of the size n of the matrix.

Matrix multiplication

- 

- Let's give a rough approximation of the number of operations as a function of the size n of the matrix.
- It should then be of order $\mathcal{O}(n^3)$. Remark: However, some **sub-cubic** algorithms exists : faster than n^3

Measuring the time ?

- Why is **time** maybe not the best tool to evaluate the complexity of an algorithm ?

Measuring the time ?

- Why is **time** maybe not the best tool to evaluate the complexity of an algorithm ?
- It depends on the machine

Measuring the time ?

- Why is **time** maybe not the best tool to evaluate the complexity of an algorithm ?
- It depends on the machine
- We could count the number of elementary operations instead.

Experimental evaluation

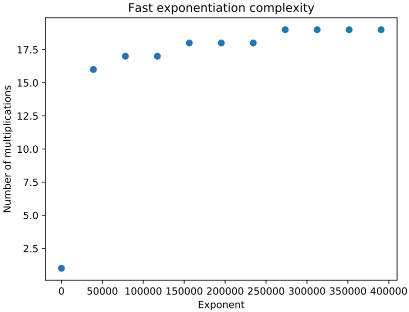
Exercice 3 : Counting the number of elementary operations

- Please use a variable in **exponentiation_complexity.py** to compute the number of operations in fast exponentiation and normal exponentiation.

Experimental evaluation

Exercice 3 : Counting the number of elementary operations

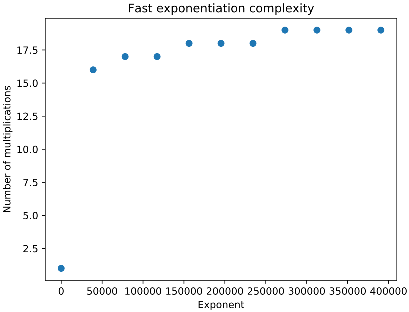
- Please use a variable in **exponentiation_complexity.py** to compute the number of operations in fast exponentiation and normal exponentiation.
- It should look like :



Experimental evaluation

Exercice 3 : Counting the number of elementary operations

- We note the **logarithmic complexity** $\mathcal{O}(\log n)$



Asymptotic behavior

- What matters is the **asymptotic** behavior, when $n \to \infty$

Asymptotic behavior

- What matters is the **asymptotic** behavior, when $n \to \infty$
- This tells if the algorithm **scales** (still works when the instance of the problem is larger)

Asymtptic behavior : $\mathcal{O}$ notation (notation de Landau)

- Mathematically speaking, we say that $f = \mathcal{O}(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ is **bounded**.

$$\exists A \geq 0, \forall n \in \mathbb{N} |\frac{f(n)}{g(n)}| \leq A \tag{1}$$

- $||$ means "absolute value"
- intuitively, this means that f is not bigger than g

Asymptotic behavior : examples

- $$n^2 + n = \mathcal{O}(?) \tag{2}$$

- $$5 \times n^4 + 2178 \times n^3 + \log 3n = \mathcal{O}(?) \tag{3}$$

Asymptotic behavior : examples

- $$n^2 + n = \mathcal{O}(n^2) \tag{4}$$

- $$5 \times n^4 + 2178 \times n^3 + \log 3n = \mathcal{O}(n^4) \tag{5}$$

Asymtptic behavior : o notation

- Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \rightarrow +\infty$

$$\lim_{n \rightarrow +\infty} \left|\frac{f(n)}{g(n)}\right| = 0 \quad (6)$$

- intuitively, this means that f is smaller than g

Asymtptic behavior : o notation

- Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \to +\infty$

$$\lim_{n \to +\infty} \left|\frac{f(n)}{g(n)}\right| = 0 \qquad (7)$$

- Please define this limit mathematically ?

Asymtptic behavior : o notation

- Mathematically speaking, we say that $f = o(g)$ if the ratio $\frac{|f(n)|}{|g(n)|}$ converges to 0 when $n \to +\infty$

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0 \tag{8}$$

- $$\forall \epsilon > 0, \exists A \in \mathbb{R}, \forall n \geq A, |\frac{f(n)}{g(n)}| \leq \epsilon \tag{9}$$

Asymptotic behavior : general rules

When $n \to +\infty$:

- if $\alpha < \beta$, $n^\alpha = o(n^\beta)$
- if $0 < a < b$,)
- if $\alpha > 0$, $\beta \in \mathbb{R}$, $(\log n)^\beta = o(n^\alpha)$
- if $a > 1$, $n^\alpha = o(a^n)$

Asymptotic behavior : equivalence

- We say that $f(n) \underset{n\to+\infty}{\sim} g(n)$ when

$$f(n) \underset{n\to+\infty}{=} g(n) + o(g(n)) \tag{10}$$

Asymptotic behavior : equivalence

- We say that $f(n) \underset{n \to +\infty}{\sim} g(n)$ when

$$f(n) \underset{n \to +\infty}{=} g(n) + o(g(n)) \tag{11}$$

- When talking about complexities, we will be interested in the **simplest equivalent.**

Equivalence

Exercice 3 : Find equivalents and the limits for the following functions :

- $u_n = 3n^3 - n^2(\sqrt{n} \sin n) + \cos(\sqrt{n})$
- $v_n = -0.2 * n^n + 10 * n^2 * n!$
- Maximum number of edges in a simple directed graph
- $n!$

Examples of algorithms

- Fast exponentiation
- Naive exponentiation
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- Enumeration of permutations

Examples of algorithms

- Fast exponentiation $\mathcal{O}(\log n)$
- Naive exponentiation $\mathcal{O}(n)$
- Merge sort $\mathcal{O}(n \log n)$
- Insertion sort $\mathcal{O}(n^2)$
- Matrix multiplication $\mathcal{O}(n^{2.37})$
- Enumeration of subsets, TSP, coloring $\mathcal{O}(2^n)$
- Enumeration of permutations $\mathcal{O}(n!)$

Orders of magnitude

Orders of magnitude

Taille	$n \log n$	n^3	2^n
$n = 20$	60	8000	1048576
$n = 50$	196	125000	1125899907000000
$n = 100$	461	1000000	1267650600000000000000000000000

$\Longrightarrow$ Hence the idea of a border between polynomial and exponential algorithms.

Profiling

- Another useful tool to monitor the execution of a program is **profiling**
- From the python docs : "A profile is a set of statistics that describes how often and for how long various parts of the program executed"
- `https://docs.python.org/3.6/library/profile.html`

Profiling

Exercice 4 : Profiling a piece of code

- **cd profiling** and profile some programs that we used before

Profiling

Exercice 4 : Profiling a piece of code

- **cd profiling** and profile some programs that we used before
- However note that when profiling **profiling_demo.py**, the elementary multiplications are not taken into account in the profiling output.

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- For a sequence of blocks :
- For a loop :

Computing complexities

We now want to compute some complexities with paper and pen.
Let us focus on some intuitive rules :

- For a sequence of blocks : complexities sum up
- For a loop :

Computing complexities

We now want to compute some complexities with paper and pen. Let us focus on some intuitive rules :

- For a sequence of blocks : complexities sum up
- For a loop : complexities of all iterations sum up
- If a loop consists in similar iterations, its complexity is the product of the compexity of one iteration by the size of the loop.

Running time

Exercice 5 : Computing a running time I

Please compute the running time and give the complexity of the following algorithm.

```
result = 0
for i in range(n):
    result += i**2
```

Running times

Exercice 6 : Computing a running time II

Please compute the running time and give the complexity of the following algorithm.

```
for i in range(n):
    for j in range(i):
        l = [i+j+k for k in range(n)]
```

Running times

Exercice 7 : Computing a running time II
Could we have known that is was polynomial without performing the exact computation ?

```
for i in range(n):
    for j in range(i):
        l = [i+j+k for k in range(n)]
```

Some mathematical concepts

- Mathematical induction
- Applications : prime factors decomposition, $\sum_{k=1}^{n} k$
- Optional

$$\sum_{k=1}^{n} k^2 \ ? \tag{12}$$

$$\sum_{k=1}^{n} k^3 \ ? \tag{13}$$

Insertion Sort

- We will study the classic **Insertion sort algorithm**, in order to illustrate the concept of **average-case complexity**.

Insertion Sort

Exercice 8 : **Insertion sort** :
cd insertion_sort/ and fix the function in **insertion_sort.py** in order to perform the algorithm.

Average-case complexity

- We assume a **uniform_distribution** on the integer that we want to sort. All values have the same probability.
- What is the average-case complexity of the algorithm ?

Complexity

Exercice 9 : use the file **complexity.py** in order to check if our theoretical reslut is correct. You will need to fix the function **number_of_operations()**

Python sorting

In python, *sort*() uses a variant of mergesort. `https://github.com/python/cpython/blob/master/Objects/listsort.txt`

Horner Algorithm

- Let us consider the case of evaluating polynoms
- A polynom is a function of the form
  $f : x \to a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$
- How many multiplications are involved with the naive method ?

Horner Algorithm

- Let us consider the case of evaluating polynoms
- A polynom is a function of the form
  $f : x \to a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$
- How many multiplications are involved with the naive method ?
- We look fot an algorithm that is faster than the naive solution.

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \tag{14}$$

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \tag{15}$$

- How many multiplications are now involved ?

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(a) = (((7a + 2)a + 0)a - 5)a + 1 \tag{16}$$

- How many multiplications are now involved ? $\mathcal{O}(n)$.
- So we went from quadratic to linear.

Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \qquad (17)$$

- We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \ldots, a_0]$

Evaluating polynoms

Exercice 9 : Implementation of Horner Algorithm

- Example of Horner algorithm when $P : x \to 7x^4 + 2x^3 - 5x + 1$:

$$P(x) = (((7x + 2)x + 0)x - 5)x + 1 \tag{18}$$

- We input the polynom to the algorithm as the list of the coefficients $[a_n, a_{n-1}, \dots, a_0]$
- Please modify **complexity/horner.py** so that it performs the horner algorithm.
- In order to test that our method is correct, we will test it against the method **polyval** from **numpy**.

Horner

- What do you see if you write **help(numpy.polyval)** inside python ?

Horner

- What do you see if you write **help(numpy.polyval)** inside python ?

```
Horner's scheme [1]_ is used to evaluate the polynomial. Even so,
for polynomials of high degree the values may be inaccurate due to
rounding errors. Use carefully.

References
----------
.. [1] I. N. Bronshtein, K. A. Semendyayev, and K. A. Hirsch (Eng.
   trans. Ed.), *Handbook of Mathematics*, New York, Van Nostrand
   Reinhold Co., 1985, pg. 720.
```

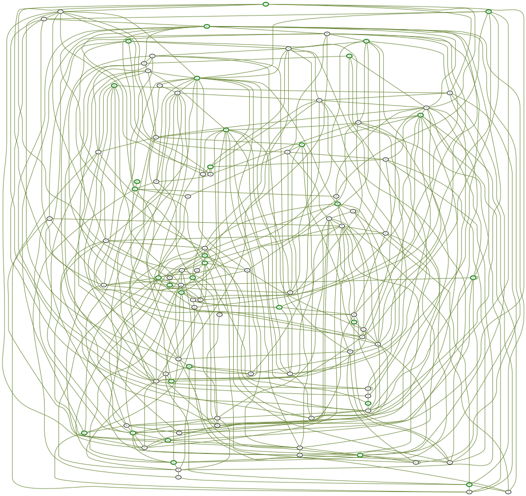
Figure: Horner is actually the method used by numpy

Space complexty

Space complexity is the sum of :

- input space
- auxiliary space : temporary space used during the algorithm

Graph problems



Graph problems

We will look at famous graph problems, typically of the form :

- "what is the largest subset of nodes of the graph, verifying some property ?"
- "what is the largest subset of edges of the graph, such that some property is verified ?"

networx

We will use **networx** to visualize graphs.

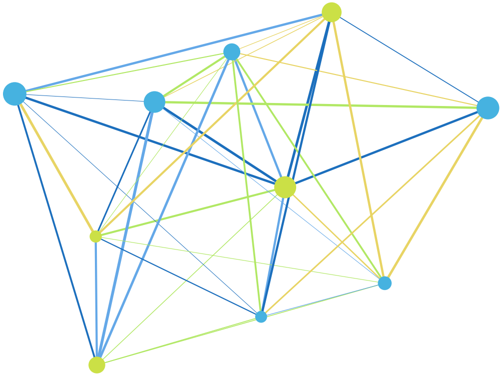


Figure: Undirected random graph generated with python

Warm up question

Given an **unoriented** graph with n nodes, how many edges can we build ?
Notation of a graph : $G(V, E)$

- V : set of n vertices
- E : set of edges

Warm up question

Given an **unoriented** graph with n nodes, how many edges can we build ?

Notation of a graph : $G(V, E)$

- V : set of n vertices
- E : set of edges, maximum size : $\frac{n(n-1)}{2} = \binom{n}{2} = \frac{n!}{2!(n-2)!}$

Networkx

- In order to do the following exercises, you will need **networx** (installed with the notebook).

Exercice 10 : Please **cd ./graphs/random_graphs** and use the notebook **Random_undirected_graph.ipynb** or **random_undirected_graph.py** to generate a random undirected graph with a chosen number of nodes and edges.

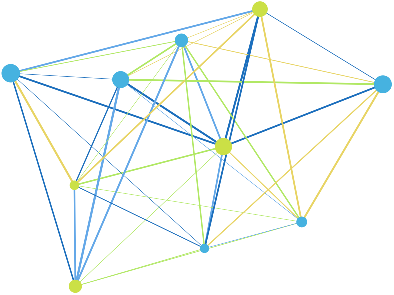


Figure: Random undirected graph with 10 nodes, 40 edges

Exercice 11 : Please use **random_directed_graph.py** to generate a random directed graph with a chosen number of nodes and edges.

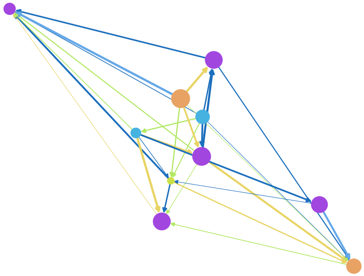
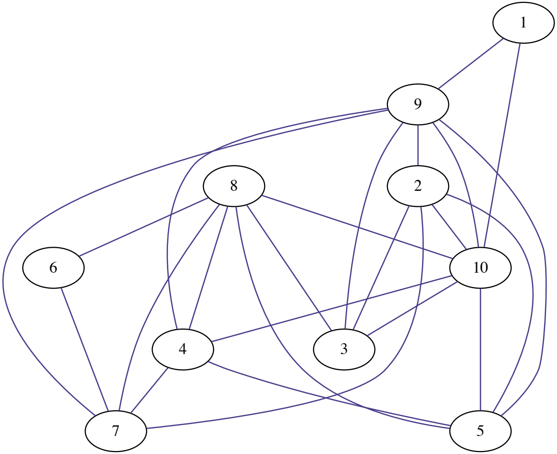


Figure: Random directed graph with 10 nodes, 30 edges

The dominating set problem



Dominating set

Say you want to cover a internet network. Some nodes (the emitters) are able to transmit information in the network, but not to all nodes : only to the nodes that are close enough.

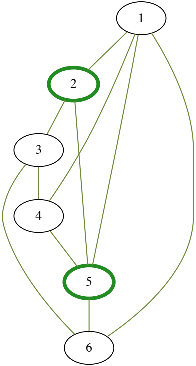
Dominating set

Say you want to cover a internet network. Some nodes (the emitters) are able to transmit information in the network, but not to all nodes : only to the nodes that are close enough.

Optimization problem: You need to cover the network, but with the smallest possible number of emitters (because then it is less work).

Exercice 12 : How would you formalize this problem with a **graph** ?

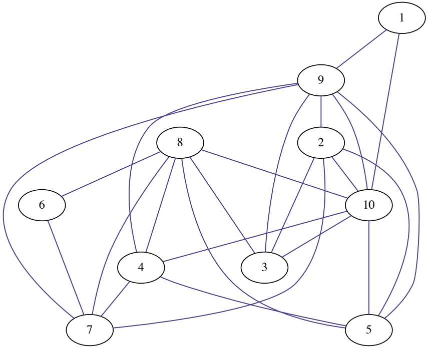
The dominating set problem



Mathematically speaking : if $G(V, E)$ is the graph. We look for a **subset of nodes** D such that **all nodes in the graph** are the neighbor of **at least one node** in D. linewidth

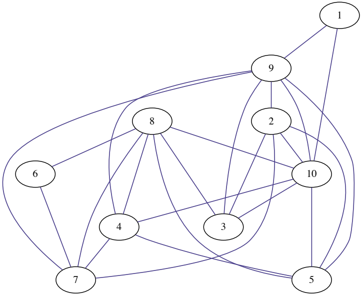
The dominating set problem

Mathematically speaking : if $G(V, E)$ is the graph. We look for a **subset of nodes** D such that **all nodes in the graph** are the neighbor of **at least one node** in D.



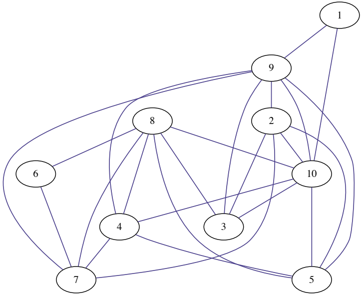
The dominating set problem

Mathematically speaking : if $G(V, E)$ is the graph. We look for a **subset of nodes** D such that **all nodes in the graph** are the neighbor of **at least one node** in D. And we want to pick the **smallest D** that "dominates" the network.



The dominating set problem

What is the most trivial dominating subset ?



Dominating set : example 1

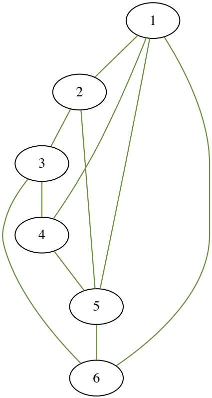


Figure: Some simple graph

Dominating set : example 1

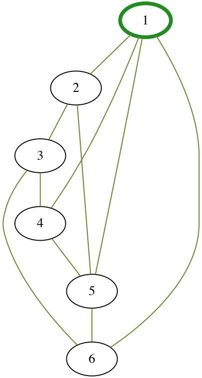


Figure: Is this a dominating subset ?

Dominating set : example 1

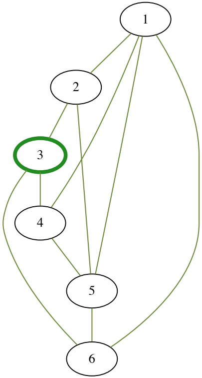


Figure: Is this a dominating subset ?

Dominating set : example 1

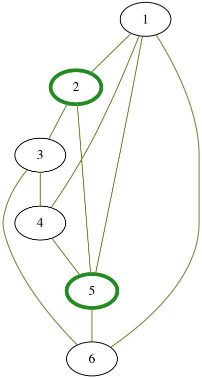
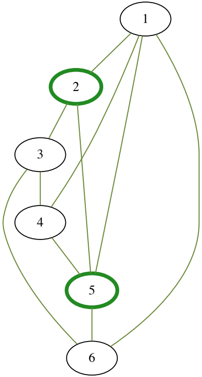


Figure: Is this a dominating subset ?

Dominating set : example 1

A **minimal dominating set** is a dominating set D such that removing any node from D prevents it from still beung dominating.



Dominating set : example 1

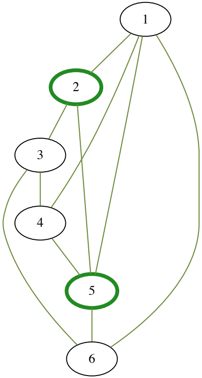
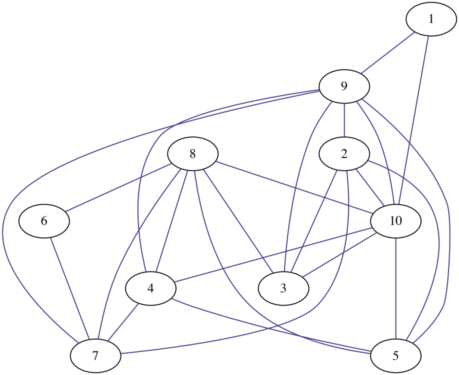


Figure: Is this a dominating subset ? Yes. Is it minimal ?

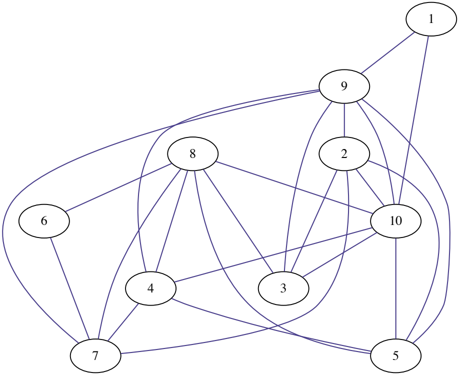
Dominating set : example 2

Please find a dominating set in this graph.



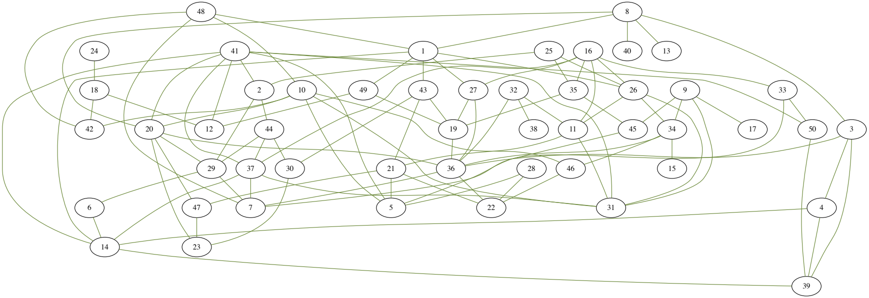
Dominating set : example 2

Please find a **minimal** dominating set in this graph.



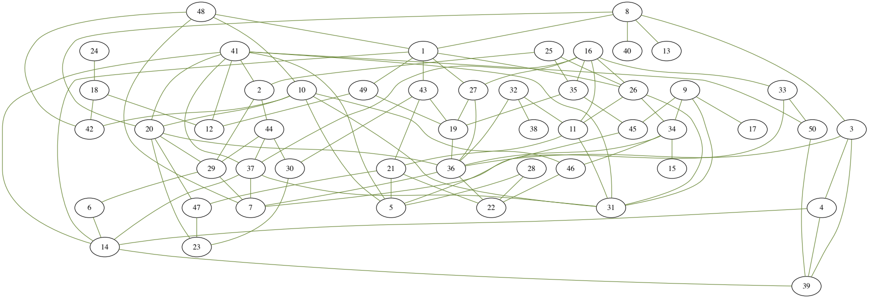
Dominating set : example 3

Please find a **minimal** dominating set in this graph.



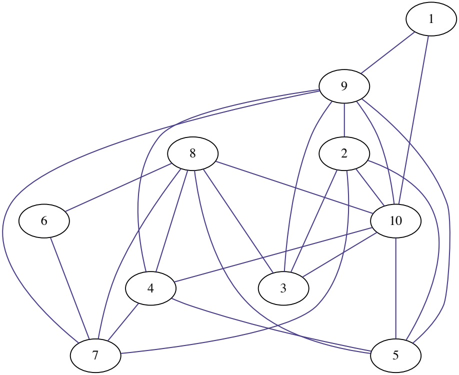
Dominating set : example 3

Is **minimal** the same thing as minimum ?



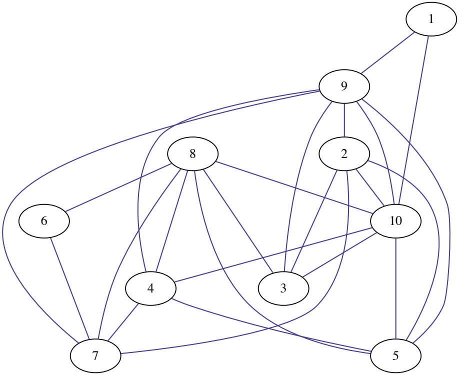
Dominating set : exhaustive search

What would be the **exhaustive search** in the case of the Dominating set problem ?



Dominating set : exhaustive search

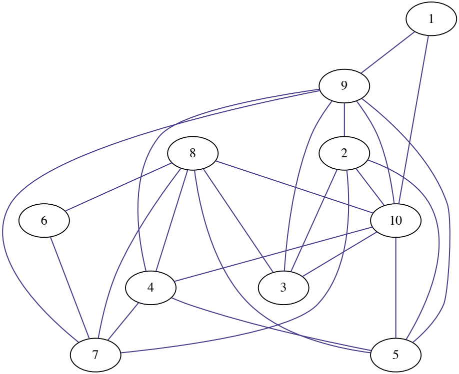
How many possibilities do have to try as a function of n ?



Dominating set : exhaustive search

How many possibilities do have to try as a function of n ?

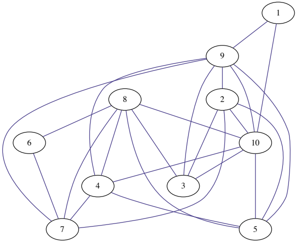
The number of subsets in $[1 : n]$ is :



Dominating set : exhaustive search

How many possibilities do have to try as a function of n ?
The number of subsets in $[1 : n]$ is :

$$2^n = \sum_{k=0}^{n} \binom{n}{k} \tag{19}$$



Heuristic

Ok so the exhaustive search is no possible. So what method should we use ?

Heuristic

Ok so the exhaustive search is no possible. So what method should we use ?

Let's build a **greedy algorithm**.

Greedy algorithm

In a graph (unweighted), the **degree of a node** is its number of neighbors.

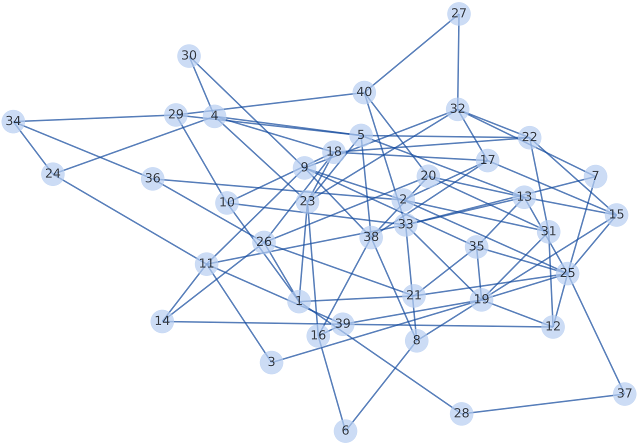
dominating set

Exercice 13 : Greedy algorithm implementation

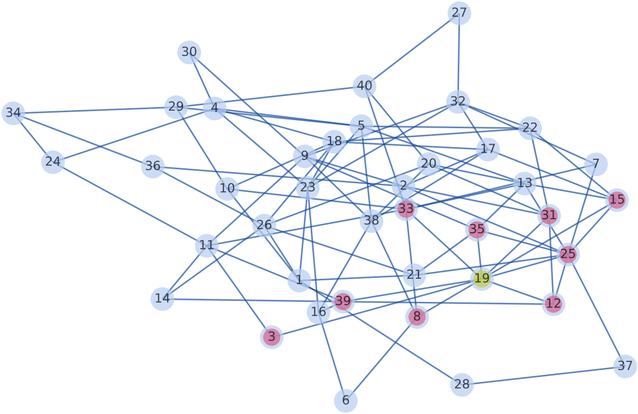
cd graphs/dominating_set and modify **greedy_standard.py** in order to apply the greedy algorithm :

- sort nodes by degree
- progressively add the to the set until it's dominating

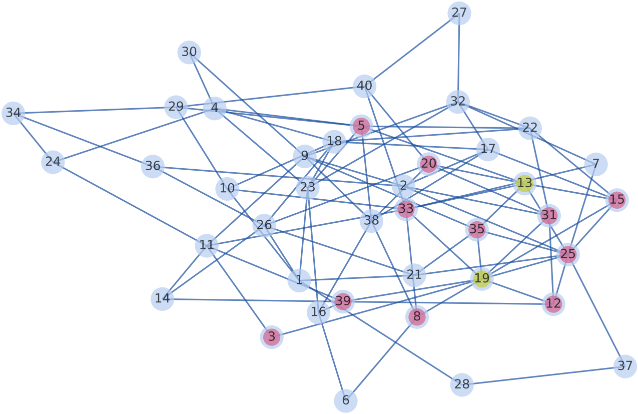
Initial graph



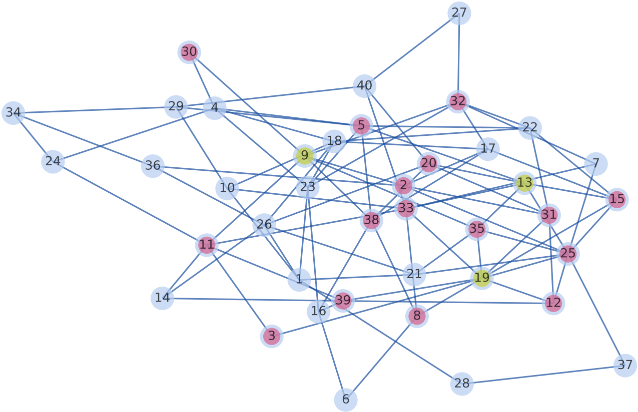
Subset size: 1
Algo step: 1



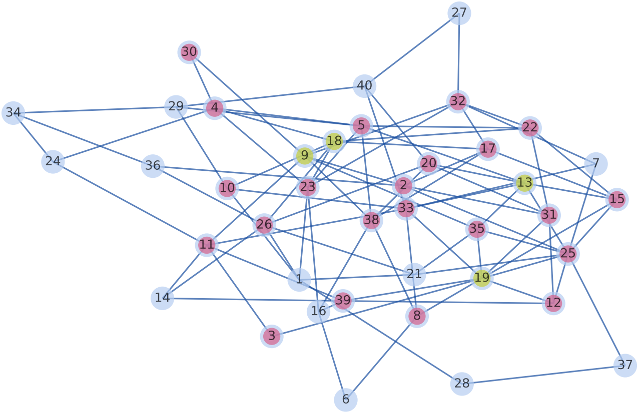
Subset size: 2
Algo step: 2



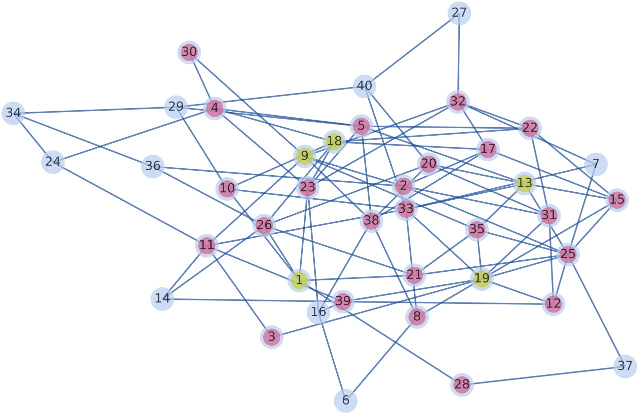
Subset size: 3
Algo step: 3



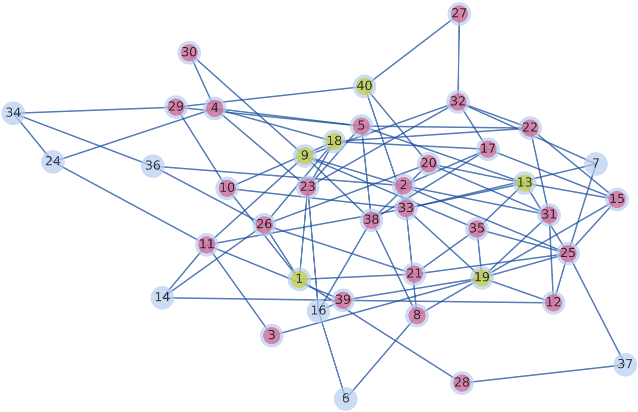
Subset size: 4
Algo step: 4



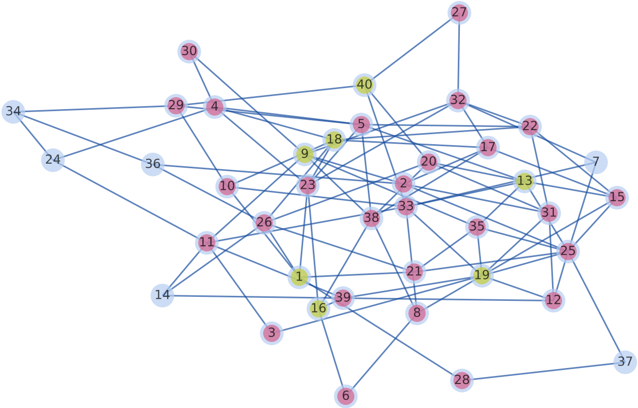
Subset size: 5
Algo step: 5



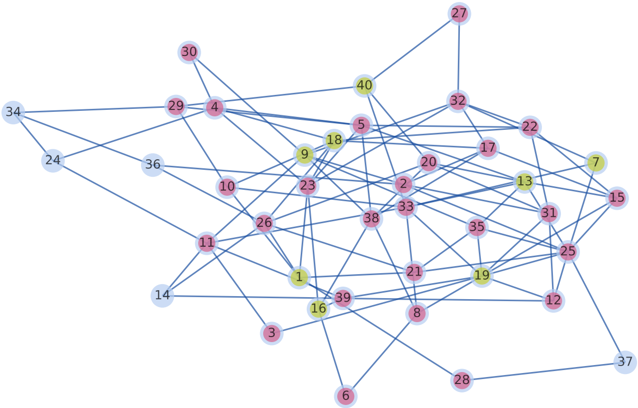
Subset size: 6
Algo step: 6



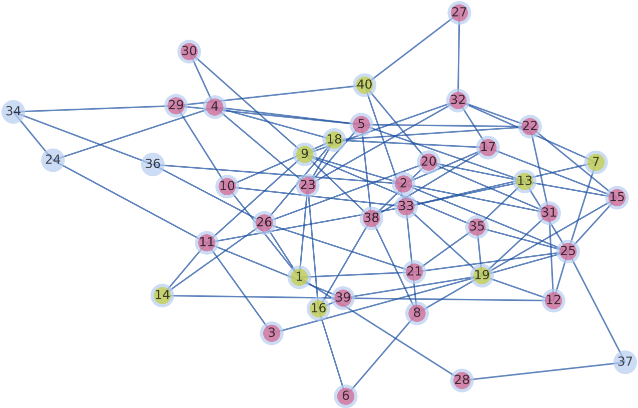
Subset size: 7
Algo step: 7

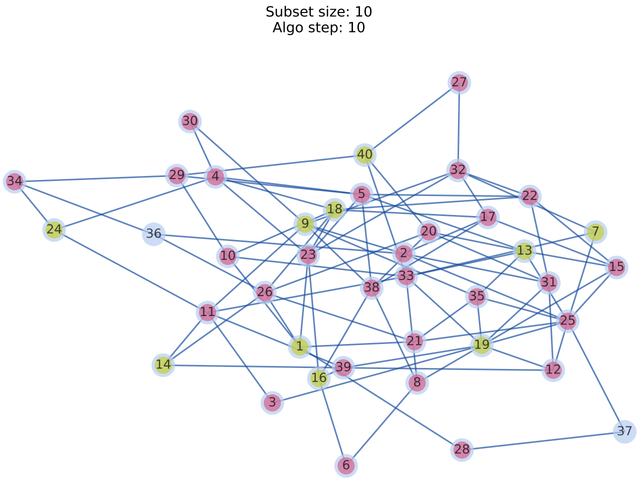


Subset size: 8
Algo step: 8

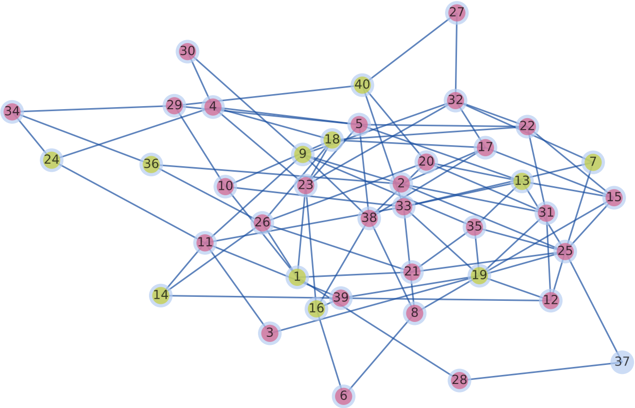


Subset size: 9
Algo step: 9

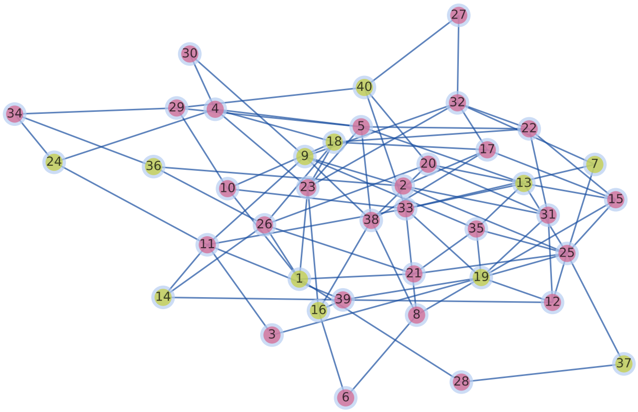



Subset size: 10
Algo step: 10
27
30
40
32
29
4
34
5
22
18
17
9
7
24
36
20
13
10
23
2
15
33
31
38
26
35
11
25
21
19
1
14
39
12
16
8
3
37
28
6

Subset size: 11
Algo step: 11



Subset size: 12
Algo step: 12



dominating set

Exercice 13 : Greedy algorithm implementation
Generate new instances of the problem using **generate_problem_instance.py** and apply the algorithm to them. You can use the file **params.txt**.

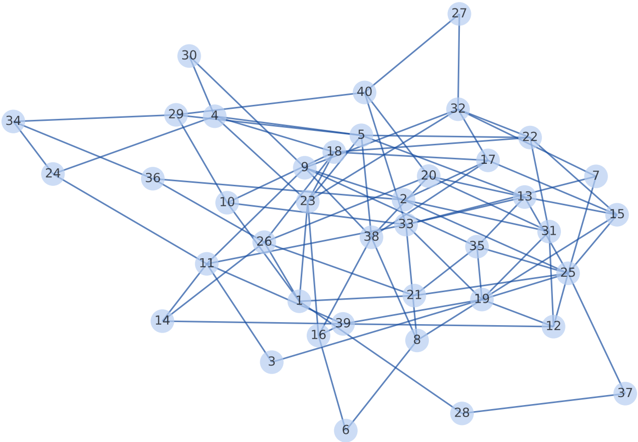
Complexity

Exercice 14 : What is the complexity of the greedy algorithm ?

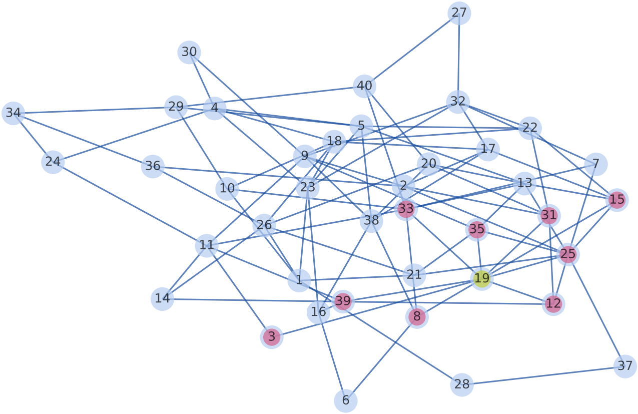
Variant

Exercice 15 : Try to see what happens using a variant of the heurstic, where we can add nodes that are already dominated, to the (built) dominating set. Which method is faster ?
You can use **greedy_bis.py**

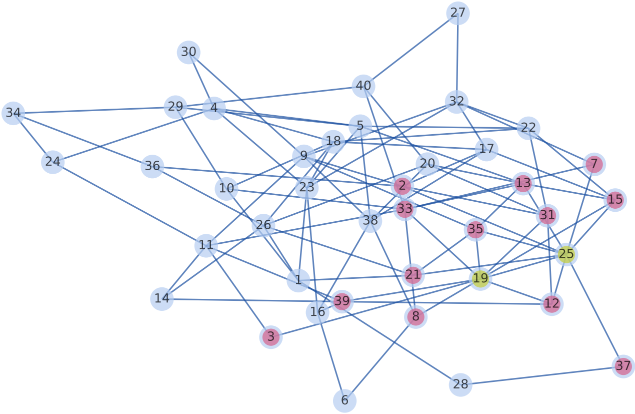
Initial graph



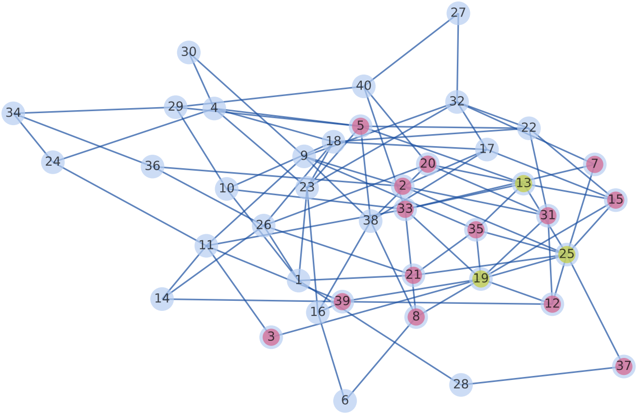
Subset size: 1
Algo step: 1



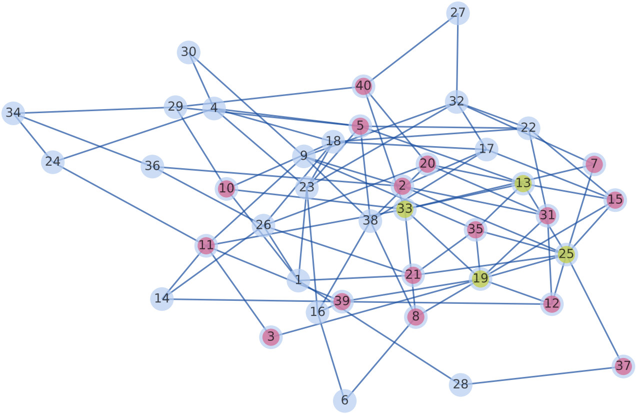
Subset size: 2
Algo step: 2



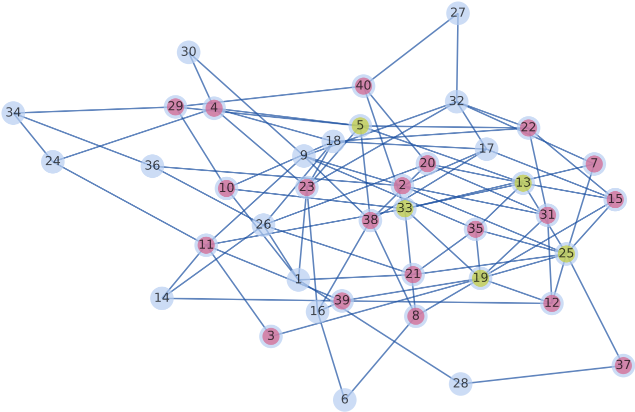
Subset size: 3
Algo step: 3



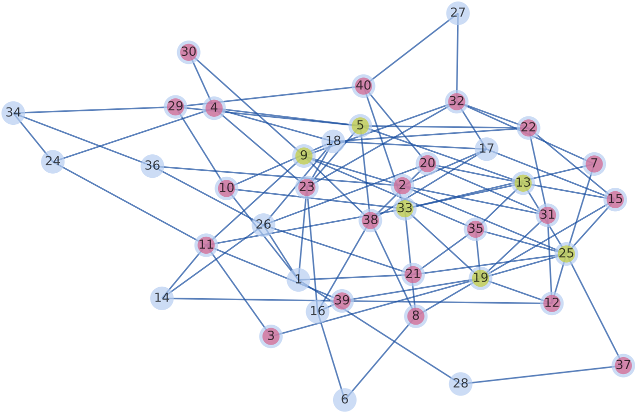
Subset size: 4
Algo step: 4



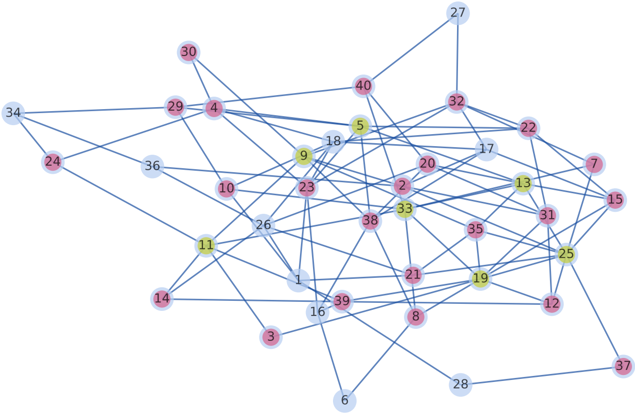
Subset size: 5
Algo step: 5



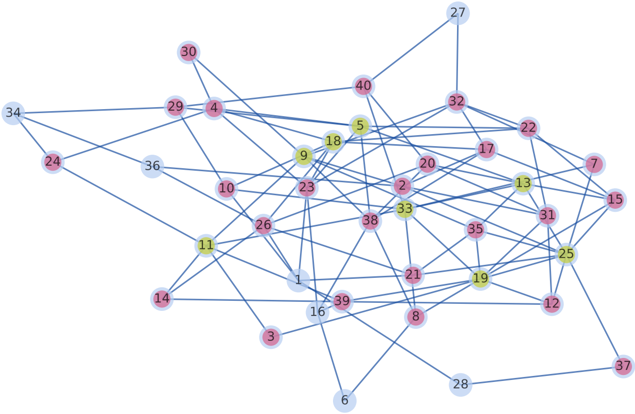
Subset size: 6
Algo step: 6



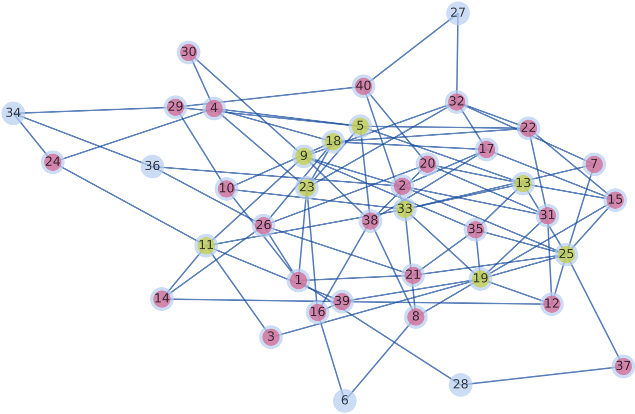
Subset size: 7
Algo step: 7



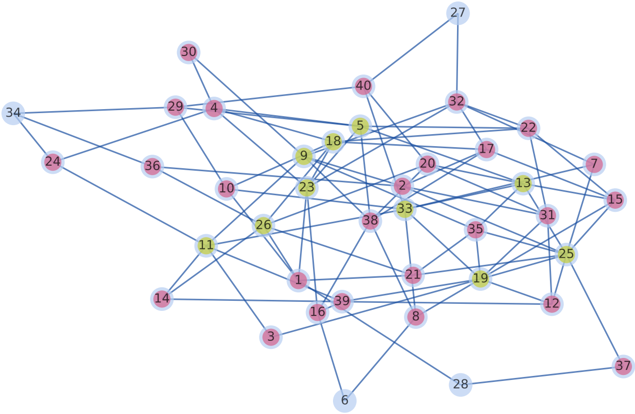
Subset size: 8
Algo step: 8



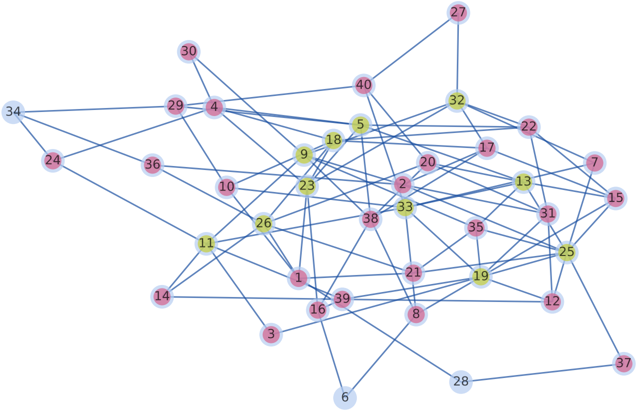
Subset size: 9
Algo step: 9



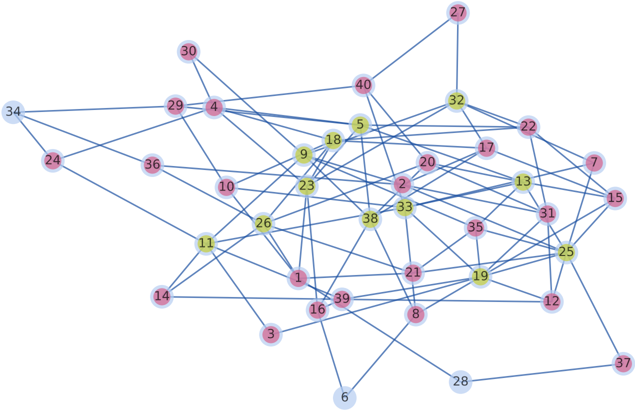
Subset size: 10
Algo step: 10



Subset size: 11
Algo step: 11



Subset size: 12
Algo step: 12



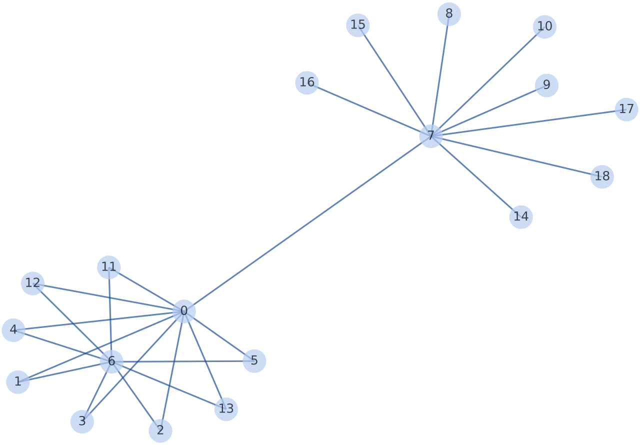
Variant 2

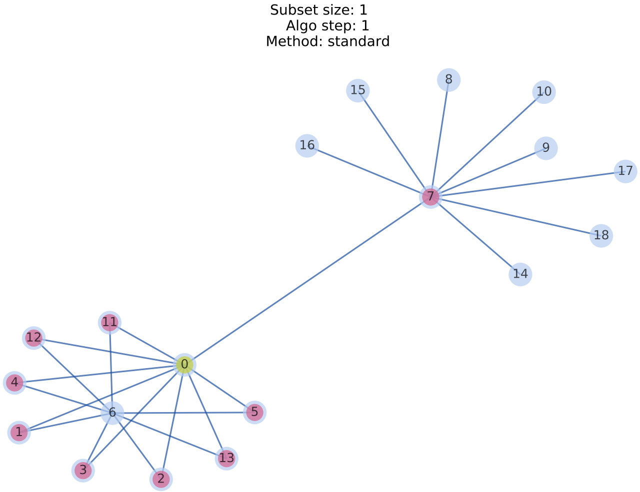
Exercice 16 : Implement of another variant where the degrees of the nodes are recomputed after each algorithm step.
You can use **greedy_ter.py**

Different performances

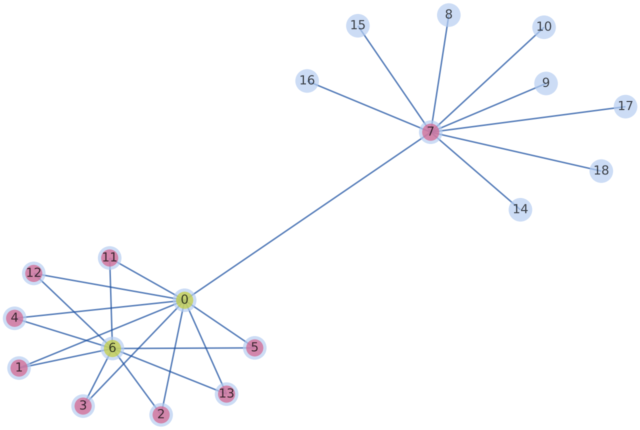
We have 3 variants of the algorithm, it seems that on most random cases "ter" works better (gives a smaller dominating set).
Exercice 17 : Can you find graph for which "standard" and "ter" are beaten by "bis" ?

Initial graph

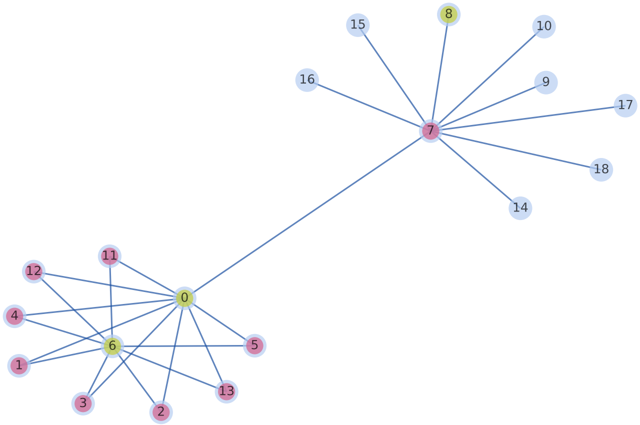


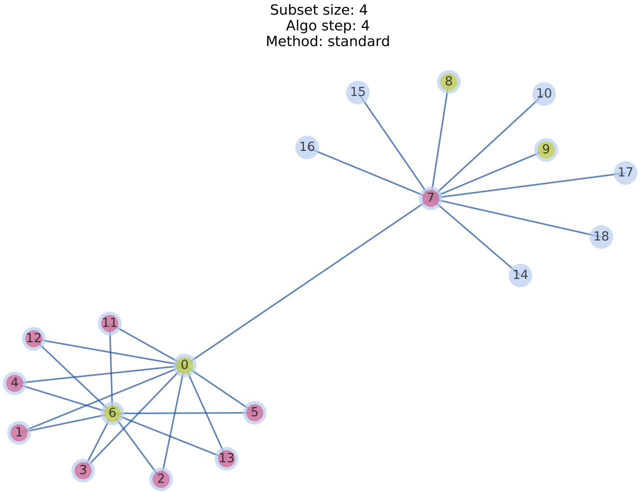

Subset size: 1
Algo step: 1
Method: standard
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2

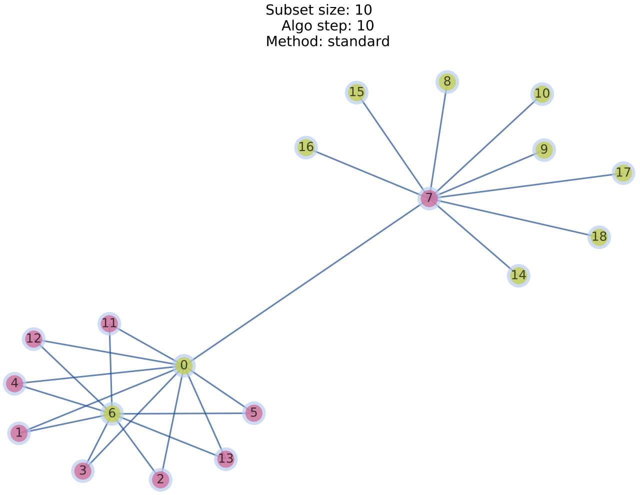
Subset size: 2
Algo step: 2
Method: standard

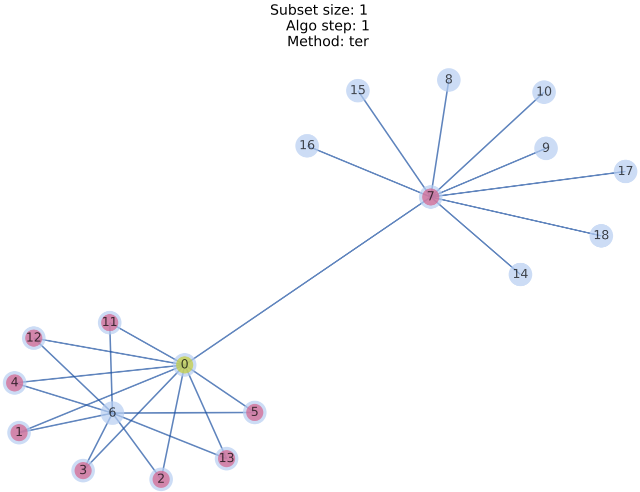


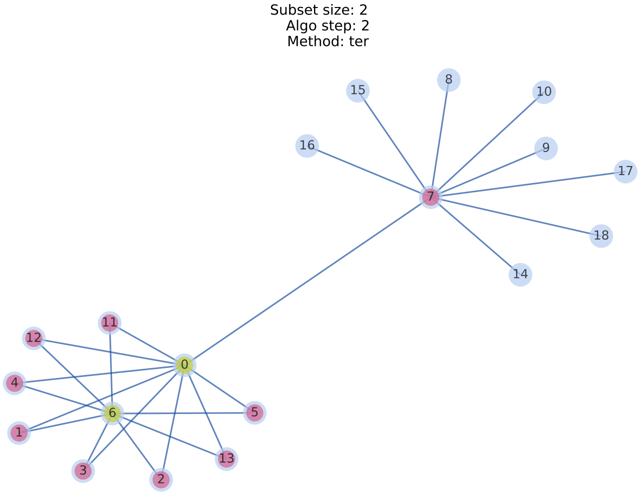
Subset size: 3
Algo step: 3
Method: standard



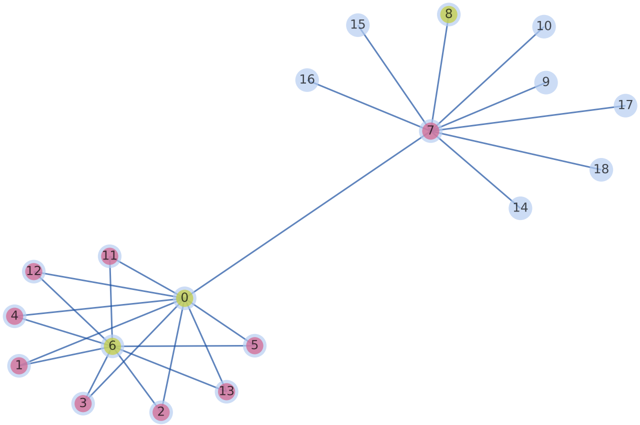

Subset size: 4
Algo step: 4
Method: standard
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2


Subset size: 10
Algo step: 10
Method: standard
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2

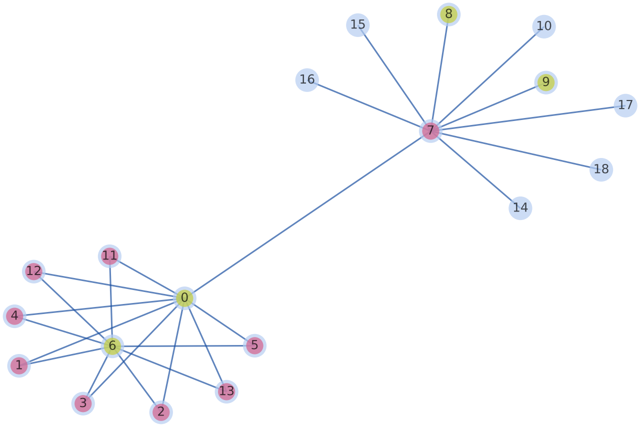

Subset size: 1
Algo step: 1
Method: ter
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2


Subset size: 2
Algo step: 2
Method: ter
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2

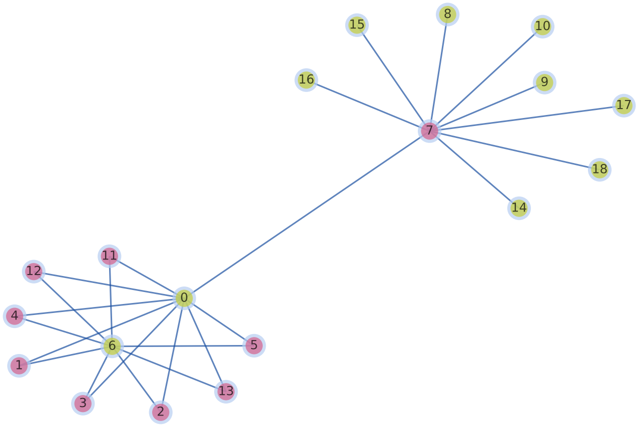
Subset size: 3
Algo step: 3
Method: ter



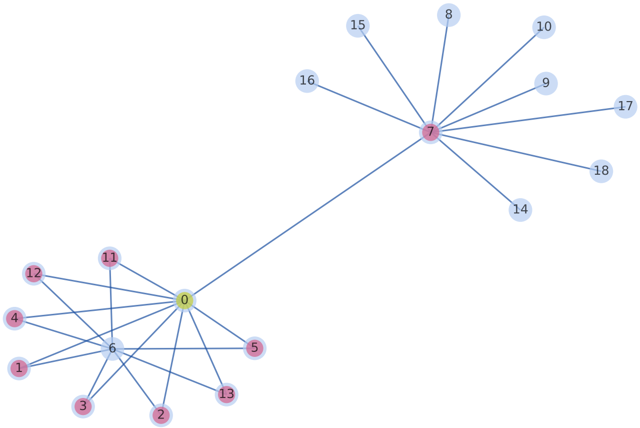
Subset size: 4
Algo step: 4
Method: ter



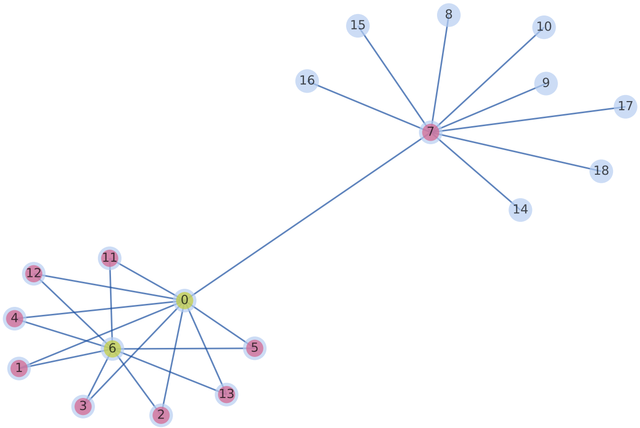
Subset size: 10
Algo step: 10
Method: ter

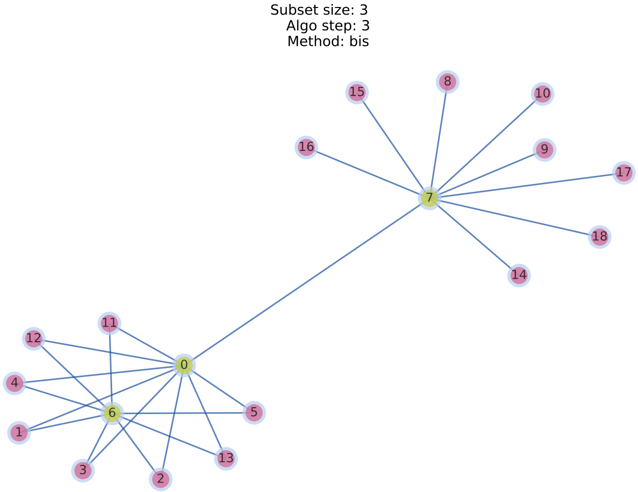


Subset size: 1
Algo step: 1
Method: bis



Subset size: 2
Algo step: 2
Method: bis




Subset size: 3
Algo step: 3
Method: bis
15
8
10
16
9
17
7
18
14
11
12
0
4
6
5
1
13
3
2

Non optimal greedy algorithm

Exercice 18 : Find a graph for which ”standard” gives a very bad solution.

Non optimal greedy algorithm

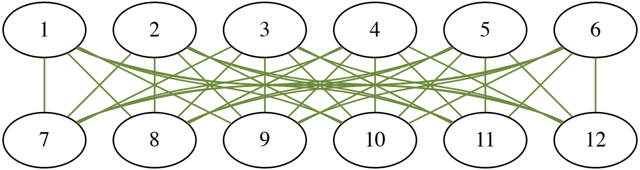


Figure: Complete bipartie graph

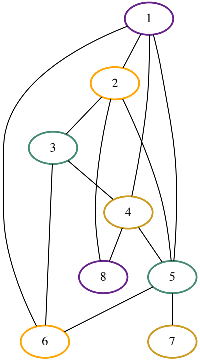
Networkx

The library networkx has some functions to work with most graph problems : `https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.dominating.dominating_set.html`

The coloring problem

Say you have a map with different countries. You need to assign a color to each country, so that two countries that have a common border are filled with a different color. We assume that we would like to use a small number of colors (the smaller, the better).
Exercice 19 : How would you formalize this problem with a graph ?

The coloring problem

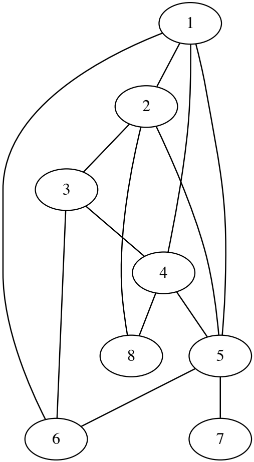


We want to find the smallest number of **fully disconnected subgraph** in a graph.

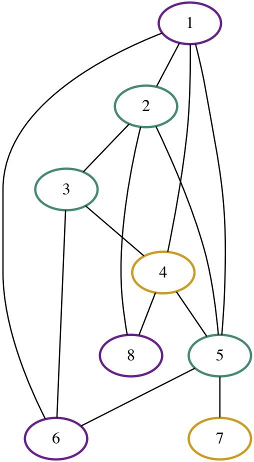
The coloring problem

We want to find the smallest number of **fully disconnected subgraph** in a graph.
Each subgraph will be associated with a color.

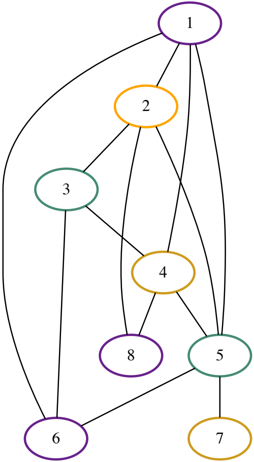
Coloring



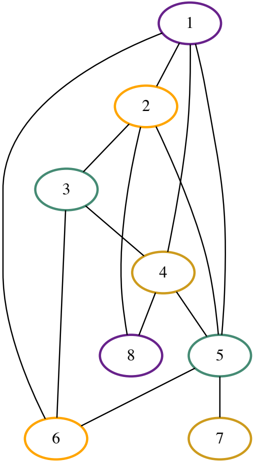
Is this a coloring ?



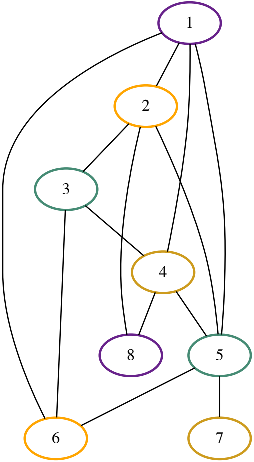
Is this a coloring ?



Is this a coloring ? yes



Could we have used only 3 colors ?



Coloring

- What would be a trivial coloring ?

Coloring

- What would be a trivial coloring ? assign a color to each node (very bad solution)
- Could you think of a heuristic ?

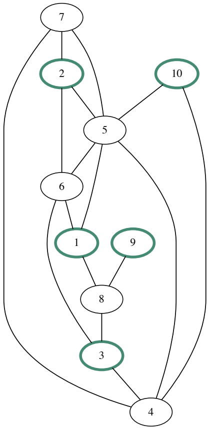
Other applications

- Planning activities (color : time in the day)
- Assigning frequencies (color : frequency)

Independent set

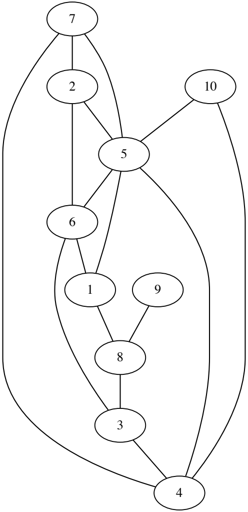
You have a group of people. Some people cannot work with each other. You want to build to largest possible team of people.
Exercice 20 : How would you formalize this with a graph ?

Independent Set

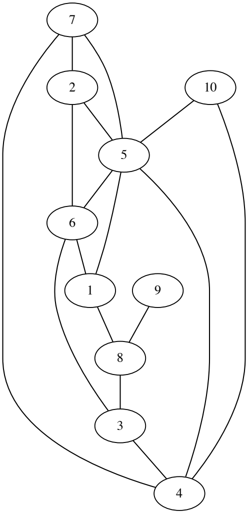


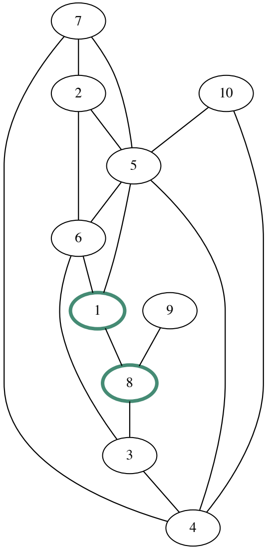
Assuming that an edge represents the fact that two persons cannot work with each other, we want to find **the largest disconnected subgraph**.

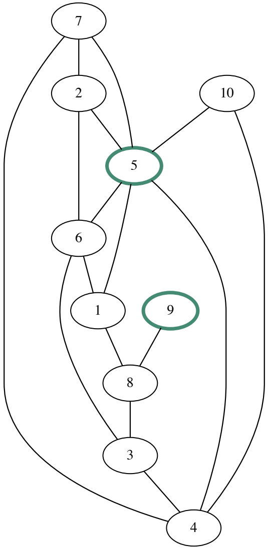
Independent set

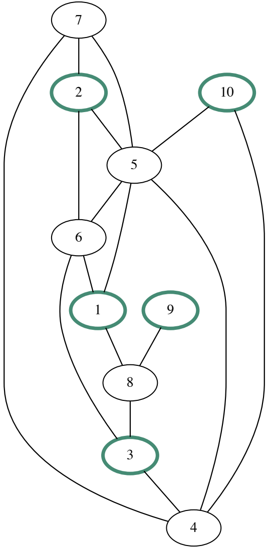


Independent set : what is a trivial independent set ?

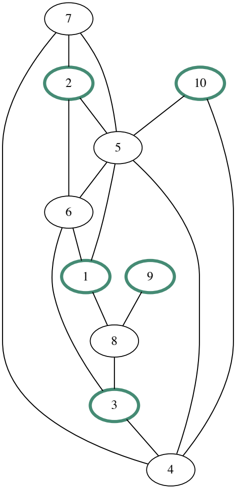



7
2
10
5
6
1
9
8
3
4


7
2
10
5
6
1
9
8
3
4


7
2
10
5
6
1
9
8
3
4

Maximal vs maximum independent set



Complexity

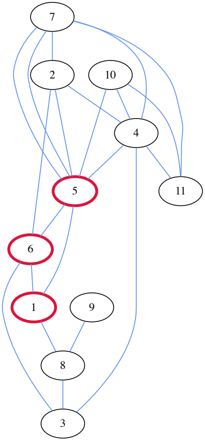
- The running time T of an algorithm A is its running time on the worst possible input (instance I) it can get (for a given size)
- The complexity $T(P)$ of a problem P is the running time of the best possible algorithm for that problem.

$$T(P) = \min_A \max_I T(P, A, I) \tag{20}$$

Equivalence between problems

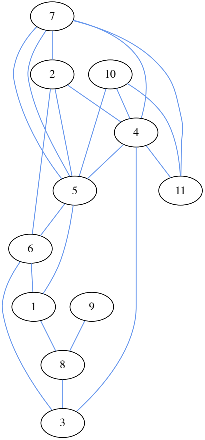
- Some problems have the same difficulty because they are equivalent
- Some are strictly more complex than others
- Hard problems : Maximum independent set, minimum coloring, smallest dominating set, TSP, etc.
- Easier problem : Shortest Path

Maximum clique problem

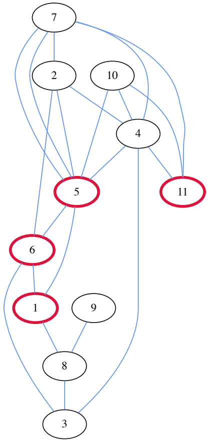


The **maxium clique** problem consists in finding the largest completely connected subgraph (the induced subgraph is **complete**)

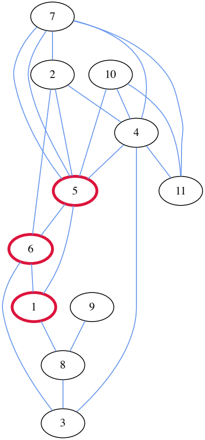
Maximum clique problem



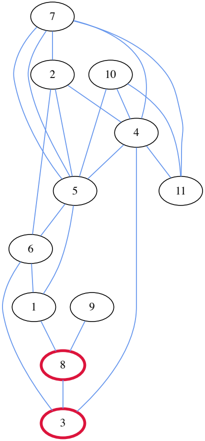
Maximum clique problem



Maximum clique problem



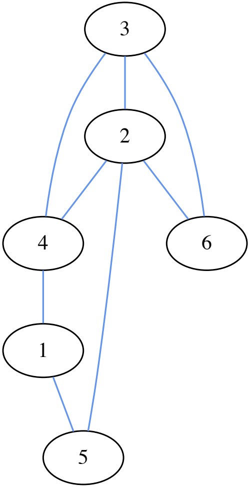
Maximum clique problem



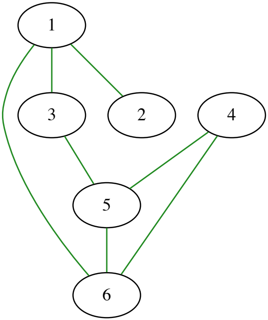
Equivalence between problems

Exercice 21 : Can you relate the maximum clique problem to another problem we saw before ?

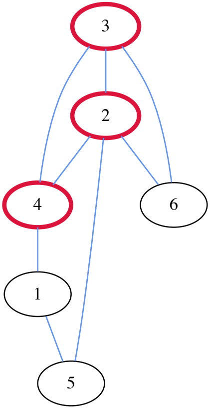
Linking problems



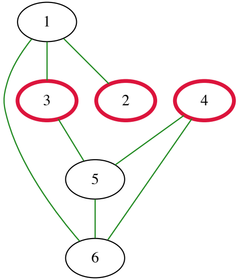
Linking problems



Linking problems



Linking problems



Polynomial-time reduction

To study a problem, it is sometimes useful to transform it into another.

Exercice 22 : Transformation

cd clique/ and use **complement_graph.py** in order to transform a graph into its **complement graph**.

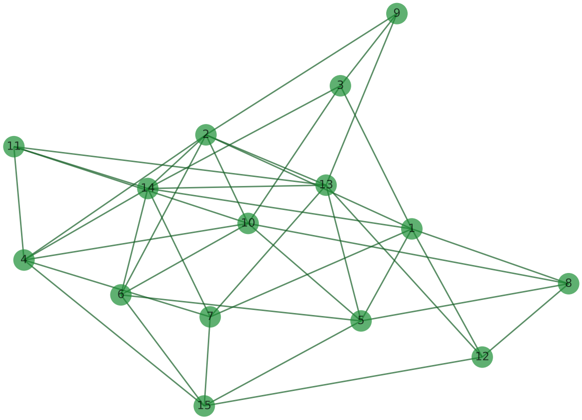
Exercice 22 : Transformation

cd clique/ and use **complement_graph.py** in order to transform a graph into its **complement graph**.

What is the complexity of this operation ?

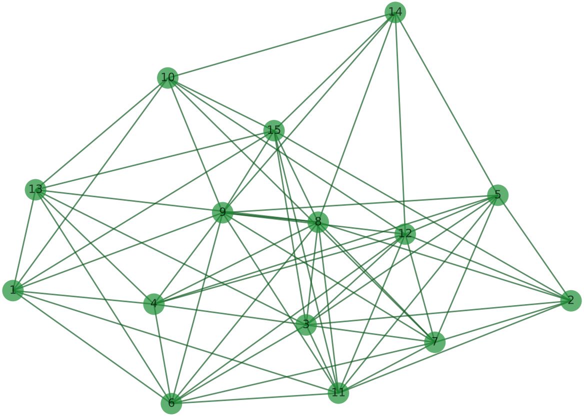
Complement graph

images/initial graph.pdf



Complement graph

images/complement graph.pdf



Dominating set to set covering

- This is another example of two problems that are equivalent.

Problems that are not equivalent

- Eulerian paths and hamiltonian paths

Classes of complexity

- Problems have been gathered under **classes of complexity**
- **P** : we can obtain a solution with polynomial complexity
- **NP** : we can verify a solution in polynomial time (doesn't mean we can find a solution)
- **NP hard** : if it is in P, all NP problems are in P.
- **NP complete** : NP and NP hard

P=NP ?