Graphs (Graphes)
Implementations and traversals

1 Representations / Implementations

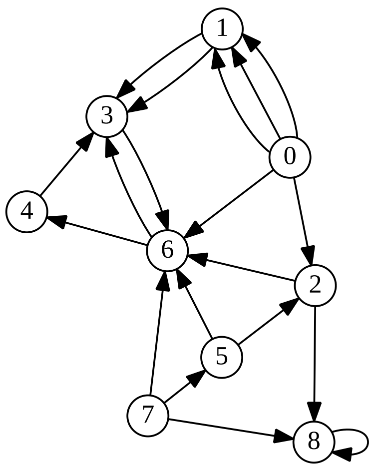

Figure 1: Digraph (Graphe orienté) G'_1

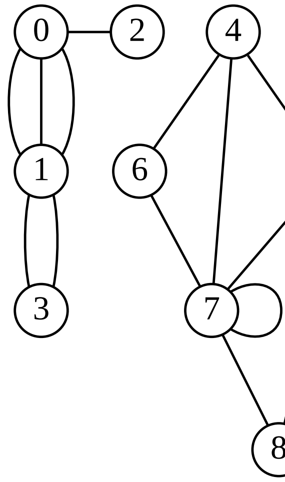

Figure 2: Graph (Graphe non orienté) G'_2

Exercise 1.1 (`GraphMat`: Static implementation)

This first implementation uses adjacency matrices.

1. With this implementation, what differences exist between a directed graph (or *digraph*) and a "undirected" one, a weighted graph and a none one, a simple graph and a mutigraph?
2. Give the matrix representations of the graphs in figures 1 and 2.
3. We want to use the same type to implement directed and undirected graphs, simple graphs and multigraphs. What should the implementation contain ?

Exercise 1.2 (`Graph`: Dynamic implementation)

1. What is the other way to represent/implement graphs?
2. With this representation, what differences exist between a directed graph and a "undirected" one, a simple graph and a mutigraph ?
3. Give the representations of the graphs in figures 1 and 2.
4. We want to use the same type to implement any kind of graphs: directed or not, simple and multigraphs. What should the implementation contain ?

Exercise 1.3 (dot)

Write the functions that return the `dot` representation of a graph for both implementations.

Examples:

- Graph G'_1 (figure 1)

```
>>> print(toDot(G1))
digraph G {
  0 -> 1
  0 -> 2
  0 -> 6
  1 -> 3
  2 -> 6
  2 -> 8
  3 -> 6
  4 -> 3
  5 -> 2
  5 -> 6
  6 -> 3
  6 -> 4
  7 -> 5
  7 -> 6
  7 -> 8
  8 -> 8
}
```

- Graph G'_2 (figure 2)

```
>>> print(toDot(G2))
graph G {
  1 -- 0
  1 -- 0
  1 -- 0
  2 -- 0
  3 -- 1
  3 -- 1
  5 -- 4
  6 -- 4
  7 -- 4
  7 -- 5
  7 -- 6
  7 -- 7
  8 -- 5
  8 -- 7
}
```

Exercise 1.4 (Load)

Our file format `GRA` is a text file, composed as follow:

- a first line containing 0 or 1: 0 for (undirected) graphs, 1 for digraphs
- a second line with a single integer representing the order of the graph
- serie of lines containing 2 integers representing each edge

See the provided files: `digraph1.gra graph2.gra`.

Write the functions that build a graph from a `".gra"` file in both implementations.

Bonus:
Write the functions that build a graph from a `".dot"` file (simplified).

Notes: Thereafter, we will essentially use simple graphs. The examples used here will be the graph G_1 (simple digraph from G'_1) and G_2 (simple graph from G'_2).

---

2 Traversals

Exercise 2.1 (Breadth-first traversal)

1. Draw the spanning forests associated with the breadth-first traversals of the graphs G_1 and G_2 from vertex 0 (vertices are chosen in increasing order).
2. Give the principle of the breadth-first traversal algorithm. Compare with the traversal of a general tree.
3. How can we efficiently store the spanning forest?
4. Write in both implementations the breadth-first traversal functions. The functions have to give the spanning forests.

Exercise 2.2 (Depth-first traversal)

1. Draw the spanning forests associated with the depth-first traversals of the graphs G_1 and G_2 from vertex 5 (vertices are chosen in increasing order).
2. Give the principle of the depth-first traversal algorithm. Compare with the traversal of a general tree.
3. **Graphs (undirected)**
   (a) What are the different kinds of arcs (egdes) met during the traversal?
   Add and name the missing arcs to the spanning forest of the depth-traversal of G_2 obtained in question 1.
   (b) What has to be added to the depth traversal to classify arcs ?
   (c) Write the depth-first traversal function, when the graph is undirected and in matrix implementation. Add, during the traversal, the kinds of met arcs.
4. **Digraphs**
   (a) What are the different kinds of arcs (egdes) met during the traversal?
   Add and name the missing arcs to the spanning forest of the depth-traversal of G_1 obtained in question 1.. How distinguish the different arcs?
   (b) We assign to each vertex a prefix value (first encounter) and a suffix value (last encounter). Write the conditions to classify arcs with these values (using an unique counter).
   (c) Write the depth-first traversal function, when the graph is directed and represented with adjacency lists. Add, during the traversal, the kinds of met arcs.

3 Applications

Exercise 3.1 (Bipartite graph (Graphes bipartis) – *P3 - 2015*)

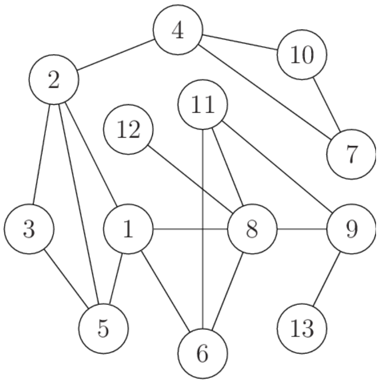

Figure 3: Graph G_3

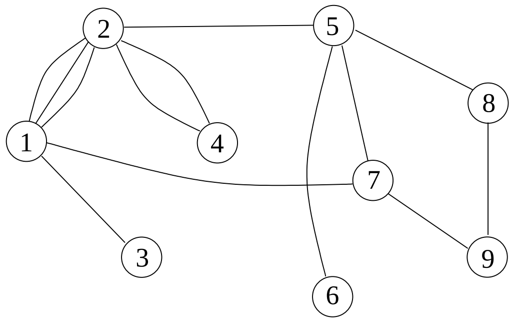

Figure 4: Graph G_4

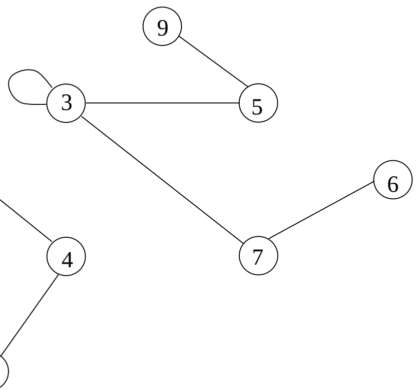

Figure 5: Graph G_5

A bipartite graph is an undirected graph $G =< S, A >$ where vertices can be partitioned into two sets S_1 et S_2, such that $(u, v) \in A$ imply either $u \in S_1$ and $v \in S_2$, or $u \in S_2$ and $v \in S_1$. That is, no edge connects vertices in the same set.

1. Are the graphs of figures 3 to 5 bipartite? For each bipartite graph give the two sets S_1 and S_2.
2. Write a function that tests whether a graph is bipartite.

Exercise 3.2 (Path – P3 2014)

1. How to find a path (a chain) between two vertices in a graph? Give two different methods and compare them.
2. Write a function that searches for a path between two vertices. If a path is found, it has to be returned (a vertex list).

Exercise 3.3 (What is this? – *P3 - 2015*)

Consider the following algorithm:

```
algorithm procedure build_graph(graph G, integer  s, n,graph  ref NG)
variables
   t_int_vect   map, dist
   queue        q
   integer        i, adj
begin
   for i ← 1 to N do   /* N = G order */
      map[i] ← 0
      dist[i] ← -1
   end for
   q ← new_queue
   q ← enqueue(s, q)
   dist[s] ← 0
   NG ← emptygraph
   add-vertex 1 to NG
   nb ← 1
   map[s] ← 1
   do
      s ← dequeue(q)
      for i ← 1 to d°(s, G) do
         adj ← nthsucc(i, s, G)
         if (dist[adj] = -1) and (dist[s] < n) then
            dist[adj] ← dist[s] + 1
            nb ← nb + 1
            add-vertex nb to NG
            map[adj] ← nb
            q ← enqueue(adj, q)
         end if
         if dist[adj] <> -1 then
            add-egde <map[s], map[adj]> to NG
         end if
      end for
   while not is_empty(q)
end
```

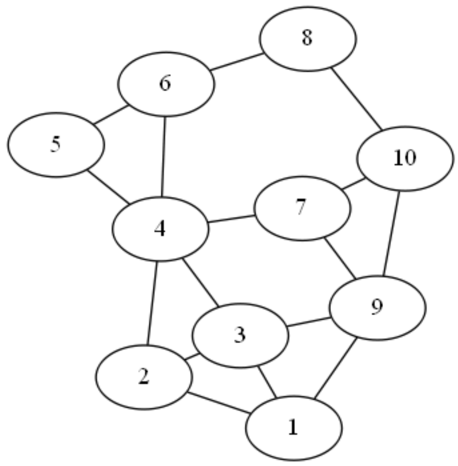

FIGURE 5 – Graph G_4

1. This algorithm is called with `build_graph(`G_4`, 5, 2, `NG`)` (G_4 the graph in figure 5).
   (a) Fill the array `dist`.
   (b) Fill the array `map`.
   (c) Draw the built graph (NG).
2. `build_graph(`G`, `s`, `n`, `NG`)` is called with G any non-empty graph, s a vertex of G, and n a positive integer.
   (a) During the execution, what does the array `dist` represent?
   (b) During the execution, what is the array `map` used for?
   (c) After the execution, what does the graph NG represent?

Translate this algorithm in Python.

Exercise 3.4 (Compilation, cooking...)

1. *Scheduling, a simple example:*

   The following statements have to be executed with one processor:

   ① read(a)
   ② b ← a + d
   ③ c ← 2 * a
   ④ d ← e + 1
   ⑤ read(e)
   ⑥ f ← h + c / e
   ⑦ g ← d * h
   ⑧ h ← e - 5
   ⑨ i ← h - f

   What are the possible orders of running?

   How to represent this problem with a graph?

   Each solution is called a *topological sort*.

2. What property should have the graph so that a topological sort exists?

3. When the graph is drawn lining up the vertices in a topological order, what can be observed?

4. (a) Let $suffix$ be the array of the last encounter of the vertices: the suffix order during the depth-first tarversal.
   Prove that for any pair of different vertices $u, v \in S$, if there is an arc in G from u to v, and if G has the property of question 2, then $suffix[v] < suffix[u]$.

   (b) Deduce an algorithm that finds a solution of topological order in a graph in static implementation (Here, we assumed that a solution exists.)

   (c) What has to be changed in the algorithm if we want it to check if a solution exists?

   (d) Write a Python function that returns a topological order as a vertex list.

What about cooking?