CentraleSupélec

Algorithmics and complexity

TD 1/7 – Graph search

The goal of this TD is to practice first algorithms seen in class, graph search.

Exercise 1 : Breadth-First Search and Bipartite graphs

First part – Depth of vertices and odd (length) cycle

Question 1

Using graph traversal algorithms seen in the first lecture, propose an algorithm that computes the number of edges between a given vertex (called a root) and all other vertices.

We will call this value the *depth* of each vertex in the traversal.

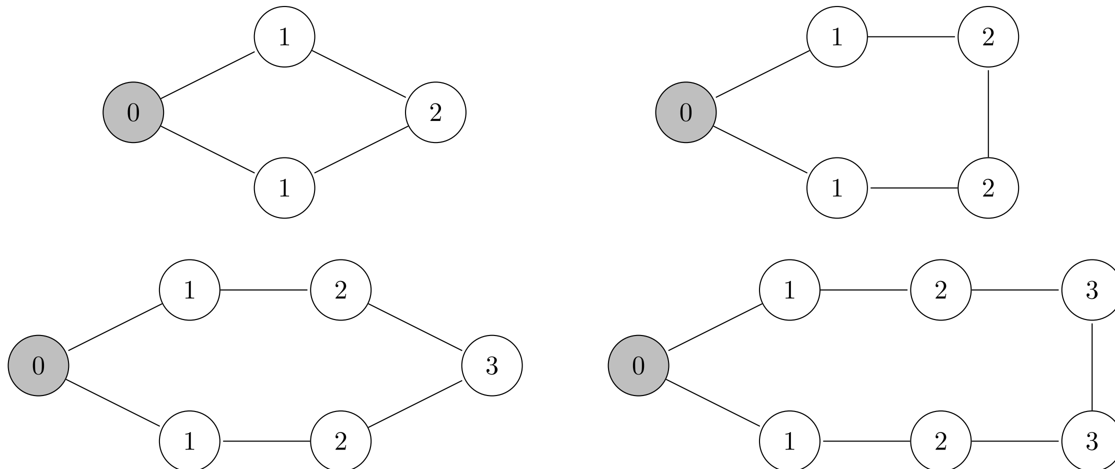
Solution elements :

We will use a breadth-first

```
def BFS_depth(V,E,s):
    depth = { s: 0 } # dispense de reached
    next = [s]
    while len(next)>0:
      n = pop_begin(next)
      for v in neighbours(n,E):
        if not v in depth:
          add_end(v,next)
          depth[v] = depth[n] + 1
    return depth
```

Question 2

Given the following cycles with even and odd length for which we have calculated the depths:



What do you think about (on the depths) the case of graphs with an **odd** cycle (in number of edges)? Is this a characteristic property? State the general case and give a proof!

Solution elements :

Characteristic property: A graph contains a cycle C with an odd number of edges iff:

$$\exists (x, y) \in E. \quad depth(x) = depth(y)$$

Proof:

all edges connect vertices of "neighboring" depths:

$$\forall (x, y) \in E. \qquad |depth(x) - depth(y)| \leq 1$$

- $\Rightarrow$ by contrapositive, we suppose that $\forall (x, y) \in C.\ depth(x) \neq depth(y)$, then $\forall (x, y) \in C.\ depth(x) = depth(y) \pm 1$. Let $C = (u_0, u_1, \ldots, u_n)$ be the cycle. The sum $\sum_{i=1}^{n} depth(u_i) - depth(u_{i-1}) + depth(u_0) - depth(u_n)$ is zero. For this to be the case, there must be as many edges with a weight difference of -1 as there are edges with a weight difference of +1. This means there must be an even number of edges along the cycle.
- $\Leftarrow$ if there exists an edge $(x, y) \in E$ for which $depth(x) = depth(y)$. We consider the path tree that was used to annotate the depths. In this tree x and y have a first ancestor z in common (possibly the root) from which we can form an odd cycle of size $2(depth(x) - depth(z)) + 1$ by adding the edge (x, y) to this subtree starting at z.

Question 3

Propose an algorithm that determines if a graph contains an odd cycle.

Solution elements :

We run the depth labeling algorithm (`BFS_depth` in question 1) then for each edge of the graph $(x, y) \in E$, we have to test whether x and y have the same depth

Second part – Bipartite graph

A graph $G = (V, E)$ is bipartite if its vertices can be divided into two disjoint subsets $V_1 \subseteq V$ and $V_2 \subseteq V$ such that each edge connects a vertex in V_1 to a vertex in V_2.

Question 4

In a bipartite graph, can there be a cycle with an odd number of edges? Is this a characteristic property? Justify your answer.

Solution elements :

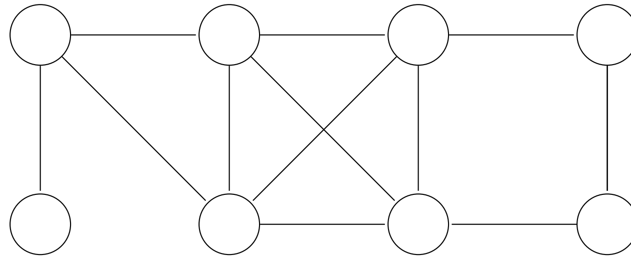
- $\Rightarrow$ If the graph is bipartite, any path alternates between a vertex of each partition so to form a cycle and return to the starting vertex, an even number of edges is necessary. In a bipartite graph, all cycles are therefore even.
- $\Leftarrow$ It is enough to consider the partition of vertices of even depth V_1 on one side and odd depth V_2 on the other. Suppose that there is no odd cycle then (question 2):

$$\forall (x, y) \in E. \qquad depth(x) = depth(y) \pm 1$$

Then all the edges are in $V_1 \times V_2$. The graph is therefore bi-partite.

Question 5

Propose an algorithm that allows to determine if a graph is bipartite. Test your algorithm on the following graph. Is it bipartite? Justify your answer

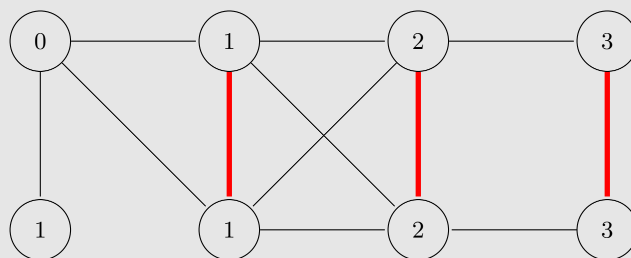


Solution elements :

Dans les questions précédentes, nous avons montré:

$$\text{Bipartite} \Leftrightarrow_{question4} \text{no odd cycle} \Leftrightarrow_{question2} \text{no neighboring vertices having the same depth}$$

On the proposed graph, starting from the top left vertex, we obtain the following depths:



The three bold edges are the proof that the graph is not bipartite.

Exercise 2 : Depth-First Search and 2-colorable graphs

Graph coloring is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color.

A 2-colorable graph is a graph that can be colored with only 2 colors.

Question 1

What is the link with the previous exercise? Justify your answer.

Solution elements :

Theorem : a graph is 2-colorable if and only if it is bipartite.

- $\Longrightarrow$ we partition V into $V_1 \cup V_2$ such that V_1 gathers all white nodes and V_2 all black nodes ($V_1 \cap V_2 = \emptyset$). Any edge connects a white node (in $V1$) to a black node (in V_2).
- $\Longleftarrow$ it is enough to choose the white for V_1 and the black for V_2 and as all the edges are in $V_1 \times V_2$, we are done.

Question 2

We want to write an algorithm, inspired by the **Depth-First Search** seen in the lecture, which takes as input a graph `G` and which returns a pair `(result, color)` where `result` is `True` if the graph is colorable, `False` otherwise and `color` is a dictionary associating a color 0 or 1 to each vertex.

This algorithm should *stop as soon as possible* when the graph is not 2-colorable.

Propose an **iterative** version and a **recursive** version.

Solution elements :

Iterative algorithm:

```
def coloring2_iter(V,E,s=V[0]):
   color = {s: 0}
   next = [s]
   while len(next)>0:
      n = pop_end(next)
      for v in neighbours(n,E):
        if not v in color:
          add_end(v,next)
          color[v] = 1-color[n]
        elif color[v] == color[n]:
          return False, color
   return True, color
```

and the recursive version:

```
# recursive function called by coloring2
def coloring2_rec(V,E,n,color):
  for v in neighbours(n,E):
    if not v in color:
      color[v] = 1-color[n]
      if not coloring2_rec(V,E,v,color):
        return False
    elif color[v] == color[n]:
      return False
  return True

# the calling function for recursive version
def coloring2(V,E,s=V[0]):
   color = {s:0}
   colorable = coloring2_rec(V,E,s,color)
   return colorable, color
```

Question 3

Is your algorithm correct, which means:

1. When it returns a coloring, is this 2-coloring correct?
2. When it returns `False`, are we sure that the graph is not 2-colorable?

Solution elements :

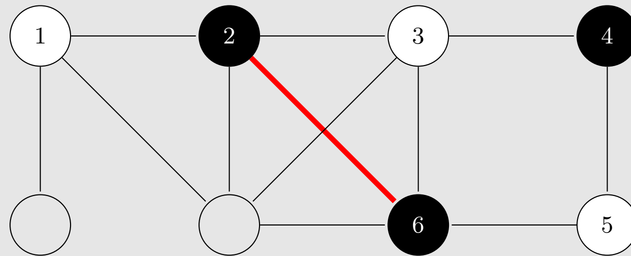
1. If the algorithm finds a coloring then **it has already tested all the edges**
2. If the algo does not find a coloring, the algo has found 2 neighboring nodes x and y colored the same! One could imagine that another path would have led to not coloring these neighbors with the same color, and that the graph is potentially colorable. It is enough to consider the traversal(colored) tree realized before the algorithm stopped on the edge (x, y). In this tree, the color of a vertex indicates the parity of the path from the root (the parity of the depth in this tree but not necessarily that of the graph). We then reason in a similar way to the reasoning of question 2 of exercise 1

Question 4

Test your algorithm on the above graph. Is it 2-colorable?

Solution elements :

We choose any node (here the top left one) and we apply a DFS, by annotating the colors, we obtain the following graph, the numbers correspond to the order of visit:



This graph is not 2-colorable because we get the edge in bold which connects 2 vertices of same color.

Exercise 3 : One DAG property

Prove the following theorem:

In a directed graph without cycles (i.e., a DAG–*Directed Acyclic Graph*) there is at least one vertex without incoming arcs.

Solution elements :

Let G be a DAG. Assume that all its vertices have at least one incoming edge.

1. Let v be a vertex of G. It has an incoming arc (u, v).
2. Let us go back, and consider now the vertex u. It has an incoming arc (x, u). If $x = v$, we have a cycle. Otherwise, we go back again.
3. We thus go back up the vertices until we find a vertex already met. This means the presence of a cycle, hence the contradiction.

This situation will happen since every vertex has an incoming arc: even if we eliminate all the vertices one by one, when we arrive at the last vertex z, it must have an incoming arc (y, z) with y a vertex already visited!