Algorithms: Assignment 4

Due on May 21, 2018 at 20:00

Prof. Antonio Carzaniga

A. Romanelli

Exercise 1

We are considering a graph G and we want to write a Python program that computes and prints the articulation points of G. We should point out that an articulation point is any given point of the graph G, such that if it is removed, the graph becomes disconnected.

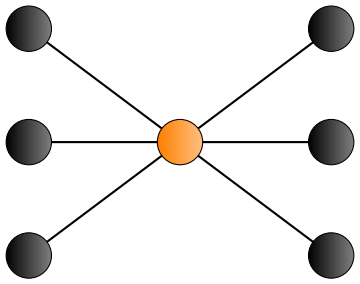
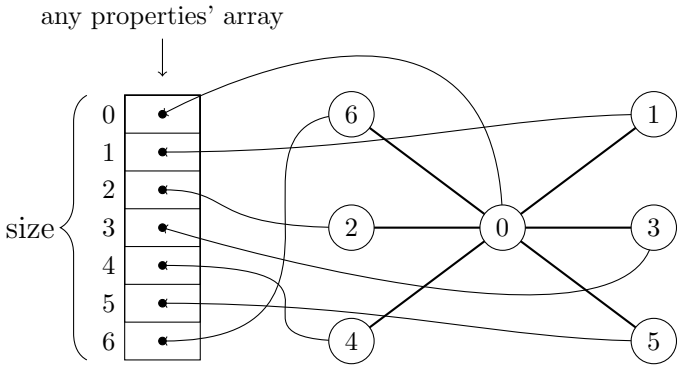


Figure 1: Representation of an articulation point of a given G

I decided to implement my graph in Python with an adjacency list: this means that my `Graph` structure is going to contain the following general properties:

- The number of vertices: `self.size`;
- The adjacent vertices to every vertex: `self.edges[[]]`.
  (Each vertex is assigned a list of other vertices, which define the edges of the graph);
- A global counter, which is going to keep track of "time": `self.globalTime`.

Now, my idea is to "map" every vertex to more specific properties:



The result is a one-to-many correspondence between a node and its properties, which are stored in array within the graph. For this exercise I implemented the following vertex properties:

- Adjacent vertices;
- Time of first visit;
- Fastest visit time;
- Previous vertex;
- Flag whether the vertex is an articulation point;
- A colour to flag whether we visited the vertex already.

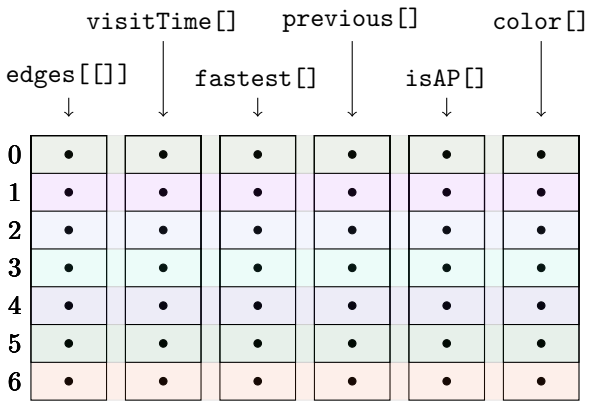


Figure 2: Each row in this column space represents the set of properties common to a given vertex.

From my illustrations it clearly follows that the properties of the vertices can be grouped up into arrays of length $|G|$, so that to access a property of a given vertex v one can simply access `property[v]`:

```
class Graph:
    def __init__(self, size):
        self.size = size
        self.globalTime = 0
        self.edges = []
        for i in range(size):
            self.edges.append([])
        self.visitTime = [float("Inf")] * size
        self.fastest = [float("Inf")] * size
        self.previous = [None] * size
        self.isAP = [False] * size
        self.color = ['white'] * size

    def insertEdge(self, u, v):
        self.edges[u].append(v)
        self.edges[v].append(u)
```

The only code here worth explaining might be the initialisation of `self.edges`, which gave me a lot of problems initially, since I was creating it like so:

```
  self.edges = [[]]*size
```

But this would instantiate a multidimensional array having as element the same reference to another array, however this was only a Python's pettiness.

The `insertEdge(u,v)` method takes care of establishing a non-directed edge between two vertices.

To find out the articulation points in the graph G, I'm going to use a *Depth First Search algorithm*, because I want to find vertices with two possible specific properties.

An articulation point is then a such point if it satisfies one of the following two conditions when contained in a *Depth First Search tree*:

A node is at the root of the tree of the *Depth First Search* and has two children.

A node is not at the root of the *DFS* tree and it has a child which is the only connection to its adjacent nodes.

The first condition is evident by the fact that in a *DFS*, if a vertex has more then two children it means that it already recursed all the way to the bottom and could not traverse between the two branches and thus the vertex at the root is an articulation point.

We are going to implement this search recursively and in order to do that we will have our case to be when all the nodes of the graph are visited and we are going to use an helper function in order to recurse on the adjacent vertices when iterating on each one of them.

```
39      def findArticulationPoints(self):
40          for i in range(self.size):
41              if self.color[i] == 'white':
42                  self.checkVertex(i)
43
44          counter = 0
45          for i in range(self.size):
46              if self.isAP[i] == True:
47                  counter+=1
48          print(counter)
49
50          for i in range(self.size):
51              if self.isAP[i] == True:
52                  print(i)
```

For each vertex present in the graph that hasn't been visited yet we are going to check whether it is an articulation point. Once we have flagged all our articulation points, we are going again to iterate over all the vertices and print out the ones that were flagged.

```
17      def checkVertex(self, vertex):
18          self.color[vertex] = 'gray'
19          self.globalTime += 1
20          self.visitTime[vertex] = self.globalTime
21          self.fastest[vertex] = self.globalTime
22          children = 0
23          for adj in self.edges[vertex]:
24              if self.color[adj] == 'white':
25                  children += 1
26                  self.previous[adj] = vertex
27                  self.checkVertex(adj)
28                  self.fastest[vertex] = min(self.fastest[vertex], self.fastest[adj])
29                  if self.previous[vertex] == None and children > 1:
30                      self.isAP[vertex] = True
31                  if self.previous[vertex] != None and self.fastest[adj] >= self.visitTime[
    vertex]:
32                      self.isAP[vertex] = True
33              elif adj != self.previous[vertex]:
34                  self.fastest[vertex] = min(self.fastest[vertex], self.visitTime[adj])
35              self.color[adj] = 'black'
```

The first thing we do when we access a vertex is to make sure we flag it as a vertex that is currently being visited (colour "gray"), then we can log into its corresponding property key the time at which we discovered the vertex. Then we initialise the children counter to zero and make sure to step forward to global timer by one unit.

Secondly, we will want to traverse to the adjacent vertices just like we would normally do with our *DFS* tree, keeping count of the children and by properly flagging our vertices. Each adjacent vertex will also be recursively checked for the conditions of existence of an articulation point.

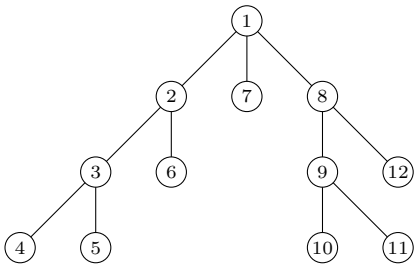


Figure 3: Order in which the nodes of the *DFS* tree are visited from node **1**, which corresponds to the *visitTime* property.

Finally we check whether the two above-stated conditions actually verify and if that is the case we assign the key of the vertex with the appropriate boolean flag. I also had to log an additional property, namely `fastest`, which is a property that keeps track of whether the node could have been accessed in a faster way than its initial `visitTime`, it is always smaller or equal than the `visitTime`.

I'll show with an example the implementation of our second condition of existence given the following example:

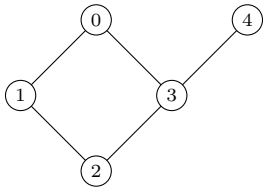


Figure 4: The graph has only one articulation point being at node **3**.

Our first condition of existence in this case would not apply, since the node **3** is not at the root of our *DFS*. The iteration in this case is going to look as follows:

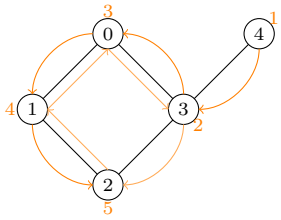


Figure 5: Visit times of the nodes in the graph

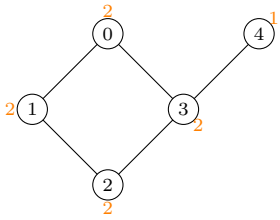


Figure 6: Fastest times of the nodes in the graph

If the following condition is true:

```
1  if self.previous[vertex] != None and self.fastest[adj] >= self.visitTime[vertex]
```

It must mean that when iterating through the adjacent vertices, we somehow found a way to reconnect ourselves to our initial node or another node that was visited before the one we started from, and its fastest time was transmitted to all the nodes from the bottom of the recursion.

Complexity Analysis

Our complexity will be the same as a normal DFS tree which is — as we know from theory — in the order of $\mathcal{O}(n+m)$ with some additional book-keeping, modifying the arrays stored within the `Graph` structure in constant time. Then we only iterate $2n$ times to compute the amounts of articulation points and to print all of them. Thus, our overall complexity will be $\mathcal{O}(n+m)$.

Exercise 2

We are given a maze that is composed by $n \times m$ of equally sized cells, that are arranged in a grid and that can be four different types:

□ Empty — ■ Blocked — ■ Start — ■ Exit

Starting from the start block, our task is to find the exit, if we can find it, with the shortest possible path, only by being able to move along the two orthogonal axis x, y of our grid.

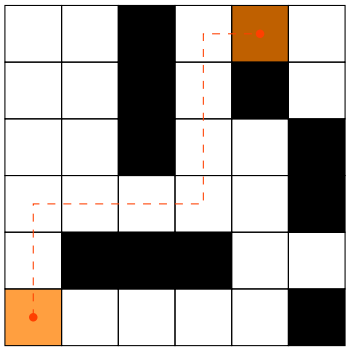


Figure 7: An example with shortest path of length 9

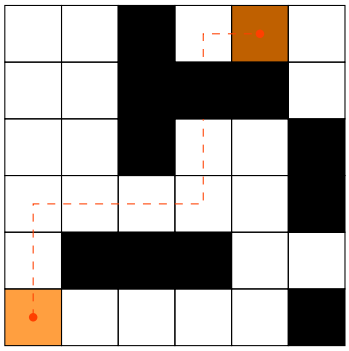


Figure 8: An example with no solution

In order to solve this problem, it comes in handy to consider our grid as a graph, where each cell is connected to the ones adjacent to it, excluding the diagonals, since we cannot move diagonally.

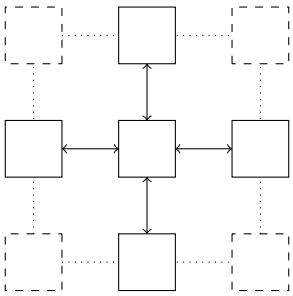


Figure 9: An edge between two white vertices represents a possible move

The strategy to solve this exercise will consist in running a *Breath First Search* from our start point, which will traverse all the viable cells (the non-blocked ones) incrementing a counter by one each time we traverse to an adjacent cell, until it eventually I find the exit cell or until there are no more vertices left to be explored, in which case there won't be a solution.

This suggest us that our complexity should be in the order of $\mathcal{O}(n + m)$ since that's the complexity of the *BFS* algorithm. The most of our complexity, which should instead be $\mathcal{O}(n \cdot m)$, will derive from the fact that we need to parse our standard input into a useful data structure on which we can run our *BFS*.

Exercise 2 continued on next page...

```
1  class Cell:
2      def __init__(self, color, isExit=False, isStart=False):
3          self.x = 0
4          self.y = 0
5          self.isExit = isExit
6          self.isStart = isStart
7          self.distance = float("Inf")
8          self.color = color
```

Structure of an individual cell of a grid

```
11 class Grid:
12     def __init__(self, n, m):
13         self.matrix = []
14         self.x = m
15         self.y = n
16         self.start = None
17 
18     def addRow(self, n, row):
19         self.matrix.append(row)
20 
21     def BFS(self, start):
22         q = queue.Queue()
23         start.distance = 0
24         q.put(start)
25         while not q.empty():
26             cell = q.get()
27             adjacents = []
28             if cell.x-1 >= 0:
29                 adjacents.append(self.matrix[cell.y][cell.x-1])
30             if cell.x+1 < self.x:
31                 adjacents.append(self.matrix[cell.y][cell.x+1])
32             if cell.y+1 < self.y:
33                 adjacents.append(self.matrix[cell.y+1][cell.x])
34             if cell.y-1 >= 0:
35                 adjacents.append(self.matrix[cell.y-1][cell.x])
36             for adj in adjacents:
37                 if adj.color == 'white':
38                     adj.color = 'gray'
39                     adj.distance = cell.distance + 1
40                     if adj.isExit:
41                         print(adj.distance)
42                         return
43                     q.put(adj)
44             cell.color = 'black'
45         print('no')
```

Implementation of the grid structure, along with class methods, to insert a cell into the grid and to perform a *BFS* from a given cell. The queue used inside the *BFS* is the one from the Python's library.

When we happen to find the exit, then we will return the distance that we counted so far, else we will print "no" when our queue is empty and there are no other cells to traverse.

Our adjacent cells aren't stored anywhere, so from line `33` to `41` we take care to populate the array of adjacents by checking if their coordinate fits within the grid.

Finally, this is the code that will handle the parsing of the input from string to our meaningful data structure:

```
53 n,m = [int(x) for x in input().split()]
54 g = Grid(n,m)
55 for i in range(n):
56     string = str.lower(input())
57     row = []
58     for j in range(m):
59         if string[j] == "o":
60             cell = Cell('white')
61         elif string[j] == "x":
62             cell = Cell('black')
63         elif string[j] == "s":
64             cell = Cell('gray', False, True)
65             g.start = cell
66         else:
67             cell = Cell('white', True, False)
68         cell.y = i
69         cell.x = j
70         row.append(cell)
71     g.addRow(i, row)
```

Starting from an empty graph, we will add a cells to it according to the given input, compiling each column for each row of our grid and making our preprocessing time complexity in the order of $\mathcal{O}(n \cdot m)$.

Input:

```
1 3 3
```

```
2 osx
```

addRow(0, [00, 10, 20]) →

00	10	02

```
3 xoo
```

addRow(1, [01, 11, 21]) →

00	10	02
01	11	21

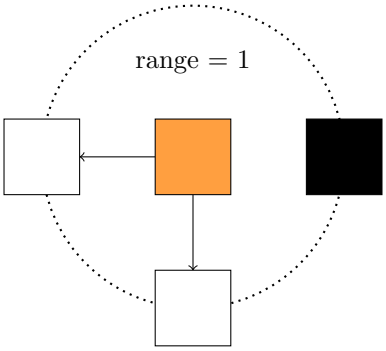
```
4 eox
```

addRow(2, [02, 12, 22]) →

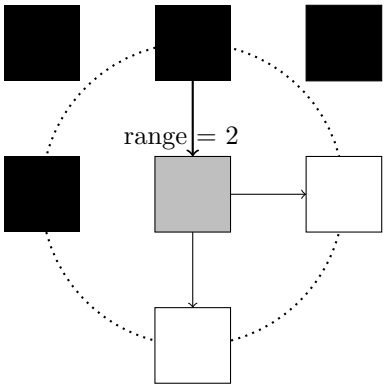
00	10	02
01	11	21
02	12	22

After our input has been preprocessed, the *BFS* will be initialised from our start cell, the reference to which has been saved within our grid data structure. From there, it will traverse all its adjacent cells and the adjacent of those, until it hits the exit cell. The blocked cells have been flagged as "already visited" in order for our *BST* to ignore them.

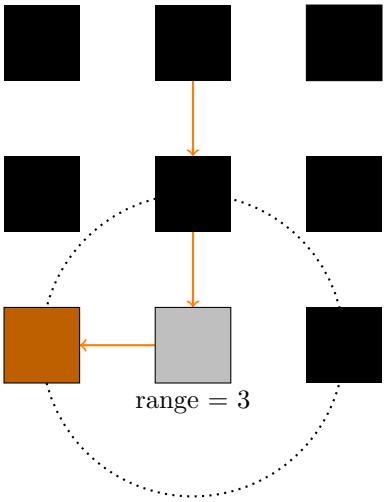
Initially, the BFS will start from the start cell that we had defined in the Grid and start to look for other adjacent cells:



We first visit the top left cell but it does not lead anywhere, the top right isn't accessible so we recurse down on the central cell:



The central left cell is blocked, whereas the central right cell does not lead anywhere. Moreover the cell from which we came from can't be revisited, thus we go down the central bottom cell:



Finally, an exit cell was found next to the current one, thus there is a solution to our maze and the shortest path has been found at range 3!

Complexity Analysis

The expected time complexity of the program is $\mathcal{O}(n \cdot m)$, since the preprocessing part will take $\mathcal{O}(n \cdot m)$, because we will need to create our adjacency matrix, that is the maze grid that we build from scratch that contains all `None` cells and then inserting all our cells, which is going to take $2(n \cdot m)$ operations. Then our BFS is, according to theory, going to take linear time $n + m$.
This means that our complexity will be $\mathcal{O}(n \cdot m)$.

Credits

Both problems were approached alongside **Cristian Buratti** and **Riccardo Corrias** the same day the assignment came out. Starting from 11PM we have worked throughout the whole night and managed to solve the assignment by 5AM. As always, we did not share any code among ourselves, however we extensively discussed about the best ways to tackle the problems and we cross checked our programs by running some random inputs and helping in debugging each other's code.

Given Help

I helped **Alessandra Vicini** in the development of her solution, as she had troubles implementing her own ideas into working code.