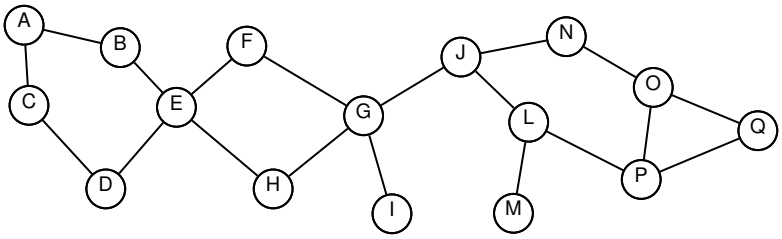
PRÀCTICA 6 - GRAPHS

CURS 2015-2016

Figura 1: Connected unweighed undirected graph



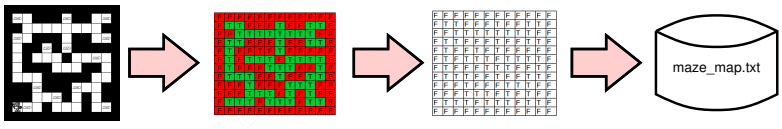
This exercise is about Graphs and is separated in 4 principal tasks. In the first task you will use the robot to create a map of the maze. You will extend the brain from exercise 4 and through backtracking we will create a map of all accessible blocks in the maze and store it to a file. In the second task you will implement a Graph class in python. The Graph class will be able to read the map generated by the robot and use it to set its nodes and vertices. The third task consists of implementing a Breadth First Search algorithm so that you can compute the shortest path between any pair of nodes in the Graph. The fourth task is about figuring out whether a graph is connected or not.

1 Task 1: Occupancy matrix and Brain 6

1.1 Introduction

The first task of the exercise is to create a brain for pyrobot named Brain_6.py which will generate a boolean matrix containing a map of the maze using the previously implemented backtracking algorithm. Brain_6.py should extend Brain_4.py which navigated through the maze with back tracking. In figure **??** you can see an outline of task 1.

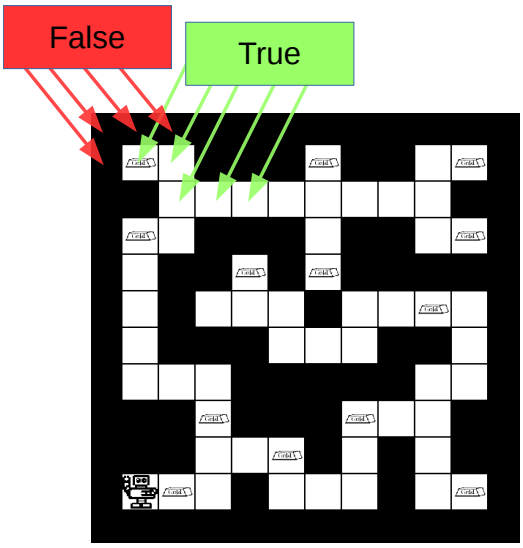
Figura 2: From maze to occupancy matrix



The goal is to store the blocks of the maze that are accessible by the robot in a matrix with the size of the maze. Blocks should be marked as 1 (True) when they are accessible and 0 (False) when they are not. Such a boolean matrix is called occupancy matrix. In Figure **??** the correspondence between the maze blocks and the boolean matrix can be seen. The specific maze can be encoded as a matrix of 12×12.

In Listing **??** a python expression of the occupancy matrix of the maze seen in Figure **??**

Figura 3: Boolean values from maze



Listing 1: Python representation of the occupancy matrix from the pyrobot maze

```
[[False, False, False, False, False, False, False, False, False, False, False, False],
 [False, True , True , False, False, False, True , False, False, True , True , False],
 [False, False, True , True , True , True , True , True , True , True , False, False],
 [False, True , True , False, False, False, True , False, False, True , True , False],
 [False, True , False, False, True , False, True , False, False, False, False, False],
 [False, True , False, True , True , True , False, True , True , True , True , False],
 [False, True , False, False, False, True , True , True , False, False, True , False],
 [False, True , True , True , False, False, True , False, False, True , True , False],
 [False, False, False, True , False, False, False, True , True , True , False, False],
 [False, False, False, True , True , True , False, True , False, True , False, False],
 [False, True , True , True , False, True , True , True , False, True , True , False],
 [False, False, False, False, False, False, False, False, False, False, False, False]]
```

1.2 Brain_6.py

In order to record the occupancy matrix and save it, you must create Brain_6.py by incorporating the logic from the Brain_4.py in to the provided skeleton of Brain_6 provided in listing **??**

Listing 2: Tree.py

```
from math import *
from CStack import * #change this to the name of the stack you implemented

import CPilot;CPilot=reload(CPilot) #change this to the name of the pilot you
    implemented

from pyrobot.brain import Brain
from pyrobot.brain import avg
import os

```

```
class WB(Brain):
    def saveMap(self,fname):
        """
        Method that stores a boolean matrix as a text file.
        :param fname: a string containing the full path to the filename
        """
        bool2str={True:'T',False:'F'}
        rows=[''.join([bool2str[col] for col in line]) for line in self.__map]
        open(fname,'w').write('\n'.join(rows))

    def setup(self):
        # Adding functionality for recording the map
        self.cols = 12 #same as world cols
        self.rows = 12 #same as world rows
        self.__map = [[0 for y in range(self.cols)] for x in range(self.rows)]
        yx = self.robot.getItem('location')
        self.__map[self.rows-yx[1]][yx[0]-1]= 1
        #fill in

    def step(self):
        if not(self.robot.getItem('win')):
            #fill in
            pass

        else:
            self.saveMap(os.path.dirname(os.path.realpath(__file__))+'/maze_map.txt')

def INIT(engine):
    return WB('WB', engine)
```

The file Brain_6.py should define a class WB which defines 3 methods.

- method **saveMap**:
  A method used to save the data member **self.__map** to a text file. This method is provided fully implemented. The file format is to have one character ('T' or 'F') for each element of a matrix row. Matrix rows are separated with a newline character.

- method **setup**:
  This method is used to initialise the brain but it is not a constructor. This method is partially filled so that the **self.__map** member can be created and initialised. You have to add the remaining of the setup function from the previous Brain_4.py.

- method **step**:
  This method must implement the backtracking algorithm exactly as in Brain_4.py. In addition to the previous functionality, the method must also update **self.__map** so that every square of the matrix that corresponds to a position visited by the robot is set to True.

1.3 Hints

The best strategy for solving this part of the exercise is to use Brain_4.py as a basis and add the code provided in listing **??**. The test case jocproves_graph.py will test if the file maze_map.txt that should be saved from Brain_6.py exists and is correct.

In practice, after merging Brain_4.py and the Brain_6.py skeleton provided here, all you should do is update the map every time you move and save it once you have won.

2 Task 2: Graph from Occupancy Matrix

For this exercise you will have to implement in python a class providing the basic functionality of graph.

2.1 Brief overview of graphs

According to theory Graphs can be modelled in many different ways and grouped in to several categories:

- **Directed vs Undirected:**
  The directed graph is the case where a connection from one node to an other doesn't guaranty the same connection will exist on the opposite direction: $A \rightarrow B \not\Rightarrow B \rightarrow A$.
  When a graph is undirected, a vertex between node *A* and *B* is always symmetric. In undirected directed graphs, always $A \rightarrow B \Rightarrow B \rightarrow A$ and therefore vertices are marked as simple lines between the nodes.

- **Weighted vs Unweighted:**
  In the most simple case vertices between nodes either exist or they don't, such a model is called an unweighed graph. An example of an unweighted graph is course prerequisites in a study guide. For more complicated problems graph vertices can be described by a numerical value; such graphs are called weighted graphs. A typical example of weighted graphs is the resistance in electrical circuits.

- **Graphs vs Multigraphs:**
  In graphs there can be at most a single vertex between two nodes.
  Graphs that allow more than one vertex between two nodes on the same direction are called multigraphs.

- **Connected vs Disconnected :**
  When for a graph exists a path from every node to every other node, the graph is considered to be connected. When there exists even one pair of nodes for which there is no path connecting them, the graph is considered disconnected.

Depending on the type of Graph we are using different representations and different algorithms are required. In this exercise the class Graph is supposed to represent an unweighed undirected graph that is possibly disconnected. An efficient representation of this type of graph is to keep a list of all directly connected nodes for every node. Although a bit redundant, this representation allows for storing directed graphs as well and simplifies the implementation of several algorithms. In python more specifically this would be easily be stored as a dictionary with the node names as keys and and python array with the names of the direct neighbours as the value for each key.

In Listing **??** the python dictionary containing the graph in Figure **??** can be seen.

Listing 3: Adjagency Map in python

```
adjagency_map = {'A':['B','C'],
                 'B':['A','E'],
                 'C':['A','D'],
                 'D':['C','E'],
                 'E':['B','D','F','H'],
                 'F':['E','G'],
                 'G':['F','H','I','J'],
                 'H':['E','G'],
                 'I':['G'],
                 'J':['G','L','N'],
                 'L':['J','M','P'],
                 'M':['L'],
                 'N':['J','O'],
                 'O':['N','P','Q'],
                 'P':['L','O','Q'],
                 'Q':['O','P']}
```

Listing 4: Graph.py

```
class Graph:
    def __init__(self,adjacent=[]):
        self.adj=dict(adjacent)


    def getNodes(self):
        nodes=self.adj.keys()
        nodes.sort()
        return nodes

    def createFromBoolMatrix(self,mat):
        def pos2name(y,x):
            return 'p_'+str(x)+'_'+str(y)
        self.adj={}
        #Fill in

    def loadFromBoolMatrix(self,fname):
        mat=[[col=='T' for col in line.strip()] for line in open(fname).read().split('\n
    ') if len(line.strip())]
        self.createFromBoolMatrix(mat)


    def getBFS(self,p1Label,p2Label):
        pass
        #Fill in


    def isConnected(self):
        pass
        #Fill in


    def __repr__(self):
        adjList=self.adj.items()
        for a in adjList:
            a[1].sort()
        adjList.sort()
```

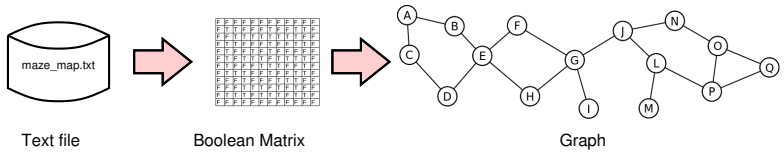
```
37          return 'Graph('+str(adjList)+')'
38
39      def __eq__(self, other):
40          return repr(self)==repr(other)
```

2.2 Graph.py

Tasks 2, 3 and 4 are about implementing methods for a class graph. In Listing **??** the skeleton of this class is provided. Other than the three methods you have to implement, several other methods are provided implemented.

2.3 Creating a graph from an occupancy matrix

Figura 4: Overview of task 2



2.3.1 Introduction

A grid-like structure, such as an occupancy matrix, can be converted to a graph. Each element of the matrix becomes a node of the graph. Each node of the grid lies directly to left, on top, the right, or bellow an element is considered connected and a vertex is created between the neighbouring nodes. When only these neighbours are considered, it's called the 4-neighbourhood. There exist other types of neighbourhoods, but they do not concern this exercise as the robot can only move in these directions. In the case where the grid is fully accessible as the pyrobot maze, there are several ways to encode the obstacles in the graph. The solution of ignoring all nodes that are marked as false in the matrix is the most simple strategy.

2.3.2 Task 2 description

In this task you have to take the occupancy matrix and convert it to an adjacency list. In practice this means you have to implement the method **`Graph.createFromBoolMatrix(self,mat)`**. This method takes as input parameter **`mat`** which should have a form equivalent to what we see in Listing **??** and should set the data member **`self.__adj`** to contain the graph that corresponds to that matrix in the form that is shown in Listing **??**.

The Graph class should be able to load from any file following the same specification. Several files containing different graphs than the one your brain has to create are provided: maze_map_circle.txt, maze_map_edge.txt, maze_map_empty.txt, and maze_map_noedge.txt.

2.3.3 Hints

Because the graph is organised as a dictionary, node names have to be guaranteed to be unique. The auxiliary function `pos2name(y,x)` returns a string that is unique for every combination of coordinates x and y. For example, the node referring to the element of the matrix at the third column and second row in the matrix will become `'p_2_1'`, because of zero-indexing.

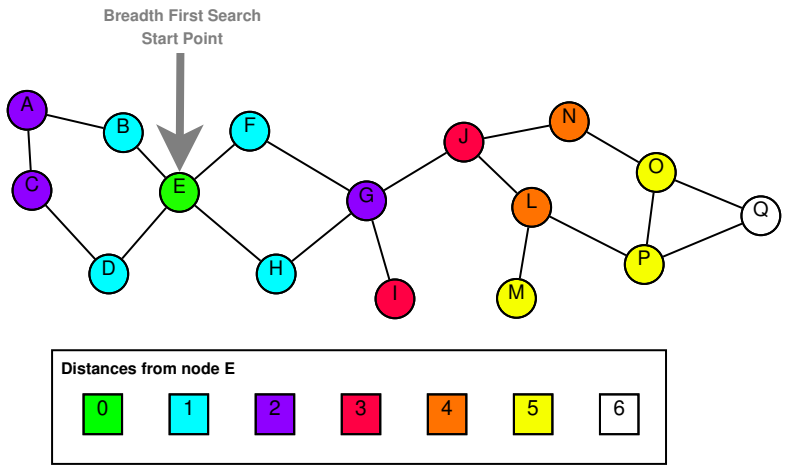
The boolean matrix is a python array containing other arrays of a fixed length. In order to address for example the element at the second row and third column all is needed is `mat[1][2]`

The most convenient way to access all elements in the 4-neighbourhood of position `(x,y)` is to iterate over a python array, containing the offsets towards the element and its neighbours `[[0,1],[1,0],[0,-1],[-1,0]]` and add its values to `(x,y)`.

3 Task 3: Breadth First Search (BFS)

In this task you have to implement a shortest path algorithm. The algorithm should be able to compute for every node the distance of the shortest path to all other nodes in the graph if such a path exists. The algorithm should also be able to reply which is the shortest path for every two nodes. Since the graph is unweighted, the Breadth First Search is an optimal algorithm to solve the problem. In Figure **??** you can see the distances between node *E* and all other nodes.

Figura 5: Example of Breadth First Search



You should implement the method `Graph.getBFS(self,node1,node2)` which takes two nodes in the graph as input and returns a tuple with the length of the shortest path between nodes named `node1` and `node2`. If there doesn't exist a path between the two nodes the method should return `(-1,[])` and if the two node names are the same, the method should return `(0,[node1])`.

4 Task 4: Graph connectivity

A graph is connected when for each pair of nodes exists a path between the two of them. In this task you have to implement the method `Graph.isConnected(self)`. This method should return `True` if the graph is connected and `False` otherwise. An empty graph is also considered connected.

5 Grading

For a perfect score in class, you have to finish: the occupancy graph generation, the auxiliary functions of the graph, and loading the matrix in to a graph. For the final submission you have to submit all parts of the exercise in optimal and well formatted and documented code. The total grade is a weighted sum of the percentage you obtain in class and the grade from the final submission.

Where	Points	Job
Classroom	2.0	Occupancy map creation: **`Brain_6.py`** and **`maze_map.txt`** .
Classroom	0.5	class Graph: Method **`Graph.getChilds(self)`**.
Classroom	3.0	Occupancy map to Graph: Method**`Graph.createFromBoolMatrix(self, map)`**.
Home	3.5	Shortest path in Graph: Method**`Graph.getBFS(self, map)`**.
Home	1.0	Connectivity of Graph: Method**`Graph.isConnected(self)`**.

$$total\_grade = 0.4 * (classroom\_work/5.5) + 0.6 * (final\_submission/10.0)$$

6 Other information

If you need tutoring contact anguelos@cvc.uab.es to arrange for a tutoring session (in English) before the 19th of December.