Practical Assignment № 4

Manuel Pita, Universidade Lusófona April 5, 2020

Problem 1: Thinking with code about uninformed search

During the lectures in the topic of Artificial Intelligence and Search Algorithms we studied how we can explore an entire search graph when we have no information about what paths are better to expand than others. In the previous practical session we implemented a basic form of these algorithms (see Listing 1). In this context, what could make a path better than other? And in that same line of thought, what makes an uninformed search algorithm stupid? The goal of this first problem is to run a few simulations, think about everything that was said in the theory lecture with a critical mindset and use the algorithms you are implementing to think with.

1. What sort of indicator measures could you collect from the execution of uninformed search algorithms to think about how stupid they are?
2. Sometimes DFS is referred to as the *thief's path.* Why do you think this is the case?
3. DFS is said to backtrack. What is this? Does BFS backtrack too?
4. Before DFS and BFS there is another way of getting the paths between an initial node and a goal node in a search graph which is known as the *British Museum* algorithm. Find out what that is and explain it clearly. Complete the implementation that allows to get it in Listing 2.

Problem 2: Visited nodes and open paths

During the lecture on uninformed search we spoke about the possibility that these algorithms get stuck going around in forever loops, which of course we do not want. We discussed two different add-on mechanisms to deal with loops. The first approach, let us call it *Approach A,* depends on a variable v where we keep the nodes that have been processed as heads of the queue. When any expansion–that will add new nodes to the queue–leads to a node in the list of visited nodes that expansion is not added to the queue. On the other hand, we implemented a different version of this by keeping in the queue, not only the nodes to visit in the future, but the open paths that lead to those nodes from the initial node. Let us call this method *Approach B*. In this case what we do is that when the open path of a node has a repeated node, we do not add that expansion to the queue. A basic implementation of the code that implements Approach A is given below in Listing 1.

1. Validate the implementation of Approach A (Listing 1), and using it as a basis, implement and validate Approach B. To validate, create at least five different graphs including some with more nodes and more links

2. What is the difference between the two mechanisms in terms of what happens when they are executed? How can you show the difference between these two approaches? Hint: simulate various problems and analyse behaviour of variables such as the queue and others

3. Would it make sense to implement an Approach C that integrates approaches A and B? Think, justify, implement and explain.

Listing 1: A version of uninformed search that marks visited nodes and does not expand paths going through them once visited.

```
1  def uninformed_search_a(graph, start, goal, option):
2      q = [start]
3      v = []
4      # option 0: DFS
5      # option 1: BFS
6      enq = 1
7      i=0 # vamos manter um contador da iteracao na qual estamos
8      while q:
9          i+=1
10         print('current q ', q)
11         h = q[0] #primeiro separo a q em cabeca e resto
12         r = q[1:]
13         print('iteration', i)
14         print('q size starts', enq)
15         if h == goal: #pergunto se a cabeca e o no objetivo
16             print('found') # se e, acabou o programa
17             break
18         else:
19             enq = enq - 1
20             v.append(h) # se nao e, marco o h como "visitado"
21             e = [] # e faco a expansao do h, mas nao considerando qualquer no ↩
                   filho que ja tenha sido visitado
22             for node in graph[h]:
23                 if node not in v:
24                     e.append(node)
25                     enq += 1
26             if option == 0: # DFS
27                 q = sorted(e)+r
28                 print('q size end it', len(q))
29             else: #BFS
30                 q = r+sorted(e)
31                 print('q size end it', len(q))
32         print('- - - - ')
```

Listing 2: Partial implementation to study the British Museum approach to uninformed search.

```
def bm_paths(graph, start, goal):
    q = [(start, [start])]
    while q:
        (vertex, path) = # here I have to break the q, read and rest. Use pop.
        for next in graph[vertex] - set(path):
            if next == goal:
                # here I want to show a path. If I return, only one is shown think about yield

            else:
                # add the code to updat the queue
```

Problem 3: Visualising the algorithms

One invaluable way to analyse and understand different search algorithms and add-ons that we add to them is to visualise their execution, both in trivial problems we can work out by hand to validate that they are working; but also in more complex problems, to see how they compare to each other.

1. Do some research about the uses of the package `networkx` to visualise graphs
2. Write a function `build_nx_graph` that takes as argument an a graph represented as an adjacency list and returns a graph we can visualise in `networkx`
3. Play around with the different options to draw nodes and edges and find out how could you dynamically update these properties to show what a search algorithm is doing
4. From the items above, figure out how to use `networkx` to create a visual animation of the execution of your search algorithms, show your start node in green, your end goal in red, all the edges in black or gray, and the path currently explored with thick red edges. Keep in mind that this would have to change in every iteration and that we can to see it as an animation. Research, implement, validate and explain