# Tutorial 2

## Name: Zhaokun Su

## Breakout group: 4

## Zid: z5235878

**(i) Try running the various search algorithms to make sure you understand their properties (as above), their implementations and the relationships between the algorithms. You can edit searchTest.py to call the search methods on the Romania map problem.**

Using the Romania map, we could call this 5 below search methods on this problem:

A screenshot of a cell phone

Description automatically generated

**(ii)  The supplied code adds successors to the frontier in the order they are defined in the associated definition of the Romania map in searchProblem.py. By experimenting with different orderings of the successors (such as alphabetical ordering, as above), determine how sensitive the algorithms are to this ordering.**

Before changing the orderings, the result is shown in the (i) part.

We can change ordering in specific Arc list by adding the following code in the searchGeneric.py file, and we sort every node’s neighbors by their **Alphabetical order**.

A screenshot of a cell phone

Description automatically generated

Rerunning this program to test our method:

We get a different result as following:

A screenshot of a cell phone

Description automatically generated

The program searching method for DFS seems like going to infinite loop and never end, which means if we change our neighbor nodes order to alphabetical order, DFS searching will not find a final solution for this problem.

Eventually, we terminate our program by ^C:

A screenshot of a cell phone

Description automatically generated

In conclusion, for this tutorial code, the **Arc order matters and ordering is sensitive** **to our searching appoarch, especially for DFS.**

**(iii)  (Harder) Although breadth-first search is a special case of A∗ search (explain how!), write a class BreadthFirstSearcher extending Searcher in searchGeneric.py that implements breadth-first search directly, i.e. maintains a set of states from nodes pre- viously expanded and only adds a neighbour of an expanded node to the frontier if its state is not in the explored set or in a node already on the frontier, and terminates when a goal state is generated (not expanded). You will need to code the constructor and the search method.**

Write an extra class BFSearcher (Inherited class Searcher):

class BFSearcher(Searcher):  
 def \_\_init\_\_(self, problem):  
 super().\_\_init\_\_(problem)  
 self.frontier\_city = []  
 self.expanded\_city = []  
 self.add\_to\_frontier\_city(Path(problem.start\_node()).end())  
  
 def add\_to\_frontier\_city(self, city):  
 self.frontier\_city.append(city)  
  
 def add\_to\_expanded\_city(self, city):  
 self.expanded\_city.append(city)

And we define BFS method like below: city means nodes, which is evey city passing through:

@visualize  
def search(self):  
 path = Path(None)  
 while not self.empty\_frontier():  
 path = self.frontier.pop(0)  
 self.add\_to\_expanded\_city(path.end())  
  
 self.frontier\_city.remove(path.end())  
 self.display(2, "Expanding:", path, "(cost:", path.cost, ")")  
 self.num\_expanded += 1  
  
 neighbors = self.problem.neighbors(path.end())  
 self.display(3, "Neighbors are", neighbors)  
  
 for arc in neighbors:  
 if (arc.to\_node not in self.expanded\_city) or (arc.to\_node in self.frontier\_city):  
 self.add\_to\_frontier(Path(path, arc))  
 self.frontier\_city.append(Path(path, arc).end())  
 self.display(3, "Frontier:", self.frontier)  
 if self.problem.is\_goal(arc.to\_node): # solution found  
 self.display(1, self.num\_expanded, "paths have been expanded and",  
 len(self.frontier), "paths remain in the frontier")  
 self.solution = Path(path, arc)  
 return Path(path, arc)

Finally, it turns out that BFS costs is 450.

And we find an anther path which passes through only 4 cities.

A screenshot of text

Description automatically generated