ICE – 2
UNINFORMED SEARCH

Search algorithms can be classified into two types:

1. **Uninformed search algorithms**:
2. **Informed search algorithms**:

CONTENTS

OVERVIEW

Uninformed search algorithms: Search algorithms which explore the search space without having any information about the problem other than its definition.

- Breadth First Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Search
- Uniform Cost Search

PROBLEM

psource(Problem)

```
class Problem:
def __init__(self, initial, goal=None):
def actions(self, state):
def result(self, state, action):
def goal_test(self, state):
def path_cost(self, c, state1, action, state2):
def value(self, state):
```

The abstract class for a formal problem. You should subclass this and implement the methods actions and result, and possibly __init__, goal_test, and path_cost. Then you will create instances of your subclass and solve them with the various search functions.

The constructor specifies the initial state, and possibly a goal state, if there is a unique goal. Your subclass's constructor can add other arguments.

Return the actions that can be executed in the given state. The result would typically be a list, but if there are many actions, consider yielding them one at a time in an iterator, rather than building them all at once Return the state that results from executing the given action in the given state. The action must be one of self.actions

Return the cost of a solution path that arrives at state2 from state1 via action, assuming cost c to get up to state1. If the problem is such that the path doesn't matter, this function will only look at state2. If the path does matter, it will consider c and maybe state1 and action. The default method costs 1 for every step in the path.

For optimization problems, each state has a value. Hill Climbing and related algorithms try to maximize this value

NODE

A node in a search tree. Contains a pointer to the parent (the node that this is a successor of) and to the actual state for this node. Note that if a state is arrived at by two paths, then there are two nodes with the same state. Also includes the action that got us to this state, and the total path_cost (also known as g) to reach the node. Other functions may add an f and h value; see best_first_graph_search and astar_search for an explanation of how the f and h values are handled. You will not need to subclass this class

psource(Node)

```
class Node:
def __init__(self, state, parent=None, action=None, path_cost=0):

def expand(self, problem):
def child_node(self, problem, action):
def solution(self):
def path(self):
```

These 4 methods override standards Python functionality for representing an object as a string, the less-than (<) operator, the equal-to (=) operator, and the hash function.

```
def __repr__(self):
def __lt__(self, node):
def __eq__(self, other):
def __hash__(self):
```

Abstract class Problem to define our real **problem** named GraphProblem. You can see how we define GraphProblem by running the next cell.

psource(GraphProblem)

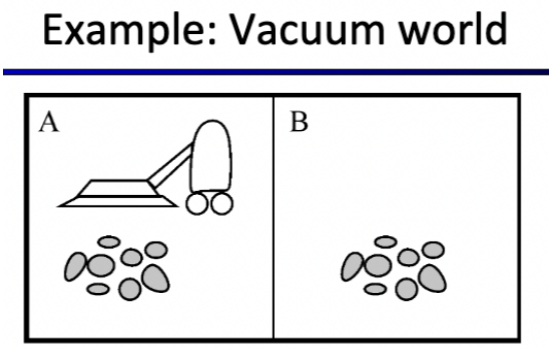
```
class GraphProblem(Problem):
def __init__(self, initial, goal, graph):
def actions(self, A):
def result(self, state, action):
def path_cost(self, cost_so_far, A, action, B):
def find_min_edge(self):
def h(self, node):
```

SIMPLE PROBLEM SOLVING AGENT PROGRAM

psource(SimpleProblemSolvingAgentProgram)

```
class SimpleProblemSolvingAgentProgram:
def __init__(self, initial_state=None):
def __call__(self, percept):
def update_state(self, state, percept):
def formulate_goal(self, state):
def formulate_problem(self, state, goal):
def search(self, problem):
```

- Example: VacuumAgent



Percepts: [location,status], e.g., [A,Dirty]
Actions: Left, Right, Suck, NoOp

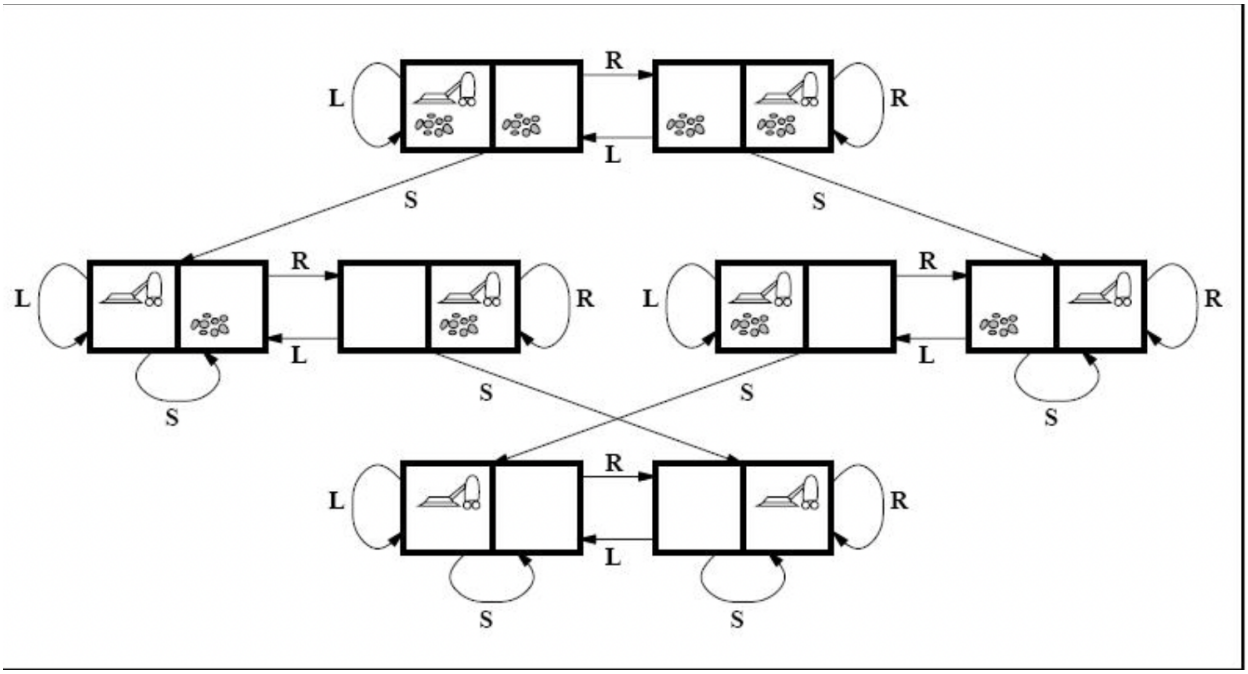
Agent function

Percept sequence	Action
[A,Clean]	Right
[A,Dirty]	Suck
[B,Clean]	Left
[B,Dirty]	Suck
[A,Clean],[B,Clean]	Left
[A,Clean],[B,Dirty]	Suck
etc	etc

Agent program

```
function Reflex-Vacuum-Agent([location,status])
         returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left
```

REPRESENTATION OF 8 DIFFERENT STATES



Here in state 1 both the columns are dirty
state1 = [(0, 0), [(0, 0), "Dirty"], [(1, 0), ["Dirty"]]]

```
state2 = [(1, 0), [(0, 0), "Dirty"], [(1, 0), ["Dirty"]]]
state3 = [(0, 0), [(0, 0), "Clean"], [(1, 0), ["Dirty"]]]
state4 = [(1, 0), [(0, 0), "Clean"], [(1, 0), ["Dirty"]]]
state5 = [(0, 0), [(0, 0), "Dirty"], [(1, 0), ["Clean"]]]
state6 = [(1, 0), [(0, 0), "Dirty"], [(1, 0), ["Clean"]]]
state7 = [(0, 0), [(0, 0), "Clean"], [(1, 0), ["Clean"]]]
state8 = [(1, 0), [(0, 0), "Clean"], [(1, 0), ["Clean"]]]
```

```
class vacuumAgent(SimpleProblemSolvingAgentProgram):
      def update_state(self, state, percept):
def formulate_goal(self, state):
search(self, problem):
          if problem == state1:
             seq = ["Suck", "Right", "Suck"]
          elif problem == state2:
             seq = ["Suck", "Left", "Suck"]
          elif problem == state3:
             seq = ["Right", "Suck"]
          elif problem == state4:
             seq = ["Suck"]
          elif problem == state5:
             seq = ["Suck"]
          elif problem == state6:
             seq = ["Left", "Suck"]
```

RESULT is all states

Suck
Right
Suck
Suck
Suck
Left
Suck

VISUALIZATION:

1. **Un-explored:** The which we did not approach
2. **Frontier:** Successor nodes to currently exploring node

3. **Currently exploring:** The node which is to be compared with
4. **Explored:** node which is already explored and visited
5. **Final Solution:** The goal of getting the shortest distance.

BREADTH FIRST TREE AND BREADTH FIRST GRAPH SEARCH:

BFS is a method for searching a tree data structure for a node that meets a set of criteria. It begins at the root of the tree and investigates all nodes at the current depth level before moving on to the nodes at the next depth level. To maintain track of the child nodes that have been encountered but not yet investigated, more memory, usually in the form of a queue, is required.

DEPTH FIRST TREE AND DEPTH FIRST GRAPH SEARCH:

The depth-first search (DFS) algorithm is used to traverse or explore data structures such as trees and graphs. The algorithm starts at the root node (in the case of a graph, any random node can be used as the root node) and examines each branch as far as feasible before retracing.

Uniform-cost search:
Uniform-cost search is an uninformed search algorithm that finds a path from the source to the destination by calculating the lowest cumulative cost. Starting at the root, nodes are expanded according to the lowest cumulative cost. A Priority Queue is then used to implement the uniform-cost search.

Depth limited search:

The new search algorithm for uninformed searches is depth limited search. The depth-first search algorithm encounters the unbounded tree problem, which can be solved by setting a boundary or limit on the depth of the search space.

Iterative deepening search:

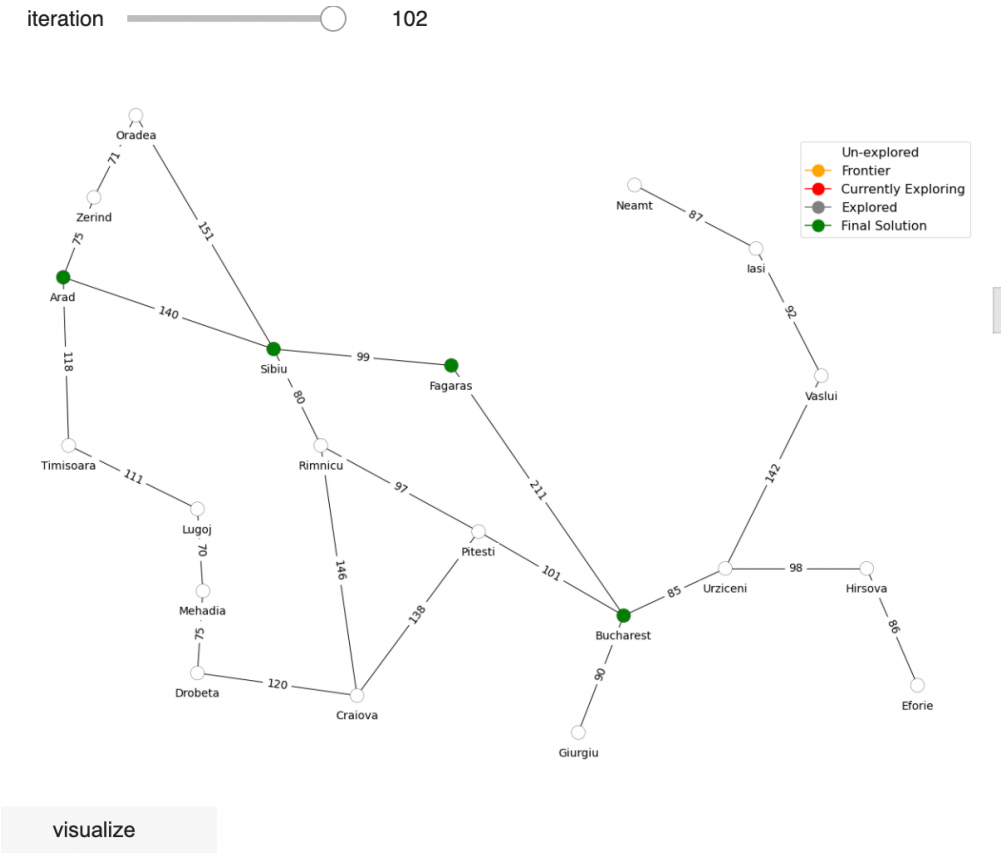
Iterative deepening search (IDS or IDDFS) is a state space/graph search approach in which a depth-limited version of depth-first search is run repeatedly with increasing depth limitations until the target is reached. IDDFS is as good as breadth-first search but consumes a lot less memory; each iteration explores the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are visited first is essentially breadth-first

Undirected Graph - Romania_map

TASK 2

Run and analyze the whole code that had been included in the assignment, Understand the code, and explain the working mechanism of each part of the code.

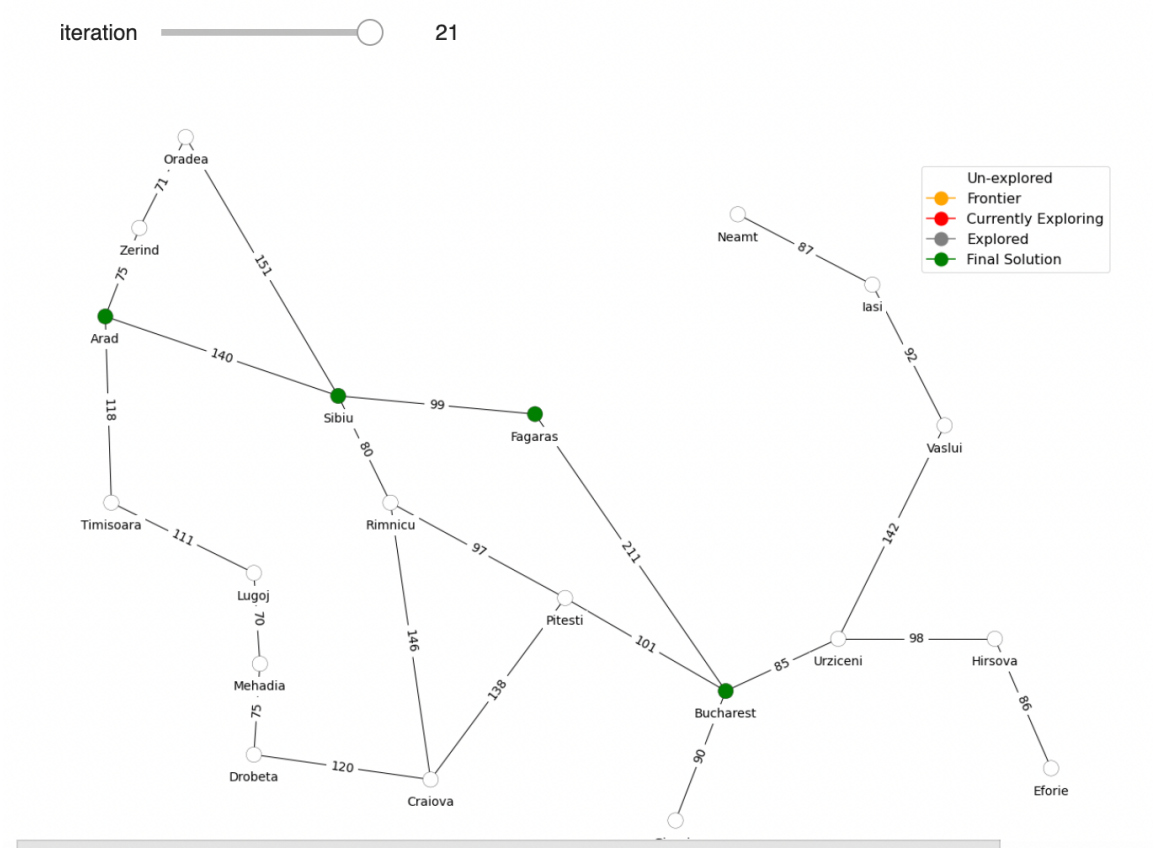
1. Breadth First Tree Search



2. Depth First Tree Search
3. Breadth First Graph Search

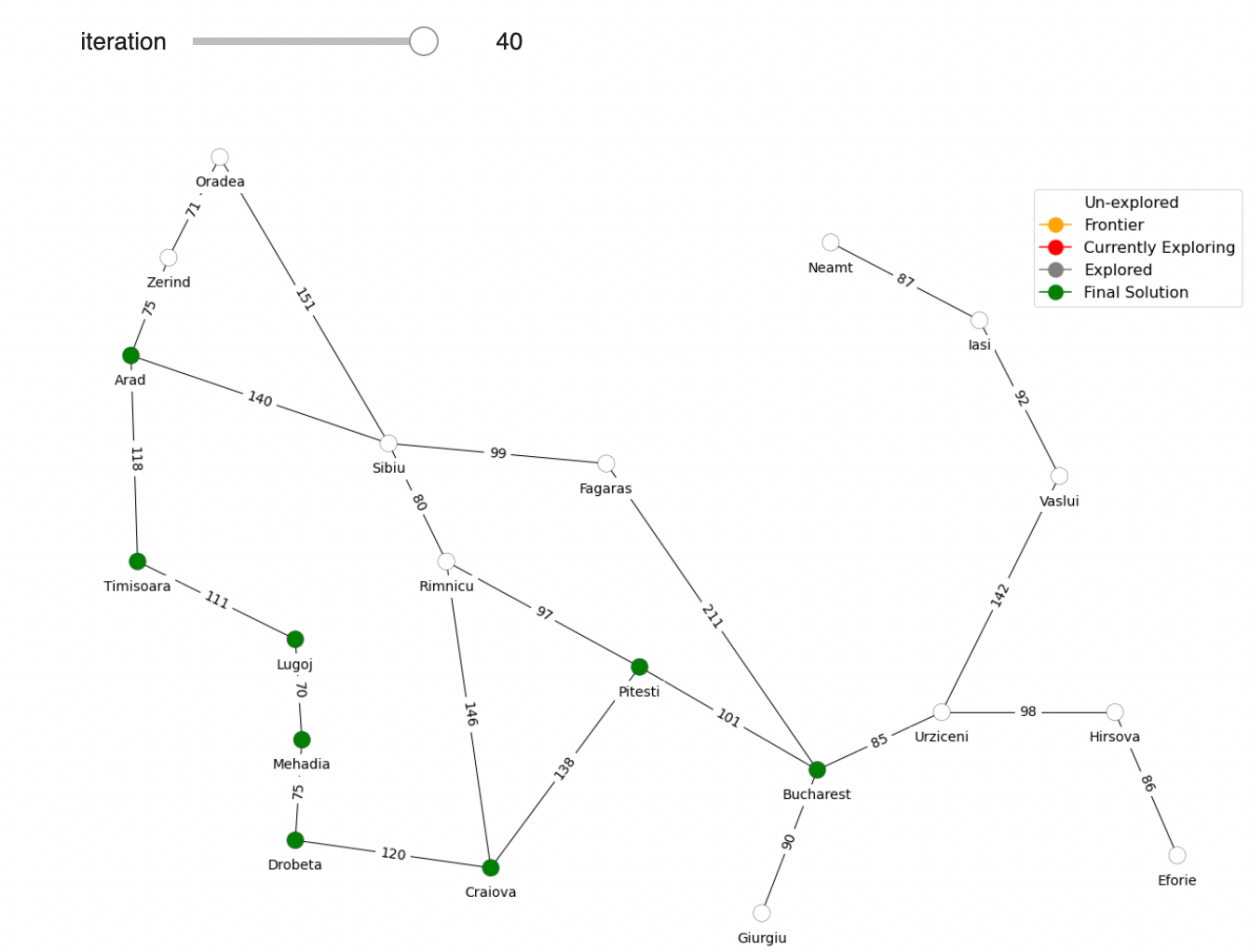
4.

```
algorithm=breadth_first_search_graph,
problem=romania_problem)
```



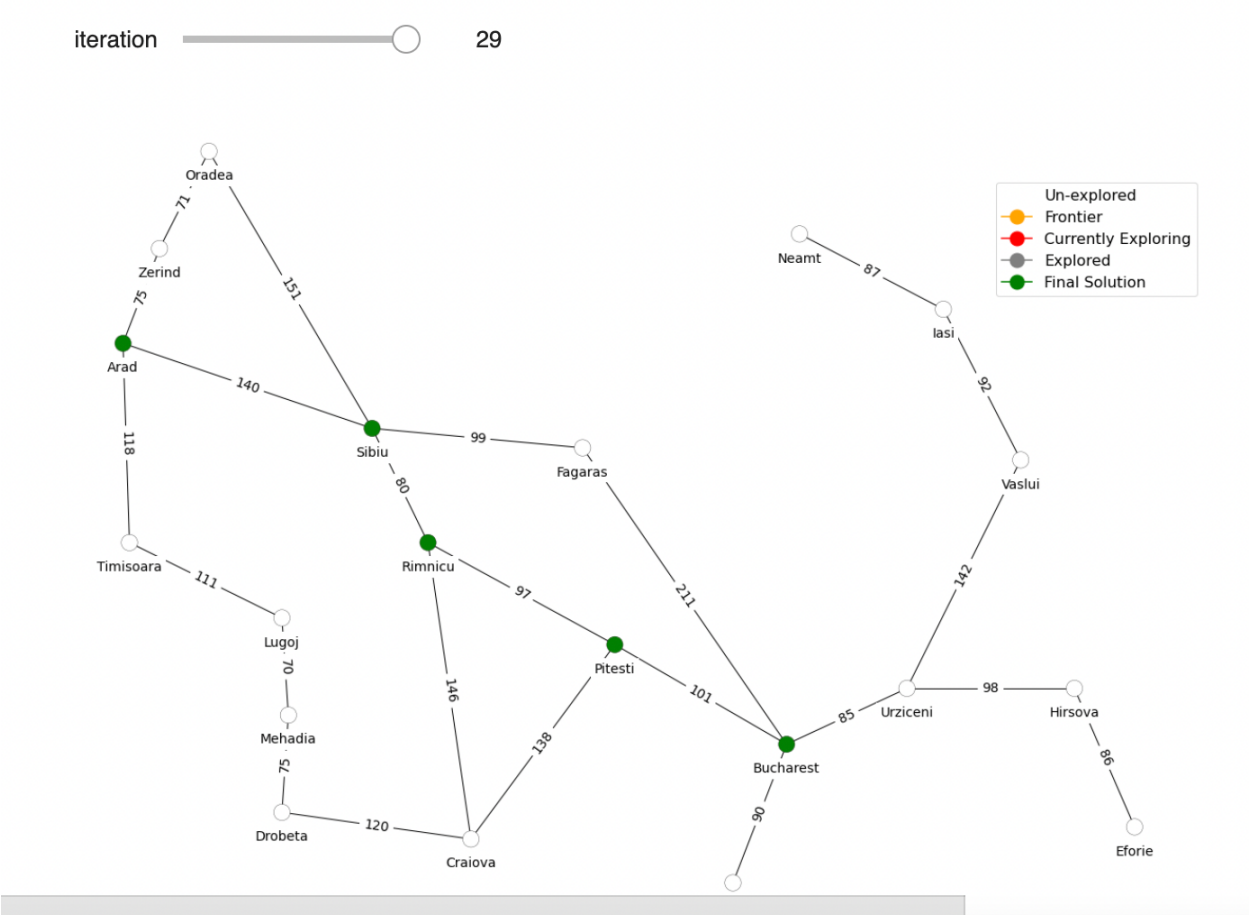
5. Depth First Graph Search

```
algorithm=depth_first_graph_search,
problem=romania_problem)
```



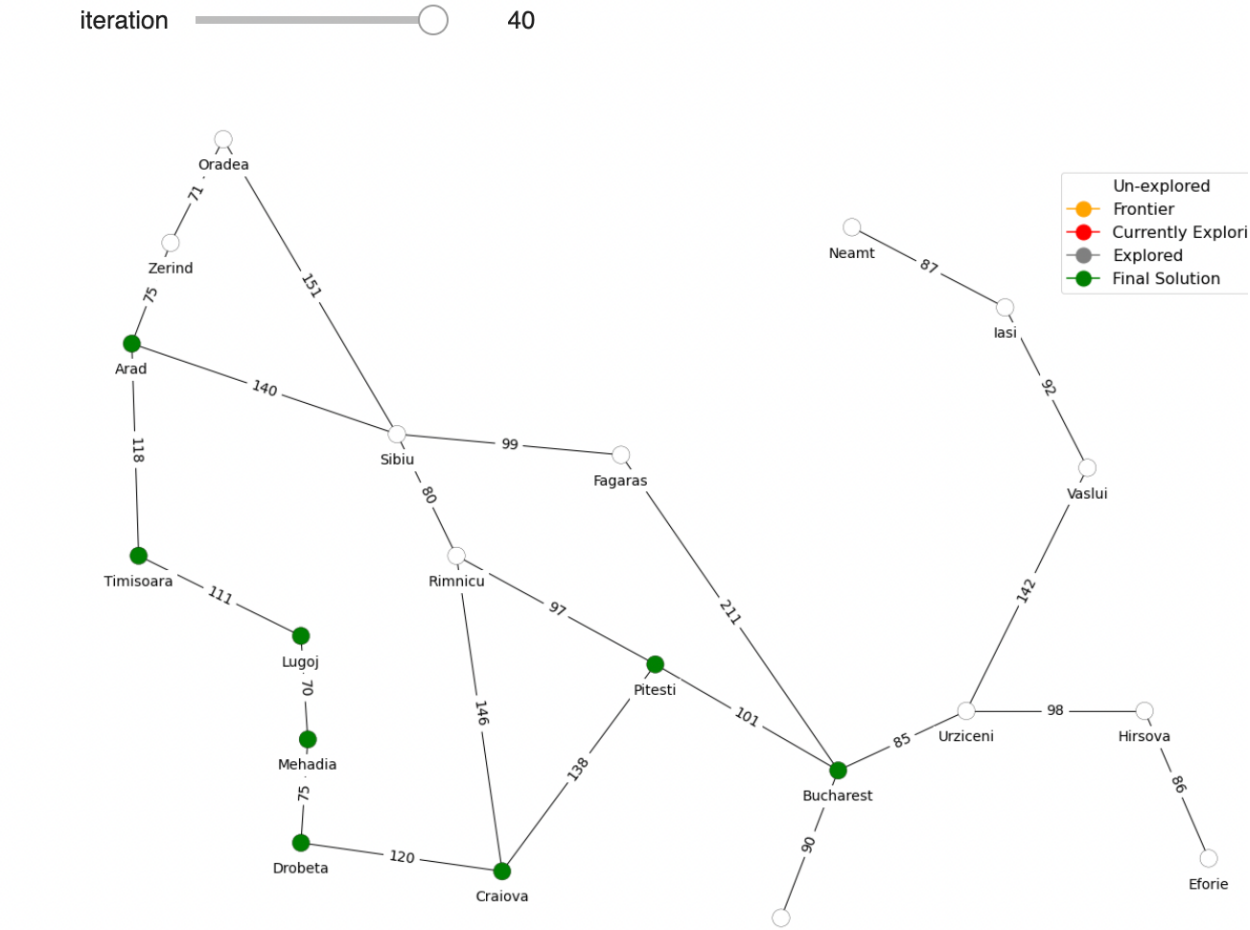
6. Uniform Cost Search

```
display_visual(romania_graph_data, user_input=False,
               algorithm=uniform_cost_search_graph,
               problem=romania_problem)
```



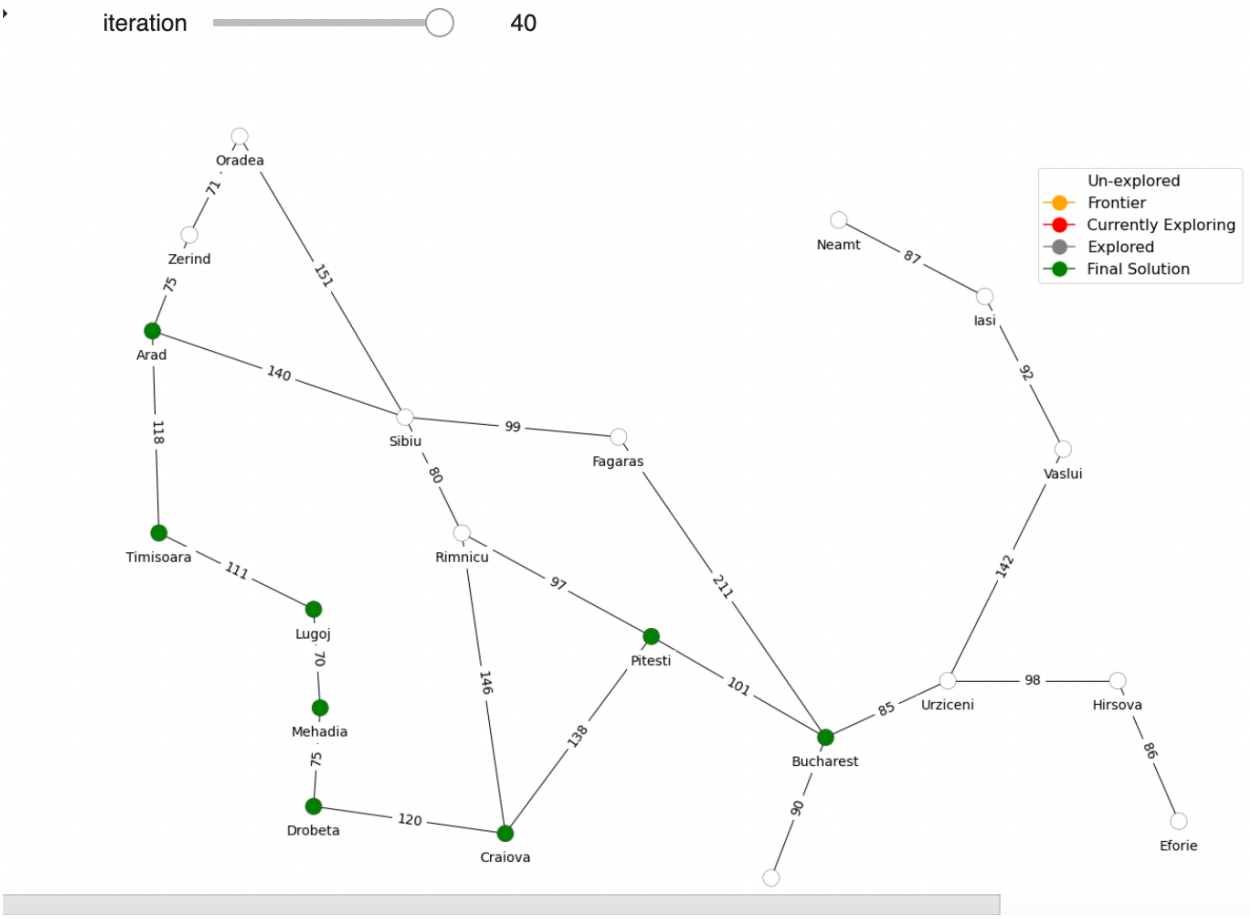
7. Depth Limited Search

```
                algorithm=depth_limited_search_for_vis,
                problem=romania_problem)
```



8. Iterative Deepening Search

```
display_visual(romania_graph_data, user_input=False,
               algorithm=iterative_deepening_search_for_vis,
               problem=romania_problem)
```



TASK 3

visualization part

SEARCHING ALGORITHMS VISUALIZATION

1. Breadth First Tree Search
2. Depth First Tree Search
3. Breadth First Search
4. Depth First Graph Search
5. Uniform Cost Search
6. Depth Limited Search
7. Iterative Deepening Search

TASK 4- SANTA_BARBARA _MAP

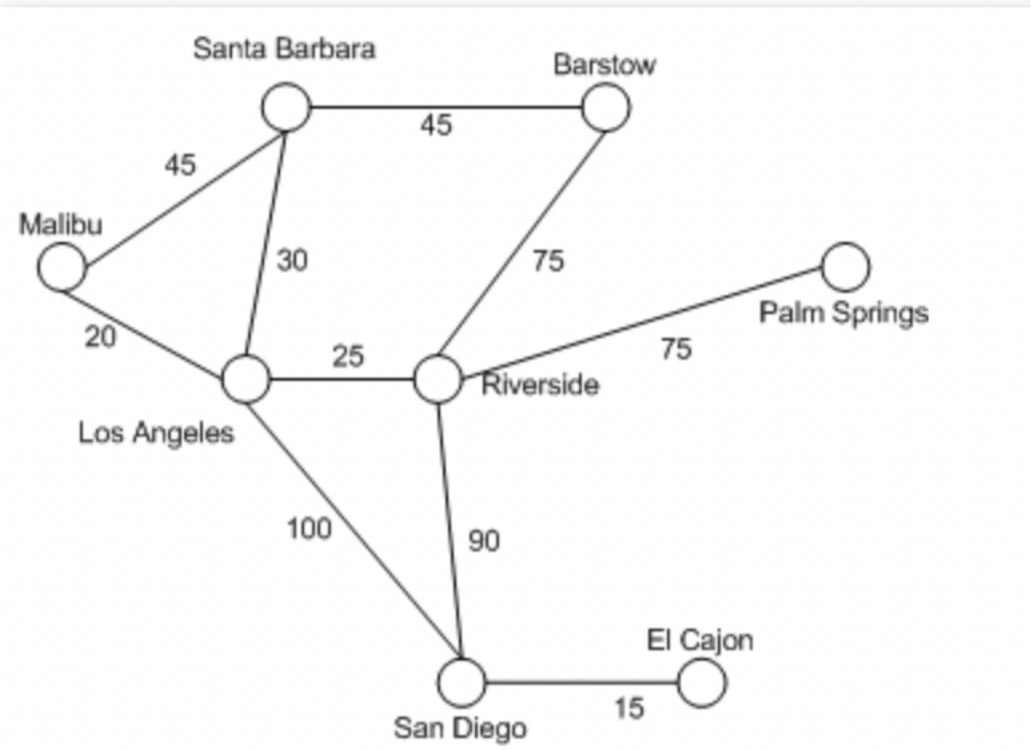
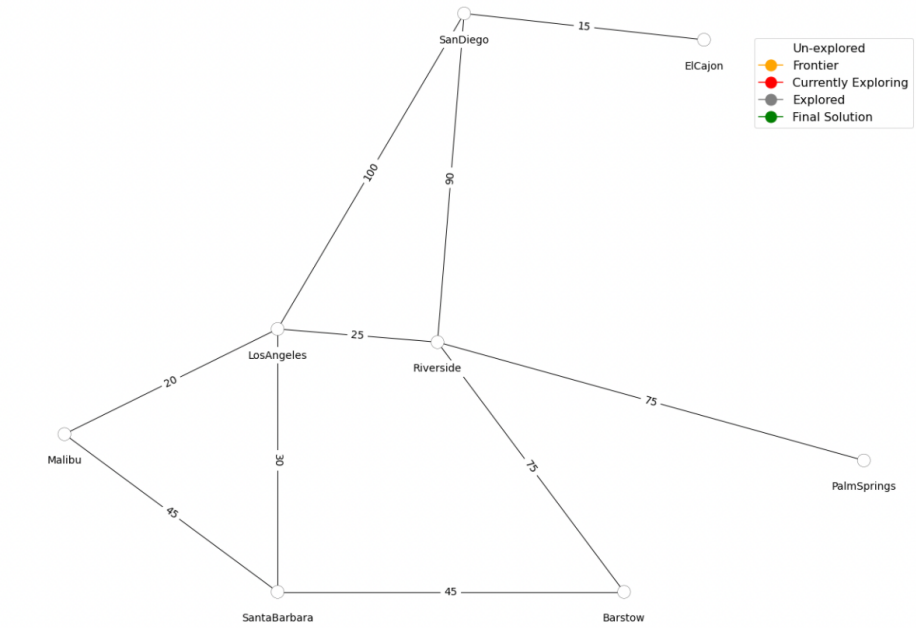
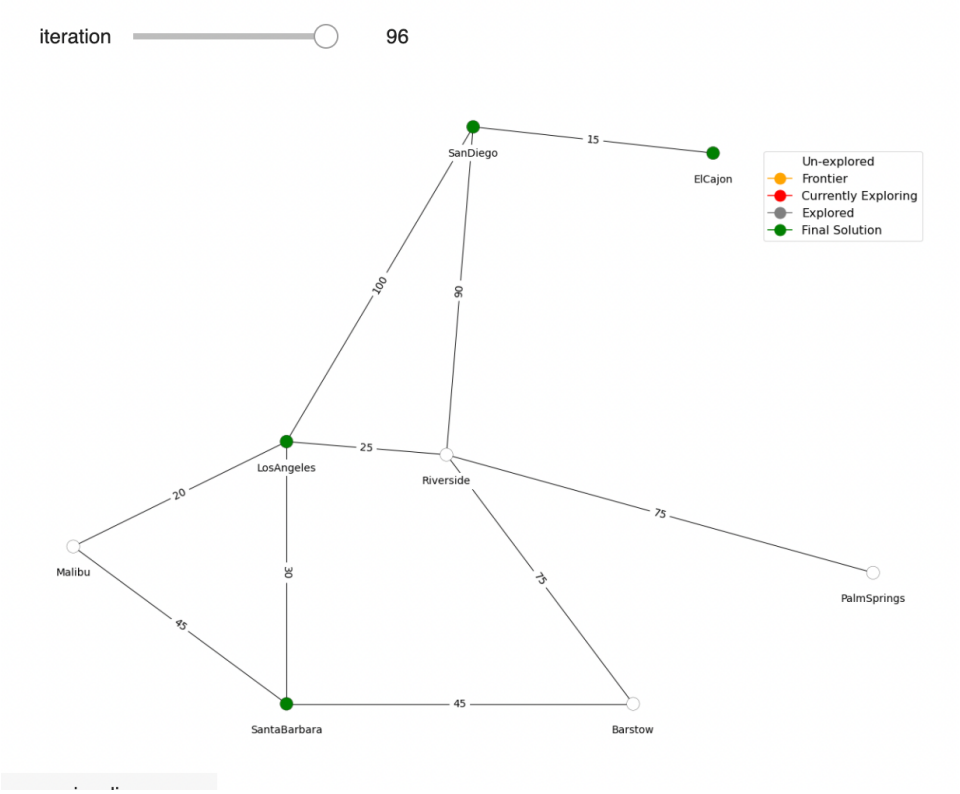


Fig 1: santa_barbara_map



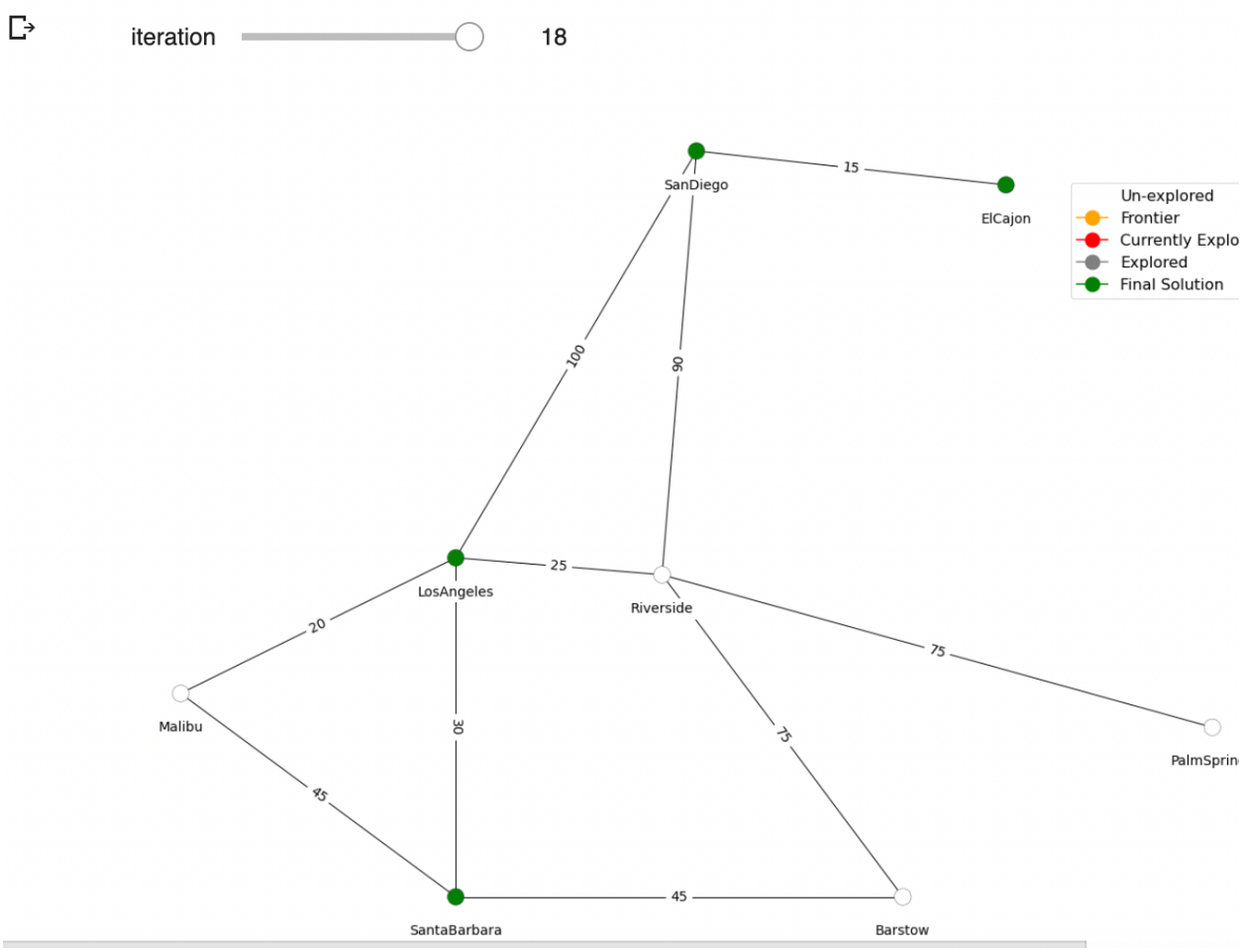
1. **BREADTH FIRST TREE SEARCH**

```
                algorithm=breadth_first_tree_search,
                problem=santa_barbara_problem)
```



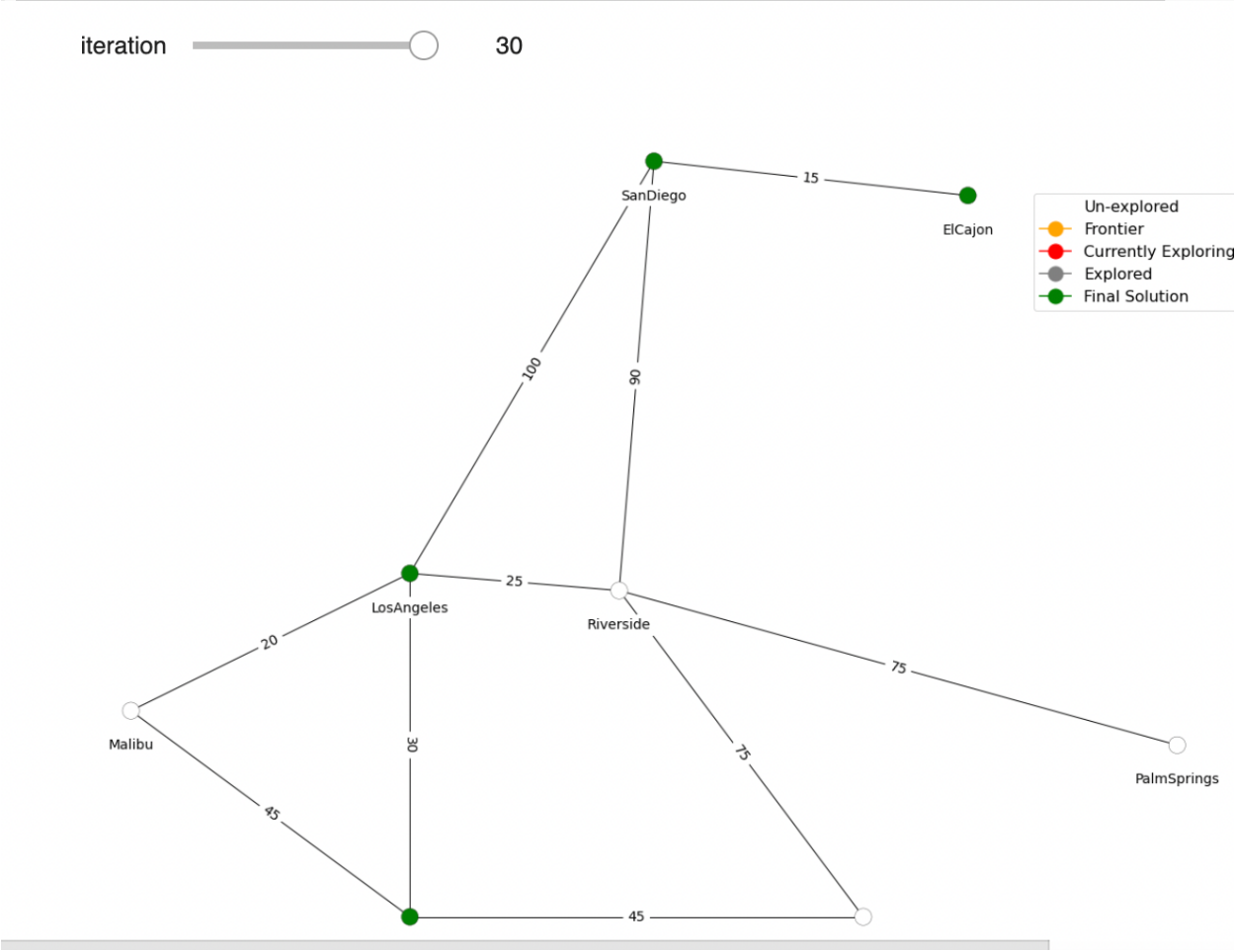
UNIFORM COST SEARCH

```
algorithm=uniform_cost_search_graph,
problem=santa_barbara_problem)
```



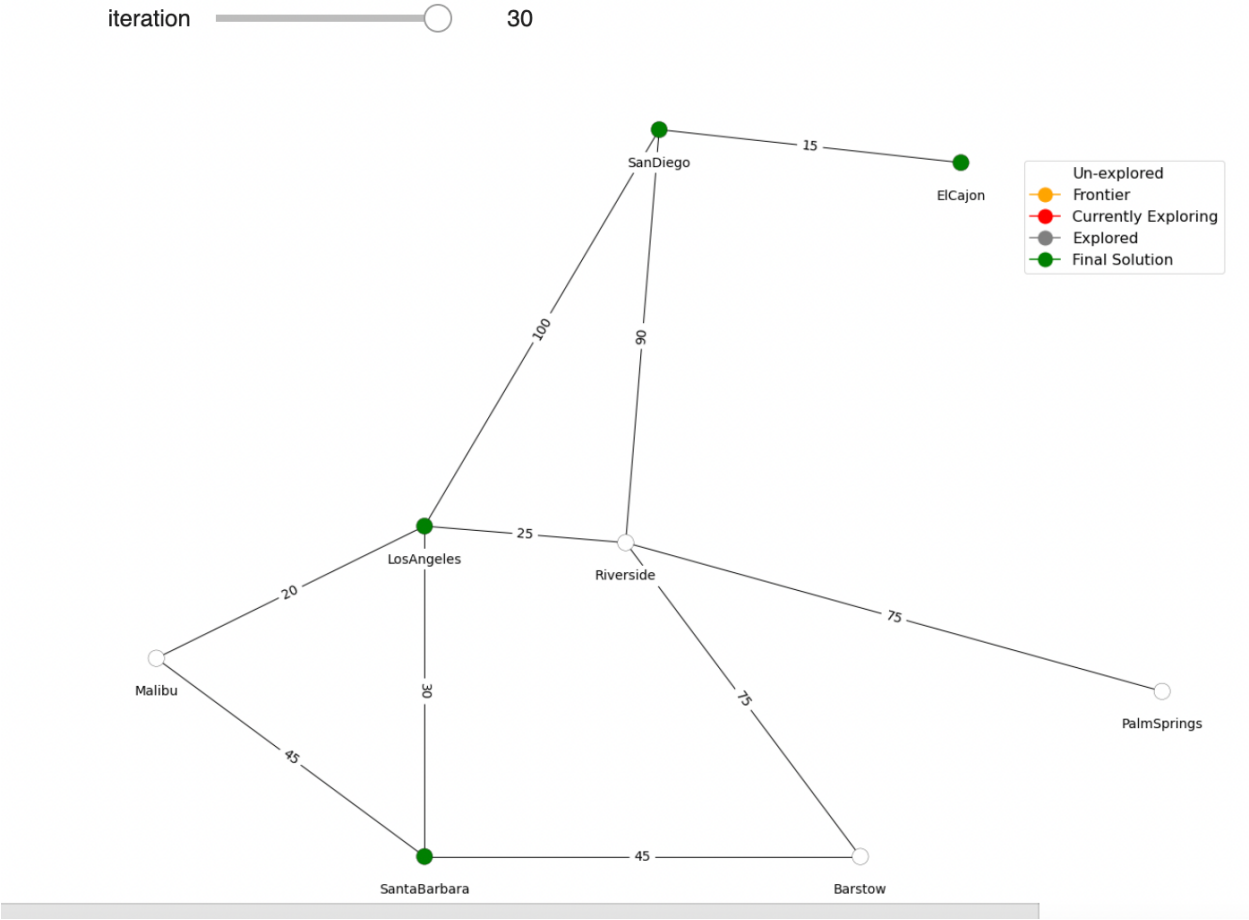
DEPTH LIMITED SEARCH:

```
                algorithm=depth_limited_search_for_vis,
                problem=santa_barbara_problem)
```



ITERATIVE DEEPENING SEARCH

```
algorithm=iterative_deepening_search_for
problem=santa_barbara_problem)
```



Task 5 – BREST_MAP

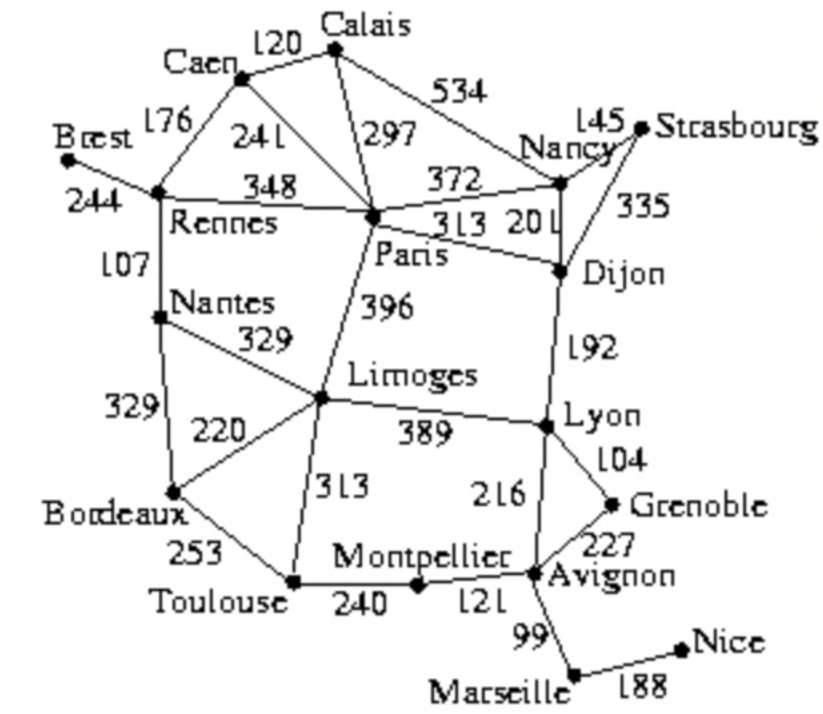
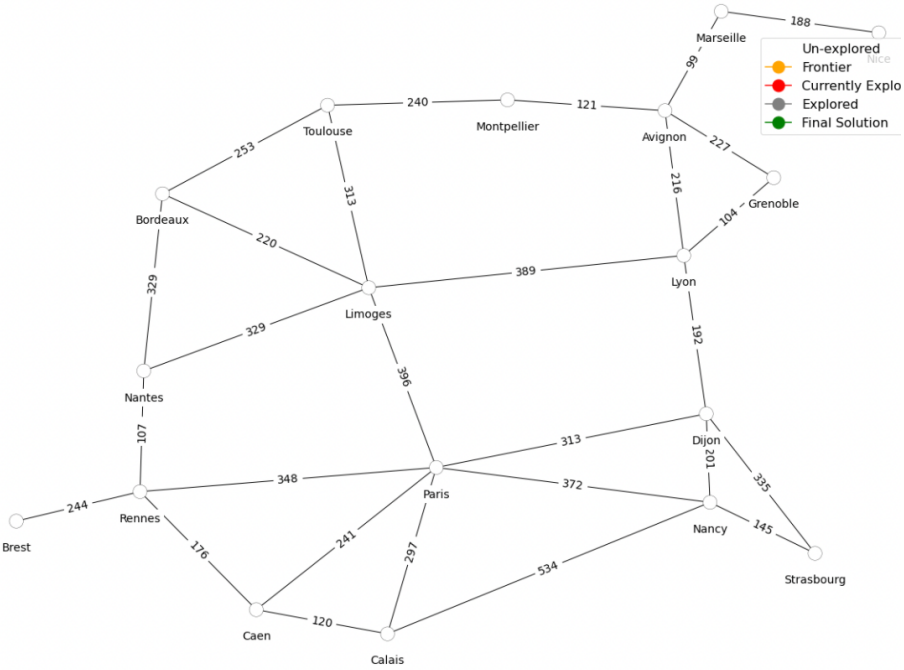


Fig 2: brest_map

```
[ ]
```

Your session crashed after using all available RAM. If you are interested in access to high-RAM runtimes, you may want to check out Colab Pro.

View runtime logs



Similarly as above we observe different paths to brest map.