Solving problems by Searching

This notebook serves as supporting material for topics covered in **Chapter 3 - Solving Problems by Searching** from the book *Artificial Intelligence: A Modern Approach*. This notebook uses implementations from search.py module. Let's start by importing everything from search module.

```
from google.colab import drive  #mount the code
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

```
cd /content/drive/MyDrive/Colab Notebooks/aima-python-master
```

```
/content/drive/MyDrive/Colab Notebooks/aima-python-master
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dr
```

```
pip install qpsolvers #run it
```

```
Requirement already satisfied: qpsolvers in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: quadprog>=0.1.8 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: scipy>=1.2.0 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (
```

```
pip install ipythonblocks  #run it
```

```
Requirement already satisfied: ipythonblocks in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: requests>=1.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipython>=4.0 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: notebook>=4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pexpect in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/py
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/dist
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.7/dis
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7/dist
Requirement already satisfied: nbformat in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-packag
```

```
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter-core>=4.4.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: jupyter-client>=5.2.0 in /usr/local/lib/python3.7
Requirement already satisfied: pyzmq>=13 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/di
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/l
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: MarkupSafe>=0.23 in /usr/local/lib/python3.7/dist
Requirement already satisfied: entrypoints>=0.2.2 in /usr/local/lib/python3.7/di
Requirement already satisfied: testpath in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: mistune<2,>=0.8.1 in /usr/local/lib/python3.7/dis
Requirement already satisfied: pandocfilters>=1.4.1 in /usr/local/lib/python3.7/
Requirement already satisfied: defusedxml in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: bleach in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/python3
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dis
Requirement already satisfied: importlib-resources>=1.4.0 in /usr/local/lib/pyth
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/di
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: packaging in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: webencodings in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python
```

```
pip install -r requirements.txt #run it
```

```
Requirement already satisfied: cvxopt in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: image in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: ipython in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: ipythonblocks in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: ipywidgets in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: keras in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: networkx in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: opencv-python in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pillow in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: pytest-cov in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: qpsolvers in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: sortedcontainers in /usr/local/lib/python3.7/dist
Requirement already satisfied: tensorflow in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: django in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/dist-p
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/py
```

```
Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: pexpect in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/dist
Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.7/dis
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: requests>=1.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: notebook>=4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: nbformat in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: nbconvert in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: Send2Trash in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter-core>=4.4.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: terminado>=0.8.1 in /usr/local/lib/python3.7/dist
Requirement already satisfied: tornado>=4 in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: jupyter-client>=5.2.0 in /usr/local/lib/python3.7
Requirement already satisfied: ipykernel in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.7/dist
Requirement already satisfied: jinja2 in /usr/local/lib/python3.7/dist-packages 
Requirement already satisfied: pyzmq>=13 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/di
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/l
Requirement already satisfied: ptyprocess in /usr/local/lib/python3.7/dist-packa
Requirement already satisfied: widgetsnbextension~=3.5.0 in /usr/local/lib/pytho
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /usr/local/lib/pytho
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /usr/local/lib/python3
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dis
Requirement already satisfied: importlib-resources>=1.4.0 in /usr/local/lib/pyth
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/di
Requirement already satisfied: pyrsistent!=0.17.0,!=0.17.1,!=0.17.2,>=0.14.0 in 
Requirement already satisfied: attrs>=17.4.0 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: zipp>=3.1.0 in /usr/local/lib/python3.7/dist-pack
Requirement already satisfied: qtconsole in /usr/local/lib/python3.7/dist-packag
```

```
from search import *
from notebook import psource, heatmap, gaussian_kernel, show_map, final_path_colors,

# Needed to hide warnings in the matplotlib sections
import warnings
warnings.filterwarnings("ignore")
```

CONTENTS

- Overview
- Problem
- Node
- Simple Problem Solving Agent
- Search Algorithms Visualization

- Breadth-First Tree Search
- Breadth-First Search
- Depth-First Tree Search
- Depth-First Search
- Uniform Cost Search

OVERVIEW

Here, we learn about a specific kind of problem solving - building goal-based agents that can plan ahead to solve problems. In particular, we examine navigation problem/route finding problem. We must begin by precisely defining **problems** and their **solutions**. We will look at several general-purpose search algorithms.

Search algorithms can be classified into two types:

- **Uninformed search algorithms**: Search algorithms which explore the search space without having any information about the problem other than its definition.
  - Examples:
    1. Breadth First Search
    2. Depth First Search
    3. Depth Limited Search
    4. Iterative Deepening Search
    5. Uniform Cost Search

Don't miss the visualisations of these algorithms solving the route-finding problem defined on Romania map at the end of this notebook.

For visualisations, we use networkx and matplotlib to show the map in the notebook and we use ipywidgets to interact with the map to see how the searching algorithm works. These are imported as required in `notebook.py`.

```
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines

from ipywidgets import interact
import ipywidgets as widgets
from IPython.display import display
import time
```

PROBLEM

Let's see how we define a Problem. Run the next cell to see how abstract class `Problem` is defined in the search module.

```
psource(Problem)
```

```
class Problem:
    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
```

The `Problem` class has six methods.

- `__init__(self, initial, goal)` : This is what is called a `constructor`. It is the first method called when you create an instance of the class as `Problem(initial, goal)`. The variable `initial` specifies the initial state s_0 of the search problem. It represents the beginning state. From here, our agent begins its task of exploration to find the goal state(s) which is given in the `goal` parameter.
- `actions(self, state)` : This method returns all the possible actions agent can execute in the given state `state`.
- `result(self, state, action)` : This returns the resulting state if action `action` is taken in the state `state`. This `Problem` class only deals with deterministic outcomes. So we know for sure what every action in a state would result to.
- `goal_test(self, state)` : Return a boolean for a given state - `True` if it is a goal state, else `False`.
- `path_cost(self, c, state1, action, state2)` : Return the cost of the path that arrives at `state2` as a result of taking `action` from `state1`, assuming total cost of `c` to get up to `state1`.
- `value(self, state)` : This acts as a bit of extra information in problems where we try to optimise a value when we cannot do a goal test.

```
        else:
```

NODE

Let's see how we define a Node. Run the next cell to see how abstract class `Node` is defined in the search module.

```
        state2. If the path does matter, it will consider c and maybe state1
```

```
psource(Node)
```

```python
    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."""

    def __init__(self, state, parent=None, action=None, path_cost=0):
        """Create a search tree Node, derived from a parent by an action."""
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost
        self.depth = 0
        if parent:
            self.depth = parent.depth + 1

    def __repr__(self):
        return "<Node {}>".format(self.state)

    def __lt__(self, node):
        return self.state < node.state

    def expand(self, problem):
        """List the nodes reachable in one step from this node."""
        return [self.child_node(problem, action)
                for action in problem.actions(self.state)]

    def child_node(self, problem, action):
        """[Figure 3.10]"""
        next_state = problem.result(self.state, action)
        next_node = Node(next_state, self, action, problem.path_cost(self.path_c
        return next_node

    def solution(self):
        """Return the sequence of actions to go from the root to this node."""
        return [node.action for node in self.path()[1:]]

    def path(self):
        """Return a list of nodes forming the path from the root to this node."'
        node, path_back = self, []
        while node:
            path_back.append(node)
            node = node.parent
        return list(reversed(path_back))

    # We want for a queue of nodes in breadth_first_graph_search or
    # astar_search to have no duplicated states, so we treat nodes
    # with the same state as equal. [Problem: this may not be what you
    # want in other contexts.]

    def __eq__(self, other):
        return isinstance(other, Node) and self.state == other.state

    def __hash__(self):
        # We use the hash value of the state
        # stored in the node instead of the node
        # object itself to quickly search a node
        # with the same state in a Hash Table
```

The `Node` class has nine methods. The first is the `__init__` method.

- `__init__(self, state, parent, action, path_cost)` : This method creates a node. `parent` represents the node that this is a successor of and `action` is the action required to get from the parent node to this node. `path_cost` is the cost to reach current node from parent node.

The next 4 methods are specific `Node`-related functions.

- `expand(self, problem)` : This method lists all the neighbouring(reachable in one step) nodes of current node.
- `child_node(self, problem, action)` : Given an `action`, this method returns the immediate neighbour that can be reached with that `action`.
- `solution(self)` : This returns the sequence of actions required to reach this node from the root node.
- `path(self)` : This returns a list of all the nodes that lies in the path from the root to this node.

The remaining 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.

- `__repr__(self)` : This returns the state of this node.
- `__lt__(self, node)` : Given a `node`, this method returns `True` if the state of current node is less than the state of the `node`. Otherwise it returns `False`.
- `__eq__(self, other)` : This method returns `True` if the state of current node is equal to the other node. Else it returns `False`.
- `__hash__(self)` : This returns the hash of the state of current node.

We will use the abstract class `Problem` to define our real **problem** named `GraphProblem`. You can see how we define `GraphProblem` by running the next cell.

```
psource(GraphProblem)
```

```python
class GraphProblem(Problem):
    """The problem of searching a graph from one node to another."""

    def __init__(self, initial, goal, graph):
        super().__init__(initial, goal)
        self.graph = graph

    def actions(self, A):
        """The actions at a graph node are just its neighbors."""
        return list(self.graph.get(A).keys())

    def result(self, state, action):
        """The result of going to a neighbor is just that neighbor."""
        return action

    def path_cost(self, cost_so_far, A, action, B):
        return cost_so_far + (self.graph.get(A, B) or np.inf)

    def find_min_edge(self):
        """Find minimum value of edges."""
        m = np.inf
        for d in self.graph.graph_dict.values():
            local_min = min(d.values())
            m = min(m, local_min)

        return m

    def h(self, node):
        """h function is straight-line distance from a node's state to goal."""
```

Have a look at our romania_map, which is an Undirected Graph containing a dict of nodes as keys and neighbours as values.

```python
romania_map = UndirectedGraph(dict(
    Arad=dict(Zerind=75, Sibiu=140, Timisoara=118),
    Bucharest=dict(Urziceni=85, Pitesti=101, Giurgiu=90, Fagaras=211),
    Craiova=dict(Drobeta=120, Rimnicu=146, Pitesti=138),
    Drobeta=dict(Mehadia=75),
    Eforie=dict(Hirsova=86),
    Fagaras=dict(Sibiu=99),
    Hirsova=dict(Urziceni=98),
    Iasi=dict(Vaslui=92, Neamt=87),
    Lugoj=dict(Timisoara=111, Mehadia=70),
    Oradea=dict(Zerind=71, Sibiu=151),
    Pitesti=dict(Rimnicu=97),
    Rimnicu=dict(Sibiu=80),
    Urziceni=dict(Vaslui=142)))

romania_map.locations = dict(
    Arad=(91, 492), Bucharest=(400, 327), Craiova=(253, 288),
    Drobeta=(165, 299), Eforie=(562, 293), Fagaras=(305, 449),
    Giurgiu=(375, 270), Hirsova=(534, 350), Iasi=(473, 506),
```

```
    Lugoj=(165, 379), Mehadia=(168, 339), Neamt=(406, 537),
    Oradea=(131, 571), Pitesti=(320, 368), Rimnicu=(233, 410),
    Sibiu=(207, 457), Timisoara=(94, 410), Urziceni=(456, 350),
    Vaslui=(509, 444), Zerind=(108, 531))
```

It is pretty straightforward to understand this `romania_map`. The first node **Arad** has three neighbours named **Zerind**, **Sibiu**, **Timisoara**. Each of these nodes are 75, 140, 118 units apart from **Arad** respectively. And the same goes with other nodes.

And `romania_map.locations` contains the positions of each of the nodes. We will use the straight line distance (which is different from the one provided in `romania_map`) between two cities in algorithms like A*-search and Recursive Best First Search.

Define a problem: Now it's time to define our problem. We will define it by passing `initial`, `goal`, `graph` to `GraphProblem`. So, our problem is to find the goal state starting from the given initial state on the provided graph.

Say we want to start exploring from **Arad** and try to find **Bucharest** in our romania_map. So, this is how we do it.

```
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
```

Romania Map Visualisation

Let's have a visualisation of Romania map [Figure 3.2] from the book and see how different searching algorithms perform / how frontier expands in each search algorithm for a simple problem named `romania_problem`.

Have a look at `romania_locations`. It is a dictionary defined in search module. We will use these location values to draw the romania graph using **networkx**.

```
romania_locations = romania_map.locations
print(romania_locations)
```

```
{'Arad': (91, 492), 'Bucharest': (400, 327), 'Craiova': (253, 288), 'Drobeta': (
```

Let's get started by initializing an empty graph. We will add nodes, place the nodes in their location as shown in the book, add edges to the graph.

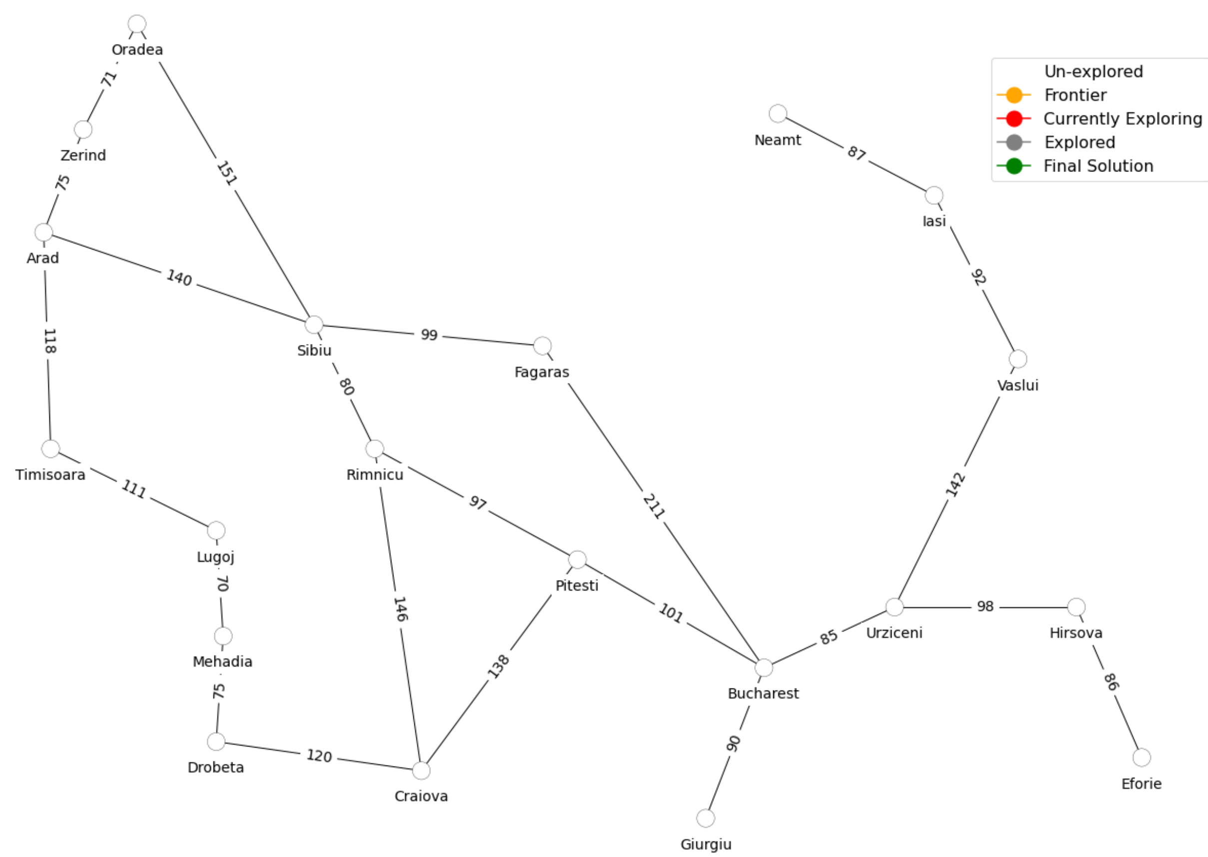
```
# node colors, node positions and node label positions
node_colors = {node: 'white' for node in romania_map.locations.keys()}
```

```
node_positions = romania_map.locations
node_label_pos = { k:[v[0],v[1]-10]  for k,v in romania_map.locations.items() }
edge_weights = {(k, k2) : v2 for k, v in romania_map.graph_dict.items() for k2, v2 in

romania_graph_data = {  'graph_dict' : romania_map.graph_dict,
                        'node_colors': node_colors,
                        'node_positions': node_positions,
                        'node_label_positions': node_label_pos,
                         'edge_weights': edge_weights
                     }
```

We have completed building our graph based on romania_map and its locations. It's time to display it here in the notebook. This function `show_map(node_colors)` helps us do that. We will be calling this function later on to display the map at each and every interval step while searching, using variety of algorithms from the book.

We can simply call the function with node_colors dictionary object to display it.

```
show_map(romania_graph_data)
```



Voila! You see, the romania map as shown in the Figure[3.2] in the book. Now, see how different searching algorithms perform with our problem statements.

SIMPLE PROBLEM SOLVING AGENT PROGRAM

Let us now define a Simple Problem Solving Agent Program. Run the next cell to see how the abstract class `SimpleProblemSolvingAgentProgram` is defined in the search module.

```
psource(SimpleProblemSolvingAgentProgram)
```

```
class SimpleProblemSolvingAgentProgram:
    """
    [Figure 3.1]
    Abstract framework for a problem-solving agent.
    """

    def __init__(self, initial_state=None):
        """State is an abstract representation of the state
        of the world, and seq is the list of actions required
        to get to a particular state from the initial state(root)."""
        self.state = initial_state
        self.seq = []

    def __call__(self, percept):
        """[Figure 3.1] Formulate a goal and problem, then
        search for a sequence of actions to solve it."""
        self.state = self.update_state(self.state, percept)
        if not self.seq:
            goal = self.formulate_goal(self.state)
            problem = self.formulate_problem(self.state, goal)
            self.seq = self.search(problem)
            if not self.seq:
                return None
        return self.seq.pop(0)

    def update_state(self, state, percept):
        raise NotImplementedError

    def formulate_goal(self, state):
        raise NotImplementedError

    def formulate_problem(self, state, goal):
        raise NotImplementedError

    def search(self, problem):
        raise NotImplementedError
```

The SimpleProblemSolvingAgentProgram class has six methods:

- `__init__(self, intial_state=None)`: This is the `contructor` of the class and is the first method to be called when the class is instantiated. It takes in a keyword argument, `initial_state` which is initially `None`. The argument `initial_state` represents the state from which the agent starts.

- `__call__(self, percept)`: This method updates the `state` of the agent based on its `percept` using the `update_state` method. It then formulates a `goal` with the help of `formulate_goal` method and a `problem` using the `formulate_problem` method and returns a sequence of actions to solve it (using the `search` method).
- `update_state(self, percept)`: This method updates the `state` of the agent based on its `percept`.
- `formulate_goal(self, state)`: Given a `state` of the agent, this method formulates the `goal` for it.
- `formulate_problem(self, state, goal)`: It is used in problem formulation given a `state` and a `goal` for the `agent`.
- `search(self, problem)`: This method is used to search a sequence of `actions` to solve a `problem`.

Let us now define a Simple Problem Solving Agent Program. We will create a simple `vacuumAgent` class which will inherit from the abstract class `SimpleProblemSolvingAgentProgram` and overrides its methods. We will create a simple intelligent vacuum agent which can be in any one of the following states. It will move to any other state depending upon the current state as shown in the picture by arrows:

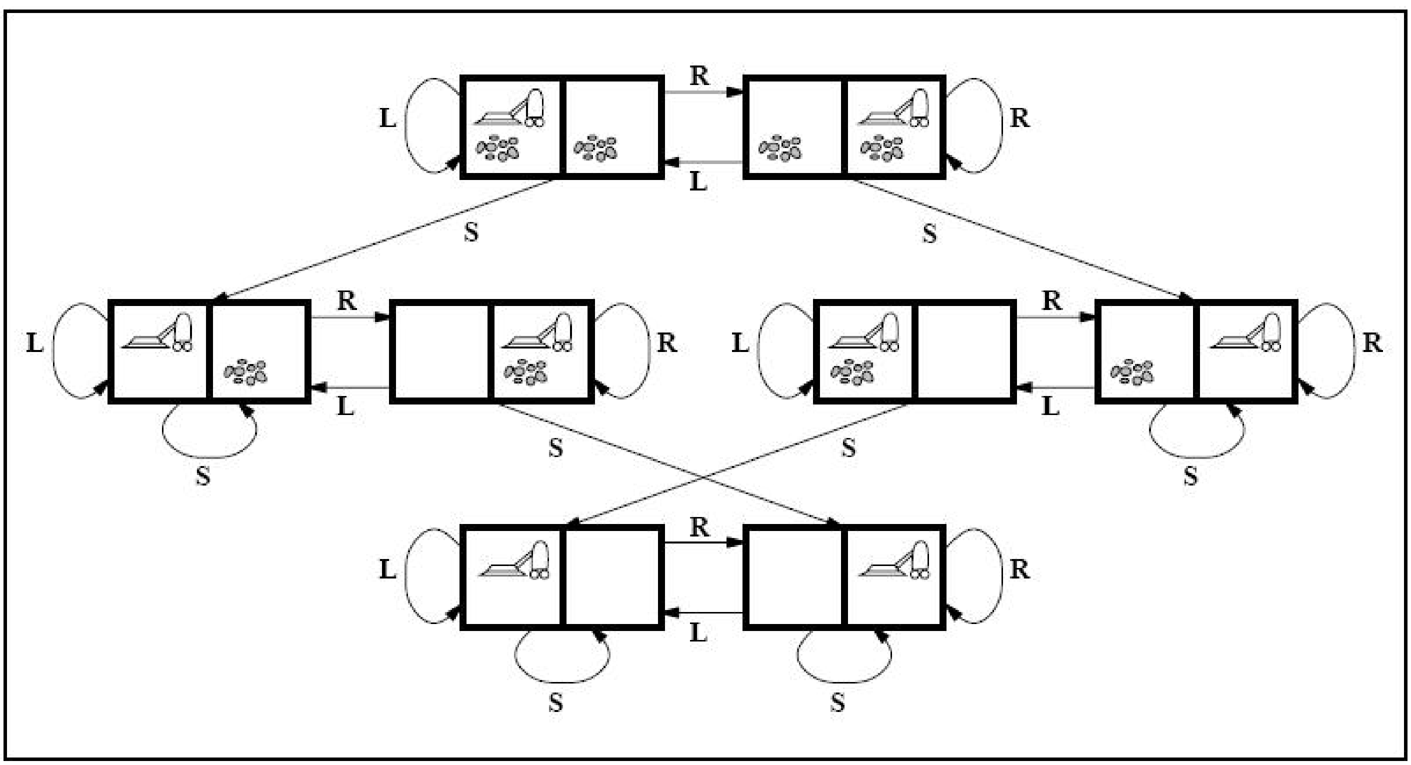


Fig : Simple problem solving agent

```
class vacuumAgent(SimpleProblemSolvingAgentProgram):
        def update_state(self, state, percept):
            return percept

        def formulate_goal(self, state):
            goal = [state7, state8]
            return goal

        def formulate_problem(self, state, goal):
            problem = state
            return problem

        def 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"]
            return seq
```

Now, we will define all the 8 states and create an object of the above class. Then, we will pass it different states and check the output:

```
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"]]]

a = vacuumAgent(state1)

print(a(state6))
print(a(state1))
print(a(state3))
```

```
Left
Suck
Right
```

Task 1 [5%]

1) Print the output of the robot at every state, considering that is its current state and explain the output logic.

2) From each current state, describe where would it move next.

```
print(a(state1))
print(a(state2))
print(a(state3))
print(a(state4))
print(a(state5))
print(a(state6))
print(a(state7))
```

```
Suck
Suck
Left
Suck
Suck
Left
Suck
```

Double-click (or enter) to edit

The robot, i.e. the vacuum cleaner, will move from stage 1 to state 8 in the aforementioned sequence. Because we defined "Dirty" at that place, we declared the variable "a" as state1 where the Robot's function is to Suck. The robot then proceeds to the right, to state 2, where it Sucks the Dirt and then returns to state 3. It then moves right to state 4, where it Sucks, and then moves left to state 5, where it Sucks, and then goes to state 6, where it Sucks the Dirt and then returns to state 8

SEARCHING ALGORITHMS VISUALIZATION

In this section, we have visualizations of the following searching algorithms:

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

Useful reference to to know more about uninformed search:

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://medium.com/nothingaholic/depth-first-search-vs-breadth-first-search-in-python-81521caa8f44

https://algodaily.com/lessons/dfs-vs-bfs

https://towardsdatascience.com/search-algorithm-dijkstras-algorithm-uniform-cost-search-with-python-ccbee250ba9

https://ai-master.gitbooks.io/classic-search/content/what-is-depth-limited-search.html

https://www.educative.io/edpresso/what-is-iterative-deepening-search

We add the colors to the nodes to have a nice visualisation when displaying. So, these are the different colors we are using in these visuals:

- Un-explored nodes - white
- Frontier nodes - orange
- Currently exploring node - red
- Already explored nodes - gray

1. BREADTH-FIRST TREE SEARCH

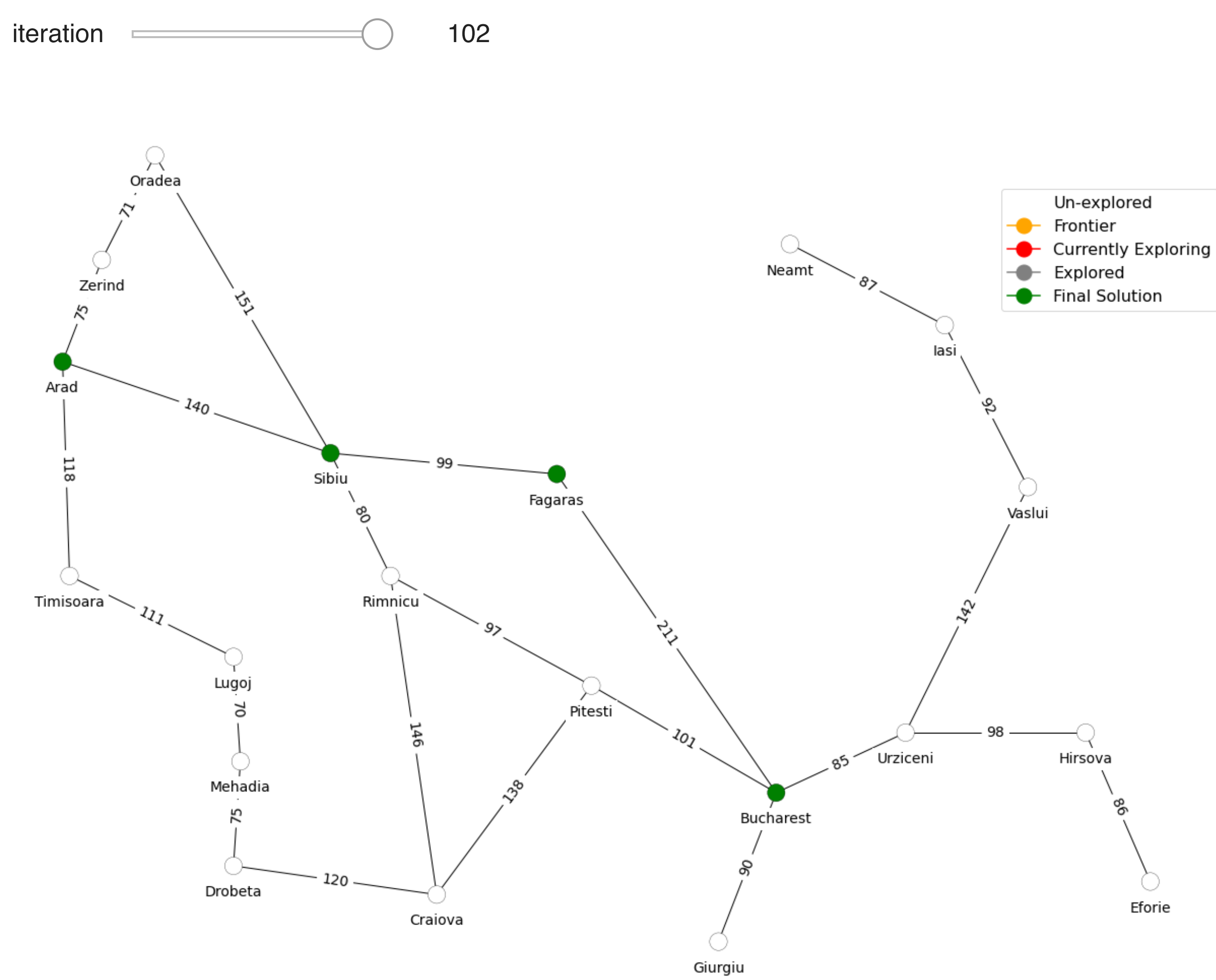
We have a working implementation in search module. But as we want to interact with the graph while it is searching, we need to modify the implementation. Here's the modified breadth first tree search.

```
def tree_breadth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the queue
    frontier = deque([Node(problem.initial)])

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of queue
        node = frontier.popleft()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
```

```
    return None

def breadth_first_tree_search(problem):
    "Search the shallowest nodes in the search tree first."
    iterations, all_node_colors, node = tree_breadth_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

Now, we use `ipywidgets` to display a slider, a button and our romania map. By sliding the slider we can have a look at all the intermediate steps of a particular search algorithm. By pressing the button **Visualize**, you can see all the steps without interacting with the slider. These two helper functions are the callback functions which are called when we interact with the slider and the button.

```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
a, b, c = breadth_first_tree_search(romania_problem)
display_visual(romania_graph_data, user_input=False,
               algorithm=breadth_first_tree_search,
               problem=romania_problem)
```



2. DEPTH-FIRST TREE SEARCH

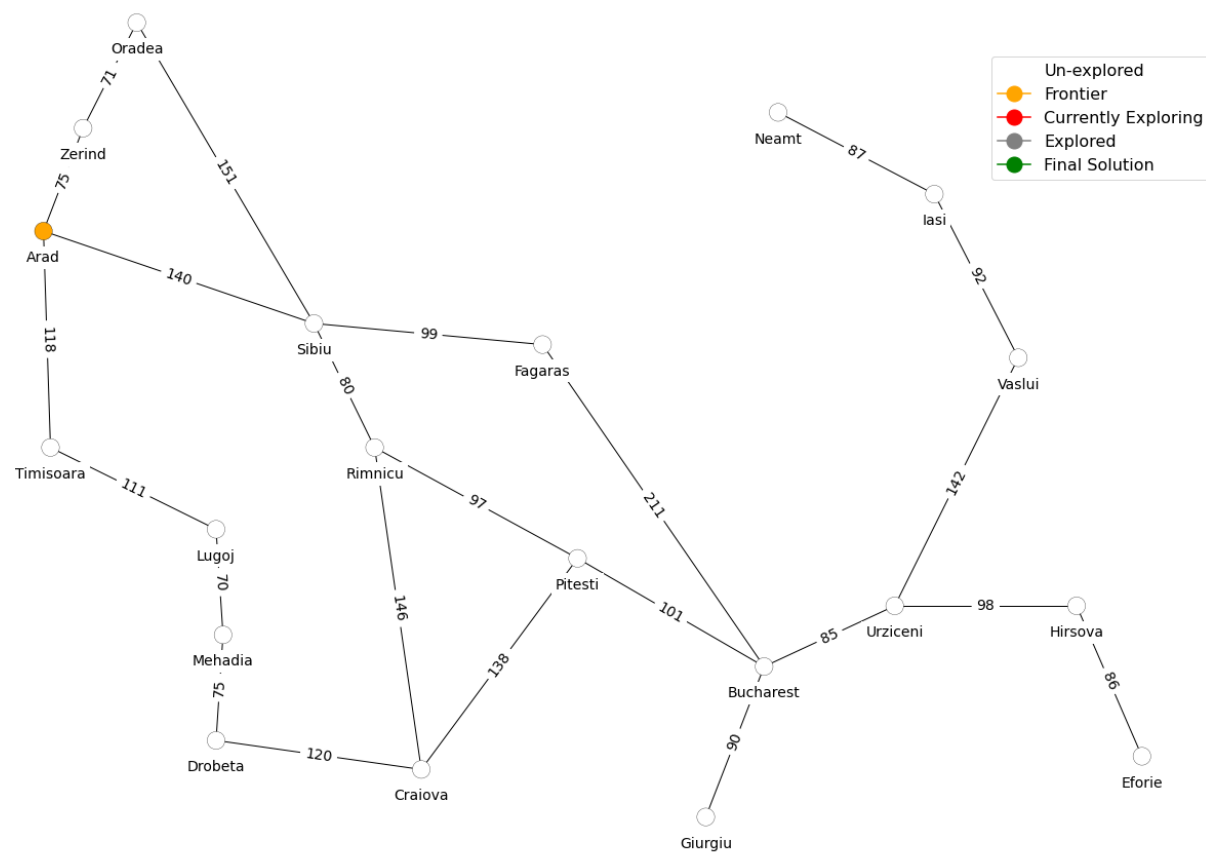
Now let's discuss another searching algorithm, Depth-First Tree Search.

```
def tree_depth_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    Don't worry about repeated paths to a state. [Figure 3.7]"""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    #Adding first node to the stack
    frontier = [Node(problem.initial)]

    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        #Popping first node of stack
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        frontier.extend(node.expand(problem))

        for n in node.expand(problem):
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))
```

```
    return None

def depth_first_tree_search(problem):
    "Search the deepest nodes in the search tree first."
    iterations, all_node_colors, node = tree_depth_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_first_tree_search,
               problem=romania_problem)
```

iteration 0



3. BREADTH-FIRST GRAPH SEARCH

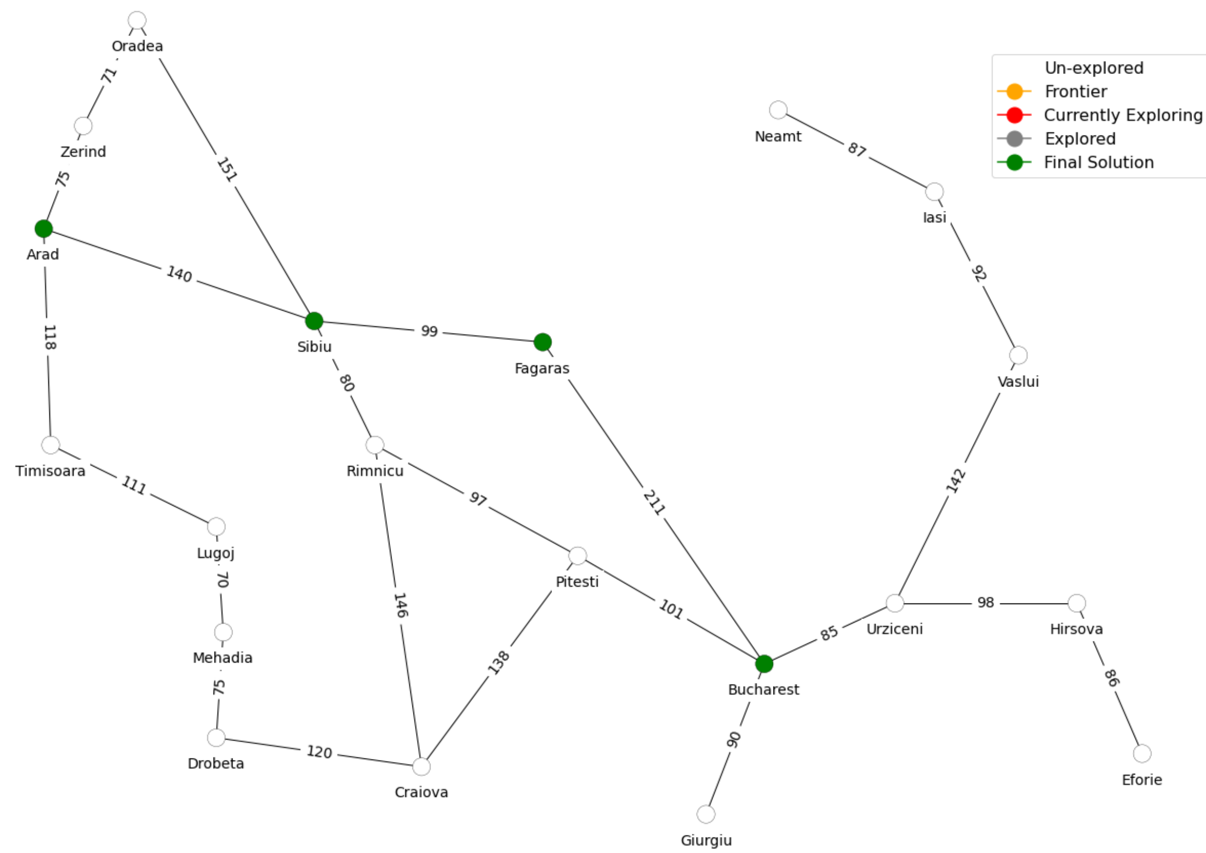
Let's change all the `node_colors` to starting position and define a different problem statement.

```
def breadth_first_search_graph(problem):
    "[Figure 3.11]"
```

```
# we use these two variables at the time of visualisations
iterations = 0
all_node_colors = []
node_colors = {k : 'white' for k in problem.graph.nodes()}

node = Node(problem.initial)

node_colors[node.state] = "red"
iterations += 1
all_node_colors.append(dict(node_colors))

if problem.goal_test(node.state):
    node_colors[node.state] = "green"
    iterations += 1
    all_node_colors.append(dict(node_colors))
    return(iterations, all_node_colors, node)

frontier = deque([node])

# modify the color of frontier nodes to blue
node_colors[node.state] = "orange"
iterations += 1
all_node_colors.append(dict(node_colors))

explored = set()
while frontier:
    node = frontier.popleft()
    node_colors[node.state] = "red"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    explored.add(node.state)

    for child in node.expand(problem):
        if child.state not in explored and child not in frontier:
            if problem.goal_test(child.state):
                node_colors[child.state] = "green"
                iterations += 1
                all_node_colors.append(dict(node_colors))
                return(iterations, all_node_colors, child)
            frontier.append(child)

            node_colors[child.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

    node_colors[node.state] = "gray"
    iterations += 1
    all_node_colors.append(dict(node_colors))
return None
```

```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=breadth_first_search_graph,
               problem=romania_problem)
```

iteration 21



4. DEPTH-FIRST GRAPH SEARCH

Although we have a working implementation in search module, we have to make a few changes in the algorithm to make it suitable for visualization.

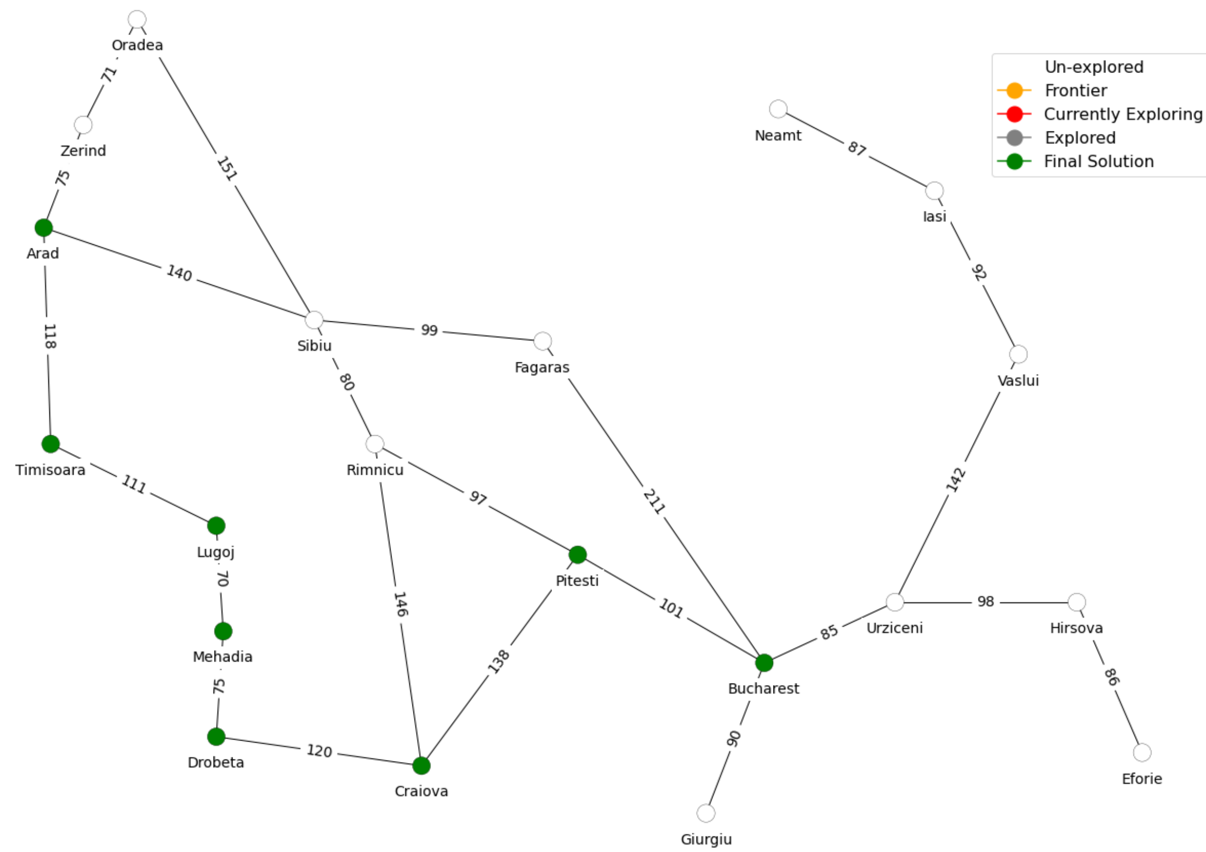
```
def graph_search_for_vis(problem):
    """Search through the successors of a problem to find a goal.
    The argument frontier should be an empty queue.
    If two paths reach a state, only use the first one. [Figure 3.7]"""
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
```

```
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [(Node(problem.initial))]
    explored = set()

    # modify the color of frontier nodes to orange
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of stack
        node = frontier.pop()

        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return None


def depth_first_graph_search(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = graph_search_for_vis(problem)
    return(iterations, all_node_colors, node)
```

```
all_node_colors = []
```

```
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_first_graph_search,
               problem=romania_problem)
```

iteration 40



5. UNIFORM COST SEARCH

Let's change all the `node_colors` to starting position and define a different problem statement.

```
def best_first_graph_search_for_vis(problem, f):
    """Search the nodes with the lowest f scores first.
    You specify the function f(node) that you want to minimize; for example,
    if f is a heuristic estimate to the goal, then we have greedy best
    first search; if f is node.depth then we have breadth-first search.
    There is a subtlety: the line "f = memoize(f, 'f')" means that the f
    values will be cached on the nodes as they are computed. So after doing
    a best first search you can examine the f values of the path returned."""

    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
```

```
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    f = memoize(f, 'f')
    node = Node(problem.initial)

    node_colors[node.state] = "red"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    if problem.goal_test(node.state):
        node_colors[node.state] = "green"
        iterations += 1
        all_node_colors.append(dict(node_colors))
        return(iterations, all_node_colors, node)

    frontier = PriorityQueue('min', f)
    frontier.append(node)

    node_colors[node.state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    explored = set()
    while frontier:
        node = frontier.pop()

        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                frontier.append(child)
                node_colors[child.state] = "orange"
                iterations += 1
                all_node_colors.append(dict(node_colors))
            elif child in frontier:
                incumbent = frontier[child]
                if f(child) < incumbent:
                    del frontier[child]
                    frontier.append(child)
                    node_colors[child.state] = "orange"
                    iterations += 1
                    all_node_colors.append(dict(node_colors))
```

```
    node_colors[node.state] = "gray"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    return None
```

```
def uniform_cost_search_graph(problem):
    "[Figure 3.14]"
    #Uniform Cost Search uses Best First Search algorithm with f(n) = g(n)
    iterations, all_node_colors, node = best_first_graph_search_for_vis(problem, lamb
    return(iterations, all_node_colors, node)
```

```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=uniform_cost_search_graph,
               problem=romania_problem)
```

6. DEPTH LIMITED SEARCH

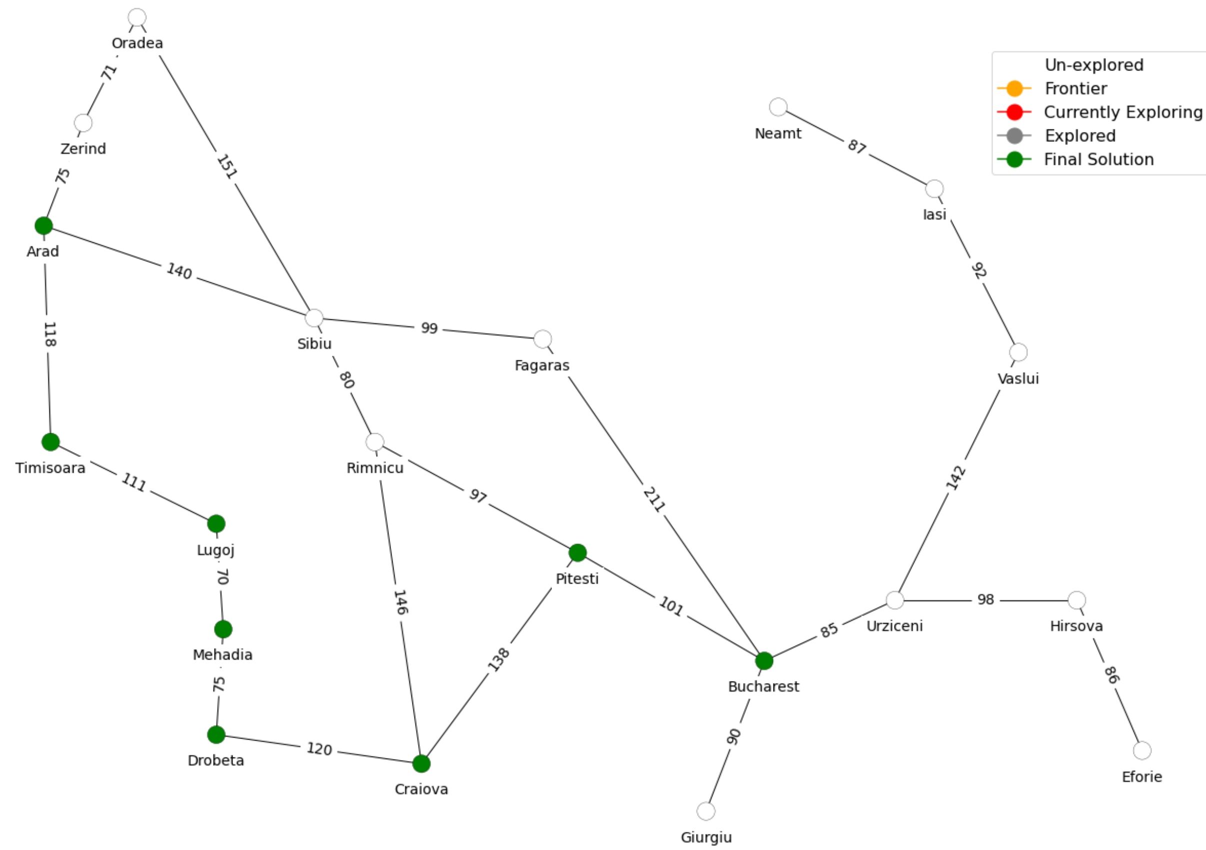
Let's change all the 'node_colors' to starting position and define a different problem statement. Although we have a working implementation, but we need to make changes.

```
def depth_limited_search_graph(problem, limit = -1):
    '''
    Perform depth first search of graph g.
    if limit >= 0, that is the maximum depth of the search.
    '''
    # we use these two variables at the time of visualisations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    frontier = [Node(problem.initial)]
    explored = set()

    cutoff_occurred = False
    node_colors[Node(problem.initial).state] = "orange"
    iterations += 1
    all_node_colors.append(dict(node_colors))

    while frontier:
        # Popping first node of queue
        node = frontier.pop()
```

```python
        # modify the currently searching node to red
        node_colors[node.state] = "red"
        iterations += 1
        all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            # modify goal node to green after reaching the goal
            node_colors[node.state] = "green"
            iterations += 1
            all_node_colors.append(dict(node_colors))
            return(iterations, all_node_colors, node)

        elif limit >= 0:
            cutoff_occurred = True
            limit += 1
            all_node_colors.pop()
            iterations -= 1
            node_colors[node.state] = "gray"


        explored.add(node.state)
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)

        for n in frontier:
            limit -= 1
            # modify the color of frontier nodes to orange
            node_colors[n.state] = "orange"
            iterations += 1
            all_node_colors.append(dict(node_colors))

        # modify the color of explored nodes to gray
        node_colors[node.state] = "gray"
        iterations += 1
        all_node_colors.append(dict(node_colors))

    return 'cutoff' if cutoff_occurred else None


def depth_limited_search_for_vis(problem):
    """Search the deepest nodes in the search tree first."""
    iterations, all_node_colors, node = depth_limited_search_graph(problem)
    return(iterations, all_node_colors, node)
```

```python
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=depth_limited_search_for_vis,
               problem=romania_problem)
```

iteration 40



7. ITERATIVE DEEPENING SEARCH

Let's change all the 'node_colors' to starting position and define a different problem statement.

```
def iterative_deepening_search_for_vis(problem):
    for depth in range(sys.maxsize):
        iterations, all_node_colors, node=depth_limited_search_for_vis(problem)
        if iterations:
            return (iterations, all_node_colors, node)
```

```
all_node_colors = []
romania_problem = GraphProblem('Arad', 'Bucharest', romania_map)
display_visual(romania_graph_data, user_input=False,
               algorithm=iterative_deepening_search_for_vis,
               problem=romania_problem)
```

iteration 0

TASK 2 [10%]

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.

TASK 3 [15%]

For each search method, explan the graph in the visualization part and the complete route taken to each the goal node. Compare the route taken in this notebook's visualization with the search methods's logic and see if it matches or not.

As per the above visualizations,the code executes as per the logic and gives an output with lowest cost .

TASK 4 [35%]

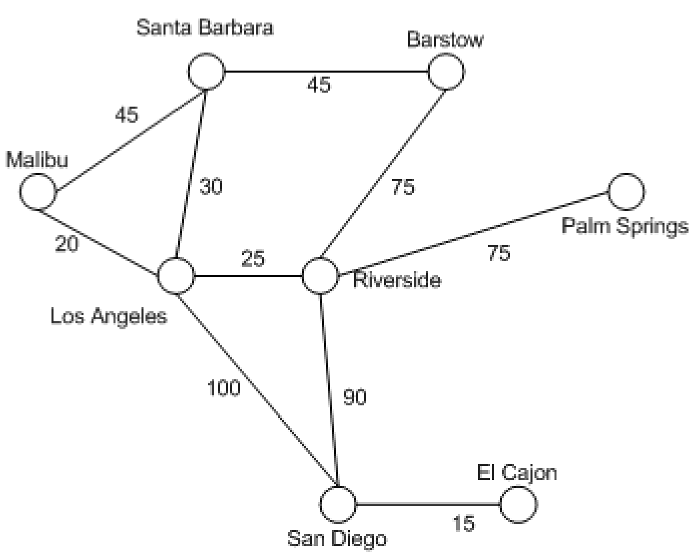


Fig 1: santa_barbara_map

- Now create an Undirected Graph such as the romania_map, containing a dict of nodes as keys and neighbours as values.
- Start exploring from Santa Barbara and try to find El Cajon in the map using.
- Start exploring from Barstow and try to find El Cajon in the map.

- Now show the visualisation of the map [Figure 1] from the task and see how different searching algorithms perform / how frontier expands in each of the following search algorits:

> 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

```
santa_barbara_map = UndirectedGraph(dict(
    SantaBarbara=dict(Barstow=45, LosAngeles=30, Malibu=45),
    ElCajon=dict(SanDiego=15),
    Barstow=dict(SantaBarbara=45, Riverside=75),
    Riverside=dict(Barstow=75, PalmSprings=75, SanDiego=90, LosAngeles= 25),
    PalmSprings=dict(Riverside=75),
    SanDiego=dict(ElCajon=15, Riverside=90, LosAngeles=100 ),
    LosAngeles=dict(SanDiego=100, Riverside=25, SantaBarbara=30, Malibu=20),
    Malibu=dict(LosAngeles=20, SantaBarbara=45)))

santa_barbara_map.locations = dict(
    SantaBarbara=(90, 30), ElCajon=(250, 240), Barstow=(220, 30),
    Riverside=(150, 125), PalmSprings=(310, 80), SanDiego=(160, 250),
    LosAngeles=(90, 130), Malibu=(10, 90))
```

```
santa_barbara_problem = GraphProblem('SantaBarbara ', 'ElCajon', santa_barbara_map)
```

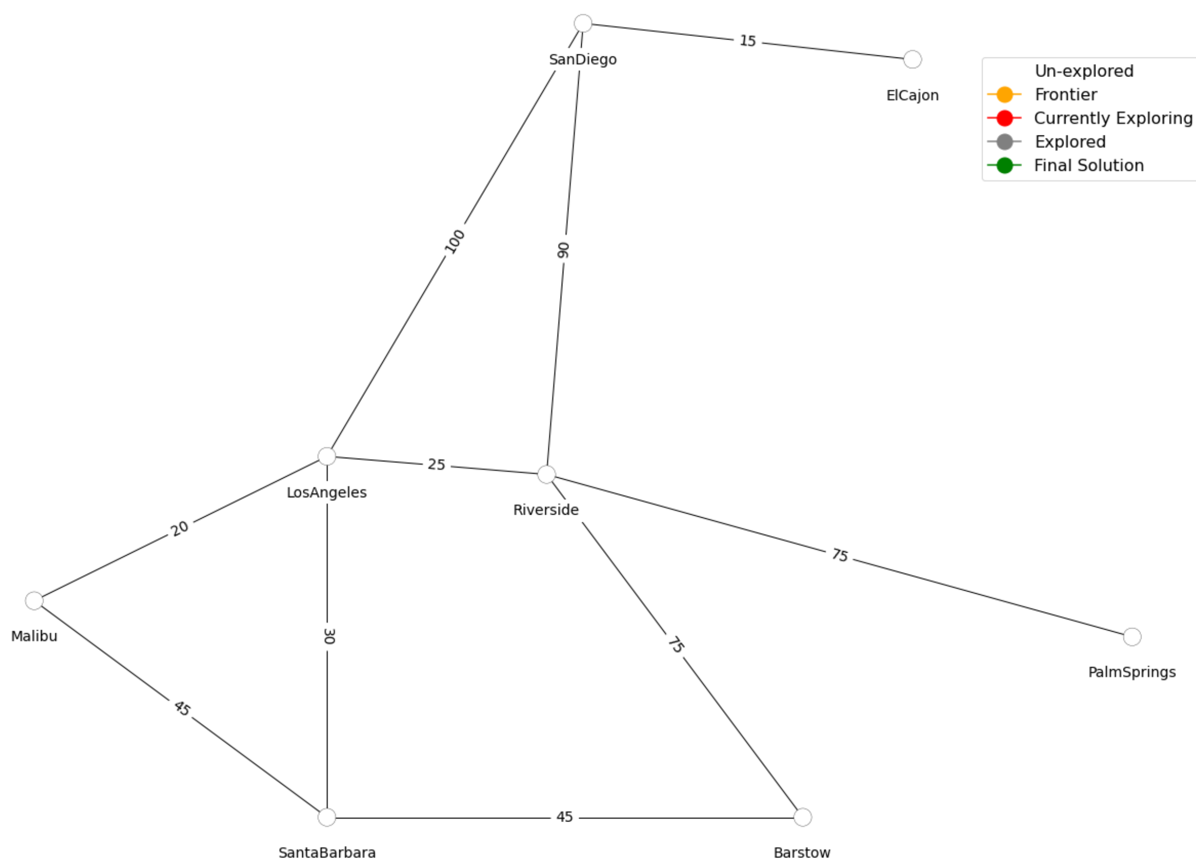
```
santa_barbara_locations = santa_barbara_map.locations
print(santa_barbara_locations)
```

```
{'SantaBarbara': (90, 30), 'ElCajon': (250, 240), 'Barstow': (220, 30), 'Riversi
```

```
# node colors, node positions and node label positions
node_colors = {node: 'white' for node in santa_barbara_map.locations.keys()}
node_positions = santa_barbara_map.locations
node_label_pos = { k:[v[0],v[1]-10]  for k,v in santa_barbara_map.locations.items() }
edge_weights = {(k, k2) : v2 for k, v in santa_barbara_map.graph_dict.items() for k2,

santa_barbara_graph_data = {  'graph_dict' : santa_barbara_map.graph_dict,
                        'node_colors': node_colors,
```

```
                                'node_positions': node_positions,
                                'node_label_positions': node_label_pos,
                                 'edge_weights': edge_weights
                            }
```

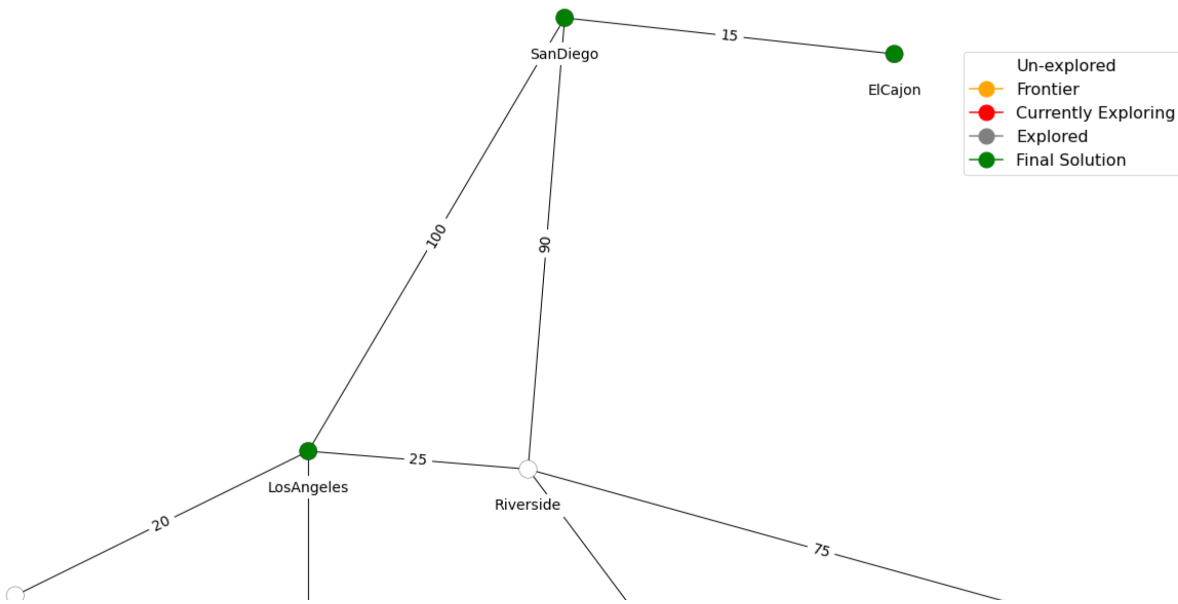
```
show_map(santa_barbara_graph_data)
```



Breadth First Tree Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
a, b, c = breadth_first_tree_search(santa_barbara_problem)
display_visual(santa_barbara_graph_data, user_input=False,
               algorithm=breadth_first_tree_search,
               problem=santa_barbara_problem)
```

iteration 96



Depth First Tree Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
               algorithm=depth_first_tree_search,
               problem=santa_barbara_problem)
```

iteration 0

<Figure size 1296x936 with 0 Axes>

visualize

Breadth First Graph Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
               algorithm=breadth_first_search_graph,
               problem=santa_barbara_problem)
```

Depth First Graph Search:

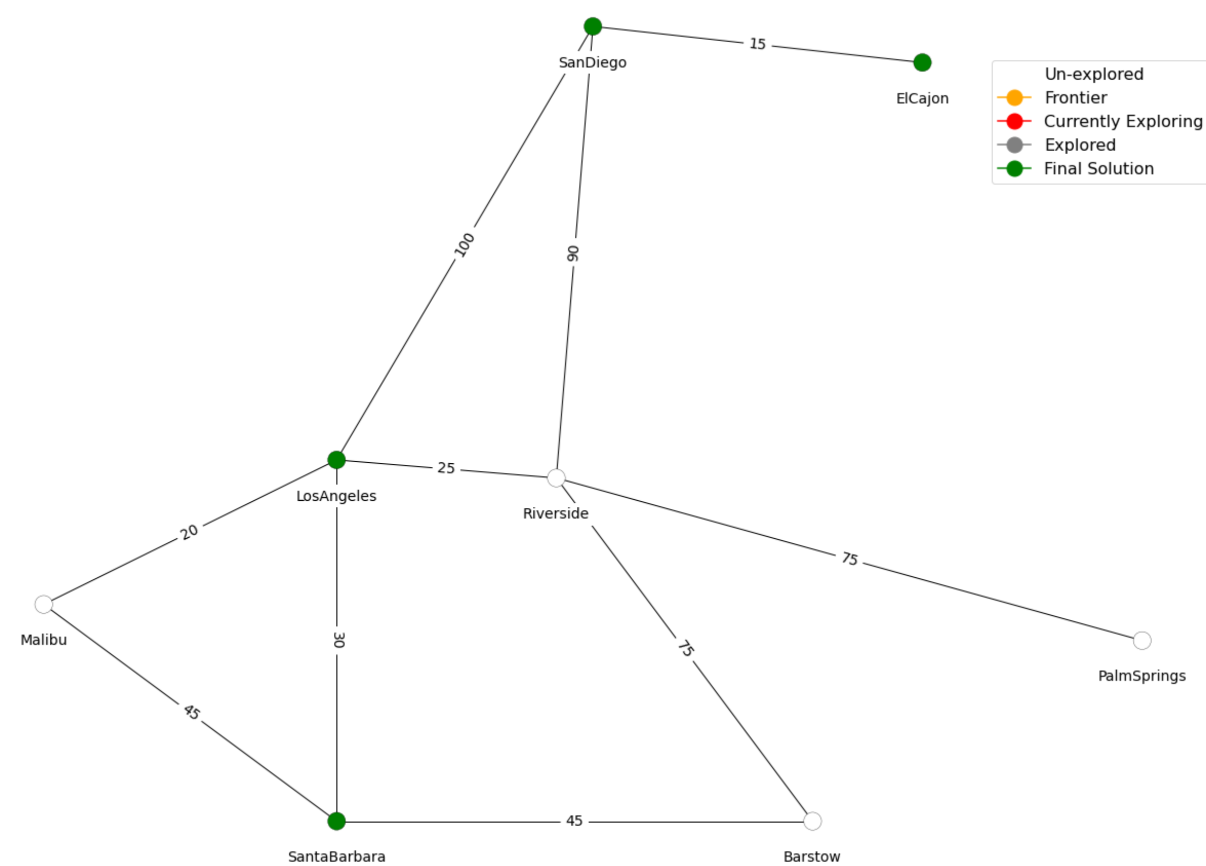
```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
```

```
                algorithm=depth_first_graph_search,
                problem=santa_barbara_problem)
```

Uniform Cost Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
                algorithm=uniform_cost_search_graph,
                problem=santa_barbara_problem)
```

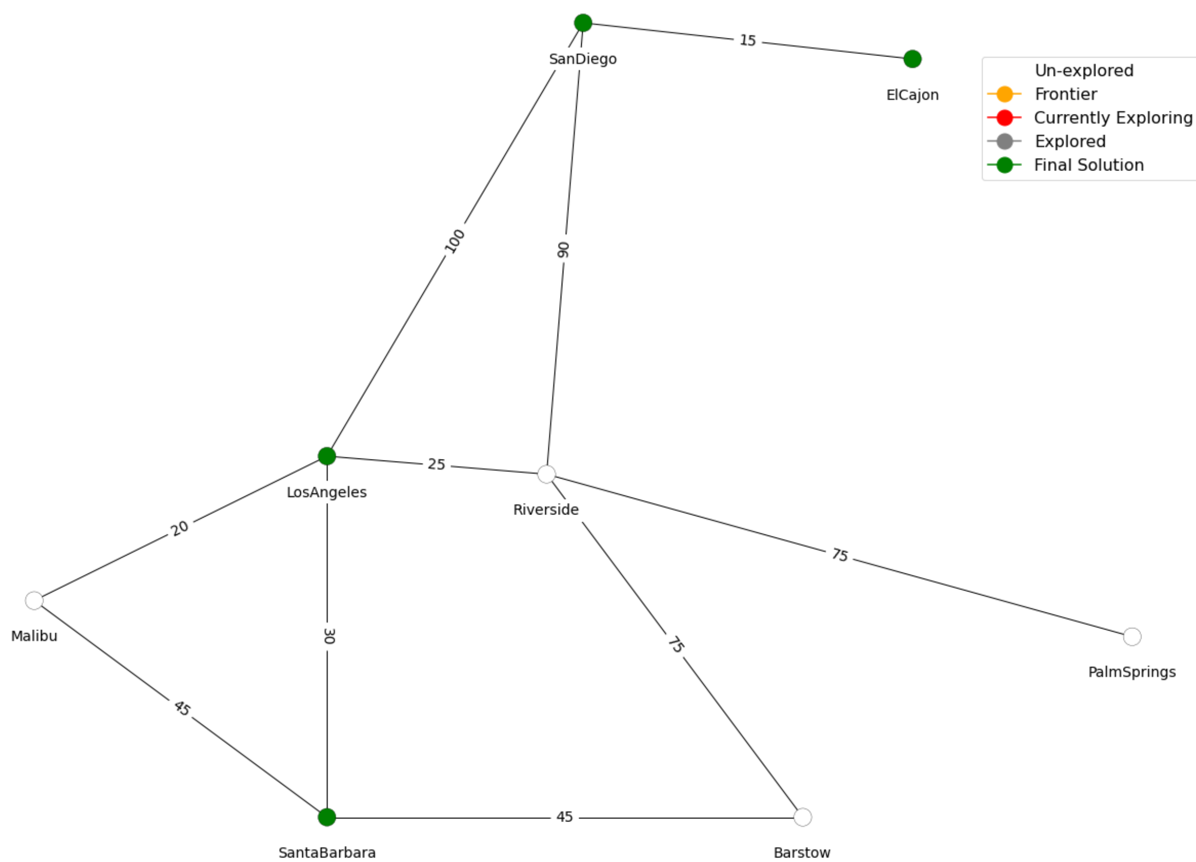
iteration 18



Depth Limited Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
                algorithm=depth_limited_search_for_vis,
                problem=santa_barbara_problem)
```

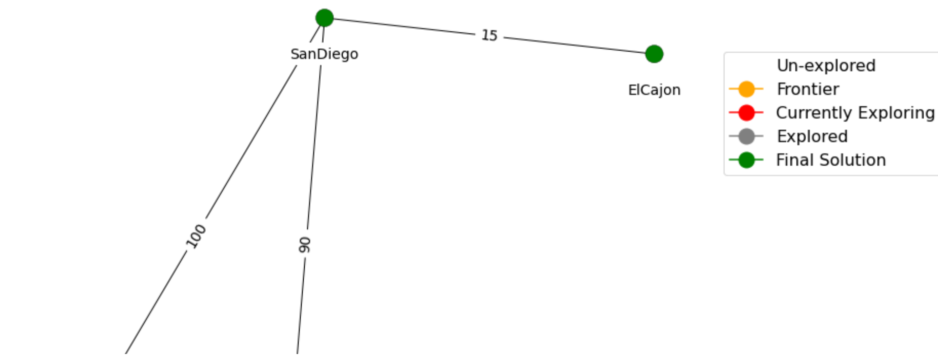
iteration 30



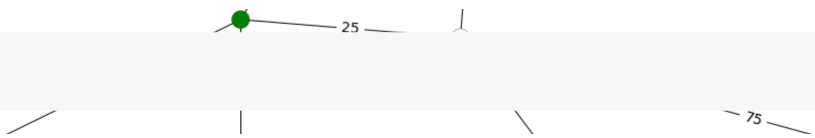
Iterative Deepening Search:

```
all_node_colors = []
santa_barbara_problem = GraphProblem('SantaBarbara', 'ElCajon', santa_barbara_map)
display_visual(santa_barbara_graph_data, user_input=False,
               algorithm=iterative_deepening_search_for_vis,
               problem=santa_barbara_problem)
```

iteration 30



Task 5 [35%]



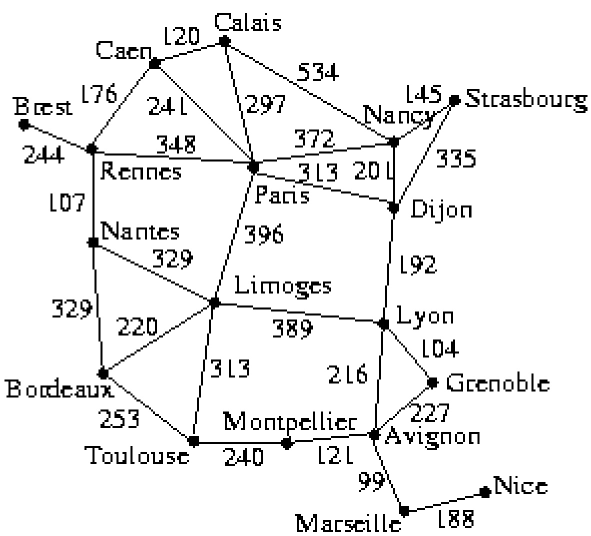


Fig 2: brest_map

- Now create an Undirected Graph such as the romania_map, containing a dict of nodes as keys and neighbours as values.
- Start exploring from Bordeaux and try to find Stasbourg in the map using.
- Start exploring from Brest and try to find Nice in the map.
- Now show the visualisation of the map [Figure 1] from the task and see how different searching algorithms perform / how frontier expands in each of the following search algoriths:

> 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

```
brest_map = UndirectedGraph(dict(
    Brest=dict(Rennes=244),
    Rennes=dict(Caen=176, Paris=348, Nantes=107, Brest=244),
    Nantes=dict(Rennes=107, Limoges=329, Bordeaux=329),
    Bordeaux=dict(Nantes=329, Limoges= 220, Toulouse= 253),
    Toulouse=dict(Bordeaux=253, Limoges= 313, Montpellier= 240),
    Montpellier=dict(Toulouse=240, Avignon=121),
    Avignon=dict(Montpellier=121, Lyon=216, Grenoble=227, Marseille=99),
    Grenoble=dict(Avignon=227, Lyon=104),
    Lyon=dict(Grenoble=104, Avignon=216, Limoges= 389, Dijon=192),
    Dijon=dict(Lyon=192, Paris=313, Nancy=201, Strasbourg= 335),
    Strasbourg=dict(Dijon=335, Nancy=145 ),
    Nancy=dict(Strasbourg=145, Dijon=201, Paris=372, Calais= 534),
    Calais=dict(Nancy=534, Paris=297, Caen=120),
    Caen= dict(Calais=120, Paris=241, Rennes=176),
    Paris= dict(Caen=241, Calais= 297, Dijon= 313, Nancy=372, Rennes=348, Limoges=396
    Limoges= dict(Paris=396, Nantes=329, Bordeaux=220, Toulouse=313, Lyon=389),
    Marseille= dict(Avignon=99, Nice=188),
    Nice= dict(Marseille=188)))

brest_map.locations = dict(
    Brest=(25, 77), Rennes=(58, 88), Nantes=(59, 133),
    Bordeaux=(64, 199), Toulouse=(108, 232), Montpellier=(156, 234),
    Avignon=(198, 230), Grenoble=(227, 205), Lyon=(203, 176),
    Dijon=(209, 117), Strasbourg=(238, 65), Nancy=(210, 84),
    Calais=(124, 35), Caen=(89, 44), Paris=(137, 97),
    Limoges=(119, 164), Marseille=(213, 267), Nice=(255, 259))
```

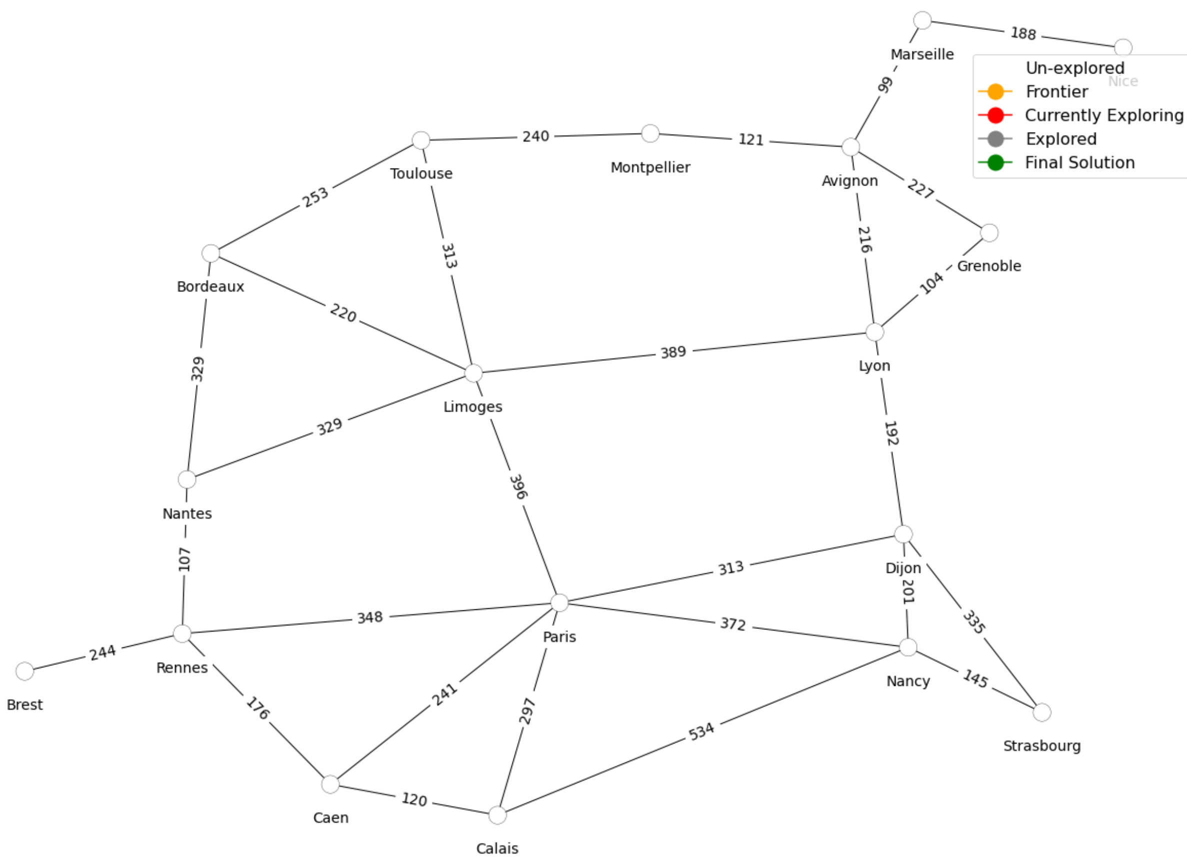
```
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
```

```
brest_locations = brest_map.locations
print(brest_locations)
```

```
{'Brest': (25, 77), 'Rennes': (58, 88), 'Nantes': (59, 133), 'Bordeaux': (64, 19
```

```
# node colors, node positions and node label positions
node_colors = {node: 'white' for node in brest_map.locations.keys()}
node_positions = brest_locations
node_label_pos = { k:[v[0],v[1]-10]  for k,v in brest_locations.items() }
edge_weights = {(k, k2) : v2 for k, v in brest_map.graph_dict.items() for k2, v2 in v

brest_graph_data = {  'graph_dict' : brest_map.graph_dict,
                        'node_colors': node_colors,
                        'node_positions': node_positions,
                        'node_label_positions': node_label_pos,
                         'edge_weights': edge_weights
                }
```

```
show_map(brest_graph_data)
```




```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
a, b, c = breadth_first_tree_search(brest_problem)
display_visual(brest_graph_data, user_input=False,
               algorithm=breadth_first_tree_search,
               problem=brest_problem)
```

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
```

```
                 algorithm=depth_first_tree_search,
                 problem=brest_problem)
```

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
                 algorithm=breadth_first_search_graph,
                 problem=brest_problem)
```

iteration 0

visualize

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
                 algorithm=depth_first_graph_search,
                 problem=brest_problem)
```

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
                 algorithm=uniform_cost_search_graph,
                 problem=brest_problem)
```

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
                 algorithm=depth_limited_search_for_vis,
                 problem=brest_problem)
```

```
all_node_colors = []
brest_problem = GraphProblem('Bordeaux', 'Stasbourg', brest_map)
display_visual(brest_graph_data, user_input=False,
                 algorithm=iterative_deepening_search_for_vis,
                 problem=brest_problem)
```

iteration 0

visualize

0s completed at 10:32 PM