Solving problems by Searching

This notebook serves as supporting material for topics covered in **Chapter 3 - Solving Problems by Searching** and **Chapter 4 - Beyond Classical Search** 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.

In [ ]:
```
# Add --break-system-packages at the end of the pip install
# for local installs on OS's like Ubuntu
!git clone https://github.com/ucsd-cse150b-f25/notebooks  > /dev/null 2>&1
!mv notebooks/* ./
# After first run, comment the line and rerun
```

In [ ]:
```
# Run once and restart your session.
# After restart comment the line and rerun
!pip install -r requirements.txt > /dev/null 2>&1
```

In [ ]:
```
# After pip install, run this once, restart your session
# After restart comment the line and rerun
!playwright install chromium > /dev/null 2>&1
```

In [ ]:
```
from search import *
from notebook import psource, heatmap, gaussian_kernel, show_map, final_path
from IPython.display import display, Image
import imgkit
# Needed to hide warnings in the matplotlib sections
import warnings
warnings.filterwarnings("ignore")
```

CONTENTS

- Simulated Annealing
- Genetic Algorithm
- AND-OR Graph Search
- Online DFS Agent
- LRTA* Agent

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
- **Informed search algorithms**: These type of algorithms leverage any information (heuristics, path cost) on the problem to search through the search space to find the solution efficiently.
    - Examples:
        1. Best First Search
        2. Uniform Cost Search
        3. A* Search
        4. Recursive Best First 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`.

In [ ]:
```
%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.

In [ ]:
```
psource(Problem)
```

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.

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.

In [ ]:
```
psource(Node)
```

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.

In [ ]:
```
psource(GraphProblem)
```

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

In [ ]:
```
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.

In [ ]:
```
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**.

In [ ]:
```
romania_locations = romania_map.locations
print(romania_locations)
```

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.

In [ ]:
```
# 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

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.

In [ ]:
```
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.

In [ ]:
```
psource(SimpleProblemSolvingAgentProgram)
```

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:

 simple problem solving agent

In [ ]:
```
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:

In [ ]:
```
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))
```

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. Best First Graph Search
6. Uniform Cost Search
7. Depth Limited Search
8. Iterative Deepening Search
9. Greedy Best First Search
10. A*-Search
11. Recursive Best First 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.

In [ ]:

```
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.

In [ ]:
```
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.

In [ ]:
```
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)
```

In [ ]:
```
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)
```

3. BREADTH-FIRST GRAPH SEARCH

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

In [ ]:
```
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
```

In [ ]:
```
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)
```

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.

In [ ]:
```
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)
```

In [ ]:

```
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)
```

5. BEST FIRST SEARCH

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

In [ ]:
```
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
```

6. UNIFORM COST SEARCH

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

In [ ]:
```
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(prob
    return(iterations, all_node_colors, node)
```

In [ ]:
```
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)
```

7. 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.

In [ ]:
```
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()

        # 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)
```

In [ ]:
```
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)
```

8. ITERATIVE DEEPENING SEARCH

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

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

In [ ]:
```
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)
```

9. GREEDY BEST FIRST SEARCH

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

In [ ]:
```
def greedy_best_first_search(problem, h=None):
    """Greedy Best-first graph search is an informative searching algorithm
    You need to specify the h function when you call best_first_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    iterations, all_node_colors, node = best_first_graph_search_for_vis(prob
    return(iterations, all_node_colors, node)
```

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

10. A* SEARCH

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

In [ ]:
```
def astar_search_graph(problem, h=None):
    """A* search is best-first graph search with f(n) = g(n)+h(n).
    You need to specify the h function when you call astar_search, or
    else in your Problem subclass."""
    h = memoize(h or problem.h, 'h')
    iterations, all_node_colors, node = best_first_graph_search_for_vis(prob
                                                                        lambda n: n.
    return(iterations, all_node_colors, node)
```

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

11. RECURSIVE BEST FIRST SEARCH

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

In [ ]:
```
def recursive_best_first_search_for_vis(problem, h=None):
    """[Figure 3.26] Recursive best-first search"""
    # we use these two variables at the time of visualizations
    iterations = 0
    all_node_colors = []
    node_colors = {k : 'white' for k in problem.graph.nodes()}

    h = memoize(h or problem.h, 'h')

    def RBFS(problem, node, flimit):
        nonlocal iterations
        def color_city_and_update_map(node, color):
            node_colors[node.state] = color
            nonlocal iterations
            iterations += 1
            all_node_colors.append(dict(node_colors))

        if problem.goal_test(node.state):
            color_city_and_update_map(node, 'green')
            return (iterations, all_node_colors, node), 0  # the second valu

        successors = node.expand(problem)
        if len(successors) == 0:
            color_city_and_update_map(node, 'gray')
            return (iterations, all_node_colors, None), infinity

        for s in successors:
            color_city_and_update_map(s, 'orange')
            s.f = max(s.path_cost + h(s), node.f)
```

```
        while True:
            # Order by lowest f value
            successors.sort(key=lambda x: x.f)
            best = successors[0]
            if best.f > flimit:
                color_city_and_update_map(node, 'gray')
                return (iterations, all_node_colors, None), best.f

            if len(successors) > 1:
                alternative = successors[1].f
            else:
                alternative = infinity

            node_colors[node.state] = 'gray'
            node_colors[best.state] = 'red'
            iterations += 1
            all_node_colors.append(dict(node_colors))
            result, best.f = RBFS(problem, best, min(flimit, alternative))
            if result[2] is not None:
                color_city_and_update_map(node, 'green')
                return result, best.f
            else:
                color_city_and_update_map(node, 'red')

    node = Node(problem.initial)
    node.f = h(node)

    node_colors[node.state] = 'red'
    iterations += 1
    all_node_colors.append(dict(node_colors))
    result, bestf = RBFS(problem, node, infinity)
    return result
```

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

In [ ]:
```
all_node_colors = []
# display_visual(romania_graph_data, user_input=True, algorithm=breadth_firs
algorithms = {  "Breadth First Tree Search": tree_breadth_search_for_vis,
                "Depth First Tree Search": tree_depth_search_for_vis,
                "Breadth First Search": breadth_first_search_graph,
                "Depth First Graph Search": graph_search_for_vis,
                "Best First Graph Search": best_first_graph_search_for_vis,
                "Uniform Cost Search": uniform_cost_search_graph,
                "Depth Limited Search": depth_limited_search_for_vis,
                "Iterative Deepening Search": iterative_deepening_search_for
                "Greedy Best First Search": greedy_best_first_search,
                "A-star Search": astar_search_graph,
                "Recursive Best First Search": recursive_best_first_search_f
display_visual(romania_graph_data, algorithm=algorithms, user_input=True)
```

RECURSIVE BEST-FIRST SEARCH

Recursive best-first search is a simple recursive algorithm that improves upon heuristic search by reducing the memory requirement. RBFS uses only linear space and it attempts to mimic the operation of standard best-first search. Its structure is similar to recursive depth-first search but it doesn't continue indefinitely down the current path, the `f_limit` variable is used to keep track of the f-value of the best *alternative* path available from any ancestor of the current node. RBFS remembers the f-value of the best leaf in the forgotten subtree and can decide whether it is worth re-expanding the tree later. However, RBFS still suffers from excessive node regeneration.
Let's have a look at the implementation.

In [ ]:
```
psource(recursive_best_first_search)
```

This is how `recursive_best_first_search` can solve the `romania_problem`

In [ ]:
```
recursive_best_first_search(romania_problem).solution()
```

`recursive_best_first_search` can be used to solve the 8 puzzle problem too, as discussed later.

In [ ]:
```
puzzle = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
assert puzzle.check_solvability((2, 4, 3, 1, 5, 6, 7, 8, 0))
recursive_best_first_search(puzzle).solution()
```

A* HEURISTICS

Different heuristics provide different efficiency in solving A* problems which are generally defined by the number of explored nodes as well as the branching factor. With the classic 8 puzzle we can show the efficiency of different heuristics through the number of explored nodes.

8 Puzzle Problem

The *8 Puzzle Problem* consists of a 3x3 tray in which the goal is to get the initial configuration to the goal state by shifting the numbered tiles into the blank space.

example:-

```
Initial State                    Goal State
| 7 | 2 | 4 |                  | 1 | 2 | 3 |
| 5 | 0 | 6 |                  | 4 | 5 | 6 |
```

| 8 | 3 | 1 | | 7 | 8 | 0 |

We have a total of 9 blank tiles giving us a total of 9! initial configuration but not all of these are solvable. The solvability of a configuration can be checked by calculating the Inversion Permutation. If the total Inversion Permutation is even then the initial configuration is solvable else the initial configuration is not solvable which means that only 9!/2 initial states lead to a solution.
Let's define our goal state.

In [ ]:
```
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

Heuristics :-

1. Manhattan Distance:- For the 8 puzzle problem Manhattan distance is defined as the distance of a tile from its goal state(for the tile numbered '1' in the initial configuration Manhattan distance is 4 "2 for left and 2 for upward displacement").
2. No. of Misplaced Tiles:- The heuristic calculates the number of misplaced tiles between the current state and goal state.
3. Sqrt of Manhattan Distance:- It calculates the square root of Manhattan distance.
4. Max Heuristic:- It assign the score as the maximum between "Manhattan Distance" and "No. of Misplaced Tiles".

In [ ]:
```
# Heuristics for 8 Puzzle Problem
import math

def linear(node):
    return sum([1 if node.state[i] != goal[i] else 0 for i in range(8)])

def manhattan(node):
    state = node.state
    index_goal = {0:[2,2], 1:[0,0], 2:[0,1], 3:[0,2], 4:[1,0], 5:[1,1], 6:[1
    index_state = {}
    index = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
    x, y = 0, 0

    for i in range(len(state)):
        index_state[state[i]] = index[i]

    mhd = 0

    for i in range(8):
        for j in range(2):
            mhd = abs(index_goal[i][j] - index_state[i][j]) + mhd
```

```
    return mhd

def sqrt_manhattan(node):
    state = node.state
    index_goal = {0:[2,2], 1:[0,0], 2:[0,1], 3:[0,2], 4:[1,0], 5:[1,1], 6:[1
    index_state = {}
    index = [[0,0], [0,1], [0,2], [1,0], [1,1], [1,2], [2,0], [2,1], [2,2]]
    x, y = 0, 0

    for i in range(len(state)):
        index_state[state[i]] = index[i]

    mhd = 0

    for i in range(8):
        for j in range(2):
            mhd = (index_goal[i][j] - index_state[i][j])**2 + mhd

    return math.sqrt(mhd)

def max_heuristic(node):
    score1 = manhattan(node)
    score2 = linear(node)
    return max(score1, score2)
```

We can solve the puzzle using the `astar_search` method.

In [ ]:
```
# Solving the puzzle
puzzle = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
puzzle.check_solvability((2, 4, 3, 1, 5, 6, 7, 8, 0)) # checks whether the i
```

This case is solvable, let's proceed.

The default heuristic function returns the number of misplaced tiles.

In [ ]:
```
astar_search(puzzle).solution()
```

In the following cells, we use different heuristic functions.

In [ ]:
```
astar_search(puzzle, linear).solution()
```

In [ ]:
```
astar_search(puzzle, manhattan).solution()
```

In [ ]:
```
astar_search(puzzle, sqrt_manhattan).solution()
```

In [ ]:
```
astar_search(puzzle, max_heuristic).solution()
```

And here's how `recursive_best_first_search` can be used to solve this problem too.

In [ ]:
```
recursive_best_first_search(puzzle, manhattan).solution()
```

Even though all the heuristic functions give the same solution, the difference lies in the computation time.
This might make all the difference in a scenario where high computational efficiency is required.
Let's define a few puzzle states and time `astar_search` for every heuristic function. We will use the %%timeit magic for this.

In [ ]:
```
puzzle_1 = EightPuzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
puzzle_2 = EightPuzzle((1, 2, 3, 4, 5, 6, 0, 7, 8))
puzzle_3 = EightPuzzle((1, 2, 3, 4, 5, 7, 8, 6, 0))
```

The default heuristic function is the same as the `linear` heuristic function, but we'll still check both.

In [ ]:
```
%%timeit
astar_search(puzzle_1)
astar_search(puzzle_2)
astar_search(puzzle_3)
```

In [ ]:
```
%%timeit
astar_search(puzzle_1, linear)
astar_search(puzzle_2, linear)
astar_search(puzzle_3, linear)
```

In [ ]:
```
%%timeit
astar_search(puzzle_1, manhattan)
astar_search(puzzle_2, manhattan)
astar_search(puzzle_3, manhattan)
```

In [ ]:
```
%%timeit
astar_search(puzzle_1, sqrt_manhattan)
astar_search(puzzle_2, sqrt_manhattan)
astar_search(puzzle_3, sqrt_manhattan)
```

In [ ]:
```
%%timeit
astar_search(puzzle_1, max_heuristic)
astar_search(puzzle_2, max_heuristic)
astar_search(puzzle_3, max_heuristic)
```

We can infer that the `manhattan` heuristic function works the fastest.
`sqrt_manhattan` has an extra `sqrt` operation which makes it quite a lot slower than the others.
`max_heuristic` should have been a bit slower as it calls two functions, but in this case, those values were already calculated which saved some time. Feel free to play around with these functions.

For comparison, this is how RBFS performs on this problem.

In [ ]:
```
%%timeit
recursive_best_first_search(puzzle_1, linear)
recursive_best_first_search(puzzle_2, linear)
recursive_best_first_search(puzzle_3, linear)
```

It is quite a lot slower than `astar_search` as we can see.

In [ ]:
```
# Download your notebook, and upload it as agents.ipynb
# Run the code below and download the agents.html file
!jupyter nbconvert --to webpdf search.ipynb
```