Optimal Search and Route Finding Using UCS

Authors
Mohamed Sawy Abdelsalam
Yousry Essam Ayoub
Ziad Ali Mohamed
Habiba Mostafa Ahmed
Mennatullah Salah Mohamed

CS361 Artificial Intelligence

College of Computers and Information, Assiut University

May 2023

Contents

I Introduction

We're going to talk about some of the problems that are solved by the optimal search, especially the problem of route finding and its solution using the UCS.

I What Is Optimal Search

Optimal search in AI refers to the process of finding not only valid but also the best possible solution to a problem by exploring all possible paths and selecting the one that leads to the most desirable outcome. It involves evaluating different options and choosing the one that minimizes the cost or maximizes the benefit based on a set of predefined criteria.

- Optimal search involves finding a solution that satisfies the following conditions:

1. **Completeness**: The search algorithm should be guaranteed to find a solution if one exists, given a finite search space and a well-defined problem.
2. **Optimality**: The solution path returned by the algorithm should have the lowest cost or the highest value among all possible paths leading to the goal state(s).
3. **Efficiency**: The search algorithm should aim to find the optimal solution in a time and resource-efficient manner, minimizing the number of states or actions examined during the search process.

II Route-Finding and Romania Problem

Route-finding actions are transitions from one location (starting point) to another (goal location). The task of finding the route is summed up in defining the initial state, which is to start from a specific point in a road that will lead to the destination, then we simply move from one state to another. Observe that a road is just a group of states and actions taken, which are connected to reach the specified destination. And that's all about the problem of Romania, being at a specific city what's the **shortest path** to reach the destination?

- Problem formulation:

1. **States**: various cities
2. **Initial state**: The city that the agent starts in
3. **Actions**: Moving from one city to another
4. **Transition model**: $RESULT(C1, A) = C2$ which means that being at city $C1$ and doing an action A will lead to another city $C2$
5. **Goal test**: Complete the path and reach the targeted city

III The Algorithm Chosen to Solve The Romania Problem

There are quite a lot of search algorithms out there such as Depth-First Search, Breadth-First Search, Best-Cost Search, Uniform-Cost Search, A* Search, and others. Nevertheless, the fact that we're trying to get the **shortest path** optimally leaves us with a smaller subset with **BFS** as the simplest algorithm to do this task but due to the non-constant edge costs, it won't work as it assumes that the cost is always 1 (or the same for all actions). Therefore the **Uniform-Cost Search** algorithm will be a more suitable choice as it's a more general form of BFS considering edge costs and that'll be our way to go.

II Methodology

As we stated, we're going to use the Uniform Cost Search (UCS) algorithm to find the shortest path between the two cities. It can be applied to solve the Romania problem by considering the map of Romania as a graph, where each city represents a node and the distances between cities are represented as edge costs. The algorithm aims to find the shortest path between a given start city and a goal city by exploring graph nodes while considering the cost of each path and always selecting the path with the lowest cumulative cost.

I The Formal Algorithmic Steps

To formally apply the UCS algorithm to the Romania problem and be able to solve it, we follow these steps:

1. Define the Romania map as an undirected weighted graph $G = (V, E)$, where V represents a set of nodes (cities in Romania), and E represents a set of edges (roads) connecting the nodes. Each edge $(u, v) \in E$ is associated with a non-negative cost or distance $C(u, v)$ representing the travel distance between cities u and v.
2. Specify the starting city **S** and the target city **T** for the search problem. These cities should be elements of the set V.
3. Use a priority queue to prioritize the nodes to be explored based on their cumulative cost.
4. Maintain a set of visited nodes so far, as well as dictionaries to keep track of the cumulative cost to reach each node and the predecessors of each node along the path.
5. Explore graph **G** by expanding nodes starting from city **S**.
6. Select the node with the lowest cumulative cost from the priority queue and examines its neighbors.
7. For each unvisited neighbor, calculate the new cumulative cost by adding the edge cost between the current node and the neighbor.
8. If the neighbor has not been visited before or the new cumulative cost is lower than the previously recorded cost for that neighbor, update the cumulative cost, add the node to the priority in the priority queue, and also record the current node as the predecessor of its neighbor.
9. Continue exploring the graph until you reach the targeted city **T** or exhaust all of the possible paths.
10. If the targeted city **T** is reached, reconstruct the shortest path from the starting city **S** to the targeted city **T** using the predecessor information stored in the dictionaries.
11. Start from the targeted city **T** and iteratively follows the predecessors until it reaches the start city **S**, recording each city in the path. Note that the path should be reversed to start from the city **S** to **T**.
12. Finally, return the reconstructed shortest path as a list of cities and the total cost associated with that path or state that it cannot be reached.

II Algorithm Illustration

To understand the algorithm properly let's visualize it using a simple flowchart that provides a concise overview of the straightforward steps of the algorithm, and a more detailed pseudo-code. To ease the process of implementing the algorithm, that's discussed in section[III] and implemented in [A].

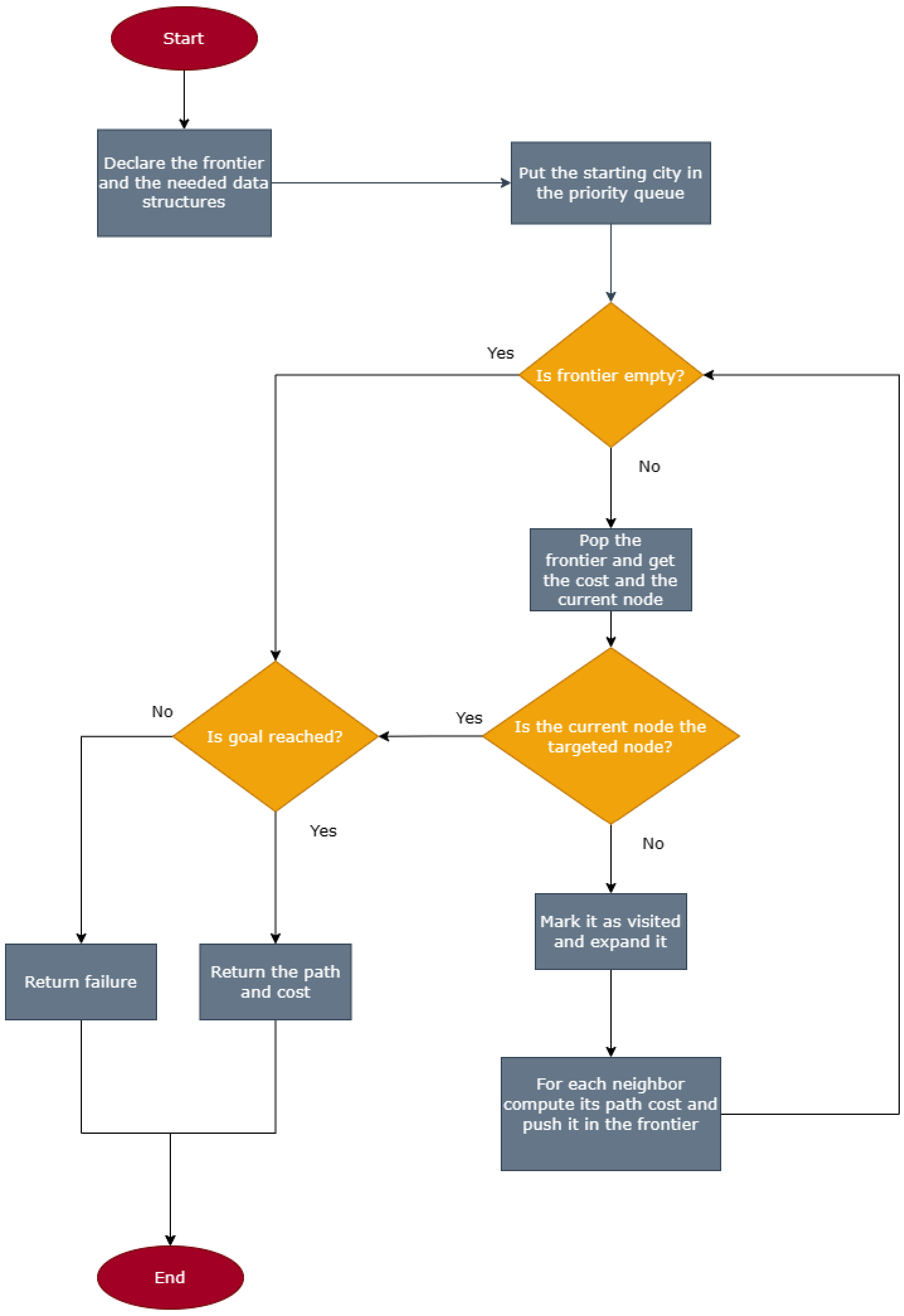
Start
Declare the frontier and the needed data structures
Put the starting city in the priority queue
Is frontier empty?
Yes
No
Pop the frontier and get the cost and the current node
Is the current node the targeted node?
Yes
No
Is goal reached?
No
Yes
Mark it as visited and expand it
For each neighbor compute its path cost and push it in the frontier
Return failure
Return the path and cost
End

Figure 1: A flowchart of the UCS algorithm

Algorithm 1 Uniform Cost Search (UCS)

```
1: function UNIFORM_COST_SEARCH(graph, start, goal) return a solution path and cost or failure
2:     if $start$ not in $graph$ then
3:         return failure
4:     end if
5:     $visited \leftarrow$ Set()
6:     $frontier \leftarrow$ PriorityQueue()
7:     $frontier$.put((0, $start$))
8:     $previous\_node \leftarrow$ Dictionary()
9:     $goal\_reached \leftarrow$ False
10:    $cost\_so\_far \leftarrow$ Dictionary()
11:    $cost\_so\_far$.put(start: 0)
12:    $path \leftarrow$ List()
13:    $total\_cost \leftarrow 0$
14:    while not $is\_empty(frontier)$ do
15:        $current\_cost, current\_node \leftarrow pop(frontier)$
16:        if $current\_node = goal$ then
17:            $goal\_reached \leftarrow$ True
18:            break
19:        end if
20:        $visited$.add($current\_node$)
21:        for each $next\_node, edge\_cost$ in $graph[current\_node]$ do
22:            $new\_cost \leftarrow cost\_so\_far[current\_node] + edge\_cost$
23:            if $next\_node \notin cost\_so\_far$ or $new\_cost < cost\_so\_far[next\_node]$ then
24:                $cost\_so\_far[next\_node] \leftarrow new\_cost$
25:                $frontier$.put(($new\_cost$, $next\_node$))
26:                $previous\_node[next\_node] \leftarrow current\_node$
27:            end if
28:        end for
29:    end while
30:    if $goal\_reached$ is False then
31:        return failure
32:    end if
33:    $current \leftarrow goal$
34:    while $current \neq start$ do
35:        $path$.append($current$)
36:        $total\_cost \leftarrow total\_cost + graph[previous\_node[current]][current]$
37:        $current \leftarrow previous\_node[current]$
38:    end while
39:    $path$.append($start$)
40:    $path$.reverse()
41:    return $path$, $total\_cost$
42: end function
```

III Algorithm Complexity

The complexity of Uniform-Cost Search is characterized in terms of the cost of the optimal solution C^*, and a lower bound on the cost of each action ϵ where $\epsilon > 0$. The algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ where b is the number of successors for each state. However it can be much greater than $O(b^d)$, where d is the depth of the tree, this is because the Uniform-Cost Search algorithm can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^d, and Uniform-Cost Search becomes similar to Breadth-First Search.[1]

III Experimental Simulation

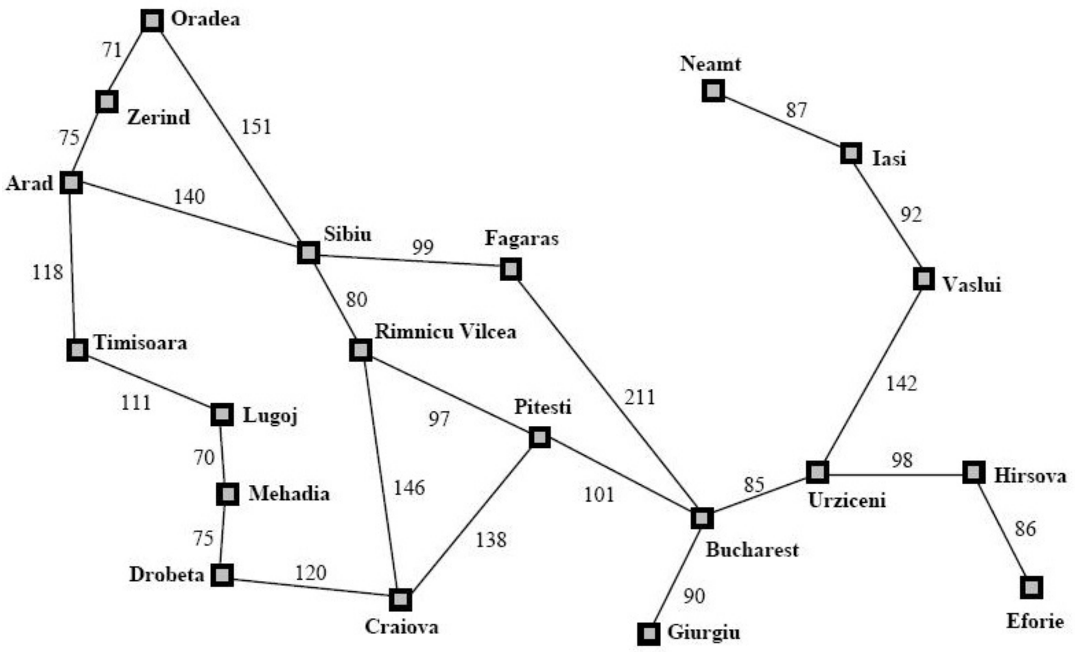

Figure 2: A simplified road map of part of Romania, with road distances in miles.[1]

The experiment is applied to the Romania Map which is shown in Fig[2] and the idea is about finding the shortest available path between the current location and a target city using the Uniform Cost Search algorithm.

I Programming Languages and Environments

We implemented the UCS algorithm in Python. It was the primary language used in our experiments due to its simplicity, readability, and the variety of libraries for data analysis and visualization in addition to its ease to run the algorithm on almost any machine by only installing Python. And for the problem-solving agent, the environment will be any given map/graph which is in this case the Romania map.

II Details of Programming The Primary Function

The primary function of our program, **uniform_cost_search**(graph, start, goal), takes three input parameters:

1. **graph**: which is the map and it's represented as a dictionary with each key representing a node (a city in Romania) and its value is another dictionary representing its neighbors (cities) and edge weights (distance in miles) as key-value pairs.
2. **start**: the starting city is represented as a string.
3. **goal**: the target city is represented as a string.

And it returns a tuple of:

1. **list of strings** that represents the path from the start to the goal.
2. **integer** that represents the total cost of this path.

After we talked about the parameters and the return value of the function let's take a look at some important variables that should be initialized:

- **visited:** a set used to mark each city that has been processed to prevent processing the same node multiple times, and it's initially empty.

- **frontier:** a priority queue used to explore nodes and sort them based on their path cost to get the minimum path-cost node at each extraction, and initially it has only the starting node with path cost 0.
- **previous_node:** a dictionary storing the previous node for each node to help construct the path between the start and goal nodes, and it's initially empty.
- **goal_reached:** a boolean used to check if we reached the goal node, and it's initially False.
- **cost_so_far:** a dictionary that's used to store the current cost of reaching any node from the start node, and initially it has the start node with cost 0.
- **path and total_cost:** the tuple to be returned that's described above and they're initially empty and 0 respectively.

Now we're ready to deep dive into the internal details of the function and how the algorithm works. The UCS algorithm's main idea is about maintaining a priority queue of nodes to be explored, sorted based on their path cost. The algorithm repeatedly extracts the node with the lowest path cost from the queue, checks if it's the goal then the loop breaks, otherwise marks the current node as visited, explores its neighbors, computes the new cost, and adds them to the queue if they haven't been visited before or the new cost of reaching this node is less than any previous cost. This procedure terminates when the goal node is reached or when the queue is empty. In the end, it constructs the path and the cost and returns them.

IV Results and Technical Discussion

As we discussed the algorithm, how it works, and its implementation in the previous section now we need to make sure that it's correct and try to evaluate and analyze the algorithm's performance. So, it's time for testing.

I Test Cases

- For this section let's try some straightforward test cases and see if it's working correctly.

1. Output for [Bucharest Sibiu] [1]

```
Start City:  Bucharest
Target City:  Sibiu
Path:  ['Bucharest', 'Pitesti', 'Rimnicu Vilcea', 'Sibiu']
Total Cost: 278
```

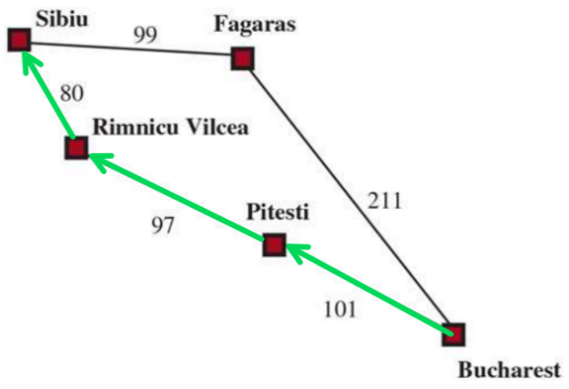

Figure 3: Part of the Romania state space, selected to illustrate the path between Bucharest and Sibiu.

2. Output for [Eforie Oradea]

```
Start City:  Eforie
Target City:  Oradea
Path:  ['Eforie', 'Hirsova', 'Urziceni', 'Bucharest', 'Pitesti', 'Rimnicu Vilcea',
'Sibiu', 'Oradea']
Total Cost: 698
```

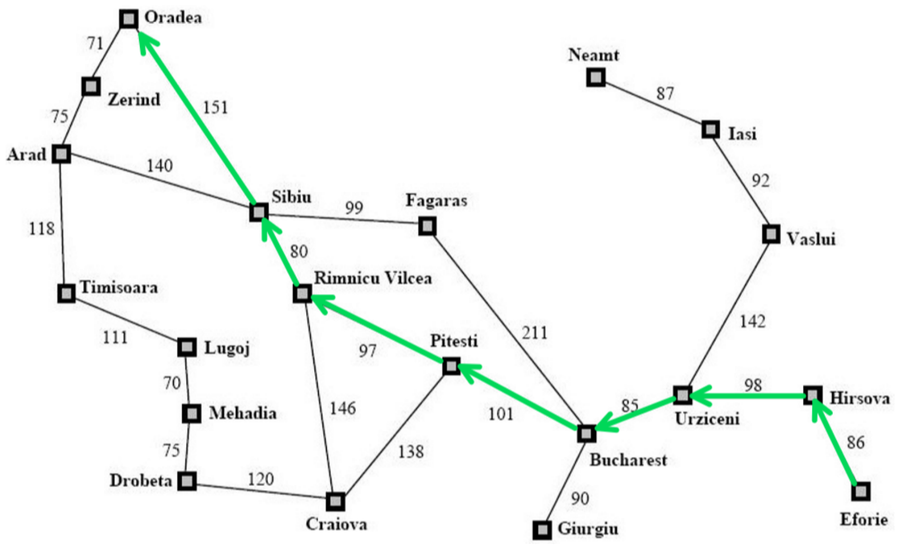

Figure 4: Figure of the Romania state space, selected to illustrate the path between Eforie and Oradea.

3. Output for [Neamt Timisoara]

```
Start City:  Neamt
Target City:  Timisoara
Path:  ['Neamt', 'Iasi', 'Vaslui', 'Urziceni', 'Bucharest', 'Pitesti',
'Rimnicu Vilcea', 'Sibiu', 'Arad', 'Timisoara']
Total Cost: 942
```

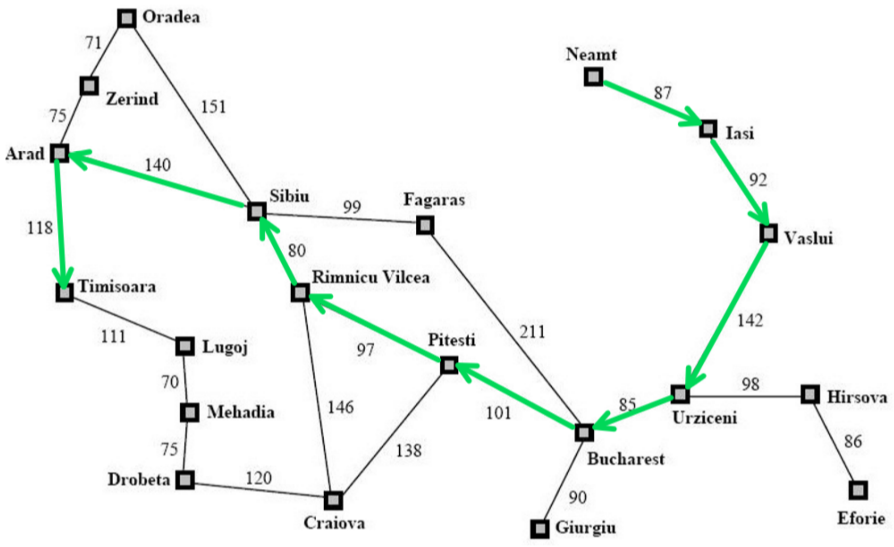

Figure 5: Figure of the Romania state space, selected to illustrate the path between Neamt and Timisoara.

II Edge Cases

- Now let's make it more challenging by running some tricky test cases.

1. Output for a test case that includes an invalid start city [Cairo to Bucharest].

```
Start City:  Cairo
Target City:  Bucharest
Path:  Invalid starting point
Total Cost: -1
```

2. Output for a test case that includes an invalid goal city [Bucharest to Cairo].

```
Start City:  Bucharest
Target City:  Cairo
Path:  Cannot be reached
Total Cost: -1
```

3. Output for a test case that includes the same start and goal city [Bucharest Bucharest].

```
Start City:  Bucharest
Target City:  Bucharest
Path:  ['Bucharest']
Total Cost: 0
```

III Technical Discussion

According to the previous results, it looks like the algorithm is working correctly and it handles invalid inputs and edge cases which implies that the algorithm is **Complete** and it can find the solution if it exists or reports that it doesn't exist, also it's **Optimal** as the performance is quite good without wasting many resources. Let's discuss the complexity in more detail.

1. **Time Complexity:**

   The time complexity of uniform cost search depends on the number of nodes and edges in the graph, as well as the cost of traversing each edge. In the worst case, where all nodes are explored, the time complexity is $O((|V| + |E|)log|V|)$, where $|V|$ is the number of nodes (vertices) and $|E|$ is the number of edges in the graph. The Priority Queue operations (put and get) have a time complexity of $O(logn)$, where n is the number of elements in the queue. The main time-consuming step is the while loop, which iterates until the frontier is empty which means you are gonna visit most of the nodes and edges. Each iteration involves removing a node from the frontier to check its neighbors, which takes $O(log|V|)$ time. Overall, the time complexity of the code is dominated by the while loop and the Priority Queue operations, resulting in a time complexity of $O((|V| + |E|)log|V|)$.

2. **Space Complexity:**

   The space complexity is determined by the additional data structures used during the search, including the **visited** set, **frontier** (Priority Queue), **previous_node** dictionary, and **cost_so_far** dictionary. The space required to store these data structures depends on the number of nodes in the graph. In the worst case, where all nodes are visited, the space complexity is $O(|V|)$, as each node is stored in the visited set, frontier, and dictionaries. The space complexity can be further increased if the graph is represented with an adjacency matrix instead of an adjacency list, resulting in a space complexity of $O(|V|^2)$. Additionally, the **path** list stores the path from the start node to the goal node, which can have a length of at most $|V|$ in the worst case. Therefore, the overall space complexity of the code is $O(|V|)$. In summary, the time complexity of the code is $O((|V| + |E|)log|V|)$, and the space complexity is $O(|V|)$.

   Observing the previous details means the uniform cost search algorithm provides an optimal solution for finding the minimum-cost path from a start node to a goal node as it explores nodes in increasing order of their cumulative costs from the start node. It ensures that the node with the lowest cost is always selected next for exploration. This property guarantees that when the goal node is reached, the accumulated cost to reach it is the minimum among all possible paths from the start node to the goal node. By considering the cost of each edge and updating the **cost_so_far** dictionary accordingly, the algorithm explores paths in a way that minimizes the total cost. It continues until it reaches the goal node or exhausts all possible paths in the graph. Therefore, the resulting path and total cost obtained from the uniform cost search are indeed optimal for the given problem.

References

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Pearson, 4th ed., 2020.

A Project Source Codes

```python
from queue import PriorityQueue


def uniform_cost_search(graph, start, goal):
    if start not in graph:
        return "Invalid starting point", -1

    visited = set()
    frontier = PriorityQueue()
    frontier.put((0, start))  # (distance, node) the priority is the cost
    previous_node = {}  # dictionary to store the path
    goal_reached = False
    cost_so_far = {start: 0}
    path = []
    total_cost = 0

    while not frontier.empty():
        current_cost, current_node = frontier.get()

        if current_node == goal:
            goal_reached = True
            break

        visited.add(current_node)

        for next_node, edge_cost in graph[current_node].items():
            new_cost = cost_so_far[current_node] + edge_cost

            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                frontier.put((new_cost, next_node))
                previous_node[next_node] = current_node

    if goal_reached is False:
        return "Cannot be reached", -1

    # Reconstruct the path from start to goal
    current = goal
    while current != start:
        path.append(current)
        total_cost += graph[previous_node[current]][current]
        current = previous_node[current]
    path.append(start)
    path.reverse()

    return path, total_cost


# Graph representation of the Romanian Map
graph = {
    'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Craiova': {'Rimnicu Vilcea': 146, 'Pitesti': 138, 'Drobeta': 120},
    'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
    'Giurgiu': {'Bucharest': 90},
    'Urziceni': {'Bucharest': 85, 'Vaslui': 142, 'Hirsova': 98},
    'Vaslui': {'Urziceni': 142, 'Iasi': 92},
    'Hirsova': {'Urziceni': 98, 'Eforie': 86},
    'Iasi': {'Vaslui': 92, 'Neamt': 87},
    'Neamt': {'Iasi': 87},
    'Eforie': {'Hirsova': 86}
}

start_node = 'Bucharest'
goal_node = 'Sibiu'
result, total_cost = uniform_cost_search(graph, start_node, goal_node)
print("Start City: ", start_node, "\nTarget City: ", goal_node)
print("Path: ", result)
print("Total Cost:", total_cost)
```

A Project Team Plan and Achievement

Here's an overview of the project team plan and the achievements accomplished throughout the duration of the project. The team's composition, roles and responsibilities, and key milestones achieved are outlined below.

The first step was to ensure effective collaboration, that's why each team member was assigned specific roles and responsibilities. The following were the key responsibilities undertaken by each team member:

- **Introduction:**
  - Habiba Mostafa Ahmed
  - Mennatullah Salah Mohamed
- **Methodology:**
  - Ziad Ali Mohamed
- **Experimental Simulation, Results, and Technical Discussion:**
  - Mohamed Sawy Abdelsalam
  - Yousry Essam Ayoub

Secondly, we chose the tools that we're going to use to increase productivity and get the most optimal outcomes, so we chose Overleaf, Online LaTeX Editor to get a professional collaborative environment.

Furthermore, we established a flexible timeline with an early deadline for the main piece of the project which is the report, that allowed us to work freely and be more productive.

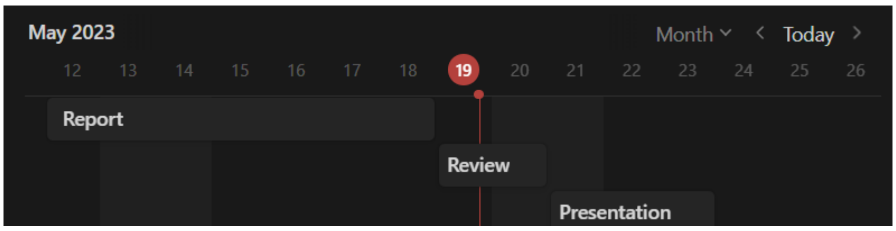

Figure 6: Project expected timeline

Throughout the project life cycle, the team accomplished several milestones, demonstrating progress and successful outcomes, such as:

1. Completion of Project Planning: We successfully defined project goals, deliverables, and timelines, laying the foundation for subsequent activities.
2. Learning LaTeX, Choosing a suitable template, and Structuring the report as required.
3. Completion of the Project with a three-day buffer ahead of the deadline.

A Link to the presentation file