

## INTRODUCTION

The goal of this project is to simulate a Uniform-Cost Search Program using Python Language. Uniform-Cost Search is a variant of Dijkstra's algorithm. Instead of inserting all vertices into a priority queue, we insert only the source, then insert one by one as needed. At each step, we check to see if the item is already in the priority queue (using the visited array). If yes, we perform the decrease key; otherwise, we insert it. (*Uniform-Cost Search (Dijkstra for Large Graphs)*, 2023).

The UCS algorithm explores the graph by taking into account the cost of each path and expanding nodes with the lowest cost first. The code accepts a graph representation, a start node, and one or more goal nodes as input and uses the UCS algorithm to calculate the optimal path and its associated cost. It chooses the next node to explore based on the total cost thus far, gradually constructing the optimal path until it reaches the goal node(s). This algorithm is widely used in a variety of applications, including route planning, puzzle solving, and AI systems.

## CODE IMPLEMENTATION AND PSEUDOCODE

### Implementation Details

The language that was used to implement the Uniform Cost Search Algorithm (UCS) is Python.

The UCS hinges on a priority queue data structure where elements popped from a set are guaranteed to be the minimum or maximum cumulative cost, depending on the goal. For this implementation, the `.sort()` built-in method by Python is used in an array called *frontier*, then the first element in the *frontier* is popped.

To get the cumulative cost of each node, a dynamic programming technique is used. To get the cumulative cost of the succeeding level of nodes, the cumulative cost of the current node and the edge cost of the succeeding node is added. This sum is then associated with the successor using a tuple data structure.

For more insight, the tuple data structure is implemented as follows. It has 3 values per tuple, making it a triplet. The structure of it would be *(cost, key\_node, (path))*. The *cost* would be the cumulative cost of the node while the *key\_node* is the name of the node while *path* is a sequence of *key\_node* which contains all the nodes that were passed to get to a particular node. A proper node structure is not implemented in this project as the group believes it would be much simpler to simulate a node using this

tuple setup. As seen in our source code, it only contains code less than 100 lines all in all.

The Breadth First Search Algorithm (BFS) is used to locate neighboring nodes. For each neighbor, a *new\_cost* is grouped to them using the dynamic programming technique. This is then pushed into the queue or what is called *frontier* which will then be sorted and popped.

All of this will be repeated until a goal node is popped out of *frontier* or when *frontier* is empty.

For our implementation, the minimum cost is calculated.

### Pseudocode

Let *frontier* be the array of triplet

Let *.sort()* sort the *frontier* by cost in ascending order.

Let *.pop()* pop the first element in *frontier*

Let *in* return a boolean value for the presence of an element in an array

Let *continue* skip an iteration of a loop

	<i>function UCS(graph, start, goal_nodes):</i>
1	<i>frontier = [(0, start, [ ])]</i>
2	<i>explored = [ ]</i>
3	<i>while frontier.length &gt; 0:</i>
4	<i>frontier.sort()</i>
5	<i>cummulative_cost, current_node, path = frontier.pop()</i>
6	<i>if current_node in explored:</i>
7	<i>continue</i>
8	<i>if current_node in goal_nodes:</i>
9	<i>return path + current node, cost</i>
10	<i>explored.push(current_node)</i>

11	<i>for neighbor_node, edge_cost in graph[current_node].neighbors:</i>
12	<i>if neighbor not in explored:</i>
13	<i>new_cost = cummulutive_cost + edge_cost</i>
14	<i>new_path = path + current_node</i>
15	<i>frontier.push((new_cost, neighbor, new_path))</i>
16	<i>return NIL</i>

Line 1 initializes the *frontier* with the starting Tuple. It has no cost and path yet.

Line 3 and Lines 8-9 are the conditions for whether the loop will be terminated. If the *frontier* runs out of elements, it indicates that there is no path possible to reach the goal state. But, if a goal state is popped out of the *frontier*, that would be the state with the lowest cost with a path.

Line 4 sorts the *frontier*. This makes sure that at any moment, the tuple that is being popped has the lowest cost.

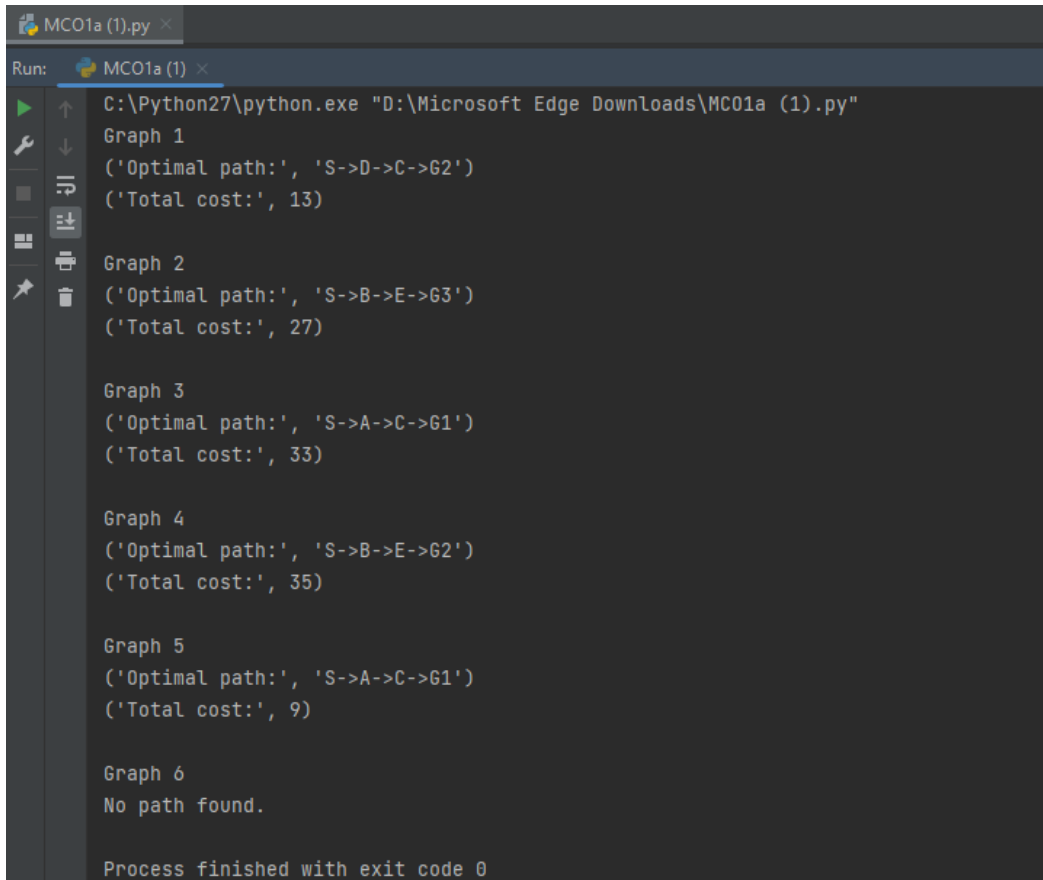
Line 5 pops the *frontier* and assigns each element in the tuple to *cumulative\_cost*, *current\_node*, and *path* respectively.

Line 11 uses the BFS to explore neighboring nodes.

For lines 13-15, as stated earlier, calculate the cumulative cost and the path for neighboring nodes.

Line 12's purpose is to exclude already explored nodes in the *frontier*. Lines 6-7's prevent nodes from being popped when it's already been explored. There are cases when a node has already been explored but is still in the *frontier*.

## RESULTS AND ANALYSIS



```
MC01a (1).py x
Run: MC01a (1) x
C:\Python27\python.exe "D:\Microsoft Edge Downloads\MC01a (1).py"
Graph 1
('Optimal path:', 'S->D->C->G2')
('Total cost:', 13)
Graph 2
('Optimal path:', 'S->B->E->G3')
('Total cost:', 27)
Graph 3
('Optimal path:', 'S->A->C->G1')
('Total cost:', 33)
Graph 4
('Optimal path:', 'S->B->E->G2')
('Total cost:', 35)
Graph 5
('Optimal path:', 'S->A->C->G1')
('Total cost:', 9)
Graph 6
No path found.
Process finished with exit code 0
```

Figure 1. Screenshot of output after running the code

After inputting the full graph into the program and running the Uniform Cost Search algorithm to find the lowest cost path, two main things are presented to the user for each graph: the optimal path, and the total cost. The optimal path shows the path taken to reach the goal, from one node to another, up until the goal node. The total cost is the accumulated path cost that the algorithm calculated when taking the optimal path.

After performing multiple tests with the algorithm, we can conclude that the main strength of the Uniform Cost Search algorithm is that it is quite optimal as it is somewhat similar to a greedy algorithm, in that it always chooses the path with the lowest cost. It is also a complete algorithm as it goes through the entire graph and does not immediately stop once it has reached a goal state. It also avoids getting stuck in infinite loops as it keeps track of all the nodes that the algorithm has already visited, removing the need to visit nodes more than once.