Graph Search:Uniform Cost Search

Uniform Cost Search (UCS) is a widely used graph search algorithm. The goal of UCS is to find the path from a starting node to a goal node with the most minor cost. Uniform-Cost Search is similar to Dijkstra's algorithm.

UCS is an informed search algorithm, which means that it uses heuristic information to guide its search. The heuristic function used in UCS is the path's cost from the starting node to the current node. The algorithm maintains a priority queue of nodes to be expanded, where the nodes are ordered by their path costs. The node with the smallest path cost is always chosen for expansion.

UCS is guaranteed to find the optimal solution, assuming that the path costs are non-negative. This is because the algorithm always chooses the node with the smallest path cost for expansion. Additionally, UCS can handle graphs with cycles and graphs with infinite paths.

UCS is a powerful algorithm for searching a graph, and it is widely used in artificial intelligence and robotics. However, it can be slow for large graphs or graphs with high branching factors, since it explores all nodes in order of increasing path cost.

Uniform Cost Search (UCS) is a variant of Dijkstra's algorithm that is used to find the shortest path from a source node to a target node in a weighted graph. It is guaranteed to find the optimal path, if the weights are non-negative.

The algorithm works by maintaining a priority queue of nodes to be explored, where the priority of each node is determined by the cumulative cost of the path from the source node to that node. At each iteration of the algorithm, the node with the lowest priority is selected from the queue and its adjacent nodes are examined.

Here is a step-by-step breakdown of the UCS algorithm:

1. Initialize the starting node: Set the distance of the starting node to 0 and insert it into the priority queue.
2. Repeat until the priority queue is empty:
   a. Select the node with the lowest priority (i.e., the smallest cumulative cost) from the priority queue.
   b. If the selected node is the target node, stop the search and return the path.
   c. Otherwise, for each neighbor of the selected node:

I. Calculate the cumulative cost of the path to the neighbor by adding the cost of the edge from the selected node to the neighbor to the cumulative cost of the path to the selected node.

i. If the neighbor has not been visited or if the new cumulative cost is lower than the previous one, update the neighbor's cumulative cost and insert it into the priority queue.

3. If the target node was not found, return "target not found".

Logic:

The UCS algorithm can be broken down into the following steps:

1. Initialization: The algorithm starts by setting the distance of the source node to 0 and inserting it into the priority queue.

2. Exploration: The algorithm repeatedly selects the node with the lowest priority (i.e., the smallest cumulative cost) from the priority queue and examines its adjacent nodes.

3. Path update: For each adjacent node, the algorithm calculates the cumulative cost of the path to the node by adding the cost of the edge from the selected node to the adjacent node to the cumulative cost of the path to the selected node. If the adjacent node has not been visited or if the new cumulative cost is lower than the previous one, the algorithm updates the adjacent node's cumulative cost and inserts it into the priority queue.

4. Goal check: If the target node is found during the exploration phase, the algorithm terminates and returns the path to the target node.

5. Termination: If the target node is not found, the algorithm terminates and returns "target not found".

The UCS algorithm works by maintaining a priority queue of nodes to be explored, where the priority of each node is determined by the cumulative cost of the path from the source node to that node. This priority queue is implemented as a min-heap, which allows for efficient retrieval of the node with the lowest priority.

During the exploration phase, the algorithm repeatedly selects the node with the lowest priority from the priority queue and examines its adjacent nodes. The adjacent nodes are then processed, and if a better path to an adjacent node is found, the adjacent node's cumulative cost is updated, and it is inserted back into the priority queue.

The UCS algorithm guarantees that it will find the optimal path from the source node to the target node, if the weights of the edges are non-negative. This is because the algorithm always selects the node with

the lowest priority (i.e., the smallest cumulative cost) from the priority queue, which ensures that it explores the path with the lowest cost first.

However, the UCS algorithm can be slow for large graphs or graphs with high branching factors, since it explores all nodes in order of increasing path cost. Additionally, the UCS algorithm may not be suitable for graphs with negative edge weights, as the algorithm assumes that all edge weights are non-negative.

Here is the pseudocode for the UCS algorithm:

```
function uniform_cost_search(start, goal)
initialize the priority queue with the start node initialize an empty set of
visited nodes while the priority queue is not empty:
dequeue the node with the lowest priority from the queue if the dequeued node is
the goal node:
return the path to the goal node if the dequeued node is not in the
visited set:
add the dequeued node to the visited set for each neighbor of
the dequeued node:
calculate the cumulative cost of the path to the neighbor if the neighbor has not been
visited or if the new cumulative cost is lower than the previous one:
update the neighbor's cumulative cost and insert it into the priority queue return "target not
found"
```

Now let's break down each line of the pseudocode in detail:

The uniform_cost_search function takes two parameters: the starting node and the goal node. This function implements the UCS algorithm and returns the path to the goal node if it is found.

We initialize the priority queue with the starting node. We will use a priority queue to store nodes in order of increasing path cost.

We also initialize an empty set of visited nodes. We will use this set to keep track of which nodes have already been visited.

We start a loop that will continue until the priority queue is empty. We will explore nodes in the priority queue until we find the goal node or we exhaust all possibilities.

We dequeue the node with the lowest priority from the priority queue. We choose the node with the lowest priority because it has the smallest cumulative cost to reach that node.

If the dequeued node is the goal node, we have found the shortest path to the goal node, so we return the path to the goal node.

If the dequeued node is not the goal node, we need to continue exploring its neighbors.

We check whether the dequeued node has been visited before. If it has, we skip it because we've already explored all its neighbors.

If the dequeued node has not been visited, we add it to the visited set.

We loop through all the neighbors of the dequeued node.

For each neighbor, we calculate the cumulative cost of the path to the neighbor. We do this by adding the cost of the edge from the dequeued node to the neighbor to the cumulative cost of the path to the dequeued node.

We check whether the neighbor has been visited before or if the new cumulative cost is lower than the previous one. If either of these conditions is true, we update the neighbor's cumulative cost and insert it into the priority queue.

If we update the neighbor's cumulative cost, we add it to the priority queue. This ensures that we explore the neighbor nodes in order of increasing cumulative cost.

If the target node was not found, we return "target not found".

Code Example:

Here is the python implementation of the algorithm:

```
import heapq

def uniform_cost_search(start, goal, graph):
    """Performs uniform cost search from start to goal in graph""" # Initialize priority queue
    with start node and cost 0 pq = [(0, start, [])]
    # Initialize set of visited nodes visited = set()

    while pq:
        # Dequeue node with lowest cost from priority queue
        (cost, node, path) = heapq.heappop(pq)
```

```
            # If goal node found, return path if node == goal:
            return path + [node]

            # Add node to visited set visited.add(node)

            # Explore neighbors for neighbor, edge_cost in graph[node].items(): #
            Calculate cumulative cost of path to neighbor new_cost = cost + edge_cost
                                                    # If neighbor not visited or new cost is lower than
previous cost if neighbor not in visited:
                                # Add neighbor to priority queue with new cost heapq.heappush(pq, (new_cost,
                                        neighbor, path +
[node]))

    # If goal not found, return empty path return []
```

Now let's go through the code line by line to understand it better:

We import the heapq module, which provides an implementation of a heap (priority queue) in Python.

We define the uniform_cost_search function, which takes three arguments: the starting node, the goal node, and the graph in which the search will be performed.

We initialize the priority queue (pq) with a tuple containing the cost (0 since we start from the starting node), the starting node, and an empty path.

We initialize the set of visited nodes (visited) to an empty set.

We start a loop that will continue until the priority queue is empty. We will explore nodes in the priority queue until we find the goal node or we exhaust all possibilities.

We dequeue the node with the lowest cost from the priority queue using heapq.heappop(pq). The heappop() function removes and returns the smallest item from the heap (priority queue) based on the first element of each tuple in the queue, which is the cost in our case.

If the dequeued node is the goal node, we have found the shortest path to the goal node, so we return the path to the goal node by adding the current node to the path.

We add the dequeued node to the visited set.

We loop through all the neighbors of the dequeued node using the items() method of the dictionary graph[node]. The graph dictionary contains the edges and their corresponding costs for each node.

For each neighbor, we calculate the cumulative cost of the path to the neighbor by adding the cost of the edge from the dequeued node to the neighbor to the cumulative cost of the path to the dequeued node.

We check whether the neighbor has been visited before or if the new cumulative cost is lower than the previous one. If either of these conditions is true, we add the neighbor to the priority queue with the new cost and the updated path.

If the target node was not found, we return an empty path.

Time Complexity:

The time complexity of the Uniform Cost Search (UCS) algorithm can be analyzed in terms of the number of nodes expanded and the branching factor of the search tree.

In the worst case, UCS can expand all the nodes in the search tree until it finds the goal state, which makes the time complexity exponential. This can happen if the branching factor of the search tree is very high or if the cost of the optimal path is very large.

However, the time complexity of UCS is typically much lower in practice, especially when the cost function is monotonic, meaning that the cost of a path never decreases as more nodes are added to it. In this case, UCS is guaranteed to find the optimal solution, and its time complexity is proportional to the number of nodes expanded.

The time complexity of UCS can also be affected by the data structure used to implement the fringe, which is the set of nodes that have been generated but not yet expanded. Using a priority queue to store the fringe can reduce the time complexity of UCS by allowing it to prioritize nodes with lower costs. The time complexity of UCS with a priority queue is O(b^(C*/ε)), where b is the branching factor, C* is the cost of the optimal solution, and ε is the minimum cost difference between any two adjacent nodes.

Space Complexity:

The space complexity of the Uniform Cost Search (UCS) algorithm is determined by the amount of memory needed to store the search tree and the nodes that are generated during the search.

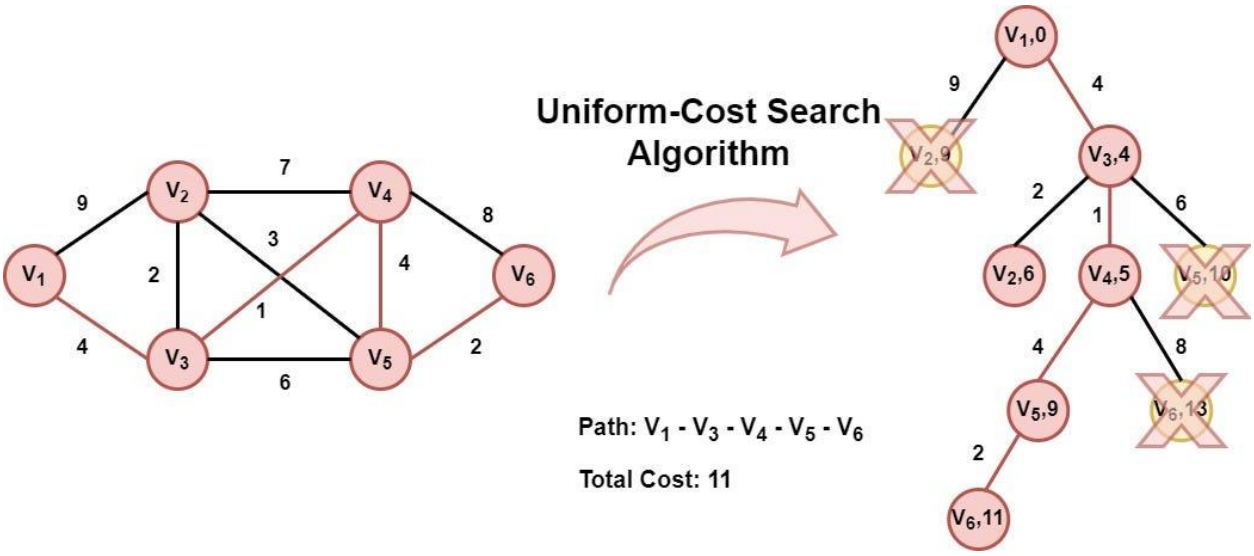
UCS uses a fringe data structure to keep track of the nodes that have been generated but not yet expanded. The fringe can be implemented using a variety of data structures, such as a queue, a stack, or a priority

queue. Each of these data structures has different space requirements, with the priority queue being the most space-efficient option.

In the worst case, UCS may need to store all the nodes in the search tree in memory, which makes the space complexity exponential. This can happen if the branching factor of the search tree is very high or if the cost of the optimal path is very large. However, in practice, UCS often prunes parts of the search tree that are unlikely to lead to the optimal solution, which can reduce the space complexity significantly.

Uniform Cost Search (UCS) Algorithm in Python:

Using the Uniform Search Algorithm to find the best solution in a graph modeled problem.



search algorithms like Breadth-First Search, Depth First Search, the Greedy algorithm, and of course the UCS algorithm, are used to find a solution in a graph. The graph represents the given problem, in which each vertex (node) of the graph represents a state of the problem and each edge represents a valid action that takes us from one state (vertex) to the other.

Graph Data Structure — Theory and Python Implementation

In contrast to BFS and DFS algorithms that don't take into consideration neither the cost between two nodes nor any heuristic value, the Greedy algorithm uses a heuristic value, such as the Manhattan distance, or the Euclidean distance to estimate the distance from the current node to the target node. On the other hand, the UCS algorithm uses the path's cost from the initial node to the current node as the extension criterion**. Starting from the initial state (starting node), the UCS algorithm, in each step chooses the node that is **closer **to the initial node. When the algorithm finds the solution, returns the path from the initial state to the final state. The UCS algorithm is characterized as complete, as it always returns a solution if exists. Moreover, the UCS algorithm guarantees the optimum solution.

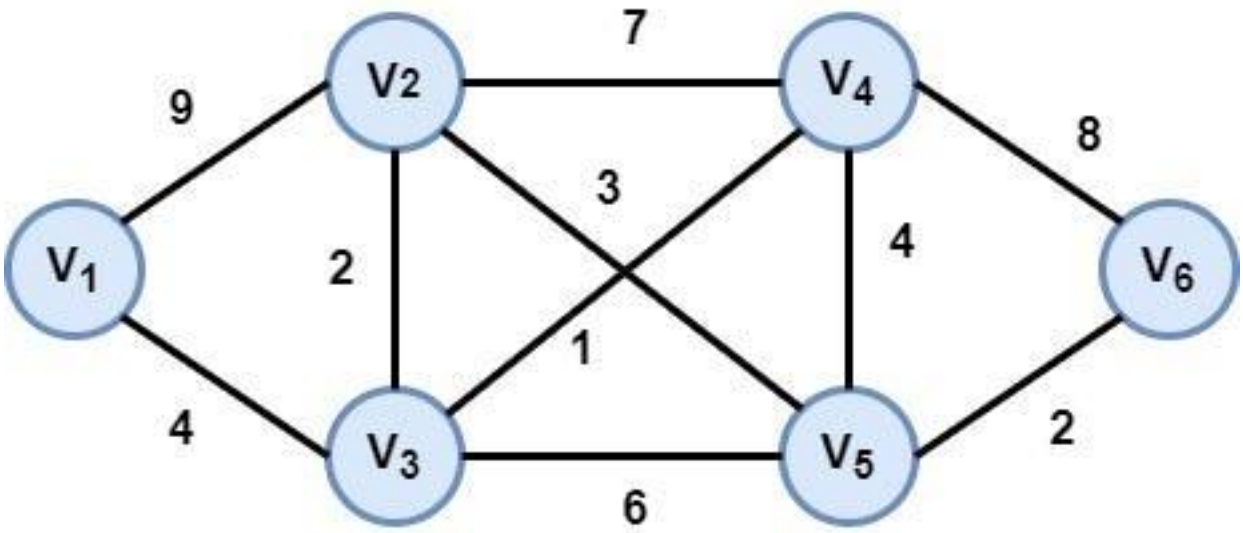
Pseudocode

The UCS algorithm takes as inputs the graph along with the starting and the destination nodes and returns the optimum path between these nodes if exists. Like the Greedy algorithm, the UCS algorithm uses two lists, the *opened *and the *closed *list. The first list contains the nodes that are possible to be selected and the closed list contained the nodes that have already been selected. Firstly, the first

node (initial state) is appended to the opened list (initialization phase). In each step, the node (selected node) with the smallest distance value is removed from the opened list and is appended to the closed list. For each child of the selected node, the algorithm calculates the distance from the first node to this. If the child does not exist in both lists and is in the opened list but with a bigger distance value from the initial node, then the child is appended in the opened list in the position of the corresponding node. The whole process is terminated when a solution is found, or the opened list be empty. The latter situation means that there is not a possible solution to the related problem. The pseudocode of the UCS algorithm is the following:

```
function UCS (Graph, start, target):
Add the starting node to the opened list. The node has
has zero distance value from itself
while True:
if opened is empty:
break # No solution found
selecte_node = remove from opened list, the node with
the minimun distance value
if selected_node == target:
calculate path
return path
add selected_node to closed list
new_nodes = get the children of selected_node
if the selected node has children:
for each child in children:
calculate the distance value of child
if child not in closed and opened lists:
child.parent = selected_node
add the child to opened list
else if child in opened list:
if the distance value of child is lower than
the corresponding node in opened list:
child.parent = selected_node
add the child to opened list
```

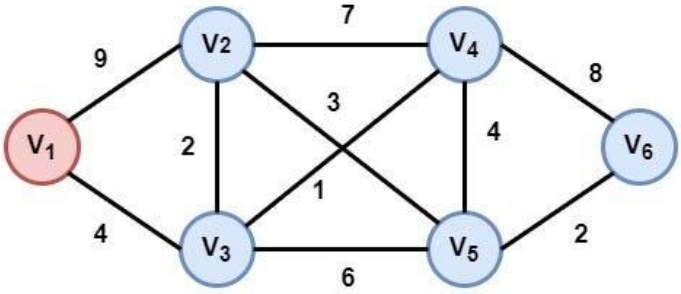
we have a graph representing a roadmap of a country, in which there are six cities (vertices — nodes) and a couple of edges connecting these cities. The graph has the following form:



Graph that models the roadmap of the problem
Our target is to go from the city (node) V1 to V6 following the path with the smallest cost (shortest path). Let's execute the UCS algorithm:

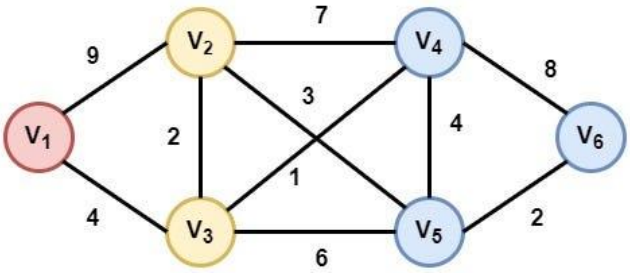
Step 1: Initialization

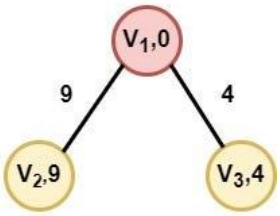
The first node V1 (initial state) of the graph is appended to the opened list. The distance of this node from itself is zero.



Pace	Opened	Closed
0	{(V_1,0)}	{-}

Step 2:

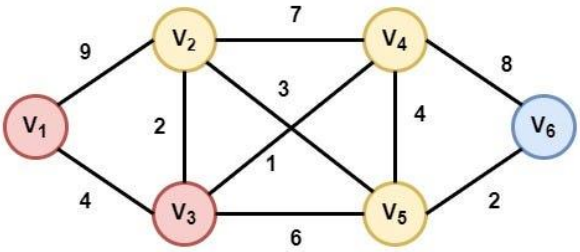


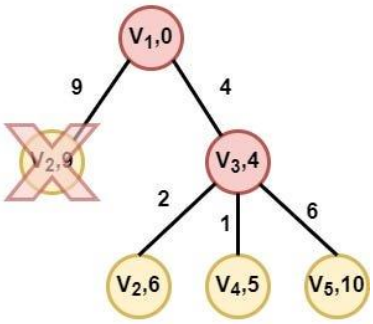


Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}

The V1 is selected as it is the only node in the opened list. Its children V2 and V3 are appended in the opened list after the distance calculation from node V1.

Step 3:

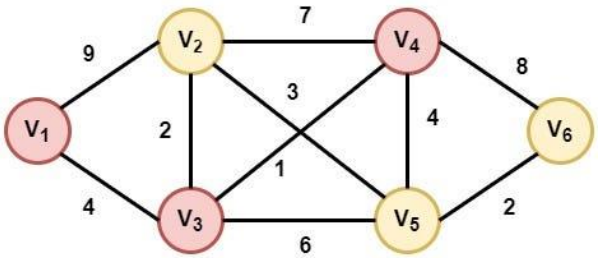


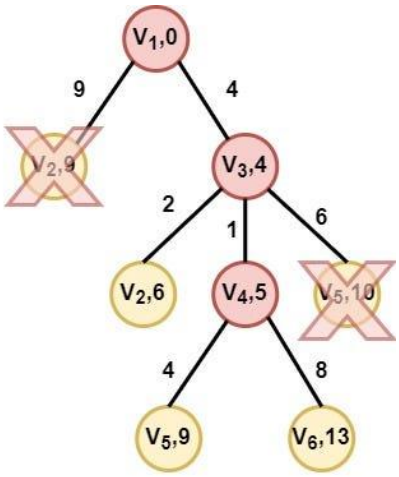


Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}

Node V3 is selected as it has the smallest distance value. As we can see, extending the node V3 we find the node V2 with a smaller distance value. So we replace node (V2,9) with the new node (V2,6).

Step 4: Node V4 is selected

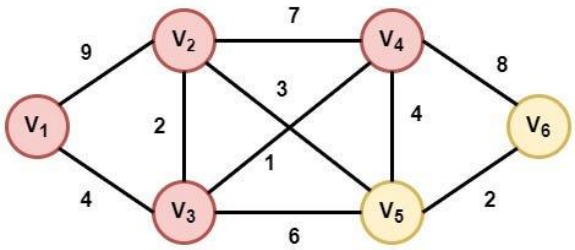


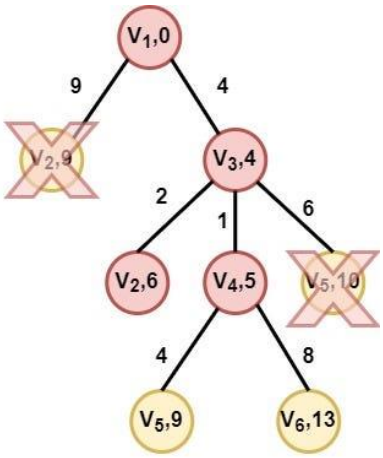


Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}

4 is selected as it has the smallest distance value. In this step, we find a better distance value for node V5, so we replace the node (V5, 10) with node (V5, 9).

Step 5: Node V2 is selected

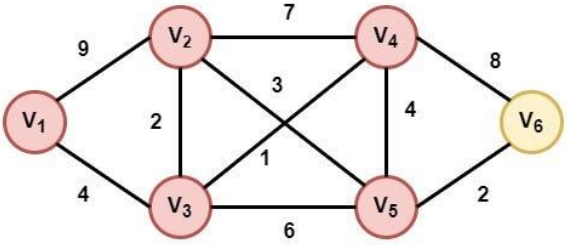


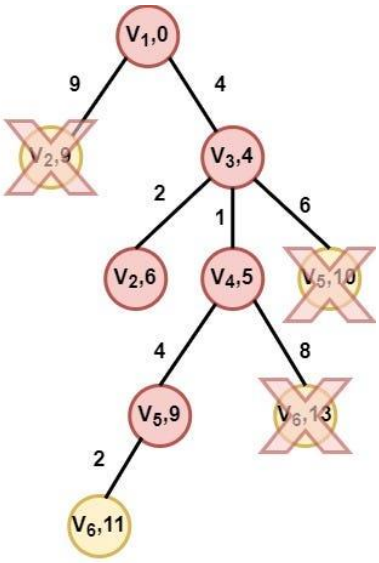


Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}
4	{(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6)}

Node V2 is selected as it has the smallest distance value. However, none of its children is appended in the opened list, as nodes V3 and V4 are already inserted in the closed list and the algorithm doesn't find a better distance value for node V5.

Step 6: Node V5 is selected

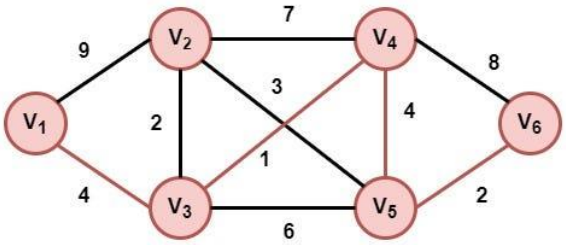


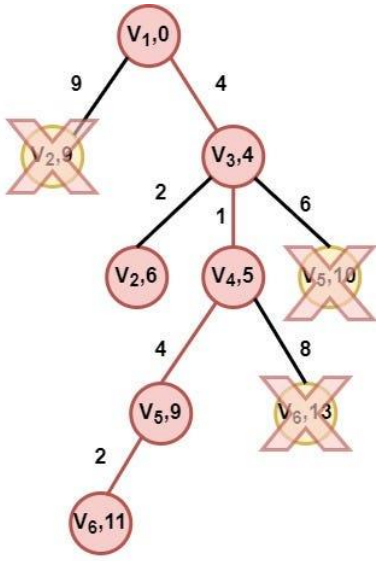


Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}
4	{(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6)}
5	{(V_6,11)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}

Node V2 is selected as it has the smallest distance value. A better path to node V6 is found in this step. So, we replace the old node (V6, 13) with node (V6, 11)

Step 7: Node V6 is selected





Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}
4	{(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6)}
5	{(V_6,11)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}
6	{-}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}

Path: V_1 - V_3 - V_4 - V_5 - V_6

Total Cost: 11

Node V6 (target node) is selected.

So the algorithm returns the path from node V1 to node V6 with cost 11, which constitutes the best solution. python implementation:

```python
class Node:
    """
        This class used to represent each Vertex in the graph
        ...
        Attributes
        ----------
        value : str
            Represent the value of the node
        heuristic_value : int
            Coresponds to the distance from the start node to current node. Default value is inf (infinity)
        neighbors : list
            A list with the nodes the current node is connected
        parent : Node
            Represents the parent-node of the current node. Default value is None

        ...
        Methods
        -------
        has_neighbors(self) -> Boolean
            Check if the current node is connected with other nodes (return True). Otherwise return False
        number_of_neighbors(self) -> int
            Calculate and return the number the of the neighbors
        add_neighboor(self, neighboor) -> None
            Add a new neighbor in the list of neighbors
        extend_node(self) -> list
            return a list of nodes with which the current node is connected
        __eq__(self, other) -> Boolean
            Determines if two nodes are equal or not, checking their values
        __str__(self) -> str
            Prints the node data
    """

    def __init__(self, value, neighbors=None):
        self.value = value
        self.heuristic_value = inf
        if neighbors is None:
            self.neighbors = []
        else:
            self.neighbors = neighbors
        self.parent = None
```

We compare the nodes according to their heuristic value (distance from the initial node). So, we implement the magic of the dunder method gt(), which defines the way the nodes are compared to each other.

```
def __gt__(self, other):
    """
        Define which node, between current node and other node, has the greater value.
        First examine the heuristic value. If this value is the same for both nodes
        the function checks the lexicographic series
        Parameters
        ----------
            other: Node:
                Represent the other node with which the current node is compared
        Returns
        -------
            Boolean
    """
    if isinstance(other, Node):
        if self.heuristic_value > other.heuristic_value:
            return True
        if self.heuristic_value < other.heuristic_value:
            return False
        return self.value > other.value
```

The main class of the algorithm is the *UCS *class that represents the UCS algorithm. This class has a couple of attributes, such as the *graph *which represents the model of the problem, the *opened *and *closed *lists, etc. An overview of the class is the following:

```
class UCS:
  """
    This class used to represent the Greedy algorithm
    ...
    Attributes
    ----------
    graph : Graph
      Represent the graph (search space of the problem)
    start : str
      Represent the starting point
    target : str
      Represent the destination (target) node
    opened : list
      Represent the list with the available nodes in the search process
    closed : list
      Represent the list with the closed (visited) nodes
    number_of_steps : int
      Keep the number of steps of the algorithm
    ...
    Methods
    -------
    calculate_distance(self, parent, child) -> int
      Calculate the distance from the starting node to the child node
    insert_to_list(self, list_category, node) -> None
      Insert a new node either ot opened or to closed list according to list_category parameter
    remove_from_opened(self) -> Node
      Remove from the opened list the node with the smallest heuristic value
    opened_is_empty(self) -> Boolean
      Check if the opened list is empty or not
    get_old_node(self, node_value) -> Node
      Return the node from the opened list in case of a new node with the same value
    calculate_path(self, target_node) -> list
      Calculate and return the path from the stat node to target node
    search(self)
        Implements the core of algorithm. This method searches, in the search space of the problem, a solution
    """

  def __init__(self, graph, start_position, target):
    self.graph = graph
    self.start = graph.find_node(start_position)
    self.target = graph.find_node(target)
    self.opened = []
    self.closed = []
    self.number_of_steps = 0
```

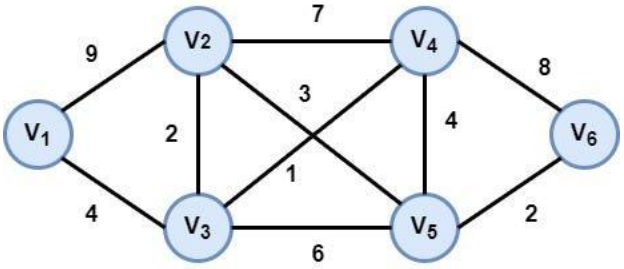
To calculate the distance from the initial node to the current node we use the following method: gt() in python

```
def calculate_distance(self, parent, child):
    """
      Calculate and return the distance from the start to child node. If the heuristic value has already calculated
      and is smaller than the new value, the method return theold value. Otherwise the method return the new value
      and note the parent as the parent node of child
      Parameters
      ----------
      parent : Node
        Represent the parent node
      child : Node
        Represent the child node
      ...
      Return
      ------
        int
    """
    for neighbor in parent.neighbors:
      if neighbor[0] == child:
        distance = parent.heuristic_value + neighbor[1]
        if distance < child.heuristic_value:
          child.parent = parent
          return distance

        return child.heuristic_value
```

Finally, the core algorithm is the *search() *method that represents the whole algorithmic procedure.

```
def search(self):
    """
      Is the main algorithm. Search for a solution in the solution space of the problem
      Stops if the opened list is empty, so no solution found or if it find a solution.
      ...
      Return
      ------
        list
    """
    # The heuristic value of the starting node is zero
    self.start.heuristic_value = 0
    # Add the starting point to opened list
    self.opened.append(self.start)

    while True:
      self.number_of_steps += 1

      if self.opened_is_empty():
        print(f"No Solution Found after {self.number_of_steps} steps!!!")
        break

      selected_node = self.remove_from_opened()
      # print(f"Selected Node {selected_node}")
      # check if the selected_node is the solution
      if selected_node == self.target:
        path = self.calculate_path(selected_node)
        return path, self.number_of_steps

      # extend the node
      new_nodes = selected_node.extend_node()

      # add the extended nodes in the list opened
      if len(new_nodes) > 0:
        for new_node in new_nodes:

          new_node.heuristic_value = self.calculate_distance(selected_node, new_node)
          if new_node not in self.closed and new_node not in self.opened:
            self.insert_to_list("open", new_node)
          elif new_node in self.opened and new_node.parent != selected_node:
            old_node = self.get_old_node(new_node.value)
            if new_node.heuristic_value < old_node.heuristic_value:
              new_node.parent = selected_node
              self.insert_to_opened(new_node)
```

Example:that the search space of our example is the following:



Model above graph as:

```python
def run():
    # Create graph
    graph = Graph()
    # Add vertices
    graph.add_node(Node('V1'))
    graph.add_node(Node('V2'))
    graph.add_node(Node('V3'))
    graph.add_node(Node('V4'))
    graph.add_node(Node('V5'))
    graph.add_node(Node('V6'))

    # Add edges
    graph.add_edge('V1', 'V2', 9)
    graph.add_edge('V1', 'V3', 4)
    graph.add_edge('V2', 'V3', 2)
    graph.add_edge('V2', 'V4', 7)
    graph.add_edge('V2', 'V5', 3)
    graph.add_edge('V3', 'V4', 1)
    graph.add_edge('V3', 'V5', 6)
    graph.add_edge('V4', 'V5', 4)
    graph.add_edge('V4', 'V6', 8)
    graph.add_edge('V5', 'V6', 2)

    # Execute the algorithm
    alg = UCS(graph, "V1", "V6")
    path, path_length = alg.search()
    print(" -> ".join(path))
    print(f"Length of the path: {path_length}")

if __name__ == '__main__':
  run()

# V1 -> V3 -> V4 -> V5 -> V6
# Length of the path: 6
```

As we can see, we have the same results. Remember, that the UCS algorithm always returns the optimum solution, so this path is the shortest path as we also have calculated

Advantages:

The Uniform Cost Search (UCS) algorithm has several advantages, including:

1. Completeness: The UCS algorithm is guaranteed to find a solution if one exists, as long as the cost of the path to the goal is finite.

2. Optimality: The UCS algorithm finds the optimal solution (i.e., the path with the lowest cost) if the path cost is non-negative.

3. Efficiency: The UCS algorithm explores the search space in order of increasing cost, which means it is more efficient than uninformed search algorithms such as breadth-first search or depth-first search.

4. Flexibility: The UCS algorithm can be used with a variety of cost functions, which makes it adaptable to different types of problems.

5. Applicability: The UCS algorithm can be used in a wide range of applications, including route planning, robotics, game AI, and more.

Limitations:

The Uniform Cost Search (UCS) algorithm also has some limitations and drawbacks, including:

1. Memory requirements: The UCS algorithm can require a lot of memory if the search space is large, as it needs to store all the visited nodes and their costs in the priority queue.

2. Time complexity: The UCS algorithm can be slow if the cost of the optimal path is high, as it needs to explore all the nodes with a lower cost before finding the optimal solution.

3. Uninformed search: The UCS algorithm is an uninformed search algorithm, which means it does not use any domain-specific knowledge to guide the search. This can lead to exploring unnecessary paths and slowing down the search process.

4. Non-admissibility: The UCS algorithm can produce suboptimal solutions if the cost function is not admissible (i.e., it overestimates the true cost of the path).

5. Incomplete in infinite graphs: The UCS algorithm is incomplete if the search space is infinite and the cost function can produce negative costs or cycles, as it may get stuck in an infinite loop.

Real-world Applications:

The Uniform Cost Search (UCS) algorithm has a wide range of real-world applications in various domains, including:

1. Route planning: UCS is commonly used in navigation systems to find the shortest or least costly path between two locations.

2. Game AI: UCS can be used in-game AI to find the optimal path or strategy for non-player characters or game agents.

3. Robotics: UCS can be used in robotics to plan the optimal path for a robot to move from one location to another while avoiding obstacles and minimizing energy consumption.

4. Natural language processing: UCS can be used in natural language processing to find the shortest or least costly path between words or phrases in a text.

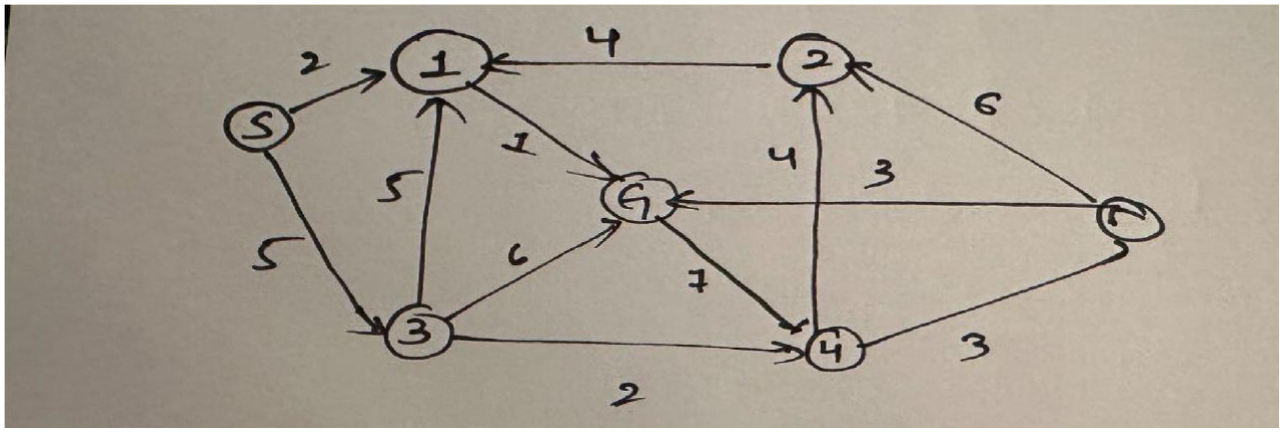
5. Supply chain optimization: UCS can be used in supply chain optimization to find the optimal route for the transportation of goods, minimizing transportation costs and time.

6. Network routing: UCS can be used in network routing to find the most efficient path for data packets to travel from one node to another.

7. Medical diagnosis: UCS can be used in medical diagnosis to find the most probable diagnosis or treatment plan based on symptoms and medical history.

Example:

INPUT:



S is the starting state G is
the goal state

Output :
Minimum cost from S to G is =3

Output :
Minimum cost from S to G is =3

Conclusion:

In conclusion, the Uniform Cost Search (UCS) algorithm is a popular search algorithm used in AI to find the optimal path in a graph or a tree with weighted edges. The algorithm expands the nodes with the lowest cost to reach them, effectively searching the paths with the least total cost.

In the context of adding points in a Crash Course game, the UCS algorithm can be used to find the optimal path that passes through each point in the game while minimizing the total distance traveled. By assigning a weight to each edge in the game board based on the distance between the points, the UCS algorithm can find the path that visits each point in the shortest possible time.

Overall, the UCS algorithm is a powerful tool for solving optimization problems, and its applications are not limited to AI or game development. It can be used in many fields where finding the optimal path is essential, such as logistics, transportation, and urban planning.

References:

Artificial Intelligence Foundations: A Case Study Approach - https://www.coursera.org/lecture/ai-foundations-for-everyone/uniform-cost-search-acase-study-8x6aw

Wikipedia: Uniform-cost search https://en.wikipedia.org/wiki/Uniform-cost_search

Search Algorithms - Uniform Cost Search - https://www.tutorialspoint.com/search_algorithms/uniform_cost_search_algorithm.ht m

GeeksforGeeks: Uniform Cost Search (UCS) - https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

Uniform-Cost Search - Artificial Intelligence https://www.javatpoint.com/uniform-cost-search