01 Programming Assignment

Abdul Khader, Syed

10 October 2022

```
from collections import deque

class Graph:
    # example of adjacency list (or rather map) containing node: (neigbhor node, weight of
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    # initialises object with the given node: neighbor list
    def __init__(self, adjacency_list: dict):
        self.adjacency_list = adjacency_list

    # gets the neighbor of a given node v from the list
    def get_neighbors(self, v: str):
        """
        Returns the List of nodes connected to the node v.
        """
        try:
            return self.adjacency_list[v]
        except KeyError:
            return []

    # heuristic function with values for all nodes
    def h(self, n):
        """
        Heuristic Function, denoting the cost value from n to the goal state.
        """
        H_dist = {
```

```
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,
        }
        return H_dist[n]

    def back_track(self,start_node, goal_state, back_track_dict):
        """
        Returns the Path from goal_state to start_node
        It traces the dictionary, starting from goal_state
        and check it's parent, than again check parent's parent
        and so on, till the parent node is initial node while adding
        each parent node to a string.

        Return the reverse of string to get path.
        """
        curr_node = goal_state
        path = goal_state
        while start_node != curr_node:
            curr_node = back_track_dict[curr_node]
            path += curr_node

        return path[::-1]

    def a_star(self, start_node, stop_node):
        # store visited nodes with uninspected neighbors in open_list, starting with the s
        # store visited nodes with inspected neighbors in closed_list
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        # It is the priority queue to select the next node for exploration
        # node: (heuristic_cost to each here(h), cost to go to node(c), total_cost (h+c))
        g = {}

        g[start_node] = self.h(start_node), 0, self.h(start_node)+0
```

```
# Parents: dictionary where key is a node and it's value
# is the parent of that node.
parents = {}
parents[start_node] = start_node
current_node = start_node

while len(g) > 0:

    # if current node is the goal state, return the path.
    if current_node == stop_node:
        return self.back_track(start_node, current_node, parents)

    closed_list.add(current_node)
    # Traversing the Graph
    for (child, cost) in self.get_neighbors(current_node):
        # if node is not in frontier and explored sets
        if (child not in g) and (child not in closed_list):
            # Add the new node to frontier as key, and it's value
            # as a tuple of 3 values.
            g[child] = (self.h(child) , cost, self.h(child) + cost)
            parents[child] = current_node

        # if the child is present in frontier
        elif child in g:
            # if the new path to the node is cheaper than already present
            # in frontier, replace the new cost
            if g[child][-1] > g[current_node][1] + cost + self.h(child):
                g[child] = (g[current_node][1] + cost,
                            self.h(child),
                            g[current_node][1] + self.h(child) + cost)
                # Also replace the parent of node
                parents[child] = current_node

    # Remove from frontier the explored node.
    del g[current_node]
    # Select new node to be explored based on minimum cost value.
    current_node = min(g.items(), key=lambda item: item[1][-1])[0]


return None
```

```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}

# Driver code for the given graph
graph = Graph(adjacency_list)
graph.a_star('A', 'D')
```

```
'ABD'
```