

Assignment 1 – AI

Pan Eyal – 208722058, Roee Weiss Lipshitz – 208347039

Heuristic Function:

We chose our heuristic function to be the sum of weights in the MST containing only the state's current vertex and the vertices with people that needs saving, as it was defined in class.

We ignore the vertex forms (safe/brittle/broken), as it simplifies the problem that we are trying to solve. Thus, providing an optimistic heuristic. It returns a value greater or equal to 0. Since, the sum of edge weights always greater than 0, unless there are no edges, and then returns 0. If there are no edges, we arrived at a goal, because then the only vertex in the graph is our state's current vertex, and there are no remaining vertices with people to save.

It follows that our heuristic is admissible.

```

1 import copy as c
2 from typing import Callable, Dict, List
3
4 import graph as g
5 import priority_queue as pq
6 import state as s
7 import vertex as v
8
9
10 def generate_sequence(get_edge_weight: Callable, state_wrapper
11 : s.StateWrapper) -> List[v.Vertex]:
12     if state_wrapper.parent_wrapper is None:
13         return []
14     edge_weight = get_edge_weight(state_wrapper.parent_wrapper.
15     state.current_vertex, state_wrapper.state.current_vertex)
16     current_move = []
17     for i in range(edge_weight):
18         current_move.append(state_wrapper.state.current_vertex)
19     current_sequence = generate_sequence(get_edge_weight,
20     state_wrapper.parent_wrapper)
21     current_sequence.extend(current_move)
22     return current_sequence
23
24
25
26 def validate_back_traversing(state_wrapper: s.StateWrapper) ->
27     bool:
28     current_vertex = state_wrapper.state.current_vertex
29     parent_wrapper = state_wrapper.parent_wrapper
30     if parent_wrapper is None:
31         return True
32     parent_unsaved_vertices = parent_wrapper.state.
33     get_unsaved_vertices()
34     grand_parent_wrapper = parent_wrapper.parent_wrapper
35     if grand_parent_wrapper is None:
36         return True
37     grand_parent_vertex = grand_parent_wrapper.state.
38     current_vertex
39     grand_parent_unsaved_vertices = grand_parent_wrapper.state.
40     get_unsaved_vertices()
41
42     # make sure that either this is not a back traversal or it
43     # was necessary for saving people

```

```

41     return current_vertex != grand_parent_vertex or
42     parent_unsaved_vertices != grand_parent_unsaved_vertices
43
44 class Agent:
45
46     def __init__(self, starting_vertex: v.Vertex,
47                  vertices_saved: Dict[v.Vertex, bool],
48                  vertices_broken: Dict[v.Vertex, bool], h:
49                  Callable, expansion_limit: int, time_limit: int, T: float):
50         self.state = s.State(starting_vertex,
51                           vertices_saved, vertices_broken)
52         self.h: Callable = h
53         self.score: int = 0
54         self.terminated: bool = False
55         self.act_sequence: List[v.Vertex] = []
56         self.num_of_expansions: int = 0
57         self.num_of_movements: int = 0
58         self.time_passed: int = 0
59         self.expansion_limit: int = expansion_limit
60         self.time_limit: int = time_limit
61         self.T: float = T
62
63     def _search_fringe(self, world: g.Graph, fringe: pq.
64                       PriorityQueue) -> int:
65         counter = 0
66         state_wrapper_of_self_state = s.StateWrapper(c.copy(
67             self.state), None, 0)
68         fringe.insert(state_wrapper_of_self_state)
69         while not fringe.is_empty():
70             current_state_wrapper = fringe.pop()
71             current_vertex = current_state_wrapper.state.
72             current_vertex
73             acc_weight = current_state_wrapper.acc_weight
74             current_state_wrapper.state.save_current_vertex()
75             current_state_wrapper.state.
76             break_current_vertex_if_brittle()
77             if counter == self.expansion_limit or
78             current_state_wrapper.state.goal_test():
79                 self.act_sequence = generate_sequence(world.
80                 get_edge_weight, current_state_wrapper)
81                 break
82             counter += 1
83             for neighbor, edge_weight in world.expand(
84                 current_vertex):
85                 if not current_state_wrapper.state.
86                 is_vertex_broken(neighbor):
87                     neighbor_state = s.State(neighbor, c.copy(

```

```

76 current_state_wrapper.state.vertices_saved),
77                                     c.copy(
78                                         current_state_wrapper.state.vertices_broken))
79                                         neighbor_state_wrapper = s.StateWrapper(
80                                         neighbor_state, current_state_wrapper,
81                                         acc_weight + edge_weight)
82                                         if validate_back_traversing(
83                                         neighbor_state_wrapper):
84                                         fringe.insert(neighbor_state_wrapper)
85                                         self.num_of_expansions += counter
86                                         return counter
87
88     def _search(self, world: g.Graph) -> int:
89         pass
90
91     def _expand(self, world: g.Graph) -> None:
92         expansions_in_search = self._search(world)
93         self.terminated = len(self.act_sequence) == 0
94         self.time_passed += self.T * expansions_in_search
95
96     def _enter_dest_vertex(self, vertex: v.Vertex) -> None:
97         if vertex.people_to_rescue > 0:
98             print("Saving: " + str(vertex))
99             self.score += vertex.people_to_rescue
100            vertex.people_to_rescue = 0
101            if vertex.form == v.Form.brittle:
102                print("Breaking: " + str(vertex))
103                vertex.form = v.Form.broken
104
105        def _move(self) -> None:
106            print("Current vertex: " + str(self.state.
107 current_vertex))
108            print("Current sequence: " + v.vertex_list_to_string(
109 self.act_sequence))
110            self.num_of_movements += 1
111            next_vertex = self.act_sequence[0]
112            print("Moving to: " + str(next_vertex))
113            if next_vertex != self.state.current_vertex:
114                self._enter_dest_vertex(self.state.current_vertex)
115                self.state.current_vertex = next_vertex
116                self.time_passed += 1
117                self.act_sequence = self.act_sequence[1:]
118                if len(self.act_sequence) == 0:
119                    self._enter_dest_vertex(self.state.current_vertex)
120
121        def _no_op(self) -> None:
122            print("Current vertex: " + str(self.state.

```

```

117 current_vertex))
118         print("Current sequence: " + v.vertex_list_to_string(
119             self.act_sequence))
120         print("No-Op: Staying in current vertex")
121         self.time_passed += 1
122     def act(self, world: g.Graph) -> None:
123         if not self.terminated:
124             self.state.update_vertices_saved()
125             self.state.update_vertices_broken()
126             if len(self.act_sequence) == 0:
127                 self._expand(world)
128                 print("Search returned sequence: " + v.
129                     vertex_list_to_string(self.act_sequence))
130             if not self.terminated and self.time_passed + 1 <
131                 self.time_limit:
132                 next_vertex = self.act_sequence[0]
133                 if next_vertex != self.state.current_vertex
134                     and next_vertex.form == v.Form.broken:
135                     self.act_sequence = []
136                     self._no_op()
137                     return
138                     self._move()
139             else:
140                 self.terminated = True
141                 print("TERMINATED")
142             else:
143                 print("TERMINATED")
144
145         def __str__(self):
146             agent_str = f"Score: {str(self.score)}\n"
147             agent_str += f"Number of expansions: {str(self.
148                 num_of_expansions)}\n"
149             agent_str += f"Number of movements: {str(self.
150                 num_of_movements)}\n"
151             agent_str += f"Total time passed: {str(self.
152                 time_passed)}\n"
153             agent_str += f"Agent reached goal!!! \U0001f60e\n" if
154                 self.state.goal_test() else "Goal wasn't reached \U0001f61f"
155             return agent_str
156

```

```
157
158 class GreedyAgent(Agent):
159     def __init__(self, starting_vertex: v.Vertex,
160                  vertices_saved: Dict[v.Vertex, bool],
161                  vertices_broken: Dict[v.Vertex, bool],
162                  h: Callable, expansion_limit: int, time_limit
163                  : int, T: float):
164         super().__init__(starting_vertex, vertices_saved,
165                         vertices_broken, h, expansion_limit, time_limit, T)
166
167
168
169 class AStarAgent(Agent):
170
171     def __init__(self, starting_vertex: v.Vertex,
172                  vertices_saved: Dict[v.Vertex, bool],
173                  vertices_broken: Dict[v.Vertex, bool],
174                  h: Callable, expansion_limit: int, time_limit
175                  : int, T: float):
176         super().__init__(starting_vertex, vertices_saved,
177                         vertices_broken, h, expansion_limit, time_limit, T)
178
179     def _search(self, world: g.Graph) -> int:
180         fringe = pq.PriorityQueue(lambda x: self.h(x, world
181 ) + get_g(x))
182         return self._search_fringe(world, fringe)
183
184
185 class RealTimeAStarAgent(Agent):
186
187     def __init__(self, starting_vertex: v.Vertex,
188                  vertices_saved: Dict[v.Vertex, bool],
189                  vertices_broken: Dict[v.Vertex, bool],
190                  h: Callable, expansion_limit: int, time_limit
191                  : int, T: float):
192         super().__init__(starting_vertex, vertices_saved,
193                         vertices_broken, h, expansion_limit, time_limit, T)
194
195     def _search(self, world: g.Graph) -> int:
196         fringe = pq.PriorityQueue(lambda x: self.h(x, world
197 ) + get_g(x))
198         return self._search_fringe(world, fringe)
```

```

1 import itertools
2 import sys
3 from typing import Tuple, List, Dict
4
5 import vertex as v
6
7
8 class Graph(object):
9
10     def __init__(self, graph_dict=None):
11         if graph_dict is None:
12             graph_dict = {}
13         self.graph_dict: Dict[v.Vertex, (v.Vertex, v.Vertex, int)] = graph_dict
14
15     def get_vertices(self) -> List[v.Vertex]:
16         return list(self.graph_dict.keys())
17
18     def get_savable_vertices(self) -> List[v.Vertex]:
19         return list(filter(lambda ve: ve.people_to_rescue > 0,
20                           self.graph_dict.keys()))
21
22     def get brittle_vertices(self) -> List[v.Vertex]:
23         return list(filter(lambda ve: ve.form == v.Form.brittle
24                           , self.graph_dict.keys()))
25
26     def get_edges(self):
27         return self.generate_edges()
28
29     def get_vertex(self, id):
30         vertex_to_ret = None
31         for vertex in self.get_vertices():
32             if vertex.id == id:
33                 vertex_to_ret = vertex
34         return vertex_to_ret
35
36     def vertices_ids(self) -> List[int]:
37         id_list = []
38         for vertex in self.get_vertices():
39             id_list.append(vertex.id)
40         return id_list
41
42     def expand(self, vertex: v.Vertex) -> List[Tuple[v.Vertex, int]]:
43         return self.graph_dict[vertex]
44
45     def expand_just_vertices(self, vertex: v.Vertex) -> map:
46         return map(lambda neighbor_tup: neighbor_tup[0], self.

```

```

44 expand(vertex))
45
46     def get_edge_weight(self, vertex1: v.Vertex, vertex2: v.
47         Vertex) -> int:
48         neighbors = self.expand(vertex1)
49         for neighbor in neighbors:
50             if neighbor[0].id == vertex2.id:
51                 return neighbor[1]
52
53     def is_vertex_exists(self, vertex: v.Vertex) -> bool:
54         return vertex.id in self.vertices_ids()
55
56     def edge_exists(self, vertex1, vertex2):
57         neighbor_list = self.expand(vertex1)
58         for neighbor_tup in neighbor_list:
59             if vertex2 == neighbor_tup[0]:
60                 return True
61         return False
62
63     def get_sum_weights(self):
64         sum_weight_mst = 0
65         for edge in self.get_edges():
66             sum_weight_mst += edge[2]
67         return sum_weight_mst / 2
68
69     def add_vertex(self, vertex: v.Vertex) -> None:
70         if not self.is_vertex_exists(vertex):
71             self.graph_dict[vertex] = []
72
73     def add_edge(self, vertex1, vertex2, weight) -> None:
74         if vertex2 not in self.expand_just_vertices(vertex1)
75             and vertex1 not in self.expand_just_vertices(vertex2):
76             self.graph_dict[vertex1].append((vertex2, weight))
77             self.graph_dict[vertex2].append((vertex1, weight))
78
79     def delete_edge(self, vertex1, vertex2):
80         index_of_vertex2_in_vertex_1 = 0
81         index_of_vertex1_in_vertex_2 = 0
82         for neighbor_tup in self.expand(vertex1):
83             if neighbor_tup[0] == vertex2:
84                 break
85             index_of_vertex2_in_vertex_1 += 1
86         for neighbor_tup in self.expand(vertex2):
87             if neighbor_tup[0] == vertex1:
88                 break
89             index_of_vertex1_in_vertex_2 += 1
90         del self.graph_dict[vertex1][
91             index_of_vertex2_in_vertex_1]

```

```

89         del self.graph_dict[vertex2][
90             index_of_vertex1_in_vertex_2]
91
92     def add_or_replace_edge(self, vertex1, vertex2, weight):
93         if vertex2 not in self.expand_just_vertices(vertex1)
94             and vertex1 not in self.expand_just_vertices(vertex2):
95                 self.add_edge(vertex1, vertex2, weight)
96             elif weight < self.get_edge_weight(vertex1, vertex2):
97                 self.replace_edge(vertex1, vertex2, weight)
98
99
100    def replace_edge(self, vertex1, vertex2, weight):
101        self.delete_edge(vertex1, vertex2)
102        self.add_edge(vertex1, vertex2, weight)
103
104    def generate_edges(self) -> List[Tuple]:
105        edges = []
106        for vertex in self.graph_dict:
107            for neighbor_tuple in self.graph_dict[vertex]:
108                edges.append((vertex, neighbor_tuple[0],
109                             neighbor_tuple[1]))
110
111        return edges
112
113    def __str__(self):
114        res = "vertices: "
115        for k in self.graph_dict:
116            res += str(k) + ", "
117
118            res = res[:len(res) - 2]
119            res += "\nedges: "
120            for edge in self.generate_edges():
121                res += "(" + str(edge[0].id) + ", " + str(edge[1].
122 id) + ", " + str(edge[2]) + ")", "
123            res = res[:len(res) - 2]
124
125        return res
126
127    def copy_graph(self):
128        new_graph = Graph()
129        for vertex in self.get_vertices():
130            new_graph.graph_dict[vertex] = list(self.expand(
131                vertex))
132
133        return new_graph
134
135
136    def remove_unessential_vertices(self,
137        unessential_vertices_array):
138        for vertex in unessential_vertices_array:
139            self.delete_all_occurrences(vertex)
140
141    def delete_all_occurrences(self, vertex):

```

```

131         for u in self.graph_dict.keys():
132             neighbors_tuples = self.graph_dict[u]
133             for tup in neighbors_tuples:
134                 if tup[0] == vertex:
135                     neighbors_tuples.remove(tup)
136             self.graph_dict.pop(vertex)
137
138     def zip_edges(self):
139         for vertex in self.get_vertices():
140             min_edges = {}
141             neighbors_tuples = self.expand(vertex)
142             for neighbor_tup in neighbors_tuples:
143                 if min_edges.get(neighbor_tup[0]) is not None:
144                     if min_edges[neighbor_tup[0]] >
145                         neighbor_tup[1]:
146                         min_edges[neighbor_tup[0]] =
147                             neighbor_tup[1]
148                     else:
149                         min_edges[neighbor_tup[0]] = neighbor_tup[
150                             1]
151                         self.graph_dict[vertex] = [(key, val) for key, val
152                             in min_edges.items()]
153
154     def connect_all_neighbors(self, vertex):
155         neighbor_tuples = self.expand(vertex)
156         for neighbor_tuple1, neighbor_tuple2 in itertools.
157             combinations(neighbor_tuples, 2):
158             both_edges_weight = neighbor_tuple1[1] +
159             neighbor_tuple2[1]
160             self.add_or_replace_edge(neighbor_tuple1[0],
161             neighbor_tuple2[0], both_edges_weight)
162
163     def get_lowest_cost_edge_between_sets(self, in_set,
164         out_set):
165         min_edge = None
166         min_weight = sys.maxsize
167         for in_vertex, out_vertex in itertools.product(in_set
168             , out_set):
169             if self.edge_exists(in_vertex, out_vertex) and
170                 min_weight >= self.get_edge_weight(in_vertex, out_vertex):
171                     current_edge_weight = self.get_edge_weight(
172                         in_vertex, out_vertex)
173                     min_weight = current_edge_weight
174                     min_edge = (in_vertex, out_vertex, min_weight)
175
176     def MST(self):
177         mst = Graph()

```

```
168     in_set = {self.get_vertices()[0]}
169     out_set = set(self.get_vertices()[1:])
170     edge_set = set()
171     while len(out_set) > 0:
172         edge = self.get_lowest_cost_edge_between_sets(
173             in_set, out_set)
174         edge_set.add(edge)
175         in_set.add(edge[1])
176         out_set.remove(edge[1])
177         for vertex in in_set:
178             mst.add_vertex(vertex)
179         for edge in edge_set:
180             mst.add_edge(*edge)
181     return mst
182
183 def zip_graph(original_graph: Graph, essential_vertices):
184     essential_graph = original_graph.copy_graph()
185     for vertex in essential_graph.get_vertices():
186         if vertex not in essential_vertices:
187             essential_graph.connect_all_neighbors(vertex)
188     essential_graph.remove_unessential_vertices(
189         list(filter(lambda u: u not in essential_vertices,
190             essential_graph.get_vertices())))
191     essential_graph.zip_edges()
192     return essential_graph
```

```

1 import copy as cp
2 from typing import Union, List
3
4 import vertex as v
5
6
7 class State:
8
9     def __init__(self, current_vertex: v.Vertex, vertices_saved
10        : dict, vertices_broken: dict):
11         self.current_vertex = current_vertex
12         self.vertices_saved = cp.copy(vertices_saved)
13         self.vertices_broken = cp.copy(vertices_broken)
14
15     def __str__(self):
16         s = "Current vertex: " + str(self.current_vertex) + "\n"
17         {
18             for vertex, saved in self.vertices_saved.items():
19                 s += vertex.id + ": " + ("saved" if saved else "not
19 saved") + "\n"
20             for vertex, broken in self.vertices_broken.items():
21                 s += vertex.id + ": " + ("broken" if broken else "
21 intact") + "\n"
22         return s + "}"
23
24     def save_current_vertex(self) -> None:
25         if self.current_vertex in self.vertices_saved:
26             self.vertices_saved[self.current_vertex] = True
27
28     def break_current_vertex_if_brittle(self) -> None:
29         if self.current_vertex in self.vertices_broken:
30             self.vertices_broken[self.current_vertex] = True
31
32     def get_unsaved_vertices(self) -> List[v.Vertex]:
33         unsaved = []
34         for vertex, saved in self.vertices_saved.items():
35             if not saved:
36                 unsaved.append(vertex)
37         return unsaved
38
39     def get_intact_vertices(self) -> List[v.Vertex]:
40         intact = []
41         for vertex, broken in self.vertices_broken.items():
42             if not broken:
43                 intact.append(vertex)
44         return intact
45
46     def num_of_vertices_to_save(self) -> int:

```

```
45         return len(self.get_unsaved_vertices())
46
47     def goal_test(self) -> bool:
48         return self.num_of_vertices_to_save() == 0
49
50     def update_vertices_broken(self) -> None:
51         for vertex in self.vertices_broken:
52             if vertex.form == v.Form.broken:
53                 self.vertices_broken[vertex] = True
54             else:
55                 self.vertices_broken[vertex] = False
56
57     def update_vertices_saved(self) -> None:
58         for vertex in self.vertices_saved:
59             if vertex.people_to_rescue == 0:
60                 self.vertices_saved[vertex] = True
61             else:
62                 self.vertices_saved[vertex] = False
63
64     def does_current_vertex_need_saving(self):
65         if self.current_vertex in self.vertices_saved:
66             return not self.vertices_saved[self.current_vertex]
67         return False
68
69     def is_vertex_broken(self, vertex):
70         if vertex in self.vertices_broken:
71             return self.vertices_broken[vertex]
72
73
74 class StateWrapper(object):
75
76     def __init__(self, state: State, parent_wrapper: Union['  
StateWrapper', None], acc_weight: int):
77         self.state = state
78         self.parent_wrapper = parent_wrapper
79         self.acc_weight = acc_weight
80
81
```

```
1 from enum import Enum
2 from typing import List
3
4
5 class Form(Enum):
6     stable = 0
7     brittle = 1
8     broken = 2
9
10
11 class Vertex(object):
12     def __init__(self, id: int, people_to_rescue: int, form: Form):
13         self.id = id
14         self.people_to_rescue = people_to_rescue
15         self.form = form
16
17     def __str__(self):
18         return "[V:" + str(self.id) + ", P:" + str(self.
19         people_to_rescue) + ", F:" + str(self.form.name) + "]"
20
21 def vertex_list_to_string(vertices_list: List[Vertex]) -> str:
22     s = "["
23     for vertex in vertices_list:
24         s += f'V{str(vertex.id)}, '
25     last_index_of_comma = s.rfind(",")
26     if last_index_of_comma != -1:
27         s = s[:last_index_of_comma]
28
29     return s + "]"
30
```

```

1 import yaml
2 from typing import List
3
4 import agent as a
5 import graph as g
6 import state as s
7 import vertex as v
8
9 def generate_graph(file_name: str):
10     graph = g.Graph()
11     file = open(file_name)
12     lines = file.readlines()
13     vertices = {}
14
15     for line in lines:
16         line = (line.partition(';'))[0].split(' ')
17         if len(line[0]) <= 1:
18             continue
19         object_type = line[0][1]
20
21         if object_type == 'N':
22             pass # what is it good for?
23
24         elif object_type == 'V':
25             id = int(line[0][2])
26             people_to_rescue = 0
27             brittle = False
28             for property in line[1:]:
29                 if len(property) == 0:
30                     continue
31                 if property[0] == 'P':
32                     people_to_rescue = int(property[1])
33                 elif property[0] == 'B':
34                     brittle = True
35
36             u = v.Vertex(id, people_to_rescue, v.Form.brittle
37             if brittle else v.Form.stable)
38             vertices[u.id] = u
39             graph.add_vertex(u)
40
41         elif object_type == 'E':
42             id = int(line[0][2])
43             v1 = vertices[int(line[1])]
44             v2 = vertices[int(line[2])]
45             edge_weight = int(line[3][1:])
46             graph.add_edge(v1, v2, edge_weight)
47
48     file.close()

```

```

48     return graph
49
50
51 def savable_vertex_list_to_vertices_saved_dict(v_list: List[v.
    Vertex]):
52     v_dict = dict()
53     for vertex in v_list:
54         v_dict[vertex] = False
55     return v_dict
56
57
58 def Breakable_vertex_list_to_vertices_broken_dict(v_list: List[v.
    Vertex]):
59     v_dict = dict()
60     for vertex in v_list:
61         v_dict[vertex] = False
62     return v_dict
63
64
65 def mst_heuristic(state_wrapper: s.StateWrapper, world: g.Graph
) -> int:
66     unsaved_vertices = state_wrapper.state.get_unsaved_vertices
    ()
67     essential_vertices = unsaved_vertices
68     if state_wrapper.state.current_vertex not in
        essential_vertices:
69         essential_vertices.append(state_wrapper.state.
            current_vertex)
70     zipped_graph = g.zip_graph(world, essential_vertices)
71     mst_zipped = zipped_graph.MST()
72     return mst_zipped.get_sum_weights()
73
74
75 def create_agent(agent_type: int, starting_vertex: v.Vertex,
    world: g.Graph, expansion_limit: int, time_limit: int,
    T: float):
76     vertices_saved = savable_vertex_list_to_vertices_saved_dict
    (world.get_savable_vertices())
77     vertices_broken =
        Breakable_vertex_list_to_vertices_broken_dict(world.
            get brittle_vertices())
78     if agent_type == 1:
79         return a.GreedyAgent(starting_vertex, vertices_saved,
        vertices_broken, mst_heuristic, expansion_limit,
        time_limit, T)
80     elif agent_type == 2:
81         return a.AStarAgent(starting_vertex, vertices_saved,
        vertices_broken, mst_heuristic, expansion_limit,

```

```

84                     time_limit, T)
85             elif agent_type == 3:
86                 return a.RealTimeAStarAgent(starting_vertex,
87                                             vertices_saved, vertices_broken, mst_heuristic,
88                                             expansion_limit,
89                                             time_limit, T)
90
91
92     def simulator():
93         print('----Welcome to Hurricane Evacuation Problem----')
94
95         print('loading config.yaml file...')
96         with open("Inputs/config.yaml", "r") as f:
97             config = yaml.safe_load(f)
98
99         world = generate_graph(config['RUN']['INPUT_PATH'])
100        vertices_ids = world.vertices_ids()
101
102        agents = []
103        agents_locs_name = []
104        for agent_name, starting_vertex_index in config['RUN']['AGENTS']:
105            starting_vertex = world.get_vertex(vertices_ids[
106                starting_vertex_index - 1])
107            new_agent = create_agent(config[agent_name]['ID'],
108                                     starting_vertex, world, config[agent_name]['TIME_LIMIT'],
109                                     config['WORLD']['TIME_LIMIT'], config['WORLD']['T'])
110            agents.append(new_agent)
111            agents_locs_name.append((agent_name,
112                                      starting_vertex_index))
113            print('agents: ')
114            print(agents_locs_name)
115            print('world: ')
116            print(world)
117            input('Press Enter to start..')
118
119            while not a.all_agents_terminated(agents):
120                for agent_idx, agent in enumerate(agents):
121                    print(f'\nAGENT {agent_idx+1}: {str(
122                        agents_locs_name[agent_idx][0])}')
123                    agent.act(world)
124
125                print("\n\n\nAGENTS SCORES: ")
126                for agent_idx, agent in enumerate(agents):
127                    print(f'\n#### AGENT {agent_idx + 1}: {str(
128                        agents_locs_name[agent_idx][0])} ####')
129                    print(agent)
130
131                print("World at End: ")

```

```
123     print(world)
124
125
126
127 if __name__ == "__main__":
128     simulator()
129
```

```
1 from typing import List, Union, Callable
2
3 import state as s
4
5
6 class PriorityQueue(object):
7
8     def __init__(self, f: Callable):
9         self.queue: List[s.StateWrapper] = []
10        self.f = f
11
12    def __str__(self):
13        return ' '.join([str(i) for i in self.queue])
14
15    def is_empty(self) -> bool:
16        return len(self.queue) == 0
17
18    def insert(self, data: s.StateWrapper) -> None:
19        self.queue.append(data)
20
21    def pop(self) -> Union[s.StateWrapper, None]:
22        if self.is_empty():
23            return None
24        min_elem = self.queue[0]
25        min_value = self.f(self.queue[0])
26        min_element_amount_to_save = self.queue[0].state.
27            num_of_vertices_to_save()
28
29        for elem in self.queue[1:]:
30            elem_value = self.f(elem)
31            elem_amount_to_save = elem.state.
32            num_of_vertices_to_save()
33
34            if elem_value < min_value \
35                or (elem_value == min_value
36                    and elem_amount_to_save <
37                        min_element_amount_to_save) \
38                or (elem_value == min_value
39                    and elem_amount_to_save ==
40                        min_element_amount_to_save
41                    and elem.state.
42                        does_current_vertex_need_saving()
43                    and not min_elem.state.
44                        does_current_vertex_need_saving()):
45                min_elem = elem
46                min_value = elem_value
47                min_element_amount_to_save =
48                    elem_amount_to_save
```

```
42             self.queue.remove(min_elem)
43         return min_elem
44
45
```

```
1 #N 4      ; number of vertices n in graph (from 1 to n)
2 #V1       ; Vertex 1, nothing of interest
3 #V2 P1 B   ; Vertex 2, initially contains 1 person to be
               rescued, and is brittle
4 #V3 B      ; Vertex 3, has no people and is brittle
5 #V4 P2     ; Vertex 4, initially contains 2 persons to be
               rescued
6
7 #E1 1 2 W1 ; Edge 1 from vertex 1 to vertex 2, weight 1
8 #E2 3 4 W1 ; Edge 2 from vertex 3 to vertex 4, weight 1
9 #E3 2 3 W1 ; Edge 3 from vertex 2 to vertex 3, weight 1
10 #E4 1 3 W4 ; Edge 4 from vertex 1 to vertex 3, weight 4
11 #E5 2 4 W5 ; Edge 5 from vertex 2 to vertex 4, weight 5
```

```
1 #N 5
2 #V1
3 #V2 P2
4 #V3 B
5 #V4 P1
6 #V5 P1
7
8 #E1 1 2 W1
9 #E2 2 3 W1
10 #E3 3 4 W1
11 #E4 1 4 W10
12 #E5 3 5 W1
```

```
1 WORLD:
2   TIME_LIMIT: 10000
3   T: 0.01
4
5 GREEDY_AGENT:
6   ID: 1
7   TIME_LIMIT: 1
8
9 A_STAR_AGENT:
10  ID: 2
11  TIME_LIMIT: 10000
12
13 REAL_TIME_A_STAR_AGENT:
14  ID: 3
15  TIME_LIMIT: 10
16
17 RUN:
18  INPUT_PATH: Inputs/input1.txt
19  # [[Agent Type, Starting Vertex]]
20  AGENTS: [[REAL_TIME_A_STAR_AGENT,1]]
21
```

```
1 C:\Users\paney\anaconda3\python.exe C:\Users\paney\Documents\Studies\AI_Class\Assignment_1\enviroment.py
2 ----Welcome to Hurricane Evacuation Problem----
3 loading config.yaml file...
4 agents:
5 [('GREEDY_AGENT', 1)]
6 world:
7 vertices: [V:1, P:0, F:stable], [V:2, P:1, F:brittle], [V:3, P:0, F:brittle], [V:4, P:2, F:stable]
8 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)
9 Press Enter to start..
10
11 AGENT 1): GREEDY_AGENT
12 Search returned sequence: [V2]
13 Current vertex: [V:1, P:0, F:stable]
14 Current sequence: [V2]
15 Moving to: [V:2, P:1, F:brittle]
16 Saving: [V:2, P:1, F:brittle]
17 Breaking: [V:2, P:0, F:brittle]
18
19 AGENT 1): GREEDY_AGENT
20 Search returned sequence: [V4, V4, V4, V4, V4]
21 Current vertex: [V:2, P:0, F:broken]
22 Current sequence: [V4, V4, V4, V4, V4]
23 Moving to: [V:4, P:2, F:stable]
24
25 AGENT 1): GREEDY_AGENT
26 Current vertex: [V:4, P:2, F:stable]
27 Current sequence: [V4, V4, V4]
28 Moving to: [V:4, P:2, F:stable]
29
30 AGENT 1): GREEDY_AGENT
31 Current vertex: [V:4, P:2, F:stable]
32 Current sequence: [V4, V4, V4]
33 Moving to: [V:4, P:2, F:stable]
34
35 AGENT 1): GREEDY_AGENT
36 Current vertex: [V:4, P:2, F:stable]
37 Current sequence: [V4, V4]
38 Moving to: [V:4, P:2, F:stable]
39
40 AGENT 1): GREEDY_AGENT
41 Current vertex: [V:4, P:2, F:stable]
42 Current sequence: [V4]
43 Moving to: [V:4, P:2, F:stable]
44 Saving: [V:4, P:2, F:stable]
45
```

```
46 AGENT 1): GREEDY_AGENT
47 Search returned sequence: []
48 TERMINATED
49
50
51
52 AGENTS SCORES:
53
54 ##### AGENT 1): GREEDY_AGENT #####
55 Score: 3
56 Number of expansions: 2
57 Number of movements: 6
58 Total time passed: 6.02
59 Agent reached goal!!! []
60
61 World at End:
62 vertices: [V:1, P:0, F:stable], [V:2, P:0, F:broken], [V:3, P:0
, F:brittle], [V:4, P:0, F:stable]
63 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (
3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)
64
65 Process finished with exit code 0
66
```

```
1 C:\Users\paney\anaconda3\python.exe C:\Users\paney\Documents\  
Studies\AI_Class\Assignment_1\enviroment.py  
2 ----Welcome to Hurricane Evacuation Problem----  
3 loading config.yaml file...  
4 agents:  
5 [('A_STAR_AGENT', 1)]  
6 world:  
7 vertices: [V:1, P:0, F:stable], [V:2, P:1, F:brittle], [V:3, P:  
0, F:brittle], [V:4, P:2, F:stable]  
8 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (  
3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)  
9 Press Enter to start..  
10  
11 AGENT 1): A_STAR_AGENT  
12 Search returned sequence: [V2, V3, V4]  
13 Current vertex: [V:1, P:0, F:stable]  
14 Current sequence: [V2, V3, V4]  
15 Moving to: [V:2, P:1, F:brittle]  
16  
17 AGENT 1): A_STAR_AGENT  
18 Current vertex: [V:2, P:1, F:brittle]  
19 Current sequence: [V3, V4]  
20 Moving to: [V:3, P:0, F:brittle]  
21 Saving: [V:2, P:1, F:brittle]  
22 Breaking: [V:2, P:0, F:brittle]  
23  
24 AGENT 1): A_STAR_AGENT  
25 Current vertex: [V:3, P:0, F:brittle]  
26 Current sequence: [V4]  
27 Moving to: [V:4, P:2, F:stable]  
28 Breaking: [V:3, P:0, F:brittle]  
29 Saving: [V:4, P:2, F:stable]  
30  
31 AGENT 1): A_STAR_AGENT  
32 Search returned sequence: []  
33 TERMINATED  
34  
35  
36  
37 AGENTS SCORES:  
38  
39 ##### AGENT 1): A_STAR_AGENT #####  
40 Score: 3  
41 Number of expansions: 3  
42 Number of movements: 3  
43 Total time passed: 3.0300000000000002  
44 Agent reached goal!!! []  
45
```

```
46 World at End:  
47 vertices: [V:1, P:0, F:stable], [V:2, P:0, F:broken], [V:3, P:0  
, F:broken], [V:4, P:0, F:stable]  
48 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (3,  
3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)  
49  
50 Process finished with exit code 0  
51
```

```
1 C:\Users\paney\anaconda3\python.exe C:\Users\paney\Documents\Studies\AI_Class\Assignment_1\enviroment.py
2 ----Welcome to Hurricane Evacuation Problem----
3 loading config.yaml file...
4 agents:
5 [('REAL_TIME_A_STAR_AGENT', 1)]
6 world:
7 vertices: [V:1, P:0, F:stable], [V:2, P:1, F:brittle], [V:3, P:0, F:brittle], [V:4, P:2, F:stable]
8 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)
9 Press Enter to start..
10
11 AGENT 1): REAL_TIME_A_STAR_AGENT
12 Search returned sequence: [V2, V3, V4]
13 Current vertex: [V:1, P:0, F:stable]
14 Current sequence: [V2, V3, V4]
15 Moving to: [V:2, P:1, F:brittle]
16
17 AGENT 1): REAL_TIME_A_STAR_AGENT
18 Current vertex: [V:2, P:1, F:brittle]
19 Current sequence: [V3, V4]
20 Moving to: [V:3, P:0, F:brittle]
21 Saving: [V:2, P:1, F:brittle]
22 Breaking: [V:2, P:0, F:brittle]
23
24 AGENT 1): REAL_TIME_A_STAR_AGENT
25 Current vertex: [V:3, P:0, F:brittle]
26 Current sequence: [V4]
27 Moving to: [V:4, P:2, F:stable]
28 Breaking: [V:3, P:0, F:brittle]
29 Saving: [V:4, P:2, F:stable]
30
31 AGENT 1): REAL_TIME_A_STAR_AGENT
32 Search returned sequence: []
33 TERMINATED
34
35
36
37 AGENTS SCORES:
38
39 ##### AGENT 1): REAL_TIME_A_STAR_AGENT #####
40 Score: 3
41 Number of expansions: 3
42 Number of movements: 3
43 Total time passed: 3.0300000000000002
44 Agent reached goal!!! []
45
```

```
46 World at End:  
47 vertices: [V:1, P:0, F:stable], [V:2, P:0, F:broken], [V:3, P:0  
, F:broken], [V:4, P:0, F:stable]  
48 edges: (1, 2, 1), (1, 3, 4), (2, 1, 1), (2, 3, 1), (2, 4, 5), (3,  
3, 4, 1), (3, 2, 1), (3, 1, 4), (4, 3, 1), (4, 2, 5)  
49  
50 Process finished with exit code 0  
51
```