File - agent.py

```python
import copy as c
from typing import Callable, Dict, List

import graph as g
import priority_queue as pq
import state as s
import vertex as v


def generate_sequence(get_edge_weight: Callable, state_wrapper: s.StateWrapper) -> List[v.Vertex]:
    if state_wrapper.parent_wrapper is None:
        return []
    edge_weight = get_edge_weight(state_wrapper.parent_wrapper.state.current_vertex, state_wrapper.state.current_vertex)
    current_move = []
    for i in range(edge_weight):
        current_move.append(state_wrapper.state.current_vertex)
    current_sequence = generate_sequence(get_edge_weight, state_wrapper.parent_wrapper)
    current_sequence.extend(current_move)
    return current_sequence


def get_g(state_wrapper: s.StateWrapper) -> int:
    return state_wrapper.acc_weight


def validate_back_traversing(state_wrapper: s.StateWrapper) -> bool:
    current_vertex = state_wrapper.state.current_vertex

    parent_wrapper = state_wrapper.parent_wrapper
    if parent_wrapper is None:
        return True
    parent_unsaved_vertices = parent_wrapper.state.get_unsaved_vertices()

    grand_parent_wrapper = parent_wrapper.parent_wrapper
    if grand_parent_wrapper is None:
        return True
    grand_parent_vertex = grand_parent_wrapper.state.current_vertex
    grand_parent_unsaved_vertices = grand_parent_wrapper.state.get_unsaved_vertices()

    # make sure that either this is not a back traversal or it was necessary for saving people
```

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

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

```
117 current_vertex))
118         print("Current sequence: " + v.vertex_list_to_string(
    self.act_sequence))
119         print("No-Op: Staying in current vertex")
120         self.time_passed += 1
121 
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.
    vertex_list_to_string(self.act_sequence))
129             if not self.terminated and self.time_passed + 1 <
    self.time_limit:
130                 next_vertex = self.act_sequence[0]
131                 if next_vertex != self.state.current_vertex
    and next_vertex.form == v.Form.broken:
132                     self.act_sequence = []
133                     self._no_op()
134                     return
135                 self._move()
136             else:
137                 self.terminated = True
138                 print("TERMINATED")
139         else:
140             print("TERMINATED")
141 
142     def __str__(self):
143         agent_str = f"Score: {str(self.score)}\n"
144         agent_str += f"Number of expansions: {str(self.
    num_of_expansions)}\n"
145         agent_str += f"Number of movements: {(self.
    num_of_movements)}\n"
146         agent_str += f"Total time passed: {str(self.
    time_passed)}\n"
147         agent_str += f"Agent reached goal!!! \U0001f60e\n" if
    self.state.goal_test() else "Goal wasn't reached \U0001f61f"
148         return agent_str
149 
150 
151 def all_agents_terminated(agent_list: List[Agent]) -> bool:
152     for agent in agent_list:
153         if not agent.terminated:
154             return False
155     return True
156 
```

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

File - graph.py

```
 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,
   self.graph_dict.keys()))
20
21     def get_brittle_vertices(self) -> List[v.Vertex]:
22         return list(filter(lambda ve: ve.form == v.Form.brittle
   , self.graph_dict.keys()))
23
24     def get_edges(self):
25         return self.generate_edges()
26
27     def get_vertex(self, id):
28         vertex_to_ret = None
29         for vertex in self.get_vertices():
30             if vertex.id == id:
31                 vertex_to_ret = vertex
32         return vertex_to_ret
33
34     def vertices_ids(self) -> List[int]:
35         id_list = []
36         for vertex in self.get_vertices():
37             id_list.append(vertex.id)
38         return id_list
39
40     def expand(self, vertex: v.Vertex) -> List[Tuple[v.Vertex,
   int]]:
41         return self.graph_dict[vertex]
42
43     def expand_just_vertices(self, vertex: v.Vertex) -> map:
44         return map(lambda neighbor_tup: neighbor_tup[0], self.
```

File - graph.py

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

File - graph.py

```
 89             del self.graph_dict[vertex2][
    index_of_vertex1_in_vertex_2]
 90
 91         def add_or_replace_edge(self, vertex1, vertex2, weight):
 92             if vertex2 not in self.expand_just_vertices(vertex1)
    and vertex1 not in self.expand_just_vertices(vertex2):
 93                 self.add_edge(vertex1, vertex2, weight)
 94             elif weight < self.get_edge_weight(vertex1, vertex2):
 95                 self.replace_edge(vertex1, vertex2, weight)
 96
 97         def replace_edge(self, vertex1, vertex2, weight):
 98             self.delete_edge(vertex1, vertex2)
 99             self.add_edge(vertex1, vertex2, weight)
100
101         def generate_edges(self) -> List[Tuple]:
102             edges = []
103             for vertex in self.graph_dict:
104                 for neighbor_tuple in self.graph_dict[vertex]:
105                     edges.append((vertex, neighbor_tuple[0],
    neighbor_tuple[1]))
106             return edges
107
108         def __str__(self):
109             res = "vertices: "
110             for k in self.graph_dict:
111                 res += str(k) + ", "
112
113             res = res[:len(res) - 2]
114             res += "\nedges: "
115             for edge in self.generate_edges():
116                 res += "(" + str(edge[0].id) + ", " + str(edge[1].
    id) + ", " + str(edge[2]) + "), "
117             res = res[:len(res) - 2]
118             return res
119
120         def copy_graph(self):
121             new_graph = Graph()
122             for vertex in self.get_vertices():
123                 new_graph.graph_dict[vertex] = list(self.expand(
    vertex))
124             return new_graph
125
126         def remove_unessential_vertices(self,
    unessential_vertices_array):
127             for vertex in unessential_vertices_array:
128                 self.delete_all_occurrences(vertex)
129
130         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]] >
    neighbor_tup[1]:
145                         min_edges[neighbor_tup[0]] =
    neighbor_tup[1]
146                 else:
147                     min_edges[neighbor_tup[0]] = neighbor_tup[
    1]
148             self.graph_dict[vertex] = [(key, val) for key, val
     in min_edges.items()]
149 
150     def connect_all_neighbors(self, vertex):
151         neighbor_tuples = self.expand(vertex)
152         for neighbor_tuple1, neighbor_tuple2 in itertools.
    combinations(neighbor_tuples, 2):
153             both_edges_weight = neighbor_tuple1[1] +
    neighbor_tuple2[1]
154             self.add_or_replace_edge(neighbor_tuple1[0],
    neighbor_tuple2[0], both_edges_weight)
155 
156     def get_lowest_cost_edge_between_sets(self, in_set,
    out_set):
157         min_edge = None
158         min_weight = sys.maxsize
159         for in_vertex, out_vertex in itertools.product(in_set
    , out_set):
160             if self.edge_exists(in_vertex, out_vertex) and
    min_weight >= self.get_edge_weight(in_vertex, out_vertex):
161                 current_edge_weight = self.get_edge_weight(
    in_vertex, out_vertex)
162                 min_weight = current_edge_weight
163                 min_edge = (in_vertex, out_vertex, min_weight)
164         return min_edge
165 
166     def MST(self):
167         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(
    in_set, out_set)
173             edge_set.add(edge)
174             in_set.add(edge[1])
175             out_set.remove(edge[1])
176         for vertex in in_set:
177             mst.add_vertex(vertex)
178         for edge in edge_set:
179             mst.add_edge(*edge)
180         return mst
181 
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,
    essential_graph.get_vertices())))
190     essential_graph.zip_edges()
191     return essential_graph
192 
```

File - state.py

```python
import copy as cp
from typing import Union, List

import vertex as v


class State:

    def __init__(self, current_vertex: v.Vertex, vertices_saved: dict, vertices_broken: dict):
        self.current_vertex = current_vertex
        self.vertices_saved = cp.copy(vertices_saved)
        self.vertices_broken = cp.copy(vertices_broken)

    def __str__(self):
        s = "Current vertex: " + str(self.current_vertex) + "\n{"
        for vertex, saved in self.vertices_saved.items():
            s += vertex.id + ": " + ("saved" if saved else "not saved") + "\n"
        for vertex, broken in self.vertices_broken.items():
            s += vertex.id + ": " + ("broken" if broken else "intact") + "\n"
        return s + "}"

    def save_current_vertex(self) -> None:
        if self.current_vertex in self.vertices_saved:
            self.vertices_saved[self.current_vertex] = True

    def break_current_vertex_if_brittle(self) -> None:
        if self.current_vertex in self.vertices_broken:
            self.vertices_broken[self.current_vertex] = True

    def get_unsaved_vertices(self) -> List[v.Vertex]:
        unsaved = []
        for vertex, saved in self.vertices_saved.items():
            if not saved:
                unsaved.append(vertex)
        return unsaved

    def get_intact_vertices(self) -> List[v.Vertex]:
        intact = []
        for vertex, broken in self.vertices_broken.items():
            if not broken:
                intact.append(vertex)
        return intact

    def num_of_vertices_to_save(self) -> int:
```

File - state.py

```
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      def is_vertex_broken(self, vertex):
69          if vertex in self.vertices_broken:
70              return self.vertices_broken[vertex]
71          return False
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
```

File - vertex.py

```python
from enum import Enum
from typing import List


class Form(Enum):
    stable = 0
    brittle = 1
    broken = 2


class Vertex(object):
    def __init__(self, id: int, people_to_rescue: int, form: Form):
        self.id = id
        self.people_to_rescue = people_to_rescue
        self.form = form

    def __str__(self):
        return "[V:" + str(self.id) + ", P:" + str(self.people_to_rescue) + ", F:" + str(self.form.name) + "]"


def vertex_list_to_string(vertices_list: List[Vertex]) -> str:
    s = "["
    for vertex in vertices_list:
        s += f'V{str(vertex.id)}, '
    last_index_of_comma = s.rfind(",")
    if last_index_of_comma != -1:
        s = s[:last_index_of_comma]

    return s + "]"

```

File - enviroment.py

```python
import yaml
from typing import List

import agent as a
import graph as g
import state as s
import vertex as v

def generate_graph(file_name: str):
    graph = g.Graph()
    file = open(file_name)
    lines = file.readlines()
    vertices = {}

    for line in lines:
        line = (line.partition(';'))[0].split(' ')
        if len(line[0]) <= 1:
            continue
        object_type = line[0][1]

        if object_type == 'N':
            pass  # what is it good for?

        elif object_type == 'V':
            id = int(line[0][2])
            people_to_rescue = 0
            brittle = False
            for property in line[1:]:
                if len(property) == 0:
                    continue
                if property[0] == 'P':
                    people_to_rescue = int(property[1])
                elif property[0] == 'B':
                    brittle = True

            u = v.Vertex(id, people_to_rescue, v.Form.brittle if brittle else v.Form.stable)
            vertices[u.id] = u
            graph.add_vertex(u)

        elif object_type == 'E':
            id = int(line[0][2])
            v1 = vertices[int(line[1])]
            v2 = vertices[int(line[2])]
            edge_weight = int(line[3][1:])
            graph.add_edge(v1, v2, edge_weight)

    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,
76                   T: float):
77      vertices_saved = savable_vertex_list_to_vertices_saved_dict
    (world.get_savable_vertices())
78      vertices_broken =
    Breakable_vertex_list_to_vertices_broken_dict(world.
    get_brittle_vertices())
79      if agent_type == 1:
80          return a.GreedyAgent(starting_vertex, vertices_saved,
    vertices_broken, mst_heuristic, expansion_limit,
81                               time_limit, T)
82      elif agent_type == 2:
83          return a.AStarAgent(starting_vertex, vertices_saved,
    vertices_broken, mst_heuristic, expansion_limit,
```

```
                                time_limit, T)
    elif agent_type == 3:
        return a.RealTimeAStarAgent(starting_vertex,
vertices_saved, vertices_broken, mst_heuristic,
expansion_limit,
                                    time_limit, T)

def simulator():
    print('----Welcome to Hurricane Evacuation Problem----')

    print('loading config.yaml file...')
    with open("Inputs/config.yaml", "r") as f:
        config =  yaml.safe_load(f)

    world = generate_graph(config['RUN']['INPUT_PATH'])
    vertices_ids = world.vertices_ids()

    agents = []
    agents_locs_name = []
    for agent_name, starting_vertex_index in config['RUN']['
AGENTS']:
        starting_vertex = world.get_vertex(vertices_ids[
starting_vertex_index - 1])
        new_agent = create_agent(config[agent_name]['ID'],
starting_vertex, world, config[agent_name]['TIME_LIMIT'],
config['WORLD']['TIME_LIMIT'], config['WORLD']['T'])
        agents.append(new_agent)
        agents_locs_name.append((agent_name,
starting_vertex_index))
    print('agents: ')
    print(agents_locs_name)
    print('world: ')
    print(world)
    input('Press Enter to start..')

    while not a.all_agents_terminated(agents):
        for agent_idx, agent in enumerate(agents):
            print(f'\nAGENT {agent_idx+1}): {str(
agents_locs_name[agent_idx][0])}')
            agent.act(world)

    print("\n\n\nAGENTS SCORES: ")
    for agent_idx, agent in enumerate(agents):
        print(f'\n#### AGENT {agent_idx + 1}): {str(
agents_locs_name[agent_idx][0])} ####')
        print(agent)

    print("World at End: ")
```

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

File - priority_queue.py

```
 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.
   num_of_vertices_to_save()
27
28         for elem in self.queue[1:]:
29             elem_value = self.f(elem)
30             elem_amount_to_save = elem.state.
   num_of_vertices_to_save()
31
32             if elem_value < min_value \
33                     or (elem_value == min_value
34                         and elem_amount_to_save <
   min_element_amount_to_save) \
35                     or (elem_value == min_value
36                         and elem_amount_to_save ==
   min_element_amount_to_save
37                         and elem.state.
   does_current_vertex_need_saving()
38                         and not min_elem.state.
   does_current_vertex_need_saving()):
39                 min_elem = elem
40                 min_value = elem_value
41                 min_element_amount_to_save =
   elem_amount_to_save
```

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

File - input1.txt

```
 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
```

File - input2.txt

```
#N 5
#V1
#V2 P2
#V3 B
#V4 P1
#V5 P1

#E1 1 2 W1
#E2 2 3 W1
#E3 3 4 W1
#E4 1 4 W10
#E5 3 5 W1
```

File - config.yaml

```yaml
WORLD:
  TIME_LIMIT: 10000
  T: 0.01

GREEDY_AGENT:
  ID: 1
  TIME_LIMIT: 1

A_STAR_AGENT:
  ID: 2
  TIME_LIMIT: 10000

REAL_TIME_A_STAR_AGENT:
  ID: 3
  TIME_LIMIT: 10

RUN:
  INPUT_PATH: Inputs/input1.txt
  # [[Agent Type, Starting Vertex]]
  AGENTS: [[REAL_TIME_A_STAR_AGENT,1]]

```