Introduction to Artificial Intelligence

Assignment 1

Environment simulator and agents for the Hurricane Evacuation Problem

In this first exercise you are asked to implement an environment simulator that runs a path optimization problem, optionally in the presence of deadlines. Then, you will implement some agents that live in the environment and evaluate their performance.

We are given a weighted graph, and the goal is (starting at a given vertex) to visit as many as possible out of a set of vertices, before an optional deadline. Unlike standard shortest path problems in graphs, which have easy known efficient solution methods (e.g. the Dijkstra algorithm), here the problem is that there is more than 1 vertex to visit, their order is not given, and even the number of visited vertices may not be not known in advance. This is a problem encountered in many real-world settings, such as when you are trying to evacuate people who are stuck at home with no transportation before the hurricane arrives.

Hurricane Evacuation problem environment

The environment consists of a weighted unidrected graph. Each vertex may contain a number of people to be evacuated. An agent (evacuation vehicle) at a vertex automatically picks up all the people at this vertex just before starting the next move, thereby saving them. It is also possible for edges (roads) to be blocked, in this assignment we assume complete knowledge so w.l.o.g. all edges are initially unblocked.

An agent can only do **traverse** and **terminate** actions. The time for traverse actions is equal to w, the edge weight (assumed to be a positive integer). The action always succeeds, unless the time limit is breached before the end of the traversal. The simulation ends when all the time limit is reached, all agents have terminated, or when all people have been collected, whichever comes first.

The simulator should keep track of time, the number of actions done by each agent, and the total number of people successfully evacuated.

Implementation part I: simulator + simple agents

Initially you will implement the environment simulator, and several simple (non-AI) agents. The environment simulator should start up by reading the graph from a file, as well as the contents of vertices and global constants, in a format of your choice. We suggest using a simple adjancency list in an ASCII file, that initially specifies the number of vertices. For example (comments beginning with a semicolon):

```
#N 4      ; number of vertices n in graph (from 1 to n)
#D  4.5             ; Deadline is at time 4.5
#V1                 ; Vertex 1, nothing of interest
#V2 P1              ; Vertex 2, initially contains 1 person to be rescued
#V3                 ; Vertex 3,
#V4 P2              ; Vertex 4, initially contains 2 persons to be rescued

#E1 1 2 W1                ; Edge 1 from vertex 1 to vertex 2, weight 1
#E2 3 4 W1                ; Edge 2 from vertex 3 to vertex 4, weight 1
#E3 2 3 W1                ; Edge 3 from vertex 2 to vertex 3, weight 1
#E4 1 3 W4                ; Edge 4 from vertex 1 to vertex 3, weight 4
#E5 2 4 W5                ; Edge 5 from vertex 2 to vertex 4, weight 5
```

The simulator should query the user about the number of agents and what agent program to use for each of them, from a list defined below. Global constants and initialization parameters for each agent (initial position) are also to be queried from the user.

After the above initialization, the simulator should run each agent in turn, performing the actions retured by the agents, and update the world accordingly. Additionally, the simulator should be capable of displaying the state of the world after each step, with the appropriate state of the agents and their score. The score of an agent is the number of people saved by the agent. A simple screen display in ASCII is sufficient (no bonuses for fancy graphics - this is not the point in this course!).

Each agent program (a function) works as follows. The agent is called by the simulator, together with a set of observations. The agent returns a move to be carried out in the current world state. The agent is allowed to keep an internal state (for example, a computed optimal path, or anything else desired) if needed. In this assignment, the agents can observe the entire state of the world.

You should implement the following agents:

1. A **human** agent, i.e. print the state, read the next move from the user, and return it to the simulator. This is used for debugging and evaluating the program.
2. A **greedy** agent, that works as follows: the agent should compute the shortest currently unblocked path to the next vertex with people to be rescued, and try to follow it. If there is no such path, do **terminate**. Here and elsewhere, if needed, break ties by prefering lower-numbered vertices and/or edges.
3. A **saboteur** agent, that blocks roads. The saboteur works as follows: it does V no-ops, and then blocks the lowest-cost edge adjacent to its current vertex (takes 1 time unit). Then it traverses a lowest-cost remaining edge, and this is repeated. Prefer the lowest-numbered edge in case of ties. If there is no edge to block or traverse, do **terminate**.

At this stage, you should run the environment with **three** agents participating in each run: one greedy agent, one saboteur agent, and one other agent that can be chosen by the user. Your program should display and record the scores. In particular, you should run the greedy agent with various initial configurations. Also, test your environment with several agents in the same run, to see that it works correctly. You would be advised to do a good job here w.r.t. program extensibility, modularity, etc. much of this code may be used for some of the following assignments as well.

Clarification and rationale: Note that this part of the assignment will not really be checked, as it contains no AI, so details are not important. The goal of this part of the assignment is constructing infrastructure for the rest of the assignment(s). E.g. the "human agent" is intended as a debugging and demo aid, and also towards assignment 2, and the greedy agent contains code for shortest path that would be a component in a heuristic in the 2nd part of this assignment.

Implementation part II: search agents

Now, after chapter 4, you will be implementing intelligent agents (this is part 2 of the assignment) that need to act in this environment. Each agent should assume that it is acting alone, regardless of whether it is true. You will be implementing a "search" agent as defined below. All the algorithms will use a **heuristic evaluation function** of your choice. **Note**: the search algorithms you implement should ignore the dealine, and just search for a path that minimizes the time to collect all people in the graph!

1. A greedy search agent, that picks the move with best immediate heuristic value to expand next.
2. An agent using A* search, with the same heuristic. Assume a global constant of LIMIT expansions (default 10000) beyond which we just return "fail", and the agent does just the "terminate" action.
3. An agent using a simplified version of real-time A* doing L (user determined constant, L=10 by default) expansions before each move decision.

The performance measure will be based on the idea of situated temporal planning, that is, recognizing that the search algorithm run also takes time! Rather than using a real-time clock, we will simply have a per-expansion time T, and if N expansions are run in the search algorithm this means that N*T time has passed in real time. The performance of an agent will thus be equal to S, when the path is actually executed in "real time". For simplicity, we will start with T=0, that is, assuming that search takes no time.

Note that if T is non-zero (and/or if the dealine comes before the path can be completed), a path that looks optimal may actually be disastrous because the deadine may be missed in the real world! The number of expansions are: 1 per action for the greedy agent (always 1 expansion), and L for the "real-time A*". We will run all algorithms with values of T being 0, 0.000001, and 0.01 and report the performance results.

Observe that for "situated" A* this is a very hard problem for an algorithm that considers the deadline, as we do not know the number of expansions before we do the search! Many search applications just assume that the number of expansions is maximal and equal to LIMIT, or ignore the effect on meeting the deadline, but determining how to do this in a reasonable manner is an open research problem. (Do this to earn an MSc, or even a PhD.)

Bonus version: construct a search agent as above, but in addition allow one saboteur agent also acting in the environment. Your search agent needs to take this into account. Observe that although this seems to be a multi-agent problem, the fact that vandal is perfectly predictable makes this in essence a single agent search problem.

Addtional bonus - theoretical: What is the computational complexity of the Hurricane Evacuation Problem (single agent)? Can you prove that it is NP-hard? Or is it in P? If the latter, can you design an algorithm that solves the problem in polynomial time?

Deliverables

The program and code sent to the grader, by e-mail or otherwise as specified by the grader, a printout of the code and results. A document stating the heuristic function you used and the rationale for selecting this function. Set up a time for frontal grading checking of the delivered assignment, in which both members of each team must demonstrate at least **some** familiarity with their program...

Due date for part 1 (recommended, not checked or enforced): Monday, November 4, 2019.

For the complete assignment: Tuesday, November 19.