Introduction to AI - assignment 2

Oded Yechiel Matan Rusanovsky

27/11/18

1 Introduction

In this assignment we had to use the hurricane simulator from assignment 1, but operate several agents in parallel. The agents have 3 different options of operation:

1. Adversarial - where the agent tries to get higher score than the opponent.
2. Semi-cooperative - where each agent tries to maximize its score, ties are broken cooperatively.
3. Cooperative - where the agents have to cooperate with each other to maximize the global score.

2 Base agent – Assignment 1 recap

In assignment 1 a basic agent was developed, in which the method BestAction (In actual code you should search for `choose_next_action`) had to be implemented by any other inheriting agent. The basic agent had all of the information required by the agent to make decisions, such as the,

- current vertex
- number of people in the vehicle
- number of people saved

3 Game tree agent

The three agents that are implemented in this assignment are all based on the game-tree-agent.

Algorithm 1 Game tree best action

```
1: procedure BestAction(State, Simulator)
2:     if IsTerminal(Simulator) then
3:         return NoOp()
4:     end if
5:     return RecursiveTree(Simulator, 0, IsZeroSumGame)
6: end procedure
```

Algorithm 2 Terminal position

```
1: procedure IsTerminal(Simulator)
2:     if Simulator.Time ≥ Simulator.Deadline then
3:         return True
4:     else if Simulator.PeopleInTowns == 0 and ∀agent   agent.carrying == 0 then
5:         return True
6:     else
7:         return False
8:     end if
9: end procedure
```

The BestAction function is described in Algorithm 1 and is fairly simple. It just checks if we are at a terminal state, and if so return a NoOp result, otherwise it starts the the recursive function described in Algorithm 3.

The terminal state, shown in Algorithm 2, is a boolean function. It returns true if the time surpassed the deadline, or if there are no more people to save.

The recursive tree function in Algorithm 3 receives a simulator, that hold all of the data and used to emulate each action, depth value representing the depth of tree, a Boolean stating whether it is a zero sum game, and an α, β values (initially set to $-\infty$ and to ∞) for $\alpha - \beta$ pruning.

The algorithm starts by checking for terminal states and checks if the depth is maxed out for cutoff purposes. For terminal states the actual value, i.e., number of people saved, of each agent is returned. Maxing out depth returns a heuristic (line 6) of how many people each agent may save, disregarding the other agent and assuming each town with people is the only town with people. This same heuristic was used in assignment 1.

The function continues, if not returned, to a for loop (line 10) that branches every possible action of an agent, and calls recursively to itself with an emulated simulator where its the next agent's turn, and depth is increased by 1.

The result of this action is compared with all of the other actions and the best action's value is returned if the depth is larger than 0, otherwise the best action itself. For each type of agent (adversarial, cooperative, semi-cooperative) the comparison function is different and will be elaborated in the following section.

In case the game is a zero sum game (in our case, adversarial) an alpha-beta pruning is implemented to try and trim branches that will not affect the final result. The value that was used for the pruning was the score that the max agent achieved ("MaxAgentScore" in the algorithm below).

Algorithm 3 Game tree recursive tree expansion

```
1: procedure RecursiveTree(Simulator, Depth, IsZeroSumGame, α, β)
2:     if IsTerminal(Simulator) then
3:         return Value(Simulator)
4:     end if
5:     if Depth reached MAX_DEPTH: then
6:         return Heuristic(Simulator)
7:     end if
8:     BestValue ⟵ [−inf, −inf]
9:     BestMove ⟵ −1
10:    for each Action ∈ possible actions of current player do
11:        Simulator.EmulateAction(Action)
12:        NewValue ← RecursiveTree(Simulator, Depth+1, IsZeroSumGame, α, β)
13:        if IsBetterAction(NewValue, BestValue) then
14:            BestValue ⟵ NewValue
15:            BestMove ⟵ Action
16:        end if
17:        if IsZeroSumGame is True then
18:            if MaxAgent then
19:                if MaxAgentScore ≥ β then
20:                    return NewValue
21:                end if
22:                α = max(α, MaxAgentScore)
23:            else
24:                if MaxAgentScore ≤ α then
25:                    return NewValue
26:                end if
27:                β = min(β, MaxAgentScore)
28:            end if
29:        end if
30:    end for
31:    if Depth == 0 then
         return BestMove
32:    else
         return BestValue
33:    end if
34: end procedure
```

3.1 Adversarial agent

The adversarial agent is a zero sum game agent. In plain words, the agent prefers that everyone will die as long as the opponent agent will not save more people than him.

Therefore, the IsBetterAction shown in Algorithm 3 in line 13 will be as follows:

Algorithm 4 Adversarial action assessment

```
1: procedure IsBetterAction(()NewValue, BestValue)
2:     Best ← BestValue[CurrentAgent] − BestValue[OtherAgent]
3:     New ← NewValue[CurrentAgent] − NewValue[OtherAgent]
4:     if New > Best then
5:         return True
6:     elsereturn False
7:     end if
8: end procedure
```

The action is better than other options in the case where the difference in values is higher than previous difference in value.

3.2 semi-cooperative agent

The semi-cooperative agent tries to maximize its own value, disregarding the other agent value. This is partially true, the agent will help the other agent if he does not sacrifice his own score. The IsBetterAction will be as follows:

Algorithm 5 Semi-cooperative action assessment

```
1: procedure IsBetterAction(()NewValue, BestValue)
2:     Best ← BestValue[CurrentAgent]
3:     New ← NewValue[CurrentAgent]
4:     if New > Best or (New == Best and NewValue[OtherAgent] > BestValue[OtherAgent]) then
5:         return True
6:     elsereturn False
7:     end if
8: end procedure
```

3.3 Cooperative agent

The cooperative agent tries to maximize the global score, sacrificing his own score for the good of the team. The IsBetterAction will be as follows:

Algorithm 6 Cooperative action assessment

```
1: procedure IsBetterAction(()NewValue, BestValue)
2:     Best ← BestValue[CurrentAgent] + BestValue[OtherAgent]
3:     New ← NewValue[CurrentAgent] + NewValue[OtherAgent]
4:     if New > Best then
5:         return True
6:     elsereturn False
7:     end if
8: end procedure
```

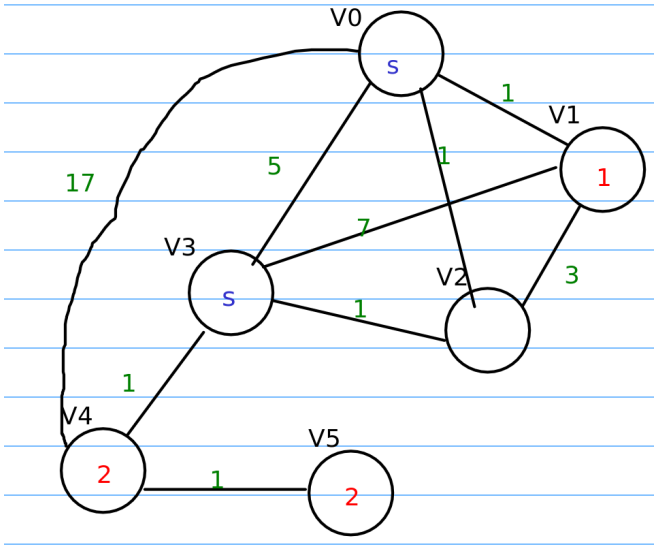

Figure 1: An example of a world where Agent0 will choose a different option per type of game. Agent1 starts at V0. Agent 2 starts at V4 and immediately picks up the 2 people.

4 Example

Fig. 1 shows a world where there are 5 vertices. There are people in vertices, V1, V4, and V5 with 1, 2, and 2 people respectively. There are shelters in vertices V0 and V3. Agent0 start at V0. Agent1 starts at V4 and immediately pick up the people there. The deadline is in 16[time].

- Adversarial agent

  It is clear that agent0 has to go to V4, thus finishing the game with 0 people saved for each agent. Otherwise, agent1 will reach a shelter and rescue 2 people.

- Semi-cooperative agent

  Agent0 will go to V1 picking up the person and will go back to V0 maximizing the amount of people he could save. Agent1 will go immediately to V3 saving the 2 people he is carrying, as there is no time to go to V5.

- Fully cooperative agent

  Agent0 will go to V2 in order to conserve as much time as he can so agent1 will reach V5 and save 1 more person. Agent0 is sacrificing his score for the greater good.