

CS51 Project 2c: Dead or Alive

Due: Friday, April 24th 2009 at 4:59 PM

Total Points: 51 (including 10 style points)

1 Assignment Overview

In this portion of the assignment, you will develop strategies for a simple game. In the game, there are two types of agents: there are “missile” agents that try to catch the other agents in the game and “fleeing” agents that are attempting to reach a safe spot on the graph while avoiding the missile targeting it.

2 Introduction to the World

Again we have given you some code to initialize an engine with the game rules for this week. If you run `proj2c.scm` after downloading it you should see a graph appear with several agents. When you hit `step`, several more agents will appear. Some of these agents are missiles (the faster agents), and are labelled with the id of the agent they are targeting. The other agents are the fleeing agents, and they must attempt to reach the safe zone in the graph, represented by the yellow node. The game is over when all of the fleeing agents have either been destroyed (hit by the particular missile targeting them) or have reached a safe spot. The game will also end if more than 20 time steps have passed. When the game ends, the game begins anew with newly chosen nodes for the safe-spot and the missile spawning point.

Note that, at the start of a game it is possible for missile agents or the safe zone to appear on top of a node already occupied by a fleeing agent. In this case, you may see fewer agents on the board.

Your job will be to implement the code for both the missiles and the fleeing agents. We will do this using a dynamic programming algorithm.

3 Game Theory Redux: Memoization

Recall from class that we talked about dynamic programming in the context of fitting multiple lines to a set of points. As mentioned in lecture, Dijkstra's algorithm is another example of a dynamic programming algorithm. It turns that dynamic programming is a very general tool that is often used to solve complicated programming problems very efficiently. We'll explore ideas similar to dynamic programming for solving this game. Unfortunately, even with these tools at our disposal, the solution to this game will still be quite expensive computationally.

Let's first review the basic tenets of dynamic programming. There are several components to a problem that make it amenable to dynamic programming. The most essential component is that the the solution to the overall problem should be computable by decomposing the problem into smaller sub-problems and optimally solving them. For example, one can find the shortest path to a node on a graph by first finding the optimal shortest path to nodes adjacent to it and then choosing the one with the smallest overall shortest path. In our case, the optimal strategy to a game can be computed given the optimal strategy for each game state resulting from some move by the opponent.

The second component is that sub-problems should happen multiple times. This is not necessary to apply dynamic programming (DP), but makes DP more efficient when it occurs. This is because DP can memoize (save) the result of its computations and reuse them in the future. In our case, it is possible that the same game state will reappear several times, meaning that memoization is advantageous to performance.

4 Applying Memoization to our Game *(30 points total)*

The first step is to formalize exactly what the game is. We view this multiplayer game as a series of two-player games: each fleeing player has exactly one missile tagged for them. Only contact with this missile will cause them to lose. Also, multiple players can occupy the safe spot. Thus, each player's goal is to reach the safe spot before the missile reaches that player. In order to make the game interesting, the missiles travel faster than the players. Thus there are some situations where the fleeing player cannot avoid the missile, though the player should still try to stay alive as long as possible. The goal of this section is to develop algorithms such that both the missile and the player make the optimal decisions to achieve their objectives.

We begin by making a few simplifying assumptions. First, we assume that edge weights and speeds are small integers. We will also assume that a player is always on an integral fraction of the path from one node to the next. By this we mean that if an edge has length 4 and a player is traveling on this edge, they are either $1/4$, $1/2$, $3/4$ of the way along the edge at any given timestep. The next simplifying assumption we make is that players can only travel from one node to the next; they cannot turn around in the middle of an edge and return to where they started. The only place where they can make a decision is at the

nodes.

The game is over when one of the players wins. The missile wins if it occupies the same space as the fleeing player at the same time. The fleeing player wins if they reach a safe spot. In case of ties, the fleeing player wins.

Because the computation of the game space is expensive, we will compute it once before hand and store it in our handy-dandy onboard flash memory. This way, while actually playing the game, each player need only lookup the optimal next move for the current game situation, rather than recompute the entire game every time. This technique of memoization is a key component of efficiency—it makes it unnecessary to recompute the same sub-problem multiple times. Unfortunately, just as in real life, our model may not be exactly the same as reality. As a consequence, we may sometimes need to recompute the game play if the other player either does not behave optimally, or if the actual outcome of the game differs from our model's projected outcome.

The *state* of the game is a pair containing the position of the missile and the position of the fleeing player. The goal of the strategy is to find a winning set of moves for your player or, barring that, avoid your death for as long as possible (hoping that the other player will make a mistake). Thus, the game search algorithm should return not only the outcome of the game, but also the number of steps to that outcome. Our algorithm is quite similar to our approach to the tree game from problem set 3. We assume that at each stage, the opposing player makes optimal moves in reaction to ours and decide what move to make based on this assumption. Thus, to find the optimal move for an agent, we look at each possible move they could make and then recursively compute the next optimal move given that move and the opponent making their best move in response. Choose the move that gives the best result. For simplicity, we assume that when both players are at a choice state (i.e., a node), the chasing agent decides where to move first, and the fleeing agent can see this decision.

If you're looking for a challenge, skip the next paragraph and jump straight to the code. If you want more guidance in how to implement a strategy for this game, read on.

4.1 Hierarchy of Outcomes

The first thing we need to do is define a way to compare outcomes. There are three possible outcomes: either the fleeing agent is caught, it escapes to the safe spot, or the missile runs out of fuel (also known as the attacker timing out). We also consider the amount of time until the outcome happens, and the first (optimal) moves both agents make to reach this outcome. Thus, an outcome is a tuple $(r, t, m1, m2)$ where r is the result, t is the amount of until this result, and $m1$ and $m2$ are the first moves that the agents make. Given two outcomes o_1 and o_2 , we write $o_1 < o_2$ if o_2 is more favorable to the chased player. From the chased agent's perspective, escaping is always preferable the missile timing out, which is preferable to being caught. Given two outcomes that both result in escaping, the outcome leading to escape in less time is preferred. Given two outcomes both resulting in capture, capture in the longer time is preferred.

4.2 Game Play Strategy

The following pseudocode implements one particular strategy for solving this game reasonably efficiently. You will need to translate it into scheme. This pseudocode does not memoize the results, and also does not return the first optimal moves for both players. You should make sure to do both.

```

fun get-perfect-outcome chaser-loc chased-loc time-rem
  ;; s is either "CAUGHT", or it contains the new positions of the agents.
  ;; t is time until a choice point is reached.
  let s,t = <run to next choice point from chaser-loc and chased-loc>
      new-time-rem = time-rem - t in

  ; notice the order (tie goes to the chased agent)
  if new-time-rem <= 0
    then return {TIMEOUT in t}
  else
    let chaser-loc-e = <get chaser-loc from s>
        chased-loc-e = <get chased-loc from s> in

    ; again, tie goes to the runner
    if chased-loc-e = refuge
      then return {ESCAPED in t}
    else if chaser-loc-e = chased-loc-e
      then return {CAUGHT in t}

    ; For simplicity, we assume the chaser moves first, and the chasee
    ; gets to see the chaser's move. So, if the chaser is at a choice
    ; point, optimize over its moves.
    else if chaser-loc-e is at a node
      then <recursively determine the optimal move. To do this, you'll
      need use a helper function (that we've given you) to find all
      possible moves, and optimize over them using the helper functions
      we've given you. You may find the function outcome-join useful
      as well. You should assume that the chased player plays perfectly
      given your choice of move>

    ; we must be at a choice point for the chased-player
    else
      <use the same idea as above to recursively determine the optimal
      move>

```

Notice that catches can happen in one of two ways: either both players end up at a node at the same time, in which case the missile should win (unless that node is the refuge) or

the missile catches the player along an edge (in which case 'CAUGHT is returned by the function that simulates playing the game until the next choice point is reached).

Exercise 1.*30 points***(a)** [5 points]

Implement `outcome-cmp`. This function should take two outcomes, o_1 and o_2 , and return an integer x such that $x < 0$ if and only if $o_1 < o_2$ in the sense discussed above.

(b) [25 points]

In the `strategizer` class, implement the body of `get-perfect-outcome` following the pseudocode we define above. (Hints: use `hash-ref!` and `l2l` to handle memoization; think carefully about how to exploit the functions we've given you and general higher-order combinators to write elegant and short code.)

5 Editing the Agent

(10 points total)

Now we are ready to begin editing the agent to follow our strategy.

Exercise 2.*10 points***(a)** [5 points]

Add a member variable in both the fleeing agent and the seeking agent to store the game strategizer you wrote earlier. Add code to the `init-agent` function to initialize the strategizer with the graph. You may use any of the functions defined in section 6 in your solution.

(b) [5 points]

Update `new-goal-node` in both the fleeing and the seeking agents to make an appropriate call to the strategizer to compute the optimal next move. Again, look at the methods supplied in `query-obj` (defined in section 6) to figure out which methods provide the data you need to feed to your model.

6 Supplied Code

As in the past projects, you can access bits of game-specific data through the `query-obj` variables, which are instances of the class that implements the game-specific rules for this week. You are free to look through the code to see how we are implementing the rules (catch-me.ss in your `plt` directory), but the functions you will most likely need to use in this assignment are among the following.

```
;; Public accessor methods of "query-obj" variables
;; Returns the speed of the chasers or the chased agents.
(define/public (get-chaser-speed) ...)
(define/public (get-chased-speed) ...)

;; Returns the time before the chaser agents die.
(define/public (get-chaser-time-rem) ...)

;; Returns the node at which the safe zone is residing.
(define/public (get-refuge) ...)
```

7 Testing Your Code

There are no points allocated to testing your code this project, but you still should be testing your code in order to verify that your solutions are correct. One way to test your strategizer class will be to hand-compute the solutions it should be finding. The easiest way to test your agents however, will likely be to run the simulation and observe the actions of your agents yourself.

You may change the variable `width`, `height`, and `missing-nodes` in order to change the size and layout of the graph that your agents are running on.

8 Finishing Up

(1 point total)

Exercise 3.

1 point

Please tell us how much time you spent on this problem set. You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.