University of Alberta
Computing Science Department
CMPUT 366 - Fall 2024

Assignment 3
Due date: November 01
8 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on Canvas as a zip file (the entire "starter" folder). If you are using an environment to install your Python packages, you should not include its folder in your submission.

Overview

In this assignment you will implement a Hill Climbing search to find programs for playing MicroRTS, a real-time strategy game developed for research purposes. Such programs are known as "programmatic policies". It is called a policy because it informs exactly what the agent needs to do in every step of the game. It is programmatic because it uses a domain-specific language to define the search space.

MicroRTS

MicroRTS is a minimalist real-time strategy game, played between two players, where each player controls a set of units of different types. The goal of MicroRTS is to defeat your opponent by gathering resources, building units, and eventually battling the opponent's units. Figure 1 shows a screenshot of the game.

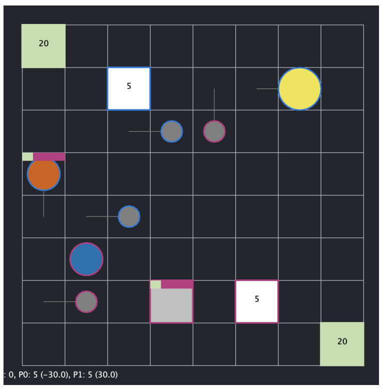

Figure 1: Screenshot a match of MicroRTS played on an 8 × 8 map.

You will be able to complete the assignment even if you do not understand the game. Thus, if you are not interested in understanding the programs your system will come up with, you can skip to the next section.

In Figure 1, except the green squares on the top-left and bottom-right corners, all other circles and squares represent units one of the players controls. The green squares represent resources the players can collect and use to build structures and train more units. Here is a list of unit types in MicroRTS.[1]

- **Base** - Base units have 10 hit points, cost 10 resources to build, and take 250 time units to be built. They can only execute one action, produce. And they can only produce Workers.
- **Barracks** - Barracks units have 4 hit points, cost 5 resources to build, and take 200 time units to be built. They can only execute one action, produce. And they can produce Light unit, Heavy units or Ranged units.
- **Worker** - Worker units have 1 hit point, cost 1 resource to build, and take 50 time units to be built. Workers can carry one resource. They can move (10 units of time per move), attack (5 units of time per attack, and cause 1 damage), and harvest (20 units of time), and return (10 units of time). Workers can only attack, harvest or return to units that are in the cell immediately up, left, right or down.
- **Light** - Light units have 4 hit points, cost 2 resources to build, and take 80 time units to be built. Light units can move (8 units of time per move) and attack (5 units of time per attack, and cause 2 damage).
- **Heavy** - Heavy units have 4 hit points, cost 2 resources to build, and take 120 time units to be built. Heavy units can move (12 units of time per move) and attack (5 units of time per attack, and cause 4 damage).
- **Ranged** - Heavy units have 1 hit points, cost 2 resources to build, and take 100 time units to be built. Ranged units can move (12 units of time per move) and attack location (5 units of time per attack, and cause 1 damage) that is at distance 3 or less.

Figure 1 shows an example of a MicroRTS game state, where gray small circles represent the Worker units; the blue circle represents a Ranged unit; the yellow circle represents a Heavy unit; the orange circle represents a Light unit; green squares represent the Resources, which can be collected by Workers; white squares are the Bases, one for each player; the gray square is a Barracks. The line around the shapes shows who owns the unit (you might have to zoom in to see the line). A blue line denotes player 1, and a red line denotes player 2. For instance, the yellow circle (the Heavy unit) belongs to player 1.

The numbers you see on the Base units (the white squares) are the number of resources each player owns that can be used for training units and building other Bases and Barracks. Also, this number can increase if Workers collect resources from the green squares and return them to the Base.

We will consider policies written in the MicroLanguage, which is a domain-specific language for MicroRTS.

```
for(Unit u) {
  u.build(Barracks,Up,1)
  for(Unit u) {
      u.attack(Closest)
      }
}
```

Figure 2: Example of a programmatic policy written in the MicroLanguage.

[1]The following description was extracted from `https://github.com/Farama-Foundation/MicroRTS/wiki/Game-Definition`.

The program encoding a policy is called in every iteration of the game to determine the action of each unit the player controls. In the example shown in Figure 2, the outer for-loop iterates over all units the player controls. The instruction `u.build(Barracks, Up, 1)`, will attempt to build a Barracks with unit `u`. Note that this is possible only if `u` is a Worker unit. If `u` is not a Worker unit, then the instruction will be ignored. Generally, any line of code that does not apply to a unit will simply be skipped.

The program from Figure 2 has an inner for-loop, which also goes over all units the player controls. The instructions inside the inner for-loop have higher priority than instructions outside it. This is because, in all programs written in the Microlanguage, once an action is assigned to a unit, this action cannot be changed. The inner for-loop will then attempt to assign an action to all units that can attack the opponent. The Microlanguage has an order of the type of units in which the for-loop iterates: first Workers, then Barracks, Base, Light, Ranged, and finally Heavy. If the player controls one Base, one Barracks and four Workers, the Worker will be the first unit to be iterated over in the for-loop, then the Barracks, and finally the Base.

Problem Definition

You will implement a Hill Climbing algorithm to find programs able to defeat the policies shown in Figure 3.

```
for (Unit u) {
      u.build(Barracks, Down, 2)
    u.attack(Closest)
    u.train(Ranged, Right, 15)
}
```

```
for (Unit u) {
  u.train(Worker, Left, 10);
  u.harvest(3);
  u.attack(Weakest);
  u.harvest(10);
  u.attack(Farthest);
}
```

Figure 3: The programs we need to compute a best response for.

Since this is a two-player game, in game theory terms, you will search in the space of programs given by the MicroLanguage for programs encoding a best response to the strategies the programs in Figure 3 encode.

Starter Code

The starter code already implements most of the functions you will need to find the best responses to the programs in Figure 3. In this section, we list the important functions you will need to implement the Hill Climbing search. As usual, feel free to explore the project. Note that this is not a standard implementation of MicroRTS; this version of the game was written by David Aleixo specifically for this course assignment. We hope some of you might want to use this code base for the optional project of the course.

The initial candidate solution used to seed the search can be generated with the following instruction, which generates an empty program.

```
initial_candidate = ScriptsToy.scriptEmpty()
```

A candidate can be evaluated with the following instructions.

```
utt = UnitTypeTable(2)
```

```
pgs = PhysicalGameState.load(map, utt)
gs = GameState(pgs, utt)
eval, auxiliary_eval = evaluate(candidate, target_program, gs, max_tick)
```

The variables `utt`, `pgs`, and `gs` are boilerplate. They define the configuration of MicroRTS we use and load the map in which the game will be played. The function `evaluate` takes as input the candidate program we want to evaluate, the target program, the initial game state `gs`, and the maximum number of game steps we allow in our simulations, `max_tick`. This function will play the candidate program against the target program in the two different locations of the map: in one of the matches, the candidate starts in location 1, while in the other match, it starts in location 2. The value of variable `eval` is the average score of the candidate in these matches. We count the score of 1.0 for a win, 0.0 for a loss, and 0.5 for a draw.

In addition to the evaluation we care about, the function `evaluate` also returns the value of an auxiliary function (`auxiliary_eval`). The signal for the win-draw-loss function is sparse. Here is why. If you have a candidate program that cannot defeat the target program, it is unlikely one of its neighbors will be able to defeat the target program. The auxiliary function measures other metrics of the game, such as resources collected and units trained. The auxiliary function is meant to help form a better optimization landscape for Hill Climbing. In your implementation, you will use the auxiliary function to break ties in search.

The last component that is required to implement Hill Climbing is a function to generate the neighbors of a candidate solution, as shown in the instructions below.

```
neighborhood_function = Neighborhood()
...
neighbors = neighborhood_function.get_neighbors(candidate, n_neighborhood)
```

The instance `neighborhood_function` only needs to be instantiated once, and this is already done for you in the starter code. The instance can then be used by calling its `get_neighbors` method, which receives a candidate and an integer specifying the number of neighbors and returns the set of neighbors as a list. This function is stochastic. This means that even if we always start the search from the initial candidate given by `ScriptsToy.scriptEmpty()`, we will potentially observe different solution candidates in search.

The starter code also includes a function that allows you to see a MicroRTS match played by one of your programs. For example, the instruction below plays `program_1` against `program_2`.

```
visualize_game(program_1, program_2)
```

The main function in the starter code is programmed to show the match between the best response you compute to each of the policies shown in Figure 3.

Implement Hill Climbing (4 Marks)

Implement Hill Climbing for computing the best responses to the policies in Figure 3. The two policies are already provided in `main.py`, with variables `program_target_1` and `program_target_2`. The starter code provides the first few lines of the search, as shown in Figure 4; complete this implementation. The function should return two parameters. The first is the program encoding the best response, while the second is the number of evaluations performed in search (i.e., number of calls to the evaluate function).

```
def search(target_program, neighborhood_function, num_neighbors, max_tick, map):
        utt = UnitTypeTable(2)
        pgs = PhysicalGameState.load(map, utt)
        gs = GameState(pgs, utt)

        prog = ScriptsToy.scriptEmpty()
        best_eval, best_auxiliary = evaluate(prog, target_program, gs, max_tick)
        total_number_evaluations = 0
        # continue implementation from here

        return None, total_number_evaluations
```

Figure 4: Search function that needs to be completed in the assignment.

The evaluation function returns the average outcome of the matches played against the target program, as computed by the `evaluate` function. In case of a tie—there will be many—you should favor programs with larger values of the auxiliary function. Note that the search can stop once the main evaluation encounters a program with a score of 1.0, as this means we have encountered a best response to the policy.

Implement Restarts (4 Marks)

Implement a restarting strategy for Hill Climbing. The search will return the last solution encountered, as it reaches a local optimum. The restarting strategy will then reinitialize the Hill Climbing search. The initial candidate will be chosen as follows: with 50% chance, we will seed the search with the best program, according to the evaluation function, across all restarts (case 1); with the remaining 50% chance we will initialize the search from scratch, with the initial candidate `ScriptsToy.scriptEmpty()` (case 2).

In case 1, the search will explore the region of the space around the most promising program encountered across all previous searches. While in case 2 the search removes the biases from the best program encountered thus far and will start from an empty program.

Notes on Evaluation

Testing your implementation in this assignment is trickier than the previous assignments where we provided test cases. In this assignment, we only provide two target policies for which you need to compute a best response. While your implementation might be successful in finding best responses, it might not necessarily be implementing Hill Climbing and/or the restarting strategy correctly. You must ensure correctness.

The code starter fixes the seed used with the random generator, as an attempt to control the stochasticity of the experiments. However, the behavior of your program will be different than mine if you perform the operations that involve a call to the random library of Python in a slightly different order than what I have in my own implementation. This does not necessarily mean that your implementation is incorrect; it only means it is different. Having said that, my implementation takes approximately 8 minutes to find a best response to the first program and about 1 minute to find a best response to the second program.

Questions to Explore

Some questions you might want to explore as you write your code. Why does restarting from the best solution encountered across different restarts can be helpful? Why does restarting from scratch can also be helpful? What do you think will happen if you do not use the auxiliary function in search? Can you think of ways to improve your implementation?