University of Alberta
Computing Science Department
CMPUT 366 - Winter 2025

Assignment 3
Due date: March 14
12 marks

Search & Planning in AI (CMPUT 366)

Submission Instructions

Submit your code on Canvas as a zip file (the entire "starter" folder) and the answer to the question of the assignment in a pdf. The pdf must be submitted as a separate file so we can visualize it on Canvas.

Overview

In this assignment, you will implement Levin tree search (LevinTS) to solve one of the puzzles from the game The Witness. Most of the code you need has already been implemented in the assignment package. In the next section, we explain the puzzle your system will learn how to solve and some of the key functions you will use from the starter code. You can reimplement all these functions if you prefer; their use is not mandatory. However, the assignment must be implemented in Python.

You will use several data structures in this assignment, including lists, sets, dictionaries, and heaps. This is similar to the structures used in previous assignments. So, if you have implemented the previous assignments, you can skip the Python tutorial available on Canvas.

This document is organized as follows: First, we describe the Witness puzzle that your system will learn to solve. Then, we will describe some of the key functions in the starter code. Note that we will leave the discussion of some other functions from the starter code to the sections where we detail what needs to be implemented. Finally, the last section includes a couple of questions that need to be answered.

The Witness Puzzle

To solve the puzzles, you will draw a line on a grid from an initial location to a goal location while separating the bullets in the grid into regions. The puzzle is solved if the line connects the initial to the goal location and no region contains bullets of different colors. Figure 1 shows an example of a puzzle. The line is drawn by always starting at the initial location and choosing one of the four actions: up, down, left, and right. The line cannot cross itself. A solution to this problem is a sequence of actions that draws the line while satisfying the puzzle's constraints. The solution on the right of Figure 1 is given by the following actions

`right, right, up, up, up, up, right, down, right, up.`

You do not need to know how to solve the puzzles, but knowing how to do so could help with debugging.

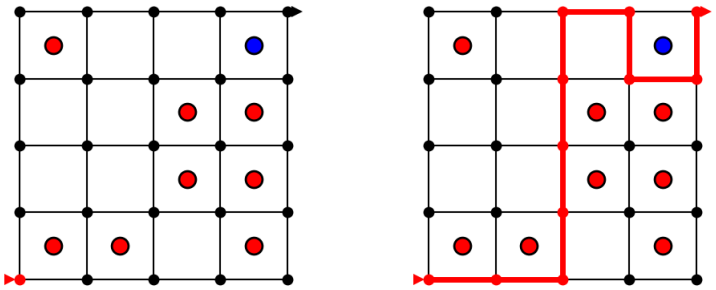

Figure 1: An initial state of the puzzle (left) and a goal state (right). The line starts on the bottom-right corner and finishes at the top-left corner. The goal state shown on the right has three regions: two of them only have red bullets, and the third only has one bullet, which is blue.

Starter Code: `WitnessState` and `TreeNode`

The starter code includes an implementation of the puzzle, in file `witness.py`. You do not need to understand how the puzzle is implemented, only how to use the `WitnessState` class. Next, we describe its key functions.

The `WitnessState` includes the function `successors_parent_pruning` to generate the actions available at a given state. As the function's name suggests, it already performs parent pruning, and it does not generate the action that would undo the move performed by the parent of the node in the tree. The function `successors_parent_pruning` takes an action as input. This action is one used at the node's parent to generate the node. We will discuss below how to store this action. `WitnessState` also includes a function called `apply_action`, which also takes an action as input and transforms the current state according to that action. `successors_parent_pruning` and `apply_action` generate the children of a state as follows.

```
actions = parent.successors_parent_pruning(action_parent)
for a in actions:
    child = copy.deepcopy(parent)
    child.apply_action(a)
```

The cost function LevinTS uses to sort the nodes in the priority is $\frac{d(n)}{\pi(n)}$, where $d(n)$ is the depth of the node n and $\pi(n)$ is the probability of reaching n according to the policy employed. The starter code implements the node in the LevinTS tree in the class `TreeNode` in file `levin_tree_search.py`. `TreeNode` stores all information we need to compute the $\frac{d(n)}{\pi(n)}$-value of a node. It also allows one to recover the solution path once a goal node is found. As we explain below, these paths are used to learn a policy to guide the search.

We will use instances of `TreeNode` in our `OPEN` list. As you can see, the `__lt__` method of `TreeNode` is implemented in the starter code, so Python's `heap_push` will organize the heap sorted by the $\frac{d(n)}{\pi(n)}$-values.

The following shows how to instantiate tree nodes for the initial state and for a node generated in search.

```
# Root of the tree
node = TreeNode(None, state, 0, 0, -1)

# Node generated in search; child was generated by applying action a in the parent
child_node = TreeNode(parent, child, probability, depth, a)
```

The probability and depth of a node have to be computed based on the probability and depth values of its parent. The depth is easy, as it can be computed as `parent.get_depth() + 1`. The probability is trickier due to numerical instability. A naïve implementation would compute the probability as `parent.get_p() * probability_action`. This is problematic because of limited arithmetic precision in computers, which will quickly lead to an underflow. Instead, we will compute the Levin cost in log space. That is, instead of having $\pi(n_i) = \pi(n_0, a_0) \times \pi(n_1, a_1) \times \cdots \times \pi(n_{i-1}, a_{i-1})$, where $\pi(n_j, a_j)$ is the probability of taking action a_j at state n_j, we will have $\log(\pi(n_i)) = \log(\pi(n_0, a_0)) + \log(\pi(n_1, a_1)) + \cdots + \log(\pi(n_i, a_i))$. Since we will store the log of the probability, the Levin cost of each state should be computed as follows

$$\log(d(n)) - \log(\pi(n))\,. \tag{1}$$

This cost function is equivalent to the original $\frac{d(n)}{\pi(n)}$, but it prevents numerical issues for being in log space.

Let us return to discussing how to compute the probability of a node. Whenever we generate a node in the tree, we use our policy to compute the probability of each action at that node. The following excerpt shows how this is done. Here, `child` is an instance of `WitnessState` and `child_node` is an instance of `TreeNode`.

```
action_distribution = model.get_probabilities(child.get_pattern())
action_distribution_log = np.log(action_distribution)
child_node.set_probability_distribution_actions(action_distribution_log)
```

The instance `child_node` will be added to the priority queue and eventually expanded. Before adding to the queue, you will have to set its cost with `child_node.set_levin_cost(levin_cost)`, just like how we did with the A* search (before adding a node to the queue, we had to compute the node's f-value). The code above uses the method `get_pattern()` from `WitnessState`, we will come back to this method later.

Once a node is popped out of the queue and is expanded, we will do the following.

```
probability_distribution = parent.get_probability_distribution_actions()
for a in actions:
    child = copy.deepcopy(parent.get_game_state())
    child.apply_action(a)
    prob = parent.get_p() + probability_distribution[a]
    child_node = TreeNode(parent, child, prob, parent.get_depth() + 1, a)
```

The method `get_probability_distribution_actions()` retrieves the vector with the probability of each action that we stored with the method `set_probability_distribution_actions` when the node was generated. Note how we sum the probabilities, and we do not multiply them. This is because we are operating in log space, which is numerically stable.

Finally, in `main.py`, we have the following line, where `easy_instance` is an instance of `WitnessState`.

```
easy_instance.plot()
```

This will plot on the screen the state `easy_instance` represents. This plotting function can help debug your LevinTS implementation later.

Starter Code: `Model` and `Bootstrap`

We learn a policy by using a search-and-learn approach, called Bootstrap, with a context model. A context model defines a set of features of the states given in search, and for each context, it stores the probability of taking each action. In this assignment, we will use a simple context. We will look at the cells around the tip of the line that we are drawing to solve the puzzle, as illustrated by the three examples in Figure 2.

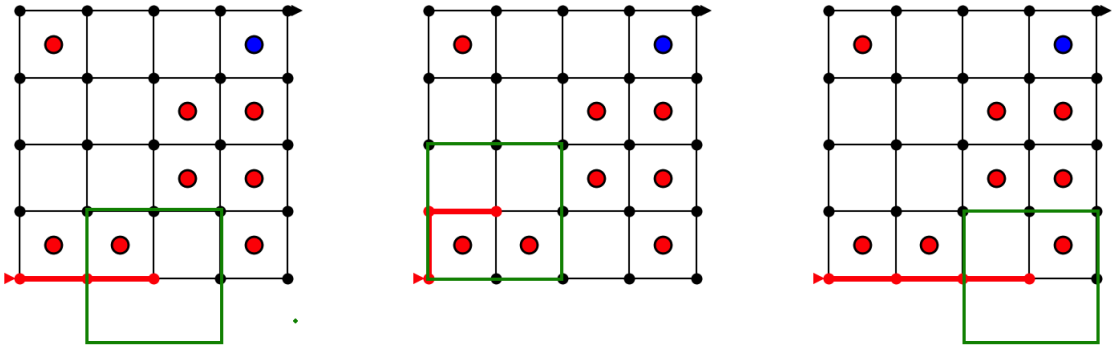

Figure 2: The green box defines the context of each of these states. We will look at the cells around the tip of the line. Since we will look at four cells, each context will have four numbers. One for the cell at the top-left, top-right, bottom-left, and bottom-right corners of the square around the tip of the line. The contexts for the states on the left, middle and right are $(1, 0, -1, -1)$, $(0, 0, 1, 1)$, and $(0, 1, -1, -1)$. Here, 0 means an empty cell, 1 means a cell with a red bullet, and -1 means a cell outside of the puzzle's grid.

The model is a dictionary, where the keys are the contexts, and the values are vectors with one entry for each of the four actions in the puzzle (up, down, left, and right). Initially, the model returns 0.25 for any action and context. As we solve problems, we will use the solution paths to update the model so it becomes more accurate and helpful in solving these puzzles. For example, if we find a solution path to a puzzle, where the action "up" was taken in context $(0, 1, -1, -1)$, then we will increase the probability of taking action "up" from that context and reduce the probability of all other actions. One way of implementing this is by storing a counter in each vector entry. For example, initially, we have

```
pattern = (0, 1, -1, -1)
context_model[pattern] = [1, 1, 1, 1]
```

Here, the first entry of the vector corresponds to a counter to action up, the second to action down, then right, and left. The probability of taking the up action according to this model can be computed as

```
context_model[pattern] / np.sum(context_model[pattern])
```

Here, `np.sum` is the Numpy function for summing the values of a vector. For the initial values of $[1, 1, 1, 1]$, the probability of moving up will be 0.25. However, after seeing a solution path in which the action up was taken i in the context $(0, 1, -1, -1)$, we will update the model as follows, where the index of action up is 0.

```
context_model[pattern][0] += increment
```

The variable `increment` tells us how much we will increase the probability of taking the action up in the future, given that we used this action in a solution path in the past. For example, if `increment` is 10, then the vector for this context will be $[11, 1, 1, 1]$ and the probability of going up whenever this context is encountered will now be $\frac{11}{14} = 0.785$, which is much higher than the 0.25 from the initial model.

The class `Boostrap` implements the training procedure that will allow us to learn a policy through this context model. The idea is simple. We will perform a series of budgeted searches on a set of problems. That is, given a budget B (we will use $B = 2000$), we will attempt to solve each problem in our data set, but limiting ourselves to B expansions. If a solution is found, we will use the solution path to update the context model table, as we described above. For each context and action taken in that context, we will update the counter in the table, to favor taking this same action in future searches. If a problem is not solved, we will move on to the next one. If we finish a complete iteration through all problems in the data set without solving any, then we will double B. This way, we compensate for our weak policy with more search.

The process described in the previous paragraph is known as the Bootstrap process, and it is already implemented for you in the starter code. You should take a moment to read the file `bootstrap.py` to understand how it works. Its main method is `train_model`, which receives a search algorithm and a model to be updated. The method modifies the model as it learns how to solve the set of problems passed to its constructor. The bootstrap process generates a log file with important information about the training process. You should also take a moment to understand the information saved in that file.

The file `main.py` already comes with the correct usage of the Bootstrap process. You need to uncomment the lines marked in the file, as we detail in the instructions below.

Implement LevinTS (5 Marks)

You will implement methods `get_levin_cost` (1 Mark) and `search` (4 Marks) of file `levin_tree_search.py`.

The method `get_levin_cost` receives an instance of a `TreeNode` and returns the Levin cost in log space for that node (Equation 1). If you follow our suggested implementation, described in the "Starter Code" sections of this document, the `TreeNode` will store the actual depth of the node and the probability in log form. So, when computing the Levin cost, you need to use Numpy's log function to transform the depth to log space.

The `search` method receives the initial state (instance of `WitnessState`), a model, and a budget value. The method returns the solution cost, the number of expansions, and the solution path. The solution path can be recovered using the method `recover_path`, which is given in the starter code. If the expansion budget is reached without finding a solution, then return `-1, expansions, None`.

Recall that LevinTS is a best-first search that uses the Levin cost to sort the nodes in the `OPEN` list. Similarly to A*, it also uses a `CLOSED` list to avoid transpositions. Since LevinTS does not have guarantees on the quality of the solution it returns, you should return the solution path as soon as a goal state is encountered. You can verify if a node is a solution by calling the function `is_solution` of the `WitnessState` class.

Test your LevinTS implementation by running it on the easy instance from `main.py`. Note how it uses a context model that always returns a uniform probability distribution for all actions (`UniformModel` in

`main.py`). The use of `UniformModel` allows you to debug your search without worrying about learning a policy. The instance `easy_instance` from `main.py` is easy enough to be solved with a uniform policy.

It is advisable to proceed to the next section only once your LevinTS can solve the easy problem.

Implement Update Function of the Model (3 Marks)

Implement the method `update` of class `Model` in `model.py`. This method receives an instance of the class `Trajectory` (see `levin_tree_search.py`), encoding a solution path of a search problem. You should use the methods `get_states` and `get_actions` to iterate through the states and the actions taken in each of these states. The i-th entry of the list `get_actions` returns stores the action taken at the i-th state in the list `get_states` returns. You will use this information to update the entries of the contexts in `self._table`.

You will use two methods for updating the entries of `self._table`. For each state in the solution path, use the method `get_reversed_pattern` to retrieve two contexts: regular and reversed, as in the line below.

```
regular, reversed = state.get_reversed_pattern()
```

This function returns two contexts for the same state. In this Witness puzzle, the colors are symmetric. For example, the behavior of the algorithm should be the same for the contexts $(0, 0, 1, 1)$ and $(0, 0, 2, 2)$—the first has two empty cells and blue cells and the second has two empty cells and two red cells. The method `get_reversed_pattern` returns the two possibilities, the first is whatever context we see on the screen if we plotted the puzzle; in the second, we swap red to blue and blue to red. You will test two implementations:

1. Update only the entry of the table of the regular context for each state on a solution path (version 1).
2. Update the entries of the table of both contexts, the regular and the "reversed" (version 2).

Experiments and Questions (4 Marks)

1. (2 Marks) Use Bootstrap to learn a policy using the context model implemented in the starter code while **updating the model only with the regular context (version 1)**. The code for calling Bootstrap is in `main.py`, you need to uncomment it once you are ready to run the experiment.

   Then, explain what you observe as the output of the Bootstrap system. Namely, the Bootstrap process prints on the screen, for each problem p in the data set, if it is solved. Explain what you observe in the printed output in terms of the chances of the search solving an instance in the early attempts versus the later ones. Why does it have this behavior?

2. (2 Marks) Use Bootstrap to learn a policy using the context model implemented in the starter code while **updating the model only with the regular and reversed contexts (version 2)**. Then, examine the log file `training_bootstrap`, which is generated automatically. In that file, you will find information from experiments with version 1 and version 2. Explain the numbers that you see, comparing version 1 to version 2. In particular, why do you see a difference in the number of problems solved in each iteration of the Bootstrap (each sweep over all problems in the data set)?