EECS 391: Introduction to AI (Spring 2020) Programming Exercise 4

General Instructions:

Please comment your code extensively so we can understand it, write efficient code using good data structures and sensible variable names. We will use the git logs to ensure that each person is contributing equally to all exercises. Therefore, any code written by you must be clearly logged under your own name/network ID by git. You will not receive credit for code committed by anyone other than yourself even if you wrote it, or if we cannot make out who committed the code. **There will be NO exceptions to this rule.**

The redmine system used by csevcs has issues with some Unicode characters. Please ensure your git names use only standard ASCII characters.

Your commits are due on csevcs.case.edu by 11:59pm on the due date specified after the question. You will receive a 10% bonus for any solution turned in a week or more in advance of the due date. You must notify us of such a commit by email. You can use one late day each week (up to Saturday 11:59pm) with a penalty of 20%. Submissions after Saturday 11:59pm for any week will not be graded. Other bonus points may be awarded for exceptionally well-written and commented, easy to read, clean and efficient code.

These programming exercises will use SEPIA ("Strategy Engine for Programming Intelligent Agents"), a game environment written by other CWRU students tailored to writing AI players. SEPIA is a real-time strategy (RTS) game (though we will not use the real-time aspects in these exercises). RTS games generally have "economic" and "battle" components. In the economic game, the goal is to collect different types of resources in the map. Typical resources are "Gold" and "Wood." Resources are collected using units called "Peasants." Having resources allows the player to build other buildings (Peasants can be used to build things) and produce more units. Some of these units are battle units that can be used to fight the opponent. Games generally end when one player has no more units left; however, in SEPIA, a variety of victory conditions can be declared through XML configuration files. For example we can declare a victory condition to be when a certain amount of Gold and Wood have been collected, some number of units of a certain type built, etc.

You need Java 1.8 to run SEPIA. Note that, although the components of SEPIA you will use have been fairly well tested by now, there is still a possibility of bugs remaining. If you encounter behavior that seems strange, please let me or the TAs know.

Programming 4: Automated Resource Collection (First Commit 3/20 (30 points), Final Commit 3/27 (45 points))

The data for this exercise is in the ProgrammingExercise4.zip file on Canvas.

In this exercise you will write a forward state space planner to solve resource collection scenarios in SEPIA.

The scenarios you will solve are built around the "rc_3m5t.xml" map and the "midas*" configuration files. In this map, there is a townhall, a peasant, three goldmines and five forests. Assume the peasant can only move between these locations. When the peasant is next to a goldmine, it can execute a HarvestGold operation. This requires the peasant to be carrying nothing and the goldmine to have some gold. If successful, it removes 100 gold from the goldmine and results in the peasant carrying 100 gold. The three goldmines in this map have capacities 100 (nearest to townhall), 500 and 5000 (farthest from townhall) respectively. When the peasant is next to a forest, it can execute a HarvestWood operation. This requires the peasant to be carrying nothing and the forest to have some wood. If successful, it removes 100 wood from the forest and results in the peasant carrying 100 wood. The five forests in this map each contain 400 wood. Finally, when the peasant is next to the townhall, it can perform a Deposit[Gold/Wood] operation. This requires the peasant to be carrying something. If successful, it results in the peasant being emptyhanded, and adds to the total quantity of gold or wood available by the amount carried by the peasant.

Your code should consist of two parts: a planner and a plan execution agent (PEA). The planner will take as input the action specification above, the starting state and the goal and output a plan. The PEA will read in the plan and execute it in SEPIA.

Implement a forward state space planner using the A* algorithm that finds minimum makespan plans to achieve a given goal. Here makespan is time taken by the action sequence when executed. Most actions take unit time, but note that for some actions, such as compound moves, you will only be able to estimate the makespan---that is fine for this assignment. Design good heuristics for the planner to guide it towards good actions. However, it is important *NOT* to "pre-plan" by using your knowledge of the game. For example, do not decide to first instantiate the Gold actions followed by the Wood actions, though we know the order does not matter. Similarly, the planner needs to figure out that Deposits should follow Harvests; you should not hardcode this. Once a plan is found, write it out to a plain text file as a list of actions, one per line, along with any parameters. Your PEA can then execute this plan in SEPIA. (The PEA does not have to read the text file, you can just directly pass it the plan.)

In order to execute the found plans, you will have to translate the plan actions into SEPIA actions. Since you will be planning at a fairly high level, you may need to write some code to automatically choose target objects if needed so actions can execute properly. You are

welcome to do this in a heuristic manner. If at some point the plan read in by the PEA is not executable in the current state, it should terminate with an error. Else, if all actions could be executed, it should terminate with a "success" output.

Use the midasSmall and midasLarge config files for this assignment. Set the initial state to be: the peasant is emptyhanded, the gold and wood tallies are zero and the capacities of all mines and forests are as above. Write STRIPS-like descriptions of the actions. (a) Set the goal state to be a gold tally of 200 and a wood tally of 200. Produce a plan and execute it in Sepia. (b) Set the goal state to be a gold tally of 1000 and a wood tally of 1000. Produce a plan and execute it in SEPIA. In each case, output the total number of steps taken to actually execute the plan.

In the files provided, there is an included StripsAction interface which has two functions. preconditionsMet(GameState) takes in a GameState and returns true if that state satisfies all of the preconditions of the action. apply(GameState) takes in a GameState and applies that action's effects, returning the resulting game state. You can use this to define different classes that implement actions like Move and Harvest if you like. This is similar to SEPIA's Action class, but specific to this assignment.

The GameState class is similar to the previous assignment. It is intended to capture the abstract state the planner reasons over, computed from SEPIA's state.

The PlannerAgent class contains an empty AstarSearch function that takes in a GameState and returns a Stack of objects implementing the StripsAction Interface. The PlannerAgent includes a predefined method that writes the stack to a file. It calls the toString method on each Strips Action in the plan and writes the output to a line. The PlannerAgent also includes an instance of the PEAgent which is instantiated with the plan found by AstarSearch. There is a createSepiaAction function in PEAgent that takes in a StripsAction and returns a SEPIA Action where you will construct an implementable action corresponding to your plan actions.

The Position class abstracts the position of a unit.

Your git repository is located at https://csevcs.case.edu/git/2020_spring_391_N, where N is your Canvas group number. **Because this is a long assignment, you are expected to make intermediate weekly commits to the repository.** Each week, we expect to see substantial progress made in the implementation by each person. Progress must be *proportional to the points allocated that week*. Create a separate subdirectory "P4agents" in your git repository, place your agent in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do **NOT** commit any files other than the java files containing your agent and the README.