EECS 391: Introduction to AI (Spring 2018) Programming Exercises

Please comment your code extensively so we can understand it, use good data structures and sensible variable names. It is expected that each person in a group doing an assignment will perform equal work. If the weekly commits do not reflect this, the person putting in less work will receive a lower grade.

Your commits are due on csevcs.case.edu by 11:59pm on the due date specified after the question. You will receive a 20% 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 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.

Download the ProgrammingData.zip file from the course website or Canvas. This file contains the maps, configuration files, enemy agents and agent skeletons referred to in the exercises below. 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 1: Familiarization with SEPIA (Due 2/2 (50 points))

Read the documentation at http://engr.case.edu/ray_soumya/sepia/html/ and go through the exercises of building and compiling a simple resource collection and combat agent. Using the same framework, write simple extensions to the resource collection and combat agent. For example, you might try building a Farm (500 gold, 250 wood), a Barracks (700 gold, 400 wood) and a Footman or two (600 gold) in the resource scenario, or coming up with a smarter attack strategy in the combat scenario.

If you see errors such as “No class found” or “Unable to instantiate” when trying to run SEPIA, check that (i) you are running Java 1.8, (ii) your classpath is correctly specified, (iii) code for your agents and the enemy agents provided are in the correct locations as expected by their packages and (iv) you are using the correct XML config file (the maps are also XML, so check that you did not specify the map as the config).

Create a separate subdirectory “P1agents” in your git repository, place your agents in it and push to csevcs. Include a short README file containing a brief description of what your modified agents do, and (iii) your experience with the documentation, and anything you found confusing. Do NOT commit any other code.

Programming 2: Pathfinding (First Commit 2/9 (40 points), Final Commit 2/16 (60 points))

Write an agent that can move around a given map by implement the A* search algorithm discussed in class. In the file AstarAgent.java, find the function AstarSearch.java and fill it in. This function should return a path from the starting location to the goal. The rest of the agent code has already been filled in so that once you implement the search, the agent will execute it in the game. During this step, it will output its progress with helpful messages that should let you debug your code. You can also watch VisualAgent (the GUI window showing the map) to see how your found path is being followed. The terminal output will also show the total time taken by the process. You should try to reduce this as much as possible (i.e. write efficient code!). For a proper estimate of the time taken, you can stop VisualAgent from running by deleting or commenting out the corresponding <agentclass> lines from the configuration file.

For A*, you will need a heuristic function. The Chebyshev distance is a good heuristic for this purpose. This is defined as follows: $D((x1,y1),(x2,y2))=\max(|x2-x1| , |y2-y1|)$. To test your algorithm, use the maze maps provided. In each map, a Footman is trapped in a maze. Somewhere in the maze is an enemy Townhall. The provided code will use your implementation to find a path that takes the Footman to the Townhall and attack it. When the Townhall is destroyed, the game will end. If it is not possible to guide the Footman to the Townhall, print “No available path.” in the terminal and quit (call System.exit(0)).

You can use VisualAgent to check that your agent is behaving correctly. You can run the maze maps just using VisualAgent, left click on the Footman and right click on the Townhall to see the solution according to the built in pathfinding routines.

A* allows to you to do basic pathfinding, but in a real game, many units will be wandering around, and pathfinding is complicated. We will now simulate this using a simple scenario.

One of the provided maps (maze_16x16h_dynamic) is an environment with an enemy footman, controlled by the wicked EnemyBlockerAgent. This agent will try to prevent your

agent from destroying the enemy Townhall by blocking their path (it won't attack you otherwise). To get around this, use the "shouldReplanPath()" function in AstarAgent. Here, you can check to see if the current path is blocked. If so, you should return true and the agent will redo the search from that point. Try to write a nice function which is smart about when it needs to redo the search so you minimize the total time spent in searching and execution.

We may award up to 10 bonus points if (i) your code is clean and comprehensible and (ii) your total runtimes are among the top three in the class. Please do NOT use map-specific or heuristic-specific optimizations in your A* implementation. This means your implementation should be able to handle any map and any heuristic.

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 proportional to the points allocated that week. Create a separate subdirectory "P2agents" 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 other code.

Programming 3: Playing against an Opponent (First Commit: 2/23 (20 points), Second Commit: 3/2 (30 points), Final Commit: 3/9, (50 points))

In this exercise, you will implement the alpha-beta algorithm for playing two player games to solve some SEPIA scenarios.

Provided are three maps and config files in data/, an opponent agent in archer_agent/ and skeleton files for your agent in src/. The archer agent is not part of any package, so place it at the root of your class hierarchy. The Minimax agent is part of the edu.cwru.sepia.agent.minimax package. The maps have two Footmen belonging to player 0 and one or two Archers belonging to player 1. Footmen are melee units and have to be adjacent to another unit to attack it, and they have lots of health. Archers are ranged units that attack at a distance. They do lots of damage but have little health. In these scenarios, your agent will control the Footmen while the provided agent, ArcherAgent, will control the Archers. The scenario will end when all the units belonging to one player are killed. So your goal is to write an agent that will quickly use the Footmen to destroy the Archers. However, these Archers will react to the Footmen and try to outmaneuver them and kill them if they can. You will use game trees to figure out what your Footmen should do.

Your agent should take one parameter as input. This is an integer that specifies the depth of the game tree in plys to look ahead. This is specified in the configuration XML file as the "Argument" parameter under the Minimax agent. At each level, the possible moves are the joint moves of both Footmen, and the joint moves of the Archers (if more than one). For this assignment, assume that the only possible actions of each Footman are to move up, down,

left, right and attack if next to the Archer(s). The Archers have the same set of actions: move up, down, left right and attack (which means they stay where they are). Thus when your agent is playing, there are 16 joint actions for the two Footmen you control (if you are next to an Archer, you also have the Attack action). When ArcherAgent is playing, it has either 5 or 25 (joint) actions depending on whether there are one or two Archers. You can see that even for this simple setting, the game tree is very large!

Implement alpha-beta search to search the game tree up to the specified depth. Use linear evaluation functions to estimate the utilities of the states at the leaves of the game tree. To get these, you will need state features; use whatever state features you can think of that you think correlate with the goodness of a state. A state should have high utility if it is likely you will shortly trap and kill the Archer(s) from that state.

As part of your implementation, you will need to track how the state changes as you take actions. You should write your own simple state tracker for this. *Don't* use SEPIA's state cloning functions; they will be inefficient and may introduce additional problematic issues.

Since the game tree is very large, the order of node expansion is critical. Use heuristics to determine good node orderings. For example, at a Footman level in the game tree, actions that move the Footmen away from the Archers are almost always guaranteed to have low utility and so should be expanded last. If adjacent to an Archer, a Footman should always attack. Similarly, if the Archer(s) is (are) very far away from your Footmen, they will not run but shoot your Footmen, so expand that action first, and so forth.

We will award up to 10 bonus points for well written code that is able to quickly search a large number of plies relative to the rest of the class (e.g. if you are in the top three runtimes to finish a scenario with a fixed large number of plies). Note that we will test your code with other maps than the ones provided with this assignment.

In the skeleton agent files, MinimaxAlphaBeta is the main class. It includes the main alphaBetaSearch method and a node reordering method (orderChildrenWithHeuristics). These are the methods you will fill in. A helper class, GameState, is provided to track SEPIA's state, which can then be used to compute the utility (getUtility) and to find the possible results after taking an action from a state (getChildren). You will fill this in as well. You should not modify the GameStateChild class. It just pairs an action map with a GameState.

The code is structured in this way so that your alphaBetaSearch method can be implemented abstractly (i.e. it will not need to contain any SEPIA specific code). The SEPIA related code should reside in GameState. You can add fields and functions to GameState as needed (add comments to explain what they are doing).

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 proportional to the points allocated that week. Create a separate subdirectory "P3agents" 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 other code.

Programming 4: Automated Resource Collection (First Commit 3/23 (30 points), Final Commit 3/30 (45 points))

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.

Note that this description goes beyond the STRIPS language in that we are describing numeric quantities. For this assignment, you should ignore this aspect and write a planner to handle this scenario assuming a "STRIPS-like" semantics according to the description above. 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.

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 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, (ii) the text files containing the plans you found and (iii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.

Programming 5: Enhanced Automated Resource Collection (First Commit 4/6 (30 points), Final Commit 4/13 (45 points))

Use the midas*BuildPeasant config files for this assignment. This is a continuation of resource collection. Now suppose you have an additional action, BuildPeasant (the Townhall can execute this action). This action requires 400 gold and 1 food. The Townhall supplies 3 food and each peasant currently on the map consumes 1 food. If successful, this operator will deduct 400 gold from the current gold tally and result in one additional peasant on the map, which can be subsequently used to collect gold and wood. Since your planner is a simple state space planner, it only produces sequential plans, which will not benefit from the parallelism possible with multiple peasants. To solve this, define additional actions as follows. Write additional Move, Harvest and Deposit operators, Move_k, Harvest_k and Deposit_k, that need k peasants to execute and have the effect of k=1 to 3 parallel Moves, Harvests and Deposits, but will only add the cost of a single action to the plan. To execute such operations, your PEA should then find k "idle" peasants and allocate them to carrying out the Move_k, Harvest_k and Deposit_k operator by finding the nearest goldmine/forest/townhall to go to. Note that your PEA can further heuristically parallelize your found plans, though this reduction in cost cannot be accounted for by the planner. For example, with 3 peasants, suppose you have a Move_1(townhall,goldmine) and a Move_2(townhall,forest) in sequence. Your PEA can parallelize these actions to execute at the same time by noticing that their preconditions can be simultaneously satisfied. This sort of behavior by the execution agent falls under scheduling, a part of automated planning that we did not discuss in class. Be careful when writing heuristics for the BuildPeasant operator. Note that it has an immediate negative effect, i.e. it moves the plan farther from the goal. Somehow your heuristic needs to trade this off against the longer-term positive effect that the parallelism will allow.

Write a STRIPS-like description of the BuildPeasant action. Use the same initial state as above. (a) 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. (b) Set the goal state to be a gold tally of 3000 and a wood tally of 2000. Produce a plan and execute it in SEPIA. In each case, output the total time taken to actually execute the plan found. As before, be careful not to "pre-plan" by using your knowledge of the game.

If you feel ambitious, think about how to incorporate an additional action, BuildFarm. This action creates a new Farm that supplies additional food, which can be used to build even more peasants. At this point, however, you will need a proper scheduler to handle the parallelized action dispatching at each time step.

We will award up to 10 bonus points for well written code that is able to quickly find a short plan so that the total runtime is fast relative to the rest of the class (e.g. if you are in the top three runtimes to finish a scenario with a fixed large resource target). Note that we will test your code with other maps than the ones provided with this assignment.

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. Create a separate subdirectory "P5agents" 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, (ii) the text files containing the plans you found and (iii) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.

Programming 6: Automated Tactical Battles (First Commit 4/27 (50 points), Final Commit 5/4 (50 points))

In this assignment you will write a reinforcement learning agent to learn to fight a tactical battle situation in SEPIA.

The scenario you will solve is built around the "rl_5fv5f.xml" and the "rl_10fv10f.xml" maps and the associated configuration files. In these maps, there are 5 or 10 "Footmen" for each side. Footmen are melee units. Each footman has a fixed amount of "hitpoints." When hit, it loses some hitpoints. If the current hitpoints reach zero, it dies. Your agent will control one side and the provided combatAgent (no package, place it at the root of your class hierarchy) will control the other side. Your reinforcement learning agent's goal is (obviously) to learn a policy to win the battle---i.e., to kill the enemy footmen while losing as few footmen of its own as possible. Note that, unlike in planning, there is no separate "offline" component---the agent will learn by interacting directly with the environment and repeatedly playing the scenario. Also observe that, in this situation, an accurate model is not easy to specify ahead of time, so planning techniques are problematic to implement here.

The parameterized actions available to your agent will be of the form "Attack(F,E)," where F is a friendly footman and E is an enemy footman. At each event point, you will loop through all living friendly footmen and reallocate targets to each one. An event is a "significant" state change, such as when a friendly unit gets hit or a target is killed. You are free to define your own set of events. Note that if you make the events too fine grained, you will have a lot of decision points with very long times between feedback (rewards), which will make the

decision making problem much harder. On the other hand, if they are too coarse, your policy will be suboptimal because you will not react to changes quickly enough during the battle.

When an action Attack(F=f, E=e) has been selected for footman f, it must be executed in SEPIA. Note that such an action is a composite action that involves moving to e and then attacking e. You can use the built-in SEPIA compound attacks to handle this, but be careful of pathfinding issues in close quarters.

The reward function should be set up as follows. Each action costs the agent −0.1. If a friendly footman hits an enemy for d damage, the agent gets a reward of +d. If a friendly footman gets hit for d damage, the agent gets a penalty of –d. If an enemy footman dies, the agent gets a reward of +100. If a friendly footman dies, the agent gets a penalty of −100.

Implement the Q-learning algorithm with linear function approximation (chapter 21.3.2-21.4 in the book) to solve this problem. The Q(s,a) function will be defined as w*f(s,a)+w0, where w is a vector of learned weights and f(s,a) is a vector of state-action features derived from the primitive state. For example, a feature might be: “Is e my closest enemy in terms of Chebyshev distance?” (Note that the enemy e is part of the action.) You should write your own set of features to use. Think about features that will help the agent to come up with good policies. Some useful features are “coordination” features such as “How many other footmen are currently attacking e?”. Some others are “reflective” features such as “Is e an enemy that is currently attacking me?” Yet other features could be things like “What is the ratio of the hitpoints of e to me?”

All of the friendly footmen will share the same Q-function. Thus this same function will be updated whenever any unit gets feedback, and will be consulted to determine the action of every unit. This is OK because all the units have identical capabilities and prevents combinatorial explosion due to multiple units. Note that each footman still senses the “global” state, and “knows” what the other friendly footmen are doing. This is an example of “central control.” Consult the book and the slides to see the update rules for learning the weights w. Note that in order to do the update properly, you need to “decompose” the rewards on a per-footman basis. Fortunately this is easy to do in this case as this is how I have specified the reward signal. (In general the reward signal would just be a function of the joint state and action.)

One other issue to note is that based on your event definitions, the time between successive decision points may vary. (You could of course also decide to have a “dummy event” always happen every 10 steps, say.) In this case you still need to track the accumulated reward over the intermediate time steps and discount them suitably when performing the Q-update.

Your agent should take the number of episodes to play as an option. Each episode is a complete battle up to a victory or defeat. Given this option, the agent should play the number of specified episodes against the opponent. Every 10 episodes, it should freeze its current

policy (Q-function) and play another 5 “evaluation” episodes against the opponent (during which the Q function is used to select actions, but not updated). Then it should produce the following output:

Games Played	Average Cumulative Reward
0	xyz
10	xyz
20	xyz
30	xyz
…	

Each “xyz” is the average (undiscounted) cumulative reward obtained by the current policy/Q-function after 0, 10, 20 etc. games played, averaged over the five evaluation games. This gives you an idea of the rate at which the agent is learning and can be plotted as a learning curve.

Important note: You can fix the PRNG seed for this agent right at the start to 12345 to ensure repeatability, however, be careful not to reset the seed at the start of each episode! It is important to let each episode play out differently for the policy to improve.

Generate learning curves for your agent, one for each of the scenarios, with the x-axis going up to at least 30,000 episodes (you can do more if you wish). The y-axis is the average cumulative reward. Average each curve over five runs, that is, run the Q-learner five times so that you get five curves for each scenario, and then average the curves. In order to run these experiments, you should disable VisualAgent. The instructions to do this are in the config files. Finally, save the weights corresponding to your best found policy in a file called “bestweights.data”.

We may run your agents against each other and award up to a 10 point bonus to agents that perform well. (You can run competitions like this as well if you can convince someone to share their agent with you! Put the other agent in an appropriate location and modify the config files to load it instead of the provided combat agent.)

The provided RLAgent.java has several functions pre-written for you, as well as the facility to save and load the learned policy weights. It also has comments explaining what to do in each function and what SEPIA functions you may need. Note that this agent takes two parameters; the second is a Boolean value that causes weights to be loaded from a file for the initial policy.

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. Create a separate subdirectory “P6agents” 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, (ii) saved weights from your best policy in “bestweights.data”, (iii) pdfs showing the averaged learning curves and (iv) your experience with the API and documentation, and anything you found confusing. Do NOT commit any other code.