Craig Ellis, Andrew Knueven

CS 4150

Assignment 3

**Algorithm Comparison**

For this assignment, we implemented a decision tree and a rap based system for controlling the actions of Ms. PacMan. Both systems read in from a file, create all of the necessary objects to run, and are then fed a game state and asked to make a move.

**Decision Tree**

For the decision tree, the file is a JSON document. Inside the document are mappings from an ID to a JSON document. The IDs act as unique identifier for each node. The inner JSON document has all of the necessary info for that node. First, there is a type. A 1 represents a decision node and a 0 represents an action node (leaf). The rest of the document varies depending on the type. For the decision node, there is a name which corresponds to an implemented class within the code. There is also a list of values which are any values that are needed for the corresponding type (e.g. if you choose ClosestPowerPill, you need to supply a value for the maximum distance that the power pill can be from PacMan for this decision to be correct). Finally, the decision node has a success and a failure node, which are the IDs of the node to go to depending on whether the decision was correct or not. The action node has 3 other fields besides the type. It has a direction which can be AWAY or TOWARDS. This dictates whether PacMan should move away or toward the target. There is a heuristic field, which denotes which heuristic to use when determining the distance between points. Finally, there is a target field. The target can be chosen from a list of predefined target and tells PacMan what it should be moving toward/away from.

The code for creating the decision tree is called from DTPacMan which passes a file name. The Decision Tree class then reads this file and parses it into JSON. The code then builds a binary tree starting with ID 1 as the root node. The code runs recursively down the tree until it hits the bottom and builds itself back up. For an action node, it will instantiate a MoveAway or MoveTowards class. For the decision nodes, it uses a switch statement to instantiate an instance of whatever decision the user wants. Once this tree is built, the makeDecision method can be called, which in turn calls makeDecision on the root node. The tree will handle determining which action to make and then return it.

We fully tested all of the decision tree code. We wrote unit tests for each class and a functional test to ensure that PacMan successfully starts, creates the DT, and can make a decision on the next move. We did find a few minor bugs from our testing, but most of them had been worked out by running the game and finding any issues. The tests were written using Junit5 and mockito.

**RAP**

For the RAP architecture, the file is an array of JSON documents. There are 3 types of JSON documents possible in the RAP architecture, denoted by the type field of the document. Type 0 is a primitive action, type 1 is a task net, and type 2 denotes the list of goals for the system. For a primitive action, there are 5 other fields besides the id and type. There is an action, which is either MOVETOWARD or MOVEAWAY, just like above. There is also a target, just like above, which determines what the other node is that we should compare to. There is also a heuristic which denotes which heuristic to use when determining the distance between two nodes. There is an array of preconditions. The preconditions are strings denoting what needs to be true for this action to run. The string is of the format ‘test,val1,val2,…’ where there are as many values as needed for that test. Finally, there is a post condition, which can set the system goal to a specific goal if desired.

For the task net documents, there are also 5 other fields besides the id and type. There is a goal field which denotes the goal of this set of action. There is a post condition which can set the system goal to a specific goal if needed. There is also a list of preconditions which must be met for this task net to be valid. A task net also has a list of actions. The list of actions will be run sequentially until one passes. Since PacMan is a simple game, we do not need parallel actions so we will simply try the first action then the next and so on until one passes which will return a move. We will use that move for this iteration and then restart on the next iteration. Finally, a task net has a priority field. This field can be any value between 0 and 1. The system uses this to prioritize which task net will be run first, second, etc. for a specific goal.

Finally, there is the list of goals. There should only ever be one type 2 document. It has a field goals that is a list of goals. These have the format of “test,goal” where test is one of the precondition strings above with its appropriate values, followed by a goal from the list of goals (e.g. "nonEdibleGhostWithinDistance,15,StayAlive"). This array is saved by the system and on each iteration, it checks the goals in order to find RAPs that can be run in this iteration. The last entry in the array should be “goal” which is the default goal if none of the others are valid, otherwise the system defaults to StayAlive.

When a RAPPacMan is instantiated, it creates an instance of the RAP class and passes it a string which denotes the file to read. The RAP class then reads in this file and parses each document. If it is type 0, a primitive, it instantiates a MoveAway or MoveToward class and puts it in the map of ID->RAPInstance. If it is type 1, a task net, it instantiates a Task Net class with various fields and adds it to the map and to an array of non-primitive raps. Finally, if it is a type 2, a list of goals, it parses the goals and creates an array of them. Then, it goes through each goal and creates a queue of the RAPs that are applicable for that goal in priority order. When PacMan is asked for a move, it calls the execute method of the RAP class. The RAP class then goes through the list of goals to find the appropriate goal or defaults to StayAlive, if a goal was not previously set. It then gets a copy of the priority queue for that goal and uses that as its execution queue. It checks the first RAP in the queue and if it is a primitive, it checks if it is valid and if it is it executes it and returns the move given. If it is not valid, it removes it from the queue and proceeds. If the first RAP is a task net, it is removed from the queue and the RAP class checks if the RAP is valid. If it is, it adds each of the actions in the task net’s action array to the queue in the order that they are in the array. If not, it proceeds to the next RAPInstance in the queue. On each iteration, the system determines the current goal and then executes the queue for that goal until a move is returned.

We fully tested all of the code for the RAP system. We found a couple small bugs, including some in our files, that we fixed. We also tested RAPPacMan which will instantiate the entire system and make a move, ensuring that the entire functionality of the system is working properly. The tests were written using Junit5 and mockito.

**Compare and Contrast**

For this environment, the decision tree seems much more practical than the rap system. PacMan is a rather simple game where a decision tree would suffice in being able to quickly determine a move to make based on the environment. The rap system is a bit complex for this type of environment. We did not need to add the ability to run actions in parallel because we only ever wanted a single move to be returned. We also added the ability to set the system goal from the post condition of a task net, but we never used it because we wanted to determine the goal on each iteration since it is hard to determine what to do without the entire game state. The rap system also works with a notion of goals, where a system is trying to reach a goal and once it does, it is marked as a success. We wrote code to alert the parent that the action was successful, but since the goal is never really completed, we do not ever remove the goal from the list of possible goals. The environment of PacMan is not able to take advantage of many of the features that are provided by a rap based system. The decision tree system was also much easier to code and understand and was much easier to create test files for. Overall, both systems were implemented and performed well, but for this environment, due to it being rather simple and the fact that goals are never fully achieved, a decision tree system seems to be the better option.

**Results**

Our decision trees and RAP files were tuned using an iterative process. The “data” folder contains 5 decision trees and 7 RAPs, each tuning different bits of the architecture and adding preconditions until we reached what we believe is the most efficient version of each system. To test the performance of each, we hooked up our controllers DTPacMan and RAPPacMan to the Executor class and created a runExperiment method that runs the game a set number of times in asynchronous mode with the visuals off, so it doesn’t take an inordinate amount of time to run a large amount of trials. We selected 100 runs as a good estimate of the average score obtained by each system – the experiment collects and prints the score for each run and then the average at the end. Results for the final iteration of each system can be found in “results.txt” of the root directory. The DT and RAP performed about the same – since the decision tree and RAP are both really only giving Pac-Man one move to do at any given point, they perform about the same given the same set of actions to perform. The average score for each of these was about 11,000 over the 100 runs.