João Silva Nº98737, Rafael Henriques Nº89530, Tiago Jacinto Nº87127

Biased MCTS

Before advancing for the secret levels, it is important to explain which heuristics we decided to use for the biased version of the MCTS algorithm. Firstly, to select an action during the playout we used the Gibbs formula. We also tried Softmax but quickly abandoned this approach due to poor behavior. For each of the actions we created a heuristic that is explained in the following table:

Actions	Heuristics
GetHealthPotion	Firstly, if there are no enemies there is no need to heal, so we return a high heuristic value. If the player has lost 10 or more life points, he uses the distance heuristic otherwise we also return a high heuristic value.
GetManaPotion	The mana is normalized resulting in the higher its value, the less likely for the player to get a mana potion. The heuristic is a sum of this value and the distance heuristic from the base class.
PickUpChest	If the chest is close to the player, we return a high heuristic value, so that the player picks up the chest immediately. Otherwise we use the distance heuristic.
ShieldOfFaith	If the shield value is less than 2 then the heuristic returns a large value, forcing the player to choose that action. Otherwise, the heuristic returns 2 * the fraction of the shield hp the player currently has.
Rest	If there are no enemies, the player does not rest. If the player lost less than 10 points of health it is reasonable to rest, depending on the time left for the end of the game.
Teleport	The player shall use the teleport if he has enough mana and does not need it for the shield of faith.
LevelUp	We simply return a very low value so that the player levels up as soon as he can.
DivineSmite	Only the distance heuristic from the base class is considered.
SwordAttack	If the player health + shield health is higher than the expected hp change, then the heuristic returns the base distance heuristic. Otherwise it returns a large value preventing the player from executing the attack.

Secret Level 1

After analysing the performance of the current world model code using the unity profiler we realized that the GetProperty(), SetProperty() and GetExecutableActions() methods were the ones taking up the most CPU time. In the following table, we show on a single frame, how many calls and how much time it took to make those calls.

Method	Calls	Execution time (ms)
GetProperty	268	3.86
SetPropery	78	0.28
GetExecutableActions	74	39

It is important to mention that this number of GetProperty() calls is only from the WalkToTargetAndExecuteAction.CanExecute() method. We did this for simplicity reasons since GetProperty is being called in many methods therefore to correctly find out the total number of calls we would have to go through all the methods called in a single frame in the Unity profiler. The same applies to SetProperty() but instead of CanExecute(), they were from ApplyActionEffects().

The reason why GetProperty was taking up so much CPU time was due to the recursive world model. Every time GetProperty is called it would have to navigate backwards checking if each parent's dictionary contained that property. A solution is removing the dictionaries and adding a fixed size array that contains all the properties values. Now to access the array we only need the index for the property. To implement this solution we firstly created the PropertiesId static class, like the already existing Properties class, which contains the indexes for the predetermined properties (HP, MAXHP, MANA, TIME etc..). For the properties that vary from map to map, we added a TargetId to the WalkToTargetAndExecuteAction class which is the index of that consumable property (Enemy, Potion or Chest) on the fixed size array. This TargetId is attributed to the action in the constructor of the new world model (WorldModelFEAR). This means that now a call to GetProperty or SetPropery is a simple access to an array which drastically reduced the execution time.

To improve the execution time from GetExecutableActions we removed the IEnumerable.Where() call and the ToArray() call changing the method signature to return a List<Actions> instead of an array. All we had to now was go through each of the actions and if CanExecute() returned true then we add the action to the list. Finally we return the list directly and no longer have to call ToArray().

After all these changes the methods now have the following execution times:

Method	Calls	Execution time (ms)
GetProperty	378	0.04
SetPropery	48	0.02
GetExecutableActions	88	9.15

Secret Level 2

After implementing the biased version of the MCTS algorithm, we implemented yet another version of the MCTS: Limited Playout MCTS. The main idea of this version is to not let the playouts reach the end of the game. Instead, we define a certain depth which will be the max depth a playout can reach. After that we will apply a heuristic function that will evaluate how good, or how likely that state being evaluated is to win the game.

The heuristic: To determine if said state is a good state or not, we thought we could use the various stats of the play. This is at first, a state where the player has more health is better than one with less HP. The same logic applies for the mana, for the remaining time and for the money. Each of these properties is associated with a specific weight which depicts its importance. Having said this, the heurist function we used was the following:

$$h = 0.4 \cdot HP + 0.4 \cdot Money + 0.1 \cdot \frac{1}{RemaingTime} + 0.1 \cdot Mana$$

The HP is the current hp divided by the max hp, the same applies for the Money and Mana so that the resulting value of this expression is always is always between 0 and 1. With this expression we can have a good idea of the likelihood of winning from a certain state. It is important to note that, the HP and the Money have a higher weight, because, the more health the player has the more likely it is to defeat all the monsters and the more money the more closer the player is to winning the game.

Secret Level 3

To test the performance of each of the algorithms per frame we tested the execution time of the Run() and Playout() methods because these are the only ones that vary for each variant of MCTS. Lastly, since all the algorithms perform the same amount of iterations per frame that value is always the same.

MCTS Variant	Run ProcessTime (ms)	Playout ProcessTime (ms)	Iterations Per frame
MCTS	6.33	4,84	20
MCTS+Biased	16.67	15.33	20
MCTS+Biased+Limited	12.99	11.62	20

We also calculated the average total processing time for each of the variants and the win rate. Its important to mention this is the total processing time for a single decision of the algorithm, which in our case takes up 500 iterations.

MCTS	Total Processing Time (s)	WinRate (win/total)
MCTS	0.01	2/10
MCTS+Biased	0.30	8/10
MCTS+Biased+Limited	0.28	7/10

As expected, the regular version of MCTS is the one that has the best performance because it does not have to calculate heuristics when selecting an action during the playout. On the other hand, the MCTS + Biased is the least performant of the three because it must iterate though the executable actions multiples times and calculate the heuristic for each of them. MCTS + Biased + Limited had slightly better performance than MCTS + Biased because it stops the playouts before the world model reaches the terminal state.

After this analysis we decided that MCTS + Biased + Limited is our main algorithm because it had a lower execution time than MCTS + Biased while still having a very good win rate.