TÉCNICO LISBOA

Artificial Intelligence in Games
Project 3 - Decision Making and MCTS

Grupo 19

1. Implemented Features

Lab 5 – Depth Limited GOAP: **Implemented.**
Lab 6 – MCTS: **Implemented.**
Lab 7 – Dealing with Stochastic Environment : **Implemented.**
Lab 7 – MCTS Biased Playout: **Implemented.**
Secret Level 1 – Optimized World State: **Implemented.**
Secret Level 2 – Limited Playout MCTS: **Implemented.**
Secret Level 3 – Comparison of Variants **Implemented.**
Secret Level 4 –Optimization: **Implemented.**

2. Implementation & Design Decisions and Commentaries

Depth Limited GOAP & Vanilla MCTS

For Depth Limited GOAP we just implemented the algorithm as seen in the theoretical slides. We use 10.000 action combinations as a hard limit for each run.
For the implementation of Vanilla MCTS we implemented the algorithm as explained in the theoretical slides. At first we used a naive approach while selecting the best child after exploration, by using Upper Confidence Bounds (UCT). After some runs of our algorithm we discovered that sometimes we had two possible actions with the same score and UCT does not know how to decide which one could be a better child. To resolve this issue we implemented a method that selected an action with the most visited child (Robust Child). In this step we also implemented a mechanism that limits the number of iterations performed per frame, and additional debugging tools.

MCTS Biased Playout

In order to bias our playout we use heuristics in the random selection of actions.
Our idea is to favour some of the executable actions by their heuristic value. Then, we generate a random number between zero and the best heuristic value and select the action with the next heuristic value bigger than that number. For the heuristics themselves, we use positive values between 0 and 200, worst to most preferred, respectively. The idea behind each heuristic for the actions was to avoid attacking strong monsters early, favour the use of "*DivineSmite*" at level 1 to kill skeletons and to conserve mana to use "*DivineWrath*" later on. The rest of the heuristics are obvious choices, such as using a health potion increases its heuristic the more health is recovered, etc. A notable exception is "ShieldOfFaith", which we value low, as it is counterintuitive to our strategy of conserving mana. We also added to the heuristic value a distance bonus so we could influence the character to go for closer item/monster options.

We initially used high variance heuristics, but in order to increase the win rate we ended up using tighter heuristic functions, such as ones based on the character level.

Dealing with a Stochastic Game

To deal with the stochastic actions introduced in lab 7, we use two strategies simultaneously:

Multiple MCTS Runs with best child selection

One of the strategies used was simply to run several MCTS trees at the same time over several instances of the initial node. After these have finished, we add up all the children from the initial node, and select the most visited one overall in the method `BestChildFromSeveral()` in `MCTS`. We use 10 MCTS runs, with all of them sharing a pool of 2000 iterations, for around 200 iterations each.

Multiple playouts for each action with averaging

The second strategy used was to perform a number of actions on the same state, and then average the resulting states into a single one, and using that as the final state from the action applied. For example, if the enemy died more often than not, then the resulting state will have the enemy as dead, with the according XP, and no death or XP otherwise. This is only done when the action selected is "*SwordAttack*", as all other actions are deterministic. This logic is performed in the methods *StochasticPlayout()* and *MergeStates()* in MCTS. We use 5 simulated playouts for all our tests.

Optimized World State Representation

In order to achieve a better performance, we created an alternative World Model implementation similar to F.E.A.R's system, where we have three fixed size arrays, one with the player's properties and two with the world resources (We use one for the resource name, and a mirror array to save the state). When creating a new world state we simply copy the contents of the old state into new arrays. We use a switch case for getting and setting properties, which feed directly into the respective arrays. In order to maintain backwards compatibility with previous world models, we now use an interface *IWorldModel* which all world model implementations share. The following table compares both world models in different methods, using an average of 10 samples each:

MCTSBiased time in millseconds		
Method	**FutureWorldModel**	**WorldModelFEAR**
MCTSBiasedPlayout.Playout()	43.26	23.65
MCTSBiasedPlayout.Run()	1324	830

Limited Playout MCTS & Reward Heuristic

To implement the Limited Playout, we use a hard cap on the playout depth allowed for any simulation and return the result of the last state allowed to be visited. As such, we had to create a representation of the state that differed from the binary win/loss reward that the unlimited MCTS used. As such, our reward functions is as follows:

```
Reward =(XP*20%) + (HP*30%) + (Enemies Killed*30%) + (Resources Consumed*10%) + (Time*10%)
```

All the values are normalized from 0 to 1. The logic for each is as follows: The XP, which represents the level, rewards states with a higher level, as one has better chances of winning in a higher level. The same logic is applied for enemies killed, where 0% represents no enemies killed, and 100% represents all 10 enemies dead. The HP is there to ensure that states where the player is healthy are better than those where the player is not. We use CurrentHP/30 for the HP value, even at level one, to encourage health potion and lay on hands usages at higher levels for bigger total benefit. The Resources Consumed value starts at 100% in the beginning of the game, and reduces 25% for each potion used. This is once more to encourage meaningful usage of these consumables, as the overall state reward will only increase if they are put to good use, and will lower otherwise. The time value starts at 100% and lowers as the game progresses. This is done to help choose actions closer to the player.

We left out the number of coins as this tended to be counterproductive to performing quality actions, as the character doesn't know that close to each chest is a potentially deadly creature. We however chest pickups are biased heavily on proximity, so that the character will pick them up when close to one.

We also left out mana to encourage exploration with the resource and different combinations.

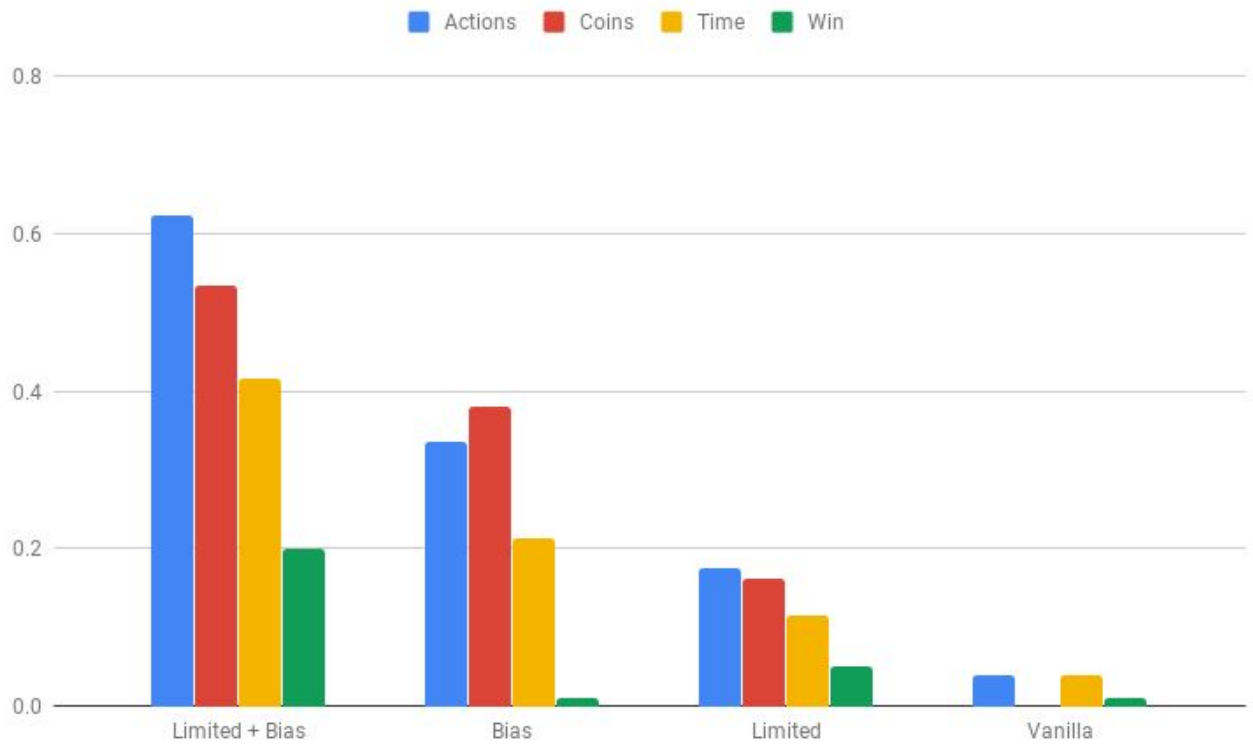
3. Comparison of MCTS Variants Implemented

We tested 4 variants of the MCTS implemented, and tested them for quality of behavior, processing time, and number of playouts. We did not compare for number of iterations as on each version the stopping condition was the iterations themselves. We tested each version for 2000 iterations maximum per MCTS.Run(), and for dealing with stochastic playouts, we performed 5 playout tests, and 10 MCTS trees each run (the 2000 limit is shared between all trees).

3.1 Behavior Analysis

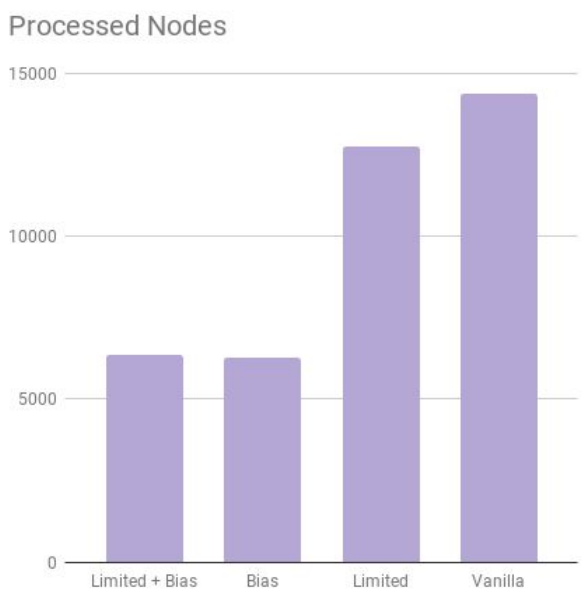
Each version was tested 20 times for behavior, and analysed for win rate, coins gathered, actions taken and time spent. As expected, the base variant of MCTS performed the worst, never winning any games and barely performing any actions before dying. Both the Biased and the Limited+Reward Heuristic versions perform better, acquiring more coins overall, and with more actions performed and longer time spent in game. Although the Biased version runs longer and deeper on most games than Limited+Reward, it will almost always fail to win any games as it tunnels its positive reward sampling on states with 25 coins only, disregarding the fact that next to the chest are enemies which will often kill the character.

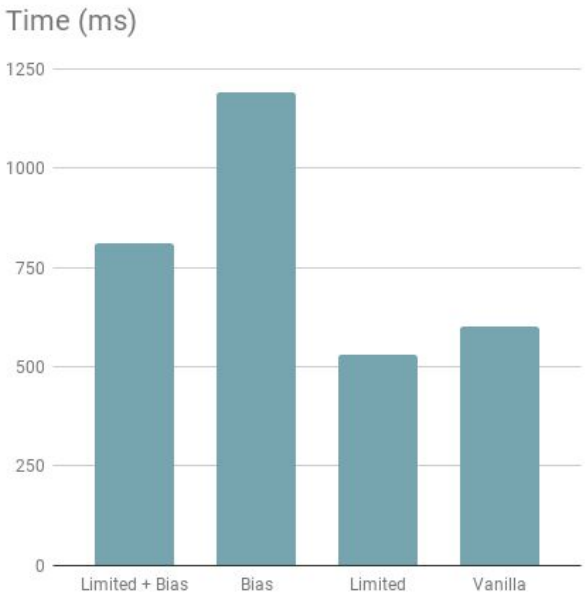
Finally, the Bias+Limited+Reward Heuristic version performs the best on behavior, with better metrics on all 4 categories analysed, and a win rate of around 20%. Thus, this version is clearly the best one for optimising behavior.



3.2 Processing time and Number of Playouts Analysis

For performance testing, we ran each version 5 times, and analysed the first run of *MCTS.Run()* as it is a state where the actions available are the same in all versions.





The Vanilla version performed the most playouts as it was not limited by biases that funnel it into possibly terminal states, nor has a hard limit on playouts possible. However, the Limited version performs faster as it is essentially the same as Vanilla, but with a hard cap on playout depth.

The Bias version is the slowest, as the playout code itself performs slower (checking for heuristics), and doesn't have a limit for playout depth. Lastly, Limited + Bias reaps the benefits of limiting the depth to bring down the execution time closer to vanilla values, while allowing biased playouts.

3.2 Conclusion

Although the Limited version is the fastest in terms of performance, we believe that the vast net gain in behavior from Limited+Biased outweighs the 30% performance loss in time execution, and thus we choose the full version of the algorithm as our main algorithm.

4. Optimizations

Check always best action

There are some actions in the game that are always the best option to pick, irregardless of the situation. As such, when a new MCTS run finds one of these actions in its `InitialNode` child actions, it will instead return one of these actions immediately without spending computation with a full MCTS run. Not only does this improve performance on situations where these actions are available, but it also improves the win rate of the agent by making sure that these actions are not overlooked when they could have been. These actions are:

LevelUp: This action is always the best action to perform, no matter what. It consumes no resources and improves the state of the character, by allowing more actions to be undertaken in the future.

DivineWrath: Given that using this ability kills all enemies, if it is available it should always be picked, as it results in an easy victory for the character afterwards.

PickUpChest (In melee range): When a chest is in melee range from the player at the start of an MCTS run, this signals that the player has just defeated the monster in the chest's vicinity. As such, picking a chest up is an effortless move that saves time in the future should the player need to return to this location to complete the 25 coin goal.

Hard-Pruning Strategies

Conversely to the principle used above, some actions are always bad, no matter what. For example, drinking a mana potion with full mana, or using `ShieldOfFaith` when the shield is at maximum value. We altered the `CanExecute()` methods of these classes to reflect their poor choice, and made them not executable in those circumstances.