IAJ Project 3 - Decision Making

This report has been made in accordance to the guidelines provided for the IAJ's (Inteligência Artificial para Jogos) third project.

Authors

- Diogo Silva (98776)

Index

Depth-Limited GOAP

Algorithm

Starting off with the base GOAP algorithm, we basically implemented it as specified in the theoretical classes. We then limited the number of action combinations that can be processed per frame to 200 (i.e, in each frame we can reach the maximum depth a total of 200 times). As for the maximum depth we settled on a maximum depth of 4. We also added some debug information such as acounter of the total number of actions combinations processed and the processing time.

Improvements

In terms of improvements we managed to decrease the total processing time of the action choice to an average as low as **0.001** by ignoring redundant/less promising branches. Basically, each time we reach a new depth we compute the discontentment value of the current state at said depth. If

we're not on our root state (i.e our depth is bigger than 0), and the current discontentmnet value is worse than our current best discontentment value, then we shouldn't keep expanding that branch, hence we simply backtrack to the previous depth. This tweak allows us to utilize much higher values for our maximum depth than if we decided not to implement it.

Monte-Carlo Tree Search

Base Algorithm

Once again the base algorithm for our MCTS Decision Making was implemented as specified in the theoretical classes. In our MCTS Class we start in the *Run* method which runs our MCTS search for a total of 800 iterations (30 iterations per frame). We start by perfroming the *Selection* of a new node (which selects the first leaf node it finds, by *expanding* a node corresponding to an avaialable but yet not tried out action of the current node and, if all actions have been tried, picking the best child and expanding one of it's children instead and so on). We then compute the reward value using our *Playout* (which in this case selects actions at random). We run a total of 6 playouts and then use the average of their computed reward value as our final reward value for the selected node. Then we *Backpropagate* the computed reward from the final node all the way to the root (by updating the Q and N values of the MCTSNode). In the end we simply select as an action the action corresponding to the best first child of our root node. To compute which child is best we use UCT's (Upper Confidence Bounds applied to Trees) formula with an exploration factor of 1.4f (approximately the square root of 2) for computing the best child in the selection phase, and an exploration factor of 0 to compute the best final child. In case of a tie in the estimated value, we consider the best child to be the one whose duration is smallest (i.e the quickest one to execute). The MCTS class served as a base super class that all other forms of MCTS inherited from, as we will see in the following sections.

Biased Playout

The base MCTS class estimates the reward of a given child node by performing several playouts (i.e simulations) of how the game will evolve after picking the action corresponding to that child. In the base MCTS we simply pick actions at random during the playout phase, but this is somewhat unrealistic. If we have even the slightest amount of faith in our algorithm, we'll believe that it will usually pick, at each step, the best possible action and not just a random one. As such, the base MCTS might sometimes compute a reward score that does not truly correspond to what might be expected. As such, in the Biased Playout MCTS we altered the *Playout* method instead of picking actions at random we compute the probability of each action at each step of the playout being picked according to a Heuristic Value (that is defined in the Action classes).

We start off by using *Gibbs* to assign each of our possible actions a probability that is higher the smaller it's corresponding heuristic is (our probabilities will follow a Gibbs distribution, meaning they all sum up to 1).

Then all we have to do is pick the action. There were several ways of doing this. We could've simply ignored all actions that were smaller than the average value of the probabilities. We could've simply picked the N highest probabilities and picked between them equally. In the end what we did was we computed the average value of the probabilities and in a while loop we computed a random index as normal. We then verified if the random index corresponded to an action whose value was higher than the average value, if it wasn't then we computed another random index and so on. To avoid infinite loops we also limited this while by a maximum of 1000 iterations. The code snippet can be seen below.

```
double averageValue = 1.0f / executableActions.Length;
int randomIndex;
int maxCounter = 0;
while (true){
    randomIndex = this.RandomGenerator.Next(0, executableActions.Length);
    if (probabilities[randomIndex] > averageValue || maxCounter++ > 1000){ // Make sure the
        break;
    }
}
```

We tested all of the aformentioned techniques, but in the end this one seemed to be the one that stroke the right balance of quality and performance.

Limited Biased Playout

The next version of MCTS implemented was the Limited Biased Playout. Until now, both the base MCTS and the Biased MCTS run their playouts until they reach a terminal node, i.e, a node in which the player either wins (has 25 gold) or loses (has less than 1 HP or runs out of time). In this version we limit our playout's depth to go to a max depth of 5. This will surely make our algorithm faster since each of our playouts will be smaller (on the other versions we were running playouts with depths of upwards to 15), but the way we computed the score of each playout was just by checking whether the player won - score of 1.0 - or not - score of 0.0. This won't really work in our new version since most of the times we won't reach a terminal state node. As such we had to create a heuristic function that would take into account the state of the node at the maximum depth, and compute a score accordingly to it's properties. And that is exactly what our *ComputeHeuristicScore* function does.

The way we compute each selected node's playout score is basically done two-fold. First we check to see if the node we have currently selected is a direct child of the node that represents the current world model. By doing this we can force our agent to either avoid certain children whose actions don't make sense (such as getting HP when in the current world we already have full hp), or trying to get a chest that is next to an enemy with a high probability of killing the agent at it's current HP, or force the agent to prioritize others, such a child whose action is leveling up, simply by returning a really low or really high reward accordingly. Basically what we hope with this is to force our agent to ignore first children that are factually bad, or immediately pick a first child that

is too good to pass up. Note that this culling isn't world-dependent (i.e, if the world configuration changes this works equally well without needing any modifications). Secondly, for non-first children we start by checking whether the final state of the playout corresponds to a death (at which point we return -10). If the final state of the playout is NOT a death state, then we compute the difference in HP, Mana, Money, Level and Time between the final state of the playout and the current state of the world. The heuristic score is then computed with the formula `var heuristicScore = HPGain / maxHP + ManaGain / 10.0f + MoneyGain + LvlGain - TimeLoss * 2.5f;` We tried several combinations of these variables and tested giving different weights for each of them, but in the end, this, in conjuction with our "First Child Culling", as we're calling our aformentioned step, resulted in the best observable results.

RAVE - Additional Optimization

One additional optimization we decided to apply to our base biased MCTS algorithm was RAVE - Rapid Action Value Estimation. This optimization required us to add two new variables to our MCTSNodes - QRave and NRave - which hold seperate AMAF (All Moves as First) statistics for each node, and that are equally updated during the Backpropagation stage. Besides having to change the Backpropagation function (in order to also propagate these new values) we had to alter our Biased Playout function in order to record our action history, i.e, the action picked at each iteration of the playout. We also had to alter our Reward class since we now needed to store, for each playout, what the history of chosen actions were. Finally, the new QRave and NRave values come into play when selecting the Best Child, with us having altered the form we computed the estimated value to go in accordance to the RAVE selection as specified in the theoretical slides - `estimatedValue = ((1 - beta) * niu + beta * niuRave) + C * Mathf.Sqrt(Mathf.Log10(node.N) / child.N);` . Note the usage of a new variable - beta - which also has to be computed. Given a child node, it's beta can be computed using it's parent's N value and a new variable *k* which defines the number of iterations at which Q/N and QRave/NRave should be given the same weight. We tried several values of K, and in the end, settled in having a K equal to 10 since that seemed to be what achieved the best performance in terms of quality.

There was a sever problem with this algorithm however: It doesn't really accomodate for several playouts. We can use several playouts and then average their score to obtain our final reward value, but the same can't be done for the action history, since each playout will have a different one. We could've recorded every single action history and then concatenated all of them into a single action history array for example, but after talking to the teacher (and experimentation), we figured that it would be best if, for this version of MCTS, several playouts not be considered (which impacted the performance of the algorithm, especially in stochastic worlds).

Child Culling - Additional Optimization

The idea for Child Culling came from our first step when computing the heuristic score of our Limited Playout MCTS. In that case we were culling first children during the playout stage by returning really high or really low reward values. An alternative that would still allow us to not

consider bad actions or only consider obviously good actions, however, was to modify the Best Children functions to compute an estimated value that was high if the action was obviously the best (such as leveling up), or obviously bad (like picking up hp when currently at full hp). We added this option to our base MCTS, meaning it can be applied to any of our MCTS versions and, as can be seen in the comparison section, it does seem to help our agent behave better overall.

FEAR World Model

The baseline World Model that was given to us by the teacher consisted in a super class - World Model - a subclass, used to represent the Future World State and a subclass of this one used to represent the Current World State (which accesses the properties of the world directly from the Game Manager). Whilst certainly robust, this recursive representation of the world isn't the most performance efficient. As such, we implemented a new World representation in our WorldModelFear class. Firstly, and in order to make it so all of our previously implemented algorithms worked with both the new and old world representation models we created an interface class - IWorldModel - which both the base WorldModel and new WorldModelFear classes implement. In terms of implementation, rather than having dictionaries whose keys correspond to the properties of the world and whose values correspond to the values of said properties at the corresponding state, our WorldModelFear class instead stores all of these in a simple array structure. Additionally, upon initialization it also stores what the index of the array corresponds to what property in a single dictionary that is immutable upon the creation of the class. When getting a property's value, we simply check if said property name is in our dictionary, if it is we get what it's index in our array is, and retrieve it from the index. A similar approach is done when trying to change the value of a property. We wanted to avoid having to have 2 different classes to represent either the current or the future world state (which would also require us to store things like who the state's parent is and iterate recursively over them when doing things like getting the property values). As such, we made sure that the WorldModelFear class can be used to represent both current and future states, hence removing the need for recursion and for keeping track of parent states. The problem with this however, is that every time an action is effectively applied to the world (i.e the game manager actually changes), we need to make sure to call an update function that updates all values in the properties array to the new game manager values (something that wouldn't need to be done if we were to use a specific class for the current world state). Even with this requirement, however, our FEAR-like world model (which can be used both in our MCTS algorithms aswell as with GOAP) leads to faster processing times overall. This is, obviously, more noticeable on the MCTS algorithms (since GOAP is incredibly fast despite whichever world representation is used). The table below shows the average processing time for each of our algorithms in a stochastic world with non-sleeping enemies. For each algorithm we did 10 runs and recorded the average time it took for a decision to be made (and then averaged the values obtained for the 10 runs). We can see that indeed our Fear World representation managed to obtain noticibly better results for all of algorithms except for Depth-Limited GOAP which, due to how small it's average processing time is regardless of the world representation is, could have been due to error. For all other algorithms we our representation managed to decrease the time it

takes to pick an action by around **30%**.

	Normal World	Fear World	Improvement
Depth-Limited GOAP	0.001	0.002	50%
MCTS	0.41	0.23	-44%
MCTS w/ Biased Playout	0.67	0.63	-6%
MCTS w/ Limited Biased Playout	0.48	0.38	-21%
MCTS w/Biased RAVE	0.16	0.1	-37.50%
MCTS w/ Child Culling	0.51	0.38	-26%
MCTS w/ Biased Playout & Child Culling	0.59	0.41	-37%
MCTS w/ Limited Biased Playout & Child Culli	0.47	0.28	-41%
MCTS w/ Biased Rave & Child Culling	0.24	0.17	-30%

Performance Comparison

To evaluate each of our algorithms performances we ran each of them 10 times, computing for each run several parameters, such as average processing time, number of iterations, average selection depth, average playout depth, action combos processed (for GOAP), number of wins (and the average time the agent had left before losing), losses by timeout, losses by death and the average win rate. We did this in three environments - Sleeping Enemies + Deterministic World ; Non-Sleeping Enemies + Deterministic World ; Non-Sleeping Enemies + Stochastic World. All of our environments forced our agent to complete it's objective within 150 seconds.

Sleeping Enemies + Deterministic World

	Avg. Processing Time (s	No. Iterations	Avg. Selection Depth	Avg. Playout Depth	Action Combos Processed	Wins (Avg. Time Left)	Losses (Timeout)	Losses (Death)	Win Rate (%)
Depth-Limited GOAP	0.018	NA	NA	NA	389.4	8 (47.5)	1	1	**80%**
MCTS	0.518	800	5.234	17.35	NA	9 (27.6)	1	0	**90%**
MCTS w/ Biased Playout	0.57	800	4.7025	12.4	NA	9 (22.5)	1	0	**90%**
MCTS w/ Limited Biased Playout	0.52	800	8.4	5	NA	10 (35.3)	0	0	**100%**
MCTS w/Biased RAVE	0.1325	800	4.847	10.8	NA	9 (29.5)	1	0	**90%**
MCTS w/ Child Culling	0.42	800	5.76	18.05	NA	10 (62)	0	0	**100%**
MCTS w/ Biased Playout & Child Culling	0.58	800	6.23	13.57	NA	10 (35)	0	0	**100%**
MCTS w/ Limited Biased Playout & Child Culling	0.37	800	9.45	5	NA	10 (26)	0	0	**100%**
MCTS w/ Biased Rave & Child Culling	0.14	800	6.03	11.35	NA	10 (47.4)	0	0	**100%**

This was the easiest of all environments so it shouldn't come to a surprise that all of our algorithms managed to, more or less, win every time. Depth Limited Goap managed to be our fastest algorithm (which holds true for all environments), with astonishing average processing times way under 0.1. It also managed to win quite a lot with quite a bunch of time left. Managed to outperform it however, with RAVE being the fastest out of all of them. It's also noteworthy that all of our MCTS losses were due to timeouts rather than deaths, but were pretty rare (happening only once or zerto times every 10 runs).

Non-Sleeping Enemies + Deterministic World

	Avg. Processing Time (s	No. Iterations	Avg. Selection Depth	Avg. Playout Depth	Action Combos Processed	Wins (Avg. Time Left)	Losses (Timeout)	Losses (Death)	Win Rate (%)
Depth-Limited GOAP	0.0024	NA	NA	NA	8.385	1 (2)	1	8	**10%**
MCTS	0.425	800	5.112	17.585	NA	9 (22)	2	0	**80%**
MCTS w/ Biased Playout	0.85	800	4.06	16.54	NA	7 (14)	3	0	**70%**
MCTS w/ Limited Biased Playout	0.428	800	6.702	5	NA	10 (10.4)	0	0	**100%**
MCTS w/Biased RAVE	0.236	800	4.886	15.7	NA	4 (2.5)	6	0	**40%**
MCTS w/ Child Culling	0.355	800	6	15.135	NA	10 (33.6)	0	0	**100%**
MCTS w/ Biased Playout & Child Culling	0.42	800	6.61	14.12	NA	10 (30)	0	0	**100%**
MCTS w/ Limited Biased Playout & Child Culling	0.32	800	8.29	5	NA	10 (22)	0	0	**100%**
MCTS w/ Biased Rave & Child Culling	0.1	800	5.68	10.62	NA	8 (14)	2	0	**80%**

It was at this point that GOAP started showing how bad it truly was. The time it took to pick actions was still astonishingly low, but it rarely picked the best actions since it's win rate went down from 90% all the way to 10% (it won only once in our 10 runs and only had 2 seconds left!). Our MCTS'

managed to still do pretty well with win rates of 80%-100%, except for RAVE surprisingly, which showed a really poor 40% win rate, maybe due to it's lack of multiple playouts.

Non-Sleeping Enemies + Stochastic World

	Avg. Processing Time (s)	No. Iterations	Avg. Selection Depth	Avg. Playout Depth	Action Combos Processed	Wins (Avg. Time Left (s)	Losses (Timeout)	Losses (Death)	Win Rate (%)
Depth-Limited GOAP	0.001	NA	NA	NA	224.3	0	1	9	**0%**
MCTS	0.41	800	5.81	19.3	NA	0	1	9	**0%**
MCTS w/ Biased Playout	0.67	800	4.745	24.82	NA	0	0	10	**0%**
MCTS w/ Limited Biased Playout	0.48	800	6.302	5	NA	6 (12.3)	1	3	**60%**
MCTS w/Biased RAVE	0.16	800	5.3	22.3	NA	0	1	9	**0%**
MCTS w/ Child Culling	0.51	800	5.2	21.34	NA	1 (12)	0	9	**10%**
MCTS w/ Biased Playout & Child Culling	0.59	800	8.81	18.1	NA	1 (16)	0	9	**10%**
MCTS w/ Limited Biased Playout & Child Culling	0.47	800	7.32	5	NA	5 (14.3)	2	3	**50%**
MCTS w/ Biased Rave & Child Culling	0.24	800	7.04	18.3	NA	1 (30)	5	4	**10%**

Finally, and more interestingly, we have the hardest possible environment for our agent. None of our algorithms managed to hit or even come close to 100% win rate. A lot ouf our game overs were due to deaths. Basically our algorithms were avoiding fighting enemies they knew they couldn't win against, but they were failing to realize that trying to pick up a chest guarded by an enemy could be lethal to them. Child Culling seemed to help, but not really by much. All in all, the score function in these algorithms failed to take into account how dangerous each action might be since all it cared about was whether the final state was a win or a loss. With this said, our star was the Limited Biased Playout with an 60% win rate. Due to our heuristics and First Child Culling combination, this version of MCTS was proficiently avoiding having our agent get into dangerous sittuations, and it forced it to take safety measures that gave the agent a much higher survibability. It still wasn't fault proof, however, since there were situations in which the RNG simply was not in our favour. There would be instances where, for example, the agent had 25 HP and decided to fight the dragon (which it should be capable of doing), but due to bad dice rolls, still died. More safety measures could be taken in order for the agent to be EVEN MORE careful, but we if we did that we would run the risk of not having enough time to complete our objectives. Processing Time-wise it should also be mentioned that Biased playout always had worse processing times than the other MCTS' due to the whole computing of probabilities and such, meanwhile Limited Biased also had to perform these computations, but since we limited our playout depth to a max of 5, the performance hit wasn't nearly as sever. RAVE was obviously the fastest since it didn't run several playouts. As for our Child Culling option, it did worsen our processing times in general, but not by a big enough number to be concerning.

Settings

To alter which algorithm is being used refer to the checkboxes in the game Manager.

- **Stochastic World**
  - If checked, attacks will be stochastic. Else they will be deterministic
- **Sleeping NPCs**
  - If checked enemies won't attack the player unless provoked. Else, as soon as the player gets near an enemy it will attack them.
- **MCTS Active**
  - If checked, the decision making algorithm used will be the base MCTS

- **MCTS Biased Active**
  - If checked, the decision making algorithm used will be the Biased MCTS
- **MCTS Limited Biased Playout Active**
  - If checked, the decision making algorithm used will be the Biased MCTS with Limited Playout
- **MCTS Rave Active**
  - If checked, the decision making algorithm used will be the Biased MCTS with RAVE
- **FEAR**
  - If checked, the World Representation used will be the one specified in the WorldFear class.
- **Child Culling**
  - If checked, and if using any form of MCTS, Child Culling will be applied when selecting the best child.
- **None of the MCTS Checkboxes Selected**
  - If neither MCTS Active, MCTS Biased Active, MCTS Limited Biased Playout nor MCTS Rave Active are selected, the decision making algorithm used will be the **Depth Limited GOAP**!

Additionally due to a bug related to pathfinding sometimes the agent will fail to generate a path and follow it. When this happens, pressing **SPACE BAR** fixes it.