Report Second Project IAJ

João Vítor ist199246 Sebastião Carvalho ist199326 Tiago Antunes ist199331

2023-10-14

Contents

1 Introduction

The goal of the project was to test different decison making algortihms, and compare their performace and efficiency in terms of win rate.
For the enemies we used Behaviour Trees, having a more basic one for the Mighty Dragon and the Skeletons, and a more advanced one for the Orcs, which we will describe better later.
For the player character, we compared 6 different algorithms: Goal Oriented Behaviour (GOB), Goal Oriented Action Planning (GOAP), Monte Carlo Search Tree (MCTS), MCTS with Biased Playout, MCTS with Limited Playout, and MCTS with Biased Playout and Limited Playout.
To improve performance of MCTS algorithms, we created a flag that allows us to decide if being near an enemy triggers a change in the world state. This flag can be toggled in Character → Decision Algorithm Options → Enemy Changes State.
The tests done for each algorithm were made on the standard grid, not the one with the Orc formation.

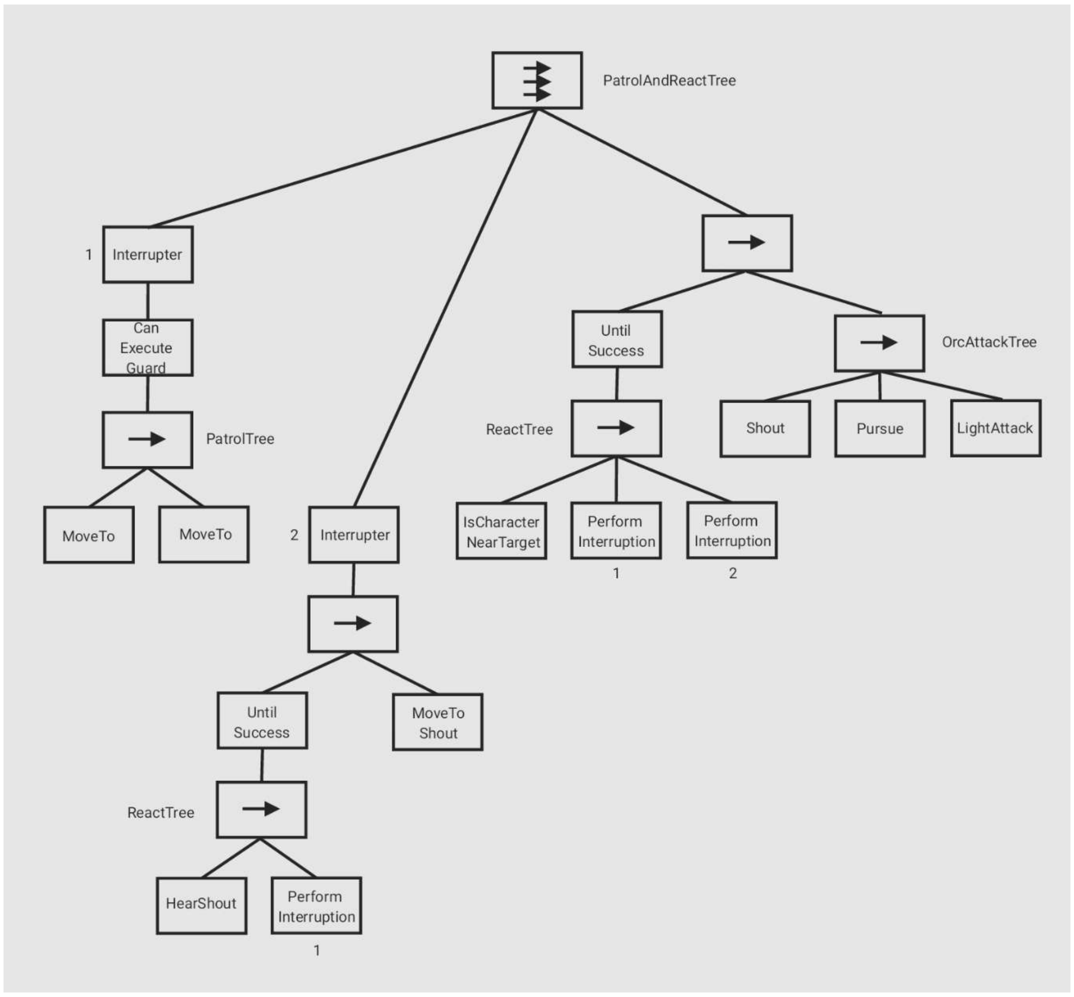
2 Orc's Behaviour Trees (Including Bonus Level)

2.1 Basic Idea

For the Orcs, we needed to implement a more complex Behaviour Tree that made the Orcs patrol around 2 points, and when one saw the player, they should alert the other Orcs by shouting and pursue the player. The Orcs, also, need to be listening for other Orcs' shouts, and when they hear one, they move to the position where that shout occured.
To do this, we designed a Behaviour Tree using Parallel Tasks and Interruptors, that allows the Orcs to stop the patrol movement whenever they hear a shout or see the player. Also if an Orc is moving towards a shout position and sees the player, he should pursue the player and not keep moving to the shout position.
Since the requirement for the Bonus Level was that Orcs would stop between patrol points to chase the player, we consider we have completed it.

2.2 Schema

The schema of the Behaviour Tree is the following:



3 GOB

3.1 Algorithm

GOB with an overall utility function is an algorithm that uses Goals and a discontentment function, which it tries to minimize in other to find the action that better fullfills the Goals.
The discontentment function used was

$$discontentment = \sum_{i=1}^{k} w_i * insistence_i, \text{for k Goals}$$

3.2 Data

Table 1: GOB performance

Processing time (ms) (of 1st decision)	Number of actions processed	Win Rate
0	27	100%

3.3 Initial Analysis

Looking at the data, we can see GOB is a very basic algorithm but shows very good performance if the goal's weights, change rates and initial insistences are well tuned.
This makes it dependent of the adjustment of the weigths according to the initial position, and only shows good win rate because there was an adjustment of the weights until Sir Uthgard won.

4 GOAP

4.1 Algorithm

This algorithm tries to use the idea of GOB, but applying it to sequences of actions, instead of using only one action. For this, we use a WorldModel representation and Depth-Limited search to find the best sequence.
For our specific case, we had to use some modifications, like pruning the actions' tree of branches that had actions leading to death. The algorithm chose actions like killing an enemy and recovering health later, which isn't allowed in the game, so this pruning was needed.

4.2 Data

Table 2: GOAP performance with depth 2 and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
0	356	10%

Table 3: GOAP performance with depth 2 and using WorldModel

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
0	356	10%

Table 4: GOAP performance with depth 3 and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
0.08	6138	0%

Table 5: GOAP performance with depth 3 and using WorldModel

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
0.08	6138	0%

Table 6: GOAP performance with depth 4 and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
1.2	97351	0%

Table 7: GOAP performance with depth 4 and using WorldModel

Processing time (ms) (of 1st decision)	**Number of actions processed**	**Win Rate**
1.3	97351	0%

4.3 Comparison

Comparing GOB and GOAP, we can see that GOB shows better performance, both in win rate and processing time.
Since GOAP computes sequences of actions and not only a single action, it's expected to take more processing time. The win rate being lower is less clear at first, but by looking at the data, we can see that the GOAP algorithm works better for less depth. This is probably due to GOAP looking too far ahead, and planning actions that he won't be able to execute because there are enemies in the way that he doesn't detect.

Since GOB is basically GOAP with depth 1, we can understand why GOB works better than GOAP. Also, GOAP tends to always put the LevelUp action as the last, probably due to bad use of the Goals in the WorldModel implementation. This results in the player facing the dragon still at level 1, and dying.

5 MCTS

5.1 Algorithm

MCTS is an algorithm that was created as an alternative to the Minimax algorithm. Its basic implementation combines breadth-first tree search with local search using random sampling, in order to reason whether a state leads to a winning situation.
It uses 4 steps: Selection, Expansion, Playout and Backpropagation. Selection serves to traverse the tree, selecting the optimal nodes, until a terminal node or a node that still has child nodes to expand is reached. Expand, will then create that child node. In our implementation, we expand only one child node per iteration. Then, we perform a Playout from this node. This consists in choosing random actions until we reach a terminal state, and then determine the result. In the end, all the nodes in the path are updated with this result, through Backpropagation.
For the testing we used the following configurations : this.MaxIterations = 1000, this.MaxIterationsPerFrame = 500, this.MaxPlayoutIterations = 10, this.PlayoutDepthLimit = 25.

5.2 Data

Table 8: MCTS performance with enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.7	1000	0%

Table 9: MCTS performance with enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.75	1000	0%

Table 10: MCTS performance without enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.68	1000	0%

Table 11: MCTS performance without enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.75	1000	0%

5.3 Comparison

By looking at the data, we can see that the basic implementation of MCTS doesn't get us very far. This is mostly due to the randomness of the algorithm, since every time we see an enemy or get close enough to a chest or potion, the algorithm runs again, which can give a new decision and waste all the time spent to reach the previous target.
Also, since the playout is random, every time it shows a different gameplay, but the chance of winning rappidly decreases.

6 MCTS with Biased Playout

6.1 Algorithm

By using an heurisitc to guide the Playout phase of MCTS, this algorithm achieves better results than the basic version.
Each action gets an H value assigned, based on their class and the current world state, and then we use Gibbs distribution to sample the actions and get a probability to choose them. Since we use Gibbs distribution, lower H values mean bigger

probabilities.

$$P(s, a_i) = \frac{e^{-h(s,a_i)}}{\sum_{j=1}^{A} e^{-h(s,a_i)}} \text{ (Gibbs distribution)}$$

For the testing we used the following configurations : this.MaxIterations = 1000, this.MaxIterationsPerFrame = 500, this.MaxPlayoutIterations = 3, this.PlayoutDepthLimit = 25.

6.2 Data

Table 12: MCTS with Biased Playout performance with enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.41	1000	0%

Table 13: MCTS with Biased Playout performance with enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.50	1000	0%

Table 14: MCTS with Biased Playout performance without enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.41	1000	0%

Table 15: MCTS with Biased Playout performance without enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.50	1000	0%

6.3 Comparison

Comparing this data with the previous ones, we can see it has better performance time-wise than the basic MCTS. This is due to having less Playout iterations, since using a bias "guides" the playout, and there shouldn't be much change if we do many playouts like in the random MCTS.
We can see that the win rate continues at 0, thus proving that just an heuristic doesn't help much.

7 MCTS with Limited Playout

7.1 Algorithm

This algorithm is a version of the basic MCTS, but limits the maximum depth of a Playout. This is so the algorithm spends less time performing playouts. However, a drawback is that we may not reach a terminal branch, and in that case, we can't determine the result. To address this issue, we create an heuristic function, which gives a score between 0 and 1 to a world state, that represents how good that state is or, in other words, how likely it is to lead to victory.
Even though this comparison was not necessary, we thought it would be good to include it since it shows progression relatively to the Biased version and will be useful to compare with the next algorithm as well.
For the testing we used the following configurations : this.MaxIterations = 1000, this.MaxIterationsPerFrame = 500, this.MaxPlayoutIterations = 3, this.PlayoutDepthLimit = 2.

7.2 Data

Table 16: MCTS with Limited Playout performance with enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.12	1000	30%

Table 17: MCTS with Limited Playout performance with enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.14	1000	30%

Table 18: MCTS with Limited Playout performance without enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.12	1000	30%

Table 19: MCTS with Limited Playout performance without enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.14	1000	30%

7.3 Comparison

We can see in the data that this algorithm has better performance then the previous versions of MCTS. This is due to the use of an heurisitc that allows us to have less depth in the playouts, and evaluate better the state we're in.
This is similar to what happened with GOB and GOAP, where planning too far ahead proves to be detrimental because Sir Uthgard gets intercepted by orcs and therefore needs to react in a more short-term manner.
For the testing we used the following configurations : this.MaxIterations = 1000, this.MaxIterationsPerFrame = 500, this.MaxPlayoutIterations = 3, this.PlayoutDepthLimit = 2.

8 MCTS with Biased Playout and Limited Playout

8.1 Algorithm

This algorithm uses Biased MCTS as a base, but combines it with small Playout depth and the heuristic from the previous algorithm, thus aiming to achieve even better results.

8.2 Data

Table 20: MCTS with Biased and Limited Playout performance with enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.15	1000	40%

Table 21: MCTS with Biased and Limited Playout performance with enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.19	1000	40%

Table 22: MCTS with Biased and Limited Playout performance without enemy detection and using WorldModelImproved

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.15	1000	50%

Table 23: MCTS with Biased and Limited Playout performance without enemy detection and using WorldModel

Processing time (ms) (of 1st decision)	**Number of iterations**	**Win Rate**
0.18	1000	50%

8.3 Comparison

We can see in the data that this algorithm has better performance then the previous versions of MCTS. This is due to the combination of the bias with the limit heuristic. Even though it has worse time performace than the random MCTS with Limited Playout, due to the calculation of the playout heuristic, it shows better win rate.
It still doesn't have a very good win rate. This might be due to a tradeoff between the time it takes to make another decision and the commitment to the decision the character takes. Since we don't use the tree computed on previous searches, each time we do another decision the tree resets and this can lead to another decision, thus wasting all the time spent in the commitment to the previous decision.
Even though lowering the time to update the tree would help detecting enemies in our path, if we do this too often, the character just wouldn't finish any action and would still lose to time, like in the basic MCTS.
We can also look at the data and see that using enemy detection lowers the win rate. This is due to the problem spoken above, the player resets the tree and loses to time.
Note, however, that while in the basic MCTS the character would change decision every time the algorithm executed, in this version he sticks with the same decision more often, since the best action being determined is more likely to be the same between executions due to both heuristics.
Also, the bias heuristic and the scoring for non-terminal world states could be better tuned to work in tandum.

9 Conclusions

Analysing all algorithms we can acess that GOB is the one that shows better performance, both in win rate and processing time.
Even though MCTS with Limited and Biased playout should be better theoretically, since GOB uses goals, and their weights were well adjusted to fit the specific layout of the level, it shows a better win rate.
The biggest problem with MCTS and its variants is that sometimes we can't detect enemies in our path, and can only detect them close to the target of our action. Even though we can recompute the tree when we are close to an enemy, this process makes Sir Uthgard waste all the time he already spent on the previous action if he switches actions. This is also not optimal, since we die not by fighting enemies, but by reaching the time limit.
One optimization that we think would help is to use the previous computed tree in new searches and keep expanding it, but account for changes in enemy positions. This would be helpful especially in cases where we are interrupted by enemies. This would make the process faster, since we wouldn't compute as much actions as before.
It would also be a good improvement if in the WorldModel we took account for the movement of the enemies but this would be much harder to implement, since enemies can change their behaviour when seeing the player, so the first optimization might be better. Relatively to the world state representation, which we didn't mention before, but the data relative to it is in each section, we can see that the array implementation does outperform the dictionary implementation more as the depth increases. In GOAP, this is hardly noticeable but in the playouts for MCTS it becomes more relevant. So, it does offer a slight improvement in computational times.