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 5 different algorithms: Goal Oriented Behaviour (GOB), Goal Oriented Action Planning (GOAP), Monte Carlo Search Tree (MCTS), MCTS with Biased Playground, and MCTS with Biased Playground and Limited Playout.

2 Orc's Behaviour Trees

2.1 Basic Idea

3 GOB

3.1 Algorithm

GOB with an overall utility function is an algorithm that uses Goals and a discontentment function, which he tries to minimize in other to fullfill the Goals.
The discontentment function used was

$$discontentment = \sum_{i=1}^{k} w_i * insistence_i, forkGoals$$

3.2 Data

Table 1: GOB performance

Processing time (of 1st decision)	Number of iterations	Win Rate
1	1	1

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.

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 WorldState 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 is'nt allowed on the game, so this pruning was needed.

4.2 Data

Table 2: GOB performance

Processing time (of 1st decision)	Number of iterations	Win Rate
1	1	1

4.3 Comparison

Comparing GOB and GOAP, we can see that GOB shows better performance, both in win rate and processing time. Add more justification.

5 MCTS

5.1 Algorithm

MCTS is an algorithm that was created as an alternative to the Minimax algorithm. It's basic implementation combines breadth-first tree search with local search using random sampling, in order to have data about if a state leads or not to a winning situation.
It uses 4 steps: Selection, Expansion, Playout and Backpropagation. Add more description.

5.2 Data (Next Page)

Table 3: GOB performance

Processing time (of 1st decision)	**Number of iterations**	**Win Rate**
1	1	1

5.3 Comparison

By looking at the data, we can see that the basic implementation of MCTS does'nt lead us really far. This is mostly due to the randomness of the aglorithm, since every time we see an enemy or get close enough to a chest or potion, the algorithm runs again, giving a new decision and wasting all the time spent to reach the previous target. Add more conclusions.

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 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.

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

6.2 Data

Table 4: GOB performance

Processing time (of 1st decision)	**Number of iterations**	**Win Rate**
1	1	1

6.3 Comparison

Comparing this data with the previous ones, we can see that this is by far the best optimization in terms of runtime. This is due to the use of bounding boxes, that shorten the amounts of nodes we process, and thus the amount of calls to add, remove and search in the open and closed set.

7 MCTS with Biased Playout and Limited Playout

7.1 Algorithm

Add description.

7.2 Data

Table 5: GOB performance

Processing time (of 1st decision)	Number of iterations	Win Rate
1	1	1

7.3 Comparison

We can see in the data that the Dead-End heuristic is a good heuristic, since it improves the time of the Search function, which is the most time consuming function in the A* algorithm, even though it has a big initialization cost calculating the DFS in the room graph. But it's still not a big optimization, since it shows results worse than NodeArray A*. It's also worth noting that the giant grid is not very fit for this algorithm as it creates many interconnecting clusters, which causes finding all the paths very costly in some cases. With a clustering algorithm better fit for this map the results could be better.

8 Conclusions

Analysing all algorithms we can acess that A* by itself is already a good algorithm, but it's optimizations can make it much faster, without compromising finding the best path.
Implementing better data structures that significantly reduce time spend on commonly used operations, like in the case of the Array Node A*, gave good results, but we still explored many nodes which were not part of the best path.
Then, we saw that adding preprocessing to the algorithm can improve it's runtime performance a lot, although it can take some time to perform it, especially on bigger maps, with many nodes. This is the case with goal bouding and the calculation of the Dead-End heurisitic.
The combination of the search efficiency of the Array Node A* with bouding boxes, which significantly reduced the nodes explored in directions other than the desired one, resolved both main issues with the basic A* algorithm. Due to this, goal bouding proved to be the best algorithm.