### Level 1 – Behavior Trees

For the first level we were tasked with the implementation of behavior tree for the Orcs that included the Patrol task. For this purpose, we added the option on the *LevelCreator* script to include patrol points for the characters (represented by the letter *p*). This made it possible to generate patrol points for different maps without having to add empty objects directly on the editor. Since we had implemented the *Selector* composite task, we derived the *OrcPatrolTree* from that class. As the guide pointed out, it was impossible, at first, to interrupt the *Patrol* task when the player was spotted. To counter this issue, we changed the order of the tasks within the *OrcPatrolTree.* This solved the problem, given that the *Selector* runs the first successful task. By putting the *Patrol* task after *Pursue* task, the Orcs started behaving correctly.

Uma imagem com texto

Descrição gerada automaticamenteFor the shout task (class *HearOrcShout*), we added an audio source to the Orcs for the auditive part and a canvas with text in world space for the visual effect. If the Orcs hear a shout they’ll move to the position where the shout came from. If the Orc stops shouting (either because it stopped seeing the player or was killed), the Orcs will resume their “normal” behavior. This led to the problem where the Orcs would pursue the player but not attack it, since the *LightAttack* task was ordered after all other actions. As such, it was placed first on the list and modified to consider whether the target was in range or not, which solved the problem.

Figure 1. Visual representation of the Pursue task with shout

### Level 2 – GOB and GOAP

For the GOB algorithm, the *Rest* action was a challenge, since the player would continuously choose to rest to recover HP instead of achieving the end goal (even with the Sleeping NPCs option enabled). This happened because the Survive goal would achieve bigger numbers than the Be Quick goal. To counter this, and to prevent the player from never healing, we limited the value of the Be Quick goal to half of the player’s maximum HP. With this, the AI would now heal until the relative safety of over half HP and then continue to play, not being stuck resting until the full health was recovered. We also had to change the value of the Gain Level goal to account for this.

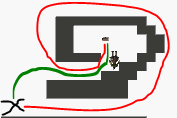
For the GOAP algorithm we attempted to exclude actions that would lead to losses by attributing a bigger value to world models that resulted in exceeded time limit or death. This will lead to the algorithm sometimes being unable to decide the next action, stating that all of them will exceed the time limit. This happens as a result of our pessimist approach to calculating the duration of an action. Sometimes the player will run out of time because the *NavMeshAgent* will not choose the shortest path. For example, in figure 2 the optimal path from X to the chest is displayed in green and the path chosen by Unity is displayed in red. This problem was very visible with the MCTS, reason why we had to recalculate the timing of actions. We decided it was still better to keep the pessimist approach, since if not kept the AI would continuously pick actions that could never result in a win within the time limit.

Figure 2. Difference between optimal and chosen paths

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Time Until Win (Average) | Processing Time Per Run (Average) | Best Discontentment (Average) | Total Actions Taken (Average) | Win Rate (Average) |
| GOB (sleeping NPCs) | 134 s | Virtually 0 | 11,08 | 5 | 100% |
| GOB (stochastic world) | 140 s\* | Virtually 0 | 21,01\* | 13\* | 50% |
| GOB (non-stochastic) | 186 s | Virtually 0 | 21,01 | 19 | 100% |
| GOAP (sleeping NPCs) | 132 s | 0.01 s | 0.32 | 6 | 100% |
| GOAP (stochastic world) | -- | 0.03 s | -- | -- | 0% |
| GOB (non-stochastic) | 142 s | 0.03 s | 0.96 | 12 | 20% |

\*Statistics only apply in case of victory. For the stochastic world the wins depend heavily on whether the enemy damage is closer to the simple damage or greater than it. For example, some orcs will one shot the player at 10 HP.

The differences in processing time in the GOAP algorithm ties to whether it must account for the actions of other players (enemies). The win rate is also heavily influenced by this. The duration of actions would often be affected by being intercepted by an enemy while attempting to execute it. The low value for the max depth allowed the algorithm to still be fast but decreased its effectiveness. Changing the max depth to 5, we notice a visible increase in processing time (averaging 5.6 s per run) but better results.

The use of discontentment by the GOAP algorithm would sometimes lead to choosing “dumb” actions (i.e., attack an enemy to gain XP instead of picking up the last chest). This did not occur with GOB, as this algorithm focuses on the top goal, which we were able to manipulate as already described.

### Level 3 and Level 4 – Sir Uthgard’s Actions and MCTS

**Sword Attack –** For this already implemented action, we limited the execution to only when NPCs aren’t sleeping, to avoid the dumb move of attacking harmless enemies. We also realized the AI would sometimes decide to get more XP even when it was possible to level up, so we corrected this situation by changing the *GetGoalChange* method to return 0 if leveling up was possible. Furthermore, the *CanExecute* method verifies if the player has enough HP + ShieldHP to sustain a basic attack from the enemy.

**Divine Smite –** Aside from the need for mana and the unchanged HP, we followed the same logic as for *Sword Attack*. The heuristic value is slightly better so that the player will prefer to DivineSmite a skeleton rather than SwordAttack it.

**Shield of Faith –** For this action we had to treat the ShieldHP as part of the HP, which led to the *GetGoalChange* for the Survive goal returning the amount of ShieldHP the action would restore. The heuristic varies with the HP and is especially good when health is below half. This will allow the player to avoid using health potions if this action is available.

**Pick Up Chest –** To guarantee the AI would want to choose this action and complete the game, we modified the *GetGoalChange* to return a negative value if the goal was to be quick. As for the heuristic, it accounts for how close the chest is so that the player won’t decide to ignore closer chests.

**Get Health Potion –** The heuristic for this action prioritizes low health, as it is a waste to use a potion to recover less than 5 health.

**Get Mana Potion –** The *GetGoalChange* was tricky. We decided that it would have to consider possible future actions that depend on mana. So, if the player’s mana is lower than what’s needed for a certain action, the goal change will be like what that action would have. If the future action can be performed, the function returns 0. The heuristic is good for when Mana is below a certain value and the NPCs aren’t sleeping. This is so the other mana-consuming valuable actions will be available.

**Level Up –** We recalculated the goal change for Gain Level goal to return a minimal value whenever it was possible to execute the action, as to consume as little XP to level up as possible. With decreased the heuristic value to account for other actions’ heuristics.

**Rest –** The heuristic is designed to consider if there is a quicker option to gain HP.

**Teleport –** The heuristic for this action is proportional to the distance to the initial position.

The *MCTSBiasedPlayout* class implements a method to sort the executable actions according to their heuristic value. It then chooses one at random, mostly from the first positions of the list, to optimize the chance of picking a good action.

### Secret Level 1 – Optimizing World State Representation

For the optimization of the World State Representation, we opted to create an object array in WorldState class with 24 positions and stored all properties, consumables and enemy states there. With this optimization, instead of going recursively through each parent to find a property, we can access the position corresponding to it. This proves very useful, because it avoids unnecessary and successive calls to *GetProperty* (through recursion) and, instead, only needs one call, speeding up the process.

### Secret Level 2 – Limited Playout MCTS

Limiting the depth of playout allows the algorithm to “cut” the search without having to reach a terminal state. To decide the value of a non-terminal state, we divide the number of caught chests by the total play time of the state minus the amount of remaining HP, to penalize states with low HP.

### Secret Level 3 – Comparison of MCTS variants

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | Total Processing Time Per Run (Average) | Total Actions Logged (Average) | Time Until Win – Sleeping (Average) | Win Rate – Sleeping (Average) | Win Rate – Stochastic (Average) | Win Rate – Non Stochastic (Average) |
| MCTS | 0.089 s | 18 | 132 s | 100% | 0% | 0% |
| MCTS with UCT | 0.097 s | 21 | 150 s | 100% | 0% | 0% |
| MCTS Biased Playout | 0.079 s | 19 | 134 s | 100% | 0% | 0% |
| MCTS Biased Playout with UCT | 0.053 s | 17 | 160 s | 100% | 0% | 0% |
| MCTS Biased Playout + Limited | 0.064 s | 7 | 144 s | 100% | 0% | 0% |
| MCTS Biased Playout with UCT + Limited | 0.061 s | 8 | 174 s | 100% | 0% | 0% |

The MCTS implementation doesn’t consider that the enemies can pursue and attack the player, only checking for the player final position upon completing an action to calculate the impact of the enemy’s turn. The random factor in the MCTs causes the algorithm to sometimes disregard good actions, as they aren’t picked within the number of specified playouts. This also led to sudden shifts in actions, as shown in the Total Actions Logged column.

Although the Biased algorithm had the best overall processing time per run, the basic MCTS algorithm led to a faster victory (on average). However, the basic algorithm had constant action shifts, which led to “erratic” behavior of the player. Given that all the algorithms are functional for the Sleeping NPCs option, the MCTS Biased Playout + Limited would be a good option, since it eliminates the shift in actions and has a better processing time per run, as well as a fast time until win.

### Secret Level 4 – Additional Optimization

We chose to implement the “Check generated actions for immediate win/loss” present in the slides. It was important since it allowed the player to avoid scenarios like being almost at the end (missing just one chest) but choosing a different action instead of picking up the last chest and losing by time limit exceeded. It also altered the runtime per frame from 0.089 s (basic MCTS with 10 playouts) to 0.131 s. This optimization led to erratic behavior on the MCTS Biased Playout + Limited.

### Conclusion

The more predictions we tried to make about the state of the world with progressively more complex algorithms, the lower the Win Rate for non-sleeping NPCs worlds would get. Stochastic worlds rely on luck for the player to win, as the value of the attacks varies greatly. However, for non-stochastic worlds we were able to adapt the GOB algorithm to a satisfying degree of efficiency. As it makes no assumptions about the future, the algorithm’s choice to solve in-the-moment crisis made it the perfect match for our dungeon.