IAJ - Decision Making

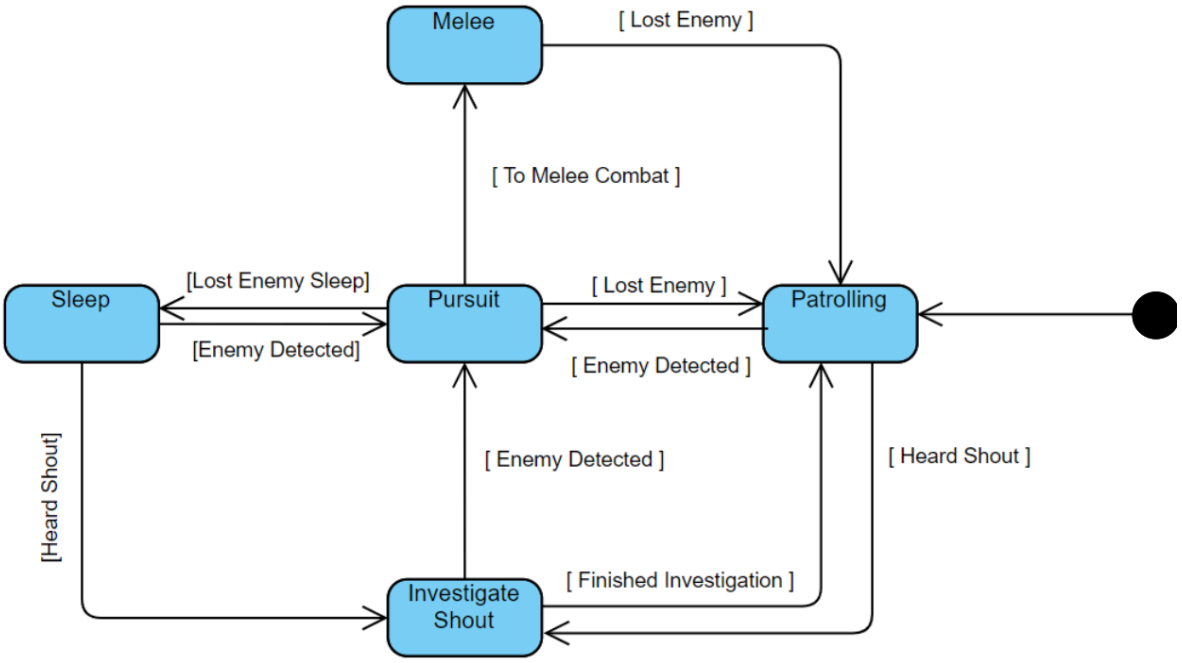
Grupo: 2

Aluno(s): João Azevedo (96877) - João Costa (99985) - Óscar Ferrão (113182)

State Machine

In our implementation, orc enemies follow a state machine-based behavior pattern the orcs patrol between two designated points within their environment. If an orc spots the player, it will shout to alert nearby orcs, prompting them to move toward the shout's location.

While on their way to the shout location, if any orc spots the player, it will immediately pursue the player. Once the player is no longer in sight or the chase concludes, the orc will return to its original patrol route. Look at the diagram of our state machine below (Picture 1).



Picture 1 - Diagram of the State Machine behavior for the monster.

World State Representation

The original World State class used a recursive dictionary structure, which offered the advantage of lower memory consumption. However, this structure slowed down the algorithms that interact with the World State, as they often needed to iterate multiple times to access specific properties. To address this issue, we introduced an alternative World State representation utilizing an auxiliary class called "Properties."

In the new representation, each World State directly includes every property. This enables algorithms to access required properties without repeated iterations, significantly improving performance. While this approach increases memory usage,

the performance gains outweigh the additional memory cost due to the relatively small number of properties involved.

For comparison, we used the Unity Profiler to evaluate the performance of both World State representations. The testing scenario involved measuring their behavior with Goal-Oriented Action Planning (GOAP) at a depth of 4 during the first action selection phase. You can see the results of this comparison in Tables 1 and 2, which show the performance with the dictionary-based and fixed World State representations, respectively.

Table 1 - Table with profiler statistics of Dictionary world state

	Calls	GC Alloc	Time ms	Self ms
GenerateChildWorldModel	50000	27.8 MB	1587.64	16.85
SwordAttack.GetProperty	50490	159.4 KB	196.97	17.63
SwordAttack.SetProperty	66952	4.2 MB	88.75	15.94
PickUpChest.GetProperty	14003	97.3 KB	63.41	4.86
PickUpChest.SetProperty	42009	3.5 MB	61.79	9.93
GetHealthPotion.GetProperty	4992	89.2 KB	26.64	1.74
GetHealthPotion.SetProperty	9984	1.2 MB	17.63	2.42

Table 2 - Table with profiler statistics of Fixed world state

	Calls	GC Alloc	Time ms	Self ms
GenerateChildWorldModel	50000	42.5 MB	1938.42	19.07
SwordAttack.GetProperty	50490	0	39.83	8.74
SwordAttack.SetProperty	66952	0	47.77	12.14
PickUpChest.GetProperty	14003	0	12.44	2.72
PickUpChest.SetProperty	42009	0	31.50	8.22
GetHealthPotion.GetProperty	4992	0	3.76	0.83
GetHealthPotion.SetProperty	9984	0	6.02	1.72

The comparison reveals that the GenerateChildWorldModel function is less efficient in terms of space and time with the fixed World State. This is due to the need to store all properties in each child instance, increasing memory usage and initialization time. In contrast, the recursive dictionary representation allocates memory dynamically, which is more efficient for this function.

However, the fixed World State significantly improves GetProperty and SetProperty operations by storing properties directly in each instance, allowing faster, direct access

without the recursive lookups required by the dictionary-based model. As a result, the fixed World State reduces the overall time needed to calculate the first action from 10.3 seconds with the dictionary-based model to 6.7 seconds.

Despite these improvements, neither approach is fully optimal because both are subclasses of WorldModel. This inheritance structure can introduce additional overhead, as methods and properties may need to traverse the superclass hierarchy, adding latency. Further refinements could potentially improve performance by reducing reliance on the superclass or restructuring the inheritance to allow for more direct and efficient access paths within the WorldModel hierarchy.

Comparison of Decision-Making algorithms

In our project, we tested a variety of algorithms to evaluate their performance when guiding Sir Uthgard through the game. Each algorithm was tested ten times with all available actions, and the outcomes were measured across multiple metrics to determine their effectiveness. The key performance indicators were tracked: Win Rate, Time, Processing Time per Action Number of Actions, Gold Accumulation, and Algorithm Depth.

Table 3 - Table with the comparison of the different algorithms

	Win rate	Time	Proc. Time per Action	Num. Actions	Gold	Depth
GOB	100%	112	0.01	8	25	n/a
GOAP	30%	130	0.01	19.5	21.5	3
GOAP 160s	60%	149.6	0.01	20.5	23	3
MCTS w/ shout	0%	150	14.26	9	15	n/a
MCTS + Biased w/ shout	0%	100	9.82	8	10	n/a
MCTS + Biased w/o shout	100%	154	8.28	12	25	n/a
MCTS + Biased + Limited w/ shout	100%	140	4.67	12	25	3
MCTS + Biased + Limited w/o shout	100%	124	4.53	10	25	3

In our **GOB**, we identified a heuristic strategy that consistently leads to victory. This strategy emphasizes survival and leveling up, particularly during the early stages when Sir Uthgard, the player character, is at level one. Moreover, we assigned significant value to collecting the final piece of gold. Upon reaching 20 gold pieces, Sir Uthgard is programmed to prioritize

securing the last piece, essential to achieving victory. The goals are weighted as follows: Survival (150), Leveling Up (100), Speed (25), and Rich(200). This prioritization ensures a balanced yet effective approach, enhancing Sir Uthgard's chances of success in the game.

In our **GOAP** implementation, we simulate 3 actions into the future and calculate the discontentment to decide on the best sequence of 3 actions, but this has one problem, that being the order of the actions doesn't affect how the discontentment changes so if Sir Uthgard really needs to heal using this algorithm it doesn't matter if we do it now or in the next 2 actions, and since it was heavily prioritizing collecting gold, it always decided to just heal later, causing it to find enemies during the way and dying. To force him into healing we decided to change the heights to the following: Survival (300), Leveling Up (100), Speed (20), and Rich(0), this change allowed for some wins but since we knew we couldn't prioritize gold it was taking just a bit too long seen in the win rate of **GOAP** if it had 160s instead, doubling the win rate from 30% to 60%.

Lastly, we evaluated three **MCTS** configurations using identical parameters: a maximum of 5000 iterations, 5000 iterations per frame, and 10 playouts per move. A key function, CalculateNextPlayer, was employed to detect nearby enemies, enabling the algorithm to anticipate when an enemy orc might be close to Sir Uthgard's target. However, the game's shout mechanic, attracts additional orcs, because they stop being close to the target.

1. Simple MCTS

This variant has two main issues. First, it does not account for time constraints, which means that the simulation can sometimes run out of the time frame. Additionally, when an orc is called by the shout and travels in the middle of Sir Uthgard's path, the player doesn't detect it, potentially leading to fatal outcomes. This is reflected in the table, where the **MCTS without shout** consistently reached 150 in time, while the version with shout often died before reaching the time limit.

2. MCTS + Biased

In the same way as normal **MCTS,** it can also encounter problems with the shout mechanic. However, if we use the shout-less version Sir Uthgard managed to win every single time very close **after** the time.

3. MCTS + Biased + Limited

In the final algorithm, we set a depth of 3. This version not only evaluates end states (winning or losing) but also intermediate states, allowing for the development of a heuristic that assigns weights to factors like time, money, health, and level. These weights help determine the overall world state value. As expected, this version consistently achieved victory within the anticipated time.

In conclusion, **MCTS + Biased + Limited** stands out as the main algorithm of choice due to its strategic depth, adaptability, and robust handling of the shout mechanic. This variant consistently enables Sir Uthgard to navigate complex scenarios effectively and win reliably. While **GOB** performed well in this specific setup, its reliance on heuristics makes it less scalable and adaptable to larger and more chaotic environments. Therefore, **MCTS + Biased + Limited** is the most effective option for consistently guiding Sir Uthgard to victory, particularly as game complexity increases.

Formations

In the game manager's "Awake" function, all formation types are initialized. A different grid setup includes three additional orcs: Orc3, Orc4, and Orc5, with Orc5 designated as the leader of this squad. In the line formation, the positions of the following orcs are determined using an offset distance from the squad leader. In the triangle formation, the positions of the orcs are calculated by adjusting the orientation of the leader by either +60 or -60 degrees, forming a triangular arrangement with the three orcs.

Although the position calculations for these formations are straightforward, it can be challenging to observe the squad's behavior during gameplay. This is likely due to the orcs' collision size being too large, as well as the high knockback effect upon collision. Additionally, the orc's shout mechanic can disrupt the formations, as it prompts nearby orcs to attack the player, causing the squad to break the formation.

Conclusion

This project provided an excellent opportunity to apply various decision-making algorithms and explore modern techniques, including Monte Carlo Tree Search (MCTS), which is used in advanced projects like AlphaZero. One major challenge was balancing the game's difficulty, particularly due to widely spaced orc patrols that initially made gameplay too hard. Additionally, developing effective heuristics for each algorithm required considerable fine-tuning. We also encountered some issues in Unity, such as items or enemies drifting unintentionally because of this we slightly increased the range of the detection of an enemy from 10 to 15 to try to minimize this problem.

During testing, we also discovered a bug: when leveling up, players would lose all of their XP instead of just the amount required for the level-up. Additionally, we found that the CalculateNextPlayer function assumes that attacks occur only after picking up a target, meaning that if an enemy is positioned between the player and the target, the player might be unexpectedly attacked and killed—an outcome that conflicts with our calculations.

Bonus

For the bonus, we applied our improvements to both GOB and GOAP. Since we had already achieved a 100% win rate with GOB, these enhancements didn't significantly impact its performance. Therefore, we focused primarily on the improvements to GOAP.

As part of the bonus, we implemented a reaction mechanism to the enemy flag. This allows Sir Uthgard to reassess his situation and select a more advantageous action when encountering an enemy. To influence his decision-making, we adjusted the system so that actions enhancing his survival considerably reduce his discontentment, especially if the nearby enemy poses a lethal threat.

This change more than doubled the previous win rate, allowing Sir Uthgard to win most encounters, as demonstrated in the table below. However, the issue of the extended time required to achieve victory remains.

GOAP	30%	130	0.01	19.5	21.5	3
GOAP w/ react to enemy	70%	140	0.01	27.6	23.5	3
GOAP w/ react to	90%	144	0.01	28.6	24.5	3

enemy & 160s						