Study of decision making and path planning algorithms with an AI game

Anshul Jethvani
Dept. of Computer Science
North Carolina State University
ajethva@ncsu.edu

Raj Kumar Shrivastava
Dept. of Computer Science
North Carolina State University
rkshriva@ncsu.edu

Tanay Agrawal
Dept. of Computer Science
North Carolina State University
tagrawa3@ncsu.edu

Abstract

In this project, we present a game equipped with artificial intelligence involving a human player and some AI bots. The movement of the bots would be planned and automated through AI algorithms. This is a zero-sum game wherein the aim of the human player is to steal the treasure from the haunted house and bring it back to the safe-house. The treasure is initially guarded by one or more AI bots. The computer bots use decision making algorithm to decide among various actions like chasing the human player to kill it, defend the treasure or just wander aimlessly depending upon the state of the game. The decisions are supported by path-finding algorithms from their position to the prospective destinations. There are some static and some randomly moving obstacles, and random power-ups in the game. The aim of the project is to bring out the impact of AI and study the comparison between different path-finding and decision-making algorithms of the AI bots.

Introduction

Artificial intelligence is speculated to be present in the core of a lot of technologies in the near future. It has become quite popular not only in the industry for predictive analysis or other business use cases but it is also being massively incorporated in game development to give a real world experience to the user. Developing games with AI capabilities provides the person playing the game a much richer and interactive outlook. The usage of AI in game development has given a whole new dimension to multi-player gaming world. By incorporating AI, game developers have increased the competitiveness of the computer bots, consequently making it difficult for the player to win against these computer bots. One of the upside is that it has increased the user engagement. The users tends to play more with higher ambitions to win against the bot enhancing the overall gaming experience.

In this paper, we describe and present the evaluations of several path-finding and decision making algorithms which are an essential part of the game development. The algorithms we would be studying in this paper are Recursive Best First Search algorithm, A^* (A star) algorithm for path-finding. For decision-making, we have explored the algorithms of Behavior Trees and Goal Oriented Action Planning (GOAP) using Iterative deepening A^* algorithms.

The evaluation of algorithms is done on various parameters. The main parameters are empirical time complexity and space complexity. Theoretically, these parameters form the base for evaluation of any computer science algorithm but it is important to test them and see how they turn out to be in a game environment. Apart from these, the shortest distance computed by these algorithms in real time should also one of the evaluation parameter. And the same goes for the decision making algorithms.

Background

One of the most important challenge in developing a smooth-running game is implementing the right algorithms for an AI-based game. This can often be a deciding factor for success or failure of a game. A bad algorithm could use up more resources than necessary. It could also lead to bad user experience. The best of Game AI algorithms based on Deep Learning or Reinforcement Learning take too much time and resources while training and are also seldom difficult to get the hyper parameters tuned appropriately. Through the course of this project, we aim to study and analyze various algorithms for path planning and decision making and find out which works out best to adapt to the performance of the player while at the same time is computationally efficient.

With this project, we aim to study and analyze various path finding and decision making algorithms. The research will help to distinguish the use cases and trade-offs between various algorithms in terms of execution speed, memory requirements and heuristics.

From the perspective of its research importance, it is important to find the best algorithm with properly adjusted parameters. Analyzing different algorithms would make sure we make most out of the resources while at the same time creating a better user experience. A deliberately weak AI might be enjoyable for human players whereas a strong AI would make game humanly non-playable. Thus different parameters like evaluation function, heuristic search, machine learning, automatic knowledge generation, mathematical morphology, cognitive science, etc. can be used to tune the AI algorithms affecting how enjoyable the gaming experience is for the player.

The work in this project brings out the trade off between different algorithms than can be performed for the same task of path-finding or decision-making.

Literature Review

Algorithms for path-finding and decision-making have been significantly explored in the past. A major part of our review was focused on (Millington and Funge 2009). Most of our evaluations have been adapted from the implementation of algorithms like A^* algorithm, Behavior trees and Goal-oriented action planning from this book. In the book, Intelligent Search Strategies for Computer Problem Solving(Cui and Shi 2011) comprehensively describes a number of search strategies and path finding algorithms. It discusses the performance of the algorithms depending on the heuristics involved. Application of these algorithms in game design in this book ensures a good grasp of the algorithms and their usage in games.

In (Russell and Norvig 2002), the authors have elaborately explained about path finding, decision making and planning algorithms in modern era. The chapter on path finding algorithms where he talks about the informed search strategies is the place where we get all the theoretical and mathematical knowledge on how to compare these algorithms. This book was helpful in performing an extensive study on Recursive best first search algorithm and A-star search algorithm. The chapter that describes Decision making algorithms provides vital information on how to implement these algorithms in a game like environment. The implementation details in the book made it easier to compare the experimental results with the known theoretical information about the algorithms we have studied in this work.

(Cui and Shi 2011) has efficiently described the A^* algorithm in terms of game development. The paper later shows its application in path-finding involved in games like Age of Empires, Civilization V and Counter Strike. The paper talks about how the maps are represented in these games.

(Sinthamrongruk, Mahakitpaisarn, and Manopiniwes 2013), published their results when they compared the A^* Path finding algorithm by running it on IOS and Android Operating Systems with Waypoint Navigator Algorithm based on the time it takes to search for new path i.e. the time complexity. The results they published in the paper concludes that A^* was better than Waypoint algorithm. The results also stated that the space taken for computation among these two algorithms were almost same i.e. there was no specific difference in the amount of memory utilised in computation for these two algorithms.

The work in the paper (Goyal et al. 2014) compares two widely known path finding algorithms i.e. A^* and Dijkstra's Algorithm for finding the shortest path between a source and destination. The paper concludes that the running time or the execution time of A^* is nearly half to that of the Dijkstra's algorithm i.e. the A^* algorithm solves the problem at hand in nearly half the time it takes for Dijkstra's algorithm to solve the same problem.

In the paper (Magzhan and Jani 2013), gives us an analysis on a good number of path finding algorithms back in 2013. They took only Shortest Path algorithm for performing their analysis. The algorithms taken into consideration were Dijkstra's algorithm, Bellman-Ford Algorithm, Floyd-Warshall Algorithm and Genetic Algorithm. In the results, they emphasize on the fact that Genetic algorithms provided better results than the rest of the algorithms. Moreover, genetic algorithms provided the optimal solution which was not the case with the other algorithms.

(Vamja 2017), published their analysis on the commonly used shortest path algorithm used in game development. Her research included study on algorithms like Dijkstra's algorithm ,A^* algorithm and IDA^* algorithm. The results gave us a comparative analysis between space complexity, etc. Their research lacked the comparative results of these algorithms on one of the most important factor which is execution time.

(Bhadoria and Singh 2014), in their research optimized the Angular A^* algorithm for searching the Global Path based on evaluation of Neighbor Nodes. The results published shows the analysis was performed on partially known environment situations. Finally, she concludes that the optimal path planned by the A^* algorithm using the heuristic approach in this paper is better than the Dijkstra's algorithm in robotic path finding in games developed after year 2000.

(Li, Harms, and Holte 2007), published new algorithm for shortest path finding algorithms. He came up with a new algorithms known as "Fast Exact Multi Constraint Algorithms", which were shorted path algorithms. He also proposed two new algorithms i.e. the A^* & Fringe MCSP (i.e. multi constraint shortest path) and fringe MCSP based on A^* . The results he published in research shows that the MCSP algorithm has better performance in both the scenarios mentioned above. These algorithms were used to solved NP hard problems in his research.

(Littman 1996) talks extensively about decision making. It explains the variations and presents applications of Reinforcement learning and Markov Decision Process in game development. (Ogren 2012) talks about behavior trees. Its application in the field of robotics has been demonstrated in this work.

Goal-oriented action planning and Hierarchical Task Network representations has been explained in (Hoang, Lee-Urban, and Muñoz-Avila 2005). The paper presents an interesting case study involving Unreal Tournament bots for planning tasks.

The work in (Korf 1993) presents a good analysis of the Recursive Best First Search Algorithm. It presents a good comparison between the algorithms like Depth-First Iterative Deepening, Iterative-Deepening A^*, Recursive Best First Search and Simple Recursive Best-First Search algorithms.

Methodology

In this project, we evaluate the path-finding and decision making algorithms by implementing them from scratch for a game. We start with explaining the game dynamics and design and then move to explain path finding and decision making algorithms. Overall, the human player and the bots play against each other. The aim of the player (human) is to reach safely at haunted house to steal the treasure and bring it back safely to the safe house which is their starting point. The goal of the bots can be divided in two parts, protect the treasure from human player and later if the player steals the

treasure, prevent the human player to go back to the safe house. The decision making capability of the bots is tested in the later stage of the game where the bots are required to make optimal choices suitable at any given scenario relating to the state of the game, position of the human player and some other utilities. The following subsections provide insights on our approach and the algorithms which be utilized in the project.

Game Design

The game has been designed in a way that would extensively make use of path-finding, decision making and path planning algorithms. This facilitated us to clearly evaluate the performance of different algorithms for any given task.

Q1	Q2
Q3	Q4

Table 1: Screen Divided in Quadrants

For evaluation purpose, we have assumed the game canvas to be divided into four quadrants as shown in the table above. The top-left part of the first quadrant (Q1) hosts the safe house, top-right part of screen is the second quadrant (Q2), bottom-left part is the third quadrant (Q3), and the bottom-right part of the fourth quadrant (Q4) hosts the haunted house. These quadrants help us categorize the vicinity of the player to the key locations.

When the game begins with a (human) player at the safe house, there are two guards (enemy bots) protecting the treasure. The human player moves manually with direction arrow keys. The challenge for human player is to prevent themselves from colliding with the guards or dangerous obstacles and traps that come in the way. If the enemy bots catch the player, the player loses the game. If the player successfully brings the treasure back, the player wins the game. The world map also begins with four deadly and two not harmful (non-deadly) static danger spots. These numbers may increase or decrease subject based on whether certain special power ups as secured by the player or the bots as the game progresses.

Currently, we are performing all our evaluations by implementing the algorithms for automating the enemy bots and deadly obstacles, generating traps and power-ups at strategically random locations. These algorithms are categorized as follows:

Path-finding

We evaluate the performance of path-finding algorithms by varying the level of obstacles in the map. This ranges from no obstacles at all to complicated obstacle shapes which effectively tests the path-finding algorithms based on time consumed and the memory consumed in finding the path.

The game would use several algorithms, namely, A-star algorithm and Recursive Best First Search algorithm. We evaluate the performance of these algorithms in different scenarios and based on different parameters. From the perspective of game-play, these variations would also introduce

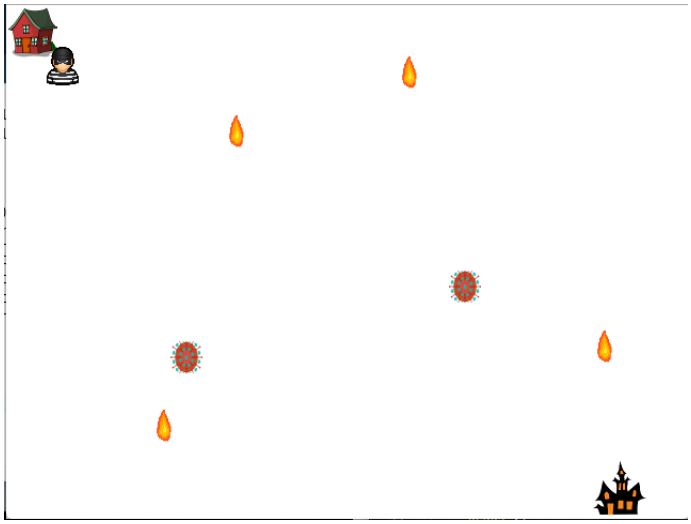

Figure 1: Simple map without obstacles

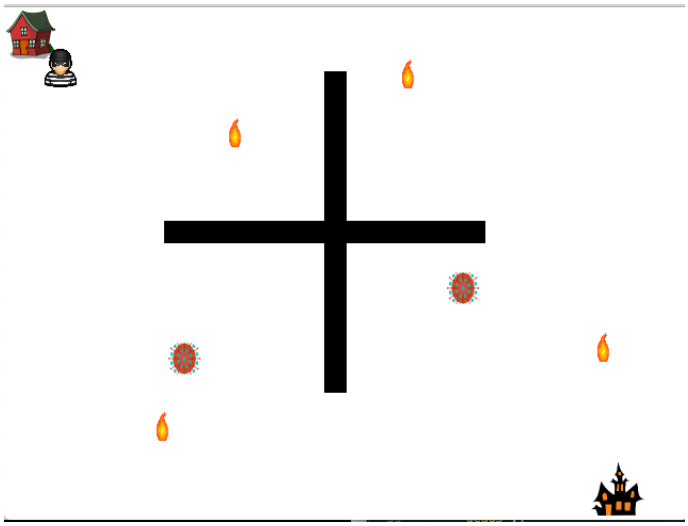

Figure 2: Map with a static obstacle

increase level of difficulty which would be directly related to increasing level of efficiency of the bots.

A-star algorithm A-star algorithm proves to be a powerful algorithm when the task is to find the shortest distance between the given source and a single the target location. This algorithm proves to be very efficient for a grid graph. Unlike Dijkstra's algorithm, this algorithm trims down the number of sub-paths smartly by taking into consideration a heuristics function. The heuristics function gives an estimate to the remaining distance from the current neighbour to the final target. The heuristic could be the Euclidean distance, Manhattan distance or any other metric depending on the graph.

In our case, we have a grid graph with some inaccessible points represented by obstacles. Had it been a graph without obstacles, the path could be a straight line from the source

coordinates to the destination coordinates and a Greedy Best Search would also work. The presence of obstacles requires the path to circumvent around the obstacle and traversing the minimum possible distance at the same time. We are evaluating the performance of A-star in varying level of obstacles forcing the algorithm to find convoluted paths resulting in high memory stack.

In our game, A-star algorithm is implemented for the guard bots as the source coordinates. The target coordinates could be the position of the human player, a power-up or any other key location as decided by the decision-making algorithm.

Recursive Best First Search The Recursive best first search is a type of informed search strategy. As we know that, the recursive best first search algorithm was created to mimic the operation of a standard best first search algorithm which also uses heuristic to get to the goal node in shortest possible path. But, the main motive was to create an algorithm that could do that using linear space.

This algorithm runs similar to recursive depth first search, but it does not go indefinitely down the path as depth first search does. It uses an f_limit feature value to limit the depth of the recursive sub-calls. Thus, whenever the algorithm finds out the f values of the nodes is increasing above the threshold f_limit value, it start to backtrack to find the next biggest f_limit value and then starts to traverse recursively from that node. So, in other words this f_limit value is used to keep track of best alternative track available from any ancestor of the current node from where the recursion has started.

If the heuristic chosen for this algorithm is admissible then the algorithm is both optimal and complete. That means the algorithm will always find the optimal path from the start node to the goal node if there exists one and complete because it will return failure if there is no such path. The time complexity and space complexity of this algorithm will be highlighted in the results and interpretation section of this document. The pseudo code for the algorithm that has been implemented in the game has been taken from (Russell and Norvig 2002).

Decision-Making

The decision making algorithms highly depend on the state of the game. In our game, we have the following 11 basic states:

1. Human Player is in Q1.
2. Human Player is in Q2 or Q3.
3. Human Player is in Q4.
4. Treasure is not Stolen.
5. Player reaches Haunted House (Treasure stolen state changes).
6. Treasure is Stolen.
7. Player gets a power-up (Player wins).
8. Bot gets a power-up (Player wins).
9. Player collides with a deadly obstacles (Player Loses).
10. Player collides with a Bot (Player Loses).
11. Player reaches back to the Safe House (Player wins).

The AI bots based on their current behavior, decided by decision-making algorithms, would either play defensive, in which case they will try to be in the vicinity of the treasure, or play attacking, in which case they will try to kill the human player, or play neutral, in which case they will just wander around aimlessly depending the proximity of the player to them or the player's goal.

The decision making algorithms for the bots would have different goals and set of possible actions based on the current state of the game which could potentially be a combination of some of the above states.

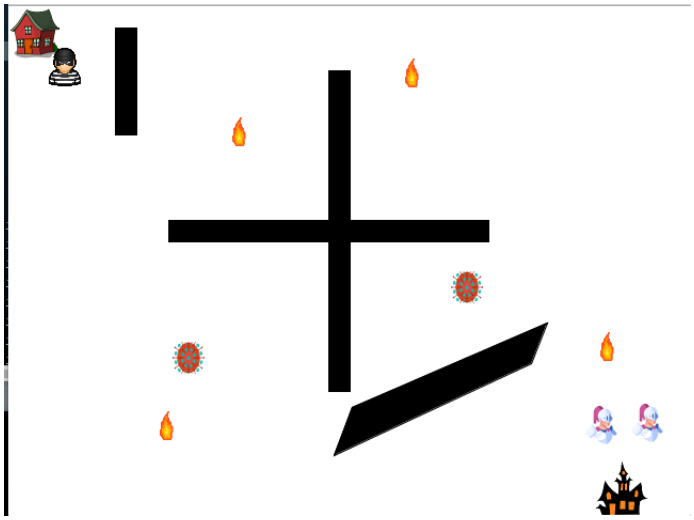

Figure 3: Map with random obstacles

As can be seen in figure 3 the game is dynamically changing. There are lots of moving objects, randomly generated traps and power-ups for bots and the player. The decision making algorithms also need to incorporate such changes in the world state. In each scenario, the bots need to decide best possible to action to prevent the player from winning. Similarly, the human player will try to win as hard as they can. This makes it a zero-sum game.

In this game, we have analysed following two decision making algorithms:

Behavior Trees The behavior tree represents all possible actions that a character can take. The route from the top level to each leaf represents one course of action and the behavior tree algorithm searches among those courses of action in a left-to-right manner. In other words, it performs a depth-first search and that is why is it also called Depth First Reactive Planning.

A node in behavior tree can be a Condition, Action, Composite Tasks such as a selector (kind of like an OR gate) or a sequencer (kind of like an AND gate), or a decorator. A decorator usually has only one child. We have modified our decorator to have multiple children but during execution

only one of them is selected based on their respective probability, making it a kind of non-deterministic random selector.

Consider an example: Initially, the player is in the first quadrant - in this situation, the guards can either protect the treasure by moving to and fro guarding around the Haunted House with some probability (say, p) or wander around the map with probability (1-p). Similarly, at different point in time, the available actions for the bots change and while iterating the behavior tree a randomly suitable action will be chosen based on current state of the game. All of these probability values are dynamic and can be tweaked or configured as required. An online learning algorithm could be used to improve these parameters but it is outside the scope of our focus in this work.

Goal Oriented Action Planning (GOAP) using Iterative deepening A^* (IDA^*) In a game, a character may have one or more goals, also called motives. There may be hundreds of possible goals, and each character may have any number of them active at a given point in time. Each goal has a level of importance, called an insistence. A goal with high insistence will tend to influence the character's behavior more strongly. A character will try to fulfill a goal or reduce its insistence. In addition to a set of goals, there are a set of possible actions to choose from in order to achieve that goal.

There are various algorithm such as: simple greedy selection based on insistence values, time-based selection, or overall utility selection based on discontentment, etc. to perform the decision making under this notion of Goal Oriented Behavior. But in this scenario, getting inspired from behavior tree or depth first reactive planning, a state space is recursively generated based on the current state of the game and every possible valid action based on the current state of the game. This can grow exponentially based on the number recursive steps and searching such a big state space that could be very slow. Also, this will give best possible actions for all the goals just like the Dijkstra's algorithm gives shortest path to all vertices from a single source. But, this is mostly not true in our case. For example, the goal could be either to kill the player or get a power-up to prevent player from reaching the safe house. So, we use A^* for planning - more specifically the Iterative Deepening A^* as it handles huge numbers of actions without swamping memory and allows us to easily limit the depth of the search.

For any A^*, we need a heuristic function that estimates how far a state space node is from having the goal fulfilled. When the treasure is not stolen a random target location near the Haunted house is selected so as to protect the treasure. And when the treasure is stolen, we try to guess where the human will go based on his current speed on the line joining his position and safe house's position. Then, we think of all the ways we can reach that point - go directly, or pick-up a power-up, or don't go at that point altogether and rather move back and forth near the safe hoping that player strikes us, etc. based on the time it'll take to do the action and the insistence value. Now this needs to be fast so have a max depth cut off and current search cut off. Also, we also keep transposition table to avoid searching the same set of actions over and over in each depth-first search. The hash value for the table is chosen as the action code that was last taken and the combination of python's id(state_space_object) as it is guaranteed to be unique and constant for this object during its lifetime. So, that if we come same object we can either choose to ignore it or take another action than last time if the depth thresholds would get exceeded.

Evaluation

We evaluate various statistics obtained from different path planning and searching algorithms that we use in the game for the AI bots. These statistics give a holistic difference of why one algorithm could be better than the other one in the context of this game.

The research will draw out trade-offs between various algorithms in terms of the following parameters.

1. Execution time: The time used by an algorithm remains one of the critical method for evaluation of an algorithm with the rest. It talks about how much time each algorithm could take in computing the next shortest path for the sub problem we are trying to solve.
2. Space Complexity: One of the major distinguishing factor among algorithms to be used in games is determined by the amount of space it takes to perform computation. In the world of cloud computing where there could be thousand of players playing on a bunch of servers, the memory consumption becomes a major determining factor as to which algorithm is better. Obviously, the one with less memory foot print becomes the better algorithm compared to the rest.
3. Number of blocks Computed: When comparing shortest path finding algorithms, one must compare the number of blocks computed by the algorithm to reach the goal or destination block from the source block. Algorithms like Dijkstra's algorithm and Breadth first search algorithm are known to compute far more number of blocks than algorithms like A^* which uses an heuristic function to find the next block among all the neighboring blocks to select from.

Results

Path finding algorithms

Before the start of the project, we explored among various algorithms for path finding. The ones that we found most suitable are the A star search algorithm and the Recursive best first search algorithm. Both of these algorithms could be better than one another in different contexts. We know that both of these algorithms are dependent on heuristic for path finding as they come under the category of Informed search strategies. They rely heavily on heuristics for reaching the goal state or the destination. Thus, if the heuristic given is admissible, then these algorithms are optimal and complete, i.e. they give the optimal path to reach to the goal state and also no matter what if the path does exists, then these algorithms will find and return that path.

Talking about the use of these algorithms in the games, we can say that these algorithms prove to be a major part

of the category of games that involve movement of objects inside the game space. The game we have developed has also been extensively utilizing these algorithms to find solutions and move bots in the game and tries to beat the human player in its pursuit to find the treasure. So, we decided to get some important metrics from these algorithms when they are run in the game and what could be better than plotting the data collected by these algorithms and then analysing them to find which one is the better of the lot.

For a computer scientist, the comparison of time complexity among algorithms is the most profound way to compare algorithms. Consequently, we have collected the time taken by each of these algorithms i.e. A-star and Recursive best first search for each search call the game makes in-order to move the bot from one place to another. Following is the graph that has been plotted over individual search call and subsequently after that is the graph for cumulative time taken by these search algorithms over a period of time in the game. These graphs based on time usage will allow us to perform a comparative study along with some other data that will be shown after these graphs.

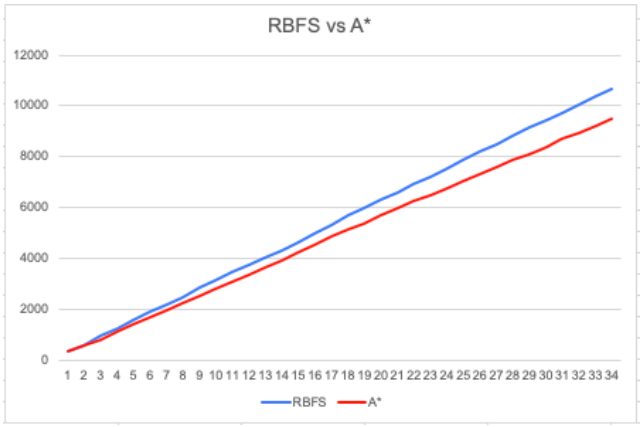

Figure 4: Time usage cumulative

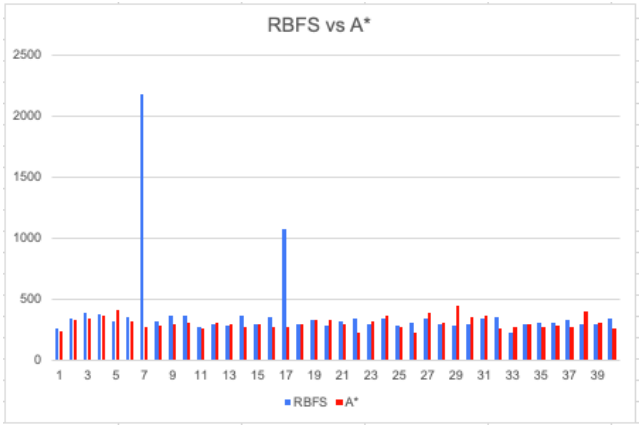

Figure 5: Time usage individual

Next, we have taken the data over the amount of size these algorithms take when a search call is made by the game to again move the bot from one place to another. These algorithms compute in memory queues to keep track of locations that they already accessed to make sure those locations are not accessed again. We have two graphs representing data from these algorithms from which the first one represents the individual memory consumption in each search call and the next one is the cumulative time taken by these algorithms over a period of time in the game.

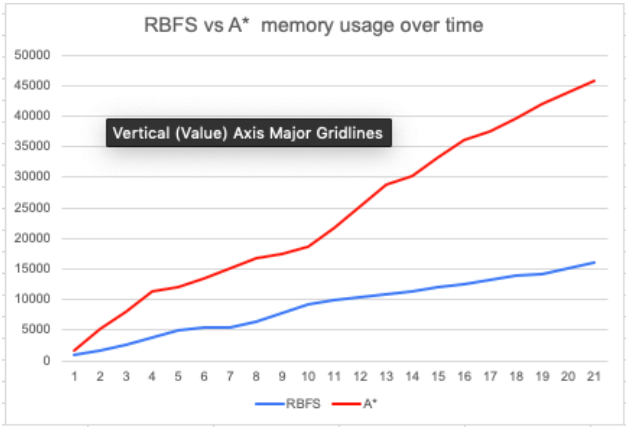

Figure 6: Memory usage cumulative

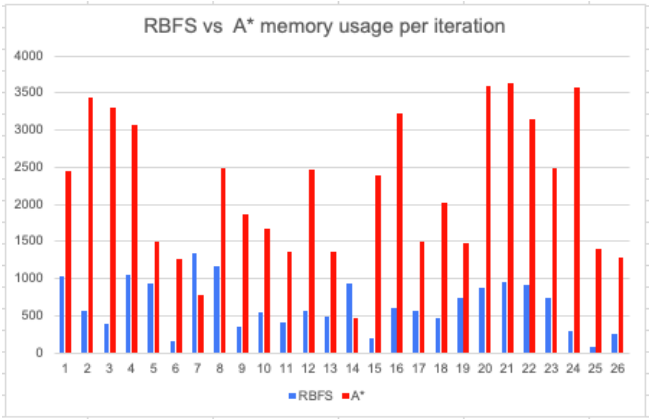

Figure 7: Memory usage individual

Decision Making algorithms

In the game, we have implemented Probabilistic Behavior Tree which represents all possible actions that a character can take at the current state of the game. We have introduced probability values in our custom decorator node, which usually has only one child node, to have multiple children representing possible actions bots can take at the current state. But only one of them is selected based on their respective probability and generating random number, making it a kind of non-deterministic random selector. A detailed example is given under Decision making algorithms in Methodology Section.

The next algorithm that we have implemented for decision making is Goal Oriented Action Planning IDA* which has heuristic based selection of actions. In this algorithm, we have a set of actions in every game state mapped to some insistence value that would make the game-play difficult for the user player. But the action selection here will not be based on only insistence values of the actions in every state. The heuristic based action selection algorithm will account for distance between the player and it's next destination, (i.e. either the safe house or treasure or chase human) and will add it to the initial insistence value of that corresponding action to determine the new set of actions that the bots will perform to make game play more difficult for the user.

We have calculated the metric for time take to make every decision when the player's actions change the state of the game. Here, every bar in the graph shows the time taken in microseconds to run the algorithm for decision making. To plot the graph, for every change in the state of the game, we run both the algorithms to find the time taken to find the next state of the game. So x-axis represents i^{th} decision made by respective algorithms during a single game.

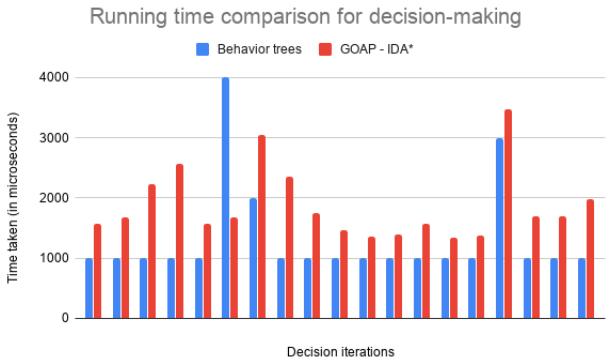

Figure 8: Time usage cumulative

Interpretation

Path finding algorithms

We observe that the time taken to execute the search calls by these two algorithms are fairly equal. This is apparent from the results shown in the 4 and 5. Theoretically, we learn that the time complexity of A-star algorithm increases exponentially with the increase in the branching factor of nodes in the algorithm. In contrast, the time complexity of the Recursive Best first search algorithm cannot be given mathematically, as the number of backtracking might vary because of the presence of goal node at different places in different iterations. But, it is generally believed that the time complexity of Recursive Best First Search can be approximately linear in best cases.

According to our findings, the time usage of both of these algorithms are almost linear in time because of the way these two algorithms are implemented in the code.

- In Recursive best first search algorithm, there are recursive calls to the same function whenever a new successor node is selected in the algorithm. So, the time required for switching from one subroutine call to another is more as compared to a subsequent memory location access.
- In A-star search algorithm, the openSet and the ClosedSet that have been formed to account for accessed nodes is a set or list of memory locations which are consecutive in memory. Thus, accessing these memory locations does not take much unit time.

So, theoretically, at least in some cases we must have less time usage for Recursive best first search algorithm as compared to A star search algorithm. But, because of the way it is implemented in the code the time taken by Recursive best first search increase by some factor due to subroutine calls which take more time than consecutive memory access. Therefore, because of this reason the time taken by these algorithms are slightly similar when compared with each other.

Each iteration	**RBFS**	**A***
Average time(in miliseconds)	369.60	303.45
Average queue size (count)	639	2201

Now talking about the space complexity usage of these algorithms, we know that the recursive best first search algorithm is a simple recursive algorithm that attempts to mimic the operation of standard best first search algorithm but using only linear space in memory while computing the shortest path. In case of A start algorithm the computation time is not the main drawback. Since it keeps all the generated node in memory, the A star algorithm usually runs out of space long time before it runs before time. This is quite evident from the figure 7, where all the bars in blue color that correspond to the memory usage of RBFS algorithm show that the it uses fairly less amount of time as compared to that of A star which is shown in red color.

The growth in memory space of the A star algorithm as shown in figure 6 infer that the memory space consumption is almost linear with respect to the number of search calls over a period of time in the game. Theoretically, we also know that the space complexity of the A star algorithm is O(b^m) where as the space complexity of the Recursive best first search algorithm is O(bm), where b is the branching factor and m is the size of the queue. Thus, our results over the memory consumption data is in lieu with the theoretical concepts.

Decision Making algorithms

In this section we will interpret the results shown in the figure 8. This figure shows the variations in time taken to execute both the simple selection variation and heuristic based variation of the Goal oriented action planning algorithm.

To interpret the results, we need to first understand theoretical time complexity and back-end implementation of the algorithms.

For Behavior Trees, we have used dictionaries in python. We have initialized the game tree using dictionary. At upper level we have all the states which are encoded with special

characters for representing Composite Tasks such as a selector, sequence or a decorator. For each state, we have a list of actions and the probability values at which each actions could occur. As explained our custom decorator will run of these actions and take the taken based on the probability value which is determined by rand number generation. Since we are using dictionaries with have on average $O(1)$ access time, but the depth of the tree will be equal to the possible number of actions. We have total time complexity of $O(n)$.

For GOAP using IDA*, we have created a heap data structure corresponding to all the actions of that state of the game. The heap data structure has a extract maximum property which can be used to find the maximum of all the available values in the heap. We calculate the distance heuristic of user player with its next goal and add it to the insistence value of the corresponding action and then create the heap data structure with these values. When we call extract maximum, it will give us the action with maximum heuristic value every time state changes and decision needs to be made for the next action. Upon extracting maximum value from heap, the heap data structure needs to heapify $O(log(n))$ itself based on the heuristic values that we have provided. This is then coupled with IDA* taking a total of $O(n \log(n))$ time to find the maximum among set of actions that were loaded in the heap data structure.

The spike in one of the iterations in figure 8 shows the probabilistic nature of our implementation of Behavior Tree. If a less probable and potentially a bad state is selected during last state change, it affects the time taken as we would have to traverse further deep in the behavior to make decision during next state change. Such spikes also correspond to a slight lag in the game which is bad for user experience. However the probability of happening the same is low. And overall as it takes less time on average, it is more simple and effective algorithm.

Iteration	**Simple Selection**	**Heuristic Selection**
Average time	999.60μs	1310.45μs

In the experimental results, we can see that the time taken by both of these algorithms do not vary much in term of the theoretical complexity of running these algorithms. The reason for the more time taken by heuristic based selection algorithm is that it is an $O(n \log(n))$ as compared to Behavior Tree which runs in $O(n)$ time. The extra $log(n)$ difference is small here because the number actions possible in our game is small and the total game states are only 11 as listed in Methodology Section. In case of behavior tree, for higher game complexity the assumption of constant access of dictionaries might fail. However behavior tree there is a great scope of writing parallel algorithm for even faster performance. For current game complexity, these algorithms have run time in microseconds, so the user does not find any difference in execution time while playing the game and thus we can say that both of these algorithms have only minor difference in time complexity which have been proven both theoretically and experimentally.

Future Scope

As a future work, we plan to work upon the decision tree implementation to enhance the performance by incorporating blackboard architecture, reusing trees, and improving reactivity. An online learning algorithm could be used to tweak the dynamic probabilities values used by our custom decorator for the bot's decision making to choose its actions.

The project can be extended to understanding a pattern in the movements of the user and facilitating the bots with more advanced path planning against the human player. Such algorithms would require huge computations and is apparently beyond the scope of this project. Nevertheless, the behavior of the bots would more closely resemble human enemies with a sense of intuition and experience.

Further, we can use self-training of the AI bots. For this, we could implement various state of the art algorithms like Deep Reinforcement Learning, Monte Carlo Q-learning, Markov decision process, Policy gradient methods such as Proximal Policy Optimization and Trust Region Policy Optimization

Finally, in context to this game, there can be additional power ups and traps which would challenge the decision-making behaviour of the bots. Presently, the game display is simply a two-dimensional canvas. There are some glitches too during the game-play. With better tools and resources, we could improve the rendering speed. There is also a huge scope of improvement in the user experience and user interaction in the game. There can be a treasure hunt mode in which the player would need to solve some quests before reaching to the final treasure.

Conclusion

The AI bots play against the human player in this zero-sum game. The ultimate goal of the human player would be achieved by completing different sub-tasks. This provides a platform to evaluate the behavior of the bots in multiple scenarios to best fathom the pros and cons of the algorithms.

The project helped us study and analyze the best use cases of different algorithms suited for different tasks like path finding and decision making. By using algorithms with different efficiencies, we were able to simulate to different levels of difficulty in the game.

By varying the parameters, we generate different level of efficiency of the AI bots while studying the intuition involved in tuning the parameters. We analyze the A^* algorithm by implementing the standard heuristics along with some other innovative heuristic functions.

Overall, the goal of this project is to study the trade offs between user experience and the computational costs involved. We sincerely believe that it is just not about the best gaming experience but also about the resources and the processing power required for the game-play. Sometimes, innovating ideas with relatively less complex implementations can provide a great gaming experience if programmed the right way.

References

Bhadoria, A., and Singh, R. K. 2014. Optimized angular a star algorithm for global path search based on neighbor node evaluation. *International Journal of Intelligent Systems and Applications* 6(8):46.

Cui, X., and Shi, H. 2011. A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security* 11(1):125–130.

Goyal, A.; Mogha, P.; Luthra, R.; and Sangwan, N. 2014. Path finding: A* or dijkstra's? *International Journal in IT & Engineering* 2(1):1–15.

Hoang, H.; Lee-Urban, S.; and Muñoz-Avila, H. 2005. Hierarchical plan representations for encoding strategic game ai. In *AIIDE*, 63–68.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Li, Y.; Harms, J.; and Holte, R. 2007. Fast exact multiconstraint shortest path algorithms. In *2007 IEEE International Conference on Communications*, 123–130. IEEE.

Littman, M. L. 1996. *Algorithms for sequential decision making*. Brown University Providence, RI.

Magzhan, K., and Jani, H. M. 2013. A review and evaluations of shortest path algorithms. *International journal of scientific & technology research* 2(6):99–104.

Millington, I., and Funge, J. 2009. *Artificial intelligence for games*. CRC Press.

Ogren, P. 2012. Increasing modularity of uav control systems using computer game behavior trees. In *Aiaa guidance, navigation, and control conference*, 4458.

Russell, S., and Norvig, P. 2002. Artificial intelligence: a modern approach.

Sinthamrongruk, T.; Mahakitpaisarn, K.; and Manopiniwes, W. 2013. A performance comparison between a* pathfinding and waypoint navigator algorithm on android and ios operating system. *International Journal of Engineering and Technology* 5(4):498.

Vamja, H. 2017. Comparative analysis of different path finding algorithms to study limitations and progress. *Int J Emerg Technol Adv Eng* 7(9):68–75.