Introduction

Artificial intelligence (AI) can be found in a range of technological fields, such as medicine, cyber security, data analysis, games and much more. Over recent years AI had a huge surge in popularity due to the creation of ChatGPT, Microsoft copilot and other generative AI.

AI has been incorporated into games since the 1950s. The development of non-player characters (NPC) using AI allows them to path-find, traverse various terrain, navigate through the virtual world, and even play strategically in some games such as FIFA. With the ever-growing gaming industry, there is a higher demand for smarter NPC that can not only navigate through a virtual world but can also engage with the player in challenging gameplay experiences. Smarter AI could also make for a more immersive gameplay experience.

There are multiple different AI programs that are used in-game such as:

1. A Finite State machine is a state machine that has finite states implemented. FSM states can be represented as an action or a state for an NPC. All states are connected to at least one other state. To move from state to state there needs to be a connection which is known as a transition. A transition will happen when certain conditions are met (Jagdake, 2021). For instance, an NPC might have states that include “*patrol*”, “*attack*” and “*runaway*”. With these states, the AI will first “*patrol*” around the virtual world until it sees an enemy and it will switch state to “attack”, while in the “*attack*” state if its health gets too low it will switch to “*runaway*” state. However, when we need a more complex logical system, we can look into behaviour trees. (Razan Ghzouli et, al)

A diagram of a patrol attack

Description automatically generated

Figure1.1 Finite State Machine process

1. Behaviour trees are a more flexible approach to AI behaviour. Behaviour trees consist of hierarchical structures of nodes that represent specific actions, conditions, or states. These nodes are connected to form a tree that tells the NPC what possible actions or behaviours are for in the current situation. Behaviour trees also allow NPCs to adapt to changing situations dynamically. (Ogen and Sprague, 2022)
2. Pathfinding AI which is commonly used for NPC to navigate through the game environment. The A\* algorithm is the best-known path planning algorithm, which can be used on metric or topological configurations. A\* uses heuristic searching and searching based on the shortest path (Duchon et al, 2014).
3. Goal-Oriented Action Planning System or GOAP is a planning architecture specifically designed for real-time autonomous character behaviours, unlike state machines and behaviour trees, GOAP decouples action from plans meaning the system will piece together a solution to problems in a changing environment. (Hartala, 2012)

This dissertation will look into the planning system for AI in games, specifically the Goal-Oriented Action Planning system (GOAP).

An Introduction to GOAP

GOAP was developed by Jeff Orkin at Monolith Productions (Hartala, 2012). Orkin et al. made F.E.A.R. and designed it to be an over-the-top action movie experience and with combat as intense as an online multiplayer game, Orkin wanted the A.I. to be able to “take cover, blind fire, dive through windows, flush out the players with grenades, communicate with teammate and more”. (Orkin, 2006)

In a GOAP system goal and action need to be defined when each action has preconditions and effects.

A goal would contain a condition and a desire.

A diagram of a goal

Description automatically generated

Figure 1.2 an example of a goal.

Action “*shoot*” can have a precondition and an effect.

A diagram of a movie shooting

Description automatically generated

Figure 1.3 an example of an action.

In such systems, the AI would also have a world state which allows the AI to know what items it has, what it can see, is it hurt etc.

A black and white rectangular with black text

Description automatically generated

Figure 1.4 an example of NPC world state.

Each NPC can have a different world state such as in F.E.A.R. What if the NPC does not have ammo in their gun to shoot the player, then the NPC would need to reload its weapon first. In this case, the GOAP system will use the actions that are available to construct a graph that leads to the desired goal state. For this, GOAP will use the A\* algorithm to find the shortest path to the goal state. Therefore, if the NPC already has a weapon, it will not need to craft a weapon.

GOAP has three key concepts which are Actions, Preconditions and Effects. Action can only be performed when certain preconditions are met (Orkin, 2006), such as, whether a certain NPC has enough stamina to complete a certain action or has enough money to buy a certain item in the shop, if not the GOAP system will look for a different way to complete its goal. Effects are the changes that happens in the NPC’s world such as changes in the environment, when a tree is cut down or a change in the total amount of money the NPC has, when it buys an item, and Effects can cause a Precondition to be satisfied.

Aim

The game will be made with a top-down perspective within an enclosed space. The AI's main objective will be to survive as long as it can. It would need to manage its hunger and thirst as well as build shelter. AI will be using different implementations of GOAP. Multiple GOAP Algorithms will be implemented and evaluated.

Objective

The objective of this dissertation will be to research and implement different Goal-Oriented Action Algorithms in the Unity Engine, once multiple GOAP algorithms are created an NPC will use it to complete sets of different tasks using different implementations of the GOAP algorithm.

Each algorithm will be tested multiple times and each time the world state will change e.g. starting environment can change, there could be one water source instead of two, etc.

Each algorithm will be tested and evaluated on how it behaves in certain situations e.g. changing weather conditions, how it would manage its hunger and thirst as well as building shelter. The AI will also be tested on performance such as CPU, Memory usage and the completion rate of each task.

Literature Review

STRIPS

Stanford Research Institute Problem Solver (STRIPS) was developed in 1971 by Richard Fikes and Nils Nilsson at SRI International (Wikipedia, no date). STRIPS consist of goals and actions where goals are the desired state of the world that the NPCs would like to reach, and actions are defined as preconditions and effect, each action will only execute if all its preconditions are met, and each action can change the state of the world. (Orkin, 2006)

STRIPS belongs to the class of problem solvers that search a space of “world states” to find goals that can be achieved. For each world state, there will be a set of applicable operators(actions) each of which can transform the world state to some other world state. The task for STRIPS is to find a composition of operators(actions) which can be chained together to transform a given initial world state into one that satisfies some particular goal condition. (Nilsson and Fikes, 1970)

A diagram of a process

Description automatically generated

Figure 2.1 Flowchart for the STRIPS executive (Nilsson and Fikes, 1970)

GOAP and STRIPS – A Comparison

GOAP is similar to STRIPS however Orkin et al. stakes four key differences, first would be GOAP's use of cost per action, this can be either a made-up number that is created by the user depending on the environment of their game, or it could be a cost of buying a sword compared to crafting one yourself within the game. In this case, there would need to be an algorithm to search through all the costs of different paths that can achieve the same goal. This is where the A\* algorithm comes in, A\* would search towards the lowest cost sequence of each action to satisfy a goal.

Whereas in STRIPS there is no cost assessment for each action therefor in STRIPS it will only look for one path that can satisfy a specific goal (Orkin, 2006).

The second difference stated by Orkin et al. is that there is no need to add/delete lists. Assuming an NPC is using a STRIPS system, and it is hungry then the NPC can either call a takeout company or it can make food for themselves at home. Within the STRIPS system, you would need to delete one assignment to the value of “*Hungry*” and add another, rather than changing it. STRIPS need to do this due to there is nothing in the formal logic to contain a variable to only one value. So, if “*Hungry*” was “*YES”*and the effect of “*OrderingPizza”* was to add knowledge that *Hungry* is now set to “*NO*”, the situation would now be “*Hungry*”is both “*YES*” and “*NO*”. (Orkin, 2006)

Orkin et al. chose to represent preconditions and effects as a fixed-sized array representing the world states. This will make it trivial to find the actions that can have an effect which satisfies some goal or precondition. For example, a “*ShootGun”* has the precondition that the gun is loaded, and the “Reload”action will have the effect that the gun is loaded. This makes it easy to see that both the “*ShootGun*” and “*Reload*” actions can be chained.

A close-up of words

Description automatically generated

Figure 2.2 Examples of types of variables that are stored in an array. (Orkin, 2006)

Orkin et al. state that the two versions of the “*AtNode*” variable show that some variables may have a constant or a variable value. “A variable value is a pointer to the value in the parent goal or action’s precondition world state array” (Orkin, 2006). An example of this is a “*Hide*” action, which can satisfy the goal of “*GetCover*” allowing the AI to go to a desired cover node. The “*GetCover*” goal specifies which node to go to in the array representing the goal world state.

Having a fixed array does have its own limits, such as, while an AI can have many weapons and have multiple enemies to deal with, it can only be able to reason with one of each during planning because the world state array only has one slot for each. In this case, an attention-selection sub-system outside the planner will deal with this. The Targeting and weapon system will choose which weapon and enemy are currently in focus, and the plan will only need to concern itself with them. (Orkin, 2006)

The third difference stated by Orkin et al. is procedural precondition. Since it was not practical to represent everything that was needed in the entire virtual world in a fixed-sized array of variables, Orkin decided to add the ability to run additional procedural precondition checks. For the game F.E.A.R, an action is a C++ class that has preconditions that both represent an array of the world state variables and as a function that can run additional filters. The “*Escape*” action will allow the NPC to find a safe route to run away, but it can only do so if a method called “*CheckForEscapePath*” returns true after searching through the NaveMesh for a safe path. It would be impractical to keep track of an escape path that exists in the world. The procedural precondition function will only run on demand whenever it is necessary, this would be a much more effective way to run the program because pathfinding is expensive.

STRIPS do not have procedural preconditions making it hard for an NPC to find a safe path to escape, in a STRIPS system it will find the first path that leads to a safe space and go for it no matter how dangerous the path may be. (Orkin, 2006)

The final difference stated by Orkin et al. is procedural effect which is like procedural preconception. An effect should not be applied instantaneously which would indicate that something has happened even if an action has not been fully completed. An effect would take time such as reacting shelter to hide or eliminating an enemy. This is where the planning system can connect to the FSM. When a plan is executed, an action is sequentially activated, which will set the current state and any associated parameters. Procedural effect is not part of STRIPS.

GOAP Algorithms

### Dijkstra Algorithm

Dijkstra algorithm is for finding the optimum path, this algorithm searches for the minimum cost path among all the paths in order, beginning with the start point. This method thus has some disadvantages such as poor search efficiency and a long search time when the distance to the destination is large. (Noto and Sato, 2000)

The algorithm follows six steps according to Jaskia et al (2012).

1. Choose the starting node
2. Define a set list of nodes. As the algorithm progresses, the list nodes will store those nodes in which the shortest path has been found.
3. Give the starting node a value of 0 and add it to the list.
4. Consider each node not in the list connected by an edge from the newly inserted node. Label the node not in the list with the label on the newly inserted node plus the cost of the edge.
5. Pick a node that is not in the list with the smallest cost and add it to the list
6. Repeat step 4 until the destination node has been added to the list or all the nodes have been added.

Overview

Dijkstra's algorithm will always find the shortest path towards its destination no matter how big the graph gets. However, there are problems with Dijkstra, when a graph gets too big it takes a long time to search through all the paths until it finds the shortest path, another problem is when Dijkstra’s algorithm cannot have a negative number as the cost of the edges, this is because Dijkstra’s algorithm is a Greedy Approach, once a node is marked as visited it cannot visit it again even if there is another path with less cost or distance. (Noto and Sato, 2000)

The greedy algorithm can make locally optimal choices in hopes that this will lead to a globally optimal solution. The greedy does not always give the optimal solution but for a wide range of important problems, the greedy algorithm is quite powerful (Vince, 2002). However, a different sentiment is given by Jensen et al (2004) where that even for the polynomially solvable problem the greedy algorithm may produce the unique worst possible solution. This was proven by Gutin et al (2002) showing that the greedy algorithm fails on other combination optimization problems.

Use Cases

As a path-finding algorithm, there are many usages for Dijkstra’s method such as map navigation, traffic engineering in IP Networks and improving Intelligent and Transportation Systems. (Jasika et al, 2000). Shown in a study by Tirastittam and Waiyawuththanapoom (2014) they used the Dijkstra algorithm to create a public transport planning system: A case study Bangkok Metropolitan Area focusing on buses. This application evaluated 30 normal users and found the results were a good level of satisfaction. From the results, they concluded that the system can be used properly and effectively.

Dijkstra’s Algorithm can be used in a GOAP system since the GOAP system will construct a graph with each node having a cost assigned to them, we can use Dijkstra’s Algorithm to find the shortest path and assign that path to the NPC to complete a task.

Pseudocode

A screenshot of a computer code

Description automatically generated

Figure 2.3 Dijkstra Pseudocode (Wikipedia, no date)

### A-Star Algorithm

Balster et al (2019) state that there are two main components that make up an A\* algorithm.

1. OPEN list that keeps track of the needs that need to be explored and the list starts from the root node.
2. CLOSE list that keeps track of the nodes that have been explored and the list starts off empty.

The decision to move from one node to another is dependent on the f(n) value which can be calculated using the equation: f(n) = g(n) + h(n).

The n is the next node, g(n) is the cost of the path from the start node to n, h(n) is the heuristic function that estimates the cost from n to the goal, and f(n) is the sum of both g(n) and h(n). The nodes with the lowest f(n) values make up the path taken from start to finish. (Balster et al, 2019)

Overview

In an A-Star algorithm, you first will need to assign it a starting node and then it will check for any adjacent nodes that are next to it and see if the node is open. After moving from the current node, update the f(n), g(n) and h(n) calculations on the OPEN list. After the calculations are made check to see if the target node was reached, if the target node has been reached then record the final score of f(n), g(n) and h(n). If the target node has not been reached, then check for the next adjacent node. If the adjacent node is not open, add it to the CLOSED list. Repeat the process checking through all the nodes until they are all closed. Then form the OPEN list and sort the nodes from the start to the target node, with the lowest f(n) cost. The one with the lowest f(n) cost will make up the path. (Balster et al, 2019)

A diagram of a flowchart

Description automatically generated

Figure 2.4 A\* Diagram (Balster et al, 2019)

Balster et al (2019) state one of the main advantages of A\* is its simplicity. The calculation needed to complete for each node does not need a lot of time or a lot of space to store the node. One disadvantage is that the algorithm cannot react to unexpected added or moving objects. The A\* cannot adjust the list made for creating a path to take.

Liu et al (2011) state that A\* algorithms use different heuristic functions which may lead to different search preferences. In Liu et al paper they explored three typical A\* algorithms and compared their rescue-oriented maze search efficiency.

Use Cases

There are multiple use cases for A\* such as:

* Navigation and Maps: A\* is extensively used in GPS to find the most optimal path
* Video games: the algorithm helps NPCs navigate through the game world
* Robotics: A\* guild robots to navigate through obstacles.
* Puzzle Solving: A\* can be applied to solve puzzles such as mazes and Rubik’s Cube.

(Umar, 2023)

Liu et al (2011) used three different A\* algorithms to complete a maze search comparing the capabilities and efficiency of their different heuristic function. The experiment validated the usefulness of the heuristic function with the result that the A\* algorithm outperformed the depth-first search algorithm in most cases.

Pseudocode

A computer screen shot of a program

Description automatically generated

Figure 2.5 A-Star Pseudocode

### Dynamic A-Star (D\*)

The differences between A\* and D\* are very similar, however, D\* has two additional lists: RAISE and LOWER. The RAISE list is an array that contains all the nodes' path costs that are higher than the last time the node was on the OPEN list and the node would not be automatically moved to the CLOSED list. The LOWER list is an array that contains all the nodes whose path cost is lower than the last time a node was on the OPEN list and the node is not automatically moved to the CLOSED list. Nodes are only moved to the CLOSED list when the cost to move to a node is too high or it is a dead-end path. (Balster et al, 2019)

Overview

Similar to A\* algorithm D\* will be given a start good and then check if the adjacent node is open. If the adjacent node is open then it will update the f(n), g(n) and h(n) values and compare the past calculation. If h(n) is greater than g(n) the node will be added to the RAISE list but if h(n) is equal to g(n) then it will be added to the LOWER list. If the target node has not been reached check if the next adjacent node is open if not add it to the CLOSED list. Repeat this cycle until the target node is reached. Once the target node is reached the nodes with the lowest calculation make up the selected path of the algorithm. (Balster et al, 2019)

Balster et al (2019) also state that if an unexpected obstacle is found, the original path will be discarded in favour of a new path that will be found where the obstacle is no longer in the way. All obstacles have a constant h cost for exploration, the final path is decided by the h(n) cost alone unlike A\* relies on the calculation result.

A diagram of a process

Description automatically generated

Figure 2.6 D\* Diagram (Balster et al, 2019)

One advantage of D\* is the ability to re-plan a path if the first path is blocked by using the calculation already done by the algorithm. This allows for a higher chance of finding a successful path through a maze. D\* also works better with more complex puzzles. The downside of this algorithm is the amount of time it takes to calculate a plan and if the plan must be replanned the length of time will increase even more. (Balster et al, 2019)

Likhachev et al (2005) stated that D\* is up to two orders of magnitude more efficient than planning from scratch with A\*, and it has been used extensively by field robotic systems. D\* maintain the least cost path between a start state and any number of goal states as the cost of arcs between states changes. Likhachev et al (2005) also stated that D\* can handle increasing or decreasing arc states which are suitable for solving goals-directed mobile robot navigation problems, entailing a robot moving from the initial state to many different goal states.

Use Cases

Similarly to A\*, dynamic A\* can also be used for navigation in GPS and maps, Dynamic A\* can also be used to solve mazes and help robots navigate through obstacles as stated by Likhachev et al (2005).

Likhachev et al (2005) used D\* to simulate a robot kinematic arm, their algorithm tunes the quality of its solution based on available search time, at every step reusing previous search effects. The result was D\* was able to generate solutions to complex dynamic path planning problems effectively, the algorithm works by continually decreasing suboptimality bonds on its solution and reusing previous search effects as much as possible.

Pseudocode

A computer screen shot of a code

Description automatically generated

Figure 2.7 Dynamic A-Start Pseudocode

Design

World Design

AI Mechanics

Goals

Actions

### Weather System

### Hunger And Thirst System

### Building Shelter System

GOAP Algorithm Design

Evaluation

Comparing AI Movement Patterns

Efficiency of Each Algorithm

### Dijkstra Algorithm

### A-Start Algorithm

### Dynamic A-Start Algorithm

Reference