Evaluating Performance and Usage of Goal-Orientated Action Planning Algorithm in Interactive Digital Environment.

A DISSERTATION SUBMITTED TO MANCHESTER METROPOLITAN UNIVERSITY

FOR THE DEGREE OF MASTER OF SCIENCE

IN THE FACULTY OF SCIENCE AND ENGINEERING

A blue and white logo

Description automatically generated

2020

By

Hualiang Zhao

Department of Computing and Mathematics

Contents

[1 Contents 2](#_Toc177221246)

[2 List of Figures 4](#_Toc177221247)

[3 Abstract 5](#_Toc177221248)

[4 Declaration 6](#_Toc177221249)

[5 Acknowledgement 7](#_Toc177221250)

[6 Abbreviations 8](#_Toc177221251)

[7 Introduction 9](#_Toc177221252)

[7.1 AI In Medical Practice 9](#_Toc177221253)

[7.2 AI In Games 9](#_Toc177221254)

[7.3 An Introduction to GOAP 11](#_Toc177221255)

[7.4 Aim 14](#_Toc177221256)

[8 Literature Review 14](#_Toc177221257)

[8.1 STRIPS 14](#_Toc177221258)

[8.2 GOAP and STRIPS – A Comparison 15](#_Toc177221259)

[8.3 GOAP Algorithms 17](#_Toc177221260)

[8.3.1 Dijkstra Algorithm 17](#_Toc177221261)

[8.3.2 A-Star Algorithm 18](#_Toc177221262)

[8.3.3 Dynamic A-Star (D\*) 20](#_Toc177221263)

[9 Design 22](#_Toc177221264)

[9.1 World Design 22](#_Toc177221265)

[9.2 AI Mechanics 22](#_Toc177221266)

[9.3 Goals 23](#_Toc177221267)

[9.4 Actions 23](#_Toc177221268)

[9.4.1 Weather System 23](#_Toc177221269)

[9.4.2 Hunger And Thirst System 23](#_Toc177221270)

[9.4.3 Building Shelter System 23](#_Toc177221271)

[9.5 GOAP Algorithm Design 24](#_Toc177221272)

[9.6 Methodology 24](#_Toc177221273)

[10 Evaluation 24](#_Toc177221274)

[10.1 Comparing AI Movement Patterns 24](#_Toc177221275)

[10.2 Efficiency of Each Algorithm 24](#_Toc177221276)

[10.2.1 Dijkstra Algorithm 25](#_Toc177221277)

[10.2.2 A-Start Algorithm 25](#_Toc177221278)

[11 Reference 25](#_Toc177221279)

List of Figures

[Figure1.1Finite State Machine process 4](#_Toc177220705)

[Figure 1.2 Nodes are represented by rectangles. service nodes are represented by an infinity symbol, the question mark is a decision node, the observer decorator is the eye symbol, and the star symbol is the task node. (Agis et al 2020) 5](#_Toc177220706)

[Figure 1.3 an example of a goal. 6](#_Toc177220707)

[Figure 1.4 an example of an action. 7](#_Toc177220708)

[Figure 1.5 an example of NPC world state. 7](#_Toc177220709)

[Figure 2.6 A-Star Pseudocode 14](#_Toc177220710)

[Figure 2.7 D\* Diagram (Balster et al, 2019) 15](#_Toc177220711)

[Figure 2.8 Dynamic A-Start Pseudocode 16](#_Toc177220712)

Abstract

Declaration

No part of this project has been submitted in support of an application for any other degree or qualification at this or any other institute of learning. Apart from those parts of the project containing citations to the work of others, this project is my own unaided work. This work has been carried out in accordance with the Manchester Metropolitan University research ethics procedures, and has received ethical approval number Your EthOS Number.

Signed:

Date:

Acknowledgement

Abbreviations

FSM Finite State Machine

GOAP Goal-Oriented Action Planning

STRIP Stanford Research Institute Problem Solver

AI Artificial intelligence

ms Millisenconds

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.

AI In Medical Practice

A paper published by Miller and Brown (2018) states that AI in the medical field uses machine learning to detect patterns that are not decipherable using biostatistics by processing large amounts of data through layered mathematic models, by correcting algorithm mistakes, it would add to the AI predictive model confidence. AI has been successfully used in imaging analysis in radiology, pathology and dermatology, with diagnostic speed exceeding and accuracy paralleling medical experts. However, despite this, similar to human thinking predictive model confidence never reaches 100%.

Within the medical field, there are different types of Artificial intelligence algorithms being used such as Machine Learning (ML), this program uses self-improvement and learning with no experience or being trained over some time. ML can evaluate the medical results automatically and present them with a probabilistic degree of accuracy, ML can make decisions with algorithms and methods such as supervised learning, unsupervised learning and reinforced learning. Within the medical field ML is used to identify the probability of disease and ML is helpful in saving the record of patients for better treatment (Haleem et al, 2019).

Miller and Brown (2018) state that AI was not specifically developed as a tool for health care. While AI is poised to address indurate medical practice, it is neither astute nor intuitive, therefore, humans will remain essential to the intelligent use of AI within the medical space. However, a paper written by Haleem et al (2019) has a different conclusion where they state that AI can help to monitor and proper treatment for patients, it can assess images or results without needing a doctor, surgeon or clinician. The practical use of AI in medical space would be the prevention of disease and reduce medical cost and this technology is used to answer patient questions and reduce unnecessary hospital visits. AI can also help identify cancer and its appropriate treatment. In the coming years. Its application will be for digital supervision in hospitals to improve patient care

AI In Games

Although AI originated in a laboratory, it has now been coopted by designers of video games, and work is underway to increase the learning power of video game cast of characters with human players (Cass, 2002).

The research field of artificial intelligence in games, mainly game AI, has existed as an individual one for roughly 15 years. Games have served as a useful measure of the signs of progress in AI. Games tend to have priorates of vast state space and high complexity making them excellent benchmarks for the evaluation of AI (Xia et al. 2020).

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

1. A Finite State machine (FSM) 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.1Finite 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. (Ogren and Sprague, 2022)



Figure 1.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.2 Nodes are represented by rectangles. service nodes are represented by an infinity symbol, the question mark is a decision node, the observer decorator is the eye symbol, and the star symbol is the task node. (Agis et al 2020)

1. 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). A\* algorithm was aimed to process process-efficient path planning with multiple nodes by using heuristic functions. Peter Hart, Nils Nilsson and Bertram Raphael from Stanford Research Institute introduced this in 1968 (Candra et al, 2021). Candra et al (2021) state A\* algorithm is a best-first search and finds a path with the lowest cost of the initial node given to one destination node.
2. 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)GOAP goes a step further than other systems by allowing the AI to decide what to do and how it should be done. AI using GOAP periodically reevaluates its situation and chooses the optimal path to achieve its most prioritised goal. (Sloan et al, 2011)

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

An Introduction to GOAP

Creating an illustration that NPCs are effectively coordinated with each other is what matters, after all, this is what players perceive. However, in F.E.A.R. these tricks may not be feasible to achieve. GOAP is a simplified STRIPS-like planning architecture that was specially designed to handle real-time autonomous character behaviours, it was developed by Jeff Orkin at Monolith Productions (2006) unlike state machines and behaviour trees GOAP’s actions are decoupled meaning the system will need to piece together a solution to a problem in a changing environment (Hartala, 2012). Hartala et al (2012) also stated that GOAP is at its best when used as a high-level decision-making system, such as using it for the brain of the AI agent.

Orkin et al (2006). 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.3 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.4 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.5 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.

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, the algorithms will be compared by how efficient when finding a path towards its goal, such as how long it takes for it to find the lowest costing path. The algorithm will also be compared by which path it would take to complete the goal, since more than one planning algorithm will be used one might choose a different path than the other one.

When comparing the models the main thing to look for is the time it takes to find a path with the lowest cost, and how each algorithm may choose to use a different path from the other.

Literature Review

STRIPS

Stanford Research Institute Problem Solver (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)

STRIPS was developed in 1971 by Richard Fikes and Nils Nilsson at SRI International it was developed in conjunction with robot research at SRI. STRIPS belongs to a class of problem solvers that search a space of world models, the task for problem solvers is to find some composition of operators that can transform a given initial world model into one that satisfies some goal condition, this framework for problem solvers has been central to much of the research in artificial intelligence, however Nilsson et al (1971) main interest was the class of problems faced by robots in re-arranging objects and in navigating (Nilsson and Fikes, 1971).

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)

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. (2006) state four key differences first would be GOAP's use of cost per action, which 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 therefore 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. (2006) 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). Nilsson et al (1970) stated that STRIPS used the assumption that the initial world states would not change by the application of operators. Given these assumptions, Nilsson et al (1970) implemented a scheme for handling changing world states where each state produced by STRIPS is defined by two clause lists. The first list is the DELETION list, where all those clauses from the initial world states are no longer present in the world state that is being defined. The second list is the ADDITION list, where all those clauses in the world state that are being defined are not in the initial world state that is needed to form the new world state being defined. This statement by Nilsson et al (1970) goes hand in hand with Orkin's statement that states need to be deleted because there is no logic in STRIPS that handles changing world states.

Orkin et al. (2006) 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. (2006) 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 can be added outside the planner to deal with this. This system will allow the AI to pick different enemies to target depending on either their distance from the AI or the threat level of the enemy. 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 (2006). 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. (2006) 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.

A group of numbers and lines

Description automatically generated with medium confidence

Figure 2.2 Example of Dijkstra Algorithm (Jasika et al, 2012)

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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.6 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.7 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.**Error! Use the Home tab to apply 0 to the text that you want to appear here.**.8 Dynamic A-Start Pseudocode

Design

World Design

When designing the world, it would need trees, water sources and food which will allow the NPC to survive for a long time. There will also need to be enough space for the NPC to build a shelter which will allow it to hide from the rain and dry off.

This world will be created in Unity and the models that are used to make this world will be from Kenny et al (2024). The world would be in a 3D environment with a top-down view of the whole world. With a 3D top-down environment, it would be easy to see what the NPC is doing when given a goal, this makes it easier to compare different from each algorithm.

AI Mechanics

The AI will have three Mechanics that they would need to manage first, hunger when the AI first spawn it will have full hunger (100%) over time it will go down slowly and the AI will have to find food to satisfy the hunger, within the world, there will be apples scattered around and the AI can go to one of the apples and eat it to satisfy their hunger.

Second is their thirst, it would work similarly to hunger where it will start at a hundred per cent and slowly go down over time, when it gets below a certain threshold then the AI will need to find a pond and drink from it to satisfy its thirst.

Lastly is their shelter building and keeping dry for this mechanic when the AI detects that it is raining in the world it starts to plan and build a shelter, the dryness of the AI will decrease over time as long as it is raining, to keep the dryness as high as possible the AI will need to hide in the shelter, once in the shelter, the AI’s dryness will go up over time.

Goals

The goal will be in their own class and all goals will be stored in a dictionary therefore there will be a string and an integer value for each goal. The string value will be the name of the goal that the planner will be able to use to construct a plan to complete the task and the integer value will be the priority of the goal the higher the integer is the higher the priority the goal will be. The goal will also have a remove Boolean which determines if the goal should be removed when it is complete this would be useful such as when creating a shelter you would want to remove this goal once it is completed and not make multiple shelters.

Actions

The action class will be inherited by Actions such as “MoveToApple” and “BuildShelter”, the action class will include preconditions and aftereffects, the precondition will be the condition that has to be met before the action can be done, an action can have multiple preconditions there for it will use a dictionary with string and integer value the string will be the name of the precondition and the integer value will be the amount of that conditional value.

The aftereffects will be the effect that will be applied to either the AI or the world state such as “BuildingShelter”, this action’s aftereffect will be applied to the world states where a shelter will be added to the world state. “MoveToApple” action will have an aftereffect on the AI where the AI will gain hunger value when moved to the apple. After-effect will also be stored in a dictionary because once an action is completed there can after-effect that can happen, the dictionary will have a string and an integer value where the string will be the name of the after-effect and the integer value is the amount of that conditional value

For each action, there will also be a cost assigned to them this will allow the planning algorithm to find the lowest cost possible to complete a task. This cost is important because it allows the AI to construct a cost-effective plan and complete it as quickly as possible.

### Weather System

The weather state will be stored in the overall world state, however, since the world states only take in string and integers the easiest way to do this would be to make 0 where there is no rain and 1 where it is currently raining. This will also allow other weather types in the future such as snow and wind which can use the values of 2 and 3 giving the AI more challenges to overcome.

### Hunger And Thirst System

Both hunger and thirst systems will work the same once the game starts the AI will begin to lose hunger and thirst over one tick per second and when the hunger or thirst gets below a threshold a goal will be assigned to the AI telling it to get some food or find water, the priority of the goal for finding water or food will go up depending on how low the hunger or thirst value is.

### Building Shelter System

For the build system, the AI should move to a tree first, gather some wood, then move to a dedicated spot and build a shelter. For the building shelter goal to activate, it should be raining and the dry percentage on the AI should be 50 or lower, the priority of building shelter would go up depending on how low the dry percentage of the AI is.

The AI would also need to keep its dry percentage as high as possible therefore once the shelter is built and it is still raining the AI will move to the shelter to dry off once its dry percentage is below a threshold. The priority of drying off would also increase depending on how low the dry percentage is on the AI.

GOAP Algorithm Design

The GOAP algorithm will work backwards starting from the goal state, it will then look for an after-effect of each action and find a state that matches the goal state, if the current action has a precondition then the algorithm will look at other actions to find an after-effect that will match with that precondition, this will repeat itself until an action is found without any preconditions. Once a path is found it will then store that path in a list, this will continue until all unique paths are found. A new list will then be created called “tree” and this list will have all the paths inside it and will act like the root node.

Methodology

To collect data on both A\* and Dijkstra’s algorithm, Unity has an inbuilt timer that displays how long it took to run a function. The game will be kept running for a few hours for A\* and Dijkstra and all the data will be stored in a CSV file. CSV file was chosen because it can be easily written into and extracted out of Unity folders, since CSV file is in Excel gathering the mean, median lower and upper quartile would be simple to calculate.

To compare AI movement patterns, Unity will output the path plan into the console, and I would write down what path it took making sure it sees if there are any inconsistencies when comparing both algorithms. When testing for this the program would be stopped and started again once a few goals have been performed this will avoid chances of doing different tasks at the same time interval since hunger, thirst and dryness are ticking down and inconsistency can occur.

All testing will run on a computer with Intel core i7-10700K, graphic card RTX 3060, 32GB of RAM and with the latest drivers installed for both Windows and graphics card on 14th September 2024.

Evaluation

GOAP Planning

For building shelter goal that was given to the AI, it generated 120 paths that could complete the task, however when A\* and Dijkstra’s algorithms were run to find the most optimal path both algorithms were able to find the most optimal path to complete a task, when checking the plans that were output by the algorithm it seems that all the plans end with the goal state.

A screenshot of a computer program

Description automatically generated A screenshot of a computer program

Description automatically generated

Figure 3.1 Comparing A\* and Dijkstra building shelter paths

Figure 3.1 shows the number of plans that were generated when the AI is given the goal to build shelter and keep dry. The path it chooses to go for, the AI will first go to a tree and get wood then build a shelter and sit inside of it to dry off before coming back out. As shown in the comparison between A\* and Dijkstra’s algorithm both algorithms have chosen to go with the same path.

A black screen with white text

Description automatically generated A screenshot of a computer

Description automatically generated

Figure 3.2 A\* and Dijkstra Finding Food Path

Figure 3.2 shows that when the AI is hungry it generates 71, and the AI will move to an apple and then eat it, both algorithms have chosen to go with the same path even though there are 71 different outcomes generated.

A screenshot of a computer

Description automatically generated A screenshot of a computer program

Description automatically generated

Figure 3.3 A\* and Dijkstra Collecting Water Path

Figure 3.3 shows when the AI is thirsty it will go to a pond to collect water, the AI will first visit a tree and then visit a pond to collect water, the GOAP algorithm has generated 88 plans for thirst and both A\* and Dijkstra’s algorithms has chosen to go down the same path.

A black background with white text

Description automatically generated A black background with white text

Description automatically generated

Figure 3.4 A\* and Dijkstra Drying off Path

Figure 3.4 shows what happens when the shelter is already built and the AI needs to dry off, it will only choose to move to the shelter and not build another one. There are 164 paths generated by the GOAP algorithm and both A\* and Dijkstra’s algorithm have chosen the same path.

Overall, the GOAP algorithm is working as expected since it can generate planes to complete a task and both A\* and Dijkstra’s algorithm is taking the shortest path possible to complete the goals that were given. This is most apparent when the AI need to dry off a second time as both A\* and Dijkstra’s algorithms choose not to rebuild the shelter and only just move into the shelter.

Comparing AI Movement Patterns

When comparing the A\* and Dijkstra plan patterns there was no change in patterns the AI seemed to have taken the exact same route to complete a goal, with the same actions running, this could be due to the simplicity of the heuristic function in the A\* algorithm, the heuristic function that was used is the distance between the current node that was being looked at the destination node, this simplistic heuristic function coupled with cost that I believed was suitable to this research would of made it possible for A\* and Dijkstra to take the same path.

Efficiency of Each Algorithm

To compare the efficiency of each algorithm I have run both A\* and Dijkstra in the same world environment, with the same starting point, the same cost for each action to combat, and switching the computer's anti-virus to combat any inconsistency.

Each algorithm has been running for around an hour and thirty minutes with 200 results in the CSV file. The boxplots shown in Figure 3.1 and Figure 3.2 are the mean, median, lower quartile, upper quartile, minimum, and maximum time it took to find a path with each algorithm. Throughout this experiment, there have been no crashes to Unity client therefore I believe the result should be sufficient.

Explain average median q1, q2, etc

On average the time to find a plan on the Dijkstra algorithm was 2.895 milliseconds(ms), compared to A\* 2.479 milliseconds(ms), the maximum time spent on finding a plan with Dijkstra’s algorithm is 2.901ms whereas A\* maximum time spent was 2.483ms. This means that A\* was around 0.419ms faster on average than Dijkstra's algorithm, and when it came to the worst case it was 0.418ms faster for the A\* algorithm, this is due to A\* not needing to go through all the nodes to find a path, whereas Dijkstra needs to look through all the nodes to be able to find the most cost-effective path.

### Dijkstra Algorithm

A graph with blue lines and a white background

Description automatically generated A table with numbers and letters

Description automatically generated

Figure 3.5 Dijkstra Time Efficiency

### A-Start Algorithm

A graph with blue lines and white text

Description automatically generated A table with numbers and letters

Description automatically generated

Figure 3.6 A\* Time Efficiency

Testing Evaluation

As shown in in sections above

Reference