Develop an AI Agent Using Goal Oriented Action Planning that can Demonstrate Effective Solution Finding in Distinct World Environments

Samuel Buzz Appleby

Student Number: 170348069

11/05/2020

BSc Computer Science

Supervisor: Giacomo Bergami

Word Count: 14,984

Abstract

This project aims to investigate and implement an Artificial Intelligence (AI) system that is used across the games industry, Goal Oriented Action Planning (GOAP). In doing so, I aim to create an agent that can demonstrate varying behaviours across three different environments, dynamically adjusting to its surroundings.

To do this, I will track the paths that the AI follows with the use of state modelling and manipulate the world in ways such that new pathways must be explored. The resulting AI will traverse all possible pathways to a solution; if no pathways can be further explored then we reach an impasse and the goal is unattainable.

Declaration

“I declare that this document represents my own work except where otherwise stated”

Acknowledgements

I would like to thank my supervisor Giacomo Bergami for his advice and assistance throughout my project, in addition to Dr. Neil Speirs and Dr. Richard Davison for their help, it is enormously appreciated.

Contents

CHAPTER 1: INTRODUCTION 13

1.1 BACKGROUND 13
1.2 PURPOSE 13
1.3 PROJECT AIM 14
1.4 PROJECT OBJECTIVES 14
1.5 APPROACH 15

CHAPTER 2: RESEARCH 17

2.1 RESEARCH STRATEGY 17
2.2 GOAL ORIENTED ACTION PLANNING 17
2.21 Evolution of AI 17
2.3 AUTOMATA THEORY 19
2.31 Details of Automata 19
2.32 Finite State Machine 19
2.33 Pushdown Automata 23
2.34 The Turing Machine 25
2.4 GRAPH SEARCHING ALGORITHMS 27
2.41 Theory 27
2.42 A Algorithm 29*
2.43 Dijkstra's Algorithm 34
2.44 Bellman-Ford Algorithm 36
2.5 AREAS OF INTEREST 41
2.6 SUMMARY 41

CHAPTER 3: DESIGN AND IMPLEMENTATION 43

3.1 EXISTING SOLUTIONS 43
3.2 CHOICE OF AUTOMATON 43
3.21 DFA (FSM) 43
3.22 PDA 44
3.2.3 TM 44
3.24 Summary 45
3.3 CHOICE OF GRAPH SEARCH ALGORITHM 45
3.31 Uninformed Algorithms 46
3.32 Informed Algorithms 46
3.33 Summary 48
3.4 TOOLS AND TECHNOLOGIES 49
3.41 Finite State Machines with Variables 49
3.42 Graph Representations 50
3.44 Pathway Validity 53
3.45 Unity Game Engine 54
3.5 SUMMARY 57

CHAPTER 4. TESTING 59

4.1 METHOD 59
4.11 Correctness 59
4.12 Deadlock States 59

4.2 Strategy 61
4.21 Make Campfire *61*
4.22 Make Furnace *65*

CHAPTER 5: RESULTS AND EVALUATION 67

5.1 Computational Complexities 67
*5.11 A** *67*
5.12 Dijkstra *67*
5.13 Bellman-Ford *67*
5.14 Summary *67*
5.2 Correctness and Performance 68
5.3 Comparison Between Environments 71
5.4 Framerate 74
5.4 Summary 76

CHAPTER 6: CONCLUSION 77

6.1 Satisfaction of Aims and Objectives 77
6.11 Satisfaction of Objectives *77*
6.12 Satisfaction of Aim *78*
6.2 Improvements for the Future 78

REFERENCES 81

Resources 81
Scripts 84
Assets 84

APPENDICES 87

Personal Resources 88

Table of Figures

Figure 1: Time Plan 15
Figure 2: Nim Game Computer Visual [5] 17
Figure 3: GOAP Action Plan [7] 18
Figure 4: Overview of Automata [9] 19
Figure 5: FSM of a Turnstile [14] 20
Figure 6: DFA Automata Representation [15] 20
Figure 7: FSM after partial code implementation [16] 21
Figure 8: F.E.A.R.'s State Machine [17] 22
Figure 9: Pushdown Automaton [18] 23
Figure 10: Graph of a DFA [20] 24
Figure 11: A Turing Machine [22] 26
Figure 12: Input Tape of a Turing Machine [23] 26
Figure 13: State Diagram of Inverse Function [23] 27
Figure 14: Action Plan with Multiple Paths [7] 27
Figure 15: Action Plan with Associated Costs [7] 28
Figure 16: Neighbouring Nodes [7] 29
Figure 17: Manhattan Distance [25] 30
Figure 18: Euclidean Distance [25] 30
Figure 19: A* Pseudocode [25] 31
Figure 20: A* Example [26] 32
Figure 21: Dijkstra's Algorithm Pseudocode [30] 34
Figure 22: Dijkstra's Algorithm Example [31] 35
Figure 23: Bellman-Ford Pseudocode [33] 37
Figure 24: Negative-Weight Cycle [32] 37
Figure 25: Bellman-Ford Example [34] 38
Figure 26: Automata Theory [4] 43
Figure 27: Automata Characteristics Summary [37] 45
Figure 28: Bellman-Ford vs Dijkstra [40] 46
Figure 29: Case 1, Dijkstra vs A* [41] 47
Figure 30: Case 2, Dijkstra vs A* [41] 47
Figure 31: A* vs Dijkstra [41] 48
Figure 32: FSM of AI 49
Figure 33: UML Class Diagram 51
Figure 34: Graph, Wood = 0 52
Figure 35: Graph, Wood = 3 52
Figure 36: Resource Checking 53
Figure 37: Setting Resource Availability 53
Figure 38: Game World Selection 54
Figure 39: Desert Island 55
Figure 40: Forest 55
Figure 41: Arctic 56
Figure 42: Tool Checking 56
Figure 43: Eat Action Condition 57
Figure 44: Process Indicator 57
Figure 45: State Diagram 59
Figure 46: UI Resource Feature 60

FIGURE 47: UI PROCESS FEEDBACK 60
FIGURE 48: GRAPH, 0 WOOD 61
FIGURE 49: CSV, WOOD = 0 63
FIGURE 50: GRAPH, WOOD = 2 64
FIGURE 51: CSV, WOOD = 2 64
FIGURE 52: GRAPH, STONE = 2 65
FIGURE 53: CSV, STONE = 2 65
FIGURE 54: GRID, STONE = 3 66
FIGURE 55: CSV, STONE = 3 66
FIGURE 56: GRAPH, WOOD = 0 69
FIGURE 57: GRAPH, WOOD = 5 69
FIGURE 58: GRAPH, WOOD = 2 70
FIGURE 59: TIME TAKEN FOR ALL RESOURCES 70
FIGURE 60: AVERAGE RUN TIMES 71
FIGURE 61: GRAPH, WOOD = 3 71
FIGURE 62: AVERAGE RUN TIME ACROSS WORLDS 72
FIGURE 63: AVERAGE RUN TIMES FOR DIFFERENT WORLDS 73
FIGURE 64: AVERAGE FRAMERATE (IDLE) 74
FIGURE 65: AVERAGE FRAMERATE (MAKECAMPFIRE) 74
FIGURE 66: AVERAGE FPS COMPARISON 75
FIGURE 67: STATE MACHINE 79

Chapter 1: Introduction

This chapter will contain information of my initial research into my field of study. I will highlight what I hope to achieve with my finished project.

1.1 Background

Almost all video games depend upon AI. In many of these games, Non-Playable Characters (NPCs) have objectives, or goals, they must fulfil. NPCs are diverse; they can be friendly, hostile or exist purely to give an environment life. For almost all these characters, they will have a goal, ranging from killing the player-controlled character (for example a hostile enemy in a first person shooter (FPS)), to moving back and forth between two points (such as an NPC guarding an area).

NPCs can have multiple objectives, and the complexity varies depending on the AI being developed. Their prominence ranges between games; for example, a massively multiplayer online game will have less influence by AI than a real time strategy game, but it exists, nonetheless.

It is important, therefore, that these characters are functioning to their requirements; a player does not want to be stuck in a level because the enemies cannot correctly find their way towards them.

This is the idea behind Goal Oriented Action Planning (GOAP). GOAP is an AI system that allows agents to fulfil certain goals when specific preconditions are met. This design strategy was invented by developer Jeff Orkin for the game F.E.A.R. [1] where the developers wanted to create 'a 'cinematic' experience' [2].

A good example is an enemy in a First-Person Shooter (FPS). Their aim (goal) is to kill the player. However, they first must be within a certain range and have the ammunition necessary to shoot (if such implementation exists). These are the preconditions: they MUST be met for the goal to be achieved. Interestingly, if the same goals are applied to different agents, you can get a 'completely different sequence of actions' [3]. Often there are several paths that can lead to these different outcomes, using the previous example there could be these potential paths:

1) Within range of player? -> Shoot -> Player's Hit points fall to 0? -> Goal achieved

2) Within range of player? -> Grenade Thrown - > Player's Hit points fall to 0? -> Goal achieved

1.2 Purpose

This development method is powerful and is used across the games industry. For this reason, I am going to develop my own GOAP system. I want to be able to create an environment where I can show off agents that are developed in this way. Unlike other projects, I will go deeper and explore my scripts in varying scenarios, where the goal may be unreachable. This will involve exploring all the possible paths for the current objective, in addition to placing the AI in situations where the objective is unfeasible as soon as it enters the world (such as finding water in a desert), or it becomes stuck down a certain path and must backtrack to find other possible solutions. In this way I hope to achieve a robust set of scripts where the behaviour is consistent and deterministic.

1.3 Project Aim

My main aim in tackling this project is to develop an AI agent using goal oriented action planning that can demonstrate effective solution finding in distinct world environments. This involves the AI performing correctly and consistently: I should be able to predict the outcome of a given scenario and repetitions of identical environments should yield the same results.

1.4 Project Objectives

1) *Identify and evaluate three different uses of GOAP currently being used in the games industry.*

This will involve researching papers that have been produced recently enough for the content to be relevant to the development of modern AI. I will read and analyse their methods of development and identify areas of neglect, focussing mainly on types of testing.

2) *Determine the fields of GOAP that have not been thoroughly assessed.*

Having researched into existing papers on the topic, I was able to target a specific area that previously has been missed. For this project I will be interested in the evaluation of the AI and ensure that the tests undergone are thorough and varied.

3) *Plan and create distinct game worlds and scenarios.*

I will create 3 game worlds: Desert Island, Forest, and Arctic. This will allow me to explore all the possible paths that the agent can follow, by varying the amount of resources across them.

4) *Create at least 2 goals which can be achieved through various pathways.*

Establish the objectives of the AI and create the pathways through which they can be achieved. My implementation includes three goals, two of which can be achieved through various pathways: MakeCampfire() and MakeFurnace().

5) *Research and implement an effective testing strategy for the AI.*

To gauge the success of my development, I will need to discover a testing pattern that suits my needs. Multiple tests will be observed and repeated, ensuring my program performs correctly.

6) *Debug and evaluate the agents and their corresponding scripts.*

Having completed all the required testing, I will inspect my findings. This will involve drawing mock-ups of how the AI was expected to perform (i.e. the path that should have been followed) and compare it to the actual result. Here I will conclude the success of my design.

1.5 Approach

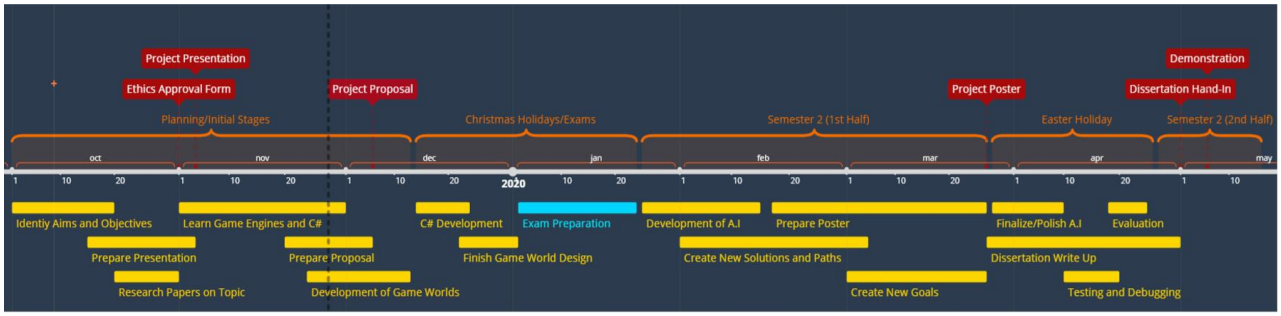


Figure 1: Time Plan

Figure 1 represents my work plan and how I aim to structure the time spent on this project. Within each work slot I will be tackling the aforemnetioned objectives. Tasks may pertain to multiple objectives, so to these I have not given any strict deadlines. The game engine software I will be using for designing and testing the AI will be Unity. The scripts used to implement GOAP will be coded in C# and programmed in the Visual Studio Integrated Development Environment (IDE) [4].

Chapter 2: Research

In this chapter I will perform literature reviews of existing papers on the topic. I will investigate the origins of this AI system and analyse its practicality.

2.1 Research Strategy

To conduct my research, I will search for a variety of sources including:

- Research Papers
- Academic textbooks and articles
- Reports and news
- Webpages

For the formal research (papers, journals) I will be using Google Scholar, including international publications.

Any other research will be carefully selected, majority consisting of online materials.

All information gathered and used in this document will be referenced accordingly.

2.2 Goal Oriented Action Planning

2.21 Evolution of AI

Since the introduction of video gaming, AI has been in continuous development. One of the earliest examples of AI in games is *Nim,* released in 1952.

This was the software version of an ancient board game, requiring two players to take it in turns to remove at least one object from a set number of rows. The winner (or loser) was decided by whoever took the last object [5].

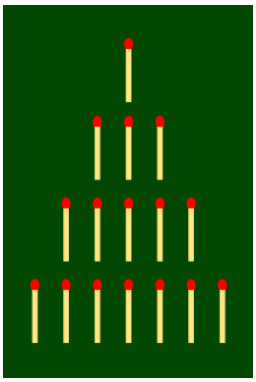


Figure 2: Nim Game Computer Visual [5]

The simple ruleset of this game is one of the reasons it was the first of its nature to be transferred across to software. The video game adaptation was not purely designed as a multiplayer game, it was among the first to have a functioning AI that would act as the opponent.

'the game took the form of a relatively small box and was able to regularly win games even against highly skilled players of the game' [6]

The development followed a different development method in place of GOAP. This is often the case, because in a game like Nim we cannot possibly represent all possible states the game could take, and so limits the way we can develop the AI; we must find an alternative design strategy.

However, almost 70 years has passed, and AI in games often have different requirements than in games like Nim or Chess. During this time, there has been the need for the introduction of different techniques when designing various AI.

The conditions for GOAP are:

- Different pathways the AI can take to reach its goal; this is represented as a graph such that a graph search algorithm can be applied (e.g. A*)

- Preconditions, for example world resources (ammunition, distance from player), and actions (chopping down a tree, shooting the player) that transfer the AI from one state to the next; represented as a state machine

Here is how they are combined:

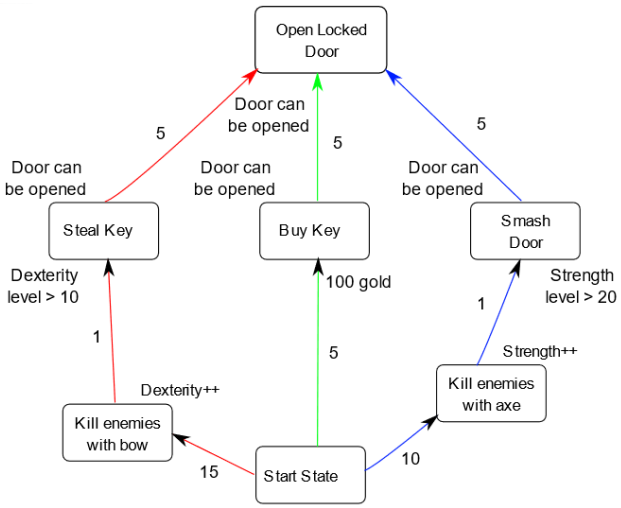


Figure 3: GOAP Action Plan [7]

Figure 3 represents a state machine where all actions have their corresponding cost. The goal of the AI is to open the locked door, but there are three preconditions, one of which must be met, the AI either: steals a key, buys a key or smashes down the door. It is up to the graph search algorithm to decide which path has the lowest associated cost. Having found this route, the program will then command the agent to take the actions necessary to reach each state in the route.

The AI makes smart decisions on which pathway should be taken; it not only proceeds down the one with the lowest cost, but also a route that is reachable by traversing between states.

F.E.A.R. was the first video game that used GOAP as its design architecture, led by developer Jeff Orkin, and has been implemented in many games since, such as Bethesda Softworks' Fallout 3 and Just Cause 2 by Avalanche Studios [8].

Fundamentally, GOAP is a combination of a pathfinding algorithm with a state machine.

2.3 Automata Theory

2.31 Details of Automata

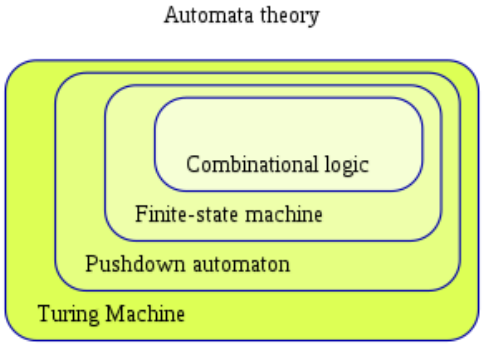


Figure 4: Overview of Automata [9]

To implement GOAP in our game, we must represent the gaming environment as a state machine. This is because without having a model, there is no way to calculate the associated cost with each action, and our AIs behaviour will become non-deterministic (effectively, the path that the AI takes will be close to random). These machines branch from an area of computer science called *automaton theory*.

Automata theory is the study of abstract machines and automata [10]. The AI represent the automata and the model we design to represent the game world is our abstract machine. The automata can update the current state on a given set of conditions and what is held in the current configuration.

FSMs are the least expressive machines that are included in automata theory, the others can be seen in Figure 3. The scale and structure of the AI we are developing influences the type of machine we will implement, with Turing Machines being the most advanced model.

2.32 Finite State Machine

The most primitive version of automata modelling is a Finite State Machine (FSM). The characteristics are:

- Inputs: assumed to be sequences of symbols selected from a finite set I
- Outputs: sequences of symbols selected from a finite set Z
- States: finite set Q, whose definition depends on the type of automaton [11]

Formally, an FSM is a finite automaton (FA) [12], defined mathematically as a quintuple (A, S, s, T, F), where:

- A is a finite non-empty set of symbols (input alphabet)
- S is a finite non-empty set of states
- s is an initial state, an element of S
- T is the state transition function: T: S x A -> S
- F is the set of final states, which is a subset of S [13]

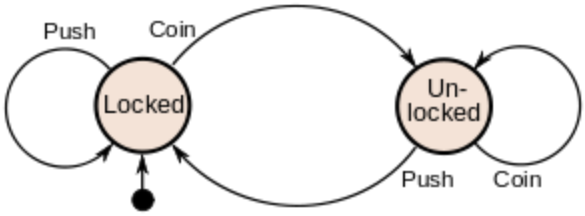


Figure 5: FSM of a Turnstile [14]

One type of FSM is the Deterministic Finite Automata (DFA) [15]. These are state machines that accept or reject strings of a language. A DFA must have:

- An alphabet that defines all symbols
- N possible non-empty states
- One Initial State
- A set of transactions
- One or more final states

A DFA can be represented as such:

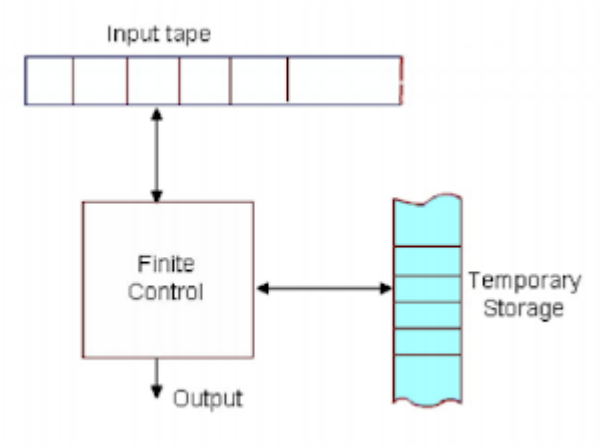


Figure 6: DFA Automata Representation [15]

The inputs received take the form of a tape. This tape is divided into cells, each holding a symbol. The control unit, which will be in one of finite number of states, can read these symbols and change its state, but cannot write.

All FSMs regarding this project accept or reject forms of 'strings', so the words DFA and FSM are interchangeable here forth.

In video games, our AI can be thought of as an agent that moves between the states of a system: we may want an enemy to move towards a player when not in shooting range, but also to flee when the player throws a grenade. We may also want the AI to have objectives that can be represented as another state machine entirely, for example a 'guard' status where the agent would simply move among a set of points.

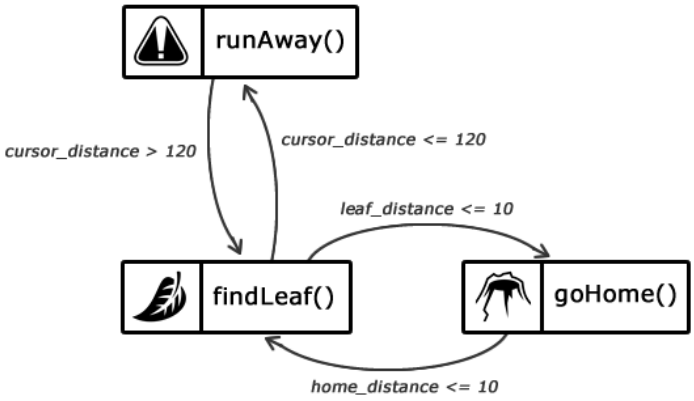


Figure 7: FSM after partial code implementation [16]

An example of how an agent could behave can be seen in Figure 7, representing an ant in a colony that can build a nest with leaves, and run away when close to the mouse cursor.

The nodes represent the states and the edges represent the action. The AI will often establish itself in an *Idle* state, defining the initial behaviour. The initial state here would be 'find leaf()'.

The functions: findLeaf(), goHome() and runAway() are actions causing the agent to perform a specific behaviour. The state findLeaf() is the initial state. Here an algorithm would tell the ant to search a specific area.

The AI will exist in this idle state until one of the other states preconditions is met:

- goHome(): leaf_distance <= 10
- runaway(): cursor_distance <= 120

Creating the preconditions for all other states defines them for the Idle state, the inverse of all immediate neighbouring states' preconditions:

- findLeaf(): leaf_distance > 10 && cursor_distance > 120

Updating the FSM, we must take into considering four properties:

- Reversibility
    - An FSM is reversible if the initial state can be reached from every other reachable state
- Determinism
    - An FSM is deterministic if for every state every unique transaction can move only to itself or at most one other state
- Deadlock
    - A state is a deadlock if it contains no exit arcs
- Deadlock freeness
    - An FSM is deadlock free if none of its reachable states is a deadlock

For example, if we had a 'Dead()' state, we would give it no exit arcs, making this state a deadlock. This would fail the deadlock free condition and make the FSM non-reversible.

When looking at FSMs developed in industry, we can view the one developed by Jeff Orkin for F.E.A.R.:

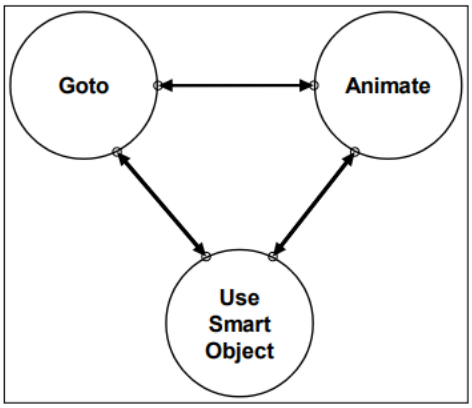


Figure 8: F.E.A.R.'s State Machine [17]

At first glance this appears simple. This is on purpose. The reason why is that whenever an AI changes its state, it changes its animation, and whenever an action happens that changes the position of the agent, it will change to the Goto() state. In the words of Orkin:

'*all A.I. ever do is move around and play animations! Think about it. An A.I. going for cover is just moving to some position, and then playing a duck or lean animation.'* [17]

All animations are different, and some use and manipulate the game world. For example, an enemy firing a gun should create bullet objects that interact with the world, while a duck animation should only have effect on animation. How is this determined in F.E.A.R.'s state machine? Orkin explains:

'Sure there are some implementation details; we assume the animation system has key frames which may have embedded messages that tell the audio system to play a footstep sound, or the weapon system to start and stop firing, but as far as the A.I.'s decision-making is concerned, he is just moving around or playing an animation.' [17]

The respective effects of each individual animation are not handled by the AI, because all the AI knows is when it should move between the three designated states. There are scenarios where the agent needs to head to a designated position, in place of arbitrarily performing an animation at any place in the game world. We therefore have our Goto() state, which gives the agent information about its exact destination:

'In fact, moving is performed by playing an animation! And various animations (such as recoils, jumps, and falls) may move the character. So the only difference between Goto and Animate is that Goto is playing an animation while heading towards some specific destination, while Animate just plays the animation, which may have a side effect of moving the character to some arbitrary position.' [17]

F.E.A.R. was the first game to use GOAP as a design structure for the AI. Other games studios have implemented it since then, and the versatility of FSMs allows for a great range of uses. The diagrams can be high level and easier to design, like in the case of F.E.A.R., where the detailed application of each state can be handled further down. They can be created from the base structure of how we want our AI to behave, as can be seen in Figure 7, which could be abstracted into a more encompassing machine.

2.33 Pushdown Automata

The problem with modelling using a FSM is that we are limited in terms of memory. When we need a more versatile model that solves this issue, we can use a Pushdown Automata (PDA) [18]. PDAs are finite state machines but make use of a stack, giving us infinite memory. As such PSAs can solve any problem that an FSM can.

Like our DFAs, PDAs have:

- An input tape, containing the input string
- A finite control unit, determining state to state movement, expressible as a graph like DFAs

However, to alleviate any pressure we have on memory constraints, we also include:

- A stack with infinite size

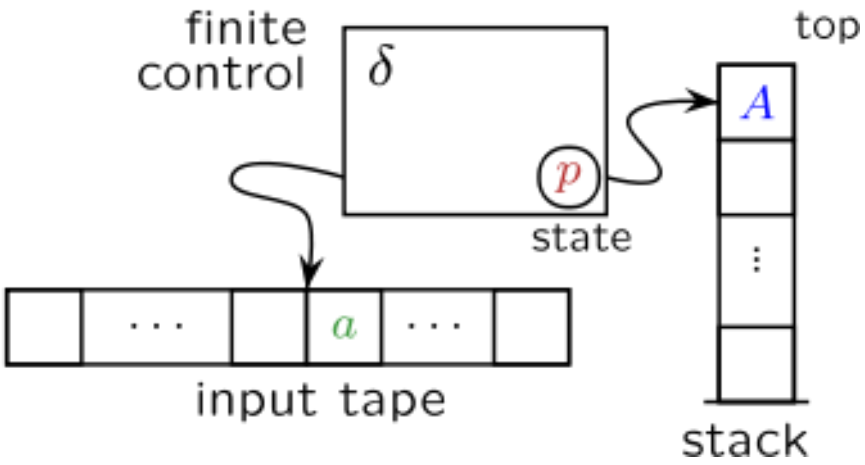


Figure 9: Pushdown Automaton [18]

We now have a stack that we can read from and are given more possibilities as to how we can transition from one state to the next. At any state, the next transaction is decided by reading the current input symbol on top of the stack. Then, depending on this value, the PDA can:

- Pop the top of the stack
  or
- Push new symbols onto the stack

Like FSMs, they can be represented formally, though here we need to consider the stack elements in addition to the states as before. PDAs are a 7-tuple (Q, ∑, S, δ, q0, I, F):

- Q is the finite number of states
- ∑ is input alphabet
- S is stack symbol
- δ is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- q0 is the initial state (q0 ∈ Q)
- I is the initial stack top symbol (I ∈ S)
- F is a set of accepting states (F ∈ Q) [19]

With our definition complete, we need to then create the transition., These take the form:

a, b -> c

Where:

- a is the current input symbol
- b is the symbol on top of the stack
- c is the symbol that we push on top of the stack

For example, the transition 0, 1 -> 00, means if we are reading the symbol 0, and the symbol on top of the stack is 1, then pop 1 and push 00 to the top of the stack. At the bottom of our stack we always define an initial symbol, $. This allows us to define a transition to deal with an empty stack, an identify when no symbols remain.

For example, lets define some transitions:

1) Whenever we see a 0, push it onto the stack
2) Whenever we see a 1, pop the corresponding from the stack
3) When input is consumed, if the stack is empty, accept [20]

Having defined the transitions, we can then represent our PDA as a graph:

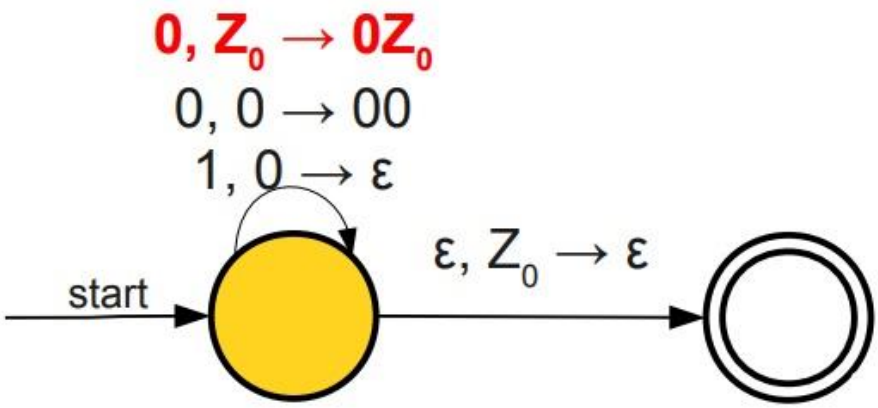


Figure 10: Graph of a DFA [20]

When defining our transitions, we must consider all possible combinations of symbols that could appear in the input string, and the symbol currently on top of the stack.

1) Whenever we see a 0, push it onto the stack:

In Figure 10, it would appear that all combinations of when we read 0 are not covered; our alphabet consists of {0, 1, Z_0} so we would require three transitions but only have two (0,Z_0 -> 0Z_0 and 0, 0 -> 00). However, if we inspect all the transitions, we can see that we never actually add 1 to our stack, so therefore checking if it lies on top is not needed. We implicitly cover all situations where we would be reading a 0.

2) We have said that whenever we see a 1, we should pop the corresponding 0:

We can allow the DFA to fail when a situation occurs that is not covered by a transition. The transition 1, 0 -> Ɛ does not cover all possibilities. Here, we have two instances that would cause the DFA to fail: 1, 1 -> X and 1, Z_0 -> X (where X represents any valid symbol). No transition is defined that deals with this case, and the model fails.

By allowing our model to fail under certain circumstances, we have no memory restrictions and more versatility with PDAs compared to DFAs. PDAs can be used for GOAP when our models grow too large.

2.34 The Turing Machine

The Turing Machine (TM) is a machine invented by Alan Turing in 1936. The machine is simple, but it aims to simulate any computer algorithm, no matter the complexity [21].

A Turing machine consists of:

- An infinitely long piece of tape (acting like the computers' memory)
- A tape head pointing to the currently inspected cell of memory
- A state transition table that controls the behaviour of the machine

Below is a diagrammatic example of a Turing machine:

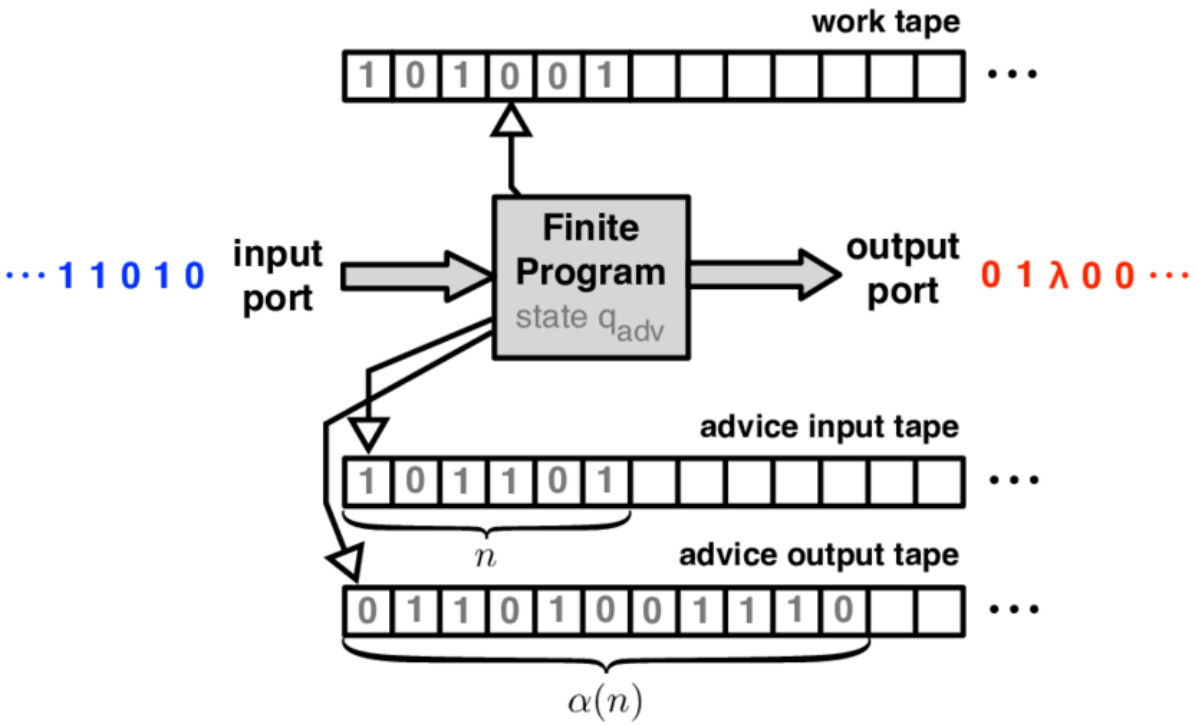


Figure 11: A Turing Machine [22]

One of the main utilities of a TM is that it can write to the tape, for both input and output. The piece of tape does can be initially blank, or already hold symbols. If our TM is computing a function that requires an input, it could also start with a non-empty tape.

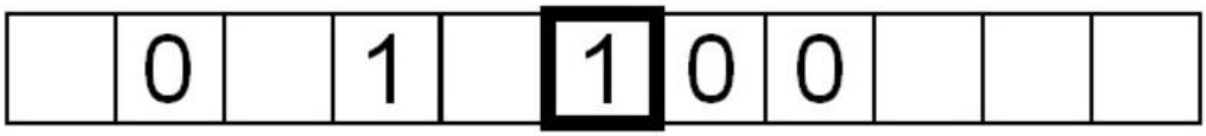


Figure 12: Input Tape of a Turing Machine [23]

Figure 13 shows how a tape would look after running a specific program that writes bits to the tape and then navigates left or right by following the programs instructions. At any point, the machine can do one of three things:

- Read – Inspect the value in memory of the cell currently pointed to by the tape head
- Write – Modify the cell by changing its value to any valid substitute
- Move – Tell the program to move the tape left or right so it can read a neighbouring cell

We need to ensure that the table is stable and avoid any cases where we are in an unsolvable state. Like PDAs and FSMs, if our model receives an input that is not recognised, the model may fail. For example, if we were to write a program that told the machine to inverse the values of the cells without defining what to do when reaching a blank state, then it would continuously read this blank state forever.

Here we need to consider the *machine state,* i.e. we can represent the behaviour with a state diagram:

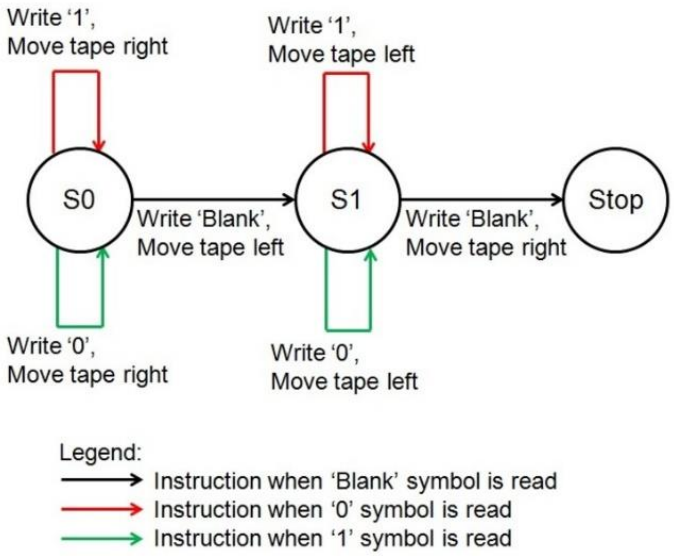


Figure 13: State Diagram of Inverse Function [23]

With a state machine defined like Figure 13, we can solve the problem of any infinite loop that our program should encounter. By representing it in this way, it is much easier to identify any possible flaws in our program.

2.4 Graph Searching Algorithms

2.41 Theory

So, we have represented our AI as a state machine using an appropriate model from automata theory. All different states have been constructed as nodes and actions as transitions. Here, however, we reach a problem:

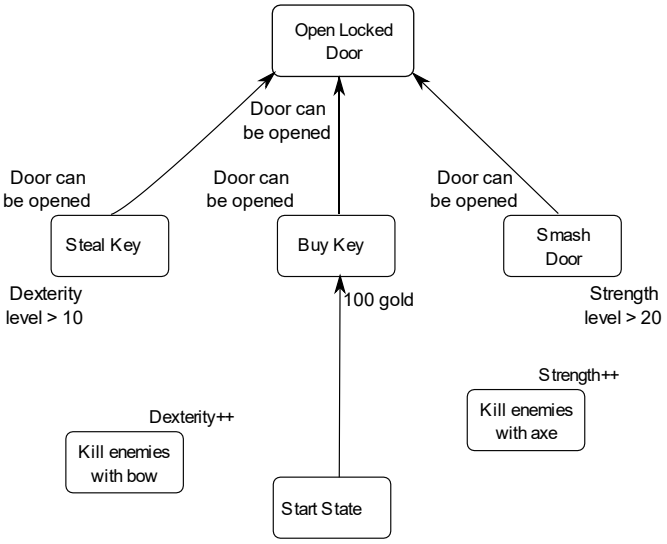


Figure 14: Action Plan with Multiple Paths [7]

This is the action plan mentioned in Section 2.21, where the goal is ‘Open Locked Door’. If we work backwards from the final state, there are multiple paths that have the desired outcome: ‘Steal Key’, ‘Buy Key’ or ‘Smash Door’. The agent can choose to follow any one of these paths and it will succeed in its goal.

Here lies the problem. As far as the AI is concerned, taking any of these pathways is a valid option; there is nothing that indicates the cost of taking each action. Therefore, when we run the program, our AI will show non-deterministic behaviour, it will pick any of the three paths without consideration of the resources required to do so (or it will always follow the same path, depending on the structure of the code).

Running an AI composed in such a way would result in poor behaviour. For example, if the cost of taking the pathway via ‘Smash Door’ is much less than that of ‘Steal Key’ and ‘Buy Key’, then we don’t want our AI to use up valuable time and resources performing the more expensive actions. To solve this, we attribute costs to each action, followed by applying a graph searching algorithm, which will allow the AI to choose the least costly path.

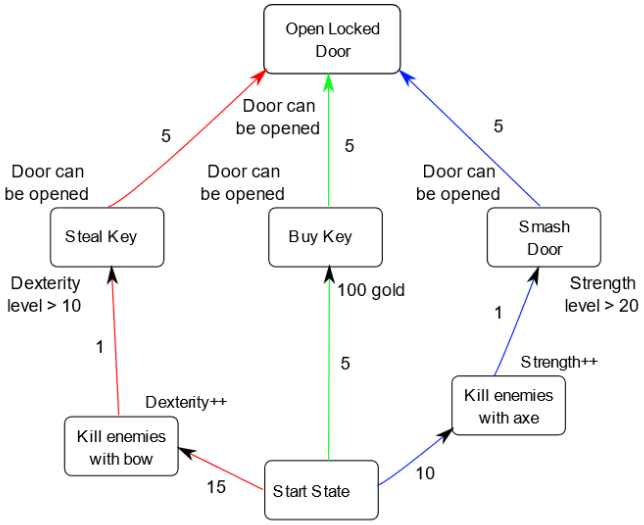


Figure 15: Action Plan with Associated Costs [7]

The cost given to each transition is decided by the developer, choosing the actions that are the most expensive. From Figure 15, we can see that ‘Kill enemies with bow’ has a cost of 15, while ‘Kill enemies with axe’ has a cost of 10. This would be the first step, however, because we can see that we only need > 10 dexterity to perform ‘Steal Key’ while we need > 20 strength for ‘Smash Door’. The cost of opening the door itself is identical no matter which path is taken. In addition to this, ‘Buy Key’ only has a cost of 5, making it the lowest cost, but the agent must fulfil the precondition that it has 100 gold.

We need to start calculating the cost of all possible routes. These calculations must consider the current state and its variables, here the Strength and Dexterity fields. Performing such a calculation can be costly, and so there are different algorithms that try to achieve the best balance between performance and accuracy.

2.42 A* Algorithm

The A* algorithm is one of the most popular graph searching algorithms. For every single node, if there exists a path to the destination, A* will find it. Because A* will check every single path to see if it can be improved; it fulfils two important properties of search algorithms: optimality and completeness.

'*When a search algorithm has the property of optimality, it means it is guaranteed to find the best possible solution. When a search algorithm has the property of completeness, it means that if a solution to a given problem exists, the algorithm is guaranteed to find it*' [24]

For every node, we need to keep a track of the current cost from the start node to it, as well as the current cost of the search iteration. Whenever we reach a node, we consider all of those in its neighbourhood, i.e. the nodes that it can reach in one transition:

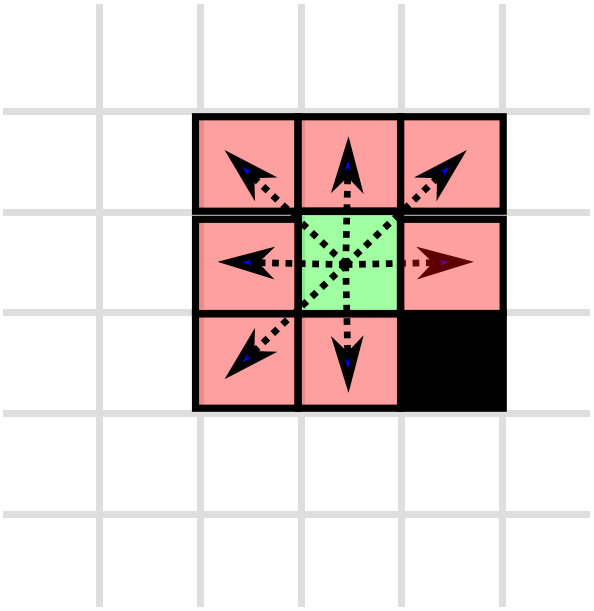


Figure 16: Neighbouring Nodes [7]

Some nodes in certain directions may be impossible, represented here by the black area. When A* chooses a node that it believes to be the best option, it will 'expand it' i.e. traverse to this node and then consider all this node's neighbours.

To make accurate calculations on which path we should follow, we need some more variables:

- g(n) — the exact cost of the path from the starting node to any node n
- h(n) — the heuristic estimated cost from node n to the goal node
- f(n) — lowest cost in the neighbouring node n [24]

Calculating g(n) is straightforward, we already know the costs to each neighbouring cell, so this is the sum of the costs to reach the node so far.

To find h(n), we have two options, we can either calculate the:

- Exact heuristic – this involves computing the distance between every pair of cells before running the algorithm, very costly
- Approximate heuristic – using one of two methods to calculate a rough estimate from the current node to the goal node [25]

Because calculating the exact heuristic is inefficient, we will consider other approximations. There are two methods for this:

Manhattan distance:

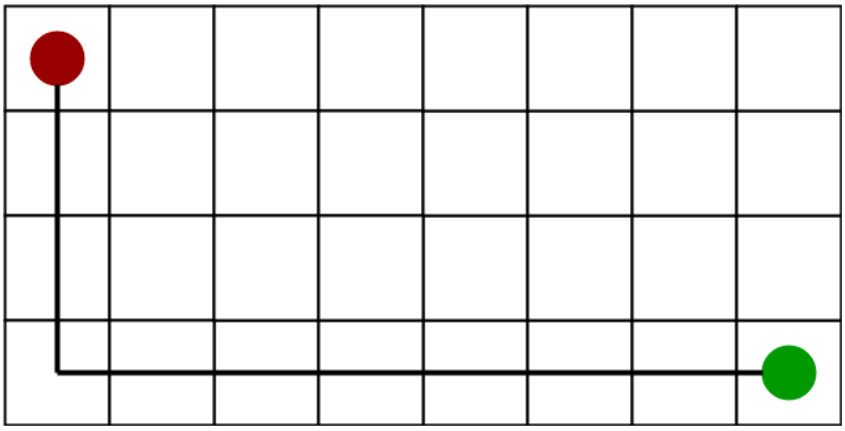


Figure 17: Manhattan Distance [25]

Manhattan distance is purely just the sum of the absolute difference between the x and y coordinates between our source and destination node:

```
h = abs(current_cell.x - goal.x) + abs(current_cell.y - goal.y)
```

Euclidean distance:

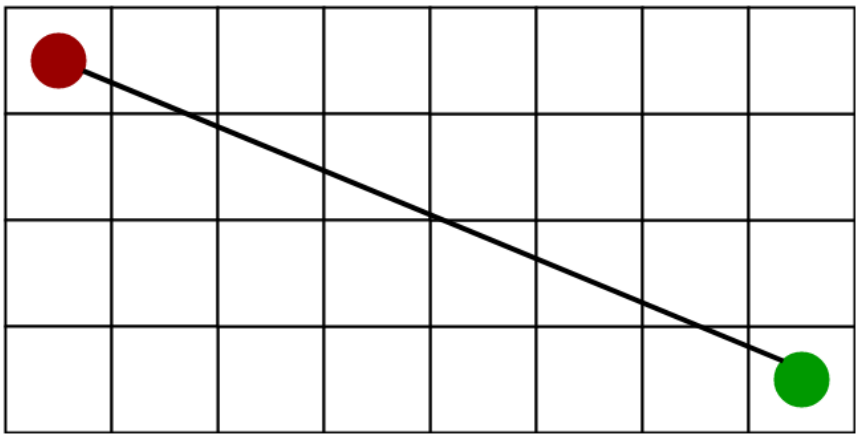


Figure 18: Euclidean Distance [25]

Euclidean distance is the exact length between the two nodes. It is more costly than Manhattan distance due to the use of a square root:

```
h = sqrt (pow((current_cell.x – goal.x), 2) + pow((current_cell.y – goal.y), 2))
```

Now that we have our heuristic, we can calculate f. This is the value that the A* uses to decide which node should be chosen to expand next, and is calculated as:

- f(n) = g(n) + h(n)

Implementation

Pseudocode:

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iii) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```

Figure 19: A Pseudocode* [25]

Whenever we view a node that has already been considered on the open list, its f value changes based on the current cost of the path taken to it (its heuristic will never change). Consider the graph:

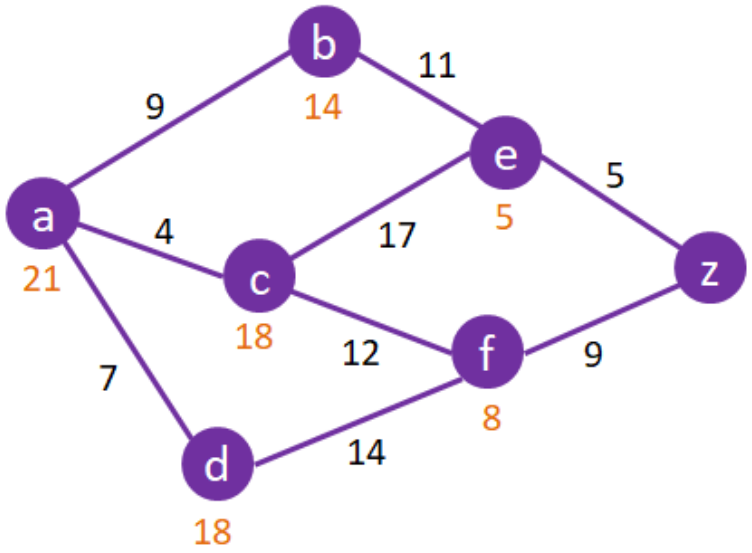


Figure 20: A Example [26]*

Take the node e, it can be reached by both b and c:

If we then took the path a -> c the f value would be 4 + 17 + 5 = 26

If we were to expand b at any stage, the f value of e here would be 9 + 11 + 5 = 25.

Here, traversing to e via b is cheaper than travelling via c, as 9 + 11 < 4 + 17. If any f values subsequently calculated on an already expanded node would result in a worse value, we ignore it.

Example

Final Table:

A		B		C		D		E		F		Z	
0	-	-	-	-	-	-	-	-	-	-	-	0	-
0	-	23	a	22	a	25	a	-	-	-	-	0	-
0	-	23	a	22	a	25	a	26	c	24	c	0	-
0	-	23	a	22	a	25	a	25	b	24	c	0	-
0	-	23	a	22	a	25	a	25	b	24	c	25	f

A step by step process using A* on the graph above would be as follows (we order the nodes by their associated f costs, from lowest to highest, called a priority queue):

1) Expand and pop a. Nodes b, c and d have newly acquired f values:

   c: (4 + 18 = 22) via a
   b: (9 + 14 = 23) via a
   d: (7 + 18 = 25) via a

2) Lowest f value is c, this becomes our current node

3) Expand and pop c. Nodes e and f have newly acquired f values:

   b: 23 via a
   f: (4 + 12 + 8 = 24) via c
   d: 25 via a
   e: (4 + 17 + 5 = 26) via c

4) Lowest f value is b, this becomes our current node

5) Expand and pop b. Nodes e has already been visited, but we find a better new cost:

   f: 24 via c
   d: 25 via a
   e: 26 via c -> (9 + 11 + 5 = 25) via b

6) Lowest f value is f, this becomes our current node

7) Expand and pop f. Nodes z has a newly acquired f values. Check to see if d should be updated:

   d: 25 via a -> (4 + 12 +14 + 18 = 48) via f, 48 < 25 DO NOT UPDATE
   e: 25 via b
   z: (4 + 12 + 9 = 25) via f

We can see that going from d -> f will not result in an optimal solution, but the algorithm must check it anyway because there could be a possible solution, hence the calculation above.

We have now found one path to z, going from a -> c -> f -> z, but the f value associated with this is the same as the other unchecked nodes. We cannot end the algorithm until this floats to the top of our priority queue. So, we must continue from the top of our priority queue:

8) d is at the top of our priority queue, becoming our current node

9) Expand and pop d, all neighbours have already been added to the closed list, so we know we have found the best path and no need to update or do any other calculations:

   e: 25 via b
   z: 25 via f

10) e is at the top of our priority queue, becoming our current node

11) Expand and pop e, z is still on the open list so check to see if z should be updated:

   z: 25 via f -> (25 + 5) via e, 30 > 25 DO NO UPDATE

We are now left with only one pathway left on our priority queue:

   z: 25 via f

So, to determine the shortest path we backtrack through the table, resulting in:

z -> f -> c -> a, Length = 25

2.43 Dijkstra's Algorithm

Another graph search algorithm is Dijkstra's algorithm. Unlike A*, this algorithm is uninformed [27]. This means that there is no knowledge about any previous costs of prior and future nodes (until it is a neighbour).

Because of this, there is no way we can estimate the distance between the current node and the destination. With A*, we could make decisions based on the current travel distance in and the estimated distance to the destination node. With Dijkstra, we only know the most immediate distances, and must perform these calculations spontaneously.

At every iteration step, the algorithm will choose the path with the least distance, without the knowledge of any future costs. This makes Dijkstra a *greedy* algorithm:

'*a heuristic algorithm that at every step selects the best choice available at that step without regard to future consequences*' [28].

This seems inferior to A*, however it is more optimal when we have multiple target nodes without knowing which is the closest. This is because a larger area of the graph is covered as more steps are required in comparison to A* [29].

Implementation

Pseudocode:

```
function Dijkstra(Graph, source):
       dist[source]  := 0                    // Distance from source to source is set to 0
       for each vertex v in Graph:           // Initializations
           if v ≠ source
               dist[v]  := infinity          // Unknown distance function from source to each node set to infinity
           add v to Q                        // All nodes initially in Q

      while Q is not empty:                 // The main loop
          v := vertex in Q with min dist[v]  // In the first run-through, this vertex is the source node
          remove v from Q

          for each neighbor u of v:          // where neighbor u has not yet been removed from Q.
              alt := dist[v] + length(v, u)
              if alt < dist[u]:              // A shorter path to u has been found
                  dist[u]  := alt            // Update distance of u

      return dist[]
  end function
```

Figure 21: Dijkstra's Algorithm Pseudocode [30]

During the main loop, at any stage a shorter possible route is found to any given vertex, it will be updated. Because we have initialised all distances to infinity, the first time a node is visited, alt < dist[u] will always return true, because dist[u] is ∞.

Any further iteration that reaches the same node will only update the cost if it is smaller than the previous best. In the final run of the algorithm, all nodes will contain the shortest distance to them.

Like with A*, we use a priority queue, ordering unvisited nodes from shortest to greatest distance, visiting the nodes at the top of the queue.

Example

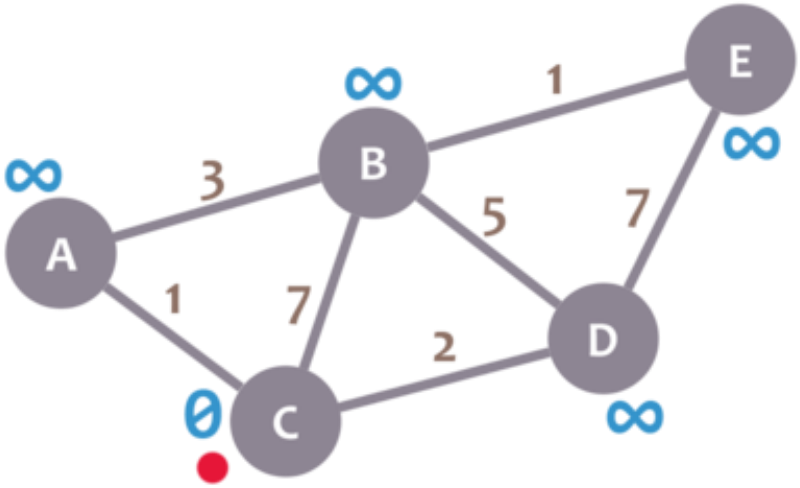


Figure 22: Dijkstra's Algorithm Example [31]

Final Table:

A		B		C		D		E	
∞	-	∞	-	0	-	∞	-	∞	-
1	C	7	C	0	-	2	C	∞	-
1	C	4	A	0	-	2	C	∞	-
1	C	4	A	0	-	2	C	9	D
1	C	4	A	0	-	2	C	5	B

A step by step process using Dijkstra on the pre initialised graph (node queue = Q, visited queue = S) above would be as follows:

1) Choose the node with the shortest distance from C, current node becomes C, remove C from Q, and add to S

2) Calculate distance from source node to all neighbours of C:

   A: 1 via C
   D: 2 via C
   B: 7 via C

3) Shortest distance from source is 1, current node becomes A, remove A from Q and add to S

4) Calculate distance from source node to all neighbours of A:

   D: 2 via C
   B: 7 via C -> (1 + 3 =4) < 7 via A, UPDATE B

Here we have a situation where we have managed to find a shorter path to a node, B here, so we must update the value currently held by dist in B.

5) Shortest distance from source is 2, current node becomes D, remove D from Q and add to S

6) Calculate distance from source node to all neighbours of D:

   B: 4 via A
   E: (2 + 7 = 9) via D

We have now found a path to the destination node; however, we do not finish here as possible shorter paths could exist and we only terminate when there are no more nodes left in Q, or our destination node becomes the top of our priority queue.

7) Shortest distance from source is 4, current node becomes B, remove B from Q, and add to S

8) Calculate distance from source node to all neighbours of D:

   E: 9 via D -> (1 + 3 + 1 = 5) < 9 via B, UPDATE B

By visiting our previously unvisited nodes, we have managed to find a shorter route to our destination, we update this new distance and continue to visit all nodes.

9) Shortest distance from source is 5, current node becomes E, remove E from Q and add to S

By removing the final node contained in Q, we terminate the algorithm. We then backtrack using the table from the destination to the source to find our shortest path:

E -> B -> A -> C, Length = 5

2.44 Bellman-Ford Algorithm

Many graph searching algorithms fail with negative weights. The Bellman-Ford algorithm is one such algorithm that does not. There are lots of instances where this can apply:

'For instance, if there are different ways to reach from one chemical A to another chemical B, each method will have sub-reactions involving both heat dissipation and absorption' [32]

In games, this could be a scoring system, where a players has a numeric variable representing their current score, and actions that take away from this value.

Bellman-Ford works differently to Dijkstra and A*, it does not progressively iterate through the graph finding the shortest distance at every step. Instead, we make a list of all the edges in the graph and on each iteration, we *relax* each edge by calculating the distance to the outgoing node.

By the final iteration we find the shortest distance to *all* other nodes in the graph, not just the destination. We do not need to give the algorithm the destination node, only the source.

After the first iteration, the paths found are unlikely to be efficient, and there may still be unvisited nodes; any paths we do have will likely be an overestimate of the best. We iteratively relax the estimates by finding new paths that are shorter than the previously ones [33].

Implementation

Pseudocode:

```
function BellmanFord(list vertices, list edges, vertex source,
                     distance[], parent[])

// Step 1 - initialize graph. At the beginning, all vertices have a weight of
// infinity and a null parent, except for the Source, where the weight is 0

for each vertex v in vertices
    distance[v] = INFINITY
    parent[v] = NULL

distance[source] = 0
// Step 2 - relax edges repeatedly
    for i = 1 to V-1    // V - No. of vertices
        for each edge (u, v) with weight w
             if (distance[u] + w) is less than distance[v]
                distance[v] = distance[u] + w
                parent[v] = u

// Step 3 - check for negative-weight cycles
for each edge (u, v) with weight w
    if (distance[u] + w) is less than distance[v]
        return "Graph contains a negative-weight cycle"

return distance[], parent[]
```

Figure 23: Bellman-Ford Pseudocode [33]

From the pseudocode, we can see that the algorithm can either return the completed shortest path graph or can return a statement saying if the graph contains a negative-weight cycle:

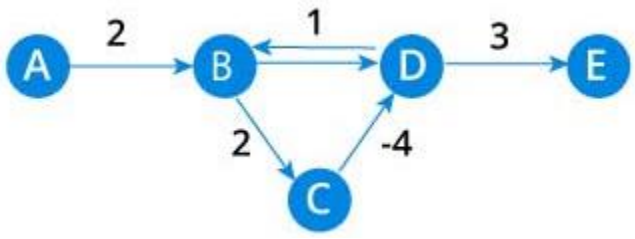


Figure 24: Negative-Weight Cycle [32]

If a negative weight cycle exists, then the algorithm will continuously travers through it, because the cost will continue to get lower.

Here, the cost of path B -> C -> D -> B is (2 + -4 + 1 = -1). This is less than 0, so on every iteration the algorithm will take this path, never reaching the destination E.

Therefore, the check is required (step 3 of Figure 25). We return a statement declaring this cycle has been found, indicating the algorithm is incompatible with the current graph.

Example

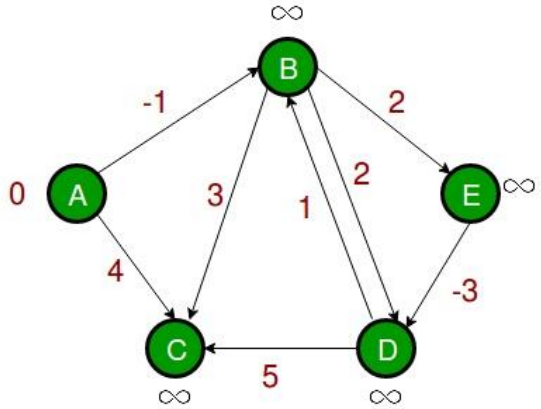


Figure 25: Bellman-Ford Example [34]

Final Table:

A		B		C		D		E	
0	-	∞	-	∞	-	∞	-	∞	-
ITERATION 1									
0	-	-1	A	∞	-	∞	-	∞	-
0	-	-1	A	4	A	∞	-	∞	-
ITERATION 2									
0	-	-1	A	4	A	∞	-	1	B
0	-	-1	A	4	A	-2	E	1	B
0	-	-1	A	2	B	-2	E	1	B

A step by step process using Bellman-Ford on this graph:

- Start Node = A
- Edge Process Order: (B, E), (E, D), (D, B), (B, C), (D, C), (B, D), (A, B), (A, C)

N.B: The order of the edge order *does not matter;* any order will result in the same result after V - 1 iterations.

Iteration 1:

1) Relax edge (B, E):

   A: 0
   E: ∞ + 2 = ∞

2) Relax Edge (E, D):

   A: 0
   D: ∞ + -3 = ∞

3) Relax Edge (D, B):

   A: 0
   B: ∞ + 1 = ∞

4) Relax Edge (B, C):

   A: 0
   C: ∞ + 3 = ∞

5) Relax Edge (D, C):

   A: 0
   C: ∞ + 5 = ∞

6) Relax Edge (B, D):

   A: 0
   D: ∞ + 2 = ∞

7) Relax Edge (A, B):

   A: 0
   B: 0 + -1 = -1 via A

8) Relax Edge (A, C):

   A: 0
   B: -1
   C: 0 + 4 = 4 via A

A		B		C		D		E	
0	-	∞	-	∞	-	∞	-	∞	-
ITERATION 1									
0	-	-1	A	∞	-	∞	-	∞	-
0	-	-1	A	4	A	∞	-	∞	-

This completes our first iteration of the algorithm; we repeat up to a maximum of V – 1 times.

Iteration 2:

1) Relax Edge (B, E):

   A: 0
   B: -1
   C: 4
   E: (-1 + 2 = 1) via B

2) Relax Edge (E, D):

   A: 0
   B: -1
   C: 4
   D: (1 + -3 = -2) via E
   E: 1

3) Relax Edge (D, B):

   A: 0
   B: -1 -> (-2 + 2 = 0) > -1 DO NOT UPDATE
   C: 4
   D: -2
   E: 1

The first time we reach the same node twice, the distance is greater, so we do not update.

4) Relax Edge (B, C):

   A: 0
   B: -1
   C: 4 -> (-1 + 3 = 2) < 4 via B UPDATE
   D: -2
   E: 1

We have managed to find a faster way to C, so update.

5) Relax Edge (D, C):

   A: 0
   B: -1
   C: 2 (-2 + 5 = 3) > 2 DO NOT UPDATE
   D: -2
   E: 1

6) Relax Edge (B, D):

   A: 0
   B: -1
   C: 2
   D: -2 (-1 + 2 = 1) > -2 DO NOT UPDATE
   E: 1

7) Relax Edge (A, B):

   A: 0
   B: -1 (0 + -1 = -1) = -1 DO NOT UPDATE
   C: 2
   D: -2
   E: 1

8) Relax Edge (A, C):

   A: 0
   B: -1
   C: 2 (0 + 4 = 4) > 2 DO NOT UPDATE
   D: -2
   E: 1

This concludes our second iteration, the algorithm performs two more, but we have already computed all minimum distances, so I will not display these.

If our destination is D, with our source A, we backtrack from D using the table to find the shortest path:

D -> E -> B -> A, Length = -2

Our algorithm successfully returned a graph. Had our graph contained a negative-weight cycle, an appropriate message would have been returned.

2.5 Areas of Interest

Regarding Objective 2, I aimed to identify a key area of GOAP that lacked exploration. The papers I found here [17][44] go in depth about the architecture and design techniques behind the system but did not contain thorough information about how the AI can be used across different game worlds entirely. Therefore, I will further investigate this field of research in my own development.

2.6 Summary

GOAP is a powerful tool in AI development. Since its creation by Jeff Orkin for F.E.A.R. in 2005, developers have been influenced by its effectiveness and it has been implemented in games of varying genres.

By representing our AI as a state machine, we can apply a graph search algorithm to find the most efficient path to the destination. We can create agents that respond realistically; they acknowledge the costs of performing actions and will make smart decisions to achieve their goal.

Cases occur where the current path no longer leads to a solution (such as running out of wood for building a campfire). Here, we can backtrack through the graph, attempting to follow other solutions instead. If there are no valid paths to reach the goal, we wait until an event occurs that causes a precondition to be fulfilled (for example waiting for a resource to become available). Our agent is knowledgeable about its environment and will adapt if necessary.

Chapter 3: Design and Implementation

In this chapter I will demonstrate my own implementation of GOAP and adaptive AI. I will discuss the methods used to approach the development and the reasons for any design choices made.

3.1 Existing Solutions

Before developing my program, I researched to see if there were any available attempts at implementing GOAP. The most applicable solution was found here [35]. The reason for not using these resources is because part of my project's aim was to 'Develop an AI Agent…' and feel that, to meet my projects requirements, I should approach the solution from scratch.

3.2 Choice of Automaton

The choice of automaton involved assessing the benefits and drawbacks of each model, identify the requirements for my simulation and decide which would be the most suitable. I will go over those that were discussed earlier in Section 2.3.

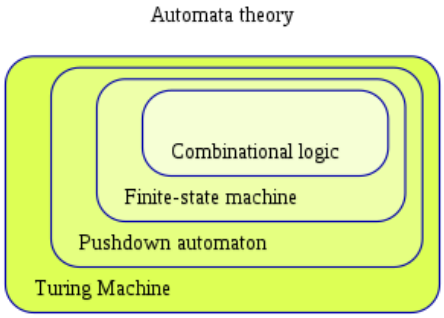


Figure 26: Automata Theory [4]

Game requirements:

1) The ability to read input variables
2) Can change state based on these variables
3) Can modify these and other variables: a memory is contained
4) Can revisit previous states and return to start position

From Figure 26, we can take 3 different approaches: DFA (FSM), PDA or TM.

3.21 DFA (FSM)

DFA are the most fundamental model that could be considered to represent GOAP. Advantages here include fast processing times and less memory usage. They are effective at representing small numbers of actions where the total number of nodes is small. Requirements 1) and 2) are met.

However, when using FSMs, we may encounter one issue, memory. DFAs do not have memory information, and do not have the ability to count the previous events that were met.

This violates Requirement 3) and we cannot use this model. The knowledge of previous states is required and so we must increase the expressiveness of the model, the PDA.

3.22 PDA

As discussed in Section 2.3, PDAs make use of a stack, giving us unlimited memory.

In addition to moving between states, it can also push symbols onto the stack, pop them off the stack, and make decisions based on the symbol on the top of the stack (plus its current state) [36].

Therefore, we have greater flexibility compare with the DFA. While the DFA could only make decisions on the status of its current state, PDAs consider the input symbol, giving us a much greater range of outcomes.

As such, the benefits are using a PDA vs a DFA include:

- The ability to store long sequences of inputs
- Can be constructed for context free (Type 2) grammar
- All DFAs can be simulated using a PDA (meeting Requirements 1) and 2))

Because we have a memory, we can count how many resources the NPC has been collected and can consider our progress towards actions such as building a campfire, meeting Requirement 3). However, with only 1 stack, we count and collect multiple resources, but cannot remember the previous state. This is because we must pop states off the stack to reach the location we want to visit, however those states removed will be lost. This violates Requirement 4).

A PDA can only access the top of its stack and is not useful if we want to access any position within it. In cases where we want to reach any area among its memory, we must advance our model further, into a Turing Machine.

3.2.3 TM

A PDA can only access the top of its stack, whereas a TM can access any position on an infinite tape. The tape cannot be simulated with a single stack, so a TM is much more computationally powerful--there are algorithms that can be programmed with a TM that cannot be programmed with a PDA.

A TM can write to the same tape that it reads from, in addition to changing its state, unlike a PDA which only has reading permission [37].

Turing machines are the most advanced way that we can represent any automata. However, there is one limitation that even the TM cannot address – the Halting Problem [38]. For the TM, the answer to this is undecidable but a PDA can solve it. So, while a TM is more powerful than a PDA, the behaviour regarding the Halting Problem is undefined.

As such, the benefits are using a TM vs a PDA include:

- The ability to move to any position along the input tape
- Can read *and* write to any position on the tape
- All PDAs can be simulated using a TM (meeting Requirement 1), 2) and 3))

As we can move to any location along our data, we no longer must pop memory off the stack. This allows us to traverse along our tape to any location and traverse the same way back. Because of this Requirement 4) is met.

3.24 Summary

As we move from combinatorial logic to TMs, each model gets more expressive power. When deciding which model will be the most effective for our AI (i.e. how expressive should we make our system) we can consider the characteristics of each:

Model/Type	Input tape	Head Direction	Memory
Finite State	Read Only	One way	-
Pushdown	Read Only	One way	Stack
Linear - Bounded	R / W	Two-way direction	(bounded)
Turing Machine	R / W	Two-way direction	(unbounded)

Figure 27: Automata Characteristics Summary [37]

- DFAs are good enough for read only memory where we can only read memory as we receive it
- PDAs are needed when we want to manipulate a stack, giving us unlimited memory and the ability to push and pop items
- When we need to access any element along our tape, giving us the ability to read and write at any position, or would like the ability to write to the tape, we would use a TM

As mentioned in Section 3.23, TMs seem like a suitable candidate for our model. However, TMs are computationally expensive and have too much expressive power for our system. There is a way we can be less expressive while meeting all our games requirements.

This is solved by using a PDA with 2 stacks. With the addition of another stack, we can push the elements onto one stack that are popped from the other. Once we reach our location, we can return these elements back to the other stack, allowing us to return to our start position. Therefore, my solution is best modelled using a FSM associated to some state variables, acting like a less expressive version of a PDA with 2 stacks.

3.3 Choice of Graph Search Algorithm

In Section 2.4 I researched different graph search algorithms which could be implemented in my project. Having considered my possible graph structures, I limited down to three options: A*, Dijkstra and Bellman-Ford.

I chose these because I wanted to research the differences between informed and uniformed algorithms. I also wanted to investigate the efficiency of a greedy algorithm like Dijkstra against a dynamically programmed [39] algorithm like Bellman-Ford.

By working through examples of each, I was able to determine the scenarios where some would outperform the others, and the types of graphs would cause others to fail altogether.

3.31 Uninformed Algorithms

As A* is the only informed algorithm, I first wanted to compare Bellman-Ford and Dijkstra:

BELLMAN FORD'S ALGORITHM	DIJKSTRA'S ALGORITHM
Bellman Ford's Algorithm works when there is negative weight edge, it also detects the negative weight cycle.	Dijkstra's Algorithm doesn't work when there is negative weight edge.
The result contains the vertices which contains the information about the other vertices they are connected to.	The result contains the vertices containing whole information about the network, not only the vertices they are connected to.
It can easily be implemented in a distributed way.	It can not be implemented easily in a distributed way.
It is relatively less time consuming.	It is more time consuming than Bellman Ford's algorithm.
Dynamic Programming approach is taken to implement the algorithm.	Greedy approach is taken to implement the algorithm.

Figure 28: Bellman-Ford vs Dijkstra [40]

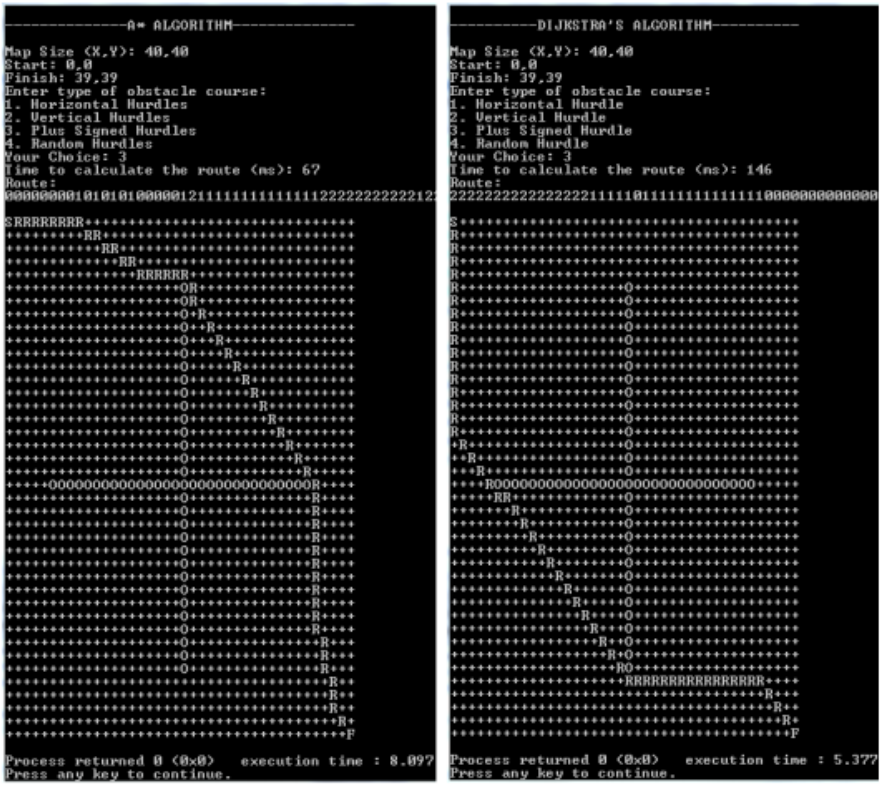
My graph contains no negative weights, and so Dijkstra was the preferred option.

Both algorithms operate in an uninformed way, they do not have the knowledge of the destination nor how far away they are. In AI pathfinding however, we often *do* know the direction that we want the agents to head in, and so it was time to compare Dijkstra to A*.

3.32 Informed Algorithms

This [41] paper goes into detail comparing these algorithms. It involved performing tests on these algorithms using two different graphs:

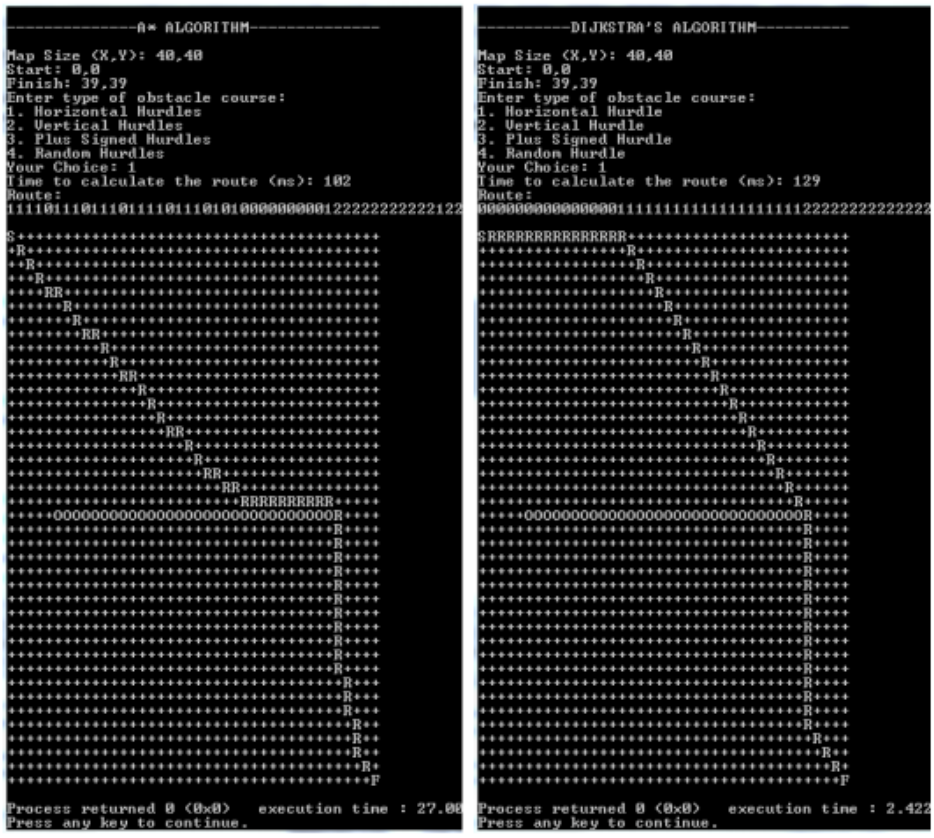
Case 1:



A* Algorithm **Dijkstra's Algorithm**

Figure 29: Case 1, Dijkstra vs A [41]*

Case 2:



A* Algorithm **Dijkstra's Algorithm**

Figure 30: Case 2, Dijkstra vs A [41]*

For both solutions there was a large difference between the time required for each to run:

COMPARISION IN TIME (ms)			
A* ALGORITHM		**DIJKSTRA'S ALGORITHM**	
CASE 1	67	**CASE 1**	146
CASE 2	102	**CASE 2**	129

Figure 31: A vs Dijkstra [41]*

The major disadvantage of Dijkstra's algorithm is the fact that it does a blind search thereby consuming a lot of time waste of necessary resources. [41]

We can see that knowledge of the graph gives us an improvement in computational cost, in both time and memory.

3.33 Summary

I analysed the computational complexity of each algorithm, discussed in Section 5.1. Because the AI would always be aware of its destination, I would always be able to use an informed algorithm. Because of the advantages these have over uniformed algorithms, I used A*. For distance approximation, I will choose Manhattan heuristic. This is due to the better performance than Euclidean, which must compute a square root, rendering it much more computationally expensive.

3.4 Tools and Technologies

3.41 Finite State Machines with Variables

Before being able to apply the A* algorithm, I had to create my state machine containing all possible states the AI could be in and the transitions from one state to another.

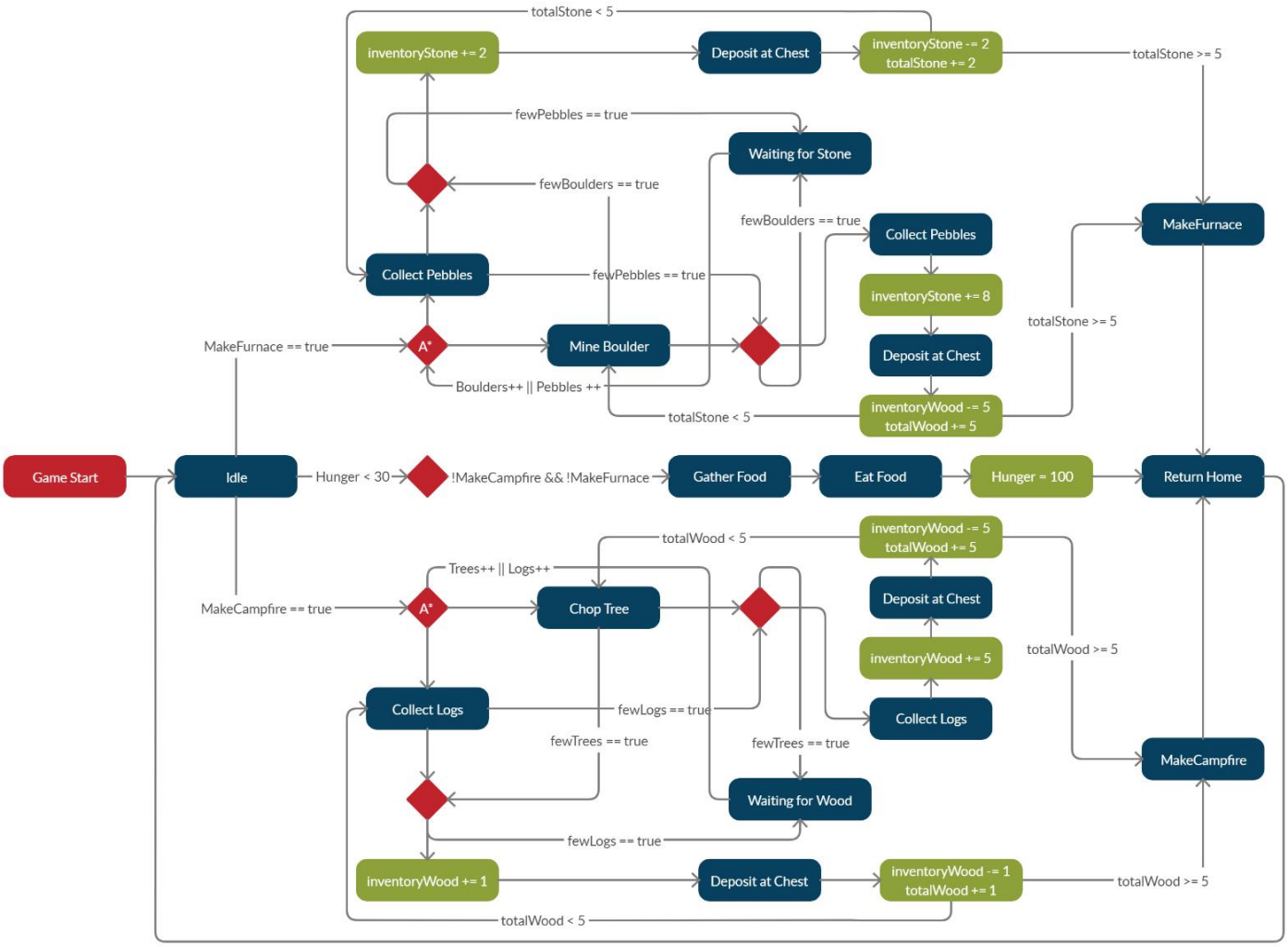


Figure 32: FSM of AI

Figure 32 is the model representing my AI:

- Red rectangles represent the starting state
- Blue rectangles represent other possible states
- Red diamonds represent a decision
- Green rectangles indicate field changes within the program

My agent could be in a total of eighteen states (including twelve unique states). The six duplicated states represent the same action; however, they are located on different paths, and future destinations will vary. The user triggers the transitions MakeFurnace() and MakeCampfire(), which then call the A* algorithm.

This diagram allowed me to locate any areas that may have contained a deadlock; that is any state that contain no exit arcs. This included the 'Waiting for Wood' and 'Waiting for Stone' states, which would occur if we ran out of resources. To alleviate this problem, I implemented some UI features to escape such a scenario, as discussed in Section 4.1, but for future improvements I would like to develop a more robust solution, mentioned in Section 6.2.

3.42 Graph Representations

Before considering if I needed to develop a graph representation myself, I researched into existing packages and libraries that met my specifications. One such solution was QuickGraph [42], providing:

'generic directed/undirected graph data structures and algorithms for .NET'

I chose not to use this library in my project. My current adaptation of an existing A* solution already included its own graph constructors, and as the focus was mainly on the program in Unity, I did not feel it necessary for more complex graph generation.

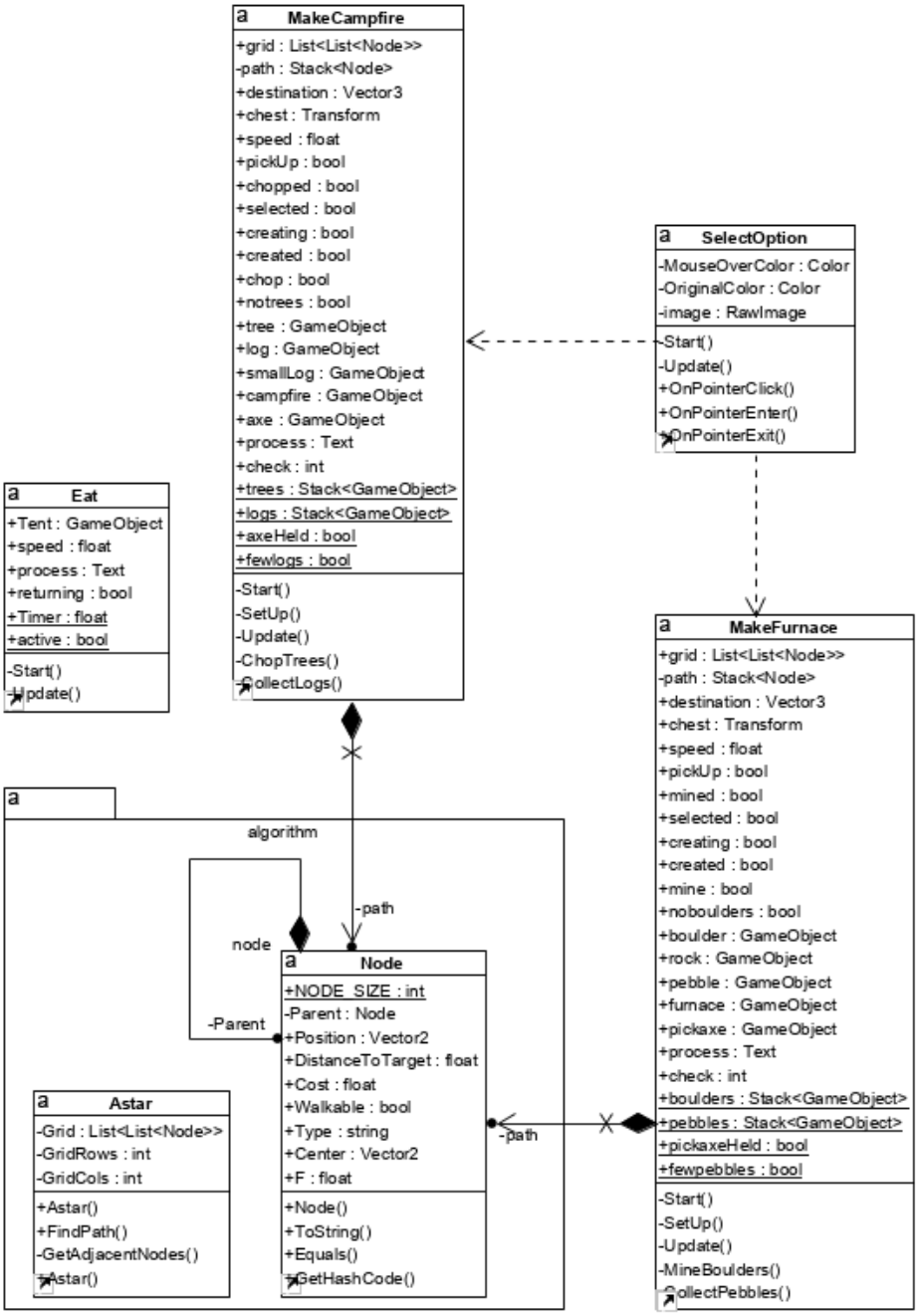


Figure 33: UML Class Diagram

As shown by the dependencies in the Figure 33, the user triggers the events MakeFurnace() and MakeCampfire() through interaction with the UI through the 'Select Option' class. There is one action that is always active, Eat(). Because there is only one pathway to complete this action, a graph is not required, and so Eat() in unconnected from user interaction and the A* algorithm.

Having designed the state machine, I could then create the graphs that would represent the states and possible pathways. After this, I could apply A* to choose the path with the least cost. The algorithm is interconnected with the automaton, providing a simulation of GOAP. Here is an example of the graph for MakeCampfire, when SceneResources = 0:

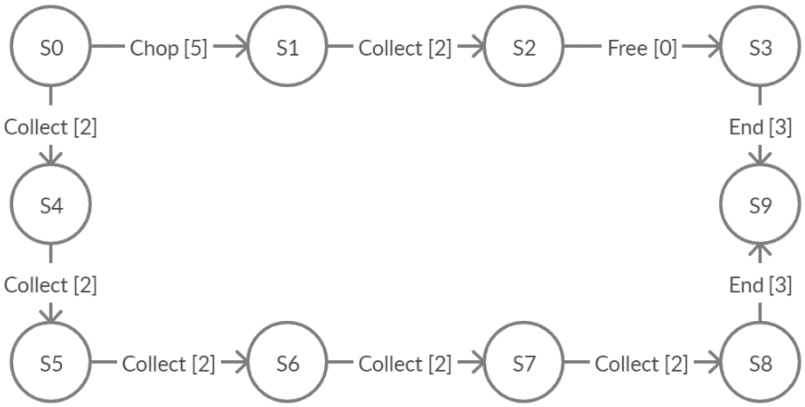


Figure 34: Graph, Wood = 0

The numbers in brackets for each edge represent the cost of performing an action and S represents the states. The cost of END is always the same, no matter the path taken.

Here, our AI can either: chop down a tree (depositing five wood), collect the resources and return them to the chest; or it can search the world for five smaller logs (worth one wood each) collect each and return them to the chest.

Examples of how this grid is generated can be found in the MakeCampfire() and MakeFurnace() classes of my GitHub repository: https://github.com/SamuelAppleby/GoalOrientedActionPlanning.

For each different resource amount, a different grid would be made, here is the grid generated when we have 3 wood in our SceneResources:

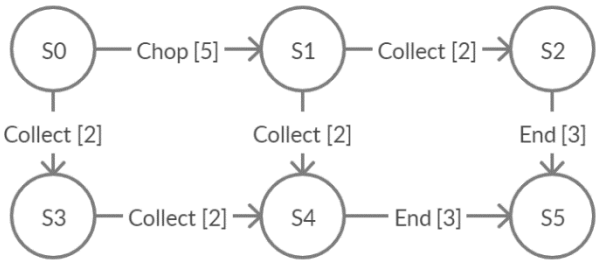


Figure 35: Graph, Wood = 3

In this case, we could still perform the Chop() action and collect the wood, but we only need to collect two small logs otherwise. This is where I then apply the A* algorithm to determine which path should be sought.

The idea is that when we need to collect large amounts of wood, it is more beneficial for us to chop down a tree, generating all the resources needed, instead of searching the world for smaller logs. Figure 35 is a smaller graph because fewer actions are required to gather the required resources and allows A* to traverse the graph more efficiently.

We can see from the graph in Figure 34 the pathway taken to cut down a tree is (almost, due to heuristic costs) the same as Figure 35. However, the cost of collecting logs in the first graph is much greater than in the second. The details of testing this can be found in Section 4.2 of this document.

3.44 Pathway Validity

Our AI could choose a path it believed was the best but there may not be enough resources in the world.

For example, the A* algorithm could determine that the agent should cut down a tree, however we might not have any trees left. In this scenario, the AI would permanently be blocked, until a tree was added to the stack.

Instead of manipulating the graphs to solve this, I let the algorithm run as normal, then only after we receive the path do, I check if it valid or not:

```
if (chop)
{
    if (notrees)
    {
        CollectLogs();      // If the best path has told us to chop trees but there aren't trees available, collect logs instead
    }
    else
    {
        ChopTrees();        // Otherwise do the requried action
    }
}
else
{
    if (fewlogs)
    {
        ChopTrees();        // If the best path has told us to collect logs but there aren't enough available, chop trees instead
    }
    else
    {
        CollectLogs();      // Otherwise do the requried action
    }
}
```

Figure 36: Resource Checking

The *notrees* and *fewlogs* booleans would be set every time ChopTrees() and CollectLogs() were finished:

```
if (homeDirection.magnitude < 10)
{
    check = 0;
    GameObject.Find("MakeCampfireText").GetComponent<Text>().color = Color.black;
    GameObject.Find("MakeFurnaceText").GetComponent<Text>().color = Color.black;
    GameObject.Find("Player").GetComponent<MakeCampfire>().enabled = false;
    if (trees.Count == 0)
    {
        notrees = true;     // After making the campfire, if there are now no tree, default to collecting logs instead
    }
```

Figure 37: Setting Resource Availability

Whenever our algorithm determined the best path involved chopping trees or collecting logs, we check if the resources are available. If not, then we would perform the other action. The agent would *always* try to reach its goal, even if the resource required by the A* path is unavailable. Our AI makes efficient use of resources it can and limits the times a deadlock will be reached.

There are situations where neither action is possible (where there are no trees and too few logs), and so making the campfire was impossible. In this case we reach the 'Waiting for Wood' state, where the AI remains idle until resources are available. To solve this, I included a UI resource controller where we can increment and decrement the resources at run time to escape any deadlock scenarios.

In addition to MakeCampfire(), I also included a MakeFurnace() action. Here, the costs of performing the possible pathways are different, and testing of this can be found in Chapter 4.

3.45 Unity Game Engine

3.451 Why Unity?

The focus of my project is developing AI in established game worlds. Therefore, I did not want to pick engines that gave the user very limited control over the code, such as Construct 2, or others that were effective but contained limited assets, such as CryEngine. Unity was an ideal solution due to its ability to create 3D environments with ease (with a wide range of assets available), while still including the ability to apply scripts, making it very flexible.

3.452 Implementation

To both test my scripts in controlled environments, and meet the criteria my third project requirement, I made three unique worlds: Desert Island, Forest, and Arctic. I was able to modify the resources available, and test how the AI would be able to adapt to these distinct environments:

Figure 38: Game World Selection

The Forest worlds contained more logs that in the Arctic world, while the Arctic world contained more pebbles. This would allow me to see how the AI would behave to varying resources.

Desert Island:

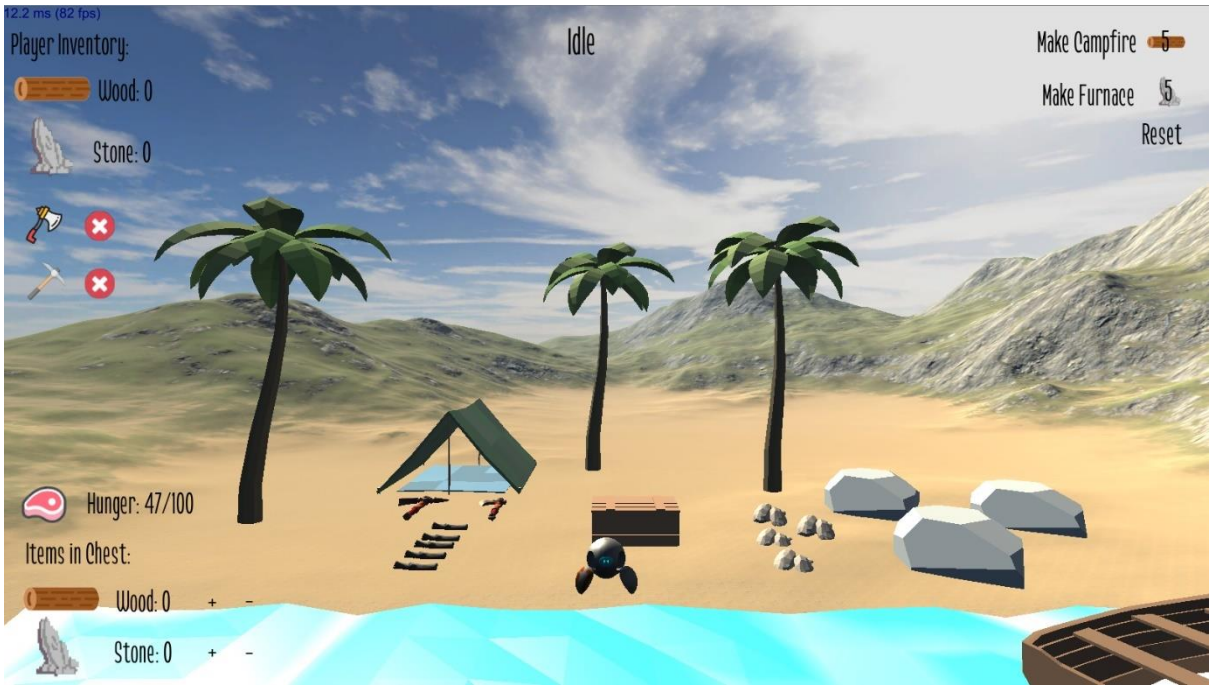


Figure 39: Desert Island

Forest:

Figure 40: Forest

Arctic:

Figure 41: Arctic

To mine rocks for stone, the agent required a pickaxe, and to chop trees, an axe. These conditions were further checked within the code, here is the check *axeHeld* for the ChopTree() action:

```
if (!axeHeld)       // Can only chop trees if we have the axe
{
    destination = axe.transform.position;
    selected = true;
}
if (axeHeld)
{
    if (trees.Count != 0)
    {
        tree = trees.Pop();     // Pop the top tree from the stack, this is the tree we mine
        destination = tree.transform.position;
        selected = true;
    }
}
```

Figure 42: Tool Checking

Once the agent was in acquisition of either of these tools, it could then chop a tree or mine a boulder. No tools are required for gathering pebbles or logs. In addition to these two actions, I implemented another that had a greater priority, Eat().

This required a hunger meter, decreasing with time would be performed whenever hunger dropped below 30%. The AI would return to the tent where it would gather and eat food, setting it back to 100%. When it returned to the Idle position it could then carry out its other actions.

If hunger dropped below 30% during another action, the agent would finish the current process and then immediately perform Eat:

```
if (Timer < 7.5f && GameObject.Find("Player").GetComponent<MakeCampfire>().enabled == false && GameObject.Find("Player").GetComponent<MakeFurnace>().enabled == false)
{
    active = true;
    process.text = "Gathering Food";
    UnityEngine.Vector3 tentDirection = Tent.transform.position - transform.position;
    transform.Translate(speed * tentDirection.x * Time.deltaTime, 0f, speed * tentDirection.z * Time.deltaTime);
    if (tentDirection.magnitude < 10)
    {
        process.text = "Eating Food";       // Stay in the tent to decrease hunger
        if (tentDirection.magnitude < 5)
        {
            returning = true;
            Timer = 25.0f;      // Reset our hunger metert
        }
    }
}
```

Figure 43: Eat Action Condition

The current process of the AI is displayed above the screen, allowing me to identify the current state that the agent is in:

Figure 44: Process Indicator

If my agent changed state, this identifier would be updated. This is useful not only for visual feedback to the user but also for debugging purposes.

3.5 Summary

Having designed my state machine and UML class diagram, programmed the graphs in C# and created the game worlds in Unity, I was then ready to test the program. I ran through all the possible states, all transitions, and, for each call to the algorithm, I printed the algorithms' decision making. These details are discussed in Chapter 4.

Chapter 4. Testing

This chapter details the approach I took when testing my AI. It is here that I try to achieve my objectives set in Chapter 1, in hopes to accomplish my overall project aim.

4.1 Method

4.11 Correctness

I wanted to ensure my AI was following the desired behaviour at every given state. This was especially important for duplicate states, such as 'Deposit At Chest', where checking that the agent was on the correct path was important.

Other considerations included the objects contained in each of the game worlds. As each scenario contained varying amounts of resources, I needed to ensure that the AI was aware of those in its current world only, and that the removal of objects was bound to that world's state.

4.12 Deadlock States

To ensure that my AI was never in an inescapable state, I analysed my state diagram:

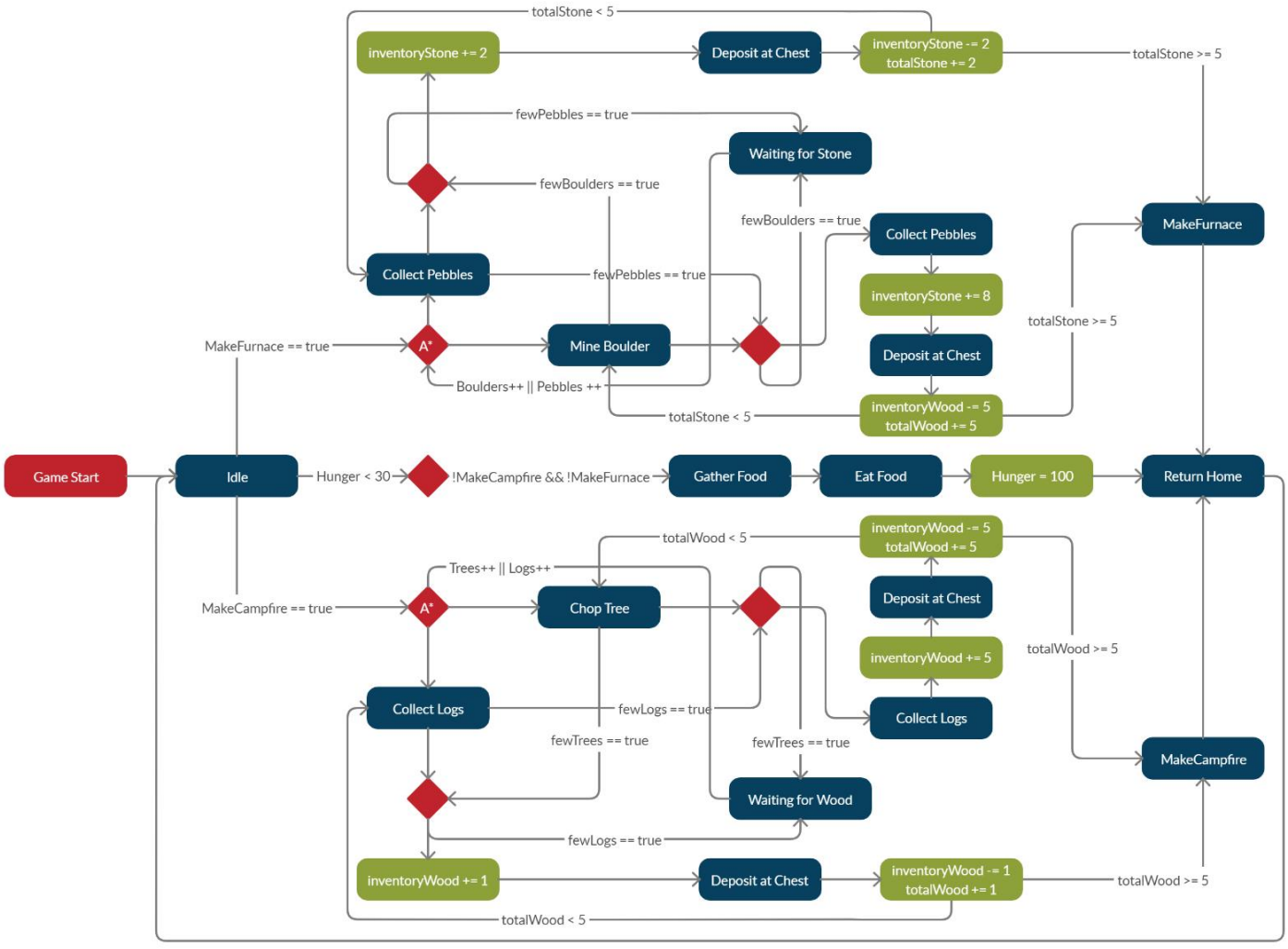


Figure 45: State Diagram

I could use this to deduce that running out of resources may cause our AI to fall into a deadlock state: either ‘Waiting for Wood’ or ‘Waiting for Stone’. After the actions CollectLogs(), ChopTree(), MineBoulder() or CollectPebbles(), there is a check to see if such resources exist; if not then we enter either of these states and the agent would wait forever. I had to change how my program functioned.

Therefore, I implemented a feature where resources could be manipulated at run time: a UI element that allowed for wood and stone to be added and removed:

Figure 46: UI Resource Feature

If were in the state ‘Waiting for Wood’, then this field could be incremented and allow the AI to complete its goal. This was also useful for debugging, as it allowed me to perform multiple tests for a given resource amount during the running of the program.

Another feature included was a ‘Current Process’ UI element. When another state was reached, or a transition was taken, this field was updated.

Figure 47: UI Process Feedback

This was especially practical when multiple actions would be attempted at once. The feedback made me aware of the current position in the program and allowed me to identify where certain conditions were not meeting expectations.

4.2 Strategy

4.21 Make Campfire

GOAP should ensure the AI will take make decisions which it believes are the least costly. I had to allocate costs to my actions in such a way that they would become promising or unfavourable depending on the current state of the world.

In my simulation, chopping down a tree should be the more favourable option if four or more wood was required. Let us apply the algorithm operating on the grid when we have zero wood in our resources:

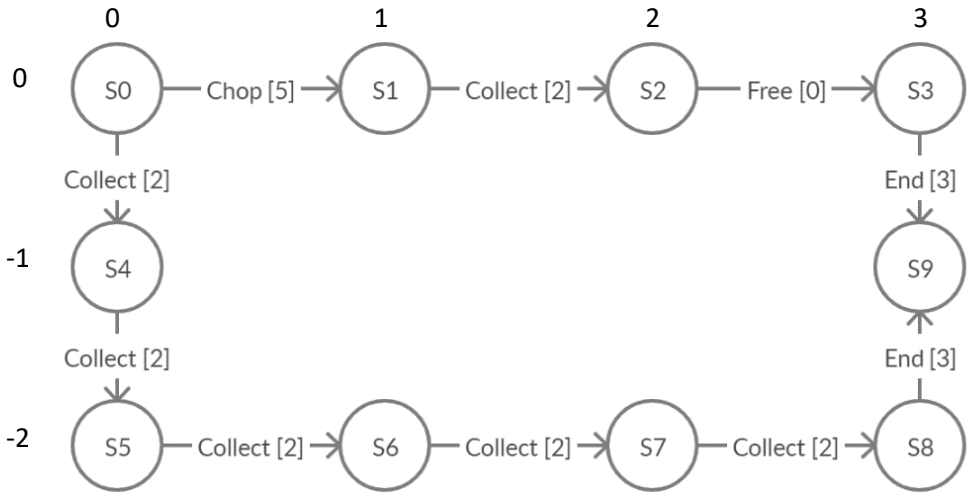


Figure 48: Graph, 0 Wood

Heuristic: Manhattan

Final Table:

(0, 0)		(1, 0)		(2, 0)		(3, 0)		(0, -1)		(3, -1)		(0, -2)		(1, -2)		(2, -2)		(3, -2)	
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
0	-	8	(0,0)	-	-	-	-	5	(0,0)	-	-	-	-	-	-	-	-	-	-
0	-	8	(0,0)	-	-	-	-	5	(0,0)	-	-	8	(0,-1)	-	-	-	-	-	-
0	-	8	(0,0)	9	(1,0)	-	-	5	(0,0)	-	-	8	(0,-1)	-	-	-	-	-	-
0	-	8	(0,0)	9	(1,0)	-	-	5	(0,0)	-	-	8	(0,-1)	9	(0,-2)	-	-	-	-
0	-	8	(0,0)	9	(1,0)	8	(2, 0)	5	(0,0)	-	-	8	(0,-1)	9	(0,-2)	-	-	-	-
0	-	8	(0,0)	9	(1,0)	8	(2, 0)	5	(0,0)	10	(3,0)	8	(0,-1)	9	(0,-2)	-	-	-	-
0	-	8	(0,0)	9	(1,0)	8	(2, 0)	5	(0,0)	10	(3,0)	8	(0,-1)	9	(0,-2)	10	(1,-2)	-	-

1) Expand and pop (0, 0). Nodes (1, 0) and (0, -1) have newly acquired f values:

   (0, -1): (2 + 3 = 5) via (0, 0)
   (1, 0): (5 + 3 = 8) via (0, 0)

2) Lowest f value is (0, -1), this becomes our current node

3) Expand and pop (0, -1). Node (0, -2) has newly acquired f values:

   (1, 0): 8 via (0, 0)
   (0, -2): (4 + 4 = 8) via (0, -1)

4) Joint lowest f value, take that from the top of the priority queue, (1, 0), this becomes our current node.

5) Expand and pop (1, 0). Node (2, 0) has newly acquired f values:

   (0, -2): 8 via (0, -1)
   (2, 0): (7 + 2 = 9) via (1, 0)

6) Lowest f value is (0, -2), this becomes our current node

7) Expand and pop (0, -2). Node (1, -2) has a newly acquired f value:

   (2, 0): 9 via (1, 0)
   (1, -2): (6 + 3 = 9) via (0, -2)

8) Join lowest f value, take that from the top of the priority queue, (2, 0), this becomes our current node.

9) Expand and pop (2, 0). Node (3, 0) has a newly acquired f value:

   (3, 0): (7 + 1 = 8) via (2, 0)
   (1, -2): 9 via (0, -2)

10) Lowest f value is (3, 0), this becomes our current node.

11) Expand and pop (3, 0). Node (3, -1) has newly acquired f value:

   (1, -2): 9 via (0, -2)
   (3, -1): (10 + 0) via (3, 0)

12) We have reached our destination node, though there may still be a faster way as it has not reached the top of our priority queue. Lowest f value is (1, -2), this becomes our current node.

13) Expand and pop (1, -2). Node (2, -2) has newly acquired f value:

   (3, -1): 10 via (3, 0)
   (2, -2): (8 + 2 = 10) via (1, -2)

14) Lowest f value is (3, -1), this becomes our current node.

15) Expand and pop (3, -1). This is our destination, terminate the algorithm.

So, to determine the shortest path we backtrack through the table, resulting in:

(3, -1) -> (3, 0) -> (2, 0) -> (1, 0) -> (0, 0), Length = 10

We have found our path manually and know what nodes should be on the open and closed lists.

Upon every call to the algorithm, I write the closed and open lists to a csv file, and the path taken. This is done using the FileWriter class. Here is the generation after we run the algorithm:

OpenList	
Element	Position
0	(2, -2)
1	(3, -2)
ClosedList	
Element	Position
0	(0, 0)
1	(0, -1)
2	(1, 0)
3	(0, -2)
4	(2, 0)
5	(3, 0)
6	(1, -2)
7	(3, -1)
Path	
Element	Position
0	(0, 0)
1	(1, 0)
2	(2, 0)
3	(3, 0)
4	(3, -1)

Figure 49: CSV, Wood = 0

First, we look at the path to see if it what we expected. From our manual calculation, we determined that the optimum path was (0, 0) -> (1, 0) -> (2, 0) -> (3, 0) -> (3, -1). This matches the path generated by A* in Figure 49.

We next view the closed list to see what elements were considered along the way. This should include all the positions contained in the path, in addition to the other attempted routes. From the manual example, we can see that along the bottom path we popped (0, -1), (0, -2) and (1, -2) at steps 3), 6) and 13) respectively. Again, this corresponds with our table; the closed list contains what we expected.

Finally, we view the open list, this should contain only the elements still to be considered. After step 13) (2, -2) was still contained in our open list, and (3, -2) had been added due to being the (2, -2) being its child. This is exactly what we see in the table.

In total the algorithm has included all ten positions on the grid and correctly found the shortest path in the order that we calculated it. This proved the algorithm works for this resource level and I could begin to test other scenarios.

As previously mentioned, the AI should choose collecting logs only when its current wood count was two or more. Here is the graph representation:

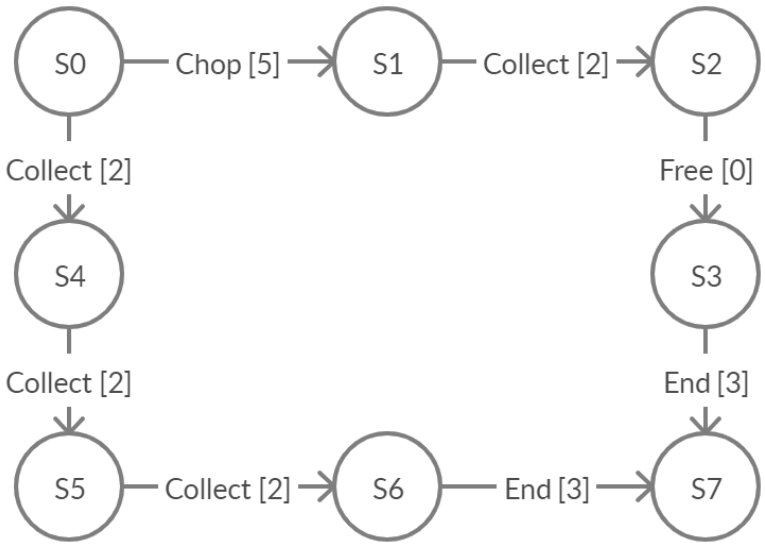


Figure 50: Graph, Wood = 2

Here is the table generated when after running with two wood:

OpenList	
Element	Position
0	(2, 0)
1	(2, -1)
ClosedList	
Element	Position
0	(0, 0)
1	(0, -1)
2	(0, -2)
3	(1, -2)
4	(1, 0)
5	(2, -2)
Path	
Element	Position
0	(0, 0)
1	(0, -1)
2	(0, -2)
3	(1, -2)
4	(2, -2)

Figure 51: CSV, Wood = 2

The algorithm calculated the path with the least cost was to collect logs. Amounts of wood higher than this will cause the route to be less and less costly, so the AI will always choose this path.

4.22 Make Furnace

I wanted to see how varying the costs of certain actions would affect the behaviour of the agent, so I made another action, MakeFurnace(), requiring either the mining of boulders or collecting of pebbles.

I wanted to vary costs such that the more favourable option was to mine boulders. This is because mining a boulder yields eight stone, compared to chopping a tree which yields five wood, but both their actions cost the same, five.

I manipulated the costs such that if the current amount of stone held is less than three, mining a boulder should be favourable. Therefore, the AI should be more inclined to do the more costly action with MakeFurnace() compared to MakeCampfire().

This is the graph created when current stone = 2:

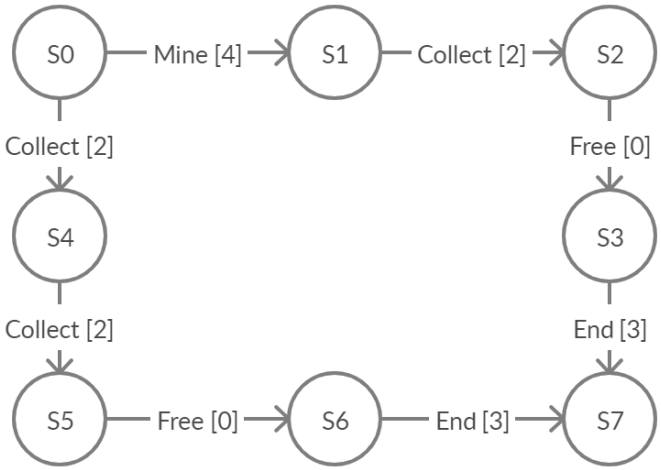


Figure 52: Graph, Stone = 2

And the CSV file generated:

OpenList	
Element	Position
0	(1, -2)
1	(0, -2)
ClosedList	
Element	Position
0	(0, 0)
1	(0, -1)
2	(1, 0)
3	(2, 0)
4	(2, -1)
5	(2, -2)
Path	
Element	Position
0	(0, 0)
1	(1, 0)
2	(2, 0)
3	(2, -1)
4	(2, -2)

Figure 53: CSV, Stone = 2

The cost of mining a boulder is four, in comparison to cutting a tree which is five. This is the only difference, as far as the algorithm knows, when we compare Figure 50 to Figure 52. However, it is enough to cause this path to be chosen instead, and the AI performs the MineBoulder() action.

We can test that CollectPebbles() will be called when the amount of stone held is greater than two. Here is the grid generated when we have three stone:

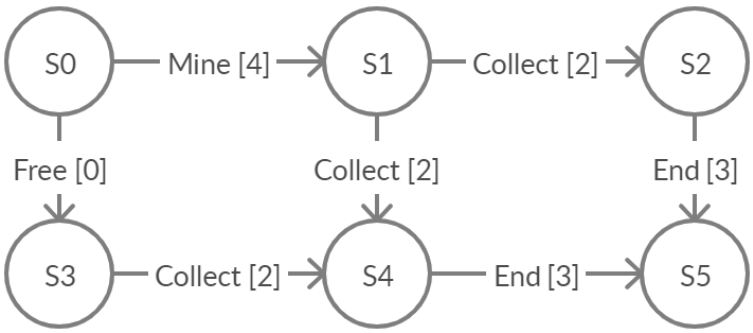


Figure 54: Grid, Stone = 3

Because the action CollectPebbles() gives us two stone per call, we only need to perform it once. This means that this grid is identical for current stone of four also. The CSV file becomes:

OpenList	
Element	Position
0	(2, 0)
ClosedList	
Element	Position
0	(0, 0)
1	(0, -1)
2	(1, -1)
3	(1, 0)
4	(2, -1)
Path	
Element	Position
0	(0, 0)
1	(0, -1)
2	(1, -1)
3	(2, -1)

Figure 55: CSV, Stone = 3

Here we can see that collecting pebbles was the least costly action. Any amount of stone higher than or equal to three will continue to choose this path as the graphs will be identical.

For all twelve combinations of resources (Stone = 0 -> Stone = 5, Wood = 0 -> Wood = 5), I generated all the path produced, which can be found in the csv folder here:
https://github.com/SamuelAppleby/GoalOrientedActionPlanning

Chapter 5: Results and Evaluation

This chapter contains the results from testing my agent. I shall review the areas where the AI changed decisions, discuss why, and refer to data and graphs of my findings.

5.1 Computational Complexities

5.11 A*

The complexity of A* depends on the heuristic used, more complex methods will result in a worse order, but in general we can always achieve better than Dijkstra's algorithm (O(ElogV)). We can only use A* if we have knowledge about our graph, otherwise we must use an uninformed algorithm like Dijkstra's.

5.12 Dijkstra

The complexity of Dijkstra's algorithm is $O(V^2)$. This can be optimised with a min-priority queue, and the complexity drops down to O(ElogV), where E is the number of edges and V is the number of vertices [43].However, when dealing with negative weights, Dijkstra's algorithm breaks down, and we must use a different algorithm, the Bellman-Ford algorithm.

5.13 Bellman-Ford

The time complexity of Bellman-Ford is O(EV) [44], where E the number of edges and V is the number of vertices. This is worse than optimised Dijkstra's, with order O(ElogV).

5.14 Summary

When deciding on the graph search algorithm to use for implementing into a GOAP action plan, it much depends on the structure and content of the graph. Instead of implementing and testing all the different algorithms independently for each graph scenario, I was able to study their complexities which would give me an insight of their efficiency, and make my decision based on these finding. Let us compare the orders (worst case) for each of the three algorithms:

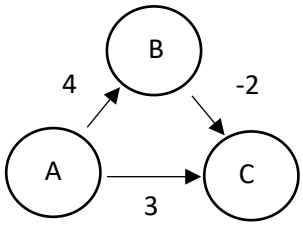
A* -> Dependant on heuristic, almost always faster than O(ElogV)

Dijkstra (optimised) -> O(ElogV)

Bellman-Ford -> O(EV)

When comparing the complexity of A* to Dijkstra, we must consider the heuristic. Technically, Dijkstra is A* but where all heuristic costs are 0. If we set all h(v) in A* to 0, the algorithms would behave identically. However, by using the knowledge of the graph and future costs, the heuristic can help us achieve both a more optimum answer, and in less time.

Dijkstra therefore seems to be the best to use if we require an uninformed algorithm. However, negative weights will cause Dijkstra to break. For example, given the following graph:



With our source node as A and destination node as C, Dijkstra will pick the edge AC first, the goal is reached, and we never consider B. The cost of this is 3 but the optimum is AB + AC (4 + -2 = 2), which would have been discovered with Bellman-Ford.

However, this solution harms the complexity, making Bellman-Ford perform worse than optimised Dijkstra.

Overall:

Informed? -> A*
Not informed and no negative weights? -> Dijkstra
Not informed with negative weights? -> Bellman-Ford

A* is complete, performs well and will always find a route from any node to the destination. In games, we almost always can estimate the distance to the goal node, and the algorithm also can deal with negative weights.

In addition to this, with an appropriate heuristic A* can perform better than any other graph searching algorithm. For these reasons, A* is the most popular algorithm to use for GOAP. Therefore, this is the algorithm I chose to implement in my solution.

5.2 Correctness and Performance

The two major characteristics of an algorithm's effectiveness is correctness and performance. From Chapter 4, we can see that the algorithm gives us the results we expect for every combination of outcome. Now it is time to test the speed at which the algorithm runs.

From Chapter 4, we can see how the size of the graphs vary, and so I wanted to analyse the correlation between running time and graph size.

One every call to MakePath(), I record the time taken to complete the main loop and include them to the csv file containing the path. The results below come from the MakeCampfire() method.

The algorithm will be running five times for each graph. This will allow me to compute an average time for every scenario and increase the precision of the results. I now perform the tests on the 'Forest' world; a comparison of performance in the other environments can be found in Section 5.3.

Extra time is taken for the first instantiation of the call, and we retrieve much higher values for the run time. These results are not an accurate representation of the speed of the run, and therefore, I will exclude the results from the first run of the method.

The time values represented by the data is the exact time taken for the A* to run, there is no consideration of the time taken for the agent to successfully compete the task.

I chose to represent it this way because I wanted an exact value of the efficiency of the algorithm, not the time taken for the entire GOAP to complete.

The largest grid size occurs where wood = 0, containing ten possible states. The graph is represented as such:

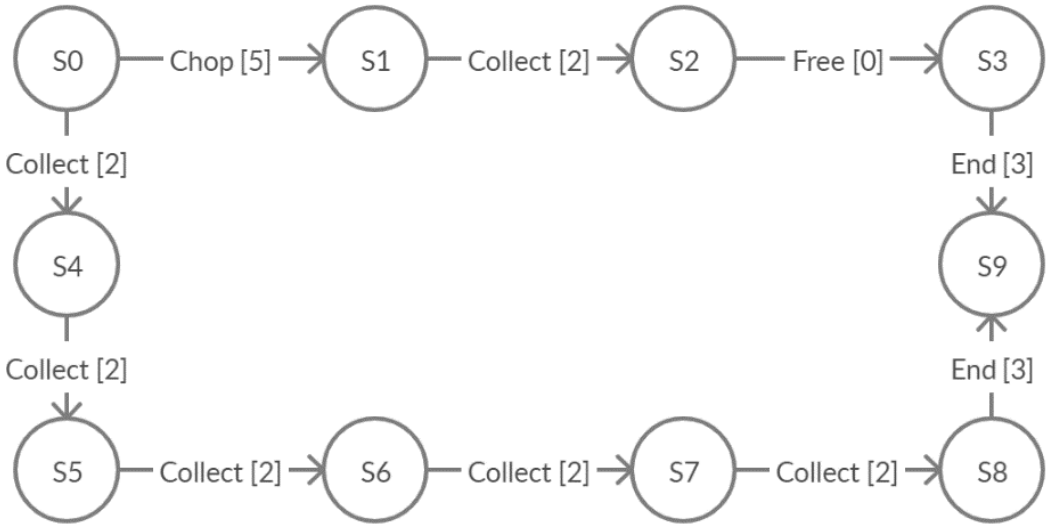


Figure 56: Graph, Wood = 0

Average time = 0.03706ms

The smallest grid size occurs where Wood = 5, containing only 2 states. Here, the only action needed is to create the campfire (with a cost of 3), there is only one path that the algorithm can take:

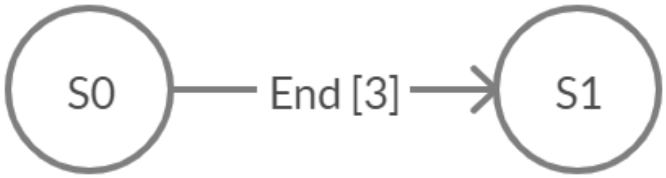


Figure 57: Graph, Wood = 5

Average time = 0.01468ms

We can see how the cost to search a larger graph is much more costly, it takes ~2.5 * longer in the worst case. To check this, I will test the middle ground.

A medium size graph occurs where wood = 2, graph represented as:

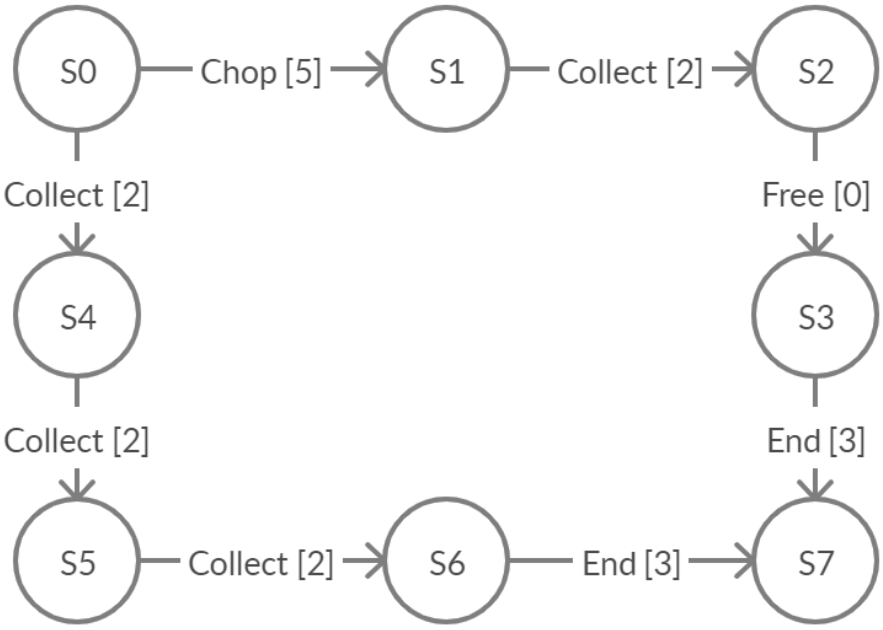


Figure 58: Graph, Wood = 2

Average time = 0.02986ms

This average lies between Wood = 5 and Wood = 0.

I will now combine all possibilities into a single graph:

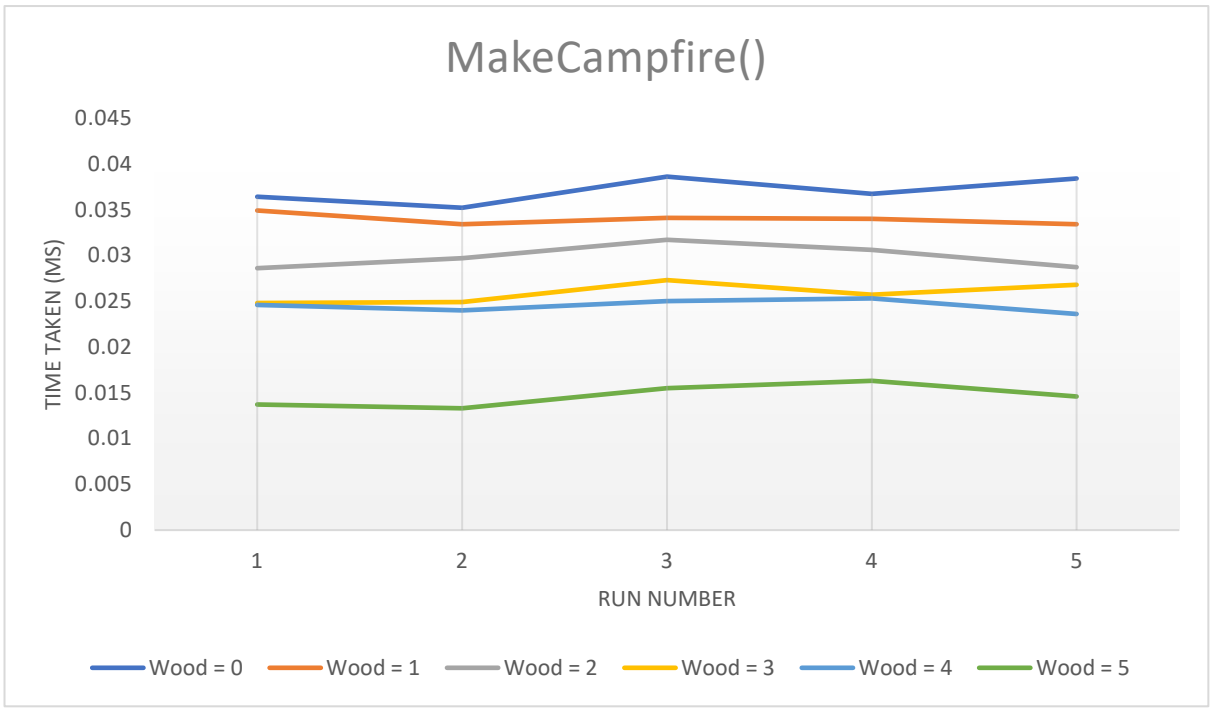


Figure 59: Time Taken for all Resources

And if we compare the averages:

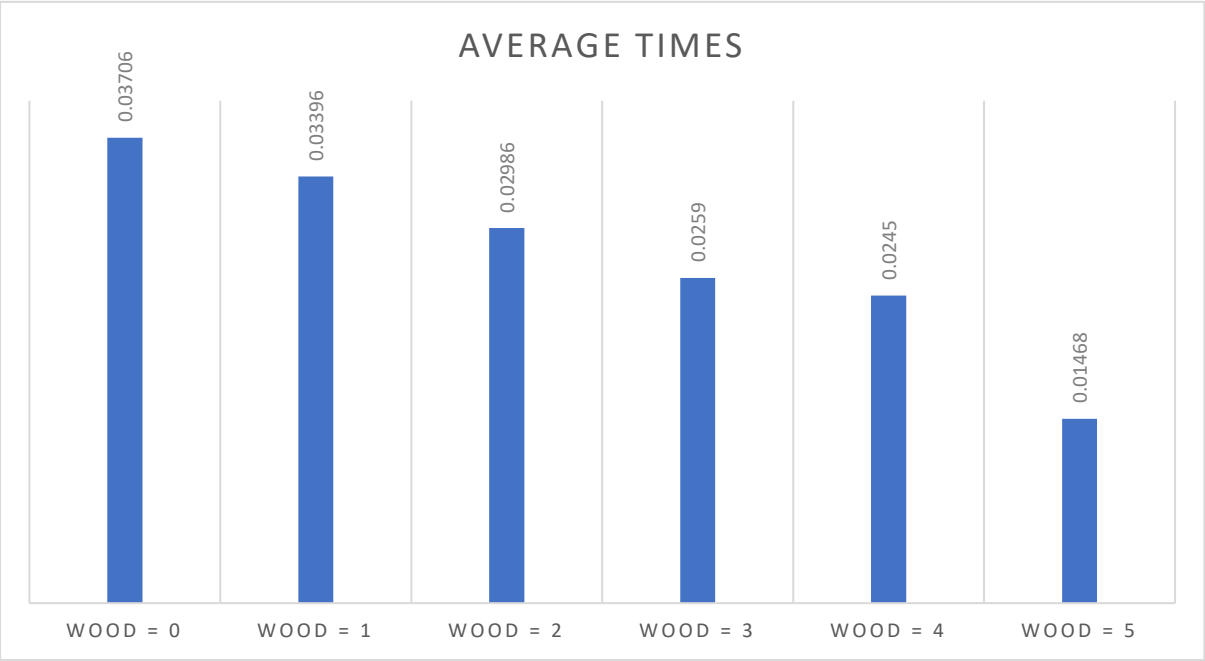


Figure 60: Average Run Times

From Figure 59 and Figure 60 we can see there is a strong linear correlation between the size of the graph, and time required to traverse it. This gives us the benchmark for our algorithm. In the worst case, the average time is 0.03706ms.

However, because my results show a direct correlation between graph size and run time, I can conclude that this algorithm is working as expected. I can hypothesise that more complex graphs with a greater number of paths would increase the run time.

5.3 Comparison Between Environments

The previous tests were all run on the 'Forest World'. As the running time is calculated based purely on the A* process time , it means that running these same tests in the other worlds should yield very similar results. I will run MakeCampfire() with wood = 3 across the three environments and test these predictions.

Graph Generated:

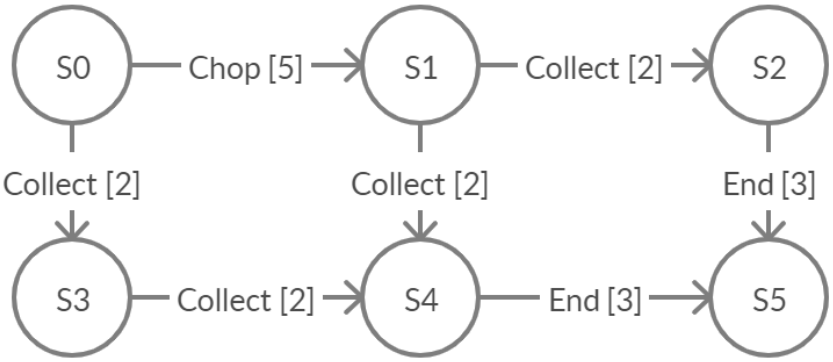


Figure 61: Graph, Wood = 3

Desert Island:

Average time = 0.02628ms

Forest:

Average time = 0.02618ms

Arctic:

Average time = 0.02552ms

Combined Graphs:

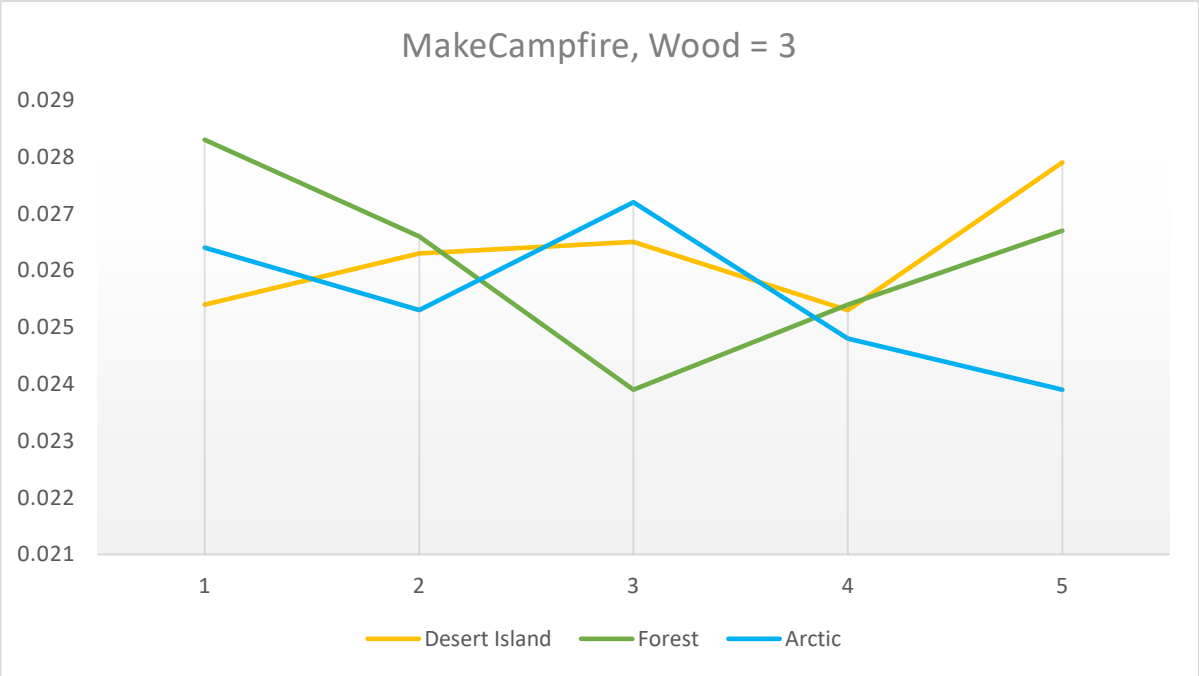


Figure 62: Average Run Time across Worlds

Averages:

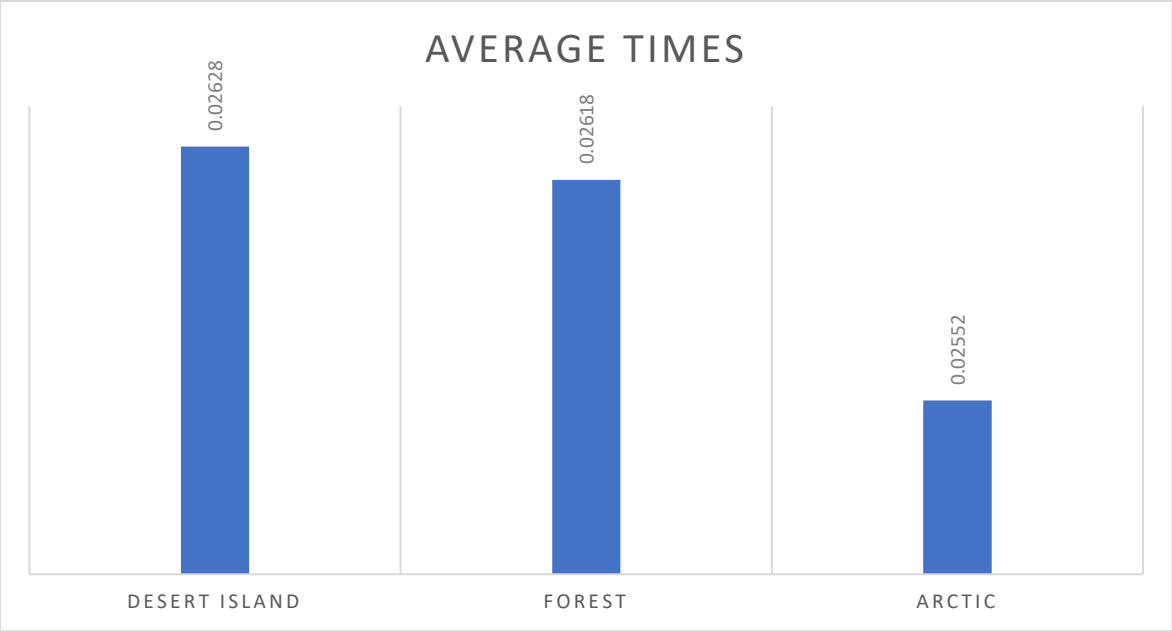


Figure 63: Average Run Times for Different Worlds

As we can see from the graphs, notably Figure 62, there seems to be no correlation for the time taken for the algorithm to operate in different environments. Even the difference seen above in Figure 63 shows Desert Island takes only 3% more time to run than Arctic. This value is low enough to conclude that further testing would result in this difference being further reduced.

5.4 Framerate

My project solution contains a ShowFPS() class that displays the current framerate. I modified it so that it will show the average framerate over a given time. This would allow me to compare the time required for graphical processing against running time of the logic (including the algorithmic computation).

First, I will obtain the average framerate for the 'Idle' state in each environment, having run for thirty seconds:

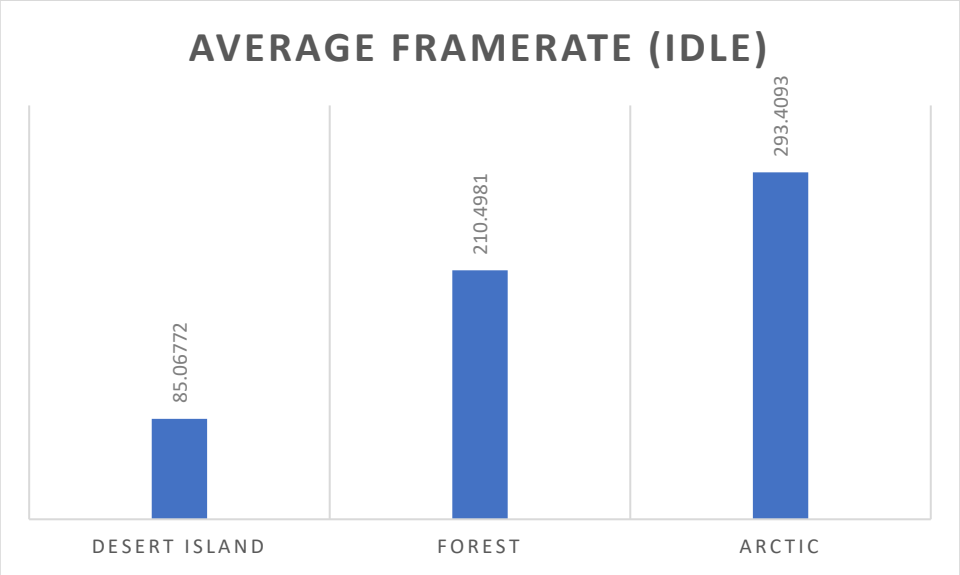


Figure 64: Average Framerate (Idle)

I will now obtain the average framerates for the duration of the MakeCampfire() action only:

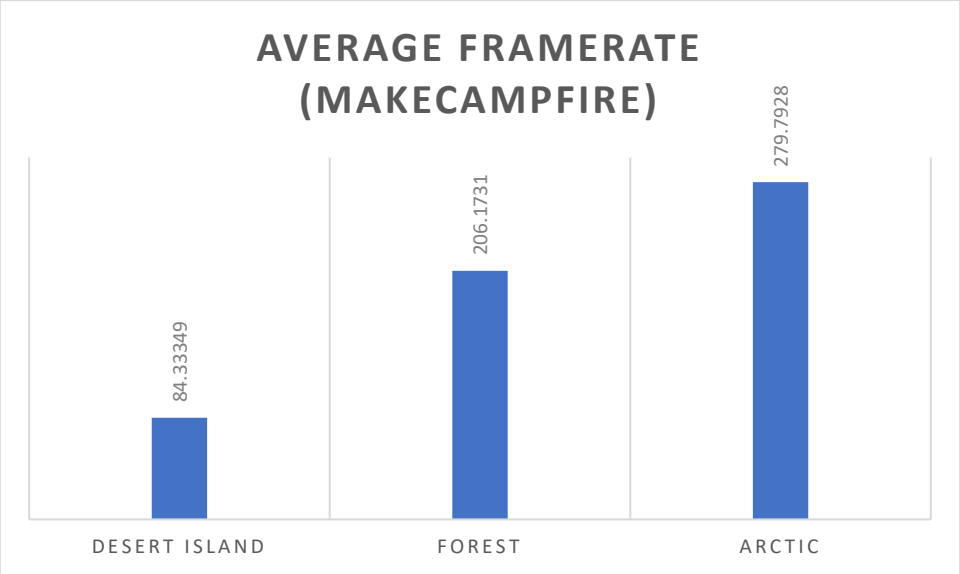


Figure 65: Average Framerate (MakeCampfire)

And we can compare the difference between these states:

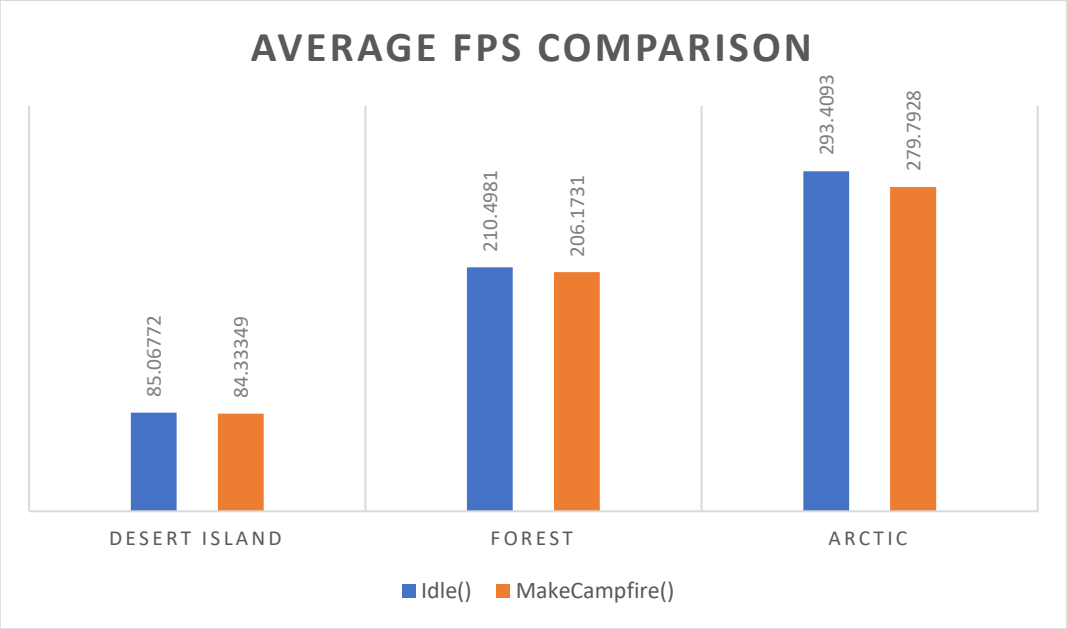


Figure 66: Average FPS Comparison

From Figure 66, we can see how to fps varies between the worlds. This is due to the graphical demands of each scenario. The Desert Island world includes multiple animated objects (water forming the sea), whereas the others contain only stationary objects. This causes the large difference in framerate, with the averages of Desert Island and Forest being 85fps and 210fps, respectively. The Arctic world is the most barren, containing the fewest game objects of them all, and is why it has the highest score of 293fps.

When we introduce more logic that must be processed, in the form of any of the actions: MakeCampfire(), MakeFurnace() or Eat(), the framerate is affected. Figure 66 shows the impact of the MakeCampfire() action on each environment. In all cases the framerate dropped, with the largest decrease from 293fps to 279fps in the Arctic world.

However, it is clear from these graphs that computational time required for graphical rendering dominates over logic. This is evident from the difference between Desert Island and Arctic in Figure 65, against the drops in framerates in Figure 66.

I can conclude that graphical rendering has the largest influence on framerate from my own implementation only. I cannot say that these finding are certain for all programs, as other solutions with different logic may see different outcomes. For example, even the largest graph in my project contains only twelve nodes, and so the loop that traverses this terminates relatively fast than it would to other graphs containing hundreds of nodes. For other programs, the decision may take much longer to process, and we could see logic have a much greater impact on fps.

5.4 Summary

From the testing and results conducted in Chapter 4, in addition to the time values calculated for each scenario, all results are as expected. This includes:

1) Performing A* on larger graphs results in slower computation time
2) Performing A* on the same graph across varying worlds has no effect on computation time
3) Graphical rendering will have a greater impact on framerate than logic processing

The outcomes gained from Chapter 4 and Chapter 5 proves that my agent is robust; it performs as predicted given the current resources and multiple testing of the same scenario yields the same results.

It was difficult to test my solution against other GOAP applications. This is because most cases are implementation dependant , and so the results gained from another example would be difficult to make a valid comparison with my own.

However, while I was not able to test against other solutions; my AI was able to successfully change its behaviour in different worlds. This confirms that my agent is adaptable; it will respond to differing environments and make the most efficient decision based on the A* algorithm.

Chapter 6: Conclusion

This chapter concludes my findings. I will discuss the extent to which I completed my objectives, how they helped fulfil my aim and what I would do to take this project further.

6.1 Satisfaction of Aims and Objectives

6.11 Satisfaction of Objectives

1) *Identify and evaluate three different uses of GOAP currently being used in the games industry.*

In Chapter 2 I researched the origins behind GOAP and its lead designer, Jeff Orkin. By reading his papers I was able to determine the benefits of using such a strategy when designing AI. In Section 2.21 I was able to find several games that have implemented this technique, including Fallout 3 and Just Cause 2.

In addition to the games involving GOAP, papers such as this [45] identify where the strengths of the architecture lie. As such I have been able to identify at least 3 uses of GOAP in industry.

2) *Determine the fields of GOAP that have not been thoroughly assessed.*

While there were many online materials that assessed and analysed GOAP, one field that I found unexplored was the idea for AI to be able to display the same actions, but across different world environments.

For example, the development of an AI that could demonstrate effective behaviour across F.E.A.R. *and* another similar game. Therefore, I chose to create three distinct worlds into my simulation, to show how the agent is not dependant on only one implementation.

3) *Plan and create distinct game worlds and scenarios.*

To be able to prove my ability to tackle Aim 2, I needed to construct different environments where the agent could show adaptability. I managed to create three worlds: Desert Island, Forest, and Arctic, all containing varying resources.

In terms of creating different scenarios/events this was limited. I would have like to include features such as sleep cycles or weather events which change what is available to the AI, however, chose to focus on refining the existing solution.

4) *Create at least 2 goals which can be achieved through various pathways.*

My solution contained three goals, MakeCampfire(), MakeFurnace() and Eat(). The first two of these each contained more than one way of being achieved, the efficiency of which path was appropriate was decided by the A* algorithm.

I would have liked to been able to extend Eat(), for example give the AI the ability to hunt or collect food, that must be cooked before eating etc., which would further diversify the abilities of the AI.

5) *Research and implement an effective testing strategy for the AI.*

By creating a FileWriter class, I was able to identify the choice the algorithm made with every decision. In addition to this, I created UI elements which displayed how all resources were allocated in the world, and every choice made would be displayed to the user.

I was able to test every combination of resources and view all the decisions, which made for an effective testing strategy.

6) *Debug and evaluate the agents and their corresponding scripts.*

Through my csv files and debugging tools (such as breakpoints), I was able to navigate to specific areas that were causing any errors in the program. The result is that my program now should contain no major bugs.

Evidence of this is displayed in the graphs and tables of Chapter 4 and Chapter 5, where the behaviour expected of the AI was consistently met.

6.12 Satisfaction of Aim

Aim: *Develop an AI Agent Using Goal Oriented Action Planning that can Demonstrate Effective Solution Finding in Distinct World Environments.*

The extent to which I achieved my objectives set in Section 1.4 can be validated through my testing and results yielded in Chapter 4 and Chapter 5, respectively. As mentioned in Chapter 2, GOAP is fundamentally applying a graph searching algorithm to a state machine.

The resulting program causes the AI makes decisions based on the output from the A* algorithm, determining the least costly solution based on the current resources that the AI possesses. Even after the algorithm calculates the best path, there are further checks to see if there are enough objects available to perform the given action, which may cause the path to be further modified. In addition to this, the agent can perform these action in three different game world scenarios, where varying levels of resources cause the AI to perform the most efficient action in each environment.

As a result, I have managed to '*Develop an AI can Demonstrate Effective Solution Finding in Distinct World Environments'*, thus meeting my aim.

6.2 Improvements for the Future

Through reflection on the final AI there are a few areas that I would like to see modified and some additions I could add to the project:

1) One area of GOAP that I did not address was programming certain actions that were available purely to one game world. I could have included a world specific resource that would result in the AI choosing a new action that could not be performed in the others. For example, I a 'timber' resource which collecting was more favourable than any of the other actions to make a campfire but could only be found in the forest world.

2) Another improvement I would like to add, as mentioned in 6.11, is adding natural events to the game worlds. An example of this could be a weather storm, that causes all trees to be destroyed. Such events would further force the agent to adapt to the new situation, where the availability of certain resources has suddenly been affected.

3) I would have liked to include goals that contain more than 2 pathways, as mentioned in Section 6.11. The action Eat() could be made much more complex, including a variety of methods to reach the destination, and so this action in Figure 67 would include multiple paths as seen with MakeFurnace() and MakeCampfire().

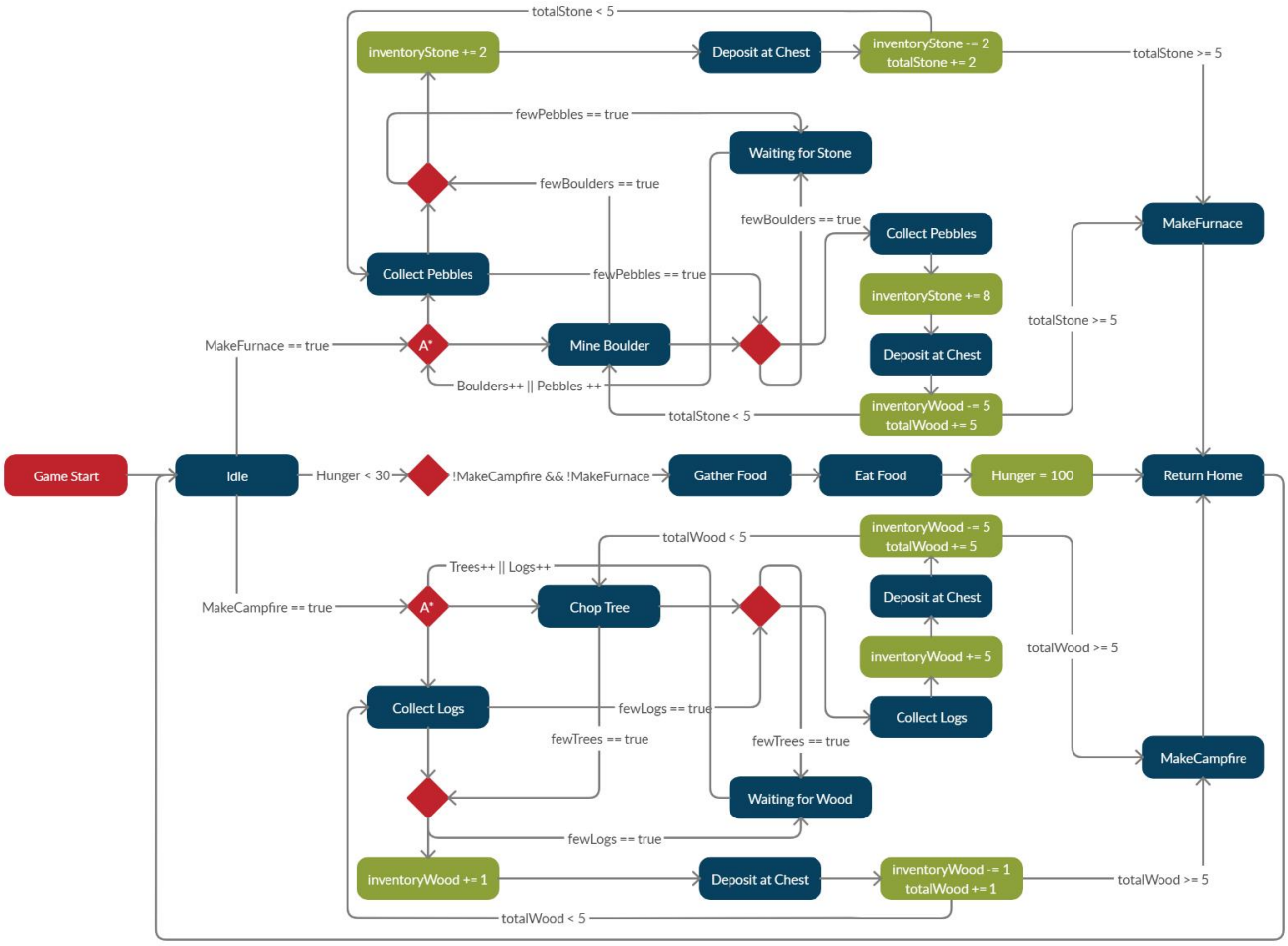


Figure 67: State Machine

4) The fix for the deadlock situation in the AI is inadequate, requiring the user to change the resource level (or activate the 'Reset' button) to escape the situation. I would like the program to be able to determine when a possible deadlock scenario is reach and give the AI methods of leaving the current state, in place of user interaction.

5) In terms of evaluating the correctness of my state machine model, I would use an existing framework to formally prove the conditions of my model. One such way is the addition of Hennessy Milner Logic. This can confirm multiple proofs of a model, including requiring users to formally prove that a final configuration will always be reached or testing if our model is deadlock free.

If I were to take this project further, I would have liked to include the additions above, aiming to create something akin to AI seen in industry, in games such as F.E.A.R. and Just Cause 2. However, my final project met the standards I set and was able to accomplish my aim.

References

All my references are automatically generated following Harvard style.

Resources

[1] En.wikipedia.org. (2019). F.E.A.R. (video game). [online] Available at: https://en.wikipedia.org/wiki/F.E.A.R._(video_game) [Accessed 28 Nov. 2019].

[2] Medium. (2019). Goal Oriented Action Planning. [online] Available at: https://medium.com/@vedantchaudhari/goal-oriented-action-planning-34035ed40d0b [Accessed 27 Nov. 2019].

[3] Owens, B. (2019). Goal Oriented Action Planning for a Smarter AI. [online] Game Development Envato Tuts+. Available at: https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-actionplanning-for-a-smarter-ai--cms-20793 [Accessed 27 Nov. 2019].

[4] En.wikipedia.org. 2020. Integrated Development Environment. [online] Available at: <https://en.wikipedia.org/wiki/Integrated_development_environment> [Accessed 5 April 2020].

[5] En.wikipedia.org. 2020. *Nim*. [online] Available at: <https://en.wikipedia.org/wiki/Nim> [Accessed 3 April 2020].

[6] En.wikipedia.org. 2020. Artificial Intelligence In Video Games. [online] Available at: <https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games> [Accessed 3 April 2020].

[7] Davison, Richard: Notes for CSC3222, Newcastle University, 2019

[8] Alumni.media.mit.edu. 2020. *Goal-Oriented Action Planning (GOAP)*. [online] Available at: <http://alumni.media.mit.edu/~jorkin/goap.html> [Accessed 3 April 2020].

[9] En.wikipedia.org. 2020. Automata Theory. [online] Available at: <https://en.wikipedia.org/wiki/Automata_theory> [Accessed 5 April 2020].

[10] Merriam-webster.com. 2020. *Definition Of AUTOMATON*. [online] Available at: <https://www.merriam-webster.com/dictionary/automaton> [Accessed 5 April 2020].

[11] Medium. 2020. What Is A Finite State Machine?. [online] Available at: <https://medium.com/@mlbors/what-is-a-finite-state-machine-6d8dec727e2c> [Accessed 5 April 2020].

[12] En.wikipedia.org. 2020. Deterministic Finite Automaton. [online] Available at: <https://en.wikipedia.org/wiki/Deterministic_finite_automaton> [Accessed 4 April 2020].

[13] Sakharov.net. 2020. Finite State Machines. [online] Available at: <http://sakharov.net/fsmtutorial.html> [Accessed 4 April 2020].

[14] En.wikipedia.org. 2020. Finite-State Machine. [online] Available at: <https://en.wikipedia.org/wiki/Finite-state_machine> [Accessed 4 April 2020].

[15] Thecompletecodes.com. 2020. Automata Theory | Deterministic Finite Automata (DFA) | Transition Table | Transition Diagrams | Language Of DFA | Theory Of Computation (TOC). [online] Available at: <https://www.thecompletecodes.com/2019/08/automata-theory.html> [Accessed 7 April 2020].

[16] Bevilacqua, F., 2020. Finite-State Machines: Theory And Implementation. [online] Game Development Envato Tuts+. Available at: <https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867> [Accessed 4 April 2020].

[17] Alumni.media.mit.edu. 2020. [online] Available at: <http://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf> [Accessed 3 April 2020].

[18] En.wikipedia.org. 2020. Pushdown Automaton. [online] Available at: <https://en.wikipedia.org/wiki/Pushdown_automaton> [Accessed 5 April 2020].

[19] Tutorialspoint.com. 2020. Pushdown Automata Introduction - Tutorialspoint. [online] Available at: <https://www.tutorialspoint.com/automata_theory/pushdown_automata_introduction.htm> [Accessed 5 April 2020].

[20] Web.stanford.edu. 2020. [online] Available at: <https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/17/Small17.pdf> [Accessed 5 April 2020].

[21] En.wikipedia.org. 2020. Turing Machine. [online] Available at: <https://en.wikipedia.org/wiki/Turing_machine> [Accessed 5 April 2020].

[22] 2020. [online] Available at: <https://www.researchgate.net/figure/An-interactive-Turing-machine-with-advice_fig2_272684665> [Accessed 7 April 2020].

[23] Cl.cam.ac.uk. 2020. Department Of Computer Science And Technology – Raspberry Pi: Introduction: What Is A Turing Machine?. [online] Available at: <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html> [Accessed 7 April 2020].

[24] Medium. 2020. *A-Star (A*) Search Algorithm*. [online] Available at: <https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb> [Accessed 11 April 2020].

[25] GeeksforGeeks. 2020. *A* Search Algorithm - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/a-search-algorithm/> [Accessed 11 April 2020].

[26] 101 Computing. 2020. *A* Search Algorithm | 101 Computing*. [online] Available at: <https://www.101computing.net/a-star-search-algorithm/> [Accessed 11 April 2020].

[27] www.javatpoint.com. 2020. *Uninformed Search Algorithms - Javatpoint*. [online] Available at: <https://www.javatpoint.com/ai-uninformed-search-algorithms> [Accessed 12 April 2020].

[28] Matwbn.icm.edu.pl. 2020. [online] Available at: <http://matwbn.icm.edu.pl/ksiazki/cc/cc35/cc3536.pdf> [Accessed 12 April 2020].

[29] compare?, H., 2020. *How Does Dijkstra's Algorithm And A-Star Compare? - Intellipaat Community*. [online] Intellipaat.com. Available at: <https://intellipaat.com/community/940/how-does-dijkstras-algorithm-and-a-star-compare> [Accessed 12 April 2020].

[30] Brilliant.org. 2020. *Dijkstra's Shortest Path Algorithm | Brilliant Math & Science Wiki*. [online] Available at: <https://brilliant.org/wiki/dijkstras-short-path-finder/> [Accessed 12 April 2020].

[31] CodinGame. 2020. *Coding Games And Programming Challenges To Code Better*. [online] Available at: <https://www.codingame.com/playgrounds/1608/shortest-paths-with-dijkstras-algorithm/dijkstras-algorithm> [Accessed 12 April 2020].

[32] Programiz.com. 2020. *Bellman Ford's Algorithm*. [online] Available at: <https://www.programiz.com/dsa/bellman-ford-algorithm> [Accessed 12 April 2020].

[33] Techie Delight. 2020. *Single-Source Shortest Paths – Bellman Ford Algorithm - Techie Delight*. [online] Available at: <https://www.techiedelight.com/single-source-shortest-paths-bellman-ford-algorithm/> [Accessed 12 April 2020].

[34] GeeksforGeeks. 2020. *Bellman–Ford Algorithm | DP-23 - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/> [Accessed 12 April 2020].

[35] GitHub. 2020. Luxkun/Regoap. [online] Available at: <https://github.com/luxkun/ReGoap> [Accessed 25 April 2020].

[36] Cs.cmu.edu. 2020. *Models Of Computation*. [online] Available at: <http://www.cs.cmu.edu/~ref/pgss/lecture/12/index.html> [Accessed 21 April 2020].

[37] Apiar.org.au. 2020. [online] Available at: <https://apiar.org.au/wp-content/uploads/2018/10/16_APCCR_Aug18_BRR748_ICT_v2_50-56.pdf> [Accessed 7 April 2020].

[38] En.wikipedia.org. 2020. Halting Problem. [online] Available at: <https://en.wikipedia.org/wiki/Halting_problem> [Accessed 7 April 2020].

[39] GeeksforGeeks. 2020. Greedy Approach Vs Dynamic Programming - Geeksforgeeks. [online] Available at: <https://www.geeksforgeeks.org/greedy-approach-vs-dynamic-programming/> [Accessed 21 April 2020].

[40] GeeksforGeeks. 2020. *What Are The Differences Between Bellman Ford's And Dijkstra's Algorithms? - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/what-are-the-differences-between-bellman-fords-and-dijkstras-algorithms/> [Accessed 21 April 2020].

[41] Hindex.org. 2020. [online] Available at: <http://www.hindex.org/2014/p520.pdf> [Accessed 21 April 2020].

[42] CodePlex Archive. 2020. Codeplex Archive. [online] Available at: <https://archive.codeplex.com/?p=quickgraph> [Accessed 26 April 2020].

[43] HackerEarth. 2020. *Shortest Path Algorithms Tutorials & Notes | Algorithms | Hackerearth*. [online] Available at: <https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/> [Accessed 12 April 2020].

[44] Brilliant.org. 2020. *Bellman-Ford Algorithm | Brilliant Math & Science Wiki*. [online] Available at: <https://brilliant.org/wiki/bellman-ford-algorithm/> [Accessed 12 April 2020].

[45] Alumni.media.mit.edu. 2020. [online] Available at: <http://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf> [Accessed 23 April 2020].

Scripts

Unity3d.com. (2012). FramesPerSecond - Unify Community Wiki. [online] Available at: http://wiki.unity3d.com/index.php?title=FramesPerSecond&_ga=2.15151733.2136993790.1580487035-501062010.1574528390 [Accessed 31 Jan. 2020].

A* algorithm code adapted from:

GitHub. 2020. *Davecusatis/A-Star-Sharp*. [online] Available at: <https://github.com/davecusatis/A-Star-Sharp/blob/master/Astar.cs> [Accessed 15 April 2020].

Assets

Assetstore.unity.com. (2019). *Low Poly Survival Essentials - Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/3d/props/tools/low-poly-survival-essentials-109444> [Accessed 24 Nov. 2019].

Assetstore.unity.com. (2019). *LowPoly Water - Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/tools/particles-effects/lowpoly-water-107563> [Accessed 24 Nov. 2019].

Assetstore.unity.com. (2019). *Free Island Collection - Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/3d/environments/landscapes/free-island-collection-104753> [Accessed 24 Nov. 2019].

Assetstore.unity.com. (2019). *Low-Poly Resource Rocks - Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/3d/props/exterior/low-poly-resource-rocks-76150> [Accessed 24 Nov. 2019].

Assetstore.unity.com. (2019). *Free Trees - Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/3d/vegetation/trees/free-trees-103208> [Accessed 24 Nov. 2019].

Assetstore.unity.com. 2020. Forest - Low Poly Toon Battle Arena / Tower Defense Pack | 3D Environments | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/environments/forest-low-poly-toon-battle-arena-tower-defense-pack-100080> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. Winter Mountains And Stamps | 3D Landscapes | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/environments/landscapes/winter-mountains-and-stamps-129245> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. Stylized 3D Tools | 3D Tools | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/props/tools/stylized-3d-tools-91080> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. *Fallen Tree Barrier - Free | 3D | Unity Asset Store*. [online] Available at: <https://assetstore.unity.com/packages/3d/fallen-tree-barrier-free-49089> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. Robot Sphere | 3D Robots | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/characters/robots/robot-sphere-136226> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. LOWPOLY MEDIEVAL WORLD - Lowpoly Medieval Peasants | 3D Humanoids | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/characters/humanoids/lowpoly-medieval-world-lowpoly-medieval-peasants-122225> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. Winter Mountains And Stamps | 3D Landscapes | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/environments/landscapes/winter-mountains-and-stamps-129245> [Accessed 18 April 2020].

Assetstore.unity.com. 2020. Hand Painted Nature Kit LITE | 3D Environments | Unity Asset Store. [online] Available at: <https://assetstore.unity.com/packages/3d/environments/hand-painted-nature-kit-lite-69220#content> [Accessed 20 April 2020].

Appendices

Goal Oriented Action Planning (GOAP) – An artificial intelligence system for autonomous agents that allows them to dynamically plan a sequence of actions to satisfy a set goal. [3]

Integrated Development Environment (IDE) – A software application that provides comprehensive facilities to computer programmers for software development. [4]

Automatons – A machine or control mechanism designed to automatically follow a predetermined sequence of operations or respond to encoded instructions. [8]

Finite State Machines (FSM) – A Finite State Machine, or FSM, is a computation model that can be used to simulate sequential logic, or, in other words, to represent and control execution flow. [10]

Finite Automaton (FA) - An FSM that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string. [11]

Deterministic Finite Automaton (DFA) – A FA, but one for which every state every unique transaction can move only to itself or at most one other state.

Pushdown Automaton (PDA) - A finite automata with extra memory called stack which helps pushdown automata to recognize Context Free Languages. [16]

Turing Machine - A Turing machine is an abstract computational model that performs computations by reading and writing to an infinite tape. [19]

Halting Problem – The problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever. [25]

Uninformed Search Algorithms - Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search. [29]

Negative Weight Cycle – A cycle which will reduce the total path distance by coming back to the same point. [35]

Greedy Algorithm - A Greedy algorithm is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit [40].

Dynamic Programming - Dynamic programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming [40].

Personal Resources

GitHub, containing scripts and csv files:

GitHub. 2020. Samuelappleby/Goalorientedactionplanning. [online] Available at: <https://github.com/SamuelAppleby/GoalOrientedActionPlanning> [Accessed 23 April 2020].

OneDrive account, containing complete project folder and executable build version:

Newcastle-my.sharepoint.com. 2020. Onedrive For Business. [online] Available at: <https://newcastle-my.sharepoint.com/:f:/g/personal/b7034806_newcastle_ac_uk/En98Wf7HFuVKmgRxWsCaRNIB930Aeu_66ayd724kOp0DiA?e=sHi1zv> [Accessed 23 April 2020].