Minecraft as an Experimental World for AI Agents

Marlon Calleja

June, 2017

Submitted to the Institute of Information and Communication Technology in partial fulfilment of the requirements for the degree of B.Sc. (Hons.) Software Development

Authorship Statement

This dissertation is based on the results of research carried out by myself, is my own composition, and has not been previously presented for any other certified or uncertified qualification.

The research was carried out under the supervision of Mr. Gerard Said

Copyright Statement

In submitting this dissertation to the MCAST Institute of Information and Communication Technology I understand that I am giving permission for it to be made available for use in accordance with the regulations of MCAST and the College Library.

Marlon Calleja

17, Santa Marija, Triq l- Imnarja, Birzebbuga BBG2564

Acknowledgements

Mr. Gerard Said who supported me during my dissertation work as my tutor. My family for their patience and loving encouragement. MCAST for administering the educational courses made available to myself, with particular emphasis on my lecturers over the course of this degree. My friends for their morale support and motivation.

Table of Contents

Abstract 1

Chapter 1 - Introduction 2

1.1 Motivation 2

1.2 Objectives 3

1.3 Artificial Intelligence 3

Chapter 2 - Literature Review 4

2.1 General AI Model 4

2.2 Pathfinding 5

2.3 A-Star Search Algorithm 6

2.4 Jump Point Search Algorithm 6

2.5 Navigation Mesh 7

2.6 Observation Grid 8

2.7 Machine Learning 10

2.8 Genetic Algorithms 11

2.9 Behaviour Trees 11

2.10 Utility Systems 15

2.11 Goal Based Actions 18

2.12 Subgoal Selection 19

Chapter 3 - Research Methodology 22

3.1 Prototype 22

3.2 API and Workspace Environment 23

3.3 Framework 23

3.4 AI Agent 24

3.5 Action Planning 25

3.6 Action Setup 25

3.7 Action Pool 28

3.8 Decision Tree 30

3.9 Traversal Algorithms 33

3.9.1 A-Star Search 33

3.9.2 Breadth First Search 33

3.9.3 Depth First Search 34

3.10 Scenario 36

Chapter 4 - Findings 39
4.1 Decision Tree Population 39
4.2 Comparison of similar map settings 42
4.3 Tree traversal algorithms for solution finding 45
4.4 Comparison of solution results 49
4.5 Comparison of traverse time and running cost 49
4.6 Comparison of similar tree structure with fewer solution nodes 52
Chapter 5 - Discussion of Results 53
5.1 Decision Tree Generation 53
5.2 Traversal Algorithms 54
5.2.1 A-Star 54
5.2.2 BFS 55
5.2.3 DFS 56
Chapter 6 - Conclusion 58
6.1 Objectives 58
6.2 Algorithm Utility 58
6.3 Prototype Evaluation 58
6.4 Future work 59
Bibliography 62
Appendix 64
System Specifications 64
Decision Tree - Visual Representation 65

List of Tables

Table 2.1 - Behaviour Tree Mapping 15

Table 3.1 - AI Agent Actions (Action pool) 28

Table 4.1 - Decision Tree Population 39

Table 4.2 - Tree Summary 41

Table 4.3 - Tree Population of Similar Maps 43

Abstract

With modern video games becoming increasingly dependent on the behavioural nature of non-player characters (NPCs), the need for smarter AI agents to take the next best action in a complex environment is pressing.

The study led to the exploration of existing methods, employing similar techniques to goal oriented action planning (GOAP), to craft an AI agent within a sandbox game environment. In this case Minecraft was chosen for its open-world setting, allowing for free form experimentation. An action pool was employed to instruct the AI agent's behaviour, implementing logic in populating decision trees with available actions in the direction of reaching a set goal. Several tree traversal algorithms were tested in finding a solution from the generated decision tree, taking world-state data into account.

A-Star (A*) search, being a pathfinding algorithm, its application onto a decision tree has allowed the assignment of costs to each step in the action sequence, being performed by the AI agent. The study concluded that the application of a pathfinding algorithm is still worth employing in the context of building an AI agent, even though it takes longer to compute than Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms in the majority of cases, for having to calculate action cost.

Retrieval of the best solution from the tree is always guaranteed, obtaining an action sequence with the least number of steps required to reach a set goal. Turn-based games or any setting which dedicates time for the NPC to 'think' and decide about a solution is the ideal instance to employ A-Star search. Real-time games involving fast paced combat hinders the algorithm's efficiency.

Chapter 1 - Introduction

The field of artificial intelligence (AI) is vast when taken as a whole, however general concepts can be applied into video game mechanics to develop better experiences. Modern video games have become increasingly dependent on the behavioural nature of non-player characters (NPCs). Hence, more importance has been given to AI programming, for the behaviour of such independent agents involving pathfinding, decision making and action planning. Players constantly demand fresh engaging content from uniquely designed characters, expecting different interactions. This includes, how enemies manage their actions and movement within the game-world environment. All this can be hard coded within the system, but the level of 'intelligence' a character can reach is limited. With artificial intelligence, agent behaviour has no boundaries when the correct methods are applied.

1.1 Motivation

The motivation for this thesis mainly constitutes building a better understanding both of how current independent agents form their behavioural strategies, as well as ascertaining how best to implement an algorithmic structure to govern independent agents in the context of a sandbox game. Most game reviews argue about the same factor, being that certain games can only reach a certain level of difficulty. The issue is derived mainly from weak AI programming, allowing the player to beat NPCs merely easily. If the computer's movement become predictable after a few minutes of game-time, it will become boring and won't offer any challenge to the player. Also, the game will be beaten shortly despite hours spent in development to produce the final product. For this reason, available AI techniques were researched for the possibility to further advance NPC behaviour, replicating human-like actions if necessary.

1.2 Objectives

This study intends to provide analysis on game AI, on how modern techniques can improve independent agents and manipulate NPC behaviour. A sandbox game environment will be used for testing, in this case Minecraft. By employing pathfinding techniques in decision trees, the AI agent will be allowed to search the next action towards a specified goal. Using a variety of tree traversal algorithms for solution finding, their selection behaviour will be compared across similar tree structures.

1.3 Artificial Intelligence

AI agent behaviour is tied to 3 factors which must be balanced to obtain best results:

- Memory space the system occupies
- Time taken to compute a solution
- Level of intelligence

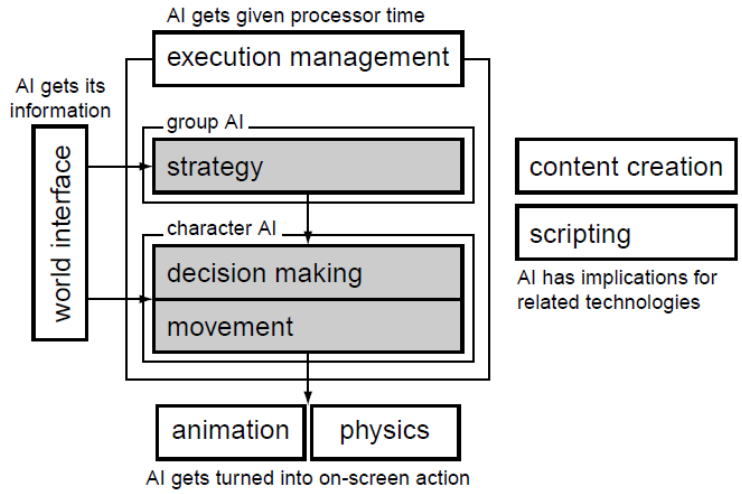
The system's intelligence can be rated by different attributes concerning AI, this includes how fast the system learns new techniques and how it behaves to changing environment. Overall amounting to how human-like its reaction is, presenting an unpredictable nature when choosing actions. If a high level of intelligence it reached, players will have the illusion of playing against another person instead of a machine.

Chapter 2 - Literature Review

This literature review will focus on AI relating directly to video games, and will provide a more detailed description of path finding techniques and agent behaviour, including decision making.

2.1 General AI Model

Movement, decision making, and strategy are three main components which compile a general mission for AI, as per (Millington, 2006). Not all video games implement all levels of AI, board games focus more on the strategy element to coordinate a group of units while platform games focus more on character movement. The following diagram illustrates a general model for artificial intelligence.



The strategy component is mainly used to coordinate multiple AI units at the same time. Algorithms in this category influence the behaviour of a whole team or set of characters. Each character in the group usually has its own decision making and movement algorithms, while being influenced by a group strategy.

The decision making component is involved with the AI agent sensing the environment, performing some background reasoning and acting on the present scenario. A set of defined rules will allow the character to work out what to do next, selecting the ideal action to perform from its knowledge base.

Pathfinding algorithms are all tied to the movement component which allows the character AI to move strategically on a given surface while obeying world dimensions and game controls.

2.2 Pathfinding

Non-player characters (NPCs) usually need to find their way around a level. A patrolling path can be hard coded, however this can cause problems when an object is placed in the way since the character is 'blind' in nature, causing NPCs to get stuck unintentionally. By implementing a pathfinding algorithm, characters can dynamically calculate their shortest path to an end point, rerouting their way around present objects, hence making them act more 'intelligently'.

Different representations for pathfinding techniques exist:

- Grids
- Waypoint graphs
- Navigation meshes (Triangulation or polygonization)

The general approach taken in pathfinding to find an optimal path for a given game character is comprised of 3 stages. (Cui et al, 2012). First the game world needs to be transformed into a geometric representation, choosing one technique from the above mentioned options. A pathfinding algorithm then acts on the generated mesh or nodes to find a preliminary path. Further processing is done in the final stage to find the 'real path' which guarantees the shortest route to reach a goal position.

2.3 A-Star Search Algorithm

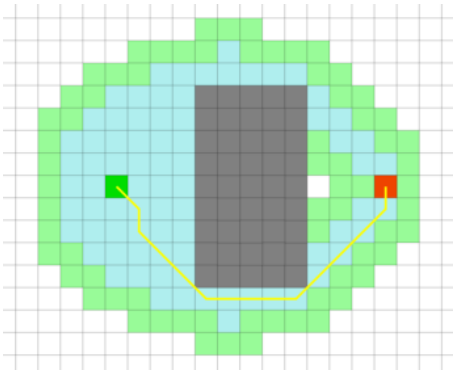
A-Star (A*) search is designed for point-to-point pathfinding, widely used in games for its simplistic implementation nature and scope for optimization. (Millington, 2006). The algorithm works in a similar way as Dijkstra algorithm, rather than considering the open node with the lowest cost, the node that is most likely to lead to the shortest overall path is chosen. The calculation is controlled by a specified heuristic, working in iterations to evaluate surrounding nodes and their connections, pinning an F cost to each. Starting from a particular position, the node with the lowest F cost are always chosen, surrounding nodes will persist with evaluation until the target position is reached, eventually finding the ideal path.

Hash tables are the best data structure option for storing initialised nodes, allowing for constant time storing, fast retrieval and look up of data, as per (Cui et al, 2011). The table keeps record of nodes which are either in the 'closed' or 'open' list. Nodes in the open set are those nodes which still need to be explored, nodes in the closed set are those nodes which have already been evaluated. To maintain these lists, a priority queue can be implemented using a binary heap.

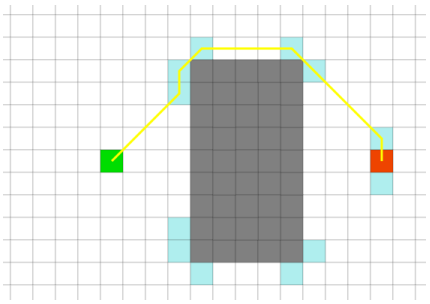
2.4 Jump Point Search Algorithm

Jump Point Search is an optimised pathfinding algorithm to A* for grid based representations. (Harabor et al, 2011). Using same start and end points, JPS results in lesser operations, taking less time to compute when applied in large scale situations. When symmetrical paths are present, A* algorithm tends to perform needless search calculation. The Manhattan distance heuristic is limited to 4-direction movement, hence is not favoured if diagonal movement is allowed. To simulate 3D game environment, the Euclidean distance heuristic is used in both cases:

A-Star algorithm:



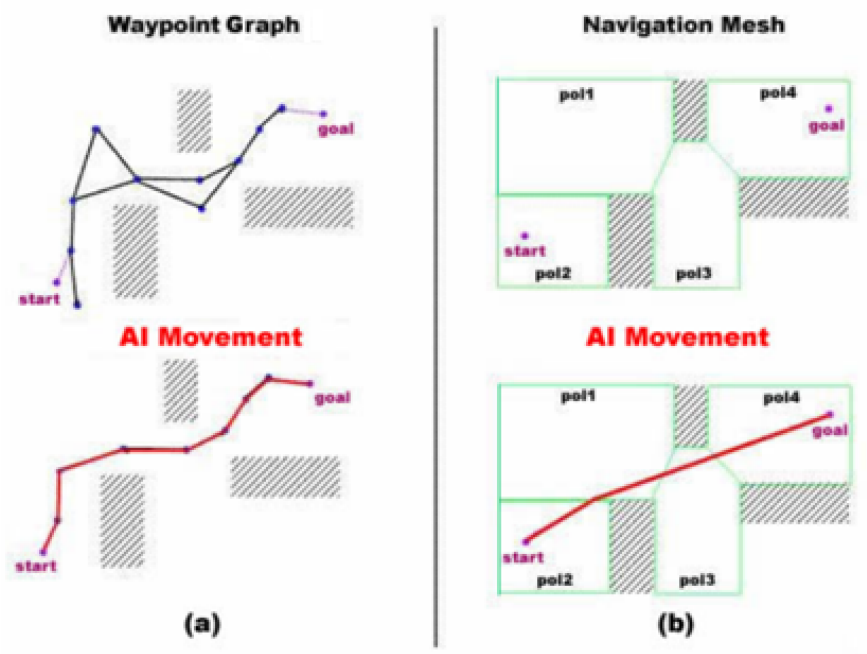
Jump Point Search (JPS) algorithm:



Using purely local node information the algorithm reasons about symmetry using Pruning and Stopping Rules. Unlike A* algorithm, Jump Point Search performs symmetry breaking during pathfinding which reduces the noise generated when calculating the optimal path.

2.5 Navigation Mesh

A NavMesh is another technique used in AI pathfinding. It splits the map into multiple polygons which describe the movable terrain of an environment, providing a surface plan for the AI character to navigate on. (Cui et al, 2011). Compared with a waypoint graph, the navigation mesh pathfinding movement behaviour matches more to an actual human, is much more effective and can find a near optimal path by searching much less data.

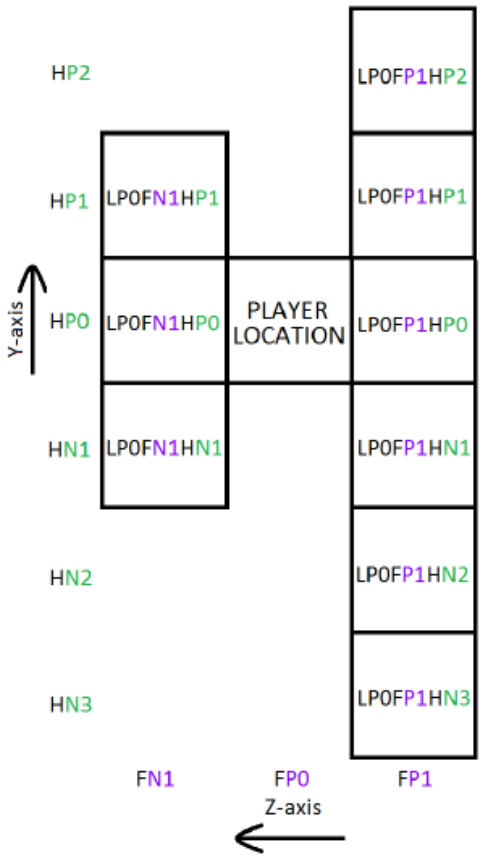


Using the generated NavMesh to reflect where a character can safely walk, its path can be smoothed out with splines, eliminating any 'zig-zag' nature which waypoints introduce. Navigation mesh simplicity means that lesser nodes need to be searched when calling a pathfinding algorithm at runtime, hence resulting in faster character reaction time. Moreover, characters with different dimensions and movement physics may adjust dynamically to the mesh instead of having to generate individual waypoint network for each.

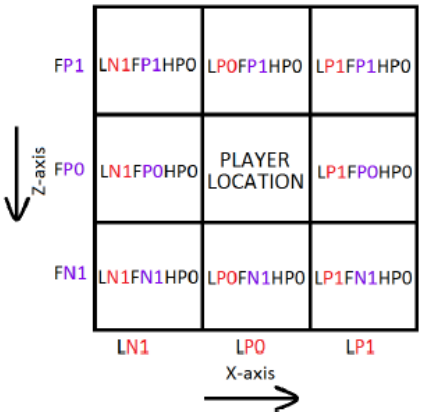
2.6 Observation Grid

AI agents are initially 'blind' to the game environment, some kind of observation system or sensory is required for the agent to make decisions based on the provided data. Observations can take the form of a grid (based on the world representation), showing a set of game states around the character that are in proximity range, representing movable terrain or surrounding objects. (Bonanno et al, 2016). The following diagram shows observable blocks around the character, from a respective view angle:

Side view (Player is facing right)



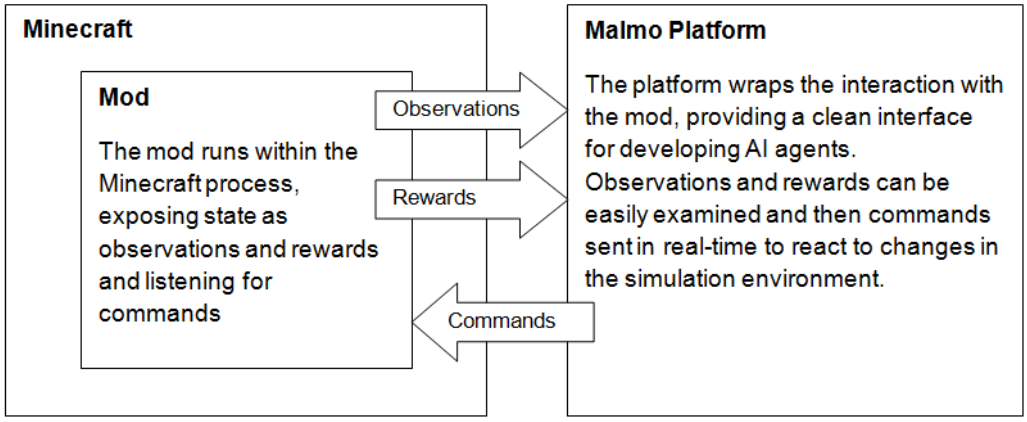
Top view (Player is facing up)



Observation from grid arguably constitutes cheating in some cases, if the game is designed to be played in day and night cycles which dramatically alter the visual world environment i.e. a room without torches will be pitch-black at night, hence will be difficult for a normal player to navigate in the surroundings when compared to an agent having written state information. Visual producer can be used as an observation for agents to make decisions based purely on what they 'see'. (Abel et al, 2016). Pre-processing techniques of raw RGB observation can be employed to read the game state from visual images. Speeded-up Robust Features (SURF) key-point detection can be performed to recognise objects or detect interest points from the provided imagery which show the player's point-of-view. (Bay et al, 2008).

2.7 Machine Learning

A Microsoft research team was assembled to undergo a project, known as Project Malmo, originally developed by Hofmann K. For this project, a sandbox type game Minecraft is being utilised as a learning and testing ground for artificial intelligence. (Johnson et al, 2016). Since the game offers endless options for players to carry out, ranging from world exploration to building structures, using reinforcement learning an AI agent can be trained to perform a particular task, aiming to improve its overall behaviour. Minecraft is a great test-bed for artificial intelligence problems in general because it offers flexibility.



Malmo platform provides an intuitive API that allows AI experimentation on top of Minecraft. To perform alterations in source code, a third-party application 'Minecraft Forge' is being utilised to load mod packages, this tool will ensure mod compatibility between versions.

This platform was used by researchers to further extend the field of reinforcement learning. Gradient boosting was experimented with in a complex 3D domain, using pre-processed raw visual images from Minecraft to read the agent's state. (Abel et al, 2016). Boosting is a particular kind of ensemble method that combines weak learners (regressors) to produce an overall better learner. The algorithm works by using a very simple regression model as a foundation and then subsequent models are trained to predict the error residual of the previous predictions.

2.8 Genetic Algorithms

Artificial neural networks can evolve through genetic algorithms, Neuroevolution of Augmented Topologies (NEAT) is an example of such algorithm (Stanley et al, 2002). A neural network is a mathematical model for how a human brain works, simulating life behaviour and decision making based on present problems.

Nodes with different weights can be randomly generated and connected together to form a neural network. Every single node communicates with neighbouring nodes, each passing the required information to the next, until reaching the expected output. In a neural environment, parent nodes are used to manage the flow and direction of information from the given inputs.

The AI system can learn through evolution, allowing child species to adapt new techniques from their parents. The entities with the most fitness (score) are chosen for the next generation, creating a new population of species, hence the neural network is restructured in a way to act more 'intelligently'. This cycle will repeat itself until a satisfactory level of behaviour is reached, eventually reaching a plateau where it can no longer evolve (evolution will be minimal to notice any performance change), consequently matching the projected output.

2.9 Behaviour Trees

Initially Finite State Machines (FSM) were used as a model to implement AI in games, represent and control execution flow. FSMs are procedural in nature and tell a character exactly how to behave in every situation, hence the design can get increasingly complicated with expanding and open-world games, also becoming a nightmare to maintain. As an improvement over FSMs, planning systems implement a declarative technique. They are being used in modern games to tell AI characters what their goals and actions are, and let them decide how to sequence these actions to satisfy respective goals. Based on the work of Jeff Orkin, the first game which utilised planning techniques was a first-person shooter named F.E.A.R. (Orkin, 2006). The enemy AI uses a STRIPS planning algorithm to search through possible

actions, and eventually finding a world state which matches with a goal criteria. The search starts from the goal state rather than the current world state, searching backwards is faster than searching in a forward manner however can present less flexibility with complex goals.

Recent AI implementation shifted towards a more hierarchical approach, gaining additional design control and efficiency over character behaviour from using Behaviour Trees (BT) or Hierarchical Task Network (HTN) based on SHOP planner. (Champandard, 2013). BTs provide a hierarchical way of organising character behaviour in descending order by complexity, starting with broad tasks on top (the root behaviour) and splitting into sub-tasks at the bottom. (Peters et al, 2013). Each tree has one defined high level behaviour, being associated with a distinct goal which they needs to achieve. Multiple trees can be linked together to define an AI unit, hence by first defining smaller sub-behaviours, implementation of complex behaviours is made possible.

BTs implement a similar technique to hierarchical state machines however the main building block of a behaviour is a task rather than a state. They also integrate elements from HTN planners in which dependencies among actions are given in the form of networks, the main difference being that a BT does not search ahead of the current behaviour, it stops at an action which appears satisfactory to the present scenario (best-first search).

Behaviour Tree (BT) implementation became popular for AI modelling in video games like Halo and Spore which managed the complexity of their code into controlling various in-game characters. Façade is an interactive drama game where characters communicate with each other, enter dialogue with the player and behave in an emotional manner to the present scenario. (Mateas et al, 2003). Actions that the player performs have both immediate and long-term effects on the story's agents. The story is split into a number of 'beats' which mainly consist of preconditions and rules that influence the characters' goals and behaviour. Different situations are created which may or may not occur in a single run through of the game, thus presenting a less predictable nature and making the game more interesting for the player.

This shows that behaviour techniques can be applied to different genres of games, hence presenting design flexibility. BTs are simpler to design and implement over traditional AI approaches, easily scalable, modular and reusable in complex evolving games. Their ease of human understanding due to their graphical tree nature make BTs less error prone and popular in the game development area.

Behaviour Trees are composed from 2 types of constructs:

Primitive constructs form Leaf Nodes and define low level actions which describe the overall behaviour:

- Conditions - make queries about the game state.
- Actions - make decisions, carry out specific tasks which follow a sequence of actions and conditions.

Composite constructs form Control Nodes and are used to group Leaf Nodes to perform a higher-level function. They drive execution flow through the tree, controlling which nodes to execute next.

- Sequence - execute children nodes from left to right until one fails (AND gate logic).
- Selector - execute children nodes from left to right until one succeeds (OR gate logic).
- Decorators/Filter - modify the execution flow of attached nodes, whether or not a loop is performed or the result of a node is negated. Adds additional behaviour to an existing method without modifying that action's code.

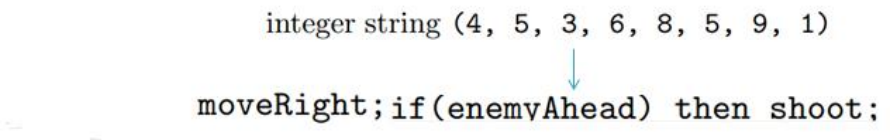
Evolutionary techniques to develop Behaviour Trees for competitive AI units were investigated for feasibility in a real-time strategy game, named DEFCON. (Lim et al, 2010). Originally a set of BTs were utilised to hand craft an AI-bot in order to cater for all basic actions that a player could perform. These BTs were used as a basis to produce more complex behaviours through evolution, altering the AI unit's behaviour with genetic operators and fitness functions.

Genetic Operators are used in conjunction with tree structures to naturally modify their behaviours. Mutation on nodes can act in different ways, can have random mutation, incremental mutation which adds a new branch to a behaviour, or point mutation which alters a game bit nature (Change the spawn point of structure or unit).

Fitness Functions are defined as rules for choosing BTs with high game score. Credit will be awarded to overall behaviour which ended the game with a positive game score or met a set mission. Randomly generated trees for individual behaviours are evolved, the best performing trees are chosen and combined to form an advanced AI control unit. Experimental results indicate that the average score of each fitness function increased as more generations were produced, producing a higher win rate percentage.

The applicability of Genetic Programming to evolve BTs was investigated with Mario AI. (Perez et al, 2011). Grammar Based Genetic Programming systems such as Grammatical Evolution can be used as an extension to previous work, this simplifies the task of syntax encoding for BTs, reducing their complexity. Moreover, specific structures such as and-or logic trees can be easily specified to define actions. Syntax of possible solutions can be specified through a context free grammar, syntactically correcting solutions by mapping binary strings. The following code shows an example of the grammar to a generic shooting game:

```
<BT>        ::= <BT> <Node> | <Node>
<Node>      ::= <Condition> | <Action>
<Condition> ::= if(obstacleAhead) then <Action>;
              | if(enemyAhead) then <Action>;
<Action>    ::= moveLeft; | moveRight; | jump; | shoot;
```



Integer strings of variable length evolve following a genetic algorithm, these then choose the production rules from the grammar until all symbols are mapped. The following table shows detailed steps for the mapping process, given an Integer string of (4, 5, 3, 6, 8, 5, 9, 1):

Table 2.1 - Behaviour Tree Mapping

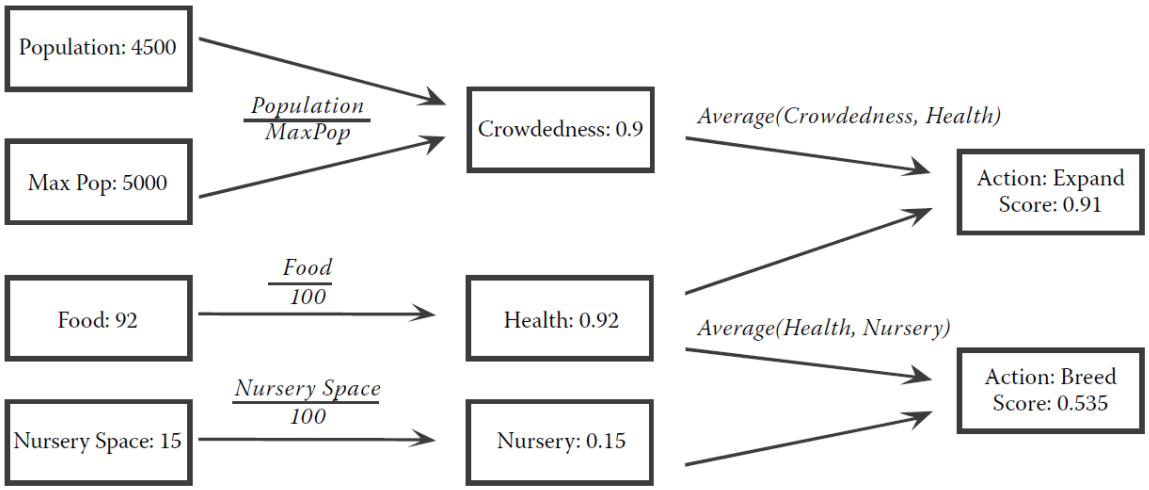
Mapping String		Description
(4, 5, 3, 6, 8, 5, 9, 1)		Integer string
<BT> 4%2 = 0		Use first value '**4**' from provided integer string, compute **4**%2 formula. Following production rules for <BT> symbol, result '0' signifies <BT><Node> production while '1' signifies <Node> production.
<BT> 5%2 = 1	<Node>	Result was '0' so mapping string becomes <BT><Node>. Compute 5%2 formula on <BT> symbol.
<Node> 3%2 = 1	<Node>	Result was '1' so the <BT> symbol is replaced by <Node>, thus mapping string becomes <Node><Node>. Compute 3%2 formula on left <Node> symbol.
<Action> 6%5 = 1	<Node>	Result was '1' so the left <Node> symbol is replaced by <Action>, thus mapping string becomes <Action><Node>. Compute 6%5 formula on <Action> symbol.
moveRight	<Node>	Result was '1' so the <Action> symbol is replaced by moveRight, thus mapping string becomes moveRight; <Node>.
. . .	. . .	More steps involved
moveRight; if(enemyAhead) then shoot;		After all symbols are mapped, the final program becomes moveRight; if(enemyAhead) then shoot.

The resulting solutions are easy for humans to understand and analyse, the combination of a Genetic Programming type algorithm with BTs provides flexibility to fine-tune AI behaviour towards evolutionary approaches. This shows that a carefully designed syntax can accelerate the evolution process when defining and exchanging meaningful behaviour blocks.

2.10 Utility Systems

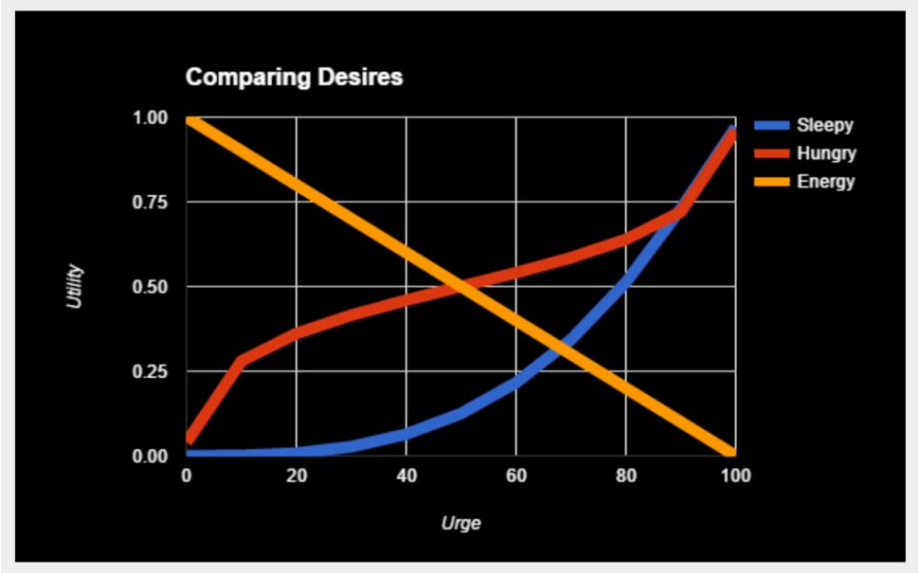
Next generation games are expanding and becoming more complex in nature to design, their demand for AI requirements growing by the second. Rival companies are still researching for better game AI techniques even with Behaviour Trees being a great solution. Utility systems are a new addition to AI decision making, they assign a 'score' value to distinct goals and evaluate what is mostly important to the present scenario. (Mark et al, 2010). Actions with a higher utility score are chosen over those with lower utility.

Utility scores need to be kept consistent across the entire decision making system so values can be easily compared and calculated, normalising scores from a range of 0 to 1 is a practical option. (Graham, n.d.). Moreover, a common technique to determine the score of an action is to multiply the utility score by the probability of each possible outcome and summing up these weighted scores. This provides the expected utility of the respective action. As an example, a player which is currently engaged in melee combat has a 60% chance of hitting the target with a weapon having a utility score of 0.8, hence the expected utility score is adjusted to 0.48. A decision normally relies on a combination of data, hence each piece of information is a decision factor that needs to be weighed into calculation of the final utility score.



The above diagram shows how the utility score of different decision factors is combined to obtain the final score. By averaging the normalised scores together, an endless chain of combinations can be built.

A good understanding of the relationship between the input and the output for a given action is required, as to write a correct utility function which can describe an arbitrary game value as a resulting curve, and thus allow conversion to utility score. The key to utility theory is to choose the best formula which has a realistic calculation to the respective context, some being Linear or Quadratic curves as a basic example. When designers need to fine-tune AI behaviour, custom curves such as piecewise linear curves come into play.



The application of 'curves' in Utility AI was outlined by Dave Mark, showing how this method can be used to score different options available to the AI. (Mark et al, 2010). Reflecting on an example from the above data, the urge of a character to sleep is based on the time since the last nap, hence the utility of sleep rises as the urge becomes stronger. Curve data enables the Utility AI to make decisions across a broad range of inputs, practically adjust for unforeseen events or perform actions without requiring knowledge of the game world in advance.

Utility AI methodology is very versatile for decision-making and its functionality is easily extendable when compared to BTs, as new branches cannot be added dynamically to the tree. Since its model is based solely on scores being recorded in a table there are no important relationships to break, values can change dynamically with the present environment or state. BTs can quickly grow in complexity when required to simulate hundreds of tactical positions for a given group of characters, however still remain feasible for simpler tasks.

2.11 Goal Based Actions

Further to this, there is the concept of a set of actions which can lead to a favourable outcome. Training for a generic set of actions could be similar to pathfinding but instead of movement would be the choice of action that the agent takes.

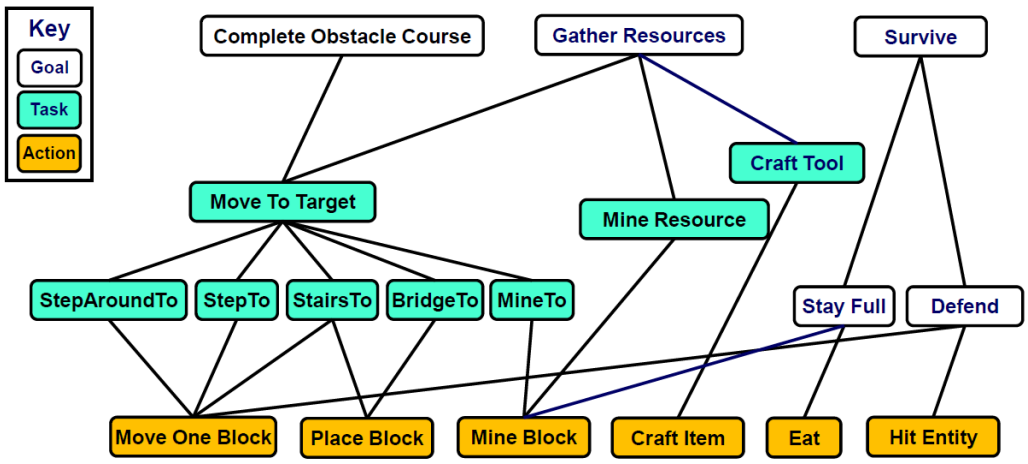
Placing an AI character in a large environment presents an endless list of possible actions to perform. (Abel et al, 2015). The size of these actions grown exponentially with the number of objects found in the scene. Depending on the goal, most of the actions may deem to be irrelevant to the scenario. The agent needs to be trained on a smaller scale problem before exposing it to a larger one. This can be explained with the following image:

The setting on the left requires less movement to smelt the gold block in the furnace. Both problems expect the same result, however mapping between actions is easier on the smaller version since less noise is present. The approach uses goal based actions priors to prune suboptimal actions on a state by state basis. This solves the problem when transferring knowledge acquired from a simple problem to a much harder one. These decision making problems are modelled following Markov Decision Process (MDP). By giving reward to favoured behaviour, the agent will learn to prioritise goals in a larger state-action space, hence pruning away irrelevant actions and dramatically reducing the time taken to find a near-optimal plan.

2.12 Subgoal Selection

Deep learning can be applied in computer vision to label images, extracting subgoals from a simulated environment which is presented as a raw array of pixels. Neural networks are formed which bridge gaps between highly complex representation of pixels in an image to their respective classification label which is easily understood by a human.

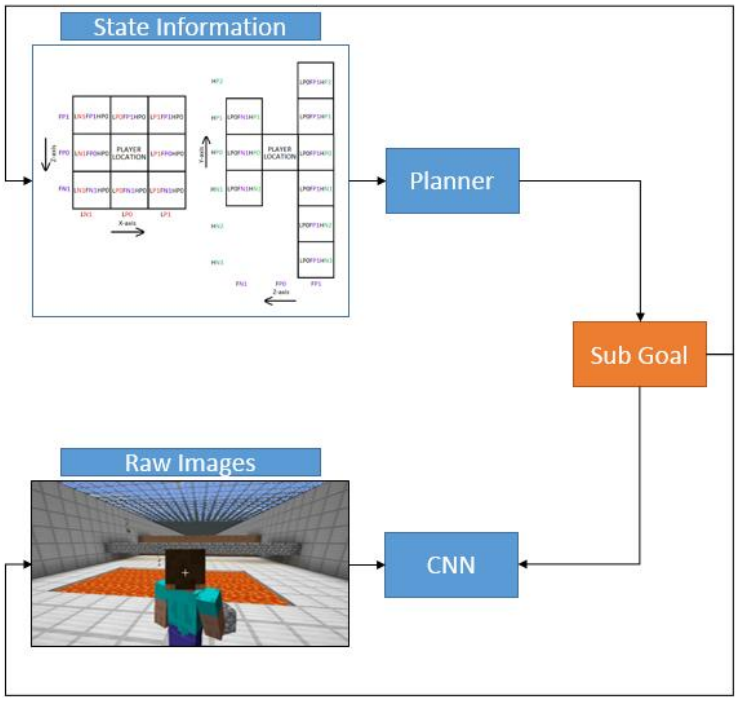
By combining Deep Learning with Reinforcement Learning the network can be tasked to accomplish a goal. (Bonanno et al, 2016). Selecting the appropriate sequence of actions revolving around moving or jumping can achieve a high score in a video game, which can be a desired goal for accomplishment. The system can recognize and control decisions through Deep Learning, using either a supervised (trained by a human expert) or unsupervised training technique to select goals. This technique however lacks performance in games which involve long sequences of events. Hierchical planning can be introduced to counter this weakness, providing a more sparse decision space for selecting among subgoals that deem to be useful.



After compiling a series of planning and goal reasoning models, complex tasks can be decomposed into combinations of subgoals or subtasks and represented in a Goal-Task network as shown above, providing a clearer picture of linked goals, tasks and actions. (Alford et al, 2016).

The Actor Simulator (ActorSim) platform implements a goal lifecycle and Goal-Task Network logic. (Roberts et al, 2016). It complements existing open source planning systems with a uniform implementation of goal reasoning, also providing links to simulators that can replicate agent interaction within dynamic environments. Forge API is used to connect Minecraft with this platform (ActorSim Connector), providing methods for code manipulation.

Examined data revealed that effective choices can be achieved at the Goal-Task Network level by learning from examples which show a more efficient behaviour, including actions which complete tasks in less time or execute with minor errors. (Bonanno et al, 2016). An expert can teach the system a subgoal selection policy (Typically hand-coded), where it examines detailed state information to select the best choice. A subgoal selected by the expert trace is used to train a Convolutional Neural Network (CNN), a Deep Learning architecture which processes images for problem classification. The ActorSim Connector produces a subgoal given state information of Minecraft, given observable blocks surrounding the character. Initially both imagery and the corresponding subgoal are fed to the network in training. The following diagram illustrates this approach:



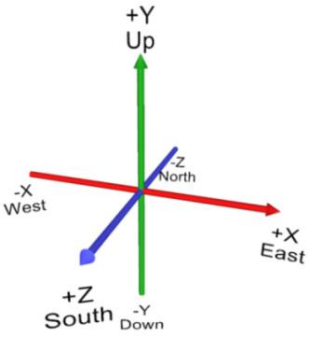
The system should then predict the respective subgoal given imagery alone. This shows a Goal-Reasoning process that can incorporate increasingly complicated goal-task networks and learned experiences about when being applied. Eventually the system will learn an effective policy to recognise the present scenario, and select the best actions in the correct order to reach specific goals.

Chapter 3 - Research Methodology

This study aims to investigate agent behaviour, using pathfinding techniques to find solutions in a decision tree setup, also compare different traversal algorithms for the agent's decisions to select the next action. The first step was to convert tasks into automated actions that the player could perform, turning them into executable tree nodes. Following was the implementation of algorithms, including tree generation and various tree traversal methods. Each tree traversal algorithm was compared to other alternatives, being evaluated according to the realism of the agent behaviour in the different scenarios.

3.1 Prototype

Minecraft was chosen as the foundation for the prototype due to the familiarity the author has with its modding code base and for being a sandbox type game, allowing for free form experimentation. The game consists of a virtual world which is randomly generated, players can wander freely and create their own experiences by using their own creativity. The game is presented in a 3D environment, consisting of X - Y - Z coordinates which forms a cube space. Terrain objects take the shape of a block with equal dimensions, each having different texture properties and physics. The game uses fractional coordinates which allow the character to move freely without snapping to a block's centre, hence can utilise the full size of the block for micro-movement. The cubes themselves can be used to form a grid map for navigation.



3.2 API and Workspace Environment

Java was used as the main coding language for the prototype, choosing Eclipse as the IDE for framework support and compatibility, it also provided powerful debugging features allowing for breakpoints to respond while the game instance was running. Eclipse was required to run Minecraft Forge API and allow game modding, Forge MDK version 1.8.9 was used for this environment. After extracting the mod development kit, a command window was issued with the following lines to install required dependencies and associate Forge with Eclipse's workspace environment.

```
gradlew setupDecompWorkspace --refresh-dependencies
gradlew eclipse
```

3.3 Framework

Knowledge on how to code an AI agent derived from available Minecraft mods. Minebot v0.4 mod[1] was chosen as the framework for the prototype, it includes the essential tools to get started and make improvements on existing work. The available code did not implement an 'intelligent' agent since a command input strategy was used, only allowing the bot to follow given steps from the user. It already implements a base AI which interacts with the game, such as pathfinding and action strategies which allows the agent to interact with objects. Consequently, it was deemed sufficient to advance with the project, to adapt decision tree logic.

[1] Mod available online at: https://github.com/michaelzangl/minebot/

3.4 AI Agent

Since an unmodified game character is normally controlled by a player in Minecraft, the approach for bot automation was required to give the character an AI nature, meaning that the character is controlled by the program source code rather than the user's input. This provided a virtual player, allowing for AI behaviour to be tested.

Actions were coded in the form of strategies, each having a Task component. The term 'Strategy' is used since the agent is instructed on how to tackle a problem with a series of defined steps, coded at Minecraft source level. If the action requires the player to interact with an object (GUI related, including mouse and keyboard input), i.e. Open chest to retrieve item or place material on crafting table, then a Task object is attached with a set of instructions for the agent to follow. Alternatively, if the action requires the player to search an object in the surrounding area, then a Pathfinder component is added which calculates the distance and path for same-type objects, the smaller the rating, the closer the object. Rules for different tiles were defined under a related BlockSet, i.e. blocks that can be destroyed, are walkable or harmful for the player.

A strategy stack was used to queue strategies in order of priority. With every game tick, the next best strategy is selected for activation based on current state. The strategy with the highest priority wanting to take over obtains control. Strategies with high priority are constantly queried for wanting to take over, based on given conditions returning to 'true' or the signalled strategy is in a good state for other strategies to take over. Strategies can also give control to other strategies in the stack by returning different values. The following table shows values for available states:

TICK_AGAIN	The strategy has more work to do. It did not do any actions that influenced the bot state. If no higher priority strategies have work, this strategy is just called again for the next tick.
TICK_HANDLED	The bot was controlled for this tick.
NO_MORE_WORK	The current strategy has no more work to do, hence releases control.
ABORT	The whole stack should exit

The 'AIHelper' class contains several utility methods which help with obtaining game data for the currently generated world, including player's state, blocks' position and item filters for checking bag inventory (e.g. Check if the player is holding a specific item). The 'Utility' class was added by the author as an extension to the existing helper methods.

3.5 Action Planning

The purpose of this study includes exploration of methods where the AI agent acts on its own, making smart decisions for itself and evaluate what the next best action shall be, based on given world state and goal data. A similar approach to goal oriented action planning (GOAP), proposed by Jeff Orkin, was chosen for implementation since it was deemed viable to the current setup. (Orkin, 2014).

For the AI agent to be considered ‘intelligent’, resulting behaviour should not be based simply on ‘If’ conditions. Consequently, an action pool was built to select the best sequence of actions for designated scenarios, hence decision making is not hardcoded but merely planned in real-time. Actions were coded separately, being independent in nature. This allowed for new actions to be added to the pool without affecting other actions. All actions were set to point in one direction, in the direction of achieving a defined goal. The structure used allows for actions to contribute towards multiple goals, where deemed necessary.

3.6 Action Setup

An abstract 'Action' class was added to follow a common structure when adding new actions. Each action has a set of preconditions and effects attached, these sets are stored in a separate HashMap. Any action with preconditions requires the agent to be in the mentioned state to run. Effects define the state the agent will be in, after the respective action has been successfully handled. This shows that an action with the

required 'Effect' can run beforehand to meet the requirements for the current action's 'Precondition', hence allowing it to run.

```
public ShearSheep(){
    addPrecondition("hasShears",true);
    addEffect("hasWool", true);
    addEffect("hasFuel", true);
    setWorldPosition();
    this.fixedCost = 14 - 1;
    enable = true;
}
```

This code snippet shows that the 'ShearSheep' action requires a state where the agent has a 'Shears' tool in hand. If the action is handled, it will produce a new state where the agent now has acquired 'Wool' material or simply 'Fuel'. Additionally, each action is defined with a fixed cost, the lower the cost, the higher the priority for the respective action to be chosen over others that require more work or material. The fixed cost value is only used to plainly rate the action's work, hence consistency can be maintained for similar actions. The efficiency of fuel can also be calculated. i.e. Some material can burn longer than other, hence rendering it more valuable and thus resulting in a lower cost. Using the same reasoning for food, hunger diminishes more rapidly when consuming larger meals.

Each action has a target objective on the world map on which it needs to perform the action itself. Objectives can range from entities to interactive objects. The location is saved in the form of X - Y - Z coordinates as a 'BlockPos' object so it can be represented later by a node, and used for planning in A-Star search. The following method obtains the closest entity limited by the given radius (15) and uses its current position for the action's world position.

```
@Override
public void setWorldPosition() {
    final Predicate<Entity> sheepSelector = new SheepSelector();

    //get closest entity to player
    Entity foundEntity = getClosestEntity(15, sheepSelector);

    if(foundEntity != null)
        this.worldPosition = foundEntity.getPosition();
}
```

A variable cost attribute was added to further calculate the work required when handling the selected action. The following method is generalised for all actions, the difference between the actions' world position is projected as the variable cost.

```
public void calcVariableCost(Action a , Action b) {

    int cost = this.variableCost;

    cost = getDistanceBetweenNodes(a, b);

    this.variableCost = cost;
}


public int getDistanceBetweenNodes(Action a, Action b)
{
    int distance = 999; //default to high variable cost

    BlockPos positionA = a.getWorldPosition();
    BlockPos positionB = b.getWorldPosition();

    if((positionA != null) && (positionB != null))
    distance = (int) Math.round(Math.sqrt
            (positionA.distanceSq(positionB.getX(), positionB.getY(), positionB.getZ())));
    //get distance between points

    return distance;
}
```

The tides can turn when calculating the total cost (fixed cost + variable cost), although an action may have a low fixed cost, the variable cost can render it inefficient if the value is high. The above method returns a rounded square root value as the distance. 'Math.sqrt' was used to rate closer objects more accurately, the value is more sensitive at small distances and will increase rapidly; however, will fall over long distances.

```
@Override
public boolean checkRunnable() {

    if(!enable)
        return false;

    final Predicate<Entity> sheepSelector = new SheepSelector();

    //search for sheep in area
    List<Entity> foundEntities = getEntities(15, sheepSelector);

    if (!foundEntities.isEmpty())
        return true;
    else
        return false;
}
```

A procedural precondition check method was added to query the world state. If there are no sheep entities in proximity the list will result to 'empty', hence shearing sheep will not be possible. A ‘false’ boolean value is returned, rendering the action unusable for planning. Other actions can run different types of scans, checking for interactive objects rather than entities, mineable ore or other resources.

Further to this, the respective AI strategy component was added to each action, instructing the agent what work is involved to complete the action itself. An action can be composes of multiple strategies which are added to a List, ordered by the sequence of execution. The following method returns the necessary strategy list, the 'ShearSheep' action only takes one strategy in this case:

```
public List<AIStrategy> strategy()
{
    List<AIStrategy> list = new ArrayList<AIStrategy>();

    list.add(new ShearStrategy(null));

    return list;
}
```

Finally, all available actions were loaded in a repository HashSet for easy retrieval during planning, the loading process is done by the 'Agent' class. This collection of actions is considered as the action pool. It provides the agent with a library to construct a solution towards the goal by picking a sequence of actions from this defined pool.

3.7 Action Pool

Each action includes an 'Effect' state that the agent will succeed to after an action has been performed. The 'Precondition' state requires the agent to be in the mentioned state before being able to execute that action. All available actions that the agent can handle are listed in the following table, final generation of results will be based on these actions:

Table 3.1 - AI Agent Actions (Action pool)

Action No.	Class	Fixed Cost	Effect	Precondition
1	ChopWood	12 (14 - 2)	hasFuel - true	hasAxe - true
2	CookMeat	6 (10 - 4)	hasFood - true	hasMeat - true
3	CraftAxe	7	hasAxe - true	hasAxeMats - true
4	CraftBread	5 (7 - 2)	hasFood - true	hasWheat - true
5	CraftPickaxe (M)*	7	hasPickaxe - true	hasPickaxeMats - true
6	CraftShears	7	hasShears - true	hasShearsMats - true

7	CraftSword	7	hasSword - true	hasSwordMats - true
8	GatherRawMeat	14	hasMeat - true	hasSword - true
9	MineCoalOre	10 (14 - 4)	hasFuel - true	hasPickaxe - true
10	MineIronOre	14	hasIronOre - true	hasPickaxe - true
11	ShearSheep	13 (14 -1)	hasWool - true hasFuel - true	hasShears - true
12	SmeltShearsMats	10	hasShearsMats - true	hasIronOre - true
13	UnstoreApple	4 (5 - 1)	hasFood - true	
14	UnstoreAxe	5	hasAxe - true	
15	UnstoreAxeMats	5	hasAxeMats - true	
16	UnstoreCoal	1 (5 - 4)	hasFuel - true	
17	UnstoreIronOre	5	hasIronOre - true	
18	UnstorePickaxe (M)*	5	hasPickaxe - true	
19	UnstorePickaxeMats (M)*	5	hasPickaxeMats - true	
20	UnstoreRawMeat	5	hasMeat - true	
21	UnstoreShears	5	hasShears - true	
22	UnstoreShearsMats	5	hasShearsMats - true	
23	UnstoreSword	5	hasSword - true	
24	UnstoreSwordMats	5	hasSwordMats - true	
25	UnstoreWheat	5	hasWheat - true	
26	UnstoreWood	3 (5 - 2)	hasFuel - true	
27	UnstoreWool	4 (5 - 1)	hasWool - true hasFuel - true	

(M)* - These actions are marked for multiuse, hence they are not cleared from the action pool during tree generation and can be used by multiple branches.

These rules were followed when assigning action fixed cost:

- Actions which instruct the agent to 'unstore' an item from a chest are assigned a cost of 5.
- Actions which instruct the agent to 'craft' an item are assigned a cost of 7.
- Actions which instruct the agent to use a furnace are assigned a cost of 10.
- Actions which instruct the agent to perform open world chores such as 'mine ore', 'chop wood', shear sheep' and 'attack entity' are assigned a cost of 14.

Various fixed costs was recalculated to par with Fuel and Food efficiency.

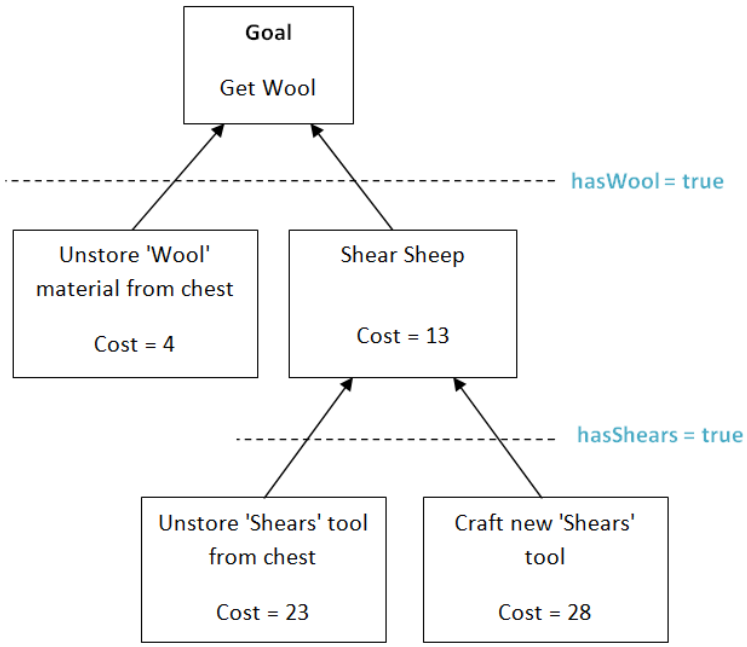
- Actions which contribute towards getting 'coal' are rated with a cost of -4.
- Actions which contribute towards getting 'wood' are rated with a cost of -2.
- Actions which contribute towards getting 'wool' are rated with a cost of -1.
- Actions which contribute towards getting 'meat' are rated with a cost of -4.
- Actions which contribute towards getting 'bread' are rated with a cost of -2.
- Actions which contribute towards getting 'fruit' are rated with a cost of -1.

The use of costs is mainly intended for A-Star search, fixed costs were assigned strategically to favour actions with less effort over those with far more work, at equal variable cost.

3.8 Decision Tree

The first technique used was a bottom-up approach, starting from the current world state and climbing up towards the goal state. The branch is expanded from a leaf Node, having a stepping sequence of parent nodes. This technique however was not logically practical for the current setting, a tree which can be expanded from a goal state was found more sensible by the author. Having the root Node represent the goal state allows for easier visual relationship between trees.

Actions were given a Node nature so further tree information can be stored. Each Node contains a reference to its parent node and a List containing references to its children nodes. The following diagram shows a visual representation of a partially generated tree as an example:



The tree is populated from top to bottom starting from the root Node which is considered as the ‘Goal’, available actions are branched out accordingly. The desired goal is normally defined in-game, entering a preset command for the AI agent to follow. If the command </minebot getwool> is issued within the chat window, a <hasWool, true> goal state is attached to the agent, expecting the bot to process actions until the set state is reached. All executable goals were registered in the 'AIChatController' class of the framework.

The following logic was followed to build the tree:

1. Goal defined as root Node, initiated with a ‘null’ parent reference since it’s at start position.
2. Iterate through available actions to build upper branching level. Actions with similar Effect state as the Goal state are declared as the root’s children.
3. Clear previously declared actions from pool to eliminate duplicate selection.
4. Use latest children nodes as the new build level.
5. Iterate through available actions, link the selected child node's Precondition state to the action Effect state. If similar values are found, the current action is declared as the child of child node.
6. The action's variable cost is also calculated with previous step.

7. Clear previously declared actions from pool to eliminate duplicate selection, unless it is marked as 'Multiuse' (Actions which are used in multiple branches).
8. Recursively call method, repeat steps from 4 to 7 until Action pool is empty, or no actions were solved from previous recursion.
9. Return root Node when finished.

Each node has a running cost property which is calculated on tree population. Costs are dynamic in nature and can change as reflected by the action's variable cost (E.g. If an object is further away from the second object, as represented by their world position, the cost increases).

```
public int calculateCost(Action action, Node parent)
{
    int cost = 0;

    if(action!= null)
        //cost = action.fixedCost;
        cost = action.calculateTotalCost();  //fixed cost + variable cost

    if(parent != null)
        cost += parent.getCost();

    return cost;
}
```

The variable cost is added to the fixed cost of the respective action, projecting the total cost. Child nodes carry their parents' cost to the end leaf node. The cost attribute is mainly used for evaluation by A-Star search which will be explained shortly. The main purpose of the tree is to make it easier for the agent to find a solution towards the goal. The search was tested with various tree traversal algorithms, explained in the next section.

3.9 Traversal Algorithms

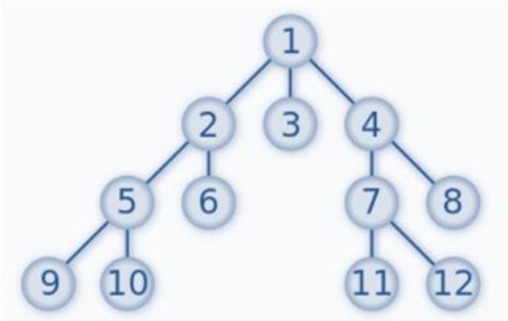
A sequence of actions towards solving a goal can be searched from the tree by visiting nodes in a selected order, defined by the algorithm. Various algorithms used in the tree traversal are discussed below.

3.9.1 A-Star Search

A-Star search uses a cost system to evaluate which branch to expand first, lower cost nodes are given more priority. An 'open' HashSet was used to queue nodes for evaluation and a 'closed' HashSet to store explored nodes. Initially the root node produces its children nodes within the open list, the lowest cost node will then be selected and added to the closed list, while removing it from the open list. If the selected node has children nodes, they are added to the open list. The above work is repeated until the open list results to empty, having no more actions to evaluate.

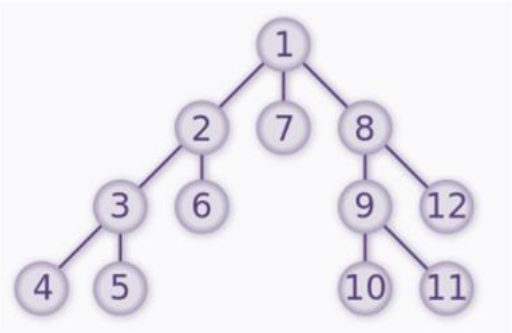
3.9.2 Breadth First Search

Breadth First Search (BFS) traverses the Tree level by level starting from the root Node. (Larmore, n.d.). The root Node is pushed in a 'Queue' and repeats the following mentioned steps until the queue is empty. The currently selected node is removed from the queue, if unvisited children nodes are found, they are marked as visited and are inserted in the queue. The following image shows an example how the tree is traversed for this algorithm:



3.9.3 Depth First Search

Depth First Search (DFS) traverses the Tree in a downward motion, trying to advance further from the root Node where the search starts. (Larmore, n.d.). The root Node is pushed onto a 'Stack' and repeats the following mentioned steps until the stack is empty, using an iterative technique. The currently selected node is peeked (Look at the object at the top of the stack without removing it). If unvisited children nodes are found, the first child node found is selected and marked as visited, then it is pushed onto the top of the stack. If the node does not have any unvisited child nodes, it is popped from the stack (The object at the top of the stack is removed). The following image shows an example how the tree is traversed for this algorithm:



For all search techniques used, actions which fail the 'checkRunnable' method will not be added to the list for evaluation, consequently sub actions will not be accessed and thus the search is narrowed.

World state data is defined in the 'Agent' class, where every object state is added to a 'HashMap'. This defines the initial state the bot will be in, including tools and materials that are held in the inventory. Some of the definitions are defined by the scanner class 'StateScanner', which scans the inventory automatically, returning value 'true' if item is found. Other definitions are hardcoded due to API support issues. The following code snippet shows world state data related to 'Get Wool' actions.

```
//WOOL
this.worldState.put("hasShears", ss.hasShears());
this.worldState.put("hasShearsMats", false); //*filter is not supported - cannot scan for iron ingot*
this.worldState.put("hasIronOre", ss.hasIronOre());
```

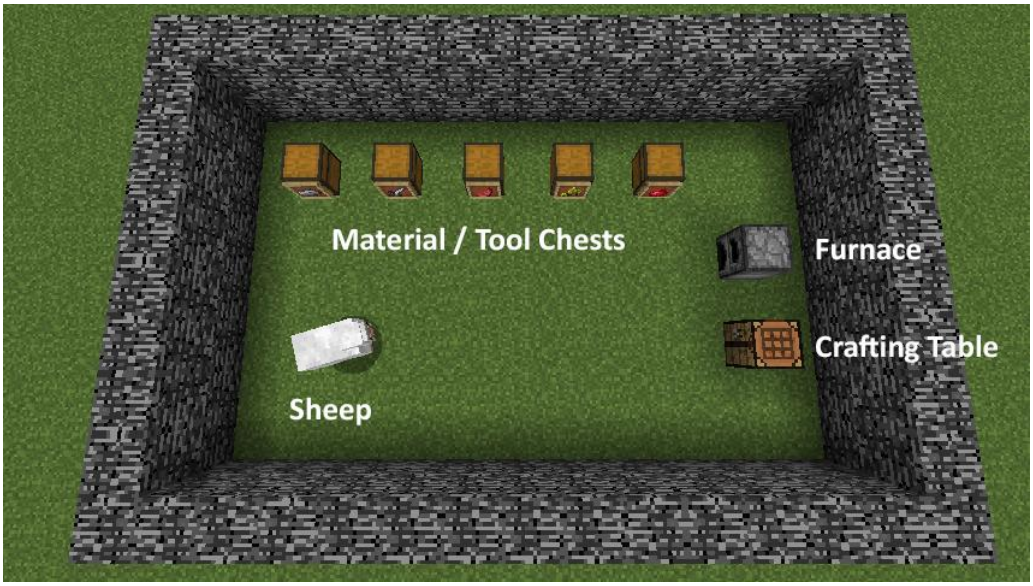
To find a solution, world-state data is supplied along with the root Node. The tree is traversed, using a selected search algorithm, until world-state data satisfies the conditional state values of a respective Node, or the Node is simply valid to run, hence resulting the solution node.

Finally the solution node is passed to the 'Planner' class so a sequence of actions towards the goal can be built. Each node's parent is used to climb the Tree, adding proceeding nodes to a 'List' until the parent node results to value 'null', meaning that the root node has been reached and a solution has been formed. Further to this, the actions are sorted by order of execution into a 'Queue', then the 'GoalStack' class passes the sequence to the strategy stack for the agent to follow. When the game is launched from the Eclipse environment [2], after a command is issued and a successful solution towards the goal has been found, the work projected by the AI will be displayed in the top-right corner. The following screenshot shows an example for the 'GetWool" command:

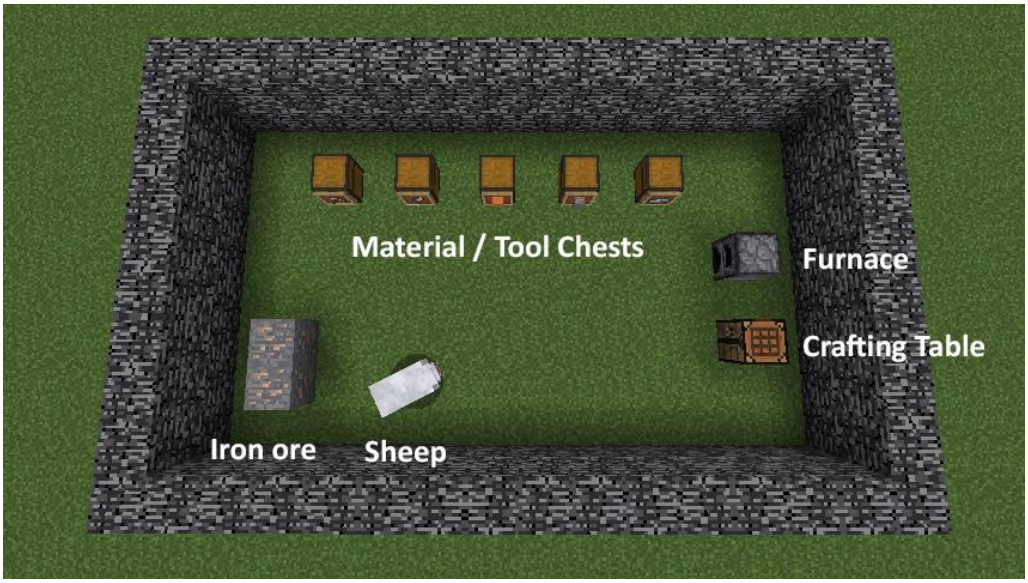
[2] Project available online at: https://github.com/marloncalleja/MC-Experiment-AI

3.10 Scenario

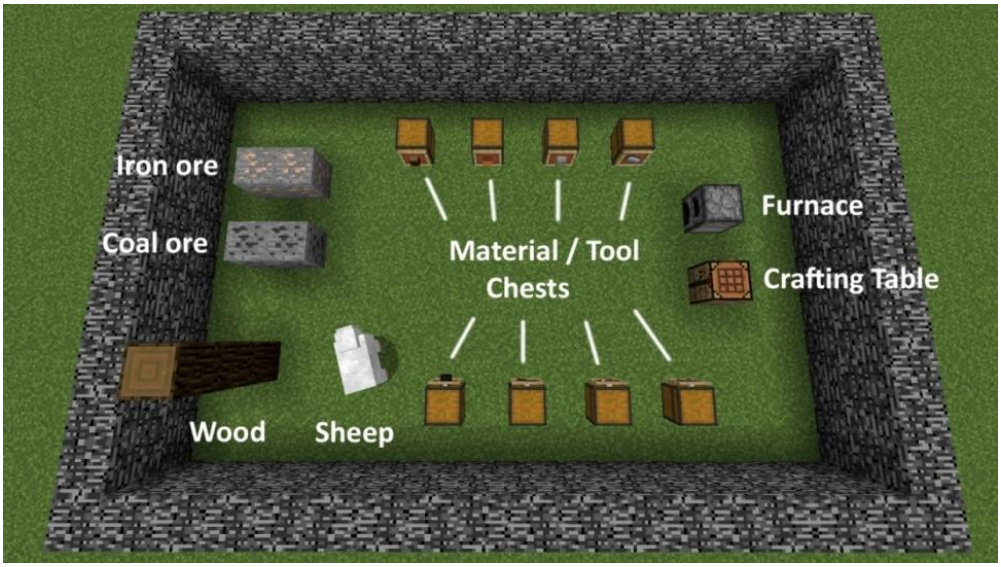
A new Minecraft world was generated with a flat surface so the environment can be simplified for the scenario. The following setting maps were built manually to test the mentioned methods applied for the prototype:



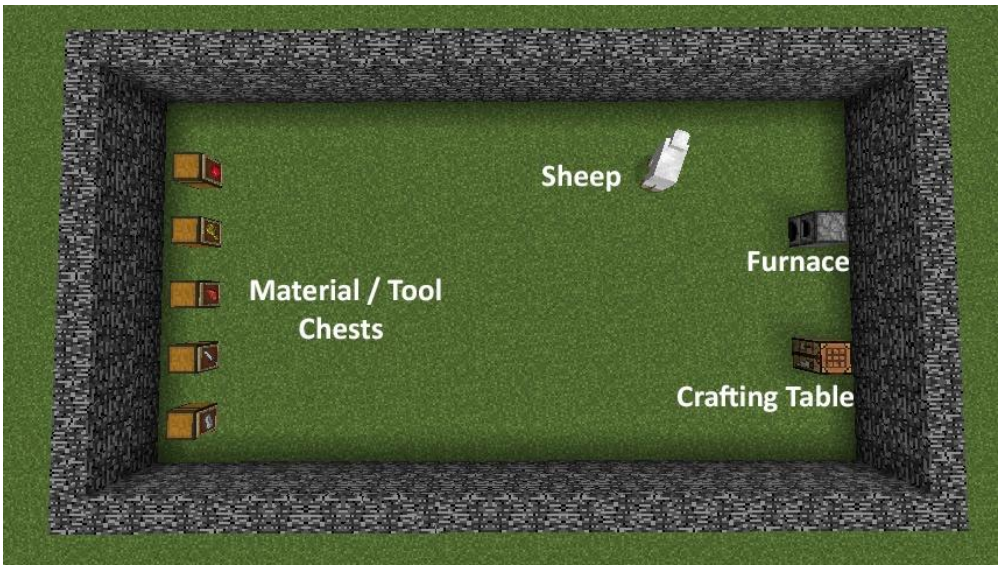
Test Map No.1 (GetFood) - Chests: Material, Sword, Raw meat, Wheat, Apple (From left to right). This map setting was built to test actions revolving around getting food.



Test Map No.2 (GetWool) - Chests: Pickaxe, Shears, Wool, Iron ore, Material (From left to right). This map setting was built to test actions revolving around getting wool.

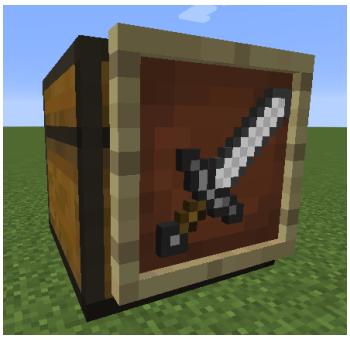


Test Map No.3 (Get Fuel) - Chests: Wood, Wool, Iron ore, Material, Coal, Axe, Shears, Pickaxe (Starting from top-left, continuing on bottom-left). This map setting was built to test actions revolving around getting fuel, it provides an expansion from the previous setting since getting wool also contributes towards getting fuel.



Test Map No.4 (Get Food) - Chests: Apple, Wheat, Raw meat, Sword, Material (From top to bottom). This map setting was built to test actions revolving around getting food, it provides the same setting from Test Map No.1 however objects are more dispersed and their positions has been changed. This was built mainly to investigate change in variable cost.

In-game chest objects are identified with an item frame, the respective material or tool icon was attached to the frame on the designated chest so the AI can make a distinction. The following image shows a chest which stores a 'Sword' item.



The default Minecraft item ID was used to identify different objects when scanning for blocks on the map or items inside the player's inventory. Map settings were also built with increasing tree complexity so traversal algorithms can be tested in different scenarios.

Chapter 4 - Findings

4.1 Decision Tree Population

Tree generation has been sampled across 3 different map settings, each corresponding to 3 different goals. The map settings used are described in the previous chapter. The running-time for the algorithm was measured using System.nanoTime() and the time taken was averaged over 10k runs of the same algorithm to reduce margin of error. Actions used were processed before the test to pre-load classes and diminish measurement variations. Each test took less than 3 minutes to complete, system specifications can be found in the Appendix section. The following table shows a portion of the resulting values in nanoseconds generated by the program, results were printed for every 500 iterations, mean time was calculated across 10k iterations.

Table 4.1 - Decision Tree Population

	Tree No.1 *(Test Map No.1 - Get Food)*		**Tree No.2** *(Test Map No.2 - Get Wool)*		**Tree No.3** **(***Test Map No.3 - Get Fuel)*	
Iteration No.	**No Cost**	**With Cost**	**No Cost**	**With Cost**	**No Cost**	**With Cost**
1	127685	143986	151834	163304	185944	220054
500	35015	27167	29581	30488	35317	44071
1000	29582	29280	26563	27770	42260	38940
1500	23243	23847	26865	27469	32298	34412
2000	23544	23847	29280	31092	32299	38336
2500	41354	27167	27168	28676	31997	33506
3000	23846	23847	32299	27167	31997	33808
3500	41355	22941	32299	27469	33204	38034
4000	23243	24149	26563	27469	35318	33808
4500	25658	23545	26563	51617	32903	33808
5000	24450	24148	31695	32902	32601	34110
5500	23243	25658	26262	26865	33205	34109
6000	24148	25658	31091	29581	32299	33506
6500	23847	24148	28073	33506	33808	35317
7000	28073	23243	32601	32903	33204	39543
7500	22338	28375	30185	28374	32902	33506
8000	25054	24148	31695	31393	31997	33204
8500	23242	23847	26563	26865	36223	38940
9000	23847	28374	26564	27167	38939	32600
9500	25960	29280	34110	26865	37733	32902
10000	26865	23545	26564	27168	39241	32601
Mean Time	**26602.193**	**27004.4164**	**29361.8022**	**30713.6289**	**35711.7152**	**36184.8751**

The gathered data shows that tree generation time drops significantly with succeeding iterations. This is only resulted due to classes being cached into memory after multiple algorithm execution, hence process time becomes faster. The value that matters mostly for this study, is the average generation time (Mean Time) for the algorithm to populate the respective tree based on a given goal. Decision trees generated from each map setting were printed for further analysis. A visual representation can be found in the Appendix section. (Test maps used are mentioned in the previous chapter).

Tree No.1 *(Test Map No.1 - Get Food goal)*

```
CookMeat (6 + 0)     Running Cost: 6.0
UnstoreApple (4 + 0)     Running Cost: 4.0
CraftBread (5 + 0)     Running Cost: 5.0
-----------------------------------------------------------
CookMeat <- GatherRawMeat (14 + 10)    Running Cost: 30.0
CookMeat <- UnstoreRawMeat (5 + 6)    Running Cost: 17.0
CraftBread <- UnstoreWheat (5 + 6)    Running Cost: 16.0
-----------------------------------------------------------
GatherRawMeat <- CraftSword (7 + 10)    Running Cost: 47.0
GatherRawMeat <- UnstoreSword (5 + 5)    Running Cost: 40.0
-----------------------------------------------------------
CraftSword <- UnstoreSwordMats (5 + 11)    Running Cost: 63.0
-----------------------------------------------------------
```

Tree No.2 *(Test Map No.2 - Get Wool goal)*

```
ShearSheep (13 + 0)     Running Cost: 13.0
UnstoreWool (4 + 0)     Running Cost: 4.0
-----------------------------------------------------------
ShearSheep <- CraftShears (7 + 8)    Running Cost: 28.0
ShearSheep <- UnstoreShears (5 + 4)    Running Cost: 22.0
-----------------------------------------------------------
CraftShears <- SmeltShearsMats (10 + 2)    Running Cost: 40.0
CraftShears <- UnstoreShearsMats (5 + 4)    Running Cost: 37.0
-----------------------------------------------------------
SmeltShearsMats <- MineIronOre (14 + 14)    Running Cost: 68.0
SmeltShearsMats <- UnstoreIronOre (5 + 4)    Running Cost: 49.0
-----------------------------------------------------------
MineIronOre <- CraftPickaxe (7 + 12)    Running Cost: 87.0
MineIronOre <- UnstorePickaxe (5 + 6)    Running Cost: 79.0
-----------------------------------------------------------
CraftPickaxe <- UnstorePickaxeMats (5 + 4)    Running Cost: 96.0
-----------------------------------------------------------
```

Tree No.3 *(Test Map No.3 - Get Fuel goal)*

```
ShearSheep (13 + 0)     Running Cost: 13.0
UnstoreWood (3 + 0)     Running Cost: 3.0
UnstoreWool (4 + 0)     Running Cost: 4.0
UnstoreCoal (1 + 0)     Running Cost: 1.0
MineCoalOre (10 + 0)     Running Cost: 10.0
ChopWood (12 + 0)     Running Cost: 12.0
------------------------------------------------------------
ShearSheep <- UnstoreShears (5 + 7)    Running Cost: 25.0
ShearSheep <- CraftShears (7 + 11)    Running Cost: 31.0
MineCoalOre <- UnstorePickaxe (5 + 24)    Running Cost: 27.0
MineCoalOre <- CraftPickaxe (7 + 28)    Running Cost: 31.0
ChopWood <- CraftAxe (7 + 14)    Running Cost: 33.0
ChopWood <- UnstoreAxe (5 + 8)    Running Cost: 25.0
------------------------------------------------------------
CraftShears <- SmeltShearsMats (10 + 2)    Running Cost: 43.0
CraftShears <- UnstoreShearsMats (5 + 4)    Running Cost: 40.0
CraftPickaxe <- UnstorePickaxeMats (5 + 8)    Running Cost: 40.0
CraftAxe <- UnstoreAxeMats (5 + 4)    Running Cost: 42.0
------------------------------------------------------------
SmeltShearsMats <- UnstoreIronOre (5 + 4)    Running Cost: 52.0
SmeltShearsMats <- MineIronOre (14 + 15)    Running Cost: 72.0
------------------------------------------------------------
MineIronOre <- UnstorePickaxe (5 + 24)    Running Cost: 101.0
MineIronOre <- CraftPickaxe (7 + 28)    Running Cost: 107.0
------------------------------------------------------------
CraftPickaxe <- UnstorePickaxeMats (5 + 8)    Running Cost: 120.0
------------------------------------------------------------
```

As tree population was done using a top-down approach, the root node was taken as the start node for iterative expansion of children nodes, printing the tree level by level until the leaf of each branch is reached. Each level is divided by a separator and nodes are linked to their parent using a directed arrow sign. Every action cost is also listed (total cost = fixed cost + variable cost) along with the Node running cost which sums up the total cost to the parent's cost. Deeper tree levels constitute higher running cost actions since they are the farthest nodes from the goal.

Significant tree data is summarised in the following table:

Table 4.2 - Tree Summary

	Tree No.1		Tree No.2		Tree No.3	
	No Cost	**With Cost**	**No Cost**	**With Cost**	**No Cost**	**With Cost**
Generation mean time	26602.193	27004.4164	29361.8022	30713.6289	35711.7152	36184.8751
Branches from root node	3	3	2	2	6	6
Maximum depth (Levels)	4	4	6	6	6	6
Total nodes	9	9	11	11	21	21

Each map setting has increasing tree complexity, meaning that more actions are evaluated. A moderate increase in generation time was noted in trees which have a higher count in nodes, branches and levels. Minimal difference was observed in generation time when calculating nodes' running cost, compared to computation with no cost value. As already discussed, the use of cost is mainly intended for A-Star search for the expansion of more promising paths, this will be examined shortly within this chapter.

4.2 Comparison of similar map settings

Maps which include similar environment/objects were used for data comparison when running the same actions in a different setting. The following trees were generated for the 'Get Wool' goal, using 2 different map settings. (Test maps used are mentioned in the previous chapter).

Tree No.2 *(Test Map No.2 - Get Wool goal) - Print results taken from previous section*

```
ShearSheep (13 + 0)     Running Cost: 13.0
UnstoreWool (4 + 0)     Running Cost: 4.0
-----------------------------------------------------------
ShearSheep <- CraftShears (7 + 8)    Running Cost: 28.0
ShearSheep <- UnstoreShears (5 + 4)    Running Cost: 22.0
-----------------------------------------------------------
CraftShears <- SmeltShearsMats (10 + 2)    Running Cost: 40.0
CraftShears <- UnstoreShearsMats (5 + 4)    Running Cost: 37.0
-----------------------------------------------------------
SmeltShearsMats <- MineIronOre (14 + 14)    Running Cost: 68.0
SmeltShearsMats <- UnstoreIronOre (5 + 4)    Running Cost: 49.0
-----------------------------------------------------------
MineIronOre <- CraftPickaxe (7 + 12)    Running Cost: 87.0
MineIronOre <- UnstorePickaxe (5 + 6)    Running Cost: 79.0
-----------------------------------------------------------
CraftPickaxe <- UnstorePickaxeMats (5 + 4)    Running Cost: 96.0
-----------------------------------------------------------
```

Tree No.4 *(Test Map No.3 - Get Wool goal)*

```
UnstoreWool (4 + 0)     Running Cost: 4.0
ShearSheep (13 + 0)     Running Cost: 13.0
-----------------------------------------------------------
ShearSheep <- CraftShears (7 + 11)    Running Cost: 31.0
ShearSheep <- UnstoreShears (5 + 7)    Running Cost: 25.0
-----------------------------------------------------------
CraftShears <- UnstoreShearsMats (5 + 4)    Running Cost: 40.0
CraftShears <- SmeltShearsMats (10 + 2)    Running Cost: 43.0
-----------------------------------------------------------
SmeltShearsMats <- UnstoreIronOre (5 + 4)    Running Cost: 52.0
SmeltShearsMats <- MineIronOre (14 + 15)    Running Cost: 72.0
-----------------------------------------------------------
MineIronOre <- CraftPickaxe (7 + 14)    Running Cost: 93.0
MineIronOre <- UnstorePickaxe (5 + 12)    Running Cost: 89.0
-----------------------------------------------------------
CraftPickaxe <- UnstorePickaxeMats (5 + 4)    Running Cost: 102.0
-----------------------------------------------------------
```

The first map environment is limited to the actions being used, only including objects which are relevant to the 'Get Wool' goal. The second map is an expansion of the first map, it includes additional objects which are intended for the 'Get Fuel' goal, hence the 'Get Wool' goal can still be processed. This test was performed to examine whether alteration in environment may affect tree population, due to additional objects being scanned. Affected by change in object placement, a minor difference between action variable cost was noted. As a result, the same action rating was calculated, hence no functional difference between the trees exist. To assert completely that no major operation is affected, the following table shows samples taken during tree generation for both map settings, for the same 'Get Wool' goal. By examining the mean time for tree generation, one can say that minimal difference is present, thus it can be concluded that operation is not altered.

Table 4.3 - Tree Population of Similar Maps

	Tree No.2		Tree No.4	
Iteration No.	**No Cost**	**With Cost**	**No Cost**	**With Cost**
1	151834	163304	160284	163303
500	29581	30488	35015	30488
1000	26563	27770	33204	27167
1500	26865	27469	32299	27469
2000	29280	31092	28676	34110
2500	27168	28676	26865	34109
3000	32299	27167	32902	33204
3500	32299	27469	28978	25960
4000	26563	27469	33204	32298
4500	26563	51617	27771	26865
5000	31695	32902	29280	27167
5500	26262	26865	27167	26262
6000	31091	29581	27469	27167
6500	28073	33506	26865	30789
7000	32601	32903	28677	33204
7500	30185	28374	27166	27166
8000	31695	31393	33505	26261
8500	26563	26865	32298	35317
9000	26564	27167	28072	27468
9500	34110	26865	32600	33506
10000	26564	27168	31997	26865
Mean Time	**29361.8022**	**30713.6289**	**29958.3648**	**30048.9192**

The results for Tree No.2 are the same values taken from Table 4.1.

The following trees were generated for the 'Get Food' goal, using 2 different map settings (Test maps used are mentioned in the previous chapter).

Tree No.1 *(Test Map No.1 - Get Food goal) - Print results taken from previous section*

```
CookMeat (6 + 0)     Running Cost: 6.0
UnstoreApple (4 + 0)     Running Cost: 4.0
CraftBread (5 + 0)     Running Cost: 5.0
-----------------------------------------------------
CookMeat <- GatherRawMeat (14 + 10)    Running Cost: 30.0
CookMeat <- UnstoreRawMeat (5 + 6)    Running Cost: 17.0
CraftBread <- UnstoreWheat (5 + 6)    Running Cost: 16.0
-----------------------------------------------------
GatherRawMeat <- CraftSword (7 + 10)    Running Cost: 47.0
GatherRawMeat <- UnstoreSword (5 + 5)    Running Cost: 40.0
-----------------------------------------------------
CraftSword <- UnstoreSwordMats (5 + 11)    Running Cost: 63.0
-----------------------------------------------------
```

Tree No.5 *(Test Map No.4 - Get Food goal)*

```
UnstoreApple (4 + 0)     Running Cost: 4.0
CookMeat (6 + 0)     Running Cost: 6.0
CraftBread (5 + 0)     Running Cost: 5.0
-----------------------------------------------------
CookMeat <- GatherRawMeat (14 + 6)    Running Cost: 26.0
CookMeat <- UnstoreRawMeat (5 + 20)    Running Cost: 31.0
CraftBread <- UnstoreWheat (5 + 20)    Running Cost: 30.0
-----------------------------------------------------
GatherRawMeat <- CraftSword (7 + 8)    Running Cost: 41.0
GatherRawMeat <- UnstoreSword (5 + 16)    Running Cost: 47.0
-----------------------------------------------------
CraftSword <- UnstoreSwordMats (5 + 20)    Running Cost: 66.0
-----------------------------------------------------
```

The second map uses a more dispersed setting for objects. As the distance between objects increased, it affected the variable cost of some actions, hence changing the objects' position alters the running cost of several nodes. Some actions resulted to be more efficient in the first map, however they were expensive in the second map, and vice-versa. The above tree data shows clearly the running cost adjustment made between the two maps. Comparing the second level nodes of both trees, the 'GatherRawMeat' action resulted to be more efficient to run in the second map than the first, as it projected a lower running cost when compared to 'UnstoreRawMeat' and 'UnstoreWheat' actions.

4.3 Tree traversal algorithms for solution finding

Multiple solutions towards a set Goal can be searched from their respective tree using a range of 3 different algorithms which were discussed in the previous chapter. Each algorithm was tested independently on every tree generated, for 3 different test maps and their respective Goal setting. The following table provides brief characteristics of the trees generated earlier, on which tests will be performed.

Tree No.1	**Tree No.2**	**Tree No.3**
Few branches	Few branches	Numerous branches
Medium level depth	High level depth	High level depth
Low to moderate node count	Moderate node count	High node count
Small sized tree overall	Medium sized tree overall	Large sized tree overall

Trees with diverse characteristics were selected on purpose to test search algorithms in different settings. World-state data was not taken into account for these tests, all object values were set to 'false' to imitate an empty inventory. Consequently, the agent will not posses any in-game items which may alter the action sequence (Solution output) towards the goal (E.g. having a 'Shears' tool inside the inventory will avoid the 'craft shears' process, jumping directly to the 'shear sheep' strategy as a solution, this was negated during the test).

Each time a successful solution node is returned, the action sequence is printed and visited tree nodes are reset to 'non-visited' so a new search can be performed. To obtain multiple search results, the start node of every action sequence found is disabled so it won't return as a solution in the following search, hence subsequent solutions will be unique. This strategy was only used to test the traversing sequence of the chosen algorithm, evaluating the order in which solution nodes are found.

Solutions are printed by order of exploration. Every action sequence (solution towards the goal) is separated by a divider, total running cost is also printed for each solution. The following solutions were found when testing the 3 mentioned traversal algorithms, on Tree No.1.

A-Star *(Solution Result Set No.1)*	**BFS** *(Solution Result Set No.2)*	**DFS** *(Solution Result Set No.3)*
Running Cost: 4.0 1: UnstoreApple --- Running Cost: 16.0 1: UnstoreWheat 2: CraftBread --- Running Cost: 17.0 1: UnstoreRawMeat 2: CookMeat --- Running Cost: 40.0 1: UnstoreSword 2: GatherRawMeat 3: CookMeat --- Running Cost: 63.0 1: UnstoreSwordMats 2: CraftSword 3: GatherRawMeat 4: CookMeat ---	Running Cost: 4.0 1: UnstoreApple --- Running Cost: 17.0 1: UnstoreRawMeat 2: CookMeat --- Running Cost: 16.0 1: UnstoreWheat 2: CraftBread --- Running Cost: 40.0 1: UnstoreSword 2: GatherRawMeat 3: CookMeat --- Running Cost: 63.0 1: UnstoreSwordMats 2: CraftSword 3: GatherRawMeat 4: CookMeat ---	Running Cost: 63.0 1: UnstoreSwordMats 2: CraftSword 3: GatherRawMeat 4: CookMeat --- Running Cost: 40.0 1: UnstoreSword 2: GatherRawMeat 3: CookMeat --- Running Cost: 17.0 1: UnstoreRawMeat 2: CookMeat --- Running Cost: 16.0 1: UnstoreWheat 2: CraftBread --- Running Cost: 4.0 1: UnstoreApple ---

Solutions may be explored in a different order when using BFS or DFS, significant data was hand-picked for evidence, aiming more towards worst-case scenarios (Based on running cost) where applicable so an argument can be made.

The following solutions were found when testing the 3 mentioned traversal algorithms, on Tree No.2.

A-Star *(Solution Result Set No.4)*	**BFS** *(Solution Result Set No.5)*	**DFS** *(Solution Result Set No.6)*
Running Cost: 4.0 1: UnstoreWool ---------- Running Cost: 22.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 37.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 49.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 79.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 96.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ----------	Running Cost: 4.0 1: UnstoreWool ---------- Running Cost: 22.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 37.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 49.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 79.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 96.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ----------	Running Cost: 96.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ---------- Running Cost: 79.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 49.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 37.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 22.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 4.0 1: UnstoreWool ----------

The following solutions were found when testing the 3 mentioned traversal algorithms, on Tree No.3.

A-Star *(Solution Result Set No.7)*	**BFS** *(Solution Result Set No.8)*	**DFS** *(Solution Result Set No.9)*
Running Cost: 1.0 1: UnstoreCoal ---------- Running Cost: 3.0 1: UnstoreWood ---------- Running Cost: 4.0 1: UnstoreWool ---------- Running Cost: 25.0 1: UnstoreAxe 2: ChopWood ---------- Running Cost: 25.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 27.0 1: UnstorePickaxe 2: MineCoalOre ---------- Running Cost: 40.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineCoalOre ---------- Running Cost: 40.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 42.0 1: UnstoreAxeMats 2: CraftAxe 3: ChopWood ---------- Running Cost: 52.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 101.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 120.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ----------	Running Cost: 4.0 1: UnstoreWool ---------- Running Cost: 1.0 1: UnstoreCoal ---------- Running Cost: 3.0 1: UnstoreWood ---------- Running Cost: 25.0 1: UnstoreAxe 2: ChopWood ---------- Running Cost: 27.0 1: UnstorePickaxe 2: MineCoalOre ---------- Running Cost: 25.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 42.0 1: UnstoreAxeMats 2: CraftAxe 3: ChopWood ---------- Running Cost: 40.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineCoalOre ---------- Running Cost: 40.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 52.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 101.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 120.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ----------	Running Cost: 120.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineIronOre 4: SmeltShearsMats 5: CraftShears 6: ShearSheep ---------- Running Cost: 101.0 1: UnstorePickaxe 2: MineIronOre 3: SmeltShearsMats 4: CraftShears 5: ShearSheep ---------- Running Cost: 52.0 1: UnstoreIronOre 2: SmeltShearsMats 3: CraftShears 4: ShearSheep ---------- Running Cost: 40.0 1: UnstoreShearsMats 2: CraftShears 3: ShearSheep ---------- Running Cost: 25.0 1: UnstoreShears 2: ShearSheep ---------- Running Cost: 3.0 1: UnstoreWood ---------- Running Cost: 40.0 1: UnstorePickaxeMats 2: CraftPickaxe 3: MineCoalOre ---------- Running Cost: 27.0 1: UnstorePickaxe 2: MineCoalOre ---------- Running Cost: 25.0 1: UnstoreAxe 2: ChopWood ---------- Running Cost: 42.0 1: UnstoreAxeMats 2: CraftAxe 3: ChopWood ---------- Running Cost: 4.0 1: UnstoreWool ---------- Running Cost: 1.0 1: UnstoreCoal ----------

4.4 Comparison of solution results

The running cost is only perceived by A-Star search, with reference to solution results, a clear picture can be obtained. The resulting trend shows an ascending order of solutions, starting from the lowest running cost action sequence. This shows an efficient selection of solutions, returning an action sequence with the least work required first.

The other 2 algorithms use a different technique to find solutions. BFS and DFS base their selection on the first 'unvisited' Node deeming valid, while following their unique search nature, and thus running cost is not considered.

BFS shows a close resemblance to A-Star search in its selection order, having only a minor deviation in efficiency in some tree structures. The sole case where it can hinder the algorithm's efficiency is when working with high disparity running costs, found on the same tree level. Hence, having the risk of selecting an expensive solution over a cheap action sequence, while both being a valid selection. The current setting however makes the algorithm efficient since a small deviation in running cost is present between nodes found on the same level.

DFS algorithm resulted with the highest chance of selecting an expensive solution, when compared to A-Star and BFS. Its search nature allows it to dive deep into the tree, selecting action sequences which require more work to complete. Depending on the structure of the tree, solutions which are cheap or moderately expensive can also be selected from time to time, however its nature is inconsistent.

4.5 Comparison of traverse time and running cost

A randomised tree generation technique was used for this experiment to gather algorithm traverse data, for three different Tree structures. Actions were given an activation factor of 0.85 while world-state objects were given an activation factor of 0.50. This means that a number is generated within the range of 0.00 to 1.00, if the number generated is above the activation factor value then the respective action or world-state object is activated. It allowed for multiple trees to provide diverse

functionality, presenting different paths when searching for solutions during this experiment, hence imitating game-play situations. The test was conducted on a total of 10k randomised tree generations, obtaining essential data for A-Star search, BFS and DFS when averaged. The same strategy used in the 'Decision Tree Population' section was applied to read measurements. A sample is given every 1k tests for observation. The following data was gathered when testing the 3 mentioned traversal algorithms, based on Tree No.1 structure (Get Food goal).

Test Result Set No.1	**A-Star**		**BFS**		**DFS**	
Test No.	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**
1	675847	4	679469	4	680375	4
1000	495339	4	498962	4	498056	4
2000	493226	4	917932	4	820132	47
3000	505905	4	505301	4	503792	4
4000	493830	4	910085	4	416858	30
5000	699392	4	461230	6	418065	6
6000	504697	4	926384	4	930007	17
7000	505300	4	413538	5	410519	5
8000	492925	4	1320301	4	901029	16
9000	824660	6	420178	6	418368	6
10000	502282	4	926384	4	425913	30
MEAN	**538399.7251**	**4.7046**	**639046.0088**	**5.2329**	**603491.4526**	**10.1740**
No solution found for 53 out of 10000 tests						

The following data was gathered when testing the 3 mentioned traversal algorithms, based on Tree No.2 structure (Get Wool goal).

Test Result Set No.2	**A-Star**		**BFS**		**DFS**	
Test No.	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**
1	518280	4	4226	13	5735	13
1000	506810	4	514658	4	1417196	49
2000	509527	4	517677	4	515865	4
3000	511941	22	517073	22	513450	22
4000	515564	4	523110	4	949928	37
5000	515261	4	520997	4	430743	28
6000	510131	22	555710	22	523412	22
7000	525524	4	535184	4	967738	37
8000	520091	4	518582	4	518582	4
9000	505905	4	514054	4	931817	37
10000	504697	4	509828	4	511035	4
MEAN	**481543.4686**	**5.8980**	**393720.8575**	**7.5746**	**412381.7177**	**12.20010**
No solution found for 230 out of 10000 tests						

The following data was gathered when testing the 3 mentioned traversal algorithms, based on Tree No.3 structure (Get Fuel goal).

Test Result Set No.3	**A-Star**		**BFS**		**DFS**	
Test No.	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**
1	474215	1	479649	3	853648	40
1000	516777	1	517683	1	491723	25
2000	513758	1	1208	12	1208	12
3000	471197	3	476932	3	476630	3
4000	477535	1	482365	4	479045	27
5000	485685	1	487799	1	385168	31
6000	485384	1	505910	4	893493	42
7000	477536	3	906	12	905	12
8000	486290	1	494440	3	489308	25
9000	482365	1	483874	4	503193	4
10000	477234	1	496553	1	508325	1
MEAN	**494504.8354**	**1.373**	**340443.6837**	**5.5689**	**383709.4265**	**11.8749**
All tests resulted to be valid (At least 1 solution has been found in every test)						

The data gathered shows the time taken in nanoseconds for the selected algorithm to traverse the tree and find a solution, also showing its respective running cost. Tests with no solutions were not accounted for in the final average measurement.

A-Star search has proven to be the most efficient algorithm when prioritising solutions by lowest running cost. Although it takes longer to find a solution in some cases, the algorithm never fails to select the cheapest action, showing a consistent nature. A-Star search takes longer by default since it requires the algorithm to compare nodes for the lower cost before exploring new child nodes. BFS has shown reasonable results for traverse time, on average returning the first valid solution in a low amount of time which can be sub-optimal in terms of cost. With the current setup DFS has shown an inconsistent nature in its selection process, solutions range from low to high cost. On a positive note, valid solution are returned in a fair amount of time.

4.6 Comparison of similar tree structure with fewer solution nodes

Multiple actions were disabled, to further test the functionality of search algorithms on the same tree structure, while having fewer solution nodes. Actions were given an activation factor of 0.30, world-state items were kept at an activation factor of 0.50. The following data was gathered when testing the 3 mentioned traversal algorithms, based on Tree No.3 structure (Get Fuel goal).

Test Result Set No.4	**A-Star**		**BFS**		**DFS**	
Test No.	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**	**Traverse Time**	**Solution Cost**
1	*N.S.*	*N.S.*	*N.S.*	*N.S.*	*N.S.*	*N.S.*
1000	474513	1	3622	13	3018	13
2000	*N.S.*	*N.S.*	*N.S.*	*N.S.*	*N.S.*	*N.S.*
3000	565069	1	481455	1	473607	1
4000	457005	4	462137	4	459420	4
5000	373392	31	373694	31	366751	31
6000	1509	10	1208	10	1509	10
7000	467268	4	476626	4	479041	4
8000	464552	4	473305	4	470287	4
9000	470890	3	3622	13	3018	13
10000	474815	1	480852	1	403879	31
MEAN	**398609.9663**	**5.1348**	**327247.4010**	**6.8460**	**338710.4505**	**8.7373**
No solution found for 1684 out of 10000 tests N.S. - No solution						

Comparing data to Test Result Set No.3 shows that traverse time values have dropped considerably for having fewer solution nodes to explore, affecting mostly A-Star search. In terms of running cost efficiency, the same ranking was maintained though values have changed.

Chapter 5 - Discussion of Results

In this section, data obtained in the previous chapter will be discussed in more detail, investigating decision tree generation and traversal algorithms results concerning AI behaviour.

5.1 Decision Tree Generation

Decision tree results have shown that generation time inflates with action pool size. Increase in AI action count will result in more iterations towards tree population since more elements will be searched. This will become problematic over a large scale implementation when using the current setting. A tree is generated each time a new goal is set for the AI agent to carry, having to iterate through every available action stored inside the HashSet and populate a new tree for that specific instance.

The tree generation algorithm is fairly complex for having a number of nested operations; required to match actions with nodes, and having to call the same method recursively until the tree is fully built. Every action compares its 'Effect' states with every node's 'Precondition' states at the present tree level, creating a new child node if a match is found. A feature was added at a later stage to improve this algorithm, using a 'dump' strategy. Unless actions are marked as 'multiuse', solved actions are dumped from the pool so they won't be evaluated in the next recursion.

A minor variation was noted in tree population time between sets which ignored cost calculation and sets which included cost. As results indicate, a similar trend was maintained for different test maps; however, tree population time is expected to increase substantially over increasing node count. Although this statement may be true, in practice trees are still limited to the number of actions they can carry with the current setup, unless a broad goal is specified. If goals are maintained at a specific level of instructions like "Get Wool", high node count expectations will not become an issue, therefore calculation of action cost will be worth the implementation.

For the experiment, a same amount of blocks were scanned as defined by the 'BlockRangeScanner' class. Results from Table 4.3 have shown that change in map objects does not impact tree generation time directly since the same values were obtained; however, map scanning duration is practically affected overall although it is not shown.

```
public class BlockRangeScanner {
    private static final int HORIZONTAL_SCAN = 15;
    private static final int VERTICAL_SCAN = 15;
    private final BlockPos center;
```

The above code snippet shows that a radius of 15 x 15 blocks has been scanned from the player's position. If a large area is defined, the state of objects greatly increases, hence having more items to evaluate. If similar objects are scanned, the closest object must be considered. As a result, scan value must be chosen wisely to mask irrelevant objects which are too far away, hence improve scanning time and reduce necessity for memory space. Since blocks can be modified during game-play, rescanning of an area is required to reflect these changes, hence a one-time scan of the whole map is not feasible.

5.2 Traversal Algorithms

5.2.1 A-Star

In the case of A-Star search, being a pathfinding algorithm, its application onto a decision tree has allowed the assignment of costs to each step in the action sequence, being performed by the AI agent. To fully compile the time taken by A-Star to compute a search, Table 4.1 was used to extract the time difference of generated trees which included cost. This measurement must be considered since it is the sole algorithm (from BFS and DFS) that utilizes cost for its node expansion decision. After adding tree generation time and traverse time from available result sets, the application of a pathfinding algorithm was still worth employing in the context of building an AI agent, even though it takes longer to compute than BFS and DFS in the majority of cases. As a result, retrieval of the best solution from the

tree is always guaranteed, obtaining an action sequence with the least number of steps required towards achieving a set goal.

A-Star search should be used in scenarios where selection of the cheapest solution is required, based on running cost efficiency. This is ideal for missions that do not require an urgent solution, maybe the player wants to craft an item in the most effective way possible, requiring the least number of steps or cheapest material. If goals are maintained at a specific level of instructions like "Get Wool", high node count expectations will not become an issue towards total computation time, for the agent to come up with a solution. A-Star search should be prioritised in the context of making strategic decisions rather than searching for quick solutions. It will not obtain best results in fast paced situations since a slower reaction time will be resulted by the AI agent, when compared to BFS and DFS, for having to calculate action cost. Turn-based games or any setting which dedicates time for the NPC to 'think' and decide about a solution, is the ideal instance to employ this algorithm. The algorithm's efficiency is hindered in real-time games involving fast paced combat.

In the situation of extremely complex decision trees, A-Star search is supposed to obtain the best results, both in traverse time and solution cost efficiency when compared to BFS and DFS. Although the tree will constitute multiple branches and deep paths (worst-case for BFS or DFS), the algorithm would still expand nodes with the most promising path, ultimately ignoring expensive nodes, hence reducing amount of nodes being traversed overall.

5.2.2 BFS

From gathered results, BFS was rated as the 2nd alternative to A-Star search. Resulted solutions are still effective in terms of running cost; however, they can turn into expensive choices if same level nodes with a high disparity cost value are populated. Since the selection rule is based on a "first valid solution at current tree level", it provides an unpredictable nature for newly generated trees since the position of nodes may vary. This randomness factor however, gives it a more human-like characteristic since it does not select the 'perfect' solution. BFS resulted in optimal traverse time since the branching factor is small for the current setting,

and solutions were more likely to be found in the lowest depth. If a broader goal is specified however, BFS will use the most memory since space requirement to search nodes grows exponentially. If it leads to this, branching should be split into *x* amount, starting the search from the left-most segment of the tree and advancing the search to the next segment, right after reaching a leaf node. This technique will reduce memory consumption during that search instance.

BFS should be used in fast decision environments, taking combat situations for example, were fast return of solutions is more of a priority than choosing the best sequence of actions. Although fast retrieval of solutions is witnessed, the resultant action sequence is always sub-optimal, hence the agent will employ good reflexes with slightly expensive decisions. In terms of game environment, the algorithm is appropriate for real-time, head-to-head combat. The enemy will pose fast reaction with a less predictable behaviour due to the algorithm's traverse nature. This algorithm should be avoided at all costs when decision trees constitute a high branching factor since it is considered as the algorithm's worst-case

5.2.3 DFS

DFS isn't favourable when prioritising solution running cost. Unless the agent is searching for a solution that requires a high amount of work intentionally, it is not the best option. The most memory it can consume during a search, is the longest possible path, hence being affected by tree depth. To avoid endless search in a dead branch (having no solutions), cut-off depth rules can be specified to switch branch if no solutions are found for an *x* amount of nodes.

A scenario where it could be used; maybe a side decision tree considers an additional quest which asks the player to explore new resources by roaming the game world while still progressing on the main mission. On a different note, the inconsistent branch selection nature for newly generated trees makes it a good choice for AI testing, it is the sole algorithm (From A-Star search and BFS) which is designed to select solutions in a depth motion, hence reaching deeper results which in turn makes it ideal for AI demonstration. Solutions with more steps are more likely

to be selected, hence making it entertaining for spectators. Alternatively, its search behaviour can be applied to multiple NPC (Non-player characters) so they would act in a random fashion, varying from each other. Subsets of the complete action pool can be used for different character types to obtain different solutions, hence NPC will possess a different AI behaviour. Take a group of villagers for example, solution cost doesn't matter but it provides a better game-play aesthetic. This algorithm should be avoided at all costs when decision trees constitute a high depth level since it is considered as the algorithm's worst-case.

Chapter 6 - Conclusion

6.1 Objectives

The main reason for this study was to employ pathfinding techniques onto decision trees, allowing the AI agent to search the next action towards a specified goal. Commands were built in the form of executable AI actions, so they could be compiled into tree nodes and utilised within a decision tree network. A variety of tree traversal algorithms were used for solution finding, comparing their selection behaviour across similar tree structures.

6.2 Algorithm Utility

A-Star search, being a pathfinding algorithm, has proven to be worthwhile for decision tree traversal, in the context of building an AI agent. The assignment of cost to each step in the action sequence has developed an efficient method for selecting the cheapest of solutions available for the AI agent to carry. An optimal sequence of actions is always guaranteed from the search result, instructing the AI agent with the least amount of work to reach a Goal state.

6.3 Prototype Evaluation

The concept of Goal-Oriented Action Planning (Orkin J. 2004) has shown excellent AI potential when compiled into an adequate strategy for Minecraft. Using a "work-towards-goal" approach, allowed the agent to plan ahead a sequence of actions and ignore surrounding objects that were of less importance to the current task. The prototype's functionality was kept at a minimum scale compared to the full build of Minecraft, otherwise results would have been ambiguous to examine if taken to full scale. The time allotted for this research was limited, hence made it impossible to implement every AI action for Minecraft since possibilities are infinite. Moreover,

every action required adaptation for item scanners, being time consuming to employ into methods. To test similar approaches to GOAP, a total of 27 unique AI actions were implemented. The game offers unlimited possibilities for decision tree application or similar logic due to its sandbox game nature, endless results can be obtained from this open-world environment.

GOAP has shown beneficial features in both development and runtime, the source code behind the AI agent's behaviour is structured, reusable and maintainable. New actions can be added independently to the pool without affecting other actions. If an action is no longer needed for solution input, it can be deactivated with a sole boolean value, without physically removing it. New AI strategies can also be attached to the action without altering its structure. AI planning can adapt to its current surroundings, actions which do not meet requirement are not considered during solution finding.

The AI level achieved by the prototype is still limited. Although the agent decides what sequence of actions to take to reach a set objective, the goal is still specified by the user rather than the AI agent deciding for itself what the next goal should be. Take the two goals for example, 'Get Food' and 'Get Fuel'. If 'Fuel' items are not available to light up the furnace, food which must be cooked first cannot be obtained. This shows that 'Get Fuel' should run first, followed by 'Get Food'. Moreover, if the player is extremely hungry, the 'Get Food' goal should be activated from high priority, allowing the agent to scavenge any food that does not require any cooking.

6.4 Future work

A more complex tree structure can be designed to instruct the AI agent with more detailed actions. Each Node in the tree can expand itself further into another tree, employing a number of strategies for a sole action, this will result in a tree of trees. Take the 'CraftAxe' action for example, different types of axes can be crafted, i.e. Wooden Axe, Stone Axe, Iron Axe, hence different strategies can be described under the same node. Actions can be abstracted to a desired amount of levels, refining behaviour with each layer. Moreover, different traversal algorithms can be employed at different levels.

Another suggestion for improvement involves a one-time population of all possible decision trees, storing only the logical frame for AI agent behaviour. Alternatively, tree structures can be recorded as soon as the related goal is specified, hence using the same tree for recursive calls. Various trees can then be searched from a repository, each identified by their unique 'Goal' state (Root Node). Having the tree structure already generated, the only work which would be required is the re-calculation of action cost, overwriting existing data from the previous call.

More points to consider for prototype improvement:

- Implement algorithm switching between tree traversal techniques, changing according to the tree being searched or scenario presented. Algorithms have their strength and weaknesses for specific environment settings.
- Experiment with technique where action validation is done during tree generation. If the 'checkRunnable' method returns a 'false' boolean value, the action is not transform into a tree Node, hence further evaluation is not performed on the respective action and is dumped from the list. A validation check should run on the action pool pre-population of the tree, passing a list containing only valid actions.
- Experiment with technique where solution finding is done during tree generation, meaning that a valid action sequence is obtained during the course of tree population. If a Node with appropriate conditions is found, further building will not be required, hence returning the solution more urgently.
- Make variable cost calculation more dynamic. The nodes' world-position (distance from goal) should update upon entity movement. Take 'Sheep' entity for example, it can move from its primary recorded position during tree generation, hence it will result to inaccurate solutions if compute time takes too long.
- Make block range scanning more realistic, scanning blocks which are not visible to the player is considered cheating. Moreover, store objects of interest in a list which would be referred to during solution finding, hence deleting unwanted scanned blocks from memory.

- Consider player position when calculating actions, maybe the first solution found is too far away from the player than the second best solution.

Bibliography

Abel, D., Hershkowitz, D.E., Barth-Maron, G., Brawner, S., O'Farrell, K., MacGlashan, J. and Tellex, S., 2015, April. Goal-Based Action Priors. In ICAPS (pp. 306-314).

Abel, D., Agarwal, A., Diaz, F., Krishnamurthy, A. and Schapire, R.E., 2016. Exploratory gradient boosting for reinforcement learning in complex domains. arXiv preprint arXiv:1603.04119.

Alford, R., Shivashankar, V., Roberts, M., Frank, J. and Aha, D.W., 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In Proc. of the Int'l Joint Conf. on AI (IJCAI). AAAI Press.

Bay, H., Tuytelaars, T. and Van Gool, L., 2006. Surf: Speeded up robust features. Computer vision–ECCV 2006, pp.404-417.

Bonanno, D., Roberts, M., Smith, L. and Aha, D., 2016. Selecting subgoals using deep learning in minecraft: A preliminary report. In Deep Learning for Artificial Intelligence: Papers from the IJCAI Workshops.

Champandard, A.J., 2013. Planning in games: An overview and lessons learned. AIGameDev.com.

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

Cui, X. and Shi, H., 2012. An overview of pathfinding in navigation mesh. IJCSNS, 12(12), pp.48-51.

Graham D. (n.d.). An Introduction to Utility Theory. [PDF] Available at: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf [Accessed 10th January 2017]

Harabor, D.D. and Grastien, A., 2011, August. Online Graph Pruning for Pathfinding On Grid Maps. In AAAI.

Johnson, M., Hofmann, K., Hutton, T. and Bignell, D., 2016. The malmo platform for artificial intelligence experimentation. In International joint conference on artificial intelligence (IJCAI) (p. 4246).

Larmore L.L., n.d. DFS and BFS Algorithms using Stacks and Queues. [PDF] Available at: http://www.egr.unlv.edu/~larmore/Courses/CSC477/bfsDfs.pdf [Accessed 15th March 2017]

Lim, C.U., Baumgarten, R. and Colton, S., 2010, April. Evolving behaviour trees for the commercial game DEFCON. In European Conference on the Applications of Evolutionary Computation (pp. 100-110). Springer Berlin Heidelberg.

Mark, D., Dill, K. and Engineer, A.I., 2010. Improving ai decision modeling through utility theory. In Game Developers Conference.

Mateas, M. and Stern, A., An Experiment in Building a Fully-Realized Interactive Drama. In Game Developers Conference, Game Design Track (March 2003).

Millington, I., 2006. Artificial intelligence for games. Morgan Kaufmann series in interactive 3D technology. Taylor & Francis. pp 9-11, 223-246.

Orkin, J., 2004. Applying goal-oriented action planning to games. AI Game Programming Wisdom, 2(2004), pp.217-227.

Orkin, J., 2006, March. Three states and a plan: the AI of FEAR. In Game Developers Conference (Vol. 2006, p. 4).

Perez, D., Nicolau, M., O'Neill, M. and Brabazon, A., 2011. Evolving behaviour trees for the mario ai competition using grammatical evolution. Applications of evolutionary computation, pp.123-132.

Perez, D., Nicolau, M., O'Neill, M. and Brabazon, A., 2011, August. Reactiveness and navigation in computer games: Different needs, different approaches. In Computational Intelligence and Games (CIG), 2011 IEEE Conference on (pp. 273-280). IEEE.

Peters, C., Kyaw, A.S. and Swe, T.N., 2013. Unity 4. x Game AI Programming. Wydawnictwa PACKT. pp 159-184.

Roberts, M., Alford, R., Shivashankar, V., Leece, M., Gupta, S. and Aha, D.W., 2016. Goal reasoning, planning, and acting with actorsim, the actor simulator. In ICAPS Workshop on Planning and Robotics (PlanRob).

Stanley, K.O. and Miikkulainen, R., 2002. Evolving neural networks through augmenting topologies. Evolutionary computation, 10(2), pp.99-127.

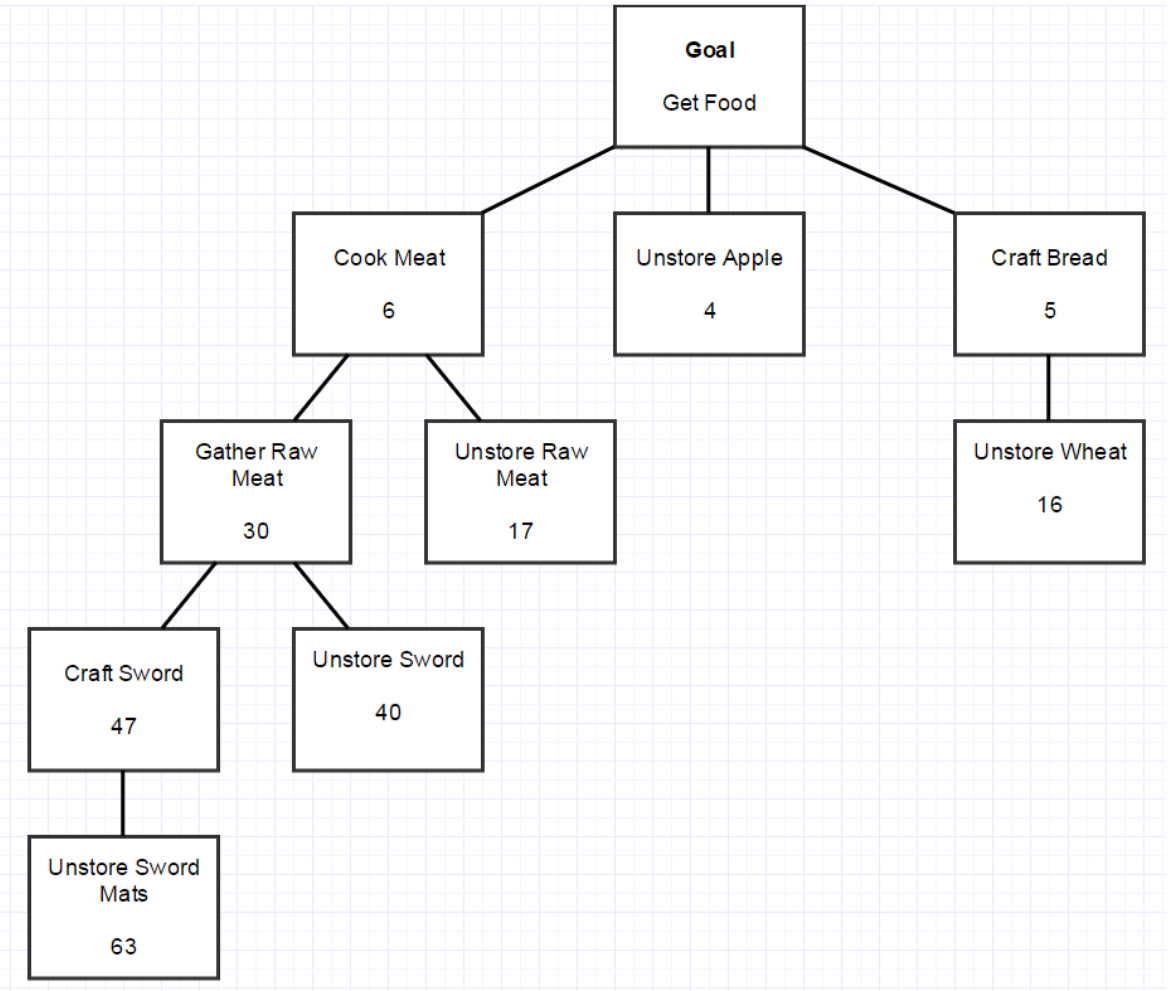
Appendix

System Specifications

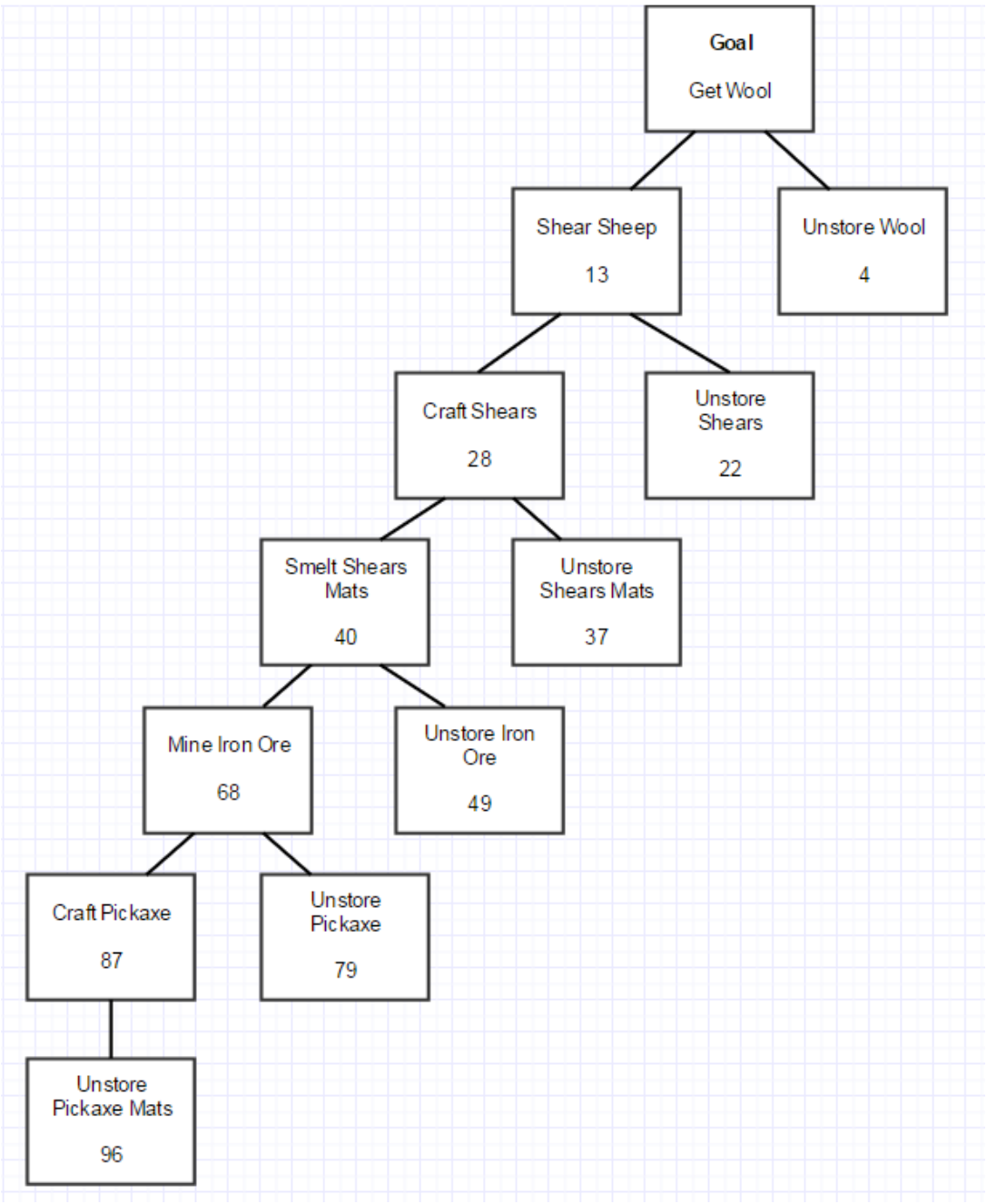
Processor	Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz
Video Card	NVIDIA GeForce GTX 960
RAM	8.0 GB
Operating System	Microsoft Windows 7 Home Premium Edition Service Pack 1 (build 7601), 64-bit

Decision Tree - Visual Representation

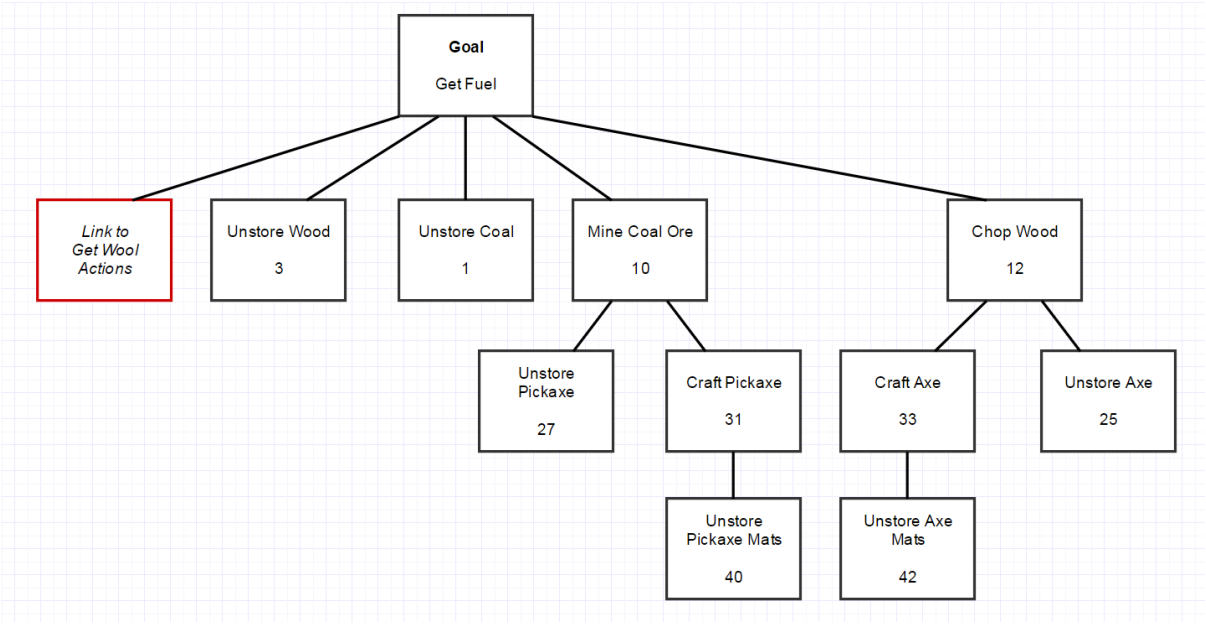
Tree No.1 *(Test Map No.1 - Get Food goal)*

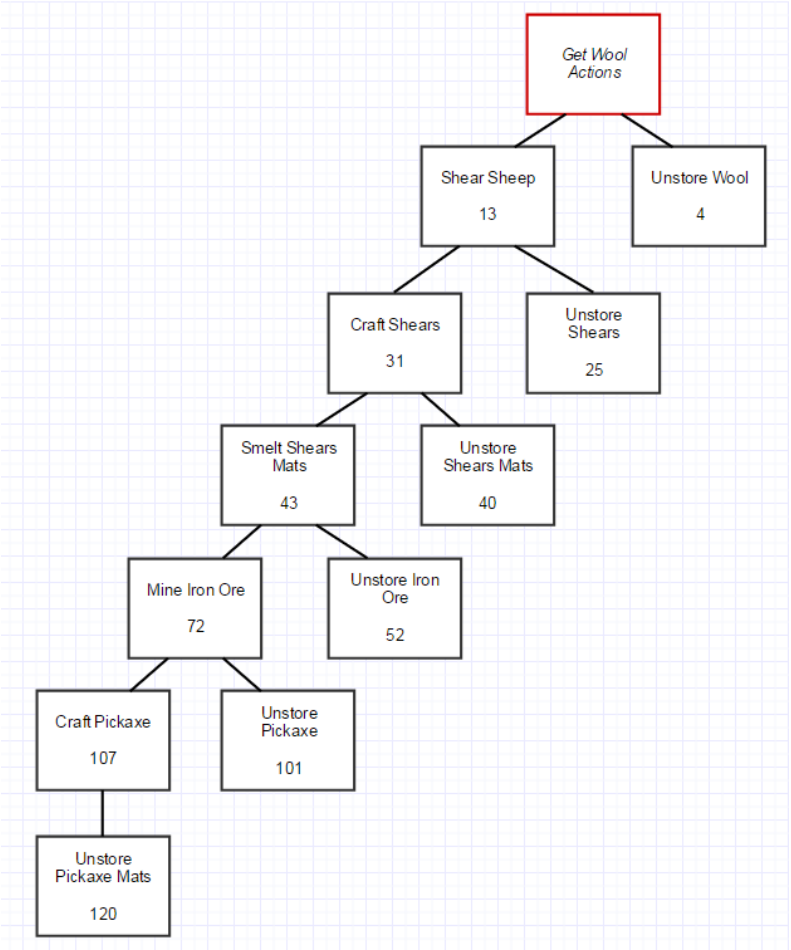


Tree No.2 *(Test Map No.2 - Get Wool goal)*



Tree No.3 *(Test Map No.3 - Get Fuel goal)*





Project Source

Framework available online at:

https://github.com/michaelzangl/minebot/

Prototype available online at:

https://github.com/marloncalleja/MC-Experiment-AI