PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA

Guillermo Borges

BDI Agents in Minecraft

Porto Alegre

2015

Abstract

As video games and the hardware that powers them grow in complexity, the demand for challenging, believable, engaging artificial intelligence grows. Games are made in a wide variety of programming languages, frameworks and environments, yet the resources available for AI development are either sparse or requires expertise on the field that many aspiring game developers do not have.

This final year project attempts to address that issue, providing a framework design and implementation aimed at dynamic languages by leveraging the expressive nature of popular languages such as JavaScript, Lua, Ruby, and others. The framework can be implemented with minimal difficulty using simple algorithms, code conventions and string manipulation. The Lua implementation used as proof of concept is a fully featured BDI (Belief, Desires, Intentions) agent framework slightly over a hundred lines of code, without any dependencies.

The Lua framework served as proof of concept using Minecraft as a robot simulation environment. Thanks to a modification of the game called ComputerCraft, I loaded Lua scripts into programmable turtles inside the game. The result was an agent simple to develop and extend, able to mine shafts for the player or find its way around obstacles to random locations. The scale and complexity of the tasks were limited by the turtle's range of visibility and the virtual GPS accuracy, but nevertheless was sufficient to showcase the capabilities of this framework.

The source code of this project can be found at <https://github.com/elite5472/turtleai>

Keywords: Artificial Intelligence, Agent Theory, Planning, BDI, Minecraft.

Contents

1 INTRODUCTION . . . 5
1.1 Justification . . . 6
1.2 Approach . . . 7
1.3 Objectives . . . 8
1.3.1 General Objective . . . 8
1.3.2 Specific Objectives . . . 8
1.4 Document Overview . . . 8

2 AGENTS . . . 11
2.1 Environment . . . 12

3 PLANNING . . . 13
3.1 Classical Planning . . . 13
3.2 Hierarchical Planning . . . 14
3.2.1 Hierarchical Task Networks . . . 14
3.3 Beliefs, Desires, Intentions . . . 15
3.3.1 Beliefs . . . 16
3.3.2 Desires . . . 16
3.3.3 Intentions . . . 17
3.3.4 Planning During Execution . . . 17
3.4 Discussion . . . 17

4 PATH FINDING . . . 19
4.1 A*, a Heuristics approach. . . . 19

5 RELATED WORK . . . 21
5.1 Jadex . . . 21
5.2 Jason . . . 22
5.3 Discussion . . . 22

6 MINECRAFT . . . 25
6.1 Game Objectives . . . 25
6.2 World Structure . . . 26
6.3 World Entities . . . 27
6.4 Internal Structure . . . 27
6.5 Challenges for Artificial Intelligence . . . 28
6.6 ComputerCraft . . . 29

6.6.1 Turtles . . . 29
6.7 Turtles and the Environment. . . . 30
6.8 Discussion . . . 31

7 FRAMEWORK SPECIFICATION . . . 33
7.1 Design . . . 33
7.1.1 Programming Language . . . 33
7.1.2 Execution Workflow . . . 34
7.1.3 Structure . . . 35
7.2 Architecture . . . 36
7.2.1 Condition Strings . . . 37
7.2.2 Agent Object . . . 37
7.2.3 Agent Engine . . . 39
7.2.3.1 Parse Condition String . . . 39
7.2.3.2 Evaluate Conditions . . . 39
7.2.3.3 Check Desire . . . 39
7.2.3.4 Check Desires . . . 40
7.2.3.5 Set Intent . . . 40
7.2.3.6 Step . . . 40
7.3 Implementation . . . 40

8 METHODOLOGY . . . 45
8.1 Development and Experiment . . . 46
8.2 Challenges . . . 49

9 CONCLUSIONS . . . 51
9.1 Observations . . . 51
9.2 Future Work . . . 52
9.3 Closing Remarks . . . 52

BIBLIOGRAPHY . . . 53

1 Introduction

Within the past few decades of the history of video games, the industry has evolved along with technology. As of today, anyone can download the latest version of Unity or Unreal Engine, free of charge, and develop their own video game with the same tools many professional studios use for their titles. Distribution platforms like Steam, or Microsoft and Sony's indie publishing platforms for their game consoles let anyone with a computer, the knowledge and the assets, to create the experience they envision and distribute it commercially around the world.

Often times a game has entities within a fictional world that need to behave in complex ways to give the player a sense of realism or immersion. Games often want to convey to the player that its world has an internal, consistent framework of rules, a history behind it, to make an artificial world feel more substantial. Other games require an opponent that challenges the player, or many opponents that act as obstacles between the starting point and the game's objective. In these cases, developers are faced with the challenge of creating artificial intelligence, or AI, to bring their ideas into a functional game. An embodied intelligent entity is often referred to as an agent, responsible for their own actions and separate from the space they interact with. Through *sensors*, agents perceive their environment and maintain an internal model of *knowledge*. Through *effectors*, agents perform *actions* that alter the environment. In other words, agents are input and output interfaces by which an AI can observe and interfere with an environment.(RUSSELL; NORVIG, 2009, p31)

The process by which an agent uses its acquired knowledge of the environment and its current state, to select and perform actions is *reasoning*. The result of the reasoning process is a *decision*. There are many models and algorithms that perform this reasoning process, each with their own set of advantages and limitations. Some are more suitable than others for different types of problems.

In the context of video games, with the endless variety of rules, scope and constraints, finding or designing the most suitable AI for a particular game entity is a problem of its own. It requires analyzing the environment the agent will inhabit and interact with, the scope and limitations of the agent's capabilities, as well as its purpose within the game.

Minecraft (MINECRAFT, 2015) is a video game where players are placed into an endless world made up of blocks that can be removed and placed. Trees and mountains alike are made out of these blocks. Blocks are of different materials, trees made from blocks of wood and leaves, mountains from dirt, stone and minerals. It is up to the player to decide what to do in this artificial world. As the game progresses and the player begins

to craft tools and build structures, the focus shifts from survival to envisioning creative ideas. The player is then faced with the tedious task of obtaining the resources required for building and crafting. ComputerCraft (COMPUTERCRAFT, 2015) was developed as a solution for this problem. It is a modification of the game that adds computers and robots named turtles into the game to automate tasks. The turtles once crafted behave as programmable robots that can gather resources and build for the player, provided they have the materials and fuel required. Turtles are merely an interface, as they do not come with an artificial intelligence to control them. It is up to the player to use the in-game editor provided by the modification, or to download from the web a Lua script that will put the turtle in motion.

The environment in Minecraft is dynamic. Blocks can be blown by explosions, and mining without precautions can cause various disasters, such as water or even lava floods. Monsters inhabit underground caves and roam the world at night. Moreover, the field of view of turtles is limited to the blocks directly adjacent to them. While bots are indestructible, players are not, and valuable minerals can be lost as a result of mining without precaution. These are some of the problems an AI has to deal with when operating a turtle. As such, it is imperative to design a model that can react to an ever changing, hazardous environment.

Moreover, to assist the player with complex tasks in games such as Minecraft, AI may be required to be more than just reactionary. The process of building an efficient mine requires thorough planning even for players themselves, from size to accessibility, material yields and area coverage. Stairways and transportation methods must be built as the mine grows, or else the player will simply be unable to explore them. Additionally, monsters will spawn inside without placing torches to light the tunnels, hindering both players and turtles and possibly making the mine unusable.

1.1 Justification

Creating reasoning models from scratch, or even an implementation of it can be challenging to programmers lacking knowledge on the subject. With the ever increasing number of programming languages, frameworks and hardware configurations involved in making games, several projects do not have the option of using an existing framework that does most of the work regarding the reasoning process. For example, the Lua programming language has few resources available to develop AI for games. (LUA. . . , 2015)

It becomes apparent with the lack of AI resources in Lua, the language used to program ComputerCraft turtles, making them perform useful tasks with so little to work with is difficult enough put off aspiring developers from even trying. The same situation occurs with many popular game engines, which often use dynamic languages for scripting

in game events. Unity for example uses JavaScript, a language very similar to Lua. How is it then that a developer with little expertise in AI can make something that works for their game?

1.2 Approach

Some AI frameworks, such as Jason use their own language to express agent behavior. (JASON. . . , 2007, p.1) The advantage of this approach is an agent that is not tied to a platform or language, and a syntax that is specialized to develop agents according to their respective design philosophies. However, this approach has trade-offs that ought to be considered. Platform independence also means an interpreter or compiler has to be designed for each platform it is intended to run on, or platform integration has to be in place in order for the existing interpreter to interact with the application. Additionally, a dedicated language is an entire new tool-set that requires learning and training, whereas a solution that uses the application's same language can leverage the existing knowledge of the programmer.

A middle-ground solution to this problem of developing AI is a Domain Specific Language, or DSL, as a subset of a general purpose language. A DSL provides constructs, design conventions and workflow relevant to a specific domain, such that similar results to a dedicated language can be obtained within an existing one. In other words, a language can be extended through rules and conventions. While languages and platforms are all different in their own ways, these rules can remain universal through implementations of the same framework, therefore providing a consistent experience while simultaneously leveraging the capabilities of a particular platform. One well known example of a DSL is OpenGL, a graphics API. OpenGL wrappers are by themselves a set of functions, some of which represent language-like constructs such as *glBegin* and *glEnd*. (OPENGL. . . , 2015)

ComputerCraft's Lua interpreter does not contain the stack of libraries that come with the standard distribution of Lua. Moreover, integrating with an agent framework such as Jason makes installation and distribution more difficult, as more dependencies, and additional setup is added, severely impacting accessibility for the end user. In order facilitate the creation of more complex AI for turtles, thus enhancing the player experience, a framework has to be lightweight and developed in lua. The same can be said for many game projects that make use of less common languages, an AI solution has to be approachable to *use* and to *create* at the same time.

1.3 Objectives

1.3.1 General Objective

The purpose of this project is to create a framework for BDI agent development, oriented towards people with basic programming knowledge and artificial intelligence concepts, then utilize it to operate turtles in Minecraft to automate tasks with a proactive BDI agent.

1.3.2 Specific Objectives

- Design a framework specification for BDI agents
- Implement and test the framework in Lua as a proof of concept
- Define an experiment within Minecraft, with problems to solve using the BDI framework on ComputerCraft turtles
- Implement a BDI agent for turtles capable of completing tasks and solve problems as defined by the experiment
- Documennt the agent's behavior during the experiments
- Determine the quality of the written code using the framework, making considerations of ease of use, readability and dependencies.

1.4 Document Overview

This document is split into nine chapters:

- Chapter 2 describes the basic concepts and terminology regarding agents and the way they interact with the environment they reside in and their intrinsic properties
- Chapter 3 goes over various models that address the problem of planning for AI, outlining their strengths and weaknesses
- Chapter 4 explains briefly how path finding relates to this project and what algorithm will be used
- Chapter 5 summarizes two other planning frameworks that relate to this project
- Chapter 6 describes in debt how the virtual world of Minecraft functions, the potential applications of agents in the game and the characteristics of the environment from their perspective

- Chapter 7 defines, justifies and explains the framework specification, its design philosophy, architecture and the Lua example implementation
- Chapter 8 documents the experience of using the framework to develop a Minecraft turtle agent, the methodology leading up to development and experiments and the challenges involved
- Finally, Chapter 9 delivers the closing thoughts and conclusions, summarizing subjective observations, future work and closing remarks.

2 Agents

There are multiple interpretations of what constitutes an agent, but in its most basic form an agent is, by the meaning of the word, an entity or object with agency over its own behavior (WOOLDRIDGE; JENNINGS, 1995). An agent is at least in one way or another an independent entity that interacts with an environment. The underlying algorithms, or reasoning that the agent performs to choose its actions

Agents usually have some kind of purpose they were created for. Said purpose is what guides the process of designing and deploying agents in an environment. What makes an agent is the independence of their actions and their ability to interact with an environment.

An agent's concept of independence is relative, it has been conceived by a designer and that such a design inherently defines the set of actions it can perform under a given set of circumstances. Still, an agent is considered an independent entity because its inner, decision making process is isolated and constrained by a set of sensory inputs (sensors) and actions it can perform (effectors) (RUSSELL; NORVIG, 2009, p34).

A sensor is any mechanism by which an agent can obtain information from the environment. Cameras for visual input, microphones for audio, or anything that can provide the agent with valuable data that it can use to make a decision. Sensory data, by itself, is not necessarily reliable. Data must be interpreted into *knowledge* for it to be useful, and often times the data received is inaccurate or leads the agent to the wrong conclusions. Some environments guarantee reliable data, while others, such as real-world sensors, can be inaccurate or prone to malfunction.

An effector, in contrast is any means by which an agent can interact with the environment. A robotic arm or wheels, for example, are effectors that allow a robot to grab objects and move itself around the place. If sensors are the input, effectors are the output, providing means by which the agent can achieve its objectives in the domain they are working with. Just like sensors, effectors may not always be reliable. Actions taken by the agent may fail due to unforeseen circumstances, although there are domains where every action is undoubtedly successful.

What ties sensors and effectors together is an underlying mechanism. The agent's behavior. With both sensors and effectors being a defined, limited scope, AI development can be encapsulated into a simpler process that is analogous to a function.The concept is key to building complex planing algorithms despite its simplicity.

2.1 Environment

In the context of agent theory, an environment is everything the agent can either perceive or act upon. As mentioned briefly previously, the conditions in which an agent operates can alter the way it functions. There are several classifications by which an environment can be characterized, and they affect how an agent interacts with it, as well as the necessary design considerations involved for an agent's inner logic when dealing with some of these characteristics. (RUSSELL; NORVIG, 2009, p.42)

- Accessibility. An environment's accessibility is related to what information the agent can draw upon it. An environment is accessible if an agent does not need to keep an internal model of its composition, therefore at any given time it can perceive any information it may require from it.

- Deterministic or Nondeterministic. Whether the environment is deterministic or not is up to the observer and thus, the agent. When every influencing variable, factor, and event occurring in the environment is perceived by the agent, it can be considered deterministic. Given a set of inputs, the agent can then predict the outcome of its following actions with complete certainty. On the other hand there are nondeterministic environments that have unpredictable and chaotic elements to it. A nondeterministic environment has factors in it that the agent cannot perceive, therefore seemingly random. An agent can account to the randomness of these events, or even predict their consequences with a certain degree of accuracy, but the guarantee of success is never assured. When an environment has seemingly random events, an agent's actions may or may not yield the expected results.

- Static or dynamic. Refers to what happens during the discrete time the agent spends reasoning about its following action. If the environment remains frozen and unchanging during that period, it is static. If, however, things can happen while the agent is determining its next course of action, therefore possibly interfering with its chosen decision's outcome, it is dynamic. A dynamic environment has many unexpected elements that the agent may have to account for when making decisions.

- Discrete or Continuous. This refers to the range of the sensors and effectors an agent has. For example, a card game is a discrete environment because all the information is discrete and punctual, whereas an adventure game in a 3D world has continuous spatial information.

3 Planning

Tasks are often problems with multiple step solutions, so when considering agents that complete tasks assigned to them, planning is as much a problem as the solution itself. Coming up with plans requires information on the problem domain, as well as expertise on how to solve it. It only gets more complex when there are multiple, simultaneous problems to solve. Over time researchers have come up with a variety of planning algorithms to address this issue, each with their strengths and weaknesses.

3.1 Classical Planning

Classical Planning is referred to as the earlier models introduced for planning systems, most notably STRIPS or Stanford Research Institute Problem Solver. (NAU; GHALLAB; TRAVERSO, 2004, 76) One important characteristic of STRIPS and other variations of the classical model is their similarities with logic languages, such as Prolog, with the use of a backtracking engine over a knowledge base that represents the problem domain.

These models were designed to solve problems with certain characteristics regarding the environment: (RUSSELL; NORVIG, 2009, p65-66)

- Observable: accurate perception of the environment.
- Discrete: a finite set of possible actions at each state of the problem.
- Known: the result of an action is assumed to be predictable.
- Deterministic: agent's cannot have multiple possible states at a given point.

The structure of STRIPS is separated into two different elements: the problem domain and the problem solver. The problem solver is a backtracking engine that uses the constraints and actions specified in the domain to find a solution to the problem presented. The problem domain has all the knowledge necessary to come up a solution: the initial state of the environment, its desired state, and a set of actions the agent can effectuate, each with preconditions and the resulting effect of said actions. The plan is the byproduct of proving that the final state (the solution) can be reached from the initial state (the problem). (FIKES; NILSSON, 1971, p.190).

When defining the domain of the problem, the most essential step is to first define what the agent can do to solve the problem. Each action has a set of preconditions that

are required for it to be available as the next step of the plan. For example, to pick up a box, said box must be within reach. Likewise, actions have consequences. They alter the state of the environment. When a box is picked up, it has been dislocated from its original position and is now in the possession of the agent. The model is uninterested in how the actions themselves work, so there is no need for any further information. (FIKES; NILSSON, 1971, p.197)

3.2 Hierarchical Planning

Hierarchical solutions arose from the necessity of reasoning systems that could handle higher complexity problems with fewer resources. STRIPS is a single layer of abstraction, with each action being atomic and indivisible. Therefore, it can either provide basic, high level solutions, or what can end up seemingly a huge list of actions with no contextual relation to each other. (RUSSELL; NORVIG, 2009, p.406)

This is unpractical, and in many cases, depending on the complexity of the problem, expensive enough to be considered unsolvable. However the problem of complexity can be addressed by grouping actions together. A mechanical arm, for example, can have several hundreds of small actions that lead to its arm movement, yet a planner can group these together into specific actions such as lift and drop to elaborate a plan regarding object placement, and then expand these high level steps into the granular motor movements the machine understands.

By using generalized, hierarchical terms, plans can be formulated in a higher level context, and then expanded further under a smaller set of possible choices.

3.2.1 Hierarchical Task Networks

A Hierarchical Task Network is a structured planning tree where each task has a set of sub-tasks required to complete them, and those that do not are considered leaves or atomic tasks. Atomic tasks are the lowest level of detail for the problem's scope, and once a plan is formulated the result will be a list of atomic tasks that will achieve the desired result. (EROL; HENDLER; NAU, 1994)

Each task has a precondition attached to it, and if the precondition is not met the particular branch is then considered unreachable, and the planner will try to find a different solution. This, along with the tree structure makes HTN quite efficient at making plans, avoiding branches that do not lead to the solution entirely.

HTN Shares some similarities with classical planning, in fact being an extension of the concept, with the addition that actions, on top of having prerequisites and effects, also have sub-tasks which are required to be executed for the task to be considered complete.

The output of the HTN algorithm is the list of atomic, or leaf tasks that the agent executes in order to achieve its goal. That means that the entire plan is formulated from the start, rather than during the process of executing it. This process is referred to as expansion or decomposition.

What allows HTN to be fast and efficient is its structure. Unlike STRIPS, the task composition already constrains the domain of the problem significantly. Choosing one approach constrains the following actions. When an agent, for example, decides to kick a ball, the hierarchic nature of HTN already delimits the problem to a possible set of movements, which won't involve things like punching it instead. That said, it also means that HTN intrinsically requires information on the relationship between actions, something STRIPS does not require. (GEORGIEVSKI; AIELLO, 2014, p.2)

3.3 Beliefs, Desires, Intentions

When looking at artificial intelligence as a way to mimic human reasoning, there are multiple ways to interpret how humans actually think and produce an AI that goes through a similar process. If the objective is to produce reasoning that mimics that of us humans, it does not have to be at all like how humans actually think, just close enough to fulfill its purpose. In many cases, all an AI has to accomplish is to behave as a human expert would in a given domain. Therefore, one solution is to provide that expertise to the AI, regardless of how.

When consulting a professional in any given domain what the reasoning behind their actions and methodology are, the information obtained is not the actual thought process, but the impression said professional has on their own behavior. It is a reflective, high level description. Simply using that knowledge can be far easier than emulating said thought process in its entirety. For example, translating the knowledge of a plane's pilot into an automated computer agent: said pilot has to be consulted, and the idea is to have a model in place that can, through a well directed interview, capture the core of the pilot's reasoning process and put it into something the computer can understand. This is the folk psychology approach, an it is the underlying principle by which the Beliefs, Desires, and Intentions model is based on. (NORLING; SONENBERG, 2004)

A BDI agent makes decisions based on what it currently knows about the environment, its *beliefs*, whether accurate or mistaken. It decides to commit to certain desires or goals and attempt to fulfill them. Once an agent has committed to a goal, it has the *intent* of carrying out a plan that can achieve it. Even then, plans can change at any time as the circumstances in the environment produce emerging problems that have to be dealt with. The BDI agent is continuously evaluating what tasks take priority given the circumstances. BDI agents are by design never so committed to a plan that they ignore more important

issues as they arise.

The BDI model makes *no assumptions* about the properties of environment it interacts with. It can be deterministic, nondeterministic, known, unknown, discrete, continuous, and the environment may even change while the Agent is in the middle of reasoning. Its focus is actually introspective. All three components of the model and its core philosophy are oriented towards the agent itself, and not the nature of the problems it has to solve. (RAO; GEORGEFF, 1995, p313)

The model not only deals with ways to solve one particular problem, but any amounts of problems a given domain has by weighting priorities of each desire or goal. This is an increase in scope, which comes with the advantage of allowing the Agent to work on multiple tasks or problems as single, isolated entities called goals. A BDI agent's objective is to fulfill its *desires* to the best of its ability, which may or may not involve solving complex problems.

3.3.1 Beliefs

A BDI agent's knowledge of the environment is completely self-contained. The agent does not know any hard facts about the environment, but rather makes assumptions based on the raw data obtained through its sensors. That data is then turned into information it can use to reason with. This is distinction is what allows an agent to function in a continuous environment where time passes even as the agent is busy during the reasoning process. Information may become outdated as the circumstances change and then updated as the agent obtains new data from the environment.(RAO; GEORGEFF, 1995, 313)

The accuracy of the belief may also be in question, as the agent can even doubt the validity of it and formulate plans to obtain more accurate information. Information, not being factual, may be outdated, wrong, inaccurate or misinterpreted. An agent may be as optimistic or as pessimistic as it is designed to be. Even if the information the agent perceives is factual and always correct, the distinction between fact and belief aids in situations where this is not the case.

3.3.2 Desires

BDI agents work with series of goals to formulate plans. Desires are those goals. An agent may look at all the desires it has and then plan accordingly to fulfill them to the best of its ability. Desires are chosen based on their importance at any given time, as well as the viability to fulfill them. Desires that cannot currently be fulfilled are not chosen over those that can. (RAO; GEORGEFF, 1995, 314)

Not all desires are discrete or even permanently satisfiable. A desire's priority can vary as the agent's belief's changes, or it can be such that it can never be truly

accomplished. The distinction occurs because continuous goals are passive, or at least until some condition occurs in which case the goal is no longer in effect. A plane pilot agent may, after gaining enough altitude, obtain the desire of maintaining altitude until it is time to land. This desire is never truly fulfilled, but is present regardless. Some desires may never be achieved at all, or even considered, depending on the circumstances. Not everything the agent wants can be done given constraints, and the distinction allows flexibility in the agent's thought process.

3.3.3 Intentions

Once the agent has decided a given desire is worth working towards, it changes into an intention. Intentions are goals the agent is committed to attain, and work towards. Once the course of action is decided that planning starts. The process of choosing goals, or decision making defines what the agent will do, while the planning process decides how.(RAO; GEORGEFF, 1995, 314)

3.3.4 Planning During Execution

The planner does not need to concern itself with desires the agent has not committed itself to, which reduces the scope to only the exact problem at hand, greatly reducing the the calculations required. Since goals can change at any time, it is pointless to expand plans before it is required. *The planner itself works out the details only as they become relevant, reducing greatly the cost of having to discard a plan for another.* The downside is that the planner never knows the entire plan until nearly the entirety of it has been carried out. Therefore, unlike classic planners that provide complete solutions, BDI is emergent, and works on the plan as it is executed. The agent may not immediately know if a plan cannot be carried out until and if it fails.(RAO; GEORGEFF, 1995, 314)

3.4 Discussion

Classical models make very clear assumptions that exclude them as solutions for problems in environments with these characteristics. STRIPS relies on an accurate knowledge base and description of the effects of its actions. Real numbers and other infinite sets as parameters make the possible set of actions infinite, therefore a classical planner such as STRIPS is not be able to solve it. Many, if not all of these conditions are often present in games. Real-time simulations can cause the circumstances to change while AI is still processing its course of action. Plans may fail partway through due to the many unforeseen variables, including the actions of the players themselves. Finally, even a simple game can have more possible actions than the planner can realistically handle.

One of the reasons classical planners struggle with dynamic environments is the *the cost of computing the entirety of a complex plan is relatively expensive.* Under ever changing circumstances, that can happen even while the agent is reasoning, the cost adds up to the point where it is not feasible. Producing plans quickly and efficiently is specially important in real time systems like video games, where computing time is of essence and an agent may be stuck in an endless planning and discard process, never getting anything done.

With BDI's philosophy based in folk psychology, translating knowledge from players or designers to a model a program understands is relatively trivial. Agents can then leverage this knowledge to adapt and thrive in environments. It is also simple to implement, being essentially a plan *chooser* than a plan *maker*. For this final year project's placement as a game AI framework, the BDI model is the most suitable option studied.

4 Path Finding

A prevalent challenge agents may face in spatial environments is moving from one place to another. It can be as simple as moving in the general direction, however the presence of obstacles and irregularities often make that course of action inviable. These obstacles can cause greedy path searches to take much longer than expected to find the most optimal path.

Effective path finding is extremely important in a wide variety of applications. Particularly when resources or time is limited, sub-optimal routes may result in the agent failing its intended objective, or simply poor performance on the system as a whole.

A* is the most widely known path searching algorithm. (RUSSELL; NORVIG, 2009, p.95). Since this is not the focus of this project, it was chosen with little considerations as it is thoroughly documented and easy to implement.

4.1 A*, a Heuristics approach.

The A* algorithm utilizes heuristics to dramatically increase its efficiency. Its use is widespread due to its good performance, simplicity and accuracy.(NOSRATI; KARIMI; HASANVAND, 2012, p.251)

Its heuristic function is the sum of the calculated cost of the path plus an estimate to the goal. It works similarly to Dijkstra's Algorithm, with the addition of the heuristic to sort the search order. A* *expands* possible paths each iteration, prioritizing the ones with the lowest heuristic value. Once a path is found, regardless of being the shortest or not, A* will discard any paths that have lower heuristic value than it.

That said, A*'s performance greatly depends on the accuracy of its heuristics. In order for it to work there can be no path with a lower cost than the optimistic estimate from the start to the goal.

5 Related Work

5.1 Jadex

Jadex is a Java and XML based middleware focused on simplifying distributed systems and BDI agents. One of its defining characteristics is that Agents are not programmed using a third-party language, but instead Java and XML with the use of various conventions and annotations. The reasoning behind the conservative approach is it allows the programmer to leverage existing knowledge, tools, and the latest integrated development environments for agent development.(JADEX. . . , 2015b, p.4)

Agents in Jadex can be either defined declarative using an XML file, or by creating a Java class with the necessary fields. No Inheritance is required, instead, a set of annotations tell the Jadex Engine how to interpret the agent class' various fields and methods. This approach allows the framework to be loosely coupled to the agent implementation. A pure java approach is more verbose, although it provides more flexibility.

```
1  @Agent
2  public class MyAgentBDI
3  {
4  ...
5  }
6
```

Figure 1: Example of an empty Jadex agent.

Agents in Jadex are simple Java classes with an *@Agent* annotation and the BDI postfix. This serves as metadata for the BDI engine which informs what it is and how to operate it. There are multiple instances of annotations being used over a strict object oriented hierarchy in Jadex, including *@Plan, @Belief, and @Goal*, as well as many others. Most of the main annotations also have others associated with relevant methods and objects, such as *@PlanBody*.(JADEX. . . , 2015a, p.5, p.6, p.11)

Following Jadex's loosely coupled approach, beliefs are defined at the programmer's discretion by using the *@Belief* annotation on an object property or method. Jadex does not put restrictions on the type of data a belief is represented with, it can be any java object, and can be masked with get and set methods respectively. Jadex uses standard Java conventions to infer all the necessary information. Belief's can then be referenced by using identifier strings. By allowing any sort of object to function as a belief, a Jadex agent can more easily model complex environment knowledge, at the cost of not having any logic-based inference mechanisms integrated into the system. (JADEX. . . , 2015b, p.5)

Jadex differs from the BDI model by making goals a concrete mechanism, treated as objects with several attributes that include information about the current state of the goal. To track whether a goal is currently a desire or an intention, Jadex has three states associated with them: option, active and suspended. Plans can also be of various types: perform, achieve, query and maintain. Perform goals do not have expected results, but rather an action that has to be carried out. Achieve goals define an objective to be achieved, regardless of the means. Queries express the need for information gathering, and become active when said information is potentially available. Lastly, maintenance goals are conditions that must be kept through specified periods of time, or until maintaining said conditions is no longer important to the agent.(JADEX. . . , 2015b, p.5, p.6)

5.2 Jason

Jason is a Java based implementation of the AgentSpeak interpreter. Rather than being a framework to facilitate BDI agents in Java, AgentSpeak is an extension of the logic programming paradigm for BDI agents. (JASON. . . , 2007, p.2) A BDI agent in Jason is simply a set of base beliefs and plans, as seen in figure 2.

```
1 +pos(r1,X1,Y1) : checking(slots) & not garbage(r1) (p1)
2   <- next(slot).
3 +garbage(r1) : checking(slots) (p2)
4   <- !stop(check);
5    !take(garb,r2);
6    !continue(check).
7
```

Figure 2: Excerpt of Jason's AgentSpeak plans. (JASON. . . , 2007, p.5)

The language is designed for BDI agents, which means there is none of the verbosity present in Jadex. Instead, Jason agents use a syntax similar to that of Prolog, with the actions themselves coded in Java on a backend class that defines what the agent actually does in java code. The advantage of this approach is the Jason interpreter has full control over the execution, as well as a clean, simple syntax for the developer to interact with. Jason comes with its own IDE suite for agent development, designed around a workflow where designing BDI agents is front and center.

5.3 Discussion

Using Jason comes with plenty of benefits. Making BDI agents is simple and straightforward. However when it comes to the creation of games, the compromises in its design are clearly apparent. In order for AgentSpeak agents to work the Jason interpreter

must be functioning in the application or through a separate service. Jason agents are restricted to a Java environment, which requires the system to have a Java virtual machine. All too often these are compromises that cannot be made, such as game consoles where developers are restricted to work with a specific SDK. Distribution is also a problem. Popular engines such as Unity, Unreal and Cryengine, for example, are not made in Java, thus a lot of development effort would be required just to get the agents up and running in such environments. In many cases, not worth the time savings of having an agent oriented language to program for.

Jadex's vision and design reflects an engineering approach to BDI Agents, with the main focus being the ability to leverage existing tools, developer experience and interoperability. The inherent drawback to this approach is added verbosity as Java was not designed to easily represent the agent-oriented paradigm, being object-oriented instead. It is also not just a BDI framework, as that is only one of the many components integrated into the framework, built upon their own Active Component middleware for distributed computing.

Jadex's strength clearly lies on its ecosystem. It does not require a separate IDE for agent development, seamlessly integrates with any Java code, and runs on top of a robust distributed computing middleware. That said, it is not a catchall solution. There are a wide variety of languages without BDI frameworks, and Jadex's dependency on Java and its virtual machine can make it inviable in projects for many embedded devices and native client applications.

6 Minecraft

Minecraft is an open-world video game first released in 2009 as an alpha version, gradually updated since. It has become widely popular, with nearly 20 million copies sold on the PC platform alone (MINECRAFT, 2015). The game's premise is simple: the player is tossed into a procedurally generated world and left to survive against monsters that appear during night time. The entire landscape is made of blocks, which the player can harvest to use in construction or crafting of various tools and weapons. It is then up to the player to find ways to survive and use the blocks to construct anything they can imagine.

Figure 3: Screenshot of Minecraft, an open world made of blocks.

6.1 Game Objectives

One of the appealing aspects of the game is its open-ended nature. There are no actual objectives besides surviving, and it is up to the players to figure out how to exploit the environment they are thrown into to maintain their character's food supply and stay safe from monsters. In order to obtain food, players must hunt, gather, and farm. To do so safely, they must build bases and dig mines to obtain useful minerals for better tools.

That is the game as it comes. Fans of the game have developed a variety of modifications to add machinery, electricity, automation, magic, and many other contraptions.

One such modification is ComputerCraft, which is explained in section 6.6.

6.2 World Structure

Figure 4: Screenshot of Minecraft, procedurally generated forest landscape. Modifications made by fans add new environments with giant trees, and changes to the UI such as mini maps.

The world of Minecraft is made entirely out of various blocks stacked together on what is effectively an infinite horizontal plane. Blocks can be rocks, dirt, wood, leaves, grass, minerals, water, lava and so on. With those blocks, a procedural generator creates a world with oceans, grasslands, mountains, forests, caves, and other natural formations. In other words, the Minecraft world is a block-based emulation with environments similar to those found on Earth. Those blocks can be picked and then used by players to build and craft tools, or simply make up places to explore.

In order to provide a seamless, near infinite world, it is segmented into "chunks" of 16 by 16 by 256 blocks, making up 65536 blocks total for each chunk. A configuration-defined amount of those are loaded around the players as they move, unloading them once they are out of sight. The sea level is at a constant height of 60 blocks, underneath being mostly stone with veins of minerals, dirt, and caves spread along. At the very bottom, heights 1 to 5, is the bedrock, unbreakable blocks that are impossible to get past through without cheating or modifying the game. It keeps players and blocks from falling down indefinitely. On the other end, mountains well over a hundred blocks tall are placed around the world, with clouds made of transparent, pass-through blocks looming under their peaks. While it

Figure 5: Screenshot of Minecraft, chunks are 16x16x256 grids of blocks that are loaded progressively as the player travels and unloaded as they fade from view.

is possible for a player to fly higher than the height limit of 256, blocks cannot be placed any further.

Plenty of blocks are "dynamic," with animations and scripts that perform given functions. Workbenches and Furnaces allow players to craft, while "redstone" circuits carry signals that trigger contraptions at the flip of a lever or, in case of traps, when an entity steps on a pressure plate.

6.3 World Entities

To inhabit the world, a wide variety of entities are randomly generated. Besides the player, animals inhabit the surface in packs, reproducing and jumping about. At night, monsters spawn in dark areas, such as skeletons, spiders, zombies, and the iconic creeper. They will hunt down the player on sight, and are very dangerous, specially so to the frequently poorly equipped player. Unlike blocks, entities have free range of motion in all directions, but cannot pass through blocks.

6.4 Internal Structure

Minecraft is implemented in the Java programming language, and it is available in most platforms, including Windows, Linux, Mac OS X, and even recent game consoles such as the Playstation 4 and the XBox One. Since its inception, communities have been

Figure 6: Screenshot of Minecraft, creatures roam the land along with the player, pigs are an important source of food.

forming for the express purpose of exploiting the game's modular nature to modify the game and add features, or changing gameplay mechanics. Modifications or Mods would include anything from new textures for the game's assets to complex new mechanics like the ability to craft flying vehicles to traverse the landscape.

This flexibility makes it well suited for academic research, providing a virtual world that can be easily interacted through modification.

6.5 Challenges for Artificial Intelligence

- Procedurally generated landscapes provide vast, varied worlds created by the game's algorithm that are ideal sandboxes
- Grid-based path finding is relatively inexpensive and simple
- There are many challenges. The environment is dynamic and sometimes unpredictable when entities and dynamic blocks are involved. Deadly hazards exist thorough the environment, particularly lava
- A variety of mundane tasks often required from the player provide several problems of automation, such as construction, mining and farming.

6.6 ComputerCraft

ComputerCraft is a Minecraft modification that implements various simulated computers that allow Lua scripts to interact with the environment. This ranges from actual computers with in-game command interfaces to displays and, of particular interest, programmable turtles that can traverse and interact with the environment, allowing the player to automate menial tasks such as mining and gathering of rare materials. Turtles also have their own inventory, allowing them to collect and even craft their own items. All ComputerCraft computers are programmable using the Lua programming language and an in-game editor and command-line interface. Scripts can be downloaded from the internet by uploading them to <www.pastebin.com> and using an integrated program to download them into the turtle's internal file system. (COMPUTERCRAFT, 2015)

6.6.1 Turtles

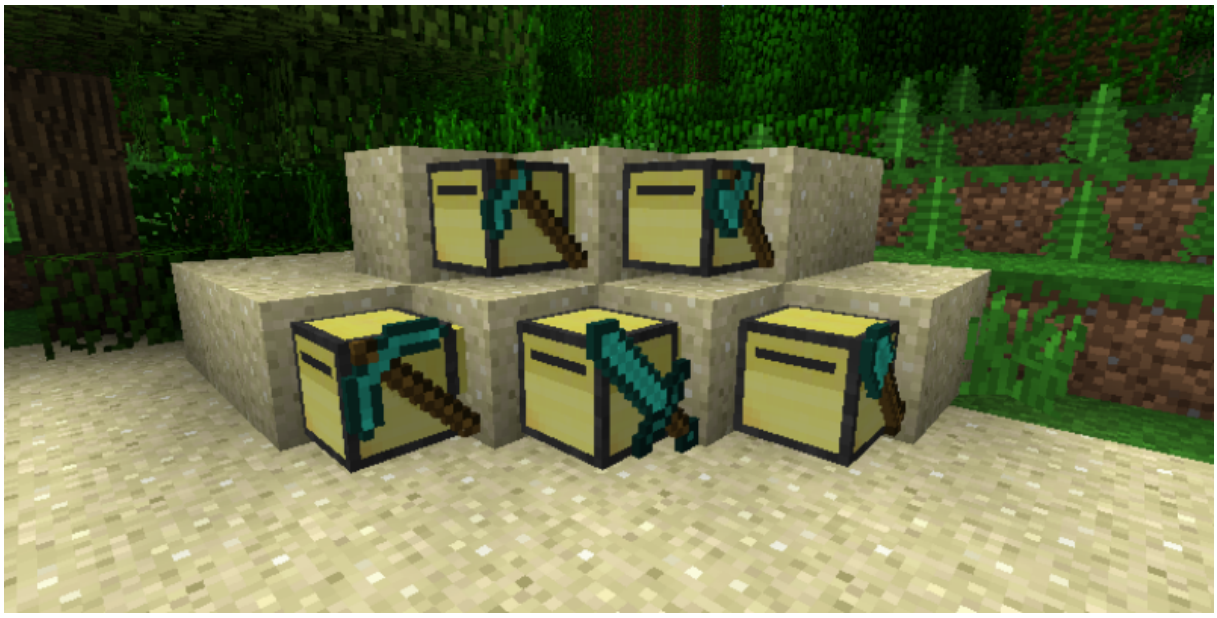

Figure 7: Screenshot of turtles, multipurpose robots that are controlled by Lua scripts.

Turtles are programmable dynamic blocks. They can be crafted by the played and placed into the world. Being computers, they have an in-game terminal, accept commands and run programs. They use fuel to operate, which can be either coal or lava that must be placed in their environment.

Turtles have simplistic range of motion. They can move forward, up or down, rotate left and right, pick blocks in front, below or above them, or place them from their environment. They can also craft items, and when equipped with a wireless interface, they can communicate with other computers through a virtual network. As for sensors, they

are very limited. Turtles can only see the blocks directly in front, above and below them, requiring to turn in order to inspect blocks on their sides and back. Turtles are also not affected by gravity, so they can float up in the air.

Scripts execute in parallel with the game, with a fixed length of time required for each action. Asynchronous execution is possible through the Lua multithreading API, although unsupported and largely untested.

6.7 Turtles and the Environment.

From the perspective of Agent Theory and turtles, the environment in Minecraft has the following characteristics:

- Not Accessible: turtles can only see what is directly in front, below and above them. They are blind to non-block entities such as the player or the various monsters that appear, although it is worth noting that as a block, the turtle pushes back any entities as it moves, potentially killing them in the process

- Nondeterministic: despite being impervious to having its movement blocked by entities, such entities can alter the environment by removing or placing blocks, altering the state of the environment dynamically. Multiple turtles can interfere with each-other, and the state of the world changes even while the agent reasons its next action. Information gathered may become outdated during said time, and the limited scope of perception adds even more uncertainty

- Dynamic: the game runs continuously even as the turtles reason their actions, as they work in a separate thread. The player sees a continuous progression of time even if the turtle is stuck thinking for several minutes. This makes the environment inherently dynamic as anything can happen within the time the agent spends reasoning

- Discrete: despite its dynamic, nondeterministic nature, the Minecraft world perceived by the agent is conceived by a discrete array of blocks. Movements, actions, and perceived data are all discrete. The turtles can only move one block at a time, and no less than a block, they can gather exactly one block at a time, and each action performed takes a fixed length of time. For the player and other entities, the environment is indeed continuous, with range of movement and progression of time, but to the turtles it is nothing more than a turn based environment from its point of view.

6.8 Discussion

When I first looked for possible projects involving AI, Minecraft immediately came to mind. Its open, blocky, simple world is both fascinating and full of opportunity, specially for artificial intelligence. It is a world where the complications of robotics and image processing are not present, so I can explore AI, planning in particular, without having to worry about solving other problems that are entire projects on their own.

Rather than creating a test environment, developing AI for games in a game that is particularly suited for it makes a better choice.

7 Framework Specification

7.1 Design

Upon examining the various frameworks already available and considering the purpose and audience the frameworks intends to reach, the main requirements can be outlined. The framework intends to be:

- Platform agnostic: With the large variety of languages used to create games for an equally varied number of target platforms and devices, it is counterproductive to design a framework that can only be truly realized in one programming language or hardware platform
- Simple: The target audience does not require extensive programming knowledge, basic usage should be as simple as following a recipe, while providing an effective solution to a single problem: BDI plan selection. It should not require any dependencies to use, as many development environments cannot use third party libraries.
- Well performing: performance is essential to video games, which are always pushing hardware further with better graphics and complex physics. A framework that does not perform sufficiently well will not get used.
- Adaptable: Games use a vast variety of game mechanics, architecture, components, dependencies, etc. Therefore the framework has to be able to adapt to their needs and style, rather than the other way around.

Granted, no matter how ambitious a framework aims to be, it will never be for everyone. There are too many edge cases to even cover them all. What can be done is to cast the largest net possible and cover the requirements of most would-be users. The programming language features used in the algorithm, as well as its general execution work-flow and structure are the main points of consideration of this design.

7.1.1 Programming Language

Programming languages are the main tool developers have at their disposal. As stated before, it cannot be assumed that developers will be using one given language, or would be willing to switch over simply to be able to use a small BDI framework. The Framework has to easily adapt to whatever language it is written in or ported to. It can be assumed that if the framework design is simple enough to implement, clearly laid out

and with worthwhile benefits to the project, developers may find it worth their resources to borrow the design into language of their choosing. Therefore, making use of unique features, like C#'s LinQ or Java's anonymous classes is out of question. However it can be safely assumed most procedural and object oriented languages have these features, or equivalent:

- Functions
- Objects or structures
- Functions
- Lists
- Loops

C, C++, C#, Java, JavaScript, Ruby, Python, PHP, Lua all have these features or equivalent. C requires the most workarounds for various things that are not provided with syntactic sugar, while interpreted languages like JavaScript and Lua have all these features built in. Currently, JavaScript is the main programming language for Unity, a popular game engine that has gained a lot of traction in the indie market.

ComputerCraft, the Minecraft modification that adds programmable robots to the game uses Lua. Lua is also very similar to JavaScript, the main programming language for Unity and HTML5. Therefore, it is the selected language for a proof of concept implementation of the framework.

7.1.2 Execution Workflow

The runtime execution of games depends on many factors: how the game engine is designed, what support it has for multithreading, what design patterns the developers follow, etc. Therefore, the framework cannot expect to dictate how the AI will execute and when. It is a subordinate of the system it is integrated into. As long as it brings the desired result, its conventions can be broken at discretion.

How many atomic actions an agent can take or schedule during a single reasoning cycle, or plan selection and execution, is a design consideration that can affect various elements of the system as a whole. A plan that takes over an execution thread to do a long list of steps can cause serious problems when higher priority tasks like screen redrawing have to be done in a timely fashion. It makes sense to limit an Agent's thought process to one action per reasoning cycle. However, there are many cases where the developer feels it is appropriate to make a different approach. Rather than making it impossible to approach

edge cases this way, for the sake of adaptability, the approach taken is to let the developer ultimately decide, using conventions and examples to suggest the intended form of use.

Lastly, Parallelism and multithreading is a problem that a simplistic, platform agnostic framework cannot solve. It is assumed that the agent's object structure is either thread safe or not shared by multiple threads, and that one agent engine manages exactly one agent, and that agent is only managed by that one engine. The engine executes reasoning cycles step by step, which means as long as plans are sensibly designed by the developer, a single step will not hog a thread execution for long. Engines can be set to run in their own, separate threads as well, or have multiple engines share a thread by each taking a step in order. It is all up to the developer to accommodate the engine into their system in whichever way they see fit.

7.1.3 Structure

The framework design is split into two sections. One is developed by the user: the agent object, a structure that contains definitions for all the beliefs, desires and intentions. Depending on the language, it might simply have to follow a design specification, or inherit a base class. Like controllers in an MVC framework, it's where most of the programming will take place in the form of plans.

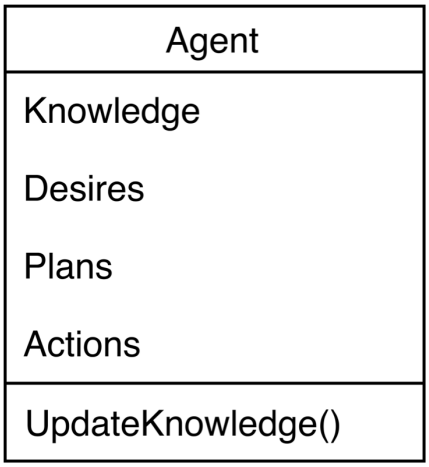

Figure 8: Agent object structure.

The method, *UpdateKnowledge()* exists for passive belief perception such as vision, location, etc. This way plans are guaranteed to always have up to date information to work with. The other elements of the Agent object are the basic components of a BDI agent: a knowledge base, a set of desires to fulfill, a set of plans to fulfill them, and actions it can perform. Actions may or may not be part of the agent object or an external source, as only the agent itself will execute such.

Next is the agent engine, an object that manages the BDI plan selection. The

engine has several functions, but it comes down to one basic set of tasks, the reasoning cycle:

1. For each desire, calculate the priorities of each using the knowledge base
2. Select the most appropriate plan to fulfill the desire with the highest priority
3. If there is no such plan, repeat step 2 with the next highest desire, if any
4. Execute the selected plan, if any.

Without the whole picture, it seems a a first glance that the engine doesn't do much in the first place, however the agent itself plays as much part in the plan selection process as the engine itself. There are several nuances the engine takes into consideration during plan selection. The priority of a desire can be an arbitrary number, or a function. A function allows arbitrarily complex calculation and fine tuning, from a discrete range of numbers to a continuous curve. Plans can have preconditions as well as a fitness value that determines how it measures against other plans that fulfill the same desire, which can also be functions and much like desires add wide range of elements into consideration. Certain desires can be made unattainable by purposely not making the plan change the knowledge base after completing it, allowing for maintenance goals that are always live. Plans can schedule sub plans by setting flags in the knowledge base that trigger them. By allowing as much customization as possible, with sensible defaults and various conventions, the framework comes together into a fully featured BDI suite that is adaptable and trivial to implement, while retaining a simple agent implementation within reason. Needless to say an agent in this framework will certainly be more verbose than one made in Jason's agent oriented programming language, depending on the programming language used, but the benefits earned from it are an acceptable tradeoff to projects that cannot easily integrate Jason and other frameworks into their system.

7.2 Architecture

The framework components each follow different paradigms. An agent object is in many ways declarative from the perspective of the engine, a detailed description of what it wants and how it intends to obtain it. Therefore we'll look at the agent object in a declarative manner, rather than algorithmic. The plans in question, or much of the actual algorithms involved for that matter are of no interest to the engine, which only needs to know certain key pieces of information. Meanwhile, the relevant parts of the engine are the algorithms that make plan selection using the agent data structure.

7.2.1 Condition Strings

Condition strings are critical for the integration of the engine and the agent object. They are a form of metadata that serves various purposes. Condition strings have the following formal definition using the Backus-Naur form:

```
1 expression ::= item | item and expression
2 item ::= nflag | flag
3 nflag ::= not flag
4 flag ::= [A-Z1-9_]+
5 and ::= &
6 not ::= !
```

Spaces are ignored, and these strings are later decomposed into key-value dictionaries and used to perform checks such as if a Desire has its condition met, if a plan fulfills a Desire's condition, or if a plan's preconditions are met. The criteria for A to fulfill B is that every fact in B that is true must be true in A, and every fact in B that is false must be either true in A, or not be in A. This is due to the assumption that unknown facts are false. When these conditions are met, then it is considered that A fulfills B, and therefore if A is a plan and B is a desire, A qualifies as a plan that may result in B being fulfilled.

7.2.2 Agent Object

The agent object has three components: *Knowledge*, *Desires* and *Plans* are each key-value dictionary structures.

Out of these three, Knowledge is loosely defined as an object that holds all of the agent's beliefs. Depending on the language used that will be all that's required, but on static typed languages either a generic workaround, or a strongly defined Knowledge class will be required. As far as the engine goes, the only knowledge that it requires are boolean flags which are the realization of desires, and other things like preconditions or other binary data. One way to approach the static limitation in languages like C++/Java is to define an interface or base class that has one property which is a Key-Value dictionary of boolean flags. This way, the engine can interact with the knowledge base only at the level which it requires to, while leaving the developer with freedom to do anything else with subclasses or interface implementations.

Desires are more clearly laid out. Desires is a key-value dictionary, but of a specific type: Desire. Since only Desire objects have any purpose being in the Desires dictionary, there is no need for workarounds for static languages. A Desire is composed of the following:

- Condition string: a condition string that represents all the conditions necessary for the desire to be considered fulfilled

- Priority: a number of function that returns a number that specifies how important the desire is relative to others. Depending on the implementation, priority might take the agent object as a parameter, as to query the knowledge base of the agent for information.

Much like Desires, plans are very similar in their structure, except they have more information:

- Precondition string: a condition string that represent the current state of the knowledge base required for the plan to be viable.
- Goal string: the promised results of the plan. While it might fail its execution, the goal tells the engine what the state of the knowledge base will be should the plan succeed. For a plan to qualify to fulfill a desire, its goal must fulfill the desire's condition
- Fitness: an optional function that returns a number specifying how fit the plan is to satisfy its goal under given circumstances. Depending on the implementation it might take a reference to the agent as an argument
- Execute: the plan body. When a plan is selected this method will be called, with references to the agent and the engine passed as parameters
- State: A plan may have its own state properties, at the discretion of the developer.

Besides those three main structures, an agent has a method called *UpdateKnowledge()*, which gets called once before every reasoning cycle. It serves as a passive belief processor for information that is expected to be constantly updated, such as vision or hearing. The purpose of *UpdateKnowledge* is not to process the information itself, but to store any relevant information into the knowledge structure. For example, a robot's video and audio feed might be decoded and processed by a neural network in a completely different thread or process, then fetched by the *UpdateKnowledge()* method and added to the belief base. This Ensures that knowledge is only changed by the same thread executing the thought process, therefore maintaining thread safety in the space of plan execution, as the entire process becomes single threaded and isolated. Plans can rely on the fact that no knowledge changes while they execute, unless they themselves make a call to this method.

Lastly, the agent must have a reference to all the agent actions it can execute, as plans are given an instance of the agent to work with. This does not mean that agent actions have to be programmed within the agent object, as a reference to an "agent body" will suffice in most cases.

7.2.3 Agent Engine

In order to make use of all the information declared in the agent object, the agent engine has to go through it all and do a reasoning cycle that will include partial or complete execution of a plan as well. There are several algorithms that are needed to make this happen. This section will cover them all in bottom up order.

7.2.3.1 Parse Condition String

- Argument: a condition string
- Returns: a string key, boolean value dictionary, empty dictionary if string is nil
- Purpose: parse a human readable set of conditions into a dataset
- Process: remove spaces from the string, split it by using the *'&'* operator as separator. For each sub-string, if the first character is *'!'*, then store the sub-string in the dictionary as a key with *false* as a value. Otherwise, the value is true. Return the dictionary once all sub-strings have been evaluated.

7.2.3.2 Evaluate Conditions

- Argument: two string key, boolean value dictionaries. One is the expected condition, the other is the evaluating condition
- Returns: true or false
- Purpose: will return true if the *evaluating* boolean set meets the *expected* set
- Process: ensure neither the expected set nor the evaluating set is null or empty, otherwise the result is automatically true or false respectively, and in that order. Then for each key value pair, if any value in evaluating is not the same as expected then the result is false, otherwise true. For the purpose of this comparison, nil and false are considered equivalent.

7.2.3.3 Check Desire

- Argument: desire identifier string
- Returns: a desire object with name, condition string and the calculated priority value
- Purpose: checks a desire's current priority value, returning a different object with the value, name and its condition string.
- Process: find the desire object in the agent's Desires collection. Evaluate the priority value and create the resulting object.

7.2.3.4 Check Desires

- Argument: none
- Returns: a desire object with name, condition string and the calculated priority value
- Purpose: evaluates the entire collection of desires, returning the one with the highest priority.
- Process: iterate through every desire in the agent's Desires collection. Evaluate the condition string of the desire against the knowledge base, ensuring it has not been met. Choose the highest scoring desire that remains unfulfilled

7.2.3.5 Set Intent

- Argument: a chosen desire
- Returns: a return value from the plan, or a failure value if no matching plan is found
- Purpose: finds the most fitting plan to fulfill a desire and executes it
- Process: iterate through every plan in the agent's Plans collection. Ensure the plan's goal fulfills the desire with evaluate condition, and that evaluating the current knowledge base against the preconditions yields true. From the valid plans that fit this criteria, pick the one with the highest fitness value after evaluation and execute it.

7.2.3.6 Step

- Argument: none
- Returns: a return value from the plan, or failure if no matching desire or plan is found
- Purpose: performs a reasoning cycle
- Process: call the agent's *UpdateKnowledge()* method. Find the highest priority desire with Check Desires and call Set Intent with that as its argument, then return the plan's execution result.

7.3 Implementation

The following sample implementation is made in Lua for the purposes of using ComputerCraft Turtles in Minecraft as proof of concept. Both the engine and the agent

objects use a meta-programming construct called Class, made as a means of adding the basic object oriented construct into the language for ease of use:

```
Class = {
  new = function(self, o)
    o = o or {}
    setmetatable(o, self)
    self.__index = self
    return o
  end;
}
```

This is simply a method that adds a basic constructor. Lua does not have object oriented programming, instead using "tables" as a replacement of objects. The Class table has a *new()* function that takes another table as an argument, injecting all the properties of the current table (Class) into it. This can be used for both inheritance and instancing. Using this method, an agent object is defined as follows.

```
Agent = Class:new({
  --contents
})

a = Agent:new()
```

Agent becomes an instance of a class, inheriting the *new()* method from it as well as defining its own. On the other hand, *'a'* is an instance of the class Agent. The difference is merely a convention, as Agent is also an instance of Class. As the inner structures of Agent do not require to be instanced, they are not classes and objects, but regular tables.

```
knowledge = {
  flag_a = true;
  flag_b = false;
  value_a = 3;
};
```

Due to Lua's dynamic nature, knowledge does not require any extra work to fit the model. Entries can be added, changed and removed from a table at any time, therefore a simple table suffices as the knowledge data structure. Same goes for Desires and Plans.

```
desires = {
  a = {
    conditions = "!flag_a";
    priority = function(self, agent)
      return agent.knowledge.value_a * 2
    end;
  };
```

```
8
9   b = { conditions = "flag_b", priority = 5 };
10 };
```

The desire *'a'* has the precondition that *flag_a* must be false. It's priority depends on the *value_a* stored in the agent's knowledge. The higher it is, the more important it becomes to fulfill *'a'*. If the condition never becomes true, it can be said to be a maintenance desire, as it remains forever unfulfilled, only becoming a priority if the value it monitors increases to a point at which it requires attention. Another, simpler form to express a desire is to use a number as the priority value, as is the case with *'b'*. As long as *flag_b* is false, the agent will have a priority of 5 to fulfill this desire, and otherwise ignore it.

```
1  plans = {
2    plan_a = {
3      preconditions = "flag_a";
4      goals = "!flag_a";
5      execute = function(self, agent, engine)
6        --do something that reduces value_a
7      end;
8    };
9
10   plan_b_a = {
11     preconditions = nil;
12     goals = "flag_b";
13     fitness = function(self, agent)
14       return agent.knowledge.value_a
15     end;
16     execute =  function(self, agent, engine)
17       --do something to fulfill b
18       agent.knowledge.flag_b = true
19     end;
20   };
21
22   blan_b_b = {
23     preconditions = nil;
24     goals = "flag_b";
25     fitness = 2;
26     execute = function(self, agent, engine)
27       --do something to fulfill b
28         agent.knowledge.flag_b = true
29     end;
30   }
31 }
```

Plans can be defined in various ways. *plan_a*, for example, does not have a fitness function or value. This means it is automatically chosen if it's a viable plan to satisfy the

current desire. To satisfy *'b'*, however, there are two different plans. One has a fitness value of 2, while the other depends on the value of *value_a* as a function. This means that depending on the circumstances, all three different plans can be chosen depending on what the agent knows about its current situation. Since *flag_a* is never set to false, the desire always remains active. As long as *value_a* remains higher than the priority of other desires, the engine will seek to fulfill this desire. Once *'b'* is fulfilled, the engine will repeatedly attempt to fulfill *'a'*.

```
1 update_knowledge = function(self)
2   --update the knowledge base each reasoning cycle
3 end;
```

Finally, *update_knowledge()* is the method that by convention, the engine will use to tell the agent to update its belief before a reasoning cycle occurs. With all four elements, the agent has all the necessary components for the BDI engine to function.

8 Methodology

Chapter 6 explains in detail various aspects of Minecraft as an environment and ComputerCraft turtles as agents. As previously mentioned turtles can only see whichever block is directly ahead, above or below them. Having information at its disposal is crucial for an agent to be able to plan and perform tasks of anything but the most basic complexities. For example, in order to chop down trees, the turtle has to know where they are. Exploration with such a limiting field of vision is highly time consuming, as a block in real life equivalent is a mere cubic meter in volume. For all intents and purposes, turtles are essentially blind.

Therefore it is of utmost importance for turtles to remember what little information they have access to and to work assuming as little as possible. Chopping trees is not particularly hard for turtles to do, as trees are block structures of predictable shape. The problem lays in finding them, but this is not a problem if the turtle itself plants them.

Figure 9: Mining all the way down to the bedrock can be quite tedious and time consuming, depending on the size of the shaft.

One iconic part of the game, as hinted by its name, is mining. The player requires various resources only found underground to progress through the game, and mining shafts all the way down to the world's bedrock is one of the most time consuming tasks, and a good target for automation. Another simple task is to travel from one location to another. For more complex tasks to be even possible, movement is involved, so it makes sense for it

to be a task in and on itself.

Additionally, since this is a proof of concept for a BDI agent, it must be *proactive*. Which means it must decide on its own to move about or dig a mine. It would be ideal to communicate in some way with the agent, but that is not possible. Therefore, *something* must incite the desire on the agent to dig a mine, or to move from one location to another.

For simplicity's sake, I made it so upon seeing a *chest*, the agent will suddenly desire to cave a mine in that location, of a 4x4 size, and down to the 50th vertical layer of blocks, or the y50 coordinate. This means at sea level the agent will dig down somewhere between ten to fifteen blocks of depth, well more than enough to see a pattern.

In order to test movement, a maze was prepared using bedrock, a type of block that is unbreakable by anything but a player using creative mode (which for the purpose of this test, I am). The maze can be seen in figure 10. Five places were selected as points the turtle would have to travel to at random. To test the agent's adaptability, blocks were placed or removed at different locations as to encourage it to look for alternate routes.

Figure 10: A maze made of bedrock, a test ground for the traveling plan and the A* algorithm.

8.1 Development and Experiment

To develop the turtle agent, there were three requirements needed beforehand:

1. BDI framework.

2. A* Algorithm

3. Some kind of memory structure.

The A* algorithm came with its own set of requirements, as Lua does not come with premade lists or vectors, which had to be implemented beforehand as I needed more than concatenated references. Support for sorting, iterators, copying, and so on had to be added and tested.

The memory structure needed to store block locations as the agent found them made use of Lua's hash tables, adding safety checks to avoid null reference exceptions. Two methods were used for this, one that safely writes to the table, and another that safely retrieves data from any arbitrary coordinate if there is any. The object in question is a multidimensional, dynamic table that can grow to any size, as long as there is memory. Since it is unlikely that a turtle will ever explore enough for caching to be necessary, this proved sufficient and most importantly fast.

Then came the actual agent development. With the framework developed and functional I thought of the tasks from a BDI perspective. The most vital desire is to move to its intended location, nothing gets done otherwise, so that was the starting point.

However, moving from one place to another cannot be done without knowing which direction the turtle is facing. The virtual GPS provides location, but not direction. Finding that direction had to be the highest priority task.

```
1 get_direction = {
2   conditions = "direction_known";
3   priority = 150;
4 };
5
```

Figure 11: Desire to find the direction the agent is facing. It has the highest priority of all, thus is always the first thing the agent does. Once the direction is found, the desire is satisfied and no longer chosen.

The plan was simple, to move in any direction possible, dig a block if necessary and to infer the direction by comparing the coordinates of the initial and final position. This means that for proper calibration, the turtle always moves from its starting position one step.

Once direction is known, the next highest desire is to *be* at its target location, if any. This is a maintenance desire, which means as long as there is a target, the turtle will move to it. And if there isn't, the desire remains fulfilled, as seen in figure 12.

The actual traveling algorithm is quite a bit more complex as there are many conditions to account for. To other plans, however, traveling to any location is as simple

```
1  get_to_target = {
2  conditions = "target_maint";
3  priority = function(self, agent)
4  if agent.knowledge.target ~= nil then return 100 else return 0 end
5  end;
6  };
7
```

Figure 12: *target_maint* never gets changed to true, therefore the desire always remains unfulfilled. It's priority changes from 0 to 100 if there's a target to reach, coming up higher on the list than other desires.

as setting a target in the knowledge base and returning control to the BDI framework. Once movement is complete, if the plan is still a priority, it will get invoked again and then it can resume where it left off.

In order to test the traveling plan and the A* algorithm and its heuristics, I just created a plan and corresponding desire: if there are no other desires with higher priority, go to one of five random locations in the maze.

```
1  make_up_target = {
2    preconditions = nil;
3    goals = "make_target_maint";
4
5    execute = function(self, agent, engine)
6      local r = math.random(5)
7      if r == 1 then
8        agent.knowledge.target = Vector:new({x = 360, z = -450, y = 63})
9      elseif r == 2 then
10       agent.knowledge.target = Vector:new({x = 359, z = -448, y = 63})
11     elseif r == 3  then
12       agent.knowledge.target = Vector:new({x = 357, z = -448, y = 63})
13     elseif r == 4  then
14       agent.knowledge.target = Vector:new({x = 360, z = -446, y = 63})
15     elseif r == 5 then
16       agent.knowledge.target = Vector:new({x = 353, z = -448, y = 63})
17     end
18     return "SUCCESS"
19   end;
20 };
21
```

Figure 13: *make_target_maint* is the lowest priority desire the agent has. Whenever the agent is idle, it will set a target to one of five predefined locations inside the maze.

The idle desire had some unintended consequences, as turtles that finished digging mines went back into the maze as they were done, successfully skirting around the bedrock walls by digging underground to reach inside and breaking the glass window I had mate to observe.

Meanwhile, the desire to mine shafts was set to only trigger whenever a turtle sees a treasure chest in front of it. It would then calculate the coordinates of all the blocks to be mined, sort them by distance to the agent, and set each one as targets one by one. The plan delegating scheme worked perfectly, the resulting behavior identical to that of the less flexible, preprogrammed movements for shaft digging. Therefore, with some minor modifications, this same algorithm can be used to have the turtle build complex structures by providing some kind of blueprint.

Placing one chest within the maze and having a turtle run into it caused the turtle to attempt to dig the four by four shaft inside the maze, only to wind up failing due to bedrock being unbreakable. It would then resume its idle task of traveling between random locations of the maze.

The tests made were sufficient to conclude that the framework is fully functional, along with the suite of unit tests made to check each component of it behaved as expected.

8.2 Challenges

In the process of creating a BDI proof of concept agent, many challenges were met, some directly related to the game, the ComputerCraft modification, or the Lua language itself. Delivering the source code into the turtle was a challenge in and on itself. Without a viable method of delivering multiple files automatically and with the 25 uploads per day cap of <www.pastebin.com>, a separate program needed to be developed to write all the dependencies down into a single file. This required significant time and effort, but it was completed regardless. Eventually the upload limit was bypassed by finding the turtle's file system folder inside the game's world folder, using symbolic links, a special type of shortcut, to direct it to the code location.

The next major problem was geolocation. Turtles have no means of knowing where they are by themselves, information that is crucial for path finding and any task that requires movement. ComputerCraft offers an emulated GPS system that requires multiple computers in different locations to triangulate coordinates. However, it is not reliable. The GPS system loses accuracy and outright stops functioning as distance from the array increases. Moreover, even at close distances, GPS coordinates are often wrong, causing problems for the agent's memory and other unpredictable issues. This issue was worked around by ensuring the GPS coordinates are indeed correct (not decimal) with repeated tries, or fatal failure if the turtle is out of range.

While the BDI framework itself required little more than a hundred lines of code, the Lua programming language and the precarious development environment significantly slowed down progress. Lists and vectors had to be implemented and tested as they were necessary to implement the A* algorithm, the language's arguably extreme flexibility

Figure 14: GPS array with four computers.

becoming a liability as small mistakes like capitalization caused major, hard to trace bugs with bizarre consequences. One egregious example was accidentally modifying List inside an iterator, creating an infinite loop in only specific circumstances that did not arise during initial tests. This is not uncommon in dynamic programming languages, but the slow debugging process of testing things in Minecraft and the bugs that only appeared in certain edge cases were extremely difficult to track down.

9 Conclusions

Due to the nature of this project, just having a function agent as proof of concept is not enough. Designing a framework is not just about function or effectiveness, which can be measured through observation, but also about subjective factors like ease of use, or simply, how it feels to work with it.

After all, it is pointless to make a framework with great functionality that is not used. Ultimately, a framework's end user is the developer, and as a product, it has to appeal to them. Otherwise, something else will. Therefore, the closing thoughts of this project are, fundamentally, subjective from the point of view of myself as a developer.

9.1 Observations

Through my time developing the framework as a user, rather than its creator, I found myself wondering why a framework was needed in the first place. BDI is a very simple concept. It all comes down to priorities and strategies, but the framework, despite its simplicity guided me to think in terms of tasks and their importance at any given time. The concept of priority becomes central to the entire process, rather than tucked away in a deep chain of ifs and elses.

In that regard, the framework does what it is supposed to, and the very workflow I used was as much part of the framework as the hundred or so lines of code that made it. The agent object structure, and the way it is interacted with, it is all as much part of the specification as the actual code.

That said, there are questionable elements that could use refinement. The excessive use of convention over configuration makes it all too easy for someone to misuse the framework and get unintended results. Just like a monolithic class in object oriented programming, plans can also be monolitic. They can do far more than they are supposed to, rather than being small, self contained, and one step at a time affairs. Nothing stops a plan for going on endlessly without ever returning control to the framework, executing action after action. That violates the core concept of BDI, that intentions can change at any time to react to emerging problems. The framework trusts the developer with the responsibility of making plans that only execute few actions per execution. Should it be, ideally, one action per cycle? Absolutely. That said, sometimes, two or three is not that bad. I cannot forget the audience I'm targeting, one in which getting things working is more important than following the concepts to the letter.

9.2 Future Work

The framework's future work lays in its application. Extending functionality beyond what BDI is supposed to do is not in the scope of this project. Rather, increasing accessibility further to more audiences seems a more productive endeavor. The framework's strength lays on its small size and functionality. Versions of other languages, like JavaScript, Ruby, C#, Python, and any language lacking a variety of resources for AI development would help provide to more people a solid foothold on their AI related projects.

Another area of possible expansion is incorporating other AI models, as developers could find a need to use multiple models to achieve their desired results.

9.3 Closing Remarks

This project was in no small part inspired by my great passion for video games. The virtual reality is in many ways more *humane* than the cold, harsh reality of nature and survival of the fittest. Games are a form of art, a reflection of a group's mind and vision, and the player's interpretation of it. The AI that interacts with the players in some way reflects the personality of those who wrote them. That is what drove the tone and perspective of this project.

Planning is only a small part of the field of artificial intelligence, but in games making choices is fundamental, it is part of what makes a game a game. Therefore, AI are by extension players as well. They make choices, for or against the player. The AI's creator knows how to play the game, naturally, thus transferring that knowledge as easily as possible is what makes a planning model good in this context. This is where BDI stands out from the other models studied, it is by design meant to make the transfer of knowledge from the expert to the agent as easy as possible, as we ourselves do when we communicate with others. BDI is by no means meant to be complex or difficult, and it showed as implementing the entire framework took a very small amount of code, plus thorough testing. It shows BDI is not a model that requires ample expertise or knowledge to work with, nor special languages and huge frameworks. Anyone can use it, just like anyone can make a game, on any language or system.

Bibliography

COMPUTERCRAFT. 2015. Website. <http://www.computercraft.info/>. Citado 2 vezes nas páginas 6 and 29.

EROL, K.; HENDLER, J. A.; NAU, D. S. Htn planning: Complexity and expressivity. In: . [S.l.]: AAAI Press / The MIT Press, 1994. p. 1123–1128. Citado na página 14.

FIKES, R. E.; NILSSON, N. J. Strips: A new approach to the application of theorem proving to problem solving. In: *Proceedings of the 2Nd International Joint Conference on Artificial Intelligence*. [S.l.]: Morgan Kaufmann Publishers Inc., 1971. p. 608–620. Citado 2 vezes nas páginas 13 and 14.

GEORGIEVSKI, I.; AIELLO, M. An overview of hierarchical task network planning. *arXiv.org*, p. 45, 2014. Citado na página 15.

JADEX BDI Tutorial. 2015. Website. <http://www.activecomponents.org/bin/view/Documentation/Overview>. Citado na página 21.

JADEX BDI User Guide. 2015. Website. <http://www.activecomponents.org/bin/view/Documentation/Overview>. Citado 2 vezes nas páginas 21 and 22.

JASON, A Java-based interpreter for an extended version of AgentSpeak. 2007. Website. <http://jason.sourceforge.net/Jason.pdf>. Citado 2 vezes nas páginas 7 and 22.

LUA, Artificial Intelligence. 2015. Website. <http://lua-users.org/wiki/ArtificialIntelligence>. Citado na página 6.

MINECRAFT. 2015. Website. <https://minecraft.net/>. Citado 2 vezes nas páginas 5 and 25.

NAU, D.; GHALLAB, M.; TRAVERSO, P. *Automated Planning: Theory & Practice*. [S.l.]: Morgan Kaufmann Publishers Inc., 2004. Citado na página 13.

NORLING, E.; SONENBERG, L. Creating interactive characters with bdi agents. In: *Proceedings of the Australian Workshop on Interactive Entertainment (IE'04)*. [S.l.: s.n.], 2004. p. 69–76. Citado na página 15.

NOSRATI, M.; KARIMI, R.; HASANVAND, H. A. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, v. 2, n. 4, p. 251–256, 2012. Citado na página 19.

OPENGL API Documentation of glBegin. 2015. Website. <https://www.opengl.org/sdk/docs/man2/xhtml/glBegin.xml>. Citado na página 7.

RAO, A. S.; GEORGEFF, M. P. Bdi agents: From theory to practice. In: *In proceedings of the first international conference on multi-agent systems. (ICMAS-95)*. [S.l.: s.n.], 1995. p. 312–319. Citado 2 vezes nas páginas 16 and 17.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach (3rd Edition)*. 3. ed. [S.l.]: Prentice Hall, 2009. Citado 6 vezes nas páginas 5, 11, 12, 13, 14, and 19.

WOOLDRIDGE, M.; JENNINGS, N. R. An overview of hierarchical task network planning. *Cambridge University Press*, n. Knowledge Engineering Review Volume 10 No 2, p. 62, 1995. Citado na página 11.