2015

Drive-Based Utility-Maximizing Computer Game Non-Player Characters

Colm Sloan
Technological University Dublin

Recommended Citation

Sloan, C. (2015). Drive-based utility-maximizing computer game non-player characters. Doctoral Thesis. Technological University Dublin. doi:10.21427/D7KK57

Drive-Based Utility-Maximizing Computer Game Non-Player Characters

by

Colm Sloan

Supervisors: Dr. Brian Mac Namee
Dr. John D. Kelleher

School of Computing

Dublin Institute of Technology

A thesis submitted for the degree of

Doctor of Philosophy

January, 2015

To my family.

Declaration

I certify that this thesis which I now submit for examination for the award of Doctor of Philosophy, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work.

This thesis was prepared according to the regulations for postgraduate study by research of the Dublin Institute of Technology and has not been submitted in whole or in part for an award in any other Institute or University.

The work reported on in this thesis conforms to the principles and requirements of the institute's guidelines for ethics in research.

The Institute has permission to keep, to lend or to copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signature_____ Date_____

Acknowledgements

I would like to start by thanking my supervisors John and Brian. I feel lucky to have gotten such amazing supervisors who were not just academically helpful, but also patient and understanding. For reasons beyond my understanding these two guys continued to believe in me even when I had stopped believing in myself. Their cheer and guiding hand always kept me plodding through the treacherous trenches of this Ph.D.

I would like to thank my colleagues: Ken, Niels, Rob, Mark, Amy and Yan. These folks made my office feel like a welcoming home. I'd like to give a special thanks to Patrick. Aside from helping whenever I needed it, my daily conversations with Patrick were always the highlight of my work day. He'd listen to my fearful, ignorant ramblings and reply with thoughtful insights that have changed how I view the world.

I would like to thank my family for their unconditional love and support, in particular my parents. My dad and Pam moved the moon and mountains to help me.

I would finally like to thank Lin, my beautiful girlfriend. I'm certain that I would not have finished this thesis without her

constant love and support. A poem for my gorgeous girl:

You exhumed me from unharmonious lands

So desolate and larkless,

And swung your enkindled melodious hands

That torched my darkest darkness.

Abstract

The are a number of systems used to select behaviour for non-player characters in computer games. Action planners are a powerful behaviour selection system that have been used in a number of computer games. There are action planners that can make plans to achieve multiple goals, apply actions that partially satisfy action preconditions, complete actions in a contextually appropriate manner, and be able to measure how good a particular state is for the planning non-player character, but to the best of our knowledge, there is no system that does all of these things. The purpose of this thesis is to describe such an action planner and empirically demonstrate how this system can outperform other behaviour selection systems used in computer games.

This thesis presents Utility-Directed Goal-Oriented Action Planning (UDGOAP), an action planner that combines utility, drives and smart objects to be able to simultaneously plan for multiple goals, select actions that partially satisfy preconditions, create a measure of the usefulness of a particular state, and execute actions in a more contextually appropriate manner. This system is evaluated in two very different environments and against two

behaviour selection systems that have been used successfully in industry. The evaluations show that UDGOAP can outperform these systems in both environments.

Another novel contribution of this thesis is smart ambiance. Smart ambiance is an area of space in a virtual world that holds information about the context of that space and uses this information to have non-player characters inside the space select more contextually appropriate actions. Information about the context comes from events that took place inside the smart ambiance, objects inside the smart ambiance, and the location of the smart ambiance. Smart ambiance can be used with any cost based planner. This thesis demonstrates different aspects of smart ambiance by causing an industry standard action planner to select more contextually appropriate behaviours than it otherwise would have without the smart ambiance.

Contents

1 Introduction **1**

1.1 Structure 5

1.2 Publications 6

2 Behaviour Selection Systems in Computer Games **9**

2.1 Game Trees 11

2.2 Rule-Based Systems 12

2.3 Learning Systems 17

2.4 Cognitive Architectures 18

2.5 Action Planning 21

2.5.1 Non STRIPS-Based Planners 22

2.5.2 STRIPS-Based Planners 24

2.5.2.1 Stanford Research Institute Problem Solver . 25

2.5.2.2 Goal-Oriented Action Planning 29

2.5.2.3 GOAP Extensions and Variations 39

2.5.3 Utility-Driven Action Planning 42

2.5.4 Smart Objects 45

2.6 Conclusions 48

3 The UDGOAP System **51**

3.1 Overview 51

3.2 Facts and Memory 59

3.3 Drives 60

3.4 Goals 63

3.5 NPCs 65

3.5.1 Sensing 66

3.5.2 Updating Drives 67

3.6 Smart Objects 70

3.7 Actions 71

3.8 Planner 74

3.8.1 Root Plan State and Successor State Generation 78

3.8.2 Best State Selection and Goal Completion 80

3.8.3 Finding Applicable Actions 80

3.8.4 Mapping Action Keys to Facts 83

3.8.5 Action Application and World Effect Simulation to Generate New States 85

3.9 Worked Example 94

4 Utility and Action Planning Viability in Computer Games **99**

4.1 Method 100

4.2 Evaluation 106

4.3 Results 108

4.4 Discussion 109

5 Multiple Goals and Complex Environments **111**

5.1 Method . . . 112

5.1.1 Behaviour Selection System Design . . . 115

5.1.1.1 GOAP . . . 115

5.2 Results . . . 117

5.3 Discussion . . . 118

6 Dynamic Goals and Environments **121**

6.1 Method . . . 122

6.1.1 Behaviour Selection System Configuration . . . 125

6.1.1.1 Half-Life 2 Finite State Machine . . . 126

6.1.1.2 GOAP . . . 130

6.1.1.3 UDGOAP . . . 131

6.2 Results . . . 131

6.3 Conclusion . . . 134

7 Smart Ambiance **137**

7.1 Smart Ambiance Overview . . . 140

7.2 Aspects of Smart Ambiance . . . 143

7.2.1 Location Ambiance . . . 143

7.2.2 Event Ambiance . . . 146

7.2.3 Object Ambiance . . . 148

7.3 Conclusions . . . 150

8 Conclusions **153**

8.1 Summary of Contributions 154

8.2 Open Problems and Future Work 160

A Detailed UDGOAP Worked Example **165**

A.1 Summary 179

List of Tables

4.1 Processor usage for behaviour selection systems in simulation . 108

5.1 Actions available to the NPC during the simulation. 113

5.2 Time taken in seconds for an NPC using a given behaviour system to reach the experiment termination condition of a near-optimal state for all NPC drives having started in a state where all drive began with a value of zero. 117

6.1 The final set of GOAP action costs. 130

6.2 The final set of UDGOAP weights. 131

6.3 The mean lifespan and the standard deviation of the lifespan of the wizard over 300 episodes for each behaviour selection system with its configuration optimized for lifespan in the arena.134

6.4 The p-values of a Dunn test comparing the lifespan of the wizard using each behaviour system for 300 episodes, checking if each system performed significantly better than the others, where the values are adjusted using the Hochberg procedure. . 135

A.1 Facts known to wizard after first round of sensing. 168

A.2 Wizard object-action pairs. 171

A.3 Plan states after initial state generation. 175

A.4 Plan states after one iteration of the main loop. 178

A.5 Plan states after two iterations of the main loop. 178

A.6 Plan states after three iterations of the main loop. 179

List of Figures

2.1 Schematic for an NPC with a rule-based behaviour selection system for behaviour control. 14

2.2 A behaviour tree for a dog who should seek out and eat food. 16

2.3 A schematic of a Goal-Oriented Action Planning NPC and how it interacts with its environment and a Goal-Oriented Action Planner. 33

2.4 A representation of the Goal-Oriented Action Planning plan formulation. 35

3.1 A wizard with the actions of drinking the health potion, drinking the mana potion, or drinking the elixir potion (where the elixir restores both health and mana) available to him. 53

3.2 A class diagram of the UDGOAP system. Rounded rectangles represent classes. The underlined words at the top of these rounded rectangles denote the class name. Each line of text above the dashed line within the class represents a data member of that class. Each line of text below the dashed line represents a function belonging to that class. Lines with filled diamonds denote a has-a relationship which may be either one-to-one or one-to-many and the arrow denotes inheritance. 57

3.3 An overview of how the different components in the UDGOAP system interact. 58

3.4 A representation of how the hierarchy of facts, goals, drives, and top-level utility functions are connected in UDGOAP. Each goal references a fact. Each drive references the goals it generated. The top-level utility function references the drives. 59

3.5 A representation of UDGOAP plan formulation. 77

3.6 The process of mapping action precondition and effect key to their relevant fact. 84

3.7 Representation of planner simulating an action and calculating the utility of the plan state. 88

3.8 Representation of calculating the utility of a plan state. 92

4.1 A screenshot of the hospital simulation. 101

4.2 The finite state machines used during the hospital simulation. Rectangles represent states. Rectangles with double borders represent start states. Directed lines represent the transition of one state to another. The words enclosed in square brackets along a line are the conditions that must be true to transition between states. Rounded rectangles represent classes. 103

4.3 A representation of some of the actions available to a nurse in the simulation. 104

5.1 A screenshot of the Sims-like household environment. 112

6.1 A screenshot of the alien wizard fighting enemy antlions in the arena. 122

6.2 A flowchart showing the behaviour selection process used by the HL2FSM. 126

6.3 The surface produced by the mean 40 episodes for a pair of health and mana thresholds. 129

6.4 Histograms showing the wizard lifespan for each of the three behaviour selection systems. 132

7.1 A class diagram of smart ambiance. Lines connected by filled diamonds denote a has-a relationship. 141

7.2 A screenshot taken in the virtual library. 145

7.3 A screenshot pedestrians crossing a road. 147

Chapter 1

Introduction

Computer games use Artificial Intelligence (AI) to serve a number of functions.[1] Systems used to select the behaviour of non-player characters (NPC) in games are a topic of particular interest. An NPC is any character in a game that is not controlled by a player. As the graphical fidelity of in-game NPCs nears photo-realism, this has created an expectation in players that these characters will behave in an appropriately realistic and intelligent manner. There are a variety of systems that have been developed to select behaviour of NPCs (see Rabin (2002) for a good overview). One way to distinguish between these different systems is to divide them into rule-based systems and planning systems.

[1] AI is used for path planning (Higgins, 2002), action planning (Orkin, 2003), movement planning (Champandard, 2003), story management (Mateas & Stern, 2002), story generation (Riedl & Young, 2006), camera control (Burelli & Jhala, 2009), map generation (Togelius *et al.*, 2010), animation management (Van Basten & Egges, 2009), dialogue management (Pinto, 2008), terrain reasoning (van der Sterren, 2001), NPC direction (Kline, 2011), music management (Rossoff, 2009), game commentary (Frank, 1999), and more (Bauckhage & Thurau, 2004; Cole *et al.*, 2004; Galli *et al.*, 2009).

Rule-based systems that specify exactly how an NPC should behave in some predefined set of situations have been widely used in games. Planning systems instead give an NPC a set of goals, a set actions that can be performed to achieve those goals, and a procedure to specify which actions are preferable. This separation of what actions can be performed from when actions should be performed allows the NPC to plan out appropriate behaviours for situations that were unforeseen by the designer. This ability to select appropriate behaviours for unforeseen situations is becoming more valuable as worlds in computer games become larger and more complex.

Although planning systems are less common than rule-based systems in commercial computer games, the focus of the research described in this thesis is on a planning system developed for computer games because we believe planning systems have greater potential.

Goal-Oriented Action Planning (GOAP) is a planning system designed specifically for computer games (Orkin, 2003). GOAP is the basis for many planning systems used in computer games today. GOAP associates actions with static costs that denote action preference and uses a heuristicly guided search to find the lowest cost sequence of actions to achieve a goal. GOAP was used to create critically acclaimed AI for NPCs in the game F.E.A.R. (Champandard, 2007b).

However, GOAP has a number of weaknesses that can reduce the quality of the plans it produces:

- GOAP cannot create plans that consider multiple goals.

- GOAP cannot create plans with actions that only partially achieve a precondition, where a precondition is a predicate that must be true to achieve a goal or perform an action containing the precondition.

- GOAP does not know how close a goal or condition is to being satisfied. This is a problem because the planner will not be able to know the best way to satisfy the goal or condition. For example, the goal is to kill an enemy by reducing its health to zero. Without knowing how close to death the enemy is, a risky and high damage attack might be selected when in fact the enemy may only have a little health and would die even from a safer and weaker attack.

- Each GOAP goal can only be associated with one target at a time, e.g. the kill_enemy goal can only target one enemy at a time. This causes the planner to overlook a plan that would kill multiple enemies if they could be targeted.

- GOAP assumes the world will remain static during the execution of a plan though this may not be true. This assumption can cause the GOAP NPC to make plans that might not be sensible in a dynamic environment, e.g. performing an attack and assuming that the enemy the action is being performed on will not defend itself.

A number of concepts can be used to address these weaknesses. **Utility** is a numeric representation of the desirability of a state and can be used to consider the effects of a plan on multiple goals (Mark, 2010). Utility can also be used to represent how close a goal is to being complete and

this information can be used to select actions that partially satisfy goals. Hawes (2011) describes how **drives** have been used to dynamically generate goals during play, which allow the planner to consider multiple goals that contribute toward satisfying some higher level goal and can also be used to find plans that satisfy multiple goals simultaneously. **Smart objects**, introduced by Kallmann & Thalmann (1998), and **smart ambiance** introduced by Sloan *et al.* (2011b), can place information in objects in the environment that can be used to select more contextually appropriate actions. **Action simulation** was used by Laird (2001) to simulate possible future states by predicting what actions the other NPCs in the environment might take. This thesis describes Utility-Driven Goal-Oriented Action Planning (UDGOAP), an action planning system that combines utility, drives, smart objects, and a simulation system to create a system suitable for a variety of computer game environments. UDGOAP addresses all of the previously listed weaknesses of GOAP.

In this thesis, it is shown that UDGOAP can be used to produce plans that cause more favourable outcomes than GOAP in two very different environments. The first environment is a slow-paced, single-NPC, static, and deterministic. The second environment is a fast-paced, multi-NPC, dynamic, and non deterministic environment. These different environments were selected to test how UDGOAP performs across a spectrum of environment types.

The following is a summary of the main contributions of this thesis:

- A review of literature regarding behaviour selection systems in computer games (Chapter 2).
- A novel action planner that combines utility, drives, and smart objects to create a behaviour selection system capable of planning for multiple goals, using partially satisfying actions, dynamically generating goals, and executing actions in a more contextually appropriate manner (Chapter 3).
- A feasibility study testing if action planners require a prohibitively large amount of resources for real-time computer games (Chapter 4).
- An empirical evaluation of UDGOAP running against GOAP in a static, single-NPC, deterministic environment (Chapter 5).
- An empirical evaluation of UDGOAP running against industry standard behaviour selection systems in a highly dynamic, multi-NPC, non-deterministic environment (Chapter 6).
- A novel system using smart ambiance to dynamically alter action costs to produce more contextually appropriate behaviours (Chapter 7).

1.1 Structure

The remainder of this thesis has the following structure. Chapter 2 surveys behaviour selection systems that have been used in computer games with a particular focus on action planners. Chapter 3 provides a detailed description of UDGOAP, the final action planning system designed as part of

this research. Chapter 4 describes an experiment to test the feasibility of a utility-based behaviour selection system for computer games. Chapter 5 describes an experiment that compares UDGOAP to GOAP in a slow-paced environment. Chapter 6 describes an experiment that takes place in a fast-paced environment and compares the final version of UDGOAP to GOAP and to a the finite state machine system used in a commercially successful computer game. Chapter 7 details an extension of smart objects that allows more contextually appropriate actions to be selected by a planning system and describes demonstrations of this extension in use. Chapter 8 summarizes and draws conclusions on the main contributions of this thesis and highlights potential avenues for future work.

Throughout this thesis, examples will use wizards and goblins to provide concrete examples of how UDGOAP may act in a particular situation. However, these wizards and goblins are illustrative and UDGOAP may be applied be domains beyond these fantastical creatures.

1.2 Publications

Publications supporting the contributions of this thesis are listed below.

Sloan, C., Kelleher, J. & Mac Namee, B. (2011a). Feasibility study of utility-directed behaviour for computer game NPCs. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 5, ACM.

Sloan, C., Kelleher, J. & Mac Namee, B. (2011b). Feeling the ambiance: using smart ambiance to increase contextual awareness in game NPCs. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 298–300, ACM.

Sloan, C., Mac Namee, B. & Kelleher, J.D. (2011c). Utility-directed goal-oriented action planning: A utility-based control system for computer game NPCs. In *Proceedings of the 22nd Irish Conference on Artificial Intelligence and Cognitive Science*.

Sloan, C., Mac Namee, B. & Kelleher, J.D. (2010). A comparison of computer game behaviour control systems for background characters in a simulated hospital environment. In *STAIRS 2010: Proceedings of the Fifth Starting Ai Researchers' Symposium*, IOS Press.

Kelleher, J.D., Ross, R.J., Sloan, C. & Mac Namee, B. (2011). The effect of occlusion on the semantics of projective spatial terms: a case study in grounding language in perception. *Cognitive processing*, **12**, 95–108.

Kelleher, J.D., Sloan, C. & Mac Namee, B. (2011). An investigation into the semantics of English topological prepositions *Cognitive processing*, **10**, 233–236.

Chapter 2

Behaviour Selection Systems in Computer Games

Computer game worlds can contain characters not controlled by a human player. These character are called Non-Player Characters (NPCs). These NPCs must operate autonomously. A **behaviour selection system** chooses which actions will be performed by an NPC. This chapter provides an overview and critique of the sort of behaviour selection systems, particularly state-space action planning behaviour selection systems, that have been used in computer games. State-space planning is the focus of this chapter because it is a popular method of planning in computer games. The chapter will also introduce the concept of utility and its application in computer games. The use of utility in games is explored because it has been largely ignored for use in behaviour selection systems for computer games but has great potential

for improving these systems. Smart objects are another focus of this chapter because of their potential to help NPCs select more contextually appropriate actions and because of their potential to allow behaviours to be performed in more contextually appropriate ways.

Behaviour selection systems are used to instruct an NPC in how the NPC should act. Game designers want the most general and simple behaviour selection that they can have for their game. Designers want the system to be general such that the system can produce behaviours that are appropriate for a wide variety of unforeseen situations. Designers want the system to be simple because it is more easily setup, extended, and maintained. Behaviour selection systems must make trade-offs between generality and simplicity because creating more robust, general systems often makes the system more complex. There is a spectrum of generality across behaviour selection systems with simple, very domain specific systems at one end of the spectrum and complex but general systems at the other end. This section describes behaviour selection systems used in computer games, starting at the simple, less general end of the spectrum and working toward the general end.

This chapter has the following structure. The remaining sections of this chapter describe a number of behaviour selection systems that have been used in computer games. In Section 2.5, we define action planning, which is a family of behaviour selection systems, and describe action planning systems that have been used effectively in a number of games. The section goes on to describe STRIPS, a seminal planner upon which many other states-space planners have been built. The section concludes with an in depth description

of Goal-Oriented Action Planning (GOAP), a planner based upon STRIPS and the planner that has been used as the foundation of a number of action planners used in computer games. Several systems based on GOAP are described with their strengths and weaknesses. Section 2.5.3 defines utility, the concept around which UDGOAP was built, and describes a number of systems that have used utility for behaviour selection in computer games. Section 2.5.4 describes smart objects, used by UDGOAP for calculating utility, explains the advantages of using smart objects in planning, and describes how smart objects have been used in several computer games. The chapter concludes with a summary in Section 2.6.

2.1 Game Trees

A **game tree** is a structure commonly used in turn-based computer games. A game tree is a directed graph that represents states that can exist in a game world (Russell & Norvig, 2009). Each node in the game tree represents a state of a game and each edge represents an action that causes the transition of one state to another. The root of the game tree is the current state of the game world. In a complete game tree, leaf nodes are win or lose game states. A goal state is a leaf node that results in winning the game. Games trees have been used by systems such as Deep Blue (Campbell *et al.*, 2002) to select actions for a computer-controlled player in games of chess. A computer-controlled player can search the game tree for a favourable game state and perform the actions required to generate that state.

An optimal strategy for a game can be produced from a game tree by searching the game tree using a **minimax** algorithm (Willem, 1996). Minimax first rates nodes to know the **utility** (usefulness, described in more detail in Section 2.5.3) of each node for winning the game. The rating of a node is equal to the rating of the leaf node that would be reached if all players performed actions that minimize the maximum utility of a position to the opponents. For example, in a game of tic-tac-toe, a minimax player would place its x's such that it creates the worst position for the second player, assuming the opponent will place its o's such that it creates the worst position for the first player.

It is often impossible to calculate an entire game tree. Although a simple game, the complete game tree for tic-tac-toe has approximately 25,000 nodes (Chu-Carroll, 2008). Game trees for more complex games, such as chess, are even larger but can be used by employing methods that reduce the number of nodes considered, such as **alpha-beta pruning** (Knuth & Moore, 1976). However, games with many agents, capable of performing many actions in a continuous environment may simply be too complex for game trees to be used to select actions, necessitating more simple systems, such as rule-based systems.

2.2 Rule-Based Systems

This section will first describe simple rule-based systems and will then move on to finite state machines, decision trees and behaviour trees. It will then

describe more general systems that learn rules. It follows with a description of cognitive architectures used in games, where cognitive architectures are considered the most general type of behaviour selection system that have been used for computer games. The section concludes with a summary of these systems and an argument that action planners have a level of generality that is desirable to NPC behaviour selection.

Rule-based systems are very popular in computer games. The simplest and least general behaviour selection systems are purely reactive, mapping from the world the NPC is currently perceiving to actions. This mapping occurs using a set of predefined rules. A schematic for an NPC with such a rule-based behaviour selection system is shown in Figure 2.1[1]. This figure and others like it throughout the thesis are based on work in Russell & Norvig (2009). Rounded rectangles represent classes and the underlined word at the top represents the name of the class. Rectangles represent processes and the phrase inside the rectangle describes the process. Arrows represent data flow, where arrows pointing into a process are input to the process and arrows pointing out of a process are output of the process. Each arrow is accompanied by a phrase describing the data the arrow represents. The origin of an arrow denotes the origin of the data, such that arrows originating from inside a class mean that the data represented by the arrow resides within the class.

In Figure 2.1, the set of percepts and a set of rules are passed to a func-

[1]Actuators, internal memory and the distinction between internal and external actions have been omitted from the figure for simplicity, though they would still be a part of the system.

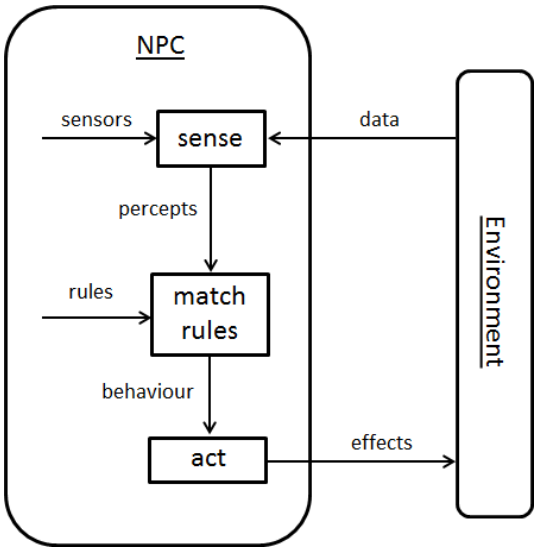


Figure 2.1: Schematic for an NPC with a rule-based behaviour selection system for behaviour control.

tion that will check which rules should be activated. The output of the rule matching process will be the behaviour that the NPC should execute. Examples of simple rule-based systems in games include Agre & Chapman (1987), Shapiro (1999), and Khoo *et al.* (2002).

A more complex version of a rule-based system is a finite state machine. A finite state machine considers both percepts and the internal NPC state during rule matching. In a finite state machine, each state contains rules that control which behaviour to select and rules that control when another state should be entered. Consequently, during rule matching, a finite state machine only considers a subset of rules defined in the current state. Examples of finite state machines in games include those described in Houlette & Fu (2003) and still more variations described in Buckland (2005), Straatman (2009), Hoang *et al.* (2005), Laming (2008), Kolhoff (2008), Fu & Houlette (2002), Tozour (2004), and Dybsand (2001).

A decision tree is a hierarchical rule-based system. A decision tree can

be used in games as a behaviour selection system where each non-leaf node of the tree is an if statement that decides which node to move to next, and where each leaf node is a behaviour. The hierarchical structure of the tree gives more control to designers over a flat rule set because the hierarchy can be used to eliminate undesirable combinations of rules being triggered. This elimination is made possible by allowing only a subset of rules to be applicable once a particular condition is satisfied. Examples of decision trees in games include Evans (2002), Lau (2008) and Fu & Houlette (2004).

A behaviour tree is more general than a decision tree and is designed to be more robust and modular. Leaf nodes in behaviour trees are usually actions that an NPC should perform. Nodes can have many forms, including the simple if statements of the decision trees, marking branch priority, or altering the value of a variable. An example of a behaviour tree for a dog getting food is shown in Figure 2.2. Nodes can report their success or failure and branches of the next highest priority can be tested. Examples of behaviour trees in games include Isla (2005) and Isla (2008).

Rule-based systems can be quickly authored, are light-weight as they require little memory and computation, and are usually easily tested because they are deterministic. These characteristics are considered desirable to game designers.

However, rule-based systems have a number of weaknesses (Wallace, 2004). These systems are mostly reactive as they only consider the present state of their world. Selected behaviours might be sensible for the immediate situation the NPC using the system is in, but may not be sensible for the situation

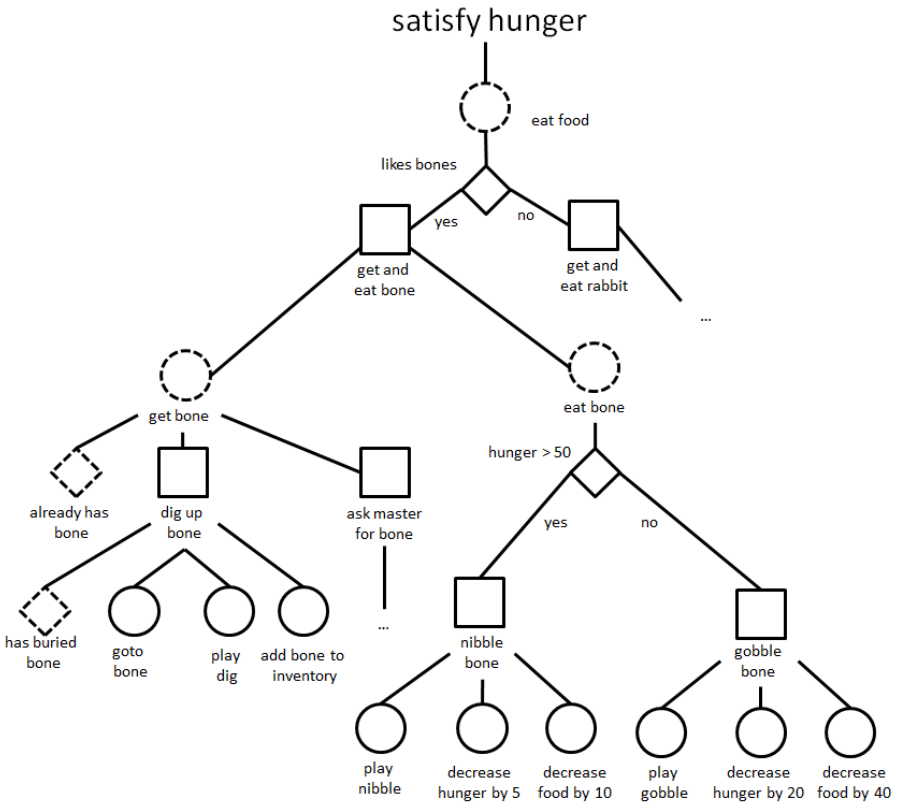


Figure 2.2: A behaviour tree for a dog who should seek out and eat food.

the NPC will be in moments later. These simple systems also require the designer to anticipate many or all situations in which an NPC using the system may find itself. Consequently, a rule-based behaviour selection system for an NPC can require a very large rule set. For example, the Soar Quakebot, a bot in Quake[1] controlled by a system using the Soar architecture (Laird *et al.*, 1987), uses over 800 rules (Laird & VanLent, 2001). Large rule sets are problematic because rules in rule-based systems can interact with each other in unintended and undesirable ways when the rule set becomes large (Bourg & Seemann, 2004).

[1]Quake - id Software - http://www.idsoftware.com/en-gb

2.3 Learning Systems

Rule-based systems are brittle because they can easily select undesirable behaviours in unforeseen situations. They also have the weakness of using an unchanging set of rules. Players can learn these rules, which can lead to predictable gameplay. Learning systems don't require hand-authoring of behaviours and can alter their behaviour as players interact with the system, which can lead to less repetitive behaviour. Approaches to building learning systems that have been used in games include:

- case-based reasoning (Aha *et al.*, 2005; Flórez-Puga *et al.*, 2009; Jaidee *et al.*, 2011a; Ontañón *et al.*, 2007)
- Markov models (Zubek, 2006)
- genetic algorithms (Lichocki *et al.*, 2009; Lim *et al.*, 2010)
- neural networks (Generation5, 2005; Hefny *et al.*, 2008; MacNamee & Cunningham, 2003; Sweetser, 2004; Thompson & Levine, 2009)
- neuroevolution (Cornelius *et al.*, 2006; Jang *et al.*, 2009; Parker & Bryant, 2009; Schrum *et al.*, 2011; Traish & Tulip, 2012)

However, learning systems may not be desirable for games because, among other reasons, they often require a large dataset to train on to produce desirable behaviours and such large datasets can be difficult to produce while developing a game. This problem is exacerbated by the fact that values upon which the system was trained, e.g. the strength of some unit, often change

during development, which could invalidate previously learned behaviours that only made sense given the old value.

2.4 Cognitive Architectures

Cognitive architectures are behaviour selection systems that are designed with the goal of achieving general intelligence. The designers of cognitive architectures try to imitate the behaviour selection process used by humans and other animals (Newell, 1994). Cognitive architectures are opposite the rule-based systems on spectrum of generality for behaviour selection systems. Cognitive architectures can learn complex interactions and rules in an online environment using approaches such as short-term memory describing things recently known to the NPC, procedural memory describing sequences of actions useful in particular situations, episodic memory like the memory used by case-based reasoners, and long-term memory where useful information from short-term memory can be stored. Examples of cognitive architectures used by NPCs in games include:

- Soar (Laird *et al.*, 1987; Laird, 2001; Magerko *et al.*, 2004; Wintermute *et al.*, 2007)

- ICARUS (Choi *et al.*, 2007; Langley & Choi, 2006; Li *et al.*, 2009)

- Global Workspace Theory (Arrabales *et al.*, 2009; Baars, 1993; Fountas *et al.*, 2011)

Although cognitive architectures are powerful (Laird & Nielsen, 1994), they may not be suitable for a number of games because they can require a lot of resources and can be very complex.

This section described rule-based systems, learning systems and cognitive architectures. Rule-based systems are very domain specific and must be have designers hand make each rule for a wide variety of situations for the NPC to behave sensibly. Learning systems are more general as they can learn rules based on previous experience, freeing designers from the burden of worrying about many unforeseen situations but allowing the system to learn unusual behaviours or behaviours that may become invalid when game values change. Cognitive architectures try to alleviate this problem by trying to design very general systems that are biologically inspired, but are very complex and resource intensive. However, the increased generality of cognitive architectures over rule-based systems has numerous benefits. By allowing for more generality, a behaviour selection system could have abilities including:

- storing memories (Choi *et al.*, 2007; Li *et al.*, 2009; Orkin, 2003)
- building up a solution in steps (Fikes & Nilsson, 1971; Orkin, 2003)
- comparing goals to know which is preferable (Molineaux *et al.*, 2010; Young & Hawes, 2012)
- comparing actions to know which is preferable (Benton *et al.*, 2007; Orkin, 2003)
- comparing solutions to know which is preferable (Hawes, 2003; Nareyek,

1999; Orkin, 2003)

- explaining why a plan has failed (Molineaux *et al.*, 2010)
- inferring which beliefs are more likely to be true (Baars, 1993)
- generating goals and reasoning on which goals should be generated (Molineaux *et al.*, 2010)
- managing goals e.g. aborting and suspending goals (DePristo & Zubek, 2001; Molineaux *et al.*, 2010; Muñoz-Avila *et al.*, 2010a)
- recognizing and handling conflicting goals (Do *et al.*, 2007)
- justifying decisions on more than just a numeric basis (Nareyek, 2001a)
- understanding which information is obsolete and can be removed (Orkin, 2003)
- reasoning on the current intention of another NPC (Laird, 2001; Weber, 2012)
- predicting future states (Muñoz-Avila *et al.*, 2010b)
- learning goal priorities (Young & Hawes, 2012)
- learning how goals can become invalid (Jaidee *et al.*, 2011b)
- learning and deploying generalisations e.g. when in state s with goal g, action a is a good choice (Laird *et al.*, 1987)
- learning the effects of actions (Jaidee *et al.*, 2011b; Laird *et al.*, 1986)

Each new ability can give greater justification for behaviours and result in a more robust behaviour selection system. However, every additional ability comes with an added complexity cost, which will make system more difficult to test, debug, integrate with tools, and maintain. The designer should find the system that gives the best trade-off between simplicity and generality. A number of computer game studios[1] have settled on a middle ground by using action planners. Action planners use powerful but intuitive processes (Ghallab *et al.*, 2004) that make them an ideal choice for many types of computer games.

2.5 Action Planning

Action planning is a form of behaviour selection where a planning system searches for a plan that satisfies a goal or goals, where a plan is a sequence of actions. For brevity, a system that implements action planning will be referred to as a **planner** and **action planning** will be referred to as **planning** throughout the remainder of this thesis.

This section will briefly describe a number of planners used in games, where planners are divided into two categories: planners based upon the seminal STanford Research Institute Problem Solver (STRIPS) (Fikes & Nilsson, 1972) planner, and planners not based upon STRIPS. Goal-Oriented Action Planning (GOAP), a popular STRIPS-based planner for computer games, receives particular focus on this section because the system developed as part

[1]Bethesda Softworks, Monolith Productions, Creative Assembly, Eidos Interactive, High Moon Studios

of this research is strongly based upon GOAP and the two systems are later compared in experiments.

2.5.1 Non STRIPS-Based Planners

There are a number of action planning systems not based on STRIPS, including hierarchical task networks, case-based planners, and constraint-based satisfaction planners.

A hierarchical task network is a behaviour selection system where a specified task is recursively decomposed down to actions that an NPC can execute as a plan (Ghallab *et al.*, 2004). Hierarchical task networks are expressive and fast enough to be used in a complex real-time environment, as shown by Champandard (2012) and Hawes (2004). However, developers of the hierarchical task networks used in multiplayer Killzone 2[1], where the hierarchical task networks featured over 1000 branches, claimed hierarchical task networks became difficult to work with (Straatman, 2009). Furthermore, hierarchical task networks specify exactly how to decompose tasks. This task decomposition adds to behaviour authorship time and makes behaviours more brittle because the designer might have authored a behaviour with certain assumptions that can become invalid later in the development process because of changes in game values. A number of variations of hierarchical task networks that have been developed for NPCs, including Gorniak & Davis (2007); Hawes (2003); Hoang *et al.* (2005); Meneguzzi & Luck (2007); Muñoz-Avila *et al.* (2010a); Straatman (2009).

[1]Killzone 2 - Guerrilla Games - http://www.killzone.com

Case-based planning (Spalzzi, 2001) is the use of case-based reasoning (Aamodt & Plaza, 1994) to formulate plans for an NPC. A database of plans is created either by an expert (Ontañón *et al.*, 2007), through the collection of data from many non-experts (Weber, 2012), or some other means (Ontanón *et al.*, 2009). Planners then lookup what advantageous plans were performed in situations similar to the current situation of the planning NPC.

A problem with the case-based planners approach is that they can require laborious labelling by experts, but such labelling can yield excellent accuracy for associating player plans with player intentions. Automated labelling can be used to reduce the need for human experts by taking a large number of games and labelling them based on some set of rules or gameplay strategies (Weber, 2012). However, this automated labelling method can incorrectly associate plans with an intention when the plan might have coincidentally, rather than causally, achieved goals relevant to the intention. This can lead to the case-based planner selecting ineffectual plans to achieve goals.

A constraint satisfaction problem (Kumar, 1992) is a problem where a fixed number of variables must be assigned values that don't violate the rules associated with those variables. A structural constraint satisfaction problem is a type of constraint satisfaction problem where the variables or constraints involved don't need to be known by the planner when plan formulation begins. Instead, only the types of constraints and the structural constraints of the problem need to be known. In a structural constraint satisfaction problem, search for the correct structure of the problem is part of the constraint satisfaction process. The Excalibur project (Nareyek, 2001b)

has been used in games to model the problem of selecting behaviour as a structural constraint satisfaction problem (Nareyek, 2000).

A structural constraint satisfaction problem solver always has a plan available because it immediately populates all variables and very quickly improves the plan generated. However, a weakness of this planner (Nareyek, 1998) is that it may return a plan with inconsistent values that may not result in intelligent NPC behaviour when executed. Also, plan quality is measured by its inconsistency with the constraints rather than how beneficial a plan is to an NPC.

This section described non STRIPS-based action planning systems that have been used in computer games. These systems have problems with authorship burden (hierarchical task network and case-based planner) and/or returning inappropriate behaviours (constraint satisfaction planner and case-based planner). STRIPS-based planners take approaches that may alleviate these problems.

2.5.2 STRIPS-Based Planners

This section describes STRIPS and Goal-Oriented Action Planning (GOAP), a STRIPS successor modified for use in modern computer games. GOAP is described in detail because the planner developed as part of this research is based upon GOAP. The section goes on to describe other GOAP-based systems that have been used in computer games. The design, strengths, and weaknesses of these systems are explained, as well as which parts of these systems inspired the system created as part of this research.

2.5.2.1 Stanford Research Institute Problem Solver

Fikes & Nilsson (1972) created the STanford Research Institute Problem Solver (STRIPS). STRIPS is a state-space planner, where a **state-space planner** is described by Ghallab *et al.* as the following.

> "A search algorithm in which the search space is a subset of state space: Each node corresponds to the state of the world, each arc corresponds to a state transition, and the current plan corresponds to the current path in the search space." (Ghallab *et al.*, (2004), page 69).

STRIPS was used on a mobile robot named Shakey to reason about how to achieve a goal condition by chaining together a sequence of actions[1], where the execution of an action causes a state transition in the search space. The world state is a logical abstraction of information about the state of the physical world in which Shakey resides.

In STRIPS, the world state is represented as a set of variables. Each state variable in the world state is represented using first order predicate calculus clauses e.g. at(Box1, x), where x could be any location in the environment.[2] Each STRIPS action has a name, a set of parameters, a set of preconditions, and a set of effects. The **preconditions** are a set of clauses that must be satisfied in the world state for the action to be applicable. The **effects** of

[1]The term **operator** will be replaced by the word **action** throughout this document even though an operator causes changes to the planners model of the world and actions change the world itself. This replacement is done in the interest of readability.

[2]The term **predicate** is used throughout this thesis to refer to any function that returns a boolean value.

actions alter the world state according to their predicates.

The following is an example of an action being executed by Shakey using STRIPS, where the example is taken from (Fikes & Nilsson, 1972). The action that symbolises pushing an object from one place to another could have the form push(u, x, y), where push is the name of the action, u is the object being pushed, x is the starting position of the object u, and y is the position of the object u upon the completion of the **push** action. Together, u, x, and y make up the parameters of the **push** action. The preconditions could be $(\exists\ x,\ u)[\mathsf{at}(u,\ x) \wedge (\mathsf{at}(\mathsf{Shakey},\ x))]$, which means that for the push action to be executed, the object u must be at position x and Shakey must be at the same position x.

The remainder of this section will describe the recursive STRIPS planning algorithm as implemented in (Ghallab *et al.*, 2004) and shown in Algorithm 1. This action planning algorithm, here called STRIPS_plan, was the behaviour selection system used by Shakey, and is the core of the STRIPS approach.

STRIPS planning requires a current state to be set as the initial world state, a goal state consisting of clauses that form the conditions of the goal, and a set of actions available to the planning NPC. Planning begins with no existing plan. This is the root of the planning graph, where the graph models the plan search space, nodes in the graph are states and edges are actions that cause the transition between states. STRIPS uses a **regressive planning** search to find solutions to goals.

Input: *current_state*, *goal*, *actions*
Output: *existing_plan*

existing_plan ← ∅
loop
  if *current_state* satisfies *goal* **then**
    return *existing_plan*
  end
  applicable_actions ← { *a* | *a* ∈ *actions*, *a* is relevant to *goal* }
  if *applicable_actions* = ∅ **then**
    return failure
  end
  action ← non-deterministically select action from *applicable_actions*
  first_unsat_precond ← *p* | *p* ∈ preconds(*action*), *p* is the first precondition not satisfied in *current_state*
  plan_to_satisfy_pre ← STRIPS_plan(*current_state*, *first_unsat_precond*, *actions*)
  if *plan_to_satisfy_pre* = failure **then**
    return failure
  end
  existing_plan ← *existing_plan* + *plan_to_satisfy_pre* + *action*
  current_state ← transition(*current_state*, *existing_plan*)
end

Algorithm 1: The STRIPS_plan algorithm (Ghallab *et al.*, 2004).

Regressive planning is a search for a solution that starts from the goal state and works backwards by finding actions that when inversely applied[1] would lead to the initial state, where the inverse application of an action is the goal minus the effects of the action plus the preconditions of the action. Regressive planning first takes goal conditions not satisfied in the initial state and searches for relevant actions to satisfy those conditions, where a condition is satisfied if its predicate is true in the given state. An action is relevant in STRIPS if the effects of the action satisfy at least one unsatisfied condition. An action is then selected from this set of relevant actions. Fikes & Nilsson does not specify the mechanism used to select an action in STRIPS but one strategy may be to select the action with the fewest unsatisfied conditions. The application of the selected action transforms the initial state by adding the effects of the action and creates a new branch in the planning graph. An applied action may itself have unsatisfied conditions which require the application of more actions. This continuous chaining of actions backward from the goal state may eventually result in a branch that would create a state where all conditions are satisfied. The actions that created that branch are extracted as the plan to be performed. Any branch that cannot apply an action to satisfy a condition results in failure and another branch is developed instead. If no branches are left to develop, there is no solution.

STRIPS and STRIPS-based planners were used for a number of years in academia. In 2005, a STRIPS-based planner was adapted for use in a commercial computer game. The adaptations included the association of

[1]The "application" term will be used throughout this thesis instead of "inverse application" because this thesis focuses only on regressive planners.

costs with actions to allow for more contextually appropriate behaviours, and a heuristic to guide action selection. This STRIPS successor is called Goal-Oriented Action Planning (Orkin, 2003).

2.5.2.2 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) (Orkin, 2005) is a planning framework that uses a heuristicly guided planner based on STRIPS. GOAP has been used in several AAA commercial games, such as F.E.A.R.[1], S.T.A.L.K.E.R.: Shadow of Chernobyl[2], Fallout 3[3], Empire: Total War[4] and Deus Ex: Human Revolution[5].

A GOAP NPC has a set of facts, a set of sensors, a set of predefined actions, a set of predefined goals, and a set of subsystems used to find targets for goals.

Each GOAP NPC has a set of facts that describe itself and its world. Each **fact** is associated with an entity, a fact type, a timestamp, and a value. For example, a fact could have the form: (entity: goblin1, type: position, timestamp: 19:30, value: (82, 0, 109)). The **fact type** could be a Position, Disturbance or any other category of fact that will be used to help the designer represent things that the GOAP NPC is able to know about. An entity is anything in the game; either an NPC or a non-NPC world object. The entity of the fact is the world object to which the fact pertains, e.g. goblin1.

[1]F.E.A.R. - Monolith Productions - http://www.fear3.co.uk/the-game.html

[2]S.T.A.L.K.E.R.: Shadow of Chernobyl - GSC Gameworld - http://www.stalker-game.com/

[3]Fallout 3 - Bethesda Game Studios - http://fallout.bethsoft.com/eng/home/home.php

[4]Empire: Total War - The Creative Assembly - http://www.totalwar.com/en_us/

[5]Eidos Montreal - Deus Ex: Human Revolution - http://www.deusex.com/

The timestamp denotes when the fact was last verified. Facts are generated through reasoning on sensor input and are stored in memory.

A state in GOAP is a set of key-value pairs, where a key is a symbol representing a predicate and the corresponding value represents the result of that predicate. The values of key-value pairs are derived from the set of facts available to the NPC. For example, if the predicate associated with the enemy_dead symbol was evaluated using the (entity: goblin1, type: health, timestamp: 09:20, value: 0) fact, the value would return true because the health of the goblin is 0, resulting in the (enemy_dead, true) key-value pair.

Each GOAP NPC has a set of goals made at design-time that do not change during the life of the NPC. A goal is a desired substate of the world state and consists of a set of key-value pairs. For example, the get_in_cover goal has a single condition in the form of the (in_cover, true) key-value pair. Each goal has a subsystem that searches for the best way to achieve that goal. For example, the get_in_cover goal subsystem considers all places that provide cover and stores the one that is currently best so that if the get_in_cover goal is ever selected for achievement, the plan made will use that place of cover. The chosen place of cover becomes the target of the goal. A goal is selected by first prioritizing all goals and then selecting the goal with the highest priority.

An action in GOAP has a name, a set of symbolic preconditions, a set of symbolic effects, a set of context preconditions, a set of context effects, and a cost. The set of symbolic preconditions of an action is a set of key-value pairs that must be present in the world state for the action to be applicable.

The set of symbolic effects is also a set of key-value pairs. These symbolic effects will be applied to the world state upon the application of the action. A context precondition is an arbitrary predicate that determines if an action can be applied. For example, the predicate might test if the majority of the last 10 allied combat units sighted had less than half health. This would be a context precondition because it is cumbersome to model with symbols. A context precondition is used as an escape hatch if a symbolic precondition isn't enough to determine if an action should be applied, perhaps because the symbols available aren't expressive enough to accurately model the precondition. A context effect is the effect analogue of a context precondition. A context effect can be used by a designer when it is cumbersome or unnecessary to specify effects using the symbolic key-value format. The cost of a GOAP action is specified at design-time and is used to give preference to actions, such that lower cost actions are preferable. For example, the fire_from_cover action might have a lower cost than the fire_without_cover action, but the fire_from_cover action would have the additional precondition that the NPC be in cover, making it preferable to the fire_without_cover action but only when there is cover available.

The purpose of a planner is to create a plan of actions that achieve a particular goal. GOAP plan formulation works as follows. The planner is queried to select a plan when the current plan is finished executing, the current plan has become invalid, or if a predefined amount of time has elapsed since the last time a plan was selected. A goal in GOAP is any set of conditions that an NPC wants to satisfy, be it to get an item or to kill an

opponent. GOAP subsystems find the best object in the world for achieving each of the goals of the planning NPC. For example, subsystems responsible for finding a target for the kill_enemy goal might search for the highest threat enemy, which is set as the target of that goal. A predefined goal selection mechanism selects the highest priority goal. The GOAP planner attempts to generate a solution to satisfy the selected goal, where the solution is a plan and where a plan is a sequence of actions.

To find this sequence of actions, GOAP uses a regressive A* (Hart *et al.*, 1968) search from the goal state to the initial state, where the initial state is a set of key-value pairs derived from the facts known by the planning NPC at the time that planning begins. In its more common use in path planning, A* is used to search for the lowest cost sequence of edges traversed while moving from an initial node to a goal node, where each node represents a position in the world, and each edge represents the transition from one position to another and is associated with a distance cost (Hart *et al.*, 1968). In GOAP, A* is used to search for the lowest cost sequence of edges traversed while moving from an initial node to a goal node, where each node represents a world state, and each edge represents the transition from one world state to another and is associated with an action cost (Orkin, 2006). NPCs using GOAP periodically re-evaluate their situation and choose the lowest cost plan to achieve their selected goal.

The GOAP planner works the same way as the STRIPS planner. The plan begins with an initial world state as it is known to the planning NPC, a goal state, and a set of actions available to the planning NPC. The planner

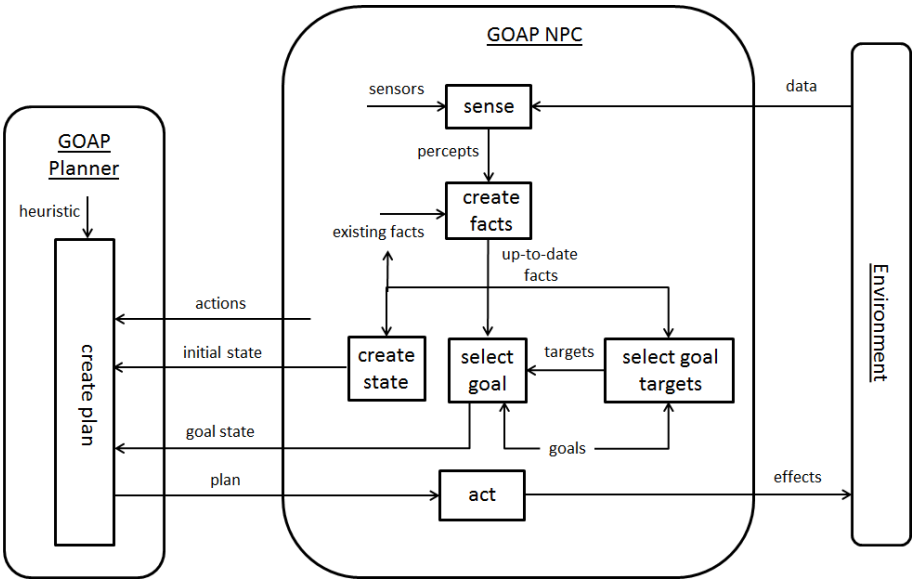


Figure 2.3: A schematic of a Goal-Oriented Action Planning NPC and how it interacts with its environment and a Goal-Oriented Action Planner.

checks if the goal is satisfied by the current world state, and if not, selects an unsatisfied condition of the goal for which to find a solution. Plan formulation terminates when the GOAP planner has found the lowest cost plan to satisfy the goal based on the cost of actions and guided by the A* heuristic.

Figure 2.3 shows a schematic of a GOAP NPC, how the NPC interacts with its environment and how the NPC plans. The GOAP NPC senses its environment and internal state and creates percepts from that data. These percepts, along with the existing facts known to the NPC, are used to generate an up-to-date set of facts that describes the world as the NPC knows it. These up-to-date facts are stored and used to generate a set of key-value pairs that form the initial plan state. The up-to-date facts are used by the goal selection subsystems to select a target for each of the goals of the NPC. The plan state of the highest priority goal is selected and sent to the planner, along with the initial state and the set of actions available to the NPC. The

planner produces a plan using the actions to transition from the initial state to a state in which the goal is satisfied. That plan is executed by the NPC as a behaviour.

An example of GOAP plan formulation is shown in Figure 2.4, which is based on work from Orkin (2003). The planner requires an initial state, a goal, and a set of actions that the planning NPC can perform. In this case, the enemy is alive in the initial state, dead in the goal state, and the actions available to the planning NPC are the attack, reload_weapon and draw_weapon actions.

Plan formulation begins by checking if the goal is satisfied given the current state. It is not because the current value of targetIsDead is false and the goal state value of targetIsDead is true so planning continues. The planner then searches through the set of actions for the lowest cost action with the effect of making the enemy target dead. This returns the attack action. The attack action has a precondition that the gun being used for the attack is loaded, causing a lookup in the set of actions for an action with the effect of the planning NPC loading its weapon. This returns the reload_weapon action. Planning continues from the state created through the regressive application of the attack and reload_weapon actions, as this is the lowest cost (and only) plan so far. The reload_weapon action has a precondition that the gun being reloaded is armed, causing a lookup in the set of actions and returning the draw_weapon action. The final state was generated through the application of a sequence of the lowest cost actions to satisfy the goal of the enemy target being dead. Plan formulation is complete because there are no unsatisfied

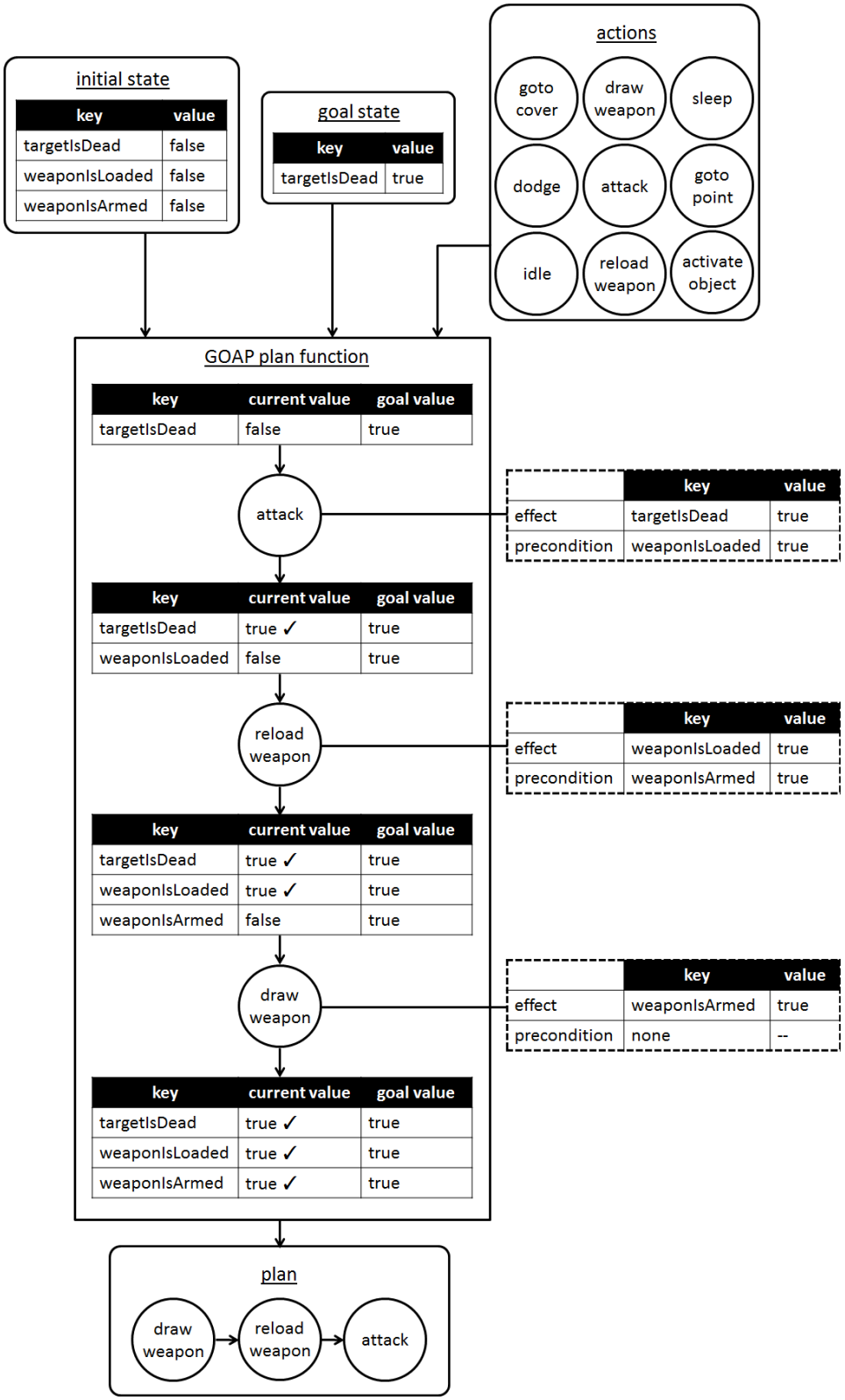


Figure 2.4: A representation of the Goal-Oriented Action Planning plan formulation.

conditions. The sequence of actions is then reversed so that the plan is in the order in which it should be executed.

```
   Input: actions, current_state, goal, heuristic
   Output: existing_plan
 1 existing_plan ← ∅
 2 loop
 3 |   if current_state satisfies goal then
 4 |   |   return existing_plan
 5 |   end
 6 |   applicable_actions ← {a | a ∈ actions, a is relevant to goal}
 7 |   if applicable_actions = ∅ then
 8 |   |   return failure
 9 |   end
10 |   applicable_actions ← calc_f_values(heuristic, current_state, goal,
11 |                                       applicable_actions)
12 |   lowest_cost_action ← argmin(f_value(applicable_actions))
13 |   first_unsat_precond ← p | p ∈ preconds(lowest_cost_action),
   |                          p is not satisfied in current_state
14 |   plan_satisfying_preconds = GOAP_plan(actions, current_state,
15 |
   |   first_unsat_precond, heuristic)
16 |   if plan_satisfying_preconds = failure then
17 |   |   return failure
18 |   end
19 |   current_state ← regress_state(current_state,
20 |                                 plan_satisfying_preconds.lowest_cost_action)
21 |   existing_plan ← existing_plan.plan_satisfying_preconds.
22 |                   lowest_cost_action
23 end
```

Algorithm 2: The GOAP_plan formulation algorithm.

Algorithm 2, based on the STRIPS algorithm from (Ghallab *et al.*, 2004), shows an implementation of a GOAP plan formation algorithm. This GOAP algorithm differs from the STRIPS algorithm shown in Algorithm 1 in that the GOAP algorithm takes a heuristic as input, which is used to guide search by selecting the lowest cost plan that achieves the goal, where the cost of a plan is the sum of the cost of the actions in the plan. The heuristic is used when calculating the A* F values for each plan state (line 10).

GOAP has several strengths:

- GOAP is modular and reusable because actions are small logical blocks that could be used over many different types of NPC or different games.
- GOAP decouples actions and goals which allows the planner to find the most contextually appropriate solution and solutions that designers may not have considered.
- GOAP has been used in a number of AAA games showing it is capable of satisfying the high expectations of such games.
- GOAP uses A* technology that has been highly optimized through extensive research. This reduces the resource burden of the algorithm and capitalizes upon familiarity that A* already has because it is already used extensively in path finding for NPCs.
- GOAP is easily debugged because the reason for making a decision does not use any black box technology, unlike some learning systems such as a neural net. This makes it easier to reproduce and correctly identify bugs, decreasing development time.
- GOAP can use a standardized planning language like PDDL (Gerevini & Long, 2005). PDDL has been standardized and used for a great deal of development, making it easier for programmers to reuse existing action definitions.

GOAP, however, also had a number of weaknesses:

- GOAP only plans with binary goals and conditions that are either satisfied or not satisfied. This means that GOAP does not have the ability to know how far a goal is from being complete and so may select plans that may not be optimal if the goal is already partially complete.

- GOAP can only plan for one goal at a time. This may cause better plans that consider more than one goal to be overlooked.

- GOAP can be hard to apply to anything but very short term goals because GOAP can only plan to a achieve a single goal without regard to state that will exist after that short-term plan is executed as GOAP assumes the world will remain static during plan execution.

- GOAP is limited in how contextually appropriate the selected plan is because actions are associated with a predefined, static cost.

- GOAP NPCs have a fixed set of goals. This limits the NPC to a maximum of one target per goal because of how the goal target subsystems are designed. For example, the kill_enemy goal might only be able to target the highest priority to kill at a given time, even though the planning NPC could perform an action to kill both the highest priority enemy and some other enemy too.

- If there is a set of actions where more than one action in the set have the same preconditions, the lowest cost action is always selected. For example, if the kick and punch actions for an NPC both have only the (enemy_within_melee_distance, true) precondition, and the kick action is

associated with a lower cost than the punch action, the punch action will never be performed. This limits the actions an NPC can perform in any situation.

A number of variations of GOAP have been created that augment GOAP and attempt to make up for its shortcomings.

2.5.2.3 GOAP Extensions and Variations

Pittman (2008) created an implementation of GOAP using Reynolds's (2002) **command hierarchy** for goal selection with the intention of selecting more contextually appropriate behaviour. The command hierarchy consists of three levels: squad level, fireteam level, and soldier level. Goals are rated based on the desires of the highest level of abstraction first and are later modified by less abstract levels. For example, the squad rates the goals based on the needs of the squad, then the fireteam modifies these ratings based on the needs of the fireteam, then the individual soldier modifies these ratings based on the needs of the soldier. This allows the soldier take an action that might not be best for the squad, but the action best for himself. Whether or not this selfishness is desirable depends on the game being played. Pittman's algorithm was implemented for NPCs in a modification for Unreal 2004[1]. Pittman's command hierarchy was used to address the GOAP weakness of not allowing for very contextually specific plans.

Layered Goal-Oriented Action Planning (LGOAP) (Maggiore *et al.*, 2013) is a hierarchical progressive-search planner where each layer of the planning

[1]Unreal Tournament 2004 - Epic Games - http://www.unrealtournament.com

process is associated with less abstract actions. A progressive-search planner is a planner that starts at the initial state and chains actions to move toward the goal state, rather than a regressive-search planner that starts at the goal state and chains actions until the initial state is reached. The planner first generates a high-level plan by forward chaining actions belonging to the most abstract level to achieve some high-level goal. Planning in LGOAP continues by successively forward chaining actions that are one level less abstract than those used in the previous layer, where the goals for a layer are the actions of the layer that is one level more abstract. The use of an action hierarchy, a heuristic, lazy evaluation of plans, and an aggressive pruning strategy mean that the planner can scale well, but may overlook optimal plans. LGOAP is able to make longer term plans than GOAP because LGOAP can plan for what to do after the most short-term goal is achieved.

GOAP was originally designed for the commercial computer game called F.E.A.R.[1] but was adapted further for a number of other commercial games. Cerpa (2008) extended GOAP for use in War Leaders: Clash of Nations[2]. This system uses a blend of GOAP and a hierarchical task network (where a hierarchical task network is described in Section 2.5.1) in a system where motivations[3] create complex goals which can be broken into simple goals and concrete tasks. This allows the creation of multiple separate, partially satisfying plans that, when combined, can completely achieve a goal. Each planning NPC has a set of motivations that generate goals based on rules.

[1]F.E.A.R. - Monolith Productions - http://www.fear3.co.uk/the-game.html
[2]War Leaders: Clash of Nations - Enigma Software
[3]Motivations are described in more detail in Hawes (2011).

Goals can be either compound or primitive. Compound goals are decomposed into less abstract goals. All goals are associated with satisfaction values.

Cerpa (2008) used this system to address the GOAP weakness of each goal or condition being either a binary satisfied or not satisfied, rather than actions being able to partially satisfy conditions. Cerpa also addresses the GOAP weaknesses of having a fixed set of goals because Cerpa's system dynamically generated goals from motivations. This allows the system to work in more dynamic environments where an NPC might not have a fixed set of goals. The use of motivations may generate unexpected combinations of goals, which Cerpa claims may promote emergent behaviour (Cerpa, (2008), page 376).

S.T.A.L.K.E.R.: Shadow of Chernobyl[1] is a first person shooter that uses a GOAP implementation that is optimized so that new plans are only built if any differences were detected between the current state of the world and the world as it was when the plan was made. The game contained 70 actions and used hierarchies of planners in an attempt to reduce complexity.

Fallout 3[2] is a first person shooter that used GOAP with a modification where small state machines could be added to plans. This allowed designers to give more predictable behaviours to the Fallout NPCs while still having some of the dynamism of a planner. Details on these industry implementations are lacking because no academic papers were published describing them.

[1]S.T.A.L.K.E.R.: Shadow of Chernobyl - GSC Gameworld - http://www.stalker-game.com/

[2]Fallout 3 - Bethesda Game Studios - http://fallout.bethsoft.com/eng/home/home.php

A number of variations of GOAP have been created for a wide variety of genres. Each variation addresses some weakness in GOAP, such as Pittman's variation giving more autonomy to individuals in a group or Cerpa's variation that allowed actions to partially satisfy conditions. In the next section, we will discuss more weaknesses that can be addressed by incorporating features taken from systems developed outside of gaming, specifically those that use the concept of utility.

2.5.3 Utility-Driven Action Planning

GOAP has several weaknesses that can be addressed using the concept of utility. This section describes utility and existing uses of utility in behaviour selection systems in computer games.

Utility is a numeric representation of the desirability or usefulness of a state that is used to give preference to states[1] (Ghallab *et al.*, 2004). Utility allows states to be rated based on the expected benefit of a state. It can be helpful for an NPC[2] to evaluate the usefulness of a state so that the NPC can pursue the generation of advantageous states and avoid disadvantageous states.

Weaknesses in GOAP that can be addressed by utility include the inability of GOAP to:

1. create a measure of the degree to which a goal or condition is satisfied

[1]Other research also uses utility to give preferences to actions but the research in this thesis focuses only on giving preferences for states.

[2]An **agent** here refers to any virtual entity that can sense and act upon its sensed information.

instead of only knowing if it is satisfied or not satisfied.

2. measure the full effect of an action by considering how preferable the state generated by the action execution is instead of only considering the number of unsatisfied preconditions in the generated state.

3. find the most contextually appropriate behaviour without the need of action costs.

4. gracefully handle unanticipated situations and environments because utility does not rely on action costs set by designers who may have only had certain situations in mind when assigning these costs.

There are multiple ways utility can be used in an action planner. Utility can be used to assign a level of benefit to each goal. A cost can be given to each action, and the planner can search for the plan with the highest net benefit, calculate as the benefit of the goal minus the cost of the actions required to achieve the goal (Benton *et al.*, 2007). Utility may be used to rate a set of states created by actions and then select a single action associated with the highest rated state for execution (Champandard, 2010). Utility could also be used to represent the preference of a sequence of action executions (Ghallab *et al.*, 2004).

The novel planning system developed as part of our research is based on the way that Mark (2009) uses utility. Mark (2009) describes systems that can use utility for selecting behaviours of NPCs where the behaviour consists of a single action. Mark uses a **preference** to measure the preferability of

a substate of the world state. Instead of a goal being a set of boolean conditions, Mark's system expresses goals as functions that return how close a set conditions is to being complete. Each NPC is associated with a set of preferences tailored by a designer and given at the beginning of the existence of an NPC. Each preference is also associated with a weight that denotes its importance. The utility of a world state is calculated for each NPC preference. Together, this set of preferences are used to calculate the utility of a state generated through the execution of an action. A weighted sum function takes each of the calculated utilities, multiplies them by their respective weights, and adds them together to give a new **final utility**. The world that results from each action the planning NPC can perform is rated using the method described to calculate final utility, and the action associated with the highest utility state is selected for execution. Mark's system helps to overcome the GOAP weakness of only considering one goal at a time because the preference utility functions can consider any number of goals at a time.

The following are examples of using utility for sub-processes within overall agent behaviour selection. Harmon (2002) describes how utility could be used to determine which type of unit another unit is most suited to fight against. Straatman *et al.* (2006) uses utility in the evaluation of tactical positions. Garces (2006) demonstrated how utility could be used with response curves[1], weighted-sum, and a rule-based system to decide if a virtual village should be invaded. Bradley & Hayes (2005) used utility functions with reinforcement learning to teach NPCs cooperative behaviours. Utility was used to evaluate

[1]More details on response curves can be found in Alexander (2002)

potential board states in the turn-based strategy game, Greed Corp[1] . The system used in Greed Corp worked by iterating through all possible actions, calculating the utility for all board states that would result from the execution of the action, and selecting the action with the highest utility. Difficulty in Greed Corp NPCs was easily varied because actions were rated and ordered by utility rating. High difficulty NPCs selected actions with high ratings and low difficulty NPCs selected actions with low ratings.

Planning with utility is useful when there are many actions available because utility functions give an easily understood numeric representation of the usefulness of expected states. Systems using utility have been developed that can handle planning with multiple criteria (Garces, 2006), goals that depend on each other (interacting goals) (Do *et al.*, 2007) and temporal goals (Haddawy & Hanks, 1998). However, to the best of our knowledge, there is no NPC behaviour selection system that uses utility for the formulation of a plan composed of more than one action. The action planning system developed as part of this research uses utility to create plans containing any number of actions.

2.5.4 Smart Objects

This section describes smart objects and how they can be used to produce more contextually appropriate behaviours than would be possible using only GOAP. Kallmann & Thalmann (1998) developed the idea of a **smart ob-**

[1]Greed Corp - W! Games - http://www.wgames.biz, with the utility implementation described in Champandard (2011b)

ject — an object within an intelligent virtual environment that contains more information than its inherent properties (e.g. position), usually containing information about how an NPC can interact with it. For example, a smart object could tell an NPC how to grasp it (Kallmann & Thalmann, 1998), gaze at it (Peters *et al.*, 2003), or which animation to play when using the object (Funge, 1999). Champandard (2007a) describes how smart objects were used in The Sims[1] use smart objects to play animations when interacting with the object. Smart objects have also been used for planning. Kallmann (2001) created implementations of smart objects used for planning by embedding in the object entire plans to achieve a goal involving that object. Later representations used a more flexible and scalable method of STRIPS-like preconditions and effects associated with performing an action with the object (Abaci *et al.*, 2005). Brom (2007) developed upon role passing created by MacNamee *et al.* (2002) by adding a hierarchical task network to create plans for virtual humans using smart objects. A hierarchical classifier system and motivations are used by de Sevin & Thalmann (2005) with a reactive system that can take advantage of newly provided information given by smart objects throughout the process of plan execution.

There are several advantages to using smart planning objects for planning:

- The smart object the action is being performed on can inform the NPC performing the action how the action should be performed. This allows actions to be performed on smart objects in a more specific and contextually appropriate manner. For example, an NPC performing

[1]The Sims - Maxis - http://thesims.com/en_US/home

the goto action on a door smart object is told by the door that the NPC should perform the action by standing very close to the door so the NPC is able to reach the door handle. However, an NPC performing the goto on a person who is a smart object is told by that person object that the NPC should perform the goto action by going to the person, keeping a little distance from person, and to face the person.

- There is a decentralization of logic as logic can be removed from the NPC and placed into the objects. This can make it easier to create and maintain the actions associated with an object.

- Action authoring is conceptually easier with smart planning objects because the designer only needs to consider what an object can do and not whether some particular kind of NPC can do it as that logic is stored within the NPC e.g. a piano only needs to know that someone can play it but not all people will be able to play the piano.

- Smart planning objects can allow faster authoring through action inheritance e.g. a keyboard can inherit all piano actions and then have some of its own.

A weakness of smart objects is that abstract smart objects, such as a weather object, can be hard to conceptualise. However, smart objects are a simple, robust, and tried-and-tested method for implementing game objects.

2.6 Conclusions

Artificial intelligence has a number of uses in computer games. The focus of this research is on the artificial intelligence systems used to select behaviour for NPCs. A wide variety of these behaviour selection systems have been developed, including simple systems, such as purely rule-based systems, and very complex systems, such as cognitive architectures. The simple systems allow for quick behaviour authoring but might be limited in their ability to consider future events and may not scale well. The complex systems may be excellent for creating long term strategies and handling a large number of unexpected behaviours but can be difficult to setup and maintain, particularly with limited time and money. The number of STRIPS-based systems used in computer games suggests that there may be a desirable middle ground in action planning systems used for behaviour selection.

GOAP was the first STRIPS-based action planning system used in a AAA computer game. A number of systems have been developed based upon GOAP to address certain weaknesses of GOAP. Pittman (2008) extended GOAP to allow individuals to overrule orders from superiors if the individual had more contextual knowledge and believed a better plan was available. Cerpa (2008) extended GOAP with motivations to dynamically generate goals, and allowed actions that only partially satisfied a condition, making it possible to perform multiple actions to satisfy a condition. Cerpa's improvements addressed GOAP weaknesses of having a fixed set of goals and having goals and conditions that are either completely satisfied or completely

unsatisfied.

There were other improvements developed independent of GOAP but that can be used to address GOAP weaknesses. Mark (2009) developed a utility-based behaviour selection system that combined information about how an action would effect all goals belonging to an NPC. Abaci *et al.* (2005) created smart objects with planning information embedded within them, creating a more decentralized planning system and allowing objects to specify how an action should be performed in a more contextually specific way.

The research presented in this thesis attempts to create a behaviour selection system that combines the aspects of these systems that address weaknesses of GOAP into one cohesive system and add additional features to address other weaknesses of GOAP. The system created as a result of this research is called Utility-Driven Goal Oriented Action Planning (UDGOAP) and will be described in the next chapter.

Chapter 3

The UDGOAP System

This chapter describes the Utility Driven Goal-Oriented Action Planning (UDGOAP) system, a single NPC behaviour selection system which has been developed for use in static or dynamic environments with a focus on formulating plans that generate states most useful to the **drives** of the planning NPC.

The chapter begins by giving an overview of UDGOAP. The chapter goes on to describe each component and process of the UDGOAP system. The chapter continues with a worked example of how the UDGOAP system runs and ends with conclusions and a summary.

3.1 Overview

Champandard (2011a) performed a survey of commercial computer game de-

velopers that showed that one of the top open challenges for game AI is to have NPCs simultaneously consider multiple goals in a highly dynamic environment. This is challenging because behaviours are often hand crafted by designers for certain world states and the same state might warrant different behaviour depending on the current set of NPC goals. As the goal set of an NPC may include a different set of goals at any time, a designer would have to craft the appropriate behaviour for every state and every combination of goals. This is often too much work so many designers instead fall back to only considering the most important goal, but doing this may overlook behaviours that satisfy multiple goals.

GOAP is a planning system that has been used successfully in a number of commercial games, such as F.E.A.R.[1] and S.T.A.L.K.E.R.: Shadow of Chernobyl[2]. However, GOAP suffers from the limitation that by only considering a single goal, NPCs might attempt to complete one goal at the expense of others, which often leads to undesirable behaviour. For example, the wizard in the situation shown in Figure 3.1 has goals for increasing its health and mana (magic points required and consumed through spell casting). The wizard is in an environment with a potion that increases health by 30, a potion that increases mana by 30 and an elixir potion that increases both health and mana by 20. The wizard is low on both health and mana. Ideally, the wizard would search for plans that achieve both the goal of increasing health and the goal of increasing mana. If the wizard followed a

[1]F.E.A.R. - Monolith Productions - http://www.fear3.co.uk/the-game.html

[2]S.T.A.L.K.E.R.: Shadow of Chernobyl - GSC Gameworld - http://www.stalker-game.com/

behaviour selection system that restricted him to only attempting to achieve one goal, the wizard would have to choose to satisfy the goal of improving either health or mana. If the wizard chose the goal of increasing its health, the best plan for satisfying this goal would be to drink the potion that gives 30 health. A system that only considers a single goal during plan formulation, such as GOAP, would have overlooked the best overall plan of drinking the elixir potion that increases both health and mana. UDGOAP has been designed to address this short-coming.

Figure 3.1: A wizard with the actions of drinking the health potion, drinking the mana potion, or drinking the elixir potion (where the elixir restores both health and mana) available to him.

UDGOAP has several major differences to GOAP and many other single goal planners:

- **UDGOAP attempts to achieve multiple goals simultaneously.** As UDGOAP can consider the effects of an action on a set of goals instead of a single goal and can measure the completeness of each goal instead of just whether the goal is entirely complete or not, the UD-

GOAP planner can be guided to prefer plans that move closest to the achievement of multiple goals, rather than a single goal.

- **UDGOAP considers not just if a condition is satisfied or not, but how satisfied.** The ability of UDGOAP to know both how many goal conditions are unsatisfied and how unsatisfied each condition is makes it possible to know how close to achievement a goal is after the execution of an action or sequence of actions. This makes it possible to know that a behaviour that nearly achieves a goal is better than a behaviour that has no impact on a goal.

- **UDGOAP can chain multiple actions together to satisfy a single precondition.** There may be situations where an NPC cannot perform a single action to satisfy some precondition but where chaining multiple actions will satisfy it. Consider the following example. The planner is looking for a plan to satisfy the (target_is_dead, true) precondition. The shoot action in GOAP would have the (target_is_dead, true) effect because this action should be considered when trying to kill a target. However, a single shot might not be enough to kill an enemy, but the planner must assume it will be as the planner would otherwise not consider the shoot action when trying to satisfy the (target_is_dead, true) precondition. The result is a plan that likely won't actually kill the enemy but will still result in a sensible plan. However, by instead making the precondition that the health of the enemy become zero and making it so an action can partially satisfy a precondition, the planner

could know that the current gun available will take, say, half of the health of the enemy, and would know to follow up with another action, such as a melee.

- **UDGOAP uses smart objects that tell the NPC how to perform the action in a more contextually appropriate way.** For example, when the goto action is performed on a door, the NPC performing the action should move right up close to the door so that the door handle is easily within reach. When the goto action is performed on a person, however, the NPC performing the action should go to the person but stay far enough away to not invade their personal space.

- **UDGOAP integrates object selection with plan formulation.** In GOAP, each goal is associated with some domain-specific subsystem that, before plan formulation, determines which particular object is best for achieving the goal. The GOAP planner only considers that object when trying to achieve the goal. In UDGOAP, the UDGOAP planner considers a set of objects, rather than a single object, and the set is selected during plan formulation, rather than before plan formulation. By considering a set of objects, the planner may discover that although a particular object seemed best at first glance, developing the plan a little unveiled that a different object was best. By delaying object selection until during plan formulation rather than before it, the planner can consider objects that might be better for the state the planning NPC would be in at the time of actually executing the plan

rather than its state at the beginning of the plan. For example, after a goto action has been performed, the potion nearest the planning NPC might not be the one nearest to the NPC before the goto was executed. By delaying the selection of which potions to consider until this point in the plan, a better plan might be found.

- **UDGOAP simulates the world that the planner believes would exist after executing an action or sequence of actions, rather than only considering more abstract key-value pairs.** GOAP builds a set of key-value pairs that represent the world in which the planning NPC exists. These key-value pairs are derived from facts about the world that the NPC has learned through its sensors. Working with these key-value pairs make plan formulation faster at the expense of abstracting over some lower-level details that were present in the facts known top the NPC. Gaming hardware has become more powerful since the creation of GOAP and can now consider more details during plan formulation. As a result, the UDGOAP planner can work directly with a detailed fact set during plan formulation. This fact set makes more information available during plan formulation than the GOAP planner key-value pairs do. UDGOAP can use these facts, combined with information on how actions alter facts, to accurately simulate the state that would exist if an action was executed.

The purpose of the UDGOAP system is to formulate plans that consider multiple goals for a single NPC in a highly dynamic, object rich environment.

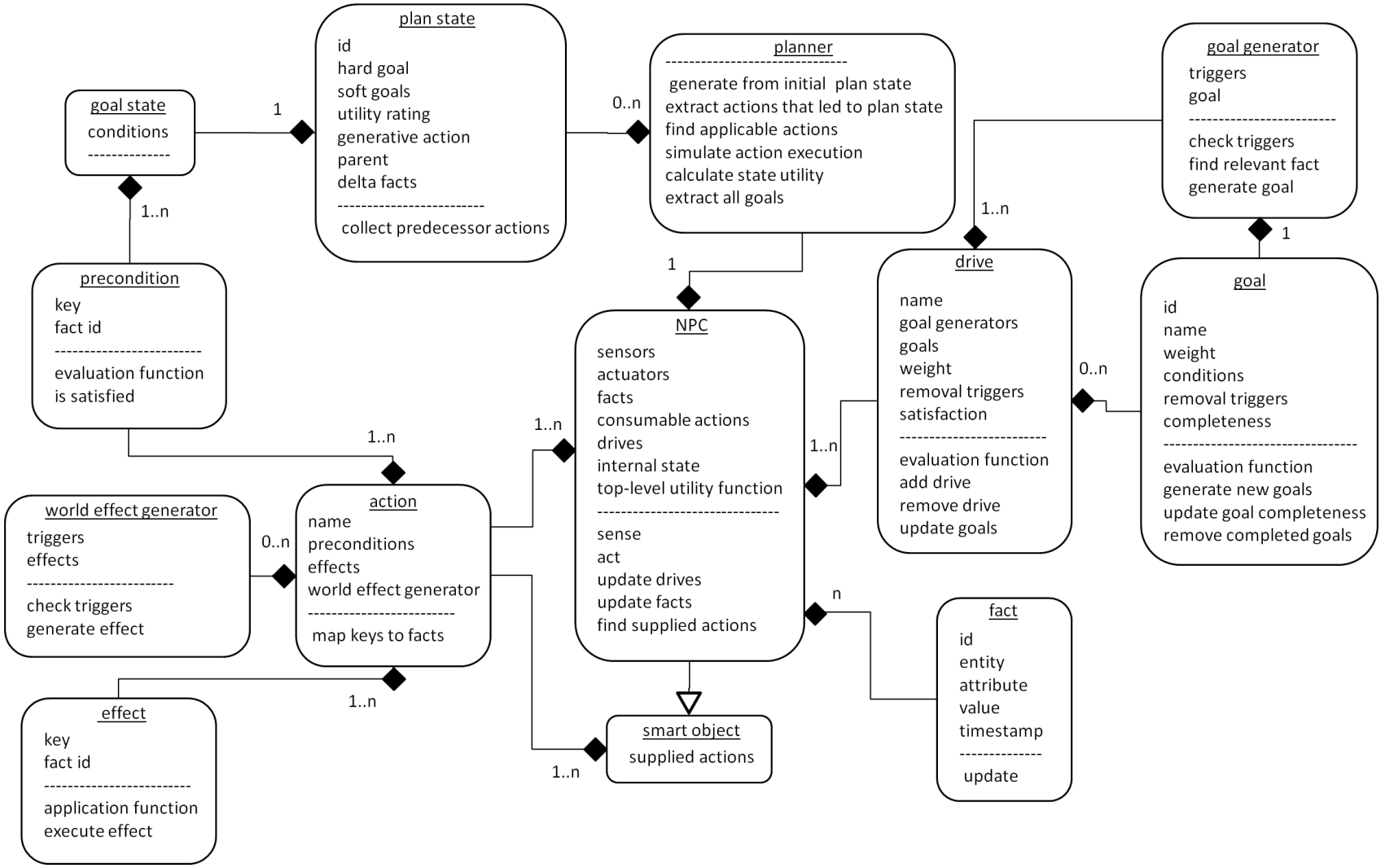


Figure 3.2: A class diagram of the UDGOAP system. Rounded rectangles represent classes. The underlined words at the top of these rounded rectangles denote the class name. Each line of text above the dashed line within the class represents a data member of that class. Each line of text below the dashed line represents a function belonging to that class. Lines with filled diamonds denote a has-a relationship which may be either one-to-one or one-to-many and the arrow denotes inheritance.

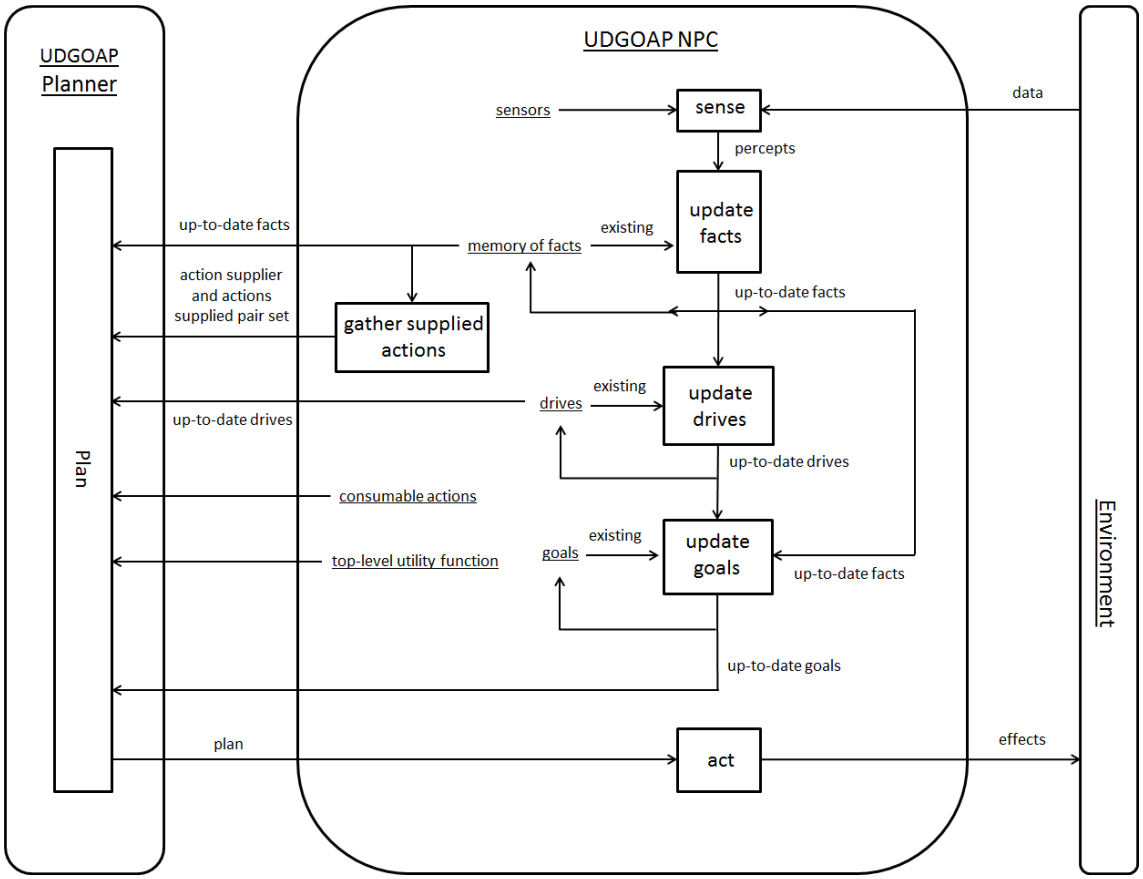


Figure 3.3: An overview of how the different components in the UDGOAP system interact.

Figure 3.2 shows a class diagram of the major components and functions in UDGOAP. Figure 3.3 shows an overview of how the components interact with each other. UDGOAP NPCs receive data from the environment using a set of **sensors** and derive **facts** about the environment from what was sensed. An NPC selects behaviour based on these facts.

Drives are given to the NPC at design-time and steer the high-level behaviour of an NPC. Drives are responsible for generating **goals** for the NPC at run-time based on the set of available facts. The hierarchical relationship between drives, goals, conditions and facts is shown in Figure 3.4. Each drive is equipped with **evaluation functions** that specify how useful a state is

based on the closeness to completion of the generated goals. Goals evaluate their completeness based on the satisfaction of its conditions. Conditions evaluate their satisfaction based upon the facts available to the NPC. Facts are generated from the sensors of the NPC. A top-level utility function takes the satisfaction of all drives as input for generating a state utility. The planner evaluates and selects plans based on the overall utility of the states that the plans lead to, which is calculated using this top-level utility function. The remainder of this chapter will describe each the components of the UDGOAP system in more detail.

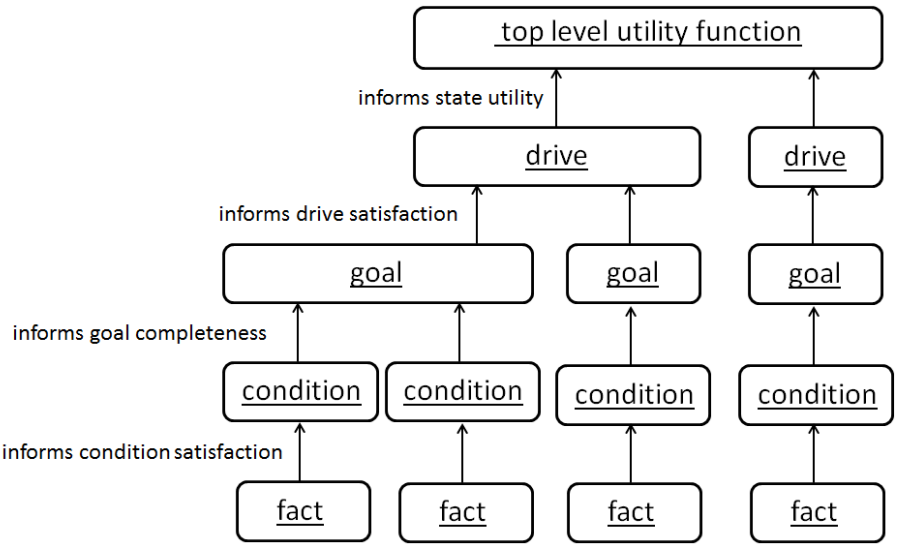


Figure 3.4: A representation of how the hierarchy of facts, goals, drives, and top-level utility functions are connected in UDGOAP. Each goal references a fact. Each drive references the goals it generated. The top-level utility function references the drives.

3.2 Facts and Memory

The world as a UDGOAP NPC knows it is described by facts. A fact in UDGOAP has five parts:

- **ID** — A number for unique identification of a fact.
- **entity** — The entity in the world to which the fact refers. The entity could be an object or NPC and is usually a smart object.
- **attribute** — The attribute of the entity to which the fact refers. For example, the health attribute would mean that the fact refers to the health of the entity.
- **value** — The value of the attribute the fact refers to.
- **timestamp** — The time at which the fact was last updated.

For example, an NPC might have a fact regarding its own health in the form (ID: 48, entity: hero1, attribute: health, value: 50, timestamp: 1.2), which indicates that the NPC that is called hero1 has a value of 50 for its health attribute. Each NPC has a set of facts that represents everything it knows about the world. The timestamp is used to give preference to more recent facts and to make it easier to garbage collect facts when there are too many. The fact set is updated every time the NPC queries its sensors. The newest set of facts, as well as the previously known facts that were stored in memory, are used to inform which behaviour will be selected by the NPC.

3.3 Drives

A drive is high level director of NPC behaviour. This direction is performed through the pursuit of goals that are generated by **goal generators** belonging to drives. For example, the kill_all_enemies drive of the wizard could have

one goal to kill one goblin and another goal to kill another goblin, both generated upon seeing the goblins. Each drive has of the following components:

- **name** — The unique identifier of the drive.
- **goal generator** — A set of rules that will create goals that, if achieved, will increase the satisfaction of the drive.
- **goals** — The set of goals generated by the goal generator of the drive that will improve the satisfaction of the drive when one of its goal is nearer to achievement.
- **weight** — A number used to represent the importance of the drive. The weight may be used in the top-level NPC utility function.
- **removal triggers** — Triggers activated when certain conditions are satisfied, resulting in the removal of the drive from the set of drives belonging to an NPC.
- **satisfaction evaluation function** — The function that calculates and returns a normalised value denoting how satisfied the drive is based on the degree to which the goals of the drive are achieved.
- **satisfaction** — Stores the value output from the satisfaction utility function.

A **goal generator** can create goals to be pursued by an NPC. Each drive is associated with one goal generator. Each goal generator is associated with a set of trigger-goal pairs. When a trigger is activated at run-time, the

corresponding goal is added to the goal set of the drive to which the goal generator belongs. Triggers are activated by the presence of certain facts, e.g. an enemy has been spotted.

The goal generation process takes the newest set of facts, the previous (or old) set of facts, and a goal generator for some drive d. The condition of the goal generator triggers are checked for satisfaction against the available facts. When a trigger is activated, the goal generator to which the activated trigger belongs runs a process to find the fact relevant to the goal that is about to be generated. For example, if the goal generated will be to keep a particular ally alive, the fact relevant to that goal is the current health level of the ally. The goal associated with the goal generator is then generated for use with the relevant fact. The newly generated goal is then added to the drive d if it is not already in the list of goals for drive d. A newly generated goal has its completeness evaluated after it is added to the set of goals of the drive responsible for the generation of the goal.

For example, the kill_all_enemies drive has a goal generator with a trigger that activates upon the presence of a fact that an enemy has been sensed and that the NPC does not already have a goal to kill that NPC. For a second example, the maximize_mana drive has a goal generator with a trigger that activates when the mana of the NPC is less than its maximum.

Each drive also has a satisfaction function that can evaluate how useful a particular state is to the goals it has generated. The completeness of a goal depends on facts available to the NPC. The drive satisfaction function considers the completeness and weight of the goals it generated and returns a

value from zero to one. The satisfaction function could be something simple, such as a weighted sum, or something more complex and domain specific.

Consider the example of an NPC with the kill_all_enemies satisfaction function for a drive that only has the goal of reducing the health of a particular goblin to zero and the fact associated with the goal regards the health of the goblin. After applying the effects of the attack action, the health of the goblin is 50. If the drive satisfaction function is updated, the normalized current value of the health of the goblin is 0.5, indicating that the current satisfaction level of the drive is 0.5.

3.4 Goals

Goals in UDGOAP have the purpose of increasing the satisfaction of the drive from which they were spawned, where a state in which the goal is closer to achievement is better. Each UDGOAP NPC has a set of goals it endeavours to achieve. Each UDGOAP goal has the following components.

- **ID** — The unique identifier of the goal.
- **name** — The name of the goal, e.g. kill_enemy.
- **conditions** — The set of conditions required to achieve the goal. For example, the goal of killing an enemy may have a condition that the health of the enemy is zero. Each goal must have at least one condition.
- **weight** — A number used to represent the importance of the goal. The weight may be used in the evaluation function of the drive to which the

goal belongs. Goals belonging to the same drive might have different weights. For example, the goal of killing a bomb slinging goblin might be weightier than a goal to kill a sword wielding goblin.

- **removal triggers** — Triggers activated when certain conditions are satisfied, resulting in the removal of a goal from the goal set of the drive that owns the goal. Removal triggers are used to remove goals that are complete or no longer relevant.

- **evaluation function** — The function used to calculate the degree to which the goal is complete based on the set of conditions relevant to the goal. The level of completion of a goal ranges from zero to one, where zero represents a goal with no progress towards its completion and one represents a goal that is complete and fully achieved. For example, the evaluation function of a goal to kill an enemy would return a value of zero when the associated enemy has full health and a value of one when the enemy has zero health.

- **completeness** — Holds the completeness value output by the evaluation function.

Just as there are triggers that generate goals, there are triggers that destruct existing goals. The removal triggers of a goal specify when a goal should be removed from the goal set of a drive. For most goals, this is simply when the completeness of the goal reaches one. For example, the goal of killing a particular enemy has a completeness value of one when the fact referring to the health of the enemy is associated with a value of zero. After

the drive updates the completeness value of each goal, it checks if any of the removal triggers are satisfied. For example, the kill_all_enemies drive could generate a goal that is achieved and destructed when the health of the NPC for whom it was generated reaches zero. Alternatively, the maximize_health drive could generate a goal of maximizing the health of the planning NPC and has no destruct condition because maximizing NPC health will always be a concern to the NPC.

3.5 NPCs

An NPC in the UDGOAP system is responsible for sensing the world using sensors, building and storing a set of facts to represent the world, keeping its drives up to date, keeping its goals up to date, and enacting behaviours returned from the UDGOAP planner. Each UDGOAP NPC has the following components:

- **ID** — The unique identifier of the NPC.
- **sensors** — The set of sensors through which the NPC can sense its environment.
- **facts** — The set of facts known to the NPC about itself and its environment.
- **consumable actions** — The set of actions the NPC is able to perform (described in more detail in Section 3.7).

- **supplied actions** — The set of actions that other NPCs are able to perform on this NPC (described in more detail in Section 3.7).
- **drives** — The set of drives that direct the behaviour of the NPC.
- **internal state** — The set variables that describe the information available to the NPC about itself, e.g. its current health level.
- **top-level utility function** — The utility function that is used to determine how preferable a world state is based on the weighted satisfaction of the drives belonging to the NPC.

These NPC processes are represented in Figure 3.3. This section details how each of these processes work as part of the UDGOAP system.

3.5.1 Sensing

The NPC receives data about its own internal state and data from the environment that it is able to sense with its set of currently active sensors. The NPC creates a set of facts, described in Section 3.2, about the percepts (data generated by sensors) it has sensed. The facts generated from sensor information are combined with the existing facts from the memory of the NPC, including information about the current internal state of the NPC, e.g. the health of the NPC. This combination yields the most up-to-date set of facts available to the NPC, which is stored in memory to be used in the next planning iteration. The sensing and fact update processes are the first and second processes in the NPC update process, as shown in Figure 3.3. The up-to-date facts are used to update the NPC drive set.

3.5.2 Updating Drives

The process of updating drives has three phases.

1. Add new drives given by some outside source.

2. Remove drives that are no longer applicable to the NPC.

3. Update the goals belonging to each drive.

There are two ways in which an NPC receives drives - internally and externally. Internal drives are assigned to a particular type of NPC at design-time. For example, a soldier starts with the kill_all_enemies drive but a medic does not. External drives are given to an NPC by other world entities at run-time. For example, a commanding officer could give an order to a soldier NPC that it must protect a particular medic. An annotated map could also add drives. For example, when an NPC reaches a certain point in the map, it should man a machine gun. All orders to add a new drive to the existing drive set comes through the sensors of an NPC.

Each drive has a set of removal triggers that are activated upon the satisfaction of a particular condition, just like goal removal triggers. Such a condition being satisfied will result in the drive being removed from the list of drives for an NPC. Most internal drives would not have a removal trigger because it might not make sense to stop being concerned about. For example, there is no removal trigger for the goal of maximizing health because that the NPC should be concerned with that goal for its entire lifetime. External drives are more likely to be removed when they are no longer relevant. For

example, a commanding officer might add the drive of defending a base from an attack but the drive is no longer relevant when the base is no longer under attack. When a drive is removed, so are all goals generated from the goal generators of that drive.

The process of updating drives comes after the sensing process and before the planning process of the NPC update process, as shown in Figure 3.3. Updating goals belonging to a drive is part of the process of updating a drive.

All goals in UDGOAP belong to the drive that generated them. In order for drives to accurately evaluate the utility of a state, each goal must know how complete the goal would be for a particular world state. The process of keeping drive goals up to date has three steps:.

1. Generate new goals.

2. Update goal completeness.

3. Remove complete goals.

The drive update process requires:

- the input of the up-to-date set of facts.
- the old set of facts that were the most up-to-date in the previous sensing iteration.
- the set of goal generation triggers.
- the set of existing goals.

The output of the process are the goals that have been removed, and the goals that were not destructed, which become the new goal set for the NPC.

The first step in the goal update process is to check which goal generation triggers have been activated. The up-to-date set of facts is used to check triggers that only depend on the current state, e.g. the current mana level of the planning NPC. The newly generated goals are added to the goal set of the drive responsible for their creation.

Once this is done then the completeness of each goal in the goal set is updated. Each goal has its own evaluation function that calculates the completeness of the goal as a normalized value. These functions can be simple linear functions or more complicated response curves, as described by Alexander (2002). The evaluation function type most frequently used throughout this research was the inverse linear function. This works by normalizing a variable and subtracting the normalised value from 1.0. For example, when being used the calculate how complete the goal of killing an enemy is, the current health of the enemy is divided by the maximum health of the enemy and the result is subtracted from 1.0.

After all goals have had their completeness updated, the next part of the process is to see which goals should be destructed and which should remain. Each goal is associated with a set of removal triggers that indicate if a particular goal should be removed from the set of NPC goals. All triggers on all goals are checked. Any goal that does not have a removal trigger activated remains part of the goal set of the drive responsible for its creation.

3.6 Smart Objects

Actions in UDGOAP can only be applied to **smart objects** (described in Section 2.5.4). UDGOAP uses smart objects to reduce the number of objects the planner needs to consider, to perform actions in suitable manners, and to help find better plans.

Smart objects help reduce the number of objects the UDGOAP planner must consider by encapsulating actions and advertising to the planner which actions can be applied to them. Actions not in this list cannot be applied to the smart object. For example, a potion smart object advertises that the drink action can be applied to the potion. A door smart object does not contain nor advertise the drink action so although the planner might consider applying drink to the potion, it wouldn't consider applying drink to the door. Smart objects help NPCs perform actions in a a way that is more suitable for the object by having the object specify how to perform the action. For example, the NPC can perform the goto action, but if performed on a door, the NPC should stand directly in front of the door, facing it, and close enough to open it. However, if the goto action is performed on another person, the performing NPC should get close to the person but not so close as to invade their personal space.

UDGOAP has two types of smart objects with which it is concerned: objects and NPCs. Objects are any entity to which actions are applicable. Each UDGOAP object has a set of actions it supplies. For example, a door supplies the open_door, close_door, lock_door and unlock_door actions. Objects

cannot perform actions — objects can only have actions applied to them. For example, a door can not open of its own accord. A UDGOAP NPC is a type of smart object that can both apply actions and have actions applied to it. Each NPC has a set of actions that it can supply and a set of actions it can consume, called the **supplied action set** and **consumable action set**, respectively. For example, the knight supplies the heal action, indicating that other NPCs can apply that action on the knight. The knight may not have the heal action in its consumable action set, meaning that it cannot heal itself or others. However, the wizard NPC has the heal action in its consumable action set. The set of actions available to UDGOAP NPCs and objects are decided at design-time, but the applicability of these actions may change at run-time.

3.7 Actions

An action is something that can be performed by an NPC to change the world. Each UDGOAP action has the following components:

- **name** — A unique identifier of the action.
- **preconditions** — A set of functions returning a value between zero and one that must be satisfied (have a value of one) in the world state for the action to be executable.
- **effects** — A set of changes that will become part of the world state after the execution of the action. Each effect has an application function that

returns a fact that would exist if that fact was applied to another fact.

- **supplier** — The smart object upon which the action is being performed. It is called the supplier because the smart object made the action available for application.

- **consumer** — The NPC performing the action. It is called the consumer because it uses the action made available by the action supplier.

- **world effect generator** — A function that can dynamically add additional effects to an action based on the current world state (described in more detail in Section 3.8.5).

The preconditions associated with a UDGOAP action indicate the predicates that must be true before that action can be performed. A UDGOAP action precondition consists of a key, a fact ID, and a predicate evaluation function. A key describes how the precondition might be satisfied and takes the form entity_attribute_change. For example, a precondition for the heal action is that the action consumer has the required amount of mana, so the key to describe how to satisfy that condition is consumer_mana_increase. The fact attribute and fact ID are used to find which fact the precondition applies to, described in more detail in Section 3.8.4. The key is also used to reduce the number of actions that need to be considered during plan formulation, as described in Section 3.8. The evaluation function tests if some predicate is true for a particular fact, e.g. if the NPC consuming the heal action has enough mana to execute the action. Each precondition evaluation function is made by a designer specifically for each action.

The effects associated with a UDGOAP action describe how the world will change upon the execution of the action. A UDGOAP action effect consists of a key just like that used in the UDGOAP precondition, and an application function. The key is used during the planning process to reduce the number of actions considered during plan formulation and in the process of mapping keys to facts. The process of mapping keys to facts is described in Section 3.8.4. Effect application functions calculate how a fact would change as a result of the application of the effect. For example, one of the effects of the heal action increases the health of the action supplier. The application function could take the fact associated with the current health of the action supplier and return the same fact but with an increased health value of 20. Each effect application function is made by a designer specifically for each action.

Each action must have both a supplier and a consumer. For example, the wizard unit has a heal action that decreases the mana of the wizard, and increases the health of the target of the heal action, e.g. a knight. When the wizard applies the heal action to the knight, the heal action supplier is set to the knight and the heal action consumer is set to the wizard. The wizard can apply the heal action to a knight only if the knight has the heal action in its supplied action set and the wizard has the heal action in its consumable action set.

3.8 Planner

The UDGOAP planner is a regressive planner (previously described in Section 2.5.2.1) that creates plans that create a world state of maximum utility to the NPC. The utility of a state is increased by achieving goals belonging to the NPC drives. To improve the accuracy of the estimated utility of a state, the UDGOAP planner executes actions in a simulated world and evaluates the utility of the world it believes will exist after the action execution. A representation of the UDGOAP planner is shown in Figure 3.5. The UDGOAP planning algorithm is shown in Algorithm 3.

The output of the UDGOAP planning algorithm is the plan that maximizes state utility for the planning NPC. The inputs to UDGOAP planning algorithm are:

- initial_facts — The set of facts in the fact set of the planning NPC at the time the planning process begins.
- consumable_actions — The set of actions that the planning NPC can perform.
- [(action_supplier, action_supplied)] — A set of tuples where the first part of the tuple specifies which smart object supplied the action and the second part of the tuple is the action being supplied, e.g. (door1, open_door).
- planning_agent — The NPC for whom the plan is being made.

- drives — The set of drives belonging to the planning NPC.

- top_level_utility_fn — The top-level utility function being used by the planning NPC.

Everything from lines 9 to 22 in Algorithm 3.5 is for generating successor plan states to be considered further during planning. A **plan state** in UDGOAP is a snapshot of the world and the state of the planning NPC along with several components of the NPC relevant to the plan, where a new plan state is generated at the beginning of plan formulation and through the application of an action[1] during planning. Planning can be represented as a graph where plan states are nodes and actions are edges that cause transitions from one plan state to another. The root node in this graph would be initial plan state and leaf nodes would represent the most developed plans in this graph. A plan state has the following components:

- ID — A unique identification number.

- hard goal — A set of conditions that must be satisfied as part of plan formulation. Failure to find a solution for this goal will result in plan failure. A hard goal in UDGOAP is equivalent to a goal state in GOAP.

- soft goals — A set of soft goals, where each soft goal is a goal that is desirable to satisfy but if unsatisfied, will not result in plan failure. These goals are used when calculating the utility of a state.

[1]To make the explanations easier to follow, whenever an action is just being simulated, we will say the action is being **applied**. Whenever the action is being performed by the agent, we will say the action is being **executed**.

```
   Input: initial_facts, consumable_actions, [(action_supplier,
          action_supplied)], planning_agent, drives, top_level_utility_fn
   Output: plan
 1 plan_states = generate_from_initial_state(initial_facts,
                          extract_all_goals(drives), consumable_actions,
                          [(action_supplier, action_supplied)])
   /* begin planning main loop                                          */
 2 while plan_states ≠ ∅ do
 3     best_plan_state = plan_states.argmax((s) ⇒ s.utility)
 4     plan_states -= best_plan_state
 5     unsatisfied_pre = select_unsatisfied_precondition(best_plan_state)
 6     if unsatisfied_pre = null then
 7         return extract_actions_that_led_to_plan_state(best_plan_state)
 8     else
           /* begin generating successor plan states                    */
 9         applicable_actions =
           find_applicable_actions(unsatisfied_precondition,
           consumable_actions, [(action_supplier, action_supplied)])
10         actions = ∅
11         for (action, supplier) ∈ applicable_actions do
12             actions += map_keys_to_facts(action, initial_facts)
13         end
14         unrated_successor_states = ∅
15         for a in actions do
16             unrated_successor_states += simulate_action_execution(a,
               best_plan_state)
17         end
18         rated_successor_states = ∅
19         for s in unrated_successor_states do
20             rated_successor_states += calculate_utility(s, drives,
               top_level_utility_fn)
21         end
           /* end generating successor plan states                      */
22         plan_states += rated_successor_states
23     end
24 end
   /* end planning main loop                                            */
25 return empty plan
```

Algorithm 3: The UDGOAP algorithm.

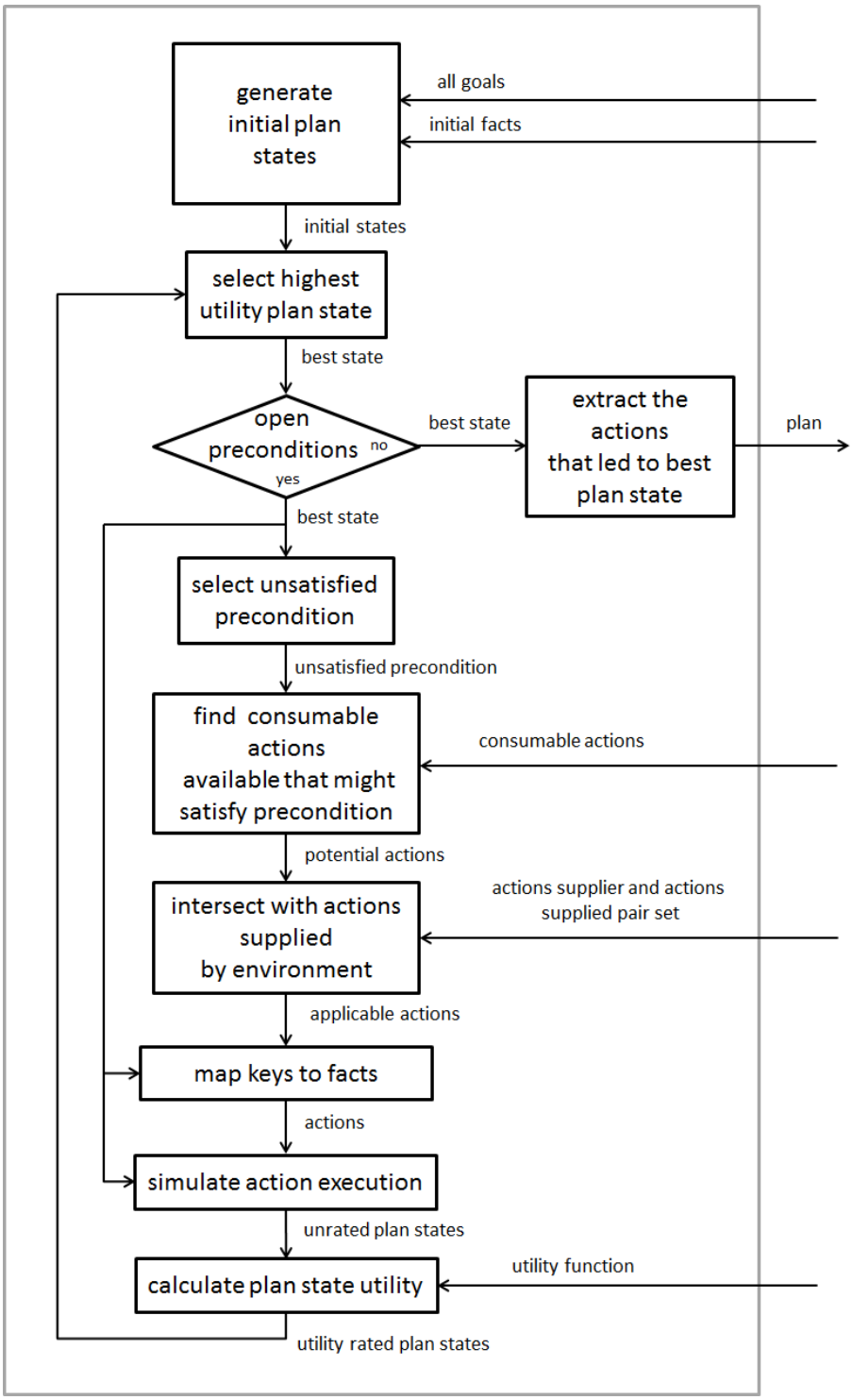


Figure 3.5: A representation of UDGOAP plan formulation.

- utility rating — A rating to represent how useful the world state represented in the plan state is to the set of drives of the planning NPC, where higher is better.

- generative action — The action responsible for the generation of the plan state.

- parent — The plan state from which this plan state was generated via an action.

The collect_predecessor_actions function generates a list of the actions that led to a plan state. This is done by iterating through each of the predecessors of a plan state and collecting their generative actions. This list of actions is used later during the process of rating a plan state.

The rest of this section will describe the processes involved in UDGOAP plan formulation:

1. Root plan state generation.

2. Best state selection and goal completion.

3. Finding applicable actions.

4. Mapping action precondition and effect keys to facts.

5. Action application and world effect simulation to generate new plan states.

6. Evaluating the utility of newly generated plan states.

3.8.1 Root Plan State and Successor State Generation

The **initial state** is the state of the world state that existed at the when the planning process initiated. The generate_from_initial_state function (called on

line 1 in Algorithm 3) generates a new plan state, based on the initial state, for every action the NPC can perform that has an effect key that matches a condition key of any goal. This includes the processes of finding applicable actions, mapping action preconditions and effects to facts, simulating action execution to produce new plan states, and evaluating the utility of the newly generated plan states.

The function first generates a root plan state from the initial state: s_{root} = new PlanState(ID: 1, hard_goal: null, soft_goals: all_goals, rating: 0, action: null, parent: null), where initial_facts is the set of facts in the fact set of the planning NPC at the time the planning process begins, and where all_goals is the set of all goals belonging to all drives of the planning NPC that were collected in the extract_all_goals function (line 1). This root plan state becomes the parent of the remainder of the plan states generated in this function.

A plan state is generated for each action in the consumable action set with an effect key that matches a condition key of any goal in the set of all goals of the planning NPC. For example, if wizard1 had the melee action that it could apply to goblin1, the following plan state would be generated:

- s_1 = (ID: 2, hard_goal: (fact key: supplier_health_decrease, fact ID: 5, evaluation function: fn_linear_minimize), soft_goals: all_goals, rating: 0.9, generative action: (name: melee, consumer: wizard1, supplier: goblin1, ...), parent: 1)

where the fact with the ID of 5 refers to the health of goblin1. The successor plan state of the root plan state associates itself with its parent

using a plan state ID, which is set to 1 as that is the plan state ID of the root plan state. The set of soft goals of the initial successor plan state is the set of all goals belonging to all drives. This is so all drives are considered during planning.

3.8.2 Best State Selection and Goal Completion

After the successors to the root plan state have been generated and added to the set of plan states, the main loop of the planner is entered (line 2) with the purpose of developing plans states in the search for the plan that leads to the state with the highest utility to the set of drives of the planning NPC.

The main planning loop begins by selecting the best (highest utility) plan state from the set of plan states for further development (line 3). The best plan state is the plan state associated with the highest utility.

If the goal associated with the selected plan state has no unsatisfied conditions and is not the initial state, the goal state is reached and plan formulation is complete. The complete plan is extracted from the plan state by extracting all of the actions that had to be performed to reach that plan state (lines 6 and 7). These actions would be returned from the planner to the UDGOAP NPC who would execute them as behaviour.

3.8.3 Finding Applicable Actions

If a goal state has unsatisfied conditions, the planning process searches for actions that will satisfy an unsatisfied condition using the find_applicable_actions function (line 9). The function to find these applicable actions is shown in

Algorithm 4. The inputs to the algorithm are:

- unsatisfied_precondition — the unsatisfied precondition belonging to the highest utility plan state.

- consumable_actions — the set of actions the planning NPC is able to perform.

- [(action_supplier, action_supplied)] — the set of pairs of all actions being supplied by the smart objects in the environment, where the first part of the pair is the smart object supplying the action and the second part is the action being supplied. For example, this set might be {(knight1, heal), (knight1, melee), ...}, meaning that the heal and melee actions can be applied to knight1.

In Algorithm 4, the planner searches through the set of consumable actions of the planning NPC for an action with an effect key that matches the key of the unsatisfied condition being considered. This yields the set of applicable actions that the action consumer (the planning NPC) could potentially apply. These potentially applicable actions require a smart object that supplies the potentially applicable actions.

The intersection of actions the planning NPC can consume and the actions that the smart objects in the environment supply is the set of actions applicable to the unsatisfied precondition (line 11). The next step is to map from precondition and effect keys to precondition and effect facts (line 12).

```
Input: unsatisfied_precondition, consumable_actions, [(action_supplier,
       action_supplied)]
Output: applicable_actions
potential_consumable_actions = ∅
for a ∈ consumable_actions do
    if a.effects ∩ unsatisfied_precondition ≠ ∅ then
        potential_consumable_actions += a
    end
end
applicable_actions = ∅
for (supplier, supplier_action) ∈ [(action_supplier, action_supplied)] do
    if supplier_action ∩ potential_consumable_actions ≠ ∅ then
        applicable_actions += new Action(
                                  preconds: supplier_action.preconds,
                                  effects: supplier_action.effects,
                                  consumer: planning_agent,
                                  supplier: supplier,
                                  world_effect_gen:
                                  applicable_action.world_effect_gen)
    end
end
return applicable_actions
```

Algorithm 4: The UDGOAP planner function to find applicable actions.

3.8.4 Mapping Action Keys to Facts

Algorithm 5 shows the function for mapping the precondition and effect keys of an action to facts. The inputs to the algorithm are:

- unmapped_action — the action with preconditions and effects that are not yet associated with any facts. This action is already associated with a supplier and consumer when it was created during the phase of finding applicable actions.
- best_plan_state_facts — all of the facts known to the NPC at the time the plan state refers to.

```
   Input: unmapped_action, best_plan_state_facts
   Output: actions
 1 mapped_action = unmapped_action.clone()
 2 for precond ∈ mapped_action.preconds do
 3 |   precond.fact_id = key_to_fact(precond.key, mapped_action.consumer,
 4 |                       mapped_action.supplier, best_plan_state_facts)
 5 end
 6 for effect ∈ mapped_action.effects do
 7 |   effect.fact_id = key_to_fact(effect.key, mapped_action.consumer,
 8 |                     mapped_action.supplier, best_plan_state_facts)
 9 end
10 return mapped_action
```

Algorithm 5: The UDGOAP planner function for mapping keys to facts.

The function first maps each precondition of the passed action to its relevant fact and then each effect of the passed action to its relevant fact. This mapping is done to provide each precondition with the fact needed to evaluate if the precondition is satisfied, and to provide each effect with the fact needed to know which fact the effect will alter. The key_to_fact function is an implementation specific mapping from keys to facts.

An example of the mapping of precondition and effect keys to facts follows. There is a wizard NPC, wizard1, that has the goal of increasing the health of a knight, knight1. The knight has the heal action in its supplied action set and the wizard has the heal action in its consumable action set. The heal action is selected by the planner belonging to the wizard for application. The precondition and effect keys are mapped to facts in the set of facts available for the current plan state.

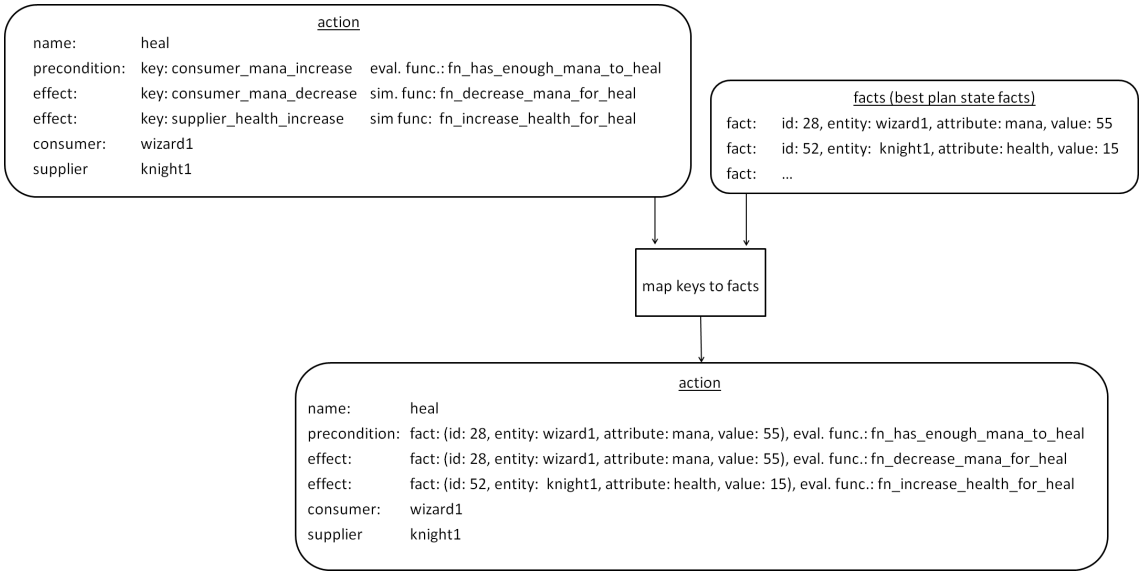


Figure 3.6: The process of mapping action precondition and effect key to their relevant fact.

As shown in Figure 3.6, the function map_keys_to_facts takes the input of the facts that comprise the world state at that particular time in the planning process and the action that will have its precondition and effect facts determined. The output of the map_keys_to_facts function is the action to be applied, but with the keys mapped to facts so that the precondition evaluation functions can be tested on the current facts and the effect application functions can be used to see how action effects will change facts.

For example, consumer_mana_increase is the precondition key for the heal action, which will cause a lookup for a fact in the set of facts available to the plan state for some fact about the mana level of the wizard, as the wizard is consuming the action. In this instance, the consumer_mana_increase key corresponds to the fact with the ID of 28 because the object portion of the key specifies the consumer, which is wizard1, the attribute portion of the key specifies mana, and the entity referred to by fact 28 matches with the value wizard1 and the attribute referred to by fact 28 matches with the mana attribute.

The precondition of the new action now has a fact that can be used in the evaluation to determine if the action can be applied. This fact will be used later with the fn_has_enough_mana_to_heal function to determine if the heal action is applicable. For example, it might check if the value associated with fact 28 is above the amount of mana required to execute the heal action. The effect of the new action associated with the fn_increase_health_for_heal application function will take fact 52 as input and output the state of that fact will be after the effect has been applied, which in this case is increasing the health of the knight.

3.8.5 Action Application and World Effect Simulation to Generate New States

The UDGOAP planner works by trying to simulate what the world state will be after the execution of a particular action. This is done in an attempt

to give the planner a more accurate view of the world so that better plans can be generated. The simulation tries to compensate for the fact that the actions other agents take are not accounted for during plan formulation. A new plan state will be generated that reflects the effects of the executed action on the previous world state. The new plan state will later have is utility rated against the set of drives of the planning NPC, as described in later in this section. This is similar to how Laird (2001) used a bot in Quake[1] simulated possible futures by predicting what actions the other NPCs in the environment might take.

The plan state is created by first collecting all actions that led to the plan state. These actions are then applied in forward order, as they would be executed in the game, from the initial state, where the initial state is the fact set of the planning NPC at the beginning of plan formulation. Additional effects beyond just the effects of the actions that led to the plan state are simulated to help give a more accurate representation of the plan state.

The planner cannot perfectly simulate what state the world will be in after executing an action because that calculation would be prohibitively expensive. To have a perfectly accurate simulation, the process would have to create a sandbox simulation by copying all the data from the game world, playing the simulation in the sandbox for a few seconds, and then returning the planning NPC fact set for evaluation. Copying the entire game world would double the data requirement of the game. The sandbox world simulation would have to be sped up substantially to finish within an acceptable

[1]Quake - id Software - http://www.idsoftware.com/en-gb

duration. This would require massive computational resources. Furthermore, this sandbox world would have to be created every time a UDGOAP NPC wants to select a behaviour, which could be several times a second. Finally, these computational and memory costs would be incurred for every NPC using UDGOAP. For these reasons, the approach of instead associating each action with an approximate simulation function is taken, where the simulation function includes the application of all effects in the action plus the world effects of the action. As shown in Figure 3.7, there are three steps to simulating action execution:

1. Apply the effects of each action to the plan state that was selected for additional development, called the **state under operation**.

2. Detect any **world effects** triggered by the applied effects.

3. If there are any world effects detected and equilibrium isn't reached, apply the world effects to the plan state under operation and go back to step 2. Otherwise, continue to the next planning step. **Equilibrium** is the state at which there are no more world effects that have been generated.

An example of action simulation follows. There is a knight, knight1, who can engage in melee combat with a nearby goblin, goblin1, using the melee action. The melee action has the following effect: (key: supplier_health_decrease, application function:
fn_reduce_supplier_health_for_melee_attack), where the application function takes the fact associated with the health of goblin1 and returns the same fact but

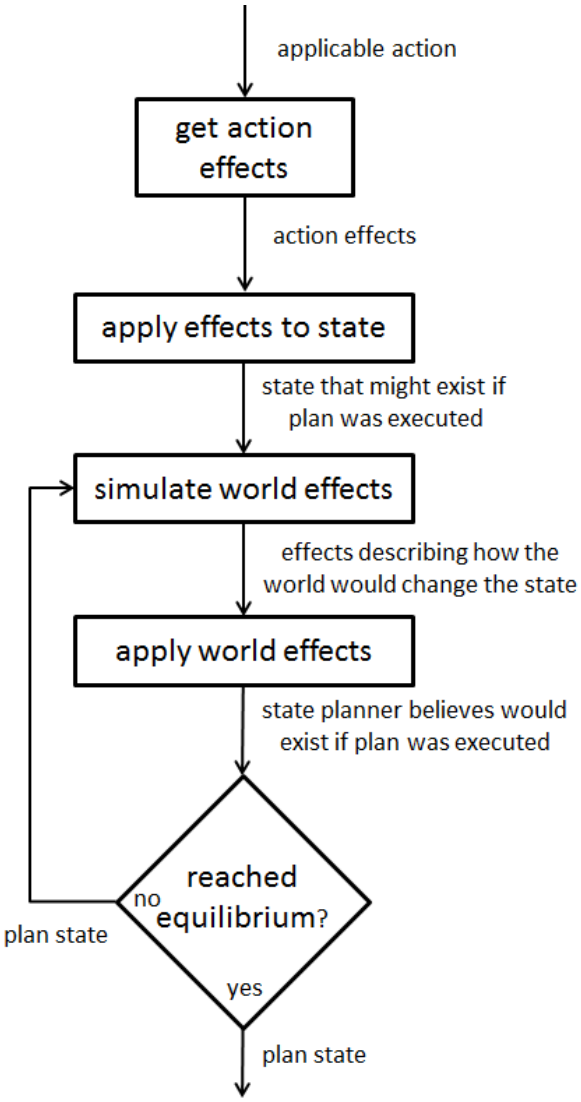


Figure 3.7: Representation of planner simulating an action and calculating the utility of the plan state.

where the value representing the health of the goblin is reduced by the melee damage of the knight, which in this case represents a reduction of 50 health.

The execution of the melee action would generate a new plan state in which the goblin has less health. The difference between this new plan state and its parent plan state are the facts changed by the effects of the melee action and the action that generated the plan state. Each action effect applied may also generate **world effects**, which will change other facts within

the newly generated plan state. World effects are used so that the planner does not assume the world remains static during the execution of a plan.

World effects are what move planner action execution from simple planning to simulation. World effects are the effects entities in the world are likely to produce as the result of an action and are used to get a more accurate picture of the world state after the application of an action or sequence of actions.

A world effect is created by a world effect generator in much the same way a goal is created by a goal generator. A **world effect generator** has a set of triggers and a set of effects. When an action is executed during plan formulation, the set of facts that make up the world state of the plan state are checked against the set of triggers for each world effect generator. If any trigger of a world effect generator is activated, all of the effects belonging to the world effect generator are applied to the world state of the plan state created through the execution of the action. World effect generators are created at design-time.

An example of world effect generation follows. A wizard performs the goto action so that it can melee attack the goblin. The performance of the goto action on the goblin puts the wizard within melee attack distance with a goblin. It can be assumed whenever the wizard within melee attack range of a goblin that the goblin will attack the wizard, so the designer made a world effect generator that is triggered if the distance of a wizard is less than the melee attack range of a goblin. The effect of this world effect generator is to subtract the melee damage of the goblin from the health of the wizard.

By evaluating the world state after the execution of the melee action, the planner discovers that performing the melee action would actually reduce the health of the wizard so much that it would result in the death of the wizard, something that would not have been known without the additional world effect that reduced the health of the wizard. The planner then finds a better way to kill the goblin such as to use a range attack. This plan would not have been considered by only applying the effects of the **goto** action if the planner had assumed the world remained static during plan execution.

World effects make simulations more accurate and cause better plans to be found as a result. However, they make it necessary to run the simulation of action execution in the order that the actions would actually be executed. Therefore, although actions are chained together in reverse order when searching for a solution during plan formulation, the simulation of action execution is performed in forward order.

Reusing the previous example, a wizard has the plan of going to a goblin and then performing a melee attack, which would reduce the health of the goblin to zero. Let's say that there is a world effect generator with a trigger that is activated if an NPC moves to any new location within melee attack range of an enemy and that the effect generated is that the enemies range cause damage to the NPC. For example, if a wizard performs the goto action on a goblin, bringing the wizard within the attack distance of the goblin thus triggering the world effect of the goblin causing damage to the wizard. If the wizard's melee action was simulated before its goto action, the melee action would simulate that the goblin is no longer alive. As a result, when

the goto action is simulated, there is no world effect applied for the wizard taking damage because the goblin is not alive due to the melee action. Hence, the simulated world will need the actions applied in the order they will be performed in, even though the planner is chaining actions in reverse order.

It may be the case that a consumer would die if the actions were applied in forward order. In this case, all actions after the action that causes the consumer to die are ignored and not applied.

The utility of a plan state is evaluated after all of the effects of the actions that led to the plan state have been applied. The process of rating the utility of a state depends on many factors. A representation of the process is shown in Figure 3.8. In short, each plan state holds a set of facts that describes the world. The goals in the plan state use these facts with an evaluation function to determine goal completeness. The completeness and weight of these goals are used in a drive evaluation function to determine the satisfaction of a drive, where the drive function belongs to the drive that generated the goals. Drive satisfaction and weight is passed to a top-level utility function that rates the utility of the state.

For example, a plan state contains facts describing **goblin1** as having 0 health and **goblin2** as having 50. The wizard has one goal to kill **goblin1** and one to kill **goblin2**. Both goals have a weight of 1.0 and use the inverse linear function to rate their completeness, giving completeness values of 1.0 and 0.5, respectively. Both goals belong to the **kill_all_enemies** drive which has a drive evaluation function that takes the completeness of the goals and outputs their average, which in this case is 0.75. This is the only drive this

NPC has so the top-level utility function will output a utility value of 0.75 for this plan state.

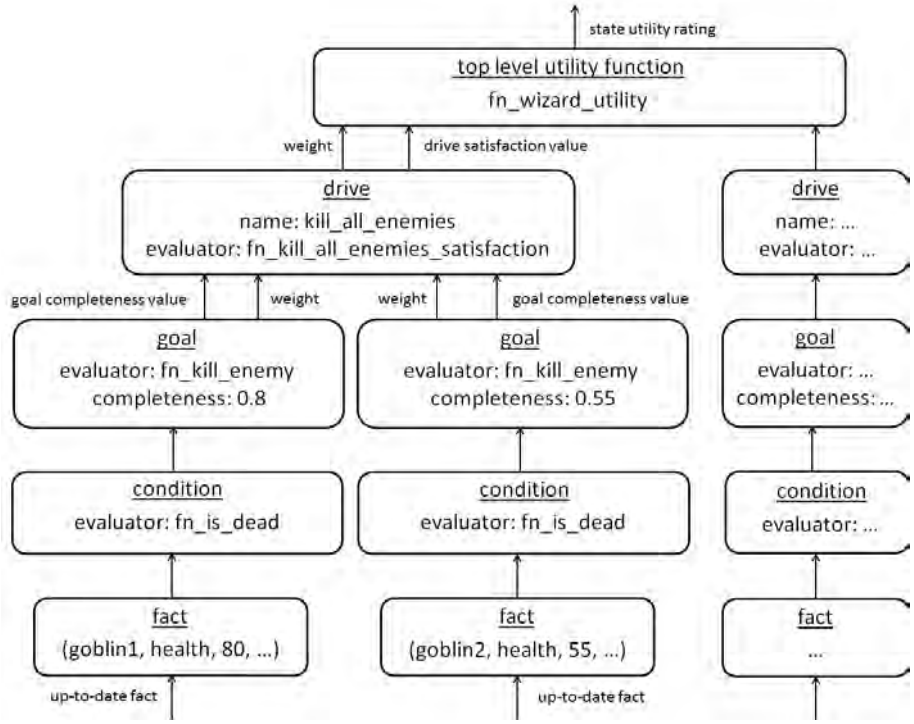


Figure 3.8: Representation of calculating the utility of a plan state.

The completeness of the hard goal and soft goals are used to calculate the level of satisfaction of the drives with which they are associated when rating the utility of a state. The hard goal is a set of conditions that is analogous to the goal during the GOAP planning process. The hard goal will start as the goal of, for example, increasing the health of an NPC, but more conditions will be added to it as additional actions are added to the plan to achieve the goal e.g. have enough mana to heal. A soft goal is a goal that does not require satisfied for the plan to be considered a solution. Moving soft goals toward a more complete state during planning is preferred but not necessary. For example, a hard goal might be to kill an enemy but a simultaneously considered soft goal is maximizing the amount of mana the wizard has. The

planner will then prefer plans that minimize the amount of mana used while killing the enemy but the planner will still consider plans that use mana as valid solutions so long as the plan results in the enemy being killed.

Every goal in UDGOAP, hard or soft, is associated with a function that is used to evaluate the completeness of the goal. This function can be a simple, linear function or something more complex, such as a response curve (Alexander, 2002). For example, a knight has the goal to kill a particular goblin. The function for evaluating the completeness of this goal takes the fact associated with the health of the goblin as input and outputs a normalized value describing how complete the goal of killing the goblin is, where 0.0 could mean the goblin is on full health and 1.0 means the goblin has no health. For example, the goal evaluation function used for this scenario could be a simple linear function that will return a normalized inverse of the maximum health of the goblin. If the goblin has a maximum health of 100 and is currently on 60 health, then the goal is 0.4 complete. This normalized value is stored in the goal to be used during the calculation of the the utility of the state to the NPC based on the drive that generated the goal, e.g. the kill_all_enemies drive.

A drive has an evaluation function that determines the satisfaction of the drive by considering the completeness of the goals the drive generated. A state will have a utility rating from zero to one for a particular drive, where zero means all goals associated with that drive are in no way complete and one means all goals are fully complete. Each goal in UDGOAP is associated with a weight that is used as part of a weighted sum calculation to compute

state utility. Goals from the same drive can have different weights. For example, the kill_all_enemies drive could have one goal to kill a melee goblin where the weight of the goal is 1.0, and a second goal to kill a ranged attack goblin with a higher weight of 2.0 because ranged attack goblins are more dangerous.

The weighted sum output of each drive is then fed into the NPC utility function as input and the utility of the entire plan state is output. This output is assigned as the utility of the plan state from which it was generated. This utility function is likely to be more complicated than that of a single drive utility function and likely to be very domain specific. For example, the maximize_health drive might be considered particularly important and so when it hits zero, it means the planning NPC health is zero, so whatever else might be happening in the plan state can be ignored as the planning NPC would be dead in this state. However, any function would be allowed for use as the top-level utility function so long as it takes drive satisfaction values and returns a number representing the utility of the plan state that produced those satisfaction values.

3.9 Worked Example

This section will give a simple example of two opposing NPCs, a wizard and a goblin, using UDGOAP to select a behaviour. The wizard can perform a **melee** attack with his staff and use mana to cast a **lightning bolt** spell to kill a distant enemy. The wizard has the drive to kill all enemies and

already has a goal to kill this particular goblin. The wizard also has a drive to maximize his own health points and another drive to maximize his own mana points. The wizard has many mana points but does not have many health points.

The goblin can perform a **slash** attack and **charge** attack with his sword, where a **charge** attack reduces target health points more than the **slash** attack does. The wizard could survive a goblin **slash** attack but could not survive a goblin **charge** attack. The goblin does not have many health points and would die if the wizard successfully landed a **melee** attack or a **lightning bolt**. The goblin has the drive to kill all enemies and already has a goal to kill the wizard. Both NPCs can perform the **goto** action to move to the target of their attack.

The goblin starts making his plan. The goblin first considers the state that would exist if he performed the **slash** attack. In this state, the wizard is low on health points but is still alive. The goblin then considers the state that would exist if he performed the **charge** attack. In this state, the wizard dies as a result of the goblin attack, which would completely satisfy the goal of killing the wizard. The goblin selects the **charge** attack for execution as it results in the highest utility state.

Meanwhile, the wizard has been making a plan of his own. The wizard considers the state that would exist if he performed the **melee** attack. In this state, the goblin would die as a result of the attack. The wizard then considers that would exist if he performed the **lightning bolt** attack. In this state, the goblin would die as a result of the attack but the attack itself

would consume mana. Both **melee** and **lightning bolt** attacks result in the death of the goblin but because casting the **lightning bolt** uses mana, this lowers the state utility for the drive of maximizing mana. The result is that the state generated from the **melee** action has a higher utility and so the plan that generated that state is selected for further development. The next planning iteration requires that the wizard satisfies the **melee** attack precondition of being at the goblin that the attack is to be used on. The **goto** action is selected because that action has an effect that satisfies the precondition of being near the goblin. However, the **goto** action also has a world effect. This world effect counts the number of enemies that would be within range of performing their own melee attack on the wizard and subtracts an amount of health equal to the damage that would be caused by such attacks. The world effect function finds that the goblin would be within range of attacking the wizard once the wizard moved to the goblin and subtracts wizard health accordingly in this plan state. In the state after the **goto** and **melee** actions are performed, the wizard would be dead as a result of being attacked by the goblin. This state is given a very low utility rating. The wizard instead chooses the plan that led to the highest utility state, that is, performing the **lightning bolt** attack.

Both the wizard and goblin have finished planning. The goblin immediately charges at the wizard. The wizards begins casting his lightning bolt spell. The goblin is just about to land his charge attack when the wizard finishes casting the spell and zaps the goblin dead with a lightning bolt.

A similar but more detailed worked example is provided in Appendix A.

The next three chapters describe experiments concerning utility-based behaviour selection systems. Chapter 4 describes an experiment with two purposes. The first purpose is to test if GOAP, which requires more resources than rule-based systems, is feasible for use in a real-time environment with many NPCs and also compares GOAP to a simple utility-based system. The second purpose of the experiment is to test if a simple utility-based system is feasible in the same environment. Chapter 5 describes an experiment that compares GOAP to UDGOAP by measuring how long it takes an NPC to go from a bad state to a near optimal state. This experiment is designed to test how each planner handles the pursuit of multiple goals. The experiment takes place in a virtual household like one found in The Sims[1]. This household contains just one NPC and is used to test how UDGOAP works in a discrete, static, deterministic, single-NPC environment. Chapter 6 describes an experiment that compares the final version of UDGOAP (the version described in this chapter) to GOAP and a finite state machine by measuring how long an NPC controlled by each system can survive in an arena. The experiment takes place in a fast-paced arena to test how UDGOAP compares to the other behaviour selection systems in a continuous, dynamic, non-deterministic, multi-NPC environment. Chapter 7 describes an extension to GOAP that uses **smart ambiance** to select more contextually appropriate actions for an NPC.

[1]The Sims - Maxis - http://thesims.com/en_US/home

Chapter 4

Utility and Action Planning Viability in Computer Games

A computer game is composed of many systems that must work together simultaneously. Physics, graphics, sound and other systems all require computational and memory resources to run. If the designer wants more out of a system, more sophisticated AI from the AI system for example, more resources must be allocated to that system. It is important that each system stays within the resource limits specified for the system by the designer. If an insufficient amount of resources are allocated to any system, the amount of time required to perform processing will take so long that the frame rate of the game can drop, which can cause noticeable lag and negatively impact on the game playing experience.

The percentage of CPU allocated for AI in commercial games is reported

to vary considerably over time and genre. In 1998, according to Woodcock (1998), AI in real-time games was allocated 10 percent of the CPU processing time. In 2000, due to increased offloading of computation to the GPU, real-time games were allocated approximately 30 percent of CPU for AI with a trend of AI being given an increasingly larger slice of CPU time (Woodcock, 2000).

This chapter details an experiment that was performed with two goals. The first goal was to test if GOAP was feasible for use in a real time environment with dozens of NPCs. Feasibility is judged on the amount of computational and memory resources required to run the system. Other games that have used GOAP, such as F.E.A.R.[1], had only a small number of NPCs active at any time to limit resource consumption. The experiment described in this chapter has dozens of active GOAP NPCs. The second goal is to see if it is feasible to use a simple utility-based behaviour selection system[2], introduced in Sloan *et al.* (2011a) and described in the next section, in the same environment with dozens of NPCs. The overall purpose of this experiment was to give an idea of the feasibility of using a utility-based action planner in a real-time environment.

4.1 Method

A simple utility-based behaviour selection system was compared to two other behaviour selection systems: a finite state machine and a GOAP system.

[1]F.E.A.R. - Monolith Productions - http://www.fear3.co.uk/the-game.html

[2]This utility-based system is an early version of the system described in the previous chapter.

All of the systems were compared in terms of the processor and memory resources required to run each system. A finite state machine was chosen for comparison because it has a reputation for being a resource-light behaviour selection system, and has been used in many games. GOAP was selected because it has a reputation for being a resource-heavy behaviour selection system and has also been used in many games. These two systems can be used to form a range to better gauge the resource requirements of the utility-based system.

Figure 4.1: A screenshot of the hospital simulation.

A simulated environment was developed as part of the experiment. The simulation takes place within a virtual hospital where game NPCs play the roles of nurses, doctors, patients, and visitors. A screenshot of this hospital environment is shown in Figure 4.1. Nurses perform administrative duties, socialize with other nurses and doctors, greet visitors, take breaks, and care for patients by giving them food and medicine when necessary. Each nurse has a level of fatigue and an inventory tracking if the nurse is currently carrying any food or medication. Doctors occasionally check on patients by looking at their charts. Patients ask for company when levels of loneliness

rise above a predefined threshold. Visitors chat with patients when called and rest when they are finished chatting. The actions executed by a game NPC are chosen by the behaviour selection system controlling it.

Separate finite state machines were developed for patients, visitors, doctors and nurses. These state machines are shown in Figure 4.2. A game NPC controlled by a finite state machine starts off in a state that has been selected as the starting state. In each state, the behaviour selection system checks if certain conditions have been met that would trigger the NPC to move into another state. For example, a nurse performing administrative duties would switch into a state of giving medication if the medication level of any patient fell below a certain threshold.

The particular implementation of GOAP is the same as the version described in Section 2.5.2.2. This version of GOAP has no performance enhancements e.g. no caching. Abstaining from performance enhancements was done to better gauge the maximum amount of resources a GOAP system might require. The planning NPC controlled by GOAP has a set of fixed priority goals, where goals and goal priority were authored at design-time. A new plan is made if the planning NPC has no plan, has discovered that its current plan has failed, or if a higher priority goal arises. The planner will create a plan for the highest priority goal found. The list of actions a game NPC can execute are specified at design-time and are used to form plans at run-time. The actions available to a nurse are shown in Figure 4.3, where rectangles represent actions, the arrow coming out of the top of a rectangle represent the effects of the action, the inverted arrow heads coming from the

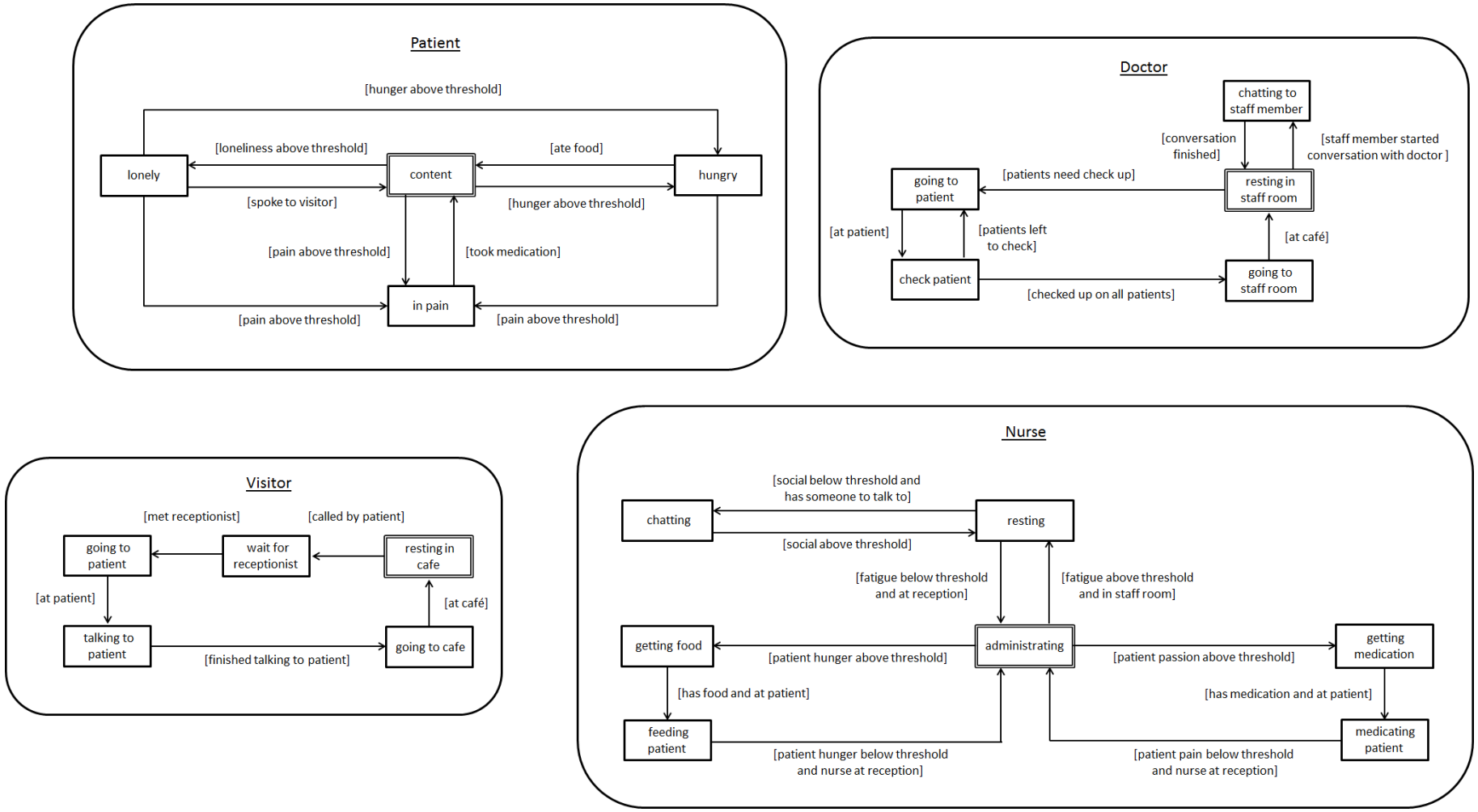


Figure 4.2: The finite state machines used during the hospital simulation. Rectangles represent states. Rectangles with double borders represent start states. Directed lines represent the transition of one state to another. The words enclosed in square brackets along a line are the conditions that must be true to transition between states. Rounded rectangles represent classes.

bottom of a rectangle represent preconditions, and the number next the the action represents the cost associated with the action.

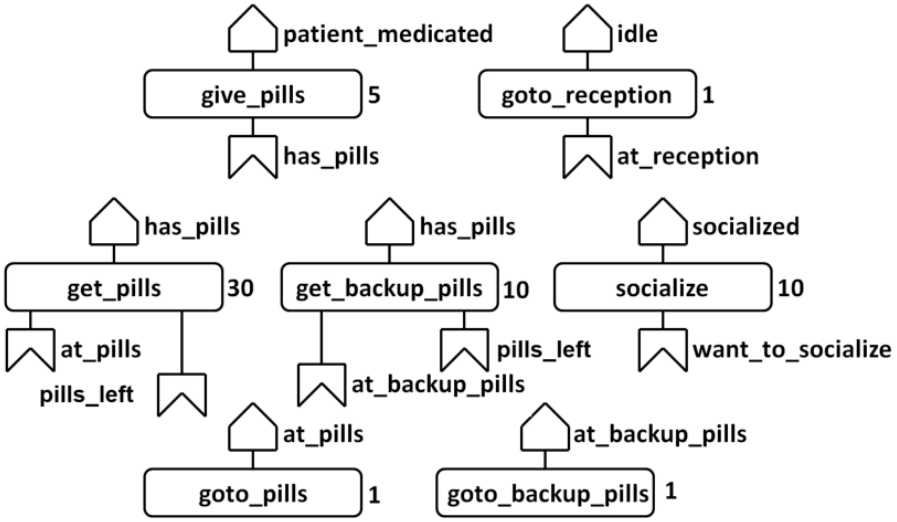


Figure 4.3: A representation of some of the actions available to a nurse in the simulation.

The utility-based system that was evaluated as part of this experiment was the earliest form of what would later become UDGOAP. The system had a set of drives, a set of predefined plans, and a utility function. Drives were associated with a fixed weight and one fixed goal. The set of drives an NPC has are defined at design-time and do not change at run-time. Like GOAP, each goal was associated with one object that was selected as the best object for satisfying the goal. Unlike GOAP, rather than having a set of actions, the utility-based system had a set of predefined plans. For example, the keep_patients_medicated goal has a precondition that could be satisfied with some effect in a predefined plan, such as {goto_pills, get_pills, goto_patient, medicate_patient}. The utility-based system rates the state that would exist after executing the best plan for each goal. For example, it rates the state

generated after performing the best plan to keep patients medicated, the state after performing the best plan to feed a patient and so on. The plan that results in the highest utility state is selected for execution.[1]

We believe it is acceptable to compare GOAP, a system that dynamically generated plans, with this utility-based system, which uses predefined plans, because the predefined plans are a subset of the plans that GOAP can dynamically generate. However, we are aware that the search space for the utility-based system is smaller than that of the GOAP system. This problem is addressed in the later version of this utility-based system that we described in Chapter 3 where the planner dynamically generates plan.

The environment was created as a modification of the game Half-Life 2[1] which use the Source Engine. This was selected because it was necessary to run the simulation in a game engine to ensure that all behaviour selection systems tested work within the computational and memory restraints placed on a commercial game. All behaviour selection systems were run through a Lua interface[2].

First-person games often try to strictly limit the number of NPCs active at any time. This is because each NPC can require a large amount of resources to perform at an acceptable standard. There were 32 NPCs in the hospital simulation because this is the default maximum (it can be increased with

[1] Although a utility-based behaviour selection system does not require predefined plans, they were used in this implementation because this experiment was conducted early in the research process and a method for dynamically chaining together actions based on state utility had not yet been developed.

[1] Half-Life 2 - Valve Software - http://orange.half-life2.com/

[2] Lua is a light-weight scripting language designed for use in computer games and has been used in a number of Source Engine based games.

mods) for the Source Engine (Valve, 2014) as it is generally considered the largest number that will not overburden the CPU of current generation mid-end devices. The human player could not influence the simulation.

4.2 Evaluation

An NPC does not have to query its behaviour selection system during each game frame because the difference in state from one frame to the next is often so small that no new behaviour will be selected. As a result, each NPC queries their behaviour selection system every 200 milliseconds. This frequency was selected for two reasons. The first reason is because 200 milliseconds is a long enough time that the game state for this particular simulation might have changed enough to warrant the selection of a new behaviour. The second reason is that is not so infrequent that an NPC will be left with an invalid behaviour for very long.

The amount of time taken to perform behavioural selection for each NPC is recorded every 200 milliseconds. The sum of the behaviour selection times at these 200 millisecond snapshots is said to be the amount of time required by that behaviour selection system. Only three runs were used for each behaviour selection system because the simulation is deterministic, causing the behaviour selection to be the same each time. However, a simulation was run three times for each system because although behaviour was the same, resource usage varied slightly between runs. The duration of a simulation was approximately four minutes because it took that long for all NPCs to

perform every behaviour available to them at least once. It was also the amount of time it took to reach a state very similar to the initial state of the simulation, where all patients are not in need of medication, food, check-ups, or company. Snapshots were taken every 200 milliseconds of the amount of memory used.

Processor and memory usage are calculated using the same approach as Khoo & Zubek (2002). For each behaviour selection system, the amount of time taken to select behaviours for all NPCs in the simulation is recorded in milliseconds. The average of the recorded times for each system is then divided by the number of NPCs in the simulation, giving the average number of milliseconds taken to select behaviour for an NPC using a particular behaviour selection system. This average per NPC is then divided by the frequency at which the behaviour selection system is queried, which is 200 milliseconds for this hospital simulation, giving the average CPU usage per NPC for a particular system. This CPU usage per NPC is used to approximate how many NPCs could be supported by a portion of the CPU allocated for behaviour selection.

Khoo & Zubek (2002) calculate system memory usage based on the number of static bytes required by the system. This is calculated by analysing the code for a system and counting the number of bytes required for the data members used by the system. Counting only the number of static bytes used and ignoring dynamic allocation makes sense because commercial computer games are designed to minimize dynamic memory allocation for increased efficiency. We use the same approach for calculating resource use as Khoo &

Zubek (2002).

The computer used for all of the tests described had an Intel Core 2 CPU 6600 at 2.4Ghz, 4096MB of RAM and ran Windows 7 Ultimate 64 bit.

4.3 Results

system	**time taken (ms)**	**% of CPU**	**max no. of NPCs**
FSM	5	0.025	400
GOAP	14	0.07	142
Utility-based	10	0.05	200

Table 4.1: Processor usage for behaviour selection systems in simulation

Table 4.1 shows the processor usage for each behaviour selection system used during the simulation. The **time taken** column refers to the average number of milliseconds it took the system to select behaviour for one of the 32 NPCs. The **% of CPU** columns refers to what percentage of the CPU was required to select behaviour for an NPC. This **% of CPU** was calculated by dividing the corresponding value from the **time taken** column by 200, as behaviour is selected every 200 milliseconds. The **max no. of NPCs** column shows the approximate number of NPCs that could simultaneously exist in the simulation using the corresponding control system given only 10 percent of the CPU. If 30 percent of CPU is allocated for AI, as specified in Woodcock (2000), we estimate that a third of that is for behaviour selection while the remainder could be for pathfinding and other AI.

GOAP required the most memory but this was a trivially small 13 kilobytes. The FSM used about 12 times less memory than GOAP and the

utility-based system used slightly less memory than GOAP.

4.4 Discussion

The purpose of the experiment was to understand if GOAP and a utility-based behaviour selection system are viable for use in environments with dozens of real-time NPCs, where viability is judged on the amount of computational and memory resources required to run the system. The amount of resources used by GOAP in the hospital simulation environment containing dozens real-time NPCs was the greatest of the three systems but far below the amount typically made available to AI systems in modern commercial games. The amount of resources required by the utility-based system was small enough to easily fit within the 10% of the CPU we believe would be allocated to behaviour selection in a commercial game. Furthermore, if GOAP is used in commercial games and the utility-based system using predefined plans requires fewer resources than GOAP, the utility-based system should be viable for commercial games, from a resource perspective.

An interesting observation was made while watching the nurses under the control of the utility-based system. Nurses under GOAP control always preferred to medicate patients as soon as their medication level fell below a certain threshold even in the presence of an extremely hungry patient because the goal of medicating patients had a higher priority. With the utility-based system, nurses would prefer to feed the very hungry more than medicating those only slightly in need. This happened because the utility-based system

associated goals with weights and calculated that although hunger is less important than medication level, a very hungry patient is more important than a patient only slightly in need of medication. The utility function acted as a kind of dynamic goal prioritizer.

The utility-based system had the ability to consider how a plan might affect more than one goal but the design of the environment and plans (unintentionally) never gave an opportunity for the system to make such considerations. The experiment described in the next chapter was designed to test if a utility-based system could generate plans to achieve a set of goals faster than GOAP can achieve a set of goals. The experimental environment was designed to give ample opportunities for the selection of plans that can affect the completeness of multiple goals.

Chapter 5

Multiple Goals and Complex Environments

The experiment in Chapter 4 showed that both GOAP and a utility-based behaviour selection system were capable of controlling dozens real-time NPCs using an amount of resources that could make the system feasible for use in commercial games.

The experiment (introduced in Sloan *et al.* (2011c)) in this chapter, we compare the ability of NPCs to satisfy a number of drives within an environment, where the behaviour of the NPCs is controlled by different systems. The systems compared are the GOAP system and UDGOAP. The main metric used to compare the performance of NPCs using these systems is how quickly the NPCs can nearly completely satisfy all of their drives. The hypothesis is that the NPC whose behaviour is controlled by the utility-based

planning system will satisfy its drives more quickly because it is able to consider how its selected plans affect multiple goals, unlike GOAP which only considers a single goal.

5.1 Method

The experiment takes place in a virtual household like those in the game *The Sims*[1]. A screenshot of the household is shown in Figure 5.1. The house is filled with objects with which the NPC can interact.

The NPC has a set of drives just like those in the Sims. These drives are: entertainment, energy, hunger, comfort, hygiene, bladder, and social. Each drive has a value ranging from zero to 100. In the initial state of the simulation, all drives are in their least preferred state, i.e. they start with a value of zero. The termination criteria of the simulation is when the value of all drives are in a near-optimal state. A near-optimal state is one where all drives simultaneously have a value above 90. The NPC uses the actions available to increase the value of its drives and reach a near optimal state.

Figure 5.1: A screenshot of the Sims-like household environment.

[1]The Sims - http://thesims.ea.com/

action	cost	attribute affected
wash_in_bath	10	hygiene, comfort
sleep_in_bed	10	energy, comfort
sit_on_chair	2	comfort, energy
drink_cappuccino	2	bladder, energy
eat_meal	2	hunger
cook_meal	2	
get_meal	2	
play_computer_game	10	entertainment
browse_social_network	10	social
chat_on_phone	10	social, entertainment
wash_in_sink	2	hygiene
eat_snack	1	hunger
get_snack	2	
watch_tv	2	entertainment
pee_in_toilet	2	bladder
poop_in_toilet	10	comfort, bladder
goto_bed	1	
goto_fridge	1	
goto_coffee_machine	1	
goto_chair	1	
goto_pc	1	
goto_food	1	
goto_phone	1	
goto_toilet	1	

Table 5.1: Actions available to the NPC during the simulation.

There are 25 actions available to the game NPC, shown in Table 5.1. For each drive, there are at least two actions available to the NPC that affect the value of the NPC drive. For example, the **watch_tv** and **play_computer_game** actions both have some effect on the **entertainment** drive. Many actions affects more than one drive. For example, the wash_in_bath action increases both the hygiene and comfort drive. The reason for having multiple actions that affect each drive was to make it so there were many ways to reach one of the goal states and to encourage more complex plans.

Actions in this simulation generally follow the rule that an action with a longer duration results in a larger increase to drives. For example, the wash_in_bath action takes longer than the wash_in_sink action but has a greater positive effect on **hygiene**. Some actions, such as wash_in_bath always take the same length of time to complete, while others, such as sleep_in_bed, only lasted as long as needed to maximize some drive. In many cases, it takes longer to increase a drive by performing multiple short duration actions than just one long duration action.

Furthermore, all actions that do not increase the value of the energy drive, decrease its value. For example, the drink_cappuccino action increases the value of the energy drive, but the make_cappuccino action decreases the value of energy.

All drives decrease over time. As a result, even if the NPC used the phone until it maximized the value associated with his social drive his social level might need topping up again after performing other actions.

The evaluation metric is the time it takes an NPC to reach a near-optimal state while under the control of a particular behaviour selection system. As a secondary focus, the complexity and diversity of behaviours performed by the NPCs under each of the control systems was observed. The near-optimal state required for the termination of the simulation was that each NPC drive, comfort, energy etc., were above 90% of their maximum level.

There is one NPC in the Sims household. The NPC begins with all NPC drives set to zero. The NPC performed under the control of GOAP and UDGOAP. The experiment takes place in a single-NPC, deterministic

environment so only one simulation run with each behaviour selection system was necessary to determine the time required by each behaviour selection system to reach the termination condition. As in the hospital experiment, the behaviour selection system was queried every 200 milliseconds.

The simulation took place using a modified version of the *Half-Life 2*[1] Source engine. The computer used to run the simulation had an Intel Core 2 CPU 6600 running at 2.4Ghz, 4096MB of RAM and ran Windows 7 Ultimate 64 bit.

5.1.1 Behaviour Selection System Design

This section describes the design decisions and the justification for these design decisions regarding the two behaviour selection systems used as part of this experiment.

5.1.1.1 GOAP

This experiment used the same version of GOAP described in Section 2.5.2.2, which is the same used in Chapter 4. The only difference is the new set of actions specific to the Sims environment. These actions include wash_in_bath, sleep_in_bed, and drink_cappuccino. The costs associated with these actions are important because it is the cost of an action that decides if it will be selected during plan formulation.

The Sims environment presents an obstacle to GOAP regarding action costs. There are only simple preconditions for the actions available. For

[1]Half-Life 2 - Valve Software - http://orange.half-life2.com/

example, the wash_in_sink and wash_in_bath actions only have the precondition of being at the sink and bath, respectively. As the goto action costs the same amount for satisfying the first precondition of each action, the lowest cost operator of either wash_in_sink and wash_in_bath will always be selected to achieve the goal of increasing the value of the hygiene drive. If the designer gives a lower cost to wash_in_sink than wash_in_bath, the NPC will never perform wash_in_bath. This is a limitation of static action costs in GOAP. The use of dynamic action costs to create more context sensitive behaviour is described in Chapter 7.

This action cost problem limits GOAP to only ever being able to apply less than half of the actions available to the NPC in the Sims environment. A way around this may be to add more preconditions, such as "only use the bath when very dirty", but this is tailoring behaviour in a manner used by systems like FSMs. This is not how GOAP is meant to work. A GOAP action should specify only when it *can* be applied, not when it *should* be applied. Determining when an action should be applied must be done based on action cost.

After testing, it was decided that action cost would be based on how long it would take the corresponding action to complete, such that shorter action execution time meant lower action cost. This decision was made based on the fact that it was impossible for a GOAP NPC to satisfy the simulation termination condition if long execution time actions were associated with low costs. The simulation termination condition was never satisfied because by the time a long execution time action is executed to improve one drive, other

system	duration (seconds)
GOAP	146
UDGOAP	118

Table 5.2: Time taken in seconds for an NPC using a given behaviour system to reach the experiment termination condition of a near-optimal state for all NPC drives having started in a state where all drive began with a value of zero.

drives fall below the termination condition threshold. This meant that for consistency and ability to achieve the simulation termination condition, the actions associated with short execution durations had the lowest cost.

The goal selected to be achieved by GOAP is the goal associated with the lowest drive value at the time of goal selection. For example, if hunger is associated with a value of 30, and this value is lower than the values associated with entertainment, energy, comfort, hygiene, bladder, and social, then the goal of improving hunger will be selected.

5.2 Results

Table 5.2 shows how long it took for an NPC under the control of each selection system to change from the worst state for its drives to a near-optimal state. The table shows that the UDGOAP NPC finished approximately 19% faster than the GOAP NPC.

As we have already noted, the version of GOAP used in this experiment is the same as the one used in the hospital experiment described in the previous chapter but the utility behaviour selection system is quite different. The differences in the utility behaviour selection system did have an impact on

the resource requirement of the system: UDGOAP required approximately 23% more CPU time than GOAP. UDGOAP also required approximately 37% more memory than GOAP.

5.3 Discussion

The GOAP NPC took longer to finish than the UDGOAP NPC. There appeared to be two main reasons behind this. The first is that the UDGOAP NPC used its ability to consider multiple goals to find the best overall action. The second is that GOAP only performed short duration actions and missed out on potentially more beneficial long actions as a result.

UDGOAP used its ability to consider multiple goals to find better plans. Observation of the UDGOAP NPC showed that after it selected a hard goal, it always preferred actions that improved not only the drive associated with that goal, but also associated with other goals. For example, although the goal might be to improve NPC hygiene, rather than executing the wash_in_sink action, improving only NPC hygiene, it instead preferred the wash_in_bath action, which improved both NPC hygiene and comfort.

The GOAP NPC performed many short duration actions that had a small impact on the drives, whereas the UDGOAP NPC performed some long duration actions that had a large impact on the drives, followed by short duration actions. For example, eat_snack and eat_meal both affect just the same drives, hunger and energy, but eat_meal takes longer and has a greater effect on hunger. Using long, high impact actions first, UDGOAP could make

a large improvement to a very low value drive. Following up with short, low impact actions, such as the eat_snack action, allowed the UDGOAP NPC to then cause small increases to drives without taking too much time.

Another observation in the behaviour of the GOAP NPC was how the inability of GOAP to perform long actions caused it to move around the apartment more. This happened because the drive with the lowest value changed more often those of the UDGOAP NPC drives. For example, the NPC might wash itself a little, increasing hygiene and causing entertainment to become the new lowest value drive, then the NPC would watch a little television for entertainment, causing hygiene to have the new lowest drive value, then the NPC would wash a little again to improve hygiene. On the other hand, the UDGOAP NPC would have a bath, hugely improving hygiene so that it was far from the lowest drive. This behaviour the GOAP NPC displayed is not to be confused with dithering as the NPC always completed the actions after one was selected.

UDGOAP may have outperformed GOAP in the Sims environment but this was a static, single-NPC, deterministic environment where all goals were known at design-time. The next iteration of UDGOAP built upon UDGOAP to work in a dynamic, multi-NPC, non-deterministic environment where goals are not known until run-time.

Chapter 6

Dynamic Goals and Environments

The experiment in the previous chapter tested if the ability of UDGOAP to consider multiple goals when planning could produce superior plans to the GOAP system that only considers a single goal. However, the experiment took place in a static, single-NPC, deterministic environment and not the dynamic, multi-NPC, non-deterministic environments for which GOAP was designed. This chapter describes an experiment that takes place in a complex, continuous, highly dynamic, multi-NPC, non-deterministic environment similar to those found in modern action games that GOAP and finite state machines have been designed for. The purpose of the experiment is to compare the performance of UDGOAP to a finite state machine and GOAP in a fast-paced environment. The performance of the UDGOAP

NPC is compared to that of a GOAP NPC and a finite state machine NPC designed for Half-Life 2, a fast-paced action game.

6.1 Method

The simulation takes place in a large arena containing a magical alien wizard and an endless supply of enemies, giant antlions that try to kill the wizard. The wizard is placed in the centre of the arena at the beginning of the simulation. The objective of the wizard is to live as long as possible. A screenshot of the arena is shown in Figure 6.1.

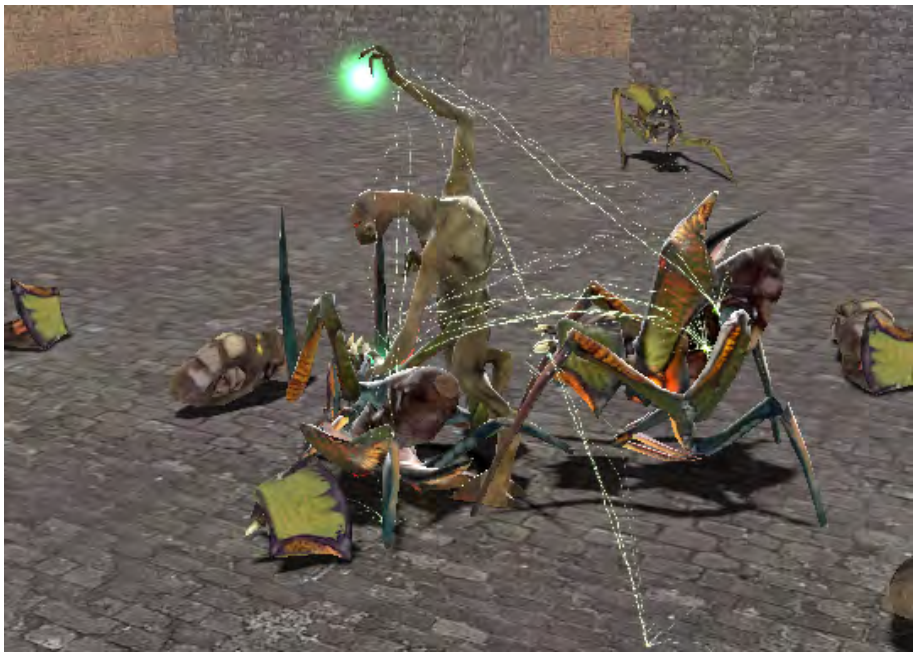


Figure 6.1: A screenshot of the alien wizard fighting enemy antlions in the arena.

Four spawn points create enemy antlions with a randomly selected frequency from 2.5 to 5 seconds. Antlions attempt to kill the wizard by performing close-combat melee attacks that reduce the health of the wizard. Antlion behaviours are those that are already built into the Half-Life game.

The behaviour selection system built into the Half-Life game is described in Section 6.1.1. Only the wizard used different behaviour selection systems.

The wizard can attack antlions either by using a melee attack or one of two spells: a long-range lightning attack that kills exactly one antlion, or a blast attack that kills all antlions within a short range of the wizard. The wizard also has a heal spell that increases his health. All spells expend a fixed amount of a resource called mana which is subtracted upon beginning to cast the spell. A spell can fail if it is interrupted by an attacking enemy antlion or if the wizard moves.

The wizard starts in the arena with his maximum 100 health and maximum 100 mana. Items that increase health or mana by a random amount from 1 to 100 spawn in random locations within the arena. These items spawn with a random frequency from 10 to 20 seconds. Only the wizard can use these items. Items are immediately used by the wizard upon coming within a small distance of the item.

The arena is completely reset upon the death of the wizard. The reset causes the removal of all existing antlions and items. The time in between which the wizard spawned and died is referred to as an **episode**. The wizard can perform the following actions.

- **goto** — Moves the wizard from one position to another. This action can only be performed on object positions rather than arbitrary (X,Y,Z) coordinates. This is a limitation of GOAP and UDGOAP that both require an object for an action to be executed upon, be they potions

or way point nodes.

- **melee** — A close-combat attack. The wizard will usually take damage while performing this attack. This attack can sometimes miss the target.

- **lightning** — A spell that kills one visible target within a long range at the cost of 20 mana. This spell can sometimes miss the target if the target moves quickly while the wizard is casting the spell. This missed attempt still spends the wizard's mana. This spell can be interrupted if the wizard receives heavy damage during the casting of the spell.

- **blast** — A spell that kills all enemies within a short distance of the wizard at a cost of 60 mana. This spell cannot miss any targets inside the blast radius.

- **heal** — A spell that heals the wizard 25 health at the cost of 15 mana. This spell cannot miss the target. This spell can be interrupted if the wizard receives heavy damage during the casting of the spell.

The goal of the experiment is to test if UDGOAP can use its ability to consider both multiple goals and closeness to the completion of a goal to have the controlled NPC live longer than an NPC controlled by industry tested systems in the kind of environment for which those systems were designed. These industry tested systems are GOAP and the finite state machine used by the Half-Life 2 engine, both of which were designed for fast-paced, complex, many-NPC environments. The arena in which the wizard fights is such an

environment. The arena is typical of arenas in other games where there is an endless supply of enemies but health and other items are made available periodically. Those typical arena designs were imitated during the design of this wizard arena so as to create an environment not tailored for the benefit of the UDGOAP system.

The wizard performed 300 episodes in the arena under the control of each behaviour selection system. UDGOAP is compared to GOAP and the Half-Life 2 finite state machine by measuring the average lifespan of the wizard under the control of those systems.

6.1.1 Behaviour Selection System Configuration

This section describes the behaviour selection systems used in this experiment and how these systems were configured in an attempt to produce behaviours optimized for the experiment environment. The three behaviour selection systems that were used in the experiment are the finite state machines used in the Half-Life 2 game, GOAP, and UDGOAP.

Each behaviour selection system had values that had to be optimized for the arena environment. Finding the optimal configuration for the finite state machine involved searching for the health and mana thresholds at which the wizard would take actions to increase its health and mana levels. For GOAP, the search for the optimal configuration was a search for the set of action costs that maximize the lifespan of the wizard. For UDGOAP, the search for the optimal configuration was a search for the set of drive weights that maximize the lifespan of the wizard.

6.1.1.1 Half-Life 2 Finite State Machine

The behaviour selection system used in Half-Life 2 is a type of finite state machine. This behaviour selection system will be referred to as the Half-Life 2 finite state machine (HL2FSM). An overview of the system is shown in Figure 6.2.

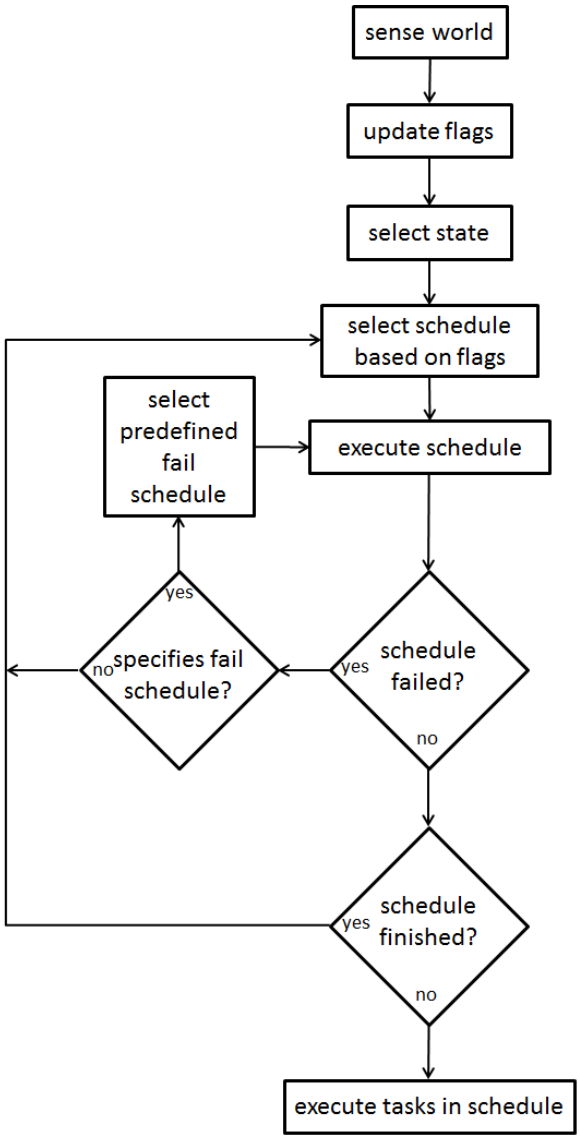


Figure 6.2: A flowchart showing the behaviour selection process used by the HL2FSM.

Each HL2FSM has a set of **flags**, states, and **schedules**. The pro-

cess of behaviour selection for the HL2FSM begins by the NPC sensing the world. The sensed data is used to set boolean flags[1]. For example, the cond_heavy_damage flag is set if the NPC to whom the HL2FSM flag belongs has taken a lot of damage since the last time a behaviour was selected. An NPC state is then selected based on the values of the flags. These states include the idle, combat, and dead states.

A schedule is a behaviour within a HL2FSM that takes the form of a predefined plan. Each schedule has a set of optional failure flags indicating under what conditions the schedule can fail, an optional predefined failure schedule to execute should the schedule fail, and a sequence of **tasks**. A task is similar to a low-level action. Each task can change some variable specific to the schedule to which it belongs, or the NPC performing the schedule. Every action the wizard could perform, described in the previous section, has an equivalent schedule. For example, the heal action has the equivalent schedule: heal = {(set_fail_schedule, alert), (get_heal_target, null), (stop_moving, null), (face_target, null), (heal_target, null), failure: cond_heavy_damage}, which sets a schedule to switch to should this schedule fail, selects a target to heal, stops the NPC from moving, faces the selected feeling target, and heals the target. The schedule only fails of the NPC takes heavy damage.

The failure flags are a subset of the recently updated flags. If any flag is set to true, the schedule fails. Upon failure, a new schedule will be selected. If the schedule specifies a failure schedule, the failure schedule will be selected for execution. When in a particular state, an NPC can only select a

[1]These flags are referred to as conditions within the engine but we avoid that term because the term condition is already used to refer to something else in this thesis.

schedule relevant to that state. However, any specified failure schedule can be transitioned to, regardless of if the other schedule could only have been selected while the NPC was in a different state.

One of the goals of this experiment was to implement behaviour for the wizard using the HL2FSM while preserving as much of the existing behaviour as possible so the other systems could be compared to a system used in a successful action game. However, certain behaviours were specific to the arena simulation, such as the behaviours to pick up mana potions when the cond_critical_mana flag is set, and to pick up health potions or cast a healing spell when the cond_critical_health flag is set. These flags required thresholds to be set so as to determine when the flag would be set to true.

A brute-force approach was used to determine the optimal values for the thresholds at which the cond_critical_mana and cond_critical_health flags should be set. A pair of values was generated where one value was used as the health threshold and the other was used as the mana threshold. Each value in the pair was a multiple of five, had a minimum value of zero, and had a maximum value of 100. Each permutation of the pair was tested, e.g. (0, 0), (0, 5), (0, 10), ..., (100, 95), (100, 100). The wizard performed 40 episodes in the arena with each threshold pair permutation and the lifespan of the wizard was recorded for each episode.

Figure 6.3 shows a surface plotted using the mean lifespan of the wizard for each health-mana threshold pair. Each mean was based on 40 episodes of the wizard in the arena with a particular pair of values for the health and mana thresholds at which the wizard will select behaviour to increase his

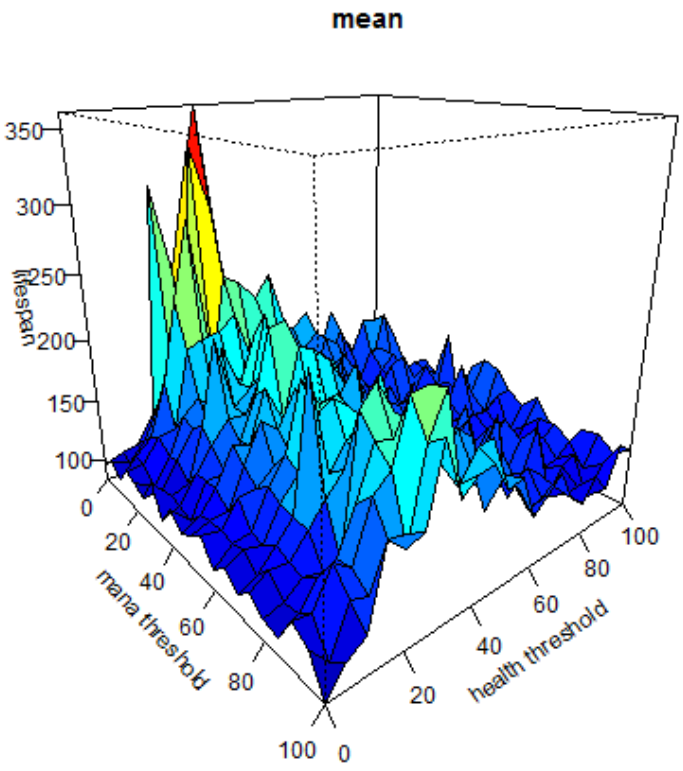


Figure 6.3: The surface produced by the mean 40 episodes for a pair of health and mana thresholds.

health or mana.

The (health: 35, mana: 0) pair had both the highest mean lifespan of all of the health-mana threshold pairs. This pair intuitively makes sense because the wizard should only pick up mana potions when his mana level is very low because spells do not cost much mana and mana is not crucial, but the wizard should take actions to increase his health before its too late, as the wizard often takes damage while running to a health potion. The HL2FSM for the wizard with this optimal health-mana threshold pair was selected as the final configuration for use during the experiment.

6.1.1.2 GOAP

The GOAP system used in the arena experiment was the same as that used in the hospital and Sims experiment. The GOAP wizard has all five of the actions described in Section 6.1. Ideally, the actions would be associated with their optimal costs, where the *optimal* action cost set maximizes the lifespan of the wizard using the actions associated with that cost set. A brute force approach was not used for finding this optimal GOAP action cost set because the brute force search on just two variables used in the HL2FSM system took approximately three weeks to complete, and scaling to the six values, one for each GOAP action, would be computationally infeasible.

In the search to find the optimal action cost set, random action cost sets were created consisting of five action costs[1]. A total of 270 random action cost sets were tested for 10 episodes each. The action cost set associated with the longest average lifespan from random sets was selected as the final set. The final set of action costs is shown in Table 6.1.

Table 6.1: The final set of GOAP action costs.

action	**cost**
goto	0.23
melee	0.38
lightning	0.71
blast	0.53
heal	0.3

[1]A hill climbing approach was originally taken but was found ineffective because of the geography of the action cost landscape. The random start approach finally used covered more ground and produced cost sets associated with longer lifespans.

6.1.1.3 UDGOAP

The UDGOAP system used in the experiment is the same as the system described in Chapter 3. Each drive in UDGOAP requires a weight that denotes its importance to the utility of the state to the NPC. The wizard had six drives: kill_enemies, maximize_health, maximize_mana, maximize_health_potions, maximize_mana_potions, and minimize_enemies_nearby.

In the search to find the optimal drive weight set, random drive weight sets were created consisting of six drive weights[1]. A total of 270 random drive weight sets were tested for 10 episodes each. The drive weight set associated with the longest average lifespan from random sets was selected as the final set. The final set of weights selected is shown in Table 6.2.

Table 6.2: The final set of UDGOAP weights.

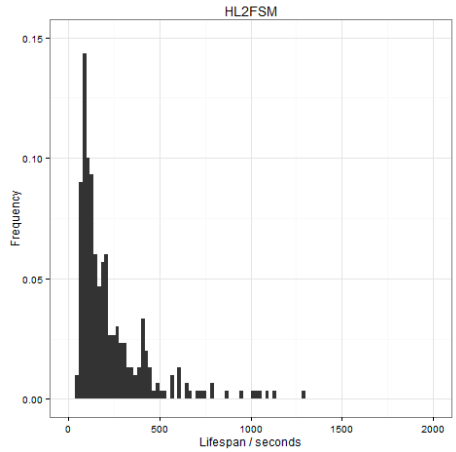
action	**cost**
kill_enemies	0.38
maximize_health	0.48
maximize_mana	0.34
maximize_health_potions	0.38
maximize_mana_potions	0.33
minimize_enemies_nearby	0.66

6.2 Results

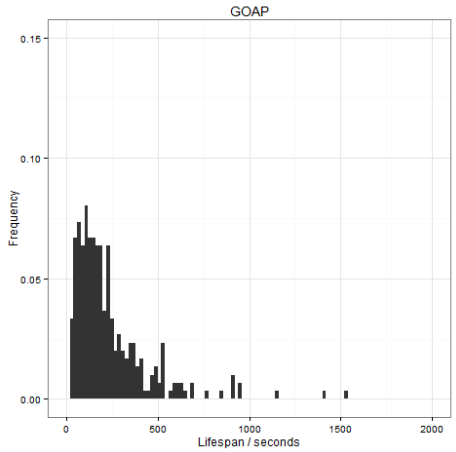
The histograms in Figure 6.4 are produced from the lifespans of the wizard in the arena. Each histogram was created from the wizard performing 300 episodes under the control of the behaviour selection system specified in

[1]As with GOAP, a hill climbing approach was originally taken but was found less ineffective than a random start approach.

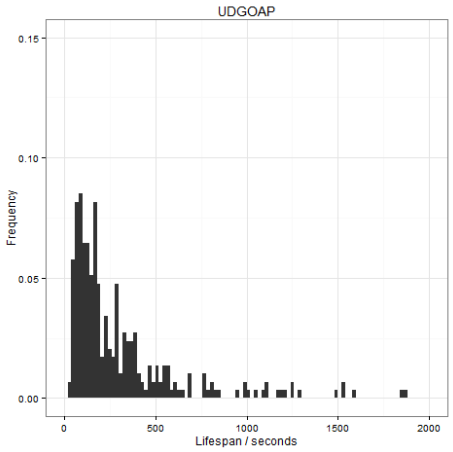
the histogram. The lifespans of the HL2FSM wizard are bundled together at values less than 500 seconds with a peak near the 125 seconds mark. Lifespans for the GOAP wizard have a much more even distribution. Lifespans for the UDGOAP wizard are much more scattered across the histograms.



(a) HL2FSM



(b) GOAP



(c) UDGOAP

Figure 6.4: Histograms showing the wizard lifespan for each of the three behaviour selection systems.

Table 6.3 shows the mean and standard deviation in lifespan of the wizard using each behaviour selection system for 300 episodes. The table shows that the wizard using UDGOAP lives substantially longer on average than the wizard using the other systems, but the standard deviation of the lifespans

of the UDGOAP wizard is far greater than those of the other two systems. This suggests that the UDGOAP wizard sometimes lived a very long time and sometimes lived for a very short time whereas the wizard using the other systems would have more predictable lifespans. This is reflected in the histograms in Figure 6.4. This large standard deviation is probably due to the UDGOAP wizard preferring melee attacks and using a blast attack when surrounded by antlions. At the beginning of each episode, the UDGOAP wizard quickly meleed with the antlions. Health potions spawn periodically but sometimes the health increase from the first couple of potions would be small and the UDGOAP wizard could not regain the health it had lost during the first several melees and die as a result. However, other times the first couple of health potions healed very well and by the time the wizard needed health again, several health potions would have spawned and but there would be very few antlions alive because of how the UDGOAP wizard would melee when there were few antlions and blast when there were many. The result was that once the UDGOAP wizard got past the initial phase where health potions were unavailable, the melee attacks when there were few antlions and blast attack when surrounded was a good strategy. The HL2FSM wizard took a more conservative approach of often shooting enemies from afar but this back fired as the HL2FSM wizard might later have no mana available for his blast attack when it became surrounded by antlions. The GOAP wizard frequently meleed like the UDGOAP wizard but did not use its blast attack when surrounded because the cost of that action was higher than the melee action.

Table 6.3: The mean lifespan and the standard deviation of the lifespan of the wizard over 300 episodes for each behaviour selection system with its configuration optimized for lifespan in the arena.

System	Mean	Std. Dev.
HL2FSM	233.62	203.52
GOAP	242.71	236.52
UDGOAP	381.98	639.41

We wanted to know if there was a significant difference between the lifespans of the wizards under the control of the behaviour selection systems. The experiments performed using the three behaviour control systems generated unpaired, non-normalized, non-parametric data, so a Kruskal-Wallis test (Kruskal & Wallis, 1952) was applied to the lifespan data generated from the 300 episodes run by each of the behaviour selection systems. This Kruskal-Wallis test returned a p-value of 0.038, suggesting that the at least one of the three wizard lifespan populations came from a different source. This rejection of the null hypothesis was followed up by a post hoc Dunn test (Dunn, 1961) where p-values were adjusted for multiple comparisons using the Hochberg procedure (Hochberg, 1988). Table 6.4 shows the results of this test. There is a significant difference in the lifespan of the wizard using the UDGOAP system when compared to the other two behaviour selection systems.

6.3 Conclusion

This chapter detailed an experiment to compare a UDGOAP NPC to HL2FSM and GOAP NPCs in the kind of highly dynamic, multi-NPC environments

Table 6.4: The p-values of a Dunn test comparing the lifespan of the wizard using each behaviour system for 300 episodes, checking if each system performed significantly better than the others, where the values are adjusted using the Hochberg procedure.

	HL2FSM	**GOAP**	**UDGOAP**
HL2FSM	—	0.439	0.0332
GOAP	0.439	—	0.0337
UDGOAP	0.0332	0.0337	—

in which the HL2FSM and GOAP systems have been used successfully in games. The purpose of the experiment was to determine if UDGOAP can use its ability to consider multiple goals to outperform the HL2FSM and GOAP in the kind of environment for which they were designed.

The performance of each system was based on the lifespan of the NPC controlled by that system in an arena. The results showed that the UDGOAP NPC had the longest lifespan and that the UDGOAP NPC lived significantly longer than the NPCs controlled by both HL2FSM and GOAP.

These results show that even in the kind of environment that HL2FSM and GOAP were designed for and even in an environment where there are few opportunities for selecting behaviours that affect multiple goals, UDGOAP still performed best.

Chapter 7

Smart Ambiance

The last three chapters described the evaluation of different versions of the UDGOAP system and how utility can benefit action planners games. This chapter focuses on a system called **smart ambiance** to select more contextually appropriate actions, demonstrated by examples of NPCs preferring quiet actions in a library, NPCs becoming nervous and running across a road after seeing other NPCs run, and a spy who walks when near cameras but runs when not near cameras. Smart ambiance uses the environment to select more contextually appropriate actions, unlike UDGOAP that used smart objects to enact plans in a more contextually appropriate manner. Smart ambiance alters action costs so that contextually appropriate actions are more likely to be selected. This chapter provides the architecture for smart ambiance and demonstrations using smart ambiance. All of the demonstrations in this chapter were performed by using smart ambiance with GOAP but smart

ambiance can be used with any numeric based planner, including UDGOAP.

GOAP is a powerful behaviour selection system. The costs associated with actions make it possible for a designer to specify which plan an NPC should prefer given multiple ways of achieving a particular goal. However, static action costs become a hindrance when there are either multiple actions have the same preconditions and a shared effect, or certain actions become more or less appropriate at certain times or places.

If an NPC has multiple actions with the same preconditions and a shared effect, the lowest cost action will always be selected when trying to achieve a precondition that requires that effect[1]. For example, an NPC might have the kick and punch actions, both of which have the precondition that the enemy they are performed on is within melee distance and both of which have the effect of killing the enemy. If the punch action has a lower cost than the kick action, the punch action will always be selected when attacking an enemy within melee distance. This can lead to repetitive behaviour but also means that the kick action is essentially unavailable to the NPC.

There may also be times and places where certain actions become more or less appropriate. For example, shouting at a football pitch is normal but shouting in an office is not. Adding a precondition so that the shout action cannot be performed when in an office is equivalent to a rule-based system where the designer is trying to anticipate every situation in which the NPC might find itself because not being in an office is not really a precondition of

[1]This assumes the plans needed to reach these actions are the same cost, but in our experience, most plans consist of a goto action followed by some other action, so this assumption has very often been true for us.

shouting. This additional precondition specifies when the action should be performed rather than how the action is performed, robbing GOAP of one of its best attributes.

Smart ambiance, introduced in Sloan *et al.* (2011b), uses the environment of an NPC to inform the behaviours of the NPC. This makes it possible to have multiple actions with the same preconditions and still have different actions selected according to different contexts. It also makes it possible to have the NPC prefer more contextually appropriate actions, for example, making it unlikely for the NPC to shout in an office.

As well as creating more contextually appropriate behaviours, smart ambiance can lead to the generation emergent behaviours. Emergent behaviours are described by Li *et al.* (2007) as behaviours that have not explicitly been designed for but instead emerge from a combination of rules for other behaviours. Emergent behaviours are desirable in some game environments because such behaviours have been shown to be interesting to players and add longevity to games (Bauckhage & Thurau, 2004).

Smart ambiance works as an extension of GOAP by associating actions with ambiance annotations that specify which actions are more appropriate given a particular ambiance. GOAP actions associated with appropriate ambiance annotations have their cost reduced.

We use three proof-of-concept demonstrations to show how different aspects of smart ambiance can be used to cause the selection of more contextually sensitive behaviour. These demonstrations take place in an environment

like those found in The Sims[1] games, where several ordinary human characters exist in a world not too unlike the real world but where interactions and emotions are exaggerated. The three aspects of smart ambiance that will be described in this chapter are location ambiance, event ambiance, and object ambiance. All demonstrations are implemented using Valve's Source Engine. The implementation of GOAP used with smart ambiance is the same as the implementation used in the hospital, Sims, and arena experiments and is described in Section 2.5.2.2.

This chapter has the following structure. An overview of smart ambiance is given in Section 7.1. Each of the three aspects of smart ambiance — location, event and object ambiance — are then described along with demonstrations of each in use in Section 7.2. The chapter ends with conclusions on smart ambiance in Section 7.3.

7.1 Smart Ambiance Overview

A **smart ambiance** is an area of space inside a virtual world where the area itself, the presence of objects in the area, and the occurrence of events in the area, all alter the cost of actions for NPCs inside the area so that more appropriate actions cost less and less appropriate actions cost more. Luck & Aylett (2000) describes **intelligent virtual environments** that can assist NPCs by providing the NPC with information about the environment. Smart ambiance is a type of intelligent virtual environment that uses information

[1]The Sims - Maxis - http://thesims.com/en_US/home

from the environment to help create a context for the environment.

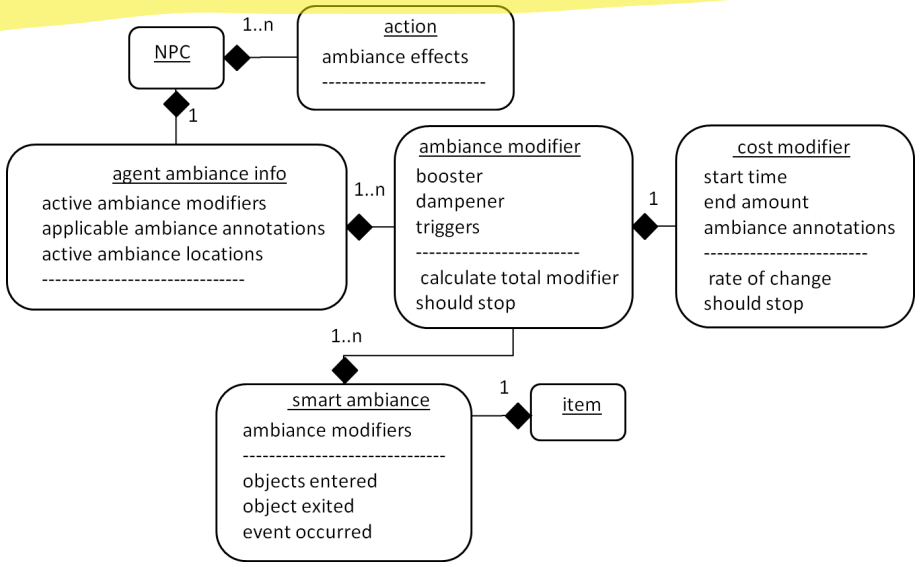


Figure 7.1: A class diagram of smart ambiance. Lines connected by filled diamonds denote a has-a relationship.

Figure 7.1 shows a class diagram of the components of smart ambiance that make these cost modifications possible. Each smart ambiance action is associated with a set of ambiance annotations, e.g. the work action might have the serious ambiance annotation and a set of ambiance effects that describe how an ambiance is altered by the performance of the action. Each NPC has smart ambiance information. This information describes which annotations are applicable to the NPC. For example, an NPC with a serious personality would have the serious ambiance annotation in its applicable ambiance annotation set, meaning that this NPC can have its serious action costs modified by the smart ambiance. The NPC ambiance information also contains a set of ambiance modifiers that are active and currently applied to an NPC, called the active ambiance modifier set.

An **ambiance modifier** adds a value to the predefined cost of an action,

either positive or negative, where the predefined cost is a static cost chosen by a designer. This cost modification occurs during plan formulation and applies only to an NPC with the ambiance modifier in its active ambiance modifier set. A ambiance modifier has the following components:

1. **cost booster** — A cost booster alters the cost of an action to be different from the predefined action cost.

2. **cost dampener** — A cost dampener has the opposite effect of the booster in that it brings the cost of an action back to its predefined value.

3. **triggers** — A set of rules that will generate and activate the ambiance modifier.

4. **calculate total modification** — A function that combines the booster and dampener cost modifications to get a final cost modification amount.

5. **should stop** — A function that checks if both the booster and dampener are finished applying their cost alteration effects. If they are finished, the ambiance modifier should be removed from the list of ambiance modifiers that are currently applied to the NPC.

The cost booster and cost dampener are both cost modifiers. A **cost modifier** alters a GOAP action cost. Each cost modifier has a start time at which it was created, a total amount denoting how much it should change action cost by, a set of ambiance annotations that the cost modifier applies

to, and a function that describes the rate at which the cost modification should occur.

For example, the **m** = (start time: 123.0, end amount: 10, ambiance annotations: serious, rate of change: fn_instantaneous) cost modifier instantaneously causes a change of 10 to the cost of all actions that have a serious ambiance annotation, assuming the ambiance modifier that contains this cost modifier is in the set of active ambiance modifiers for the NPC.

A smart ambiance is an area inside the world. A smart ambiance contains a list of all of the ambiance modifiers. These modifiers are applied to any NPC inside the area when that NPC is selecting an action. If an ambiance modifier is relevant to the NPC, the modifier alters the costs of the relevant actions. The NPC ambiance info determines which modifications are relevant to the NPC.

7.2 Aspects of Smart Ambiance

This section defines each of the three aspects of smart ambiance, describes their implementation, describes a demonstration that was created to showcase each aspect, and mentions previously researched techniques similar to these aspects.

7.2.1 Location Ambiance

Location ambiance is the inherent ambiance of an area. For example, a graveyard has an inherently serious ambiance, so actions associated with the

serious ambiance annotations should be more likely to occur.

Location ambiance works by associating ambiance modifiers with a smart ambiance and causes any NPC who enters the area of the ambiance to immediately have those associated ambiance modifiers added to their set of active ambiance modifiers of the NPC. These modifiers then alter the cost of certain actions. The location ambiance of an area is defined at design-time and remains unchanged at run-time.

A scenario was constructed to demonstrate location ambiance. In the demonstration, an outgoing character like those found in The Sims is in a library occupied by other characters and several computers, as shown in Figure 7.2. The outgoing character has the goal of socializing. There are two actions with effects that satisfy this goal: the chat action, which is associated with the loud ambiance annotation, and the check_social_network action, which is not associated with any ambiance annotations. This is an outgoing NPC so when the costs of these actions were being set, the designer assigned the chat action a lower cost than the check_social_network action to encourage the character to be chatty. However, in the context of a library, the chat action is less appropriate. In the library, there is a friend of the character that can be chatted with, and there is a computer that the character can use to check his social network. The scenario is played twice: once for the NPC with smart ambiance enabled and once without ambiance enabled.

When the NPC without smart ambiance enabled enters the library, no ambiance modifiers are applied to the NPC, which causes the chat action to remain the lowest cost action that achieves the goal of socializing. As

a result, the character walks up to his friend in the library and begins to chat loudly, which is something that normally does not happen in a library. However, when the scenario is played for the character affected by smart ambiance, the costs associated with all actions that have the loud ambiance annotation are increased. This increases the cost of the chat action, causing the check_social_network action to become the lowest cost action to achieve the goal of socializing. The character then performs the check_social_network action. The consideration of location ambiance allows the character to perform a more contextually appropriate action in a given location.

Figure 7.2: A screenshot taken in the virtual library.

There is some overlap with location ambiance and Paanakker (2008), where soldiers would stay away from parts of a map that had dead allies in them. The work most similar to location ambiance is Sung *et al.* (2004). Sung's approach uses **spatial situations**. The spatial situations use smart objects and predefined zones that give possible actions to NPCs when they come within a certain range of them. For example, a bus stop area would add the actions of getting on or alighting a bus. However, Paanakker's method only works for path finding and Sung's method enables certain actions in

certain areas. Location ambiance does not enable or disable actions in a particular area but instead makes certain actions more likely to occur or not occur through the alteration of action costs.

7.2.2 Event Ambiance

An **event ambiance** changes the smart ambiance in a location based on the occurrence of a particular event. Each action in the smart ambiance framework has the usual GOAP symbolic and context effects, but may also have ambiance effects. Each **ambiance effect** is associated with an ambiance modifier that is added to the smart ambiance in which the action was performed. The ambiance then queries each NPC inside the ambiance to see if the newly added ambiance modifier should be applied to the NPC. The ambiance modifier should be applied to the NPC if any of the ambiance annotations of the ambiance modifier is also present in the set of ambiance annotations applicable to the NPC. If such an ambiance annotation is present, the newly attached ambiance modifier is then added to the set of active ambiance modifiers of the NPC.

For example, consider a smart ambiance within which the give_out_to action occurred. This action has an ambiance effect that causes the generation of a cost modifier. This cost modifier has the serious annotation. The smart ambiance queries each NPC inside the smart ambiance and finds an NPC that has the serious annotation in its set of applicable ambiance annotations. The NPC has this serious ambiance annotation in its set of applicable ambiance annotations because the designer wanted this NPC to respond to events that

affect the serious actions of the NPC. All actions that are associated with the serious ambiance annotation then have their costs modified by the cost modifier associated with the ambiance effect of the give_out_to action.

Figure 7.3: A screenshot pedestrians crossing a road.

A scenario was constructed to demonstrate event ambiance. In the demonstration, there is a traffic crossing with a number of pedestrians who wish to cross, as shown in Figure 7.3. The crossing and all NPCs are contained within a smart ambiance. The pedestrians have the goal of crossing the road. There are two actions available to the pedestrians: the walk action, which is not associated with any ambiance annotations or ambiance modifiers, and the run action, which has the nervous ambiance annotation and has an event ambiance effect that is associated with an ambiance modifier that decreases the cost of all nervous actions. This means that every time a pedsetrian runs across the road, the ambiance effect of making nervous actions cost less is added to the ambiance modifiers of the smart ambiance, making nervous actions, such as the run action, more likely to occur.

There are two types of pedestrians: a normal type with a lower cost for

the walk action than the run action, and a nervous type with a lower cost for the run action than the walk action. When there are only normal pedestrians, all pedestrians walk across the road. When there are several nervous pedestrians, at first, only the nervous pedestrians run, but shortly after, even the normal pedestrians run. This happens because several ambiance modifiers are added from the ambiance effects of the run action, causing a threshold to be crossed where the cost of the run action becomes lower than the cost of the walk action for normal pedestrians. This makes it so ordinarily, a normal character will perform regular actions, but the presence of a nervous person will have an infectious nervous effect on the normal character, making it also become nervous and perform actions associated with being nervous.

7.2.3 Object Ambiance

Object ambiance is the effect an object has on the smart ambiance. An ambiance is modified when an object with object ambiance enters the area of a smart ambiance. Object ambiance allows an area that is not inherently associated with any ambiance to have its ambiance dynamically generated by the objects within the area. Object ambiance is interesting because the cost modifiers generated by the objects may interact in unpredictable ways, facilitating emergent behaviour. Furthermore, object ambiance allows the likelihood of actions to be be influenced by the surroundings of the NPC, making the actions performed more contextually appropriate.

Object ambiance works by associating each object, be it an item or NPC, with a set of ambiance modifiers that are added to the smart ambiance in

which the object resides. Objects may instead not add their ambiance to the entire smart ambiance but only add their ambiance to NPCs that trigger certain rules belonging to the object, such as coming within a certain distance the object. In each case, the ambiance modifiers of the object are added to the set of active ambiance modifiers for the NPC affected by the object ambiance.

A scenario was constructed to demonstrate object ambiance. In the demonstration, there is a spy trying to escape from a facility as quickly as she can but without behaving suspiciously in the presence of security cameras. The area containing the spy and cameras is within a smart ambiance. The cameras have an object ambiance that is associated with an ambiance modifier that has the effect of increasing the cost of suspicious actions. This ambiance modifier is associated with the suspicious ambiance annotation. The spy can perform two actions: walk and run. Both actions have no preconditions but the run action has a lower predefined cost because it completes any goal of getting to a destination faster than walking. However, the run action is associated with the suspicious ambiance annotation. The walk action is not associated with any ambiance annotations.

The scenario is played twice: once for a spy with smart ambiance enabled and once for a spy without smart ambiance enabled. The spy that is not affected by smart ambiance runs straight from the beginning to the exit, including running past every camera. This behaviour does not make sense given the context that the spy is trying to avoid suspicion. The spy that is affected by smart ambiance performs the walk action to move down the

corridor toward the exit as that is the lowest cost action due to the cost modifiers applied by the object ambiance of the cameras. Once the spy is beyond the ambiance of the camera, she runs for the exit.

A variation of object ambiance was used in The Sims 4[1], although the research described in this chapter came a number of years earlier (Sloan *et al.*, 2011b). In the Sims 4, the objects in the environment affected the emotional state of a Sim. For example, being near a painting might cause the Sim to enter the inspired emotional state, causing the Sim to become more likely to play an instrument or paint on an easel.

7.3 Conclusions

This chapter described smart ambiance, a system that makes use of an environment to cause NPCs within the environment to prefer actions that are appropriate in the context of that environment. Three aspects of smart ambiance were demonstrated: location ambiance, event ambiance, and object ambiance. Smart ambiance adjusts the costs of actions so that more contextually appropriate actions cost less than inappropriate actions.

Smart ambiance has several strengths. Smart ambiance increases the likelihood of more contextually appropriate behaviours being selected. This was shown in all the demonstrations. Smart ambiance facilitates the emergence of behaviours unforeseen by the designer. The pedestrian crossing scenario demonstrated this. One could imagine that there might be a city filled with

[1]The Sims 4 - Maxis - http://thesims.com/en_US/home

characters with different personalities, for example, some with a nervous disposition. Although the designer might not have ever considered the situation where a cluster of nervous people might simultaneously arrive at a road crossing with less nervous people, when the situation occurred, the behaviour exhibited by normally calm pedestrians might be considered interesting. The pedestrian crossing scenario shows another smart ambiance strength in how smart ambiance creates infectious actions. Normal pedestrians were walking across the road until they felt nervous because others were running across. Smart ambiance also allows actions to be selected that might otherwise never be. Consider that the spy could perform two actions, walk and run, both with no preconditions and both that achieve the same goal of arriving at a destination. If action costs are fixed as they are in GOAP, the lower cost action of the two will allows be selected to achieve that goal. However, by making action costs change with context, a different action may be selected for particular situations. This selection of different actions may cause behaviours to be less repetitious and may be more enjoyable for players.

Smart ambiance also has some weaknesses. Functions must be implemented to model each ambiance modifier, burdening the designers. The emergent behaviour that is created by combining several ambiance modifiers might not be desirable. Also, the infectious behaviours could cause a runaway effect where behaviour no longer becomes sensible and would have to be moderated by the designer. An example of this runaway ambiance occurring would be if the nervous actions of the pedestrians spread, causing more people to become more nervous, having a domino effect that might eventually

cause the entire city to become perpetually nervous.

The demonstrations showed aspects of smart ambiance in isolation but these would ideally be combined. It is hoped that this combination of aspects would create an ambiance that results in more appropriate behaviours that players would find more enjoyable.

Chapter 8

Conclusions

There are many types of systems for selecting the behaviour of NPCs in computer games. Many games have used action planning systems as their behaviour selection system. GOAP is a popular action planning system for computer games. However, GOAP has several weaknesses.

- GOAP cannot plan for multiple goals, which results in plans that help achieve multiple goals being overlooked, and may also result in plans being selected that achieve one goal at the expense other goals held the NPC.

- GOAP cannot tell how complete a goal is and so may select actions that are more powerful than necessary to achieve a goal, e.g. using a large healing spell when only a small healing spell is necessary.

- GOAP action effects are unable to partially satisfy a precondition. This

makes it impossible to use multiple actions to satisfy a single precondition, which can cause better plans to be overlooked.

- GOAP can only select one action when there are multiple actions available with the same desired effect, all preconditions of the actions available for application are satisfied, and the plan to reach those actions costs the same amount.

The research in this thesis details how the UDGOAP system was designed to address these weaknesses in GOAP. The remainder of this section provides a summary of the contributions toward addressing these weaknesses and suggests areas of research for future development.

8.1 Summary of Contributions

The following is a brief summary of the contributions made in this thesis:

- *A review of literature regarding behaviour selection systems in computer games.* Chapter 2 described different behaviour selection systems that have been used in games, discussed why action planning systems are a promising approach, detailed some weaknesses of GOAP, and described systems based upon GOAP that addressed some these weaknesses. The chapter also defined utility and showed how utility had been used in a number of computer games.

- *A novel action planner that combines utility, drives, and smart objects to create a behaviour selection system capable of planning for mul-*

tiple goals, using partially satisfying actions, dynamically generating goals, and executing actions in a more contextually appropriate manner. Chapter 3 introduced UDGOAP, the system developed as part of this research to address weaknesses in GOAP. UDGOAP uses utility to help measure the completeness of any number of goals. UDGOAP also uses utility to be able to partially satisfy preconditions by measuring the effects of actions more accurately. UDGOAP uses drives to generate goals, allowing UDGOAP to have multiple goals of the same type and allowing the planning for all of those goals simultaneously. UDGOAP also uses smart objects to help reduce the complexity of the planning system by placing information in the environment and also uses smart objects to perform actions in a more contextually appropriate manner.

- *A study of the feasibility of using action planners for real-time computer games.* Chapter 4 described an experiment that tested if GOAP could be used in a virtual hospital environment with many NPCs. The experiment also tested how a simple utility-based system compared to GOAP regarding resource consumption. The results showed that GOAP was a viable behaviour selection system as it used an amount of resources far below the amount usually made available for a behaviour selection system in a game. The results also showed that the utility-based system used fewer resources than GOAP.

- *An empirical evaluation of UDGOAP running against GOAP in a static, single-NPC, deterministic environment.* Chapter 5 described

an experiment that took place in a virtual household like one found in The Sims. The experiment compared GOAP to UDGOAP by measuring how long it took an NPC to start from an undesirable state and reach a near-optimal state. The results showed that the NPC controlled by UDGOAP reached the near-optimal state substantially faster than the NPC controlled by GOAP.

- *An empirical evaluation of UDGOAP running against industry standard behaviour selection systems in a highly dynamic, multi-NPC, non-deterministic environment.* Chapter 6 described an experiment that took place in a virtual arena environment. The experiment compared the lifespans of a wizard in the arena under the control of the finite state machine used in Half-Life 2, GOAP, and UDGOAP. The results showed that the wizard controlled by UDGOAP lived a statistically significant time longer than the wizard controlled by the finite state machine and the wizard controlled by GOAP.

- *A novel system using smart ambiance to dynamically alter action costs to produce more contextually appropriate behaviours.* Chapter 7 described the architecture of smart ambiance and its different aspects: location ambiance, event ambiance, and object ambiance. The chapter also detailed demonstrations that had been performed using each of the three aspects. The demonstrations showed that smart ambiance could be used with GOAP to have NPCs select more contextually appropriate behaviour than NPCs using GOAP without smart ambiance.

Overall, the final UDGOAP system is satisfactory but far beneath the original expectations for the system. The original goal of this research was to create a system that would be so obviously better than the existing systems used in industry that game developers would feel compelled to use it. Instead, the UDGOAP system improves upon several aspects of GOAP but not enough to make it as enticing as originally hoped.

Improvements made by UDGOAP were not properly highlighted because of the experimental setups. This was particularly true for the arena experiment. A number of other things were left out of the simulation, such as enemies with weakness to certain elements, wizard allies and spells that have effects that are difficult to predict, though these things may have given UDGOAP a chance to show where it could do things far better than either GOAP or the Half-Life 2 finite state machine. These additions were not implemented because of time constraints and difficulty controlling the lifespan of the wizard so that he would live long enough to produce useful data but not so long that it would take too much time to perform the required number of runs.

It was interesting to watch UDGOAP perform actions that originally thought would not be wise but ended up being better than what had been thought to be the optimal behaviour in the arena. For example, it was assumed that it would be better for the wizard to use its magical ranged attacks against the antlions so that he would avoid taking damage. UDGOAP instead preferred melee attacks and saved the mana that would have been spent on ranged attacks and instead used that mana to heal damage taken

during melee combat. Another example occurred while searching for a bug in the code. The mana cost of the ranged attack had temporarily been set to zero. The wizard immediately adjusted its behaviour and performed ranged attacks instead of melee attacks. There were no weight, cost or formula changes performed to keep the NPC working in a sensible manner, instead UDGOAP immediately found the sensible behaviour itself.

Much effort had been put into the design of UDGOAP to make it simple enough that any developer could pick it up. However, this seems top have been a failed effort. UDGOAP requires a lot of work to setup. GOAP requires NPCs to be setup a particular way but UDGOAP requires even more by being needing the world to be represented a particular way in the game engine. This is because of the way UDGOAP plans based on facts rather than the more abstracted key-value pairs.

The authorship burden of NPC behaviours may actually be decreased for some situations using UDGOAP. The design of UDGOAP lends itself well to tooling so designers with little programming knowledge could use tools to author action preconditions and effects. However, it may be difficult to author tools to make it easier for these designers to tweak the drives of the NPC to prefer certain situations over others. This is because UDGOAP in its current form was not designed with tooling in mind.

UDGOAP may have performed well if it had been used on an NPC that went to a lot of different locations filled with a variety of objects. A companion in a game like Skyrim[1] who accompanies the player throughout their

[1]The Elder Scrolls: Skyrim - Bethesda Game Studios - http://www.elderscrolls.com

adventure into a number of varied environments may be an excellent use of UDGOAP because of the number of opportunities to interact with world objects in situations unforeseen to the designer. However, this would be difficult to evaluate because the player would have to play for a long time to see how the companion behaves in several situations. It would also require a lot of authoring to setup all of the items in the game to use the UDGOAP system, so much so that it would only be possible for use in games with a large team of developers.

The simulation system in UDGOAP was added late in the UDGOAP development process. The simulation was added because UDGOAP was failing to consider how the world might change during the execution of a plan, in particular the behaviour of NPCs that were not involved in the plan. GOAP was designed for the enemy NPCs in F.E.A.R.[1] where they often only need to think of the player as the only target that can cause them harm. The wizard, however, had to consider several antlions at all times. I felt that the simulation was good enough to capture how these other NPCs would hurt the wizard, causing the wizard to be less cavalier, no longer charging into the middle of a horde of antlions and instead picking off antlions at the edge of a pack. I was disappointed that there was not time to expand upon this simulation concept, allowing the simulation of antlions that explode upon death, hurting the wizard, or a heat shields for the wizard that hurt nearby antlions passively without the performance of any action.

[1]F.E.A.R. - Monolith Productions - http://www.fear3.co.uk/the-game.html

I believe that players would notice the interesting NPC behaviours in situations where an NPC should consider multiple goals. It was interesting when the UDGOAP wizard used his blast to kill nearby enemies when the wizard became surrounded. This happened instead of ranged or melee attacks because UDGOAP calculated that the health that would be lost fighting the antlions one by one was more valuable than the mana lost performing the blast. Players may notice similar behaviour in other games. For example, where a soldier has to decide if his grenade ammunition is more valuable than the health it believes might be lost in the oncoming skirmish.

The added complexity of UDGOAP over GOAP may be worth the investment but only for reasonably large games and only for games with NPCs with long lifespans. In small games, the developer can know all of the situations in which an NPC will exist so these situations can be specifically coded for. In games where enemies do not live long, for example like zombies in Left 4 Dead[1], enemies are fun because they single-mindedly charged at the player to be killed immediately. Using UDGOAP on such enemies would be a waste of resources.

8.2 Open Problems and Future Work

The UDGOAP system was developed to address a number of weaknesses in GOAP. However, a number of questions arose during the development of UDGOAP that, if answered, could yield interesting research. This section

[1]Left 4 Dead - Turtle Rock Studios, Valve Corporation - http://www.l4d.com

describes directions for future research relating to UDGOAP.

Every experiment described in this thesis was quantitative. This is common among theses in the game AI domain but it does overlook an important aspect of that differentiates many games from simulations. Game AI is not just meant to be about making an non-player opponent as strong as possible, the opponent should also be fun to play against and ideally even behave in a manner more like a human to add to the realism of the experience. Although UDGOAP achieved goals more effectively than GOAP, A
B tests could be performed in future to help determine which system is more fun to play with and which system behaves in the most realistic manner. This could be done by having participants watch a GOAP NPC and a UDGOAP NPC and have them rate how fun and realistic the behaviours of these NPCs are for both the arena and Sims environments. It would also be interesting to see how a player controlled wizard would fare in these environments compared to the NPCs. It would also be interesting to see which system players would have preferred to have controlling an ally.

GOAP has the ability to use a heuristic during plan formulation. This reduces the number of actions that needed to be considered when searching for the lowest cost solution and makes planning more efficient. Heuristics are key in making path planning solutions feasible for use in real-time games. The number of possible paths available during a path planning search can be substantial whereas the number of possible plans available during plan formulation is usually quite small, even for big budget games. This makes a heuristic in GOAP less important than a heuristic in path planning because

GOAP contains fewer possibilities. Having an admissible heuristic (Korf & Reid, 1998) to guide plan formation in UDGOAP would be a great improvement as the environments in computer games grow in complexity.

The soft goal selection mechanism in UDGOAP was quite simple. There was no filtering process on soft goals. This meant that the UDGOAP planner considered many more plans than GOAP, which only searched for a plan that achieved one goal. Although this did not cause any performance related issues, we feel that there is room for a more elegant solution that would select only a smaller set of soft goals, focusing on soft goals that are more likely to be achieved in tandem.

Actions in UDGOAP were associated with a world effects generator that added effects based on the world state at the time of the execution of the action. This leads to situations where the planner will check for effects that could not possibly apply to certain actions. For example, the goto action checks if the execution of the action puts the consumer of the action in range of an enemy. This happens regardless of if an enemy is present in the environment. There may be another way to implement world effects by embedding more information into smart objects and their environment instead of into the actions so that world effects are only checked in situations where they are able to occur.

The UDGOAP system is designed to work for just a single agent. There is massive opportunity for NPCs in an environment to make plans that require actions from other NPCs. Using a utility-based planning system that allowed multiple NPCs to coordinate their plans, it would be possible to find more

effective plans than those that could be found by either NPC working on its own.

There is still a lot of work that can be done to improve the state of behaviour selection for NPCs in computer games. Our hope is that the research in this thesis can be built upon to create a system that will provide compelling NPC behaviour in computer games that will be enjoyed by thousands of people for many years.

Appendix A

Detailed UDGOAP Worked Example

To illustrate how UDGOAP works, this section will run through the entire process of a UDGOAP NPC selecting a behaviour, spanning the sensing state through to the output of the UDGOAP planner. The example takes place from the perspective of a wizard named wizard that is in an environment where there is a mana potion named mana_potion and one enemy goblin named goblin.

The initial state of the wizard is as follows[1]:

- sensors: { internal_sensor, potion_sensor, enemy_sensor}
- facts: {}

[1]Some details have been omitted for brevity, e.g. the full list of actions the wizard supplies.

- supplied actions: {melee}
- consumable actions: {melee, lightning, goto, drink_mana_potion}
- drives: {

  ((name: kill_all_enemies, goal generators: {enemy_spotted}, goals: {}, eval func: linear, weight: 1)),

  ((name: maximize_health, goal generators: {self_below_max_health}, goals: {}, eval func: linear, weight: 1)),

  (name: maximize_mana, goal generators: {self_below_max_mana}, goals: {}, eval func: linear, weight: 1)}
- internal state: {position: (0, 0, 0), health: 15, mana: 0}
- top-level utility function: linear_with_bad_death

There are four consumable actions available to the wizard. The melee action causes the wizard to engage in hand-to-hand combat with an enemy. The lightning action causes the wizard to shoot a lightning bolt from his hands, killing his target but consuming some of the wizard's mana. The goto action causes the wizard to move to some destination. The drink_mana_potion action causes the wizard to consume the target mana potion. The actions are defined as follows:

- name: melee, preconditions: {(key: consumer_position_change, evaluation function: supplier_within_melee_distance)}, effects: {(key: supplier_health_decrease, application function: melee_damage)}, world effect generators: {}

- name: lightning, preconditions: {(key: consumer_mana_increase, evaluation function: supplier_enough_mana_lightning)}, effects: {(key: consumer_mana_decrease, application function: reduce_mana_lightning), (key: supplier_health_decrease, application function: lightning_damage)}, world effect generators: {}

- name: goto, preconditions: {}, effects: {(key: consumer_position_change, application function: goto_supplier)}, world effect generators: {nearby_enemy_damage}

- name: drink_mana_potion, preconditions: {(key: consumer_position_change, evaluation function: within_potion_distance)},
  effects: {(key: consumer_mana_increased, application function: increase_mana_by_supplier), (key: supplier_existence_ended, effect function: destroy_potion)}, world effect generators: {}

The goblin supplies the melee, lightning and goto actions. The mana potion supplies the drink and goto actions.

When the simulation starts, the wizard runs its sensing process (described in Section 3.5.1) when the behaviour selection system has been queried for a behaviour after, for example, 2000 milliseconds has passed in the game. This first runs the internal_sensor sensing process, then the potion_sensor sensing process, and the the enemy_sensor sensing process. This sensor sweep generates the facts shown in Table A.1 which are saved into the memory of the wizard planning NPC:

The drives will then be updated (as described in Section 3.5.2). The drive

id	entity	attribute	value	timestamp
1	wizard	position	(0, 0, -10)	2.00
2	wizard	health	15	2.00
3	wizard	mana	0	2.00
4	mana_potion	position	(0, 0, 10)	2.00
5	goblin	position	(0, 0, 20)	2.00
6	goblin	health	30	2.00

Table A.1: Facts known to wizard after first round of sensing.

update process first generates new goals, then evaluates the completeness of those goals, and then evaluates the satisfaction for each drive according to the completeness of its goals. The goal generators iterate through their triggers, checking if any have been activated by the up-to-date set of facts. The kill_all_enemies drive checks its enemy_spotted goal generator, which looks for facts regarding the position of enemy units where a goal to kill that enemy is not already present in the goal set of the kill_all_enemies drive. The goal generator trigger for this drive will be activated by fact 6 (which contains the details of the health of the goblin). The goal generated is the kill_enemy goal because that is the goal associated with the enemy_spotted goal generator. The goals generated will be as follows:

- id: 1, name: kill_enemy, weight: 2.0, conditions:

  {key: supplier_health_decrease, fact id: 6, evaluation function: inverse_linear}, evaluation function: inverse_linear, removal triggers: enemy_no_health, completeness: null

This goal is added to the goal set of the kill_all_enemies drive, as it was the goal generator belonging to the drive that created the goal. The completeness of the goal is updated using the inverse_linear evaluation function of the goal[1].

[1]Each goal evaluation function depends on the satisfaction of the conditions the goal

In this instance, the goblin has 30 health, which is its maximum, giving a normalized value of 1.0, which when inverted gives a value of 0.0, indicating that the goal of killing the goblin has no progress made toward its completion. Hence, because the goblin is on full health, the completeness value associated with the goal of killing the goblin is 0.0.

The triggers for the goal generators in the maximize_health and maximize_mana drives are checked against the up-to-date facts. These goal generators trigger upon the fact of health and mana being less than their maximum. The maximum health and mana of the wizard is 100. Currently, the wizard has 15 health and 0 mana, according to facts 2 and 3, referring to the health and mana of the wizard, respectively. These facts cause the triggers to fire for the self_below_max_health and self_below_max_mana goal generators because the predicates in those triggers test if the current value of the relevant fact is less than the maximum value that the attribute can have. This triggering generates the following goals with their completeness calculated using the linear evaluation function. These goals will be added to the their parent drive after the goals have been generated:

- id: 2, name: maximize_health, weight: 1.0, conditions: {key: consumer_health_increase, fact id: 2, evaluation function: linear}, evaluation function: linear, removal triggers: self_max_health, completeness: 0.15

- id: 3, name: maximize_mana, weight: 1.0, conditions: {key: con-

depends on but because every goal in this worked example only has one condition, the condition satisfaction functions have been omitted for brevity.

sumer_mana_increase, fact id: 3, evaluation function: linear}, evaluation function: linear, removal triggers: self_max_mana, completeness: 0.0

After all goal triggers have been tested, the utility of the state for each drive based on the completeness of the goals associated with that drive is updated. The evaluation function for each drive in this example will use a weighted sum of the completeness of the goals belonging to the drive. The maximize_health drive uses the linear evaluation function and takes its one goal of maximizing the health of the wizard as input, the goal with the ID of 2, and because the goal has a completeness of 0.15 and a weight of 1.0, returns a utility rating of 0.15 for the drive (0.15 * 1.0). This utility rating is saved in the drive. The same process of rating and saving the utility is performed on the maximize_mana drive, giving it a rating of 0.0. The same utility calculation process is performed on the kill_all_enemies drive, returning a rating of 0.0 because the goal to kill a goblin has not made any progress toward completion.

The wizard now searches for objects to which it can apply actions. For simplicity in the example, it simply selects the objects within a particular range of the wizard, which includes the goblin and the mana potion. For each of these objects, a pair is created for each action they supply. This results in the following object action pairs:

The set of all goals belonging to all drives, the set of up-to-date facts, the set of consumable actions for the wizard, and the set of object-action

object	action
mana_potion	drink_mana_potion
mana_potion	goto
goblin	lightning
goblin	melee
goblin	goto

Table A.2: Wizard object-action pairs.

pairs are passed into the planner to generate a behaviour that maximizes the utility of the world state to the set of drives of the NPC. The planner will first generate root plan states (as described in Section 3.8.1) and then generates successors from this until either the planner finds the plan state with the highest utility and with all goal conditions satisfied, or it will return an empty plan if no plans are found.

The initial plan state generation phase will first set up a root plan state: (id: 1, hard goal: null, soft goal ids: {1, 2, 3}, rating: 0, generative action: null, parent: null). The generative action is null as this state was not generated through actions, the predecessor plan state is null as this is the root plan state not generated from any previous state, and the utility rating is calculated.

After root plan state generation, we then generate successor plan states for each action with an effect key matching any condition key in the set of goals. The melee and lightning actions have an effect with the supplier_health_decrease key, which matches the condition key for the goal of killing the goblin, and the drink_mana_potion action has an effect with the consumer_mana_increase key, which matches the condition key for the goal of maximizing mana. Each of the actions that contained a match will have its keys mapped to facts be-

cause later this action will become the generative action of a successor plan state of the root plan state.

Each of the three actions that had an matching effect key has its preconditions and effects mapped by combining information about the action with the up-to-date set of facts (as described in Section 3.8.4). This key to fact mapping results in the following action set:

- name: melee, preconditions: {(key: consumer_position_change, fact ID: 1, evaluation function: supplier_within_melee_distance)}, effects: {(key: supplier_health_decrease, fact ID: 6, application function: melee_damage)} world effect generators: {} supplier: goblin, consumer: wizard

- name: lightning, preconditions: {(key: consumer_mana_increase, evaluation function: supplier_enough_mana_lightning)}, effects: {(key: consumer_mana_decrease, fact ID: 3, application function: reduce_mana_lightning), (key: supplier_health_decrease, fact ID: 6, effect function: lightning_damage)}, world effect generators: {}, supplier: goblin, consumer: wizard

- name: drink_mana, preconditions: {(key: consumer_position_change, fact id: 4, evaluation function: within_potion_distance)}, effects: {(key: consumer_mana_increased, fact ID: 3, application function: increase_mana_by_supplier), (key: supplier_existence_ended, fact ID: 4, application function: destroy_potion)}, world effect generators: {}, supplier: mana_potion, consumer: wizard

The actions and their world effect generators now need to be applied, altering the facts that will be associated with each successor plan state (as described in Section 3.8.5). We will focus just on the melee action. The melee action applies its only effect. This effect has its melee_damage application function run, which takes the fact ID of the effect and returns a new version of the a health fact with 30 less value than the one passed in. This effect is associated with fact 6, the health of the goblin, so 30 is subtracted from its current value and a new fact referring to goblin health is generated: (id: 6, entity: goblin, attribute: health, value: 0, timestamp: 2.00). This action has no world effect generator so action application is complete.

The world that would exist if the melee action was applied is then rated for utility by using the most up-to-date facts as input to the relevant soft goals, which will be input to their associated drive, which will be input to the top-level utility function.

The planner first gathers all of the newest facts before calculating goal completeness. Here the newest facts are the set of initial facts but with the health value of the goblin at 0. The completeness functions of the soft goals are then updated based on these facts. Focusing just on the goal related to killing the goblin, (id: 1, name: kill_enemy, weight: 2.0, conditions: {key: supplier_health_decrease, fact id: 6, evaluation function: inverse_linear}, evaluation function: inverse_linear, removal triggers: enemy_no_health, completeness: null), we can evaluate goal completeness. Fact 6, which refers to

the fact of the newly reduced goblin health, is input to the inverse_linear evaluation function. The normalized inverse of the goblin health of 0 is calculated, returning 1.0 for the completeness value of the goal.

The drive satisfaction is calculated using the goal completeness value and the goal weight as input to the linear evaluation function for the kill_all_enemies drive, which also returns a value of 1.0 because, in this instance, the evaluation function combines goal completeness and goal weight to create a weighted sum. The linear evaluation function maximize_health drive returns 0.15 because the health of the wizard is 15 and the goal associated with the drive has a weight of 1.0. The linear evaluation function maximize_mana drive returns 0.0 because the mana of the wizard is 0 and the goal associated with the drive has a weight of 1.0.

The top-level utility function of the wizard receives the satisfaction value and weight of all drives as input and returns a value that will become the utility rating of the plan state that was generated by the melee action. The linear_with_bad_death top-level utility function gets a weighted sum of the satisfaction value and weight of each drive and sums them, then normalizes this sum. The linear_with_bad_death top-level utility function considers the maximize_health drive to be special, outputting a utility rating of 0.0 if the satisfaction level of the maximize_health drive is 0.0. This is done to reflect the idea that no matter how good the state is for the satisfaction of the other drives, if the health of the wizard is 0, he is dead and everything else is irrelevant. The linear_with_bad_death function takes the drive satisfaction and weight triples: (name: kill_all_enemies, satisfaction: 1.0, weight: 1.0), (name:

maximize_health, satisfaction: 0.15, weight: 1.0), (name: maximize_mana, satisfaction: 0.0, weight: 1.0), which gives (1.0 * 1.0) + (0.15 * 1.0) + (0.0 * 1.0), giving 1.15, which is normalized against a maximum weight sum of 3.0, finally giving 0.38 as the utility of the state generated from the application of the melee action.

The same calculation of goal completeness, drive satisfaction, and top-level utility is calculated for the plan states generated from the application of the drink_mana_potion and lightning actions. This particular potion gives 40 mana. The final result is three rated plan states shown in Table A.3 (some details omitted for brevity).

generative actions	**utility**
{melee}	0.38
{lightning}	0.38
{drink_mana}	0.18

Table A.3: Plan states after initial state generation.

A new plan state is then generated for each these actions. These plan states will be successors to the root plan state. These plan states would have their generative actions and ratings set using the values just calculated and shown in Table A.3. The hard goals of these plan states is the set of preconditions not satisfied for the respective action of the plan state. These plan states are as follows.

- ID: 2, hard goal: {(fact key: consumer_position_change, fact id: 1, eval func: supplier_within_melee_distance)}, soft goals: {1, 2, 3}, rating: 0.38, generative action: (name: melee, consumer: wizard, supplier: goblin, ...), parent plan state id: 1

- ID: 3, hard goal: {(fact key: consumer_mana_increase, fact id: 3, eval func: supplier_enough_mana_lightning)}, soft goals: {1, 2, 3}, rating: 0.38, generative action: (name: lightning, consumer: wizard, supplier: goblin, ...), parent plan state id: 1

- ID: 4, hard goal: {(key: consumer_position_change, fact id: 4, evaluation function: within_potion_distance)}, soft goals: {1, 2, 3}, rating: 0.18, generative action: (name: drink_mana, consumer: wizard, supplier: mana_potion, ...), parent plan state id: 1

These new plan states are then added to the set of plan states to be considered in the main plan loop. The main loop is entered and continues because there is at least one plan state to be developed in the set of plan states. The plan state with the highest utility rating is selected for development. In the case of a draw, the first will be selected, which means that the melee plan state is selected. A check is performed to see if all hard goal conditions of the selected plan state are satisfied by running the supplier_within_melee_distance evaluation function with fact 1, referring to the position of the wizard. The evaluation returns false so planning continues by trying to find actions applicable to the unsatisfied condition, i.e. actions with the consumer_position_change key in their set of effects (as described in Section 3.8.3). The goto action has that key in its effects set. The keys of the effects of the goto action are mapped to facts.

The action application phase used at this point has two differences from the phase used in the beginning when generating plan states as successors to

the root plan state. The first difference is that those initial plan states were generated from single actions being applied, but the plan state currently being developed, that is associated with the goto action, is a successor to the plan state generated through the application of the melee action. The application of both the goto and melee actions will need to be simulated (in forward order) so as to determine the utility of the plan state that would arise from applying these actions. These actions will be applied to the initial facts. For example, the effect of the melee action will be applied to its respective plan state, which changes the fact relating to the health of the enemy from the enemy being on full health to the enemy being full health minus the damage done by the melee action.

The goto action that just had its keys mapped to actions is applied and simulated first. Its goto_supplier application function creates a new fact with a new position for the wizard that is next to the position of the goblin who supplied the action. The nearby_enemy_damage world effect generator, which was specified as part of the goto action at design-time, generates an effect because the generator has a trigger that activates when the consumer comes into range of a supplier melee attack. The effect generates a fact where the health of the wizard is reduced by the melee damage of the goblin, which would set the wizard health to zero. The application of the melee action is ignored because the wizard would be dead at this time because its health is zero. The utility of this plan state is rated at 0.0 because the linear_with_bad_death top-level utility function returns zero if the soft goal relating to the health of the consumer is zero. A new plan state is generated

with the goto generative action and a utility rating of zero. The new plan state is added to the set of plan states and the main loop performs another iteration. At this time in plan formulation, the set of potential actions looks as follows.

generative actions	**utility**
{melee, goto}	0.0
{lightning}	0.38
{drink_mana_potion}	0.18

Table A.4: Plan states after one iteration of the main loop.

The plan state generated from the lightning action is selected for further development as it is the highest rated plan state. The condition of the hard goal is that the consumer has 20 mana and the only action found to satisfy the condition is the drink_mana_potion action performed on the potion object. This results in a plan state where the wizard has 40 additional mana points and where the goblin is dead. This plan state would have the rating of 0.51, giving the newest set of plan states.

generative actions	**utility**
{melee, goto}	0.0
{lightning, drink_mana}	0.51
{drink_mana}	0.18

Table A.5: Plan states after two iterations of the main loop.

At the next iteration of the plan loop, the plan state generated from the lightning and drink_mana actions is selected for development because it has the highest rating. The next unsatisfied condition of the hard goal has the consumer_position_change key. The goto action supplied by the potion object has an effect matching this key. The facts are mapped for the goto action.

The plan state generated from that action is rated. The rating is the same as its parent state in this instance because no goal depends on the position of the wizard. The new plan state is added to the set of plan states that now is as follows.

generative actions	**utility**
{melee, goto}	0.0
{lightning, drink_mana, goto}	0.51
{drink_mana}	0.18

Table A.6: Plan states after three iterations of the main loop.

The highest rated plan state is selected for development. All of the conditions of the hard goal have been satisfied so the plan is extracted from the plan state for execution by the wizard. The final plan is to go to the mana potion, drink the potion, and shoot lightning at the goblin.

A.1 Summary

This chapter described the UDGOAP behaviour selection system. This system attempts to overcome a number of challenges faced by planning systems in modern computer games, namely the problems of:

- planning for multiple goals.
- creating an accurate estimate of the efficacy of a plan and how close to completion a goal is.
- being limited to a single action to satisfy a single precondition.
- brittle behaviours that can break when the values of items change e.g.

the strength of a particular enemy increasing, invalidating behaviours that were made based on the strength of that enemy.

UDGOAP plans for multiple goals by using a utility function that is based on the completeness of a set of conditions belonging to the soft and hard goals. UDGOAP tells how effective a plan is by judging the utility of the state that the plan is likely to create. UDGOAP is able to use an action to partially satisfy a precondition, which allows it to use multiple actions to satisfy a single precondition. UDGOAP creates robust behaviours that are not invalidated even when the value of items change by not using a predefined set of actions costs or a predefined behaviour to achieve some goal.

References

Aamodt, A. & Plaza, E. (1994). Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, **7**, 39–59. 23

Abaci, T., Ciger, J. & Thalmann, D. (2005). Planning with Smart Objects. *WSCG (Short Papers)*, 25–28. 46, 49

Agre, P. & Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the sixth National conference on Artificial intelligence*, vol. 1, 268–272, Seattle. 14

Aha, D., Molineaux, M. & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. *Case-based reasoning research and development*, 5–20. 17

Alexander, B. (2002). The beauty of response curves. *AI Game Programming Wisdom*, **1**. 44, 69, 93

Arrabales, R., Ledezma, A. & Sanchis, A. (2009). Towards conscious-like behavior in computer game characters. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 217–224, IEEE. 18

Baars, B. (1993). *A cognitive theory of consciousness*. Cambridge University Press. 18, 20

Bauckhage, C. & Thurau, C. (2004). Towards a fair'n square aimbot–Using mixtures of experts to learn context aware weapon Handling. In *Proc. GAME-ON*, 20–24, Citeseer. 1, 139

Benton, J., van den Briel, M. & Kambhampati, S. (2007). A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems. In *Proceedings of ICAPS*. 19, 43

Bourg, D.M. & Seemann, G. (2004). *AI for game developers*. " O'Reilly Media, Inc.". 16

Bradley, J. & Hayes, G. (2005). Adapting reinforcement learning for computer games: Using group utility functions. In *IEEE Symposium on Computational Intelligence and Games*, 133–140, Citeseer. 44

Brom, C. (2007). *Action Selection for Virtual Humans in Large Environments*. Ph.D. thesis, Charles University in Prague. 46

Buckland, M. (2005). *Programming game AI by example*. Jones & Bartlett Learning. 14

Burelli, P. & Jhala, A. (2009). Dynamic artificial potential fields for autonomous camera control. *Artificial Intelligence and Interactive Digital Entertainment*. 1

Campbell, M., Hoane, A.J. & Hsu, F.h. (2002). Deep blue. *Artificial intelligence*, **134**, 57–83. 11

Cerpa, D. (2008). An advanced motivation-driven planning architecture. *AI Game Programming Wisdom*, **4**, 373–382. 40, 41, 48

Champandard, A.J. (2003). An Overview of Navigation Systems. *AI Game Programming Wisdom 2*. 1

Champandard, A.J. (2007a). Living with the simsai: 21 tricks to adopt for your game. 46

Champandard, A.J. (2007b). Top 10 most influential game ai. 2

Champandard, A.J. (2010). Programming utility systems for single decisions in practice. 43

Champandard, A.J. (2011a). Open challenges in first person shooter (fps) ai technologies. 51

Champandard, A.J. (2011b). Turn-based ai in greed corp from neural networks to minimax. 45

Champandard, A.J. (2012). Hierarchical task networks for mission generation and real-time behavior. 22

Choi, D., Konik, T., Nejati, N., Park, C. & Langley, P. (2007). A believable npc for first-person shooter games. In *Proceedings of the third annual artificial intelligence and interactive digital entertainment conference*, 71–73. 18, 19

Chu-Carroll, M.C. (2008). Solving tic-tac-toe: Game tree basics. 12

Cole, N., Louis, S. & Miles, C. (2004). Using a genetic algorithm to tune first-person shooter bots. In *Evolutionary Computation, 2004. CEC2004. Congress on*, vol. 1, 139–145, IEEE. 1

Cornelius, R., Stanley, K. & Miikkulainen, R. (2006). Constructing adaptive ai using knowledge-based neuroevolution. *AI Game Programming Wisdom*, **3**, 693–708. 17

de Sevin, E. & Thalmann, D. (2005). A motivational model of action selection for virtual humans. In *Computer Graphics International 2005*, 213–220, IEEE. 46

DePristo, M. & Zubek, R. (2001). being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, 31–34. 20

Do, M., Benton, J., Van Den Briel, M. & Kambhampati, S. (2007). Planning with goal utility dependencies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-2007)*, 1872–1878. 20, 45

Dunn, O.J. (1961). Multiple comparisons among means. *Journal of the American Statistical Association*, **56**, 52–64. 134

Dybsand, E. (2001). A generic fuzzy state machine in c+. *Game Programming Gems 2*, 337. 14

Evans, R. (2002). Varieties of learning. *AI Game Programming Wisdom*, **2**. 15

Fikes, R. & Nilsson, N. (1971). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, **2**, 189–208. 19

Fikes, R. & Nilsson, N. (1972). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, **2**, 189–208. 21, 25, 26, 28

Flórez-Puga, G., Gomez-Martin, M., Gomez-Martin, P., Díaz-Agudo, B. & González-Calero, P. (2009). Query-enabled behavior trees. *Computational Intelligence and AI in Games, IEEE Transactions on*, **1**, 298–308. 17

Fountas, Z., Gamez, D. & Fidjeland, A. (2011). A neuronal global workspace for human-like control of a computer game character. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 350–357, IEEE. 18

Frank, I. (1999). Explanations count. In *AAAI 1999 Spring Symposium on Artificial Intelligence and Computer Games*, 77–80. 1

Fu, D. & Houlette, R. (2002). Putting ai in entertainment: An ai authoring tool for simulation and games. *Intelligent Systems, IEEE*, **17**, 81–84. 14

Fu, D. & Houlette, R. (2004). Constructing a decision tree based on past experience. *AI Game Programming Wisdom*, **2**, 567–577. 15

Funge, J. (1999). *AI for computer games and animation: A cognitive modeling approach*. AK Peters. 46

Galli, L., Loiacono, D. & Lanzi, P. (2009). Learning a context-aware weapon selection policy for unreal tournament iii. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 310–316, IEEE. 1

Garces, S. (2006). Extending simple weighted-sum systems. *AI Game Programming Wisdom*, **3**, 331–339. 44, 45

Generation5 (2005). Interview with jeff hannan. 17

Gerevini, A. & Long, D. (2005). Plan constraints and preferences in pddl3. *Technichal Report, Department of Electronics for Automation, University of Brescia, Italy*. 37

Ghallab, M., Nau, D. & Traverso, P. (2004). *Automated Planning: Theory & Practice*. Morgan Kaufmann. 21, 22, 25, 26, 27, 36, 42, 43

Gorniak, P. & Davis, I. (2007). Squadsmart: Hierarchical planning and coordinated plan execution for squads of characters. In *Proceedings of the*

3rd artificial intelligence and interactive digital entertainment conference. AAAI Press, Menlo Park, 14–19. 22

Haddawy, P. & Hanks, S. (1998). Utility Models for Goal-Directed, Decision-Theoretic Planners. *Computational Intelligence*, **14**, 392–429. 45

Harmon, V. (2002). An economic approach to goal-directed reasoning in an rts. *Ai Game Programming Wisdom*, **3**, 402–410. 44

Hart, P., Nilsson, N. & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, **4**, 100–107. 32

Hawes, N. (2003). *Anytime Deliberation for Computer Game NPCs*. Citeseer. 19, 22

Hawes, N. (2004). *Anytime deliberation for computer game NPCs*. Ph.D. thesis, University of Birmingham. 22

Hawes, N. (2011). A survey of motivation frameworks for intelligent systems. *Artificial Intelligence*, **175**, 1020–1036. 4, 40

Hefny, A., Hatem, A., Shalaby, M. & Atiya, A. (2008). Cerberus: Applying supervised and reinforcement learning techniques to capture the flag games. In *Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, Stanford, California*. 17

Higgins, D. (2002). How to Achieve Lightning-Fast A*. *AI Game Programming Wisdom*, 133–145. 1

Hoang, H., Lee-Urban, S. & Muñoz-Avila, H. (2005). Hierarchical plan representations for encoding strategic game ai. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. 14, 22

Hochberg, Y. (1988). A sharper bonferroni procedure for multiple tests of significance. *Biometrika*, **75**, 800–802. 134

Houlette, R. & Fu, D. (2003). The Ultimate Guide to FSMs in Games. *AI Game Programming Wisdom*, **2**. 14

Isla, D. (2005). Handling complexity in the halo 2 ai. In *Game Developers Conference*, vol. 12. 15

Isla, D. (2008). Halo 3-building a better battle. In *Game Developers Conference*. 15

Jaidee, U., Muñoz-Avila, H. & Aha, D. (2011a). Case-based learning in goal-driven autonomy npcs for real-time strategy combat tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, 43–52. 17

Jaidee, U., Muñoz-Avila, H. & Aha, D. (2011b). Integrated learning for goal-driven autonomy. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, 2450–2455, AAAI Press. 20

Jang, S., Yoon, J. & Cho, S. (2009). Optimal strategy selection of non-player character on real time strategy game using a speciated evolutionary

algorithm. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 75–79, IEEE. 17

Kallmann, M. (2001). *Object Interaction in Real-Time Virtual Environments*. Ph.D. thesis, Citeseer. 46

Kallmann, M. & Thalmann, D. (1998). Modeling objects for interaction tasks. In *Proc. Eurographics Workshop on Computer Animation and Simulation*, vol. 98, 73–86. 4, 45, 46

Khoo, A. & Zubek, R. (2002). Applying inexpensive ai techniques to computer games. *Intelligent Systems, IEEE*, **17**, 48–53. 107

Khoo, A., Dunham, G., Trienens, N. & Sood, S. (2002). Efficient, realistic npc control systems using behavior-based techniques. In *AAAI Spring Symposium on Artificial Intelligence and Computer Games*, 46–51. 14

Kline, D. (2011). The ai director of dark spore. 1

Knuth, D.E. & Moore, R.W. (1976). An analysis of alpha-beta pruning. *Artificial intelligence*, **6**, 293–326. 12

Kolhoff, P. (2008). Level up for finite state machines: An interpreter for statecharts. *AI Game Programming Wisdom*, **4**, 317–332. 14

Korf, R.E. & Reid, M. (1998). Complexity analysis of admissible heuristic search. In *AAAI/IAAI*, 305–310. 162

Kruskal, W.H. & Wallis, W.A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, **47**, 583–621. 134

Kumar, V. (1992). Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine*, **13**, 32. 23

Laird, J. & VanLent, M. (2001). Human-level ai's killer application: Interactive computer games. *AI magazine*, **22**, 15. 16

Laird, J., Rosenbloom, P. & Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine learning*, **1**, 11–46. 20

Laird, J., Newell, A. & Rosenbloom, P. (1987). Soar: An architecture for general intelligence. *Artificial intelligence*, **33**, 1–64. 16, 18, 20

Laird, J.E. (2001). It knows what you're going to do: adding anticipation to a quakebot. In *Proceedings of the fifth international conference on Autonomous agents*, 385–392, ACM. 4, 18, 20, 86

Laird, J.E. & Nielsen, E. (1994). Coordinated behavior of computer generated forces in tacair-soar. *AD-A280 063*, **1001**, 57. 19

Laming, B. (2008). The marpo methodology: Planning and orders. *AI Game Programming Wisdom*, **4**, 239–255. 14

Langley, P. & Choi, D. (2006). A unified cognitive architecture for physical npcs. In *Proceedings of the National Conference on Artificial Intelli-*

gence, vol. 21, 1469, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 18

Lau, N. (2008). Knowledge-based behavior system: A decision tree/finite state machine hybrid. *AI Game Programming Wisdom*, **4**. 15

Li, N., Stracuzzi, D., Cleveland, G., Könik, T., Shapiro, D., Molineaux, M., Aha, D. & Ali, K. (2009). Constructing game npcs from video of human behavior. In *Proceedings of the Fifth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 64–69. 18, 19

Li, Z., Sim, C. & Low, M. (2007). A Survey of Emergent Behavior and its Impacts in NPC-Based Systems. In *Industrial Informatics, 2006 IEEE International Conference on*, 1295–1300, IEEE. 139

Lichocki, P., Krawiec, K. & Jaśkowski, W. (2009). Evolving teams of cooperating npcs for real-time strategy game. *Applications of Evolutionary Computing*, 333–342. 17

Lim, C., Baumgarten, R. & Colton, S. (2010). Evolving behaviour trees for the commercial game defcon. *Applications of Evolutionary Computation*, 100–110. 17

Luck, M. & Aylett, R. (2000). Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments. *Applied Artificial Intelligence*, **14**, 3–32. 140

MacNamee, B. & Cunningham, P. (2003). Creating socially interactive no-player characters: The μ-siv system. *Int. J. Intell. Games & Simulation*, **2**, 28–35. 17

MacNamee, B., Dobbyn, S., Cunningham, P. & OSullivan, C. (2002). Men behaving appropriately: Integrating the role passing technique into the aloha system. In *Proceedings of the AISB02 symposium: Animating Expressive Characters for Social Interactions (short paper)*, 59–62, Citeseer. 46

Magerko, B., Laird, J., Assanie, M., Kerfoot, A., Stokes, D. *et al.* (2004). Ai characters and directors for interactive computer games. *Ann Arbor*, **1001**, 48109–2110. 18

Maggiore, G., Santos, C., Dini, D., Peters, F., Bouwknegt, H. & Spronck, P. (2013). Lgoap: adaptive layered planning for real-time videogames. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8, IEEE. 39

Mark, D. (2009). *Behavioral Mathematics for Game AI*. Course Technology Cengage Learning. 43, 49

Mark, D. (2010). *Behavioural Mathematics for Game AI* . Delmar Publishing. 3

Mateas, M. & Stern, A. (2002). Towards integrating plot and character for interactive drama. *Socially Intelligent NPCs*, 221–228. 1

Meneguzzi, F. & Luck, M. (2007). Motivations as an abstraction of meta-level reasoning. *Multi-NPC Systems and Applications V*, 204–214. 22

Molineaux, M., Klenk, M. & Aha, D. (2010). Goal-driven autonomy in a navy strategy simulation. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 1548–1554. 19, 20

Muñoz-Avila, H., Aha, D., Jaidee, U., Klenk, M. & Molineaux, M. (2010a). Applying goal driven autonomy to a team shooter game. In *Proceedings of FLAIRS*, 465–470. 20, 22

Muñoz-Avila, H., Jaidee, U., Aha, D. & Carter, E. (2010b). Goal-driven autonomy with case-based reasoning. *Case-Based Reasoning. Research and Development*, 228–241. 20

Nareyek, A. (1998). A planning model for agents in dynamic and uncertain real-time environments. In *Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments at the Fourth International Conference on Artificial Intelligence Planning Systems*, 7–14. 24

Nareyek, A. (1999). Applying Local Search to Structural Constraint Satisfaction. In *In Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management*, Citeseer. 19

Nareyek, A. (2000). Open World Planning as SCSP. In *AAAI-2000 Workshop on Constraints and AI Planning*, 35–46. 24

Nareyek, A. (2001a). *Constraint-based NPCs: an architecture for constraint-based modeling and local-search-based reasoning for planning and scheduling in open and dynamic worlds*. Springer-Verlag. 20

Nareyek, A. (2001b). Review: Intelligent npcs for computer games. *Computers and Games*, 414–422. 23

Newell, A. (1994). *Unified theories of cognition*, vol. 187. Harvard University Press. 18

Ontañón, S., Mishra, K., Sugandh, N. & Ram, A. (2007). Case-based planning and execution for real-time strategy games. *Case-Based Reasoning Research and Development*, 164–178. 17, 23

Ontanón, S., Bonnette, K., Mahindrakar, P., Gómez-Martín, M., Long, K., Radhakrishnan, J., Shah, R. & Ram, A. (2009). Learning from human demonstrations for real-time case-based planning. 23

Orkin, J. (2003). Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom*, **2**, 217–228. 1, 2, 19, 20, 29, 34

Orkin, J. (2005). NPC Architecture Considerations for Real-Time Planning in Games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*. 29

Orkin, J. (2006). Three states and a plan: the ai of fear. In *Game Developers Conference*, vol. 2006, 4. 32

Paanakker, F. (2008). Risk-Adverse Pathfinding Using Influence Maps. *AI Game Programming Wisdom*, **4**. 145

Parker, M. & Bryant, B. (2009). Lamarckian neuroevolution for visual control in the quake ii environment. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, 2630–2637, IEEE. 17

Peters, C., Dobbyn, S., Mac Namee, B. & OSullivan, C. (2003). Smart Objects for Attentive NPCs. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, vol. 13, 14, Citeseer. 46

Pinto, H. (2008). Dialog managers. *AI Game Programming Wisdom*, **4**. 1

Pittman, D. (2008). Command hierarchies using goal-oriented action planning. *AI Game Programming Wisdom*, **4**, 383–391. 39, 48

Rabin, S. (2002). *AI game programming wisdom*. Charles River Media, Inc. 1

Reynolds, J. (2002). Tactical team ai using a command hierarchy. *AI Game Programming Wisdom*, **1**, 260–271. 39

Riedl, M. & Young, R. (2006). From linear story generation to branching story graphs. *Computer Graphics and Applications, IEEE*, **26**, 23–31. 1

Rossoff, S. (2009). *Adapting personal music based on game play*. Ph.D. thesis, University of Victoria. 1

Russell, S. & Norvig, P. (2009). *Artificial Intelligence: A Modern Approach*. Prentice Hall. 11, 13

Schrum, J., Karpov, I. & Miikkulainen, R. (2011). Ut? 2: Human-like behavior via neuroevolution of combat behavior and replay of human traces. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, 329–336, IEEE. 17

Shapiro, D. (1999). Controlling gaming npcs via reactive programs. In *AAAI Spring Symposium on Artificial Intelligence and Computer Games*, 73–76. 14

Sloan, C., Kelleher, J. & Mac Namee, B. (2011a). Feasibility study of utility-directed behaviour for computer game npcs. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, 5, ACM. 100

Sloan, C., Kelleher, J. & Namee, B. (2011b). Feeling the ambiance: using smart ambiance to increase contextual awareness in game npcs. In *Proceedings of the 6th International Conference on Foundations of Digital Games*, 298–300, ACM. 4, 139, 150

Sloan, C., Mac Namee, B. & Kelleher, J.D. (2011c). Utility-directed goal-oriented action planning: A utility-based control system for computer game npcs. In *Proceedings of the 22nd Irish Conference on Artificial Intelligence and Cognitive Science*. 111

Spalzzi, L. (2001). A survey on case-based planning. *Artificial Intelligence Review*, **16**, 3–36. 23

Straatman, R. (2009). The ai in killzone 2's bots: Architecture and htn planning. *AIGameDev.com*. 14, 22

Straatman, R., Beiji, A. & Van Der Sterren, W. (2006). Dynamic tactical position evaluation. *AI Game Programming Wisdom*, **3**, 389–404. 44

Sung, M., Gleicher, M. & Chenney, S. (2004). Scalable behaviors for crowd simulation. In *Computer Graphics Forum*, vol. 23, 519–528, Wiley Online Library. 145

Sweetser, P. (2004). Strategic decision-making with neural networks and influence maps. *AI Game Programming Wisdom 2*. 17

Thompson, T. & Levine, J. (2009). Realtime execution of automated plans using evolutionary robotics. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, 333–340, IEEE. 17

Togelius, J., Preuss, M. & Yannakakis, G. (2010). Towards multiobjective procedural map generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 3, ACM. 1

Tozour, P. (2004). Stack-based finite-state machines. *AI Game Programming Wisdom*, **2**, 303–306. 14

Traish, J. & Tulip, J. (2012). Towards adaptive online rts ai with neat. 17

Valve (2014). Team fortress 2 official wiki. 106

Van Basten, B. & Egges, A. (2009). Evaluating distance metrics for animation blending. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 199–206, ACM. 1

van der Sterren, W. (2001). Terrain reasoning for 3d action games. *Game Programming Gems*, **2**, 307–323. 1

Wallace, N. (2004). Hierarchical planning in dynamic worlds. *AI Game Programming Wisdom*, **2**, 229–236. 15

Weber, B. (2012). Integrating Learning in a Multi-Scale NPC. 20, 23

Willem, M. (1996). *Minimax theorems*. 24, Springer Science & Business Media. 12

Wintermute, S., Xu, J. & Laird, J. (2007). Sorts: A human-level approach to real-time strategy ai. *Ann Arbor*, **1001**, 48109–2121. 18

Woodcock, S. (1998). Game ai: The state of the industry. 100

Woodcock, S. (2000). Game ai: The state of the industry. 100, 108

Young, J. & Hawes, N. (2012). Evolutionary learning of goal priorities in a real-time strategy game. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 19, 20

Zubek, R. (2006). Introduction to Hidden Markov Models. *AI Game Programming Wisdom*, **3**. 17