Extending Behavior Trees with Classical Planning

Dat6, spring semester 2011
Group f11d624a
Master Thesis
Department of Computer Science
Aalborg University
9th of June 2011

AALBORG UNIVERSITY
STUDENTS STUDYREPORT

Title: Extending Behavior Trees with Classical Planning
Theme: Artificial Intelligence in Video Games
Project period: Dat6, spring semester 2011
Project type: Master Thesis
Project group: f11d624a
Group members:

Søren Larsen

Jonas Groth

Supervisor: Yifeng Zeng

Copies: 5
Number of pages: 72
Appendices: 4
Completion date: 9th of June, 2011

Abstract:

We investigate the area of behavior trees and planning in video game AI. The syntax and semantics of behavior trees and the concept of classical planning is described along with the theory behind the widely used search algorithm A*. A comparison of scripting and behavior trees is performed to identify the advantages of behavior trees. The advantages are used to validate that our extension of behavior trees does not violate them. We describe an extension of behavior trees with classical planning including a method for using states with the otherwise stateless behavior tree formalism. An implementation of the A* search algorithm for finding a sequence of behavior trees that will change the state of the world to some predefined goal state is also proposed. We test the approach with good results including the fact that the advantages of behavior trees are maintained as intended.

The contents of this report are openly available, but publication (with reference to the source) is only allowed with the agreement of the authors.
Additional material available on the attached CD, at `http://homes.student.aau.dk/jgroth07/thesis/` *or by request from the authors.*

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

FSM	Finite State Machine
HTN	Heirachical Task Network
BT	Behavior Tree
GOAP	Goal Oriented Action Planning
AI	Artificial Intelligence
FPS	First Person Shooter
NPC	Non-player Character
STRIPS	Stanford Research Institute Problem Solver
PDDL	Planning Domain Definition Language
GOAP	Goal Oriented Action Planning
DLL	Dynamic Link Library
RTS	Real-time Strategy
DAG	Directed Acyclic Graph
RPG	Role-playing Game
ME	Max Entropy
MDE	Max Discounted Entropy

1 INTRODUCTION

The area of Non-player Character (NPC) behavior in games is an area that is widely characterized by static Artificial Intelligence (AI). This is easily seen when looking at the wide application of techniques such as scripting and Finite State Machines (FSMs). These methods both provide the tools to create complex and even collaborative behavior. However they will still be static in the sense that the behavior always has a certain set of paths it can follow based on observations in the game environment. BTs also exhibit the same static behavior as the two previously mentioned methods; however it has the advantage of being more intuitive and easier to understand and implement.

Still BTs have yet to be used in the same degree as scripting and FSMs. In recent years however it has been used in very big and popular titles such as Crysis [16] and several games from the Halo series [8] [2] [1]. However there is little to no cohesion between the different BT implementations. The main problem is the fact that there is no single formalism for BTs and for this reason every game development company will call its implementation of a behavior Directed Acyclic Graph (DAG) structure for AIs a BT structure. Though, they all share the same abstract definition of the tiered tree structure defining execution flow of the complete behavior.

In academic AI several techniques exist that are able to adapt to the environment and learn from the successes and failures experienced. The most prominent areas of these are planning and learning. The problem with these techniques is that they typically are very computationally heavy and thus difficult to implement in the real-time and complex domain of game AI.

The advantage of using learning or planning is that it is possible to construct an AI that will adapt to the environment without having to specify every single possible scenario by simply planning what to do or learn what is optimal in the exact situation. This of course opens up a whole new set of complexities that needs to be addressed for the planning or learning system to function correctly. These complexities include defining a sensible Q-function for Q-learning or a proper cost and heuristic function for an A* search. Finding these can prove quite a challenge and as the state or action space grows it becomes even more challenging.

This is why the academic AI techniques often are not used in the complete form when creating game AI. An example of a game that uses a method from the traditional academic AI is the game F.E.A.R., [12], which uses a planning system in conjunction with a simple FSM to control the AI in the game, [14].

The aim of this project is to investigate what the advantages of using BTs in game AI are to identify the properties that need to be maintained by an extension. Further more we will investigate if extensions similar to those available for scripting and FSMs are applicable for BTs. All tests will be performed in a modern game engine to show the applicability of our approach in modern games.

1.1 PROBLEM STATEMENT

The problem statement for this thesis is derived from our previous work on BTs where two main problems were proposed for future work and research on the subject [10]. However, we have based our problem statement on only one of these problems:

> *Can Behavior Trees be extended with planning so that the creation of more diverse NPC behavior is possible?*

To further investigate the above given problems, we propose the following sub questions that will be addressed as well:

- What are the advantages of BTs compared to current game AI techniques?
- Is planning a feasible approach to game AI?
- How can planning be incorporated into BTs?
- And is it possible to maintain these advantages of BTs, when extended with planning?

2 THEORY

The purpose of this chapter is twofold. First, we want to introduce and discuss the theoretical aspects that serves as the basis for the work conducted in this thesis, and thus, establish a notational foundation for the following chapters. Secondly, we will also describe related work as inspiration for our proposed extension.

2.1 BEHAVIOR TREES

BTs are a relatively new up and coming approach to designing behaviors primarily aimed at NPCs. Although no single defined formalism exists several game development companies are adopting the idea of BTs. Companies often develop their own formalism that fits well into their development process thus achieving the goal of making it easier to create more complex behavior for their new games [16] [8] [2] [1].

BTs exhibit the same static properties of both scripting and FSMs in the sense that all behavior is predefined. The most significant advantage that BTs hold over scripting and FSMs is the easily understandable graphical representation and the ease of changing behavior at specific points in the tree. We will compare scripting and BTs in chapter 3.

BTs however lack some features compared to FSMs. The primary feature that FSMs have over BTs is reactive control. FSMs make transitions between states based on events in the game while BTs use the tree structure to determine what to execute next and how to handle failures. The event driven approach allows FSMs to react to events in the world in any state of the FSM if there is a transition that handles the event. BTs do not have this feature as you will need to check in each node if the event has occurred in the world and even if this is done BTs still lack the ability to specify a point in the tree to jump to, that can handle the event.

We will now describe the syntax and semantics of BTs by explaining each of the nodes available, ending with a description of the execution flow.

2.1.1 *Syntax & Semantics*

A BT is syntactically represented as a rooted tree structure, comprised of the nodes described below. Each node in a BT will have a final status after execution. Each status describes information that parent nodes can use to define what to execute next. The statuses are defined below.

- *Succeeded*
- *Failed*
- *Exception*

Once a node has executed it will have one of the above statuses. The status *Succeeded* is the status of a node that has successfully executed. The status *Failed* is used when a node has failed during execution. This does not define how it failed, just that it did. What defines a success or failure depends on the node and will be explained below. The last status, *Exception*, defines if something unexpected happened during execution. We will not be using this status but only make note that it is available in the formalism.

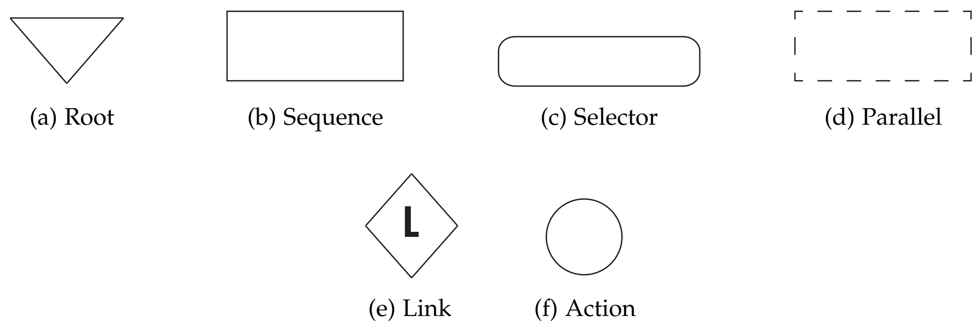


Figure 2.1: Syntax of the nodes of the BT formalism.

Root

The *Root* node, seen in Figure 2.1a, is of no functional interest. It is used as a graphical tool to easily identify the first node in a tree. Graphically the root node can only have one child and this child is the first node in the tree. During execution the node is ignored and its child is executed instead.

Sequence

The *Sequence* node, seen in Figure 2.1b, is used to execute multiple branches in sequence with the catch that if one of the branches fail then the node stops executing further and sets its status to failed. If all children successfully execute the sequence node succeeds. Sequence nodes can be augmented with a guarding condition that applies to all children. This requires the condition to be true in order for the sequence to continue to execute its children. The sequence can also have a condition attached to each child that is required to be true in order for that child to execute. This can help avoid attempting branches that cannot successfully complete. The condition of the node is written inside the node. However, as the condition can become large the node size should not grow indefinitely so only a subpart of the condition can be chosen to be shown in the node and the rest being defined in the code behind the node. The condition of children are attached to the link between the sequence and the children.

Selector

The *Selector*, seen in Figure 2.1c, node has a logically inverted execution compared to the sequence. The selector will attempt to execute its children from left to right until one of them succeed. If all children fail then the selector fails, but if one child succeeds the Selector succeeds As with the sequence node it is possible to augment the node itself or each of its children with conditions that guard execution.

Probability Selector

The *Probability Selector* node, seen in Figure 2.1c, has the same success criteria as the selector but the process of choosing a child is different. Each child branch of the probability selector is assigned a probability and during execution, a child branch is selected probabilistically among the the child branches based on the attached probabilities. The probabilities are written next to the link between the probability selector and its children. This gives BTs a way of generating random behavior with some control. The probability selector succeeds if one child succeeds and fails if all children fail just as with the selector.

Again as this node is a variant of the selector node it also pertains the same condition properties.

Parallel

The *Parallel* node, seen in Figure 2.1d, adds the notion of concurrent execution to BTs. The concurrency is however not actual concurrency but rather opens the possibility of executing multiple branches in the tree sequentially during a single execution of the tree. How this is actually done will be discussed in subsection 2.1.3. The success or failure of a parallel node can be hard to define but we have defined three different success types:

- *All Succeed*
- *Minimum Succeed*
- *Specific Succeed*

The *All Succeed* requires all child branches to succeed in order for the Parallel node to succeed, which resembles the sequence node. The *Minimum Succeed* requires a predefined minimum of child branches to succeed in order for the node to succeed. It disregards which of them succeed as long as the given minimum number of branches succeed. The *Specific Succeed* has a set of specific branches that must succeed. Branches that are not required to succeed are disregarded when determining if the node has succeeded. All nodes that are required to succeed must succeed or else the parallel node fails.

The parallel node also has the option of being augmented with a condition that guards execution and the option of augmenting each child with a condition to also guard their execution individually.

In this thesis we will not be using the parallel node but the syntax and semantics have been included to complete the BT formalism.

Action

The *Action* node, seen in Figure 2.1f, is used to perform actions in the game environment. The name of the action and the parameters are written beneath the action node. These actions can be any action available in the game environment for the given NPC. Actions succeed and fail based on the action they want to perform in the environment. An action can, as most nodes, also result in exception if for some reason the action does not end in success or failure. An example of this could be that an NPC is ordered to open a door. If the NPC opens the door it succeeds but if the door is locked it fails. However if perhaps the door was stuck, this could result in exception status for the action.

It is important to keep action nodes as concise as possible to avoid implementing entire behaviors in a single action node thus defeating the purpose of building the BT in the first place.

Link

The *Link* node, seen in Figure 2.1e, is used in a BT to reference and execute another BT in place of the node. The name of the tree executed is written beneath the node. The success and failure of a Link node is based on the referenced BT. This node adds the ability to make modular BTs which supports reuse of behavior.

2.1.2 *Changes to the Formalism*

As opposed to our previous work in [10] we have changed the notion of conditions in BTs. In [10] we used explicit conditions as leaf nodes in the trees but this proved to be problematic as it increased the number of sequence nodes in the tree. This situation occurred when a branch of the tree needed to be guarded by some condition. This would require the insertion of a sequence node with first a condition and then the guarded branch. As one can imagine this would grow the tree unnecessarily just for the sake of guarding branches.

We adapted our formalism based on other incarnations of BTs that include conditions on nodes and children. In [3] they describe conditions on sequences and selectors and in addition on the child branches of selectors. While in [16] all nodes are considered actions and all actions are guarded by conditions. As we have described above we have adapted these above approaches to work in our BT formalism. We removed the condition node completely and replaced it by allowing a condition to guard sequences, selectors and parallels and also their children. These conditions must always be true and if we at some point discover that an active branch's guarding condition becomes false, that branch must be halted and the tree structure should define what to do next. For instance a selector node that handles the failure of a branch by initiating another branch. Although we will not be using these conditions we have described them here as they will be important for future work described later.

We have also removed the *Decorator* node from the syntax as this proved to have a incomprehensible role in the tree. The role of being able to do almost anything made it hard to understand which role it had in the constructed behavior. That being said it may be interesting to have it return to the formalism in the form of a looping construct that can loop its child branch on some conditions. These conditions could be a set number of iterations, while some condition is true or false repeat etc. This would give the decorator a much more defined and understandable role. As we did not see the need for the looping construct in our thesis we have excluded the node.

2.1.3 *Execution Flow*

To illustrate the execution flow of a BT we have made an example of a complete BT, illustrated in Figure 2.2. This example shows a BT with a single sequence node as the first node. This sequence has two selectors as its children, a probability selector and a selector and is guarded by a condition that requires the controlled NPC to be alive. The selectors each have a link node and an action node as their children. The probability selector will probabilistically select with 50% probability between the two leaf nodes while the selector will attempt to execute the **GetFood** link node if the NPC has no food, else it will eat the food.

The execution of the tree will start by executing the sequence node. This will cause the probability selector to be executed. Which of the two branches will be executed is selected probabilistically, as described above. Once one of the branches has succeeded the sequence node will execute its second branch that will instruct the controlled NPC to get food, if it has none, and otherwise eat it. If both branches of one of the selector nodes fail, the selector node will fail causing the sequence and thus the entire tree to fail.

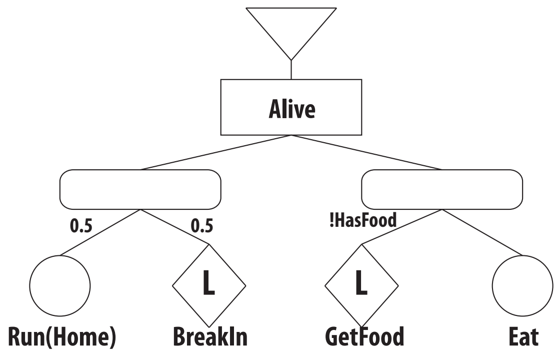


Figure 2.2: Example of a behavior tree using the nodes from Figure 2.1.

2.2 PLANNING

In relation to the general field of AI, we can categorize planning within the area of reasoning. That is, in planning we reason about how to act given our perception of the surroundings and by the expected outcome of the actions available. The purpose of this reasoning process is to acieve a sequence of actions that result in reaching some predefined objectives in the best way possible. The following elaboration on this reasoning process we call planning is heavily based on the work in [4, Chapter 1,2,4 & 11].

As with many other areas of AI, planning relies on a general model to describe the actual problem. More specifically, the model of state-transition systems is used. A state-transition system can be defined as a 4-tuple $\Sigma(S, A, E, \gamma)$ where $S = \{s_1, s_2, \dots\}$ is the set of states, $A = \{a_1, a_2, \dots\}$ the set of actions, $E\{e_1, e_2, \dots\}$ a set of events and γ is a state-transition function. However, we will assume some restrictions to this typical notion of transition systems:

Finite Σ: We require that the set of states in Σ is finite.

Fully observable Σ: The current state of Σ is fully observable. That is, we have complete knowledge about it.

Deterministic Σ: Transitions in Σ result in a deterministic change of the current state.

Static Σ: The system Σ has no internal dynamics. State transitions are always a result of an applied action by the planner, and not caused by events in the domain. Thus, the set E is no longer part of Σ and γ.

Restricted Goals: We only plan for a single goal state or a set of goal states. A solution is any sequence of state transitions that end at one of the goal states.

Sequential Plans: Planning solutions are a sequence of sequentially ordered actions.

Implicit Time: Actions applied to Σ implies instantaneous state transitions. Thus, actions are considered to have no duration and time is therefore implicit.

Offline Planning: Any changes in Σ while planning are disregarded.

Unless otherwise stated, all of these restrictions will be assumed in the state-transition systems used in the rest of this thesis.

Now we have a restricted state-transition system defined as a 3-tuple $\Gamma = (S, A, \gamma)$. The sets S and A are as previously defined, and the state-transition function is now defined as $\gamma : S \times A \to S^2$.So, given an action and a state, a and s, $\gamma(a, s)$ gives a state s' that is the result of taking a in s.

A well known representation of a state-transition system is a directed graph. We use nodes to represent states in S and edges between states s and s' are labeled with an action a if $\gamma(s,a) = s'$. Thus, edges represent state transition between states. With this graph representation, the actual problem of planning is reduced to finding a path in the graph between an initial state s_0 and one or more goal states in the set S_g of possible goal states we aim to reach. The resulting path is a sequence of action $\langle a_1, a_2 \ldots, a_k \rangle$ between states $(s_1, s_2 \ldots, s_k)$ such that $\gamma(\gamma(\ldots\gamma(\gamma(s_0, a1), a_2), \ldots, a_{k-1}), a_k) \in S_g$

2.2.1 *Classical Planning*

When planning for the restricted state-transition system we refer to it as classical planning. Though in other literature it may be referred to as Stanford Research Institute Problem Solver (STRIPS) planning, Planning Domain Definition Language (PDDL) planning etc. depending on the planning language used to represent the planning domain and problem. They all rely on the same restricted state-transition system however. In the following on classical planning we use a set-theoretic representation, where states of the world are sets of propositions. Thus a basic understanding of propositional logic is required from the reader.

Now, with a set-theoretic representation, the planning domain $\Sigma = (S, A, \gamma)$ is as defined in Definition 2.1.

Definition 2.1 – Planning Domain Σ
Assume a finite set of propositions denoted by $L = \{p_1, \ldots, p_n\}$. Then a planning domain on L is a state state-transition system $\Sigma = (S, A, \gamma)$ where:

- *$S \subseteq 2^L$ such that each state s can be any possible subset of L. That is, if proposition $p \in s$ then we say that p holds in the state represented by s. It follows that s denotes the set of propositions that make up the state. Then if $p \notin s$, by definition, p does not hold in s.*

- *The actions $a \in A$ denote a triple of subsets of L. We write this as $a = (precond(a), effects^-(a), effects^+(a))$. Here, $precond(a)$ is the set of propositions that are required to hold for a to be applicable. Then if $precond(a) \subseteq s$, a may be applied when in s. The set $precond(a)$ is also referred to as the preconditions of a. We call the other two sets, $effects^-(a)$ and $effects^+(a)$, the effects of a, and these are required to be disjoint.*

- *The set S of states need not be fully specified. We may infer new states from a state $s \in S$ by the actions applicable to s. That is because when an action a is applied to s we have $(s - effects^-(a)) \cup effects^+(a) \in S$.*

- *We define the state-transition function as $\gamma(s,a) = (s - effect^-(a)) \cup effect^+(a)$. This requires that $a \in A$ is applicable to $s \in S$, and if not, $\gamma(s,a)$ is undefined.*

The property that we me infer new states from the effects of actions, reliefs us from enumerating all states in S. This is useful, because even with a small set P of propositions, the number of possible states can become infeasable to enumerate.

We now define a planning problem $\mathcal{P}$ in Definition 2.2.1.

Definition 2.2 – Planning Problem $\mathcal{P}$
A set-theoretic planning problem is a triple $\mathcal{P} = (\Sigma, s_0, g)$, where:

- *Σ as defined above.*
- *s_0 is the initial state, and $s_0 \in S$.*
- *$g \subseteq L$ is a set of propositions that a state must satisfy to be considered a goal state. That is, $s \supseteq g$ for s to be a goal state. The set of goal states is denoted $S_g = \{s \in S | g \subseteq s\}$*

Finding the solution to $\mathcal{P}$ requires finding a path from s_0 to g in the directed graph we can construct from Σ. This can be done by using either forward or backward graph search algorithms. The A* search algorithm has proven very popular for this purpose.

2.3 A* SEARCH

The *A* search* approach is probably one the most widely used in graph search and pathing problems, in which we seek to obtain a path from a start node to one or more goal nodes. Because of it's applicability it has seen use within many fields. Computer gaming and AI is no exception, where A* is probably the most used algorithm for pathfinding allowing *NPC*s to navigate and move within a game environment. Recall also, that in section 2.2 we referred to A* as a purposeful approach to searching the state-space of a planning problem.

A* is an implementation of so called *best-first search*, where we use an *evaluation function*, $f(n)$, to guide the node, n, traversal in graph and tree searching problems [18, Section 3.5]. The evaluation functions adds problem-specific knowledge about the problem that is beyond that of the problem definition itself. This is compared to for instance depth-first search, where the search is performed solely on basis of the information available from the search problem itself. The evaluation function serves as a cost estimate of the search, such that at a node n we only expand the adjacent node n' with the lowest cost.

For A* the evaluation function is defined as follows.

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

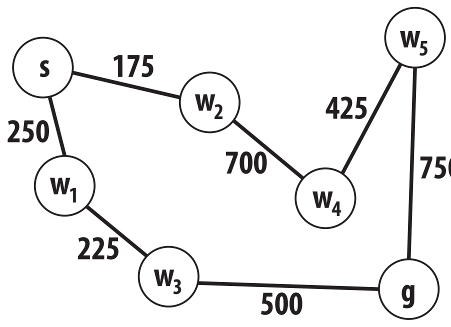
Here $g(n)$ denotes the accumulated cost when reaching the node n, and $h(n)$ is an estimate of the cost to reach the goal from n. The $h(n)$ function in $f(n)$ is also referred to as a *heuristic function*. It follows that for any node n, the cost of the cheapest solution through this node is estimated by $f(n)$. Traversing the nodes in a graph based on the lowest $f(n)$ value should then intuitively result in the cheapest path.

A* has several advantages [18, Section 3.5.2]. It is proved *complete*, meaning that if a solution exists, A* search will find it. Furthermore A* is *optimal* in the sense that the solution found will always be the cheapest. The optimality of A* however requires that the heuristic function is *admissible* and *consistent* for graph-search problems. An admissible heuristic never overestimates the cost to reach the goal from the current node. We call a heuristic consistent if for every node n and successor node n' as a result of taking the action a, the $h(n)$ value is no greater than $h(n')$ plus the cost of a. When the heuristic is consistent, $f(n)$ will be nondecreasing.

Now, consider a simple example like the one in Figure 2.3b to illustrate how A* is used for graph searching. The example illustrates a pathing problem where an NPC must find the shortest route from an initial location s to a goal location g with a set of waypoints $\{w_1, w_2, w_3, w_4, w_5\}$ that can be traversed. Edges between the waypoints indicate if the NPC

Node	$h(n)$
s	1000
w_1	750
w_2	800
w_3	500
w_4	75
w_5	750

(a) The $h(n)$ values for the nodes



(b) Graph representation

Figure 2.3: A* search example of a pathfinding problem.

may move from one waypoint to another. Edges are labeled with their cost, which in this case are relative distances. The heuristic used here is the straighline distance to g from the given location. That is, from e.g. w_3 the straightline distance to g is $h(w_3) = 500$. Note that this heuristic is also admissible because the straightline distance between two points can never be an overestimate. In Figure 2.3a all the $h(n)$ values for the nodes are listed.

In Figure 2.4 we see the search graphs that A* search stepwise generates when used to solve the pathing problem. A* first considers s, computes $f(s)$ and then expands the two

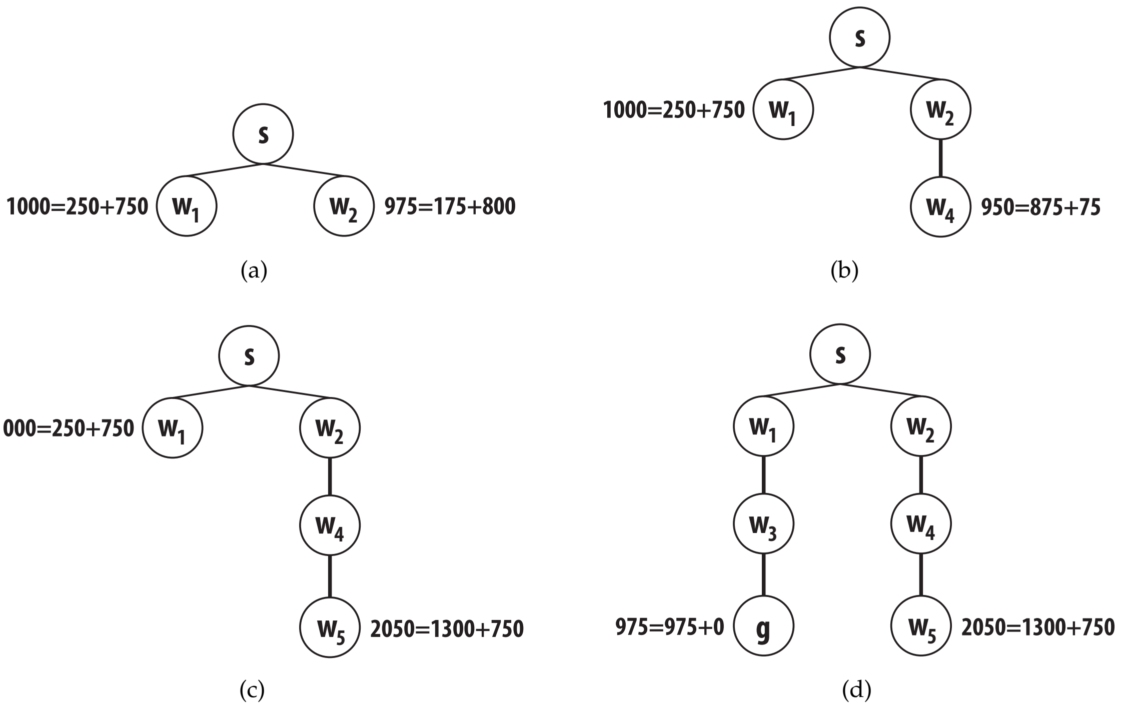


Figure 2.4: A* search example expansions based on the graph from Figure 2.3.

children w_1 and w_2. These are then considered for further expansion of the search graph by comparing $f(w_1)$ and $f(w_2)$, as is illustrated in Figure 2.4a. Because $f(w_2)$ has the lowest value of 975 it is chosen for expansion, making w_1 and w_4 the leaf nodes of the search graph. This is illustrated in Figure 2.4b. In Figure 2.4c we see that w_5 was chosen for expansion because $f(w_4) = 950$. Now w_1 becomes the leaf node with the lowest $f(n)$ value, and is thus expanded. The search graph continues expansion along this branch until the goal node g is reached. The final state of the search graph showing this, is illustrated in Figure 2.4d.

2.4 RELATED WORK

There are several interesting attempts at making it easier to make complex AI behavior. We have looked at related work to get inspiration for the approach we want to develop and will shortly describe this work in the following.

Query-Enabled Behavior Trees

In [3] an extension of BTs incorporating elements from case-based reasoning is described. The proposed idea is to allow for the BT to be constructed at runtime by querying a case-base to retrieve preconstructed BTs that match some criteria. One of the primary reasons for doing the dynamic construction is that often behaviors constructed late in the design phase are not utilized in behavior constructed earlier.

The main parts of the proposed approach are the case-base containing preconstructed BTs and the *Query* node. The node is implemented in a way that makes a minimal impact on the general structure of BTs. A Query node contains a query that at runtime is executed upon the case-base to find the most suitable behavior to run. The selection of an appropriate behavior is done based on a set of parameters called *Relevant descriptors*. The descriptors are variables from the game state that are relevant to the query. As it may not be possible to find an exact match in the case-base it is allowed to find behaviors that are similar based on the *similarity* parameter. This parameter defines how similar the retrieved behavior must be to fit the query.

Changes in the game state can occur when running a selected behavior, making another behavior more suitable. It is then possible to *Requery* the case-base. Which changes should trigger a requery can be defined in the requery parameter.

Goal Oriented Action Planning

The notion of Goal Oriented Action Planning (GOAP) combines FSMs and planning to achieve an AI capable of planning how and when to utilize the states available in a FSM [14] [12]. The planning used in GOAP is an extension of STRIPS planning with an A* implementation as the search algorithm.

The approach requires for each NPC, a set of goals that the NPC then has to prioritze between. The current prioritized goal is then used to plan actions for the NPC to excecute. At any point during planning or plan execution, another goal may be prioritized which requires the NPC to initiate re-planning.

The extensions of GOAP compared to STRIPS includes adding costs to actions which help guide the A* search and allowing procedural preconditions and effects of actions. Procedual effects is what connects planning with FSMs in GOAP. Recall from section 2.2 that STRIPS relies on a model where time is implicit. In GOAP however, effects of an action includes setting the state of the FSM, and thus some of the changes to the world state are not instantaneous.

GOAP was developed with the intent to be used in the First Person Shooter (FPS) game *F.E.A.R.* by *Monolith Productions*. It has since been used in other games and game genres, e.g., the Real-time Strategy (RTS) game *Empire Total War* or the FPS game *Deus Ex: Human Evolution* [15].

Heirachical Task Networks and Scripting

[9] describes an approach to generating scripts for NPCs, tested in the *Bethesda* game *The Elder Scrolls IV: Oblivion*. The approach relies on offline Heirachical Task Network (HTN) planning to avoid expensive computations on runtime.

The architecture of the system relies on the game developers to design an HTN that encodes the game world and implement a set of components called *AI packages* for each action of the HTN. A list of planning problems and the corresponding world state for each planning problem is also required. The planning system then plans each planning problem by giving the planner the HTN, the planning problem and the initial world state. The planner then outputs a plan for all NPCs. It is not possible to plan for each NPC separately, as it can be required that a set of NPCs must collaborate or interact on a task.

After the plans are generated a script generator converts the plans into scripts in the The Elder Scrolls IV: Oblivion scripting language. The generated scripts call the AI packages defined above in the planned order. As these packages do not have preconditions attached, the generated script maintains state information on where in the plan it currently is and when it is appropriate to call the different packages by using a conditional structure and the above mentioned state information.

Though offline planning has to deal with uncertainty at runtime by doing a lot of contingency planning to account for different types of outcome it has the advantage of being executed prior to runtime. The planning can then be a lot more complex comparing to online planning which has to be optimized to avoid using excessive resources.

Heirachical Task Networks in Unreal Tournament

The authors of [17, Sec. 5.4] and [6] have applied HTN planning to *Epic Games* game *Unreal Tournament* (UT). Using a combination of the *Gamebot* API software that allows remote execution of bots on UT servers and an adapted version of the *SHOP* HTN planning system they are able to formulate strategic plans for a team of bots and communicate these plans back to the UT server.

The results show that the approach gives consistently better performance versus the standard AI and also better versus a modified version of the standard AI that has been improved in both navigation and domination tactics.

As opposed to the above approach of using offline HTN planning on NPCs, this approach relies on online planning which allows adaption to changes in the game environment instead of relying on having to make contingency steps in the generated plan.

3 COMPARISON

In this section we will compare scripting and BTs to determine the advantages of creating AI in computer games so that we can maintain these advantages in our extension. One important note is that in normal game environments there is a small overhead in the creation of the nodes to be used in the BTs. However if the choice is made to use BTs early in the development process this overhead can be disregarded as the construction of nodes is easy when having access to the game's API.

3.1 SCENARIO

The scenario we have created for the purpose of this thesis, resembles a tavern like setting, greatly inspired by role-playing games such as *Never Winter Nights*, *The Elder Scrolls* etc. Typically taverns in Role-playing Game (RPG) games are social gathering points, where multiple NPCs socialize and interact with each other and the environment. Environmental interacting such as purchasing and consuming drinks, leaving and entering rooms and interacting with furniture etc. Thus, this setting gives a great foundation for different behaviors that both have to react to and interact directly with the environment. Figure 3.1 depicts the scenario, as it looks in Unity. In the following description of the scenario we

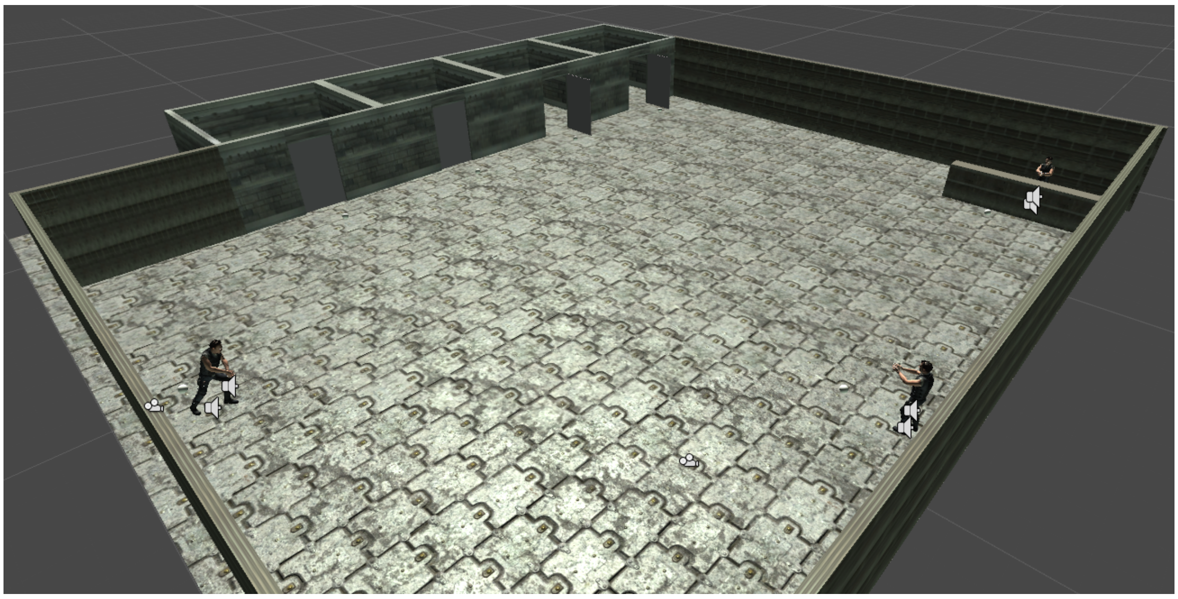


Figure 3.1: The scenario viewed in the Unity 3D editor.

will use a much simpler depiction however. The reason for this is twofold. Firstly, labeling and referencing different objects in the scenario will become much clearer. Secondly, the behaviors and their figurative depiction used later in this chapter will be easier to interpret. Figure 3.2 illustrates the simple depiction of the scenario viewed from above. The layout of the scenario is comprised of one large room, with four adjacent smaller rooms **R1** through **R4**. The smaller rooms are each seperated from the large room by doors. The start location of our NPC is indicated by a star symbol, while the two uncontrolled NPCs, the patron and the barkeeper, have their locations indicated by labeled circles. In the following when refering to locations in the scenario, the labels will be used for the noncontrolled NPCs, so

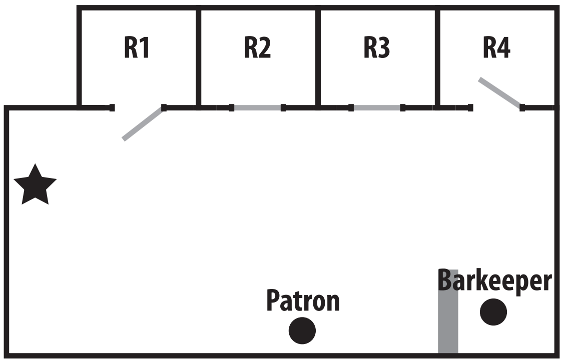


Figure 3.2: The scenario from Figure 3.1 viewed from above.

that **Patron** will refer to the patron's location and **Barkeeper** the barkeeper's. However, the shorter **Bar** will be used interchangeably with **Barkeeper**.

Likewise, **R1** through **R4** will be used to refer to the rooms and a general location inside them. Additionally, room locations will be postfixed to indicate a location outside or inside the room. For instance, **R1I** is the location just inside **R1** and **R1O** is the location just outside **R1**.

The general properties of the scenario adhere to the restrictions previously stated in chapter 2 for the model that classical planning relies on. The scenario is fully observable to the NPC with some subtle restrictions however. For instance, the four doors all have a property called *locked* which only return the actual boolean value indicating whether the door is locked or not if the NPC has attempted to open the door. If this has not been attempted, *locked* will return false. The scenario is also completely static, in the sense that neither the barkeeper, the patron or any other object in the scenario can make changes to the scenario.

3.1.1 *Unity* 3D

Unity 3D (Unity), is a game development software tool, complete with a game engine, graphical environment editor, scripting support etc. It is made available in a professional and a standard version, the latter being free to use.

Unity is very popular among indie game developers. Especially a lot of games developed for handheld platforms, such as iPhone and Android, utilize Unity. More ambitious cross-platform games such as *Interstellar Marines* by danish developer *Zero Point Software*, also utilizes Unity [20]. Thus, we are able to show the integration of our BT framework in a current and commercially used game engine.

Because of the graphical environment editor, we are able to implement the scenario without spending too much time learning the workings of a game engine.

Unity utilizes Mono for scripting support. Thus, according to Mono's compatibility with .Net [11], scripting in Unity is almost fully .Net 3.5 compliant. This is one of the main features as to why Unity was chosen. Given that our BT framework is written in C#, we are able to integrate it directly into Unity. Furthermore, using MonoDevelop as the script editor, it is possible to also directly debug Unity scripts with MonoDevelop, which has proven to be of great help during the integration of our BT framework.

Unity has also previously been used in acadamic settings. Most notably the engine was used in a master thesis for implementation and testing of BTs with a framework written in C# [7].

Our implementation of BT is connected to Unity by including a Dynamic Link Library (DLL) file containing the entire framework. The next step is then to create the scenario specific BT nodes which act as the interface between the framework and the scenario. In Figure 3.3 the connection between Unity and the BT framework is illustrated.

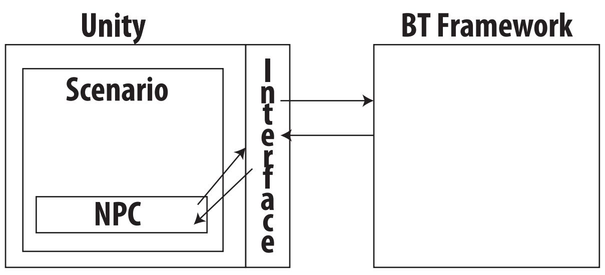


Figure 3.3: Overview of Unity and the BT framework.

3.2 COMPARISON OF BEHAVIOR TREES

To compare BTs with scripting we will describe a basic behavior that will be implemented both as a script and as a BT and then evaluate the two approaches. We will then describe two extensions to this basic behavior. The first extension will be comparing the approaches on modularity while the second extension will compare the approaches on both selective behavior and modularity. The comparisons are made in the scenario as described in section 3.1.

3.2.1 *Basic Behavior*

This is the basic behavior that we will use for the first comparison and as the base for the two extensions.

Description of behavior

This basic behavior will make the NPC run from **Start** to **Bar**. Afterwards it will run to the **Patron**. After the **Patron** it will go to **R3O** and open the door. It will run to the back wall and then back out of the room and back to **Start** location. The behavior then restarts performing the same sequence of actions again. In Figure 3.4 this behavior is illustrated.

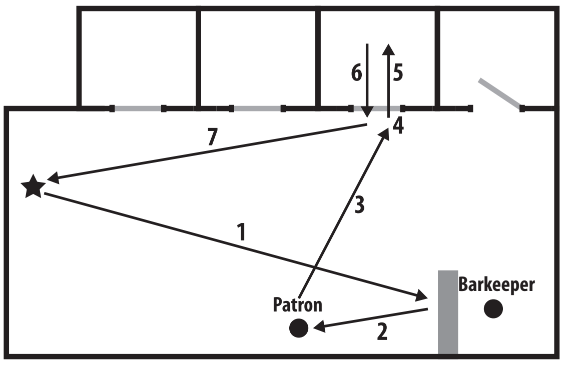


Figure 3.4: The basic behavior illustrated in the scenario.

Script

In Listing 3.1 the basic behavior is implemented as a script. Here *NPC* refers to the NPC executing the behavior. A few methods implemented on the *NPC* object are used in the script. Their meaning is self-explanatory. An if clause construct is used to verify the NPC's current location and how to act onwards.

```
if (NPC.IsAtPosition(Start) && NPC.State == Idle){
    NPC.Run(Bar);
}
else if (NPC.IsAtPosition(Bar) && NPC.State == Idle){
    NPC.Run(Patron);
}
else if (NPC.IsAtPosition(Patron) && NPC.State == Idle){
    NPC.Run(R30);
}
else if (NPC.IsAtPosition(R30) && NPC.State == Idle){
    NPC.OpenDoor();
    NPC.Run(R3);
}
else if (NPC.IsAtPosition(R3) && NPC.State == Idle){
    NPC.Run(Start);
}
```

Listing 3.1: The basic behavior from Figure 3.4 as a script.

Behavior Tree

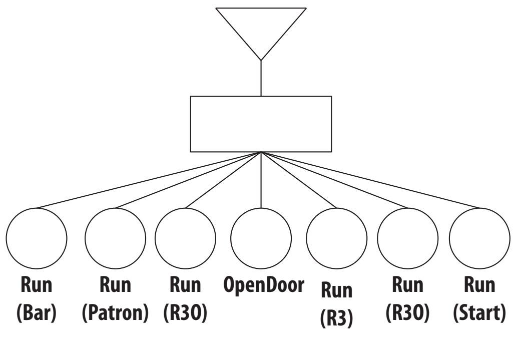


Figure 3.5: Behavior Tree for the basic behavior from Figure 3.4.

Figure 3.5 shows the BT that implement the basic behavior defined above. Each *Run* node is given a parameter which is a location in the scene. Looking at the figure it is easy to see that this BT will perform the intended behavior which is also proven by running the scenario with an NPC controlled by the BT.

Evaluation

From a construction point of view it was fast and easy to build the tree as it was a matter of using the *Run* node and give it as parameter the desired location. The *OpenDoor* relies on the NPC being at the location of a door to work. This is guaranteed by the success of the node *Run(R3O)* which is the location outside the door of room **R3**.

Appart from the apparant difference in visual representation, there are some key differences between the BT and the script. In scripting a method to check the current position is used to keep track of the NPCs progress with the behavior. This introduces a great deal of redundancy. The location checks are performed implicitly as part of the success of the action nodes in the BT. Furthermore, locations are not used to keep track of the NPCs progress with the behvaior, and the semantics of a sequence node is instead utilized for this.

There is no noticeable problems with either the script or BT implementation and they perform the intended behavior.

3.2.2 *Extension One*

This behavior is extended from the basic behavior and will be used for the second comparison where the focus is modularity.

Description of behavior

The behavior extends upon the basic behavior by injecting a sequence of actions right before the basic behavior returns to **Start**. The sequence of actions will make the NPC run to room **R3**'s door and open it. If it is locked it will unlock it and open the door. It will run into the room and close the door and then run to **R3**. After this it will run back to **R3I** and open the door and run to **R3O** and close the door again. It will then resume the basic behavior and run to **Start**. Once at **Start**, the behavior will restart performing the actions again. In Figure 3.6 this behavior is illustrated.

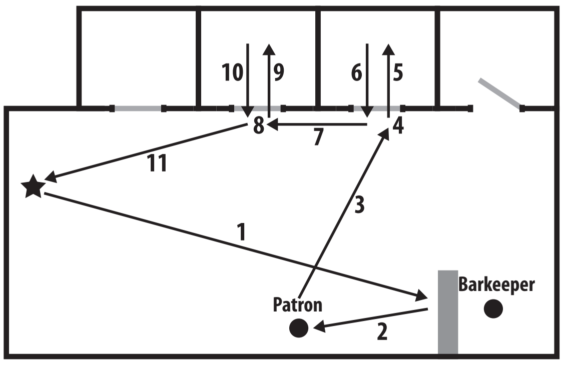


Figure 3.6: The first extension to the basic behavior.

Script

The script seen in Listing 3.2 only includes what is an extension to the basic behavior. Unlike the compact representation of BTs where we are able to illustrate the complete behavior, including the entire script in the listing only further complicates the interpretation for the reader. Once again the invoked methods on the *NPC* object are self-explanatory.

```
else if (NPC.IsAtPosition(R3) && NPC.State == Idle){
    NPC.Run(R10);
}
else if (NPC.IsAtPosition(R10) && NPC.State == Idle ){
    if (!NPC.GetClosestDoor().open){
        if (!NPC.GetClosestDoor().locked){
            NPC.OpenDoor();
        }
        if (!NPC.GetClosestDoor().open) {
            NPC.UnlockDoor();
            NPC.OpenDoor();
        }
        NPC.Run(R1I);
    } else {
        NPC.CloseDoor();
        NPC.Run(Start);
    }
}
else if (NPC.IsAtPosition(R1I) && NPC.State == Idle){
    if (NPC.GetClosestDoor().open){
        NPC.CloseDoor();
        NPC.Run(R1);
    } else {
        NPC.OpenDoor();
        NPC.Run(R10);
    }
}
else if (NPC.IsAtPosition(R1) && NPC.State == Idle){
    NPC.Run(R1I);
}
```

Listing 3.2: The first extension from Figure 3.6 as a script.

Behavior Tree

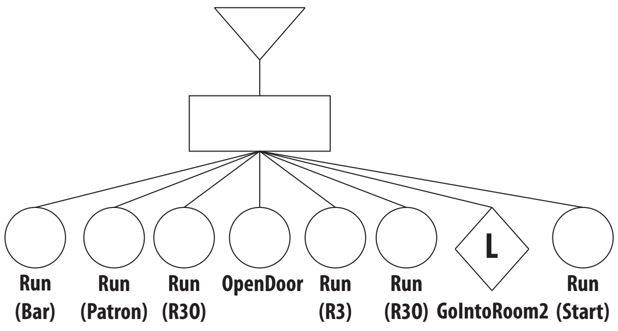


Figure 3.7: Behavior Tree for the first extension from Figure 3.6.

Figure 3.7 shows the extension of the basic behavior. The change is the inserted link node which links to the BT shown i Figure 3.8. This clearly shows the adaptability of BTs. The change in the behavior to go into room **R3** is as easy as creating a BT that does this and then inserting a link node that links to that tree. No other changes were needed to adapt the BT, for the basic behavior, to the extended behavior.

In Figure 3.8 it is worth noticing the subtle detail that the BT will first attempt to open the door. If this fails it will then try to unlock the door and then open it. This is achieved by taking advantage of the semantics of both selector and sequence nodes.

Evaluation

This comparison clearly indicates the modularity of BTs compared to scripting. A link node is utilized to include a complete behavior describing the actual extension. This could however also have been wrapped inside a method in scripting, and a call to this method would be analogous to the link node. However the inclusion of the sub-behavior appears more elegant in the BT, where a single node was added to the basic behavior in order to achieve the expected behavior. In the script it was necessary to alter the last clause of the if control structure and move the line sending the NPC back to **Start** and insert it into the extended script. This realocation of code in the script opens up to the possibility of making mistakes when altering the basic behavior.

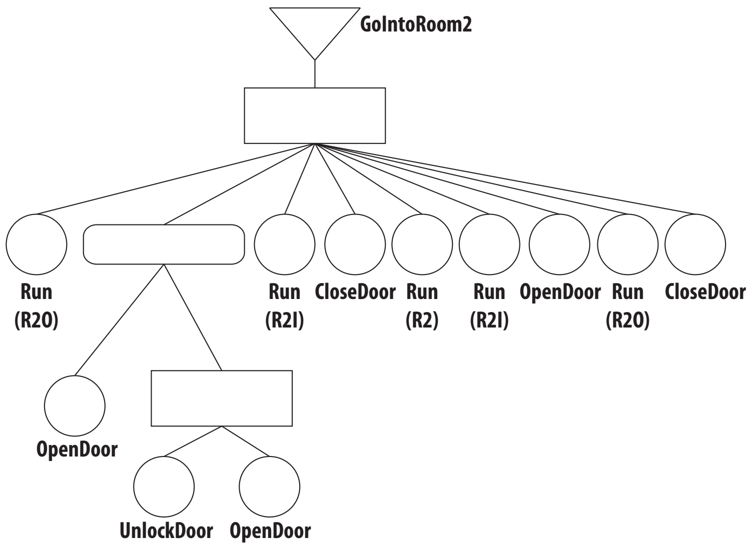


Figure 3.8: Behavior Tree: GoIntoRoom2 linked in Figure 3.7.

The matter of opening the possibly locked door to room **R2** is in the BT implemented by using a selector node that first attempts to open the door and if this fails, it will try the other child that first attempts to unlock and then open the door.

3.2.3 *Extension Two*

This is the last extension to the behavior that extends upon the first extension by adding selective behavior.

Description of behavior

This behavior makes the NPC run, as in the two previous, to **Bar** and afterwards to **Patron** and into room **R3**. After running out of room **R3** the behavior must now select whether to go into room **R1** or **R2**. There is a probability of 0.5 for each possibility. Each choice makes the NPC walk up to the door of the room and open the door. If the door is locked it unlocks it and opens it. Afterwards it walks into the room and closes the door. It walks to the back wall and then back to the door and opens it. Once the door is open it will walk out of the room and close the door and return to **Start**. After completing the behavior it restarts. In Figure 3.9 this behavior is illustrated.

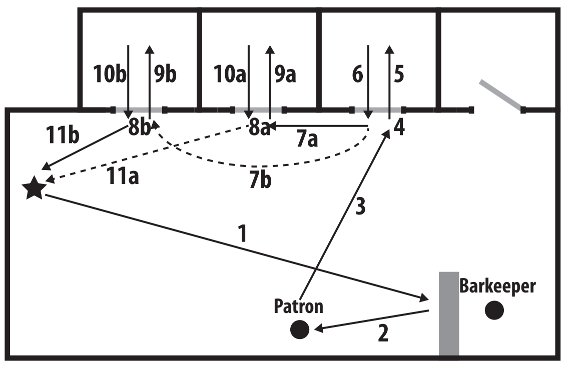


Figure 3.9: The second extension to the basic behavior.

Script

Again, Listing 3.3 only includes what is extended compared to the basic behavior and the first extension for the same reasons as discussed previously. The selective behavior is achieved in the script by using a randomized double in the interval $[0.0, 1.0]$ and use this double to select between the two sub-behaviors.

```
else if (NPC.IsAtPosition(R3) && NPC.State == State.Idle){
    double tmp = rand.NextDouble();
    if (tmp < 0.5){
        NPC.Run(R10);
    } else {
        NPC.Run(R20);
    }
}
else if (NPC.IsAtPosition(R20) && NPC.State == State.Idle){
    if (!NPC.GetClosestDoor().open){
        if (!NPC.GetClosestDoor().locked){
            NPC.OpenDoor();
        }
        if (!NPC.GetClosestDoor().open) {
            NPC.UnlockDoor();
            NPC.OpenDoor();
        }
        NPC.Run(R2I);
    } else {
        NPC.CloseDoor();
        NPC.Run(Start);
    }
}
else if (NPC.IsAtPosition(R2I) && NPC.State == State.Idle){
    if (NPC.GetClosestDoor().open){
        NPC.CloseDoor();
        NPC.Run(R2);
    } else {
        NPC.OpenDoor();
        NPC.Run(R20);
    }
}
else if (NPC.IsAtPosition(R2) && NPC.State == State.Idle){
    NPC.Run(R2I);
}
```

Listing 3.3: The second extension from Figure 3.9 as a script.

Behavior Tree

In Figure 3.10 we have made an extension of the previous BT. We have moved the link node down as a child of a newly inserted probability selector node. The other child of the probability selector is a link node that links to a BT that is identical to the BT *GoIntoRoom2* except it will move the NPC into **R1**. The probability given in the above description of the behavior is implemented on the probability selector by giving each child a probability of 0.5.

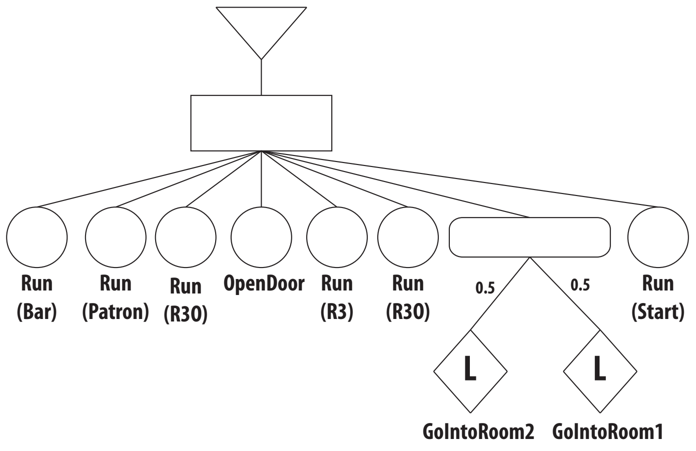


Figure 3.10: Behavior Tree for the second extension from Figure 3.9.

Evaluation

Implementing selective behavior is easy in BTs because of the selector node. As it is apparent in the scripting implementation, it requires additional variables and "housekeeping" of these to the achieve the expected behavior. However, if the chosen child of the probability selector node were to fail, the probability selector would execute the second child, which is not the intended behavior for this test. Scripting achieves this behavior correctly. Adding a condition on the probability selector that would only allow it to execute a single child, will alleviate this behavioral difference.

Were the intended behavior to change, from selective to simply going in to the rooms in sequence, this will only require the probability selector node to be exchanged with a sequence node. In scripting this change will however require a substantial amount of restructuring and recoding.

A feature of BTs that were not used for this behavior, is the possibility of parameterized BTs. It would have been possible to give the room as a parameter for the BT and then implementing the nodes in such a way that they, given a room parameter, know which location they reference. This would allow for the two link nodes to reference the same tree but with different parameters thus further improving the modularity of the BT.

3.3 EFFICIENCY

To compare the efficiency of BTs and scripts we measured the difference in *frames per second* (FPS) between the two. We chose FPS as a measurement as it is the most reliable performance measure retrievable in Unity and if BTs cause a slowdown in the game engine, due to taking longer to execute. It would be clearly visible as a drop in FPS, because if frames take longer to execute, the fewer frames can be completed each second. During execution of the comparisons we set each script and BT to execute its behavior 10 times.

We calculated an average FPS during these 10 iterations and in Table 3.1 it can be seen that both BTs and scripts hold an FPS of around 64. The BTs are marginally slower in the order of $\frac{1}{10}$ of a percent. This is most likely caused by the recursive execution of nodes in the BTs as the actions themselves are the same functions being called in both BTs and scripts.

We did not measure the memory consumption of each approach as it is clear, from the implementation of the BT framework, that BTs will use more memory. Some of the reasons for this is the use of classes which gives some overhead in the form of object headers etc. and the memory used to store the BT XML file. However it is commenly known that modern computers come with very large amounts of memory and most modern games will not use the entire available memory thus leaving space for the BT framework.

Table 3.1: FPS comparison of scripting and Behavior Trees

Behavior	Scripting (FPS)	Behavior Trees (FPS)	Delta (%)
Behavior 1	64.016	64.004	−0.02%
Extension 1	64.046	64.017	−0.05%
Extension 2	64.048	64.015	−0.05%

3.4 SUMMARY

With the scenario defined and a game engine chosen, the comparison of BTs and scripting has been performed as expected, and thus we have come to several conclusions about the differences between scripting and BTs. The comparison has shown both strengths and weaknesses between the two approaches.

The basic behavior and the two proposed extension, shows the adaptability and reusability of BTs. It was easy to change and extend the behaviors without having to make major changes in the structure of the tree. This was not the case for scripting as there was a need for making some changes in the original behavior to support the extensions.

When looking at the graphical representation of BTs versus the textual representation of the scripts it is also easy to see that non-programmers may find it harder to interpret scripted AIs. The graphical representation of BTs gives a fast overview of the intended behavior through the easily understandable construction blocks. The only problem that can be present here is if the game developers construct action nodes that perform actions in the game environment with effects that are not easily comprehensible.

We saw however that in extension two the intended behavior required that the NPC had to make a choice of going into one room or the other. This behavior was not correctly attained in the BT as this would require a custom condition on the probability selector node which would override the normal execution flow of the node.

4 EXTENDING BEHAVIOR TREES

We propose a planning architecture that can provide game developers with a system for defining goals for NPCs and components that solve these goals. Before explaining goals, components and the planning system we first need to look at three basic components: preconditions, effects and the world state.

4.1 PRELIMINARIES

Before diving into the actual planning we will first need to define the basics of the planning architecture.

4.1.1 *Preconditions, Effects and the World State*

A precondition is a boolean expression that can be evaluated to true or false. Preconditions can be satisfied implicitly by the *World State* or by the expected effect of a component when we use them in planning. Upon execution the preconditions must be satisfied by the world state. If a precondition, that during planning was satisfied by an effect of a component, is not satisfied upon execution then a discrepancy has occurred. This means that the component did not have the expected effect and thus we must trigger a replanning of the current goal to accommodate.

An effect is an expected change in the world state after the execution of the component having said effect. The satisfaction of preconditions by effects and the world state can be represented by the following function where P is the set of preconditions and E is the set of effects:

$$satisfied(P,E) = \begin{cases} true & \text{if } P \cap E = P \\ false & \text{if } P \cap E \neq P \end{cases} \tag{4.1}$$

In Equation 4.1, E can be replaced by the world state. The world state includes all the preconditions from all components and goals that at the current time are satisfied. We denote the world state by s_w, and this is how we introduce states to the otherwise stateless BTs.

Note that our definition of E does not differentiate between negative and positive effects, like it is seen in classical planning with $effect^-(a)$ and $effect^+(a)$ of an action a. Now, if the current state holds the precondition $\neg$**DoorOpen**, and we use an component with **DoorOpen** $\in E$ we set the precondition's value to true in the resulting state.

4.1.2 *Goal*

To explain what the planning system, described below, revolves around, we first define what a goal in the planning architecture is. The planning system uses goals to define what to plan for. The planning system is invoked by giving a goal to achieve. It will first find a decomposition of the goal that is relevant and then plan the sequence of components that will take the game environment from the initial state, s_{w0} ,to a state, s_{wi}, where the goal is achieved. The satisfaction of a goal is defined by its set of preconditions.

Definition 4.1 – Goal
A goal G consists of a set, P, of n preconditions and a set, D, of tuples representing the decomposition of the goal:

- $G = (P, D)$
- $P = \{p_1, p_2, \ldots, p_n\}$
- $D = \{\langle G_1, G_2, \ldots G_m \rangle, \ldots, \langle G_1, G_2, \ldots G_k \rangle\}$

A goal G is satisfied if and only if:

- $p_1 \wedge p_2 \wedge \cdots \wedge p_n = \textbf{TRUE}$

A goal is as defined in Definition 4.1. The set of decompositions, D, can be used to represent a tiered tree of goals and their decomposition. The tiering is possible due to one very important definition. A goal cannot use another goal from its own tier as a subgoal. This will make the goal of a higher tier than that of the goal it includes. It is allowed for all higher tier goals to access lower tier goals. That is, it is allowed to use goals that are not only in the tier immediately below the goal but also from lower tiers. The intuition behind allowing goals to be decomposed into multiple goals is simple. It allows the game developer to guide the planning through some desired goals, instead of just allowing the planner to plan for one single goal.

This differs somewhat from the definition of a goal in classical planning. Recall that a planning problem $\mathcal{P}$ is defined as a triple (Σ, s_0, g). That is, in classical planning we plan for a single goal g. Consider now our definition of a goal, where it is decomposed into e.g. $\langle G_1, G_2 \rangle$. This is a sequential ordering of the goals G_1 and G_2. In classical planning we could solve this by the two planning problems $\mathcal{P}_\infty = (\Sigma, s_0, G_1)$ and $\mathcal{P}_\infty = (\Sigma, G_1, G_2)$.

Effectively, our goal decompositions relaxes on the restriction in classical planning on restricted goals. Figure 4.1 shows an example of the decomposition graph.

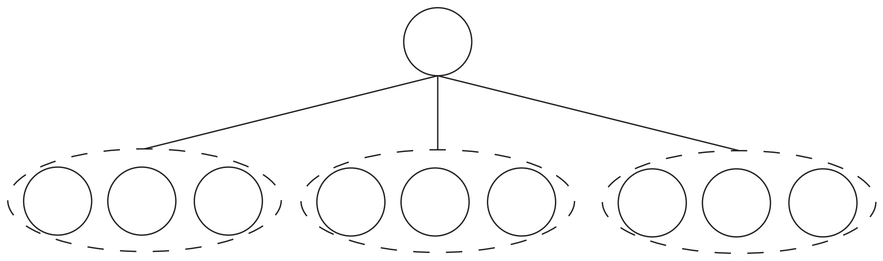


Figure 4.1: Simple Goal with three decompositions, each containing three goals

4.1.3 *Components*

The architecture has a library of components that can be used for solving goals, and they relate closely to actions in classical planning but with some modifications. This will come apparent in the following. The components are BTs that have been constructed to have some effect in the game environment and this may also lead to a set of preconditions the must be satisfied before that component can be used. The effects of one component can satisfy the preconditions of another component thus allowing the components to follow each other in sequence. The effects of a component can also satisfy a goal, as defined above, and this is used to find the sequence of components that may solve a selected goal or sequence of goals. It is not required that it must satisfy all preconditions of a goal or component but at least one, as opposed to what was earlier described. Any remaining preconditions will be moved to the node in the search graph that was selected as we eventually must satisfy them.

Definition 4.2 – Component
A component is a four-tuple (P, B, C, E) *where:*

- $P = \{p_1, p_2, \ldots, p_n\}$
- B *: is a behavior tree*
- $E = \{e_1, e_2, \ldots, e_m\}$
- $C = |E|$ *: is the integer cost of using the component*

Definition 4.2 shows that a component consists of a set of precondition that may be empty and must be satisfied for the component to work, a BT which is the logic of the component. Also part of the definition is an integer cost of using the component which is used to guide the planning and a set of effects that define which preconditions in the world state are expected to be satisfied after the component has finished executing.

The selected components that solve a given goal make up a branch in a virtual BT that we describe later. The branch consists a sequence node with link nodes referencing all components in the selected sequence of components.

4.2 PLANNING

The planning system runs in two phases: the goal decomposition phase and the component planning phase. Each phase has a distinct effect in the planning system and can be tweaked to provide alternate planning outputs.

4.2.1 *Goal Decomposition*

The first phase, the goal decomposition phase, has the purpose of making the the goal or sequence of goals diverse to make the AI using the architecture act more diverse. This phase relies heavily on *Entropy*. Entropy describes the unpredictability of a variable and can be used as a measurement to increase the diversity of an AI. One of the more well known types of entropy is the *Shannon Entropy* as described in [19]. Shannon entropy revolves around the function $H(X)$ where X is the discrete random variable with n possible outcomes. The function can be represented as:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b(p(x_i)) \tag{4.2}$$

Here $p(x_i)$ is the probability of outcome x_i for variable X and $b = n$. This function will return its highest value (1) if for all x_i, $p(x_i) = \frac{1}{n}$ and its lowest if all $x_i = 0$ but one $x_i = 1$. Intuitively this makes sense because if the probability distribution of the outcomes is uniform then there is maximum uncertainty as to which outcome is expected thus giving maximum entropy. In the other case there is no doubt that the variable will always have the outcome with probability 1 thus there is no uncertainty giving minimum entropy.

In the planning architecture we use a simplified calculation that gives a similar result. We select goals solely on the amount of times the goal has been picked as part of the goal sequence. All goals start with a selected value of 1 to allow for the discounted entropy described below.

The planning system allows for two different approaches to selecting a goal decomposition:

- Max Entropy (ME)
- Max Discounted Entropy (MDE)

We will now discuss these two approaches to goal decomposition and their proposed algorithms.

Max Entropy

Algorithm 1 shows the ME decomposition algorithm. The goal that we are determining whether to decompose or not is used as initial values for the loop. We assign i with the goals number of times selected. This variable is used to determine if a decomposition has a lower number of times selected and thus is more ideal to use. *sel* is the set containing the selections that currently have the lowest number of times selected. Initially it is empty. We run through all possible decompositions of the initial goal and if the number of times selected is lower than the current lowest the set is overwritten and i is set to the lowest. If the number of times selected is the same as the current lowest the decomposition is added to the set. If number of times selected is higher than the current lowest the decomposition is ignored.

Algorithm 1 Max Entropy decomposition selection

```
1:  $i \leftarrow g.TimesSelected$
2:  $sel \leftarrow \{\epsilon\}$
3:  for $k = 0 \rightarrow g.Decompositions.Count$ do
4:      if $g.Decompositions[k].TimesSelected < i$ then
5:          $sel \leftarrow \{k\}$
6:          $i \leftarrow g.Decompositions[k].TimesSelected$
7:      else if $g.Decompositions[k].TimesSelected == i$ then
8:          $sel \leftarrow sel \cup \{k\}$
9:      end if
10: end for
11: if $|sel| == 0$ then
12:     return $g$
13: else
14:     if $|sel| == 1$ then
15:         return $sel[0]$
16:     else
17:         return $RandomSelect(sel)$
18:     end if
19: end if
```

Once all decompositions have been checked we check whether *sel* has a length of 1 and return the single element in the set. If the length is 0 we return the initial goal as no decompositions had a lower number of times selected. If there are more than 1 elements in the set we randomly select on of the elements and return it.

It is easy to see that this will always go towards having a uniform distribution of the times selected as we always look for the possible selections that have been selected the least. If for any $sel_i, sel_j \in g \cup g.Decompositions$, that is for any two goals or decompositions in the set of the goal and its decompositions, it holds that $sel_i.TimesSelected = sel_j.TimesSeleted$ then the probability distribution of the possible outcomes is uniform. This is because probability can be calculated as $\frac{m}{n}$ where m is the number of times the outcome has occurred and n is the total number of outcomes. As an example if three different outcomes are possible and each has occurred 3 times then $\frac{3}{9} = \frac{1}{3}$ for each outcome giving a uniform distribution.

Max Discounted Entropy

MDE uses a slightly different method to calculate the effective number of times selected than ME. The algorithm is shown in Algorithm 2. With MDE we recognize that some goals or decomposition may be more desirable than others. This desirability can of course change over time or as a result of a change in the game environment.

Algorithm 2 Max Discounted Entropy decomposition selection

```
1:  $i \leftarrow g.TimesSelected * g.Discount$
2:  $sel \leftarrow \{-1\}$
3:  for $k = 0 \rightarrow g.Decompositions.Count$ do
4:    if $g.Decompositions[k].TimesSelected * g.Decompositions[k].Discount < i$ then
5:      $sel \leftarrow \{k\}$
6:      $i \leftarrow g.Decompositions[k].TimesSelected$
7:    else if $g.Decompositions[k].TimesSelected == i$ then
8:      $sel \leftarrow sel \cup \{k\}$
9:    end if
10: end for
11: if $|sel| == 0$ then
12:   return $g$
13: else
14:   if $|sel| == 1$ then
15:     return $sel[0]$
16:   else
17:     return $RandomSelect(sel)$
18:   end if
19: end if
```

To allow for this desirability we introduce a discount factor, γ, on goals and decompositions. This discount value is bound by the following $\gamma \in (0, 1]$. The desirability of the goal or decomposition is then defined by how low the discount value is; the lower the discount value the higher the desirability.

The algorithm for selecting goals and decompositions based on MDE is the same as for ME with the exception that the number of times selected, used for lowest value, is multiplied by the discount factor to give a different effective number of times selected.

4.2.2 *Component planning*

Once a sequence of goals has been selected it is time to plan how to achieve each goal while minimizing the impact on the environment explained below. Each goal is planned from the first goal in the goal sequence to the last. The objective of planning a goal is to find a sequence of components that will take the NPC from the state of the world to the state of the world where the goal is satisfied. This immediately poses a problem as BTs are stateless.

This is solved by creating a world state object which contains all preconditions of all goals and components that currently hold, as described previously. We then check each precondition at the time of starting the planning. We plan from the first goal in the sequence of goals to this world state. Once a valid sequence of components has been found we apply the effects of the components to the world state object and then use the modified state object for planning the next goal in the sequence and repeat this process for each goal.

To find a sequence of components that solve a goal we use the A* search algorithm from section 2.3 to find a path backwards from the goal to the current or expected world state.

There is no explicit directed graph to search through but instead we construct a graph backwards from a node that has preconditions of the goal. The objective of the search is then to reach a node whose preconditions are contained in the current or expected world state. For the search we utilize an A* search algorithm.

At any node in the graph, we find all components in the component library that satisfy at least one of the preconditions of the node. These components will form the edges in the search graph. A new node is added to the search graph for each component. These new nodes each contain the union of the component's preconditions and any preconditions that the component did not satisfy from the previous node. The edge from the new nodes to the current node reference the component that was selected. This process is repeated until we find a node whose preconditions are contained in the current or expected world state.

To compare this to classical planning from chapter 2, the node containing the preconditions of the goal can be seen as a goal state in a state-transition system. The node whose preconditions are contained in the world state can be seen as s_0, the initial state in the state-transition system. Components are equal to actions in the state-transition system.

The A* search algorithm relies on a function that calculates the current cost of the path chosen plus a heuristic for how much it will cost to go from the current node to the goal node, in our case the world state node. The function is defined as:

$$f(n) = h(n) + g(n) \tag{4.3}$$

This requires us to define the two functions $g(n)$ and $h(n)$.

To define $g(n)$ we use the cost of each component. The cost is defined as the number of effects the component has, as previously mentioned. The more effects the higher the cost. The reason for selecting the number of effects as the cost of a component is that we want to leave as little an impact on the game environment as possible. We want to minimize the effects as this could, although we do not account for it, have a negative effect on later plans.

The definition of $h(n)$ is more complicated. As defined in the general A* algorithm the heuristic function must be admissible, meaning that it must never overestimate. To make the function admissible we define it as:

$$h(n) = |P_n - (P_n \cap P_s)| \tag{4.4}$$

Where P_n is the set of preconditions of node n and P_S is the set of preconditions that the world state contains. This function calculates the number of preconditions that are in P_n that are not in P_S and thus still need to be satisfied. We use this as the value of the heuristic function. The relates to the approach in GOAP [13]. The intuition behind this is that the fewer preconditions that still are not satisfied by the world state the fewer effects we need to find in the component library which also relates to $g(n)$ as we work to minimize the needed effects. The function never overestimates as we will necessarily need a sequence of components that has $h(n)$ effects to satisfy the preconditions at the current step.

Our definition of the $h(n)$ is also consistent. From section 2.3 consistency requires that $h(n)$ is not greater than the cost of using the component c to go from n' to n plus $h(n')$. Observe that the cost of c is at least the number of preconditions of n that we satisfy and that $h(n')$ is at least the remaining number preconditions of n that c did not satisfy. This means that the minimum value of $cost(c) + h(n')$ is $h(n)$.

The success or failure of the planned sequence of components can now be defined. If the sequence was successfully executed and the goal is satisfied, the sequence was successful. If the sequence was successfully executed and the goal was not satisfied something went wrong and a re-planning is triggered. If re-planning is triggered and no plan can be found the goal is no longer relevant.

4.3 IMPLEMENTATION

In Figure 4.2 the syntactical implementation of the planning architecture in BTs is done by using a single node, the *Planning* node. This implementation was inspired by the implementation of the query node in [3] as this approach has a very little impact on the general BT structure. Also, we believe that the planning node allows us to maintain the advantage identified in chapter 3. This node uses a single goal written below it and uses the planning system to plan how to achieve the goal. The planning node is similar to the link node in the sense that it too cannot have any children and that it uses a BT internally.

The status of the node is defined by the virtual BT. If the virtual BT succeeds, the node succeeds. If the virtual BT fails a re-plan is triggered. If no plan can be found the node fails. As described previously the planning system will select a sequence of goals and a

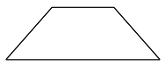


Figure 4.2: Node syntax for the Planning Node.

sequence of components solving each goal in the sequence. This translates very nicely to a BT structure. Figure 4.3 shows a general model of the internal, or virtual, BT that is generated and used inside the planning node. It has a general structure of first a sequence node as root with a child for each of the goals in the selected sequence of goals. Each child is a sequence node with a child for each component for solving the goal. The leafs in the virtual BT are link nodes that link to the respective BT that the component references. The planning system

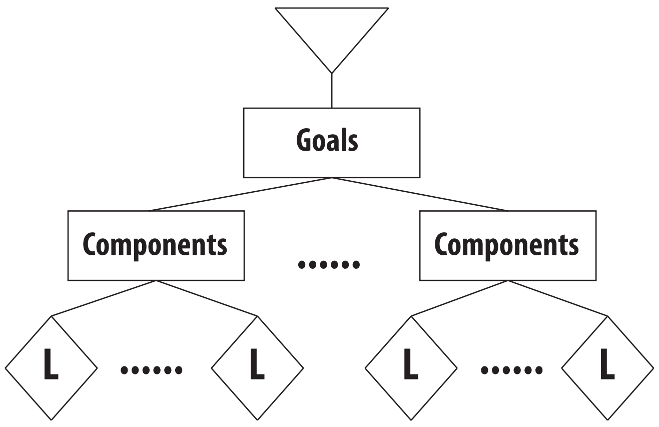


Figure 4.3: Generic structure of the virtual BT used internally in the planning node.

has not been implemented with focus on performance but rather with focus on showing that it is possible to utilize planning in BTs in an easy and sensible way. This means that the planning system will be evaluated with main focus on showing that it works and with less focus on actual efficiency.

To show what the extension with planning does to the overview of Unity and the BT framework, we have updated the Figure 3.3 from subsection 3.1.1. This change is shown in Figure 4.4. The BT framework and the planning framework come in two DLL files that make up the entire framework. The planning DLL is dependent on the BT DLL but the BT DLL can be used alone. The DLLs are included in the code behind the Unity project and this way all the required classes are available in the game environment. As we described in section 2.1

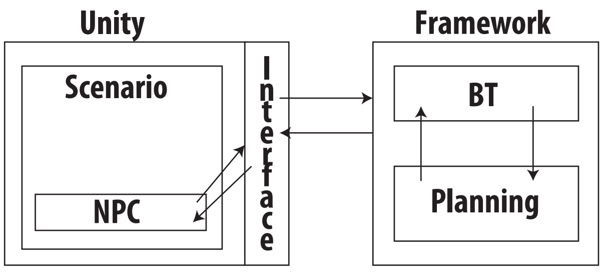


Figure 4.4: Overview of Unity and the BT framework extended with planning.

we changed the syntax and semantics as well as the general execution of BTs compared to the previous work done in [10]. The changes have not been fully implemented as we have not moved conditions onto sequences, selectors, parallels and their child branches. As we have not and will not need to use these in any of the tests in this report we chose to not make those changes in the framework.

4.4 SUMMARY

This concludes the description of the architecture for goal decomposition and planning that the planning node uses. We have described what preconditions, effects and the world state are along with how these interact. Goals have been defined and related to preconditions, effect and the world state. Components were described as BTs and how effects and preconditions are set on each component in the library.

With all the above in place we described how the goal decomposition works with both ME and MDE. Once a goal decomposition is selected the actual planning of how to achieve each goal was initiated by using an A* search from the goal, through components in the component library, to the world state.

Finally we described how this affected the execution of the initiating planning node and what triggers a re-planning.

5 EXPERIMENTS

The planning architecture has now been explained and we will present experiments that will demonstrate the different components of the architecture and how they work compared to static BTs generating similar behavior. When we use the word static about the behaviors in the following experiments we refer to the fact that these behaviors are all predefined. Even though they contain random elements in the form of probability selectors the outcome of each choice of is still static.

During the experiments we use the scenario as it is defined in section 3.1.

5.1 EXPERIMENT 1

Description

In this experiment we demonstrate the ME goal decomposition algorithm. The behavior that will be run will make the NPC run from **Start** to **Bar**, then to **Patron** and finally back to **Start**. The experiment is designed so that the entropy measured will be based on which of the three actions, walk, run and teleport, the NPC chooses when moving to one of the location, thus we expect to see a uniform distribution between the possible selections. The experiment will run for 60 iterations to allow an adequate dataset to base our conclusions on.

Static Behavior

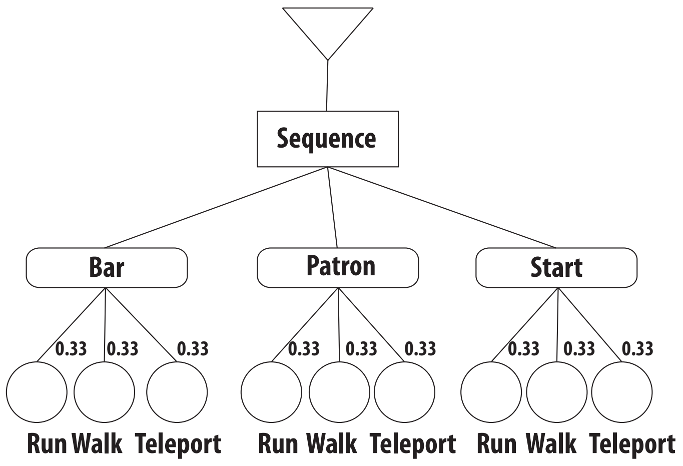
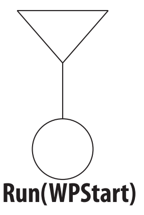


Figure 5.1: Static behavior for Experiment 1.

The static behavior, as shown in Figure 5.1, illustrates a simple BT consisting of a single sequence node as the root and with three probability selectors as its children. Each probability selector will be handeling how to get to each location. The probability selectors have three children each, one for each of the possible movement actions. The children have a uniform probability of 0.33 but due to implementation we will be setting the last, **Teleport**, to 0.34.

Planning Behavior

The planning based BT is built by using a single sequence node as the root of the tree. As its children are three planning nodes, each with a goal of moving to one of the three waypoints. Each goal has three possible decompositions, containing a single goal that describe exactly how to perform the move for achieving the goal. The goal graph for one of the goals is shown in Figure 5.2. The graph is identical for the two remaining goals and can be seen in Appendix C. Each of the three decompositions have a single component in the library that can satisfy it and thus also the primary goal.

Name:	RunToWPBar
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtBar \wedge \neg AtStart \wedge RanTo\}$
Cost:	3

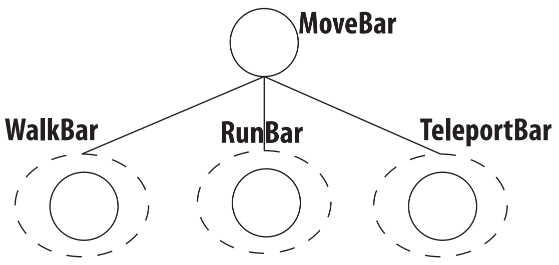


Figure 5.2: Part of the goal decomposition graph for Experiment 1.

Evaluation

Each of the two approaches performed the expected behavior with the exception of the planning behavior's move goal, e.g. **MoveBar**, allowing the planning BT to randomly select one of the three movement actions to solve it. To illustrate the difference between the probability selector and the goal decomposition algorithm's ability to perform the actions in an, as close to, uniform way, we show graphs that indicate how close to uniform the selections are made. Note that the graphs show the probabilities on the y-axis and number of selections on the x-axis.

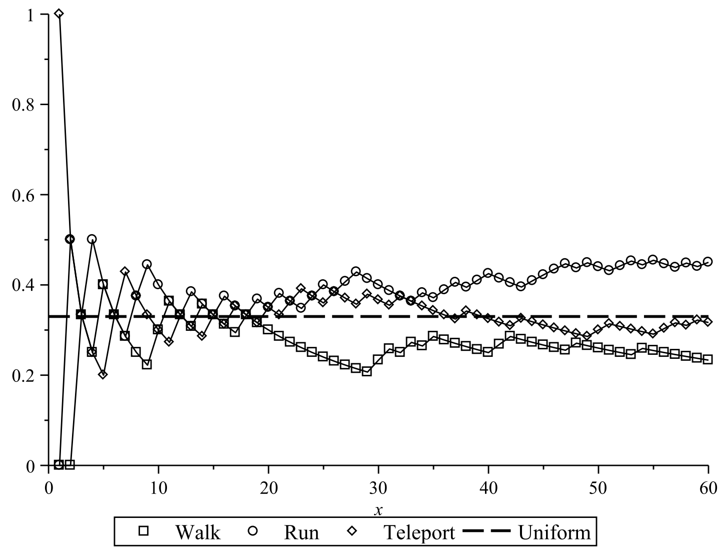


Figure 5.3: The graph for the static behavior's selections of movement to **Bar**.

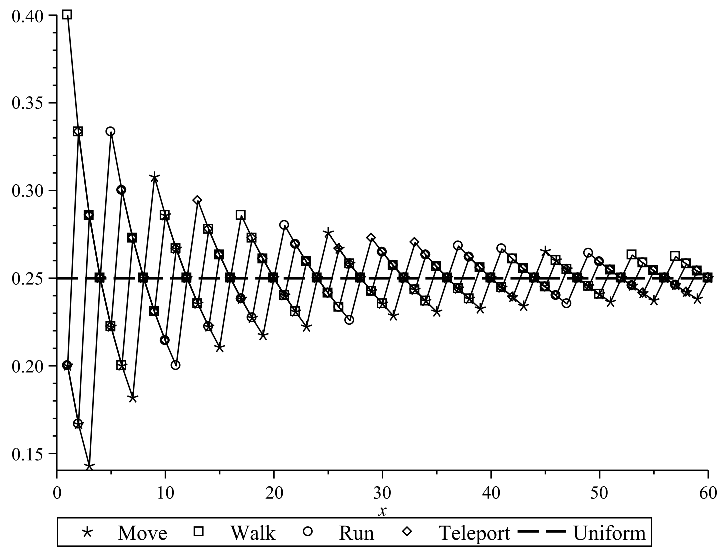


Figure 5.4: The graph for the planning behavior's selections of movement to **Bar**.

Figure 5.3 and Figure 5.4 show the actual probabilities of each action or goal at each of the 60 selections. The straight dashed line is set at uniform distribution and thus the results are expected to be close to or on the dashed line. As we only select on of the possibilites each time there will be a little curve on the line. It is easy to see that the planning behavior is able to stay consistently around a uniform distribution compared to the static behavior.

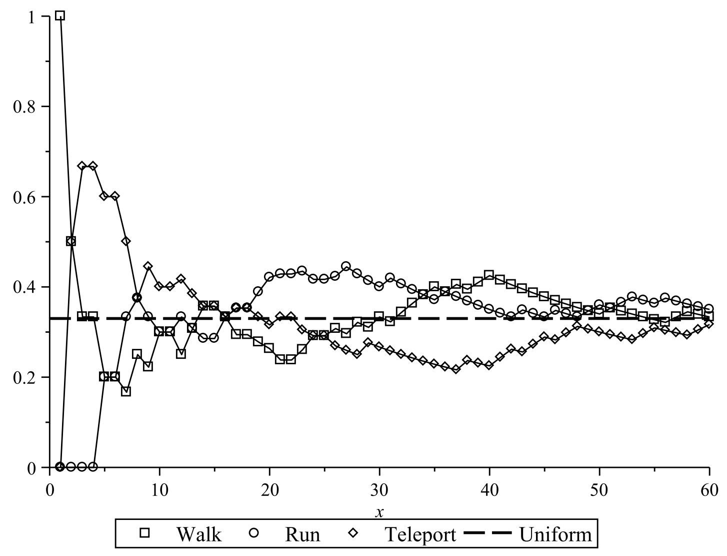


Figure 5.5: The graph for the static behavior's selections of movement to **Patron**.

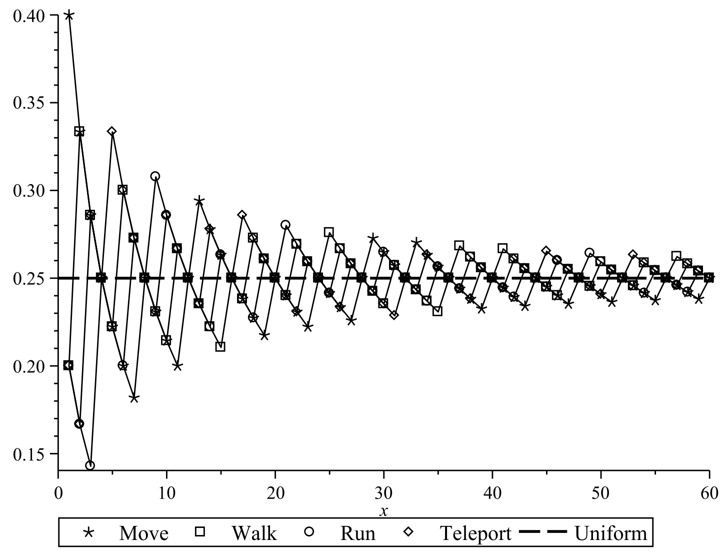


Figure 5.6: The graph for the planning behavior's selections of movement to **Patron**.

Figure 5.5 and Figure 5.6 show the same tendencies as above, but for moving to **Patron**. that the planning behavior is better at maintaining uniform distribution.

As with the above figures, Figure 5.7 and Figure 5.8 with results for moving to **Start**, show that the uniform distribution is best maintained by the planning behavior. The reason for this is because where the probability selector never remembers what it has done on the last execution, the goal decomposition algorithm maintains a counter for each goal and its decompositions. Based on this it can always select the goal or decomposition that best maintains uniform distribution of total selections.

The results indicate that the ME algorithm is working as intended and will continue to work towards a uniform distribution at each selection. Depending on the structure of the goal graph this can give a much more controlled and accurate diversity of the NPC.

Looking at performance, there is not much of a difference between the two approaches. The static behavior maintains an average of 64.07 *FPS* and the planning behavior maintains 64.08 *FPS*. The experiments can vary by 0.05 between experiments. This means that the execution performance of the static approach and the planning approach is close to equal.

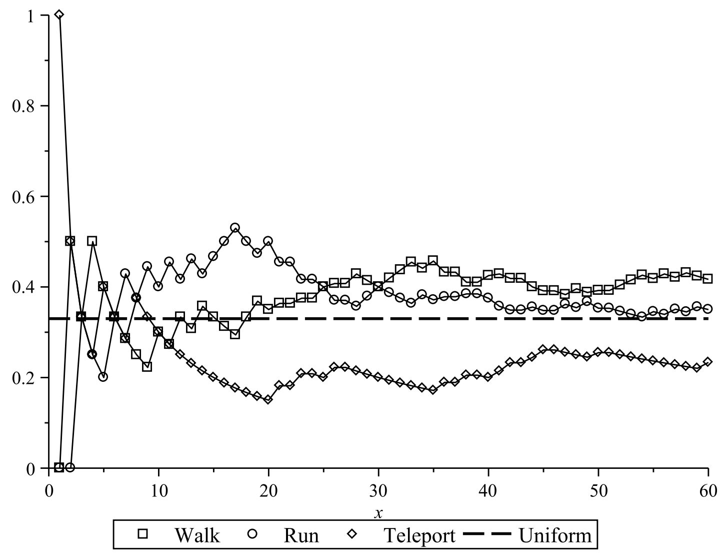


Figure 5.7: The graph for the static behavior's selections of movement to **Start**.

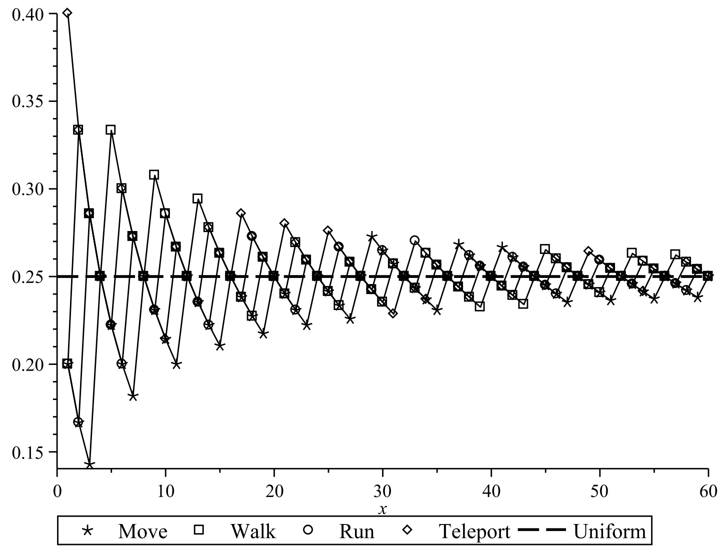


Figure 5.8: The graph for the planning behavior's selections of movement to **Start**.

5.2 EXPERIMENT 2

Description

This experiment is similar to the previous experiment except instead of using the ME algorithm we will be using the MDE algorithm. We want the NPC to use the **Run** action ten times more often than the **Walk** action and the **Teleport** action twice as often as the **Walk** action. Based on this we expect to see the number of selections of **Run** to be substantially higher than the two others and **Teleport** to be higher than **Walk**. The experiment is again run for 60 iterations to allow for enough data to be collected.

Static Behavior

The BT in this experiment is identical to the static BT in the previous experiment with the exception that the probability distribution of the probability selectors has been changed. As

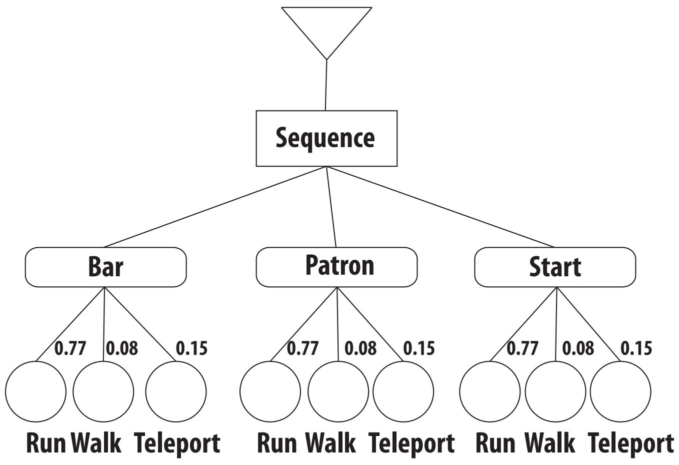


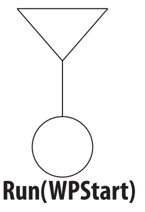
Figure 5.9: Static behavior for Experiment 2.

described above we want to favor **Run** the most, then **Teleport** and lastly **Walk**. We calculate the probabilities by counting that each time **Walk** has been executed once, **Run** has to be executed ten times and **Teleport** two times. This gives a total of 13 and thus the probability of **Walk** is $\frac{1}{13} = 0.08$, **Run** is $\frac{10}{13} = 0.77$ and **Teleport** is $\frac{2}{13} = 0.15$. The corresponding BT can be seen in Figure 5.9.

Planning Behavior

The goal structure and component library for this planning BT is the same as the previous except we have added discount values to the move goals and their decompositions. An example can be seen in Figure 5.10, and the full goal graph can be seen in Appendix C.

The move goal and the walk goal have a discount of 1.0. This is done not to modify the number of times selected. The run goal has a discount of 0.1 meaning that its effective number of selections is a tenth of the real number of times selected. This means it will be chosen ten times as often as the walk and move goals. The teleport goal has a discount of 0.5 making its effective number of selections half of the real number. This means it will be chosen twice as often as the move and walk goals. Thus we achieve the intended probabilities as described above. There is however one exception. The move goal has a discount value of 1.0 and each of the components for that goal graph can solve it and thus a random move action is selected roughly every 13th selection. This breaks with the described behavior but it does not invalidate the experiment as the additional selection option for the planned behavior does not affect how well it selects between them. In Figure 5.10 a part of the goal graph is shown. This goal graph is identical for each of the three move goals.

Name:	RunToWPBar
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtBar \wedge \neg AtStart \wedge RanTo\}$
Cost:	3

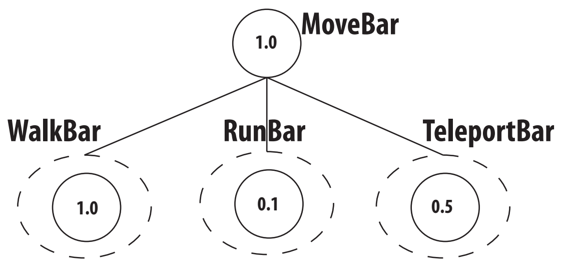


Figure 5.10: Part of the goal decomposition graph for Experiment 2.

Evaluation

As in the previous experiment both behaviors are performed as expected. Note that the graphs show the probabilities on the y-axis and number of selections on the x-axis. The **Run** action is selected substantially more than **Walk** and **Teleport**. And **Teleport** is selected about twice as often as the **Walk** action.

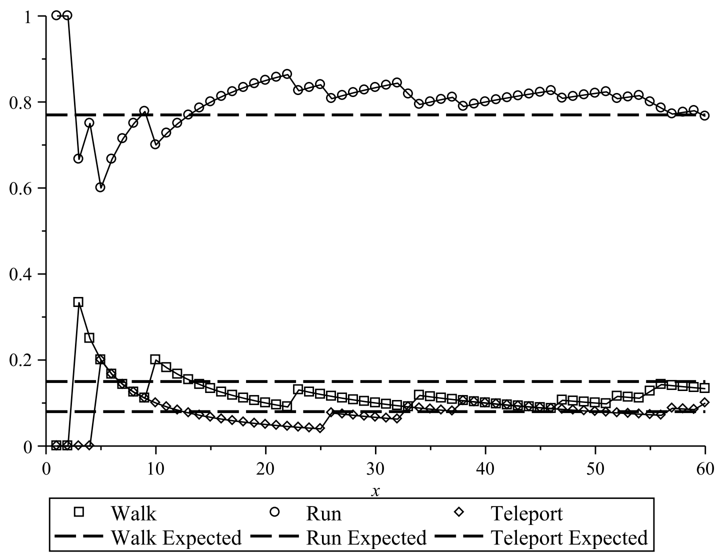


Figure 5.11: The graph for the static behavior's selections of movement to **Bar**.

Figure 5.11 and Figure 5.12 show the actual probabilities of each action or goal at each selection for moving to **Bar**. There are now three dashed lines. The top dashed line is the expected probability of the **Run** action and goal. The middle dashed line is the expected probability of the **Teleport** action and goal and lastly the lowest dashed line is the expected probability of the **Walk** action and goal, and the move goal. It is again easy to see that the planning behavior consistently attempts to approach the intended probability values given by the discounts. The static behavior also exhibits the same tendencies but is less successful at consistently maintaining the intended probability distribution.

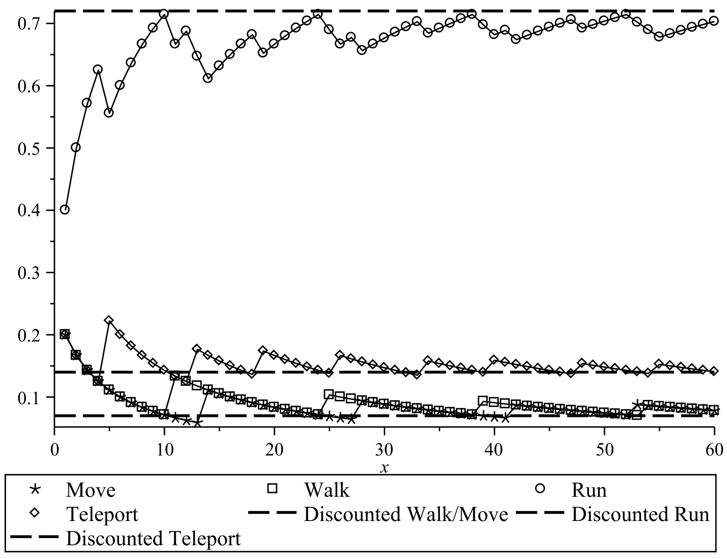


Figure 5.12: The graph for the planning behavior's selections of movement to **Bar**.

Figure 5.13 and Figure 5.14 show the same tendency as above, but for moving to **Patron**. The planning behavior attempts to get as close to the intended probabilities as possible and does so successfully, while the static behavior is less consistent.

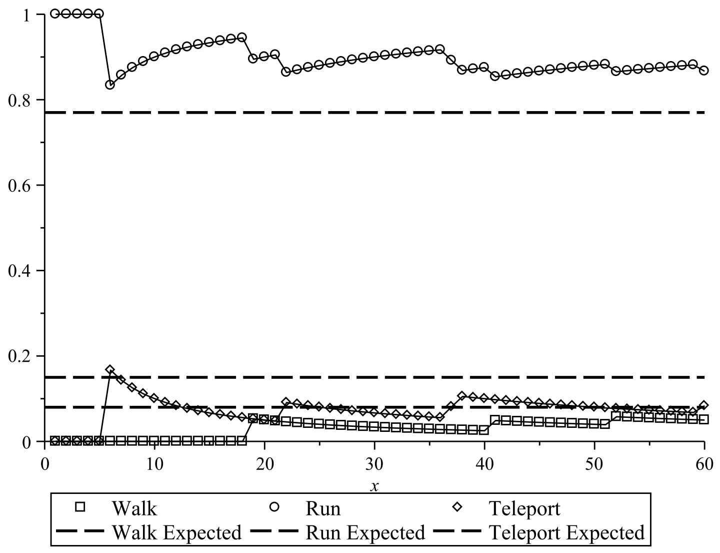


Figure 5.13: The graph for the static behavior's selections of movement to **Patron**.

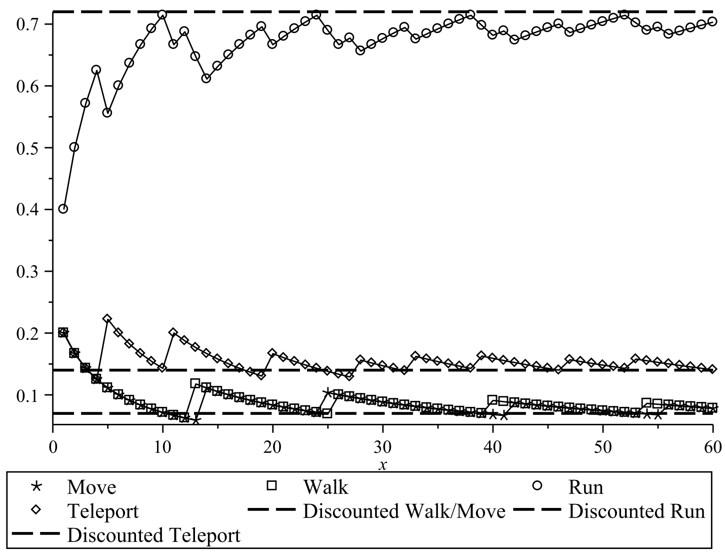


Figure 5.14: The graph for the planning behavior's selections of movement to **Patron**.

As with the above figures, Figure 5.15 and Figure 5.16, show the same tendencies, but now for moving to **Start**. The probability selectors are again performing less successfully because theu do not keep a history of previous selections. They are only concerned with current selection thus it is possible to select the same action several times in a row, even though it has a low probability.

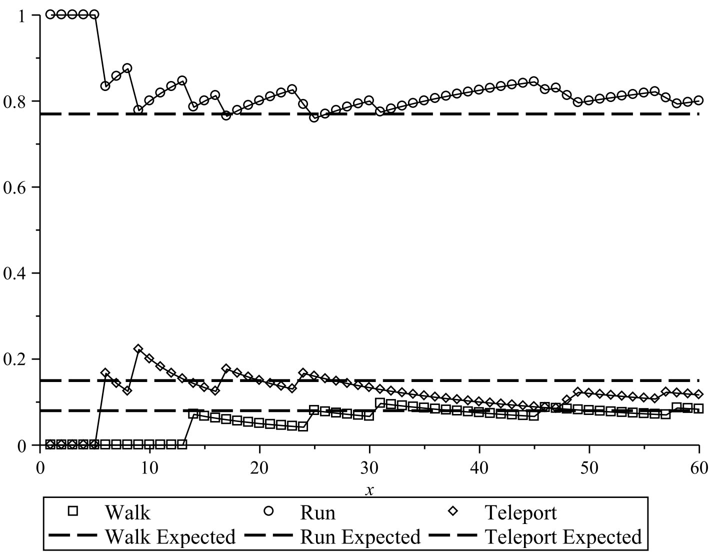


Figure 5.15: The graph for the static behavior's selections of movement to **Start**.

The experiment shows that the MDE algorithm is working as we expect it to. As opposed to the previous experiment we are now concerned with controlling more precisely each goal and its decompositions. This gives a more defined control over the NPC and if this is e.g. combined with a process that will change the discount values at run-time based on changes in the environment this can be a very powerful way of managing the NPC behavior.

Performance wise we see the same tendency as in the previous experiment. The static behavior maintains an average of 64.06 *FPS* and the planning behavior maintains an average of 64.07 *FPS*. The difference is again so small that it is irrelevant due to a normal variation of *FPS* between experiments.

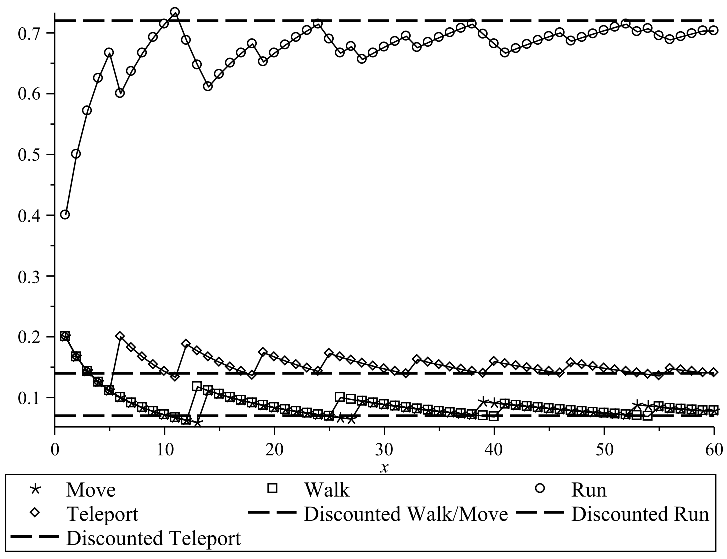


Figure 5.16: The graph for the planning behavior's selections of movement to **Start**.

5.3 EXPERIMENT 3

Description

The purpose of this experiment is to demonstrate the A* search algorithm to find a component sequence for solving a goal. We provide a single simple goal that is to move to **R1** where the door is both closed and locked. We compare the planning approach with a static BT that performs this behavior.

Static Behavior

The static behavior in this experiment is a modified version of a previously used BT. The BT in question was used in chapter 3 and described the behavior of moving the NPC into a room. We have modified which room is used and also the NPC should not in this case exit the room again. The BT is shown in Figure 5.17

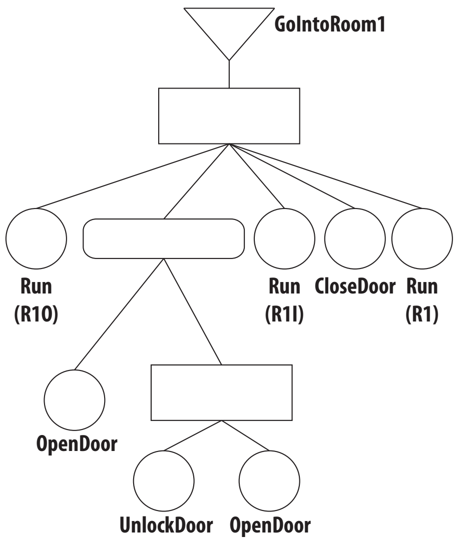
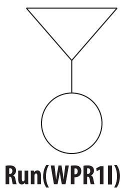


Figure 5.17: Static behavior for Experiment 3

Planning Behavior

The planning behavior uses the single simple goal, of moving into room $R1$, as the primary goal. This goal has no decompositions, so the decomposition algorithms are disregarded. The interesting part here is the component library which is what the experiment is designed to show. The component library for this experiment is shown in Appendix B. Below the component **RunToWPR1I** is given as en example:

Name:	RunToWPR1I
Preconditions:	$\{DoorOpen \land AtR1O\}$
Behavior Tree:	
Effects:	$\{AtR1I \land \neg AtR1O\}$
Cost:	2

Evaluation

The experiments were performed with success by both the static and the planning behaviors, as was expected. The static BT moved the NPC up to the door and using its sequential selector it was able to first attempt to open the door, andwhen that failed, triedd its other branch which included a **UnlockDoor** action followed by a **OpenDoor** action. These both succeeded, making it possible for the NPC to move to the goal destination.

The planning behavior had a much more complex task at hand and we will describe here which steps it took to createthe solution plan.

In the following please note that the circles refer to states in the search graph. The initial circle is the goal, and is denoted by G. The world state node will also be shown, and is denoted by s_W. Text on the arcs define the component that will change the state from the originating state of the arc to the target state. Recall that we use backward planning and the arcs are directed accordingly. The text inside the circle defines the cumulative preconditions at that node. Note that the $h(x)$ and $g(x)$ values from the A* search are also included at each state.

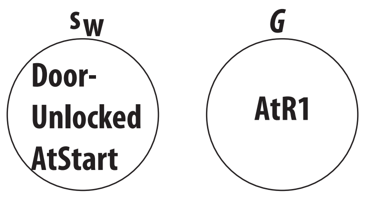


Figure 5.18: The preconditions of the world state and the goal

Figure 5.18 shows the goal and the world state for Experiment 3. The current world state for the first planning attempt defines that the NPC is at **Start** and **DoorUnlocked**. The goal requires the NPC to be at **R1** to be satisfied.

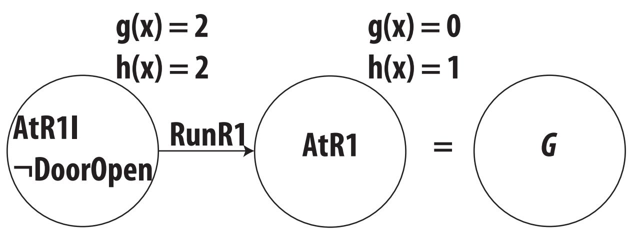


Figure 5.19: Expansion one of the A* search

In Figure 5.19 we see the first step of the search through the component library. The goal is seen to the right with only one single component capable of solving it. The resulting state is connected to the goal via the component **RunR1**. The new leaf node of the search graph has two preconditions that need to be satisfied; **AtR1I** and ¬**DoorOpen**. They stem from the component **RunR1**

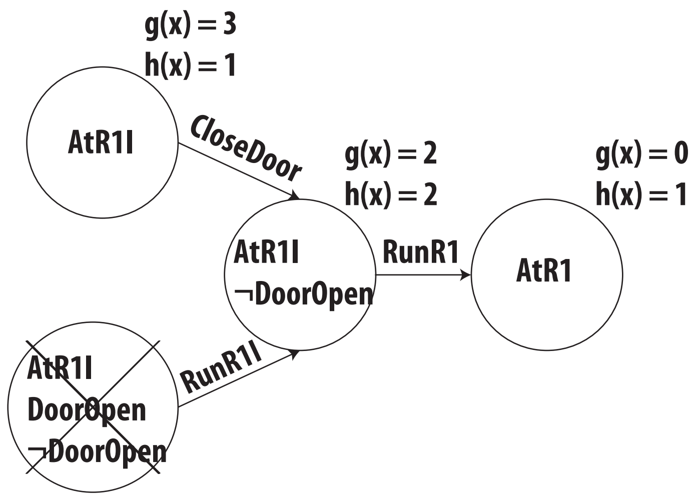


Figure 5.20: Expansion two of the A* search

In Figure 5.20 the algorithm expands the state by finding components that can satisfy the preconditions. This results in two candidates; **CloseDoor** and **RunR1I**. However the **RunR1I** component must be discarded as it will generate a state that has two contradicting preconditions, **DoorOpen** and ¬**DoorOpen**, and thus unreachable.

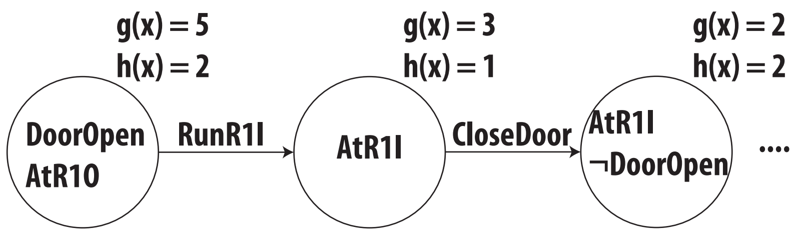


Figure 5.21: Expansion three of the A* search

We then attempt to expand the state before the **CloseDoor** component in Figure 5.21. This state has a single precondition **AtR1I**, because the precondition of the **CloseDoor** component and the component's resulting state is the same. The expand finds a single component able to satisfy the preconditions, namely the **RunR1I** component.

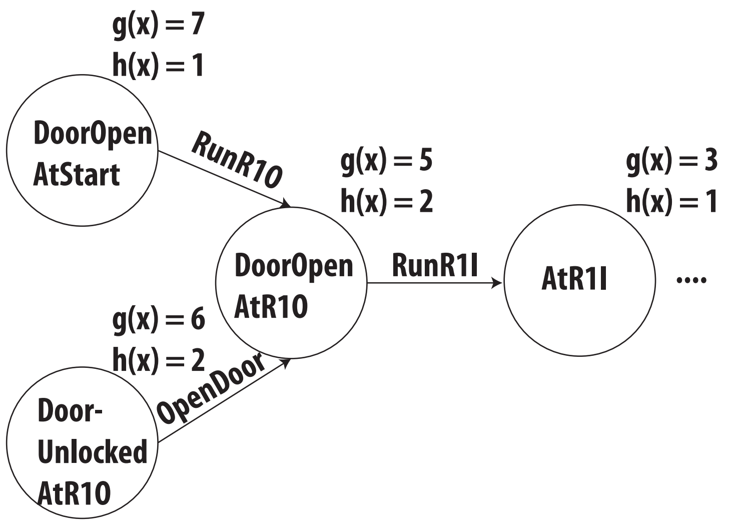


Figure 5.22: Expansion four of the A* search

The current state is now defined by the preconditions of the **RunR1I** component, namely **DoorOpen** and **AtR1O**. We expand this state in Figure 5.22. This gives two candidate components; **RunR1O** and **OpenDoor**. Each will have one precondition that they cannot satisfy from the current state. The states before the two components are added to the leaf set with the preconditions of the components and the additional precondition that was not satisfied.

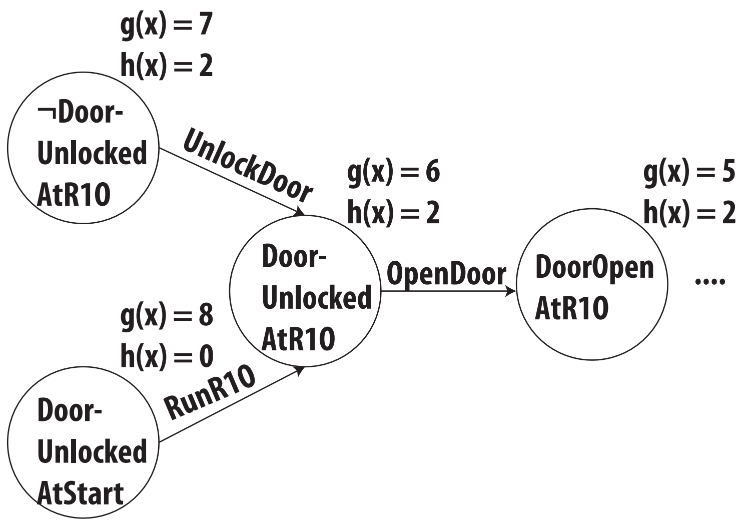


Figure 5.23: Expansion five of the A* search

The leaf set now contains two nodes. Both states have an $f(x) = 8$ and we rely on the implementation to select between them. We select the state before the **OpenDoor** component. We expand it as shown in Figure 5.23. The states before the two candidate components, **UnlockDoor** and **RunR1O**, are added to the leaf set.

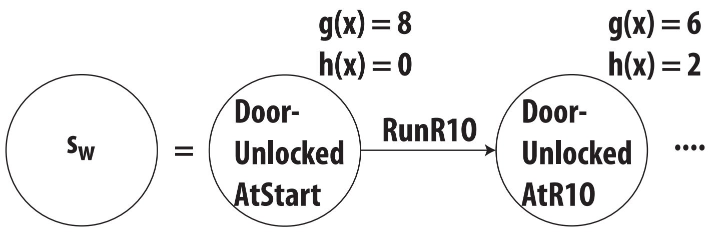


Figure 5.24: Expansion six of the A* search

We examine the leaf set and find that the state before **RunR1O** component from expansion 5 is currently cheapest. Before we try to expand we always check to see if the current cheapest node is satisfied by the world state. In Figure 5.24 we see that the world state satisfies the state before the **RunR1O** component, as the preconditions of the state hold in the world state from Figure 5.18. We now have a plan that is able to take the NPC from **Start** to **R1** in the scenario.

The planning system is however fooled by the fact that it thinks the door is unlocked when in reality it is not. Thus when the NPC attempts to open the door, this component fails. This causes the planning node to detect a failure and attempts to replan with the new information that the door is locked. The planning sequence proceeds exactly as above with the exception that a valid plan will be found by starting at the state before the **UnlockDoor** component, as shown in Figure 5.23.The planning behavior now successfully enters the room completing the goal. As with the previous experiments we looked at the speed at which it was possible to execute the two approaches.

The planning behavior maintained an average of 57.79 *FPS* while the static behavior maintained 58.33 *FPS*. This gives a slight performance advantage to the static behavior, as expected. However the difference between the two is very small considering that the planning behavior manages to do an A* search twice through the small component library.

Both *FPS* counts are lower than in the previous tests and this can be explained by the fact that a lot of initializing is done when starting the scenario. This cost is more visible in an experiment that runs for a short amount of time, as in Experiment 3, compared to the much longer running experiments Experiment 1 and Experiment 2.

5.4 SUMMARY

We have now tested the two major parts of the planning architecture, the goal decomposition algorithms and the component planning search and shown that they work in the scenario presented.

The results show that the approach of using a two phased planning architecture can work in conjunction with BTs using the planning node. The use of this single node, with a hidden BT generated on runtime, made little intrusion in the overall readability of BTs as shown in the expirements but it of course makes the AI more complex and harder to fully grasp. With a larger scenario, e.g. a full game, the component library will be much larger and the amount of goals and the complexity of the goal graph will be larger. This will make it harder for designers to understand the capabilities of the AI but is in turn a powerful tool for making the AI more diverse and compared to a static implementation as shown in Experiments 1 and 2.

Experiment 3 showed that the planning architecture was able to make plans in a simulated fully observable game environment. It also showed that the architecture is able to cope with errors in the expected world state, e.g. the door was discovered to be locked, and replan according to the new perception of the world. Thus showing adaptive behavior.

One thing that shined through in all three experiments was that formulation of preconditions and effects in relation to the environment proved difficult. The correct formulation and implementation of all preconditions and effects is of very high importance as these impact the correct functioning of the planning architecture. It will be necessary to improve the preconditions and effects and how they are defined as further improvement of the architecture.

6 CONCLUSION

We hereby conclude this thesis by summing up the contributions and results found. The purpose of this thesis has been to investigate and propose a possible extension of BTs with planning while maintaining the advantages and strengths of the BT formalism.

In chapter 2 we presented the formalism of BTs, including both syntax and semantics, used in our work. The formalism includes a brief discussion of some proposed changes to the formalism compared to that used in our previous work on the subject. These changes concerns condition and decorator nodes. We propose for condition nodes that they are removed from the syntax and instead included on the other nodes in the syntax, while maintaining their semantics.

We investigated the notion of planning and A* search, both on their own, to build a notational foundation for the thesis, but also in relation to game AI and how these approaches can be used in that context. Our look at related work revealed that planning is a feasible approach to game AI, where examples of both STRIPS and HTN planning were mentioned. This answers the subquestion from our problem statement on the feasibility of using planning for game AI.

A comparison of BTs and scripting was carried out in chapter 3 to identify advantages and disadvantages in the BT formalism. Our results indicates that BTs were comparable performance wise to scripting, while offering the following advantages:

- Readability
- Modularity
- Reusability

We assume that the advantages hold for both programmers and non-programmers alike, making it generally easier to utilize for game AI. With these advantages identified we have a basis for investigating the impact of the proposed extension to BTs.

We described the extension to BTs in chapter 4, which is our proposed approach for incorporation classical planning into BTs. We implemented a new node in the BT syntax called a planning node. Connecting planning with BT through the use of a new node construct ensures that we maintain the identified advantages of BTs. Though, the complexity of the planning node and it's semantics can be hard to interpret compared to the existing nodes. The planning node allows both goal decomposition, through entropy, and planning to achieve these goals with components using A* search.

The experiments carried out in chapter 5 were aimed at validating the expected functionality of the goal decomposition and component planning algorithms. Our results on the goal decomposition algorithms show that the use of entropy ensures controlled diversity in the goal decomposition compared to using a similar BT. Interestingly, the results also indicated that the probability selector node is not guaranteed to adhere to the intended probability distribution depending on the number of node executions taken into consideration. This is a result of the inherent flaws with current randomization approaches. The final experiment showed that the planning node is able to plan a sequence of components for the NPC to execute, and initiate replanning if discrepancies to the expected world state are detected during execution.

With the above summarization of the work conducted in our thesis, we now turn to the main problem of our problem statement from section 1.1. To conclude, we have proposed a working approach that extends the notion of BTs with an adapted notion of classical planning. Diversity is achieved in both goal decomposition through entropy, and component planning. That is, although a planning node will always be statically defined with respect to its location in a BT, the world state can differ between executions of the BT. This effectively results in diversity in the constructed component plan. The goal decomposition also provides diversity in the sense that the algorithms guarantee goal selection distribution to be either uniform or discounted uniform.

6.1 FUTURE WORK

Though the implementation and testing succeeded there are still a number of things that need further investigation. We will mention a few of those that we find to be the most important.

Optimizing the planning algorithms

The planning architecture has not been implemented with focus on performance. This leaves room for optimizing both the goal decomposition algorithms and the searching.

The goal decomposition algorithms run through the entire set of decompositions every time. This could be avoided by sorting the list on the number of times selected as a general solution. An even more ideal solution would be to keep a list of the goals that currently lowest times selected values. Currently this is computed everytime.

There are also still a few corner cases where the planning algorithm cannot find a valid plan, even though one exists. An example of this is if the planning algorithm cannot find a valid component sequence for the chosen goal decomposition. The planning node will then fail and not attempt to re-plan. There might however be another goal decomposition that is solvable.

Another problem that needs a solution is the amount of time the planning needs to finish. If a sufficiently complex goal needs to be planned the process of planning it might stretch beyond the scope of a single game frame. What could be done to avoid this *FPS* slowdown, is to pause the plannning, and reinitiate it the following game frame. This requires that the chosen goal is decomposed to multiple goals, such that the planning can be naturally paused as each of these goals have a solution plan. Thus the NPC can remain active although the planning node has not found solutions for all goals. The general principle is described in [5].

Save generated plans

Plans generated by the component search are currently not saved in any way. Meaning, if a goal has to be solved again we do not remember the previous solution plan and need to re-plan it. This is also related to the above section above on optimization.

It can also be used to control diversity. We could force the algorithms to attempt to generate alternate plans instead of often reaching the same plan for the same goal. This would be a matter of comparing the generated plans, which is just comparing the two BTs generated. The matter of comparing the trees is easy and could be a good tool for controlling the NPC.

Adapting effects and preconditions

During the experiments in chapter 5 we discovered that the formulation and implementation of preconditions and effects was harder than expected. One of the largest problems we found was that some preconditions may be abstractions of other, both conjuctive and disjunctive preconditions, which we in no way support.

To exemplify this, consider two components that interact with a door. These could intuitivly have a precondition **AtDoor**. Now consider two components that move an NPC. Each component moves the NPC to one side of the door, but the two locations are different. Intuitivly both locations are at the door but the effect of the two components cannot solely be **AtDoor** as it then would not be possible to distinguish which component moves the NPC where. We would then need to add additional effects that define at which location the NPC will be after executing the component.

Another solution to this is to allow preconditions to consist of other preconditions, as described above. This would allow us to construct the precondition **AtDoor** as a disjunctive precondition consisting of e.g. **OutsideDoor** and **InsideDoor**. The effect **OutsideDoor** of one of the movement components is then enough to satisfy the precondition **AtDoor**.

It would also be a great improvement to allow preconditions and effects to be parameterized e.g. replace **AtR1** and **AtR1I** from the experiments with a single parameterized precondition, **AtLocation**(**x**), where **x** could then be replaced by **R1** or **R1I**. However, one should be aware that allowing too high level of expressiveness in the preconditions can have negative impact on both the decideability and complexity of planning [4, Chapter 3].

Further use for preconditions

With the addition of conditions on sequences, selectors, parallels and their children, in the BT formalism, the preconditions of the planning architecture can be used for an additional purpose. For each sequence node of components that solve a goal, each branch to the corresponding link node, could have the component's preconditions added as a condition. This will allow a component to detect a discrepancy in the world state with respect to the expected world state. A discrepancy could occur if the component, prior to the current component, succeeded, but the world was not left in the expected state. It is then possible to have the planning node fail and trigger a re-planning before attempting to execute a component that cannot succeed as its preconditions are not met.

Large scale testing

The planning architecture has in this thesis only been tested on a simple scenario where only one NPC was controlled and with a very small amount of possible actions. To further investigate the viability of the approach it should be applied to a more complex scenario with multiple NPCs and also with collaboration between them. The number of actions in the environment should also be increased to increase the complexity of the possible behaviors of the NPCs. This would also increase the number of possible goals and components allowing for proper performance testing of the architecture.

A SCRIPTS FOR COMPARISON

Note for the listing below that *this.controller* refers the NPC and is specific for Unity. The listings show the exact implementation of the scripts in Unity. In chapter 3 some of the implementation specific methods and object references were exchanged with something more meaningful.

BASIC BEHAVIOR

```
if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "
    WPStart").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPBar"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name ==
    "WPBar").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPPatron"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name ==
    "WPPatron").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR30"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name ==
    "WPR30").transform.position) && this.controller.State == State.Idle){
    this.controller.OpenDoor();
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR3"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name ==
    "WPR3").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPStart"));
}
```

Listing A.1: The basic behavior as a script in Unity.

EXTENSION ONE

```
if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPStart").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPBar"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPBar").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPPatron"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPPatron").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR3O"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR3O").transform.position) && this.controller.State == State.Idle){
    this.controller.OpenDoor();
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR3"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR3").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1O"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR1O").transform.position) && this.controller.State == State.Idle ){
    if (!this.controller.GetClosestDoor().open){
        if (!this.controller.GetClosestDoor().locked){
            this.controller.OpenDoor();
        }
        if (!this.controller.GetClosestDoor().open){
            this.controller.UnlockDoor();
            this.controller.OpenDoor();
        }
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1I"));
    }
    else{
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPStart"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR1I").transform.position) && this.controller.State == State.Idle){
    if (this.controller.GetClosestDoor().open){
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1"));
    }
    else{
        this.controller.OpenDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1O"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR1").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1I"));
}
```

Listing A.2: The Unity script inplementing the first extension to Listing A.1.

EXTENSION TWO

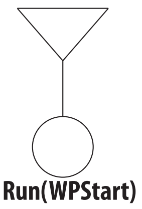
```
if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPStart").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPBar"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPBar").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPPatron"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPPatron").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR30"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR30").transform.position) && this.controller.State == State.Idle){
    this.controller.OpenDoor();
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR3"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR3").transform.position) && this.controller.State == State.Idle){
    double tmp = rand.NextDouble();
    Debug.Log(tmp);
    if (tmp < 0.5){
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR10"));
    }
    else{
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR20"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR10").transform.position) && this.controller.State == State.Idle){
    if (!this.controller.GetClosestDoor().open){
        if (!this.controller.GetClosestDoor().locked){
            this.controller.OpenDoor();
        }
        if (!this.controller.GetClosestDoor().open){
            this.controller.UnlockDoor();
            this.controller.OpenDoor();
        }
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1I"));
    }
    else{
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPStart"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR1I").transform.position) && this.controller.State == State.Idle){
    if (this.controller.GetClosestDoor().open){
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1"));
    }
    else{
        this.controller.OpenDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR10"));
    }
}
```

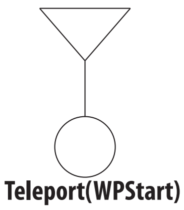
```
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR1").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR1I"));
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR2O").transform.position) && this.controller.State == State.Idle){
    if (!this.controller.GetClosestDoor().open){
        if (!this.controller.GetClosestDoor().locked){
            this.controller.OpenDoor();
        }
        if (!this.controller.GetClosestDoor().open){
            this.controller.UnlockDoor();
            this.controller.OpenDoor();
        }
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR2I"));
    }
    else{
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPStart"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR2I").transform.position) && this.controller.State == State.Idle){
    if (this.controller.GetClosestDoor().open){
        this.controller.CloseDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR2"));
    }
    else{
        this.controller.OpenDoor();
        this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR2O"));
    }
}
else if (this.controller.IsAtPosition(waypoints.First<Waypoint>(wp => wp.name == "WPR2").transform.position) && this.controller.State == State.Idle){
    this.controller.Run(waypoints.First<Waypoint>(wp => wp.name == "WPR2I"));
}
```

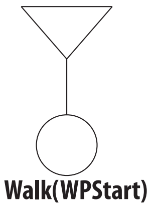
Listing A.3: The Unity script inplementing the second extension to Listing A.1.

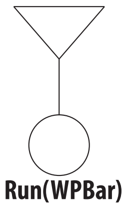
B COMPONENT LIBRARIES FOR EXPERIMENTS

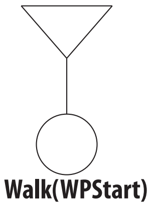
TEST 1 AND 2:

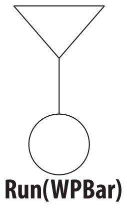
Name:	RunToWPBar
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtBar \wedge \neg AtStart \wedge RanTo\}$
Cost:	3

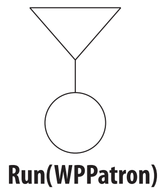
Name:	TeleportToWPBar
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtBar \wedge \neg AtStart \wedge TeleportedTo\}$
Cost:	3

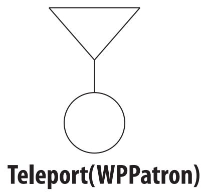
Name:	WalkToWPBar
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtBar \wedge \neg AtStart \wedge WalkedTo\}$
Cost:	3

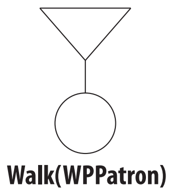
Name:	RunToWPPatron
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtPatron \wedge \neg AtBar \wedge RanTo\}$
Cost:	3

Name:	WalkToWPPatron
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtPatron \wedge \neg AtBar \wedge WalkedTo\}$
Cost:	3

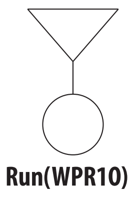
Name:	TeleportToWPPatron
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtPatron \wedge \neg AtBar \wedge TeleportedTo\}$
Cost:	3

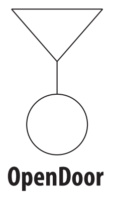
Name:	RunToWPStart
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtStart \land \neg AtPatron \land RanTo\}$
Cost:	3

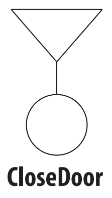
Name:	TeleportToWPStart
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtStart \land \neg AtPatron \land TeleportedTo\}$
Cost:	3

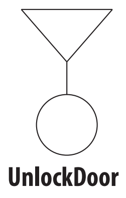
Name:	WalkToWPStart
Preconditions:	$\{\epsilon\}$
Behavior Tree:	
Effects:	$\{AtStart \land \neg AtPatron \land WalkTo\}$
Cost:	3

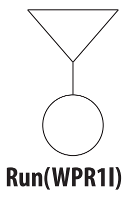
TEST 3

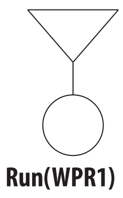
Name:	RunToWPR1O
Preconditions:	$\{AtStart\}$
Behavior Tree:	
Effects:	$\{AtR1O \wedge \neg AtStart\}$
Cost:	2

Name:	OpenDoor
Preconditions:	$\{AtR1O \wedge DoorUnlocked\}$
Behavior Tree:	
Effects:	$\{DoorOpen\}$
Cost:	1

Name:	CloseDoor
Preconditions:	$\{AtR1I\}$
Behavior Tree:	
Effects:	$\{!DoorOpen\}$
Cost:	1

Name:	UnlockDoor
Preconditions:	$\{AtR1O \wedge !DoorUnlocked\}$
Behavior Tree:	
Effects:	$\{DoorUnlocked\}$
Cost:	1

Name:	RunToWPR1I
Preconditions:	$\{AtR1O \wedge DoorOpen\}$
Behavior Tree:	
Effects:	$\{AtR1I \wedge \neg AtR1O\}$
Cost:	2

Name:	RunToWPR1
Preconditions:	$\{AtR1I \wedge !DoorOpen\}$
Behavior Tree:	
Effects:	$\{AtR1 \wedge \neg AtR1I\}$
Cost:	2

C GOAL GRAPHS FOR EXPERIMENTS

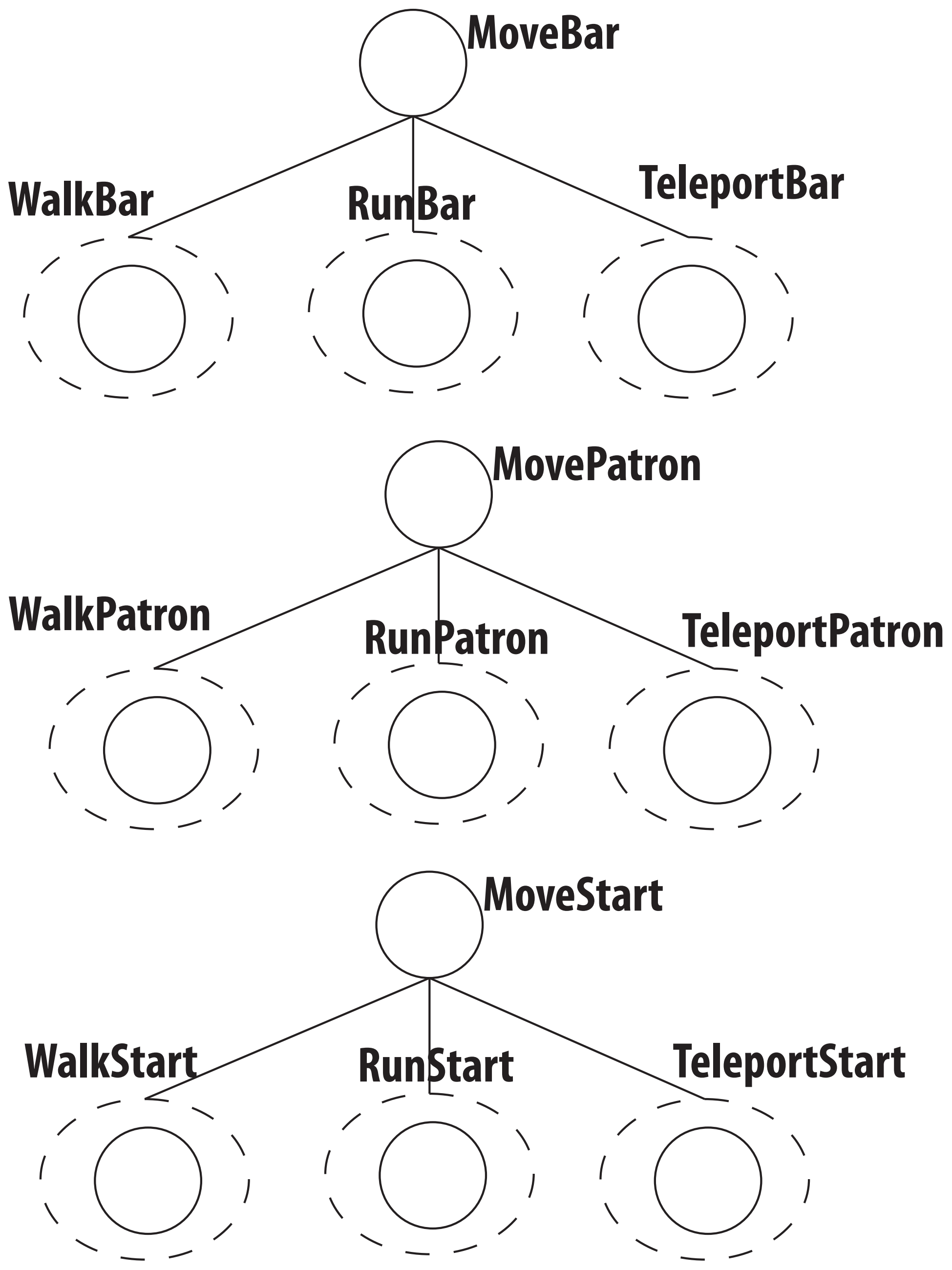


Figure C.1: Goal graph for experiment one in chapter 5.

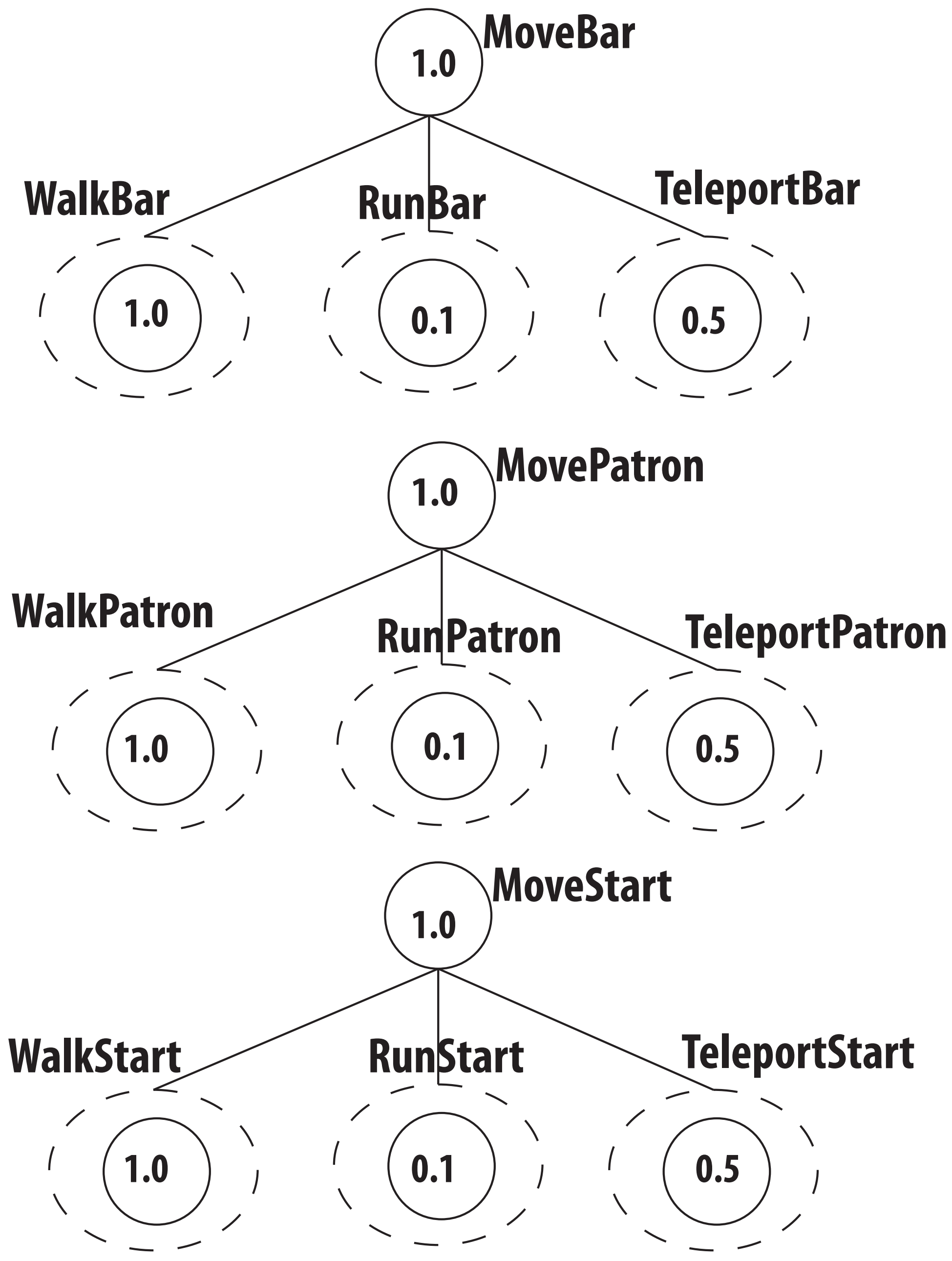


Figure C.2: Goal graph for experiment two in chapter 5.

D
SEARCH GRAPH FOR TEST 3

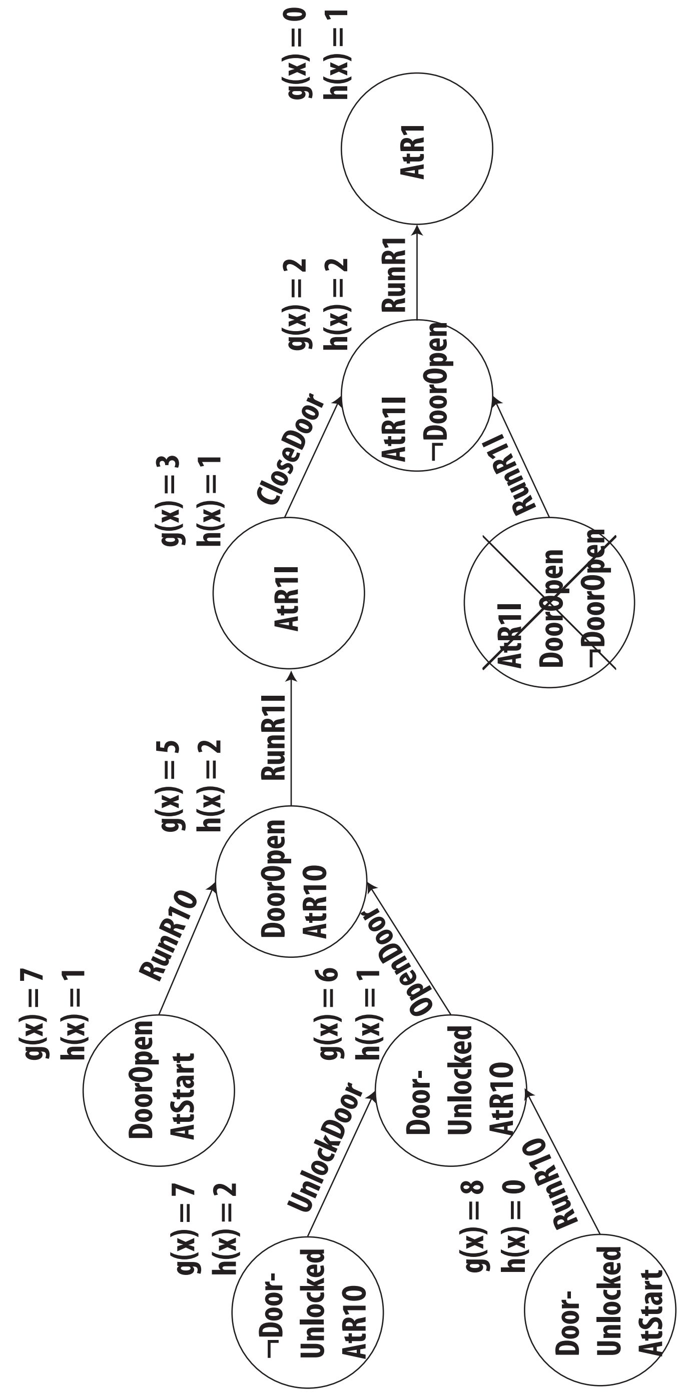


Figure D.1: Complete search graph for Test 3 in chapter 5

BIBLIOGRAPHY

[1] M. Dyckhoff. Decision Making and Knowledge Representation in Halo 3. http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/nips07.pdf (Checked: 07-06-2011). Presentation.

[2] M. Dyckhoff. Evolving halo's behaviour tree ai. http://www.bungie.net/images/Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf (Checked: 07-06-2011). Presentation.

[3] G. Flórez-Puga, M. A. Gómez-Martín, P. P. Gómez-Martín, B. Díaz-Agudo, and P. A. González-Calero. Query-Enabled Behavior Trees. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 4:298–308, 2009.

[4] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning - Theory and Practice*. Number ISBN: 1-55860-856-7 in 1st Edition. Morgan Kaufmann, 2004.

[5] P. Gorniak and I. Davis. Squadsmart: Hierarchical Planning and Coordinated Plan Execution for Squads of Characters. *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 14–19, 2007.

[6] H. Hoang, S. Lee-urban, and H. Muñoz-avila. Hierarchical plan representations for encoding strategic game ai. In *In Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press, 2005.

[7] A. T. Holm and M. Bøgeskov. AI Modelling: Behavior Trees. Master's thesis, Department of Computer Science, AAU, https://services.cs.aau.dk/public/tools/library/details.php?id=1275469049 (Checked: 07-06-2011), 2010.

[8] D. Isla. Handling Complexity in the Halo 2 AI. http://www.gamasutra.com/gdc2005/features/20050311/isla_01.shtml (Checked: 07-06-2011), 2005.

[9] J.-P. Kelly, A. Botea, and S. Koenig. Offline Planning with Heirachical Task Networks in Video Games. In *In Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-08)*. AAAI Press, 2008.

[10] S. Larsen, T. Olsen, K. Sejrsgaard-Jacobsen, L. H. Phan, and J. O. Groth. Applying Behavior Trees to Starcraft AI. Technical report, Student project Aalborg University, http://homes.student.aau.dk/jgroth07/Dat5.pdf (Checked: 07-06-2011), 2010.

[11] Mono. Compatibility - Mono. http://www.mono-project.com/Compatibility (Checked: 07-06-2011), 2010.

[12] Monolith Productions. F.E.A.R. http://www.lith.com/Games/F-E-A-R- (Checked: 07-06-2011), 2005.

[13] J. Orkin. Agent Architecture Considerations for Real-Time Planning. In *Artificial Intelligence & Interactive Digital Entertainment (AIIDE-05)*. AAAI Press, 2005.

[14] J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. Technical report, Monolith Productions / M.I.T. Media Lab, Cognitive Machines Group, http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf (Checked 07-06-2011), 2006.

[15] J. Orkin. Goal-Oriented Action Planning. http://web.media.mit.edu/~jorkin/goap.html (Checked: 07-06-2011), 2010. Collection of resources about GOAP.

[16] R. Pillosu. Coordinating Agents with Behavior Trees. http://staff.science.uva.nl/~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees.pdf (Checked: 07-06-2011). Presentation.

[17] S. Rabin. *AI Game Programming Wisdom 3*. Number ISBN: 1-58450-457-9 in 1st Edition. Charles River Media, 2006.

[18] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach*. Number ISBN: 0-13-207148-7 in Third Edition. Pearson, 2010.

[19] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–426,623–656, October 1948.

[20] Z. P. Software. Interstellar Marines. http://www.interstellarmarines.com (Checked: 07-06-2011), 2009.