PLANNING FOR NON-PLAYER CHARACTERS BY LEARNING FROM DEMONSTRATION

John Drake

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2018

Supervisor of Dissertation

_____-
Maxim Likhachev
Adjunct Assistant Professor
Computer and Information Science

Graduate Group Chairperson

_____-
Lyle H. Ungar
Professor
Computer and Information Science

Dissertation Committee

Sven Koenig, Professor of Computer Science

Norman Badler, Professor of Computer and Information Science

Kostas Daniilidis, Professor of Computer and Information Science

Jianbo Shi, Professor of Computer and Information Science

ACKNOWLEDGEMENTS

I thank my parents, Janet and Daniel, without whose encouragement and support I might never have pursued graduate school.

I thank my thesis committee members Sven, Norm, Kostas, and Jianbo. Special thanks to Norm for advising me as an undergraduate and hosting me in the SIG Lab where I began my formation as a researcher.

I thank the CG@Penn community, especially Ben, Catherine, and Aline for mentoring me on my first projects and publications.

I thank my advisors Maxim and Alla and the Search Based Planning Lab community for hosting me during my time in graduate school. Special thanks to Mike, whose work I extended, and Brian, Jon, and Sameer for their good humor.

Also, I thank my research assistants Pow, Roy, Max, and Stephen, who helped me develop my work while tending to their own projects. I hope that I was as good a mentor to all of them as they were assistants to me.

I thank the communities of Tilbury and The Oratory of St. Philip Neri in Pittsburgh for forming my person in ways that the school of engineering does not. *Deo gratias.*

Finally, I thank my wife Maria for sticking with me while I finished the doctoral program despite the student stipend and the times that I was too busy to do anything fun. I especially appreciate the late-night car rides home when I had unwisely walked to campus on days that were wet and cold.

ABSTRACT

PLANNING FOR NON-PLAYER CHARACTERS BY LEARNING FROM DEMONSTRATION

John Drake

Maxim Likhachev

In video games, state of the art non-player character (NPC) behavior generation typically depends on hard-coding NPC actions. In many game situations however, it is hard to foresee how an NPC should behave to appear intelligent or to accommodate human preferences for NPC behavior. We advocate the creation of a more flexible method to allow players (and developers) to train NPCs to execute novel behaviors which are not hard-coded. In particular, we investigate search-based planning approaches using demonstration to guide the search through high-dimensional spaces that represent the full state of the game. To this end, we developed the Training Graph heuristic, an extension of the Experience Graph heuristic, that guides a search smoothly and effectively even when a demonstration is unreachable in the search space, and ensures that more of the demonstrations are utilized to better train the NPC's behavior. To deal with variance in the initial conditions of such planning problems, we have developed heuristics in the Multi-Heuristic A* framework to adapt demonstration trace data to new problems. We evaluate our approach in the Creation Engine game engine by modifying The Elder Scrolls V: Skyrim (Skyrim) to accommodate our NPC behavior generators and experiments. In Skyrim, players are given "quests" which are composed of several objectives. NPCs in the game sometimes accompany the player on quests, but state-of-the-art companion NPC AI is not sophisticated enough to behave according to arbitrary player desires. We hope that our work will lead to the creation of trainable NPC AI. This will enable novel gameplay mechanics for video game players and may augment video game production by allowing developers to train NPCs instead of hard-coding complex behaviors.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Planning video game non-player character (NPC) behavior is a complex problem, with solutions typically depending on hard-coded components or minimization of some given objective function [1]. Moreover, there did not exist ways to train NPC tactical behavior by demonstration. Heuristic graph search techniques such as A* search [2] can be used for planning NPC tactics, however they can be computationally expensive and do not by default incorporate player preference on how NPCs should behave.

The previously-developed Experience Graph (E-Graph) [3] method allows the use of experience or demonstration data in a graph search. However, this breaks down in the context of video game quests. We introduced an alternative Training Graph (T-Graph) heuristic formulation to address these issues. Our method utilizes demonstration data that does not lie directly on the search graph and is not accessible from the search graph. Our method encourages more complete use of demonstration data than the E-Graph method does, in order to train complete tactics. Also, our heuristic can be combined with other deterministic AI behavior control methods. These features together enable demonstration-based behavior planning to generate behaviors for NPCs in situations similar to those encountered during their training.

Open-world games often present the player with quests or missions that give the player new tasks in already-visited locations. For example, the player may be asked to clear a dungeon of enemies in one quest, and then to go back to the same location in another quest to retrieve a special item. When the NPC behavior needed to accomplish a quest goal differs from behavior which was demonstrated in training, the use of the E-Graph or T-Graph heuristics can be *more* computationally costly than planning a solution without the use of any training data. It can also produce strongly sub-optimal solution behaviors, since the search prefers to use the training data, which leads it astray. To address these issues, we developed a method to use Multi-Heuristic A* Search (MHA*) [4] to adapt demonstration from one quest configuration to a new one. Our MHA* heuristics enable demonstration-based NPC behavior training to work more efficiently in environments similar to those of training, despite changes to the tasks to complete.

1.1 Motivation

We consider open-world video games such as the games in the Elder Scrolls and Borderlands series. These kinds of games often send the player and supporting NPCs on quests (or "missions") through portions of the game environment we will call "dungeons" here. Each dungeon is accessible from the greater game world, but itself only represents a limited physical area. The game's storyline presents the player with quests, and so-called "side-quests" tangential to the main storyline are also available to the player. See Figure 1.1 and Figure 1.2 for examples of quest objectives and the corresponding dungeon areas from both the Borderlands and The Elder Scrolls V: Skyrim (hereafter referred to simply as Skyrim) games.

Within these types of games, we consider in this thesis the following common scenario in a game. The player encounters a friendly NPC who winds up joining the player on a

Figure 1.1: A Borderlands game mission with three objectives. The first objective's location (waypoint) is marked with a diamond marker at the bottom of the local map.

mission. The mission is short and has a clearly specified goal, e.g. to pull a lever at the end of a small dungeon. The player and his companion begin at the door to the dungeon. There may be forks in the passageways between the door and the lever, creating different route options. There may be different actions to take before reaching the lever, for example to kill particular enemies along the way or to leave them alone. The player and NPC can also choose to move stealthily or run through quickly.

The player navigates the dungeon to get to the lever at the end (Figure 1.3). His companion NPC acts according to its programming. It often happens that this is insufficient. For example, the NPC may only know to charge ahead despite the fact that the level demands a stealthy approach. There may be a very strong enemy NPC in one part of the level

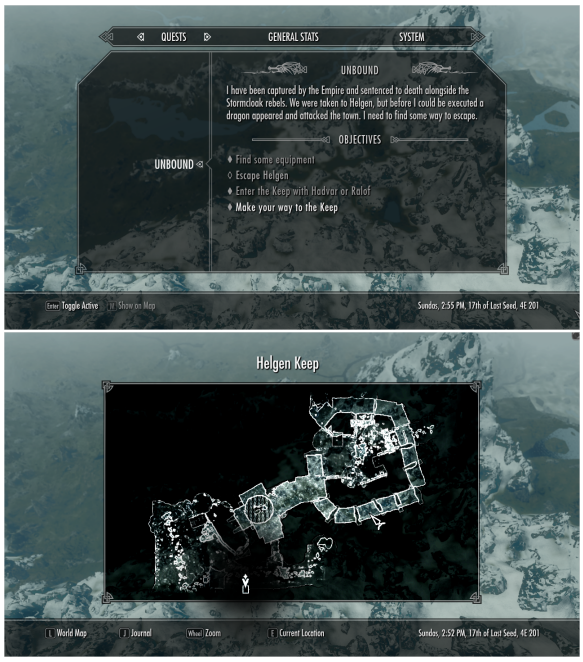

Figure 1.2: A Skyrim game quest with four objectives. The local map of the associated game area shows a quest objective marked with a **V** marker at the bottom of the view.

which could be bypassed via another route. There may be a navigational trap, such as a spear pit. It is not possible to foresee every such situation while designing the AI code for the NPC. The player may wish to instruct the NPC how to behave, and indeed many games incorporate commands which can be given to friendly NPCs, but it is often not possible to issue compound commands for complex tactics and not possible to foresee command schemes to handle every scenario. The game developer might also wish to author NPC behavior by demonstration, perhaps to augment other behavior control code for complex scenarios that leave the NPC incapable of solving them.

These quests often re-use particular dungeons at different points in the game. The player may be sent through a dungeon to retrieve a special item in one quest, but then be sent back to the same dungeon to activate a device in another quest. Parts of the dungeon may be exactly the same in a later quest as they were previously, but other important parts of the dungeon may have moved around. A key to open a door might be moved to a new

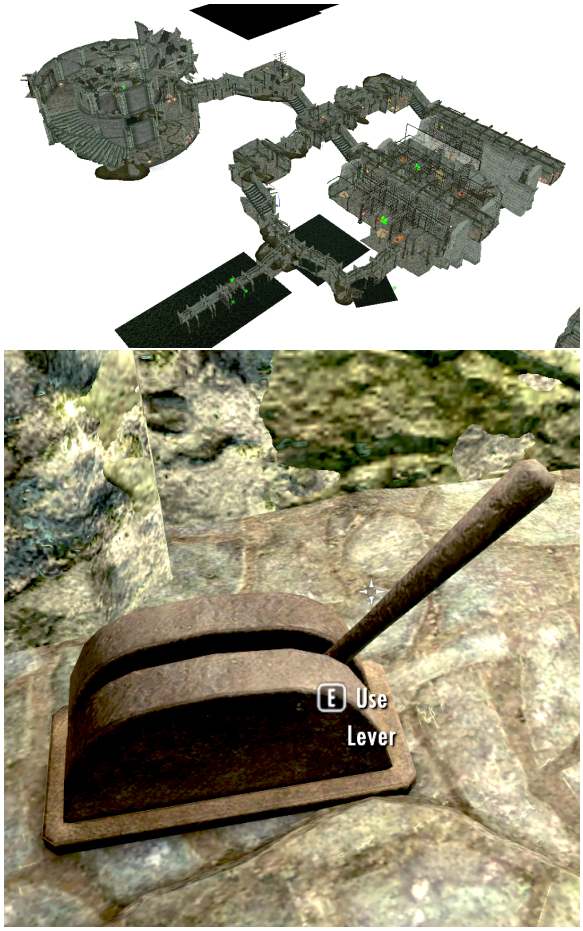

Figure 1.3: Example dungeon and lever as goal.

location, there might be additional health pack pick-ups available, and enemies may have been randomly placed in new locations. See examples of environment reuse across quests in Figure 1.4 and Figure 1.5.

It can be useful to train NPC behavior in these kinds of games, so that a companion NPC can more effectively assist the player, or so that the game developer can augment the NPC's ordinary AI capabilities with trained behavior for special circumstances. Training can be done by allowing the player (or developer) to record a trace of gameplay in a dungeon environment and then feeding this trace into a demonstration-based planning method, as we discuss in this thesis.

Figure 1.4: A Borderlands mission sharing the same environment with Figure 1.1. Note that the objective has moved.

In this chapter, we review the game Skyrim in more detail in order to illustrate the context of our work. The game's goals are presented to the player in the form of quests. Quests also serve to expose the player to the game's storyline. Skyrim's various gameplay features enable the player to achieve the goals set forth by quests, thereby advancing the game's plot. By reviewing Skyrim's quests and gameplay, we can understand what is possible in games like this, and which areas are lacking in existing NPC behavior generation schemes.

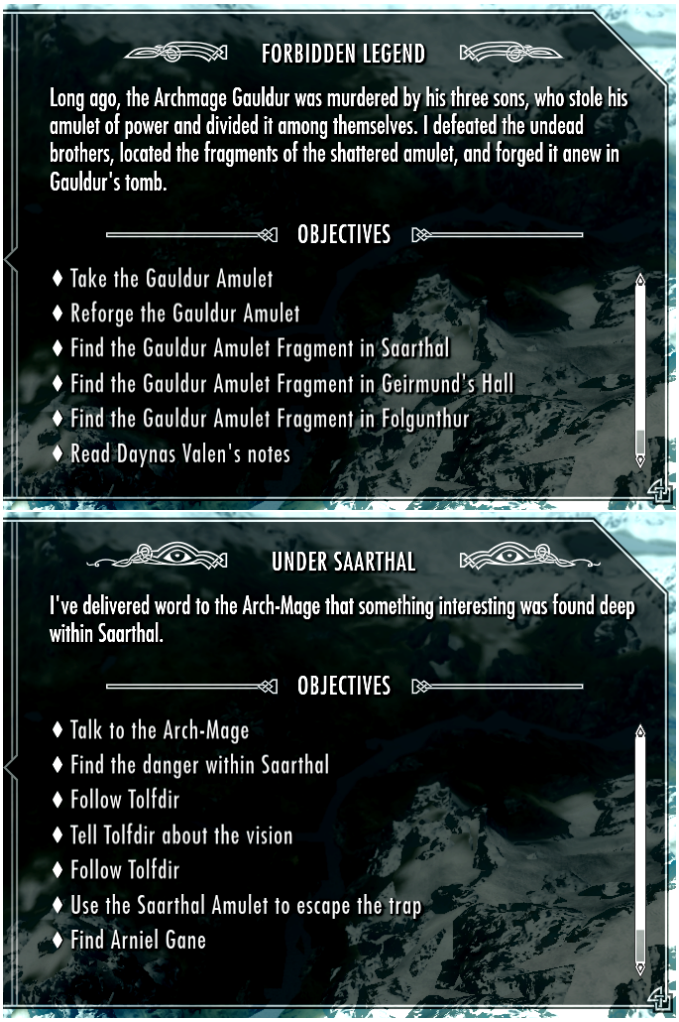

Figure 1.5: Two Skyrim quests sharing objectives in the same dungeon, Saarthal.

1.2 Quests

We wanted to make sure that our techniques worked across the most common types of quests encountered in contemporary open-world games. We examined the video game Skyrim and used its design for guidance while testing our work.

1.2.1 Main Quest Line

Open-world games typically have a primary sequence of quests which reveal the story of the game as they are completed. These quests are carefully crafted by the developer. Skyrim's main quest line starts with a quest called "Unbound," in which the player character finds himself bound as a prisoner on way to be executed at Helgen Keep. During the execution, a dragon appears in the sky above and it begins to attack the keep. The player escapes custody and eventually fights his way through the interior section of the fort, and then through an attached cave before reaching the greater part of the game world. Along the way, there are many weapons, pieces of armor, potions, and other items the player can pick up and equip. The player must fight a few opponents, including dangerous wildlife in the cave. This all serves to familiarize the player with the game.

In the third quest of Skyrim's main quest line, the player is sent into a "dungeon" location called Bleak Falls Barrow by a character named Farengar to retrieve an item called the "Dragonstone." The barrow is largely linear in layout, with traps and puzzles along the way. A special key must be acquired by killing a character within. The key is used to open the last door of the crypt. Beyond the door, the player must fight one more opponent and when this enemy is killed, the Dragonstone is found on its corpse. At this point, the exit to Bleak Falls Barrow is accessible, and the player can return the Dragonstone to Farengar.

We distill the main or final objective of each of Skyrim's main quests in Table 1.1 to illustrate that most of the game's challenges can be encoded as a simple goal. Many of these quests, such as the game's third main quest discussed above, can be decomposed into several component tasks with specific sub-goals. These individual sub-goals follow the same formats as the overarching quest goals: go to location, retrieve item, kill target opponent, etc.

Unbound	Go to location
Before the Storm	Go to location
Bleak Falls Barrow	Retrieve item
Dragon Rising	Kill target opponent
The Way of the Voice	Go to location
The Horn of Jurgen Windcaller	Retrieve item
A Blade in the Dark	Kill target opponent
Diplomatic Immunity	Talk to character
A Cornered Rat	Talk to character
Alduin's Wall	Go to location
The Throat of the World	Talk to character
Elder Knowledge	Retrieve item
Alduin's Bane	Kill target opponent
Season Unending	Talk to character
The Fallen	Talk to character
Paarthurnax	Kill target opponent
The World-Eater's Eyrie	Go to location
Sovngarde	Go to location
Dragonslayer	Kill target opponent

Table 1.1: Main Skyrim quests and their primary goal types.

1.2.2 Side Quests

Skyrim's developers have also filled the game world with ways to initiate what are called side quests. These quests are not part of the main storyline, but bring the player to new parts of the game world and offer the player opportunities to advance their skills, earn game money, and acquire new weapons and other equipment. Side quests are typically initiated by talking to an NPC, who will then ask the player to do some kind of errand for them in exchange for something valuable. Side quests generally have the same formats as main storyline quests: go to a location, retrieve an item for an NPC, or kill something for an NPC. Sometimes, side quests are open-ended or ambiguous, such as the "No Stone Unturned" quest, which tasks the player with locating 24 unusual gemstones scattered across the entire game world without giving the player any specific direction regarding where they can be found. Usually, the objectives of these quests are specific and the player is shown where to go to complete the objectives.

1.2.3 Radiant Story Quests

Skyrim has a system, called *Radiant Story* which provides the player of the game with an innumerable number of randomly-generated quests in addition to the other game quests which can only be completed once. *Radiant* quests give the player a random task to complete in a random location the player likely has not yet visited. Though randomized in this manner, these quests follow the general format of other side quests.

1.3 Gameplay

Skyrim presents players with a variety of actions that they can perform to change the game's world state. These actions comprise either navigation, combat, speech, or other miscella-

neous world interactions. Navigation actions move the player through the large open game world. Combat actions generally effect the death of opponents. Speech actions advance the storyline of the game, develop relationships between the player and NPCs, enable the sale and purchase of items from NPCs, among other purposes. Finally, there are other miscellaneous interactions the player can have with the world, such as inventory transfers between the player and storage containers, the activation of various mechanisms, crafting of items, and more. All of these at different times contribute to the completion of quest objectives and to the advancement of the player's own personal goals within the game.

1.3.1 Navigation

In Skyrim, the player is generally free to navigate on relatively flat ground. The player can jump, but only a limited height and distance. The player can choose to sneak, walk, or run. Sneaking reduces the chances of being detected by other characters or creatures in the game, but causes reduced movement speed. Various obstacles such as doors, gates, and drawbridges prevent the player from navigating certain ways until those obstacles are removed. Doors can be locked, requiring either a key or (sometimes) lockpicking skills to unlock. Sometimes an obstacle can only be moved by activating a button or by solving a puzzle. Sometimes navigation actions cannot be reversed, such as jumping down a ledge to a lower area. In deep water, the player can swim, but swimming time under water is limited before the player's character will drown. The player may acquire a special ability in game to dash forward a short distance very quickly. In our work in this thesis, we use a simplified navigational action set. We allow our NPCs in experiments to sneak and walk and include navigational obstacles such as locked doors.

1.3.2 Combat

Combat in Skyrim can be done with bare fists, bows and arrows, blades, magical items, magical spells, and other miscellaneous weapons. Successful attacks reduce the target's health level. Attacks can be blocked with the right equipment. Weapons can be poisoned for additional damage. Weapons can be enchanted to cause additional types of damage or other effects when they hit a target. Extra-powerful attacks can be made at the expense of the player's stamina reserve. Stamina returns naturally over time while resting, but can also be replenished with some types of food and potions. Stamina is drained by running. Magic attacks deplete the player's "magicka" reserve. Like stamina, this returns slowly on its own naturally, but can also be restored with consuming certain potions. Melee combat takes place in close quarters to enemies, but ranged attacks by archery or magic occur over a distance. The player can also acquire in game other special combat abilities, such as the ability to freeze an opponent solid or the ability to become invisible to enemies for a short time. In our work in this thesis, we keep things simple and only model bare fist combat by NPCs.

1.3.3 Speech

Very often in Skyrim, the player needs to talk with NPCs. This is often done during quests to advance the plot of the game's story. Conversation with NPCs is done via scripted dialogs where the player is presented with several options of things to say to the NPC, and based on which option is selected, the NPC responds with its own scripted reaction. The choices the player makes in these conversations can have significant impact on what happens next. For example, if the player is caught stealing by a town guard, the player may be able to choose to comply with the guard's orders, to intimidate the guard, or to persuade the guard to look the other way. Since speech decisions are generally only made by the player, we do

not model speech mechanics in our NPC behavior generation work.

1.3.4 World Interaction

There are many other additional ways that the player can interact with the game world. The world is scattered with such things as treasure chests, plants which can be harvested, mechanisms operated by buttons and levers, and puzzles to solve. Some artifacts in the game world, when encountered, grant the player special perks. For example, The Shadow Stone artifact enables the player to become invisible for one minute once a day. This perk is available to the player until the player chooses to replace it with another. We have not modeled most of these ancillary gameplay mechanics in our work, but given access to the source code for a game, any of these features could be incorporated.

1.4 Proposed Approach

We propose using search-based planning to enable the training of tactical behaviors for video game non-player characters (NPCs). At a high level, this kind of planner begins with a starting state in a graph representing the accessible states in a search space and the edges which validly connect them. A search algorithm finds a path through this graph to connect to a goal state. For example, imagine a video game quest where an NPC must navigate a dungeon in order to reach a goal position on the other side. The video game developer might wish that the NPC prefer to take a particular path through part of the dungeon for artistic reasons, or a video game player might wish that every time he enters the dungeon, that the NPC keep out of his way. These preferences can be provided as demonstrations. We desire that the search process will tend to re-use the demonstrations when possible, thereby producing trained behavior as output.

We build on the work done on Experience Graphs[3], using a special heuristic to guide

the search process to re-use experience or demonstration[5] data while finding a solution to the problem at hand. To address particular challenges in the NPC behavior planning domain, we modify the Experience Graph approach into our Training Graph method. We also apply a Multi-Heuristic A* algorithm[4] to our problem domain, affording us the opportunity to use additional special heuristics (based on our Training Graph heuristic) which helps adapt to new problems.

1.5 Evaluation

We evaluated our *Training-Graph* and *Multi-Heuristic A* With Training-Graph Heuristics* methods with experiments in both a simulated game environment and in the context of the video game The Elder Scrolls V: Skyrim.

In simulation, we authored many test scenarios mimicking video game quests and environments with various types of constraints and types of goals. We model combat with enemies, shields for protection in combat, keys, locked doors, and walls and other obstacles. Goal types include *kill a target opponent* and *navigate to a target position*. We extracted the navigation meshes from several environments from Skyrim in order to test both our own contrived scenarios and real in-game challenges. We coupled our system with Skyrim so that NPC behavior can be planned by our methods and then executed in-game.

We investigated the use of skill rating system TrueSkill[6] to compute a notion of the *skill* of an NPC behavior planner by treating planning episodes as competitions between the algorithm and the problem. This allows algorithms to be compared to each other in terms of skill and for problems to be compared to each other in terms of difficulty. Furthermore, we show that hypothetical algorithm-problem pairs can be evaluated using TrueSkill ratings for a prediction of whether the algorithm would succeed in solving the problem.

We implemented a small user study to investigate the perceived qualities of our planned

NPC behaviors. Since the ultimate goal is to generate behaviors for NPCs in video games operating beside human players, it is important to understand what humans think about the behavior of these NPCs.

1.6 Contributions

This thesis makes the following contributions:

- The Training Graph heuristic, adapting the Experience Graph approach to the context of tactical NPC behavior planning. We discuss the theoretical properties of this new heuristic and compare its performance to other methods.

- The MHA* With Training Graph Heuristics approach to NPC behavior planning, which utilizes our Training Graph heuristic in a special way to further adapt training data to new situations.

- A framework to apply NPC behavior planning techniques to a real video game, The Elder Scrolls V: Skyrim. We implemented our methods in this framework and used it for parts of our analysis.

- An experimental evaluation of planning with our methods. We compute computation performance figures for our methods across several representative planning problems. We use a rating system called TrueSkill to compute a notion of algorithm *skill* at solving those planning problems. We analyze the quality of planned NPC behaviors and demonstrate that they succeed in mimicking training input. Finally, we conducted a small human subject study to test the perceived qualities of our planned NPC behaviors.

1.7 Outline

This thesis is arranged as follows:

- Chapter 2 reviews related research for generating NPC behavior and planning from experience or demonstration.
- Chapter 3 explains in more detail the particular antecedents of our work: A*, Weighted A*, the Experience Graph heuristic, and Multi Heuristic A*.
- Chapter 4 describes and analyzes our first contribution, the Training Graph heuristic and our framework for evaluating it in a real video game: Skyrim.
- Chapter 5 describes and analyzes our second contribution, the use of Training Graph heuristics within Multi Heuristic A* to adapt NPC training to new scenarios.
- Chapter 6 summarizes work completed in [7] which extends our systems to include player modeling and implements cooperative NPC behavior planning from demonstration.
- Chapter 7 details our experimental analysis of the T-Graph and MHA* With T-Graph Heuristics approaches from qualitative and quantitative perspectives.
- Chapter 8 concludes our thesis with discussion, avenues for future work, and a summary of contributions.

Chapter 2

Related Work

Our work focuses on the use of demonstrations to train the behavior of NPCs in video games. We review several state of the art methods for generating NPC behavior in games. Since we accomplish NPC behavior generation in our work using heuristic search-based planning techniques, we review several heuristic search algorithms. For comparison, we also touch on related sampling-based and optimization-based approaches. Finally, we review existing work on planning from experience or demonstration, since our goal is to create a system for training NPC behavior.

2.1 NPC Behavior Generation

NPC behavior planning is typically accomplished via hard-coded methods (such as rule-based systems, finite state machines [8], or behavior trees [9]) or by minimizing some given objective function, as in planning approaches (such as POMCoP [10] or GOAP[11]).

Rule-Based systems (RBS) operate on a set of rules used to govern AI behavior. These are generally the most basic form an artificial intelligence system can take[12], but are also the most popular form of AI found in games [13]. For example, an AI blackjack dealer

might always *hit* when the cards dealt add up to 17 or less. The enemy characters in Pac-Man are also controlled with a Rule-Based System. One ghost enemy always turns left, one always turns right, one turns randomly, and one turns toward the player. Individually, the rule controlling each ghost would be easy to figure out, however the group of four ghosts together presents a confusing challenge to the human player.

Finite State Machines (FSMs) [14] offer a more sophisticated approach to video game NPC control. FSMs model agents which go through a set of distinct states during gameplay. For example, a guard NPC might ordinarily be in an *at post* state, and when he hears a noise switches to a *searching* state. The *searching* state could transition back to the *at post* state, or if the cause of the noise is discovered, the FSM could transition to another state such as *pursuit*. The NPC executes different specific behaviors in each state, and there is logic to control transitions between states. The behavior executed at a particular state can be coded with e.g. a rule-based system, or any other method. FSMs are appealing for game development because they are easy to understand and efficient to execute. Manually authoring detailed FSMs for the diverse multitude of NPCs in modern video games is a cumbersome task. Work [8] done in 2013 presents a method to automatically generate NPC behavior-control FSMs from the results of nondeterministic planning episodes.

A Behavior Tree[9] is a tree controlling the behavior of an NPC. Nodes either make an NPC perform an action (generally leaf nodes) or control the visitation of other nodes (generally nodes with children). Behavior is produced by traversing the behavior tree from the root node, following the direction of parent nodes. For an example, see Figure 2.1. A sequence parent node (e.g. the root of our example) chains the actions of its children in order to produce a complex behavior composed of multiple parts. A selector parent node determines which one of its children is visited depending on the results of some computation. The example behavior tree in Figure 2.1 encodes the logic and actions of going to a door, determining if it is locked, unlocking it if it is, opening the door, and then

walking through. Components like this are combined into a larger behavior tree capturing all of the desired behaviors for an NPC. Work in [15] investigates parameterization of behavior trees, encapsulating subtrees for better reuse. Behavior Trees are efficient, easy to understand, and have been used to great effect in video games such as those in the Halo series [9, 16] and Spore[17].

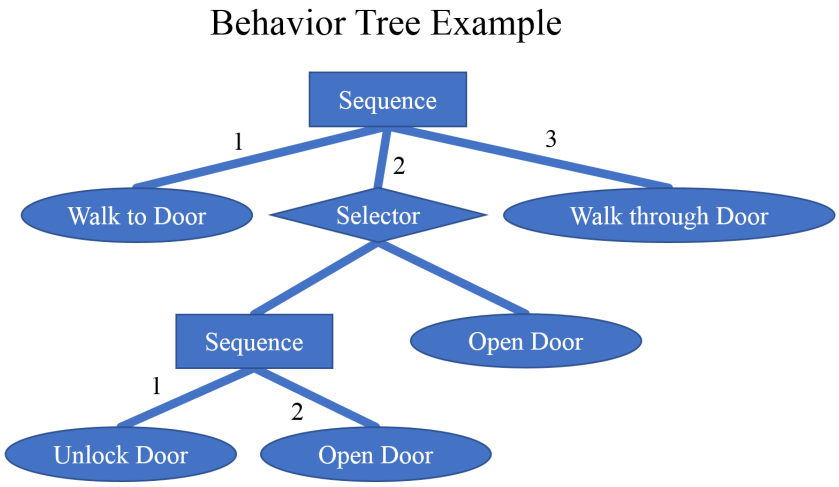

Figure 2.1: Example of a behavior tree for generating NPC behavior.

Search-based planning techniques are also used to generate NPC behavior in games. Games very often used A* graph search to plan navigation behavior for NPC agents, even when at another level of control, the agent is controlled with something like a Behavior Tree. In our earlier behavior tree example (Figure 2.1), the *Walk to Door* action might be accomplished with the help of A* for path planning. Commonly used to facilitate search-based path planning in video games is the navigation mesh (navmesh)[18] structure. A navmesh is a mesh structure which encodes the navigable portions of a game environment. An NPC navigation path to a desired goal can be planned on the navmesh and then the paths can be followed by the NPC in the game to reach the goal. Methods also exist to plan more than just the navigational component of NPC behaviors.

Goal-Oriented Action Planning (GOAP)[11, 19] is an NPC behavior planning approach

inspired by STRIPS [20] classical planning and similar methods. GOAP uses abstractions of game elements and of the state space to produce a graph representing potential NPC action choices and their high-level effects. A search through the graph is executed to generate an NPC behavior plan, which can then be executed by the NPC in the video game. Low-level behavior in GOAP is controlled by a simple three-state FSM. Each NPC can be in one of these states: Go To (controlling low-level navigation toward a specific goal location), Animate (enabling the NPC to act out specific movements like *shoot at player*), and Use Smart Object, which is just a data-driven version of Animate. Master's thesis work in [21] compared the performance of GOAP with FSMs in the context of NPC behavior generation, finding that although GOAP was associated with more overhead in creation and at runtime, the benefits to NPC competitive advantage when controlled by GOAP outweighed the costs. GOAP is the foremost planning approach used in video games In our work, we choose not to rely on abstractions from the model representing the mechanics of the game and we do not use a low-level FSM. Thus, we have to search a much higher-dimensional search space, but our plans are precise and detailed behavior plans for the NPC to carry out. Our solution plans do not have to rely on special controllers to execute each segment of a behavior plan. Moreover, our work can produce NPC behavior plans which follow training examples, but GOAP alone does not. We suspect that our work could be adapted to other graph search techniques like GOAP for a combined system generating trained behaviors.

In [22], the authors explore the use of a combined action representation for both plan recognition and planning so that an NPC can both figure out what the player is trying to do and generate plans to assist the player cooperatively. The system described builds on classical planning approaches and so it operates at an abstract level and in a limited domain, however the authors argue that it would scale to much more complex domains.

Recent work on tactical behavior planning by N. Sturtevant [23] proposed a method which takes into account relationships between NPCs while planning paths. The proposed

method focuses on path planning alone and decomposes NPC interaction into parameters which, in sum with distance traveled, make up the cost function of the search. Our work in this thesis plans for behavior actions beyond only navigation and provides an avenue for player preference to affect tactical planning. Our methods could be combined with cost formulations such as this one to yield a combined system.

Research by Uriarte and Ontanon published in 2016 on the use of Monte-Carlo Tree Search (MCTS) in "real-time strategy" (RTS) games [24] processes experience traces of past gameplay to inform a policy used to plan new gameplay decisions. RTS games require the simultaneous control of many combat units and other gameplay elements, whereas our work is focused on shooter or role playing games, where the player would only interact with a small number of NPCs at any one time. Moreover, RTS games involve unpredictable opponents, so it is difficult to model the behavior of the enemy through the rest of the game. MCTS is well suited to that situation, however we assume that we have an accurate world model which we can exploit in planning for better results.

2.2 Heuristic Search

Heuristic search finds a path on a graph from a start state to a goal state. Heuristic search techniques such as A*[2] build on Dijkstra's algorithm[25], but incorporate the use of *heuristics* to guide the search process. A* uses a heuristic to focus the search toward the goal state. For many kinds of search spaces and problems, this permits the search to explore many fewer states than Dijkstra's algorithm does. In order to apply graph search to the NPC behavior planning domain, the space of game world states is discretized into an implicit graph with NPC actions connecting neighboring states. Thus, these searches are "resolution-complete," meaning that if a path exists in the discretized search space, it will be found. However, the discretization process might turn a problem which is solvable in

continuous space into an impossible problem on the search graph. A*, like Dijkstra's algorithm, yields an optimal solution (if a solution is found at all). A more detailed description of A* search is provided in Section 3.1.

Weighted A*[26] is a heuristic search technique which enables the heuristic function to bias the search toward the goal more than it does in ordinary A*. In practice, this yields many fewer state expansions and much shorter computation times than A*. Solutions computed with Weighted A* are not guaranteed to be optimal in cost, however the suboptimality of Weighted A* solutions is bounded by a user-chosen multiple of the optimal solution cost. More discussion on Weighted A* is available in Section 3.2.

Other algorithms used for heuristic search come with additional special properties. *Anytime planners*, such as ARA*[27], produce an initial suboptimal solution quickly, but given additional search time yield better-quality solutions, ultimately yielding the optimal solution. *Incremental planners*, such as D*[28] and D* Lite[29] allow for efficient re-planning after changes occur in the search space. These methods attempt to reuse the search tree from previous planning episodes and repair it to account for changes to the graph. AD*[30] is an example of an anytime incremental planner.

Independent Multi-Heuristic A*, Shared Multi-Heuristic A*, and Improved Multi-Heuristic A* are examples of heuristic search techniques which utilize multiple different heuristics to guide the search process. The one heuristic used in many heuristic search planners such as A* and Weighted A* must be *admissible* and *consistent* in order for the algorithms to yield optimal or bounded-suboptimal solutions. Multiple heuristic planners also permit the use of other inadmissible and/or inconsistent heuristics to guide the search out of particularly challenging portions of the search space without affecting the guarantees of the search. A more detailed description of MHA* techniques is presented in Section 3.4.

2.3 Sampling-Based Approaches

A sampling-based motion planner generates a path by finding valid edges to connect a sampling of states in a search space from start to goal[31]. In practice, sampling-based methods can solve high-dimensional planning problems very efficiently[32]. These methods tend to provide a probabilistic guarantee on completeness, meaning that as the number of samples increases toward infinity, if there is a solution to the problem, a solution is more likely to be found.

The Rapidly-exploring Random Trees (RRT)[33] algorithm iteratively grows a tree in a search space rooted at the start state. In each iteration, a random state in the space is chosen. If the state is invalid (for example, in the domain of robot motion planning, if the configuration at the state is outside of joint limits, makes the body collide with an obstacle, etc.), a new random configuration x_{rand} is selected until it is valid. Then, the nearest tree node to that configuration, x_{near} is found. If x_{near} and x_{rand} can be connected with an edge, x_{rand} becomes x_{new} (a new node for the tree) and is added to the tree with that edge. This process is repeated for as many iterations as it takes to reach the goal state.

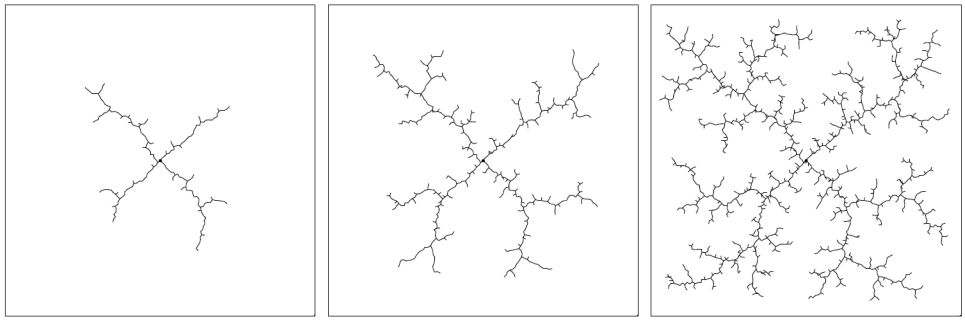

Figure 2.2: RRT exploration of a square 2D space[33]

In this simple form of RRT, the outward growth of the tree can be limited by projecting x_{rand} toward x_{near} to some chosen maximum growth rate distance (before it becomes x_{new}) if x_{rand} is initially farther from x_{near} than that amount. Then, new nodes are never more

than the maximum growth distance from the existing tree. Alternatively, and as proposed in the original paper, a set of valid state transition "inputs", U, can describe possible transitions outward from some configuration. Then, the input u which could bring x_{near} closest to x_{rand} is chosen and the tree is grown by u from x_{near} to its destination x_{new}. U can encode "non-holonomic" movement constraints on the robot, such as minimum turning radius. Pseudocode for this original form of RRT is shown in Algorithm 1.

Algorithm 1: GENERATE_RRT(x_{init},K)

```
τ.init(x_init);                                   // τ is the tree
for k = 1 to K do
    x_rand ← RANDOM_STATE();
    x_near ← NEAREST_NEIGHBOR(x_rand,τ);
    u ← SELECT_INPUT(x_rand,x_near);      // Finds best u to reach
      x_rand
    x_new ← NEW_STATE(x_near,u);      // Evaluates u to create new
      state
    τ.add_vertex(x_new);
    τ.add_edge(x_near,x_new,u);
return τ;
```

RRT-Connect Explanation

RRT-Connect[34] works by growing two RRT trees simultaneously. One tree is rooted at the start state, as in RRT, and the other tree is rooted at the goal state. When the trees meet one another, a continuous path is formed from start to goal. Moreover, the algorithm attempts to grow the trees greedily toward one another instead of always toward any random configuration as in RRT.

The Probabilistic Roadmap approach (PRM) [35] operates in two phases: construction and query. A roadmap structure is constructed by uniformly sampling a search space. Invalid samples are discarded (e.g. a state might be invalid because it is inside an obstacle) and connections to neighboring samples are created when possible (e.g. an edge between

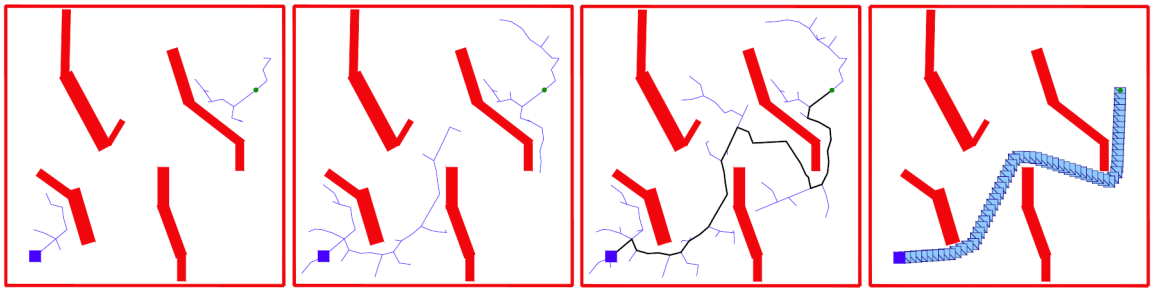

Figure 2.3: RRT-Connect planning process in a square 2D space with obstacles[34]

two states intersecting an obstacle is not valid, and so that connection is not included). This process yields a graph approximating the accessible portions of the search space. In the query phase, start and goal nodes are added to the roadmap graph with valid connections, and then a graph search technique, such as Dijkstra's algorithm or A*, is used to find a path connecting the start and goal states. Extensions to PRM include biasing the sampling of states toward the perimeters of obstacles [36] (see also Figure 2.4) and delaying expensive collision checking operations until run-time when portions of the graph are actually used[37].

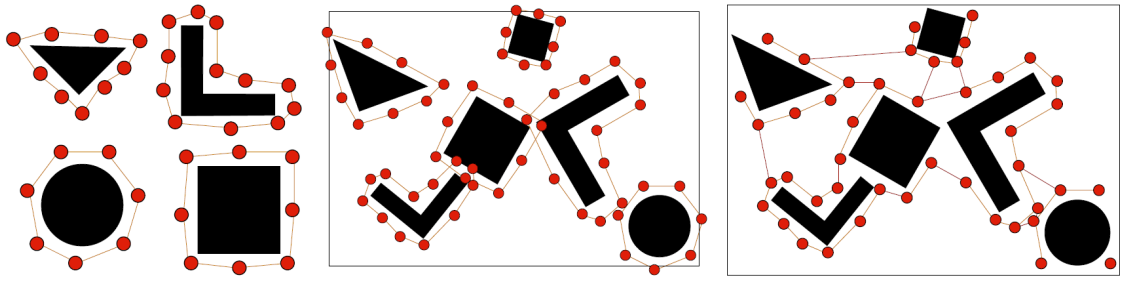

Figure 2.4: ReUse-Based PRM (RU-PRM) constructed from obstacle perimeter samples[38]

In loosely constrained situations, a sampling-based planner can rapidly find valid paths through the space. However, in highly constrained situations (e.g. a narrow passageway which must be used to get from start to goal), it can be difficult for a sampling-based planner to discover a valid path through the graph[31]. The NPC behavior planning domain

unfortunately does contain many of these narrow passageways in portions of the search space. For example, consider a scenario where a key is needed to open a door and the action to pick up a key can only be executed from states within a short distance of the key. The high dimensionality and large size of the search space makes it is unlikely that random samples will both place the NPC near to the key and also then find another nearby state holding the key to connect with a *pick up the key* action. A scheme to bias samples near to these kinds of regions might help, but heuristic search methods can be guided with good heuristics and also provide guarantees on solution quality.

In sampling-based approaches, the distribution of samples and the edges selected affect the quality of the path. As noted in [32], sampling-based planners tend to generate solutions which look random. Paths zig-zag through open portions of the search space to meet random samples along the way. Sampling-based output behaviors must be smoothed as a post-process, but heuristic search techniques naturally produce smooth goal-directed output. Though unusual motions may be acceptable in e.g. robotics applications, the perceived quality of NPC behavior output is important because NPCs are designed to interact with human players.

2.4 Optimization Approaches

Optimization approaches have been used in the robotics domain to generate robot trajectories. A good example is CHOMP (Covariant Hamiltonian Optimization for Motion Planning[39]). CHOMP typically starts with a straight line path from start to goal and iteratively optimizes the path out of and away from obstacles according to a cost function. Path smoothness is achieved with a soft constraint limiting the distance between adjacent path vertexes. Other optimization techniques operate in a similar fashion. In practice, optimization techniques tend to converge on solutions in particular local minima highly

dependent on the initial trajectory provided before optimization begins[32]. Our NPC behavior generation problem is described by a largely non-continuous space with discrete state parameters, so while these optimization methods may work for trajectory generation, they are a poor fit to our domain.

2.5 Planning from Experience and Demonstration

Like the MCTS method discussed in Section 2.1, methods in robotics research have also incorporated demonstration data in order to *learn a policy* to use in planning. For example, Inverse Optimal Control [40] has been used to learn a cost function from demonstration for a planning problem. In our work presented in this thesis, we leave the game's standard cost function intact and instead guide the search by altering search heuristics. A 2009 research survey [41] collects some other methods for learning robot control policies from demonstration.

There are existing methods which utilize demonstration and experience to *guide* the planning process for better performance. One example of this is the Lightning [42] method, an extension of Rapidly-exploring Random Trees (RRT) [33]. Lightning uses solution plans from past experience to try to reduce the time it takes to find a new solution. It retrieves saved paths from a path library and then repairs them to match the start, goal, and obstacles in the current search problem. In many cases, this process can complete faster than planning from scratch, reusing experience and saving computation time.

Another method from robotics, the Experience Graph (E-Graph) heuristic [5], is an extension from A* heuristic graph search [2]. A* uses a heuristic estimate of the transition cost between nodes to constrain the amount of the search graph which needs to be examined before finding a solution to the problem. The E-Graph algorithm extends A* search to reuse the results of previous searches and/or demonstration data. It does it in a way that

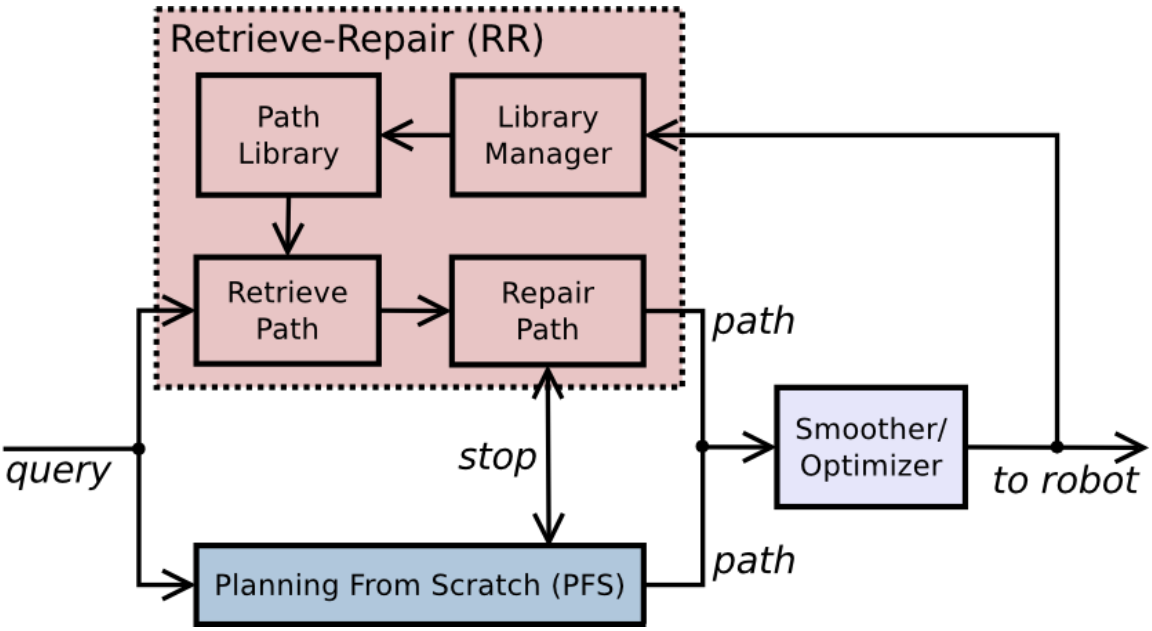

Figure 2.5: Lightning system diagram, from [42]

provides provable bounds on the sub-optimality of the solution. The E-Graph heuristic is described in more depth in Section 3.3. The standard formulation of E-Graph search, assumes that the E-Graph is reachable from the search graph and traversable along its length. Unfortunately, in video game contexts, demonstrations may not be accessible from the search graph, because NPC behavior is often more constrained than the behavior actions available to the demonstrator. For instance, a game player may demonstrate a behavior while jumping on top of obstacles, but the NPC may be constrained only to navigate on the game's navigation mesh (for information on navigation meshes, see [18]) without jumping, as in Skyrim.

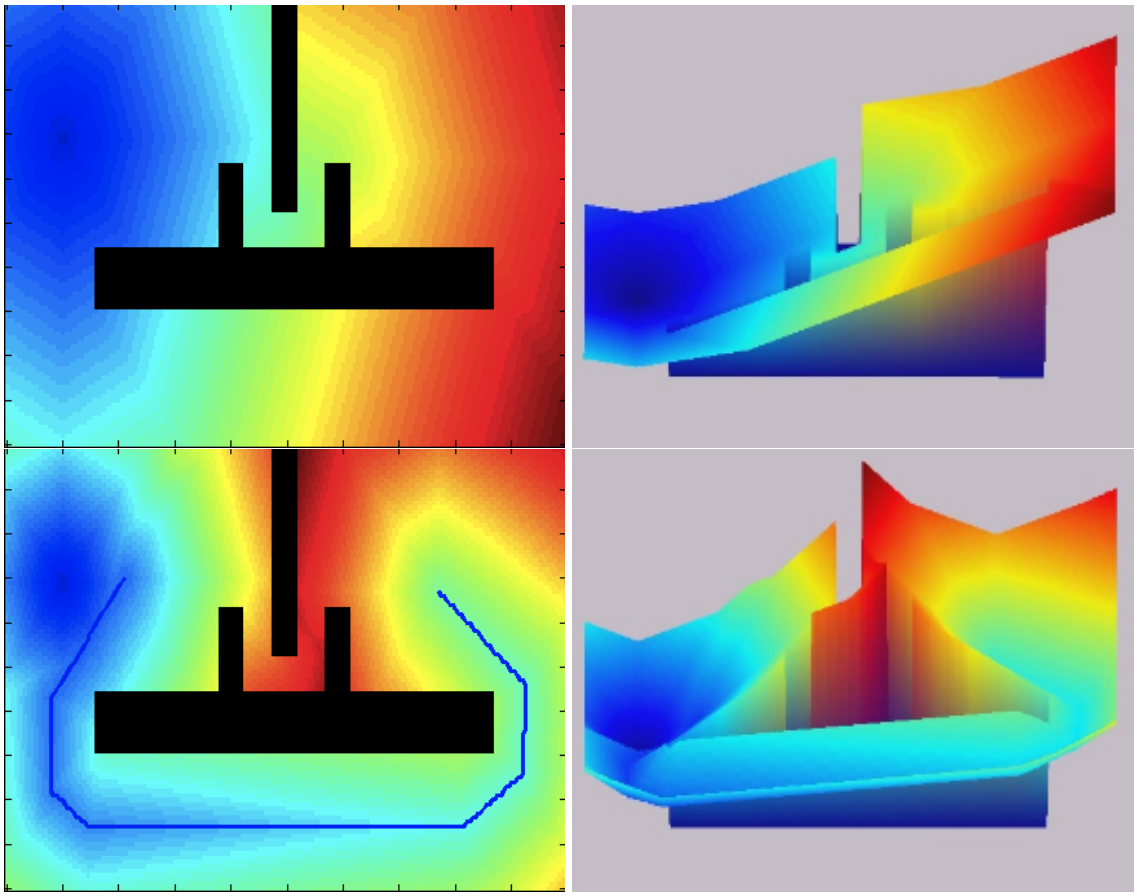

Figure 2.6: This figure shows the obstacles and heuristic values for an (x, y, θ) nonholonomic constraint navigation problem with a heuristic calculated on a down-projected (x, y) graph. The robot is not able to navigate the U-shaped corridor.
Top Row: regular 2D Dijkstra heuristic, oblique height map view.
Bottom Row: example E-Graph (blue line) and corresponding h^E, oblique height map view. The E-Graph heuristic forms a valley bypassing the un-navigable corridor. This leads the search away from that local minima, saving computation time.
Images courtesy of E-Graph authors[3]

Chapter 3

Background

3.1 A*

A* [2] is a widely used heuristic graph search algorithm which improves on the performance of Dijkstra's Algorithm[25] with the use of a "heuristic". The heuristic in an A* search is an estimation of the minimal cost of traveling from some graph node to the goal. In order for A* search to yield a minimum-cost path, the heuristic must never overestimate the minimal cost from some node to the goal. For example, Euclidean distance is a common heuristic for simple 2D or 3D navigation problems.

A* Explanation

The A* algorithm[2] explores outward from the start node, bookkeeping information at each node along the way to guarantee that a minimal cost route is found, and eventually encounters the goal, if there is some path from start to goal. For bookkeeping, the algorithm keeps a set of nodes called the "closed set" which have been wholly evaluated, an "open set" of nodes yet to be evaluated, a "g-value" at each explored node representing the cost from the start to the node along the yet best known path, and an "f-value" at each node

storing the g-value at that node added to the heuristic estimate from the node to the goal.

Pseudocode for A* is available in Algorithm 2, and it is explained here in greater detail. At the beginning, the start node is given a zero g-value and its f-value is the heuristic estimate from start to goal. The start node is put in the open set. Until the open set is empty, the following exploration and update procedure is executed repeatedly.

The node in the open set with the lowest f-value is selected as the current node to explore from and it is moved from the open set to the closed set because by the time this iteration is over, its connections to its neighbors will all be entirely explored. If the current node is the goal node, then the exploration process is finished. If it is not the goal node, then each of the neighbors (often called "successors") is evaluated as follows.

If the neighbor is already in the closed set, that neighbor is skipped since it is already entirely evaluated [1]. Otherwise, a tentative g-value (remember, the lowest yet known path cost from the start to some node) for the neighbor is calculated by adding the current node's g-value to the cost it would take to travel from the current node to the neighbor. If the neighbor is not already in the open set, then it has not been explored yet and needs to be added to the open set and given some g-value and f-value, so it is added to the open set and receives the tentative g-value as its g-value and its f-value as the tentative g-value added to the heuristic estimate from the neighbor to the goal. If the neighbor is already in the open set, that means that the node has already been given some g-value and f-value via a different direction than from the current node. So in this case, there is an opportunity to update its g-value and f-value if it happens that the tentative g-value from the current node is lower than the already stored g-value in the neighbor. If this is the case (like in the case

[1] The point of this neighbor examination procedure is to update the neighbor's ideas of which is the lowest cost path from the start to the neighbor (the g-values and f-values). It might seem bad to ignore closed set nodes because the current node might be a node on a better path from the start to the neighbor. However, because nodes are selected from the open set according to lowest f-value (and because the heuristic is a consistent underestimate), when a node n is thus selected and moved into the closed-set, there is no possible way another open set node could be part of a different path with a lower cost from start to n

it was not previously in the open set), the neighbor is given the tentative (lower) g-value as its g-value and its f-value is set to the tentative g-value plus the heuristic estimate from neighbor to goal. This completes the exploration loop.

If the open set is exhausted before the goal state is encountered, there is no path from the start to goal in the current graph. If the goal was encountered, then the least-cost path is found by backtracking through the graph from goal to start, following all edges to nodes with the lowest g-values. $RECONSTRUCT\_PATH(a, b)$ in Algorithm 2 and Algorithm 4 operates this way.

Alternatively, "back pointers" can be kept during graph exploration which always point back from a node to its neighbor which is closest to the start (by g-value). These back pointers are set in the neighbor update step, after the tentative g-value is calculated and either the neighbor is found not to be in the open set or has a large g-value which needs to be updated. In these cases, we know that the current node is currently the best way to get from the neighbor back to the goal (and this can change during the search if a better path is found), so the back pointer from the neighbor is set to the current node. If this is done, path reconstruction is done by following back pointers from the goal to the start. The $bp()$ values in Algorithm 3 facilitate this approach to path reconstruction.

3.2 Weighted A*

Weighted A*[26, 43] works much like A*, but with one change. A parameter, $w \geq 1$, is introduced. When the heuristic is calculated and used in the algorithm, it is multiplied by w. In Algorithm 2, this happens on lines 4 and 17, where $h(..., ...)$ would be replaced with $w \cdot h(..., ...)$. Thus, when $w = 1$, it is equivalent to ordinary A* search. If $w > 1$, this tends to bias the search more toward the goal (because the seemingly-bad paths come to seem even worse), so in most scenarios less exploration is needed to find some path from start to

Algorithm 2: A_STAR($start$,$goal$,h,c)

Input: start and goal nodes, heuristic function h, and cost(distance) function c
Output: Path from start to goal, or EMPTYPATH if no path exists from start to goal

```
1  closedset = ∅;
2  openset = {start};
3  start.gvalue = 0;
4  start.fvalue = start.gvalue + h(start, goal);
   // Explore graph
5  while openset is not empty do
6      current = node in openset with lowest f-value;
7      if current == goal then
8          return RECONSTRUCT_PATH(goal,start);
9      openset.remove(current);
10     closedset.add(current);
       // Expand neighbors
11     foreach neighbor ∈ current do
12         if neighbor ∈ closedset then
13             continue;                          // Skip closed nodes
14         tentative_g_value = current + c(current, neighbor);
15         if neighbor ∉ closedset ∨ tentative_g_value < neighbor.gvalue then
16             neighbor.gvalue = tentative_g_value;
17             neighbor.fvalue = neighbor.gvalue + h(neighbor, goal);
18             if neighbor ∉ openset then
19                 openset = openset ∪ {neighbor}
20 return EMPTYPATH;                    // No path from start to goal
```

goal. However, when a node is selected and moved to the closed set, it could be that it is being closed too soon and its current g-value might not get a crucial (smaller) update later. So while Weighted-A* runs faster than A*, it does not produce an optimal solution. The suboptimality of the solution is bounded by a factor of w[43]. This means that the solution has a cost at worst w times the cost of the optimal solution.

3.3 Experience Graph Heuristic

The E-Graph algorithm[3] introduces a method to guide a graph search to use trajectory data encoded in an E-Graph to help the search avoid local minima. The E-Graph can be created from the solutions of prior planning problems in the same domain, or it can be created by demonstration [5]. If necessary, new edges are added to the search graph to connect it to the E-Graph. The E-Graph heuristic h^E is used to bias the search toward reuse of the E-Graph edges while searching for a solution.

E-Graph E is a directed graph with nodes E^N encoding search state information. Successor function $succ(a) = b$ records valid transitions between states a and b as observed in the experience with an associated cost function $c^E(a, b)$ which records the observed costs of these transitions. The experience graph heuristic $h^E(a, b)$ between nodes a, b on the search graph can be computed in a general way as follows. Note that node b is typically the goal state in a search problem, so $h^E(a)$ can be understood as shorthand for $h^E(a, goal)$. Let h^G be some heuristic on the search graph which is admissible and consistent.

1. The E-Graph is augmented with virtual edges from every E-Graph node s to every other E-Graph node s' at their known E-Graph cost $c^E(s, s')$ if the transition already existed in the E-Graph (if $s' \in succ(s)$) or at cost $\epsilon^E h^G(s, s')$ otherwise.

2. The E-Graph is further augmented with virtual edges from every E-Graph node s to b at cost $\epsilon^E h^G(s, b)$.

3. Dijkstra's algorithm is run on the augmented graph from b as the source. The Dijkstra output distances are used as $h^E(s, b)$ for all nodes $s \in E^N$.

4. Finally, $h^E(s) = \min_{s' \in \{E^N \cup s_{goal}\}} [\epsilon^E h^G(s, s') + h^E(s', s_{goal})]$ That is, the smallest of the direct path $\epsilon^E h^G(s, goal)$ and every sum $\epsilon^E h^G(s, s') + h^E(s', goal)$ among all $s' \in E^N$ is selected as $h^E(s, goal)$.

In a more general way, the E-Graph heuristic $h^E(s_0, s_{goal})$ can also be defined as in Equation 3.1.

$$\min_{\pi} \sum_{i=0}^{N-1} min \left\{ \epsilon^E h^G(s_i, s_{i+1}), c^E(s_i, s_{i+1}) \right\} \tag{3.1}$$

In Equation 3.1, π is a path $\langle s_0 ... s_{N-1} \rangle$ and s_{N-1} is s_{goal}. $c^E(..., ...)$ returns the ordinary cost $cost(..., ...)$ if its inputs are both on the E-Graph, otherwise returns an infinite value. This heuristic returns the minimal path cost from s_0 to s_{goal} where the path π is composed of an arbitrary number of two types of segments. One type of segment is a jump between s_i and s_{i+1} at a cost equal to the original graph heuristic inflated by ϵ^E. The other type of segment is an edge on the E-Graph, and its cost in π is its actual cost. In this way, the larger ϵ^E gets, the more the search prefers to utilize path segments on the E-Graph, since searching off of the E-Graph becomes costly.

There are some situations where this can be optimized. Some heuristics are computed using dynamic programming in a lower dimensional state space. For example, in a (x, y, θ) space, where x and y represent Cartesian positional information and θ represents an orientation, a 2D heuristic can be computed with Dijkstra's algorithm on just (x, y). In this case, the E-Graph heuristic can be computed by running the same reduced-dimensionality Dijkstra computation, but with additional edges in the graph added by projecting the E-Graph down to the lower dimensional space. This can be done with similar efficiency to the original Dijkstra-computed heuristic, so it does not add any significant computational overhead. The original E-Graph authors used this approach in their experiments.

3.4 Multi Heuristic A*

Multi-Heuristic A* enables multiple heuristics to be used to guide a graph search. The authors of [4] present two implementations: Independent Multi-Heuristic A* and Shared

Multi-Heuristic A* (SMHA*). We choose to focus on SMHA* and refer to it as simply MHA* throughout most of this thesis. In order to guarantee a bound on the suboptimality of solution costs, one heuristic, called the anchor heuristic, must be admissible and consistent. The other heuristics need not be admissible or consistent, and are generally referred to as the algorithm's set of inadmissible heuristics. SMHA* uses multiple open sets (individually similar to A*'s single open set), one for each heuristic. Open set zero is for the anchor search. There are two closed sets: one for the anchor search, and the other one for all inadmissible searches. The open sets are sorted by key, where the key for a particular state is computed differently for each open set. The pseudocode for SMHA* (including the Key function) is presented in Algorithm 3.

When the anchor search is admissible and consistent, SMHA* search is guaranteed to yield solutions with suboptimality bounded by a factor of search parameters $w1 \cdot w2$. The proof of this, and several other theoretical properties of SMHA* can be found in [4]

3.4.1 Inadmissible Heuristic Calibration

Though SMHA* utilizes heuristics which are arbitrarily inadmissible, when the scale of these heuristics greatly exceeds the scale of the anchor heuristic, search performance suffers. In our trials, this happened because the condition tested at Line 15 in Algorithm 3 can be rarely, if ever, satisfied, even for relatively large values of w_2. Thus, the inadmissible searches using $OpenSet_1$ through $OpenSet_n$ are not explored, and only the anchor search makes progress toward the goal. When this happens, the inadmissible heuristics do not contribute anything to the search process and the problem is essentially reduced to A* with only the anchor search. A solution is to calibrate the scales of the heuristics so that they can all contribute to guidance of the search.

Algorithm 3: Shared Multi-Heuristic A* (SMHA*)

```
1  Function Key(s, i)
2      return g(s) + w_1 · h_i(s)
3  Procedure ExpandState(s)
4      Remove s from OpenSet_i ∀ 0 ≤ i ≤ n
5      forall s' ∈ Succ(s) do
6          if s' was never before generated then
7              g(s') = ∞; bp(s') = null
8          if g(s') > g(s) + c(s, s') then
9              g(s') = g(s) + c(s, s'); bp(s') = s
10             if s' ∉ ClosedSet_anchor then
11                 Insert/Update s' in OpenSet_0 with Key(s', 0)
12                 if s' ∉ ClosedSet_inad then
13                     for 1 ≤ i ≤ n do
14                         if Key(s',i) ≤ w_2 · Key(s,0) then
15                             Insert/Update s' in OpenSet_i with Key(s', i)
16 Function Main()
17     g(s_start) = 0; g(s_goal) = ∞; bp(s_start) = bp(s_goal) = null
18     for 0 ≤ i ≤ n do
19         OpenSet_i = {s_start with Key(s_start, i)}
20     ClosedSet_anchor = ∅; ClosedSet_inad = ∅
21     while OpenSet_0.MinKey() < ∞ do
22         for 1 ≤ i ≤ n do
23             if OpenSet_i.MinKey() ≤ w_2 · OpenSet_0.MinKey() then
24                 if g(s_goal) ≤ OpenSet_i.MinKey() then
25                     if g(s_goal) < ∞ then
26                         Terminate and return path pointed to by bp(s_goal)
27                 else
28                     s = OpenSet_i.Top()
29                     ExpandState(s)
30                     Insert s in ClosedSet_inad
31             else
32                 if g(s_goal) ≤ OpenSet_0.MinKey() then
33                     if g(s_goal) < ∞ then
34                         Terminate and return path pointed to by bp(s_goal)
35                 else
36                     s = OpenSet_0.Top()
37                     ExpandState (s)
38                     Insert s in ClosedSet_anchor
```

3.4.2 Improved Multi Heuristic A*

Later work on a method called Improved Multi-Heuristic A* [44] also addresses the heuristic calibration problem. Pseudocode for Improved MHA* is presented in 4. Improved MHA* requires a termination criterion `Term-Criterion()`, a state priority function `Priority()`, and a potential set criterion `P-Criterion()`. Note that with the appropriate choices for `Term-Criterion()` and `Priority()`, Improved Multi-Heuristic A* is identical to Weighted A* when lines 17-24 are excluded from Algorithm 4. The potential set ($PSet$) holds a set of states which may potentially be expanded by the n arbitrary (potentially inadmissible and uncalibrated) heuristics. On line 17, the $PSet$ is constructed. On lines 18-24, an expansion is made for each arbitrary heuristic h_i and the state chosen to be expanded is selected according to h_i via `Rank(`s, i`)`. `Rank(`s, i`)` is $h_i(s)$ for uncalibrated heuristics and $g(s) + w \cdot h_i(s)$ for calibrated heuristics. All variants of Improved MHA* have theoretical guarantees similar to MHA*. The suboptimality of the solution is bounded by w times the cost of the optimal solution. For the proof of this and further discussion on the properties of the algorithm with particular sample instantiations for `Term-Criterion()`, `Priority()`, and `P-Criterion()`, see [44].

Algorithm 4: Improved Multi-Heuristic A*

```
1  Procedure ExpandState(s)
2      Remove s from OpenSet
3      forall s' ∈ Succ(s) do
4          if s' was not seen before then
5              g(s') = ∞
6          if g(s') > g(s) + c(s, s') then
7              g(s') = g(s) + c(s, s')
8              if s ∉ ClosedSet_a then
9                  Insert/Update s' in OpenSet with Priority(s')
10 Function Main()
11     OpenSet = ∅
12     ClosedSet_a = ∅; ClosedSet_u = ∅
13     g(s_start) = 0; g(s_goal) = ∞
14     Insert s_start in OpenSet with Priority(s_start)
15     while ¬ Term-Criterion(s_goal) do
16         if OpenSet.Empty() then return null;
17         PSet = {s : s ∈ OpenSet ∧ s ∉ Closed_u ∧ P-Criterion(s)}
18         for 1 ≤ i ≤ n do
19             s_i = arg min_{s∈PSet} Rank(s,i)
20             ExpandState(s_i)
21             ClosedSet_u = ClosedSet_u ∪ {s_i}
22         s_a = OpenSet.Top()
23         ExpandState(s_a)
24         ClosedSet_a = ClosedSet_a ∪ {s_a}
25     return solution path;          /* Path extracted by examining
        g-values or with backpointers as in Algorithm 3 */
```

Chapter 4

Training Graph Heuristic

4.1 Graph Search

NPC tactic planning can be accomplished as a graph search problem. Our search graph is composed of nodes representing search states and edges representing possible transitions between search states. Graph search states are composed of world state information and the state of the NPC whose behavior is being planned. World state information is composed of the state of every NPC (including enemies), and other miscellaneous world parameters. NPC state information encodes values for position, health, stamina, navigation information, gait (sneaking, walking, running), etc. for the NPC. Graph edges have an associated cost defined as the amount of time it takes to complete that state transition. A transition that results in companion NPC death has infinite cost.

We use heuristic graph search to find a feasible path from the start state to the goal. We tested our planner on different partially-specified goal states, including one that is satisfied when the NPC is within two meters of a destination location and one that is satisfied when a target enemy NPC is dead. Heuristic graph search algorithms use a *heuristic* which focuses search efforts to dramatically speed up the search process. A heuristic estimates

the distance between two states. A graph search heuristic can estimate the distance between a state and the goal, even if the goal is partially-specified.

A graph heuristic used throughout this paper, $h^S(a, b)$ between search states a and b (visualized in a test environment in Figure 4.1), is defined as Euclidean distance between the planned NPC positions in a and b divided by the maximum possible travel speed for the NPC. It therefore estimates the time-to-goal, which makes it consistent with the cost function.

4.2 E-Graph Heuristic

The E-Graph heuristic (as detailed in Section 3.3) enables graph search to re-use experience or demonstration data while searching for a goal-satisfying state on a graph. The E-Graph heuristic could be applied to our NPC behavior planning problem, however problems were encountered. This standard implementation of the E-Graph heuristic guides the graph search in a way that assumes the E-Graph is traversable and directly reachable from the search graph. The search is often guided toward E-Graph nodes, so if the E-Graph density is sparse compared to the search graph's discretization, the search may be guided backwards (away from the goal and backward along the E-Graph) to reach an E-Graph node before making progress toward the goal (these are the local minima visible in Figure 4.2). Also, due to the use of Dijkstra's algorithm in the existing E-Graph heuristic formulation, the search may leave the E-Graph long before reaching its end. This can prevent the E-Graph demonstration data from leading the solution path first toward the goal and then away from it, though that kind of behavior is important to some tactical maneuvers.

4.3 T-Graph Heuristic

We modify the computation of the heuristic so that it encourages the search to expand states that *follow* the demonstration without requiring the state to be *exactly* on the demonstration. In other words, we need to make the heuristic decrease between the start and the demonstration graph, then decrease along the length of the demonstration graph and, finally also decrease between the end of the demonstration graph and the goal. Our new heuristic is visualized for a test environment in Figure 4.3. Training graph (T-Graph) T is represented the same way as an E-Graph. Let T^N indicate the set of nodes making up T. Let $succ(s)$ return the set of successor neighbors of T-Graph node s. Let $pred(s)$ return the set of T-Graph nodes s' such that $succ(s') = s$. For all $s \in T^N$ and $s' \in succ(s)$, let $c^T(s, s')$ describe the cost of transitioning from s to s'. Let $T^N_{term} \subset T^N$ represent the set of terminal nodes s in T^N such that $succ(s) = \emptyset$. Let $T^N_{init} \subset T^N$ represent the set of initial nodes s in T^N such that $pred(s) = \emptyset$. A *shortest path* graph heuristic $h^P(a, b)$ is defined as the shortest path length from a to b on the search graph divided by the maximum possible travel speed.

We compute our training heuristic $h^T(a, b)$ as follows:

1. Let ϵ^T be a heuristic inflation factor like ϵ^E in the E-Graph method.

2. For each terminal node $s \in T^N_{term}$, assign $h^T(s, b) = h^P(s, b)$.

3. Working backwards from each terminal node s, where $s' \in pred(s)$, assign $h^T(s', b) = h^T(s, b) + c^T(s', s)$. Repeat this until every T-Graph node s' is assigned a $h^T(s', b)$ value. This step replaces the Dijkstra calculation in the E-Graph method.

4. An estimation $h^T_{est}(a, b, s, s')$ of the heuristic between a and b is computed for every pair of T-Graph nodes $s \in T^N$ and $s' \in succ(s)$ by observing that $h^S(s, s')$, $h^S(s, a)$, and $h^S(a, s')$ are available to compute a notion of how far a is located between s and s'. This is used to choose a value for h^T_{est} between $h^T(s, b)$ and $h^T(s', b)$ (or larger).

In our method, we decided to compute this as a projection π and rejection ρ from an imaginary line between s and s' as follows:

(a) Let $\pi = \frac{h^S(a,s')^2 - h^S(s,a)^2 + h^S(s,s')^2}{2h^S(s,s')^2}$

(b) Let $\alpha = \pi / h^S(s, s')$

(c) Let $\rho = \sqrt{h^S(s,a)^2 - (h^S(s,s') - \pi)^2}$

(d) If $\alpha < 0$, let $h^T_{est}(a, b, s, s') = \epsilon^T h^S(a, s') + h^T(s', b)$

(e) If $\alpha > 1$, let $h^T_{est}(a, b, s, s') = \epsilon^T h^S(s, a) + h^T(s, b)$

(f) If $0 \leq \alpha \leq 1$, let

$h^T_{est}(a, b, s, s') = \epsilon^T \rho + \alpha c^T(s, s') + h^T(s', b)$

5. An estimation $h^T_{est}(a, b, s, b)$ of the heuristic between a and b is computed for every pair of terminal node $s \in T^N_{term}$ and node b in the same way as step 4. This allows the search to be drawn in a focused way from the training data toward the goal. In our work, we use h^P here instead of h^S as it tended to make this last section of the heuristic computation conform to the navmesh better and therefore produced better results.

6. An estimation $h^T_{est}(a, b, a, s)$ of the heuristic between a and b is computed for every pair of node a and initial node $s \in T^N_{init}$ in the same way as step 5. This allows the search to be drawn in a focused way from the start node toward the training data.

7. The smallest of the direct path $\epsilon^T h^S(a, b)$ and every $h^T_{est}(a, b, s, s')$ estimate among all $s \in T^N \cup \{a\}$, $s' \in T^N \cup \{b\}$ is selected as $h^T(a, b)$.

On state expansions, for every state s and successor state s' (with transition cost $t = c^T(s, s')$), a deterministic world simulation function $w' = sim(s_w, t)$ is used to forward-simulate world state s_w over a duration of t, outputting updated world state w'. Successor

state s' is then assigned w' as its world state information. The simulation function evaluates models of behavior for all dynamic components of the world. In this function, behavior models for all NPCs are evaluated. The NPC being planned may be partially modeled by sim, for example if despite the planner, parts of this NPC's behavior are to be controlled by another technique. In our implementation, planned parameters include desired location (as a position) and willingness to fight (as a Boolean value) and then deterministic scripted behavior handles the details of navigation and combat in sim. As long as accurate NPC behavior models are available to sim, sim accurately predicts how all NPCs behave during the graph search. Other dynamic components of the world relevant to the planning problem can be modeled here, such as doors which automatically open or close, physics on moving objects, or consideration for how some kinds of attack damage (e.g. *splash damage* which could affect multiple targets) should be resolved.

4.4 Theoretical Properties

Some general properties of ϵ-admissible heuristic graph search are preserved in our method. The search is *complete*; if a solution is possible on the search graph, the search will return a solution. We do not modify the search graph in any way and we use Weighted A* graph search, which is a complete planner, so our search is also complete.

Our heuristic is ϵ^T*-admissible*. An admissible heuristic between nodes a and b never overestimates the actual transition cost between a and b. An ϵ-admissible heuristic never overestimates the transition cost by more than a factor of ϵ. Our heuristic $h^T(a, b)$ is computed as the minimum of several options, one of which is the direct connection (step 7 above) between nodes a and b, computed as $\epsilon^T h^S(a, b)$, so $h^t(a, b)$ must always be less than or equal to $\epsilon^T h^S(a, b)$. $h^S(a, b)$ is an admissible heuristic, so $\epsilon^T h^S(a, b)$ is an ϵ^T-admissible heuristic, and since $h^T(a, b) \leq \epsilon^T h^S(a, b)$, it must also be an ϵ^T-admissible

heuristic.

A consistent heuristic obeys the triangle inequality as specified here: $h(a,c) \leq c(a,b) + h(b,c)$. That is, the heuristic value between nodes a and c should be less than or equal to the cost to transition from a to successor node $b \in succ(a)$ plus the heuristic value between b and c. An ϵ-consistent heuristic obeys this inequality: $h(a,c) \leq \epsilon \cdot c(a,b) + h(b,c)$. ϵ^T-Consistency of our h^T heuristic depends on the formulation of h^T_{est} in steps 4 and 5 above. We found that our searches using the T-Graph heuristic functioned well and path costs were always well within sub-optimality bound $w \cdot \epsilon^T$ times the cost of the optimal path. However, by testing the ϵ-consistency condition during some sample searches using the T-Graph heuristic, we discovered counterexamples to the claim that the T-Graph heuristic is always ϵ-consistent. Occasionally, inconsistent expansions were made. In these cases, the difference between the sides of the inequality was very small (under thousandths of a heuristic unit), so there is a possibility that the ϵ-inconsistency we observed was actually an accumulation of floating point error. Future work using the T-Graph heuristic would have to consider how to remedy the problem of ϵ-inconsistency or expect to utilize a search algorithm tolerant to inconsistent heuristics, such as MHA*[4]. In the rest of our work, we use the T-Graph heuristic in MHA* in a manner where the ϵ^T-inconsistency of the T-Graph heuristic has no effect on the guarantees of the algorithm.

4.5 Implementation in Skyrim

We have used the video game Skyrim as a testing ground for our work. Since we do not have access to full models of how the game's existing NPCs behave, we have simplified the game's wolf and bear AI models so that they can be better modeled in our system. We depopulated game areas and repopulated them with our custom wolf and bear agents for testing. We extracted game navmesh data, including Skyrim's unidirectional

drop down edges, from these game areas for use in our system's navigation routines. We have used the game's *Creation Kit* editing software and modified the *Skyrim Script Extender* (SKSE, http://skse.silverlock.org/) for integration with our behavior planner and demonstration recorder. Demonstration data is recorded by sampling all relevant state information of the game world (health values, agent positions, etc.) at a regular interval and saving this information to a log file. The player presses a key (handled with SKSE and *Papyrus* scripting) to start and stop each log file. These logs are then used by our algorithm as demonstration data. They player has another key control to begin the NPC's autonomous planned control, using the recorded demonstration data.

4.6 Analysis

Our h^T heuristic calculation guides the graph search along training demonstration data paths, even where they might move away from the goal. Like with the E-Graph heuristic computation, if $\epsilon^T = 1$, the heuristic degenerates to the graph heuristic h^S (see Figure 4.4). Higher values of ϵ^T encourage the search to use the demonstration data more closely. Our formulation tolerates situations where the demonstration data does not lie directly on the search graph.

Since our method tolerates sparse demonstration data samples, it permits demonstration data to be sampled below its full resolution for performance improvements at the expense of training precision, if desired.

Typically, many parts of the h^T computation can be precomputed and accessed from memory at runtime. However, if the sim function can affect the heuristic functions, then h^T needs to be evaluated in full at runtime. For example, if the goal of a problem is to kill a mobile enemy NPC and the heuristic guides the search toward the location of the NPC, then the T-Graph heuristic needs to be fully reevaluated every time the NPC moves.

Some experimental results comparing the performance of the T-Graph algorithm to A* with a Euclidean distance heuristic follow. These are an expansion of the results from [45], and additional experimental results are presented later in this thesis in Chapter 7. We used two scenarios to test the T-Graph algorithm. The first scenario is the one presented above in Figure 4.1, which we call the Bear Bypass scenario. The second test scenario we constructed by extracting the navmesh representing the navigable portion of a cave in Skyrim and then populating it with a dangerous bear. We name this scenario after the name of the same cave in Skyrim, Greywater Grotto. See Figure 4.5 for the layout of the scenario including start position S, goal position G, and killer bear B. Also in that figure is a solution path found by Weighted A* (the dark dots are waypoints on the path). Figure 4.6 shows a solution found using the T-Graph heuristic. The T-Graph heuristic is visualized in Figure 4.7.

Scenario	Sub-optimality Bound	w	ϵ^T	Time	Expansions	Path Cost
Greywater Grotto	1	1		0.20 s	508	625 s
	10	10		0.14 s	412	646 s
	100	100		0.11 s	253	646 s
	1	1	1	0.17 s	508	625 s
	10	10	1	0.11 s	412	646 s
		1	10	0.18 s	447	655 s
	100	100	1	0.10 s	253	646 s
		10	10	0.11 s	234	703 s
		1	100	0.14 s	272	720 s
Bear Bypass	1	1		0.16 s	940	48 s
	10	10		0.10 s	544	54 s
	100	100		0.10 s	547	52 s
	1	1	1	0.22 s	940	48 s
	10	10	1	0.11 s	544	54 s
		1	10	0.06 s	121	55 s
	100	100	1	0.13 s	547	52 s
		10	10	0.06 s	80	55 s
		1	100	0.06 s	92	55 s

Table 4.1: Experimental results for T-Graph heuristic and Weighted A*. These results are plotted in Figure 4.8 and Figure 4.9

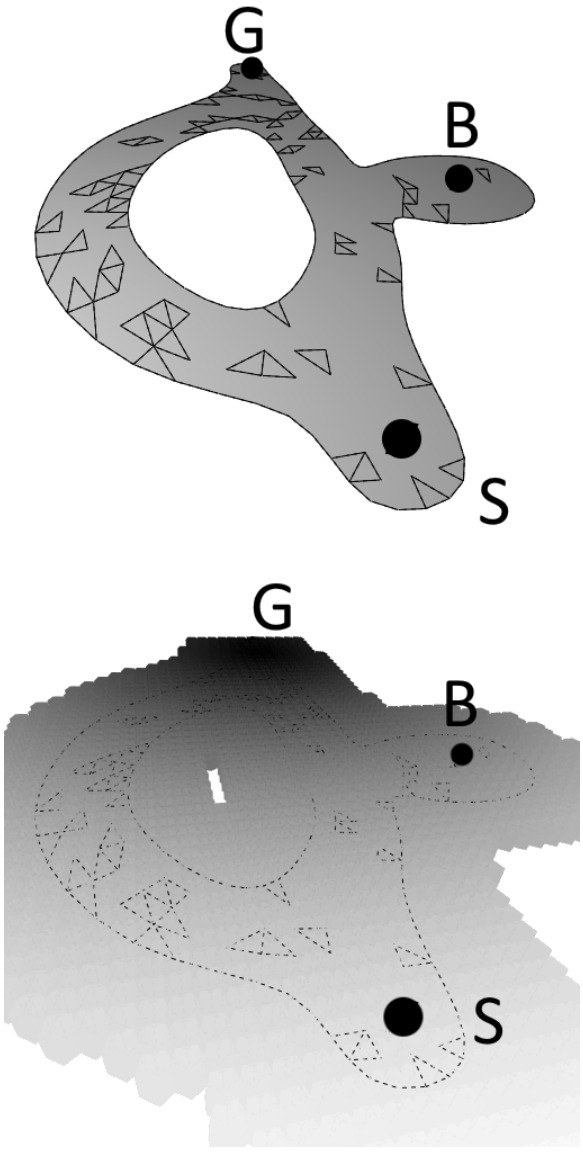

Figure 4.1: Top: Test environment with forked navmesh, start position S, goal G, and killer bear B hiding in den.
Bottom: Standard graph heuristic h^S, based on Euclidean distance. The values decrease (darken) near the goal in a smooth manner.

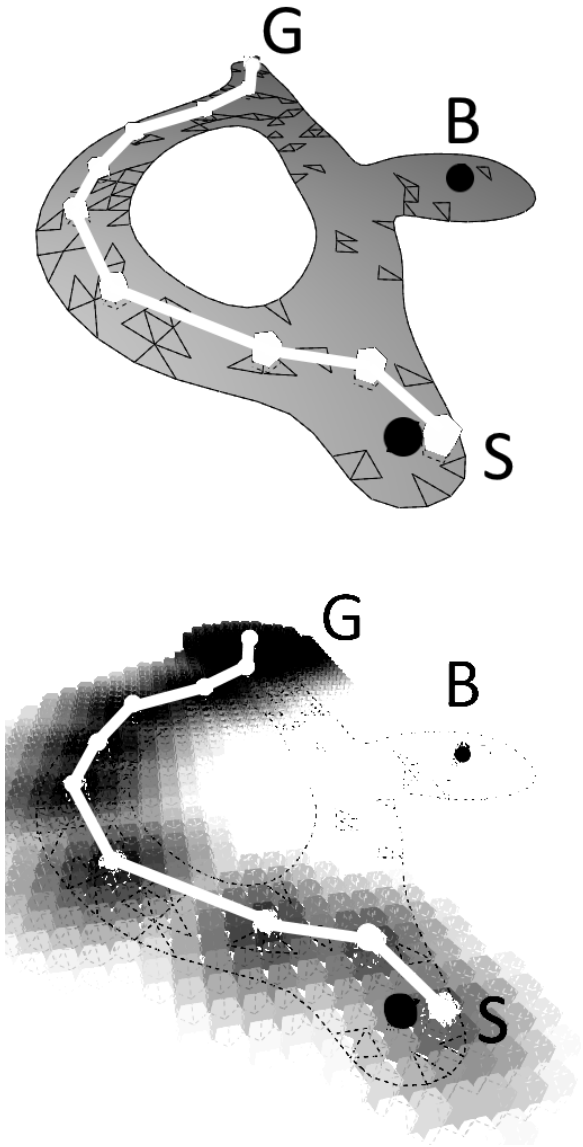

Figure 4.2: Left: Test environment with training path (white lines) visualized.
Right: E-Graph heuristic used without E-Graph being connected to the search graph. The white lines represent the E-Graph. Heuristic values decrease both toward the E-Graph nodes and overall toward the goal, but each node forms a local minima, which significantly delays search progress.

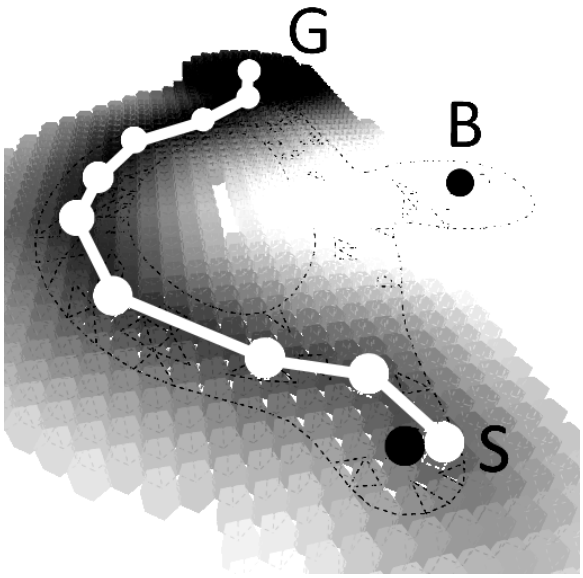

Figure 4.3: T-Graph heuristic values decrease both toward the T-Graph and along it toward the goal in a smooth manner. White lines represent the T-Graph.

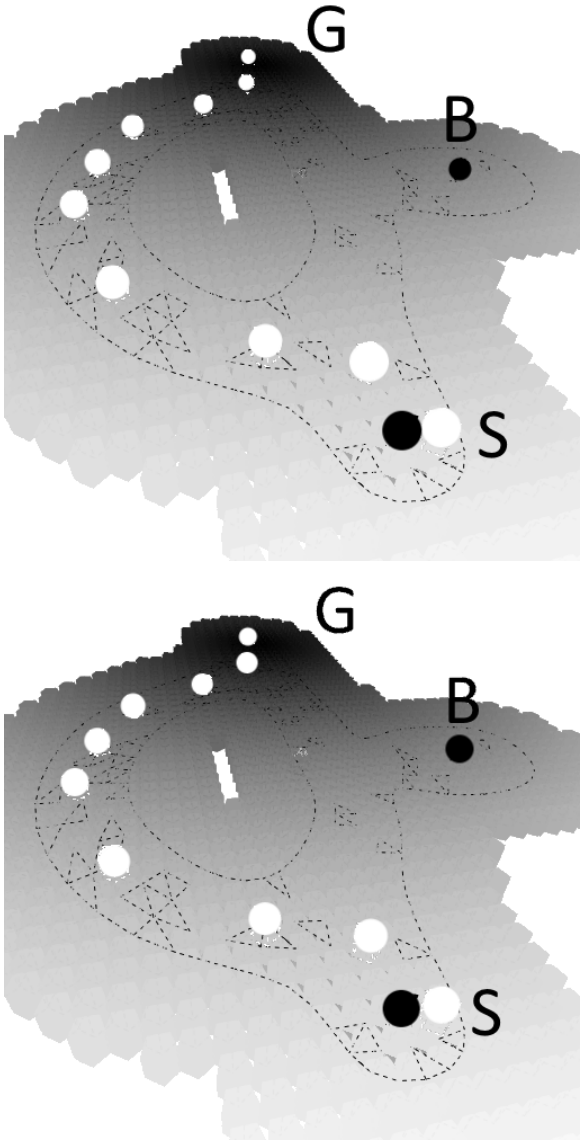

Figure 4.4: Top: E-Graph heuristic with $\epsilon^E = 1$.
Bottom: T-Graph heuristic with $\epsilon^T = 1$. Note that these both degenerate to the simple graph heuristic h^s seen in Figure 4.1.

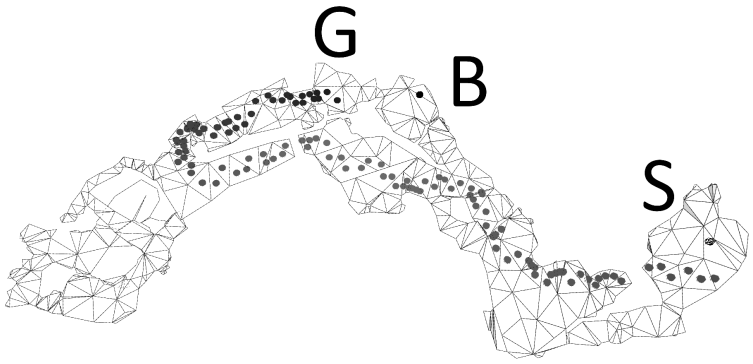

Figure 4.5: Greywater Grotto scenario and Weighted A* solution path.

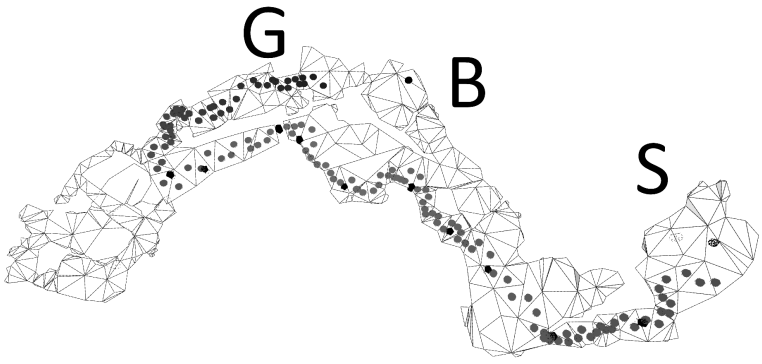

Figure 4.6: Greywater Grotto scenario and T-Graph solution path.

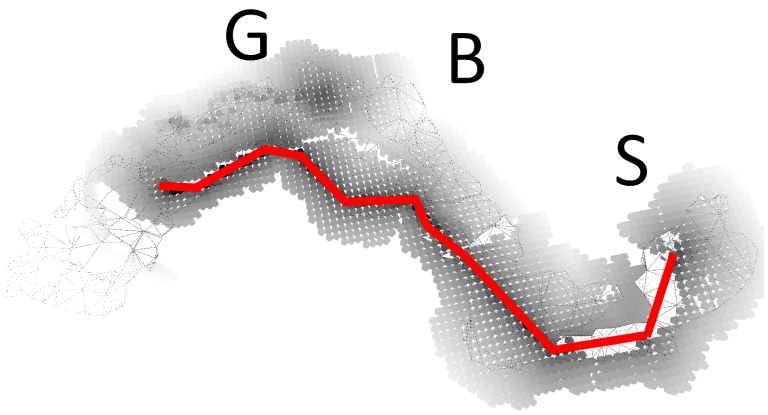

Figure 4.7: Visualization of the T-Graph heuristic in the Greywater Grotto scenario. The red line represents the T-Graph. Darker samples represent lower heuristic values. The region near the goal actually has the lowest heuristic values, but due to a sampling artifact, this area appears in this visualization to be brighter than parts along the T-Graph.

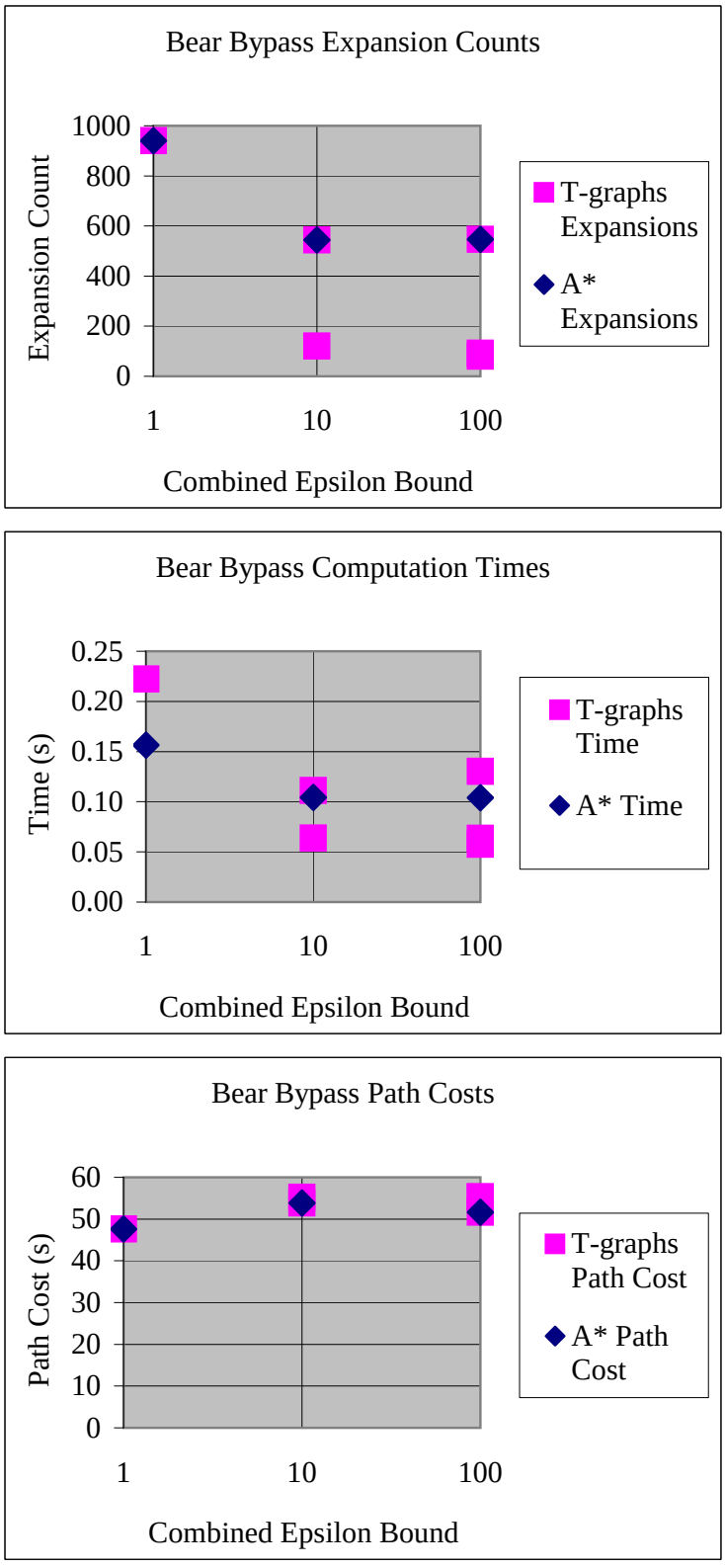

Figure 4.8: Experimental results for Bear Bypass scenario.

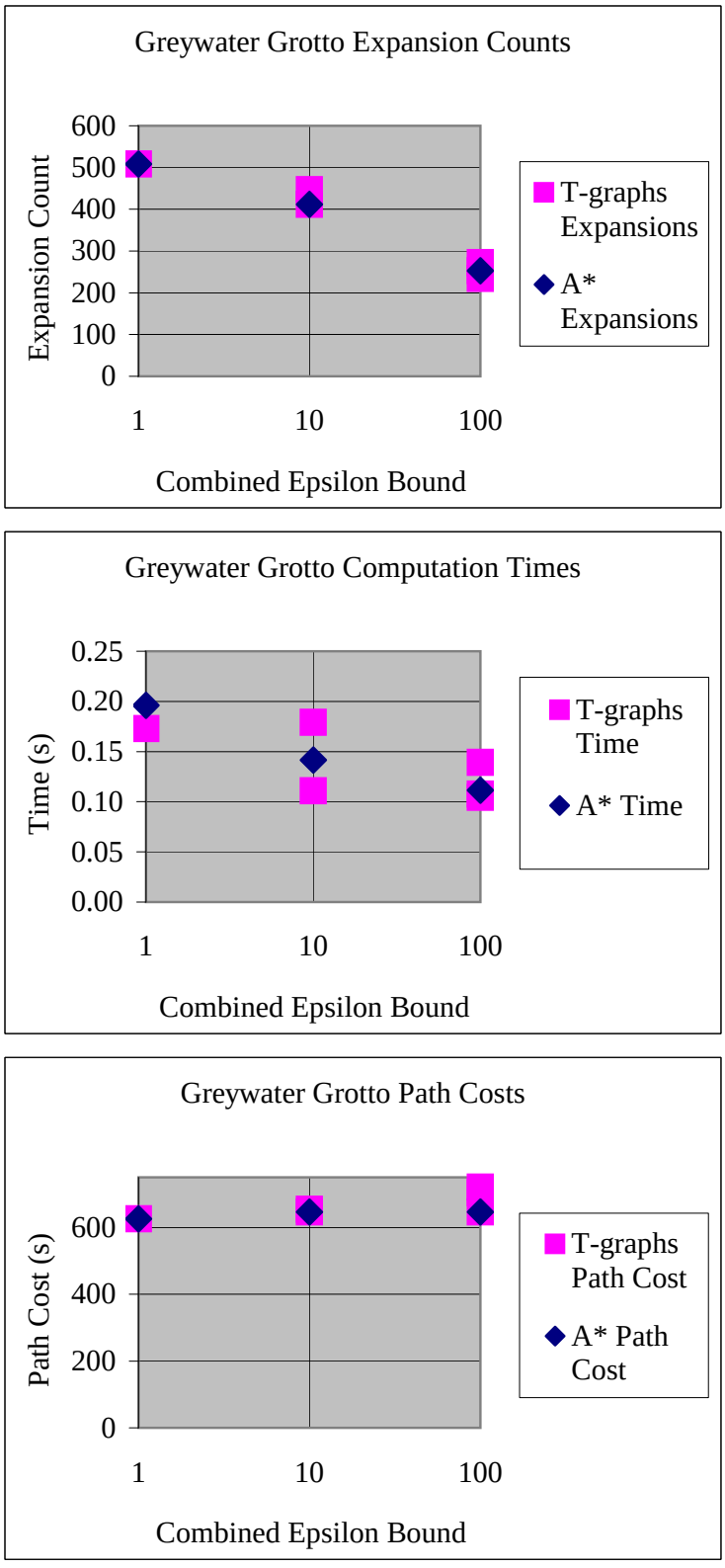

Figure 4.9: Experimental results for Greywater Grotto scenario.

Chapter 5

Adaptability Across Quests

When the E-Graph heuristic (or T-Graph heuristic from [45], explained in Section 4.3) is used to guide an NPC behavior planner search to reuse training paths and the current quest configuration of a dungeon differs from its configuration in the training quest, search progress can be delayed by large local minima due to these differences. A contribution of this thesis is in the use of Multi-Heuristic A* graph search to alleviate such issues.

Multi-Heuristic A* (MHA*) [4] performs a graph search while exploiting the guidance of multiple different heuristics. We will focus on the MHA* implementation called Shared Multi-Heuristic A*. MHA* essentially conducts multiple separate graph searches, each using a different heuristic. One search, called the anchor search, uses an admissible heuristic, while the other searches do not need to be admissible. In A*, Weighted A*, and many other graph search algorithms, the solution path is entirely discovered by just one search using one heuristic. However, in MHA*, search g-values are shared across searches, which enables the solution path to be composed of segments first explored by different searches with different heuristics.

5.1 MHA* With T-Graph Heuristics

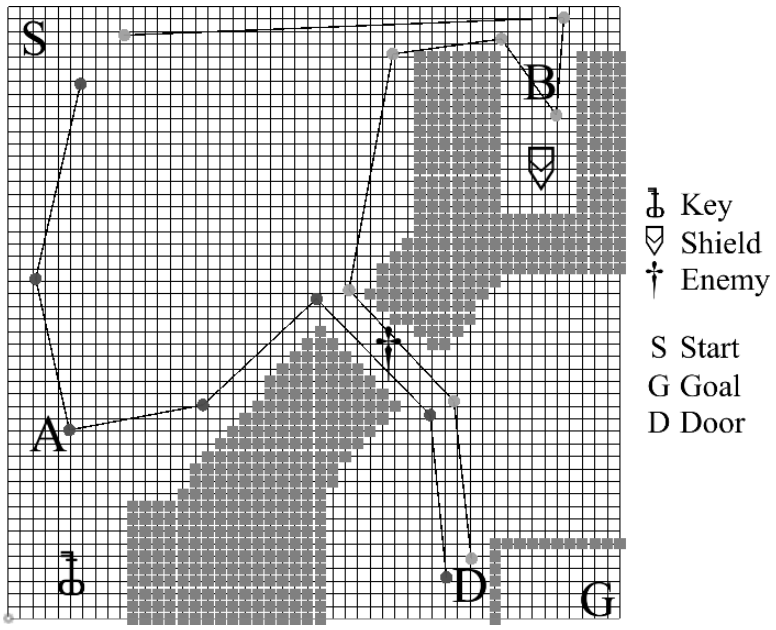

Figure 5.1: Test environment used in this paper to illustrate the advantage of using MHA*. The two lines with dots at each vertex represent E-Graph paths where the key and shield were instead located at points A and B, respectively.

Consider the following example of a simplified game with an NPC and a dungeon environment. The NPC can navigate the environment by walking or sneaking, but there are obstacles in the environment blocking some paths. In addition, the NPC can complete some spatial events in the environment, such as picking up an item. A key and a shield are available in the environment. The NPC starts without holding the key nor the shield and must walk over these items to pick them up. There may be an enemy in the environment which attacks when the NPC gets near it. The enemy's attacks hurt less if the NPC is carrying a shield. There may be a door in the environment, which will not open unless the NPC possesses the key.

See Figure 5.2 and Figure 5.1 to see how a dungeon configuration can change between quests. In this example there was one dungeon iteration with a shield and an enemy, another

dungeon iteration with a key and a door, and finally a third iteration with a shield in a new position, an enemy, a key in a new position, and a door.

See Figure 5.3 for a visualization of how the T-Graph method can perform well when the demonstration path closely matches the current quest configuration, but performance suffers severely when the quest changes.

We use h^S (Euclidean Distance heuristic) as the anchor heuristic in our MHA*-based NPC behavior planner. We use several inadmissible heuristics as follows. Let each event instance i be defined as a partially-specified world state where an important quest event occurs. Examples of quest events include the moment an NPC picks up an important item and the moment an NPC interacts with an important part of the environment. A property $T(i, s)$ is true whenever the effect of i being *T*riggered is detectable on state s (e.g. after picking up a shield, the shield is present in NPC inventory). For each event instance i, we include an additional heuristic $h^{Q_i}(a, b)$ we call a *quest-event heuristic*. This heuristic guides the search first toward the event location and then from the event location to b according to the E-Graph heuristic. Specifically, the heuristic is computed conditionally as defined in Equation 5.1:

$$h^{Q_i}(a,b) = \begin{cases} h^S(a,i) + h^T(i,b) & : \quad \neg T(i,a) \\ h^T(a,b) & : \quad T(i,a) \end{cases} \tag{5.1}$$

If the event instance i has not been triggered (e.g. for the shield pick-up event, if the shield is not yet in inventory) at state a, then $h^{Q_i}(a, b)$ is the distance from a to i plus the E-Graph heuristic from i to b; otherwise if the event instance i has already been triggered (e.g. if the shield is already in inventory) at a, then $h^{Q_i}(a, b)$ is simply the T-Graph heuristic from a to b. A variation on our method would use a different heuristic (perhaps the T-Graph heuristic) for the distance between a and i in the case $\neg T(i, a)$, the case where event i has not yet been triggered at a.

Important quest events in games are often revealed as sub-goals in the description of a quest. For example, in Skyrim's Bleak Falls Barrow quest, the player knows upfront that they need to locate and acquire the "Dragonstone" item. Quest events can also be extracted from the game data describing the world. Stored in the game data file for the Bleak Falls Barrow location are the controls of the various puzzles and the key needed to open the last door of the crypt. Only very rarely are there things like keys and levers added to a game location without them being relevant to the tasks which need to be completed in that location, so it may be safe to automatically treat these elements as quest events. The game developer could manually mark quest events in special situations. Also, it is not unreasonable to let NPCs "know" the locations of quest events. In Skyrim, the vast majority of quests requiring the player to locate something also pinpoint exactly where on the world map it can be found. The game usually provides a compass heading toward the active quest objective, and a spell named "Clairvoyance" visualizes the path to it. NPCs may also be able to utilize this kind of guidance without it seeming that they are cheating.

5.2 Calibration of Heuristics

As mentioned in Section 3.4.1, MHA* may suffer from poorly calibrated heuristics. Since our inadmissible T-Graph heuristics have the same general form, they can all be calibrated to the anchor heuristic in the same way. We multiply our inadmissible heuristics by a factor of $1/(2 \cdot \epsilon^T)$ to account for the inflating effect of ϵ^T. The two in the divisor accounts for the inadmissible T-Graph heuristics generally leading the search along longer paths than the optimal solution. Our T-Graphs did not lead the search more than two times the optimal path cost out of the way, but if such demonstrations were used, the factor used to calibrate these heuristics might have to be made smaller. Because we do not adjust the anchor heuristic, the theoretical properties of MHA* are not affected by this calibration process.

5.3 Theoretical Properties

Our technique inherits theoretical properties from MHA*. Though we have focused on simply generating *feasible* solutions in less time, MHA* provides a bound on the cost of the solution path. Two parameters used within the MHA* algorithm are w_1 and w_2. The w_1 parameter inflates the heuristics used within the MHA* searches, and the w_2 parameter is a factor to prioritize the inadmissible searches over the admissible anchor search. The cost of the solution path found by MHA* is guaranteed not to be more than $w_1 * w_2$ times the cost of the optimal solution.

Though the T-Graph heuristic on its own introduces the possibility of sub-optimality in solution paths (in proportion to the size of the ϵ^E parameter), since we only use the T-Graph heuristic within the inadmissible searches of MHA*, ϵ^T has no effect on the sub-optimality guarantee of the MHA* search. MHA* provides its bound on sub-optimality independently of how inadmissible the inadmissible heuristics are.

5.4 Analysis

To illustrate the benefit of using MHA*, we implemented a simplified NPC behavior planning problem. In the example test game (Figure 5.1) is an NPC whose state includes 2D (x & y) position (each axis discretized into fifty possible values), a health value from 0 to 100 (discretized in multiples of twenty, so there are six possible values), stealth mode (sneaking or not sneaking), and an inventory which can hold (or not) a key and hold (or not) a shield.

This test quest requires the NPC to acquire both the shield and the key to reach the goal. The quest task is to traverse the dungeon space to reach the goal position G. The goal is within a room blocked by a locked door, and the vicinity of the room can only be reached by making it past an enemy at a choke point. The enemy can only be passed alive while

holding the shield. See Figure 5.4 to see an example of the kind of NPC behavior required to complete the quest task.

Two demonstrations, shown in Figure 5.1 and separately in Figure 5.2, each from a different quest in the same environment, are provided to the NPC. In one, there is no enemy at the choke point, and the key is in a slightly different location. In the other, there is no door, and the shield is in a slightly different location. The planning problem at hand is a compound problem incorporating versions of both the key-door challenge and the shield-bear challenge.

We compiled results for Weighted A* (including $w = 1$, which is standard A*), the T-Graph algorithm (ϵ^T is its inflation factor, like E-Graph's ϵ^E), and MHA* as described in the previous section. Various configurations of the w, ϵ^T, w_1, and w_2 parameters were tested and from these we chose the best for each algorithm. Listed in Figure 5.5 are the configurations we used. Because of the poor performance of the T-Graph method at adapting across quests, the optimum values for ϵ^T were actually all found to be 1, degenerating it to A* search, so we picked the value 2 instead to illustrate the problem. We generated MHA* results for $\epsilon^T = 2$ to compare, and also for $\epsilon^T = 100$ to show its capacity for improved performance for the same bounds.

As seen in Figure 5.7 the overhead of MHA* can cause it to run slower than the other methods in some cases when the bound factor is low, but as the bound factor increases, MHA* manages to outperform the other methods. A visualization of one MHA* solution can be seen in Figure 5.6. Compare to Figure 5.3 where Weighted A* with the T-Graph heuristic encounters a huge local minima and must expand many states before finding a valid solution.

As seen in Figure 5.8, our MHA* approach outperforms the other methods in terms of expansion counts. Our MHA* method performs the best because it adapts the demonstrations to the new event locations and utilizes information from both experience paths

together while searching for the goal. Note that the MHA* samples for $\epsilon^T = 2$, N*, O*, P*, Q*, and S* outperform the corresponding T-Graph samples H†, I†, J†, K†, and L†, even though ϵ^T is the same and they have the same sub-optimality bound.

For a more complex problem, where each state expansion is more expensive (e.g. more expensive successor generation due to collision checking, world modeling, etc.), MHA*'s advantage in expansion counts outweighs the overhead of using MHA*. We tested this by adding several parameters to the NPC state, increasing the number of successors generated for a state from 16 to 256. Results are in Figure 5.9. The state parameters added to the search space include rotation (θ, discretized into four possibilities), combat mode (willing to engage in combat or not), and blocking mode (blocking incoming attacks or not).

We also tested randomizing the key and shield locations to see how our MHA* method performs over a range of initial conditions. The key and shield were randomly placed in feasible locations (so they could be acquired before they are needed to solve the problem). Our results can be seen in Figure 5.10. MHA* greatly outperformed Weighted A* and T-Graph in these trials.

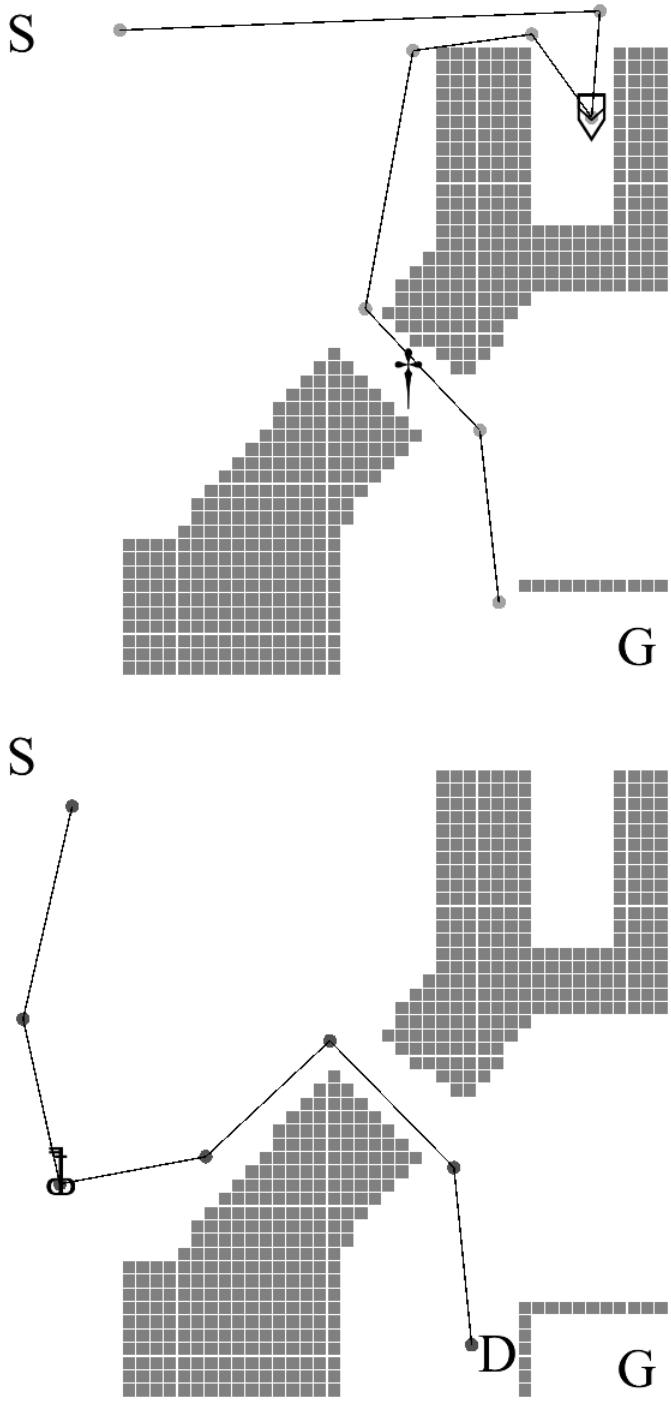

Figure 5.2: The configurations of the dungeon when the two demonstration paths were each recorded. On the top is a path demonstrating a behavior to pick up a shield before facing an enemy at the choke-point. The bottom path demonstrates picking up a key before reaching the locked door near the goal.

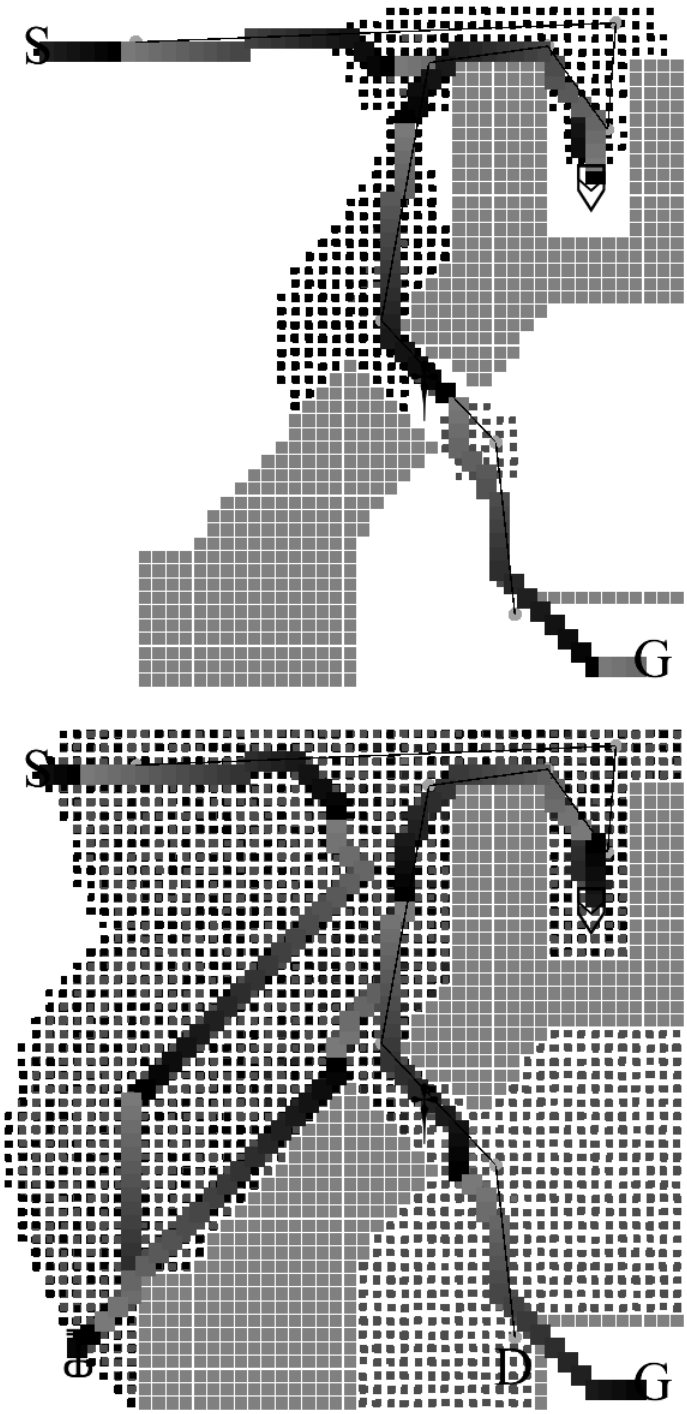

Figure 5.3: Top: The T-Graph planner is used to reach the goal quickly when given a demonstration similar to the solution.
Bottom: The T-Graph planner encounters huge local minima when used to solve a new scenario with the door and key introduced. Many states need to be expanded before the goal can be found.

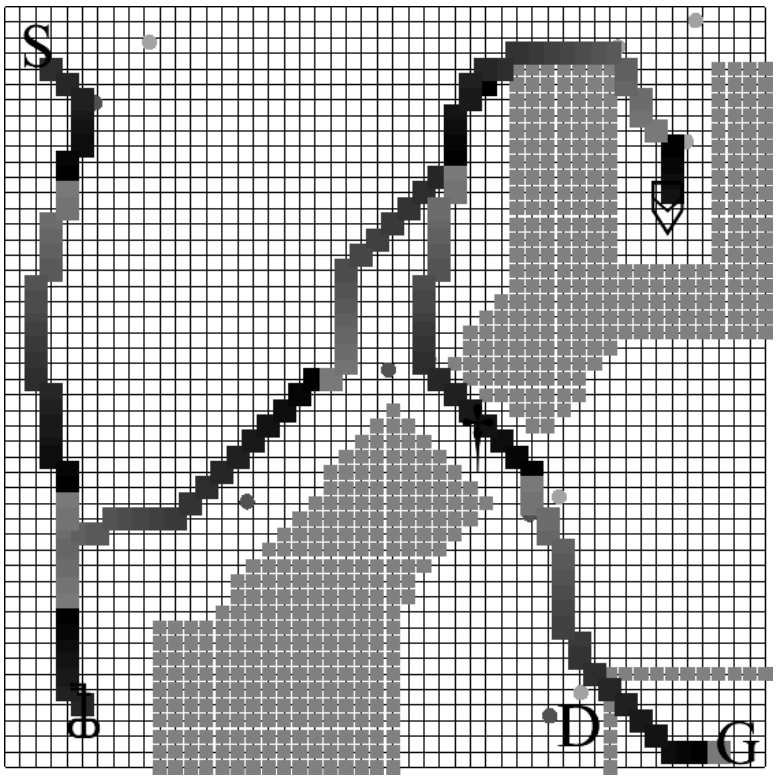

Figure 5.4: An example of a typical path planned to solve the quest task. The NPC starts at S, the start position, moves south to pick up the key, moves northeast and around a wall to get the shield, then moves to the center of the map to pass through the choke point with the enemy (which can only be survived with the shield), then finally moves southeast through the door (D), to the goal (G).

Label	w	ϵ^T	w_1	w_2	Bound Factor
					w
A	1				1
B	10				10
C	100				100
D	1000				1000
E	10000				10000
F	100000				100000
					$w * \epsilon^T$
G†	1	1			1
H†	5	2			10
I†	50	2			100
J†	500	2			1000
K†	5000	2			10000
L†	50000	2			100000
					$w_1 * w_2$
M*		2	1	1	1
N*		2	10	1	10
O*		2	20	5	100
P*		2	200	5	1000
Q*		2	2000	5	10000
R*		2	20000	5	100000
S*		100	1	1	1
T*		100	1	10	10
U*		100	1	100	100
V*		100	20	50	1000
W*		100	200	50	10000
X*		100	2000	50	100000

Figure 5.5: The parameter configurations used to generate performance results in Figure 5.7 and Figure 5.8. † indicates a T-Graph search, * indicates an MHA* search using our heuristics, all others are Weighted A*. The final column shows the sub-optimality bound for each configuration, used as a common reference to compare results between algorithms.

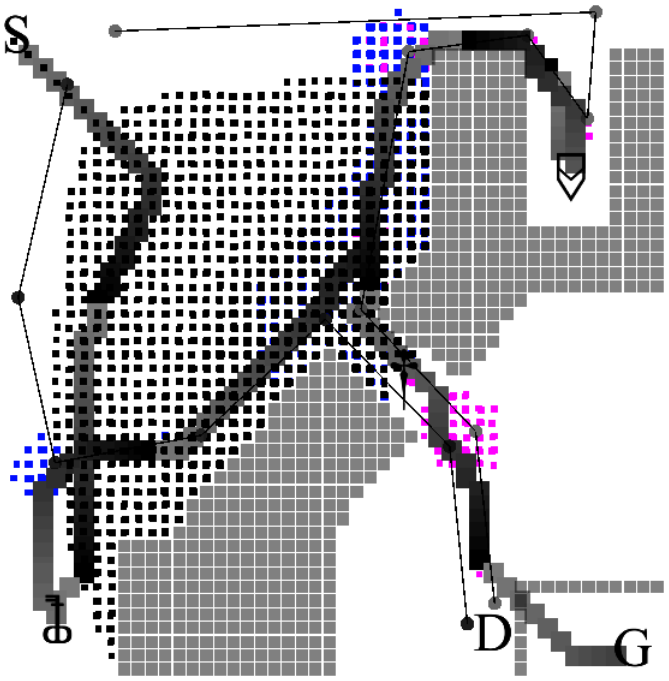

Figure 5.6: Sample MHA* solution, showing limited number of states expanded despite the compound problem.

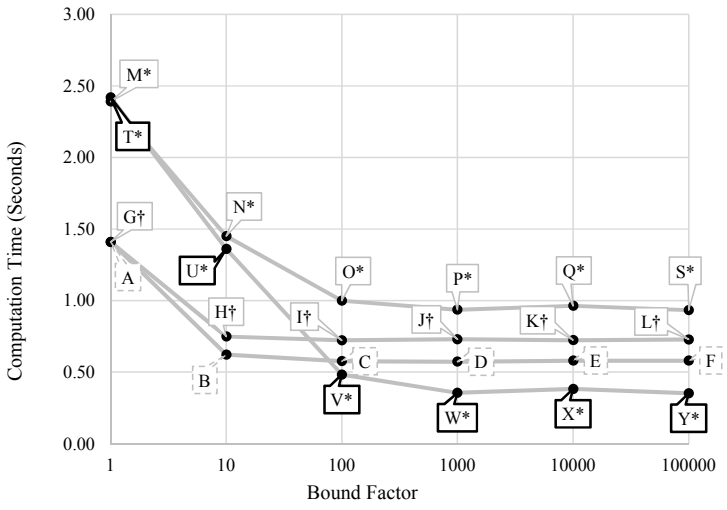

Figure 5.7: Computation time results. Reference Figure 5.5 for the configuration associated with each label.

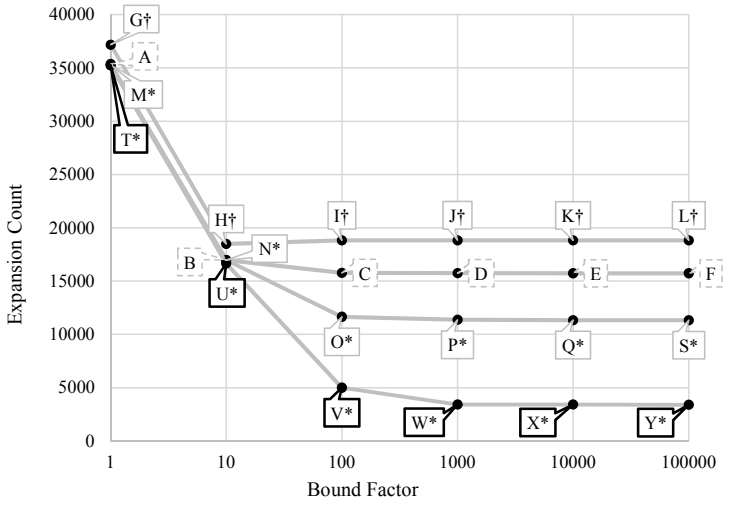

Figure 5.8: Expansion count results. Reference Figure 5.5 for the configuration associated with each label.

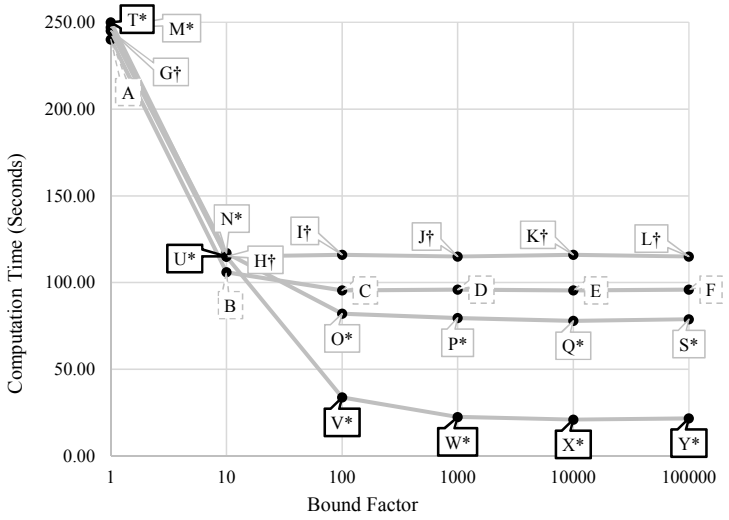

Figure 5.9: Computation time results for an expanded state space. Reference Figure 5.5 for the configuration associated with each label. Note that the computation times for MHA* have lowered significantly relative to the other search times, as compared to the standard state space results in Figure 5.7.

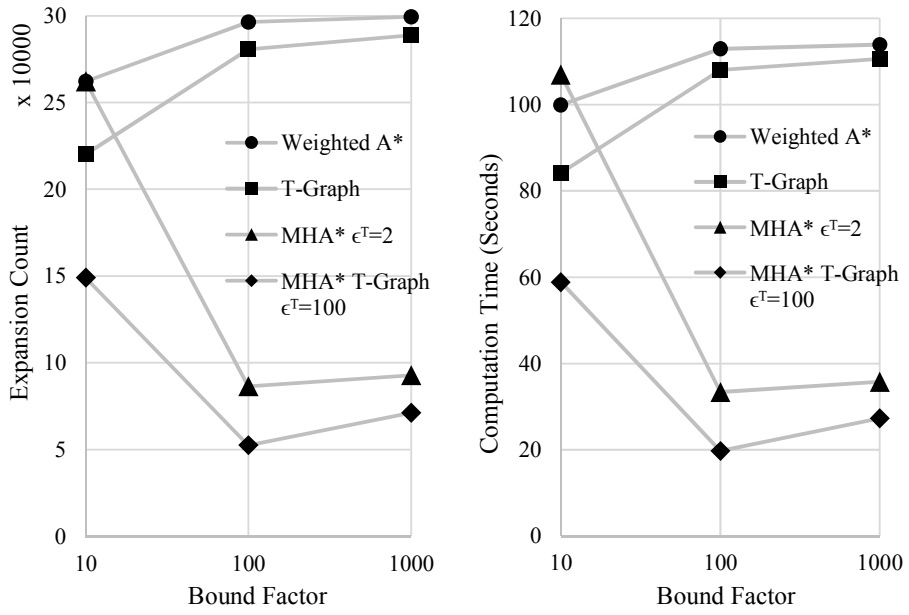

Figure 5.10: Randomized initial condition results. Like in the other trials, favorable parameters were selected to give each algorithm its best chance.

Chapter 6

Cooperative Planning

Master's thesis work by Stephen Chen [7] done in collaboration with the authors of this thesis, extends our NPC trained behavior planning framework to incorporate player behavior classification, player modeling, and cooperative behavior planning. Cooperative NPC behavior planning is the task of planning NPC behavior so that the NPC can work cooperatively with the player to solve the current game goal together. A game's level may be designed such that an NPC's assistance is required to complete the level. For example, in the Angarvunde tomb level in Skyrim, a key is required to access portions of the tomb necessary to proceed through to the end. The NPC named Medresi Dran holds the key, so excepting the rather impolite options of pickpocketing Medresi or outright killing her, the player needs to agree to assist her navigating the tomb and then she assists the player by giving the player the tomb key. Some NPCs are so important to the progression of game quests that they are made invincible by the game's developer and cannot be killed by any means. In existing games, these cooperative scenarios are accomplished with scripted NPC behaviors, and with scripted behavior comes its disadvantages of development cost and inflexibility. The player cannot ask these NPCs to assist in arbitrary ways: they are limited to their scripting.

An example of another game requiring cooperation to complete is the two-player campaign of Portal 2. Portal 2's cooperative mode is designed to be played by pairs of human players, but a sufficiently sophisticated NPC could be substituted for one of the players, and the player could train his NPC companion how to behave in the game. A primitive system of this kind was released in 2014 as a *mod* for Portal 2, called Time Machine[46]. In Time Machine, the player is given an opportunity to play through a level in the shoes of his companion. Then, the player can play through again as himself while his previous actions are replayed exactly by code controlling the NPC. In this way, the game's puzzles can be solved cooperatively. We endeavored to create a smarter and more flexible system like this, capable of adapting NPC behavior to solve new problems not precisely and completely demonstrated by the training input.

In order to use the player's behavior in our search-based planning framework, a model predicting the behavior of the player in the near future is required. We assume the set of the player's behaviors can be divided into distinct classes known as *playstyles*. We formalize the process of learning this playstyles as an offline unsupervised learning problem. With a set of learned playstyles, the player's current behavior in the current problem can be classified into a playstyle with an online classifier. Then, in the NPC behavior planning process, a model for predicted player behavior is extracted from the playstyle class matching the player's recent actions. Plans generated in this way exhibit cooperative behavior, with the NPC assisting the player to complete the goals of the game.

6.1 Learning Playstyles

We provide our playstyle classification system with a set of example traces of player behavior playing through various challenges in a video game. These traces are provided in a form we call *activities*. Activities are ordered sequences of *events*. Events tag important interac-

tion of agents or objects in the game environment. For example, in Skyrim, events might be as specific as the moment that a character swings a sword or abstract like "initiation of combat." Depending on how events are defined, activities can encode detailed low level behaviors or high level tactics and strategies such as "flank the enemy" and then "engage in melee combat." To capture playstyles, we use abstract events like "initiating combat" and "sneaking past an enemy." Though outside the scope of our work, the processing of raw gameplay data into events and activities could be done post-hoc by careful analysis of the raw gameplay traces. Alternatively, since the environment and opponents in video games are often controlled by such things as FSMs and RBSs, important high-level changes in these structures (such as an enemy FSM transitioning to a "pursuit" state) can be used as event triggers, building up an activity in an online fashion while gameplay happens.

We further process activities into what are called n-Gram histograms. This is done by counting the number of instances of each unique substring of length n in an activity for several values of n. Work in [47] shows that for human behavior, n-Grams tend to perform poorly above $n = 5$. We use $1 \leq n \leq 3$, a choice shown in our experiments (see results in [7]) to be the best choice in our domain.

Next, our activity n-Grams are clustered into playstyles with an unsupervised clustering algorithm. To facilitate clustering, we created a similarity metric between activities, defined in Equation 6.1. $K_s \in [0, 1]$ is the weight of substring s and $\sum_{s \in S_A, S_B} K_s = 1$. Notation $s \in S_A, S_B$ means that each element of S_A and S_B is considered individually, even if it appears in both S_A and S_B (then it is used as substring s twice). Our similarity metric always returns similarity values in the range $[0, 1]$. Activities which are exactly the same have similarity value 1, and activities with mutually exclusive substrings have similarity 0.

$$sim(A, B) = 1 - \sum_{s \in S_A, S_B} K_s \frac{|f(s|H_A) - f(s|H_B)|}{f(s|H_A) + f(s|H_B)} \tag{6.1}$$

We treat all input activities as nodes in a fully connected graph. Edges between activity nodes are assigned weights according to our similarity metric (Equation 6.1). In partially-connected graphs, a *maximal-clique* is defined as the largest fully-connected subgraph. Since our graph is fully connected, we instead find a *dominant set* in the graph, defined as the maximal subgraph with high-weight edges connecting the vertices in that subgraph. This approach is described in [48]. We identify playstyle classes by repeatedly finding the dominant set in the graph and then removing it from the graph. Each dominant set found represents a playstyle, since all of the behaviors in that set are highly similar to one another. We find dominant sets in our graph using an optimization technique known as *replicator equations*. For details on how this works, please see Stephen Chen's thesis[7] and [48].

6.2 Player Behavior Classification

Now, a new player behavior activity τ can be classified as belonging to a particular playstyle. To accomplish this, we score τ according to its similarity with each existing playstyle class $c \in C$ with the function in Equation 6.2. $p_c(j)$ is a function detailed in [7] which represents the *participation* of an activity in its playstyle class (how well it fits in). Then, the best playstyle class, c^* for an activity is simply the one with the highest similarity: $c^* = \arg\max_{c \in C} A_c(\tau)$. Alternatively, similarities can be normalized to produce a probability distribution over the possible classes. Using normalization factor $\alpha = \sum_{c \in C} A_c(\tau)$, probability that any particular class is the optimal class for τ, $Pr[c^* = c] = \frac{A_c(\tau)}{\alpha}$.

$$A_c(\tau) = \sum_{j \in c} sim(\tau, j) p_c(j) \tag{6.2}$$

Described above is our *activity classifier* method for producing a single playstyle clas-

sification given a player activity. However, the human player may adopt a new playstyle while playing the game, and the system should be able to detect this over time and issue a new playstyle classification. Also, when play first begins, until a record of recent player activity is built, the methods we described above cannot yet be used to classify the player's playstyle. To account for these problems, we combine several factors in a Bayesian Belief Network (see Figure 6.1). The probability distribution on classifications at the next time stem $t + 1$ given the current observations OBS_t is ultimately determined by Equation 6.3. For a detailed description of each component, see [7].

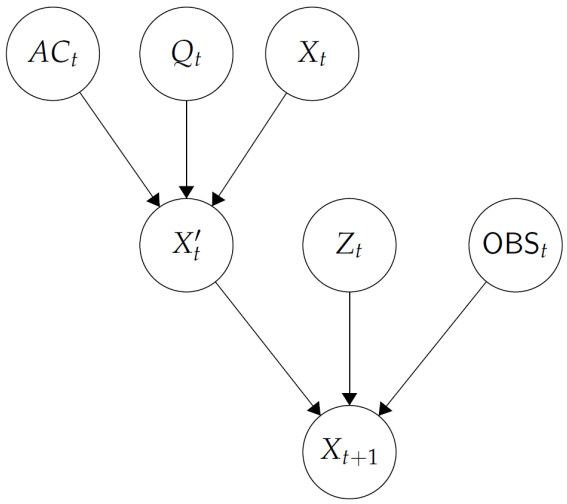

Figure 6.1: Bayesian Belief Network used in online playstyle classification.

This formulation has the following properties. It factors in confidence in the activity classifier, weighing it lower when there few events have been observed. It provides classification at each game time step, rather than at the resolution of individual triggered events. This is important when events are not triggered frequently. It factors in the motion of the player as they approach new potential events not yet triggered (in addition to the events already in the activity used for activity classification). Because of this, an online classification is available even when the activity classifier alone cannot yet provide a classification because there are zero or few events in the recent activity.

$$Pr[X_{t+1}|OBS_t] = Pr[Z_t = 0] \cdot Pr[X'_t] + Pr[Z = 1] \cdot Pr[Y_t|OBS_t] \quad (6.3)$$

6.3 Toward Cooperative Planning

We present an example to motivate the application of cooperative planning. We use a cave environment from Skyrim and populate it with a very difficult to defeat killer bear (B), easy to defeat wolves (W), and difficult to defeat superwolves (W*). See Figure 6.2 which illustrates the scenario, including start position S and goal G. The killer bear can only be defeated when two characters combine their forces to defeat it. The superwolves are best evaded by sneaking past them. Two playstyles relevant in this example are "COMBAT" and "SNEAK". In the COMBAT style, the player chooses to attack the bear, necessitating assistance from the NPC. In the SNEAK style, the player chooses to sneak past the superwolves to reach the goal, requiring the NPC to also sneak past the superwolves, since it could not defeat the bear on its own. Figure 6.3 shows example activity traces for each of these playstyles.

Planning NPC behavior to reach the goal in this scenario results in a path past the superwolves, shown in Figure 6.4. When a player model is introduced, planning complexity goes up due to the extra state information needed to model the player. An example solution with a player model (COMBAT type) is shown in Figure 6.5. As recorded in [7], search expansion counts go up from 478 to 1797 and search times increase from 0.135 seconds to 0.497 seconds due to the addition of the player model. The planned behavior includes a fight with the bear, since it can now be defeated.

However, a local minima in the search space is encountered when the SNEAK playstyle is detected and the player model attempts to sneak past the superwolves. The shortest path distance heuristic used previously guides the NPC toward the killer bear, which cannot now

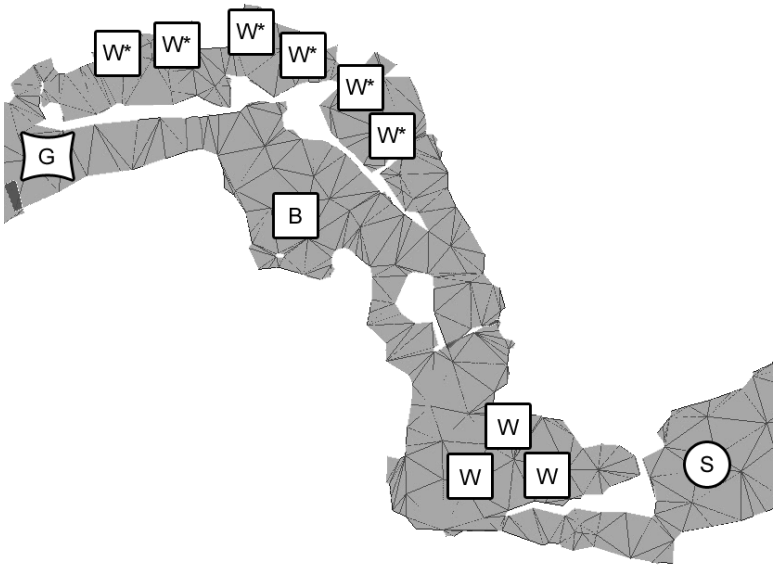

Figure 6.2: Cooperative NPC planning example.

be defeated since the player model predicts the player going the other way. This forms a local minima in the search space, slowing the search process so much that it takes over five minutes to complete. The solution path can be seen in Figure 6.6.

The Training-Graph heuristic is introduced to the problem to guide the NPC along the appropriate route. Since the playstyle is SNEAK, a sneaking example is used as the T-Graph, helping cut planning time nearly in half to 185 seconds. An example solution path for this configuration is provided in Figure 6.7. Finally, we also examined the use of MHA* with a T-Graph heuristic. The heuristics used in this instantiation of MHA* were: shortest path distances (admissible and consistent) for the anchor search, the T-Graph heuristic, and a special heuristic favoring states where the NPC was in sneak mode like the player. This cut down planning times for the SNEAK playstyle to 2.047 seconds, a vast improvement over both Weighted A* and T-Graph alone. Table 6.1 collects the experimental performance results for these cooperative planning episodes.

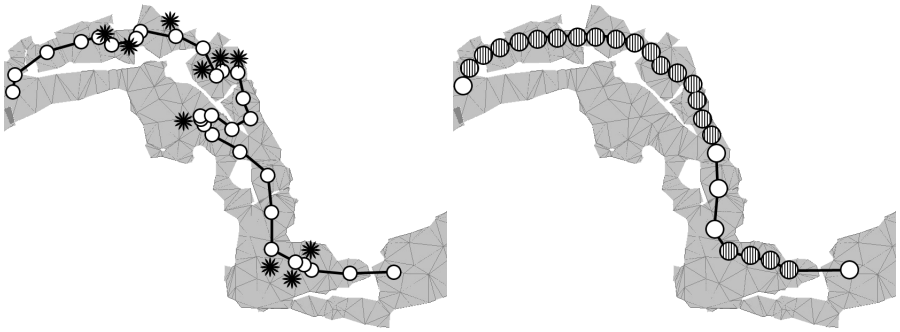

Figure 6.3: Playstyle exemplars for COMBAT (left) and SNEAK (right).

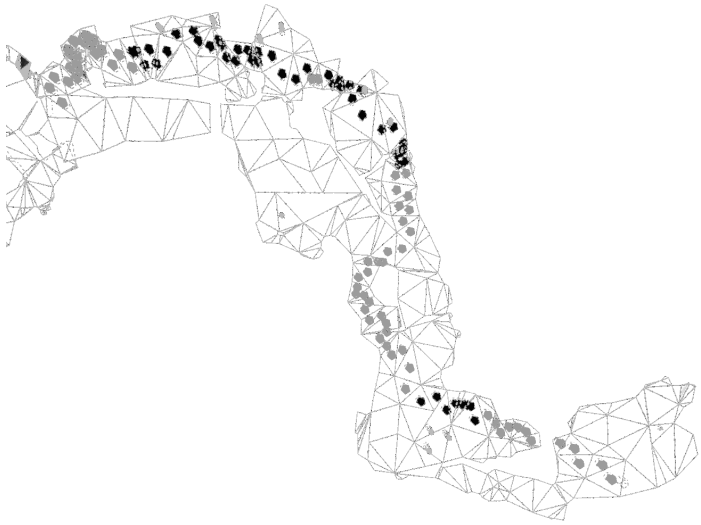

Figure 6.4: NPC planning example.

Algorithm (and Heuristic)	States Expanded	Planning Time	Path Cost
Weighted A* Shortest Path	898,008	306.003 s	140.1 s
Weighted A* T-Graph ($w = 10, \epsilon^T = 10$)	578,781	185.249 s	137.1 s
MHA* T-Graph ($w = 10, \epsilon^T = 10$)	345,208	117.892 s	197.3 s
Weighted A* T-Graph ($w = 20, \epsilon^T = 5$)	17,261	4.985 s	137.1 s
MHA* T-Graph ($w = 20, \epsilon^T = 5$)	9,356	2.047 s	166.7 s

Table 6.1: NPC planning results when following the SNEAK playstyle.

Figure 6.5: NPC planning example, A*, with COMBAT player model.

Figure 6.6: NPC planning example, A*, SNEAK player model.

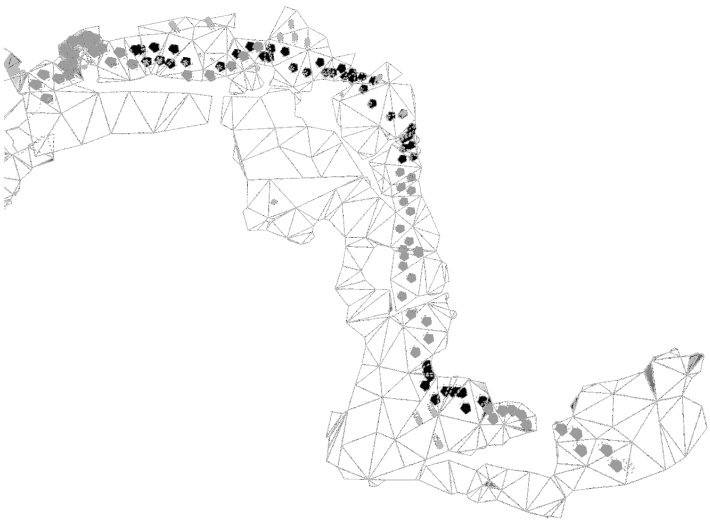

Figure 6.7: NPC planning example using T-Graph and player model.

Chapter 7

Experimental Analysis

We designed and executed several experiments to test the qualities of our methods. First, we tested the ability of the Training-Graph heuristic to produce NPC behaviors which mimic the demonstrated behaviors provided as training data. This is essentially a qualitative test, so several figures illustrating the capabilities of the Training-Graph approach are presented. Next, we provide an analysis of the computational performance of our methods in comparison to each other and to the alternative A* search. We demonstrate the computational plausibility of using our methods to generate tactical NPC behaviors in video games. Finally, we propose the use of skill-based rating algorithms to estimate the relative skill levels of NPC behavior generators as a means to compare them. We present preliminary skill rating results from pitting several NPC behavior generators against several sample quests.

7.1 Training Quality

Our goal in our work on the T-Graph and MHA* with T-Graph Heuristics methods is to enable planned NPC behaviors to follow training demonstrations in order to produce trained output behaviors. This is largely a qualitative pursuit, so we provide here some demon-

strations of the qualitative capabilities of our work, including a small user study soliciting human feedback.

Figure 7.1 shows that using the T-Graph heuristic in Weighted A* graph search produces solutions which mimic the training input. Figure 7.2 shows that using the T-Graph heuristic in Weighted A* graph search produces solutions which mimic the training input even when the training input cannot be followed exactly because it lies outside the current problem's search space. Figure 7.3 shows that the T-Graph heuristic will mimic the training input even if doing so requires taking actions besides simple navigation actions. Here, the search finds that it can use gait-changing actions to switch into sneak mode to both follow the spatial layout of the T-Graph and avoid being spotted by the bear.

While conducting the various experiments mentioned in this chapter, we stumbled upon a strategy for increasing the reuse of experience in MHA* solutions. In addition to the quest event heuristics described in Section 5.1, the T-Graph heuristic (calibrated the same way as the others) can be used as an additional inadmissible heuristic to guide the search. This causes states near the T-Graph to be expanded. By itself, this does not have a very large effect on the solutions found. However, if this one heuristic is expanded multiple times per search expansion (equivalent to duplicating this heuristic multiple times), it can form a tunnel through the search space, along the T-Graph, from which the other inadmissible heuristics branch off toward their intermediate quest event goals. This can result in solutions which reuse more of the training data. This technique for re-using more of the demonstration data was sensitive to the parameters chosen for w_1 and ϵ^T, so we did not include it in our experimental analyses, but figures illustrating the possibilities are shown below in Figure 7.4. A similar strategy of expanding heuristics which make more progress toward the goal is investigated in [49], but the technique described in this paragraph is different in that a heuristic which *reuses training data* is expanded more often. We leave it to future work to perform a thorough analysis of this technique.

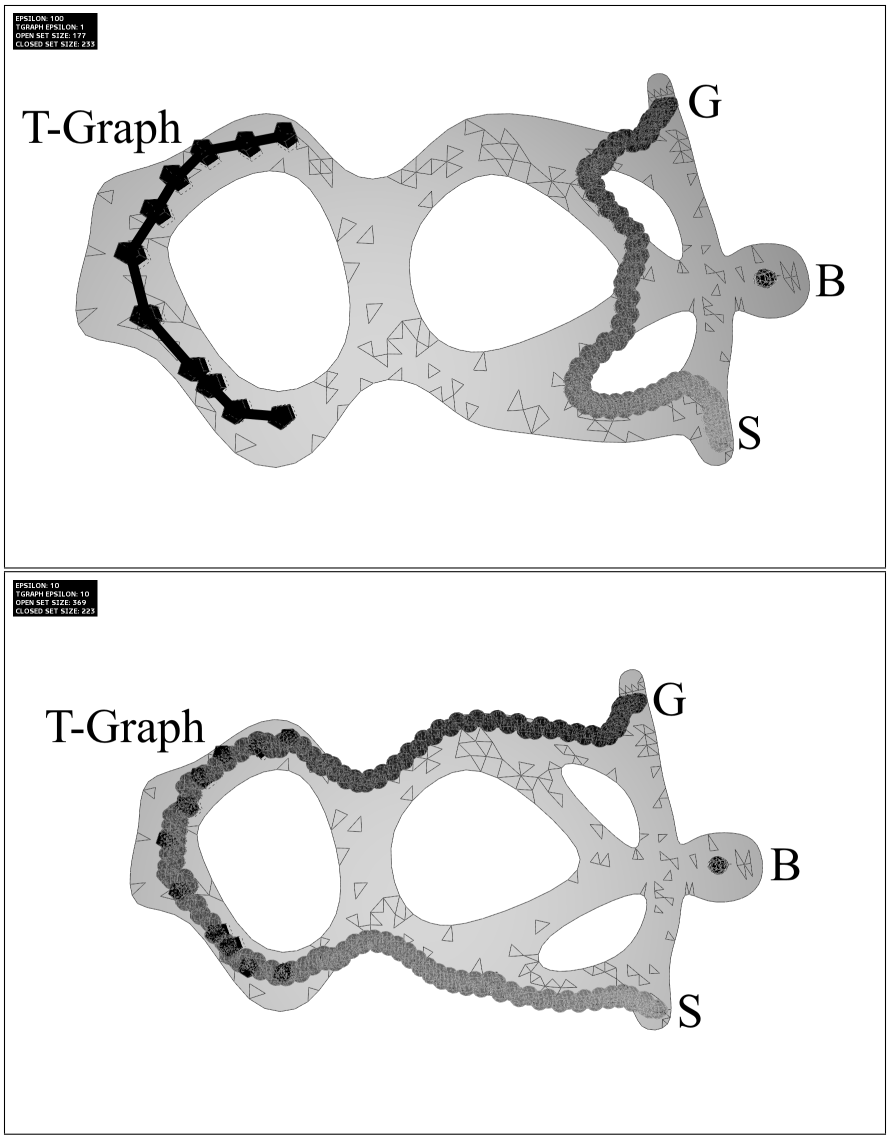

Figure 7.1: This figure shows a cave scenario with an unbeatable killer bear in its den labeled 'B'. The starting position for our NPC is labeled 'S' and its goal location is labeled 'G'. The T-Graph is the black segmented line on the left and the T-Graph heuristic with Weighted A* search was used to generate these solutions. The solution is the thick dotted line which connects 'S' to 'G'.
The top image shows a solution when $\epsilon^T = 1$, which is the same as the ordinary Weighted A* solution for the same suboptimality bound. When $\epsilon^T = 1$, the T-Graph has no influence on the search process. However, when ϵ^T is increased, the search produces a solution which mimics the training behavior encoded in the T-Graph. In the bottom image, $\epsilon^T = 10$, which guides the search to take the long way through the left side of the cave, following the T-Graph which runs through the same area.

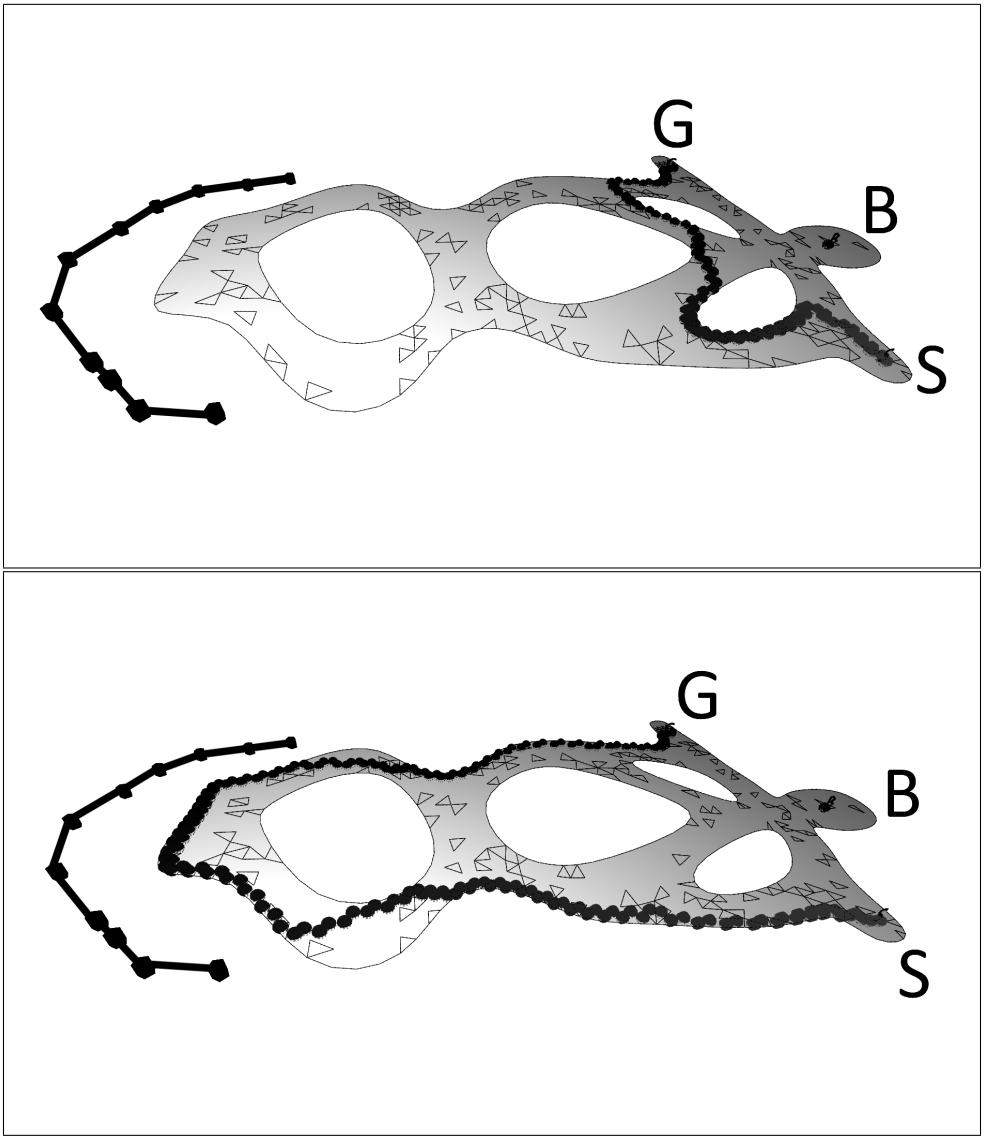

Figure 7.2: This figure shows a cave scenario with an unbeatable killer bear in its den labeled 'B'. The starting position for our NPC is labeled 'S' and its goal location is labeled 'G'. The T-Graph is the black segmented line on the left and the T-Graph heuristic with Weighted A* search was used to generate these solutions. The solution is the thick dotted line which connects 'S' to 'G'.
In this case, as opposed to Figure 7.1, the T-Graph was recorded in an environment similar but different to the cave environment being used for planning now. The top image shows a solution when ϵ^T is set to a low value, too low to influence the search solution. However, the bottom image shows that when ϵ^T is increased, the search produces a solution which mimics the training behavior encoded in the T-Graph, even though the T-Graph is not directly accessible in this search space.

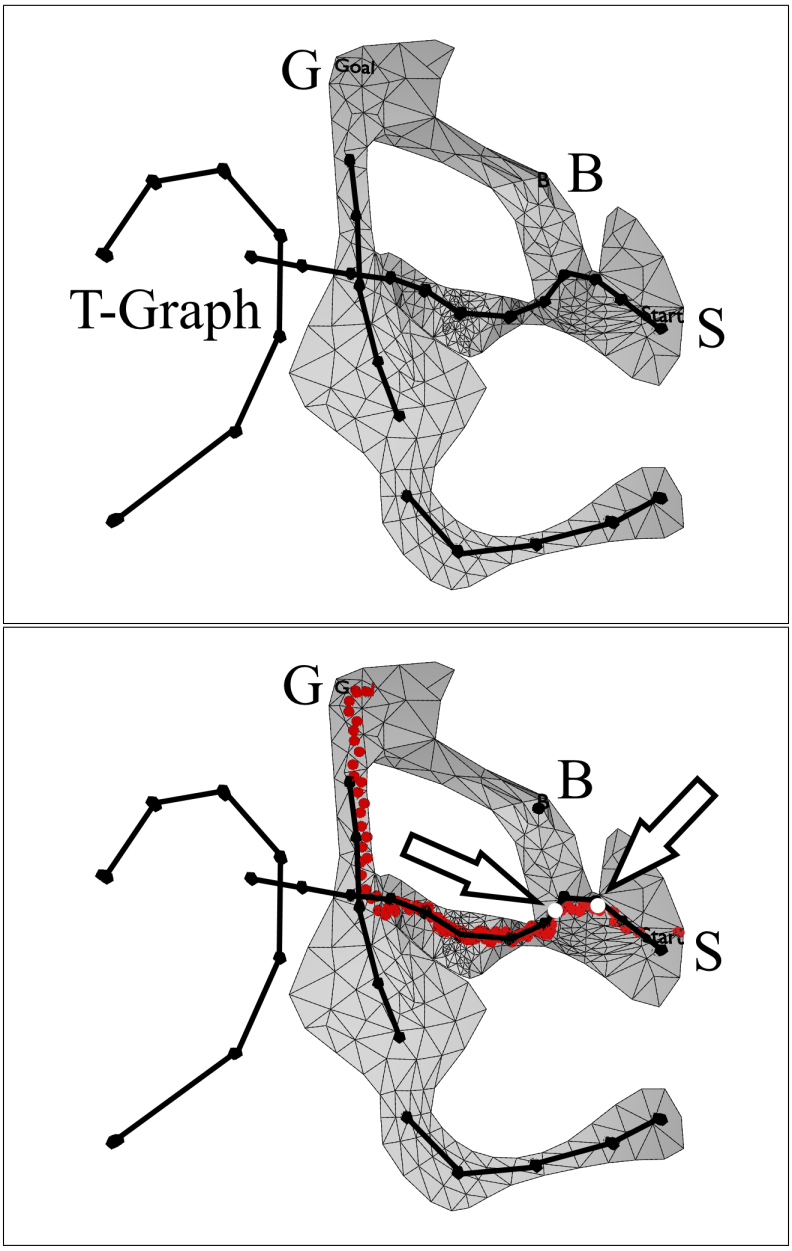

Figure 7.3: This figure shows another scenario with an unbeatable killer bear labeled ‘B’. The starting position for our NPC is labeled ‘S’ and its goal location is labeled ‘G’. The T-Graph is composed of multiple separated T-Graph components (the segmented black lines), some of them relevant and some of them irrelevant. Moreover, one of the T-Graph components demonstrates a dangerous maneuver moving within the killer bear’s ordinary detection radius. In the solution (marked in red), the planner discovered that it could mimic the training data most closely in this region and also avoid the killer bear by slowing down and sneaking past the bear. The arrows and white dots indicate the beginning and end of the sneaking segment.

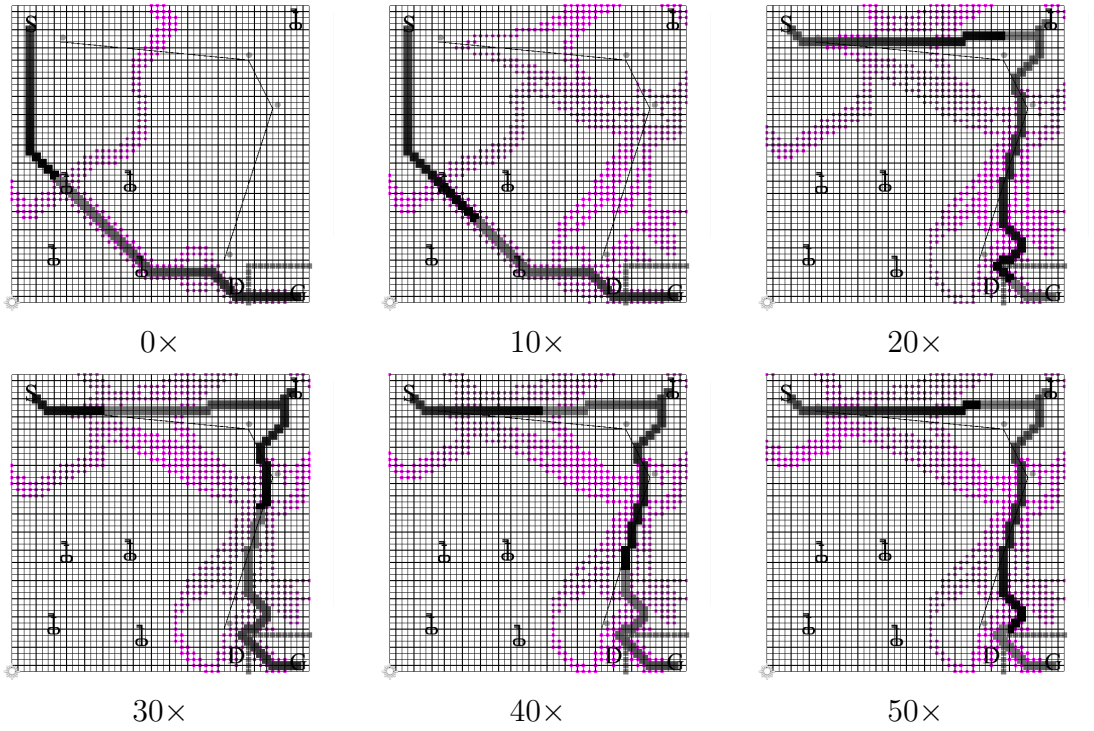

Figure 7.4: *5 Keys 1 Door* scenario (MHA*, $w_1 = 100$, $w_2 = 1$, $\epsilon^T = 100$) with the T-Graph heuristic is used as one of our MHA* inadmissible heuristics. Depicted are solutions when the T-Graph heuristic is expanded 0, 10, 20, 30, 40, and 50 extra times per MHA* expansion. This can enable better training data re-use in MHA*. For $0\times$ and $10\times$, other heuristics manage to guide the search to the goal before the T-Graph is explored, but for $20\times$ and above, the search zooms along the T-Graph and encounters a key to open the door near to the T-Graph before the ones which are farther away.

7.1.1 User Study

We conducted a small user study to evaluate the qualities of our results as perceived by human subjects. Since our work can be applied to e.g. cooperative planning (see Chapter 6), it is important that the NPC behavior appeal to human players.

We invited participants to play a simple video game of our design. The game presents a 2D environment to the player. The goal of the game is to navigate the player's character through the environment to reach a goal position. The player uses the computer keyboard arrow keys to navigate. In each game scenario, there are several obstacles in the environment blocking progress to the goal. The player and NPC must both work to overcome these obstacles and must both reach the goal to proceed. In order to keep participation time around 20 minutes, we presented two scenarios to each participant. The first scenario is the 5 Keys 5 Doors In Series scenario we used in our computational performance results (see Figure 7.9). For the second user study scenario, we designed a new scenario (we will call it the "Bear Lever" scenario) required cooperative behavior to complete (see Figure 7.5). In the Bear Lever scenario, there is a killer bear guarding a lever. The lever must be activated to open the doors in the level which block the path to the goal. If the player walks into the bear's vicinity, the bear will attack. However, the NPC has a special ability enabling it to walk past the bear undetected.

Four NCP types were used: Weighted A* (w=1000), T-Graph ($w = \sqrt{1000}$,$\epsilon^T = \sqrt{1000}$), MHA* with our Quest Event Heuristics ($w1 = 1000, w2 = 1, \epsilon^T = \sqrt{1000}$), and a rule-based NPC which moves to a new position near the player every couple of seconds. Both scenarios can be solved by all four NPC types, however the solution to the Bear Lever scenario for the rule-based NPC may not be obvious to some participants. At the moment each trial is started, the planner-based NPCs begin their planning process on a new thread and when it completes, the NPC begins to execute its planned behavior. The

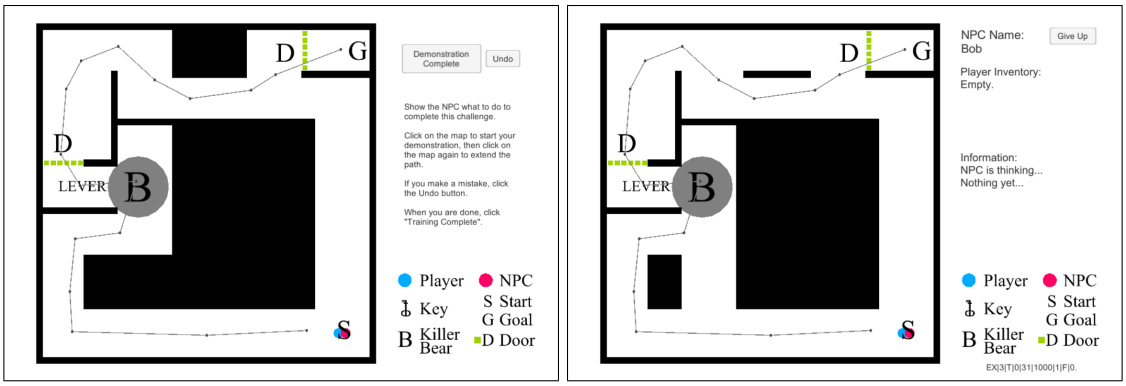

Figure 7.5: Bear Lever scenario on demonstration screen with a demonstration visualized, and in game (note additional pathways opened).

player can move the player character immediately, even while the planned NPCs are still planning.

First, we show participants an empty environment with no obstacles, so that they can become accustomed to the controls. Then we begin the first section of game trials using the 5 Keys 5 Doors In Series scenario. This scenario is repeated four times, once with each NPC. After each of these times, we presented participants with a short questionnaire asking several questions about the NPC's behavior. Then, the Bear Lever scenario is shown on the screen. We explain that the participant must draw out a line on the screen as a demonstration for the NPC. We told them to draw the demonstration the same way they might choose to draw a line on a map while showing another human directions. Once they were satisfied with their demonstration, the Bear Lever scenario was presented four times to the player, once with each NPC type. In-game, two sections of the Bear Lever obstacles are removed so that participants can see whether each algorithm exactly re-uses the demonstration path or takes a shortcut (see Figure 7.5). After each of the Bear Lever trials, the questionnaire is presented to the participant, and it includes additional questions about how well the participant thought the NPC followed the provided demonstration.

The questionnaire items included these statements:

- Q1: The NPC's behavior resembles behavior that a human player would exhibit in the same situation.
- Q2: The NPC took too much time (including time spent standing still) for a video game.
- Q3: The NPC's behavior was reasonable for a video game.
- Q4: The NPC's behavior was efficient.

The statements below only appeared for the Bear Lever trials.

- Q5: The NPC followed the demonstration.
- Q6: The NPC should have followed the demonstration more closely.
- Q7: The NPC should have followed the demonstration less closely.

Participants were asked to respond to each statement on a 5-point Likert scale [50] indicating how much they agreed with the statement. The choices available were "Strongly Disagree," "Disagree," "Neutral," "Agree," and "Strongly Agree." We also asked participants to write down any additional comments they had about the NPC behavior or about the game. The experimenter wrote down any spoken comments made by participants while playing.

The questionnaire choices ("Strongly Disagree," "Disagree," etc.) were given attitude scores -2, -1, 0, 1, and 2, respectively. We grouped the results by NPC type and then computed average attitude values for each statement, shown in Table 7.1.

Our MHA* with Quest Event Heuristics approach had the highest attitude rating for Q1 (human-like behavior), the lowest rating for Q2 (excessive time spent), the highest rating for Q3 (reasonability of behavior for a video game), and the highest rating for Q4 (behavior

efficiency). This NPC inspired the most positive comments from participants, e.g. "I like this one!"

The T-Graph approach had the best responses for demonstration re-use. The T-Graph approach achieved the highest attitude rating for Q5 (following the demonstration) and the lowest attitude rating for Q6 (the need to follow demonstration more closely). Some participants tried to demonstrate actions to the rule-based NPC (not knowing how it worked internally) by moving their own character around on the screen. One participant commented aloud while playing "What are you thinking? Let me show you what you need to get."

Statement Q7 (demonstration should have been followed less closely) was given in an attempt to understand when participants thought that demonstration was followed too closely, at the cost of something else (path optimality, human-like behavior, etc.). Sometimes participants gave both Q6 and Q7 "Disagree" or "Strongly Disagree" responses, indicating that they thought the algorithm followed the demonstration with approximately the right level of precision. This happened most often for the T-Graph NPC. The MHA* with Quest Event Heuristics approach had the highest attitude rating for Q7, indicating that participants thought it followed the demonstration too closely. This is a strange result, because MHA* solutions will only follow the demonstration after reaching the quest-event (here, the lever), whereas T-Graph solutions follow the demonstration wherever possible. This was noticeable to participants; one commented that because the MHA* NPC followed the demonstration closely part of the way but also left the demonstration for a more optimal path in another part of the level that there was a "good balance" in its behavior. Another participant hesitated while answering Q7 because it confused them; there may have been confusion among several of the participants and perhaps Q7 should have been worded differently.

Trials with the rule-based NPC took the longest to solve, averaging 67.3 seconds, com-

pared to an average 43.6 seconds for the planned NPCs. Participants gave this NPC the least favorable responses to Q2, indicating that they thought the NPC took too long. However, one participant remarked that this NPC "is responsive." Another participant commented that it was fun to lead this NPC through the game. This NPC attracted the lowest attitude rating for Q1 (human-like behavior), though one participant remarked that it was like leading a human child. One participant wrote in "less intelligent" on the questionnaire page. A participant commented on the autonomy of this NPC: "[the rule-based NPC] could be used as a companion for searching surroundings but not for following demonstrations and doing remote tasks." This participant also mused that "the perfect [NPC] could be a mix of [MHA* NPC] and [Weighted A* NPC]," marking positive attitude responses for the MHA* NPC's quick solution time and for Weighted A* not following the demonstration exactly.

Participants indicated that NPC "thinking" (computation) time was an extremely important factor for their enjoyment of the experience. One participant commented that he was starting to feel competitive and wanted to beat his own completion times. Another said that time was "all that matters to me." We reveal completion times in the game's user interface, so this may have communicated a sense that the game's objective was to finish it as quickly as possible.

One participant commented on the particular ways that the NPCs moved. Human players, because of the arrow key controls, generally only move in the cardinal directions or at 45 degree angles diagonally, however the NPCs can trace out paths through the game world at other angles. This participant also noticed a zig-zag movement in one of the T-Graph solutions and he asked if this was "necessary." Most participants had nothing to say about the details of NPC movements.

NPC	Time	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Weighted A*	43.5	0.5	0.4	0.8	0.6	0.3	-0.2	-1.0
T-Graph	44.6	0.6	0.5	1.1	0.8	1.7	-1.3	0.0
MHA*	42.8	1.5	-0.9	1.4	1.3	1.2	-0.7	0.2
Rule-based	67.3	-1.3	0.7	-0.8	-1.2	-1.3	1.5	-1.5

Table 7.1: Questionnaire response scores by NPC type. The MHA* NPC utilized our Quest Event Heuristics.

7.2 Computational Performance

We compared the computational performance of Weighted A*, our T-Graph method, and our MHA* method using T-Graph inadmissible heuristics. We computed parameters for each of the methods so that we could compare results for suboptimality bounds 1, 10, 100, 1000, and 10000. To illustrate computational performance differences between our three test algorithms and to expand on the results in [51], we authored a handful of new test environments and quests. Each algorithm was used to solve each scenario for each of the test suboptimality bounds. These trials were repeated 10 times and the results were averaged. Results show that our MHA* method using T-Graph heuristics, for sufficiently high bound settings, can find a solution in fewer state expansions than both the Weighted A* and T-Graph methods. This, in turn, results in better computation times, especially as world model and search branching factor become more complex. Despite the higher suboptimality bounds, path costs remain similar across all trials.

7.2.1 Parameter Selection

To ensure a fair comparison of the three algorithms, we started by determining how to configure the parameters of each algorithm for good performance across the experiment scenarios. The suboptimality bound for Weighted A* is determined by its w parameter, so w was set to match the bound. The T-Graph and MHA* methods each have two parameters

determining the suboptimality bound, so we tested several ways to balance these parameters. While performing these tests, we discovered several interesting properties of these algorithms.

The T-Graph method's two parameters are w and ϵ^T. As discussed in Section 4.3, w biases the search to follow the heuristic function (which here is the T-Graph heuristic), and ϵ^T affects how much the T-Graph heuristic function biases re-use of training data. The original E-Graph authors suggested a ratio of 2:10 between ϵ (the same as our w) and ϵ_E (like our ϵ_T) [3]. Although the T-Graph method is inspired by the E-Graph heuristic, our formulation is different enough that we could expect a different optimal distribution of these parameters. T-Graph parameter balance test results for suboptimality bound 10 are displayed in Figure 7.6 and results for bound 100 are displayed in Figure 7.7. From these results, we concluded that the best parameter balance to use in our computational performance experiments for the T-Graph method was approximately a 1:1 ratio between the values of w and ϵ^T. When there is a 1:1 ratio, both parameters are set to the square root of the suboptimality bound, since they are multiplied to determine the bound.

MHA* has two parameters, w_1 and w_2, which affect the suboptimality bound on solutions. As discussed in Section 3.4, w_1 acts something like w does in Weighted A*, and w_2 can cause the search to utilize the inadmissible heuristics more. Initially we computed parameter test results using the MHA* with T-Graph heuristics algorithm as it is described in [51] and found that there could be very dramatic performance changes with very small changes in the parameters. This turned out to be a symptom of the inadmissible heuristics being poorly calibrated relative to the anchor heuristic. This problem was addressed in [44] with the Improved Multi-Heuristic A* (Improved MHA*) algorithm, but we chose here to scale our inadmissible MHA* heuristics to better match up with the anchor heuristic. The matter of calibrating our inadmissible heuristics is discussed in more detail in Section 3.4.1. Parameter balancing test results for suboptimality bound 10 are shown in Figure 7.8. Re-

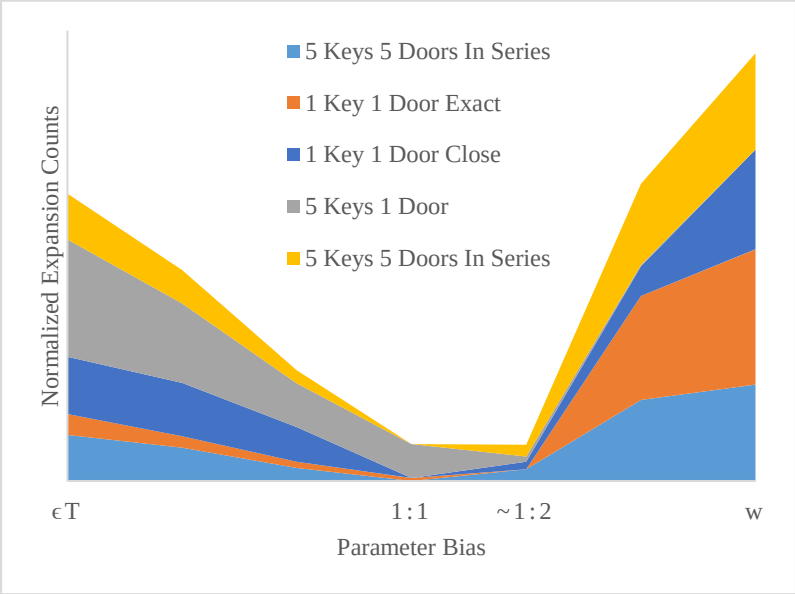

Figure 7.6: This stacked area plot shows that in general, the best parameter ratio between w and ϵ^T is around the range of 1:1 - 1:2. Each test scenario's plotted area is determined by expansion counts normalized so that each scenario type has the same area on the plot.

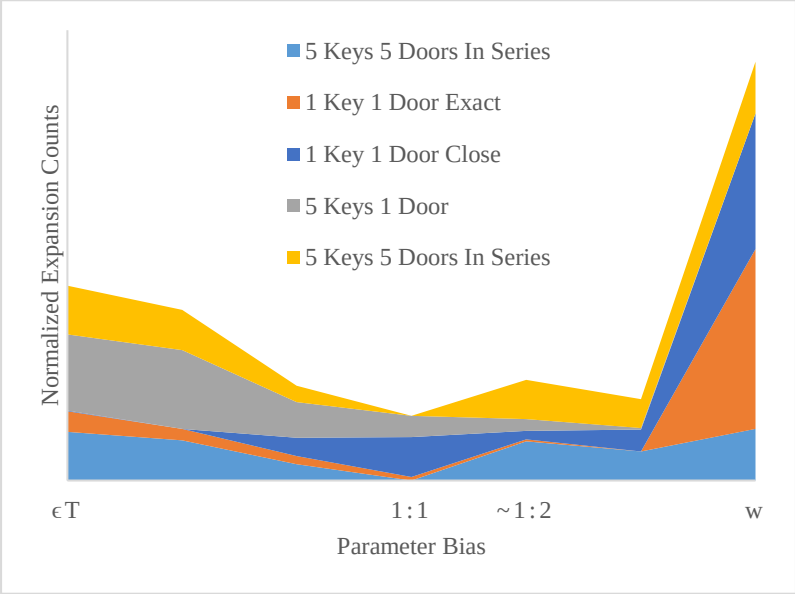

Figure 7.7: This plot is like Figure 7.6, but shows the results for suboptimality bound 100. Results for higher suboptimality bounds follow a similar trend.

sults for larger suboptimality bounds all showed the same trend, so they are omitted here. From these results, we concluded that the best parameter balance to use in our computational performance experiments for the calibrated MHA* with T-Graph heuristics method was to set w_2 to 1.0 and set w_1 to the suboptimality bound. We expected the parameter ratio to be something more balanced, like the 1:1 ratio found for the T-Graph method, but we believe that our w_1-dominated balance is a consequence of the inadmissible heuristics being calibrated as we discussed above. w_2 is effectively useless, and in keeping with this is the observation that Improved MHA*, which also solves the heuristic calibration problem, has only one parameter affecting the suboptimality bound.

Since we use the T-Graph heuristic in our MHA* inadmissible Quest Event heuristics, we actually have a third parameter, ϵ^T, in MHA* trials. However, it has no effect on the suboptimality of solutions, so we simply set ϵ^T to the square root of the suboptimality bound as done for the T-Graph-only trials so that these results could be directly compared to T-Graph-only results.

7.2.2 Computational Performance Results

We ran experiments with all combinations of these parameter configurations for scenarios, branching factors, and suboptimality bounds:

Scenarios:

- 1 Key 1 Door Exact, representing a simple quest task with a locked door, key, and training data which shows exactly how to get the key on the way to the door. (Figure 7.9)

- 1 Key 1 Door Close, the same task as 1 Key 1 Door Exact, but the training data only goes near to the key on the way to the door. (Figure 7.9)

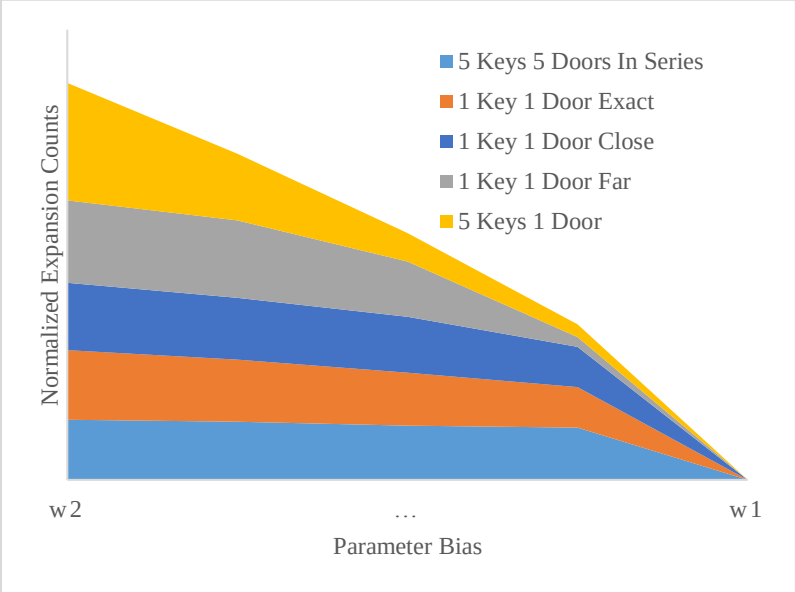

Figure 7.8: This stacked area plot shows that in general, the best parameter distribution between w_1 and w_2 is to set w_2 to 1 and set w_1 to equal the suboptimality bound. At the left side of this plot, w_2 is given a value much larger than w_1 and at the right side of the plot, the opposite is true. Each test scenario's plotted area is determined by expansion counts, normalized so that each scenario type has the same area on the plot. The results shown here were computed for suboptimality bound 10, and the results for larger suboptimality bounds all showed the same trend.

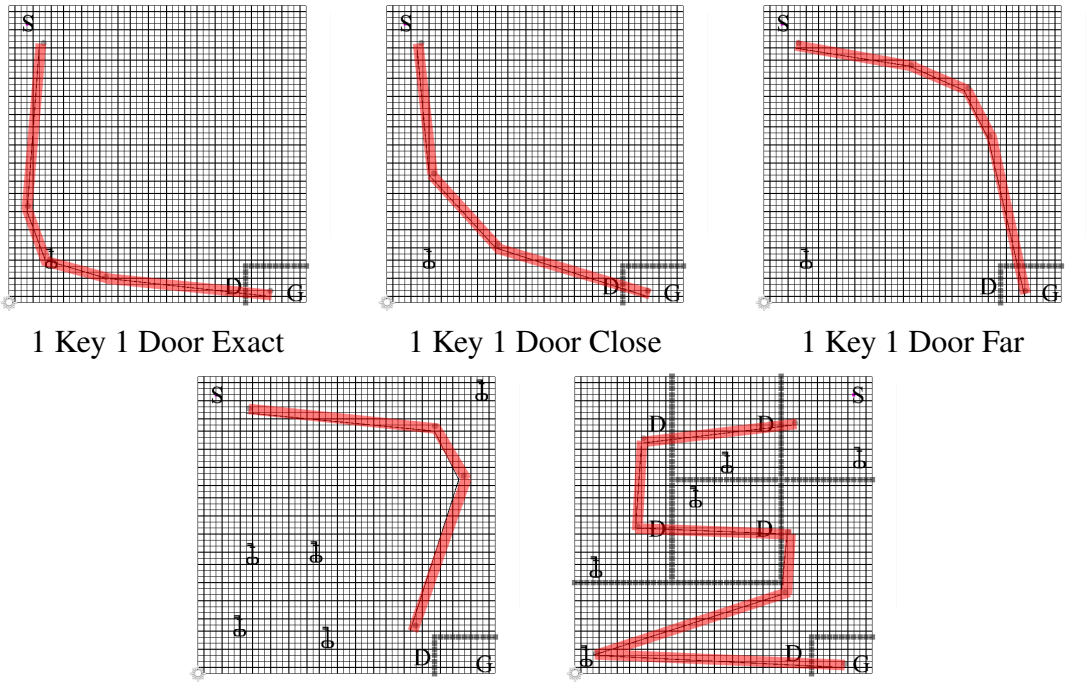

Figure 7.9: Scenarios used in new computational performance results. T-Graphs in red.

- 1 Key 1 Door Far, the same task as 1 Key 1 Door Close, but the training path is much farther from the key. (Figure 7.9)
- 5 Keys 1 Door, a similar task to those above, but with multiple keys which all unlock the door and a training path passing by one of the keys. (Figure 7.9)
- 5 Keys 5 Doors In Series, a more complex task requiring each of five rooms' keys to be found before advancing further toward the goal. (Figure 7.9)

These scenarios were designed to expand on the situations already tested in [45] and[51]. They represent the approximate complexity level of quest sub-goals which might be encountered in a game like Skyrim.

Search Branching factors: 16, 256 (increased by adding dimensions to the world state and altering the successor generation functions)

Suboptimality Bound Factors: 1, 10, 100, 1000, 10000

These results were computed on a Pentium i7-860 machine with 12GB of RAM. For these particular results, our algorithm was implemented to run on a single thread in C# for compilation with and execution in the Unity3D[52] engine. Compiled here are plots showing the expansion counts, computation times, and solution costs for all of the experiments. Plots are grouped together in Figures by scenario type.

Gaps in any of the plotted lines represent search failures due to exceeding a 30 second search time limit. 30 seconds was chosen because it approximately matches the loading time of video game environments – if a search can succeed in 30 seconds, the planned NPC has a chance to begin executing its behaviors once the human player is ready to join it. Lower computation times are always better, and if they are low enough, one could imagine re-planning NPC behavior multiple times during the play-through of a quest to account for unexpected changes to the game world caused by the player, or to improve solution quality in an anytime replanning framework.

Solution Costs

Solution plan costs did differ between choices of algorithm and suboptimality bound, however the highest cost solutions never exceeded $2\times$ the optimal cost, even when the suboptimality bound was as high as 10000 (meaning that in the worst case, the solution cost could be $10000\times$ larger than the optimal solution). Much of the extra cost in our solutions is due to the influence of demonstration data, which is a a desired property. Our goal is not to find optimal solutions, but solutions which follow provided demonstrations. Some of the greatest deviation in solution cost (compared to the optimal cost) was found with the T-Graph method for large suboptimality bounds in the 1 Key 1 Door Far scenario ((Figure 7.12). This example was specifically designed to draw the search far out of the way of the optimal path. A highly suboptimal cost was also seen in the 5 Key 1 Door scenario (Figure 7.13) for

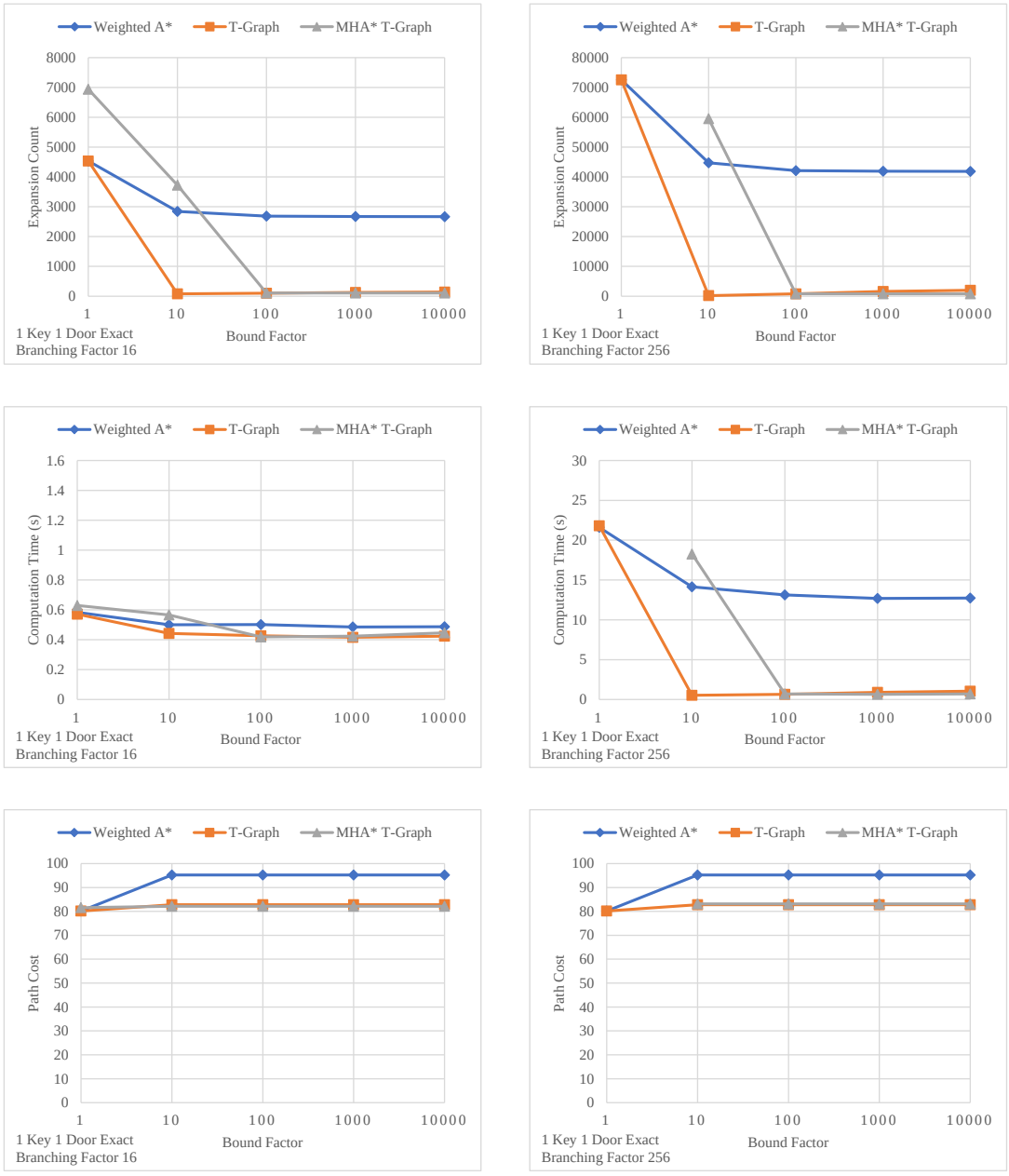

Figure 7.10: Performance results for 1 Key 1 Door Exact scenario. When the T-Graph exactly demonstrates a solution, the T-Graph results are the best when the suboptimality bound is greater than 1, however this situation is unlikely to be found in a video game between different quests.

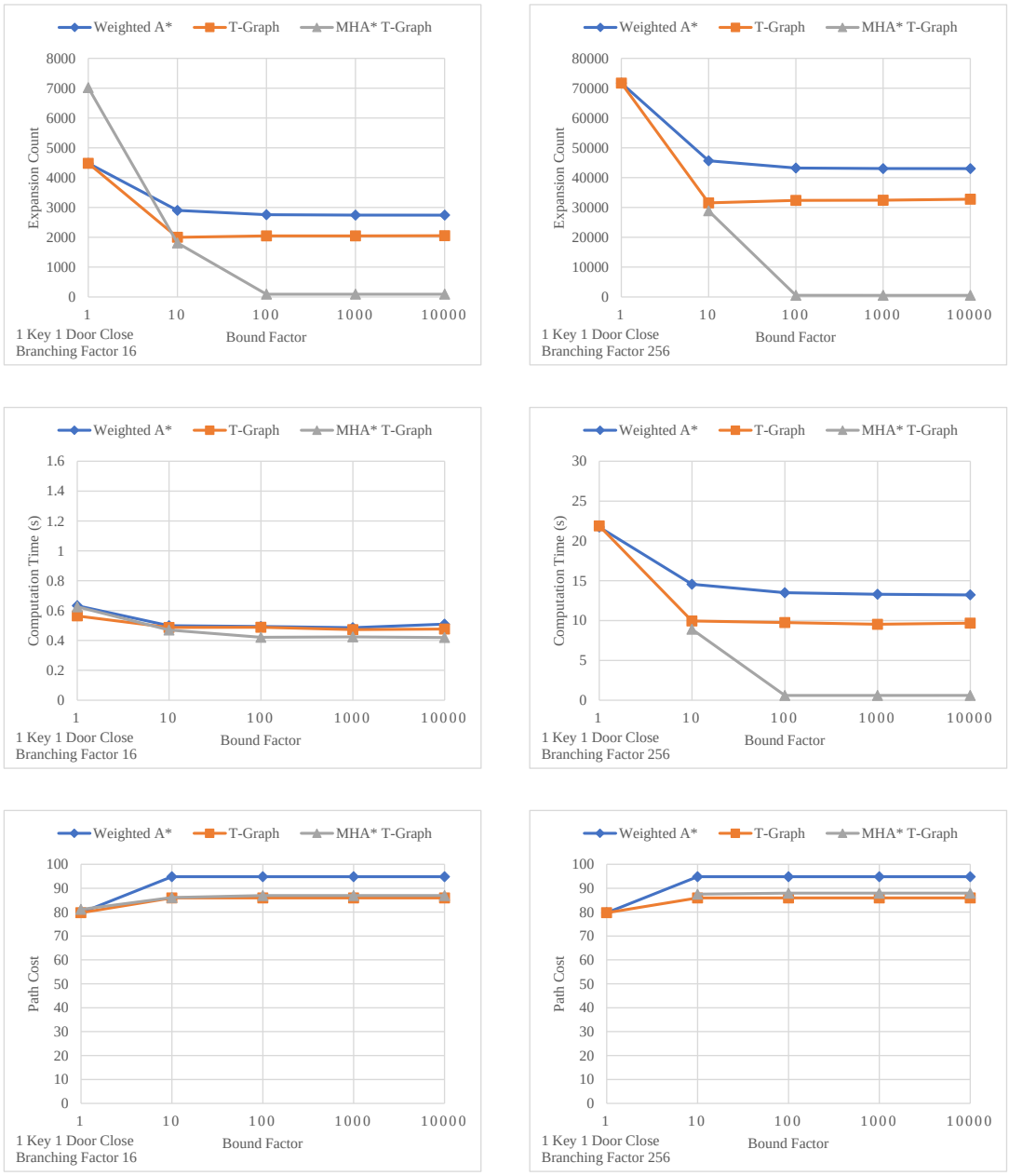

Figure 7.11: Performance results for 1 Key 1 Door Close scenario.

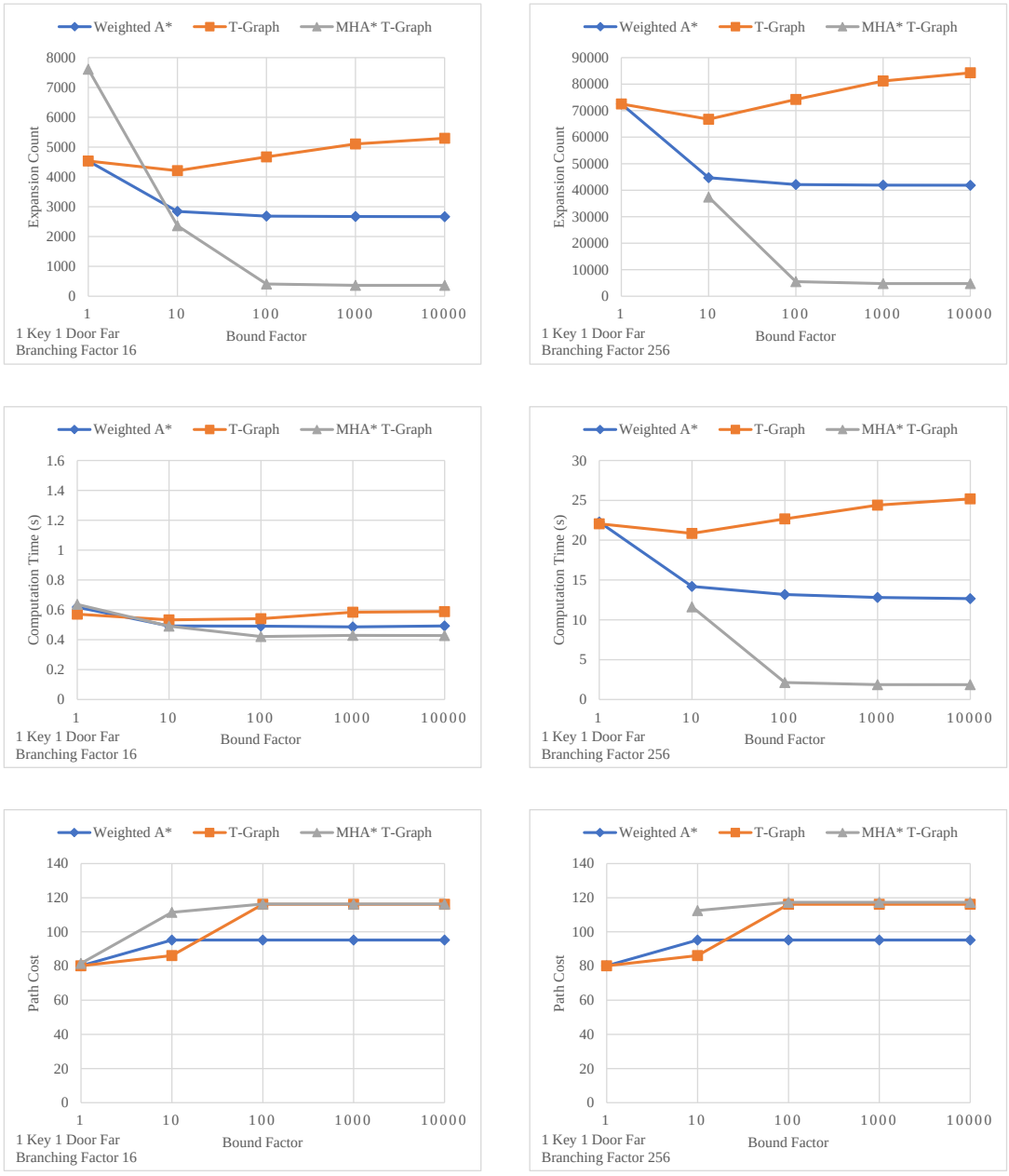

Figure 7.12: Performance results for 1 Key 1 Door Far scenario. When the training data strays far from the optimal solution, performance of the T-Graph method suffers even compared to Weighted A*. Our MHA* with T-Graph Heuristics approach outperforms the other methods.

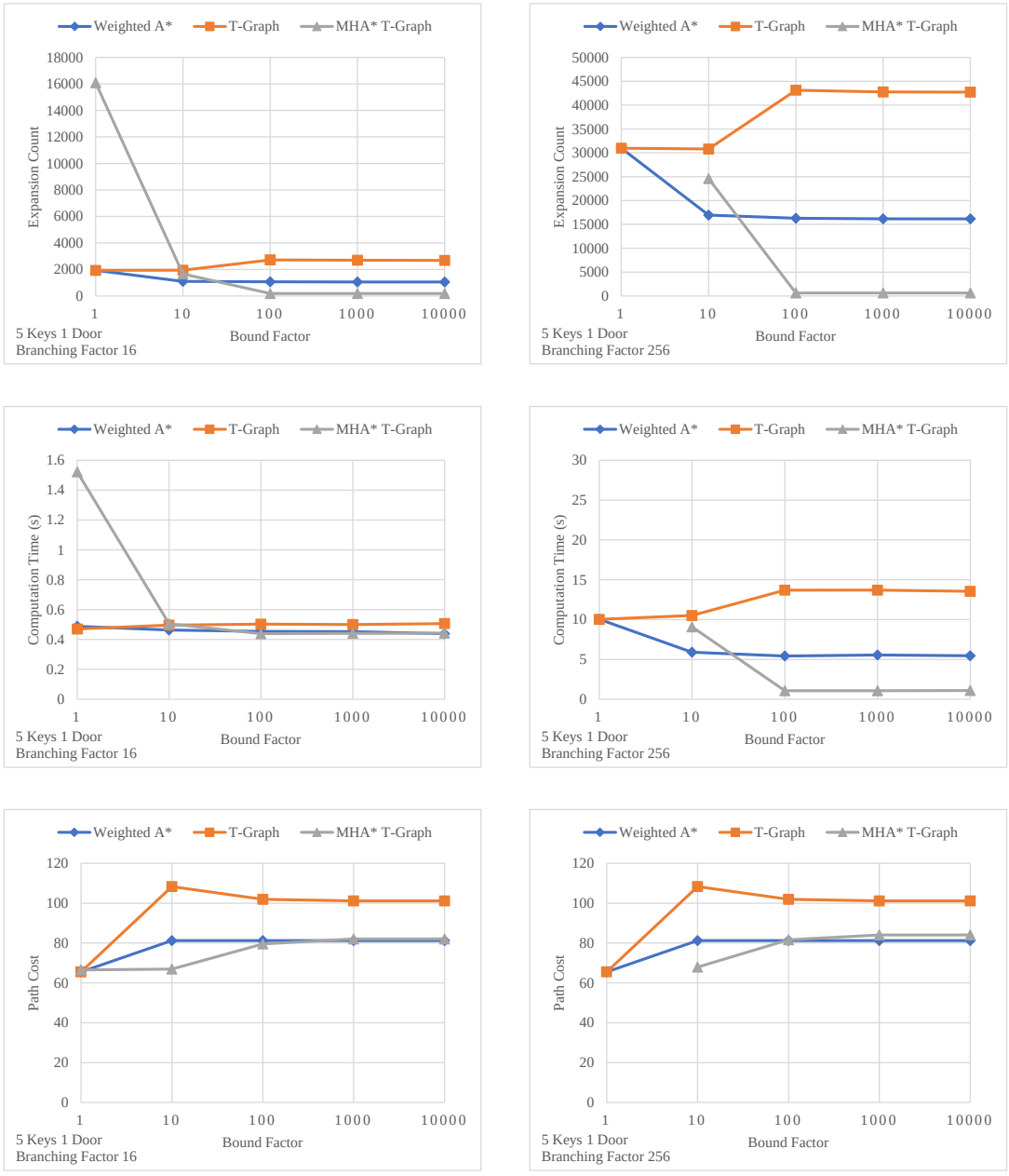

Figure 7.13: Performance results for 5 Keys 1 Door scenario. Since the goal in this scenario can be reached by several different behaviors, for low suboptimality bounds, the Weighted A* and T-Graph methods naturally find a solution quickly compared to MHA*, which spreads out its efforts. However, for higher bounds, MHA* performance is superior, even in the trials with a lower branching factor.

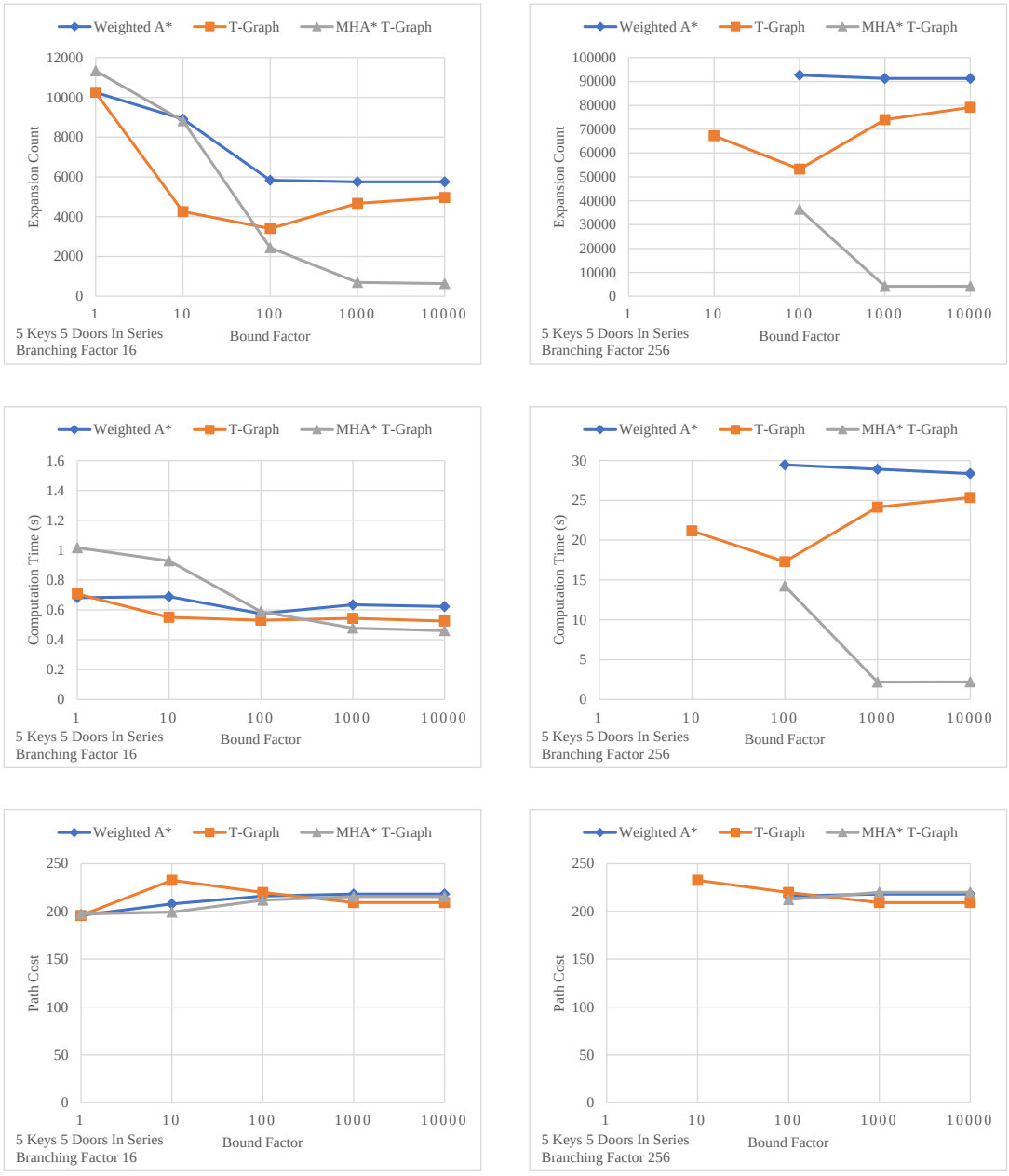

Figure 7.14: Performance results for 5 Keys 5 Door In Series scenario. In this scenario, A* fails across the board when the branching factor is increased. Our methods, however, are capable of succeeding and our MHA* with T-Graph Heuristics method does particularly well even compared to T-Graph.

the T-Graph search with suboptimality bound 10. In this scenario, five keys can all open the door at the goal and the T-Graph guides the search toward a key which is far out of the way. There is another key nearly on the optimal path, so the optimal solution is to pick up that key instead. Our search methods instead produce the alternative (but longer) solution according to the desired behavior encoded in the demonstration. Our MHA* approach tended to yield solution costs similar to T-Graph.. However, the computational performance advantage of the MHA* approach makes it the most appealing algorithm among the three for solving these kinds of NPC behavior generation problems.

The Effect of Demonstration Quality

Our results for the 1 Key 1 Door "Exact," "Close," and "Far" scenarios (Figures 7.10, 7.11, and 7.12) reveal the effects of demonstration quality on computational performance. Weighted A*, since it does not utilize the demonstrations, performs the same in all of these results. However, the T-Graph algorithm exhibits its best performance when the "Exact" demonstration is provided, since the search can proceed directly along the T-Graph from start to goal. T-Graph performance is worst in the "Far" scenario, when the demonstration strays far from the key necessary to reach the goal. In this case, the search process is lead astray and needs to wastefully expand many states near the T-Graph before finding the key. Our Quest Event (T-Graph) heuristics in MHA* fared better, since the search is guided to the key despite the poor demonstration. For higher suboptimality bounds, MHA* computation time results were better than the T-Graph results, except when the demonstration exactly passed over the key (see the computation time plots in Figure 7.10).

The Effect of Demonstration Size

To test the effect of T-Graph size on computation time, we selected the 1 Key 1 Door Close scenario and tried subdividing the T-Graph's edges into additional pieces. We only

subdivide the line segments: T-Graph still traces out the same shape and still has the same overall cost structure, so it still encodes the same demonstration and the search completes in the same number of expansions. The computation of the T-Graph heuristic visits each T-Graph edge once per heuristic request, so for searches with the same number of expansions, we expect a linear increase in search times as the number of T-Graph edges increases. In Figure 7.15, we plot the computational slowdown caused by additional T-Graph subdivision amounts up to 200. At 200, each edge of the T-Graph has gone from having 1 component to having 201 components, so the T-Graph is 201 times larger. The 1 Key 1 Door Close scenario has 3 edges initially, so at 200 additional subdivisions, it has 603 edges which each must be visited while computing the T-Graph heuristic, yielding a 7.9$\times$ overall search time increase. If a player is providing a demonstration by drawing it out on a map of the game environment, as we tested in our user study (Section 7.1.1), they are unlikely ever to use so many segments. A user interface enabling demonstration authoring like this could be programmed to limit the number of segments available, putting a cap on the computational slowdown caused by demonstration segment count. Alternatively, demonstration paths can be downsampled, at the risk of losing some detail in the demonstration which might be important.

The Effect of Additional Quest Events

We tested the relationship between quest event count and computation times for our MHA* with T-Graph Heuristics approach. Since each quest event means the addition of another heuristic to the search, we expected that computation times would slow down in proportion to the number of uninformative quest events. We tested the addition of 1 to 50 randomly-positioned quest events and repeated these tests to average results. We computed these results for both the 1 Key 1 Door Close scenario and the 5 Keys 5 Doors In Series scenario, to see if the scenario structure would make a difference. As seen in Figure 7.16, it did

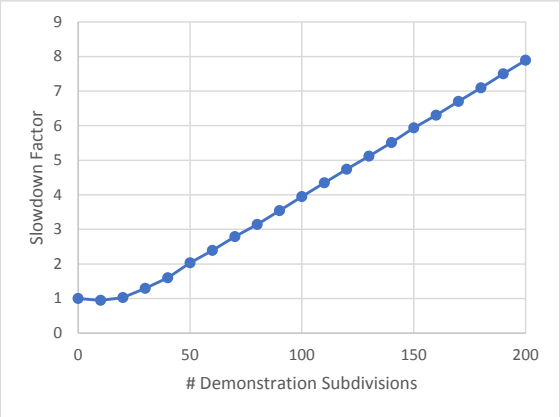

Figure 7.15: Effect of demonstration size on computation times.

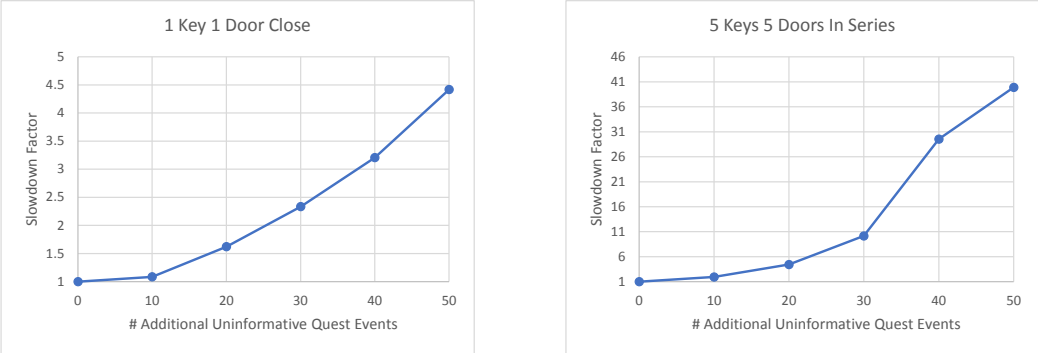

Figure 7.16: Effect of uninformative quest events on computation times.

make a difference. Adding uninformative quest events dilutes the helpful guidance of the informative heuristics, degrading search performance. The effect is more pronounced in some quest and environment types, and gets worse as the number of additional quest events increases. These results show that it is important to limit the number of uninformative quest events used. If instead of using random positions, additional quest-event heuristics are added by mimicking the existing quest event heuristics in a scenario, the slowdown observed is much smaller, and seems to increase in a linear manner (see Figure 7.17). In fact, in the open environment of the 1 Key 1 Door Close scenario, duplicating the key quest event actually improves performance until the number of duplications exceeds 50. This is because the additional heuristics guide the search toward the key faster.

7.3 NPC Skill

We investigated the use of rating systems such as ELO[53], Glicko[54], Glicko-2[55], and TrueSkill [6], to learn a rating of the relative difficulties of different scenarios and the relative skill levels of planning algorithms. Normally these systems are used to rate human game players when playing games against one another.

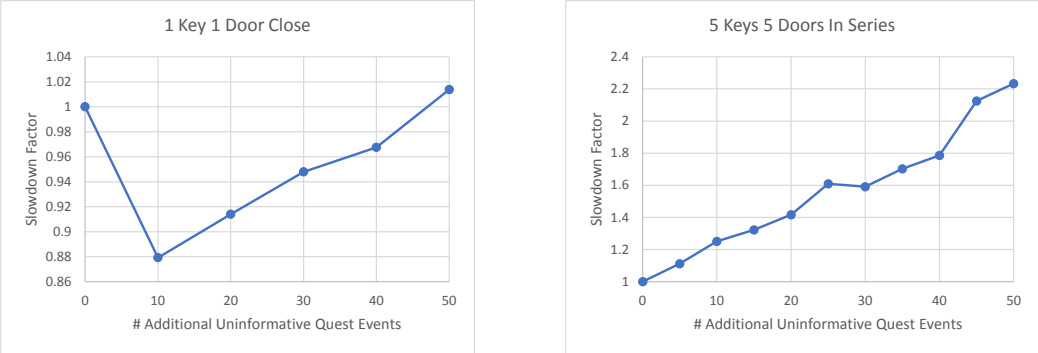

Figure 7.17: Effect of duplicated quest event heuristics on computation times.

We borrow an idea from [56], where a rating of level difficulty and player skill is learned by treating each level and player as opponents to one another: if the player can solve the problem presented by a level, the player is counted as the winner, but if the player becomes stuck, the level is counted as the winner. In this context and also in ours, players (NPC behavior generators) do not play against one another and levels (scenarios) cannot play against one another. Despite this, rating systems produce rating values which can be used to compare all entities.

Rating systems have been used to rate the skill levels of competitive computer-controlled video game players (bots). In [57], ELO is used to rate game AI bots in matches against other bots in the real-time strategy game StarCraft. That work also utilizes a proprietary rating system named "SSCAIT rank"[58], based on the "ICCUP ranking system"[59] used for rating competitive human StarCraft game players. Another bot competition, The Student StarCraft AI Tournament (SSCAIT)[58], also uses ELO ratings to compare bot performance.

A simple rating function for our analysis could be devised based on the numbers or percentages of successful NPC behavior plans (under a time limit, as in Section 7.2.2). In [60], the problems using this kind of "Empirical Winning Rate" (EWR) are discussed.

EWR does not take into account the difficulty of a particular opponent; winning against a strong opponent should count for more than winning against an easy opponent. We chose here to use the TrueSkill rating system to rate the relative skill levels of three NPC behavior generation algorithms.

We began by using the 30 second time limit success and failure results from Section 7.2.2, but found that in some cases, methods which differed greatly in performance were rated the same. For example, for the 5 Keys 5 Doors In Series scenario results (Figure 7.14), both the Weighted A* and MHA* With T-Graph Heuristics methods have the same numbers of successes and failures. This produced approximately the same rating for these two algorithms, but the MHA* approach had significantly faster computation times.

TrueSkill can take into account "partial play" information about how much time a player spent in a competition. We tested using this as a means to break rating ties when one algorithm finished faster than another. We indicated to TrueSkill the fraction of the 30 second time limit that the algorithm used to generate its plan as its partial play time. Because of how TrueSkill uses partial play information, this did not work as a means to break rating ties. For example, imagine two teams of two human players each competing in a real world physical sport. Half of the way through a match, one of the players on Team 1 suffers an injury and sits out the rest of the match. If the one-player team wins against the two-player team despite its disadvantage, TrueSkill rates the disadvantaged team's remaining player slightly higher than it would otherwise. However, since the injured player was not there the full time, the results of the match do not affect the injured player's rating as strongly as if he were there the entire time. This seems to hold even for 1-vs-1 matches. When the partial play time is over 50% of the match, a win counts for a little bit extra, but when partial play time is under 50%, a win counts for a little bit less than it would otherwise. In the context of our skill analysis, when an algorithm used less than 50% of the time limit, it had a negative effect on the rating. It is thus clear that algorithm computation time differences

are not analogous to partial play times, so we had to look at a different solution.

To avoid these problems, instead we computed results for a range of several time limits up to 30 seconds. We excluded results for suboptimality bound 1. In the case of suboptimality bound 1, it never makes sense to use an algorithm with more overhead than A*, so these results are irrelevant to skill rating. We only used results for branching factor 256 because this posed a harder challenge to the planners and therefore allowed better rating discrimination between them. To compare algorithms with scenarios at different time limits in a rating system, we could consider each scenario as being instantiated once in the ratings for each time limit tested (e.g. the 1 Key 1 Door Close scenario would be rated separately for time limit 10 as for time limit 15). Instead, we decided to treat time limits and scenarios as teammates in 2-vs-1 matches against NPC behavior generation algorithms. This discriminates better between the algorithms according to their computation times and yields in the end a single difficulty rating for each scenario, a difficulty rating for each time limit, and a skill rating for each algorithm. Because the ratings are done one trial at a time, the order of the ratings has some effect on the rating outputs. We repeated the full set of ratings 10 times and randomized the order of ratings each time to ensure that the ratings have all converged.

Skill rating results can be seen in Table 7.2. Each TrueSkill rating represents a normal distribution with mean μ and standard deviation σ. For each entry in the table, the TrueSkill μ value indicates the learned mean skill level. The σ value represents the standard deviation of the skill level for that entity. Default parameters of $\mu = 50$ and $\sigma = 50 \div 3$ were used to initialize all ratings before processing. A conservative estimate of the true skill of each entity is provided in the last column of Table 7.2. This column holds TrueSkill's conservative skill rating estimate, computed as $Rating = \mu - 3\sigma$. Thus, with over 99% certainty (the area of a normal distribution above $\mu - 3\sigma$), the entity's true skill is at or above TrueSkill's conservative skill rating. Our MHA* With T-Graph Heuristics method

		TrueSkill		
		μ	σ	Skill
Algorithms	Weighted A*	68.2	0.984	65.2
	T-Graph	66.9	1.0	63.9
	MHA* T-Graph	80.2	1.12	76.9
Scenarios	1 Key 1 Door Exact	22.9	1.04	19.8
	1 Key 1 Door Close	24.7	1.07	21.5
	1 Key 1 Door Far	31.1	1.02	28.0
	5 Keys 1 Door	21.9	1.07	18.7
	5 Keys 5 Doors In Series	34.1	0.985	31.2
Time Limits	3	50.3	0.955	47.4
	6	48.4	0.941	45.6
	9	47.7	0.941	44.9
	12	44.7	0.934	41.9
	15	36.1	1.01	33.1
	18	35.6	1.02	32.5
	21	33.3	1.07	30.1
	24	31.6	1.11	28.3
	27	27.0	1.27	23.2
	30	11.2	3.2	1.64

Table 7.2: Skill rating results for branching factor 256.

tops the skill ratings with Skill=76.9, followed Weighted A* with Skill=65.2, and then our T-Graph method with Skill=63.9. Since we have also treated the scenarios as competitors in the rating process, we have learned μ values for each scenario as well. These represent the relative difficulty of finding a solution for each scenario. TrueSkill regards the 5 Keys 5 Doors In Series scenario as the most difficult, at Rating=31.2. This parallels the computational performance results, where there were many failures to complete a plan within the 30 second time limit for this scenario. The 5 Keys 1 Door and 1Key 1 Door Exact scenarios are rated as the easiest. These results also parallel the computational performance results, where even for suboptimality bound 10, all algorithms managed to find solutions well within the 30 second time limit.

The TrueSkill rating system permits hypothetical matches to be evaluated for quality even if the competing players or teams have never faced one another before. TrueSkill's design operates on an assumption that the skill level of a team is equal to the sum of the skill levels of the team's members [61]. This ignores the possibility that the performance of a team could be greater than the sum of its parts. Hypothetical pairings of algorithm and scenario/time limit teams can be evaluated for an estimate of whether the algorithm could produce a solution to the scenario under the imposed time limit, even if that combination had never been tried before. If we look at the ratings of algorithms and of scenarios, we see that all algorithms rate higher than all of the scenarios. This may reflect the fact that all algorithms here are "complete", and in the absence of a time limit will eventually find a solution if a solution exists. However, since we performed all TrueSkill rating operations as 1-vs-2 team matches, to properly compare an algorithm's skill against a scenario, the scenario should be paired with a time limit.

For example, Weighted A*'s TrueSkill rating of 65.2 tested against scenario 5 Keys 1 Door (rating 18.7) and time limit 6s (rating 45.6) on a team together (combined rating 64.3) is close match predicting a tie. In our experiments of this configuration, Weighted A*

		EWR
Algorithms	Weighted A*	55.5%
	T-Graph	58.0%
	MHA* T-Graph	86.0%
Scenarios	1 Key 1 Door Exact	18.3%
	1 Key 1 Door Close	25.0%
	1 Key 1 Door Far	40.0%
	5 Keys 1 Door	18.3%
	5 Keys 5 Doors In Series	65.8%
Time Limits	3	70.0%
	6	63.3%
	9	61.7%
	12	50.0%
	15	23.3%
	18	21.7%
	21	18.3%
	24	15.0%
	27	8.33%
	30	3.33%

Table 7.3: EWR results for branching factor 256.

always succeeded, however computation times average 5.6s, nearly at the 6s time limit.

Another example: MHA* With T-Graph Heuristics (rating 76.9) vs. scenario 1 Key 1 Door Exact (rating 19.8) and time limit 30 (rating 1.64) on a team together (combined rating 21.4). This is a highly unbalanced match strongly in favor of the search algorithm. Indeed, all experiments of this configuration yielded successful solutions for the algorithm.

We compared the TrueSkill rating results with the Empirical Win Rate (EWR) results for the same data from Section 7.2.2. EWR results are available in Table 7.3. The relative ordering of these results matches the relative ordering of the TrueSkill skill values for the same scenarios and time limits. However, the relative ordering of algorithms does *not* match. The MHA* with T-Graph heuristics method had the highest EWR and the highest skill rating, but Weighted A* had a higher skill rating even though T-Graph had a higher EWR. This means that the times when Weighted A* succeeded and T-Graph failed, it was

in trials with more difficult scenarios and time limits than the times that T-Graph succeeded and Weighted A* failed. The TrueSkill ratings suggest that the 1 Key 1 Door Far scenario (rating 28.0) and the 24 second time limit (rating 28.3) both contribute approximately the same amount of challenge to algorithms attempting to produce an NPC behavior plan successfully. The same kind of information cannot be gleaned from EWR results, where, e.g., the 1 Key 1 Door Far scenario has EWR 40.0% and time limit 24 has EWR 15.0%.

We generated our TrueSkill ratings by considering only wins and losses, however it may yet be possible to incorporate the "margin of victory" (in our case, the margin of computation time) into the rating system. In our context, the computation time of an algorithm's solution could be treated as a kind of score so that even when two algorithms both succeed, the faster one would rate higher. It might also be possible to score success according to how closely a solution re-uses demonstration data (with e.g. dynamic time warping [62]), for ratings based on trainability. ELO has been adapted to account for margin of victory [63, 64]. TrueSkill has also been extended to accommodate "score-based match outcomes"[65]. We leave it as future work to use one of these rating systems which take into account the margin of victory to analyze results like ours.

Another avenue for future work in this area is the use of skill rating systems to compare the skill of NPCs with the skill of human players. In this way, players might be matched with opponents of similar skill. Also, weak players might be matched with strong NPCs to help them solve difficult challenges or defeat powerful opponents. Using the team-based ratings enabled by TrueSkill, NPC skill and player skill can be analyzed in the same framework with quest difficulty, time limit difficulty, and other game elements which affect how challenging the game can be. Skill ratings cannot be used to determine what makes a skilled player better than an unskilled one. However, given a process to learn the methods of a skilled player, skill ratings could inform that process better than empirical winning rates alone.

Chapter 8

Concluding Remarks

8.1 Discussion and Future Work

A* search and other heuristic graph search techniques are most widely used in video games for navigation path planning. Our heuristic formulations for planning NPC behavior from demonstration can trivially be applied to path planning. This is accomplished by only including navigation state in the search space. Without modifying a path planner's cost function, the T-Graph heuristic can be used to make certain routes through a game environment more likely to be used by NPCs. This might be used to help keep NPCs from planning "as the crow flies" shortest-distance paths when there are more sensible human-like routes to take. MHA* with inadmissible heuristics like ours based on the T-Graph would be easy to apply to video game path planning most likely improving computation times and permitting training of path outputs.

As noted in Section 4.4, our T-Graph heuristic is not ϵ-consistent, so care must be taken if the T-Graph heuristic is used in a domain which requires guarantees on the suboptimality of solutions. Weighted A* search with the T-Graph heuristic does not provide a guarantee that solutions will be at most ϵ times the cost of the optimal solution. However, other

algorithms such as MHA* can utilize some arbitrarily inconsistent heuristics while still providing guarantees on the suboptimality of solution costs.

In Section 7.2.1 we found that a 1:1 ratio of the parameters w and ϵ^T worked well in our trials, however this might need to be assigned differently for different types of problems. For w_1 and w_2 in MHA*: though we found that all weight should be put on w_1, for uncalibrated heuristics, w_2 should be raised. Alternatively, we suggest exploring the use of a different algorithm like Improved MHA*[44], which does not have a second parameter to tune.

One can imagine many situations in video games where training data collected in the past is not relevant to the current problem at hand. For example, see our experimental results in Section 7.2.2 for the 1 Key 1 Door Far scenario, which was specifically crafted to lead T-Graph search astray, but represents a situation that could easily be encountered in practice when items like the key in this example move far from their previous locations. Computation times using Weighted A* and the T-Graph heuristic can be very bad in this kind of situation. It would be important to develop methods to filter irrelevant experience. Multi-heuristic approaches can avoid some of these problems, but it is still true that irrelevant experience will degrade search performance. Machine learning techniques might be used to help classify experience traces as relevant or irrelevant.

The authors of [8] describe a method using nondeterministic NPC behavior planning to generate diverse solution policies which can be executed as FSMs controlling NPC behavior. We imagine that an application of nondeterministic heuristic search ideas and the use of our T-Graph heuristic could bias this kind of FSM-generation process to produce FSMs encoding trained NPC behaviors. The use of FSMs is desirable in video game NPC control because they are computationally inexpensive at runtime. Since behavior trees can be thought of as hierarchical finite state machines, we imagine the same approach could be applied to the automated generation of trained behavior trees using heuristic search.

Since our approach requires a discretization of the game state space, our method may be applied well to games which are inherently discretized in a reasonable way, such as board games and tile-based games. This includes some three-dimensional games, such as Minecraft. In our experiments, we sometimes observed strange behavior because our discretization of the game world did not correspond closely enough with the spatial and temporal continuity of Skyrim. In some instances we had to modify our world model to be more conservative. For example, we had to make the killer bear enemy dole out additional damage in simulation and use an inflated detection radius. Otherwise, the planner could generate edge-case solutions stepping near to the bear and then running away. These edge-case solutions worked in simulation but did not work in real-time in the continuous game world. This same kind of problem is mentioned in [66], where attack ranges had to be inflated and a special movement delay had to be added to account for shortcomings in the model compared to the full game. An interesting avenue for future work would be implementing the T-Graph or MHA* With T-Graph Heuristics approaches in the framework of Goal-Oriented Action Planning (GOAP)[11]. Planning trained tactical NPC behavior at that level of abstraction could yield good NPC behavior solutions for video games in much faster computation times. It may be most fruitful to use a system at a higher level of abstraction, such as GOAP, because low-level discrepancies are expected and accounted for with flexibility in the control schemes used to execute a high-level plan.

In the same vein, another approach which we think could pair well with our work is that of State Lattice Planning with Controller-based Motion Primitives [67, 68], where controller-based actions make up some of the edges on the search graph. We believe that video games are full of many scenarios which can effectively be handled by a controller (for example, melee combat with an enemy, or the navigation of obstacles such as moving platforms requiring precision and dexterity). The “controllers” used could be defined using other NPC behavior techniques such as FSMs or behavior trees. Planning detailed behavior

for some of these things (without controllers) could require adding dimensions to the search space and refining the discretization of the search space. However, the use of controller-based motion primitives able to accomplish the same things could permit a simpler (and faster) search process.

Our two NPC behavior training methods suffer from the limitation that their guidance is dependent on the spatial configuration of demonstrations. When the spatial aspects of a demonstration do not correspond well with the spatial layout of the problem at hand, it may happen that the planner completely ignores the demonstration data, even though it might have been a demonstration of a useful high-level tactic. We leave it as future work to devise ways to utilize demonstration in the planning process without dependence on the spatial similarity of demonstration to the problem at hand. The abstract tactic encoded in a demonstration could be extracted, generalized, and then applied to new problems with differing spatial arrangements. The *events* and *activities* used for playstyle classification in Chapter 6 are a possible first step in this direction.

Because of the geometric formulation use in step 4 of the T-Graph procedure described in Section 4.3, triangle inequality $h(s,a) + h(a,s') \geq h(s,s')$ must be satisfied. This is equivalent to the condition defining a search property called h-consistency. When the simple graph heuristic used within the T-Graph computation is based on Euclidean distance in the search space (as it is in our work), h-consistency is trivially satisfied, however there can be other search spaces where it is not. h-consistency is an assumption of our T-Graph heuristic formulation.

We think that rating systems may be a useful tool in the development of NPC control schemes for games. Skill ratings offer some insights into the nature of NPC behavior planning challenges that empirical win rates cannot. We hope to see future work use rating systems to determine which NPC behavior generation schemes could pair best with humans in cooperative challenges. Human player skill rating information could inform attempts to

analyze which behavior patterns are most associated with skilled players and then use these skilled behaviors to train NPC behavior via our methods.

Finally, we review what it would take to implement our methods in a new video game. We found in our user study that players were eager to demonstrate behavior to NPCs, even without being told that there was any NPC control mechanism utilizing demonstration. A video game implementation requires a gameplay feature allowing players to demonstrate behaviors to the NPC. This could take the form of a play diagram in an American football playbook, wherein the player draws out a path for the NPC to take. Alternatively, the player could act out a demonstration in-game for the NPC to use later. We took the latter approach in our example implementation for Skyrim (see Section 4.5) and the former approach in our user study (see Section 7.1.1). As discussed above, there should be a mechanism to filter out irrelevant demonstrations since they negatively affect search performance. If our MHA* with Quest Event Heuristics approach is used, performance is best if the quest events correspond with places in the environment where visitation is required to complete the task at hand. Care should be taken while choosing these quest events; it is not a good idea to arbitrarily create a quest event for every potentially-relevant place in the environment. We reviewed some of these considerations in Section 5.1. Our planner output is a sequence of actions the NPC can take to accomplish its goal. Depending on the mechanics of the game and the discretization of the search space, path-following code may be needed to translate the planned behavior into high-resolution in-game control for the NPC. Because of nondeterminism in the behavior of the player (and potentially, other game elements), it would be important to monitor the state of the game world during NPC behavior execution. If the game state strays far from the game state as modeled during the NPC behavior planning process, a new behavior plan should be generated. The automated FSM-generation technique discussed above offers one solution to this problem, since it incorporates nondeterminism in the planning process and in the structure of the generated

FSMs. Finally, cooperative companion NPC behavior planning is best done with good models of player behavior, as we discussed in Chapter 6.

8.2 Conclusions

In this thesis, we present our work on training video game NPC behaviors. We accomplish NPC behavior training by using our Training-Graph heuristic to bias a search-based planning process. We adapt training data to novel game quests by using Multi-Heuristic A* with heuristics based on the Training-Graph heuristic.

We presented a description and analysis of the Training-Graph heuristic, which allows training data to be used to guide an NPC behavior planner graph search. The T-Graph approach solves problems encountered when trying to use the E-Graph heuristic on video game training data.

We described and analyzed an approach using Multi-Heuristic A* search with the T-Graph heuristic to better adapt the planning process to new quest tasks. Because MHA* shares partial solutions while being guided by multiple different heuristics, our method also enables demonstrations of particular tactics to be combined to solve a compound problem quickly.

We described work done in collaboration with another researcher on playstyle classification, player modeling, and cooperative planning. This work was built as an extension to our framework and tested our methods in the context of cooperative planning.

We tested and analyzed the qualitative performance of both of our algorithms. We implemented both methods as extensions to the popular video game Skyrim and evaluated their potential impact on human gameplay with a small user study. Both of our methods tend to re-use demonstration data to find a trained behavior solution to quest tasks. We also tested and analyzed the computational performance of our methods and reviewed their

theoretical properties. We tested a means to rate and compare the *skill* of algorithms and the difficulty of planning challenges.

Bibliography

[1] Ian Millington and John David Funge, *Artificial Intelligence for Games 2nd Edition*, Morgan Kaufmann, 2009.

[2] P.E. Hart, N.J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, July 1968.

[3] Mike Phillips, Benjamin Cohen, Sachin Chitta, and Maxim Likhachev, "E-graphs: Bootstrapping planning with experience graphs," in *Proceedings of Robotics: Science and Systems*, Sydney, Australia, July 2012.

[4] Sandip Aine, Siddharth Swaminathan, Venkatraman Narayanan, Victor Hwang, and Maxim Likhachev, "Multi-heuristic A*," *International Journal of Robotics Research (IJRR)*, 2015.

[5] Mike Phillips, Victor Hwang, Sachin Chitta, and Maxim Likhachev, "Learning to plan for constrained manipulation from demonstrations," in *Proceedings of Robotics: Science and Systems*, Berlin, Germany, June 2013.

[6] Thomas Minka, Thore K H Graepel, and Ralf Herbrich, "Determining relative skills of players," U.S. Patent US8538910 B2, 2013.

[7] Stephen Chen, "Learning player behavior models to enable cooperative planning for non-player characters," M.S. thesis, Carnegie Mellon University, 2017.

[8] Alexandra Coman and Héctor Muñoz-Avila, "Automated generation of diverse npc-controlling fsms using nondeterministic planning techniques," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[9] Damian Isla, "Handling complexity in the halo 2 ai," in *GDC 2005 Proceedings*, 2005.

[10] Owen Macindoe, Leslie Pack Kaelbling, and Tom´as Lozano-P´erez, "Pomcop: Belief space planning for sidekicks in cooperative games," in *Proceedings, The Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[11] Jeff Orkin, "Symbolic representation of game world state: Toward real-time planning in games," in *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2004.

[12] Donald Kehoe, "Designing artificial intelligence for games," Online, jul 2009, `https://software.intel.com/en-us/articles/designing-artificial-intelligence-for-games-part-1`.

[13] James Wexler, "Artificial intelligence in games: A look at the smarts behind lionhead studio's "black and white" and where it can and will go in the future," Online, may 2002, `https://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf`.

[14] Jack Belzer, Albert George Holzman, and Allen Kent, *Encyclopedia of Computer Science and Technology*, vol. 25, p. 73, CRC Press, 1975.

[15] Alexander Shoulson, Francisco Garcia, Matthew Jones, Robert Mead, and Norman Badler, "Parameterizing behavior trees," in *Motion in Games*, Jan Allbeck and Petros Faloutsos, Eds., vol. 7060 of *Lecture Notes in Computer Science*, pp. 144–155. Springer Berlin / Heidelberg, 2011.

[16] Damian Isla, "Building a better battle: Halo 3 ai objectives," in *Game Developers Conference*, 2008.

[17] Marc-Antoine Argenton, Max Dyckhoff, Chris Hecker, and Lauren McHugh, "Three approaches to halo-style behavior tree ai," in *Game Developers Conference*, 2007.

[18] Greg Snook, "Simplified 3d movement and pathfinding using navigation meshes," in *Game Programming Gems*, Mark DeLoura, Ed., pp. 288–304. Charles River Media, 2000.

[19] Jeff Orkin, "Three state and a plan: The a.i. of f.e.a.r.," in *Game Developers Conference*, 2006.

[20] Richard E Fikes and Nils J Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, 1971.

[21] Edmund Long, "Enhanced npc behaviour using goal oriented action planning," M.S. thesis, University of Abertay Dundee, 2007.

[22] Christopher Geib, Janith Weerasinghe, Sergey Matskevich, Pavan Kantharaju, Bart Craenen, and Ronald P. A. Petrick, "Building helpful virtual agents using plan recognition and planning," in *Proceedings of the Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 2016.

[23] Nathan R. Sturtevant, "Incorporating human relationships into path planning," in *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013.

[24] Alberto Uriarte and Santiago Ontanon, "Improving monte carlo tree search policies in starcraft via probabilistic models learned from replay data," in *Proceedings, The Twelfth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-16)*, 01 2016.

[25] E.W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[26] Ira Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1, pp. 193–204, 12 1970.

[27] M. Likhachev, G. Gordon, and S. Thrun, "ARA*: Anytime A* search with provable bounds on sub-optimality," in *Proceedings of Conference on Neural Information Processing Systems (NIPS)*, S. Thrun, L. Saul, and B. Schölkopf, Eds. 2003, MIT Press.

[28] Anthony Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *International Journal of Robotics and Automation*, vol. 10, pp. 89–100, 1993.

[29] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *Robotics, IEEE Transactions on*, vol. 21, no. 3, pp. 354–363, June 2005.

[30] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime dynamic a*: An anytime replanning algorithm," in *International Conference on Automated Planning and Scheduling*. 200, AAAI.

[31] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Tsai-Yen Li, Rajeev Motwani, and Prabhakar Raghavan, "A random sampling scheme for path planning," *INTERNATIONAL JOURNAL OF ROBOTICS RESEARCH*, vol. 16, pp. 759–774, 1996.

[32] Michael Phillips, *Experience Graphs: Leveraging Experience in Planning*, Ph.D. thesis, Carnegie Mellon University, 2015.

[33] Steven M. Lavalle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep. TR 98-11, Computer Science Department, Iowa State University, 1998.

[34] James J. Kuffner Jr. and Steven M. Lavalle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*, 2000, pp. 995–1001.

[35] L.E. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 4, pp. 566–580, Aug 1996.

[36] V. Boor, M. Overmars, and A van der Stappen, "The gaussian sampling strategy for probabilistic roadmap planners," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '99)*, 1999, vol. 2, pp. 1018–1023.

[37] R. Bohlin and E. Kavraki, "Path planning using lazy prm," in *IEEE International Conference on Robotics and Automation, Proceedings (ICRA '00)*, 2000, vol. 1, pp. 521–528.

[38] Jyh-Ming Lien and Yanyan Lu, "Planning motion in similar environments," in *Proceedings of Robotics: Science and Systems V*, Seattle, USA, June 2009.

[39] M. Zucker, N. Ratliff, A. Dragan, M. Pivtoraiko, M. Klingensmith, C. Dellin, J. A. Bagnell, and S. Srinivasa, "Chomp: Covariant hamiltonian optimization for motion planning," *International Journal of Robotics Research*, 2013.

[40] Chelsea Finn, Sergey Levine, and Pieter Abbeel, "Guided cost learning: Deep inverse optimal control via policy optimization," in *Proceedings of the 33rd International Conference on Machine Learning*, 2016, vol. 48.

[41] Brenna Argall, Sonia Chernova, Manuela Veloso , and Brett Browning , "A survey of robot learning from demonstration," *Robotics and Autonomous Systems*, vol. 67, pp. 469–483, 2009.

[42] Dmitry Berenson, Pieter Abbeel, and Ken Goldberg, "A robot path planning framework that learns from experience.," in *ICRA*. 2012, pp. 3671–3678, IEEE.

[43] Judea Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[44] Venkatraman Narayanan, Sandip Aine, and Maxim Likhachev, "Improved multi-heuristic a* for searching with uncalibrated heuristics," in *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, 2015.

[45] John Drake, Alla Safonova, and Maxim Likhachev, "Demonstration-based training of non-player character tactical behaviors," in *Proceedings of the Twelfth Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2016.

[46] Ruslan "Strideman" Rybka, "Thinking with time machine," Online, `http://www.moddb.com/mods/portal-2-thinking-with-time-machine`.

[47] Ian Millington and John Funge, *Artificial Intelligence for Games, Second Edition*, Morgan Kaufmann Publishers, Inc., 2008, ISBN: 0123747317, 9780123747310.

[48] Massimiliano Pavan and Marcelo Pelillo, "A new graph-theoretic approach to clustering and segmentation," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[49] Michael Phillips, Venkatraman Narayanan, Sandip Aine, and Maxim Likhachev, "Efficient search with an ensemble of heuristics," in *International Joint Conference on Artificial Intelligence (IJCAI)*, Pittsburgh, PA, July 2015.

[50] Rensis Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, 1932.

[51] John Drake, Alla Safonova, and Maxim Likhachev, "Towards adaptability of demonstration-based training of npc behavior," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2017.

[52] "Unity 2017: The world-leading creation engine for gaming," Online, 2017o, `https://unity3d.com/unity`.

[53] Arpad E. Elo, *The Rating of Chessplayers, Past and Present*, Arco Pub., 1978.

[54] Mark E. Glickman, "Parameter estimation in large dynamic paired comparison experiments," *Applied Statistics*, vol. 48, pp. 377–394, 1999, Part 3.

[55] Mark E. Glickman, "Dynamic paired comparison models with stochastic variances," *Journal of Applied Statistics*, vol. 28, no. 6, pp. 673–689, 2001.

[56] Anurag Sarkar and Seth Cooper, "Level difficulty and player skill prediction in human computation games," in *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-17)*, 2017.

[57] Michal Certicky and David Churchill, "The current state of starcraft ai competitions and bots," in *AIIDE 2017 Workshop on Artificial Intelligence for Strategy Games*, 10 2017.

[58] "Student starcraft ai tournament & ladder," Online, https://sscaitournament.com/index.php?action=scoresCompetitive.

[59] "Starcraft rating system," Online, https://iccup.com/starcraft/sc_rating_system.html.

[60] Severin Hacker and Luis von Ahn, "Matchin: Eliciting user preferences with an online game," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 10 2009, pp. 1207–1216.

[61] Jeff Moser, "Computing your skill," Online, mar 2010, http://www.moserware.com/2010/03/computing-your-skill.html.

[62] HIROAKI SAKOE and SEIBI CHIBA, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 26, pp. 43–49, 1978.

[63] Nate Silver, "Introducing nfl elo ratings," Online, sep 2014, https://fivethirtyeight.com/features/introducing-nfl-elo-ratings/.

[64] Nate Silver, "It's brazil's world cup to lose," Online, jun 2014, https://fivethirtyeight.com/features/its-brazils-world-cup-to-lose/.

[65] Shengbo Guo, Scott Sanner, Thore Graepel, and Wray Buntine, "Score-based bayesian skill learning," in *Proceedings of the 2012 European Conference on Ma-*

chine Learning and Knowledge Discovery in Databases - Volume Part I, Berlin, Heidelberg, 2012, ECML PKDD'12, pp. 106–121, Springer-Verlag.

[66] David Churchill, Zeming Lin, and Gabriel Synnaeve, "An analysis of model-based heuristic search techniques for starcraft combat scenarios," in *AAAI Publications, Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2017.

[67] Jonathan Butzke, Krishna Sapkota, Kush Prasad, Brian MacAllister, and Maxim Likhachev, "State lattice with controllers: Augmenting lattice-based path planning with controller-based motion primitives," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014.

[68] Jonathan Butzke, *Planning for a Small Team of Heterogeneous Robots: from Collaborative Exploration to Collaborative Localization*, Ph.D. thesis, Robotics Institute, Carnegie Mellon University, 2017, to be published.