Towards Smart Assistants in Two-Party Collaboration

NGUYEN DINH TRUONG HUY

Bachelor of Computer Science, Honors

National University of Singapore

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2012

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Nguyen Dinh Truong Huy

20 December 2012

Acknowledgements

This dissertation would not have been completed without the support and encouragement of many people. I would like to reserve this section to express my gratitude to all of them.

Firstly, I would like to thank my thesis advisors, Prof Leong Tze-Yun and Prof Lee Wee-Sun, for having spent many years patiently showing me how to do research. Every meeting with them enlightens me in various ways. Prof Leong taught me how to identify meaningful problems, formulate them and brainstorm about possible approaches, while Prof Lee often showed me how to work out the theoretical analysis of many algorithms. Personally, I am deeply indebted to Prof Leong for trying her best in finding financial sources to fund my work, allowing me to focus on drafting new research, instead of survival, ideas. This thesis is not possible without you two.

I owe a great deal to Dr. Tomi Silander for sparing his time in providing his precious critique on my paper and dissertation drafts. All meetings with him are not without a mixed feeling between anxiety and excitement. On the one hand, I know he would not hesitate in telling me that my prose is in bad shape, but on the other hand, it is such a joy to learn a great deal about presenting and crafting research stories with this guy. Awesome work, Tomi!

I have to thank Chu Duc Hiep, Dinh Thien Anh and Truong Duc Thang (the names are obviously in alphabetical order to avoid any personal bickering/nagging) for spending this ordeal with me. Our random discussions were

so random, but you have no idea how much they lighted up my days. Now that I think about it, I believe the fact that you guys suck in Pro Evolution Soccer (does not matter which version really, PES 6, 2008, 2009 or 2010, the list goes on...) accounts a lot for those fond memories. And in FIFA, too.

I owe Le Thi Diem Thu, my *Heo Map*, much more than I have ever told her. Her faith in me is a constant source of energy that fuels my determination in this long and windy journey. Now that I am ready to turn a new page of my life, I am glad that we can finally walk the road together. It has been an eternity enduring this hardship without you beside me.

Last but not least, I want to thank my family for having made my journey especially memorable. Dad, Mom, there have been moments when I wanted to quit everything altogether just because of you, but I went on anyway because I know this accomplishment is something that both of you will be so proud of. Hai, my long journey left you alone in the mess and trust me, I know, through first hand experiences, how desperate that could be. Now that you have your little family to care for, I wish you all the best things in this world. I will always have your back.

Publications

- **Truong-Huy D. Nguyen, Tomi Silander, Wee-Sun Lee, and Tze-Yun Leong.** Bootstrapping Simulation-based Algorithms with a Suboptimal Policy. *Submitted and under review*.

- **Truong-Huy D. Nguyen, Wee-Sun Lee, and Tze-Yun Leong.** Bootstrapping Monte Carlo Tree Search with an Imperfect Heuristic. In *Proceedings of the Twenty-Third European Conference on Machine Learning (ECML/PKDD 2012)*, Bristol, UK, September 2012.

- **Truong-Huy D. Nguyen, David Hsu, Wee-Sun Lee, Tze-Yun Leong, Leslie P. Kaelbling, Tomas Lozano-Perez, and Andrew Grant.** CAPIR: Collaborative Action Planning with Intention Recognition. In *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2011)*. AAAI, AAAI Press, 2011.

- **Truong-Huy D. Nguyen and Tze-Yun Leong.** A Surprise-Triggered Adaptive and Reactive (STAR) Framework for Online Adaptation in Non-stationary Environments. In Christian J. Darken and G. Michael Youngblood, editors, *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2009)*, pages 82-87. AAAI Press, 2009.

Table of Contents

Abstract

Towards Smart Assistants in Two-Party Collaboration

Nguyen Dinh Truong Huy

In this work, we build a framework for smart assistants in two-party collaboration, which depicts the scenario where the AI planner takes the role of a subordinate helping the *lead* agent achieve a set of common objectives, or *subgoals*. The main difficulties in acting smart in such a setting are (1) to understand the hidden intention of the lead agent, and (2) to select the most appropriate actions. As shown analytically in the first part of the thesis, these two challenges are both **PSPACE**-complete.

To address these challenges, we model each subgoal as a Markov Decision Process (MDP). The solutions of these MDPs dictate how the team should behave to achieve each subgoal. By assuming that the lead agent is optimal at the subgoal level, we can estimate his expected behavior as part of the optimal subgoal solution. This allows an efficient way to track the lead's intention using Bayesian inference, based on observations of his action history. We then use utility-theoretic maximization to select actions for the assistant. The approach is shown to yield near-human level assistance while incurring a running time that scales quadratically to the number of subgoals.

In real-life domains, we need to solve large MDPs when the subgoal models yield prohibitively large state spaces. In such cases, the aim is to obtain high-quality estimations of the subgoal solutions quickly. To achieve this goal, we propose to use simulation-based algorithms bootstrapped by offline-learnt heuristic policies. A simple yet novel technique to enhance the performance of state-of-the-art algorithms such as Sparse Sampling (SS), Forward Search Sparse Sampling (FSSS) and Upper Confidence Bound applied in Trees (UCT) is proposed and evaluated. We demonstrate that the technique outperforms some commonly used enhancements when coupled with suitable heuristics. The results and insights on the approximate solutions for large state-space MDPs are relevant to the planning community in general.

Equipped with the online approximate algorithms, our framework for intelligent collaborative assistance can operate in complex situations, which often yield high-dimensional models. Although the domain of focus in this work is the genre of collaborative games, our proposed problem formulation and solution framework are intended to be general and may still be applicable in other domains such as robot assistance.

List of Tables

List of Figures

Chapter 1

Introduction

A desire to make our environment actively attend to our needs lies in the heart of pervasive or *ubiquitous computing* (ubicomp) [63], an emerging interdisciplinary research domain. Ubicom can be informally described as comprising of "machines that fit the human environment instead of forcing humans to enter theirs" [78]. An example of a ubicomp environment is a *smart space* [66, 41], which can be "an enclosed area such as a meeting room or corridor". This enclosed region has computing capabilities embedded in the the building infrastructure and is able to automatically customize the space's conditions to satisfy the occupant's need. For instance, if the occupant enters a smart room tired, the cooling and lighting of the room should be adjusted accordingly to comfort him. The smartness could extend to other objects in the room as well. While ubiquitous computing is a wide concept, encompassing components in many fields such as systems design and engineering, systems modeling, and user interface design, it can benefit greatly from advancements in artificial intelligence. In fact, the existence of an autonomous smart assistant, be it virtual or physical, can benefit almost any current system. A doctor in the surgery room can use some help from a robot assistant that execute tasks quickly and with high precision. A patient at night could rely on a robot nurse to attend to his urgent need,

such as getting some water in the middle of the night or at least alarming a centralized team of human nurses when unexpected incidents occur. An explorer with a hand-held device that tells him relevant and vital information about the surrounding, based on his current condition and intention, could anticipate and plan his trip better, thereby avoiding unanticipated dangers.

In this work, our aim is to create a smart assistant and assess it in the modern game domain. One main reason for this choice of benchmark domain is the controlled experiment environments offered by games. Video games provide a much safer medium for hypothesis testing and validation than other domains such as the health care environments. Moreover, simulated game worlds are perfect ecosystems that can be altered and examined quickly, as compared to real-life setting with physical entities and agents. Consequently, a thorough verification of our proposed approach can be done through the means of simple computer simulations instead of reconstructing a physical scenario over and over again. Moreover, any advancement in game AI can instantly improve the quality of modern video games, which constitute one of the fastest growing and most profitable entertainment industries in the world at the moment. This directly translates to significant financial gain. That said, our proposed problem formulation and solution framework are intended to be general and may still be applicable in other domains such as robot assistance.

1.1 The Video Game Domain

The game industry is one of the fastest growing entertainment industries in the world; in 2011, it is allegedly worth more than $100 billion dollars [35]. In particular, in January 2011, “Call of Duty: Black Ops”, the latest installment of US game publisher Activision’s most successful franchise, set a five-day record sale of $650 million dollars for any movie, book or video game ever released [16].

Creating games that deliver premium entertainment experience has never been more profitable.

Rapid improvements in computer graphics have raised the expectation bar of gamers by presenting them virtual environments filled with aesthetically genuine-looking objects and entities. In recently released games, forests look staggeringly real with waterfalls and beautiful reflections; from a close distance, all virtual characters look like they are wearing the complexion of their real-life counterparts. Inevitably, with such entities populating the game world, the gamers cannot expect any less than comparably natural behavior of the *non-player characters* (NPCs). However, this is where many AI authoring techniques currently employed in modern video games fall short. In particular, many recent games are still released with ally or opponent NPCs exhibiting buggy AI behavior [20]. For instance, in the final boss encounter of the game *Assassin's Creed: Brotherhood*[1], released in 2010, the boss sometimes appears lying dead on the floor, offering no chance for the protagonist to attack him. As a result, the guards can continuously attack the protagonist infinitely[2]. AI systems that are able to produce bug-free and human-like NPCs are therefore highly demanded.

Traditionally the behavior of NPCs in games is hand-crafted by programmers using techniques such as Scripting [23], Hierarchical Finite State Machines (HFSMs) [29] and Behavior Trees [15]. These techniques are simple and straight-to-the-point, i.e., if designed correctly, the NPCs will behave exactly as intended by the programmers. As a result, they are the most widely used AI techniques at the moment. However, in complex environments of modern games, due to the large number of game states, programmers adopting these techniques have to solve the non-trivial tradeoff between *comprehensiveness* and *production cost*. The *comprehensiveness* of an AI system is determined by the ratio of situations in which the NPC behaves as intended by the programmers

[1] http://assassinscreed.uk.ubi.com/brotherhood/

[2] http://assassinscreed.wikia.com/wiki/Bugs/Assassin%27s_Creed:_Brotherhood

over all possible scenarios. If an NPC exhibits buggy behavior in too many scenarios, the AI system is not comprehensive enough. However, it is costly in terms of human labor to ensure adequate comprehensiveness; as the number of different scenarios in a game increases, so does the amount of human hours spent on both designing and testing phases. As a result, games that are released prematurely to meet deadlines usually feature AI characters that exhibit poor behavior. To alleviate some of the difficulties of having to anticipate all possible scenarios, planning-based techniques such as Hierarchical Task Networks (HTNs) [43] and Goal-Oriented Action Planner (GOAP) [55] specify goals for the NPCs and use planning algorithms to search for appropriate actions while in-game, or *online*.

1.2 Problem Description

In classical multi-agent setting the planner is tasked to compute collaborative actions for a subset of agents in order to optimize some common reward criterion. In this work, we will focus on two-party collaboration, in which the planner is to control only one member, i.e., the *assistant*, in a team of two agents. The other member, i.e., the *lead agent*, is assumed to play the role of a mastermind and act according to a hidden agenda that is not influenced by the planner's course of actions. It, however, may be impossible for the lead agent to complete the goals without the assistant's aid; therefore, coordination is almost obligatory to satisfactorily achieve the objectives. Besides the two agents, the environment is populated with other entities, the interaction with which gives our agents rewards or inflicts penalties.

In this formulation, the *lead agent* decides the action plan for the team, and the assistant needs to identify this hidden plan, which is not explicitly conveyed, and selects actions that are most appropriate to achieve the common objectives.

For generality, we assume that no lead action directly communicates the intention to the assistant. In many cases, this requirement is desired due to geographical/communicative limitations or in order to conceal the team plan from opponent agents. The main difficulties in assisting the lead agent in such a setting are (1) to understand his intention, and (2) to select actions that are most helpful to the lead agent in achieving his hidden goal.

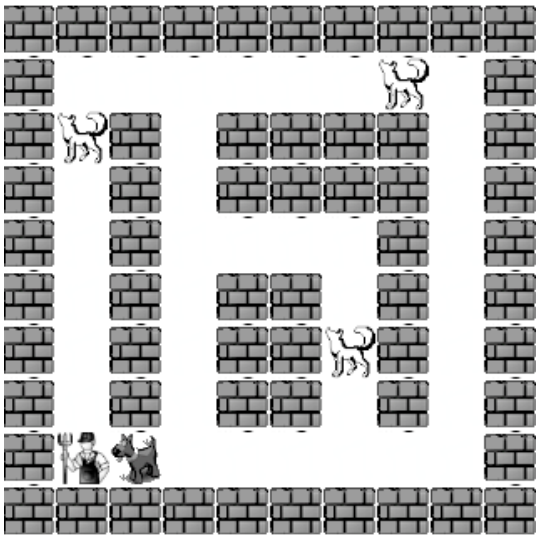


Figure 1-1: *Collaborative Hunters*, a sample game that requires the protagonists, Farmer and Dog, to surround the wolves and kill them. The Farmer is the human-controlled lead agent, while Dog is the *assistant NPC* and wolves are antagonist NPCs.

For instance, let us consider a game called *Collaborative Hunters*. In this game, the assistant (illustrated as a dog) has to help the lead agent (the sheep farmer) kill several wolves in a maze-like environment. A wolf will run away from the farmer or the dog when they are within its vision limit, otherwise it will move randomly. Since wolves can only be inflicted damage by the farmer, the dog's role is strictly to round them up. A sample game map is shown in Figure 1-1. Note that collaboration is often truly required in this game - without rounding up a wolf in order to cut off its escape paths, attacking a wolf can be quite difficult. The challenges for the assistant dog in this case are to identify the wolf the farmer is targeting, i.e., his *hidden intention*, and how to select actions

that are most helpful in stopping that wolf from running away.

1.2.1 Decision Making Elements

In its decision making process, an AI assistant needs to take into account the following elements: the state in which its team operates, the team's capabilities (actions) and their effects, and the terminal states when the team coordination is over. Besides, in a *finite-horizon* setting, the assistant must also take note of the time limit that the coordination is allowed to last. We will describe these decision-making elements in detail below.

State captures essential information about the environment that is pertinent to the decision making process. For instance, in the game described in Figure 1-1, the most important elements are the characters' positions. Besides these fully observable elements, the lead agent's intention is also important; with the same state configuration, depending on the target that the farmer is pursuing, the dog has to select different assistive actions.

Assistant action space includes all the valid actions that the AI assistant could do. For instance, in the aforementioned game, the assistant action space contains movement actions {*no_move, up, down, left, right*}, which indicates that at any point of time, the dog has the choice to move in any of four directions or stay at the same position. It is worth noting that this formulation is not constrained to grid-based games; it is a matter of convenience to use this particular game type for illustration purpose. Therefore, in other game settings, for example Role Playing Games, we could have something like *chase_peasant* or *eat_bone* in the assistant action space.

Lead action space contains all possible actions that the lead agent could execute. For example, in *Collaborative Hunters*, besides movement ac-

tions, the farmer's action space contains an attacking move that allows the farmer to shoot a nearby wolf.

Action effect informs the assistant about two aspects of consequence when a team action is executed in a given state: how the state is transformed and how desirable (rewarding) it is. Note that since NPCs not belonging to the team are completely out of control, their actions, which also influence the outcome of the team's action, are usually assumed to be part of the environment dynamics. For instance, when a wolf is within the attacking range of the farmer, any joint action that includes the attacking move from the farmer inflicts damage to the wolf. If the wolf is dead after that move, the team is rewarded certain points.

Terminal states define the end of the assistance. These states make up a subset of the state space, in which certain variables must have some specific desired values. For instance, in the game above, terminal states include all the states in which all wolves are exterminated.

Subgoals denote the possible *targets* or *intentions* the lead agent may be pursuing at any point of time. In *Collaborative Hunters*, as the objective is to exterminate all wolves in the map, the lead agent can be targeting some wolf at any point of time. Each wolf is therefore a possible subgoal.

Horizon defines the maximum time that coordination experience lasts, which can be infinitely long in *infinite-horizon* setting.

In this setting, the assistant's objective is to sequentially select an action that maximizes the team's expected return when the game episode is over, either by reaching a terminal state or when the time limit elapses in the finite-horizon case.

1.3 Objectives

Motivated by recent theoretical advancements in sequential action planning, we want to devise a unified framework for smart autonomous assistants, with the primary assisted target being a human lead. As such, the framework needs to account for a variety of suboptimal behaviors such as mind switch or bounded rationality [71]. One important performance indicator often overlooked by current research is the ability to perform in conditions with limited computational resources. Many theoretical results only consider asymptotic behavior. Aiming at a practical implementation of assistance, the focus in our work is to evaluate the solution when only limited resources are allocated, for instance when algorithms are only allowed to run for a fixed amount of time.

1.4 Contributions

The main content of this thesis describes the components of our framework for smart assistants, namely *Collaborative Action Planning with Intention Recognition* (CAPIR). In realistic settings, the framework often has to deal with large planning problems, so a significant part of the thesis is devoted for the investigation of combining an offline learnt heuristic with an online planning algorithm to solve large state-space Markov Decision Processes (MDPs). In summary, our work constitutes the following contributions:

1. A novel problem formulation of two-party coordination which accounts for human-like elements of behavior such as mind switch and bounded rationality (optimal only at subgoal levels).

2. A general framework for smart assistants that scales well with the number of subgoals.

3. A novel enhancement technique, to bootstrap simulation-based algorithms in approximately solving large state-space MDPs; specifically, we describe enhancements to

   (a) *Upper Confidence Bound applied in Trees (UCT)* with convergence analysis and experiment results.

   (b) *Sparse Sampling (SS)* with non-trivial improvement bounds and empirical evidence.

   (c) *Forward Search Sparse Sampling (FSSS)* with empirical evidence.

4. A complete implementation of the CAPIR framework. The engine, ready for use by game programmers, supports two modes of deployment

   - *Client-Server:* The AI planner resides on a centralized server and gaming devices connect to the server for AI-related computation.
   - *Standalone:* The AI planner runs on gaming devices.

Publications

Portions of the work in this thesis have appeared in international peer-reviewed conferences [50, 51] or are under review [53]. Aimed at devising smart assistants in realistic domains, a general description of the framework, together with preliminary human subject evaluation, was first presented by Nguyen et al. [50] at the *Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE)* in 2011. The approach is based on decomposing the global task into subgoals, each of which is modeled as an MDP. In investigating the framework in more complex settings in which the subgoal MDPs have large state spaces, we found the need to adopt approximate methods to solve these MDPs. In the latter stage of the project, the focus was on bootstrapping simulation-based algorithms so that a reasonable approximation can be achieved

quickly. The analysis on enhancing UCT in solving large state-space MDPs was reported at the *23rd European Conference on Machine Learning (ECML 2012)* [51], while the analysis with SS and FSSS has been submitted and is currently under review [53]. In our implementation of the framework, to estimate the behavior of the lead agent, we adopted an adaptive mechanism to dynamically switch between the cheap offline-learnt policy and the computationally demanding, albeit more accurate, online policy. This is a derivative of our earlier work on cost-effective adaptation in non-stationary environments [52], reported at *AIIDE* in 2009.

1.5 Thesis Outline

This thesis describes the work revolving around the development of the CAPIR framework and engine, and it is organized into seven main chapters besides this introduction.

Chapter 2: Background

Since our motivation is derived from the very practical demand of modern video games, i.e., collaborative games, in Section 2.1 we introduce the prominent planning approaches in game AI and their limitations in implementing smart assistants. Next, we will briefly present an overview on decision-theoretic formulations, starting with the classical Partially Observable Markov Decision Process (POMDP) and its subclass MDP. Built on the success of POMDPs and MDPs, there have been various attempts to extend these formulations to the multi-agent settings such as Multi-agent MDP (MMDP), Decentralized POMDP (Dec-POMDP) and Interactive POMDP (I-POMDP). An overview on these approaches is presented in Section 2.4.

Chapter 3: Formalization of Lead-Assistant Collaboration

In Chapter 3, we formalize the Lead-Assistant Collaboration problem using the formalism of MMDP, which exposes two key challenges to the assistant planner. The first challenge is caused by the lack of knowledge on what the lead agent will do when he targets a certain subgoal, while the second challenge is due to the drifting intention of the lead agent. As presented in Chapter 3, these challenges are both **PSPACE**-complete.

Chapter 4: CAPIR framework

To solve the challenges identified in the previous chapter, in Chapter 4, we propose two important assumptions on the lead agent's behavior. As a result, we can devise a general framework, namely Collaborative Action Planning with Intention Recognition (CAPIR), to solve a subset of the Lead-Assistant Collaboration problems. Inspired by recent research in action understanding in the field of cognitive science [4, 3], we avoid the first issue by assuming that the lead agent is optimal only at the subgoal level and tackle the second issue by tracking the likely target using Bayesian inference. In this chapter, the details of the framework are presented, together with qualitative and quantitative assessments in Collaborative Hunters. An evaluation with human subjects shows that the resultant AI system exhibits near-human performance in assisting human players.

Chapter 5 - Bootstrapping UCT

In the CAPIR framework, each subgoal is modeled as an MDP, the solution of which constitutes the knowledge necessary for intention tracking and utility-theoretic action selection. When the resultant formulation yields a state space with overwhelming size, we cannot use exact methods such as Value Iteration

to solve it. In such case, we can use simulation-based algorithms, bootstrapped with heuristics, to approximate the optimal policies of large subgoal MDPs online.

Chapter 5 introduces a novel yet simple method to bootstrap Upper Confidence Bound applied in Trees (UCT) [44], a highly practical algorithm, and shows that the resultant algorithm UCT-Aux outperforms the state-of-the-art bootstrapping methods when coupled with suitable heuristics. The proposed *Aux* method however suffers from a worst case behavior, inherited from the original UCT algorithm due to the use of optimal bandit-based controls, such as UCB1 [2], for exploration. In particular, if the coupled heuristic is not "extreme", a characteristic that might not be known *a priori*, UCT-Aux requires super-exponential time before its convergence to the optimal policy starts.

Chapter 6 - Bootstrapping SS and FSSS

To alleviate the UCT-Aux's worst case behavior, we examine the application of the Aux bootstrapping method in other simulation-based algorithms that do not exhibit similar worst case behavior. In particular, Chapter 6 shows that the bootstrapped version of Sparse Sampling (SS) [42], *SS-Aux*, yields guaranteed improvement in theoretical terms, and that FSSS-Aux, the bootstrapped Forward Search Sparse Sampling (FSSS) [75] algorithm, empirically outperforms UCT-Aux in a long run, even when UCT-Aux is coupled with preferable heuristics. On the other hand, the greediness of UCT-Aux allows it to converge faster to locally optimal values. Therefore, if ample computing resources is available, FSSS-Aux should be adopted; otherwise, UCT-Aux should be preferred.

Chapter 7: Conclusions

Finally, in Chapter 7, we summarize the results reported in this dissertation and discuss possible extensions of the CAPIR framework together with future work.

Chapter 2

Background

Traditionally the behavior of NPCs in games is hand-crafted by programmers using techniques such as scripting [23], hierarchical Finite State Machines (HFSMs) [29] and Behavior Trees [15]. These techniques are simple and straight-to-the-point, i.e., if designed correctly, the NPCs will behave exactly as intended by the programmers. However, in complex environments of modern games, due to the large number of game states, programmers adopting these techniques have to solve the non-trivial tradeoff between *comprehensiveness* and *production cost*. In order to ensure that the behavior of the NPC is comprehensive, the programmer might end up with a huge script which dictates the NPC's behavior in all possible scenarios. While anticipating all of these scenarios is a daunting task, scripting the behavior for just one such scenario may require substantial work too, especially in complex environments where many virtual entities interact in an intricate manner. Factoring this with the number of scenarios, the human labor needed to create a script-based AI system for a video game explodes quickly with the complexity of the game. As video games cannot be released before their AI system is completed, the lengthy process of constructing such system significantly increases the production cycle length, thus cost.

To alleviate some of the difficulties of having to anticipate all possible sce-

narios, planning-based techniques such as Hierarchical Task Networks (HTNs) [43] and Goal-Oriented Action Planner (GOAP) [55] specify goals for the NPCs and use planning algorithms to search for appropriate actions while in-game, or *online*. We devote the first part of this chapter, Section 2.1, to a quick review of these planning-based game AI authoring techniques. We will also discuss their limitations in constructing a smart assistant for collaborative game environments.

To address this assistance problem, our formulation and proposed architecture builds on the success of decision-theoretic formalisms in sequential decision making. Therefore, in subsequent sections, we provide an overview of planning frameworks from single agent setting (Section 2.2 and 2.3) to multi-agent setting (Section 2.4).

2.1 Automated Action Planners in Games

The authoring of NPC behavior via planning yields several benefits that make it an attractive alternative to the more commonly used non-planning techniques such as scripts or Finite State Machines; most notable benefits include unexpectedness handling, code reusability/maintenance, and data-implementation separation [56]. There are two lines of work that implement automated planning in games and are currently in active use: Hierarchical Task Networks (HTNs) and Goal-Oriented Action Planner (GOAPs). These two approaches build on the original idea of STRIPS planning [27] which computes a sequence of actions to transform the world to a desired state.

In STRIPS, states are expressed in the form of a set of literals. The search space consists of operators in the form of (pre, del, add) where pre is the set of preconditions for the operator to be applicable, while del and add update the set of state literals as the result of applying the operator. STRIPS planning refers

to the construction of action plans by sequentially adding actions (operators) to the final plan. Depending on the direction of adding (from initial state, from goal state, or arbitrarily), it is called forward-chaining, backward-chaining or plan-space search.

2.1.1 Hierarchical Task Networks

Hierarchical Task Networks (HTNs), termed Procedural Nets [65] when devised in the 1970s, are offline crafted abstract structures that are used to guide the process of online STRIPS planning. Instead of sequential construction of action plan, HTN algorithm uses expert knowledge encoded in these HTNs to resolve an initial task to primitive subtasks that are directly executable.

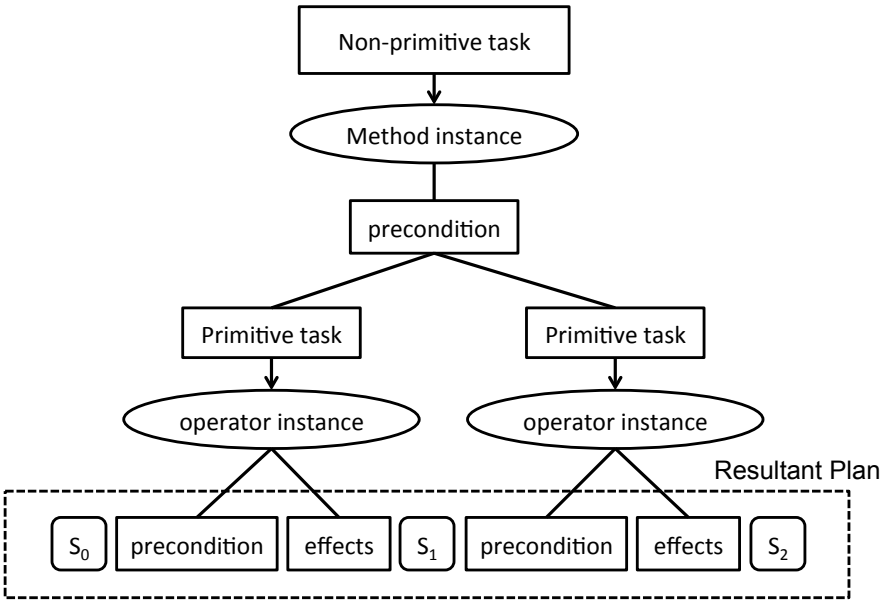


Figure 2-1: General form of task networks expressing decomposition methods. Each *method instance* defines one way of decomposing a non-primitive tasks into other task. Each *operator instance* defines the action and effects of a primitive task.

There are two types of tasks in HTNs: non-primitive and primitive. Non-primitive tasks are decomposed into lists of other tasks, further enhanced by restrictions such as precedence of tasks, variable binding or mutual exclusion. Primitive tasks are directly executable. Each non-primitive task could have more

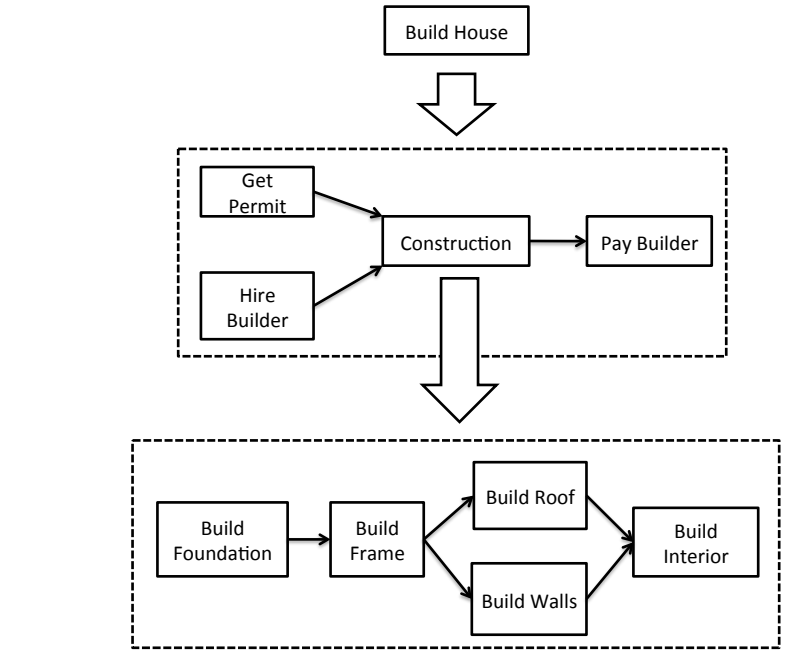
than one method instances (Figure 2-1).

The planning process takes as input one or more initial tasks. It then repeatedly decomposes the partially ordered tasks into simpler tasks until all tasks are primitive (Figure 2-2). If there are restriction conflicts detected at any point of time, the process backtracks and tries new decomposition steps.

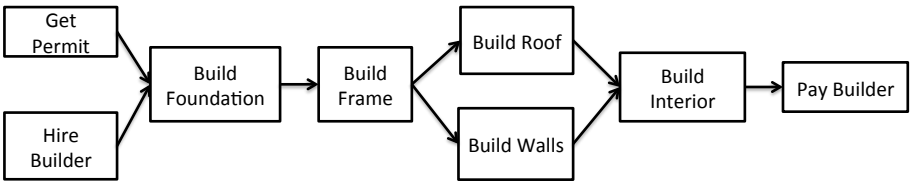
For example, suppose a character wants to build house and his knowledge base consists of two pieces of knowledge: one task breakdown for *Build House* and one for *Construction* (Figure 2-2a). When the task problem with the initial state of (*Money, Land*) and the desired goal state of (*House*) is fed into the planner, the non-primitive tasks are repeatedly broken down from *Build House* to the valid plan in Figure 2-2b.

HTNs, being much more modular than Behavior Trees or HFSMs, could be reused in game titles with similar settings. Typically method instances of higher-level tasks are more abstract and reusable than those at the lower levels. For instance, the decomposition method of *Build House*, as expressed in Figure 2-2, could be used without change in any game that requires the human player to build cities. The method instance of *Construction*, however, could require some change to fit each actual game.

Characteristics. Unlike hand-crafted scripts, HTN planning allows dynamic generation of plans that are most suitable for the current circumstances. However, it relies completely on the correctness of predefined task networks that encode the prior knowledge. Naive HTN planning is known to be undecidable if the set of task networks has tasks that form recursive calls (cycle). It is easy to avoid the infinite looping, for example by limiting the number of actions in a plan, but that does not improve the quality of the returned plan: the NPC would still act in a looping manner. In a huge project, where the design of an AI system spans months or even years and is performed by many individual programmers, the effort to maintain the consistency of the task network set grows quickly with



(a) Hierarchical Task Networks



(b) Resultant plan

Figure 2-2: Example of HTN planning. The action plan for the initial task of *Build House* is constructed after two steps of decomposition. The first step applies one method instance of *Build House* and the second applies that of *Construction*.

the scale of the project.

2.1.2 Goal-Oriented Action Planner

Goal-Oriented Action Planning or GOAP [57] is a planning architecture that combines the power of A*, an effective search technique widely used in games, and the modularity of operators in STRIPS. This technique dynamically chains operators in the action space in real time, using search algorithms such as A*, by matching their preconditions and effects to direct the subject NPC's behavior (Figure 2-3).

While HTN is a way to bootstrap STRIPS planning with prior knowledge's

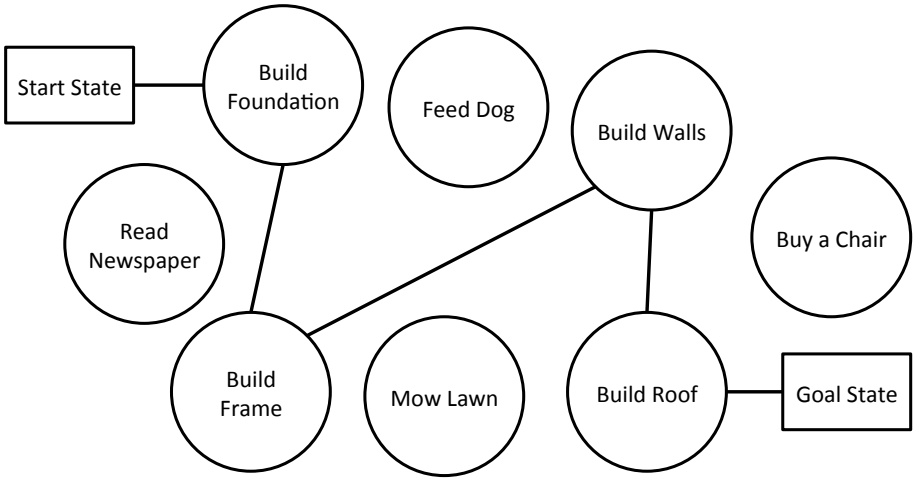


Figure 2-3: The plan formulation process.

guidance in the form of task networks, GOAP is a slight expansion of traditional STRIPS to make it practical for real-time games such as F.E.A.R. [57]. At the core, GOAP is STRIPS planning: The system uses a search algorithm to construct action sequences that transform the game world from an initial state to a desired goal state. However, various modifications are introduced in GOAP:

1. *Cost per Action*. This works as an additional directorial layer. If the game designers want an NPC to prefer acting in one way more than the other, they can decrease the cost for those actions.

2. *No Add/Delete Lists*. Using fixed-sized arrays to represent the preconditions and effects removes the need to handle data structures of variable sizes, thus speeds up the planning process considerably.

3. *Procedural Preconditions*. This helps overcome the limitations introduced by fixed-sized arrays when they could not capture all the information needed for the precondition matching process. Additional checking can be done in an external routine.

4. *Procedural Effects*. Since the effects of actions in real-time games are not instantaneous as in STRIPS, the updating of world states is done gradually

as the action plan is executed.

Other than STRIPS planning, GOAP also uses other techniques that do not relate to action planning but give positive impressions of squad coordination such as intention vocalization. For instance, in troop attack, when an enemy AI soldier realizes he is the last surviving member of his squad, he says something along the line of "I need reinforcements." As the game progresses, the player surely encounters more and more enemies, thus interpreting what he heard as a squad order that other fellow soldiers of the first soldier have complied to.

2.1.3 Limitations

As compared to traditional scripting, FSMs and Behavior Trees, HTN and GOAP provide more elegant tools in authoring NPC behavior via the use of online planning. However, while particularly good in creating opponent NPCs, there are a few fundamental limitations of these solutions in creating smart assistants for a human player.

The first drawback is that these solutions do not explicitly condition their behavior on the human's intention. As a result, it is implicitly assumed that the appropriate reaction of the assistant is deterministic given the current game state. Whenever the game is in that state, the NPC behaves the same way. In fact, they do not account for any hidden or only partially observable element in the decision making process.

Another limitation is that since the solution is stringed from many solution fractions, the AI system still needs a significant amount of domain-specific expert knowledge to build. In HTN, the task networks need to be designed offline, and in GOAP, it is the space of possible actions that the programmer has to construct before game start. Especially in GOAP, selecting an appropriate action space that well trades between speed and behavior variety is an art. Fur-

thermore, ill-tested actions with incorrect preconditions and effects constitute potential bugs that can surface in unanticipated scenarios.

Adopting decision-theoretic research approaches can alleviate these drawbacks. Firstly, decision-theoretic solutions explicitly account for partially observable elements such as human intent in selecting actions. Specifically, the Partially Observable Markov Decision Process formulation conditions the output actions on the belief of latent elements. Secondly, the solver of these formulations compute the optimal course of actions for different level layouts from the knowledge of only basic, level-independent capabilities of the agent. AI authoring becomes completely descriptive instead of imperative, freeing the game programmers from having to transfer their domain-specific expertise to the AI system. We will next present an overview of these decision-theoretic planning formulations, starting with Markov Decision Processes.

2.2 Markov Decision Process

Many sequential decision making problems in practical settings can be formulated naturally as Markov Decision Processes (MDPs) [9]. In this formulation, the *agent*, or the *planner*, is assumed to have a complete perception of the factors relevant to his decision making. However, despite the full knowledge about the current state, the outcomes of the agent's actions are typically stochastic. In the MDP framework, the planning task is reduced to an optimization problem in which the agent's aim is to maximize its long-term accumulated reward.

2.2.1 Formulation

A Markov Decision Process characterizes a planning problem with tuple (S, A, T, RD, γ), in which

- S is the state set,

- A is the action set,
- $T(s, a, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability of the agent moving to state s' when action a is taken in state s at time t. Alternatively, we may refer to $T(s, a) = T(s, a, *)$ as the respective probability distribution; this distribution is assumed to be time-independent.
- $RD(s, a, s')$ is the probability distribution of the immediate reward received after the state transition from s to s' triggered by action a. In many cases, it suffices to execute planning with the knowledge of the expected reward $\mathbb{R}(s, a, s') = E[RD(s, a, s')]$ instead of the full distribution. We can further denote $R(s, a) = \sum_{s'} T(s, a, s')\,\mathbb{R}(s, a, s')$ as the expected reward of executing action a in state s.
- $\gamma \in [0, 1]$ is the discount factor to downplay temporally remote rewards.

As formulated above, each MDP model satisfies the Markov property: the effect of an action (the next state distribution and the immediate reward) only depends on the current state.

Simulator or generative model of an MDP

In modeling a problem as an MDP it may be overly complicated to obtain exact numerical mappings between state-action pairs and resultant rewards $R(s, a)$ as well as next states' probabilities $T(s, a)$. A *simulator*, or *generative model*, of an MDP is a low-cost alternative to the explicit representation. While the full specification of transition and reward functions is akin to a transparent box, a simulator plays the role of a black box that takes as input a state-action pair (s, a) and outputs a valid next state and reward with probability governed by the distribution associated with (s, a) (Figure 2-4).

Intuitively, the availability of the transition and reward functions entails the simulator, but not the other way round. For instance, in modern video games,

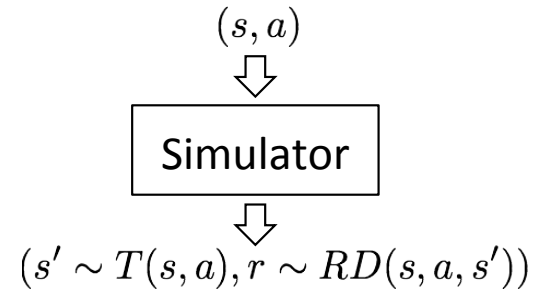


Figure 2-4: A generative model of an MDP.

which always provide a simulator of the game virtual world, it is much easier and faster to get a next state sample than to compute the probability of landing in that state. Specifically, in order to sample a next state, the simulator can sequentially apply each player's action on the given state and update the state according to the actions' effect using some random number generator, while collecting the accumulated reward or cost. In contrast, obtaining the transition probabilities requires the collation of much more information about the game components' dynamics.

2.2.2 Solving MDPs

A *policy* π is a possibly stochastic function that returns an action $\pi(s) \in A$ for every state $s \in S$. In infinite-horizon, discounted MDPs, the value function V and the action value function Q of a policy π measure the expected long-term utility of following π, starting from a state, e.g., s_0:

$$V^{\pi}(s_0) = \mathop{\mathbb{E}}_{T}\left[\sum_{i=0}^{\infty} \gamma^i R(s_i, \pi(s_i))\right], \text{ and} \tag{2.1}$$

$$Q^{\pi}(s_0, a) = R(s_0, a) + \gamma \mathop{\mathbb{E}}_{s' \sim T(s_0,a)}\left[V^{\pi}(s')\right] \tag{2.2}$$

The objective of solving MDPs is to obtain a policy π^* with the maximum expected values in all the states, i.e., $V^*(s) \geq V^{\pi}(s), \forall \pi, s$. A common approach

is to first compute the optimal Q-function Q^* without the knowledge of the optimal policy π^*, and then construct the corresponding $\pi^*(s) = \text{argmax}_a\, Q^*(s, a)$. Value Iteration [9] is an efficient algorithm that adopts this approach.

Value Iteration

If we have the optimal value function V^*, the optimal Q-function can be constructed as follows:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')(R(s, a) + \gamma V^*(s')),$$

which then entails the optimal policy by taking the action that has highest Q^*-value at any state.

For obtaining the optimal value function V^*, a simple and effective algorithm was proposed by Bellman in 1957 [9]. This value iteration algorithm maintains a value function $V(s)$ and iteratively updates the value of each state using the Bellman equation

$$V_{t+1}(s) = \max_a \left(\sum_{s'} T(s, a, s')(R(s, a) + \gamma V_t(s')) \right).$$

This algorithm is guaranteed to converge to the optimal value function $V^*(s)$, which gives the expected cumulative reward of running the optimal policy from state s.

Policy Iteration

Policy Iteration [38] computes the optimal policy by directly searching in the policy space for the best performing one. Starting with some initial policy, its steps alternate between *policy evaluation* and *policy improvement*. Policy eval-

uation computes the value of a policy π by solving the equations[1]

$$V(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} T(s, \pi(s), s')V(s'), \tag{2.3}$$

while policy improvement induces a new policy based on the value function obtained in the previous step

$$\pi(s) \leftarrow \underset{a}{\operatorname{argmax}} \left(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V(s') \right).$$

The algorithm terminates when π converges, at which point π is guaranteed to be optimal.

Intractability

Note that aforementioned exact algorithms require a running time that scales proportionally with the MDP's state space size. Therefore, the key issue that hinders them from being widely used in solving real-life planning tasks is the large state space that is often required to model realistic problems.

Typically in game domains, a *state* needs to capture all the essential aspects of the current configuration, thus may include a large number of state variables. For instance, in a maze game with size m (number of valid positions) consisting of n characters, the set of states is of size $\Theta(m^n)$.

2.2.3 Simulation-based Approximate Approach

Both exact algorithms, Value Iteration and Policy Iteration, require polynomial time with respect to the size of the state space [59, 64]. However, this means that for complex planning tasks modeled as MDPs with very large or even infinite (continuous) state spaces, these exact algorithms require a long running

[1] One way to solve this linear system is by iteratively applying the operator (2.3) at all states until convergence.

time. In many real life domains where only limited resources are allocated, this computational demand may be undesirable.

It is worth noting that the long running time requirement of these algorithms originates from the fact that they both take an exhaustive approach. The optimal action for every state is computed before execution time, just in case such a state will be encountered during the actual deployment of the policy. However, in complex domains such as video games, usually only a fraction of the possible states are ever encountered. With that insight, one alternative in solving MDPs is to adopt a "lazy" approach towards finding optimal actions. This approach does not compute the optimal actions beforehand, but does it "lazily" by computing only for a given current state. The optimal action is then executed to transit to a new state before the process is started all over again. Adopting this idea, simulation-based approaches constitute a highly practical class of algorithms which approximate the action values "lazily". These algorithms estimate the Q-values of actions starting from a given current state by collecting reward statistics from a randomly sampled look-ahead tree that covers only a small fraction of the state space around the current state. The branching factor and depth of the tree can be constrained to suit the allocated computing resources, trading precision for computation. Simulation-based algorithms have recently achieved noticeable successes when applied in practical domains [31, 28, 5]. Among the members of this class, three algorithms stand out as notable representatives, namely Sparse Sampling (SS), Upper Confidence Bound applied in Trees (UCT) and Forward Search Sparse Sampling (FSSS).

SS, proposed by Kearns et al. [42], is the first reported algorithm that produces near-optimal solutions with no dependence on the problem's state space size, given only a generative model of the MDP. Its theoretical characteristics provide a solid foundation and inspiration for UCT and FSSS; more in-depth analysis of SS is presented in Chapter 6.

UCT [44], one of the most successful simulation-based algorithms, is an improvement over SS by *adaptively sampling* the neighborhood. The algorithm makes better use of computing resources by focusing tree expansion towards regions that are more likely to be optimal, thus more relevant to the value approximation. UCT leads to some recent advancements in solving notoriously hard problems such as the board game Go [31, 19] or Real Time Strategy Games [5]. More details on UCT are provided in Chapter 5.

Finally, FSSS [75] is another recent successor of SS that has been instrumental in a state-of-the-art reinforcement learning framework when dealing with large domains. We will discuss FSSS more thoroughly in Chapter 6.

2.3 Partially Observable Markov Decision Process

Partially Observable Markov Decision Processes (POMDPs) [40] are generalizations of MDPs. They formulate the general case in which the planner only has partial perception of the state. As a result, the action policy operates on the space of state beliefs instead of states. This modification increases the generality of the framework, but at the cost of reducing its practicality: while the problem of finding optimal policies for MDPs is **P**-complete, the respective decision problem extended to POMDPs is already **PSPACE**-hard [59].

2.3.1 Formulation

Formally, a POMDP is described by a tuple $(S, A, T, RD, \gamma, O, \Omega, b_0)$, in which (S, A, T, RD, γ) are the same as those described in the MDP formulation, and

- O is the set of observations,
- $\Omega(a, s', o) = P(o_{t+1} = o \mid a_t = a, s_{t+1} = s')$ defines the probability of receiving observation o when action a, taken at time t, causes the transition

to next state s', and

- b_0 is the initial state distribution, or *belief*, with $b_0(s_i)$ being the probability that the agent starts in state $s_i, i = 1..|S|$, i.e., $0 \leq b_0(s_i) \leq 1$, and $\sum_{i=1}^{|S|} b_0(s_i) = 1$.

As mentioned above, since there is no direct access to the state, the agent needs to base his decision on state beliefs. Given the observation o after taking action a when believing b, the state belief can be updated to b' using Bayes rule as follows

$$b'(s') = \frac{\Omega(a, s', o) \sum_{s \in S} T(s, a, s') b(s)}{Z}, \tag{2.4}$$

with Z being the normalizing constant independent of s'.

2.3.2 Solving POMDPs

A *policy* π in POMDP is a function that returns an action $\pi(b) \in A$ for every state belief b. The objective of solving a POMDP is to obtain a policy π^* that maximizes the expected long-term reward with respect to the initial belief. Since the state belief b as updated in 2.4 is a sufficient statistic about the current state, the POMDP is equivalent to a continuous state "belief MDP", whose state space is the belief space and the rest of the components can be constructed from components of the original POMDP. As shown by Kaelbling et al. [40], the solution of this belief MDP is also the optimal policy for the original POMDP. More in-depth discussions on the conversion and adapted value iteration routines to compute the optimal policy of the belief MDP is beyond the scope of this thesis. Interested readers may consult the original paper by Kaelbling et al. [40].

Finding the optimal policies in general POMDPs is **PSPACE**-hard. Therefore, recent advancements are largely in the realms of approximate approaches [62, 72, 45, 70].

2.4 Decision-theoretic Approaches in Multi-agent Settings

In previous sections, we have discussed decision-theoretic formulations for single-agent planning tasks formulated as MDPs and POMDPs. This section is devoted to a background review on numerous attempts to extend MDP and POMDP to the multi-agent setting. These formalisms build the foundation for our Lead-Assistant Collaboration formulation in Chapter 3.

2.4.1 Multi-agent MDP

Multi-agent MDP (MMDP) [11] models a fully cooperative team-work setting in which all agents work towards the same goal. This is a straightforward extension of MDP by conditioning the rewards and transformations of world states on the actions of multiple agents instead of one as in the case of MDP. Specifically, an MMDP can be defined as a tuple $(I, S, A_i, T, RD, \gamma)$ where

- I is the set of agents, identified by $i \in 1, 2, ..., n$ with $|I| = n$.
- S is the set of states.
- A_i is the set of agent i's actions. For convenient reference, we denote $A = \times_{i=1}^{n} A_i$ and joint action $\overrightarrow{a} \in A$.
- $T(s, \overrightarrow{a}, s') = P(s_{t+1} = s' \mid s_t = s, \overrightarrow{a}_t = \overrightarrow{a})$ is the probability of the agent being in state s' when joint action $\overrightarrow{a}$ is taken in state s at time t,
- $RD(s, \overrightarrow{a}, s')$ is the probability distribution of the immediate reward received by the team after transiting to state s' from state s by executing the joint action $\overrightarrow{a}$.
- $\gamma \in [0, 1]$ is the discount factor.

Since the agents are taken as acting on behalf of one *individual*, solving an MMDP results in a joint action policy that maps each state to an action vector which each agent can extract its own part to execute in coordination with the rest of the team. In other words, the planner assumes a centralized perspective of planning with the assumption that all agents will strictly follow the joint action policy. In practice, this assumption is often violated when the assembled team consists of heterogeneous members that come from different backgrounds, such as *ad hoc team* setting [73] or the Lead-Assistant Coordination problem studied in this thesis.

2.4.2 Decentralized MDP

Decentralized MDP (Dec-MDP) [10] takes the team coordination problem as formulated by MMDP one significant step further. While MMDP assumes that each team member has a complete view of the global state, Dec-MDP considers the case when each agent only has partial information about the state. Dec-MDP imposes that the state is *jointly observable*, i.e., by combining the agents' partial views of the world, the full state is completely recovered, hence, completely observable albeit decentralized. Consequently, the key challenge in this class of problems is in terms of information collation.

The general Dec-MDP formulation is known to be **NEXP**-complete even when there are only two agents in the team, in contrast to MDP's **P**-completeness and POMDP's **PSPACE**-hardness [59, 49]. Note that while it is unclear whether **P**, **NP** and **PSPACE** are distinct, it is *known* that **NEXP** $\neq$ **NP** [69], thus Dec-MDP is truly intractable. Moreover, if we assume that **EXP** $\neq$ **NEXP**, the problems require super-exponential time to solve in the worst case.

Therefore, most research on this problem concentrates on subclasses of Dec-MDP. For instance, most works solve the Dec-MDP subclass in which limited communication for plan synchronization is allowed [54, 77, 7, 6] or the agents

are related only through the reward function, and not the transition function (*transition independence*) [7, 6]. In our formulation of Lead-Assistant Collaboration, the assistant is assumed to have a complete view of the global state, thus constituting a different set of challenges.

2.4.3 Interactive POMDP

Departing from the assumption that the agents are fully cooperative in MMDP and Dec-MDP, Gmytrasiewicz and Doshi [33] proposed a decision-theoretic framework, *Interactive POMDP* (I-POMDP), for sequential planning in environments populated with other, possibly adversarial, agents.

The key innovation of I-POMDP is that the framework accounts for the reasoning processes that govern other agents' behavior by maintaining a belief over both physical states and models of other agents. Similar to that in POMDP, Bayes' rule is used to update this *interactive* belief given current belief, last executed action, and the newest observation. The authors also showed that important properties of POMDPs, such as convergence of value iteration, the rate of convergence, and piece-wise linearity and convexity of the value functions carry over to the framework [33].

Unfortunately, the generality of the framework is traded for a huge sacrifice in terms of practicality: the largest simulated problem examined has only 36 physical states [25].

2.5 Chapter Summary

In this chapter, we have provided an overview on planning techniques that alleviate the laboriousness of hand-crafted scripts, hierarchical FSMs and Behavior Trees in constructing NPC behavior. However, while particularly good in creating opponent NPCs, these approaches cannot appropriately deal with un-

observable elements that are inherent to a human player's behavior such as his intention. Moreover, the design of such system still requires significant expert knowledge in designing the solution space. We therefore shifted our focus to decision-theoretic approaches such as MDPs, POMDPs and their generalizations in mult-agent setting (MMDP, I-POMDP and Dec-MDP), which address these problems naturally. By characterizing NPCs as utility-theoretic entities, these formulations produce agents that condition their behavior on all the hidden elements, including the human intention. Moreover, the behavior is computed automatically without requiring any prior knowledge about the solution.

In chapter 3, we are going to describe in detail and formulate the problem of Lead-Assistant Collaboration, using the formalism of Multi-agent MDP. Since the crucial component of the formulation that informs the planner about the lead's behavior is not available before execution time, even with simplifying assumptions, two major challenges remain for the planner: the lead's intention and his goal-oriented behavior. As proven analytically, these challenges are still very hard: each of them is by itself a **PSPACE**-complete problem. Therefore, as will be elaborated in subsequent chapters, further assumptions, inspired by recent findings in cognitive science about human-like reasoning, are proposed to make the problem tractable yet relevant to many practical domains.

Chapter 3

Lead-Assistant Collaboration

In this chapter, we will first describe the decision problem of Lead-Assistant Collaboration and its applicable domains. We then go on to formulate the problem as an optimal decision making problem, which is subsequently proven to be **PSPACE**-hard. For practical use, this general mathematical formulation will be later constrained and implemented in Chapter 4 as our CAPIR-framework and engine.

In the literature of multi-agent coordination there is usually no assumption on the number of involved agents. However, with this generality comes low practicality: since the number of decision making elements grows linearly with the number of involved agents, we often encounter the so-called *Curse of Dimensionality* that makes coordination problems intractable even in trivially small domains. In this work, we focus on the coordination of two agents. This constrained setting allows us to identify many core issues of coordination, especially when one of the agents is a human. These insights allow us to impose further simplifications and assumptions on the general problem to devise practical solutions for two-agent collaboration.

In this work, we use the modern game domain, specifically *collaborative games*, as our motivation and testbed. In fact, as presented in the following

section, by emphasizing the team-work aspect of game playing, this genre has recently attracted much attention and received highly critical acclaim from both critics and gamers alike; more details will be presented in Section 3.1.1.

3.1 Applicable Domains

The problem of Lead-Assistant collaboration is a special case of two-party collaboration which depicts the cooperation of two agents in achieving common objectives. One of the agents takes the role of a *lead agent* that with more power or authority plays the role of a leader and mastermind. The other agent, the *assistant*, is less capable and its purpose is to help the lead agent to carry out the tasks. The lead agent is assumed to know the optimal way to solve the tasks, but this usually involves assistant's aid; in some cases, it may be impossible for the lead-agent to achieve the goal without the help of the assistant.

This type of collaboration occurs in many domains that require coordination to complete common objectives. For example, when coordinating robot rescue teams operating in remote and dangerous regions, instead of giving all robots equal rights and influence to the overall action plan, we may assign one robot as the team captain and all the other as its subordinates; this role can be reassigned if needed, such as in times when the current captain runs out of energy. By having biased relationships, we can centralize the decision making process and avoid many complications arising when all members of the team are given the same share of responsibility in establishing a common action plan. For instance, since the captain has the final word on what the plan is like, the communication cost expended in order to agree on one common action plan is greatly reduced.

Although the formulation and analysis here can be applied in many realistic domains, in this work, we use the domain of modern collaborative games as our primary medium of illustration. In fact, motivated by recently growing interest

in collaborative games, there is a genuine demand in this domain for smart AI assistants.

3.1.1 Collaborative Games

The idea of having a partner while completing game objectives is not new. It has in fact been the key element for many classic titles dated as far back as the 1990s such as the series Contra by Konami, which features single and two-player modes whereby players control Rambo-styled characters to attack enemies' bases. However, the "team" here has almost always comprised of all human players or all computer agents, each member of which could lead his own path as there is no apparent need to rely on the others. Only recently has the idea been revisited with a stricter concept of cooperation: the players must rely on their teammates' unique abilities in order to overcome challenges and proceed the game. For instance, *Collaborative Hunters*, the game described in Figure 3-1a, pits two players against running Wolves through a maze of escape paths. If the two protagonists are unable to work out a collaborative plan to surround these wolves, perhaps one by one, they could run out of time fruitlessly.

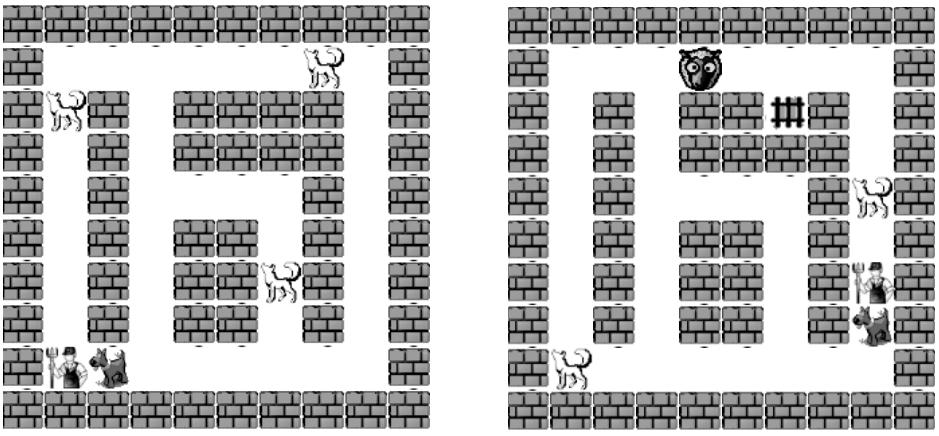
Commercial examples of Collaborative game genre include the critically acclaimed game series Left 4 Death[1] by Valve Corporation and Army of Two[2] by Electronic Arts. Recently, the title Lara Croft and the Guardian of Light[3], featuring collaboration as a key game mode, released on Xbox, has been placed at #14 in the top 25 Xbox Live Arcade titles of all times. Another representative worth mentioning is the new installment of the stealth game series *Assassin's Creed*, namely *Assassin's Creed: Brotherhood*[4], which includes a multi-player coop game mode that pits at least three teams of human players against one

[1] http://www.l4d.com
[2] http://games.ea.com/armyoftwo/home.jsp
[3] http://http://www.laracroftandtheguardianoflight.com/
[4] http://assassinscreed.uk.ubi.com/brotherhood/

another in a cat-and-mouse play style; this game mode essentially earned them the *Best Multi-player Game* title at Electronic Entertainment Expo 2010 in Los Angeles [30]. Typically, this game type pits a small group of players against challenges that they need to solve collaboratively in order to advance to next stages. We expect coop game modes with smarter AI teammates will show new doors of game play to the game industry, which is estimated at over $100 billion US dollars as of June 2011 [35].

We are looking specifically into applying decision-theoretic research techniques in two-player collaborative games such as *Lara Croft* or *Army of Two*. One player-character is supposedly controlled by a human gamer while the other's behavior is dictated by our action planner. Each of them could have different sets of capabilities and most if not all of the tasks require their teamwork to complete; without a partner, it could be impossible to achieve the game objectives.

We will next introduce a couple of coop games that will be used as illustrative examples in subsequent chapters.



(a) Collaborative Hunters (b) Sheep Farmer

Figure 3-1: Two collaborative games we are using for illustrative purpose. In *Collaborative Hunters*, The protagonists, Farmer and Dog in the bottom right corner, need to kill all three wolves to pass the level. In *Sheep Farmer*, they need to herd the sheep into its pen while trying to protect it from being killed by the wolves.

Example Game 1: Collaborative Hunters

A sample of our targeted class, which is going to be used for illustrative purpose, is described in Figure 3-1a. In this game, called *Collaborative Hunters*, the assistant (illustrated as a dog) has to help the lead agent (the farmer) kill several wolves in a maze-like environment. A wolf will run away from the agent or assistant when they are within its vision limit, otherwise it will move randomly. Since wolves can only be inflicted damage by the farmer, the dog's role is strictly to round them up. Note that collaboration is often truly required in this game - without surrounding a wolf with both players in order to cut off its escape paths, capturing a wolf can be quite difficult.

Example Game 2: Sheep Farmer

Sheep Farmer is an extension of *Collaborative Hunters*. In this game (Figure 3-1b), the AI-controlled dog has to help the human player to herd one Sheep NPC into its pen in a maze filled with Wolves. Sheep is pursued by Wolves but is able to defend itself by running away from Wolves while being chased, although at the same speed as Wolves'. Any of these NPCs will move away from the protagonists when they are within the vision limit. If the Wolves are next to the Sheep, they will kill the Sheep instantly, which means the end of the game. The protagonists' task is therefore more complicated than the previous game: they must carry out the task while protecting the sheep from Wolves' threats by blocking Wolves' runway or killing them. Killing Wolves or successfully herding the Sheep into its pen gives them some reward.

3.2 The general formulation

In the following formulation of the problem, we will use Collaborative Hunters as illustration, with the Dog being the assistant and Farmer the lead agent.

We will next formalize the setting in which the lead-assistant behavior takes place, using the formalism of Multi-agent MDP. A Lead-Assistant Collaboration problem can be formulated as a tuple $(S, A, A', T, RD, \gamma)$, in which

- the finite **state space** S contains all the possible configurations of pertinent elements of the world. These states are often defined as vectors of state attribute values (state variable values). *Each state in Collaborative Hunters is defined by the locations of Farmer, Dog and all wolves and the number of times each wolf has been hit by the Farmer.*

- the **action sets** A and A' contain actions agents can take to change the state of the world. We denote the actions available to the lead agent by A, and the assistant's actions by A'. *Farmer and Dog each has four movement actions (north, south, east and west) that change the locations of the agents. The Farmer is further endowed with a Shoot action to attack wolves.*

- the **transition function** $T(s, a, a', s') = P(s_{t+1} = s' \mid s_t = s, a_t = a, a'_t = a')$ defines how the performed actions are likely to change the state of the world. The next state may not depend deterministically on actions but there may be some stochasticity due to other (possibly unmodeled) forces. Therefore, the $T(s, a, a', s')$ is defined as a conditional probability that action pair (a, a') in state s will lead to state s'. *Farmer and Dog move deterministically but a wolf may move to a random direction if there are no agents near it.*

- the **reward distribution** $RD(s, a, a', s')$ defines a distribution of immediate rewards the lead-assistant team receives after the state transition from s to s' triggered by the action pair (a, a'). *In Collaborative Hunters, a non-zero reward is given only if the wolf is killed in that move.* When finding the optimal behavior of of the team, we do not usually need to

know the whole reward distribution RD but it is sufficient to know only the expected rewards $R(s, a, a', s') = E[RD(s, a, a', s')]$.

- the discount factor $\gamma \in [0, 1]$ is to favor rewards received in the near future.

All the aspects of the task environment listed above are assumed to be known to both the lead agent and the assistant. The lead agent follows a possibly stochastic **behavior model (or policy)** $\pi_L : S \rightarrow A$ (or, in case of stochastic policy, $\pi_L : S \rightarrow P(A)$). Note that while the lead policy π_L may be based on the lead agent's idea about the desirable team policy $\pi_T : S \rightarrow A \times A'$ (or its stochastic version), there is generally no constraint on how the lead agent selects its actions. The assistant's choices of supportive actions, however, are conditioned by the history of observed lead-agent actions. Denoting Θ as the set of all possible lead-agent action histories, the problem can now be formalized in our framework as a problem of finding the assistance policy $\pi' : (S, \Theta) \rightarrow A'$ that for all states s_0 maximizes the expected future reward $V(s_0) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t = \pi_L(s_t), \pi'(s_t, \theta_t), s_{t+1})]$, where $\theta_t = (s_0, a_0, s_1, a_1, ..., s_t, a_t) \in \Theta$ is the lead agent action history; the expectation is taken with respect the transition dynamics T and the lead agent policy π_L. If we knew the behaviour model π_L, the optimal π' could be computed in theory by first considering the lead action to be part of the state space and augmenting the transition model T and expected reward model R with π_L, and then solving the resulting Markov Decision Process using standard techniques such as value iteration. In our case, however, the lead agent is often human and the behaviour model is unknown. The optimization problem then involves an expectation over an unknown distribution policy π_L.

In practical domains, there is usually little clue as to what the lead agent policy π_L is like. For instance, in playing modern video games, gamers usually join in with different play styles or preferences. In the same circumstances, some aggressive gamers prefer to attack all in, while some others could choose

a more defensive approach. We will next show that without the knowledge about how the lead agent behaves, the problem of finding an optimal assistant policy is computationally very hard. This will later motivate us to make simplifying (but still realistic) assumptions about the lead agent's behavior so that a good supportive assistant policy can be determined.

3.3 Simplifications

In a general setting, when the lead agent's behavior model wildly varies and might not be cooperative, it is impossible to devise an assistance plan. To better-define the collaboration problem, certain assumptions about the problem settings must be made.

3.3.1 Related work

In a previous work [26], Fern and Tadepalli addressed a subclass of Lead-Assistant Collaboration by imposing two key assumptions. Firstly, the lead agent is presumably firm about his intended target; from the beginning to the end of the experience, he only sticks to one goal and and never changes it halfway through. Under this assumption, the assistance task is formulated as a Hidden Goal MDP (HGMDP), the goal of which is to identify the lead agent's unchanged intention throughout the course of the experience. To further reduce the hardness of the problem, the authors next assumed that the objectives can be completed by the lead agent without any help from the AI assistant; the assistant's role is strictly to speed up the process of completing intended tasks. As a result, the assistant's actions are never detrimental: they are either helpful or useless to the lead agent. This additional assumption leads to a subclass of HGMDP, namely Helper Action MDP (HAMDP). Although more restricted than HGMDP, the hardness of HAMDP is however not reduced; both HGMDP and

HAMDP were shown to be **PSPACE**-complete.

Nonetheless, the above restrictions on the domains and lead behavior allow the authors to devise an efficient policy which yields a bounded number of unhelpful actions. In exchange for the theoretical guarantee, the assumptions, however, limit their solution's practicality. Specifically, not allowing the lead agent to change his mind during the experience (in HGMDP) confines the formulation to short and trivial coordination situations. The solution is therefore inapplicable to more complex domains such as video games, in which each gaming experience usually spans over a long period of time and the involved players change their minds all the time. Besides, assuming that the helper's actions do not impede the lead agent's progress (in HAMDP) plays down the role of team spirit, as the lead agent alone can achieve the goal. In collaborative games whereby game objectives must be completed by the whole team, such assumption is not appropriate.

3.3.2 Problem Specifications

In order to formulate coordination problems in which the team emphasis in problem solving is retained and the lead can change his mind during task execution, we make the following restrictions or specializations on our targeted class of Lead-Assistant Collaboration. The setting can be considered as a generalization of that scoped in Fern and Tadepalli's work [26].

Firstly, let us assume that the assistant is aiding a lead agent whose behavior consists of a set of *subgoals*' behavior models. As such, there are many possible ways for the agents together to achieve the main objective but that each of these ways can be decomposed as a sequence of *subgoals*. In this case at each time point there is a set of subgoals the agents may choose to pursue. **Secondly**, we generally assume that the environment is fully observable to both agents, but the assistant does not directly know the subgoal the lead-agent would like to pur-

sue. The lead-agent is also free to change the subgoal he is pursuing at any given time. The assistant has to infer what subgoal the lead-agent is pursuing in order to maximally support the lead-agent. To make this general problem non-trivial and applicable to a wider class of real-life domains, we assume that the lead-agent can not explicitly communicate its subgoal or the desired assistive actions to the assistant, but that the assistant has to infer the most supportive actions based on the current situation including the observed actions of the lead-agent. This assumption is due to the fact that in many settings, especially when there is time pressure, communicating the intention is not feasible. For example, in a smart home for the elderly, the human-lead, in this case the elder person, would not be able to communicate his desire or even current status when undergoing a stroke. The robot caregiver, if conditioning its behavior on the, albeit minimal, explicit commands from the lead, will fail to timely react to this emergency situation. Similarly in the setting of virtual assistants in video games, during a quick base attack, which is prone to possibly unexpected counter-actions from the enemy, there is no time for the lead player to explain any change of plan to his AI subordinates. The assistants have to figure it out by themselves, by observing how their leader is behaving. Besides, in cases when the lead agent can communicate his intention to the assistant, the problem becomes easier, since the assistant does not need to track the lead's intention anymore. **Lastly**, we lift Fern and Tadepalli's assumptions that the lead agent can finish the episode by himself, i.e., without any help from the assistant, and that the lead agent has to stick to one fixed subgoal throughout the episode.

In this setting, the assistant encounters two main challenges while aiding the lead agent: (1) tracking the drifting subgoal the lead-agent is pursuing (*Subgoal Tracking*), and (2) determining what to do once the intended subgoal has been identified (*Action Planning given Subgoal*). Taking the decision-theoretic approach, we can use the framework of Markov Decision Processes to formu-

late the first challenge as a Changing Goal MDP (Section 3.3.4) and the second as a Hidden Behavior MDP (Section 3.3.3). Changing Goal MDP models the scenario when the hidden element in the assistant's decision making is the lead agent's drifting intention, such as whether the lead agent is trying to get a drink, go to bed, or get a book. In contrast, Hidden Behavior MDP models the scenario when there is ambiguity in achieving a known goal, e.g., the assistant knows exactly that the lead agent wants to read a book, but the information on how he will achieve it is fuzzy (hidden behavior). For instance, in order to read a book, the lead agent could follow the trajectory of grasping a chair before taking a book from the shelf to read at a table, or taking the book from the shelf and reading it while in bed.

In the following sections, we will show that each of the challenges identified above, i.e., Subgoal Tracking and Action Planning given Subgoal, in our setting is **PSPACE**-complete.

3.3.3 Action Planning given Subgoal: Hidden Behavior MDP

If the assistant knows the subgoal the lead-agent is pursuing it has to perform actions that support reaching the goal. However, just knowing the goal is not enough for selecting the most supportive actions; the assistant also needs to know what the lead-agent is going to do in order to achieve the goal.

The field of Player Modeling directly addresses this problem by first assuming that the lead agent's behavior model takes certain structure and then learning the parameters online [37]. This approach has its challenges, like how to choose an appropriate model structure, and how to most efficiently learn the parameters with minimal training data to ensure quality performance as quickly as possible. In general, different problem domains tend to require different player models. Aiming at a more general and automatic solution, we have chosen to pursue a decision theoretic approach.

We are interested in the problem of selecting assistant actions that are most beneficial given only observations about the lead-agent's actions, thus without full knowledge of lead-agent's behavior model. Adopting the Partially Observable MDP (POMDP) formalism, we will show that even a simple case of this problem in which the assistant knows beforehand a candidate set of possible behavior models of the lead-agent is very hard. The problem of finding the optimal behavior of an assistant can be formulated as an POMDP defined by a tuple $M =< S, A, A', T, R, \Xi, I_\Xi >$ in which

- S is the state space, while A and A' are the lead and assistant action sets.
- $T(s, a, a', s') = P(s_{t+1} = s' \mid s_t = s, a_t = a, a'_t = a')$ is the conditional probability that action pair (a, a') in state s will lead to state s'.
- $R(s, a, a', s')$ is the expected immediate reward received after the state transition from s to s' triggered by the action pair (a, a').
- Ξ is the candidate set of lead-agent's behavior models. A lead-agent behavior model $\xi \in \Xi$ defines the probability distribution $\xi(A \mid s)$ of lead actions issued in state s. Note that this information about possible lead behaviors is known by the assistant at decision time.
- I_Ξ is the initial belief distribution over members of Ξ.

In a so called finite-horizon episodic setting, the objective is to find a policy $\pi' : S \times \Theta \rightarrow A'$ that determines the assistant action for each state and lead-agent action history in a way that maximizes the expected discounted reward $E_T \left[\sum_{t=0}^{h} \gamma^t R(s_t, a_t, \pi(s_t, \theta_t), s_{t+1}) | I_\Xi \right]$ in which h is the horizon length. We call this formulation the Hidden Behavior MDP (HBMDP).

Given a HBMDP M, a horizon $h = O(|M|)$ with $|M|$ being the encoding length of M and a reward target r, the *short-term reward maximization prob-*

lem asks whether there exists an action policy for the assistant that achieves an expected discounted rewards of at least r.

Theorem 3.1. *The short-term reward maximization problem for HBMDPs is* **PSPACE***-complete.*

Proof. Each HBMDP problem can be trivially converted to a partially observable MDP, hence it is in **PSPACE**. Since HBMDP extends Hidden Goal MDP (HGMDP) of Fern and Tadepalli [26] by allowing the assistant's actions to be harmful, it is straightforward to reduce from HGMDP to HBMDP. Since HGMDP is **PSPACE**-hard, so is HBMDP, which means HBMDP is indeed **PSPACE**-complete. □

Theorem 3.1 shows that collaboration with a partner is a difficult problem if the partner's behavioral strategy is unknown *a priori*.

3.3.4 Subgoal Tracking: Changing Goal MDP

While the Hidden Behavior MDP concerns with how a collaboration plan can be computed if the lead agent's behavior is unknown, the following formulation characterizes the challenge in guessing which subgoal the lead agent is pursuing, by observing his actions. Unlike in HBMDP, in this setting, we assume that for each subgoal the lead agent only adopts one behavior model, which is known *a priori* by the assistant.

The Changing Goal MDP (CGMDP) problem can be described by a tuple $< S, G, A, A', T, R, \Gamma, \Pi >$ in which

- S is the state space, G the goal set, and A and A' respectively the lead agent's and assistant's action sets.
- $T(s, a, a', s')$ is the probability that action pair (a, a') in state s at time t will lead to state s' at time $t + 1$; T models the environment's dynamics.

- $\Gamma(g'|g, s)$ is the probability that the lead agent decides to pursue g' in the next time slice given current state s and goal g; it captures the "mind change" of the agent due to environment-related factors. We assume the lead agent is not influenced by its assistant's actions.

- For the lead agent's behavior, $\Pi(s, g)$ is the distribution over A. For each action $a \in A$, $Best\_Assistant\_Actions(s, g|a) \subset A'$ is a subset of assistant actions that are deemed most efficient when coupled with the agent's action a in advancing the progress towards goal g.

- $R(s, g, a, a')$ rewards the assistant based on the usefulness of a' when coupled with a in achieving goal g at state s; $R(s, g, a, a') = 1$ if $a' \in Best\_Assistant\_Actions(s, g|a)$ and 0 otherwise.

Similar to the *Action Planning* problem, in a finite-horizon episodic setting, the assistant's objective is to maximize the expected discounted reward $E\left[\sum_{t=0}^{h} \gamma^t R(s_t, g_t, a_t, a'_t)\right]$ in which h is the horizon length. We call this formulation the Changing Goal MDP (CGMDP).

Given a CGMDP M, a horizon $h = O(|M|)$ with $|M|$ being the encoding length of M and a reward target r, the *short-term reward maximization problem* asks whether there exists an action policy for the assistant that achieves an expected discounted rewards of at least r. In the following theorem, we show that this general formulation is hard even under fairly restricted setting.

Theorem 3.2. *Short-term reward maximization for CGMDPs with deterministic environment and no mind change is* **PSPACE***-complete.*

Proof. The proof follows closely that of Theorem 1 for Hidden Goal MDP in [26], since CGMDP with no mind change is actually Hidden Goal MDP (HGMDP), the proposed formulation of assistance by Fern and Tadepalli. By showing that CGMDP is in **PSPACE** and reducing QSAT to CGMDP, it follows that CGMDP is **PSPACE**-complete. □

In deterministic environments with a lead agent that does not change his goal, the problem's hardness comes mainly from the inability of the assistant to interpret the lead agent's goal based on his behavior. This usually happens when different subgoals are not well-separated, so that pursuing them results in similar action sequences.

3.4 Problems of assuming global optimality

For two-party collaboration in which each agent only has partial control and influence on the final result, inaccurate assumptions may lead to a failure to complete the task. In Collaborative Hunters, one obvious approach is to assume that the farmer is globally optimal, i.e., always executes the lead action that is part of some optimal action pair at any state. This assumption leads to a fully observable planning problem for the dog since there is no hidden element in the lead's behavior. The problem is reduced to a classic planning problem for a single agent, whose actions are now the cross product of the dog and farmer's action sets. The dog's actions are then extracted from the resultant optimal policy[5]. If the human controlling the farmer is indeed globally optimal, the dog's behavior would be perfect as a collaborator. However, this seemingly natural approach poses a major issue due to the strong assumption on how the human-controlled agent behaves. Human's expertise in a domain is known to influence their perception about problems in that domain [18, 68]. Novices usually perceive problems on the basis of "surface structure", thus exhibiting behavior that tends to optimize obvious or short-term goals. Experts on the other hand are able to grasp deeper and more complex principles to solve the problems on a larger scale [68]. Conditioning the assistant's behavior on a globally optimal lead model is therefore inadequate, especially in complex settings with many

[5]In cases when there are multiple equally optimal action pairs for a state, the dog's selected action can be conditioned on the observed lead actions to break ties.

intertwining objectives.

Moreover, assuming one rigid, fixed behavior model such as global optimality to assist can be counter-productive when the environment can change beyond the assistant's anticipation. For example, suppose at training phase, a robot assistant is briefed to prioritize the objective of helping the lead to collect rare gems G_1 over that of gems G_2. While deployed in a remote area, such as Mars, it may so happen that this ranking changes, due to the reverse in actual demand of G_1 and G_2, which can then be conveyed to the lead agent. The lead agent therefore has access to information that potentially changes the team's strategy; an assistant that relies on a perfect model obtained at design phase will fail to deliver in such dynamic settings.

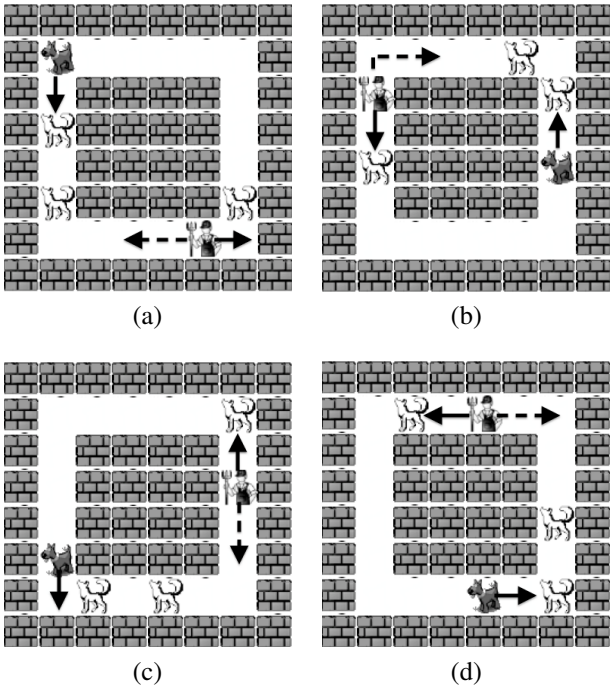


Figure 3-2: If the dog's assumption about the farmer being globally optimal is wrong, they fail to collaborate and could get caught in an infinite circle movement around the map, counter-clockwise from (a) to (c), (d), (b) then back to (a); the dotted lines are the assumed movements of the farmer if he were globally optimal.

In the context of collaborative games with many objectives such as our Collaborative Hunters game, the assumption can be detrimental due to the wrong

idea of the assistant about the lead's behavior. Figure 3-2 shows an example of a case where wrong assumption of global optimality leads to a failure to complete the task. Suppose that Farmer cannot attack a wolf if it is not in an adjacent position. In the depicted scenario, the dog perseveres to chase two wolves nearest to it because during planning time, it knows that a globally optimal farmer would turn around to attack the wolves currently pursued by his dog; that is the best action plan for them. However, if the farmer behaved otherwise, i.e., stubbornly following the wolf nearest to him due to his sub-optimality, they would fail to collaborate and get caught in an infinite circle movement around the map.

In fact, the assumption of subgoal behavior subsumes that of globally optimal behavior, by considering the global goal as just another subgoal of the lead agent in the set of all possible subgoals. This formulation therefore gives the assistant more flexibility in determining the best course of assistive actions, able to collaborate with different human lead types of different expertise levels.

3.5 Chapter Summary

As discussed in this chapter, the problem of Lead-Assistant Collaboration when assisting a self-interest agent could be extremely hard. In particular, two key problems that are inherent to a globally suboptimal lead agent, i.e. unpredictable behavior and hidden intention, are both **PSPACE**-complete.

In Chapter 4, we will make further assumptions to render the problem more tractable. Inspired by recent research advancements in action understanding in the field of cognitive science, we avoid the issue of unpredictable lead behavior by assuming that the lead agent is optimal at subgoal level and tackle the issue of hidden intentions by tracking the likely lead target using Bayesian inference.

Chapter 4

CAPIR - Collaborative Action Planning with Intention Recognition

As presented in Chapter 3, finding the optimal assistant behaviour when the lead agent policy is unknown is computationally demanding and often impossible in practice. Without reasonable assumptions on the lead agent's behavior, the assistant has to face at least two **PSPACE**-hard problems. In order to apply these formulations in realistic domains, we need to reduce the problems' complexity with further simplifications. Assuming the lead agent behaviour to be globally optimal leads to an easier computational task but even after this restriction the time and space requirements may be prohibitive. Furthermore, in case of human lead agents, the assumption of global optimality in complex tasks may be overly optimistic, in which case the assistant behaviour is based on a wrong model. In order to apply these formulations in realistic domains, we need to reduce the problems' complexity with some simplifications

In this chapter, we elaborate our assumptions and restrictions on the general formulation in implementing a framework of assistants. The proposed

framework, namely Collaborative Action Planning with Intention Recognition (CAPIR) [50], adopts the simplified formulations, which model the lead agent as being optimal at the subgoal level and changing subgoals according to a given finite state machine. The lead's targeted subgoal is then tracked online and decision-theoretic action selection is applied to choose suitable assistive actions to execute at game time. Our evaluation with human subjects shows that the framework yields near-human assistance in the two-player game *Collaborative Hunters*.

4.1 Related work

Our work is an interdisciplinary product that lies at the junction of various research topics such as action understanding (cognitive science), plan recognition and planning under uncertainty.

4.1.1 Action Understanding as Inverse Planning

The idea of interpreting an agent's intention by reasoning on his actions has been explored in the field of cognitive science. Taking the planning approach, our work draws its motivation from a cognitive scientific line of works on action understanding using inverse planning, which interprets an agent's intention by reasoning on his actions under the assumption of rational behavior. It is an inverse process based on the popular "belief-desire psychology" which posits that intentional agents' beliefs and desires are the causes of their behavior ([24], [36], [61], [76]). Dennett [24] argues that this relation is governed by the *principle of rationality*: intentional agents are expected to choose most efficient actions that satisfy their desires, given their beliefs of the world.

Recently, Baker et al. [3] have conducted experiments to confirm the validity of simple rational models of actions in representing the human intention

inference. The process is called *inverse planning*, due to the fact that it can be considered the reverse of action planning given a certain intention/desire under specific beliefs of the world. The rational models of actions and goal tracking mechanism they used in their experiments are well-aligned with what we propose in our approach

1. The *human behavior model* is constructed using optimal action plans resulted from MDP solvers.

2. The agent's *intention* is tracked using Bayes' theorem.

The experiments show that human subjects' judgments match the outcomes of the aforementioned model in inferring the intention of an entity based on its actions. In other words, the model successfully performs near-human interpretation of an intentional agent's behavior. This exciting result hints that if this approach is adopted for modern video games, we could create NPCs with the ability to understand players "the human way".

The work however only addresses the situation when the human actor can complete the task alone without any help, focusing more on the reasoning aspect instead of the action aspect. Our focus, on the other hand, is in capitalizing the assistant's understanding about the human partner's intention to select meaningful cooperative actions.

4.1.2 Plan Recognition in Game AI Research

One important part of our work is related to the problem of Intention/Plan Recognition. Since plan recognition was identified as a problem on its own right in 1978 [67], there have been various efforts to solve its variant in different domains. In the context of modern game AI research, Bayesian-based plan recognition has been inspected using different techniques such as Input Output Hidden Markov Models [34], Plan Networks [58], text pattern-matching [47],

n-gram and Bayesian networks [48] and dynamic Bayesian networks [1]. All of these research attempts focus on constructing the most comprehensive data structure to capture the behavior of a subject in each particular domains, so that the subject's hidden plan can be inferred.

The major difference that sets apart these lines of works from our own is that we focus more on the reasoning process and action selection, which comes after plan recognition. Research in Plan Recognition therefore compliments our framework in that their intention tracking models could be used in place of our belief update component. On the other hand, as far as we know, our work is the first to use a combination of precomputed MDP action policies and online Bayesian belief update to solve the plan recognition problem in a collaborative game setting.

4.1.3 Decision-theoretic framework of Assistants

Related to our work in the collaborative setting is the work reported by Fern and Tadepalli [26] who proposed a decision-theoretic framework of assistance, which builds on similar principles as those of the cognitive scientific counterpart. By assuming that the lead agent has *a priori* chosen to follow a task in a predefined set and never changes this hidden intention, they also model each task as an MDP, the optimal policy of which constitutes the behavior model of the lead agent if he selects that task (Principle of Rationality). Bayes' rule is used to track the hidden intention.

Similar to the work on Action Understanding using Inverse Planning [3], there are however fundamental differences between their targeted problem and ours. Firstly, they assume the task can be completed by the main subject without any help from the AI assistant. This is not the case in many cooperative games that present scenarios in which the effort from one lone player would amount to nothing and a good collaboration is necessary to close down on the enemies.

Secondly, they assume a static human intention model, i.e. the human only has one goal in mind from the start to the end of one episode, and it is the assistant's task to identify the sole intention. This restriction allows them to devise a myopic policy that has a bounded expected regret. If the human is allowed to change goals during the episode, such result is not possible. In contrary, our framework allows for a more dynamic human intention model and does not impose a restriction on the freedom of the human player to change his mind mid way through the game. This helps ensure our AI's robustness when inferring the human partner's intention.

4.2 Assumptions and Simplifications

In order to make the task of finding the assistant behaviour computationally feasible under plausible assumptions about the lead agent behaviour, we suggest adopting following simplifications and assumptions:

- The task can be decomposed (by a domain expert) into a set of subgoals.
- the lead agent follows only one behavior model per subgoal; this significantly reduces the complexity of collaborating given subgoal (Hidden Behavior MDP) and allows efficient computation of the assistant's action offline in design phase.

In the current implementation, we further assume that the agent's behavior is optimal at the subgoal level to avoid the phase of encoding a suitable agent model; this assumption can be safely lifted if such an expert elicitation is affordable.

Moreover, in order to further reduce the size of the state space for each subgoal, we implement a simple form of attention by ignoring aspects of the state that do not significantly affect the decision making pertinent to the subgoal.

For instance, in the Collaborative Hunters game, subgoal formulation ignores the presence of NPCs that are not part of the subgoal (Figure 4-1). The MDP for a subgoal consists of only two players and a small set of wolves and hence is likely to have manageable complexity.

In game domains such as Collaborative Hunters, the lead agent is often controlled by a human player with the assistant accompanying him; therefore, in subsequent sections, we are going to refer to the lead agent as *human player* interchangeably.

4.3 CAPIR - Formulation

By assuming that the lead agent is optimal at the subgoal level, we do not have to deal with the issue of the lead agent changing his behavior while completing subgoals, as modeled by HBMDP. CAPIR models the assistant's planning problem with three components: Subgoal MDPs, Human Subgoal Behavior and Goal Change Model. These components are used to track the drifting intention of the lead agent (CGMDP), and to select the most assistive actions when the hidden intention is inferred.

4.3.1 Subgoal MDPs

Following the MDP formulation proposed in Section 3.3.3, under the assumption of the human player adopting optimal behavior at subgoal level, the assistant's subgoal planning task can be described by tuple (S, A, T, RD, γ) in which

- S is a finite set of game states. *In single wolf subgoal, the state consists of the positions of the human player, the assistant and the wolf, together with the number of hits it has taken.*
- A is a finite set of action pairs available to the players; each action $a \in A$

could be a compound action of both players. *If the human player and the assistant, each has 4 moves (north, south, east and west), A would consist of the 16 possible combination of both players' moves.*

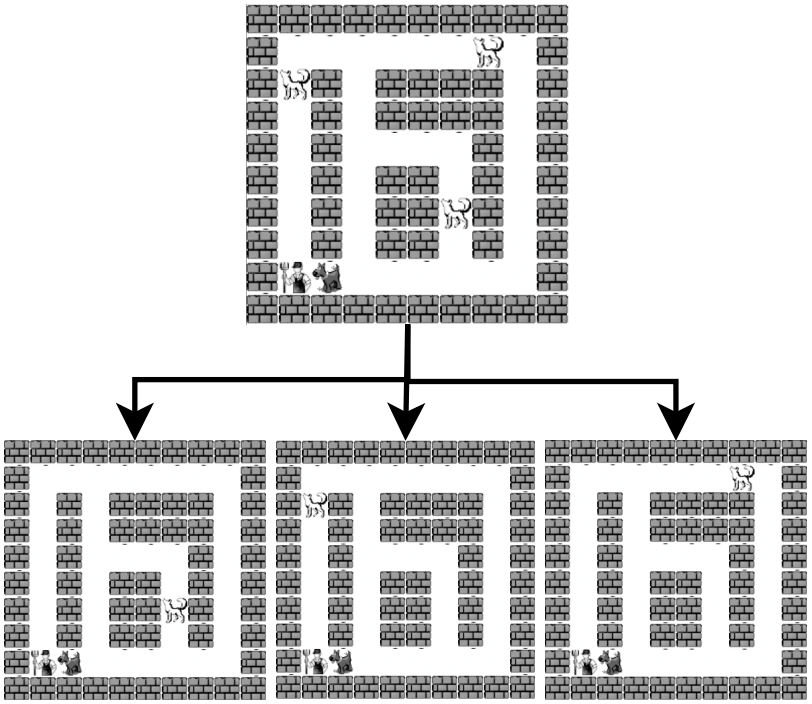


Figure 4-1: Subgoal formulation in Collaborative Hunters.

- $T(s, a, s') = P(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$. *The human and assistant move deterministically in Collaborative Hunters but the wolf may move to a random position if there are no agents near it.*

- $RD(s, a, s')$ is the probability distribution of the immediate reward received after the state transition from s to s' triggered by action pair a. In our computation, we will use the expected immediate reward $R(s, a, s') = E[RD(s, a, s')]$ instead of the distribution RD. *In Collaborative Hunters, a non-zero reward is given with certainty when the wolf is killed in that move.*

- $\gamma \in [0, 1]$ is the discount factor.

Note that if a different model for human behavior is furnished, the formulation could be adjusted by merging the human behavior into the planning environment's dynamics; we would have a classic single-actor MDP formulation with a smaller action space.

The objective is to obtain a policy π that maximizes the expected cumulative reward $\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t), s_{t+1})$ where $0 < \gamma < 1$ is the discount factor.

4.3.2 Human Subgoal Behavior

This model is used at run time for intention tracking; we assume an optimal human at subgoal level but make it more realistic by using soft-max to model the action probability. Specifically, we assume

$$P(a|g_i, s) = \frac{e^{max_{a'} Q_i^*(s,a,a')/\tau}}{Z} \tag{4.1}$$

where Z is the normalizing constant and $g_i \in G$, the set of subgoals; Q_i^* is the Q-function obtained after Subgoal MDP for goal g_i is solved, and $\tau > 0$ a temperature parameter. As such, the human player is assumed to (noisily) know the best response from the assistant and only need to play his part in choosing the action that constitutes the most valued action pair.

Note that τ dictates how noisy the human player's behavior model is; high values of τ translate to a completely random behavior, while low values ($\tau \rightarrow +0$) means a deterministic model. A slightly different view is that τ reflects the assistant's assumption on the expertise level of the lead in solving subgoals g_i. In our implementation, we fix the value of τ such that CAPIR behavior can cater to most average players ($\tau = 10$ in Collaborative Hunters); future work can be to investigate how τ can be dynamically changed according to the player's subgoal expertise.

4.3.3 Goal Change Model

We use a probabilistic state machine to model the human's mind change for intention recognition and tracking. At each time instance, the player is likely to continue on the subgoal that he or she is currently pursuing. However, there is a small probability that the player may decide to switch subgoals. This is illustrated in Figure 4-2, where we model a human player who tends to stick to his chosen sub-goal, choosing to solve the current subgoal 80% of the times and switching to other sub-tasks 20% of the times. The transition probability distributions of the nodes need not be homogeneous, as the human player could be more interested in solving some specific subgoal right after another subgoal. For example, if the wolves need to be captured in a particular order, this constraint can be encoded in the state machine. The model also allows the human to switch back and forth from one subgoal to another during the course of the game.

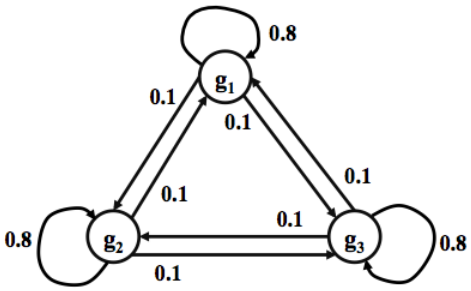


Figure 4-2: A probabilistic state machine, modeling the transitions between subgoals.

In this work, we assume that the goal change model is given to us at the design phase and does not change over time. However this is not strictly required; if the model can be learnt at game time, it is totally valid to use this model for belief update at each time step because the assistant does not need to know this model before game start.

Note that if necessary, there are some ways to compute the model online without having to meticulously hand-craft it at the design phase. The simplest is

to keep the model fixed and state-independent. For example, Figure 4-2 denotes that the lead agent always has 20% chance to deflect his pursuit to other goals, regardless of how close he is to completing his current intended goal. A slightly more complex approach is to keep track of the lead agent's personal preference in pursuing goals and use this information to construct the model in real time. The lead agent's preference can be conditioned on many game-specific factors such as line-of-sight distance (e.g. nearer goals might be preferred) or goal types (e.g. for certain violence-seekers, killing monsters might be preferred over saving civilians). The inter-goal transition probabilities are then learnt while in game as functions of the pertinent factors, before normalized to make the complete model. Since this technique is domain-specific, its investigation is beyond the scope of this thesis.

4.4 CAPIR - Algorithm

Typically, the subgoal MDP is small and could be solved offline. The resultant optimal policies are loaded into memory during game time so that the assistant can select actions with highest utility and reason about the possibly drifting hidden intention of the human player.

4.4.1 Solving Subgoal MDP

An MDP can be effectively solved using Value Iteration [9]. The algorithm maintains a value function $V(s)$, where s is a state, and iteratively updates the value function using the equation

$$V_{t+1}(s) = \max_a \left(\sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V_t(s')) \right).$$

This algorithm is guaranteed to converge to the optimal value function $V^*(s)$, which gives the expected cumulative reward of running the optimal policy from state s.

The optimal value function V^* can be used to construct the optimal action by taking action a^* in state s such that $a^* = \text{argmax}_a \{\sum_{s'} T(s, a, s')V^*(s')\}$. The optimal Q-function is constructed from V^* as follows:

$$Q^*(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^*(s')).$$

The function $Q^*(s, a)$ denotes the maximum expected long-term reward of an action a when executed in state s instead of just telling how valuable a state is, as does V^*. In Subgoal MDP, A is the set of compound actions, so $Q^*(s, a)$ corresponds to $Q^*(s, a_{human}, a_{assist})$ for each $a \in A$.

Intractability

When the Subgoal MDPs yield state spaces of large size, Value Iteration could be prohibitively expensive to run, even at offline phase. In such case, approximate methods that trade accuracy for feasibility constitute a viable choice. These methods will be discussed in details in Chapter 5 and 6.

4.4.2 Belief Update

The belief at time t, denoted $B_t(g_i|\theta_t)$, where θ_t is the game history, is the conditional probability of that the human is performing subgoal i. The belief update operator takes $B_{t-1}(g_i|\theta_{t-1})$ as input and carries out two updating steps.

First, we obtain the next subgoal belief distribution, taking into account the goal change model's transitions $T(g_k \rightarrow g_i)$

$$B_t(g_i|\theta_{t-1}) = \sum_j T(g_j \rightarrow g_i)B_{t-1}(g_j|\theta_{t-1}), \tag{4.2}$$

where $T(g_j \rightarrow g_i)$ is the switching probability from subgoal j to subgoal i.

Next, we compute the posterior belief distribution using Bayesian update, after observing the human action a and subgoal state $s_{i,t}$ at time t, as follows:

$$B_t(g_i|a_t = a, s_t, \theta_{t-1}) = \frac{1}{Z} B_t(g_i|\theta_{t-1}) P(a_t = a|g_i, s_{i,t}) \tag{4.3}$$

where Z is a normalizing constant. Absorbing current human action a and current state into θ_{t-1} gives us the game history θ_t at time t.

Complexity

This component is run in real time, and thus its complexity dictates how responsive our AI is. We are going to show that it is $O(|G|^2)$, with G being the set of subgoals.

The first update step as depicted in Equation 4.2 is executed for all subgoals, thus of complexity $O(|G|^2)$.

The second update step as of Equation 4.3 requires the computation of $P(a_t = a|g_i, s_i)$ (Equation 4.1), which takes time $O(|A|)$ with A being the set of action pairs. Since Equation 4.3 is applied for all subgoals, that sums up to $O(|G||A|)$ for this second step.

In total, the complexity of our real-time Intention Recognition component is $O(|G|^2 + |G||A|)$, which will be dominated by the first term $O(|G|^2)$ if the action set is smaller than the set of subgoals.

4.4.3 Decision-theoretic Action Selection

Given a belief distribution on the human players targeted subgoals as well as knowledge to act collaboratively optimally in each of the subgoals, the assistant

chooses the action that maximizes its expected reward.

$$a^*_{assist} = \text{argmax}_{a'} \left\{ \sum_i B_t(g_i|\theta_t) Q^*_i(s_t, a_{human}, a') \right\}$$

in which a_{human} is the observed human action and a' ranges over all assistant actions.

4.5 CAPIR - System Implementation

CAPIR was implemented as a toolkit for designing two-player collaborative games. Each game level is a GameWorld object that holds references to two Players and multiple SubWorld objects. State update is carried out in each individual class and collated at GameWorld (Figure 4-3). The engine is written in C++, so for the time being games using the engine will need to have their AI module implemented in C++ as well. Allowing AI description via scripts, i.e., Lua or Python, will be future work.

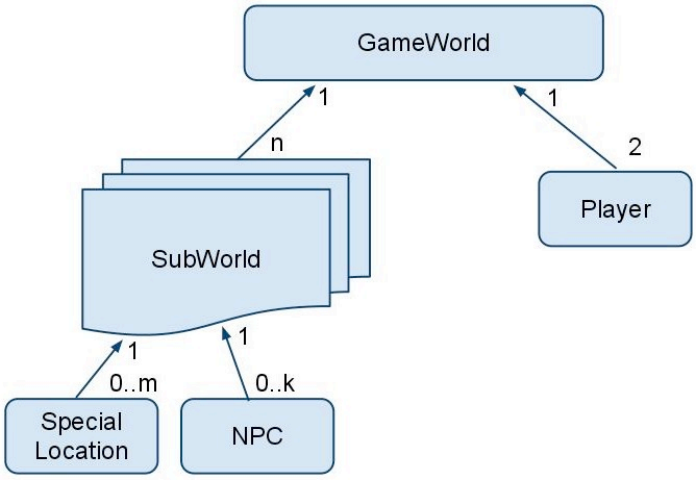


Figure 4-3: GameWorld's components.

4.5.1 Architecture

GameWorld

This is the base class for a game level. In GameWorld, there are two Player objects, at least one of which is controlled by a human player, and multiple SubWorld objects that hold subsets of the NPCs. The game objective is typically to interact with these NPCs in such a way that gives the players the most points in the shortest given time. The players are given points in major events such as successfully killing a monster-type NPC or saving a civilian-type NPC. Each SubWorld implements the scope of a subgoal and is responsible for updating the dynamics of the NPCs in the respective subgoal. A SubWorld reaches its terminal in the current state if the state attributes pertinent to its NPCs have certain values, e.g., all NPCs' hit points are zero.

At each game clock tick, GameWorld updates a game state vector by applying the human player's action on the current state first. The assistant's action planning module is then run to produce an assistant action, taking into account the observed human action. Finally, after the players' actions are applied, some SubWorlds could have ended; the remaining ones will update the game state with their respective NPCs' dynamics, e.g., wolves run away from nearby players. The game episode ends when all SubWorlds are in terminal states.

SubWorld, NPC and SpecialLocation

Each SubWorld holds references to a number of NPCs, which are subclasses an NPC base class. An NPC can be declared as having

- one or many movement actions,
- none or many special actions which affect anything other than its location, and

- none or many properties such as Hit Point, Power and Inventory.

The default behavior of a NPC is to move away from the players upon sighting and acting randomly otherwise. However, each type of NPCs can be extended to have its own kind of behavior.

Besides NPCs, SubWorld could be responsible for updating the state of some immobile in-game items such as doors, guns or shovels, encoded by SpecialLocation classes. These items could be interacted by the NPCs in this SubWorld or the players and could be shared among different SubWorlds.

Player

This class describes essential attributes and abilities the players possess. Each of the two players could have different sets of abilities and characteristics. In general, Players

- are capable of making one or many movement actions,
- are capable of making none or many special actions, and
- have none or many properties.

Special actions are non-movement action such as attacking or casting a spell and are only available depending on the current state.

4.5.2 State Space Reduction with Location Abstraction

In the current implementation of CAPIR engine, we use an abstraction scheme to further reduce the size of Subgoal MDPs' state space.

We observed that regardless of how the NPCs behave, when they are far away from the players, we do not need to know the exact positions of the players, but a rough idea on where they are suffices for planning what actions to take. Only when the players and NPCs are nearby is detailed location information

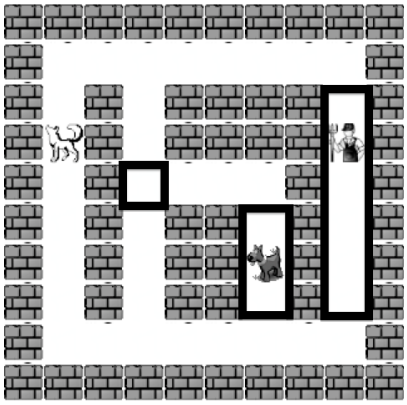


Figure 4-4: Region-based abstraction; some sample regions for the players are shown as dark boxes.

required. For instance, Figure 4-4 depicts a situation in which the farmer needs to continually move right while the dog could stand by and wait regardless of where they are in the occupied corridors. Therefore, we formulate Subgoal MDPs with abstract states, in which the protagonists' regions are stored instead of exact positions. Each region is defined as contiguous sets of grid squares, in which a player has the same set of valid actions regardless of its location in the region. For instance, in Figure 4-4, when in the current region, Farmer can only move left, right or stay still. When the players are near to the NPCs, relative positional information is added.

This abstraction scheme helps reduce the state space of each subgoal from $O(m^{2+n})$ to as low as $O(R^{2+n})$[1], in which m is the total number of unobstructed grid squares in the map, n is the number of NPCs in the subgoal and R is the number of regions.

4.5.3 Workflow

At the planning stage, for each SubWorld, an MDP is generated and a collaboratively optimal action policy is computed accordingly (Figure 4-5). These

[1]The actual size could be larger because abstract states may need to store relative position, i.e., left, right, top and bottom, when the characters are nearby, together with non-position attributes.

policies, represented as subgoals' Q-functions, are used by the AI assistant at run time to determine the most appropriate action to carry out.

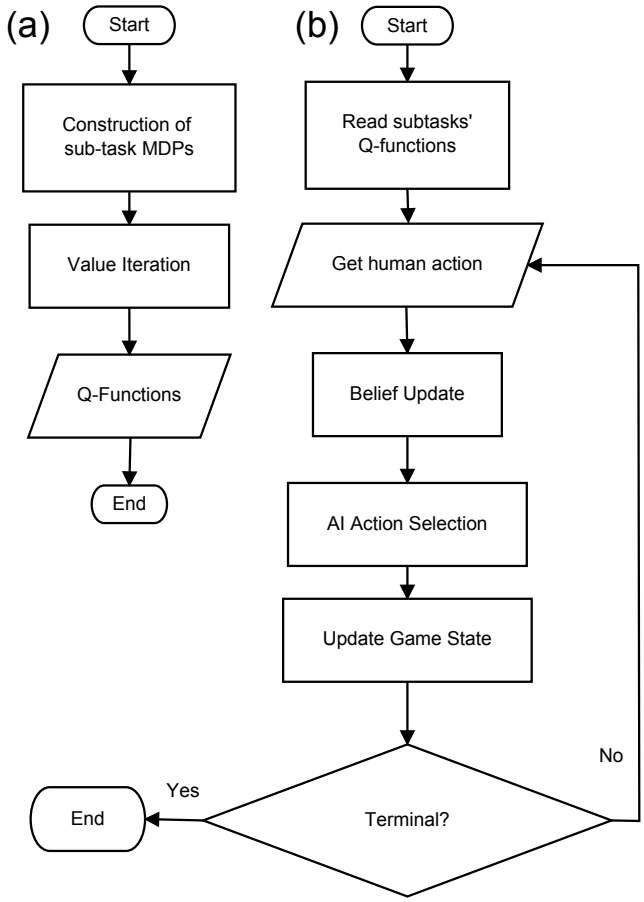


Figure 4-5: CAPIR's action planning process. (a) Offline subgoal Planning, (b) in-game action selection using Intention Recognition.

During game time, upon observing the human player's action in the current state, the assistant updates its belief on the human's hidden intended subgoal according to the process described in Section 4.4.2. It then uses the updated belief to select the action with highest expected Q-value described in Section 4.4.3. After the game state is updated by GameWorld, the game clock ticks to the next moment and the cycle goes on until the game is over or time is up.

4.6 Experiments and Analysis

In these experiments, we want to assess how CAPIR's performance varies as the number of subgoals increases; the tests are done with respect to three computational models of lead behavior. Next, we conduct a human subject study to investigate how CAPIR fares in direct comparison with an expert human partner.

4.6.1 Experiment platform

In this set of experiments, the player's aim is to kill three wolves in a maze, with the help of the assistant dog.

The wolves stochastically run away from any protagonists if they are near the players. Specifically, if a wolf is within K steps away from either or both player, it respectively moves away from either or both of them with probability p and performs some random action with probability $1 - p$. At each game step, the protagonists could move to an adjacent free grid square or stay at the same place; additionally the human player could shoot any wolf if one is found within $K - 1$ steps away, in which case the wolf's hit point (HP) is reduced by one unit. This condition forces the players to collaborate in order to score well. Each wolf has two HP units and once they are depleted due to the farmer's attacks, the wolf is killed, yielding a reward of 5 points. This score is discounted with factor 0.99 so that quicker kills are preferred. The game ends when all wolves are killed or when a limit of 300 game steps is reached.

In the current game instance, we set $K = 3$ and $p = 0.9$ so that the game, though an experiment platform, is still fun to play. If K is too high, the player can attack wolves from far away, so he does not need to move around too much to pursue the wolves; the action element of the game becomes minuscule. Conversely, if K is set too low, the pursuit of wolves can be tedious because the player needs to get real close to the wolves before any damage can be inflicted.

The perseverance probability p should be sufficiently close to 1 to ensure evasion behavior of the wolves. However, with $p \simeq 1$, the game becomes too predictable, reducing the surprise factor that partly contributes to the fun of the game.

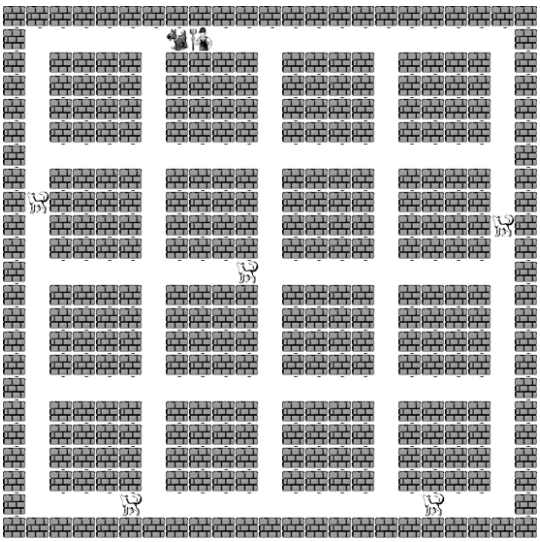


Figure 4-6: Map layout for scalability test; this is a sample starting state of level 3 with five wolves.

4.6.2 Numerical experiments with behavior scripts

In this first set of benchmark experiments, we want to assess how CAPIR performs with increasing number of subgoals as a scalability test. CAPIR is compared with two agents, i.e., Random and Approximately Globally Optimal, when tasked to control the Dog.

- The *random* Dog uniformly selects an action in the action set {*no_move*, N, E, S, W} to execute at each time step.

- The *approximately globally optimal* Dog assumes that the human player is globally optimal, viewing the game as a single large MDP in which the planner controls both the Dog and the Farmer (i.e., the action set is the cross product of the lead and assistant's actions). The task is then

solved approximately; at each time step, the assistant action is extracted from the computed policy's output to execute. Since the formulated task is high-dimensional, we use an approximate algorithm called UCT [44] to estimate the globally optimal policy[2]. This Dog model is henceforth referred to as the UCT Dog.

Note that we could not come up with a suitable Dog model to sensibly assist the human lead in this domain using traditional game design techniques such as HFSMs or scripting [3]. The attempts usually resulted in sub-standard behavior, level-dependent, and especially laborious. We therefore could not conduct meaningful comparison tests of our approach with Dog models constructed using these techniques.

We couple the above assistants with three human models, two heuristic scripts based on subgoals' optimal policies and one globally near-optimal model.

1. *NearestGhost* human picks the nearest alive wolf to pursue and persistently chases that wolf until it is killed.

2. *RandomGhost.0.9* human uniformly picks an alive wolf and maintains it as the main target with probability 0.9 at every step; this model implements the human's mind change.

3. *UCT* human is similar to UCT Dog but instead of one second of planning per step, it runs UCT for 2000 playouts per step. On our machine powered by Intel Xeon CPU 2.83GHz, depending on the map size and number of NPCs in the map, it can take up to 30 seconds to finish one step of plan-

[2]UCT is a state-of-the-art algorithm to approximate an optimal policy in an online fashion; more details on UCT will be furnished in Chapter 5. In this experiment setup, the approximately globally optimal Dog's behavior is constructed by running UCT algorithm one second per game step to compute the estimated optimal action pairs.

[3]As far as we know, commercial games with strong emphasis on cooperative puzzle solving such as *Lara Croft and the Guardian of Light*, which is in the same genre as our game Collaborative Hunters, has yet to deliver computer-aided, collaborative single player mode.

ning for this human model (the program is compiled with optimization but not parallelization).

We examine the collaboration of agent pairs in five map levels of the same size, but with increasing number of wolves, respectively 1, 2, 5, 10 and 20 wolves; the map layout is shown in Figure 4-6. Simulations are run with 100 randomized starting configurations per level, and the average statistics are shown in Figure 4-7.

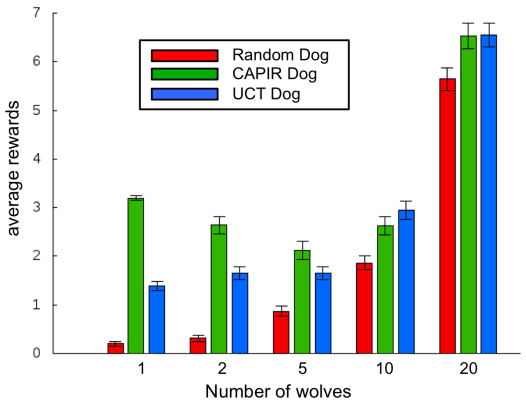
Among three human models, CAPIR cooperates very well with all of them when the number of wolves or subgoals is relatively small (up to 5). When the number of wolves is larger, i.e., 10 and 20, CAPIR is better than the baseline Random agent but is slightly outperformed by the UCT agent. Note that to deliver the charted performance, the computation time per move taken by both Random and CAPIR agents is negligible in all levels, i.e., $< 1ms$ while UCT requires one second of planning for every move.

Clearly, when the number of subgoals is small, the subgoal density is sparse, thus the human's behavior is more pronounced and it is easier to track which subgoal he is pursuing. Conversely, when the number of subgoals is large, the same action sequence performed by the human player has a high chance to be optimal for multiple subgoals at the same time. In this situation the assistant is confused and its performance drops until the subgoals either are better separated or reduce in number.

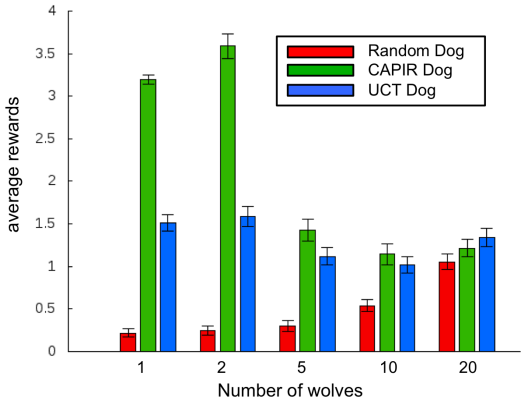
In a nutshell, CAPIR performs well in scenarios in which the subgoals are well separated and the human behaviors among subgoals do not significantly overlap.

4.6.3 Human subject study

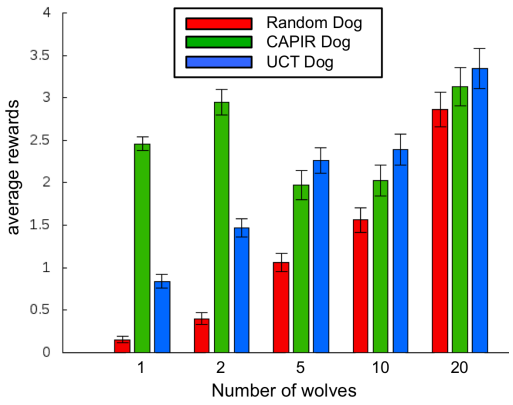
In order to evaluate the performance of our AI system, we conducted a human experiment using Collaborative Hunters. We chose five game levels (Ap-



(a) NearestGhost Human



(b) RandomGhost.0.9 Human



(c) UCT Human

Figure 4-7: Average scores of Random, CAPIR and UCT dog, with standard error of the mean as error bars, when coupled with three human behavior models.

pendix D) with roughly increasing state space size and game play complexity to assess how the technique can scale with respect to these dimensions. The twenty participants are all graduate students at the School of Computing, NUS, seven of whom rarely play games, ten once to twice a week, and three more often.

The participants were requested to play five levels of the game as Farmer

twice, each time with a helping Dog controlled by either CAPIR or a member of our team, the human expert in playing the game. The identity of the dog's controller was randomized and hidden from the participants. After each level, the participants were asked to compare the assistant's performance between two trials in terms of usefulness, without knowing who controlled the assistant at which turn. When we match the answers back to respective controllers, the comparison results take on one of three possible values, being the AI assistant performing "better", "worse" or "indistinguishably" to the human counterpart. The AI assistant is given a score of 1 for a "better", 0 for an "indistinguishable" and -1 for a "worse" evaluation.

Qualitative evaluation

For simpler levels 1, 2 and 3, our AI was rated to be better or equally good more than 50% the times. For level 4, our AI rarely got the rating of being indistinguishable, though still managed to get a fairly competitive performance. Subsequently, we realized that in this particular level, the map layout is confusing for the dog to infer the human's intention; there is a trajectory along which the human player's movement could appear to aim at any one of three wolves. As presented in Section 3.3.4, when this happens, observing the human action sequence alone is not enough to infer the hidden subgoal. In that case, the dog's initial subgoal belief plays a crucial role in determining which wolf it thinks the human is targeting. Since the dog's belief is always initialized to a uniform distribution, that causes the confusion. If the human player decides to move on a path that is easier to tell apart the subgoals, the AI dog is able to efficiently assist him, thus getting good ratings instead. In level 5, our AI gets good ratings only for less than one third of the times, but if we count "indistinguishable" ratings as satisfactory, the overall percentage of positive ratings exceeds 50%.

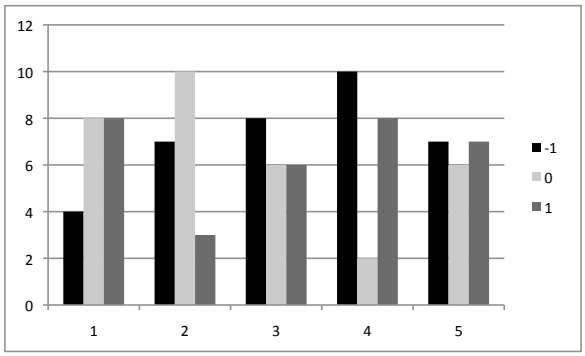


Figure 4-8: Qualitative comparison between CAPIR and human assistant. The y-axis denotes the number of ratings.

Quantitative evaluation

Besides qualitative evaluation, we also recorded the time taken for participants to finish each level (Figure 4-9). Intuitively, a well-cooperative pair of players should be able to complete Collaborative Hunters's levels in shorter time.

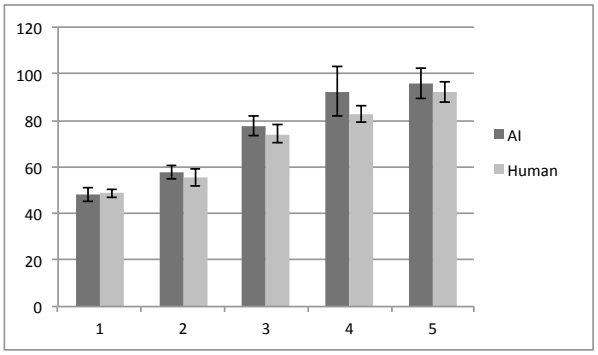


Figure 4-9: Average time, with standard error of the mean as error bars, taken to finish each level when the partner is CAPIR or human. The y-axis denotes the number of game turns.

Similar to our qualitative result, in levels 1, 2 and 3, the AI controlled dog is able to perform at near-human level in terms of game completion time. Level 4, which takes the AI dog and human player more time on average and with higher fluctuation, is known to cause confusion to the AI assistant's inference of the human's intention and it takes a larger number of game turns before the AI realizes the true target, whereas our human expert is quicker in closing down on the intended wolf. Level 5, larger and with more escape points for the wolves

but less ambiguous, takes the protagonist pair (AI, human) only 4.3% more on average completion time.

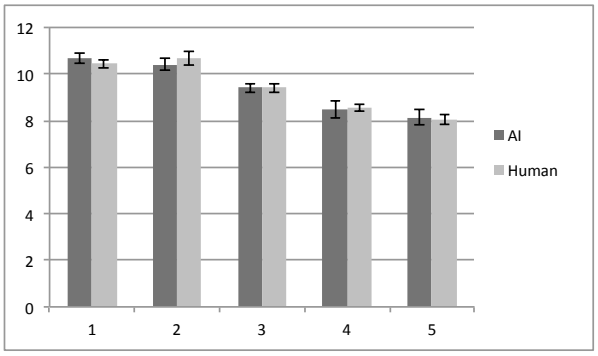


Figure 4-10: Average score, with standard error of the mean as error bars, achieved at the end of each level when the partner is CAPIR or human.

The average scores charted in Figure 4-10 give us more details on the behavior of CAPIR. Note that while CAPIR's finish time is usually longer than that of human assistant, its score is not worse than the counterpart's. Because each wolf kill is discounted, longer game episodes with similar rewards hint that CAPIR assistant pursues single wolves more often than helping the human agent to herd wolves first and kill them all together. This is expected due to the formulation.

4.7 Discussion

CAPIR relies on the solutions of subgoal MDPs for intention tracking; these solutions are assumed to be available while in game. A subgoal MDP usually includes only the most crucial aspects pertinent to the decision making process in that subgoal, hence having manageable complexity. Nonetheless, as the number of factors related to the subgoal increases, the MDP's state space could grow exponentially and overwhelm even state-of-the-art exact solvers. In such cases, approximate methods such as Monte Carlo Tree Search (MCTS) [22] can be used instead. However, while these methods yield running time that is independent of the state space's size, they introduce inaccuracy in estimating the

optimal policy and are resource-demanding. This is an unavoidable limitation of our framework, as we rely on research advancements in tackling large state-space MDPs.

Another limitation of the framework occurs when dealing with too many subgoals, as exposed in the first set of experiments. When the same actions of the lead agent are simultaneously optimal in multiple subgoals, the assistant can be confused about the identity of the true target. In fact, it is not trivial to correctly obtain such knowledge in this case, because clearly the lead agent could be aiming at any of the candidate targets. Dynamic coalition and segregation of subgoals could be the key to address this limitation, which we will leave as future work.

4.8 Chapter Summary

We have described a general decision theoretic approach for constructing smart assistants in collaborative games. Using the MDP formulation to characterize subgoal behavior and intention recognition to infer the subgoal that the lead agent is targeting, the proposed CAPIR framework empirically exhibits near human-level assistance. Moreover, the framework is also very economical in computational demand, requiring a running time scalable with respect to the number of subgoals,

One challenge for the applicability of our framework in practical domains is when dealing with large state-space subgoal MDPs. The subgoal MDP is meant to model subtasks that are relatively independent from one another; therefore, when the NPCs in the game interact in a more intricate manner, the formulation has to group these NPCs into one large subgoal. For instance, in Sheep Farmer, the wolves have the capability to attack and kill the sheep; if we split the sheep from wolf worlds, the MDPs would be unable to model this interactivity. Conse-

quently, the dog may even force the sheep into deadly configurations in which it tries to group wolves and sheep together, oblivious to the fact that sheep should not wander too close to wolves. In the next chapter, we directly address this issue by applying algorithms that are specifically designed for large-state space MDPs and bootstrapping them with heuristic policies to obtain reasonable behaviors as fast as possible.

Chapter 5

Bootstrapping Monte Carlo Tree Search with an Imperfect Heuristic Policy

An important component of CAPIR framework includes the Subgoal MDPs' optimal action-pair policies. If the MDPs yield state spaces with reasonably small size, classical solutions such as Value Iteration and Policy Iteration could be used to compute the exact solution at design time, i.e., offline. These algorithms are very efficient, requiring only linear time in the size of the state space. However, as the size of the MDP grows, so do these algorithms' running time, rendering them infeasible for large to infinite state-space MDPs. Unfortunately, when applying CAPIR formulation in collaborative games, it is not rare to encounter large state-space MDPs, especially when the NPCs impact each other in an involved manner. For instance, in a game of Sheep Farmer, because a wolf can kill a nearby sheep, thereby effectively ending the game, we need to model subtasks that consist of two players, the sheep and surrounding wolves when these NPCs are close to one another. As such, if there are 100 free grid squares on the map, the state space of a subgoal MDP can consist of about 300 millions

states. As the players have totally 30 compound actions, storing the optimal Q-function of such MDPs alone requires tens of gigabytes of memory at execution time, if using tabular representations. In reality when only limited resources are available, this demand is undesirable.

It is worth noting that the immense resource requirement of exact algorithms, such as Value and Policy Iteration, originates from the fact that they take an exhaustive approach. The optimal action for every state is computed before execution time, just in case such a state will be encountered during the actual deployment of the policy. However, in complex domains such as video games, usually only a fraction of the possible states are ever encountered. With that insight, one alternative in solving MDPs is to adopt a "lazy" approach towards finding optimal actions. This approach does not compute the optimal actions beforehand, but does it "lazily" by finding the optimal actions only for a given current state. The optimal action is then executed to transit to a new state before the process is restarted with the new state at root. Adopting this idea, simulation-based approaches constitute a highly practical class of algorithms which approximate the action values "lazily". These algorithms estimate the Q-values of actions starting from a given current state by collecting reward statistics from a randomly sampled look-ahead tree that covers only a small fraction of the state space around the current state. The branching factor and depth of the tree can be constrained to suit the allocated computing resources, trading precision for computation. Simulation-based algorithms have recently achieved noticeable successes when applied in practical domains [31, 19, 28, 5]. Among the members of this class, Upper Confidence Bound applied in Trees (UCT) [44], a Monte Carlo Tree Search (MCTS) [13] flavor, is behind some noticeable advancements in solving notoriously hard problems such as the board game Go [31, 19], General Game Playing [28] or strategy games [5].

While offering many attractive theoretical properties such as near-optimal

guarantees [42] and polynomial convergence rates [44], the analyzed behavior of these algorithms is in asymptotic terms. Moreover, their empirical evaluation often disregards the time constraint in performance assessment. In practice when there is a constraint on computing resources, e.g., the video game domain in which characters need to react in real time, we might not be able to observe this behavior. In fact, Kearns et al. [42] showed that given access to only a generative model of the MDP, in the worst case, a vanilla simulation-based algorithm must run exponentially long to achieve a certain proximity to the optimal solution. Therefore, when deployed in practical setting, these algorithms must leverage on heuristics to arrive at quality solutions within spatial and temporal constraints [12, 31, 28, 19].

In this chapter, we consider the problem of using an offline-learnt heuristic to improve the approximated value function computed by UCT, thereby reducing the time required to find an approximately optimal action. To this goal, there are two widely adopted approaches in the literature, both of which use the heuristic to bias exploration. While the first method is to initialize the search tree with with heuristic values, i.e., biasing initial exploration [12, 31, 28, 19], the second method is to use the heuristic directly for exploration [31, 19]. As intended, these two methods could greatly influence the search control by guiding exploration into more promising state regions. However, when the heuristic function does not accurately reflect the prospect of the states, it could feed the algorithm with false information, thereby leading the search into regions that should be kept unexplored otherwise. To alleviate this potential problem of a heuristic, we propose a novel enhancement technique, which aims to get fast convergence to optimal values at states where the heuristic policy is optimal, while retaining similar approximation as the original UCT at other states. Our new technique, *Aux*, empirically outperforms the aforementioned methods in two experiment benchmarks, one of which is Sheep Farmer, our target domain.

The rest of the chapter is structured as follows. We first give a brief overview of UCT and its popular enhancements before presenting UCT-Aux, our enhanced UCT version. Next, we describe two experiments for comparing the agents' performance and analyze the results. We also identify the common properties of the heuristics used in two experimental domains and provide some insights on why UCT-Aux works well in those cases. Finally, we conclude by discussing the possible usage of UCT-Aux.

5.1 Upper Confidence Bound Applied to Trees

UCT [44] is an anytime algorithm that approximates the state-action value in real time using Monte Carlo simulations. It was inspired by Sparse Sampling [42], the first near-optimal policy whose runtime does not depend on the size of the state space. The approach is particularly suitable for solving planning problems with very large or possibly infinite state spaces. Algorithm 1 details the steps taken by UCT.

The algorithm searches forward from a given starting state, building up a tree whose nodes alternate between reachable future states and state-action pairs (Figure 5-1). State nodes are called *internal* if their child state-action pairs have been expanded and *leaf* otherwise. Starting with a root state, the algorithm iteratively rolls out simulations from this root node; each time an internal node is encountered, it is regarded as a multi-armed bandit and UCB1 [2] is used to determine the action or *arm* to sample, i.e., the edge to traverse. In particular,

Algorithm 1 UCT(s, t)

1: **Input:** state s, number of rollouts t
2: **Output:** an action
3: **for** t times **do**
4: UCT-Rollout($s, 0$) {Depth 0 at root}
5: **end for**
6: **return** BestAction($s, 0$)

Algorithm 2 UCT-Rollout (s, d)

1: **Input:** state s, depth d
2: **if** Terminal(s) **then**
3: **return** 0
4: **end if**
5: **if** Leaf(s, d) **then**
6: **return** RunEpisode(s, Random)
7: **end if**
8: **if** ~HasChildren(s, d) **then**
9: Children(s, d) $\leftarrow$ {}
10: **for all** $a \in A$ **do**
11: Children(s, d) $\leftarrow$ Children(s, d) $\cup$\{(s, a)\}
12: **end for**
13: **end if**
14: ($s, a, type$) $\leftarrow$ SelectChild(s, d) {Selects the child with highest UCB1}
15: (s', r) $\leftarrow$ SimulateAction(s, a)
16: $q \leftarrow r + \gamma$ UCT-Rollout($s', d + 1$)
17: UpdateValue(s, a, q, d)
18: **return** q

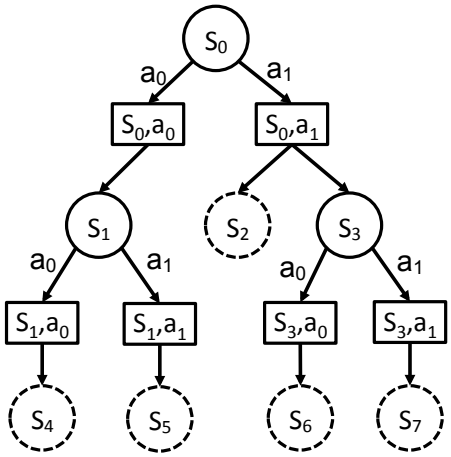


Figure 5-1: A sample UCT search tree with two valid actions a_0 and a_1 at any state. Circles are *state* nodes and rectangles are *state-action* nodes; solid state nodes are *internal* while dotted are *leafs*.

at an internal node s, the algorithm selects an action (Algorithm 2, line 14) according to

$$\pi_{UCT}(s) = \underset{a}{\text{argmax}} \left\{ Q_{UCT}(s,a) + 2C_p \sqrt{\frac{\log n(s)}{n(s,a)}} \right\}, \tag{5.1}$$

in which

- $Q_{UCT}(s, a)$ is the estimated value of state-action pair (s, a), taken to be the weighted average of its children's values.
- $C_p > 0$ is a suitable hand-picked constant.
- $n(s)$ is the total number of rollouts starting from s.
- $n(s, a)$ is the number of rollouts that execute a at s.

At the chosen child state-action node, the simulator is randomly sampled for a next state with accompanying reward (Algorithm 2, line 15); new states automatically become leaf nodes. From the leaf nodes, rollouts are continued using random sampling until a termination condition is satisfied, such as reaching terminal states or simulation length limit. Once finished, the returned reward propagates up the tree (Algorithm 2, line 17), with the value at each parent node being the weighted average of its child nodes' values; suppose the rollout executes action a at state s and accumulates reward $R(s, a)$ in the end.

- at state-action nodes, $n(s, i) = n(s, i) + 1$ and $Q_{UCT}(s, a) = Q_{UCT}(s, a) + \frac{1}{n(s,a)}(R(s, a) - Q_{UCT}(s, a))$
- at state nodes, $n(s) = n(s) + 1$.

Typically one leaf node is converted to internal per rollout, upon which its child state-action nodes are generated. When the algorithm is terminated, the root's arm with highest $Q_{UCT}(s, a)$ is returned[1] (Algorithm 1, line 6).

5.1.1 Enhancement methods

In vanilla UCT, new state-action nodes are initialized with uninformed default values and random sampling is used to finish the rollout. Given a source of prior knowledge, Gelly and Silver [31] proposed two directions to bootstrap UCT:

[1] In practice, returning the arm with highest $n(s, a)$ is also a common choice.

1. Initialize new action-state nodes with $n(s,a) = n_{prior}(s,a)$ and $Q_{UCT}(s,a) = Q_{prior}(s,a)$, and

2. Replace random sampling by better-informed exploration guided by π_{prior}.

We refer to these two algorithms as UCT-I (UCT with new nodes **i**nitialized to heuristic values) and UCT-S (UCT with **s**imulations guided by π_{prior}); UCT-IS is the combination of both methods. UCT-I and UCT-S can be further tuned using domain knowledge to mitigate the flaw of a bad heuristic and amplify the influence of a good one by adjusting the dependence of the search control on the heuristic at internal nodes. In this work, we do not investigate the effect of such tuning to ensure a fair comparison between techniques when employed as is.

In the same publication [31], the authors proposed another bootstrapping technique, namely Rapid Action Value Estimation (RAVE), which we do not examine in this work. The technique is specifically designed for domains in which an action from a state s has similar effect regardless of when it is executed, either at s or after many moves. RAVE uses the All-Moves-As-First (AMAF) heuristic [14] instead of $Q_{UCT}(s,a)$ in Equation 5.1 to select actions. Many board games such as Go or Breakthrough [28] have this desired property. In our experiment domains, RAVE is not applicable, because the actions are mostly directional movements, e.g., $\{N, E, S, W\}$, thus tied closely to the state they are performed at.

5.2 UCT-Aux: Algorithm

Given a heuristic policy π, we propose a new algorithm UCT-Aux that follows the same search control as UCT except for two differences.

1. At every internal node s, besides $|A(s)|$ normal arms with $A(s)$ being the set of valid actions at state s, an additional arm labeled by the action $\pi(s)$

is created (Figure 5-2).

2. When this arm is selected by Equation 5.1, it stops expanding the branch but rolls out a simulation using π; value update is carried out from the auxiliary arm up to the root as per normal.

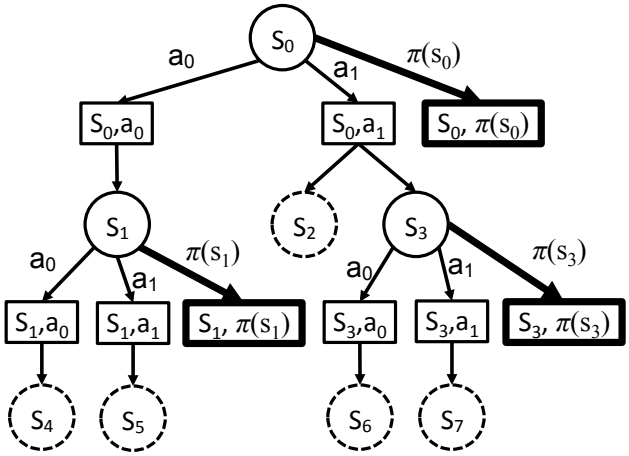


Figure 5-2: Sample search tree of UCT-Aux.

The method aims to better manage mixed-quality heuristics. If the heuristic π's value estimation at a state is good, we expect the added arm to dominate the distribution of rollouts and quickly give a good estimate of the state's value without the need to inspect other arms. Otherwise, the search control will focus rollouts in ordinary arms, thus retaining similar approximation as vanilla UCT. The pseudo code for a UCT-Aux rollout is detailed in Algorithm 3; UCT-Aux is run by invoking the UCT algorithm (Algorithm 1) with the rollout routine at Line 4 being UCT-Aux-Rollout instead of UCT-Rollout.

Convergence Analysis

We will show that regardless of the added policy's quality, UCT-Aux converges in finite-horizon MDPs[2]. The proof follows closely that of UCT analysis by

[2]As mentioned in [44], for use with discounted infinite-horizon MDPs, the search tree can be cut off at the effective ϵ_0-horizon with ϵ_0 being the desired accuracy at root.

Algorithm 3 UCT-Aux-Rollout (s, d, π)

Input: state s, depth d, heuristic policy π
if Terminal(s) **then**
  return 0
end if
if Leaf(s, d) **then**
  return RunEpisode(s, Random)
end if
if ~HasChildren(s, d) **then**
  Children(s, d) $\leftarrow$ {}
  for all $a \in A$ **do**
    Children(s, d) $\leftarrow$ Children(s, d) $\cup$\{($s, a, type = Normal$)\}
  end for
  $\hat{a} \leftarrow \pi(s)$ {Adds an auxiliary arm}
  Children(s, d) $\leftarrow$ Children(s, d) $\cup$\{($s, \hat{a}, type = Aux$)\}
end if
($s, a, type$) $\leftarrow$ SelectChild(s, d) {Selects the child with highest UCB1}
if $type = Aux$ **then**
  $q \leftarrow$ RunEpisode(s, π)
else
  (s', r) $\leftarrow$ SimulateAction(s, a)
  $q \leftarrow r + \gamma$ UCT-Aux-Rollout($s', d + 1$)
end if
UpdateValue(s, a, q, d)
return q

treating the auxiliary arms as any other ordinary arms. As a recap, UCT convergence analysis revolves around the analysis of non-stationary multi-armed bandits with reward sequences satisfying some drift conditions, which is proven to be the case for UCT's internal nodes with appropriate choice of bias sequence C_p[3]. In particular, the drift conditions imposed on the payoff sequences go as follows:

- The expected values of the averages $\overline{X}_{in} = \frac{1}{n}\sum_{t=1}^{n} X_{it}$ must converge for all arms i with n being the number of pulls and X_{it} the payoff of pull t. Let $\mu_{in} = E[\overline{X}_{in}]$ and $\mu_i = lim_{n\to\infty}\mu_{in}$.

- $C_p > 0$ can be chosen such that the tail inequalities $P(\overline{X}_{i,n(i)} \geq \mu_i + c_{t,n(i)}) \leq$

[3]Empirically, C_p is often chosen to be an upper bound of the accumulated reward starting from the current state.

t^{-4} and $P(\overline{X}_{i,n(i)} \leq \mu_i - c_{t,n(i)}) \leq t^{-4}$ are satisfied for $c_{t,n(i)} = 2C_p\sqrt{\frac{\ln t}{n(i)}}$ with $n(i)$ being the number of times arm i is pulled up to time t.

We will first prove the statement that all internal nodes of UCT-Aux have arms yielding rewards satisfying the drift conditions.

Suppose the horizon of the MDP is D, the number of actions per state is K and the heuristic policy is deterministic ($\kappa = 1$); this can be proven using induction on D. Note that i.i.d. payoff sequences satisfy the drift conditions trivially due to Hoeffding's inequality.

- $D = 1$: Suppose the root has already been expanded, i.e., become internal. It has $K+1$ arms, which either lead to leaf nodes (ordinary arms) or return i.i.d. payoffs (auxiliary arm). Since leaf nodes have i.i.d. payoffs, all arms including the auxiliary satisfy drift conditions. Therefore, the statement is true for $D = 1$.

- $D > 1$: Assume the statement is true for all horizon up to $D - 1$. This means all internal state nodes under the root have arms satisfying the drift conditions, e.g., s_1 and s_3 in Figure 5-2. Consider any ordinary arm of the root node (the added arm's payoff sequence is already i.i.d.), for instance, (s_0, a_1). Its payoff average is the weighted sum of payoff sequences in all leafs and state-action nodes on the next two levels of the subtree, i.e., leaf s_2, arms (s_3, a_0), (s_3, a_1) and $(s_3, \pi(s_3))$, all of which satisfy drift conditions due to either the inductive hypothesis or producing i.i.d. payoffs. Theorem 4 by Kocsis and Szepesvari [44] posits that the weighted sum of payoff sequences conforming to drift conditions also satisfies drift conditions; therefore, all arms originating from the root node satisfy drift conditions.

As a result, the theorems on non-stationary bandits by Kocsis and Szepesvari [44] hold for UCT-Aux's internal nodes as well. Therefore, we can obtain similar results to Theorem 6 as shown by Kocsis and Szepesvari [44], with the

difference being statistical measures related to the auxiliary arms such as μ_{aux} and Δ_{aux}. As such, the new algorithm's probability of selecting a suboptimal arm converges to zero as the number of rollouts tends to infinity.

5.3 Experiments

We compare the performance of UCT-Aux against UCT, UCT-I, UCT-S and UCT-IS in two domains: Obstructed Sailing and Sheep Farmer. Obstructed Sailing extends the benchmark Sailing domain by placing random blockage in the map; the task is to quickly move a boat from one point to a destination on a map, disturbed by changing wind, while avoiding obstacles. Sheep Farmer features a two-player maze game in which the players need to herd a sheep into its pen while protecting it from being killed by two wolves in the same environment.

5.3.1 Experiment in the Obstructed Sailing domain

The Sailing domain [74], originally used to evaluate the performance of UCT, features a control problem in which the planner is tasked to move a boat from a starting point to a destination under disturbing wind conditions as quickly as possible. In our version, there are several obstacles placed randomly in the map (see Figure 5-3).

In this domain, the state is characterized by tuple $\langle x, y, b, w_{prev}, w_{curr} \rangle$ with (x, y) being the current boat position, b the current boat posture or direction, w_{prev} the previous wind direction and w_{curr} the current wind direction. Directions take values in {*N, NE, E, SE, S, SW, W, NW*}, i.e. clockwise starting from North. The controller's valid action set includes all but the directions against w_{curr}, out of the map or into an obstacle. After each time step, the wind has roughly equal probability to remain unchanged, switch to its left or its right, as depicted in

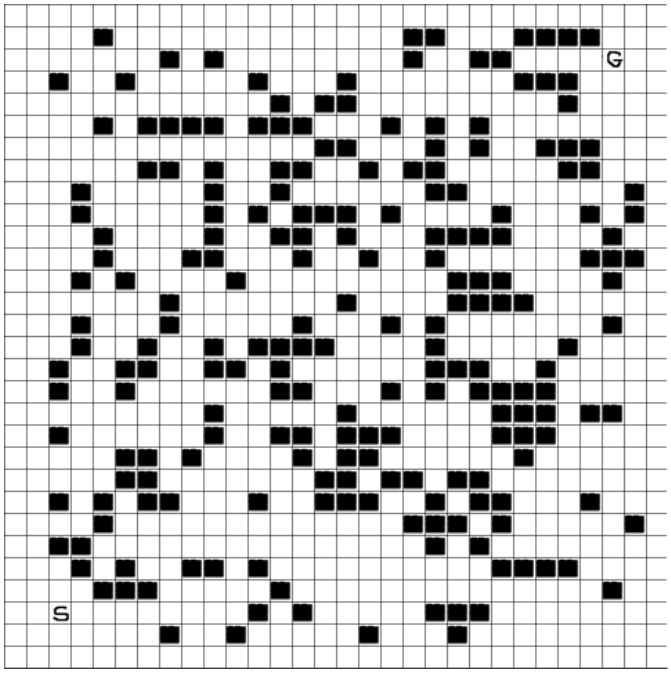


Figure 5-3: Obstructed Sailing sample map with a randomized map configuration.

Table 5.1.

Depending on the relative angle between the action taken and w_{curr}, a cost from one to four minutes is incurred. More specifically, the cost is one if the boat is sailed in the same direction as the wind (O degree angle), two with angle 22.5 , three 45, and four for 77.5. It is prohibited to sail against the wind. Additionally, changing from a port to a starboard tack or vice versa causes a *tack delay* of three minutes. In total, an action can cost anywhere from one to seven minutes, i.e., $C_{min} = 1$ and $C_{max} = 7$. We model the problem as an infinite-horizon discounted MDP with discount factor $\gamma = 0.99$ and horizon $D = 300$ to make sure the search is long enough for value estimation.

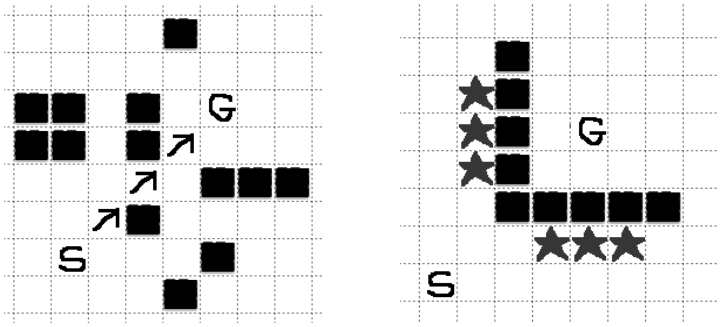
Choice of heuristic policies

A simple heuristic for this domain is to select a valid action that is closest to the direction towards goal position regardless of the cost, thereafter referred to as SailingTowardsGoal. For instance, in the top subfigure of Figure 5-4b, at the

	N	NE	E	SE	S	SW	W	NW
N	0.4	0.3	0	0	0	0	0	0.3
NE	0.4	0.3	0.3	0	0	0	0	0
E	0	0.4	0.3	0.3	0	0	0	0
SE	0	0	0.4	0.3	0.3	0	0	0
S	0	0	0	0.4	0.2	0.4	0	0
SW	0	0	0	0	0.3	0.3	0.4	0
W	0	0	0	0	0	0.3	0.3	0.4
NW	0.4	0	0	0	0	0	0.3	0.3

Table 5.1: Transition probability of wind directions.

starting state marked by "S", if current wind is not SW, SailingTowardsGoal will move the boat in the NE direction; otherwise, it will execute either N or E.



(a) Good case (b) Bad case

Figure 5-4: SailingTowardsGoal produces near-optimal estimates/policies in good cases but misleads the search control in others.

This heuristic is used in UCT-I and UCT-IS by initializing new state-action nodes with the minimum cost incurred when all future wind directions are favorable for desired movement. Specifically,

$$n_{STG}(s,a) \leftarrow 1$$

$$Q_{STG}(s,a) \leftarrow C(s,a) + C_{min}\frac{1-\gamma^{d(s',g)+1}}{1-\gamma}$$

with $C(s,a)$ being the cost of executing action a at state s, $C_{min} = 1$ and $d(s',g)$ the minimum distance between next state s' and goal position g. For UCT-S, the random rollouts are replaced by $\pi(s) = \text{argmax}_a\, Q_{STG}(s,a)$.

Heuristic quality. This heuristic works particularly well for empty spaces, producing near-optimal plans if there are no obstacles. However, it could be counterproductive when encountering obstacles. In the bottom subfigure of Figure 5-4b, if a rollout from the starting position is guided by SailingTowardsGoal, it could be stuck oscillating among the starred tiles, thus giving inaccurate estimation of the optimal cost.

Setup and results

The trial map size is 30 by 30, with fixed starting and goal positions at $(2, 2)$ and $(27, 27)$ respectively (Figure 5-3). We generated 100 random instances of the map, where obstacles are shuffled by giving each grid tile $p = 0.4$ chance to be blocked. Each instance is tried five times, each of which with different starting boat postures and wind directions.

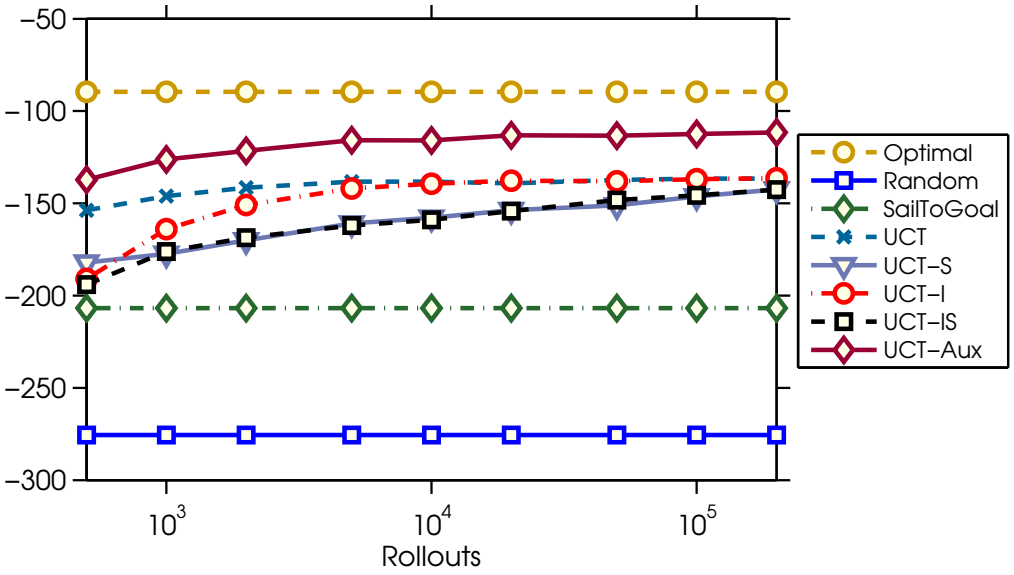


Figure 5-5: Performance comparison of UCT, UCT-S, UCT-I, UCT-IS and UCT-Aux when coupled with the heuristic SailingTowardsGoal; y-axis is the reward average.

All UCT variants (UCT, UCT-I, UCT-S, UCT-IS and UCT-Aux) use the same $C_p = C_{max}/(1-\gamma) = 700$ and the search horizon[4] is set to be 300; an

[4]The search horizon is chosen to be long enough so that the cost accumulated after the

optimal path should not be very far from 60 steps as most actions move the boat closer to the goal. The exact optimal policy is obtained using Value Iteration. Note that the performance of Optimal agent varies because of the randomization of starting states (initial boat and wind direction) and map configurations.

Given the same number of samplings, UCT-Aux outperforms all competing UCT variants, despite the mixed quality of the added policy SailingTowards-Goal when dealing with obstacles (Figure 5-5). Note that without parameter tuning, both UCT-I and UCT-S are inferior to vanilla UCT, but between UCT-I and UCT-S, UCT-I shows faster performance improvement when the number of samplings increases. The reason is because when SailingTowardsGoal produces inaccurate heuristic values, UCT-I only suffers at early stage while UCT-S endures misleading guidance until the search reaches states where the policy yields more accurate heuristic values. The heuristic's impact is stronger in UCT-S than UCT-I: UCT-IS's behavior is closer to UCT-S than UCT-I.

5.3.2 Experiment in the Sheep Farmer domain

The following experiment is conducted in the Sheep Farmer domain that is an extension of the *Collaborative Hunters* game. The game features two players (a farmer and a dog) whose task is to herd a sheep into its pen while protecting it from being killed by two wolves in a maze-like environment. All wolves and sheep run away from the players within a certain distance, otherwise the wolves chase the sheep and the sheep runs away from wolves. Wolves can only be attacked by the Farmer, when they are within three grid squares away from the Farmer (Figure 5-6).

Both protagonists have 5 movement actions (no_move, N, S, E and W) while Farmer has an additional action to inflict damage on a nearby wolf, hence a total of 30 compound actions. The two players are given rewards for successfully

horizon has small effect to the total cost.

killing wolves (5 points) or herding sheep into its pen (10 points). If the sheep is killed, the game is terminated with penalty -10. The discount factor in this domain is set to be 0.99.

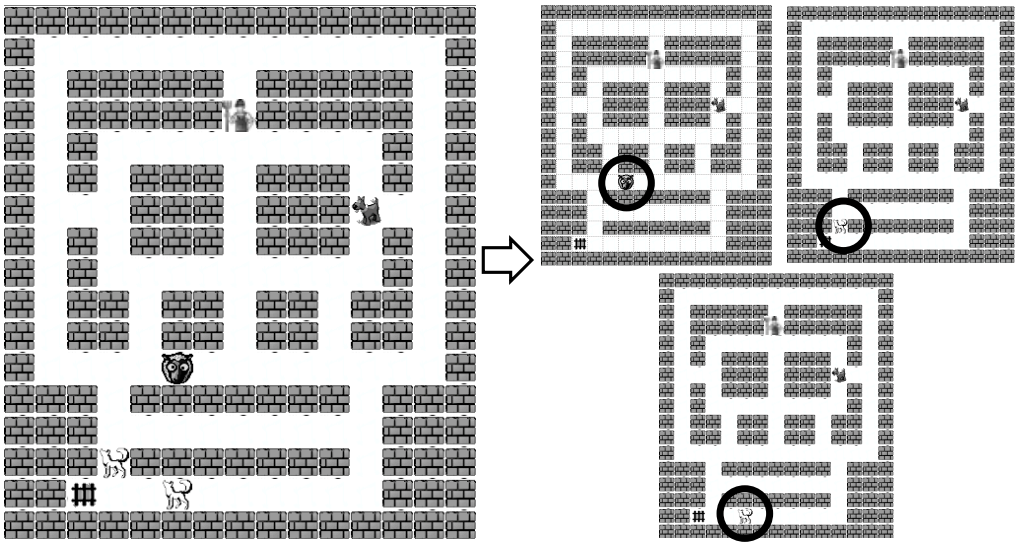


Figure 5-6: Task decomposition in Sheep Farmer.

Choice of heuristic policies

The game can be seen as having three subtasks, each of which is the task of killing a single wolf or herding a single sheep, as shown in Figure 5-6. Each of these subtasks consists of only two players and one NPC, hence has manageable complexity and can be solved exactly offline using Value Iteration.

A heuristic Q-value can be obtained by taking the average of all individual subworlds', or subtasks', Q-values, as an estimate for one state-action pair's value. Specifically, at state s the GoalAveraging (GA) heuristic yields

$$n_{GA}(s,a) \leftarrow 1$$

$$Q_{GA}(s,a) = \frac{1}{m}\sum_{i=1}^{m} Q_i(s_i,a)$$

in which s_i is the projection of s in subtask i, m is the number of subtasks, i.e.

three in this case, and $Q_i(s_i, a)$ are subtasks' Q-values. The corresponding GA policy can be constructed as $\pi_{GA}(s) = \text{argmax}_a Q_{GA}(s, a)$.

Heuristic quality. GA works well in cases when the sheep is well-separated from wolves. However, when these creatures are close to each other, the policy's action estimation is no longer valid and could yield suboptimal results. The under-performance is due to the fact that the heuristic is oblivious to the interactivity between subtasks, in this case, wolf-killing-sheep scenarios.

Setup and results

The map shown in Figure 5-6 is tried 200 times, each of which with a different random starting configurations. We compare the means of discounted rewards produced by the following agents: Random, GoalAveraging, UCT, UCT-I, UCT-S, and UCT-Aux. The optimal policy in this domain is not computed due to the prohibitively large state space, i.e., $104^5 * 3^2 \approx 10^{11}$ since each wolf has at most two health points. All UCT variants have a fixed planning depth of 300. In our setup, one second of planning allows roughly 200 rollouts on average, so we do not run simulations with higher numbers of rollouts than 10000 due to time constraint. Moreover, in this game-related domain, the region of interest is in the vicinity of 200 to 500 rollouts for practical use.

As shown in Figure 5-7, UCT-Aux outperforms the other variants, especially early on with small numbers of rollouts. UCT-S takes advantage of the heuristic GoalAveraging better than UCT-I, which yields even worse performance than vanilla UCT. Observing the improvement rate of UCT-S we expect it to approach UCT-Aux much sooner than others, although asymptotically all of them will converge to the same optimal value when enough sampling is run and the search tree is sufficiently expanded; the time taken could be prohibitively long though.

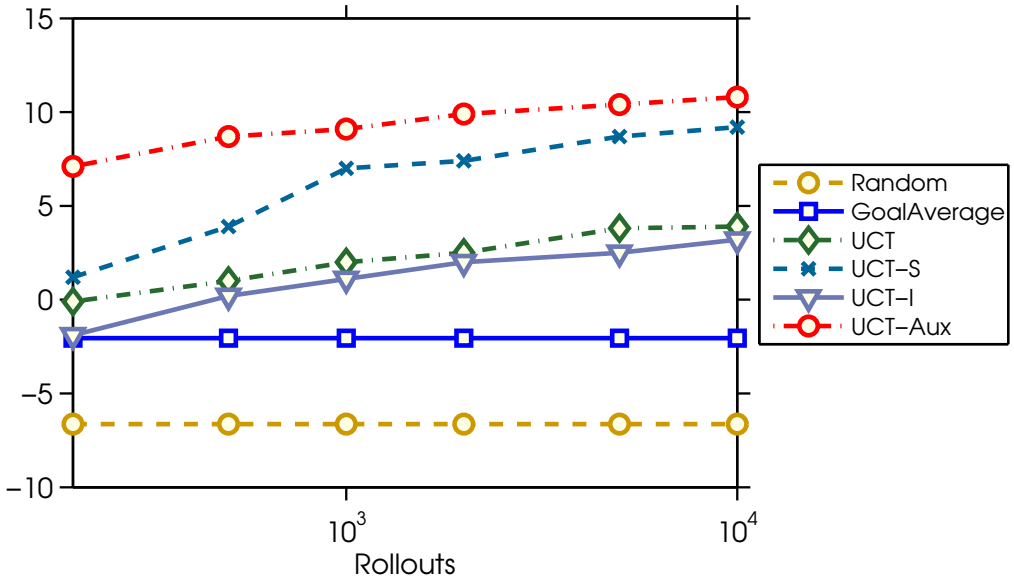


Figure 5-7: Performance comparison of Random, GoalAveraging, UCT, UCT-S, UCT-I, and UCT-Aux when coupled with Goal Averaging.

5.4 Discussion

Although UCT-Aux shows superior performance in the experiments above, we observe that the chosen heuristic policies share a common property that is crucial for UCT-Aux's success: they "*make it or break it*". In other words, at most states s, their action value estimate $Q_\pi(s,a)$ is either near-optimal or as low as that of a random policy $Q_{rand}(s,a)$.

Specifically, in Obstructed Sailing, if following SailingTowardsGoal can bring the boat from a starting state to goal position, e.g., when the line of sight connecting source and destination points lies entirely in free space, the resultant course of actions does not deviate much from the optimal action sequences. However when the policy fails to reach the goal, it could be stuck forever. For instance, Figure 5-4b depicts one such case; once the boat has reached either one of three starred tiles underneath the goal position, unless at least three to five wind directions in a row are E, SailingTowardsGoal results in oscillating the boat among these starred tiles. The resultant cost is therefore very far from optimal and could be as low as the cost incurred by random movement.

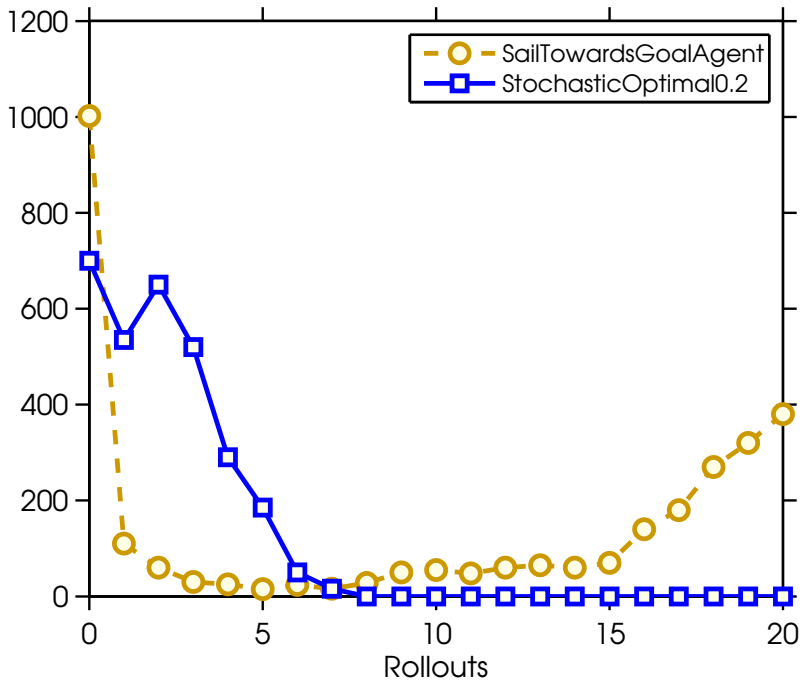


Figure 5-8: Performance histograms of heuristics in Obstructed Sailing. The returned costs of a heuristic are allocated relatively into bins that equally divide the cost difference between Random and Optimal agents; x-axis denotes the bin number and y-axis the frequency.

In contrast, an example of a heuristic that is milder in nature is the policy StochasticOptimal.0.2 which performs optimal actions with probability 0.2 and random actions the rest of the time. This policy is also suboptimal but almost always yields better value estimation than random movement; it is not as "*extreme*" as SailingTowardsGoal. Figure 5-8, which charts the performance histograms of StochasticOptimal.0.2 alongside with SailingTowardsGoal, shows that a majority of runs with SailingTowardsGoal yield costs that are either optimal or worse than that of Random.

Similarly, GoalAveraging in Sheep Farmer is also extreme: By ignoring the danger of Wolves when around Sheep, it is able to quickly herd the Sheep in the Pen or kill nearby Wolves (good), or end the game prematurely by forcing the Sheep into Wolves' zones (bad). We hypothesize that one way to obtain extreme heuristics is by taking near-optimal policies of the relaxed version of the original planning problem, in which aspects of the environment that cause negative

effects to the accumulated reward are removed. For instance, SailingTowards-Goal is in spirit the same as the optimal policy for maps with no obstacle, while GoalAveraging should work well if the wolves do not attack sheep.

As UCT-Aux is coupled with heuristic policies with this "extreme" characteristic, rollouts are centralized at auxiliary arms of states where $\pi(s)$ is near-optimal, and distributed to ordinary arms otherwise. Consequently, the value estimation falls back to the default random sampling where π produces inaccurate estimates instead of relying entirely on π as does UCT-S.

5.4.1 When does UCT-Aux not work?

Figure 5-9 charts the worst-case behavior of UCT-Aux when the coupled heuristic's estimate is mostly better than random sampling but much worse than that of the optimal policy, e.g. the heuristic StochasticOptimal.0.2 in Obstructed Sailing.

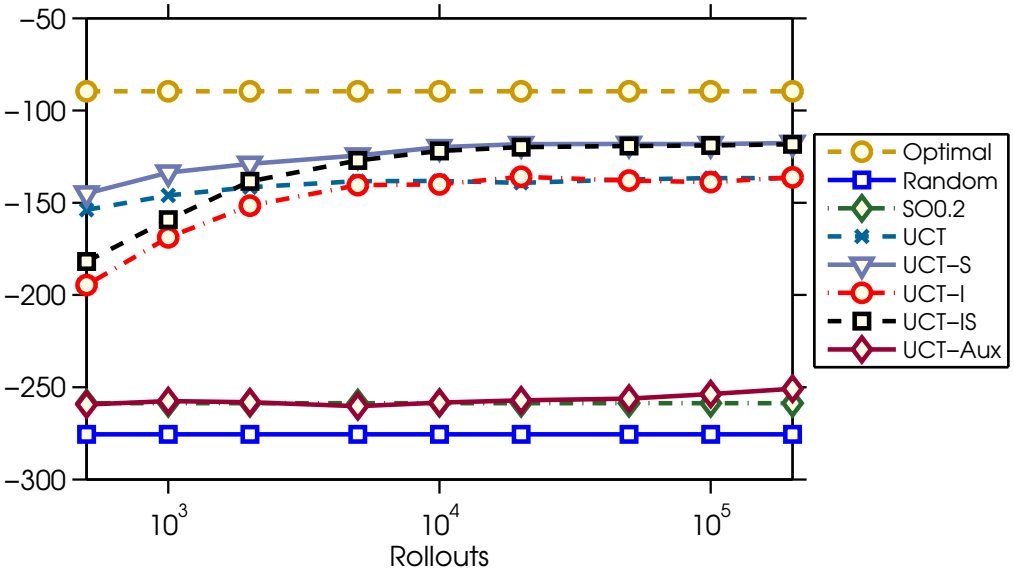


Figure 5-9: Bad case of UCT-Aux when coupled with StochasticOptimal.0.2.

The reason behind UCT-Aux's flop is the same as that of UCT, i.e., due to the "over-optimism" of UCB1, as described in [21]. At each internal node, samples are directed into suboptimal arms that appear to perform the best so far,

exponentially more than the rest (Theorem 1 [44]) when convergence has not started. Even though each arm is guaranteed to be sampled an infinite number of times when the number of samplings goes to infinity (Theorem 3 [44]), the sub-polynomial rate means only a tiny fraction of samplings are spent on attempting bad-looking arms. As a result, in a specially designed binary tree search case, UCT takes at least $\Omega(exp(exp(...exp(2)...)))$ samplings before the optimal node is discovered; the term is a composition of $D - 1$ exponential functions with D being the number of actions in the optimal sequence.

Samplings	500	2000	5000	20000	50000	200000
UCT	268.4	1134.6	2694.5	11041.7	27107.4	107107
UCT-S	268.9	1157.3	2733.2	11212.7	27851.2	109308
UCT-I	279.2	1200.7	2805	11519.4	28123.8	111418
UCT-IS	278.6	1209.2	2834.8	11704.4	28872.5	113682
UCT-Aux	**159.2**	**447.8**	**889.5**	**1981.3**	**3117.9**	**5970.11**

Table 5.2: The average number of tree nodes for UCT variants in Obstructed Sailing when coupled with StochasticOptimal.0.2.

UCT-Aux falls into this situation when coupled with suboptimal policies whose estimates are better than random sampling: At every internal node, it artificially creates an arm that is suboptimal but produces preferable reward sequences when compared to other arms with random sampling. As a result, the auxiliary arms are sampled exponentially more often while not necessarily prescribing a good move. Table 5.2 shows some evidence of this behavior: Given the same number of samplings, UCT-Aux constructs a search tree with significantly less nodes than other variants (up to 20 times). That means many samplings have ended up in non-expansive auxiliary arms because they were preferred.

5.4.2 Combination of UCT-Aux, UCT-I and UCT-S

UCT-Aux bootstraps UCT in an orthogonal manner to UCT-I and UCT-S, thus allowing combination with these common techniques for further performance boost when many heuristics are available. Figure 5-10 charts the performance of such combinations in Obstructed Sailing. UCT-Aux variants use SailingTowardsGoal at the auxiliary arms while UCT-I/S variants use StochasticOptimal.0.2 at the ordinary arms. UCT-Aux-S outperforms both UCT-Aux and UCT-S at earlier stage, and matches the better performer among the two, i.e. UCT-Aux, in a long run.

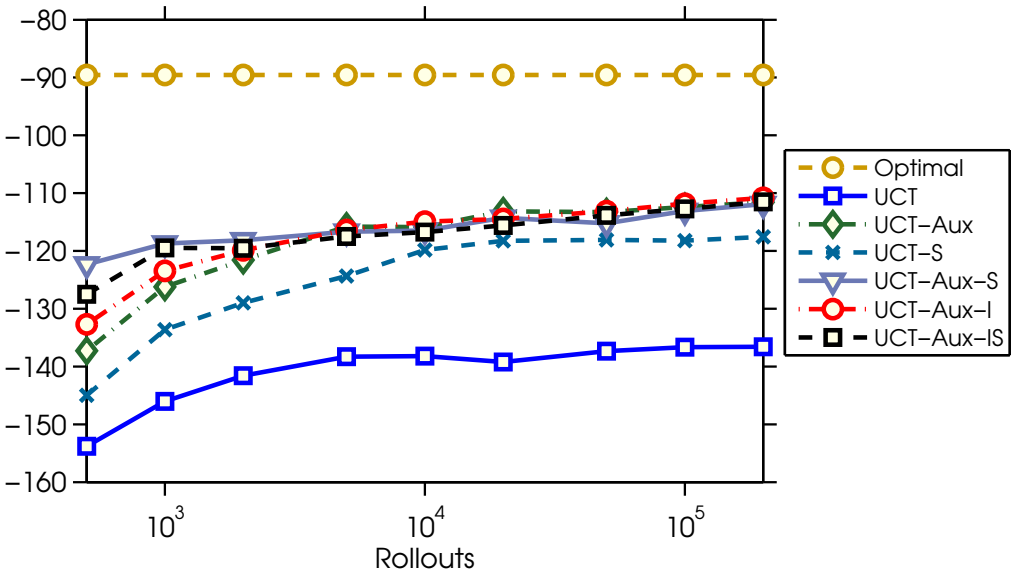


Figure 5-10: Combination of UCT-Aux with UCT-I/S/IS in Obstructed Sailing.

5.5 Chapter Summary

In this chapter, we have introduced a simple technique to bootstrap UCT with an imperfect heuristic policy in solving large-state space subgoal MDPs. It was shown to be able to leverage on the well-performing region while avoiding the bad regions of the policy, empirically outperforming other state-of-the-art bootstrapping methods when coupled with the right kind of policy, i.e, the "extreme"

kind. Our conclusion is that if such property is known beforehand about a certain heuristic, UCT-Aux can be expected to give a real boost over the original UCT, especially in cases with scarce computational resource; otherwise, it would be safer to employ the currently prevalent methods of bootstrapping. As such, a different mentality can be employed when designing heuristics specifically for UCT-Aux: instead of safe heuristics that try to avoid as many flaws as possible, the designer should go for greedier and "riskier" ones. Lastly, since UCT-Aux is orthogonal to other commonly known enhancements, it is a flexible tool that can be combined with others, facilitating more options when incorporating domain knowledge into the vanilla MCTS algorithm.

Even though we have a rough idea about when to use which enhancement method to bootstrap UCT, the worst case analysis of UCT-Aux exposed one weakness that is inherent to UCT: they can be very slow in escaping local optima, e.g., exponentially slow in the case of UCB1. The main reason is that such multi-armed bandit policies usually guarantee to sample highly rewarding arms significantly more often than others. Consequently, the algorithm might distribute more samplings into an overly explored branch than necessary. This drawback does not exist in other simulation-based algorithms that do not use bandit policies for adaptive sampling. In Chapter 6, we are going to investigate the possibility of combining offline-learnt heuristics with such algorithms as Sparse Sampling [42] and Forward Search Sparse Sampling [75].

Chapter 6

Bootstrapping Sparse Sampling and Forward Search Sparse Sampling with a Suboptimal Policy

6.1 Introduction

As presented in the previous chapter, in finding the optimal policies of large state-space subgoal MDPs in CAPIR, approximate methods that "lazily" compute action values online are preferred over exhaustive algorithms, e.g., Value Iteration, that require prohibitively long running time. However, given only a generative model of the MDP, these approximate algorithms may still be too slow to reach adequate estimation. This limitation renders them unsuitable for scenarios where real-time behavior is required, such as video games. In such cases, heuristics can be used so that better estimation can be obtained quicker. As detailed in Chapter 5, the Aux method is an interesting enhancement technique to bootstrap the state-of-the-art simulation-based algorithm UCT with a heuristic policy. Experiment results have shown that in Sheep Farmer, the method outmatches other widely used enhancement techniques in bootstrapping UCT.

Given a heuristic in the form of an imperfect policy π, the method adds an additional arm at every internal node of the search tree. This special arm is labeled by the action suggested by π and once selected, rolls out the rest of the sampling episode using π. When coupling with heuristic policies that are "extreme", UCT-Aux outperforms commonly used techniques such as UCT-S, UCT-I and UCT-IS. However, the same performance is not guaranteed with non-extreme heuristics; in fact, these heuristics cause UCT-Aux to suffer the worst case performance of UCT, which takes super exponential transitory time to discover the optimal course of actions. This cost is unavoidable without significant change to the original action selection policy UCB1, which lies at the heart of UCT.

To avoid the worst case performance of UCT, we will next investigate how the Aux method behaves when applied in bootstrapping two well-known algorithms, namely Sparse Sampling (SS) [42] and Forward Search Sparse Sampling (FSSS) [75]; as far as we know, our work is the first to investigate the problem of bootstrapping these simulation-based algorithms. SS was the first algorithm to produce near-optimal policies while yielding a running time independent of the size of the state space; it can be said that SS is the backbone of many modern simulation-based algorithms. The problem with SS is practicality, since SS could require planning steps that are prohibitively long, i.e. exponential in the horizon time. Its successor FSSS, one of the state-of-the-art sparse sampling algorithms, takes an *anytime* approach, i.e., it can be terminated anytime and yield better approximation if more resources are allocated. FSSS is thus more practical while having a similar performance guarantee to that of SS. Moreover, the algorithm is shown to directly address the worst case scenarios of UCT; the running time of FSSS never exceeds that of SS, i.e., exponential in the horizon time, as opposed to the super-exponential time as required by UCT in the worst case. Intuitively, the Aux method aims at speeding up the value estimation of SS and

FSSS when the heuristic policy is near-optimal, while leaving them unaffected otherwise. We will provide empirical evidence that both SS-Aux and FSSS-Aux outperform their respective vanilla versions by converging faster to near-optimal policies in two benchmark domains, Obstructed Sailing and Sheep Farmer. We will also demonstrate how SS-Aux and FSSS-Aux exhibit better convergence rates than UCT-Aux with FSSS-Aux eventually outperforming UCT-Aux.

6.2 Definitions

In this section, we would like to formally define the *regret* of a policy and describe how we intend to use a heuristic policy to estimate a state's value using rollouts.

6.2.1 Regret of a policy

Given a policy π of a planning problem $M = (S, A, T, R, \gamma)$, we can define its performance guarantee as follows:

Definition 1. *A policy π is called **ϵ-optimal** if its **regret**, $\max_{s \in S} (V^*(s) - V^\pi(s))$, is at most ϵ.*

If the reward function of MDP M is bounded, i.e., there exists R_{max} such that $R(s, a, s') \leq R_{max}, \forall s, a, s'$, we obtain that for any policy π, $|V^*(s) - V^\pi(s)| < \frac{2R_{max}}{1-\gamma} = 2V_{max}, \forall s \in S$. Therefore, when we discuss about an ϵ-optimal policy, it is implicitly understood that $0 \leq \epsilon < 2V_{max}$.

6.2.2 Sampling π to estimate $V^*(s)$

Most simulation-based algorithms aim to approximate the optimal value $V^*(s)$ for the current state s. The Aux enhancement's key idea is to opportunistically select the higher between the value estimation of the core algorithm and that of

the heuristic policy at each tree node; the latter is the main source of estimation improvement on the host algorithm when it yields higher estimation.

We will now study the regret in approximating $V^*(s)$ if we simply simulate a sub-optimal policy π for B times for L steps, and use the average accumulative reward as an estimate. In such an estimate, the total estimation error stems from three sources: (1) the true regret of π, (2) the length L of rollouts, and (3) the limited number B of samples. The following lemma bounds the error in estimating $V^*(s)$ using π guided rollouts.

Lemma 6.1. *Let π be an ϵ_0-optimal policy, $X_{\pi,L,i}$ be the accumulative reward of π-guided rollout i from state s and $\bar{V}^{\pi}_{B,L}(s) = \frac{1}{B}\sum_{i=1}^{B} X_{\pi,L,i}$. For all $\epsilon_1 > 0$, with probability at least $1 - \epsilon_1$,*

$$|V^*(s) - \bar{V}^{\pi}_{B,L}(s)| \leq \epsilon_0 + \gamma^L V_{max} + O\left(V_{max}\sqrt{\frac{\ln(1/\epsilon_1)}{B}}\right).$$

Proof Sketch. By using triangle inequality we get

$$|V^*(s) - \bar{V}^{\pi}_{B,L}(s)| \leq |V^*(s) - V^{\pi}(s)| + |V^{\pi}(s) - V^{\pi}_{L}(s)| + |V^{\pi}_{L}(s) - \bar{V}^{\pi}_{B,L}(s)|$$

with $V^{\pi}_{L}(s) = \mathbb{E}(X_{\pi,L,i})$ for all i. By definition, the first term is bounded by ϵ_0. The second term is the expected estimation error for V^{π} due to simulating π only for L steps; this is bounded by $\gamma^L V_{max}$. Finally, using Hoeffding inequality for B i.i.d. and bounded values $X_{\pi,L,i}$, we bound the last term with probability at least $1 - \epsilon_1$. Collectively, the estimation error of $\bar{V}^{\pi}_{B,L}(s)$ is the sum of three aforementioned terms with probability at least $1 - \epsilon_1$. The detailed proof with numerical derivations can be found in Appendix A. □

By increasing the values of B and L, we can get as close as desired to the $(-\epsilon_0, +\epsilon_0)$ vicinity of a state's value with high probability, at the cost of longer running time that scales with BL.

6.3 Bootstrap Sparse Sampling with Aux

The Sparse Sampling (SS) [42] algorithm estimates the best action to take in a state s_0 based on a set of random simulations of policy "rollouts" or action execution sequences. The approach is motivated by the observation that for discounted reward functions the knowledge about rewards and transition probabilities in a vicinity of the current state suffices for decision making; events far in future are irrelevant due to the discount factor γ. Consequently, a sufficiently extensive simulation of the near future would adequately represent the expected rewards and transition probabilities.

6.3.1 Sparse Sampling (SS)

The SS algorithm (Algorithm 4 called with $h = H$ and aux=False) estimates the state-action values $Q^*(s_0, a)$ of the optimal policy by constructing a look-ahead tree rooted at s_0 at level H (see Figure 6-1). Starting from s_0, it simulates each of the $k = |A|$ possible actions C times in the depth-first postorder down to level 0 where all values $V_0^{SS}(s)$ are set to some default value, e.g., 0. After the subtrees at level $h-1$ have been sampled, the node values are backed up to level h by $Q_h^{SS}(s,a) = R(s,a) + \frac{\gamma}{C}\sum_{c=1}^{C} V_{h-1}^{SS}(s'_c)$, where s'_c denotes a state at level $h-1$ that was obtained when simulating action a in state s the c^{th} time. The V-values of states at each level are based on the same level Q-values by $V_h^{SS}(s) = \text{argmax}_a\, Q_h^{SS}(s,a)$.

As detailed above, the value-estimates of leaf nodes in a SS look-ahead tree were bluntly set to zero. In general, given a leaf value estimation $V_0^{SS}(s)$ that is ϵ_0-optimal, i.e., $\forall s, |V^*(s) - V_0^{SS}(s)| \leq \epsilon_0$, the value estimation error at the root node s_0 is composed of two factors: one due to the limited sampling parametrized by H and C, and one due to the leaf error ϵ_0. The following lemma shows that as H and C increases, with C kept at some magnitude larger than H,

Algorithm 4 SS (s, h, aux)

Input: state s, height h, boolean flag aux
if $h = 0$ or **Terminal**(s) **then**
  $V_0^{SS}(s) \leftarrow 0$
else if $\neg$ Visited(s, h) **then**
  Visited$(s, h) \leftarrow$ true
  for all $a \in A$ **do**
    for C times **do**
      $s' \sim T(s, a)$
      $Q_h^{SS}(s, a) \mathrel{+}= \frac{1}{C} R(s', a) + \frac{\gamma}{C} \mathrm{SS}(s', h-1, aux)$
    end for
  end for
  $V_h^{SS}(s) \leftarrow \max_a Q_h^{SS}(s, a)$
  if $aux = true$ **and** $\bar{V}^{\pi}_{L,B}(s) > V_h^{SS}(s)$ **then**
    $V_h^{SS}(s) \leftarrow \bar{V}^{\pi}_{L,B}(s)$
  end if
end if
return $V_h^{SS}(s)$

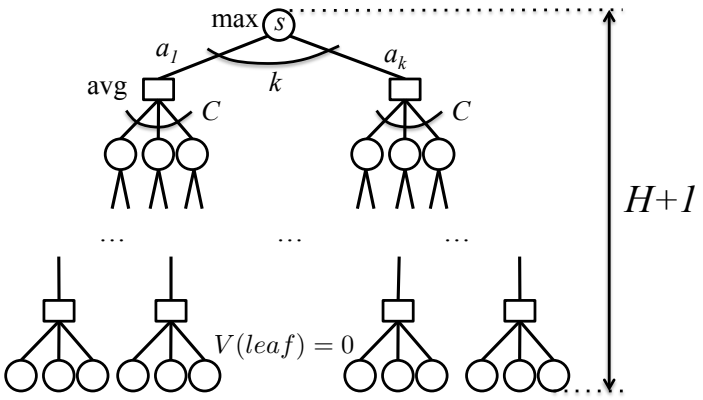


Figure 6-1: The look-ahead tree of Sparse Sampling; each state node (circles) estimates V_h^{SS} while its state-action children (rectangles) estimate Q_h^{SS} for each height h.

and ϵ_0 approaches 0, the algorithm's regret $\epsilon_{SS} = \max_s |V^*(s) - V^{SS}(s)|$ goes to 0.

Lemma 6.2. *Consider SS with leaf nodes' values being ϵ_0-optimal, i.e. $\forall s, |V^*(s) - V_0^{SS}(s)| \leq \epsilon_0$. We have*

$$\epsilon_{SS} \leq T_{SS}(H, C) + \frac{\gamma^H \epsilon_0}{1-\gamma}$$

with $T_{SS}(H,C) = \dfrac{2\gamma^H V_{max}}{1-\gamma} + \dfrac{V_{max}}{(1-\gamma)^2}\sqrt{\dfrac{H}{C}\ln\dfrac{kC}{\gamma}}$.

In the bound, the first error term $T_{SS}(H,C)$ is due to limited sampling, while the second one partly depends on the quality of leaf estimation ϵ_0. If high values of H, C are affordable, the leaf estimation plays little role in the algorithm's quality.

Specifically, Kearns et al. [42] showed that by setting the leaf values to 0, i.e., leaf regret $\epsilon_0 = V_{max}$, and H and C dependent on $\epsilon > 0$ as follows

$$H = \lceil \log_\gamma(\lambda/V_{max}) \rceil, \tag{6.1}$$

$$C = \frac{V_{max}^2}{\lambda^2}\left(2H\ln\frac{kHV_{max}^2}{\lambda^2} + \ln\frac{R_{max}}{\lambda}\right), \tag{6.2}$$

with

$$\lambda = \epsilon(1-\gamma)^2/4, V_{max} = R_{max}/(1-\gamma) \tag{6.3}$$

in which $\epsilon > 0$ is the desired regret, the resultant estimation of the algorithm is ϵ-optimal. It follows that the running time of SS is at most $O((kC)^H)$, i.e.,

$$\left(\frac{k}{\epsilon(1-\gamma)}\right)^{O\left(\frac{1}{1-\gamma}\ln\left(\frac{1}{\epsilon(1-\gamma)}\right)\right)}.$$

With the following theorem, Kearns et al. further showed that this is not very far from the best possible running time given only a simulator [42].

Theorem 6.1. *Let $\mathcal{A}$ be any algorithm that is given access only to a generative model for an MDP M, and inputs s (a state in M) and ϵ. Let the policy implemented by $\mathcal{A}$ satisfy $|V^{\mathcal{A}}(s) - V^*(s)| \leq \epsilon$ simultaneously for all states $s \in S$. Then there exists an MDP M on which $\mathcal{A}$ makes at least $\Omega(2^H) = \Omega((1/\epsilon)^{(1/\ln(1/\gamma))})$ calls to the generative model.*

Effectively the theorem 6.1 posits that Sparse Sampling's running time is reasonably near the shortest possible if given only a generative model of the targeted planning problem. This result suggests that efforts might be better spent on devising algorithms that take advantage of other sources of knowledge besides a simulator.

SS_π: Sparse Sampling with leafs evaluated using heuristic simulations

Given a heuristic policy π, one can propose a method to bootstrap SS by replacing trivial leaf nodes' values (zero) with π-estimates as detailed in Section 6.2.2, i.e., by using the empirical average of B simulated rollouts of length L (Figure 6-2). This naive bootstrapping technique, namely SS_π, poses two key issues.

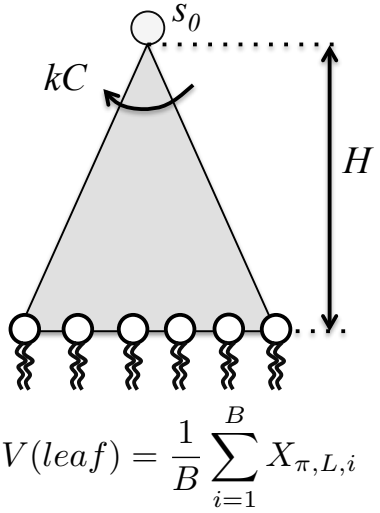


Figure 6-2: The look-ahead tree of SS_π.

Firstly, SS_π requires a significantly longer running time than vanilla SS, specifically incurring time $LB(kC)^H$ in evaluating $(kC)^H$ leaf nodes (LB calls to the simulator at each leaf node). Although the multiplicative factor LB is asymptotically dominated with L and B being fixed constants, in practice with limited resource, the overhead cost could result in a much shorter tree height afforded, negatively impacting the root's estimation.

Secondly, intriguingly, the estimation of SS_π could be worse than that of

vanilla SS. The reason is that while the estimation $V_{leaf}(s) = 0 \forall s$ yields a regret of at most V_{max}, the regret of a bad policy π could be greater V_{max}. For instance, consider the case when π is so bad that it always selects an action with reward $-R_{max}$ at every state, while the optimal action would yield reward R_{max}. Under this policy π, the expected value of any state would be very close to $2V_{max}$. In such case, it is much better to use the trivial value of 0 instead of relying on π-estimation.

As detailed in the following section, the Aux method is able to avoid these issues of SS_π. The resultant algorithm SS-Aux retains the same performance guarantee as SS if the coupled heuristic is overly suboptimal, while having an assured performance boost otherwise. In all cases, SS-Aux yields a running time similar to SS's with the overhead cost being a negligible additive term.

6.3.2 SS-Aux: Sparse Sampling with π-guided Auxiliary Arms

Given a deterministic heuristic policy π, the proposed algorithm, called *SS-Aux* (Algorithm 4 called with aux=True), adds one auxiliary arm into each internal state node of the look-ahead tree[1]. The auxiliary arm stemming from s is labeled by the action $a = \pi(s)$ (Figure 6-3) and it does not expand to subtrees as do its siblings, but its value is computed as the average of accumulated rewards returned by π-guided simulation traces as described in Section 6.2.2. The auxiliary arm is treated equally to its ordinary siblings, i.e., the value of the parent node is computed by taking the maximum of all child nodes' estimated values. Note that the number of auxiliary arms increases exponentially with the height of the tree if these arms are added to every internal node. Therefore, in order to manage the incurred extra cost, we can stop adding auxiliary arms below a certain height $\hat{H}$ with $\hat{H} < H$.

[1]The method can be generalized to stochastic policies by adding κ auxiliary arms to each state node s, with κ being the number of actions a such that $P(\pi(s) = a) > 0$.

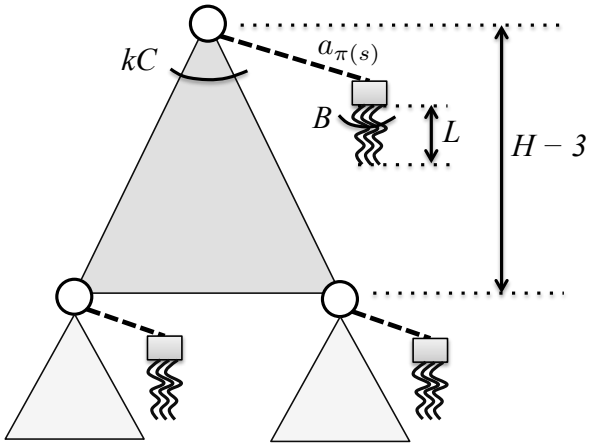


Figure 6-3: The look-ahead tree of SS-Aux.

Unlike SS which completely relies on the value estimation of expanded subtrees, whenever the heuristic π appears to yield higher estimated value than any of its siblings, the parent node's value is estimated by this heuristic value instead. Eventually, this improved estimation propagates and contributes to the root's value. Intuitively, if the heuristic is good at states that are near the root on the optimal path, the improvement is instantiated early in the process. Consequently, SS-Aux can be run with shorter height H, saving computing resources.

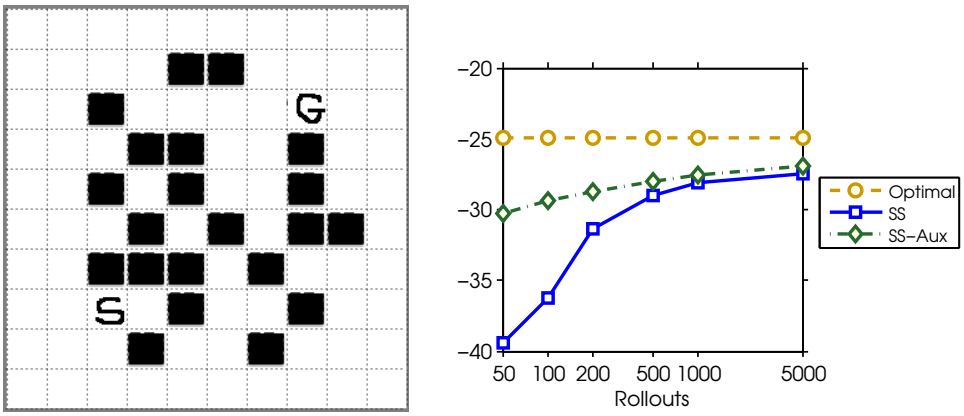
While the required running time of SS is $(kC)^H$, that of SS-Aux is $(kC)^H + LB(kC)^{H-\hat{H}} = (kC)^H(1 + LB/(kC)^{\hat{H}})$. Depending on the resources allocated, $\hat{H}$ can be adjusted so that $LB/(kC)^{\hat{H}}$ is negligible, in which case the running time of SS-Aux is asymptotically equal to that of SS.

Experiments with Obstructed Sailing

In this section we demonstrate empirically how SS-Aux performs as compared to SS. The experiments are run in the Obstructed Sailing domain as described in Section 5.3.1. In order to show the convergence behavior of SS, which is relatively impractical in large domains due to its exhaustive nature, the experiment uses maps of small size, i.e., 10 by 10, with starting position at (2, 2) and the goal position at (7,7). The algorithms are pitted in 1000 different random maps

for which the benchmarked optimal policy is obtained by Value Iteration [8].

Instead of setting a fixed value for the tree height H in SS-algorithms, we implement them using Iterative Deepening with respect to H. These online versions of SS are allocated a certain amount of time for each planning step, starting out with an initial small value of H. Upon termination, if there is still time for planning, H is increased and the algorithm is restarted. However, the sampling width factor C is kept fixed. Even though the theoretical bound is established with C larger than H, empirically a value for C roughly reflecting the stochasticity of the domain suffices.



(a) Map 10x10 (b) Average discounted accumulated reward

Figure 6-4: SS variants in Obstructed Sailing with heuristic SailsToGoal. The charted lines denote the average accumulated reward (negative cost).

The coupled π-heuristic is called SailsToGoal (STG). It selects a valid action closest to the direction towards goal position regardless of the cost [51]. This heuristic is particularly good for maps with few obstacles, producing near-optimal plans for many states, but counterproductive with highly obstructed maps, since it is often stuck in dead ends and corners that surround the goal position. On average, STG yields an accumulated cost of 78.22, far from the optimal cost of 24.89. Nonetheless, when STG is used to bootstrap SS, the resultant planner SS-Aux completely dominates vanilla SS (Figure 6-4) and STG. As the planning time increases, the performance gap reduces; after all we expect

both SS and SS-Aux to converge to the optimal value eventually.

6.4 Bootstrap Forward Search Sparse Sampling

SS-Aux suffers from the impracticality of SS due to unfocused sampling which allocates a fixed amount of computation for every tree branch and node. As a result, SS requires a long running time before delivering any meaningful approximation. The problem is more pronounced in domains that are highly stochastic and where discount factor is close to 1.

There have been many attempts such as Heuristic Sampling [60], Adaptive Sampling [17] and Upper Confidence Bound applied in Trees (UCT) [44] to improve the practicality of the forward sampling approach. All the forward sampling methods use a similar tree-like structure as SS, so there is potential to enhance them using the Aux method. The latest in this series is the Forward Search Sparse Sampling (FSSS) [75] that tries to smartly allocate the computation to promising branches of the look-ahead tree in order to more quickly approximate the optimal policy in large domains. In this section, we investigate the application of the Aux method to enhance FSSS.

6.4.1 Forward Search Sparse Sampling (FSSS)

FSSS is the latest successor of SS that is able to combine the performance guarantee of SS with the selective sampling [75]. Instead of directly forming point estimates of the Q- and V-values in a look-ahead tree, it maintains interval estimates that allow it then to guide the sampling to the promising and/or unexplored branches of the look-ahead tree based on the upper bounds and widths of the intervals. The algorithm is detailed in Algorithms 5 and 6 with the boolean flag *aux* set to false.

Similar to SS, FSSS also constructs a tree of height H and branching factor

Algorithm 5 FSSS-Aux(s, t, aux)

Input: state s, number of rollouts t, boolean aux
Output: estimated lower bound for state s
for t times **do**
  $\hat{L} \leftarrow$ FSSS-Aux-Rollout(s, H, aux) {H is the maximum height}
end for
return $\hat{L}$

Algorithm 6 FSSS-Aux-Rollout (s, h, aux)

Input: state s, height h, boolean aux
if $h = 0$ or **Terminal**(s) **then**
  $L^h(s), U^h(s) \leftarrow 0, 0$
  return 0
end if
if ¬Visited(s, h) **then**
  Visited(s, h) $\leftarrow$ true
  for all $a \in A$ **do**
    $L^h(s,a), U^h(s,a) \leftarrow V_{min}, V_{max}$
    $Children_h(s,a) \leftarrow \{\}$
    $Count_h(s,a,*) \leftarrow 0$ {Count stores the number of times a state s' is encountered at depth h when action a is executed at state s }
    for C times **do**
      $s' \sim T(s,a)$
      $Count_h(s,a,s') \leftarrow Count_h(s,a,s') + 1$
      $Children_h(s,a) \leftarrow Children_h(s,a) \cup \{s'\}$
      $L^{h-1}(s'), U^{h-1}(s') \leftarrow V_{min}, V_{max}$
    end for
  end for
  if $aux = true$ **then** {Adds auxiliary arm}
    $\hat{a} \leftarrow \pi(s)$
    $L^h(s,\hat{a}), U^h(s,\hat{a}) \leftarrow$ EstimateQ($s, \hat{a}, \pi$)
  end if
end if
$a^* \leftarrow \text{argmax}_a\, U^h(s,a)$
$s^* \leftarrow \text{argmax}_{s'}\, Count_h(s,a^*,s')(U^{h-1}(s') - L^{h-1}(s'))$
FSSS-Aux-Rollout($s^*, h-1, aux$)
$L^h(s,a^*) \leftarrow R(s,a^*) + \frac{\gamma}{C}\sum_{s'} Count_h(s,a^*,s')L^{h-1}(s')$
$U^h(s,a^*) \leftarrow R(s,a^*) + \frac{\gamma}{C}\sum_{s'} Count_h(s,a^*,s')U^{h-1}(s')$
$L^h(s) \leftarrow \max_a L^h(s,a)$
$U^h(s) \leftarrow \max_a U^h(s,a)$
return $L^h(s)$

k. However, instead of constructing the whole tree in one go, FSSS traverses the tree from root down using simulated episodes of length H called *trials*. At each

state node s of the trial, the algorithm *expands* the state by simulating all the k actions C times. Instead of then recursing to all next states like SS, the FSSS picks the action a^* with a highest upper-bound for Q(s,a) and *selects* the next state s^* with the widest interval for V(s') among the states s' that were obtained by simulating a^* at s. The selected state s^* is then expanded until the leaf node is reached and the interval bounds of V- and Q-values are *updated* recursively on the path from leaf to root. When there is an action at the root with the Q lower bound greater than all other arms' upper bounds, i.e., the action is surely more highly rewarding than its siblings, the algorithm terminates.

As such, FSSS may also skip subtree expansions at branches where no further exploration would be helpful. When no pruning occurs, FSSS requires at most $(kC)^H$ trials. However, its running time may be up to $O(H(kC)^H)$.

Proposition 6.4.1. *The running time of FSSS is bounded by* $O(H(kC)^H)$.

Proof. At each non-leaf state node, *expansion* samples the simulator kC times, thus incurring a cost of $O((kC)^H)$ in total. In one FSSS trial, *selection* and *update* take $O(H)$, so since there are at most $(kC)^H$ trials, altogether these two actions cost $O(H(kC)^H)$. The total running time is therefore dominated by $O(H(kC)^H)$.

□

This means that in the worst case FSSS may require longer running time than SS. However, since the running time of FSSS is only at most exponential to the horizon time, its worst case behavior is much better than UCT's, which incurs a running time that grows super-exponentially to the horizon time in the worst case.

When applied in practical domains that require real-time behavior, a major drawback of FSSS is that it could be less responsive than UCT due to the time requirement for each rollout. Even though FSSS is anytime, at early stage of the search tree construction when there are only few nodes, its rollout could take

as much as $O(kCH)$ as compared to $O(H)$ of UCT. As we know from the SS analysis, the branching factor kC is usually much larger than H; in particular, the choice of C, H in [42] is such that $C = \Theta\left(\frac{H^2}{\gamma^{2H}}\right)$. As such, UCT could be more responsive than FSSS in real-time domains.

6.4.2 FSSS-Aux: FSSS with π-guided auxiliary arms

Similar to SS-Aux, we add one auxiliary arm at every internal state node of the tree up to a certain depth. At each of these auxiliary arms, the lower and upper bounds are computed using B π-guided rollouts of length L. The procedure is detailed in Algorithm 6 with the *aux-flag* set to true and $EstimateQ(s, a, \pi)$ being the empirical averaging estimation of π-guided simulations. Similar to the behavior of SS-Aux, it is straightforward to show that in the worst case, when auxiliary arms do not help to prune any branch, FSSS-Aux yields the same estimation as that of FSSS. In the good case when the heuristic policy is near-optimal, we expect many state nodes have their lower-upper bound gap closed faster. Additionally, when the algorithm is terminated prematurely due to resource constraints and the heuristic policy achieves good approximation, FSSS-Aux could leverage on this improvement and yield an improvement over vanilla FSSS. In fact, the following proposition states that, after terminating, FSSS-Aux is as good as SS-Aux, while the guarantee of FSSS remains the same as that of SS [75].

Proposition 6.4.2. *At termination, the action chosen by FSSS-Aux is the same as that chosen by SS-Aux.*

Proof. At termination of FSSS-Aux, there are two scenarios. The first scenario is when there is no pruning possible, in which case FSSS-Aux expands the same tree as SS-Aux. The output action is therefore the same as that returned by SS-Aux. The other scenario is when FSSS-Aux terminates before expanding the

full tree. By convention, this is when there is no point in further tightening the bound gaps because one arm surely has higher value than the rest, as its lower bound exceeds the upper bounds of other arms. This means the selected arm is guaranteed to yield the highest estimated value possible with respect to the fully expanded tree, which is exactly the one selected by SS-Aux. □

Complexity-wise, the number of trials required by FSSS-Aux has the same bound as that of FSSS.

Proposition 6.4.3. *The total number of trials of FSSS-Aux before termination is bounded by the number of leaves in the tree.*

Proof. The proof is identical to that of Proposition 3 by Walsh et al. [75], by noting that every trial of FSSS-Aux has to end at a node with zero bound gap. If such a node is not a leaf, it would not be selected in the first place, because it cannot be the node with widest bound gap. As such, every trial must end at a leaf node, i.e., the number of FSSS-Aux rollouts is bounded by the number of leaves in the tree. □

Similar to the situation with SS-Aux, the additional time required by FSSS-Aux is that of evaluating auxiliary arms. With suitable values of $\hat{H}$, this incurred cost is negligible, rendering the running time of FSSS-Aux being asymptotically comparable to that of vanilla FSSS, i.e. $O(H(kC)^H)$. Nevertheless, in the good case, when the heuristic policy is near-optimal, we expect the algorithm to terminate faster and produce a better approximation than FSSS, as the heuristic helps discover the optimal arm earlier. In fact, as depicted in Figure 6-5, FSSS-Aux quickly discovers a near-optimal policy for Obstructed Sailing.

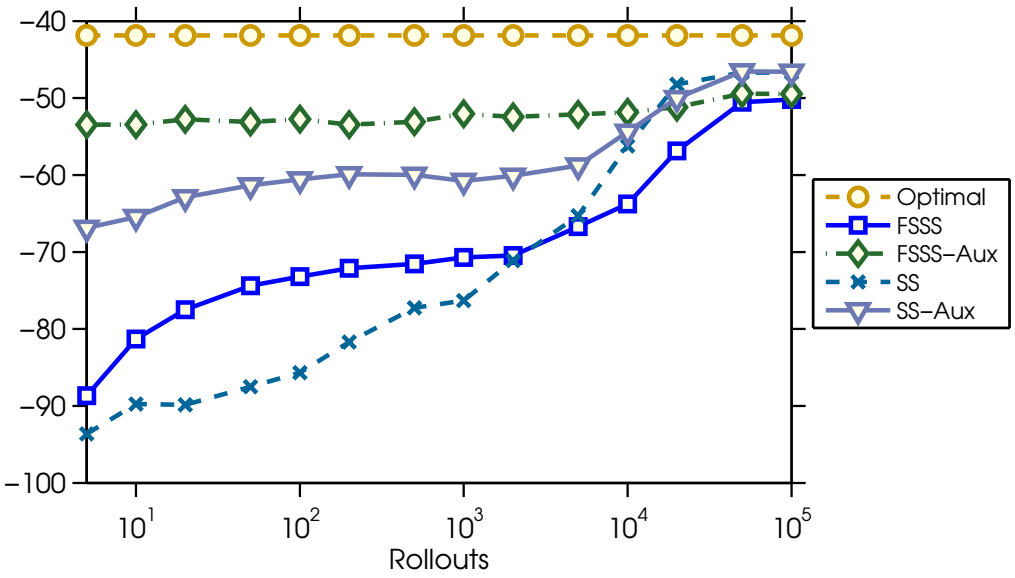


Figure 6-5: FSSS and SS variants in Obstructed Sailing with heuristic SailsTo-Goal; the charted lines denote the average accumulated reward (negative cost). We use 1000 random maps of size 20 by 20 with starting position at (5, 5) and goal at (15, 15), and the obstacles are placed with probability 0.4 at each grid square. The Aux versions of FSSS and SS respectively outperform the host algorithm.

Comparison with UCT-Aux

As analyzed in Section 5.4.1, UCT-Aux experiences its worst case behavior when coupled with a non-extreme heuristic, such as the policy StochOpt0.2[2] in Obstructed Sailing. In such cases, the algorithm wastes too many exploratory samples on auxiliary arms, and converges very slowly to the optimal policy. This behavior is inherent to all UCT variants, due to the over-optimism of the upper bound formula UCB1. In contrast, as FSSS and SS do not explore with such optimistic mentality, we expect them to be immune to such behavior.

To study this hypothesis, we compared the performance of UCT, FSSS and SS with their respective Aux-boostrapped versions when coupling with SO0.2, the unfavored heuristic of UCT-Aux. Note that since SS does not use rollouts to adjust its anytime behavior and each FSSS rollout typically takes more time to

[2]This heuristic performs optimal actions with probability 0.2 and random actions for the rest, which renders it always suboptimal but better than mere random walks.

complete than UCT rollouts, to ensure fairness in comparison, the algorithms' performance is assessed with respect to allocated time per planning step, instead of the number of rollouts as in Chapter 5. The results as charted in Figure 6-6 and 6-7 are averaged over 1000 runs with 200 randomized starting configurations, in maps of size 20 by 20.

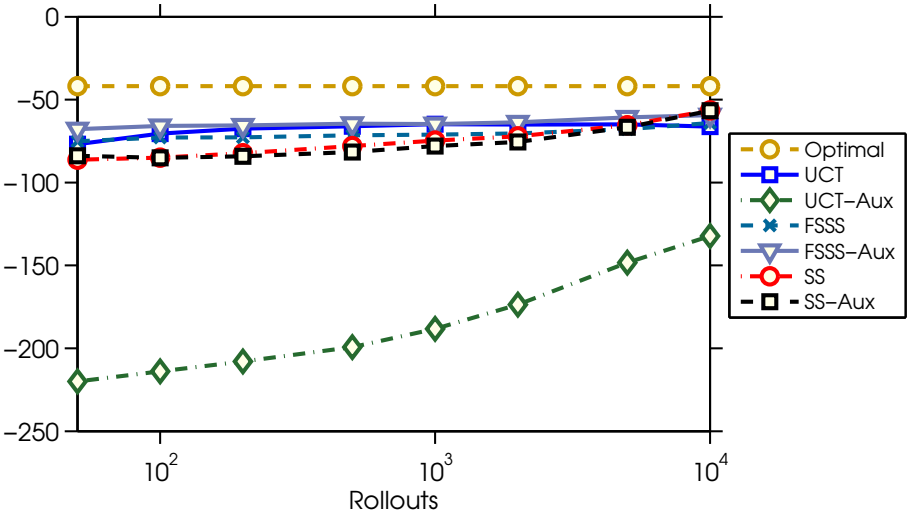
As shown in Figure 6-6a, UCT-Aux performs worse than its host algorithm, while FSSS-Aux and SS-Aux's performance is not affected as much, when combined with SO0.2. In fact, FSSS-Aux can even capitalize on the slight betterment of SO0.2, completely dominating vanilla FSSS in the chosen time allocations (Figure 6-6c). On the other hand, when coupled with suitable heuristics, i.e., SailsToGoal in this domain, UCT-Aux's performance is comparable to that of FSSS-Aux (Figure 6-7); both algorithms outperform SS-Aux in early stages of the experiments.

6.5 Experiments in Sheep Farmer

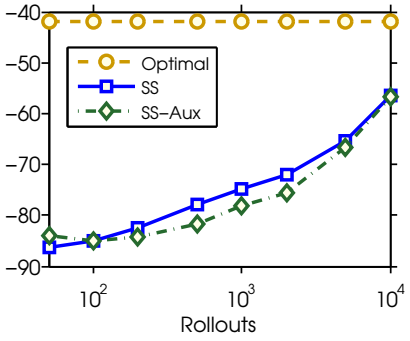
In this section, we investigate how the proposed algorithms work in solving large subgoal MDPs in the collaborative game of Sheep Farmer.

6.5.1 Sheep Farmer

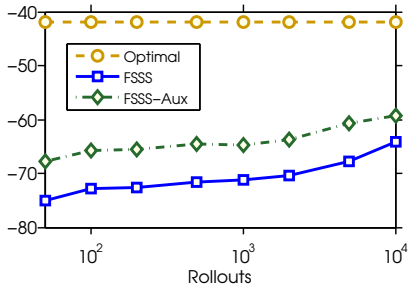
The setting is similar to that described in Chapter 5, i.e., with a farmer, a dog, two wolves and a sheep, in a subgoal MDP. The planner controls the farmer and his dog (farmer-dog team) to herd a sheep into a pen and kill two wolves in a maze-like environment. All wolves and sheep run away from the farmer and dog when close to them, otherwise the wolves chase the sheep and the sheep runs away from wolves. Both farmer and dog have 5 moves (no-move, N, S, E and W) but the farmer has an additional action to inflict damage on a nearby wolf, hence a total of $6 \times 5 = 30$ joint actions. The farm-team is given 5 points for



(a) UCT-Aux stumbles with SO0.2



(b) SS-Aux



(c) FSSS-Aux

Figure 6-6: Aux algorithms in Obstructed Sailing when coupled with StochOpt0.2.

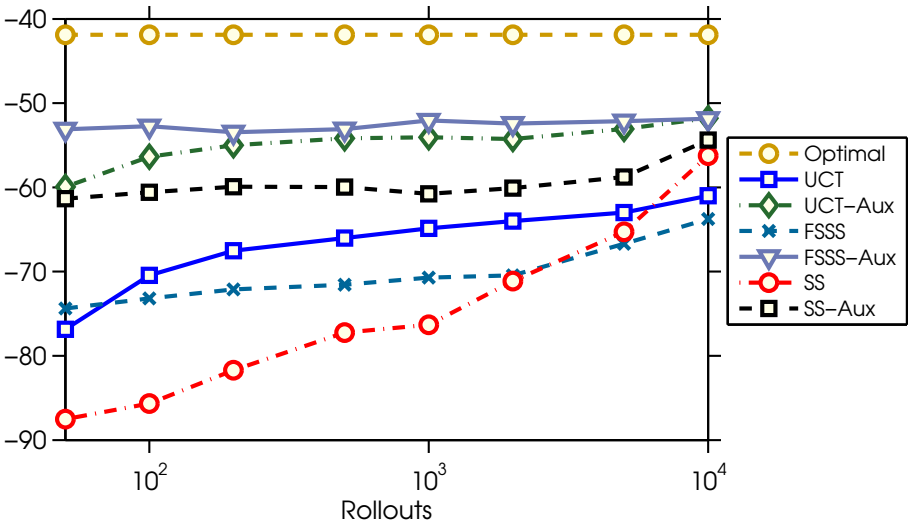


Figure 6-7: Aux algorithms in Obstructed Sailing when coupled with SailsToGoal.

successfully killing wolves and 10 points for herding sheep into its pen. In the event that the sheep is killed, the game ends with penalty -10. We used discount factor $\gamma = 0.99$.

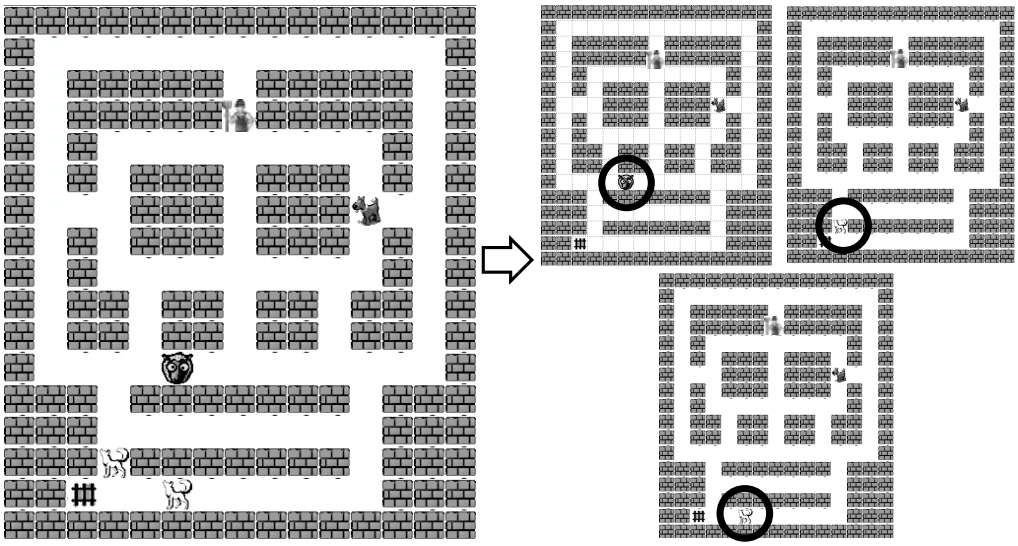


Figure 6-8: Task decomposition in Sheep Farmer.

We ran SS, FSSS and UCT side by side with their respective bootstrapped versions using the Aux method[3] in the map shown in Figure 6-8. The optimal policy in this domain was not computed due to the prohibitively large state space of size $104^5 * 3^2 \approx 10^{11}$ (each wolf has two health points). For fairer comparison, SS agents were implemented using the iterative deepening approach with respect to height H of the look-ahead tree. Each algorithm was allocated a fixed amount of time per planning step; due to time constraints, we did not run simulations with more than 50 seconds per planning step. In a real time game domain like this, the region of interest is typically from less than 1 to 5 seconds. The experiment was run 200 times, each with a different random starting position of the characters.

[3]The implementation of *UCT-Aux* is the same as that described in Chapter 5.

Heuristic policy: GoalAveraging

The heuristic policy used for auxiliary arms is the same as that used in the experiments of Chapter 5, i.e., GoalAveraging (GA). This heuristic adopts the task decomposition scheme of CAPIR, treating each NPC as one subtask MDP, and selects the action that maximizes the average of all subtask Q-values. Note that GA ignores the interaction between NPCs, i.e., wolves attacking sheep, and can be detrimental if the animals are in close proximity. Nonetheless, when the sheep is far away from the wolves, GA can be quite effective by prioritizing actions that lead to high rewards in each individual subtask. GA is therefore an "extreme" type, suitable for coupling with UCT-Aux.

Experiment results

We run the simulation-based algorithms in five maps (Appendix D) with increasing state space sizes. Figure 6-9 shows how the average accumulated rewards of SS, FSSS and UCT variants, the Random-action policy and our GoalAveraging-heuristic vary with respect to the allocated planning time, in map 6 (Figure 6-8). The charts for map 1 to 5, which show similar results and characteristics in performance, can be found in the Appendix E.

When coupled with a relatively poor (average reward -4) heuristic π_{GA}, both SS and FSSS improve over their original vanilla versions. Moreover, while vanilla FSSS exhibits much slower convergence to the optimal policy than SS does, FSSS-Aux convincingly dominates all other methods. Between two state-of-the-art algorithms UCT and FSSS, UCT-Aux shows improvements much earlier than FSSS-Aux, but is eventually outperformed by the latter. This result is the evidence of how different UCT and FSSS choose to explore the state space.

UCT's guarantee to examine highly rewarding branches exponentially more often than others has two effects on UCT-Aux's behavior

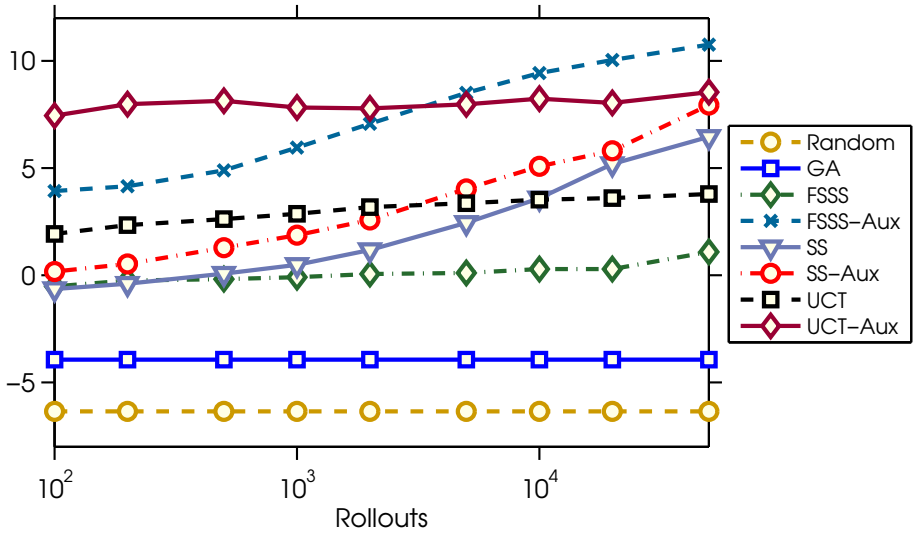


Figure 6-9: SS, FSSS and UCT variants in Sheep Farmer with GoalAveraging-heuristic. The charted lines denote the average accumulated reward of the algorithms in 200 experiments as function of planning time.

1. UCT-Aux quickly disregards the heuristic by wasting very few rollouts on auxiliary arms if they yield low rewards.

2. UCT-Aux hastily concentrates rollouts into non-expansive auxiliary arms if they appear to yield better rewards than random movement.

While the first characteristic is good, the second one is not desirable because it slows down the process of discovering the globally optimal action plan. Nonetheless, the technique is useful if the supplied heuristic is “extreme”, so that when the heuristic yields good reward, it is assured that the resultant reward is near-optimal.

On the contrary, FSSS sends simulations into most promising and uncertain regions of the search tree, where the actions’ upper bounds are high and the lower-upper bound gaps are wide. This approach slows down the process of discovering highly rewarding branches of vanilla FSSS as compared to that of UCT, but plays a crucial role in making FSSS-Aux safer than UCT-Aux, as it keeps exploring and improving even when a local minimum is discovered by the heuristic, as long as there are still uncertain regions in the search tree.

6.6 Chapter Summary

In this chapter, we examined the Aux technique of bootstrapping simulation-based algorithms SS and FSSS with a suboptimal heuristic policy. When coupled with suitable heuristics, the technique was shown to yield empirical improvement over the host algorithm in approximating the optimal action policy when competing in the same conditions of resource allocation. Among the bootstrapped algorithms, we observed that FSSS-Aux, inheriting a more conservative approach in exploration from FSSS than that of UCT, is the preferred algorithm in solving large state-space MDPs where abundant computing resource is available. On the other hand, in cases when the allocated resource is scarce, UCT and its variants are able to discover a reasonably near-optimal solution more quickly. UCT-Aux, however, must be adopted with care because its performance can vary greatly depending on the coupled heuristic, be it either "extreme" or non-extreme. This chapter also concludes our work in solving large state-space MDPs by combining offline-learnt heuristics with online approximate algorithms, the aim of which is to solve large subgoal MDPs that CAPIR may encounter when deployed in practical domains.

Chapter 7

Conclusion

In this thesis, we detailed a framework for autonomous agents that caters for a special class of two-party coordination, namely the Lead-Assistant Collaboration. This type of coordination is present in many real-life applications that require automated assistance to a leading agent; oftentimes this agent is a human being. This self-interest human agent, i.e., the *lead*, wants to achieve some objectives with the aid of an autonomous AI assistant in an environment whereby direct and explicit commands are not desirable.

For example, the assistant could be a smart software playing the role of a personal virtual assistant, such as Siri on the Apple iPhone[1], whose mission is to turn phone users' experience from a unidirectional, commanding mode to informal, human-like conversations. The aim of the assistant is therefore to invoke correct electronic commands (e.g. open up the right software with appropriate prompts for input) through informal chats. This possibly confusing mode of communication, while not directly expressing the intention of the user, can be used by the virtual assistant software as cues to figure out the user's concealed demand. In another scenario, the human expert can be the leader in a team of two, tasked to achieve some objective in an area whereby communication is pro-

[1] http://www.apple.com/ios/siri/

hibitive due to energy constraint (Mars Rovers) or the presence of competitors (spy team). In such settings, it is detrimental to assume a fixed behavior model of the lead agent, such as the globally optimal model. The reason is (1) because the human agent is usually prone to suboptimal behavior due to his bounded rationality and change of interest (mind switch), and (2) due to possible changes in objective ranking. As a result, the lead agent's behavior has great potential to deviate from any fixed model. Without accounting for such dynamics of the lead agent's behavior, the assistant could select actions that aim to achieve a different goal than that of the lead agent, rendering the collaboration a failure.

7.1 Contributions

In the general form, the Lead-Assistant Collaboration problem poses two major challenges that were proven to be extremely hard for the state-of-the-art technologies. In particular, the challenges, which are caused by two unobservable factors, i.e., the hidden intention and the behavior model of the lead agent, are both **PSPACE**-complete.

To achieve a practical system, we simplify the problem by modeling the lead agent's behavior as being optimal only at the subgoal levels. As such, each subgoal is modeled as an MDP, which can be solved either offline using traditional approaches such as Value Iteration, or online using simulation-based algorithms such as UCT. The formulation allows an efficient way to track the hidden intention using inverse planning by estimating the lead agent's intention based on his action history. The resultant framework, namely Collaborative Action Planning with Intention Recognition (CAPIR), is scalable with respect to the number of subgoals. Our qualitative and quantitative empirical evaluation demonstrate that the implementation of the framework can produce a smart assistant whose behavior is comparable to a human-controlled expert assistant in conditions close

to real-time requirement (one second of computation per planning step). Note that while many decision-theoretic frameworks in multi-agent settings such as I-POMDP and Dec-POMDP provide theoretically sound solutions, they deal mostly with asymptotic behavior.

7.2 Limitations of CAPIR

CAPIR solves only a subclass of multi-agent coordination problems. As presented in Chapter 4, we actually trade generality for practicality with the framework, imposing assumptions such as the *lead* agent is not within the planner's control, and that the lead is optimal at subgoal level. While such assumptions limit the scope and relevance of our work in more general settings, they make it easier, or rather more *tractable*, to reason for collaborative actions within the limitation of computing resources. As a matter of fact, what we strive for in this work is *not* an optimal assistant, but rather a most *helpful* one. If the lead agent decides to abandon all subgoals but one, we do not want the assistant to contemplate on the global goal; it should delve right into helping the lead to achieve that chosen subgoal.

7.3 CAPIR Beyond Modern Video Games

Although our investigation with the CAPIR framework is framed within the domain of modern video games, this choice of application domain is due to the inherently exploratory nature of research, and not the limitation of the approach. While the demand for a smart assistant in video games is already pressing, as argued in Chapter 1, one crucial factor that contributes to our decision is the controlled simulated environments offered by games. Different testing conditions can be setup simply by writing some code lines, instead of manually construct-

ing physical scenes, which is both costly and time-consuming. As formulated in Chapter 3 and 4, the generality of the problem formulation allows our CAPIR solution to be applicable to other real life scenarios such as creating smart spaces populated by smart virtual and physical autonomous agents. However, since our trial setting is simulated and fully controlled, there are some considerations and challenges in deploying CAPIR for real life domains.

7.3.1 Challenges

The crucial challenge when constructing smart assistants in general, and when applying CAPIR framework specifically, is to ensure real-time quality behavior. While "real-time" imposes the pressure of time on the planning process, "quality" translates to an efficient mechanism to track the human intent so that appropriate actions can be issued.

Real-time performance is the next stage of research, especially in the field of artificial intelligence, after having built theoretical foundations for the solutions. In fact, a smart assistant for a human being in the real world is often useless if it takes hours to come up with a good aiding move. With CAPIR, real-time behavior translates to an efficient formulation of the planning tasks such that the operators (belief update and action selection) can be executed quickly. While these operators only take time that is quadratically proportional to number of subgoals, the bottleneck here is the computation of subgoal optimal policies or their approximation. Domain-specific expert knowledge is therefore needed to formulate the subgoal MDPs such that their optimal policies can be computed exactly offline as much as possible. Ideally, there should be as few as one subgoal, which is the global task, whose optimal policy is approximated online. In the worst case when the subgoal MDPs cannot be reduced to a manageable level, CAPIR's performance relies directly on research advancements in sequential decision making, in particular the problem of solving large-sized/continuous

MDPs.

In order to select the most relevant assistive actions in a multi-intention setting, the CAPIR framework assumes that the exhaustive set of subgoals is furnished by the designer. In practice, subgoals could usually be organized in a hierarchy structure, in which those at the top levels can be decomposed into those lying at lower levels. Extracting an appropriate set of subgoals from this hierarchy could be troublesome. In many situations, this set could differ greatly from one person to another, and may even dynamically change as one gets more and more hand-on experience with the task and environment. The more familiar a person is at handling some types of tasks, the larger each of his possible subgoal gets. In other words, he can look at the tasks from a grander perspective than the first-timers can. Since our current CAPIR implementation only handles the tasks with stationary sets of subgoals, in real life, the possible dynamics of the lead agent's improved expertise, which indirectly changes the set of subgoals at execution time, could undermine the performance of CAPIR assistants. Tackling this challenge is left as a possible extension for the framework.

7.4 Future work

As an interdisciplinary product, CAPIR can lead to many derivative works as extensions on the results we report in this thesis.

One possible direction is to generalize CAPIR to the case of multi-agent coordination in which there are more than two agents involved. The idea is to maintain the biased relationship between agents by having a set of *lead agents*, each member of which is assisted by one or many members in the set of *assistants*. Among lead agents, there could be further hierarchical categorization, i.e., some lead agents are assistants of other lead agents, and so on. As a result, we could obtain a tree structure to represent the behavior dependencies among

the team members, since the assistants need to condition their behavior on their immediate lead.

Another interesting line of future work is to incorporate abstraction into the framework; at the current state, CAPIR operates on flat MDP representations[2]. In solving MDPs, state abstraction is an attractive option for reducing the dimensionality, thus state space's size, and has been investigated extensively in the planning community [32, 39, 46]. Therefore, if we could leverage on this body of research work, allowing CAPIR to deal with higher-dimensional domains, the frameworks' practicality would improve significantly.

Lastly, as mentioned in Section 7.3.1, CAPIR assumes the mind switch model of the lead agent and the set of subgoals are hand-crafted by domain experts; currently the framework relies on the intuition of experts, not having a guideline on how these CAPIR components should be constructed. Cognitive scientists could therefore leverage on the implemented engine and use it as an experiment platform to learn about the human mind's dynamics and behavior when dealing with integrable and decomposable options. Conversely, such psychological results can be transferred back into the engine to create a computational assistant that reasons with more dynamics, understanding the human mind better.

With many directions to carry on, we hope this work could inspire further research attempts in creating an ultimate AI entity: an assistant that can assist us intelligently in complex environments.

[2]Location abstraction implemented in CAPIR exploits the corridor characteristic of maze-like environments and is not generalizable to other domains.

Bibliography

[1] David W. Albrecht, Ingrid Zukerman, and Ann E. Nicholson. Bayesian models for keyhole plan recognition in an adventure game. *User Modeling and User-Adapted Interaction*, 8(1):5–47, March 1998.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[3] Chris L. Baker, Rebecca R. Saxe, and Joshua B. Tenenbaum. Action Understanding as Inverse Planning. *Cognition*, 113(3):329–349, December 2009.

[4] Chris L. Baker, Joshua B. Tenenbaum, and Rebecca R. Saxe. Goal Inference as Inverse Planning. In *Proceedings of the Twenty-Ninth Annual Conference of the Cognitive Science Society*, pages 779–784, 2007.

[5] R. K. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *21st International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.

[6] R. Becker, S. Zilberstein, V. Lesser, and C. V. Goldman. Solving transition independent decentralized markov decision processes. *Journal of Artificial Intelligence Research*, 22(1), 2004.

[7] Raphen Becker, Shlomo Zilberstein, Victor Lesser, and Claudia V. Goldman. Transition-independent decentralized Markov decision processes. In *Proceedings of the Second International Conference on Autonomous Agents and Multi Agent Systems*, pages 41–48, Melbourne, Australia, 2003.

[8] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, March 1957.

[9] Richard Bellman. A Markovian Decision Process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[10] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The Complexity of Decentralized Control of Markov Decision Processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[11] Craig Boutilier. Sequential Ooptimality and Coordination in Multiagent Systems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1 of *IJCAI'99*, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[12] B. Bouzy and B. Helmstetter. Monte-Carlo Go developments. *Advances in computer games*, 10:159–174, 2004.

[13] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[14] B. Brügmann. Monte Carlo Go. *Physics Department, Syracuse University, Tech. Rep*, 1993.

[15] Alex J. Champandard. Behavior Trees for Next-Generation Game AI. In *Game Developers Conference*, 2007.

[16] Edwin Chan. "Call of Duty: Black Ops" sets record for Activision. Available via http://games.yahoo.com/blogs/plugged-in/call-duty-black-ops-sets-record-activision-278.html, January 2011.

[17] Hyeong S. Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, January 2005.

[18] William G. Chase and Herbert A. Simon. Perception in chess. *Cognitive Psychology*, 4(1):55–81, January 1973.

[19] G. Chaslot, C. Fiter, J. B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. *Advances in Computer Games*, pages 1–13, 2010.

[20] Peter Chubb. Assassin's Creed 3 Review Fails to Mention Bugs. Available at http://www.inentertainment.co.uk/20121029/assassins-creed-3-review-fails-to-mention-bugs/, October 2012.

[21] Pierre-Arnaud Coquelin and Rémi Munos. Bandit algorithms for tree search. In *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, pages 67–74, July 2007.

[22] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In *Proceedings of the 5th International Conference on Computers and Games*, pages 72–83. Springer-Verlag, 2006.

[23] M. Cutumisu, C. Onuczko, M. McNaughton, T. Roy, J. Schaeffer, A. Schumacher, J. Siegel, D. Szafron, K. Waugh, M. Carbonaro, and Others. ScriptEase: A generative/adaptive programming paradigm for game scripting. *Science of Computer Programming*, 67(1), 2007.

[24] Daniel C. Dennett. *The Intentional Stance (Bradford Books)*. A Bradford Book, reprint edition, March 1989.

[25] Prashant Doshi and Piotr J. Gmytrasiewicz. Monte Carlo Sampling Methods for Approximating Interactive POMDPs. *Journal of Artificial Intelligence Research*, 34(1):297–337, March 2009.

[26] Alan Fern and Prasad Tadepalli. A computational decision theory for interactive assistants. In *Advances in Neural Information Processing Systems (NIPS-2010)*, 2010.

[27] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[28] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to General Game Playing. In *AAAI'08: Proceedings of the 23rd National Conference on Artificial Intelligence*, pages 259–264. AAAI Press, 2008.

[29] Daniel Fu and Ryan Houlette. *The Ultimate Guide to FSMs in Games*, volume 2 of *AI Game Programming Wisdom*, pages 283–302. Charles River Media, Massachusetts, USA, first edition, 2004.

[30] GameTrailers. Best of E3 2010 Awards Video Game, Best Multiplayer Game. Available at http://www.gametrailers.com/video/best-multiplayer-best-of-e3/701203, June 2010.

[31] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th International Conference on Machine Learning*, pages 273–280, New York, NY, USA, 2007. ACM.

[32] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1-2):163–223, 2003.

[33] Piotr J. Gmytrasiewicz and Prashant Doshi. A Framework for Sequential Planning in Multi-Agent Settings. *Journal of Artificial Intelligence Research*, 24(1), July 2005.

[34] Kevin Gold. Training goal recognition from low-level inputs in an action-adventure game. In *Proceedings of The Artificial Intelligence and Interactive Digital Entertainment Conference (2010)*. AAAI Press, 2010.

[35] Tom Goldman. Videogame Industry Worth Over $100 Billion Worldwide. Available at http://www.escapistmagazine.com/news/view/103064-Videogame-Industry-Worth-Over-100-Billion-Worldwide, August 2010.

[36] Alison Gopnik and Andrew N. Meltzoff. *Words, thoughts, and theories*. MIT Press, Cambridge, MA, 1998.

[37] Ryan Houlette. *Player Modeling for Adaptive Games*. Charles River Media, December 2003.

[38] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT˜Press, Cambridge, Massachusetts, 1960.

[39] Nicholas K. Jong and Peter Stone. State Abstraction Discovery from Irrelevant State Variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 752–757. Citeseer, 2005.

[40] Leslie P. Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, May 1998.

[41] R. H. Katz, D. Long, M. Satyanarayanan, and S. Tripathi. Workspaces in the Information Age. *Report of the NSF Workshop on Workspaces in the Information Age*, October 1996.

[42] Michael Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes. *Machine Learning*, 49(2):193–208, November 2002.

[43] John-Paul Kelly, Adi Botea, and Sven Koenig. Offline Planning with Hierarchical Task Networks in Video Games. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2008.

[44] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Proceedings of the 17th European conference on Machine Learning*, volume 4212 of *ECML'06*, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[45] H. Kurniawati, D. Hsu, and W. S. Lee. SARSOP: Efficient point-based POMDP planning by approximating optimally reachable belief spaces. In *Proc. Robotics: Science and Systems*, 2008.

[46] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a Unified Theory of State Abstraction for MDPs. In *Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics*, pages 531–539, 2006.

[47] Michael Mateas and Andrew Stern. Writing Façade: A case study in procedural authorship. *Second Person: Role-Playing and Story in Games and Playable Media*, pages 183–208, 2007.

[48] Bradford Mott, Sunyoung Lee, and James Lester. Probabilistic goal recognition in interactive narrative environments. In *Proceedings of the Twenty-first National Conference on Artificial Intelligence (AAAI-06)*, volume 21, pages 187–192. AAAI Press, 2006.

[49] M. Mundhenk, J. Goldsmith, C. Lusena, and E. Allender. Complexity of finite-horizon Markov decision process problems. *Journal of the ACM (JACM)*, 47(4), 2000.

[50] Truong-Huy D. Nguyen, David Hsu, Wee-Sun Lee, Tze-Yun Leong, Leslie P. Kaelbling, Tomas Lozano-Perez, and Andrew H. Grant. CAPIR: Collaborative Action Planning with Intention Recognition. In *Proceedings of the Seventh Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2011)*. AAAI, AAAI Press, 2011.

[51] Truong-Huy D. Nguyen, Wee-Sun Lee, and Tze-Yun Leong. Bootstrapping Monte Carlo Tree Search with an Imperfect Heuristic. In *Proceedings of the 23rd European Conference on Machine Learning (ECML 2012)*, September 2012.

[52] Truong-Huy D. Nguyen and Tze-Yun Leong. A Surprise Triggered Adaptive and Reactive (STAR) Framework for Online Adaptation in Non-stationary Environments. In Christian J. Darken and G. Michael Youngblood, editors, *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment International Conference (AIIDE 2009)*, pages 82–87. AAAI Press, 2009.

[53] Truong-Huy D. Nguyen, Tomi Silander, Wee-Sun Lee, and Tze-Yun Leong. Bootstrapping Simulation-based Algorithms with a Suboptimal Policy. Manuscript submitted for publication, April 2013.

[54] J. M. Ooi and G. W. Wornell. Decentralized control of a multiple access broadcast channel: Performance bounds. In *Decision and Control, 1996., Proceedings of the 35th IEEE*, volume 1. IEEE, 1996.

[55] Jeff Orkin. Applying Goal-Oriented Action Planning to Games. *AI Game Programming Wisdom*, 2:217–228, 2003.

[56] Jeff Orkin. Symbolic representation of game world state: Toward real-time planning in games. In *Proceedings of the AAAI Workshop on Challenges in Game Artificial Intelligence*, pages 26–30. AAAI Press, 2004.

[57] Jeff Orkin. Three states and a plan: the A.I. of F.E.A.R. In *Game Developers Conference*, volume 2006. Citeseer, 2006.

[58] Jeff Orkin and Deb Roy. The restaurant game: Learning social behavior and language from thousands of players online. *Journal of Game Development*, 3(1):39–60, 2007.

[59] Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov Decision Processes. *Mathematics of Operations Research*, 12(3):441–450, August 1987.

[60] Laurent Péret and Frédérick Garcia. On-line search for solving markov decision processes via heuristic sampling. In Ramon L. de Mántaras and Lorenza Saitta, editors, *The 16th European Conference on Artificial Intelligence (ECAI 2004)*, pages 530–534, 2004.

[61] Josef Perner. *Understanding the Representational Mind*. The MIT Press, Cambridge, MA, 1991.

[62] Joelle Pineau, Geoff Gordon, and Sebastian Thrun. Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1025–1032, August 2003.

[63] S. Poslad. *Ubiquitous computing: smart devices, environments and interactions*. Wiley, 2011.

[64] Martin L. Puterman and Moon C. Shin. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, 24(11), 1978.

[65] Earl D. Sacerdoti. *A Structure for Plans and Behavior*. PhD thesis, Stanford University, 1975.

[66] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4), 2001.

[67] C. F. Schmidt, N. S. Sridharan, and J. L. Goodson. The plan recognition problem: An intersection of psychology and artificial intelligence. *Artificial Intelligence*, 11(1-2):45–83, August 1978.

[68] Alan H. Schoenfeld and Douglas J. Herrmann. Problem perception and knowledge structure in expert and novice mathematical problem solvers. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 8(5):484–494, 1982.

[69] Joel I. Seiferas, Michael J. Fischer, and Albert R. Meyer. Separating Nondeterministic Time Complexity Classes. *Journal of the ACM (JACM)*, 25(1):146–167, January 1978.

[70] David Silver and Joel Veness. Monte-Carlo Planning in Large POMDPs. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS 2010)*, December 2010.

[71] Herbert A. Simon. *Models of Man: Social and Rational*. John Wiley & Sons, Inc., New York, 1st edition / 1st printing edition, January 1957.

[72] Trey Smith and Reid Simmons. Heuristic search value iteration for POMDPs. In *UAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 520–527, Arlington, Virginia, United States, 2004. AUAI Press.

[73] Peter Stone, Gal A. Kaminka, Sarit Kraus, and Jeffrey S. Rosenschein. Ad hoc autonomous agent teams: Collaboration without pre-coordination. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence*, July 2010.

[74] R. Vanderbei. Sailing strategies: An application involving stochastics, optimization, and statistics (SOS). Available via http://orfe.princeton.edu/˜rvdb/sail/sail.html, 1996.

[75] Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*, 2010.

[76] Henry W. Wellman. *The child's theory of mind*. Learning, development, and conceptual change. MIT Press, 1990.

[77] P. Xuan and V. Lesser. Multi-agent policies: from centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems*, pages 1098–1105. ACM, ACM Press, 2002.

[78] Judy York and Parag C. Pendharkar. Human-computer interaction issues for mobile computing in a variable work context. *International Journal of Human-Computer Studies*, 60(5), 2004.

APPENDIX

A Proof of Lemma 6.1

Lemma 6.1. *Let π be ϵ_0-optimal with $\epsilon_0 \geq 0$. For all $\epsilon_1 > 0$, with probability at least $1 - \epsilon_1$,*

$$|V^*(s) - \tilde{V}^{\pi}_{B,L}(s)| \leq \epsilon_0 + \gamma^L V_{max} + O\left(V_{max}\sqrt{\frac{\ln(1/\epsilon_1)}{B}}\right)$$

Proof. By using triangle inequality we get

$$|V^*(s) - \tilde{V}^{\pi}_{B,L}(s)| \leq |V^*(s) - V^{\pi}(s)| + |V^{\pi}(s) - V^{\pi}_{L}(s)| + |V^{\pi}_{L}(s) - \tilde{V}^{\pi}_{B,L}(s)|$$

with $V^{\pi}_{L}(s) = \mathbb{E}(X_{\pi,L,i})$ for all i. We are going to bound each term on the right hand side.

Firstly, since π is ϵ_0-optimal, by definition,

$$|V^*(s) - V^{\pi}(s)| \leq \epsilon_0. \tag{7.1}$$

Secondly, since the second term is the expected estimation error for V^{π} due

to simulating π only for L steps, it is bounded as follows

$$\begin{aligned}|V^{\pi}(s) - V_L^{\pi}(s)| &= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i R(s_i, \pi(s_i))\right] - \mathbb{E}\left[\sum_{i=0}^{L-1} \gamma^i R(s_i, \pi(s_i))\right] \\ &= \mathbb{E}\left[\sum_{i=L}^{\infty} \gamma^i R(s_i, \pi(s_i))\right] \\ &\leq \sum_{i=L}^{\infty} \gamma^i R_{max} = \gamma^L R_{max} \sum_{i=0}^{\infty} \gamma^i = \frac{\gamma^L R_{max}}{1-\gamma} = \gamma^L V_{max} \end{aligned} \tag{7.2}$$

Lastly, applying Hoeffding's inequality on B i.i.d. and bounded values $X_{\pi,L,i}$, for all $T > 0$, we obtain

$$P(|V_L^{\pi}(s) - \tilde{V}_{B,L}^{\pi}(s)| \leq T) \geq 1 - 2\exp\left(-\frac{2T^2B^2}{4V_{max}^2 B}\right) = 1 - 2\exp\left(-\frac{BT^2}{2V_{max}^2}\right)$$

By setting $\epsilon_1 = 2\exp\left(-\frac{BT^2}{2V_{max}^2}\right)$, which is equivalent to $T = V_{max}\sqrt{\frac{2}{B}\ln\frac{2}{\epsilon_1}}$, we have

$$P\left(|V_L^{\pi}(s) - \tilde{V}_{B,L}^{\pi}(s)| \leq V_{max}\sqrt{\frac{2}{B}\ln\frac{2}{\epsilon_1}}\right) \geq 1 - \epsilon_1 \tag{7.3}$$

Combining 7.1, 7.2 and 7.3, we obtain the PAC estimation error of $\bar{V}_{B,L}^{\pi}(s)$ as presented by the lemma. □

B Proof of Lemma 6.2

Lemma 6.2. *Consider SS with leaf nodes' values being ϵ_0-optimal, i.e., $|V^*(s) - V_0^{SS}(s)| \leq \epsilon_0, \forall s$. We have*

$$\epsilon_{SS} \leq T_{SS}(H, C) + \frac{\gamma^H \epsilon_0}{1-\gamma} \tag{7.4}$$

with $T_{SS}(H, C) = \frac{2\gamma^H V_{max}}{1-\gamma} + \frac{V_{max}}{(1-\gamma)^2}\sqrt{\frac{H}{C}\ln\frac{kC}{\gamma}}$.

The proof follows closely the analysis of SS by Kearns et al. [42] with free variables being C, H and λ; the recitation is purely for completeness.

The estimation error comes from two sources of inaccuracy. The first is that we use only a finite sample of size C to approximate the next state distribution. The second is due to propagated error in the tree, i.e., instead of using true optimal values $V^*(s)$, we use the estimation from lower levels to estimate the value of a node. In essence, the proof shows how the leaf error ϵ_0 propagates to the root with respect to C and H.

Step 1: PAC bound of limited sampling

Let us define an intermediate random variable $U^*(s, a)$ that captures the inaccuracy due to the limited sampling as follows

$$U^*(s, a) = R(s, a) + \frac{\gamma}{C} \sum_{i=1}^{C} V^*(s_i),$$

where s_i are drawn from the next state distribution $T(s, a)$. The next lemma by Kearns et al. shows that with high probability, the difference between $U^*(s, a)$ and $Q^*(s, a)$ is at most λ.

Lemma 7.1. *For any state s and action a, with probability at least $1 - e^{-\lambda^2 C/V_{max}^2}$ we have*

$$|Q^*(s, a) - U^*(s, a)| = \gamma \left| \mathop{\mathbb{E}}_{s' \sim T(s,a)} [V^*(s')] - \frac{1}{C} \sum_i V^*(s_i) \right| \leq \lambda,$$

where s_i are drawn from the next state distribution $T(s, a)$.

The proof is immediate from Chernoff-bound, and the lemma applies for any of $\lambda > 0$.

Step 2: PAC bound of propagated error

Under SS algorithm, let $V^n(s)$ be the estimated value of a state s at height n

(root is at height H) and $Q^n(s,a)$ that of pair (s,a). As such,

$$Q^n(s,a) = R(s,a) + \frac{\gamma}{C}\sum_{i=1}^{C} V^{n-1}(s_i),$$

where $V^{n-1}(s) = \max_a Q^{n-1}(s,a)$. We set $Q^0(s,a) = V^0(s) = V_{leaf}(s) \forall s, a$ with V_{leaf} being our ϵ-optimal estimation. Next, we define a parameter β_n that will eventually bound the difference between $Q^*(s,a)$ and $Q^n(s,a)$.

Let λ be some positive number. Define the series β_n recursively as $\beta_0 = \epsilon_0$ and $\beta_{n+1} = \gamma(\lambda + \beta_n)$. Solving for β_H, we obtain

$$\beta_H = \left(\sum_{i=1}^{H} \gamma^i \lambda\right) + \gamma^H \epsilon_0 = \lambda\gamma\frac{1-\gamma^H}{1-\gamma} + \gamma^H\epsilon_0 \leq \frac{\lambda}{1-\gamma} + \gamma^H\epsilon_0 \tag{7.5}$$

The next lemma shows that the error in the estimation at height n is at most β_n. Intuitively, the error due to finite sampling contributes λ, while the errors in estimation at height $n-1$ contribute β_{n-1}. Discounted by γ, the error at height n is then $\gamma(\lambda + \beta_{n-1}) = \beta_n$ by definition.

Lemma 7.2. *With probability at least* $1 - (kC)^n e^{-\lambda^2 C/V_{max}^2}$ *we have* $|Q^*(s,a) - Q^n(s,a)| \leq \beta_n$ *and* $|V^*(s,a) - V^n(s,a)| \leq \beta_n$.

Proof. The proof is by induction on n. It trivially holds for $n = 0$. Now, for $n \geq 1$

$$\begin{aligned}
|Q^*(s,a) - Q^n(s,a)| &= \gamma\left|\mathop{\mathbb{E}}_{s'\sim T(s,a)}[V^*(s')] - \frac{1}{C}\sum_i V^n(s_i)\right| \\
&\leq \gamma\left(\left|\mathop{\mathbb{E}}_{s'\sim T(s,a)}[V^*(s')] - \frac{1}{C}\sum_i V^*(s_i)\right| + \left|\frac{1}{C}\sum_i V^*(s_i) - \frac{1}{C}\sum_i V^n(s_i)\right|\right) \\
&\leq \gamma(\lambda + \beta_{n-1}) = \beta_n.
\end{aligned}$$

For the last inequality to hold, we require all C estimates to be good, for each

of the k actions. This means the probability of a bad estimate increases by a factor of kC, for each n. By Lemma 7.2, the probability of a single bad estimate is bounded by $e^{-\lambda^2 C/V_{max}^2}$. Therefore the probability of some bad estimate is bounded by $1-(kC)^n e^{-\lambda^2 C/V_{max}^2}$.

Denoting $a^* = \text{argmax}_a\, Q^*(s,a)$ and $\tilde{a} = \text{argmax}_a\, Q^n(s,a)$, by definition we have

$$V^*(s) = \max_a Q^*(s,a) = Q^*(s,a^*) \geq Q^*(s,\tilde{a}), \text{ and}$$

$$V^n(s) = \max_a Q^n(s,a) = Q^n(s,\tilde{a}) \geq Q^n(s,a^*), \text{ thus}$$

$$V^*(s) - V^n(s) \leq Q^*(s,a^*) - Q^n(s,a^*) \leq \beta_n, \tag{7.6}$$

$$V^n(s) - V^*(s) \leq Q^n(s,\tilde{a}) - Q^*(s,\tilde{a}) \leq \beta_n \tag{7.7}$$

Due to the PAC bound on Q^n, each of the inequalities 7.6 and 7.7 hold with probability at least $1-(kC)^n e^{\lambda^2 C/V_{max}^2}$. Since $|V^*(s)-V^n(s)|$ takes the larger value between the two, we obtain that $|V^*(s)-V^n(s)| \leq \beta_n$ with the aforementioned probability. □

Effectively, the lemma shows that with high probability, the estimation error for $Q^*(s_0,a)$ at the root is small.

Step 3: From PAC bound to Regret

The following Lemma (the proof is in [42]) links this PAC bound to the regret of a stochastic policy's expected value.

Lemma 7.3. *Assume that π is a stochastic policy so that $\pi(s)$ is a random variable. If for each state s, the probability that $Q^*(s,\pi^*(s)) - Q^*(s,\pi(s)) < \lambda$ is at least $1-\delta$, then the discounted infinite horizon return of π is at most $(\lambda + 2\delta V_{max})/(1-\gamma)$ from the optimal return, i.e., $\forall s, V^*(s) - V^\pi(s) \leq (\lambda + 2\delta V_{max})/(1-\gamma)$.*

Applying this lemma to the PAC estimation error β_H of the root's actions leads to

$$V^*(s) - V_{SS}(s) \leq \frac{\beta_H + 2V_{max}(kC)^H e^{-\lambda^2 C/V_{max}^2}}{1-\gamma}$$
$$\overset{(7.5)}{\leq} \frac{\lambda}{(1-\gamma)^2} + \frac{\gamma^H \epsilon_0}{1-\gamma} + \frac{2V_{max}}{1-\gamma}(kC)^H e^{-\lambda^2 C/V_{max}^2} \tag{7.8}$$

for all state s and any $\lambda > 0$.

By definition, $\epsilon_{SS} = \max_{s \in S} |V^*(s) - V_{SS}(s)|$; therefore the inequality (7.8) applies for ϵ_{SS} as well. Setting $\lambda = V_{max}\sqrt{\frac{H}{C}\ln\frac{kC}{\gamma}}$ and substituting this into the inequality above, yields the posited bound. (QED)

C Using value function's estimation in Belief Update and Action Selection

CAPIR framework relies on subgoal optimal action values or their estimates to reason about the lead agent's intention and compute most assistive actions. However, in real-time applications with large state-space subgoal MDPs such as MMORPGs, approximate algorithms usually are not allocated enough time to produce near-optimal estimates, which exposes the CAPIR engine to the risk of inadequately assessing actions' prospects. In this Section, we propose a technique that is used in CAPIR engine implementation to deal with this problem. We use UCT and its behavior to motivate our solution, but a similar reasoning can apply with any approximate algorithm.

C.1 Value estimation using UCT

As a recap, UCT guarantees to converge on choosing the optimal arm at the root node. However, that does not mean that the value estimations of other arms

are close to their respective true value. Recall that at any point of time, UCT guarantees to sample the most promising-looking arm exponentially more often than the rest. As such, seemingly suboptimal arms are allocated just a small fraction of the total number of rollouts, most of which are the results of random sampling episodes. Moreover, the weighted averaging poses a great discrepancy when compared with the true values because most of the episodes do not follow the optimal traces.

Therefore, we cannot directly use UCT's approximated values in our belief update and action selection steps due to the large inaccuracy. The only piece of information gained from UCT is a rough idea on whether an arm is optimal or not based on the its ranking when compared with sibling arms; the ranking criterion can be its sampling frequency or its estimated value. We can use this ranking data to construct *pseudo* Q-values and thus human action probabilities in the assistant's planning.

C.2 Pseudo Q-value estimation

In designing heuristics, it is usually desirable to obtain an evaluation function $\hat{H}(s_0, a)$ for all state-action pairs (s_0, a). For instance, in CAPIR, our Belief Update and Action Selection operators require numerical estimates of the optimal Q values of each subgoal MDP. However, in many cases, we can only get *pseudo Q-value estimation* instead, which is some source of information on actions' prospects but not in the form of $\hat{H}(s_0, a)$, such as when

1. The estimated behavior is in the form of a policy. For instance, at any state, we know a set of recommended actions to execute, but not their values. Allowing the game designers to suggest suitable actions instead of elaborating action values makes the process of designing heuristics much more hassle-free or oftentimes, feasible at all.

2. The estimated Q-values only helps in telling apart the most rewarding action but inaccurate preference among other actions. As presented above, this is what happens with UCT; since it guarantees to sample the most rewarding tree branch exponentially more often than the rest, badly performing actions do not receive enough trials to establish a ranking order between themselves. As a result, $Q_{UCT}(s_0, a)$ for all badly performing a are worthless.

C.3 Belief Update with Pseudo Q-value estimation

We could replace the action probability P, which is computed using exact Q-values, in the update operator by a binary function

$$\hat{P}(a_h|g_i, s) = \begin{cases} \frac{\beta_i}{Z} & \text{if } Q_{UCT}(a_h, a_{ai}, s, g_i) \geq Q_{UCT}(a_h^{UCT}, a_{ai}^{UCT}, s, g_i) - \epsilon \\ \frac{1}{Z} & \text{otherwise} \end{cases}$$

with

- a_h a lead agent (human) action and a_{ai} an assistant action,
- $(a_h^{UCT}, a_{ai}^{UCT})$ the most sampled arm (action pair) or one with highest estimated value at root s in subgoal g_i,
- ϵ some small positive number, and
- $\beta_i > 1$ being the factor that the lead agent is more likely to take an action that has the highest estimated value over other actions in subgoal i. Note that different subgoals can have different values of β_i, which can reflect how *obvious* an optimal action in subgoal i is to the lead agent.
- Z being the normalizing term such that $\sum_{a_h} \hat{P}(a_h|g_i, s) = 1, \forall i$.

If the furnished piece of information is in the form of a policy, i.e., the set of actions A^* the lead agent is likely to execute in every state given his goal, we can set $\hat{P}(a_h|g_i, s) = \frac{\beta}{Z}$ for all actions $a_h \in A^*$ and $\hat{P}(a_h|g_i, s) = \frac{1}{Z}$ otherwise, in each subgoal i.

The two-step belief update in Section 4.4.2 now goes as follows.

1. Belief drift due to Goal Change Model

$$B_t(g_i|\theta_{t-1}) = \sum_j T(g_j \rightarrow g_i) B_{t-1}(g_j|\theta_{t-1}) \tag{7.9}$$

   where $T(g_j \rightarrow g_i)$ is the switching probability from subgoal j to subgoal i.

2. Bayesian update upon action observation

$$B_t(g_i|a_t = a_h, s_t, \theta_{t-1}) = \alpha B_t(g_i|\theta_{t-1}) \hat{P}(a_h|g_i, s_{i,t}) \tag{7.10}$$

   where α is a normalizing constant.

This technique of updating belief requires a longer history to infer the lead agent's intention switch since the situational urgency captured by the true action value Q is not retained. Among bad moves, there could be further ordering as there are usually one or a few moves that are deadly, and other mediocre moves that do not lead to any progress; the technique ignores the action's badness and lumps all of them into the category of "suboptimal" actions. For instance, in Collaborative Hunters, the human action to attack a wolf when it is nearby is usually the best possible action; among the movement actions, those that go away from the attackable wolf are usually not as good as those towards the attackable wolf because they will allow the wolf some space to flee.

On the other hand, by disconnecting from the concept of action values, the technique allows more flexibility in modeling the lead agent's behavior. In the

previous chapters, the lead agent is assumed to be optimal at the subgoal levels and his behavior is constructed by solving the resultant subgoal MDPs in which the planner controls both players. This does not allows customization of the lead agent's behavior, unless the designers want to work on the reward function directly, which might lead to unexpected complications. It is easier and friendlier to game programmers if the lead agent's behavior can be described by scripting. As such, what is given to the belief update function at each state is a set of possible actions that the designers consider *reasonable* or *expectable* under the current circumstances; there is no information on what value the actions carry. In many cases the concept of action "value" could be hard to define or solicited. While the belief update process described in Section 4.4.2 depends heavily on the exact Q function, this belief update technique detailed above can receive the set of *reasonable* human actions and classifies them as approximately optimal. The belief is updated accordingly using Formulas 7.9 and 7.10.

C.4 Action Selection with Pseudo Q-values

If the optimal Q values are available, the assistant chooses the action that maximizes its expected reward as follows

$$a^*_{assist} = \text{argmax}_{a_{ai}} \left\{ \sum_i B_t(g_i|\theta_t) Q^*_i(s_t, a_h, a_{ai}) \right\}$$

Similarly to how Belief Update deals with UCT's estimated values and policy-based heuristics, we can replace Q^* by $\hat{Q}$ constructed as

$$\hat{Q}_i(s_t, a_h, a_{ai}) = \begin{cases} \xi_i & \text{if } Q_{UCT}(a_h, a_{ai}, s, g_i) \geq Q_{UCT}(a_h^{UCT}, a_{ai}^{UCT}, s, g_i) - \epsilon \\ 1 & \text{otherwise} \end{cases}$$

with all variables as explained in the Belief Update section and $\xi_i > 1$.

D Game levels used for experiments

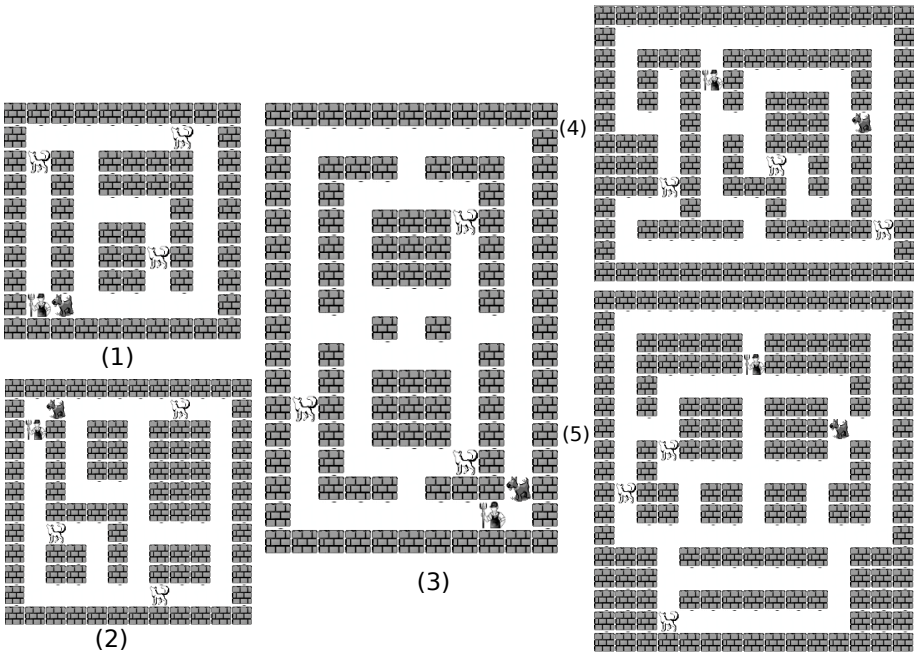


Figure D-1: Collaborative Hunters map layouts in Chapter 4.

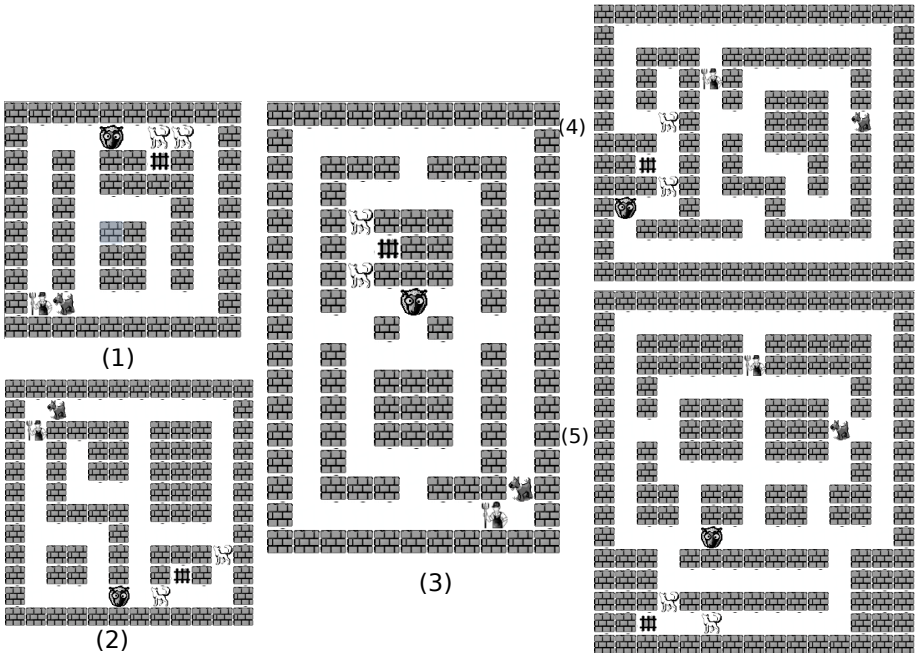
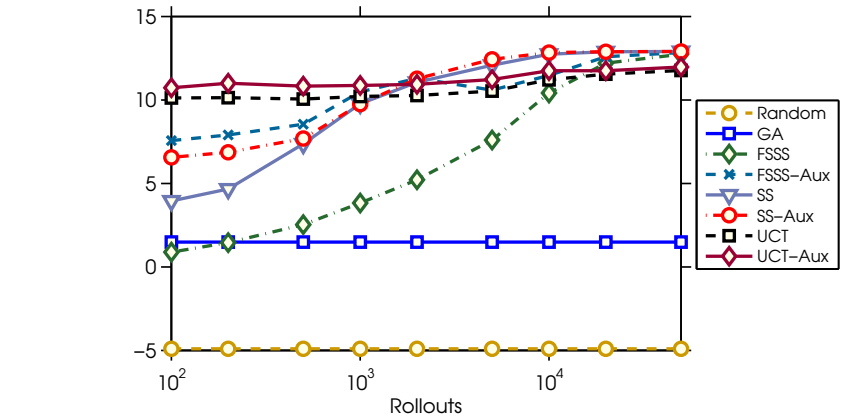
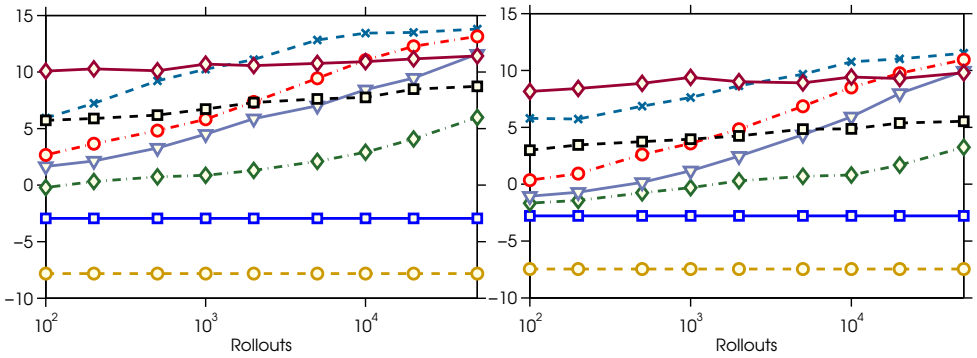


Figure D-2: Sheep Farmer map layouts in Chapter 6.

E Performance Charts in Sheep Farmer

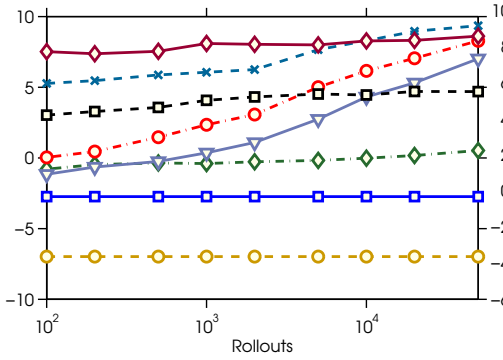


(a) Level 1

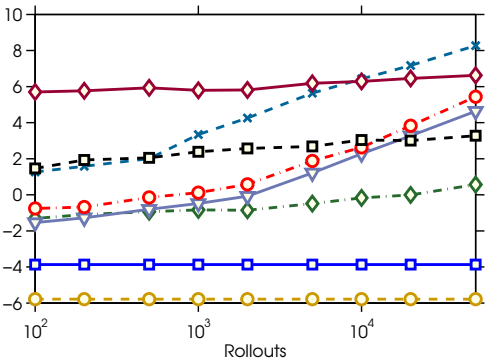


(b) Level 2

(c) Level 3



(d) Level 4



(e) Level 5

Figure E-3: SS, FSSS and UCT variants in Sheep Farmer with GoalAveraging heuristic. The charted lines denote the average accumulated reward of the algorithms in 200 experiments as function of planning time.