Active Policy Querying for Dynamic Human-Robot Collaboration Tasks

by

Jacob Broida

B.A. Carleton College (2017)

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science in Aerospace Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

© Massachusetts Institute of Technology 2021. All rights reserved.

Author . . . .
Department of Aeronautics and Astronautics
January 26, 2021

Certified by . . . .
Brian Charles Williams
Professor, Aeronautics and Astronautics
Thesis Supervisor

Accepted by . . . .
Zoltan Spakovszky
Professor, Aeronautics and Astronautics
Chair, Graduate Program Committee

Active Policy Querying for Dynamic Human-Robot Collaboration Tasks

by

Jacob Broida

Submitted to the Department of Aeronautics and Astronautics
on January 26, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Aerospace Engineering

Abstract

Success in any group task is dependent upon mutual understanding between the collaborators. Team members can use observation to infer their partner's plans, but such an approach carries great uncertainty and requires passive collaboration. For robots working alongside humans, verbal communication, especially in the form of questions, can provide definite and preemptive knowledge of a partner's policy. This knowledge in turn allows the robot to adapt its own plans to maximize team success. To that end, we propose a model and algorithms that will allow a robotic teammate to efficiently select the optimal questions to ask its partner in order to maximize the chances of team success. Our algorithms utilize decision graphs to compactly represent the policy space of team tasks. Using this compact representation, we are able to develop fast and efficient methods for determining the optimal set of questions. We exhibit four algorithms, one each for pre-execution questions, single scheduled questions, and multiple scheduled questions, and questions in tasks with communication restrictions. We show in experimental trials that these algorithm are capable of raising the success rate of a team task by up to 400% of the original value in typical scenarios.

Thesis Supervisor: Brian Charles Williams
Title: Professor, Aeronautics and Astronautics

Acknowledgments

First and foremost I would like to thank my advisor, Professor Brian Williams. It goes without saying that without him, none of this work would be possible. But, even more than that, he made the "work", more than work. So thank you, Brian, for welcoming me into your lab, and for giving me the support and encouragement I needed to succeed. Thank you for showing interest and enthusiasm in my work, and for constantly pushing me to dig deeper, and be clearer. Thank you for your patient and thorough approach to mentorship, especially in terms of this thesis. You always kept the focus on learning and improving, and because of that I cannot overstate how much I have learned and grown throughout this process. And most of all, thank you for being a kind and thoughtful advisor. You have a genuine interest in the happiness and well being of your community, and it does not go unnoticed. I never once doubted your investment in my success. Thank you Brian, truly.

A heartfelt thank you to Nelson Christensen, my undergraduate advisor at Carleton. There is no other single person that has had a greater impact on my success. I am eternally grateful for the time I got to spend working in the LIGO collaboration with you. That work was the very beginning of my love of research and coding. And thank you, Nelson, for your selfless support and encouragement throughout my early years as a scientist. When I think back to the best and most exciting parts of my career thus far, I can attribute almost all of them to your support. I am so very thankful to have been given the opportunity to work with you, and to be able to think of you as a friend. Here's hoping I'll be able buy you a Miller, once the quarantine lifts and you're back in town.

I am deeply grateful to the entirety of the MERS lab for all of the support and insight they have given me over the years. I could not have asked for a better group of people to work with. I would also like to single out the following lab members for their above and beyond contributions to my success. Steven Levine, for creating and developing Riker, the foundation upon which this thesis is built. Thank you for leading the way in this exciting and impactful project, I'm grateful to have been

able to continue it. Marlyse Reeves and Charles Dawson, for their friendship and support. Research is always more fun when you do it with friends! Allen Wang, for his steadfast friendship and constant encouragement as we went through this process (mostly) together. You really kept me motivated and grounded, and you helped me keep my eye on the light at the end of the tunnel. Nikhil Bhargava, first of all for being a fantastic TA. But, more importantly, for making my addition to the MERS lab possible, and for being a fantastic friend and mentor once I joined. I truly don't know where I would be without you. And finally, I would like to give an extra special thank you to Yuening Zhang. For her tireless and patient mentorship, her constant encouragement, and for all of the long hours she spent helping me shape the jumble of ideas in my head into a formal and well thought-out academic work. Yuening, thank you for going above and beyond. Without your mentorship, this thesis would not be what it is today.

I would of course like to thank all of my friends at MIT. The hardest part of quarantine has been having to be away from you all. Daniel Miller, thank you for being such a great friend, from Minnesota to here. We were in the trenches together, but it was much better than it could have been because you were there. Regina and Aaron, for being there to support and guide me through a very difficult time. I really cannot thank you enough for being there for me when I needed you. And Charlotte, for being my MIT BFF. I'm so grateful to have been able to mentor you through your journey. And, now and then, you gave me a little invaluable mentorship in return. I truly could not have reached the finish line without you. And to all the others, thank you for making the community special.

A big thank you to all of my friends, including (in no particular order) Emma, Bethany, Mike and Emily, Allison, Faisal, Nick, Daniel, Dylan, Jordikai, Alix, Brin, Shayna, Caleb, Hunter, Eric, Drew, Sanjay, Rachel and Amelia. Even if you didn't contribute directly to this work, your love and support allowed me to get to where I am today. From the bottom of my heart thank you all so much.

Finally, I would like to thank my cat Faye for being my most consistent and present friend, especially during quarantine. I would also like to extend an extra special thank

you for all of the times she woke me up while I was sleeping, or attacked me when I was relaxing. You were right kitty, I should have been working on my thesis. I would have made you a co-author if they had let me.

Contents

1 Introduction **17**

1.1 Motivating this Work 18

1.2 Outline of this Work 21

2 The Query Problem **23**

2.1 Identifying A Need 23

2.1.1 Case Study: Fighting Forest Fires 24

2.1.2 Traits of a Disaster Response Executive 27

2.2 Previous Work 29

2.3 Defining the Problem 32

2.3.1 The Query Problem as a Markov Decision Process 32

2.3.2 The Focus of this Work 35

2.3.3 Problem Specification 35

2.3.4 Example Problem Specification 37

3 Representing the Query Problem **39**

3.1 Execution Policy as a Decision Graph 39

3.1.1 Defining the Decision Graph 40

3.1.2 Obtaining the Decision Graph 41

3.2 Query Representation 43

3.2.1 Deterministic Query Model 43

3.2.2 General Query Model 47

3.2.3 Multi-Query Representation . . . 49
3.3 Case Study Revisited . . . 50
3.3.1 The Problem . . . 50
3.3.2 The Representation . . . 51
4 Solving the Query Problem **53**
4.1 Representative Example . . . 53
4.2 Compiling an Optimal Base Policy . . . 54
4.3 Pre-Execution Queries . . . 56
4.3.1 Problem Statement . . . 58
4.3.2 Approach . . . 58
4.3.3 Algorithm Notes . . . 62
4.3.4 Example Walkthrough . . . 64
4.3.5 Optimality . . . 66
4.4 Single Query Policy Development . . . 66
4.4.1 Problem Statement . . . 67
4.4.2 Approach . . . 67
4.4.3 Algorithm Notes . . . 68
4.4.4 Example Walkthrough . . . 69
4.4.5 Optimality . . . 71
4.5 Multi-Query Policy Development . . . 72
4.5.1 Problem Statement . . . 72
4.5.2 Approach . . . 73
4.5.3 Algorithm Notes . . . 74
4.5.4 Example Walkthrough . . . 75
4.5.5 Optimality . . . 77
4.6 Querying With Communication Dead Zones . . . 79
4.6.1 Problem Statement . . . 79
4.6.2 Approach . . . 79
4.6.3 Algorithm Notes . . . 81

5 Experimental Results **83**
5.1 Randomized Trials . . . 83
5.1.1 Experimental Methods . . . 83
5.1.2 Depth Tests . . . 84
5.1.3 Branch Factor Tests . . . 86
5.1.4 Query Saturation Tests . . . 87
5.1.5 Base SR Tests . . . 89
5.2 Case Study . . . 91
5.2.1 Fleshing out the Problem . . . 91
5.2.2 The Solution, Intuitively . . . 92
5.2.3 The Solution, Algorithmically . . . 92

6 Discussion **95**
6.1 Summary of this Work . . . 95
6.2 Tractability in Practice . . . 96
6.3 Future Work . . . 97

A Pike and Riker **105**
A.1 Pike . . . 105
A.1.1 Background . . . 105
A.1.2 Problem Statement . . . 107
A.1.3 Pre-processing . . . 108
A.1.4 Online Execution . . . 112
A.2 Riker . . . 113
A.2.1 Influence Diagrams . . . 114
A.2.2 The Conditional Stochastic Constraint Satisfaction Problem (CSCSP) . . . 114
A.2.3 Solving the CSCSP . . . 115

B Multi-Query Model **119**
B.1 Multi-Query Model . . . 119

List of Figures

1-1 Complex group tasks, such as navigating roadways and complicated intersections, require the actors to have a good understanding of others' intentions. 19

3-1 An example of one section of our decision tree representation. Here filled circles represent controllable decisions while unfilled circles represent uncontrollable decisions. The children of a given node represent possible successor decision states. Of particular note is the fact that the graph alternates layers. This is an important feature of our problem representation. 42

3-2 An example of how a query might affect the distribution over possibilities using a deterministic model. After querying, one outcome will have a probability of 1. Different models might sharpen the distribution differently. 44

3-3 Effect a question has on expected success rate 46

3-4 A compact representation of a fire fighting task assignment problem. Here controllable decisions are represented with white concentric circles while uncontrollable decisions are grey. The final transition to either a failure or success state is uncontrollable, and depends on both the controllable and uncontrollable choices made throughout execution. . 52

4-1 A representation of a collaborative task with four choices. For image clarity, rather than display the transition probabilities for the terminal uncontrollable nodes, we simply list their success rate in their node. . 55

4-2 Our example scenario after computing the optimal base policy. The optimal policy at each controllable decision is highlighted in orange. . 57

4-3 The example scenario annotated with details of the pre-execution algorithm. The ranked order of the nodes is shown in orange. The gain at the parent for asking about uncontrollable nodes is shown in green. The optimal nodes to query are outlined in green. Both are equally optimal. 65

4-4 Annotations of our representative example when we run our single query algorithm. The success rate of saving the question until the third layer is shown in orange, the success rate of asking the question right away at the root is shown in green. As comparison shows, the optimal policy is to save the question for the third layer. 70

4-5 Annotations of our representative example when we run our multi query algorithm. The optimal questions are to ask about the green node first, then whichever orange node execution pushes us towards. There is no space for any additional questions in this system. 76

5-1 Shown are randomized trials for our algorithms as a function of tree depth. For every depth, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. tree depth for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the depth increases. 85

5-2 Shown are randomized trials for our algorithms as a function of tree branch factor. For every branch factor, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. tree branch factor for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the depth increases. 87

5-3 Shown are randomized trials for our algorithms as a function of number of questions allotted. For every number of questions, we performed 100 random tests over several different tree depths and averaged the results. In the top graph, we have the success rate at the root vs. allotted questions for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the questions increase. 88

5-4 Shown are randomized trials for our algorithms as a function of base SR. For base SR, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. base SR for our algorithms. In the bottom graph, we show the total time taken by each algorithm to run as the base SR increases. 90

A-1 Pike in action on the Toyota Home Service Robot (HSR). In this image, Pike is assisting a human with a food preparation task. By observing the humans actions and making inferences about their intent, it is able to anticipate items the human will need and retrieve them. 106

A-2 An example of a standard TPNU. Concentric circles correspond to decision points in the plan; grey decisions are uncontrollable while white decisions are controllable. Blue events and yellow events correspond to actions by the autonomous agent and a human partner respectively. . 109

A-3 An example of how causal links may be used to annotate a TPNU. Image a shows the causal interactions between events in one section of a TPNU. Image b shows how a choice made earlier in the plan (x_{A_1}) can be used to prune branches later in the plan. 111

A-4 A high level overview of Riker under the hood. We will briefly cover the inner working of this system, but for a more complete picture see Dr. Levine's thesis. 117

Chapter 1

Introduction

When humans work and live together, much of the communication that happens between them is non-verbal. In fact, humans are quite good at understanding the implications behind someone's actions and from those implications inferring that person's intent. This is a natural process for us; when a friend comes to our house with a baseball glove, we understand that they want to play catch without them having to explicitly say it. Many of these inferences are not cut and dry however. High stakes tasks, complex actions with unclear implications, and limited observability all make this implicit understanding difficult. Fortunately for society, humans have a leg up on these inferences from years of social conditioning and evolutionary instinct. Unfortunately for society, when a robot is working with humans it does not have these luxuries. To compensate for this deficit, researchers have been developing ways for a robot build intent recognition capabilities based on observations [8, 16, 17, 27]. These methods are quite effective, but they do not yet capture the full extent of intent recognition tools available to humans. In order to enable fluid team work between humans and robots, it is essentially that robots are able to glean as much information about their human partners' plans as possible. To that end, we propose a simple extension to the observation-based intent recognition processes in use today: if a robot is not certain about what a human is planning, they simply ask. In this thesis, we explore and develop methods for a robot to resolve uncertainty in a human partner's plan by asking the right questions.

1.1 Motivating this Work

We focus on team plans for the context of the querying system we are developing. The success of a team is dependent upon the teammates having some mutual understanding of each other's plans and intents. Using this understanding, teammates can craft their personal plans to complement their partners'. To see this dynamic in action, consider the case of driving. Driving is a collaboration among human vehicle operators with the shared goal of each driver getting to their destinations safely and on time. Successful execution of this task hinges on each driver understanding their collaborators' intent and being understood by them in turn. For instance, one driver needs to understand when another driver wants to merge into their lane, so they can make space. On the other side, the merging driver needs to know that the other driver understands their plan and is prepared to make space so they can initiate the merge. If there is a mismatch in understanding, for instance if one driver intends to merge and doesn't realize the other driver hasn't picked up on this intention, a disastrous collision can result. The same collaboration among mixed human and autonomous teams is no different; if a human wants to merge into an autonomous vehicle's lane, that vehicle must be able to infer this intent and make space. If we want to build effective autonomous teammates, on the road or otherwise, they must be able to make these kinds of inferences.

Much work has already been done to build these intent recognition capabilities for autonomous team members. The GEORDI project has made advancements in providing exactly this capability for autonomous vehicles [14]. In a more general context, Levine and Williams demonstrated that, given a shared flexible execution plan, an autonomous collaborative agent can extrapolate human policies based on observed actions and act accordingly [18]. But while non-verbal intent recognition is important, it is not always sufficient. To see this, consider a search and rescue scenario. A rescue team might have some number of locations they need to search for a missing person. If the team needs to split up in order to accomplish this goal, the team must reach some level of consensus about who goes where. If one team member

is trying to decide which locations they should search, following another member around to see which locations they go to is not a valid option. Instead, they might ask their partner which direction they plan to travel in and infer their location choices from there. As this example shows, a few strategic questions can provide extensive information about a partner's plan. Our goal is to leverage this capability to extend the work done by Levine and Williams on collaborative robotic executives.

Figure 1-1: Complex group tasks, such as navigating roadways and complicated intersections, require the actors to have a good understanding of others' intentions.

Previous research in collaborative communication between robots and humans hasn't yet been able to provide the functionality to ask about intent in an equal-partner collaborative setting. In terms of questions, recent work often focuses on either robots as learners [4], or utilizes queries to account for deficits in perception [11]. In many deployment scenarios however, it is more effective to ask questions as a means of robust execution rather than for learning or as an alternative to sensor deficits. Research that does focus on dynamic collaborative dialogue often casts the robot as a subordinate dependent agent to the human [10]. In such situations, the robot is dependant upon clear and explicit information from the human in order to perform tasks. While safer from an execution standpoint, such restrictions limit the robot's

utility as well as consume valuable human resources in the form of time and attention. For stressful or high-intensity situations, it would be best for autonomous agents to ask their human partners questions only when necessary, and ensure the questions are strategic enough to maximize the success of their collaborative task.

Recently, Bernardini et. al. exhibited a near-optimal greedy algorithm for submodular systems that can be used quite effectively for planning information gathering actions [2]. This work focuses on crafting sequences of actions that are most likely to meet an objective, for instance finding an object via search patterns. Though questions can be viewed as simple information gathering actions, their role in a task under uncertainty will allow us to extend their utility via powerful contingent planning. Another potential approach involves framing the problem as a partially observed Markov decision process (POMDP). Work on that front has modeled singular, full-state "oracle" questions that may put too much strain on the human [1,23], or allows a wider range of specific questions but suffers from the complexity issues endemic to POMDPs [12]. These issues suggest that a model that keeps the state space of querying related actions implicit may be more successful.

For the purpose of this work, we desire a robot that is able to operate and converse on an equal footing with its human counterpart. To that end, we propose a querying system that will allow a robotic executive to select questions about human intent during collaborative execution. Coupled with a robotic system that is both autonomous and independent, active querying will greatly enhance the robotic agent's ability to predict human policies, and thereby pick its own actions to maximize their collaborative success rate.

This thesis offers five main contributions towards the goal of dynamic collaborative dialogue between robots and humans working together on a shared task: a compact decision graph formulation of the problem, an algorithm for planning pre-execution questions, an algorithm for developing a policy for a single question during execution, an extension to that algorithm for any number of questions scheduled throughout execution, and finally an algorithm for planning questions during a scenario with communication dead zones. We show through experimental trials that these querying

strategies have significant impact on the effectiveness of robot-human teams, with up to 400% increases in collaborative success rates.

1.2 Outline of this Work

We begin this work in chapter two by building and motivating the problem we are trying to solve. Our ultimate goal is to advance the capabilities of an autonomous coordination assistant for teams of human agents. To demonstrate a need for such an advancement, we examine a case study on forest firefighting response teams. From that case study, we extract underlying characteristics that make the problem space challenging. Those challenges then inform the capabilities that we desire in an ideal autonomous assistant. Next we examine how recent task executive research done by the MERS lab can be used to satisfy the assistant's needs. After examining recent work, we identify a gap in capabilities that can be filled by giving the executive the ability to ask questions of the other team members. Finally, we the problem formed by these gaps can naturally be framed as a partially observed Markov decision process. We conclude the chapter by formally defining the problem we are trying to solve and providing a grounded example problem mapped to that definition.

After formally defining the problem, we are free to build a problem representation in chapter three. We demonstrate how a decision graph representation will allow us to keep the question-related states of a POMDP implicit. Next, we model exactly how questions affect the this representation and show how it can be used to select optimal questions in the simplest scenarios. Finally, we return to the firefighting example to show how these representation can be applied to a real world scenario.

In the fourth chapter we develop algorithmic solutions for selecting queries using our problem representation. We begin building solutions to the query problem by starting with the simplest case and increasing the complexity as we go. In order, we develop solutions for a single query before the team plan begins, a single query any time during the plan's execution, n queries any time during the plan's execution, and finally n queries during a plan with questions allowed only at certain times during

task execution.

In the fifth chapter, we provide experimental benchmarks to validate the effectiveness of our algorithms. We show that providing the executives with query capabilities can increase the success rate of a team by up to 400%. Further, we show that the algorithms we develop are generally able to select these queries on time scales of less than a second, laying the groundwork for seamless communication between a human and a robot. We conclude the chapter by running our algorithms on an example from our firefighting case study to illustrate how they would work in the field.

In the sixth and final chapter, we conclude this work with a summary and discussion of the advancements made. Additionally, we discuss potential future work that could build upon the research done here. This future work includes generalizing queries to any informative action, investigating informative actions that affect the world, and developing benchmarks to compare against other possible representations of this problem.

Chapter 2

The Query Problem

In this chapter, we outline how a robust autonomous executive can be impactful in real world scenarios using currently available robotic mediums. To show this, we first examine one such scenario, forest fire fighting, highlighting what makes the problem difficult and interesting. From there, we can specify what traits a robotic executive needs in order to be an asset in fire fighting operations. Among these traits, we emphasise chance constrained execution and strong teamwork skills when working with humans. We then detail previous work by the MERS lab in pursuit of developing just such an executive. Next we identify a gap in recent work that can be filled by adding active estimation capabilities in the form of verbal queries. Finally, we tie the early discussion together with a formal definition of the query problem, which we solve in the remainder of this thesis.

2.1 Identifying A Need

In this section, we will identify a real world problem where autonomous robotic systems can provide a tangible benefit. Based on this problem, we extract capabilities that would give an autonomous system the power to provide that benefit.

The MERS lab has ongoing work on several different paradigms of the human-robot assistant relationship. One of these, robots as physical assistant, caters to tasks like handing a human worker a tool when they need one [17]. Another one, robots

as negotiators, helps members of a team arrive at a quick consensus when plans go awry [29]. Finally, robots as coordinators is a paradigm in which it is the robotic assistant's job to assign tasks to humans. These types of assistants have many traits in common, including communication skills, team-based planning capabilities, and a need for recognizing and understanding a human's plans based on observed actions. At the same time, each is unique enough that it demands its own focus. In this work, we will be focusing on robots as coordinators. In a team, the coordinator's primary role is to select actions for field teams to accomplish. Generally such an assistant does not actually perform any physical task, but instead assists the group by taking the onus of coordination away from field teams. We will be investigating ways of improving an autonomous coordinator's capability to make well informed decisions, which will in turn maximize a team's chance of accomplishing its objective. We begin by looking at a case study to build motivation for this research and emphasize its potential impact.

2.1.1 Case Study: Fighting Forest Fires

Disaster response is a field in which robotic workers already make significant impacts using widely available platforms. Drones especially have been employed for their ability to quickly and safely scout and map disaster areas [3, 15]. The potential for autonomous systems to further transform and improve the way disaster response operations are conducted is immense, but there remains plenty of work to be done to make this potential impact into a reality. In order to highlight ways that autonomous systems can assist in disaster operations, we will investigate one specific type of disaster response scenario.

For our example, we'll consider a forest fire fighting scenario. The scenario starts with a poorly managed campfire that has led to a major forest fire in a rural area. To respond to this incident, fire, medical, and engineering teams are called together from the surrounding area to form a task force. In charge of this task force is a small command team of coordinators and dispatchers, who must monitor the developing situation as they send tasks to field teams. The task force's job is to contain the

fire while rescuing any civilians trapped within the burning forest. They can contain the fire by manning choke-points and fire lines identified in the target region. Unfortunately, there are two major complications. The first is that, having been called together from a wide area, not all of the individual teams within the task force have experience working with the other teams. This can make it difficult for a given team to anticipate another team's skills, roles, and personalities, which in turn makes coordination difficult. The second complication is that there are a number of homesteads scattered throughout the at-risk area. Though an evacuation order has been issued, it is not clear exactly which homes have been evacuated and which still contain trapped civilians that need to be rescued. The task force must balance fire containment operations with search and rescue operations, and must be ready to change plans and adapt at a moment's notice as the situation develops.

Now that we've laid out a scenario, we can identify some underlying factors that make addressing that scenario difficult.

High Uncertainty

Smoke reduces visibility, homesteads may contain trapped civilians, and gusts of wind can rapidly change the tempo and direction of the fire's spread. Compounding this issue is the fact that the developments themselves can be fast, dangerous, and significantly impact the plans of the disaster workers. For example, if a certain homestead believed to be empty suddenly shows signs of life, the nearest fire team may have to drop everything to begin rescue operations. This action in turn may have ripple effects if they need medical teams on hand, their dropped work is a major priority, or they need an escape route to be maintained by another fire team.

Another significant source of uncertainty comes from the individual teams within the task force. Disaster response task forces can be quite large, numbering 70 or more individual workers organized into teams. While the teams may be familiar with their own members, there is no guarantee there is familiarity between the teams. These teams need to be able to work together well despite not being acquainted with each other or having a full understanding of their partner's roles or skills. As we shall see,

the issues raised by uncertainty among the teams presents an especially big challenge due to the fact that coordination is vital for disaster response.

Strong Coordination is Vital

Given the large, diverse, and specialized teams that often respond to disaster situations, there is an intrinsic need for good coordination at all levels. Well coordinated teams can protect each other. For example one fire team might maintain an escape route for another. Additionally, well coordinated teams ensure that their skills are applied to the maximum effect, as would be the case in a command team identifying the best possible choke point for a fire team to work on containment. Likewise, poor coordination can put rescuers in danger and reduce the efficacy of rescue operations. If a fire team attempts to rescue civilians and no team shifts to fill in on their containment work, the fire could potentially escape. Managing the team plan is an onerous task for the human coordinators given that they must juggle an immense number of factors, incoming information, and developing situations. At the same time, field teams need to stay up to date on their orders and make sure they see them through properly. All these factors come together to show that robust coordination skills are essential, especially considering the high stakes scenario responders are working with.

High Risk, High Reward

The fire fighting task force's goal is two-fold: to protect human life and contain the fire. These are high stakes, and failure to accomplish these goals quickly can result in loss of life and significant damage to the environment. At the same time, disaster workers must take their own lives into account when saving lives and fighting fires. The National Outdoor Leadership School, an educational institution that trains wilderness medical responders, teaches that the first priority is always to keep yourself and your teammates safe to avoid creating additional victims [26]. If a fire team rushes into a burning homestead on the verge of collapse to save a trapped civilian, they may become injured themselves and require rescue. This chain reaction can create a snowball effect that drains resources and reduces the effectiveness of the task force.

Thus, our high reward scenario has the additional facet of being extremely high risk. Disaster workers must carefully weigh the danger to themselves and their teammates from each action they take against the prospective lives they could save. This duality puts immense pressure on the individuals making decisions and requires them to be as certain as possible about every choice they make.

2.1.2 Traits of a Disaster Response Executive

Based on the challenges we've identified in the fire fighting problem, we can extrapolate vital properties that an autonomous team member would need in order to be helpful in such a scenario. While there are many facets of disaster response systems that present interesting research questions, we will focus on the **task executive**. It is the task executive's job to use the available information to select tasks for agents to perform in order to reach a goal. For example, if the goal is to put out a fire, a drone's task executive might direct it to first collect water from a reservoir, then navigate to the fire site, and finally drop the water onto a critical point in the fire. Our goal will be to provide the capabilities such an executive would need in order to be an asset in a fire fighting scenario. To provide a focused discussion, we will consider our work in the context of an autonomous planning assistant, who's job it is to help the task force command center select tasks for its operational teams. From the above facets of our fire fighting scenario, we can identify several important abilities the task executive for such an assistant must posses in order to be helpful:

Dynamic Execution

We are considering a problem that is highly uncertain. Paths may be blocked, victims may not be where we thought they were, unexpected developments may occur that significantly alter the team plan. Our system must be capable of **dynamic execution**, meaning that it must be able to adapt online to disturbances, developments, or alterations that affect the team plan *as it is being executed*. If a fire team attempts to rescue a surprise victim for example, the assistant must quickly provide them backup

and compensate for their lost containment work. Of course, it is not enough to simply adapt; we must also ensure that our system adapts *well*. To that end, we include a specification of robust execution.

Robust Execution

As we have seen, disaster response is an extremely high-risk high-reward scenario due to the unfortunate fact that human rescuers often must put themselves in danger in order to save the victims. To protect the operation and avoid creating more victims, we require **robust execution**. Robust execution entails the ability to cope with issues that arise while maintaining a high likelihood of plan success. For example, in our fire fighting scenario, an issue that may arise would be a sudden gust of wind that spreads the fire to a previously contained zone. To achieve robust execution, the assistant must be able to respond to that development while still maintaining a high chance of keeping everyone safe and the fire contained. To quantify the level of robustness we wish to achieve, we include the additional goal of chance constrained execution.

Chance Constrained Execution

Chance constrained execution further solidifies the concept of robust execution by specifying a level of robustness we must meet. In practice, this means specifying a **chance constraint**, which is a constraint on the likelihood of execution success. If we cannot meet that chance constraint, we must enter some sort of failure response mode. Often that means signaling failure and canceling execution. In the fire fighting context, say we want to maintain a 99% chance that the firefighters return from a burning homestead safely. If the fire spreads, it may no longer be possible to certify that level of confidence that the fire fighters can continue their mission and still make it out alive. In that case, we might signal that the situation is no longer tenable and pull the firefighters out.

Teamwork Oriented Planning

We have already discussed both why teamwork is important in a disaster scenario and why it is difficult. A good disaster response assistant should be capable of making sure the teams adhere to the plan and are synchronized with their partners. Key elements of the level of teamwork needed include understanding the intent underlying a team's actions, picking tasks to synergize the strengths and goals of multiple teams, and communicating when understanding is lost among the teams. This final element, which involves the assistant identifying and compensating for a gap in information, comprises our final key skill: active estimation.

Active Estimation

For our assistant to work in an uncertain and dynamic world, it must be constantly assessing and estimating the state of the world. In a passive system, the assistant must perform these estimations based solely on information that is provided to it. In many cases, that is not enough. Consider the homesteads dotted throughout the fire zone. If there are civilians trapped or unconscious inside a given homestead, they won't be able to make their presence known. A passive system would never find them, and so their lives would be lost. Thus, our assistant should go a step further and perform **active estimation**. Active estimation entails a system that identifies crucial and resolvable sources of uncertainty, and dispatches actions that resolve them. With active estimation capabilities, our assistant should be able identify and resolve uncertainties that could threaten the task force's work before they become a problem.

2.2 Previous Work

So far we have 1) identified a real world scenario where a robust task executive can have an impact and 2) identified facets an executive would need in order to make that impact. In this section we will review work the MERS lab has done in the past that could be used to meet this executive's needs, as well as identify a way to take that work one step further through active estimation.

The MERS lab has been developing task executives in line with the above capabilities for quite some time. Much of the work in this regard has had a focus on robust execution in coordination with humans. In 2007, Shuonan Dong presented her Master's thesis on development of automated systems that can recognize human intent. Her system used a statistical learning approach that learned plans and actions based on training data, rather than pre-specified action inputs. Dong's work built on the concept of temporal plan networks (TPNs). TPNs are a data structure that specify events, activities, choices, and the temporal constraints between them. Dong improved the TPN's utility in uncertain environments by adding in representations of probabilistic events. In terms of recognizing human activities as actions progress, Dong created an algorithm that constantly updated the probabilities held within the TPN based on information fed to the system as actions were performed. With all of these capabilities together, her system was able to predict future activities planned by the human based on a TPN in the context of data collected online at execution time [8]. By providing the capability to predict human actions, Dong's system laid the ground work for executives that can coordinate with humans in dynamic tasks.

Shuonan Dong's work on robust execution in human robot teams was carried forward by former MERS member Professor Julie Shah. In 2011, Shah presented her Ph.D. thesis on fluid coordination of human-robot teams. Shah took Dong's human intent recognition work to the next level with the creation of Chaski, a task executive for human-robot teams. Chaski was built to emulate coordination behaviors observed in humans, which led to the capability of collaborative planning with deference to verbal and gesture-based cues and commands. This collaborative behavior, along with support for just-in-time scheduling, allowed Chaski to coordinate dynamically with partners without falling prey to the intractable computation times that usually plague high-dimensional plan spaces. [27]. Both Chaski and Drake, a similar task executed focused on decision making in a general context, were demonstrated on JPL's All-Terrain Hex-Limbed Extra-Terrestrial Explorer (ATHLETE) robot. Chaski was used to handle item passing between the grippers on the ATHLETE robot, while Drake was used to execute a construction task featuring choices [9].

Most recently, Steven Levine focused his 2019 doctoral thesis on folding in risk to a teamwork oriented objective. Levine's work used an extension of TPNs known as Temporal Plan Networks under Uncertainty (TPNUs). TPNUs include all of the information encoded in a TPN, with the addition of choices that cannot be controlled by the executive. Levine's first system, Pike, was designed to use causality to infer a partner's plans based on their actions. For example, Pike might be partnered with a human to help them get ready every morning. During a given morning, Pike might see a human get a mug from a cabinet and infer based on causal logic that the human must be planning to make coffee. Pike can then recognize what the human needs to be successful in their coffee-making task and move to assist them accordingly. Levine then extended the Pike framework by adding in risk and chance constraints to the tasks. This evolution of Pike, named Riker, was able to choose it's actions in the context of it's teammates' plans in order to ensure that the chance of team success met a given chance constraint. The risk might come from an environmental factor, for example the severity of the weather, or it might be in the partner's plans, such as whether they are planning on biking or driving to work. Riker would account for uncertainty when planning to accomplish team goals, and would be able to pick it's decisions so that the team's plan was under some minimum risk value [17]. For a high level view of the Pike and Riker systems, see Appendix A.

One major challenge facing the Riker system was the issue of probabilistic deadlock, in which there is too much uncertainty to safely continue execution. This challenge highlighted a major area for growth in regards to Riker's passive approach to gaining information. In the absence of causal influences, Riker must wait for a human to make a decision before it can glean any information about what choice they will make. The issue lies in the fact that, in many cases, for a partnership to be effective, the partners cannot afford to wait and see what the other's plans are. Fire fighting is a good example of this challenge. If multiple fire fighters learn of a trapped victim at the same time, they can't afford to simply wait around to see if their partners will respond. They must either *act* on the situation or *ascertain* the information that allows them to make the right choice. Riker has a flat-footed approach to informa-

tion it can glean. If there is information that Riker can't observe directly, such as a human performing an action in another room, it has no way of resolving that source of uncertainty. In a disaster scenario when lives are on the line, it is imperative that an executive system is able to resolve uncertainty when possible. In this case, our system must be able to resolve sources of uncertainty as needed during online execution. Levine included in his thesis preliminary work for one such method of resolving uncertainty: asking insightful questions. In this work, we will continue to develop upon this idea and show it's efficacy in collaborative tasks.

2.3 Defining the Problem

2.3.1 The Query Problem as a Markov Decision Process

Before we formally define this problem, we must develop some intuition as to how it behaves. To begin, we note the problem of collaborative execution is fundamentally a decision problem, meaning there are observations interleaved with choices. The observations we are concerned with in this problem are those pertaining to a partner, primarily in regards to what decision they will make or have made. The choices we make must complement our partners' choices, meaning observations are critical to picking a good policy.

Now consider as an example an arbitrary decision juncture reached at time t during a certain team's work. When a team member is considering what choice to commit to, they may take into account the state of the world as well as any decisions made up to time t. Generally, the decision they ultimately make may depend greatly upon these factors. In this work, we consider the world state to include all relevant information to a decision, including the mental state information of partners. In other words, a given state n is made up of two parts, E, the physical environment variables, and I, the mental state, plans, and intents of other agents in the space. With this information, we can claim that this problem obeys the **Markov Property**.

Definition 2.1 (Markov Property)**.** The transition probabilities of future states depend only upon the present state.

How we consider choices to be made based upon the world state can depend greatly upon theory of mind. In a world without free will, we can consider a human's choice to always be fully determined by the state of the world. In a more relaxed model, even with absolute knowledge of the world, a partner's decision may still be stochastic to some degree. However, whether we consider the decision making process to be deterministic or fundamentally stochastic, in most contexts the result is the same. Parts of the state, most notably the information contained in E, may be observable. At the same time, other parts of the state, including the information contained in I, are not directly unobservable. Since our agent may not have full information, in order to predict what a partner might do, it must maintain a belief state over the current state of the world. In that regard, we can frame the decision problem as a **partially observed Markov decision process (POMDP)**.

Definition 2.2 (Partially Observed Markov Decision Process)**.** POMDPs are a tuple $(N, A, \mathcal{T}, \mathcal{R}, \Omega, O, \gamma)$ where

- N represents the set of states in the decision problem
- A represents the set of actions in the decision problem
- $\mathcal{T}$ represents the transition probability between states given actions: $\mathcal{T}(n_f|n_i, a)$ for the transition from states n_i to n_f given action a.
- $\mathcal{R}$ represents a reward function given a state and an action, $N \times A \to \mathbb{R}$
- Ω represents the set of observations
- O represents the probability of an observation, $O(o|n, a)$ for observation o given state n and action a
- $\gamma \in [0, 1]$ represents the discount factor on future rewards

If we consider our fire fighting problem in the absence of queries, we can easily map components to the POMDP formulation. In this context, N would represent the state space of everything relevant to fighting the fire, A represents the task assignment decisions that our coordination assistant can make, $\mathcal{T}$ represents the probability of developments in the fire space considering the current world state and our decisions so far. Crucially, at each juncture the agent receives an observation $o \in \Omega$, encompassing the knowledge that our coordination assistant has available to it when it makes its decision. This can be knowledge about a fire team's status, the status of the fire, weather forecasts, and so on. O represents the probability that each piece of relevant information reaches the assistant. In the absence of queries, there may be many pieces of information, such as a partner's intents, which the agent has no chance of accessing, making their probability of observation 0. The reward function $\mathcal{R}$ can provide an indicator of how well our team performs on its fire containment duties.

The primary change to this problem when we add queries is an expansion to the action space A. For the purposes of this work, queries are special actions which do not affect the state of the world. In other words, when we are in state n and take querying action a_q, the transition function maps back to n: $T(n|n, a_q) = 1$. Instead of altering the state, queries provide additional observations which allow the agent to update its belief state. We can now broadly define a query as follows:

Definition 2.3 (Query)**.** A query is an action which provides observations and has no impact on the state of the world. These observations allow the agent to update its belief state.

The observations obtained from queries will allow the agent to make better informed decisions, and thereby improve the likelihood of achieving a desirable outcome from the POMDP. Our ultimate objective is to develop a method to select the optimal policy for utilizing queries as part of a collaborative task. We have seen that this task can be framed as a POMDP. Further, we have shown that queries can be viewed as a special class of actions within this POMDP. The output of our solution should then be a feasible and optimal policy for a POMDP including these querying actions. The

correctness of this policy entails simply that the mapping of belief states to actions is allowable, meaning that for state $n \in N$, if the policy dictates that we take action a, then a is an allowable action from state n. The optimality of this problem is simply optimality in the traditional POMDP sense. In other words. a policy π is optimal if the expected outcome of the reward function $\mathbb{E}[\mathcal{R}(\pi)]$ is maximized.

2.3.2 The Focus of this Work

So far, we have identified an impactful scenario for the use of autonomous task executives, and from that scenario's challenges extracted traits we would like our executive to have. We then demonstrated how the components of the problem naturally comprise a POMDP. Our ultimate goal is to create an executive capable of assisting in our fire fighting case study by advancing the robust execution capabilities of a team coordination assistant. The MERS lab has already done extensive work on executives that operate robustly with humans by thinking ahead to predict the human's actions and react accordingly. This thesis will extend the capability of the executives developed in the MERS lab by adding in an active estimation component in the form of queries. These querying actions will advance the ultimate goal of robust execution by giving our system the power to reduce uncertainty in a partially observed environment, and thereby better select its actions. With all this in mind, we can formally define the query problem, which will be addressed in the remainder of this work.

2.3.3 Problem Specification

We now formally define the problem we are trying to solve. This problem takes the form of a POMDP and concerns a team comprised of at least two agents executing tasks. The goal of the team is to succeed in their task, meaning there is a set of terminal states $S \subset N$ that are considered to be success states. The reward function $\mathcal{R}$ of the POMDP can then be defined as follows: $\mathcal{R}(n) = 1 \forall n \in S$, $\mathcal{R}(n) = 0 \forall n \notin S$. In pursuance of that goal, the agent is able to execute a set of actions A. A subset of those actions, A_q, represents queries. These actions provide observations, but do

not affect the state. The compliment of A_q is referred to as C, and comprises the set of choices that affect the world available to the agent as it completes its task. Uncertainty in the task arises from either a specific partner's unknown intent, or due to information a partner has that has not been shared with the team. Resolving these sources of uncertainty is vital to the team's success.

On top of the POMDP we add additional constraints. The first is that the number of querying actions may be limited in number, with the limit given by n_q. The second is that we desire the plan to have a chance of team success above some chance constraint c. The final constraint is that the POMDP be acyclic, meaning that for all $n \in N$, there is no possible sequence of transitions to go from n to itself.

To summarize, we can define the Query Problem as follows:

Definition 2.4 (Query Problem)**.** The Query Problem is a special instance of a POMDP, $(N, A, \mathcal{T}, \mathcal{R}, \Omega, O, \gamma)$, where

- N represents the set of states in the decision problem. Each state includes both physical information about the environment and information about partners' mental states
  - S is a subset of N, comprising terminal states that are considered to be successful
- A represents the set of actions in the decision problem
  - A_q is a subset of A comprising querying actions. These actions provide observations but do not affect the state
  - C, the complement of A_q, comprises controllable choices in the agent's task. These do affect the world state
- $\mathcal{T}$ represents the transition probability between states given actions: $\mathcal{T}(n_f|n_i, a)$ for the transition from n_i to n_f given action a.
- $\mathcal{R}$ represents a reward function given a state and an action, $N \times A \rightarrow \mathbb{R}$.

- $\mathcal{R}$ returns 1 for $n \in S$ and 0 otherwise

- Ω represents the set of observations
- O represents the probability of an observation, $O(o|n, a)$ for observation o given state n and action a
- $\gamma = 1$ representing no discount on future rewards

We add the following additional constraints to the POMDP: a limit on the number of querying actions, n_q, and a chance constraint on success, c. The output of the query problem comprises a correct and optimal execution policy on this POMDP.

2.3.4 Example Problem Specification

We finish this chapter with a detailed example of the problem specification we have outlined. For this example, we will consider a simple search and rescue operation.

Consider a situation in which a robot and a human are searching for a person in need of rescue within T time. Let there be k different locations where this person could be. Each state $n \in N$ of this problem contains four pieces of information. The first is the current time t. The second is the locations visited up to time t by either the robot or the partner, L_r and L_h respectively. The third is the order of locations the human partner plans to visit, π_h. The fourth piece of information is the location of the target, l_t. Only the time t and the past locations visited, $L_r \cup L_h$, are directly observable. Thus our belief state is primarily concerned with our partner's plans and the location of the target. If we have found the target by time T, then we get reward 1. Otherwise, we get reward 0. The set S then represents all states such that $t < T$, and $l_t \in L_r \cup L_h$. Note that due to the monotonically increasing nature of time, this process is naturally acyclic.

The choices C available to us are what location to search at each time t. Additionally, we can also ask our human partner a questions about which location they plan to search at a given time. These actions comprise A_q, and take no time. When we perform an action from A_q, we get observation concerning π_h. This is the only way

to get such an observation. Together, A_q and C comprise A. At each time step we choose an action. If we choose an action from C time steps forward and our partner also searches a location. $\mathcal{T}$ then encapsulates both the probability the next location we visit will contain the target, and the likelihood of our partner searching a given location. In order to craft a successful search plan, it is imperative to estimate π_h in order to craft a complementary strategy.

We put further restrictions on our policy within the POMDP. The first is that we may ask no more than ten questions of our partner. In other words, we may select an action from A_q no more than ten times. The second is that we want to find the target before time T with a 90% chance. We desire to find a policy that maximizes the success rate of the search operation subject to these constraints. If we cannot do so, we indicate failure so the operation can be replanned.

Chapter 3

Representing the Query Problem

Recall that the goal of this thesis is to develop a methodology for performing active queries to increase the robustness of team task execution by reducing uncertainty about the plan of action. In Chapter 2 we define this problem in terms of a POMDP. In this section, we begin solving this problem by formalizing representations that will be conducive to fast, optimal solutions. Our solutions will hinge on representations of two key concepts: the policy space and queries. First, we build a compact representation for the policy space of the query problem using decision graphs. Next, we define exactly how queries affect that space. Finally in the third section, we demonstrate these representations in context with an instance of our fire fighting case study. These representations lay the groundwork for the algorithmic solutions to the query problem we provide in Chapter 4.

3.1 Execution Policy as a Decision Graph

POMDPs suffer from well know state space explosion issues for large problems. By adding in querying actions, we are only exacerbating this issue. This challenge suggests that a representation that keeps the querying actions implicit will provide greater tractability than traditional POMDP approaches. In this section, we show how decision graphs can be used to meet this goal.

3.1.1 Defining the Decision Graph

Our goal in this section is to develop a representation of the policy space of a partially observed Markov decision process that will allow us to represent querying actions implicitly. We start by noting each decision in a POMDP can be broken up into two parts. The first is the *controllable* part, in which the agent makes a fully controlled choice over the available actions. The next part is the *uncontrollable* transition, in which a stochastic outcome is determined based upon the state and action. By modeling this process as a *controllable choice* followed by an *uncontrollable choice*, we can represent the Markov decision process as a decision graph. In essence, decision graphs represent how controllable and uncontrollable choices lead into each other throughout a task, as well as how those choices effect the tasks outcome. We define a decision graph as follows:

Definition 3.1 (Decision Graph)**.** A decision graph is a directed acyclic graph with the following components:

- C representing the set of controllable decision states
- U representing the set of uncontrollable decision states
- T representing the set of terminal states
- $\mathcal{S}$, a mapping of terminal states to success states, such that $\mathcal{S}(t) = 1$ if $t \in T$ is considered a successful terminal state, and $\mathcal{S}(t) = 0$ otherwise.
- $\Pi = \{\pi_u | u \in U\}$ where $\pi_u : \mathcal{O}(u) \to [0, 1]$ for $u \in U$ is a transition function representing the probability of transitioning to a successor controllable decision state c from an uncontrollable decision state u.

Here we use $\mathcal{O}(u)$ (and similarly for $\mathcal{O}(c)$) to represent the set of potential outcomes for u. Note that the acyclic restriction we put on the query problem in Chapter 2 ensures that the decision graph is a directed acyclic graph (DAG). In this work, we will mostly focus on the most diffusive case of a decision graph, the decision tree, in

which each node in the graph has only one parent. A snippet of this decision tree design is featured in Figure 3-1.

A query problem POMDP can be very naturally mapped onto a decision graph. Each state $n \in N$ in the POMDP corresponds to a controllable decision state $c \in C$. For each action a available at a given state in the POMDP, we make an uncontrollable decision state $u \in U$. In other words, $C \times A \rightarrow U$. For a given u created by a state c_i and action a, $\pi_u(c_f)$ then corresponds to $\mathcal{T}(c_f|c_i, a \text{ for } c_f \in C\}$. The terminal states are simply those state at which the task execution is ended. Since we focus on acyclic scenarios, each execution must end in a terminal state. We can again define the set of success states $S \subset T$ such that $S = \{t \mid \mathcal{S}(t) = 1, t \in T\}$. Note that, given our problem definition, $\mathcal{R}$ in a POMDP maps directly to $\mathcal{S}$. This allows us to represent the success rate of a state c given a policy π as the likelihood of reaching a state $s \in S$ from c by following π. An optimal policy then is simply a policy that maximizes this success rate at the root node in the graph.

Recall that our definition of queries in the context of POMDPs implies no state change (Definition 2.3). In other words, these actions A_q are disjoint from C, and therefore are not represented in the decision graph. As we shall see in the next section, we can instead represent actions in A_q by directly operating on the functions in Π, and therefore avoid having to include explicit representation for this portion of the action space.

3.1.2 Obtaining the Decision Graph

We must now lay out an important scope restriction on this work. This thesis is primarily concerned with understanding how queries can change and improve the policy for a team plan. In other words, creating and framing the plan itself is out of scope. The system developed in this work is designed to perform in conjunction with a larger executive that creates and plans these decision problems without queries. As such, we consider generating the decision graph and optimal base policy without queries to be a pre-process to the algorithms we develop in Chapter 4. To provide intuition on how these systems work, we do outline a basic method for developing the

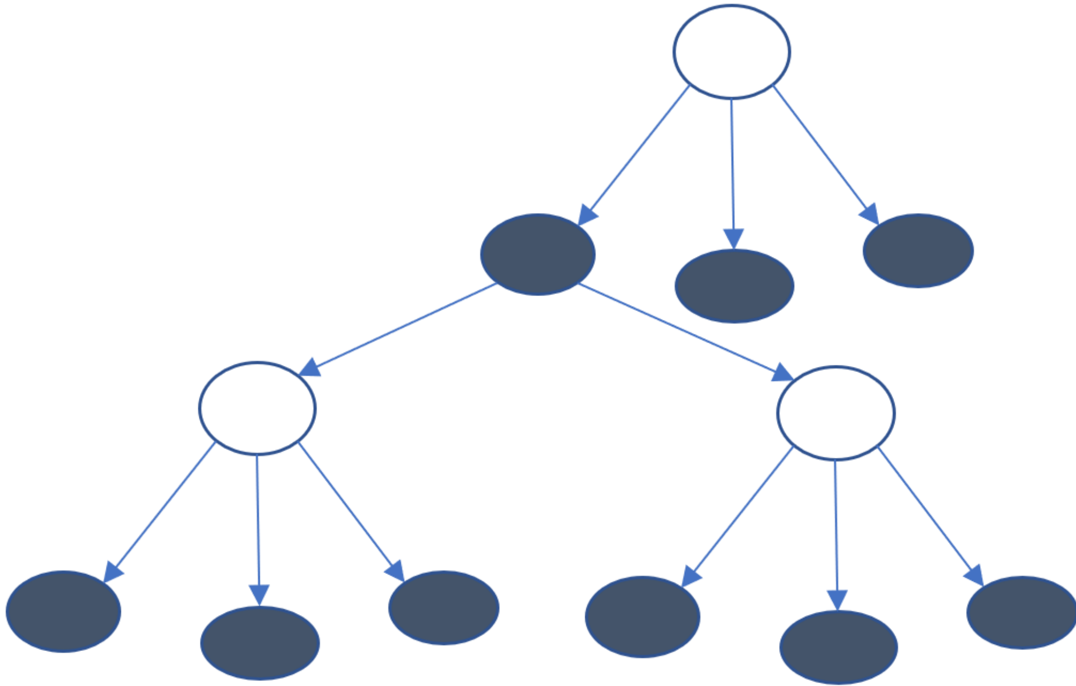

Figure 3-1: An example of one section of our decision tree representation. Here filled circles represent controllable decisions while unfilled circles represent uncontrollable decisions. The children of a given node represent possible successor decision states. Of particular note is the fact that the graph alternates layers. This is an important feature of our problem representation.

optimal base policy in the next chapter.

There are many task-planning systems that would naturally provide these prerequisites, or close approximations, for the system developed here. Any system that plans using temporal plan networks under uncertainty (TPNUs) [17], or their cousin probabilistic temporal plan networks (pTPNs) [25], is a natural choice. The system developed in this thesis is designed to complement the processes of Riker, which uses TPNUs [17]. Riker uses binary decision diagrams (BDDs) to represent the explicit graph policies in probabilistic plan space. BDDs are very compact and avoid duplicate structures within the graph. A such, a BDD as provided by Riker would be an excellent input to the querying system developed in this thesis. For more information on Riker, see Appendix A.

3.2 Query Representation

So far we have a workable representation for an agent's policy space. We must now address our key tool for manipulating that policy space: queries. Queries will allow us to gather more information in order to craft a better-informed policy. Before we can apply them however, we must carefully consider what exactly a query is in the context of a decision graph, and how it affect the policy space.

In this section, we investigate two models for queries and their effects. The first assumes that an answer to a question implies complete certainty. The second model takes a more general view. As a quick note on notation, in general we use lowercase symbols for single states and uppercase for sets of states. A notable example of this is the set S being used to represent the set of all success states, or more simply the states in which our team can be said to have correctly completed their tasks. Recall as well that we use the operator $\mathcal{O}$ to denote the outcomes of an input decision state. Our ultimate goal in the following subsections is to determine exactly how querying a specific decision affects the success rate of the team plan. In other words, we are interested in the probability of entering any state in S from a controllable choice c, given by $P(S|c)$. This value is also referred to as the *success rate* from c. We represent the augmented success rate from asking about an uncontrollable choice u as $P_u(S|c)$. This information can be used in the algorithms developed in the next chapter to find optimal questions from a robust execution perspective.

3.2.1 Deterministic Query Model

Consider some uncontrollable choice u and its set of potential outcomes $O = \mathcal{O}(u)$. Further, assume we have a belief state over a partner's intent given by π_u. In a deterministic query model, we assume the human has a fully determined plan and answers truthfully based on that plan. This means that they always give a definite answer a when asked what state they will transition to. For instance, if we ask a fire team whether they plan to rescue trapped civilians, then if they answer yes we know with certainty that is the plan they will choose. Mathematically, if we have a belief

state over the policy, π_u, and our question receives answer a, the distribution over outcomes $o \in O$ goes from $P(o|u) = \pi_u(o)$ to $P(o|u) = 1$ if $o = a$ and 0 otherwise. This process is demonstrated in Figure 3-2. With this in mind, we formally define a deterministic query as follows:

Definition 3.2 (Deterministic Query)**.** A deterministic query Q_d is defined as an operator that operates on a belief state of a partner's intent. Given an uncontrollable decision state u, a hidden decision outcome c, and a belief state over that outcome π_u, $Q_d\{\pi_u\}$ collapses π_u to c.

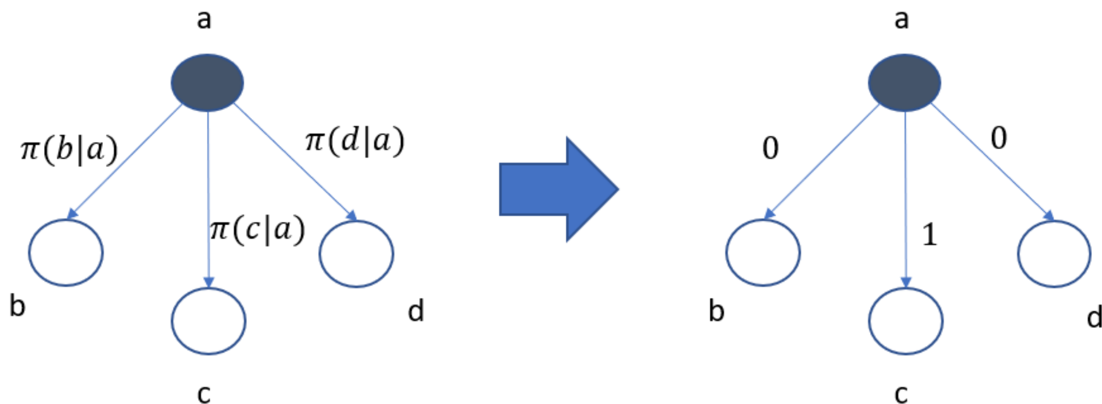

Figure 3-2: An example of how a query might affect the distribution over possibilities using a deterministic model. After querying, one outcome will have a probability of 1. Different models might sharpen the distribution differently.

Now that we have defined what a query means, we must identify how we can expect queries to be answered. We note that in the deterministic model, the probability a team gives a certain answer is the same as the probability that they choose that outcome. This is a tricky concept, but an important one. To give an example, assume based on past experience with a given fire team we know that if they learn of a trapped victim there is a 95% chance they will immediately drop everything and attempt a rescue. 95% of the time, they attempt a rescue, and their answers are always indicative of their plan, so in every such instance they say that they their plan is to rescue. Thus if we ask them what their plan is, 95% of the time they

will answer with rescue. This property allows us to use the probability of a certain uncontrollable decision outcome, $P(o|u)$, as the probability of getting that answer when asking about that decision. This relationship between potential answers and actions means that if we know the probability of team success given each potential answer, then the expected rate of success from an uncontrollable decision node is *unchanged* by questions.

Lemma 3.1. Questions have no passive effect on a plan's success.

Proof. Consider a generic plan featuring a partner policy π_p and a nominal controllable policy π_c. The success of the plan depends fully upon π_p and π_c. Since the partner's policy is considered unknown, there exists some belief state over π_p. When a question is asked, part of the belief state over π_p is collapsed to an exact representation, however π_p itself remains unaffected. As a passive application of questions, π_c remains unchanged as well. Thus the success of the plan remains unchanged. □

Lemma 3.1 presents an intuitive yet important property of querying. If we have no immediate decision to make, asking someone what they're doing doesn't have any affect on the plan. Mathematically, $P(S|u) = \sum_{o \in \mathcal{O}(u)} P(o|u) * P(S|o)$. The utility of the query, and the benefit to the task force, comes when we consider queries from the perspective of our agent's controllable choices.

We illustrate the effect of a question on the expected success rate of a parent controllable choice with an example in Figure 3-3. In this figure, consider the highlighted node in the graph to be queried. We know that the probability of getting each answer is the same as the transition probability. That means that with probability 0.5 we will obtain an answer that updates the graph to the one shown on the left. On the other hand, there is a probability of 0.5 the graph is updated to the one on the right. The expected success rate after querying is the weighted sum of the probability of success for each outcome. In this case, the expected success rate becomes 0.6 after querying the middle uncontrollable node, with an improvement over the original success rate of 0.1. Evaluating the other uncontrollable nodes in the same manner shows that the leftmost node is the optimal node to query.

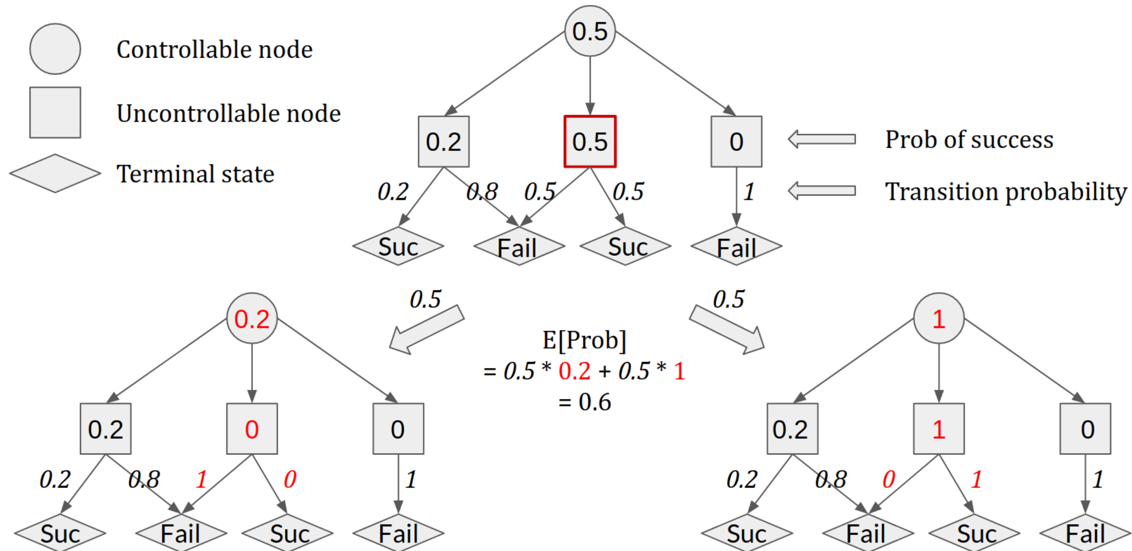

Figure 3-3: Effect a question has on expected success rate

To give an intuitive example, consider the case of a brash fire team that is prone to rush in prematurely to rescue civilians. Further, assume that they currently have a crucial containment task that *cannot* be dropped for any reason. If there are two potential homesteads we wish to scout, one near this fire team and one far away, we might consider carefully the result of that information on the team. We might ask "if we scout the home close to you and there are trapped victims, will you attempt to rescue?". If they answer 'no' then that is a *good* outcome because they keep their crucial task, so we can scout that site. If they answer 'yes' then that is a *bad* outcome, so we know we cannot scout that site and must revert to our backup site. Thus we always evaluate the effectiveness of a question in terms of a backup option. Representing the queried decision as a subscript, if the post-query success rate of a choice $P_u(S|u)$ is a maximum over all other options in U, then we make that choice. Otherwise, we pick a different state u^* with the current maximum. Since we are considering the question from a *planning* perspective, the response is always unknown and indeed may change each time the plan is executed.

We now formalize this calculation. Consider a controllable choice c with uncontrollable children $\mathcal{O}(c) \subseteq U$. For each $u \in \mathcal{O}(c)$, its set of outcomes is $\mathcal{O}(u)$. Let the node to be queried be u', with answer o received. Recall that the probability we get

answer o is the same as the transition probability from u' to o, $P(o|u')$. If the success rate from decision state o is greater than all of the remaining children in $\mathcal{O}(c)$, the execution should proceed from c to u' and to o. Otherwise, execution should proceed to the next-best child $u^* = \text{argmax}_{u \in \mathcal{O}(c), u \neq u'} P(S|u)$. Taking an expectation over all potential answers $o \in \mathcal{O}(u)$ gives us the expected success rate at c given that we ask about uncontrollable decision u, which we denote as $E[P_u(S|c)]$.

$$E[P_u(S|c)] = \sum_{o \in \mathcal{O}(u)} P(o|u) \times \max[P(S|o), P(S|u^*)] \tag{3.1}$$

Using this expectation, we can evaluate the effect of a question about an uncontrollable choice on the controllable parent. Taking the argmax of this formula over all $u \in \mathcal{O}(c)$ will give us the best query for a local decision. However, finding the best query on a plan-wide scale requires additional analysis.

While it is not a realistic reflection of every scenario, this deterministic model has two important factors that make it a centerpiece in our analysis. The first is that it does not affect the algorithmic solutions we develop in the next chapter. As we shall see, the algorithms use the model for calculations, but the model does not inform the structure of the algorithms themselves. The second factor is that the deterministic model provides a simple and elegant view of the problem. It makes the calculations our algorithms do much simpler, and therefore makes illustrating, understanding, and analyzing our solutions significantly easier. The generalized question model is quite complicated mathematically and less elegant, so we utilize the deterministic model for the majority of this work. For completeness we still investigate the general model, and appreciate its complexities, in the next subsection.

3.2.2 General Query Model

We now briefly go over how the query representation changes given a generalized model of how a question sharpens the distribution over the potential outcomes of a choice. We begin by updating Definition 3.2 for the generalized case.

Definition 3.3 (Generalized Query)**.** A generalized query Q is defined as an operator

that operates on a belief state of a partner's intent. Given an uncontrollable decision state u, a partner's unknown stochastic intent at that state π_u, and a belief state over that intent π'_u, $Q\{\pi'_u\}$ alters π'_u to more accurately reflect π_u.

This definition assumes that we have some given model of how questions affect the distribution over human choices. One example of such a model would be when a human gives a certain answer, they have an 80% chance of sticking with that choice and an equal probability of switching to any other option. In the generalized case, the distribution may be sharpened differently based on which answer is given, so we use Bayesian notation to indicate the effect of the answer on the outcome. For example, $Q\{\pi_b(a)|c\}$ is read as "the new likelihood of visiting state a given we are in state b and given that querying state b returned answer c."

We must now evaluate how this change in question model affects Equation 3.1. We are still taking an expectation, so we must still sum over all possible answers that are given to the query. Unfortunately, the probability of each answer being given is no longer necessarily $P(o|u)$ as it was in the deterministic case. Let the probability a human gives a particular answer a to a question be $\mathbb{P}(a|u)$. Note the distinction between giving a certain answer, and making a certain choice. The way human responses are modeled can have an immense impact on this answer distribution. For example, consider the scenario of asking your teenage child what they plan to do after dinner. If the options are study, watch TV, or sneak out, they will likely never say that their plan is to sneak out, making $\mathbb{P}(sneakout) = 0$. Unfortunately, this does not mean that they won't sneak out, so we may not be able to conclude that $P(sneakout) = 0$. Further, if they are modeled as sneaking out 10% of the time no matter what they answer they give, then we can conclude that $Q\{\pi(sneakout)|study\} = .1$. To compensate for this discrepancy in adapting Equation 3.1, we need some way of calculating $\mathbb{P}$ in order to build our expectation.

Suppose that there is a certain true action that the human is going to take, regardless of what their answer is. From Lemma 3.1, we know that the overall probability that they will take a certain action is immutable: $P(o|u)$. Thus we can use this value as the distribution over true actions. In order to adapt Equation 3.1, we need to know

the probability the human will give each response, $\mathbb{P}(a|u)$. We know that answers do not change the total distribution of outcomes. Thus we can say that the sum over the probability of each answer, multiplied by the probability of a particular outcome given that answer, is still equal to the original distribution of outcomes. Mathematically,

$$\sum_{a \in \mathcal{O}(u)} \mathbb{P}(a|u) \times Q\{\pi_u(o)|a\} = P(o|u) \; \forall \, o \in \mathcal{O}(u) \tag{3.2}$$

Since a and o both draw from the same set, this gives us a fully determined system of equations that we can solve for each $\mathbb{P}(a)$. Unless the response model provides an especially neat property that allows us to circumvent such calculations, $\mathbb{P}(a|u)$ must be calculated or provided explicitly.

Now that we have a way of calculating the probability of getting a given answer, we can adapt Equation 3.1. Say we are considering a potential answer a. The answer a could be given regardless of the option that the human will actually take, so we must use $\mathbb{P}(a|u)$. We then must consider each of the potential options that the human can take after having given that answer with $Q\{\pi(o|u)|a\}$. Finally, we must consider the outcome over all potential answers. The final equation then is

$$E[P_u(S|c)] = \sum_{a \in \mathcal{O}(u)} \mathbb{P}(a|u) \sum_{o \in \mathcal{O}(u)} Q\{\pi_u(o)|a\} \times max[P(S|o), P(S|u^*)] \tag{3.3}$$

As this equation shows, the general question model is not nearly as neat as the deterministic model. Thus for this work we focus on the fully deterministic model. Throughout the remainder of this thesis, keep in mind that switching between the models would be straightforward, though it may make the results much less compact or affect computational complexity.

3.2.3 Multi-Query Representation

So far we have built up a representation of how queries can affect the state of our problem, however we have yet to address how queries affect *each other*. In the pre-

vious two sections, we modeled a single query on an uncontrollable choice from the perspective of a controllable parent. However, since each controllable node has multiple uncontrollable children, there may be situations when you want to ask questions about several of those children. In this case, it takes additional Bayesian inference to understand the expected effects. Unfortunately, this analysis ends up being quite complex and does not provide any significant insight into the problem. For those still interested in a multi-query case, we perform the analysis in Appendix B, though it is not necessary for understanding the remainder of this work.

3.3 Case Study Revisited

We now have an implicit state representation that we can use for solving our query problem. Before we continue on to solution methods, it is worthwhile to go through the exercise of applying our representation to the firefighting case study.

3.3.1 The Problem

We now consider a specific instance of our forest firefighting scenario. For this scenario, assume firefighting operations are going as planned when two unexpected developments occur at once: a sudden gust of wind causes the blaze to hop a fire line and ignite a stand of trees, and a nearby homestead in the fire zone starts showing signs of life. These two developments force our command team to quickly re-plan and reassign their resources. The goals of this instance are to 1) contain the fire, 2) rescue the trapped civilians and 3) protect the first responders. If we cannot do all of these tasks with a 95% certainty of success, we are to pull back the fire teams and focus all resources on containment. We have five different teams to work with: three fire teams (a senior, a junior, and a trainee team), and two medical teams (senior and junior). To complete the mission, two containment sites must be manned by one fire team each, and one fire team must be dispatched with a medical team to save the civilians. A scouting team on site notes that one containment zone is in severe condition and one is in moderate condition. It is our assistant's job to decide who

goes where. Complicating the situation is that the resource/readiness level of each of the fire and medical teams is unknown. We can resolve the uncertainty with respect to the teams' statuses by asking them to do an equipment check. The fire spreads quickly, so time is short, and we must balance the need for certainty with the time it takes to resolve each query. A compact representation of this problem is shown in Figure 3-4.

3.3.2 The Representation

To frame the policy space of this problem as a decision tree, we must first identify what are the "decisions" in this scenario. From the description, the coordinating assistant has five decisions to make, each decision corresponding to an assignment for one of the fire teams or medicals teams. In addition, there are five sources of uncertainty which we can model as uncontrollable decision. These sources of uncertainty are the readiness level of each of the teams, with two possibilities: high or low. To build the tree, we need an ordering on these choices. In this problem time is not a factor, so the total ordering on decisions is arbitrary. We'll define the order of our controllable choices as going most to least senior fire teams, then medical teams. For our uncontrollable choices, we'll put the choice representing preparedness level of each team right after that team's assignment. With this ordering, we can build the decision tree. All that remains is to fill in the uncontrollable transition distributions. These distributions from the uncontrollable choices would be either applied from background information or defaulted to uniform. Finally, before we can solve this problem we need to settle on a model for the queries. In continuing with our earlier assessment, the deterministic query model is the most illustrative, so we will use that. we now have everything we need to begin solving this problem.

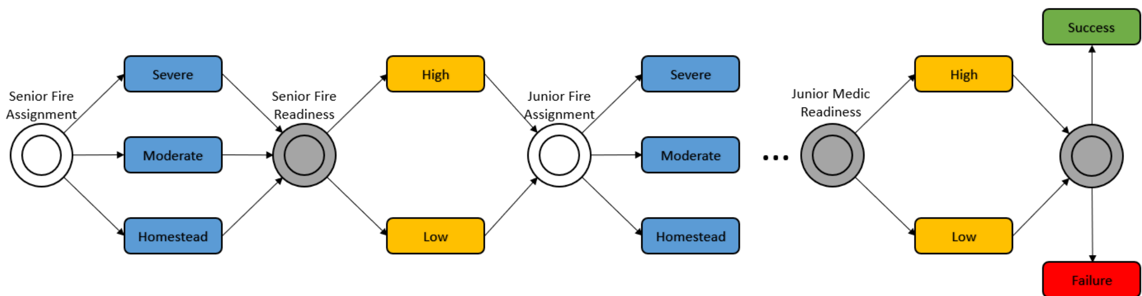

Figure 3-4: A compact representation of a fire fighting task assignment problem. Here controllable decisions are represented with white concentric circles while uncontrollable decisions are grey. The final transition to either a failure or success state is uncontrollable, and depends on both the controllable and uncontrollable choices made throughout execution.

Chapter 4

Solving the Query Problem

In the previous chapter, we used decision graphs to frame the policy space of the query problem. Additionally, we modeled how our main tool, questions, can be used to influence a team's path through that policy space. In this chapter, we use those foundations to develop algorithmic processes for selecting the best query during a complex team task.

4.1 Representative Example

In order to facilitate understanding of the algorithms, we present a representative example of a querying problem. At the end of each section, we walkthrough how the algorithm would work on this particular scenario. This example is shown in Figure 4-2.

In this example, we have a representation of a collaborative task with four choices: two controllable and two uncontrollable. For image clarity, rather than display the transition probabilities for the terminal uncontrollable nodes, we simply list their success rate in their node. The transition probability to the success state is then simply the success rate, while the transition probability to the failure state is the complement of that probability. In this example, we have not yet found the base optimal execution policy, meaning we do not know the optimal choice or success rate at any controllable node, or the success rate at non-terminal uncontrollable nodes.

Our first task will be to demonstrate how this policy can be found.

4.2 Compiling an Optimal Base Policy

Generally, for the system we are developing, we assume the base policy without queries to be pre-computed. Our strategy will then be to change the policy space through questions and perform quick repairs to the base policy to find the new optimum. Though it is not a focus of this work, understanding how such a base policy can be created will elucidate how our algorithms interact with the policy space.

Decision graphs provide a well-known method for computing the optimal policy without queries. We can use this approach, known as the average-out-and-fold-back method, for the base-case solution to the decision problem without questions [21]. This method starts by considering the terminal uncontrollable choices in plan execution. In each case, we have an uncontrollable decision node followed by terminal states that are either success states or failure states. Since uncontrollable policies are stochastic with known distributions, and since the successor states are binary success/failure, we can use a simple expectation to calculate the chance of success at each terminal uncontrollable choice.

At the next level up in the tree, the parents of those uncontrollable decisions are controllable. Since we know the success rate at their children, we can optimize their policy by greedily picking the best child. Going up one level, we are once again at uncontrollable nodes and can now determine their success rate by taking a weighted average of the controllable children. We continue in this way, computing expectations and then selecting the best among these expectations as we back propagate up the graph until we reach the root node. At that point we have the un-queried optimal probability of success at each node in the graph. This is the essence of the average-out-and-fold-back method of computing optimal policies in decision graphs [21]. Any executive performing this plan could simply make its choices greedily to achieve the highest likelihood of succeeding. By understanding how questions affect local transition functions, we can build clever ways of computing the optimal policy including

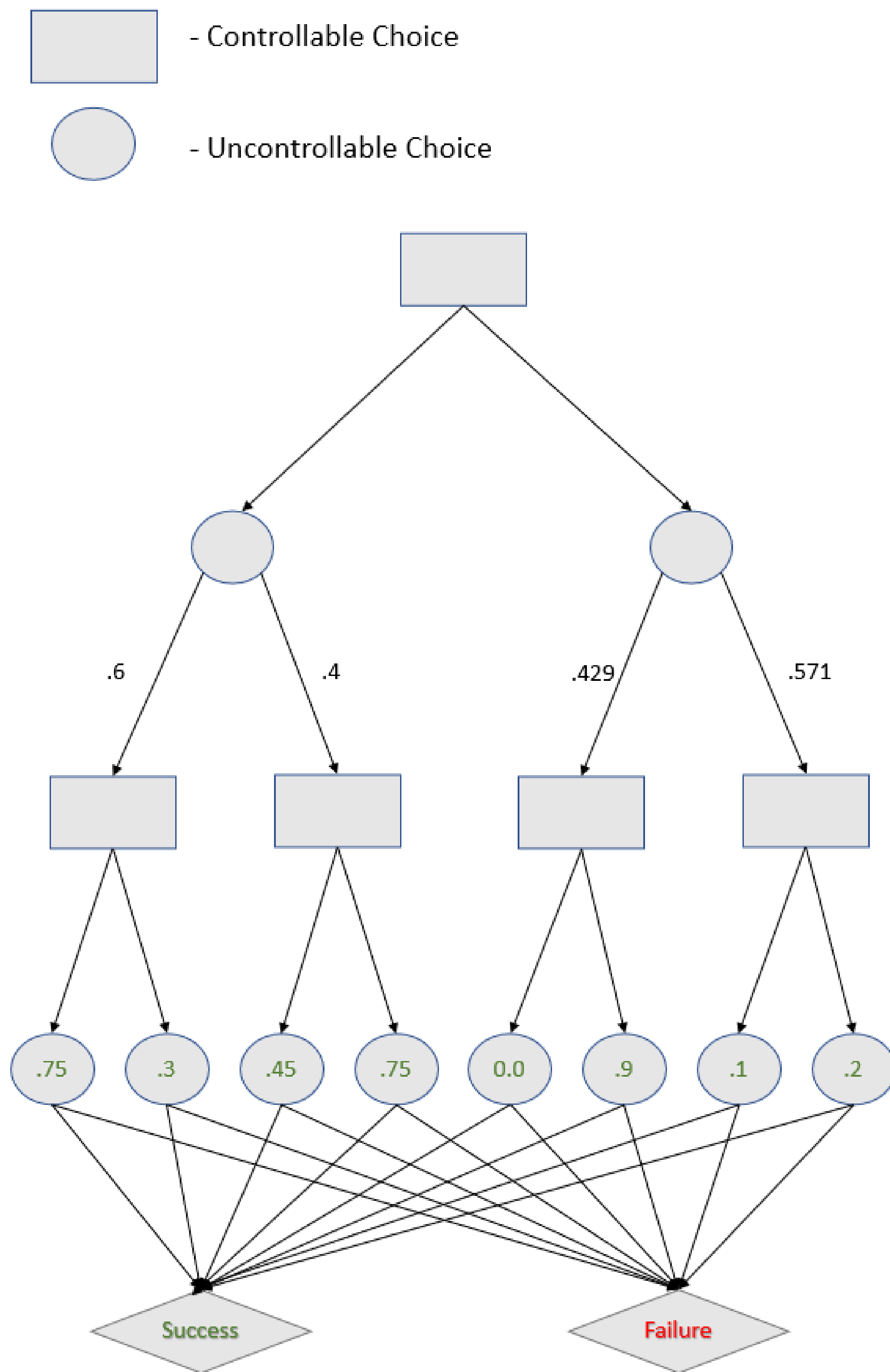

Figure 4-1: A representation of a collaborative task with four choices. For image clarity, rather than display the transition probabilities for the terminal uncontrollable nodes, we simply list their success rate in their node.

queries.

In 4-2 we run a simple example of this process on the representative scenario we introduced earlier. The success rate at the terminal uncontrollable choices are predetermined by simply taking the average over outcomes. The first step is to determine the policy at the terminal *controllable* choice, i.e. the parents of the terminal uncontrollable choices. These comprise the third layer in the example. Since these choices are controllable, we can simply pick the best of the direct children. Counting from the left, those are the first, fourth, sixth, and eighth node in the fourth layer. The success rate at each controllable node in the third layer is then just the success rate of the chosen child. The next step is to determine the success rate at the uncontrollable nodes in the second layer. At this point we simply take an average over the success rate at the outcomes of that choice weighted by the transition probabilities. Finally, we find the policy at the root node in the first layer by simply selecting the best child again. This child is the first node in the second layer. The success rate of the root node, .75, now matches that child. We now have the optimal policy and success rate at each node in the graph.

4.3 Pre-Execution Queries

We begin our discussion of queries with a basic scenario in which the controllable agent is only able to ask questions before execution starts. In our firefighting case study, we might apply this restriction if we are coordinating teams without any instant communication capabilities (perhaps due to bad reception), or if communication once a task begins becomes an untenable distraction. Other scenarios where this type of query would be used include: games where communication is forbidden, situations in which communication is unreliable, or cases when one team member must concentrate and cannot be disturbed. We will later show that this approach can also be leveraged for situations in which there are limited periods of communication blackouts within the plan.

Before we begin, we note that asking prior to execution provides a unique benefit

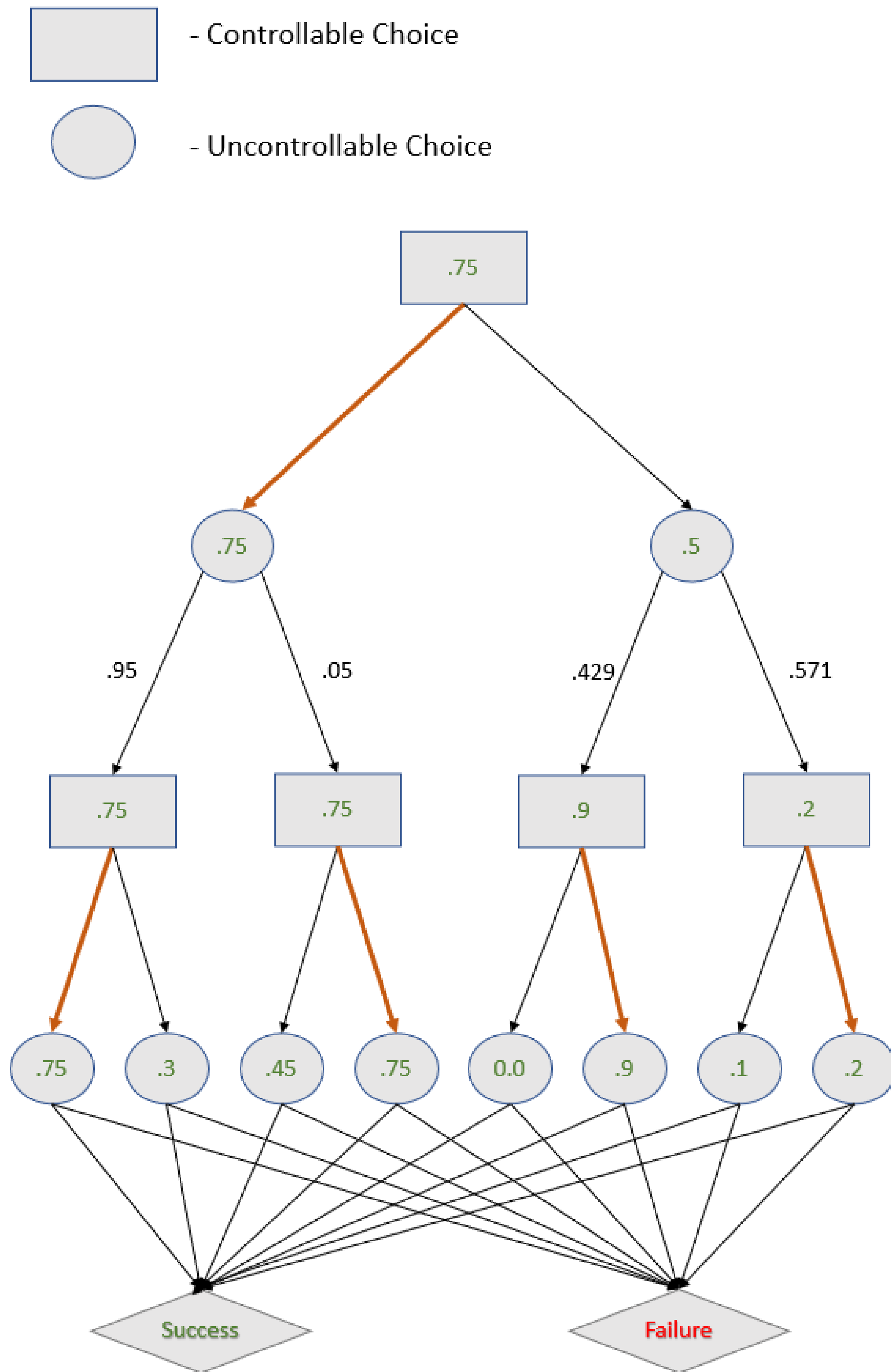

Figure 4-2: Our example scenario after computing the optimal base policy. The optimal policy at each controllable decision is highlighted in orange.

to the system. When planning questions during execution, there is no certainty about the answers we will receive. This means that all analysis must be done as expectations over all possible answers. On the other hand, if we ask prior to execution we get the answer right away, meaning we can re-evaluate our approach accordingly. The upshot is that, though we will be proceeding with our analysis in this section as though the robot only gets one question, a simple query-reevaluate-query approach will extend the capability of a one-query system to an n-query system.

4.3.1 Problem Statement

We are given a decision graph representation of the task policy space, a deterministic model of queries, and a chance constraint on task success. Further, assume we have completed the no-query optimal policy pre-processing step outlined earlier. We are allowed to resolve one source of uncertainty in the graph through use of a query. Our objective is to meet the chance constraint if possibly by outputting the query that increases the success rate of our plan the most.

4.3.2 Approach

The goal in this situation is to ask the question which optimizes the collaboration's expected chance of success. In order to find this question, we must first consider the effect questions have on the success rate at the root node, which represents the team's starting state. We will develop an approach first for the special case of decision trees, then discuss how that approach can be extended to more general decision graphs.

Query Effects on the Root

For the time being, we ignore the effect of controllable nodes in the tree and focus only on the uncontrollable nodes. We begin by considering a frontier of nodes, which we define as follows:

Definition 4.1 (Frontier of Nodes)**.** A frontier of nodes $\mathcal{F}$ of a directed acyclic graph

G is a maximal set of nodes such that no node in the set is an ancestor of another. Mathematically, $\mathcal{F}$ is a set subject to the following properties:

1. $P(a|b) = P(b|a) = 0 \ \forall \ a, b \in \mathcal{F}, a \neq b$

2. $\forall \ n \in G \ \exists \ a \in \mathcal{F}$ s.t. $P(a|n) \neq 0 \vee P(n|a) \neq 0$

A frontier describes a set of states such that execution must pass through exactly one member by the time of termination. Utilizing such a set $\mathcal{F}$, we can calculate the expected success rate at the root, $P(S|root)$, in terms of the probability of success at each $n \in \mathcal{F}$, $P(S|n)$, and the probability of reaching n, $P(n)$:

$$P(S|root) = \sum_{n \in \mathcal{F}} P(S|n)P(n) \tag{4.1}$$

Let's say we consider querying some node n in a frontier. We note that querying n does not affect the chances of reaching that state, however it may affect the chances of success from it. In order to avoid calculating the effect of a query at every potential node, we'll use the ideal outcome of a 100% post-query success rate as a heuristic and take a pruning approach. The maximum chance of success after querying is 1. $1 * P(n)$ then represents the maximum possible contribution to the success rate at the root from the queried node. We can quantify the difference between this ideal contribution and the current contribution as

$$\epsilon = (1 - P(S|n))P(n) \tag{4.2}$$

As this relation shows, ϵ represents the maximum that the success rate at the root can change after querying a given node. In the case of maximum change, $P(S|n) = 0$ and the post query success rate becomes 1. In this case, the total change is simply $P(n)$. This gives us the following lemma

Lemma 4.1. The gain in success rate from querying a node is bounded above by the probability of reaching that node

Before we can use this property, we need to consider the effect that *controllable* choices have on the $P(n)$ values and the change in success rate at the root. There are two possibilities, the current optimal policy routes through a given decision state, or it does not. In the first case, the probability of visiting a decision state is at most $P(n)$. In the second, it is 0. If we assume the information gained from a query warrants a change in the controllable choice policies so that we do route through a given state, then we still have an optimistic estimate for the way that questions will affect the root. That this estimate is optimistic is key to the heuristic search we will apply later.

To see how the success rate actually changes, consider the case of some decision state that normally has no chance of being reached during execution due to the policy of some controllable choices. Assume applying the query on that decision gives us some gain g in the success rate at that node. If the parent controllable choice does in fact pick the one we queried, then the principle discussed earlier still applies and we can continue up the tree. We can then check the parent's parent and so on, continuing in this way until we reach a choice that *did not* select the ancestor of our queried node. In this case, the success rate at the parent is not equal to the current node, but rather one of its siblings. To continue the propagation, we subtract the success rate of the chosen sibling from the success rate of the current node. This gives us the true gain g' at that parent controllable choice. We can then continue on up the chain using g' until we either reach a new state where our ancestor was passed over, in which case we repeat the process to get a g'', or we reach the root. It may be the case that g' is negative. This would correspond to the scenario where the gain in success rate at a state from querying do not warrant a change in policy further up the tree that would allow us to reach that state. In this case, the gain at the root is 0 and we can stop. We detail this process in Algorithm 1. Since we are simply proceeding up the tree, the run time for this algorithm is $O(d)$ for a tree of maximum depth d.

Algorithm 1 gain_at_root

```
1: Input: Node node, Gain g
2: while node != Root do
3:     if node.controllable then
4:         g = g * P(node | node.parent)
5:     else
6:         if node.parent.selected_child != node then
7:             g = (g + node.SR) - node.parent.selected_child.SR
8:     node = node.parent
9: Return: g
```

Picking the Best Query

Recall that we are representing the policy with a decision tree, meaning there are no converging paths. Scenarios that would obey this restriction include instances where collaborators act on a distinct set of variables. Our firefighting case study is an example of this, since teams are assigned sub plans or sub objectives that only need to be done once. In this case, $P(n)$ can only decrease as we descend the tree, meaning that we can use this value in conjunction with Lemma 4.1 for state space pruning in our search. Given some difference δ between our original chance of success and our desired chance of success (or best yet found if that value has been exceeded), we will use $P(n)$ to guide our search for the optimal question that meets the chance constraint.

To give a grounded example of how this search will work, we refer back to our firefighting scenario. In this example, the assistant is trying to pick a question to ask and is considering posing that question to a senior fire team. First it considers asking the team to give a report to determine their preparedness level. The assistant knows that the senior team is quite effective and well managed, so the chance of them having a low preparedness level is small. Because it is so small, the assistant determines that there are other more pressing sources of uncertainty that take precedence. Since the chances of the senior team being unprepared are so low, the assistant knows that it doesn't have to consider any queries that are contingent on them being unprepared, such as the query "What is your plan if you have an equipment malfunction in a

burning homestead?". Thus the entire branch of the tree contingent on the senior team being unprepared can be pruned from consideration. This pruning is the essence of what this approach will accomplish. This process run on a tree is detailed in Algorithm 2. For multiple queries Algorithm 2 can be repeated after each answer is received and the tree is updated accordingly.

Algorithm 2 pre_execution

```
 1: Input: Node ← Controllable Node, C ← Chance Constraint
 2: bestSR = baseSR = Node.SR, bestN = None
 3: δ = C - baseSR
 4: Initialize Q with Node
 5: while Q is not empty do
 6:     Node = Q.pop()
 7:     for Child in Node.Children do
 8:         gain = gain_at_root(Node, Child.queried_SR - Node.SR)
 9:         if gain > δ then
10:             bestN = Child, bestSR = baseSR + gain
11:             δ = gain
12:         for Subtree in Child.Children do
13:             if P(Subtree)(1 - Subtree.SR) > δ then
14:                 Q.push((Subtree, P(Subtree) (1 - Subtree.SR))
15: if bestN == None then
16:     Return: Failure
17: Return: bestN, bestSR
```

For non-tree directed acyclic decision graphs, the situation is a little more difficult but still manageable. Algorithm 2 requires some kind of ordering on the nodes based on $P(n)$. Trees provide a nice implicit ordering of these values, however in the absence of a tree we could build the ordering ourselves in the pre-processing stage. In that case, rather than proceed down the tree, we would simply look at the next node in the $P(n)$ order and terminate when there aren't any more that meet our δ criteria.

4.3.3 Algorithm Notes

Our search is guided by a priority queue Q. The priority level in the queue is determined based on the maximum possible change in success rate that a given query could provide in ideal conditions as given by Equation 4.2. We initialize Q by pushing

on the root node. In lines 5 - 14 we search through the priority queue for candidate questions. Each node in the queue represents a controllable decision, so the candidate queries will always be about children of the current node. We use Equation 3.1 to check the results of queries on each child. We consider this result a sub process of the algorithm held in the node member variable queried_SR. The gain for querying that uncontrollable choice is then fed into Algorithm 1 to get the gain at the root (line 8). If that gain is better than we need to exceed the chance constraint or current best success rate, we update our stored values accordingly. Otherwise, we consider the potential of the next set of controllable choices, which are the grandchildren of the current node. We evaluate the potential of these grandchildren based on the gain they would provide if a query gave them a 100% chance of success. If that gain is greater than what is necessary to beat the chance constraint or current best, we push it into the queue (line 14). Otherwise, we can prune out that node and all of its children based on the monotonically decreasing probability of reaching successive generations in the tree.

At the completion of the algorithm, we are left with the best node to query and the success rate at the root for querying that node. Since we are considering this based on an all-or-nothing metric, if we are unable to meet or exceed the chance constraint we consider the search to have failed and return nothing. This can of course be modified based on needs to always return the best query found regardless of whether or not it meets the chance constraint. We will use such a modified version of the algorithm later when we consider queries in systems with communication dead zones.

In terms of run time we visit each node exactly once, and upon each visit we calculate the effects of a query on the root note. The query check takes $O(b)$ for a tree with branch factor b, and the root check takes $O(d)$ for a tree of depth d. In total this algorithm is $O(nbd)$ for n nodes. In trees, this is $O(b^d d)$, but since we are using a pruning approach the average run-time is much better. The tractability of this approach is addressed in the discussion section.

4.3.4 Example Walkthrough

We now walkthrough an example of how this algorithm would work on the representative scenario from Figures 4-1 and 4-2. Some details of this computation are shown in Figure 4-3.

To begin, we must order the controllable nodes in the plan by their likelihood of being reached. There are five controllable nodes, one in the first layer (the root) and four in the third layer. The root node has a 100% chance of being visited, so that goes first in the order. The rest of the nodes are ranked by their likelihood of being visited should we try to reach them. Note we can reach either of the nodes in layer two with probability 1 by adjusting the policy at the root node. This means the probability of reaching the nodes in layer three is simply the transition probability from their parent. Thus we can use this value to order the remainder of the controllable nodes. In Figure 4-3, the nodes are labeled in orange with their position in the order.

The next step in the algorithm is to step through the nodes in the established order, pick the best child to query, then propagate that change to the root. Recall that the utility of querying a child is given by Equation 3.1. We start at the root node. Asking about the left child gives 0 benefit, since both children have success rate greater than the alternative (the right child). Asking about the right child raises the success rate of the root by approximately .064. Since the fifth node in the order has a likelihood of being reached of .05, we can now prune that node from consideration. Next we visit the second controllable node in the order. By a property of 3.1, since its children are both terminal uncontrollable nodes, it does not matter which one we ask about. Either way, the gain at this node is .075. We can propagate this gain to the root by multiplying by the probability of reaching this node. The eventual gain at the root is .07125, the new best. Next we visit node three. Performing the same calculation gives a gain of .08. When propagated to the root, this is worse than the current best, so we move on. At node four, there is no benefit in asking about the children since one of the children is guaranteed to fail. We have pruned node five, so we can finish execution. The optimal node to query is either child of node two, and

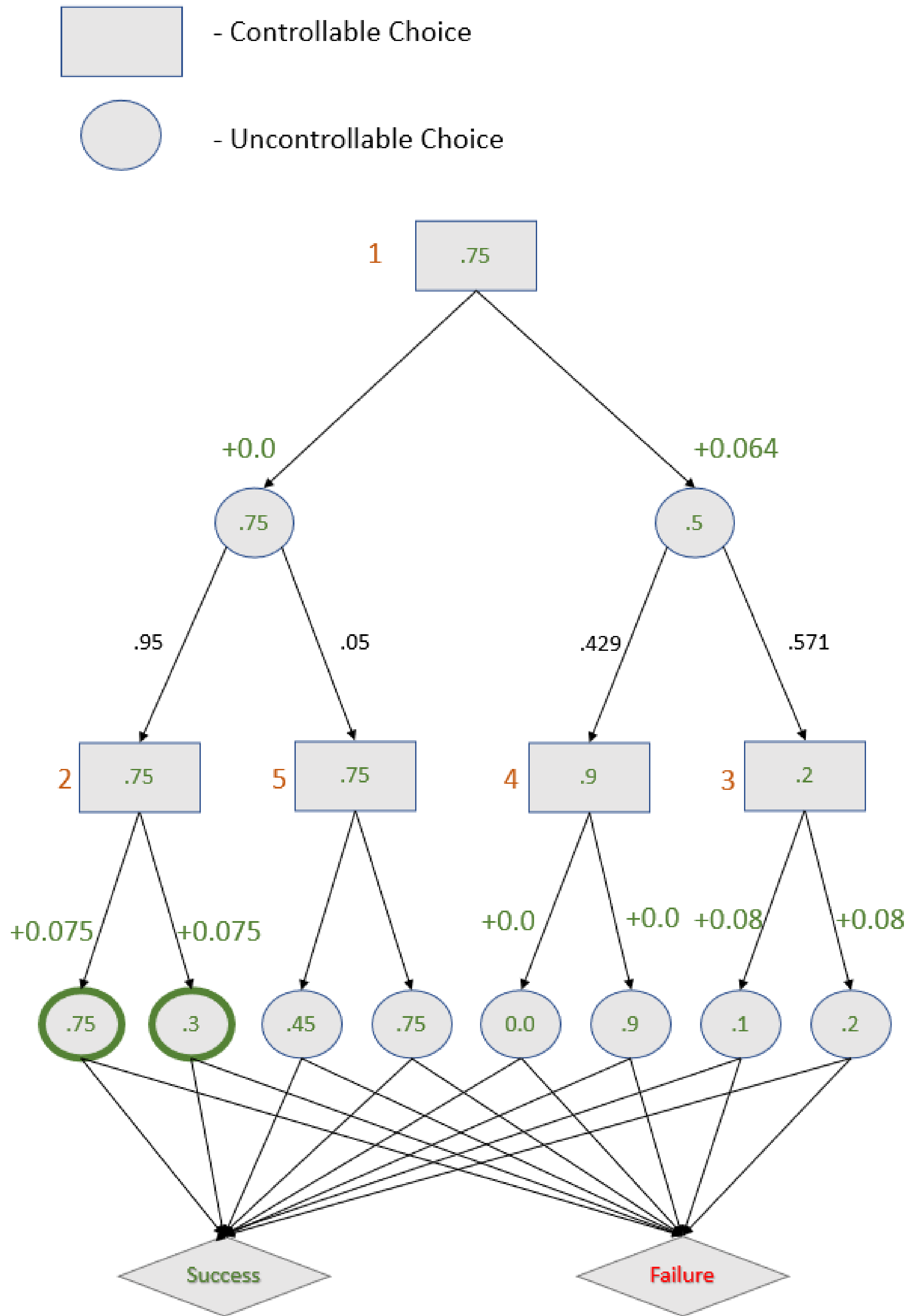

Figure 4-3: The example scenario annotated with details of the pre-execution algorithm. The ranked order of the nodes is shown in orange. The gain at the parent for asking about uncontrollable nodes is shown in green. The optimal nodes to query are outlined in green. Both are equally optimal.

the new success rate is .82125.

4.3.5 Optimality

We now prove that Algorithm 2 always returns the optimal question. We perform this proof by contradiction. Assume Algorithm 2 returns some node n from the decision tree that is *not* the optimal choice for querying. Let querying that node create a gain in success rate of δ at the root node. Now consider the true optimal node, n^*. There are two possibilities, either n^* was considered before we returned n, or it was not. In the first case, we will have used Algorithm 1 to determine precisely the gain at the root from querying n^*: δ^*. Since n^* is optimal, it must be the case that $\delta^* > \delta$. But since δ^* is greater and we always return the node with the greatest gain found, the algorithm cannot have returned n over n^*. Thus n^* must not have been considered. Since n^* was never considered, it must be the case that $P(n^*)(1-n^*.SR) <= \delta$. The value represented by $P(n^*)(1-n^*.SR)$ is the gain at the root in the ideal scenario that the post-query success rate at n^* is 1 and the pre-query plan routes through n^*. In other words, $\delta^* <= P(n^*)(1-n^*.SR)$. By the transitive property, this means that $\delta^* <= \delta$, which violates our original assumption that n^* is optimal. Thus we have reached a contradiction, so we can conclude that Algorithm 2 returns the optimal query node.

4.4 Single Query Policy Development

While it does have its applications, the use of pre-execution queries is extremely limiting. We will now relax our timing restriction, so that the agent can ask a single question at any point of the collaborative process. Note how powerful this relaxation is; if we wait until the last possible moment to ask a question, we can query a choice juncture so long as we haven't already queried an ancestor. A frontier of descendants then represents a complete set of events which we can plan to query depending upon how execution progresses. Since our success rate is a matter of expected value, and we can expect to ask each question if execution leads to that point, we can consider the

effects of ALL possible questions in a frontier when calculating our queried success rate. To make this more clear, we refer back to Equation 4.1. If we ask a question about a node n, then we can improve the value of n's parent c. But since we are asking at the last moment, we can assume just before we reach event c, we always have our question available to ask. Thus if we are to reach node c, we can *always* ask about node n, meaning we can use the updated value of all $P(S|n)$ for $n \in \mathcal{F}$. Our goal then is to select the optimal frontier of nodes for which we can plan our single query.

4.4.1 Problem Statement

We are given a decision graph representation of the task policy space, a deterministic model of queries, and a chance constraint on task success. Further, assume we have completed the no-query optimal policy pre-processing step outlined in the previous chapter. We are allowed to pick a frontier of nodes that we can apply queries to. Our objective is to meet the chance constraint if possible by outputting the frontier that increases the success rate of our plan by the most.

4.4.2 Approach

We can solve this problem compactly with a recursive approach. Consider the base case, where we have a controllable decision followed by an uncontrollable decision followed by execution termination. The optimal policy is to simply pick the best query from the controllable node's uncontrollable children. Now we take a step back and consider the next controllable ancestor of our current node. At this point in execution, we know that there are two options: ask a question now or ask a question later. If we ask a question now, we consider the children of this node for querying. If we ask a question later, then the natural choice is to pick the optimal questions for *each* descendent subgraph of our current node. If we perform our search recursively, then we should have the optimal subgraph questions and effects in hand. Thus we need only compare against asking the optimal immediate question to find the optimal

policy for the current subgraph. We can then continue our recursion until we reach the root node. The full procedure is detailed explicitly in Algorithm 3.

Algorithm 3 single_scheduled

```
1:  Input: Node ← Controllable Node, C ← Chance Constraint
2:  if Node.visited then
3:      Return: Node.bestSR, Node.bestQ
4:  if Node.terminal then
5:      Return: queried_SR(Node)
6:  bestSR, bestQ = queried_SR(Node)
7:  for child in Node.children do
8:      subtreeSRs, subtreeQs = [], []
9:      child_newSR = 0
10:     for subtree in child.children do
11:         newSR, newQ = single_scheduled(subtree)
12:         subtreeSRs.append(newSR)
13:         subtreeQs.append(newQ)
14:         child_newSR += subtree.TP * newSR
15:     bestSR, bestQ = max([(bestSR, bestQ), (child_newSR, subtreeQs)])
16: Node.visited = True
17: if Node == Root and bestSR < C then
18:     Return: None
19: Return: bestSR, bestQ
```

4.4.3 Algorithm Notes

This algorithm works recursively, with each call returning the optimal policy for a single query of any descendent below the input node. We include steps for marking nodes as visited, but that may not be necessary depending upon the structure of the particular problem. Notably, it would not be necessary on a tree.

In our algorithm, we label the grandchildren as subtrees (e.g. subtreeSR, subtreeQ), since each one becomes the root node for a recursive call. When we call the algorithm on each subtree recursively, we are returned the optimal policy for querying the nodes below that grandchild. Collecting all of these policies together in subtreeQs, we have the optimal policy for the current node if we ignore the possibility of directly querying the child of the root. We utilize the probability of transitioning to a given

subtree, held in subtree.TP, to update the new success rate of each child in line 14. In line 15, we compare the updated success rate for each child with the best one-step look ahead query, which we found in line 6. What we are left with is the optimal policy for a single question below the root node. If the algorithm determines it was called on the root of the original graph, at the end of its major processes it will check whether the final success rate meets the chance constraint. If it does, we return the policy. If it does not, we return a null value to represent failure.

In terms of run time we visit each node exactly once, and upon each visit we calculate the effects of a query and then move on to the subtrees. The query check takes $O(b)$, and the operations on each subtree are all $O(1)$, so their complexity can be included in the visit to that subtree. Thus the total complexity of this algorithm is $O(nb)$. Notably, we have a better complexity for the much more powerful scheduled case than we do for the pre-execution case! This can be attributed to the fact that the gain at the root is calculated once through recursion, rather than explicitly at each node.

4.4.4 Example Walkthrough

We now step through an example of this problem run on our demonstrative scenario. Some details of this example are highlighted in Figure 4-4.

The algorithm begins at the root. First, we check the utility of asking a question now on one of the direct children. From earlier, we know that asking about the left child gives no utility, and asking about the right child gives increases the success rate at the root by .064. The best success rate at the current node from asking now is therefore .814. These results are shown in green in the figure.

Now we recursively call the algorithm on the grandchildren of the current node, meaning that we now focus on the controllable nodes in the third layer. Again we recall the calculations from the pre-execution query example. From these same calculations, we know the gain in success rate at each controllable node from asking about the direct children. These values are shown in orange. Since they are all terminal controllable children, there is no recursive call. This means that, for *each* controllable

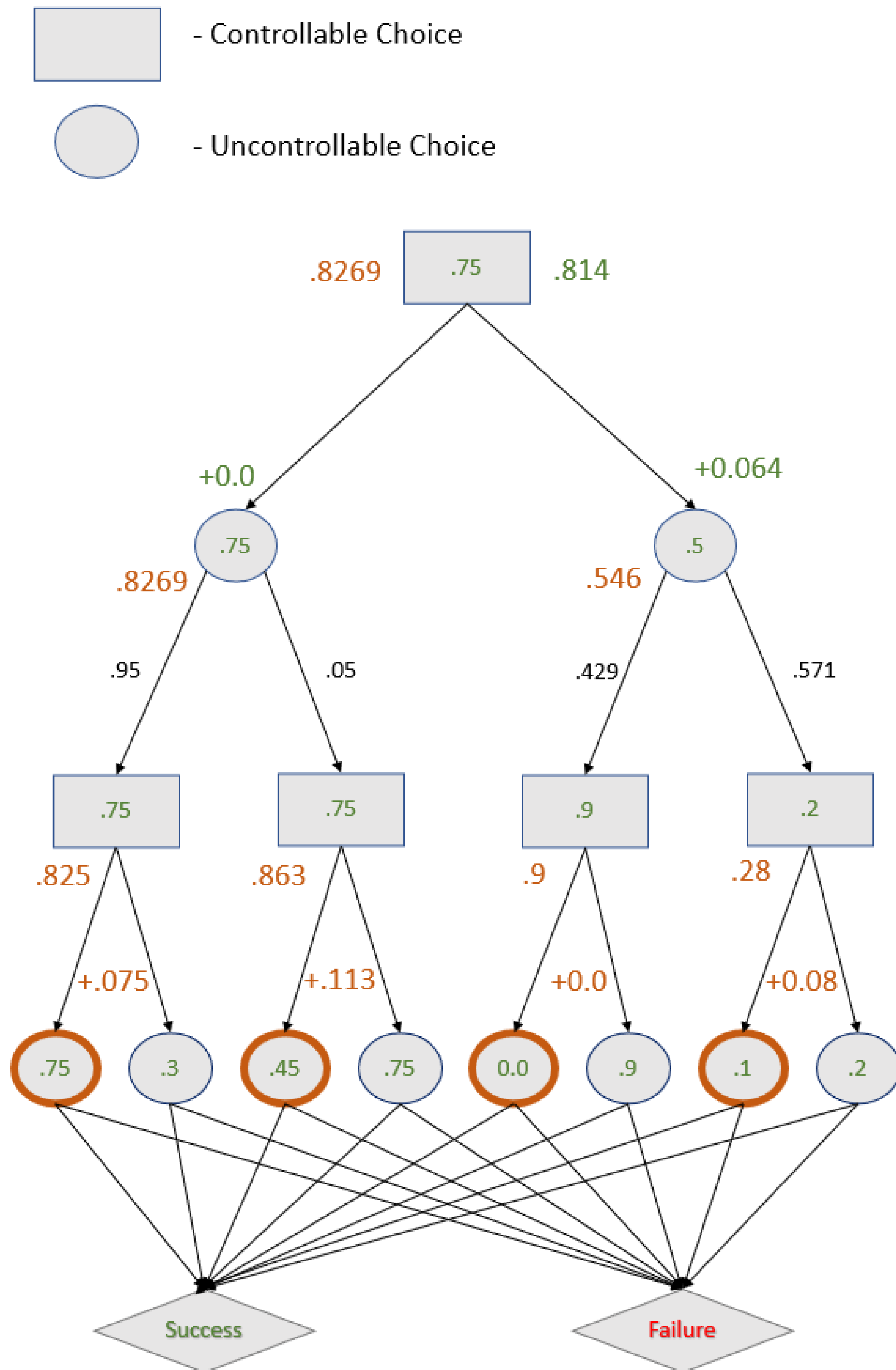

Figure 4-4: Annotations of our representative example when we run our single query algorithm. The success rate of saving the question until the third layer is shown in orange, the success rate of asking the question right away at the root is shown in green. As comparison shows, the optimal policy is to save the question for the third layer.

node, we return the success rate from asking about the best child. Note again that since there are two terminal children of each node, the math of Equation 3.1 works out such that in all cases asking about either child gives the same utility. In this case we always pick the left child.

We now know the optimal policy and success rate at the nodes in the third layer. When we return these success rates, we can calculate the success at the root if we saved all of our questions for the third layer. As before, that utility is just the max of the direct children, .8269. This success rate is shown in orange. We can now directly compare the utility of asking now (green) vs. asking later (orange). Comparison shows that asking later is more beneficial, so we save that success rate and policy and terminate execution. We now have the optimal success rate and policy for the entire graph. Note that the power of saving queries allows us to plan four question, one for each node in the third layer, rather than the one we would have been able to ask if we used our question at the root node.

4.4.5 Optimality

We will now prove that the set of queries returned by Algorithm 3 is the optimal query policy. In this case, since we are using a recursive algorithm, it makes the most sense to do a proof by induction. Before we begin, we note that uncontrollable choices are maximized by maximizing the children. So if we have the optimal policy for the controllable children of an uncontrollable choice, the optimal policy for that uncontrollable choice is implicitly the combination of the policies for those children.

Base Case

In the base case, we are dealing with a controllable choice followed by an uncontrollable choice followed by the end of execution. In this case, we can only ask questions of the direct children of the controllable node. We know that the queried_SR function finds the optimal query in such a case by direct comparison of all possibilities. Thus we are assured that in the base case, we return the optimal query.

Induction

Say we are at some arbitrary controllable choice in the graph and we have the optimal policy for all of the grand children of that choice. There are two possibilities: either the optimal policy for the current node is to ask a question now of a direct child, or wait and ask a question later. In the first case, we can easily find the optimum using queried_SR. In the second case, we want to pick a policy that gives the maximum value for the uncontrollable children of the current node. But, by induction, we already know the optimal policy of the uncontrollable children. Thus we know the optimal policy of the parent in the case that we postpone the question. We are left with the ability to directly compare the two cases and pick the better option. We now have the optimal policy at the current node. This completes the proof by induction.

4.5 Multi-Query Policy Development

When multiple queries are allowed, the problem of finding which optimal questions to ask is much more difficult than the single-question scenario, due not only to the exponentially large search space but also to the fact that questions deeper in the graph have an effect on questions higher in the graph. Fortunately, a dynamic programming approach can extend our single-query algorithm to handle arbitrarily many queries.

4.5.1 Problem Statement

We are given a decision graph representation of the task policy space, a deterministic model of queries, a chance constraint on task success, and an allowance of k questions. Further, assume we have completed the no-query optimal policy pre-processing step outlined in the previous chapter. We are allowed to pick up to k frontiers of nodes that we can apply queries to. Our objective is to meet the chance constraint, if possible, with the minimum number of optimal frontiers.

4.5.2 Approach

Consider when the execution reaches the root node c of some subgraph in the decision graph. Assuming that we have k remaining questions to allocate, then at c we have two options: ask a question now or save the question for later. If we ask a question now, then we have $k-1$ questions left for each of the child subgraphs. If we save our question, then we have k questions left for each child subgraph. The key insight that enables a dynamic programming approach is that the optimal policy and expected success rate at each child subgraph with k questions left is unaffected by previous query decisions. Thus, we can take a bottom-up approach to compute the optimal query policy. For each of these controllable decision state c with k questions left, we compute its expected success rate by recursing on c with $k-1$ questions and c's grandchildren with k questions.

Formally, we define $EP(c,k)$ as the expected success rate from controllable node c with k remaining questions. That is, $EP(c,0) = P(S|c)$ for $c \in C$. We recursively compute each $EP(c,k)$ for $k = 0, ..., q$, where q is the allowed maximum number of queries, based on $EP(c', k-1)$ and $EP(c', k)$, for $c' \in C'$, where C' is the set of c's grandchildren. This approach has two base cases. The first is $EP(c,1)$, which is computed from Algorithm 3. The second is $EP(t,k)$, where t is a terminal controllable. In this case, we assume that we only allow asking about one uncontrollable child per decision state c, and thus $EP(t,k)$ can be reduced to $EP(t,1)$ whose value is 1 if $t \in S$ or else 0. Extending the single scheduled query algorithm in this way gives us a run time of $O(qnb)$. An approach to this dynamic programming method is detailed in Algorithm 5. Since each question is optimal, this will give the minimum number of questions needed to meet the constraint.

In order to write our algorithm, we need one sub-routine of note. In the case when we are asking a question now and $n-1$ questions later, we need a targeted way to determine the new success rate at the current node based on those questions. This routine is very similar to the standard query routine, however the addition of questions in the subgraphs make for some subtle points that are highlighted well by

pseudocode. That pseudocode is shown in Algorithm 4.

Algorithm 4 enhanced_query

```
1: Input: Node ← Controllable Node, k ← Allotted Questions
2: best_SR = node.queried[k][0]
3: enhanced_children = [(child.queried[k][0], child) for child in node.children]
4: best_child_SR, best_child = max_SR(enhanced_children)
5: second_best_child, second_best_SR = second_max_SR(enhanced_children)
6: for child in node.children do
7:     backup = best_child_SR
8:     if child == best_child then
9:         backup = second_best_SR
10:    current_SR = 0
11:    for subtree in child.children do
12:        current_SR += P(grandchild|child) * MAX(subtree.queried[k][0], backup)
13:    if cur_SR >= best_SR then
14:        best_SR, selected_query = cur_SR, child
15: Return: best_SR, selected_query.queried[k][1] + selected_query
```

4.5.3 Algorithm Notes

In Algorithm 5 we include an allotment of questions in the input. An alternative approach is to only give a chance constraint and allow as many questions as it take to meet the success rate threshold. This approach uses a dynamic programming paradigm in coordination with recursion. Notably, it utilizes a double recursion. In line 4 we recurse on the input node with one less question. In line 12 we recurse on the grandchildren of the input node with the same number of questions. In this implementation, we only allow a single question for each set of children of a controllable choice. This means that for terminal nodes, the result for any allotment of questions is simply equal to a one-step look ahead case for a single question. At the conclusion of this algorithm, the policy is saved and used in previous recursive calls as necessary. The pseudocode exits upon completion of the goal without returning a value. With this structure, it would be up to a calling function to check the data generated by the dynamic program to determine success, failure, and the arrived-at minimum number of questions. The final implementation detail of note is that Algorithm 5

is only meant to be called on controllable nodes. Uncontrollable children are dealt with in the process, and recursive calls are run on the next generation of controllable children.

Algorithm 5 multi_scheduled

```
1: Input: Node ← Controllable Node, k ← Allotted Questions
2: Input: C ← Chance Constraint
3: if k <= 0 then
4:     Return
5: multi_scheduled(Node, k - 1)
6: if Node == Root and node.queried[k-1][0] >= C then
7:     Exit
8: if Node.terminal then
9:     node.queried[k] = node.queried_SR
10:    Return
11: bestSR, bestQ = 0, []
12: for child in Node.children do
13:     child_newSR, child_newQs = 0, []
14:     for subtree in child.children do
15:         multi_scheduled(subtree, k)
16:         child_newSR = P(subtree|child) subtree.queried[k][0]
17:         child_newQs = subtree.queried[k][1]
18:     child.queried[k] = (child_newSR, child_newQs)
19:     bestSR, bestQ = max((bestSR, bestQ), child.queried[k])
20: newSR, newQ = enhanced_query(node, k-1)
21: if k == 1 then
22:     Node.queried[1] = single_scheduled(Node)
23:     Return
24: bestSR, bestQ = max((bestSR, bestQ), (new_SR, new_Q))
25: node.queried[k] = (bestSR, bestQ)
```

4.5.4 Example Walkthrough

Though it is not quite large enough to show the richness of our multiple query algorithm, it is still worthwhile to go through the exercise of walking through how this algorithm would behave when run on our representative example. Some details of this computation are shown in Figure 4-5. Note that there are only two levels in this system, so the maximum number of questions we can ask is two.

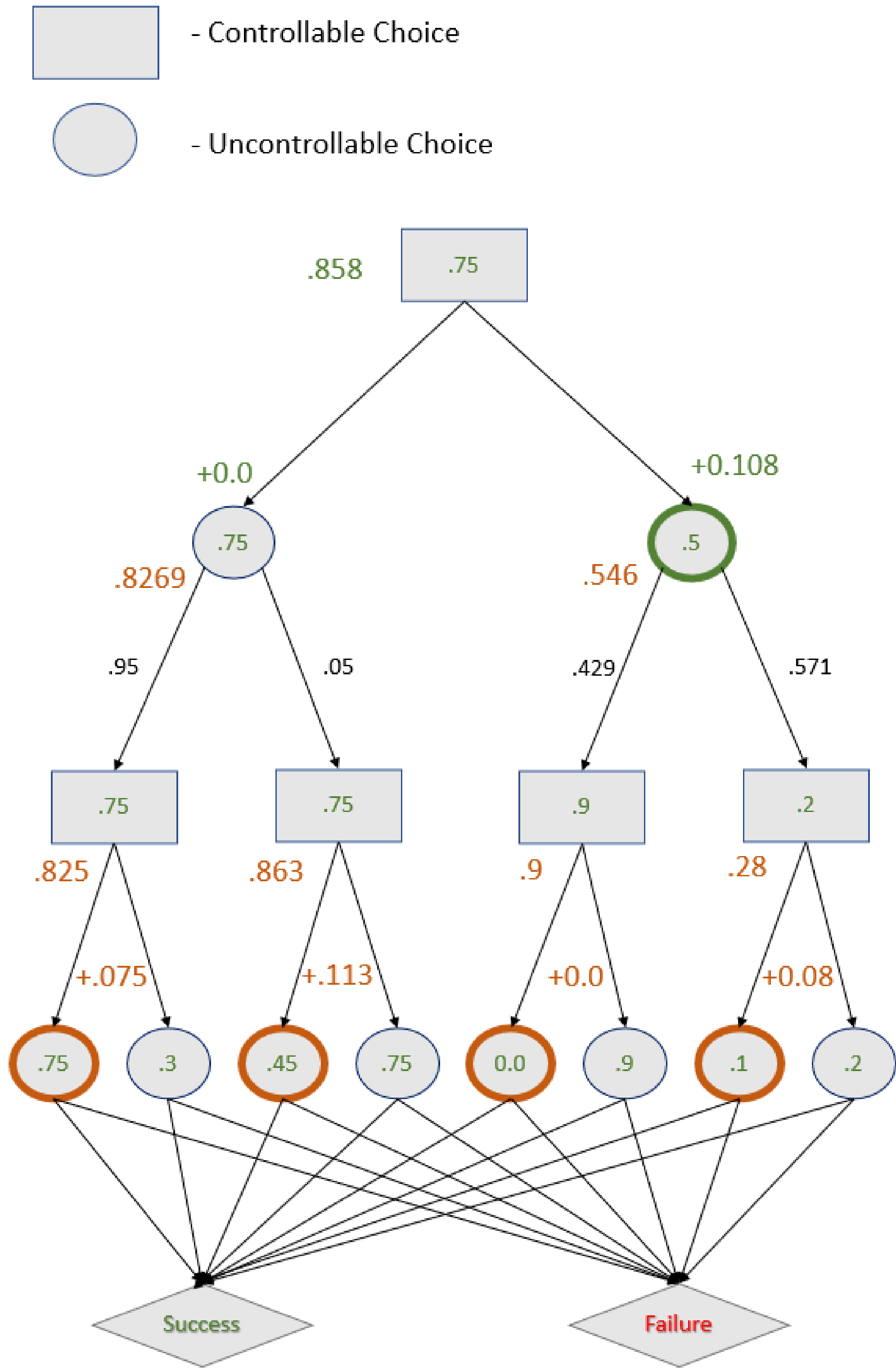

Figure 4-5: Annotations of our representative example when we run our multi query algorithm. The optimal questions are to ask about the green node first, then whichever orange node execution pushes us towards. There is no space for any additional questions in this system.

We begin by calling our multi-query algorithm for two questions on the root node. The first step is to recurse on the same node with one less question. We are now reduced to the single query scenario, as outlined in the previous section. Once this recursion is complete, we recurse on each grandchild of the current node with two questions. In other words, we run the multi-query algorithm on each node in the third layer with two questions. Since we are now at the terminal controllable nodes, and we only allow one question per node, we are now reduced to the second base case of being at the terminal controllable nodes with a single question. We have done this calculation many times, and the result is shown in orange in Figure 4-5.

Having completed the recursion, we now return to the root node. We know the optimal policy at the root node's children with one question, so what remains is to use our remaining question now. We calculate the utility of that question in terms of the success rate at the grandchildren given a single query policy there. In other words, we use the orange values rather than the base green values displayed in the node in Figure 4-5. The result of calculating the utility of asking a question now is shown in green. Performing the calculation shows that asking about the left child provides no benefit, since either outcome is still better than the alternative (the right child). Asking about the right child increases the success rate at the root to .858, so that is our optimal use of a question at this point. We now have the optimal policy at the root considering two questions, meaning we can terminate execution. The final optimal policy is highlighted in orange and green.

4.5.5 Optimality

Since this algorithm is recursive as well, we will pursue another proof by induction.

Base Case 1

In the first base case, we are at a controllable node and have one question to spend. In this case, we are reduced to the same problem that Algorithm 3 solves. Since we use that algorithm in this case, and we have already proved that Algorithm 3 returns

an optimal policy, we know that we return the optimal policy in this base case.

Base Case 2

In the second base case, we are at a terminal controllable node with k questions to spend. Since we are only allowing ourselves one question per controllable node, this reduces to the case where we are at a terminal controllable node with one question to ask. We fall back then to use of queried_SR(), which we know to be optimal. Thus we return the optimal policy in this base case as well.

Inductive step

Assume we are at some controllable decision node in the graph and we have k questions to spend. Further, assume that we know the optimal policy for the case where we are at the same node with $k-1$ questions to spend, and the optimal policies for the cases where we are at each controllable grandchild with 1 to k questions to spend. Since we are only allowing one question per decision, we have two options: ask one question now, or ask no questions now. If we ask no questions now, then the optimal policy is to select the optimal child with k questions allotted. Since we assume knowledge of these policies via the inductive hypothesis, we can compare directly and select the best. If we ask one question now, we evaluate the effect of one question considering each of the children with $k - 1$ questions allotted. Since we have the optimal $k - 1$ policies for the children, we can use the enhanced_query algorithm to find the best strategy for this case. Then we can simply compare the ask-now and don't-ask-now cases to find the new optimum. We now have the optimum policy for the current node with 1 to k questions allotted. We now satisfy the induction conditions for the parent of this node with $k - 1$ questions allotted. In short, we now have the optimal policy for the current node with k questions and the parent node with $k-1$ questions. This completes the proof by induction.

4.6 Querying With Communication Dead Zones

So far we have developed solutions to very ideal scenarios in regards to communication capabilities. We will now examine a less ideal situation, in which there are periods of time where communication is allowed and periods of time in which all communication must be halted. In the firefighting case study, this might correspond to not being able to interrupt a fire team once they have entered a burning home. Another example of this is sparse communication satellites. In that case, there might be well-known times at which all satellites are obstructed by a planetary body, so communication with them is impossible. Note that these scenarios, as well as the algorithm that we propose to solve them, focus on planned rather than spontaneous communication dead zones.

4.6.1 Problem Statement

We are given a decision tree representation of the task policy space, a deterministic model of queries, a chance constraint on task success, and an allowance of k questions. Further, assume we have completed the no-query optimal policy pre-processing step outlined in the previous chapter. Finally, we are given a set of controllable nodes at which we are not allowed ask questions. We are allowed to pick up to k frontiers of nodes that we can apply queries to. Our objective is to meet the chance constraint, if possible, with the minimum number of optimal frontiers without violating the timing restrictions.

4.6.2 Approach

Many of the same principles we used in the previous algorithms still apply in this case. Most importantly, we still want to ask questions at the last possible moment. The catch is that the last possible moment may not be right before the decision is made. Assume we start out at the leaves of our tree in the same way we did for the single and multi-query policy development algorithms. For the sake of simplicity, we will assume there is no communication dead zone at the leaves. Since we are not in

a communication dead zone we can simply proceed as we normally would, traveling up towards the root and finding the optimal policy for each subgraph as we go. Now consider the moment we hit a communication dead zone. Inside that dead zone we have a subgraph where we cannot ask questions. This doesn't mean that we can't query the decisions in that subgraph, it simply means we can't ask them at that time. Therefore, the most ideal time to query these decisions will be at the last node before we enter the dead zone. That particular node represents the ideal query point for it's children as well as all of its descendants within the dead zone. This means we need a good method to search all such descendants until we find the optimum to query at that point. Algorithm 2 for pre-execution queries solves that exact problem. The approach can be summarized as the following: apply the recursive method from Algorithm 3 or 5 as normal until a dead zone is reached. When we hit a dead zone, jump to the last available node before that dead zone starts and apply Algorithm 2 to the subgraph inside the dead zone. Then revert back to the Algorithm 3 or 5 protocol. A single-query approach to this process is detailed in Algorithm 6.

Algorithm 6 single_scheduled_dz

```
 1: Input: Node ← Controllable Node, C ← Chance Constraint
 2: if Node.visited then
 3:     Return: Node.bestSR, Node.bestQ
 4: bestSR, bestQ = max([(child.queried_SR, child) for child in Node.children])
 5: for child in Node.children do
 6:     if child.is_deadzone then
 7:         Return: pre_execution_dz(Node, C)
 8:     subtreeSRs, subtreeQs = [], []
 9:     for subtree in child.children do
10:         newSR, newQ = single_scheduled_dz(subtree, C)
11:         subtreeSRs.append(newSR)
12:         subtreeQs.append(newQ)
13:     child_newSR = child.updatedSR(subtreeSRs)
14:     bestSR, bestQ = max([(bestSR, bestQ), (child_newSR, subtreeQs)])
15: Node.visited = True
16: if Node == Root and bestSR < C then
17:     Return: None
18: Return: bestSR, bestQ
```

4.6.3 Algorithm Notes

The only component of this algorithm that bears special mention is in line 6. First we check to see if the child of the current node is in a dead zone. If it is, then we switch execution to a modified version of Algorithm 2. This modified version will run using the current node as the root and stop when it gets to a node that is no longer in a dead zone. Then it will run Algorithm 6 recursively on the leaves outside of the dead zones and send the best policy up the chain to the calling function. Note that as soon as the modified version of Algorithm 2 is called, Algorithm 6 essentially concludes. The function of the algorithm otherwise is identical to Algorithm 3.

Chapter 5

Experimental Results

In this chapter we experimentally validate both the effectiveness of questions on a team plan and the run time of the algorithms developed in this thesis. In the second half of this chapter, we run our algorithms on a scenario from our case study to illustrate how they might be applied in the field.

5.1 Randomized Trials

5.1.1 Experimental Methods

In our experimental trials we wanted to test the utility these algorithms provided against the broad range of scenarios that they might be applied to. To that end, we performed tests on our algorithms' performance as a function of tree depth, branch factor, saturation (i.e. question relative to tree depth), and base success rate. For our trials, we used a base tree depth of 8, branch factor of 3, and base success rate of about .18. For each set of benchmarks we performed one hundred trials. For each trial, the distribution over uncontrollable outcomes was randomized and the success rate at the leaves was selected using the base success rate as the mean. With the exception of communicating with deadzones, we ran every one of our algorithms on each tree. This standardization allows us to directly compare the algorithms against each other when applied to the same problem. At the end of each set of one hundred

trials we averaged out the success rates after each type of solution was applied. To contextualize these solutions we also took the average run times and plotted them in the same figure.

5.1.2 Depth Tests

Given their complexity, a primary concern for the algorithms developed here is their feasibility in real world situations. We would like our decision making assistant to be applicable to complex problems that are difficult for human coordinators to solve. To that end, we first tested how our algorithms performed as the depth of the decision tree increased. Deeper trees correspond to longer plans with more assignments for the executive and more sources of uncertainty from the teammates. Such scenarios require the robot to be much more forward thinking with its choices, as decisions early on can have radical effects on the world later. Unfortunately, trees grow exponentially with depth so fast runtimes are a necessity. In these tests, we endeavored to show that the computation time required to develop querying policies would not inhibit the assistant's ability to make quick decisions. For seamless interaction we would like the robot to be able to pick questions in one second or less, about the time it would take a human to do the same.

The results from our depth tests are shown in Figure 5-1. There are a few important notes worth mentioning in this graph. First, we see how effective even restricted querying scenarios have on otherwise very bad situations. The pre-execution queries alone increase the success rate of the six-depth case by over two times the base. Unfortunately, we see the utility of the pre-execution queries decrease significantly as the depth of the tree increases. This is likely due to the fact that pre-execution queries lose effectiveness the deeper in the tree their choice is. Essentially, as the tree gets deeper and more complex, the front-loaded questions are unlikely to play a significant role on the end of execution since so much can happen after. The only situation they would play a major role would be if early choices have a dramatic effect on the execution outcome. This is not the case on average, so we see the effectiveness of pre-execution queries taper off with tree depth.

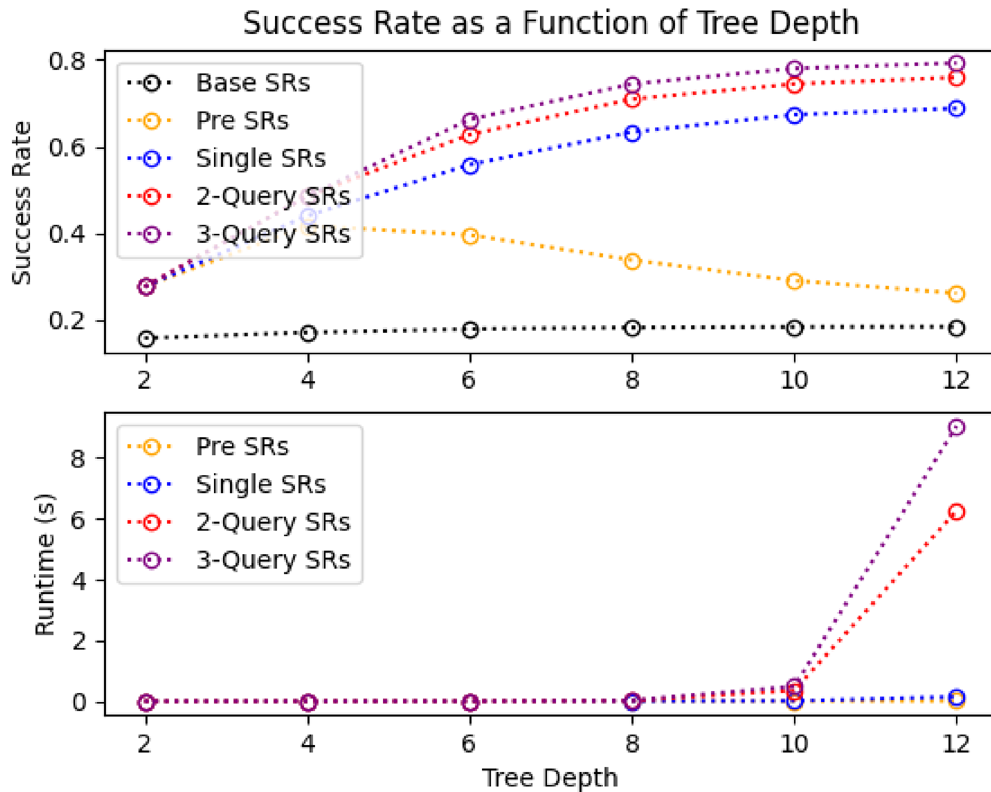

Figure 5-1: Shown are randomized trials for our algorithms as a function of tree depth. For every depth, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. tree depth for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the depth increases.

A particularly interesting aspect of the data shown in Figure 5-1 lies in the policy development methods. Even just a single-query policy has a tremendous impact on the execution success rate, as we can see. Application of the single-query policy algorithm increases the success rate at the root by about three times in the most basic case. But, perhaps even more importantly, rather than lose effectiveness as the problem becomes harder, the policy algorithms actually become *more effective*. This is thanks to the fact that allowing the assistant to save its questions till the last moment provides a flexibility that scales with tree depth. In fact, increasing the depth of the tree gives the assistant more options on how to spend its question, which is why we see the effectiveness increase. Perhaps most importantly of all, however, is the

run time of the single-query policy algorithm. As Figure 5-1 shows, its computational cost is almost identical to that of the pre-execution query algorithm. Notably, both are significantly less than one second, which is far below the point at which human team members would become impatient. This means we can get an effective increase in success rate of over 3 times the base with almost no significant time cost. The more extensive multi-query case does not paint as exciting a picture, but still shows modest success. In the case of three queries, we are able to still get almost a 33% increase in success rate over the single query case. The utility of going from a base success rate of .2, an almost hopeless situation, to .8, where the odds are very much in our favor, cannot be overstated. Unfortunately, this comes at the price of significant computation time. Due to the double-recursive nature of the mutli-query algorithm, the computation cost increases quickly with depth. At over three seconds for a depth of twelve, we are testing the limits of what a human would find acceptable in a partner. The upshot however, is that if we do a finite-horizon approach, we can extend the horizon to twelve decisions in the future.

5.1.3 Branch Factor Tests

While the depth tests present the natural case of execution getting longer, the effect of the number of choices in each decision has an equally important impact on teamwork. As the number of choices increases, we are effectively increasing the rate of exponential growth in the state space. If we have a large branching factor, even short plans can become intractable very quickly. With this in mind, we held the depth constant at eight and increased the branch factor for this round of experiments. The results of these tests are shown in Figure 5-2.

As we can see, the results from these tests are very similar to those of the depth tests. This is because they are in essence doing the same thing: increasing the size of the tree. In the varied depth case, a larger tree meant more options for *when* to ask questions, which led to an increase in query effectiveness for deeper trees. In the branch factor case, we have a similar result. A larger branch factor means more options for *what* to ask at a given time. The result is the same, more options means

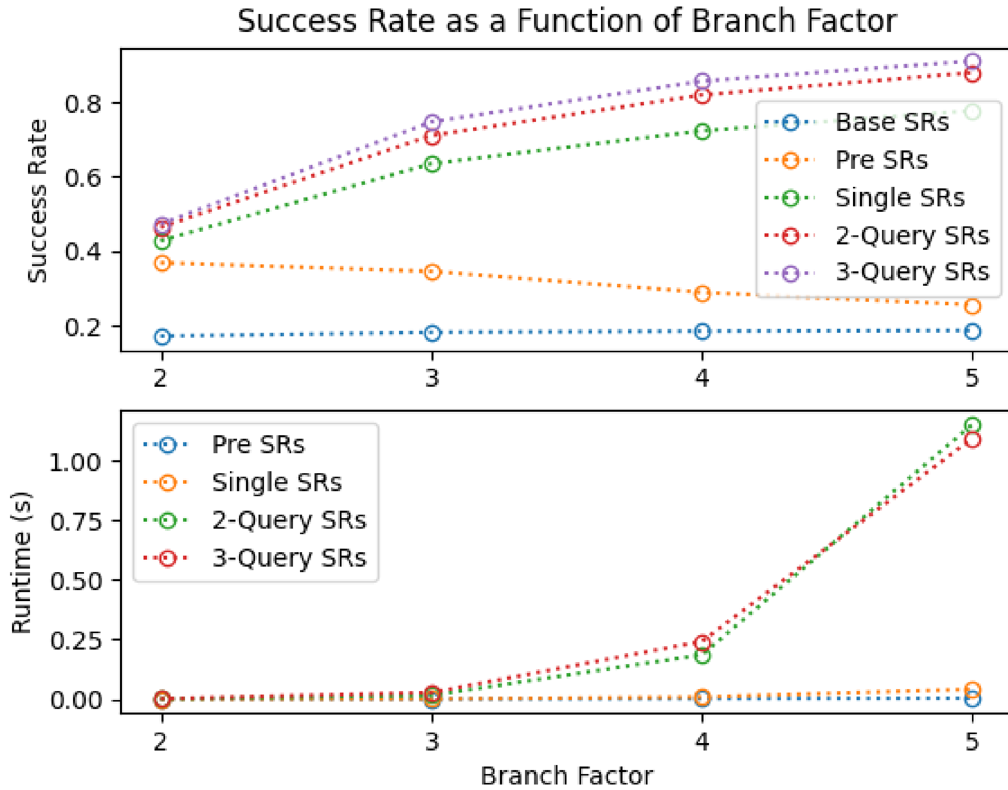

Figure 5-2: Shown are randomized trials for our algorithms as a function of tree branch factor. For every branch factor, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. tree branch factor for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the depth increases.

a higher likelihood of a good choice among them, which leads to a higher post-query success rate. Additionally, we see that the run times here are comparable to the depth cases. We still see the algorithms become quite costly for high branch factors, but they remain within reasonable human reaction times.

5.1.4 Query Saturation Tests

We have seen thus far that there is a diminishing return in the success rates as we allocate more questions to a problem. This is an unsurprising results, since asking questions about every single aspect of a partner's plan is not the most efficient way to collaborate. It is also worth mentioning this would be extremely annoying, and

likely not tolerated in most settings. In order to better understand how to balance effectiveness vs. number of questions, we tested how trees are affected when saturated with queries. Keep in mind, we are only allowing one query per decision in these trails, so there are only so many questions possible in a given tree. We would like to know at what point asking more questions provides negligible benefit. To that end, we performed 100 more randomized trials. The trees in these trials have varying depth, and we evaluated how each tree behaved as additional questions were applied. The results of these tests are shown in Figure 5-3.

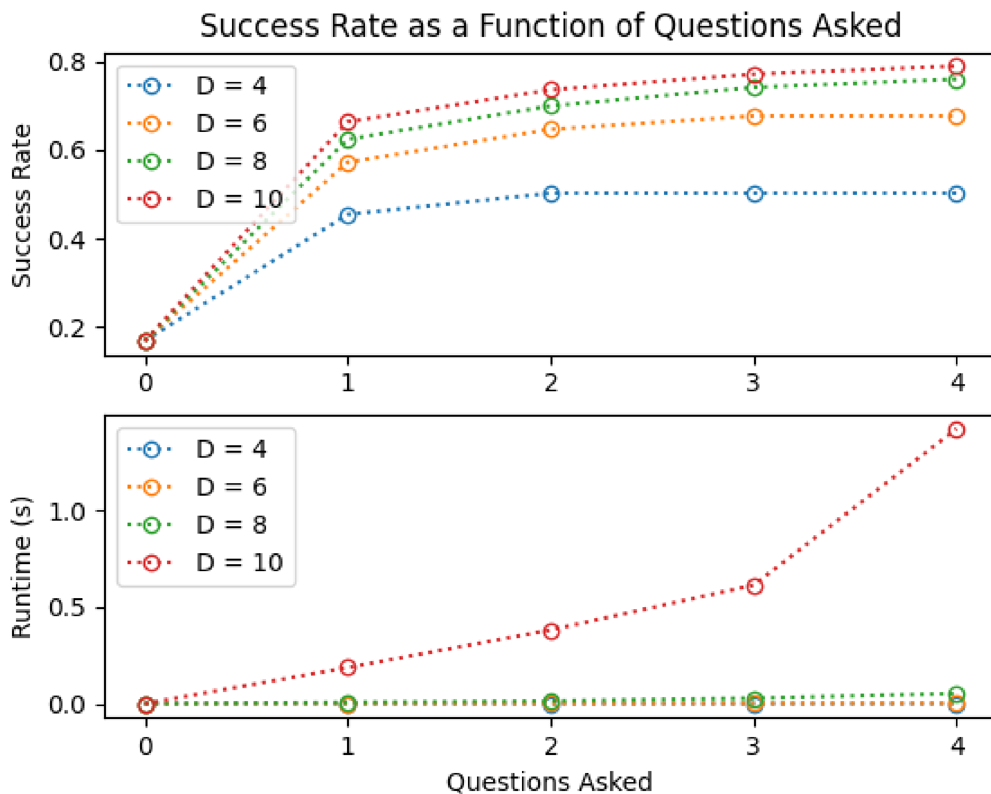

Figure 5-3: Shown are randomized trials for our algorithms as a function of number of questions allotted. For every number of questions, we performed 100 random tests over several different tree depths and averaged the results. In the top graph, we have the success rate at the root vs. allotted questions for each algorithm. In the bottom graph, we show the total time taken by each algorithm to run as the questions increase.

Since we can only ask one question per human decision, we fully saturate the tree when we allocate as many questions as there are controllable decisions. This

amounts to a full saturation point at half the tree depth. We would expect after the saturation point to see no increase in success rate for further questions. As the four and six depth lines show, that is indeed the case. Additionally, these tests further confirm the phenomenon we discussed early, in which questions are more effective in deeper, more complex trees. Finally, and perhaps more importantly, we can see just how starkly the returns diminish as we ask more questions. The most significant gains come from just one question, and after that we see only small increments in effectiveness up to the saturation point. If we are trying to balance processing speed with execution power, this might allow us to comfortably stick with the very fast single-query approach with confidence that we are close to maximum returns.

5.1.5 Base SR Tests

Our experimental trials thus far have been run with an extremely low base success rate, about .15. These trials represent situations that are nearly hopeless before we introduce queries. As our results have shown, even just a few queries are extremely successful at turning these bad situations into positives. To provide a more inclusive perspective, we would now like to see how effective the algorithms are as the situations become more hopeless, as well as how capable they are of improving an already good situation. To that end, we did further random trials, this time checking performance based on changes in the base success rate of the trees. The results of these trials are shown in Figure 5-4.

Before we start analyzing the results, it's worth noting that the base success rate value we are varying and the true original success rate of the tree are not quite the same. The value we are varying represents the mean success rate at the leaves of the tree. Since the trees are constructed with choice, the actual success rate at the root can rise above the mean. That is why we see the success rate at the root is about .9 when our input success rate is .5.

There are a few outcomes of note in these tests. The first is that we see the performance of the algorithms behave about how one might expect in good situations: starkly diminishing returns for questions spent in good scenarios. This makes sense,

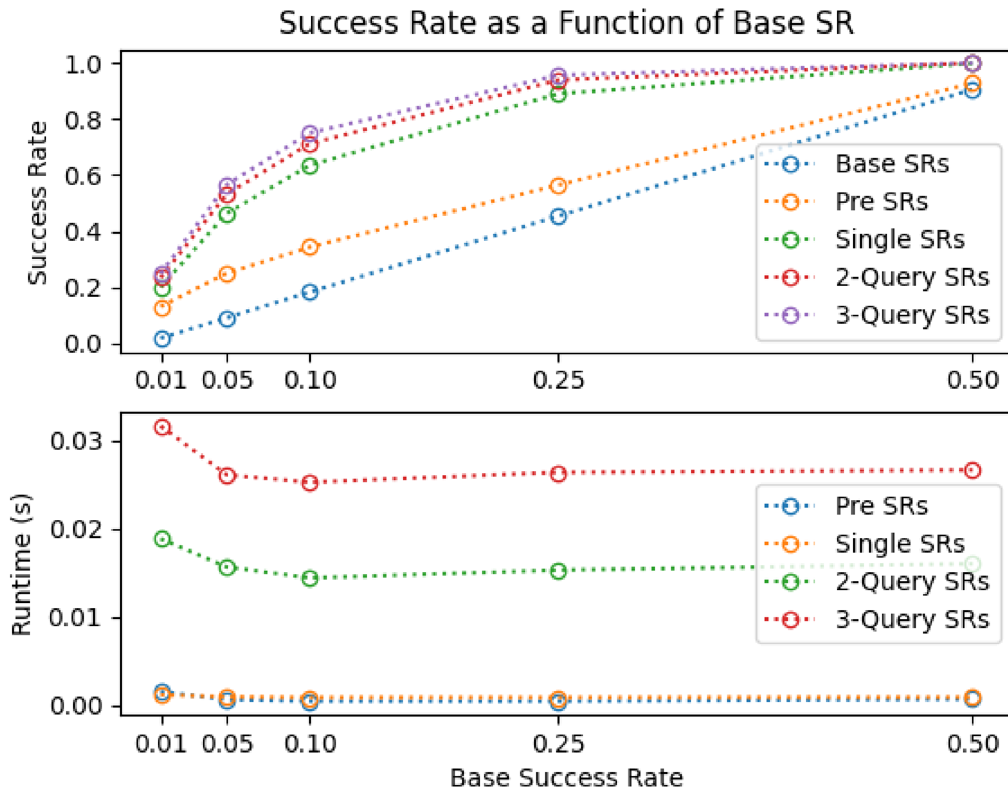

Figure 5-4: Shown are randomized trials for our algorithms as a function of base SR. For base SR, we performed 100 random tests and averaged the results. In the top graph, we have the success rate at the root vs. base SR for our algorithms. In the the bottom graph, we show the total time taken by each algorithm to run as the base SR increases.

as the more one is choosing between good options, the less a reduction in uncertainty improves results. We also see that the base success rate has no discernible impact on the computation time of these algorithms. This is exactly as we would expect. Since only the pre-execution algorithm involves pruning, it is logical that poor situations would be about as computationally expensive as good ones. In terms of the absolute difference between pre and post-query success rates, we see the greatest absolute difference at about .1. This is the base success rate we have been using in our previous tests, so it seems we have been highlighting its best performance criteria. On the other hand, the greatest *relative* difference comes as we get to the most hopeless states. In the .01 trial, we see the initial success rate rise to a little over .2. That's an increase

of almost 20 times! Though the final state is still far from ideal, that is a powerful transformation from just asking a few questions.

5.2 Case Study

While the data driven experiments give us a good idea of the general effectiveness of our querying algorithms, they don't quite give us a good look into how they might behave in practice. To give a more illustrative look at their function, we present their outputs when run on the case study instance discussed in Chapter 3.

5.2.1 Fleshing out the Problem

In Section 3.3 we laid out a specific instance of our firefighting case study, in which responders must react to the fire breaching containment and civilians needing rescue. In this problem, our task assistant's job is to assign tasks to five field teams. Three of these are fire teams (a senior, a junior, and a trainee), and two of these are medical teams (a senior and a junior). There are two critical containment zones that need to be manned by one fire team each, and both a fire team and a medical team must be sent to rescue the civilians. Additionally, one of the containment zones is designated to be in severe condition, while the other is in moderate condition. Complicating our assistant's assignment task is the fact that, due to confusion in the operations prior to this scenario, there is uncertainty in regards to each team's readiness level. Our assistant can resolve this uncertainty for a given team by asking them to do an equipment check, but due to the pressing nature of this problem there is not enough time to ask every team. Our assistant's job is to request the equipment checks that will raise the team's chance of success the most.

Before we can solve this problem, we need to specify a few more details. The first is what exactly it means for execution to be a success. We'll define the assistant to have succeeded if a fire team with high readiness is sent to the severe containment zone, and at least one team with high readiness is sent to rescue the civilians. Based on prior experience with these field teams, we'll say that the senior teams have a 75%

chance of having high readiness, the junior teams a 50% chance, and the trainee team a 25% chance. Finally, we will use the deterministic query model, so we can be sure that if a team says they are ready then they actually are.

5.2.2 The Solution, Intuitively

Before we see how our algorithms solve the problem, it is worthwhile to provide an intuitive solution that a human would pick as a comparison. Our primary concern here is the civilian rescue task and the severe containment task. Since the moderate containment zone does not have any restrictions on who is sent, we can just assign whoever is left over. Our senior fire team has a high chance of being ready, so it would probably be best not to waste a question on them. Conversely, the junior fire team probably won't be ready, so we can also assume it's best not to waste a question on them. Given time for one question, the best action seems to be to pose it to the junior fire team. If they answer that they are ready, we send them to the severe containment zone and the senior fire and medical team to the civilians. If they say they aren't ready, we send them to the moderate containment zone, the senior fire team to the severe containment zone, and the trainee fire team with the senior medical team to the civilians.

5.2.3 The Solution, Algorithmically

The pre-query likelihood of success for this problem given the average-out and fold back method is .703. While that may seem relatively high, for a high stakes situation like the one we are dealing with it is far below what we need. Put another way, it entails a 30% chance that either the fire breaks containment and spreads further or a field team is seriously injured. We can see that this is a situation in which queries are vital to save lives. Applying our single policy algorithm to the problem raises our success rate to .879. That's an increase of 25% from just asking a single question! As expected, the output policy is to always ask about either the junior medical team or junior fire team, depending upon the way assignments have progressed thus far. This

leaves our teams in a much better position where they can be confident that they can save the civilians and still contain the fire.

Chapter 6

Discussion

6.1 Summary of this Work

In this work, our goals were to advance the capabilities of autonomous executives working in teams with humans. We started in chapter two by motivating this work with a case study on forest fire fighting task forces. That case study allowed us to outline some of the features we wanted an ideal task executive to have for this use case. Next we identified a gap in recent work that could be filled by adding querying capabilities to existing executives. We finished the chapter with a formal definition of the query problem. In chapter three we built up a compact representation of the query problem policy space utilizing a decision graph structure. We then used that structure to show how queries affect the policy space without adding explicit states, and finished with a concrete example of our case study using this representation.

In the fourth chapter we developed algorithmic solutions for selecting queries using our problem representation. The algorithms we developed give the executive the power to select a single query before the team plan begins, a single query any time during the plan's execution, n queries any time during the plan's execution, and finally n queries during a plan with queries allowed only at certain times. In chapter five we tested our experiments with randomized trials. We showed that queries were able to increase the success rate of a team by up to 400% of the original success rate in typical scenarios, and do so on time scales comparable to those of a human. We

finished the chapter by running our algorithms on the fire fighting instance built in chapter three to illustrate their effectiveness in a grounded scenario.

6.2 Tractability in Practice

One major concern with implementation of the algorithms detailed here is tractability. While the dependence on graph size is linear, the size of the graphs that we search can be exponential for trees or diffusive graphs. As our experiments have shown, the computation time for multiple questions reaches the threshold of feasibility for trees at around depth twelve. While twelve decisions is a reasonable plan size, high impact deployments may need plans as long as 100 decisions or more. In these cases, use of our algorithms as provided would be infeasible. Fortunately, we can still make use of our work for these longer problems. One way to address this issue is to only search to a bounded depth in the plan space. As long as there is some method of estimating the success rate from a given decision state, this limited horizon search would be a straightforward modification to the algorithms presented here. The result would be a rolling look-ahead, where the algorithm looks for better query prospects as execution progresses. For situations with a set number of queries, this change may be painful as optimal queries late in the graph are unlikely to be discovered before the all queries are spent. On the other hand, this approach would be a great fit for situations where the number of questions allowable changes throughout execution. Examples of this type of scenario include situations where you can only ask one question a minute to avoid annoying your partner, or you can ask as many questions as you want as long as there is time to schedule them.

Another potential answer to the tractability problem is to add in state space pruning. So far, we have only incorporated pruning into the pre-execution querying algorithm, which is the weakest case. As we have seen, the single-policy algorithm runs quite fast in practice, but the addition of pruning could allow it to use time horizons of twenty or more decisions. Additionally, the greatest hit to performance for the multiple-query searches comes from the double recursion. From a pruning

perspective, the double recursion presents two potential avenues for reducing the search space and thereby double the potential gain in performance.

6.3 Future Work

This thesis lays a solid groundwork for a number of interesting future research directions. We will briefly describe several of them here.

Domain Independent Questions

In this work, we have not yet considered the scenario in which some or all of a teammate's decisions do not depend on one's own. To give an example, consider the case where you are trying to decide whether to give your cat a treat. Your decision is based on whether or not she will attack you. A domain *dependant* question would be, "Kitty, if I give you a treat will you attack me?" However, it may be the case that whether or not you give your cat a treat has no bearing on whether or not she brutalizes you. In this case, the dependent question is too restrictive, and we could simply ask instead "Will you attack me?" This question allows us to make a more informed decision on whether or not our cat deserves a treat.

Domain independent and partially domain independent decisions are a common occurrence. Fortunately, the policy development algorithms do actually handle most of these situations. The non-deterministic property of our single questions allows us to plan a different question for most domain values. The only scenario where domain independence affects policy development is in the one-step case. This would be the case where, as in the example above, you have a controllable choice followed by an uncontrollable choice that is not conditioned on your decision. In this case, some additional data markers and Bayesian inference would likely be enough to handle the situation. Unfortunately, pre-execution queries, and by extension communication with deadzones, does not have the non-deterministic property. Handling these scenarios will take additional consideration.

Queries with Effects

In this work we dealt with a very ideal form of queries, which generally took no time to resolve and had no effect on the world. Removing that restriction to allows for queries with temporal effects or queries that alter the state of the world. This relaxation would significantly increase the utility of our system. We can modify our case study scenario slightly to give a good example of this type of problem. Say we are trying to solve the same problem where the fire is in danger of spreading and we simultaneously need to plan a rescue for civilians. This time however, the severity of the fire at the containment zones is unknown. We have a scout team on site that can survey one of the containment zones, but they don't have time to do both. In this scenario, dispatching the scout team to the containment zone is a type of query. At the same time, this query has an effect on the world in that it changes the scout team's position and may put them in danger. Reasoning over these types of problems would be a good first approach to the future work of reasoning beyond questions.

Generalized Queries

Queries represent an investigative action that reduces uncertainty in the state of the world. As an extension to Riker, we consider that reduction as a tool to clarify a partner's plan in a team task. However, there is not any fundamental difference between asking a question and performing an arbitrary investigative action. This is even more clear when we consider queries with effects. Asking a question that alters the belief state is simply a specific instance of performing an action that alters the belief state. Thus it seems theoretically straightforward to extend the work done here beyond verbal questions to any investigative action that reduces uncertainty in a plan. If we combine that capability with the power to plan and schedule these actions as necessary, then we have a powerful executive capable of a broad range of independent tasks in uncertain environments.

Generalizing Beyond Queries

Queries are fundamentally investigative actions, i.e. actions that are primarily intended to reduce uncertainty in the environment. An important question for future work is understanding how the methods developed here may be of assistance in planning other types of actions as well. At their heart, the algorithms developed in this work are concerned with planning actions that affect transition probabilities within the plan at run time. Though we frame these actions as queries, and the change in transition probabilities as a result of gained information, we could theoretically extend the methods to any actions that affects these probabilities.

One example of this might be considering whether to call upon a teammate to assist you in the task. Calling the teammate may not give you any additional information, but it may significantly increase the chances of you succeeding in that task. Thus the action of asking for help could be viewed as simply a special type of action that alters the transition probability from your current state. Games are another potential example of this utility. There are many games that give you limited uses of an ability that allows you to reroll some components of the game. Poker is one example of this, where you can choose to reroll some of your cards by returning them to the deck and drawing more. Many computer games also allow for limited use abilities that reroll items, rewards, or enemies. These are generally "free-actions", meaning other than changing the outcome of the state they have no significant impact on the state of the world. Such cases would be a natural extension to the algorithms developed in this thesis.

Bibliography

[1] Nicholas Armstrong-Crews and Manuela Veloso. Oracular partially observable markov decision processes: A very special case. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2477–2482. IEEE, 2007.

[2] Sara Bernardini, Fabio Fagnani, and Chiara Piacentini. Through the lens of sequence submodularity. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 38–47, 2020.

[3] Anwesha Borthakur and Pardeep Singh. Drones: new tools for natural risk mitigation and disaster response. *Current Science*, 110(6):958, 2016.

[4] Maya Cakmak and Andrea L Thomaz. Designing robot learners that ask good questions. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 17–24. ACM, 2012.

[5] Timothy M Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[6] Patrick R Conrad and Brian Charles Williams. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research*, 42:607–659, 2011.

[7] Patrick Raymond Conrad. *Flexible execution of plans with choice and uncertainty.* PhD thesis, Massachusetts Institute of Technology, 2010.

[8] Shuonan Dong. Unsupervised learning and recognition of physical activity plans. 2007.

[9] Shuonan Dong, Julie Shah, Patrick Conrad, David Mittman, Michel Ingham, Vandana Verma, and Brian Williams. Compliant task execution and learning for safe mixed-initiative human-robot operations. In *Infotech@ Aerospace 2011*, page 1651. 2011.

[10] T Fong, C Thorpe, and C Baur. Collaboration, dialogue and human-robot interaction, 10th international symposium of robotics research (lorne, victoria, australia). In *Proceedings of the 10th International Symposium of Robotics Research*, 2001.

[11] Terrence Fong, Charles Thorpe, and Charles Baur. Robot, asker of questions. *Robotics and Autonomous systems*, 42(3-4):235–243, 2003.

[12] Janine Hoelscher, Dorothea Koert, Jan Peters, and Joni Pajarinen. Utilizing human feedback in pomdp execution and specification. In *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 104–111. IEEE, 2018.

[13] Ronald A Howard and James E Matheson. Influence diagrams. *Decision Analysis*, 2(3):127–143, 2005.

[14] Xin Xin Cyrus Huang. *Safe intention-aware maneuvering of autonomous vehicles*. PhD thesis, Massachusetts Institute of Technology, 2019.

[15] Mauro S Innocente and Paolo Grasso. Self-organising swarms of firefighting drones: Harnessing the power of collective intelligence in decentralised multi-robot systems. *Journal of Computational Science*, 34:80–101, 2019.

[16] Sang Uk Lee, Andreas Hofmann, and Brian Williams. A model-based human activity recognition for human–robot collaboration. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 736–743. IEEE, 2019.

[17] Steven James Levine. *Risk-bounded coordination of human-robot teams through concurrent intent recognition and adaptation*. PhD thesis, Massachusetts Institute of Technology, 2019.

[18] Steven James Levine and Brian Charles Williams. Watching and acting together: concurrent plan recognition and adaptation for human-robot teams. *Journal of Artificial Intelligence Research*, 63:281–359, 2018.

[19] David McAllester and David Rosenblatt. Systematic nonlinear planning. 1991.

[20] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction. In *Proceedings eighth national conference on artificial intelligence*, pages 25–32, 1990.

[21] Thomas Dyhre Nielsen and Finn Verner Jensen. *Bayesian networks and decision graphs*. Springer Science & Business Media, 2009.

[22] J Scott Penberthy, Daniel S Weld, et al. Ucpop: A sound, complete, partial order planner for adl. *Kr*, 92:103–114, 1992.

[23] Stephanie Rosenthal, Manuela Veloso, and Anind Dey. Learning accuracy and availability of humans who help mobile robots. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, 2011.

[24] Mihaela Sabin and Eugene C Freuder. Detecting and resolving inconsistency and redundancy in conditional constraint satisfaction problems. *Proceeding of Constraint Programming (CP'98)*, 1998.

[25] Pedro Santana and Brian Williams. Chance-constrained consistency for probabilistic temporal plan networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 24, 2014.

[26] Tod Schimelpfenig. Wilderness first aid: 5 steps of a scene size-up, Aug 2019.

[27] Julie A Shah. *Fluid coordination of human-robot teams.* PhD thesis, Massachusetts Institute of Technology, 2011.

[28] Toby Walsh. Stochastic constraint programming. In *ECAI*, volume 2, pages 111–115, 2002.

[29] Yuening Zhang et al. *Helping teams negotiate disruptions during task execution using distributed personal assistants.* PhD thesis, Massachusetts Institute of Technology, 2019.

Appendices

Appendix A

Pike and Riker

This module was designed as an extension of the capabilities of Pike and Riker, task executives for human-robot collaboration designed by Steven Levine. In this section, we will give a condensed overview of these executives and their mechanics to provide context for our querying module. For a full explanation of their function, see Dr. Levine's thesis.

A.1 Pike

Pike is an executive planner designed to work alongside a human on collaborative tasks. Pike's partner-centric design focuses on capability for current content recognition and adaptation. In a nutshell, Pike watches its partner's behavior and makes inferences about their plan based on past actions. As we will see, this top-level capability as well as the mechanics behind the scenes provide both excellent framework as well as a targeted area of immense benefit for the querying module we will develop later.

A.1.1 Background

Consider an autonomous robotic agent who's function is to collaborate with a human on a shared task. This agent will operate on an equal partner basis with the human,

Figure A-1: Pike in action on the Toyota Home Service Robot (HSR). In this image, Pike is assisting a human with a food preparation task. By observing the humans actions and making inferences about their intent, it is able to anticipate items the human will need and retrieve them.

meaning that the robot and the human have equal levels of authority and are capable of constraining each other's actions with their own. To give an example, consider a scenario where an equally partnered robot and human are collaborating on a road trip. When it's the human's turn at the wheel, they may go as they wish. When it's the robot's turn at the wheel, they may also go where they wish, even if their choices rule out potential destinations that the human wanted to visit. This sort of partnership provides the robot with more power to influence the success of a plan, however their actions must be that much more thoughtful to avoid constraining the human to poor choices.

In order to understand the world and place their actions in context, the human and the robot have a shared plan. If that plan were fully determined (i.e. no flexibility and no opportunity for the partners to make their own decision), then there would be little need for a task executive. Instead Dr. Levine considers the case where the

plan has multiple outcomes, and each partner is able to make its own choices along the way that may radically affect all future actions. We would describe such a plan as being *flexible*, in other words it can change based on the actor's whims, random probability, or in response to developing situations in the world. Such plans are far more interesting from a research perspective, and offer far more utility in real-world applications than a fully determined plan.

Though there is a plan in place, success for the duo may not be guaranteed. Even though there is a lose framework for what they may do, if both team members just behave randomly, there is little chance that they as a team will be successful. In fact, the very fact that they are a team implies that they can only succeed if they work together, and they work logically. That means the two team members pick and choose their actions to synergize with their partner. In other words, they must go beyond simple individualistic logic and instead plan from a team perspective. When humans go through this kind of process, they draw on logical inference as well as years of social experience to understand their partner's intents based on their actions. While the robot does not have the social conditioning that humans do, Pike aims to provide the logical inference that any human partner would exhibit when working in a team.

A.1.2 Problem Statement

Say we would like to provide an autonomous robotic agent to work on a shared flexible plan with a human. As part of the plan's flexibility, it allows for both the human and the AI to make pivotal decisions as execution progresses. In order to ensure plan success, the robot must be capable of interpreting and understanding a human's policy based on their past actions, as well as using those interpretations to select actions that ensure team success.

A.1.3 Pre-processing

All-pairs Shortest Paths

The first step in Pike's pre-processing phase is to use the temporal relationships stored in the TPNU to compute the all-pairs shortest paths between events in the plan. All pairs shortest paths finds the path and corresponding weight between all unique pairs of events in a graph. Since the weights in the TPNU correspond to time constraints, it finds the tightest possible time constraint between any two events in the plan. There are a myriad different ways to solve this problem [5], which we will not go into now. Most importantly, Pike will use this information later to determine the temporal feasibility of candidate paths through the plan.

TPNUs and PDDL World Model

As we have seen, the shared plan provides the centerpiece for the human-robot collaboration. With that in mind, we must now give concrete definition of what information the plan contains and how it is structured. Pike makes use of a data structure known as a Temporal Plan Network under Uncertainty (TPNU) as the framework for the plan. The TPNU outlines the structure of events that will happen as the plan progresses. Most importantly, it contains activities performed by both agents, controllable choices made by the robot, uncontrollable choices made by the human, and the temporal relationships between events. An example of such a TPNU is shown in Figure A-2. Perhaps just as important as what it contains is what the TPNU lacks. If you look at Figure A-2, you will see that it shows how events lead into the next temporally, but gives no information on how events are related causally. In short, it does not give an information on what through the plan are correct *causally*. There is nothing in the TPNU that would stop us from, say, getting juice and then making covfefe, even though we know that logically that sequence of events does not make sense. In order for Pike to correctly perform its inference, it needs causal information about actions and events fed to it as well. That is where the PDDL world model comes in.

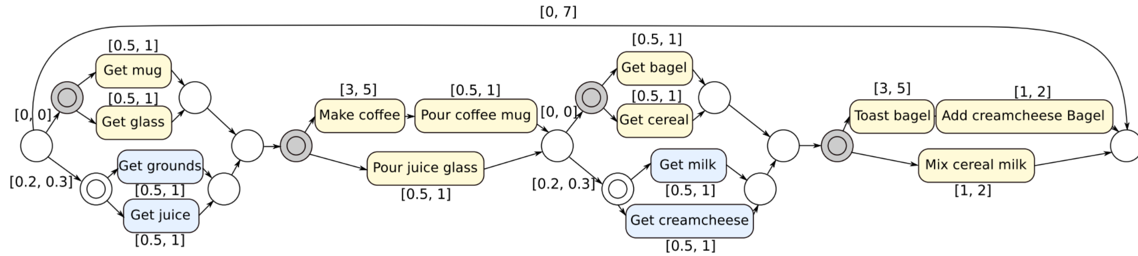

Figure A-2: An example of a standard TPNU. Concentric circles correspond to decision points in the plan; grey decisions are uncontrollable while white decisions are controllable. Blue events and yellow events correspond to actions by the autonomous agent and a human partner respectively.

The PDDL world model is an additional input to Pike that specifies the possible actions in the world, as well as the effect they have on the world state. For example, if the robot can choose to press button A or button B, the actions press-button-A and press-button-B will appear in the model, along with their preconditions (e.g. power-on) and their effects (e.g. power-off, machine-on). In terms of Figure A-2, the action Get-Juice would have effect Have-Juice and the action Pour-Juice would have the precondition Have-Juice. Using these preconditions and effects, Pike can extrapolate the causal relationships between events in the plan. For example, it can note that the action Pour-Juice is not causally complete without the prior action Get-Juice. To summarize, the TPNU encodes all potential paths through the plan as well as temporal feasibility information, while the PDDL model contains the information necessary to determine feasibility of paths. The next step is for Pike to actually perform the causal link extrapolation between events within the plan

Causal Links and Annotated TPNUs

Causal links are nothing new in the world of AI [19, 22]. Simply put, causal links are connection between a prior event which satisfies preconditions for a later event. The majority of activity and plan recognition done by Pike is performed via causal links. For example, say we are trying to understand what another driver is planning on the road. The options for their behavior mode may be to go straight, turn left, or turn right. The preconditions for turn left might be left-blinker-on, in-left-lane, speed-

slow, while the preconditions for turn right might be right-blinker-on, in-right-lane, speed-slow. Say the driver moves into the left lane. The action of moving into the left lane satisfies some of the preconditions for turning left, therefore there is a causal link between turning left and being in the left lane. On the other hand, it invalidates some of the preconditions for turning right. An action that invalidates preconditions (in this case moving to the left lane invalidates preconditions for turning right) is a special type of causal link known as a threat. Thus if a driver moves into the left lane, we know that turning left remains a possibility while turning right may be pruned from consideration. In this way, Pike uses past actions in the context of causal links to compactly maintain a set of feasible decisions both for the agent it is trying to predict and for its own actions relative to that agent.

Causal links are a distillations of the feasibility information provided by the PDDL model. They combine naturally with the event progression information encoded in TPNUs. In order to compactly represent the information from the both those domains, the causal links and the TPNU are combined into an annotated TPNU. Such a TPNU is shown in Figure A-3. This is the first step in Pike's pre-processing goal of condensing all of the information necessary for decision making into a single data structure.

Labeled Value Sets

In order to keep track of plan constraints as they are found, we turn to labeled value sets (LVS), which were created for another planner from the Model-base Embedded Robotic Systems group known as Drake [6]. The LVS is designed to compactly encode the tightest possible constraint for a value as a function of choice. In this case the "value" is generally whether or not an action is possible (and part of a route to plan success), and the constraints are conditions on the world that need to hold. Each set is made up of labeled values pairs, which are tuples of the form (α_l, ϕ_l). Over a particular relation, α_l represents a constraint value that holds whenever the environment ϕ_l holds. For instance, over a relation $<$ for variable x, $(1, (y = 1, z < 2))$ means that $x < 1$ when $y = 1$ and $z < 2$. Sets make use of the notion of dominance

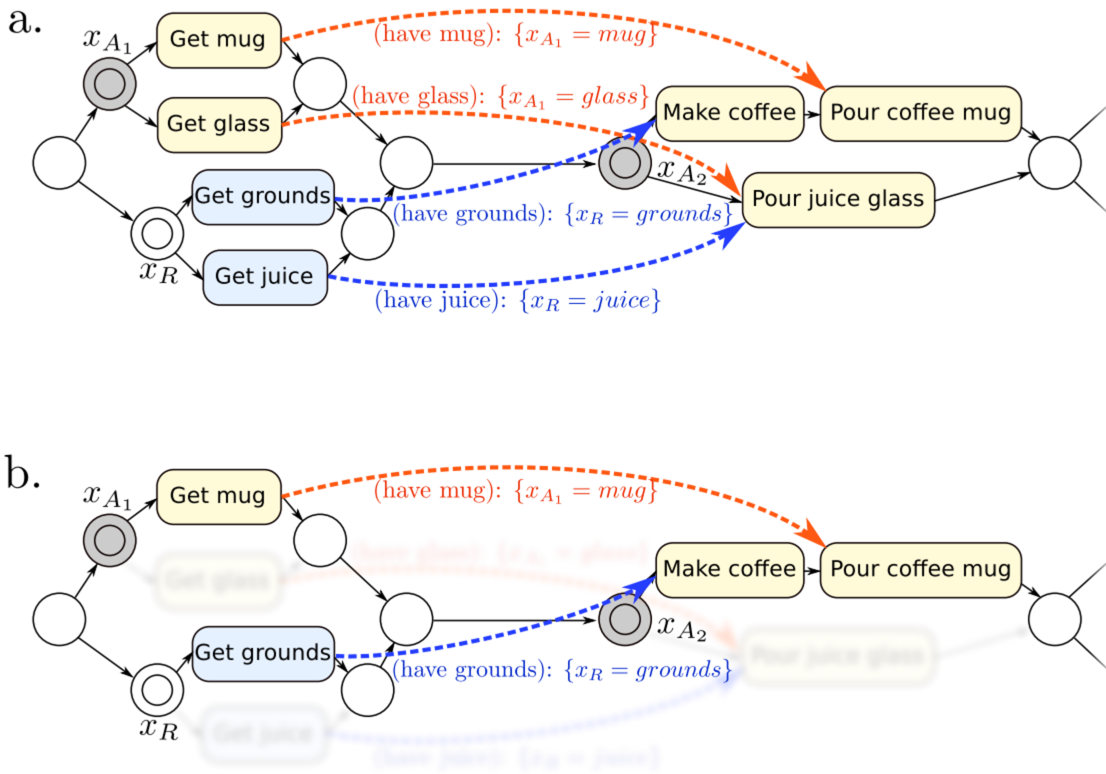

Figure A-3: An example of how causal links may be used to annotate a TPNU. Image a shows the causal interactions between events in one section of a TPNU. Image b shows how a choice made earlier in the plan (x_{A_1}) can be used to prune branches later in the plan.

to stay as compact as possible. One constraint dominates another when one one always implies the other. For example, the labeled value pair $(1, (z < 2))$ dominates $(2, (y = 1, z < 2))$ since the second is true whenever the first is. Thus we can safely ignore the second constraint and stick with the tighter bound offered by the first. Labeled value sets have the potential to grow quite large due to the possibility of a unique constraint for every environment. Fortunately, the notion of dominance tends to keep each set much more compact than the worst case. Labeled value sets are essentially compiled versions of the constraint information contained within causal links. They are used to perform very efficient checks in the annotated TPNU as to whether or not a specific action can lead to the goal.

A.1.4 Online Execution

While the pre-execution phase does the groundwork, it is the online component of Pike that actually decides on controllable choices as well as schedules and dispatches events. Pike takes a least commitment approach to dispatching, waiting until the last possible moment to commit to actions so that it can collect as much information from the world as possible (primarily through observation of the human). Unfortunately, checking to see if the current tableau of a TPNU is solvable (i.e. there exists a correct execution strategy) is NP-complete. Despite this, Pike maintains completeness and manages to do so while maintaining good performance in most cases.

As input, the online execution phase of Pike takes in the matrix of all-pairs shortest-path information, the causal links, and the labeled value sets compiled from those causal links. It also takes in relevant information that is received at run time, specifically the outcome of human choices, timestamps for when activities finish, and state predicate estimations. The Pike execution strategy is based on that of Drake, which was also designed for systems with choice [6, 7]. Pike adapts Drake for human-robot collaboration tasks by adding in considerations for causal completeness and execution monitoring for causal links. It maintains the freedom to schedule events at any time so long as there remains at least one successful execution path through the TPNU.

In a nutshell, Pike maintains execution windows for each event in the plan. The upper bound and lower bound of the execution window are labeled value sets, so they can be readily reasoned over and changed as different constraints become active. Whenever an event is scheduled, executed, or changed, the changes in the time window are propagated to the event's neighbors. As execution progresses, Pike will check all remaining events to see if they can be executed immediately. This check is based on what constraints need to be true to execute that event now, and whether a conjunction of those constraints and the current knowledge base allows for at least one successful execution path. If so, the event is executed.

Since Pike is designed to include execution monitoring as well as scheduling, there

is also infrastructure for online responses to unexpected events. Say for example that an event takes too long and violates its temporal bounds. Pike deals with such a situation by first propagating the altered time frame out to all events. If the new time windows make certain values in the labeled value set infeasible, then Pike will add in new constraints to reflect that those values are no longer attainable. It can then proceed with constraint-based scheduling as usual. Alternatively, consider the case when new information is gleaned that alters the plan. For example, say a plan involves using a car, but execution monitoring determines that the car has a flat tire and so is no longer usable. This type of disturbance is a matter of causal link monitoring rather than temporal flexibility. However, Pike will handle this situation in much the same way. First it will add in constraints to the knowledge base to reflect the new impossibilities. Next, it will check to see if the flexibility in the plan allows for a course correction to a new successful path. If so, then the plan is refactored to circumvent the problem. Otherwise, execution failure is signaled.

For a more in-depth view at Pike's online execution strategy, as well as example executions, see Dr. Levine's thesis.

A.2 Riker

Riker is the evolved version of Pike designed for executing plans with risk and uncertainty. Rather than the equal partners scenario of Pike, Riker takes a subordinate role to the human in its execution choices. To that end, it will do it's best to constrain the human as little as possible while maintaining plan success to a high degree of fidelity.

Riker considers the case where the autonomous agent and human are working together in an uncertain world. This uncertainty can be both in the state of the world (Is it going to rain? Is my tire flat?) and in the partner's policy (Are they going to work today? What do they want for dinner?). In such cases, the progression of events is stochastic rather than deterministic, so it becomes necessary to think about plan success in terms of probability. Now it is much more important for the robot to make prudent choices, since their decisions can have significant effects on

the outcome of the plan. Additionally, if the human's policy is uncertain, it is critical that the robot is able to interpret and extrapolate what the human is planning in order to mould its own policy accordingly. Thus, in order for a task executive to be successful in this scenario, it must be capable both of intelligent task selection in probabilistic instances, but also human activity and policy recognition. These are the considerations Pike was created to address [17].

A.2.1 Influence Diagrams

As the earlier section on MDPs highlights, there is no single way to represent distributions and progressions over random variables. In that section, we saw that POMDPs are intractable for the problems we wish to solve due to the need to handle temporal constraints and variable decision nodes. Instead, Riker uses a formalism known as influence diagrams, which is a graphical form generalized from Bayesian networks [13]. Riker makes use of the decision nodes that influence diagrams add to Bayesian networks to represent the controllable choices of an agent among random variables representing the environment and collaborators. Decision nodes can influence the distributions of random variables (and hence have a place in probability tables), however they are not random themselves. Optimal policies to influence diagrams can be found by imposing a variable order over the nodes in the diagram, then using the method detailed by Jenson and Nielson above to solve for the optimal policy via dynamic programming.

A.2.2 The Conditional Stochastic Constraint Satisfaction Problem (CSCSP)

Riker utilizes a novel problem formalization dubbed the conditional stochastic constraint satisfaction problem, which is based off both the conditional constraint satisfaction problem [20, 24] and the stochastic constraint satisfaction problem [28]. The stochastic elements include a subset of uncontrollable choices, probability distributions over the outcomes of those choices, and the objective to find a policy that meets

at certain minimum probability of success. On the other hand, the major conditional element is that some choices and variables within the problem may not be a part of the solution (for example if the conditions for a certain choice are never met). For a full in-depth look at the mechanics of the CSCSP, see Dr. Levine's thesis in which he provides and excellent breakdown of the novel problem formulation. The major benefits of the CSCSP are as follows: (1) it includes all of the elements of the SCSP and the CCSP that comprise this problem (2) it allows a compact and intuitive representation of complex interactions between the components, such as constraints that may be inactive due to choices never being reached (3) it allows the influence diagram to reason about scenarios where constraints and variables may be inactive, and (4) it allows the distribution of uncontrollable decision variables change in respect to the outcomes of controllable variables. Fortunately, these conditions also perfectly meet our needs for the problem we will be solving with the application of active querying.

A.2.3 Solving the CSCSP

While the CSCSP provides the problem set up, what remains is to solve the problem for workable policies. Unfortunately, the number of policies in this formulation is worst-case doubly exponential (as it would be in an MDP formulation). The challenge now is to be able to reason over such a large number of policies in a way that is both tractable and as complete as possible. In order to answer this challenge, Riker includes a novel technique dubbed the policy binary decision diagram (BDD). The BDD increases the computational tractability of this problem formulation by reducing duplicated structures in the explicit graph, exploiting structure in the factorization of the choice influence diagram, and allows incremental online updates as observations come in. These improvements allow the BDD to reduce time and space complexity significantly. In some cases, those improvements can even be exponential.

We will now provide a very brief overview of how the BDD in constructed. First, Riker compiles away the conditional variables in the problem, essentially reducing it into an SCSP. In short, it does this by adding an additional value of "o" representing inactivity to conditional variable domains, and then adding in additional constraints

to ensure that domain value is applied correctly. Riker leverages ideas from weighted model counting and stochastic games search to incorporate constraint information from the influence diagram into the explicit graph of the SCSP. The optimal policy is then computed from the explicit graph using the average out and fold back method discussed earlier. To represent the explicit graph compactly and thereby decrease the computation time, the explicit graph is transformed into a BDD representation. To save time, the explicit graph is not actually generated, rather the BDD encoding is built up symbolically (rather than being converted from a fully realized explicit graph). The best policy is then computed directly from the BDD using a version of the average out and fold back algorithm adapted to operate correctly on the new data structure. The end result is a tractable method for computing the optimal policy of the SCSP.

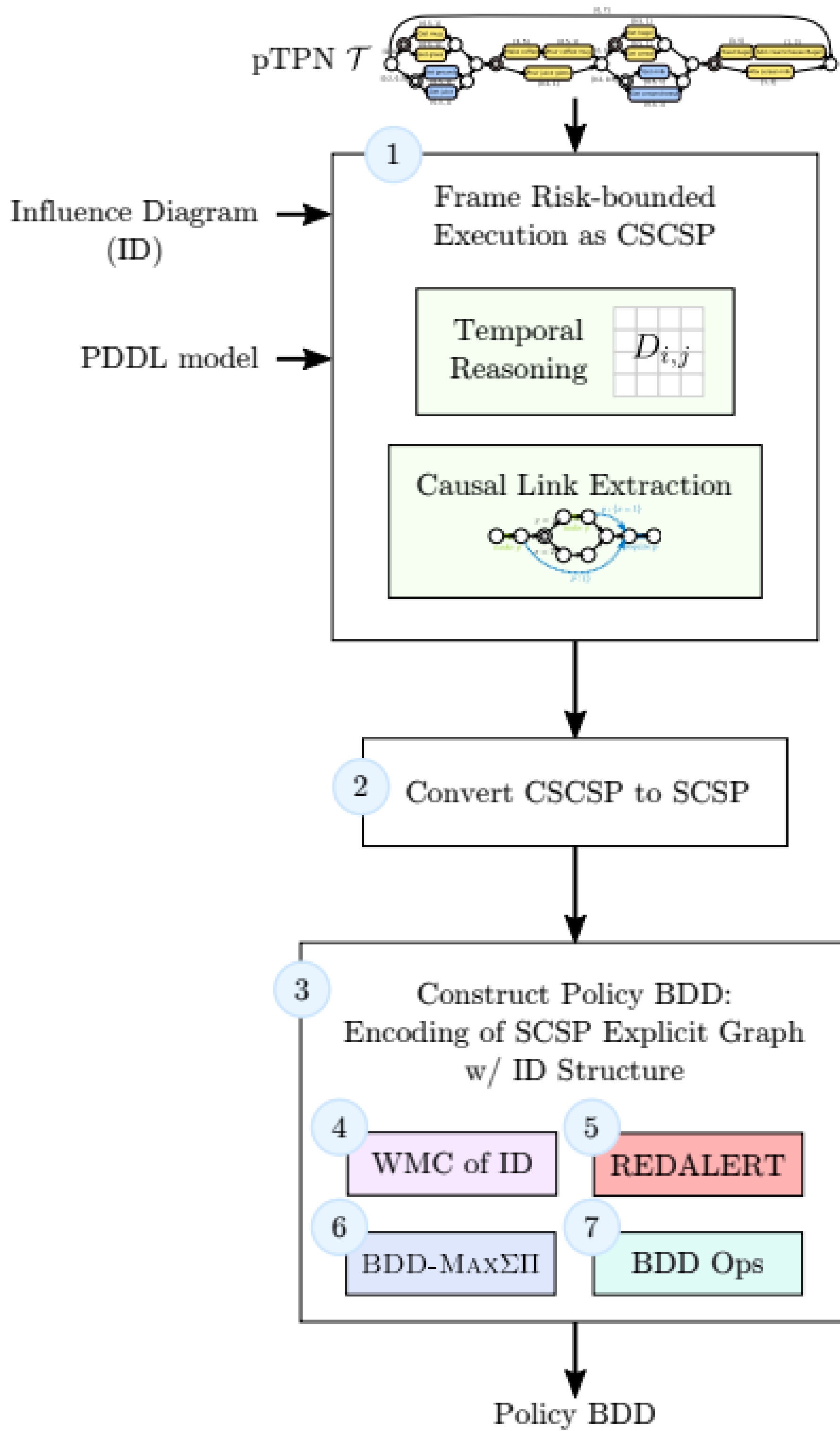

Figure A-4: A high level overview of Riker under the hood. We will briefly cover the inner working of this system, but for a more complete picture see Dr. Levine's thesis.

Appendix B

Multi-Query Model

B.1 Multi-Query Model

So far, we've developed a representation of how questions affect the probabilities within our plan space. Our final task is to investigate how queries affect *each other*. As we have discussed, our question model is considered from the perspective of a controllable choice. Since not all responses are favorable, and each controllable choice has multiple children, we must consider the possibility that we ask multiple questions about the direct children of a single controllable node. For an example, let's return to our brash fire team example. If there are multiple houses we could scout, we might want to ask about there plan for several of the different homes were considering before we make our choice. As it turns out, this scenario is rather complicated and involves additional layers of Bayesian inference.

Say we have a controllable choice with children a, b, and c. Say a is the pre-query optimal child. Now say that we query b. As before, there are two possibilities: the response is favorable or it is not. We will consider these outcomes to be case 1 and case 2 respectively. In case 1 we have a favorable response, meaning that b is now the optimal state. If we ask another question, then b is the new back-up option. In case 2, a is still the optimal state. If we use our second question on a, then the back up can be either b or c depending upon what the response was. As we can see, the expectation calculation will become quite complex for higher orders of querying. For

now, we will just develop the case in which we have two questions that have been locked-in before hand.

Let's begin building up the equation for the expected success rate of a controllable choice after querying two of its children. Say we ask about state a and state b and get back answers q and k. We will want to pick the option with the highest success rate, so we go with the option that is the maximum of q, k, and our backup state c. The success rate of that outcome then is $MAX[P(S|q), P(S|k), P(S|c)]$. The probability we get those responses is the compound probability $P(q|a, k|b)$. Assuming that the events are independent, we can use a common property of Bayesian statistics to write $P(q|a, k|b) = P(q|a)P(k|b)$. Then the contribution to the expected success rate of a given controllable choice from getting answers a and b is:

$$P(q|a)P(k|b))MAX[P(S|q), P(S|k), P(S|c)] \tag{B.1}$$

Now we must consider all outcomes. We can do so with a double summation to get the final result

$$P^c_{b,a}(S) = \sum_{k\in\mathcal{O}(b)} \sum_{q\in\mathcal{O}(a)} P(q|a)P(k|b)Max[P(S|q), P(S|k), P(S|c)] \tag{B.2}$$

Note that we are representing our backup option, c, as a superscript. Given this equation for two questions on the same controllable choice, the extension to n questions is relatively straightforward: simply add in additional summations and additional items to the max calculation. In terms of complexity, we can see that the number of items in the MAX will never exceed $n+1$, so the inner loop will take $O(n)$ time and therefore the entire process will take $O(b^n n)$ given a choice with a branch factor of b. This is unfortunately exponential, however it is unlikely that we would ever devote a large number of queries to a single controllable choice (imagine how annoying that would be for your partner!), so in practice it is quite manageable.

There is an important caveat to the analysis we just did. Equation B.2 corresponds to the case where we *plan* to ask about a and then b. But there are scenarios where we may want to hold off on committing until after we get a response to the first question.

Consider the case where state a has a number of outcomes all with a success rate of between fifty and sixty percent. On the other hand, say c has outcomes all with a success rate below 20% except for one unlikely outcome with a success rate of 100%. With that in mind, if we query b and get back a success rate of 61%, then there is no outcome from a that will improve our chances of success. There is, however, one such outcome from c that will increase our chances of success. In this case we might want to choose to query c whereas normally we would allocate our second query to a. This example shows that Equation B.2 is not enough to capture the full capabilities of dynamic querying.

We finish our analysis of query representation by developing an equation for the expected success rate given uncommitted questions. To do that, assume we have some policy π_R belonging to our robot assistant that takes in a response from a human and decides what to query next based on that response. For instance, if the human responds with $k \in \mathcal{O}(b)$ our policy might spit out $\pi_R(k) = a$. If we use Equation B.2 for reference, we can see that the outer loop will still be the same, since we are considering all possibilities of an answer to b first. The inner loop, however, will not always be a. In fact, it will be whatever the output of $\pi_R(k)$ is. Now all we need to complete the equation is a means of picking the backup. As it turns out there are only two possibilities for what that backup can be: the original backup to b or, if that is already queried, the second best (in this case c). Note that as the number of queries expands, the number of possibilities for the backup will expand as well. We can avoid adding in a complicated switching function by simply putting both values in the MAX field. The final equation then becomes

$$P_{b+} = \sum_{k \in \mathcal{O}(b)} \sum_{o \in \mathcal{O}(\pi_R(k))} P(o|\pi_R(k))P(k|b)Max[P(S|k), P(S|o), P(S|a'), P(S|c')] \tag{B.3}$$

Note that we are using prime notation to mark that a certain probability is considered post-query, but in this case the query might not have been applied to that state, so success rate is potentially unchanged. In comparison to Equation B.2, we've

only added one item to the MAX field, so our computational complexity for this more powerful case is the same! That is assuming, however, we ignore the issue of how to handle π_R. With a look-up table we can consider the complexity of that function to be $O(1)$, but that does not address the question of *how* the function is determined in the first place. Unfortunately, as we have seen the policy is determined by the individual successors of a state, not the state itself. We can see now that the calculation will become fairly complicated. Let's do it!

Say that querying b gives some answer k with success rate $SR_k = P(S|k)$. If SR_k is less than the success rate of two other states, then it is irrelevant since no matter what the outcome of the next query we will never choose to go to k. In that case, we can just pick the optimal choice for the second query normally. If SR_k is less than the success rate of only one other choice, then k will only be the backup if we query that best choice. In that case, we can simply update the calculation for the outcome of querying the best choice using k as the new backup and compare against the other original single-query options (since their result will be unchanged). In the messiest case, SR_k is the best of all the options, so it will always be the backup. In this case, we must recalculate the expected single-query success rate of all other options using k as the backup. In short, the optimal second-query policy requires that we check all options in the worst case, making it $O(b^2)$ for a branching factor of b. If we desire the optimal over all combinations of possible queries, we are reduced to a brute-force approach which gives us a complexity of $O(b^4)$. For small branching factors, this is manageable. For large branching factors however, it can be quite painful. If given a good approximation of the optimal π_R, we can reduce to the slightly more manageable $O(b^3)$.