SENSING AND INFRASTRUCTURE DESIGN FOR ROBOTS: A PLAN-BASED PERSPECTIVE

A Dissertation

by

GRACE ANNE McFASSEL

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2023

Major Subject: Computer Engineering

ABSTRACT

Currently there do not exist general-purpose robots, and the procedures by which robots are designed are often ad hoc. Additionally, designers must deal with considerations including budget, energy requirements, and the availability of parts, all of which complicate the problem. Abstract formal theories have, among other benefits, the potential to assist designers in developing and understanding the capabilities of novel robotic systems. Of particular interest is the concept of action-based sensors, which focus on the idea that a robot need only know enough to know what action to perform next. As a mathematical abstraction, action-based sensors prescribe actions to the agent; details of the sensor itself are irrelevant. From this information-oriented perspective, this concept also links planning directly to the design problem: the definition of what action "should" be taken depends upon the plan a robot is executing, serving to specify its desired behavior. While the theoretical abstractions of sensors are technology-neutral, we present ways to connect action-based sensors to the considerations and constraints faced by real robot designers.

Action-based sensors have been formalized in terms of specific plans (informally those that take the fewest actions to achieve a goal), but there exist cases in which it is useful to consider other plans. In extending this formalization to include all plans, we find that certain plans have obstructions that prevent their expression as action-based sensors. We have developed an algorithm to remove these obstructions, which result from the interactions between a robot and its environment. After this, we move from the question of what a robot must sense about the environment to the question of how an environment should provide information. The use of infrastructure for spaces shared by multiple agents is another way in which designers can simplify tasks for agents. The complexity of this design problem arises from infrastructure's ability to modify both what an agent observes and the outcome of actions. We present a method for modeling the impact of infrastructure to determine its utility to a given agent, and also consider how the utility of the infrastructure can vary depending on the differing needs of agents and how they make use of the environment.

The present work, in addition to extending Erdmann's original theory, focuses on the way in which information that must be retained by the agent can be contained within a plan's structure. Use of a graph-based framework allows for us to identify if that structure is necessary for successful execution of the plan. This dissertation then shifts to a complementary design problem, examining the ability of infrastructure to externalize information and actuation requirements. It also presents a model for predicting the impact of introducing new infrastructure. Finally, it will explore the ways in which information can be used to estimate sensor failures in robots and bound the space of possible configurations.

Transitioning from the design of robots and their environments to their operation, this dissertation also presents a method for estimating sensor failures. Through knowledge of the world structure and expected observations, inconsistencies can be tracked to form hypotheses on potential sensor failures. We introduce a lattice-based method of expressing these failures, as well as an algorithm for tracking inconsistencies. The algorithm allows for an often concise representation of a potentially exponential set of hypotheses, enabling use during a robot's execution. This basis also allows for the robot to determine if a failure interferes with its ability to complete a task. We also present a method through which the sensors that are required for task completion can be determined at any point.

The primary means to validate the theoretical results in this dissertation are a range of case studies. For action-based sensors, we consider several varieties of design problems including sensor selection and navigation problems. Moving beyond the sets of action-based sensors considered in these design problems, we also examine concise combinatorial representations for sets of sensors more generally, and apply these to settings involving robot self-diagnosis. For infrastructure, we provide a taxonomy as a guide by which to examine several different cases in which infrastructure is introduced to an environment. These case studies focus both on changes in agent behavior after being introduced, as well as ways in which the value of the introduced infrastructure can be determined. For the identification of sensor failures, an example is also presented that demonstrates the concise nature of the model, particularly when compared to naïve methods.

DEDICATION

To Vere, who never let me live down being an Aggie,

and to my cat Beeps, who has kept me sane.

ACKNOWLEDGMENTS

As the saying goes, it takes a village to finish a Ph.D. Accordingly, this dissertation exists only due to the guidance and support of my colleagues, friends, and family.

I am immensely grateful to my advisor, Dr. Dylan Shell, for his guidance and seemingly endless patience. His enthusiasm for his work and that of his students is both obvious and infectious, and this dissertation would not exist were it not for the many productive meetings we shared.

I would like to thank my committee for their valuable comments and suggestions throughout the writing and revision of this dissertation. I wish to give special thanks to Sue Chrysler and 3M for their support through funding my research.

I also could not have done this without my family. I am grateful to my parents, who have provided constant support and who will be happy to hear that this dissertation is finally finished. I am also grateful to my sister Rose, who has believed in me every step of the way and made sure I remembered to take breaks. Additionally, I would like to thank the Robinsons, who have always made me feel at home and with whom I have shared many wonderful Thanksgivings.

I must also thank the members of the Distributed AI Robotics Lab, past and current, for the various outings we shared and for making me realize that all my world-ending Ph.D problems were the same as everyone else's. I wish them all the best of luck in their research and future endeavours.

I am indebted to my many other friends who have supported me through this process: particularly Kristin and Mike, who had seen it all before during their own time in graduate school; Dav, who indulged me with multiple multi-hour voice calls while I worked through writing parts of this dissertation; and everyone who offered their endless love, support, and demands for pictures of my cat. There are too many of you to name individually, for which I am both grateful and humbled.

Finally, I would also like to thank the many people not listed here who have contributed in ways, big and small, to my academic and personal success. To name everyone who ultimately made this possible would take the full length of this dissertation, but please know that I am appreciative nonetheless.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Dylan Shell [advisor], Dezhen Song, and Shinjiro Sueda of the Department of Computer Science and Engineering, and Professor Daniel Selva of the Department of Aerospace Engineering.

The code used to generate the function shown in Figure 3.16 and initial solution shown in Figure 3.17 was provided by Professor Dylan Shell and used for a prior publication (O'Kane and Shell, 2017). Data for the layout of Palma de Mallorca Airport used in Chapter 3 was provided by Claire Liang of the Department of Computer Science at Cornell University.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a fellowship from Texas A&M University, by the National Science Foundation through awards IIS-1453652, IIS-1849249, and IIS-2034097, and from a graduate fellowship provided to Texas A&M University by the 3M Company and 3M Gives.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Robots are systems that merge both the physical and digital realms, and accordingly inherit both the advantages and problems of each. In addition to the normal challenges of hardware and software design, their union through robotics introduces a host of additional problems to consider. Figure 1.1 shows that over the lifespan of the robot, different problems arise: during the design phase, fundamental choices about the robot's capabilities will be made based on its anticipated use. Once a robot has been designed and built the changes that can be made to it are limited, and we will turn instead to modifying the environment to achieve desired behavior. During the robot's operational lifespan, failures and degradation of parts raises questions about the robot's ability to adapt to these changes, and if they impede task completion.

This dissertation will focus particularly on sensing: both how robots obtain information, and how information is relayed to robots. Robots must rely on noisy sensors and interpret the information available to them to determine their position within an environment, what action is appropriate to perform next, if an action was successful or not, and so on. The first step towards a robot's successful operation is ensuring that it is capable of obtaining the necessary information, for which we rely on robot designers to make informed choices.

Current robot designers create systems in an ad hoc manner, with each robot meant to fulfill the requirements of a given (and specific) problem. Designers are additionally constrained by budgets, energy requirements, and part availability. Due to this ad hoc approach, there exists no unified set of formal theories for robot design. However, the development of such an approach to robot design could have numerous benefits, such as enabling us to precisely understand the capabilities of robots, or to simplify the process of robot design. In this work, we attempt to take a step towards that ideal through the presentation of a principled collection of design theories and tools.

The approach of any principled theory for robotics must be applicable to the wide variety of forms robots can take, including the different sensing and actuation technologies in use. Accordingly, many approaches (such as those of O'Kane and LaValle (2008) and Sakcak et al. (2023))

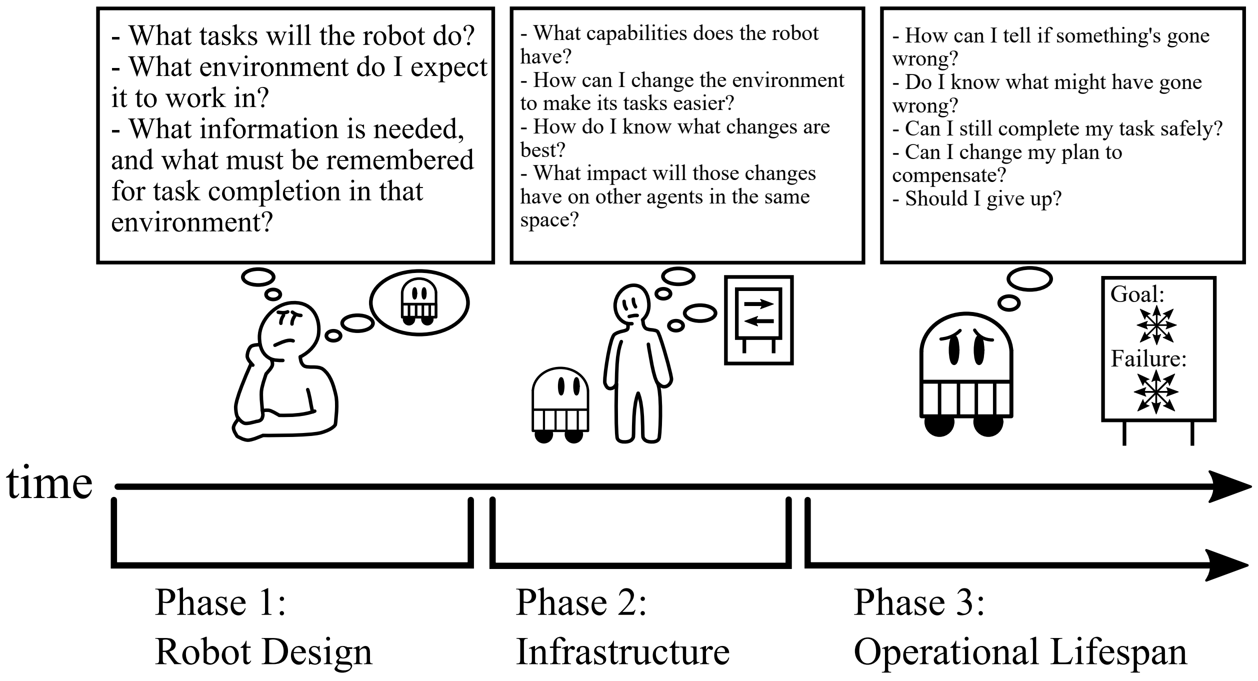

- What tasks will the robot do?
- What environment do I expect it to work in?
- What information is needed, and what must be remembered for task completion in that environment?
- What capabilities does the robot have?
- How can I change the environment to make its tasks easier?
- How do I know what changes are best?
- What impact will those changes have on other agents in the same space?
- How can I tell if something's gone wrong?
- Do I know what might have gone wrong?
- Can I still complete my task safely?
- Can I change my plan to compensate?
- Should I give up?
Goal:
Failure:
time
Phase 1: Robot Design
Phase 2: Infrastructure
Phase 3: Operational Lifespan

Figure 1.1: Problems of interest to designers and users change at different points in a robot's lifespan.

make use of an information-focused approach wherein comparison between robots is dependent on the way in which they obtain and manipulate information. This also allows for the specific implementation of a robot to be abstracted away into the way it interacts with an environment and obtains information.

To directly tie desired behavior to robot design, we make use of *action-based sensors* (Erdmann, 1995). These theoretical sensors relate observations to actions, defining what a robot must sense as what action needs to be taken next. We have extended this theory beyond the original special subset of plans considered by Erdmann, achieving completeness through the treatment of all plans (McFassel and Shell, 2020, 2022). In addition to generating large families of theoretic sensors, these sensors can not only be related to the physical sensors that designers can implement, but also reveal additional information about how the robot makes use of memory and state.

Properties of the environment are reflected in an agent's need (or lack thereof) to use memory and state. When the environment that a robot inhabits is structured, or is highly regular, this can reduce the requirements on the robot. When designing agents, structure in the environment can occasionally be exploited by agents to permit the use of simpler sensors or behaviors. It therefore follows that the environment itself can be considered as part of the design problem, providing another avenue to offload agent complexity through external information or control. The benefits of environmental design are well understood: QR codes and magnetic strips guide robots in warehouses, while IR beacons prevent robot vacuums from falling down stairs. As the environment offloads complexity from robots, it naturally encompasses additional functionality. At the extreme end of this process, the environment transitions from merely having an exploitable structure to becoming *infrastructure*, wherein the environment itself is a component of the functional system.

While the infrastructure within the environment a robot will work in is often considered by designers, it is rarely considered as part of the design process itself. To discuss infrastructure precisely, we will first present a collection of features common across most infrastructure, categorizing the many ways in which infrastructure can influence agents and their environments to obtain a deeper understanding of how interactions between agents and their world manifest. With this as our basis, we use Markov Decision Processes (MDPs) as a method for comparing the effect that the introduction of infrastructure has on agents operating within a space. This method "applies" infrastructure to an existing environment, allowing for the comparison of different proposed infrastructure and their impacts on various types of agents.

Within this dissertation, we will require an implementation-agnostic base by which to compare the abilities of agents (and infrastructure). To accomplish this, we make use of planning problems and plans as our foundation. This choice has numerous benefits for the work to follow: first, it allows for us to focus on agent behavior in terms of interaction and task achievement over the representation of technologies; second, our use of planning problems provides a common object through which changes to the environment and changes to the agent can be easily expressed without having to reconsider the overall model. Plans are rarely treated as a first-class object of study, but instead are often considered as something to be found by an algorithm targeting an existing robot. However, we can consider a planning problem as a model of the task itself, with a structure that contains information about the expected features of the robot's design. Suitable plans are directly

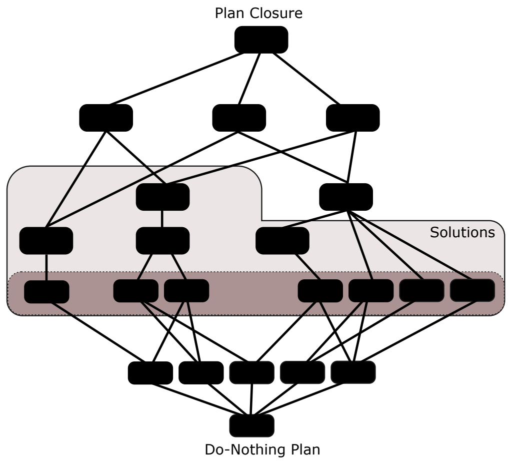


Figure 1.2: A summary of how the constructs to be presented in Chapter 3 relate to each other, showing that all plans for a planning problem can be related to each other through a formal set structure. Black boxes represent plans, while plans higher in the structure dominate those below them, indicating that the plan is able to perform the same behaviors as those it dominates. The shaded sections enclose the subset of plans that solve the planning problem, while the darker shading indicates the subset of these solutions that are action-based sensors.

impacted by the structure of the planning problem. Additionally, we treat plans as a way in which the designer can specify the desired behavior of an agent.

When considered in this way, it becomes beneficial to consider multiple plans so that the robot can exhibit a variety of behaviors, as opposed to any individual one. Figure 1.2 shows that all plans for a planning problem can be related to each other through a formal set structure, termed a lattice, making explicit the relations between how different plans behave. Up to this point, we have assumed sensors to be reliable objects that deliver truth about the world, or at least an approximation of truth with some additional noise. However, sensors may degrade in performance over their service life, and all hardware is capable of failing. When a robot is mid-task, the identification of sensor failures is critical to recognizing if a task can still be successfully completed, as well as determining how the robot must adjust its behavior to accommodate the failure. There may also exist multiple explanations that are consistent with the agent's prior observations up to the current point in time. Using insights from our work into action-based sensors and infrastructure, we will move from the design phase to the execution of tasks. We will present a method (and accompanying algorithm) through which a robot may perform a kind of "introspection," using known information about the world and its structure to identify inconsistencies between the information received and what is known to be possible.

Recognition of a fault is only one component of handling a failure that has occurred. Ideally, an agent would also be able to determine if the fault that has occurred will prevent it from completing its task. Such a determination requires the agent to have a representation of what information it will need to acquire in the future, as well as a way to determine if the current failure prevents the acquisition of that information. We present an algorithm for performing planning under sensor failure that determines if a task can still be completed; in addition to this, the algorithm can also be used before execution has begun to determine what sensing information is critical to task completion.

Figure 1.1 depicts how the three main contributions of this dissertation contribute to different aspects of agent design and behavior. The concerns of those who design and use robots vary over time between agent design, environment design, and agent runtime. We address these varying concerns through tying robot behavior to robot design. The three contributions of this dissertation can be summarized as follows: first, we extend the existing work of Erdmann (1995) to allow the definition of action-based sensors for all plans, and show that action-based sensors can be used to reason about design and state requirements. Second, we examine the problem from the opposite direction by developing a framework through which planning problems are modified to model potential impacts of infrastructure. Planning problems and plans also provide a common area through which the impacts of different infrastructure may be compared to each other. Finally, we demonstrate how a robot may use knowledge about the world encoded in a planning problem or plan to perform self-diagnostics during execution. We will show how this information can be concisely tracked, as well as how it can be used to determine if a task has become unachievable.

1.1 Contributions

1. **Contribution:** Generalization of Erdmann's work to identify all action-based sensors.

   1.1. *Sub-contribution:* Use of the connection between plans and action-based sensors to devise algorithms for plans which isolate and resolve obstructions to prior generalization.

   1.2. *Sub-contribution*: Exploration of the implications of this generalization on the handling of partial observability and state.

   1.3. *Sub-contribution*: Demonstration of the application of action-based sensors to design problems through case studies.

2. **Contribution:** Treatment of the problem of designing infrastructure shared by multiple robots, which is the dual problem to that of designing robots for their environment. Exploration of how infrastructure can be modeled as a transformation of planning problems.

   2.1. *Sub-contribution:* Introduction of a framework for assessing infrastructure based on the impact it has on agents.

   2.2. *Sub-contribution:* Definition of a taxonomy of infrastructure, along with validation of the framework on case studies which span this taxonomy.

3. **Contribution:** Development of efficient representations for combinatorial sets of sensors.

   3.1. *Sub-contribution:* Introduction of a structure called the plan lattice, allowing for both the generation of sets of action-based sensors and their expression as reactive plans.

   3.2. *Sub-contribution:* Tracking and identifying sensor failure through the design a concise representation of large sets of sensors.

   3.3. *Sub-contribution:* Presentation of algorithms to track this concise representation, as well as an algorithm for determining if a goal is still achievable after a failure is recognized.

1.2 Outline

The rest of this dissertation is organized as follows: Chapter 2 presents related work in several distinct areas: sensor design, infrastructure design, and identification of sensor failure. Chapter 3 presents an in-depth examination of one approach to understanding sensing information requirements (Erdmann, 1995), and identifies situations of interest to designers wherein existing theory is insufficient. It then introduces both conceptual extensions and an algorithm to obtain "missing" sensors. Chapter 4 first presents a definition of infrastructure based on features and behavior, and then presents an MDP-based model for comparing the behavior of agents before and after the introduction of a proposed infrastructure change. Chapter 5 presents an algorithm for identifying malfunctioning sensors, as well as an algorithm for determining if a task can still be achieved after a failure. Each chapter contains its own case studies to discuss the application and utility of the models and algorithms previously described. Chapter 6 concludes the dissertation with a discussion of known issues and future directions for research.

2. RELATED WORK

We find it useful to divide existing work into three broad categories: design of robots and sensors, design of environments and infrastructure, and detection of sensor failures. For the first category, we will focus on individual agents and present existing work that forms the theoretic foundation of the contributions discussed in Chapter 3. For the second category, we will shift focus to the design of environments, particularly when they contain multiple agents, and highlight open areas of research within this field that are addressed within Chapter 4. For the final category we will focus on the identification of sensor failures and existing recovery methods, providing context for the work presented in Chapter 5.

2.1 Robot and Sensor Design

The problem of designing sensor systems for robots is a sub-problem within general robot design, where the choice of sensor can incur different processing and power requirements that affect the overall system. Within this larger problem space, algorithmic approaches aim to reduce the burden of complexity for designers. A co-design approach makes use of descriptions of component capabilities and associated costs, which are balanced against other system considerations (Censi, 2015, 2016, 2017; Zardini et al., 2022; Carlone and Pinciroli, 2019). Such work can also extend to include non-physical components, such as overall system performance (quantified as the probability of success) or control systems (Zardini et al., 2021; Lahijanian et al., 2016; Saberifar et al., 2022). Joint human-computer approaches to design problems have also shown potential to generate better designs than either human or computer alone (Law et al., 2019). However, these approaches are constrained to libraries of existing components, and do not yield information on what the "best" sensor for a task would be outside of that library. To work on the design of sensors themselves, we require a deeper understanding of what sensing is.

The fundamental questions concerning sensing and information requirements first appeared in classical control as the notions of controllability and observability, with variants for both linear (Kalman and Bertram, 1959) and nonlinear systems (Hermann and Krener, 1977). In providing the context of their work, Hermann and Krener (1977) distinguish ways in which a space state may be "deficient" for the problem at hand: it may be "too small," and fail to capture the full relation between inputs and outputs. Conversely, (as they point out) a state space may be "too large," in which case it contains information other than that needed to model and solve the problem. These larger state spaces require more complex controls, or may be uncontrollable. Unlike that classic theory, which makes a range of algebraic assumptions (variously about linearity, smoothness, or merely regarding continuity), Chapter 3 takes an algorithmic approach to consider essentially discrete variants. However, both Hermann and Krener (1977) and the work presented in Chapter 3 share a common focus on sufficiency of information for controlling the system without tracking extraneous information.

Theoretical sensor design presents an avenue through which questions of information sufficiency can be addressed. Unlike physical design problems wherein a designer chooses between several existing sensors, theoretical sensor design is concerned with how information is obtained, represented, and used by a robot.[1] This requires first a common way to describe sensors, a problem complicated by the variety of technologies in use; accordingly, much foundational work in the area is focused on technology-independent information representation (Donald, 1995, 2012; O'Kane and LaValle, 2008). Donald (1995) approaches the idea through *information invariants*, a concept which "quantifies the tradeoff between speed, communication, and storage." Given a certain robot and its strategy for accomplishing a task, information invariants allow for comparison to another system that may have different technology and strategies. Sensors can then be compared on the basis of these invariants.

LaValle (2012, 2011, 2019) provides another approach for comparing sensors to each other. The sensor lattice described by LaValle (2019) presents sensors as partitions, defining a lattice which contains the set of all possible partitions of a given state space. An individual sensor divides

[1]For instance, imagine a robot with a camera and neural network that identifies likely faces from that input. Conceptually, we can call such an object a "face sensor," even though the camera is not a sensor that detects faces. Theoretical sensors explore the way in which sensors are abstracted into the concept of the information that they provide.

the environment into a subset of partitions based on where different sensor readings are obtained; this allows for a natural lattice structure to emerge such that sensors can be compared with each other, and boundaries on this lattice can be used to represent limitations on information (Zhang and Shell, 2021). A different method is modeling the processing of information through *filters*, which allows for a graph-based approach to how agents understand and partition their environment. This graph-based framework has been particularly useful for questions of minimization of plan and sensor complexity (Saberifar et al., 2016; O'Kane and Shell, 2017; Saberifar et al., 2019; Ghasemlou, 2020).

Of particular interest to us is the work of Erdmann (1995), which ignores the question of what information a robot should obtain or how a sensor should be implemented, and instead directly defines a sensor that outputs the next action the robot should perform. The definition of these action-based sensors requires a function known as a *progress measure*. The progress measure assigns a numeric value to each state in the world, such that actions that move from a higher-value state to a lower-value state correspond to progress towards task completion. Erdmann defines these progress measures in terms of backchained plans; Chapter 3 shows the impact of extending the definition to all possible plans. Work similar to action-based sensors includes sequential controllers (Wagner et al., 2016), in which different controllers cover parts of an environment and direct the agent from one controller to another, as well as later work exploring fundamental limits on what an agent must sense to guarantee task completion (McFassel and Shell, 2020, 2022; Majumdar and Pacelli, 2022).

Both Donald (1995) and Erdmann (1995) make use of the robot's strategy, or plan, as part of their definitions for evaluating the robot's performance and use of information. One reason for this is that plans can serve as a method to specify desired behavior, particularly for purely reactive robots where a plan captures a full description of the agent's behaviors (Schoppers, 1987). Additionally, the strategy of the robot has a direct effect on what information will be available for sensing during its execution, and therefore questions about what a robot must sense are directly related to the agent's behavior. Chapter 3 makes use of this framework, conceptually extending the problem to consider non-reactive behaviors.

The task-focused approaches to sensor design above prioritize the close link between the robot's desired operation and the sensors needed to accomplish that operation, allowing for a focus on designing robots which are well-suited to their tasks. Another approach to designing robots that fit their tasks is to simultaneously optimize controller and body design. Biologically-inspired work, such as that of Banarse et al. (2019) and Lipson and Pollack (2000), employ iterative evolutionary methods in the search of optimality, in each case with change driven by a pre-defined fitness metric. Pervan and Murphey (2021) take a different approach, beginning with either the sensor or actuator space, and then minimizing the other, also taking into consideration the question of design limitations, in which a designer may have a particular set of sensors or actuators to choose from. Majumdar and Pacelli (2022) use the Kullback–Leibler divergence to define what information a sensor can provide, which is then used to find a boundary on the reward obtainable from a Partially Observable Markov Decision Process (POMDP). Their work is similar to our own in that it examines the way in which choice of sensor affects task completion, however Majumdar and Pacelli (2022) is focused on optimality given a choice in sensor, aiming to both obtain a policy and determine the upper bound on the expected reward; this dissertation focuses on pre-defined desired behavior of an agent and determining the minimal amount of information necessary for task completion, without regard to a necessarily optimal policy.[2]

2.2 Environmental and Infrastructure Design

Robots can externalize information storage through modifying their environment, such as when placing pebbles on the ground (Blum and Kozen, 1978). Infrastructure can serve a similar purpose, as the modification of an environment can allow for multiple agents to externalize certain information and capabilities (O'Hara, 2011). In cases such as roads, infrastructure imposes rules both on how agents should act as well as how agents can expect others to act, providing a framework for managing more complicated scenarios (Li et al., 2019). Infrastructure designed for humans

[2] Chapter 3 considers several examples of non-optimal plans that are desirable for other features, such as allowing an agent to achieve a task with "worse" sensors.

such as roadways and signage systems have been successfully used by robots (Liang et al., 2020), suggesting that infrastructure designed with robots in mind could yield even greater benefits.

Most work on robot design does not consider the robot as a part embedded in a much broader system, itself amenable to design; a recent and notable exception is Zardini et al. (2020). However, the difficulty of modeling an agent's behavior directly depends on the influence of the environment on the agent, as well as the influence of other agents who may share that environment. It is this notion of *embeddedness* (Ferber and Müller, 1996; Edmonds, 1999; Dautenhahn et al., 2002) that makes infrastructure design a problem of interest. The more embedded an agent is within its environment, the more difficult it is to model that agent without knowledge of the larger system in which it exists. This concept is similar to those of *situatedness* and *embodiment* (Maes, 1993; Matarić, 2001; Matarić, 2002), which separate the interaction of an agent and a larger system into two categories. Situatedness is the idea that the agent is strongly affected by the environment and other agents which exist within it, while embodiment captures the idea that the agent is also capable of affecting change on the environment and other agents through its own actions. As a result of this entanglement between an agent and its environment, even infrastructure that has the capability to greatly simplify a robot's task can complicate the modeling process by requiring the inclusion of information about the infrastructure's behavior as well.

The subdiscipline of multi-agent systems (MAS) focuses on the coordination of autonomous individuals, as either physical or virtual systems (Dorri et al., 2018; Ferber, 1999; Resnick, 1994). In systems with a large number of agents, fully understanding and modeling the behavior of other agents may be unmanageable. A long line of robotics research has explored *stigmergic* multi-robot teams, taking the extreme approach of externalizing all relevant information to produce coordination (Payton et al., 2001, 2003; Ricci et al., 2006), including for the achievement of specific tasks (Beckers et al., 1994; Ijspeert et al., 2001; Fine and Shell, 2011, 2013; Matarić, 1993). This perspective, emphasizing the environment as something active or able to be exploited structurally (hosting common markers, or persistent shared computation) rather than merely being passive circumambient space, leads to new ways to coordinate robots (Vaughan et al., 2002; Fine and Shell, 2013; Bobadilla et al., 2012).

While infrastructure can be designed to support multi-agent systems, there also exist many cases in which the agents that make use of infrastructure are non-collaborative and lack common goals. Any coordination of agents that is achieved is imposed externally by the infrastructure itself. In cases such as traffic, agents will overall have *similar* goals and behavior, but their specific destinations vary, and the use of resources (such as road space) by an agent prevents another agent from using the same resource. Some drivers may also be more competitive, or even antagonistic, than others. The implication for this dissertation is that any framework to model infrastructure must be capable of accommodating systems that are cooperative, competitive, or consist of independent agents.

Unlike cars, which all share a similar set of abilities, other agents may have highly varied abilities and have objectives that directly conflict with each other. The first of these concerns, in which agents may have high variation in their capabilities, is the subject of ad hoc team research (Ravula et al., 2019; Stone et al., 2010), which nonetheless often still falls into the realm of assuming agents have shared or non-competing goals.

For the second of these concerns, we must consider models that allow for self-interested agents. Economic models provide one such framework (Sandholm, 1993; Gerkey and Matarić, 2002; Dias et al., 2006), and have the benefit of an existing body of work that has been directly applied to existing human infrastructure. Particularly, the economic theory of "clubs" (Sandler and Tschirhart, 1980), in which a group of members derive benefit from shared goods while dividing costs, can be used to model the shared nature of infrastructure. The framework of economic clubs also captures the fact that transition from a good owned by a single agent to one shared with a group can result in a drop in the utility for any single agent, and the resulting problem becomes one of finding a Pareto optimum (Buchanan, 1965).

Another framework for self-interested agents that can be applied to infrastructure is general game theory, wherein the rules of the game can be used to constrain and control agent behavior (Owen, 1982; LaValle and Hutchinson, 1993). Algorithmic mechanism design (Roughgarden,

2010) draws directly from game theory and economics to provide another way in which to model self-interested agents within networks. Algorithmic mechanism design is based in routing games as models of behavior, and uses incentives to guide self-interested agents to behave in a way beneficial to the overall system. Both data and vehicular traffic networks are archetypal examples for these kinds of problems. Traffic networks, of course, are a very visible example of infrastructure, with models going back to the 1950s (McNally, 2007) and increasing use of agent-based techniques (Kagho et al., 2020; Resnick, 1994).

Reinforcement learning has also been applied as a method to influence the behavior of agents, particularly in application to traffic networks using controller systems that can modify infrastructure over time. Such applications are capable of changing the pathing and behavior of individual agents (Mirzaei et al., 2018) as well as analyzing and responding to general trends such as congestion (Rasheed et al., 2020). Fundamentally, reinforcement learning examines how agents can operate (and adapt to changes in) an unknown environment. Most approaches to reinforcement learning make use of MDPs with a policy that evolves over time (Bertsekas, 2019). While such work is notable for producing desired agent behavior through infrastructure and controllers, it does not approach the problem of interest in Chapter 4; namely, it does not provide a unifying method by which we may compare implementations of infrastructure. Furthermore, while the MDP model and updated policies both feature in Chapter 4, our focus is on the impact of environment change after policies have converged, and not the transient phase progressing toward convergence.

2.3 Detecting Sensor Failures

The literature concerned with fault detection in robots is sizable, with several surveys that deal with multiple aspects of the topic (Khalastchi and Kalech, 2018; Steinbauer, 2013; Zhuohua et al., 2005; Pettersson, 2005). In a recent review, Khalastchi and Kalech (2018) provide a taxonomy of robotic failures, and organize the existing approaches along two axes: the first dealing with different types of robotic systems, and the second with different approaches (*data-driven*, *model-based*, *knowledge-based*). Another review by Abid et al. (2021) further sub-divides these approaches through the details of their implementations.

Dealing with the slightly broader topic of execution monitoring in robots, the earlier survey of Pettersson (2005) distinguishes fault detection (has some fault occurred?), fault isolation (what has gone wrong?), and fault identification (how serious is it?). While faults, generally, might involve any subsystem and its interaction with the environment, some work specializes on perception failures (e.g., tracking performance (Kalal et al., 2010)).

The approach we describe in Chapter 5 is model-based and concerns the three questions of Pettersson (2005) but with regards specifically to failure of (possibly multiple) sensors. Compared with the existing work, its primary novelty is the potential for compact representation of the full space of consistent hypotheses; additionally, no strong inductive bias is employed in the representation. As a discrete filter, its follows from kindred work on combinatorial filters (Tovar et al., 2014).

Early instances of effective fault detection and identification employed a bank of multiple models, for instance multiple concurrently executing Kalman filters (Roumeliotis et al., 1998; Goel et al., 2000; Hashimoto et al., 2001), along with some means for declaring behavior anomalous (e.g., such as thresholding error residuals). Multiple models allow the multi-hypotheses to be expressed (even if implicitly), and later, particle filters found effective use as they are a natural fit for such circumstances (Verma et al., 2004; Plagemann et al., 2006; Zapolsky and Drumwright, 2016). The work of Sadeghzadeh-Nokhodberiz and Poshtan (2017) considers multiple sensors (and single or multiple faults) and proceeds by decomposing the system into subsystems, which are then treated as distributed and interacting. In practice, robot systems have been shown to exhibit circumstances where there are multiple faults (e.g., Orrick et al. (1994)), and several techniques are specifically designed to tackle multiple failures (Price and Taylor, 1998).

Data-driven methods have also shown success for fault detection. The work of Sorsa et al. (1991) demonstrates an early application of neural networks to the diagnosis of mechanical faults. This area is closely tied to the realm of signal analysis: Shiroishi et al. (1997) demonstrates that the way in which a signal is processed has significant impact on identification of faults from vibrational signals. Samanta (2004) presents another early example of the use of classifiers being applied for mechanical fault diagnosis, focusing in particular on feature selection from signals. Work on detecting *sensor* faults through data-driven methods can be traced back to the work of Guo and Nurre (1991), set apart from mechanical faults through “sensor accommodation,” wherein the defective sensor’s readings are replaced with the output of a simulator for that sensor. Darvishi et al. (2021) provide a recent example of data-driven methods for detection and accommodation. Estimating (or simulating) the value of a non-functional sensor allows the system to continue without the influence of erroneous sensor readings, leading us into the notion of fault recovery.

A line of work, going back to the inaugural work of Srinivas (1977), has also tackled repair or recovery when a fault is detected. This has been a topic of recent interest, with work considering such problems in a variety of contexts including service robots (Hammond et al., 2019) and human-robot interaction (Chung and Cakmak, 2020; Reig et al., 2021). Lunze and Richter (2008) presents a review of the literature in adapting the behavior of a controller after a fault to recover nominal operation. Another review by Ciria et al. (2021) focuses on the emergent trend in the framework-agnostic concept of adaptive inference-inspired models, where model refinement is based on free energy minimization. The work of Pezzato et al. (2020a,b) applies this approach for controllers capable of adapting to dynamic environments, sensor noise, and faults. However, much of the work in fault recovery places the question of precise diagnosis second to the immediate recognition and compensation for a fault. Additionally, we are unaware of any work that considers how agents should behave when an unrecoverable fault occurs, nor how to identify these unrecoverable faults.

3. REACTIVITY AND STATEFULNESS: ACTION-BASED SENSORS, PLANS, AND NECESSARY STATE[*]

3.1 Introduction

In his venerable paper *Understanding Action and Sensing by Designing Action-Based Sensors*, Michael Erdmann[1] defines a class of abstract sensor that describes the information a sensor ought to provide a robot; his paper identifies a type of canonical choice for such ideal sensors. Summarizing that classic contribution to the literature, Donald (1995) writes:

> Erdmann (1995) demonstrates a method for synthesizing sensors from task specifications. The sensors have the property of being "optimal" or "minimal" in the sense that they convey exactly the information required for the control system to perform the task.

Action-based sensors embody the philosophy that sensors should be designed not to recognize states, only what actions must be taken to reach a goal. Utility in reaching goals is defined via *progress measures* and associated *progress cones*. These notions of progress are themselves computed from plans. Informally, the sequence goes like this: problems/tasks require plans to the solve them, plans give progress measures, measures give cones, and cones lead to sensors.

Erdmann's work focuses on a subclass of all plans, those created from backchaining, which yield "special" sensors. The backchained plans in his work codify the fastest way to reach the goal from any starting location. This naturally leads to the question of what would happen if one applied this method to other types of plans, as it would seem that doing so would open up a much larger family of action-based sensors. We shall show that this is indeed the case.

Part of our motivation for exploring new families of sensors is that generally, analyzing the

[*]This chapter is reprinted with permission and with material from: G. McFassel and D. A. Shell, Reactivity and statefulness: Action-based sensors, plans, and necessary state, The International Journal of Robotics Research. Copyright © 2022 The Author(s). DOI: 10.1177/02783649221078874.

[1]For clarity when reading, we often refer to Erdmann by name when making reference to his theory of action-based sensors. Unless otherwise indicated, this is a reference to Erdmann (1995).

information requirements of robotic tasks has yielded fundamental scientific insights in the past (cf. Blum and Kozen (1978); Donald (1995); O'Kane and LaValle (2008)). Construction of Erdmann's "minimal" sensors assumes that all information needed for task completion can be sensed in the environment. If the information available is insufficient, then the robot must make use of *state*. Through extension of the problem to other plans, we see that the required memory of the robot and the abilities of its sensors are linked in a way that different action-based sensors give a new, unique way to explore.

Moreover, making choices about sensors that are informed by information requirements is also important for practitioners, who need to balance considerations of cost, manufacturability, and reliability (Censi, 2015; Zhang and Shell, 2020). Considering additional plans (beyond solely backchained ones) helps make the theory of action-based sensors more applicable for roboticists by, for instance, allowing one to model some limits imposed by physical or technological constraints.

The incompleteness in Erdmann's existing theory, in not being able to obtain all action-based sensors, is irksome. This is true whether one's concern is primarily theoretical (previously overlooked sensors that have an equal claim to being "minimal") or is purely practical (sensors that can respect design constraints). The first part of this chapter, thus, is concerned with identifying further action-based sensors through considering the full set of plans for a given planning problem.

This analysis-oriented treatment differs from the typical (one might say synthesis-oriented) approach to plans, which asks how to find plans that realize some ends. Instead, the concept of a plan is used as the basis for the design of sensors. We present an algorithm, CLIP, that transforms a plan for which we cannot define an action-based sensor into a set of plans for which we can. We extend our earlier theoretical work (McFassel and Shell, 2020), and associated algorithmic methods and our implementation, to allow consideration of partially observable planning problems. Such cases lead to the identification of certain plans that are guaranteed to solve some given planning problem, but for which no Erdmann-like progress measure can be produced.

Such plans are the aforementioned plans that require state. To discuss the relation between these “stateful” plans and action-based sensors, we first define a lattice structure with which to organize the set of all plans. Action-based sensors are then placed within the lattice as a type of reactive plan. The action-based sensors to which a plan is related then become a basis for defining “stateful” sensors, further broadening the original concept of action-based sensors and allowing for one to examine state requirements alongside information requirements.

The approach followed in this chapter is, after some broader context and preliminary formalization of plans and planning problems generally (3.2 and 3.3), to reexamine Erdmann’s classic theory of action-based sensors through that lens (3.4 and 3.5). Because the generalized treatment we offer (especially with partial observability) helps make the theory more practical, we discuss relationships to problems of sensor design (3.6) and also LaValle’s sensor lattice concept (3.7). Having done that, the chapter pivots (3.8) so that action-based sensors, now in the guise of reactive plans, become a way to improve our understanding the space of plans (3.9–3.10), culminating in several examples (3.11), before concluding (3.12).

3.2 Context

This section provides some additional detail to relevant works presented in Chapter 2, as well as additional context. Our motivation for revisiting action-based sensors stems from an interest in what sensing, fundamentally, *is*. Often we take sensors for granted: as a distance sensor, or wall sensor, and so on. But as Brooks and Matarić (1993) note: “The data delivered by sensors are not direct descriptions of the world. They do not directly provide high level object descriptions and their relationships.” How easily we say that a sensor can detect “walls”! These mental categories are ingrained so deeply as to have a pernicious influence on our thinking.

In conceiving his theory, Erdmann (1995) asked the question of what sensors are *for*. The action-based sensor, then, relates what a robot should do with what it needs to perceive. The approach conceptualizes sensors as abstractions which entirely sidestep issues with the representation of information to provide what is required: what action to take next.[2] His definition appeared

[2]More than any other robotics work of which we are aware, Erdmann’s theory embodies the perspective of Pragmatism, in the William James and John Dewey sense of the term.

to give the utmost leeway in its requirements, being most relaxed or unconstrained so the set of sensors seems to be maximally inclusive — forming a sort of “free object” for sensors. It is hardly surprising, then, that little work has sought to expand directly upon Erdmann’s highly-original paper, for it looks to be the final word on the subject.

Recalling the biologically-inspired works presented in Chapter 2 (such as the work of Banarse et al. (2019), Lipson and Pollack (2000) and Pervan and Murphey (2021)), we see that these recent works share overall goals similar to our own within this chapter: each demonstrates how to make effective use of external constraints. In our case, we will show how Erdmann’s action-based sensor approach might also incorporate constraints (by defining planning problems —and therefore obtaining plans— which exclude certain movement or sensing) so that these affect the corresponding robot designs, limiting how they should behave or what can be sensed.

Section 3.7 presents a more in-depth look at LaValle’s sensor lattice (LaValle, 2019) first presented in Chapter 2, allowing for precise comparison of how sensors may be stronger or weaker than another, or if one sensor can be used to solve a problem instead of another. We show that his sensor lattice also contains the set of action-based sensors within it, and use it as an inspiration for a lattice with which to compare plans by their executions.

Through the approach of designing action-based sensors we will re-cast several questions, such as what information a robot must distinguish, as well as how to act at states that appear identical when observed by the robot. Subsection 3.10.2 focuses on how a state space can be extended through a triple relation that relates specific executions to states; this triple relation can then be treated as an extended state space, containing information that would have been lost within the original world’s state space. Action-based sensors, by construction, reduce the state space through their conflation of different states in the world, with states grouped together by the lack of a need to distinguish them, akin to the equivalence relations explained in Hermann and Krener (1977).

Orthogonal to the problem of robot design is the design of environments that include information cues. Liang et al. (2020) presents a method for robot navigation using signage intended for humans. The robot’s ability to navigate via signs, without any internal map, depends on certain properties of the signage. Properties include consistency between the individual signs, as well as whether there is a sufficiency of information to reach any destination from an arbitrary starting point. If the signage fails to have these properties, then that information must be supplemented with “virtual signs”. Their ideas in the specialized context of navigational signs correspond closely to properties we consider more generally—e.g., the idea of information being adequate to make progress. In fact, the algorithms presented in this chapter can be applied to that practical context. Specifically, 3.11.3 analyzes an example from Liang et al. (2020). This provides clear, self-contained, practical use for the algorithms in an interesting robotics setting, and supplements their perspective. By doing so, deficiencies in the environment’s signage that would interfere with navigation (that is, locations that require “virtual signs”) translate into lower bounds on the complexity of plans required to solve the planning problem, indicating that the robot must make use of information beyond what it can observe.

Ultimately, the close link between what a robot must do and how it must be designed to do it impacts a broad spectrum of problems within the robotics community. Extending Erdmann’s work to a larger family of plans allows for the concept of action-based sensors to be applied where previously it could not. This provides designers and theorists with another tool for analysis of information requirements, particularly with the theory now being applicable to plans for which the robot must keep state.

3.3 Preliminaries

In this chapter, the environment and robot inhabiting it are described in terms of two symmetric structures: planning problems and plans. The former defines both the world and task, while the latter defines the robot and its operation. Mutual interaction between the world and the robot determines whether the plan solves the planning problem. Operating in discrete time, a robot receives observations and chooses an action to take. This action is then performed upon the world, which has its own structure that decides the action’s outcome. This outcome then determines what observations are next received by the robot. Both can be conceived of, speaking intuitively about casualty, as instances of “choice.” Thus, there is a back-and-forth between the choice the robot makes (the action selected to be executed), and the choice the world makes (the action's outcome and subsequent observation). A bipartite graph called a *procrustean graph*, or *p-graph*, will be used to formalize these aspects next. Definitions 1–4 are slightly less general versions of those from Saberifar et al. (2019); we refer the reader to that original reference for more comprehensive discussion.

Definition 1 (p-graph (Saberifar et al., 2019))**.** *A* procrustean graph (p-graph) $P = (V, V_{\text{init}}, Y, U, E)$ *is a finite edge-labeled bipartite directed graph in which:*

1. *a finite vertex set* V *can be partitioned into two disjoint subsets, called the* action vertices $\mathrm{V_u}$ *and the* observation vertices $\mathrm{V_y}$*, with* $\mathrm{V = V_u \cup V_y}$*,*
2. *a non-empty set of vertices (*$\mathrm{V_{init} \subseteq V_y}$*) are designated as* initial vertices*,*
3. *each edge* $e \in E$ *originating at an observation vertex is labeled with a set of observations* $\mathrm{Y(e) \subseteq Y}$ *and leads to an action vertex,*
4. *each edge* $e \in E$ *originating at an action vertex is labeled with a set of actions* $\mathrm{U(e) \subseteq U}$ *and leads to an observation vertex.*

With a p-graph, we can model both planning problems and plans. A planning problem (or, as we write interchangeably, *world*) models goal attainment tasks that require the robot to arrive in some condition in the world.

Definition 2 (planning problem)**.** *A* planning problem $W = (V, V_{\text{init}}, Y, U, E, V_{\text{goal}})$ *is a p-graph* W *equipped with a* goal region $V_{\text{goal}} \subseteq V$.

A plan prescribes actions for particular circumstances in order to solve planning problems. As it is a directed graph, it is potentially governed by internal state, encoded directly into its branching structure.[3]

Definition 3 (plan)**.** *A* plan $P = (V, V_{\text{init}}, Y, U, E, V_{\text{term}})$ *is a p-graph* P *equipped with a* termination region $V_{\text{term}} \subseteq V$.

Since both planning problems (i.e., worlds) and plans are p-graphs with similar structures, we will use (W) and (P) respectively to help designate to which p-graph some thing belongs, e.g. the vertices $V(W)$, or $V(P)$. Although it is not strictly necessary given the scoping implicit in the definitions above, we will persist with this convention.

The preceding two definitions have exactly identical form: a p-graph and a set of vertices. The p-graph describes dynamics, while the set of vertices describe some notion of termination. A termination semantics (made precise in Definition 4) ties these two objects together, and differs slightly between plan and planning problem. It is expressed in terms of an *execution*, a sequence of alternating observations and actions, and (for this chapter) always beginning with an observation, or is the empty sequence. We trace execution $s = y_0u_1y_1u_2 \ldots y_n$ over a p-graph Q by beginning at some vertex in $V_{\text{init}}(Q)$, and following an edge labeled with some set containing the observation y_0 in the execution, and proceeding by following an edge labeled with a set containing action u_1, and so on for the whole sequence. If an execution is possible on both plan and world, it is a *joint-execution*. Joint executions describe the dynamic intermeshing of the plan and the world. We desire that either such a sequence must lead to vertices that are in V_{term} on the plan and in V_{goal} on the world, or the execution must be a prefix of a longer execution which does.

Definition 4 (solves). *A plan (P, V_{term})* solves *the planning problem (W, V_{goal}) if:*

1. *P is* finite on W. *The length of all joint-executions must be bounded.*

2. *P is* safe *on W. In tracing executions, P must never have an action leaving a plan vertex if there is no outgoing edge with that action at the vertex reached in the world. Additionally, the plan must always be ready to receive observations as can arise from possible executions on W, starting at $V_{\text{init}}(W)$, with actions chosen via P.*

3. *P is* live. *Every joint-execution $y_0u_1 \cdots y_k$ or $y_0u_1 \cdots u_k$ of P on W either reaches a vertex in V_{term}, or is a prefix of some execution that reaches V_{term} and, moreover, all the joint-*

[3]Within this chapter, the word 'state' has been used with extreme care thus far and only in precisely one form, namely to mean something memorized, tracked, or remembered; within the p-graph we have called the elements vertices; other words like "some condition" and "particular circumstances" have been deliberately used to avoid the conceit of declaring something to be 'state' and taking its existence so seriously as to believe it an objective concept.

executions reaching a vertex $v \in V_{\mathrm{term}}(P)$, when traced on W must reach a vertex $w \in V_{\mathrm{goal}}(W)$.

Occasionally we will be interested in the vertices reached by some given execution. To find these vertices, one traces the execution following the process outlined above, and those vertices reached by tracing sequence s on Q will be designated $\mathcal{V}_s^Q$. When tracing any such sequence, if, as one proceeds, at most one edge can be traversed, the p-graph is *deterministic*. Otherwise it is *nondeterministic*. The *language* of a p-graph Q (or plan and world), denoted $\mathcal{L}(Q)$, is the set of all executions.

3.3.1 Example

We give an extended example to make the preceding definitions more concrete. Figure 3.1 depicts part of the University of Freiburg campus via a map constructed using a robot equipped with a laser rangefinder (Stachniss and Grisetti, 2010), and available as part of the Robotics Data Set Repository (Howard and Roy, 2003). Shown superimposed is an (approximate) generalized Voronoi graph (Choset and Burdick, 1995) that has been constructed to help reduce complexity and aid with discussion of the space. We will assume for the purposes of this example that the Voronoi graph vertices correspond to locations that our hypothetical robot can clearly and reliably distinguish from each other, e.g., using unique range signatures. We will suppose that our robot can start anywhere in the environment and we wish for it to reach the position represented by the rightmost vertex.

The p-graph depicted in Figure 3.2 shows this scenario modeled as a planning problem. In this example, each point on the original figure has become an observation vertex (shown as filled circles), which has an associated action vertex (shown as white squares). At each observation vertex, the transition for the robot is determined by what observation is received from the world. Owing to the ability to reliably distinguish locations, here edges departing any circular vertex will bear a label directly corresponding to that vertex. At each action vertex, the robot can select an action that labels any of the outgoing edges. For this example, they are the cardinal and intercardinal directions, although sophisticated motion primitives might also be involved.

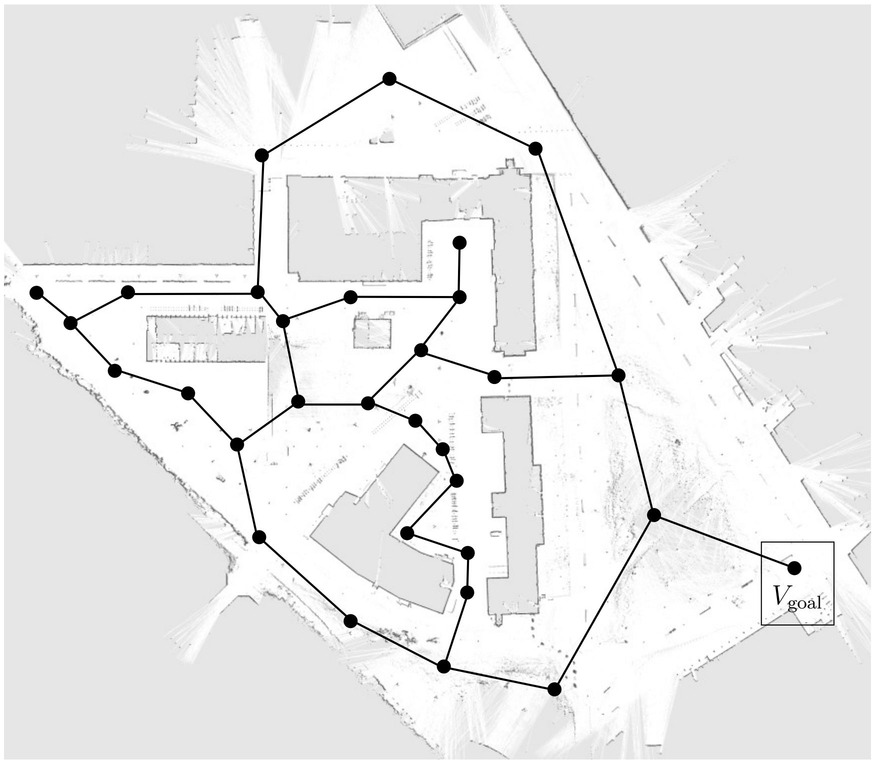


Figure 3.1: Initial working example: A mobile robot navigates in a planar environment amongst obstacles. For simplicity, a generalized Voronoi graph is superimposed over the top; the robot can start anywhere and we desire that it navigate toward the region represented by the rightmost vertex. (This is part of the University of Freiburg campus, with thanks to Cyrill Stachniss and Giorgio Grisetti for making the dataset available.)

Figure 3.3a and Figure 3.3b show different kinds of plans for reaching the rightmost vertex (the goal). These plans specify a certain subset of actions available from the original planning problem—each selects actions for the robot to take at different locations within the world. Figure 3.3a shows a plan derived through backchaining from the goal, while Figure 3.3b shows a more arbitrary plan. Looking at these plans, the backchained plan's routes are significantly shorter.

3.3.2 Generality, Scope, and Problem Variations

Quite apart from cluttering the diagram, the use of separate action and observation vertices in the previous example seems unnecessary. So too, the fact that plan's actions are not simply shown as direct prescriptions on the planning problem itself; perhaps selecting and highlighting a subgraph of Figure 3.2 could have sufficed to describe Figure 3.3a? But the preceding definitions are rather more general, and that generality will be put to use. The definitions permit, for instance,

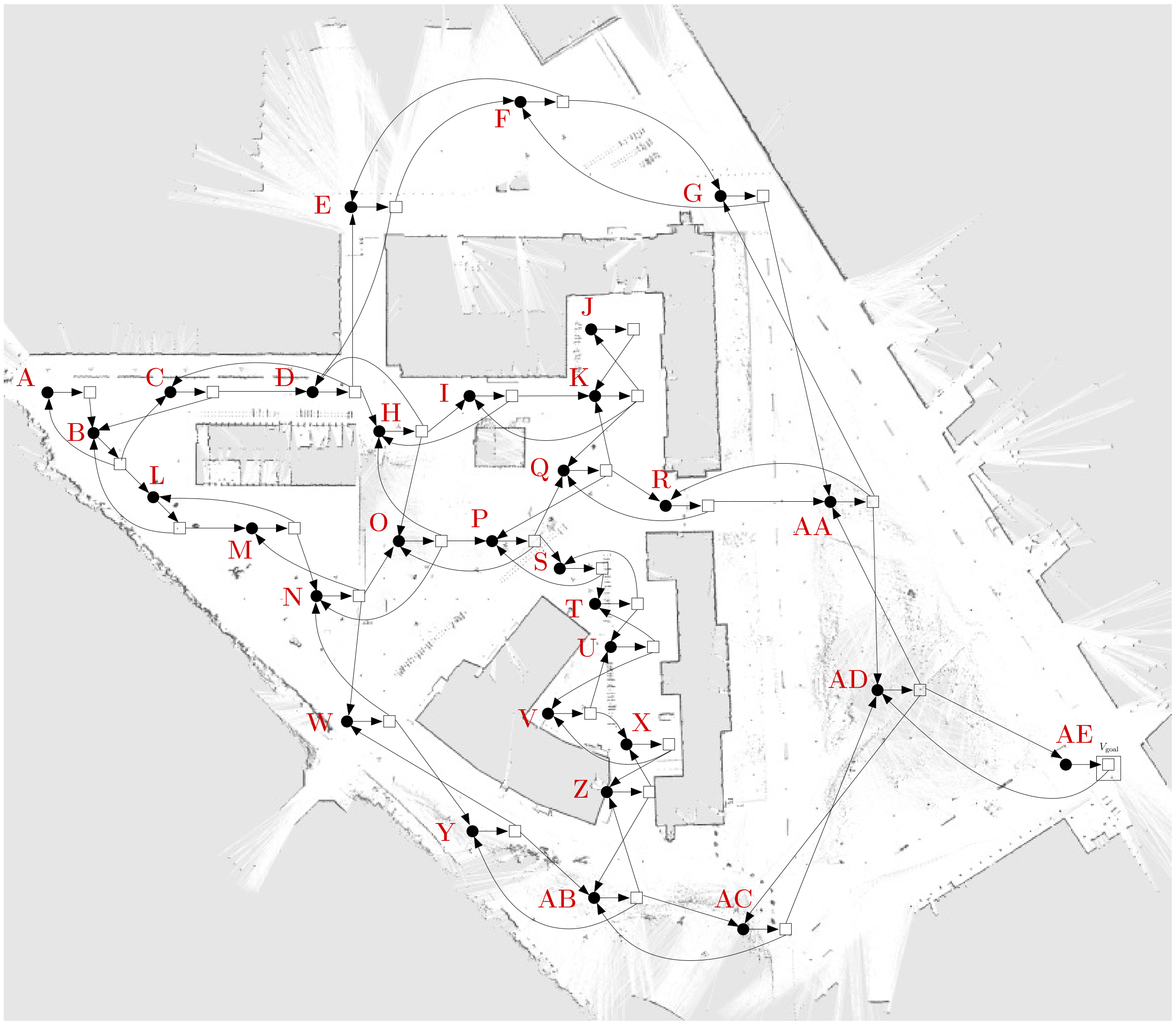
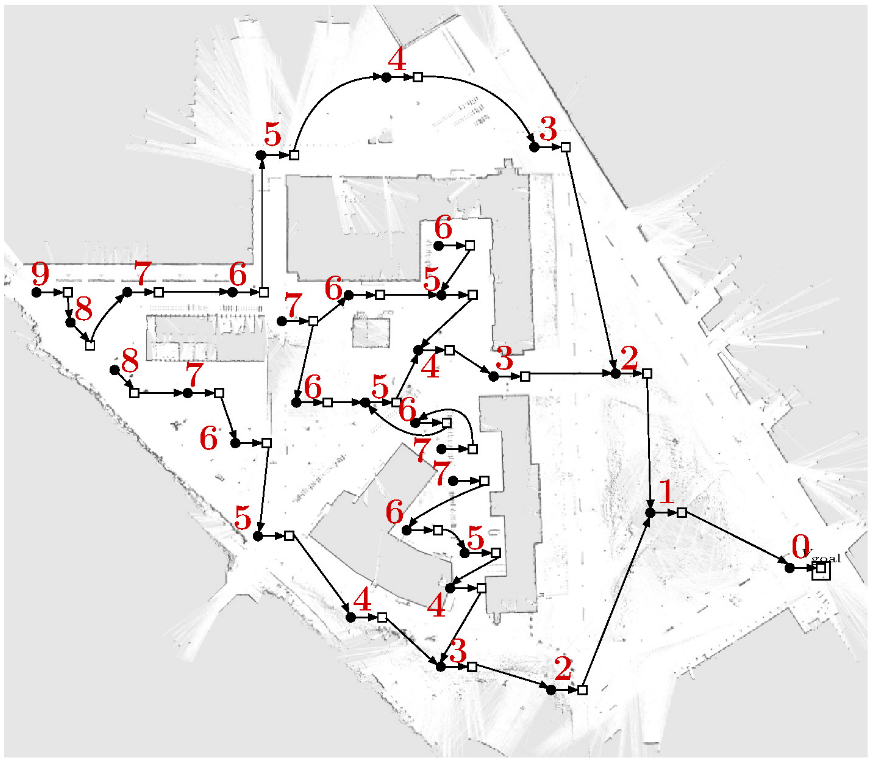
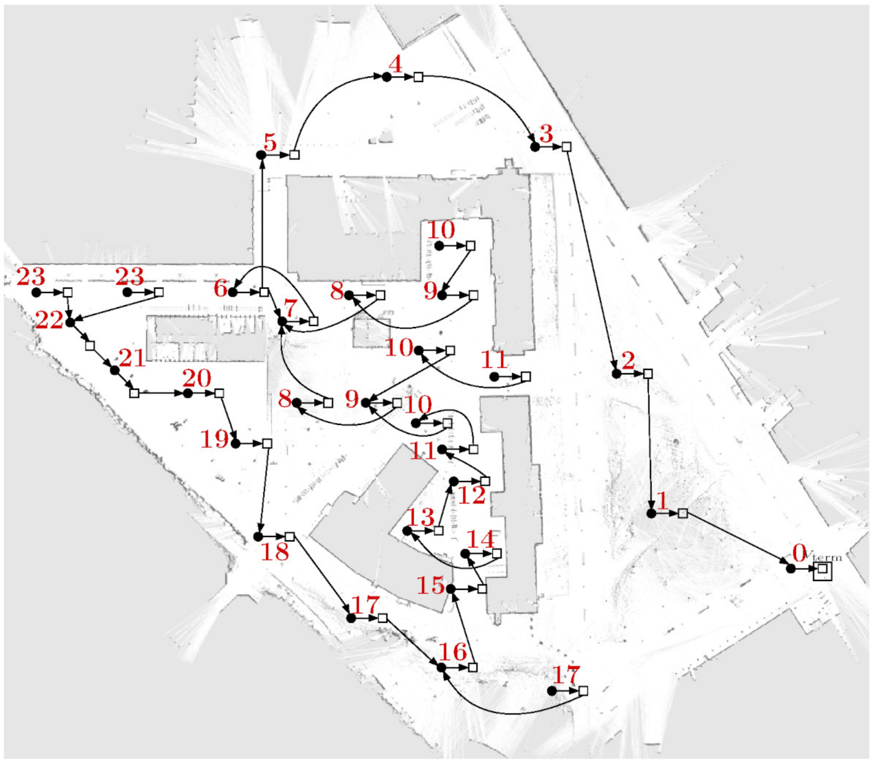


Figure 3.2: The example world from Figure 3.1 as a p-graph with goal region. Here, we consider every observation vertex (filled circle) to be a possible initial configuration. Observation vertices give distinct observations as output, while the actions include movement in the cardinal and intercardinal directions. Labels showing these have been omitted for clarity of the figure.



(a) A plan derived via backchaining from the goal.



(b) A different set of choices that also always reach the goal.

Figure 3.3: Two plans, as p-graphs with termination regions, that solve the planning problem in Figure 3.2. Progress measures (Section 3.4) are labeled in red.

the plan to track “beliefs” about potential world vertices that are consistent with the execution sequence (for instance, consider a nondeterministic world p-graph with nondeterminism arising from aliased sensor readings, unreliable action outcomes, or both). Later, important conditions will be examined in detail when the resemblance between plan and planning problem is limited. The example above emphasizes familiarity, but the added generality will come to the fore as we examine more exotic plans which yield additional action-based sensors.

Without further forestalling, we add some detail here by touching particularly on considerations of observability. In Figure 3.2, the p-graph has observation and action vertices which form a direct correspondence with the original points on the graph in Figure 3.1. For this example, each observation vertex yields a distinct observation, allowing the robot to localize itself directly throughout its execution via the last observation received. Full observability, as here where observations uniquely map to vertices in the world, simplifies matters in searching for (and expressing) a plan, because plans need only prescribe actions on the basis of vertices. This is the familiar understanding of full observability in regard to planning problems when considering plans as objects that we (or our algorithms) seek.

As we shall see, action-based sensors partition the world into regions which can be conflated without risk of task failure. Plans can yield action-based sensors and when the underlying planning problem is fully observable, this observability expresses a different aspect, though still in the nature of a simplification. Instead, it places no *a priori* limits on what the robot could in principle distinguish. When an action-based sensor conflates regions of the world as partitions, it represents a degree of acceptable additional partial observability. To apply the theory when there are known limitations owing to features of the environment, or because of other considerations (e.g., technological, practical resource limits, or design constraints), then partial observability can be used to specify particular perceptual limits. Modeling the constraints of existing sensors results in a different interpretation: the existence of an action-based sensor then implies that the existing sensor can enable the robot to complete the task without additional information.

We will further discuss impact of partial observability in defining progress measures and deriving action-based sensors. Here we have emphasized that it is a mistake to believe that because the preceding example is fully observable, the theory which follows does not consider partial observability. In fact, action-based sensors are directly concerned with giving a degree of non-destructive partial observability. Starting in Section 3.8, a subclass of plans for which no action-based sensors exist, and for which none can be derived, is considered. This non-existence can be directly tied to the requirement of additional information which a sensor cannot provide, and hence to complete the task, plans will require internal state.

3.4 The Progress Measure

In his work, Erdmann's sensors are defined using *progress measures*, which are real-valued functions on the state space of a planning problem that indicate how movement between states leads toward a goal. Given such a function, for each action, one labels regions of state space where that action makes progress, forming what are called progress cones. These regions must be distinguished sufficiently for the robot to determine which action to execute. This can be realized via action-based sensors, sensors that output actions guaranteed to make progress, which describe a subset of the progress cones containing the current state. As an abstraction of information attainment, such sensors do not specify which environmental features or associated technologies are actually used to compute (or evaluate) these functions. Erdmann formalizes the idea that the information a robot needs is precisely and solely that which is needed to determine how to act now.

To determine how plans make progress toward a goal, we start with defining what it means to make progress. Erdmann uses a framework of progress measures to develop progress cones. Given a task, to get from planning problems via progress to sensors, Erdmann (1995) prescribes:

“ 1a. Determine a sequence of actions that accomplishes the task.

1b. Define a progress measure on the state space that measures how far the task is from completion, relative to the plan just developed.

1c. For each action, compute the region in state space at which the action makes progress.”

The first step requires one to construct a sequence of actions, and subsequently in his paper Erdmann uses a very specific kind of plan when discussing progress measures — those obtained via backchaining from the goal. However, plans created using backchaining yield a unique progress measure corresponding to the fastest strategy which Erdmann calls "very special". Our agenda is to broaden this set of plans, and doing so has implications for progress measures. In particular, we now extend this definition to obtain a little more nuance, which manifests as two separate definitions in what follows. (For any set A, we denote its powerset with 2^A.)

Definition 5 (execution progress measure)**.** *A* progress measure over executions *on a solution P to a planning problem W is a function $\phi : 2^{V(W)} \to \mathbb{R}^+$ such that:*

a) $\phi(V) = 0 \implies V \subseteq V_{\text{goal}}$;

b) at least one $V \subseteq V_{\text{goal}}$ satisfies $\phi(V) = 0$;

c) for any two joint-executions p and q, if p is a prefix of q, then $\phi(\mathcal{V}_p^W) > \phi(\mathcal{V}_q^W)$.

The execution progress measure applies to sets of vertices in the world. All sets that take the value of 0 are required to have only goals within them, and there must also be at least one goal with value 0. We restrict joint-executions, requiring that if one is a prefix of another, its value must be strictly greater. In this way, the executions of the plan visit vertices in the world such that the resulting progress measure is strictly decreasing.

Definition 6 (vertex progress measure)**.** *A* progress measure over vertices *on a solution P to a planning problem W is a function $g : V(W) \to \mathbb{R}^+$ such that $\phi_g(V) := \max_{w \in V} \{g(w)\}$ is an execution progress measure.*

The intuition here is to give a measure on singleton vertices and require that we get an execution progress measure when it is lifted, in a natural way, to sets. When discussing the existence and properties of either kind of progress measure, we will simply use *progress measure*; when necessary, we disambiguate between an *execution progress measure* or *vertex progress measure*, or provide context to resolve any ambiguity.

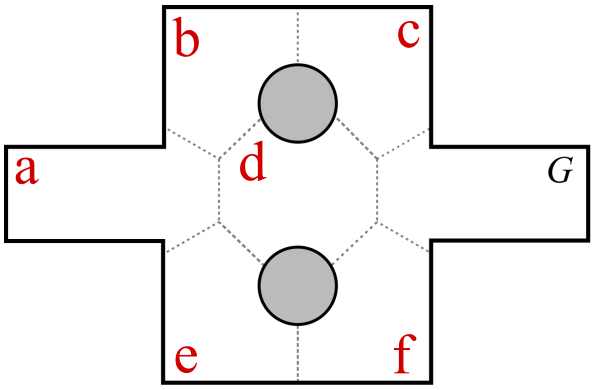
Progress measures can be seen in the red numerals in Figures 3.3a and 3.3b. Progress measures are defined in terms of the plan's actions on the world, which can lead to a measure in which making progress leads away from the goal (in terms of increasing physical straight line distance) before reaching it. Such an example can be seen in Figure 3.3b, exemplifying the difference between a progress measure and a metric of physical distance from the goal region.

3.4.1 Progress Measures and Crossovers: Lack of Uniqueness and Existence

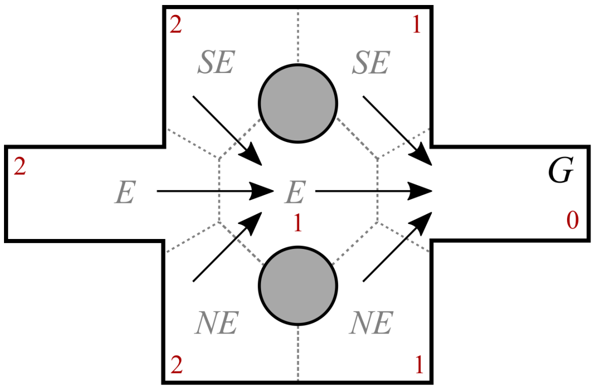
For clarity in the following section, we will put aside the planning problem in Figure 3.2 to consider a minimal example. Figure 3.4 shows a small, seven-vertex planning problem with a single goal G, as well as three plans which are able to solve it from any starting region in the world. The three plans here give three different progress measures; the plan obtained by backchaining, as in Erdmann's work, is but one choice.

The pair of plans in Figures 3.4c and 3.4d are worth contrasting. These two plans have some vertices where they take the same action, and some where they differ. There is little intrinsic reason to prefer one over the other, and we could even have a plan that considers both of the routes such as the plan in Figure 3.5, which chooses one of the two routes arbitrarily (via nondeterminism) and commits to that route once it has been selected. Even within this small example, the plans will construct contradictory progress measures, and so the plan which combines them has no progress measure at all. This stems from the fact that, though the plan informs our definition of progress, the progress measure is ultimately defined on vertices in the world.

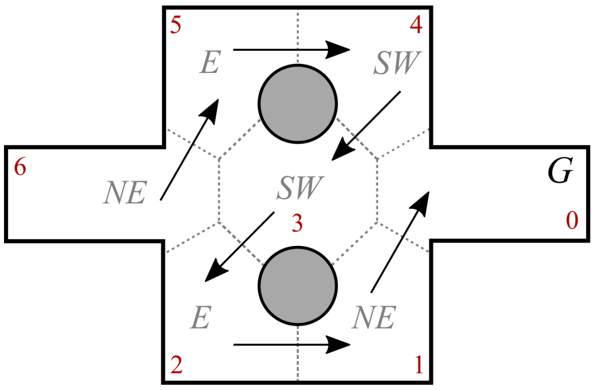
Figure 3.5's plan contains executions that go in opposite directions from each other; for example, the 'Z' route (which represents the movement according to Figure 3.4c) visits the location **b** before **d**; however the 'Σ' motion (akin to Figure 3.4d) visits **d** before **b**. For the p-graph to represent a solution, the finiteness requirement means it must be structured such that the robot could never actually cycle infinitely. Indeed, it meets this requirement. But the progress measure, considering only the corresponding world vertices, has the impossible task of satisfying $\phi(\{\mathbf{b}\}) < \phi(\{\mathbf{d}\}) < \phi(\{\mathbf{b}\})$. The extension to a broader family of plans has introduced new complications in defining progress measures.



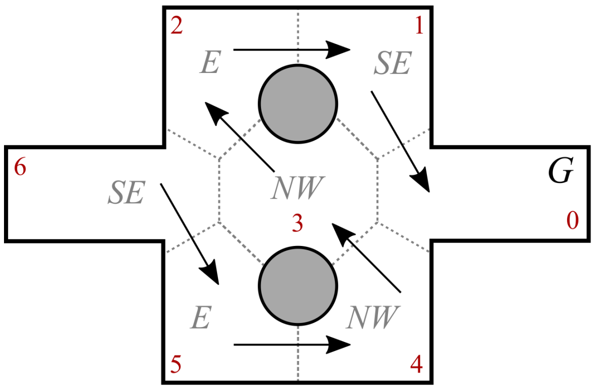
(a) A small world.



(b) A solution derived via backchaining.



(c) A plan which solves the planning problem of reaching the goal, G, from any vertex in the world.



(d) Another solution.

Figure 3.4: A small world with goal vertex G, along with three plans that solve it. As we will see, this is sufficient to demonstrate the existence and handling of crossover conflicts. For each plan, the red integers are a progress measure.

Progress measures fail to exist when there are contradictory requirements on the values that the execution progress measure must take. We refer to this issue as a *crossover conflict*, or simply as the existence of *crossovers*, due to the fact that the lack of progress measure extends from the fact that plan executions "cross over" each others' paths when traced over the world. Crossovers can be thought of as cyclic dependencies induced on the world by the plan. If a plan is a solution (like Figure 3.5) then the set of joint executions (or, equivalently, the intersection of its language with that of the world) must be finite. To re-visit a vertex in the world, such a plan can make use of multiple vertices to distinguish executions. However, the progress measure's definition considers all potential paths to and from a world vertex. Therefore, executions that visit vertices in differing

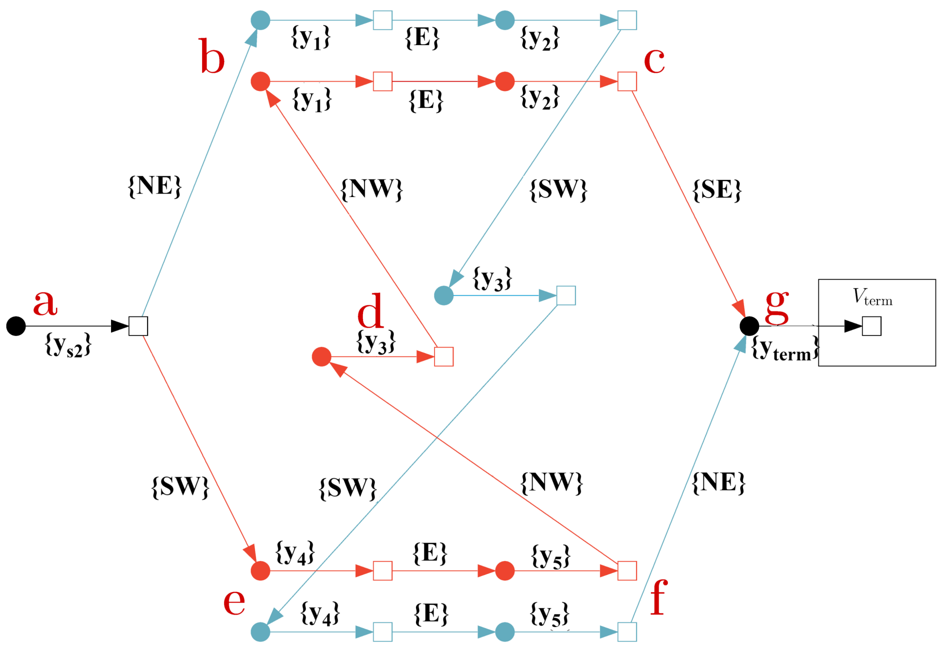


Figure 3.5: A single plan in p-graph form that solves the problem of reaching the goal in Figure 3.4a. The letters **a**–**g** here serve to expose the correspondence to locations in the world. It includes both the 'Z' route of Figure 3.4c and the 'Σ' one of Figure 3.4d. Every (circular) observation vertex is in $V_{\rm init}$ and, at the start of its execution, the robot receives an observation that can be traced to no more than two possible observation vertices. Once an action is chosen (the choice is resolved arbitrarily), the robot is ether committed to the blue or the red subgraphs. This plan induces numerous crossovers, arising from the lack of a global ordering on when vertices are visited.

orders create a cyclic dependency.

Definition 7 (crossover conflict)**.** *A plan P that solves a planning problem W has a* crossover conflict *if there are two distinct joint-executions s_1, s_2 such that:*

1) s_1 and s_2 both visit the world vertices v and v', and

2) s_1 requires an execution progress measure where $\phi(\{v\}) > \phi(\{v'\})$, while s_2 requires an execution progress measure where $\phi(\{v'\}) > \phi(\{v\})$.

Crossovers are the primary cause of failure in a plan that does not produce a progress measure.

Theorem 1. *A progress measure exists if and only if there is no crossover within the plan.*

Proof.

$\Longleftarrow$ **A Crossover Implies no Progress Measure Exists.** By definition, a crossover involves two joint-executions that induce the ordering $\phi(\{v\}) > \phi(\{v'\}) > \phi(\{v\})$. There exist no two numbers a, b that satisfy this condition unless $a = b$. However, this violates the definition of a progress measure. The definition of a crossover also requires that under a certain execution, v is a prefix of v', while under another execution v' is a prefix of v. Definition 5 requires that vertices reached by a prefix of an execution take values that are strictly greater than those of the vertices which follow. Therefore, there exists no pair of values which construct a progress measure for these two vertices.

$\Longrightarrow$ **The Lack of a Progress Measure Implies a Crossover Exists.** Consider a plan P that solves a planning problem W, and which lacks a progress measure. Then, by definition of the execution progress measure, one of the following must be true:

(a) $\phi(V) = 0, V \not\subseteq V_{\text{goal}}$,

(b) there is no $V \subseteq V_{\text{goal}}$ where $\phi(V) = 0$,

(c) there exist joint-executions s_1 and s_2 with s_1 a prefix of s_2, and $\phi(\mathcal{V}^W_{s_1}) \leq \phi(\mathcal{V}^W_{s_2})$.

For a progress measure it is enough to conceive of an ordering on the vertices of the world. At least one goal must come last in the ordering. For executions, the requirement of prefixes having a higher measure than sequences which they precede imposes an ordering on those vertices, as well.

To attempt to fix the progress measure, we first assign a goal (the final element in the ordering) value 0 and increment up as one goes earlier in the order. This resolves the issues presented by (a) and (b), should they exist.

Assume we try to correct (c) in such a way. If we are able to do so, obeying the induced ordering and assigning values to the vertices reached by s_1 and s_2 such that they no longer fulfill the condition of (c), then a progress measure does in fact exist. Otherwise, however, because we were assigning values to the ordering of vertices based on back-tracking from the goal, we must have seen the vertex reached by s_1 in the ordering before the vertex reached by s_2 and assigned it a value accordingly. If s_1 is a prefix, that means the vertex reached by s_1

is visited by an execution both before *and* after it visits the vertex reached by s_2. Therefore, there is an unsatisfiable requirement of $\phi(\mathcal{V}_{s_1}^W) \leq \phi(\mathcal{V}_{s_2}^W) \leq \phi(\mathcal{V}_{s_1}^W)$, which is the definition of a crossover.

□

The presence of crossovers is the necessary and sufficient condition for the non-existence of a progress measure, and their existence impedes our ability to craft action-based sensors. To identify crossovers, the relationship between plan vertices, plan executions, and world vertices must be made explicit, ideally without directly enumerating the plan's language. Next, we introduce an algorithm, based on a graph product construction, that is useful in this regard.

3.5 Removing Crossovers and Enumerating Progress Measures

Given a plan that solves a planning problem and which has no progress measure, we now discuss a method to produce the set of all plans which can be derived from this initial plan, which have progress measures, and which are also solutions to the planning problem. To make precise what we mean when we say one plan is *derived from* another, we define a set of actions from the world called the *operative action* set.

Definition 8 (operative actions)**.** *For a plan P that solves planning problem W, an action u_k is an* operative action *if there exists a function $\mathrm{u}_P : V(W) \to 2^{U(W)}$ such that $\mathrm{u}_P(v)$ includes an action u_k if and only if P and W have a joint-execution $e = y_1u_1y_2 \ldots u_{k-1}y_k$ for which:*

- *e arrives at $w \in V(W)$ when traced on W;*
- *action u_k is a label on an outgoing edge from w;*
- *e arrives at $p \in V(P)$ when traced on P;*
- *action u_k is a label on an outgoing edge from p.*

For a given world, intuitively, the operative action set at any vertex in that world consists of only those actions that the robot (using its plan) may actually end up taking during its execution. When a plan doesn't induce any progress measures, we will consider alternate plans with the property that they select the same actions in the same places in the world as that original plan. Operative actions formalize this:

Definition 9 (derived plan)**.** *Given world W, a plan P' is* derived from *another plan P if the operative action set of P' is contained in the operative action set of P; that is for all $v \in V(W)$, $\mathfrak{u}_{P'}(v) \subseteq \mathfrak{u}_P(v)$.*

Next, we generate all possible plans without crossovers from some other plan, but using only actions from the operative action set. The algorithm we describe next, named CLIP, has two parts: first, given a plan and a planning problem, it builds an intermediate data-structure (a graph) through which all of the crossovers can be identified. Secondly, it enumerates resolutions to these crossovers. For edges that are not involved in any crossover, we may take any subset of them so long as we ensure it is a solution. Thus, as an object on which various combinatorial operations act, this graph implicitly represents a large number of plans.

3.5.1 The Plan-World Interaction Graph

CLIP starts by constructing a type of product graph we call the *Plan-World Interaction Graph*, or simply the *I-Graph*. It is formed by corresponding plan and world vertices, while also encoding information about the operative action set in an organized fashion. In the original plan, different executions, possibly separated and obscured by the plan's structure, may interact so as to obstruct the existence of progress measures (as in Figure 3.5). These various instances are collapsed within the I-Graph, which records when multiple plan vertices map to the same world vertex, transforming crossovers into explicit cycles.

The I-Graph is itself a p-graph, but its structure is defined by the joint-executions of the plan and world. Algorithm 1 provides pseudo code for the incremental construction of the I-Graph. To know which plan vertices correspond to certain world vertices at different points in execution, the I-Graph starts as a single vertex that corresponds to the initial vertices of both the plan and the world. Each vertex in the graph will have a pair (v_P, v_W) — for this initial vertex, $(V_{\text{init}}(P), V_{\text{init}}(W))$ — indicating what vertices within the plan and world it corresponds to. This single observation forms

Algorithm 1: Constructing the Plan-World I-Graph

```
Data: World $W = (V_W, V_{\text{init}W}, Y_W, U_W, E_W, V_{\text{goal}})$,
      Plan $P = (V_P, V_{\text{init}P}, Y_P, U_P, E_P, V_{\text{term}})$
Result: I-Graph $I = (V_I, V_{\text{init}I}, Y_I, U_I, E_I, V_{\text{term}})$
Procedure I-GRAPH CONSTRUCTION($W$, $P$):
    igraph ← New PGraph()
    igraph_vtx ← New Observation_Vertex(I-Graph)
    world_vtxs, plan_vtxs ← $V_{\text{init}W}$, $V_{\text{init}P}$
    I-GRAPH SEARCH(world_vtxs, plan_vtxs, igraph, igraph_vtx)
Function I-GRAPH SEARCH(world_vtxs, plan_vtxs, igraph, igraph_vtx):
    if no outgoing edges for world_vtxs, plan_vtxs then
        return I-Graph
    if vertices are type "observation" then
        world_tgts = New Set()
        plan_tgts = New Set()
        foreach world vertex $v_W$ do
            foreach outgoing edge from $v_W$ do
                observation ← outgoing edge label $y$
                add edge.tgt to world_tgts
                foreach out edge from plan_vtxs do
                    if plan edge has label $y$ then
                        add plan edge.tgt to plan_tgts
    else
        world_tgts = New Set()
        plan_tgts = New Set()
        foreach plan vertex $v_P$ do
            foreach outgoing edge from $v_P$ do
                action ← outgoing edge label $u$
                add edge.tgt to plan_tgts
                foreach out edge from world_vtxs do
                    if world edge has label $u$ then
                        add world edge.tgt to world_tgts
    if world_tgts corresponds to existing vertex $v_I \in V_I$ then
        if world_tgts $\subseteq V_{\text{goal}}(W)$, plan_tgts $\subseteq V_{\text{term}}(P)$ then
            add $v_I$ to $V_{\text{term}}(I)$
        if vertices are type "observation" then
            label_set ← outgoing obv. from $world\_vtxs$
        else
            label_set ← outgoing act. from $plan\_vtxs$
        for each label $n$ in label_set do
            backlabel ← label
            add edge (igraph_vtx $\rightarrow v_I$, with label) to igraph
            I-GRAPH SEARCH(world_tgts, plan_tgts, igraph, $v_I$)
    else
    if vertices are type “observation” then
        label_set ← outgoing obv. from world_vtxs;
        new_vtx ← New Observation_Vertex(IGraph);
    else
        label_set ← outgoing act. from plan_vtxs;
        new_vtx ← New Action_Vertex(IGraph);
    if world_tgts $\subseteq V_{\text{goal}}(W)$, plan_tgts $\subseteq V_{\text{term}}(P)$ then
        add new_vtx to $V_{\text{term}}(Igraph)$;
    for each label $n$ in label_set do
        backlabel ← $y/u$;
        add edge (igraph_vtx → new_vtx, with label $y/u$) to igraph;
        I-GRAPH SEARCH(world_tgts, plan_tgts, igraph, new_vtx)
return I-Graph
```

the initiating layer of the I-Graph, modeling the moment execution begins and the robot has yet to receive any observations.

The rest of the graph is constructed by tracing the possible executions of the plan, alternatingly giving priority to the plan and world when determining structure. For each observation vertex, the current set of world vertices determines the outgoing edges. The plan is safe on the world, and therefore for each observation that might occur from any of these outgoing edges, the plan has at least one corresponding edge labeled with the same observation from each vertex within the set under consideration. For each of these edges, an action vertex is added to the I-Graph, consistent with the world and plan vertices reached by that observation.

In the examples discussed above, a single observation completely localized the robot to one vertex in the world. However, one observation need not isolate the robot to a single vertex within the plan as there may be multiple ways (e.g., the blue and red routes in Figure 3.5) from which our robot may have chosen arbitrarily. For the plan in Figure 3.5, the initial observation is (for most cases) consistent with up to two possible plan vertices.

As per the earlier discussion of partial observability, this can also be true within the world itself: information in an observation may correspond to multiple places within the world. In such cases, v_W is a set of vertices, just as v_P is in the example shown here.

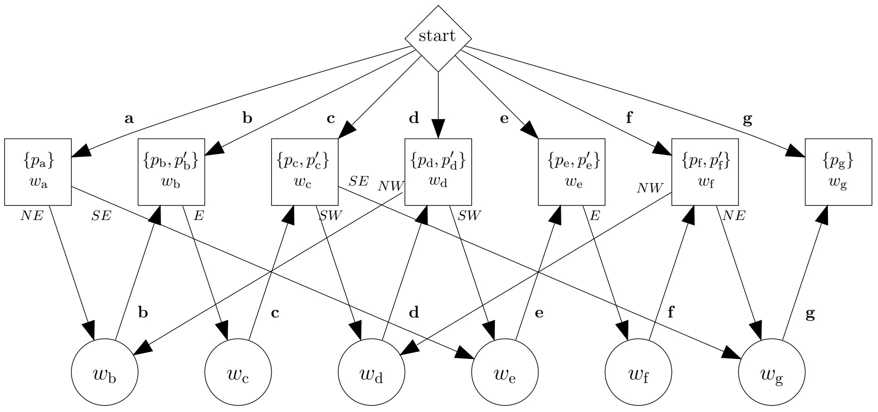


Figure 3.6: An I-Graph for the plan seen previously in Figure 3.5. Plans may have multiple vertices that correspond to the same vertex in the world. The I-Graph merges these into a single vertex.

For each action vertex in the I-Graph, unless it is a terminating vertex, each corresponding plan vertex in v_P has outgoing edges with actions the robot can take. (It is also possible that vertices in V_{term} have their own outgoing actions, indicating the option to terminate.) These actions are added as outgoing edges, leading to a new set of observation vertices for the plan and world. If a vertex within v_P does not have that action as an outgoing edge from itself, it is dropped from the current plan sets, as our robot has committed to a certain part of the plan.

Again at an observation vertex, the process repeats until we have explored all possible executions of the plan. Should the execution return the robot to a set of world vertices already encountered, the edge connects back to the existing vertex. If new plan vertices are associated with this set of world vertices, they are appended to the existing list of associated vertices.

The result is a graph with three layers: the initiating layer, the plan layer, and the world layer. The I-Graph for the running example is shown in Figure 3.6. The initiating layer is our single starting vertex that corresponds to all possible starting locations. (Although illustrated with a diamond, this is simply to highlight that this observation vertex differs from the others.) The plan layer consists of all action vertices, and derives its name from the fact that actions are the part of execution dictated by the plan. Its departing edges go to the world layer. Though there may be multiple observations a robot may receive, the various edges correspond to the world's choice of observation to provide. The edges from this layer return to the plan layer. The I-Graph mimics the structure of the world and its connections, but is restricted by the operative action set of P.

Lemma 1. *The set of all actions for I-Graph I generated from plan P has the same operative action set as P on the original planning problem W.*

Proof. Beginning the construction, the algorithm starts with the set of initial world vertices $V_{\text{init}}(W)$ and the set of initial plan vertices $V_{\text{init}}(P)$. The set of all outgoing observations from the vertices in $V_{\text{init}}(W)$ represent observations the robot can receive initially. As the plan is presumed to be safe on the world, each vertex in $V_{\text{init}}(P)$ has an outgoing edge labeled with the possible observations.

Therefore, for each possible observation y, the algorithm obtains the result of tracing a joint-execution consisting of that single observation as a new set of world and plan vertices: $(\mathcal{V}_y^P, \mathcal{V}_y^W)$.

For u_k to be an operative action at vertex $v_w \in V(W)$, there must exist a joint execution e arriving at v_w and also $v_p \in V(P)$, where both v_w and v_p have an outgoing edge labeled with u_k. The two sets of vertices just reached by the algorithm, $(\mathcal{V}_y^P, \mathcal{V}_y^W)$, comprise action vertices. As shown in lines 19–28 in Algorithm 1, the outgoing edges for each action vertex in $\mathcal{V}_y^P$ are considered, and their targets collected into a set. As the plan is presumed to be safe on the world, an outgoing action from a vertex $v \in \mathcal{V}_y^P$ indicates that each vertex in $\mathcal{V}_y^W$ has an outgoing edge with the same label. These edges indicate the operative action set for each world vertex within our current set under the joint-execution y, by definition.

Now, for each outgoing action u, following the edges labeled with u leads to the set of vertices obtained by tracing yu on the plan and world: $(\mathcal{V}_{yu}^P, \mathcal{V}_{yu}^W)$. As y is an element in the larger set $W(Y)$, and u is an element in the set $P(U)$, each set $(\mathcal{V}_{yu}^P, \mathcal{V}_{yu}^W)$ is a subset of all vertices which can be reached by joint-executions of length two. As the algorithm has considered all possible observations that could be obtained from the world, as well as all possible actions the plan could take, it has visited all vertices resulting from any joint-execution of length two.

For each y, u pair, the algorithm passes $(\mathcal{V}_{yu}^P, \mathcal{V}_{yu}^W)$ to a recursive call of itself, upon which the same process is repeated. As the incoming actions do not impact which outgoing edges may be obtained, $(\mathcal{V}_{yu}^P, \mathcal{V}_{yu}^W)$ can be treated within the iteration in the same way as the initial set, without regard to the prior actions or observations received.

The algorithm continues until the joint-executions end, which they must as they have bounded length, and further they end in termination (i.e., in $V_{\text{term}}(P)$). Thus, the algorithm obtains the operative action set for all sets of world vertices visited. As each vertex within the constructed I-Graph corresponds to a pair of sets $(\mathcal{V}_e^P, \mathcal{V}_e^W)$, and all actions found by the algorithm are appended as outgoing edges from the vertex associated with that pair, the I-Graph captures the full operative action set of the original plan P.

$\square$

A subset of the I-Graph vertices are also designated as terminating vertices, just as for plans. During construction, if the set of plan vertices for the current execution are all within $V_{\text{term}}(P)$, and the corresponding world vertices are all within $V_{\text{goal}}(W)$, then the vertex is included in $V_{\text{term}}(I)$. Termination is not required for *every* set of plan vertices that reaches this vertex, as a robot may pass through goal vertices before actually stopping. However, it must guarantee goal achievement when it eventually does terminate.

This not only handles cases in which the robot is unsure if it has reached the goal or not, but also permits cases in which the robot may bypass one goal to reach another. In such a case, the robot's plan has two trajectories: one which goes to the goal vertex and terminates, and another that passes through that vertex to reach another goal further along. In the I-Graph, both will be terminating vertices, but the first will have outgoing edges, providing the robot the ability to terminate the execution *or* continue.

Though the I-Graph is a p-graph augmented with a set of terminating vertices, and is constructed from a plan, it is not necessarily a plan itself. It satisfies the same safety requirements as a plan, but crossovers from the source plan manifest as cycles, which may mean that the set of joint-executions with the world is not finite. Resolution of all these cycles within the I-Graph will result in a plan that is derived from the original plan P and which both solves the planning problem W and has a progress measure. (Later we will show that the resolution of the crossovers demands goal vertices be reached, retaining the requirement that our plans have no dead ends.)

3.5.2 The Comes-Before Relation

The essence of the proof of Theorem 1, in dealing with progress measures, only used an ordering property and then constructed a function from that ordering. (The definitions for progress measures are written in terms of functions to closely retain Erdmann's original form.) Next, we describe a relation put on the action vertices of the I-Graph, which helps determine whether those vertices can be ordered. We focus on the I-Graph's action vertices because actions are the sole means by which the robot exercises control and hence has the power to split apart crossovers.

We call this relation the *comes-before relation*, denoted $K \subseteq V_u(I) \times V_u(I)$. For action vertices v_1 and v_2 within the I-Graph, we say v_1 comes-before v_2 (written $(v_1, v_2) \in K$) if, after an action taken from v_1, the observation vertex reached by that action has v_2 as a target. Also, that K is transitive: $(v_1, v_2) \in K$ and $(v_2, v_3) \in K$ means $(v_1, v_3) \in K$.

The I-Graph itself is used in initial construction; once the transitive closure of K is computed, this gives for each action vertex all vertices that it precedes. There are two points of note: firstly, the nature of the I-Graph means that its vertex set is constantly changing during its construction. Additionally, the I-Graph's p-graph form means that it is similar to the plan and world from which it derives; as mentioned previously, this means that the resolution of crossovers results in a new plan that is already expressed using the same structure as the original plan.

Once the vertices and edges of the I-Graph are fully established, experience has shown that an adjacency matrix representation provides simple, compact way of representing the relation between vertices. It also affords computationally attractive optimization (e.g., obtaining the transitive closure via linear algebraic methods).

For the purposes of resolving crossovers, we consider each vertex within the I-Graph, and not the sets of vertices they correspond to in the world. As a result, a vertex v_1 in the I-Graph 'comes before' another vertex v_2 if there is a path in the I-Graph between two vertices which correspond to v_1 and v_2, respectively.

When dealing with fully observable planning problems, this matrix has a one-to-one correspondence to world vertices, and resolving the crossovers present in the I-Graph yields a vertex progress measure. In some instances of partially observable problems, it is possible for the same vertex in the world to correspond to multiple vertices in the I-Graph. In these cases, resolution of the crossovers in the I-Graph may yield only an execution progress measure on sets of vertices, as opposed to a vertex progress measure on individual vertices in the world. The impact of this is discussed starting in 3.8, with important implications following.

3.5.3 Resolving Crossovers

When the I-Graph collapses down all executions, conflicting orderings on world vertices that only existed conceptually now present as cycles within the I-Graph itself. Resolving crossovers now becomes a matter of "clipping" away some of the edges that constitute these cycles, which is the function of the second part of CLIP.

Each cycle within the I-Graph yields a set of edges, which can be further reduced. For each set, we consider a subset called the *candidate edges*. Edges are disqualified from being a candidate for removal if the algorithm can determine in advance that their removal will result in a non-solution.

If the planning problem is both fully observable and requires that the robot is capable of starting from anywhere, then edges are excluded from candidacy if their removal would leave a non-goal vertex with no outgoing edges, which would leave the robot stranded at the corresponding world vertex.

In all other problem cases, removing edges and stranding certain vertices in the I-Graph may only result in a change in executions. Therefore, the only edges which can be immediately disqualified are those that are the sole outgoing edge from an action vertex at the start of execution, as the robot must have choice in action from all starting locations. The algorithm also removes cycles that are fully contained within larger cycles in the I-Graph, so that the same sets of vertices are not considered multiple times.

In our implementation, the size of the search space is highly dependent on the structure of the problem. Factors such as the proportion of edges that are shared between crossovers, or how many edges can be removed from candidacy, depend on the specific plan and planning problem. In the hypothetical worst case, every edge of the I-Graph is a candidate, and the size of the search tree is $2^{|E(I)|}$. As even moderately sized problems can give a search space that will be very large, one feature of our implementation is that it includes a human-guided mode of operation wherein the user can interactively prune parts of the search (inspired by tactic-based proof methods). The user has the option of manually resolving some cycles before the main search. As the user often has additional insight into the problem structure (for instance, understanding how choices taken in one cycle may impact desired choices in future cycles), this knowledge can be used to help the algorithm search a reduced space and to return resolutions that are relevant to conditions of interest to them.

CLIP constructs a search tree starting from the original I-Graph, using the sets of candidate edges for each crossover. The search tree considers the powerset of the candidate edge set for removal. For each set of edges from the powerset, we add to a list of proposed edges to remove from the original I-Graph.

In addition to limiting the set of edges considered, CLIP also prunes branches of the search tree once it is known that they will have no solutions. CLIP only removes edges if the I-Graph's initial vertex cannot reach the goal, this is an irreversible disconnection and it will continue to fail to be a plan given any additional removals. Therefore, the validity of each currently proposed set of removed edges is checked to see if the current node in the tree still permits the initial vertex within the I-Graph to reach the goal before CLIP generates its children.

3.5.4 The Output of CLIP

We call the output plans of CLIP the *representatives* collectively (or individually, a representative or *representative plan*). CLIP produces numerous subgraphs of the I-Graph, each of which has a method of resolving all crossovers that is different from the other outputs. Each representative plan in the output set is both a plan derived from the I-Graph by CLIP, and has an operative action set that is a subset of that of the I-Graph used to generate it.

These output plans are representatives of the full set of all solutions in the sense that they capture all possible ways of resolving the crossovers. Any plan in the full set of solutions that is not directly generated by CLIP can be derived from a representative.

Theorem 2. *All plans generated by* CLIP *using a plan P are derived from P, have progress measures, and solve the planning problem W.*

Proof:

1. **All plans generated by CLIP are derived from the plan P.**

   CLIP only removes edges from an input plan, and cannot add actions to output plans that are not part of the input. CLIP creates an I-Graph, which via Lemma 1 has the same operative action set as P, and then generates plans from subgraphs. Therefore, any plan CLIP generates must have an operative action set that is equivalent to the operative action set of P, or a subset.

2. **All plans generated by CLIP have a progress measure.**

   By the comes-before relation, CLIP verifies that no cycles in the world exist before acceptance. As the lack of cycles is indicative of a lack of crossovers, by Theorem 1 all plans generated by CLIP have a progress measure.

3. **All plans generated by CLIP solve the planning problem.**

   By definition, before accepting any solution, CLIP calculates the comes-before relation. If, for any world vertex, that vertex does not 'come before' at least one goal vertex, CLIP rejects it. CLIP also rejects any plans with cycles still present. Therefore, any plan CLIP accepts is a solution.

□

Theorem 3. *All plans derived from* CLIP*'s representatives are in the solution set.*

Proof. We define plans derived from CLIP's results as any plans constructed from a subset of edges from a result produced by CLIP. Unless removal of an edge results in the breaking of all paths from an initial vertex to the goal, edge removal will not result in a plan no longer being a solution. In addition, as removing action edges cannot induce a cycle on a plan that does not have one, the resulting plan keeps a progress measure, ensuring that it is also part of the solution set. □

Theorem 4. *Every plan in the solution set can be derived from a representative plan.*

Proof. Assume that there is a plan P' that is not generated by CLIP nor derived from CLIP's solutions. Then P' must solve the original planning problem W, use actions found only in the operative action set of P, have a progress measure, and not be a representative produced by CLIP or derived from these results.

We define two sets: E^{loop} and E^{static}. E^{loop} is the set of edges in the world that are part of the operative action set of all cycles in the I-Graph. By definition, CLIP considers E^{loop} as candidates for removal. E^{static} is the set of all other action edges in the operative action set. As P' uses the operative action set that is also used to generate the solutions of CLIP, it must differ from any given plan provided by CLIP in E^{static} or E^{loop} (or both).

If P' differs in E^{static}, it can be in one of two ways: either it contains an element not in any representative produced by CLIP, or it is a subset of any given representative's E^{static}. If it is a subset, then P' is actually derived from that representative. If it contains an element not in a representative, then that element is not from the operative action set, and P' is not truly in the solution set, as CLIP does not remove any edges from E^{static}. Therefore all representatives generated by CLIP contain the entire set of E^{static} from the original plan P.

If P' differs in E^{loop}, it can be in one of two ways: if it has an extra edge, then it either contains an action not from the operative action set or it still contains a cycle that causes its language with the world to not to be finite. If it is smaller than any representative, then it implies CLIP does not enumerate all possible values of the cycle edges. As CLIP enumerates all cycle edge possibilities (through generating the powerset), it must enumerate all possible values of the cycle edges, so this is a contradiction.

Therefore, P' must be derived from some representative in order to be a valid plan; any derivation other than from the representative set violates our requirements for plans to solve.

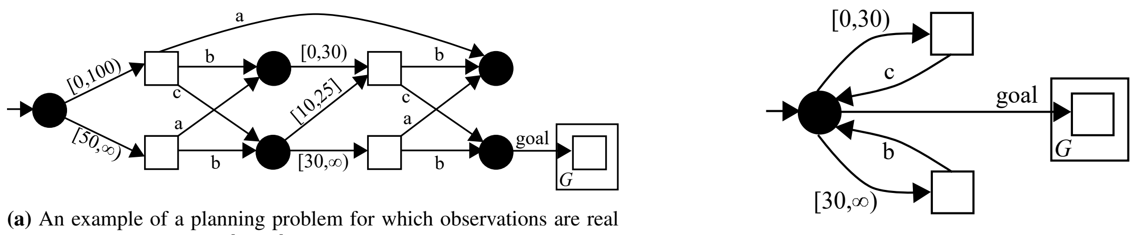
□

The input plan therefore yields numerous representatives, all of which have progress measures and which can be used to generate the entire set of plans of interest. As the executions of the representatives are derived from the original plan, they may contain executions that are not included within the original language of the input, but are a subset of its set of operative actions. Having resolved the issue of plans without vertex progress measures, we will shortly turn to the question of how to use progress measures to obtain sensors. However, a brief excursion first will help to emphasize that the simplistic graphs considered in this section do not mean the discussion cannot be brought to bear on problems that more naturally model robotics settings.

3.5.5 Continuous Sensors

The examples presented thus far have used sensors that return discrete observations. In practice, sensors are seldom so simple as to give distinct, individual readings. Noisy sensors may return values within some range of readings. Also, even when the digital encoding of the sensor data means there are finitely many possible readings, it may be useful for the model to assume otherwise. (The formalism, tacitly, already allows for this: for instance, Definition 1 allows edges to be *labeled* with sets.) As an elementary example, consider Figure 3.7a, which shows a planning problem whose observation edges bear labels that are subsets of the non-negative reals, expressed as ranges of values (think, for instance, of a distance sensor). Notice that when the ranges overlap with one another, a given reading in their intersection is essentially ambiguous and permits a transition across either edge.

In this example, no two observation edges carry identical labels, but there exist subsets of sensor readings which match multiple labels. Consider the pair of edges labeled with $[0, 100)$ and $[50, \infty)$; though distinct labels, they share a common sub-range, $[50, 100)$. This range can



(a) An example of a planning problem for which observations are real numbers within the range $[0, \infty]$.

(b) An action-based sensor solution.

Figure 3.7: An example planning problem for which observations are real numbers.

be extracted via a modification of the original labels, producing a new set of labels which can be operated on directly by CLIP in the same way that the discrete observations are. (The result is Figure 3.7b.)

As a pre-processing step, transforming edge labels (whether they describe finite or infinite, continuous or discrete sets) requires splitting the set such that the edge labels are separated into pieces of sufficient fineness. This is achieved by taking intersections and differences — the description of the set must be rich enough to support those two basic operations. Starting with the initial labels, the computation grows the collection of labels by adding the intersections and differences of all the sets described by the current labels, and repeating this until the labels reach a steady state. The intersection allows for the identification of common observations within the labels, while taking the differences includes in the labels what remains after extracting those common observations. Algorithm 2 gives pseudocode for this process, where operations on the labels are given simply as operations on the sets they describe. After this algorithm has completed, the original edges in the world (and plan) are then updated, with the original labels being replaced with the all of the new, finer labels that are a subset of said original label. In the case that an observation edge's label contains multiple sets, these are treated (and replaced) individually.

Once these labels are generated and CLIP has been run using the finer labels, different labels that result in the same action can be re-combined into a larger label via post-processing. Figure 3.7b shows a plan (which is also an action-based sensor) that has only three labels: $[0, 30)$, under which action c is safe and makes progress, $[30, \infty)$, under which action b is safe and makes progress, and *goal*, which identifies goal attainment.

Lemma 2. *Via union, the finer labels (Y'_E) generated by Algorithm 2, recover the original set of observations Y_E.*

Proof. Given any two different labels A and B, we can regenerate both of the original sets. The algorithm produces three new labels from A and B: $A \cap B$, $A \setminus B$, and $B \setminus A$. After execution of the algorithm, the label A is then replaced with the two labels $A \cap B$ and $A \setminus B$, while label B is replaced with the labels $A \cap B$ and $B \setminus A$. By definition, $A \cap B$ contains all elements in

Algorithm 2: Label Intersection on Observations

```
Data: Collection of edge labels $Y_E \subseteq 2^Y$
Result: Collection of finer labels $Y'_E$
1  $Y'_E = Y_E$
2  while $Y'_E$ Not In a Fixed State do
3      Labels = NewCollection()
4      for $i$ in $1 \cdots |Y'_E|$ do
5          $A = Y'_E[i]$
6          for $j$ in $(i+1) \cdots |Y'_E|$ do
7              $B = Y'_E[j]$
8              Labels.Add($A \cap B,\ A \setminus B,\ B \setminus A$)
9      $Y'_E = Y'_E \cup (\text{Labels} \setminus \{\varnothing\})$
10 return $Y'_E$
```

common between A and B, while $A \setminus B$ contains all elements of A that are not also within B (and vice versa). The union of these two labels recreates the original set A. As no observations are lost during the modification of labels, any original label can be obtained through the union of the sub-labels that have been created by Algorithm 2. $\square$

The resultant set of labels (Y'_E) is a π-system as it is closed under finite intersections; however, taking only intersections of all original elements of the set of labels is not sufficient to generate the desired output set, as set differences must also be taken and included. (Note: For formal completeness, the resultant set of labels is also *not* a λ-system; even though it is closed under the relative complement, it need not be closed under all unions of the finite set.)

3.6 Translating Progress Measures into Sensors

Now that we have obtained a plan with a progress measure, we can use it to link actions with how to make progress. This is achieved through the notion of a progress cone, as proposed by Erdmann. Every action $u \in U$ has an associated progress cone, which is a set of observations. At any vertex in the world labeled with an observation in this set, the action u makes progress toward the goal (transitioning from a higher-valued vertex to a lower-valued vertex) according to the progress measure.

Definition 10 (progress cone). *For a planning problem W and plan P with a progress measure ϕ, the* progress cone of an action $u \in U(W)$ *is the largest subset of* $Y(W)$, $\{y_1, y_2, \ldots, y_k\}$ *where* u *makes progress under* ϕ*, from all vertices with an outgoing edge labeled with some* y_i.

Two views of the cones are possible. The first, which is natural from the preceding definition, maps from actions to observations; the second asks which actions make progress given a certain observation. Since both views are useful, we now define the progress cone as a new relation between observations and actions so that each observation has an associated set of actions that make progress.

Definition 11 (cone relation). *For a planning problem W and plan P with a progress measure ϕ, the* cone relation $\mathbf{C} \subseteq Y(W) \times U(W)$ *contains* (y, u) *if there exists a progress cone for the action* u *containing* y. *We will also write* $y \underset{\mathbf{C}}{\sim} u$, *when* $(y, u) \in \mathbf{C}$.

Given the set of observations $Y(W)$ for a planning problem, any observation y that is used as an edge label of a vertex in $V(W)$ must have at least one action u where $y \underset{\mathbf{C}}{\sim} u$. This forms a covering over $Y(W)$.

For an observation, there are potentially many progress-making actions. However, only one action is needed for any given y to guarantee that the robot will eventually arrive at the goal. Translating Erdmann's concept of an abstract sensor into our framework, we can define a class of functions that, for each observation y, return a single action u, transforming the covering into a collection of partitions. We call these functions *singleton action-based sensors*.

Definition 12 (singleton action-based sensor). *A function $f : Y(W) \to U(W)$ is a* singleton action-based sensor *if* $y \underset{\mathbf{C}}{\sim} f(y)$ *for every* y.

The connection between singleton action-based sensors and real sensors may not be immediately apparent. A "traditional" sensor can be represented as a function $s : V(W) \to Y(W)$, taking world vertices as inputs and returning some observation.

To bridge the gap we define a new set of observations $Y' = \{y'_u \mid u \in U(W)\}$ by making a correspondence of each element to an action, as indicated by the subscript. For an element

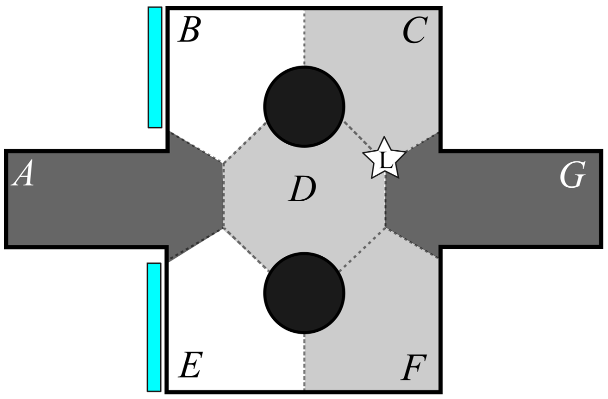


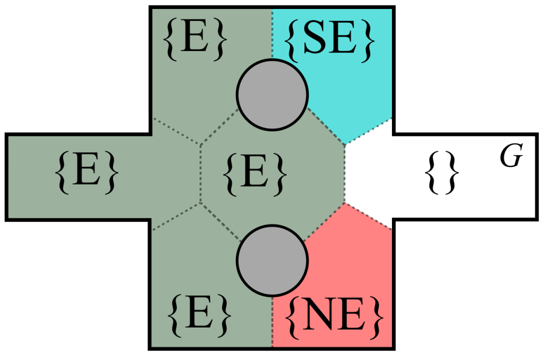
Figure 3.8: A world in which conflation between vertices precludes Erdmann's sensors being functional. Vertices C and D both are in mid-range lighting and can see the landmark L, making them indistinguishable. For such a sensing setup, the partition required by the backchained plan cannot be realized.

$y \in Y(W)$, $y \mapsto y'_u$ if $f(y) = u$ according to a singleton action-based sensor. Each element in Y maps to a set of vertices in the world, and so we can think of the elements of Y as a stand-in for the identifiable regions of W. $W \to Y \to Y'$ is therefore equivalent to $W \to Y'$, and takes in vertices in the world and maps them to this new set.

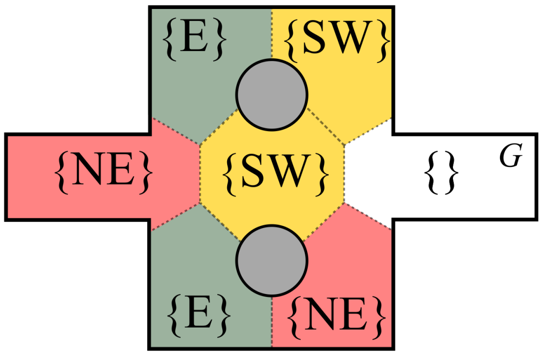
The process above transforms a covering into a partition through use of a function. However, perhaps we would like multiple (or even all) possible progress-making actions for a single observation. To achieve this, we present an extension of action-based sensors, which we define as *permissive action-based sensors*.

Definition 13 (permissive action-based sensor)**.** *A function* $f : Y(W) \to 2^{U(W)} \setminus \{\varnothing\}$ *is a* permissive action-based sensor *if, for every* $u \in f(y)$*,* $y \underset{\mathbf{C}}{\sim} u$*.*

The relationship of this object to more traditional sensors is less clear than for a singleton action-based sensor. We can construct a Y' as before, where now each element y' corresponds to some subset of U, but the semantics of actions within multiple different sets becomes a matter of interpretation. Some recent work has examined covers as models of sensors that have imperfections, such as noise or cross-talk (Zhang and Shell, 2020, 2021). It is curious that permissive action-based sensors appear to be more powerful owing to the choices inherent in the cover, not less powerful.



(a) Partition consistent with the backchained plan in Figure 3.4b, and its resulting progress measure.



(b) Partition consistent with the plan in Figure 3.4c and its resulting progress measure.

Figure 3.9: Two partitions consistent with a progress measure, implementing an action-based sensor. Figure 3.9a, derived from the backchained plan in Figure 3.4b, cannot be used to solve the planning problem in Figure 3.8, while 3.9b, derived from the plan in Figure 3.4c, can.

3.6.1 A Concrete Example

We illustrate some of preceding definitions by adding to the example of Figure 3.4a. Figure 3.8 shows our example environment in a bit more detail. Windows (in aquamarine blue) allow light to enter and result in varying light levels throughout the space (in shades of gray). A landmark (L) can be seen from vertices C, D, and G, but not elsewhere.

Figure 3.9 shows two singleton action-based sensors, each of which maps the observable regions of the world to single actions. Each of the two partitions is consistent with the progress measure of the plan from which they were derived. The action-based sensor given in Figure 3.9a is derived from backchaining, and is an action-based sensor that would be found with Erdmann's method of obtaining them. The second action-based sensor given in Figure 3.9b is derived from the plan in Figure 3.4c. This plan is only considered when we expand the concept of progress measures and action-based sensors to a wider set, as done in the preceding definitions.

Now, return to the detail in Figure 3.8 where, due to constraints on what sensors are available, a robot can only recognize brightness levels and the presence of a nearby landmark. Such a robot cannot distinguish vertices C and D. This presents a problem for the action-based sensor obtained from a backchained plan, which requires these vertices be distinguished. The action-based sensor realized in Figure 3.9b, however, takes the same actions in vertices C and D, and therefore describes a partition of the world that can be realized with this setup. The action-based sensors defined by the extended family of plans we consider allow one to better respect the constraints that designers may actually face.

3.7 The Sensor Lattice

A single plan may result in multiple action-based sensors, each of which may have multiple partitions that are consistent with it. These partitions draw parallels to the virtual sensors discussed in LaValle (2019). In his model, sensors are defined as a function from the world to sensor readings; for a given sensor value, its preimage is a set of locations in the world that map to that sensor reading. Each sensor then yields a partition over the entire world, based on which locations give which sensor readings.

The sensors-as-partitions are then arranged in a lattice structure based on the fineness of their partitions, with coarser partitions towards the bottom and the partition that can uniquely identify every location in the world as the supremum. The previous example, in 3.6.1, showed two partitions (Figs. 3.9a, 3.9b) from two plans. The partitions obtained from their action-based sensors are naturally included within the full possible sensor lattice, which contains all such possible partitions of the state space.

LaValle (2019) addresses the question of state requirements indirectly. Environments that change over time are modeled through adding time as an additional component in their state space. The sensor readings are then functions of both space and time. In the case of trajectory tracking, the history of states and times in which they were visited becomes the state space upon which the sensor acts.

His treatment of time as a separate variable is a substantial difference from any of the ideas we have considered here. Still, the concept of modifying the state space to accommodate how different observations are received over the course of the execution is reminiscent of I-Graphs with execution progress measures, where action depends on the current execution. However, implementing execution progress measures requires more information than just the history of states in the world that the robot has visited. Rather than the state space reflecting only external changes to the world, the sensor must also change based on what actions have been taken by the robot. The requirement of task achievement means that the sensor must be a function both of the world and of the robot's internal state. Additionally, unlike trajectory tracking, the robot does not require its full history to provide the required information, but a particular subset of it.

To convert these execution progress measures into sensors, Definition 12 is insufficient. We must include the history of relevant actions as a sort of *context* that informs what actions are given by the action-based sensor.

Definition 14 (contextual sensor)**.** *A function* $g : (Y(W) \times F) \to (U(W) \times F)$ *is a* contextual sensor *if, for all* $y \in Y$*:*

1. *for a given* context f, $g(y \times f) \underset{\mathbf{C}}{\sim} u$*, or*
2. $g(y \times f) \underset{\mathbf{C}}{\sim} \{\varnothing\}$*,*

and *the resulting execution language is safe on the planning problem* W *from which the relation* ***C*** *is derived.*

(We use f for a certain context as opposed to c for two reasons: first, **C** is already used to indicate the progress cone relation; second, we will later see that each of these contexts relates to a kind of graph we call a *flower graph*.)

As the context varies over time, it may be that the function g is not defined for all possible pairs of (y, f), but only those y which the robot may encounter under the given context f. The consequence is multiple partially-defined action-based sensors. The resulting sensor requirements for the group of sensors are the intersection of each action-based sensor's individual requirements.

While the intersection of these partitions exists within LaValle's lattice, the lattice has no way to capture this notion of state. The following sections discuss precisely how execution progress measures lead to the need for context, as well as a method with which to express these stateful sensors.

3.8 Execution Progress Measures, and What They Mean for Sensors

A new example of a planning problem and plan can be seen in Figure 3.10. Although a robot within this small world is capable of reaching the goal, even from an unknown initial state, it must first take actions that are safe in multiple locations to disambiguate its position in the world. Although we can create an execution progress measure on the sets of world vertices (shown in red), no vertex progress measure exists. Therefore, we cannot define an action-based sensor for this plan either, as there is no mapping between observations in the world and actions to take.

In this example there exists no one-to-one mapping between the plan and world vertices. Such mappings are useful conceptually.

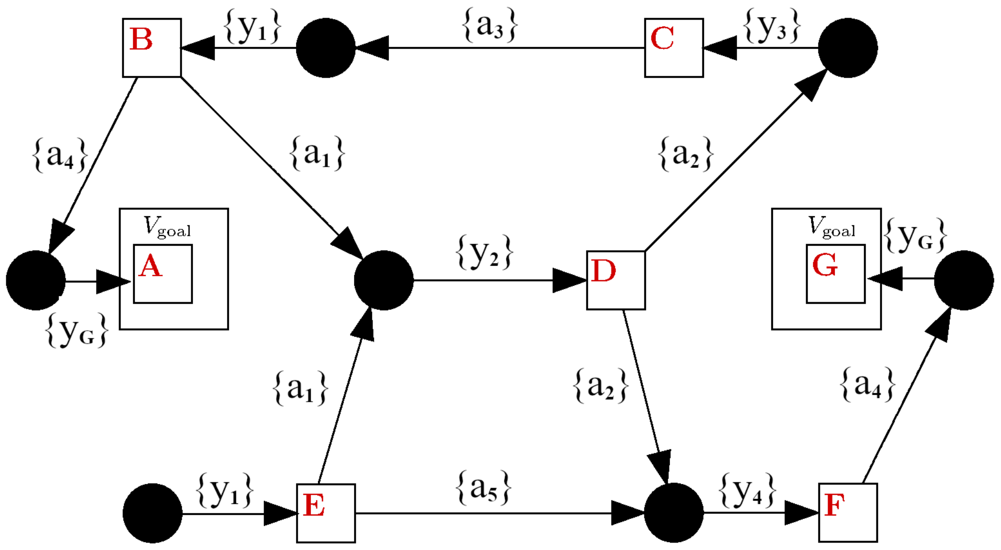
Definition 15 (homomorphic solution (Saberifar et al., 2019))**.** *For a plan P that solves planning problem W, consider the relation $R \subseteq V(P) \times V(W)$, in which $(v, w) \in R$ if and only if there exists a joint-execution on P and W that can end at v in P and in w in W. A plan for which this relation is a function is called a* homomorphic solution.

We consider all plans for which this relation is not a function to be *non-homomorphic*. These plans present unique challenges when designing action-based sensors, and we find that for many of them no action-based sensor may exist at all.

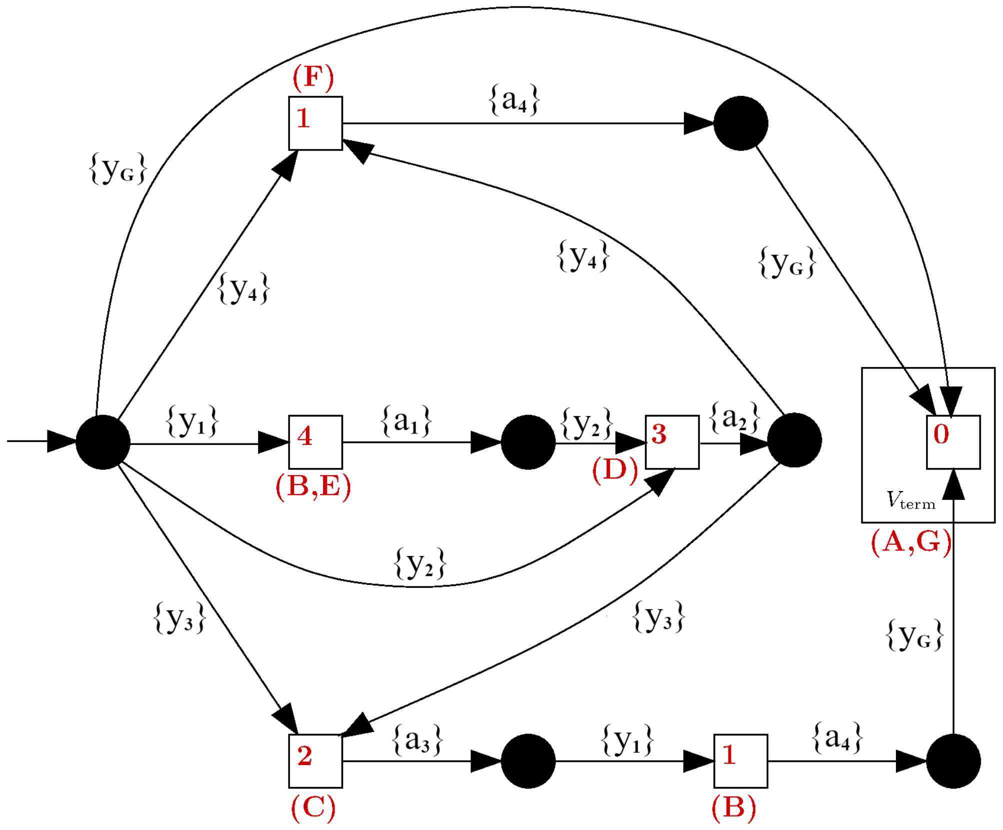
We see in Figure 3.10 that though the robot may revisit the same world state, it occurs under different contexts, which specify different actions. Therefore, we can define a contextual sensor, but not an action-based sensor.

The idea that some robots require some kind of memory or internal state to function is not new; we refer the reader once again to the plans in Figure 3.3. These plans can be converted into action-based sensors, but in their original form are homomorphic, taking after the world's structure. Therefore, while certain structure within plans encodes information necessary to task completion, other structure has no impact on the robot's ability to complete the task. (We return to examine non-homomorphism in more detail in Subsection 3.10.2.)

By changing our perspective on action-based sensors, we can begin to bring these plans into



(a) An (approximately) hourglass-shaped world. If we can start at any observation vertex, then a starting observation of y_1 could indicate that the robot is either in B or E.



(b) A plan which solves the planning problem. In the case the y_1 is the first observation received, the robot must take actions to disambiguate its state before it can reach a goal.

Figure 3.10: An example planning problem, and a plan which is non-homomorphic.

our existing framework. Rather than treating action-based sensors purely as a method with which to define sensors, we focus on their ability to act as reactive plans. An example of an action-based sensor presented as a reactive plan can be seen in Figure 3.11. This representation is the aforementioned flower graph, so called due to its appearance.

While the contextual sensors are not *quite* action-based sensors, they have reactive components that must be connected through transitions. These transitions permit switching between virtual sensors, creating a kind of "stateful sensor". We call these structured plans *vine graphs*, as they consist of small flower graphs connected to each other by chains of observations and actions.

Before exploring vines in more detail, we must first make a detour to explore how the diversity of plans affects both the diversity of sensors and the requirements of stored state. This demands a systematic way to compare plans. With that in place, we can then identify the structure within plans that indicates when the robot switches from one flower to another — the previously mentioned context — as well as how we can use the concept of action-based sensors as a basis from which to construct these plans that incorporate state.

3.9 Expanding the Languages: The Plan Lattice

Depending on the structure of the world, there may potentially be an infinite number of plans. In order to talk precisely about the role of structure and what is meant by our use of the word "state," we must first develop a way in which we can talk about plans in a comparative sense. The *plan lattice* is a structure that, for a given planning problem, affords comparsion of plans to one another.

Though form influences function of plans, we discard their graph structure and examine the language of each plan's joint-executions. If it is possible to generate the same language of joint-executions with different plan structures (for example, a plan and its action-based sensor), these plans should be treated as equivalent.

However, the list of joint-executions alone is insufficient to compare plans with each other. Although they may have the same language, a plan that reaches a world state and terminates has different behavior from another that reaches that world state and does not. We complement the

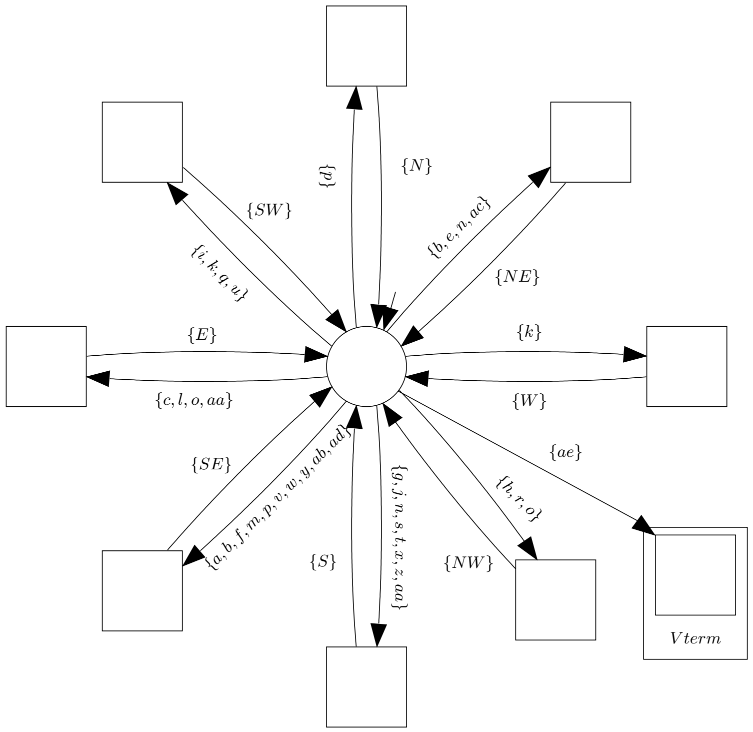


Figure 3.11: A permissive action-based sensor which solves the planning problem given in Figure 3.2, presented in flower graph form.

language with an additional symbol, $\dashv$, to indicate that a given execution ends with the plan terminating.

Definition 16 (cessation language)**.** *The* cessation language *of a plan P supplements $\mathcal{L}(P)$ by duplicating executions arriving at $V_{\text{term}}(P)$, but marking those as such by appending a '$\dashv$' symbol. Formally,*

$$\mathcal{L}_{\dashv}(P) = \mathcal{L}(P) \cup \left\{ s\dashv \mid s \in \mathcal{L}(P), \mathcal{V}_s^P \subseteq V_{\text{term}} \right\}.$$

Recall that the symbol $\mathcal{V}_s^P$ denotes the non-empty set of vertices that one arrives at after tracing an execution s on P. This allows for us to determine if a plan contains the full language of another, including all of the executions for which it terminates. We call this the *plan subsumption relation*.

Definition 17 (plan subsumption relation)**.** *For two plans P_1 and P_2 we say that P_1* is subsumed by *P_2, denoted $P_1 \preceq_{\dashv} P_2$, if $\mathcal{L}_{\dashv}(P_1) \subseteq \mathcal{L}_{\dashv}(P_2)$.*

3.9.1 The Plan Lattice

A plan lattice will be constructed modulo a particular planning problem (W), but since we generally keep a fixed planning problem in mind, we won't hesitate to call it *the* plan lattice. The language of a plan considered as a standalone graph may differ from the language that results from interaction on W. Thus, we rationalize the set of all possible plan languages with respect to a given planning problem W, reducing each language to its joint-executions.

This resulting subset of languages should consist of those which are safe on W. While they must be valid *plans* for the planning problem they are not required to be solutions as defined in Definition 4, and as a result the subset includes languages with executions of infinite length and those that terminate outside of V_{goal}.

Given that the termination symbol $\dashv$ is not included in W's language, we extend W's language for the sake of definition, permitting any string in W to end with the termination symbol $\dashv$, that is:

$$\mathcal{L}_{\dashv}(P) \cap (\{x{\dashv} | x \in \mathcal{L}(W)\} \cup \mathcal{L}(W)) .$$

A bounded lattice is then constructed based on plan subsumption.

The $\subseteq$ relation on languages implies reflexivity and antisymmetry, as languages contain their full selves as a subset, and two languages cannot subsume each other unless they have the same language. The join of any two languages $\mathcal{L}_{\dashv}(P), \mathcal{L}_{\dashv}(P')$ is their union, $\mathcal{L}_{\dashv}(P) \cup \mathcal{L}_{\dashv}(P')$. The requirement of transitivity follows, as all $P \preceq_{\dashv} P' \preceq_{\dashv} P''$ implies that $\mathcal{L}_{\dashv}(P) \subseteq \mathcal{L}_{\dashv}(P') \subseteq \mathcal{L}_{\dashv}(P'')$. The meet of any two languages is their intersection, $\mathcal{L}_{\dashv}(P) \cap \mathcal{L}_{\dashv}(P')$.

Figure 3.12 shows an example plan lattice. At the bottom of the lattice is the plan that does nothing: the $\varnothing$-language plan. Above that is the family of plans that all possess a vertex progress measure, as well as the language of their action-based sensors. Eventually we reach a boundary, beyond which action-based sensors cannot be derived directly from a plan. Above this boundary lie plans with crossovers. All of these plans can be expressed as vine graphs, however only some of them can be reduced by CLIP into families of action-based sensors. At the top of the lattice are

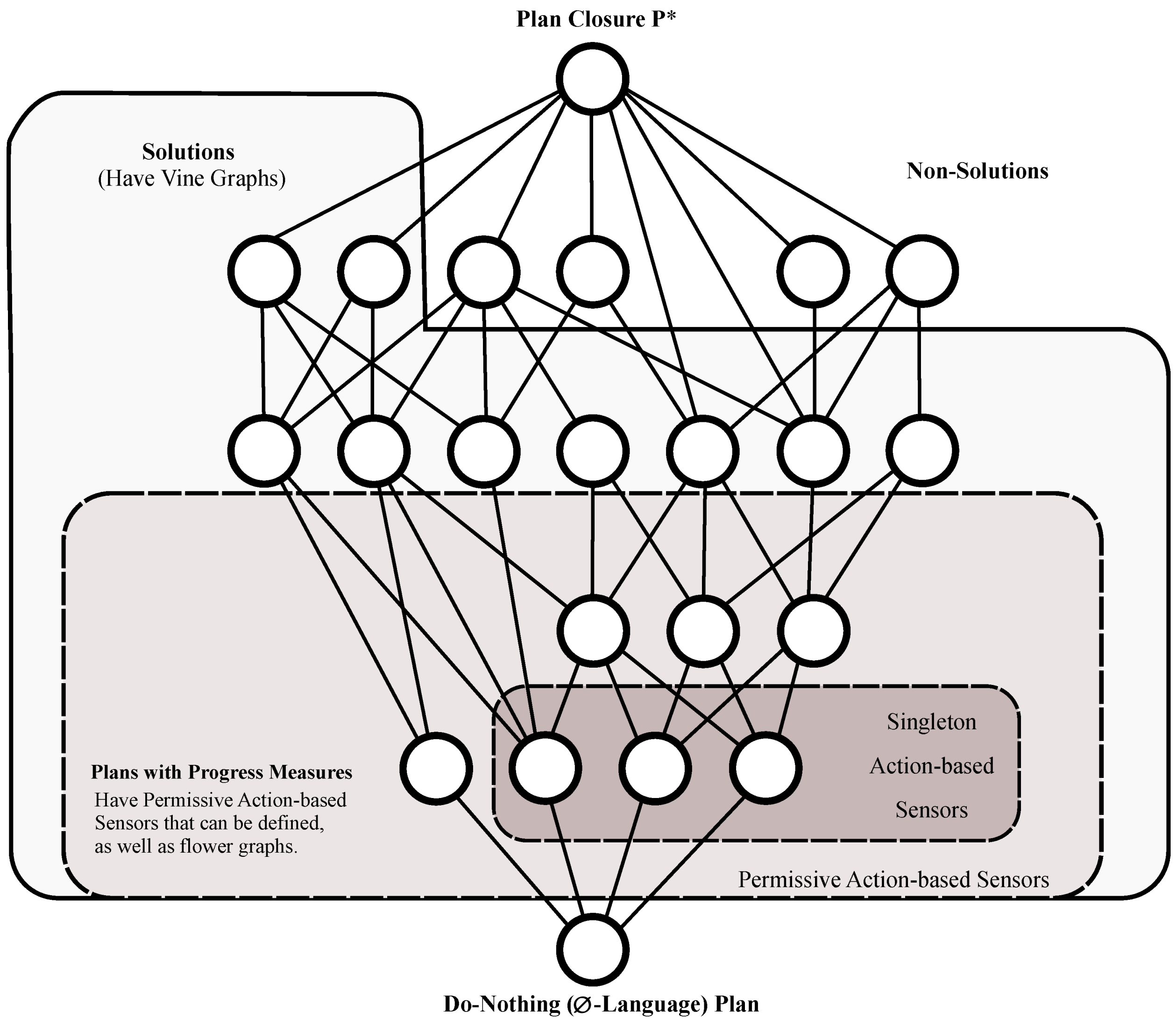

Plan Closure P*
Solutions
(Have Vine Graphs)
Non-Solutions
Singleton
Action-based
Sensors
Plans with Progress Measures
Have Permissive Action-based
Sensors that can be defined,
as well as flower graphs.
Permissive Action-based Sensors
Do-Nothing (∅-Language) Plan

Figure 3.12: A sketch of a *plan lattice*.

non-solutions. We call the maximal element of the plan lattice the *plan closure*, borrowed from Zhang et al. (2018).

Definition 18 (plan closure (Zhang et al., 2018))**.** *Given a world* (W, V_{goal}) *and a (potentially infinite) set of solutions for* W $\{P_1, P_2, , P_3, \ldots\}$*, there exists a p-graph* $P^\star$*, which we term the* plan closure*, such that*

$$\mathcal{L}(P^\star) = \mathcal{L}(P_1) \cup \mathcal{L}(P_2) \cup \mathcal{L}(P_3) \ldots .$$

Using the plan lattice, we can bring action-based sensors, as well as their upcoming stateful counterparts, into a single construction. At the level of their languages, this allows for us to examine how the combinations of certain executions induce requirements of state, as well as in what instances memorization is actually required. In terms of plans, the set of action-based sensors that are subsumed by a plan are the representatives that would be provided by CLIP; these are the subsets of the original plan language for which no crossovers occur. Through the structure of the lattice, we can see not only that some plans require state while others do not, but also how those requirements come about as well as what kind of behaviors can be modified or discarded to cross between that boundary of reactivity and statefulness.

3.10 What Lies Beyond Flower Graphs

Figure 3.11 presents a flower graph form of a plan that solves the problem presented in Figure 3.2; the current location of the robot in the world has no bearing on what action should be taken next. If we consider a plan such as that in as in Figure 3.5, the robot's current observation is insufficient for us to determine what actions will make progress. Only through also knowing where the robot is within the plan, which is specific to the route the robot previously committed to (blue or red, in that example), can we determine the robot's future actions.

This observation is the basis for what we consider *state*.

3.10.1 Defining State, Defining Vines

If state can arise implicitly from plan structure, then it can also be expressly used to encode execution history for our own needs. This raises the question of how to capture only the information needed to execute the full language of the plan. For example, a plan's graph could branch off two different ways depending on if a door is open or closed. While keeping some history is vital for the robot to determine what actions to take in the future, other information can be discarded without impact.

Plans which can be transformed into a reactive plan with an action-based sensor were keeping information about their execution that was unnecessary, while plans with only execution progress measures rely on their structure to encode information so that they can achieve the goal.

3.10.2 The Impact of Non-homomorphism

Non-homomorphism generally arises from uncertainty in the robot's model of the world. This may be due to uncertainty in the robot's physical components, such as sensors and actuators, but may also arise due to the structure of the plan itself. Even if we can determine a robot's precise location within the world given its joint-execution history, a single vertex within the plan may correspond to different world vertices each time it is visited.

This changing relation between plan vertices and world vertices is captured by the I-Graph. It is also possible for the I-Graph itself to create a non-homomorphic plan, even if the input was homomorphic to the world. By definition, the initial vertex of the I-Graph corresponds to all starting vertices in the world. The empty string ϵ would then potentially end in any starting location in the world, making the plan non-homomorphic. This trivial case aside, this transformation can also occur when obtaining an action-based sensor. After CLIP operates on the I-Graph, the algorithm attempts to reduce the I-Graph down into a reactive plan (as an action-based sensor). Accordingly, the existence of the vertex progress measure means that world vertices that have the same incoming observations will be conflated within the final graph, resulting in a non-homomorphic plan.

3.10.2.1 Ambiguity and Dynamism in Plans

To discuss the various ways in which plans can be non-homomorphic, we extend the relation presented in Definition 15. This extension augments the association to encompass not only the correspondence between how a plan vertex and world vertex can be reached with a joint-execution, but also *which* joint-executions reach those pairs of vertices in P and W. If we disregard s in this definition, we see that this relation again becomes the one defined in Definition 15.

Definition 19 (triple relation). *For a plan P and planning problem W, we can define a relation $T \subseteq V(P) \times V(W) \times (\mathcal{L}(W) \cap \mathcal{L}(P))$, where a tuple $(v, w, s) \in T$ if and only if there exists a joint-execution s on P and W that can be traced to a vertex v in P and a vertex w in W.*

To explain how different types of non-homomorphism arise and are handled, we present two new definitions. The first captures that in nondeterministic plans, the same joint-execution may result in the robot reaching different vertices in the world. We say that plan vertices with this relation are *ambiguous*.

Definition 20 (ambiguity in plans). *A vertex v in plan P is* ambiguous *if there exists a joint-execution s and vertices w, w' in planning problem W, where $w \neq w'$, such that both (v, w, s) and (v, w', s) are in T.*

Ambiguity is a statement about our ability to determine the outcome of a single joint-execution, all else being equal. Such nondeterminism may arise from sensor noise, or from lack of precision in the robot's actuation. Imprecision in location can also lead to the possible world vertices of the robot moving from set to set. In addition to ambiguity, it is also possible that the same vertex in the plan will map to different vertices in the world depending on the current joint-execution:

Definition 21 (dynamism in plans). *A vertex v in plan P is* dynamic *if there exist joint-executions s, s' and world vertices w, w' in planning problem W, where $s \neq s'$ and $w \neq w'$, and both $(v, w, s) \in T$ and $(v, w', s') \in T$.*

Plan vertices with a changing relation to which vertices in the world they correspond to are *dynamic*. The plan in Figure 3.10b models a version of dynamism. If the robot receives the observation o_G at the start of execution and immediately terminates, the vertex within V_{term} could correspond either to the goal A or the goal G. However, if the robot receives any other observation at the start of its execution, it will (eventually) localize to a single location within the world. At that point, when it reaches the goal, the execution could tell us for certain whether the goal reached was A or G. (Although the definition of dynamism uses single vertices such as w, w', the principle also applies over sets, as seen here.) Although these definitions are properties of individual vertices, we can broadly refer to plans as ambiguous or dynamic. Additionally, plans can be both ambiguous and dynamic, just ambiguous or just dynamic, or neither.

The impact of ambiguity in plans can be seen when creating an I-Graph, where ambiguity results in some vertices in the I-Graph mapping to sets of world vertices as opposed to single vertices. This is the partially observable case discussed throughout Section 3.5, which can easily lead to requirements of state. By contrast, dynamism disappears during the I-Graph creation. The I-Graph's purpose is to remove the constraints of the plan structure through relating plan vertices to world vertices. As dynamism results purely from plan structure, we see that while the same plan vertex may correspond to multiple vertices in the I-Graph, this has no impact on the final I-Graph itself.

Progress measures and action-based sensors can be defined for some non-homomorphic plans, and Section 3.8 describes the requirements for action-based sensors to exist. For those plans for which action-based sensors cannot be defined, we now further extend CLIP to recognize what information must be retained, and use this knowledge to reconstruct those plans as vine graphs.

3.10.3 From Execution Progress Measures into Vines

The final part of our algorithm defines a new vine graph based on an execution progress measure. For each instance of a given observation, the algorithm obtains a list of actions from the I-Graph. If the set of actions differs depending on the instance, then the observation is one which the robot must keep track of.

This graph is constructed similarly to the method used to create the I-Graph. Starting with an initial vertex for the vine and the initial vertex for the I-Graph, the algorithm will trace the I-Graph's possible executions in order to construct the new vine graph.

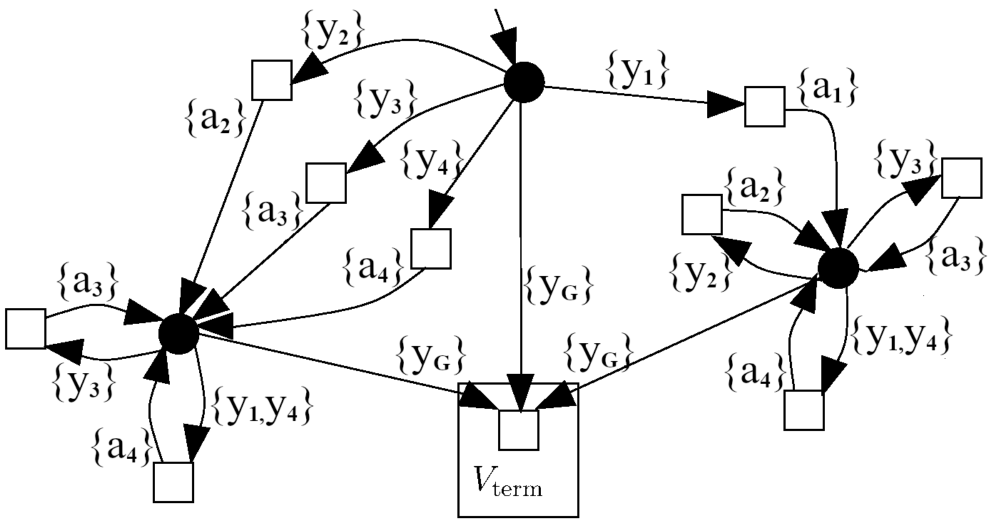
Given a set of outgoing edges from an observation vertex within the I-Graph, one of two cases may occur. Either the set of outgoing observations includes one that the robot must remember, or it does not. In cases such as the plan in Figure 3.10b, the robot may receive the observation o_1 at the start of execution, which requires it to take a different action than when o_1 is encountered at any other point in the plan. Considering the definition of a contextual sensor in Definition 14, the plan starts in a context f_1, where o_1 corresponds to action a_1. Upon receipt of o_1, a new flower graph is created, reached by the execution $o_1 a_1$. This flower has its own context, f_2. However, if o_1 is not encountered as the first observation at the start of execution, then for subsequent o_1 encounters the robot should take the action a_4. Therefore, the other observations (collectively) transfer to a new flower of their own, a context f_3, representing that the robot needs to remember that an observation *was not* encountered.

In the case that none of the outgoing observations in a set are ones that need to be memorized, they can be added to the vine in the same way a normal action-based sensor is constructed, with the observation going to an action vertex that loops back to the observation.

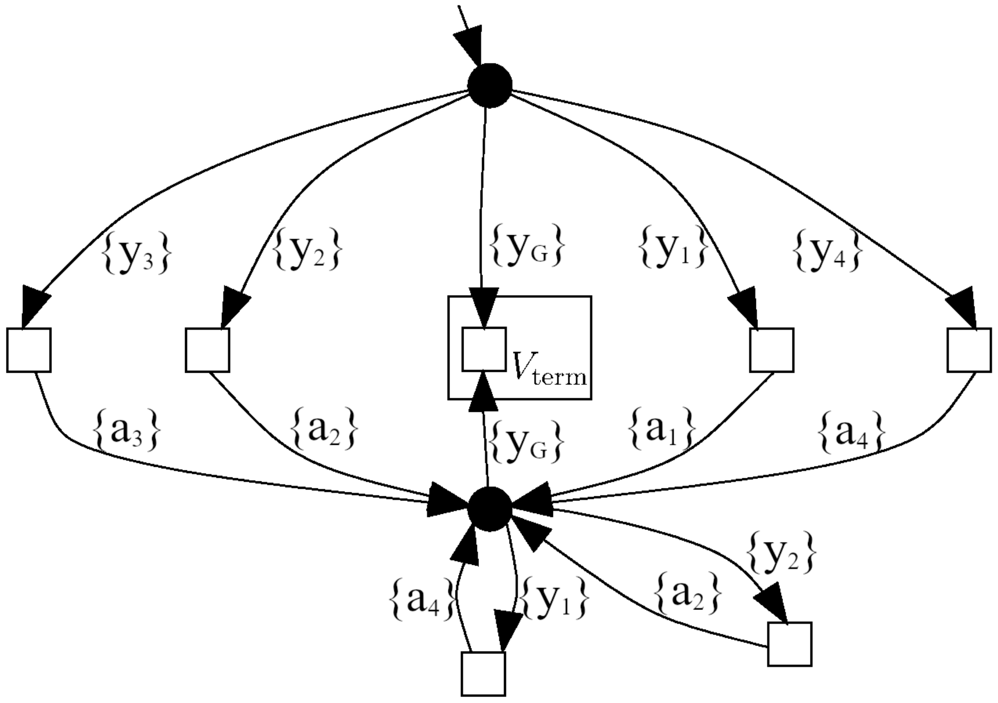
In addition to this branching, it is also possible for the robot to stop tracking information once it is no longer valuable. For example, for both contexts f_2 and f_3, o_1 will always correspond to the same action throughout the rest of execution. Therefore, the list of observations to be tracked is updated after branching. If the action vertex reached by an instance of the observation does not have conflicting actions with any other instances that may follow it, that observation is removed from the list.

Regardless of whether the path branches or not, or if a new action vertex is created or not, the search continues on the I-Graph vertices reached by these observations and actions until all executions have finished. After this, the various goal vertices are then merged into a single vertex. The end result is a graph consisting of several individual flowers, each connected to others by a transitional observation and action. An example of the vine graph for the plan in Figure 3.10b can be seen in Figure 3.13a.

Keen readers may notice that the p-graph in Figure 3.13a is not the smallest possible representation of this plan. This plan can be represented with only two observation vertices, as show in Figure 3.13b. One of these flowers corresponds to the start of execution, while the other represents



(a) The resulting vine given by the CLIP algorithm.



(b) A small vine that makes use of only two observation vertices.

Figure 3.13: Two vine graphs for the plan in Figure 3.10b.

that after that initial observation, no additional information needs to be kept. After the initial o_1, all other instances of o_1 have an action in common, and therefore do not need to be distinguished.

While in this instance the two-flower graph could be obtained by checking that all further instances of the observation agreed on actions before the branch, instead of after, it should be noted that in general the question of obtaining a concise plan for a general planning problem W is NP-complete. O'Kane and Shell (2017) explain this further in their discussion of how filter reduction differs from minimization of a deterministic finite automaton:[4]

> '...we do not require the reduced filter to produce identical results for every observation string in Y, but only on those observation strings in $\mathcal{L}(F)$. In practice, this means that the reduced filter may generate colors for observations strings [*sic*] that are not in the language induced by the original graph, which allows I-states to be "merged" even when their outgoing edges differ.'

To translate that terminology: the filter is represented with F, the set Y remains the set of all observations, and I-states can be thought of as individual observation vertices. Although p-graphs and filters are slightly different structures, they share the complexity that arises from the difference between their structure and their interaction language. An example of this concept applied to p-graphs is the action-based sensors themselves, which have an infinite language when considered alone, but which have a finite language when reduced to their joint-executions with the world. (Though beyond the scope of this work, interested readers are referred to O'Kane and Shell (2017) for further detail and presentation of heuristic algorithms for concise planning.)

Despite not necessarily being the minimal graph, the resulting vines yield useful information about the plan execution. Each branch out from a flower into two more indicates an additional bit of memory required by the robot to track the decision. The resulting vine structure can be examined for various features, such as if certain choices lead to future requirements for memory when compared to other executions, in which case elimination of certain actions can reduce overall

[4]A filter (O'Kane and Shell, 2017) is a directed graph that is somewhat similar to a p-graph, but which consists only of observations on its edge transitions and yields an output "color" dependent on the given state. The complexity of p-graph plan reduction follows from similar structure.

memory requirements. The changes in sensor requirements that result from previous choices also become more apparent within the vine's structure, which may be used to inform design choices.

3.10.3.1 Turning Flowers into Vines

In addition to our ability to create vines for objects that only possess execution progress measures, we can also use vines to combine sets of action-based sensors. In the case of a crossover, CLIP will produce two (or more) action-based sensors, each of which captures part of the original language of the plan from which they were derived. In certain cases, this behavior may be undesirable; as such, recovering the full set of executions requires construction of a vine.

By definition, each action-based sensor disagrees with all other children on at least one action. Separation is therefore required if one does not want to re-introduce crossovers.

There are a variety of ways to implement this; an action-based sensor could be selected nondeterministically from the set at the start of execution, or a vine graph could be constructed based on a concept of "least commitment," maintaining the largest possible set of action-based sensors until the robot must eliminate ones inconsistent with its current history of actions. The desired behavior and resulting structure is up to the designer.

3.11 Applications

This section presents several examples of applications and extensions of action-based sensors and vine graphs to various domains. First, Subsection 3.11.1 demonstrates how action-based sensors and vines can be used for robot design, as well as for modeling operation under sensor failure. Next, Subsection 3.11.2 provides an example in which the vertices of the world correspond to information states as opposed to a physical configuration. Finally, Subsection 3.11.3 explores an example from Liang et al. (2020) of how a robot can navigate using signage systems for humans. We show that the algorithms presented in the current chapter can be used to determine if the robot can obtain sufficient information in the environment for navigation.

3.11.1 From Action-based Sensors to Vines

Throughout this chapter, we have claimed that the structure of the action-based sensor and vine graphs can be of value in answering questions of design. The example in Subsection 3.6.1 shows an instance in which broadening the family of action-based sensors allows for accommodation of unusual constraints on the environment. As we will now show, the fact that vines and action-based sensors can both be found by CLIP means that designers can also model how small changes to the world and available sensing information impact the operability and requirements of a robot over time.

As an example, imagine that the robot we have been designing is a rover being launched to another planet. For obvious reasons, once the robot is in service it will not be able to be repaired or upgraded. In addition to ensuring that our robot is capable of sensing all information needed to remain operational and perform science, as designers we would also like for it to have as long an working life as possible — for example, by retaining some functionality even in the face of partial component failure.

As this system will naturally have a large number of components, we will focus on a single (simplified) subsystem: power management. Our hypothetical robot has several sensors that monitor its power consumption and current battery levels, and can temporarily shut down systems that are drawing too much power. Additionally, our robot has solar panels that it uses to charge these batteries. To assist the positioning of these panels, the robot has several other tools: sensors for light direction and current, which detect the position of the Sun and how quickly the batteries are charging. It also has a set of motors, which it can use to fine-tune the position of its solar panels, to maximize light exposure.

As numerous other systems will be vying for power and chip space, we would like the controller for managing the robot's charging to be as simple as possible. While continually pointing the solar panels towards the Sun may gather the most possible current, it restricts the bearing of our robot and requires constant adjustment of the panels. Instead, the power system examines the energy demands and the current battery level. If power demands are high, or if the battery is low, the robot can prioritize adjusting the solar panels to maximize the incoming current, and throttle other systems to keep the battery from dying. If the battery is at a high level, or power demands are low, the robot can divert power from systems for adjusting the panels and their charging circuitry to other systems for exploration of the planet.

While this power management must be performed for as long as the robot functions, we can still describe this problem as a task with a goal. Each time our robot checks on the state of its power systems, we can define its goal state based on the power load. If the load is balanced with the incoming current, or if the battery is high and the load is small, the system can terminate. If the system is unbalanced, the robot should make adjustments until the power consumption is stable.

Figure 3.14 shows an example action-based sensor for this behavior. We consider that our observation set consists of two readings; first, the current load of the system may be high, low, or balanced when compared to the charge current, designated $l_{\text{HIGH}}, l_{\text{LOW}}, l_{\text{BAL}}$. We also know the general level of the battery: high, low, or critically low, designated $b_{\text{HIGH}}, b_{\text{LOW}}, b_{\text{CRIT}}$. A single observation $y \in Y$ is a pair from these two sets, indicating the current state of the system. (We also assume that the observation set includes the ability to sense the input current, $current_{\text{HIGH}}$ and $current_{\text{LOW}}$, which are also included in any given observation, but for clarity in the figures they are omitted until relevant.)

For actions, our robot may move the solar panels to face the Sun, lay the solar panels in a neutral (flat) position, or turn non-vital systems off. (We assume that, if not being held in an off state by the power system, the robot may turn non-vital systems on as needed.) This set is designated $U : \{u_{\text{move\_to\_sun}}, u_{\text{lay\_flat}}, u_{\text{nonvital\_OFF}}\}$. (As part of moving the panels, we assume the robot can tilt its panels left and right, and so these actions are also included in U. These individual actions will become relevant later in the example.)

Our robot faces many potential failures in space. In particular, the above example uses an action to move the panels towards the Sun — which presumes that our robot can always identify that direction, and that this sensor is working. Although not part of the explicitly defined observation set, this observation is used to drive whatever routine moves the panels towards light. If this ability

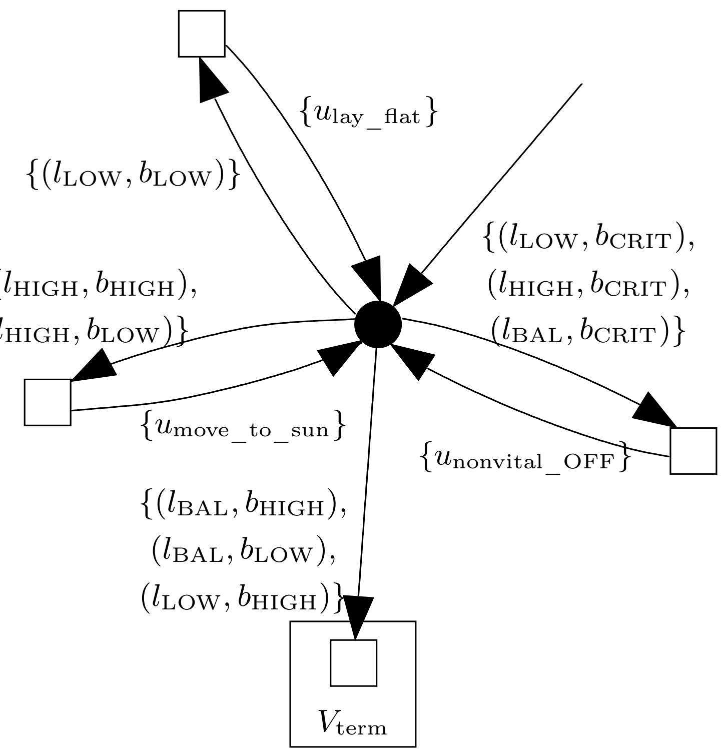

$\{u_{\text{lay\_flat}}\}$
$\{(l_{\text{LOW}}, b_{\text{LOW}})\}$
$\{(l_{\text{LOW}}, b_{\text{CRIT}}), (l_{\text{HIGH}}, b_{\text{CRIT}}), (l_{\text{BAL}}, b_{\text{CRIT}})\}$
$\{(l_{\text{HIGH}}, b_{\text{HIGH}}), (l_{\text{HIGH}}, b_{\text{LOW}})\}$
$\{u_{\text{move\_to\_sun}}\}$
$\{u_{\text{nonvital\_OFF}}\}$
$\{(l_{\text{BAL}}, b_{\text{HIGH}}), (l_{\text{BAL}}, b_{\text{LOW}}), (l_{\text{LOW}}, b_{\text{HIGH}})\}$
V_{term}

Figure 3.14: An action-based sensor for a system with the goal of balancing power consumption for a robot. This plan is finite on the world; even if the load is not balanced, as systems are turned on and off, eventually the battery level will be high and the load will be low. If an I-Graph was created for this, our world states would correspond to input currents, battery levels, and loads. The progress measure would therefore be defined as the choices in action that prioritizes either a balanced load, or a high battery level with minimal load.

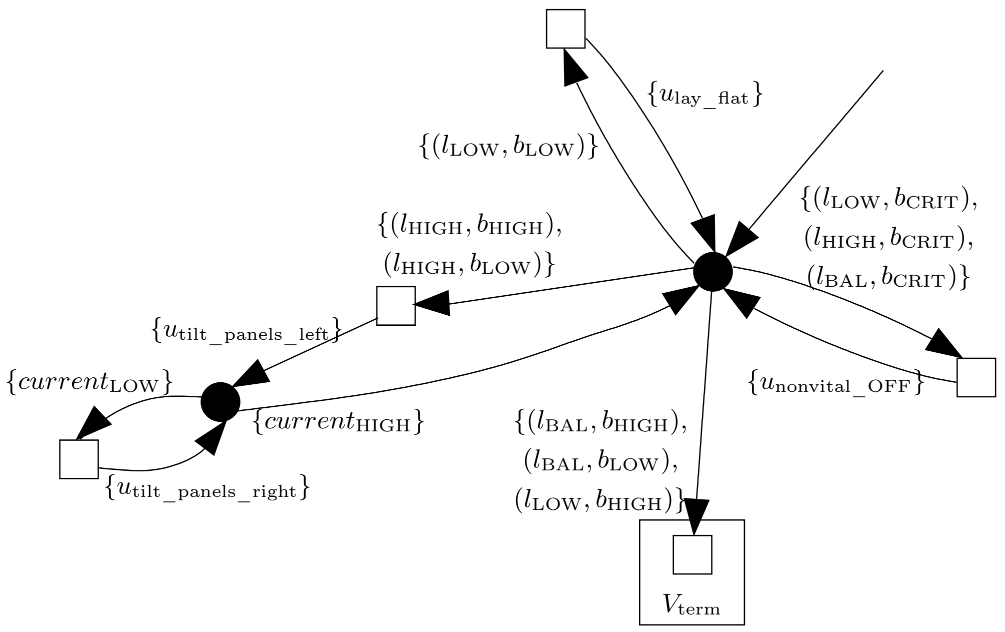

$\{u_{\text{lay\_flat}}\}$
$\{(l_{\text{LOW}}, b_{\text{LOW}})\}$
$\{(l_{\text{LOW}}, b_{\text{CRIT}}), (l_{\text{HIGH}}, b_{\text{CRIT}}), (l_{\text{BAL}}, b_{\text{CRIT}})\}$
$\{(l_{\text{HIGH}}, b_{\text{HIGH}}), (l_{\text{HIGH}}, b_{\text{LOW}})\}$
$\{u_{\text{tilt\_panels\_left}}\}$
$\{current_{\text{LOW}}\}$
$\{current_{\text{HIGH}}\}$
$\{u_{\text{tilt\_panels\_right}}\}$
$\{u_{\text{nonvital\_OFF}}\}$
$\{(l_{\text{BAL}}, b_{\text{HIGH}}), (l_{\text{BAL}}, b_{\text{LOW}}), (l_{\text{LOW}}, b_{\text{HIGH}})\}$
V_{term}

Figure 3.15: A vine graph demonstrating behavior that adapts to a broken light sensor. We exclude the panel current, $current_{\text{HIGH}}$ and $current_{\text{LOW}}$, from edges where they are not relevant to the action taken, and the same is done with the load and battery observations when they are not relevant to the resulting choice in action.

fails, then the robot must develop new behaviors for when it has high loads on the system. In such a case, although the Sun cannot be directly sensed, its direction can be generally inferred from what angle yields the highest input current. Therefore, our robot now must use a multi-step process to maximize this input. Beginning with a maximum tilt in one direction, it can then adjust the tilt of the panel in the other direction. The output current should rise as it faces a light source, and decrease as it moves away.

Figure 3.15 shows an example vine graph for this behavior. Observations which would have triggered the robot to adjust its panels towards the Sun now transition to a new observation vertex. Slightly abusing our observation labeling, we ignore the current load and battery level and focus only on the input current. Assuming the robot is not keeping past information, the robot should return to normal behavior once the current reaches a certain level. This behavior also makes use of actions for adjusting the panels, $u_{\text{tilt\_panels\_right}}$ and $u_{\text{tilt\_panels\_left}}$.

Another way in which the system might fail is that the motors for adjusting the angle of the solar panels may fail. If they fail at an angle, the robot will have to physically turn to face the Sun. This will result in a similar vine as the one before, although the robot will turn towards a light instead of adjusting panels to hit a desired current, but will also impact other systems on the robot. The requirement of facing in a certain direction to charge the batteries means that other actions, such as exploration and movement, are restricted by the battery level and the direction in which it must face.

Although we cannot service our robot once it leaves Earth, we can predict ahead of time what kind of problems may occur, and how this changes the sensor readings and desired responses of the robot. As a result, we can construct vine graphs that suggest new behaviors to compensate, and identify how those behavioral changes may result in impacts on other parts of the system.

3.11.2 Representing Information Spaces: Part Manipulation via Squeezing

In the examples presented in Figures 3.2 and 3.5, both p-graphs have vertices which directly correspond to locations within the world. However, recall that 3.10.2 shows that the relation between a plan's vertices and world vertices is not always one-to-one. Furthermore, an individual

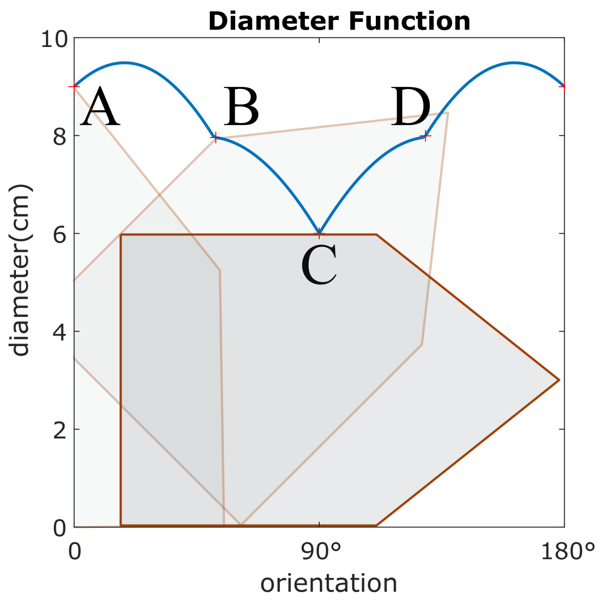


Figure 3.16: A diameter function obtained by rotating a five-sided polygon that describes our part.

vertex in the world need not correspond to a single configuration. This section presents an example of a p-graph in which the vertices represent information (or belief, or knowledge) states. The properties of solutions to this planning problem are discussed, and an action-based sensor is given for a problem variant.

In an influential early paper, Taylor et al. (1988) present a near sensorless approach to the problem of part orientation using a parallel-jaw gripper. One feature of this problem is that the robot can very quickly reduce a potentially infinite number of starting configurations into a finite set. Figure 3.16 shows a function of a five-sided part's diameter as it is rotated. The crosses on the line of the function indicate stable configurations and are labeled with a letter; as the part is rotated and squeezed, it will fall into one of these configurations depending on its current orientation.

Given that the measurement of the diameter of the part is not directional, the function is mirrored across the line where orientation $= 180^\circ$; as a consequence, no matter what information is obtained, every configuration (e.g., A) remains fundamentally ambiguous with respect to its mirror (which we designate A'). Therefore, although the robot may apply a series of rotation and squeezing actions to orient the part, the problem is only solvable up to symmetry.

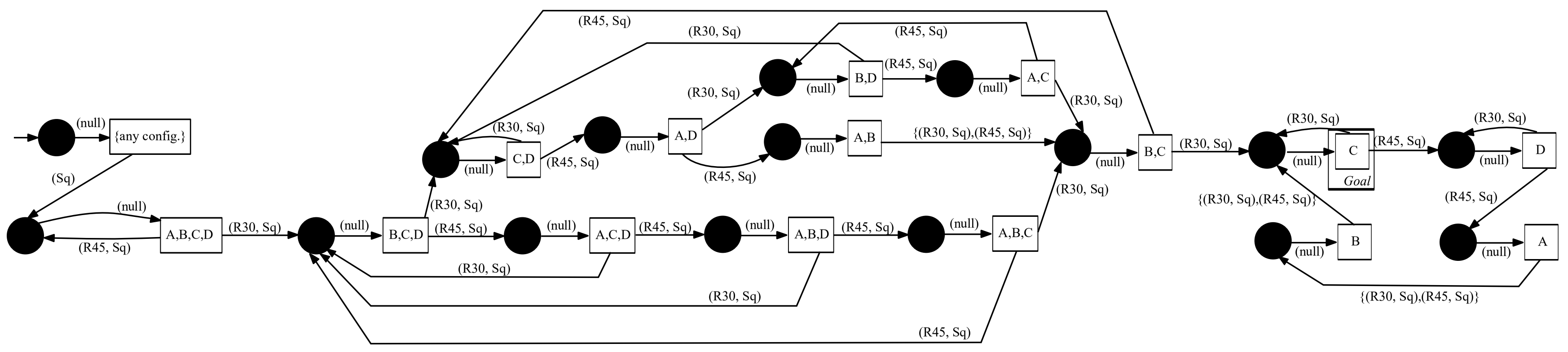


Figure 3.17: A p-graph representation of a part alignment problem given the ability to turn the part $30°$ or $45°$ before squeezing (Sq), as done in O'Kane and Shell (2017). The R on each edge label indicates a *range* of values around the intended rotation, any of which may be the resulting rotation due to imprecision in the robot's movement. Each vertex label (A,B,C,D) cannot be distinguished from that position + $180°$, and so each vertex represents no fewer than two configurations.

Figure 3.17 shows a p-graph model of a robot equipped with a sensorless jaw gripper. (We use the process described in O'Kane and Shell (2017) to construct this model.) The robot is capable of squeezing the part, as well as rotating either 30° or 45° before applying a squeeze. These rotations need not be exact, and we assume that the true rotation may be in a range of a few degrees (approximately $\pm 2^\circ$) from the intended movement. Due to the squeezing operation that follows a rotation putting the part into a set of fixed states, the error caused by this imprecise movement does not accumulate over time. The initial vertex of the p-graph corresponds to any of the infinite starting configurations of the part. Application of the gripper at the start of execution reduces this set to eight states: the three shown in Figure 3.16, the configuration indicated by D, as well as the mirror of these four configurations. The goal vertex is the first state distinguishable, C (and C').

Solving of this problem requires that the robot shift between turning the part 30° and 45° before squeezing. However, this shifting has no pattern, and so the sequence of rotations must be memorized and executed in a specific order. Additionally, because the robot cannot recognize the goal state through observation, this memorized sequence is the only way in which the robot can be certain that the part has been properly aligned.

We now augment the robot with a sensor which allows for it to sense when the grippers have closed at a diameter less than $7\,\mathrm{cm}$, which corresponds to a single pair of configurations: C, C'. Given this modified world and the ability to recognize goal attainment, it is then possible to construct an action-based sensor for the robot. Instead of tracking the number of steps it has taken to determine what action to take next, the robot can rotate the part repeatedly by 45° to move between stable configurations until the sensor signals the goal has been reached.

3.11.3 Recent Applications: Large-scale Navigation in the Built Environment

The recent work of Liang et al. (2020) in robot navigation has explored the idea of how robots may make use of signage designed for humans for their own wayfinding. As part of this work, they identified several properties of signage within spaces. They analyze signs (e.g., directional markers, arrows, etc.) by constructing navigation graphs. For signage to lead to well-formed

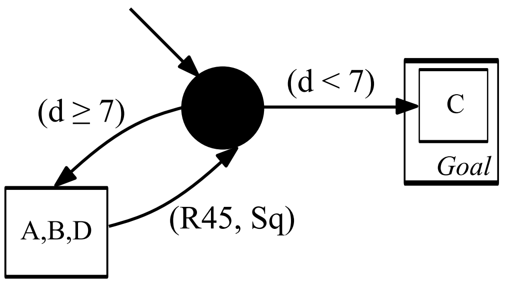


Figure 3.18: An action-based sensor for aligning the five-sided part to a known configuration, given the ability to sense if the diameter of the jaws are open beyond a threshold of 7 cm.

navigational graphs, signs should be *consistent*, *fully specified*, and *complete*.

To briefly summarize the definitions given in their work, consistency indicates that arbitrary pairs of signs are in agreement with their directions and do not have conflicting instructions on any edge. Fully-specified graphs are those for which all edges in the graph are directed. Complete signage assumes that for some chosen goal, there exists a path from all other points to that goal along the directed edges given.

Such properties are similar to those discussed in this chapter for action-based sensors. Fully-specified navigation graphs are similar to the planning problems considered in this work where agents may begin from any possible vertex in the world; just as those problems are designed with the goal of defining general action-based sensors that operate anywhere in the world, full specification of a signage system allows for navigation regardless of starting location.

The requirement for consistency, where signs do not conflict on which way along an edge the robot should move, prevents the robot from moving back and forth on the same edge endlessly. This cycling behavior forms a very small crossover between the two vertices to which the edge connects. Consistency is a property of multiple signs labeling a single edge. A different failure, and one that is captured by CLIP, is the existence of larger crossovers that span multiple edges.

The resolution of these crossovers and the existence of an action-based sensor may also seem similar to a complete signage system. A complete signage system provides enough information for a robot to navigate without knowledge of its history or location in the world; similarly, action-based sensors are an indication that what can be sensed about the environment is sufficient for goal attainment. However, unlike the existence of a progress measure, complete signage systems have no guarantee of all paths to the goal being finite; the generation of signage systems in Liang et al. (2020) makes use of a planner, which results in the implicit enforcement of finiteness.

Far from being a flaw in either approach, these differences highlight the ways in which CLIP's formulation results in additional generality. The ability to specify arbitrary plans to CLIP means that additional care must be taken to ensure that the resultant action-based sensors are actual solutions. However, this also allows for CLIP to handle problems of different varieties, such as those that are partially observable or which have nondeterministic outcomes. As a tool for analysis of plans and sensors, CLIP combines the potential robot's capabilities with a model of the world that can itself be constrained by the robot's limits. All of these things together allow for it to be applied to a wide range of problems.

The following problem shows an application of CLIP to a particular example. For this specific example, much faster procedures exist to generate a plan from any arbitrary point to a goal (and therefore to determine what directions must be on a sign). This example, like the ones before it, highlights the ways in which the generality of CLIP allows its analysis to be useful in a variety of circumstances, and to accommodate differing settings. The question of how a robot should navigate can be framed as a variant of the sensing problem that motivates this work, where the relation between what is sensed and what action is taken is externalized into information on the sign itself.

Figure 3.19 shows a graph derived from an airport within the dataset analyzed by Liang et al. (2020). Given a chosen goal, there exists from each point in the graph a single correct direction to move to make progress towards that goal. Using CLIP, it is possible to determine what direction the agent must travel from a current point — indicating what information should be provided on a sign.

Originally, the 34 edges in Figure 3.19 are defined as bi-directional. As each bi-directional edge is itself a crossover between its two endpoints, there exist 2^{68} ways in which to resolve all crossovers, depending on what direction is chosen for each edge. However, only one of these

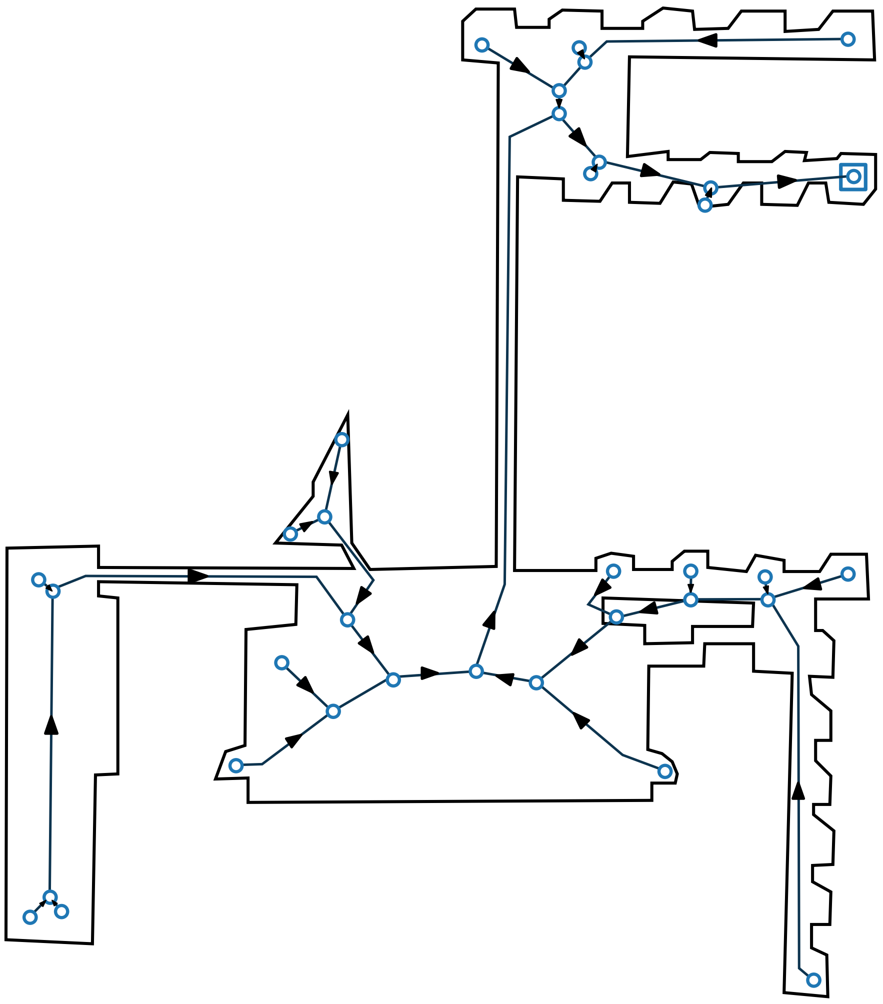


Figure 3.19: A connectivity graph derived from the layout of Palma de Mallorca (PMI) Airport. An arbitrary node of the graph has been chosen as the goal, and is indicated with a square border.

potential solutions is correct in the sense that it will allow all starting points to have a path to the goal. The graph in Figure 3.19 shows the solution for an example where an arbitrary vertex, indicated with a box, is the goal. Given these directed edges, the signs at each vertex can be labeled to ensure that the robot makes continual progress toward the goal.

As previously mentioned in 3.5.3, it is possible for a human to provide input that can assist CLIP in focusing the search by resolving certain crossovers manually, which reduces the search space. In such navigational examples, there are several possible sources for such input. One can incorporate the information from existing signs into the graph, eliminating some of the bi-directional edges altogether. One might also use existing knowledge of the airport layout to resolve some crossovers, such as the fact that there exist multiple offshoot regions connected by a central area, and that this central area must be visited to move from one of these regions to another. The manual resolution of some crossovers can also be used to impose constraints or reflect other higher-level preferences, such as to direct robots through different areas than humans.

3.12 Conclusion

Having extended the work done by Erdmann we can now obtain not only the sensors that correspond to the backchained solution, but the complete set of all action-based sensors. Given a library of components, this set of sensors can guide practitioners in determining what is simultaneously realizable, respectful of environmental constraints, and sufficiently powerful to accomplish the task. Design and selection of sensors then becomes just a question of intersecting constraints.

However, we have seen that sensing alone may be inadequate, leaving our robots to explore, to learn, and to memorize the information they need. Through modeling plans as directed graphs, the required internal state is encoded directly into their structure. Identifying what precisely this state is, how it manifests in stateful plans, and what properties these stateful plans have, is valuable in understanding how limitations in one aspect of the design (sensing) can be compensated for by other components of the system.

By identifying only the information that must be retained, we gain additional insight into the plan's requirements, and can express these requirements through vine graphs, which extend action-based sensors.

From the perspective of the theorist, vine graphs and action-based sensors provide a useful tool for analyzing information and state requirements of systems. By comparing closely related plans within the plan lattice, the effect of minor changes in behavior on the system can be easily explored. Additionally, the structure of the plan lattice allows for understanding of where requirements for internal state begin, and where sensing alone becomes insufficient to solve the task.

From a designer's perspective, the action-based sensors and vine graphs allow for direct comparison of designs. Given a planning problem, the sensing limits imposed by different sensors that are available to the designer can be applied to the planning problem and its resulting solutions. The resulting action-based sensors can be compared to see if the different sensors require different behaviors, or if the required change in behavior to implement them is suitable for the desired task. If a vine graph is required, then the variation in sensors can be used to compare the resulting complexity of the plans in terms of their state.

We have also seen that the ability to compare the application of different sensors allows for exploration of additional questions, such as if changes to the same sensor over time result in a planning problem that can no longer be solved, or if the changes can be planned around through the use of more complicated behaviors from the robot.

It is our belief that the tools presented in this work provide a new way for roboticists to explore the tricky connections between desired behavior and required sensing, as well as how choices in either of those areas can have significant impact on the options for the other.

4. INFRASTRUCTURE DESIGN

4.1 Introduction

A growing body of work proposes computational approaches to robot design, including research on the selection/optimization of actuators and sensors that meet some desired level of performance while balancing cost and efficiency (Censi, 2017; Carlone and Pinciroli, 2019; Desai et al., 2018; Shell et al., 2021). Such work considers elements that are part of, and physically internal to, the robot itself. While it is generally understood that structured environments ease many of the challenges involved in developing and deploying useful robots, this paper approaches the idea with a fresh twist by treating this within a design problem.

Consider a group of autonomous tug vehicles deployed in a specially instrumented fulfillment center. It is not entirely obvious for such a system where the boundary of the "robot" ought to be:— aside from the mobile tug shunting things around, the instruments and the facility itself are key to the vehicle's efficacy. This paper introduces a method for making informed design choices involving infrastructure *for* robots; the focus is on environmental elements possessing infrastructure-like attributes, which is not solely a question of physical placement (external vs internal). To begin with then, the nebulous concept of infrastructure will be clarified, including the distillation of a collection of common traits that lead to certain design questions and considerations.

4.1.1 Informal Definitions and Chapter Contributions

Infrastructure is commonly used to describe a variety of services and projects that are made available to large numbers of users. Here, we aim to informally identify the traits that are often common across different kinds of infrastructure, which will provide this paper's working definition. Infrastructure that we examine has these features:

▸ **Group Utilization.** Multiple agents are able to access the infrastructure to benefit from it; e.g. roadways, satellites, etc. While it may not be available to *all* agents, it should service some group of individuals.

▸ **Elastic Scaling.** It should be capable of supporting the intended number of users and of being extended in the future.

▸ **Reusable.** Infrastructure should endure multiple uses before being consumed, repaired, etc. and not be perishable.

▸ **Cost Distribution.** Recuperation of the upfront construction and maintenance costs are distributed over the users in some way; e.g. taxes, tolls, monthly bills.

▸ **Fairness.** It should not harm any one group unduly.

▸ **Impacts Agent Behavior.** Finally, we expect that the infrastructure should alter the operation of agents in the environment, having some measurable impact.

Given this working definition, we aim to answer the following questions:

- How can infrastructure be formally defined, given the many forms it can take?
- Can we examine how infrastructure will impact large populations of agents without exhaustively simulating them all? Assessment should include some subset of robots altering their behavior to suit new infrastructure.
- How can different proposed implementations of infrastructure be compared?
- What is the impact on agents with differing abilities and goals, and how can that impact be translated into a concept of "fairness"?
- How can the cost of infrastructure be compared to benefits in performance?

In order to formalize the above description, we propose a model that treats robots via Markov Decision Processes (MDPs); the approach prioritizes practicality by being simple and flexible. The infrastructure perspective offers a subtly different approach to improving robot performance from standard methods. Later sections in the paper will discuss specific instances of infrastructure and demonstrate its promise.

4.2 Problem Formalization

Our treatment is intended as a tool to be applied to a variety of questions and across a range of settings: accordingly, this generality demands that the presentation be quite abstract. The basis of this treatment is an underlying set of MDPs that are transformed by infrastructure choices. These elements will then be instantiated to fit particular scenarios (e.g., where domain knowledge is used to form the appropriate MDPs, decide on specific concrete 'changes', etc.). Among the variations considered are scenarios in which agents are oblivious to the introduction of infrastructure, adapt to it, or some combination thereof.

4.2.1 Preliminaries

We model each agent's interactions with its environment, indicated as M, as a Goal MDP, which we recall next (Bertsekas, 2019):

Definition 22 (Goal Markov Decision Process). *A Goal Markov Decision Process, or MDP, consists of:*

- *a finite nonempty set of states S, with an initial state $s_0 \in S$;*
- *a finite nonempty set of actions A;*
- *a transition model $T : S \times S \times A \rightarrow [0, 1]$ such that $T(s', s, a) = \Pr(s'|s, a)$ denotes the probability of arriving in state s' having issued action a from state s;*
- *a function $C : S \times A \times S \rightarrow \mathbb{R} \cup \{+\infty\}$ where $C(s, a, s')$ is the expected cost expended for the agent occupying state s taking action a and arriving at state s';*
- *a nonempty selection of goal states $G \subseteq S$.*

An agent's behavior will be described via a policy:

Definition 23 (Policy). *A policy $\pi : S \rightarrow A$ assigns, for each state $s \in S$, an action $\pi(s) \in A$ for an agent to take.*

While we will discuss optimal policies in Section 4.3 for comparative purposes, the ensuing definitions apply to all potential policies for an MDP.

Definition 24 (Expected Cumulative Cost)**.** *For an MDP $M = (S, s_0, A, T, C, G)$, the* expected cumulative cost *of policy $\pi : S \to A$ is*

$$E[\pi|M] = \mathop{\mathbb{E}}_{s_0, s_1, \ldots, s_m} \left(\sum_{i=0}^{m} C(s_i) \right),$$

where the expectation is over finite sequences $s_0, s_1, \ldots, s_m$ arriving at some goal state $s_m \in G$, with probabilities $T(s_i, s_{i-1}, \pi(s_{i-1}))$, for all $i \in \{1, \ldots, m\}$. If there are no such sequences with positive probability, then we declare $E[\pi|W] = +\infty$.

4.2.2 Defining and Applying Infrastructure

In the following, we will distinguish $K \in \mathbb{N}$ different *classes* (or types) of agent.

Definition 25 (Environment)**.** *For K classes of agent, an* environment *$\mathcal{E}$ is a collection of MDPs, one for each class:*

$$\mathcal{E} := \{M_1, M_2, M_3, \ldots, M_K\}.$$

In order to determine the value of a proposed piece of infrastructure, each MDP must be modified to "apply" the effects of the new infrastructure. Accordingly, we formalize infrastructure as a transformation on environments that results in altered transitions and changed costs:

Definition 26 (Infrastructure)**.** *A collection $\mathbb{I}$ of K triples is termed* infrastructure *if we have $\mathbb{I} := \big\{(h_1^{\mathrm{T}}, h_1^{\mathrm{C}}, c_1), \ldots, (h_K^{\mathrm{T}}, h_K^{\mathrm{C}}, c_K)\big\}$, with each triple $(h_i^{\mathrm{T}}, h_i^{\mathrm{C}}, c_i)$ comprising two maps and a scalar:*

- *function h_i^{T} mapping from the original transition function of MDP M_i to a new transition function T':*

$$T_{M_i}(\cdot, \cdot, \cdot) \overset{h_i^{\mathrm{T}}}{\longmapsto} T'_{M_i'}(\cdot, \cdot, \cdot);$$

- *function h_i^{C} mapping from the original cost function C of MDP M_i to a new cost function C':*

$$C_{M_i}(\cdot, \cdot, \cdot) \overset{h_i^{\mathrm{C}}}{\longmapsto} C'_{M_i'}(\cdot, \cdot, \cdot); \quad \textit{and}$$

- $c_i \in \mathbb{R}^{\geq 0}$, *an associated construction cost.*

For each $i \in \{1, \ldots, K\}$, the pair h_i^{T} and h_i^{C} modify MDP M_i, altering its transition and cost functions, to give the new MDP M_i'. Then, infrastructure $\mathbb{I}$ is an operator that modifies some environment $\mathcal{E} \overset{\mathbb{I}}{\mapsto} \mathbb{I}(\mathcal{E}) = \{M_1', M_2', M_3', \ldots, M_K'\}$.

In the above definition, each triple $(h_i^{\mathrm{T}}, h_i^{\mathrm{C}}, c_i)$ contains an associated construction cost c_i, which represents the one-time cost of constructing the infrastructure, not to be confused with the cost functions of the MDPs within $\mathcal{E}$.[1]

While here we consider only the initial price of construction, much infrastructure requires continual upkeep and consideration of maintenance costs incurred over time is the topic of Section 4.4. The benefit provided by this infrastructure's construction relies on several factors: the number and types of agents within $\mathcal{E}$, if (and how) agents change their behavior in the presence of infrastructure, and how this affects the expected cumulative costs of all agents. To formalize these aspects, we start with the fact that environments are inhabited by (typically multiple) agents:

Definition 27 (Agent population). *An* agent population $\mathbf{P}$ *for environment $\mathcal{E} = \{M_1, M_2, \ldots, M_K\}$ is a collection of sub-populations, each representing a collection of a (fixed) number of agents of a particular class, along with a policy describing their behavior:*

$$\mathbf{P} \coloneqq \{\mathscr{P}_1, \mathscr{P}_2, \ldots, \mathscr{P}_{|\mathbf{P}|}\},$$

where each sub-population $\mathscr{P}_i \coloneqq (n_i, c_i, \pi_i)$ has $n_i \in \mathbb{N}$ agents of class $c_i \in \{1, \ldots, K\}$, whose behavior is modeled as following policy $\pi_i : S(M_{c_i}) \to A(M_{c_i})$.

(In the preceding, we have wrtten $S(M)$ for the states of MDP M; also, $A(M)$ for actions, analogously.) The connection between environments—including those modified by forms of infrastructure—and populations is captured in the next definition.

[1]We require that each MDP's costs and infrastructure cost c_i be in the same units—in our examples, the MDP's cost is converted to dollars so we may look at recouped costs. Determining the equivalent "worth" of a cost function is not always straightforward, but picking other units may help.

Definition 28 (Infrastructure response)**.** *For environment $\mathcal{E}$, an* infrastructure response *is some rule that takes $\mathbb{I}$ and a given agent population*

$$\mathbf{P} = \big\{(n_1, c_1, \pi_1), (n_2, c_2, \pi_2), \ldots, (n_{|\mathbf{P}|}, c_{|\mathbf{P}|}, \pi_{|\mathbf{P}|})\big\}$$

and produces

$$\mathbf{P}' = \big\{(n'_1, c'_1, \pi'_1), (n'_2, c'_2, \pi'_2), \ldots, (n'_{|\mathbf{P}'|}, c'_{|\mathbf{P}'|}, \pi'_{|\mathbf{P}'|})\big\}$$

such that, for each $k \in \{1, \ldots, K\}$:

$$\sum_{\substack{(n_i, c_i, \pi_i) \in \mathbf{P}, \\ c_i = k}} n_i \quad = \sum_{\substack{(n'_i, c'_i, \pi'_i) \in \mathbf{P}', \\ c'_i = k}} n'_i.$$

The intuition is that an infrastructure response reflects a change in a population where there may be a different apportioning of sub-populations, but where the total agents of each class is preserved. Clearly, also, the total number of agents in the population is conserved.

Next, to make this more tangible, we provide some concrete examples of infrastructure responses:

▷ An *oblivious utilization* is the identity infrastructure response $\mathbf{P} \mapsto \mathbf{P}$ regardless of infrastructure $\mathbb{I}$.

▷ Given any operator $\mathbf{S}(\cdot)$ that produces a policy from an MDP,[2] an $\mathbf{S}$-*based fully adaptive utilization* is the infrastructure response $\mathbf{P} \mapsto \mathbf{P}'$ where each $(n_i, c_i, \pi_i) \in \mathbf{P}$ becomes $(n_i, c_i, \pi'_i) \in \mathbf{P}'$ where π'_i is a policy obtained via $\mathbf{S}(M'_i)$, assuming $\mathbb{I}(\mathcal{E}) = \{M'_1, M'_2, M'_3, \ldots, M'_K\}$.

▷ Again using $\mathbf{S}(\cdot)$, for adoption rate $\alpha \in [0, 1]$, the *adoption-based utilization* is the infrastructure response $\mathbf{P} \mapsto \mathbf{P}'$ with every $(n_i, c_i, \pi_i) \in \mathbf{P}$ contributing two elements to $\mathbf{P}'$:

1) $(n_i - \lfloor \alpha \cdot n_i \rfloor, c_i, \pi_i) \in \mathbf{P}'$, and

2) $(\lfloor \alpha \cdot n_i \rfloor, c_i, \pi'_i) \in \mathbf{P}'$, where π'_i is obtained via $\mathbf{S}(M'_i)$, assuming $\mathbb{I}(\mathcal{E}) = \{M'_1, M'_2, \ldots, M'_K\}$.

For the final case, when the adoption rate $\alpha = 0$ or $\alpha = 1$, one recovers the two previous instances.

As the term "infrastructure response" connotes, these are representations of how populations of agents react to changes made to the world. Even if a change has taken place in the environment, agents may not be aware of this, resulting in oblivious utilization. (We will see in Section 4.3.1 that even oblivious agents may see benefits from infrastructure.) Conversely, $\mathbf{S}$-based fully adaptive utilization is where all agents within the population update their policies according to the infrastructure transformation.

The ability of an agent to operate obliviously on a transformed MDP M' is due to the way in which infrastructure was defined. The transformation function h^{T} cannot eliminate any actions available to the agent at that state, although it may change outcomes. (This follows naturally from the fact that infrastructure is only an operation on the world, not on the agent's capabilities.) Thus, while there is no guarantee that an oblivious agent will *succeed*, the original policy π can be used on M'.

The notion of an adoption rate α is used in our analysis of adoption-based utilization to indicate what proportion of a class of agents create an updated policy π'. It models situations in which either some agents remain unaware of the modified environment, or, if aware, choose not to alter their behavior. The realization splits a single sub-population into two groups, one with the identity infrastructure response while the other generates a new policy.

With these elements rigorously defined, we next formalize aspects relating to measurement and evaluation.

Definition 29 (Infrastructure Returns). *Given environment $\mathcal{E}$, population $\mathbf{P}$, and infrastructure $\mathbb{I}$ which produces $\mathbf{P}'$ after some infrastructure response, the* infrastructure returns *over all classes K is:*

$$\underbrace{\sum_{(n_i', c_i', \pi_i') \in \mathbf{P}'} n_i' \cdot \Big(E[\pi_i'|M_i']\Big) \; - \sum_{(n_i, c_i, \pi_i) \in \mathbf{P}} n_i \cdot \Big(E[\pi_i|M_i]\Big)}_{\textit{Change in Agent Costs}} \; + \sum_{(h_k^{\mathrm{T}}, h_k^{\mathrm{C}}, c_k) \in \mathbb{I}} c_k,$$

[2]Specific instances of $\mathbf{S}(\cdot)$ might be Value- or Policy-Iteration solvers, or some Reinforcement Learning method.

The first summation in this expression is the expected cumulative cost of all agents within the population $\mathbf{P}'$, while the second summation is the expected cumulative cost of the original population $\mathbf{P}$. Together, these give the change in expected cumulative costs under the infrastructure response, and the final term includes the upfront construction expense for the infrastructure. The total is the final cost incurred by all agents under the response to infrastructure $\mathbb{I}$ on $\mathcal{E}$.

Infrastructure returns allow one to determine the "break-even" point of a proposed piece of infrastructure. By changing the original population sizes n_k for each $(n_k, c_k, \pi_k) \in \mathbf{P}$ or by modifying adoption rate α, the change in agent costs may be adjusted until the returns are equal to zero, at which point the original construction costs have been recouped. Section 4.4 discusses the use of break-even points.

While the break-even point of a proposed piece of infrastructure depends on the population of each agent class, these populations may also be thought of as a usage rate where each agent of class k must make use of the infrastructure n_k times before saved execution costs equal expended construction costs. Interpretation of the model in such a way disregards potential outside interference from other agents—this was deliberate, as we are interested in the average behavior of agents over time and assume the base MDP to be reflective of the agent's normal interactions in the environment, already including any regular interference. The case studies in Section 4.3 also omit inter-agent interference.

4.2.3 Formal Definitions of Informal Concepts of Infrastructure

We may now precisely discuss some features of infrastructure treated informally before. The following defines impactfulness:

Definition 30 (Impactful Infrastructure). *Given environment $\mathcal{E}'$ derived from $\mathcal{E}$ and infrastructure $\mathbb{I}$, $\mathbb{I}$ is considered* impactful *if:*

1. *For at least one adaptive agent class k, $\mathbf{S}(M_k') \neq \mathbf{S}(M_k)$ or, alternately, $\pi_k' \neq \pi_k$, (under operator $\mathbf{S}(\cdot)$) and*

2. *$E[\pi_k' | M_k'] \neq E[\pi_k | M_k]$.*

The definition reflects the two aspects of impactfulness discussed in Section 4.1.1: $\mathbb{I}$ should both alter the operation of agents (through adaptive agents generating a policy different to their original) and have a measurable impact (through the new policy having a different expected cumulative cost).

Impactful infrastructure may further be classified by the manner in which it affects agents, and changes in the expected cost of policies can be used to determine if infrastructure is beneficial to an agent or not. Generally, if the expected cost once an agent has adapted is higher than the cost of the original policy on the original MDP, $E[\pi'|M'] > E[\pi|M]$, the infrastructure is *harmful* to the agent. Conversely, if $E[\pi'|M'] < E[\pi|M]$, the infrastructure has reduced overall expected costs and is considered *beneficial*.

Some infrastructure may also have the goal of inducing behavior change, oftentimes for some social engineering aim (Thaler and Sunstein, 2009). Instances that seek to "encourage" behavior π' over π may introduce infrastructure that changes $E[\pi'|M] \geq E[\pi|M]$ to $E[\pi'|M'] \ll E[\pi|M']$. As this entails neither $E[\pi'|M'] < E[\pi|M]$ nor $E[\pi'|M'] > E[\pi|M]$, it is neither directly beneficial nor harmful. Possible modifications could be via a penalty/punishment system, where $E[\pi|M'] \gg E[\pi|M]$ becomes $E[\pi|M'] > E[\pi'|M']$, or via rewards, where $E[\pi'|M] \geq E[\pi|M] \approx E[\pi|M'] \gg E[\pi'|M']$.

Each class of agent has an individual infrastructure response to some $\mathbb{I}$. The resulting individual changes in expected costs may be lost when we look at infrastructure returns across all classes. By examining classes individually, we can determine if any one group is harmed unduly by changes in costs, satisfying the definition of fairness as given in Section 4.1.1.To do so, we introduce the idea of a *fairness boundary* ϵ, which allows us to compare the original expected cost against the new expected cost for individual classes. (Choosing the numerical value of "fairness" is naturally a problem-dependent and somewhat abstract decision. We show in Section 4.3.3 an example that makes use of such a boundary as part of the decision-making process.)

Definition 31 (Fairness)**.** *Given a population* $\mathbf{P}$*, infrastructure* $\mathbb{I}$*, and resulting population* $\mathbf{P}'$*, as* *well as a fairness boundary* $\epsilon \in \mathbb{R}^{\geq 0}$*,* $\mathbb{I}$ *is* ϵ-fair *if, for all classes* $k \in \mathbf{P}$*:*

$$E[\pi'_k|M'_k] - E[\pi_k|M_k] < \epsilon \cdot E[\pi_k|M_k] \tag{4.1}$$

This captures one aspect of fairness: changes in average costs. While this captures the definition given in Section 4.1.1, there exist many notions of fairness, particularly when it concerns multiple disparate groups. Another (Rawls, 1999) definition of fairness is that the biggest benefits should be experienced by the class that had the highest costs in the original world, which we define as follows:

Definition 32 (Fairness, Again)**.** *Given a population* $\mathbf{P}$*, infrastructure* $\mathbb{I}$*, and resulting population* $\mathbf{P}'$*,* $\mathbb{I}$ *is* fair *if there exists a class* $k \in \mathbf{P}$ *where the following both hold:*

$$\max_{(n_i,c_i,\pi_i)\in\mathbf{P}} E[\pi_i|M_i] = E[\pi_k|M_k], \textit{ and} \tag{4.2}$$

$$\max_{(n'_i,c'_i,\pi'_i)\in\mathbf{P}'} E[\pi_i|M_i] - E[\pi'_i|M'_i] = E[\pi_k|M_k] - E[\pi'_k|M'_k] \tag{4.3}$$

Yet another concept of fairness is the idea that the fairest option is the one that results in benefits for the largest number of agents:

Definition 33 (Fairness, Yet Again)**.** *Given a population* $\mathbf{P}$*, infrastructure* $\mathbb{I}$*, and resulting population* $\mathbf{P}'$*,* $\mathbb{I}$ *is* fair *if, for all classes* $k \in \mathbf{P}$*:*

$$\sum_{(n'_i,c'_i,\pi'_i)\in\mathbf{P}'} n'_i \forall (n'_i, c'_i, \pi'_i) s.t.[E[\pi'_k|M'_k] < E[\pi_k|M_k]] \geq \sum_{(n'_i,c'_i,\pi'_i)\in\mathbf{P}'} n'_i \forall (n'_i, c'_i, \pi'_i) s.t.[E[\pi'_k|M'_k] > E[\pi_k|M_k]]. \tag{4.4}$$

We see that these differing definitions of fairness are focused on different aspects of how agents will be affected by the introduction of a piece of infrastructure $\mathbb{I}$. The first definition, Definition 31, seeks to limit how much any one class can gain or lose from the introduction of infrastructure. The second definition seeks to provide the greatest improvements to the class with the highest costs relative to the other classes. The third seeks to provide the greatest benefit to the largest number of agents. Despite these differences in definition, the model is capable of handling all of these conceptions of fairness, as well as others.

Although we do not formally define it here, another way to consider fairness is through how cost of the infrastructure itself is distributed between agent classes. Each triple $(h_k^{\mathrm{T}}, h_k^{\mathrm{C}}, c_k)$ assigns to a class k a given cost which is paid by members of that class, and it is possible to impose bounds on how much the cost can differ between agent classes in a manner similar to that of the boundary given in Definition 31.

4.2.4 Agent Interaction

Readers may notice that the above definitions do not involve any direct notion of interaction between agents. The change of an agent's policy within the framework is a result solely of the application of infrastructure, and not from the influence of other agents in the space. This may seem contradictory to our intentions: infrastructure, by definition, is intended to manage large number of agents. This is not an oversight, but instead an intentional choice based on the types of questions this chapter seeks to answer.

While we can think of the expected value of a policy as relating to a single robot executing a policy one time, it is also reflective of what value we would expect the costs of many agents over time to trend towards. Section 4.3.3 uses the expected cost as averages for different types of agents, which are then used as the basis for determining how construction costs are recouped over time. In cases where interference is rare, one can expect that the impact of any interference that is encountered would be lost when considering the average costs. This viewpoint allows for us to use the results of a single MDP to extrapolate to the behavior of arbitrarily large groups of agents without simulation of each individual.

The following section, Section 4.3, presents several examples where we consider the assumption of non-interference to be reasonable. For example, Section 4.3.3 considers travel times over footpaths in a park. A park's sidewalks are rarely congested with individuals to the point that people cannot move; it is far more likely that agents are able to avoid interference during most trips through the park. Similarly, one can expect that the delivery robots that are presented in Section 4.3.4 will have different rooms as their destinations, meaning that it will be rare for the robots to seek to occupy the same space and interfere with each other. These case studies suggest that there is a large body of problems of interest that can be treated with a non-interference approach.

However, there do exist other cases where interference is key to the problem. Vehicular traffic, for example, frequently suffers from congestion, and much road infrastructure is dedicated to reducing this congestion as much as possible. To ignore the way in which vehicles interfere with each other would be to lose the core features of the problem itself. Furthermore, some of the effects of infrastructure are indirect, where the induced behavioral change of an agent may have an effect on others that does not become apparent until much later. We acknowledge in Section 4.5 that this is a primary direction for future work.

4.3 Case Studies

Classification of Induced Changes

Definition 26 builds upon two functions: one modifying transitions (h_i^{T}) and another modifying costs (h_i^{C}). However, the way in which infrastructure can alter agent behavior can vary widely. Broadly, we consider the effects on two dimensions *perception* vs *actuation*, and *precision* vs *efficiency*. The case studies to follow exemplify different ways in which agents can be affected (see Table 4.1).

Table 4.1: A guide to the examples presented in Section 4.3.

	Precision	**Efficiency**
Perception	Improved sensing through changes in the environment. Ex: 4.3.2	Improved efficiency through modifying what is sensed. Ex: 4.3.4
Actuation	Improved actuation outcomes through changes in the environment. Ex: 4.3.1	Improved efficiency through modifying actuation outcomes. Ex: 4.3.3

The first way in which we categorize infrastructure's effect is to determine if it impacts an agent's ability to perceive the world, the outcome of agent actions on a world, or both. Transition functions may change the targets of actions or make desirable outcomes more probable, which is an obvious effect on an agent's actuation. However, infrastructure can also modify what an agent senses, and this change also manifests in the transition function. For instance, newly sensed information may allow the agent to transfer to a state that was previously unreachable. We further categorize infrastructure as affecting precision or efficiency. When sub-divided in this way, this taxonomy provides guidance on the types of transformations that an infrastructure tuple $(h^{\mathrm{T}}, h^{\mathrm{C}}, c)$ should perform. Generally speaking, efficiency improvements will lead to changes in transition targets, while improvements to precision will change the probabilities of transitions themselves. The concept follows similarly for impacts on perception. Through the presentation of four quite different case studies, we now demonstrate how the formalism of Section 4.2 can be instantiated concretely, and how it is capable of spanning the classifications in Table 4.1.

4.3.1 Carpeted Care Facility: Actuation that Improves Precision

Within the consumer and healthcare domain, one area of increasing automation is the introduction of robots into long-term care facilities to help handle tasks like medication delivery, laundry, and night patrols (Jacobs and Graf, 2012). Also, "social" robots to encourage interaction and entertain residents are becoming common. The commercialization of robots for caring tasks needs careful consideration of how these robots will interact with residents, staff, and the building itself (Niemelä and Melkas, 2019).

Figure 4.1 shows a communal living space based on existing care facilities in Benbow (2011). The layout includes private living spaces for residents, communal areas (e.g., dining and common areas), and staff-only spaces (the reception, kitchen, housekeeping, and utilities). Robots are being considered for tasks like delivering medication to residents, retrieving laundry, and escorting residents and visitors around the facility. The robot could start at many points and its navigation goal could be almost any interior point.

Low-pile carpet such as berber carpet is common within residential facilities to reduce dust and noise. Though not a major impediment, compared to tile, these carpets increase the energy needed per unit distance. Friction from the carpet can also reduce the accuracy of rotation and

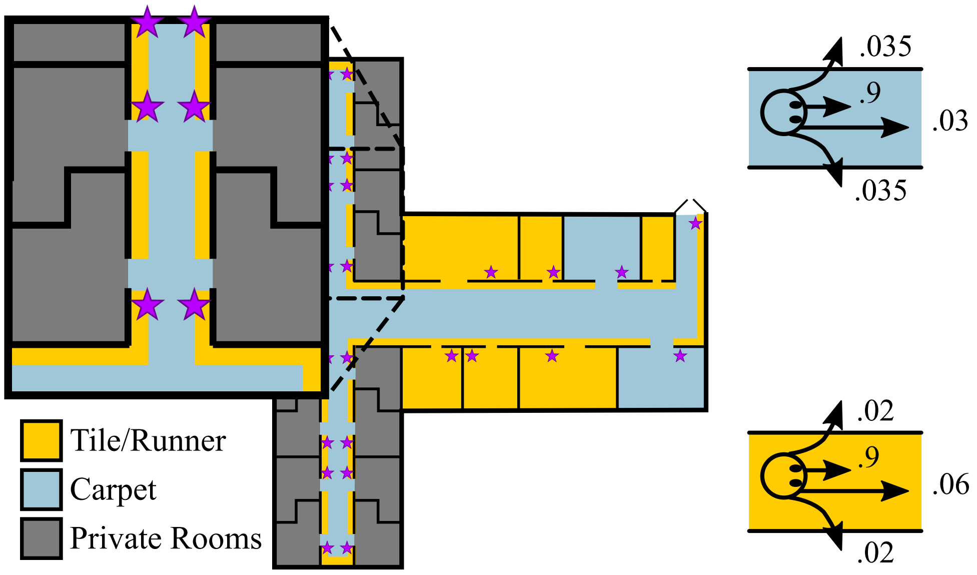


Figure 4.1: An example long-term care facility. Purple stars are potential goals. The robot takes $5\,\mathrm{s}$ to travel $1\,\mathrm{m}$ on carpet, and $3\,\mathrm{s}$ to travel $1\,\mathrm{m}$ on tile. Transition probabilities for the floors are shown.

other movements. Also, robots travelling the same routes many times in a day will accelerate carpet wear. Adhesive carpet runners are protective plastic strips that can be applied on top of carpets. In addition to protecting carpets, they provide a smooth surface that is ideal for a robot to traverse; placed at a hallway edge, they provide an efficient "lane" for robots to use.

Human residents and facility staff result in additional considerations. Transitions between different types of flooring can present a trip hazard, and therefore the laying of paths should seek to balance safety considerations with efficiency requirements of the robots. To attempt to provide increased efficiency while managing these concerns, the runners (shown in yellow) in Figure 4.1 do not cross across hallways or in front of doorways that lead to carpeted rooms. In keeping these areas clear, the tripping danger is reduced while still spanning much of the facility.

In this problem, the MDP's transition costs reflect the time required to move on different types of flooring. Figure 4.1 shows the difference in transition probabilities for carpet vs the carpet runners.[3] Aside from improving the precision of the robot's motion, the cost for the action is also reduced. A time penalty is incurred for collisions with obstacles or walls. The robot's initial and goal locations are selected uniformly at random from the locations marked with stars in Figure 4.1, leading to the expected cost of a robot's trip to be $157.86\,\mathrm{s}$. Keeping the same policy on the updated facility, i.e., oblivious use, resulted in a slight reduction ($16\,\%$ improvement) to an expected trip time of $132.26\,\mathrm{s}$, while adaptive robots had their times improved by $36\,\%$ to $100.68\,\mathrm{s}$.

In the initial configuration, the robot has no preference for any part of the hallway it travels in, and the resulting policy is just the shortest route in terms of distance. The benefits seen in oblivious use are a result of travelling over runners incidentally. With an updated model, the runners become natural travel routes for the robot. Improved motion dynamics yield policies where robots approach the nearest runner en route their destination—the robots staying near walls has the unintended benefit of reducing interference with residents/staff in the hallways.

4.3.1.1 Conceptual Extensions

The placing of specific paths for robots has the benefit that these paths can have custom designs applied during construction, such as distance markers or QR codes. While the current robots in the facility cannot sense this information, future robots might use such patterns for more accurate navigation. Our next case study considers the idea of infrastructure modifying sensing.

4.3.2 Material Handling: Perception that Improves Precision

Figure 4.2 depicts a fulfillment warehouse in which robots assist with loading trucks. Each collects goods from one of three pickup locations (A, B, or C) and transports them to one of three drop-off zones, where other agents perform sorting and finally loading. Where goods appear, and where their drop-off location will be, is assumed to be random (i.i.d.). When not active, the robots occupy one of two maintenance bays; when tasked, the robot could be in either bay with equal likelihood.

The robots use low-resolution cameras to determine if an area is open space or contains an obstacle. Being old technology, these low-resolution cameras are imprecise and the detected obstacles may be differ in size from reality. Consequently, despite there being navigable gaps between the trucks, the robots do not have sufficient certainty to that they will avoid trucks when passing

[3]All examples in Section 4.3 were implemented in Python and based upon the code of Russell and Norvig (2021). Policies were found through Value Iteration.

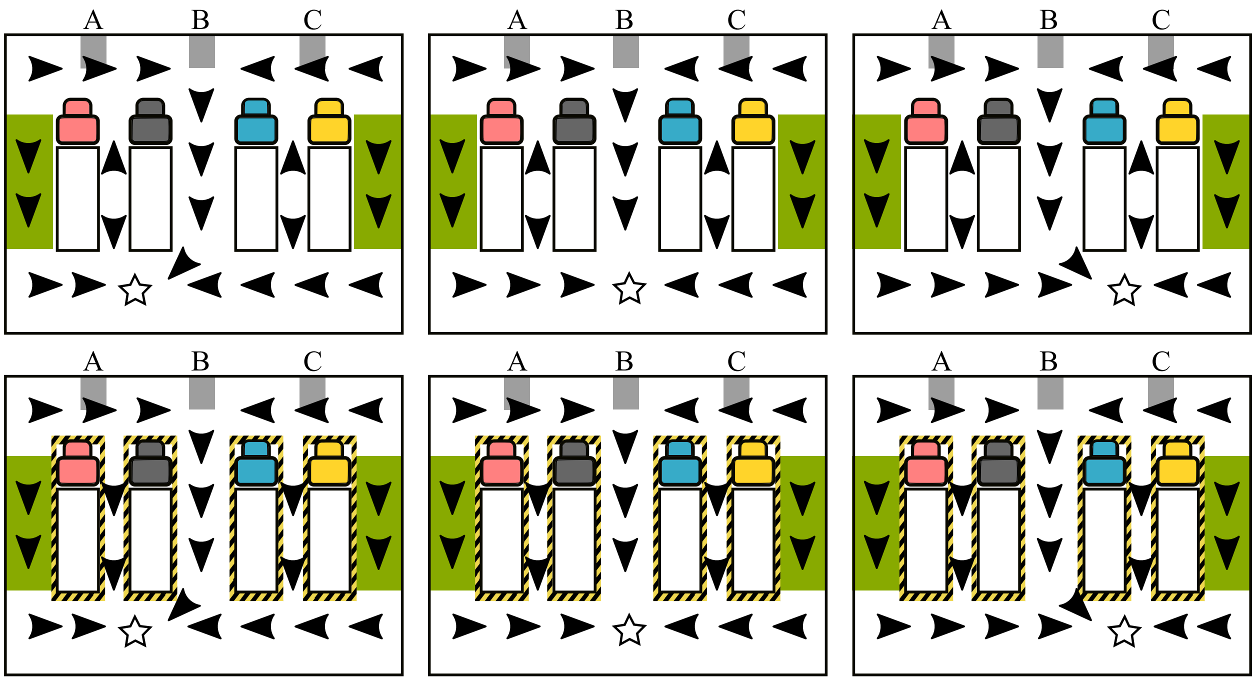


Figure 4.2: Various policies for different drop-off locations in a warehouse, indicated with a star. **Top:** Uncertainty in sensing causes agents to take a longer path to avoid obstacles. **Bottom:** High-visibility markings allow agents to pass between trucks safely.

between them, which would incur a substantial penalty. The top row of Figure 4.2 shows policies under these conditions: robots favor the middle path to avoid uncertainty-induced risk. The warehouse manager wants robots to take the shortest possible path to improve operational efficiency and avoid congestion. However, any environmental modifications must respect her limited budget. She decides to mark the trucks' parking spots with a high-visibility tape, enhancing contrast between trucks and the floor. This improves identification of obstacles, allowing the robots to move between trucks with less risk.

This problem is modeled as multiple MDPs, sequenced together: Starting in a random maintenance bay, the robot is assigned to pick up goods at one of three locations. After achieving this initial goal, that location becomes its new initial state and it is assigned one of three drop-off locations. Finally, the agent begins at a drop-off spot and is assigned one of two maintenance bays to return to. Perception improvements are modeled through the controller: the initial movement model for agents near obstacles has a chance that the agent will drift when moving forward representing the probability of the low-level control loop driving the system forward when the space ahead is misidentified as free when it is not. More accurate sensing reduces the chance of drift, decreasing the chance of being penalized and making movement through narrow areas more attractive.

The top and bottom rows of Figure 4.2 show examples of policies for a robot proceeding from any of the pickup zones to several different drop-off points. As the optimal policy depends on which of the several goals the robot is assigned, it maintains different policies for its various starting locations, pick-up zones, and drop-off locations. The bottom row shows policies after the introduction of high-visibility tape, and this introduction results in agents taking deliberate paths between trucks that they previously would have avoided.

The application of high-visibility tape allows for the current agents to remain in use with minimal environmental changes or cost. This change has the largest impact on the cost for agents to transport loads from pickup to drop-off. Given that the regions between trucks are unlikely to be visited during the other parts of the process, the sequencing of MDPs also enables us to uncover precisely where infrastructure offers the greatest benefit (i.e, the transport step).

4.3.3 Bridges in the Park: Actuation that Improves Efficiency

Two businesses in nearby buildings —separated by a park and river, but with a roadway connecting them— wish to transport goods back and forth (Figure 4.3). Both sides maintain a fleet of robots for transporting goods. The park is popular with employees and visitors, who take walks during their breaks. Both robots and people can access the road, though absence of sidewalks means there is some risk of an accident, incurring a high cost. People prefer shorter routes to maximize the area they can visit in the park during their breaks. Robots also prefer a shorter route to reduce travel time as maintenance is performed after a certain number of hours of service.

The businesses consider two possibilities: (i) speed bumps to slow road traffic and reduce accidents; (ii) a bridge in the park that provides a safer and faster path. While the businesses may favor a bridge positioned to enable fast routes for their robots, employees petition for a bridge facilitating easy travel between landmarks of interest. But what is the benefit of a bridge—for the robot fleet and employees? Will safety and speed justify the expense of bridge construction? If so,

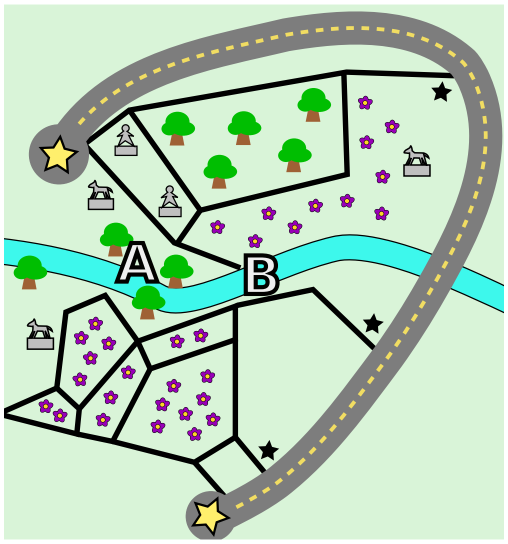


Figure 4.3: A park layout highlighting the complexity of choices involved when modifying environments through infrastructure. The pair of large yellow stars indicate the locations of the two businesses, while smaller stars indicate additional entry and exit points for pedestrians. The two locations labeled **A** and **B** are potential sites for a bridge.

at which location should it be built?

To obtain answers, robots will naturally be treated as if solving an MDP. While humans cannot be controlled *per se*, a policy is still a serviceable modeled: we employ a basic MDP constructed via some simple assumptions—a more sophisticated one, based on observations of how they move through the park (say using inverse reinforcement learning (Adams et al., 2022)) could be used if desired. We indicate the MDP which describes the robots as M_{r}, and the MDP based on human observation M_{h}. The robots may begin at one of the two buildings and have the other as their goal, while the human policy generally has employees returning to the entrance they started at. For this example, we find the optimal policy for the robots, denoted π_{r}. Similarly, we designate the MDPs that have undergone a transformation from infrastructure as M'_{r} and M'_{h}.

The two proposed bridge locations are shown as **A** and **B** in Figure 4.3. The original MDPs (M_{r} and M_{h}) reflect a world without bridges. Three different infrastructure transformations were performed: one that places a bridge at **A**, one putting a bridge at **B**, and one with bridges at both **A** and **B**. Speed bumps are constructed in all cases. Table 4.2 shows expected travel time for various routes, created from the randomly chosen of sub-goals.

Table 4.2: Expected travel times in seconds given the two different bridge locations. The *bumps* column marks the introduction of speed bumps, but no use of bridges. **Top:** The robot fleet moves at a speed of $5\,\mathrm{km/h}$. **Bottom:** The humans move at $3\,\mathrm{km/h}$.

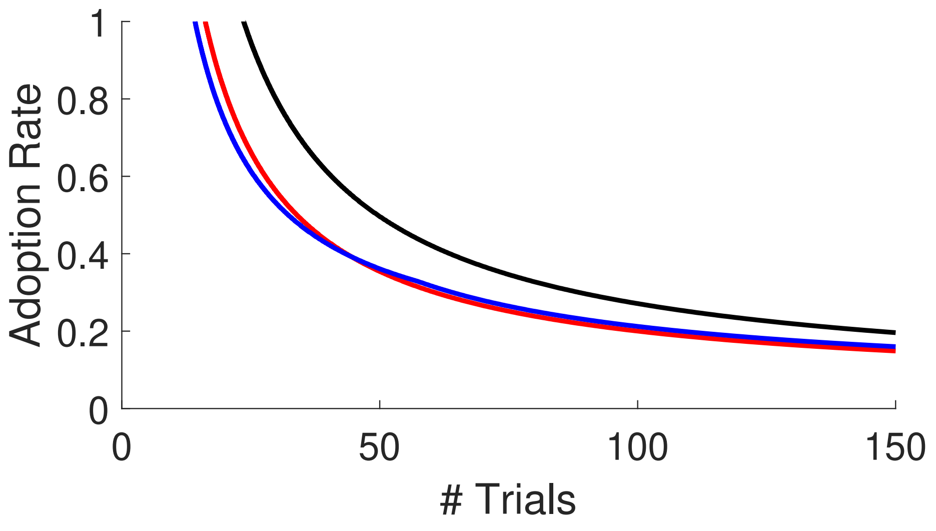
Robot	No Bridge	Bumps	Bridge **A**	Bridge **B**	**A & B**
Route 1	201.7	213.0	115.3	101.9	101.9
Route 2	199.4	210.6	115.3	101.9	101.9
Route 3	236.0	243.7	148.9	119.3	119.3
Route 4	223.5	234.8	211.4	198.0	198.0
Route 5	353.2	361.0	192.0	220.5	192.0
Average	242.8	252.6	156.6	148.3	142.6

Human	No Bridge	Bumps	Bridge **A**	Bridge **B**	**A & B**
Route 1	914.9	914.9	455.1	549.5	455.1
Route 2	672.6	672.6	446.9	348.2	348.2
Route 3	303.3	303.3	303.3	303.3	303.3
Route 4	744.3	744.3	536.2	463.7	463.7
Route 5	1437.9	1437.9	514.1	587.4	451.9
Average	814.6	814.6	451.1	450.4	404.4

Neither robots nor humans take advantage of the infrastructure obliviously; routes are planned to cross bridges only when they're known, while the speed bumps will impact the agents regardless. Humans are unaffected by speed bumps, but the robots have difficulty traversing them and now incur a small additional time cost on the roads. Using Table 4.2, we can define 10 classes, where each class is either a human or robot as well as a given route. Under Definition 31, the lack of change in the humans' routes means that it is perfectly fair to them. However, satisfying this inequality for all robot routes requires $\epsilon \geq .06$. While this may seem like a small change, its impact is magnified over the entire fleet. Therefore, while the speed bumps alone may result in a reduction of accidents, this choice is neither desirable nor fair.

The adaptation of agents results in a new policy π'_{r}, with an associated expected reward $E[\pi_{\mathrm{r}}|M'_{\mathrm{r}}]$. As M_{r}'s cost function represents the total travel time, the difference between the updated expected cost $E[\pi'_{\mathrm{r}}|M'_{\mathrm{r}}]$ and the original expected cost $E[\pi_{\mathrm{r}}|M_{\mathrm{r}}]$ directly reflects the resulting time-saving. For comparison to the implementation cost, the travel time is converted into a dollar amount by pro-rating the cost of maintenance over the amount of time the robot can run between services.

If robots do not adapt to the new environment, the additional time incurred by the speed bumps results in an increase in overall travel time. The introduced infrastructure therefore is detrimental



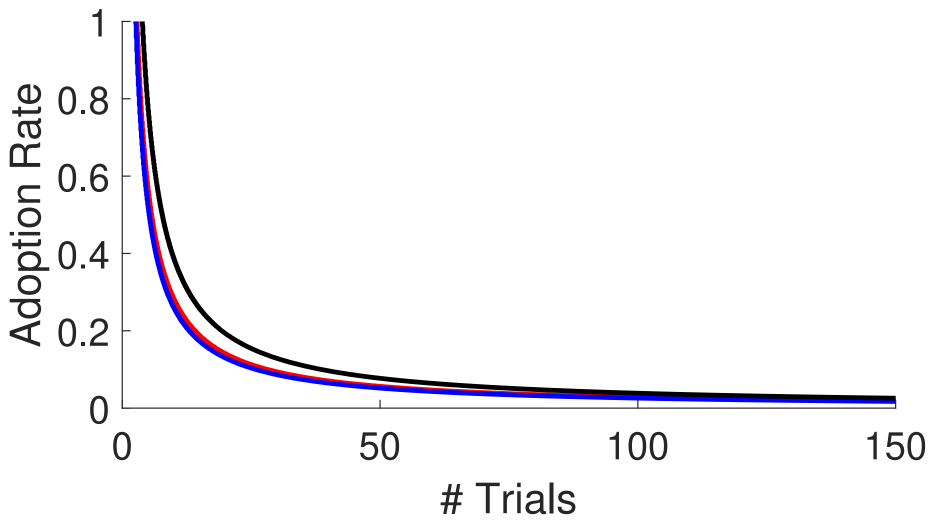


Figure 4.4: Impact of adoption rate and the number of infrastructure uses on recouped costs. These graphs show, for robots **(top)** and humans **(bottom)**, the number of uses needed at different adoption rates in order for saved costs to equal construction costs. Red lines correspond to Bridge **A**, blue lines to Bridge **B**, and black lines to both bridges simultaneously. Each line is a different route.

to oblivious robot agents. However, once either bridge is introduced and the robots adapt and change their policies, the robot agents' costs are reduced.

Figure 4.4 shows, for both types of agents, the number of trips the agents must take to recoup the infrastructure's construction costs. Using the equation in Section 4.2, there exists a non-linear relation between the adoption rate α and the number of trips necessary. For the robots, who incur additional costs on their original routes, higher adoption rates are necessary to offset costs of the oblivious part of the fleet.

We now consider the impact of infrastructure on the people. The assignment of a monetary cost to their trips through the park is more difficult than for the robots. Although the humans are not looking to enter and leave the park as fast as possible (unlike the robots), there are other factors (such as the duration of their break) that mean they still benefit from bridges that reduce their route and prevent them from having to use the roadways. For clarity in this example (and, perhaps, somewhat bleakly), we will consider the time spent on paths to correspond to time spent not working, and therefore shorter paths result in increased profits for the business. Additionally, the introduction of bridges results in humans avoiding the road. This results in far fewer accidents, which is a strong indication that a bridge is worth its construction cost.

The final selection of where to place the bridge gives rise to a consideration of "fairness," wherein the businesses must compare the impact of the bridges on both robots and humans. Suppose that Bridge **A** costs $\$1600$, and Bridge **B** costs $\$1200$. The cost of the infrastructure is recouped directly through reduced traffic accidents and maintenance costs, and indirectly recouped through employees. As employees are not negatively impacted by the introduction of speed bumps on the road, their graph does not show much variation between bridge options; Bridge **B** results in slightly shorter paths on average for employees than Bridge **A**, but both show that costs are quickly recouped even when adoption rates are low. For the robot fleet, the introduction of speed bumps and resulting additional costs requires agents to adapt before any benefit is seen. The trending of the graphs towards a limit as adoption rate decreases also suggests that for a given expected amount of usage there is a minimum number of agents who must adapt for the bridge to be practical. Overall, Bridge **B** recoups costs more quickly than Bridge **A**. Given this, and the lack of strong employee preference, Bridge **B** would be the best option.

4.3.4 The Hotel: Perception that Improves Efficiency

Figure 4.5 shows an example of a hotel with an indoor atrium. Bordering the atrium are 18 rooms for guests, as well as a kitchen for room service. The kitchen uses robots to deliver room service orders (this is one task the service robots may soon play in the hospitality industry, see Lin and Mattila (2021) and references therein). The robot makes use of a fluxgate-like compass to help determine its position and heading in the hotel, and occasionally pauses at certain waypoints to let its sensor settle and obtain an updated reading.

Each room can be set up for four different occupants: a family, an individual on vacation, a couple, or a business traveler. Different occupants have different values for the daily average number of orders they make to room service. Rooms can also be unoccupied, in which case no orders are generated from them. The hotel has two questions: first, given several different potential room layouts, which results in the least travel time for the robots (and thus greater efficiency)? Second, the hotel is considering upgrading the sensors on the robots to a beacon system (such as Gutmann et al. (2013)); what is the benefit of this upgrade, considering the new layouts?

For this example, the cost function includes both the time it takes for the robot to complete its movement as well as the time taken for the robot to obtain and verify a sensor reading. When the robot ends up in a state other than the one intended, there is an additional delay as it confirms this with another sensor reading. The atrium itself is divided into three concentric rings, each of which is divided into 5° slices. These represent the way the robot periodically checks its sensors to ensure that it is staying on course. We use a Poisson distribution ($\lambda = 10$) to model the number of orders received by the kitchen each hour. For each order, the robot's goal is selected with a weighting based on the allocation of guest rooms.

Both types of sensor (compass and beacon) have areas in which they receive interference. The compass receives interference from underground pipes, while there exist certain blind spots from where the robot cannot see the beacon, and must estimate its position through dead reckoning (see

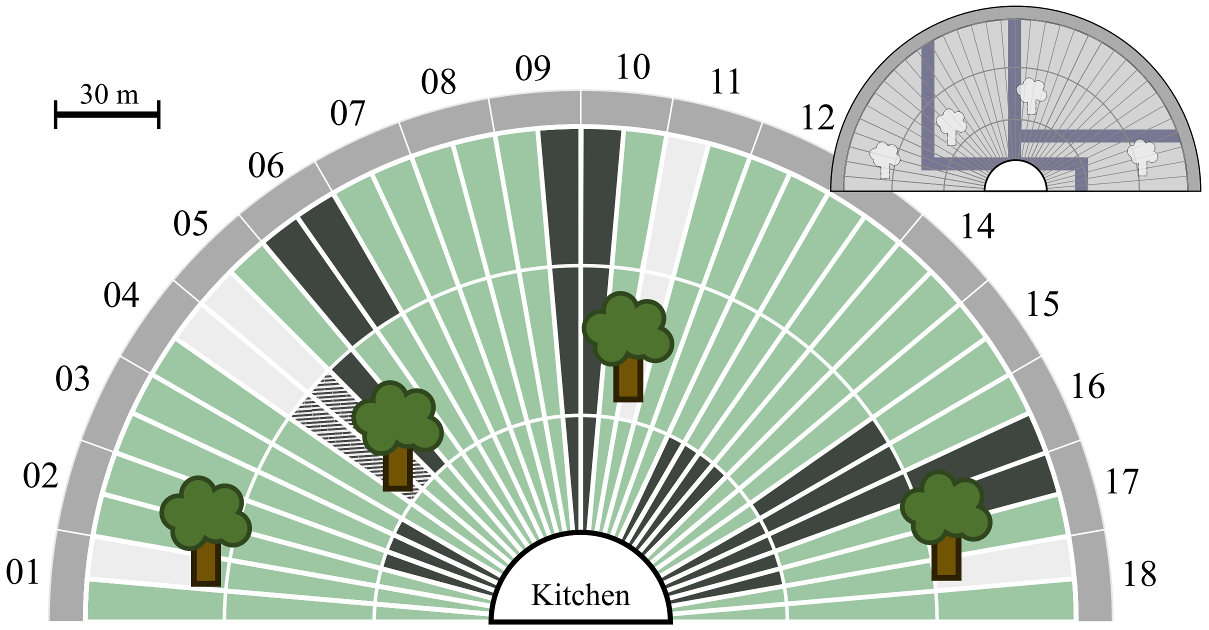


Figure 4.5: An indoor atrium surrounded by guest rooms, with multiple regions where different types of sensor interference can occur. It is divided up into states based on $5°$ segments of three concentric rings. Sources of sensing interference include trees and underground pipes (see inset). Legend: *White:* with a beacon located at the kitchen, trees within the atrium cause interference when line-of-sight is broken. *Black:* with a fluxgate compass, underground pipes cause interference. *Striped:* interference for both sensors.

Figure 4.5). In both cases, these cause the robot to take slightly longer to obtain a reading for its position.

Table 4.3 shows three potential room layouts where all rooms are in use: (1) one where rooms are assigned at random, (2) one where groups are housed closer to the kitchen to concentrate noise, and (3) one where rooms are allocated in a repeating pattern. We can compare the average travel times under no interference, interference imposed upon a compass, and interference imposed upon a beacon.

Over all three layouts, the robot with a beacon performs better than the robot with a compass. This result is easily explained by looking at Figure 4.5, where we see that the compass has 27 regions where underground pipes cause interference, as opposed to the 8 regions where the beacon receives interference. However, Table 4.3 also shows that despite this large difference, the actual impact of this interference is quite small: the robot equipped with the fluxgate compass spends on average just over $26\,\mathrm{s}$ longer delivering meals each hour. Considering that this difference would be

Table 4.3: Comparison of average time spent travelling each hour given an average of 10 meal orders per hour, three different guest layouts and robot speed of $5\,\mathrm{km/h}$. Values given are for the model under three interference models: no interference, interference to a beacon, and interference to a compass (Fig. 4.5).

	Average Travel Time (s)		
	No Interference	Compass	Beacon
Layout 1	2934	2970	2942
Layout 2	2997	3033	3006
Layout 3	2964	2999	2975

distributed over multiple orders in an hour, it becomes almost negligible when considered against other factors, such as the time it takes for hotel guests to take their food from the robot when it arrives.

4.4 Maintenance

Infrastructure can be costly to maintain. Worn-out carpet runners must be replaced, bridges repaired, footpaths repaved, and so on. Including maintenance costs into the equations given in Section 4.2 requires first that the reduced costs seen by the agents be matched to a timescale. We will revisit the example from Section 4.3.1 to demonstrate the general steps needed. The cost of carpet runners of the type described is dependent on thickness and width, but for this example we will assume they cost $\$10/\mathrm{m}$. Runners are replaced once a year to remove any that have become discolored or cracked. The facility requires just over $150\,\mathrm{m}^2$ of runners, leading to a yearly maintenance cost of $\$1500$.

Any savings the robots induce must also be estimated over a year, a highly problem-dependent calculation. We assume that the facility has three robots and, at a minimum, these robots make 48 trips per day, delivering medication three times a day to each resident. They likely carry laundry back and forth, and guide residents around but, for a pessimistic estimate of break-even point, we will ignore these uses.

Without the runners, the expected time of a delivery trip was $157.86\,\mathrm{s}$. With the runners, this decreased to $100.68\,\mathrm{s}$. This results in a daily average of $7577.28\,\mathrm{s}$ and $4832.64\,\mathrm{s}$, respectively. With the runners and this minimum amount of trips, the robots reduce their time spent traveling daily by just under $46\,\mathrm{min}$. To assign a monetary value to this, assume each robot's performance of these tasks frees up a nursing assistant to perform other duties. Therefore, valuing the work of the robots at the cost of employing a human for the same amount of time at $\$16/\mathrm{h}$ the "cost" of the robots becomes $\$33.68$ per day and $\$21.49$ per day. Given that care facilities operate every day, if the facility transitions from carpeted hallways to hallways with runners, the yearly maintenance cost will be recouped after 123 days.

4.5 Conclusion and Future Work

This work has aimed to bring environmental design, via infrastructure, into the realm of robot design problems. Because infrastructure takes a wide variety of forms, the model introduced here enables analysis of different types of infrastructure. The model treats infrastructure as an operator that transforms existing MDP models to reflect changes to behavior induced by altering the environment. Within the framework, we interpret the MDP's expected value not as a representative statistic for a single agent making decisions about an uncertain future, but as the cost over many independent repetitions. This allows us to understand the average behavior of a class of agent without direct large-scale simulation of all agents involved.

The model has room for refinement, which future work could explore. As noted, the model disregards interference between agents—this was a deliberate choice to allow tractable analysis of aggregate effects across many repetitions. In certain settings, contrary to the case studies examined herein, interference may be critical and models of how interactions manifest would be a useful addition.

Infrastructure itself can also be more complex, containing internal state or introducing new states into the world. The need to capture complex behavior and the impact of infrastructure as something that interferes with agents (via its transformation functions) suggests that infrastructure itself may be modeled as a type of agent.

Finally, while MDPs allow for us to easily examine the benefit from infrastructure in ideal cases, the use of Partially Observable Markov Decision Processes (POMDP) would allow for direct treatment of imperfect perception.

5. CONCISE TRACKING AND REPRESENTATION OF COMBINATORIAL SETS OF SENSORS

5.1 Introduction

Failure of equipment, devices, and technological contrivances is inevitable. Even if a robot is equipped with the most rugged sensors available at the time of its creation, the wear imposed by operating in the real world will eventually cause a deterioration in its performance, or even complete malfunction. Choosing to design robots with an emphasis on sturdy components has implications for mass, energy/fuel efficiency, and so forth — it represents but one way to achieve reliability. An alternative option, which is the superior option in certain settings, is to design with some notion of graceful degradation in mind so that the robot might continue to operate despite individual component failures. A useful step toward that would be to endow the system with an ability to estimate its own integrity, and to detect and self-diagnose faults (Khalastchi and Kalech, 2018).

This chapter considers the specific subproblem of sensor failure. Identifying these failures as they happen is useful in many cases: it could allow the robot to work around the failed sensors, or to make judgments on whether it can continue its current task with the current state of the system. Even if not used for decision-making in real time, summarizing sensor statuses can aid *ex post facto* incident investigation (e.g., the mandatory "black-box" flight-recorders in aviation (Federal Aviation Administration, 2007)).

The question becomes how to use the limited (and hard-to-predict) data to pinpoint the failed sensor when, of course, the information available may never uniquely determine a single defective sensor. And this need not be due to the inadequacy of information, but because multiple sensors may have simultaneously failed. This points to what we consider the key underlying challenge: the space of explanations is combinatorial, so trackers must consider multiple hypotheses, and manage them efficiently. Several prior approaches have tackled this problem in the probabilistic setting with Bayesian estimators, either as banks comprising multiple univariate models (e.g., Roumeliotis et al. (1998)) or other multi-hypothesis methods (e.g., particle filters as in Plagemann et al. (2006)). Such methods emphasize practicability. But handling distributional/likelihood information does not always succeed in clarifying the inherent representation of ambiguity, and may obscure rather than illuminate the underlying problem structure.

The present chapter tackles the combinatorial aspect of the hypothesis space directly, and shows where savings can be had without introducing other approximations or simplifications. To do this, and to make claims of exactness, we study the problem in a formal setting employing nondeterminism as a model of uncertainty rather than stochasticity. This formalism gives, for the smallest initial structures, enough complexity to express the underlying problem, and does so particularly crisply and, we feel, with elegance. We consider an example to examine the effectiveness of the modeling method presented here, and show a significant reduction in the size of the model when compared to naïve methods. The example also includes the impact of additional information on the size of the model, such as explicit knowledge that a sensor is failed or functioning.

We also consider how an agent can determine the severity of a failure's impact on its ability to complete a task. In many cases it may be possible for the agent to adapt its plan around the failure and still bring the task to a satisfactory close (e.g., the works presented in Lunze and Richter (2008)). However, the ability of an agent to recover is highly situation-dependent, involving not only determining if the use of other sensors is capable of compensating for those which have failed, but also: the usage of previously obtained information,[1] the potential for the agent to disambiguate states that may become conflated, and the ability to manage situations in which there may be multiple possible hypotheses for which sensor has failed. To accomplish this, we present an algorithm which analyzes a given p-graph to determine what sensing information is critical at different points for task achievement. We present as an example a situation in which the precise failure is ambiguous, a case which is handled with ease by the given algorithm. Additionally, we discuss the ability of this algorithm to provide additional information such as determining if the choice of a given plan results in an agent becoming particularly dependent on a sensor or set of sensors.

5.2 Problem Formulation

We consider time as progressing discretely, which permits us to define a graph structure labeled with events, and then to incorporate sensors within such a model.

5.2.1 Preliminaries

The model of the world that our robot inhabits (i.e., its environment), as well as the combinatorial filter that we will construct, will both be designed as p-graphs as defined in Definition 1 from Chapter 3. Although the definition of a p-graph does not require that E be finite, we will assume that it is throughout this chapter. Additionally, unlike in Chapter 3, we do not require all initial vertices to be observations, although they must be all of the same type (either $V_{\text{init}} \subseteq V_y$ or $V_{\text{init}} \subseteq V_u$). We use p-graphs as discrete transition systems, and make explicit here several ideas that have only been implicitly defined thus far:

Definition 34 (transitions to (Saberifar et al., 2019))**.** *For a given p-graph G and two states $v, w \in V(G)$, a finite event sequence $e_1 \cdots e_k$* transitions in G from v to w *if there exists a sequence of states $v_1, \ldots, v_{k+1}$, such that $v_1 = v$, $v_{k+1} = w$, and for each $i = 1 \ldots k$, there exists an edge $v_k \xrightarrow{E_k} v_{k+1}$ for which $e_k \in E_k$.*

The preceding is similar to the notion of tracing an execution as described in Chapter 3, with the difference being a focus on the event sequence itself over the vertices reached. We remind the reader that these event sequences incorporate a form of nondeterminism, as some may transition from a single vertex to potentially many vertices.

Definition 35 (validity and executions (Saberifar et al., 2019))**.** *Given p-graph G and state $v \in V(G)$, an event sequence $e_1 \cdots e_k$ is* valid from v *if there exists some $w \in V(G)$ for which $e_1 \cdots e_k$ transitions from v to w. An* execution *on a p-graph G is an event sequence valid from some start state in $V_{\text{init}}(G)$.*

[1] The use of prior information in this way is somewhat reminiscent of the vine graphs presented in Chapter 3, wherein success of a plan depended upon an agent's ability to recall specific prior events.

When no sequence of transitions can be found such that the sequence is valid, we will say that such a sequence *crashes*. As before, a p-graph W describes the environment. A robot operates in this environment so that, as it interacts, an event sequence $e_1 \cdots e_k$ is generated.

A final definition to be presented from previous work concerns sensor modifications, such as the behaviors that arise under failure. These modifications can be seen as a mapping from the current instance of W to a new one. Changes or defects then can be modeled by applying a function to observation labels. This approach for sensor mutation has also been employed by Saberifar et al. (2019) and Ghasemlou and O'Kane (2019).

Definition 36 (observation map (Saberifar et al., 2019))**.** *An* observation map *is a function* $\mathfrak{h} : Y \to 2^{Y'} \setminus \{\varnothing\}$ *mapping from an observation space* Y *to a non-empty set of observations in a different observation space* Y'*. For observation map* $\mathfrak{h}$*, its* extension to sets *is a function that applies the map to a set of labels:*

$$\mathfrak{h}(E) = \bigcup_{e \in E} \mathfrak{h}(e).$$

The extension to p-graphs *is a function that mutates p-graphs by replacing each edge label* E *with* $\mathfrak{h}(E)$*. We will write* $\mathfrak{h}[W]$ *for application of* $\mathfrak{h}$ *to p-graph* W*.*

5.2.2 Sensors and Information that Indicates Failure

We assume that a robot possesses a collection of sensors.

Definition 37 (sensor outputs)**.** *A robot equipped with* n *sensors operates in a world* W*, a p-graph which has observation set* $Y = Y_1 \times Y_2 \times \cdots \times Y_n$*. For each* $i \in [n] := \{1, \ldots, n\}$*, the set* Y_i *is called the valid sensor outputs for sensor* i*.*

Shortly, we will formalize operation when sensors may fail. In our model, the n individual sensors are atomic in the sense that they represent the level of granularity at which malfunction will be considered: zero, one, or more may fail, but we consider "sensor" to be the unit at which these phenomena are analyzed.

As some sensors degrade, they may still provide signals that could conceivably arise under some customary conditions of operation. The robot must then detect the error condition owing to inconsistencies, either in what is known from structure in the world, evidence from other sensors, or both. Such inconsistencies form one piece of information.

There can be further sources of information. When faulty, some sensors may provide explicit error messages or specific codes, or just fail to generate any data when they would produce some readings under standard operation. This provides a clear, unequivocal indication of a failure condition. We will use the symbol $\perp$, assumed to be distinct from all elements in Y_i, for all i, to describe the receipt of such an indication.

Symmetrically, in some cases there exist calibration steps, sequences for issue diagnosis, and/or elements in the environment (specially designed fixtures, rigs, health monitor systems) that can provide evidence of correct operation of a sensor. We use the symbol $\top$, also distinct from all elements in Y_i, to indicate interaction with some element that provides positive information for soundness of the sensor. (Thereafter, we will assume that the sensor remains operative for the remainder of the robot's execution, though modification of this treatment is straightforward.) Should the diagnosis uncover evidence of an anomaly, it can be modeled simply as providing a $\perp$, as described above. Thus, if such a diagnosis facility is possible (i.e., exists in the world), and it is definitive for sensor i, all edges departing it will be labeled either $\top$ or $\perp$ only. If the facility may be inconclusive, it would include other labels on outgoing edges.

It is worth emphasizing that these last two forms of additional information (encoded as $\perp$ and $\top$) are not required and may be safely omitted from the model that follows. If those forms of information are unavailable, they simply need not be present on any edges in the world. But when available, including these symbols enables use of this extra data to improve the performance of the tracker we will introduce. But the tracker we introduce is well-defined and correct without $\perp$, or $\top$, or with neither. For notational brevity and convenience, we will use the set $\overline{\overline{Y}}_i$ to represent sensor i's signal $\overline{\overline{Y}}_i := \overline{Y}_i \cup \{\top\}$ with $\overline{Y}_i := Y_i \cup \{\perp\}$, recognizing that the two additional certificate symbols may never arise.

Suppose some specific subset of the available sensors may give erroneous sensor readings. For any particular subset B (mnemonic: "broken"), the failure map $\mathfrak{F}_B$ is a specific type of observation map:

Definition 38 (failure map)**.** *For the observation set $Y = Y_1 \times \cdots \times Y_n$, and some set $B \subseteq [n]$, the* failure map *for B is the function:*

$$\begin{aligned}\mathfrak{F}_B \colon Y &\to 2^{\overline{\overline{Y}}_1 \times \overline{\overline{Y}}_2 \times \cdots \times \overline{\overline{Y}}_n} \backslash \{\varnothing\},\\ (y_1, \ldots, y_n) &\mapsto \left\{ (v_1, \ldots, v_n) \,\middle|\, \bigwedge_{i \in B} v_i \in \overline{Y}_i \wedge \bigwedge_{j \in [n] \backslash B} v_j \in \{y_j, \top\} \right\}.\end{aligned}$$

Intuitively, for those sensors in B, the function maps any output the sensor could produce to all possible outputs, including the explicit failure symbol, $\perp$, while those sensors not included in B may return either their original value y_j or the certificate of correct operation symbol, $\top$. (It is a simple exercise to see that, technically, the co-domain could be tighter.)

Property 1. Briefly, we note two extreme cases:

$$\begin{aligned}\mathfrak{F}_\varnothing \colon (y_1, y_2, \ldots, y_n) &\mapsto \{(y_1, y_2, \ldots, y_n)\},\\ \mathfrak{F}_{[n]} \colon (y_1, y_2, \ldots, y_n) &\mapsto \overline{Y}_1 \times \overline{Y}_2 \times \cdots \times \overline{Y}_n.\end{aligned}$$

If certificates are included in the model, the failure map $\mathfrak{F}_\varnothing$ is also capable of returning $\top$ for any observation y_j. Given a hypothesis B of a set of sensors suspected to be faulty, we can test the hypothesis through the application of the failure map for B to a p-graph. If all malfunctioning sensors are included within B, then a sequence that would have crashed on the original p-graph should no longer do so:

Definition 39 (consistent)**.** *For p-graph W and sensor set $B \subseteq [n]$, the set B is* consistent with sequence *$e_1 \cdots e_k$ if the sequence is a valid execution on $\mathfrak{F}_B[W]$.*

Note how an execution valid on the *original* W will not crash on any p-graph after the application of a failure map. Sequences from W itself never provide evidence for any sensor failures; it is only those that deviate which do. It can also be that no subset of $[n]$ is consistent: if W has a finite set of executions, it is possible for the sequence to be longer than any of them — this is an inconsistency that cannot be explained by any sensor failure, so even $\mathfrak{F}_{[n]}[W]$ will crash.

Furthermore, by its construction Definition 39 (built on Definitions 38 and 36) treats the $\perp$ and $\top$ symbols appropriately. For $\perp$, receiving such a symbol at slot i causes a crash if $i \notin B$. Conversely, receiving a $\top$ at slot i if $i \in B$ causes a crash, which is appropriate because it contradicts the claim that i is faulty. The types of information afforded by $\perp$ and $\top$ are not symmetric: a $\perp$ indicates immediately that there is a fault, whereas $\top$ attests to the fact that there is no fault now nor will there be for the remainder of the execution (these are our assumed semantics for a certificate of operability). This stronger assertion about future operation may be used in the interests of efficiency (as will will demonstrate in a later example), but for now it is sufficient to note that Definition 27 treats the information correctly.

5.2.3 Failure Collections, Posets, and Consistency

We are interested in sets of multiple B's to represent collections of consistent hypotheses. Given a sequence, we will process a collection of models *en masse* to eliminate precisely those which yield p-graphs that crash, such that those hypotheses which remain are the consistent subset. To help improve efficiency, we wish to exploit any structure that the collection affords. One example of this structure is that the relationship between the sets yields information about the relationships between their associated failure maps.

Property 2. Suppose $B_1 \subseteq B_2 \subseteq [n]$. If sequence $e_1 \cdots e_m$ is a valid execution on $\mathfrak{F}_{B_1}[W]$ then it is a valid execution on $\mathfrak{F}_{B_2}[W]$ too.

Proof. Under $\mathfrak{F}_{B_1}$, all indices $k \in B_2 \setminus B_1$ cause the y_k in $(y_1, \ldots, y_k, \ldots, y_n)$ to map to y_k (i.e., it has no effect on observations in these indices), while under $\mathfrak{F}_{B_2}$ all the elements in Y_k are included in the k^{th} position. In the sequence, $e_1 \cdots e_m$, consider any e_i to be an event that crosses an edge labeled by a set of observations. Since $\{y_k\} \subseteq Y_k$, whenever e_i crosses an edge under $\mathfrak{F}_{B_1}$, it can cross the same edge under $\mathfrak{F}_{B_2}$. $\square$

Thus, the more elements there are in B, the more accommodating the associated p-graph will be because the observation map are more permissive.

Suppose that we are given a particular collection of sets $\mathcal{B} = \{B_1, B_2, \ldots, B_m\} \subseteq 2^{[n]}$. We will call such a collection the *hypothesis space*. The hypothesis space forms a partially ordered set (poset) when ordered by set inclusion (i.e., $\subseteq$). Each element in such a collection will be termed a "model". Thus, the whole collection describes a set of models, and the task of filtering is to eliminate invalid models from such a collection.

When designating possible hypotheses, it is natural to use the powerset. However, permitting other collections turns out to be useful: for example, if we have knowledge indicating that at most ℓ sensors are potentially faulty, we might define a collection that contains all sets with ℓ or fewer elements, perhaps denoted $\mathcal{B}_{[0,\ell]}$. Thus, the degree to which the collection is smaller than the full powerset is indicative of prior information.[2]

5.2.4 Implicit Collections of Sets

Clearly collections like $\mathcal{B}$ can be large (exponential in n) if represented by mere enumeration. The filter we introduce shall avoid this, instead giving an implicit representation of consistent hypotheses. By this we mean:

Definition 40 (implicitly representing collections)**.** *A collection $\mathcal{C}$ of subsets of C is* implicitly represented *if we are provided with a characteristic function $\mathbf{1}_{\mathcal{C}}\colon 2^C \to \{0,1\}$ as a computational predicate that takes $\Theta(|C|)$ time.*

As an example, to represent the collection $\mathcal{B}_{[0,\ell]}$ containing all sets with only ℓ or fewer elements implicitly, an implementation of the function $\mathbf{1}_{\mathcal{B}_{[0,\ell]}}$ would determine the cardinality of its input and return 0 immediately if it exceeded ℓ.

5.2.5 The Diagnosis Problem

We are now ready to formally state this chapter's central problem:

DIAGNOSIS PROBLEM: Given a p-graph W, some collection $\mathcal{B} = \{B_1, B_2, \ldots, B_m\}$, and a

[2]While the powerset is a lattice, this example shows that there are practical cases where the collection has weaker structure than a lattice. In the case of sets with between k and ℓ faults, $\mathcal{B}_{[k,\ell]}$, the hypothesis space does not form even a semilattice.

finite event sequence $e_1 \cdots e_k$, return the consistent subcollection:

$$\mathcal{B}\big|_{e_1 \cdots e_k} := \{B_r \in \mathcal{B} \mid e_1 \cdots e_k \text{ is an execution on } \mathfrak{F}_{B_r}[W]\} .$$

DESIDERATA:

<u>Cumulativeness</u>: Compute the subcollection incrementally, that is use $\mathcal{B}\big|_{e_1 \cdots e_{k-1}}$ to compute $\mathcal{B}\big|_{e_1 \cdots e_k}$.

<u>Concision</u>: Express the resulting collection implicitly.

We will now present an algorithm for this problem.

5.3 The Tracker

For any given model, the set of sensors within it can be applied as a failure map to the original world. Each observation that a robot receives has the potential to change the list of failure map models which are consistent with the execution history. One might follow the definition directly and simulate all possible models at once; those that are inconsistent will crash during the execution, while consistent models will not. However, if one can track only the models which are currently relevant and generate the full set later, the time spent checking models for consistency can be greatly reduced over the direct, or naïve, implementation. This section presents an incremental algorithm that gives a concise method of tracking these relevant models.

Our initial model is generated from the following:[3]

- the p-graph W describing the environment,
- a vertex v,
- a *working set* S,
- a hypothesis B, and
- a failure map $\mathfrak{F}_B$.

For this vertex, we must determine if the incoming observation is consistent under the current model. After application of the failure map $\mathfrak{F}_B$ to the outgoing edges, the incoming observation is

[3]In the case where $\mathcal{B}$ has multiple minimal elements due to restrictions on the collection, we must generate an initial set containing a model for each minimal element to ensure full coverage of $\mathcal{B}$.

compared to the modified labels. If none of the edges are consistent (i.e., the incoming observation matches none of the updated edges under the current hypothesis), a crash will occur. When a crash occurs, the model is updated by adding additional sensors to B until the incoming observation is consistent for at least one outgoing edge. The updated set B' becomes the current hypothesis, and the target vertices of any outgoing edges that are consistent under B' each become a new model under consideration, corresponding to the possible states of the robot after this observation. Additionally, there may be multiple sets B' that result in consistent models; we desire the set of hypotheses that are not subsets of (or equal to) each other. In this case of multiple consistent hypotheses, new models are created for each combination of target vertex and hypothesis. If the incoming value is an action instead of an observation, then models which are inconsistent are removed from consideration. This is because the labels with actions are unaffected by $\mathfrak{F}_B$, and so inconsistency indicates that the model is overall impossible.

Each vertex is treated independently from the others, and is contained within an individual model. Each model can be interpreted as a different "reality" that is consistent with what has been observed thus far; this forms the basis for our algorithm, which maintains these different potential models to determine which are consistent overall with the given execution.

We also introduce a new concept within the model, which we call the *working set* (denoted S).[4] The working set is used to develop updated models after a crash. S contains the set of sensors that are still under consideration to be included within the model, and excludes sensors that are already within the model as well as those that have been confirmed as failed or functional (with $\perp$ or $\top$). If a model crashes, the algorithm checks if any single element from the working set can be added to correct the model. If that fails, it tries progressively larger subsets until it finds one that corrects the model to be consistent. If we consider the original lattice of all hypotheses $\mathcal{B}$, (such as that in Figure 5.1), this is equivalent to moving "up" a level within the lattice to find new hypotheses, and then verifying them. Such a move also "raises" the lower bound for what additional models can be derived going forward. This move up results in the disqualification of the previous model. Algorithm 3 presents our method for tracking and implementing these models, as

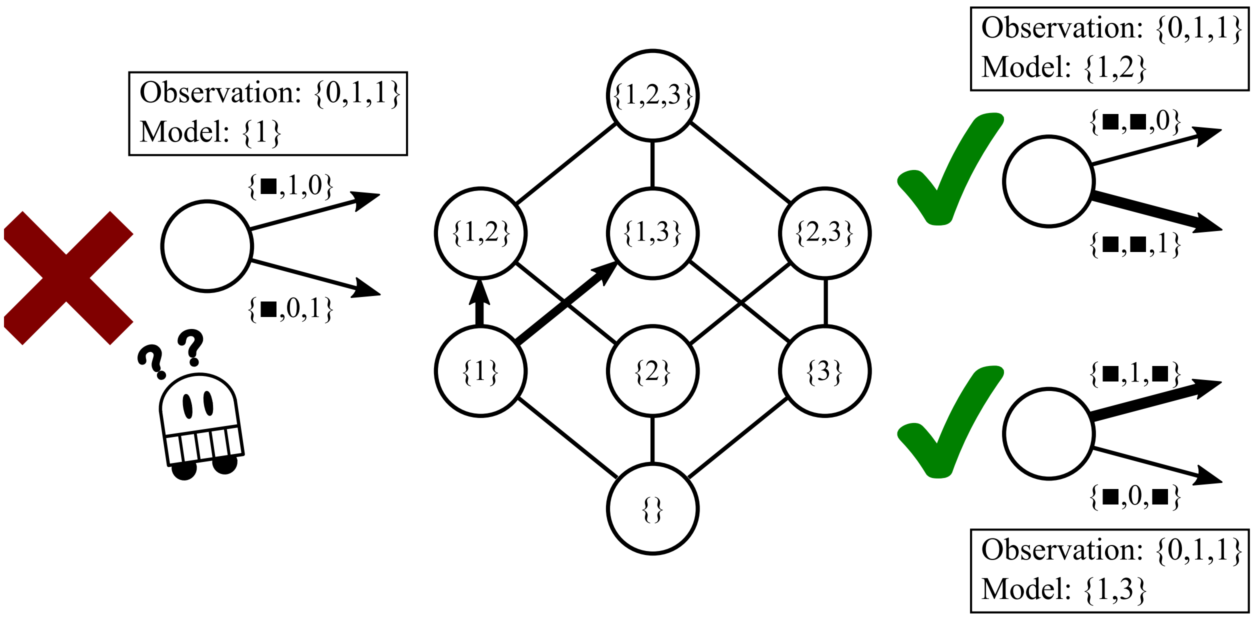


Figure 5.1: An illustrative cartoon of the model update process. Indices for observations included in the failure map have been replaced with black squares. We see that the robot, given an inconsistent observation, "moves up" the lattice to produce two consistent hypotheses.

well as a method for "raising" each model's boundary to enforce consistency.

We will now describe the use of Algorithm 3 on an execution e, which may be a full string or only a single event. By using the initial vertices of the world V_{init} and an initial empty hypothesis set B to generate a starting family of models, we may represent the situation in which the robot has just begun execution in any of its possible starting states, and in which it knows nothing about the current status of its sensors. (If S contains information that disqualifies the empty set, the initial hypotheses for these models are the minimal elements of $\mathcal{B}$ permitted by S.)

If the next event in e is an action, the outgoing edges of v are examined to see if that action is valid on any of them. If so, the model is updated with the target of that edge. In the case of multiple edges with the same action label, new models are created with the same hypothesis B for each additional vertex target. If the action is not valid on any outgoing edge, this model is inconsistent and removed from consideration.

[4]We use $\mathbf{S}$ to indicate that this information may be given in multiple ways, and is not necessarily equivalent to the hypothesis space $\mathcal{B}$. While $\mathbf{S}$ could directly give $\mathcal{B}$, it could also be individual sensors or sets. We also note that in cases where $\mathcal{B}$ does not contain the full powerset of sensors, additional overhead is required during the generation of new bounds to ensure that the algorithm does not generate new hypotheses that are not members of $\mathcal{B}$.

Algorithm 3: Verifying and Updating a Model

```
Data: a model $M = (W, v, S, B, \mathfrak{F}_B)$, event $e$
Result: a set of models $\mathcal{M}$
1  Function CHECK_CONSISTENCY(vertex $v$, event $e$, failure map $\mathfrak{F}_B$):
2      $e'$ = APPLY_FAILMAP($e$, $\mathfrak{F}_B$)
3      target_set = {}
4      for each outgoing edge $o\_edge$ of $v$ do
5          updated_edge = APPLY_FAILMAP($o\_edge$, $\mathfrak{F}_B$)
6          if $e' =$ updated_edge.label then
7              $v'$ = updated_edge.target
8              add $v'$ to target_set;
9      if target_set is not empty then
10         return True, target_set
11     else
12         return False, {}

13 $\mathcal{M} = \{\}$
14 if $e$ is an observation then
15     is_consistent, target_vertices = CHECK_CONSISTENCY($v$, $e$, $\mathfrak{F}_B$)
16     if is_consistent then
17         for target $v'$ in target_vertices do
18             add new model $M' = (W, v', S, B, \mathfrak{F}_B)$ to $\mathcal{M}$
19     else
           // current model is not consistent
20         for $s = [1 \dots \text{size}(S)]$ do
21             test_sets = all subsets of size $s \in S$
22             for proposed set $B' \in$ test_sets do
23                 is_consistent, target_vertices = CHECK_CONSISTENCY($v$, $e$, $\mathfrak{F}_{B'}$)
24                 if is_consistent then
25                     for target $v'$ in target_vertices do
26                         add new model $M' = (W, v', S, B, \mathfrak{F}_B)$ to $\mathcal{M}$
27             if $\mathcal{M}$ is no longer empty then
                   // we have found all of the smallest possible changes to $B$ to achieve consistency
28                 return $\mathcal{M}$
29             else
30                 continue
31 else
       // $e$ is an action
32     for each outgoing edge of $v$ with label $e$ do
33         $v'$ = edge.target
34         add new model $M' = (W, v', S, B, \mathfrak{F}_B)$ to $\mathcal{M}$
35     return $\mathcal{M}$
```

If the next event in e is an incoming observation y, each model currently under consideration compares y against itself according to the process described previously. Instead of producing all possible labels that could result under a given failure map (as specified by Definition 38), implementation equivalently permits the algorithm to disregard values from sensors deemed "failed" by the current hypothesis. If this map is insufficient, the hypothesis is updated, and the resulting list of consistent hypotheses and vertices is returned.

Although each model keeps only a single vertex, the algorithm's extension over all outgoing edges from that vertex means that any nondeterminism within the world's structure is preserved through the creation of multiple new models. At the end of the execution's processing, the set of models, along with the working set for each, can be used to generate the full set of models consistent with the execution without having tracked them all simultaneously. We first make this notion of "generation" precise:

Definition 41 (generated). *Given two models t_a, t_b, we can say that t_b is* generated *from t_a if:*

1) $B_a \subset B_b$: t_b's hypothesis B_b includes all sensors in B_a plus at least one additional sensor;

2) $B_b \cup S_b \subseteq B_a \cup S_a$: t_b's hypothesis B_b and working set S_b do not include any sensors that are not either already included in B_a or are in S_a.

Lemma 3. *The set of consistent models generated from the union of multiple models' lower bounds is equivalent to the set of models obtained through simulation of all possible failure maps $\mathcal{B}$.*

Proof. Given a set of sensors $\{\overline{\overline{Y}}_1, \overline{\overline{Y}}_2, \ldots, \overline{\overline{Y}}_n\}$, consider the full lattice and resulting hypothesis space $\mathcal{B}$ generated by its powerset. Some of these models are consistent under a given execution, for which there exists at least one execution that is valid on the world after application of the failure map. Given an execution e, we can obtain the full set of consistent models by generating the failure map for each element within $\mathcal{B}$, and then applying each failure map individually to W. If the execution e is valid on the modified world, that particular failure map corresponds to a consistent model, and repeating this for all elements of $\mathcal{B}$ yields the full set of consistent models. We will indicate the full set of consistent models with $T = \{t_1, t_2, \ldots, t_k\}$. We claim that Algorithm 3 is capable of generating the same set.

Property 1 of Definition 41 ensures that the generated model's hypothesis B_b is higher in the lattice $\mathcal{B}$ than B_a, whereas Property 2 ensures that B_b can be constructed from B_a through the addition of elements from S_a.

We will refer to the set of models returned by Algorithm 3 and those that may be generated from them as the set T'. In order for there to exist a valid model $t_i \in T$ that is not also included in T', one of the following conditions must hold:

1) t_i's hypothesis B_i contains a sensor that is not included by any model $t'_j \in T'$, either in its hypothesis B_j or working set S_j, or
2) B_i is a smaller consistent hypothesis than any in T', where $B_i \subset B_j$ for any $t_j \in T'$.

We will address these in turn. For the first condition, we examine the function of Algorithm 3. When a model B_j crashes on an execution e, the working set is used to find new candidate models. This begins with taking all subsets of the working set S_j with cardinality 1. Each of these subsets is individually examined to see if adding it to the model B_j produces a consistent model. Sensors are only removed from the working set if they become part of the model or if they are confirmed functional with $\top$. If the initial model (or set of models) includes all sensors within its working set and begins with the minimal element of $\mathcal{B}$, the only way in which a sensor could not be included within the hypothesis B_j or working set S_j is if it was verified with $\top$, contradicting the claim that t_i's hypothesis B_i is consistent with the current execution e.

Similarly, for the second condition we examine the way in which the algorithm builds each model's hypothesis. Given an initial model (or set of models), sensors are added to a hypothesis only when the execution e crashes on it; new models are found by adding elements from S to the hypothesis set individually. Therefore, any subset of a hypothesis B_j would have been previously considered as a model, and discarded only due to e crashing on it, contradicting the claim that it is a consistent model.

□

To make the intuition behind this algorithm precise, we will present a simple example with three sensors. (This is not the lattice for the case study described below.) Figure 5.2 illustrates the difference between the direct approach and that of the tracker we have presented. Within the hypothesis space $\mathcal{B}$, there is a subset consisting of hypotheses that generate consistent models. As a robot executes a plan, hypotheses are eliminated and the lower bound of this subset rises in the lattice.

We track the lattice through various stages of an execution: at the beginning when no observations have been received, there exist no inconsistencies. The direct approach therefore simulates all members of the lattice as models. Our method simulates the minimal member of the lattice, which is the empty set.

At the next step of execution, the robot receives an observation that is inconsistent with the assumption that all sensors are operable. For this observation, the models wherein either sensor 1 or sensor 2 have malfunctioned make the observation consistent. The direct approach removes from consideration the inconsistent models. Our approach results in the generation of two new models, and the discarding of the initial hypothesis. At the third step, another observation results in an inconsistency that is resolved if sensor 3 has malfunctioned, further reducing the set and again raising the lower bound of the lattice.

At the final step, we assume that our robot receives a certificate that verifies sensor 1 is operable. This allows for all elements containing sensor 1 to be removed from consideration. This not only modifies the lower bound, but the maximal element is *also* removed from consideration, creating a new upper bound on the subset. (The effect of receiving $\top$ during execution is discussed in Subsection 5.5.2.1.) The remaining subset of $\mathcal{B}$ that produce consistent models are our possible explanations—in this case, the single explanation $\{2, 3\}$.

Shortly, we will present several examples. Section 5.4 contains two introductory examples that demonstrate how the lattice allows for us to capture the existence of multiple consistent explanations. Section 5.5 provides a more complicated example demonstrating the use of this algorithm with an execution history to track multiple hypotheses. Before doing so, however, we briefly di-

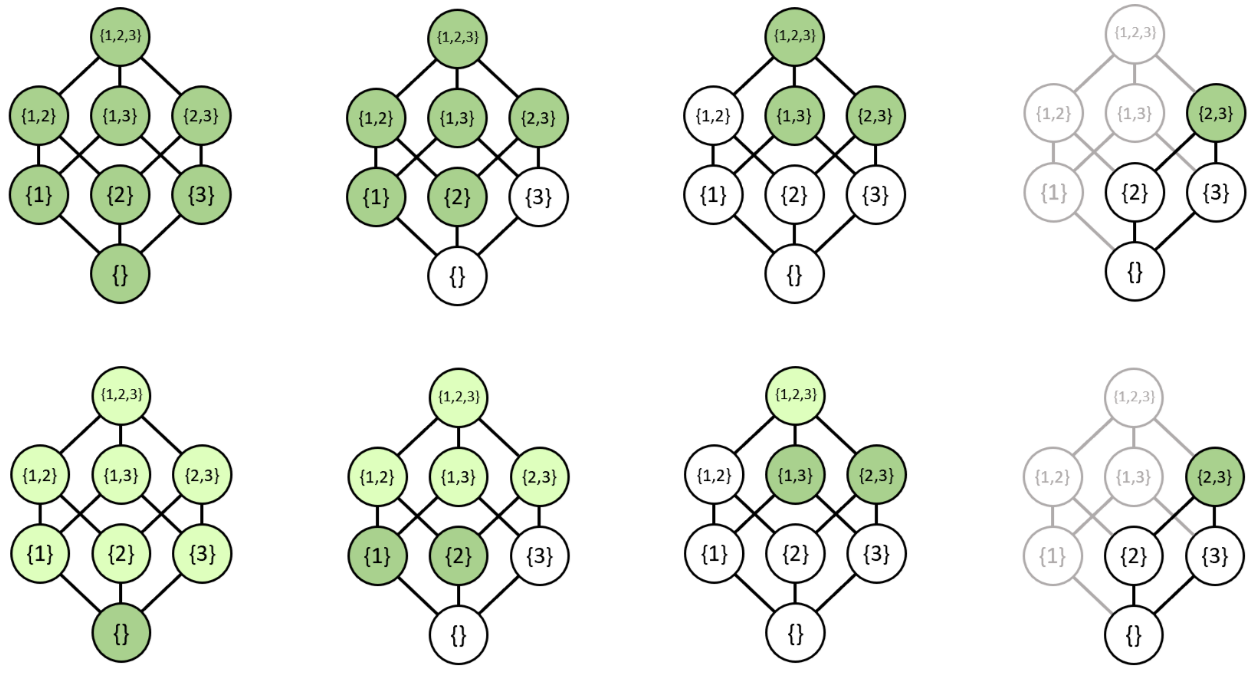


Figure 5.2: Comparison of two methods for sensor failure identification. Top Row: A direct approach that begins by simulating all possible failure maps, and which eliminates them as inconsistencies arise. Bottom Row: The algorithm presented in Alg. 3, which maintains individual models for each string. Dark green circles are models currently maintained by each method, while lighter colored circles are models that can be generated from the set.

gress to discuss the important matter of how large the set of models to track can become.

5.3.1 A Note on the Number of Models

There are two factors that influence the number of models maintained by Algorithm 3 at any point during execution. The first of these is the number of children that can be reached from any given vertex. When an observation is received, a single model will become a set of models, the size of which is equal to the outdegree of the vertex under consideration, as outgoing edges can always be made consistent by increasing the hypothesis set.[5] When considering actions, the number of children is dependent on the number of outgoing edges with the correct label.

This value can vary widely based on the problem being considered. One way in which this can be managed is through the design of additional overhead such that each model contains multiple vertices (and associated working sets), as opposed to containing a single vertex and working set.

[5]The exception to this is when certain sensors have been verified with the certificate $\top$, in which case the number of models created will be equal to the number of outgoing edges that have consistent labels for all certified sensors.

Vertices would then be passed between models, and models would be disqualified and removed when they had no more vertices. This then reduces the number of models to maintain at the cost of additional information to store within each model.

The second factor that influences the number of models is the size of the largest antichain for the lattice. An antichain for a poset is a collection of subsets for which no member within the collection is comparable to another (or, in the case of this dissertation, no subset in the collection is a subset or superset of another member of the collection). Assuming that we have a single model for each hypothesis through the method discussed above, maximum number of models that Algorithm 3 could have to maintain at any given point would be equivalent to the size of the largest possible antichain in the lattice. A long-standing result of Sperner (1928) (as quoted in the proof by Lubell (1966)) shows that this value is $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ for the poset created by a set of size n.

This could potentially be a very large set of models. However, assume a case in which our robot has no prior information and the full lattice $\mathcal{B}$ is under consideration. The tracker will start with an initial model for the empty set, and generate additional models higher up the lattice as necessary. This means that the growing number of models can be tracked over time, and action taken if the size of consistent models begins to exceed a desired size. Additionally, for each $\top$ or $\bot$ the robot receives, the number of possible consistent models is reduced by half. Therefore, while it is possible for Algorithm 3 to be required to maintain $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ models during execution, the growth of the lower bound during execution can be used to candidates for testing or calibration to manage the size of the antichain.

5.4 Example: The Two Hallways

We present Figure 5.3 as an introductory example of the failure identification problem. A robot is in a hallway that splits into two, and will nondeterministically choose one of the hallways to travel down. This robot is equipped with two types of sensors: one that is capable of detecting three different shapes, and one that detects one of three colors. The shapes may be a square (□), triangle (△), or circle (◯). The colors can be red, yellow, or blue. Whenever the robot makes an observation, it receives both a shape and a color.

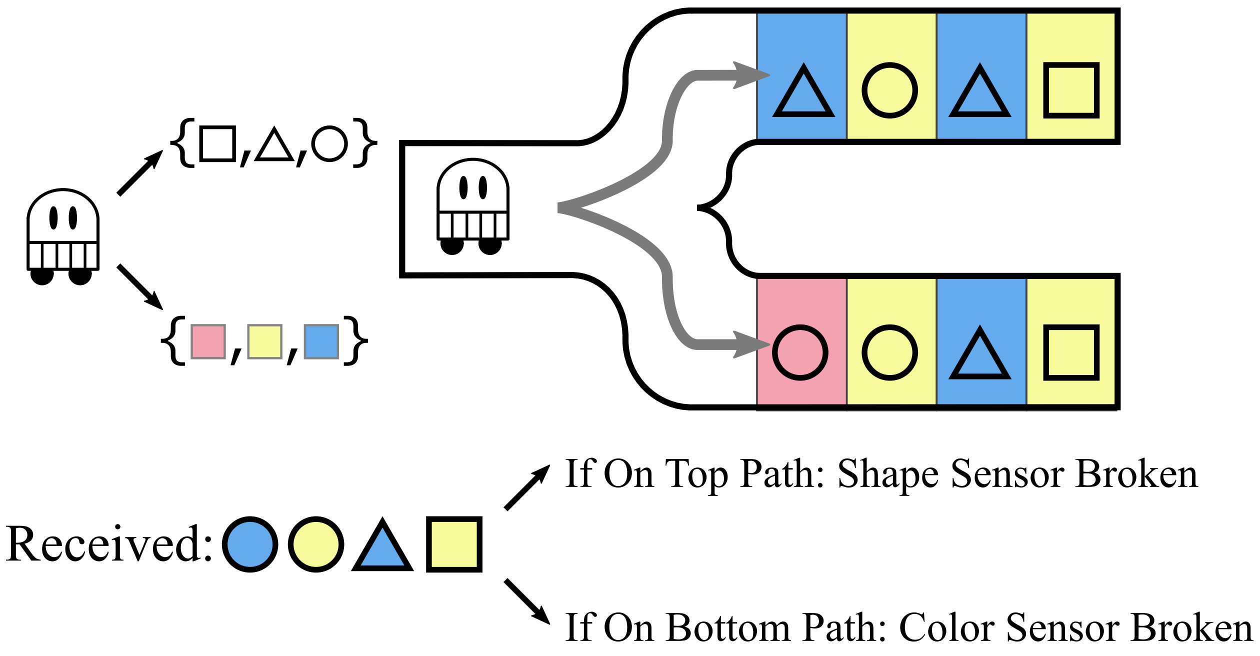

{□,△,○}
{■,■,■}
Received:
If On Top Path: Shape Sensor Broken
If On Bottom Path: Color Sensor Broken

Figure 5.3: An example in which a robot nondeterministically moves to one of two hallways. Given a sequence of observations that are inconsistent with either hallway, we cannot determine which sensor has malfunctioned unless we know which hallway the robot is in.

Assume that the robot receives the sequence of a blue circle, yellow circle, blue triangle, and yellow square (●,●, ▲ ,■). This sequence is not consistent with either of the hallways; for the top path, we would expect a triangle instead of a circle for the first observation, and for the bottom path we would expect the first observation to be red instead of blue. If the robot knew which of the hallways it had ended up in, identifying which sensor was malfunctioning would be simple. However, due to the fact that we are unaware which of the two hallways the robot has nondeterministically chosen, we cannot determine which of the sensors has malfunctioned.

Figure 5.4 introduces a slightly more complex version of the same situation. In this version the robot now has a third sensor, which we will call a line sensor, which can identify if a shape's outline is a solid line or a dashed line. Figure 5.4 also features a slightly different sequence of shapes and colors than Figure 5.3. In this example, the robot receives a sequence of a blue triangle with a dashed line, a yellow circle with a dashed line, a blue circle with a solid line, and a yellow square with a solid line (▲ , ●, ●, ■).

As the robot in this example has three sensors, the hypothesis space $\mathcal{B}$ contains nine sets, which

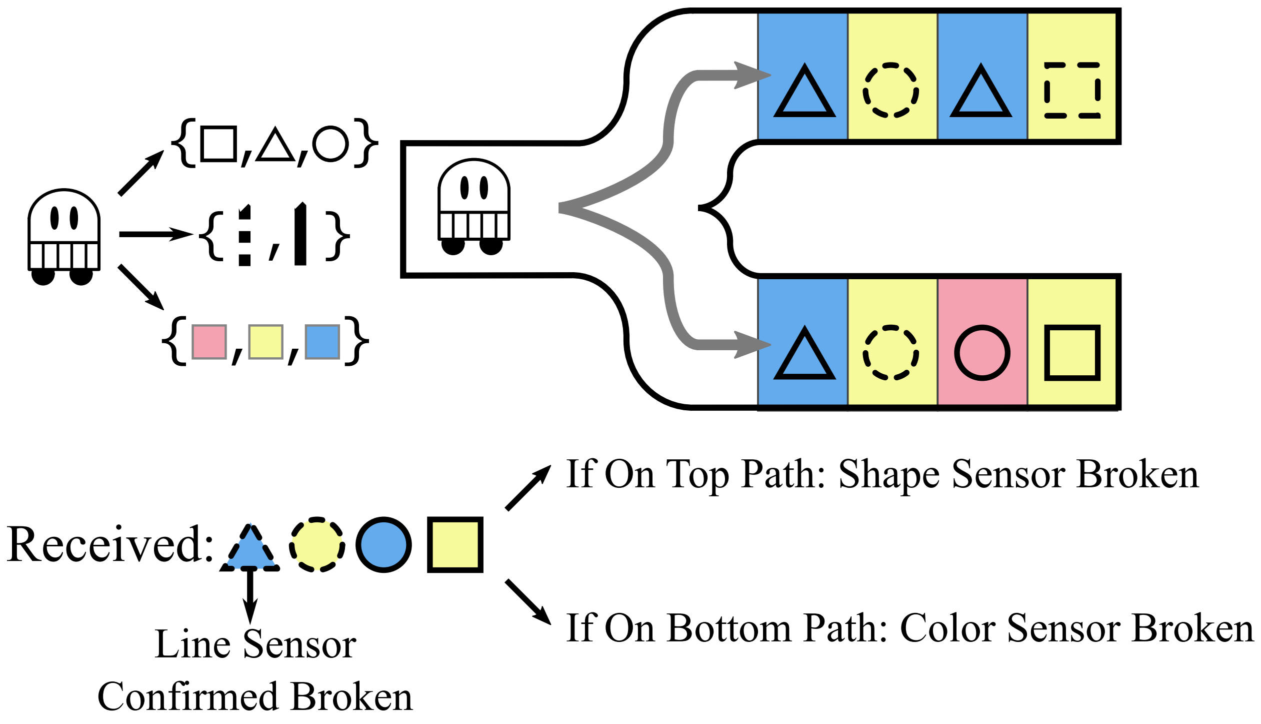

If On Top Path: Shape Sensor Broken
Received:
If On Bottom Path: Color Sensor Broken
Line Sensor
Confirmed Broken

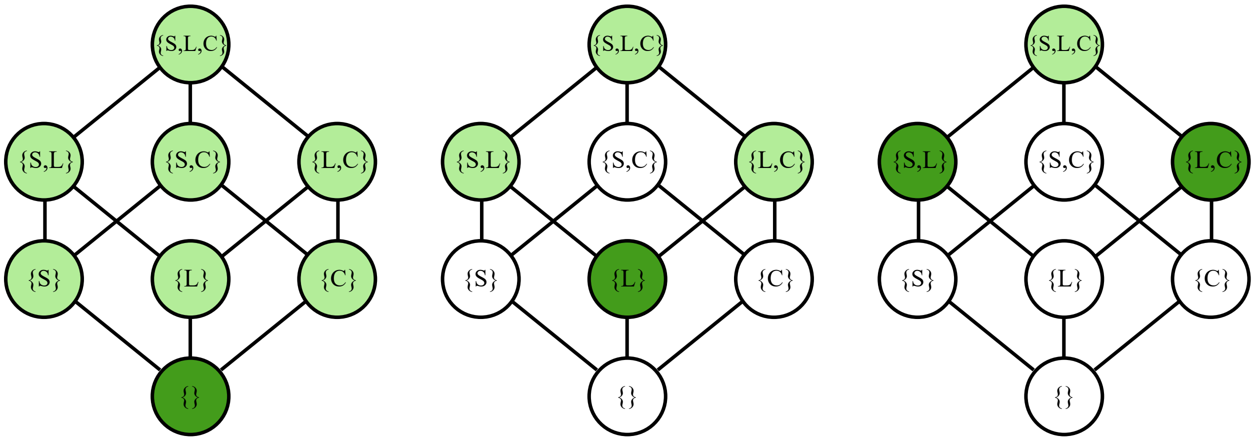
Figure 5.4: An example in which a robot randomly moves to one of two hallways, now with an additional sensor. With this sequence of observations, we can determine that the line sensor has malfunctioned, but still cannot determine which of the other sensors have malfunctioned.

we can arrange in a lattice as in Figure 5.5. For this lattice, we abbreviate the names of the sensors: **S** for **Shape**, **C** for **Color**, and **L** for **Line**. Figure 5.5a shows the lattice at the start of execution, before any observations have been received. As no observations have yet been received, there is nothing to cause inconsistency, and all elements of the lattice are a possible consistent hypothesis. The lower bound of the lattice is the empty set, $\{\}$, which is marked in dark green in the figure.

Upon receipt of the first observation (the blue dashed triangle, ▲), it can be immediately determined that the line sensor has malfunctioned. No matter what hallway the robot is in we expect the robot to sense a solid line, and thus this is inconsistent. Figure 5.5b shows the lattice after it has been updated to reflect this information. As the line sensor is known to have malfunctioned, all consistent models must now contain it within their set. This results in the lower bound of the lattice becoming $\{\text{L}\}$ and the elimination of all models that do not contain the line sensor.

Later, when the third observation is received, we experience the same issue as in the example provided in Figure 5.3. Without knowledge of which hallway the robot is in, there is no way to determine whether the shape sensor or color sensor has malfunctioned. Figure 5.5c reflects this; as it can be determined that another sensor has malfunctioned in addition to the line sensor, $\{L\}$ is no longer consistent. Instead, the lower bound of the lattice becomes the two sets $\{S,L\}$ and $\{L,C\}$, as either hypothesis is consistent.

These two examples are meant to highlight several ideas. First, it is clear even for these small examples that the size of the hypothesis set $\mathcal{B}$ can quickly grow unmanageable. Second, Figure 5.5 demonstrates how Algorithm 3 uses received information to gradually raise the lower bound of the lattice, allowing for the disqualification of hypotheses without directly testing them. Finally, the lattice at the end of the observation sequence has a lower bound that consists of sets of multiple sensors. This demonstrates an idea first stated in Section 5.1, which is that using probabilistic models to identify the failure of individual sensors does not capture information that is useful to us (in this case, that all consistent explanations involve multiple malfunctioning sensors). Even in the case of an estimator tracking probabilities over sets of sensors, we argue that the lattice-based approach described here provides a concise way of tracking information in a way that can be directly applied to questions of task completion and planning (Section 5.6).



(a) The lattice before any observations. **(b)** The lattice after the first observation. **(c)** The lattice after the third observation.

Figure 5.5: An example of how the sequence of observations in Figure 5.4 affects the lower bound of the hypothesis space $\mathcal{B}$. **S** is the **Shape** sensor, **C** is the **Color** sensor, and **L** is the **Line** sensor. Circles that are dark green indicates the lower bound of the lattice, which are the models that would be kept by Algorithm 3. Light green indicates consistent models that could be generated, and white indicates hypotheses that have been removed from consideration.

5.5 Example: Digging for Gold

A robot is searching for buried gold within a confined space. The robot operates within a 5×5 grid world. The robot's goal is to locate and dig up the buried gold, which could be in any of the 25 locations in the world. The agent is capable of moving in the four cardinal directions and digging in the square it currently occupies. The agent has been equipped with a small selection of sensors to help it navigate this space. These sensors provide the following information:

1. The *distance* of the nearest wall.

2. The *direction* of the nearest wall.

3. An intensity reading (from the set {none, low, med, high}) indicating *how close* the goal object is.

4. *Verification* of goal attainment once an object has been dug up (such as ensuring the right object has been found using a camera).

The robot does not know ahead of time which of the 25 possible locations contains the treasure it seeks. We represent this uncertainty through a collection of 25 worlds, each differing only in

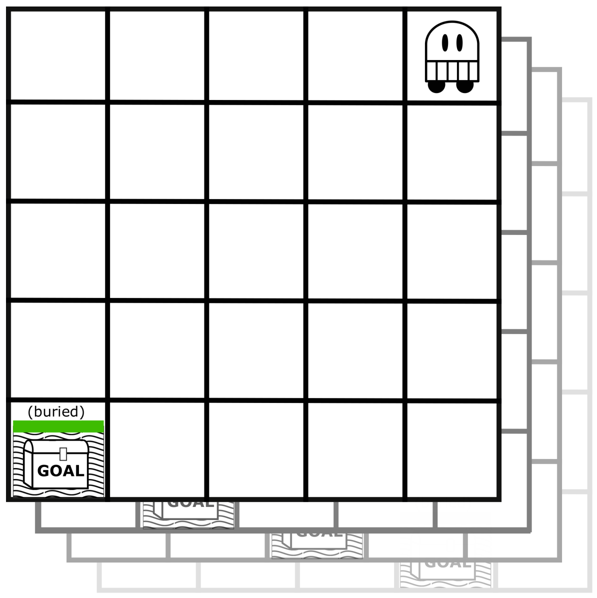


Figure 5.6: A collection of 5×5 worlds, each of which places the buried goal in a different square.

where the treasure has been buried. Observation values are also adjusted accordingly. Figure 5.6 shows a representation of this collection.

In order to analyze our algorithm, we will present the example of an execution in which a sensor failure occurs. We will compare the complexity of the algorithm throughout its execution to an approach that directly implements Definition 39, but does not make use of the insight of Property 2. The direct approach simulates all possible models of sensor failure simultaneously, and stops simulating a model only when no more strings are consistent with it. The same vertices are therefore considered independently for each model.

The number of models maintained by the algorithm depends not only on the execution of the robot, but also on the number of observations that may be valid from a location under some configuration of the world. Within this example, the main complicating factor is that any observation from the intensity sensor is valid for some configuration of the world. Numerous models which are consistent under the assumption that this sensor has failed will therefore propagate during execution, as they cannot be completely ruled out.

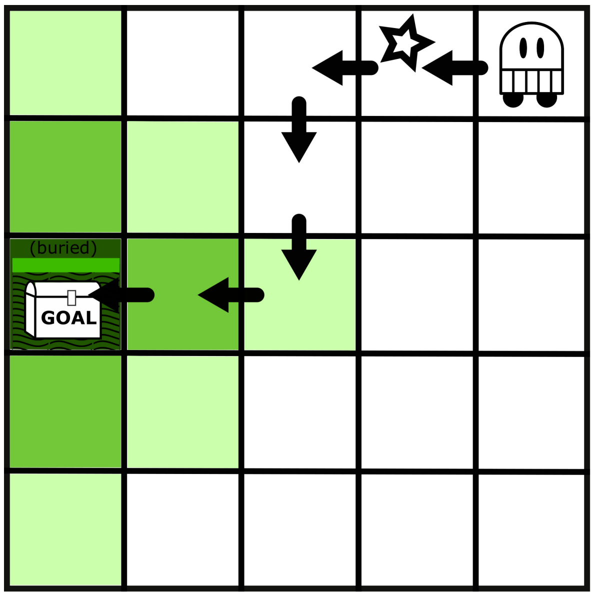


Figure 5.7: An example of a robot's path through a given world. The star indicates a sensor malfunction during execution, while the gradient is a visual representation of the proximity sensor's readings.

5.5.1 Running the Algorithm

Figure 5.7 shows an example of a path taken by a robot. We will use this execution history as the input to the algorithm. During this execution, the sensor that reports the distance to the nearest wall malfunctions at a certain step. This failure is not explicit; the agent receives an incorrect reading instead of the failure indicator $\perp$. If the robot knows its initial location in the world, this observation can be determined to be inconsistent with the world. For both the direct method and Algorithm 3, this eliminates all models in which the wall sensor has not failed.

At the start of execution, the direct method must have a model for every one of the possible 25 world configurations, along with all of the possible combinations of sensor failures. With four sensors, this method begins with 400 models to manage. The method from Algorithm 3 begins with only 25; one for each world configuration, each of which has an initial hypothesis of the empty set.

When the incorrect sensor reading is received, the robot is capable of determining that this observation is inconsistent with the world. For the direct method, this reduces the number of models to 200. Our algorithm will update its models to add the wall distance sensor to each, raising the lower bound of the lattice of the hypothesis space. For the particular execution shown in Figure 5.7, our algorithm will never have more or less than 25 models. This is due to the fact that the robot does not take any unsafe actions that would disqualify a model in that way; additionally, under the assumption that all sensors are faulty, all models would become consistent with the observation history.

This constant number is due in part to the structure of the problem: for each of the 25 possible goal locations, the corresponding world has single observations that are expected at each location. In cases in which a robot can receive multiple possible observations and transition to different vertices, the number of models will increase. When certain actions are not valid under certain models, the number will decrease.

Our main improvement over the direct method is that the number of traces tracked by our algorithm depends on the average number of outgoing edges from any given observation vertex. Conversely, the direct method must track branching in the plan for each possible failure model. Depending on the information available to the robot and the sensing history, any number of elements within the lattice may be consistent models at a given point in execution. Given that the size of the hypothesis space can be exponential in relation to the number of sensors, the overhead required to maintain all models simultaneously for large numbers of sensors quickly becomes unmanageable.

5.5.1.1 Variation on a Theme: Advanced Knowledge

We consider a variant of this problem where the robot has been given knowledge about the hypothesis space $\mathcal{B}$ *a priori*. In particular, we assume that the robot has been informed that its intensity sensor is functioning. This reduces $\mathcal{B}$ down to eight elements (for the remaining three sensors and their combinations). At the start of execution, the algorithm will create 25 initial models as before. Once the first observations are received, however, both methods will reduce the number of models tracked. As the intensity sensor is known to be working, the initial reading eliminates any worlds where the buried treasure is at the starting location, as well as any where the treasure is assumed to be within two spaces of the agent. When the models for these potential worlds are evaluated, they will crash. As the intensity sensor is not part of the hypothesis space there is no way to make these models consistent, and so they are removed from consideration. By the end of execution, our algorithm is maintaining only a single model — that which includes the wall distance sensor in its set, and which corresponds to the world with the goal at the location seen in Figure 5.7.

5.5.1.2 Variation on a Theme: Nondeterminism

The previous examples have both had the algorithm initialize with 25 models, which then may or may not be reduced. However, it is possible for Algorithm 3 to increase the number of models it is tracking. This can happen in two different ways. In the first way, the observation may not result in adding only a single model to the hypothesis space. Algorithm 3 generates a collection of sets of the same size to test, each of which is an incomparable element above the current hypothesis in the lattice. If multiple of these sets can make the observation consistent, then the algorithm will create a new model for each of them. The second way in which the algorithm can increase the number of models under consideration is through non-determinism in action outcomes. If an agent can take an action that labels multiple outgoing edges, then we must consider that any of the target vertices could be the one to which our agent transitions. As each individual model only tracks a single vertex, the algorithm must create a new model for each target. This means that exact number of models that may be under consideration at any given time is dependent on the problem structure, particularly on the amount of non-deterministic transitions and any observations that may later reduce this set.

5.5.2 Use of Information

Additional information, such as that provided by the $\perp$ and $\top$ symbol, can be valuable for eliminating large portions of the hypothesis space.[6]

[6] The following subsections are written with consideration to the initial version of the example, and not its variations.

5.5.2.1 Receiving Information on Failures and Certifications

As shown in the previous example, the design of a world (and the knowledge implicit in the representation of that world) means that even in the absence of explicit failure symbols such as $\perp$, some malfunctions can still be identified. However, there may also be sensors for which any reading *could* be a valid observation in certain cases. Knowledge of explicit failure allows for reduction of the model set in this case. Sensor testing efforts can then be guided by knowledge of which sensors may have failures that are difficult to identify.

In addition to the working set's ability to reduce the number of elements in $\mathcal{B}$, it is also possible for the robot to acquire information during the course of its execution, such as that which might be obtained with self-diagnostics. Another source of information is more abstract information about the problem; for example, we would never expect the camera to return that it has identified the treasure before the robot has even dug anything up. Nor would we expect certain combinations of wall distance and direction observations, as we know that the world is a square grid.

Unlike its counterpart $\perp$, the $\top$ symbol is unique in that it must be part of the defined world model itself. Otherwise, obtaining $\top$ in a sensor reading would result in a crash during execution, contradicting the certificate's purpose. Additionally, unlike $\perp$, which removes a sensor from the working set and adds it to the hypothesis B in the same method that Algorithm 3 does, $\top$ modifies the working set by removing sensors from consideration, as well as disqualifying any models that assume the sensor had failed.

Certificates like $\top$ allow for us to definitively know that a sensor is working. However, self-testing and calibration are not always an option for robots. Additionally, we can imagine that even a robot capable of calibrating and certifying its sensors could have additional limitations, such as it being time and energy consuming. The question that naturally arises is how the robot can determine what sensors should be calibrated to get the largest benefit — namely reducing the number of consistent failure maps to be as small as possible.

5.5.2.2 Complications of the Multi-Hypothesis Space

We note here that one result of the algorithm presented is that, for this example, the robot is unable to definitively determine which of the 25 possible world configurations is the truth. As we are jointly considering the 25 possible configurations as part of the same planning problem, determining which is the true world is a variant of a localization problem. Under traditional methods of state estimation (as covered, for instance, by LaValle (2006)), the robot would disqualify situations that were inconsistent with the history of observations and actions seen thus far, and thus obtain an increasingly accurate estimation of the configuration of the world it occupies. However, the same actions are possible at the same locations no matter where the treasure is located; therefore, actions are insufficient for disqualifying worlds from consideration.

Under the algorithm presented observations also are unable to disqualify worlds from consideration; should an inconsistent sensor reading be obtained, the sensor failure model is updated until consistency is achieved. (This is always possible under the case where we assume that all sensors have failed, which would permit any sensor reading at any location in the world.)

Furthermore, in cases such as the one described here where a failure has occurred, none of the 25 configurations are fully consistent with the agent's execution. To determine which world model is most likely to be the correct one instead requires considering the minimal remaining models after execution of the algorithm; assuming that robots are constructed without malfunctioning sensors, the smallest explanation(s) provided by Algorithm 3 are considered the most likely.

5.6 Using Knowledge of Sensor Failure

Up to this point, we have presented an algorithm that allows for an agent to track possible failures. While this enables the agent to perform diagnosis and fault discovery during execution, the recognition of a fault is only useful if the agent can then use it to make some decision. Within this section, we will turn from the question of failure recognition to how this information can be actively used by an agent.

This section is divided into two main parts: first, we will introduce an algorithm that, given a world p-graph W, is capable of determining if a goal can be reached given a set of inoperable sensors. We then present a second algorithm that is capable of returning for all vertices in W the sensors required to remain operable and reach the goal from that vertex.

5.6.1 Identifying When Problems Become Unsolvable

Given a robot with an array of n sensors, the set of sensors that must be operable for the robot to achieve its goal varies over the course of plan execution. For instance, if the robot described in Section 5.5 has already reached the location where gold is buried and dug it up, all that remains is verifying that the object has been found, and therefore only the sensor which performs this duty must be operable. However, the same robot at the start of execution requires significantly more information in order to locate the buried treasure, which requires that additional sensors be operable. If they are not, the problem may be unsolvable.

We consider a planning problem to be "unsolvable" if there exists no valid plan. Plans are valid if they satisfy the requirements of a solution given in Chapter 3, Definition 4; they produce executions which are finite in length, do not take undefined actions, and only produce event strings that either reach a goal or are a prefix to an execution that reaches a goal. In the case of failure maps, if valid plans existed for a planning problem before application of the failure map and do not after its application, it is because information has been lost from the observation labels that was required for the agent to sufficiently distinguish its state. At some point during execution, the robot will either occupy a set of potential states which share no action among them, or it will enter a condition in which actions taken may trap it in a non-goal state (including actions that may potentially create infinite length executions).

We present Algorithm 4 which, given a world W and failure map $\mathfrak{F}_B$, returns **True** if there exists a plan capable of solving the planning problem, and **False** if not.[7]

Algorithm 4 must check multiple cases before it is capable of returning **True**. In order to enforce the requirement that planning problems be solvable from all initial vertices, the initial set

[7]A simple extension of this algorithm allows for the return of a recursively constructed plan graph, wherein a set of vertices returns itself if it will evaluate to **True**, and includes the outgoing transitions where all target children also return **True**.

passed to SEARCH_UNDER_FAILMAP should be $V = V_{\text{init}}(W)$. However, we can also assign V to an arbitrary set of vertices, such as in cases where there are multiple locations within the world which our agent could be during execution. In fact, this is key to the algorithm's function, as it allows for us to propagate sets of indistinguishable vertices and maintain these indistinguishable groups as a unit. If the function on the initial set of vertices returns **False**, then the problem is not solvable from all vertices in $V_{\text{init}}(W)$, and therefore no valid plan exists. Otherwise, if the search succeeds on all initial vertices, then a plan exists and the algorithm returns **True**.

For each vertex $v \in V$, the algorithm calls the recursive function SEARCH_UNDER_FAILMAP, which takes as arguments a vertex set V and failure map $\mathfrak{F}_B$. The algorithm now must determine if these vertices could be part of a valid plan. If all vertices $v \in V$ are a goal vertex, then the function automatically returns **True**, as a valid path to a goal has been reached. If V is not a set of goals, and V consists of observation vertices, then the algorithm must apply the failure map $\mathfrak{F}_B$ to all outgoing edges and determine what edges are now indistinguishable from each other. (This is performed in a manner similar to that discussed for Algorithm 3, where instead of generating all possible labels, the sensors included in the failure map have their values ignored.)

For both action and observation vertices, labels which are indistinguishable result in groups of child vertices that are also now indistinguishable. For each different label that appears on $v \in V$'s outgoing edges, groupings are made based on targets of edges bearing that label. (Algorithm 4 makes a slight abuse of notation and uses $E(v)$ to represent the set of outgoing edges of a single vertex v, and $E(V)$ to represent the collection of outgoing vertices from all $v \in V$.)

As these vertices are indistinguishable, for a plan to exist it must be valid from all vertices within the group. Therefore, each group is now considered independently, and for each group the function SEARCH_UNDER_FAILMAP is recursively called. If the group returns **False**, what happens next depends on the type of vertex being considered. For observation vertices, the robot does not control what observations are received, and therefore must have a valid plan for any possible outgoing label. Therefore, the failure of a single group indicates that there exists no plan involving V, and the function returns **False**. If V contains action vertices, however, then by the

Algorithm 4: Forward Searching Under Failure

```
Data: world p-graph $W$, failure map $\mathfrak{F}_B$, vertex set $V$
Result: True if a valid plan exists that is capable of solving $W$ from $V$ under $\mathfrak{F}_B$, False otherwise.
1  Function SEARCH_UNDER_FAILMAP($V \subseteq V(W)$, $\mathfrak{F}_B$):
2      if $V \subseteq V_{\text{goal}}(W)$ then
3          return True
4      if all $v \in V$ are an observation vertex then
5          for each vertex $v \in V$ do
6              for each of $v$'s outgoing edges $e \in E(v)$ do
7                  APPLY_FAILMAP($e$, $\mathfrak{F}_B$)
       // obtain all sets of child vertices reached by indistinguishable edges.
8      test_groups = GROUP_INDISTINGUISHABLE($E(V)$)
9      for group in test_groups do
10         if SEARCH_UNDER_FAILMAP(group, $\mathfrak{F}_B$) is False then
11             if $v \in V$ are observations then
12                 return False
13             else
14                 remove group from test_groups
15                 proceed to next group in test_groups
16         else
17             continue
18     return (test_groups $\neq \varnothing$)
```

definition of a plan we require only that there be at least one action the agent can take from all $v \in V$ to make progress towards a goal. Therefore, disqualification of a single group results in the removal of that group from consideration, but the algorithm only returns **False** if, after treatment of all groups, all have been disqualified (and thus there exists *no* action that will take the agent towards a goal).

Algorithm 4 allows for a robot to determine if a given planning problem can still be solved under a certain failure map. If it receives **False**, then the robot has learned that there is no plan.

The notion of a plan being valid, as discussed at the beginning of Subsection 5.6.1, presumes a worst-case scenario where an adversarial world will always "pick" the observation that leads to the worst outcome for the robot. Specifically, if there exists no plan such that a robot cannot guarantee success in the worst circumstances, then it has no choice but to "give up". But whether the robot should actually "give up" is somewhat problem-dependent; in cases where no valid plan exists due to no plan being safe, the agent will have undefined behavior. However, if no valid plan exists because some choice may trap the agent in a non-goal state from which it cannot leave, no plan is live (Chapter 3, Definition 4). Depending on the consequences of ending up in that state (or those of failing the task overall), it may be "worth it" for the robot to attempt to reach the goal and risk being trapped.

It is also entirely possible for there to be some paths within a planning problem where certain sensing information is required, and other paths where it is not. We can easily imagine an optimistic robot can attempt to execute an invalid plan and rely on serendipity to achieve a goal — giving up only when it receives a symbol that crashes or it has no remaining safe actions.

In light of this, knowing only if a planning problem is unsolvable or not is somewhat unsatisfying. Adding to this, the algorithm requires the robot to perform a search over W every time it updates a model, which can be a costly process. Instead, we propose an algorithm that can preprocess a world W and determine, for each vertex, what sensors must be functioning to guarantee that the task can still be completed from that point. Such an algorithm could be run once and have its output stored, allowing the robot to look up the sensing requirements for its current vertex (or vertices) after identifying a failure; if a sensor has failed that is within the required set, the robot can make an informed decision on if it should abandon the current task.

5.6.2 Identifying Sensing Requirements and Boundaries

Sensing requirements are cumulative as we move back from a goal towards the initial vertices of a graph; if an arbitrary vertex v requires sensor j to be functional, then it must also be functional at any point in the execution before the robot reaches v. However, the way in which these requirements accumulate is not as straightforward as it may initially seem.

The existence paths to a goal may also induce differing sensing requirements, based on what information is available at different points. Determining the minimum set of sensors that must be functioning is therefore the first step to determining if an agent can still complete a task after a failure has occurred. In order to do this, we present an algorithm that, given a world W, performs a two-step search.

In order to find what sensed information is strictly required for task completion, we must process vertices moving backwards from the goal. This is because dependencies are cumulative: if a sensor must be working at one vertex in W, it must be working in all vertices that come before it and can reach that vertex. Algorithm 5 begins by performing a reversed topological ordering over all vertices in W. This ensures that all vertices will have their children processed before they are, and allows for them to inherit the dependencies of their children. (We note that this requires that W contain no cycles within it—a limitation required for this section only.)

Due to the branching nature of graphs, a vertex may end up with multiple different sets of required sensors, each of which was derived from the dependencies of a child. If an agent is capable of making a choice in action that leads to two different children with dissimilar dependencies, then it is not necessary for the full set of sensors from all children to be operable, but only that there exist an operable set that is suitable for one of the available paths.

We will refer to an individual set of sensors required for a given execution as a *dependency set*. We will refer to the collection of dependency sets that each vertex possesses as an *option set*. Each vertex is now treated in reverse topological order. For an arbitrary vertex v, this "backward step" can be broken down into several smaller steps:

1) identify cross-child dependencies,

2) identify candidates for additional dependency sets,

3) test each candidate.

For this first step, we identify "cross-child dependencies". Each child of a vertex v has its own option set. By intersecting all of the children's option sets, we obtain any dependency sets required by all children. This set must be included in any dependency set of the parent by default.

The algorithm next identifies candidates for v's option set. (In the case that v is an action vertex, this step is omitted, as an action vertex clearly does not have outgoing observation labels that could be affected by malfunctioning sensors.) Each outgoing edge from v bears a label with n observations, $(Y_1, Y_2, \ldots, Y_n)$. If a given sensor Y_i differs between labels, it is a potential candidate, as the robot possessing the capability to discriminate between them (and thus more accurately knowing its potential state in the world) may be what distinguishes goal achievement from failure.

The algorithm must now enumerate the candidate dependency sets. First, it removes the cross-child dependencies from the option sets (and subtracts them from any sets they may be a subset of). As these must be included by default, they can be excluded from enumeration. Candidates are then enumerated from the collection consisting of the previously identified candidate sensors (transformed into singleton sets) and all dependency sets. The cross-child dependencies are then added back to all generated sets.

The "forward step" of this algorithm is the testing of each candidate dependency set. For each candidate, the algorithm calls Algorithm 4 with v as the initial vertex and the candidate dependency set for the failure map. If no plan exists, then the set contains sensors required for goal attainment.

The primary way in which this algorithm could be made more efficient is means for the reduction of the number of sets enumerated and then tested. For the testing step, although all combinations of dependency sets and candidate sensors to add are generated, they do not all need to be tested. If a sensor is required for goal attainment, any failure map including it will not be able to achieve the goal. Therefore, once a set is identified as containing necessary sensors, any set which contains it as a subset can be removed from testing.

Algorithm 5: Identifying Sensor Dependencies

```
Data: World p-graph $W$
Result: A dictionary that assigns for each vertex $v \in V(W)$ an option set.
required_sensors = CREATE_DICTIONARY()
processing_queue = REVERSE_TOPOLOGICAL_SORT($V(W)$)
for vertex $v$ in processing_queue do
    if $v$ has no child vertices then
        // eliminate from consideration goal states and states that trap the robot
        associated_sensors for $v$ is the empty list []
        continue
    else
        all_dependency_sets = {}
        for each vertex $q$ that is a child of $v$ do
            for dependency set d_set in $q$'s option set do
                add d_set to all_dependency_sets
        child_cross_dependencies = INTERSECT_ALL_MEMBERS(all_dependency_sets)
        candidate_sensors = FIND_DIFFERENCES(all of $v$'s outgoing edges $e \in E(v)$)
        for dependency set d_set in all_dependency_sets do
            subtract child_cross_dependencies from d_set
        add candidate_sensors to all_dependency_sets
        test_dependency_sets = GENERATE_POWERSET(all_dependency_sets)
        for test_set in test_dependency_sets do
            add child_cross_dependencies to test_set
            plan_solvable_under_failure = TEST_FAILURE_MAP(test_set)
            if plan_solvable_under_failure = False then
                // if a plan fails to exist after application of a map,
                // then the sensor is required for goal achievement
                add test_set to required_sensors[$v$]
return required_sensors
```

5.7 Conclusion

During the operational life of a robot, it is likely that it will experience reduced sensor performance and potential failure. The degradation and failure of sensors can be an impediment to completing tasks, and may even create hazardous conditions for the robot or those around it. This chapter has presented a lattice-based method of visualizing the space of all hypotheses for which sensors have failed. We have also presented multiple algorithms that make use of this lattice to answer various questions of relevance to an agent.

We have presented an algorithm that is capable of concisely representing the hypothesis space, a set that may be exponential in size. We have demonstrated the application of the algorithm on an example and presented discussion on how insights from the lattice structure can be used to make informed decisions for calibration and testing.

Going further, we have shown how this information can be used by the agent itself. We presented an algorithm that is capable of deciding if a failure prevents task completion. To further extend this concept, we also presented an algorithm that is capable of identifying the sensors required to be operational at each point in the world, allowing a robot during execution to quickly know if a failure has caused the goal to be unattainable.

We have seen that the lattice structure can be used to identify numerous pieces of information that are of value to robots and their designers. The formalization of these concepts, such as determining if sensors can be rated by importance to a task or similar questions, remains as an avenue for future work. Other future work includes opportunities for optimization of the search algorithms presented within this chapter. Specific directions for the improvement of Algorithm 5 include removing from consideration candidate dependency sets once a subset is known to be required, as well as developing a method by which cycles within the given p-graph can be handled.

6. FUTURE WORK AND CONCLUSION

This dissertation has presented a step towards a principled collection of design theories for robots. We now finish with a discussion of open areas for improvement and continued research, before briefly summarizing the contributions presented in this work.

6.1 Future Work

Future work will be required to further expand the methods given here to cover broader families of robots, as well as to include problems of interest to designers that have been omitted here. The primary open avenues left by this dissertation are future work in identifying sensor failures, as well as in expanding infrastructure models.

As mentioned in Chapter 4, while the decision to disregard inter-agent interference was a deliberate choice to allow for analysis of aggregate effects, there exist cases where such interference cannot be ignored. Such interference from other agents can induce changes in action outcomes and sensed values, similar to infrastructure. Unlike infrastructure, which we have defined as a function applied to an MDP, inter-agent interactions could be reasonably expected to impact both of the agents involved — a difference that requires a framework for MDPs evolving alongside each other.

The development of such a framework would also permit for the introduction of increasingly complicated infrastructure. In reality infrastructure is not always passive, but can maintain information about the world or be used to provide information to agents. This invites future work seeking to extend this framework to consider infrastructure as a unique type of agent itself. POMDPs represent a potential avenue for capturing the uncertainty and complexity of these interactions.

With regard to the given work in identifying sensor failures, there exists the potential for optimization of the algorithms. Algorithm 5 in Chapter 5 could be made more efficient in its enumeration of candidates for new dependency sets, possibly through generating candidates "as needed" over enumerating the entire set. Algorithm 5 is also not capable of handling p-graphs with cycles due to the topological sort performed at the beginning. Deciding on a consistent method for eliminating cycles would increase the number of p-graphs that could be processed by the algorithm.

This dissertation presented a method through which agents can determine if faults can be recovered from, or if the fault prevents the agent from achieving task completion. As noted in Chapter 5, goal achievement for certain tasks may not be a binary objective. In such cases, when recovery is not possible, it may still be desirable for the robot to get as close to achieving the goal as possible. Such an extension would require that the graph framework be modified to include a notion of partial goal completion.

6.2 Conclusion

Over the lifespan of a robot, various problems arise that require understanding how the robot will obtain and make use of information. This dissertation has presented a principled collection of design theories and tools that seek to make explicit the relationship between environment, information requirements, and state. Determining what information is required, as well as what is available, allows for designers to make intelligent design choices. It also allows for designers to identify potential ways in which infrastructure can supplement agent design.

To approach this problem, we made use of two primary tools. First, action-based sensors (Erdmann, 1995) provided a method through which we could directly connect sensors to the tasks for which they are used. Plans and planning problems (jointly) are also treated as first-class objects of study. By doing so, we can re-frame plans from being an object obtained at the end of a process to being an initial description of desired agent behavior. This framing also provides a way in which we can identify how disparate-seeming problems (such as sensing requirements and infrastructure design) share a common basis.

Action-based sensors take a given environment and desired behavior and determine how the environment's structure induces requirements for sensing and storing of information as state. The later chapters built upon this, with infrastructure forming something like a dual of the problem by taking agent capabilities as fixed and examining how changes to the environment resulted in different agent behavior. While these works focused on questions of design and implementation of a robot, we then relaxed the assumption that sensors are trustworthy to explore identifying and handling failures. This work transitioned the focus from design to how knowledge of the environment and the robot's sensing capabilities can be used by the robot itself to make informed decisions concerning task completion.

We have also seen a lattice structure arise naturally multiple times, both in existing work such as the sensor lattice (LaValle, 2011) and within this work through the plan lattice, which allowed us to visualize boundaries between plans that will solve a planning problem and those that will not. This structure also arose in Chapter 5 with the lattice of the hypothesis space, which similarly allowed for us to determine the boundary between feasible and infeasible explanations.

To demonstrate these concepts, this dissertation presented several algorithms for enumerating action-based sensors, tracking hypothesis spaces, and determining if a task can still be completed. Case studies and examples were also presented within each chapter to demonstrate the application and utility of these tools.

REFERENCES

A. Abid, M. T. Khan, and J. Iqbal. A review on fault detection and diagnosis techniques: basics and beyond. *Artificial Intelligence Review*, 54(5):3639–3664, 2021. doi: 10.1007/s10462-020-09934-2.

S. Adams, T. Cody, and P. A. Beling. A survey of inverse reinforcement learning. *Artificial Intelligence Review*, 55(6):4307–4346, 2022. doi: 10.1007/s10462-021-10108-x.

D. Banarse, Y. Bachrach, S. Liu, G. Lever, N. Heess, C. Fernando, P. Kohli, and T. Graepel. The Body is Not a Given: Joint Agent Policy Learning and Morphology Evolution. In *Proc. of the International Conf. on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1134—-1142, 2019.

R. Beckers, O. E. Holland, and J.-L. Deneubourg. From Local Actions to Global Tasks: Stigmergy and Collective Robotics. In *Artificial Life IV: Proc. of the International Workshop on the Synthesis and Simulation of Living Systems*, pages 181–189, Cambridge, MA, USA, July 1994. The MIT Press. doi: 10.7551/mitpress/1428.003.0022.

W. Benbow. Dementia Design: H Shape Facility Layout, January 2011. URL `https://wabenbow.com/?page_id=252`. Accessed 02-26-2022.

D. P. Bertsekas. *Reinforcement Learning and Optimal Control*. Athena Scientific, Belmont, MA., USA, 2019.

M. Blum and D. Kozen. On the power of the compass (or, why mazes are easier to search than graphs). In *19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 132–142, 1978. doi: 10.1109/SFCS.1978.30.

L. Bobadilla, O. Sanchez, J. Czarnowski, K. Gossman, and S. M. LaValle. Controlling wild bodies using linear temporal logic. In *Robotics: Science and Systems*, volume 7, pages 17–24, 2012. doi: 10.15607/rss.2011.vii.003.

R. Brooks and M. Matarić. Real Robots, Real Learning Problems. In *Robot Learning*, pages 193–213. Springer, Boston, MA, USA, 1993. doi: 10.1007/978-1-4615-3184-5_8.

J. M. Buchanan. An Economic Theory of Clubs. *Economica*, 32(125):1–14, Feb 1965.

L. Carlone and C. Pinciroli. Robot co-design: beyond the monotone case. In *2019 International Conf. on Robotics and Automation (ICRA)*, pages 3024–3030. IEEE, 2019. doi: 10.1109/ICRA.2019.8793926.

A. Censi. A mathematical theory of co-design. *arXiv e-prints*, art. arXiv:1512.08055, 2015.

A. Censi. Monotone co-design problems; or, everything is the same. In *Proc. of the American Control Conf. (ACC)*, pages 1227–1234, 2016. doi: 10.1109/acc.2016.7525085.

A. Censi. A Class of Co-Design Problems With Cyclic Constraints and Their Solution. *IEEE Robotics and Automation Letters*, 2(1):96–103, 2017. doi: 10.1109/lra.2016.2535127.

H. Choset and J. Burdick. Sensor based planning. I. The generalized Voronoi graph. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA)*, pages 1649–1655 vol.2, 1995. doi: 10.1109/ROBOT.1995.525511.

M. J.-Y. Chung and M. Cakmak. Iterative Repair of Social Robot Programs from Implicit User Feedback via Bayesian Inference. In *Proc. of Robotics: Science and Systems (RSS)*, Corvalis, Oregon, USA, July 2020. doi: 10.15607/RSS.2020.XVI.028.

A. Ciria, G. Schillaci, G. Pezzulo, V. V. Hafner, and B. Lara. Predictive Processing in Cognitive Robotics: a Review. *Neural Computation*, 33(5):1402–1432, 2021. doi: 10.1162/neco_a_01383.

H. Darvishi, D. Ciuonzo, E. R. Eide, and P. S. Rossi. Sensor-Fault Detection, Isolation and Accommodation for Digital Twins via Modular Data-Driven Architecture. *IEEE Sensors Journal*, 21(4):4827–4838, 2021. doi: 10.1109/JSEN.2020.3029459.

K. Dautenhahn, B. Ogden, and T. Quick. From embodied to socially embedded agents – implications for interaction-aware robots. *Cognitive Systems Research*, 3(3):397–428, 2002. ISSN 1389-0417. doi: 10.1016/S1389-0417(02)00050-5.

R. Desai, J. McCann, and S. Coros. Assembly-aware design of printable electromechanical devices. In *Proc. of the 31st Annual Symposium on User Interface Software and Technology*, pages 457–472, Berlin, Germany, 2018. ACM. doi: 10.1145/3242587.3242655.

M. B. Dias, R. Zlot, N. Kalra, and A. Stentz. Market-based multirobot coordination: A survey and analysis. *Proc. of the IEEE*, 94(7):1257–1270, 2006. doi: 10.1109/JPROC.2006.876939.

B. R. Donald. On information invariants in robotics. *Artificial Intelligence*, 72(1):217–304, 1995. doi: 10.1016/0004-3702(94)00024-U.

B. R. Donald. The Compass That Steered Robotics. In R. L. Constable and A. Silva, editors, *Logic and Program Semantics*, pages 50–65. Springer, Berlin, Heidelberg, 2012. doi: 10.1007/978-3-642-29485-3_5.

A. Dorri, S. S. Kanhere, and R. Jurdak. Multi-Agent Systems: A Survey. *IEEE Access*, 6:28573–28593, 2018. doi: 10.1109/ACCESS.2018.2831228.

B. Edmonds. Capturing Social Embeddedness: a constructivist approach. *Adaptive Behavior*, 7 (3-4):323–347, 1999. doi: 10.1177/105971239900700307.

M. Erdmann. Understanding action and sensing by designing action-based sensors. *The International Journal of Robotics Research*, 14(5):483–509, 1995. doi: 10.1177/027836499501400506.

Federal Aviation Administration. Flight Data Recorder Systems, April 2007. Section 3 Point B, URL `http://rgl.faa.gov/Regulatory_and_Guidance_Library/`.

J. Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*. Addison-Wesley Reading, 1999. ISBN 0-201-36048-9.

J. Ferber and J.-P. Müller. Influences and reaction: a model of situated multiagent systems. In *Proc. of the Second International Conf. on Multiagent Systems*. AAAI, 1996.

B. T. Fine and D. A. Shell. Flocking: Don't need no stinkin' robot recognition. In *2011 IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 5001–5006, 2011. doi: 10.1109/IROS.2011.6095112.

B. T. Fine and D. A. Shell. Eliciting collective behaviors through automatically generated environments. In *Proc. of the IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 3303–3308, 2013. doi: 10.1109/IROS.2013.6696826.

B. P. Gerkey and M. J. Matarić. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation — Special Issue on Multi-robot Systems*, 18(5):758–768, October 2002. doi: 10.1109/TRA.2002.803462.

S. Ghasemlou. *Algorithmic Robot Design: Label Maps, Procrustean Graphs, and the Boundary of Non-Destructiveness*. PhD thesis, University of South Carolina, 2020.

S. Ghasemlou and J. M. O'Kane. Accelerating the construction of boundaries of feasibility in three classes of robot design problems. In *Proc. of IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 2532–2538, 2019. doi: 10.1109/IROS40897.2019.8968258.

P. Goel, G. Dedeoglu, S. I. Roumeliotis, and G. S. Sukhatme. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *Proc. of IEEE International Conf. on Robotics and Automation (ICRA)*, volume 3, pages 2302–2309, 2000. doi: 10.1109/ROBOT.2000.846370.

T.-H. Guo and J. Nurre. Sensor Failure Detection and Recovery by Neural Networks. In *Proc. of International Joint Conf. on Neural Networks*, volume 1, pages 221–226, 1991. doi: 10.1109/IJCNN.1991.155180.

J.-S. Gutmann, P. Fong, L. Chiu, and M. E. Munich. Challenges of designing a low-cost indoor localization system using active beacons. In *Proc. of IEEE Conf. on Technologies for Practical Robot Applications*, pages 1–6, 2013. doi: 10.1109/TePRA.2013.6556348.

J. C. Hammond, J. Biswas, and A. Guha. Automatic failure recovery for end-user programs on service mobile robots. *arXiv e-prints*, art. arXiv:1909.02778, 2019.

M. Hashimoto, H. Kawashima, T. Nakagami, and F. Oba. Sensor fault detection and identification in dead-reckoning system of mobile robot: interacting multiple model approach. In *Proc. of IEEE International Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 1321–1326, 2001. doi: 10.1109/IROS.2001.977165.

R. Hermann and A. Krener. Nonlinear Controllability and Observability. *IEEE Transactions on Automatic Control*, 22(5):728–740, 1977. doi: 10.1109/TAC.1977.1101601.

A. Howard and N. Roy. The robotics data set repository (radish), 2003. URL `http://radish.sourceforge.net/`.

A. J. Ijspeert, A. Martinoli, A. Billard, and L. M. Gambardella. Collaboration through the exploitation of local interactions in autonomous collective robotics: The stick pulling experiment. *Autonomous Robots*, 11(2):149–171, 2001. doi: 10.1023/A:1011227210047.

T. Jacobs and B. Graf. Practical evaluation of service robots for support and routine tasks in an elderly care facility. In *IEEE Workshop on Advanced Robotics and its Social Impacts (ARSO)*, pages 46–49, 2012. doi: 10.1109/ARSO.2012.6213397.

G. O. Kagho, M. Balac, and K. W. Axhausen. Agent-based models in transport planning: Current state, issues, and expectations. *Procedia Computer Science*, 170:726–732, April 2020. doi: 10.1016/j.procs.2020.03.164.

Z. Kalal, K. Mikolajczyk, and J. Matas. Forward-backward error: Automatic detection of tracking failures. In *Proc. of 20th IEEE International Conf. on Pattern Recognition (ICPR)*, pages 2756–2759, 2010. doi: 10.1109/ICPR.2010.675.

R. E. Kalman and J. E. Bertram. General synthesis procedure for computer control of single-loop and multiloop linear systems (an optimal sampling system). *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, 77(6):602–609, 1959. doi: 10.1109/TAI.1959.6371508.

E. Khalastchi and M. Kalech. On fault detection and diagnosis in robotic systems. *ACM Computing Surveys (CSUR)*, 51(1):1–24, 2018. doi: 10.1145/3146389.

M. Lahijanian, M. Svorenova, A. A. Morye, B. Yeomans, D. Rao, I. Posner, P. Newman, H. Kress-Gazit, and M. Kwiatkowska. Resource-Performance Trade-off Analysis for Mobile Robot Design. *arXiv preprint arXiv:1609.04888*, 2016.

S. LaValle and S. Hutchinson. Game theory as a unifying structure for a variety of robot tasks. In *Proc. of 8th IEEE International Symposium on Intelligent Control*, pages 429–434, 1993. doi: 10.1109/ISIC.1993.397675.

S. M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006. ISBN 0-521-86205-1.

S. M. LaValle. Sensor Lattices: A Preimage-Based Approach to Comparing Sensors. Technical report, University of Illinois Urbana-Champaign, Department of Computer Science, 2011. URL `https://msl.cs.uiuc.edu/~lavalle/papers/Lav12.pdf`.

S. M. LaValle. Sensing and Filtering: A Fresh Perspective Based on Preimages and Information Spaces. *Foundations and Trends in Robotics*, 1(4):253–372, 2012. doi: 10.1561/2300000004.

S. M. LaValle. Sensor Lattices: Structures for Comparing Information Feedback. In *International Workshop on Robot Motion and Control (RoMoCo)*, pages 239–246, 2019. doi: 10.1109/RoMoCo.2019.8787364.

M. V. Law, N. Dhawan, H. Bang, S.-Y. Yoon, D. Selva, and G. Hoffman. Side-by-side human–computer design using a tangible user interface. In J. S. Gero, editor, *Design Computing and Cognition '18*, pages 155–173. Springer, Cham, 2019. doi: 10.1007/978-3-030-05363-5_9.

B. Li, D. Song, A. Ramchandani, H.-M. Cheng, D. Wang, Y. Xu, and B. Chen. Virtual lane boundary generation for human-compatible autonomous driving: A tight coupling between perception and planning. In *2019 IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 3733–3739, 2019. doi: 10.1109/IROS40897.2019.8968198.

C. Liang, R. A. Knepper, and F. T. Pokorny. No map, no problem: A local sensing approach for navigation in human-made spaces using signs. In *Proc. of the IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 6148–6155, 2020. doi: 10.1109/IROS45743.2020.9340813.

I. Y. Lin and A. S. Mattila. The Value of Service Robots from the Hotel Guest's Perspective: A Mixed-Method Approach. *International Journal of Hospitality Management*, 94:102876, 2021. doi: 10.1016/j.ijhm.2021.102876.

H. Lipson and J. B. Pollack. Automatic design and manufacture of robotic lifeforms. *Nature*, 406: 974–978, Sept. 2000. doi: 10.1038/35023115.

D. Lubell. A short proof of sperner's lemma. *Journal of Combinatorial Theory*, 1(2):299, Sept. 1966. doi: 10.1016/S0021-9800(66)80035-2.

J. Lunze and J. Richter. Reconfigurable Fault-tolerant Control: A Tutorial Introduction. *European Journal of Control*, 14(5):359–386, 2008. doi: 10.3166/ejc.14.359-386.

P. Maes. Modeling adaptive autonomous agents. *Artificial Life*, 1(1_2):135–162, Oct. 1993. doi: 10.1162/artl.1993.1.1_2.135.

A. Majumdar and V. Pacelli. Fundamental performance limits for sensor-based robot control and policy learning. *arXiv e-prints*, art. arXiv:2202.00129v2, 2022.

M. J. Matarić. Designing emergent behaviors: From local interactions to collective intelligence. In *From Animals to Animats 2: Proc. of the 2nd International Conf. on Simulation of Adaptive Behavior*, pages 432–441. The MIT Press, April 1993. doi: 10.7551/mitpress/3116.003.0059.

M. J. Matarić. Situated Robotics. *Encyclopedia of Cognitive Science*, 4:25–30, Nov. 2002.

M. J. Matarić. Learning in behavior-based multi-robot systems: policies, models, and other agents. *Cognitive Systems Research*, 2(1):81–93, 2001. doi: 10.1016/S1389-0417(01)00017-1.

G. McFassel and D. A. Shell. Every Action-Based Sensor. In S. M. LaValle, M. Lin, T. Ojala, D. A. Shell, and J. Yu, editors, *Algorithmic Foundations of Robotics XIV*, Springer Proceedings in Advanced Robotics, pages 176–193. Springer, Cham, 2020. ISBN 978-3-030-66722-1. doi: 10.1007/978-3-030-66723-8_11.

G. McFassel and D. A. Shell. Reactivity and statefulness: Action-based sensors, plans, and necessary state. *The International Journal of Robotics Research*, 2022. doi: 10.1177/02783649221078874.

M. G. McNally. The Four Step Model. In Hensher, David A. and Button, Kenneth J., editor, *Handbook of Transport Modeling, Second Edition*, chapter 3. Elsevier, Amsterdam ; Oxford, 2007.

H. Mirzaei, G. Sharon, S. Boyles, T. Givargis, and P. Stone. Enhanced delta-tolling: Traffic optimization via policy gradient reinforcement learning. In *Proc. of 21st International Conf. on Intelligent Transportation Systems (ITSC)*, pages 47–52, Nov. 2018. doi: 10.1109/ITSC.2018.8569737.

M. Niemelä and H. Melkas. Robots as Social and Physical Assistants in Elderly Care. In M. Toivonen and E. Saari, editors, *Human-Centered Digitalization and Services*, pages 177–197. Springer Nature Singapore, Singapore, 2019. doi: 10.1007/978-981-13-7725-9_10.

K. J. O'Hara. Towards Robot Systems Architecture. In *AAAI Spring Symposium Series*, 2011.

J. M. O'Kane and S. M. LaValle. Comparing the Power of Robots. *The International Journal of Robotics Research*, 27(1):5–23, 2008. doi: 10.1177/027836490708209.

J. M. O'Kane and D. A. Shell. Concise planning and filtering: hardness and algorithms. *IEEE Transactions on Automation Science and Engineering*, 14(4):1666–1681, 2017. doi: 10.1109/TASE.2017.2701648.

A. Orrick, M. McDermott, D. M. Barnett, E. L. Nelson, and G. N. Williams. Failure detection in an autonomous underwater vehicle. In *Proc. of IEEE International Symposium on Autonomous Underwater Vehicle Technology (AUV)*, pages 377–382, 1994. doi: 10.1109/AUV.1994.518650.

G. Owen. *Game Theory*. Academic, New York, 2nd edition, 1982.

D. Payton, M. Daily, R. Estowski, M. Howard, and C. Lee. Pheromone Robotics. *Autonomous Robots*, 11(3):319–324, Nov. 2001. doi: 10.1023/A:1012411712038.

D. Payton, R. Estkowski, and M. Howard. Compound behaviors in pheromone robotics. *Robotics and Autonomous Systems*, 44(3-4):229–240, 2003. doi: 10.1016/S0921-8890(03)00073-3.

A. Pervan and T. D. Murphey. Algorithmic Design for Embodied Intelligence in Synthetic Cells. *IEEE Transactions on Automation Science and Engineering*, 18(3):864–875, 2021. doi: 10.1109/TASE.2020.3042492.

O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005. doi: 10.1016/j.robot.2005.09.004.

C. Pezzato, M. Baioumy, C. H. Corbato, N. Hawes, M. Wisse, and R. Ferrari. Active Inference for Fault Tolerant Control of Robot Manipulators with Sensory Faults. In *Active Inference: 1st International Workshop (IWAI)*, pages 20–27, Ghent, Belgium, Sept. 2020a. doi: 10.1007/978-3-030-64919-7_3.

C. Pezzato, R. Ferrari, and C. H. Corbato. A Novel Adaptive Controller for Robot Manipulators Based on Active Inference. *IEEE Robotics and Automation Letters*, 5(2):2973–2980, 2020b. doi: 10.1109/LRA.2020.2974451.

C. Plagemann, C. Stachniss, and W. Burgard. Efficient failure detection for mobile robots using mixed-abstraction particle filters. In H. Christensen, editor, *European Robotics Symposium*, Springer Tracts in Advanced Robotics. Springer, Berlin, Heidelberg, 2006. doi: 10.1007/11681120_8.

C. Price and N. Taylor. FMEA for multiple failures. In *Proc. of International Symposium on Product Quality and Integrity: Reliability and Maintainability (RAMS)*, pages 43–47, 1998. doi: 10.1109/RAMS.1998.653556.

F. Rasheed, K.-L. A. Yau, R. M. Noor, C. Wu, and Y.-C. Low. Deep reinforcement learning for traffic signal control: A review. *IEEE Access*, 8:208016–208044, 2020. doi: 10.1109/ACCESS.2020.3034141.

M. Ravula, S. Alkobi, and P. Stone. Ad hoc Teamwork with Behavior Switching Agents. In *Proc. of International Joint Conf. on Artificial Intelligence (IJCAI)*, pages 550–556, August 2019.

J. Rawls. *A Theory of Justice, Revised Edition*. Harvard University Press, Cambridge, MA, 1999.

S. Reig, E. J. Carter, T. Fong, J. Forlizzi, and A. Steinfeld. Flailing, Hailing, Prevailing: Perceptions of Multi-Robot Failure Recovery Strategies. In *Proc. of the ACM/IEEE International Conf. on Human-Robot Interaction (HRI)*, pages 158—-167, Boulder, CO, USA, 2021. ACM New York, NY, USA. doi: 10.1145/3434073.3444659.

M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. The MIT Press, Cambridge, MA, USA, 1994.

A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva. Cognitive Stigmergy: Towards a Framework Based on Agents and Artifacts. In D. Weyns, H. Parunak, and F. Michel, editors, *Environments for Multi-Agent Systems III*, pages 124–140. Springer, Berlin, Heidelberg, 2006. doi: 10.1007/978-3-540-71103-2_7.

T. Roughgarden. Algorithmic game theory. *Communications of the ACM*, 53(7):78–86, 2010. doi: 10.1145/1785414.1785439.

S. I. Roumeliotis, G. S. Sukhatme, and G. A. Bekey. Fault detection and identification in a mobile robot using multiple-model estimation. In *Proc. of IEEE International Conf. on Robotics and Automation (ICRA)*, volume 3, pages 2223–2228, 1998. doi: 10.1109/ROBOT.1998.680654.

S. J. Russell and P. Norvig. Python implementation of algorithms from "Artificial Intelligence—A Modern Approach", 2021. URL `https://github.com/aimacode/aima-python`.

F. Z. Saberifar, S. Ghasemlou, J. M. O'Kane, and D. A. Shell. Set-labelled filters and sensor transformations. In *Proc. of Robotics: Science and Systems*, 2016.

F. Z. Saberifar, S. Ghasemlou, D. A. Shell, and J. M. O'Kane. Toward a language-theoretic foundation for planning and filtering. *The International Journal of Robotics Research*, 38(2-3): 236–259, 2019. doi: 10.1177/0278364918801503.

F. Z. Saberifar, D. A. Shell, and J. M. O'Kane. Charting the trade-off between design complexity and plan execution under probabilistic actions. In *Proc. of IEEE International Conf. on Robotics and Automation (ICRA)*, pages 135–141, 2022. doi: 10.1109/ICRA46639.2022.9811751.

N. Sadeghzadeh-Nokhodberiz and J. Poshtan. Distributed interacting multiple filters for fault diagnosis of navigation sensors in a robotic system. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(7):1383–1393, 2017. doi: 10.1109/TSMC.2016.2598782.

B. Sakcak, V. Weinstein, and S. M. LaValle. The Limits of Learning and Planning: Minimal Sufficient Information Transition Systems. In S. M. LaValle, J. M. O'Kane, M. Otte, D. Sadigh, and P. Tokekar, editors, *Algorithmic Foundations of Robotics XV*, pages 256–272. Springer, Cham, 2023. doi: 10.1007/978-3-031-21090-7_16.

B. Samanta. Gear fault detection using artificial neural networks and support vector machines with genetic algorithms. *Mechanical Systems and Signal Processing*, 18(3):625–644, 2004. doi: 10.1016/S0888-3270(03)00020-7.

T. Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proc. of 11th National Conf. on Artificial Intelligence*, volume 93, pages 256–262, July 1993.

T. Sandler and J. T. Tschirhart. The Economic Theory of Clubs: An Evaluative Survey. *Journal of Economic Literature*, 18(4):1481–1521, Dec 1980.

M. J. Schoppers. Universal Plans for Reactive Robots in Unpredictable Environments. In *Proc. of 10th International Joint Conf. on Artificial Intelligence*, volume 2, pages 1039–1046, 1987.

D. A. Shell, J. M. O'Kane, and F. Z. Saberifar. On the design of minimal robots that can solve planning problems. *IEEE Transactions on Automation Science and Engineering*, 18(3):876–887, 2021. doi: 10.1109/TASE.2021.3050033.

J. Shiroishi, Y. Li, S. Liang, T. Kurfess, and S. Danyluk. Bearing condition diagnostics via vibration and acoustic emission measurements. *Mechanical Systems and Signal Processing*, 11(5): 693–705, 1997. doi: 10.1006/mssp.1997.0113.

T. Sorsa, H. Koivo, and H. Koivisto. Neural networks in process fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4):815–825, 1991. doi: 10.1109/21.108299.

E. Sperner. Ein satz über untermengen einer endlichen menge. *Mathematische Zeitschrift*, 27: 544–548, 1928.

S. Srinivas. *Error Recovery in Robot Systems*. PhD dissertation, California Institute of Technology, 1977.

C. Stachniss and G. Grisetti. Freiburg campus. Accessed June 20, 2021. Available at `http://hdl.handle.net/1721.1/62286` in *The Robotics Data Set Repository (Radish)*, 2010.

G. Steinbauer. A survey about faults of robots used in robocup. In X. Chen, P. Stone, L. E. Sucar, and T. van der Zant, editors, *RoboCup 2012: Robot Soccer World Cup XVI*, pages 344–355. Springer Berlin Heidelberg, 2013.

P. Stone, G. Kaminka, S. Kraus, and J. Rosenschein. Ad Hoc Autonomous Agent Teams: Collaboration Without Pre-Coordination. In *Proc. of the 24th AAAI Conf. on Artificial Intelligence*, volume 24, 2010. doi: 10.1609/aaai.v24i1.7529.

R. H. Taylor, M. T. Mason, and K. Y. Goldberg. Sensor-based manipulation planning as a game with nature. In R. C. Bolles and B. Roth, editors, *Proc. of International Symposium on Robotics Research*, pages 421—429. MIT Press, Cambridge MA. USA, 1988.

R. H. Thaler and C. R. Sunstein. *Nudge: Improving decisions about health, wealth, and happiness*. Penguin, London, UK, 2009. ISBN 978-0-141-04001-1.

B. Tovar, F. Cohen, L. Bobadilla, J. Czarnowski, and S. M. Lavalle. Combinatorial filters: Sensor beams, obstacles, and possible paths. *ACM Transactions on Sensor Networks*, 10(3):1–32, 2014. doi: 10.1145/2594767.

R. T. Vaughan, K. Støy, M. J. Matarić, and G. S. Sukhatme. LOST: Localization-Space Trails for Robot Teams. *IEEE Transactions on Robotics and Automation*, 18(5):796–812, Oct. 2002. doi: 10.1109/TRA.2002.803459.

V. Verma, G. Gordon, R. Simmons, and S. Thrun. Real-Time Fault Diagnosis: Tractable Particle Filters for Robot Fault Diagnosis. *IEEE Robotics & Automation Magazine*, 11(2):56–66, 2004.

G. Wagner, H. Choset, and A. Siravuru. Multirobot Sequential Composition. In *Proc. of IEEE/RSJ International Conf. on Intelligent Robots and Systems (IROS)*, pages 2081–2088, 2016. doi: 10.1109/IROS.2016.7759327.

S. Zapolsky and E. Drumwright. Particle traces for detecting divergent robot behavior. In *Proc. of the IEEE-RAS 16th International Conf. on Humanoid Robots (Humanoids)*, pages 1217–1224, 2016. doi: 10.1109/HUMANOIDS.2016.7803425.

G. Zardini, N. Lanzetti, A. Censi, E. Frazzoli, and M. Pavone. Co-Design to Enable User-Friendly Tools to Assess the Impact of Future Mobility Solutions. arXiv 10.48550/arXiv.2008.08975, 2020.

G. Zardini, A. Censi, and E. Frazzoli. Co-Design of Autonomous Systems: From Hardware Selection to Control Synthesis. In *Proc. of European Control Conf. (ECC)*, pages 682–689, 2021. doi: 10.23919/ECC54610.2021.9654960.

G. Zardini, Z. Suter, A. Censi, and E. Frazzoli. Task-driven Modular Co-design of Vehicle Control Systems. *arXiv e-prints*, art. arXiv:2203.16640, 2022.

Y. Zhang and D. A. Shell. Abstractions for computing all robotic sensors that suffice to solve a planning problem. In *Proc. of the IEEE International Conf. on Robotics and Automation (ICRA)*, pages 8469–8475, 2020. doi: 10.1109/ICRA40945.2020.9196812.

Y. Zhang and D. A. Shell. Lattices of sensors reconsidered when less information is preferred. *arXiv e-prints*, art. arXiv:2106.00805, Oct. 2021.

Y. Zhang, D. A. Shell, and J. M. O'Kane. What does my knowing your plans tell me? *arXiv e-prints*, art. arXiv:1810.03873, Oct. 2018.

D. Zhuo-hua, C. Zi-xing, and Y. Jin-xia. Fault diagnosis and fault tolerant control for wheeled mobile robots under unknown environments: A survey. In *Proc. of IEEE International Conf. on Robotics and Automation (ICRA)*, pages 3428–3433, 2005. doi: 10.1109/ROBOT.2005.1570640.