Part I
Introductory Material

Steven M. LaValle
University of Illinois

Copyright Steven M. LaValle 2006

Available for downloading at http://planning.cs.uiuc.edu/

Published by Cambridge University Press
Chapter 1

Introduction

1.1 Planning to Plan

Planning is a term that means different things to different groups of people. Robotics addresses the automation of mechanical systems that have sensing, actuation, and computation capabilities (similar terms, such as autonomous systems are also used). A fundamental need in robotics is to have algorithms that convert high-level specifications of tasks from humans into low-level descriptions of how to move. The terms motion planning and trajectory planning are often used for these kinds of problems. A classical version of motion planning is sometimes referred to as the Piano Mover’s Problem. Imagine giving a precise computer-aided design (CAD) model of a house and a piano as input to an algorithm. The algorithm must determine how to move the piano from one room to another in the house without hitting anything. Most of us have encountered similar problems when moving a sofa or mattress up a set of stairs. Robot motion planning usually ignores dynamics and other differential constraints and focuses primarily on the translations and rotations required to move the piano. Recent work, however, does consider other aspects, such as uncertainties, differential constraints, modeling errors, and optimality. Trajectory planning refers to the problem of taking the solution from a robot motion planning algorithm and determining how to move along the solution in a way that respects the mechanical limitations of the robot.

Control theory has historically been concerned with designing inputs to physical systems described by differential equations. These could include mechanical systems such as cars or aircraft, electrical systems such as noise filters, or even systems arising in areas as diverse as chemistry, economics, and sociology. Classically, control theory has developed feedback policies, which enable an adaptive response during execution, and has focused on stability, which ensures that the dynamics do not cause the system to become wildly out of control. A large emphasis is also placed on optimizing criteria to minimize resource consumption, such as energy or time. In recent control theory literature, motion planning sometimes refers to the construction of inputs to a nonlinear dynamical system that drives it from an initial state to a specified goal state. For example, imagine trying to operate a remote-controlled hovercraft that glides over the surface of a frozen pond. Suppose we would like the hovercraft to leave its current resting location and come to rest at another specified location. Can an algorithm be designed that computes the desired inputs, even in an ideal simulator that neglects uncertainties that arise from model inaccuracies? It is possible to add other considerations, such as uncertainties, feedback, and optimality; however, the problem is already challenging enough without these.

In artificial intelligence, the terms planning and AI planning take on a more discrete flavor. Instead of moving a piano through a continuous space, as in the robot motion planning problem, the task might be to solve a puzzle, such as the Rubik’s cube or a sliding-tile puzzle, or to achieve a task that is modeled discretely, such as building a stack of blocks. Although such problems could be modeled with continuous spaces, it seems natural to define a finite set of actions that can be applied to a discrete set of states and to construct a solution by giving the appropriate sequence of actions. Historically, planning has been considered different from problem solving; however, the distinction seems to have faded away in recent years. In this book, we do not attempt to make a distinction between the two. Also, substantial effort has been devoted to representation language issues in planning. Although some of this will be covered, it is mainly outside of our focus. Many decision-theoretic ideas have recently been incorporated into the AI planning problem, to model uncertainties, adversarial scenarios, and optimization. These issues are important and are considered in detail in Part III.

Given the broad range of problems to which the term planning has been applied in the artificial intelligence, control theory, and robotics communities, you might wonder whether it has a specific meaning. Otherwise, just about anything could be considered as an instance of planning. Some common elements for planning problems will be discussed shortly, but first we consider planning as a branch of algorithms. Hence, this book is entitled Planning Algorithms. The primary focus is on algorithmic and computational issues of planning problems that have arisen in several disciplines. On the other hand, this does not mean that planning algorithms refers to an existing community of researchers within the general algorithms community. This book is not limited to combinatorics and asymptotic complexity analysis, which is the main focus in pure algorithms. The focus here includes numerous concepts that are not necessarily algorithmic but aid in modeling, solving, and analyzing planning problems.

Natural questions at this point are, What is a plan? How is a plan represented? How is it computed? What is it supposed to achieve? How is its quality evaluated? Who or what is going to use it? This chapter provides general answers to these questions. Regarding the user of the plan, it clearly depends on the application. In most applications, an algorithm executes the plan; however, the user could even be a human. Imagine, for example, that the planning algorithm provides you with
1.2 Motivational Examples and Applications

Planning problems abound. This section surveys several examples and applications to inspire you to read further.

Why study planning algorithms? There are at least two good reasons. First, it is fun to try to get machines to solve problems for which even humans have great difficulty. This involves exciting challenges in modeling planning problems, designing efficient algorithms, and developing robust implementations. Second, planning algorithms have achieved widespread successes in several industries and academic disciplines, including robotics, manufacturing, drug design, and aerospace applications. The rapid growth in recent years indicates that many more fascinating applications may be on the horizon. These are exciting times to study planning algorithms and contribute to their development and use.

Discrete puzzles, operations, and scheduling Chapter 2 covers discrete planning, which can be applied to solve familiar puzzles, such as those shown in Figure 1.1.

Figure 1.1: The Rubik’s cube (a), sliding-tile puzzle (b), and other related puzzles are examples of discrete planning problems.

In this book, the user of the plan will frequently be referred to as a robot or a decision maker. In artificial intelligence and related areas, it has become popular in recent years to use the term agent, possibly with adjectives to yield an intelligent agent or software agent. Control theory usually refers to the decision maker as a controller. The plan in this context is sometimes referred to as a policy or control law. In a game-theoretic context, it might make sense to refer to decision makers as players. Regardless of the terminology used in a particular discipline, this book is concerned with planning algorithms that find a strategy for one or more decision makers. Therefore, remember that terms such as robot, agent, and controller are interchangeable.

A motion planning puzzle The puzzles in Figure 1.1 can be easily discretized because of the regularity and symmetries involved in moving the parts. Figure 1.2 shows a problem that lacks these properties and requires planning in a continuous space. Such problems are solved by using the motion planning techniques of Part II. This puzzle was designed to frustrate both humans and motion planning algorithms. It can be solved in a few minutes on a standard personal computer (PC) using the techniques in Section 5.5. Many other puzzles have been developed as benchmarks for evaluating planning algorithms.

An automotive assembly puzzle Although the problem in Figure 1.2 may appear to be pure fun and games, similar problems arise in important applications. For example, Figure 1.3 shows an automotive assembly problem for which software is needed to determine whether a wiper motor can be inserted (and removed)
1.2. MOTIVATIONAL EXAMPLES AND APPLICATIONS

Figure 1.3: An automotive assembly task that involves inserting or removing a windshield wiper motor from a car body cavity. This problem was solved for clients using the motion planning software of Kineo CAM (courtesy of Kineo CAM).

from the car body cavity. Traditionally, such a problem is solved by constructing physical models. This costly and time-consuming part of the design process can be virtually eliminated in software by directly manipulating the CAD models.

The wiper example is just one of many. The most widespread impact on industry comes from motion planning software developed at Kineo CAM. It has been integrated into Robcad (eM-Workplace) from Tecnomatix, which is a leading tool for designing robotic workcells in numerous factories around the world. Their software has also been applied to assembly problems by Renault, Ford, Airbus, Optivus, and many other major corporations. Other companies and institutions are also heavily involved in developing and delivering motion planning tools for industry (many are secret projects, which unfortunately cannot be described here). One of the first instances of motion planning applied to real assembly problems is documented in [9].

Sealing cracks in automotive assembly Figure 1.4 shows a simulation of robots performing sealing at the Volvo Cars assembly plant in Torslanda, Sweden. Sealing is the process of using robots to spray a sticky substance along the seams of a car body to prevent dirt and water from entering and causing corrosion. The entire robot workcell is designed using CAD tools, which automatically provide the necessary geometric models for motion planning software. The solution shown in Figure 1.4 is one of many problems solved for Volvo Cars and others using motion planning software developed by the Fraunhofer Chalmers Centre (FCC). Using motion planning software, engineers need only specify the high-level task of performing the sealing, and the robot motions are computed automatically. This saves enormous time and expense in the manufacturing process.

Figure 1.4: An application of motion planning to the sealing process in automotive manufacturing. Planning software developed by the Fraunhofer Chalmers Centre (FCC) is used at the Volvo Cars plant in Sweden (courtesy of Volvo Cars and FCC).
1.2. MOTIVATIONAL EXAMPLES AND APPLICATIONS

Moving furniture  Returning to pure entertainment, the problem shown in Figure 1.5 involves moving a grand piano across a room using three mobile robots with manipulation arms mounted on them. The problem is humorously inspired by the phrase Piano Mover’s Problem. Collisions between robots and with other pieces of furniture must be avoided. The problem is further complicated because the robots, piano, and floor form closed kinematic chains, which are covered in Sections 4.4 and 7.4.

Navigating mobile robots  A more common task for mobile robots is to request them to navigate in an indoor environment, as shown in Figure 1.6a. A robot might be asked to perform tasks such as building a map of the environment, determining its precise location within a map, or arriving at a particular place. Acquiring and manipulating information from sensors is quite challenging and is covered in Chapters 11 and 12. Most robots operate in spite of large uncertainties. At one extreme, it may appear that having many sensors is beneficial because it could allow precise estimation of the environment and the robot po-
1.2 MOTIVATIONAL EXAMPLES AND APPLICATIONS

This is the premise of many existing systems, as shown for the robot system in Figure 1.7, which constructs a map of its environment. It may alternatively be preferable to develop low-cost and reliable robots that achieve specific tasks with little or no sensing. These trade-offs are carefully considered in Chapters 11 and 12. Planning under uncertainty is the focus of Part III.

If there are multiple robots, then many additional issues arise. How can the robots communicate? How can their information be integrated? Should their coordination be centralized or distributed? How can collisions between them be avoided? Do they each achieve independent tasks, or are they required to collaborate in some way? If they are competing in some way, then concepts from game theory may apply. Therefore, some game theory appears in Sections 9.3, 9.4, 10.5, 11.7, and 13.5.

Playing hide and seek One important task for a mobile robot is playing the game of hide and seek. Imagine entering a cave in complete darkness. You are given a lantern and asked to search for any people who might be moving about, as shown in Figure 1.6b. Several questions might come to mind. Does a strategy even exist that guarantees I will find everyone? If not, then how many other searchers are needed before this task can be completed? Where should I move next? Can I keep from exploring the same places multiple times? This scenario arises in many robotics applications. The robots can be embedded in surveillance systems that use mobile robots with various types of sensors (motion, thermal, cameras, etc.). In scenarios that involve multiple robots with little or no communication, the strategy could help one robot locate others. One robot could even try to locate another that is malfunctioning. Outside of robotics, software tools can be developed that assist people in systematically searching or covering complicated environments, for applications such as law enforcement, search and rescue, toxic cleanup, and in the architectural design of secure buildings. The problem is extremely difficult because the status of the pursuit must be carefully computed to avoid unnecessarily allowing the evader to sneak back to places already searched. The information-space concepts of Chapter 11 become critical in solving the problem. For an algorithmic solution to the hide-and-seek game, see Section 12.4.

Making smart video game characters The problem in Figure 1.6b might remind you of a video game. In the arcade classic Pacman, the ghosts are programmed to seek the player. Modern video games involve human-like characters that exhibit much more sophisticated behavior. Planning algorithms can enable game developers to program character behaviors at a higher level, with the expectation that the character can determine on its own how to move in an intelligent way.

At present there is a large separation between the planning-algorithm and video-game communities. Some developers of planning algorithms are recently considering more of the particular concerns that are important in video games. Video-game developers have to invest too much energy at present to adapt existing techniques to their problems. For recent books that are geared for game developers, see [6, 21].

Virtual humans and humanoid robots Beyond video games, there is broader interest in developing virtual humans. See Figure 1.8. In the field of computer graphics, computer-generated animations are a primary focus. Animators would like to develop digital actors that maintain many elusive style characteristics of human actors while at the same time being able to design motions for them from high-level descriptions. It is extremely tedious and time consuming to specify all motions frame-by-frame. The development of planning algorithms in this context is rapidly expanding.

Why stop at virtual humans? The Japanese robotics community has inspired the world with its development of advanced humanoid robots. In 1997, Honda shocked the world by unveiling an impressive humanoid that could walk up stairs.
1.2. MOTIVATIONAL EXAMPLES AND APPLICATIONS

(a) (b)

Figure 1.9: (a) This is a picture of the H7 humanoid robot and one of its developers, S. Kagami. It was developed in the JSK Laboratory at the University of Tokyo. (b) Bringing virtual reality and physical reality together. A planning algorithm computes stable motions for a humanoid to grab an obstructed object on the floor [39].

Figure 1.10: Humanoid robots from the Japanese automotive industry: (a) The latest Asimo robot from Honda can run at 3 km/hr (courtesy of Honda); (b) planning is incorporated with vision in the Toyota humanoid so that it plans to grasp objects [27].

Parking cars and trailers The planning problems discussed so far have not involved differential constraints, which are the main focus in Part IV. Consider the problem of parking slow-moving vehicles, as shown in Figure 1.11. Most people have a little difficulty with parallel parking a car and much greater difficulty parking a truck with a trailer. Imagine the difficulty of parallel parking an airport baggage train! See Chapter 13 for many related examples. What makes these problems so challenging? A car is constrained to move in the direction that the rear wheels are pointing. Maneuvering the car around obstacles therefore becomes challenging. If all four wheels could turn to any orientation, this problem would vanish. The term nonholonomic planning encompasses parking problems and many others. Figure 1.12a shows a humorous driving problem. Figure 1.12b shows an extremely complicated vehicle for which nonholonomic planning algorithms were developed and applied in industry.

“Wreckless” driving Now consider driving the car at high speeds. As the speed increases, the car must be treated as a dynamical system due to momentum. The car is no longer able to instantaneously start and stop, which was reasonable for parking problems. Although there exist planning algorithms that address such issues, there are still many unsolved research problems. The impact on industry has not yet reached the level achieved by ordinary motion planning, as shown in Figures 1.3 and 1.4. By considering dynamics in the design process, performance and safety evaluations can be performed before constructing the vehicle. Figure 1.13 shows a solution computed by a planning algorithm that determines how to steer a car at high speeds through a town while avoiding collisions with buildings. A planning algorithm could even be used to assess whether a sports utility vehicle tumbles sideways when stopping too quickly. Tremendous time and costs can be spared by determining design flaws early in the development process via simulations and planning. One related problem is verification, in which a mechanical system design must be thoroughly tested to make sure that it performs as expected in spite of all possible problems that could go wrong during its use. Planning algorithms can also help in this process. For example, the algorithm can try to violently crash a vehicle, thereby establishing that a better design is needed.

Aside from aiding in the design process, planning algorithms that consider dynamics can be directly embedded into robotic systems. Figure 1.13b shows an
1.2. MOTIVATIONAL EXAMPLES AND APPLICATIONS

(a) (b)

Figure 1.11: Some parking illustrations from government manuals for driver testing: (a) parking a car (from the 2005 Missouri Driver Guide); (b) parking a tractor trailer (published by the Pennsylvania Division of Motor Vehicles). Both humans and planning algorithms can solve these problems.

application that involves a difficult combination of most of the issues mentioned so far. Driving across rugged, unknown terrain at high speeds involves dynamics, uncertainties, and obstacle avoidance. Numerous unsolved research problems remain in this context.

Flying Through the Air or in Space

Driving naturally leads to flying. Planning algorithms can help to navigate autonomous helicopters through obstacles. They can also compute thrusts for a spacecraft so that collisions are avoided around a complicated structure, such as a space station. In Section 14.1.3, the problem of designing entry trajectories for a reusable spacecraft is described. Mission planning for interplanetary spacecraft, including solar sails, can even be performed using planning algorithms [26].

Designing better drugs

Planning algorithms are even impacting fields as far away from robotics as computational biology. Two major problems are protein folding and drug design. In both cases, scientists attempt to explain behaviors in organisms by the way large organic molecules interact. Such molecules are generally flexible. Drug molecules are small (see Figure 1.14), and proteins usually have thousands of atoms. The docking problem involves determining whether a flexible molecule can insert itself into a protein cavity, as shown in Figure 1.14, while satisfying other constraints, such as maintaining low energy. Once geometric models are applied to molecules, the problem looks very similar to the assembly problem in Figure 1.3 and can be solved by motion planning algorithms. See Section 7.5 and the literature at the end of Chapter 7.

Perspective

Planning algorithms have been applied to many more problems than those shown here. In some cases, the work has progressed from modeling,
1.3 Basic Ingredients of Planning

Although the subject of this book spans a broad class of models and problems, there are several basic ingredients that arise throughout virtually all of the topics covered as part of planning.

State Planning problems involve a state space that captures all possible situations that could arise. The state could, for example, represent the position and orientation of a robot, the locations of tiles in a puzzle, or the position and velocity of a helicopter. Both discrete (finite, or countably infinite) and continuous (uncountably infinite) state spaces will be allowed. One recurring theme is that the state space is usually represented implicitly by a planning algorithm. In most applications, the size of the state space (in terms of number of states or combinatorial complexity) is much too large to be explicitly represented. Nevertheless, the definition of the state space is an important component in the formulation of a planning problem and in the design and analysis of algorithms that solve it.

Time All planning problems involve a sequence of decisions that must be applied over time. Time might be explicitly modeled, as in a problem such as driving a car as quickly as possible through an obstacle course. Alternatively, time may be implicit, by simply reflecting the fact that actions must follow in succession, as in the case of solving the Rubik’s cube. The particular time is unimportant, but the proper sequence must be maintained. Another example of implicit time is a solution to the Piano Mover’s Problem; the solution to moving the piano may be converted into an animation over time, but the particular speed is not specified in the plan. As in the case of state spaces, time may be either discrete or continuous. In the latter case, imagine that a continuum of decisions is being made by a plan.

Actions A plan generates actions that manipulate the state. The terms actions and operators are common in artificial intelligence; in control theory and robotics, the related terms are inputs and controls. Somewhere in the planning formulation, it must be specified how the state changes when actions are applied. This may be expressed as a state-valued function for the case of discrete time or as an ordinary differential equation for continuous time. For most motion planning problems, explicit reference to time is avoided by directly specifying a path through a continuous state space. Such paths could be obtained as the integral of differential equations, but this is not necessary. For some problems, actions could be chosen by nature, which interfere with the outcome and are not under the control of the decision maker. This enables uncertainty in predictability to be introduced into the planning problem; see Chapter 10.

Initial and goal states A planning problem usually involves starting in some initial state and trying to arrive at a specified goal state or any state in a set of goal states. The actions are selected in a way that tries to make this happen.

A criterion This encodes the desired outcome of a plan in terms of the state and actions that are executed. There are generally two different kinds of planning concerns based on the type of criterion:

1. Feasibility: Find a plan that causes arrival at a goal state, regardless of its efficiency.
2. Optimality: Find a feasible plan that optimizes performance in some carefully specified manner, in addition to arriving in a goal state.

For most of the problems considered in this book, feasibility is already challenging enough; achieving optimality is considerably harder for most problems. Therefore, much of the focus is on finding feasible solutions to problems, as opposed to optimal solutions. The majority of literature in robotics, control theory, and related fields focuses on optimality, but this is not necessarily important for many problems of interest. In many applications, it is difficult to even formulate the
right criterion to optimize. Even if a desirable criterion can be formulated, it may be impossible to obtain a practical algorithm that computes optimal plans. In such cases, feasible solutions are certainly preferable to having no solutions at all. Fortunately, for many algorithms the solutions produced are not too far from optimal in practice. This reduces some of the motivation for finding optimal solutions. For problems that involve probabilistic uncertainty, however, optimization arises more frequently. The probabilities are often utilized to obtain the best performance in terms of expected costs. Feasibility is often associated with performing a worst-case analysis of uncertainties.

A plan In general, a plan imposes a specific strategy or behavior on a decision maker. A plan may simply specify a sequence of actions to be taken; however, it could be more complicated. If it is impossible to predict future states, then the plan can specify actions as a function of state. In this case, regardless of the future states, the appropriate action is determined. Using terminology from other fields, this enables feedback or reactive plans. It might even be the case that the state cannot be measured. In this case, the appropriate action must be determined from whatever information is available up to the current time. This will generally be referred to as an information state, on which the actions of a plan are conditioned.

1.4 Algorithms, Planners, and Plans

Figure 1.15: According to the Church-Turing thesis, the notion of an algorithm is equivalent to the notion of a Turing machine.

1.4.1 Algorithms

What is a planning algorithm? This is a difficult question, and a precise mathematical definition will not be given in this book. Instead, the general idea will be explained, along with many examples of planning algorithms. A more basic question is, What is an algorithm? One answer is the classical Turing machine model, which is used to define an algorithm in theoretical computer science. A Turing machine is a finite state machine with a special head that can read and write along an infinite piece of tape, as depicted in Figure 1.15. The Church-Turing thesis states that an algorithm is a Turing machine (see [29, 55] for more details). The input to the algorithm is encoded as a string of symbols (usually a binary string) and then is written to the tape. The Turing machine reads the string, performs computations, and then decides whether to accept or reject the string. This version of the Turing machine only solves decision problems; however, there are straightforward extensions that can yield other desired outputs, such as a plan.

The Turing model is reasonable for many of the algorithms in this book; however, others may not exactly fit. The trouble with using the Turing machine in some situations is that plans often interact with the physical world. As indicated in Figure 1.16, the boundary between the machine and the environment is an arbitrary line that varies from problem to problem. Once drawn, sensors provide information about the environment; this provides input to the machine during execution. The machine then executes actions, which provides actuation to the environment. The actuation may alter the environment in some way that is later measured by sensors. Therefore, the machine and its environment are closely coupled during execution. This is fundamental to robotics and many other fields in which planning is used.

Using the Turing machine as a foundation for algorithms usually implies that the physical world must be first carefully modeled and written on the tape before the algorithm can make decisions. If changes occur in the world during execution of the algorithm, then it is not clear what should happen. For example, a mobile robot could be moving in a cluttered environment in which people are walking around. As another example, a robot might throw an object onto a table without being able to precisely predict how the object will come to rest. It can take measurements of the results with sensors, but it again becomes a difficult task to determine how much information should be explicitly modeled and written on the tape. The on-line algorithm model is more appropriate for these kinds of problems.
1.4. ALGORITHMS, PLANNERS, AND PLANS

Infinite Row of Switches
Turing Robot

Figure 1.17: A robot and an infinite sequence of switches could be used to simulate a Turing machine. Through manipulation, however, many other kinds of behavior could be obtained that fall outside of the Turing model.

Processes that occur in a physical world are more complicated than the interaction between a state machine and a piece of tape filled with symbols. It is even possible to simulate the tape by imagining a robot that interacts with a long row of switches as depicted in Figure 1.17. The switches serve the same purpose as the tape, and the robot carries a computer that can simulate the finite state machine.\(^1\) The complicated interaction allowed between a robot and its environment could give rise to many other models of computation.\(^2\) Thus, the term \textit{algorithm} will be used somewhat less formally than in the theory of computation. Both \textit{planners} and \textit{plans} are considered as algorithms in this book.

1.4.2 Planners

A planner simply constructs a plan and may be a machine or a human. If the planner is a machine, it will generally be considered as a planning algorithm. In many circumstances it is an algorithm in the strict Turing sense; however, this is not necessary. In some cases, humans become planners by developing a plan that works in all situations. For example, it is perfectly acceptable for a human to design a state machine that is connected to the environment (see Section 12.3.1). There are no additional inputs in this case because the human fulfills the role of the algorithm. The planning model is given as input to the human, and the human “computes” a plan.

1.4.3 Plans

Once a plan is determined, there are three ways to use it:

\(^1\)Of course, having infinitely long tape seems impossible in the physical world. Other versions of Turing machines exist in which the tape is finite but as long as necessary to process the given input. This may be more appropriate for the discussion.

\(^2\)Performing computations with mechanical systems is discussed in [52]. Computation models over the reals are covered in [5].

1. \textbf{Execution:} Execute it either in simulation or in a mechanical device (robot) connected to the physical world.

2. \textbf{Refinement:} Refine it into a better plan.

3. \textbf{Hierarchical Inclusion:} Package it as an action in a higher level plan.

Each of these will be explained in succession.

\textbf{Execution} A plan is usually executed by a machine. A human could alternatively execute it; however, the case of machine execution is the primary focus of this book. There are two general types of machine execution. The first is depicted in Figure 1.18a, in which the planner produces a plan, which is encoded in some way and given as input to the machine. There are two general types of machine execution. The first is depicted in Figure 1.18a, in which the planner produces a plan, which is encoded in some way and given as input to the machine. In this case, the machine is considered \textit{programmable} and can accept possible plans from a planner before execution. It will generally be assumed that once the plan is given, the machine becomes autonomous and can no longer interact with the planner. Of course, this model could be extended to allow machines to be improved over time by receiving better plans; however, we want a strict notion of autonomy for the discussion of planning in this book. This approach does not prohibit the updating of plans in practice; however, this is not preferred because plans should already be designed to take into account new information during execution.

The second type of machine execution of a plan is depicted in Figure 1.18b. In this case, the plan produced by the planner encodes an entire machine. The plan is a special-purpose machine that is designed to solve the specific tasks given originally to the planner. Under this interpretation, one may be a \textit{minimalist} and design the simplest machine possible that sufficiently solves the desired tasks. If the plan is encoded as a finite state machine, then it can sometimes be considered
1.4. ALGORITHMS, PLANNERS, AND PLANS

Compute a collision-free path

Design a feedback control law that tracks the trajectory

Execute the feedback plan

Geometric model of the world

Smooth it to satisfy some differential constraints

Design a trajectory (velocity function) along the path

Figure 1.19: A refinement approach that has been used for decades in robotics.

Figure 1.20: In a hierarchical model, the environment of one machine may itself contain a machine.

as an algorithm in the Turing sense (depending on whether connecting the machine to a tape preserves its operation).

**Refinement** If a plan is used for refinement, then a planner accepts it as input and determines a new plan that is hopefully an improvement. The new plan may take more problem aspects into account, or it may simply be more efficient. Refinement may be applied repeatedly, to produce a sequence of improved plans, until the final one is executed. Figure 1.19 shows a refinement approach used in robotics. Consider, for example, moving an indoor mobile robot. The first plan yields a collision-free path through the building. The second plan transforms the route into one that satisfies differential constraints based on wheel motions (recall Figure 1.11). The third plan considers how to move the robot along the path at various speeds while satisfying momentum considerations. The fourth plan incorporates feedback to ensure that the robot stays as close as possible to the planned path in spite of unpredictable behavior. Further elaboration on this approach and its trade-offs appears in Section 14.6.1.

**Hierarchical inclusion** Under hierarchical inclusion, a plan is incorporated as an action in a larger plan. The original plan can be imagined as a subroutine in the larger plan. For this to succeed, it is important for the original plan to guarantee termination, so that the larger plan can execute more actions as needed. Hierarchical inclusion can be performed any number of times, resulting in a rooted tree of plans. This leads to a general model of hierarchical planning. Each vertex in the tree is a plan. The root vertex represents the master plan. The children of any vertex are plans that are incorporated as actions in the plan of the vertex. There is no limit to the tree depth or number of children per vertex. In hierarchical planning, the line between machine and environment is drawn in multiple places. For example, the environment, $E_1$, with respect to a machine, $M_1$, might actually include another machine, $M_2$, that interacts with its environment, $E_2$, as depicted in Figure 1.20. Examples of hierarchical planning appear in Sections 7.3.2 and 12.5.1.

1.5 Organization of the Book

Here is a brief overview of the book. See also the overviews at the beginning of Parts II–IV.

**PART I: Introductory Material**

This provides very basic background for the rest of the book.

- **Chapter 1: Introductory Material**
  This chapter offers some general perspective and includes some motivational examples and applications of planning algorithms.

- **Chapter 2: Discrete Planning**
  This chapter covers the simplest form of planning and can be considered as a springboard for entering into the rest of the book. From here, you can continue to Part II, or even head straight to Part III. Sections 2.1 and 2.2 are most important for heading into Part II. For Part III, Section 2.3 is additionally useful.

**PART II: Motion Planning**

The main source of inspiration for the problems and algorithms covered in this part is robotics. The methods, however, are general enough for use in other applications in other areas, such as computational biology, computer-aided design, and computer graphics. An alternative title that more accurately reflects the kind of planning that occurs is “Planning in Continuous State Spaces.”

- **Chapter 3: Geometric Representations and Transformations**
  The chapter gives important background for expressing a motion planning problem. Section 3.1 describes how to construct geometric models, and the remaining sections indicate how to transform them. Sections 3.1 and 3.2 are important for later chapters.

- **Chapter 4: The Configuration Space**
  This chapter introduces concepts from topology and uses them to formulate the configuration space, which is the state space that arises in motion planning. Sections 4.1, 4.2, and 4.3.1 are important for understanding most of the material in later chapters. In addition to the previously mentioned
sections, all of Section 4.3 provides useful background for the combinatorial methods of Chapter 6.

- **Chapter 5: Sampling-Based Motion Planning**
  This chapter introduces motion planning algorithms that have dominated the literature in recent years and have been applied in fields both in and out of robotics. If you understand the basic idea that the configuration space represents a continuous state space, most of the concepts should be understandable. They even apply to other problems in which continuous state spaces emerge, in addition to motion planning and robotics. Chapter 14 revisits sampling-based planning, but under differential constraints.

- **Chapter 6: Combinatorial Motion Planning**
  The algorithms covered in this section are sometimes called *exact algorithms* because they build discrete representations without losing any information. They are *complete*, which means that they must find a solution if one exists; otherwise, they report failure. The sampling-based algorithms have been more useful in practice, but they only achieve weaker notions of completeness.

- **Chapter 7: Extensions of Basic Motion Planning**
  This chapter introduces many problems and algorithms that are extensions of the methods from Chapters 5 and 6. Most can be followed with basic understanding of the material from these chapters. Section 7.4 covers planning for closed kinematic chains; this requires an understanding of the additional material, from Section 4.4

- **Chapter 8: Feedback Motion Planning**
  This is a transitional chapter that introduces feedback into the motion planning problem but still does not introduce differential constraints, which are deferred until Part IV. The previous chapters of Part II focused on computing *open-loop* plans, which means that any errors that might occur during execution of the plan are ignored, yet the plan will be executed as planned. Using feedback yields a *closed-loop* plan that responds to unpredictable events during execution.

**PART III: Decision-Theoretic Planning**

An alternative title to Part III is “Planning Under Uncertainty.” Most of Part III addresses discrete state spaces, which can be studied immediately following Part I. However, some sections cover extensions to continuous spaces; to understand these parts, it will be helpful to have read some of Part II.

- **Chapter 9: Basic Decision Theory**
  The main idea in this chapter is to design the best decision for a decision maker that is confronted with interference from other decision makers. The others may be true opponents in a game or may be fictitious in order to model uncertainties. The chapter focuses on making a decision in a single step and provides a building block for Part III because planning under uncertainty can be considered as multi-step decision making.

- **Chapter 10: Sequential Decision Theory**
  This chapter takes the concepts from Chapter 9 and extends them by chaining together a sequence of basic decision-making problems. Dynamic programming concepts from Section 2.3 become important here. For all of the problems in this chapter, it is assumed that the current state is always known. All uncertainties that exist are with respect to prediction of future states, as opposed to measuring the current state.

- **Chapter 11: Sensors and Information Spaces**
  The chapter extends the formulations of Chapter 10 into a framework for planning when the current state is unknown during execution. Information regarding the state is obtained from sensor observations and the memory of actions that were previously applied. The information space serves a similar purpose for problems with sensing uncertainty as the configuration space has for motion planning.

- **Chapter 12: Planning Under Sensing Uncertainty**
  This chapter covers several planning problems and algorithms that involve sensing uncertainty. This includes problems such as localization, map building, pursuit-evasion, and manipulation. All of these problems are unified under the idea of planning in information spaces, which follows from Chapter 11.

**PART IV: Planning Under Differential Constraints**

This can be considered as a continuation of Part II. Here there can be both global (obstacles) and local (differential) constraints on the continuous state spaces that arise in motion planning. Dynamical systems are also considered, which yields state spaces that include both position and velocity information (this coincides with the notion of a state space in control theory or a phase space in physics and differential equations).

- **Chapter 13: Differential Models**
  This chapter serves as an introduction to Part IV by introducing numerous models that involve differential constraints. This includes constraints that arise from wheels rolling as well as some that arise from the dynamics of mechanical systems.

- **Chapter 14: Sampling-Based Planning Under Differential Constraints**
  Algorithms for solving planning problems under the models of Chapter 13
are presented. Many algorithms are extensions of methods from Chapter 5. All methods are sampling-based because very little can be accomplished with combinatorial techniques in the context of differential constraints.

- **Chapter 15: System Theory and Analytical Techniques**
  This chapter provides an overview of the concepts and tools developed mainly in control theory literature. They are complementary to the algorithms of Chapter 14 and often provide important insights or components in the development of planning algorithms under differential constraints.
Chapter 2

Discrete Planning

This chapter provides introductory concepts that serve as an entry point into other parts of the book. The planning problems considered here are the simplest to describe because the state space will be finite in most cases. When it is not finite, it will at least be countably infinite (i.e., a unique integer may be assigned to every state). Therefore, no geometric models or differential equations will be needed to characterize the discrete planning problems. Furthermore, no forms of uncertainty will be considered, which avoids complications such as probability theory. All models are completely known and predictable.

There are three main parts to this chapter. Sections 2.1 and 2.2 define and present search methods for feasible planning, in which the only concern is to reach a goal state. The search methods will be used throughout the book in numerous other contexts, including motion planning in continuous state spaces. Following feasible planning, Section 2.3 addresses the problem of optimal planning. The principle of optimality, or the dynamic programming principle, [1] provides a key insight that greatly reduces the computation effort in many planning algorithms. The value-iteration method of dynamic programming is the main focus of Section 2.3. The relationship between Dijkstra’s algorithm and value iteration is also discussed. Finally, Sections 2.4 and 2.5 describe logic-based representations of planning and methods that exploit these representations to make the problem easier to solve; material from these sections is not needed in later chapters.

Although this chapter addresses a form of planning, it encompasses what is sometimes referred to as problem solving. Throughout the history of artificial intelligence research, the distinction between problem solving [45] and planning has been rather elusive. The widely used textbook by Russell and Norvig [53] provides a representative, modern survey of the field of artificial intelligence. Two of its six main parts are termed “problem-solving” and “planning”; however, their definitions are quite similar. The problem-solving part begins by stating, “Problem solving agents decide what to do by finding sequences of actions that lead to desirable states” ([53], p. 59). The planning part begins with, “The task of coming up with a sequence of actions that will achieve a goal is called planning” ([53], p. 375). Also, the STRIPS system [19] is widely considered as a seminal planning algorithm, and the “PS” part of its name stands for “Problem Solver.” Thus, problem solving and planning appear to be synonymous. Perhaps the term “planning” carries connotations of future time, whereas “problem solving” sounds somewhat more general. A problem-solving task might be to take evidence from a crime scene and piece together the actions taken by suspects. It might seem odd to call this a “plan” because it occurred in the past.

Since it is difficult to make clear distinctions between problem solving and planning, we will simply refer to both as planning. This also helps to keep with the theme of this book. Note, however, that some of the concepts apply to a broader set of problems than what is often meant by planning.

2.1 Introduction to Discrete Feasible Planning

2.1.1 Problem Formulation

The discrete feasible planning model will be defined using state-space models, which will appear repeatedly throughout this book. Most of these will be natural extensions of the model presented in this section. The basic idea is that each distinct situation for the world is called a state, denoted by $x$, and the set of all possible states is called a state space, $X$. For discrete planning, it will be important that this set is countable; in most cases it will be finite. In a given application, the state space should be defined carefully so that irrelevant information is not encoded into a state (e.g., a planning problem that involves moving a robot in France should not encode information about whether certain light bulbs are on in China). The inclusion of irrelevant information can easily convert a problem that is amenable to efficient algorithmic solutions into one that is intractable. On the other hand, it is important that $X$ is large enough to include all information that is relevant to solve the task.

The world may be transformed through the application of actions that are chosen by the planner. Each action, $u$, when applied from the current state, $x$, produces a new state, $x'$, as specified by a state transition function, $f$. It is convenient to use $f$ to express a state transition equation,

$$x' = f(x, u).$$  \hspace{1cm} (2.1)

Let $U(x)$ denote the action space for each state $x$, which represents the set of all actions that could be applied from $x$. For distinct $x, x' \in X$, $U(x)$ and $U(x')$ are not necessarily disjoint; the same action may be applicable in multiple states. Therefore, it is convenient to define the set $U$ of all possible actions over all states:

$$U = \bigcup_{x \in X} U(x).$$  \hspace{1cm} (2.2)

As part of the planning problem, a set $X_G \subset X$ of goal states is defined. The task of a planning algorithm is to find a finite sequence of actions that when
applied, transforms the initial state $x_I$ to some state in $X_G$. The model is summarized as:

**Formulation 2.1 (Discrete Feasible Planning)**

1. A nonempty *state space* $X$, which is a finite or countably infinite set of states.
2. For each state $x \in X$, a finite *action space* $U(x)$.
3. A *state transition function* $f$ that produces a state $f(x, u) \in X$ for every $x \in X$ and $u \in U(x)$. The *state transition equation* is derived from $f$ as $x' = f(x, u)$.
4. An *initial state* $x_I \in X$.
5. A *goal set* $X_G \subset X$.

It is often convenient to express Formulation 2.1 as a directed *state transition graph*. The set of vertices is the state space $X$. A directed edge from $x \in X$ to $x' \in X$ exists in the graph if and only if there exists an action $u \in U(x)$ such that $x' = f(x, u)$. The initial state and goal set are designated as special vertices in the graph, which completes the representation of Formulation 2.1 in graph form.

### 2.1.2 Examples of Discrete Planning

**Example 2.1 (Moving on a 2D Grid)** Suppose that a robot moves on a grid in which each grid point has integer coordinates of the form $(i, j)$. The robot takes discrete steps in one of four directions (up, down, left, right), each of which increments or decrements one coordinate. The motions and corresponding state transition graph are shown in Figure 2.1, which can be imagined as stepping from tile to tile on an infinite tile floor.

This will be expressed using Formulation 2.1. Let $X$ be the set of all integer pairs of the form $(i, j)$, in which each grid point has integer coordinates of the form $(i, j)$. Let $U = \{(0,1), (0,-1), (1,0), (-1,0)\}$. Let $U(x) = U$ for all $x \in X$. The state transition equation is $f(x, u) = x + u$, in which $x \in X$ and $u \in U$ are treated as two-dimensional vectors for the purpose of addition. For example, if $x = (3,4)$ and $u = (0,1)$, then $f(x, u) = (3,5)$. Suppose for convenience that the initial state is $x_I = (0,0)$. Many interesting goal sets are possible. Suppose, for example, that $X_G = \{(100,100)\}$. It is easy to find a sequence of actions that transforms the state from $(0,0)$ to $(100,100)$.

The problem can be made more interesting by shading in some of the square tiles to represent obstacles that the robot must avoid, as shown in Figure 2.2. In this case, any tile that is shaded has its corresponding vertex and associated edges deleted from the state transition graph. An outer boundary can be made to fence in a bounded region so that $X$ becomes finite. Very complicated labyrinths can be constructed.

**Example 2.2 (Rubik’s Cube Puzzle)** Many puzzles can be expressed as discrete planning problems. For example, the Rubik’s cube is a puzzle that looks like an array of $3 \times 3 \times 3$ little cubes, which together form a larger cube as shown in Figure 1.1a (Section 1.2). Each face of the larger cube is painted one of six colors. An action may be applied to the cube by rotating a $3 \times 3$ sheet of cubes by 90 degrees. After applying many actions to the Rubik’s cube, each face will generally be a jumble of colors. The state space is the set of configurations for
the cube (the orientation of the entire cube is irrelevant). For each state there are 12 possible actions. For some arbitrarily chosen configuration of the Rubik’s cube, the planning task is to find a sequence of actions that returns it to the configuration in which each one of its six faces is a single color. □

It is important to note that a planning problem is usually specified without explicitly representing the entire state transition graph. Instead, it is revealed incrementally in the planning process. In Example 2.1, very little information actually needs to be given to specify a graph that is infinite in size. If a planning problem is given as input to an algorithm, close attention must be paid to the encoding when performing a complexity analysis. For a problem in which \( X \) is infinite, the input length must still be finite. For some interesting classes of problems it may be possible to compactly specify a model that is equivalent to Formulation 2.1. Such representation issues have been the basis of much research in artificial intelligence over the past decades as different representation logics have been proposed; see Section 2.4 and [23]. In a sense, these representations can be viewed as input compression schemes.

Readers experienced in computer engineering might recognize that when \( X \) is finite, Formulation 2.1 appears almost identical to the definition of a finite state machine or Mealy/Moore machines. Relating the two models, the actions can be interpreted as inputs to the state machine, and the output of the machine simply reports its state. Therefore, the feasible planning problem (if \( X \) is finite) may be interpreted as determining whether there exists a sequence of inputs that makes a finite state machine eventually report a desired output. From a planning perspective, it is assumed that the planning algorithm has a complete specification of the machine transitions and is able to read its current state at any time.

Readers experienced with theoretical computer science may observe similar connections to a deterministic finite automaton (DFA), which is a special kind of finite state machine that reads an input string and makes a decision about whether to accept or reject the string. The input string is just a finite sequence of inputs, in the same sense as for a finite state machine. A DFA definition includes a set of accept states, which in the planning context can be renamed to the goal set. This makes the feasible planning problem (if \( X \) is finite) equivalent to determining whether there exists an input string that is accepted by a given DFA. Usually, a language is associated with a DFA, which is the set of all strings it accepts. DFAs are important in the theory of computation because their languages correspond precisely to regular expressions. The planning problem amounts to determining whether the empty language is associated with the DFA.

Thus, there are several ways to represent and interpret the discrete feasible planning problem that sometimes lead to a very compact, implicit encoding of the problem. This issue will be revisited in Section 2.4. Until then, basic planning algorithms are introduced in Section 2.2, and discrete optimal planning is covered in Section 2.3.

2.2 Searching for Feasible Plans

The methods presented in this section are just graph search algorithms, but with the understanding that the state transition graph is revealed incrementally through the application of actions, instead of being fully specified in advance. The presentation in this section can therefore be considered as visiting graph search algorithms from a planning perspective. An important requirement for these or any search algorithms is to be systematic. The requirement to be systematic is that, in the limit, as the number of iterations tends to infinity, all reachable vertices are reached.

![Figure 2.3](image.png) (a) Many search algorithms focus too much on one direction, which may prevent them from being systematic on infinite graphs. (b) If, for example, the search carefully expands in wavefronts, then it becomes systematic. The requirement to be systematic is that, in the limit, as the number of iterations tends to infinity, all reachable vertices are reached.
The set of alive states is stored in a priority queue, $Q$, for which a priority function must be specified. The only significant difference between various search algorithms is the particular function used to sort $Q$. Many variations will be described later, but for the time being, it might be helpful to pick one. Therefore, assume for now that $Q$ is a common FIFO (First-In First-Out) queue; whichever state has been waiting the longest will be chosen when $Q.GetFirst()$ is called. The rest of the general search algorithm is quite simple. Initially, $Q$ contains the initial state $x_I$. A while loop is then executed, which terminates only when $Q$ is empty. This will only occur when the entire graph has been explored without finding any goal states, which results in a FAILURE (unless the reachable portion of $X$ is infinite, in which case the algorithm should never terminate). In each while iteration, the highest ranked element, $x$, of $Q$ is removed. If $x$ lies in $X_G$, then it reports SUCCESS and terminates; otherwise, the algorithm tries applying every possible action, $u \in U(x)$. For each next state, $x' = f(x, u)$, it must determine whether $x'$ is being encountered for the first time. If it is unvisited, then it is inserted into $Q$; otherwise, there is no need to consider it because it must be either dead or already in $Q$.

The algorithm description in Figure 2.4 omits several details that often become important in practice. For example, how efficient is the test to determine whether $x \in X_G$ in line 4? This depends, of course, on the size of the state space and on the particular representations chosen for $x$ and $X_G$. At this level, we do not specify a particular method because the representations are not given.

One important detail is that the existing algorithm only indicates whether a solution exists, but does not seem to produce a plan, which is a sequence of actions that achieves the goal. This can be fixed by inserting a line after line 7 that associates with $x'$ its parent, $x$. If this is performed each time, one can simply trace the pointers from the final state to the initial state to recover the plan. For convenience, one might also store which action was taken, in addition to the pointer from $x'$ to $x$.

Lines 8 and 9 are conceptually simple, but how can one tell whether $x'$ has been visited? For some problems the state transition graph might actually be a tree, which means that there are no repeated states. Although this does not occur frequently, it is wonderful when it does because there is no need to check whether states have been visited. If the states in $X$ all lie on a grid, one can simply make a lookup table that can be accessed in constant time to determine whether a state has been visited. In general, however, it might be quite difficult because the state $x'$ must be compared with every other state in $Q$ and with all of the dead states. If the representation of each state is long, as is sometimes the case, this will be very costly. A good hashing scheme or another clever data structure can greatly alleviate this cost, but in many applications the computation time will remain high. One alternative is to simply allow repeated states, but this could lead to an increase in computational cost that far outweighs the benefits. Even if the graph is very small, search algorithms could run in time exponential in the size of the

---

FORWARD_SEARCH
1. $Q.Insert(x_I)$ and mark $x_I$ as visited
2. while $Q$ not empty do
3. \hspace{1em} $x \leftarrow Q.GetFirst()$
4. \hspace{1em} if $x \in X_G$ then
5. \hspace{2em} return SUCCESS
6. \hspace{1em} for all $u \in U(x)$
7. \hspace{2em} $x' \leftarrow f(x, u)$
8. \hspace{2em} if $x'$ not visited then
9. \hspace{3em} Mark $x'$ as visited
10. \hspace{2em} $Q.Insert(x')$
11. \hspace{1em} else
12. \hspace{2em} Resolve duplicate $x'$
13. return FAILURE

---

Figure 2.4: A general template for forward search.

depicted in Figure 2.3a, then in the limit most of the space will be left uncovered, even though no states are revisited. If instead the search proceeds outward from the origin in wavefronts, as depicted in Figure 2.3b, then it may be systematic. In practice, each search algorithm has to be carefully analyzed. A search algorithm could expand in multiple directions, or even in wavefronts, but still not be systematic. If the graph is finite, then it is much simpler: Virtually any search algorithm is systematic, provided that it marks visited states to avoid revisiting the same states indefinitely.

### 2.2.1 General Forward Search

Figure 2.4 gives a general template of search algorithms, expressed using the state-space representation. At any point during the search, there will be three kinds of states:

1. **Unvisited**: States that have not been visited yet. Initially, this is every state except $x_I$.

2. **Dead**: States that have been visited, and for which every possible next state has also been visited. A next state of $x$ is a state $x'$ for which there exists a $u \in U(x)$ such that $x' = f(x, u)$. In a sense, these states are dead because there is nothing more that they can contribute to the search; there are no new leads that could help in finding a feasible plan. Section 2.3.3 discusses a variant in which dead states can become alive again in an effort to obtain optimal plans.

3. **Alive**: States that have been encountered, but possibly have unvisited next states. These are considered alive. Initially, the only alive state is $x_I$. 

### 2.2.2 SEARCHING FOR FEASIBLE PLANS

The set of alive states is stored in a priority queue, $Q$, for which a priority function must be specified. The only significant difference between various search algorithms is the particular function used to sort $Q$. Many variations will be described later, but for the time being, it might be helpful to pick one. Therefore, assume for now that $Q$ is a common FIFO (First-In First-Out) queue; whichever state has been waiting the longest will be chosen when $Q.GetFirst()$ is called. The rest of the general search algorithm is quite simple. Initially, $Q$ contains the initial state $x_I$. A while loop is then executed, which terminates only when $Q$ is empty. This will only occur when the entire graph has been explored without finding any goal states, which results in a FAILURE (unless the reachable portion of $X$ is infinite, in which case the algorithm should never terminate). In each while iteration, the highest ranked element, $x$, of $Q$ is removed. If $x$ lies in $X_G$, then it reports SUCCESS and terminates; otherwise, the algorithm tries applying every possible action, $u \in U(x)$. For each next state, $x' = f(x, u)$, it must determine whether $x'$ is being encountered for the first time. If it is unvisited, then it is inserted into $Q$; otherwise, there is no need to consider it because it must be either dead or already in $Q$.

The algorithm description in Figure 2.4 omits several details that often become important in practice. For example, how efficient is the test to determine whether $x \in X_G$ in line 4? This depends, of course, on the size of the state space and on the particular representations chosen for $x$ and $X_G$. At this level, we do not specify a particular method because the representations are not given.

One important detail is that the existing algorithm only indicates whether a solution exists, but does not seem to produce a plan, which is a sequence of actions that achieves the goal. This can be fixed by inserting a line after line 7 that associates with $x'$ its parent, $x$. If this is performed each time, one can simply trace the pointers from the final state to the initial state to recover the plan. For convenience, one might also store which action was taken, in addition to the pointer from $x'$ to $x$.

Lines 8 and 9 are conceptually simple, but how can one tell whether $x'$ has been visited? For some problems the state transition graph might actually be a tree, which means that there are no repeated states. Although this does not occur frequently, it is wonderful when it does because there is no need to check whether states have been visited. If the states in $X$ all lie on a grid, one can simply make a lookup table that can be accessed in constant time to determine whether a state has been visited. In general, however, it might be quite difficult because the state $x'$ must be compared with every other state in $Q$ and with all of the dead states. If the representation of each state is long, as is sometimes the case, this will be very costly. A good hashing scheme or another clever data structure can greatly alleviate this cost, but in many applications the computation time will remain high. One alternative is to simply allow repeated states, but this could lead to an increase in computational cost that far outweighs the benefits. Even if the graph is very small, search algorithms could run in time exponential in the size of the
state transition graph, or the search may not terminate at all, even if the graph is finite.

One final detail is that some search algorithms will require a cost to be computed and associated with every state. If the same state is reached multiple times, the cost may have to be updated, which is performed in line 12, if the particular search algorithm requires it. Such costs may be used in some way to sort the priority queue, or they may enable the recovery of the plan on completion of the algorithm. Instead of storing pointers, as mentioned previously, the optimal cost to return to the initial state could be stored with each state. This cost alone is sufficient to determine the action sequence that leads to any visited state. Starting at a visited state, the path back to \( x_1 \) can be obtained by traversing the state transition graph backward in a way that decreases the cost as quickly as possible in each step. For this to succeed, the costs must have a certain monotonicity property, which is obtained by Dijkstra’s algorithm and \( A^* \) search, and will be introduced in Section 2.2.2. More generally, the costs must form a navigation function, which is considered in Section 8.2.2 as feedback is incorporated into discrete planning.

### 2.2.2 Particular Forward Search Methods

This section presents several search algorithms, each of which constructs a search tree. Each search algorithm is a special case of the algorithm in Figure 2.4, obtained by defining a different sorting function for \( Q \). Most of these are just classical graph search algorithms [12].

**Breadth first** The method given in Section 2.2.1 specifies \( Q \) as a First-In-First-Out (FIFO) queue, which selects states using the first-come, first-serve principle. This causes the search frontier to grow uniformly and is therefore referred to as breadth-first search. All plans that have \( k \) steps are exhausted before plans with \( k + 1 \) steps are investigated. Therefore, breadth first guarantees that the first solution found will use the smallest number of steps. On detection that a state has been revisited, there is no work to do in line 12. Since the search progresses in a series of wavefronts, breadth-first search is systematic. In fact, it even remains systematic if it does not keep track of repeated states (however, it will waste time considering irrelevant cycles).

The asymptotic running time of breadth-first search is \( O(|V| + |E|) \), in which \( |V| \) and \( |E| \) are the numbers of vertices and edges, respectively, in the state transition graph (recall, however, that the graph is usually not the input; for example, the input may be the rules of the Rubik’s cube). This assumes that all basic operations, such as determining whether a state has been visited, are performed in constant time. In practice, these operations will typically require more time and must be counted as part of the algorithm’s complexity. The running time can be expressed in terms of the other representations. Recall that \( |V| = |X| \) is the number of states. If the same actions \( U \) are available from every state, then \( |E| = |U||X| \). If the action sets \( U(x_1) \) and \( U(x_2) \) are pairwise disjoint for any \( x_1, x_2 \in X \), then \( |E| = |U| \).

**Depth first** By making \( Q \) a stack (Last-In, First-Out; or LIFO), aggressive exploration of the state transition graph occurs, as opposed to the uniform expansion of breadth-first search. The resulting variant is called depth-first search because the search dives quickly into the graph. The preference is toward investigating longer plans very early. Although this aggressive behavior might seem desirable, note that the particular choice of longer plans is arbitrary. Actions are applied in the **for all** loop in whatever order they happen to be defined. Once again, if a state is revisited, there is no work to do in line 12. Depth-first search is systematic for any finite \( X \) but not for an infinite \( X \) because it could behave like Figure 2.3a. The search could easily focus on one “direction” and completely miss large portions of the search space as the number of iterations tends to infinity. The running time of depth first search is also \( O(|V| + |E|) \).

**Dijkstra’s algorithm** Up to this point, there has been no reason to prefer one action over any other in the search. Section 2.3 will formalize optimal discrete planning and will present several algorithms that find optimal plans. Before going into that, we present a systematic search algorithm that finds optimal plans because it is also useful for finding feasible plans. The result is the well-known Dijkstra’s algorithm for finding single-source shortest paths in a graph [18], which is a special form of dynamic programming. More general dynamic programming computations appear in Section 2.3 and throughout the book.

Suppose that every edge, \( e \in E \), in the graph representation of a discrete planning problem has an associated nonnegative cost \( l(e) \), which is the cost to apply the action. The cost \( l(e) \) could be written using the state-space representation as \( l(x, u) \), indicating that it costs \( l(x, u) \) to apply action \( u \) from state \( x \). The total cost of a plan is just the sum of the edge costs over the path from the initial state to a goal state.

The priority queue, \( Q \), will be sorted according to a function \( C : X \to [0, \infty] \), called the cost-to-come. For each state \( x \), the value \( C^*(x) \) is called the optimal cost-to-come from the initial state \( x_1 \). This optimal cost is obtained by summing edge costs, \( l(e) \), over all possible paths from \( x_1 \) to \( x \) and using the path that produces the least cumulative cost. If the cost is not known to be optimal, then it is written as \( C(x) \).

The cost-to-come is computed incrementally during the execution of the search algorithm in Figure 2.4. Initially, \( C^*(x_1) = 0 \). Each time the state \( x' \) is generated, a cost is computed as \( C(x') = C^*(x) + l(e) \), in which \( e \) is the edge from \( x \) to \( x' \) (equivalently, we may write \( C(x') = C^*(x) + l(x, u) \)). Here, \( C(x') \) represents the best cost-to-come that is known so far, but we do not write \( C^* \) because it is not defined for all states.

\[ ^1 \text{As in optimization literature, we will use } ^* \text{ to mean optimal.} \]
yet known whether \( x' \) was reached optimally. Due to this, some work is required in line 12. If \( x' \) already exists in \( Q \), then it is possible that the newly discovered path to \( x' \) is more efficient. If so, then the cost-to-come value \( C(x') \) must be lowered for \( x' \), and \( Q \) must be reordered accordingly.

When does \( C(x) \) finally become \( C^*(x) \) for some state \( x \)? Once \( x \) is removed from \( Q \) using \( Q.GetFirst() \), the state becomes dead, and it is known that \( x \) cannot be reached with a lower cost. This can be argued by induction. For the initial state, \( C^*(x_I) \) is known, and this serves as the base case. Now assume that every dead state has its optimal cost-to-come correctly determined. This means that their cost-to-come values can no longer change. For the first element, \( x \), of \( Q \), the value must be optimal because any path that has a lower total cost would have to travel through another state in \( Q \), but these states already have higher costs. All paths that pass only through dead states were already considered in producing \( C(x) \). Once all edges leaving \( x \) are explored, then \( x \) can be declared as dead, and the induction continues. This is not enough detail to constitute a proof of optimality; more arguments appear in Section 2.3.3 and in [12]. The running time is \( O(|V| \lg |V| + |E|) \), in which \(|V| \) and \(|E| \) are the numbers of edges and vertices, respectively, in the graph representation of the discrete planning problem. This assumes that the priority queue is implemented with a Fibonacci heap, and that all other operations, such as determining whether a state has been visited, are performed in constant time. If other data structures are used to implement the priority queue, then higher running times may be obtained.

**A-star** The \( A^* \) (pronounced “ay star”) search algorithm is an extension of Dijkstra’s algorithm that tries to reduce the total number of states explored by incorporating a heuristic estimate of the cost to get to the goal from a given state. Let \( C(x) \) denote the cost-to-come from \( x_I \) to \( x \), and let \( G(x) \) denote the cost-to-go from \( x \) to some state in \( X_G \). It is convenient that \( C^*(x) \) can be computed incrementally by dynamic programming; however, there is no way to know the true optimal cost-to-go, \( G^* \), in advance. Fortunately, in many applications it is possible to construct a reasonable underestimate of this cost. As an example of a typical underestimate, consider planning in the labyrinth depicted in Figure 2.2. Suppose that the cost is the total number of steps in the plan. If one state has coordinates \((i, j)\) and another has \((i', j')\), then \( |i' - i| + |j' - j| \) is an underestimate because this is the length of a straightforward plan that ignores obstacles. Once obstacles are included, the cost can only increase as the robot tries to get around them (which may not even be possible). Of course, zero could also serve as an underestimate, but that would not provide any helpful information to the algorithm. The aim is to compute an estimate that is as close as possible to the optimal cost-to-go and is also guaranteed to be no greater. Let \( G^*(x) \) denote such an estimate.

The \( A^* \) search algorithm works in exactly the same way as Dijkstra’s algorithm. The only difference is the function used to sort \( Q \). In the \( A^* \) algorithm, the sum

\[
C^*(x') + \hat{G}^*(x')
\]

is used, implying that the priority queue is sorted by estimates of the optimal cost from \( x_I \) to \( X_G \). If \( \hat{G}^*(x) \) is an underestimate of the true optimal cost-to-go for all \( x \in X \), the \( A^* \) algorithm is guaranteed to find optimal plans [19, 49]. As \( \hat{G}^* \) becomes closer to \( G^* \), fewer vertices tend to be explored in comparison with Dijkstra’s algorithm. This would always seem advantageous, but in some problems it is difficult or impossible to find a heuristic that is both efficient to evaluate and provides good search guidance. Note that when \( \hat{G}^*(x) = 0 \) for all \( x \in X \), then \( A^* \) degenerates to Dijkstra’s algorithm. In any case, the search will always be systematic.

**Best first** For *best-first search*, the priority queue is sorted according to an estimate of the optimal cost-to-go. The solutions obtained in this way are not necessarily optimal; therefore, it does not matter whether the estimate exceeds the true optimal cost-to-go, which was important to maintain optimality for \( A^* \) search. Although optimal solutions are not found, in many cases, far fewer vertices are explored, which results in much faster running times. There is no guarantee,
however, that this will happen. The worst-case performance of best-first search is worse than that of $A^*$ search and dynamic programming. The algorithm is often too greedy because it prefers states that “look good” very early in the search. Sometimes the price must be paid for being greedy! Figure 2.5 shows a contrived example in which the planning problem involves taking small steps in a 3D world. For any specified number, $k$, of steps, it is easy to construct a spiral example that wastes at least $k$ steps in comparison to Dijkstra’s algorithm. Note that best-first search is not systematic.

**Iterative deepening** The iterative deepening approach is usually preferable if the search tree has a large branching factor (i.e., there are many more vertices in the next level than in the current level). This could occur if there are many actions per state and only a few states are revisited. The idea is to use depth-first search and find all states that are distance $i$ or less from $x_I$. If the goal is not found, then the previous work is discarded, and depth first is applied to find all states of distance $i + 1$ or less from $x_I$. This generally iterates from $i = 1$ and proceeds indefinitely until the goal is found. Iterative deepening can be viewed as a way of converting depth-first search into a systematic search method. The motivation for discarding the work of previous iterations is that the number of states reached for $i + 1$ is expected to far exceed (e.g., by a factor of 10) the number reached for $i$. Therefore, once the commitment has been made to reach level $i + 1$, the cost of all previous iterations is negligible.

The iterative deepening method has better worst-case performance than breadth-first search for many problems. Furthermore, the space requirements are reduced because the queue in breadth-first search is usually much larger than for depth-first search. If the nearest goal state is $i$ steps from $x_I$, breadth-first search in the worst case might reach nearly all states of distance $i + 1$ before terminating successfully. This occurs each time a state $x \not\in X_G$ of distance $i$ from $x_I$ is reached because all new states that can be reached in one step are placed onto $Q$. The $A^*$ idea can be combined with iterative deepening to yield $IDA^*$, in which $i$ is replaced by $C^*(x') + G^*(x')$. In each iteration of $IDA^*$, the allowed total cost gradually increases [49].

### 2.2.3 Other General Search Schemes

This section covers two other general templates for search algorithms. The first one is simply a “backward” version of the tree search algorithm in Figure 2.4. The second one is a bidirectional approach that grows two search trees, one from the initial state and one from a goal state.

**Backward search** Backward versions of any of the forward search algorithms of Section 2.2.2 can be made. For example, a backward version of Dijkstra’s algorithm can be made by starting from $x_G$. To create backward search algorithms, suppose that there is a single goal state, $x_G$. For many planning problems, it might be the case that the branching factor is large when starting from $x_I$. In this case, it might be more efficient to start the search at a goal state and work backward until the initial state is encountered. A general template for this approach is given in Figure 2.6. For forward search, recall that an action $u \in U(x)$ is applied from $x \in X$ to obtain a new state, $x' = f(x, u)$. For backward search, a frequent computation will be to determine for some $x'$, the preceding state $x \in X$, and action $u \in U(x)$ such that $x' = f(x, u)$. The template in Figure 2.6 can be extended to handle a goal region, $X_G$, by inserting all $x_G \in X_G$ into $Q$ in line 1 and marking them as visited.

For most problems, it may be preferable to precompute a representation of the state transition function, $f$, that is “backward” to be consistent with the search algorithm. Some convenient notation will now be constructed for the backward version of $f$. Let $U^{-1} = \{(x, u) \in X \times U \mid x \in X, u \in U(x)\}$, which represents the set of all state-action pairs and can also be considered as the domain of $f$. Imagine from a given state $x' \in X$, the set of all $(x, u) \in U^{-1}$ that map to $x'$ using $f$. This can be considered as a backward action space, defined formally for any $x' \in X$ as

$$U^{-1}(x') = \{(x, u) \in U^{-1} \mid x' = f(x, u)\}.$$  

For convenience, let $u^{-1}$ denote a state-action pair $(x, u)$ that belongs to some $U^{-1}(x')$. From any $u^{-1} \in U^{-1}(x')$, there is a unique $x \in X$. Thus, let $f^{-1}$ denote a backward state transition function that yields $x$ from $x'$ and $u^{-1} \in U^{-1}(x')$. This defines a backward state transition equation, $x = f^{-1}(x', u^{-1})$, which looks very similar to the forward version, $x' = f(x, u)$.

The interpretation of $f^{-1}$ is easy to capture in terms of the state transition graph: reverse the direction of every edge. This makes finding a path in the reversed graph using backward search equivalent to finding one in the original graph using forward search. The backward state transition function is the variant of $f$ that is obtained after reversing all of the edges. Each $u^{-1}$ is a reversed edge. Since there is a perfect symmetry with respect to the forward search of Section 2.2.1, any of the search algorithm variants from Section 2.2.2 can be adapted to the template in Figure 2.6, provided that $f^{-1}$ has been defined.

**Bidirectional search** Now that forward and backward search have been covered, the next reasonable idea is to conduct a bidirectional search. The general search template given in Figure 2.7 can be considered as a combination of the two in Figures 2.4 and 2.6. One tree is grown from the initial state, and the other is grown from the goal state (assume again that $X_G$ is a singleton, \(\{x_G\}\)). The search terminates with success when the two trees meet. Failure occurs if either priority queue has been exhausted. For many problems, bidirectional search can dramatically reduce the amount of required exploration. There are Dijkstra and $A^*$ variants of bidirectional search, which lead to optimal solutions. For best-first and other variants, it may be challenging to ensure that the two trees meet.
2.2. SEARCHING FOR FEASIBLE PLANS

BACKWARD_SEARCH
1. $Q.I n s e r t (x_G)$ and mark $x_G$ as visited
2. while $Q$ not empty do
3. $x' \leftarrow Q . G e t F i r s t ()$
4. if $x = x_I$
5. return SUCCESS
6. forall $u^{-1} \in U^{-1}(x)$
7. $x \leftarrow f^{-1}(x', u^{-1})$
8. if $x$ not visited
9. Mark $x$ as visited
10. $Q . I n s e r t (x)$
11. else
12. Resolve duplicate $x$
13. return FAILURE

Figure 2.6: A general template for backward search.

quickly. They might come very close to each other and then fail to connect. Additional heuristics may help in some settings to guide the trees into each other. One can even extend this framework to allow any number of search trees. This may be desirable in some applications, but connecting the trees becomes even more complicated and expensive.

2.2.4 A Unified View of the Search Methods

It is convenient to summarize the behavior of all search methods in terms of several basic steps. Variations of these steps will appear later for more complicated planning problems. For example, in Section 5.4, a large family of sampling-based motion planning algorithms can be viewed as an extension of the steps presented here. The extension in this case is made from a discrete state space to a continuous state space (called the configuration space). Each method incrementally constructs a search graph, $G(V, E)$, which is the subgraph of the state transition graph that has been explored so far.

All of the planning methods from this section followed the same basic template:

1. **Initialization**: Let the search graph, $G(V, E)$, be initialized with $E$ empty and $V$ containing some starting states. For forward search, $V = \{x_I\}$; for backward search, $V = \{x_G\}$. If bidirectional search is used, then $V = \{x_I, x_G\}$. It is possible to grow more than two trees and merge them during the search process. In this case, more states can be initialized in $V$. The search graph will incrementally grow to reveal more and more of the state transition graph.

2. **Select Vertex**: Choose a vertex $n_{c u r} \in V$ for expansion; this is usually

BIDIRECTIONAL_SEARCH
1. $Q_I . I n s e r t (x_I)$ and mark $x_I$ as visited
2. $Q_G . I n s e r t (x_G)$ and mark $x_G$ as visited
3. while $Q_I$ not empty and $Q_G$ not empty do
4. if $Q_I$ not empty
5. $x \leftarrow Q_I . G e t F i r s t ()$
6. if $x$ already visited from $x_G$
7. return SUCCESS
8. forall $u \in U(x)$
9. $x' \leftarrow f(x, u)$
10. if $x'$ not visited
11. Mark $x'$ as visited
12. $Q_I . I n s e r t (x')$
13. else
14. Resolve duplicate $x'$
15. if $Q_G$ not empty
16. $x' \leftarrow Q_G . G e t F i r s t ()$
17. if $x'$ already visited from $x_I$
18. return SUCCESS
19. forall $u^{-1} \in U^{-1}(x')$
20. $x \leftarrow f^{-1}(x', u^{-1})$
21. if $x$ not visited
22. Mark $x$ as visited
23. $Q_G . I n s e r t (x)$
24. else
25. Resolve duplicate $x$
26. return FAILURE

Figure 2.7: A general template for bidirectional search.
accomplished by maintaining a priority queue. Let \( x_{\text{cur}} \) denote the state associated with \( n_{\text{cur}} \).

3. **Apply an Action:** In either a forward or backward direction, a new state, \( x_{\text{new}} \), is obtained. This may arise from \( x_{\text{new}} = f(x, u) \) for some \( u \in U(x) \) (forward) or \( x = f(x_{\text{new}}, u) \) for some \( u \in U(x_{\text{new}}) \) (backward).

4. **Insert a Directed Edge into the Graph:** If certain algorithm-specific tests are passed, then generate an edge from \( x \) to \( x_{\text{new}} \) for the forward case, or an edge from \( x_{\text{new}} \) to \( x \) for the backward case. If \( x_{\text{new}} \) is not yet in \( V \), it will be inserted into \( V \).

5. **Check for Solution:** Determine whether \( G \) encodes a path from \( x_1 \) to \( x_G \). If there is a single search tree, then this is trivial. If there are two or more search trees, then this step could be expensive.

6. **Return to Step 2:** Iterate unless a solution has been found or an early termination condition is satisfied, in which case the algorithm reports failure.

Note that in this summary, several iterations may have to be made to generate one iteration in the previous formulations. The forward search algorithm in Figure 2.4 tries all actions for the first element of \( Q \). If there are \( k \) actions, this corresponds to \( k \) iterations in the template above.

### 2.3 Discrete Optimal Planning

This section extends Formulation 2.1 to allow optimal planning problems to be defined. Rather than being satisfied with any sequence of actions that leads to the goal set, suppose we would like a solution that optimizes some criterion, such as time, distance, or energy consumed. Three important extensions will be made: 1) A stage index will be used to conveniently indicate the current plan step; 2) a cost functional will be introduced, which behaves like a taxi meter by indicating how much cost accumulates during the plan execution; and 3) a termination action will be introduced, which intuitively indicates when it is time to stop the plan and fix the total cost.

The presentation involves three phases. First, the problem of finding optimal paths of a fixed length is covered in Section 2.3.1. The approach, called value iteration, involves iteratively computing optimal cost-to-go functions over the state space. Although this case is not very useful by itself, it is much easier to understand than the general case of variable-length plans. Once the concepts from this section are understood, their extension to variable-length plans will be much clearer and is covered in Section 2.3.2. Finally, Section 2.3.3 explains the close relationship between value iteration and Dijkstra’s algorithm, which was covered in Section 2.2.1.

With nearly all optimization problems, there is the arbitrary, symmetric choice of whether to define a criterion to minimize or maximize. If the cost is a kind of energy or expense, then minimization seems sensible, as is typical in robotics and control theory. If the cost is a kind of reward, as in investment planning or in most AI books, then maximization is preferred. Although this issue remains throughout the book, we will choose to minimize everything. If maximization is instead preferred, then multiplying the costs by \(-1\) and swapping minimizations should suffice.

The fixed-length optimal planning formulation will be given shortly, but first we introduce some new notation. Let \( \pi_k \) denote a \( K \)-step plan, which is a sequence \((u_1, u_2, \ldots, u_K)\) of \( K \) actions. If \( \pi_K \) and \( x_1 \) are given, then a sequence of states, \((x_1, x_2, \ldots, x_{K+1})\), can be derived using the state transition function, \( f \). Initially, \( x_1 = x_I \), and each subsequent state is obtained by \( x_{k+1} = f(x_k, u_k) \).

The model is now given; the most important addition with respect to Formulation 2.1 is \( L \), the cost functional.

**Formulation 2.2 (Discrete Fixed-Length Optimal Planning)**

1. All of the components from Formulation 2.1 are inherited directly: \( X, U(x), f, X_G \), except here it is assumed that \( X \) is finite (some algorithms may easily extend to the case in which \( X \) is countably infinite, but this will not be considered here).

2. A number, \( K \), of stages, which is the exact length of a plan (measured as the number of actions, \( u_1, u_2, \ldots, u_K \)). States may also obtain a stage index. For example, \( x_{k+1} \) denotes the state obtained after \( u_k \) is applied.

3. Let \( L \) denote a stage-additive cost (or loss) functional, which is applied to a \( K \)-step plan, \( \pi_K \). This means that the sequence \((u_1, \ldots, u_K)\) of actions and the sequence \((x_1, \ldots, x_{K+1})\) of states may appear in an expression of \( L \). For convenience, let \( F \) denote the final stage, \( F = K + 1 \) (the application of \( u_K \) advances the stage to \( K + 1 \)). The cost functional is

\[
L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \tag{2.4}
\]

The cost term \( l(x_k, u_k) \) yields a real value for every \( x_k \in X \) and \( u_k \in U(x_k) \). The final term \( l_F(x_F) \) is outside of the sum and is defined as \( l_F(x_F) = 0 \) if \( x_F \in X_G \), and \( l_F(x_F) = \infty \) otherwise.

An important comment must be made regarding \( l_F \). Including \( l_F \) in (2.4) is actually unnecessary if it is agreed in advance that \( L \) will only be applied to evaluate plans that reach \( X_G \). It would then be undefined for all other plans. The

\[2\text{In some variations, the vertex could be added without a corresponding edge. This would start another tree in a multiple-tree approach}\]
algorithms to be presented shortly will also function nicely under this assumption; however, the notation and explanation can become more cumbersome because the action space must always be restricted to ensure that successful plans are produced. Instead of this, the domain of \( L \) is extended to include all plans, and those that do not reach \( X_G \) are penalized with infinite cost so that they are eliminated automatically in any optimization steps. At some point, the role of \( l_F \) may become confusing, and it is helpful to remember that it is just a trick to convert feasibility constraints into a straightforward optimization (\( L(\pi_K) = \infty \) means not feasible and \( L(\pi_K) < \infty \) means feasible with cost \( L(\pi_K) \)).

Now the task is to find a plan that minimizes \( L \). To obtain a feasible planning problem like Formulation 2.1 but restricted to \( K \)-step plans, let \( l(x, u) \equiv 0 \). To obtain a planning problem that requires minimizing the number of stages, let \( l(x, u) \equiv 1 \). The possibility also exists of having goals that are less “crisp” by letting \( l_F(x) \) vary for different \( x \in X_G \), as opposed to \( l_F(x) = 0 \). This is much more general than what was allowed with feasible planning because now states may take on any value, as opposed to being classified as inside or outside of \( X_G \).

### 2.3.1 Optimal Fixed-Length Plans

Consider computing an optimal plan under Formulation 2.2. One could naively generate all length-\( K \) sequences of actions and select the sequence that produces the best cost, but this would require \( O(|U|^K) \) running time (imagine \( K \) nested loops, one for each stage), which is clearly prohibitive. Luckily, the dynamic programming principle helps. We first say in words what will appear later in equations. The main observation is that portions of optimal plans are themselves optimal. It would be absurd to be able to replace a portion of an optimal plan with a portion that produces lower total cost; this contradicts the optimality of the original plan.

The principle of optimality leads directly to an iterative algorithm, called value iteration,\(^3\) that can solve a vast collection of optimal planning problems, including those that involve variable-length plans, stochastic uncertainties, imperfect state measurements, and many other complications. The idea is to iteratively compute optimal cost-to-go (or cost-to-come) functions over the state space. In some cases, the approach can be reduced to Dijkstra’s algorithm; however, this only occurs under some special conditions. The value-iteration algorithm will be presented next, and Section 2.3.3 discusses its connection to Dijkstra’s algorithm.

#### Backward value iteration

As for the search methods, there are both forward and backward versions of the approach. The backward case will be covered first. Even though it may appear superficially to be easier to progress from \( x_I \), it turns out that progressing backward from \( X_G \) is notationally simpler. The forward case will then be covered once some additional notation is introduced.

The key to deriving long optimal plans from shorter ones lies in the construction of optimal cost-to-go functions over \( X \). For \( k \) from 1 to \( F \), let \( G_k^* \) denote the cost that accumulates from stage \( k \) to \( F \) under the execution of the optimal plan:

\[
G_k^*(x_k) = \min_{u_k, \ldots, u_K} \left\{ \sum_{i=k}^{K} l(x_i, u_i) + l_F(x_F) \right\}.
\]

Inside of the min of (2.5) are the last \( F - k \) terms of the cost functional, (2.4). The optimal cost-to-go for the boundary condition of \( k = F \) reduces to

\[
G_F^*(x_F) = l_F(x_F).
\]

This makes intuitive sense: Since there are no stages in which an action can be taken, the final stage cost is immediately received.

Now consider an algorithm that makes \( K \) passes over \( X \), each time computing \( G_k^* \) from \( G_{k+1}^* \), as \( k \) ranges from \( F \) down to 1. In the first iteration, \( G_F^* \) is copied from \( l_F(x) \) without significant effort. In the second iteration, \( G_K^* \) is computed for each \( x_k \in X \) as

\[
G_K^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + l_F(x_F) \right\}.
\]

Since \( l_F = G_F^* \) and \( x_F = f(x_k, u_k) \), substitutions can be made into (2.7) to obtain

\[
G_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + G_F^*(f(x_k, u_k)) \right\},
\]

which is straightforward to compute for each \( x_k \in X \). This computes the costs of all optimal one-step plans from stage \( k \) to stage \( F = K + 1 \). It will be shown next that \( G_k^* \) can be computed similarly once \( G_{k+1}^* \) is given. Carefully study (2.5) and note that it can be written as

\[
G_k^*(x_k) = \min_{u_k} \left\{ \min_{u_{k+1}, \ldots, u_K} \left\{ l(x_k, u_k) + \sum_{i=k+1}^{K} l(x_i, u_i) + l_F(x_F) \right\} \right\}.
\]

by pulling the first term out of the sum and by separating the minimization over \( u_k \) from the rest, which range from \( u_{k+1} \) to \( u_K \). The second min does not affect the \( l(x_k, u_k) \) term; thus, \( l(x_k, u_k) \) can be pulled outside to obtain

\[
G_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + \min_{u_{k+1}, \ldots, u_K} \left\{ \sum_{i=k+1}^{K} l(x_i, u_i) + l_F(x_F) \right\} \right\}.
\]

The inner min is exactly the definition of the optimal cost-to-go function \( G_{k+1}^* \).

Upon substitution, this yields the recurrence

\[
G_k^*(x_k) = \min_{u_k} \left\{ l(x_k, u_k) + G_{k+1}^*(x_{k+1}) \right\}.
\]
in which \( x_{k+1} = f(x_k, u_k) \). Now that the right side of (2.11) depends only on \( x_k \), \( u_k \), and \( G^*_k \), the computation of \( G^*_k \) easily proceeds in \( O(|X||U|) \) time. This computation is called a value iteration. Note that in each value iteration, some states receive an infinite value only because they are not reachable; a \((K - k)\)-step plan from \( x_k \) to \( X_G \) does not exist. This means that there are no actions, \( u_k \in U(x_k) \), that bring \( x_k \) to a state \( x_{k+1} \in X \) from which a \((K - k - 1)\)-step plan exists that terminates in \( X_G \).

Summarizing, the value iterations proceed as follows:

\[
G^*_F \rightarrow G^*_K \rightarrow G^*_K \rightarrow \cdots \rightarrow G^*_k \rightarrow G^*_k \rightarrow \cdots \rightarrow G^*_2 \rightarrow G^*_1^* \tag{2.12}
\]

until finally \( G^*_1 \) is determined after \( O(K|X||U|) \) time. The resulting \( G^*_1 \) may be applied to yield \( G^*_1(x_l) \), the optimal cost to go to the goal from \( x_l \). It also conveniently gives the optimal cost-to-go from any other initial state. This cost is infinity for states from which \( X_G \) cannot be reached in \( K \) stages.

It seems convenient that the cost of the optimal plan can be computed so easily, but how is the actual plan extracted? One possibility is to store the action that satisfied the min in (2.11) from every state, and at every stage. Unfortunately, this requires \( O(K|X|) \) storage, but it can be reduced to \( O(|X|) \) using the tricks to come in Section 2.3.2 for the more general case of variable-length plans.

Example 2.3 (A Five-State Optimal Planning Problem) Figure 2.8 shows a graph representation of a planning problem in which \( X = \{a, c, b, d, e\} \). Suppose that \( K = 4 \), \( x_1 = a \), and \( X_G = \{d\} \). There will hence be four value iterations, which construct \( G^*_1, G^*_5, G^*_5, \) and \( G^*_1, \) once the final-stage cost-to-go, \( G^*_5 \), is given.

The cost-to-go functions are shown in Figure 2.9. Figures 2.10 and 2.11 illustrate the computations. For computing \( G^*_k \), only \( b \) and \( c \) receive finite values because only they can reach \( d \) in one stage. For computing \( G^*_3 \), only the values \( G^*_3(b) = 4 \) and \( G^*_3(c) = 1 \) are important. Only paths that reach \( b \) or \( c \) can possibly lead to \( d \) in stage \( k = 5 \). Note that the minimization in (2.11) always chooses the action that produces the lowest total cost when arriving at a vertex in the next stage.

![Example 2.3 (A Five-State Optimal Planning Problem)](image)

The ideas from Section 2.3.1.1 may be recycled to yield a symmetrically equivalent method that computes optimal cost-to-come functions from the initial stage. Whereas backward value iterations were able to find optimal plans from all initial states simultaneously, forward value iterations can be used to find optimal plans to all states in \( X \). In the backward case, \( X_G \) must be fixed, and in the forward case, \( x_1 \) must be fixed.

The issue of maintaining feasible solutions appears again. In the forward direction, the role of \( I_F \) is not important. It may be applied in the last iteration, or it can be dropped altogether for problems that do not have a predetermined \( X_G \). However, one must force all plans considered by forward value iteration to originate from \( x_1 \). We again have the choice of either making notation that imposes constraints on the action spaces or simply adding a term that forces infeasible plans to have infinite cost. Once again, the latter will be chosen here.

Let \( C^*_k \) denote the optimal cost-to-come from stage 1 to stage \( k \), optimized over all \((k - 1)\)-step plans. To preclude plans that do not start at \( x_1 \), the definition of \( C^*_1 \) is given by

\[
C^*_1(x_1) = l_F(x_1),
\]

in which \( l_F \) is a new function that yields \( l_F(x_I) = 0 \), and \( l_F(x) = \infty \) for all \( x \neq x_I \). Thus, any plans that try to start from a state other than \( x_I \) will immediately receive infinite cost.
The optimal cost-to-come functions computed by forward value iteration.

\[ C^*_k(x_k) = \min_{u_1, \ldots, u_{k-1}} \left\{ I_f(x_1) + \sum_{i=1}^{k-1} l(x_i, u_i) \right\}. \]  

(2.14)

This equation looks the same as (2.5) after substituting \( k = 1 \); however, \( I_f \) is used here instead of \( l_f \). This has the effect of filtering the plans that are considered to include only those that start at \( x_f \). The forward value iterations find optimal plans to any reachable final state from \( x_f \). This behavior is complementary to that of backward value iteration. In that case, \( X_G \) was fixed, and optimal plans from any initial state were found. For forward value iteration, this is reversed.

To express the dynamic-programming recurrence, one further issue remains. Suppose that \( C^*_{k-1} \) is known by induction, and we want to compute \( C^*_k(x_k) \) for a particular \( x_k \). This means that we must start at some state \( x_{k-1} \) and arrive in state \( x_k \) by applying some action. Once again, the backward state transition function \( f \) is used here instead of \( l_f \). This has the effect of filtering the plans that are considered to include only those that start at \( x_f \). The forward value iterations find optimal plans to any reachable final state from \( x_f \). This behavior is complementary to that of backward value iteration. In that case, \( X_G \) was fixed, and optimal plans from any initial state were found. For forward value iteration, this is reversed.

To express the dynamic-programming recurrence, one further issue remains. Suppose that \( C^*_{k-1} \) is known by induction, and we want to compute \( C^*_k(x_k) \) for a particular \( x_k \). This means that we must start at some state \( x_{k-1} \) and arrive in state \( x_k \) by applying some action. Once again, the backward state transition function \( f \) is used here instead of \( l_f \). This has the effect of filtering the plans that are considered to include only those that start at \( x_f \). The forward value iterations find optimal plans to any reachable final state from \( x_f \). This behavior is complementary to that of backward value iteration. In that case, \( X_G \) was fixed, and optimal plans from any initial state were found. For forward value iteration, this is reversed.

To express the dynamic-programming recurrence, one further issue remains. Suppose that \( C^*_{k-1} \) is known by induction, and we want to compute \( C^*_k(x_k) \) for a particular \( x_k \). This means that we must start at some state \( x_{k-1} \) and arrive in state \( x_k \) by applying some action. Once again, the backward state transition function \( f \) is used here instead of \( l_f \). This has the effect of filtering the plans that are considered to include only those that start at \( x_f \). The forward value iterations find optimal plans to any reachable final state from \( x_f \). This behavior is complementary to that of backward value iteration. In that case, \( X_G \) was fixed, and optimal plans from any initial state were found. For forward value iteration, this is reversed.

**Example 2.4 (Forward Value Iteration)** Example 2.3 is revisited for the case of forward value iterations with a fixed plan length of \( K = 4 \). The cost-to-come functions shown in Figure 2.12 are obtained by direct application of (2.16). It will be helpful to refer to Figures 2.10 and 2.11 once again. The first row corresponds to the immediate application of \( l_f \). In the second row, finite values are obtained for \( a \) and \( b \), which are reachable in one stage from \( x_f = a \). The iterations continue until \( k = 5 \), at which point that optimal cost-to-come is determined for every
2.3. DISCRETE OPTIMAL PLANNING

2.3.2 Optimal Plans of Unspecified Lengths

The value-iteration method for fixed-length plans can be generalized nicely to the case in which plans of different lengths are allowed. There will be no bound on the maximal length of a plan; therefore, the current case is truly a generalization of Formulation 2.1 because arbitrarily long plans may be attempted in efforts to reach $X_G$. The model for the general case does not require the specification of $K$ but instead introduces a special action, $u_T$.

Formulation 2.3 (Discrete Optimal Planning)

1. All of the components from Formulation 2.1 are inherited directly: $X, U(x), f, x_I,$ and $X_G$. Also, the notion of stages from Formulation 2.2 is used.

2. Let $L$ denote a stage-additive cost functional, which may be applied to any $K$-step plan, $\pi_K$, to yield

$$L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k) + l_F(x_F). \quad (2.17)$$

In comparison with $L$ from Formulation 2.2, the present expression does not consider $K$ as a predetermined constant. It will now vary, depending on the length of the plan. Thus, the domain of $L$ is much larger.

3. Each $U(x)$ contains the special termination action, $u_T$. If $u_T$ is applied at $x_k$, then the action is repeatedly applied forever, the state remains unchanged, and no more cost accumulates. Thus, for all $i \geq k$, $u_i = u_T, x_i = x_k$, and $l(x_i, u_T) = 0$.

The termination action is the key to allowing plans of different lengths. It will appear throughout this book. Suppose that value iterations are performed up to $K = 5$, and for the problem there exists a two-step solution plan, $(u_1, u_2)$, that arrives in $X_G$ from $x_I$. This plan is equivalent to the five-step plan $(u_1, u_2, u_T, u_T, u_T)$ because the termination action does not change the state, nor does it accumulate cost. The resulting five-step plan reaches $X_G$ and costs the same as $(u_1, u_2)$. With this simple extension, the forward and backward value-iteration methods of Section 2.3.1 may be applied for any fixed $K$ to optimize over all plans of length $K$ or less (instead of fixing $K$).

The next step is to remove the dependency on $K$. Consider running backward value iterations indefinitely. At some point, $G^*_1$ will be computed, but there is no reason why the process cannot be continued onward to $G^*_6$, $G^*_{-1}$, and so on. Recall that $x_1$ is not utilized in the backward value-iteration method; therefore, there is no concern regarding the starting initial state of the plans. Suppose that backward value iteration was applied for $K = 16$ and was executed down to $G^*_{-8}$. This considers all plans of length 25 or less. Note that it is harmless to add 9 to all stage indices to shift all of the cost-to-go functions. Instead of running from $G^*_{-8}$ to $G^*_{16}$, they can run from $G^*_1$ to $G^*_25$ without affecting their values. The index shifting is allowed because none of the costs depend on the particular index that is given to the stage. The only important aspect of the value iterations is that they proceed backward and consecutively from stage to stage.

Eventually, enough iterations will have been executed so that an optimal plan is known from every state that can reach $X_G$. From that stage, say $k$, onward, the cost-to-go values from one value iteration to the next will be stationary, meaning that for all $i \leq k$, $G^*_{i-1} = G^*_i(x)$ for all $x \in X$. Once the stationary condition is reached, the cost-to-go function no longer depends on a particular stage $k$. In this case, the stage index may be dropped, and the recurrence becomes

$$G^*(x) = \min_u \left\{ l(x, u) + G^*(f(x, u)) \right\}. \quad (2.18)$$

Are there any conditions under which backward value iterations could be executed forever, with each iteration producing a cost-to-go function for which some values are different from the previous iteration? If $l(x, u)$ is nonnegative for all $x \in X$ and $u \in U(x)$, then this could never happen. It could certainly be true that, for any fixed $K$, longer plans will exist, but this cannot be said of optimal plans. From every $x \in X$, there either exists a plan that reaches $X_G$ with finite cost or there is no solution. For each state from which there exists a plan that reaches $X_G$, consider the number of stages in the optimal plan. Consider the maximum number of stages taken from all states that can reach $X_G$. This serves as an upper bound on the number of value iterations before the cost-to-go becomes stationary. Any further iterations will just consider solutions that are worse than the ones already considered (some may be equivalent due to the termination action and shifting of stages). Some trouble might occur if $l(x, u)$ contains negative values. If the state transition graph contains a cycle for which total cost is negative, then it is preferable to execute a plan that travels around the cycle forever, thereby reducing the total cost to $-\infty$. Therefore, we will assume that the cost functional is defined in a sensible way so that negative cycles do not exist. Otherwise, the optimization model itself appears flawed. Some negative values for $l(x, u)$, however, are allowed as long as there are no negative cycles. (It is straightforward to detect and report negative cycles before running the value iterations.)

Since the particular stage index is unimportant, let $k = 0$ be the index of the final stage, which is the stage at which the backward value iterations begin. Hence, $G^*_0$ is the final stage cost, which is obtained directly from $l_F$. Let $-K$ denote the stage index at which the cost-to-go values all become stationary. At this stage, the optimal cost-to-go function, $G^* : X \to \mathbb{R} \cup \{\infty\}$, is expressed by assigning $G^* = G^*_{-K}$. In other words, the particular stage index no longer matters. The value $G^*(x)$ gives the optimal cost to go from state $x \in X$ to the specific goal.
If the optimal actions are not stored during the value iterations, the optimal cost-to-go, \( G^* \), can be used to efficiently recover them. Consider starting from some \( x \in X \). What is the optimal next action? This is given by

\[
u^* = \arg\min_{u \in U(x)} \left\{ l(x, u) + G^*(f(x, u)) \right\}, \tag{2.19}\]

in which \( \arg\min \) denotes the argument that achieves the minimum value of the expression. The action minimizes an expression that is very similar to (2.11). The only differences between (2.19) and (2.11) are that the stage indices are dropped in (2.19) because the cost-to-go values no longer depend on them, and \( \arg\min \) is used so that \( u^* \) is selected. After applying \( u^* \), the state transition equation is used to obtain \( x' = f(x, u^*) \), and (2.19) may be applied again on \( x' \). This process continues until a state in \( X_G \) is reached. This procedure is based directly on the dynamic programming recurrence; therefore, it recovers the optimal plan. The function \( G^* \) serves as a kind of guide that leads the system from any initial state into the goal set optimally. This can be considered as a special case of a navigation function, which will be covered in Section 8.2.2.

As in the case of fixed-length plans, the direction of the value iterations can be reversed to obtain a forward value-iteration method for the variable-length planning problem. In this case, the backward state transition equation, \( f^{-1} \), is used once again. Also, the initial cost term \( l_I \) is used instead of \( l_F \), as in (2.14). The forward value-iteration method starts at \( k = 1 \), and then iterates until the cost-to-come becomes stationary. Once again, the termination action, \( T_F \), preserves the cost of plans that arrived at a state in earlier iterations. Note that it is not required to specify \( X_G \). A counterpart to \( G^* \) may be obtained, from which optimal actions can be recovered. When the cost-to-come values become stationary, an optimal cost-to-come function, \( C^* : X \to \mathbb{R} \cup \{\infty\} \), may be expressed by assigning \( C^* = C_F^* \), in which \( F \) is the final stage reached when the algorithm terminates. The value \( C^*(x) \) gives the cost of an optimal plan that starts from \( x_j \) and reaches \( x \). The optimal action sequence for any specified goal \( x_G \in X \) can be obtained using

\[
\arg\min_{u' \in U^{-1}} \left\{ C^*(f^{-1}(x, u^{-1})) + l(f^{-1}(x, u^{-1}), u') \right\}, \tag{2.20}
\]

which is the forward counterpart of (2.19). The \( u' \) is the action in \( U(f^{-1}(x, u^{-1})) \) that yields \( x \) when the state transition function, \( f \), is applied. The iterations proceed backward from \( x_G \) and terminate when \( x_f \) is reached.

**Example 2.5 (Value Iteration for Variable-Length Plans)** Once again, Example 2.3 is revisited; however, this time the plan length is not fixed due to the termination action. Its effect is depicted in Figure 2.13 by the superposition of new edges that have zero cost. It might appear at first that there is no incentive to choose nontermination actions, but remember that any plan that does not terminate in state \( x_G = d \) will receive infinite cost.

---

**Figure 2.13:** Compare this figure to Figure 2.11, for which \( K \) was fixed at 4. The effect of the termination action is depicted as dashed-line edges that yield 0 cost when traversed. This enables plans of all finite lengths to be considered. Also, the stages extend indefinitely to the left (for the case of backward value iteration).

**Figure 2.14:** The optimal cost-to-go functions computed by backward value iteration applied in the case of variable-length plans.

<table>
<thead>
<tr>
<th></th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
<th>( d )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( G_0 )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>0</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( G_{-1}^* )</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( G_{-2}^* )</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( G_{-3}^* )</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( G_{-4}^* )</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>
### 2.3. DISCRETE OPTIMAL PLANNING

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0^*$</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$C_1^*$</td>
<td>$\infty$</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$C_2^*$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$C_3^*$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$C^*$</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2.15: The optimal cost-to-come functions computed by forward value iteration applied in the case of variable-length plans.

See Figure 2.14. After a few backward value iterations, the cost-to-go values become stationary. After this point, the termination action is being applied from all reachable states and no further cost accumulates. The final cost-to-go function is defined to be $G^*$. Since $d$ is not reachable from $e$, $G^*(e) = \infty$.

As an example of using (2.19) to recover optimal actions, consider starting from state $a$. The action that leads to $b$ is chosen next because the total cost $2 + G^*(b) = 4$ is better than $2 + G^*(a) = 6$ (the 2 comes from the action cost). From state $b$, the optimal action leads to $c$, which produces total cost $1 + G^*(c) = 1$. Similarly, the next action leads to $d \in X_G$, which terminates the plan.

Using forward value iteration, suppose that $x_I = b$. The following cost-to-come functions shown in Figure 2.15 are obtained. For any finite value that remains constant from one iteration to the next, the termination action was applied. Note that the last value iteration is useless in this example. Once $C_I^*$ is computed, the optimal cost-to-come to every possible state from $x_I$ is determined, and future cost-to-come functions are identical. Therefore, the final cost-to-come is renamed $C^*$.

### 2.3.3 Dijkstra Revisited

So far two different kinds of dynamic programming have been covered. The value-iteration method of Section 2.3.2 involves repeated computations over the entire state space. Dijkstra’s algorithm from Section 2.2.2 flows only once through the state space, but with the additional overhead of maintaining which states are alive.

Dijkstra’s algorithm can be derived by focusing on the forward value iterations, as in Example 2.5, and identifying exactly where the “interesting” changes occur. Recall that for Dijkstra’s algorithm, it was assumed that all costs are nonnegative. For any states that are not reachable, their values remain at infinity. They are precisely the unvisited states. States for which the optimal cost-to-come has already become stationary are dead. For the remaining states, an initial cost is obtained, but this cost may be lowered one or more times until the optimal cost is obtained. All states for which the cost is finite, but possibly not optimal, are in the queue, $Q$.

After understanding value iteration, it is easier to understand why Dijkstra’s form of dynamic programming correctly computes optimal solutions. It is clear that the unvisited states will remain at infinity in both algorithms because no plan has reached them. It is helpful to consider the forward value iterations in Example 2.5 for comparison. In a sense, Dijkstra’s algorithm is very much like the value iteration, except that it efficiently maintains the set of states within which cost-to-go values can change. It correctly inserts any states that are reached for the first time, changing their cost-to-come from infinity to a finite value. The values are changed in the same manner as in the value iterations. At the end of both algorithms, the resulting values correspond to the stationary, optimal cost-to-come, $C^*$.

If Dijkstra’s algorithm seems so clever, then why have we spent time covering the value-iteration method? For some problems it may become too expensive to maintain the sorted queue, and value iteration could provide a more efficient alternative. A more important reason is that value iteration extends easily to a much broader class of problems. Examples include optimal planning over continuous state spaces (Sections 8.5.2 and 14.5), stochastic optimal planning (Section 10.2), and computing dynamic game equilibria (Section 10.5). In some cases, it is still possible to obtain a Dijkstra-like algorithm by focusing the computation on the “interesting” region; however, as the model becomes more complicated, it may be inefficient or impossible in practice to maintain this region. Therefore, it is important to have a good understanding of both algorithms to determine which is most appropriate for a given problem.

Dijkstra’s algorithm belongs to a broader family of label-correcting algorithms, which all produce optimal plans by making small modifications to the general algorithm...
forward-search algorithm in Figure 2.4. Figure 2.16 shows the resulting algorithm. The main difference is to allow states to become alive again if a better cost-to-come is found. This enables other cost-to-come values to be improved accordingly. This is not important for Dijkstra’s algorithm and $A^*$ search because they only need to visit each state once. Thus, the algorithms in Figures 2.4 and 2.16 are essentially the same in this case. However, the label-correcting algorithm produces optimal solutions for any sorting of $Q$, including FIFO (breadth first) and LIFO (depth first), as long as $X$ is finite. If $X$ is not finite, then the issue of systematic search dominates because one must guarantee that states are revisited sufficiently many times to guarantee that optimal solutions will eventually be found.

Another important difference between label-correcting algorithms and the standard forward-search model is that the label-correcting approach uses the cost at the goal state to prune away many candidate paths; this is shown in line 7. Thus, it is only formulated to work for a single goal state; it can be adapted to work for multiple goal states, but performance degrades. The motivation for including $C(x_G)$ in line 7 is that there is no need to worry about improving costs at some state, $x'$, if its new cost-to-come would be higher than $C(x_G)$; there is no way it could be along a path that improves the cost to go to $x_G$. Similarly, $x_G$ is not inserted in line 10 because there is no need to consider plans that have $x_G$ as an intermediate state. To recover the plan, either pointers can be stored from $x$ to $x'$ each time an update is made in line 7, or the final, optimal cost-to-come, $C^*$, can be used to recover the actions using (2.20).

2.4 Using Logic to Formulate Discrete Planning

For many discrete planning problems that we would like a computer to solve, the state space is enormous (e.g., $10^{100}$ states). Therefore, substantial effort has been invested in constructing implicit encodings of problems in hopes that the entire state space does not have to be explored by the algorithm to solve the problem. This will be a recurring theme throughout this book; therefore, it is important to pay close attention to representations. Many planning problems can appear trivial once everything has been explicitly given.

Logic-based representations have been popular for constructing such implicit representations of discrete planning. One historical reason is that such representations were the basis of the majority of artificial intelligence research during the 1950s–1980s. Another reason is that they have been useful for representing certain kinds of planning problems very compactly. It may be helpful to think of these representations as compression schemes. A string such as 010101010101... may compress very nicely, but it is impossible to substantially compress a random string of bits. Similar principles are true for discrete planning. Some problems contain a kind of regularity that enables them to be expressed compactly, whereas for others it may be impossible to find such representations. This is why there has been a variety of representation logics proposed through decades of planning research.

Another reason for using logic-based representations is that many discrete planning algorithms are implemented in large software systems. At some point, when these systems solve a problem, they must provide the complete plan to a user, who may not care about the internals of planning. Logic-based representations have seemed convenient for producing output that logically explains the steps involved to arrive at some goal. Other possibilities may exist, but logic has been a first choice due to its historical popularity.

In spite of these advantages, one shortcoming with logic-based representations is that they are difficult to generalize. It is important in many applications to enable concepts such as continuous spaces, unpredictability, sensing uncertainty, and multiple decision makers to be incorporated into planning. This is the main reason why the state-space representation has been used so far: It will be easy to extend and adapt to the problems covered throughout this book. Nevertheless, it is important to study logic-based representations to understand the relationship between the vast majority of discrete planning research and other problems considered in this book, such as motion planning and planning under differential constraints. There are many recurring themes throughout these different kinds of problems, even though historically they have been investigated by separate research communities. Understanding these connections well provides powerful insights into planning issues across all of these areas.

2.4.1 A STRIPS-Like Representation

STRIPS-like representations have been the most common logic-based representations for discrete planning problems. This refers to the STRIPS system, which is considered one of the first planning algorithms and representations [19]; its name is derived from the STanford Research Institute Problem Solver. The original representation used first-order logic, which had great expressive power but many technical difficulties. Therefore, the representation was later restricted to only propositional logic [46], which is similar to the form introduced in this section. There are many variations of STRIPS-like representations. Here is one formulation:

Formulation 2.4 (STRIPS-Like Planning)

1. A finite, nonempty set $I$ of instances.

2. A finite, nonempty set $P$ of predicates, which are binary-valued (partial) functions of one of more instances. Each application of a predicate to a specific set of instances is called a positive literal. A logically negated positive literal is called a negative literal.

3. A finite, nonempty set $O$ of operators, each of which has: 1) preconditions, which are positive or negative literals that must hold for the operator to
2.4. USING LOGIC TO FORMULATE DISCRETE PLANNING

apply, and 2) effects, which are positive or negative literals that are the result of applying the operator.

4. An initial set $S$ which is expressed as a set of positive literals. Negative literals are implied. For any positive literal that does not appear in $S$, its corresponding negative literal is assumed to hold initially.

5. A goal set $G$ which is expressed as a set of both positive and negative literals.

Formulation 2.4.1 provides a definition of discrete feasible planning expressed in a STRIPS-like representation. The three most important components are the sets of instances $I$, predicates $P$, and operators $O$. Informally, the instances characterize the complete set of distinct things that exist in the world. They could, for example, be books, cars, trees, and so on. The predicates correspond to basic properties or statements that can be formed regarding the instances. For example, a predicate called $\text{Under}$ might be used to indicate things like $\text{Under}(\text{Book, Table})$ (the book is under the table) or $\text{Under}(\text{Dirt, Rug})$. A predicate can be interpreted as a kind of function that yields TRUE or FALSE values; however, it is important to note that it is only a partial function because it might not be desirable to allow any instance to be inserted as an argument to the predicate.

If a predicate is evaluated on an instance, for example, $\text{Under}(\text{Dirt, Rug})$, the expression is called a positive literal. The set of all possible positive literals can be formed by applying all possible instances to the domains over which the predicates are defined. Every positive literal has a corresponding negative literal, which is formed by negating the positive literal. For example, $\neg\text{Under}(\text{Dirt, Rug})$ is the negative literal that corresponds to the positive literal $\text{Under}(\text{Dirt, Rug})$, and $\neg$ denotes negation. Let a complementary pair refer to a positive literal together with its counterpart negative literal. The various components of the planning problem are expressed in terms of positive and negative literals.

The role of an operator is to change the world. To be applicable, a set of preconditions must all be satisfied. Each element of this set is a positive or negative literal that must hold TRUE for the operator to be applicable. Any complementary pairs that can be formed from the predicates, but are not mentioned in the preconditions, may assume any value without affecting the applicability of the operator. If the operator is applied, then the world is updated in a manner precisely specified by the set of effects, which indicates positive and negative literals that result from the application of the operator. It is assumed that the truth values of all unmentioned complementary pairs are not affected.

Multiple operators are often defined in a single statement by using variables. For example, $\text{Insert}(i)$ may allow any instance $i \in I$ to be inserted. In some cases, this dramatically reduces the space required to express the problem.

The planning problem is expressed in terms of an initial set $S$ of positive literals and a goal set $G$ of positive and negative literals. A state can be defined by selecting either the positive or negative literal for every possible complementary pair. The initial set $S$ specifies such a state by giving the positive literals only.

For all possible positive literals that do not appear in $S$, it is assumed that their negative counterparts hold in the initial state. The goal set $G$ actually refers to a set of states because, for any unmentioned complementary pair, the positive or negative literal may be chosen, and the goal is still achieved. The task is to find a sequence of operators that when applied in succession will transform the world from the initial state into one in which all literals of $G$ are TRUE. For each operator, the preconditions must also be satisfied before it can be applied. The following example illustrates Formulation 2.4.

**Example 2.6 (Putting Batteries into a Flashlight)** Imagine a planning problem that involves putting two batteries into a flashlight, as shown in Figure 2.17. The set of instances are

$$I = \{\text{Battery}_1, \text{Battery}_2, \text{Cap, Flashlight}\}.$$  \hspace{1cm} (2.21)

Two different predicates will be defined, $\text{On}$ and $\text{In}$, each of which is a partial function on $I$. The predicate $\text{On}$ may only be applied to evaluate whether the $\text{Cap}$ is $\text{On}$ the $\text{Flashlight}$ and is written as $\text{On}(\text{Cap, Flashlight})$. The predicate $\text{In}$ may be applied in the following two ways: $\text{In}(\text{Battery}_1, \text{Flashlight})$, $\text{In}(\text{Battery}_2, \text{Flashlight})$, to indicate whether either battery is in the flashlight. Recall that predicates are only partial functions in general. For the predicate $\text{In}$, it is not desirable to apply any instance to any argument. For example, it is meaningless to define $\text{In}(\text{Battery}_1, \text{Battery}_1)$ and $\text{In}(\text{Flashlight}, \text{Battery}_2)$ (they could be included in the model, always retaining a negative value, but it is inefficient).

The initial set is

$$S = \{\text{On}(\text{Cap, Flashlight})\}.$$  \hspace{1cm} (2.22)
2.4. USING LOGIC TO FORMULATE DISCRETE PLANNING

S. M. LaValle: Planning Algorithms

63

2.4. USING LOGIC TO FORMULATE DISCRETE PLANNING

64

Name Preconditions Effects

PlaceCap \{¬On(Cap, Flashlight)\} \{On(Cap, Flashlight)\}

RemoveCap \{On(Cap, Flashlight)\} \{¬On(Cap, Flashlight)\}

Insert(i) \{¬On(Cap, Flashlight), ¬In(i, Flashlight)\} \{In(i, Flashlight)\}

Figure 2.18: Three operators for the flashlight problem. Note that an operator can be expressed with variable argument(s) for which different instances could be substituted.

Based on $S$, both $¬In(Battery_1, Flashlight)$ and $¬In(Battery_2, Flashlight)$ are assumed to hold. Thus, $S$ indicates that the cap is on the flashlight, but the batteries are outside.

The goal state is

$$G = \{On(Cap, Flashlight), In(Battery_1, Flashlight), In(Battery_2, Flashlight)\},$$

(2.23)

which means that both batteries must be in the flashlight, and the cap must be on.

The set $O$ consists of the four operators, which are shown in Figure 2.18. Here is a plan that reaches the goal state in the smallest number of steps:

$$(\text{RemoveCap, Insert(Battery}_1\text{), Insert(Battery}_2\text{), PlaceCap}).$$

(2.24)

In words, the plan simply says to take the cap off, put the batteries in, and place the cap back on.

This example appears quite simple, and one would expect a planning algorithm to easily find such a solution. It can be made more challenging by adding many more instances to $I$, such as more batteries, more flashlights, and a bunch of objects that are irrelevant to achieving the goal. Also, many other predicates and operators can be added so that the different combinations of operators become overwhelming.

A large number of complexity results exist for planning expressed using logic. The graph search problem is solved efficiently in polynomial time; however, a state transition graph is not given as the input. An input that is expressed using Formulation 2.4 may describe an enormous state transition graph using very few instances, predicates, and operators. In a sense, the model is highly compressed when using some logic-based formulations. This brings it closer to the Kolmogorov complexity [14, 42] of the state transition graph, which is the shortest bit size to which it can possibly be compressed and then fully recovered by a Turing machine. This has the effect of making the planning problem appear more difficult. Concise inputs may encode very challenging planning problems. Most of the known hardness results are surveyed in Chapter 3 of [23]. Under most formulations, logic-based planning is NP-hard. The particular level of hardness (NP, PSPACE, EXPTIME, etc.) depends on the precise problem conditions. For example, the complexity depends on whether the operators are fixed in advance or included in the input. The latter case is much harder. Separate complexities are also obtained based on whether negative literals are allowed in the operator effects and also whether they are allowed in preconditions. The problem is generally harder if both positive and negative literals are allowed in these cases.

2.4.2 Converting to the State-Space Representation

It is useful to characterize the relationship between Formulation 2.4 and the original formulation of discrete feasible planning, Formulation 2.1. One benefit is that it immediately shows how to adapt the search methods of Section 2.2 to work for logic-based representations. It is also helpful to understand the relationships between the algorithmic complexities of the two representations.

Up to now, the notion of “state” has been only vaguely mentioned in the context of the STRIPS-like representation. Now consider making this more concrete. Suppose that every predicate has $k$ arguments, and any instance could appear in each argument. This means that there are $|P|^I$ $k$ complementary pairs, which corresponds to all of the ways to substitute instances into all arguments of all predicates. To express the state, a positive or negative literal must be selected from every complementary pair. For convenience, this selection can be encoded as a binary string by imposing a linear ordering on the instances and predicates. Using Example 2.6, the state might be specified in order as

$$(On(Cap, Flashlight), ¬In(Battery_1, Flashlight_1), In(Battery_2, Flashlight_2)).$$

(2.25)

Using a binary string, each element can be “0” to denote a negative literal or “1” to denote positive literal. The encoded state is $x = 101$ for (2.25). If any instance can appear in the argument of any predicate, then the length of the string is $|P||I|^k$. The total number of possible states of the world that could possibly be distinguished corresponds to the set of all possible bit strings. This set has size

$$2^{|P||I|^k}.$$

(2.26)

The implication is that with a very small number of instances and predicates, an enormous state space can be generated. Even though the search algorithms of Section 2.2 may appear efficient with respect to the size of the search graph (or the number of states), the algorithms appear horribly inefficient with respect to the sizes of $P$ and $I$. This has motivated substantial efforts on the development of techniques to help guide the search by exploiting the structure of specific representations. This is the subject of Section 2.5.

The next step in converting to a state-space representation is to encode the initial state $x_I$ as a string. The goal set, $X_G$, is the set of all strings that are
consistent with the positive and negative goal literals. This can be compressed by extending the string alphabet to include a “don’t care” symbol, \( \delta \). A single string that has a “0” for each negative literal, a “1” for each positive literal, and a “\( \delta \)” for all others would suffice in representing any \( X_G \) that is expressed with positive and negative literals.

Now convert the operators. For each state, \( x \in X \), the set \( U(x) \) represents the set of operators with preconditions that are satisfied by \( x \). To apply the search techniques of Section 2.2, note that it is not necessary to determine \( U(x) \) explicitly in advance for all \( x \in X \). Instead, \( U(x) \) can be computed whenever each \( x \) is encountered for the first time in the search. The effects of the operator are encoded by the state transition equation. From a given \( x \in X \), the next state, \( f(x, u) \), is obtained by flipping the bits as prescribed by the effects part of the operator.

All of the components of Formulation 2.1 have been derived from the components of Formulation 2.4. Adapting the search techniques of Section 2.2 is straightforward. It is also straightforward to extend Formulation 2.4 to represent optimal planning. A cost can be associated with each operator and set of literals that capture the current state. This would express \( l(x, u) \) of the cost functional, \( L \), from Section 2.3. Thus, it is even possible to adapt the value-iteration method to work under the logic-based representation, yielding optimal plans.

### 2.5 Logic-Based Planning Methods

A huge body of research has been developed over the last few decades for planning using logic-based representations [23, 53]. These methods usually exploit some structure that is particular to the representation. Furthermore, numerous heuristics for accelerating performance have been developed from implementation studies. The main ideas behind some of the most influential approaches are described in this section, but without presenting particular heuristics.

Rather than survey all logic-based planning methods, this section focuses on some of the main approaches that exploit logic-based representations. Keep in mind that the searching methods of Section 2.2 also apply. Once a problem is given using Formulation 2.4, the state transition graph is incrementally revealed during the search. In practice, the search graph may be huge relative to the size of the problem description. One early attempt to reduce the size of this graph was the STRIPS planning algorithm [19, 46]; it dramatically reduced the branching factor but unfortunately was not complete. The methods presented in this section represent other attempts to reduce search complexity in practice while maintaining completeness. For each method, there are some applications in which the method may be more efficient, and others for which performance may be worse. Thus, there is no clear choice of method that is independent of its particular use.

### 2.5.1 Searching in a Space of Partial Plans

One alternative to searching directly in \( X \) is to construct partial plans without reference to particular states. By using the operator representation, partial plans can be incrementally constructed. The idea is to iteratively achieve required subgoals in a partial plan while ensuring that no conflicts arise that could destroy the solution developed so far.

A partial plan \( \sigma \) is defined as

1. A set \( O_\sigma \) of operators that need to be applied. If the operators contain variables, these may be filled in by specific values or left as variables. The same operator may appear multiple times in \( O_\sigma \), possibly with different values for the variables.

2. A partial ordering relation \( \preceq_\sigma \) on \( O_\sigma \), which indicates for some pairs \( o_1, o_2 \in O_\sigma \) that one must appear before other: \( o_1 \preceq_\sigma o_2 \).

3. A set \( B_\sigma \) of binding constraints, in which each indicates that some variables across operators must take on the same value.

4. A set \( C_\sigma \) of causal links, in which each is of the form \( (o_1, l, o_2) \) and indicates that \( o_1 \) achieves the literal \( l \) for the purpose of satisfying a precondition of \( o_2 \).

#### Example 2.7 (A Partial Plan)

Each partial plan encodes a set of possible plans. Recall the model from Example 2.6. Suppose

\[
O_\sigma = \{\text{RemoveCap}, \text{Insert}(\text{Battery1})\}.
\]

A sensible ordering constraint is that

\[
\text{RemoveCap} \prec_\sigma \text{Insert}(\text{Battery1}).
\]

A causal link,

\[
(\text{RemoveCap}, \neg \text{On}(\text{Cap, Flashlight}), \text{Insert}(\text{Battery1}))
\]

indicates that the RemoveCap operator achieves the literal \( \neg \text{On}(\text{Cap, Flashlight}) \), which is a precondition of \( \text{Insert}(\text{Battery1}) \). There are no binding constraints for this example. The partial plan implicitly represents the set of all plans for which RemoveCap appears before \( \text{Insert}(\text{Battery1}) \), under the constraint that the causal link is not violated.

Several algorithms have been developed to search in the space of partial plans. To obtain some intuition about the partial-plan approach, a planning algorithm is described in Figure 2.19. A vertex in the partial-plan search graph is a partial
2.5. LOGIC-BASED PLANNING METHODS

PLAN-SPACE PLANNING

1. Start with any initial partial plan, \( \sigma \).
2. Find a flaw in \( \sigma \), which may be 1) an operator precondition that has not been achieved, or 2) an operator in \( O_\sigma \) that threatens a causal constraint in \( C_\sigma \).
3. If there is no flaw, then report that \( \sigma \) is a complete solution and compute a linear ordering of \( O_\sigma \) that satisfies all constraints.
4. If the flaw is an unachieved precondition, \( l \), for some operator \( o_2 \), then find an operator, \( o_1 \), that achieves it and record a new causal constraint, \((o_1,l,o_2)\).
5. If the flaw is a threat on a causal link, then the threat must be removed by updating \( \prec_\sigma \) to induce an appropriate operator ordering, or by updating \( B_\sigma \) to bind the operators in a way that resolves the threat.
6. Return to Step 2.

Figure 2.19: Planning in the plan space is achieved by iteratively finding a flaw in the plan and fixing it.

plan, and an edge is constructed by extending one partial plan to obtain another partial plan that is closer to completion. Although the general template is simple, the algorithm performance depends critically on the choice of initial plan and the particular flaw that is resolved in each iteration. One straightforward generalization is to develop multiple partial plans and decide which one to refine in each iteration.

In early works, methods based on partial plans seemed to offer substantial benefits; however, they are currently considered to be not “competitive enough” in comparison to methods that search the state space [23]. One problem is that it becomes more difficult to develop application-specific heuristics without explicit references to states. Also, the vertices in the partial-plan search graph are costly to maintain and manipulate in comparison to ordinary states.

2.5.2 Building a Planning Graph

Blum and Furst introduced the notion of a planning graph, which is a powerful data structure that encodes information about which states may be reachable [4]. For the logic-based problem expressed in Formulation 2.4, consider performing reachability analysis. Breadth-first search can be used from the initial state to expand the state transition graph. In terms of the input representation, the resulting graph may be of exponential size in the number of stages. This gives precise reachability information and is guaranteed to find the goal state.

The idea of Blum and Furst is to construct a graph that is much smaller than the state transition graph and instead contains only partial information about reachability. The resulting planning graph is polynomial in size and can be efficiently constructed for some challenging problems. The trade-off is that the planning graph indicates states that can possibly be reached. The true reachable set is overapproximated, by eliminating many impossible states from consideration. This enables quick elimination of impossible alternatives in the search process. Planning algorithms have been developed that extract a plan from the planning graph. In the worst case, this may take exponential time, which is not surprising because the problem in Formulation 2.4 is NP-hard in general. Nevertheless, dramatic performance improvements were obtained on some well-known planning benchmarks. Another way to use the planning graph is as a source of information for developing search heuristics for a particular problem.

Planning graph definition A layered graph is a graph that has its vertices partitioned into a sequence of layers, and its edges are only permitted to connect vertices between successive layers. The planning graph is a layered graph in which the layers of vertices form an alternating sequence of literals and operators:

\[
(L_1, O_1, L_2, O_2, L_3, O_3, \ldots, L_k, O_k, L_{k+1}).
\]

The edges are defined as follows. To each operator \( o_i \in O_i \), a directed edge is made from each \( l_i \in L_i \) that is a precondition of \( o_i \). To each literal \( l_i \in L_i \), an edge is made from each operator \( o_{i+1} \in O_{i+1} \) that has \( l_i \) as an effect.

One important requirement is that no variables are allowed in the operators. Any operator from Formulation 2.4 that contains variables must be converted into a set that contains a distinct copy of the operator for every possible substitution of values for the variables.

Layer-by-layer construction The planning graph is constructed layer by layer, starting from \( L_1 \). In the first stage, \( L_1 \) represents the initial state. Every positive literal in \( S \) is placed into \( L_1 \), along with the negation of every positive literal not in \( S \). Now consider stage \( i \). The set \( O_i \) is the set of all operators for which their preconditions are a subset of \( L_i \). The set \( L_{i+1} \) is the union of the effects of all operators in \( O_i \). The iterations continue until the planning graph stabilizes, which means that \( O_{i+1} = O_i \) and \( L_{i+1} = L_i \). This situation is very similar to the stabilization of value iterations in Section 2.3.2. A trick similar to the termination action, \( u_\tau \), is needed even here so that plans of various lengths are properly handled. In Section 2.3.2, one job of the termination action was to prevent state transitions from occurring. The same idea is needed here. For each possible literal, \( l \), a trivial operator is constructed for which \( l \) is the only precondition and effect. The introduction of trivial operators ensures that once a literal is reached, it is maintained in the planning graph for every subsequent layer of literals. Thus, each \( O_i \) may contain some trivial operators, in addition to operators from the
mutually exclusive. Both cannot be chosen simultaneously without leading to some kind of conflict. A pair in conflict is called mutex. For each layer, a mutex relation is defined that indicates which pairs satisfy the mutex condition. A pair, \( o, o' \) of operators is defined to be mutex if any of these conditions is met:

1. **Inconsistent effects**: An effect of \( o \) is the negated literal of an effect of \( o' \).

2. **Interference**: An effect of \( o \) is the negated literal of a precondition of \( o' \).

3. **Competing needs**: A pair of preconditions, one from each of \( o \) and \( o' \), are mutex in \( L_i \).

The last condition relies on the definition of mutex for literals, which is presented next. Any pair, \( l, l' \) of literals is defined to be mutex if at least one of the two conditions is met:

1. **Negated literals**: \( l \) and \( l' \) form a complementary pair.

2. **Inconsistent support**: Every pair of operators, \( o, o' \in O_{i-1} \), that achieve \( l \) and \( l' \) is mutex. In this case, one operator must achieve \( l \), and the other must achieve \( l' \). If there exists an operator that achieves both, then this condition is false, regardless of the other pairs of operators.

The mutex definition depends on the layers; therefore, it is computed layer by layer during the planning graph construction.

**Example 2.8 (The Planning Graph for the Flashlight)** Figure 2.20 shows the planning graph for Example 2.6. In the first layer, \( L_1 \) expresses the initial state. The only applicable operator is \( \text{RemoveCap} \). The operator layer \( O_1 \) contains \( \text{RemoveCap} \) and three trivial operators, which are needed to maintain the literals from \( L_1 \). The appearance of \( \neg \text{On}(\text{Cap, Flashlight}) \) enables the battery-insertion operator to apply. Since variables are not allowed in operator definitions in a planning graph, two different operators (labeled as \( I_1 \) and \( I_2 \)) appear, one for each battery. Notice the edges drawn to \( I_1 \) and \( I_2 \) from their preconditions. The cap may also be replaced; hence, \( \text{PlaceCap} \) is included in \( O_2 \). At the \( L_3 \) layer, all possible literals have been obtained. At \( O_3 \), all possible operators, including the trivial ones, are included. Finally, \( L_4 = L_3 \), and \( O_4 \) will be the same as \( O_3 \). This implies that the planning graph has stabilized.

**Plan extraction** Suppose that the planning graph has been constructed up to \( L_i \). At this point, the planning graph can be searched for a solution. If no solution is found and the planning graph has stabilized, then no solution exists to the problem in general (this was shown in [4]; see also [23]). If the planning graph has not stabilized, then it can be extended further by adding \( O_i \) and \( L_{i+1} \). The extended graph can then be searched for a solution plan. A planning algorithm derived from the planning graph interleaves the graph extensions and the searches for solutions. Either a solution is reported at some point or the algorithm correctly reports that no solution exists after the planning graph stabilizes. The resulting algorithm is complete. One of the key observations in establishing completeness is that the literal and operator layers each increase monotonically as \( i \) increases. Furthermore, the sets of pairs that are mutex decrease monotonically, until all possible conflicts are resolved.

Rather than obtaining a fully specified plan, the planning graph yields a *layered plan*, which is a special form of partial plan. All of the necessary operators are included, and the layered plan is specified as

\[
(A_1, A_2, \ldots, A_k),
\]
in which each $A_i$ is a set of operators. Within any $A_i$, the operators are nonmutex and may be applied in any order without altering the state obtained by the layered plan. The only constraint is that for each $i$ from 1 to $k$, every operator in $A_i$ must be applied before any operators in $A_{i+1}$ can be applied. For the flashlight example, a layered plan that would be constructed from the planning graph in Figure 2.20 is

$$\{\text{RemoveCap}, \{\text{Insert}(\text{Battery}1), \text{Insert}(\text{Battery}2)\}, \{\text{PlaceCap}\}\}. \quad (2.32)$$

To obtain a fully specified plan, the layered plan needs to be linearized by specifying a linear ordering for the operators that is consistent with the layer constraints. For (2.32), this results in (2.24). The actual plan execution usually involves more stages than the number in the planning graph. For complicated planning problems, this difference is expected to be huge. With a small number of stages, the planning graph can consider very long plans because it can apply several nonmutex operators in a single layer.

At each level, the search for a plan could be quite costly. The idea is to start from $L_i$ and perform a backward and/or search. To even begin the search, the goal literals $G$ must be a subset of $L_i$, and no pairs are allowed to be mutex; otherwise, immediate failure is declared. From each literal $l \in G$, an “or” part of the search tries possible operators that produce $l$ as an effect. The “and” part of the search must achieve all literals in the precondition of an operator chosen at the previous “or” level. Each of these preconditions must be achieved, which leads to another “or” level in the search. The idea is applied recursively until the initial set $L_1$ of literals is obtained. During the and/or search, the computed mutex relations provide information that immediately eliminates some branches. Frequently, triples and higher order tuples are checked for being mutex together, even though they are not pairwise mutex. A hash table is constructed to efficiently retrieve this information as it is considered multiple times in the search. Although the plan extraction is quite costly, superior performance was shown in [4] on several important benchmarks. In the worst case, the search could require exponential time (otherwise, a polynomial-time algorithm would have been found to an NP-hard problem).

### 2.5.3 Planning as Satisfiability

Another interesting approach is to convert the planning problem into an enormous Boolean satisfiability problem. This means that the planning problem of Formulation 2.4 can be solved by determining whether some assignment of variables is possible for a Boolean expression that leads to a TRUE value. Generic methods for determining satisfiability can be directly applied to the Boolean expression that encodes the planning problem. The **Davis-Putnam procedure** is one of the most widely known algorithms for satisfiability. It performs a depth-first search by iteratively trying assignments for variables and backtracking when assignments fail. During the search, large parts of the expression can be eliminated due to the current assignments. The algorithm is complete and reasonably efficient. Its use in solving planning problems is surveyed in [23]. In practice, stochastic local search methods provide a reasonable alternative to the Davis-Putnam procedure [28].

Suppose a planning problem has been given in terms of Formulation 2.4. All literals and operators will be tagged with a stage index. For example, a literal that appears in two different stages will be considered distinct. This kind of tagging is similar to situation calculus [22]; however, in that case, variables are allowed for the tags. To obtain a finite, Boolean expression the total number of stages must be declared. Let $K$ denote the number of stages at which operators can be applied. As usual, the first stage is $k = 1$ and the final stage is $k = F = K + 1$. Setting a stage limit is a significant drawback of the approach because this is usually not known before the problem is solved. A planning algorithm can assume a small value for $F$ and then gradually increase it each time the resulting Boolean expression is not satisfied. If the problem is not solvable, however, this approach iterates forever.

Let $\lor$ denote logical OR, and let $\land$ denote logical AND. The Boolean expression is written as a conjunction\(^5\) of many terms, which arise from five different sources:

1. **Initial state:** A conjunction of all literals in $S$ is formed, along with the negation of all positive literals not in $S$. These are all tagged with 1, the initial stage index.

2. **Goal state:** A conjunction of all literals in $G$, tagged with the final stage index, $F = K + 1$.

3. **Operator encodings:** Each operator must be copied over the stages. For each $o \in O$, let $o_k$ denote the operator applied at stage $k$. A conjunction is formed over all operators at all stages. For each $o_k$, the expression is

$$\neg o_k \lor (p_1 \land p_2 \land \cdots \land p_m \land e_1 \land e_2 \land \cdots \land e_n), \quad (2.33)$$

in which $p_1, \ldots, p_m$ are the preconditions of $o_k$, and $e_1, \ldots, e_n$ are the effects of $o_k$.

4. **Frame axioms:** The next part is to encode the implicit assumption that every literal that is not an effect of the applied operator remains unchanged in the next stage. This can alternatively be stated as follows: If a literal $l$ becomes negated to $\neg l$, then an operator that includes $\neg l$ as an effect must have been executed. (If $l$ was already a negative literal, then $\neg l$ is a positive literal.) For each stage and literal, an expression is needed. Suppose that $l_k$ and $l_{k+1}$ are the same literal but are tagged for different stages. The expression is

$$\left( l_k \lor \neg l_{k+1} \right) \lor (o_{k,1} \lor o_{k,2} \lor \cdots \lor o_{k,j}), \quad (2.34)$$

\(^5\)Conjunction means logical AND.
in which \( o_{k,1}, \ldots, o_{k,j} \) are the operators, tagged for stage \( k \), that contain \( l_{k+1} \) as an effect. This ensures that if \( \neg l_k \) appears, followed by \( l_{k+1} \), then some operator must have caused the change.

5. **Complete exclusion axiom**: This indicates that only one operator applies at every stage. For every stage \( k \), and any pair of stage-tagged operators \( o_k \) and \( o_k' \), the expression is

\[
\neg o_k \lor \neg o_k'
\]

which is logically equivalent to \( \neg (o_k \land o_k') \) (meaning, “not both at the same stage”).

It is shown in [34] that a solution plan exists if and only if the resulting Boolean expression is satisfiable.

The following example illustrates the construction.

**Example 2.9 (The Flashlight Problem as a Boolean Expression)** A Boolean expression will be constructed for Example 2.6. Each of the expressions given below is joined into one large expression by connecting them with \( \land \)'s.

The expression for the initial state is

\[
O(C, F, 1) \land \neg I(B1, F, 1) \land \neg I(B2, F, 1),
\]

which uses the abbreviated names, and the stage tag has been added as an argument to the predicates. The expression for the goal state is

\[
O(C, F, 5) \land I(B1, F, 5) \land I(B2, F, 5),
\]

which indicates that the goal must be achieved at stage \( k = 5 \). This value was determined because we already know the solution plan from (2.24). The method will also work correctly for a larger value of \( k \). The expressions for the operators are

\[
\neg PC_k \lor (\neg O(C, F, k) \land O(C, F, k + 1))
\]

\[
\neg RC_k \lor (O(C, F, k) \land \neg O(C, F, k + 1))
\]

\[
\neg I1_k \lor (\neg O(C, F, k) \land \neg I(B1, F, k) \land I(B1, F, k + 1))
\]

\[
\neg I2_k \lor (\neg O(C, F, k) \land \neg I(B2, F, k) \land I(B2, F, k + 1))
\]

for each \( k \) from 1 to 4.

The frame axioms yield the expressions

\[
(O(C, F, k) \lor \neg O(C, F, k + 1)) \lor (PC_k)
\]

\[
(\neg O(C, F, k) \lor O(C, F, k + 1)) \lor (RC_k)
\]

\[
(I(B1, F, k) \lor \neg I(B1, F, k + 1)) \lor (I1_k)
\]

\[
(\neg I(B1, F, k) \lor I(B1, F, k + 1))
\]

\[
(I(B2, F, k) \lor \neg I(B2, F, k + 1)) \lor (I2_k)
\]

\[
(\neg I(B2, F, k) \lor I(B2, F, k + 1))
\]

for each \( k \) from 1 to 4. No operators remove batteries from the flashlight. Hence, two of the expressions list no operators.

Finally, the complete exclusion axiom yields the expressions

\[
\neg RC_k \lor \neg PC_k
\]

\[
\neg PC_k \lor \neg O1_k
\]

\[
\neg RC_k \lor \neg O2_k
\]

for each \( k \) from 1 to 4. The full problem is encoded by combining all of the given expressions into an enormous conjunction. The expression is satisfied by assigning true values to \( RC_1, IB_{12}, IB_{23}, \) and \( PC_4 \). An alternative solution is \( RC_1, IB_{22}, IB_{13}, \) and \( PC_4 \). The stage index tags indicate the order that the actions are applied in the recovered plan. ■

**Further Reading**

Most of the ideas and methods in this chapter have been known for decades. Most of the search algorithms of Section 2.2 are covered in algorithms literature as graph search [12, 24, 43, 54] and in AI literature as planning or search methods [37, 46, 47, 49, 53, 61]. Many historical references to search in AI appear in [53]. Bidirectional search was introduced in [50, 51] and is closely related to means-end analysis [45]; more discussion on bidirectional search appears in [8, 7, 31, 40, 53]. The development of good search heuristics is critical to many applications of discrete planning. For substantial material on this topic, see [23, 36, 49]. For the relationship between planning and scheduling, see [15, 23, 57].

The dynamic programming principle forms the basis of optimal control theory and many algorithms in computer science. The main ideas follow from Bellman’s principle of optimality [1, 2]. These classic works led directly to the value-iteration methods of Section 2.3. For more recent material on this topic, see [3], which includes Dijkstra’s algorithm and its generalization to label-correcting algorithms. An important special version of Dijkstra’s algorithm is Dial’s algorithm [17] (see [59] and Section 8.2.3). Throughout this book, there are close connections between planning methods and control theory. One step in this direction was taken earlier in [16].

The foundations of logic-based planning emerged from early work of Nilsson [19, 46], which contains most of the concepts introduced in Section 2.4. Over the last few decades, an enormous body of literature has been developed. Section 2.5 briefly surveyed some of the highlights; however, several more chapters would be needed to do this subject justice. For a comprehensive, recent treatment of logic-based planning, see [23]; topics beyond those covered here include constraint-satisfaction planning, scheduling, and temporal logic. Other sources for logic-based planning include [22, 53, 60, 62]. A critique of benchmarks used for comparisons of logic-based planning algorithms appears in [30].

Too add uncertainty or multiple decision makers to the problems covered in this chapter, jump ahead to Chapter 10 (this may require some background from Chapter 9). To move from searching in discrete to continuous spaces, try Chapters 5 and 6 (some background from Chapters 3 and 4 is required).
2.5 LOGIC-BASED PLANNING METHODS

(a) Develop a backward version of Dijkstra’s algorithm that starts from the goal. Show that it always yields optimal plans.

(b) Describe the relationship between the algorithm from part (a) and the backward value iterations from Section 2.3.2.

(c) Derive a backward version of the $A^*$ algorithm and show that it yields optimal plans.

Exercises

1. Consider the planning problem shown in Figure 2.21. Let $a$ be the initial state, and let $e$ be the goal state.

   (a) Use backward value iteration to determine the stationary cost-to-go.

   (b) Do the same but instead use forward value iteration.

2. Try to construct a worst-case example for best-first search that has properties similar to that shown in Figure 2.5, but instead involves moving in a 2D world with obstacles, as introduced in Example 2.1.

3. It turns out that value iteration can be generalized to a cost functional of the form

   \[ L(\pi_K) = \sum_{k=1}^{K} l(x_k, u_k, x_{k+1}) + l_F(x_F), \]  

   in which $l(x_k, u_k)$ in (2.4) has been replaced by $l(x_k, u_k, x_{k+1})$.

   (a) Show that the dynamic programming principle can be applied in this more general setting to obtain forward and backward value iteration methods that solve the fixed-length optimal planning problem.

   (b) Do the same but for the more general problem of variable-length plans, which uses termination conditions.

4. The cost functional can be generalized to being stage-dependent, which means that the cost might depend on the particular stage $k$ in addition to the state, $x_k$ and the action $u_k$. Extend the forward and backward value iteration methods of Section 2.3.1 to work for this case, and show that they give optimal solutions. Each term of the more general cost functional should be denoted as $l(x_k, u_k, k)$.

5. Recall from Section 2.3.2 the method of defining a termination action $u_T$ to make the value iterations work correctly for variable-length planning. Instead of requiring that one remains at the same state, it is also possible to formulate the problem by creating a special state, called the terminal state, $x_T$. Whenever $u_T$ is applied, the state becomes $x_T$. Describe in detail how to modify the cost functional, state transition equation, and any other necessary components so that the value iterations correctly compute shortest plans.

6. Dijkstra’s algorithm was presented as a kind of forward search in Section 2.2.1.

Figure 2.21: Another five-state discrete planning problem.
18. Using $A^*$ search, the performance degrades substantially when there are many alternative solutions that are all optimal, or at least close to optimal. Implement $A^*$ search and evaluate it on various grid-based problems, based on Example 2.1. Compare the performance for two different cases:

(a) Using $|i' - i| + |j' - j|$ as the heuristic, as suggested in Section 2.2.2.
(b) Using $\sqrt{(i' - i)^2 + (j' - j)^2}$ as the heuristic.

Which heuristic seems superior? Explain your answer.

19. Implement $A^*$, breadth-first, and best-first search for grid-based problems. For each search algorithm, design and demonstrate examples for which one is clearly better than the other two.

20. Experiment with bidirectional search for grid-based planning. Try to understand and explain the trade-off between exploring the state space and the cost of connecting the trees.

21. Try to improve the method used to solve Exercise 18 by detecting when the search might be caught in a local minimum and performing random walks to try to escape. Try using best-first search instead of $A^*$. There is great flexibility in possible approaches. Can you obtain better performance on average for any particular examples?

22. Implement backward value iteration and verify its correctness by reconstructing the costs obtained in Example 2.5. Test the implementation on some complicated examples.

23. For a planning problem under Formulation 2.3, implement both Dijkstra's algorithm and forward value iteration. Verify that these find the same plans. Comment on their differences in performance.

24. Consider grid-based problems for which there are mostly large, open rooms. Attempt to develop a multi-resolution search algorithm that first attempts to take larger steps, and only takes smaller steps as larger steps fail. Implement your ideas, conduct experiments on examples, and refine your approach accordingly.
Bibliography


