A Survey of Machine Learning Approaches to Robotic Path-Planning

Michael Otte
PhD Preliminary Exam (Survey Paper Component)
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430
2008

Forward

To my surprise and delight, as machine learning and path planning have become more mainstream in the last decade, a number of researchers have found this survey paper to be useful. It has even been cited in a number of recent academic papers. I originally wrote this survey paper in 2008 as part of the Preliminary Exam that is required to "start" PhD research in the Department of Computer Science (CS) at the University of Colorado Boulder (CU).

Given its newfound interest, I have decided to make the paper available on my website. I can only assume that prior to this people stumbled across the work with the help of modern search engines – owing in no small part to the fact that papers at the intersection of machine learning and path-planning have been relatively scarce. For many years, the only copy available on the Internet was provided by CU Computer Science Professor and Cognitive Scientist Mike Mozer, who maintained it as an example for other students taking CU's CS Preliminary exam.

Prof. Mozer is doubly responsible for the existence of this work in the academic discourse. For, as a member of my Preliminary Exam Committee, he also refused to accept a much leaner version of the paper in fulfillment of the exam requirements – sending a pugnacious and exasperated graduate student back to refine the work for another couple of months. I am glad to find that others besides Prof. Mozer and myself have found value in that extra work.

Michael Otte

BibTex citation block (in case it is useful)

```
@techreport{Otte.prelim08,
  author = "Michael Otte",
  title = "A Survey of Machine Learning Approaches to Robotic Path-Planning",
  institution = "University of Colorado at Boulder",
  year = "2008",
  type = "Technical Report",
  number = "PhD Preliminary Exam"
}
```

A Survey of Machine Learning Approaches to Robotic Path-Planning

Michael W. Otte
Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430

Abstract

Parameters in robotic systems have traditionally been hand-tuned by human experts through painstaking trail-and-error. In addition to requiring a substantial number of man-hours, hand-tuning usually results in robotic systems that are brittle. That is, they can easily fail in new environments. In the last decade or two, designers have realized that their systems can be made more robust by incorporating concepts developed in the field of machine learning. This paper presents a survey of how machine learning has been applied to robotic path-planning and path-planning related concepts.

This survey is concerned with ideas at the juncture of the two otherwise unrelated fields of machine learning and path-planning. Both of these are mature research domains, complete with large bodies of literature. To fully understand the work at their intersection, one must have a grasp of the basic concepts relevant to both. Therefore, this paper is organized into three chapters. One each for robotics, machine learning, and the applications of machine learning to path-planning. Short descriptions of each chapter are provided in the remainder of this introduction. Readers are encouraged to use these outlines to gauge which chapters will be most useful to them, and to skip reviews of familiar work.

Chapter 1 provides a high-level overview of robotic systems in general. A complete robotic system is outlined to frame how the various subsystems interact with each other. However, the bulk of chapter 1 is dedicated to the representation and planning robotic subsystems, as these are most relevant to the path-planning problem. The last half of this chapter contains an in-depth discussion on path-planning algorithms, with a particular focus on graph-search techniques. Readers with a sufficient grounding in robotics may be able to skip the first half of chapter 1 without compromising their ability to understand the rest of the paper. However, even seasoned roboticists may find the review of graph-search algorithms interesting (starting with Section-1.5), as it covers what I consider to be the major path-planning breakthroughs of the last decade[1].

Chapter 2 provides general background information on machine learning. Short summaries are provided for the subfields of supervised learning and reinforcement learning, and a handful of specific algorithms are also examined in-depth. Note that the content of this chapter has been chosen based on successful applications to path-planning in the last few years, it does not represent an exhaustive survey of all machine learning techniques. The purpose of chapter 2 is to give readers unfamiliar with machine learning enough background information to understand the rest of this paper. Machine learning experts may opt to skip this review of basic techniques.

Chapter 3 is a review of machine learning applications to path-planning. Attention is also given to other machine learning robotics applications that are related to path-planning and/or have a direct effect on path-planning. Machine learning is a multi-purpose tool that has been used in conjunction with robotics in a variety of ways. Therefore, the organization of chapter 3 reflects this fact. Chapter sections are mostly self-contained, with each describing and discussing a unique junction between the two fields. Throughout, an attempt is made to show how each idea fits into the larger context of a robotic system.

1 Robotics Background

In order for a robot to operate autonomously, it must be capable of interacting with its environment in an intelligent way [Lee, 1996]. This implies that an autonomous robot must be able to capture information about the environment and then perform actions based on

[1] I have chosen to include this review because I am personally interested in graph-search algorithms applied to path-planning, and believe that the advances covered in Section-1.5 have lead to a fundamental shift in the way that field-robotics is approached.

that information. A hypothetical robotic system can be dissected into four subsystems:

sensing → representation → planning → actuation

Although, the lines between these subsystems are often blurred in practice.

A *sensor* is the term given to any part of the robotic system that provides data about the state of the environment. Although the definition of a sensor is necessarily broad, the type of information provided by sensors can be broken into three main categories:

1. The world (terrain shape, temperature, color, composition)
2. The system and its relationship to the world (battery charge, location, acceleration)
3. Other concepts of interest (collaborator, adversary, goal, reward).

Any information available to the robot must be provided *a priori* or obtained on-line through sensor observations.

The *representation* is the method by which a robot stores and organizes information about the world. Simple representations may consist of a single value—perhaps indicating the output of a particular sensor. Complex representations may include high-level graphical models and/or geometric maps of the environment.

The *planning* subsystem (or *planner*) is responsible for deciding how the robot should behave, with respect to a predefined task, given the information in the representation. A planner might calculate anything from a desired speed/direction of travel to an entire sequence of actions.

Actuation is the method by which the robot acts on the environment. This may involve sending power signals to motors, servos, or other devices that can modify the physical relationship between the robot and the environment.

All four system components are interrelated. However, in the context of this paper, the relationship between the representation and planning subsystems is especially important. The structure and content of the representation define what kinds of decisions the planner is capable of making, and ultimately the set of action plans available to the robot. Conversely, a particular planning system may require a specific type of representation in order to function. Most of this chapter is devoted to these two subsystems. However, because the particular sensors available to a robot may influence the type of representation used, a short discussions on sensors is also required.

1.1 Sensors

In the context of robotics, the term *sensor* is broad and ambiguous. It can be used to describe any device or module that is capable of capturing information about the world. Instead of trying to define exactly what a sensor is, it is perhaps more helpful to give examples of different kinds of sensors.

Active sensors glean information about the world by sending a signal into the world and then observing how information from that signal propagates back to the sensor. For instance, devices like radar, sonar, lasers, and lidar send a light or sound wave into the world, and then observe how it is reflected by the environment. Tactile sensors probe the environment physically, much like a human feeling their way around a room in the dark.

Passive sensors function by capturing information that already exists in the environment. This includes devices such as thermometers, accelerometers, altimeters, tachometers, microphones, bumper sensors, etc. Devices like cameras, infrared sensors, and GPS receivers are also considered passive sensors—although their assumptions about certain types of information can be violated (e.g. natural light and GPS signals seldom propagate into cavernous environments).

Sensors can sometimes be described as being either *ranged* or *contact* sensors. Ranged sensors capture information about the environment from a distance, and include devices like sonar, radar, cameras, and lidar. In contrast, contact sensors require physical contact with the part of the environment they are sensing, and include devices like thermometers, tactile sensor, strain gages, and bumper sensors.

It is also useful to make the distinction between *grid-based* (or *image*) sensors, and other types of sensors. Image sensors capture multiple (and often simultaneous) readings about a swath of the environment, while other sensors only capture information about a point or along a directional vector. Cameras are arguably the most common type of grid-based sensor. Each pixel represents a light value associated with a particular ray traveling through the environment. Similarly, a laser imaging sensor known as lidar assembles many individual laser readings into a spatially related collection of depth values. Theoretically, any collection of individual sensors can form an image sensor, as long as the spatial relationships between the individual sensors are known. Images are appealing because they provide an additional level of knowledge beyond an unorganized collection of individual sensor readings.

Often raw sensor readings are used in conjunction with prior domain knowledge to infer high-level information about the world, or to increase the accuracy of existing sensor data. For example, cameras and lasers are used to create 'people detectors' in [Haritaoglu et al., 1998] and [Bellotto and Hu, 2009], respectively, and pattern recognition is used to find the location of faces in pictures in [Rowley et al., 1998]. Particle, Kalman, and other filters are often used with GPS data to provide more accurate position measurements [Kltagawa, 1996, Kalman, 1960].

From the representation subsystem's point-of-view, these self-contained modules are essentially *meta-sensors*. That is, software-based sensors that take hardware sensor readings as input, and output more valuable (hopefully) data than a simple hardware sensor. In practice, meta-sensors may either interact with the rest of the system in much the same way as a simple hardware sensor, or they may require data from the representation subsystem to achieve their task. Meta-sensors fit into the theoretical robotic system as follows:

$$\text{sensor} \rightarrow (\text{meta–sensor} \leftrightarrow \text{representation}) \rightarrow \text{planning} \rightarrow \text{actuation}$$

Given a particular sensor, or set of sensors, there are a few things that a system designer must keep in mind when creating the representation and planning subsystems. These include: sensor accuracy, range, time/position of relevance, sampling rate, and usefulness of the data provided.

1.2 Representation

The representation subsystem decides what information is relevant to the robot's task, how to organize this data, and how long it is retained. Simple representations may consist if an instantaneous sensor reading, while complex representations may create an entire model of the environment and/or robot. It should be noted that using a complex representation is not a precondition for achieving complexity in the resulting robot behavior. It has been shown that robust and sophisticated behavior can be produced using simple representations of the environment and *vise versa* [Braitenberg, 1984]. However complex representations may allow a planning system to develop plans that 'think' further into the future. This can be advantageous because knowledge about intended future actions can decrease a system's susceptibility to myopic behavior.

Although planning is throughly addressed in sections 1.3-1.5, it is useful to define a few planning concepts that are of relevance to the representation subsystem. Planners that operate on a few simple rules are called reactive planners [Lee, 1996]. These rules may include things like 'move away from light,' 'move toward open space,' 'follow a line,' etc. [Braitenberg, 1984]. These types of planners require relatively simple representations. However, because all actions are highly Dependant on local environmental phenomena, they are also relatively short-sighted when considering what their future actions will be.

In contrast, model-based (or *proactive* planners) create relatively detailed action plans. For instance, an entire sequence of movements or a high-level path from the robot's current position to a goal position. In other words, proactive planners assume the robot has enough information to know exactly what it would do in the future, assuming it can forecast all changes in environmental state. Current actions may also be influenced by what the system expects to happen in the future. For instance, a rover might temporarily move away from its goal location, in order to avoid hitting an obstacle in the future. Proactive planners generally require more complex representations such as graphical models of environmental connectivity, maps, etc.

In practice, most planning frameworks utilizes a combination of proactive and reactive planning. Proactive planners work toward long-term goal(s), while reactive planners provide flexibility by allowing modifications in response to quickly changing or uncertain environments. Therefore, most robots use both high-level and low-level representations.

There are numerous different types of representations that have been employed in robotic systems to date. Far too numerous, in fact, to adequately address in this paper. In general, low level representations are simple—for instance, a vector of infrared values obtained from a ring of sensors placed around the robot [Minguez and Montano, 2004]. In these types of systems it is not uncommon for representation values to be used as direct input into a low level controller (i.e. the robot moves in the direction of the sensor with the smallest infrared reading).

A common type of high-level representation is a map—which, at the very least, is a world model capable of differentiating between unique locations in the environment. Theoretically, if all possible map types are considered to exist in a high dimensional space, then one of the dimensions of that space represents environmental recoverability. That is, the degree to which the organization of the environment can be recovered from the representation. [Lee, 1996] describes four representations at points along this continuum. The following list is his description of each, verbatim:

1. Recognizable Locations: The map consists of a list of locations which can be reliably recognized by the robot. No geometric relationships can be recovered.
2. Topological Maps: In addition to the recognizable locations, the map records which locations are connected by traversable paths. Connectivity between visited locations can be recovered.
3. Metric Topological Maps: this term is used for maps in which distance and angle information is added to the path description. Metric information can be recovered about paths which have been traveled.
4. Full Metric Maps: Object locations are are specified in a fixed coordinate system. Metric information can be recovered about any objects in the map.

Given Lee's description, one can immediately begin to conceptualize possible representation schemes. 'Recognizable locations' could be implemented using a probabilistic model of observations over states—given the readings from sensors A, B, and C, what is the probability that the robot is at location X? 'Topological Maps' might take this a step further by representing the environment as a connected graph. The robot's location might even be a hidden state X that the robot must infer given current sensor readings A, B, and C—possibly combined with its belief that it was previously at a neighboring location Y or

Z. 'Metric Topological Maps' reduce uncertainty in location by allowing the robot to infer things like 'assuming an initial location Y, location X can be achieved by traveling d meters southwest'. Finally, 'Full Metric Maps' attempt to model the complete spatial organization of the environment.

Although maps along the entire spectrum of environmental recoverability are of theoretical interest and have potential practical applications. There has recently been an overwhelming tendency to use metric topological and full metric maps. This has been fueled by a number of factors including: more accurate localization techniques (the development of the Global Positioning System as well as more robust localization algorithms [Smith et al., 1986,Durrant-Whyte and Bailey, 2006]), better computers that are capable of storing larger and more detailed maps, more accurate range sensors (lidar and stereo vision), and recent developments in planning algorithms that utilize these types of maps.

Other terms for environmental models include *state space* for a discrete model of the world, and *configuration space* or *C-space* for a continuous state model of the world. Note that the term 'C-space' has connotations differing from those of a geometric map. Traditionally, C-space refers to a model that is embedded in a coordinate space associated with a system's degrees-of-freedom, while a map is embedded in a coordinate space associated with the environment. In other words, a C-space is defined by all valid positions of the robot within the (geometric) model of the world, while a map only contains the latter. In practice, a system may simultaneously use both a map and a C-space, possibly deriving one from the other.

1.2.1 Map Features

A single piece of environmental information stored in the representation is called a *feature*. Features can be explicit (i.e. terrain height, color, temperature) or implicit (i.e. map coordinates). When multiple features are associated with the same piece of the environment, the set of features is collectively called a *feature vector*.

1.2.2 Map Scope

Maps may be supplied to the system *a priori* or created on-line. They may be static or updated with new information as the robot explores the environment. Due to the physical limitations of a system, a map is necessarily limited in size and/or detail. Therefore, any map must decide how much of the environment to represent. Common approaches involve:

1. Modeling only the subset of the environment that is currently relevant to the robot's task (assuming this can be determined).

2. Modeling only the subset of environment within a given range of the robot.

3. Expanding the map on-line to reflect all information the robot has accumulated.

4. Modeling different parts of the environment at different levels of detail.

5. Combinations of different methods.

Obviously, there are trade-offs between the various approaches. For instance, increasing map size with exploration may eventually overload a system's storage space, while only modeling a subset of the environment may leave a robot vulnerable to myopic behavior. As with many engineering problems, the most widely adopted solution has been to hedge one's bets by cobbling together various ideas into a hybrid system.

I am personally fond of the 'local vs. global' organization; where two separate representations are used in parallel. A global representation remembers everything the robot has experienced at a relatively coarse resolution, while a local representation models the environment in the immediate vicinity of the robot at a higher resolution (note that a similar effect can be achieved with a single multi-resolution representation). This organization provides the robot with detailed information about its current surroundings, yet retains enough information about the world to perform long-term planning (e.g. to calculate a coarse path all the way to the goal). This framework assumes the system is able to perform sufficient long-term planning in the coarse resolution of the global representation, and also that it will not exceed the system's storage capabilities during the mission.

1.2.3 Map Coordinate Spaces

When a designer chooses to use a metric representation, they must also decide what coordinate space to use for the map and/or C-space. In the case of a metric topological map, this amounts to defining the connectivity descriptions that explain how different locations are related. Obvious map choices include 2D and 3D Cartesian coordinate systems—which are commonly used for simple rovers. C-spaces may include roll, pitch, yaw, etc. in addition to position dimensions. Manipulators typically use high dimensional configuration spaces with one dimension per degree-of-freedom. An alternative representation option is to use the native coordinate space of a sensor. This is commonly done with image sensors, because sensor data is captured in a preexisting and meaningful organization (complete with a coordinate space). Finally, it may be useful to create a unique coordinate space that is tailored to the environment and/or problem domain of the robot. Stanford, the team that won DARPA's second Grand Challenge chose to use a distance-from-road-center coordinate system because the robot's task was to travel along desert roads for which GPS coordinates were provided [Thrun et al., 2006]. Similarly, the Jet Propulsion Lab's LAGR[2] team designed a hyperbolic map to account for camera prospective (i.e. the increase in ground-surface that is captured by pixels approaching the horizon line) [Bajracharya et al., 2008].

1.2.4 Representation: Map vs. C-space

As stated earlier, a map of the world is not necessarily equivalent to the C-space in which planning is achieved. Some planning algorithms require a method of transforming the former into the latter. I will not go into great detail on this topic, but background discussion is necessary.

Systems that plan through a C-space often maintain a separate full metric map of the world from which they can construct portions of the C-space as required [LaValle, 2006]. If a full metric map is used, then the designer must choose how to store information in the map. One solution is to use a polygon model of the world (e.g. a CAD[3] model), where the robot and obstacles are defined by sets of points, lines, polygons, and polytopes. This has the advantage of being able to exactly model any environment and/or robot that can be described as a

[2] DARPA's Learning Applied to Ground Robotics

[3] computer aided drafting

combination of the aforementioned primitives—and can be used to approximately model any environment/robot. Given a polygon-based map, points in the C-space are defined as 'obstacle' if they represent robot positions in the map that overlap or *collide* with obstacle positions in the map.

Often it is computationally complex to maintain a full explicit geometric map of the world. In these cases an implicit model of the environment is utilized—for instance, a black-box function that returns whether or not a proposed point in C-space is free or obstacle. Implicit models are also used when the dimensionality of the configuration space is too large to explicitly construct or efficiently search. Conversely, if an implicit model of the world is used, then the configuration space *cannot* be explicitly constructed. It can, however, be sampled to within an arbitrarily small resolution. Algorithms that maintain a sampled view of the configuration space are called *sample based methods*, and are one of the more common representations used for robots with many degrees of freedom.

When sample based methods are used, it is common to represent the C-space as a connected graph. Each sample represents a state (or graph node), and edges indicate what transitions between states are possible. Non-obstacle states are linked with edges when they are closer to each other than to any obstacle state. Sampling continues until the graph contains the start and goal states and is connected, or until a predetermined cut-off resolution has been reached. The output of this process is a discrete state-space model that can be used for path planning with one of the algorithms described in section 1.5.

Another common solution is to divide the state-space into multiple non-overlapping regions or *cells*, and then store information about each region separately. A list of points is maintained describing cell boundaries, and all points within a cell are assumed to have properties defined by that region's features. This can be advantageous if the state space can be broken into regions that 'make sense' with respect to the real world (e.g. a map containing cells for 'lake,' 'field,' 'road,' etc.). It is also useful when more information is required about a particular point than whether it represents an obstacle or not (e.g. other features exist such as elevation, color, etc.). Note that this kind of information may be provided *a priori* and/or modified/learned on-line via on-board robot sensors. A similar approach, called an *occupancy grid*, involves a discretization of the map into uniform grids organized into rows and columns. This provides the practical advantage of being able to store each feature set in an array that

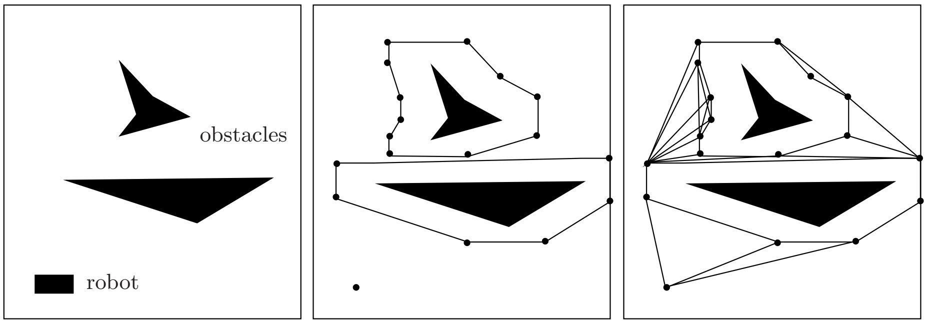

Figure 1: Left: a configuration space with robot and obstacles. Center: the STAR algorithm [Lozano-Perez, 1983] is used to expand obstacles by the size of the robot. Right: a reduced visibility graph is created. The robot is assumed to move by translation; therefore, the C-space has 2 Degrees of freedom.

has the same structure as the map. It also facilitates spatial transformations between sensor-space and map-space. If a map is composed of cells or grids, then the relationship between cells/grids can be used to construct a graph for the purposes of planning. For these reasons, cell and grid maps are commonly used in practice.

When the configuration space can be modeled explicitly in continuous space, and the dimensionality of the C-space is small, then *combinatorial* methods can be used to find the minimum number of states necessary to describe an optimal 'road-map' of the shortest paths between any two C-space points (see Figure 1) [Latombe, 1999]. combinatorial methods become computationally infeasible in more complex spaces because of exponential runtime complexity in the number of C-space dimensions. Note that, as with the other 'continuous methods' described above, the problem is eventually reduced to a discrete state-space graph.

1.3 Planning Methods that are *not* Path-Planning

The main point of this paper is to examine how machine learning is used in conjunction with a particular planning discipline called *path-planning.* Path-planning algorithms approach the planning problem by attempting to find a sequence of actions that, based on the representation, are likely to move the robot from its current configuration to a goal configuration. Path-planning is not the only planning framework available to a designer, and it is often used alongside other methods. Because a single planning technique is seldom used alone, path-

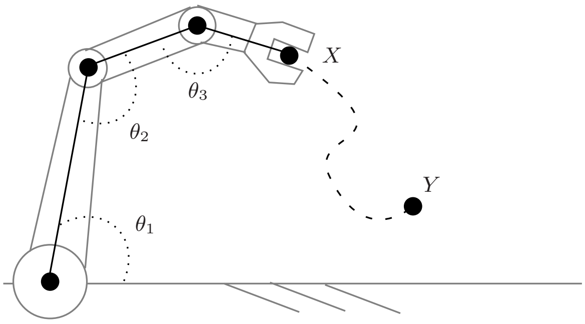

Figure 2: A mechanical arm with gripper at location X is told to move the gripper along the path (dashed line) to location Y. Inverse kinematics is used to calculate the arm angle functions θ_1, θ_2, and θ_3 that accomplish this. Note that the system is under-determined, so there are multiple solutions.

planning is sometimes confused with other planning methods. Therefore, before discussing what path-planning *is*, it is helpful to outline what path planning *is not*.

1.3.1 Feedback Control

Feedback control or *closed loop control* is a branch of control theory that deals with regulating a system based on both the desired state of the system and its current state [Hellerstein et al., 2004]. In robotics, feedback control is used as the final layer of logic between a servo/motor and the rest of the system. Feedback control is responsible for regulating things like speed and heading, and can be implemented in either hardware or software. Feedback control decides things like 'how much power should be given to the motors, in order to achieve a particular speed or position.' It does not make higher-level decisions such as 'what speed or position should be achieved.'

1.3.2 Inverse Kinematics

Inverse Kinematics is the planning problem concerned with how various articulated robot parts must work together to position a subset of the robot in a desired way [Featherstone, 2007]. For example, consider a manipulator arm with three joints and a gripper located at the end, Figure 2. The gripper is currently at location X, but is told to move to location Y along a specific path. Inverse kinematics is used to determine the joint angles of the arm that will result in the gripper following the desired path. Inverse Kinematics assumes that the gripper path is provided *a priori*, and does not violate the constraints of the manipulator arm. To summarize, inverse kinematics does not calculate the desired path of the gripper; however, it is responsible for determining how to move the rest of the robot so the gripper follows the path.

The related problem of *kinematics* describes the state of a particular subset of the robot (e.g. 'location of gripper') as a function of joint angles and other degrees of freedom. Kinematics can be used with a map or C-space to calculate the manifold of valid robot positions within the C-space. This manifold may then be used by a path-planning algorithm to find a gripper path that respects the assumptions required by inverse kinematics.

1.3.3 Trajectory-Planning

Trajectory-Planning is the problem of extending path-planning and inverse kinematic solutions to account for the passage of time [LaValle, 2006]. While path-planning calculates a path between two locations in the representation, trajectory-planning determines how the robot will move along that path with respect to time. For instance, acceleration and velocity functions may be calculated with respect to each degree-of-freedom. Considering the mechanical arm example of the previous section, trajectory-planning could be used in conjunction with inverse kinematics to calculate functions of each joint angle with respect to time. Trajectory-planning may also consider additional factors to those imposed by the path. For example, constraints may be placed on physical quantities such as momentum and force that cannot be expressed in a path.

In the literature, the term *motion-planning* is used as a synonym for both path-planning, trajectory-planning, and a related field of control theory [Lee, 1996, LaValle, 2006]. In order to avoid confusion, I will avoid using the term 'motion-planning,' and instead use the terms 'path-planning' and 'trajectory-planning' to differentiate between these ideas. To summarize, trajectory-planning is closely related to—but distinct from—path-planning because trajectory-planning considers time and path-planning does not.

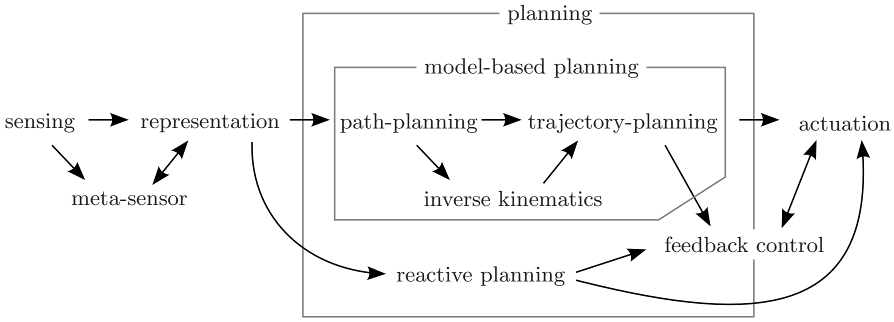

Figure 3: The components of a theoretical robotic system.

1.4 Path-Planning: General Considerations

Path-planning is the specific problem of finding a sequence of actions that will cause a robot to achieve a goal state, given its current state [LaValle, 2006]. The sequence is created based on the information stored in the representation, and is native to that frame of reference. It may consist of discrete or continuous elements, such as point coordinates or curves, respectively. In other words, path-planning finds a path that travels through the representation. A key assumption is that this can be translated into the real world. For example, a path through a scale Cartesian map of the world might consist of a list of map coordinates. Assuming the map accurately models the environment, this path can be translated into the real world using a simple projection.

It is worth reiterating that path-planning is only one part of the entire planning subsystem. Figure 3 illustrates how path-planning is related to the the other planning methods outlined in section 1.3. Note that path-planning is the planning component most closely tied to the representation subsystem. Decisions made in the representation can affect the output of the path-planning component as much *and often even more than* the specific path-planning algorithm used. The reason being that many modern path-planning algorithms provide a solution to a particular world model that is optimal with respect to a set of constraints—or at least approximates optimality in well defined ways. Thus, many different path-planning algorithms may produce paths through a specific environmental model that are similar[4]. On the other hand, changing the environmental model *itself* can produce more drastic differences

[4] I have not defined any metric by which to gauge path similarity. In the context of this discussion, let 'similar' mean that the average human would be unable to distinguish between the robotic behavior resulting from either path.

in behavior.

1.4.1 General Considerations: Assumptions

General assumptions made by path-planning algorithms beyond the existence of the representation include: localization, accuracy, and goal knowledge. The localization assumption implies that the location of the robot vs. the environment is known and can be expressed in terms of the representation. Often the location is not known with complete certainty, but is divided among a set of possible locations. All information provided to the robot *a priori* or accumulated during its mission is assumed to be accurate to within a factor of error. In other words, the extent of inaccuracies in the representation are bounded, and can either be ignored or compensated for by the planning system. Finally, the system assumes that a mission goal exists and can be expressed within the framework of the representation.

1.4.2 General Considerations: Goals

A goal defines what the path-planner is supposed to achieve. The goal may be a single objective or a set of objectives. The most common goal for a path-planning algorithm, and the one addressed in the remainder of this paper, is finding a path through the C-space from the robot's current position to another specific *goal* position. Note, this is by no means the only goal that a path-planner might expect. Examples of other objectives include: observing as much of the environment as possible (mapping), moving from one location to another while avoiding other agents (clandestine movement), following another agent (surveillance), and visiting a set of locations while simultaneously attempting to minimize movement (traveling salesman problem).

1.4.3 General Considerations: Completeness

A path-planning (or other) algorithm is called *complete* if it both (1) is guaranteed to find a solution in a finite amount of time when a solution exists, and (2) reports failure when a solution does not exist. *Resolution completeness* means that an algorithm will find a solution in finite time if one exits, but may run forever if a solution does not exist[5]. *Probabilistic*

[5] The name 'resolution completeness' comes from algorithms that attempt to build a graph of the world by randomly sampling the C-space. Assuming a solution does not exist, these algorithms are guaranteed to terminate only when the C-space is fixed in size and sample points are limited to a finite resolution.

completeness applies to random sampling algorithms. It means that given enough samples, the probability a solution is found approaches 1. Note, however, this does not guarantee the algorithm will find a solution in finite time [LaValle, 2006].

1.5 Path-Planning: Algorithms

This section is devoted to path planning-algorithms that do not use machine learning. In order to find a path without resulting to trial-and-error, the representation must contain connectivity information about the different place it represents. This means that a 'recognizable locations' type of map cannot be used. On the other hand, topological maps, metric topological maps, and full metric maps are all valid.

1.5.1 Generic Graph-Search Algorithm

The first family of path-planning algorithms I will discuss are called *graph-search algorithms*. As the name implies, these algorithms perform path search through a graph. Graph search algorithms can be used directly on topological and metric topological maps, because these representations are essentially graphs. Assuming the world is deterministic (i.e. the same action always produces the same result, given a particular world state), many graph search algorithms will find an optimal path with respect to the representation. An entire subset of graph-search algorithms have been developed for the case when sampling-based methods or combinatorial methods are used in the representation.

Let v_i represent an arbitrary vertex in a graph and let $\mathbf{V}$ represent a set of arbitrary vertices. Each node has a unique identifier i. Two vertices v_i and v_j are the same if $i = j$ and different if $i \neq j$. $\mathbf{v_{goal}}$ is a set of goal vertices that represent acceptable termination states for the path, and $\mathbf{v_{start}}$ is a set of possible starting locations. $\mathbf{G}$ is the set of all vertices in the graph. Let $e_{i,j}$ represent an edge that goes from node v_i to node v_j. Edges between nodes can either be directed $e_{i,j} \neq e_{j,i}$ or undirected $e_{i,j} = e_{j,i}$. Most graph-serach algorithms do not specifically require an edge to be directed or undirected, but will only traverse directed edges in one direction (from node v_i to node v_j). Without loss of generality, undirected edges can be though of as two directed edges, one going in either direction (the undirected edge $e_{i,j}$ accomplishes the same connectivity as the two directed edges $e_{i,j}$ and $e_{j,i}$). Therefore, for simplicity, I shall assume that all edges are directed.

Let $\mathbf{e}_i$ be the set of all edges that leave vertex v_i. That is $e_{i,j} \in \mathbf{e}_i$ if and only if $e_{i,j}$ exists. Node v_j is considered a neighbor of node v_i if $e_{i,j} \in \mathbf{e}_i$. Finally, let $\mathbf{E}$ represent the set of all edges in the graph. Graph-search algorithms assume the existence of $\mathbf{V}$, $\mathbf{E}$, $\mathbf{v_{goal}}$, and $\mathbf{v_{start}}$.

During graph-search, an algorithm starts at $\mathbf{v_{start}}$ and attempts to find a path to $\mathbf{v_{goal}}$, or *vise versa*, by exploring from node-to-node via edges. It is known as *forward search* when the search is conducted from $\mathbf{v_{start}}$ to $\mathbf{v_{goal}}$, and *backward search* when the search is conducted from $\mathbf{v_{goal}}$ to $\mathbf{v_{start}}$. *Bidirectional search* starts at both $\mathbf{v_{goal}}$ and $\mathbf{v_{start}}$ and attempts to connect the two searches somewhere in the middle. *Multi-directional search* starts at $\mathbf{v_{goal}}$ and $\mathbf{v_{start}}$ as well as other random or intuitive locations and attempts to link the searches together in such a way that a path is found between $\mathbf{v_{start}}$ and $\mathbf{v_{goal}}$. When an optimal[6] forward/backward search algorithm is used to create a bidirectional search algorithm, the resulting algorithm is also optimal. However, this does not extend to multi-directional search.

A node is said to be *unexpanded* if it has not yet been reached by the search algorithm. A node is *alive* or *open* if it has been reached, but has at least one neighbor that has not yet been reached (forward search), or is the neighbor of one node that has not yet been reached (backward search). A node is called *expanded*, *closed*, or *dead* when it has been reached by the search and so have all of its neighbors (forward searh), or so have all of the nodes it is a neighbor of (backward search). In other words, in forward search node v_i is open/closed if all v_j have not/have been reached such that $e_{i,j} \in \mathbf{E}$. In backward search node v_j is open/closed when all v_i have not/have been reached such that $e_{i,j} \in \mathbf{E}$.

As the search progresses, a list is maintained of the nodes that are currently open. This is called the *open-list*. Note that possible progress away from closed nodes has already been made, and we do not yet know how to get to unexpanded nodes. Therefore, the open-list contains the particular subset of nodes from which further progress is possible. At each step of a forward search, all unexpanded neighbors v_j of a particular node on the open-list v_i are added to the open-list, and v_i is removed from the open-list. Backward search is similar, except that all nodes v_i for which the open-list node v_j is a neighbor are added to the open-

[6] i.e. an algorithm that is guaranteed to find an optimal path with respect to some metric.

list and v_j is removed from the open-list. Search successfully terminates when a goal/start node is expanded (in forward/backward search). Search is unsuccessfully terminated when the open-list is exhausted—implying a path between $\mathbf{v_{start}}$ and $\mathbf{v_{goal}}$ does not exist.

The relative order in which nodes are expanded can be used to create a tree-representation of the graph-search. This is called a *search-tree*. In forward or backward search the root(s) of the tree are at $\mathbf{v_{start}}$ or $\mathbf{v_{goal}}$, respectively. In practice, it is common to use *back-pointers* to preserve the structure of the search-tree by creating a back-pointer from each node v_j to the earlier node v_i from which v_j was discovered[7]. In bidirectional search, there is a separate search-tree for both the forward and backward directions of the search, and in multi-directional search each search has its own tree. In bidirectional and multi-directional search, two trees are joined when a particular node is included in both search-trees. For the remainder of this paper I shall assume that search occurs in the backward direction[8]. If this is done, then a path can easily be found from $\mathbf{v_{start}}$ to $\mathbf{v_{goal}}$ by following search-tree back-pointers after a successful termination.

Algorithm 1 GRAPH-SEARCH

```
Require: G, E, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
 1: add $\mathbf{v_{goal}}$ to the open-list
 2: while the open-list is not empty do
 3:     remove a node from the open-list and call it $v_j$
 4:     if $v_j \in \mathbf{v_{start}}$ then
 5:         return SUCCESS
 6:     for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
 7:         if $v_i$ is unexpanded then
 8:             add $v_i$ to the open-list
 9:             set back-pointer from $v_i$ to $v_j$
10: return FAILURE
```

Algorithm 1 displays pseudo-code for a generic graph-search algorithm. The algorithm starts by adding the goal nodes to the open-list (line 1). Next, while there are still nodes in the open-list, a node v_j is chosen to be expanded (line 3). If v_j is a start node, then the algorithm terminates successfully (lines 4-5). Otherwise, v_j is closed by adding all of the nodes v_i for which v_j is a neighbor to the open-list (line 8). A back-pointer is created from v_i to v_j (line 9) so that the path can be extracted from the search-tree upon successful termination. If the list becomes empty then there are no possible paths from $\mathbf{v_{start}}$ to $\mathbf{v_{goal}}$, and the algorithm

[7] Depending on the algorithm, back-pointers may be modified to account for better ways of navigating the environment, as they are discovered.

[8] This is slightly confusing because the search algorithm begins at $\mathbf{v_{goal}}$ and attempts to get to $\mathbf{v_{start}}$

returns failure at line 10.

Once the search-tree has been constructed from $\mathbf{v_{goal}}$ to $\mathbf{v_{start}}$, the actual path is extracted in a post-processing step. The nodes and edges that make up the path represent states in the world and the relationships between them. Depending on the specific representation used, the path might be created from information stored in the nodes, edges, or both. For example, if nodes are associated with coordinates in a C-space, the path might be described as a 'bread-crumb' trail of the coordinates associated with path nodes. If a discrete topological map is used, edges may contain prior knowledge about how to transition between the states they connect.

Note that the method of choosing v_j from the open-list on line 3 has not been specified. This is the heart of any graph search algorithm. The next two subsections describe naive methods of choosing v_j that are well known. Subsequent subsections cover more complicated techniques.

1.5.2 Depth-First Search

As the name implies, depth-first search performs graph-search by building the search-tree as deep as possible, as fast as possible [Cormen et al., 2001]. This is accomplished by making the open-list into a stack, and always expanding the top node of the stack. Pseudo-code for depth-first search is displayed in Algorithm-2.

Algorithm 2 Depth-First Search

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
 1: for all $v_{\mathbf{goal}} \in \mathbf{v_{goal}}$ do
 2:     push-top($v_{\mathbf{goal}}$,open-list)
 3: while the open-list is not empty do
 4:     $v_j$ = pop-top(open-list)
 5:     if $v_j \in \mathbf{v_{start}}$ then
 6:         return SUCCESS
 7:     for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
 8:         if $v_i$ is unexpanded then
 9:             push-top($v_i$, open-list)
10:             set back-pointer from $v_i$ to $v_j$
11: return FAILURE
```

The functions push-top() and pop-top() place a node on the top of the stack and remove a node from the top of the stack, respectively. Given a finite graph, depth-first search is complete. However, in a countably infinite graph, depth-first search is not complete and may not even find a solution when one exists. In practice, depth-first search is used when $\mathbf{v_{start}}$ contains many nodes, and all of them are expected to be distant from $\mathbf{v_{goal}}$.

1.5.3 Breadth-First Search

Breath-first search is similar to depth-first search, except it attempts to make the search-tree as broad as possible, as quickly as possible [Cormen et al., 2001]. This is accomplished by changing the open-list into a queue, adding open nodes to the back of the queue, and expanding nodes off the front of the queue. This way, nodes are expanded in the same order they are discovered. Pseudo-code for breadth-first search is displayed in Algorithm-3. The function push-back() adds a node to the back of the queue.

Algorithm 3 Breadth-First Search

```
Require: G, E, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
for all $v_{\mathbf{goal}} \in \mathbf{v_{goal}}$ do
    push-back($v_{\mathbf{goal}}$,open-list)
while the open-list is not empty do
    $v_j$ = pop-top(open-list)
    if $v_j \in \mathbf{v_{start}}$ then
        return SUCCESS
    for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
        if $v_i$ is unexpanded then
            push-back($v_i$, open-list)
            set back-pointer from $v_i$ to $v_j$
return FAILURE
```

Breadth-First Search is complete in a finite graph. It is incomplete in a countably infinite graph; however, it will find a solution if one exists (i.e. is resolution complete). Breadth-first search is more methodical than depth-first search and is usually more useful when $\mathbf{v_{start}}$ only includes a few nodes. However, breadth-first search may be inefficient in large graphs and/or high dimensional spaces.

1.5.4 Best-First Search

Often nodes and/or edges are associated with values or *cost* that can be used to determine their relative importance with respect to the path-search problem [LaValle, 2006]. Using cost to make informed decisions about search-tree growth is known as *best-first search*. In general, best-first search is not optimal; however, it is possible to create algorithms that are.

Let c_j be the cost associated with node v_j. The specific calculation of c_j is dependent on the task the robot is trying to accomplish and the information stored by representation subsystem. As an example, consider the special case where each node knows *a priori* both the distance from itself to its neighbors and the distance through the graph from $v_{\mathbf{start}}$ to itself. Let $d_{i,j}$ represent the distance from node v_i to v_j and let $d_{\mathbf{start},i}$ represent the minimum distance from $\mathbf{v}_{\mathbf{start}}$ to v_i. It is possible make an informed search-tree expansion at each step—the next node to be expanded is the neighbor with the minimum cost c_j where cost is the sum $c_j = d_{\mathbf{start},i} + d_{i,j}$. This results in a much quicker search than either depth-first or breadth-first search.

Algorithm 4 Standard Best-First Search

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v}_{\mathbf{start}}$, $\mathbf{v}_{\mathbf{goal}}$
for all $v_{\mathbf{goal}} \in \mathbf{v}_{\mathbf{goal}}$ do
    insert($v_{\mathbf{goal}}$,cost($v_{\mathbf{goal}}$),open-list)
while the open-list is not empty do
    $v_j$ = get-best(open-list)
    if $v_j \in \mathbf{v}_{\mathbf{start}}$ then
        return SUCCESS
    for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
        if $v_i$ is unexpanded then
            insert($v_i$,cost($v_i$),open-list)
            set back-pointer from $v_i$ to $v_j$
return FAILURE
```

Algorithm 5 Greedy Best-First Search

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v}_{\mathbf{start}}$, $\mathbf{v}_{\mathbf{goal}}$
$v_j = v_{\mathbf{goal}} \in \mathbf{v}_{\mathbf{goal}}$
while $v_j \neq v_{\mathbf{start}}$ do
    $c_{\mathbf{best}} = \infty$
    for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
        if cost($v_i$) $\leq c_{\mathbf{best}}$ then
            $v_{\mathbf{best}} = v_i$
            $c_{\mathbf{best}}$ = cost($v_i$)
    set back-pointer from $v_{\mathbf{best}}$ to $v_j$
    $v_j = v_{\mathbf{best}}$
return SUCCESS
```

In my opinion, there are two general frameworks for best-first search: *standard best-first search* and *greedy best-first search.* Standard best-first search algorithms store the open-list in a priority heap so they can quickly find the best node to expand. Greedy best-first search algorithms may not store an open-list, opting instead to use only the information locally available at the current node. This minimizes storage requirements and eliminates heap operations, but sacrifices global information. Pseudo-code for standard and greedy best-first search algorithms is displayed in Algorithms 4 and 5, respectively. The function cost(v_j) returns the cost value c_j. The function insert(v_j,c_j,open-list) inserts node v_j into the open-list with the priority value c_j. The function get-best(open-list) returns the node in the open-list with the minimum cost value.

Given the contrived example described above, greedy best-first search should be used over standard best-first search whenever $\mathbf{v_{goal}} = v_{\mathbf{goal}}$. This is because a globally optimal solution can be found by making locally optimal decisions, assuming there is only one place the search can begin. The runtime is constant in the connectivity of the graph multiplied by the maximum number of edges in a minimum cost path.

In other circumstances, the standard framework may provide advantages over the greedy framework. It is important to evaluate whether or not an algorithm is complete when using either type of best-first search. Greedy algorithms are particularly susceptible to getting stuck in infinite loops when the cost function is poorly defined. The next section provides an example of a standard best-first search strategy that can find less costly paths than greedy best-first search.

1.5.5 Dijkstra's Algorithm

Dijkstra's algorithm [Dijkstra, 1959] is one of the earliest discovered optimal best-first search algorithms. Dijkstra's algorithm assumes that distances $d_{i,j}$ are known for all edges $e_{i,j}$. Cost c_i is defined as the minimum distance required to reach the goal from a particular node v_i. When Dijkstra's algorithm is run in the forward direction, cost is frequently called *cost-to-come* because it represents the minimum cost required to come to v_i from $\mathbf{v_{start}}$. This paper deals with the backward version of Dijkstra's algorithm, in which case cost is called *cost-to-go* because it represents the cost of going from node v_i to $\mathbf{v_{goal}}$.

Although each node only knows the distance between itself and its neighbors, we can calculate c_i for all nodes during the search by realizing that $c_i = \min_j d_{i,j} + c_j$, and defining $c_{\mathbf{goal}} = 0$. Pseudo-code is displayed in Algorithm 6. The function update(v_i,c_i,open-list) inserts v_i into the open-list (priority heap) with the priority value c_i, if it is not already there. Otherwise, it updates the position of v_i in the queue to reflect the new priority value c_i. This is necessary because the algorithm may find a cheaper way to reach an open node during exploration.

Algorithm 6 DIJKSTRA'S

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
 1: for all $v_{\mathbf{goal}} \in \mathbf{v_{goal}}$ do
 2:     insert($v_{\mathbf{goal}}$,0,open-list)
 3: while the open-list is not empty do
 4:     $v_j$ = get-best(open-list)
 5:     if $v_j \in \mathbf{v_{start}}$ then
 6:         return SUCCESS
 7:     for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
 8:         if $v_i$ is unexpanded or $c_i > d_{i,j} + c_j$ then
 9:             $c_i = d_{i,j} + c_j$
10:             update($v_i$,$c_i$,open-list)
11:             set back-pointer from $v_i$ to $v_j$
12: return FAILURE
```

It is easy to see that this creates a search-tree that is optimal with respect to cost. Nodes are expanded based on their cost-to-go, and the node in the heap with minimum cost-to-go is always expanded. Also, open nodes in the heap are reordered (and the search-tree adjusted) whenever a cheaper path to the goal is found through a recently expanded neighbor (lines 8 to 11). Dijkstra's Algorithm is complete for finite graphs and resolution complete for countably infinite graphs. Note that if $d_{i,j} = 0$ for all i and j then Dijkstra's algorithm degenerates into breadth-first search.

1.5.6 A* Algorithm

Often a *heuristic* function can be used to provide an estimate of the cost of traveling between two nodes. Let the heuristic estimate of cost between nodes v_i and v_j be denoted $h_{i,j}$. The heuristic estimate of cost from v_{start} to v_j is denoted $h_{\mathbf{start},j}$. The *A* search algorithm* [Hart et al., 1968] performs best-first search where cost is defined as:

$$c_i = h_{\mathbf{start},i} + d_{i,\mathbf{goal}} = h_{\mathbf{start},i} + \min_j \left(d_{i,j} + d_{j,\mathbf{goal}}\right)$$

where v_j is a neighbor of v_i. That is, the estimated cost of traveling from the v_{start} to node v_j plus the actual cost of traveling from v_j to v_i plus the actual minimum cost of traveling through the graph from v_i to the $\mathbf{v_{goal}}$. Pseudo-code for the A* algorithm is displayed in Algorithm 7.

Algorithm 7 A*

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
1: for all $v_i \in \mathbf{v_{goal}}$ do
2:     insert($v_i$,$h_{\mathbf{start},i}$,open-list)
3: while the open-list is not empty do
4:     $v_j$ = get-best(open-list)
5:     if $v_j \in \mathbf{v_{start}}$ then
6:         return SUCCESS
7:     for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
8:         if $v_i$ is unexpanded or $d_{i,\mathbf{goal}} > d_{i,j} + d_{j,\mathbf{goal}}$ then
9:             $d_{i,\mathbf{goal}} = d_{i,j} + d_{j,\mathbf{goal}}$
10:            $c_i = h_{\mathbf{start},i} + d_{i,\mathbf{goal}}$
11:            update($v_i$,$c_i$,open-list)
12:            set back-pointer from $v_i$ to $v_j$
13: return FAILURE
```

If $h_{i,j} \leq d_{i,j}$ then the heuristic is said to be *admissible* or *optimistic*. When A* uses an admissible heuristic, the path from the start to the goal is guaranteed to be optimal with respect to cost. This is because, as long as the heuristic never overestimates the true cost, the algorithm will never miss an opportunity to find a less expensive path through a node on the open-list[9]. As with other best-first search algorithms, A* modifies the search-tree to reflect better paths through nodes on the open-list as they are discovered (lines 8-12). A* is complete in a finite graph but may or may not be resolution complete in a countably infinite graph, depending on the heuristic that is used. Given an admissible heuristic, A* is resolution complete in a countably infinite graph. If the heuristic is inadmissible, then there is not enough information to determine whether or not the algorithm is resolution complete without examining the specific properties of the heuristic function. Note that if $h_{i,j} = 0$ for all i and j then A* degenerates into Dijkstra's algorithm.

The A* algorithm has a long history of being used in robotic path planning. A common application is in the context of mobile rovers planning through an occupancy grid scale-model of the world. In this case, distance can be defined as the L^2-norm between points (assuming grids are spaced unit length apart, all horizontal and vertical neighbors are 1 unit apart, all diagonal neighbors in 2D are $\sqrt{2}$ units apart, and all diagonal neighbors in 3D are $\sqrt{3}$ units apart). The Euclidean distance between grid centers can be used as an admissible heuristic.

[9] Conversely, if the heuristic overestimated the true cost of travel, than potentially good paths would not be explored, and the algorithm might not find the least expensive path.

1.5.7 Anytime Algorithms

When an algorithm name includes the term *anytime*, it implies that the algorithm quickly finds a sub-optimal solution and then works to refine that solution while time permits [Hansen and Zhou, 2007]. Given enough time for refinement, anytime algorithms eventually converge to an optimal solution. Anytime algorithms are useful for real-time applications, where performing any valid yet sub-optimal action is considered better than performing an invalid action or no action at all. For instance, when driving a car down a highway, most people are willing to settle for a solution that keeps the car on the road and free from collisions—even if the solution is slightly longer than optimal. There are a couple of general frameworks for anytime algorithms I will discuss in the context of A*. I shall refer to them as *Iterative Anytime Methods* and *Refinement Anytime Methods*, although these terms are not used in the literature.

Algorithm 8 ITERATIVE ANYTIME A*

```
Require: G, E, v_start, v_goal, $\epsilon$, $\Delta\epsilon$
while time remains do
    initailize(open-list)
    for all $v_i \in \mathbf{v_{goal}}$ do
        insert($v_i$,$\epsilon h_{\mathbf{start},i}$,open-list)
    while the open-list is not empty and time remains do
        $v_j$ = get-best(open-list)
        if $v_j \in \mathbf{v_{start}}$ then
            save the path from $v_j$ to $v_{\mathbf{goal}}$
            break
        for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
            if $v_i$ is unexpanded for this value of $\epsilon$ or $d_{i,\mathbf{goal}} > d_{i,j} + d_{j,\mathbf{goal}}$ then
                $d_{i,\mathbf{goal}} = d_{i,j} + d_{j,\mathbf{goal}}$
                $c'_i = \epsilon h_{\mathbf{start},i} + d_{i,\mathbf{goal}}$
                update($v_i$,$c'_i$,open-list)
                set back-pointer from $v_i$ to $v_j$
    if $v_j \in \mathbf{v_{start}}$ then
        $\epsilon = \epsilon - \Delta\epsilon$
        if $\epsilon < 1$ then
            return SUCCESS
    else
        break
if a path has been saved then
    return SUCCESS
return FAILURE
```

The first idea is to quickly perform a search that will find a suboptimal solution in the alloted amount of time, and then continue to perform increasingly more optimal searches as time permits. This can be accomplished with A* by weighting the admissible heuristic $h_{i,j}$ by a factor $\epsilon \geq 1$ to get $\epsilon h_{i,j}$. This causes the heuristic to over-estimate the cost of traveling between two nodes by at most ϵ. The overall effect is that solutions found using $\epsilon h_{i,j}$ are at most ϵ times more expensive than the optimal solution [Davis et al., 1988]. Larger values for ϵ cause the search to terminate more quickly because fewer nodes are expanded (because the estimated cost-to-come values of nodes are artificially inflated). This may not appear intuitive, however, the time savings can be significant—especially in graphs with a high degree of connectivity.

Iterative Anytime A* performs one search, saves the result, and then decreases ϵ by $\Delta\epsilon$ and repeats the process. It is assumed the initial value of $\epsilon = a(\Delta\epsilon) + 1$ for some constant a, so that ϵ will eventually have a value of 1 (in which case an optimal path will be found). Let c'_i represent the 'minimum' cost to the goal as determined when $\epsilon \neq 1$. Explicitly, $c'_i = \epsilon h_{\mathbf{start},i} + d_{i,\mathbf{goal}}$.

Pseudo-code for Iterative Anytime A* is given in Algorithm 8. The function initialize() removes all elements from the open-list. Lines 3 to 15 represent one execution of A*. On lines 8 and 16-17, the algorithm saves a path and decreases ϵ, assuming a path to the start has been found. On line 18 a check is performed to see if the new epsilon is less than 1. The final iteration of A* is optimal, given the assumption that the final value of ϵ is 1, so the algorithm terminates successfully on line 19. Once time expires, the algorithm terminates successfully if at least one path has been found, otherwise the algorithm terminates unsuccessfully.

Refinement anytime methods operate by first finding a sub-optimal solution, and then using the remaining search time to gradually improve that solution. After the search-tree reaches the start node for a particular value of ϵ in Iterative Anytime A* (Algorithm 8, line 7), the open-list is still populated with nodes that may well yield less costly paths. However, these nodes have been inserted with a priority value based on a pessimistic ϵ. We can modify Algorithm 8 into a refinement anytime framework by decrementing ϵ and then proceeding as normal without calling initialize(open-list). Further, the cost of the current best solution can be used to focus the search. This is accomplished by removing nodes from the open-list that have (admissible) values of c_i that are greater than the cost of the current best solution[10]. This works because c_i will not overestimate the cost of traveling from $\mathbf{v}_{\mathbf{start}}$ to $\mathbf{v}_{\mathbf{goal}}$ through

[10] Note that this requires the algorithm to store both c_i and c'_i.

v_i; and therefore, these nodes cannot possibly lead to better solutions than the best already found.

As back-pointers are added to the search-tree and modified within it, they always reflect the least costly way back to the goal discovered so far. New values of ϵ may allow less costly paths to be discovered through old nodes that have already been expanded. These nodes are re-inserted into the open-list, so that the resulting reductions in cost can be propagated through the search-tree.

Algorithm 9 Refinement Anytime A*

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$, $\epsilon$, $\Delta\epsilon$
 1: for all $v_i \in \mathbf{v_{goal}}$ do
 2:     insert($v_i$,$\epsilon h_{\mathbf{start},i}$,open-list)
 3: $B = \infty$
 4: while time remains and the open-list is not empty do
 5:     $v_j$ = get-best(open-list)
 6:     if $v_j \in \mathbf{v_{start}}$ then
 7:         if $\epsilon > 1$ then
 8:             $\epsilon = \epsilon - \Delta\epsilon$
 9:         $B = d_{\mathbf{start}}$
10:         continue
11:     else if $c_j \geq B$ then
12:         continue
13:     for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
14:         if $v_i$ is unexpanded or $d_{i,\mathbf{goal}} > d_{i,j} + d_{j,\mathbf{goal}}$ then
15:             $d_{i,\mathbf{goal}} = d_{i,j} + d_{j,\mathbf{goal}}$
16:             $c'_i = \epsilon h_{\mathbf{start},i} + d_{i,\mathbf{goal}}$
17:             $c_i = h_{\mathbf{start},i} + d_{i,\mathbf{goal}}$
18:             update($v_i$,$c'_i$,open-list)
19:             set back-pointer from $v_i$ to $v_j$
20: if $B \neq \infty$ then
21:     return SUCCESS
22: return FAILURE
```

Pseudo-code for Refinement Anytime A* is given in Algorithm 9. On lines 7 and 8 the value of ϵ is decreased every time a less expensive path is found to the goal. Note that using this condition to decrease ϵ is completely arbitrary, and many other methods could be used (for instance, ϵ could be decreased on a predetermined time schedule, assuming at least one path has been found). Further, lines 7 and 8 could be completely removed without affecting the algorithm's eventual convergence to the optimal solution. The advantage of keeping lines 7 and 8 is that reductions in path cost tend to propagate further through the search-tree before other possibilities are explored. Although, it is not possible to determine *for sure* which option will be more advantageous for a particular problem.

One detail I have not covered is how the initial value of ϵ and the step size $\Delta\epsilon$ should be chosen. Unfortunately, there is no standard method, and this problem is largely ignored in the literature. Typical assumption statements read "...the [non-admissible heuristic] is used to select nodes for expansion in an order that allows good, but possibly suboptimal, solutions to be found quickly" [Hansen and Zhou, 2007]. Regardless, Both types of Anytime algorithms have been extensively used for real-time applications, and have been built on top of other algorithms besides A*.

1.5.8 Lifelong A*

*Lifelong A** [Likhachev and Koenig, 2001] has been developed specifically for applications in which multiple searches need to be performed through a changing representation, but the the start and goal locations are guaranteed to remain the same. If the number of changes is small compared to the size of the graph[11], then it is more efficient to repair the existing search-tree than to perform an entirely new search from scratch.

Given the type of search-tree that we have been considering up to this point, a Lifelong A* algorithm could be implemented by reinserting into the open-list all nodes with modified edge cost or connectivity. These changes could then be propagated through the search-tree as was done with Refinement Anytime A*. However, [Likhachev and Koenig, 2001] take a slightly different approach because they do not explicitly store a pointer-based search-tree. Instead, Lifelong A* implicitly models the search-tree structure by storing cost-to-go values $c_{j,\mathbf{goal}}$ at each node. When the start node is reached, a path is extracted using gradient descent over cost-to-go values[12], and ties are broken arbitrarily.

Let node v_j be a neighbor of v_i. The quantity rhs_i is defined as follows:

$$rhs_i = \min_j c_{i,j} + c_{j,\mathbf{goal}}$$

and represents the minimum cost of traveling to the gaol from v_i through one of its neighbors. For the case $v_i = v_{\mathbf{goal}}$ the value $rhs_{\mathbf{goal}} \equiv 0$. As changes are made to the representation, Lifelong A* updates rhs_i if an outgoing edge of v_i has been modified. Next, if $rhs_i \neq d_{i,\mathbf{goal}}$

[11] For example, the number of nodes modified between searches is less than half the size of the search-tree

[12] i.e. all neighbors of v_i are examined to see which has the lowest $c_{j,\mathbf{goal}}$ value and the path then moves to that node.

then the cost of moving from v_i to the goal is no longer accurate, so v_i is inserted back into the open-list. When $rhs_i \neq c_{i,\mathbf{goal}}$ the node v_i is said to be *inconsistent*. The terms *under-consistent* and *over-consistent* are also used to describe the cases when $rhs_i > c_{i,\mathbf{goal}}$ and $rhs_i < c_{i,\mathbf{goal}}$, respectively. The open-list priority heap is now sorted according to a lexicographic ordering of two values $[k_1{,}k_2]_i$ where $k_1 = h_{\mathbf{start},i} + \min(c_{i,\mathbf{goal}}, rhs_i)$ and $k_2 = \min(c_{i,\mathbf{goal}}, rhs_i)$. Pseudo-code for Lifelong A* is given in Algorithm 10. rhs_i values and $c_{i,\mathbf{goal}}$ are initialized to ∞. The function top() returns the vertex on the top of the open-list without removing it.

Algorithm 10 Lifelong A*

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
procedure CalculateKey($v_i$)
    return $[h_{\mathbf{start},i} + \min(c_{i,\mathbf{goal}}, rhs_i), \min(c_{i,\mathbf{goal}}, rhs_i)]$
procedure UpdateVertex($v_i$)
    if $v_i \notin v_{\mathbf{start}}$ then
        $rhs_i = \min_j c_{i,j} + c_{j,\mathbf{goal}}$
    if $v_i \in$ open-list then
        remove $v_i$ from the open-list
    if $(c_{i,\mathbf{goal}} \neq rhs_i)$ then
        update($v_i$,CalculateKey($v_i$),open-list)
procedure PerformSearch()
    while CalculateKey(top(open-list)) < CalculateKey($v_{\mathbf{start}}$)
    or $rhs_{\mathbf{start}} \neq c_{\mathbf{start},\mathbf{goal}}$ do
        $v_j$ = get-best(open-list)
        if $(c_{j,\mathbf{goal}} > rhs_j)$ then
            $c_{j,\mathbf{goal}} = rhs_j$
            for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
                UpdateVertex($v_i$)
        else
            $c_{j,\mathbf{goal}} = \infty$
            UpdateVertex($v_j$)
            for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
                UpdateVertex($v_i$)
procedure main
    for all $v_i \in \mathbf{v_{goal}}$
        update($v_i$,CalculateKey($v_i$),open-list)
    $rhs_{\mathbf{start}} = 0$
    forever
        PerformSearch()
        Wait for changes in edge costs
        for all edges $e_{i,j}$ with changed costs do
            update UpdateVertex($v_i$)
```

When a node v_i is under-consistent, its current distance to goal value is too low and the implicit search-tree structure may need be modified so that v_i transitions to a different neighbor (line 19). Also, nodes that used to transition to v_i may be able to find a better path through a different neighbor. This is accomplished by re-initializing v_i (line 19) and then inserting all nodes that might transition to v_i onto the open-list (lines 20-21). When a node is over-consistent, its distance to goal value is too high. In that case, the search-tree may need to be modified so that other nodes transition to v_i instead of a different neighbor. This is accomplished by having v_i adopt the rhs_i value as its $c_{i,\mathbf{goal}}$ value (line 14), and then reinserting any node that can transition to v_i into the open-list (lines 15-16).

Lifelong A* inherits all theoretical properties of A*. When an admissible heuristic is used, it is optimal with respect to the graph (after a search is complete and before edge costs change). It is complete in a finite world and resolution complete in a countably infinite world.

1.5.9 D* Algorithm

*Dynamic A** or *D** is the next logical algorithmic idea after Lifelong A*. The D* algorithm is similar to lifelong A*, except that $\mathbf{v_{start}}$ is allowed to change between searches. As in Lifelong A*, edge costs are also allowed to change. The first version of D* was presented in [Stentz, 1994] and then extended in [Stentz, 1995]. These versions of the algorithm rely on the more traditional notion of explicitly updating back-pointers in the search-tree to reflect graph changes, as opposed to the implicit search-tree used in Lifelong A*. A more recent version called *D* lite* is presented in [Koenig and Likhachev, 2002] and uses the implicit search-tree representation. As a side-note, the term 'lite' has since been adopted to describe algorithms that use the implicit search-tree representation. Results in [Koenig and Likhachev, 2002] appear to show that D* lite runs more quickly than the original D* algorithm in practice[13]. As with Lifelong A*, D* and D* lite inherit the completeness and optimality properties of the A* algorithm they are built on top of. The following discussion is valid for both D* and D* lite; however, the pseudo-code in Algorithm 11 is for D* lite.

As previously noted, the only functional difference between Lifelong A* and D* Lite is that the location of $\mathbf{v_{start}}$ is allowed to move in the latter. This is advantageous in robotic

[13] I believe the reasons for this are still poorly understood at present.

Algorithm 11 D* Lite

```
Require: $\mathbf{G}$, $\mathbf{E}$, $\mathbf{v_{start}}$, $\mathbf{v_{goal}}$
procedure CalculateKey($v_i$)
    return $[h_{\mathbf{start},i} + \min(c_{i,\mathbf{goal}}, rhs_i) + M, \min(c_{i,\mathbf{goal}}, rhs_i)]$
procedure UpdateVertex($v_i$)
    if $v_i \notin \mathbf{v_{start}}$ then
        $rhs_i = \min_j c_{i,j} + c_{j,\mathbf{goal}}$
    if $v_i \in$ open-list then
        remove $v_i$ from the open-list
    if $(c_{i,\mathbf{goal}} \neq rhs_i)$ then
        update($v_i$,CalculateKey($v_i$),open-list)
procedure PerformSearch()
    while CalculateKey(top(open-list)) $<$ CalculateKey($v_{\mathbf{start}}$)
    or $rhs_{\mathbf{start}} \neq c_{\mathbf{start},\mathbf{goal}}$ do
        $v_j$ = get-best(open-list)
        if $(c_{j,\mathbf{goal}} > rhs_j)$ then
            $c_{j,\mathbf{goal}} = rhs_j$
            for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
                UpdateVertex($v_i$)
        else
            $c_{j,\mathbf{goal}} = \infty$
            UpdateVertex($v_j$)
            for all $v_i$ such that $e_{i,j} \in \mathbf{E}$ do
                UpdateVertex($v_i$)
procedure main
    for all $v_i \in \mathbf{v_{goal}}$
        update($v_i$,CalculateKey($v_i$),open-list)
    $M = 0$
    $rhs_{\mathbf{start}} = 0$
    while robot is not at the goal do
        PerformSearch()
        move robot to new $\mathbf{v_{start}}$
        $M = M +$ the magnitude of resulting movement cost
        wait for changes in edge costs
        for all edges $e_{i,j}$ with changed costs do
            update UpdateVertex($v_i$)
```

applications where the robot must move through the world on it way to a goal. Recall that the root of the search-tree is at $\mathbf{v_{goal}}$. Also, $c_{i,\mathbf{goal}}$ and rhs_i reflect the minimum cost of moving from v_i to $\mathbf{v_{goal}}$. Therefore, the structure[14] of the search-tree is valid, regardless of the location of $v_{\mathbf{start}}$. This does not happen accidentally. Search is specifically performed in the backward direction (i.e. from goal to start) so that the start can move without destroying the validity of the search-tree.

[14] Explicit back-pointer structure in the case of D* and implicit structure in the case of D* lite.

An additional consideration must be addressed: given that the open-list is sorted using heuristic estimated cost-to-come values, what do we do about the fact that modifying $\mathbf{v_{start}}$ invalidates these? The solution is to redefine the heuristic estimate of cost-to-come to additionally include the cumulative cost of previous movement. Let $h'_{\mathbf{start},i}$ be the new heuristic cost-from-start. $h'_{\mathbf{start},i} = h_{\mathbf{start},i} + M$, where M is the total cumulative movement cost the robot has incurred during its mission.

Why does this work? To answer this question, let us assume for a moment that heuristic cost-to-come is still calculated as $h_{\mathbf{start},i}$. Further, let us assume that the robot has just moved a particular distance through the word that corresponds to movement cost of d. Values of $h_{\mathbf{start},i}$ that had previously been used to populate the open-list are currently between d too high and d too low. Assuming that we wish to maintain an admissible heuristic, we could go through the entire open-list and reset each value of $h_{\mathbf{start},i}$ to $h_{\mathbf{start},i} - d$. This would ensure admissibility by guaranteeing we do not overestimate cost-to-come. This, however, is also expensive when the heap is large. Instead, the same effect can be achieved by adding d to new heuristic estimated cost-to-come values. This keeps the relative order of heap key values identical to how it would be if the open-list had been exhaustively modified as described above.

D* and D* lite have been widely adopted. In fact, I believe they have enabled a recent revolution in the sub-field of field robotics. The reason for this is because most autonomous field robotic systems perceive the world through sensors attached to the robot. Also, most sensors have a finite range and, therefore, can only perceive environmental changes within a fixed radius of the robot. Because the search-tree is rooted at the goal, any change in edge structure or cost can only affect the part of the search-tree that is further away from the goal than the edge where the change takes place. Therefore, as changes occur during the course of robotic movement, only the outer-most branches of the search-tree are affected (because of the limited range of on-board sensors) and most of the search-tree does not need to be modified. Although D* and D* lite still require an initial search that is equally time consuming to A*, subsequent searches happen much more quickly. In large environments, D* can reduce the time required for on-line search by two or three orders of magnitude compared to A*. This has allowed representations to include larger pieces of the environment and to be modeled at a higher resolution.

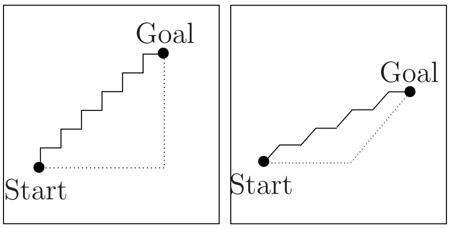

Figure 4: Optimal paths of identical cost through uniform maps. The left and right maps use 4- and 8-connected graphs, respectively. The solid paths are more desirable than the dotted paths with respect to the real world.

1.5.10 Field D*

Graph-search techniques such as Dijkstra's, A*, and D* find an optimal path with respect to a graph representation of the world. Often this representation comes from a grid map, and has a two dimensional 4- or 8-connected structure. Unfortunately, the graph structure *itself* can lead to optimal graph paths that are sub-optimal with respect to the real world. Movement must be broken into a combination of horizontal and vertical transitions in a 4-connected graph, while in an 8-connected graph it must be decomposed into directions along multiples of 45 degrees.

Many (equally) optimal paths may exist with respect to the graph. For instance, given a uniform map, a path that moves through a 4-connected (8-connected) graph horizontally as far as possible and then vertically (diagonally) will have the same cost as a path that alternates between vertical and horizontal (diagonal) movement—see Figure 4. The former path is suboptimal globally, while the latter is suboptimal on the scale of a few map grids. The local sub-optimality of the staircase-like path can be corrected locally with a simple local planner, so the staircase-like path is more desirable in practice. However, given the set of all optimal graph-paths, the task of finding the best with respect to the real world is computationally infeasible. A common solution is to break cost ties by moving toward the goal; however, this technique fails if the goal is blocked by an obstacle—see Figure 5.

*Field D** [Ferguson and Stentz, 2006b] largely avoids the tie-breaking problem by operating in the continuous domain that envelopes the 4- or 8-connected graph neighborhood. Graph node values represent a discrete sampling over a continuous field of the *costdistance* (i.e.

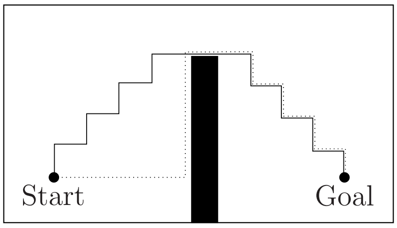

Figure 5: Optimal paths of identical cost through a uniform map with an obstacle. The map uses a 4-connected graph. The dotted path breaks ties in cost by moving toward the goal, but the solid path is more desirable with respect to the real world.

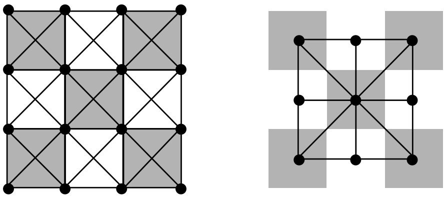

Figure 6: Grid layout over map grids. Graph nodes and edges are black, map grids are gray and white. Field-D* places nodes at grid corners (left), while A*, D*, and D* Lite traditionally place nodes at grid centers (right).

cost[15] integrated over distance) required to reach the goal. Field-D* operates much like D* and D* lite, except that it calculates the *costdistance*-to-goal for continuous points on a graph edge using a linear interpolation between the *costdistance*-to-goal values of that edge's two end-nodes. This allows paths to follow trajectories in the continuous domain. Assuming that Field D* is implemented in a 'lite' framework, paths can be extracted from the field using a form of gradient descent. Edges are not explicitly followed. However, the 8-conectevity structure is used to determine the dependence relationship between node values, and also to define the set of nodes from which continuous field values are calculated. Note that having more than one start location may cause problems with the linear interpolation idea. Therefore, it is assumed that there is only one starting location per search, $\mathbf{v_{start}} = v_{\mathbf{start}}$.

Unlike D* and D* Lite, graph nodes are placed at the corners of map grids instead of at their centers (Figure 6-left vs. 6-right, respectively). An 8-connected graph structure is used

[15] Up to this point, I have used the terms cost and distance interchangeably. This has assumed that the representation consists only of obstacles and free terrain. In practice, it is often useful to define a difficulty metric associated with terrain traversal (e.g. 'mud' or 'hill' may be more difficult to traverse than 'grass' or 'flat,' respectively). When this is done, it is common practice to refer to this metric as *cost*. Although many algorithms directly use this kind of cost to order the open-list priority heap, it can be slightly confusing when another quantity is used, instead.

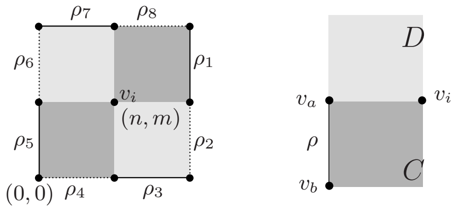

Figure 7: (Left) The 8 edges used to determine the *costdistance*-to-goal of node v_i. (Right) Edge ρ connects nodes v_a and v_b. the *costdistance*-to-goal of v_a and v_b is $d_{a,\mathbf{goal}}$ and $d_{b,\mathbf{goal}}$, respectively. C and D are map grids with *cost* c_c and c_d, respectively.

to define neighboring nodes; however, travel through the map is not restricted to graph edges. The algorithm for Field D* is basically identical to D* lite (Algorithm 11), except that *costdistance* $d_{i,j}$ is used instead of *cost* $c_{i,j}$ and linear interpolation is used to determine the *costdistance*-to-goal for points along grid boundaries. The linear interpolation is based on the *costdistance*-to-goal of the corresponding horizontal or vertical edge's two end-nodes. Note that $d_{i,j}$ and $d_{i,\mathbf{goal}}$ now represent the *costdistance* required to travel from v_i to v_j and from v_i to the goal, respectively[16].

Let (n,m) be the grid position of node v_i relative to the bottom-left node (Figure 7-left). Nodes are spaced 1 unit apart vertically and horizontally. Let (y,x) be a point along one of the 8 edges ρ_k shown in Figure 7-left. $(y,x) \in \rho_k$ for $k = 1 \ldots 8$, and (y,x) exists in the same continuous coordinate space as (n,m). When v_i is expanded, $d_{i,\mathbf{goal}}$ is calculated as follows:

$$d_{i,\mathbf{goal}} = \min\left([(n,m) \to (y,x)] + d_{(y,x),\mathbf{goal}}\right)$$

where $[(n,m) \to (y,x)]$ is the *costdistance* of moving from (n,m) to (y,x) and $d_{(y,x),\mathbf{goal}}$ is the linearly interpolated *costdistance*-to-goal of (y,x). There are 8 edges that must be examined to determine (y,x) and $d_{i,\mathbf{goal}}$. Without loss of generality, we restrict our discussion to finding the minimum *costdistance*-to-goal given a single edge (Figure 7-right). Similar calculations are performed for the remaining 7 edges and the minimum $d_{i,\mathbf{goal}}$ over all 8 is used as the final result.

Let c_c and c_d represent the map cost values of grids C and D, respectively. In [Ferguson and

[16] This is actually not as big of a change as it might appear at first glance. It is, in my opinion, beneficial to use *costdistanc* as the priority heap 'cost' measure for all of the algorithms in the A* family. Doing so reflects both the assumed degree of difficulty of moving across a particular area per unit of distance, as well as the total distance that must be traveled. Note that when *costdistance* is used, *distance* itself is as an admissible heuristic as long as *distance* ≥ 0 and *cost* ≥ 1. Further, if *cost* $\geq C_{nst} > 0$ for some constant C_{nst} than *distance* multiplied by C_{nst} can be used as an admissible heuristic for *costdistnace* ≥ 0.

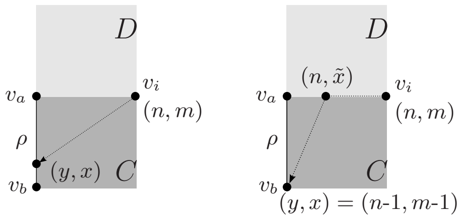

Figure 8: Two possible ways to get from v_i to a point on edge ρ. (Left) The path goes directly to (y, x) through grid C. (Right) The path goes from (n, m) to $(n, \tilde{x})$ along the bottom of grid D, and then through grid C to $(y, x) = (n - 1, m - 1)$ at node v_b.

Stentz, 2006b] it is shown that there are two possible ways a minimum *costdistance*-to-goal path can travel from v_i to point (y, x) on edge ρ. These are illustrated in Figures 8-left and 8-right, respectively, and described by Equations 2 and 4, respectively (note that travel directly from v_i to v_a along the bottom of grid D is handled during the consideration of the edge above ρ).

$$g(y) = c_c \sqrt{1 + (n - y)^2} + d_{a,\mathbf{goal}}(1 - n + y) + d_{b,\mathbf{goal}}(n - y) \tag{1}$$

$$d_{i,\mathbf{goal}} = \min_{n-1 \leq y \leq n} g(y) \tag{2}$$

.

$$g(\tilde{x}) = c_d\,(m - \tilde{x}) + c_c \sqrt{1 + (\tilde{x} - x)^2} + d_{b,\mathbf{goal}} \tag{3}$$

$$d_{i,\mathbf{goal}} = \min_{m-1 \leq \tilde{x} \leq m} g(\tilde{x}) \tag{4}$$

The minimum of Equations 2 and 4 is used as the final result with respect to edge ρ. The lesser of the two depends on the specific values of c_c, c_d, $d_{a,\mathbf{goal}}$, and $d_{b,\mathbf{goal}}$. Given the case illustrated in Figure 8-right, it is proven in [Ferguson and Stentz, 2006b] that the path will exit grid C at node v_b.

Field D* is complete in a finite sampling of the environment and resolution complete in a countable infinite sampling. Given an admissible heuristic, it is optimal in the sense that it will return the best solution with respect to the problem formulation. With respect to the real world, it much closer to optimal than D* and D* Lite.

1.5.11 Fast-Marching Level-Set Methods

Fast-Marching Level-Set Methods [Sethian, 1996,Philippsen, 2006] are similar to Field D* in the sense that they assume nodes values represent sample points from a continuous field. The term *level-set* refers to the solution of a real-valued function (with any number of variables), for a specific constant. Given a 3D map described by a height function $z = f(x, y)$, a level-set can be thought of as the set of all points at a given height z. In general, level-set methods are used to create something analogous to a height map that has one more dimension than the original representation $f(x, y)$. Instead of elevation, 'higher' z values correspond to points located further away from the goal[17]. Gradient descent can be used to find a path to the goal through this modified form of the representation.

A common analogy for this process involves a grass-fire. A small fire is started at the goal and gradually spreads throughout the environment. The boundary of the fire is called a wave-front. It is assumed that, the fire spreads at a constant rate through open terrain. Further, it spreads in a direction perpendicular to the wave-front from any point on the wave front. A any given time t, the boundary is a level-set with respect to t. Map obstacles are handled in two ways: (1) The fire cannot spread to portions of the map containing lethal[18] obstacles. (2) Non-lethal[19] obstacles only slow the progress of the fire as it sweeps through—they do not stop it. In this example, the 'height' dimension z represents time[20]. Again, a level-set can be calculated at each time for the wave-front.

In practice, the world is discretized into a grid of coordinates and the time the fire reaches each coordinate is calculated. The term 'fast marching' is used to describe a computationally efficient way to achieve this calculation. The basic idea is to only consider nodes within a predefined distance of the wave-front at time t, when calculating the new wave-front position at time $t + 1$. The discretized nature of the representation can only approximate the exact contour of the wave-front. Therefore, a *kernel function* is used to calculate how the fire propagates through the continuous domain, given the fast-marching band of values around the previous level-set. This is similar to how Field D* uses linear interpolation to calculate *costdistance*-to-goal values. In fact, Field D* uses a linear interpolation kernel. The most common kernel used in fast-marching level-set methods is a quadratic interpolation kernel.

The relationship between Field D* and fast-marching level-set methods does not appear to be as straight-forward as one might expect. Firstly, fast-marching level-set methods have

[17] With respect to some metric.

[18] The robot cannot move through them.

[19] The robot can move through them at a reduced speed.

[20] Note that the 'time' used in the algorithm as no relation to any real-world time that is experienced by the robot.

been slow to incorporate re-planning (i.e. dynamic) ideas that allow the field to be updated on-line to reflect robotic movement and new information, instead of being recreated from scratch (though a few have recently been proposed [Philippsen, 2006]). Secondly, the latter does not yet have a principled way to deal with the cost of an obstacle—although, this is not a concern if the representation only has lethal vs. free terrain. Current methods use a predefined heuristic to calculate how much an obstacle with a given cost will slow the rate of fire propagation. Further, the notion of *time* adds an additional constraint to the problem that does not necessarily reflect any real-world quantity. This is particularly important because the time gradient of the field is used during the path extraction process. That said, it is possible that an equivalence exists between fact-marching level-set methods and Field D* (assuming the same kernel is used in each) if *time* is defined in terms of *costdistance*; however, this has not yet been determined.

1.5.12 Random Trees

Until this subsection, the graph-search algorithms I have talked about have all made the assumption that the entire representation is explicitly known. The traditional graph-search algorithms (depth-first, breadth-first, best-first, A*, Anytime A*, Lifelong A*, D*, and D* Lite) all assume that the structure of the graph is known prior to each search or re-planning step. The field based techniques (Field D* and fast-marching level-set methods) use the graph structure to create a field, and then to calculate field values at a subset of continuous points. After the field is created, the latter methods only use local graph structure (a few nodes near a particular point) for gradient descent path extraction. Although these methods can function using an implicit representation[21] by simply checking the status of nodes as they are added to the open-list, this is usually not done in practice. Implicit representations *are* used in practice when the environment is too large or complex to model explicitly.

Random Trees is the name of a family of algorithms that operate somewhere between the representation and planning subsystems [LaValle and Keffner, 2001]. Random trees are the algorithms of choice when sampling-based-methods are used in the representation (see section 1.2.4). These algorithms create the graph as they go, and do not assume any predefined graph structure. That said, the resulting structure of the graph is at least as important

[21] Recall that an implicit representation uses a black-box function to determine whether or not a point in C-space is valid.

as the graph-search algorithms used to traverse it. In fact, path-search usually happens in parallel to graph creation, so that node values (distance-to-goal, heuristic estimated distance-from-start, etc.) are calculated as soon as a new node is added to the tree. Random tree algorithms usually operate in a continuous space, and the location of new nodes can be anywhere within the C-space. In practice, it is common to constrain a new node to be within a certain distance of an old node. If a new node is closer to an old node than any old node was to an obstacle, then the new node is guaranteed not to collide with an obstacle. Other variations on this idea include: connecting new nodes to the closest old node that does not involve moving through an obstacle, and/or moving any nodes that collide with an obstacle to the edge of that obstacle (instead of simply throwing them away).

Random trees are often used in high dimensional C-spaces, such as those describing an articulated manipulator and/or many robots that must simultaneously coordinate their actions. Many of the graph-search algorithms I have talked about can be implemented in a random tree framework. These include the subfamilies that can be described as 'anytime' and 'dynamic' [Ferguson and Stentz, 2006a, Ferguson et al., 2006, Ferguson and Stentz, 2007]. I do not believe the 'lite' framework has been used, and I assume this is because it is difficult to create a back-pointer *free* search-tree that is flexible enough to both (1) allow random sampling in continuous space and (2) facilitate gradient descent path-extraction.

1.5.13 Non-Random Tree Methods for Continuous Representations

The search through any sample-based representation of continuous space can be viewed as a tree. The search starts at one place ($v_{\mathbf{goal}}$) and attempts to get closer and closer to another ($v_{\mathbf{start}}$). For example, a greedy best-first search through continuous space creates a degenerate search-tree with branching factor 1. The 'hard part' of non-random sample-based methods is figuring out how to grow the tree through the C-space so that it will eventually reach $v_{\mathbf{start}}$.

Many different frameworks have been proposed for modeling the robot as a particle, and then iteratively updating its position in the representation as a function of some other variable—e.g. time[22]. All of these problems can be posed as greedy best-first search through continuous space, and therefore also as tree-generating algorithms. Common ideas assume that obstacles

[22] Again, 'time' here is unrelated to time in the real-world.

have a repulsive effect on the point-robot, while the goal(s) have an attractive effect. Obviously, the resulting point-robot path through the representation is dependent on the kernel function used to calculate repulsiveness or attractiveness. A common formulation uses a force field to model the world (i.e. along the same lines of an electrostatic or gravitational field). Others use velocity, acceleration, scent, etc. Collectively, these types of algorithms are known as *potential-field* methods [Khatib, 1986,Koren and Borenstein, 1991,Murray and Little, 1998].

As might be expected, many potential field methods suffer from local optima that can cause the algorithm to fail. For instance, the point-robot may get stuck oscillating between two points, or fall into equilibrium on the edge of an obstacle. Possible solutions include:

1. Introducing a degree of random noise into the system.
2. Defaulting to a random-walk, temporarily, if it can be determined that the algorithm is not making progress.

Obviously, neither of these addresses the root of the problem. In my opinion, these shortcomings have caused the robotics community to move away from potential field methods, especially for global planning, in favor of algorithms with better performance guarantees.

1.5.14 Analytical Methods

I have devoted the bulk of section 1.5 to path-planning methods that produce discrete paths. A discrete path can be thought of as a bread-crumb trail leading from the start to the goal. Obviously, in a discrete representation of the world, this is the only type of solution that is possible. However, in a continuous C-space it is sometimes possible to describe a continuous path as a collection of curves [LaValle, 2006]. The simplest path of this type is a parametric curve, where the position of the robot is defined by a separate function in each dimension of the C-space. More complex representations might use a set of functions—for instance $f_{k,l}()$ is used when $k \leq p < l$, for some parameter p.

Analytical solutions are not often used for global path-search. In practice, it is computationally expensive (and sometimes even mathematically impossible) to find a set of curves that will satisfy the constraints of nontrivial problems[23]. Nonetheless, there are a few common applications of analytical methods. For instance, the practice of calculating possible vehicle trajectories, given the state of the robot and a set of steering angles [Goldberg et al., 2002]. This type of solution is often welded to the front of a discrete path, creating a hybrid path that respects the real-world constraints imposed by momentum, acceleration, and/or non-holonomic vehicle dynamics [Carsten et al., 2007]. In these systems it is assumed that a new path will be found before the robot has reached the end of the analytical section of the hybrid path.

1.5.15 Discussion on representations and path-planning algorithms

The representation system that is chosen to model the environment goes hand-in-hand with the planning system that will operate on top of it. If the representation is discrete and deterministic, and connectivity information exists about the relationships between different states, then graph-search algorithms can be used directly. On the other hand, if a continuous representation is used, then the path-planning problem can sometimes be solved analytically to yield a continuous solution. The latter, however, is rarely done in practice due to computational complexity. If a continuous solution cannot be calculated, then a graph- and/or sampling-based approach must be used.

One option is to use combinatorial methods to create a graph in continuous space; and then use a graph-search technique to find a path through that graph. Although this transforms the problem into a discrete graph, in special cases it is still possible to guarantee paths that are optimal with respect to the real world[24]. For example, if the STAR algorithm [Lozano-Perez, 1983] is used to create the graph from a geometric model of an environment (and the environment consists only of obstacles and free-space), then an optimal graph-search algorithm can be used to find the desired path. The complexity of creating a discrete graph guaranteed to produce optimal real-world paths is exponential in the number of C-space dimensions. Therefore, it is impractical to use this approach in high dimensions. Also, although the complexity of the STAR algorithm is only linear in the number of points used to describe the environment, it does assumes that *points are used to describe the environment.*

[23] As a contrived example, the Navier-Stokes equations for fluid flow have yet to be completely solved in three dimensions—they are, however, often simulated via a discretization with respect to time. This is akin to the non-analytical path-search techniques I have described.

[24] In addition to being optimal with respect to the graph.

Therefore, this approach is also impractical if a point-based geometric model of the world does not exist.

Sampling-based methods are the option of choice when the C-space contains many dimensions and/or requires an implicit representation or other non-geometric model. Sampling-based methods create a discrete graph representation of the (continuous) environment. In a common implementation, the environment is sliced into non-overlapping cells. There is a fixed relationship between cells and graph nodes, and between relative cell locations and graph edges. A graph-search algorithm can be used to find a solution that is optimal with respect to the graph model of the world, but that may not be optimal with respect to the real-world. This is currently the most common technique for rovers operating in three or fewer dimensions. The method of choice in more than three dimensions is random-trees. Random trees perform graph construction simultaneously to graph-search. New nodes, created by sampling continuous points in C-space, are linked to old nodes using a predetermined procedure[25]. Paths provided by random trees can be far from optimal. However, they are often used for problems so complex that finding any solution is considered an achievement.

I have covered quite a few graph-search algorithms. I will now summarize what I consider to be the most important highlights of this discussion:

1. Guarantees on completeness are important to consider when choosing which algorithm to use. In many cases, the type of representation that is used will affect whether or not an algorithm is complete, resolution complete, probabilistic complete.

2. best-first search algorithms can be used to focus the search and decrease search time. If appropriate precautions are taken, best-first search can produce optimal results with respect to the graph—for instance, if an admissible heuristic is used in the A* algorithm.

3. 'Lifelong' algorithms modify data in the search-tree instead of performing new search from scratch. This often leads to faster re-planning calculations when map information changes but the start and goal location remain the same. When properly designed, lifelong algorithms inherit the search properties of the original algorithms they are developed from.

[25] The specific procedure is dependent on algorithm and C-space.

4. 'Dynamic' algorithms are similar to 'Lifelong' algorithms except that the start position of the robot can also change. This has allowed the speed of on-line re-planning to increase by orders of magnitude, and has contributed to the popularity of using graph-based representations for field robotics. Dynamic algorithms also inherit the search properties of the original algorithms they are developed from.

5. 'Lite' versions of dynamic algorithms do not store an explicit search-tree with pointers. Instead, they use rhs values to compare the current cost-to-goal of a node with its previously assumed best cost-to-goal. If the two are not equal, then the information in that part of the graph is updated. This propagates changes through the implicit search-tree. Paths are extracted using gradient descent, instead of the traditional method of following back-pointer through an explicit search-tree.

6. 'Field' algorithms (Field D* and fast-marching level-set methods) liberate the value-to-gaol values from considering travel only along graph edges. These techniques also allow paths to be extracted that are not constrained to nodes and/or graph edges. This allows more natural paths to be created, with respect to the real word. To date, these methods have only been applied in low dimensional (3 or fewer) C-spaces.

7. 'Random-tree' algorithms are used in complex C-spaces when other methods are computationally prohibitive. They can be implemented in many different frameworks, including those described as 'lifelong' and 'dynamic.'

8. Potential field methods and other continuous greedy best-first search techniques are theoretically equivalent to non-random-sampling-based continuous space graph-search methods. The degenerate search-tree they create has a branching factor of 1.

9. When a continuous representation is used, the 'cost' that is minimized during graph-search should reflect both the per-distance difficulty of moving between two location as well as the distance between those locations.

I have covered graph-search algorithms at this level of detail for three reasons. First, I find them interesting. Second, graph-search algorithms are extremely popular for path-planning. Many representation frameworks use graphs for no other reason than the fact that doing so allows an optimal or near-optimal path to be extracted in an intelligent way[26]. Third,

[26] i.e. with guarantees on optimality, completeness, and runtime.

understanding how graph-search algorithms work helps one to appreciate that, given the same graph, many search algorithms may produce similar or identical paths. As a result, the most straightforward way to modify the behavior of a path-planner is often to change how cost is defined in the representation. This final point is also why a large portion of this paper is devoted to looking at how machine learning is applied in the representation subsystem.

1.6 Markov Decision Processes

This subsection can be though of as a continuation of the discussion on representations (section 1.2). I have chosen to wait until this point to talk about Markov decision process because it is helpful to put them in the context of graphs and graph search algorithms.

Up to this point, all of the representations I have talked about have been deterministic—each action is assumed to have known consequences. In contrast, Markov decision processes (or *MDP*) can be conceptualized as nondeterministic graph representations of the world [Bellman, 1957]. In other words, the outcome of executing a particular action may be inherently unknowable in advance (although there may be a distribution over possible consequences).

A notion of discrete time exists in an MDP. An event is said to occur at time t, where t is an integer. Time t_a occurs after time t_b iff $t_a > t_b$. The system exists in a particular state v_i at time t. States in a MDP are analogous to the vertices in a graph. At each particular state, a set of actions are available to the system. Let $a_{k \in [i,t]}$ represent an action a_k available to the system at state i and time t. Choosing to execute a particular action at time t causes the system to transition to a particular state v_j by time $t+1$ (note that i may or may not be equal to j). Further, a reward or punishment is given to the system at each t based on its current state, where rewards/punishments of 0 are allowed.

Because the graph is nondeterministic, choosing to execute the same action in the same state may produce two different results (e.g. a different reward is given or a different state transition occurs). Further, the set of actions available at a given state may not be static. However, it is assumed the distributions of rewards, actions, and action outcomes are all static, but possibly unknown. It is also assumed that these distributions are not influenced by the system's previous actions, state visits, or rewards/punishments received—they depend only on the state that the robot is currently in (hence the 'Markov' part of the name).

From a planning point of view, an action in an MDP is essentially a marriage between the edges in a graph and the Schrödinger's cat idea[27]—choosing to execute action $a_{k \in [i,t]}$ is like simultaneously planning to traverse an entire set of edges, each with their own cost and probability of actually being traversed. There is no way to know which edge in this set will actually get used until $a_{k \in [i,t]}$ is executed. The cost graphs of section 1.5 can be considered special MDPs where there is only one possible reward and one possible edge per action, and the probability a particular action is available at a state is 1 or 0 for existent and nonexistent edges, respectively.

In a *partially observable* Markov decision process or *POMDP* the states cannot be known directly but must be inferred from observations. Given an infinite number of observations, and a means of recording previous information, the probability of knowing which state the agent is in (based on previous observations and actions) tends toward 1. Obviously, being partially observable adds an additional layer of complexity to a MDP.

Planning in a MDP (not to mention a POMDP) is a difficult problem. In most MDPs, the only type of 'path' that can be extracted is something along the lines of a most-likely scenario. Alternatively, it may be possible to describe a posterior distribution over the environment that represents the probability the robot will visit any/all states on its way to the goal. In general, the task of planning in a MDP is considered a reinforcement learning problem. Therefore, I shall discuss it in chapter 2, in the context of machine learning.

1.7 Actuation

Once a plan has been created by the planning subsystem, the actuation subsystem is responsible for actually executing the plan. In general, this involves translating the plan into an instruction sequence that is compatible with the various devices (motors, serves, switches, etc.) that the robot uses to act on the environment. As with the sensing subsystem, the actuation subsystem is highly dependent on the specific hardware available to the robot. To ensure safety and overall relevance/usefulness, any plan sent to the actuation subsystem

[27] A thought experiment originally proposed in [Schrodinger, 1935] to address the author's concerns with quantum mechanics—i.e. how does one reason about events that may or may not be occurring in the present/future, given that their occurrence depends on the unknown outcome of past nondeterminism?

must respect the latter's constraints. Therefore, the constraints of the actuation subsystem must be considered when the planning subsystem is developed.

2 Machine Learning Background

Machine learning is a field of computer science that encompasses an eclectic variety of problem domains. Many of these domains involve teaching a system to perform a task or finding the optimal solution to a complex problem (often these are one and the same). A full survey of machine learning methods is beyond the scope of this paper, but I will review a few machine learning subfields that are of particular relevance to robotics. After a general overview of these methods, I will narrow the discussion to specific algorithms that have recently[28] been applied to path-planning and/or the environmental representations used for path-planning. I have chosen this format to emphasize current trends in the application of machine learning to path-planning. The survey of machine learning presented in this chapter is by no means an exhaustive search. As previously stated in the introduction, this chapter has been included to give those uninitiated to the concept of machine learning enough background to understand chapter 3. Therefore, this chapter may seem trivial to machine learning experts, and can be skipped by the latter.

2.1 Machine learning subfields: high-level overview

Two subfields of machine learning that are of particular interest to robotics applications are supervised learning and reinforcement learning. The following two subsections give a brief high-level introduction to these problem domains, and this section is concluded with an outline of other machine learning subfields.

2.1.1 Supervised Learning

Supervised learning algorithms teach a system to discriminate between various groups or *classes* of input, based on examples of each class. The input is usually represented as a vector, where each dimension of the vector corresponding to a particular attribute or feature. For instance, a color might be represented by a vector of three values—one each

[28] Within the last five years.

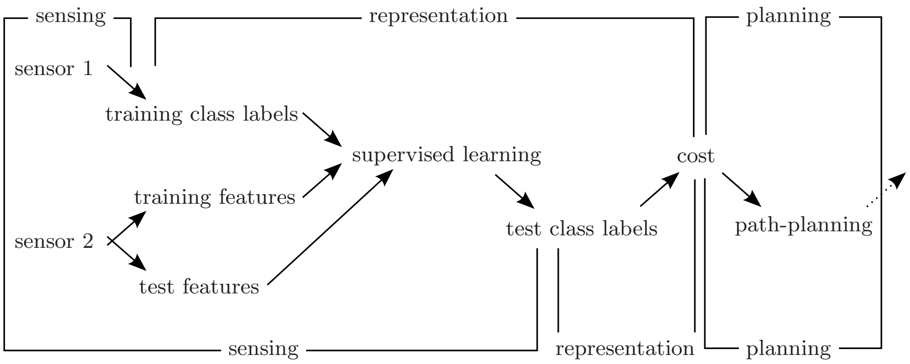

Figure 9: How supervised learning is used to provide class labels to the representation system, and ultimately cost to a path-planning algorithm. Top: as incorporated into the representation subsystem. Bottom: used to create a meta-sensor. Note that the mapping from class labels to cost is usually provided *a priori.*

for red, blue, and green. Suppose a human wishes to teach a system the difference between 'hot' and 'cool' colors. They would provide the system examples of hot and cool colors, along with the appropriate class labels—i.e. the knowledge that a particular color is either hot or cool. After a suitable number of examples have been given, the algorithm learns to label *never before seen* colors as either hot or cool. In this context, the set of examples with known labels is called the *training set* and the set of never-before-seen colors is called the *test set.*

Supervised Regression is similar to the idea described above, except that the algorithm attempts to map feature vectors to real number values instead of non-ordinal class labels. Regression is useful when an output is used as a measure.

When both the training examples and labels are provided from on-board sensors, this is called *self-supervised learning.* For example, feature vectors are provided by a camera and the corresponding labels by a depth sensor[29]. This is useful because the learned color-obstacle relationship can be extrapolated to parts of a camera image that are beyond the range of the depth sensor. Systems using self-supervised learning require the function between class labels and edge cost to be known *a priori.* In practice, this is usually accomplished by hand tuning a heuristic function. Figure 9 displays how the supervised learning algorithm described in the above example fits into a theoretical robotic system.

[29] For example, anything over 10cm high labeled as 'obstacle.'

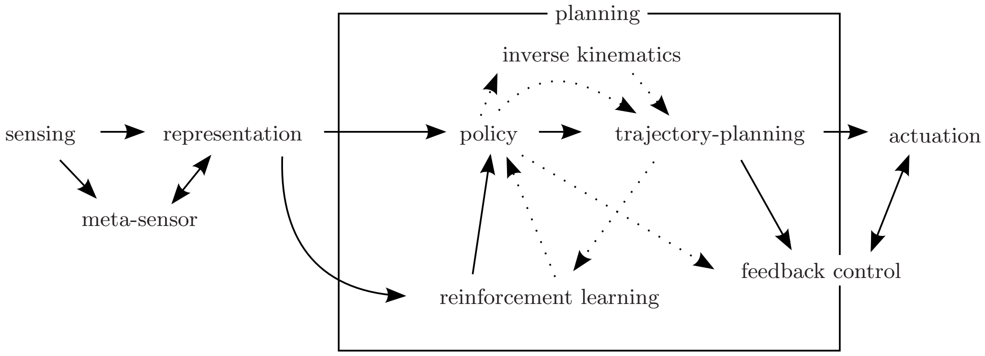

Figure 10: How reinforcement learning is used in the planning system. Note that it can be used to create a policy that acts as a path-planning/reactive-planning substitute—sitting between the representation and trajectory planning modules (solid-arrows)—and/or to refine operation within other planning modules (dotted-arrows).

2.1.2 Reinforcement Learning

Reinforcement learning algorithms teach a system appropriate actions based on the concepts of reward and punishment. The world in which the system 'lives' is assumed to be a *Markov decision process* or MDP (section 1.6). Although, depending on the specific reinforcement learning algorithm used, the MDP may or may not be explicitly modeled by the system (Algorithms that do not explicitly model the MDP are called model-free methods).

The probability distribution associated with the system executing a particular action at a particular state is called a *policy*. The system is trained to modify its policy based on rewards and/or punishments (i.e. negative rewards) it receives. In practice, it is common to compute this as a sum of the immediate reward resulting from a particular action plus a discounted sum of all additional rewards that are eventually received as a result of choosing the original action.

In *imitation learning* a system is trained to behave like an expert. Given the framework of reinforcement learning, this can be posed as a policy learning problem. The system attempts to modify its policy in such a way that the resulting behavior is most like the expert. It is called is called *inverse reinforcement learning* when the goal is to extract the particular reward function used by the expert.

Figure 10 depicts common ways reinforcement learning is applied in the planning subsystem.

2.1.3 Other Machine Learning Subfields

Other machine learning subfields that I will not cover in-depth include:

1. *unsupervised learning* models unlabeled data, often by grouping similar things together. Other terms for specific types of unsupervised learning include *clustering*, *topic modeling*, and *latent semantic analysis*.
2. *simi-supervised learning* uses a relatively small number of labeled training examples along with a relatively large number of unlabeled training examples. The goal is similar to supervised learning, except that much of the data is unlabeled.

2.2 Supervised Learning Algorithms

Although the term 'supervised learning' is sometimes used in conjunction with reinforcement learning algorithms that relay on user input, it usually denotes the use of supervised classification or regression algorithms. I have included algorithms in this section *only* if they are used by the specific systems outlined in chapter 3. There is a vast number of other supervised learning algorithms, and this section should not be viewed as an exhaustive survey.

Let x_i^{trn} be a row vector representing the ith training example and let y_i^{trn} be the known output corresponding to x_i^{trn}. Each dimension of x_i^{trn} contains a value associated with a particular feature of the ith training example—recall our toy color example where each dimension of x_i^{trn} is a value associated with a particular color plane. Depending on the machine learning technique used, y_i^{trn} may be an ordinal value—like a real number or an integer—or a non-ordinal value—like the token 'hot.' Similarly, each dimension of x_i^{trn} may contain ordinal or non-ordinal values (or a combination), depending on the specific supervised learning method used. Let x_j^{tst} be the jth test vector. A trained model will return a prediction $\hat{y}_j^{\text{tst}}$ for each x_j^{tst}. Note that $\hat{y}_j^{\text{tst}}$ is only a good guess for the actual—although unknown—value y_j^{tst}. It is often convenient to stack individual input vectors into matrices and output values into vectors. Let $\mathbf{X}^{\text{trn}}$ and $\mathbf{X}^{\text{tst}}$ be the training and test sets in matrix form, respectively; and let $\mathbf{Y}^{\text{trn}}$ and $\hat{\mathbf{Y}}^{\text{tst}}$ be the training labels and predicted labels in vector form, respectively.

Note that the ith rows of $\mathbf{X}^{\text{trn}}$ and $\mathbf{Y}^{\text{trn}}$ correspond, as do the jth rows of $\mathbf{X}^{\text{tst}}$ and $\hat{\mathbf{Y}}^{\text{tst}}$.

2.2.1 Support Vector Machines

Support vector machines (or *SVMs*) represent data x_i^{trn} and x_j^{tst} as points in a high dimensional space $\Re^d$, where d orthogonal spatial dimensions correspond to the features of x_i^{trn}. There are d features in x_i^{trn} and d dimensions in $\Re^d$. Let there be k unique class labels in $\mathbf{Y}^{\text{trn}}$ (labels are repeated as necessary). SVMs attempt to find $k-1$ high dimensional manifolds, each of local dimensionality $d-1$, that separate the data into k partitions based on the class labels. If the high dimensional manifolds are hyperplanes, then the model is called a linear SVM; otherwise, it is called a non-linear SVM. Non-linear SVMs are often implemented by first transforming $\mathbf{X}^{\text{trn}}$ using a non-linear function, and then applying a linear SVM to the transformed data.

The data points necessary to describe the manifold(s) are called *support vectors*. In the event that all examples are correctly classified, a minimum number of support vectors are needed[30]. Given a particular family of $d-1$ dimensional manifolds, it may not be possible to perfectly separate the data into k partitions. In this case, misclassified examples also become support vectors. Let x_s^{sv} denote the sth support vector and let $f(x_s^{\text{sv}})$ denote a cost associated with the misclassification of that vector. $f(x_s^{\text{sv}})$ is a function of the distance that x_s^{sv} would have to move in order to be correctly classified. The algorithm attempts to place the $k-1$ manifolds such that the sum over $f(x_s^{\text{sv}})$ of all s is minimized. For more information see [Aizerman et al., 1964, Burges, 1998, Halatci et al., 2007, Bajracharya et al., 2008].

2.2.2 Gaussian Mixture Models

As in the previous section, *Gaussian mixture models* or *GMMs* view each feature vector x_i^{trn} as a point in $\Re^d$, where d is the number of features in x_i^{trn}. Let there be k distinct possible values for y_i^{trn} (i.e. classes). GMMs create a generative model by placing k sets of d dimensional Gaussians distributions in $\Re^d$ (i.e. there is one set per class). The positions of each Gaussian is selected such that the probability of the actual data being generated by the model is maximized. In practice, each Gaussian is define by a mean and a covariance

[30] The precise number varies with d, but in the case of a line separating $\Re^2$, there are two points on one side of the line and one on the other.

matrix that describe its position and shape in $\Re^d$, respectively. Given a model, $\hat{y}_j^{\text{tst}}$ is often determined by the Gaussian distribution that has the highest probability of generating x_j^{tst}. An alternative is to say that x_j^{tst} belongs to each of the k classes with a certain degree of probability. For more information see [McLachlan and Basford, 1988, Jansen et al., 2005, Thrun et al., 2006, Angelova et al., 2007, Halatci et al., 2007].

2.2.3 Neural Networks

A neural network is a computational network model inspired by the arrangement of biological neurons in an animal brain. The network can be represented as a graph of nodes connected by edges. The edges propagate activation information from one node to another akin to how biological neurons pass electrical signals. A node is activated as a function of its incoming edge signals. In practice, the function may output binary values—e.g. 'on' or 'off,' or a range of real values that represent intermediate levels of activation. When activated, the node sends a signal along its outgoing edges. In a *feed-forward* network the graph has L layers of nodes. The nodes in layer l have directed edges to the nodes in layer l^+ where $1 \leq l < l^+ \leq L$. In more complex models the edges may be arranged arbitrarily. Levels 1 and L are called the input and output layers, respectively, and other levels are called hidden layers. Input vectors x_i^{trn} are mapped to the input layer, with the hope that a trained model will produce the desired output y_i^{trn} at the output layer. The model is iteratively trained by modifying the activation functions of nodes that lead to incorrect output, the arrangement of edges, or a combination of these. The activation level is commonly determined as a function (e.g. the hyperbolic tangent) of a weighted sum of the incoming signals. For more information see [Rosenblatt, 1962, Gurney, 1997, Happold et al., 2006, Erkan et al., 2007].

2.2.4 Adaptive Boosting

Adaptive boosting or *AdaBoost* is a meta-algorithm that iteratively calls a seperate, and usually simple, supervised learning algorithm. Each training example (x_i^{trn}, y_i^{trn}) is initially assigned an equal weight with respect to the influence it has on training the model. After each iteration, incorrectly classified examples are weighted relative more, while the correctly labeled examples are wighted relatively less. This causes the algorithm to focus its attention on the examples that are misclassified. For more information see [Freund and Schapire, 1997, Freund and Schapire, 1999, Konolige et al., 2006].

2.2.5 Histogram Methods

Histogram based methods rely on the fact that normalized histograms can approximate any probability distribution. The histogram approximation approaches the true distribution as bin width approaches 0 and the number of training examples approaches infinity. Histogram based methods usually build one histogram per class per feature dimension. The probability that x_j^{tst} belongs to a particular class is found by aggregating the in-class probabilities along each dimension. These, in turn, are found by normalizing the associated bin counts for each class (along a particular dimension) by the total number of bin counts for all classes (along that same dimension). In a 'vanilla' implementation, the influence of each dimension is weighed equally. In more complex models, these wights may be determined by a meta-algorithm or defined based on expert knowledge. More information can be found in [Nobel and Lugosi, 1994, Lugosi and Nobel, 1996, Happold et al., 2006, Grudic et al., 2007, Bajracharya et al., 2008].

2.2.6 Hypersphere Approximations

This algorithm was developed in the field robotic community by [Kim et al., 2007] for the specific task of classifying image segments as either 'obstacle' or 'traversable.' The algorithm views each feature vector x_i^{trn} as a point in $\Re^d$, where d is the number of features in x_i^{trn}. A hypersphere of radius r is defined to exist around the first example of each class ('obstacle' and 'traversable') in $\Re^d$. There is a count associated with each hypersphere that is initialized to 1. During the mission, more training examples are given to the model. If a new example exists within a hypersphere belonging to its class, then that particular hypersphere's count is incremented. If a new example exists within multiple hypershperes belonging to its class, then the count of the closest hypersphere is incremented. If a new example does not exist within a hypersphere of its class, then a new hypersphere is added to the mode,l centered on the new example. In practice, the number of hyperspheres per class can be limited to a fixed value, and old hypershperes or hyperspheres with small counts removed to accommodate new examples. $\hat{y}_j^{\text{tst}}$ is determined by choosing the K nearest hyperspheres to x_j^{tst} (as determined by the L^2 norm), then selecting the class with the highest count sum over those K hyperspheres.

An estimate of the probability of correct classification can be found by normalizing the in-class count sum over the K hyperspheres by the total count sum over the K hyperspheres.

2.3 Reinforcement Learning Algorithms

Reinforcement learning algorithms assume the world is a Markov decision process (section 1.6). An algorithm may try to infer some or all of the MDP by observing the effects of the actions it exicutes. The resulting estimation of the MDP can then be used to crate a *policy* for future decisions. These types of algorithms are known as *model-based methods*. In contrast, *model-free methods* create policies that attempt to maximize reward without modeling the underlying dynamics of the MDP.

There are many different frameworks for reinforcement learning algorithms. I have chosen to describe two common algorithms that are used by systems described in chapter 3, and two algorithms invented specifically for representation/path-planning applications. The latter describe themselves as reinforcement learning, but do not use a standard reinforcement learning template. Note that there are many other reinforcement learning algorithms I will not cover, and this section should not be viewed as an exhaustive survey of reinforcement learning algorithms. For a more comprehensive survey, I recommend [Kaelbling et al., 1996].

2.3.1 Q-learning

Q-learning is a model-free method first proposed by [Watkins, 1989] and refined in [Watkins and Dayan, 1992]. Q-learning attempts to learn a concept of delayed reward by accounting for future rewards that indirectly result from previous actions. This allows the system to avoid locally optimal actions that are suboptimal globally. Because future rewards are uncertain, they are discounted by a factor γ for each time-step into the future they are expected to occur. The algorithm is called 'Q-learning' because Q is conventionally used to represent the sum of discounted reward that is expected to be received from executing a particular action. Assuming all Q values are known, an optimal policy involves executing the action with the largest Q value during each time-step. In practice, Q values are initially unknown and must be learned through interaction with the environment.

Let $Q^*(v_i, a_{i,k})$ be the expected discounted reward of executing action $a_{i,k}$ in state v_i and then choosing all subsequent actions optimally. The *value* of v_i is the maximum expected discounted reward that can be received from v_i (i.e. assuming the best action is taken), and is denoted $V^*(v_i)$. Let the immediate reward received for executing $a_{i,k}$ at state v_i be denoted $R(v_i, a_{i,k})$. The discount factor is denoted γ, and the transition function from state v_i to state v_j is denoted $T(v_i, a_{i,k}, v_j)$ and assumed to output 1 if action $a_{i,k}$ causes the agent to perform the necessary transition between from v_i to v_j, and 0 otherwise.

$$V^*(s) = \max_k Q^*(v_i, a_{i,k})$$

Where $Q^*(v_i, a_{i,k})$ can be expressed as:

$$Q^*(v_i, a_{i,k}) = R(v_i, a_{i,k}) + \gamma \sum_{v_j} T(v_i, a_{i,k}, v_j) \max_l (Q^*(v_j, a_{j,l})$$

The expected optimal policy at state v_i is denoted $\pi^*(v_i)$ and is calculated:

$$\pi^*(v_i) = \arg\max_{a_{i,k}} Q^*(v_i, a_{i,k})$$

The update rule for the value of a state and action combination $Q(v_i, a_{i,k})$ in Q-learning is given by:

$$Q(v_i, a_{i,k}) \leftarrow Q(v_i, a_{i,k}) + \alpha \left(R(v_i, a_{i,k}) + \gamma \max_l (Q(v_j, a_{j,l}) - Q(v_i, a_{i,k}) \right)$$

Where α is a parameter that decays slowly over time. Note that Q will converge to the optimal Q^* with probability 1.

2.3.2 Prioritized sweeping

Prioritized sweeping [Moore and Atkeson, 1993] is a model-free based that keep track of possible state-transitions and rewards. The algorithm is similar to Q-learning, except that it works on state values $V(v_i)$ instead of state-action values $Q(v_i, a_{i,k})$. Prioritized sweeping only allows k changes in expected state values per iteration, and focuses changes to states that will benefit most from being changed. A priority queue is used to keep track of which states are likely to have their values significantly modified by an update. The update rule for prioritized sweeping can be expressed as:

$$V(v_i) \leftarrow \max_k \left(\hat{R}(v_i, v_{i,k}) + \gamma \sum_{v_j} \hat{T}(v_i, a_{i,k}, v_j) V(v_j) \right)$$

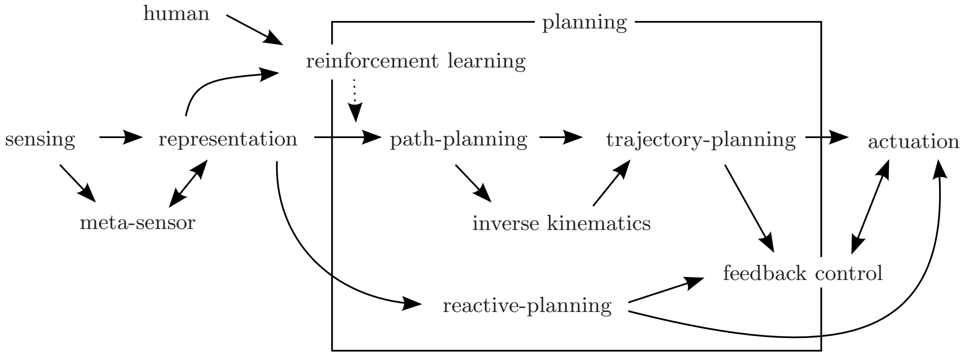

Figure 11: How reinforcement learning is employed to learn a cost-function over representation data that is then used for graph-search path-planning.

Where $\hat{R}(v_i, v_{i,k})$ and $\hat{T}(v_i, a_{i,k}, v_j)$ are the current models of the reward and transition functions. A state v_j is a *predecessor* of v_i if v_i can be reached by executing an action at state v_j. After an update takes place, the queue priorities of the predecessors of v_i are adjusted to be $\Delta V = |V_{old} - V(v_i)|$ if they are currently less than ΔV, where V_{old} was the value of state v_i before it was modified.

2.3.3 Linear Cost Function Learning

[Bagnell et al., 2006] use a technique that may be considered a hybrid of supervised learning and imitation learning to learn a graph-search cost function over map features, see Figure 11. The algorithm assumes that 'good' examples of paths through MDPs[31] are provided by an expert *a priori*. It also assumes that a loss-function exists to evaluate the similarity of two paths, given a MDP. The agent's policy is implicitly defined by the behavior of an optimal graph-search algorithm when a linear cost function over map features is used. The algorithm attempts to modify the linear cost function so that the resulting agent behavior resembles that of the expert.

Let a loss function between an agent policy produced path μ^* and the ith expert specified optimal path μ_i be defined as follows:

$$\mathcal{L}(y, y_i) = \mathcal{L}_i(y) = l_i^{\mathrm{T}} \mu$$

[31] I believe the 'MPDs' used in this algorithm are actually deterministic, and can therefore be interpreted as weighted graphs.

where l_i can be interpreted as a loss vector over state-action combinations that dictates the penalty for deviating from μ_i. The solution involves solving a quadratic program as follows:

$$\min_{w,\zeta_i} \frac{1}{2}\|w\|^2 + \frac{\gamma}{n}\sum_i \beta_i \zeta_i^q$$

such that for all i the following relationship holds:

$$w^{\mathrm{T}} f_i(y_i) \geq \max_{y \in \mathcal{Y}_i} w^{\mathrm{T}} F_i(y) + \mathcal{L}_i(y)$$

where F_i is the set of map features, $\gamma \geq 0$ is a hyper-parameter, ζ_i is a slack variable that permits violation of the constraints with a penalty that scales with γ, and q is the norm order of the slack penalty—i.e. a q of 1 or 2 denotes using the L_1 or L_2 norm, respectively. $\beta_i > 0$ is a data dependent scaler used to normalize examples of different lengths. Further simplification is achieved by only considering cases where both $F_i()$ and $\mathcal{L}_i$ are linear in the state-action frequencies μ. [Bagnell et al., 2006] note that $\mu \in \mathcal{G}_i$ expresses the Bellman-flow constraints for each Markov Decision Process. This and further simplifications are used to transform the problem into the following quadratic program:

$$\min_{w,\zeta_i,u_i} \frac{1}{2}\|w\|^2 + \frac{\gamma}{n}\sum_i \beta_i \zeta_i^q$$

such that for all i the following holds:

$$w^{\mathrm{T}} F_i \mu_i + \zeta_i \geq s_i^{\mathrm{T}} u_i$$

and also such that for all i, x, and a the following holds:

$$u_i^x \geq (w^{\mathrm{T}} F_i + l_i)^{(x,a)} + \sum_{x'} p_i(x'|x,a) u_i^{x'}$$

Where $p_i()$ is a transition probability. Finally, the the program is optimized for efficiency by using a hinge-loss. ζ_i are tight:

$$\zeta_i = \max_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu - w^{\mathrm{T}} F_i \mu_i$$

and the problem can be expressed as a single convex cost function:

$$c_q(w) = \frac{1}{n}\sum_{i=1}^{n} \beta_i \left(\max_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu - w^{\mathrm{T}} F_i \mu_i \right)^q + \frac{\lambda}{2}\|w\|^2$$

Although not differentiable, this can be solved using 'sub-gradient' descent. The sub-gradient of the objective function g_w^q is given by:

$$g_w^q = \frac{1}{n}\sum_{i=1}^{n} q\beta_i \left((w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu^* - w^{\mathrm{T}} F_i \mu_i \right)^{q-1} \cdot F_i \Delta^w \mu_i + \lambda w$$

where $u^* = \arg\max_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu$ and $\Delta^w \mu_i = \mu^* - \mu_i$. Finding the sub-gradient requires calculating:

$$u^* = \arg\max_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu$$

for each MDP (i.e. training example). However, this is equivalent to solving the MDPs while using $(w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})$ as the cost function. In other words, the final step can be solved using A* or an equivalent method (see section 1.5) to find the least cost path through each training example.

2.3.4 Non-linear Cost Function Learning

[Ratliff et al., 2007] builds directly on [Bagnell et al., 2006] (previous section). The idea is to 'boost-in' new features by using the output of a simple classifier, where the classifier is trained to differentiate between: (1) correct parts of the expert specified path *not* followed by the agent and (2) incorrect parts of the agent's path. The risk function $R()$ is similar to the cost function $c_q()$ of the previous section, and expressed as:

$$R(w) = \frac{1}{n} \sum_{i=1}^{n} \beta_i \left(w^{\mathrm{T}} F_i \mu_i - \min_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i - l_i^{\mathrm{T}})\mu \right) + \frac{\lambda}{2} \|w\|^2$$

The sub-gradient is given by:

$$g_w^q = \frac{1}{n} \sum_{i=1}^{n} F_i \Delta^w \mu_i + \lambda w$$

F_i is the set of features associated with example i, including the learned classification outputs. Also note that:

$$u^* = \arg\min_{\mu \in \mathcal{G}_i} (w^{\mathrm{T}} F_i + l_i^{\mathrm{T}})\mu$$

and $\Delta^w \mu_i = \mu^* - \mu_i$. The algorithm iteratively adds additional classifier generated features until the resulting behavior of the agent is 'good enough' on the expert provided example maps. The program must be solved at each iteration because the size of F_i is modified to include the new classifier labels, based on the agent's mistakes during the previous iteration.

3 Applications of Machine Learning to Path Planning

Robotic systems have many parameters that need to be tuned. The tuning task has traditionally been delegated to a human, and accomplished through a process of trial-and-error.

Unfortunately, hand-tuned parameters tend to make a system brittle. Even if a particular set of parameters works well in one application, it may not easily transfer to other applications. The main argument for using machine learning in robotics is that it makes a system more robust by allowing system parameters to be adjusted automatically—thus improving performance. This has the additional advantage of reducing the effort required by the robot's human masters.

Recall the theoretical robotic system described in chapter 1. It consists of four general subsystems, interacting as follows:

$$\text{sensor} \rightarrow \text{representation} \rightarrow \text{planning} \rightarrow \text{actuation}$$

In this chapter I will work through the robotic system, describing how machine learning has been applied in each part. As previously stated, the focus of this paper is on how machine learning is used for path-planning and in the representations used for path-planning. Therefore, only in-depth discussion is provided on these topics. An important point to keep in mind while reading this chapter is: *there is no 'standard' way to include machine learning in a robotic system.* Different systems use different methods to transform sensor data into actuation commands. Machine learning can be applied at many different points along the way.

3.1 Machine Learning in Meta-Sensors

Simple machine learning ideas might be applied to raw sensor data to help improve its accuracy or to help optimize sensing equipment [Kalman, 1960, Kltagawa, 1996, Parker et al., 2005]. However, the most common application of machine learning in the sensing subsystem is as part of a meta-sensor. Recall from section 1.1 that a meta-sensor uses software to provide higher-level data than might be expected from a simple sensor (for instance, a person detector [Rowley et al., 1998, Haritaoglu et al., 1998, Bellotto and Hu, 2009]). Because a meta-sensor draws on data stored in the representation to help make its decisions, it could alternatively be considered part of the representation subsystem:

$$\text{sensor} \rightarrow (\text{meta–sensor} \leftrightarrow \text{representation}) \rightarrow \text{planning} \rightarrow \text{actuation}$$

Machine learning has been used in robotics to create meta-sensors from image space data in [Jansen et al., 2005, Happold et al., 2006, Konolige et al., 2006, Thrun et al., 2006, Erkan et al., 2007,Grudic et al., 2007,Ollis et al., 2007,Halatci et al., 2007]. Recall from section 1.1 that an image consist of spatially related raw sensor readings[32].

In [Jansen et al., 2005] Gaussian mixture models (section 2.2.2) are used to classify terrain as ‘sand,’ ‘gravel,’ ‘grass,’ ‘foliage,’ or ‘sky’ in image-space. Image features are pixel-color planes that have been adjusted to account for gamma correction in the camera. Training example labels are provided off-line by a human, and cross validation is used to determine the optimal number of Gaussians per model. A separate GMM is built for each type of terrain the robot is expected to encounter (e.g. desert, forest, marshland), and an additional meta-gausian-model is used to determine which environment the robot is currently in.

In [Happold et al., 2006] histogram methods (section 2.2.5) are used on-line in image-space to learn a mapping from color to geometric classes. Class labels are provided from stereo disparity. More recent work by the same authors [Ollis et al., 2007] focuses on learning the probability that pixels are associated with the ‘obstacle’ class.

[Konolige et al., 2006] explore two different self-supervised frameworks in image-space. The first is a path detection technique that uses a Gaussian mixture model (section 2.2.2) in conjunction with AdaBoost (section 2.2.4) over decision stumps[33]. At start up, the system makes the optimistic assumption that it is on a path. The terrain in front of the robot is sampled and used to create a GMM in image space to distinguish between ‘path’ from ‘not path.’ Next, all pixels in the image are labeled according to this classification. If the resulting label pattern has a path-like shape, then the robot concludes that it is, in fact, on a path. Finally, this labeling is used with AdaBoost to create a decision stump that determines ‘path’ vs. ‘not path’ for subsequent images. The system repeats this process after every meter of movement. The second method presented in [Konolige et al., 2006] is similar to the first, except that near-field stereo is used to provide AdaBoost with ‘traversable’ vs. ‘obstacle’ examples. AdaBoost creates a decision stump that is used to classify the remaining image and/or subsequent images.

[Thrun et al., 2006] outlines the system that won the DARPA Grand Challenge in 2005. The system uses a Gaussian mixture model (section 2.2.2) to learn a mapping from color

[32] An image exists between the sensor and representation systems (or is in both of them at the same time), and in my opinion is the quintessential meta-sensor input data.

[33] A simple binary classifier that uses a single feature to make a decision. The feature chosen is the one expected to maximize accuracy.

(red, green, blue) to traversability. Class labels are provided by on-board laser sensors. As new training examples are provided, new ‘local’ Gaussians are created and then added to the ‘global’ model. This is done by either modifying the model’s previous set of Gaussians, or discarding them in favor of the new ’local’ Gaussians. The former option is chosen over the latter if the mahalanobis distance between a new and old Gaussian is less than a predefined threshold.

In [Erkan et al., 2007] a neural net (section 2.2.3) is trained off-line from log-files[34] and used to extract low dimensional texture features from normalized YUV space image patches. A scaled image pyramid is also used to normalize the image-space appearance of near- and far-field information.

A hierarchical Gaussian mixture model (section 2.2.2) is used in [Angelova et al., 2007] to classify terrain as belonging to one of many different classes (e.g. ‘sand,’ ‘soil,’ ‘mud,’ etc.). Features used include: average RGB colors, color histograms, and texture filters. Classification is performed in image space, and separate meta-classifiers are used for the near- and far-field. Training occurs off-line.

Our lab’s early work is presented in [Grudic et al., 2007]. This technique uses a collection of histograms (section 2.2.5) to learn a mapping from near-field color and texture data to ‘obstacle’ and ‘traversable’ classes in image space. A confidence in the prediction is also provided, allowing the image to be labeled as ‘obstacle,’ ‘traversable,’ or ‘unknown.’ Features include normalized RGB color values and 1D histograms of non-normalized RGB values over image patches. A single-class classifier[35] is created per class. Each classifier uses a 1D histogram of values from the first principal component of the training set. Note that the training set matrix is transposed from its usual form for this calculation—each column is a training example, and each row is a feature dimension. If a stereo disparity image sensor labels an image patch as belonging to a particular class, but the associated patch vector is in the null space of the training set basis, then the new vector is added to the training set. The histograms are also updated such that labeled examples are always more likely to belong to the appropriate class.

[34] Data recorded from an actual mission that is saved for off-line use. Log-files typically record all sensor readings and may also contain initialization information necessary to duplicate the robot’s state at the beginning of the mission.

[35] e.g. the classifier answers the question: does a training example belong to a particular class c.

A two level classification method is used in [Halatci et al., 2007] to classify terrain (e.g. as either 'rock,' 'sand,' or 'mixed'). The lower level classifier is either a support vector machine (section 2.2.1) or a Gaussian mixture model (section 2.2.2), and the high-level classifier is simply a fusion of many low-level classifiers in a naive Bayesian framework. Each low level classifier has an associated probabilistic belief per class, these are combined to yield an overall class probability per image patch. Image space feature vectors include color, texture, and range data.

A common trend among the aforementioned techniques is that supervised learning is used in image-space to map either raw color intensity, texture, or both into one or more classes. There are two general types of output class label: (1) outputs are representative of terrain features types (lake, road, etc.), and (2) outputs represent a notion of traversability (obstacle, traversable, unknown). When idea 1 is used, it is assumed that a 'down-stream' sub-system will be able to use the terrain labellings to generate cost or that the labellings can otherwise be used for path-search. With idea 2, the meta-sensor data can be used directly in the representation for path-search. The latter fact has inspired us and others to perform path-search directly in image-space [Otte et al., 2007, Ollis et al., 2008, Huang et al., 2009, Otte et al., 2009].

The above techniques can also be grouped according to how/when they are trained. Training possibilities include:

1. Off-line from human labeled data.
2. On-line using other sensors (self-supervised learning)
3. Off-line using other sensor data that has been recorded in log-files.

Off-line training allows a system to have access to mountains of data. It also removes most time constraints. The main disadvantage of off-line training is that a model may not perform well when the testing environment is not accurately reflected in the training examples. On-line learning is more likely to train models that reflect the current state of the environment. However, training must be done in real-time, which can limit the complexity of the model being used. These issues are reflected by trends in the techniques described above. Methods involving on-line training use simple models such as histograms and Adaboost [Happold et al., 2006,Grudic et al., 2007,Konolige et al., 2006], while models trained off-line use more complex models such as neural networks and cascades of Gaussians [Erkan et al., 2007,Angelova et al., 2007, Jansen et al., 2005]. Standard Gaussian mixture models are used in both on-line and off-line techniques; however, when used on-line they are either incorporated into a start-up procedure that happens before the robot starts moving [Konolige et al., 2006] or are given ample computational resources [Thrun et al., 2006].

3.2 Machine Learning in the Representation

When a cell-based metric map is cast as a connected graph for the purposes of graph-search path-planning, it is expected that each edge has a cost associated with it.

$$\text{sensor} \rightarrow \text{map–features} \rightarrow \text{cost} \rightarrow \text{path–search}$$

In the previous section, I examined several techniques in which meta-sensors output class labels such as 'obstacle' vs. 'traversable terrain.' For graph search, these correspond to cost values of 1 and ∞, respectively. Meta-sensors may also output classes labels like 'lake,' 'field,' and 'road,' for which a notion of cost is less obvious. In this section, I examine how machine learning has been used to obtain cost from the latter type of class labels (as well as other map features).

$$\text{sensor} \rightarrow \text{features} \rightarrow \text{labels} \rightarrow \text{cost} \rightarrow \text{path–search}$$

The mapping between sensor/map data and cost often involves two or three steps—for instance, depth and color data from image space are mapped to height and vegetation data in Cartesian space, which are then mapped to cost either directly or indirectly via class labels.

$$\text{sensor} \rightarrow \text{image–features} \rightarrow \text{image–classes} \rightarrow \text{cartesian–features} \rightarrow \text{cartesian–classes} \rightarrow \text{cost} \rightarrow \text{path–search}$$

The particular information stored in a map is method dependent. It may include anything from raw data (e.g. color, height, slope) to derived features (e.g. texture, obstacle probability). All of the methods described in this section assume that map features are stored in a feature vector at each map grid. Depending on the method, training and test sets (used for supervised learning) can either be constructed directly from image-space features, or by projecting and accumulating this data in Cartesian space.

3.2.1 Supervised and Self-Supervised Learning in the Representation

[Happold et al., 2006] use a two step method to map color to cost via geometric features. A neural network (section 2.2.3) is trained off-line from log-files and learns a non-parametric function of geometric features in Cartesian space (e.g. height variation of points at a grid location, terrain slope between mean grid values, etc.) to four different geometric cost classes. Histogram methods are used on-line in image-space to learn a mapping from color to the geometric classes. Class labels are provided from stereo disparity. This framework allows for a stable geometric feature map to exist in parallel to changing notion of color vs. geometry.

While color features are generally per pixel, texture features are found over pixel sets. [Kim et al., 2007] compare two different ways of creating these pixel sets, and evaluate the utility of each when used in conjunction with the 'hypersphere approximation' algorithm they developed (section 2.2.6). The two types of pixel sets are called *pixel patches* and *superpixels*. The former are created by splitting the image into a grid of small square sub-regions. This has the disadvantage of artificially grouping dissimilar parts of the image into a common set. Superpixels, on the other hand, are created by over-segmenting an image, based on the feature vectors it contains. This paper uses a version of the min-cuts algorithm for over-segmentation [Shi and Malik, 2000]. Color and texture features include: the RGB and HSV color spaces, histograms of Hue and Saturation, and 18 texture filters. Two classes are defined (i.e. 'obstacle' and 'ground'). However, uncertain labellings are not classified as one or the other. Thus, the final labeling consists of 'obstacle,' 'ground,' and 'unknown.' Two experiment are performed—the first involves manually labeled log-file data, and the second uses stereo-labeled near-field data. Superpixels are found to outperform pixel patches in both experiments.

[Erkan et al., 2007] use a two step method to learn a mapping from image appearance to three classes ('obstacle', 'traversable', 'occluded'), via a low dimensional texture representation. A neural net (section 2.2.3), trained off-line from log-files, is used to extract low dimensional texture features from normalized YUV space image patches. A 'scaled image pyramid' is also used to normalize the image-space appearance of near- and far-field information. The texture data is accumulated in a top-down Cartesian quad-tree map[36], and a log-linear regression model is trained on-line to map texture features to terrain classes. The texture values in

[36] A specific type of cell based map.

a given quad of the quad-tree are found by examining the ratios of all feature vectors that have been placed in that particular quad.

[Angelova et al., 2007] use a hierarchical Gaussian mixture model (section 2.2.2) classify terrain into one of many different classes (e.g. ‘sand,’ ‘soil,’ ‘mud,’ etc.). The hierarchy is organized in a cascade of increasing GMM complexity, such that classifications achievable with relatively few discriminating feature dimensions are applied first, followed by those requiring an increasing number of feature dimensions. The idea here is to get as much classification accomplished as easily as possible. The confusion matrix (obtained from test data) is interpreted as a connected graph. The structure of the meta-classifier is recursively determined by finding the min-cut of that graph. This breaks the original problem into two sub-problems that can be solved independently. Features used include: average RGB colors, color histograms, and texture filters. Classification is performed in image space, and separate meta-classifiers are used for the near- and far-field. The resulting Cartesian map is post-processed by neighborhood voting to minimize noise induced errors. Although not stated in the paper, I believe that class labels are provided by a human expert.

Linear support vector machines (section 2.2.1) are used in conjunction with histogram methods in [Bajracharya et al., 2008] to learn a two-step mapping from color data to the classes of ‘traversable’ and ‘obstacle’ terrain. Statistics are accumulated about terrain geometry in a top-down Cartesian grid map. These feature vectors (one each per map cell) are classified as using a histogram model (section 2.2.5) that has been previously trained via expert human tele-operation of the robot. The labellings provided by the histogram method are also stored in a Cartesian map; however, they are used to label pixels in the current image. This is done by following pointers from the top-down map back to the image-space pixels that generated the geometric feature data. Next, an SVM is trained on the current image, given the resulting image space labeling, and used to classify far-field terrain. The geometric features accumulated in the top-down map include a combination of pixel color and 1D color histograms over image patches.

3.2.2 Reinforcement Learning in the Representation

All of the methods described in the previous section have one thing in common: the mapping between class label and cost must be defined by a human. One could speculate that this is because, once the class of a map cell is known (e.g. lake vs. road), it is fairly straightforward for a human to assign 'common-sense' cost values that will perform well in practice. On the other hand, one could also make the argument that this limits a system and burdens a human with the task of determining which classes and/or features are relevant for path-search.

In this subsection I examine two reinforcement learning approaches that can be used to obtain a mapping from features and/or classes directly to map cost [Bagnell et al., 2006,Ratliff et al., 2007].

$$\text{sensor} \rightarrow \text{features} \rightarrow \text{cost} \rightarrow \text{path–search}$$

The mathematical details of these two approaches are presented in sections 2.3.3 and 2.3.4, respectively.

The training input is assumed to be a set of feature vector grid maps, along with a path through each map provided by an expert. In both papers, the system's policy is determined by the actions of an optimal graph-search algorithm (section 1.5), assuming cost is defined by a (learned) function over feature vectors. The policy is evaluated (and the cost-function modified) based on how well the resulting path matches the expert provide example. Note that this does not fit the standard reinforcement learning template. The policy itself is only implicitly defined in terms of the cost function over features.

Specific feature vectors (i.e. x_i^{trn}) now exist at every node in the MDP. Further, a single training instance includes an entire MDP (complete with feature vectors), along with the 'correct' transition probabilities at each state—as determined by an expert. In practice, the expert draws an optimal path through the MDP, and the transition probabilities are defined as a function of this path. For example, each node in the path has a probability of 1 of transitioning to the next node in the path, and all other transition probabilities are 0.

The system attempts to find the mapping from feature vectors to rewards such that the resulting behavior is similar to that of the expert. This is framed as the minimization of a cumulative loss function that measures how far the system's policy is from the expert's, and also posed according to the *maximum margin* framework. That is, optimal behavior should be better than another solution by a margin that scales with the loss-function of the latter solution. Once found, the mapping between features and cost is used on-line; however, training must be performed off-line because it requires expert provided example paths. The method also assumes the existence of a predefined metric to calculate reward/penalty based on the difference between the expert's path and an 'optimal' path given the system's policy.

The authors of [Bagnell et al., 2006] perform experiments with two different types of input feature-maps. The first is RGB color and the second is elevation. Although not stated in the paper, the penalty function appears to resemble a 1D Gaussian curve at each cross-section of the path, and cumulative penalty is used to rate a given policy. The path generated from the system's policy is found using the A* algorithm (section 1.5.6). Cost is a linear function of feature vector components, and the learning algorithm attempts to find the optimal weighting factor for each term.

The work of [Ratliff et al., 2007, Ratliff et al., 2009] build directly on [Bagnell et al., 2006]. The former extend the linear function of feature vector components of the latter by allowing non-linear cost functions. Sub-gradient descent is used in a 'space of cost functions' as opposed to the fixed linear parametrization of the earlier work.

The model is trained iteratively. At each iteration, a classifier is taught to distinguish between features associated with: (1) parts of the expert path not visited by the system, and (2) parts of the policy generated path that should not have been visited. Labellings from this classifier are then added as an additional component to all feature vectors in the map. This causes the system to actively focus it attention on areas of the map for which its 'policy' is incorrect. Terminal node classification trees are used as the classifier, and the algorithm is limited to using a fixed number of trees. The authors present several experiments in which their algorithm is shown to be effective. These include using RGB data, RGB data plus height and slope data, and also the high dimensional C-space of a quadruped robot.

Although the ideas presented in this subsection do not eliminate the human involvement entirely, they do change the human's task from fiddling with low-level cost parameters to providing high-level example paths of good behavior. This means that the human does not need to know the low-level specifics of the features used in the representation (i.e. slope, elevation, etc.), as long as they can can demonstrate a 'good' path. The main disadvantage of these techniques is the requirement of *a priori* training examples and reward/punishment functions. Obviously, this means that training examples are assumed to be similar to the test environments the robot will operate in. On the other hand, one might argue that any alternative supervised or reinforcement learning based system will also require some form of training example that is assumed to be similar to the test case.

3.2.3 Machine Learning in Representation: Recap

I have talked about two general frameworks for using machine learning in the representation subsystem of a robot. These are summarized as follows:

1. Supervised or self-supervised learning is used to obtain a mapping from sensor provided feature vectors to class labels. Map cost is assigned as a predefined function of class label.
2. Reinforcement learning is used to obtain a function between map features and cost. The cost function is constructed such that optimal graph-search paths through training maps are similar to expert provided examples.

In practice, both of these methods make assumptions about the problem domain. Method 2 assumes that maps exist of environments similar to the one in which the robot will be deployed *a priori*. It also assumes that an expert has provided path examples through these maps, and that a metric exists to evaluate the quality of agent created paths. Method 1 assumes that a means of distinguishing classes is available. With supervised learning, this is accomplished via human-labeled training examples, while in self-supervised learning the examples are provided by a 'ground truth' sensor—i.e. an instrument with the ability to determine traversability regardless of environmental appearance (such as stereo vision or lasers). Given known environmental features, both supervised and self-supervised learning models can be trained off-line with human labeled data or with log-files. However, when missions are guaranteed to occur in completely unknown environments, on-line self-supervised learning is the only option. Method 1 also assumes that a cost function over classes is provided.

In principal, both frameworks 1 and 2 can be used in the same system at the same time, although this has not yet been done in practice. Such a system might be organized as follows: self-supervised learning finds a mapping from sensor features to class labels, while reinforcement learning (trained off-line) calculates the function between class labels and cost.

In another configuration reinforcement learning might learn a function between geometric features and cost off-line, and on-line self-supervised learning could be used to find the mapping between image features and geometric features on-line. The latter is similar to the methods outlined in [Happold et al., 2006] and [Erkan et al., 2007], both of which use neural networks to learn a mapping between texture and class labels. Both of these currently determine cost with a predefined function.

Regardless of how machine learning is used, there are a number of design decisions that influence the architecture of the representation and planning subsystems. The first of these is: *in what coordinate space are the feature vectors accumulated and/or the models applied?* We have seen a variety of different ideas including:

1. The features are collected and the models applied in image space [Jansen et al., 2005, Thrun et al., 2006, Konolige et al., 2006, Angelova et al., 2007, Grudic et al., 2007, Halatci et al., 2007].
2. The features are collected and the models applied in Cartesian space [Bagnell et al., 2006, Ratliff et al., 2007].
3. Features from image space are accumulated in Cartesian space, while models using these features are applied in image space [Happold et al., 2006, Erkan et al., 2007, Bajracharya et al., 2008].

Option 1 is most widely used for self-supervised learning because it operates in image space—the native coordinate space of the training examples—and therefore minimizes projection calculations and error. Option 2 has only been used for reinforcement learning; however, it could potentially be used in other applications. Option 3 accounts for the fact that image appearances change as a function of distance, but requires more overhead than either 1 or 2.

Another decision is: *How will cost be determined?* Again, there are a variety of approaches that have been tried:

1. Appearance → class label → cost. [Jansen et al., 2005, Thrun et al., 2006, Konolige et al., 2006, Angelova et al., 2007, Grudic et al., 2007, Halatci et al., 2007]

2. Appearance → class label → geometric features → cost. [Happold et al., 2006, Erkan et al., 2007, Bajracharya et al., 2008]

3. (Appearance + geometric features) → cost. [Bagnell et al., 2006, Ratliff et al., 2007].

Methods that store geometric features (or classes based on geometric features) tend to do so in a Cartesian map, while methods that do not use geometric features tend to store features in image space. The reinforcement learning framework has not been applied in image space.

A final consideration is: *to what degree can the model be trained off-line?*

1. No off-line learning [Thrun et al., 2006, Konolige et al., 2006, Angelova et al., 2007, Grudic et al., 2007, Halatci et al., 2007].

2. Partial off-line learning [Happold et al., 2006, Erkan et al., 2007, Bajracharya et al., 2008].

3. Only off-line learning [Jansen et al., 2005, Bagnell et al., 2006, Ratliff et al., 2007].

Methods that use geometric features or classes based on them tend to learn these geometric features off line. On-line learning is used to find the appearance vs. geometric relationship. Techniques requiring human labeling must be performed off-line. In the papers I have surveyed in this subsection, non-geometric self-supervised systems use only on-line learning.

3.3 Learning the Representation

Learning the representation (i.e. the representation *itself*) is a different problem than using machine learning in the representation (previous subsection). In this subsection I will *briefly* outline the former. In my opinion, most of this topic consists of the research body known as *simultaneous localization and mapping* or *SLAM* [Smith et al., 1986]. The basic idea can be summarized as follows: in the absence of robust localization sensor (e.g. GPS), a robot in unknown terrain must relay on other observations to both: (1) construct a model of the environment and (2) localize itself within this model. Most SLAM algorithms are applied in either an outdoor field-robotic setting or on indoor mobile robots. In either case, the relative positions of landmarks[37], with respect to each other and/or the robot, are used to recover the organization of the world.

From a path-planning point of view, SLAM is an alternative (or improvement) to a naive GPS based mapping framework. One could even argue that SLAM is a meta-sensor that periodically outputs the entire organization of the environment. Although parameters in SLAM could probably be tweaked to affect the behavior of the path-planning component, this is inadvisable. In general, a good SLAM algorithm is one that recreates the environment as accurately (and quickly) as possible in the representation. This will allow the planning system to make informed decisions—regardless of the planner's implementation details, and regardless of whatever *type* of information is contained in the representation. The field of SLAM is quickly becoming mature and the body of literature is vast. Interested readers are encouraged to study the survey by [Durrant-Whyte and Bailey, 2006].

Other forms of Learning the representation combine elements of SLAM with prior knowledge about the structure of the environment. For instance, [Koenig and Simmons, 1996a, Koenig and Simmons, 1996b] assume that a topological map of the world is provided *a priori* and use a partially observable Markov decision process (section 1.6) to learn the distances between states—effectively creating a metric topological map. As with SLAM, these methods are primarily concerned about the accuracy of the environmental representation.

3.4 Machine Learning in the Planning Subsystem

Given that the name of this paper is 'machine learning applied to robotic path-planning,' one might expect this to be the largest section of the paper. This is not the case (although it not the smallest either). Part of the reason for this, as discussed in section 1.4, is that often the most direct way to modify the behavior of a planning system is to alter the data in the representation. For example, it is relatively easy to apply black-box machine learning algorithms to the representation subsystem in order to generate more intelligent map features—and thus facilitate better overall performance. In contrast, it is difficult to improve the behavior of tried-and-true graph-search algorithms—especially when the latter have theoretical guarantees on optimality, completeness, and convergence.

[37] Depending on the specific implementation 'landmarks' may consist of a variety of different things, their main requirement is that robust and repeatable recognition is possible.

There are many important machine learning application in the greater planning system that do not involve paths. Further, it is easy to confuse some of these other applications with path-planning. For this reason, I begin this section with discussions on machine learning applied to *non*-path-planning parts of the planning subsystem, before moving on to the surprisingly few actual applications of machine learning in path-planning algorithms.

3.4.1 Reinforcement Learning for Non-Path-Planning Parts of the Planning subsystem

Robotics is a natural application domain for Reinforcement learning, however reinforcement learning is seldom applied to the *path-planning* sub-problem. I have included this section to highlight where reinforcement learning has been applied elsewhere in the planning subsystem for two reasons: (1) to address why reinforcement learning has not been more widely applied in the narrow application of path-planning, and (2) to highlight other parts of the planning subsystem where reinforcement learning has been embraced.

A robot is a natural parallel for the agent in reinforcement learning algorithms. Depending on the ability of the system to accurately determine its location in the configuration space via sensor readings, the world can either be represented as a Markov decision process or as a partially observable Markov decision process (section 1.6). Finally, both robots and agents are expected to perform actions in their respective worlds using whatever information is available to them [Stronger and Stone, 2008]. In summary, it is straightforward to use an MDP as the representation and then use reinforcement learning to train the robot at a specific task.

In practice, the MDP is not always represented as a discrete graph, but instead allowed to exist in continuous space. Actions affect the location of the robot in the continuous space and lead to reward/punishment as normal. Alternatively, a continuous environment can be discretized (section 1.2.4) in order to create a standard MDP. Regardless of the MDP's structure, the configuration space of the robot used to create the MDP tends not to be in the form of an environmental map. Instead, it usually exists in the high dimensional space of the system's degrees-of-freedom.

The primary reason reinforcement learning has not been widely adopted for path-planing is due to uncertainty about the consequences of one's actions in an MDP. As previously mentioned in section 1.6, this makes it impossible to generate a typical path-plan. In other words, if the system is unsure of where it will end up after executing a particular action, why should it waste effort trying to account for all possible alternatives? Path-*like* output that can theoretically be generated about an MDP includes both the most-likely path and a probability field denoting how likely any state is to be included in a path to the goal.

From a human's perspective, the most-likely path basically provides the same sort of information as a standard graph-path[38]. This is especially true when the robot is in a dynamic and/or unknown environment. In these cases, even an optimal graph-path will probably change as new information about the world is discovered during the mission. So, given this inherent limitation, why not use reinforcement learning to find a most-likely path and then use that for navigation?

The answer is that this *is* theoretically possible. However, The question itself is slightly irrelevant. We need to take a step back, and examine the role that path-planning plays in the larger robotic system. The only reason that the robot needed a path, in the first place, was so that it could eventually reach the goal. Given a graph representation of the world, this is done by creating a search-tree. When the search reaches the robot (staring from the goal) then we have enough information to determine the 'best' way to get the robot to the goal. Specifically, by moving along the path extracted by following back-pointers through the graph. Alternatively, once the search-tree has grown to the robot's current position, we could just as easily move the robot locally using the information in the last branch of the search-tree[39] instead of explicitly extracting the entire path. Further, if we let the search-tree expand to cover the entire graph, we could easily calculate the 'optimal' behavior at every position in the C-space. This is akin to a reinforcement learning policy of actions over states. Recall that a graph is essentially a MDP where the single consequence and reward of each action occurs with probability 1 (section 1.6).

Reinforcement learning creates a policy that tells the system what the 'best' action is at any given state. If the policy of the robot has been built with respect to a goal reaching problem, then the robot will behave in such a way as to maximize its chances of reaching the goal.

[38] i.e. found using the graph-search techniques of section-1.5, and assuming a graph representation of the environment.

[39] Or in a 'Lite' algorithm, using one step of gradient descent over cost-to-goal or cost distance-to-gaol values.

Therefore, a path is not required when a standard reinforcement learning approach is used. This, combined with the fact that any path through an MDP is only a 'best guess' at what will actually happen, is why reinforcement learning is not generally used to extract complete paths directly from an MDP.

Paths provide useful high-level information that a policy cannot. In model-free reinforcement learning, the robot cannot communicate why it choses a specific action with respect to the larger task (reaching the goal), only that it thinks it is a good choice. Model-based methods can provide more information, but figuring out why the robot chose a particular action can still be unclear. Because of this, it is easier for a human to evaluate a path (vs. policy) to determine if the system is behaving intelligently or not. For example, it is difficult to evaluate if a policy will keep the system in an infinite loop, but this is obvious when a path-plan is available. Reinforcement learning also requires many training iterations per environment, and standard methods require identical training and test environments[40]. The policy created in one world might not (and probably will not) generalize well to new environments.

On the other hand, the ability to generalize to new environments is of little concern when the robot lives in a finite world (e.g. a manipulator in a factory), and is trying to figure out the best way to do something that will be repeated over-and-over. Also, most graph-search path-planning algorithms do not have a principled way to deal with inherent uncertainty in the structure of the graph. Many algorithms allow the structure of the graph or reward functions to change from one deterministic configuration to another, as long as the graph and actions are deterministic long enough to plan and execute actions (sections 1.5.7 - 1.5.13). However, when the environment is inherently non-deterministic, reinforcement learning is the better choice[41].

It is also important to realize that achieving a goal is only one application of reinforcement learning, which can also be used to accomplish many tasks that path-planning cannot. Potential applications include problems with environmental dynamics too difficult to model and tasks that cannot be defined as trying to reach a point in C-space (e.g. balancing a stick). Therefore, more natural robotic applications of reinforcement learning include reac-

[40] The nonstandard reinforcement learning frameworks presented in sections 2.3.3 and 2.3.4 do not require identical test and training environment; however, these methods do assume that training and test environments are similar.

[41] Reinforcement learning does assume the non-determinism is constant or 'slowly changing' (i.e. the probabilities of consequences and rewards vs. actions don't change [much] over time).

tive planning (see section 1.2) and inverse kinematics (section 1.3.2). I shall now give a few examples of this type of work. Obviously, this is not an exhaustive survey.

In [Wheeler et al., 1992] Q-learning (section 2.3.1) is used to train an agent how to grip an object—or modify its grip on an object. The idea is that certain manipulator planning problems involve grasping an object in a way that does not interfere with the eventual completion of the task. An analogy is drawn to toddlers grasping a spoon in order to eat applesauce. If the toddler/robot places its grip around the bowl of the spoon, then completing the task is impossible without an adjustment. The state space is a simple MDP of the world and actions include 'grasping,' 'swapping,' and 'insertion.

[Wiering et al., 1999] use a hybrid of CMACs [Albus, 1975] and prioritized sweeping (section 2.3.2) to train a simulated soccer team consisting of 1 or 3 players. The C-space of the model has 16 to 24 degrees-of-freedom, depending on the number of players, and including a boolean vector indicating who is in possession of the ball. The states of an MDP are created via a discretization of the C-space. The actions at each state include: 'go-forward,' 'turn to ball,' 'turn to goal,' and 'shoot.' The authors find a favorable comparison between their work and related previous work.

[Bagnell and Schneider, 2001] use reinforcement learning[42] to train an autonomous helicopter controller not to crash. The world is represented as a partially observable Markov decision process. Related work by [Ng et al., 2003] enables a helicopter to hover in place and also to perform a variety of competitive maneuvers taken from a radio controlled helicopter competition. This is extended in [Ng et al., 2006] where the focus is on low-speed flight and sustained inverted flight.

[Kohl and Stone, 2004] use policy gradient reinforcement learning [Sutton et al., 2000] to train a Sony Aibo robot for fast locomotion. A continuous MDP over the C-space of the robot is used. The robot is able to increase its speed from 245 millimeters per second (mm/s) to 291 mm/s. [Fidelman and Stone, 2004] also use policy gradient reinforcement learning to teach a Sony Aibo robot how to capture a miniature soccer ball. Results show that the robot is able to increase the successful capture rate to 64% from a 36% capture rate obtained with a human programmed controller.

[42] A model based policy learning method of their own design.

3.4.2 Supervised Learning for Non-Path-Planning Parts of the Planning subsystem

This section is a continuation of the (incomplete) survey of machine learning applied to reactive planning (see section 1.2) and inverse kinematics (section 1.3.2) that started at the end of the previous section. This section focuses on supervised learning algorithms.

In [Gomez et al., 2006] neural nets (section 2.2.3) are used in a reactive control framework to teach a robot how to balance a hinged pole on top of a wheeled cart. The cart itself exists on a finite length of track. Four models were used for an experiment space spanning dimensions for: (1) the number of poles (one and two), and (2) the observability of the environment (complete vs incomplete state information). The authors find that, with regard to cpu requirements, their idea compares favorably against 14 other machine learning methods including various reinforcement learning approaches.

Recurrent neural networks (section 2.2.3) are used in [Mayer et al., 2006] to teach knot tying to surgical suture robots. This problem is approached from an inverse kinematics point of view. That is, given a set of known trajectories that the end of the string must follow to tie a knot, how should the robot move based on its current configuration.

Neural networks (section 2.2.3) are also used for a different inverse kinematics problem in [Grasemann et al., 2008]. This work focuses on enabling the Sony Aibo robot to follow a desired trajectory through its C-space, despite an inaccurate model of how the PID controller effects the actuators.

[Infantes et al., 2006] uses a Dynamic Bayes Net in conjunction with an expectation maximization algorithm to learn how parameters in the planning and representation systems should be modified in response to environmental observations. The parameters include things like obstacle radius and robot speed, while environmental observations include things like geometry, how cluttered the environment is, and the minimum distance to the nearest obstacle.

[Howard et al., 2006] use a supervised learning method of their own design to teach a robot cost functions associated with the inverse-kinematics problem (section 1.3.2). That is, given multiple C-space solutions that correspond to a specific path through the real-world, a cost is used to determine which are more desirable than others.

A large (although somewhat dated) survey of genetic algorithms applied to reactive control can be found in [Ram et al., 1994].

3.4.3 Learning and Task-Planning in Discrete Space

Task-planning[43] is concerned with determining the sequence actions must be executed in to accomplish a larger task. Often the actions are high level concepts such as 'travel to location' or 'pick up object.' There is a large body of work that focuses on training robotics systems how to order tasks in this way [Billard et al., 2007]. Task-planning relies on other ideas such as learning from demonstration, imitation learning, and human to robot skill transfer. Many task-planning systems represent the state of the world as a connected graph. Actions are represented by edges, and nodes correspond to word states. The idea is to 'teach' the robot which path through the graph corresponds to accomplishing a specific task. It is easy to see that this can be posed as a path-planning problem—which is why I have included it here.

In task-planning applications, the primary challenges include: (1) getting the system to know what actions are available and (2) making the system understand which actions are involved in a specific task. Problem 1 is often addressed by assuming the system has a preexisting library of actions [Nicolescu and Mataric, 2003, Veeraraghavan and Veloso, 2008]. In some systems, the library can be extended by explicitly showing the robot how to perform a particular action [Chernova and Veloso, 2009]. Problem 2, getting the robot to recognize which actions are involved in a larger task, has been accomplished in a variety of ways including:

1. Passive observation of a human performing the task [Kuniyoshi et al., 1994].
2. Following a human as they perform the task [Rybski et al., 2007].
3. Interacting with a human as they perform the task [Nicolescu and Mataric, 2003, Chernova and Veloso, 2009].
4. Listening to instructions given via human speech [Rybski et al., 2007].

[43] Task-planning is sometimes referred to as *policy learning*. In order to avoid confusion with with reinforcement learning, I will not use the latter term.

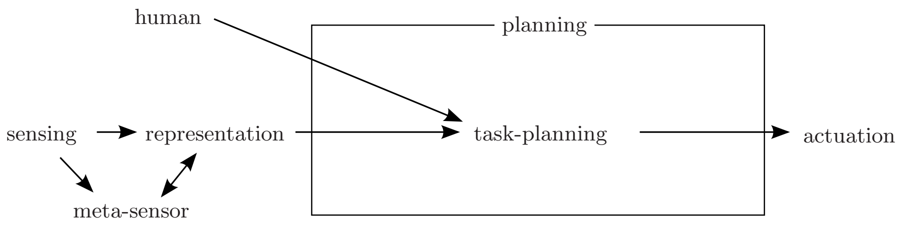

Figure 12: How task-planning is used in a robotic system.

5. Human tele-operation of example tasks [van Lent and Laird, 2001].

Regardless, the robot must decide which parts of the example tasks can be broken down into smaller primitive actions, and then how to string these actions together. Figure 12 displays how task-planning fits into a theoretical robotic system.

It is important to realize that most of the 'machine learning' that takes place in these systems consists of breaking a demonstrated task into pieces that can be encoded in a symbolic language. Thus, in this domain the terms 'learning' and 'planning' are arguably synonymous with 'memorization' and 'regurgitation,' respectively. In general, task-planning systems can perform complex action sequences, as long as they receive a demonstration or description of the sequence they are supposed to preform in advance. For this reason, these methods are predominately used in applications that require an agent to perform a specific task (or a set of similar tasks) over-and-over.

3.4.4 Learning and Task-Planning in Continuous Space

Techniques have been developed that extend task-planning ideas to continuous configuration spaces. A common application involves teaching a manipulator the best C-space path to follow, given a number of examples provided via human demonstration [Yeasin and Chaudhuri, 2000]. The most naive techniques are 'memorize and regurgitate' approaches, where the system simply duplicates the example actions. More complex techniques may use a level of abstraction between demonstrated behaviors and the underlying intent of the teacher. For instance, [Calinon and Billard, 2007] uses a Hidden Markov Model [Baum and Petrie, 1966, Rabiner, 1989] over pose recognition in conjunction with the forward kinematics (section 1.3.2) of a manipulator, enabling the system to move along paths demonstrated by a human arm. The robot is able to duplicate tasks such as drawing on a white-board with a marker.

More advanced systems use elements of machine learning to combine multiple examples into a more robust understanding of the problem. For instance, when teaching manipulators how to execute a specific task, [Calinon et al., 2007] first uses principle component analysis [Pearson, 1901] to project the configuration space into a low-dimensional representation. Next, Gaussian Mixture Regression [Sung, 2004] is used to estimate the probabilities that points in this low dimensional projection are associated with the desired path. The most likely path through the representation is used as the archetypal path for that task.

Note, however, that once a particular path is created, it is used without modification whenever that task is executed. Thus, path-planning does not happens on-line, but actually occurs as a preprocessing step. Similarly, no autonomous path-planning occurs in systems that simply reproduce human provided examples. Regardless, either type of approach is useful when a system is expected to perform the same sequence of actions over-and-over, but may not be the best option when the robot needs to extrapolate from examples to solve new problems.

3.4.5 Reinforcement Learning for Path-Planning

In [Grudic et al., 2003] a POMDP (section 1.6) is used to model the world. At any state, the mobile robot has three different path-planning 'behavior' actions it can choose from: follow potential field, avoid obstacle, and recover from collision. The system is trained to switch between these behaviors, based on the most-likely state the robot is in. The most-likely state is calculated from the robot's observations. Although machine learning is not used to create the path, it is used to determine how the path will be created.

3.4.6 Other Machine Learning for Path-Planning

Recall that analytical solutions can be found to the path-planning problem when the representation is a continuous C-space (section 1.2.4). Recent work has used support vector machines (section 2.2.1) to find this analytical solution [Miura, 2006]. Note that, although a machine learning algorithm is used, 'learning' is not being done in the classical sense. There

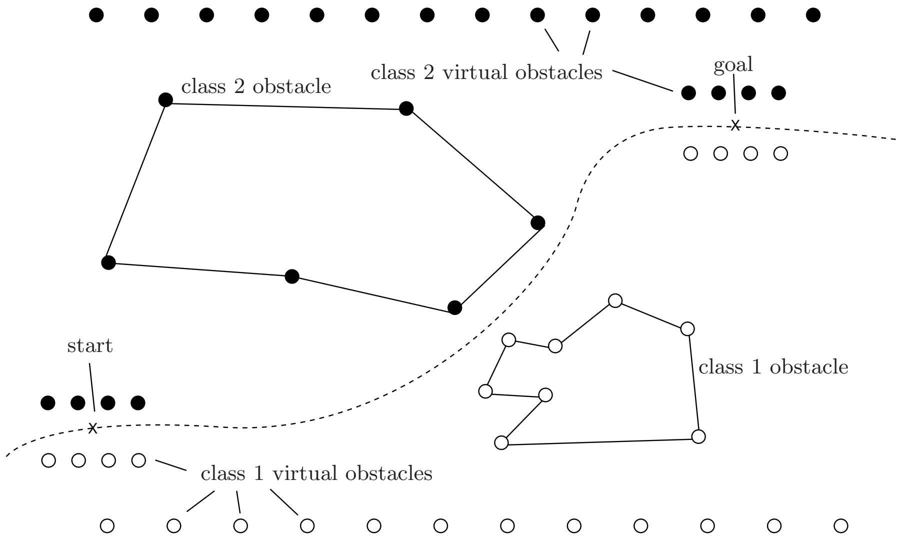

Figure 13: A support vector machine is used to find an analytical path-planning solution in continuous space. Points corresponding to class 1 and class 2 are white and black, respectively. The algorithm requires that points from a particular obstacle are all associated with the same class. It also requires virtual obstacles to be place around the start and goal to constrain the ends of the path, and around the bounds of the environment to constrain intermediate path points.

are no testing examples and the model itself is output as the path description.

Recall from section 2.2.1 that support vector machines seek to find a dividing surface that maximally separates two classes. If the C-space is in two dimensions, then the dividing surface is a curve. Note that this is precisely what we need to describe a continuous path through a continuous C-space. Assuming we desire a path between the robot and goal that avoids obstacles in a 2D configuration space, and assuming there are only two obstacles in the world (and we have a set of points describing each of them), then SVM can be used for path planning as follows:

1. Assign points from one of the obstacles to class 1 and points from the other obstacle to class 2.

2. Set 'virtual obstacles' from both classes around the start and goal points (see Figure 13).

3. Set 'virtual obstacles,' from either class 1 or class 2, around the bounds of the environment.

4. Train a support vector machine to separate the two classes.

The resulting model will contain a curve that travels from (about) the start to (about) the goal while avoiding the obstacles[44].

There are quite a few things that must be taken into consideration when using this method. First, the 'virtual obstacles' around the start and goal are necessary to constrain the resulting surface in such a way that it is useful to the problem. However, little discussion is provided in [Miura, 2006] as to how this should be achieved. It seems clear that, in order for a linear SVM to converge, the orientation of the virtual obstacles must respect the natural layout of the two actual obstacles[45]. It is possible that this could be solved by placing 'virtual obstacles' from both classes directly on the start and goal locations; however, this idea was not addressed in [Miura, 2006]. Similar considerations must be addressed for the virtual obstacles that define the bounds of the environment—which I assume are necessary to keep the separating curve from ballooning outside the area of interest.

Another concern is what to do when more than two obstacles exist. The method presented in [Miura, 2006] performs a number of searches, randomly assigning obstacles to either classes during each search, and then choosing the overall path with minimal length. This seems reasonable in environments that are mostly free space and/or contain convex obstacles, but I believe that it would run into problems in maze-like environments. The authors vaguely address these considerations by saying that an appropriate (e.g. non-linear) kernel function could be used to change a hard problem into an easier one. However, it seems unlikely that the specific kernel function needed to produce good behavior in a complex world could be known *a priori*. The main selling point of the algorithm is that it can produce a continuous path while using a geometric model of the world as input.

A 3D version of the algorithm is also presented. It requires planes of 'virtual obstacles' to exist around the start and goal and also at the world boundaries. It also requires an additional step to deal with the fact that the learned model is now a surface, but a curve is

[44] Assuming that such a path is able to exist—i.e. the goal is not cut-off from the robot by obstacles.

[45] For instance, if the robot must travel west to the goal and obstacle class 1 is north of obstacle class 2, then the virtual obstacles should be placed near the start and goal in such a way that class 1 is north of class 2.

required for a path. This is achieved by locally fitting a plane to the dividing surface where the "plane is determined by searching the space of the pitch and the roll angles for the best plane which minimizes the sum of the distances between the plane and the surface." And then "project[ing] the nominal line onto the plane and select[ing] a point on the projected line" [Miura, 2006]. Which appears to suggest that only a *discrete* path can be found when the C-space contains more than 2 dimensions.

Overall, I think this is a creative application of a machine learning algorithm to the path-planning problem. However, it makes assumptions about obstacles that are only valid in simple environments, and requires the placement of inconvenient 'virtual obstacle' points to constrain the resulting models appropriately.

3.4.7 Machine Learning in the Planning System: Recap

I began this section with a discussion on the relationship between reinforcement learning and path planning (section 3.4.1). The major points to remember are: (1) a reinforcement learning policy serves the same purpose as path-planning, but the two are not the same thing. (2) A policy deals with the nondeterministic nature of an MDP while a path operates on a deterministic graph. (3) Although a policy may be implicitly influenced by global information, it may not be able to communicate why—with respect to global information, it made a certain local decision. Paths, on the other hand, are explicitly created with respect to global information.

Section 3.4.1 concluded with an incomplete survey on how reinforcement learning has been applied in non-path-planning parts of the planning system—specifically, for inverse kinematics and reactive planning. This survey was continued for supervised learning in section 3.4.2.

Section 3.4.3 focused on discrete task-planning in robotic systems. Task-planning occurs in a graph where nodes are high-level task primitives (i.e. pick up block, move to door, etc.), and the idea is to find a specific path through this graph that represents an entire high-level task (i.e. build car, get products to Spain, etc.). Section 3.4.4 extended these ideas to the actual continuous C-space used by the robot. System from both subsections rely on human demonstration and perform more memorization than learning. The learning that does occur happens as a preprocessing step, and the resulting paths are used repeatedly without modification. Task-learning is useful because it allows a robot to be programmed at a very high level—enabling even laymen to participate in human-to-robot skill transfer.

Section 3.4.5 looked at how reinforcement learning has been used to select which member of a set of path-planners should be used, based on environmental observations. The various path-planners interact as the path is being created. Another way of interpreting this work is that the MDP is actually deterministic (i.e. a weighted graph), and reinforcement learning is used to pick a best-first-search rule at every step of a best-first search.

I finished with a discussion on how support vector machines have been used to generate an analytical solution to the path-planning problem in continuous space (section 3.4.6). The method is theoretically interesting, because it allows a continuous path to be found through a geometric model of the environment. However, its use may be limited to relatively simple environments.

4 Conclusions

I have surveyed a collection of cutting-edge applications of machine learning to robotic path-planning and path-planning related concepts. Chapter 1 presented a high-level overview of a robotic system, with a focus on the representation and planning subsystems (sections 1.2-1.6). An in-depth survey of graph-search algorithms was also included in section 1.5. Chapter 2 provided a high-level overview of machine learning, with a focus on supervised learning and reinforcement learning (section 2.1). A handful of specific algorithms were also discussed (sections 2.2-2.3). Chapter 3 looked at specific ways machine learning has been used in robotic systems, with a focus on concepts directly related to path-planning. Particular attention was given to machine learning in the following areas:

1. meta-sensors (section 3.1).
2. representation data (section 3.2).
3. representation structure (section 3.3).
4. inverse kinematics and reactive planning (sections 3.4.1 and 3.4.2).
5. discrete and continuous task-planning (sections 3.4.3 and 3.4.4).

6. path-planning algorithms (sections 3.4.5 and 3.4.6).

Machine learning can be applied at many different places in the robot system, and there is no standard way it is used for path-planning. Although machine learning has been used inside path-planning algorithms themselves, this idea has not yet been widely embraced. Much of the reason for this is that many graph-search algorithms provide optimal or near-optimal solutions with respect to the representation. Thus, as the representation becomes more accurate with respect to the environment, graph search can produce solutions that are optimal or near-optimal with respect to the real-world. In practice, most designers have opted to embrace graph-search, and then use machine learning to make the representation and sensing subsystems more intelligent—indirectly improving path-planning with respect to the real-world.

Machine learning has been used in the representation and sensing subsystems to create meta-sensors that provide the system with more useful information about the environment than raw sensor data and/or map features. Other ideas have focused on training a graph-search cost-function from expert provided examples. Finally, the accuracy of the representation itself can be improved by calculating the most likely organization, given sensor observations.

The idea of task-planning is similar to path-planning, except that discrete paths represent complete task sequences built from high-level task primitives. C-space paths are also used, but rely on human demonstration and cannot be used to infer the correct behavior for never-before-seen tasks. All of the learning that takes place in task-planning happens in a preprocessing step, and paths are not modified on-line.

Classical reinforcement learning is a machine learning native path-planning alternative that excels in nondeterministic environments. It is often used in the planning system in the context of a reactive planner or inverse kinematics. A reinforcement learning policy serves the same function as a path-planning path—both allow the robot to decide what its next action should be, given its state. However, reinforcement learning can be used for tasks that have other goals than those defined by a C-space coordinate. On the other hand, reinforcement learning requires a relatively large amount of training time to learn a policy for a specific environment. Therefore, it is used more for systems that are destined to live in a known static environment. Path-planning is better equipped to handle unknown and/or changing environments, but assumes that actions are deterministic. It also has the ability to explicitly communicate how local behavior fits into a long-term goal-directed plan.

References

Aizerman, M., Braverman, E., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (cmac). *Dynamic Systems, Measurement and Control*, pages 220–227.

Angelova, A., Matthies, L., Helmick, D., and Perona, P. (2007). Fast terrain classification using variable-length representation for autonomous navigation. In *Computer Vision and Pattern Recognition*.

Bagnell, J. A., Ratliff, N. D., and Zinkevich, M. A. (2006). Maximum margin planning. In *International Conference on Machine Learning*.

Bagnell, J. A. and Schneider, J. G. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings of International Conference on Intelligent Robotics and Automation*.

Bajracharya, M., Tang, B., Howard, A., Turmon, M., and Mathies, L. (2008). Learning long-range terrain classification for autonomous navigation. In *International Conference on Intelligent Robots and Automation*, pages 4018–4024.

Baum, L. E. and Petrie, T. (1966). Statistical inference for probabilistic functions of finite state markov chains. *Annals of Mathematics and Statistics*, 37:1554–1563.

Bellman, R. (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, 6.

Bellotto, N. and Hu, H. (2009). Multisensor-based human detection and tracking for mobile service robots. *IEEE Transactions on Systems, Man, and Cybernetics*, 39:167–181.

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2007). *Handbook of Robotics: Chapter 59, Robot Programming By Demonstration*. MIT Press, Cambridge, MA.

Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. MIT Press, Cambridge, MA.

Burges, C. J. C. (1998). A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167.

Calinon, S. and Billard, A. (2007). Learning of gestures by imitation in a humanoid robot. In *Imitation and Social Learning in Robots, Humans, and Animals:Behavioural, Social and Communicative Dimensions*, pages 153–177.

Calinon, S., Guenter, F., and Billard, A. (2007). On learning, representing and generalizing a task in a humanoid robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 37:286–298.

Carsten, J., Rankin, A., Ferguson, D., and Stentz, A. (2007). Global path planning on board the mars exploration rovers. In *IEEE Aerospace Conference*.

Chernova, S. and Veloso, M. (2009). Interactive policy learning through confidence-based autonomy. *Journal of Artificial Intelligence Research*, 34:1–25.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, Cambtidge, MA.

Davis, H., Bramanti-Gregor, A., and Wang, J. (1988). The advantages of using depth and breadth components in heuristic search. *Methodologies for intelligent systems*, 3:19–28.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. In *Numererical Mathematics*, volume 1, pages 269–271.

Durrant-Whyte, H. and Bailey, T. (2006). Simultaneous localization and mapping (slam): Part i the essential algorithms. *Robotics and Automation Magazine*, 13:99–110.

Erkan, A. N., Hadsell, R., Sermanet, P., Ben, J., Muller, U., and LeCun, Y. (2007). Adaptive long range vision in unstructured terrain. In *International Conference on Intelligent Robots and Systems*, pages 2421–2426.

Featherstone, R. (2007). *Rigid Body Dynamics Algorithms*. Springer.

Ferguson, D., Kalra, N., and Stentz, A. (2006). Replanning with rrts. In *IEEE International Conference on Robotics and Automation*.

Ferguson, D. and Stentz, A. (2006a). Anytime rrts. In *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5369–5375.

Ferguson, D. and Stentz, A. (2006b). Using interpolation to improve path planning: The field d* algorithm. *Journal of Field Robotics*, 23:79–101.

Ferguson, D. and Stentz, A. (2007). Anytime, dynamic planning in high-dimensional search spaces. In *Proc. IEEE International Conference on Robotics and Automation*, pages 1310–1315.

Fidelman, P. and Stone, P. (2004). Learning ball acquisition on a physical robot. *International Symposium on Robotics and Automation*.

Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55.

Freund, Y. and Schapire, R. E. (1999). A short introduction to boosting. *Journal of Japanese Society for Artificial Intelligence*, 15.

Goldberg, S. B., Maimone, M. W., and Matthies, L. (2002). Stereo vision and rover navigation software for planetory exploration. In *IEEE Aerospace Conference Proceedings*, Big Sky, MT, USA.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2006). Efficient non-lenear control through neuroevolution. In *Proceedings of the European Conference on Machine Learning*.

Grasemann, U., Stronger, D., and Stone, P. (2008). A neural network-based approach to robot motion control. In *RoboCup 2007: Robot Soccer World Cup XI*, pages 480–487.

Grudic, G., Mulligan, J., Otte, M., and Bates, A. (2007). Online learning of multiple perceptual models for navigation in unknown terrain. In *International Conference on Field and Service Robotics (FSR'07)*, Chamonix, France.

Grudic, G. Z., Kumar, V., and Ungar, L. (2003). Using policy gradient reinforcement learning on autonomous robot controllers. In *Proceedings of International Conference on Intelligent Robots and Systems*.

Gurney, K. (1997). *An Introduction to Neural Networks*. Routledge, London.

Halatci, I., Brooks, C. A., and Iagnemma, K. (2007). Terrain classification and classifier fusion for planetary exploration rovers. In *IEEE Aerospace Conference*.

Hansen, E. A. and Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297.

Happold, M., Ollis, M., and Johnson, N. (2006). Enhancing supervised terrain classification with predictive unsupervised learning. In *Robotics: Science and Systems*.

Haritaoglu, I., Harwood, D., and Davis, L. S. (1998). W4w: A real-time system for detection and tracking people in 2.5d. *Lecture Notes in Computer Science*, 1406/1998:877.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. In *Proc. IEEE Transactions On System Science and Cybernetics (SSC-4)*, pages 100–107.

Hellerstein, J. L., Tilbury, D. M., and Parekh, S. (2004). *Feedback Control of Computing Sustems*. John Wiley and Sons.

Howard, M., Gienger, M., Goerick, C., and Vijayakumar, S. (2006). Learning utility surfaces for movement selection. In *Proceedings of the IEEE International Conference on Robotics and Biomimetics*.

Huang, W. H., Ollis, M., Happold, M., and Stancil, B. A. (2009). Image-based path planning for outdoor mobile robots. *Journal of Field Robotics*, 26:196–211.

Infantes, G., Ingrand, F., and Ghallab, M. (2006). Learning behaviors models for robot execution control. In *Proceedings of European Conference on Artificial Intelligence*.

Jansen, P., van der Mark, W., van den Heuvel, J. C., and Groen, F. C. (2005). Colour based off-road environment and terrain type classification. In *IEEE Conference on Intelligent Transportation Systems*.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82:35–45.

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *The International Journal of Robotics Research*, 5(1):90–98.

Kim, D., Oh, S. M., and Rehg, J. M. (2007). Traversability classification for ugv navigation: A comparison of patch and superpixel representations. In *International Conference on Intelligent Robots and Systems*, pages 3166–3173.

Kltagawa, G. (1996). Monte carlo filter and smoother for non-gaussian nonlinear state space models. *Journal of Computational and Graphical Statistics*, 5:1–25.

Koenig, S. and Likhachev, M. (2002). Improved fast replanning for robot navigation in unknown terrain. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'02)*.

Koenig, S. and Simmons, R. G. (1996a). Passive distance learning for robot navigation. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 266–274.

Koenig, S. and Simmons, R. G. (1996b). Unsupervised learning of probabilistic models for robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2301–2308.

Kohl, N. and Stone, P. (2004). Policy gradient reinforcement learning for fast quadrupedal locomotiont.

Konolige, K., Agrawal, M., Bolles, R. C., Cowan, C., Fischler, M., and Gerkey, B. (2006). Outdoor mapping and navigation using stereo vision. In *Experimental Robotics: The 10th International Symposium*, pages 179–190.

Koren, Y. and Borenstein, J. (1991). Potential field methods and their inherent limitations for mobile robot navigation. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'91)*.

Kuniyoshi, Y., Inaba, M., and Inoue, H. (1994). Learning by watching: Extracting reusable task knowledge from visual observation of human performance. *IEEE Transactions on Robotics and Automation*, 10:799–822.

Latombe, J.-C. (1999). Motion planning: A journey of robots, molecules, digital actors and other artifacts. In *The International Journal of Robotics Research*, volume 18, pages 1119–1128.

LaValle, S. and Keffner, J. (2001). Rapidly-exploring random trees: Progress and prospects. In *Algorithmic and Computational Robotics: New Directions*, pages 293–308.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press, Cambridge.

Lee, D. C. (1996). *The Map-Building and Exploration Strategies of a Simple Sonar-Equipped Mobile Robot*. Cambridge University Press, New York.

Likhachev, M. and Koenig, S. (2001). Incremental a*. In *Proceedings of the Neural Information Processing Systems*.

Lozano-Perez, T. (1983). Spatial planning: A configuration approach. In *IEEE Transactions on Computers*, volume C-32, pages 108–120.

Lugosi, G. and Nobel, A. (1996). Consistency of data-driven histogram methods for density estimation and classification. *Annals of Statistics*, 24:687–706.

Mayer, H., Gomez, F., Wierstra, D., Nagy, I., Knoll, A., and Schmidhuber, J. (2006). A system for robotic heart surgery that learns to tie knots using recurrent neural networks. In *Proceedings of the International Conference on Intelligent Robotics and Systems*.

McLachlan, G. J. and Basford, K. E. (1988). *Mixture Models: Inference and Applications to Clustering*. Marcel Dekker.

Minguez, J. and Montano, L. (2004). Nearness diagram (nd) collision avoidance in troublesome scenarios. *IEEE Transactions On Robotics and Automation*, 20:45–59.

Miura, J. (2006). Support vector path planning. In *Proceedings of International Conference on Intelligent Robots and Systems*.

Moore, A. W. and Atkeson, C. G. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130.

Murray, D. and Little, J. (1998). Using real-time stereo vision for mobile robot navigation. In *Proc. of the IEEE Workshop on Perception for Mobile Agents*, Santa Barbara, California.

Ng, A. Y., Coates, A., Diel, M., Ganapathi, V., Schulte, J., Tse, B., Berger, E., and Liang, E. (2006). *Autonomous Inverted Helicoptor Flight via Reinforcement Learning*, volume 21.

Ng, A. Y., Kim, H. J., Jordan, M. I., and Sastry, S. (2003). Autonomous helicoptor flight via reinforcement learning. In *Advances in Neural Information Processing Systems*.

Nicolescu, M. N. and Mataric, M. J. (2003). Natural methods for robot task learning: Instructive demonstrations, generalization and practice. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*.

Nobel, A. and Lugosi, G. (1994). Histogram classification using vector quantization. *Proceedings of International Symposium on Information Theory*, page 391.

Ollis, M., Huang, W. H., and Happold, M. (2007). A bayesian approach to imitation learning for robot navigation. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems*, page 714.

Ollis, M., Huang, W. H., Happold, M., and Stancil, B. A. (2008). Image-based path planning for outdoor mobile robots. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'08)*.

Otte, M., Richardson, S., Mulligan, J., and Grudic, G. (2007). Local path planning in image space for autonomous robot navigation in unstructured environments. In *International Conference on Intelligent Robots and Systems (IROS'07)*, San Diego.

Otte, M., Richardson, S., Mulligan, J., and Grudic, G. (2009). Path planning in image space for autonomous robot navigation in unstructured environments. *Journal of Field Robotics*, 26:212–240.

Parker, R., Hoff, W., Norton, V., Lee, J. Y., and Colagrosso, M. (2005). Activity identification and visualization. In *Proceedings of the Fifth International Workshop on Pattern Recognition and in Information Systems*, pages 124–133.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2.

Philippsen, R. (2006). *A light formulation of the E* interpolated path replanner*. Autonomous Systems Lab, Ecole Polytechnique Federale de Lausanne.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected pplications in speech recognition. *Proceedings of the IEEE*, 77:124–133.

Ram, A., Arkin, R., Boone, G., and Pearce, M. (1994). Using genetic algorithms to learn reactive control parameters for autonomous navigation. *Adaptive Behavior*, 2:277–304.

Ratliff, N. D., Bradley, D., Bagnell, J. A., and Chestnutt, J. (2007). Boosting structured prediction for imitation learning. In *Advances in Neural Information Processing Systems*.

Ratliff, N. D., Silver, D., and Bagnell, J. A. (2009). Learning to search: Functional gradient techniques for imitation learning. In *Submitted to: Autonomous Robotics Special Issue on Robot Learning*.

Rosenblatt, F. (1962). *Principles of Neurodynamics*. Spartan Books.

Rowley, H. A., Baluja, S., and Kanade, T. (1998). Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:23–38.

Rybski, P. E., Yoon, K., Stolarz, J., and Veloso, M. M. (2007). Interactive robot task training through dialog and demonstration. In *Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction*, pages 49–56.

Schrodinger, E. (1935). Die gegenwartige situation in der quantenmechanik. *Naturwissenschaften*, 23.

Sethian, J. A. (1996). A fast marching level-set method for monotonically advancing fronts. *Proc. Nat. Acad. Sci.*, 93:1591–1595.

Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and MAchine Intelligence*, 22:888–905.

Smith, R., Self, M., and Cheeseman, P. (1986). Estimating uncertain spatial relationships in robotics. In *Proceedings of the Second Annual Conference on Uncertainty in Artificial Intelligence*, pages 435–461.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'94)*.

Stentz, A. (1995). The focussed D* algorithm for real-time replanning. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*.

Stronger, D. and Stone, P. (2008). Maximum likelihood estimation of sensor and action model functions on a mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*.

Sung, H. G. (2004). *Gaussian Mixture Regression and Classification*. Ph.D Thesis, Rice University, Huston Texas.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with functional approximation. *Advances in Neural Information Processing Systems*, 12:1057–1063.

Thrun, S., Montemerlo, M., Dahlkamp, H., Stavens, D., Aron, A., Diebel, J., Fong, P., Gale, J., Halpenny, M., Hoffmann, G., Lau, K., Oakley, C., Palatucci, M., Pratt, V., Stag, P., Strohband, S., Dupont, C., Jendrossek, L.-E., Koelen, C., Markey, C., Rummel, C., van Niekerk, J., Jensen, E., Alessandrini, P., Bradski, G., Davis, B., Ettinger, S., Kaehler, A., and Mahoney, A. N. P. (2006). Stanley: The robot that won the darpa grand challenge. *Journal of Field Robotics*, 23:661–692.

van Lent, M. and Laird, J. E. (2001). Learning procedural knowledge through observation. In *Proceedings of the 1st International Conference on Knowledge Capture*, pages 179–186.

Veeraraghavan, H. and Veloso, M. (2008). Teaching sequential tasks with repetition through demonstration. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*.

Watkins, C. J. H. C. (1989). *Learning from Delayed Rewards*. Ph.D Thesis, King's College, Cambridge, UK.

Watkins, C. J. H. C. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8:279–292.

Wheeler, D. S., Fagg, A. H., and Grupen, R. A. (1992). Learning prospective pick and place behavior. In *Proceedings of the International Conference on Development and Learning*.

Wiering, M., Salustowicz, R., and Schmidhuber, J. (1999). Reinforcement learning soccer teams with incomplete world models. *Journal of Autonomous Robots*.

Yeasin, M. and Chaudhuri, S. (2000). Toward automatic robot programming: Learning human skill from visual data. In *Proceedings of IEEE Transactions on Systems, Man, and Cybernetics*, volume 30, pages 180–185.