Chapter 13

Path Planning

Path planning allows autonomous mobile robots and manipulators to find a path to move between two points. A *path* is a set of poses from a start configuration to an end configuration that respect a set of specifications (for example, avoiding obstacles for a mobile base or respecting a specific force profile at the end effector of a manipulator). It differs from the concept of *trajectory* in that a trajectory is the execution of a path over time. Depending on the choice of the planning algorithm, a path could satisfy various degrees of optimality with respect to some criteria such as minimizing path length, minimizing turns, or minimizing the amount of braking. Algorithms to find a shortest path are important not only for robotics applications, but also in network routing, video games, and understanding protein folding.

Path planning requires a suitable representation of the environment such as a map introduced in Chapter 12, and a perceptual understanding of the robot's location with respect to such representation We will assume for now that the robot is able to localize itself, is equipped with a map, and is capable of avoiding temporary obstacles on its way. The goals of this chapter are to:

- introduce the concept of "configuration space" for planning,
- understand the difference between graph-based and sampling-based planning algorithms,
- explain basic path algorithms such as Dijkstra, A*, and RRT,
- understand variations of the path planning problem such as coverage path planning.

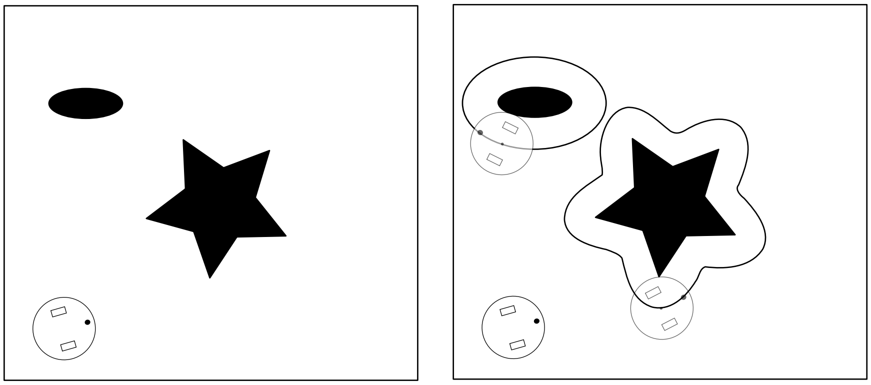

Figure 13.1. A map with obstacles and its representation in configuration space, which can be obtained by growing each obstacle by the robot's extension.

13.1. The configuration space

In the vast majority of path planning algorithms, the robot is treated as a point-mass element with no volume. In order for a path to be executed on the robot, it is important to take into account the physical embodiment of the robot and its non-zero volumetric occupancy, which complicates the path planning process. It is possible for the robot to be reduced to a point-mass while growing all obstacles by its radius. This works for a circular robot. This can be generalized for robots of any shape by growing each obstacle by the length of the longest extension of the robot from its center. This representation is known as *configuration space* as it reduces the representation of the robot to its controllable degrees of freedom (e.g., its x and y coordinates in the plane for a robot capable of planar translation). An example is shown in Figure 13.1. The configuration space can now either be used as a basis for a grid map or a continuous representation.

13.2. Graph-based planning algorithms

The problem to find a "shortest" path from one vertex to another through a connected graph is of interest for multiple domains, most prominently network routing, where it is used to find an optimal route for an internet data packet. The term "shortest" here is defined as the minimum cumulative edge cost, which could be physical distance (in a robotic application), delay

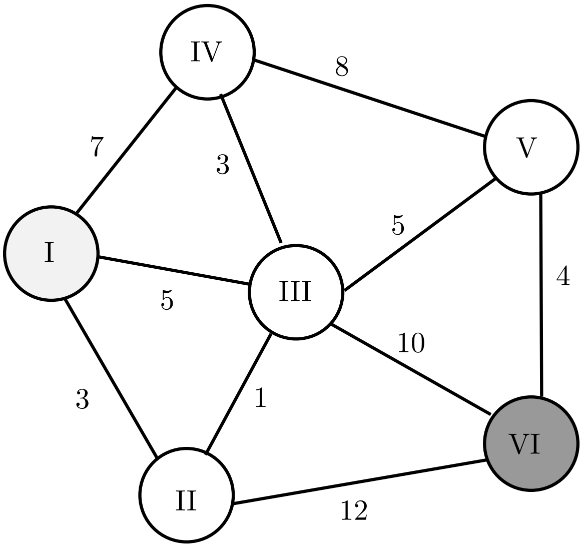

Figure 13.2. A generic path planning problem from vertex I to vertex VI. The shortest path is I-II-III-V-VI, and has length of 13.

(in a networking application), or any other metric that is relevant for the task. An example graph with arbitrary edge lengths is shown in Figure 13.2.

13.2.1. Dijkstra's algorithm

One of the earliest and simplest algorithms for path planning is Dijkstra's algorithm (Dijkstra 1959). Given a graph, Dijkstra is an iterative process where, starting from the "start" vertex, the algorithm marks all its direct neighbors with the cost to reach them. It then proceeds to inspect the neighboring vertex with the lowest cost and all its adjacent vertices and marks them with the cost to get to them via the vertex under consideration. If the cost turns out to be lower, the cost is updated accordingly. Once all neighbors of a vertex have been checked, the algorithm proceeds to the vertex with the next lowest cost. Once the algorithm reaches the goal vertex and there exist no vertex with a lower cost to the goal, it terminates and the robot can follow the edges pointing towards the lowest edge cost.

In the example in Figure 13.2, Dijkstra would first mark nodes II, III and IV with cost 3, 5 and 7 respectively. It would then continue to explore all edges of node II, which so far has the lowest cost. This would lead to the discovery that node III can actually be reached in $3 + 1 < 5$ steps, and node III would therefore be relabeled with cost 4. In order to completely evaluate node II, Dijkstra needs to evaluate the remaining edges before moving on and label node VI with $3 + 12 = 15$. The node with the lowest cost is now node III (cost of 4). We can now relabel node VI with 14, which is smaller than 15, and label node V with $4 + 5 = 9$, whereas node IV remains at $4 + 3 = 7$. Although we have already found two paths to the goal, one of which better than the other, we cannot stop as there still exist nodes with unexplored edges and overall cost lower than 14. Indeed, continuing to explore from node V leads to a shortest path I-II-III-V-VI of cost 13, with no remaining nodes to explore.

As Dijkstra would not stop until there is no node with lower cost than the current cost to the goal, we can be sure that a shortest path will be found if it exists. We can therefore say that Dijkstra is both *complete* and optimal.

As Dijkstra will always explore nodes with the least overall cost first, exploration of the environment resembles a wave front originating from the start vertex, eventually arriving at the goal. This is of course highly inefficient, in particular if Dijkstra is exploring nodes away from the goal. As an example, if we were to add a couple of nodes to the left of node I in Figure 13.2, Dijkstra would explore all of these nodes until their cost exceeds the lowest found for the goal. This can also be seen when observing Dijkstra's algorithm on a grid, as shown in Figure 13.3.

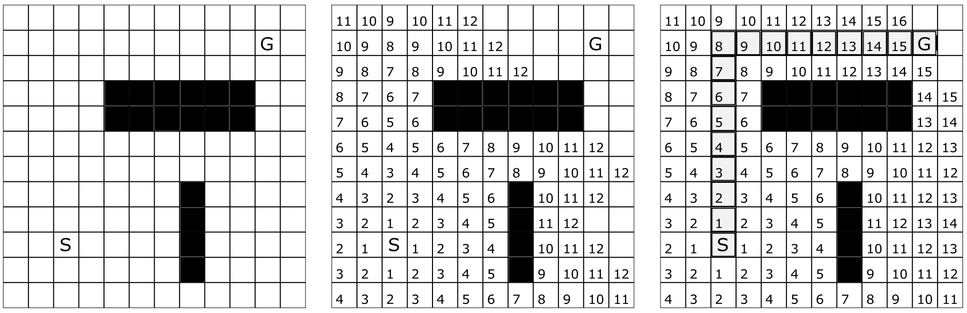

Figure 13.3. Dijkstra's algorithm finding a shortest path from 'S' to 'G' assuming the robot can only travel laterally (not diagonally) with cost one per grid cell. Note the few number of cells that remain unexplored once the shortest path (grey) is found, as Dijkstra would always consider a cell with the lowest path cost first.

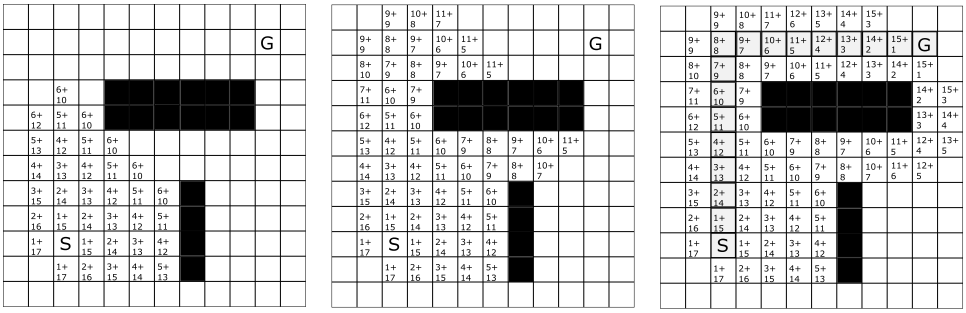

Figure 13.4. Finding a shortest path from 'S' to 'G' assuming the robot can only travel laterally (not diagonally) with cost one per grid cell using the A* algorithm. Much like Dijkstra, A* evaluates only the cell with the lowest cost, but takes an estimate of the remaining distance into account.

Note that the grid can be reduced to a graph in which each vertex, except those at the borders, have four or eight neighbors.

13.2.2. A*

Instead of exploring in all directions, knowledge of an approximate direction of exploration to reach the goal may help avoiding the exploration of nodes that are not needed to succeed in the task. As humans, we can easily interpret the task in Figure 13.3 and understand that most states in the top-left and bottom-right corner should not be explored if we want to find a solution in a short amount of time. Such knowledge may be encoded in the search algorithm via a *heuristic function*, i.e. an informed guess or estimate of sorts. For example, we could give priority to nodes that have a lower estimated distance to the goal than others. For this, we would mark every node not only with the actual distance that it took us to get there (as in Dijkstra's algorithm), but also with the estimated cost to target, for example by calculating the Euclidean distance or the *Manhattan distance* between the vertex we are looking at and the goal. This algorithm is known as A* (Hart, Nilsson & Raphael 1968), and illustrated in Figure 13.4 using the Manhattan distance metric. Depending on the environment, A* might accomplish search much faster than Dijkstra's algorithm, and performs the same in the worst case.

An extension of A* that addresses the problem of expensive re-planning when obstacles appear in the path of the robot is known as D* (Stentz 1994).

Unlike A*, D* starts from the goal vertex and has the ability to change the costs of parts of the path that include an obstacle. This allows D* to re-plan around an obstacle while maintaining most of the already calculated path.

A* and D* become computationally expensive when either the search space is large, e.g., due to a fine resolution required for the task, or when the dimensions of the search problem are high (e.g. when planning for an arm with multiple degrees of freedom). Solutions to these problems can be provided by sampling-based path planning algorithms.

13.3. Sampling-based path planning

Section 13.2 introduced a series of *complete* algorithms for the path planning problem, i.e. algorithms that are guaranteed to (eventually) find a solution if one exists. However, complete algorithms are often infeasible in practice, e.g. because of a large state space, low available memory, or limited time to execute the algorithm. This is often the case for robots with many degrees of freedom such as arms. Importantly, most algorithms are only *resolution complete*, i.e. they are only complete if the choice of environment resolution is fine enough: since the state space needs to be discretized, some solutions might be missed because of such discretization.

Sampling-based planners are an alternative to graph-based planners that evaluate all possible solution and non-complete Jacobian-based inverse kinematic solutions. In sampling-based motion planning, possible paths are generated via random sampling and stored in a tree-like structure until some solution is found or the alloted time expires. As the probability to find a path approaches one when the number of samples goes to infinity, sampling-based path planners are *probabilistic complete*. Prominent examples of sampling-based planners are *Rapidly-exploring Random Trees* (RRT)(LaValle 1998) and *Probabilistic Roadmaps* (PRM) (Kavraki, Svestka, Latombe & Overmars 1996).

An example execution of RRT is shown in Figure 13.5; in essence, RRT grows a single tree from a robot's starting point until one of its branches hits a goal. This example illustrates how a sampling-based planner can quickly explore a large portion of space and refine a solution over time. Conversely, probabilistic road-maps create a tree by randomly sampling points in the state space, testing whether they are collision-free, connecting them with neighboring points using paths that are achievable subject to the kinematics of the robot, and then using classical graph shortest path algorithms to

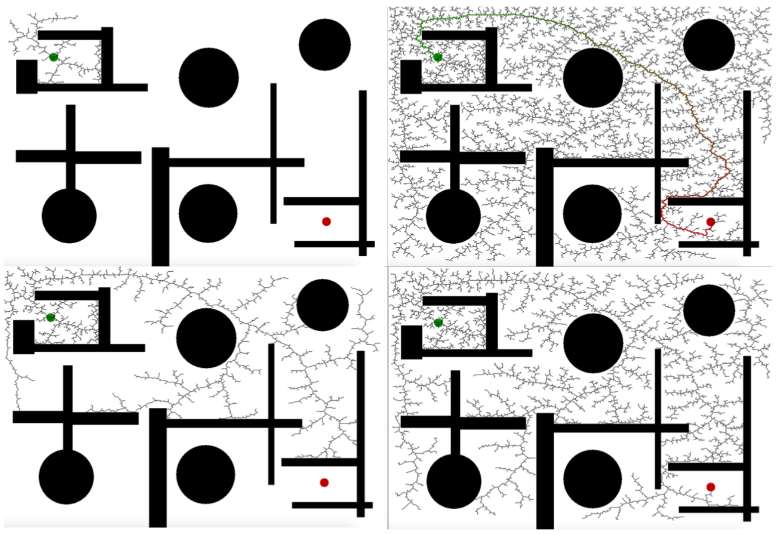

Figure 13.5. Counterclockwise from top-left: Random exploration of a 2D search space by randomly sampling points and connecting them to the graph until a feasible path between start and goal is found.

find shortest paths on the resulting structure. The advantage of PRM is that the map has to be created only once (assuming the environment is not changing) and can then be used for multiple queries. PRM is therefore a *multi-query* path planning algorithm, whereas RRT is known as *single-query* path planning algorithm. Over the years the boundary between these different algorithms has blurred, and single-query and multi-query variants of both RRT and PRM exist. In all, there is no 'silver bullet' algorithm or heuristic and even the choice of their parameters is highly problem-specific. We will therefore limit our discussion on useful heuristics to those that are common to sampling-based planners.

13.3.1. Rapidly Exploring Random Trees

Let $\mathcal{X}$ be a d-dimensional state space. This can either be the robot's state given in terms of translation and rotations (6 dimensions or a subset thereof), or the joint space with one dimension per joint. What representation you chose will determine how to compute whether a point is reachable or not, but will not affect the algorithm itself.

Let $\mathcal{G} \subset \mathcal{X}$ be a $d-$dimensional sphere in the state space that is considered to be the goal, `max_dist` the longest permissible edge length, `t` the allowed time, `k` the maximum number of vertices to allow in the tree, and `goal_bias` the percentage of the time the algorithm should try to connect to a goal state. An RRT planner would follow the below pseudo-code:

```
Tree=Init(X, G, start, max_dist, t, k, goal_bias);
iteration = 0
WHILE (ElapsedTime() < t AND iteration < k
AND NoGoalFound(Tree,G)) DO:
 iteration = iteration + 1
 IF RandomPercentage() < goal_bias THEN
  q_rand = SampleRandomGoal(G);
 ELSE
  q_rand = SampleRandomState(X);
 ENDIF
 q_nearest = NearestVertex(q_rand)
 q_new = Extend(q_nearest, q_rand, max_dist)
 edge = CreatePath(q_nearest, q_new);
 IF IsAllowablePath(edge) THEN
  Tree.addVertex(q_new);
  Tree.addEdge(edge);
 ENDIF
ENDWHILE
return Tree
```

This process can be iterated as long as time allows and maximum number of vertices or `goal_bias`) are optional parameters. RRT is known as an *Anytime algorithm*, i.e., any user interruption once an initial solution has been found would still provide some kind of solution. Given a suitable distance metric, the path cost can be stored at each node of the tree, allowing to track the shortest path to goal in case there are multiple vertices in the goal region. There are four key points in this algorithm, which will be discussed below:

1. determining the next point `q_rand` to add to the tree (`SampleRandomGoal`, `SampleRandomState`, and `Extend`);
2. finding out where and how to connect this point to the tree taking into account the robot kinematics (`NearestVertex`, `CreatePath`);

3. testing whether this path is suitable (`IsAllowablePath`)—i.e., collision-free;
4. smoothing the path (not shown in the algorithm).

Selecting the next best point. A simple approach is to randomly select a point in the state space and connect it to the closest existing point in the tree. Other solutions may assign preference to nodes with few out-degrees (i.e. those without many connections), and choose points in their vicinity in order to facilitate expansion in under-explored regions of the state space. Importantly, both approaches allow to quickly explore the entire state space; if there are constraints imposed on the robot's path—e.g., if the robot needs to hold a cup and therefore is not supposed to rotate its wrist—this dimension can simply be taken out of the state space and fixed at runtime.

Connecting points to the tree. Intuitively, the new point `q_rand` should be connected to the closest point already in the tree or to the goal. This requires iterating over all nodes in the tree and calculating their distance to the candidate point `q_rand`, which is a computationally expensive process; the resulting point `q_nearest` is the one with the shortest distance. The selection of the right data structure for storing the graph in memory may reduce the computational cost to be on average sub-linear in the number of vertices.

Importantly, following this method does not guarantee that the shortest path will be found. As an alternative, RRT* grows the tree in a way that always minimizes the overall path length from the root to every vertex. This is done in two steps. First, only points in the tree within a $d-$dimensional sphere (on a 2D map, $d = 2$, i.e. a circle) of fixed radius from `q_rand` are considered, and the point that minimizes the overall path length from the start configuration (rather than simply the shortest distance from `q_rand`) is found. With this step, we can guarantee that the new vertex `q_rand` is connected to the shortest reachable path from the root of the tree. Second, a *rewiring* step occurs where vertices near `q_rand` are evaluated to inspect if an edge between them and `q_rand` would be shorter than the current edge. If this is true and the edge is allowable (i.e., not in collision nor outside of the physical abilities of the robot), the graph is rewired so that the newfound vertex becomes the new parent of `q_rand`.

Once the nearest vertex is found, the `Extend` function uses the `max_dist` parameter to limit the maximum edge length, replacing `q_rand` with a point `q_new` on the line connecting `q_nearest` and `q_rand` that is `max_dist` away from `q_nearest`. During this step, it is also a good time to take into account the specific kinematics of a robot and its motion capabilities. In the example of a car, a local planner can be used to generate a suitable trajectory that takes into account the orientation of the vehicle at each point in the tree. Using an open-source physics simulation such has been developed for computer games also allows to consider dynamics, including drift. Using such a simulation within a planning framework has demonstrated trajectories that meet the performance of the most skilled operators ((Keivan & Sibley 2013)).

Collision checking. Efficient algorithms for testing collisions deserve a dedicated section. While the problem is intuitive in configuration-space 2D planning and can be solved using a simple point-in-polygon test (since the robot reduces to a point), this issue becomes more involved for manipulators that are essentially multiple rigid bodies connected together and that may be subject to self-collisions. Conventionally, collision checking for these kind of objects has be achieved by converting them into triangle meshes that can then be tested for intersections. More recently, physics-based computer game engines that provide built-in collision checking are increasingly used. This makes particularly sense, when such engines are also used to predict the dynamics of rigid bodies within the `CreatePath` function.

Typically, collision checking takes up to 90% of the execution time of a path planning problem; therefore, methods that aim at reducing computational cost are desirable. For example, the "lazy collision evaluation" algorithm differs from standard collision checking in that it does not evaluate every point for a possible collision. Rather, it first finds a suitable path, and only after a path is found it evaluates every edge for collisions. Segments in collision are deleted and the algorithm continues, but only collision-free segments are maintained.

Once a possible path is found, the sampling space can be reduced to an ellipsoid that bounds the maximal path length. This ellipsoid can be constructed by mounting a wire of the maximum path length between start and goal and pushing it outward with a pen. Intuitively, only points that are contained by this ellipsoidal area can provide a shorter path than the one currently known, so it becomes a waste of time to grow the tree in areas of the state space that are outside of this ellipsoid. This approach is particularly effective when running multiple copies of the same planner in parallel and exchanging the shortest paths once they are found (Otte & Correll 2013).

Path smoothing. As path planning randomly samples from discrete and arbitrarily coarse maps, the resulting paths are typically jagged and irregular—i.e., far from optimal in practice. This can be drastically improved via path smoothing. One way of doing this is to connect points of the path using splines, polynomial curves, or even trajectory snippets that are known to be feasible for a specific platform. Alternatively, one can also use a model of the actual platform and use a feedback controller such as the one described in Section 3.4.2 for mobile robots and Section 3.2.2 for manipulators, which will generate a trajectory that the robot can actually drive. When combined with dynamics, this approach is known as *model-predictive control*. Again, using a physics-based simulation environment

13.4. Planning at different length scales

The reality of performing complex, autonomous behaviors in realistic scenarios is that, in practice, no one map representation and planning algorithm might be sufficient. Planning a route for a car, for example, is a multi-step process wherein robot autonomy interleaves with human intelligence: as detailed in Figure 13.6, a hierarchy of increasingly granular map representations and path planning algorithm is needed. First, a coarse search is performed over the street network (by e.g. your preferred mapping and navigation app), followed by a more precise planner that determines which lanes to choose and how to navigate roundabouts and intersections; in both these layers of abstraction, graph-based planning algorithms are ideal. Then, a sampling-based algorithm may be used to determine how to actually move the car between lanes and what trajectory to use to avoid obstacles. Finally, such trajectories need to be turned into wheel speeds and steering angles—possibly using some form of feedback control. In Figure 13.6, downward-pointing arrows indicate the input that one planning layer provides to the one below, whereas upward-pointing arrows instead indicate exceptions that cannot be handled at the lower levels. For example, a feedback controller cannot handle obstacles, requiring the sampling-based planning layer to come up with a new trajectory. Should the entire road be blocked, this planner would need to hand-off control the lane-based planner. A similar case can be made for manipulating robots, which also need to combine multiple different representations and controllers to plan and execute trajectories efficiently.

Note that this representation does not include a reasoning level that encodes traffic rules and common sense. While some of these might be encoded

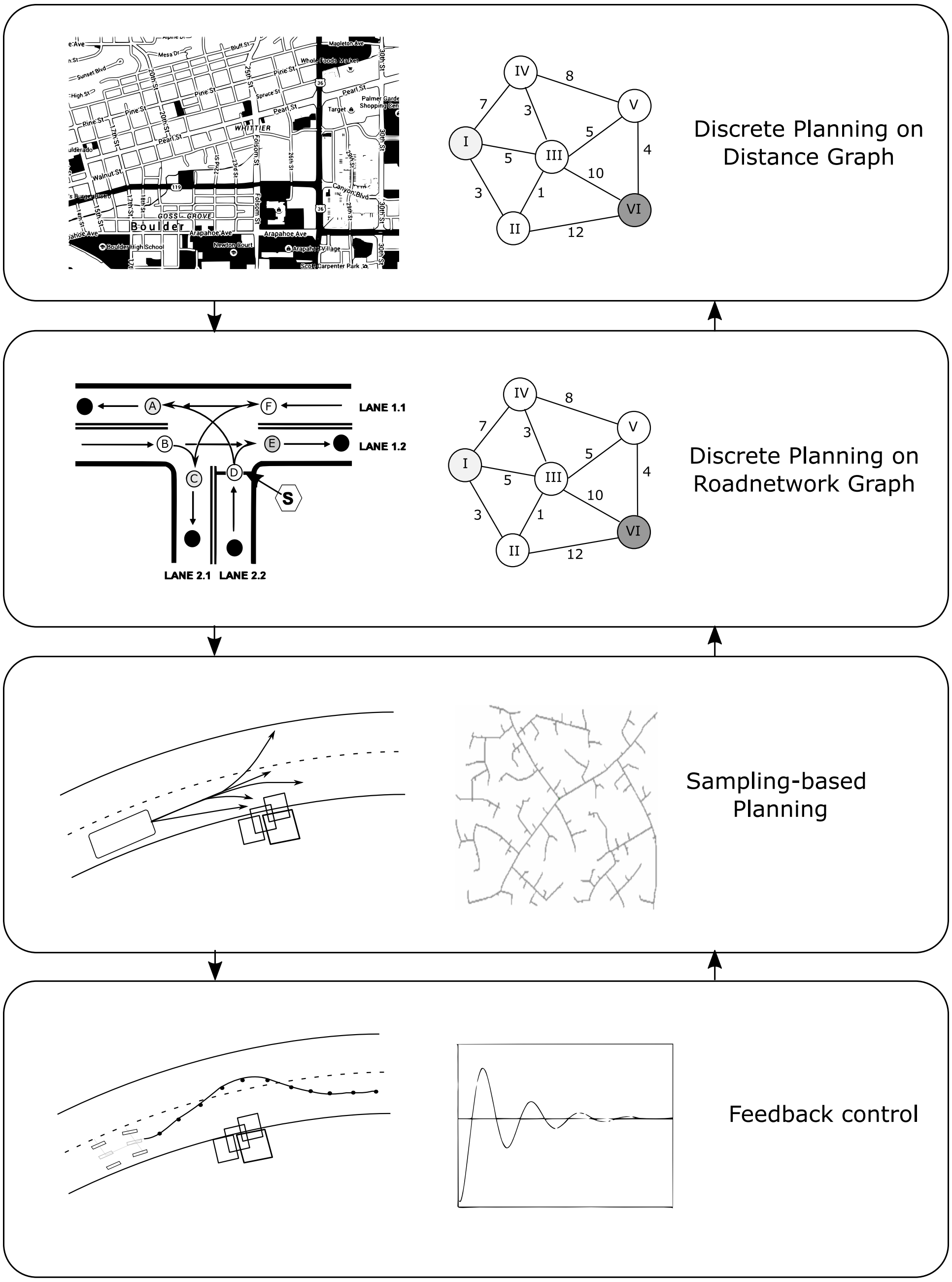

Figure 13.6. Path planning across different length scales, requiring a variety of map representations and planning paradigms. Arrows indicate information passed between layers.

using cost-functions, such as maximizing distance from obstacles or insuring smooth riding, other more complex behaviors such as adapting driving in the presence of cyclists or properties of the ground need to be implemented in an additional vertical layer that has access to all planning layers.

13.5. Coverage path planning

So far, we have only considered the problem of finding a (shortest) path. A variation of the path planning problem is *coverage path planning*. This is relevant for applications such as cleaning, mowing or painting and usually aims at minimizing the time to completion and redundancy during coverage. This problem is closely related to the shortest path problem. For example, floor coverage can be achieved by performing a depth-first search (DFS) or a breadth-first-search (BFS) on a graph where each vertex has the size of the coverage tool of the robot. "Coverage" is not only interesting for cleaning a floor: the same algorithms can be used to perform an exhaustive search of a configuration space, such as in the example shown in Figure 3.3, where we plotted the error of a manipulator arm in reaching a desired position over its configuration space. Finding a minimum in this plot using an exhaustive search solves the inverse kinematics problem.

Doing a DFS or a BFS might generate efficient coverage paths, but they are far from optimal as many vertices might be visited twice. A path that connects all vertices in a graph but passes every vertex only once is known as a *Hamiltonian Path*. A Hamiltonian path that returns to its starting vertex is known as a Hamiltonian Cycle. This problem is also known as the Traveling Salesman Problem (TSP), in which a route needs to be calculated that visits every city on his tour only once and is known to be NP Complete.

13.6. Summary and Outlook

Path planning is an ongoing research problem. Finding collision-free paths for mechanisms with high degrees of freedom (such as multiple arms operating in a shared space, multi-robot systems, or systems involving dynamics) is still a computationally intensive problem. Although sampling-based path planners can drastically speed up the time to find some solution, they are not optimal and struggle with algorithm-specific concerns such as navigating in narrow passages. There is no "silver bullet" algorithm for solving all path planning problems and heuristics that lead to massive speed-up in one scenario might be detrimental in others. Also, algorithmic parameters are mostly ad-hoc and correctly tuning them to a specific environment may drastically increase performance.

Take-home lessons

- The first step in path planning is choosing a map representation that is appropriate to the application (Chapter 12).
- The second step is to reduce the robot to a point-mass, which allows planning in the configuration space (or C-space).
- This allows the application of general-purpose shortest path graph-based algorithms, which have applications in a large variety of domains and that are not limited to robotics.
- A sampling-based planning algorithm finds paths by sampling random points in the environment. Heuristics are used to maximize the exploration of space and bias the direction of search. This makes these algorithms fast, but neither optimal nor complete.
- As the resulting paths are random, multiple trials might lead to entirely different results.
- There is no one-size-fits-all algorithm for a path planning algorithm and care must be taken to select the right paradigm (e.g. single-query vs. multi-query), heuristics, and parameters.

Exercises

1. How does the computational complexity of Dijkstra's algorithm change when moving from 2D to 3D search spaces?
2. A* uses a "heuristic" to bias the search in the expected direction of the goal. Why can it only use a heuristic, not the actual length?
3. Assuming points are sampled uniformly at random in a randomized planning algorithm. Calculate the limiting behaviour of the following ratio (area of points in tree)/(area of points sampled) as the number of points sampled goes to infinity, assuming no duplicates. Assume the total area A_{total} and the area of free space A_{free} within are known.