AA 274: Principles of Robotic Autonomy Winter 2019

Lecture 15: Combinatorial Motion Planning

15.1 Introduction

A robot must show competence in reaching its goal position as efficiently as possible based on its knowledge and sensor values given a series of goal positions and a map of its environment. This task gives rise to the navigation or motion planning problem.

Robot motion planning: compute a sequence of actions that drives a robot from an initial condition to a terminal condition while avoiding obstacles, satisfying kinematic and dynamic constraints, and possibly optimizing an objective function.

Some examples of common motion planning problems may include steering autonomous vehicles, controlling humanoid robots, protein docking, and surgery planning.

15.1.1 History

The problem of motion planning was formally defined in the 1970s as mobile robots became more prevalent in industries. However, exact combinatorial solutions in discrete configuration spaces were not developed until the 1980s with the introduction of Breadth-First Search and Depth-First Search algorithms. Grid-based search algorithms such as A* and D* which was developed by Anthony Stentz followed in the mid-1990s [**bertsekas**]. These methods worked well for simple motion planning problems but were subject to the curse of dimensionality for planning in high-dimensional state spaces. To address this challenge, sampling-based methods, notably Probability Road-Maps (PRM) and Rapidly-Exploring Random Trees (RRT), were developed in the late 1990s and deployed on real-time systems in the 2000s [**lavalle**]. With the increasing complexity of robotic systems, solving motion planning problems in real-time has become a significant challenge. Hence, current research has focused on speeding up sampling-based methods through lazy dynamic programming and massive parallelization of search algorithms.

15.2 Configuration-Space

In the context of robotic motion planning, we will first create a simple problem setup to define the variables of interest and introduce the proper nomenclature. First, consider a workspace defined in 2D, $W \subseteq R^2$, which effectively describes the world that is occupied by the robot, the obstacles, and potentially other agents in the system. Within this workspace, we define the robot as a rigid polygon, and we introduce the set of obstacles $O \subset W$ as a subset of the workspace shown in Fig 15.1.

The rigid robot can be translated or rotated according to its own kinematic constraints. With these definitions established, we can qualitatively define the robotic motion planning problem as:

Problem Statement: Compute the sequence of movements to go from a given initial placement of the robot to a desired goal placement such that the robot never touches the obstacle regions.

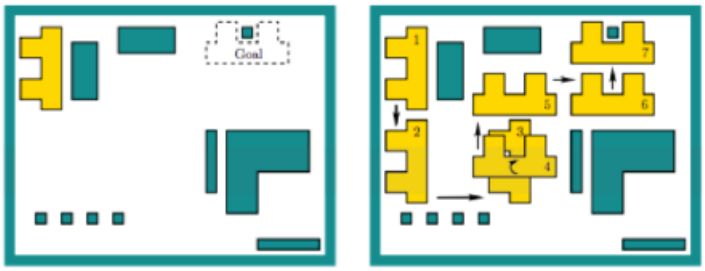

Figure 15.1: 2D Workspace

While the robot motion planning problem is described in a physically intuitive "real-world" setting (e.g., the 2D planar workspace described above), it is vital to map the problem into another space, denoted as the configuration (C-) space. The configuration space is simply the space containing the degrees of freedom of the robot. Returning to our simple example highlighted above, the configuration space contains the translational degrees of freedom and the rotational degrees of freedom of the robot. In this way, the 2D path planning problem in physical space is cast as a 3D path planning problem in the configuration space, with two dimensions describe the translational modes of the trajectory and one degree of freedom capturing the rotational mode of the trajectory in the planar workspace.

Formally, the degrees of freedom for our rigid polygon robot, R R2, is the set of generalized coordinates, $q = (xt, yt, q)$, where (xt, yt) R2 are the translational coordinates, and q is the rotational coordinate. Accordingly, the configuration space consists of every combination of q which yields a unique robot placement (in this case a subset of R3). An important distinction must be made for the generalized coordinate describing the rotational degree of freedom; namely, q and q2pk, k2 Z describe equivalent attitudes. The configuration space in this case is actually in R2 S1, where S1 is the manifold which contains the angular displacement variable and includes the aforementioned "equivalence" between angles that differ by integer multiples of 2p. If we relax the planar restriction in our working example, the angular orientation coordinates of the robot would instead be in the manifold S3. The importance of this distinction is highlighted with a simple example: consider a robot that has a current heading of q = 3p/2 rads, and we want to reconfigure the robot to heading qg = 0 subject to the constraint of avoiding a C-space obstacle at q = p. The geometry of this problem is highlighted in Figure 15.2. Now, if the equivalence between the angles 0 and 2p is not established in the definition of the configuration space, the robot would not be able to traverse a collision-free path to the desired heading in the configuration space (see red trajectory). Instead, since the configuration space is defined with respect to S1, the robot is able to achieve the desired heading (see green trajectory).

Note that for our simple planar workspace example, the dimensions of the physical space and the configuration space are quite similar. Instead, it is not uncommon for complicated dynamical systems (e.g., a robotic arm mounted on a spacecraft) to have configuration spaces of much higher dimension than the physical space counterpart. In these cases, mapping from physical space to configuration space is extremely vital to the motion planning problem as it allows all constraints of the complex kinematic system to be accounted for with higher-dimensional information (i.e., more coordinates).

15.2.1 Planning in C-Space

Now that we have introduced the necessary nomenclature and addressed the important distinction between motion planning in the real (physical) space and the configuration space, we can return to our prior

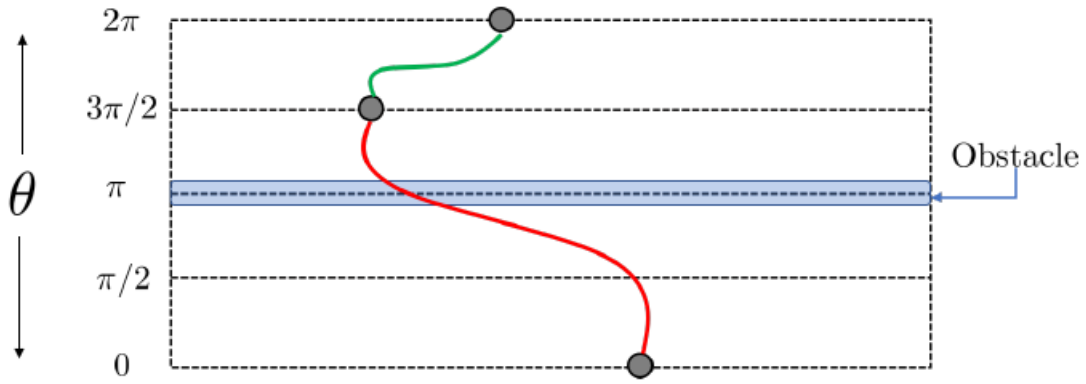

Figure 15.2: Example trajectory planning where the rotational degree of freedom is not described on S1 (red) and where it is described on S1 (green)

problem statement and re-write it in rigorous mathematical detail. Let the set of points in the workspace that are occupied by the robot in configuration q be denoted by R(q) W. Furthermore, we can make the intuitive definition of collision-free motion as that set of all robot configurations which do not intersect the obstacle regions: $R(q) \cap O = \emptyset$. Then the formal problem statement can be re-written as:

Problem Statement, Revisited: Compute the continuous path: t : [0, 1]) fq 2 CjR(q) O = Æg with t(0) = qI and t(1) = qG In general, we refer to the set fq 2 CjR(q) O = Æg as Cf ree, which represents all possible combinations of the robot's degrees of freedom which are collision-free. A key fact here is that, while a robot occupies physical dimensions in the real space, the robotic motion is described by a simple point in the configuration space that captures the translation degrees of freedom (of the center of mass for example) and the rotational degrees of freedom of the robot. In this way, mapping to the configuration space poses the motion planning problem as a simpler problem of finding a path for a point. An illustration of this problem is highlighted in Figure 15.3. The physical dimensionality of the robot is intrinsically captured in the definition of the configuration space (in particular, in Cf ree). This property represents a key advantage of moving to the configuration space for robotic motion planning, as it allows the same general planning problem formulation (i.e., path planning for a point) to be equally applicable to myriad dynamical systems.

We now have a formal mathematical definition for the robot motion planning problem. In order to simplify the procedure, we recast the problem from the real/physical space to the configuration space. The result is that the motion planning problem now becomes a path planning problem for a single point. We have not yet, however, set up an architecture for obtaining the point path solution. In the following sections, several solution methods will be highlighted which leverage the aforementioned advantages of working in the configuration space for motion planning.

15.3 Combinatorial Planning

The goal of combinatorial planning is to find paths through the configuration space without needing approximations as shown in Figure 15.4. For low dimensional convex problems, this type of planning either provides *exact* solutions or none if no feasible path exists. This can allow for better performance in some cases , compared to approximation-based methods.

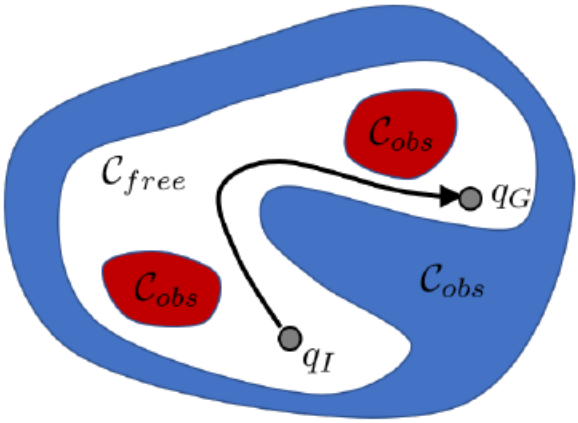

Figure 15.3: Robotic motion planning as a point path planning problem in C-space.

Figure 15.4: Combinatorial Planning

Combinatorial Planning refers to the use of all combinations of possible states in defining the robot configuration q. For example, consider the configurations used above in setting up the configuration-space: $q = (xt, yt, q)$. In this case, the x and y position and the heading q define a unique configuration.

Combinatorial motion planning approaches construct a roadmap, a graph G with conditions: 1. Accessiblity: From any configuration it must always be possible to connect to the graph. For instance, q1 can be connected to s1 on the roadmap.

2. Connectivity: If there is a path between two configurations q1 and q2, then there exists a path on the roadmap between s1 and s2. The roadmap essentially provides a discrete representation of the continuous problem. Any original connectivity information is maintained across the mapping. The typical approach involves cell decomposition. There are requirements that the cells be easy to traverse, the decomposition be easy to compute, and the cell adjacencies be straightforward to determine.

15.4 Cell Decomposition

In order to pick the roadmap, a discrete representation of the configuration space, the environment needs to be decomposed in a way where connectivity of the environment is captured. Cell decomposition refers to algorithms that partitions Cf ree into a finite set of regions called cells. Three properties should be satisfied by cell decomposition, to reduce motion planning problem to a graph search problem:

1. Each cell should be easy to traverse. For instance, for ideal convex cell, any points inside it must be connected by a line segment. 2. Decomposition should be easy to compute. 3. Adjacencies between cells should be straightforward to determine, in order to build the roadmap.

15.4.1 2D Decomposition

We first take a look at 2D vertical cell decomposition, which partitions Cf ree into a finite collection of 2-cells and 1-cells, where k-cell refers to a k-dimensional cell. Each 2-cell is either a trapezoid or a triangle. Each 1-cell is a vertical segment that serves as the border of two 2-cells. Figure 15.5 shows an environment demonstrated as a configuration space, with free configuration space Cf ree represented as the white regions and obstacle space Cobs represented as the dark regions. In other words, those configurations for the robot would lead to collision. Though the planning problem is 2-dimensional, the robot configuration space is 3-dimensional, as robot heading affects whether it collides with obstacles as well.

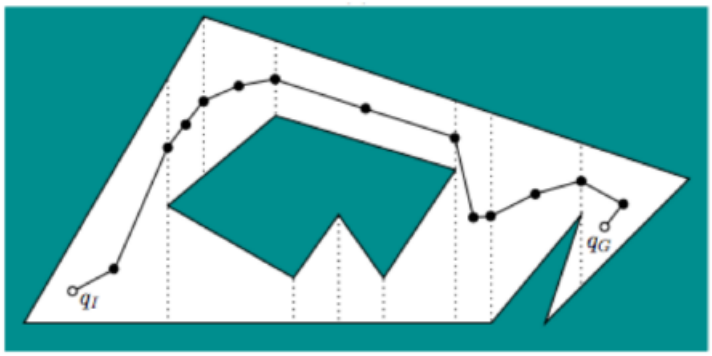

Figure 15.5: Example of 2D Cell Decomposition and Roadmap Extraction given qI and qG.

Assume the obstacle space is polygonal, then vertical cell decomposition works as follows. Let P denote the set of vertices used to define Cobs. For each vertex p 2 P, extend rays upwards and downwards through Cf ree, until Cobs is reached. Every intersection forms a cell, and thus the environment becomes a set of cells. As all cells have shapes of either triangles or trapezoids, they form convex sets. Hence, if we put a roadmap node randomly inside a cell, all other configurations within this cell can be connected to the roadmap point. In consequence, connectivity of the environment is preserved through this cell decomposition.

Now that we have different cells that compose the environment. One possible way is to place roadmap sample point at the centroid of each cell and at midpoint of each boundary. Once the roadmap is constructed, given any initial and goal configurations qI and qG, a path can always be extracted from the roadmap. To form a path from a roadmap, searching algorithms are needed, which will be introduced in section 15.5.

Other ways of constructing the roadmap are based on different requirements. Figure 15.6 shows two requirements: maximum clearance and minimum distance. Specifically, if maximum clearance from Cobs is required, especially in obstacle avoidance problem, we would like to place roadmap points along the middle of the tunnel representing the boundary of Cobs, meaning equidistant from both walls. Voronoi diagram is used to compute the roadmap by connecting qI and qG along edges of the constructed diagram in polygons. In the other case where the shortest path is the goal, a visibility graph can be used. A visibility graph represents the set of unblocked lines between vertices of obstacles, qI and qG. Usually shortest paths tend to graze the corners of Cobs as much as possible.

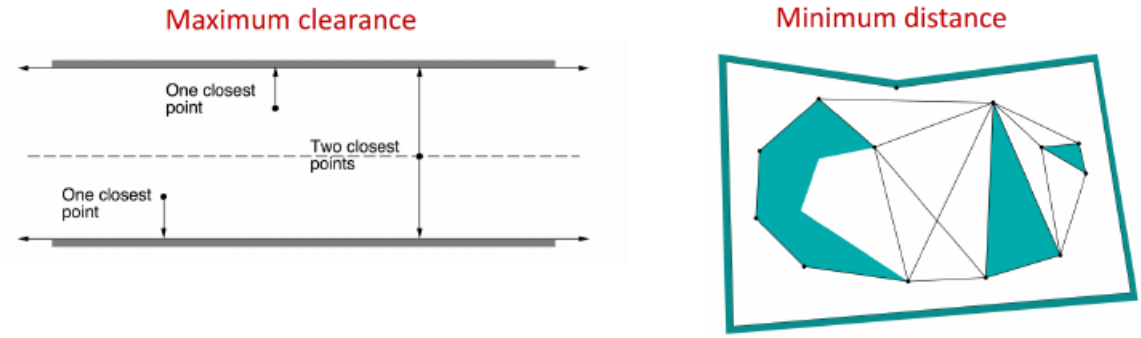

Figure 15.6: Other Cell Decomposition Methods Based on Requirements.

/subsection3D Vertical Decomposition Computing cell decomposition is much harder for higher-dimensional cases, though vertical cell decomposition is straightforward in 2D. We consider the special case where Cobs is piecewise linear and polyhedral. It turns out that we can extend the 2D vertical decomposition method by applying the idea of plane-sweeping to higher dimensions. Plane-sweeping refers to sweeping a plane across the space, only to stop where critical change occurs in information. In 3D, assume a polyhedral robot can translate in R3, and the obstacles are polyhedral. Thus Cobs R3 is polyhedral as well. The 3D vertical decomposition algorithm is as follows, as demonstrated in Figure 15.7.

Let (x, y, z) denotes a point in R3. Vertical decomposition yields 3-cell, 2-cell and 1-cell. A generic 3-cell is bounded by 6 planes, and its cross section for a fixed x yields a trapezoid or triangle in a plane parallel to the yz plane. Two sides of a 3-cell are parallel to the yz plane, and two other sides parallel to xz plane. The 3-cell is bounded above and below by two polygonal faces of Cobs.

The general idea is to sweep a plane perpendicular to x axis, where each fixed value of x produces a 2D polygonal slice of Cobs. Three example slices are shown in the bottom of Figure 15.7. Each slice is parallel to yz plane and simplify the 3D problem to a problem that can be solved by 2D vertical decomposition method. The middle slice shows the condition where the sweeping plane just encounters a vertex of a convex polyhedron, represented as a dot. This corresponds to an x value in interest, as critical change must occur in the slices. Hence, the 3D cell decomposition can be developed incrementally by sweeping through planes to update 2D vertical cell decomposition, in order to incorporate critical changes. To construct a roadmap, sample points are placed at center of each 3-cell and 2-cell. Edges are added to the roadmap by connecting each 3-cell to an adjacent 2-cell.

15.5 Graph-Search Algorithms

Suppose that our environment map has been converted into a connectivity graph using one of the graph generation methods presented earlier. Whatever map representation is chosen, the goal of path planning is to find the best path in the map's connectivity graph between the start and the goal, where best refers to the selected optimization criteria. In this section, we present several search algorithms that have become quite popular in mobile robotics.

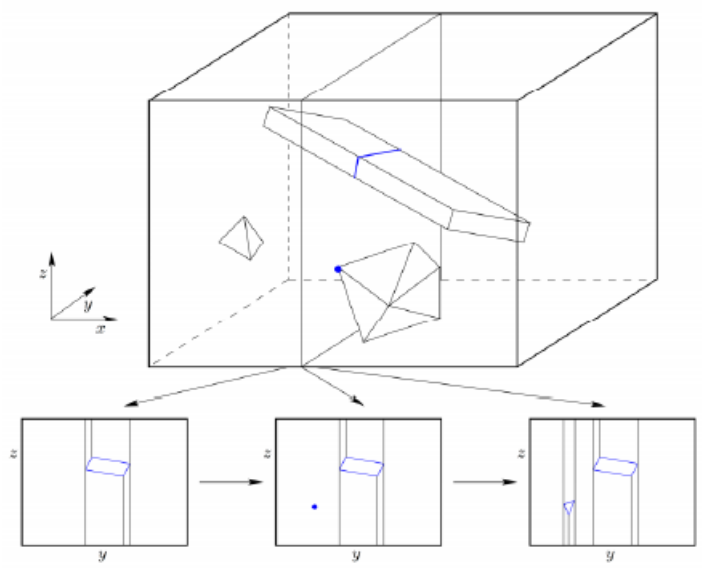

Figure 15.7: 3D Cell Decomposition Extended from 2D.

15.5.1 Concepts in graph search

We introduce the concepts which are all function of the node n (and an adjacent node n0). g(n) : Path cost(accumulated cost from the start node to any given node n) c(n, n0) : Edge traversal cost(cost from a node n to an adjacent node n0) h(n) : Heuristic cost(expected cost from a node n to the goal node) f (n) : Expected total cost(from start to goal via state n) f (n) = g(n) + eh(n), where e is a parameter that assumes algorithm-dependent values.

15.5.2 Discriminator graph search

Heuristic function unemployed – Uniform traversal cost simpler form, obtain faster execution speeds ex) depth-first, breadth-first – Nonuniform traversal cost higher algorithmic complexity ex) Dijkstra's algorithm - Heuristic function employed incorporates additional information about the problem set and thus often allows for faster convergence of the search query. ex) e = 1 : optimal A* algorithm, e ¿ 1 : suboptimal or greedy A* variant

15.5.3 Various graph search algorithms

Depth-First Search Depth-first search expands each node up to the deepest level of the graph, until the node has no more successors. As those nodes are expanded, their branch is removed from the graph and the search backtracks by expanding the next neighboring node of the start node until its deepest level and so on. Depth-First Search stores only a single path from the start nodes for each node on the path, leading to a lower memory requirement. However, previously visited nodes might be visited and the algorithm might enter redundant paths.

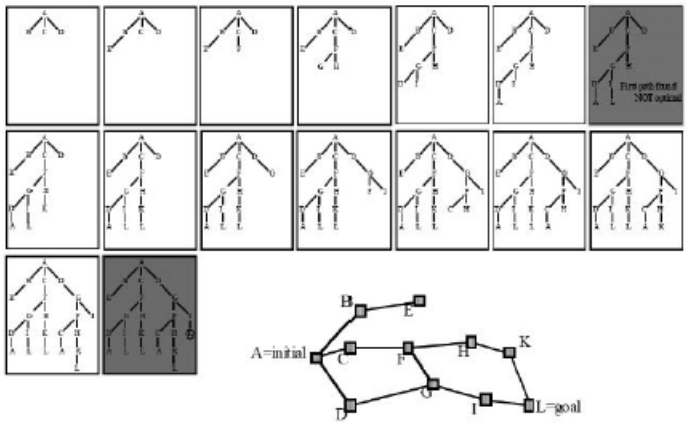

Figure 15.8: Depth First Search

Breadth-First Search Breadth-first begins with the start node and explores all of its neighboring nodes. Then, for each of these nodes, it explores all their unexplored neighbors and so on. The search always returns the path with the fewest number of edges between the start and goal node.

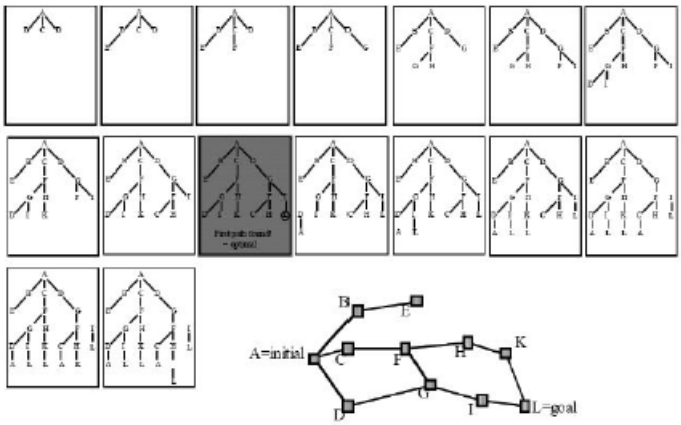

Figure 15.9: Breadth First Search

Dijkstra's algorithm (Best-First Search) Dijkstra's algorithm is similar to Breadth-First search, except that

edge costs may assume any positive value and the search still guarantees solution optimality. Dijkstra's algorithm uses the concept of the heap, a specialized tree-based data structure and selects the next q as: q = argminq2QC(q))

15.5.4 Label Correcting Algorithm

The label-correction algorithm is a general type of shortest path algorithm which includes very common graph search algorithms like breadth-first search, depth-first search, A*, Dijkstra's algorithm as special cases. The idea is to progressively discover shorter paths from the origin to every other node q, and to maintain the lowest cost path(from qI to q) found so far in a variable C(q).

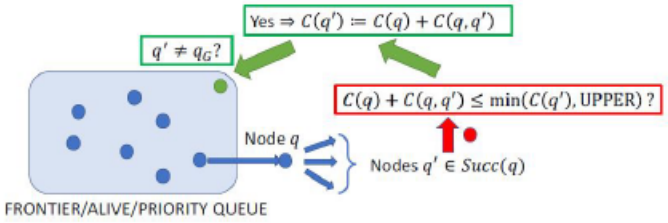

Figure 15.10: Label Correcting

Algorithm Step Step1. Remove a node q from the frontier queue and for each child q0 of q, execute step 2 Step2. If C(q) + C(q, q0) min(C(q0),UPPER), set C(q0) := C(q) + C(q, q0) and set q to be the parent of q0. In addition, if q0 6= qG, place q0 in frontier queue if it is not already there, while if q0 = qG, set UPPER to the new value C(q) + C(q, qG) Step3. If the frontier queue is empty, terminate, else go to step 1. Initialization : set the labels of all nodes to, except for the label of the origin node, which is set to 0.

15.6 Correctness and Improvements

Label correcting algorithms are guaranteed to find the shortest feasible path from an initial state to a goal state, given one exists. This attribute gives rise to the Correctness Theorem. Theorem If a feasible path exists from q1 to qG, then the algorithm terminates in finite time with C(qG) equal to the optimal cost of traversal, C(qG). However, computing this path can be computationally intensive especially in high-dimensional problems. This challenge arises from an explosion in the number of states expanded during the search procedure. Consider Fig 15.11, Depth-First Search and Breadth-First Search algorithms do not greedily select states from the priority queue for expansion. Hence, these algorithms search regions of the configuration space relatively uniformly without consideration of the direction of the expanded state to the goal. Dijkstra's algorithm improves on Depth-First Search and Breadth-First Search by greedily selecting states from the priority queue which prevents revisiting previously visited states. However, Dijkstra's algorithm searches for paths uniformly around the initial state, leading to wasted effort. Concentrating the search phase in regions leading to the goal state can reduce the number of states expanded during the search, speeding up the motion planning process.

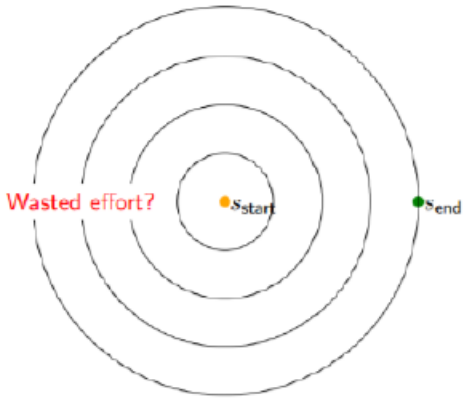

Figure 15.11: Graph Search Algorithm

15.6.1 A*: Improving Dijkstra

The A* algorithm improves on Dijkstra by including a heuristic function to the optimal "cost-to-arrival" when checking the children of the open node for addition to the priority queue. The heuristic function models the "cost-to-go" to the goal. This function guides exploration towards the region of the configuration space close to the goal, leading to faster computations as less states are expanded. Thus, the test condition changes from

$$C(q) + C(q, q0) \leq UPPER$$

to

$$C(q) + C(q, q0) + h(q) \leq UPPER$$

where h(q) is the optimal cost-to-go. The heuristic must be a positive underestimate of the true cost to the goal. The modification of the test condition reduces the number of nodes placed in the priority queue and still guarantees that an optimal path will be returned. Figure 15.12 illustrates A* implemented on a grid. States in black represent

obstacles while the light gray states represent nodes on the priority queue. States in dark gray represent opened nodes. The state with the minimum total estimated cost (including the heuristics cost) is selected from the priority queue for expansion.

Contributors

Winter 2019: Warren Cheng, Junwu Zhang
Winter 2018: Ryan Richey, Hyoungju Seo, Ola Shorinwa, Josh Sullivan, Danning Sun

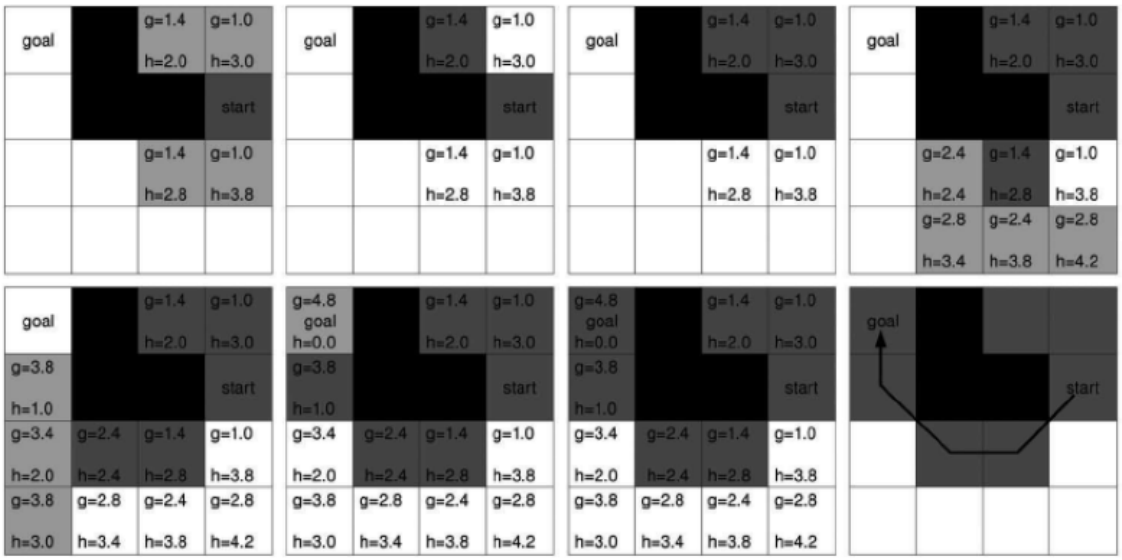

Figure 15.12: A* Algorithm