LECTURE #7

INSTRUCTOR: A. MAJUMDAR

Last Lecture: Started discussion of motion planning.

- Discussed discretization.
- Discussed how we can apply graph-search algorithms to plan in discrete spaces.
- Introduced Breadth First Search (BFS) and Depth First Search (DFS).
- We said that there are two considerations with motion planning:
  - Feasibility (achieved by BFS and DFS).
  - Optimality (we will discuss this today).

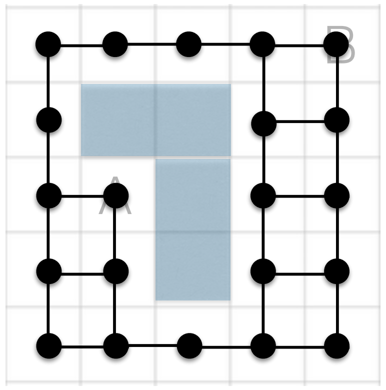
Today: Optimal planning in discrete spaces.

In order to perform optimal planning in discrete spaces, we will modify our search algorithm from the previous lecture a little bit.

Algorithm 1: General Forward Search

```
Q.Insert(A) and mark A as visited;
while $Q$ not empty do
    $x \leftarrow$ Q.GetVertex() ;          This will be implemented differently
    if $x \in x_G$ then
        // $x_G$ is goal set, e.g., {B} ;
        Return SUCCESS;
    end
    for all neighbors $x'$ of $x$ do
        if $x'$ not visited then
            Parent($x'$) $\leftarrow x$;
            Mark $x'$ as visited ;
            Q.Insert($x'$) ;
        else
            Resolve duplicate $x'$ ;          This is new
        end
    end
end
Return FAILURE ;
```

1. Dijkstra's Algorithm [LaValle, Ch. 2.2.2]



Suppose with every edge e in a graph representation of a discrete planning problem, we associate:

$$l(e) : \text{Cost (or "loss") associated with traversing edge } e.$$

The cost defines the performance metric with respect to which we would like to be optimal (e.g., time/distance/energy). The **total cost** of a plan is the sum of the costs of all edges that the plan traverses (to get from A to B). Our goal will be to find a path that minimizes the total cost.

For the algorithms we will think about today, it is helpful to think about three kinds of vertices:

- **"Unvisited".** Vertices that we have not explored yet.
- **"Alive".** These are vertices that Q stores. They are vertices that we have encountered, but have not taken out from Q yet.
- **"Dead".** Vertices that have been visited, and for which every possible neighbor has also been visited (intuitively, these are vertices that don't have any more to contribute to the search).

Main idea behind Dijkstra's algorithm: Implement Q.GetVertex() based on an *estimated cost* to get to a vertex from the start vertex A.

For each vertex, define $C^\star(x)$: the "optimal cost-to-come" from A to x. This is the cost corresponding to the optimal way to get from A to x.

The priority queue Q will be sorted according to an *estimate* $C(x)$ of the optimal cost-to-come $C^\star(x)$. The estimates $C(x)$ will be updated as the algorithm proceeds.

Initially, $C(A) = C^\star(A) = 0$.

Each time a vertex is considered, we compute the estimated cost as:

$$C(x') = C(x) + l(e), \text{ where } e \text{ is the edge connecting } x \text{ to } x'. \tag{1}$$

The line "`Resolve duplicate` x'" of Algorithm 1 above accounts for the fact that if x' has already been visited, it is possible that the newly discovered path to x' is more efficient. Then the cost-to-come estimate $C(x')$ must be lowered and Q must be reordered accordingly.

When does $C(x)$ become $C^\star(x)$ for some vertex x? Once x is "dead" and removed from Q using Q.GetVertex(), we know that x cannot be reached with a lower cost. This can be proved using induction [Proof: Ch. 2.2.2, 2.3.3 in LaValle].

Let's go through an example of Dijkstra's algorithm at work! **See the slides for this.**

Looking at the animation of Dijkstra's algorithm is a little bit frustrating. It explores uniformly in all directions. This seems wasteful. Is there a way to reduce the number of states that are explored?

2. A* algorithm

The A* (pronounced A-star) algorithm uses a heuristic to implement Q.GetVertex(). In particular, it uses:

$$F(x) \triangleq C(x) + H(x), \tag{2}$$

where $C(x)$ is the cost-to-come estimate, and $H(x)$ is an estimate of the "cost-to-go", i.e., the optimal cost to go from x to the goal.

Important: $H(x)$ needs to be an *underestimate* (i.e., a lower bound) on the cost-to-go from x to the goal. This is required for the algorithm to work correctly (we won't discuss why... the reasons are slightly complicated. But keep this in mind!).

We can often get an underestimate for $H(x)$ quite easily in practice. For example, suppose x is a cell (i, j) and suppose that the goal is at cell (i', j'). Then, $|i' - i| + |j' - j|$ is an underestimate of the cost to go (assuming the robot can only move up/down/left/right).

Note: Dijkstra's algorithm is a special case of the A* algorithm (with $H(x) = 0$).