Search and Planning

Lecture 10

Sayan Mitra
Autonomy pipeline

**Sensing**
Physics-based models of camera, LIDAR, RADAR, GPS, etc.

**Perception**
Programs for object detection, lane tracking, scene understanding, etc.

**Decisions and planning**
Programs and multi-agent models of pedestrians, cars, etc.

**Control**
Dynamical models of engine, powertrain, steering, tires, etc.

GEM platform
Control
Dynamical models of engine, powertrain, steering, tires, etc.

Decisions and planning
Programs and multi-agent models of pedestrians, cars, etc.

Perception
Programs for object detection, lane tracking, scene understanding, etc.

Sensing
Physics-based models of camera, LIDAR, RADAR, GPS, etc.
Search in different levels of navigation, planning, decision making, and control

• Global path planner --- invoked at each new checkpoint
  • finds paths from every point in the map to next checkpoint
  • dynamic programming

• Road navigation
  • For each path, the planner rolls out several discrete trajectories that are parallel to the smoothed center of the lane

• Freeform navigation (parking lots)
  • Generate arbitrary trajectories (irrespective of road structure) using modified A*

Junior: The Stanford Entry in the Urban Challenge, Thrun et al., 2008
A search-based strategy for planning

• Represent vehicle state in a *uniform* discrete grid
  • 4D grid: \(x, y, \theta\) (*heading*), \(dir\) (fwd,rev)

• A path (a) over this discrete grid is a start for a plan

• But, the discrete path (a) may not be executable by the vehicle dynamics

• *Hybrid A* solves this problem by shifting the points that represent the discrete cells
  • More on this in the next lecture
Outline

• Uninformed search
• Informed search
• Optimal search: A, A*
Starting from uninformed graph search

• Search for collision free trajectories can be converted to graph search
• Hence, we can solve such problems using the graph search algorithms like (uninformed) Breadth-First Search and Depth-First Search
• However, roadmaps are not just “generic” graphs
  • Some paths are much more preferable with respect to others (e.g., shorter, faster, less costly in terms of fuel/tolls/fees, more stealthy, etc.).
  • Distances have a physical meaning
  • Good guesses for distances can be made, even without knowing optimal paths.
• Can we utilize this information to find efficient paths, efficiently?
Problem statement: find shortest path

• Input: $\langle V, E, w, start, goal \rangle$
  • $V$: (finite) set of vertices
  • $E \subseteq V \times V$: (finite) set of edges
  • $w : E \rightarrow \mathbb{R}_{>0}$: a function that associates to each edge $e$ to a strictly positive weight $w(e)$ (cost, length, time, fuel, prob. of detection)
  • $start, goal \in V$: respectively, start and end vertices.

• Output: $\langle P \rangle$
  • $P$ is a path (starting in start and ending in goal, such that its weight $w(P)$ is minimal among all such paths
  • The weight of a path is the sum of the weights of its edges
  • The graph may be unknown, partially known, or known
Many paths and all weights are often not known upfront
Example: Find the minimal path from s to g:

a simple path P:

w(P):
Search Performance Metrics

• **Soundness**: when a solution is returned, is it guaranteed to be correct path?
• **Completeness**: – is the algorithm guaranteed to find a solution when there is one?
• **Optimality**: How close is the found solution to the best solution?
• **Space complexity**: how much memory is needed?
• **Time complexity**: what is the running time? Can it be used for online planning?
Uniform cost search (Uninformed search)

\[ Q \leftarrow \{ \text{start} \} \]  // maintains paths
\[ \text{// initialize} \]
\[ Q \neq \emptyset : \]
\[ \text{pick (and remove) the path } P \text{ with lowest cost } g = w(P) \text{ from } Q \]
\[ \text{if } \text{head}(P) = \text{goal then return } P ; \]  // Reached the goal
\[ \text{foreach vertex } v \text{ such that } (\text{head}(P), v) \in E, \text{ do} \]
\[ \text{add } \langle v, P \rangle \text{ to } Q ; \]  // for all neighbors
\[ \text{return } \text{FAILURE} ; \]  // Add expanded paths
\[ \text{// nothing left to consider} \]

Note no visited list; Use no information obtained from the environment
Example of Uniform-Cost Search

Q: | Path | Cost |
---|------|------|
⟨s⟩ | 0    |

Path Cost

\[
\langle s \rangle \quad 0
\]
Example of Uniform-Cost Search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a, s⟩</td>
<td>2</td>
</tr>
<tr>
<td>⟨b, s⟩</td>
<td>5</td>
</tr>
</tbody>
</table>

→ ⟨c, a, s⟩ 4
⟨d, a, s⟩ 6
⟨d, c, a, s⟩ 7
Example of Uniform-Cost Search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;b, s&gt;</td>
<td>5</td>
</tr>
<tr>
<td>&lt;d, a, s&gt;</td>
<td>6</td>
</tr>
<tr>
<td>&lt;d, c, a, s&gt;</td>
<td>7</td>
</tr>
</tbody>
</table>

<g, b, s> 10
<g, d, a, s> 8
<g, d, c, a, s> 9

Steps when smallest cost path had 'g' as the head
Remarks on Uniform Cost Search

• UCS is an extension of BFS to the weighted-graph case (UCS = BFS if all edges have the same cost)

• UCS is complete and optimal (assuming costs bounded away from zero)

• UCS is guided by path cost rather than path depth, so it may get in trouble if some edge costs are very small

• Worst-case time and space complexity $O(b^{W^*/\epsilon})$, where $W^*$ is the optimal cost, and $\epsilon$ is such that all edge weights are no smaller than
Greedy or Best-First Search

• UCS explores paths in all directions, with no bias towards the goal state
• What if we try to get “closer” to the goal?
• We need a measure of distance to the goal
  • It would be ideal to use the length of the shortest path...
  • but this is exactly what we are trying to compute!
• We can estimate the distance to the goal through a “heuristic function,” \( h : V \rightarrow \mathbb{R}_{\geq 0} \). E.g., the Euclidean distance to the goal (as the crow flies)
  • \( h(v) \) is the estimate of the distance from \( v \) to goal
• A reasonable strategy is to always try to move in such a way to minimize the estimated distance to the goal: this is the basic idea of the greedy (best-first) search
Greedy/Best-first search

\[ Q \leftarrow \text{(start)} \]  
\[ \text{while } Q \neq \emptyset:\]  
\[ \text{pick (and remove) the path } P \text{ with lowest heuristic cost } h(\text{head}(P)) \text{ from } Q \]  
\[ \text{if } \text{head}(P) = \text{goal} \text{ then return } P \]  
\[ \text{foreach vertex } v \text{ such that } (\text{head}(P), v) \in E, \text{ do} \]  
\[ \text{add } \langle v, P \rangle \text{ to } Q; \]  
\[ \text{return } FAILURE; \]  
\[ \text{// initialize queue with start} \]  
\[ \text{// Reached the goal} \]  
\[ \text{// for all neighbors} \]  
\[ \text{// Add expanded paths} \]  
\[ \text{// nothing left to consider} \]
Example of Greedy search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle s \rangle$</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
<th>$h$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle s \rangle$</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>
```
Example of Greedy search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨s⟩</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

→ ⟨a, s⟩ 2 2
→ ⟨b, a⟩ 5 3
→ ⟨c, a, s⟩ 4 1
  ⟨d, a, s⟩ 6 4
  ⟨d, c, a, s⟩ 7 4
→ ⟨g, b, s⟩ 10 0  

not optimal
Example of Greedy search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a, s⟩</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>⟨b, s⟩</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>
Example of Greedy search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨s⟩</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

Path cost and heuristic values for the greedy search algorithm.

Diagram:
- Nodes: s, a, d, c, g, b
- Edges with costs: s → a (2), a → c (2), a → d (4), c → g (3), d → b (2), d → g (2), g → b (5), b → g (5), s → b (5)
- Initial path: ⟨s⟩ with cost 0 and heuristic 10
Remarks on greedy/best-first search

• Greedy (Best-First) search is similar to Depth-First Search
  • keeps exploring until it has to back up due to a dead end

• **Not complete** (why?) and not optimal, but is often fast and efficient, depending on the heuristic function \( h \)

• Worst-case time and space complexity?
A search: informed search

• The problems
  • UCS is optimal, but may wander around a lot before finding the goal
  • Greedy is not optimal, but can be efficient, as it is heavily biased towards moving towards the goal. The non-optimality comes from neglecting “the past.”

• The idea
  • Keep track both of the cost of the partial path to get to a vertex, say \( g(v) \), and of the heuristic function estimating the cost to reach the goal from a vertex, \( h(v) \)
  • In other words, choose as a “ranking” function the sum of the two costs:
    \[
    f(v) = g(v) + h(v)
    \]
  • \( g(v) \): cost-to-come (from the start to \( v \))
  • \( h(v) \): cost-to-go estimate (from \( v \) to the goal)
  • \( f(v) \): estimated cost of the path (from the start to \( v \) and then to the goal)
A search

\[ Q \leftarrow \text{(start)} \]  // initialize queue with start

while \( Q \neq \emptyset \):

pick (and remove) path \( P \) with lowest estimated cost
\[ f(P) = g(P) + h(\text{head}(P)) \]

if \( \text{head}(P) = \text{goal} \) then return \( P \)  // Reached the goal

foreach vertex \( v \) such that \( (\text{head}(P), v) \in E \), do

add \( (v, P) \) to \( Q \);  // Add expanded paths

return FAILURE;  // nothing left to consider
Example of A search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>g</th>
<th>h</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨s⟩</td>
<td>0</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Diagram: A graph showing connections between nodes, with weights on the edges.
Example of A search

<table>
<thead>
<tr>
<th>Path</th>
<th>g</th>
<th>h</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a, s⟩</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>⟨b, s⟩</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

⟨c, a, s⟩ 4 1 5
⟨d, a, s⟩ 6 5 11
⟨d, c, a, s⟩ 7 5 12
⟨g, b, a⟩ 10 0 10

not optimal
Example of A search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>g</th>
<th>h</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a, s⟩</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>⟨b, s⟩</td>
<td>5</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

- Path: g, h, f
- ⟨a, s⟩: 2, 2, 4
- ⟨b, s⟩: 5, 3, 8
Remarks on A search

• A search is similar to UCS, with a bias induced by the heuristic h
• If $h = 0$, $A = UCS$.
• The A search is complete, but is not optimal
  • What is wrong? (Recall that if $h = 0$ then $A = UCS$, and hence optimal...)

A * Search

• Choose an admissible heuristic, i.e., such that $h(v) \leq h^*(v)$
  • $h^*(v)$ is the “optimal” heuristic---perfect cost to go
  • To be admissible $h(v)$ should be at most $h^*(v)$
  • A search with an admissible heuristic is called A* --- guaranteed to find optimal path
Example of A* search

Q:

<table>
<thead>
<tr>
<th>Path</th>
<th>g</th>
<th>h</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle s \rangle$</td>
<td>0</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

changed $h$

finds $\langle g, d, a, s \rangle$

optimal path
Proof of optimality of A*

• Let \( w^* \) be the cost of the optimal path
• Suppose for the sake of contradiction, that A* returns \( P \) with \( w(P) > w^* \)
• Find the first unexpanded node on the optimal path \( P^* \); call it \( n \)
• \( f(n) > w(P) \), otherwise \( n \) would have been expanded
• \( f(n) = g(n) + h(n) \)
  \[ = g^*(n) + h(n) \quad [\text{since } n \text{ is on the optimal path}] \]
  \[ \leq g^*(n) + h^*(n) \quad [\text{since } h \text{ is admissible}] \]
  \[ = f^*(n) = w^* \quad [\text{by def. of } f, \text{ and since } w^* \text{ is the cost of the optimal path}] \]
• Hence \( w^* \geq f(n) = w(P) \), which is a contradiction
Admissible heuristics

• How to find an admissible heuristic? i.e., a heuristic that never overestimates the cost-to-go.

• Examples of admissible heuristics
  • \( h(v) = 0 \): this always works! However, it is not very useful, \( A^* = UCS \)
  • \( h(v) = \text{distance}(v, g) \) when the vertices of the graphs are physical locations
  • \( h(v) = \|v - g\|_p \), when the vertices of the graph are points in a normed vector space

• A general method
  • Choose \( h \) as the optimal cost-to-go function for a relaxed problem, that is easy to compute
  • Relaxed problem: ignore some of the constraints in the original problem
Admissible heuristics for the 8-puzzle

Which of the following are admissible heuristics?

• $h = 0$
  - YES, always good

• $h = 1$
  - not valid in goal state

• $h =$ number of tiles in the wrong position
  - YES, “teleport” each tile to the goal in one move

• $h =$ sum of (Manhattan) distance between tiles and their goal position
  - YES, move each tile to the goal ignoring other tiles.

Initial state:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Goal state:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A partial order of heuristic functions

• Some heuristics are better than others
  • $h = 0$ is an admissible heuristic, but is not very useful
  • $h = h^*$ is also an admissible heuristic, and it the “best” possible one (it give us the optimal path directly, no searches/backtracking)

• Partial order
  • We say that $h_1$ dominates $h_2$ if $h_1(v) \geq h_2(v)$ for all vertices $v$
  • $h^*$ dominates all admissible heuristics, and 0 is dominated by all admissible heuristics

• Choosing the right heuristic
  • In general, we want a heuristic that is as close to $h^*$ as possible
  • However, such a heuristic may be too complicated to compute
  • There is a tradeoff between complexity of computing $h$ and the complexity of the search
Consistent heuristics

• An additional useful property for A* heuristics is called **consistency**
  • A heuristic $h: X \rightarrow \mathbb{R}_{\geq 0}$ is said **consistent** if $h(u) \leq w(e = (u, v)) + h(v)$, $\forall (u, v) \in E$
  • In other words, a consistent heuristics satisfies a triangle inequality

• If $h$ is a consistent heuristics, then $f = g + h$ is non-decreasing along paths: $f(v) = g(v) + h(v) = g(u) + w(u, v) + h(v) \geq f(u)$

• Hence, the values of $f$ on the sequence of nodes expanded by A* is non-decreasing: the first path found to a node is also the optimal path $\Rightarrow$ no need to compare costs!
Hybrid A*

- Represent vehicle state in a **uniform** discrete grid
  - 4D grid: $x, y, \theta$ (heading), $dir$ (fwd, rev)
- If the current coordinate is $\langle x, y, \theta \rangle$ and those coordinates lie in cell $c_i$ then the **representative continuous state** for cell $c_i$ will be $x_i = x, y_i = y, \theta_i = \theta$
- After applying control input $u$ to vehicle, suppose the predicted state is $x', y', \theta'$
  - $x', y', \theta' = f(x, y, \theta, u); \dot{x} = \cdots$
  - representative for $c_j = x', y', \theta'$
  - This defines a transition from $c_i$ to $c_j$
- More details in the next lecture
Summary

• A* algorithm combines cost-to-come $g(v)$ and a heuristic function $h(v)$ for cost-to-go to find shortest path
  • informed search

• heuristic function must be *admissible* $h(v) \leq h^*(v)$
  • Never over-estimate the actual cost to go
  • Are all $h(v)$ values needed?
  • What if $h$ is not admissible
  • How to find heuristics
Next: Dynamic programming/Dijkstra

• The optimality principle
  • Let $P = (s, \ldots, v, \ldots g)$ be an optimal path (from $s$ to $g$).
  • Then, for any $v \in P$, the sub-path $S = (v, \ldots, g)$ is itself an optimal path (from $v$ to $g$).

• Using the optimality principle
  • Essentially, optimal paths are made of optimal paths. Hence, we can construct long complex optimal paths by putting together short optimal paths, which can be easily computed. Fundamental formula in dynamic programming: $h^*(u) = \min_{(u,v) \in E} [w((u, v)) + h^*(v)]$. Typically, it is convenient to build optimal paths working backwards from the goal.