Incremental Path Planning
Continuous Planning and Dynamic A*
Prof. Brian Williams
(help from Ihsiang Shu)
16.412/6.834 Cognitive Robotics
March 16th, 2004

Outline

- Optimal Path Planning in Partially Known Environments.
- Continuous Optimal Path Planning
  - Dynamic A*
  - Incremental A* (LRTA*) [Appendix]

[Zellinsky, 92]

1. Generate global path plan from initial map.

2. Repeat until goal reached or failure:
   - Execute next step in current global path plan
   - Update map based on sensors.
   - If map changed generate new global path from map.

Compute Optimal Path

```
J M N O
E I L G
B D H K
S A C F
```

Begin Executing Optimal Path

```
J M N O
E I L G
B D H K
S A C F
```

- Robot moves along backpointers towards goal.
- Uses sensors to detect discrepancies along way.

Obstacle Encountered!

```
J M N O
E I L G
B D H K
S A C F
```

- At state A, robot discovers edge from D to H is blocked (cost 5,000 units).
- Update map and reinvoke planner.
Continue Path Execution

A's previous path is still optimal.
Continue moving robot along back pointers.

Second Obstacle, Replan!

At C robot discovers blocked edges from C to F and H (cost 5,000 units).
Update map and reinvoke planner.

Path Execution Achieves Goal

Follow back pointers to goal.
No further discrepancies detected, goal achieved!

Outline

- Optimal Path Planning in Partially Known Environments.
- Continuous Optimal Path Planning
  - Dynamic A*
  - Incremental A* (LRTA*) [Appendix]

What is Continuous Optimal Path Planning?

- Supports search as a repetitive online process.
- Exploits similarities between a series of searches to solve much faster than solving each search starting from scratch.
- Reuses the identical parts of the previous search tree, while updating differences.
- Solutions guaranteed to be optimal.
- On the first search, behaves like traditional algorithms.
  - D* behaves exactly like Dijkstra's.
  - Incremental A* A* behaves exactly like A*.

Dynamic A* (aka D*)
[Stenz, 94]

1. Generate global path plan from initial map.
2. Repeat until Goal reached, or failure.
   - Execute next step of current global path plan.
   - Update map based on sensor information.
   - Incrementally update global path plan from map changes.

$\rightarrow$ 1 to 3 orders of magnitude speedup relative to a non-incremental path planner.
Map and Path Concepts
- \(c(X,Y)\): Cost to move from Y to X.
  \(c(X,Y)\) is undefined if move disallowed.
- \(\text{Neighbors}(X)\): Any Y such that \(c(X,Y)\) or \(c(Y,X)\) is defined.
- \(o(G,X)\): True optimal path cost to Goal from X.
- \(h(G,X)\): Estimate of optimal path cost to goal from X.
- \(b(X) = Y\): backpointer from X to Y.
  Y is the first state on path from X to G.

D* Search Concepts
- State tag \(t(X)\):
  - NEW: has no estimate \(h\).
  - OPEN: estimate needs to be propagated.
  - CLOSED: estimate propagated.
- OPEN list:
  States with estimates to be propagated to other states.
  - States on list tagged OPEN
  - Sorted by key function \(k\) (defined below).

D* Fundamental Search Concepts
- \(k(G,X)\): key function
  - Minimum of \(h(G,X)\) before modification, and \(\forall h(G,X)\) since X was placed on the OPEN list.
  - Lowered state: \(k(G,X) = \text{current } h(G,X)\), propagate decrease to descendants and other nodes.
  - Raised state: \(k(G,X) < \text{current } h(G,X)\), propagate increase to descendants and other nodes.
  - Try to find alternate shorter paths.

Running D* First Time on Graph
Initially
- Mark G Open and Queue it
- Mark all other states New
- Run Process_States on queue until path found or empty.
When edge cost \(c(X,Y)\) changes
- If X is marked Closed, then
  - Update \(h(X)\)
  - Mark X open and queue with key \(h(X)\).

Use D* to Compute Initial Path
- States initially tagged NEW (no cost determined yet).

Use D* to Compute Initial Path
- Add Goal node to the OPEN list.
- Process OPEN list until the robot's current state is CLOSED.
Process State: New or Lowered State

- Remove from Open list, state X with lowest k
- If X is a new/ lowered state, its path cost is optimal!
  Then propagate to each neighbor Y
  - If Y is New, give it an initial path cost and propagate.
  - If Y is a descendant of X, propagate any change.
- Else, if X can lower Y's path cost,
  Then do so and propagate.

Use $D^*$ to Compute Initial Path

- Add new neighbors of G onto the OPEN list
- Create backpointers to G.

Use $D^*$ to Compute Initial Path

- Add new neighbors of G onto the OPEN list
- Create backpointers to G.

Use $D^*$ to Compute Initial Path

- Add new neighbors of K onto the OPEN list
- Create backpointers.

Use $D^*$ to Compute Initial Path

- Add new neighbors of L, then O onto the OPEN list
- Create backpointers.

Use $D^*$ to Compute Initial Path

- Continue until current state S is closed.
Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
<th>h = 3</th>
<th>h = 2</th>
<th>h = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0, G)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>2 (1, K)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>3 (1, L)</td>
<td>(3, C)</td>
<td>(3, D)</td>
<td>(3, E)</td>
</tr>
<tr>
<td>4 (2, F)</td>
<td>(4, A)</td>
<td>(4, B)</td>
<td>(4, C)</td>
</tr>
</tbody>
</table>

- Continue until current state S is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
<th>h = 3</th>
<th>h = 2</th>
<th>h = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0, G)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>2 (1, K)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>3 (1, L)</td>
<td>(3, C)</td>
<td>(3, D)</td>
<td>(3, E)</td>
</tr>
<tr>
<td>4 (2, F)</td>
<td>(4, A)</td>
<td>(4, B)</td>
<td>(4, C)</td>
</tr>
</tbody>
</table>

- Continue until current state S is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
<th>h = 3</th>
<th>h = 2</th>
<th>h = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0, G)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>2 (1, K)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>3 (1, L)</td>
<td>(3, C)</td>
<td>(3, D)</td>
<td>(3, E)</td>
</tr>
<tr>
<td>4 (2, F)</td>
<td>(4, A)</td>
<td>(4, B)</td>
<td>(4, C)</td>
</tr>
</tbody>
</table>

- Continue until current state S is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
<th>h = 3</th>
<th>h = 2</th>
<th>h = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0, G)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>2 (1, K)</td>
<td>(2, F)</td>
<td>(2, H)</td>
<td>(2, I)</td>
</tr>
<tr>
<td>3 (1, L)</td>
<td>(3, C)</td>
<td>(3, D)</td>
<td>(3, E)</td>
</tr>
<tr>
<td>4 (2, F)</td>
<td>(4, A)</td>
<td>(4, B)</td>
<td>(4, C)</td>
</tr>
</tbody>
</table>

- Continue until current state S is closed.
Use D* to Compute Initial Path

- Continue until current state S is closed.

D* Completed Initial Path

- Done: Current state S is closed, and Open list is empty.

Begin Executing Optimal Path

- Robot moves along backpointers towards goal
- Uses sensors to detect discrepancies along way.
Running D* After Edge Cost Change

When edge cost $c(X,Y)$ changes

- If $X$ is marked Closed, then
  - Update $h(X)$
  - Mark $X$ open and queue, key is new $h(X)$.
- Run Process State on queue
  - until path to current state is shown optimal,
  - or queue Open List is empty.

At state A, robot discovers edge D to H is blocked off (cost 5,000 units).
Update map and reinvoke D*

Running D* After Edge Cost Change

When edge cost $c(X,Y)$ changes

- If X is marked Closed, then
  - Update $h(X)$
  - Mark X open and queue, key is new $h(X)$.
- Run Process State on queue
  - until path to current state is shown optimal,
  - or queue Open List is empty.

At state A, robot discovers edge D to H is blocked off (cost 5,000 units).
Update map and reinvoke D*
D* Update From First Obstacle

![Diagram]

- All neighbors of D have consistent h-values.
- No further propagation needed.

Continue Path Execution

![Diagram]

- A's path optimal.
- Continue moving robot along backpointers.

Second Obstacle!

![Diagram]

- At C robot discovers blocked edges C to F and H (cost 5,000 units).
- Update map and reinvoke D* until H(current position optimal).

D* Update From Second Obstacle

![Diagram]

- Processing F raises descendant C's cost, and propagates.
- Processing H does nothing.

D* Update From Second Obstacle

![Diagram]

- Don't change C's path to A (yet).
- Instead, propagate increase to A.

D* Update From Second Obstacle

![Diagram]

- C may be suboptimal, check neighbors: Better path through A!
- However, A may be suboptimal, and updating would create a loop!
**Process_State: Raised State**

- If X is a raise state its cost might be suboptimal.
- Try reducing cost of X using an optimal neighbor Y.
  - \( h(Y) = h(X) \) (before it was raised)
  - Propagate X’s cost to each neighbor Y.
    - If Y is New, then give it an initial path cost and propagate.
    - If Y is a descendant of X, then propagate any change.
    - If X can lower Y’s path cost, postpone.
      - Postponement avoids creating cycles.

**D* Update From Second Obstacle**

- A may not be optimal, check neighbors for better path.
- Transitioning to D is better, and D's path is optimal, so update A.

**Process_State: New or Lowered State**

- Remove from Open list, state X with lowest k.
- If X is a new/lowered state its path cost is optimal, then propagate to each neighbor Y.
  - If Y is New, give it an initial path cost and propagate.
  - If Y is a descendant of X, propagate any change.
  - Else, if X can lower Y’s path cost, then do so and propagate.

**D* Update From Second Obstacle**

- A can improve neighbor C, so queue C.
- Current state reached, so Process_State terminates.
Complete Path Execution

D* Pseudo Code

Recap: Continuous Optimal Planning

Recap: Dynamic A*

- Supports search as a repetitive online process.
- Exploits similarities between a series of searches to solve much faster than from scratch.
- Reuses the identical parts of the previous search tree, while updating differences.
- Solutions guaranteed to be optimal.
- On the first search, behaves like traditional Dijkstra.

Follow back pointers to Goal.
No further discrepancies detected; goal achieved!