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]

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

<table>
<thead>
<tr>
<th></th>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td>I</td>
<td>L</td>
<td>G</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>H</td>
<td>K</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>A</td>
<td>C</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

Obstacle Encountered!

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

Begin Executing Optimal Path

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

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

Second Obstacle, Replan!

- At C robot discovers blocked edge 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.

   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 $t(X)$:
  - $\text{NEW}$: has no estimate $h$.
  - $\text{OPEN}$: estimate needs to be propagated.
  - $\text{CLOSED}$: 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
  - all values assumed by $h(G,X)$ 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
- 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

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
</tr>
</tbody>
</table>

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

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>(0,G)</td>
</tr>
</tbody>
</table>

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

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
</tr>
</tbody>
</table>

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

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>(0,G)</td>
</tr>
</tbody>
</table>

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

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
</tr>
</tbody>
</table>

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

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>J</th>
<th>M</th>
<th>N</th>
<th>O</th>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>NEW</td>
<td>(0,G)</td>
</tr>
</tbody>
</table>

- Add new neighbors of L onto the Open list
- Create backpointers.

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

<table>
<thead>
<tr>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,G)</td>
</tr>
<tr>
<td>(1,K) (1,L) (1,O)</td>
</tr>
<tr>
<td>(1,L) (1,O) (2,F) (2,H)</td>
</tr>
<tr>
<td>(2,F) (2,H) (2,I) (2,N)</td>
</tr>
<tr>
<td>(2,H) (2,I) (2,N) (3,C)</td>
</tr>
<tr>
<td>(2,I) (2,N) (3,C) (3,D)</td>
</tr>
<tr>
<td>(0,G) (3,C) (3,D) (3,E) (3,M)</td>
</tr>
<tr>
<td>(3,C) (3,D) (3,E) (3,M)</td>
</tr>
</tbody>
</table>

Continue until current state $S$ is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,G)</td>
</tr>
<tr>
<td>(1,K) (1,L) (1,O)</td>
</tr>
<tr>
<td>(1,L) (1,O) (2,F) (2,H)</td>
</tr>
<tr>
<td>(2,F) (2,H) (2,I) (2,N)</td>
</tr>
<tr>
<td>(2,H) (2,I) (2,N) (3,C)</td>
</tr>
<tr>
<td>(2,I) (2,N) (3,C) (3,D)</td>
</tr>
<tr>
<td>(0,G) (3,C) (3,D) (3,E) (3,M)</td>
</tr>
<tr>
<td>(3,C) (3,D) (3,E) (3,M)</td>
</tr>
</tbody>
</table>

Continue until current state $S$ is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,G)</td>
</tr>
<tr>
<td>(1,K) (1,L) (1,O)</td>
</tr>
<tr>
<td>(1,L) (1,O) (2,F) (2,H)</td>
</tr>
<tr>
<td>(2,F) (2,H) (2,I) (2,N)</td>
</tr>
<tr>
<td>(2,H) (2,I) (2,N) (3,C)</td>
</tr>
<tr>
<td>(2,I) (2,N) (3,C) (3,D)</td>
</tr>
<tr>
<td>(0,G) (3,C) (3,D) (3,E) (3,M)</td>
</tr>
<tr>
<td>(3,C) (3,D) (3,E) (3,M)</td>
</tr>
</tbody>
</table>

Continue until current state $S$ is closed.

Use D* to Compute Initial Path

<table>
<thead>
<tr>
<th>OPEN List</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,G)</td>
</tr>
<tr>
<td>(1,K) (1,L) (1,O)</td>
</tr>
<tr>
<td>(1,L) (1,O) (2,F) (2,H)</td>
</tr>
<tr>
<td>(2,F) (2,H) (2,I) (2,N)</td>
</tr>
<tr>
<td>(2,H) (2,I) (2,N) (3,C)</td>
</tr>
<tr>
<td>(2,I) (2,N) (3,C) (3,D)</td>
</tr>
<tr>
<td>(0,G) (3,C) (3,D) (3,E) (3,M)</td>
</tr>
<tr>
<td>(3,C) (3,D) (3,E) (3,M)</td>
</tr>
</tbody>
</table>

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

- Continue until current state S is closed.

Begin Executing Optimal Path

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

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

Update map and rerun 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.

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, and Y is suboptimal,
    - Postpone: Queue X to propagate when optimal (reach current h(X))
    - Postpone: Queue Y to propagate when optimal (reach current h(Y)).
  - Postponement avoids creating cycles.

D* Update From First Obstacle

Assign cost of 5,000 for D to H
Propagate changes starting at H

D* Update From First Obstacle

Open List

- Raise cost of H's descendant D, and propagate.

D* Update From First Obstacle

Function: Modify-Cost(X,Y,eval)
1:  c(X,Y) = eval
2:  if t(X) = CLOSED
    then Insert(X,h(X))
3:  return Get-Kmin()
**D* Update From First Obstacle**

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

**Continue Path Execution**

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

**Second Obstacle!**

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

**Function: Modify-Cost(X,Y,eval)**

1. \( c(X,Y) = \text{eval} \)
2. if \( t(X) = \text{CLOSED} \) then Insert(X,h(X))
3. return Get-Kmin()

**D* Update From Second Obstacle**

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

**D* Update From Second Obstacle**

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

**D* Update From Second Obstacle**

- Don’t change C’s path to A (yet).
- Instead, propagate increase to A.
**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, then do so and propagate.
- Postpone: Queue X to propagate when optimal (reach current \( h(X) \)).

**D* Update From Second Obstacle**

- Process State: New or Lowered State
  - Remove from open list, state X with lowest k.
  - If X is a new/lowered state it's path cost is optimal, then propagate 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.
  - Else, if X can lower Y's path cost, then do so and propagate.

**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.
Complete Path Execution

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

Recap: Continuous Optimal Planning

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.

   1 to 3 orders of magnitude speedup relative to a non-incremental path planner.

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.