

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

h=4	h=3	h=2	h=1
J	M	N	O
h=3	h=2	h=1	h=0
E	I	L	G
h=4	h=3	h=2	h=1
B	D	H	K
h=5	h=4	h=3	h=2
S	A	C	F

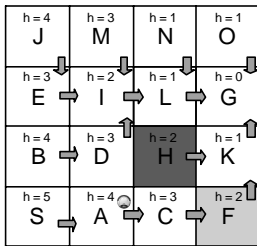
- Robot moves along backpointers towards goal.
- Uses sensors to detect discrepancies along way.

Obstacle Encountered!

h=4	h=3	h=1	h=1
J	M	N	O
h=3	h=2	h=1	h=0
E	I	L	G
h=4	h=3	h=2	h=1
B	D	H	K
h=5	h=4	h=3	h=2
S	A	C	F

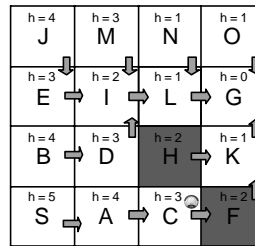
- At state A, robot discovers edge from D to H is blocked (cost 5,000 units).
- Update map and reinvok planner.

Continue Path Execution



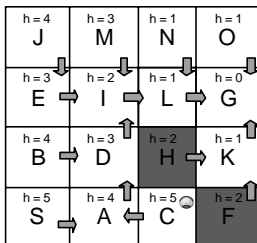
- A's previous path is still optimal.
- Continue moving robot along back pointers.

Second Obstacle, Replan!



- At C robot discovers blocked edges from C to F and H (cost 5,000 units).
- Update map and reinvoked 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* 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.
- $Neighbors(X)$:
 - 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
 - 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

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

States initially tagged NEW (no cost determined yet).

Use D* to Compute Initial Path

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

OPEN List	
1	(0, G)

8: if $h(X) = h(Y)$ then
 9: for each neighbor Y of X:
 10: if $t(Y) = \text{NEW}$ or
 11: $(b(Y) = X \text{ and } h(Y) > h(X) + c(X, Y))$ or
 12: $(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$ then
 13: $b(Y) = X; \text{Insert}(Y, h(X) + c(X, Y))$

- 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

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

OPEN List	
1	(0,G)

- if $kold = h(X)$ then
- for each neighbor Y of X:
 - if $(Y) = NEW$ or
 - $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 - $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 - $b(Y) = X; Insert(Y, h(X) + c(X,Y))$

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

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)

- if $kold = h(X)$ then
- for each neighbor Y of X:
 - if $(Y) = NEW$ or
 - $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 - $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 - $b(Y) = X; Insert(Y, h(X) + c(X,Y))$

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

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)

- if $kold = h(X)$ then
- for each neighbor Y of X:
 - if $(Y) = NEW$ or
 - $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 - $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 - $b(Y) = X; Insert(Y, h(X) + c(X,Y))$

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

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)

- if $kold = h(X)$ then
- for each neighbor Y of X:
 - if $(Y) = NEW$ or
 - $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 - $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 - $b(Y) = X; Insert(Y, h(X) + c(X,Y))$

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

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)

- Continue until current state S is closed.

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)
6	(2,I) (2,N) (3,C) (3,D)

Continue until current state S is closed.

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)
6	(2,I) (2,N) (3,C) (3,D)
7	(2,N) (3,C) (3,D) (3,E) (3,M)

Continue until current state S is closed.

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)
6	(2,I) (2,N) (3,C) (3,D)
7	(2,N) (3,C) (3,D) (3,E) (3,M)
8	(3,C) (3,D) (3,E) (3,M)

Continue until current state S is closed.

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)
6	(2,I) (2,N) (3,C) (3,D)
7	(2,N) (3,C) (3,D) (3,E) (3,M)
8	(3,C) (3,D) (3,E) (3,M)
9	(3,D) (3,E) (3,M) (4,A)

Continue until current state S is closed.

Use D* to Compute Initial Path

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

OPEN List	
1	(0,G)
2	(1,K) (1,L) (1,O)
3	(1,L) (1,O) (2,F) (2,H)
4	(2,F) (2,H) (2,I) (2,N)
5	(2,H) (2,I) (2,N) (3,C)
6	(2,I) (2,N) (3,C) (3,D)
7	(2,N) (3,C) (3,D) (3,E) (3,M)
8	(3,C) (3,D) (3,E) (3,M)
9	(3,D) (3,E) (3,M) (4,A)
10	(3,E) (3,M) (4,A) (4,B)

Continue until current state S is closed.

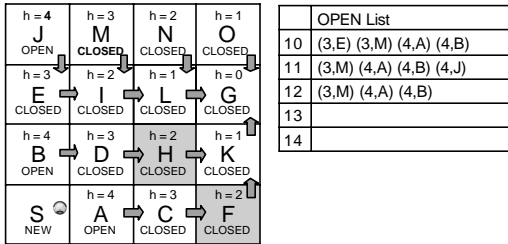
Use D* to Compute Initial Path

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

OPEN List	
10	(3,E) (3,M) (4,A) (4,B)
11	(3,M) (4,A) (4,B) (4,J)
12	
13	
14	

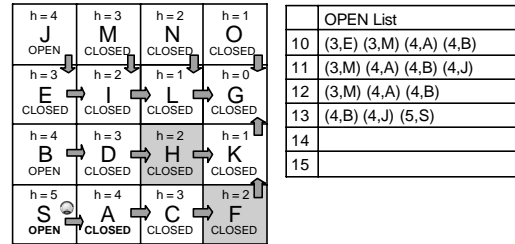
Continue until current state S is closed.

Use D* to Compute Initial Path



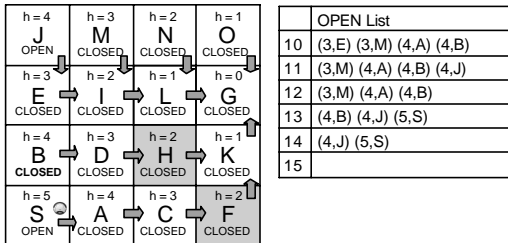
Continue until current state S is closed.

Use D* to Compute Initial Path



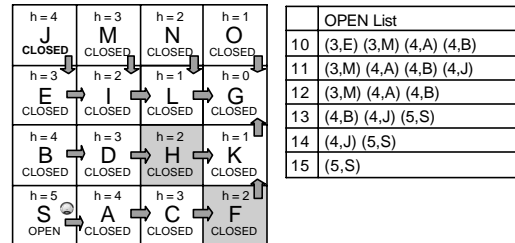
Continue until current state S is closed.

Use D* to Compute Initial Path



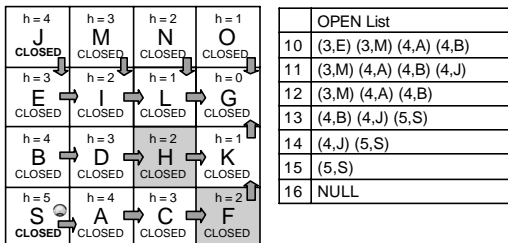
Continue until current state S is closed.

Use D* to Compute Initial Path



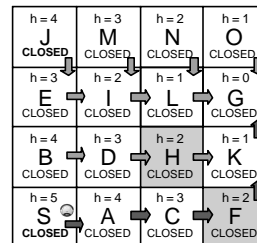
Continue until current state S is closed.

D* Completed Initial Path



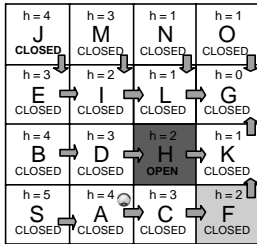
Done: Current state S is closed, and Open list is empty.

Begin Executing Optimal Path



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

Obstacle Encountered!



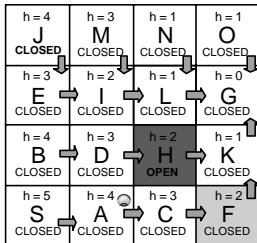
- At state A, robot discovers edge D to H is blocked off (cost 5,000 units).
- Update map and reinvok D*

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.

D* Update From First Obstacle

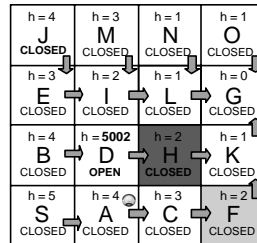


OPEN List	
1	(2,H)
2	
3	
4	

Function: `Modify-Cost(X,Y,eval)`
 1: $c(X,Y) = eval$
 2: if $h(X) = CLOSED$ then $insert(X,h(X))$
 3: return `Get-Kmin()`

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

D* Update From First Obstacle



OPEN List	
1	(2,H)
2	(3,D)
3	
4	

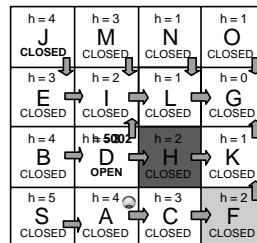
8: if $kold = h(X)$ then
 9: for each neighbor Y of X:
 10: if $h(Y) = NEW$ or
 11: $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 12: $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 13: $b(Y) = X; insert(Y, h(X) + c(X,Y))$

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

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: Queue X to propagate when optimal (reach current $h(X)$)
 - If Y can lower X's path cost, and Y is suboptimal,
 - Postpone: Queue Y to propagate when optimal (reach current $h(Y)$).
- Postponement avoids creating cycles.

D* Update From First Obstacle

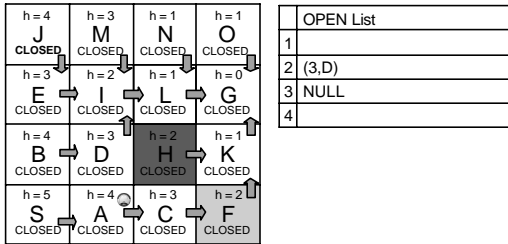


OPEN List	
1	(2,H)
2	(3,D)
3	
4	

4: if $kold < h(X)$ then
 5: for each neighbor Y of X:
 6: if $h(Y) = kold$ and $h(X) > h(Y) + c(Y,X)$ then
 7: $b(X) = Y; h(X) = h(Y) + c(Y,X)$

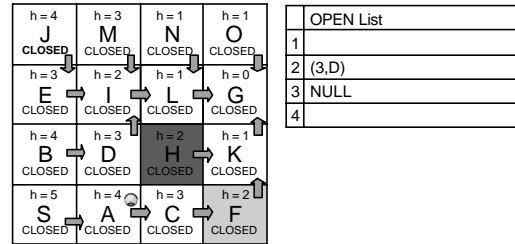
- D may not be optimal, check neighbors for better path.
- Transitioning to I is better, and I's path is optimal, so update D.

D* Update From First Obstacle



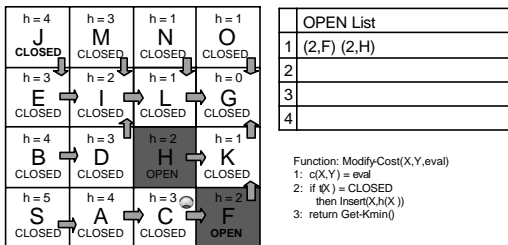
- All neighbors of D have consistent hvalues.
- No further propagation needed.

Continue Path Execution



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

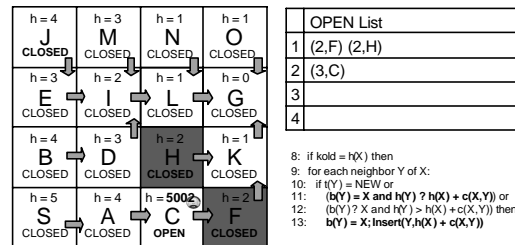
Second Obstacle!



Function: Modify-Cost(X,Y,eval)
 1: $c(X,Y) = eval$
 2: if $(X) = CLOSED$ then $Insert(X,h(X))$
 3: return $Get-Kmin()$

- 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).

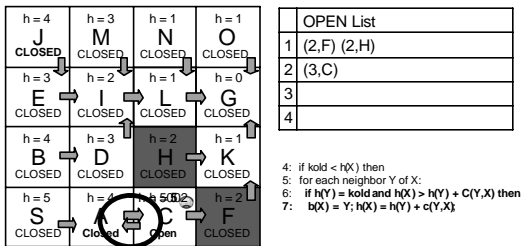
D* Update From Second Obstacle



8: if $kold = h(X)$ then
 9: for each neighbor Y of X:
 10: if $t(Y) = NEW$ or
 11: $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ or
 12: $(b(Y) ? X \text{ and } h(Y) > h(X) + c(X,Y))$ then
 13: $b(Y) = X; Insert(Y,h(X) + c(X,Y))$

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

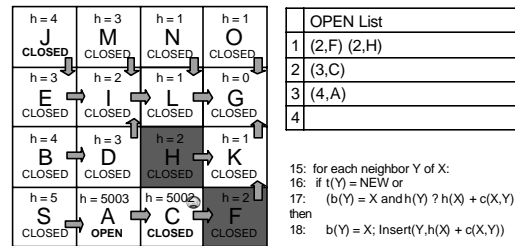
D* Update From Second Obstacle



4: if $kold < h(X)$ then
 5: for each neighbor Y of X:
 6: if $h(Y) = kold \text{ and } h(X) > h(Y) + c(Y,X)$ then
 7: $h(X) = Y; h(X) = h(Y) + c(Y,X)$

- 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



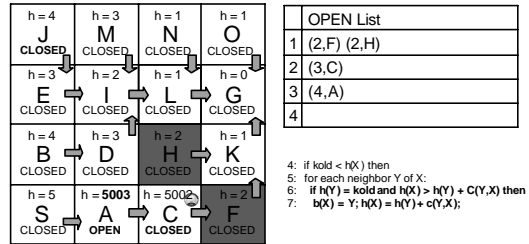
15: for each neighbor Y of X:
 16: if $t(Y) = NEW$ or
 17: $(b(Y) = X \text{ and } h(Y) ? h(X) + c(X,Y))$ then
 18: $b(Y) = X; Insert(Y,h(X) + c(X,Y))$

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

Process_State: Raised State

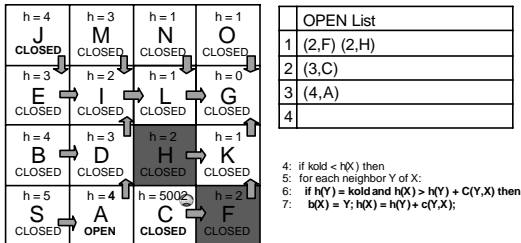
- If X is a raise state its cost might be suboptimal.
 - Try reducing cost of X using an *optimal neighbor Y*.
 - $h(Y) = \lceil h(X) \rceil$ before it was raised
 - propagate X's cost to each neighbor Y
 - If Y is New, Then give it an initial path cost and propagate.
 - If Y is a descendant of X, Then propagate ANY change.
 - If X can lower Y's path cost,
 - Postpone: Queue X to propagate when optimal (reach current $h(X)$)
 - If Y can lower X's path cost, and Y is suboptimal,
 - Postpone: Queue Y to propagate when optimal (reach current $h(Y)$).
- 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

D* Update From Second Obstacle

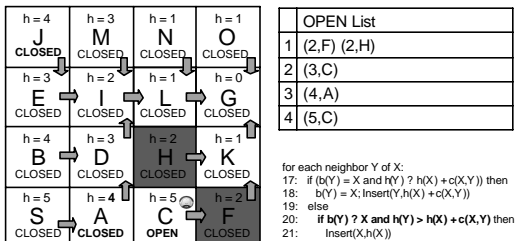


- A may not be optimal, check neighbors for better path.
- Transitioning to D is better, and D's path is optimal, so update A

Process_State: New or Lowered State

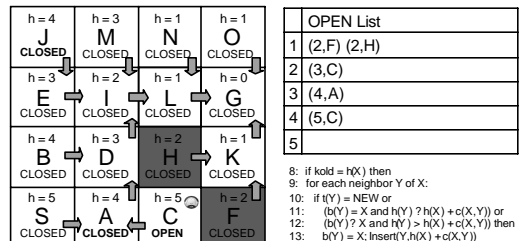
- Remove from Open list, state X with lowest k
- If X is a new/lowered state its path cost is optimal, Then propagate to each neighbor Y
 - If Y is New, give it an initial path cost and propagate.
 - If Y is a descendant of X, propagate any change.
 - Else, if X can lower Y's path cost, Then do so and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

D* Update From Second Obstacle



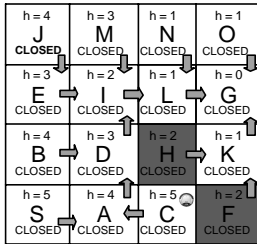
- A can improve neighbor C, so queue C.

D* Update From Second Obstacle



- C lowered to optimal; no neighbors affected.
- Current state reached, so Process_State terminates.

Complete Path Execution



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

D* Pseudo Code

```

Function: Process-State()
1: X = Min-State()
2: if X=NULL then return -1
3: kold = Get-Kmin(); Delete(X)
4: if kold < h(X) then
5:   for each neighbor Y of X:
6:     if h(Y) = kold and h(X) > h(Y) + c(Y,X) then
7:       b(X) = Y; h(X) = h(Y) + c(Y,X)
8:   if kold = h(X) then
9:     for each neighbor Y of X:
10:      if t(Y) = NEW or
11:         (b(Y) = X and h(Y) ? h(X) + c(X,Y) or
12:          (b(Y) ? X and h(X) > h(X) + c(X,Y)) then
13:        b(Y) = X; Insert(Y,h(X) + c(X,Y))
14:      else
15:        for each neighbor Y of X:
16:          if t(Y) = NEW or
17:             (b(Y) = X and h(Y) ? h(X) + c(X,Y) then
18:            b(Y) = X; Insert(Y,h(X) + c(X,Y))
19:          else
20:            if b(Y) ? X and h(X) > h(X) + c(X,Y) then
21:              Insert(X,h(X))
22:            else
23:              if b(Y) ? X and h(X) > h(Y) + c(Y,X) and
24:                 t(Y) = CLOSED and h(Y) > kold then
25:                Insert(Y,h(Y))
26: return Get-Kmin()

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()

Function: Insert(X, h_max)
1: if t(X) = NEW
   then k(x) = h_max
2: else if t(X) = OPEN
   then k(X) = min(k(X), h_max)
3: else
   k(X) = min(h(X), h_max)
4: h(X) = h_max
5: t(X) = OPEN
    
```

Recap: Continuous Optimal Planning

- Generate global path plan from initial map.
 - 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.