Robotic Motion Planning:
A* and D* Search

Robotics Institute 16-735

http://voronoi.sbp.ri.cmu.edu/~motion

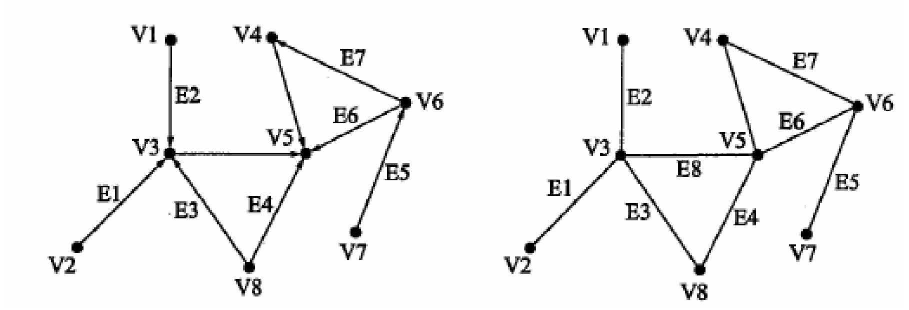
Howie Choset

http://voronoi.sbp.ri.cmu.edu/~choset

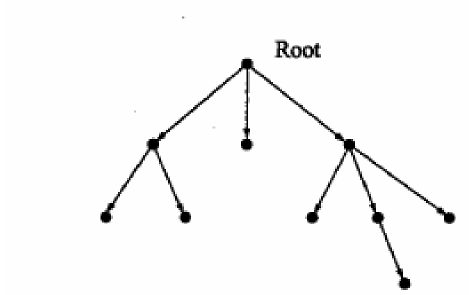
Outline

- Overview of Search Techniques
- A* Search
- D* Search

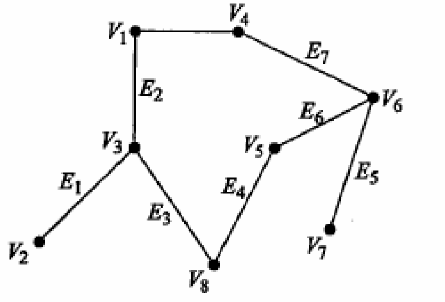
Graphs

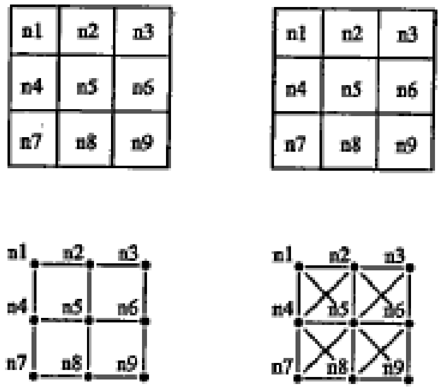


Collection of Edges and Nodes (Vertices)



A tree





Search in Path Planning

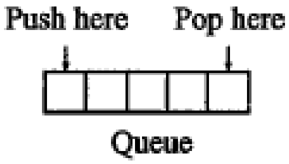
- Find a path between two locations in an unknown, partially known, or known environment
- Search Performance
  - Completeness
  - Optimality → Operating cost
  - Space Complexity
  - Time Complexity

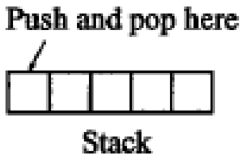
Search

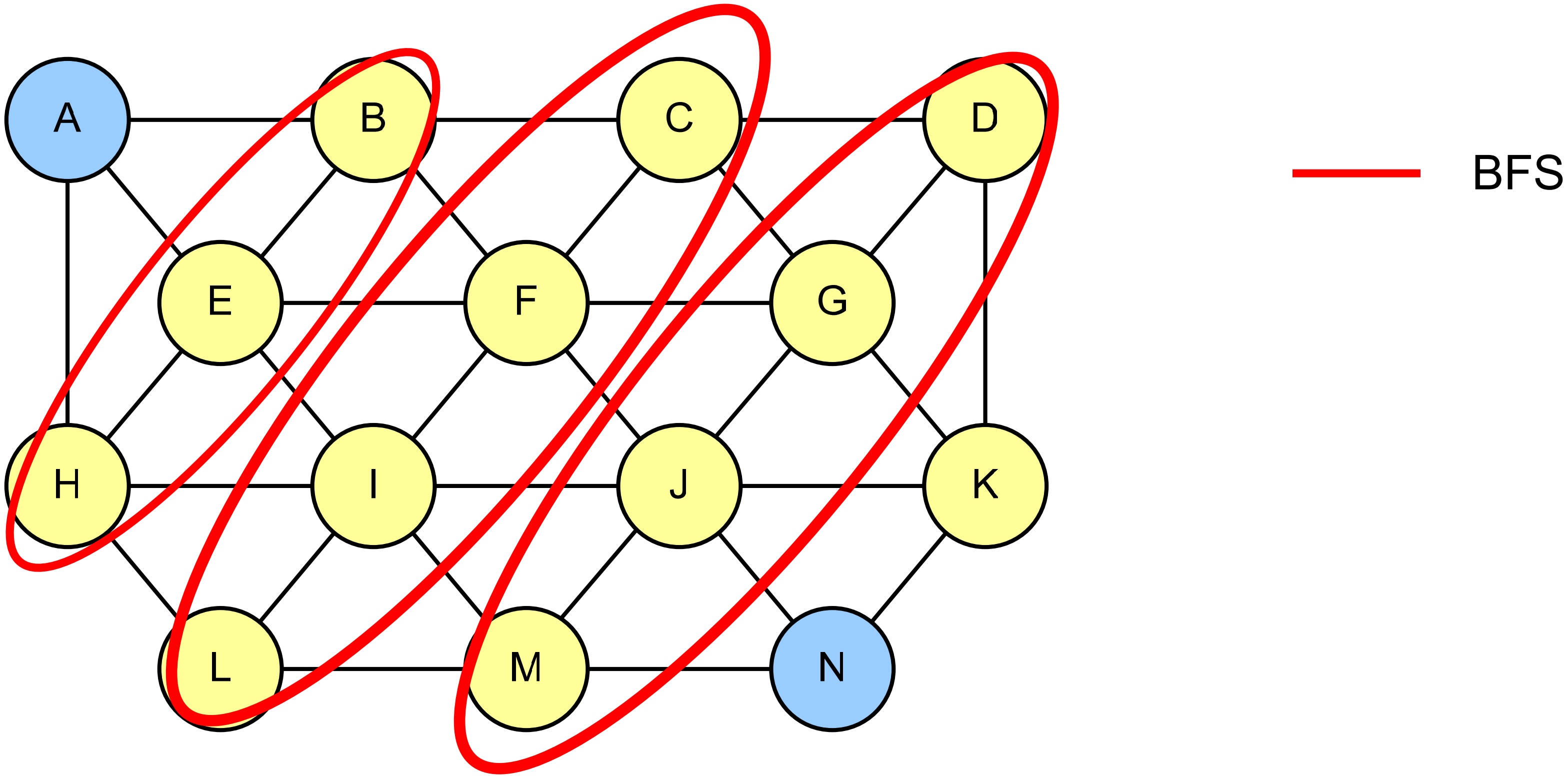
- Uninformed Search
  - Use no information obtained from the environment
  - Blind Search: BFS (Wavefront), DFS
- Informed Search
  - Use evaluation function
  - More efficient
  - Heuristic Search: A*, D*, etc.

Uninformed Search

Graph Search from A to N







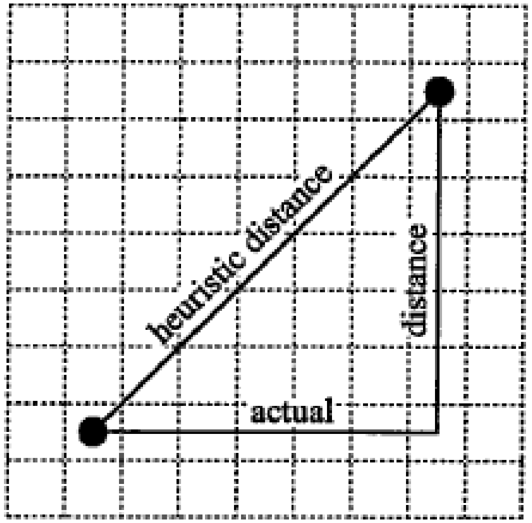
Informed Search: A*

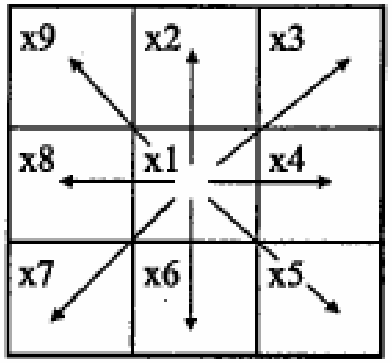
Notation

- ***n*** → node/state
- ***c(n$_1$,n$_2$)*** → the length of an edge connecting between n_1 and n_2
- ***b(n$_1$) = n$_2$*** → backpointer of a node n_1 to a node n_2.

Informed Search: A*

- Evaluation function, ***f(n) = g(n) + h(n)***
- Operating cost function, ***g(n)***
  - Actual operating cost having been already traversed
- Heuristic function, ***h(n)***
  - Information used to find the promising node to traverse
  - Admissible → never overestimate the actual path cost





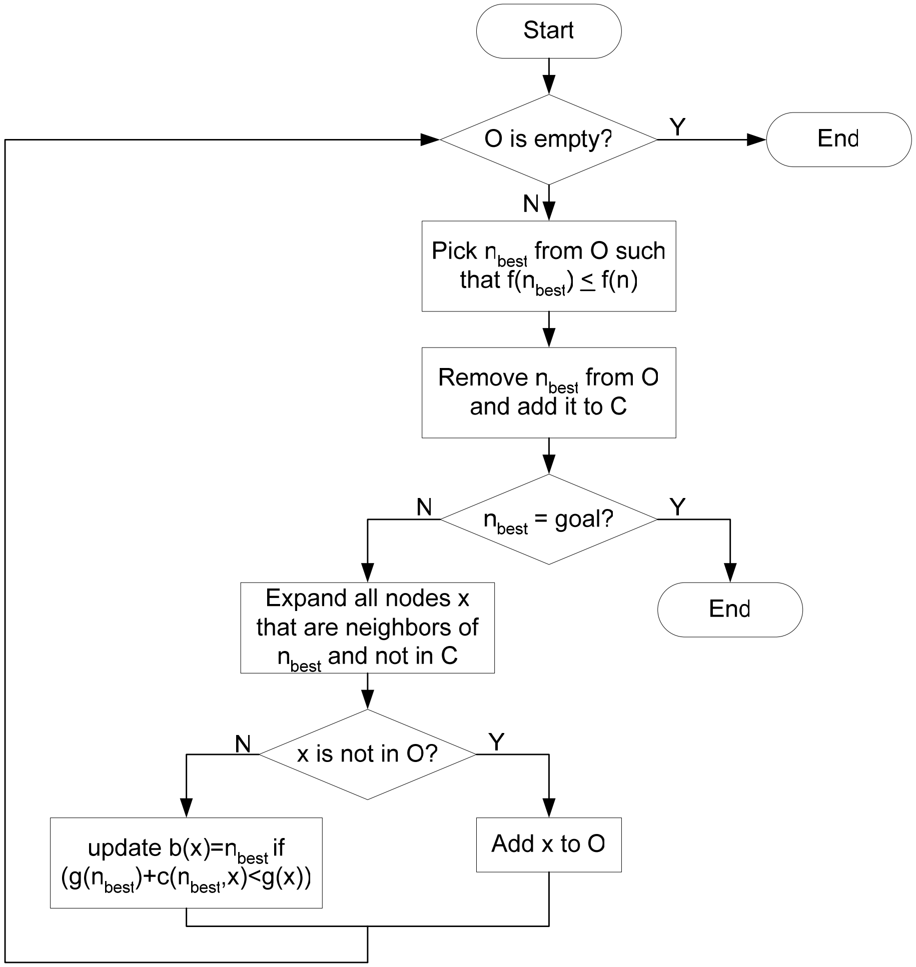
c(x1, x2) = 1
c(x1, x9) = 1.4

c(x1, x8) = 10000, if x8 is in obstacle, x1 is a free cell

c(x1,x9) = 10000.4, if x9 is in obstacle, x1 is a free cell

Cost on a grid

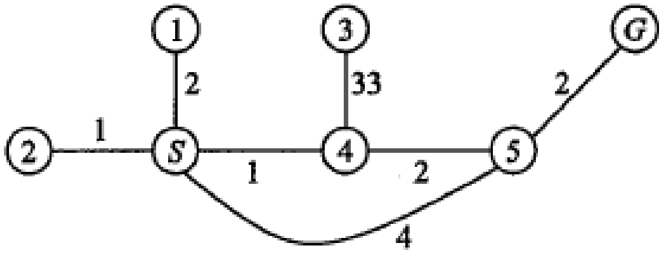
A*: Algorithm


Start
O is empty?
Y
End
N
Pick n_{best} from O such that $f(n_{best}) \leq f(n)$
Remove n_{best} from O and add it to C
n_{best} = goal?
N
Y
End
Expand all nodes x that are neighbors of n_{best} and not in C
x is not in O?
N
Y
update $b(x)=n_{best}$ if $(g(n_{best})+c(n_{best},x)<g(x))$
Add x to O

The search requires 2 lists to store information about nodes

1) **Open list (O)** stores nodes for expansions
2) **Closed list (C)** stores nodes which we have explored

Dijkstra’s Search: f(n) = g(n)



1. O = {S}

2. O = {1, 2, 4, 5}; C = {S} (1,2,4,5 all back point to S)

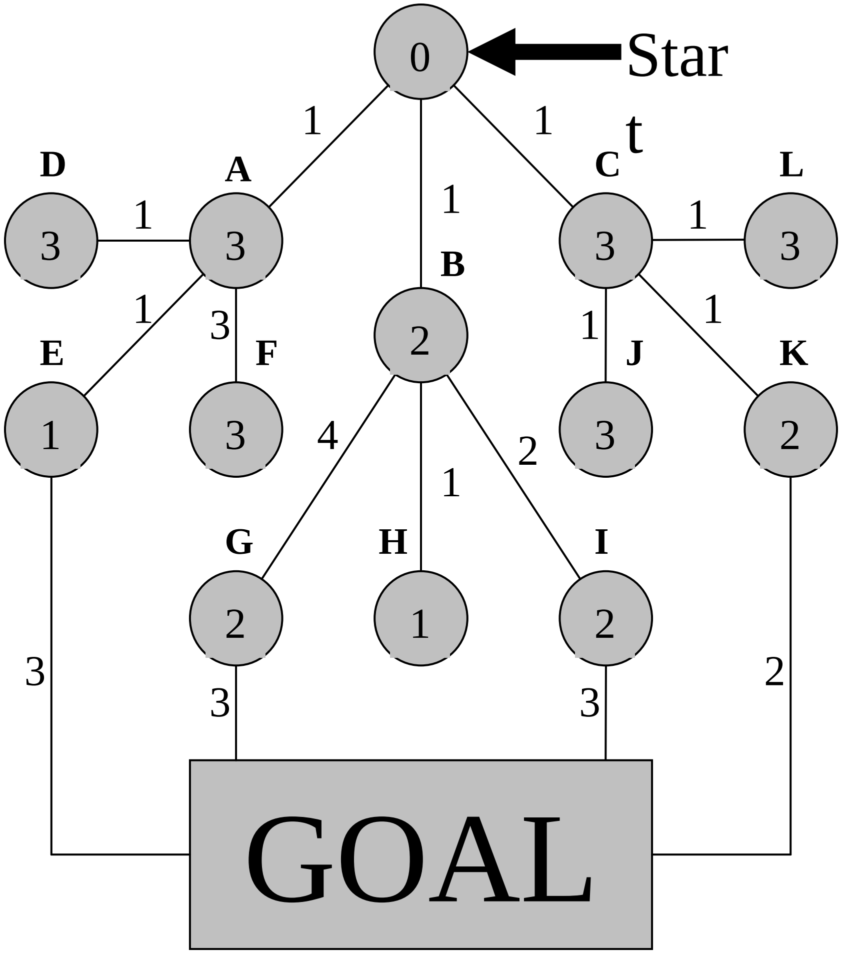
3. O = {1, 4, 5}; C = {S, 2} (there are no adjacent nodes not in C)

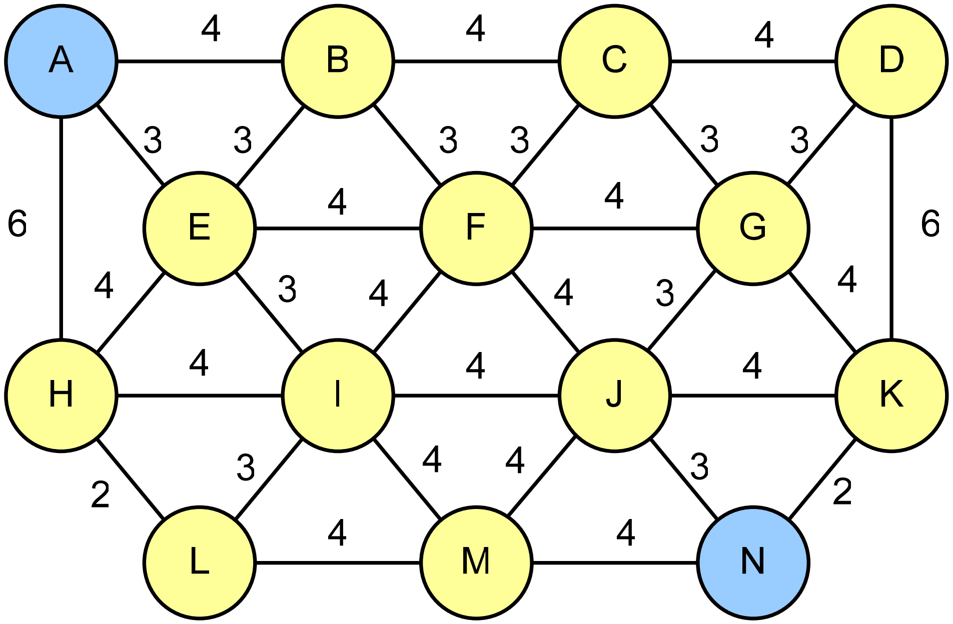
4. O = {1, 5, 3}; C = {S, 2, 4} (1, 2, 4 point to S; 5 points to 4)

5. O = {5, 3}; C = {S, 2, 4, 1}

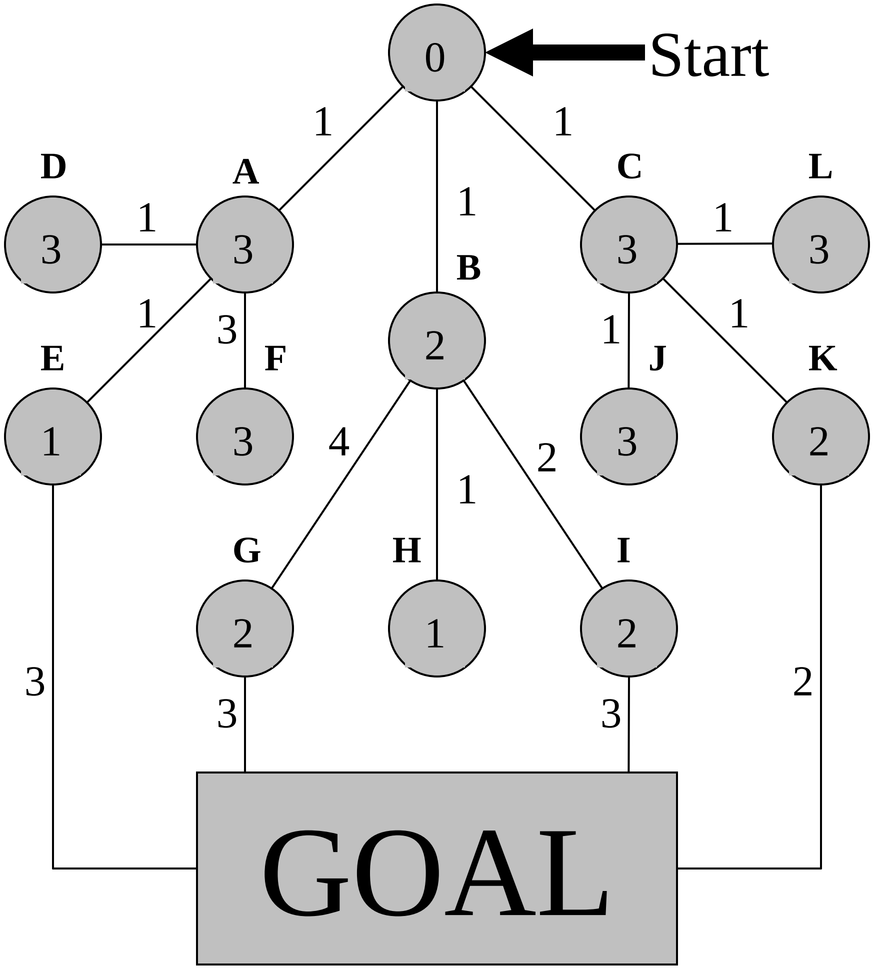
6. O = {3, G}; C = {S, 2, 4 1} (goal points to 5 which points to 4 which points to S)

Two Examples Running A*

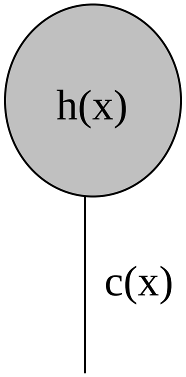




Example (1/5)



Legend



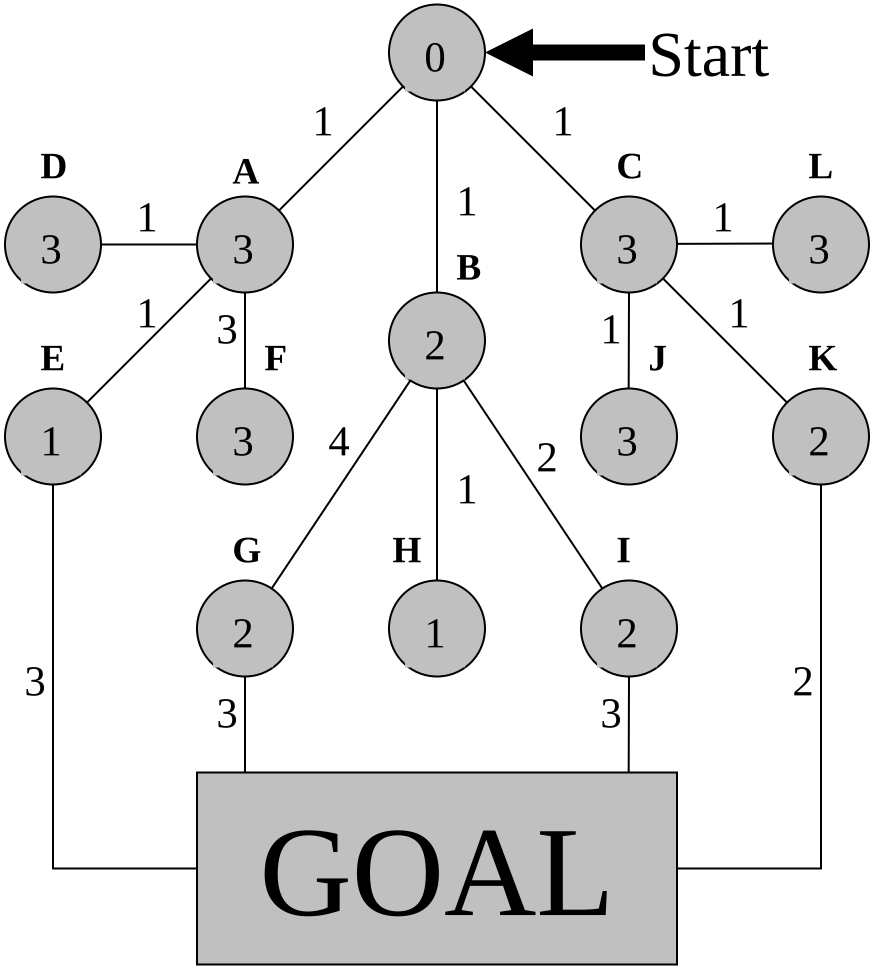
Priority = g(x) + h(x)

Note:

g(x) = sum of all previous arc costs, c(x),
from start to x

Example: c(H) = 2

Example (2/5)



First expand the start node

```
B(3)
A(4)
C(4)
```

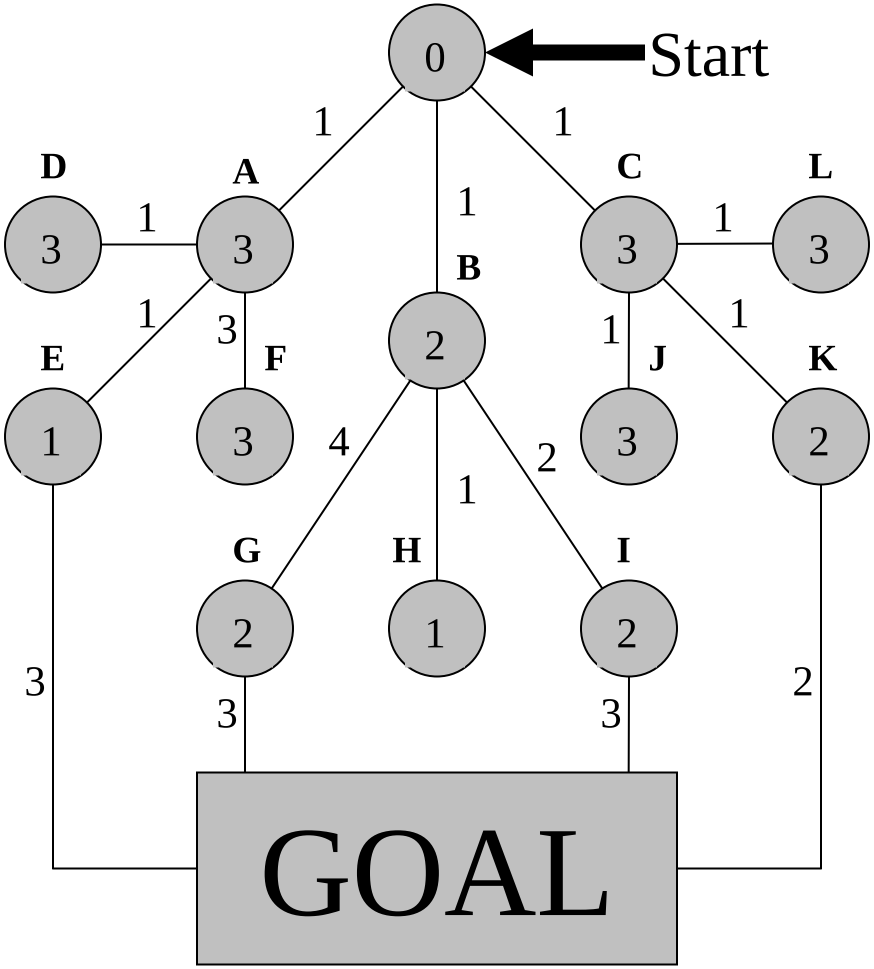
If goal not found, expand the first node in the priority queue (in this case, B)

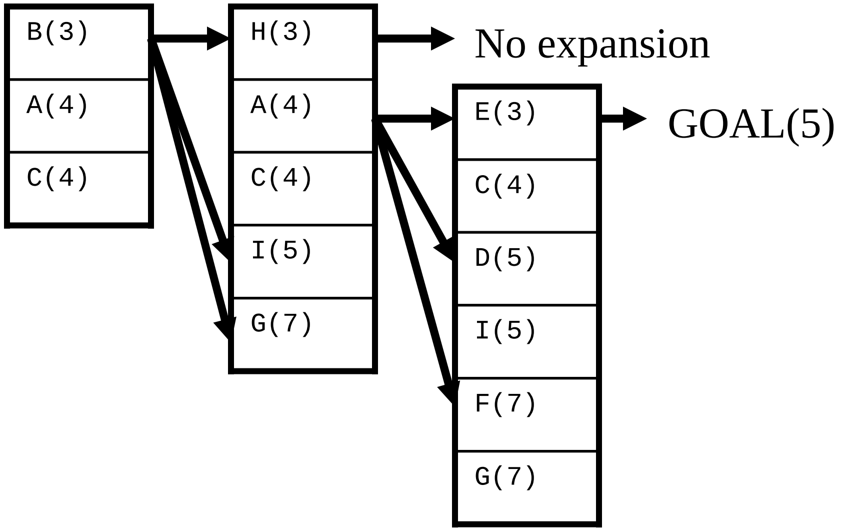
```
H(3)
A(4)
C(4)
I(5)
G(7)
```

Insert the newly expanded nodes into the priority queue and continue until the goal is found, or the priority queue is empty (in which case no path exists)

Note: for each expanded node, you also need a pointer to its respective parent. For example, nodes A, B and C point to Start

Example (3/5)




B(3)
A(4)
C(4)

H(3)
A(4)
C(4)
I(5)
G(7)

No expansion

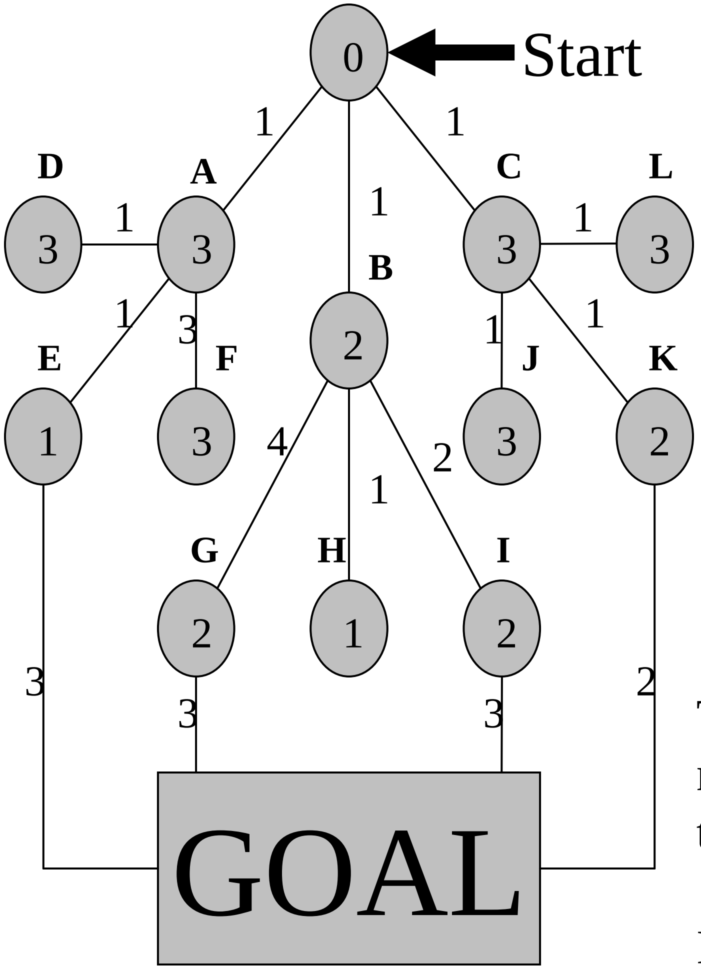
E(3)
C(4)
D(5)
I(5)
F(7)
G(7)

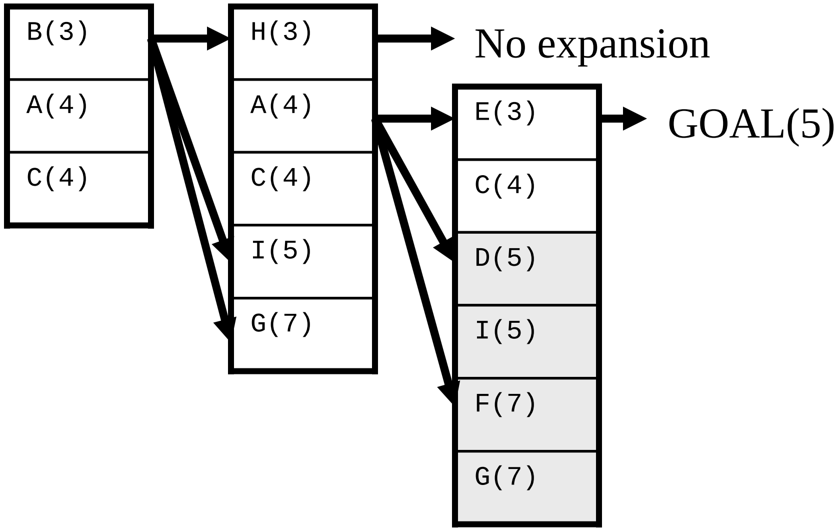
GOAL(5)

We’ve found a path to the goal:
Start => A => E => Goal
(*from the pointers*)

Are we done?

Example (4/5)





B(3)
A(4)
C(4)

H(3)
A(4)
C(4)
I(5)
G(7)

No expansion

E(3)
C(4)
D(5)
I(5)
F(7)
G(7)

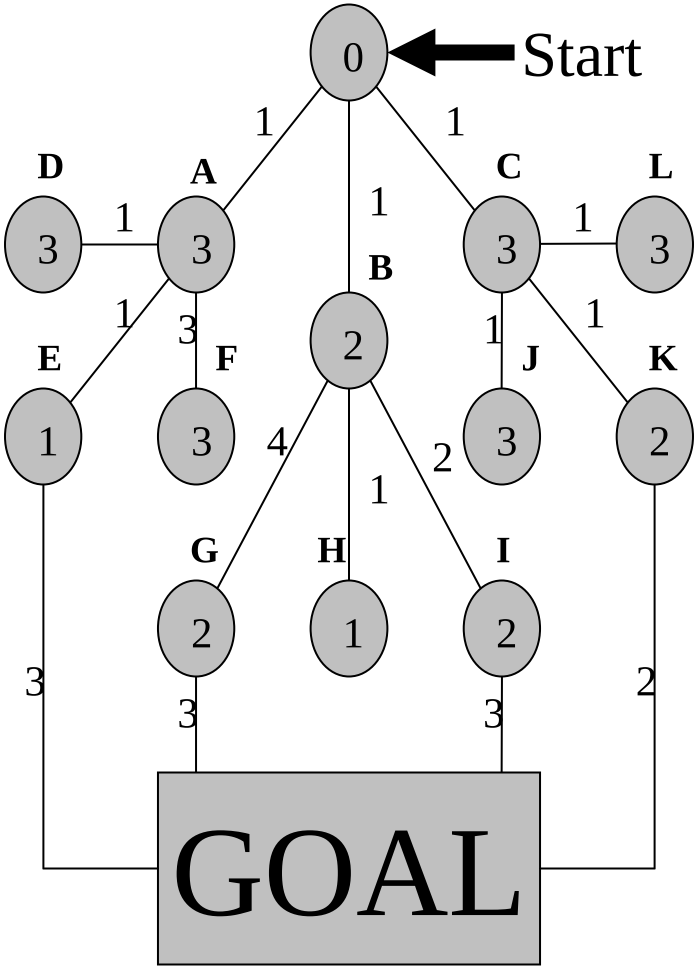
GOAL(5)

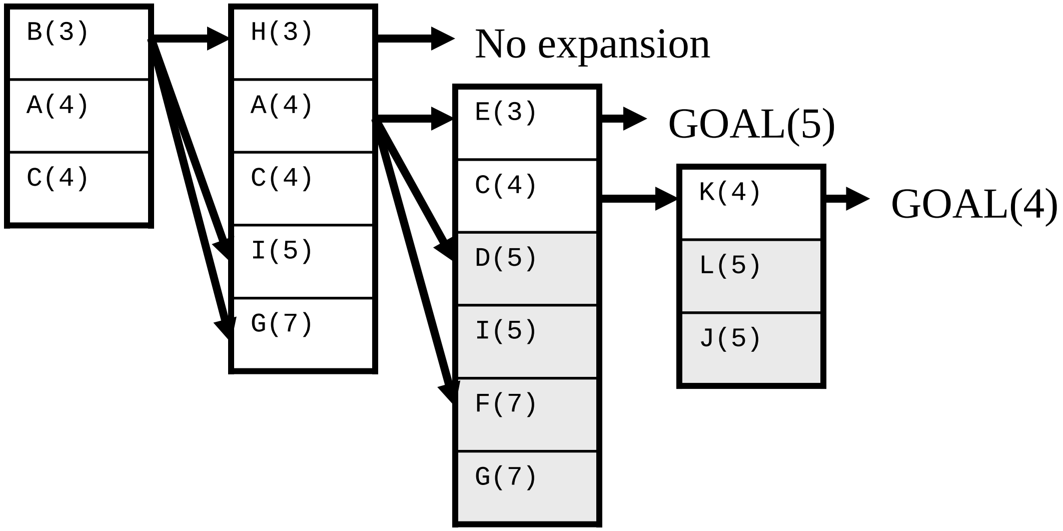
There might be a shorter path, but assuming non-negative arc costs, nodes with a lower priority than the goal cannot yield a better path.

In this example, nodes with a priority greater than or equal to 5 can be pruned.

Why don't we expand nodes with an equivalent priority? (why not expand nodes D and I?)

Example (5/5)




B(3)
A(4)
C(4)

H(3)
A(4)
C(4)
I(5)
G(7)

No expansion

E(3)
C(4)
D(5)
I(5)
F(7)
G(7)

GOAL(5)

K(4)
L(5)
J(5)

GOAL(4)

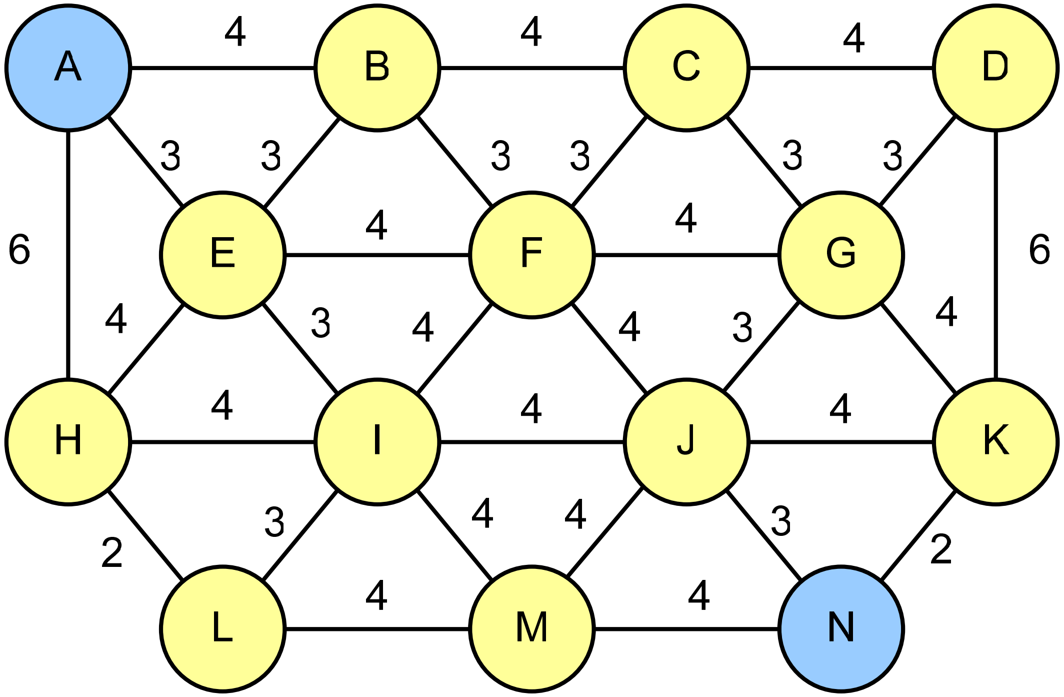
If the priority queue still wasn't empty, we would continue expanding while throwing away nodes with priority lower than 4.
(remember, lower numbers = higher priority)

We can continue to throw away nodes with priority levels lower than the lowest goal found.

As we can see from this example, there was a shorter path through node K. To find the path, simply follow the back pointers.

Therefore the path would be:
Start => C => K => Goal

A*: Example (1/6)

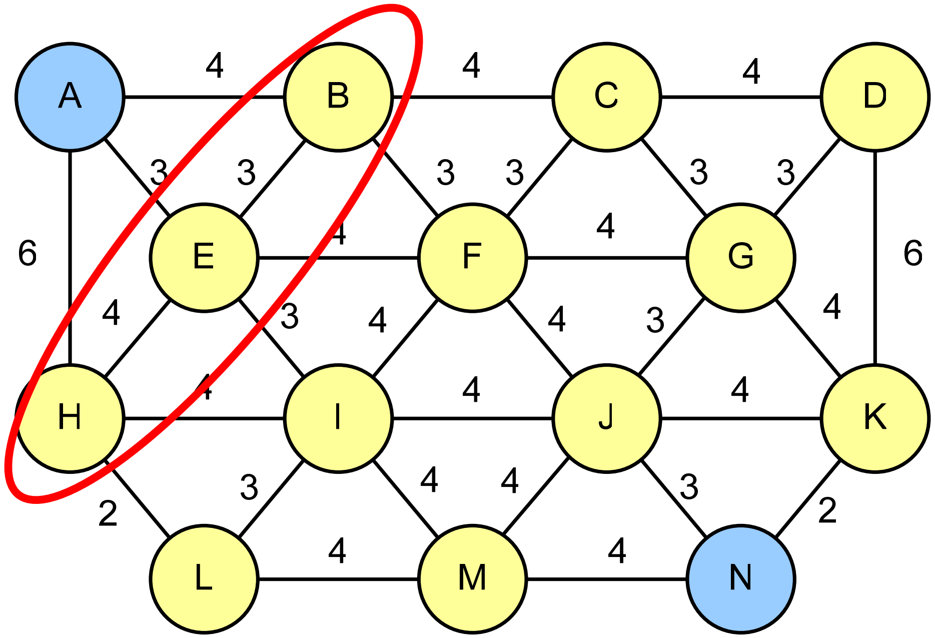


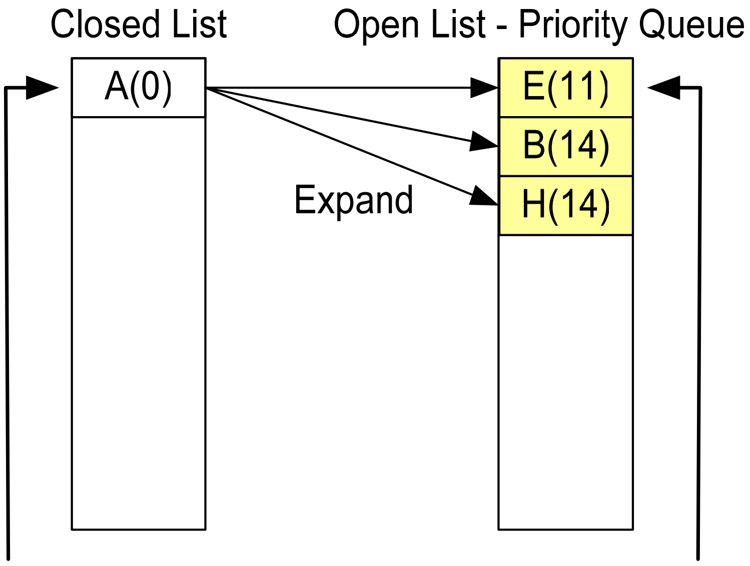
Heuristics

A = 14	H = 8
B = 10	I = 5
C = 8	J = 2
D = 6	K = 2
E = 8	L = 6
F = 7	M = 2
G = 6	N = 0

Legend ○ operating cost

A*: Example (2/6)





Closed List

A(0)

Expand

Open List - Priority Queue

E(11)

B(14)

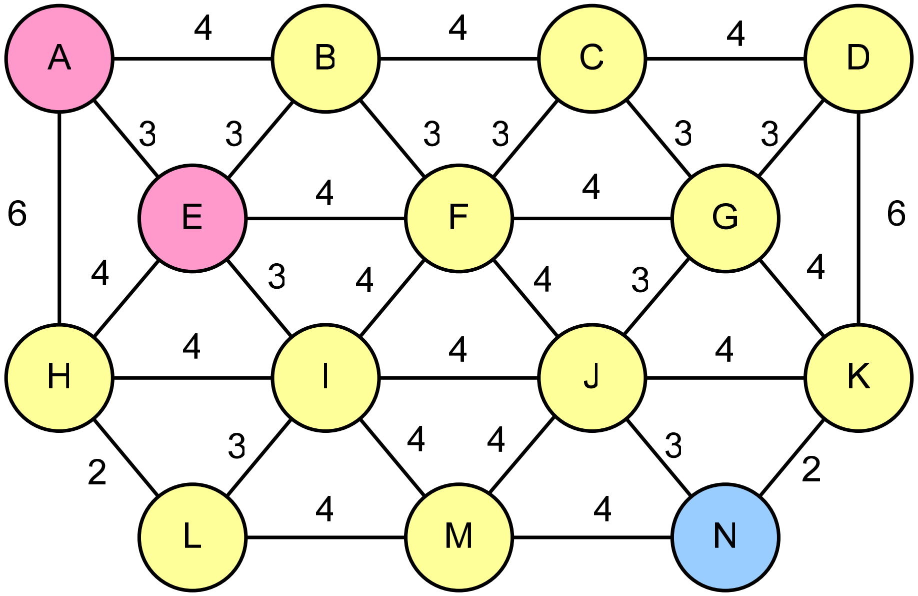
H(14)

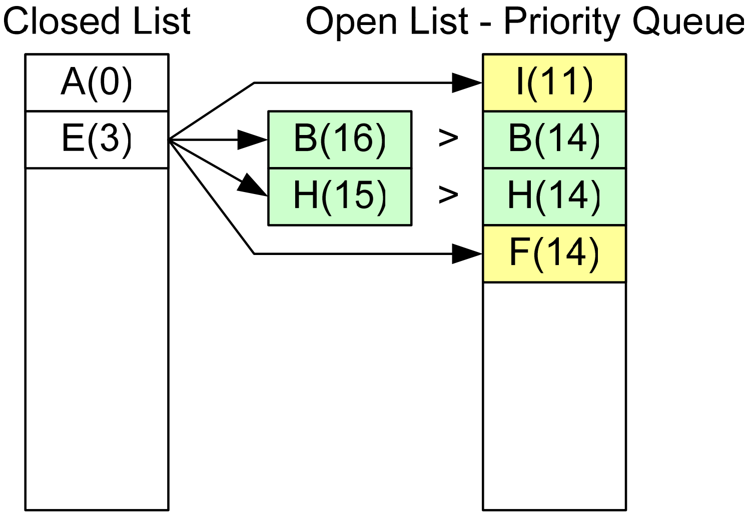
A(x) = A(g(n))

E(y) = E(f(n))
where f(n) = g(n) + h(n)
= 3 + 8 = 11

Heuristics
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (3/6)




Closed List

Open List - Priority Queue

A(0)

E(3)

B(16) > B(14)

H(15) > H(14)

I(11)

F(14)

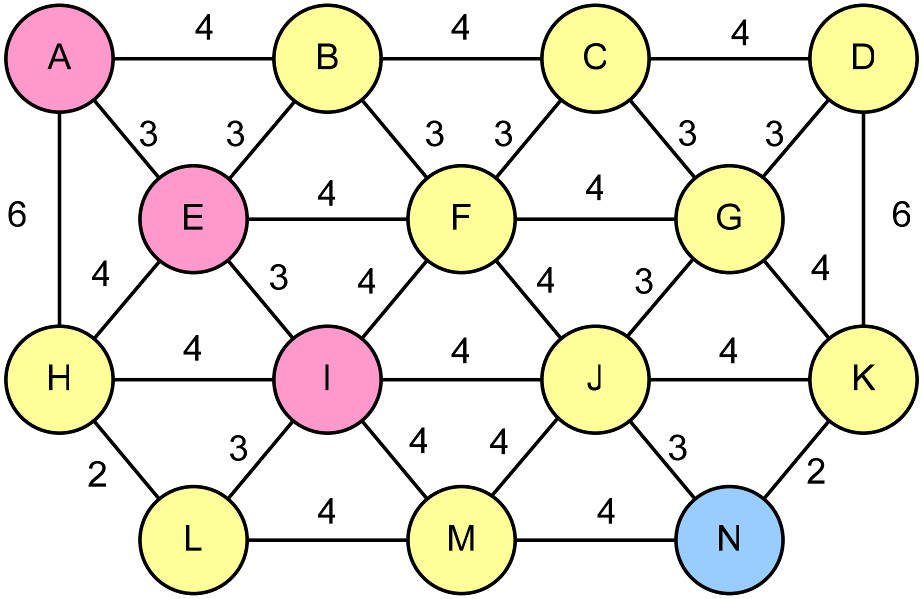
🟩 Update   🟨 Add new node

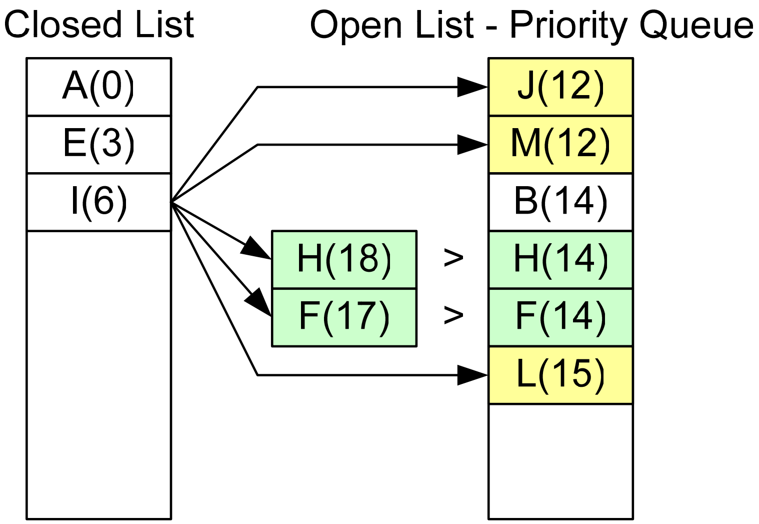
Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8,   I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

Since A → B is smaller than A → E → B, the f-cost value of B in an open list needs not be updated

A*: Example (4/6)




Closed List

A(0)
E(3)
I(6)

H(18) > H(14)
F(17) > F(14)

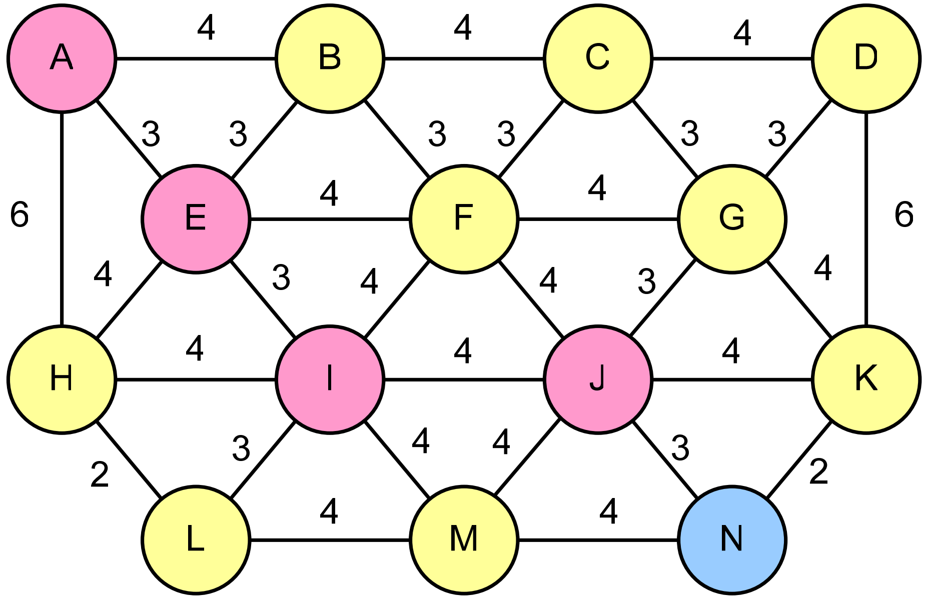
Open List - Priority Queue

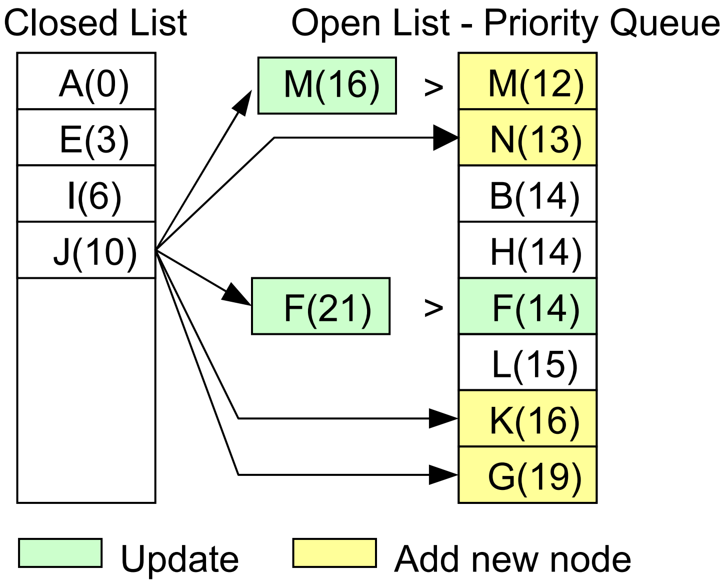
J(12)
M(12)
B(14)
H(14)
F(14)
L(15)

🟩 Update 🟨 Add new node

Heuristics
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (5/6)





Closed List

A(0)
E(3)
I(6)
J(10)

Open List - Priority Queue

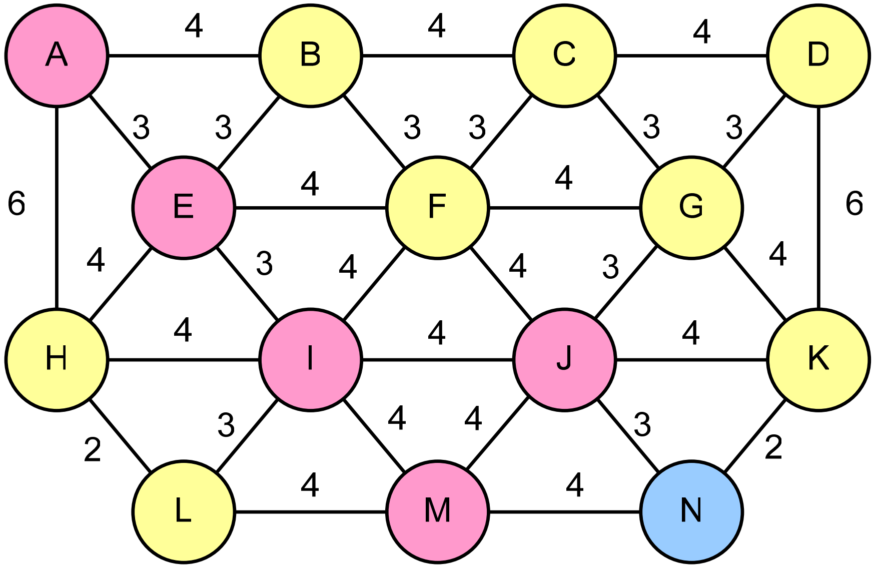
M(16) > M(12)
N(13)
B(14)
H(14)
F(21) > F(14)
L(15)
K(16)
G(19)

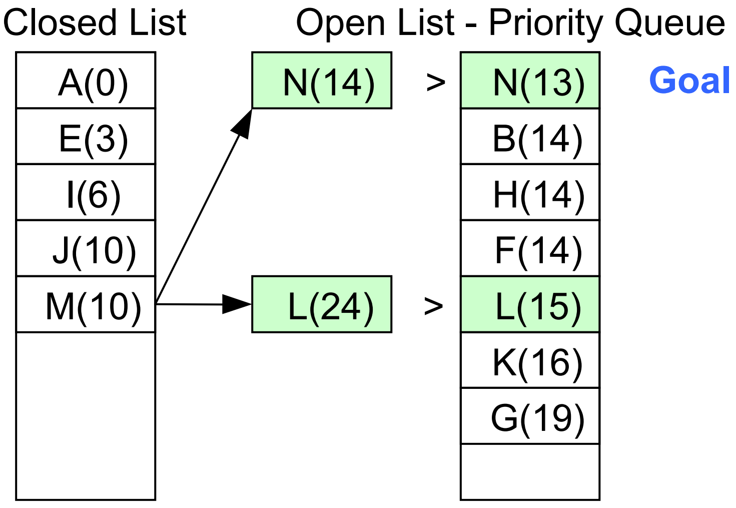
🟩 Update 🟨 Add new node

Heuristics

A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

A*: Example (6/6)





Closed List

A(0)
E(3)
I(6)
J(10)
M(10)

Open List - Priority Queue

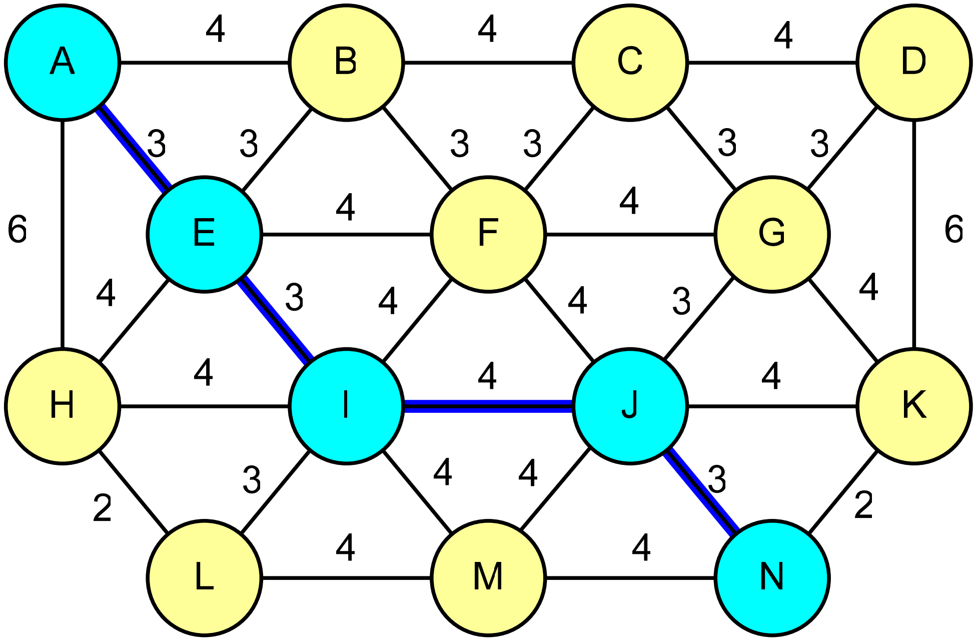
N(14) > N(13) **Goal**
B(14)
H(14)
F(14)
L(24) > L(15)
K(16)
G(19)

🟩 Update 🟨 Add new node

Heuristics
A = 14, B = 10, C = 8, D = 6, E = 8, F = 7, G = 6
H = 8, I = 5, J = 2, K = 2, L = 6, M = 2, N = 0

Since the path to N from M is greater than that from J, **the optimal path to N is the one traversed from J**

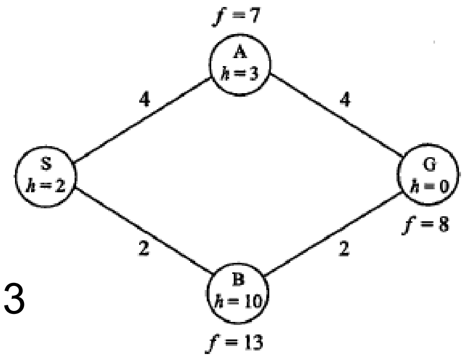
A*: Example Result



Generate the path from the goal node back to the start node through the back-pointer attribute

Non-opportunistic

1. Put S on priority Q and expand it
2. Expand A because its priority value is 7
3. The goal is reached with priority value 8
4. This is less than B’s priority value which is 13

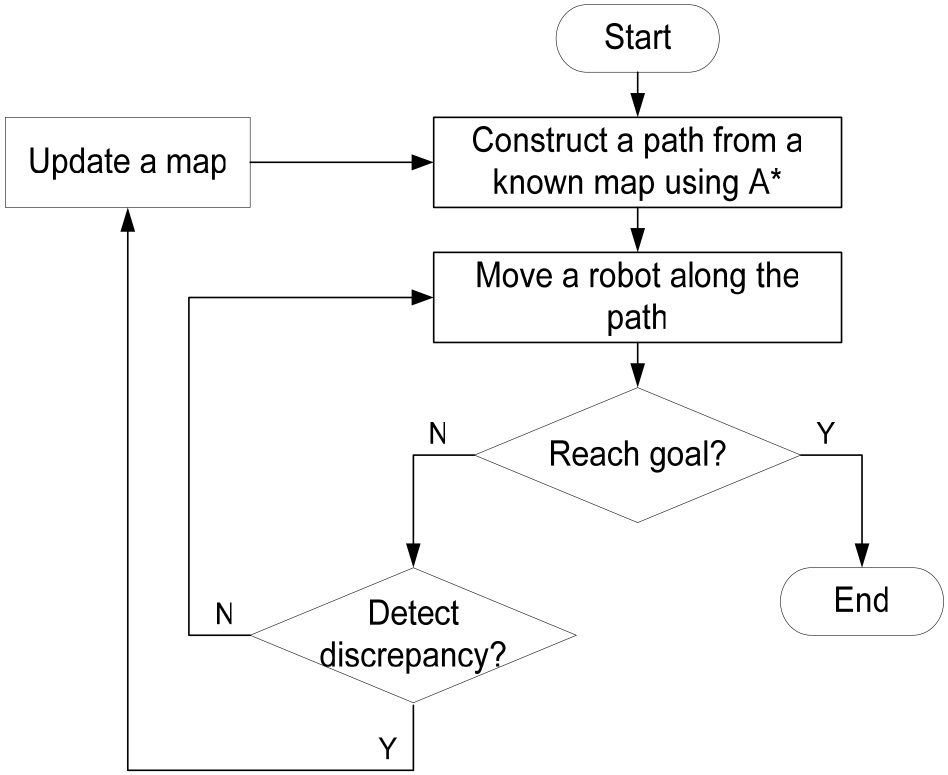


A*: Performance Analysis

- Complete provided the finite boundary condition and that every path cost is greater than some positive constant δ
- Optimal in terms of the path cost
- Memory inefficient → IDA*
- Exponential growth of search space with respect to the length of solution

How can we use it in a partially known, dynamic environment?

A* Replanner – unknown map


Start
Construct a path from a known map using A*
Update a map
Move a robot along the path
Reach goal?
N
Y
End
Detect discrepancy?
N
Y

- Optimal
- Inefficient and impractical in expansive environments – the goal is far away from the start and little map information exists (Stentz 1994)

How can we do better in a partially known and dynamic environment?

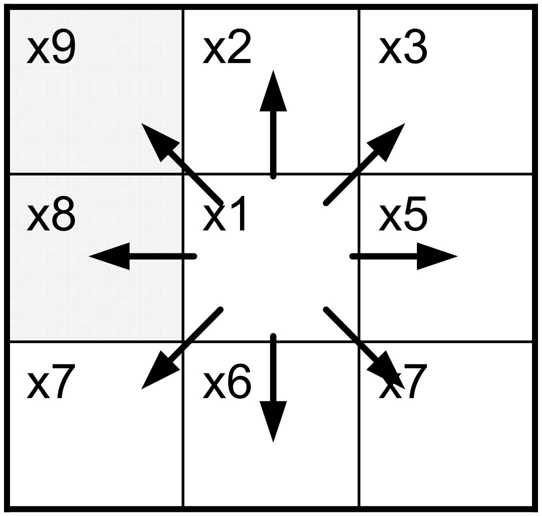
D* Search (Stentz 1994)

- Stands for “Dynamic A* Search”
- Dynamic: Arc cost parameters can change during the problem solving process—replanning online
- Functionally equivalent to the A* replanner
- Initially plans using the Dijkstra’s algorithm and allows intelligently caching intermediate data for speedy replanning

- Benefits
  - Optimal
  - Complete
  - More efficient than A* replanner in expansive and complex environments
    - Local changes in the world do not impact on the path much
    - Most costs to goal remain the same
    - It avoids high computational costs of backtracking

D* Example (1/22)

- *X, Y* → states of a robot
- *b(X) = Y* → backpointer of a state X to a next state Y
- *c(X,Y)* → arc cost of a path from X to Y
- *t(X)* → tag (i.e. NEW,OPEN, and CLOSED) of a state X
- *h(X)* → path cost
- *k(X)* → estimate of shortest path cost



- The robot moves in 8 directions
- The arc cost values, c(X,Y) are small for clear cells and are prohibitively large for obstacle cells

Horizontal/Vertical Traversal	**Diagonal Traversal**
Free cell (e.g. c(X1,X2)) = 1	Free cell (e.g. c(X1,X3)) = 1.4
Obstacle cell (e.g. c(X1,X8)) = 10000	Obstacle cell (e.g. c(X1,X9)) = 10000

D* Algorithm

```
h(G)=0;
do
{
            $k_{min}$=PROCESS-STATE();
 }while($k_{min}$ != -1 && start state not removed from Qu);
if($k_{min}$ == -1)
  { goal unreachable; exit;}
else{
     do{

            do{
                        trace optimal path();
              }while ( goal is not reached && map == environment);

            if ( goal_is_reached)
            { exit;}
            else
            {
               Y= State of discrepancy reached trying to move from some State X;
               MODIFY-COST(Y,X,newc(Y,X));
               do
               {
                  $k_{min}$=PROCESS-STATE();
               }while( k(Y) < h(Y) && $k_{min}$ != -1);
               if($k_{min}$==-1)
                 exit();
            }
       }while(1);
}
```

D* Algorithm

- PROCESS-STATE()
  - Compute optimal path to the goal
  - Initially set h(G) = 0 and insert it into the OPEN list
  - Repeatedly called until the robot’s state X is removed from the OPEN list
- MODIFY-COST()
  - Immediately called, once the robot detects an error in the arc cost function (i.e. discover a new obstacle)
  - Change the arc cost function and enter affected states on the OPEN list

MODIFY-COST(X,Y,cval)

```
c(X,Y)=cval
if t(X) =CLOSED then INSERT (X,h(X))
Return GET-MIN ( )
```

PROCESS-STATE()

```
X = MIN-STATE( )
if   X= NULL then return –1
$k_{old}$ = GET-KMIN( );  DELETE(X);
if $k_{old}$< h(X) then
    for each neighbor Y of X:
            if h(Y) <= $k_{old}$ and h(X) > h(Y) + c(Y,X) then
              b(X) = Y;  h(X) = h(Y)+c(Y,X);
if $k_{old=}$ h(X) then
    for each neighbor Y of X:
            if t(Y) = NEW or
            (b(Y) =X and h(Y) $\neq$ h(X)+c (X,Y) ) or
            (b(Y) $\neq$ X and h(Y) > h(X)+c (X,Y) ) then
            b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
else
    for each neighbor Y of X:
            if t(Y) = NEW or
            (b(Y) =X and h(Y) $\neq$ h(X)+c (X,Y) ) then
                b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
            else
                if  b(Y) $\neq$ X and h(Y) > h(X)+c (X,Y) then
                  INSERT(X, h(X))
                else
                  if  b(Y) $\neq$ X and h(X) > h(Y)+c (X,Y) and
                  t(Y) = CLOSED and h(Y) > $k_{old}$ then
                  INSERT(Y, h(Y))
Return GET-KMIN ( )
```

Other Procedures

MIN-STATE()

Return X if k(X) is minimum for all states on open list

GET-KMIN()

Return the minimum value of k for all states on open list

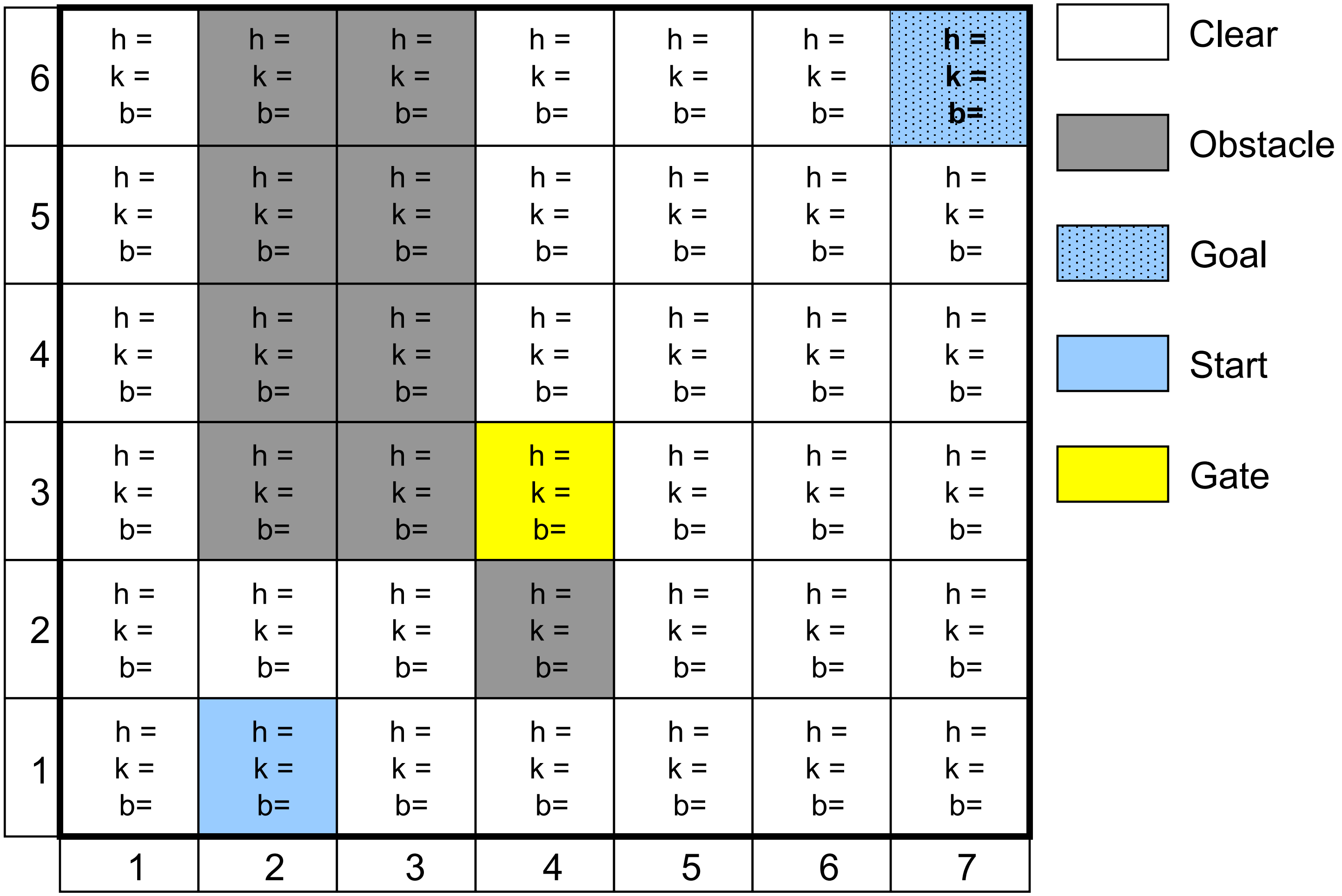
INSERT(X,h_{new})

```
if t(X) = NEW then k(X)=$h_{new}$
if t(X) = OPEN then k(X)=min(k(X),$h_{new}$)
if t(X) = CLOSED then k(X)=min(k(X),$h_{new}$) and t(X)= OPEN
Sort open list based on increasing k values;
```

D* Example (2/22)

Initially, all states have the tag NEW

All h and k values will be measured as “distance” in grid to goal



	1	2	3	4	5	6	7
6	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
5	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
4	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
3	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=

□ Clear

▨ Obstacle

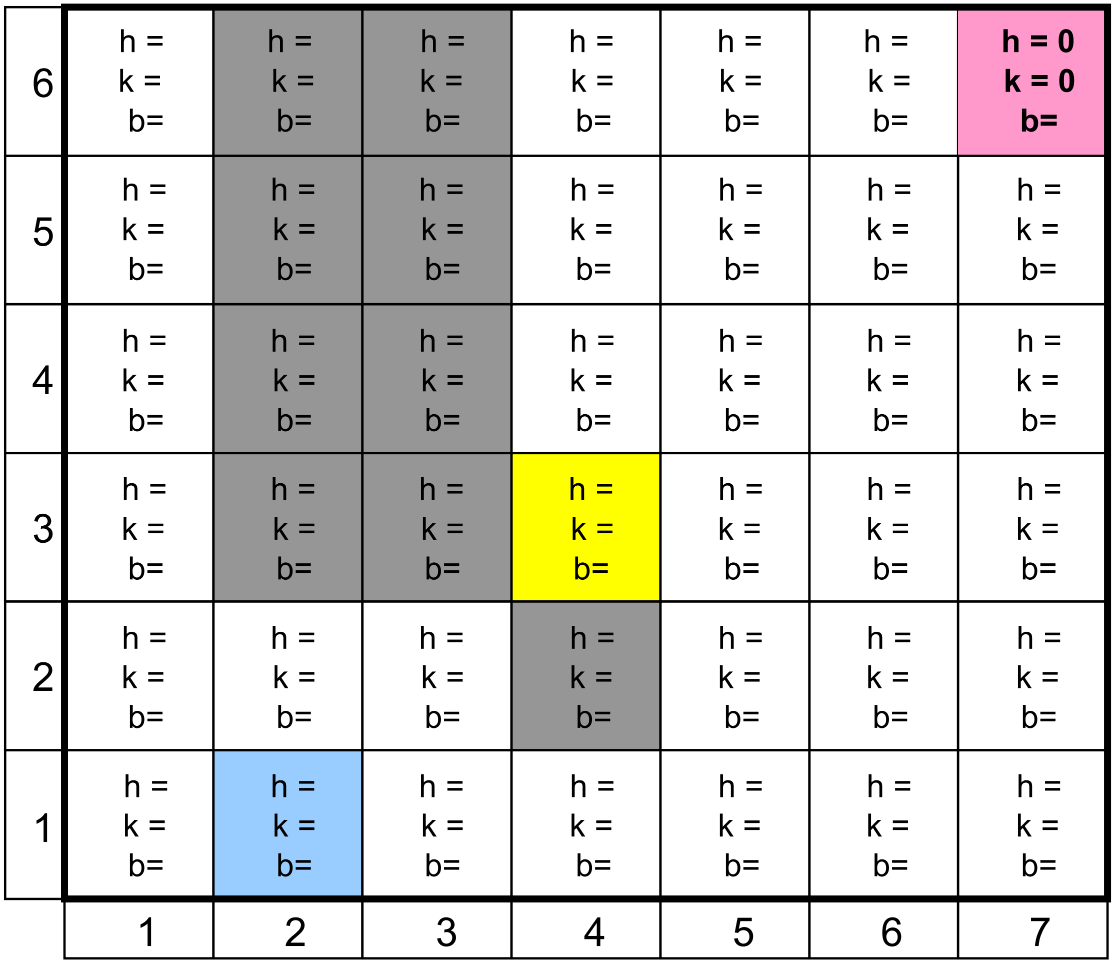
░ Goal

🟦 Start

🟨 Gate

D* Example (3/22)

Put goal node onto the queue, also called *the open list*, with h=0 and k=0. The k value is used as the priority in the queue. So, initially this looks like an Dijkstra's Search



6	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
4	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
3	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
	1	2	3	4	5	6	7

State	k
(7,6)	0

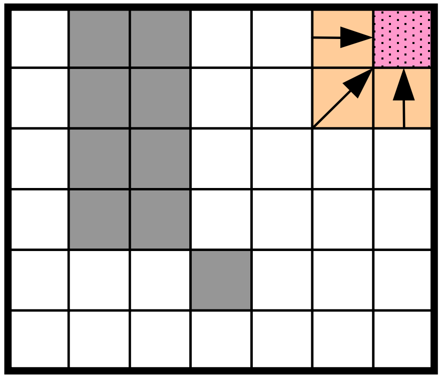
D* Example (4/22)

```
if kold= h(X) then  for each neighbor Y of X:
    if t(Y) = NEW or
    (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
   (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
      then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

Pop the goal node off the open list and expand it so its neighbors (6,6), (6,5) and (7,5) are placed on the open list. Since these states are new, their k and h values are the same, which is set to the increment in distance from the previous pixel because they are free space. Here, k and h happen to be distance to goal.

6	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 1 k = 1 b= (7,6)	~~h = 0~~ ~~k = 0~~ ~~b=~~
5	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
3	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
	1	2	3	4	5	6	7

State	k
(6,6)	1
(7,5)	1
(6,5)	1.4



🟧 “Open” – on priority queue

🟪 “Closed” & currently being expanded

☒ “Closed”

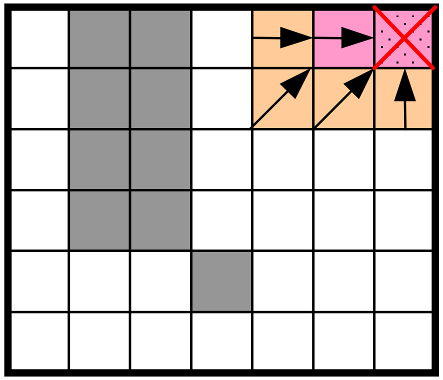
D* Example (5/22)

```
if kold= h(X) then  for each neighbor Y of X:
    if t(Y) = NEW or
    (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
   (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
      then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

Expand (6,6) node so (5,6) and (5,5) are placed on the open list

6	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	h = 0 k = 0 b=
5	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
3	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
	1	2	3	4	5	6	7

State	k
(7,5)	1
(6,5)	1.4
(5,6)	2
(5,5)	2.4



🟧 “Open” – on priority queue

🟪 “Closed” & currently being expanded

☒ “Closed”

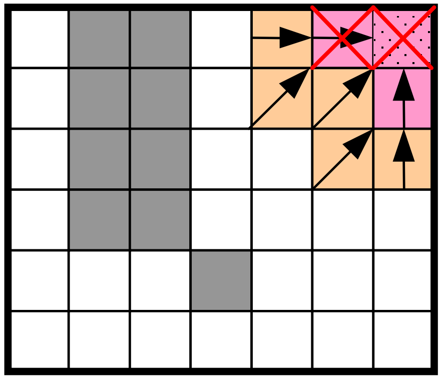
D* Example (6/22)

```
if kold= h(X) then  for each neighbor Y of X:
   if t(Y) = NEW or
   (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
  (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
      then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

Expand (7,5) so (6,4) and (7,4) are placed on the open list

6	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	h = 0 k = 0 b=
5	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
	1	2	3	4	5	6	7

State	k
(6,5)	1.4
(5,6)	2
(7,4)	2
(6,4)	2.4
(5,5)	2.4



🟧 "Open" – on priority queue

▨ "Closed" & currently being expanded

☒ "Closed"

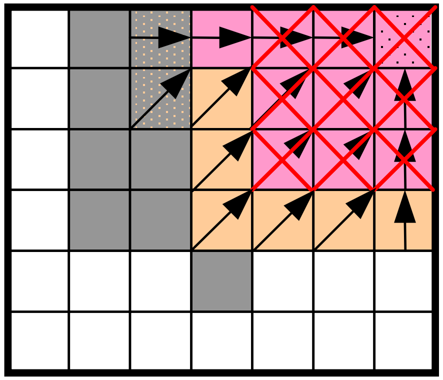
D* Example (7/22)

```
if kold= h(X) then  for each neighbor Y of X:
    if t(Y) = NEW or
    (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
   (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
        then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

When (4,6) is expanded, (3,5) and (3,6) are put on the open list but since the states (3,5) and (3,6) were obstacles their h value is high and since they are new, when they are inserted onto the open list, their k and h values are the same.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	h = 0 k = 0 b=
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h = k = b=	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = k = b=	h = k = b=	h = k = b=	h = 4.2 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
1	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=	h = k = b=
	1	2	3	4	5	6	7

State	k
(7,3)	3
(6,3)	3.4
(4,5)	3.4
(5,3)	3.8
(4,4)	3.8
(4,3)	4.2
(3,6)	10000
(3,5)	10000



🟧 “Open” – on priority queue

🟪 “Closed” & currently being expanded

☒ “Closed”

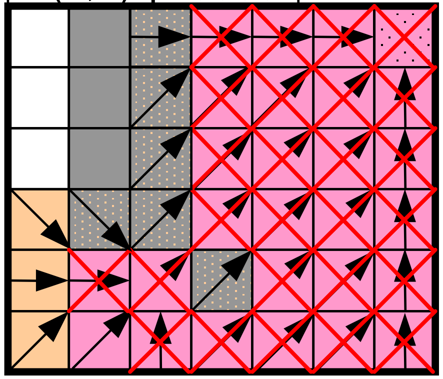
D* Example (8/22)

```
if kold= h(X) then  for each neighbor Y of X:
    if t(Y) = NEW or
    (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
   (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
      then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

The search ends when the start node is *expanded*. Note that there are still some remaining nodes on the open list and there are some nodes which have not been touched at all.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h = 4.2 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	h = 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k = 4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	h = 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000
(2,3)	10000



■ “Open” – on priority queue

■ “Closed” & currently being expanded

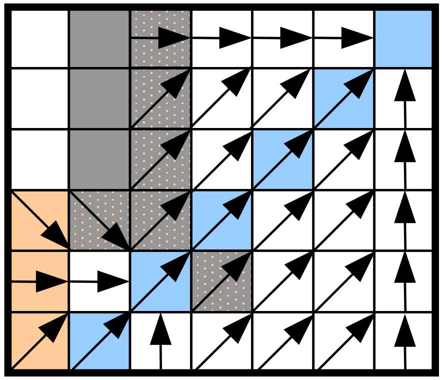
☒ “Closed”

D* Example (9/22)

Determine optimal path by following gradient of h values

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h = 4.2 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	h = 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	h = 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000
(2,3)	10000



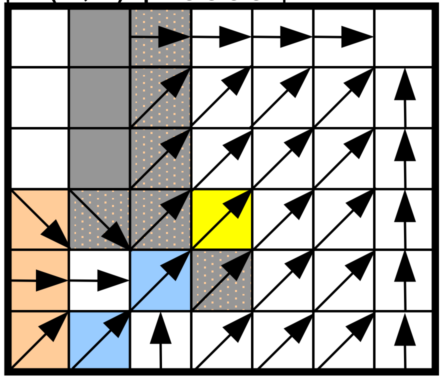
D* Example (10/22)

The robot starts moving along the optimal path, but discovers that pixel (4,3) is an obstacle!!

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	[illegible]2 [illegible] [illegible]4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	= 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000
(2,3)	10000

Robot

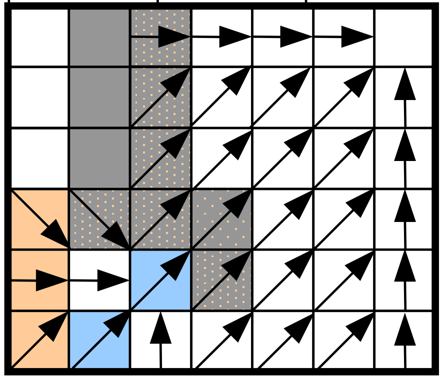


D* Example (11a/22)

Increase by a large number the transition cost to (4,3) for all nodes adjacent to (4,3). Next, put all nodes affected by the increased transition costs (all nine neighbors) on the open list including (4,3). Note that some neighbors of (4,3), and (4,3) itself have lower k values than most elements on the open list already. Therefore, these nodes will be popped first.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=4.2 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	= 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(5,4)	2.8
(5,3)	3.8
(4,4)	3.8
(4,3)	4.2
(5,2)	4.8
(3,2)	5.6
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0



Modify-Cost (X, Y, cval)

```
    c(X,Y)=cval
    if t(X) =CLOSED then INSERT (X,h(X))
```

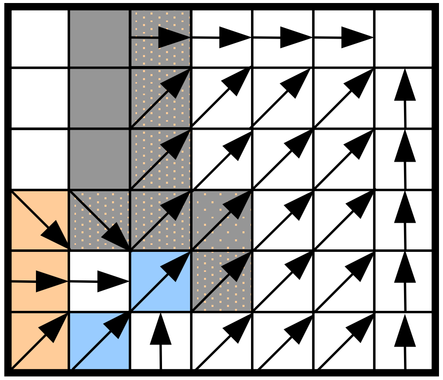
Now, things will start to get interesting!

D* Example (11b/22)

(5,4) is popped first because its k value is the smallest. Since its k and h are the same, consider each neighbor of (5,4). One such neighbor is (4,3). (4, 3)’s back pointer points to (5,4) but its original h value is not the sum of (5,4)’s h value plus the transition cost, which was just raised due to the obstacle. Therefore, (4,3) is put on the open list but with a high h value. Note that since (4,3) is already on the open list, its k value remains the same. Now, the node (4,3) is a called a ***raise*** *state* because h>k.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	= 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(5,3)	3.8
(4,4)	3.8
(4,3)	4.2
(5,2)	4.8
(3,2)	5.6
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000



```
if kold= h(X) then  for each neighbor Y of X:
        if t(Y) = NEW or
        (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
        (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) ) then
        b(Y) = X : INSERT(Y, h(X)+c(X,Y))
```

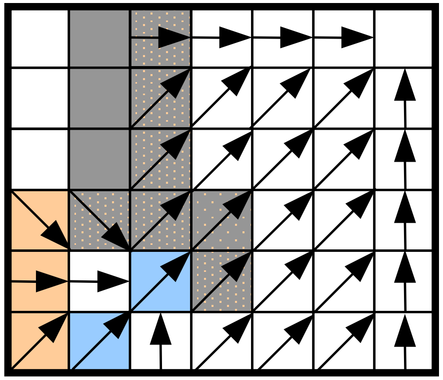
get interesting!

D* Example (11c/22)

Next, we pop (5,3) but this will not affect anything because none of the surrounding pixels are new, and the h values of the surrounding pixels are correct. A similar non-action happens for (4,4).

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	~~h = 2.8~~ ~~k = 2.8~~ ~~b= (6,5)~~	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	= 5.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(4,3)	4.2
(5,2)	4.8
(3,2)	5.6
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000



```
if kold= h(X) then  for each neighbor Y of X:
        if t(Y) = NEW or
        (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
        (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) ) then
        b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

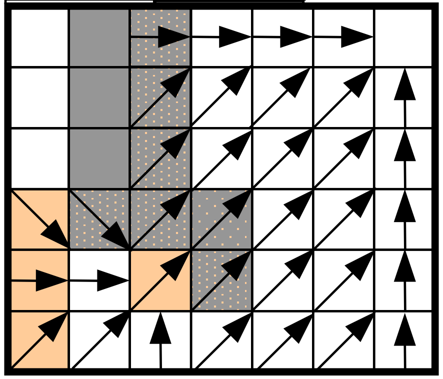
get interesting!

D* Example (12/22)

Pop (4,3) off the queue. Because k<h, our objective is to try to decrease the h value. This is akin to finding a better path from (4,3) to the goal, but this is not possible because (4,3) is an obstacle. For example, (5,3) is a neighbor of (4,3) whose h value is less than (4,3)'s k value, but the h value of (4,3) is "equal" to the h value of (5,3) plus the transition cost, therefore, we cannot improve anything coming from (4,3) to (5,3). This is also true for (5,4) and (4,4). So, we cannot find a path through any of (4,3)'s neighbors to reduce h. Next, we expand (4,3), which places all pixels whose back pointers point to (4,3) [in this case, only (3,2)] on the open list with a high h value. Now, (3,2) is also a **raise** state. Note that the k value of (3,2) is set to the minimum of its old and new h values (this setting happens in the insert function). Next, we pop (5,2) but this will not affect anything because none of the surrounding pixels are new, and the h values of the surrounding pixels are correct

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	~~h = 3.8~~ ~~k = 3.8~~ ~~b= (5,5)~~	~~h = 2.8~~ ~~k = 2.8~~ ~~b= (6,5)~~	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	~~h = 3.8~~ ~~k = 3.8~~ ~~b= (6,4)~~	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h = 6.6 k = 6.6 b=(3,2)	10005.6 k = 5.6 b=(4,3)	h=10000 k=10000 b=(5,3)	~~h = 4.8~~ ~~k = 4.8~~ ~~b= (6,3)~~	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 7.0 k = 7.0 b=(3,2)	h = 6.6 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(3,2)	5.6
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000



```
if kold< h(X) then
   for each neighbor Y of X:
     if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
        b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

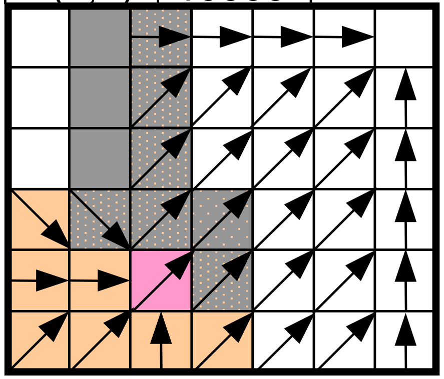
```
else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

D* Example (13/22)

Pop (3,2) off the queue. Since k<h, see if there is a neighbor whose h value is less than the k value of (3,2) – if there is, we'll redirect the backpointer through this neighbor. However, no such neighbor exists. So, look for a neighboring pixel whose back pointer does not point to (3,2), whose h value plus the transition cost is less than the (3,2)'s h value, which is on the closed list, and whose h value is greater than the (3,2)'s k value. The only such neighbor is (4,1). This *could* potentially lead to a lower cost path. So, the neighbor (4,1) is chosen because it could potentially reduce the h value of (3,2). We put this neighbor on the open list with its current h value. It is called a **lower** state because h = k. The pixels whose back pointers point to (3,2) and have an "incorrect" h value, ie. The h value of the neighboring pixel is not equal to the h value of (3,2) plus its transition cost, are also put onto the priority queue with maximum h values (making them **raise** states). These are (3,1), (2,1), and (2,2). Note that the k values of these nodes are set to the minimum of the new h value and the old h value.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h=10000 k = 6.6 b=(3,2)	h = 10000 k = 5.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	h = 10000 k = 7.0 b=(3,2)	h = 10000 k = 6.6 b= (3,2)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(4,1)	6.2
(3,1)	6.6
(2,2)	6.6
(2,1)	7.0
(1,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000



```
if kold< h(X) then
    for each neighbor Y of X:
        if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
            b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

```
for each neighbor Y of X:
    if t(Y) = NEW or b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
    else if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then INSERT(X, h(X))
    else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

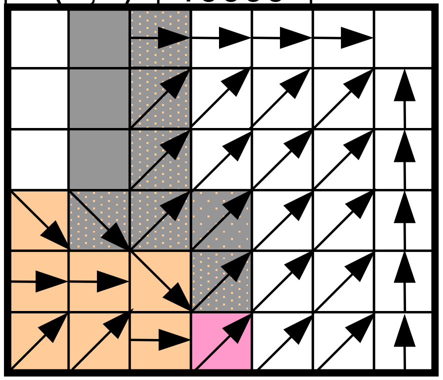
D* Example (14/22)

Pop (4,1) off the open list and expand it. Since (4,1)'s h and k values are the same, look at the neighbors whose pack pointers do not point to (4,1) to see if passing through (4,1) reduces any of the neighbor's h values. This redirects the backpointers of (3,2) and (3,1) to pass through (4,1) and then puts them onto the priority queue.

Because (3,2) was "closed", its new k value is the smaller of its old and new h values and since k==h, it is now a **lower** state, ie, new k = min (old h, new h). Because (3,1) was "open" (on the priority queue), its new k value is the smaller of its old k value and its new h value, ie, new k = min (old k, new h). See

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h=10000 k = 6.6 b=(3,2)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	h = 10000 k = 8.0 b=(3,2)	h = 7.2 k = 6.6 b= (4,1)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(3,1)	6.6
(2,2)	6.6
(2,1)	7.0
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000



```
if kold= h(X) then  for each neighbor Y of X:
    if t(Y) = NEW or
    (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
   (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
        then b(Y) = X : INSERT(Y, h(X)+c(X,Y))
```

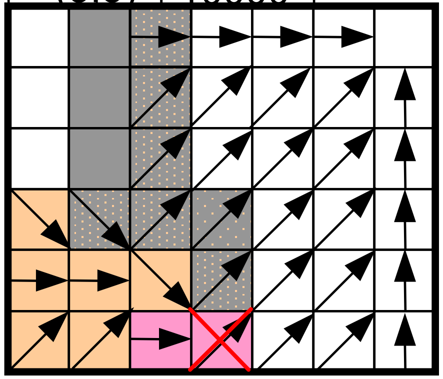
D* Example (15/22)

Pop (3,1) off the open list. Since k (6.6) < h (7.2), ask is there a neighbor whose h value is less than the k value of (3,1). Here, (4,1) is. Now, if the transition cost to (4,1) + the h value of (4,1) is less than the h value of (3,1), then reduce the h value of (3,1). However, this is not the case.

However, (3,1) can be used to form a reduced cost path for its neighbors, so put (3,1) back on the priority queue with k set to the minimum of its old h value and new h value. Thus, it now also becomes a **lower** state.

	1	2	3	4	5	6	7
6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 8.0 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 7.6 k = 7.6 b=(2,2)	h=10000 k = 6.6 b=(3,2)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 8.0 k = 8.0 b=(2,2)	= 10000 k = 7.0 b=(3,2)	h = 7.2 k = 7.2 b= (4,1)	~~h = 6.2~~ ~~k = 6.2~~ ~~b=(5,2)~~	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)

State	k
(2,2)	6.6
(2,1)	7.0
(3,1)	7.2
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000



```
if kold< h(X) then
    for each neighbor Y of X:
        if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
            b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

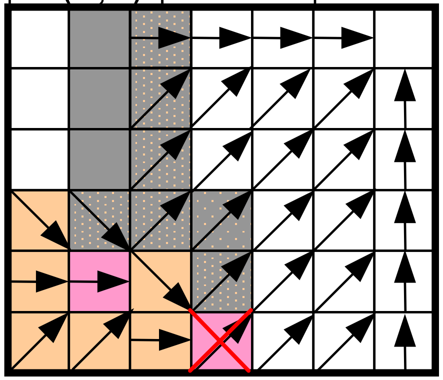
```
for each neighbor Y of X:
    if t(Y) = NEW or b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
    else if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then INSERT(X, h(X))
    else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

D* Example (16/22)

Pop (2,2) off the queue and expand it. This increases the h values of the nodes that pass through (2,2) and puts them back on the open list. It turns out that the relevant nodes (1,1), (1,2) and (1,3) are already on the open list so in effect, their position in the open list remains the same, but their h values are increased making them **raise** states.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=10000 k = 6.6 b=(3,2)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	= 10000 k = 7.0 b=(3,2)	h = 7.2 k = 7.2 b= (4,1)	~~h = 6.2~~ ~~k = 6.2~~ ~~b=(5,2)~~	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(2,1)	7.0
(3,1)	7.2
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000



```
if kold< h(X) then
    for each neighbor Y of X:
        if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
            b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

```
for each neighbor Y of X:
    if t(Y) = NEW or b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
    else if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then INSERT(X, h(X))
    else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

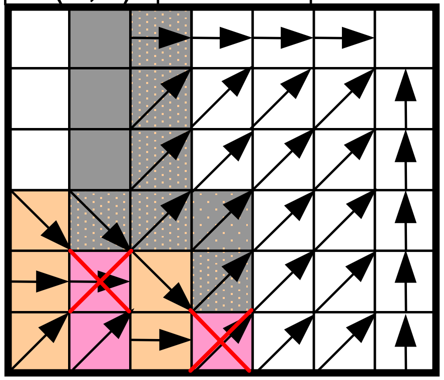
D* Example (17/22)

Pop (2,1) off the queue.

Because k<h and it cannot reduce the cost to any of its neighbors, this has no effect.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=10000 k = 6.6 b=(3,2)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	h = 10000 k = 7.0 b=(3,2)	h = 7.2 k = 7.2 b= (4,1)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(3,1)	7.2
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000



```
if kold< h(X) then
    for each neighbor Y of X:
        if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
            b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

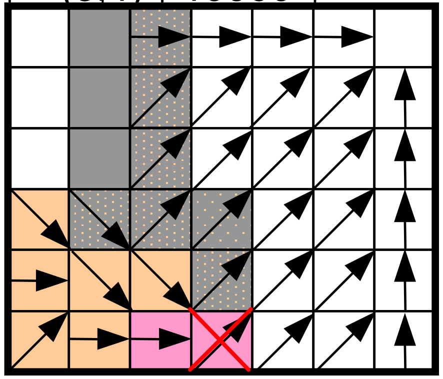
```
for each neighbor Y of X:
    if t(Y) = NEW or b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
    else if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then INSERT(X, h(X))
    else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

D* Example (18/22)

Pop (3,1) off the queue and expand it. This has the effect of redirecting the back pointers of (2,2) and (2,1) through (3,1) and putting them back on the open list with a k value equal to the minimum of the old and new h values. Because k equals h, they are now **lower** states.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=8.6 k = 8.6 b=(3,1)	= 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	= 8.2 k = 8.2 b=(3,1)	h = 7.2 k = 7.2 b= (4,1)	~~h = 6.2~~ ~~k = 6.2~~ ~~b=(5,2)~~	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(2,1)	8.2
(2,2)	8.6
(3,6)	10000
(3,5)	10000
(3,4)	10000



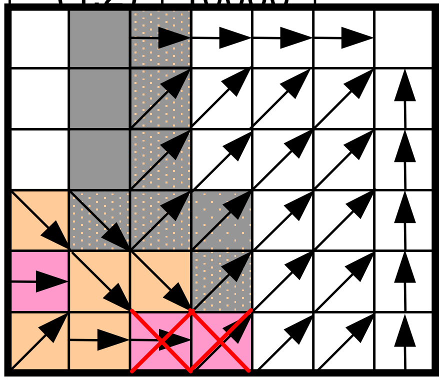
```
if kold= h(X) then  for each neighbor Y of X:
  if t(Y) = NEW or
  (b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) or
 (b(Y) ≠ X and h(Y) > h(X)+c (X,Y) )
     then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

D* Example (19/22)

Pop (1,2) off the queue and expand it. Since it cannot reduce the cost of any of its neighbors, and since the neighbors who could reduce its cost are still on the open list, this has no effect.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=8.6 k = 8.6 b=(3,1)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 1000 k = 8.0 b=(2,2)	= 8.2 k = 8.2 b=(3,1)	~~h = 7.2~~ ~~k = 7.2~~ ~~b= (4,1)~~	~~h = 6.2~~ ~~k = 6.2~~ ~~b=(5,2)~~	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(2,1)	8.2
(2,2)	8.6
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000



```
if kold< h(X) then
    for each neighbor Y of X:
        if h(Y) <= kold and h(X) > h(Y) + c(Y,X) then
            b(X) = Y;  h(X) = h(Y)+c(Y,X);
```

```
for each neighbor Y of X:
    if t(Y) = NEW or b(Y) =X and h(Y) ≠ h(X)+c (X,Y) ) then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
    else if  b(Y) ≠ X and h(Y) > h(X)+c (X,Y) then INSERT(X, h(X))
    else if  b(Y) ≠ X and h(X) > h(Y)+c (X,Y) and t(Y) = CLOSED and h(Y) > kold then INSERT(Y, h(Y))
```

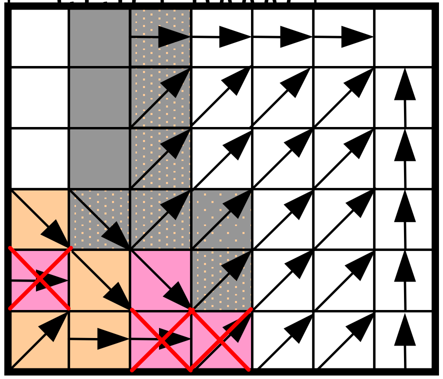
D* Example (20/22)

Pop (3,2) off the queue and expand it. Since it cannot reduce the cost of any of its neighbors this has no effect.

Since (3,2) is the robot's current position, the search ends here.

6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=8.6 k = 8.6 b=(3,1)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	h = 8.2 k = 8.2 b=(3,1)	h = 7.2 k = 7.2 b= (4,1)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,3)	8.0
(1,1)	8.0
(2,1)	8.2
(2,2)	8.6
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000



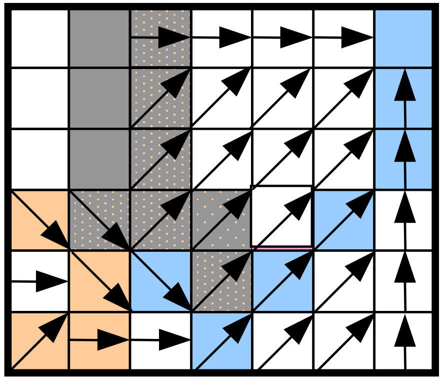
```
if kold= h(X) then  for each neighbor Y of X:
   if t(Y) = NEW or
   (b(Y) =X and h(Y) $\neq$ h(X)+c (X,Y) ) or
  (b(Y) $\neq$ X and h(Y) > h(X)+c (X,Y) )
      then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))
```

D* Example (21/22)

Determine optimal path from the current location to the goal by following gradient of h values

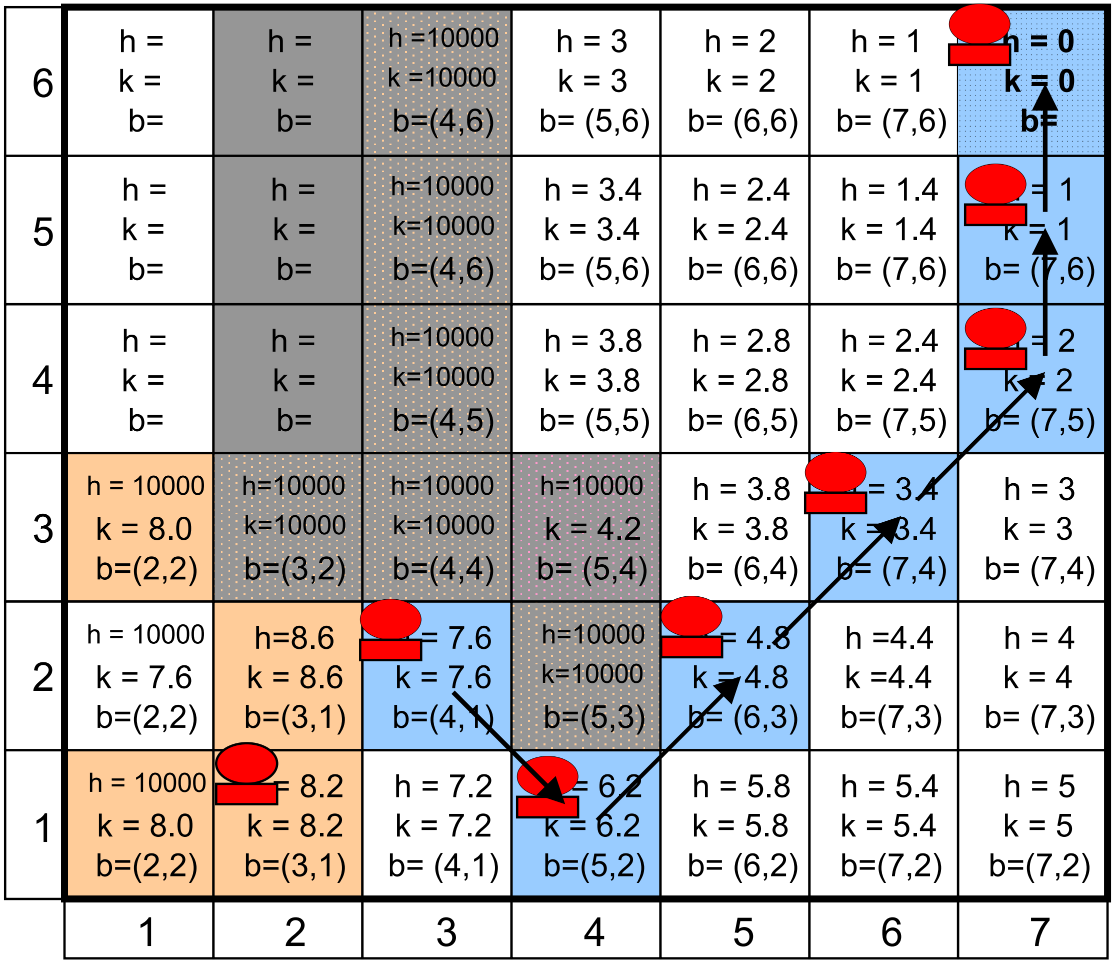
6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	h = 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	h = 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	h = 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=8.6 k = 8.6 b=(3,1)	h = 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	h = 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	h = 8.2 k = 8.2 b=(3,1)	h = 7.2 k = 7.2 b= (4,1)	h = 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7

State	k
(1,3)	8.0
(1,1)	8.0
(2,1)	8.2
(2,2)	8.6
(3,6)	10000
(3,5)	10000
(3,4)	10000
(4,2)	10000
(3,3)	10000

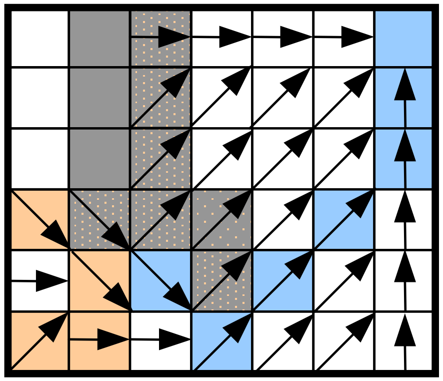


D* Example (22/22)

The robot then travels from its current location to the goal



6	h = k = b=	h = k = b=	h =10000 k =10000 b=(4,6)	h = 3 k = 3 b= (5,6)	h = 2 k = 2 b= (6,6)	h = 1 k = 1 b= (7,6)	**h = 0** **k = 0** **b=**
5	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,6)	h = 3.4 k = 3.4 b= (5,6)	h = 2.4 k = 2.4 b= (6,6)	h = 1.4 k = 1.4 b= (7,6)	= 1 k = 1 b= (7,6)
4	h = k = b=	h = k = b=	h=10000 k=10000 b=(4,5)	h = 3.8 k = 3.8 b= (5,5)	h = 2.8 k = 2.8 b= (6,5)	h = 2.4 k = 2.4 b= (7,5)	= 2 k = 2 b= (7,5)
3	h = 10000 k = 8.0 b=(2,2)	h=10000 k=10000 b=(3,2)	h=10000 k=10000 b=(4,4)	h=10000 k = 4.2 b= (5,4)	h = 3.8 k = 3.8 b= (6,4)	= 3.4 k = 3.4 b= (7,4)	h = 3 k = 3 b= (7,4)
2	h = 10000 k = 7.6 b=(2,2)	h=8.6 k = 8.6 b=(3,1)	= 7.6 k = 7.6 b=(4,1)	h=10000 k=10000 b=(5,3)	= 4.8 k = 4.8 b= (6,3)	h =4.4 k =4.4 b=(7,3)	h = 4 k = 4 b= (7,3)
1	h = 10000 k = 8.0 b=(2,2)	= 8.2 k = 8.2 b=(3,1)	h = 7.2 k = 7.2 b= (4,1)	= 6.2 k = 6.2 b=(5,2)	h = 5.8 k = 5.8 b= (6,2)	h = 5.4 k = 5.4 b=(7,2)	h = 5 k = 5 b=(7,2)
	1	2	3	4	5	6	7



We continue from (3,2)

D* Algorithm: Raise and Lower States

- *k(X)* → the priority of the state in an open list

- ***LOWER state***
  - k(X) = h(X)
  - Propagate information about path cost reductions (e.g. due to a reduced arc cost or new path to the goal) to its neighbors
  - For each neighbor Y of X, **if t(Y) = NEW** or **h(Y) > h(X) + c(X,Y)** then
    - Set h(Y) := h(X) + c(X,Y)
    - Set b(Y) = X
    - Insert Y into an OPEN list with k(Y) = h(Y) so that it can propagate cost changes to its neighbors

---

- ***RAISE state***
  - k(X) < h(X)
  - Propagate information about path cost increases (e.g. due to an increased arc cost) to its neighbors
  - For each neighbor Y of a RAISE state X,
    - If **t(Y) = NEW or (b(Y) = X and h(Y) ≠ h(X) + c(X,Y))** then
      insert Y into the OPEN list with k(Y) = h(X)+c(X,Y)
    - Else if **(b(Y) ≠ X and h(Y) > h(X) + c(X,Y))** then
      insert X into the OPEN list with k(X) = h(X)
    - Else if **(b(Y) ≠ X and h(X) > h(Y) + c(X,Y))** then
      insert Y into the OPEN list with k(Y) = h(Y)

Focused D*

- Introduce the focussing heuristic g(X,R) being an ***estimated path cost from robot location R to X***
- f(X,R) = k(X)+g(X,R) is ***an estimated path cost from R through X to G***
- Instead of k(X), sort the OPEN list by biased function values, f_B(X,Ri) = f(X,R_i)+d(R_i,R_0), where d(R_i,R_0) is the accrued bias function
  - Focussed D* assumes that the robot generally moves a little after it discovers discrepancies

Source Citation

- Atkar, P. N. and V. S. Lee Jr. (2001). “D* Algorithm”, *Presentation slide in sensor-based robot motion planning*, CMU, October, 2001.
- Russell, S. and P. Norvig (1995). *Artificial Intelligence: A modern approach*, Prentice Hall.
- Sensor-based robot motion planning lecture (2002).
- Stentz, A. (1994). "Optimal and efficient path planning for partially-known environments." *Proceedings of the IEEE International Conference on Robotics and Automation*, May, 1994.
- Stentz, A. (1995). “The focussed D* algorithm for real-time replanning.” *Proceedings of the International Joint Conference on Artificial Intelligence*, August, 1995.