

Robotic Motion Planning: A* and D* Search

Robotics Institute 16-735

<http://www.cs.cmu.edu/~motionplanning>

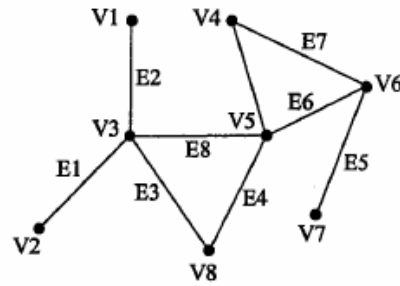
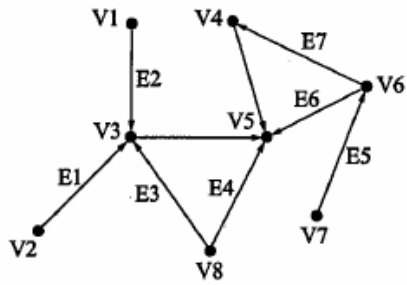
Howie Choset

<http://www.cs.cmu.edu/~choset>

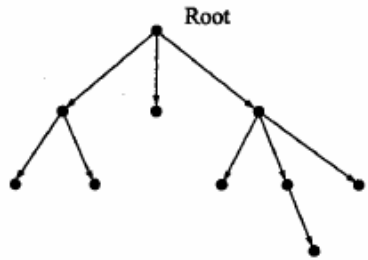
Outline

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

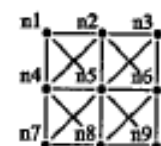
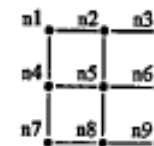
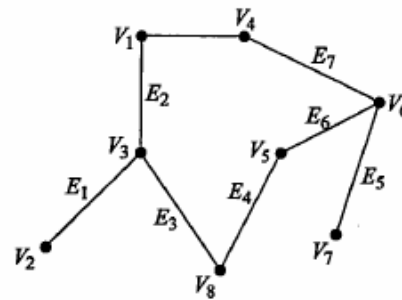
Graphs



Collection of Edges and Nodes (Vertices)



A tree



Search in Path Planning

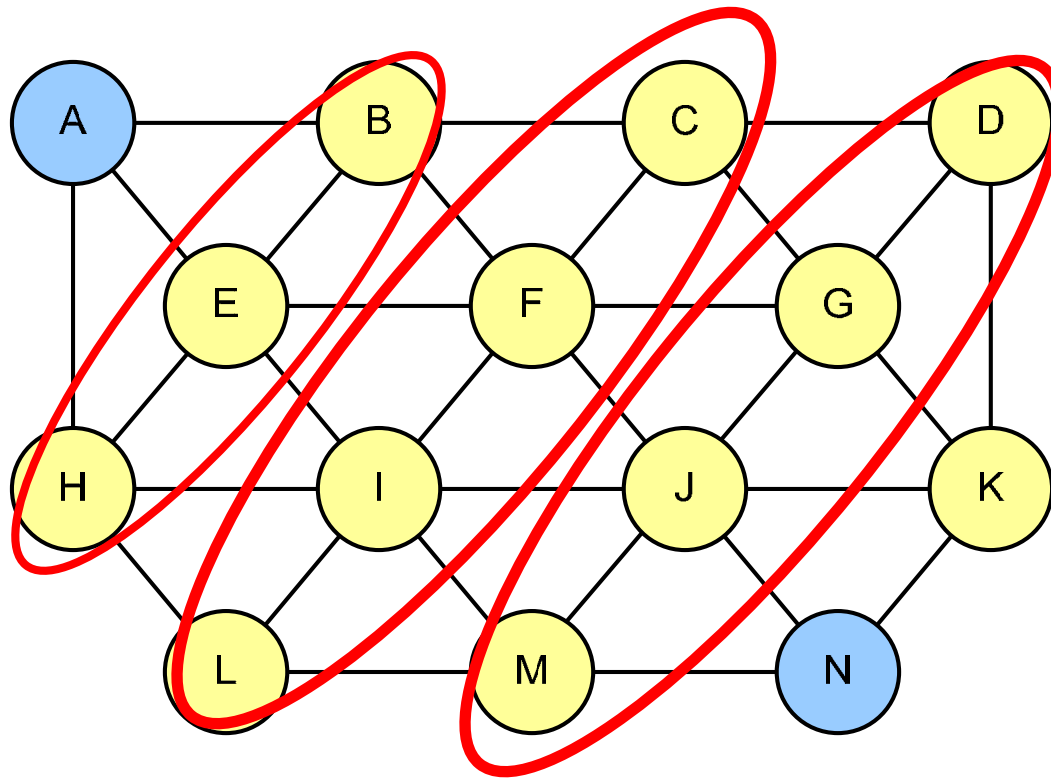
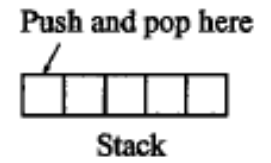
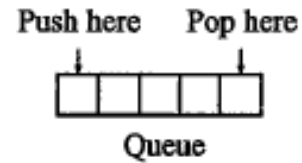
- Find a path between two locations in an unknown, partially known, or known environment
- Search Performance
 - Completeness
 - Optimality → Operating cost
 - Space Complexity
 - Time Complexity

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



— BFS

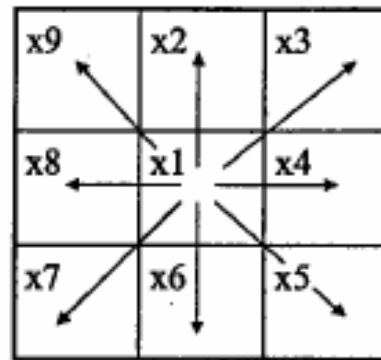
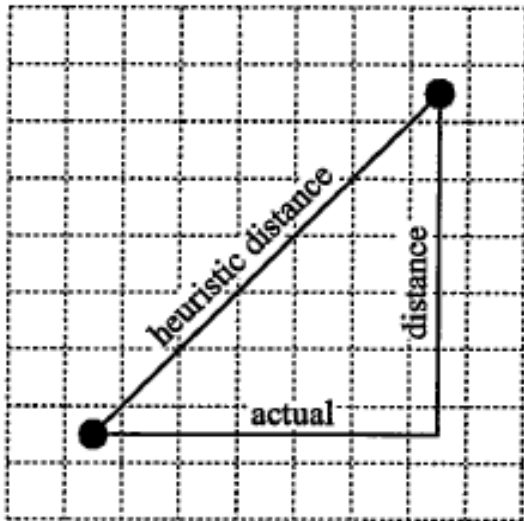
Informed Search: A*

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*

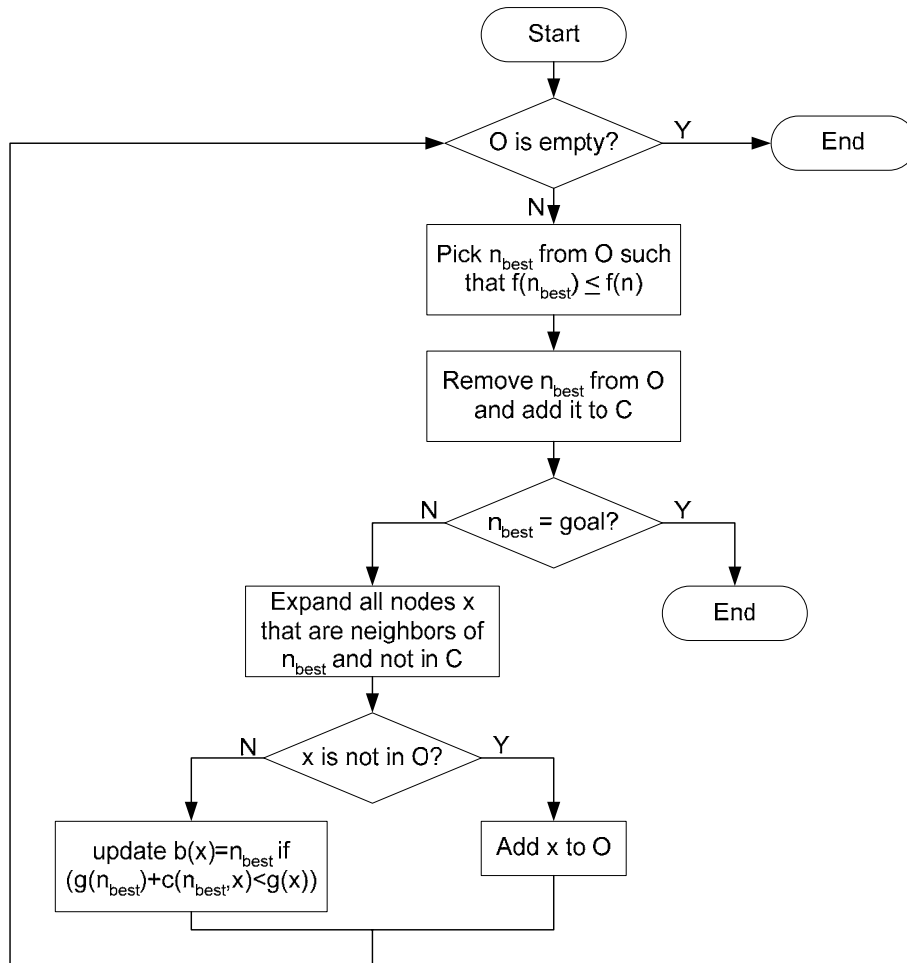
- Evaluation function, $f(n) = g(n) + h(n)$
- Operating cost function, $g(n)$
 - Actual operating cost having been already traversed
- Heuristic function, $h(n)$
 - Information used to find the promising node to traverse
 - Admissible \rightarrow never overestimate the actual path cost



$c(x_1, x_2) = 1$
 $c(x_1, x_9) = 1.4$
 $c(x_1, x_8) = 10000$, if x_8 is in obstacle, x_1 is a free cell
 $c(x_1, x_9) = 10000.4$, if x_9 is in obstacle, x_1 is a free cell

Cost on a grid

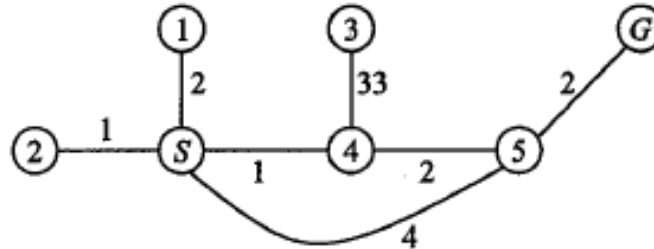
A*: Algorithm



The search requires 2 lists to store information about nodes

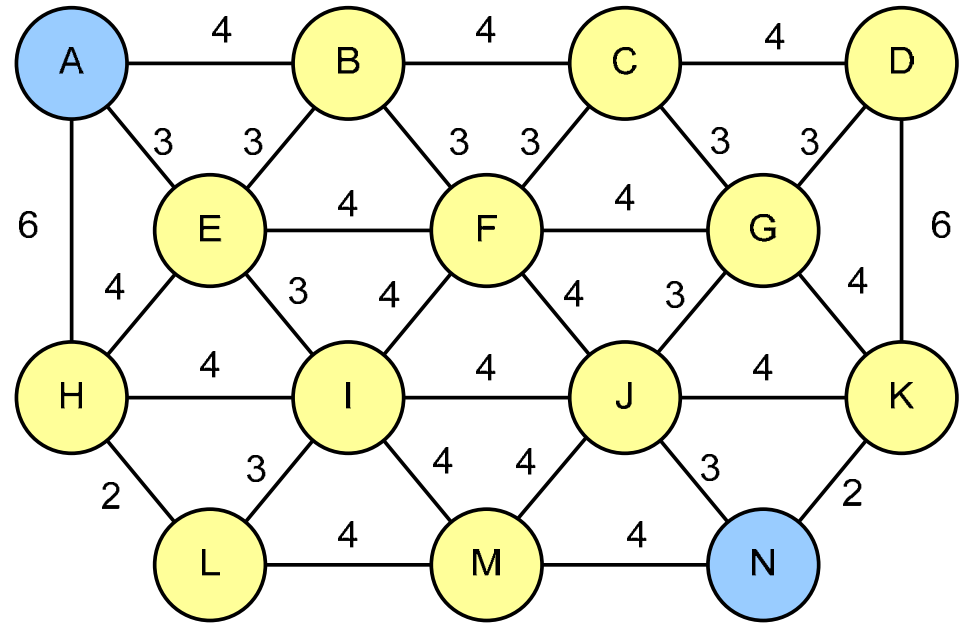
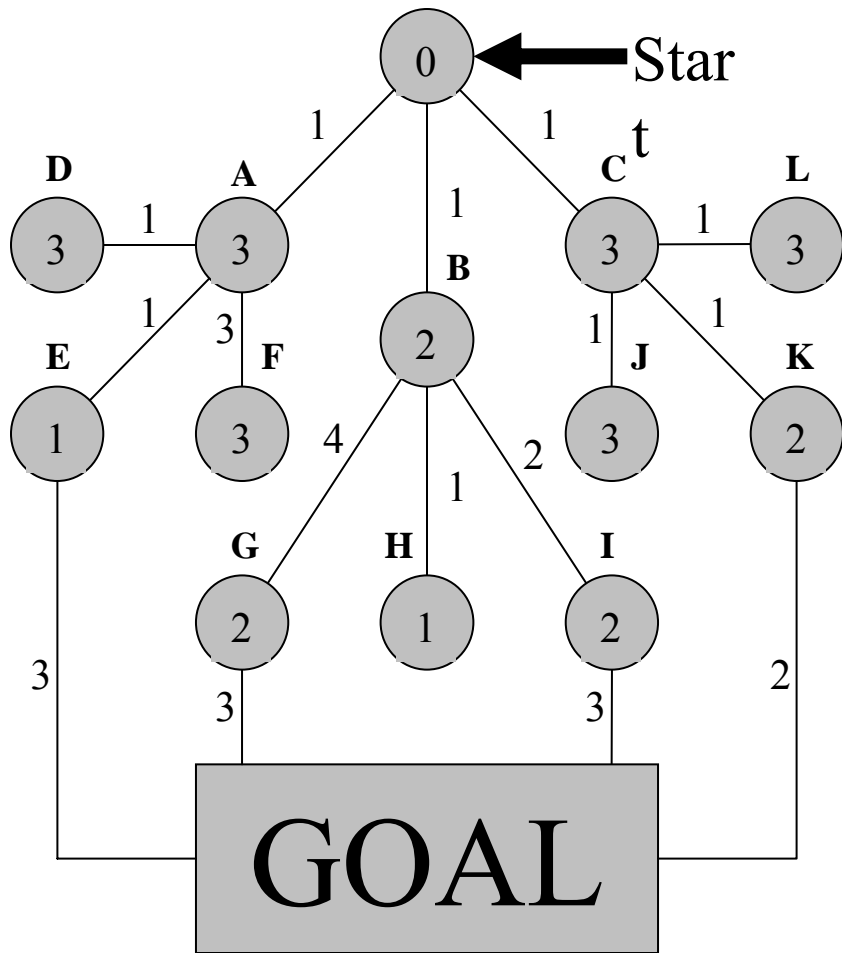
- 1) **Open list (O)** stores nodes for expansions
- 2) **Closed list (C)** stores nodes which we have explored

Dijkstra's Search: $f(n) = g(n)$

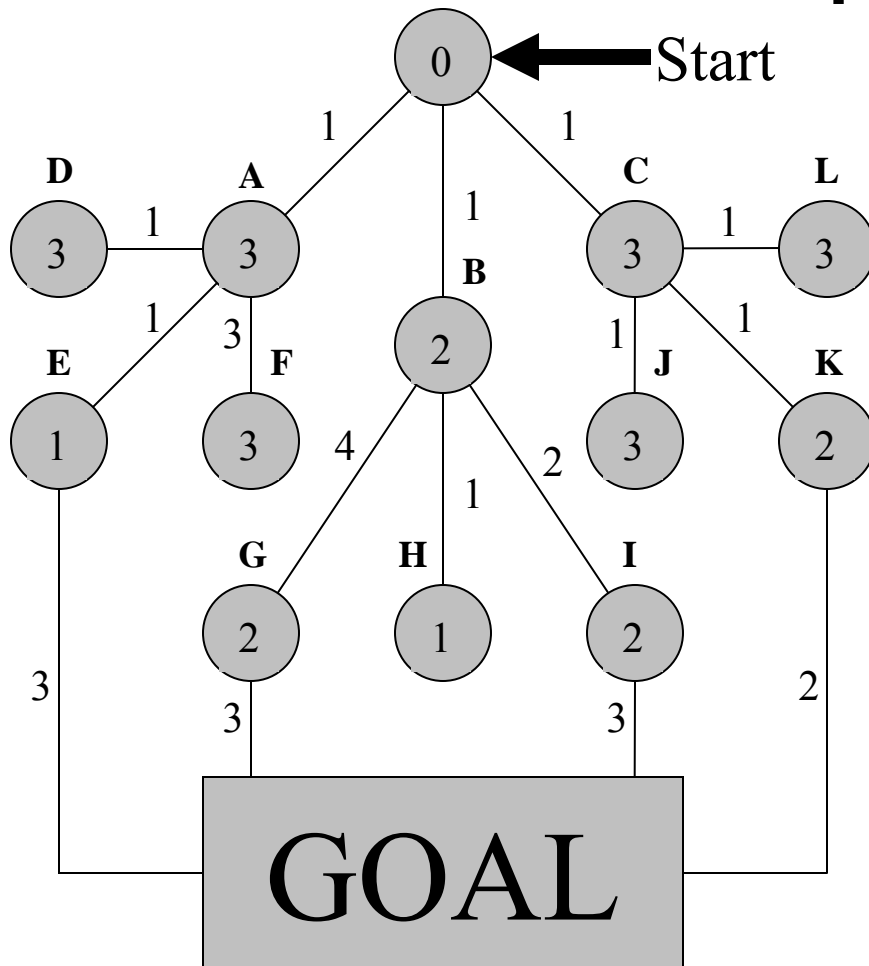


1. $O = \{S\}$
2. $O = \{1, 2, 4, 5\}$; $C = \{S\}$ (1,2,4,5 all back point to S)
3. $O = \{1, 4, 5\}$; $C = \{S, 2\}$ (there are no adjacent nodes not in C)
4. $O = \{1, 5, 3\}$; $C = \{S, 2, 4\}$ (1, 2, 4 point to S; 5 points to 4)
5. $O = \{5, 3\}$; $C = \{S, 2, 4, 1\}$
6. $O = \{3, G\}$; $C = \{S, 2, 4, 1\}$ (goal points to 5 which points to 4 which points to S)

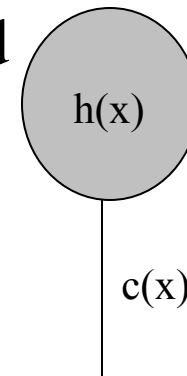
Two Examples Running A*



Example (1/5)



Legend



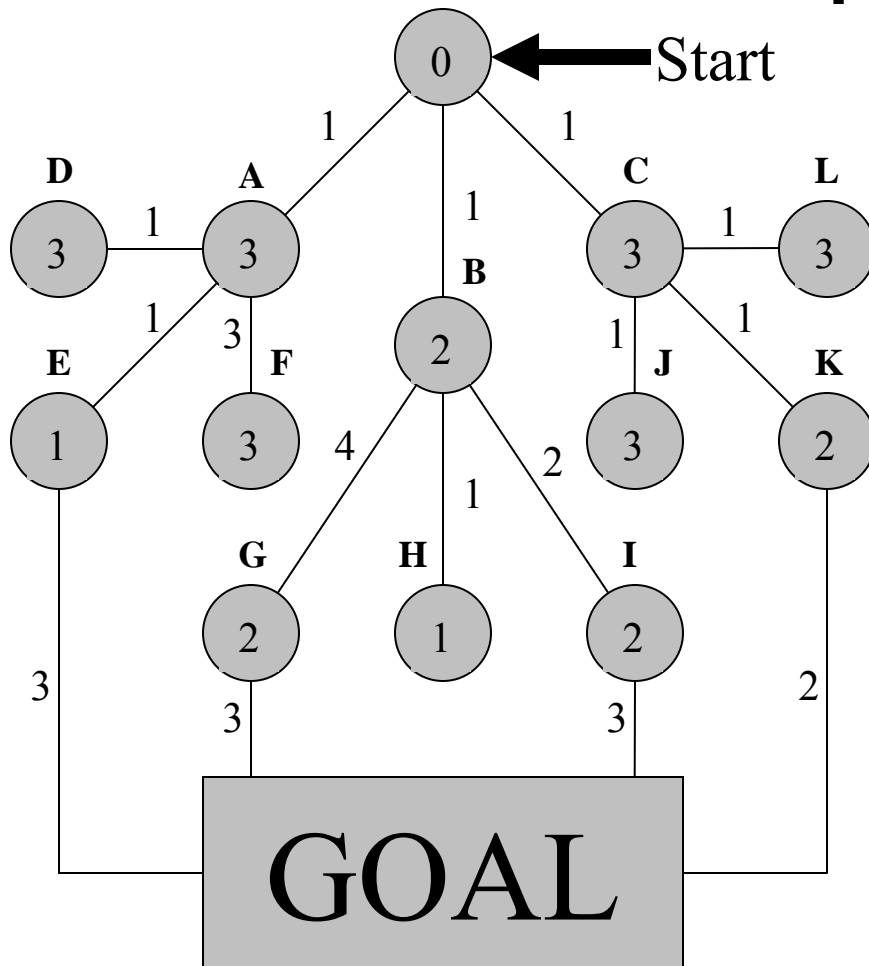
$$\text{Priority} = g(x) + h(x)$$

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)

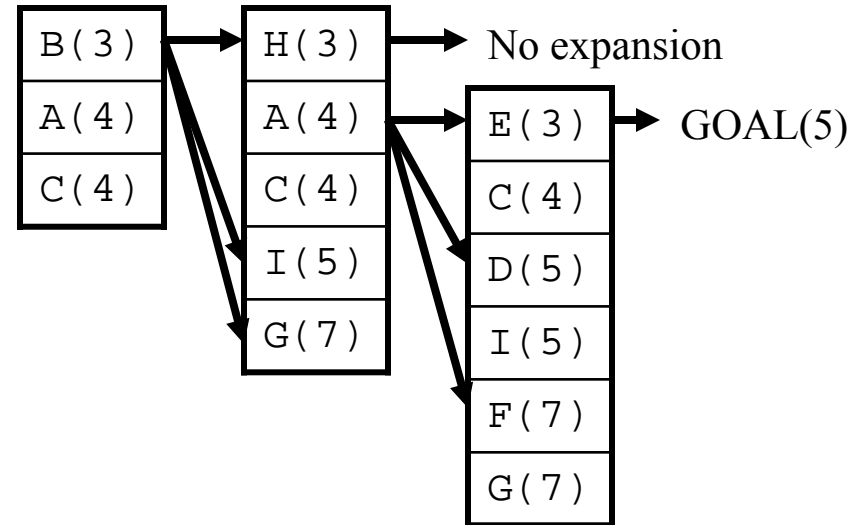
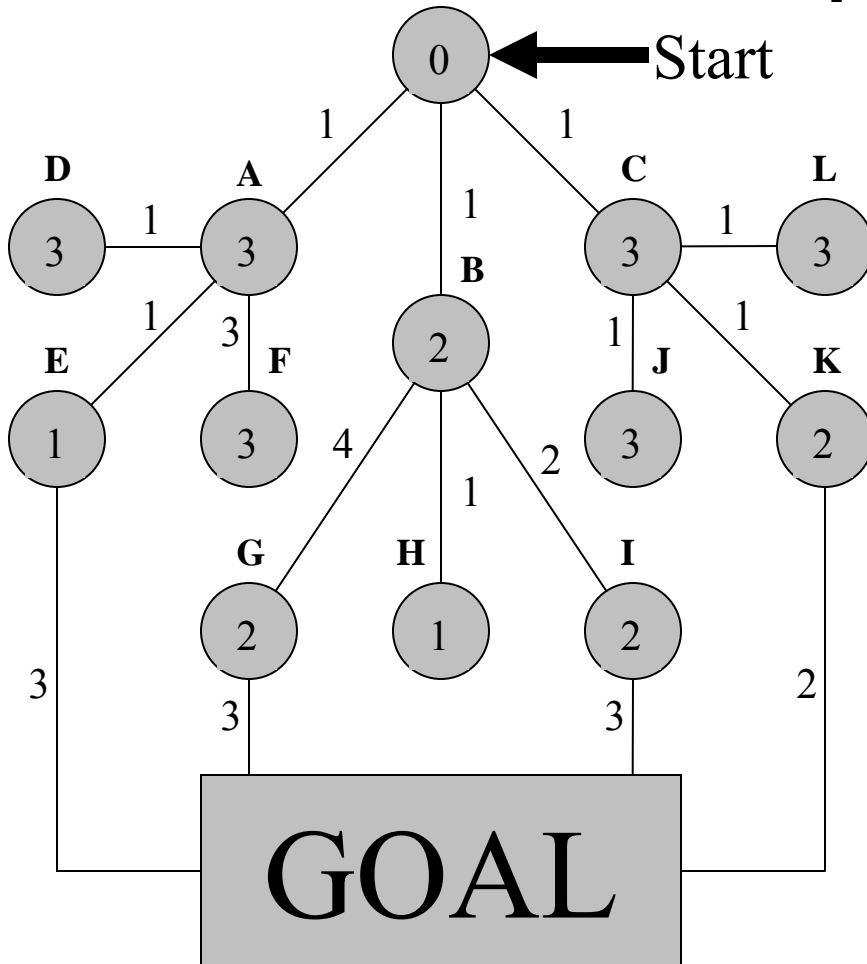
If goal not found, expand the first node in the priority queue (in this case, B)

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)



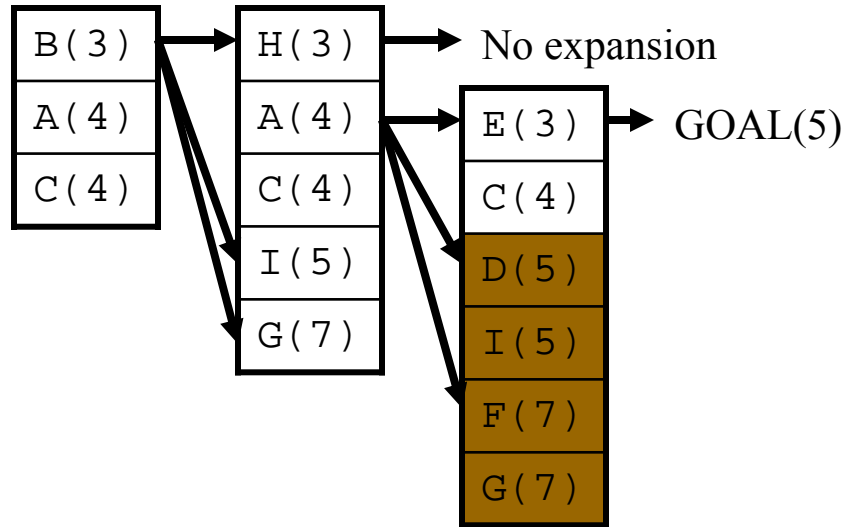
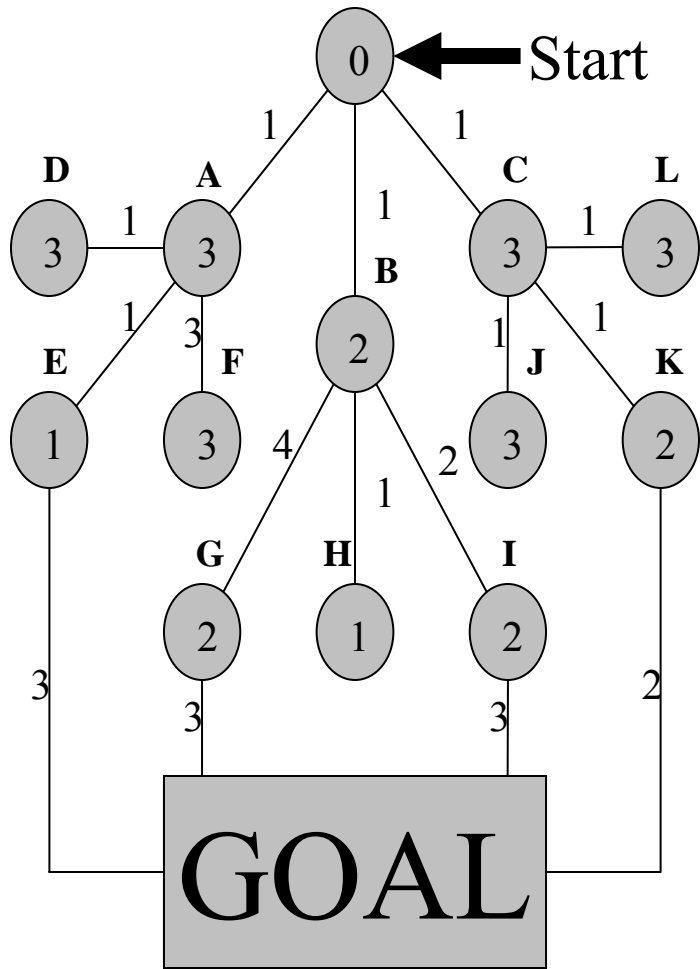
We've found a path to the goal:

Start => A => E => Goal

(from the pointers)

Are we done?

Example (4/5)

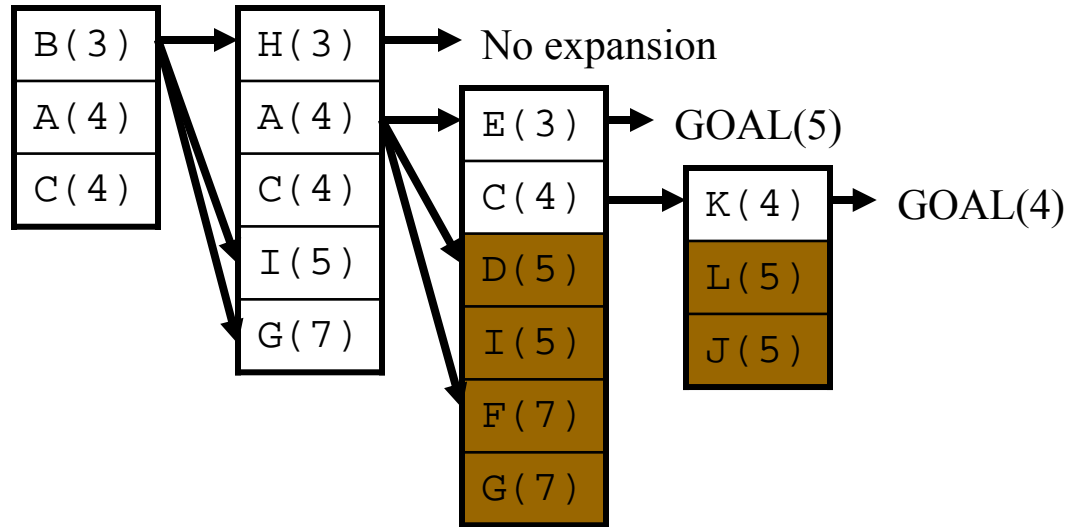
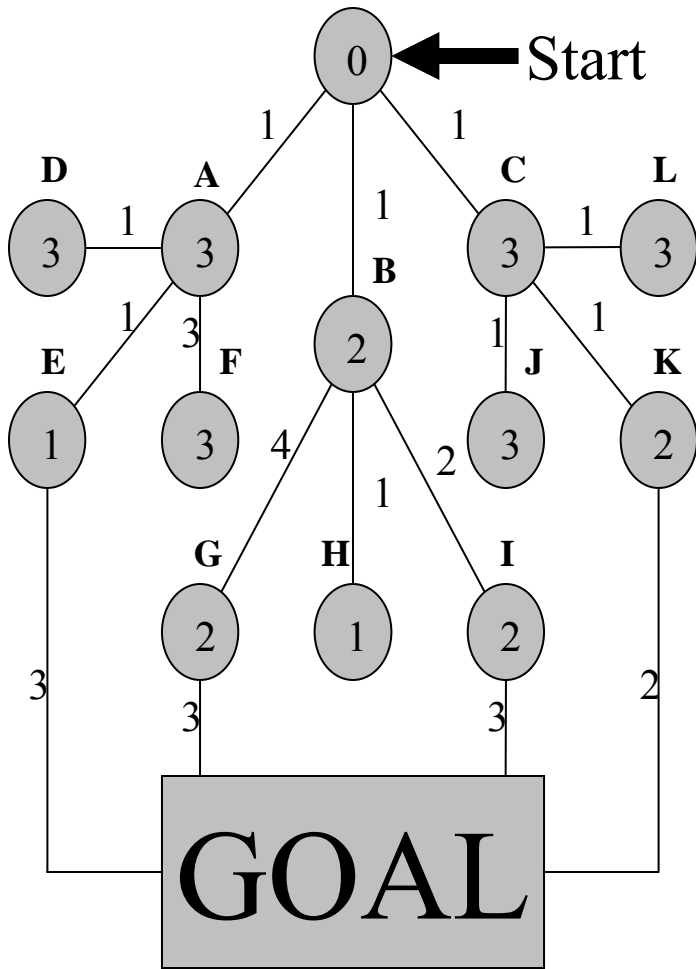


There might be a shorter path, but assuming non-negative arc costs, nodes with a lower priority than the goal cannot yield a better path.

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)



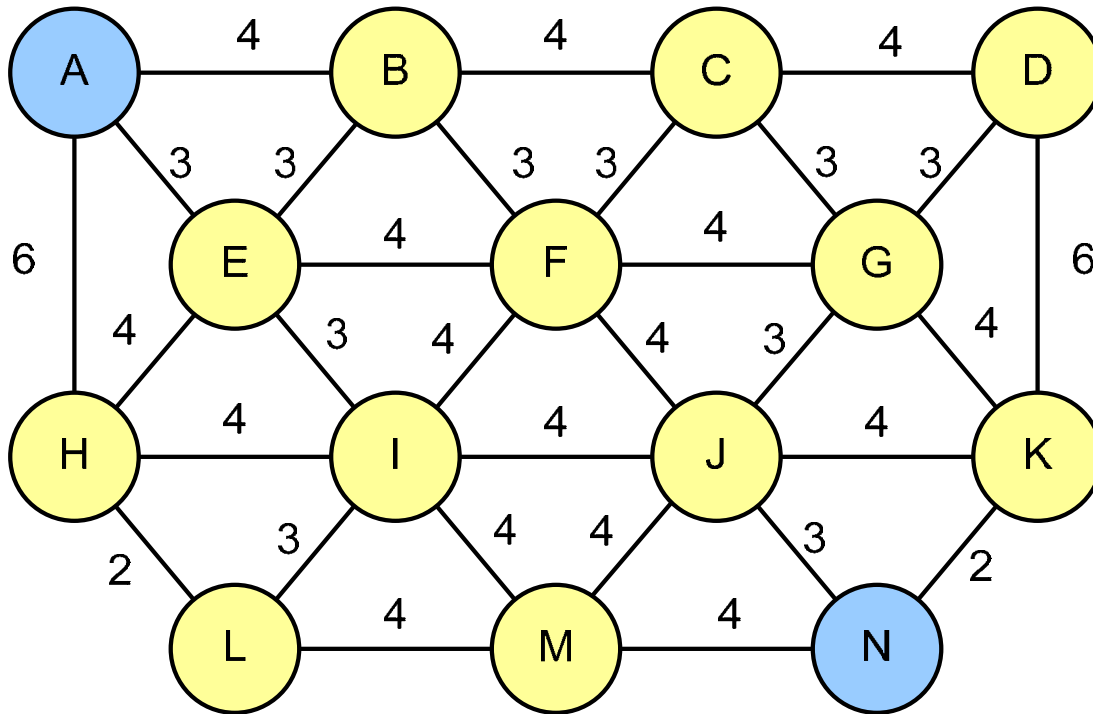
We can continue to throw away nodes with priority levels lower than the lowest goal found.

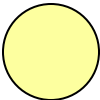
As we can see from this example, there was a shorter path through node K. To find the path, simply follow the back pointers.

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

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)

A*: Example (1/6)

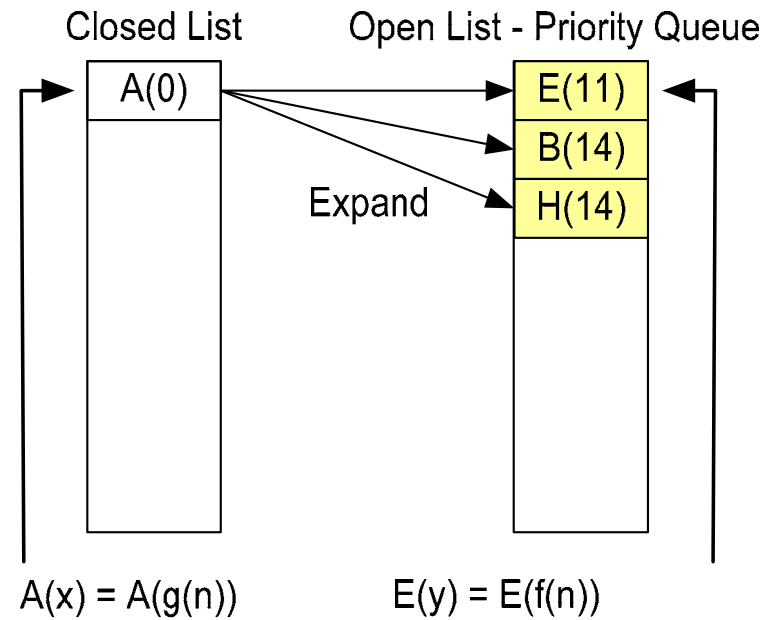
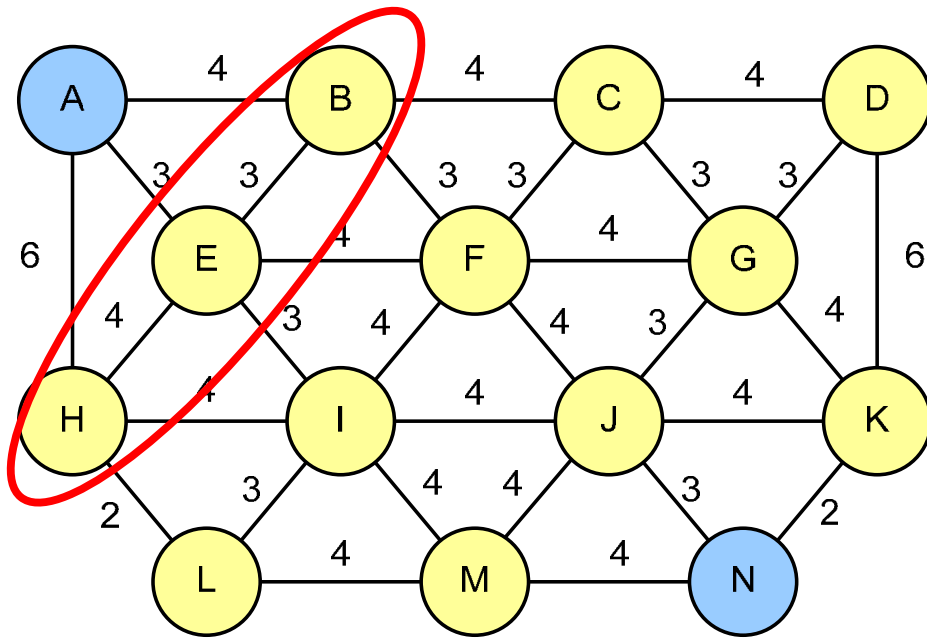


Legend  operating cost

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

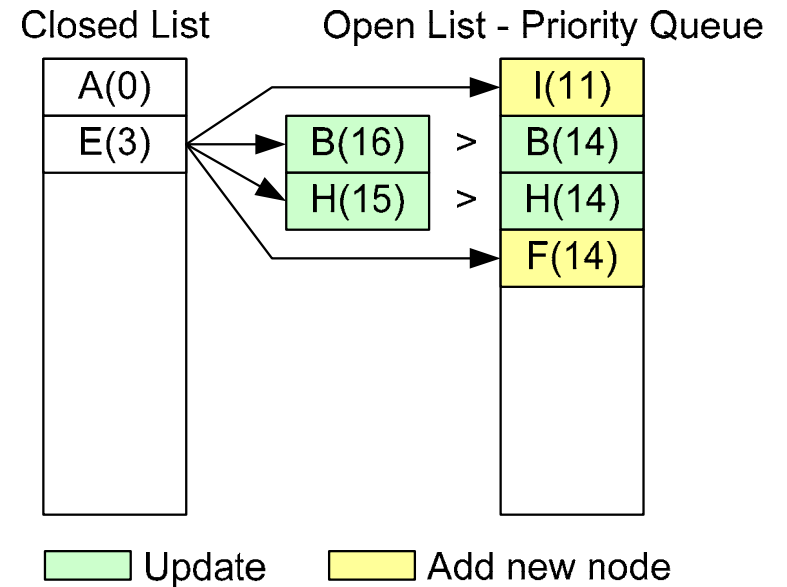
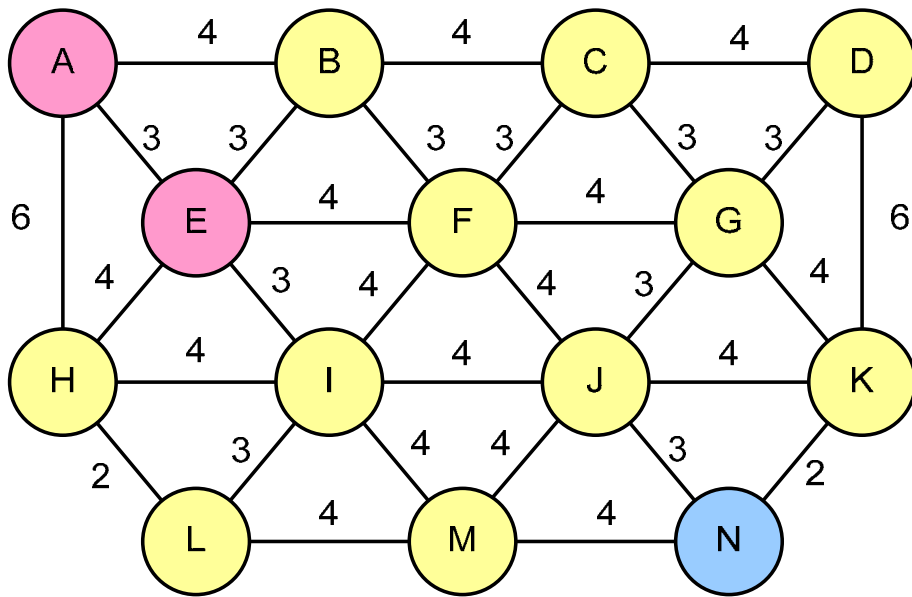
A*: Example (2/6)



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)

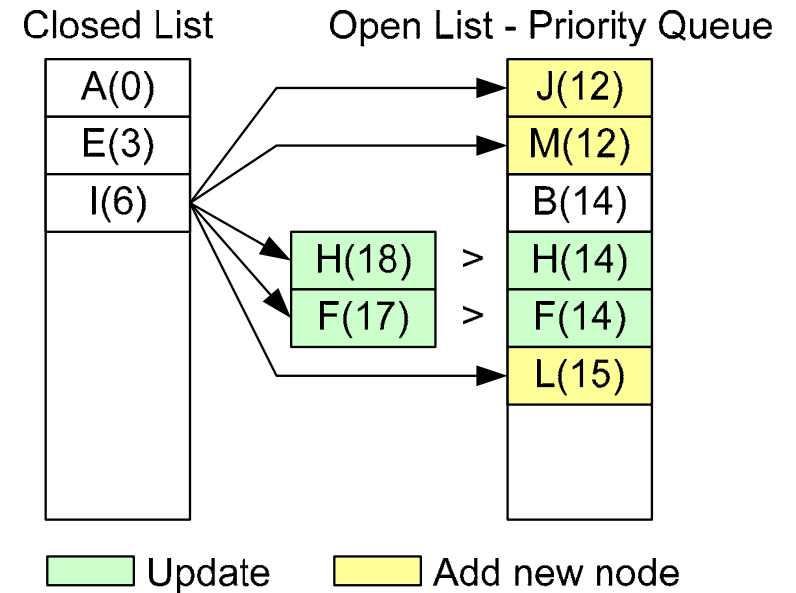
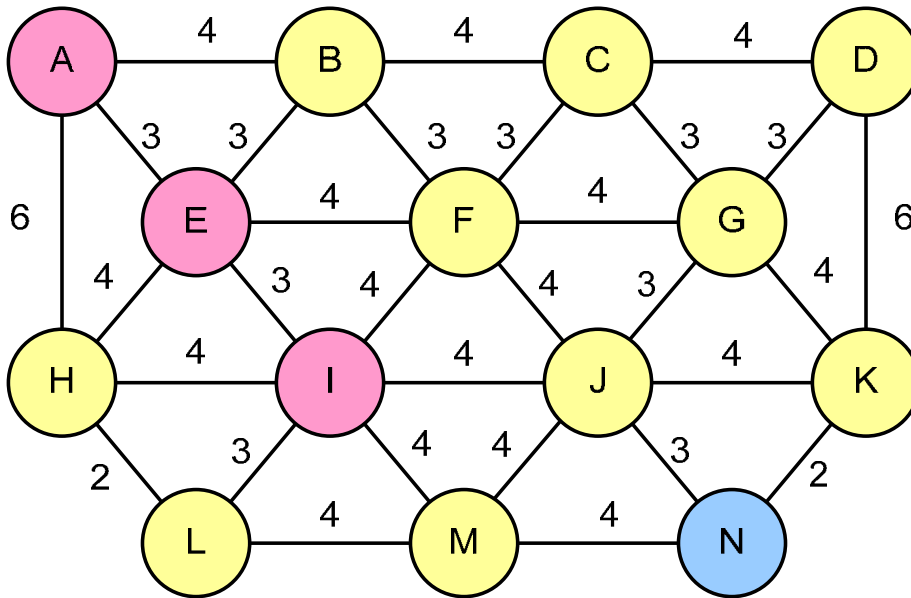


Heuristics

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

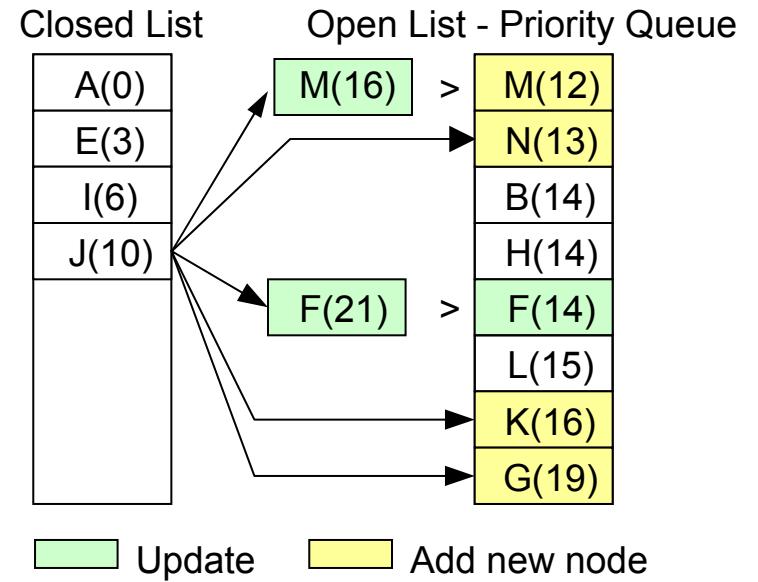
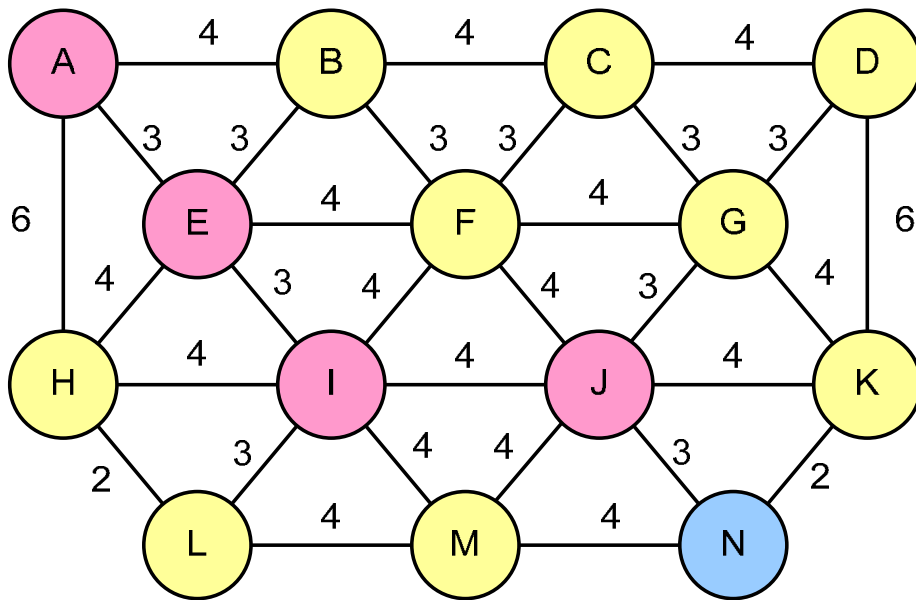
A*: Example (4/6)



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

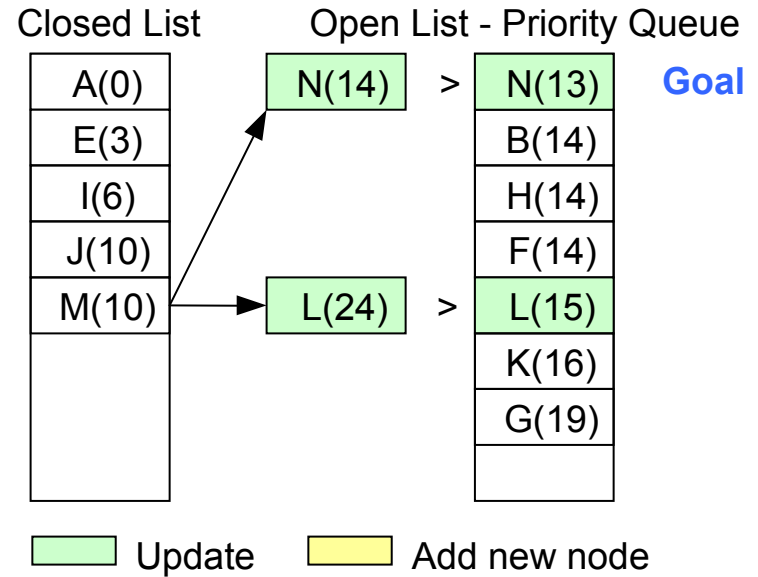
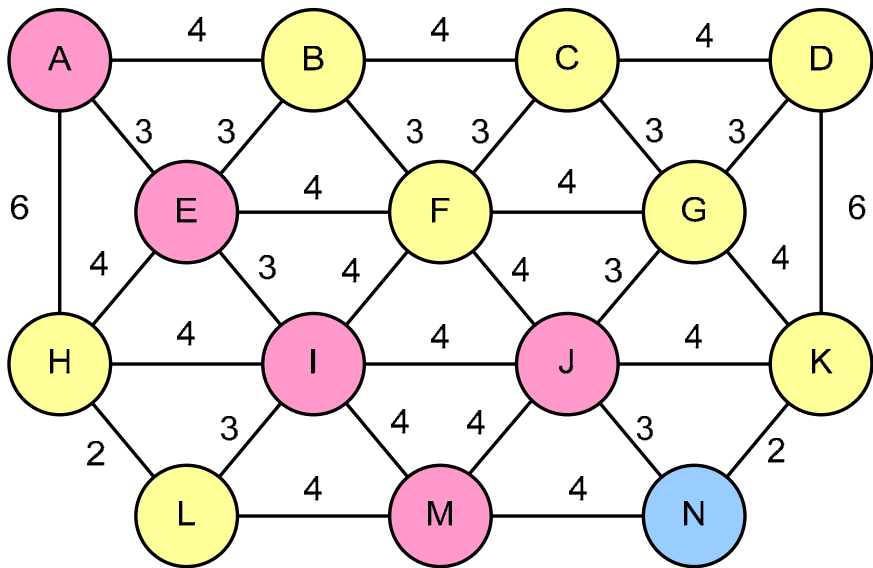
A*: Example (5/6)



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)

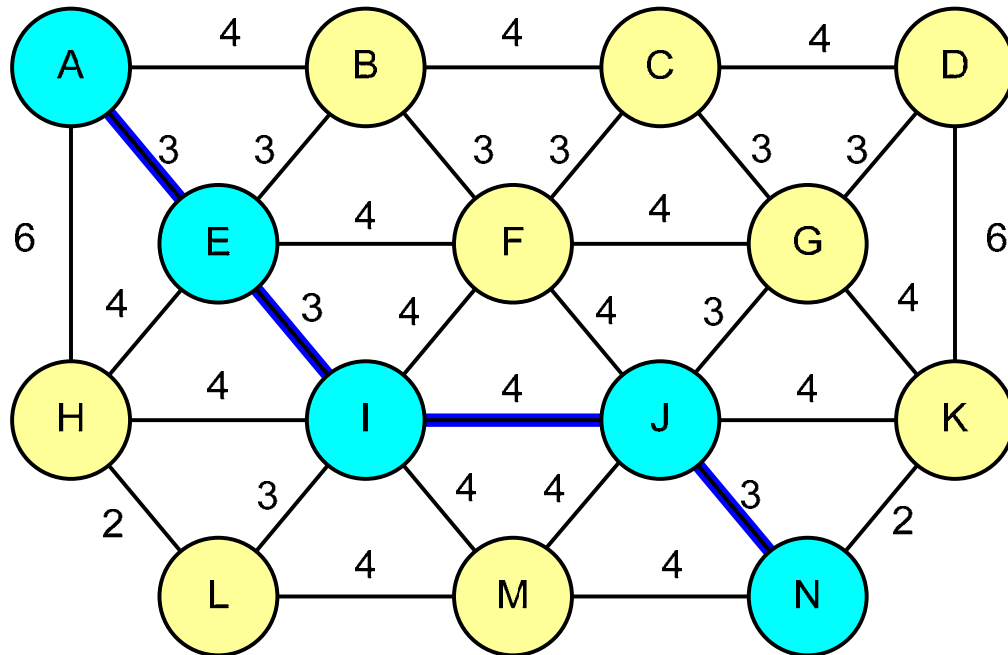


Heuristics

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**

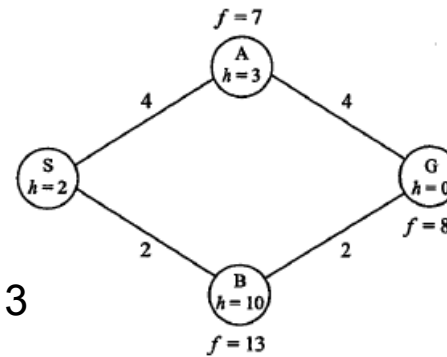
A*: Example Result



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



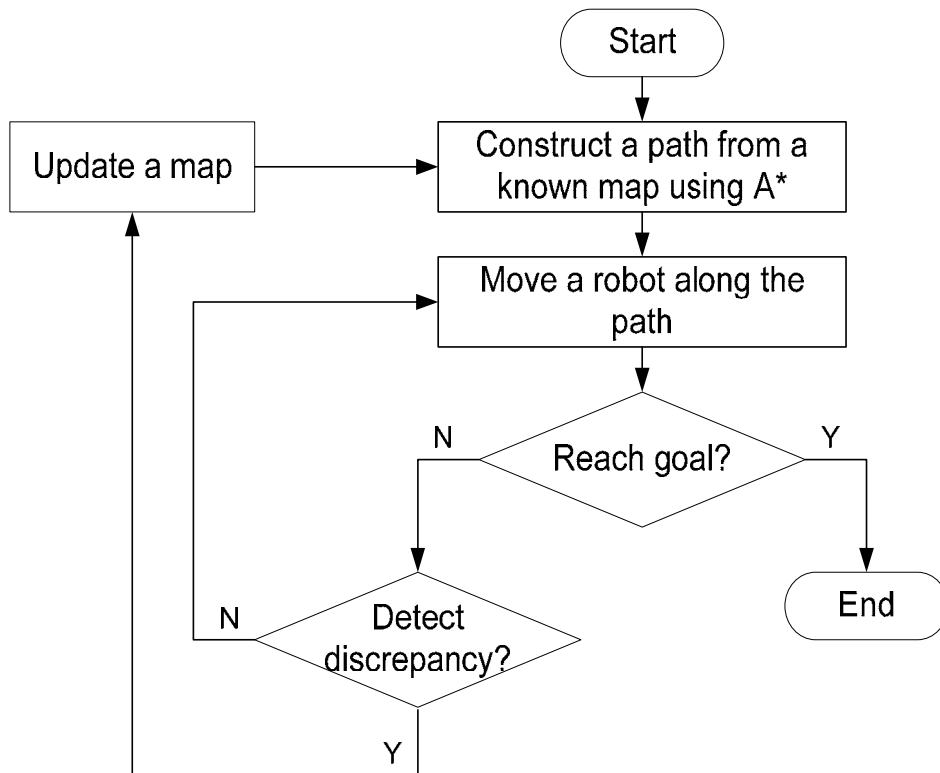
A*: Performance Analysis

- 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 \rightarrow 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



- 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

Dynamic backward search from goal to start

Dijkstra's Algorithm

$$f = g$$

D*

$$f = h$$

Not a heuristic!!

$$k = \min(h_{new}, h_{old})$$

A*

$$f = g + h$$

Just called D*

$$f = h + g$$

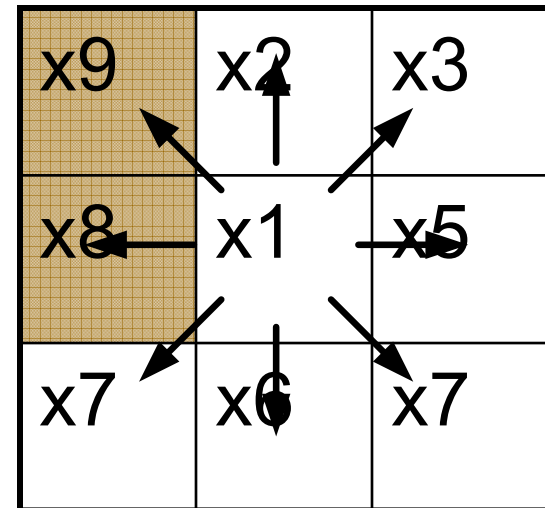
Not a heuristic!!

$$k = \min(h_{new}, h_{old})$$

$$key = k + g$$

D* Example (1/23)

- $X, Y \rightarrow$ states of a robot
- $b(X) = Y \rightarrow$ backpointer of a state X to a next state Y
- $c(X, Y) \rightarrow$ arc cost of a path from Y to X
- $r(X, Y) \rightarrow$ arc cost of a path from Y to X based on sensor
- $t(X) \rightarrow$ tag (i.e. NEW, OPEN, and CLOSED) of a state X
- $h(X) \rightarrow$ path cost
- $k(X) \rightarrow$ smallest value of $h(X)$ since X was placed on open list
- 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. $c(X1, X9) = 10000$)

Originally stated D* Algorithm

```
h(G)=0;
do
{
    kmin=PROCESS-STATE();
} while(kmin != -1 && start state not removed from open list);
if(kmin == -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
                {
                    kmin=PROCESS-STATE();
                }while(kmin< h(X) && kmin != -1);
                if(kmin== -1)
                    exit();
            }
    }while(1);
}
```

Our attempt at D* Algorithm

Input: List of all states L

Output: The goal state, if it is reachable, and the list of states L are updated so that the backpointer list describes a path from the start to the goal. If the goal state is not reachable, return NULL.

```
1: for each  $X \in L$  do
2:    $t(X) = \text{NEW}$ 
3: end for
4:  $h(G) = 0$ 
5:  $\text{INSERT}(O, G, h(G))$ 
6:  $X_c = S$ 
7:  $P = \text{INIT} - \text{PLAN}(O, L, X_c, G)$ 
8: if  $P = \text{NULL}$  then
9:   Return (NULL)
10: end if
11: while  $X_c \neq G$  do
12:    $\text{PREPARE} - \text{REPAIR}(O, L, X_c)$ 
13:    $P = \text{REPAIR} - \text{REPLAN}(O, L, X_c, G)$ 
14:   if  $P = \text{NULL}$  then
15:     Return (NULL)
16:   end if
17:    $X_c =$  the second element of  $P$  {Move to the next state in  $P$ }.
18: end while
19: Return ( $X_c$ )
```

Repair & Init

PREPARE – REPAIR(O, L, X_c)

```

1: for each state  $X \in L$  within sensor range of  $X_c$  and  $X_c$  do
2:   for each neighbor  $Y$  of  $X$  do
3:     if  $r(Y, X) \neq c(Y, X)$  then
4:       MODIFY – COST( $O, Y, X, r(Y, X)$ )
5:     end if
6:   end for
7:   for each neighbor  $Y$  of  $X$  do
8:     if  $r(X, Y) \neq c(X, Y)$  then
9:       MODIFY – COST( $O, X, Y, r(X, Y)$ )
10:    end if
11:  end for
12: end for

```

Node, as opposed to edge, perspective

INSERT(O, X, h_{new})

```

1: if  $t(X) = NEW$  then
2:    $k(X) = h_{new}$ 
3: else if  $t(X) = OPEN$  then
4:    $k(X) = \min(k(X), h_{new})$ 
5: else if  $t(X) = CLOSED$  then
6:    $k(X) = \min(h(X), h_{new})$ 
7: end if
8:  $h(X) = h_{new}$ 
9:  $t(X) = OPEN$ 
10: Sort  $O$  based on increasing  $k$  values

```

MODIFY – COST($O, X, Y, cval$)

```

1:  $c(X, Y) = cval$ 
2: if  $t(X) = CLOSED$  then
3:   INSERT( $O, X, h(X)$ )
4: end if
5: Return GET – KMIN( $O$ )

```

INIT – PLAN(O, L, X_c, G)

```

1: repeat
2:    $k_{min} = PROCESS – STATE(O, L)$ 
3: until ( $k_{min} = -1$ ) or ( $t(X_c) = CLOSED$ )
4:  $P = GET – BACKPOINTER – LIST(L, X_c, G)$ 
5: Return ( $P$ )

```

REPAIR – REPLAN(O, L, X_c, G)

```

1: repeat
2:    $k_{min} = PROCESS – STATE(O, L)$ 
3: until ( $k_{min} \geq h(X_c)$ ) or ( $k_{min} = -1$ )
4:  $P = GET – BACKPOINTER – LIST(L, X_c, G)$ 
5: Return ( $P$ )

```


D* Algorithm

- $k(X)$ → 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 $t(Y) = \text{NEW}$ 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

D* Algorithm

- **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 **$t(Y) = \text{NEW}$ or $(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$** then insert Y into the OPEN list with $k(Y) = h(X) + c(X, Y)$
 - Else if **$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$** then insert X into the OPEN list with $k(X) = h(X)$
 - Else if **$(b(Y) \neq X \text{ and } h(X) > h(Y) + c(X, Y))$** then insert Y into the OPEN list with $k(Y) = h(Y)$

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()



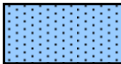


```
X = MIN-STATE()
if X= NULL then return -1
kold = GET-KMIN( ); DELETE(X);
if kold < h(X) then
    for each neighbor Y of X:
        if t(Y) ≠ new and h(Y) ≤ kold and h(X) > h(Y) +
c(Y,X) then
            b(X) = Y; h(X) = h(Y)+c(Y,X);
if kold = h(X) then
    for each neighbor Y of X:
        if t(Y) = 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))
else
    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))
Return GET-KMIN ( )
```

PROCESS-STATE()

D* Example (2/23)

Initially, all states have the tag NEW

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

6	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
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	

D* Example (3/23)

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 (4/23)

if $k(X) = h(X)$ then for each neighbor Y of X :

if $t(Y) = \text{NEW}$ or

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

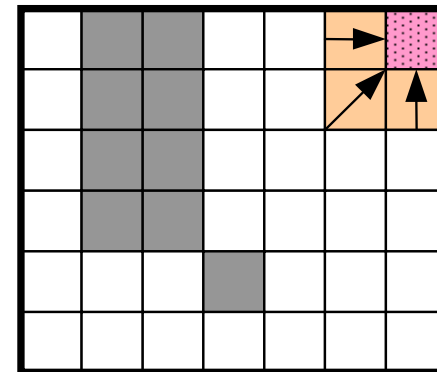
$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$

then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

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



Orange "Open" – on priority queue

Pink "Closed" & currently being expanded

Red X "Closed"

D* Example (5/23)

if $kold = h(X)$ then for each neighbor Y of X :

if $t(Y) = \text{NEW}$ or

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$

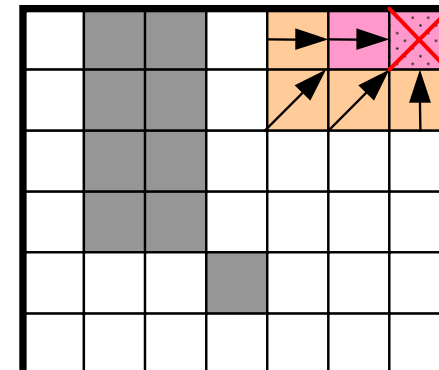
then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

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 (6/23)

if $kold = h(X)$ then for each neighbor Y of X :

if $t(Y) = \text{NEW}$ or

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$

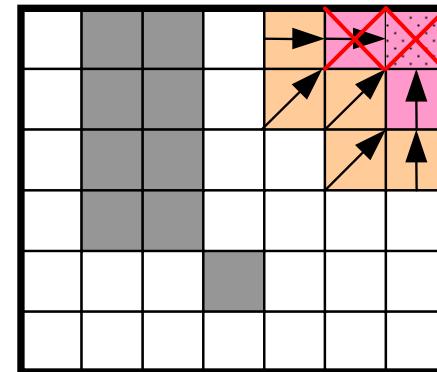
then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

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 (7/23)

if $kold = h(X)$ then for each neighbor Y of X :

if $t(Y) = \text{NEW}$ or

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

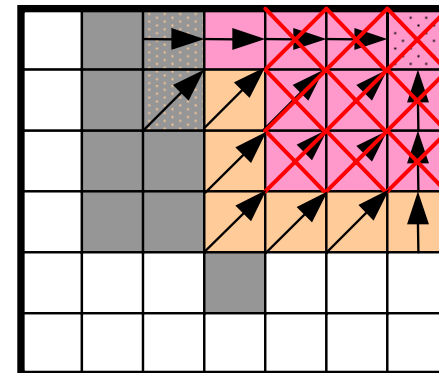
$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$

then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

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 (8/23)

if $kold = h(X)$ then for each neighbor Y of X :

if $t(Y) = \text{NEW}$ or

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

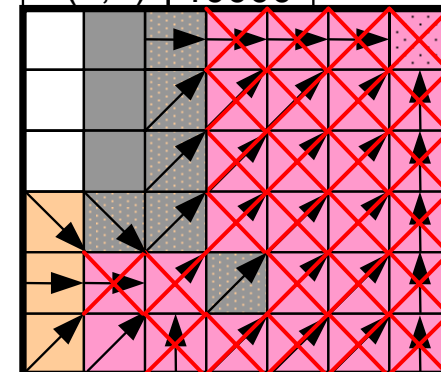
$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$

then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

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

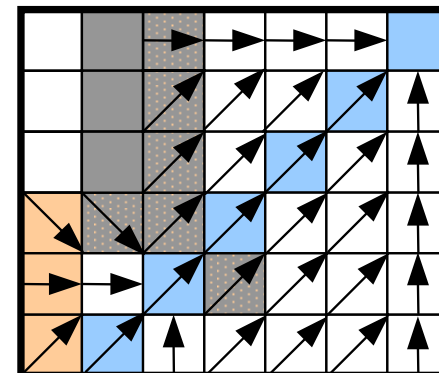
“Closed”

D* Example (9/23)

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

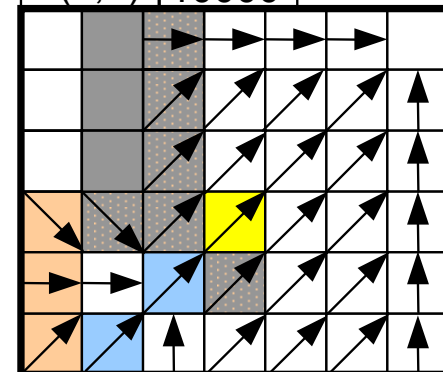
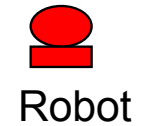


D* Example (10/23)

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)	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



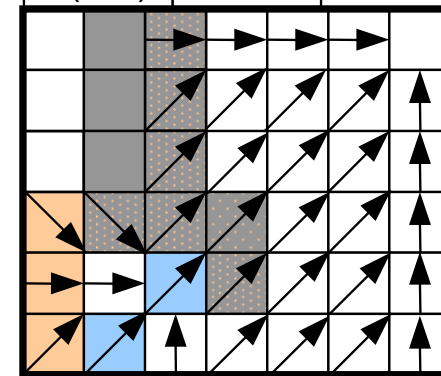
D* Example

(1 1 / 2 3)

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)	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
(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 $t(X) = \text{CLOSED}$ then INSERT (X,h(X))

Now, things will start to ⁴⁷ get interesting!

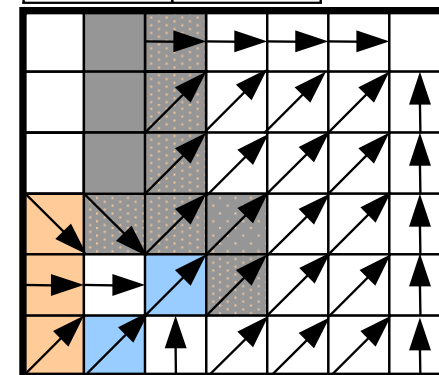
D* Example

(12/23)

(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 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)	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
(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

$(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or

$(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$ then

$b(Y) = X : INSERT(Y, h(X) + c(X, Y))$

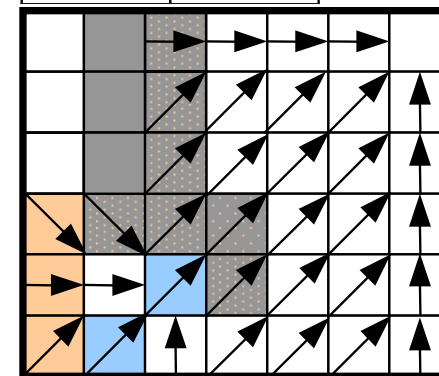
D* Example

(13/23)

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)	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
(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
 $(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or
 $(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$ then
 $b(Y) = X : INSERT(Y, h(X) + c(X, Y))$

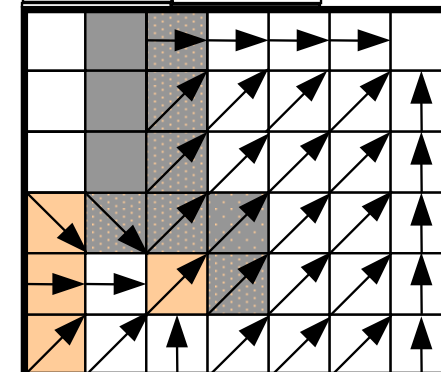
get interesting!

D* Example (14/23)

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). Note that we just assume all large numbers to be "equal," so 10004.2 is "equal" to 10003.8. 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)	h = 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)	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
(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 $h(Y) \leq kold$ and $h(X) > h(Y) + c(Y,X)$ then

$b(X) = Y$; $h(X) = h(Y) + c(Y,X)$;

else if $b(Y) \neq X$ and $h(X) > h(Y) + c(X,Y)$ and $t(Y) = CLOSED$ and $h(Y) > kold$ then $INSERT(Y, h(Y))$

D*

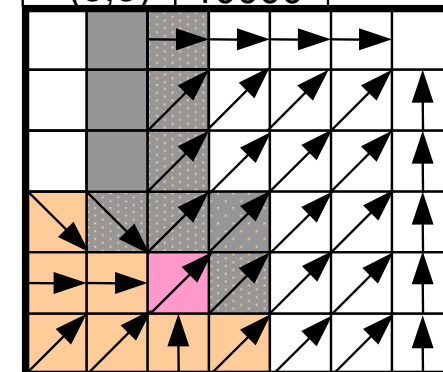
Example

(15/23)

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 $h(Y) \leq kold$ and $h(X) > h(Y) + c(X,Y)$ 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 $h(Y) \neq h(X) + c(X,Y)$ 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 $h(Y) > kold$ then $INSERT(Y, h(Y))$;

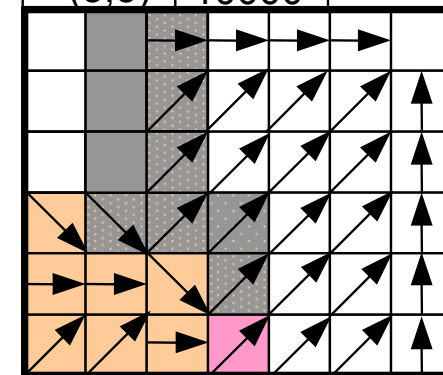
D* Example (16/23)

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 back 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 $k = \min(\text{old } h, \text{new } h)$. 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 $k = \min(\text{old } k, \text{new } h)$. 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 = 6.6 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 $k_{old} = h(X)$ then for each neighbor Y of X:
 if $t(Y) = \text{NEW}$ or
 $(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or
 $(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$
 then $b(Y) = X : \text{INSERT}(Y, h(X) + c(X, Y))$

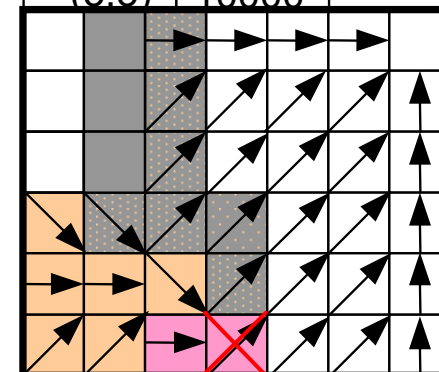
D* Example (17/23)

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.

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 = 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
(2,2)	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 $h(Y) \leq kold$ 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 $h(Y) \neq h(X) + c(X,Y)$ 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 $h(Y) > kold$ then INSERT(Y, $h(Y)$);

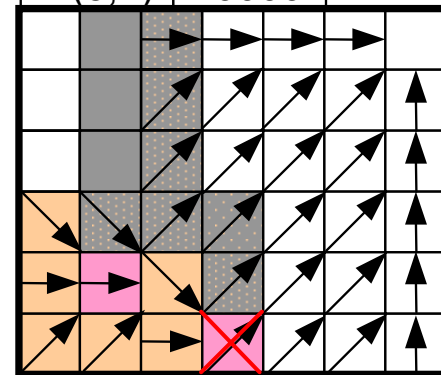
D* Example

(18/23)

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)	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
(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 $h(Y) \leq kold$ and $h(X) > h(Y) + c(X,Y)$ 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 $h(Y) \neq h(X) + c(X,Y)$ 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 $h(Y) > kold$ then $INSERT(Y, h(Y))$

D* Example

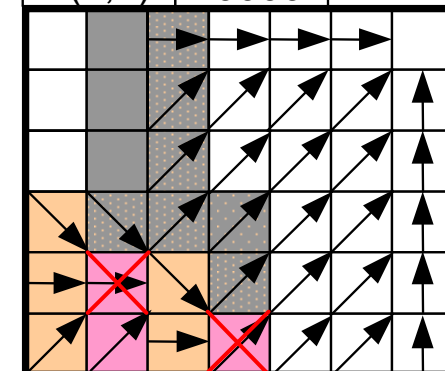
(19/23)

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 $h(Y) \leq kold$ and $h(X) > h(Y) + c(X,Y)$ 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 $h(Y) \neq h(X) + c(X,Y)$ 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, $h(X)$)

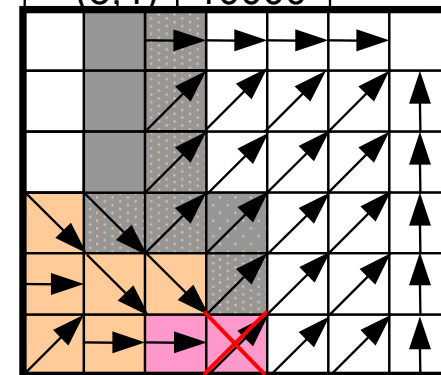
else if $b(Y) \neq X$ and $h(X) > h(Y) + c(X,Y)$ and $t(Y) = CLOSED$ and $h(Y) > kold$ then INSERT(Y, $h(Y)$)

D* Example (20/23)

Pop (3,1) off the queue and expand it. This has the effect of redirecting the backpointers 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 $k=h$, 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)	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
(1,2)	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



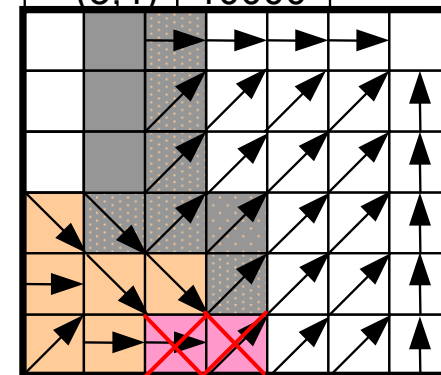
if $k_{old} = h(X)$ then for each neighbor Y of X :
 if $t(Y) = \text{NEW}$ or
 $(b(Y) = X \text{ and } h(Y) \neq h(X) + c(X, Y))$ or
 $(b(Y) \neq X \text{ and } h(Y) > h(X) + c(X, Y))$
 then $b(Y) = X$; $\text{INSERT}(Y, h(X) + c(X, Y))$

D* Example (21/23)

The search ends here because the minimum key on the open list is not less than the h value of the robot's current state. As such, we know that popping the next state off the open list will not result in a better path to the robot's current state.

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
(1,2)	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



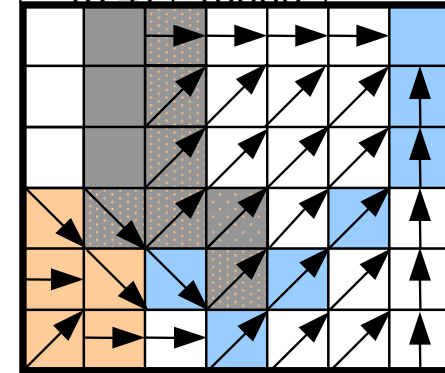
if kold= h
 if t(Y) =
 (b(Y) = X and h(Y) ≠ h(X)+c(X,Y)) or
 (b(Y) ≠ X and h(Y) > h(X)+c(X,Y))
 then b(Y) = X ; INSERT(Y, h(X)+c(X,Y))

D* Example (22/23)

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

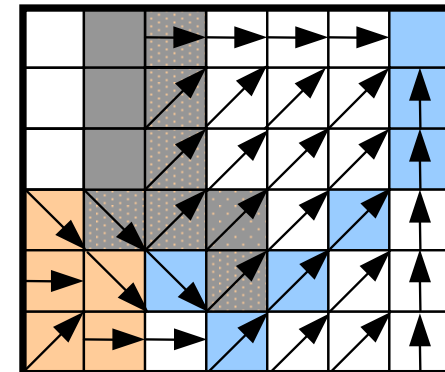
State	k
(1,2)	7.6
(3,2)	7.6
(1,3)	8.0
(1,1)	8.0
(2,1)	8.2
(2,2)	8.6
(3,6)	10000
(3,5)	10000
(3,4)	10000



D* Example (23/23)

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)	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



We continue from (3,2)

D* Algorithm, again

```
h(G)=0;
do
{
    kmin=PROCESS-STATE();
} while(kmin != -1 && start state not removed from Qu);
if(kmin == -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
                {
                    kmin=PROCESS-STATE();
                }while( k(Y) < h(Y) && kmin != -1);
                if(kmin== -1)
                    exit();
            }
    }while(1);
}
```

PROCESS-STATE()

```
X = MIN-STATE( )
if X= NULL then return -1
kold = GET-KMIN( ); DELETE(X);
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);
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))
else
  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))
Return GET-KMIN ( )
```

Other Procedures

MODIFY-COST(X,Y,cval)

$c(X,Y)=cval$

if $t(X) = \text{CLOSED}$ then INSERT $(X,h(X))$

Return GET-MIN ()

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 $t(X) = \text{NEW}$ then $k(X)=h_{new}$

if $t(X) = \text{OPEN}$ then $k(X)=\min(k(X),h_{new})$

if $t(X) = \text{CLOSED}$ then $k(X)=\min(k(X),h_{new})$ and $t(X) = \text{OPEN}$

Sort open list based on increasing k values;

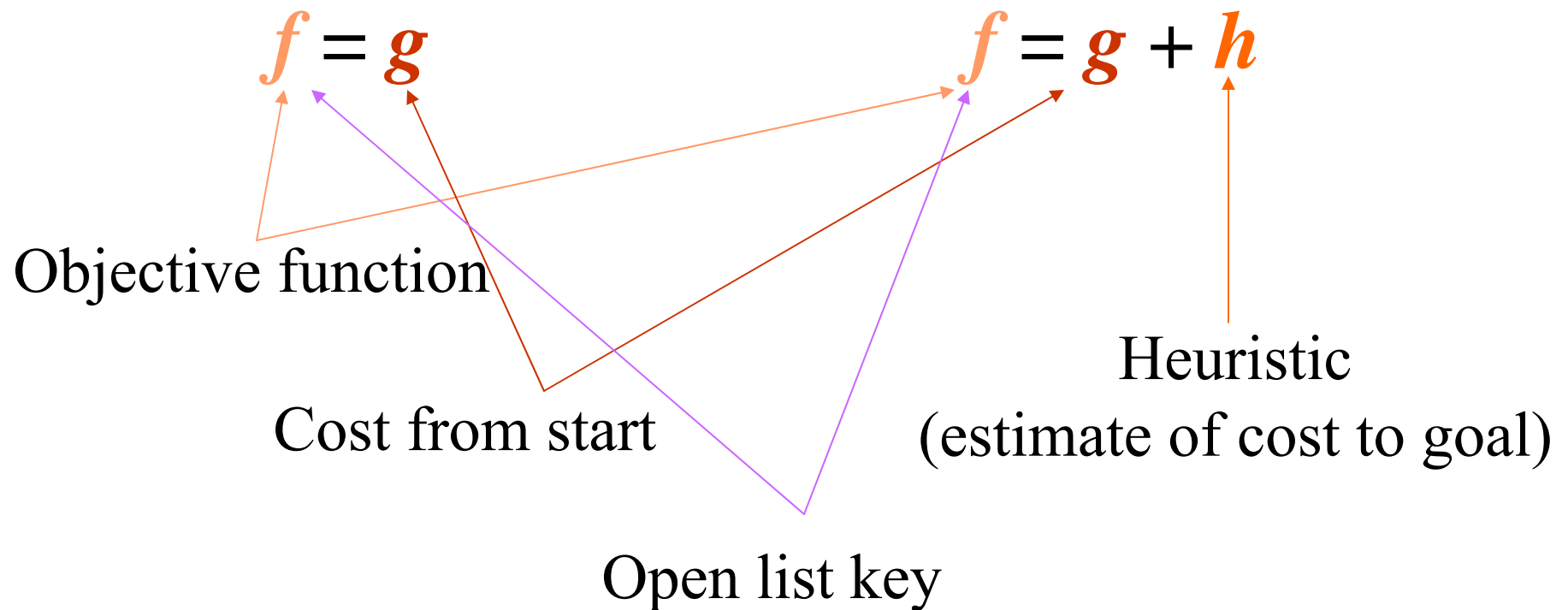
Focused D*

- 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, $f_B(X,R_i) = f(X,R_i) + d(R_i,R_0)$, 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

Forward search from start to goal

Dijkstra's Algorithm

A*

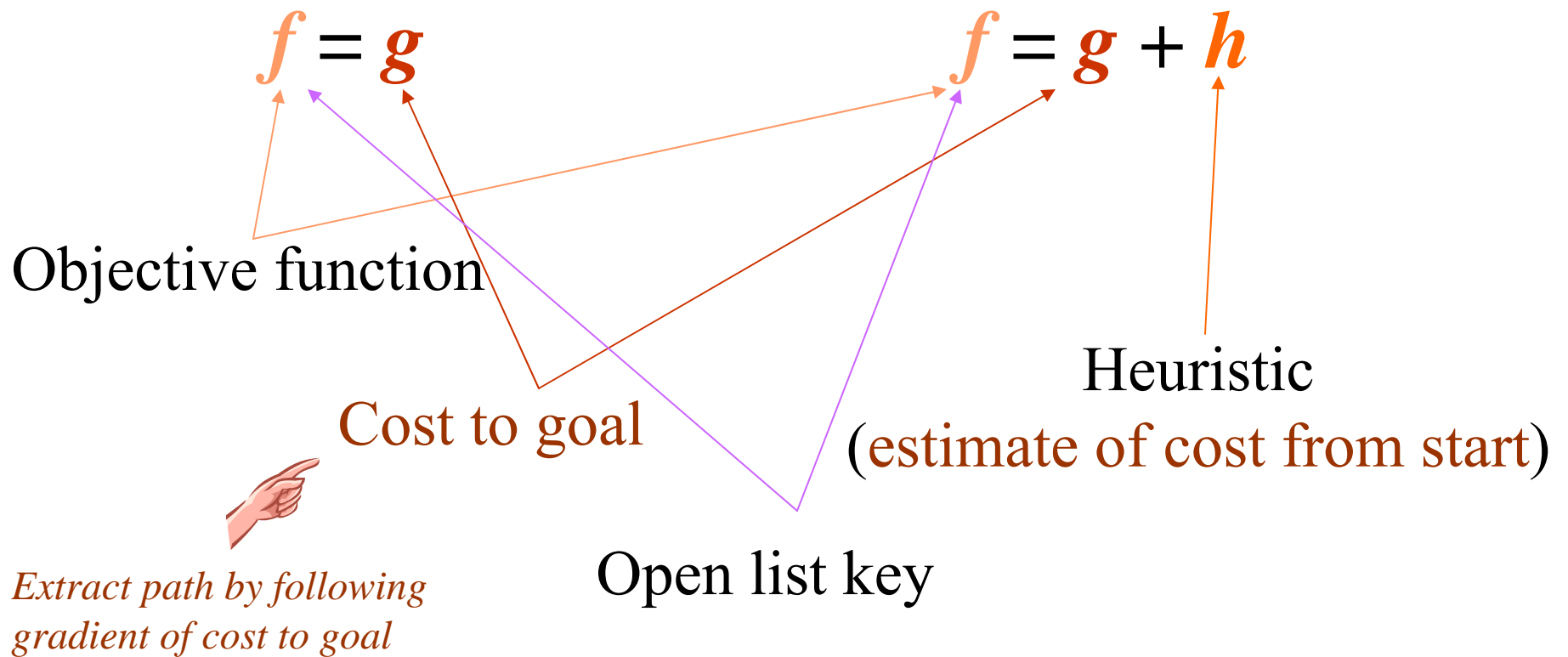


Backward search from goal to start

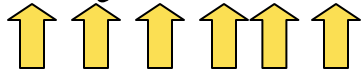


Dijkstra's Algorithm

A*

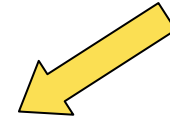


Dynamic backward search from goal to start

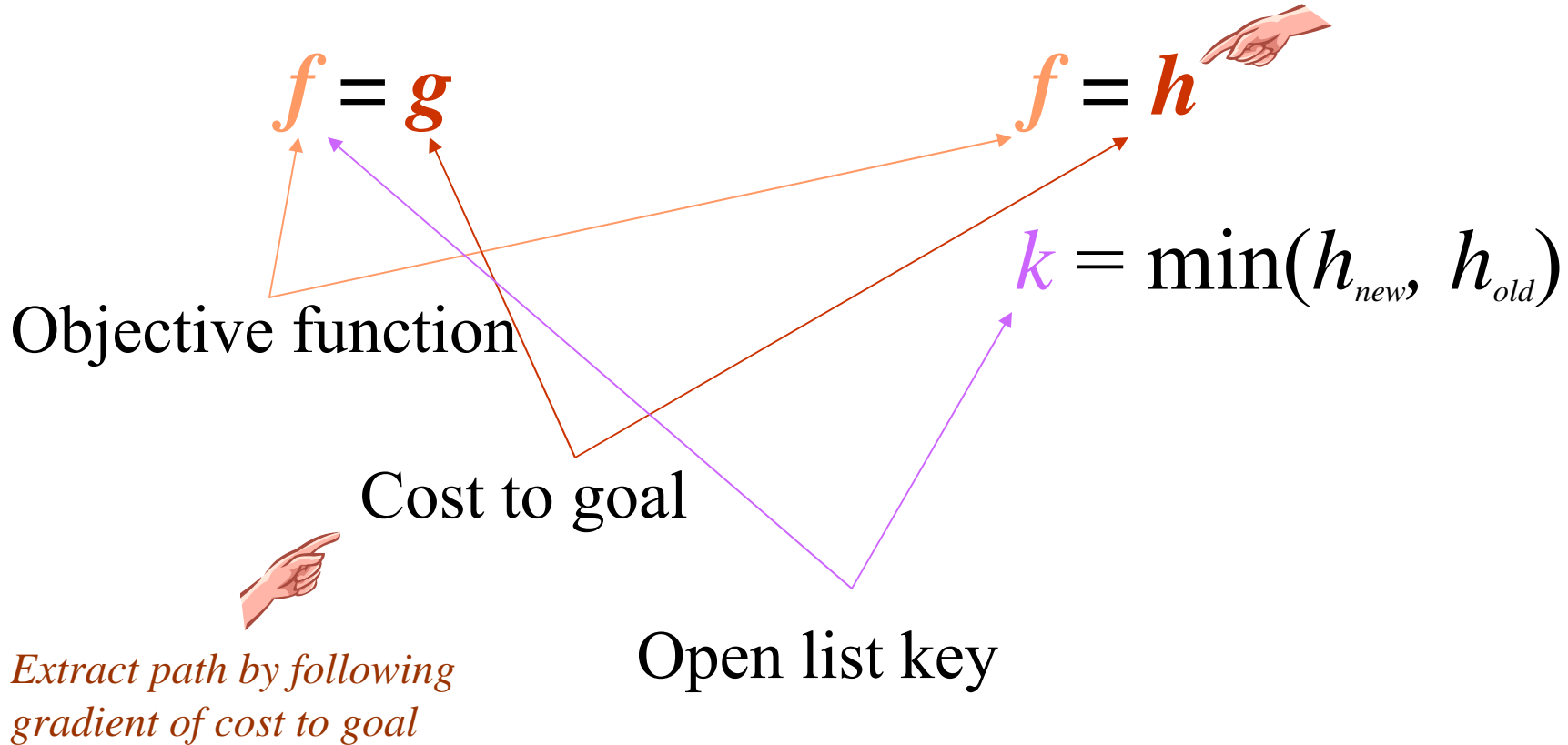


Dijkstra's Algorithm

D*



Not a heuristic!!



Unfortunate change in notation from g to h
And there is a k

Dynamic backward search from goal to start

A*

Focused D*

 *Not a heuristic!!*

$$f = g + h$$

$$f = h + g$$

Objective function

$$k = \min(h_{new}, h_{old})$$


$$key = k + g$$

Cost to goal

Open list key

Heuristic

(estimate of cost from start)

 *Extract path by following gradient of cost to goal*

D* Lite¹ notes and example

Mostly Created by Ayorkor Mills-Tettey
(so, she is our local expert!!)

¹ S. Koenig and M. Likhachev, 2002.



D*¹, D* Lite²

- Based on A*
 - Store pending nodes in a priority queue
 - Process nodes in order of increasing objective function value
- Incrementally repair solution paths when changes occur
 - ⇒ Fast replanning

¹ A. Stentz, 1994.

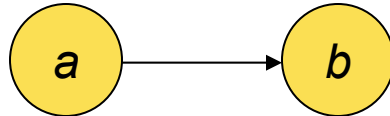
² S. Koenig and M. Likhachev, 2002.

D* Lite: Functionality

- Similar functionality to D*, simpler algorithm
- Maintains two estimates of costs per node
 - g :  the objective function value
based on what we know
 - rhs : one-step lookahead  of the objective function value
based on what we know
- Defines “consistency”
 - Consistent $\Rightarrow g = rhs$
 - Inconsistent $\Rightarrow g \neq rhs$
- Inconsistent nodes go on the priority queue (“open list”) for processing
- No Backpointers

D* Lite: Functionality

- If there's a directed edge from node a to node b in the graph, then we say that " b is a **successor** of a " and " a is a **predecessor** of b "



- The *rhs* value of a node is computed based on the g values of its successors in the graph and the transition costs of the edges to those successors. (Starting with the goal which has a g value of 0)

$$rhs(u) = \min_{s' \in Succ(u)} (c(u, s') + g(s'));$$

- The key/priority of a node on the open list is the minimum of its g and *rhs* values plus a focusing heuristic, h .

$$[\underbrace{\min(g(s), rhs(s)) + h(s_{start}, s)}_{\text{Primary key}}; \underbrace{\min(g(s), rhs(s))}_{\text{Secondary key (used for tie-breaking)}}]$$

Primary key

Secondary key (used for tie-breaking)

D* Lite: Algorithm (1/2)

procedure Main()

➔ Initialize();
➔ ComputeShortestPath();
➔ while ($s_{start} \neq s_{goal}$)
 /* if ($g(s_{start}) = \infty$) then there is no known path */
 ➔ $s_{start} = \arg \min_{s' \in Succ(s_{start})} (c(s_{start}, s') + g(s'))$;
 ➔ Move to s_{start} ;
 ➔ Scan graph for changed edge costs;
 if any edge costs changed
 for all directed edges (u, v) with changed edge costs
 ➔ Update the edge cost $c(u, v)$;
 ➔ UpdateVertex(u);
 for all $s \in U$
 ➔ U.Update(s , CalculateKey(s));
 ➔ ComputeShortestPath();

Repeat until the robot reaches the goal.

procedure Initialize()

$U = \emptyset$;
for all $s \in S$ $rhs(s) = g(s) = \infty$;
 $rhs(s_{goal}) = 0$;
U.Insert(s_{goal} , CalculateKey(s_{goal}));

D* Lite: Algorithm (2/2)

procedure ComputeShortestPath()

while ($U.TopKey() < CalculateKey(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start})$)

→ $u = U.Pop();$
→ if ($g(u) > rhs(u)$)
→ $g(u) = rhs(u);$
→ for all $s \in Pred(u)$ UpdateVertex(s);
→ else
→ $g(u) = \infty;$
→ for all $s \in Pred(u) \cup \{u\}$ UpdateVertex(s);

If node is under-consistent
- Make it over-consistent
- Propagate changed cost to neighboring nodes

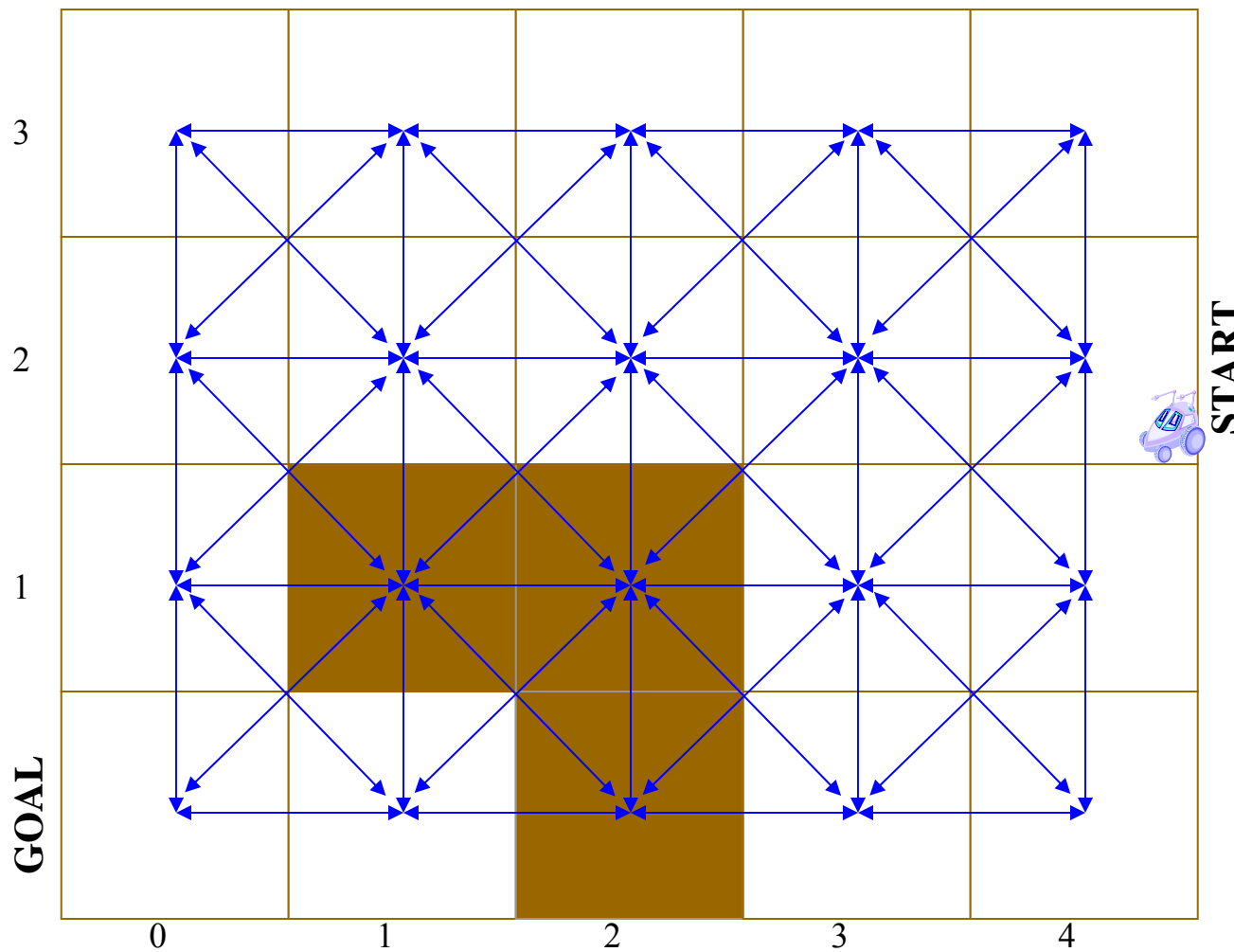
procedure UpdateVertex()

if ($u \neq s_{goal}$) $rhs(u) = \min_{s' \in Succ(u)} (c(u, s') + g(s'));$
if ($u \in U$) $U.Remove(u);$
if ($g(u) \neq rhs(u)$) $U.Insert(u, CalculateKey(u));$

procedure CalculateKey(s)

return [$\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))$];

D* Lite: Example



Assume there's an 8-connected graph superimposed on the grid. Each edge of the graph is bi-directional. E.g., there's a directed edge from (2,1) to (2,0) and another edge from (2,0) to (2,1)

Legend



Free



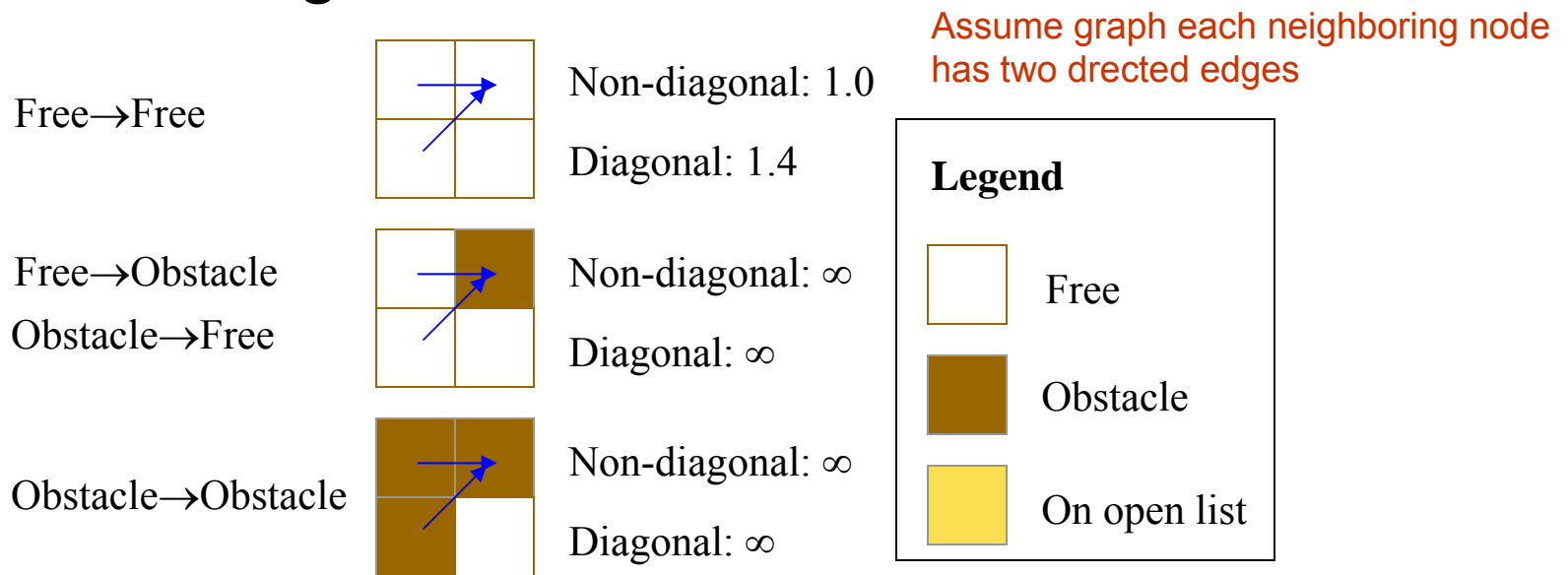
Obstacle



On open list

D* Lite Example: Notes

- The example on the next several slides assumes the following transition costs:



- Also, the example does not use a focusing heuristic ($h = 0$)

D* Lite: Planning (1)

3	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
2	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
1	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
GOAL	g: ∞ rhs: 0	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
	0	1	2	3	4



START

Initialization

Set the *rhs* value of the goal to 0 and all other *rhs* and *g* values to ∞ .

Legend



Free



Obstacle



On open list

D* Lite: Planning (2)

3	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
2	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
1	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
GOAL	g: ∞ rhs: 0	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞	g: ∞ rhs: ∞
	0	1	2	3	4



START

Initialization

Put the goal on the open list because it is inconsistent .

Legend



Free

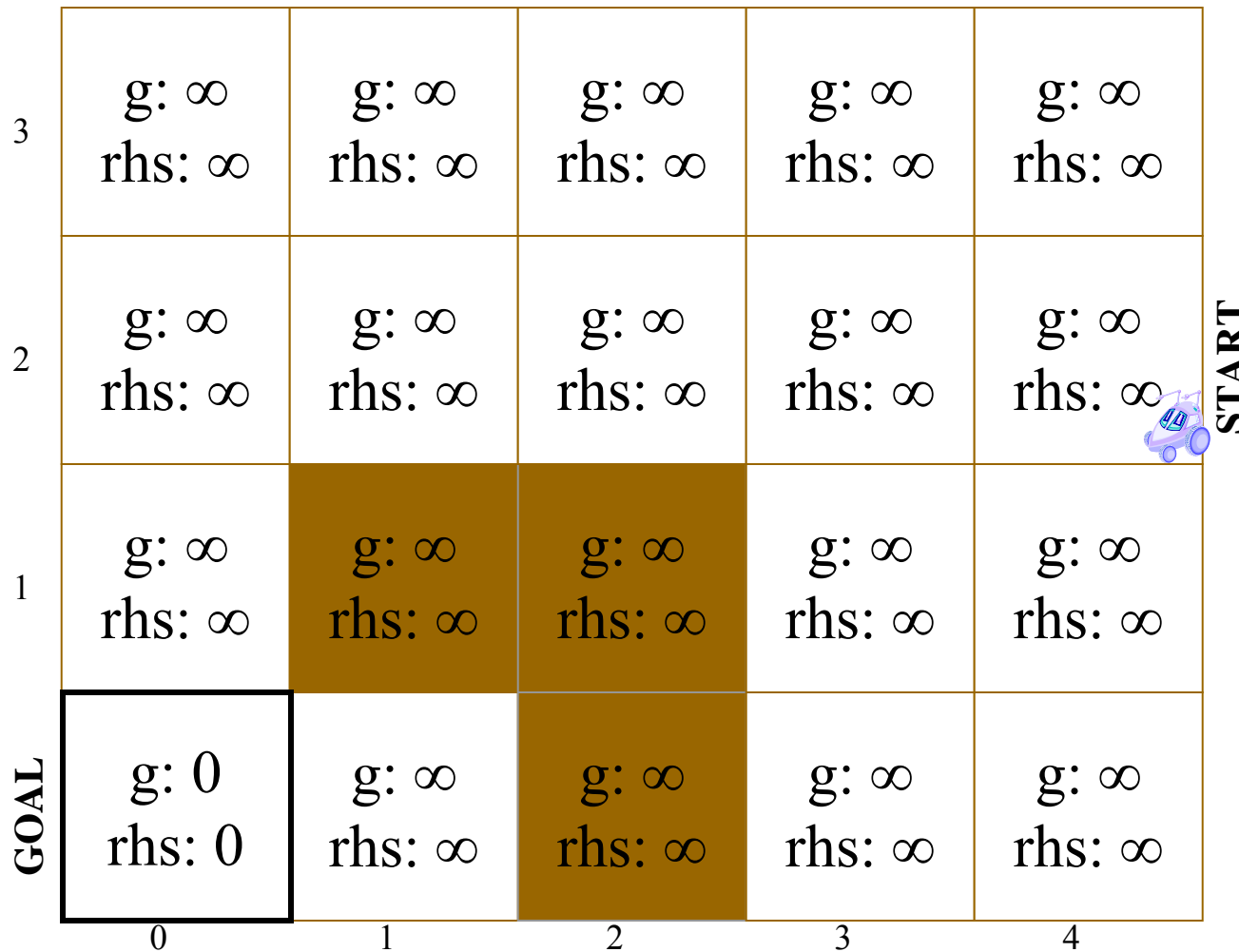


Obstacle

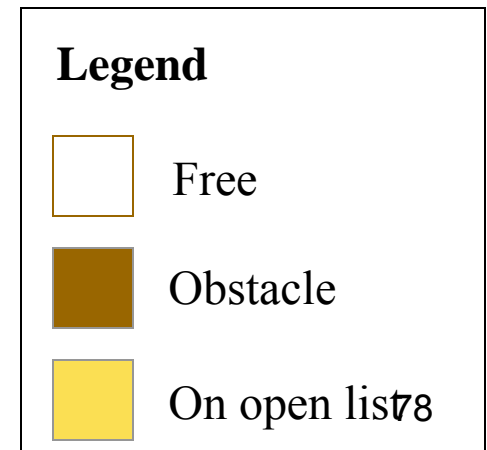


On open list

D* Lite: Planning (3)

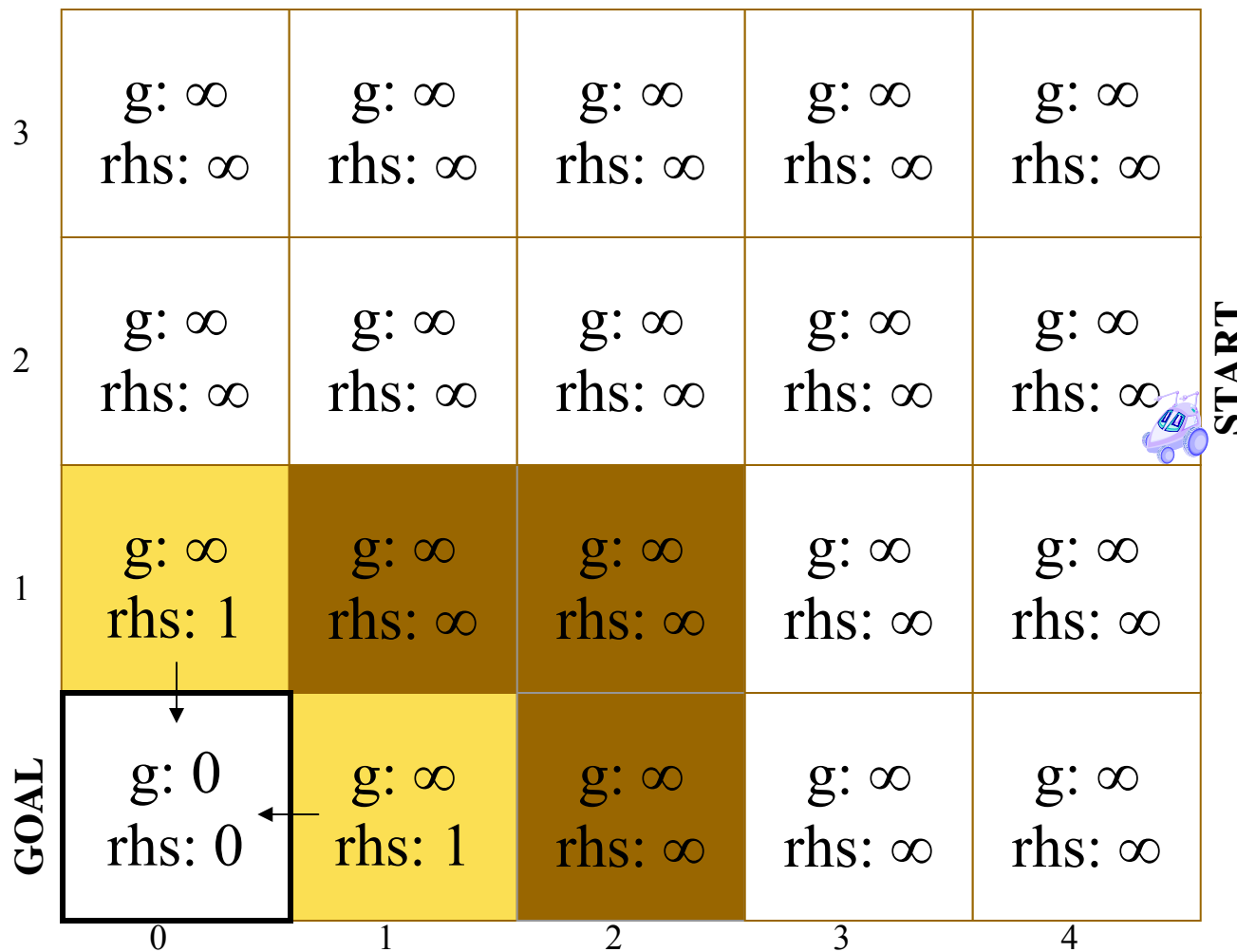


ComputeShortestPath
 Pop minimum item off
 open list (goal)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.






D* Lite: Planning (4)

The small arrows indicate which node is used to compute the *rhs* value. E.g., the *rhs* value of (0,1) is computed using the *g* value of (0,0) and the transition cost from (1,0) to (0,0): $1 = 0 + 1$

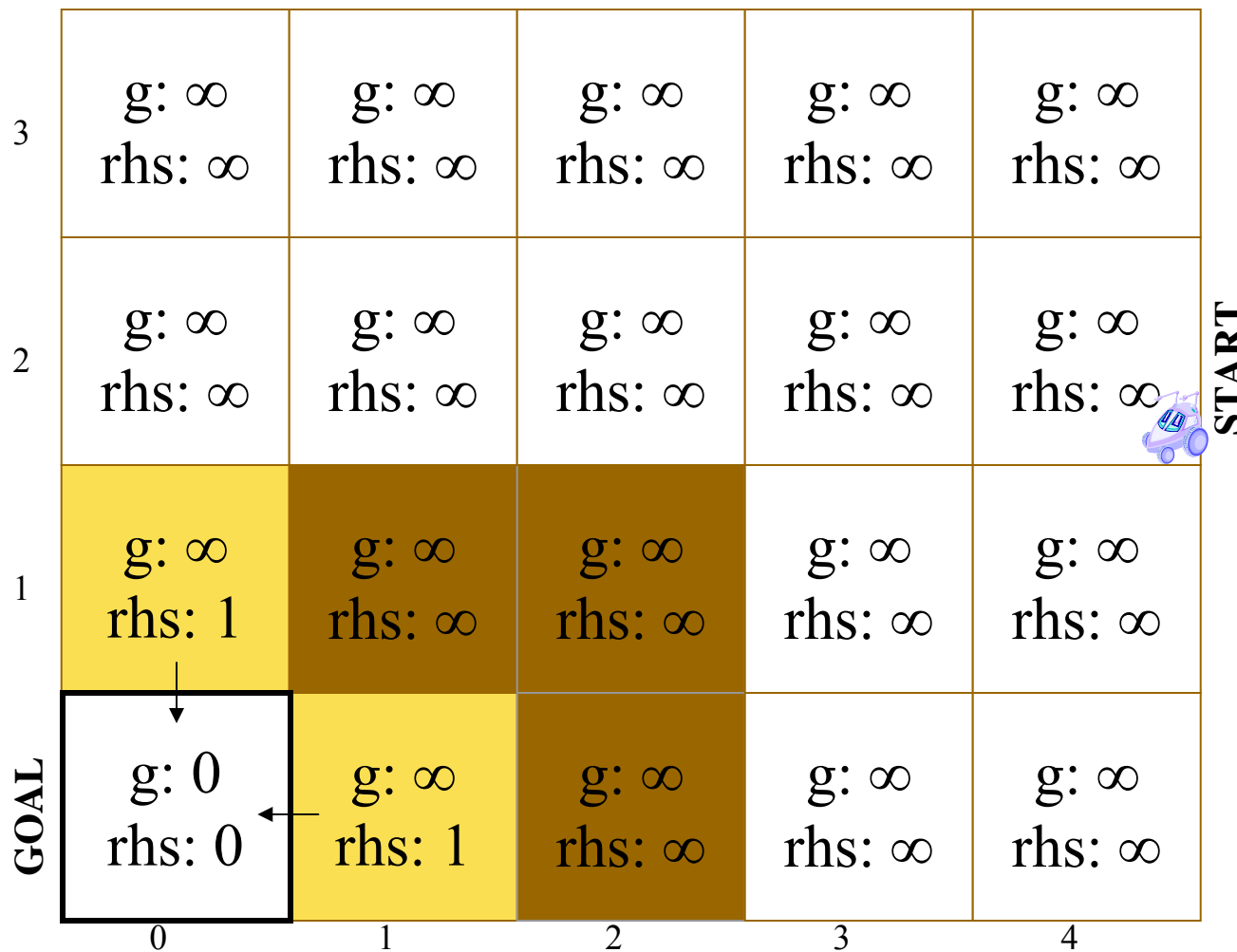


ComputeShortestPath
Expand the popped node i.e, call *UpdateVertex()* on all its predecessors in the graph. This computes *rhs* values for the predecessors and puts them on the open list if they become inconsistent.

Legend

-  Free
-  Obstacle
-  On open list

D* Lite: Planning (4b)

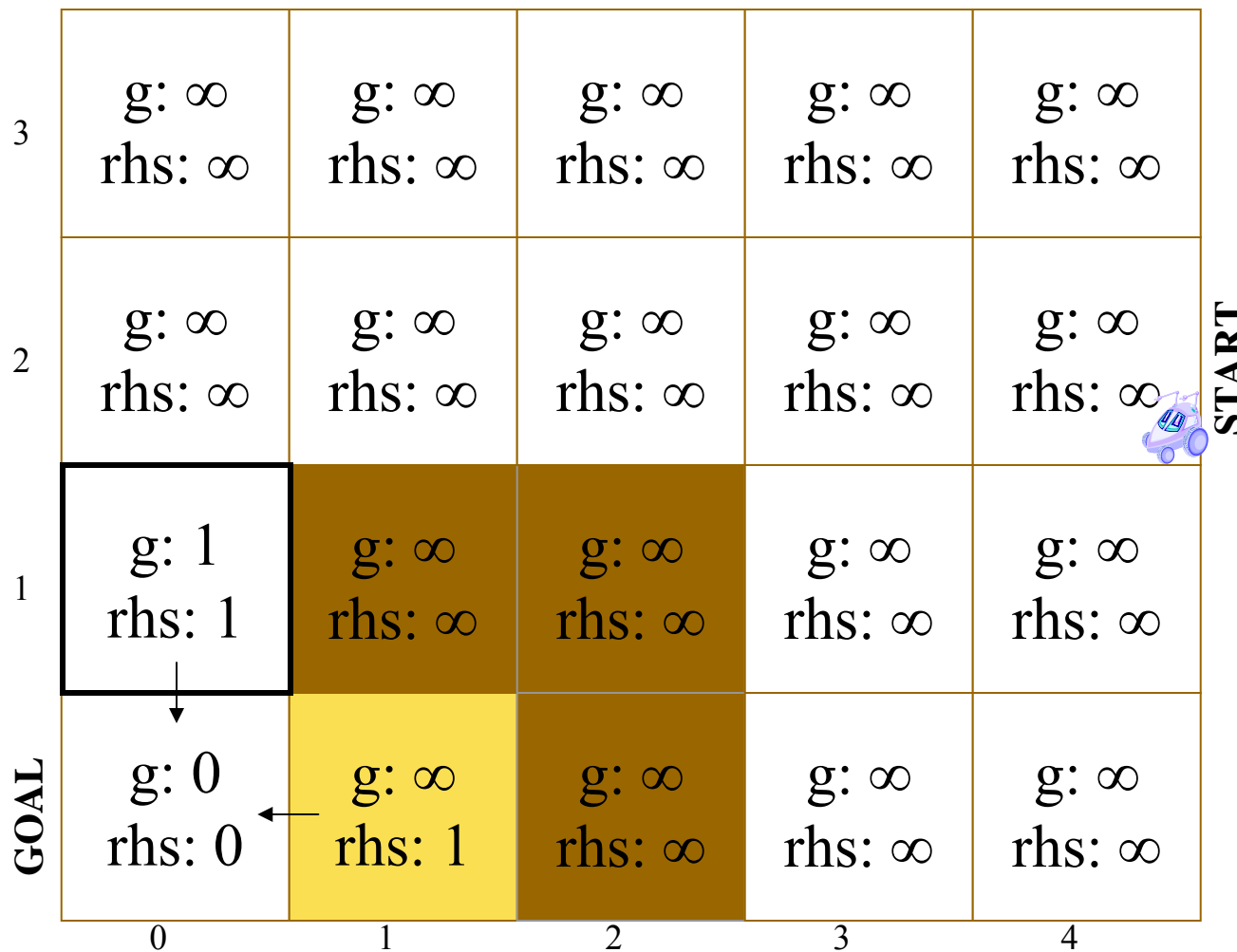


ComputeShortestPath
 Note that the *rhs* value for (1,1) is still ∞ because we assumed that the transition cost is ∞ and that $\infty + \text{anything}$ is still ∞ .

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (5)

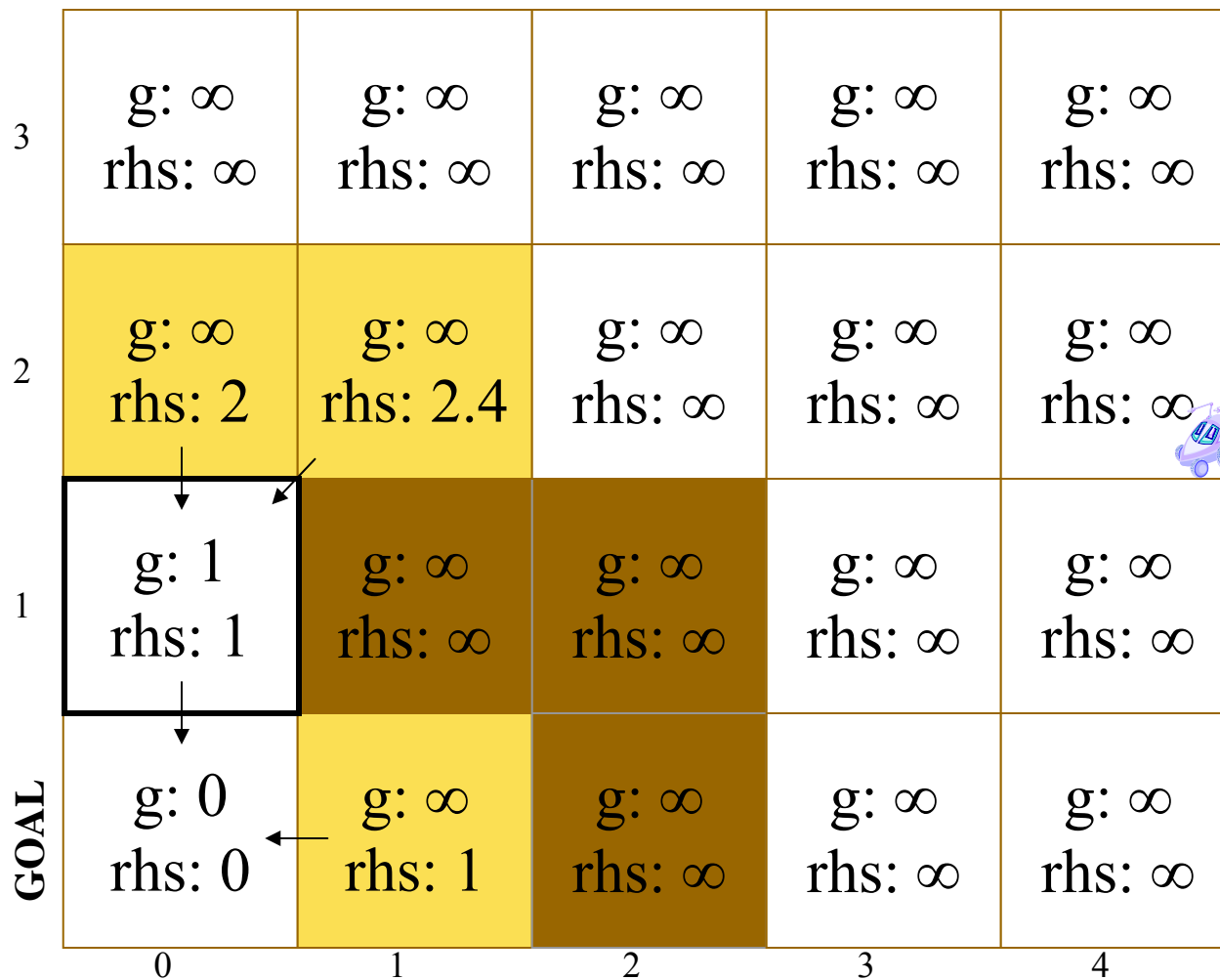


ComputeShortestPath
 Pop minimum item off
 open list - (0,1)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$..

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (6)



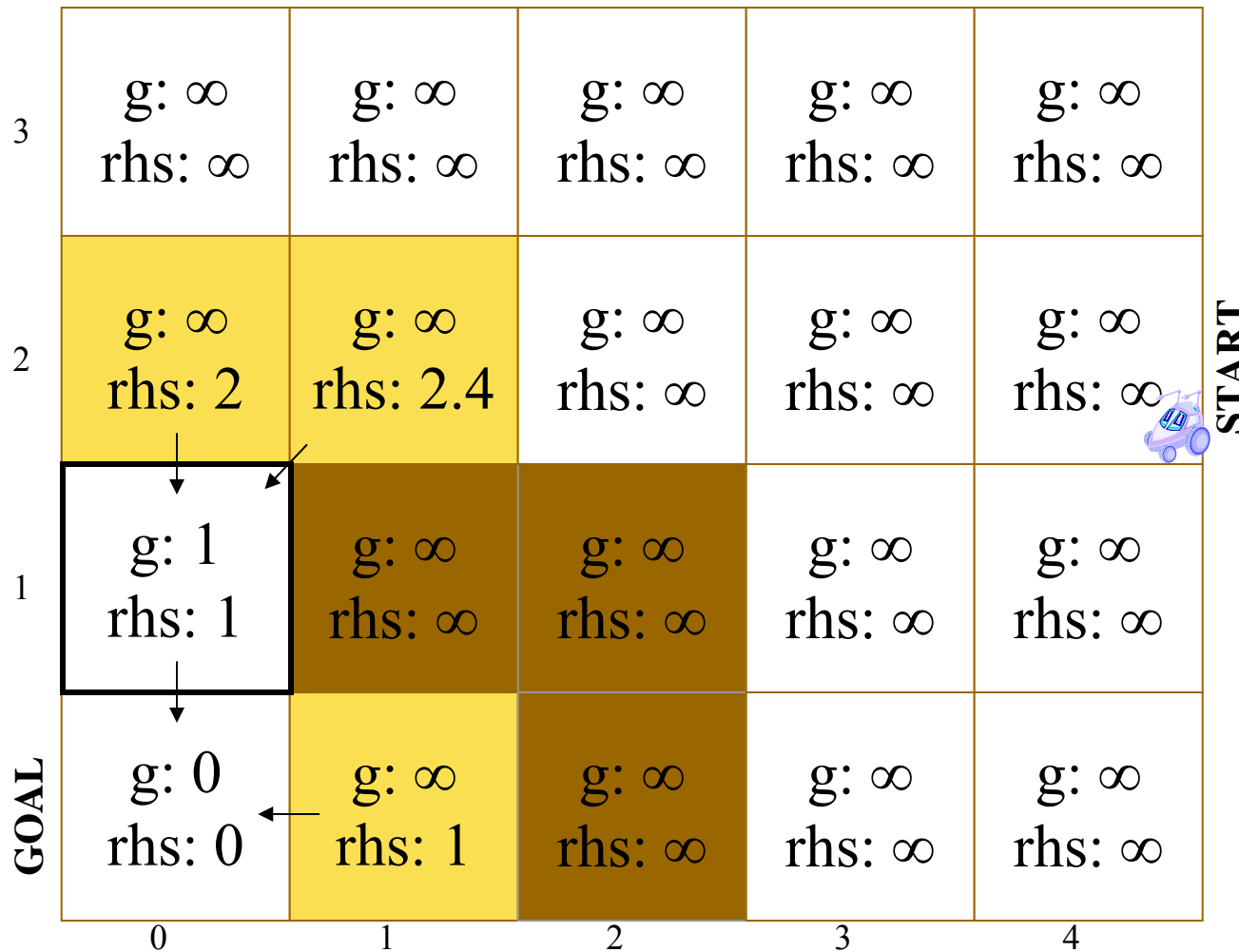
START

ComputeShortestPath
 Expand the popped node
 – call *UpdateVertex()* on all predecessors in the graph. This computes *rhs* values for the predecessors and puts them on the open list if they become inconsistent.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (6b)



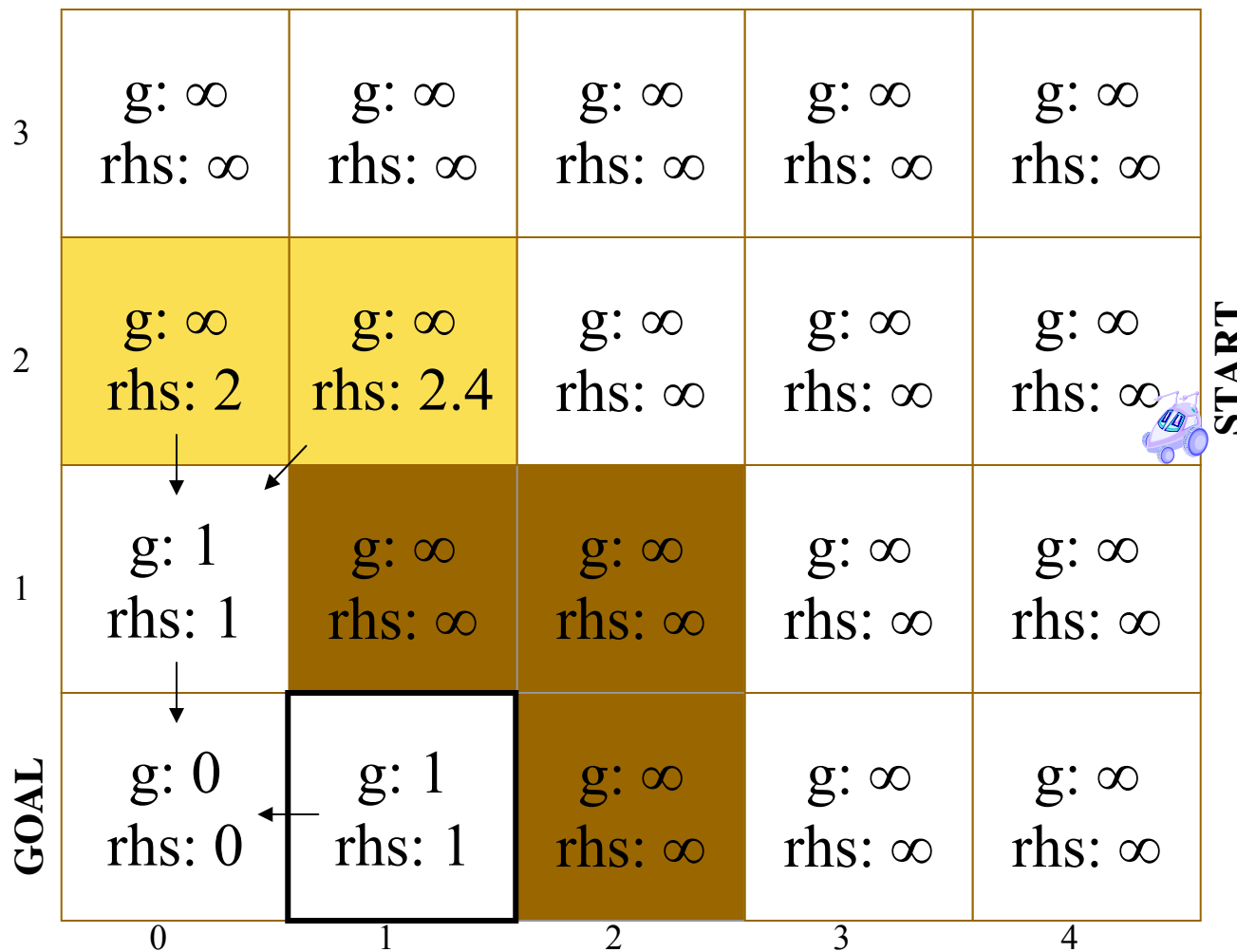
ComputeShortestPath

Note that the *rhs* values of some predecessors, e.g., (0,0) and (1,1) do not change. As such, they do not become inconsistent and are not put on the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (7)

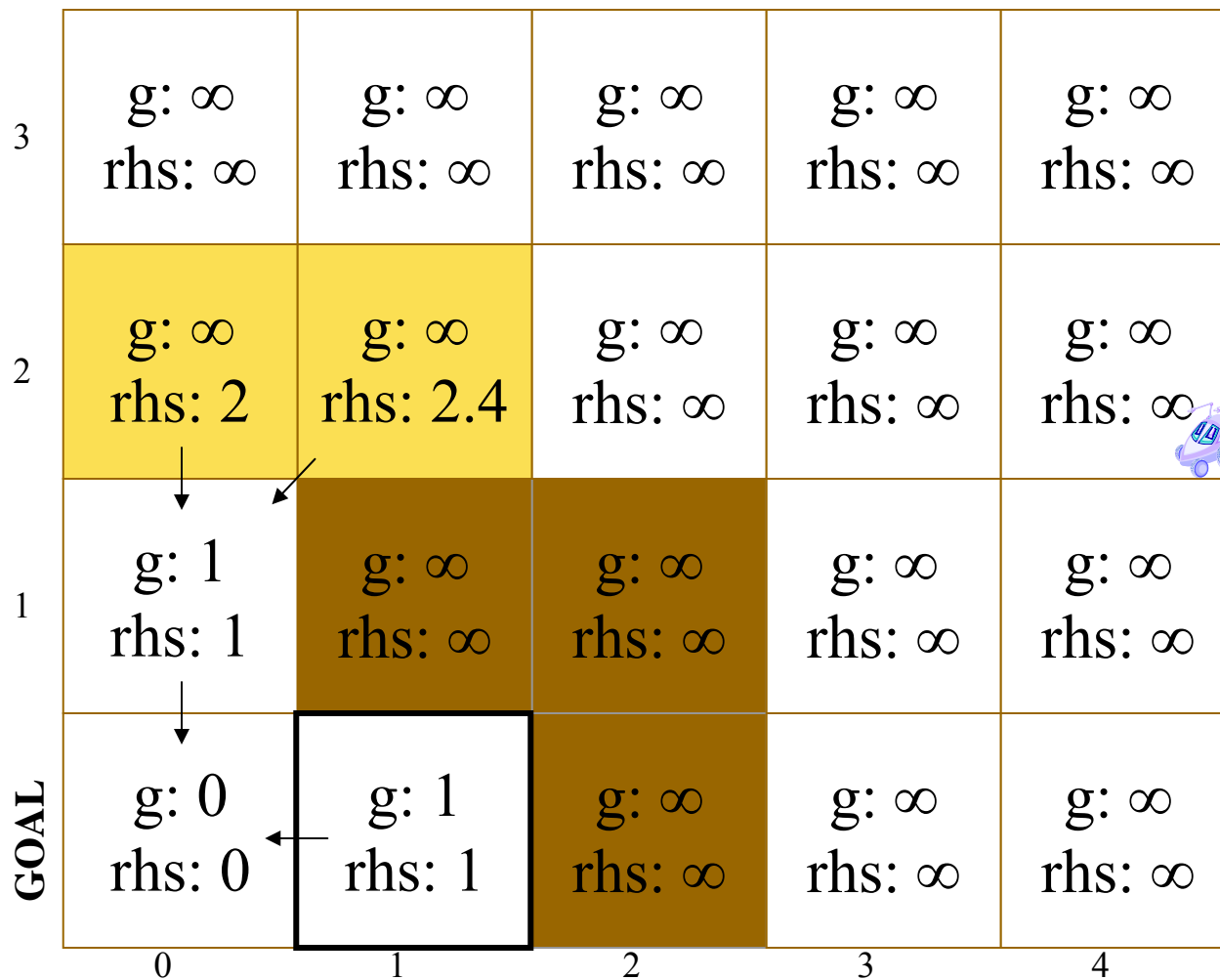


ComputeShortestPath
 Pop minimum item off
 open list - (1,0)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (7b)



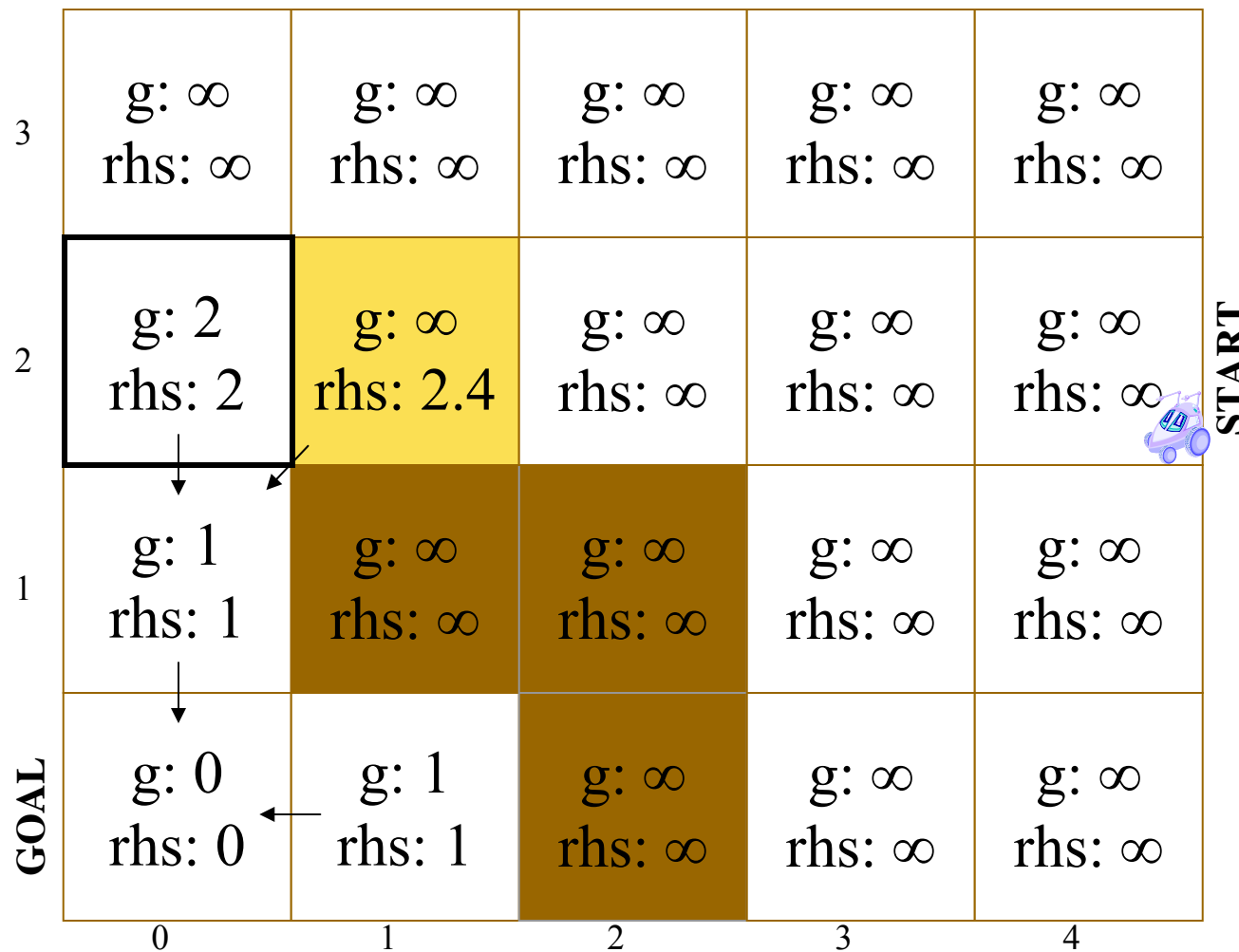
START

ComputeShortestPath
Expand the popped node by calling *UpdateVertex()* on its predecessors. This time, nothing goes on the open list because the *rhs* values of its predecessors cannot be reduced and so nothing becomes inconsistent.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (8)

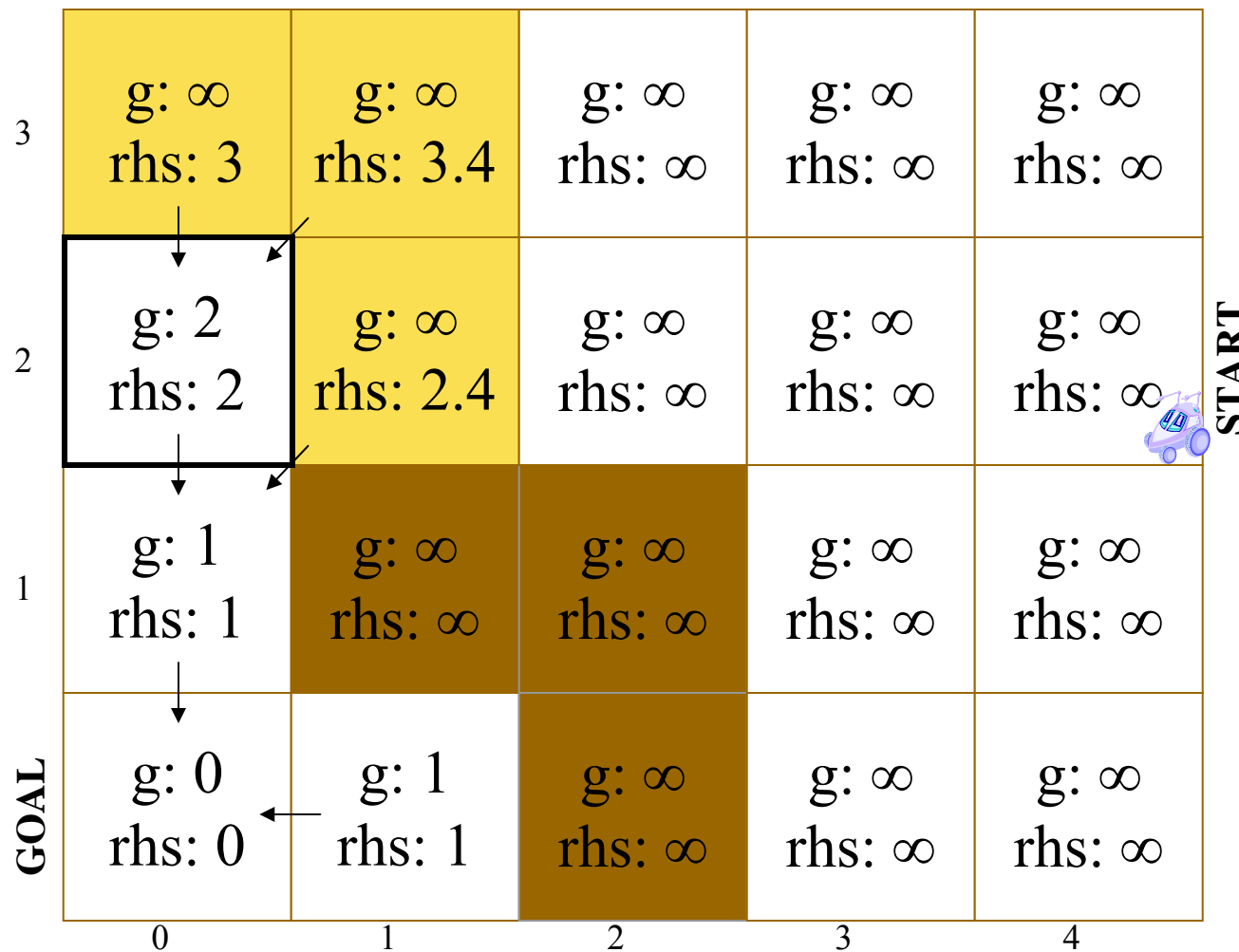


ComputeShortestPath
 Pop minimum item off
 open list - (0,2)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$..

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (9)

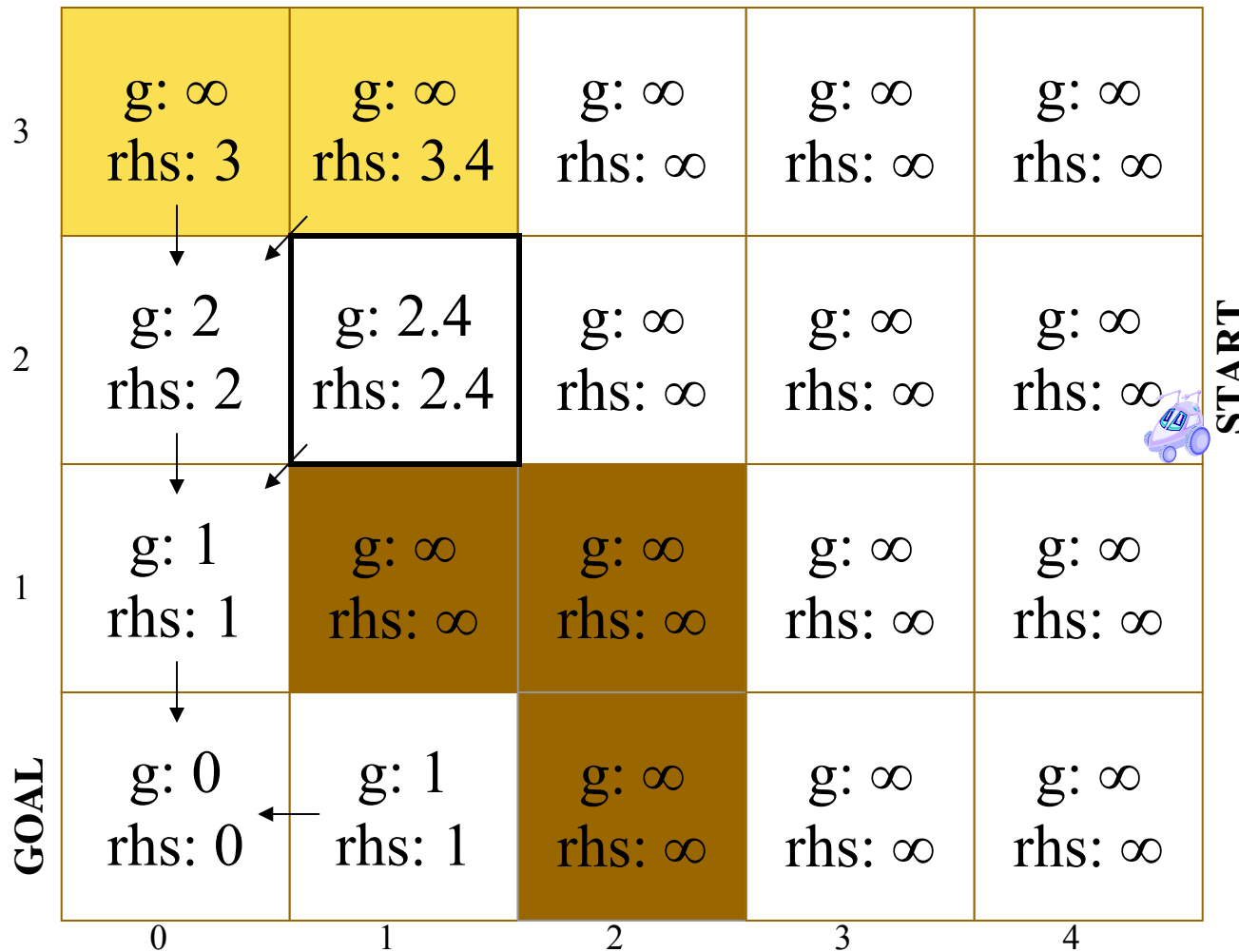


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (10)

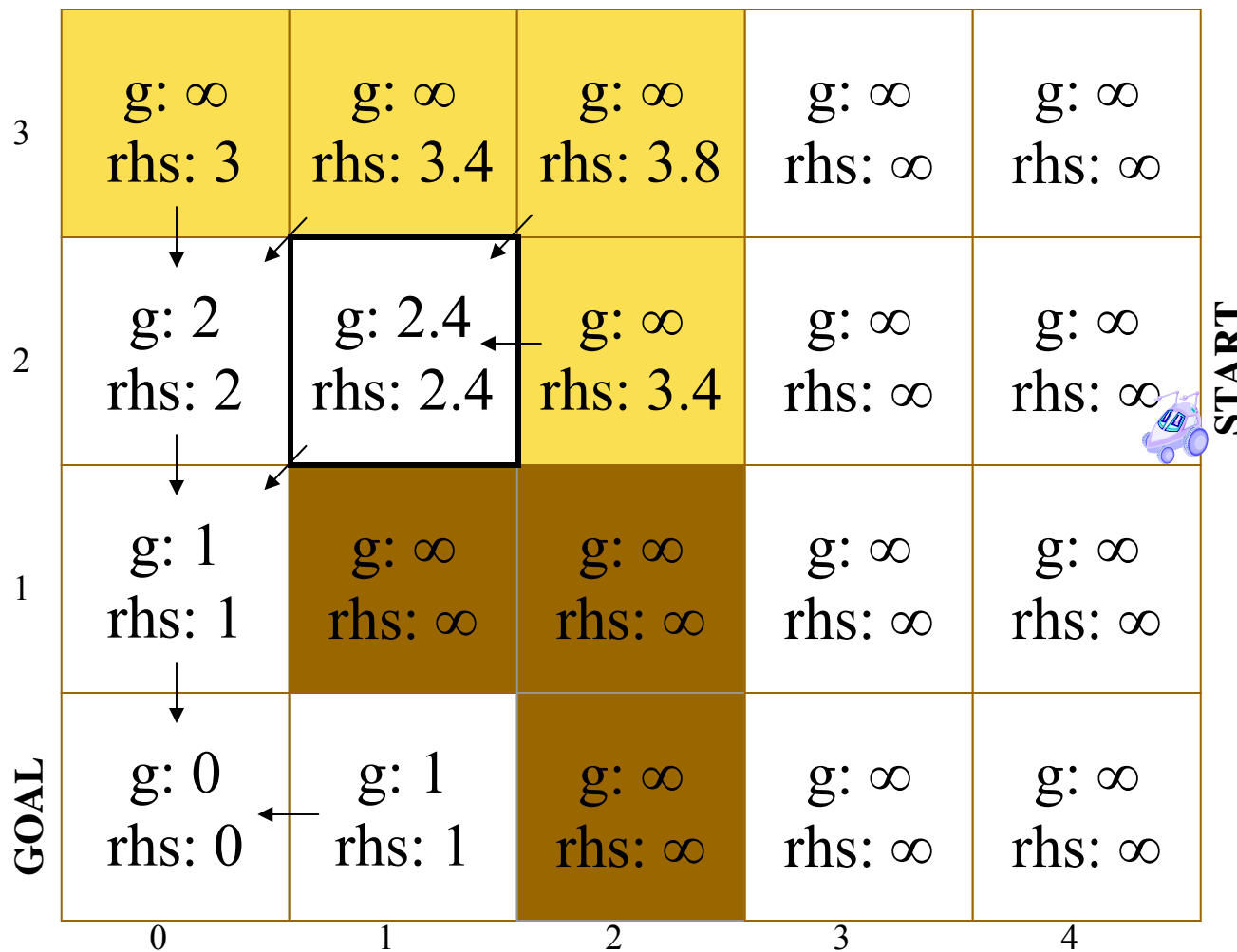


ComputeShortestPath
 Pop minimum item off
 open list - (1,2)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (11)

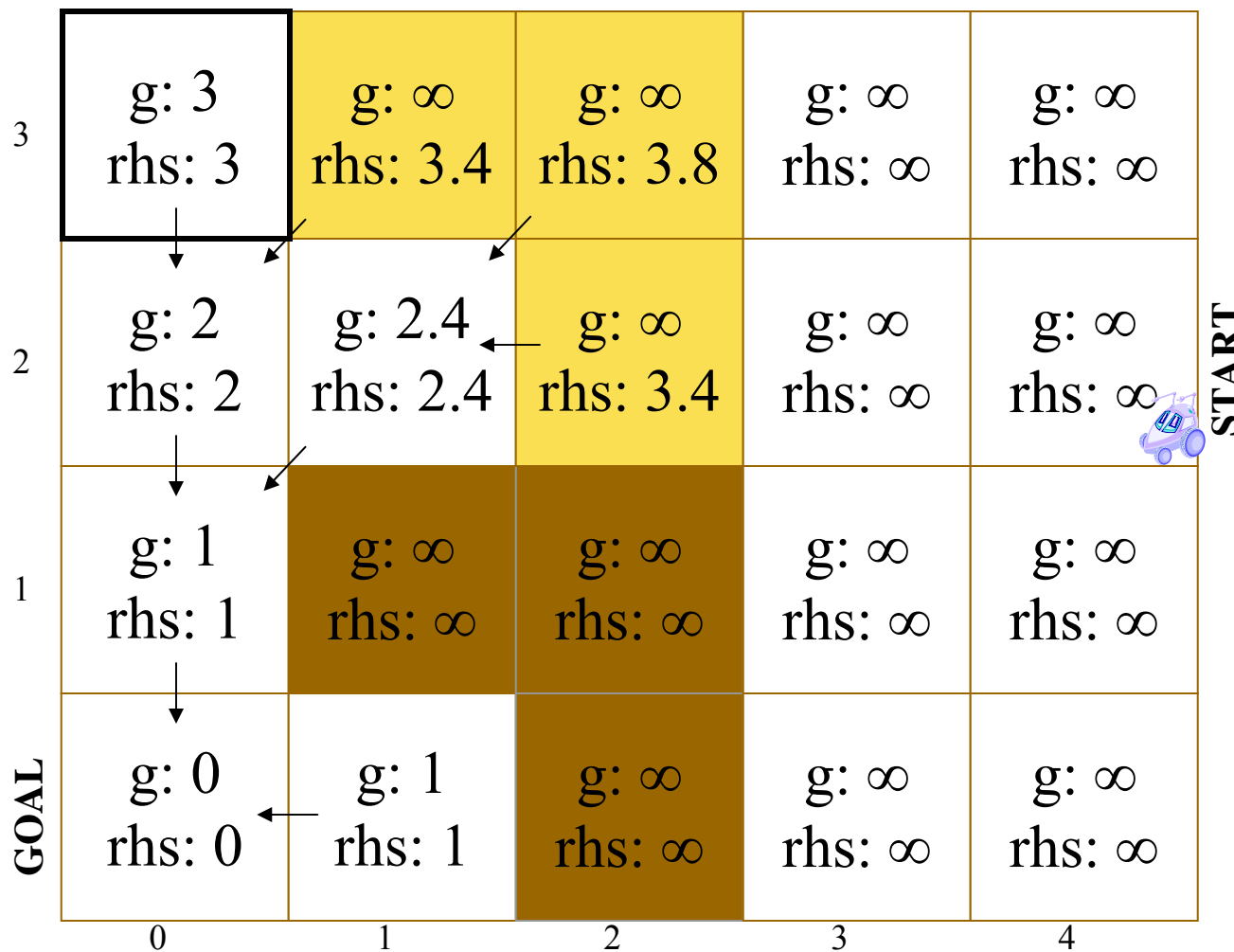


ComputeShortestPath
Expand the popped node and put predecessors that become inconsistent onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (12)

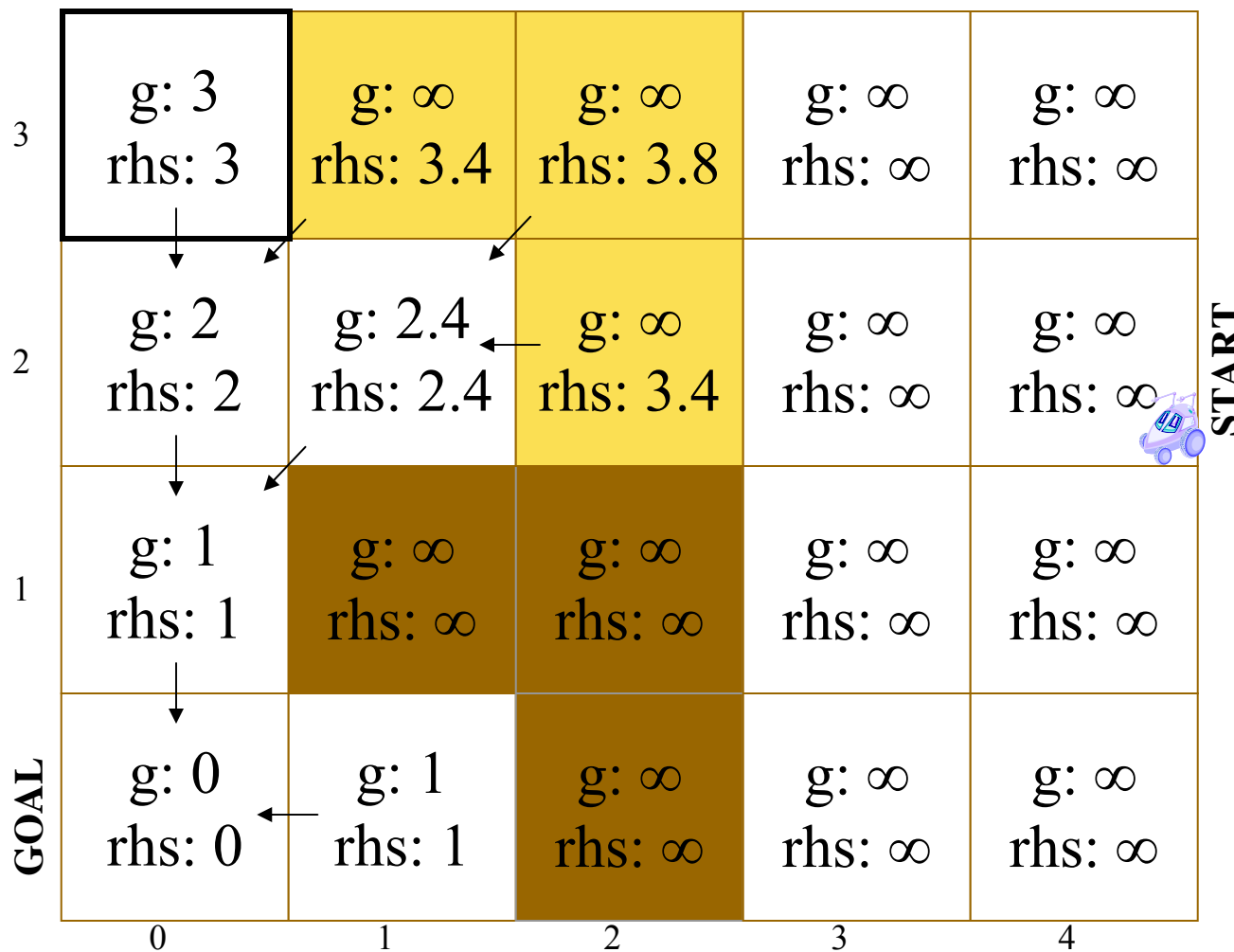


ComputeShortestPath
 Pop minimum item off
 open list - (0,3)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (12b)

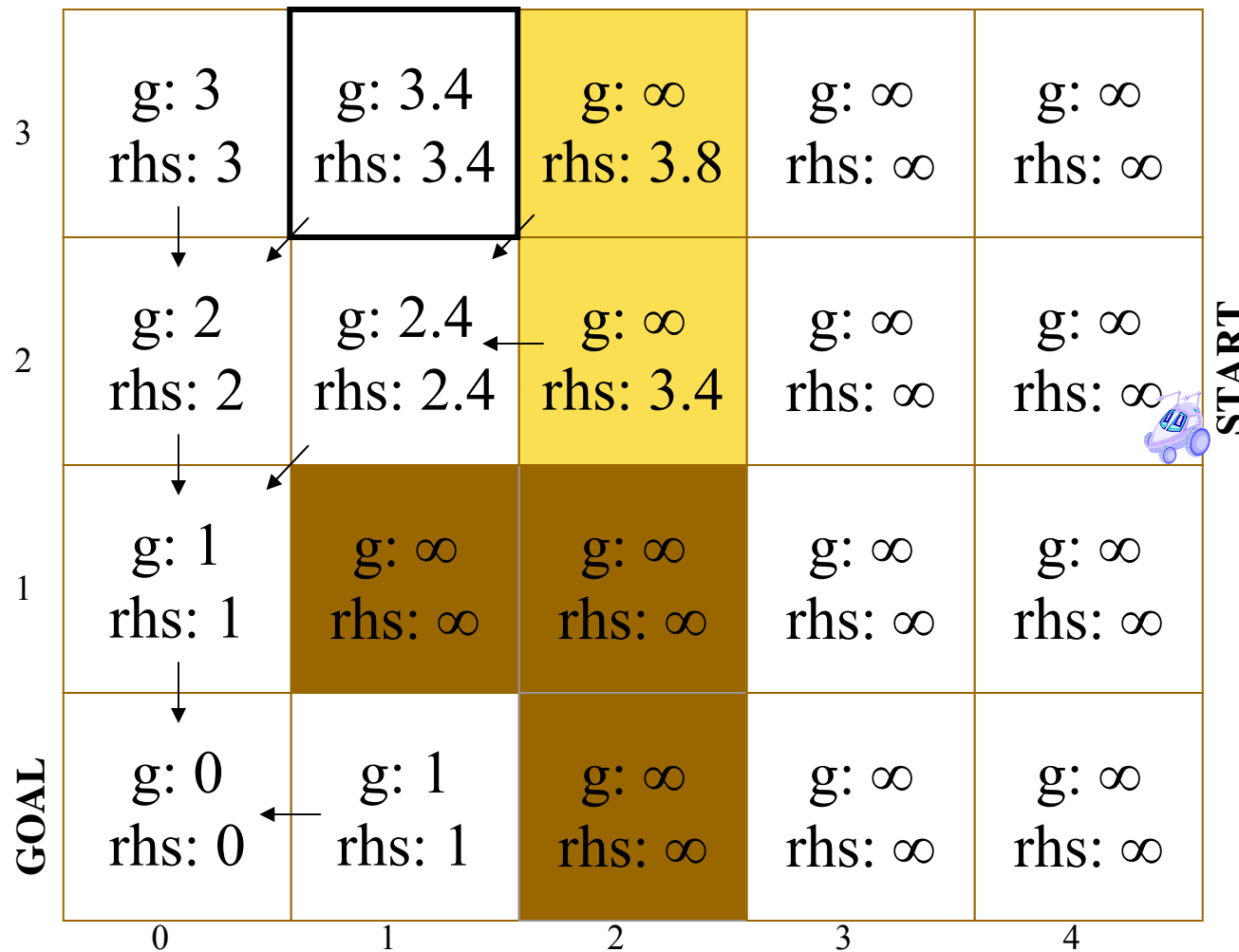


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (13)

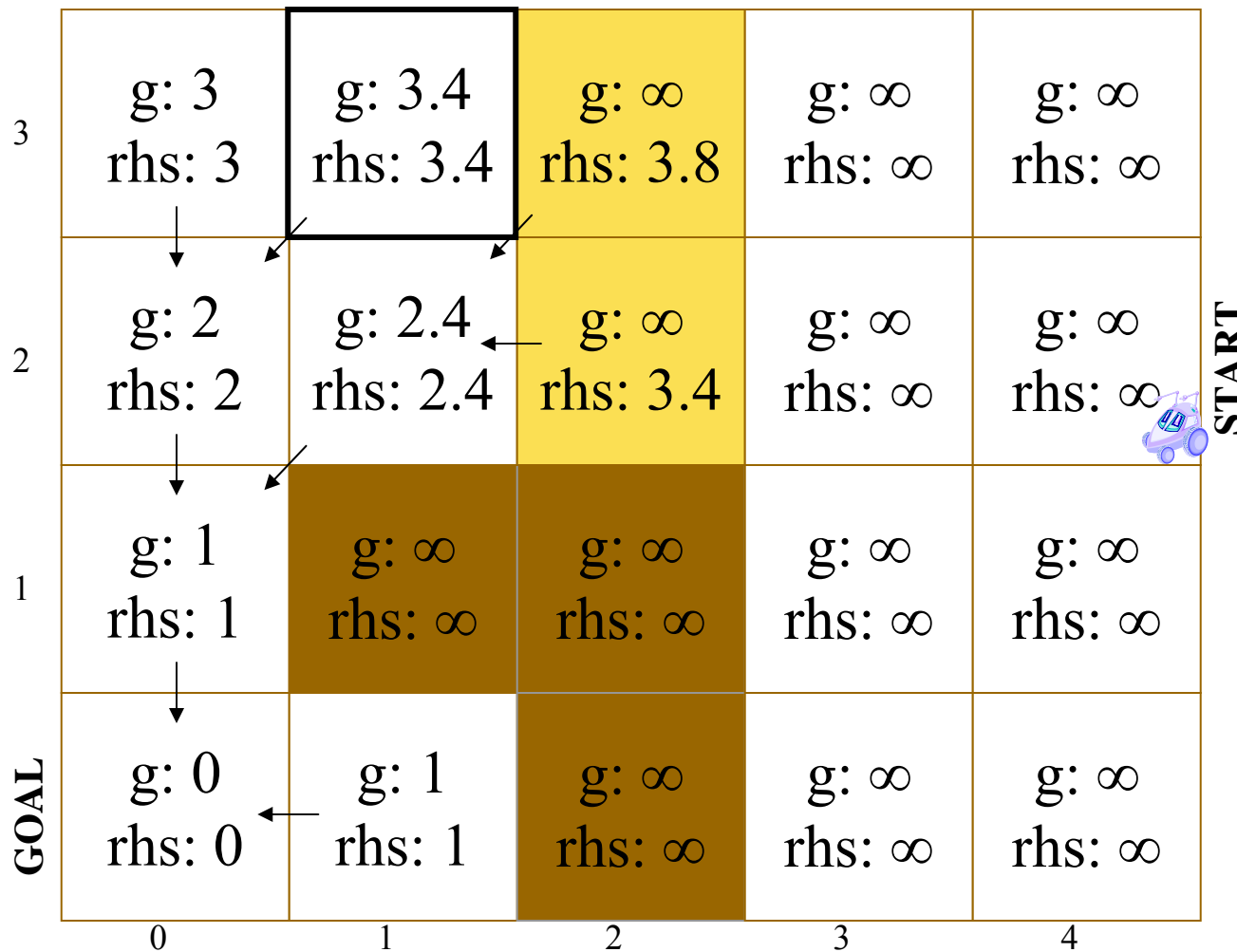


ComputeShortestPath
 Pop minimum item off
 open list - (1,3)
 It's over-consistent
 ($g > rhs$) so set $g=rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (13b)

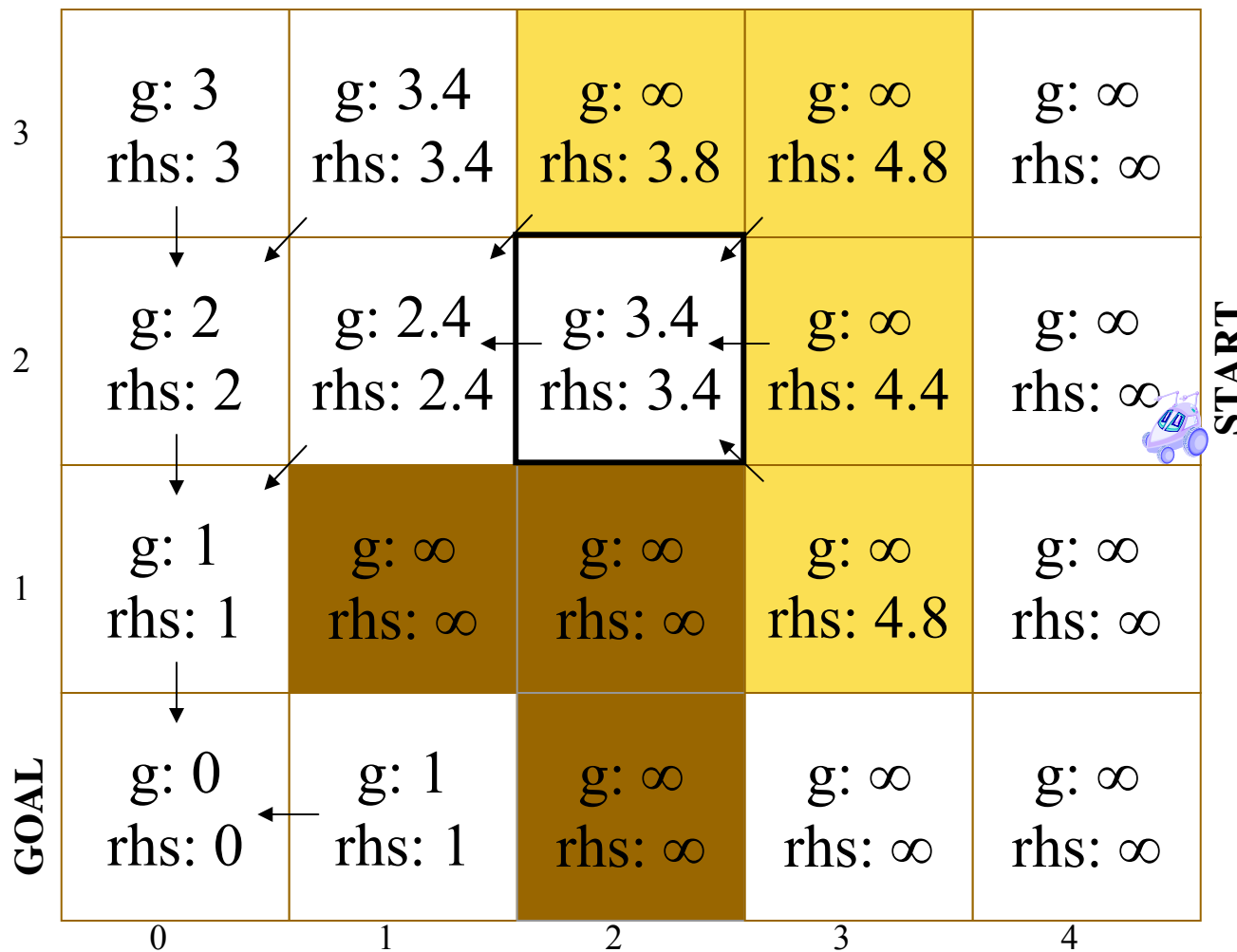


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (15)

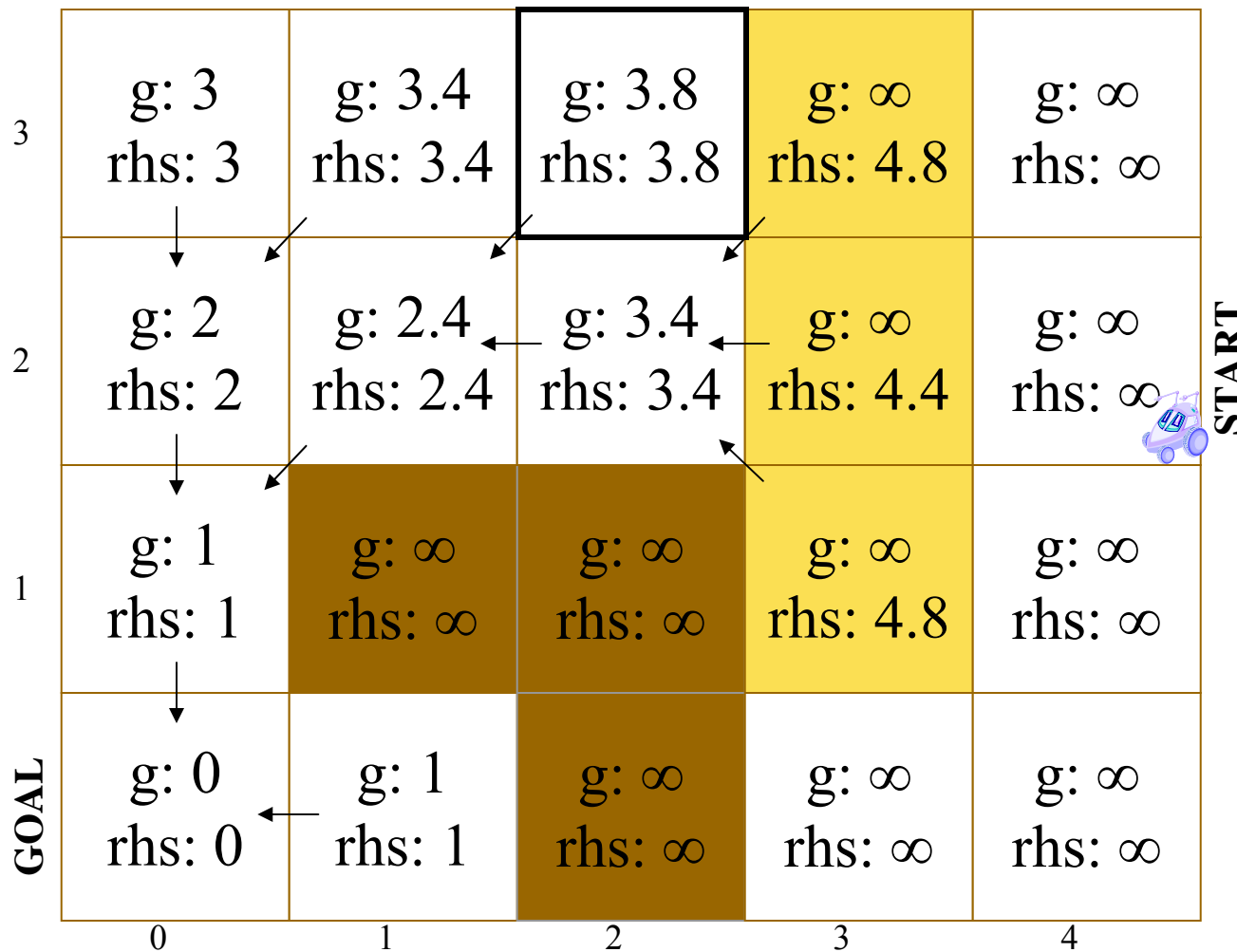


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (16)

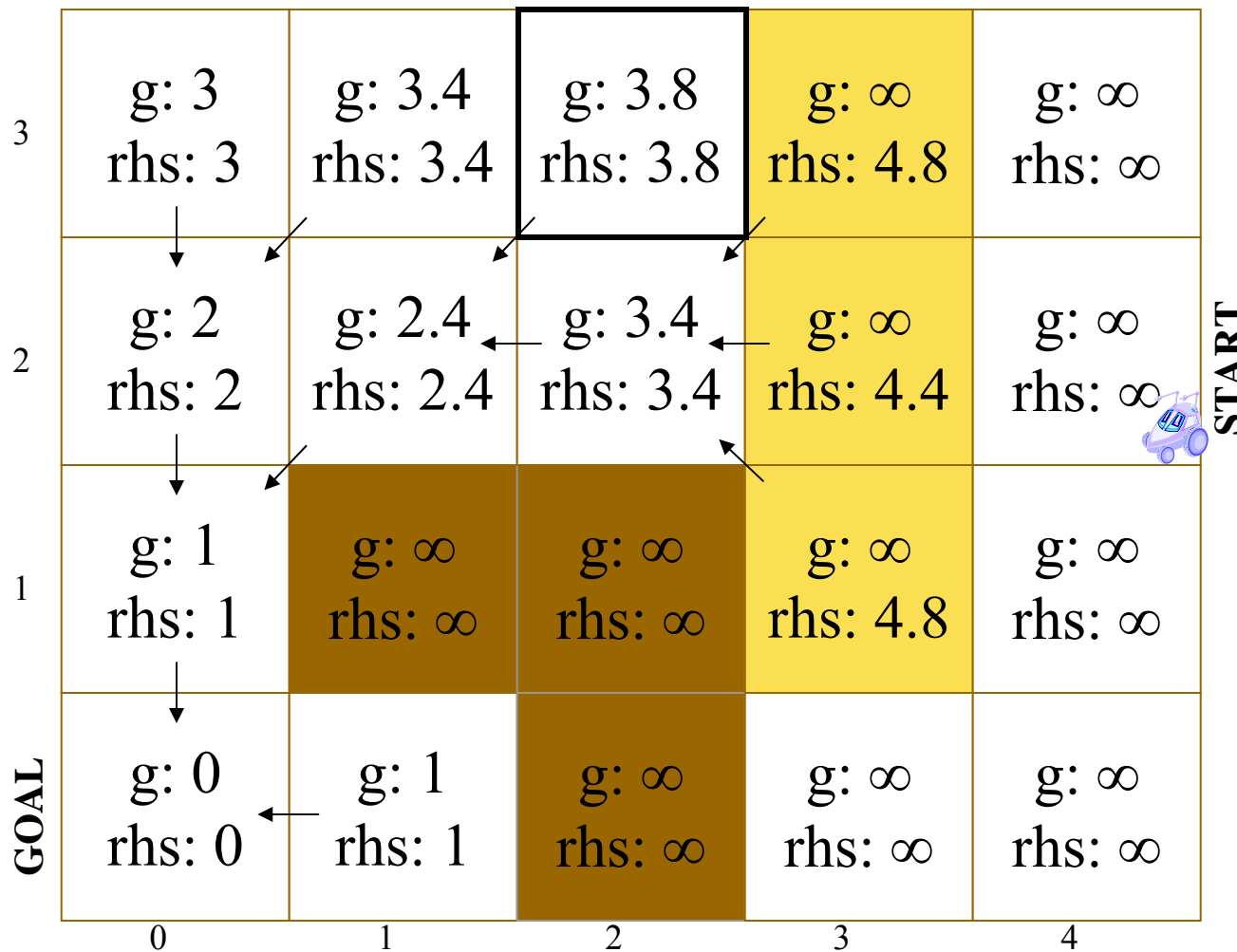


ComputeShortestPath
 Pop minimum item off
 open list - (2,3)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (16b)

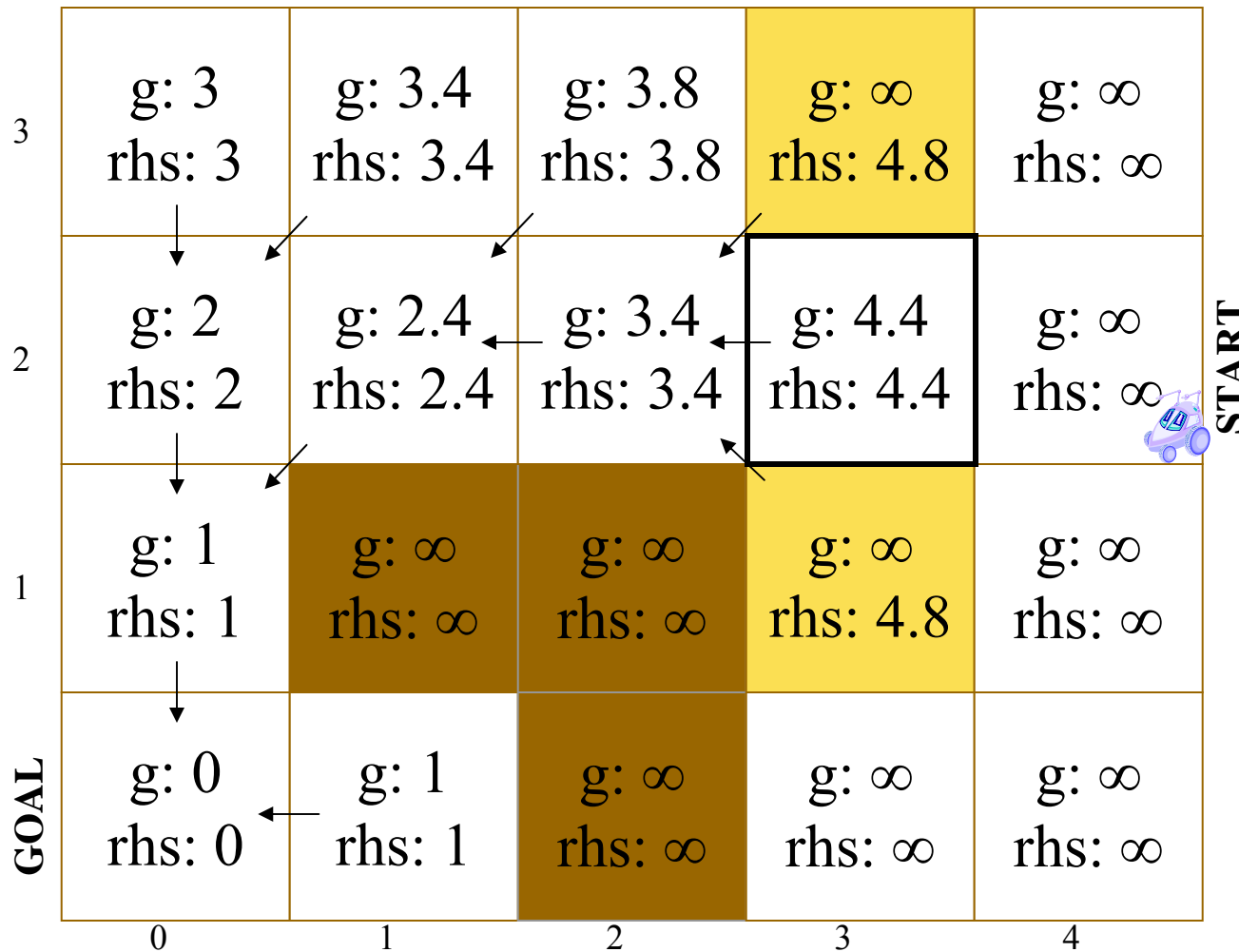


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (17)

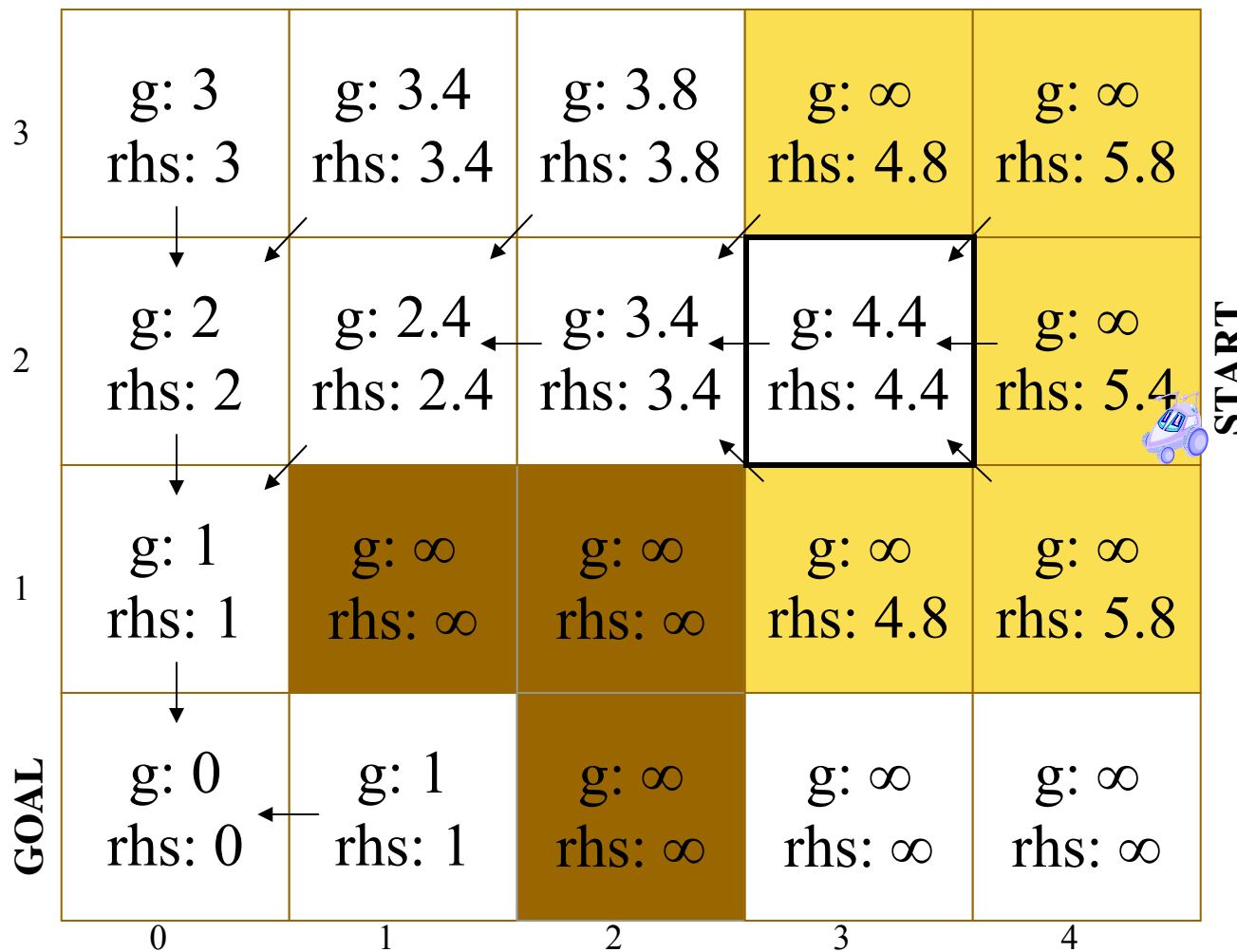


ComputeShortestPath
 Pop minimum item off
 open list - (3,2)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (18)

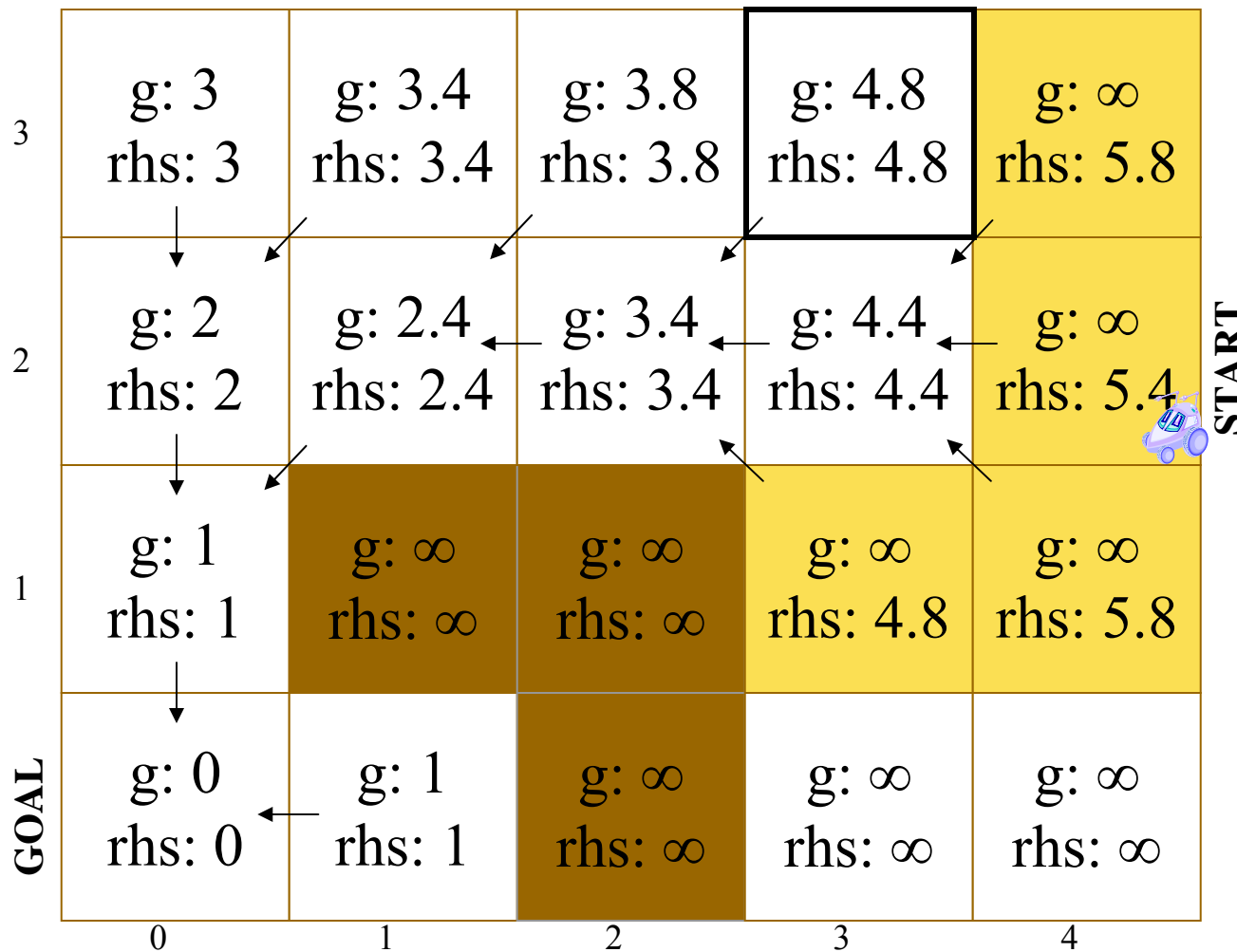


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent onto the open list. Note that the start state is now on the open list, but the search does not end here.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (19)

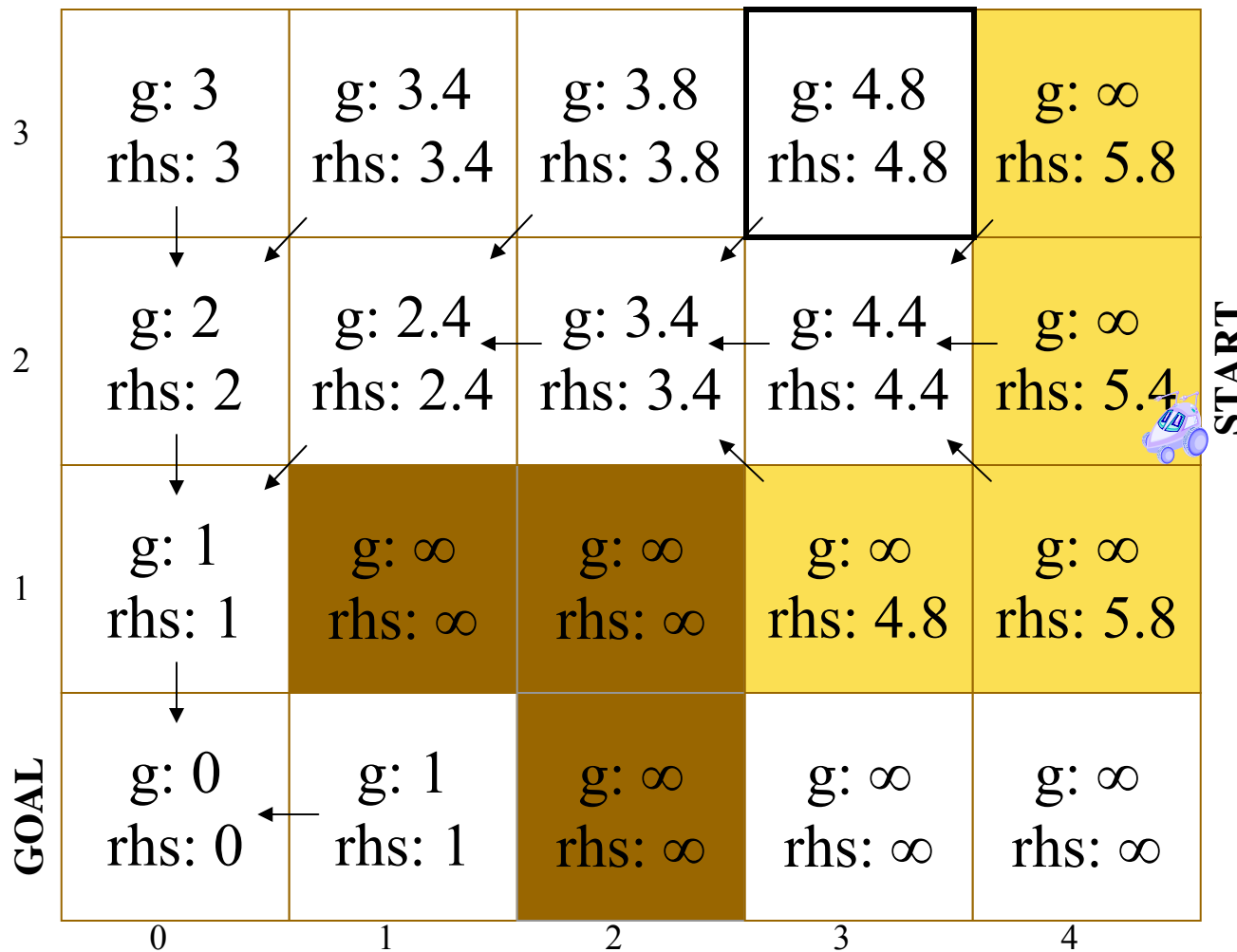


ComputeShortestPath
 Pop minimum item off
 open list - (3,3)
 It's over-consistent
 ($g > rhs$) so set $g=rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (19b)

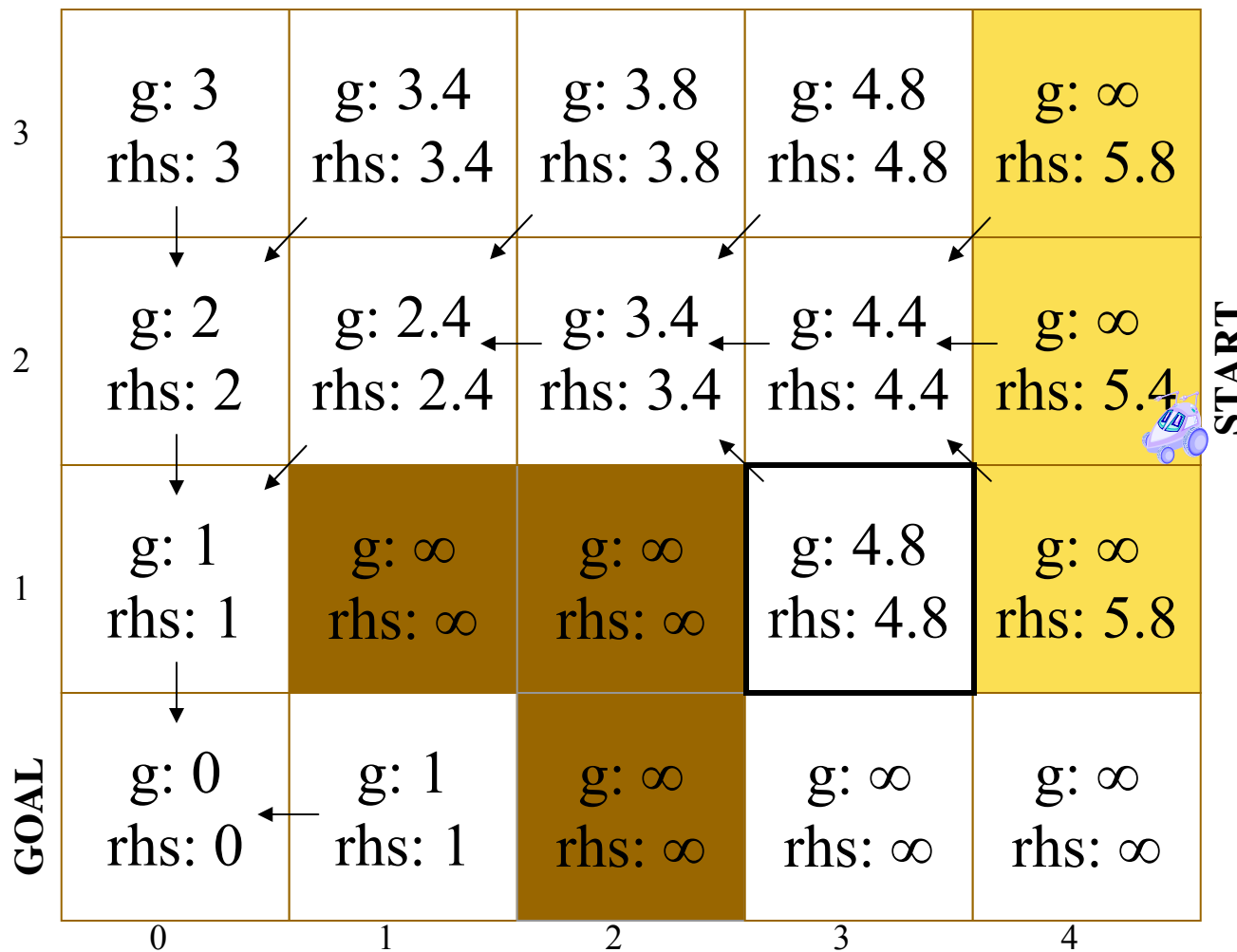


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (20)

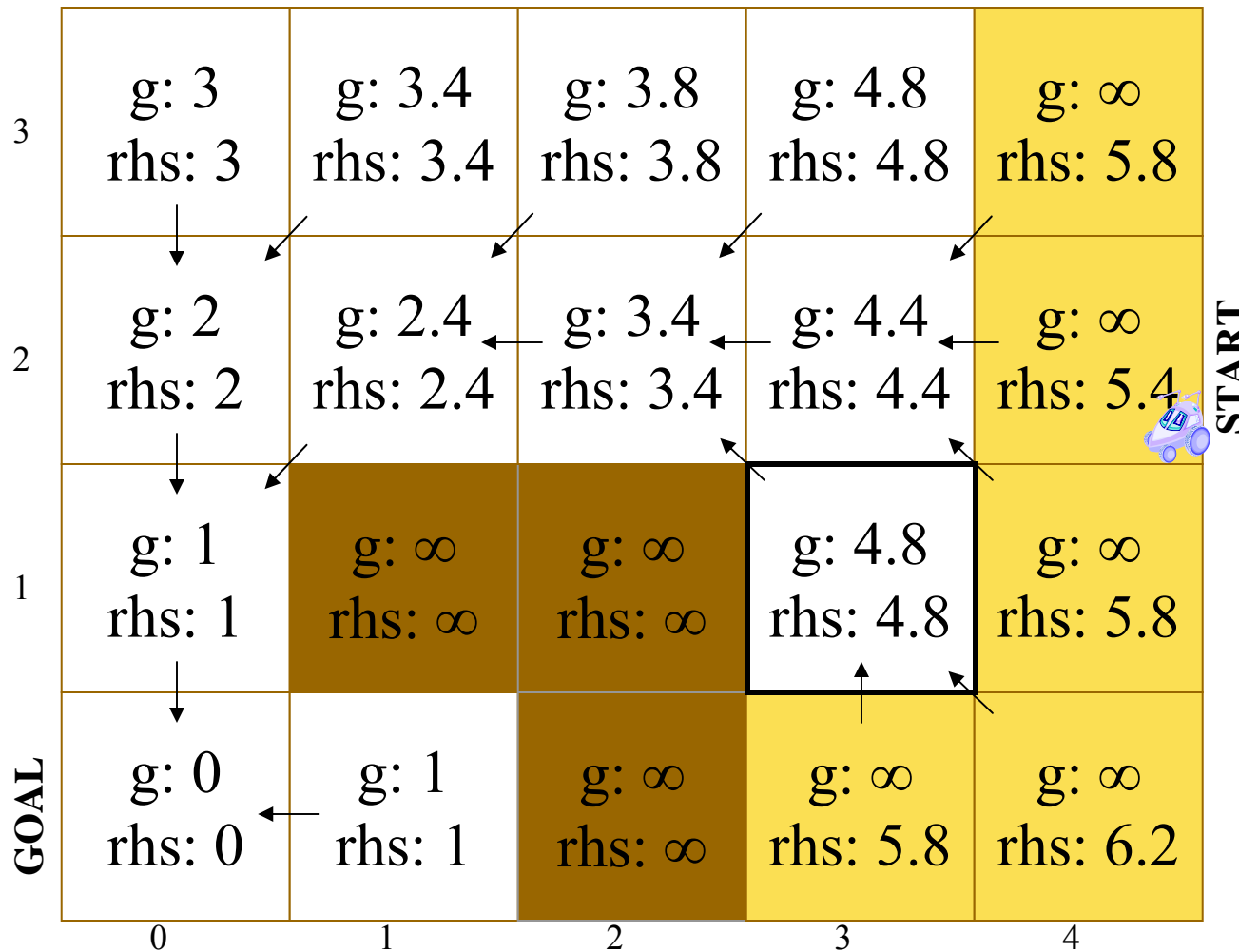


ComputeShortestPath
 Pop minimum item off
 open list - (3,1)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (21)

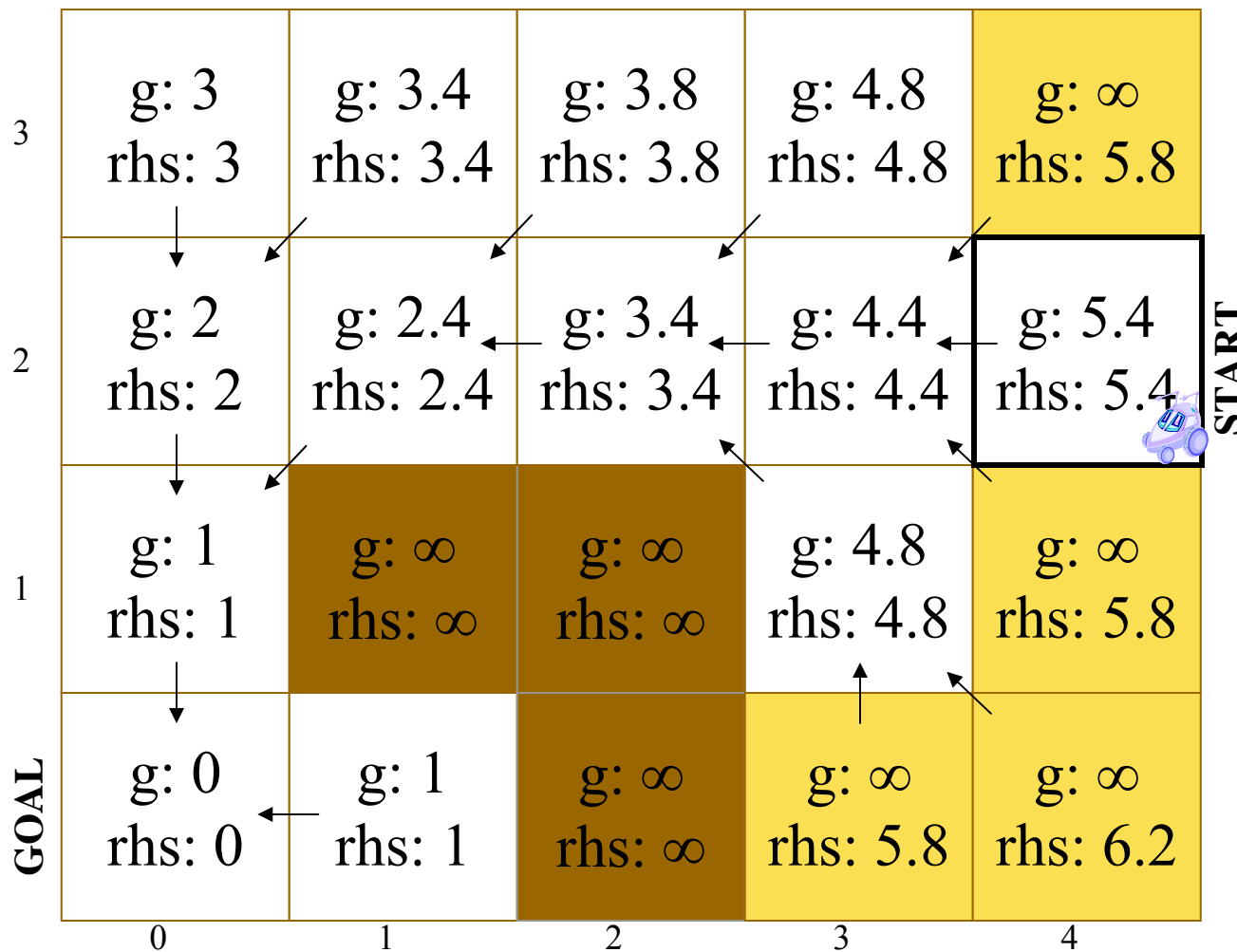


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (22)

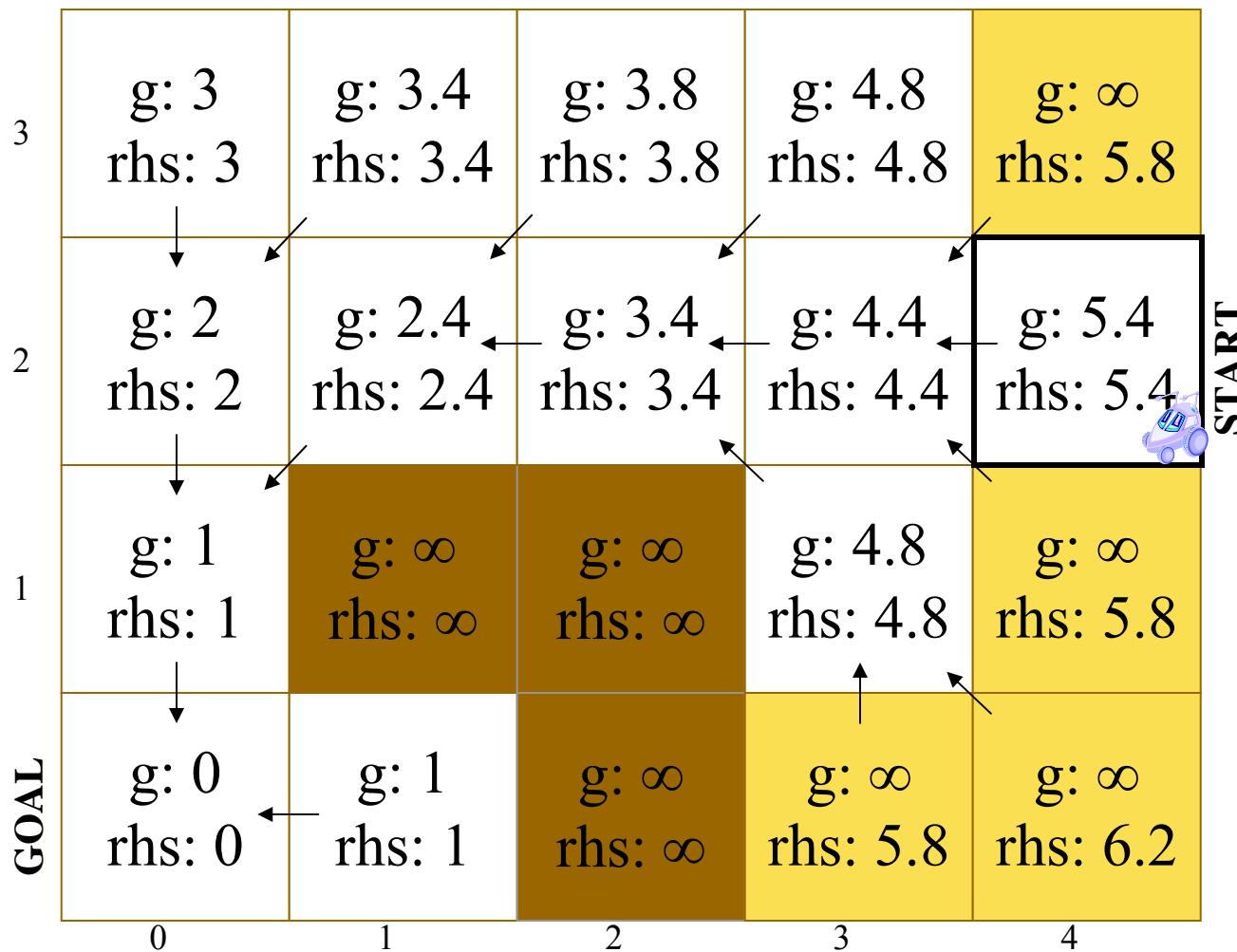


ComputeShortestPath
 Pop minimum item off
 open list - (4,2)
 It's over-consistent
 ($g > rhs$) so set $g=rhs$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (22b)

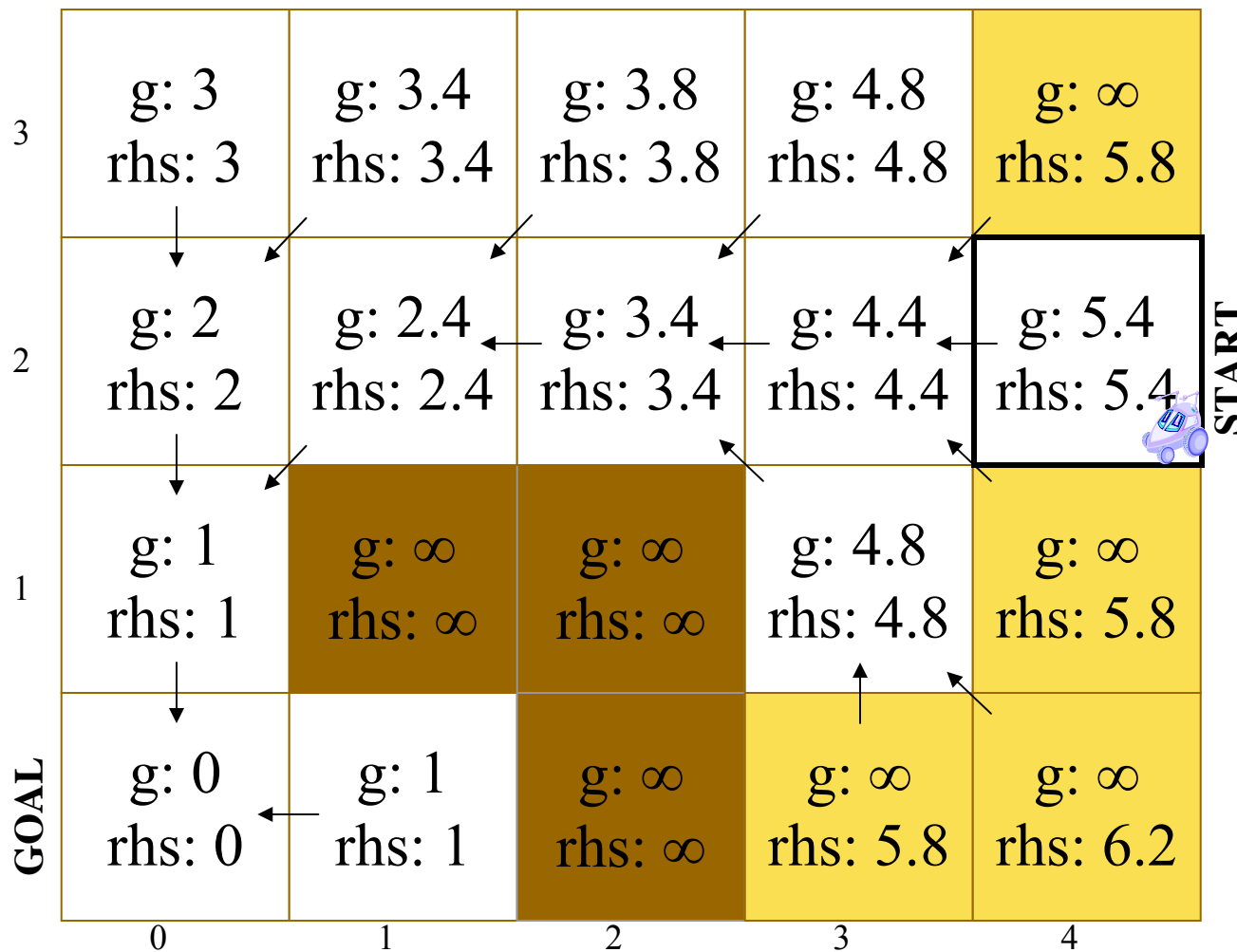


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (22c)

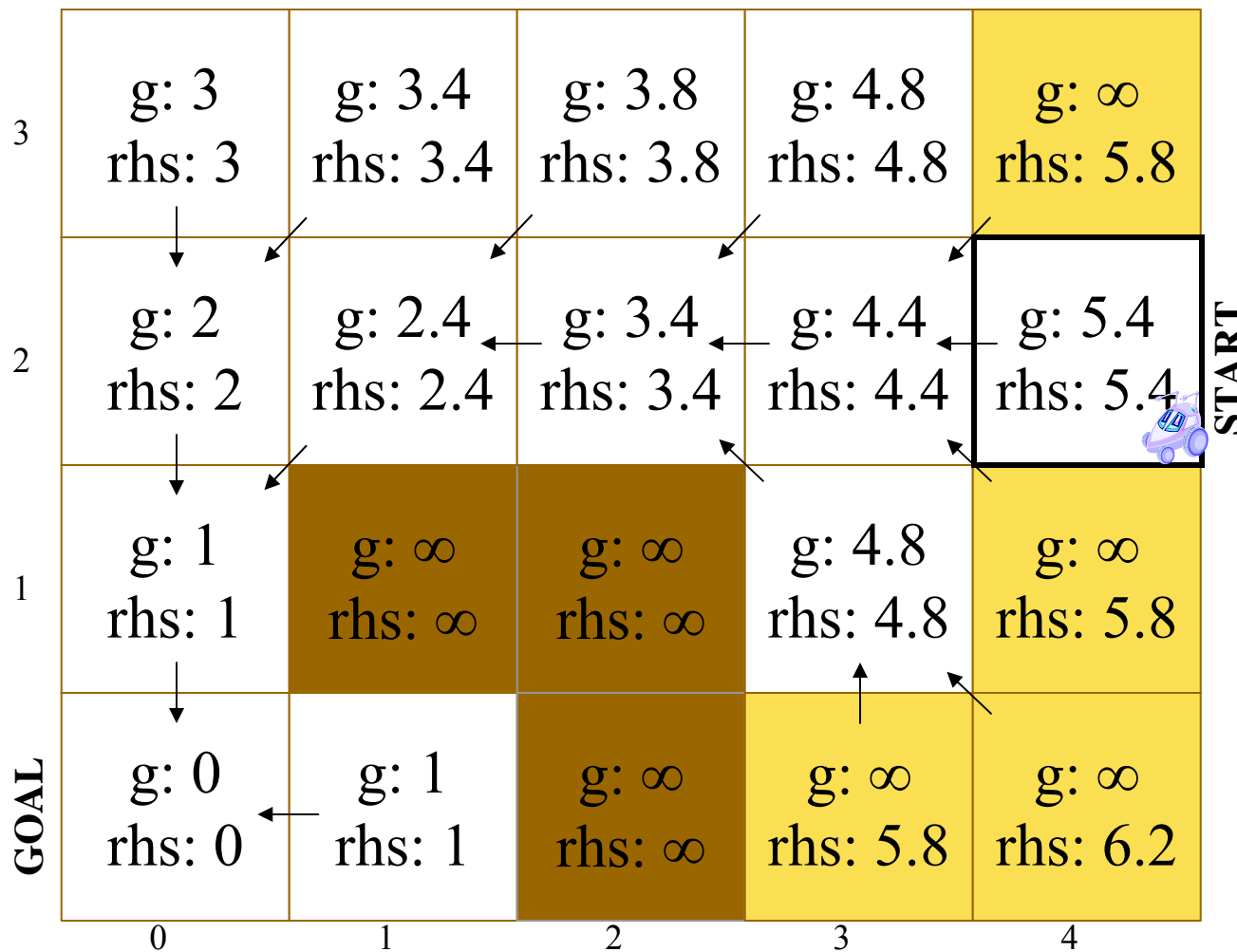


ComputeShortestPath
 At this point, the start node is consistent AND the top key on the open list is not less than the key of the start node. So we have an optimal path and can break out of the loop.

Legend

- Free
- Obstacle
- On open list

D* Lite: Planning (22d)

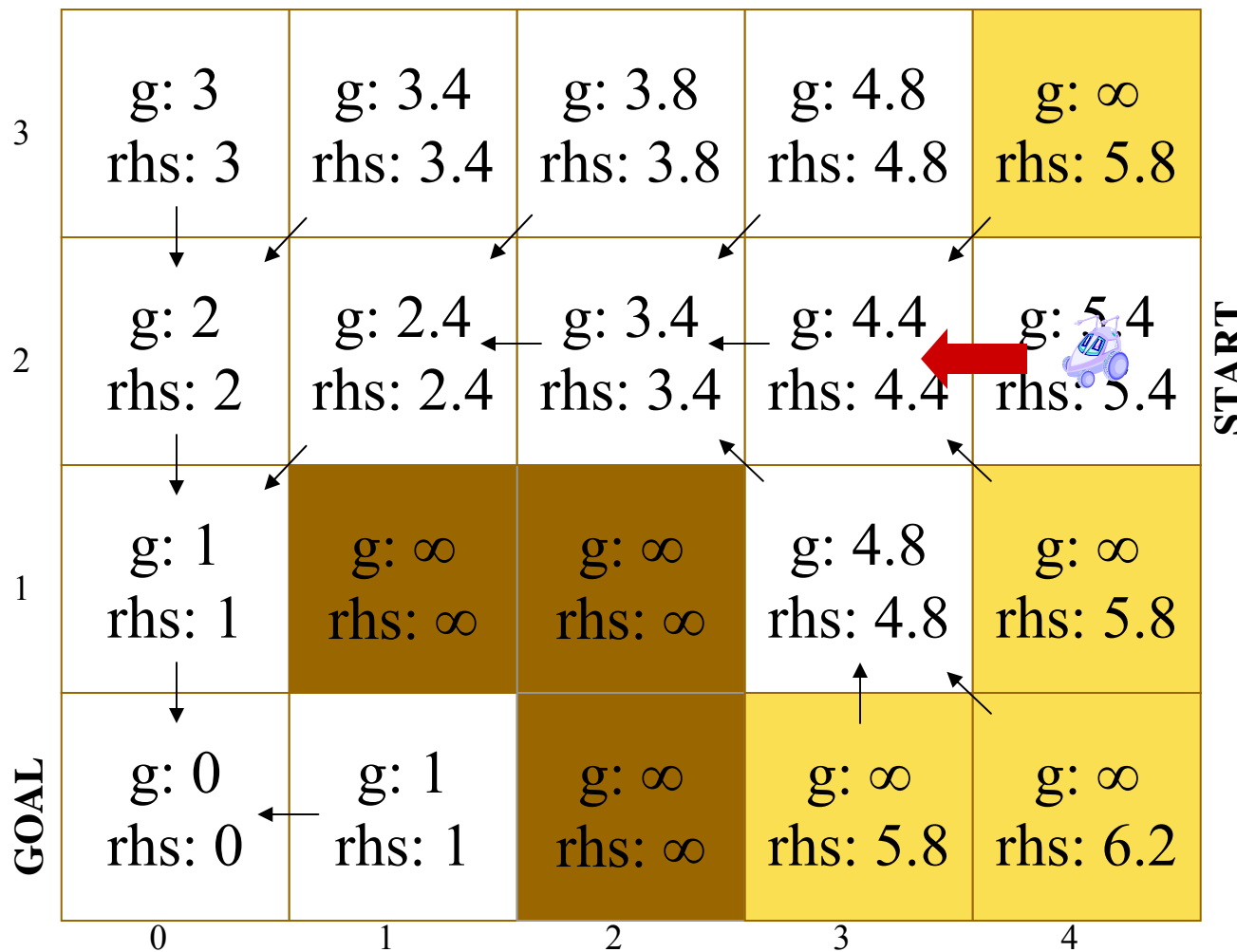


Note that there are still nodes on the open list – we leave them there. Also, for a larger map, there might be some nodes that are not touched at all but we still break out of the loop because we have an optimal path to the start.

Legend

- Free
- Obstacle
- On open list

D* Lite: Following the path (23)

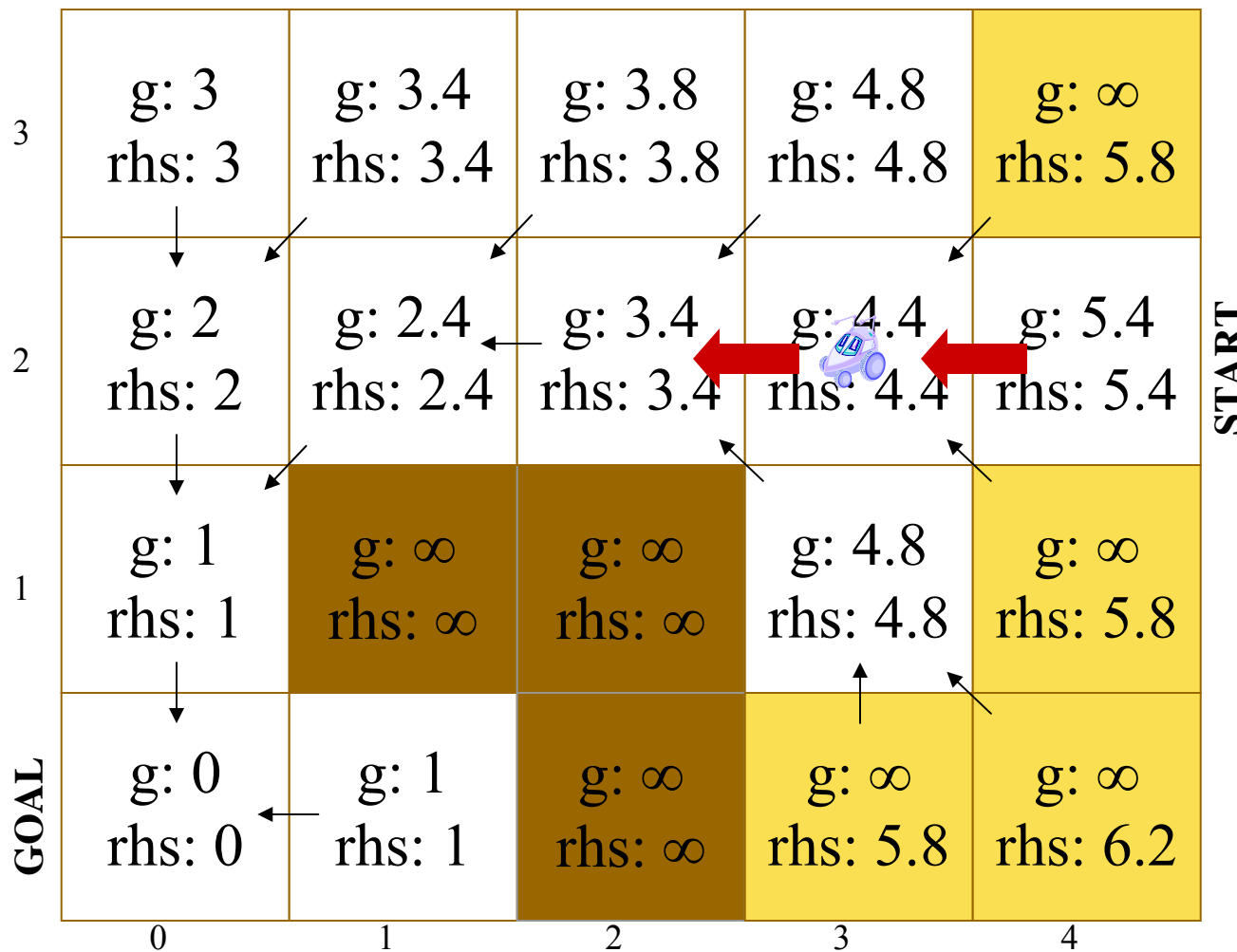


Follow the gradient of g values from the start state.

Legend

- Free
- Obstacle
- On open list

D* Lite: Following the path (24)

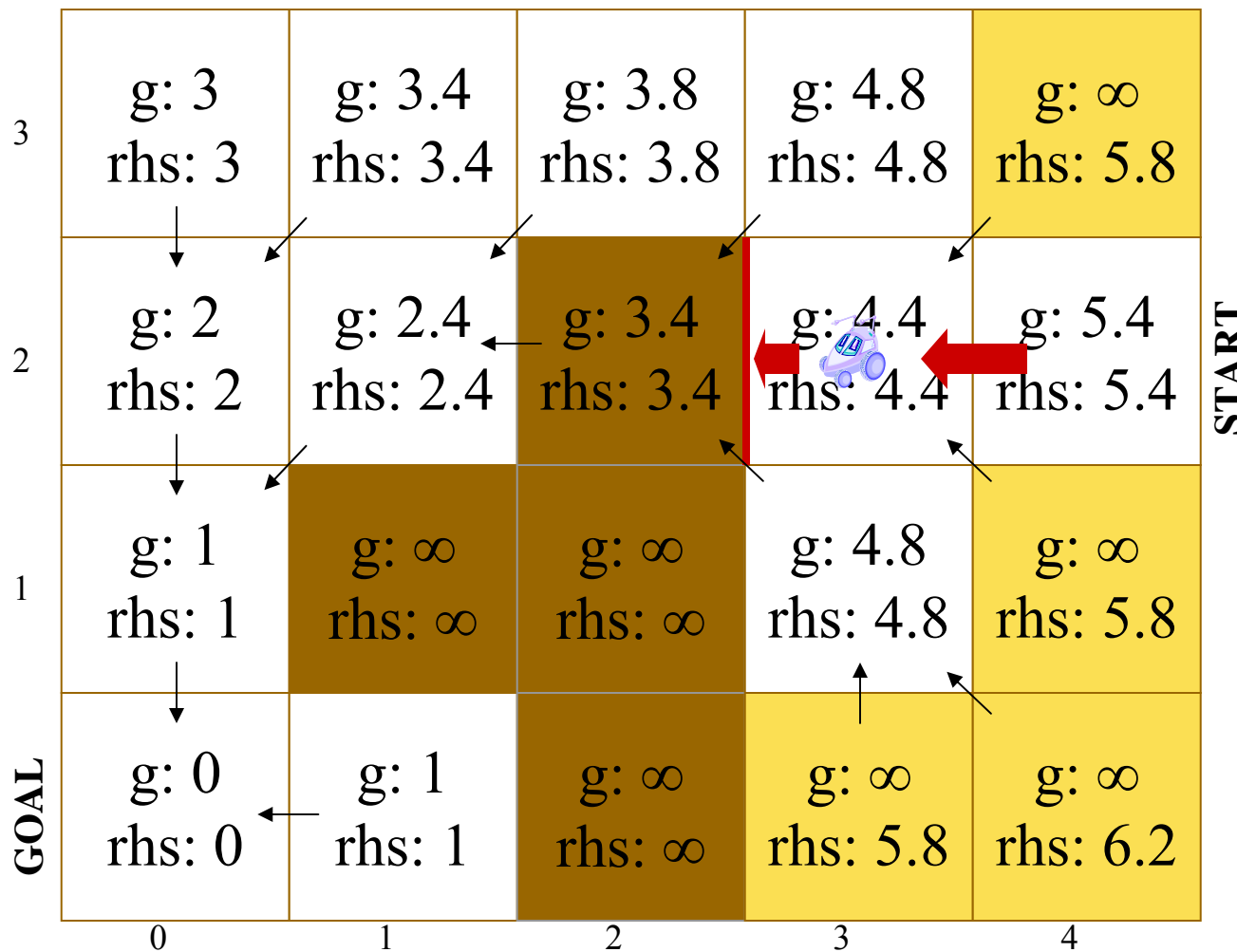


Follow the gradient of g values from the start state.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (25)



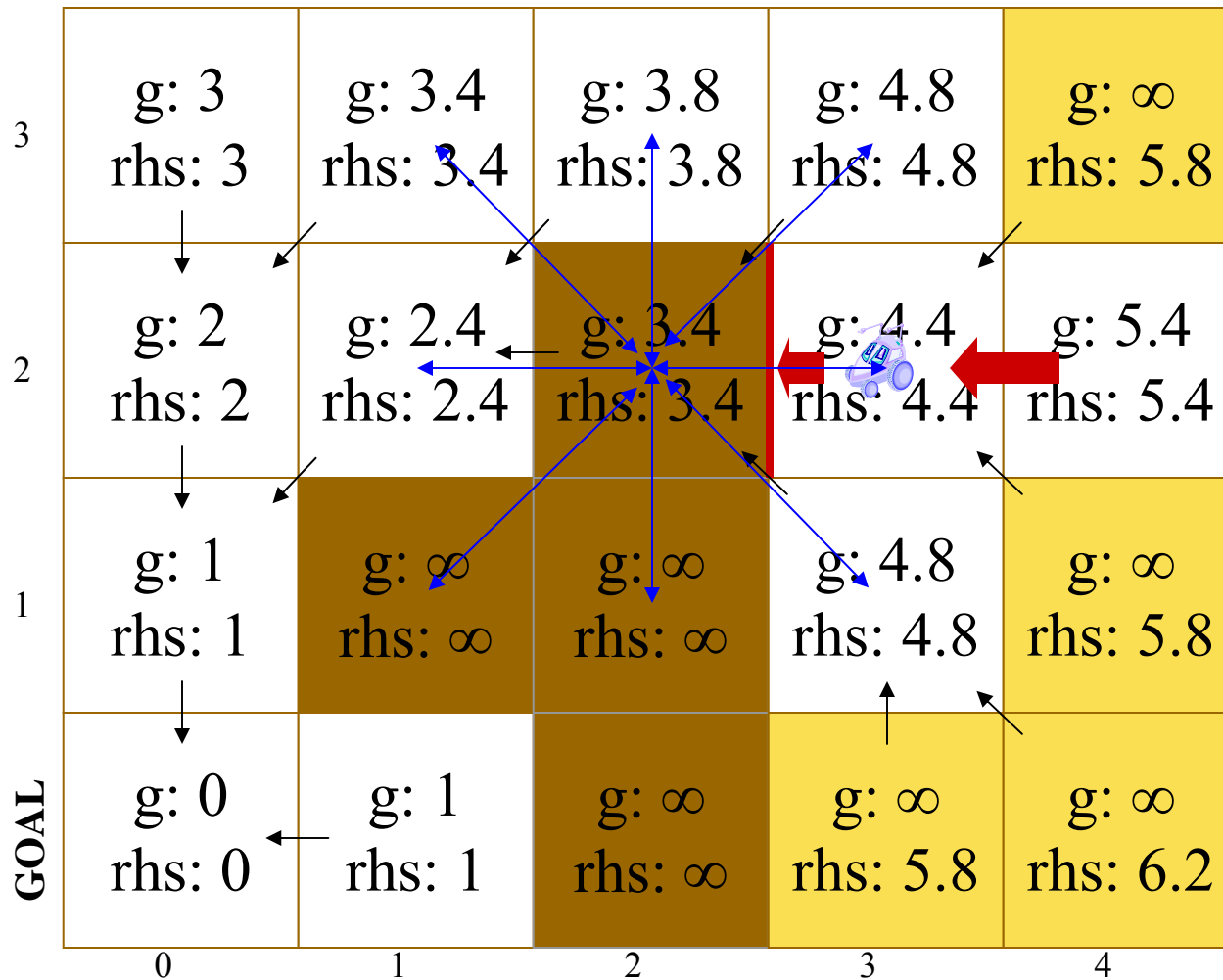
Suppose that, in trying to move from (3,2) to (2,2), the robot discovers that (2,2) is actually an obstacle. Now, **replanning** is required!

Legend

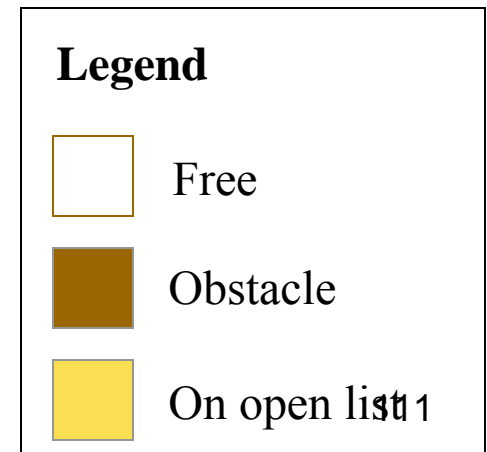
- Free
- Obstacle
- On open list

D* Lite: Replanning (25b)

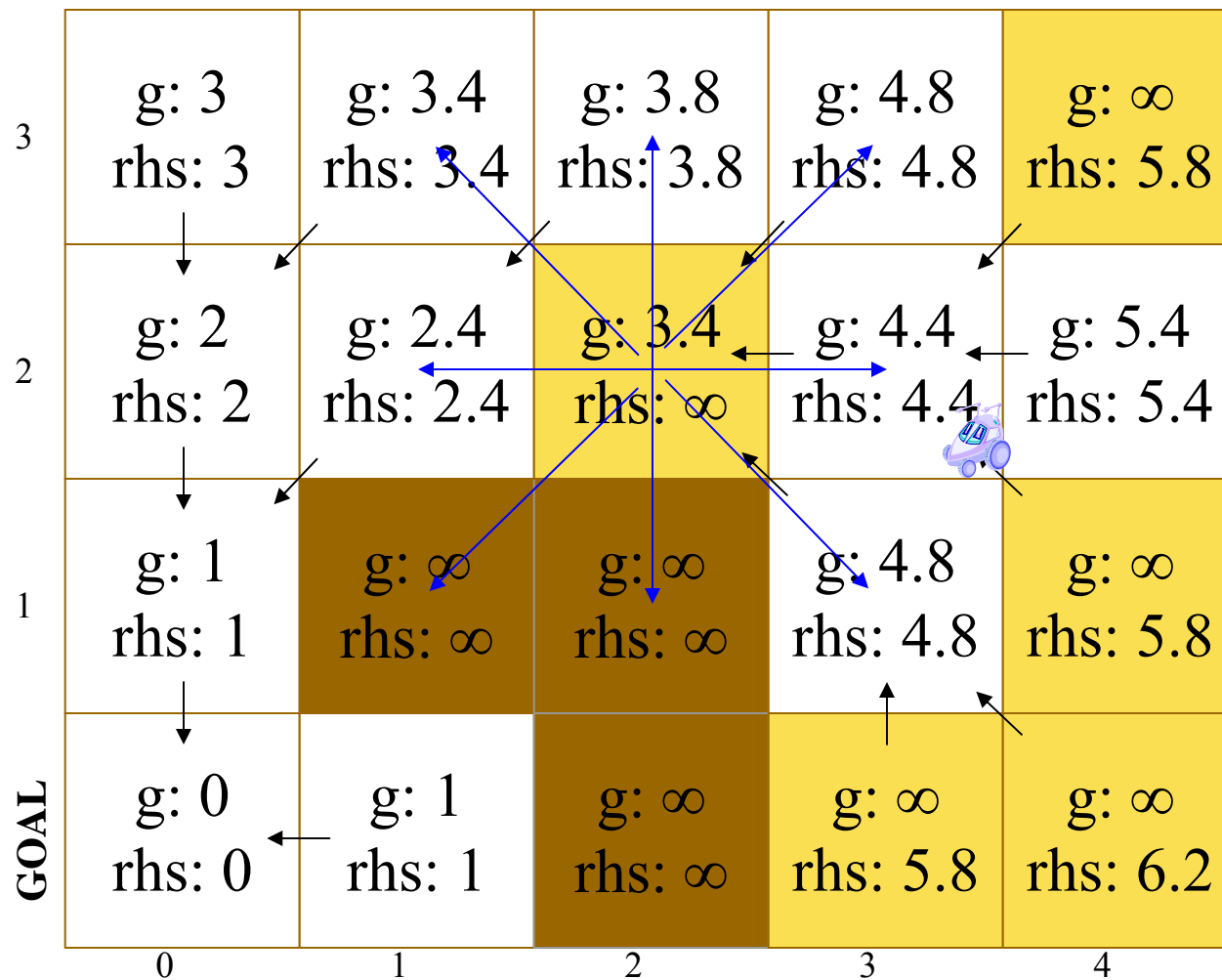
Edge (u,v) is from u to v



The algorithm dictates that for all directed edges (u, v) with changed edge costs, we should call *UpdateVertex(u)*. Since the edges in our graph are bidirectional and all edges into or out of (2,2) are affected, we need to consider 16 different edges!



D* Lite: Replanning (26)



START

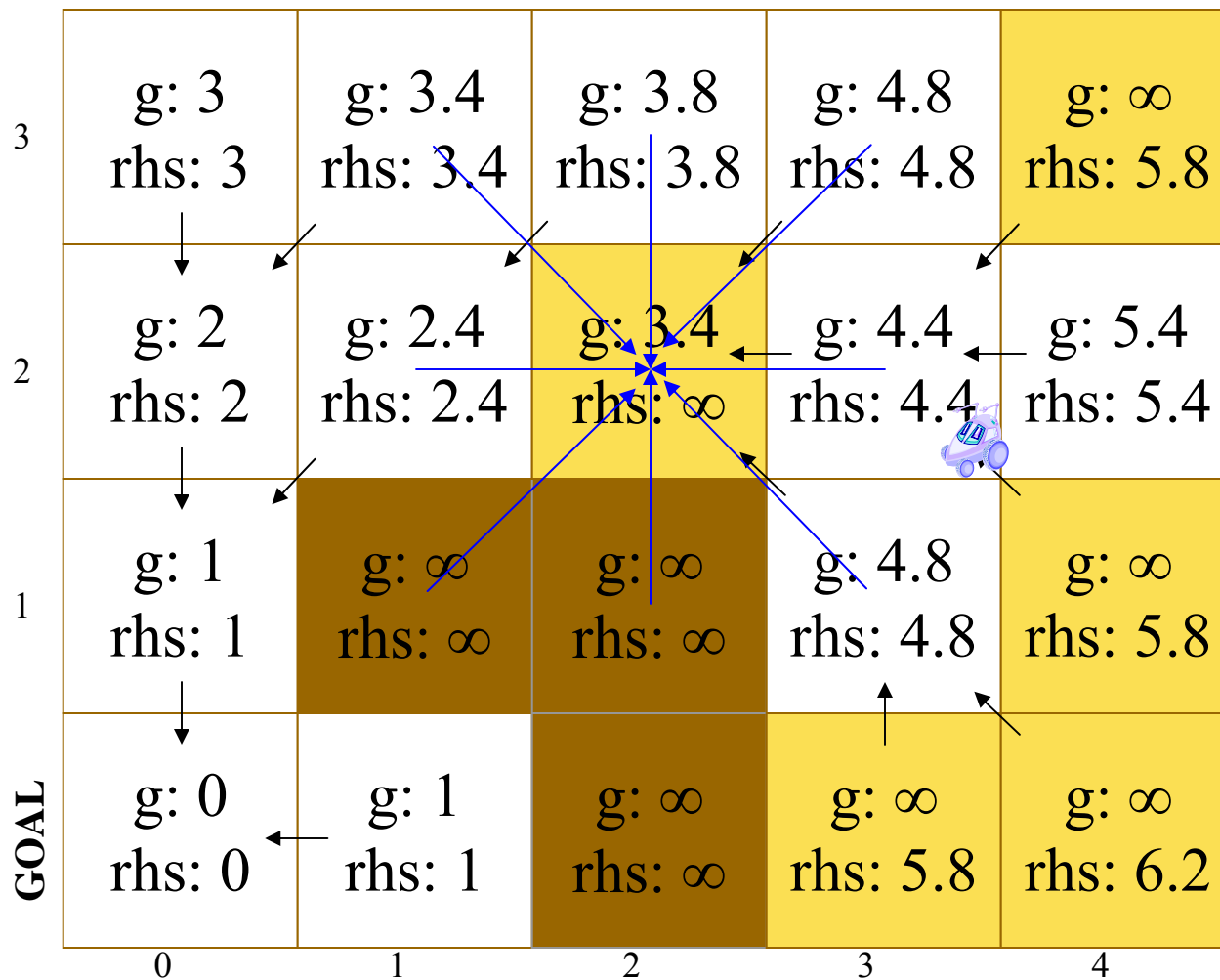
UpdateVertex

First, consider the outgoing edges from (2,2) to its 8 neighbors. For each of these, we call *UpdateVertex()* on (2,2). Because the transition costs are now ∞ , the *rhs* value of (2,2) is raised, making it inconsistent. It is put on the open list .

Legend

- Free
- Obstacle
- On open list₂

D* Lite: Replanning (26b)



START

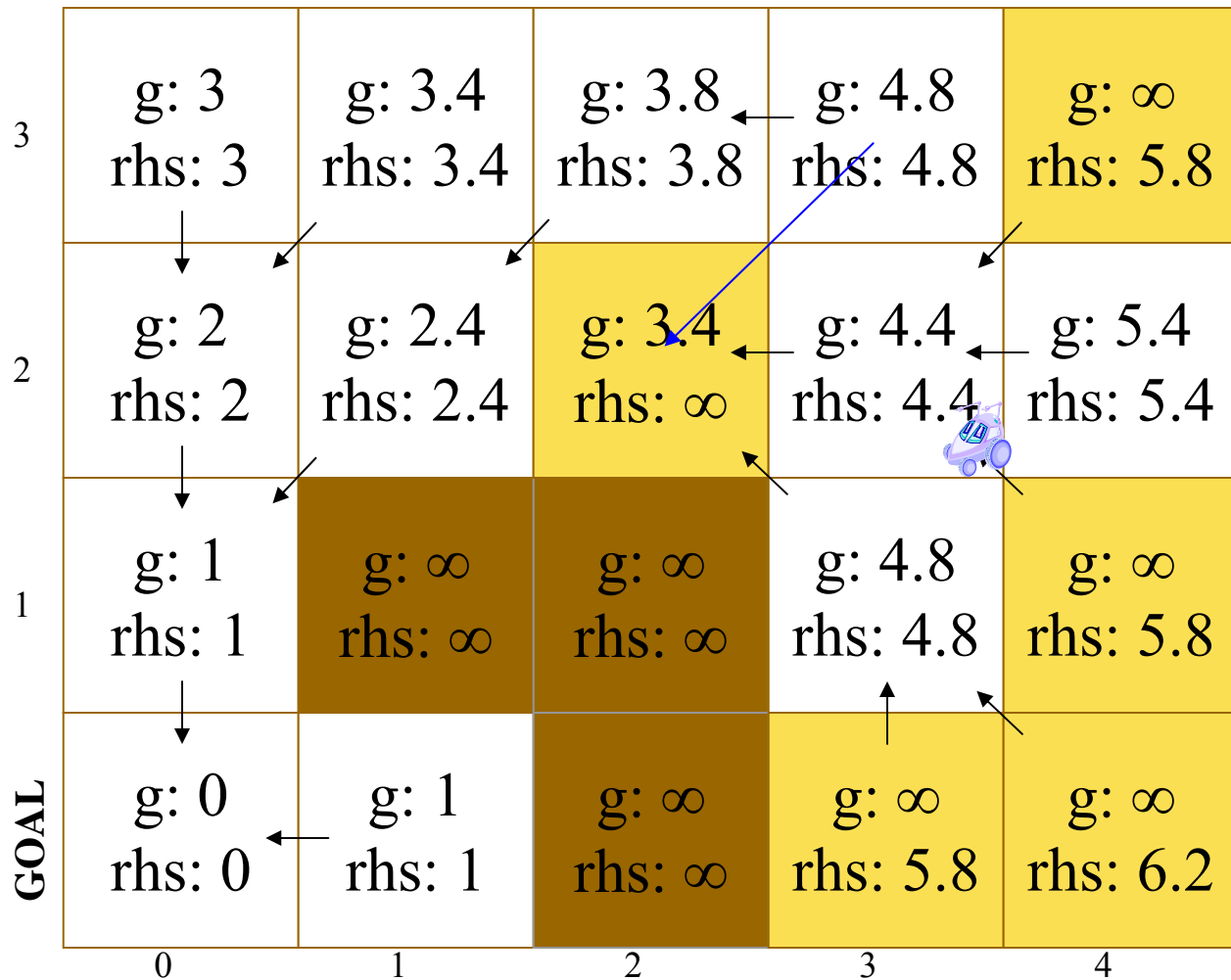
UpdateVertex

Next, consider the incoming edges to (2,2). (i.e. call *UpdateVertex()* the neighbors of (2,2)). Since the transition costs of these edges are ∞ , any *rhs* value previously computed using (2,2) will change. (next slide)

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (27)



START

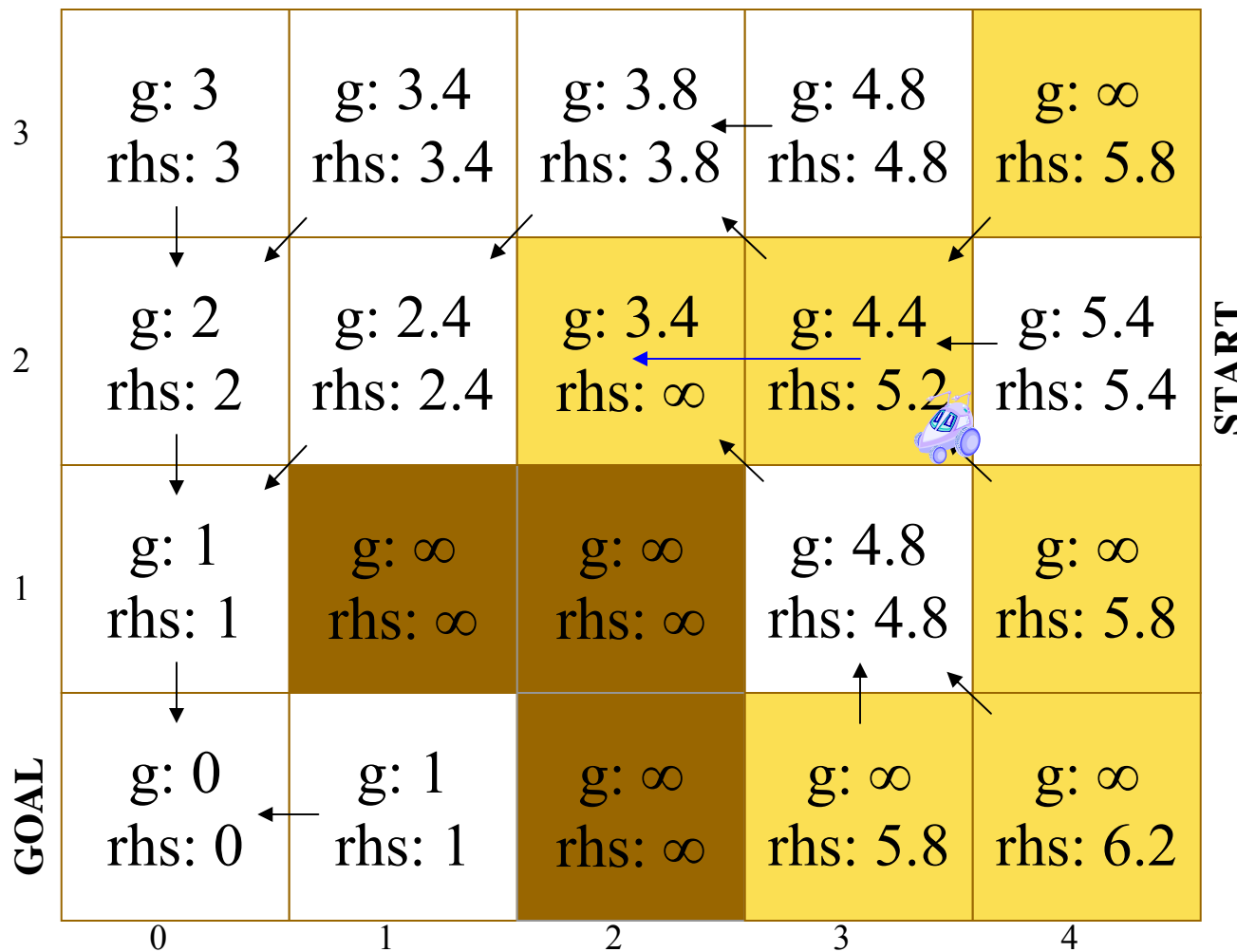
UpdateVertex

One of the neighbors of (2,2) is (3,3). The minimum possible *rhs* value of (3,3) is still 4.8, but this value is based on the *g* value of (2,3), not (2,2). Because (3,3) is still consistent, it's not put on the open list.

Legend

- Free
- Obstacle
- On open list




D* Lite: Replanning (28)



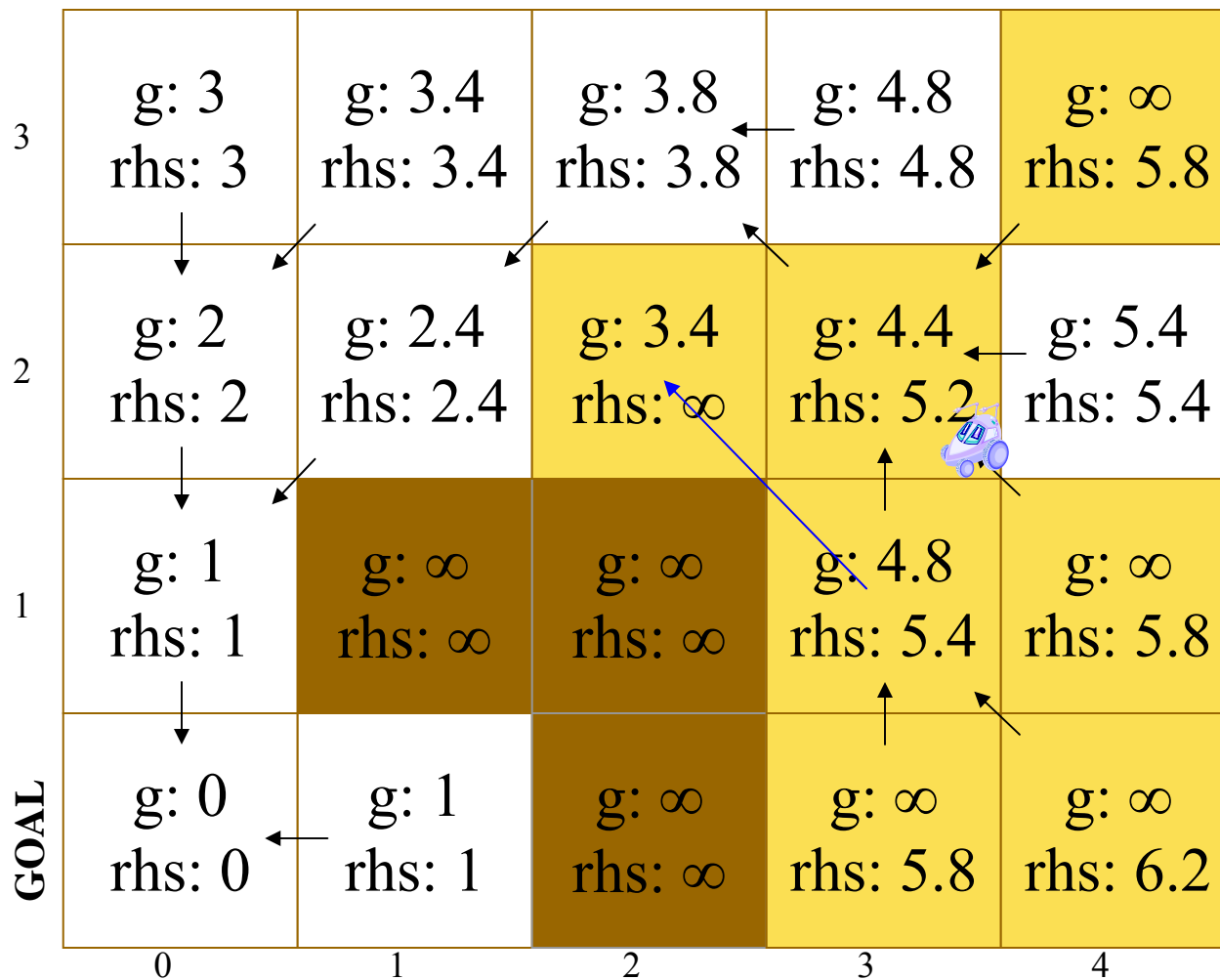
UpdateVertex

Another neighbor of (2,2) is (3,2). Because of the increased transition cost to (2,2), The minimum possible *rhs* value of (3,2) is now 5.2, computed based on the *g* value of (2,3) [3.8+1.4=5.2].

Legend

-  Free
-  Obstacle
-  On open list

D* Lite: Replanning (29)



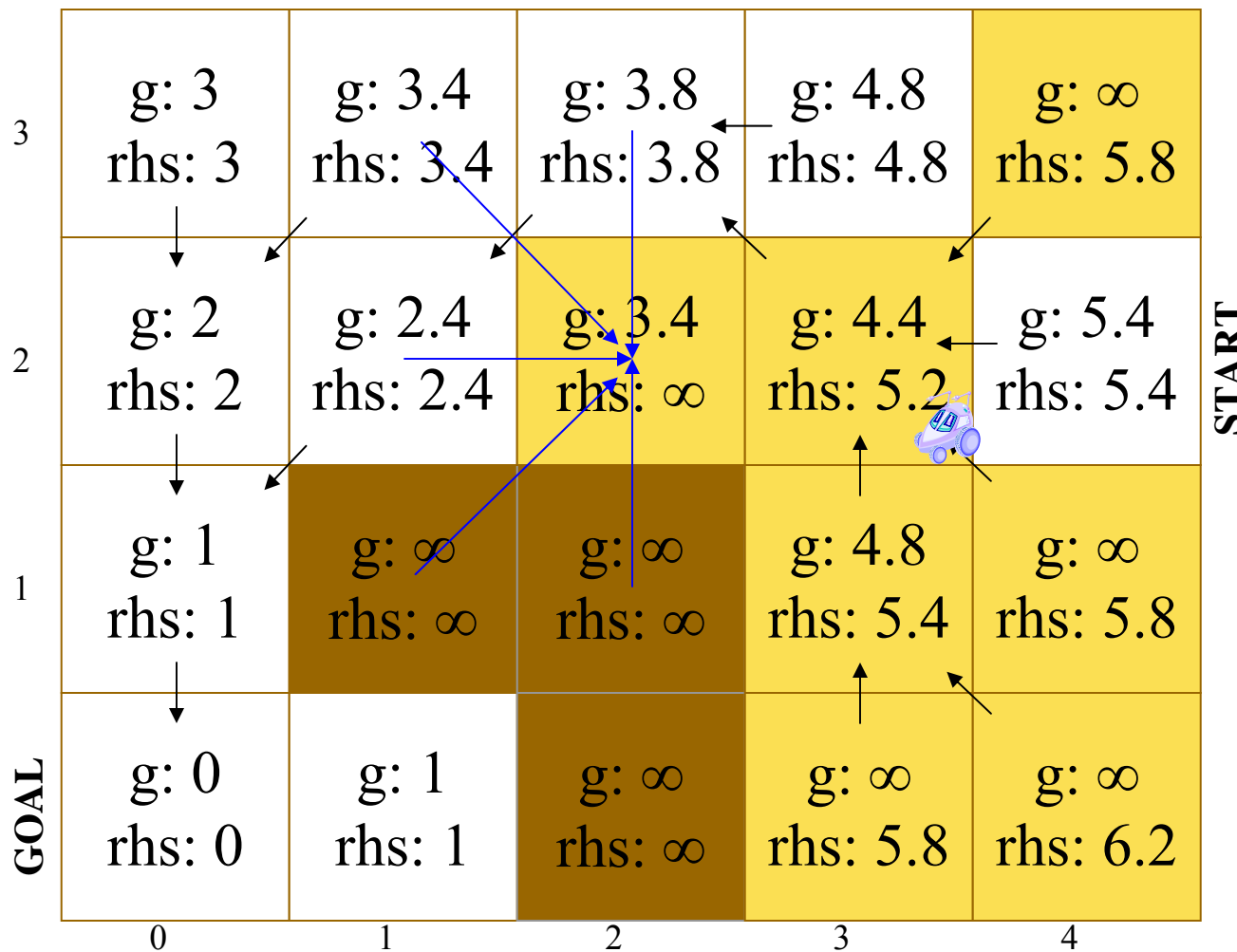
UpdateVertex

Another neighbor of (2,2) is (3,1). The minimum possible *rhs* value of (3,1) is now 5.4, computed based on the *g* value of (3,2) **Note:** the *rhs* value of a node is always computed using the *g* (not *rhs*) values of its successors.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (29b)



UpdateVertex

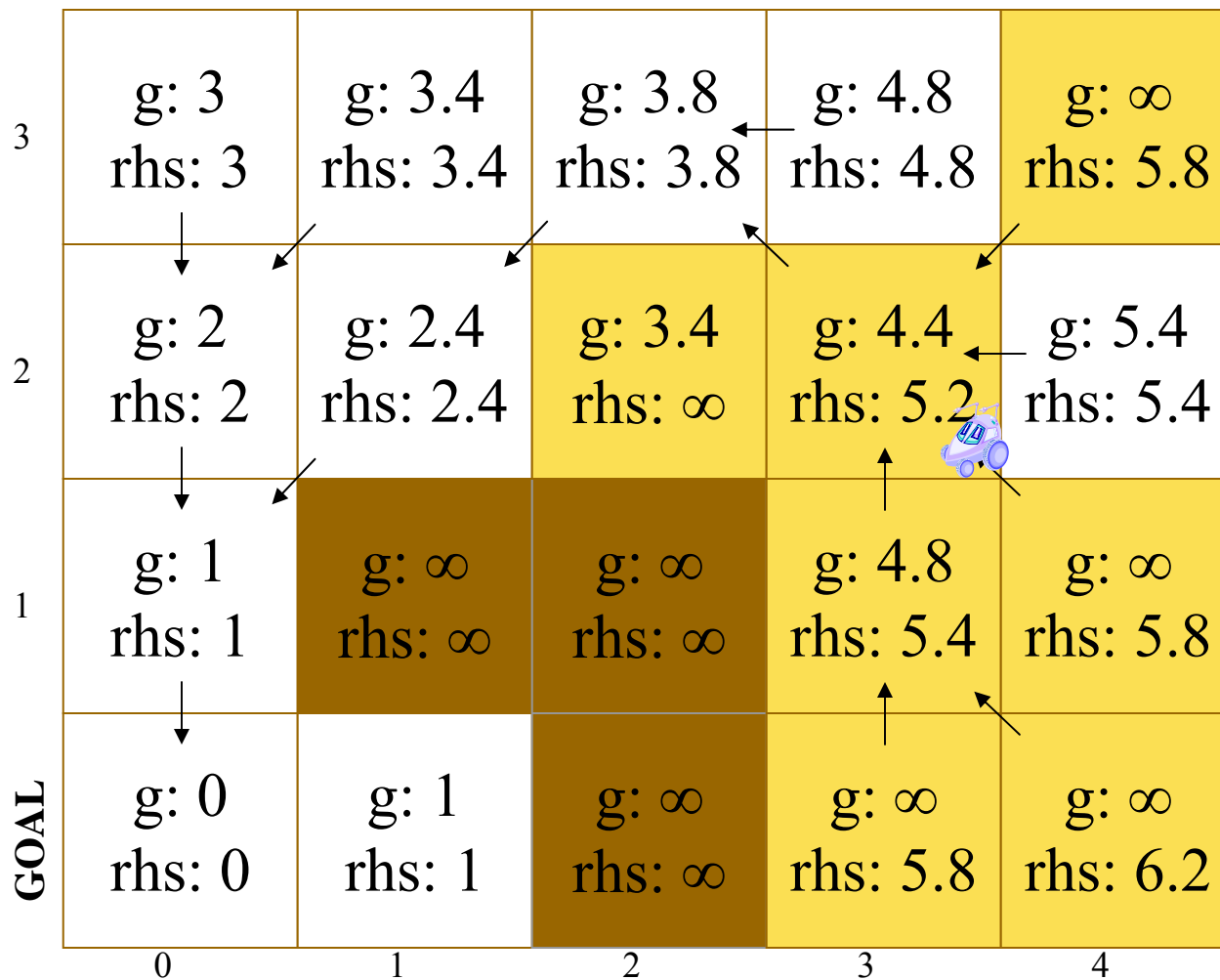
None of the other neighbors of (2,2) end up being inconsistent.

START

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (29c)

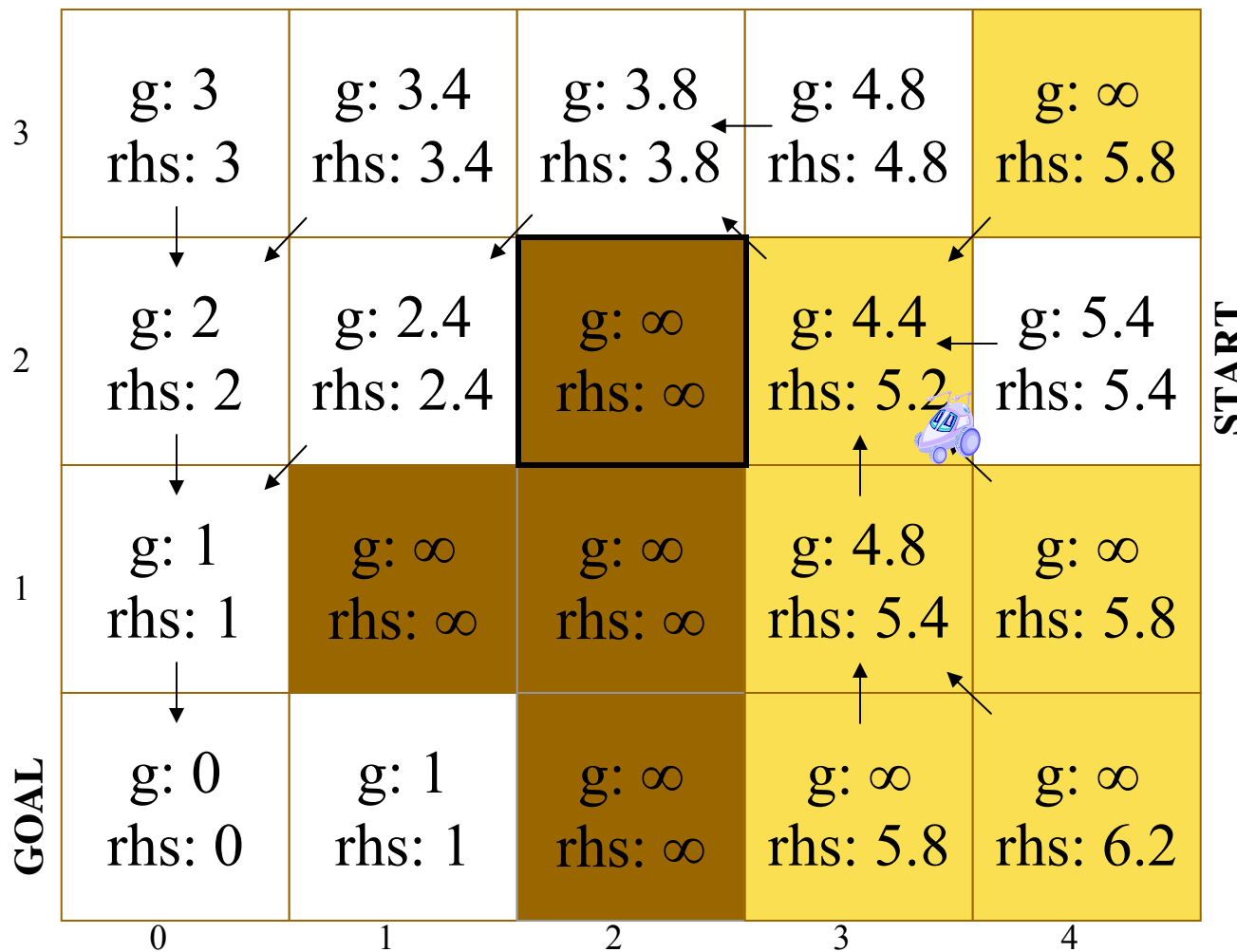


Now, we go back to calling *ComputeShortestPath()* until we have an optimal path. The node corresponding to the robot's current position is inconsistent and its key is greater than the minimum key on the open list, so we know that we do not yet have an optimal path..

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (30)

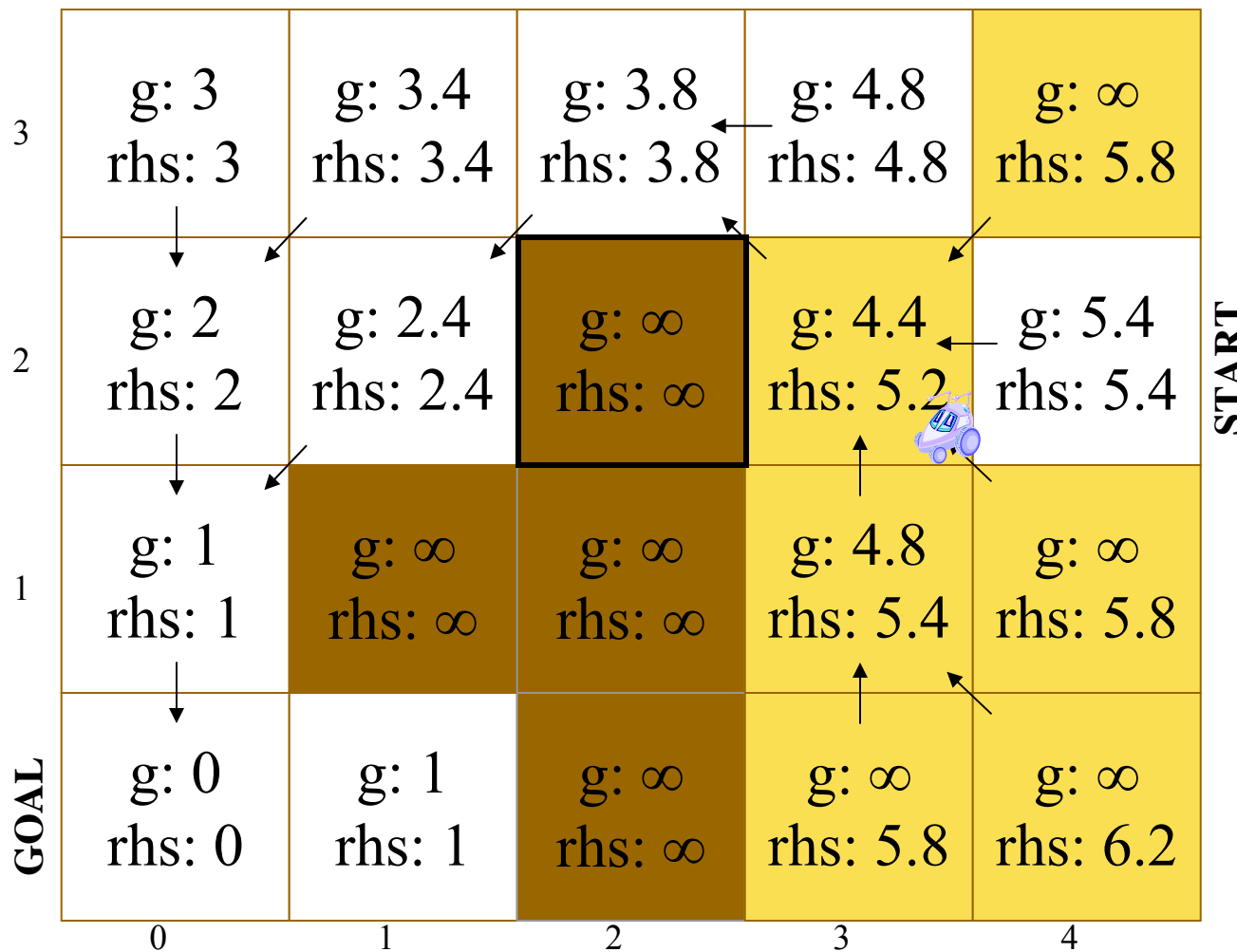


ComputeShortestPath
 Pop minimum item off
 open list - (2,2)
 It's under-consistent
 ($g < rhs$) so set $g = \infty$.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (30b)

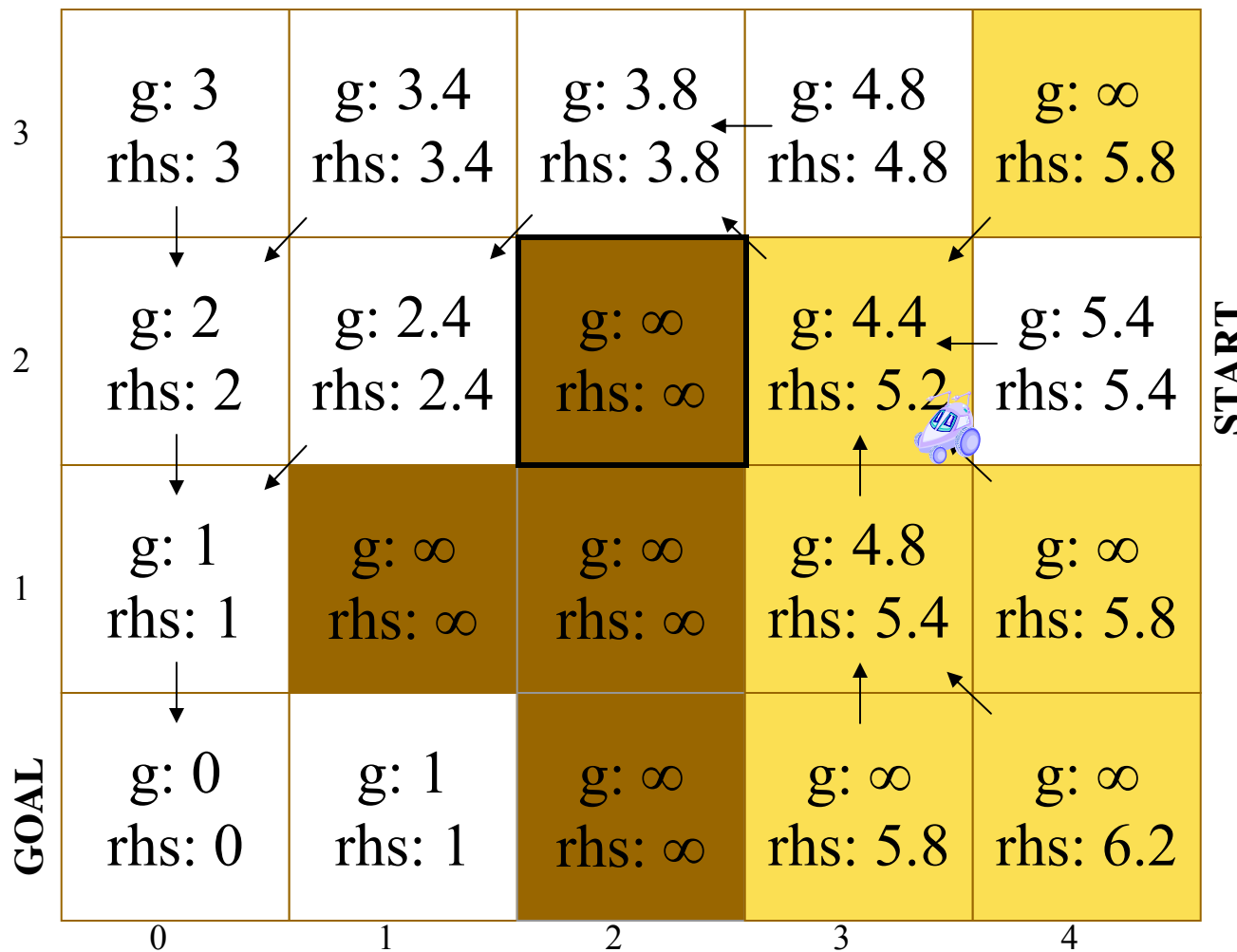


ComputeShortestPath
 Expand the popped node and put predecessors that become inconsistent (in this case, none) onto the open list.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (30c)

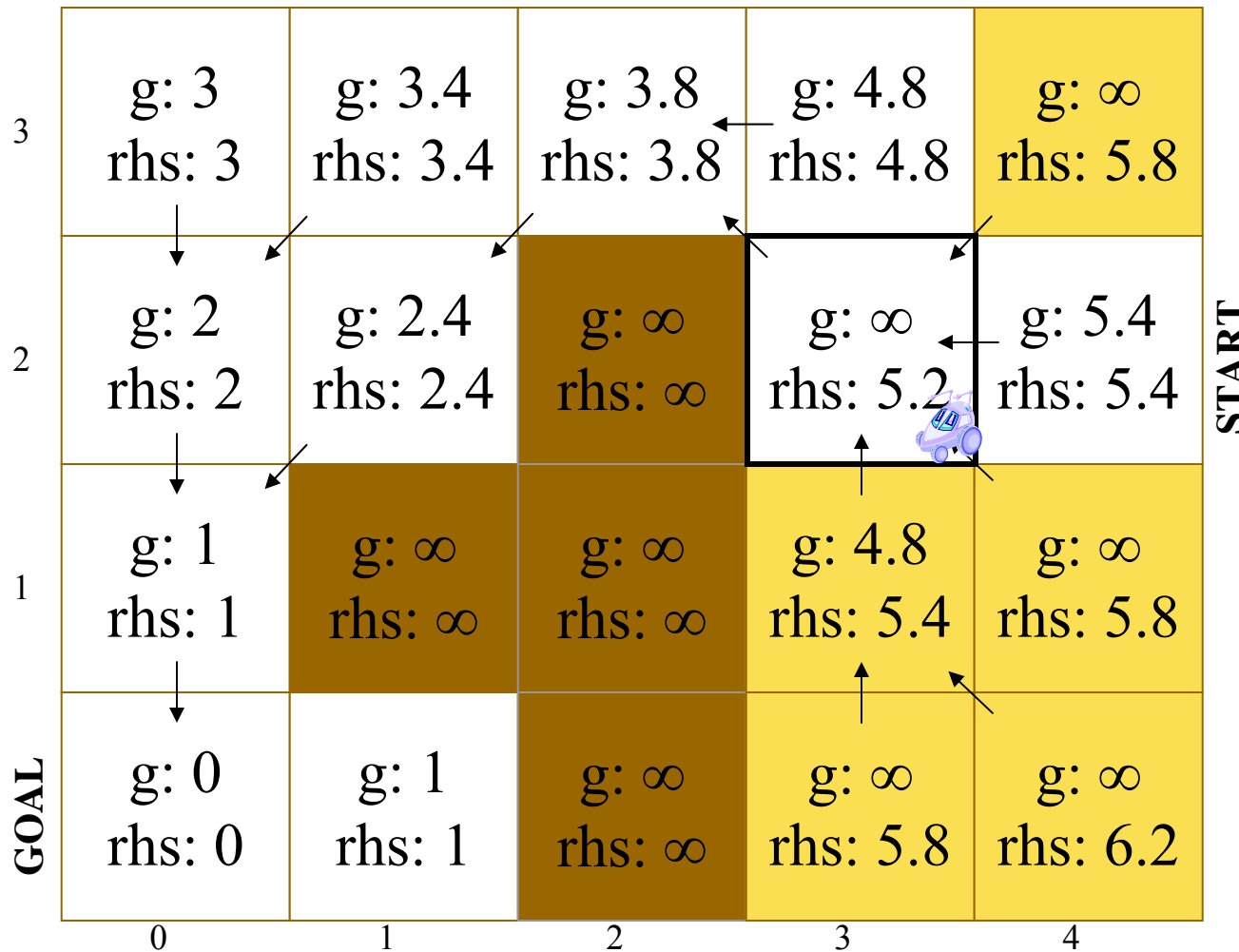


ComputeShortestPath
 Because (2,2) was underconsistent when it was popped, we need to call *UpdateVertex()* on it. This has no effect since its *rhs* value is up to date and it is consistent.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (31)

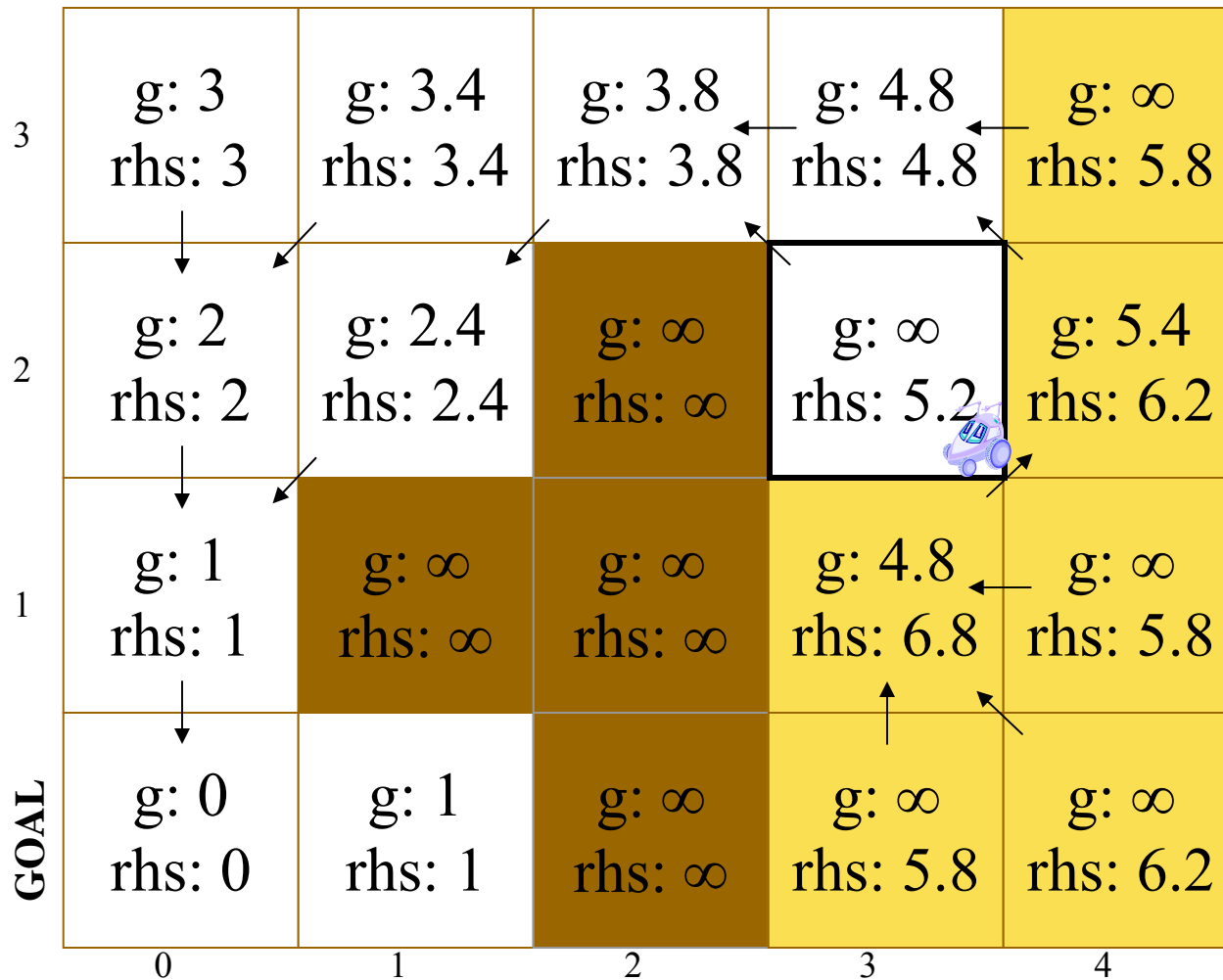


ComputeShortestPath
 Pop minimum item off open list - (3,2)
 It's under-consistent (g < rhs) so set g = ∞.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (32)



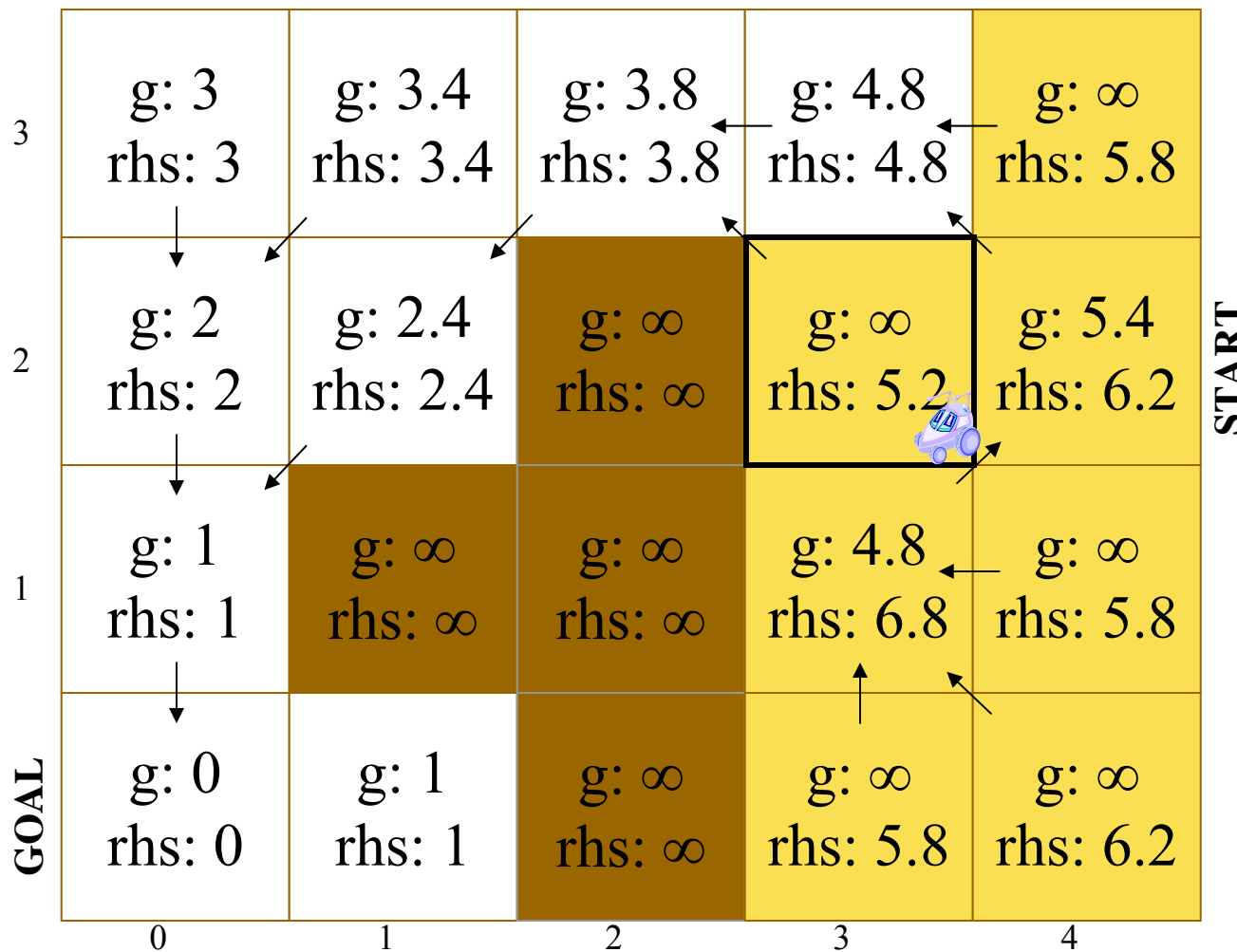
START

ComputeShortestPath
 Expand the popped node and update predecessors. (4,2) becomes inconsistent. (3,1) gets updated and is still inconsistent. (4,1)'s rhs value doesn't change but is now computed from the g value of (3,1)

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (33)



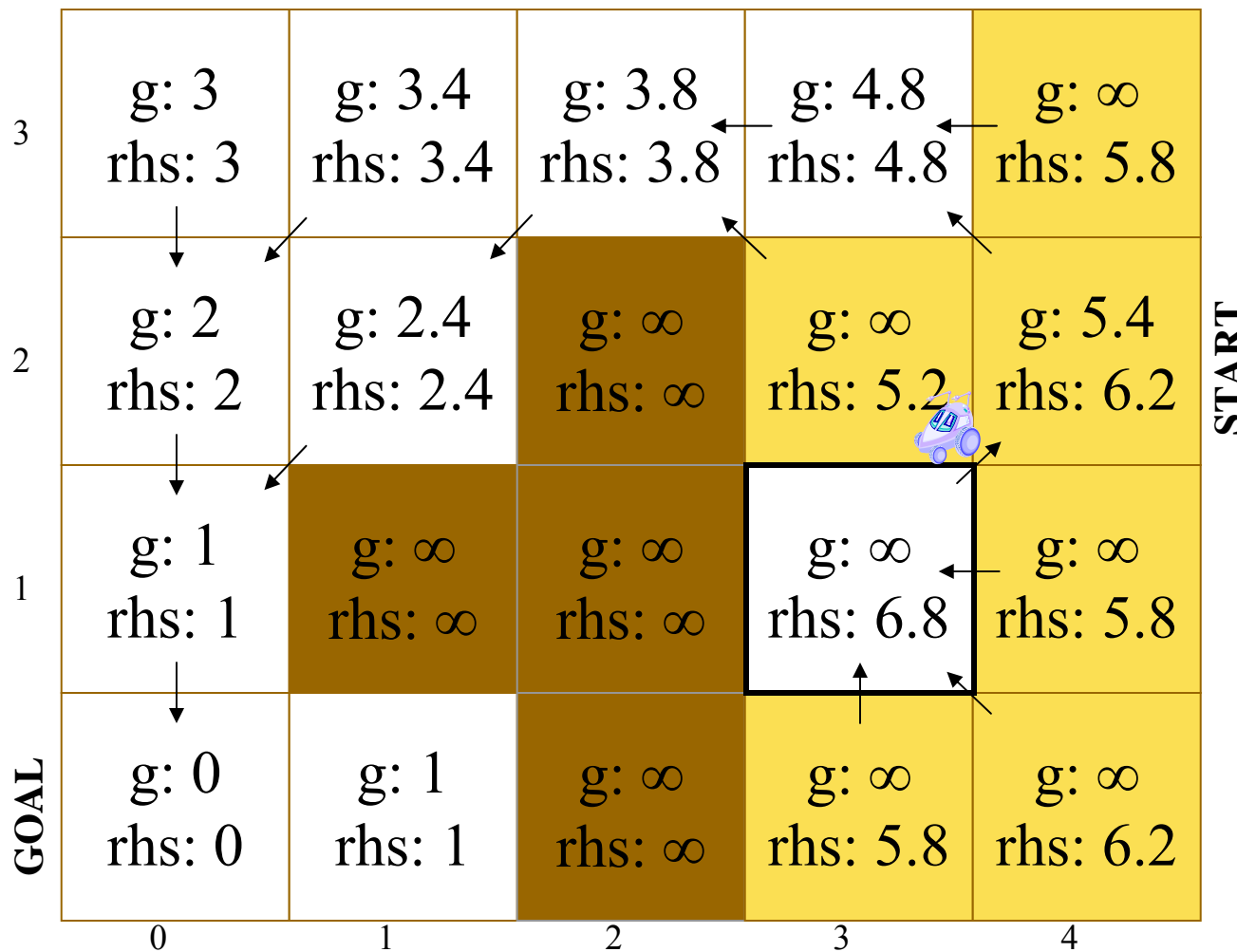
ComputeShortestPath

Because (3,2) was underconsistent when it was popped, we need to call *UpdateVertex()* on it. This results in it being put back onto the open list since it is still inconsistent.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (34)

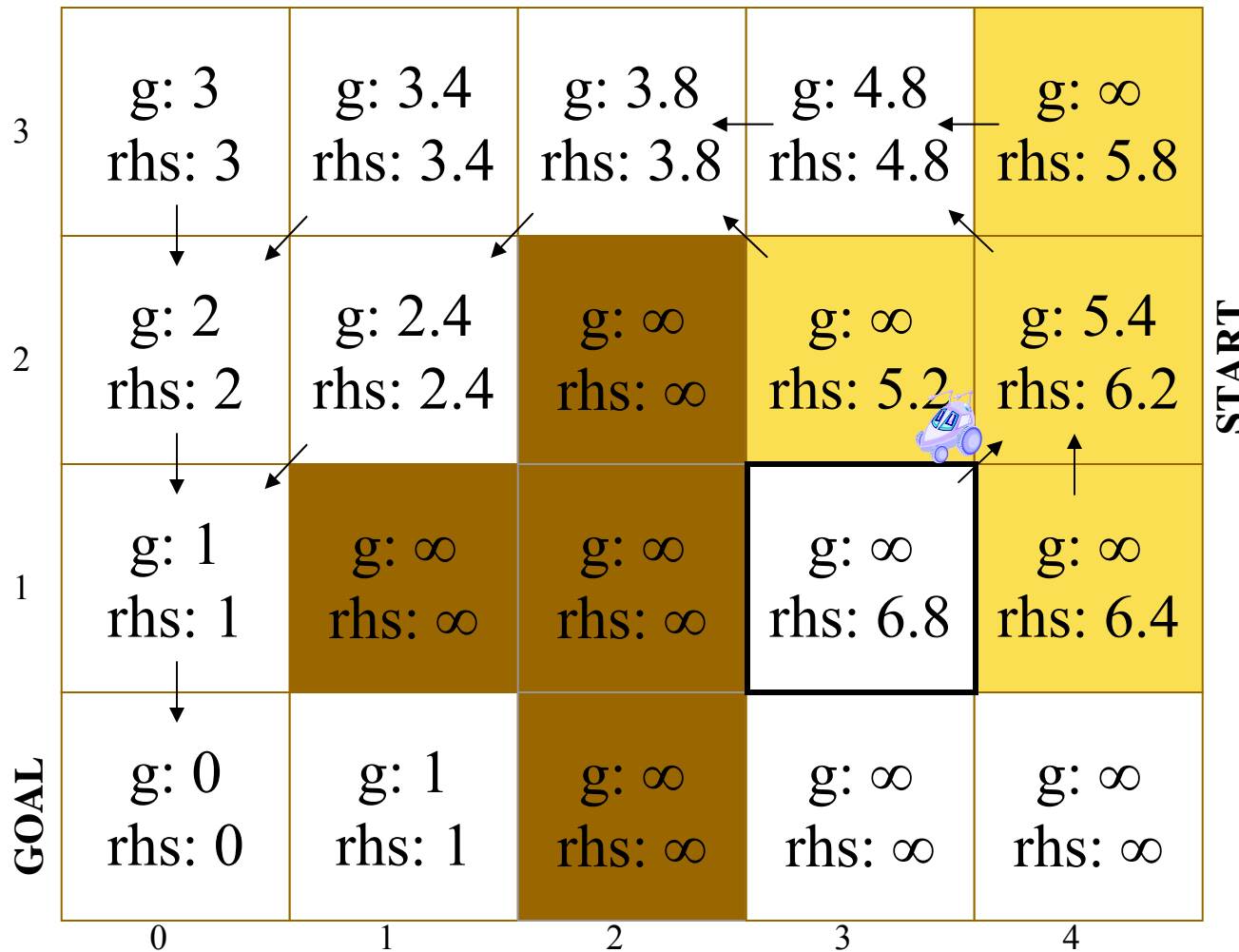


ComputeShortestPath
 Pop minimum item off open list - (3,1)
 It's under-consistent (g < rhs) so set g = ∞.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (35)

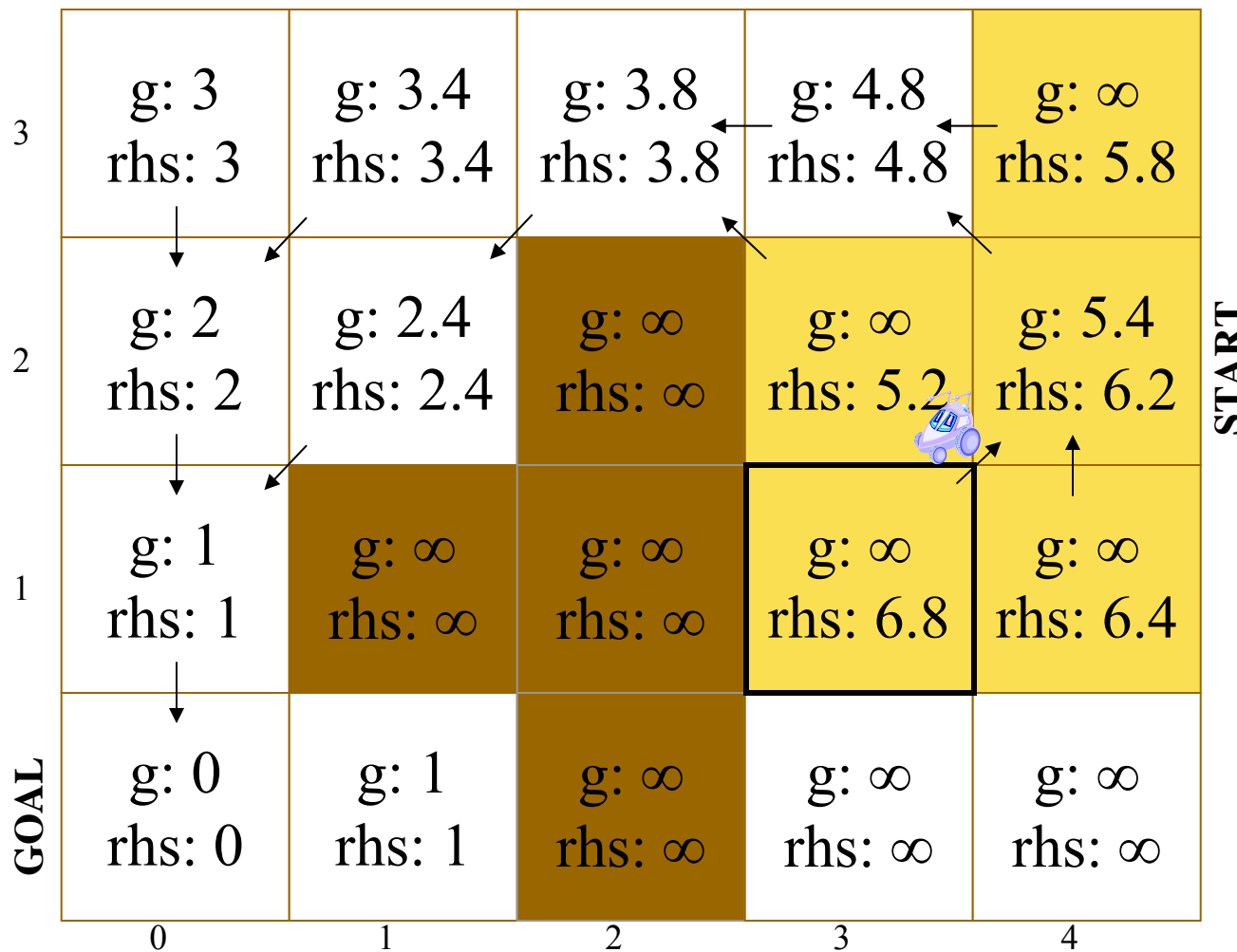


ComputeShortestPath
 Expand the popped node and update predecessors. (4,1) gets updated and is still inconsistent. (3,0) and (4,0) get updated but are now consistent since both g and rhs are ∞

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (36)



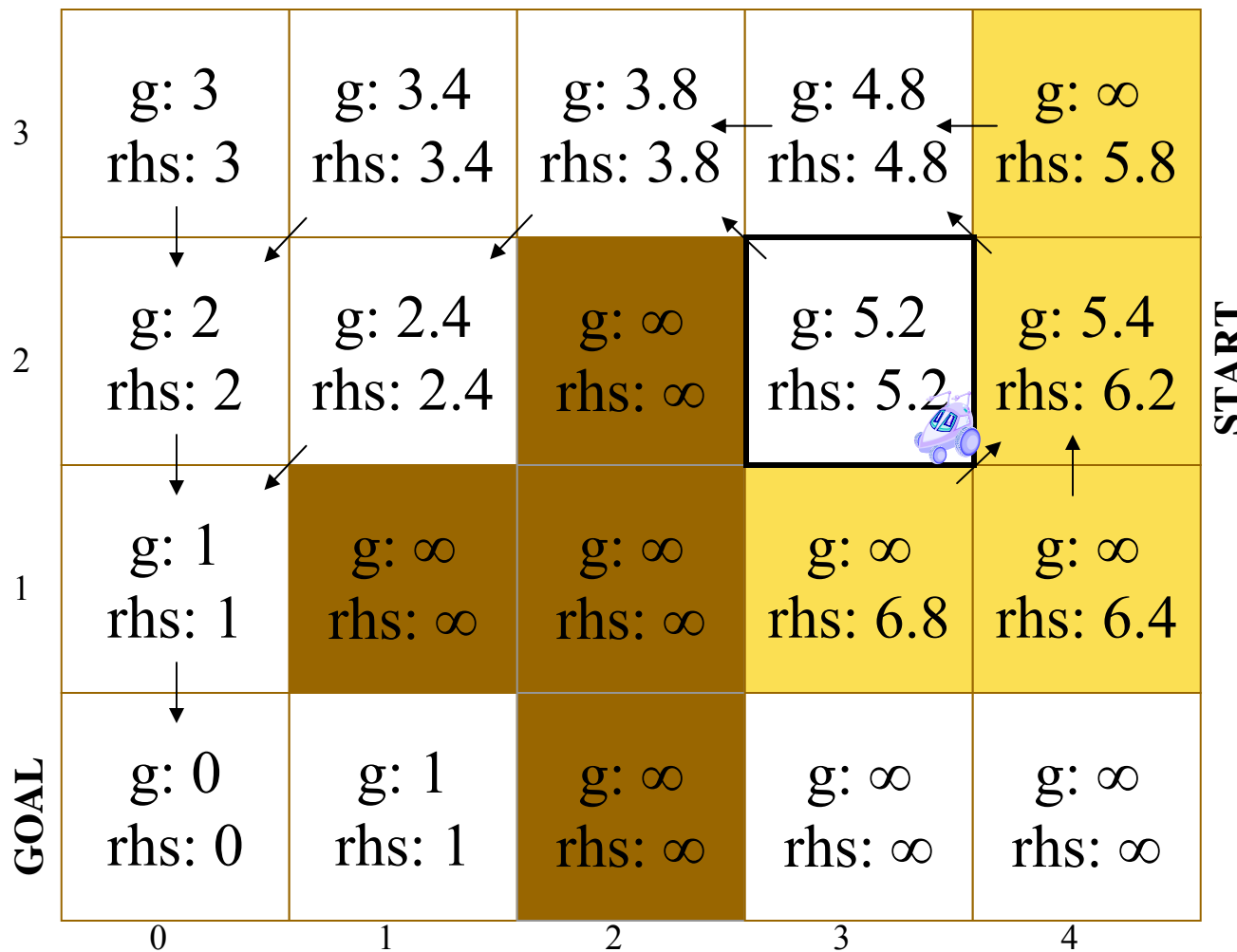
ComputeShortestPath

Because (3,1) was underconsistent when it was popped, we need to call *UpdateVertex()* on it. This results in it being put back onto the open list since it is still inconsistent.

Legend

- Free
- Obstacle
- On open list

D* Lite: Replanning (37)

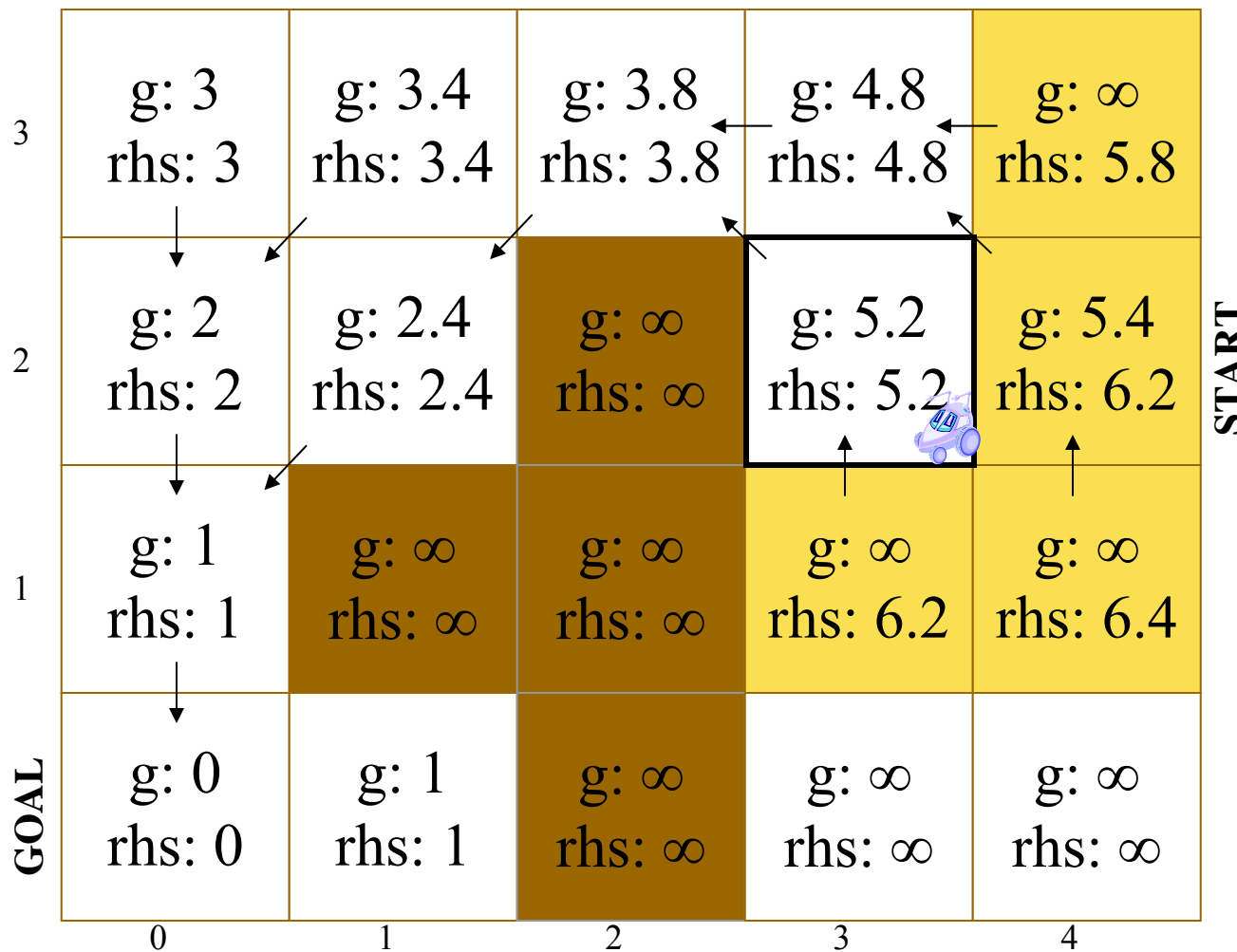


ComputeShortestPath
 Pop minimum item off
 open list - (3,2)
 It's over-consistent
 ($g > rhs$) so set $g = rhs$.

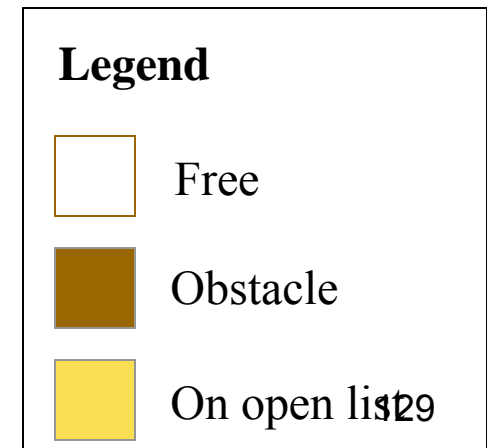
Legend

- Free
- Obstacle
- On open list

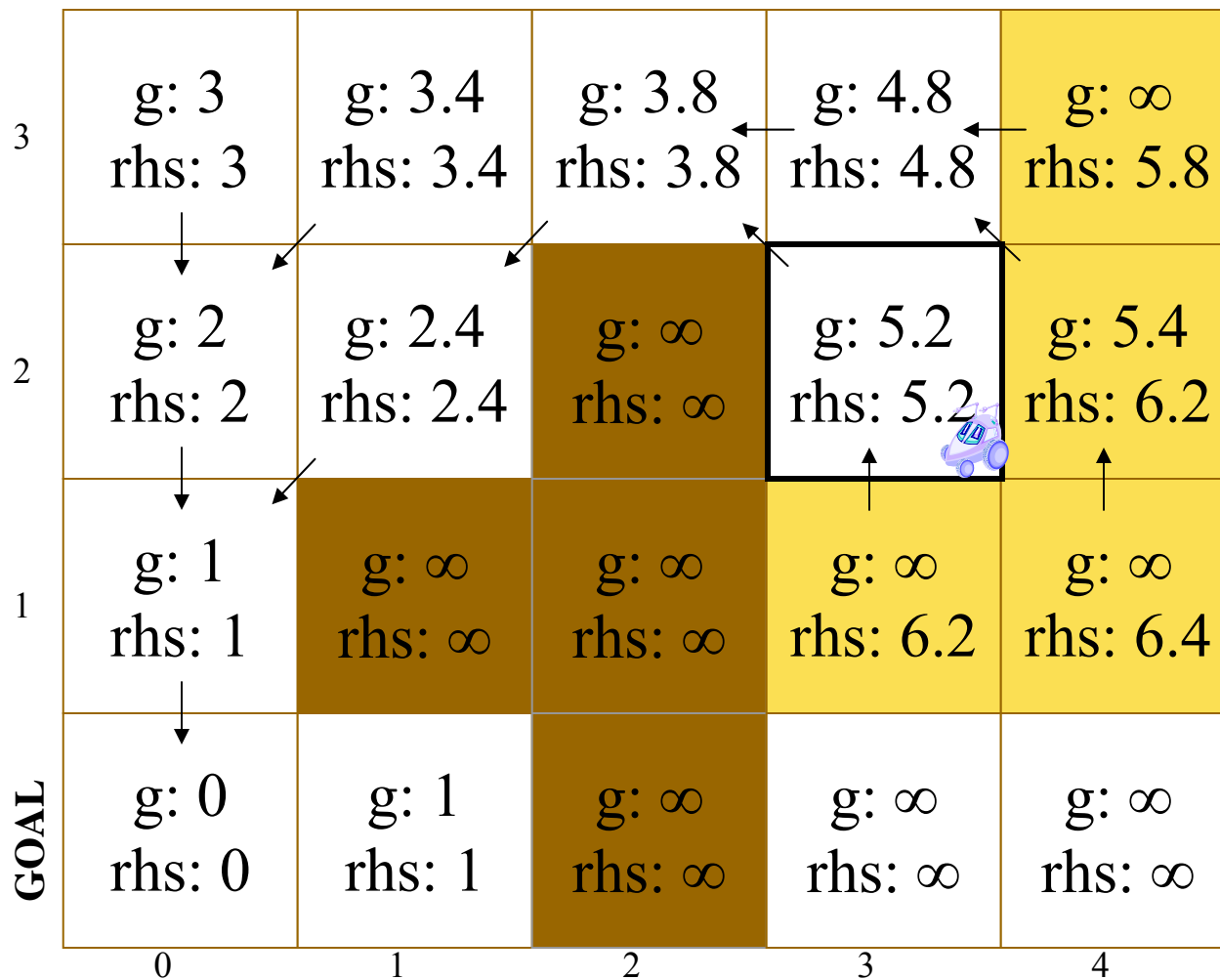
D* Lite: Replanning (38)



ComputeShortestPath
 Expand the popped node and update predecessors. (3,1) gets updated and is still inconsistent.



D* Lite: Replanning (38b)



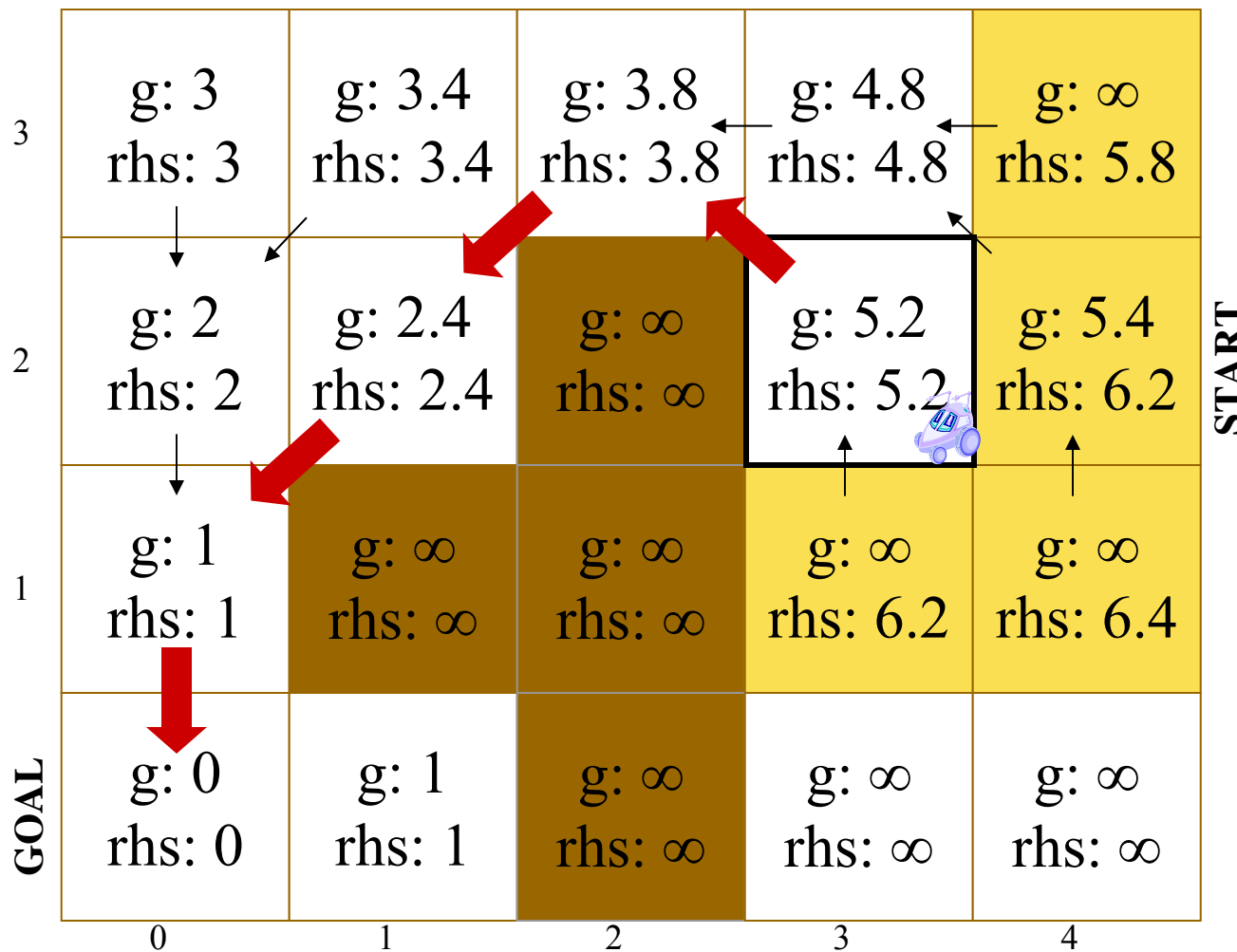
START

ComputeShortestPath
 At this point, the node corresponding to the robot's current position is consistent AND the top key on the open list is not less than the key of this node. So we have an optimal path and can break out of the loop.

Legend

- Free
- Obstacle
- On open list

D* Lite: Following the path (39)



Follow the gradient of g values from the robot's current position.

Legend

- Free
- Obstacle
- On open list

D* Lite: Additional Notes

- In practice, D* Lite works with real valued costs, rather than binary costs (e.g. free/obstacle)
- In practice, the D* Lite search would be focused with an admission heuristic, h whose value would be added to the g and rhs values. In this example $h=0$ for all nodes.
- The example presented the basic, unoptimized version of D* Lite. The final version of D* Lite includes optimizations such as:
 - Updating the rhs value without considering all successors every time
 - Re-focusing the search as the robot moves without reordering the entire open list