Grid and Graph based Path Planning Methods

Jan Faigl

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Lecture 04

B4M36UIR – Artificial Intelligence in Robotics

Overview of the Lecture

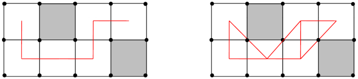
- Part 1 – Grid and Graph based Path Planning Methods
  - Grid-based Planning
  - DT for Path Planning
  - Graph Search Algorithms
  - D* Lite
  - Path Planning based on Reaction-Diffusion Process *Curiosity*

Part I

Part 1 – Grid and Graph based Path Planning Methods

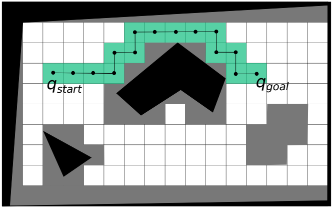
Grid-based Planning

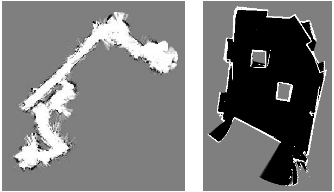
- A subdivision of $\mathcal{C}_{free}$ into smaller cells
- **Grow obstacles** can be simplified by growing borders by a diameter of the robot
- Construction of the planning graph $G = (V, E)$ for V as a set of cells and E as the **neighbor-relations**
  - 4-neighbors and 8-neighbors



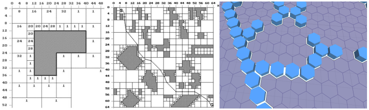
- A grid map can be constructed from the so-called occupancy grid maps

E.g., using thresholding



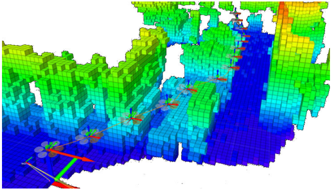


Grid-based Environment Representations



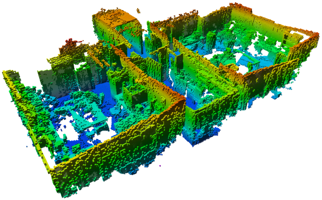
- Hiearchical planning
  - Coarse resolution and re-planning on finer resolution

Holte, R. C. et al. (1996): Hierarchical A *: searching abstraction hierarchies efficiently. AAAI.



- Octree can be used for the map representation
- In addition to squared (or rectangular) grid a hexagonal grid can be used
- 3D grid maps – **octomap**

https://octomap.github.io



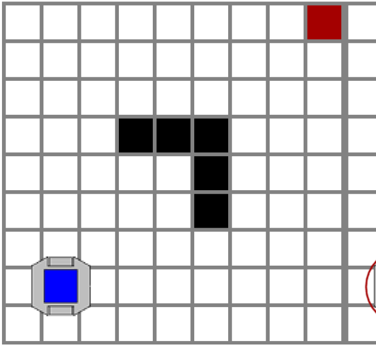
– Memory grows with the size of the environment

– Due to limited resolution it may fail in narrow passages of $\mathcal{C}_{free}$

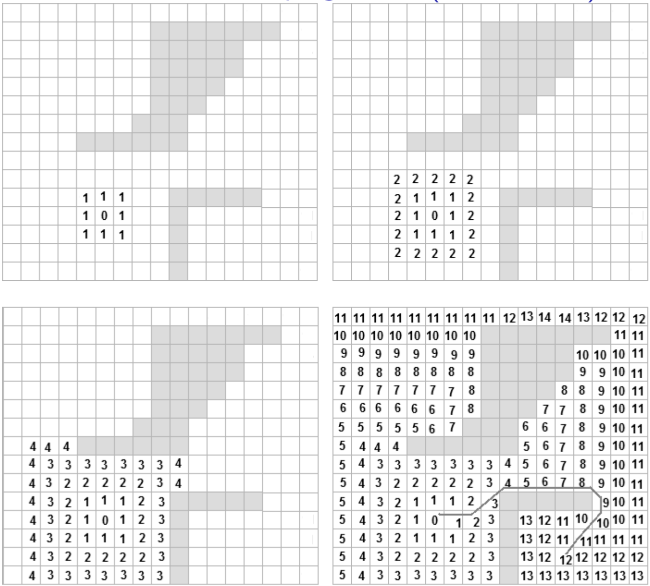
Example of Simple Grid-based Planning

- Wave-front propagation using path simplication

- Initial map with a robot and goal
- Obstacle growing
- Wave-front propagation – "flood fill"
- Find a path using a navigation function
- Path simplification
  - "Ray-shooting" technique combined with **Bresenham's line algorithm**
  - The path is a sequence of "key" cells for avoiding obstacles

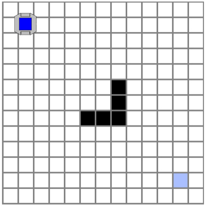


Example – Wave-Front Propagation (Flood Fill)

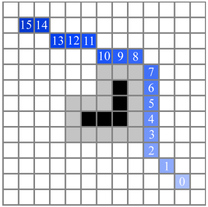


Path Simplification

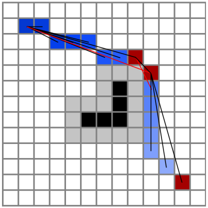
- The initial path is found in a grid using 4-neighborhood
- The rayshoot cast a line into a grid and possible collisions of the robot with obstacles are checked
- The "farthest" cells without collisions are used as "turn" points
- The final path is a sequence of straight line segments



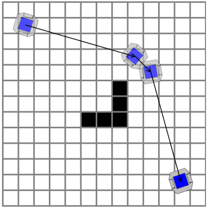
Initial and goal locations



Obtacle growing, wave-front propagation



Ray-shooting



Simplified path

Bresenham's Line Algorithm

- Filling a grid by a line with avoding float numbers
- A line from (x_0, y_0) to (x_1, y_1) is given by $y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$

```
CoordsVector& bresenham(const Coords& pt1, const
      Coords& pt2, CoordsVector& line)
{
   // The pt2 point is not added into line
   int x0 = pt1.c; int y0 = pt1.r;
   int x1 = pt2.c; int y1 = pt2.r;
   Coords p;
   int dx = x1 - x0;
   int dy = y1 - y0;
   int steep = (abs(dy) >= abs(dx));
   if (steep) {
      SWAP(x0, y0);
      SWAP(x1, y1);
      dx = x1 - x0; // recompute Dx, Dy
      dy = y1 - y0;
   }
   int xstep = 1;
   if (dx < 0) {
      xstep = -1;
      dx = -dx;
   }
   int ystep = 1;
   if (dy < 0) {
      ystep = -1;
      dy = -dy;
   }
   int twoDy = 2 * dy;
   int twoDyTwoDx = twoDy - 2 * dx; //2*Dy - 2*Dx
   int e = twoDy - dx; //2*Dy - Dx
   int y = y0;
   int xDraw, yDraw;
   for (int x = x0; x != x1; x += xstep) {
      if (steep) {
         xDraw = y;
         yDraw = x;
      } else {
         xDraw = x;
         yDraw = y;
      }
      p.c = xDraw;
      p.r = yDraw;
      line.push_back(p); // add to the line
      if (e > 0) {
         e += twoDyTwoDx; //E += 2*Dy - 2*Dx
         y = y + ystep;
      } else {
         e += twoDy; //E += 2*Dy
      }
   }
   return line;
}
```

Distance Transform based Path Planning

- For a given goal location and grid map compute a navigational function using *wave-front* algorithm, i.e., a kind of *potential field*
  - The value of the goal cell is set to 0 and all other free cells are set to some very high value
  - For each free cell compute a number of cells towards the goal cell
  - It uses 8-neighbors and distance is the Euclidean distance of the centers of two cells, i.e., EV=1 for orthogonal cells or $EV = \sqrt{2}$ for diagonal cells
  - The values are iteratively computed until the values are changing
  - The value of the cell c is computed as

$$cost(c) = \min_{i=1}^{8} \left(cost(c_i) + EV_{c_i,c} \right),$$

  where c_i is one of the neighboring cells from 8-neighborhood of the cell c
- The algorithm provides a cost map of the path distance from any free cell to the goal cell
- The path is then used following the gradient of the cell cost

Jarvis, R. (2004): Distance Transform Based Visibility Measures for Covert Path Planning in Known but Dynamic Environments

Distance Transform Path Planning

Algorithm 1: Distance Transform for Path Planning

```
for y := 0 to yMax do
    for x := 0 to xMax do
        if goal [x,y] then
            cell [x,y] := 0;
        else
            cell [x,y] := xMax * yMax; //initialization, e.g., pragmatic of the use longest distance as ∞ ;

repeat
    for y := 1 to (yMax - 1) do
        for x := 1 to (xMax - 1) do
            if not blocked [x,y] then
                cell [x,y] := cost(x, y);

    for y := (yMax-1) downto 1 do
        for x := (xMax-1) downto 1 do
            if not blocked [x,y] then
                cell[x,y] := cost(x, y);

until no change;
```

Distance Transform based Path Planning – Impl. 1/2

```
Grid& DT::compute(Grid& grid) const
{
   static const double DIAGONAL = sqrt(2);
   static const double ORTOGONAL = 1;
   const int H = map.H;
   const int W = map.W;
   assert(grid.H == H and grid.W == W, "size");
   bool anyChange = true;
   int counter = 0;
   while (anyChange) {
      anyChange = false;
      for (int r = 1; r < H - 1; ++r) {
         for (int c = 1; c < W - 1; ++c) {
            if (map[r][c] != FREESPACE) {
               continue;
            } //obstacle detected
            double t[4];
            t[0] = grid[r - 1][c - 1] + DIAGONAL;
            t[1] = grid[r - 1][c] + ORTOGONAL;
            t[2] = grid[r - 1][c + 1] + DIAGONAL;
            t[3] = grid[r][c - 1] + ORTOGONAL;
            double pom = grid[r][c];
            for (int i = 0; i < 4; i++) {
               if (pom > t[i]) {
                  pom = t[i];
                  anyChange = true;
               }
            }
            if (anyChange) {
               grid[r][c] = pom;
            }
         }
      }
```

```
      for (int r = H - 2; r > 0; --r) {
         for (int c = W - 2; c > 0; --c) {
            if (map[r][c] != FREESPACE) {
               continue;
            } //obstacle detected
            double t[4];
            t[1] = grid[r + 1][c] + ORTOGONAL;
            t[0] = grid[r + 1][c + 1] + DIAGONAL;
            t[3] = grid[r][c + 1] + ORTOGONAL;
            t[2] = grid[r + 1][c - 1] + DIAGONAL;
            double pom = grid[r][c];
            bool s = false;
            for (int i = 0; i < 4; i++) {
               if (pom > t[i]) {
                  pom = t[i];
                  s = true;
               }
            }
            if (s) {
               anyChange = true;
               grid[r][c] = pom;
            }
         }
      }
      counter++;
   } //end while any change
   return grid;
}
```

A boundary is assumed around the rectangular map

Distance Transform based Path Planning – Impl. 2/2

- The path is retrived by following the minimal value towards the goal using `min8Point()`

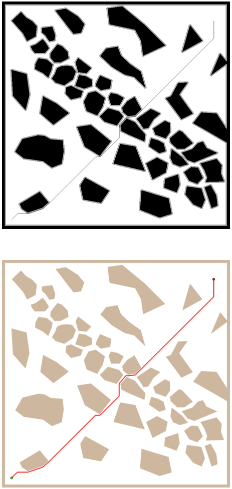
```
Coords& min8Point(const Grid& grid, Coords& p)
{
   double min = std::numeric_limits<double>::max();
   const int H = grid.H;
   const int W = grid.W;
   Coords t;

   for (int r = p.r - 1; r <= p.r + 1; r++) {
      if (r < 0 or r >= H) { continue; }
      for (int c = p.c - 1; c <= p.c + 1; c++) {
         if (c < 0 or c >= W) { continue; }
         if (min > grid[r][c]) {
            min = grid[r][c];
            t.r = r; t.c = c;
         }
      }
   }
   p = t;
   return p;
}
```

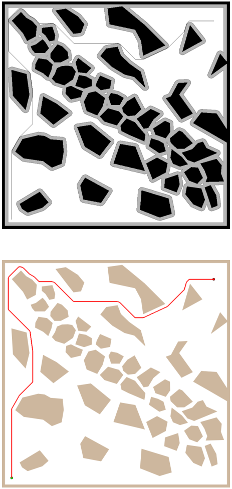
```
CoordsVector& DT::findPath(const Coords& start,
      const Coords& goal, CoordsVector& path)
{
   static const double DIAGONAL = sqrt(2);
   static const double ORTOGONAL = 1;
   const int H = map.H;
   const int W = map.W;
   Grid grid(H, W, H*W); // H*W max grid value
   grid[goal.r][goal.c] = 0;
   compute(grid);

   if (grid[start.r][start.c] >= H*W) {
      WARN("Path has not been found");
   } else {
      Coords pt = start;
      while (pt.r != goal.r or pt.c != goal.c) {
         path.push_back(pt);
         min8Point(grid, pt);
      }
      path.push_back(goal);
   }
   return path;
}
```

DT Example



$\delta = 10$ cm, $L = 27.2$ m



$\delta = 30$ cm, $L = 42.8$ m

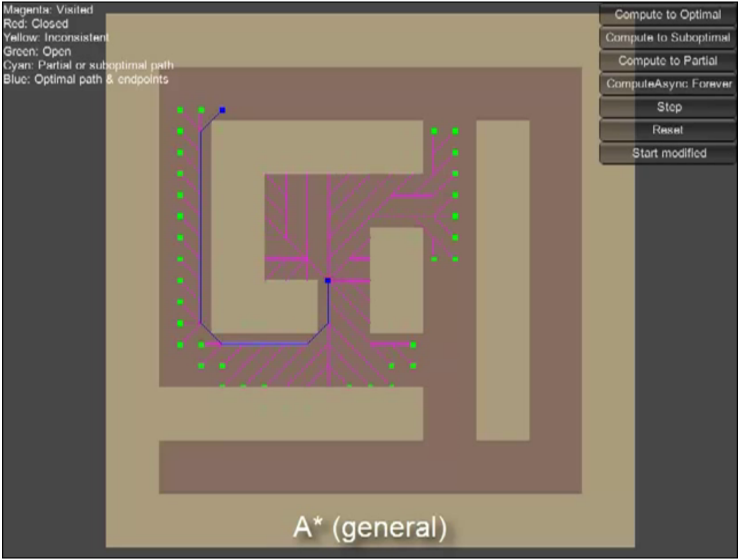
Graph Search Algorithms

- The grid can be considered as a graph and the path can be found using graph search algorithms
- The search algorithms working on a graph are of general use, e.g.
  - Breadth-first search (BSD)
  - Depth first search (DFS)
  - Dijsktra's algorithm,
  - A* algorithm and its variants
- There can be grid based speedups techniques, e.g.,
  - Jump Search Algorithm (JPS) and JPS+
- There are many search algorithm for on-line search, incremental search and with any-time and real-time properties, e.g.,
  - Lifelong Planning A* (LPA*)

    Koenig, S., Likhachev, M. and Furcy, D. (2004): Lifelong Planning A*. AIJ.
  - E-Graphs – Experience graphs

    Phillips, M. et al. (2012): E-Graphs: Bootstrapping Planning with Experience Graphs. RSS.

Examples of Graph/Grid Search Algorithms

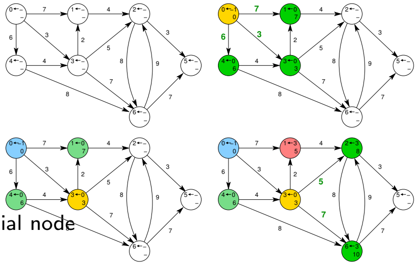


https://www.youtube.com/watch?v=U2XNjCoKZjM.mp4

Dijkstra's Algorithm

- Dijsktra's algorithm determines paths as iterative update of the cost of the shortest path to the particular nodes

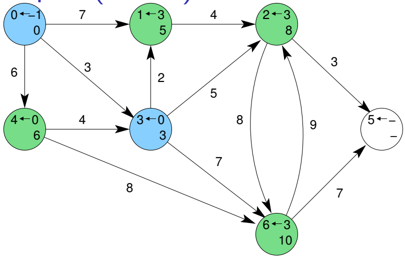
Edsger W. Dijkstra, 1956



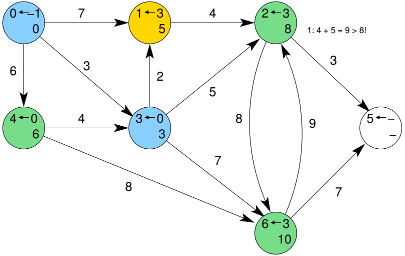
- Let start with the initial cell (node) with the cost set to 0 and update all successors
- Select the node
  - with a path from the initial node
  - and has a lower cost
- Repeat until there is a reachable node
  - I.e., a node with a path from the initial node
  - has a cost and parent (*green nodes*).

The cost of nodes can only decrease (edge cost is positive). Therefore, for a node with the currently lowest cost, there cannot be a shorter path from the initial node.

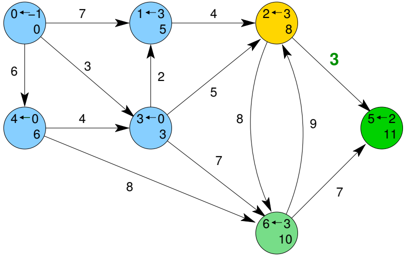
Example (cont.)



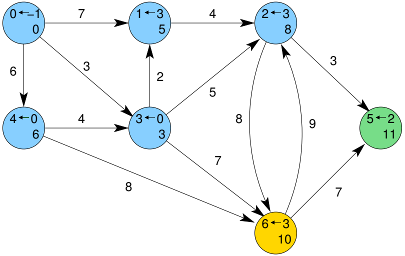
1: After the expansion, the shortest path to the node 2 is over the node 3



2: There is not shorter path to the node 2 over the node 1



3: After the expansion, there is a new path to the node 5



4: The path does not improve for further expansions

Dijkstra's Algorithm

Algorithm 2: Dijkstra's algorithm

```
Initialize(s_start);                 /* g(s) := ∞; g(s_start) := 0 */
PQ.push(s_start, g(s_start));
while (not PQ.empty?) do
    s := PQ.pop();
    foreach s' ∈ Succ(s) do
        if s' in PQ then
            if g(s') > g(s) + cost(s, s') then
                g(s') := g(s) + cost(s, s');
                PQ.update(s', g(s'));
        else if s' ∉ CLOSED then
            g(s') := g(s) + cost(s, s');
            PQ.push(s', g(s'));
    CLOSED := CLOSED ⋃ {s};
```

Dijkstra's Algorithm – Impl.

```
dij->nodes[dij->start_node].cost = 0; // init
void *pq = pq_alloc(dij->num_nodes); // set priority queue
int cur_label;
pq_push(pq, dij->start_node, 0);
while ( !pq_is_empty(pq) && pq_pop(pq, &cur_label)) {
   node_t *cur = &(dij->nodes[cur_label]); // remember the current node
   for (int i = 0; i < cur->edge_count; ++i) { // all edges of cur
      edge_t *edge = &(dij->graph->edges[cur->edge_start + i]);
      node_t *to = &(dij->nodes[edge->to]);
      const int cost = cur->cost + edge->cost;
      if (to->cost == -1) { // node to has not been visited
         to->cost = cost;
         to->parent = cur_label;
         pq_push(pq, edge->to, cost); // put node to the queue
      } else if (cost < to->cost) { // node already in the queue
         to->cost = cost; // test if the cost can be reduced
         to->parent = cur_label; // update the parent node
         pq_update(pq, edge->to, cost); // update the priority queue
      }
   } // loop for all edges of the cur node
} // priority queue empty
pq_free(pq); // release memory
```

A* Algorithm

- A* uses a user-defined h-values (heuristic) to focus the search

  Peter Hart, Nils Nilsson, and Bertram Raphael, 1968

  - Prefer expansion of the node n with the lowest value

    $$f(n) = g(n) + h(n),$$

    where $g(n)$ is the cost (path length) from the start to n and $h(n)$ is the estimated cost from n to the goal

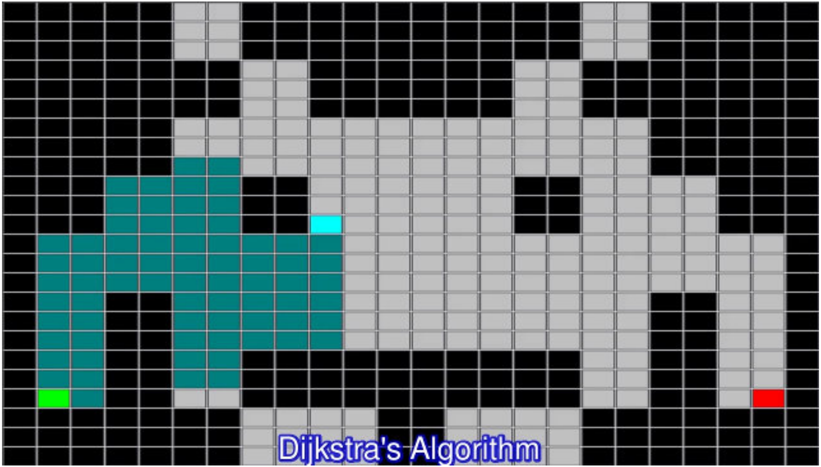
- h-values approximate the goal distance from particular nodes
- **Admissiblity condition** – heuristic always underestimate the remaining cost to reach the goal
  - Let $h^*(n)$ be the true cost of the optimal path from n to the goal
  - Then $h(n)$ is **admissible** if for all n: $h(n) \leq h^*(n)$
  - E.g., Euclidean distance is admissible
    - A straight line will always be the shortest path
- Dijkstra's algorithm – $h(n) = 0$

A* Implementation Notes

- The most costly operations of A* are
  - Insert and lookup an element in the **closed list**
  - Insert element and get minimal element (according to $f()$ value) from the **open list**
- The **closed list** can be efficiently implemented as a **hash set**
- The **open list** is usually implemented as a **priority queue**, e.g.,
  - Fibonacii heap, binomial heap, k-level bucket
  - **binary heap** is usually sufficient ($O(logn)$)
- Forward A*
  1. Create a search tree and initiate it with the start location
  2. Select generated but not yet expanded state s with the smallest f-value, $f(s) = g(s) + h(s)$
  3. `Stop` if s is the goal
  4. Expand the state s
  5. `Goto` Step 2

Similar to Dijsktra's algorithm but it used $f(s)$ with heuristic $h(s)$ instead of pure $g(s)$

Dijsktra's vs A* vs Jump Point Search (JPS)



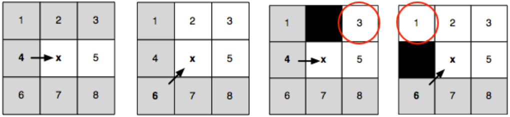
https://www.youtube.com/watch?v=ROG4Ud08lLY

Jump Point Search Algorithm for Grid-based Path Planning

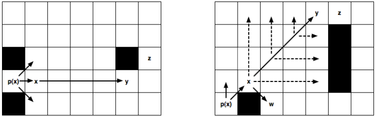
- **Jump Point Search** (JPS) algorithm is based on a macro operator that identifies and selectively expands only certain nodes (**jump points**)

Harabor, D. and Grastien, A. (2011): Online Graph Pruning for Pathfinding on Grid Maps. AAAI.

- Natural neighbors after neighbor prunning with forced neighbors because of obstacle



- Intermediate nodes on a path connecting two jump points are never expanded



- No preprocessing and no memory overheads while it speeds up A*

https://harablog.wordpress.com/2011/09/07/jump-point-search/

- JPS+ – optimized preprocessed version of JPS with goal bounding

https://github.com/SteveRabin/JPSPlusWithGoalBounding

http://www.gdcvault.com/play/1022094/JPS-Over-100x-Faster-than

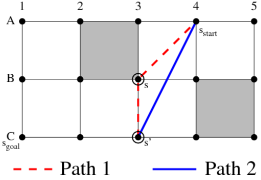
Theta* – Any-Angle Path Planning Algorithm

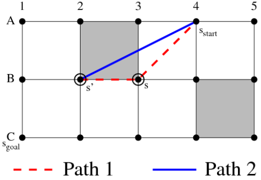
- **Any-angle path planning algorithms** simplify the path during the search
- **Theta*** is an extension of A* with `LineOfSight()`

Nash, A., Daniel, K, Koenig, S. and Felner, A. (2007): Theta*: Any-Angle Path Planning on Grids. AAAI.

Algorithm 3: Theta* Any-Angle Planning

if *LineOfSight(parent(s), s')* **then**
  /* Path 2 – any-angle path */
  if *g(parent(s))+ c(parent(s), s') < g(s')* **then**
    parent(s') := parent(s);
    g(s') := g(parent(s)) + c(parent(s), s');
else
  /* Path 1 – A* path */
  if *g(s) + c(s,s') < g(s')* **then**
    parent(s'):= s;
    g(s') := g(s) + c(s,s');





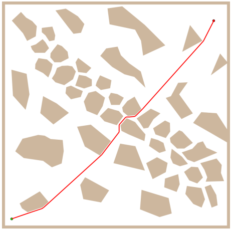
- Path 2: considers path from start to parent(s) and from parent(s) to s' if s' has line-of-sight to parent(s)

http://aigamedev.com/open/tutorials/theta-star-any-angle-paths/

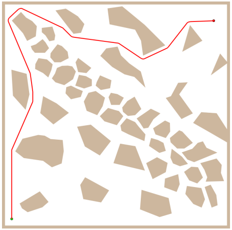
Theta* Any-Angle Path Planning Examples

- Example of found paths by the Theta* algorithm for the same problems as for the DT-based examples on Slide 16

Both algorithms implemented in C++



$\delta = 10$ cm, $L = 26.3$ m



$\delta = 30$ cm, $L = 40.3$ m

The same path planning problems solved by DT (without path smoothing) have $L_{\delta=10} = 27.2$ m and $L_{\delta=30} = 42.8$ m, while DT seems to be significantly faster

---

- **Lazy Theta*** – reduces the number of line-of-sight checks

Nash, A., Koenig, S. and Tovey, C. (2010): Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. AAAI.

http://aigamedev.com/open/tutorial/lazy-theta-star/

A* Variants – Online Search

- The state space (map) may not be known exactly in advance
  - Environment can **dynamically** change
  - True travel costs are **experienced** during the path execution
- Repeated A* searches can be computationally demanding
- **Incremental heuristic search**
  - Repeated planning of the path from the current state to the goal
  - Planning under the **free-space** assumption
  - **Reuse** information from the previous searches (**closed list** entries):
    - Focused Dynamic A* (**D***) – h^* is based on **traversability**, it has been used, e.g., for the Mars rover "Opportunity"

      Stentz, A. (1995): The Focussed D* Algorithm for Real-Time Replanning. IJCAI.
    - **D* Lite** – similar to D*

      Koenig, S. and Likhachev, M. (2005): Fast Replanning for Navigation in Unknown Terrain. T-RO.
- **Real-Time Heuristic Search**
  - Repeated planning with limited **look-ahead** – suboptimal but fast
    - Learning Real-Time A* (**LRTA***)

      Korf, E. (1990): Real-time heuristic search. JAI
    - Real-Time Adaptive A* (**RTAA***)

      Koenig, S. and Likhachev, M. (2006): Real-time adaptive A*. AAMAS.

Real-Time Adaptive A* (RTAA*)

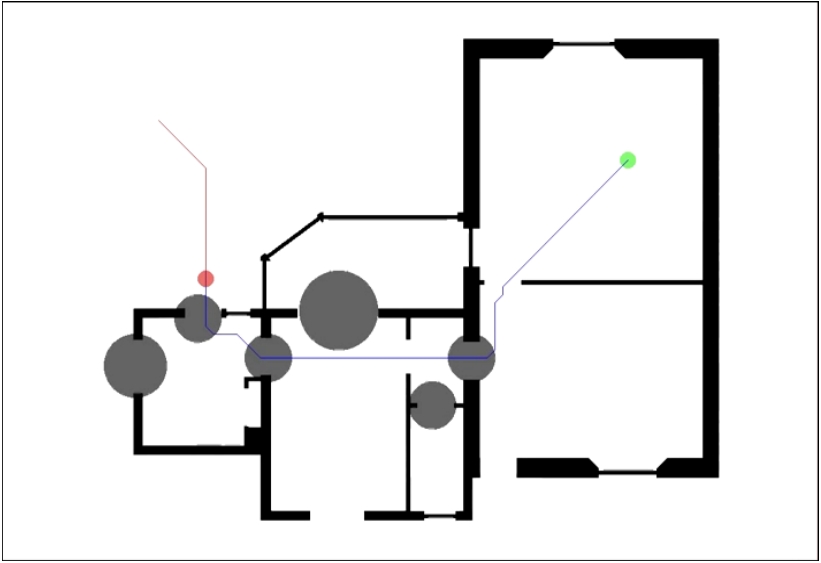
- Execute A* with limited **look-ahead**
- Learns better informed heuristic from the experience, initially $h(s)$, e.g., Euclidean distance
- Look-ahead defines **trade-off** between optimality and computational cost
  - `astar(lookahead)`

  A* expansion as far as "lookahead" nodes and it terminates with the state s'

```
while (s_curr ∉ GOAL) do
    astar(lookahead);
    if s' = FAILURE then
        return FAILURE;
    for all s ∈ CLOSED do
        H(s) := g(s') + h(s') - g(s);
    execute(plan); // perform one step
return SUCCESS;
```

s' is the last state expanded during the previous A* search

D* Lite – Demo



https://www.youtube.com/watch?v=X5a149nSE9s

D* Lite Overview

- It is similar to D*, but it is based on **Lifelong Planning A***

  Koenig, S. and Likhachev, M. (2002): D* Lite. AAAI.

- It searches from the goal node to the start node, i.e., g-values estimate the goal distance
- Store pending nodes in a priority queue
- Process nodes in order of increasing objective function value
- Incrementally repair solution paths when changes occur
- 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
- **Consistency**
  - Consistent – $g = rhs$
  - Inconsistent – $g \neq rhs$
- Inconsistent nodes are stored in the priority queue (open list) for processing

D* Lite: Cost Estimates

- *rhs* of the node u is computed based on g of its successors in the graph and the transition costs of the edge to those successors

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

- The key/priority of a node s on the open list is the minimum of $g(s)$ and $rhs(s)$ plus a focusing heuristic h

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

  - The first term is used as the primary key
  - The second term is used as the secondary key for tie-breaking

D* Lite Algorithm

- **Main** – repeat until the robot reaches the goal (or $g(s_{start}) = \infty$ there is no path)

```
Initialize();
ComputeShortestPath();
while (s_start ≠ s_goal) do
    s_start = argmin_{s' ∈ Succ(s_start)} (c(s_start, s') + g(s'));
    Move to s_start;
    Scan the graph for changed edge costs;
    if any edge cost changed perform then
        foreach directed edges (u, v) with changed edge costs do
            Update the edge cost c(u, v);
            UpdateVertex(u);
        foreach s ∈ U do
            U.Update(s, CalculateKey(s));
        ComputeShortestPath();
```

Procedure Initialize

```
U = 0;
foreach s ∈ S do
    rhs(s) := g(s) := ∞;
rhs(s_goal) := 0;
U.Insert(s_goal, CalculateKey(s_goal));
```

D* Lite Algorithm – ComputeShortestPath()

Procedure ComputeShortestPath

while $U.TopKey() < CalculateKey(s_{start})$ OR $rhs(s_{start}) \neq g(s_{start})$ **do**
  u := U.Pop();
  if $g(u) > rhs(u)$ **then**
    $g(u) := rhs(u)$;
    foreach $s \in Pred(u)$ **do** UpdateVertex(s);
  else
    $g(u) := \infty$;
    foreach $s \in Pred(u) \bigcup \{u\}$ **do** UpdateVertex(s);

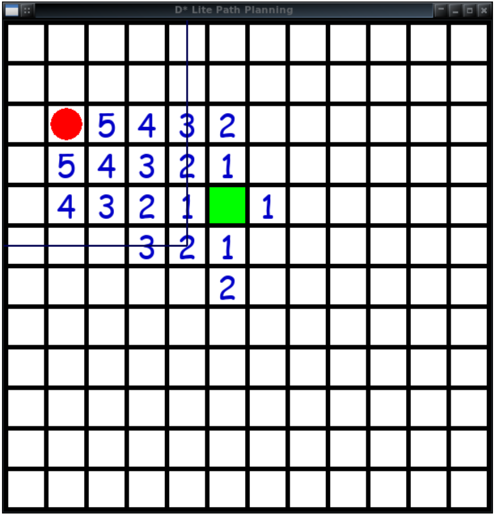
Procedure UpdateVertex

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

Procedure CalculateKey

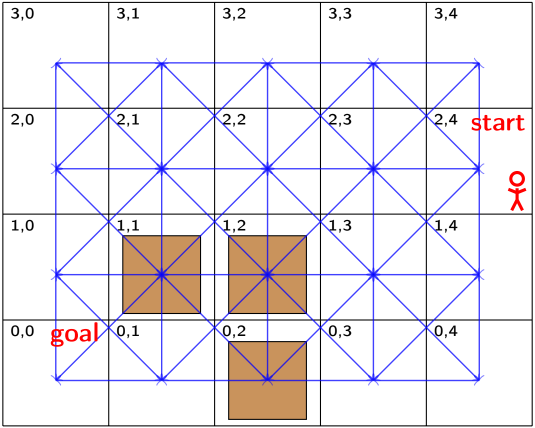
return $[\min(g(s), rhs(s)) + h(s_{start}, s); \min(g(s), rhs(s))]$

D* Lite – Demo



https://github.com/mdeyo/d-star-lite

D* Lite – Example



Legend

Free node	Obstacle node
On open list	Active node

- A grid map of the environment (what is actually known)
- 8-connected graph superimposed on the grid (bidirectional)
- Focusing heuristic is not used ($h = 0$)

- Transition costs
  - Free space – Free space: 1.0 and 1.4 (for diagonal edge)
  - From/to obstacle: ∞

D* Lite – Example Planning (1)

3,0 g: ∞ rhs: ∞	3,1 g: ∞ rhs: ∞	3,2 g: ∞ rhs: ∞	3,3 g: ∞ rhs: ∞	3,4 g: ∞ rhs: ∞
2,0 g: ∞ rhs: ∞	2,1 g: ∞ rhs: ∞	2,2 g: ∞ rhs: ∞	2,3 g: ∞ rhs: ∞	2,4 start g: ∞ rhs: ∞
1,0 g: ∞ rhs: ∞	1,1 g: ∞ rhs: ∞	1,2 g: ∞ rhs: ∞	1,3 g: ∞ rhs: ∞	1,4 g: ∞ rhs: ∞
0,0 goal g: ∞ rhs: 0	0,1 g: ∞ rhs: ∞	0,2 g: ∞ rhs: ∞	0,3 g: ∞ rhs: ∞	0,4 g: ∞ rhs: ∞

Legend

Free node	Obstacle node
On open list	Active node

Initialization

- Set $rhs = 0$ for the goal
- Set $rhs = g = \infty$ for all other nodes

D* Lite – Example Planning (2)

3,0 g: ∞ rhs: ∞	3,1 g: ∞ rhs: ∞	3,2 g: ∞ rhs: ∞	3,3 g: ∞ rhs: ∞	3,4 g: ∞ rhs: ∞
2,0 g: ∞ rhs: ∞	2,1 g: ∞ rhs: ∞	2,2 g: ∞ rhs: ∞	2,3 g: ∞ rhs: ∞	2,4 start g: ∞ rhs: ∞
1,0 g: ∞ rhs: ∞	1,1 g: ∞ rhs: ∞	1,2 g: ∞ rhs: ∞	1,3 g: ∞ rhs: ∞	1,4 g: ∞ rhs: ∞
0,0 goal g: ∞ rhs: 0	0,1 g: ∞ rhs: ∞	0,2 g: ∞ rhs: ∞	0,3 g: ∞ rhs: ∞	0,4 g: ∞ rhs: ∞

Legend

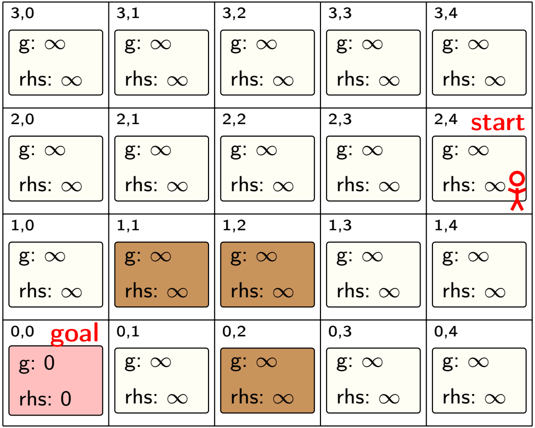
Free node	Obstacle node
On open list	Active node

Initialization

- Put the goal to the open list

  It is inconsistent

D* Lite – Example Planning (3)



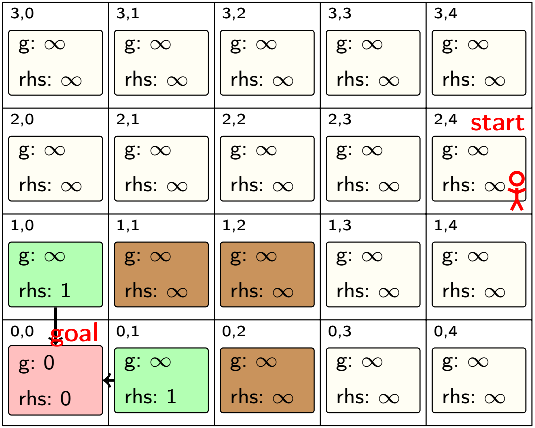
Legend

Free node | Obstacle node | On open list | Active node

ComputeShortestPath

- Pop the minimum element from the open list (goal)
- It is over-consistent ($g > rhs$), therefore set $g = rhs$

D* Lite – Example Planning (4)



Legend

Free node	Obstacle node
On open list	Active node

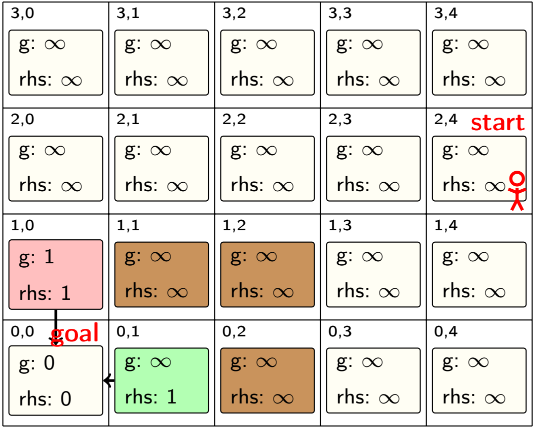
ComputeShortestPath

- Expand popped node (`UpdateVertex()` on all its predecessors)
- This computes the *rhs* values for the predecessors
- Nodes that become inconsistent are added to the open list

Small black arrows denote the node used for computing the *rhs* value, i.e., using the respective transition cost

- The *rhs* value of (1,1) is ∞ because the transition to obstacle has cost ∞

D* Lite – Example Planning (5)



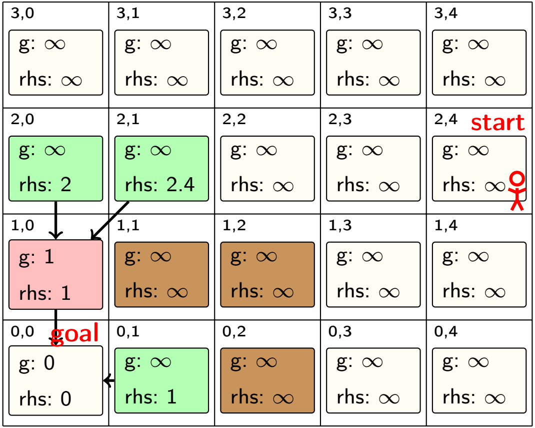
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Pop the minimum element from the open list (1,0)
- It is over-consistent ($g > rhs$) set $g = rhs$

D* Lite – Example Planning (6)



Legend

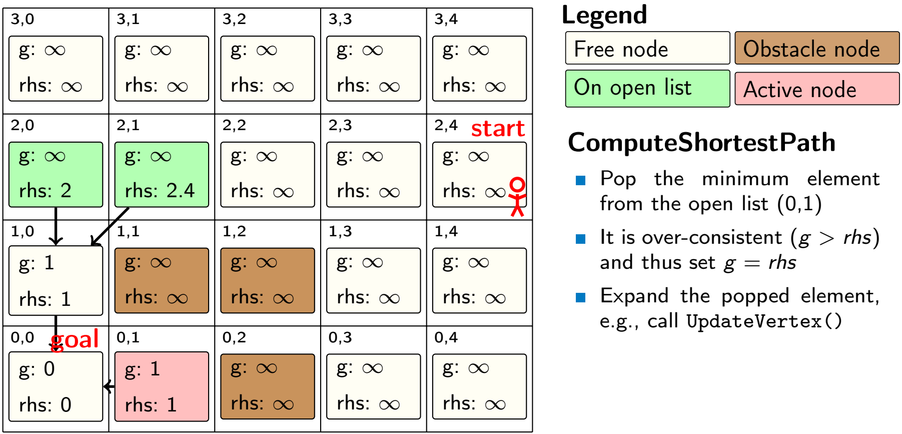
Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Expand the popped node (`UpdateVertex()` on all predecessors in the graph)
- Compute *rhs* values of the predecessors accordingly
- Put them to the open list if they become inconsistent

- The *rhs* value of (0,0), (1,1) does not change
- They do not become inconsistent and thus they are not put on the open list

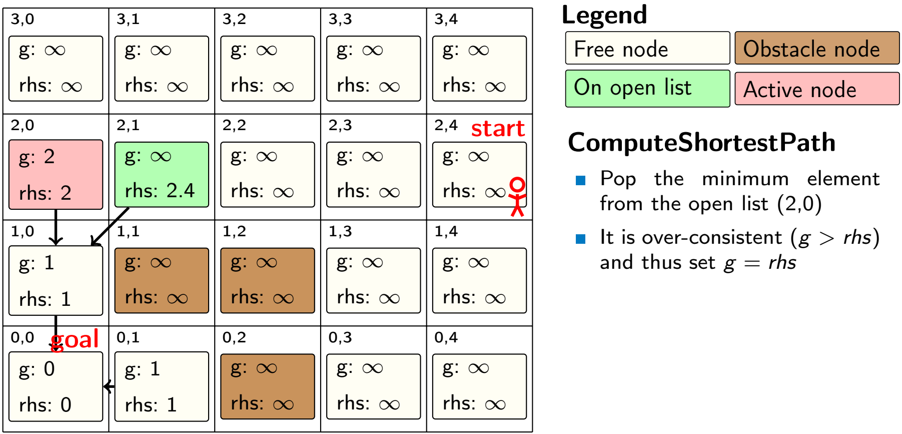
D* Lite – Example Planning (7)



ComputeShortestPath

- Pop the minimum element from the open list (0,1)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element, e.g., call `UpdateVertex()`

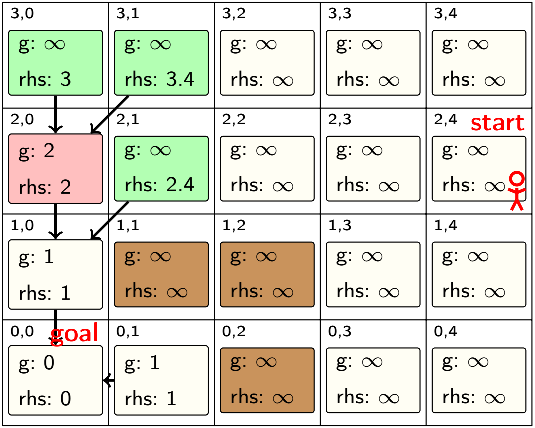
D* Lite – Example Planning (8)



ComputeShortestPath

- Pop the minimum element from the open list (2,0)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$

D* Lite – Example Planning (9)



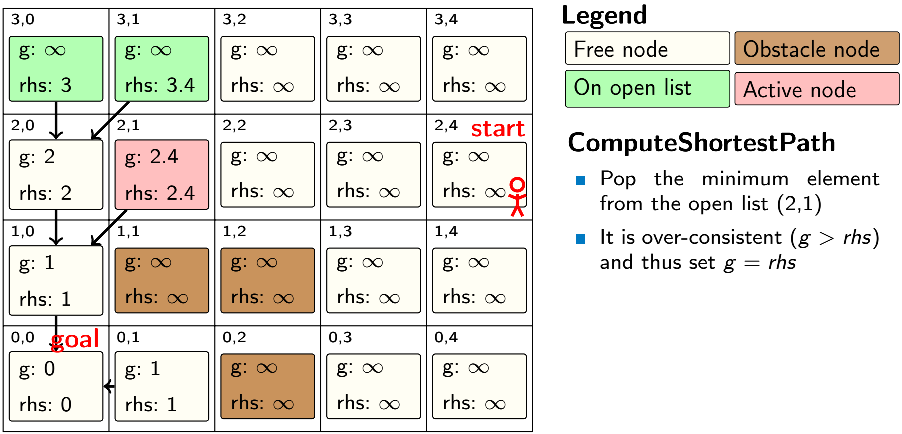
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Expand the popped element and put the predecessors that become inconsistent onto the open list

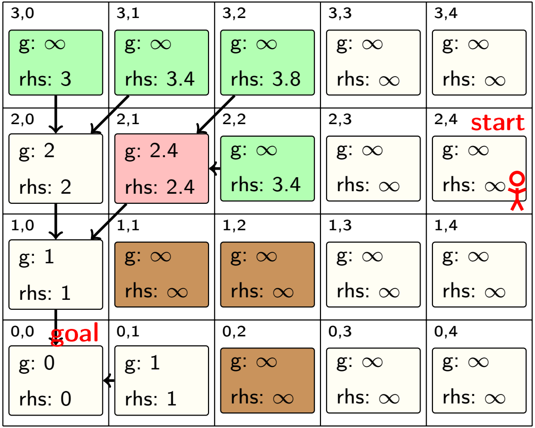
D* Lite – Example Planning (10)



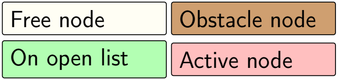
ComputeShortestPath

- Pop the minimum element from the open list (2,1)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$

D* Lite – Example Planning (11)



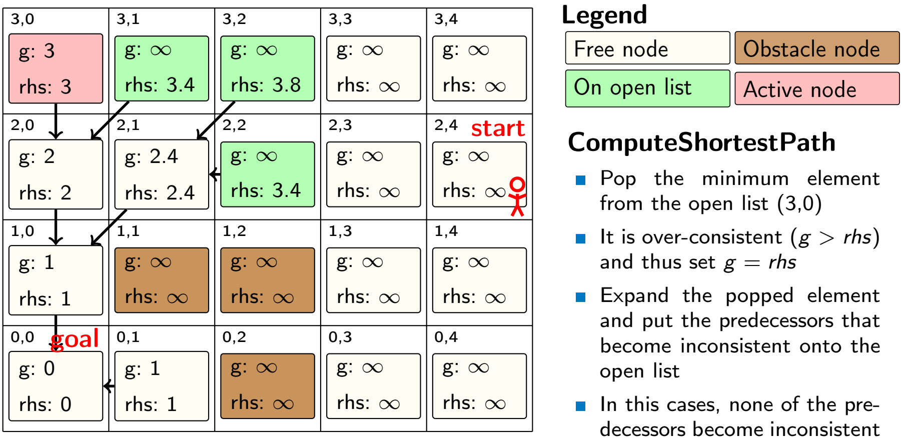
Legend



ComputeShortestPath

- Expand the popped element and put the predecessors that become inconsistent onto the open list

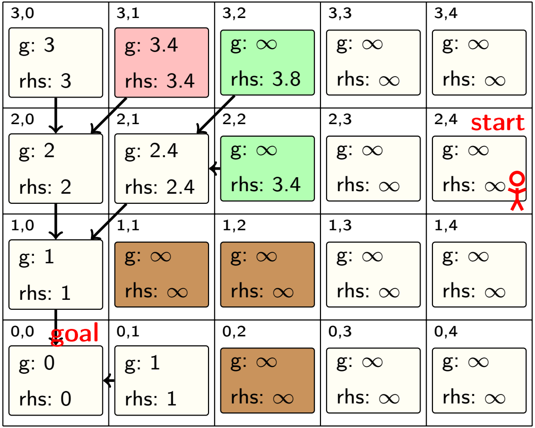
D* Lite – Example Planning (12)



ComputeShortestPath

- Pop the minimum element from the open list (3,0)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element and put the predecessors that become inconsistent onto the open list
- In this cases, none of the predecessors become inconsistent

D* Lite – Example Planning (13)



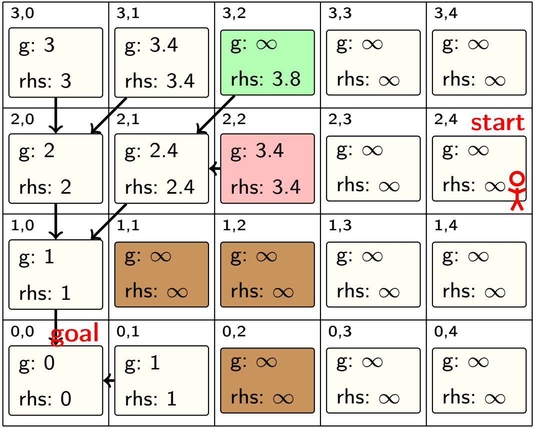
Legend

Free node	Obstacle node
On open list	Active node

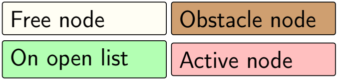
ComputeShortestPath

- Pop the minimum element from the open list (3,0)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element and put the predecessors that become inconsistent onto the open list
- In this cases, none of the predecessors become inconsistent

D* Lite – Example Planning (14)



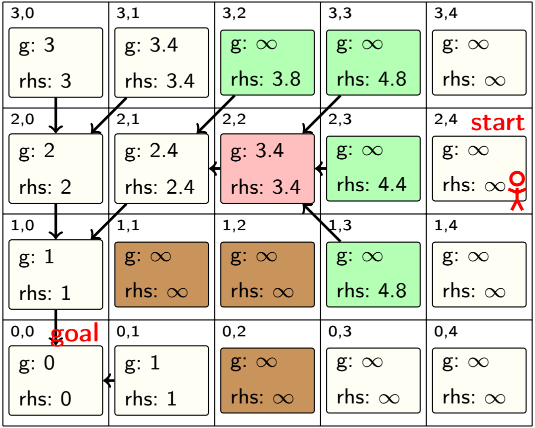
Legend



ComputeShortestPath

- Pop the minimum element from the open list (2,2)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$

D* Lite – Example Planning (15)



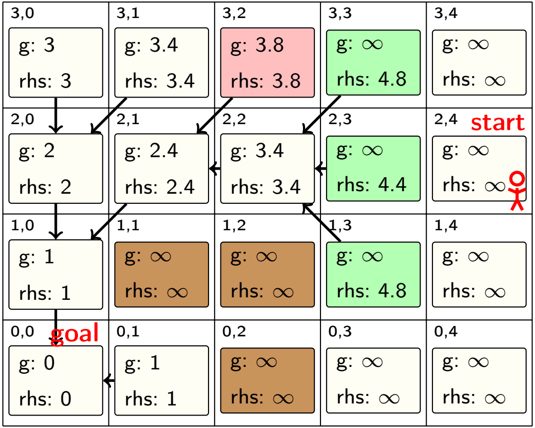
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Expand the popped element and put the predecessors that become inconsistent onto the open list, i.e., (3,2), (3,3), (2,3)

D* Lite – Example Planning (16)



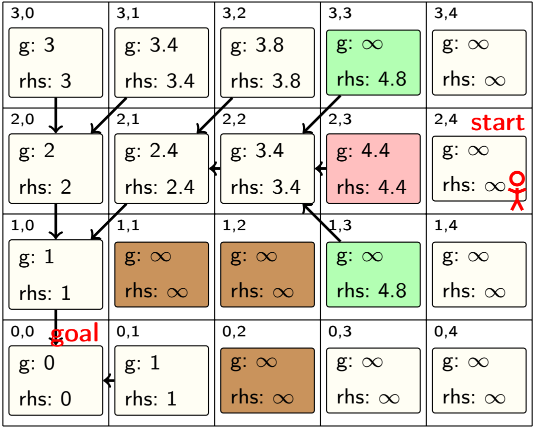
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Pop the minimum element from the open list (3,2)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element and put the predecessors that become inconsistent onto the open list
- In this cases, none of the predecessors become inconsistent

D* Lite – Example Planning (17)



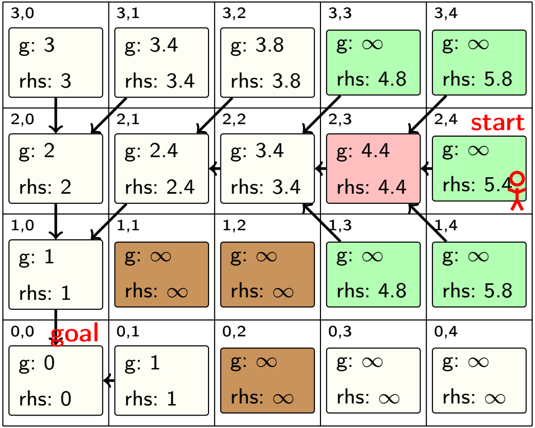
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Pop the minimum element from the open list (2,3)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$

D* Lite – Example Planning (18)



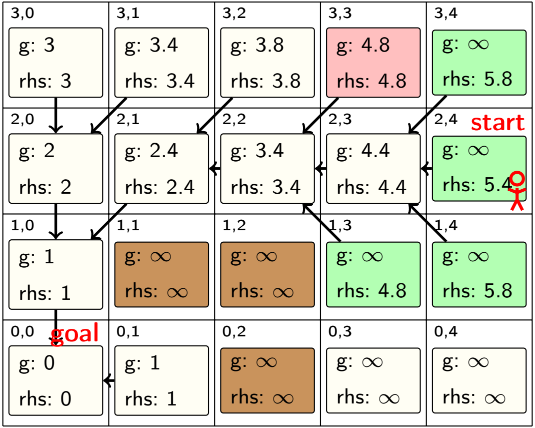
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Expand the popped element and put the predecessors that become inconsistent onto the open list, i.e., (3,4), (2,4), (1,4)
- The start node is on the open list
- However, the search does not finish at this stage
- There are still inconsistent nodes (on the open list) with a lower value of *rhs*

D* Lite – Example Planning (19)



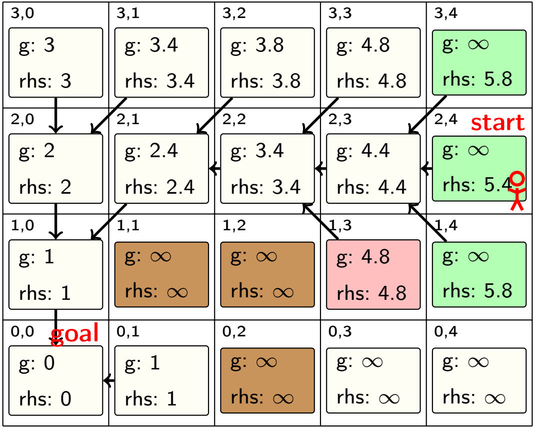
Legend

Free node	Obstacle node
On open list	Active node

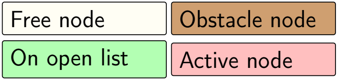
ComputeShortestPath

- Pop the minimum element from the open list (3,2)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element and put the predecessors that become inconsistent onto the open list
- In this cases, none of the predecessors become inconsistent

D* Lite – Example Planning (20)



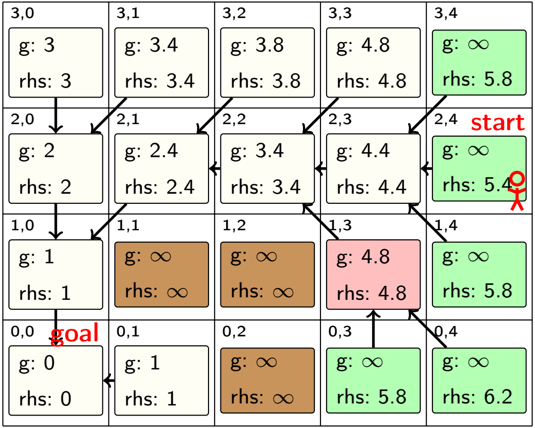
Legend



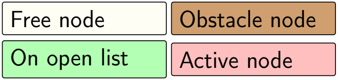
ComputeShortestPath

- Pop the minimum element from the open list (1,3)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$

D* Lite – Example Planning (21)



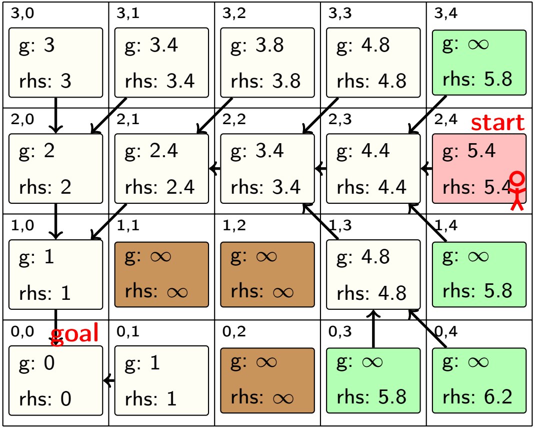
Legend



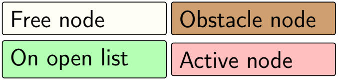
ComputeShortestPath

- Expand the popped element and put the predecessors that become inconsistent onto the open list, i.e., (0,3) and (0,4)

D* Lite – Example Planning (22)



Legend

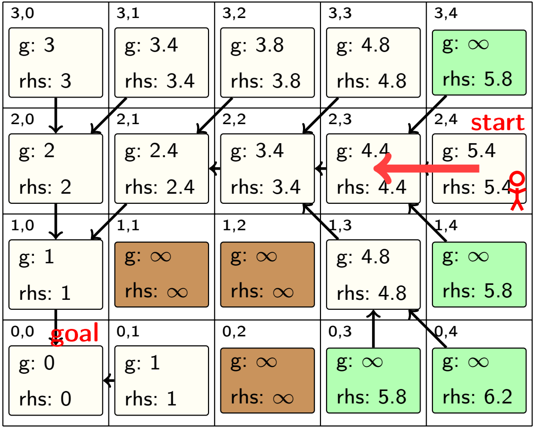


ComputeShortestPath

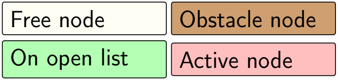
- Pop the minimum element from the open list (2,4)
- It is over-consistent ($g > rhs$) and thus set $g = rhs$
- Expand the popped element and put the predecessors that become inconsistent (none in this case) onto the open list

- The **start** node becomes consistent and the top key on the open list is not less than the key of the start node
- An optimal path is found and the loop of the `ComputeShortestPath` is breaked

D* Lite – Example Planning (23)

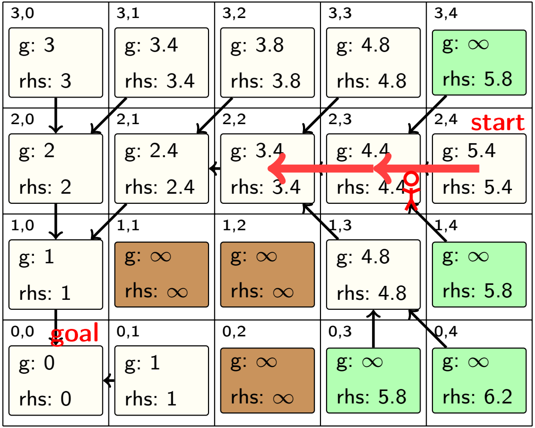


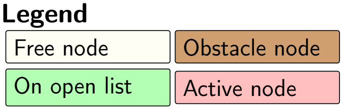
Legend



- Follow the gradient of g values from the start node

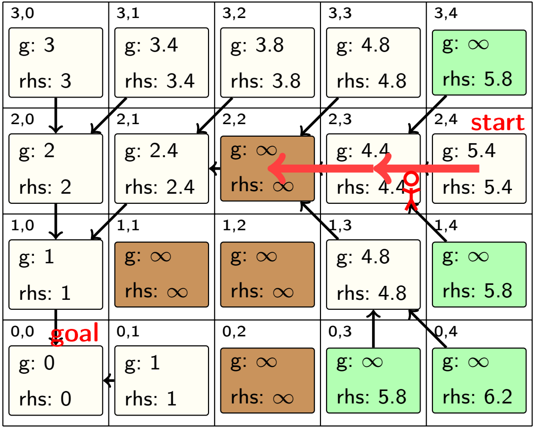
D* Lite – Example Planning (24)





- Follow the gradient of g values from the start node

D* Lite – Example Planning (25)

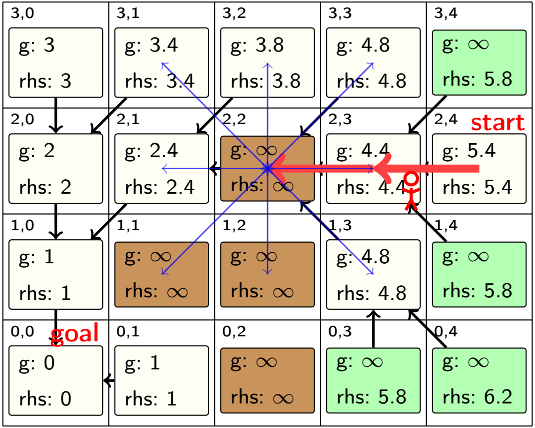


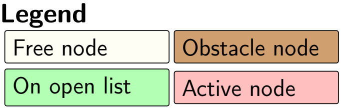
Legend

Free node	Obstacle node
On open list	Active node

- A new obstacle is detected during the movement from (2,3) to (2,2)
- **Replanning** is needed!

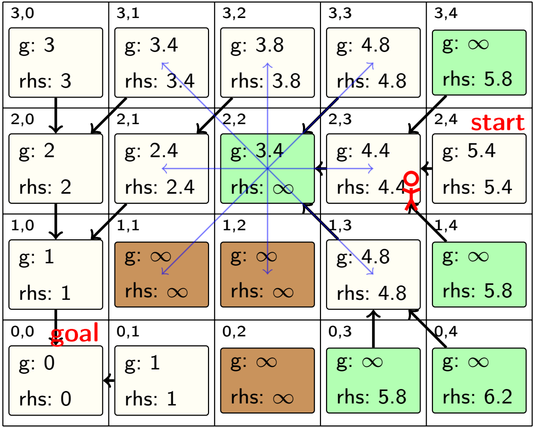
D* Lite – Example Planning (25 update)





- All directed edges with changed edge, we need to call the `UpdateVertex()`
- All edges into and out of (2,2) have to be considered

D* Lite – Example Planning (26 update 1/2)



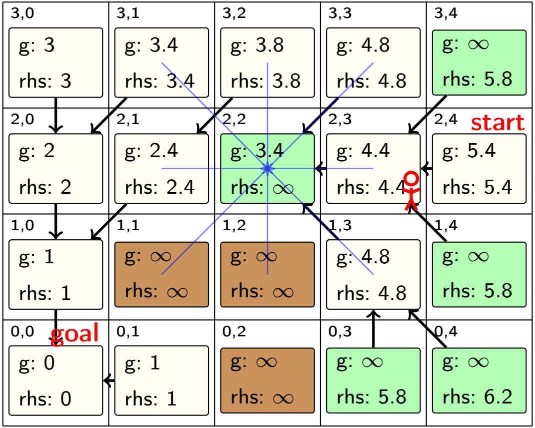
Legend

Free node	Obstacle node
On open list	Active node

Update Vertex

- Outgoing edges from (2,2)
- Call `UpdateVertex()` on (2,2)
- The transition costs are now ∞ because of obstacle
- Therefore the *rhs* $= \infty$ and (2,2) becomes inconsistent and it is put on the open list

D* Lite – Example Planning (26 update 2/2)



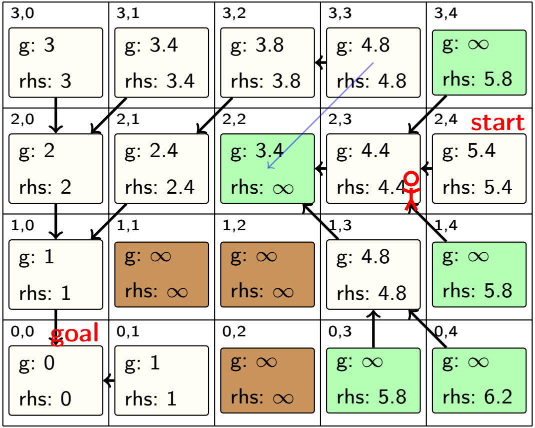
Legend

Free node	Obstacle node
On open list	Active node

Update Vertex

- Incomming edges to (2,2)
- Call `UpdateVertex()` on the neighbors (2,2)
- The transition cost is ∞, and therefore, the *rhs* value previously computed using (2,2) is changed

D* Lite – Example Planning (27)



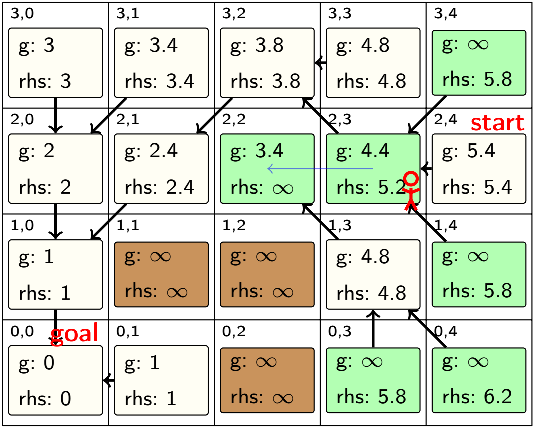
Legend

Free node	Obstacle node
On open list	Active node

Update Vertex

- The neighbor of (2,2) is (3,3)
- The minimum possible *rhs* value of (3,3) is 4.8 but it is based on the *g* value of (3,2) and not (2,2), which is the detected obstacle
- The node (3,3) is still consistent and thus it is not put on the open list

D* Lite – Example Planning (28)



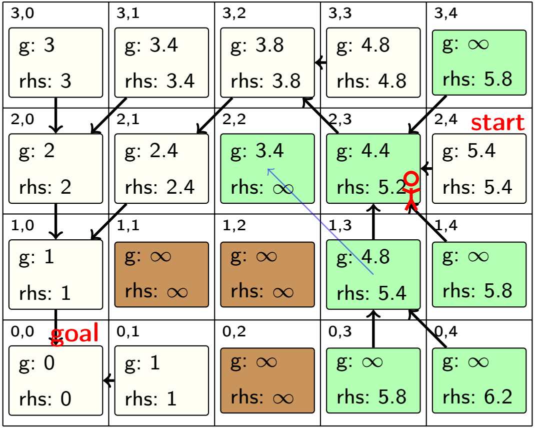
Legend

Free node	Obstacle node
On open list	Active node

Update Vertex

- (2,3) is also a neighbor of (2,2)
- The minimum possible *rhs* value of (2,3) is 5.2 because of (2,2) is obstacle (using (3,2) with 3.8 + 1.4)
- The *rhs* value of (2,3) is different than *g* thus (2,3) is put on the open list

D* Lite – Example Planning (29)



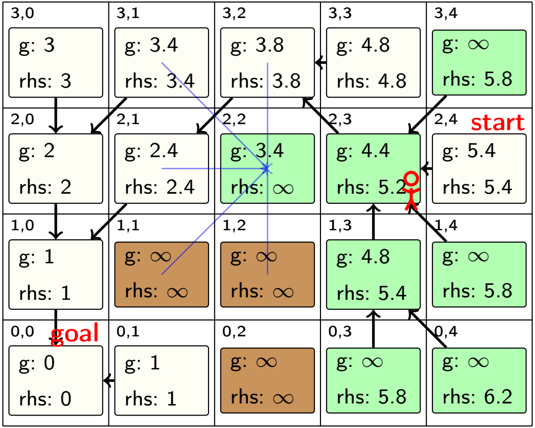
Legend

Free node	Obstacle node
On open list	Active node

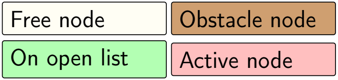
Update Vertex

- Another neighbor of (2,2) is (1,3)
- The minimum possible *rhs* value of (1,3) is 5.4 computed based on *g* of (2,3) with 4.4 + 1 = 5.4
- The *rhs* value is always computed using the *g* values of its successors

D* Lite – Example Planning (29 update)



Legend

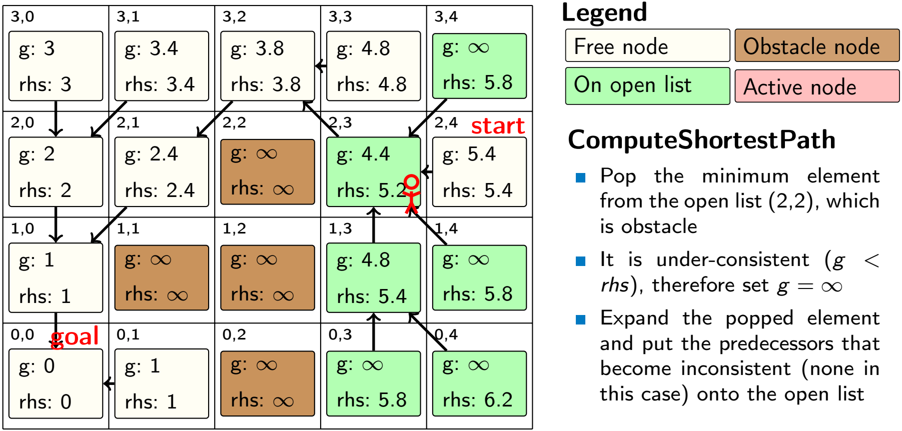


Update Vertex

- None of the other neighbor of (2,2) end up being inconsistent
- We go back to calling `ComputeShortestPath()` until an optimal path is determined

- The node corresponding to the robot's current position is inconsistent and its key is greater than the minimum key on the open list
- Thus, the optimal path is not found yet

D* Lite – Example Planning (30)

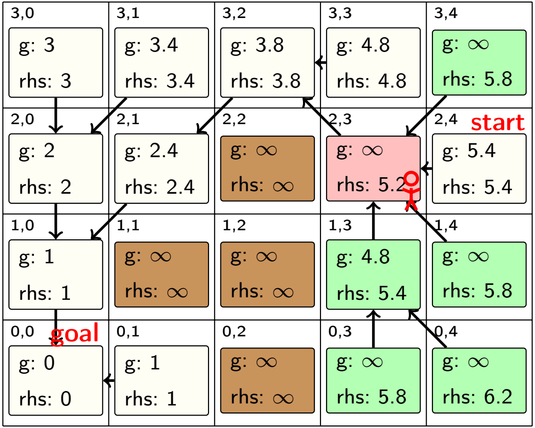


ComputeShortestPath

- Pop the minimum element from the open list (2,2), which is obstacle
- It is under-consistent ($g < rhs$), therefore set $g = \infty$
- Expand the popped element and put the predecessors that become inconsistent (none in this case) onto the open list

- Because (2,2) was under-consistent (when popped), `UpdateVertex()` has to be called on it
- However, it has no effect as its *rhs* value is up to date and consistent

D* Lite – Example Planning (31)



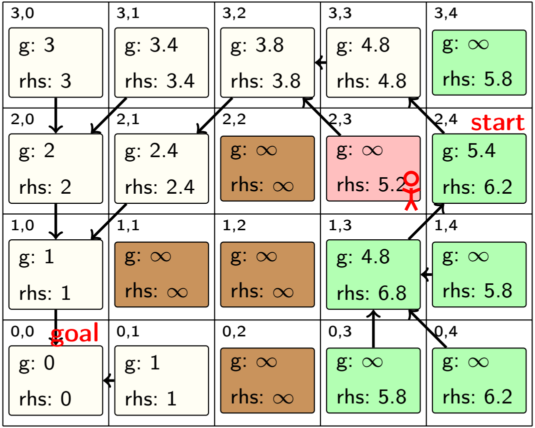
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Pop the minimum element from the open list (2,3)
- It is under-consistent ($g < rhs$), therefore set $g = \infty$

D* Lite – Example Planning (32)



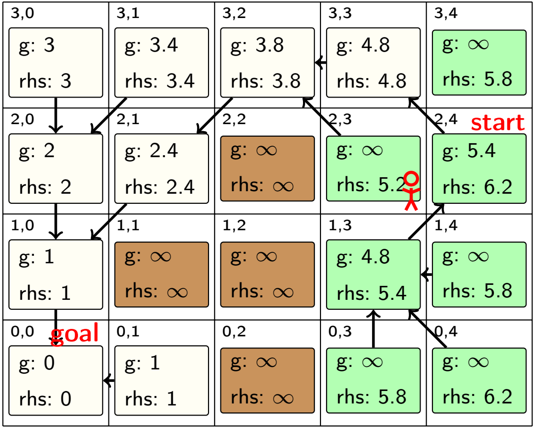
Legend

Free node	Obstacle node
On open list	Active node

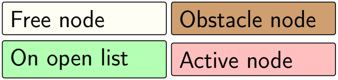
ComputeShortestPath

- Expand the popped element and update the predecessors
- (2,4) becomes inconsistent
- (1,3) gets updated and still inconsistent
- The *rhs* value (1,4) does not changed, but it is now computed from the *g* value of (1,3)

D* Lite – Example Planning (33)



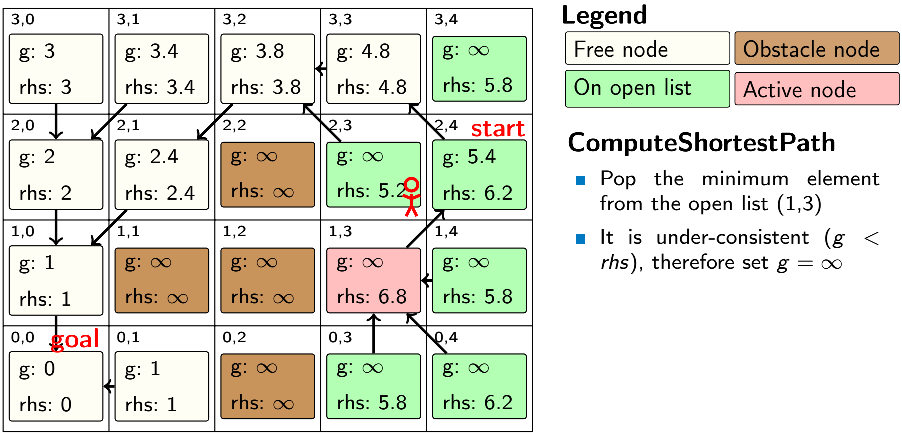
Legend



ComputeShortestPath

- Because (2,3) was under-consistent (when popped), call `UpdateVertex()` on it is needed
- As it is still inconsistent it is put back onto the open list

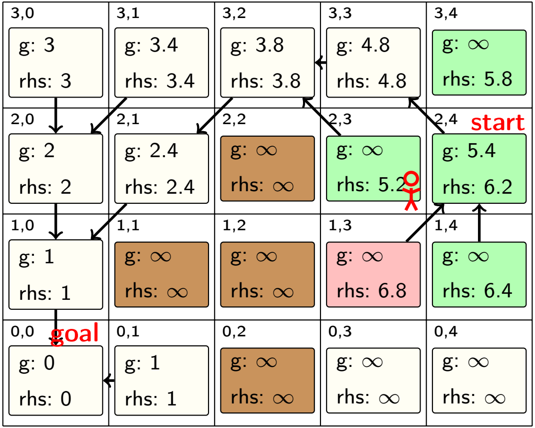
D* Lite – Example Planning (34)



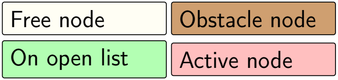
ComputeShortestPath

- Pop the minimum element from the open list (1,3)
- It is under-consistent ($g < rhs$), therefore set $g = \infty$

D* Lite – Example Planning (35)



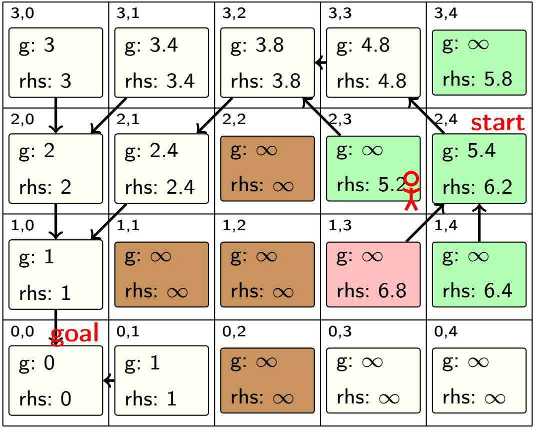
Legend



ComputeShortestPath

- Expand the popped element and update the predecessors
- (1,4) gets updated and still inconsistent
- (0,3) and (0,4) get updated and now consistent (both g and rhs are ∞)

D* Lite – Example Planning (36)



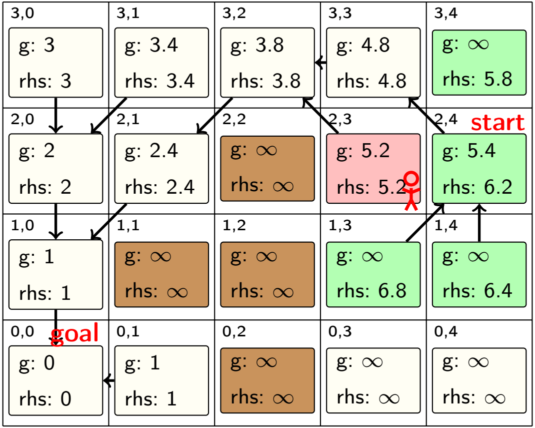
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Because (1,3) was under-consistent (when popped), call `UpdateVertex()` on it is needed
- As it is still inconsistent it is put back onto the open list

D* Lite – Example Planning (37)



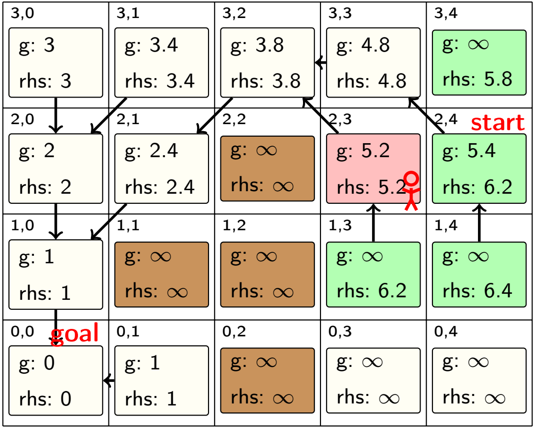
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Pop the minimum element from the open list (2,3)
- It is over-consistent ($g > rhs$), therefore set $g = rhs$

D* Lite – Example Planning (38)



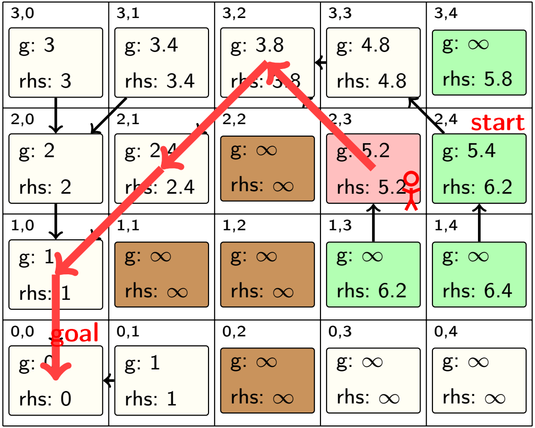
Legend

Free node	Obstacle node
On open list	Active node

ComputeShortestPath

- Expand the popped element and update the predecessors
- (1,3) gets updated and still inconsistent
- The node (2,3) corresponding to the robot's position is consistent
- Besides, top of the key on the open list is not less than the key of (2,3)
- The optimal path has been found and we can break out of the loop

D* Lite – Example Planning (39)



Legend

Free node	Obstacle node
On open list	Active node

- Follow the gradient of g values from the robot's current position (node)

D* Lite – Comments

- D* Lite works with real valued costs, not only with binary costs (free/obstacle)
- The search can be focused with an admissible heuristic that would be added to the g and *rhs* values
- The final version of D* Lite includes further optimization (not shown in the example)
  - Updating the *rhs* value without considering all successors every time
  - Re-focusing the search as the robot moves without reordering the entire open list

Reaction-Diffusion Processes Background

- *Reaction-Diffusion* (RD) models – dynamical systems capable to reproduce the autowaves
- *Autowaves* - a class of nonlinear waves that propagate through an active media

  At the expense of the energy stored in the medium, e.g., grass combustion.
- RD model describes spatio-temporal evolution of two state variables $u = u(\vec{x}, t)$ and $v = v(\vec{x}, t)$ in space $\vec{x}$ and time t

$$\begin{aligned} \dot{u} &= f(u, v) + D_u \triangle u \\ \dot{v} &= g(u, v) + D_v \triangle v \end{aligned},$$

where $\triangle$ is the Laplacian.

This RD-based path planning is informative, just for curiosity

Reaction-Diffusion Background

- FitzHugh-Nagumo (FHN) model

FitzHugh R, Biophysical Journal (1961)

$$\begin{aligned} \dot{u} &= \varepsilon\left(u - u^3 - v + \phi\right) + D_u \triangle u \\ \dot{v} &= (u - \alpha v + \beta) + D_v \triangle u \end{aligned},$$

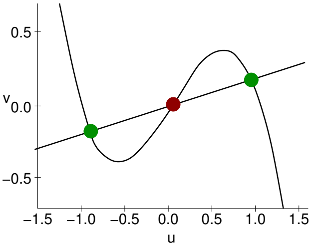
where α, β, ϵ, and ϕ are parameters of the model.

- Dynamics of RD system is determined by the associated *nullcline configurations* for $\dot{u}$=0 and $\dot{v}$=0 in the absence of diffusion, i.e.,

$$\begin{aligned} \varepsilon\left(u - u^3 - v + \phi\right) &= 0, \\ (u - \alpha v + \beta) &= 0, \end{aligned}$$

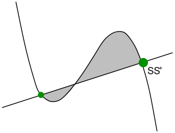
which have associated geometrical shapes

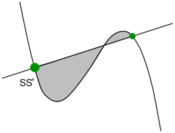
Nullcline Configurations and Steady States



- Nullclines intersections represent
  - Stable States (*SSs*)
  - Unstable States
- Bistable regime

  The system (concentration levels of (u, v) for each grid cell) tends to be in SSs.





- We can modulate relative stability of both SS

  "preference" of SS^+ over SS^-
- System moves from SS^- to SS^+,

  if a small perturbation is introduced.
- The *SS*s are separated by a mobile frontier

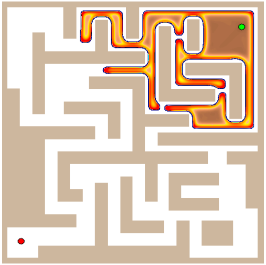
  a kind of traveling frontwave (autowaves)

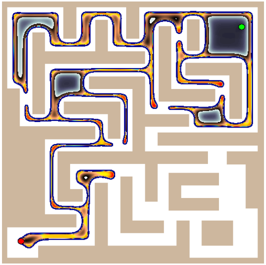
RD-based Path Planning – Computational Model

- Finite difference method on a Cartesian grid with Dirichlet boundary conditions (FTCS) *discretization → grid based computation → grid map*
- *External forcing* – introducing additional information
  i.e., constraining concentration levels to some specific values
- Two-phase evolution of the underlying RD model

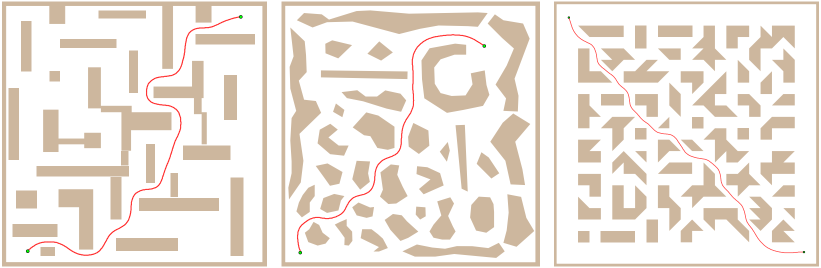
1. **Propagation phase**
   - Freespace is set to SS^- and the start location SS^+
   - Parallel propagation of the frontwave with *non-annihilation property*

     Vázquez-Otero and Muñuzuri, CNNA (2010)
   - Terminate when the frontwave reaches the goal
2. **Contraction phase**
   - Different nullclines configuration
   - Start and goal positions are forced towards SS^+
   - SS^- shrinks until only the path linking the forced points remains





Example of Found Paths

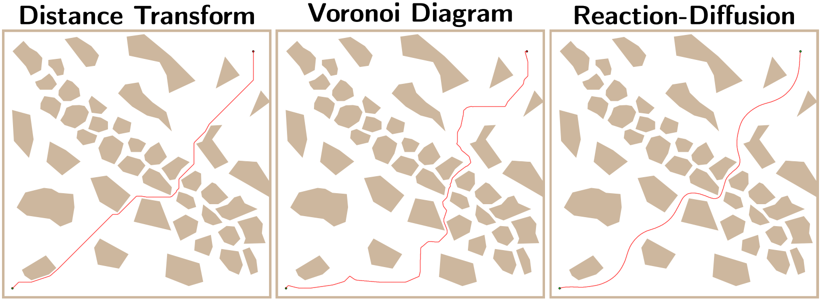


700 × 700   700 × 700   1200 × 1200

- The path clearance maybe adjusted by the wavelength and size of the computational grid.

Control of the path distance from the obstacles (path safety)

Comparison with Standard Approaches



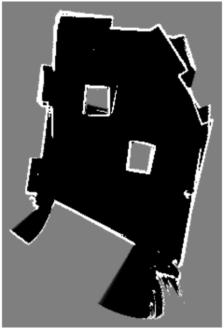
Jarvis R
Advanced Mobile Robots (1994)

Beeson P, Jong N, Kuipers B
ICRA (2005)

Otero A, Faigl J, Muñuzuri A
IROS (2012)

- RD-based approach provides competitive paths regarding path length and clearance, while they seem to be smooth

Robustness to Noisy Data



Vázquez-Otero, A., Faigl, J., Duro, N. and Dormido, R. (2014): Reaction-Diffusion based Computational Model for Autonomous Mobile Robot Exploration of Unknown Environments. International Journal of Unconventional Computing (IJUC).

Summary of the Lecture

Topics Discussed

- Front-Wave propagation and path simplification
- Distance Transform based planning
- Graph based planning methods: Dijsktra's, A*, JPS, Theta*
- D* Lite
- Reaction-Diffusion based planning (*informative*)

- Next: Randomized Sampling-based Motion Planning Methods