

# Path Planning

Jan Faigl

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

Lecture 03

B4M36UIR – Artificial Intelligence in Robotics



## Robot Motion Planning – Motivational problem

- How to transform high-level task specification (provided by humans) into a low-level description suitable for controlling the actuators?

To develop algorithms for such a transformation.

The motion planning algorithms provide transformations how to move a robot (object) considering all operational constraints.



## Real Mobile Robots

In a real deployment, the problem is more complex.

- The world is changing.
- Robots update the knowledge about the environment.  
*localization, mapping and navigation*
- New decisions have to be made based on the feedback from the environment.  
*Motion planning is a part of the mission re-planning loop.*



Josef Štrunc, Bachelor thesis, CTU, 2009.

An example of robotic mission:

Multi-robot exploration of unknown environment.

How to deal with real-world complexity?

Relaxing constraints and considering realistic assumptions.



## Overview of the Lecture

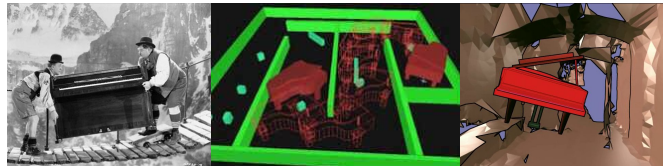
- Part 1 – Path Planning
  - Introduction to Path Planning
  - Notation and Terminology
  - Path Planning Methods
- Part 2 – Grid and Graph based Path Planning Methods
  - Grid-based Planning
  - DT for Path Planning
  - Graph Search Algorithms
  - D\* Lite
  - Path Planning based on Reaction-Diffusion Process



## Piano Mover's Problem

A classical motion planning problem

Having a CAD model of the piano, model of the environment, the problem is how to move the piano from one place to another without hitting anything.



Basic motion planning algorithms are focused primarily on rotations and translations.

- We need **notation** of model representations and formal definition of the problem.
- Moreover, we also need a context about the problem and **realistic assumptions**.  
*The plans have to be admissible and feasible.*



## Notation

- $\mathcal{W}$  – **World model** describes the robot workspace and its boundary determines the obstacles  $\mathcal{O}_i$ .  
*2D world,  $\mathcal{W} = \mathbb{R}^2$*
- A **Robot** is defined by its geometry, parameters (kinematics) and it is controllable by the motion plan.
- $\mathcal{C}$  – **Configuration space (C-space)**  
A concept to describe possible configurations of the robot. The robot's **configuration** completely specify the robot location in  $\mathcal{W}$  including specification of all degrees of freedom.  
*E.g., a robot with rigid body in a plane  $\mathcal{C} = \{x, y, \varphi\} = \mathbb{R}^2 \times S^1$ .*
  - Let  $\mathcal{A}$  be a subset of  $\mathcal{W}$  occupied by the robot,  $\mathcal{A} = \mathcal{A}(q)$ .
  - A subset of  $\mathcal{C}$  occupied by obstacles is  
$$\mathcal{C}_{obs} = \{q \in \mathcal{C} : \mathcal{A}(q) \cap \mathcal{O}_i, \forall i\}.$$
  - Collision-free configurations** are  
$$\mathcal{C}_{free} = \mathcal{C} \setminus \mathcal{C}_{obs}.$$

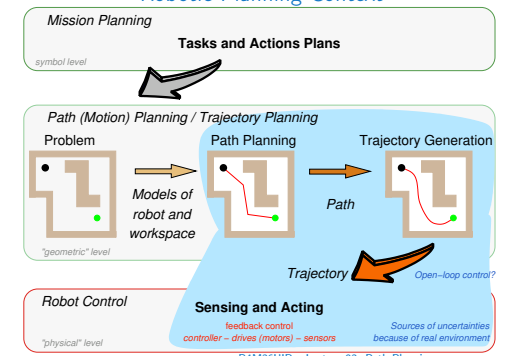


## Part I

### Part 1 – Path and Motion Planning



## Robotic Planning Context



## Path / Motion Planning Problem

- Path** is a continuous mapping in  $\mathcal{C}$ -space such that  
$$\pi : [0, 1] \rightarrow \mathcal{C}_{free}, \text{ with } \pi(0) = q_0, \text{ and } \pi(1) = q_f.$$
  - Trajectory** is a path with explicit parametrization of time, e.g., accompanied by a description of the motion laws ( $\gamma : [0, 1] \rightarrow \mathcal{U}$ , where  $\mathcal{U}$  is robot's action space).  
*It includes dynamics.*
- The path planning is the determination of the function  $\pi(\cdot)$ .

Additional requirements can be given:

- Smoothness** of the path;
  - Kinodynamic constraints**, e.g., considering friction forces;
  - Optimality criterion** – shortest vs fastest (length vs curvature).
- 
- Path planning** – planning a collision-free path in  $\mathcal{C}$ -space.
  - Motion planning** – planning collision-free motion in the **state space**.



Introduction to Path Planning Notation Path Planning Methods

### Planning in $C$ -space

Robot path planning for a disk-shaped robot with a radius  $\rho$ .

Motion planning problem in geometrical representation of  $\mathcal{W}$ .

Motion planning problem in  $C$ -space representation.

$C$ -space has been obtained by enlarging obstacles by the disk  $\mathcal{A}$  with the radius  $\rho$ .  
By applying Minkowski sum:  $\mathcal{O} \oplus \mathcal{A} = \{x + y \mid x \in \mathcal{O}, y \in \mathcal{A}\}$ .

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 12 / 118

Introduction to Path Planning Notation Path Planning Methods

### Example of $C_{obs}$ for a Robot with Rotation

A simple 2D obstacle  $\rightarrow$  has a complicated  $C_{obs}$ .

- Deterministic algorithms exist. Requires exponential time in  $C$  dimension, J. Canny, PAMI, 8(2):200–209, 1986.
- Explicit representation of  $C_{free}$  is impractical to compute.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 13 / 118

Introduction to Path Planning Notation Path Planning Methods

### Representation of $C$ -space

How to deal with continuous representation of  $C$ -space?

**Continuous Representation of  $C$ -space**

$\downarrow$

**Discretization**  
processing critical geometric events, (random) sampling  
roadmaps, cell decomposition, potential field

$\downarrow$

**Graph Search Techniques**  
BFS, Gradient Search, A\*

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 14 / 118

Introduction to Path Planning Notation Path Planning Methods

### Planning Methods - Overview

(selected approaches)

- Point-to-point path/motion planning.** Multi-goal path/motion/trajectory planning later
- Roadmap based methods** – Create a connectivity graph of the free space. (Complete but impractical)
  - Visibility graph;
  - Cell decomposition;
  - Voronoi graph.
- Discretization into a **grid-based** (or lattice-based) representation (Resolution complete)
- Potential field methods** (Complete only for a "navigation function", which is hard to compute in general) Classic path planning algorithms
- Randomized sampling-based methods**
  - Creates a roadmap from connected random samples in  $C_{free}$ .
  - Probabilistic roadmaps. Samples are drawn from some distribution.
  - Very successful in practice.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 16 / 118

Introduction to Path Planning Notation Path Planning Methods

### Visibility Graph

- Compute visibility graph.
- Find the shortest path. E.g., by Dijkstra's algorithm.

Problem Visibility graph Found shortest path

Constructions of the visibility graph:

- Naïve – all segments between  $n$  vertices of the map  $O(n^3)$ ;
- Using rotation trees for a set of segments –  $O(n^2)$ . M. H. Overmars and E. Welzl, 1988

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 17 / 118

Introduction to Path Planning Notation Path Planning Methods

### Minimal Construct: Efficient Shortest Path in Polygonal Maps

- Minimal Construct** algorithm computes visibility graph during the A\* search instead of first computation of the complete visibility graph and then finding the shortest path using A\* or Dijkstra algorithm.
- Based on A\* search with line intersection tests are delayed until they become necessary.
- The intersection tests are further accelerated using bounding boxes.

Full Visibility Graph Minimal Construct

Marcell Misura, Daniel D. Lee, and Maren Bennewitz (2018): Minimal Construct: Efficient Shortest Path Finding for Mobile Robots in Polygonal Maps. IROS.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 18 / 118

Introduction to Path Planning Notation Path Planning Methods

### Voronoi Graph

- Roadmap is Voronoi graph that maximizes clearance from the obstacles.
- Start and goal positions are connected to the graph.
- Path is found using a graph search algorithm.

Voronoi graph Path in graph Found path

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 19 / 118

Introduction to Path Planning Notation Path Planning Methods

### Visibility Graph vs Voronoi Graph

**Visibility graph**

- Shortest path, but it is close to obstacles. We have to consider safety of the path. An error in plan execution can lead to a collision.
- Complicated in higher dimensions

**Voronoi graph**

- It maximizes clearance, which can provide conservative paths.
- Small changes in obstacles can lead to large changes in the graph.
- Complicated in higher dimensions.

A combination is called Visibility-Voronoi – R. Wein, J. P. van den Berg, D. Halperin, 2004.

For higher dimensions we need other types of roadmaps.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 20 / 118

Introduction to Path Planning Notation Path Planning Methods

### Cell Decomposition

- Decompose free space into parts. Any two points in a convex region can be directly connected by a segment.
- Create an adjacency graph representing the connectivity of the free space.
- Find a path in the graph.

**Trapezoidal decomposition**

Centroids represent cells Connect adjacency cells Find path in the adjacency graph

- Other decomposition (e.g., triangulation) are possible.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 21 / 118

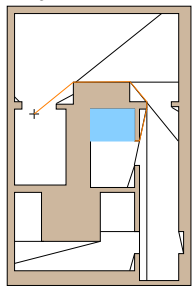
Introduction to Path Planning Notation Path Planning Methods

## Shortest Path Map (SPM)

- Speedup computation of the shortest path towards a particular goal location  $p_g$  for a polygonal domain  $\mathcal{P}$  with  $n$  vertices.
- A partitioning of the free space into cells with respect to the particular location  $p_g$ .
- Each cell has a vertex on the shortest path to  $p_g$ .
- Shortest path from any point  $p$  is found by determining the cell (in  $O(\log n)$  using point location alg.) and then traversing the shortest path with up to  $k$  bends, i.e., it is found in  $O(\log n + k)$ .
- Determining the SPM using "wavefront" propagation based on continuous Dijkstra paradigm.

*Joseph S. B. Mitchell: A new algorithm for shortest paths among obstacles in the plane, Annals of Mathematics and Artificial Intelligence, 3(1):83-109, 1991.*

- SPM is a precompute structure for the given  $\mathcal{P}$  and  $p_g$ ;
  - single-point query.



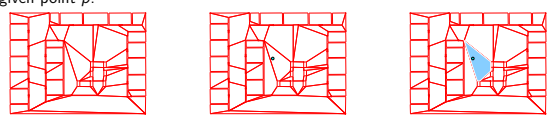
*A similar structure can be found for two-point query, e.g., H. Guo, A. Maheshwari, J.-R. Sack, 2008.*

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 22 / 118

Introduction to Path Planning Notation Path Planning Methods

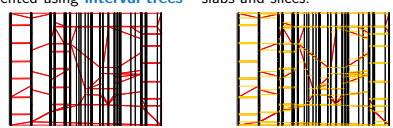
## Point Location Problem

- For a given partitioning of the polygonal domain into a discrete set of cells, determine the cell for a given point  $p$ .



*Masato Edahiro, Iwao Kokubo and Takao Asano: A new point-location algorithm and its practical efficiency: comparison with existing algorithms, ACM Trans. Graph., 3(2):86-109, 1984.*

- It can be implemented using **interval trees** – slabs and slices.



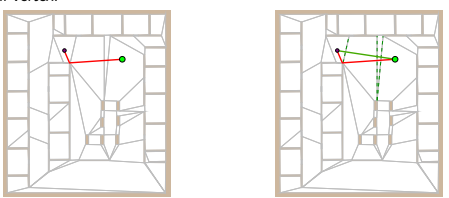
Point location problem. SPM and similarly problems are from the Computational Geometry field.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 23 / 118

Introduction to Path Planning Notation Path Planning Methods

## Approximate Shortest Path and Navigation Mesh

- We can use any convex partitioning of the polygonal map to speed up shortest path queries.
  - Precompute all shortest paths from map vertices to  $p_g$  using visibility graph.
  - Then, an estimation of the shortest path from  $p$  to  $p_g$  is the shortest path among the one of the cell vertex.



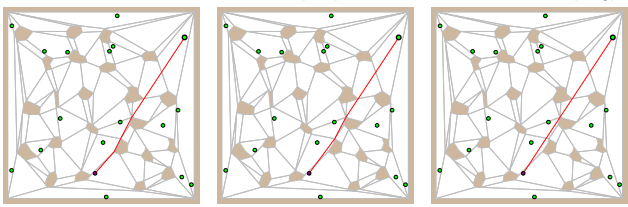
- The estimation can be further improved by "ray-shooting" technique combined with walking in triangulation (convex partitioning).

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 24 / 118

Introduction to Path Planning Notation Path Planning Methods

## Path Refinement

- Testing collision of the point  $p$  with particular vertices of the estimation of the shortest path.
  - Let the initial path estimation from  $p$  to  $p_g$  be a sequence of  $k$  vertices  $(p, v_1, \dots, v_k, p_g)$ .
  - We can iteratively test if the segment  $(p, v_i)$ ,  $1 < i < k$  is collision free up to  $(p, p_g)$ .



Path over  $v_0$  Path over  $v_1$  Full refinement

*With the precomputed structures, an estimate of the shortest path is determined in units of microseconds.*

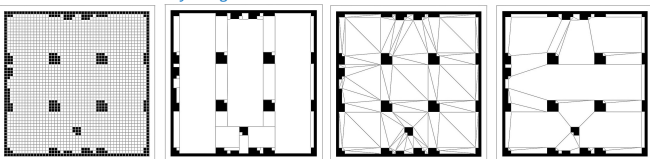
Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 25 / 118

Introduction to Path Planning Notation Path Planning Methods

## Navigation Mesh

- In addition to robotic approaches, fast shortest path queries are studied in computer games.
- There is a class of algorithms based on navigation mesh.
  - A supporting structure representing the free space.

*It usually originated from the grid based maps, but it is represented as CDT – Constrained Delaunay triangulation.*



Grid mesh Merged grid mesh CDT mesh Merged CDT mesh

- E.g., **Polyanya** algorithm based on navigation mesh and best-first search.

*M. Cui, D. Harabor, A. Grastien: Compromise-free Pathfinding on a Navigation Mesh, IJCAI 2017, 496-502. https://bitbucket.org/dharabor/pathfinding*

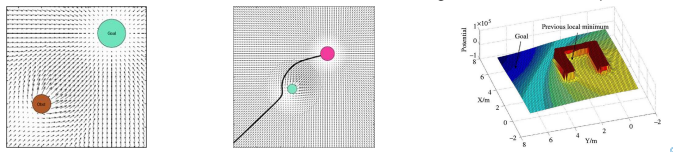
Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 26 / 118

Introduction to Path Planning Notation Path Planning Methods

## Artificial Potential Field Method

- The idea is to create a function  $f$  that will provide a direction towards the goal for any configuration of the robot.
- Such a function is called **navigation function** and  $-\nabla f(q)$  points to the goal.
- Create a **potential field** that will attract robot towards the goal  $q_f$  while obstacles will generate **repulsive potential** repelling the robot away from the obstacles.

*The navigation function is a sum of potentials.*



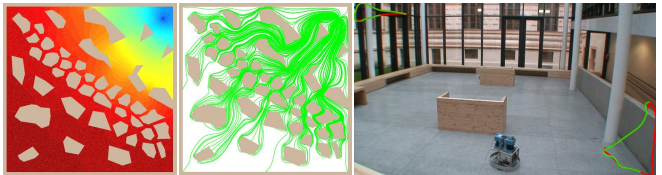
Such a potential function can have several local minima.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 27 / 118

Introduction to Path Planning Notation Path Planning Methods

## Avoiding Local Minima in Artificial Potential Field

- Consider harmonic functions that have only one extremum
 
$$\nabla^2 f(q) = 0.$$
- Finite element method with defined Dirichlet and Neumann boundary conditions.



*J. Maćák, Master thesis, CTU, 2009*

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 28 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Part II

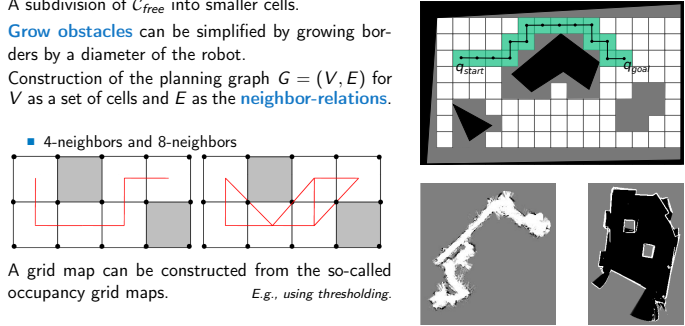
### Part 2 – Grid and Graph based Path Planning Methods

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 29 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Grid-based Planning

- A subdivision of  $C_{free}$  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.*



Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 31 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Grid-based Environment Representations

- Hierarchical planning with coarse resolution and re-planning on finer resolution.
  - Hofe, 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  $C_{free}$ .

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 32 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Example of Simple Grid-based Planning

- Wave-front propagation using path simplification
- 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 33 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Example – Wave-Front Propagation (Flood Fill)

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 34 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Path Simplification

- The initial path is found in a grid using 8-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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 35 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Bresenham’s Line Algorithm

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

```

1 CoordsVector bresenham(const Coords& pt1, const Coords& pt2,
2   CoordsVector& line)
3 {
4   // The pt2 point is not added into line
5   int x0 = pt1.c; int y0 = pt1.r;
6   int x1 = pt2.c; int y1 = pt2.r;
7   Coords p;
8   int dx = x1 - x0;
9   int dy = y1 - y0;
10  int step = (abs(dy) >= abs(dx));
11  if (step == 1)
12    SWAP(x0, y0);
13  SWAP(x1, y1);
14  dx = x1 - x0; // recompute Dx, Dy
15  dy = y1 - y0;
16  int xstep = 1;
17  if (dx < 0) {
18    xstep = -1;
19    dx = -dx;
20  }
21  int ystep = 1;
22  if (dy < 0) {
23    ystep = -1;
24    dy = -dy;
25  }
26  int twoBy = 2 * dy;
27  int twoByTwoDx = twoBy * 2 + dx; // 2*Dy - 2*Dx
28  int e = twoBy - dx; // 2*Dy - Dx
29  int y = y0;
30  int xDraw, yDraw;
31  for (int x = x0; x != x1; x += xstep) {
32    if (step == 1)
33      xDraw = x;
34      yDraw = y;
35    } else {
36      xDraw = x;
37      yDraw = y;
38    }
39    p.c = xDraw;
40    p.r = yDraw;
41    line.push_back(p); // add to the line
42    if (e > 0) {
43      e += twoByTwoDx; // E += 2*Dy - 2*Dx
44      y = y + ystep;
45    } else {
46      e += twoBy; // E += 2*Dy
47    }
48  }
49  return line;
50 }

```

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 36 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## 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
 
$$\text{cost}(c) = \min_{i=1}^8 (\text{cost}(c_i) + EV_{c,c_i}),$$
 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 cost.

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

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 38 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## 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);
        else
          cell[x,y] := xMax * yMax;
      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;

```

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 39 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Distance Transform based Path Planning – Impl. 1/2

```

1 Grid& DT::compute(Grid& grid) const
2 {
3   static const double DIAGONAL = sqrt(2);
4   static const double ORTHOGONAL = 1;
5   const int H = map.H;
6   const int W = map.W;
7   assert(grid.H == H and grid.W == W, "size");
8   bool anyChange = true;
9   int counter = 0;
10  while (anyChange) {
11    anyChange = false;
12    for (int r = 1; r < H - 1; ++r) {
13      for (int c = 1; c < W - 1; ++c) {
14        if (map[r][c] != FREESPACE)
15          continue;
16        //obstacle detected
17        double t[4];
18        t[0] = grid[r - 1][c - 1] + DIAGONAL;
19        t[1] = grid[r - 1][c] + ORTHOGONAL;
20        t[2] = grid[r - 1][c + 1] + DIAGONAL;
21        t[3] = grid[r][c - 1] + ORTHOGONAL;
22        double pom = grid[r][c];
23        for (int i = 0; i < 4; ++i) {
24          if (pom > t[i]) {
25            pom = t[i];
26            counter++;
27          } //end while any change
28          return grid;
29        }
30        if (anyChange) {
31          grid[r][c] = pom;
32        }
33      }
34    }
35  }
36  for (int r = H - 2; r > 0; --r) {
37    for (int c = W - 2; c > 0; --c) {
38      if (map[r][c] != FREESPACE)
39        continue;
40      //obstacle detected
41      double t[4];
42      t[0] = grid[r + 1][c - 1] + DIAGONAL;
43      t[1] = grid[r + 1][c] + ORTHOGONAL;
44      t[2] = grid[r + 1][c + 1] + DIAGONAL;
45      t[3] = grid[r][c - 1] + ORTHOGONAL;
46      double pom = grid[r][c];
47      for (int i = 0; i < 4; ++i) {
48        if (pom > t[i]) {
49          pom = t[i];
50          counter++;
51        } //end while any change
52        return grid;
53      }
54      if (anyChange) {
55        grid[r][c] = pom;
56      }
57    }
58  }
59  counter++;
60  //end while any change
61  return grid;
62 }

```

A boundary is assumed around the rectangular map

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 40 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

## Distance Transform based Path Planning – Impl. 2/2

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

```

1 Coords min8Point(const Grid& grid, Coords p)
2 {
3   double min = std::numeric_limits<double>::max();
4   const int H = grid.H;
5   const int W = grid.W;
6   Coords t;
7   for (int r = p.r - 1; r <= p.r + 1; ++r) {
8     for (int c = p.c - 1; c <= p.c + 1; ++c) {
9       if (c < 0 or c >= W) continue;
10      if (min > grid[r][c]) {
11        min = grid[r][c];
12        t.r = r; t.c = c;
13      }
14    }
15  }
16  }
17  }
18  p = t;
19  return p;
20 }

```

```

22 CoordsVector DT::findPath(const Coords& start, const Coords& goal, CoordsVector& path)
23 {
24   static const double DIAGONAL = sqrt(2);
25   static const double ORTHOGONAL = 1;
26   const int H = map.H;
27   const int W = map.W;
28   Grid grid(H, W, HW); // HW max grid value
29   grid[goal.r][goal.c] = 0;
30   compute(grid);
31   if (grid[start.r][start.c] >= HW) {
32     WARN("Path has not been found");
33   } else {
34     Coords pt = start;
35     while (pt.r != goal.r or pt.c != goal.c) {
36       path.push_back(pt);
37       min8Point(grid, pt);
38     }
39     path.push_back(goal);
40   }
41   return path;
42 }

```

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 41 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### DT Example

- $\delta = 10$  cm,  $L = 27.2$  m
- $\delta = 30$  cm,  $L = 42.8$  m

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 42 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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 (BFS);
  - Depth first search (DFS);
  - Dijkstra'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 algorithms for on-line search, incremental search and with any-time and real-time properties, e.g.,
  - Lifelong Planning A\* (LPA\*).
- E-Graphs – Experience graphs

Koenig, S., Likhachev, M. and Furcy, D. (2004): Lifelong Planning A\*. AIJ.  
Phillips, M. et al. (2012): E-Graphs: Bootstrapping Planning with Experience Graphs. RSS.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 44 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### Examples of Graph/Grid Search Algorithms

A\* (general)

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 45 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### A\* Algorithm

- A\* uses a user-defined  $h$ -values (heuristic) to focus the search.
  - Prefer expansion of the node  $n$  with the lowest value

Peter Hart, Nils Nilsson, and Bertram Raphael, 1968

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

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.
- Admissibility 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 –  $h(n) = 0$ .

Do we need admissible? When and why?

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 46 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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.,
  - Fibonacci heap, binomial heap,  $k$ -level bucket;
  - binary heap** is usually sufficient with  $O(\log n)$ .
- Forward A\*
  - Create a search tree and initiate it with the start location.
  - Select generated but not yet expanded state  $s$  with the smallest  $f$ -value,  $f(s) = g(s) + h(s)$ .
  - Stop if  $s$  is the goal.
  - Expand the state  $s$ .
  - Goto Step 2.

Similar to Dijkstra's algorithm but it uses  $f(s)$  with the heuristic  $h(s)$  instead of pure  $g(s)$ .

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 47 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### Dijkstra's vs A\* vs Jump Point Search (JPS)

Dijkstra's Algorithm

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 48 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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**).
  - Natural neighbors after neighbor pruning 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\*.
- JPS+ is optimized preprocessed version of JPS with goal bounding.

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

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

<https://github.com/SteveRabin/JPSPlusWithGoalBouding>

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

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 49 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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 2: 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).

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 50 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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 42.

$\delta = 10$  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://sigmeedev.com/open/tutorial/lazy-theta-star/>

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 51 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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 Focused 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 52 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 53 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### D\* Lite – Demo

Jan Faigl, 2022 <https://www.youtube.com/watch?v=X5a149n3E9s> B4M36UIR – Lecture 03: Path Planning 55 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 56 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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
 
$$rhs(u) = \begin{cases} 0 & \text{if } u = s_{start} \\ \min_{s' \in Succ(u)} (g(s') + c(s', u)) & \text{otherwise} \end{cases}$$
- 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 57 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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));
    
```

U is priority queue with the vertices.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 58 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### D\* Lite Algorithm – ComputeShortestPath()

```

Procedure ComputeShortestPath
while U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start) do
  u := U.Pop();
  if g(u) > rhs(u) then
    g(u) := rhs(u);
    foreach s ∈ Pred(u) do UpdateVertex(s);
  else
    g(u) := ∞;
    foreach s ∈ Pred(u) ∪ {u} do UpdateVertex(s);

Procedure UpdateVertex
if u ≠ s_goal then rhs(u) := min_{s' ∈ Succ(u)} (c(u, s') + g(s'));
if u ∈ U then U.Remove(u);
if g(u) ≠ 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))]
    
```

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 59 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### D\* Lite – Demo

<https://github.com/sdeyoo/d-star-lite>

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 60 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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:  $\infty$ .

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 61 / 118

### D\* Lite – Example Planning (1)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**Initialization**  
 ■ Set  $rhs = 0$  for the goal.  
 ■ Set  $rhs = g = \infty$  for all other nodes.

### D\* Lite – Example Planning (2)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**Initialization**  
 ■ Put the goal to the open list.  
 It is inconsistent.

### D\* Lite – Example Planning (3–init)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**ComputeShortestPath**  
 ■ Pop the minimum element from the open list (goal).  
 ■ It is over-consistent ( $g > rhs$ ).

### D\* Lite – Example Planning (3)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**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)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**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  $\infty$  because the transition to obstacle has cost  $\infty$ .

### D\* Lite – Example Planning (5–init)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**ComputeShortestPath**  
 ■ Pop the minimum element from the open list (1,0).  
 ■ It is over-consistent ( $g > rhs$ ).

### D\* Lite – Example Planning (5)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**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)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**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.

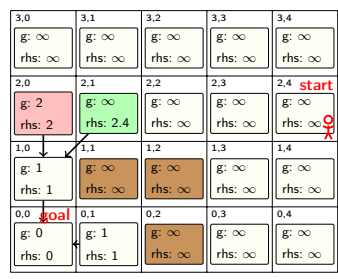
### D\* Lite – Example Planning (7)

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

**Legend**  
 Free node (white) Obstacle node (brown)  
 On open list (green) Active node (red)

**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)

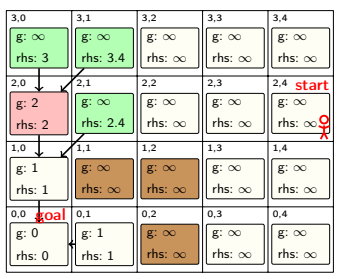


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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)

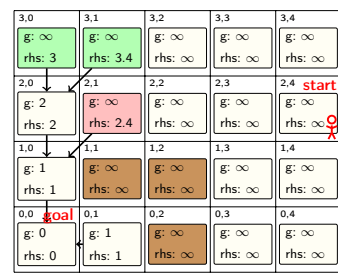


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**ComputeShortestPath**

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

### D\* Lite – Example Planning (10–init)

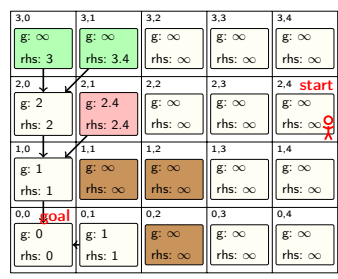


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**ComputeShortestPath**

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

### D\* Lite – Example Planning (10)

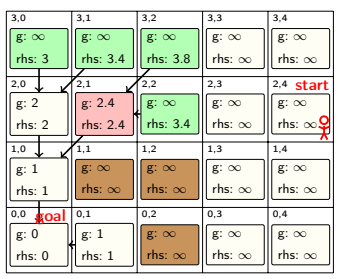


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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)

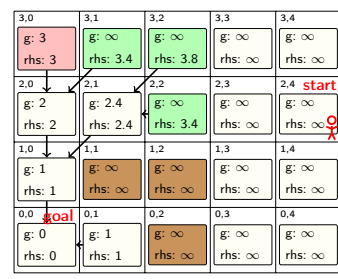


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**ComputeShortestPath**

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

### D\* Lite – Example Planning (12)

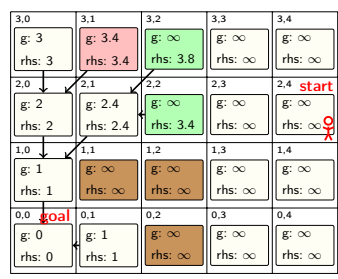


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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)

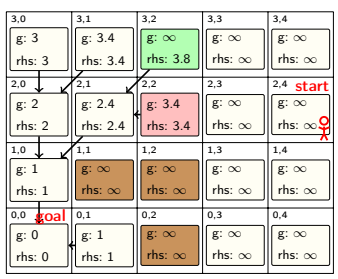


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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)

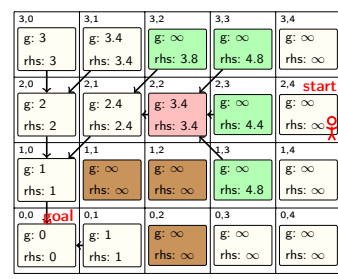


**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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)



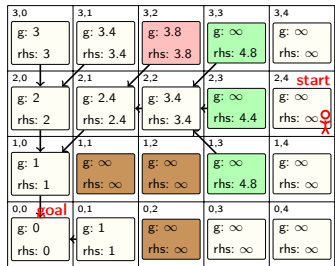
**Legend**  
 Free node (white), Obstacle node (brown), On open list (green), Active node (red)

**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

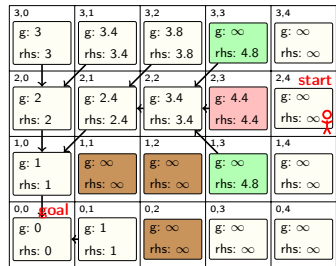


#### ComputeShortestPath

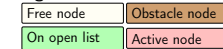
- Pop the minimum element from the open list (3,2).
- It is over-consistent ( $g > rhs$ ) and thus set  $g = rhs$ .
- Expand the popped element and put the predecessors that become inconsistent onto the open list.
- In this case, none of the predecessors become inconsistent.



### D\* Lite – Example Planning (17)



#### Legend

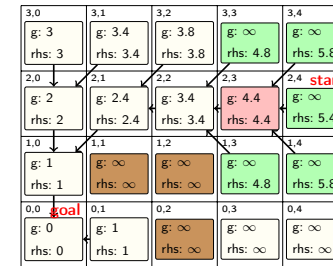


#### ComputeShortestPath

- 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

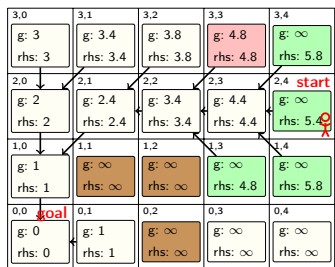


#### ComputeShortestPath

- 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

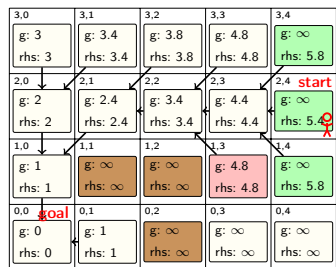


#### ComputeShortestPath

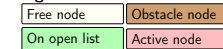
- Pop the minimum element from the open list (3,2).
- It is over-consistent ( $g > rhs$ ) and thus set  $g = rhs$ .
- Expand the popped element and put the predecessors that become inconsistent onto the open list.
- In this case, none of the predecessors become inconsistent.



### D\* Lite – Example Planning (20)



#### Legend

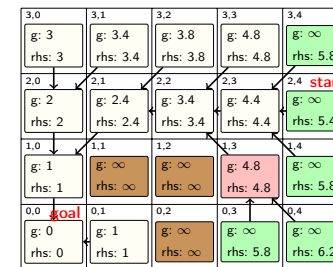


#### ComputeShortestPath

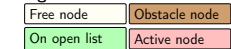
- Pop the minimum element from the open list (1,3).
- It is over-consistent ( $g > rhs$ ) and thus set  $g = rhs$ .



### D\* Lite – Example Planning (21)



#### Legend

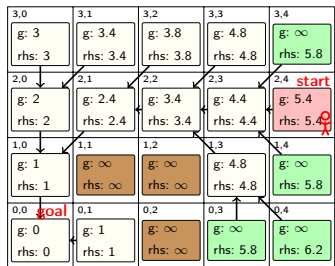


#### ComputeShortestPath

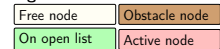
- Expand the popped element and put the predecessors that become inconsistent onto the open list, i.e., (0,3) and (0,4).



### D\* Lite – Example Planning (22)



#### Legend



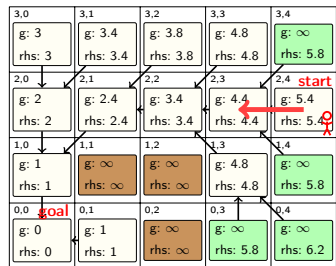
#### ComputeShortestPath

- 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.

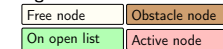


- The start node becomes consistent and the top key on the open list is not less than the key of the start node.
- An optimal path is found and the loop of the ComputeShortestPath is broken.

### D\* Lite – Example Planning (23)



#### Legend

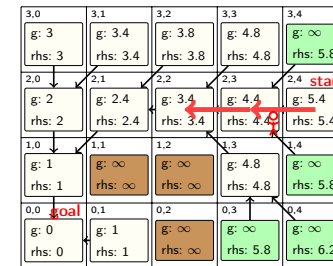


#### ComputeShortestPath

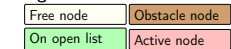
- Follow the gradient of  $g$  values from the start node.



### D\* Lite – Example Planning (24)



#### Legend

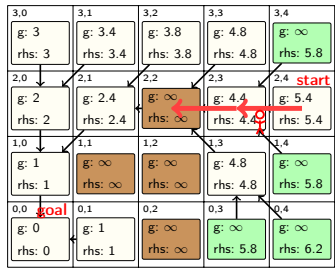


#### ComputeShortestPath

- Follow the gradient of  $g$  values from the start node.



### D\* Lite – Example Planning (25)

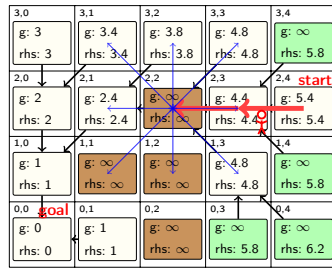


#### Legend

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)

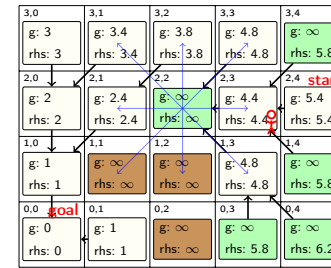


#### Legend

Free node	Obstacle node
On open list	Active node

- 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)



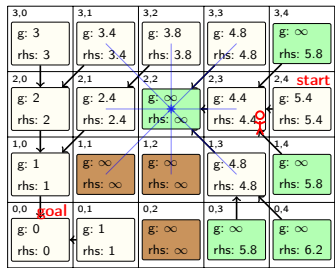
#### Legend

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 = ∞ and (2,2) becomes inconsistent and it is put on the open list.

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



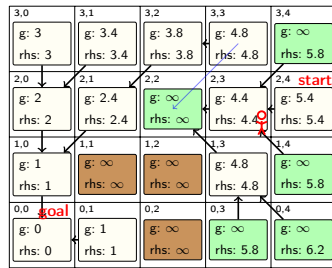
#### Legend

Free node	Obstacle node
On open list	Active node

#### Update Vertex

- Incoming 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)



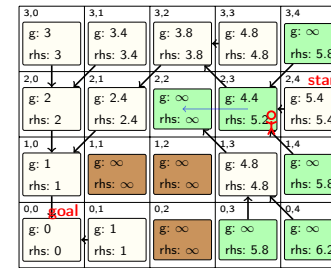
#### Legend

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)



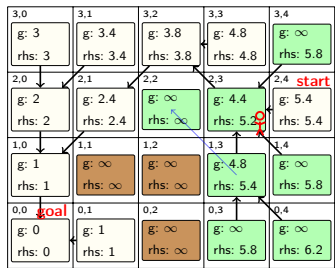
#### Legend

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 (2,2) is an obstacle (using (3,2) with 3.8 + 1.4).
- The rhs value of (2,3) is different from g; thus, (2,3) is put on the open list.

### D\* Lite – Example Planning (29)



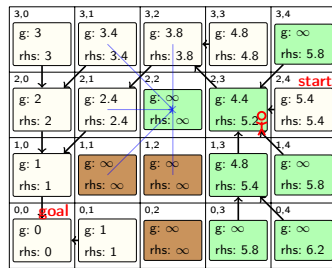
#### Legend

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)



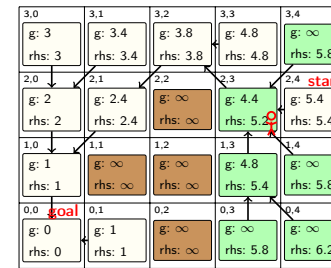
#### Legend

Free node	Obstacle node
On open list	Active node

#### 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.

### D\* Lite – Example Planning (30)



#### Legend

Free node	Obstacle node
On open list	Active node

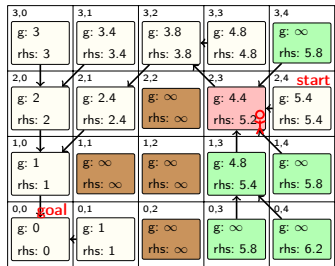
#### ComputeShortestPath

- 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.

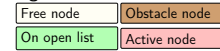
- The node corresponding to the robot's current position is inconsistent and its key is greater than the minimum key on the open list.
- Thus, the optimal path is not found yet.

- 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-init)



#### Legend

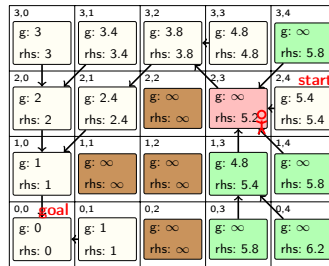


#### ComputeShortestPath

- Pop the minimum element from the open list (2,3).
- It is under-consistent  $g < rhs$ .



### D\* Lite – Example Planning (31)



#### Legend

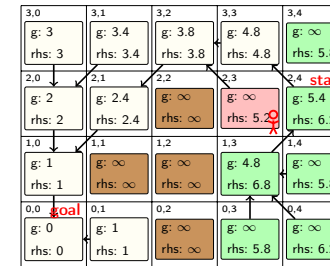


#### ComputeShortestPath

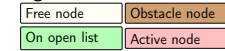
- Pop the minimum element from the open list (2,3).
- It is under-consistent  $g < rhs$  therefore set  $g = \infty$ .



### D\* Lite – Example Planning (32)



#### Legend

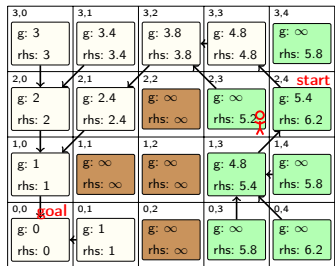


#### ComputeShortestPath

- 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 change, but it is now computed from the g value of (1,3).



### D\* Lite – Example Planning (33)



#### Legend

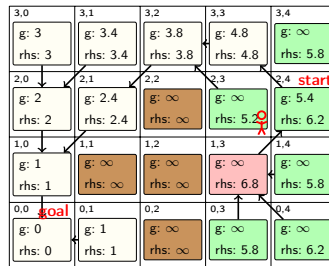


#### ComputeShortestPath

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



### D\* Lite – Example Planning (34)



#### Legend

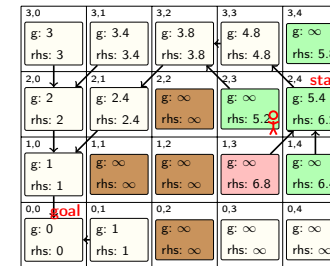


#### ComputeShortestPath

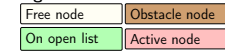
- Pop the minimum element from the open list (1,3).
- It is under-consistent ( $g < rhs$ ), therefore set  $g = \infty$ .



### D\* Lite – Example Planning (35)



#### Legend

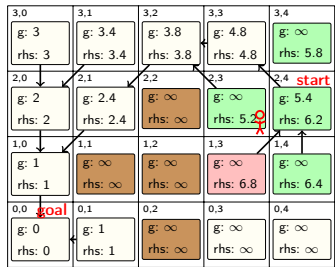


#### ComputeShortestPath

- 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  $\infty$ ).



### D\* Lite – Example Planning (36)



#### Legend

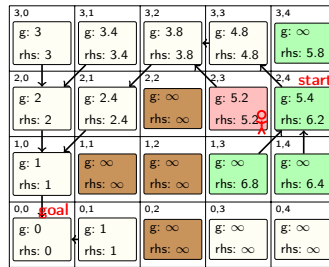


#### ComputeShortestPath

- 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

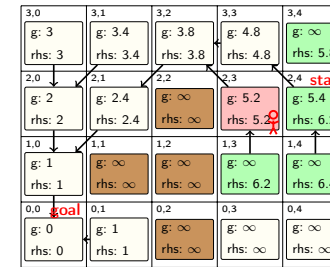


#### ComputeShortestPath

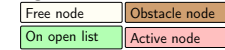
- Pop the minimum element from the open list (2,3).
- It is over-consistent ( $g > rhs$ ), therefore set  $g = rhs$ .



### D\* Lite – Example Planning (38)



#### Legend



#### 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, the 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.



Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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).

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 107 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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  $g$  and  $rhs$ .
- 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.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 108 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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  $t$ 

$$\dot{u} = f(u, v) + D_u \Delta u$$

$$\dot{v} = g(u, v) + D_v \Delta v$$
 where  $\Delta$  is the Laplacian.

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

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 110 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### Reaction-Diffusion Background

- FitzHugh-Nagumo (FHN) model *FitzHugh R, Biophysical Journal (1961)*

$$\dot{u} = \varepsilon(u - u^3 - v + \phi) + D_u \Delta u$$

$$\dot{v} = (u - \alpha v + \beta) + D_v \Delta v$$
 where  $\alpha, \beta, \varepsilon,$  and  $\phi$  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.,
 
$$\varepsilon(u - u^3 - v + \phi) = 0,$$

$$(u - \alpha v + \beta) = 0,$$
 which have associated geometrical shapes.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 111 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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 SSs are separated by a mobile frontier – a kind of traveling frontwave (autowaves).

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 112 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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.
  - Propagation phase**
    - Freespace is set to  $SS^-$  and the start location  $SS^+$ .
    - Parallel propagation of the frontwave with *non-annihilation property.* *Vázquez-Otero and Mufuzuri, CNNA (2010)*
    - Terminate when the frontwave reaches the goal.
  - Contraction phase**
    - Different nullclines configuration.
    - Start and goal positions are forced towards  $SS^+$ .
    - $SS^-$  shrinks until only the path linking the forced points remains.

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 113 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### Example of Found Paths

- The path clearance maybe adjusted by the **wavelength** and size of the computational grid. *Control of the path distance from the obstacles (path safety).*

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 114 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### Comparison with Standard Approaches

Distance Transform

Jarvis R  
Advanced Mobile Robots (1994)

Voronoi Diagram

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

Reaction-Diffusion

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

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

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 115 / 118

Grid-based Planning DT for Path Planning Graph Search Algorithms D\* Lite RD-based Planning

### 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).

Jan Faigl, 2022 B4M36UIR – Lecture 03: Path Planning 116 / 118

## Summary of the Lecture



## Topics Discussed

- Motion and path planning problems
  - Path planning methods – overview
  - Notation of configuration space
- Path planning methods for geometrical map representation
  - Shortest-Path Roadmaps
  - Voronoi diagram based planning
  - Cell decomposition method
- Distance transform can be utilized for kind of *navigational function*
  - Front-Wave propagation and path simplification
- Artificial potential field method
- Graph search (planning) methods for grid-like representation
  - Dijkstra's, A\*, JPS, Theta\*
  - Dedicated speed up techniques can be employed to decreasing computational burden, e.g., JPS
  - Grid-path can be smoothed, e.g., using path simplification or Theta\* like algorithms
- We can avoid demanding planning from scratch reusing the previous plan for the updated environment map, e.g., using **D\* Lite**
- Unconventional reaction-diffusion based planning (*informative*)
- **Next: Robotic Information Gathering – Mobile Robot Exploration**

