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

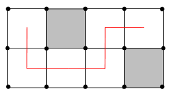
- Part 1 – Grid and Graph based Path Planning Methods
  - Grid-based Planning
  - DT for Path Planning
  - Graph Search Algorithms
  - D* Lite

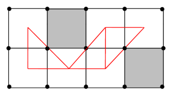
Part I

Part 1 – Grid and Graph based Path Planning Methods

Grid-based Planning

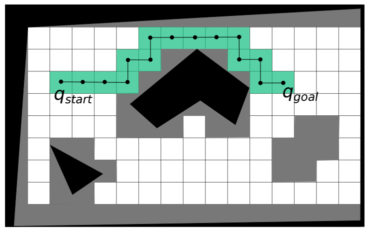
- A subdivision of $\mathcal{C}_{free}$ into smaller cells
- **Grow obstacles** can be simplified by growing borders by a diameter of the robot
- Construction of the planning graph $G = (V, E)$ for V as a set of cells and E as the **neighbor-relations**
  - 4-neighbors and 8-neighbors

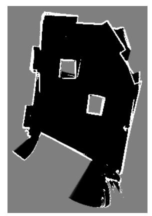




- A grid map can be constructed from the so-called occupancy grid maps

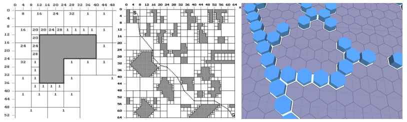
E.g., using thresholding





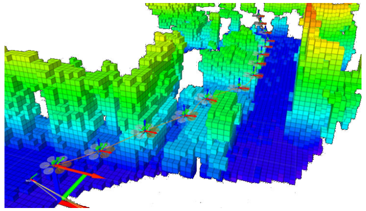
Grid-based Environment Representations

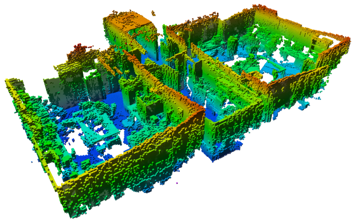
- Hiearchical planning
  - Coarse resolution and re-planning on finer resolution



- Octotree 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





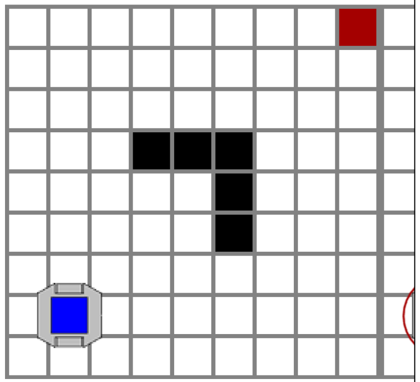
– Memory grows with the size of the environment

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

Example of Simple Grid-based Planning

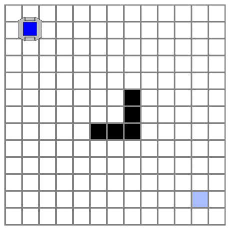
- Front-wave propagation using path simplification

- 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**

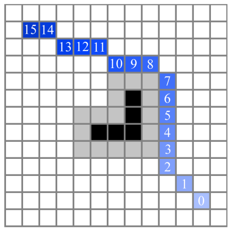


Path Simplification

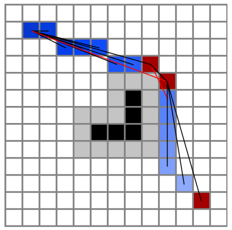
- The initial path is found in a grid using 4-neighbors of a cell
- The rayshoot cast a line into a grid and possible collisions of the robot with obstacles are checked
- The "fartherset" cells without collisions are used as "turn" points
- The final path is a sequence of straight line segments



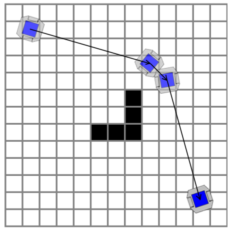
Initial and goal locations



Obtacle growing, front-wave propagation



Ray-shooting



Simplified path

Bresenham's Line Algorithm

- Filling a grid by a line with avoding using 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

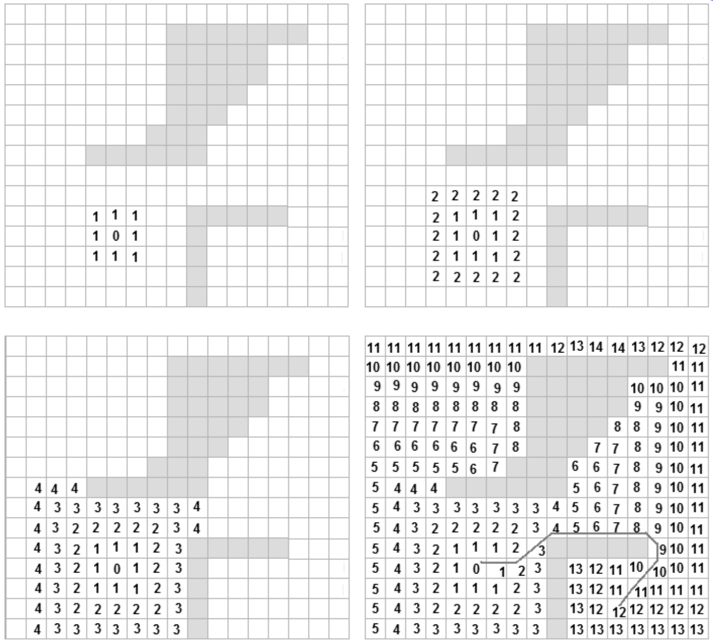
- For a given goal location and grid map compute a navigational function using *frontwave* algorithm, i.e., a kind of *potential field*
  - The value of the goal cell is set to 0 and all other free cells are set to some very high value
  - For each free cell compute a number of cells to towards the goal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

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

Example – Distance Transform based Path Planning



Distance Transform Path Planning

Algorithm 1: Distance Transform for Path Planning

for *y:=0 to yMax+1* **do**
  for *x:=0 to xMax+1* **do**
    if *goal [x,y]* **then**
      cell [x,y]:=0;
    else
      cell [x,y]:=xMax*y Max;

repeat
  for *y:=2 to yMax* **do**
    for *x:=2 to xMax* **do**
      if *not blocked [x,y]* **then**
        cell [x,y]:= min (cell[x-1,y]+1, cell[x-1,y-1]+$\sqrt{2}$,cell[x,y-1]+1, cell[x+1,y-1]+$\sqrt{2}$,cell [x,y]);
  for *y:=yMax-1downto 1* **do**
    for *x:=xMax-1 downto 1* **do**
      if *not blocked [x,y]* **then**
        cell[x,y]:=min(cell[x+1,y]+1,cell[x+1,y+1]+$\sqrt{2}$,cell[x,y+1]+1,cell[x-1,y+1]+$\sqrt{2}$,cell[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;
}
```

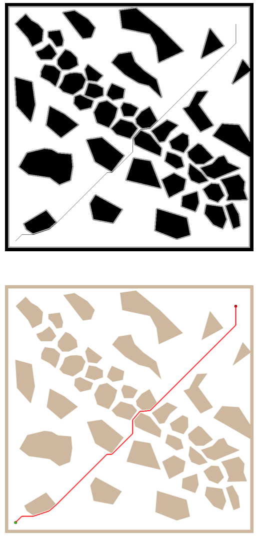
Distance Transform based Path Planning – Impl. 2/2

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

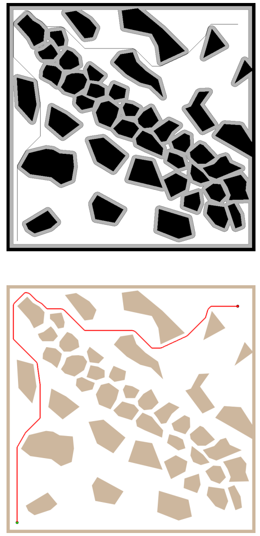
```
Coords& min8Point(const Grid& grid, Coords& p) {
   double min = std::numeric_limits<double>::max();
   const int H = grid.H;
   const int W = grid.W;
   Coords t;

   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);
   path.clear();

   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



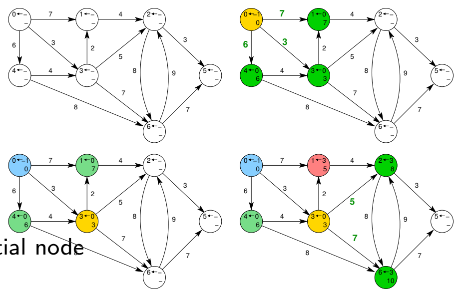
$\delta = 30$ cm, $L = 42.8$ m

Dijkstra's Algorithm

- The grid can be considered as a graph and the path can be found using graph search algorithms
- Dijsktra's algorithm determines paths as iterative update of the cost of the shortest path to the particular nodes

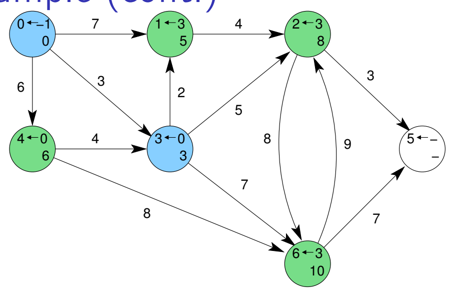
Edsger W. Dijkstra, 1956

- 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*).

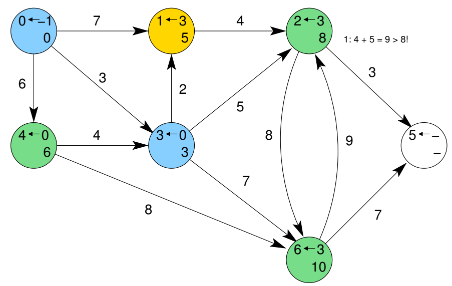


The cost of nodes can only decrease (edge cost is only positive). Therefore, for a node with the currently lowest cost, there cannot be a shorter path from the initial node.

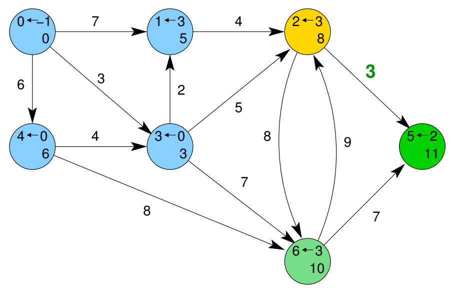
Example (cont.)



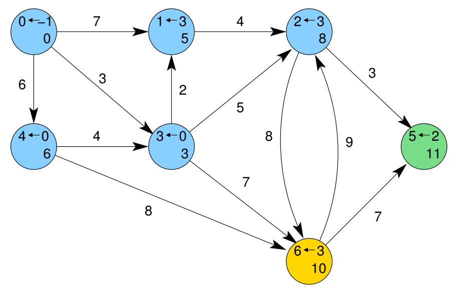
1: After the expansion, the shortest path to the node 2 is over the node 3



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 – 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

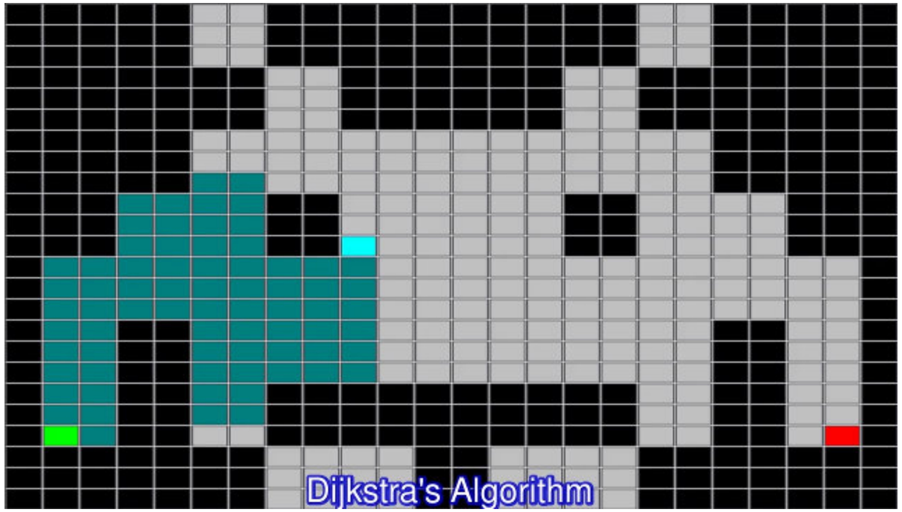
    $$f(n) = g(n) + h(n),$$

    where $g(n)$ is the cost (path length) from 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 hepa, 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 a 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

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



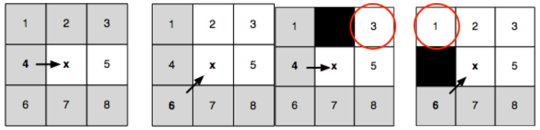
https://www.youtube.com/watch?v=ROG4Ud08lLY

Jump Point Search Algorithm for Grid-based Path Planning

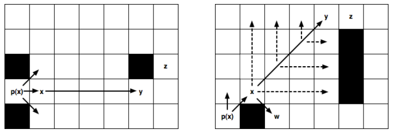
- **Jump Point Search** (JPS) algorithm is based on a macro operator that identifies and selectively expands only certain nodes (**jump points**)

  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

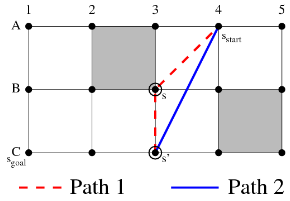
Theta* – Any-Angle Path Planning Algorithm

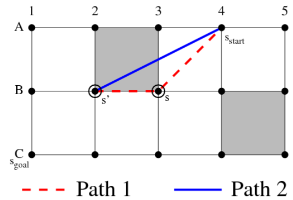
- **Any-angle path planning algorithms** simplify the path during the search
- **Theta*** is an extension of A* with `LineOfSight()`

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

Algorithm 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');
```





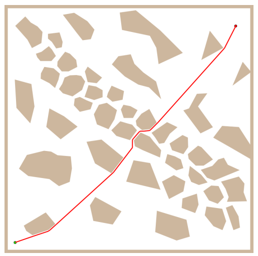
- Path 2: considers path from start to parent(s) and from parent(s) to s' if s' has line-of-sight to parent(s)

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 problems for DT with path smoothing, the path lengths are $L_{\delta=10} = 26.3$ m and $L_{\delta=30} = 40.3$ m, while DT seems to be 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**, used for 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*)

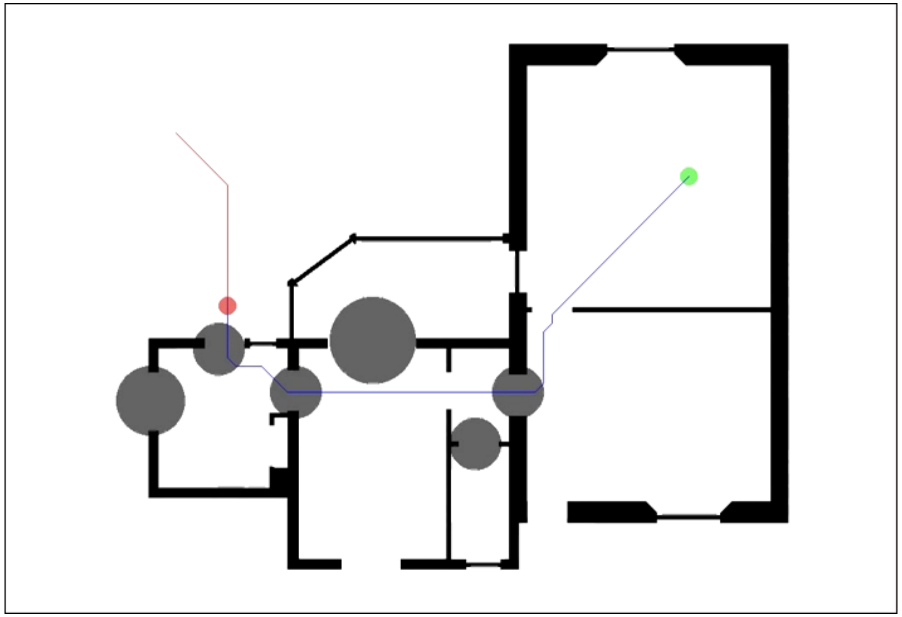
- Execute A* with limited **look-ahead**
- Learns better informed **heuristic** from the experience, initially $h(s)$, e.g., Euclidean distance
- Look-ahead defines **trade-off** between optimality and computational cost
  - `astar(lookahead)`

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

while ($s_{curr} \notin GOAL$) **do**
  astar(lookahead);
  if $s' = FAILURE$ **then**
    return FAILURE;
  for *all* $s \in CLOSED$ **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

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

- The key/priority of a node s in 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 for as the secondary key for tie-breaking

D* Lite Algorithm

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

U = 0;
foreach $s \in S$ **do** $rhs(s) := g(s) := \infty$;
$rhs(s_{goal}$:= 0;
U.Insert(s_{goal}, CalculateKey(s_{goal}));
/* end initialization */;
ComputeShortestPath();
while *($s_{start} \neq s_{goal}$)* **do**
  $s_{start} = \text{argmin}_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
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 $s \in U$ **do**
    U.Update(s, CalculateKey(s));
  ComputeShortestPath();

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

Summary of the Lecture