Faculty of Mathematics and Physics
Charles University
November 12th 2024

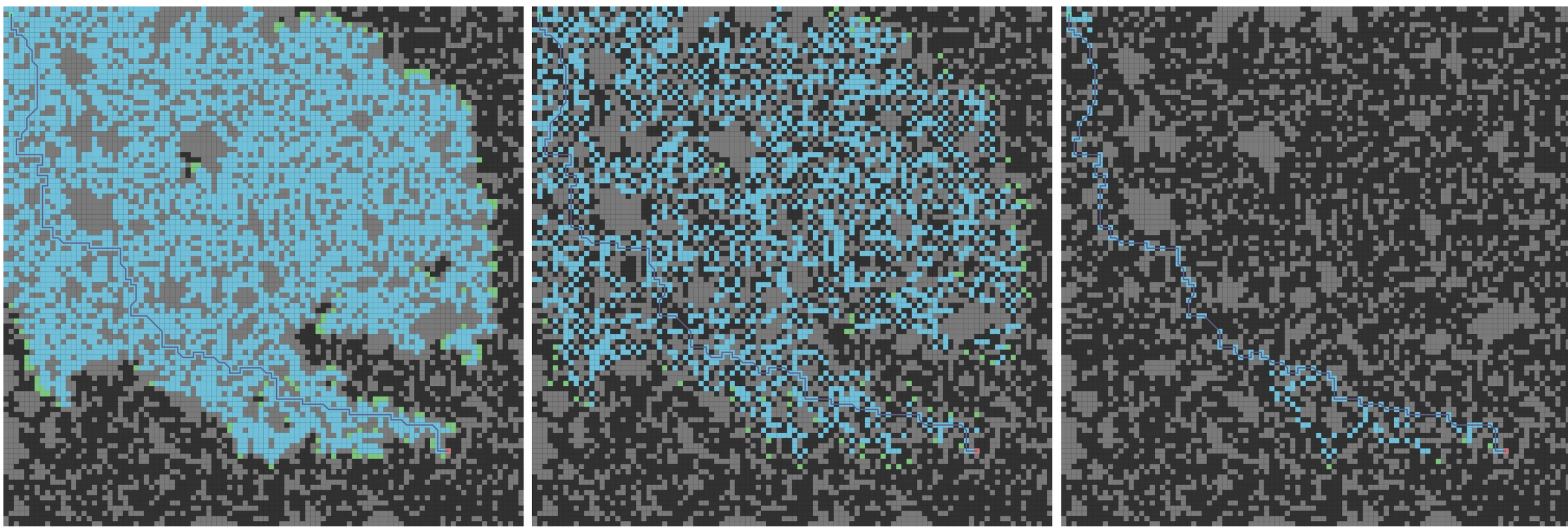
Artificial Intelligence for Computer Games

More search algorithms

Bidirectional search
Partially unknown environments
Rapid random trees

Goal Bounding
Pruning the space

Idea roughly
When searching towards the goal, do not open nodes that leads outside your goal

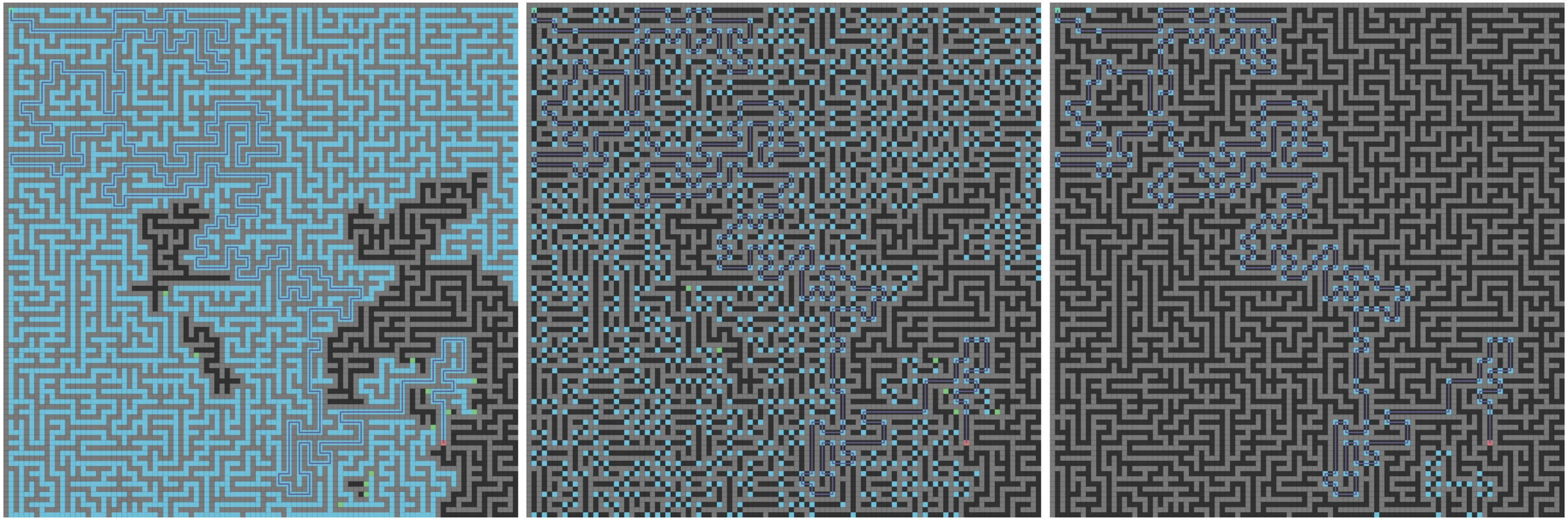


A* JPS+ JPS+ Goal Bounding

Image(s) from the presentation:
http://www.gameaipro.com/Rabin_AISummitGDC2015_JPSPlusGoalBounding.zip

Goal Bounding
Pruning the space

Idea roughly
When searching towards the goal, do not open nodes that leads outside your goal



A* JPS+ JPS+ Goal Bounding

Image(s) from the presentation:
http://www.gameaipro.com/Rabin_AISummitGDC2015_JPSPlusGoalBounding.zip

Goal Bounding
Pruning the space

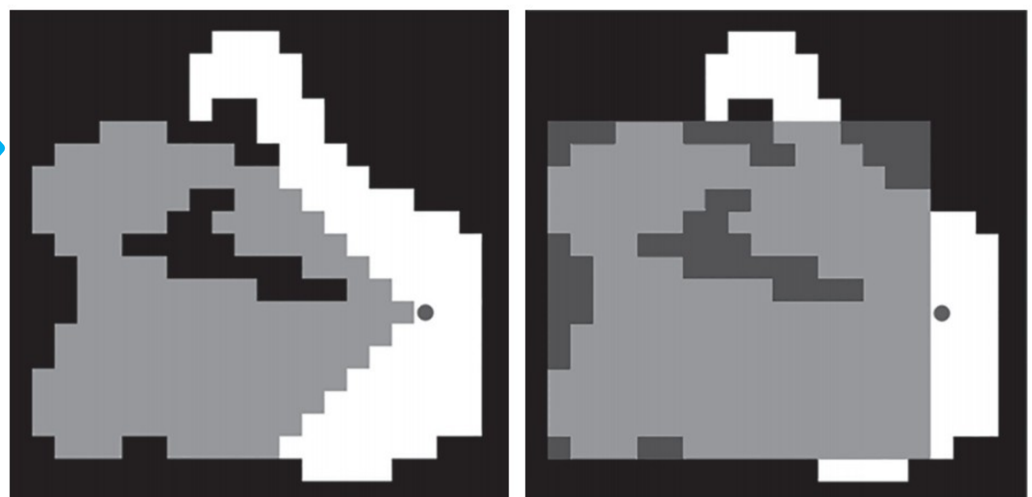
- A method to prune the search space
  - Requires preprocessing the search space offline, so map must be static
  - Assumes 2D maps, but extensible to arbitrary dimensions
- Usable for regular grids as well as navmeshes
- Two sources:
  - Rabin, S. 2015. JPS+ now with Goal Bounding: Over 1000 × Faster than A*, GDC 2015. [PPTX]
  - Rabin, S., Sturtevant, N.R. 2017. Faster A* with Goal Bounding, Game AI Pro 3. [PDF]

Goal Bounding
Pruning the space

Idea

For each oriented edge, store bounding box of the area that contains all nodes that are part of all optimal paths leading through that edge.
Use it to prune edges during expansion.

Nodes on optimal paths reachable via going left from the point

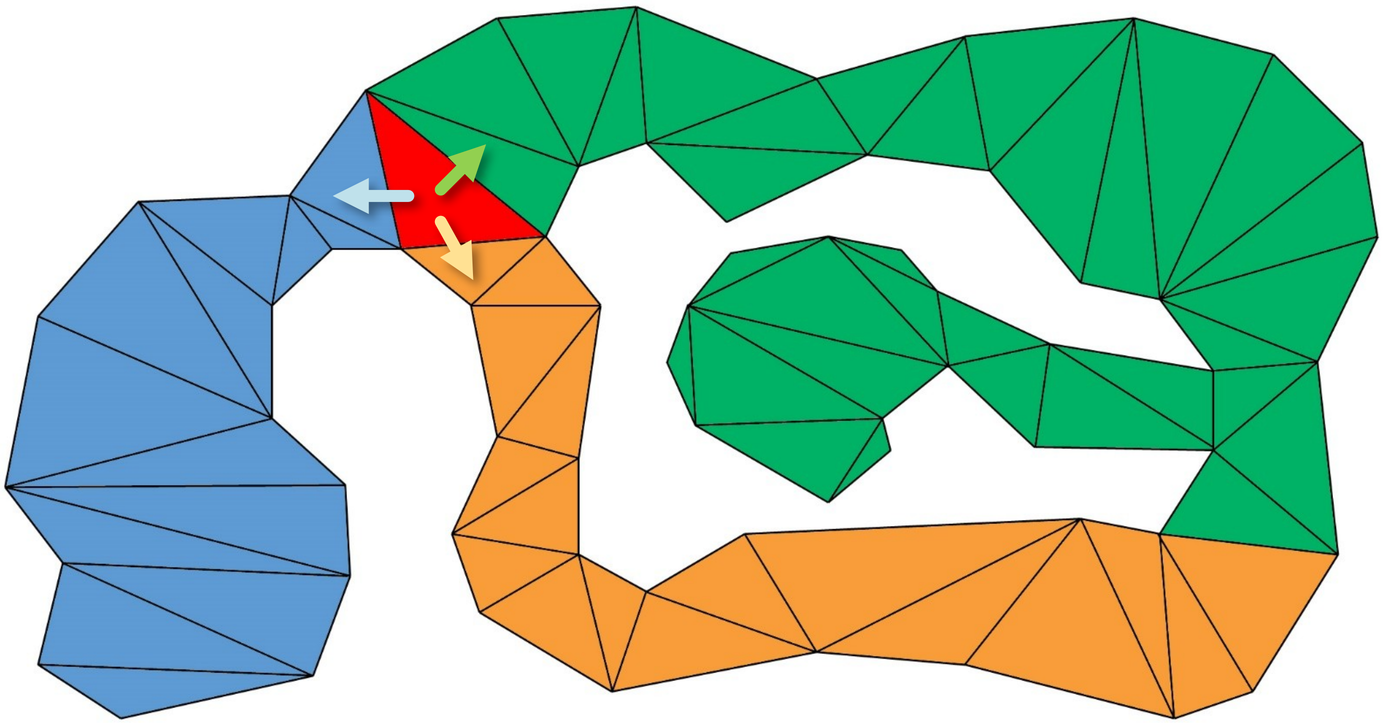


Bounding box of those nodes.

Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Goal Bounding
Pruning the space

Nodes optimally reachable through edges of red triangle.

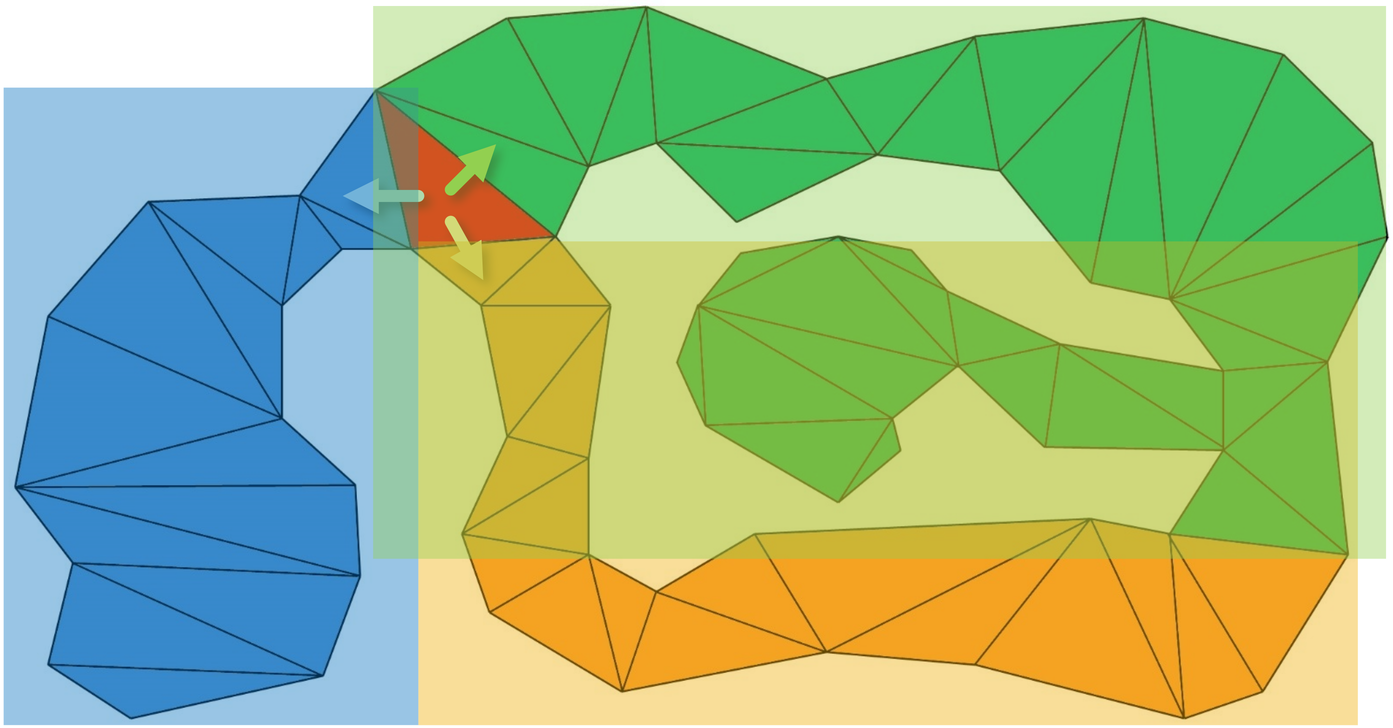


Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Goal Bounding
Pruning the space

Nodes optimally reachable through edges of red triangle.

Respective bounding boxes enclosing optimally reachable areas.



Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Goal Bounding
Graph search algorithm integration

Goal bounding can be integrated into general graph-search algorithm template! Goal bounding box check is fast ~ O(1).

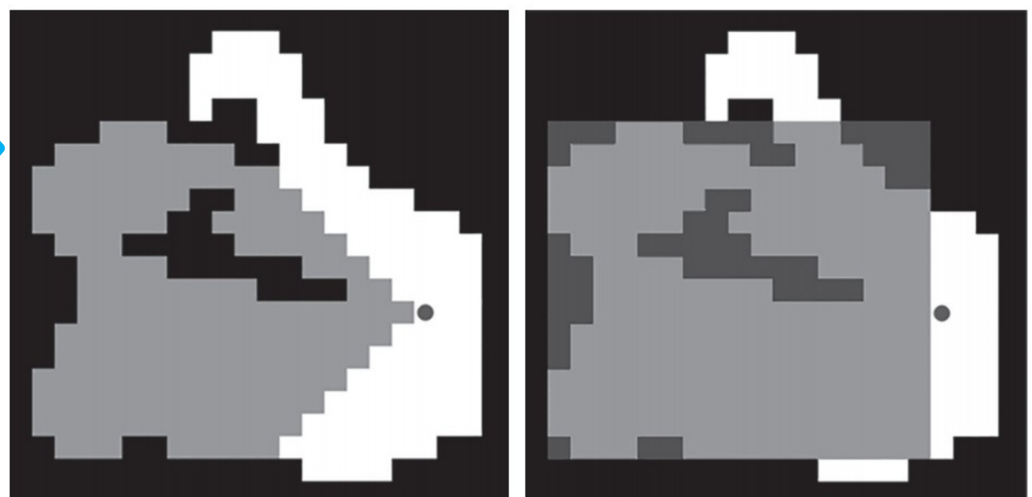
Algorithm template

```
1.  make open-list
2.  push start into open-list
3.  while open-list not empty
4.    extract node from open-list according to "strategy"
5.    if node is target
6.      return path to node
7.    else
8.      expand node by checking its direct neighbors,
              ignoring neighbors whose goal bounding box
              does NOT contain the target,
              possibly adding those who do into open-list
9.      move expanded node to closed-list
```

Goal Bounding
Precomputation phase

- Precomputation must be done for each graph node
- Can be easily run in parallel for each node

Nodes on optimal paths reachable via going left from the point

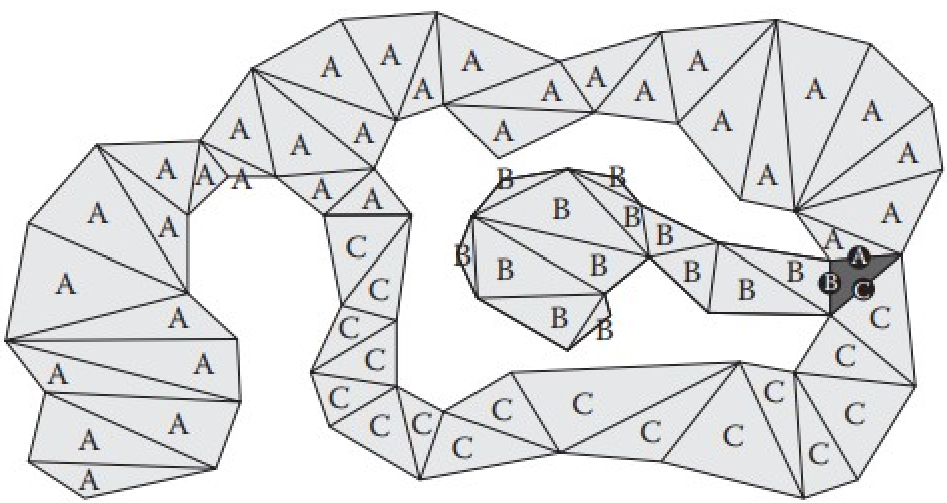


Bounding box of those nodes.

Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Goal Bounding
Precomuptation phase

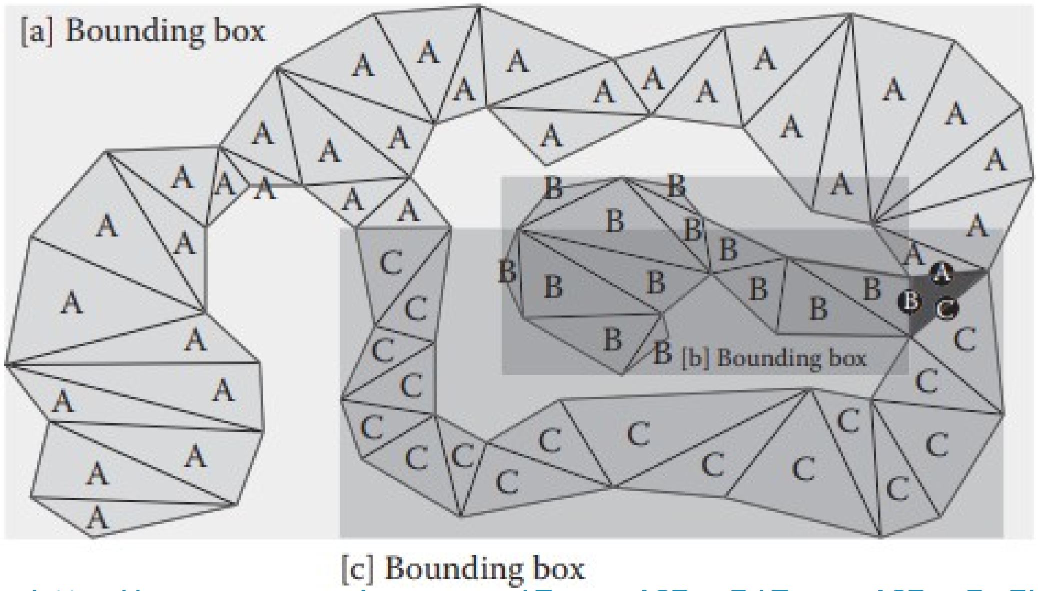
Precomputation idea – step 1
For the given node, run Dijkstra’s algorithm in flood fill mode (no target) marking each node reached with the first edge of the path towards that node.



Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Goal Bounding
Precomuptation phase

Precomputation idea – step 2
For each edge, compute the bounding box of the nodes marked in previous step, store it.



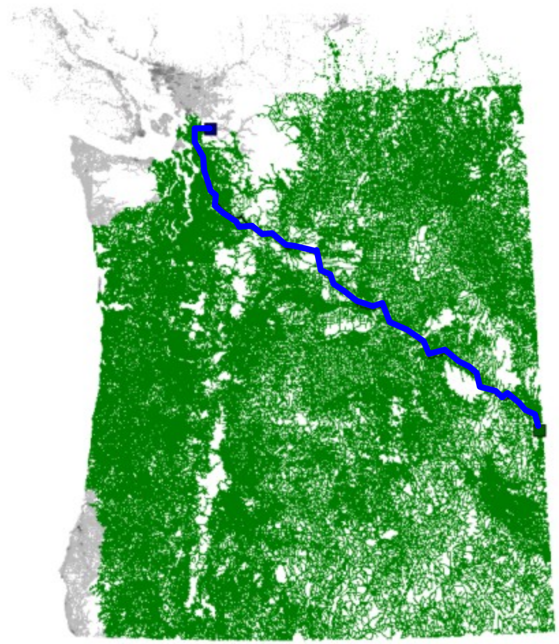
Image(s) from the paper: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter22_Faster_A_Star_with_Goal_Bounding.pdf

Pathfinding

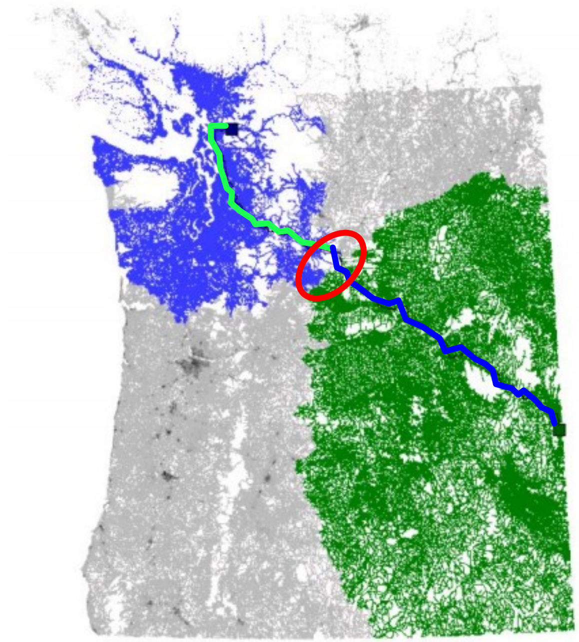
Bidirectional search

Bidirectional search
Primer

- Idea: search from both ends (start → target; target → start) until the searches meet



Dijkstra’s



Bidirectional Dijkstra’s

Images from the presentation: https://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf

Bidirectional search
Shortest paths

- Given a directed graph G with n vertices, m edges
- Every edge v → w has a length l(v, w)
- Let dist(v, w) be the shortest-path distance from v to w
- Goal: find path from start vertex s to goal vertex t with distance dist(s, t)

Bidirectional search
Review: Dijkstra's algorithm

- for every vertex v, remembers
  - d(v): shortest known distance from s to v
  - p(v): parent
  - S(v): status = unreached, frontier, expanded
- Initially d(s) = 0, p(s) = nil, S(s) = frontier
  - For all other vertices v : d(v) = ∞, p(v) = nil, S(v) = unreached
- In each iteration:
  - choose frontier vertex v with smallest d(v)
  - for each edge (v, w) in graph:
    - if d(w) > d(v) + l(v, w):
      - set d(w) = d(v) + l(v, w)
      - set p(w) = v
      - set S(w) = frontier
  - set S(v) = expanded

Bidirectional search
Dijkstra's algorithm: properties

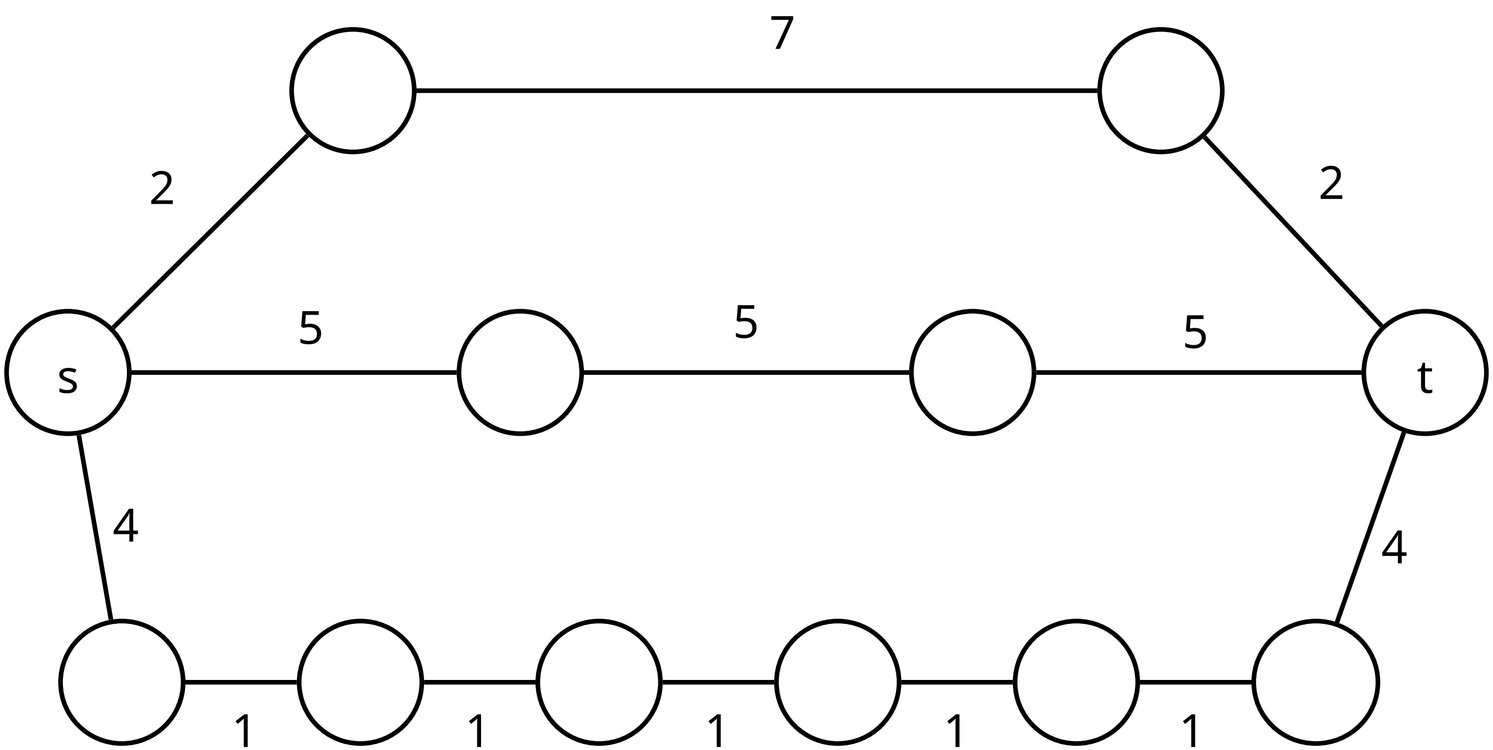
- Every vertex is expanded only once
- When a vertex v is expanded, d(v) is the shortest distance from the start s to v
- Vertices are expanded in non-decreasing order of distance from s

Bidirectional search
Dijkstra's algorithm

- A **forward search** from s to t or a **reverse search** from t to s will produce the same result
- We can run both at once!
  - Each has its own priority queue
  - Each stores independent values for d(v), p(v), S(v)
  - We write $d_f(v)$, $d_r(v)$ for the forward/reverse distances
- We alternate steps of both searches
- Stop when the searches meet
  - What does this mean, exactly?

Bidirectional search
Dijkstra's algorithm

An example



Bidirectional search
Dijkstra's algorithm

- We remember the shortest path seen so far, and its length μ (initially ∞)
- When we discover an edge (v, w) where v and w have already been expanded:
  - If $d_f(v) + l(v, w) + d_r(w) < \mu$ then update μ and the path
- Let top_f and top_r be the smallest values in the forward and reverse priority queues
- We can stop when $top_f + top_r \geq \mu$. Then we have already found the shortest path.
  - If we ever expand any vertex in both directions, the stopping condition will always be true

Bidirectional search
Dijkstra's algorithm

- Why can we stop when $top_f + top_r \geq \mu$?
  - Suppose there is a path P with length less than μ
  - So for every vertex x on P, we must have
    - dist(s, x) < top_f or dist(x, t) < top_r
  - So P must contain an edge (v, w) such that
    - dist(s, v) < top_f and dist(w, t) < top_r
  - So we have already expanded v and w
  - When we expanded the second of these, we would have already discovered this path and set μ to its length!

Bidirectional search
Review: A*

- A* uses a heuristic function h(v)
- h is **admissible** (optimistic) if for every vertex v, $h(v) \leq dist(v, t)$
- h is **consistent** if for every edge (v, w), $h(v) \leq l(v, w) + h(w)$
- Every consistent heuristic is admissible

Bidirectional search
Review: A*

- for every vertex v, A* remembers
  - g(v): shortest known distance from s to v
  - p(v): parent
  - S(v): status = unreached, frontier, expanded
- Initially g(s) = 0, p(s) = nil, S(s) = frontier
  - For all other vertices v : g(v) = ∞, p(v) = nil, S(v) = unreached
- In each iteration:
  - choose frontier vertex v with smallest f(v) = g(v) + h(v)
  - for each edge (v, w) in graph:
    - if g(w) > g(v) + l(v, w):
      - set g(w) = g(v) + l(v, w)
      - set p(w) = v
      - set S(w) = frontier
  - set S(v) = expanded

Bidirectional search
A*: graph transformation

- If h is consistent, then $h(v) \leq l(v, w) + h(w)$
  - So $l(v, w) - h(v) + h(w) \geq 0$
- Define $l_h(v, w) = l(v, w) - h(v) + h(w)$
- Consider a graph G' that's like G, but uses length function l_h
- Let $dist_h(x, y)$ be the shortest distance from x to y in G
- For all x and y, $dist_h(x, y) = dist(x, y) - h(x) + h(y)$
- A path from x to y in G is a shortest path iff it is a shortest path from x to y in G'

Bidirectional search
A*: graph transformation

- We know that
  - for all x and y, $dist_h(x, y) = dist(x, y) - h(x) + h(y)$
- A* on graph G is the same as Dijkstra's on G' !
  - A* on G picks vertex with smallest
    - $f(v) = g(v) + h(v) = dist(s, v) + h(v)$
  - Dijkstra's on G' picks vertex with smallest
    - $d_h(v) = dist_h(s, v) = dist(s, v) - h(s) + h(v)$
  - h(s) is constant, so these are the same

Bidirectional search

Bidrectional A*

- We need two heuristic functions
  - $h_f(v)$: estimate of dist(v, t)
  - $h_r(v)$: estimate of dist(s, v)
- We want these functions to produce the same transformed graph
  - so we can use the stopping criterion from bidirectional Dijkstra's
- Forward: $l_f(v, w) = l(v, w) - h_f(v) + h_f(w)$
- Reverse: $l_r(w, v) = l(v, w) - h_r(w) + h_r(v)$
- For all edges (v, w), we need $l_f(v, w) = l_r(w, v)$
  - so $h_f(v) + h_r(v) = h_f(w) + h_r(w)$
  - The function $(h_f + h_r)$ must be constant!
  - Most heuristics (e.g. Euclidean distance) are not like that

Bidirectional search
Bidrectional A*

- Given heuristic functions h_f, h_r
- Define
  - $p_f(v) = (h_f(v) – h_r(v)) / 2$
  - $p_r(v) = (h_r(v) – h_f(v)) / 2$
  - Then $p_f(v) + p_r(v) = 0$
  - We can show that p_f and p_r are consistent / admissable
- Refinement
  - $p_f(v) = (h_f(v) – h_r(v) + h_r(t)) / 2$
  - $p_r(v) = (h_r(v) – h_f(v) + h_f(s)) / 2$
  - Now $p_f(t) = p_r(s) = 0$
  - $p_f + p_r$ is still a constant function
  - p_f and p_r are still consistent / admissable

Bidirectional search
Bidrectional A*

- Using heuristic functions p_f and p_v, A* is equivalent to Dijkstra's on the graph G'
- Bidirectional Dijkstra's stops when $top_f' + top_r' \geq \mu'$
- Let v_f be the top element in the forward heap
  - $top_f = dist(s, v_f) + p_f(v_f)$
  - $top_f' = dist_{pf}(s, v_f) = dist(s, v_f) – p_f(s) + p_f(v_f)$
  - So $top_f' = top_f – p_f(s)$
  - Similarly, $top_r' = top_r - p_r(t)$
- $\mu' = dist_{pf}(s, t) = \mu – p_f(s) + p_f(t)$
- So we can stop when
  - $[top_f – p_f(s)] + [top_r – p_r(t)] \geq \mu – p_f(s) + p_f(t)$
- Simplifying and using $p_f(t) = 0$, we have
  - $top_f + top_r \geq \mu + p_r(t)$

Pathfinding
The Algorithms

How to find a path in an unknown environment?

Dynamic searches

(peeking into the field of robotics)

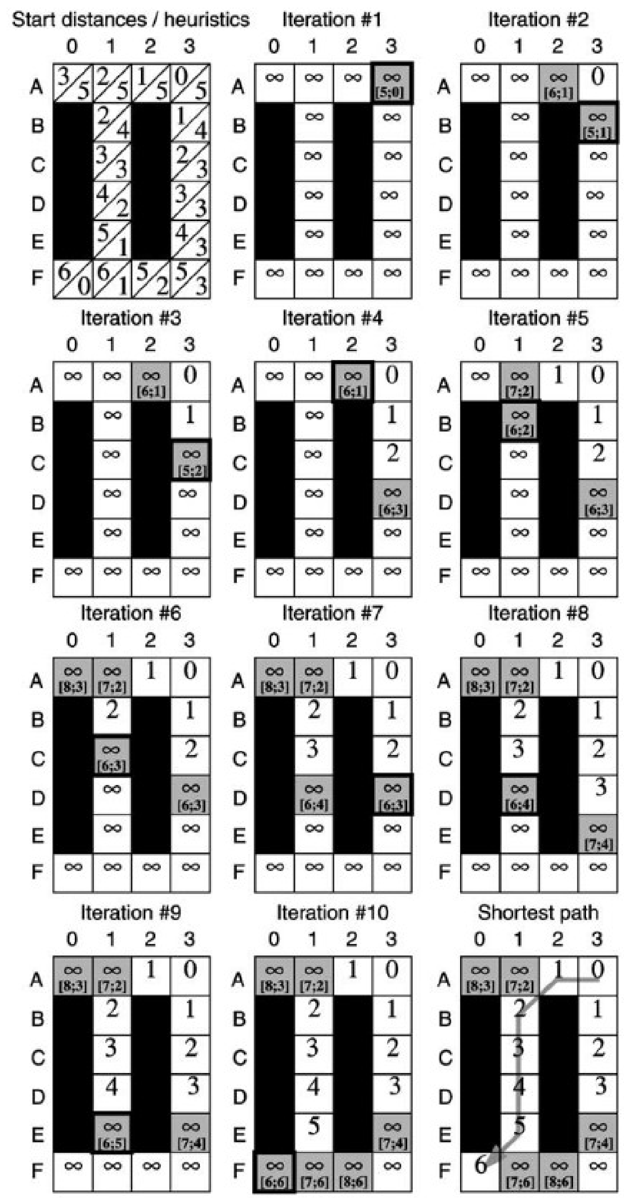
Dynamic pathfinding
Problem statement

The Problem
The environment is not known in advance and is being updated online based on sensor readings.
How to find a path to the target?

D* Lite; based on Lifelong Planning A* (LPA*)
Koenig, S., & Likhachev, M. (2002). D* Lite. *Aaai/iaai*, *15*.

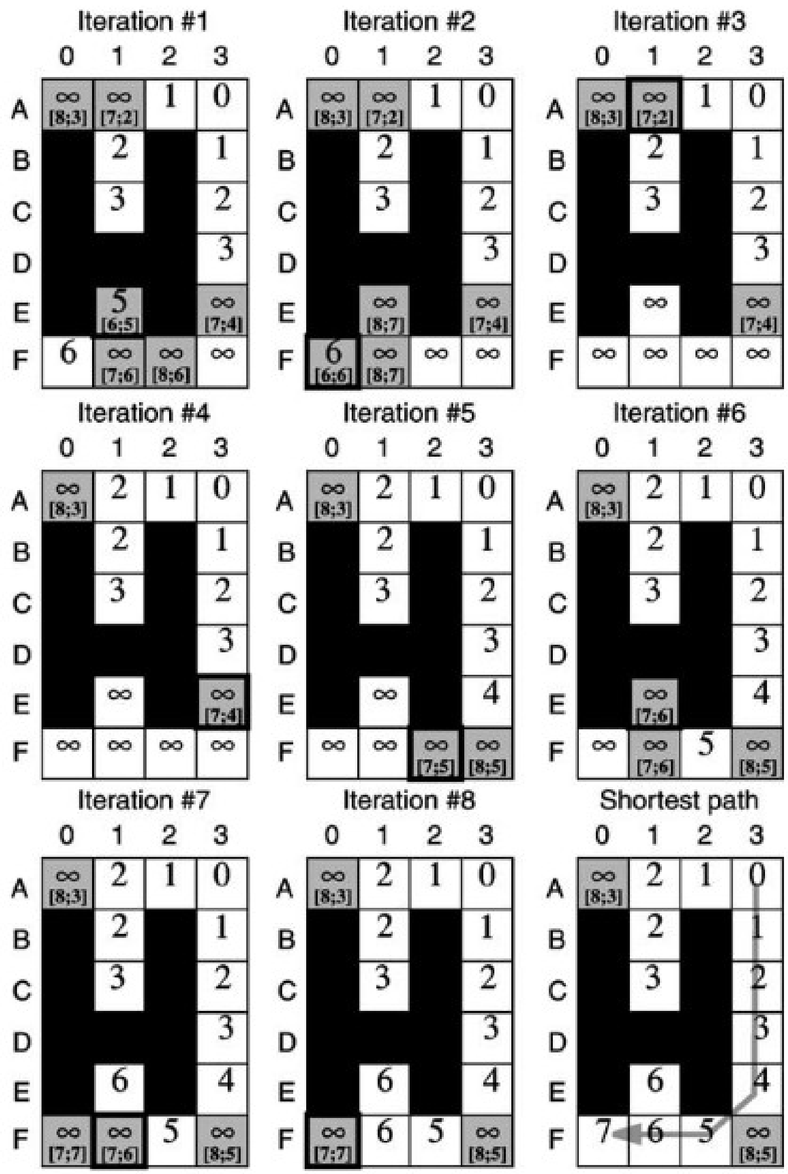
Lifelong Planning A*

Example: first search



Lifelong Planning A*

Example: second search



Lifelong Planning A* (LPA*)

- For each vertex s, LPA* maintains two estimates of the shortest distance to it:
  - g(s): like in A*
  - rhs(s): a one-step lookahead
    - $rhs(s_{start}) = 0$
    - for other states s,
      - rhs(s) = min (g(s') + c(s', s)) over all neighbors s'
- s is called **locally consistent** if g(s) = rhs(s)
  - If g(s) < rhs(s), it is **locally underconsistent**. If g(s) > rhs(s), it is **locally overconsistent**.
- If all vertices are locally consistent, then all values g(s) are shortest-path distances
- The open list holds all locally inconsistent vertices
- It is a priority queue ordered lexicographically by $k(s) = [k_1(s); k_2(s)]$, where
  - $k_1(s) = \min(g(s), rhs(s)) + h(s)$
  - $k_2(s) = \min(g(s), rhs(s))$

Lifelong Planning A*

```
procedure CalculateKey(s)
  return [min(g(s), rhs(s)) + h(s); min(g(s), rhs(s))];

procedure Initialize()
  U = Ø;
  for all s ∈ S rhs(s) = g(s) = ∞;
  rhs(s_start) = 0;
  U.Insert(s_start, [h(s_start); 0]);

procedure UpdateVertex(u)
  if (u ≠ s_start) rhs(u) = min_{s' ∈ pred(u)} (g(s') + c(s', u));
  if (u ∈ U) U.Remove(u);
  if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
  while (U.TopKey() ⋖ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
      u = U.Pop();
      if (g(u) > rhs(u))
          g(u) = rhs(u);
          for all s ∈ succ(u) UpdateVertex(s);
      else
          g(u) = ∞;
          for all s ∈ succ(u) ∪ {u} UpdateVertex(s);

procedure Main()
  Initialize();
  forever
    ComputeShortestPath();
    Wait for changes in edge costs;
    for all directed edges (u, v) with changed edge costs
      Update the edge cost c(u, v);
      UpdateVertex(v);
```

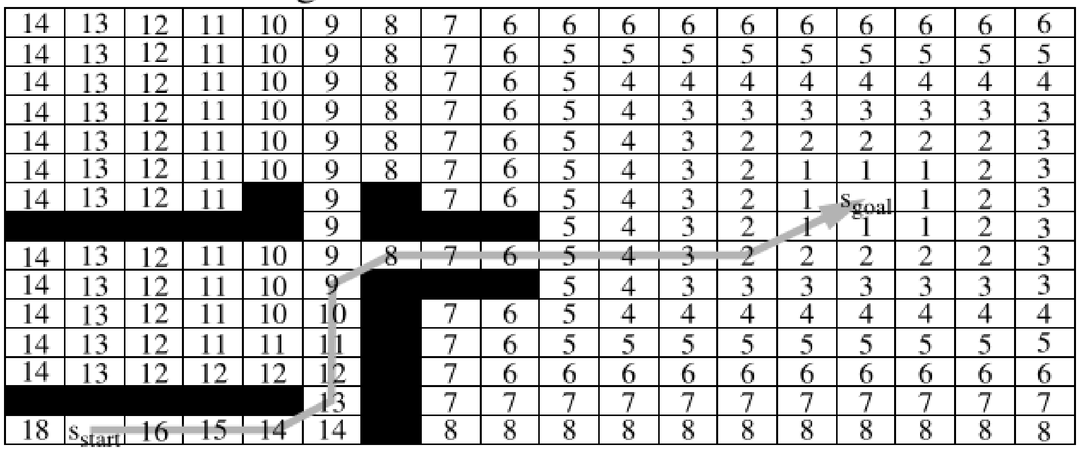
D* Lite

D* Lite (Koenig, 2002)

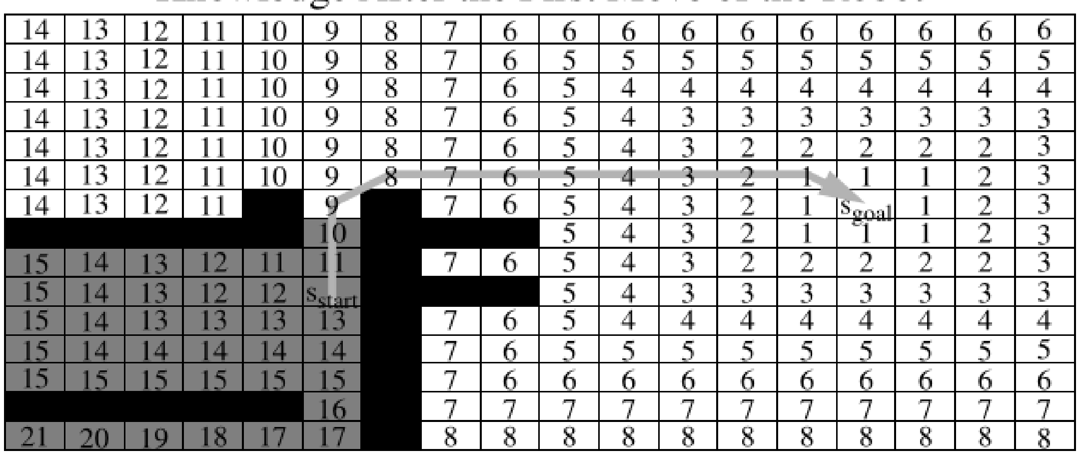
- An adaptation of LPA* for dynamic planning
- Now the start vertex may change each time we replan!

D* Lite

Knowledge Before the First Move of the Robot



Knowledge After the First Move of the Robot



D* Lite

D* Lite

- First modification of LPA*: we must search **backwards** from the goal to the start
  - because we always want shortest paths to the fixed goal
  - Now the heuristic function estimates the distance from any position to the robot
- Second modification: every time the robot moves, the heuristic function changes, so we must recalculate all priorities

D* Lite: Basic Version

```
procedure CalculateKey(s)
return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];

procedure Initialize()
U = ∅;
for all s ∈ S rhs(s) = g(s) = ∞;
rhs(s_goal) = 0;
U.Insert(s_goal, CalculateKey(s_goal));

procedure UpdateVertex(u)
if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
if (u ∈ U) U.Remove(u);
if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
while (U.TopKey() <̇ CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
   u = U.Pop();
   if (g(u) > rhs(u))
      g(u) = rhs(u);
      for all s ∈ Pred(u) UpdateVertex(s);
   else
      g(u) = ∞;
      for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);

procedure Main()
Initialize();
ComputeShortestPath();
while (s_start ≠ s_goal)
   /* if (g(s_start) = ∞) then there is no known path */
   s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
   Move to s_start;
   Scan graph for changed edge costs;
   if any edge costs changed
      for all directed edges (u, v) with changed edge costs
         Update the edge cost c(u, v);
         UpdateVertex(u);
      for all s ∈ U
         U.Update(s, CalculateKey(s));
      ComputeShortestPath();
```

D* Lite

D* Lite

- Updating all queue priorities is expensive
- We can modify the algorithm so each queue priority is only a **lower bound** on the actual priority
- When the robot moves from s to s', actual priorities can decrease by at most h(s, s')
- Instead of subtracting h(s, s') from all priorities, we can accumulate the decrease into a variable k_m and then apply it to any vertex when it comes out of the queue
- For details and pseudocode, see the D* Lite paper

Pathfinding
The Algorithms

How to find the path in an unknown environment faster?

Beating D* Lite (almost every time)

(peeking into the field of robotics)

Multipath Adaptive A* (MPAA*)
Speeding up A* to outperform D* Lite

Multipath Adaptive A*
Hernández, C., Baier, J. A., & Asín, R. (2014, May). Making A* run faster than D*-Lite for path-planning in partially known terrain.

- Proposes Multipath Adaptive A* as a simpler approach, usually faster than D* Lite

Adaptive A*

- Let gd[s] be the minimal cost from state s to the goal
- Let f* = gd[s_{start}] be the minimal cost found by an A* search
- For any state s that was expanded,
  - g[s] is the minimal cost from the start to s
  - f[s] = g[s] + h[s]
- Now
  - f* ≤ g[s] + gd[s]
  - f* - g[s] ≤ gd[s]
  - So f* - g[s] is an admissable estimate of gd[s]
  - We can use this as a new heuristic value for s
- Also, since s was expanded, we have
  - f[s] ≤ f*
  - g[s] + h[s] ≤ f*
  - h[s] ≤ f* - g[s]
  - So the new heuristic value f* - g[s] dominates the old

Adaptive A*: pseudocode (part 1)

```
procedure InitializeState(s)
    if search(s) ≠ counter then
        g(s) ← ∞
    search(s) ← counter

procedure A*(s_init)
    InitializeState(s_init)
    parent(s_init) ← null
    g(s_init) ← 0
    Open ← ∅
    insert s_init into Open with f-value g(s_init) + h(s_init)
    Closed ← ∅
    while Open ≠ ∅ do
        remove a state s from Open with the smallest f-value g(s) + h(s)
        if GoalCondition(s) then
            return s
        insert s into Closed
        for each s' ∈ succ(s) do
            InitializeState(s')
            if g(s') > g(s) + c(s, s') then
                g(s') ← g(s) + c(s, s')
                parent(s') ← s
                if s' is in Open then
                    set priority of s' in Open to g(s') + h(s')
                else
                    insert s' into Open with priority g(s') + h(s')
    return null
```

Adaptive A*: pseudocode (part 2)

```
procedure BuildPath(s)
    while s ≠ s_start do
        next(parent(s)) ← s
        s ← parent(s)

procedure Observe(s)
    for each arc (t, t') in the range of visibility from s do
        if cost of (t, t') has increased then
            update c(t, t')
            next(t) ← null

procedure main()
    counter ← 0
    Observe(s_start)
    for each state s ∈ S do
        search(s) ← 0
        h(s) ← H(s, s_goal)
        next(s) ← null
    while s_start ≠ s_goal do
        counter ← counter + 1
        s ← A*(s_start )
        if s = null then
            return "goal is not reachable"
        for each s' ∈ Closed do
            h(s') ← g(s) + h(s) − g(s') /* heuristic
                */
        BuildPath(s)
        while no action cost has just increased in path[s_start] d
            t ← s_start
            s_start ← next(s_start)
            next(t) ← null
            Move agent to s_start
            Observe(s_start)
```

Multipath Adaptive A*

- Suppose we select a state s such that
  - s belongs to a previously found path σ
  - the suffix of σ starting in s is a provably optimal path from s to the goal
- Then we can stop the search immediately
- We can check these conditions easily

function $GoalCondition(s)$

  while $next(s) \neq null \text{ and } h(s) = h(next(s)) + c(s, next(s))$ **do**

    $s \leftarrow next(s)$

  return $s_{goal} = s$

Pathfinding
The Algorithms

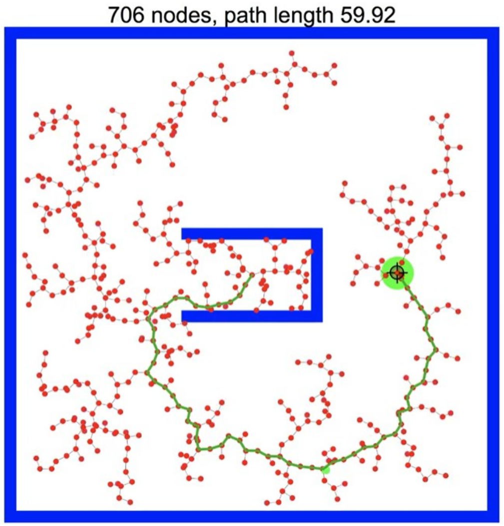
How to find a path in a continuous space?
Rapidly exploring random trees
(RRT, RRT*)
(peeking into the field of robotics)

Rapidly-exploring Random Trees
RRT and RRT*

RRT

LaValle, S. M., & Kuffner Jr, J. J. (2001). [Randomized kinodynamic planning.](#) *The international journal of robotics research*, *20*(5), 378-400.

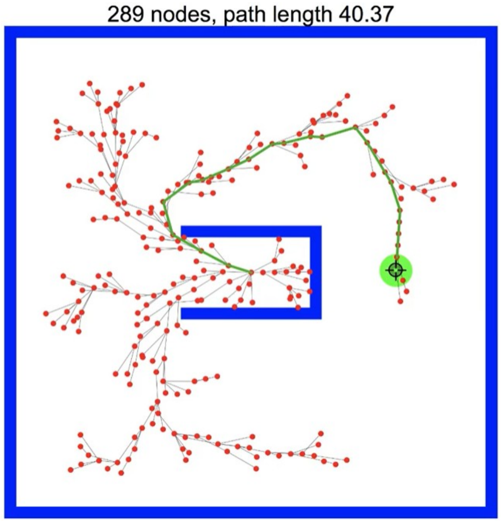
IDEA: Randomly throw a point, find nearest existing point of RRT, make a step from that point towards the random point. Repeat.



RRT*

Karaman, S., & Frazzoli, E. (2011). [Sampling-based algorithms for optimal motion planning.](#) *The international journal of robotics research*, *30*(7), 846-894.

IDEA: Do RRT but then try to rewire the tree around new point to be more optimal.



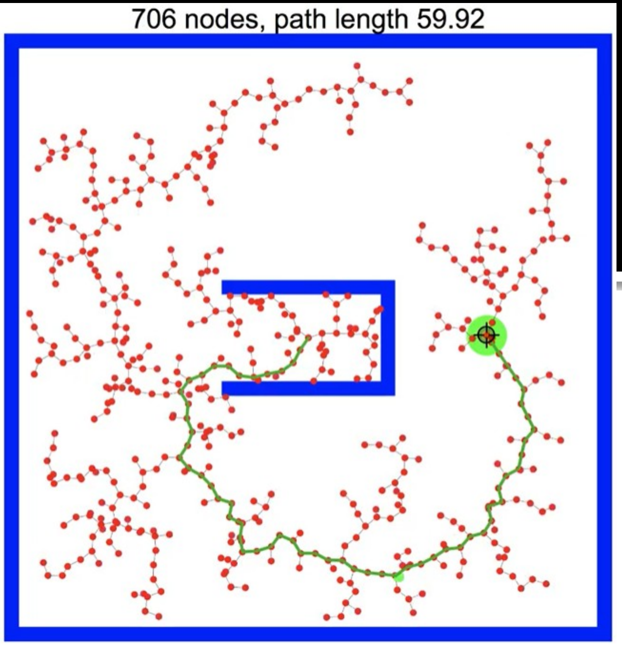
Rapidly-exploring Random Trees

RRT and RRT*

Beautiful explanation available on YouTube
by Aaron Becker.

- YouTube video: "RRT, RRT* & Random Trees" (Aaron Becker)

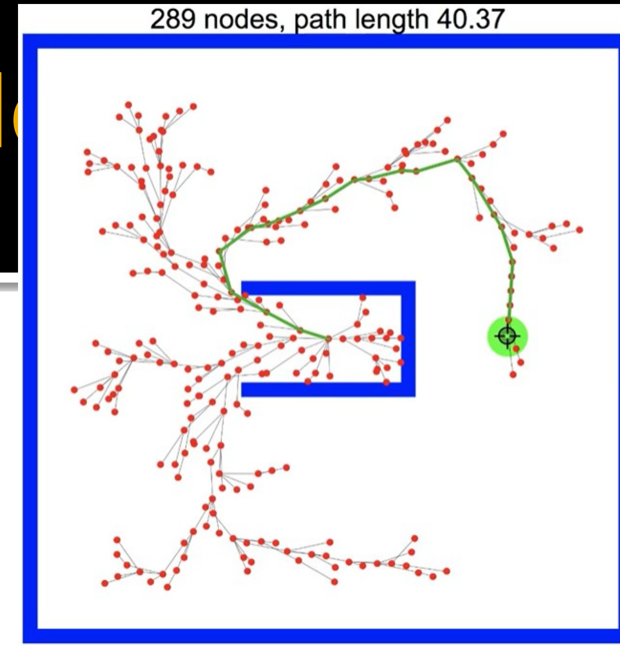
Rapidly-exploring Rand
Algorithms



RRT (trying to find path to *target* with certain *epsilon*)

```
G.init(root: Point)
while path to epsilon-area
      around target not found
  point := random
  nearest := G.nearest_vertex(point)
  new_vertex =
    nearest + (point - nearest).normalized
                                  * step
  if no obstacle nearest->new_vertex
    G.add_edge(nearest, new_vertex)
```

Rapidly-exploring Rand
Algorithms



RRT* (trying to find path to *target* with certain *epsilon*)

```
G.init(root: Point)
while path to epsilon-area
        around target not found
  point := random
  nearest := G.nearest_vertex(point)
  new_vertex =
    nearest + (point - nearest).normalized
                                    * step
  if no obstacle nearest->new_vertex
    min_cost_vertex =
      find vertex
        | from G.vertices_around(new_vertex)
        | with min path cost from root
        | and no obstacle on vertex->new_vertex
    G.add_edge(min_cost_vertex, new_vertex)
    for vertex in G.vertices_around(new_vertex)
      if obstacle on vertex->new_vertex
        continue
      if path_cost(root, vertex) >
         path_cost(root, new_vertex) + |vertex,new_vertex|
        G.remove_edge(parent(vertex), vertex)
        G.add_edge(new_vertex, vertex)
```

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

MINISTERSTVO ŠKOLSTVÍ,
MLÁDEŽE A TĚLOVÝCHOVY

Material has been produced within and supported by the project
„Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“
kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.