Approximate path planning

Computational Geometry
csci3250
Laura Toma
Bowdoin College

Path planning

- Combinatorial <—— last time
- Approximate <—— next

Combinatorial path planning

- **Idea: Compute free C-space combinatorially (= exact)**
- **Approach**
  - (robot, obstacles) => (point robot, C-obstacles)
  - Compute roadmap of free C-space
    - any path: trapezoidal decomposition or triangulation
    - shortest path: visibility graph

- **Comments**
  - Complete
  - Works beautifully in 2D and for some cases in 3D
    - Worst-case bound for combinatorial complexity of C-objects in 3D is high
  - Unfeasible/intractable for high #DOF
    - A complete planner in 3D runs in $O(2^{n\text{^}\#DOF})$

You cannot compute C-free.

Imagine being blind-folded in a maze.

Approximate path planning

- **Idea**: **Since you can’t compute C-free, approximate it**

- Approaches
  - Graph search methods on grid-graphs
    - A*, weighted A*, D*, ARA*,…
  - Sampling-based
    - PRM-type
    - RRT-type
  - Potential field
  - Hybrid approaches

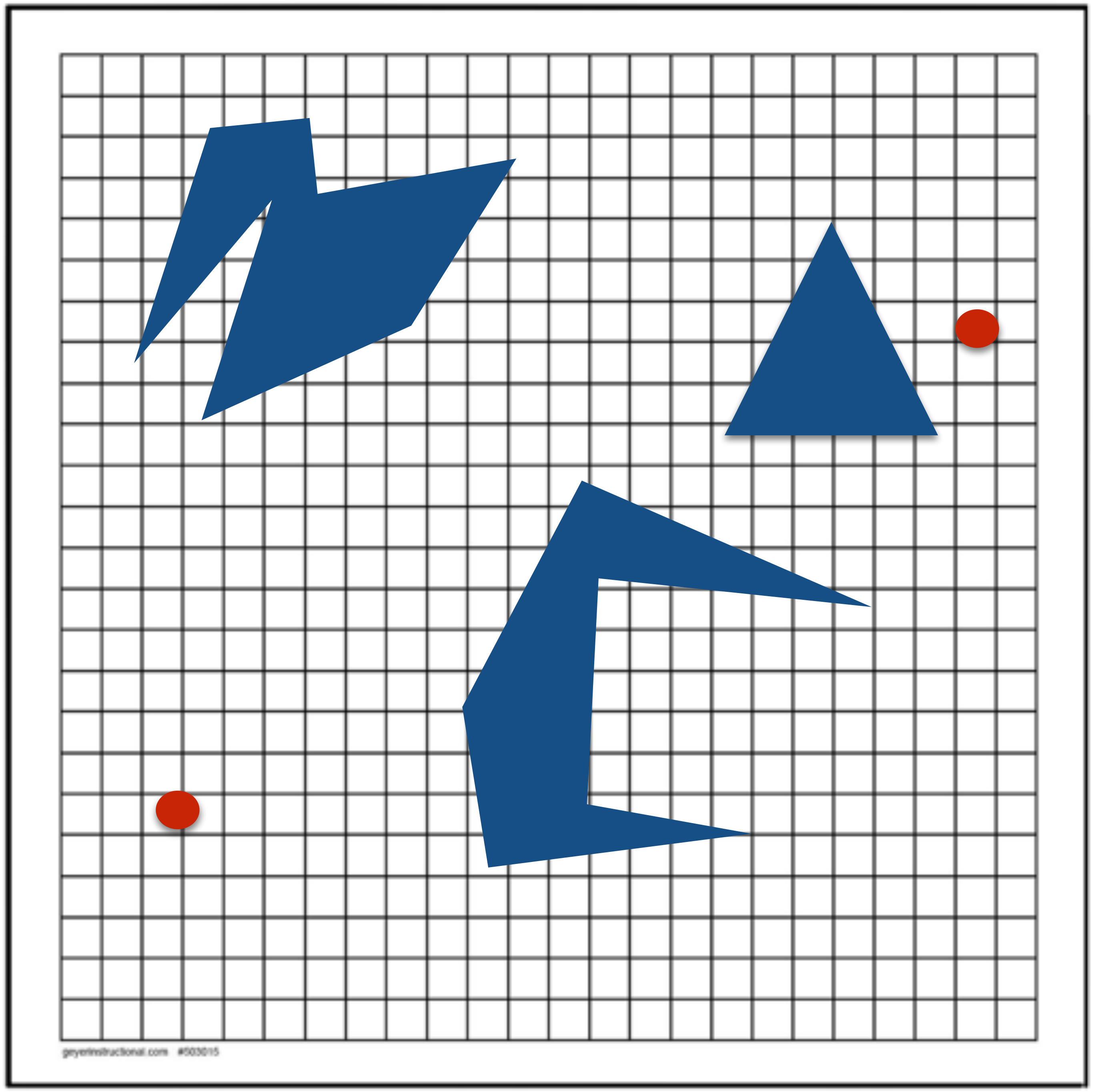
Approximate path planning

The concept of completeness is relaxed

- A planner is **resolution complete**:
    - finds a solution, if one exists, with probability —> 1 as the resolution of the sampling increases
- A planner is **probabilistically complete**:
    - finds a solution, if one exists, with probability —> 1 as computation time increases

Grid-based planners with A*

Grid-based planners with A*


geyerinstructional.com #503015

Grid-based graph search

- Sample C-space with uniform grid/lattice
  - This essentially “pixelizes" the space (pixels/voxels in C-free)
  - (refined: quadtree/octree)
- Graph is implicit
  - given by lattice topology: move +/-1 in each direction, possibly diagonals as well
  - successors(state s)
- Search the graph for a path from start to end
  - Dijkstra/A* + variants

- Variants: Graph can be pre-computed (occupancy grid), or computed incrementally
  - one-time path planning vs many times; static vs dynamic environment

Remember Dijkstra’s algorithm?

- Best-first search
    - priority(v) = d[v] = cost of getting from s to v
- Dijkstra’s algorithm
    - Initialize: d[v] = inf for all v, d[s] = 0
    - Repeat: select the **best** vertex (=smallest) priority), and relax its edges

```
Dijkstra(vertex s)
    • initialize
        • d[v] = infinity for all v, d[s] = 0
    • for all v: PQ.insert(<v, d[v]>)
    • while PQ not empty
        • u = PQ.deleteMin()
        • mark u as done  //claim: d[u] is the
        • for each edge (u,v):
            • if v not done, and if d[v] > d[u] + edge(u,v):
                • d[v] = d[u] + edge(u,v)
                • PQ.decreasePriority(v, d[v])
```

no need to check if v is done,
because once v is done,
no subsequent relaxation can improve its d[]

usually not implemented

- **data structures**
  - for each vertex u, keep: done[u] and d[u]
  - PQ of (v, d[v])

```
Dijkstra(vertex s)
  • initialize
      • d[v] = infinity for all v, d[s] = 0
  • PQ.insert(<s, d[s]>)
  • while PQ not empty
      • u = PQ.deleteMin()
      • if u not done, for each edge (u,v):
          • if d[v] > d[u] + edge(u,v):
              • d[v] = d[u] + edge(u,v)
              • PQ.insert(<v, d[v]>)
      • mark u as done
```

1. insert only the start

2. insert it (even if it's already there)

3.because we avoid decreaseKey, PQ may contain the same vertex with different d[]. Only the first time we see u we process it

```
Dijkstra(vertex s)
    • initialize
        • d[v] = infinity for all v, d[s] = 0
    • PQ.insert(<s, d[s]>)
    • while PQ not empty
        • u = PQ.deleteMin()
        • if u not done, for each edge (u,v):
            • if isFree(v) and d[v] > d[u] + edge(u,v):
                • d[v] = d[u] + edge(u,v)
                • PQ.insert(<v, d[v]>)
        • mark u as done
```

1. insert only the start

2. isFree(v): is v in C-free

3. insert it
(even if it's already there)

What to do with a partially blocked cell?

Dijkstra and A*

- Dijkstra:
  - best-first search
  - priority(v) = d[v] = distance from source
- A*
  - **priority f(v) = g(v) + h(v)**
    - g(v): cost of getting from start to v
    - h(v): estimate of the cost from v to goal
  - Dijkstra is (A* with h(v) = 0)
  - Admissibility
    - h(v) is “admissible” if h(v) < trueCost(v—>goal)
    - Theorem: If h() is admissible then A* will return an optimal solution.
    - In general it may be hard to estimate h(v)
      - path planning: h(v) = EuclidianDistance(v, goal)

Grid-based graphs: Dijkstra and A*

- A* explores fewer vertices to get to the goal, compared to Dijkstra
  - The closer h(v) is to the trueCost(v), the more efficient
- Example
  - https://www.youtube.com/watch?v=DINCL5cd_w0

- Many A* variants
  - weighted A*
    - c × h() ==> solution is no worse than (1+c) × optimal
  - anytime A*
    - use weighted A* to find a first solution ; then use A* with first solution as upper bound to prune the search
  - real-time replanning
    - if the underlying graph changes, it usually affects a small part of the graph ==> don’t run search from scratch
    - D*: efficiently recompute SP every time the underlying graph changes

Graph search methods on grid-graphs

- Comments
    - **Resolution complete (**probability of finding a solution, if one exists, —> 1 as the resolution of the grid increases)
    - The paths may be longer than true shortest path in C-space
    - Can interleave the construction with the search (ie construct only what is necessary)

⊕

- simple to understand/implement
- work in any dimension

⊖

- size and quality of path depends on the discretization of the problem
- suffers in high-d spaces => slow

Sampling-based planners

Sampling-based planning

- Motivation
  - Combinatorial: hard to construct C-obstacles exactly when D is high
  - Grid-based : space is too large when D is high
    - e.g. DOF= 6: 1000 x 1000 x 1000 x 360 x 360 x 360

Idea: Be smart about how to choose the points to sample!

- Goal: generate a sparse (sample-based) representation of free C-space

Sampling-based planning

- **Multiple-query planners**
  - Construct a graph (Roadmap)
    - Sample C-free and compute a roadmap that captures its connectivity
    - e.g. include PRM and variants
  - Use roadmap for any (start, end) pairs

- **Single-query**
  - one start and one end state
  - construct a graph trying to connect start and end
  - e.g. include RRT and variants

Multiple-queries

- The Roadmap
  - Sample C-free and compute a roadmap that captures its connectivity to the best of our (limited) knowledge
- Roadmap construction phase
  - Start with a sampling of points in C-free and try to connect them
  - Two points are connected by an edge if a simple quick planner can find a path between them
  - This will create a set of connected components
- Roadmap query phase
  - Use roadmap to find path between any two points

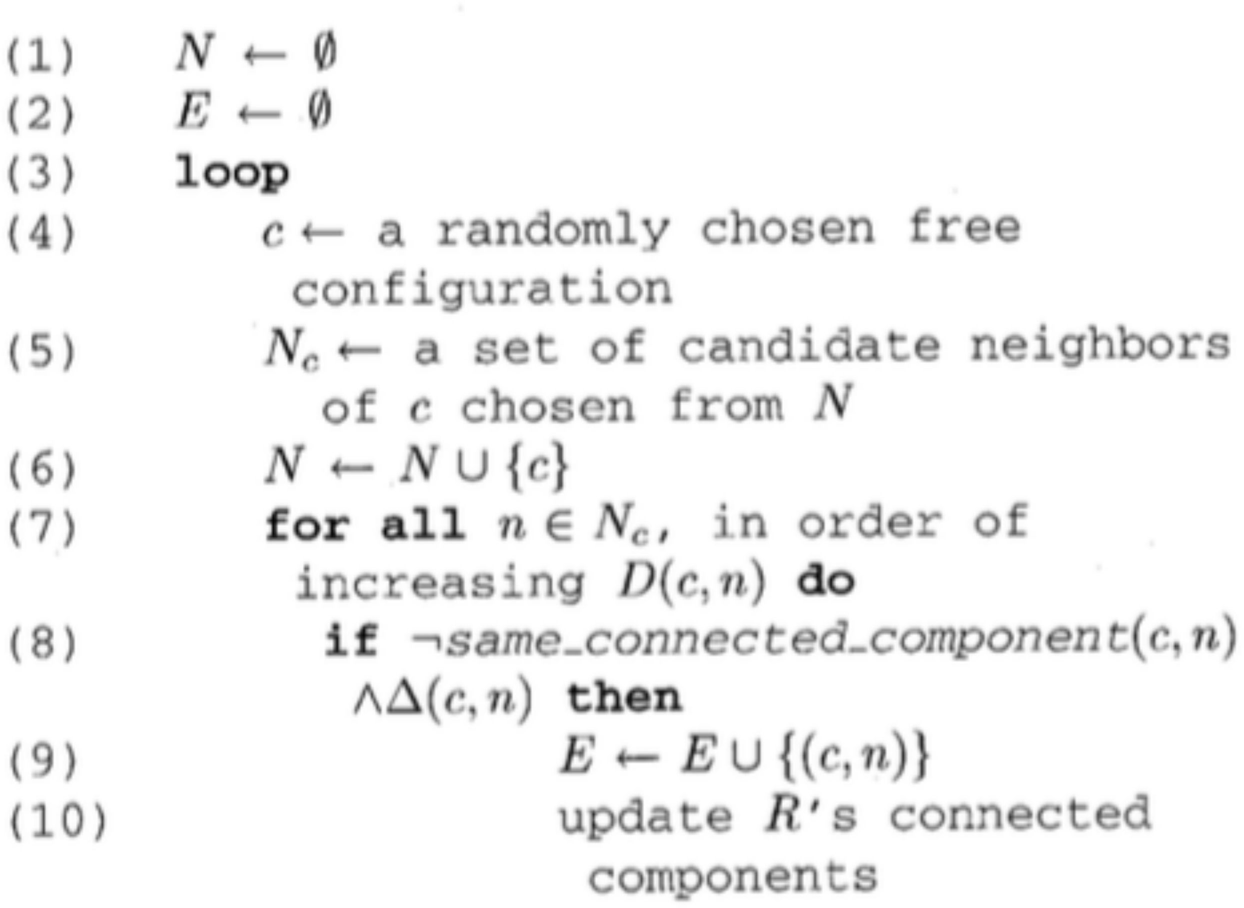
Multiple-queries

```
Generic-Sampling-based-roadmap:
    V = $p_{start}$ + sample_points(C, n);    E = { }
    for each point x in V:
        for each neighbor y in neighbors(x, V):
            //try to connect x and y
            if collisionFree(segment xy): E = E + xy
    return (V, E)
```

- Algorithms differ in
  - sample_points(C, n) : how they select the initial random samples from C
    - return a set of n points arranged in a regular grid in C
    - return random n points
  - neighbors(x, V) : how they select the neighbors
    - return the k nearest neighbors of x in V
    - return the set of points lying in a ball centered at x of radius r
  - Often used: samples arranged in a 2-dimensional grid, with nearest 4 neighbors (2^d)

Probabilistic Roadmaps (Kavraki, Svetska, Latombe, Overmars et al , 1996)

- Start with a *random* sampling of points in C-free
- Roadmap stored as set of *trees* for space efficiency
    - trees encode connectivity, cycles don't change it. Additional edges are useful for shorter paths, but not for completeness
- Augment roadmap by selecting additional sample points in areas that are estimated to be "difficult"



```
$N \leftarrow \emptyset$
$E \leftarrow \emptyset$
loop
    $c \leftarrow$ a randomly chosen free
      configuration
    $N_c \leftarrow$ a set of candidate neighbors
      of $c$ chosen from $N$
    $N \leftarrow N \cup \{c\}$
    for all $n \in N_c$, in order of
     increasing $D(c,n)$ do
      if $\neg$same_connected_component$(c,n)$
       $\wedge \Delta(c,n)$ then
            $E \leftarrow E \cup \{(c,n)\}$
            update $R$'s connected
             components
```

- Components
    - sampling C-free: random sampling
    - selecting the neighbors: within a ball of radius r
    - the local planner delta(c,n): is segment cn collision free?
    - the heuristical measure of difficulty of a node

Probabilistic Roadmaps (Kavraki, Svetska, Latombe, Overmars et al , 1996)

Comments

- One of the leading motion planning technique
- Efficient, easy to implement, applicable to many types of scenes
- Roadmap adjusts to the density of free space and is more connected around the obstacles
- Size of roadmap can be adjusted as needed
- More time spent in the “learning” phase ==> better roadmap
- Shown to be **probabilistically complete**
  - probability that the graph contains a valid solution —> 1 as number of samples increases
- Embraced by many groups, many variants of PRM’s, used in many type of scenes/applications.
    - PRM*, FMT* (fast marching tree), …
- Well-suited for high D planning

Single-query: Incremental search planners

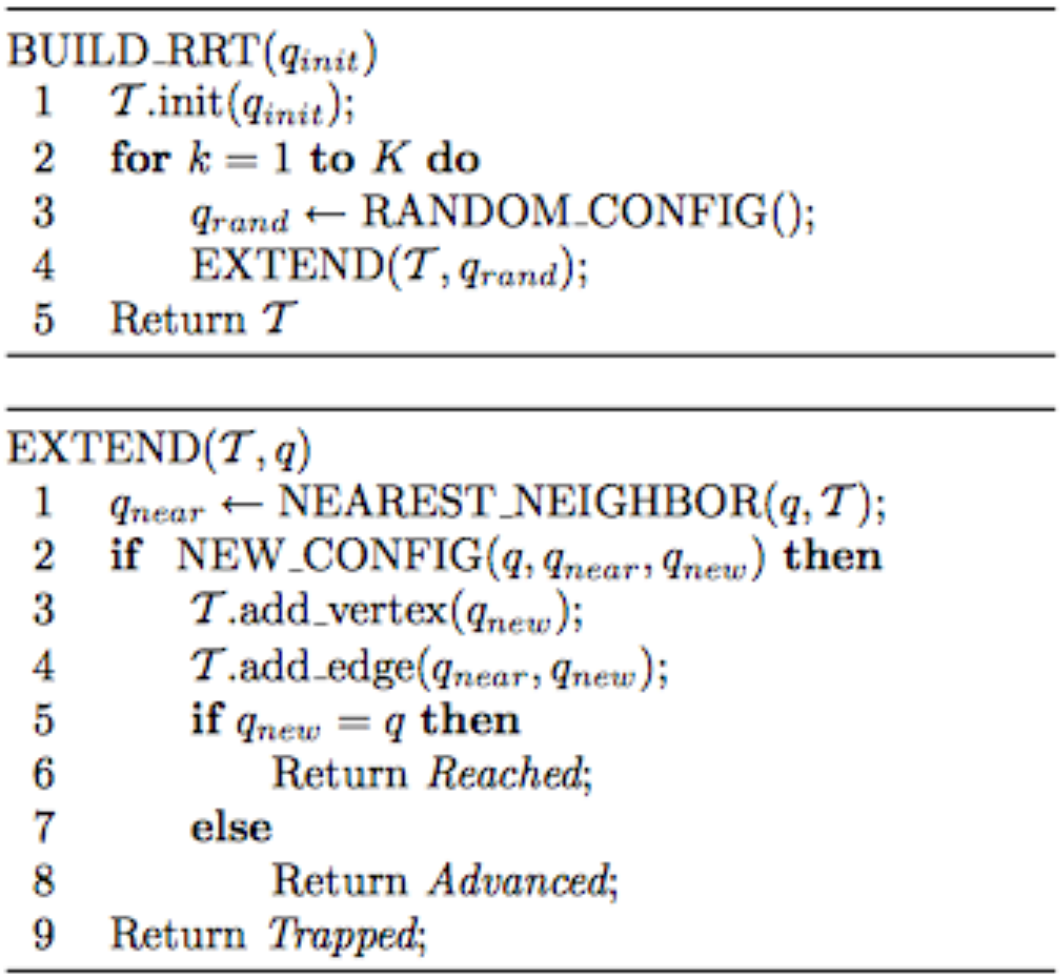
- Incrementally build increasingly finer discretization of the configuration space, while trying to determine if a path exists at each step

The RRT

- RRT (LaValle, 1998)
- Idea: Incrementally grow a tree rooted at “start” outwards

RRT demo

https://www.youtube.com/watch?v=MT6FyoHefgY



```
BUILD_RRT($q_{init}$)
    $\mathcal{T}$.init($q_{init}$);
    for $k = 1$ to $K$ do
        $q_{rand} \leftarrow$ RANDOM_CONFIG();
        EXTEND($\mathcal{T}, q_{rand}$);
    Return $\mathcal{T}$
```

```
EXTEND($\mathcal{T}, q$)
    $q_{near} \leftarrow$ NEAREST_NEIGHBOR($q, \mathcal{T}$);
    if NEW_CONFIG($q, q_{near}, q_{new}$) then
        $\mathcal{T}$.add_vertex($q_{new}$);
        $\mathcal{T}$.add_edge($q_{near}, q_{new}$);
        if $q_{new} = q$ then
            Return Reached;
        else
            Return Advanced;
    Return Trapped;
```

Figure 2: The basic RRT construction algorithm.

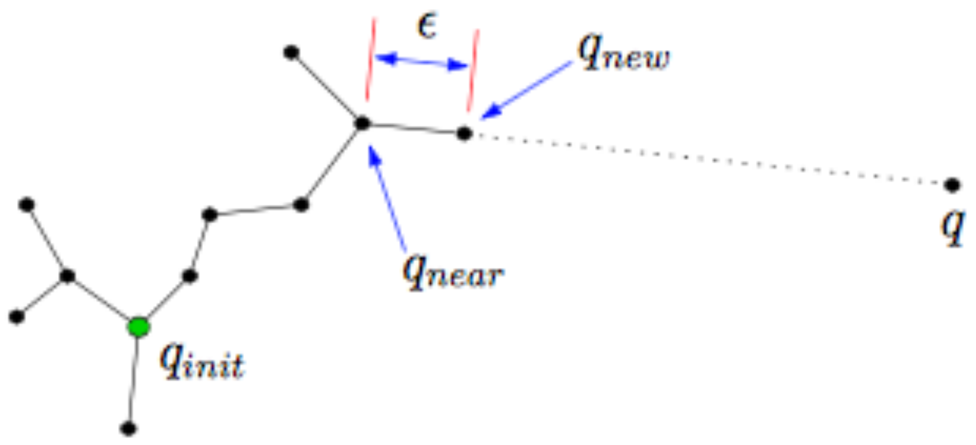


Figure 3: The EXTEND operation.

https://www.cs.cmu.edu/afs/cs/academic/class/15494-s12/readigs/kuffner_icra2000.pdf

Single-query: Incremental search planners

- + Probabilistic complete
- + Scales well to higher -d
- + no discretization (sample from a continuous space)
- - time may be unbounded

Timeline and developments

- Dijkstra 1950s
- A* 1960s
- RRT 1998
- RRT* 2010
- …

Sampling-based planning

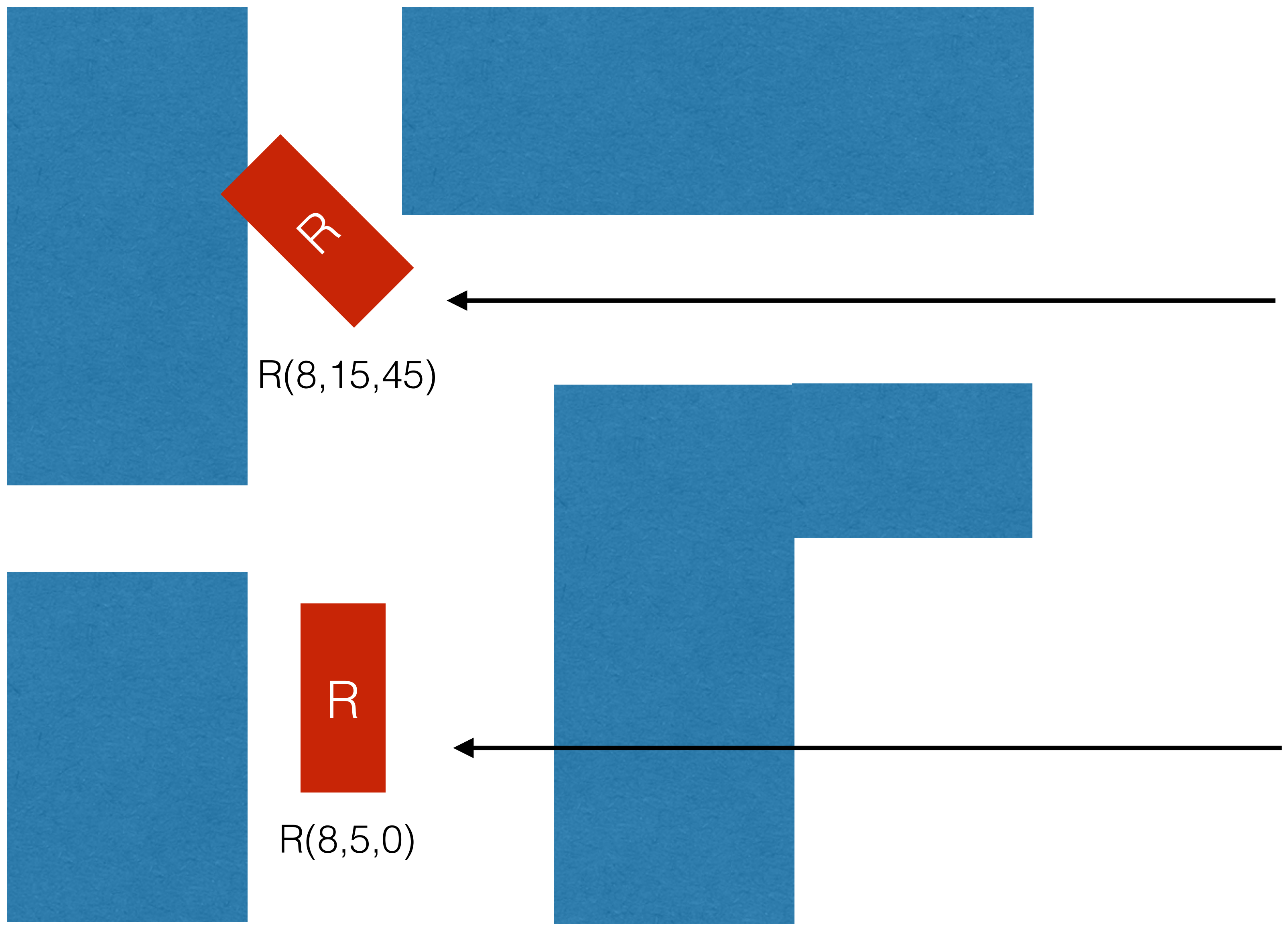
- The main function to write

  isFree((x,y,…), Robot, Obstacles): would my robot , if placed in this configuration, intersect any obstacle

2D: robot can translate and rotate

C-space: 3D

configuration p: (x, y, theta)



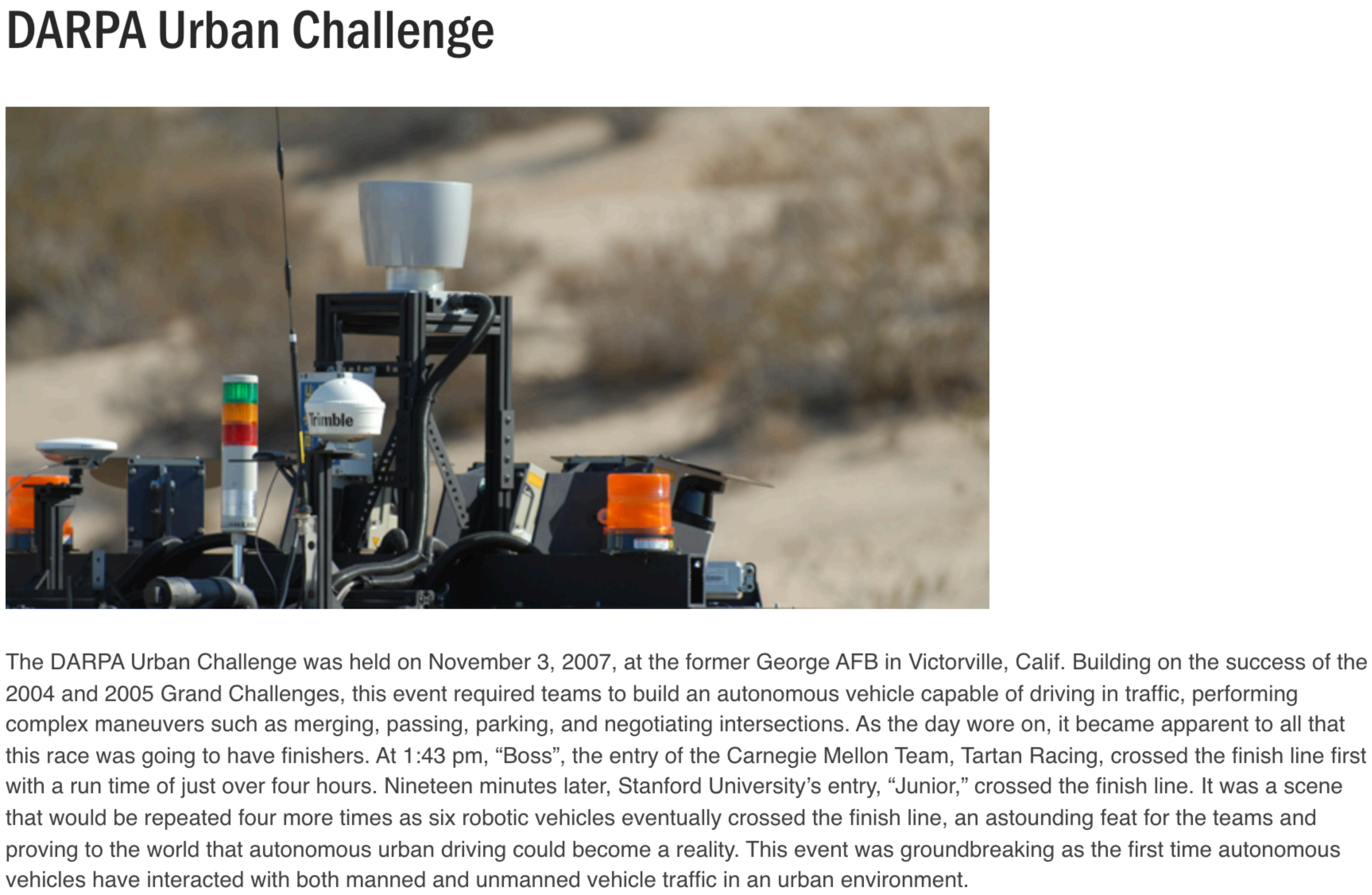
(8,15,45): not free

(8,5,0): free

How would you write: isFree((x,y,theta), Robot R, Obstacles S) ?

Demos

https://www.darpa.mil/about-us/timeline/darpa-urban-challenge



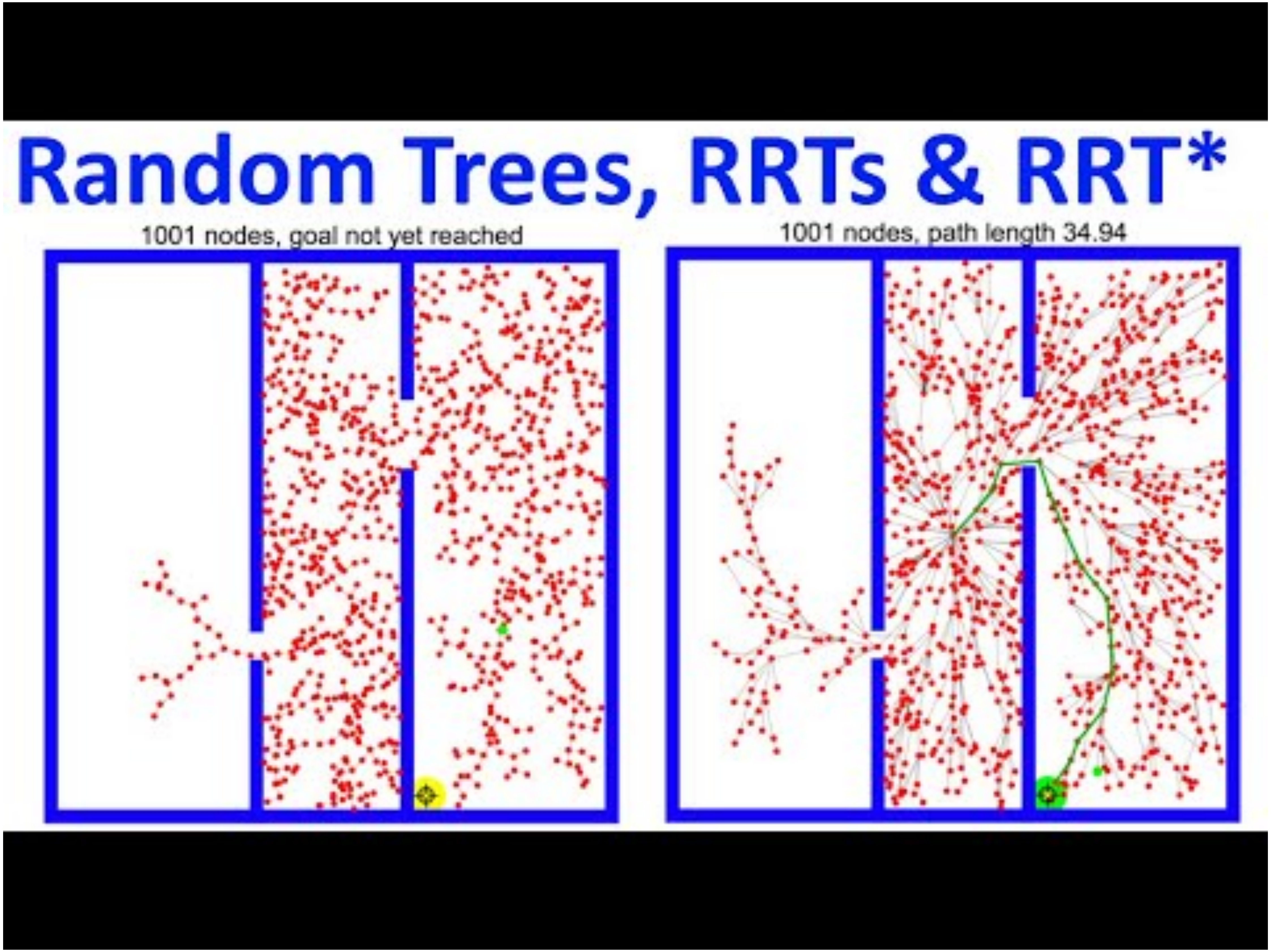
DARPA Urban Challenge

The DARPA Urban Challenge was held on November 3, 2007, at the former George AFB in Victorville, Calif. Building on the success of the 2004 and 2005 Grand Challenges, this event required teams to build an autonomous vehicle capable of driving in traffic, performing complex maneuvers such as merging, passing, parking, and negotiating intersections. As the day wore on, it became apparent to all that this race was going to have finishers. At 1:43 pm, “Boss”, the entry of the Carnegie Mellon Team, Tartan Racing, crossed the finish line first with a run time of just over four hours. Nineteen minutes later, Stanford University’s entry, “Junior,” crossed the finish line. It was a scene that would be repeated four more times as six robotic vehicles eventually crossed the finish line, an astounding feat for the teams and proving to the world that autonomous urban driving could become a reality. This event was groundbreaking as the first time autonomous vehicles have interacted with both manned and unmanned vehicle traffic in an urban environment.

overview of Darpa challenge, 3 min https://www.youtube.com/watch?v=Uqt_pRbR8rI&list=PLAwxTw4SYaPkCSYXw6-a_aAoXVKLDwnHK

DARPA 2007, MIT team

- uses RRT (starting at minute 6)
  - talk by Sertac Karaman in Darpa 2007 MIT team: https://www.youtube.com/watch?v=0fLSf3NO0-s



Random Trees, RRTs & RRT*

1001 nodes, goal not yet reached

1001 nodes, path length 34.94

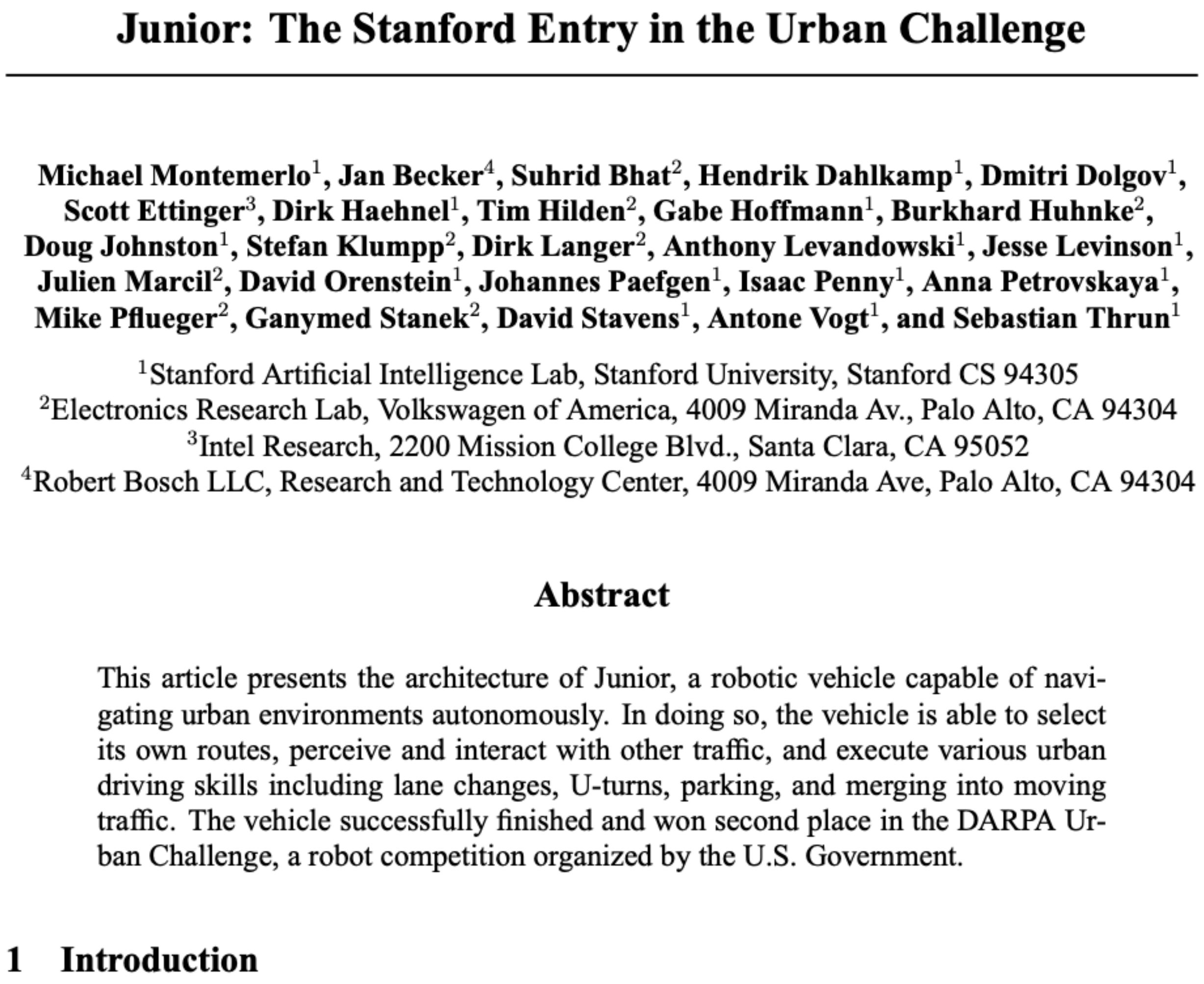
DARPA 2007, Stanford team

- uses hybrid A*

- Stanford's A*-based planner in action

https://www.youtube.com/watch?v=qXZt-B7iUyw

http://robots.stanford.edu/papers/junior08.pdf



Junior: The Stanford Entry in the Urban Challenge

Michael Montemerlo[1], Jan Becker[4], Suhrid Bhat[2], Hendrik Dahlkamp[1], Dmitri Dolgov[1], Scott Ettinger[3], Dirk Haehnel[1], Tim Hilden[2], Gabe Hoffmann[1], Burkhard Huhnke[2], Doug Johnston[1], Stefan Klumpp[2], Dirk Langer[2], Anthony Levandowski[1], Jesse Levinson[1], Julien Marcil[2], David Orenstein[1], Johannes Paefgen[1], Isaac Penny[1], Anna Petrovskaya[1], Mike Pflueger[2], Ganymed Stanek[2], David Stavens[1], Antone Vogt[1], and Sebastian Thrun[1]

[1]Stanford Artificial Intelligence Lab, Stanford University, Stanford CS 94305
[2]Electronics Research Lab, Volkswagen of America, 4009 Miranda Av., Palo Alto, CA 94304
[3]Intel Research, 2200 Mission College Blvd., Santa Clara, CA 95052
[4]Robert Bosch LLC, Research and Technology Center, 4009 Miranda Ave, Palo Alto, CA 94304

Abstract

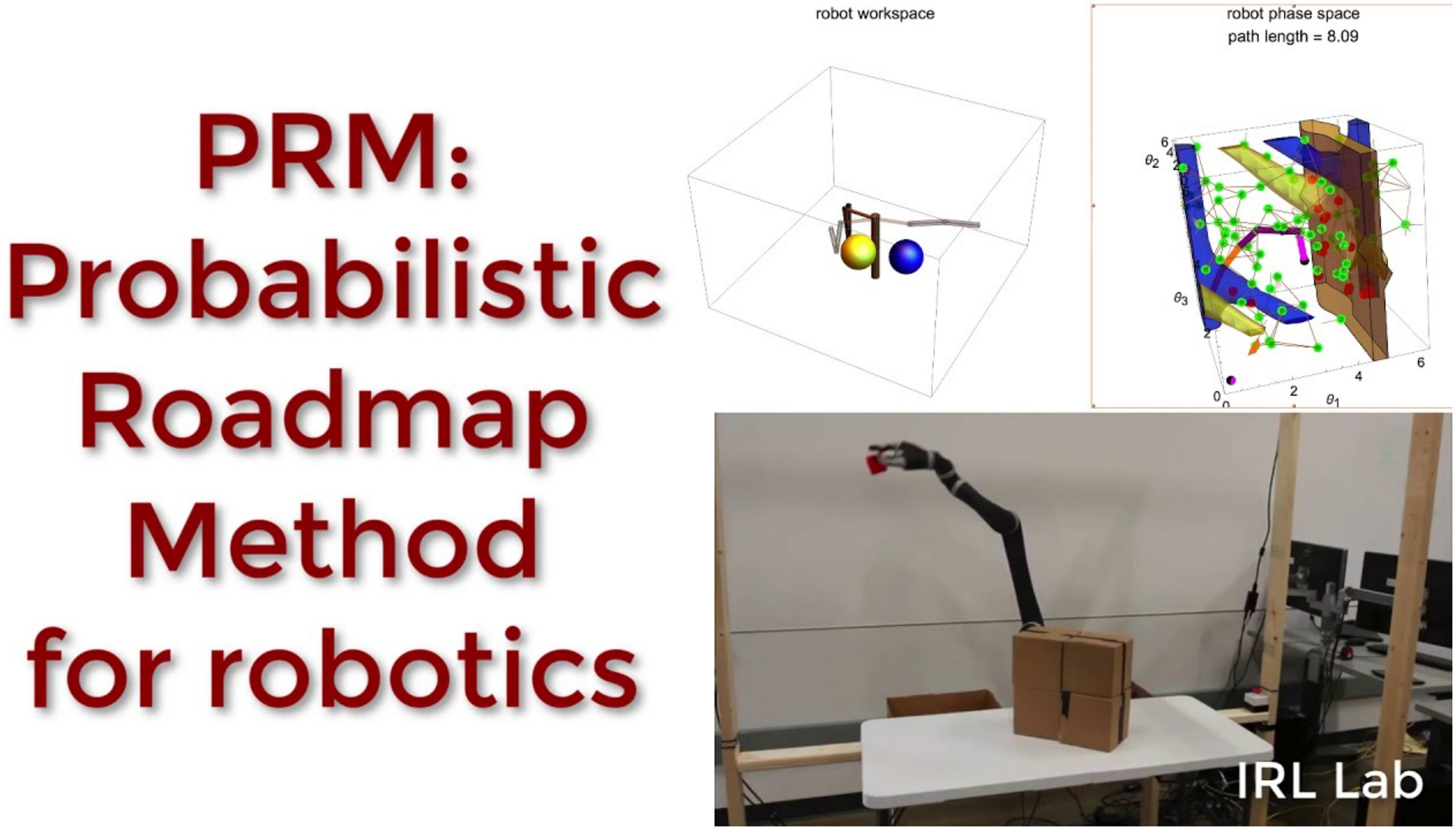
This article presents the architecture of Junior, a robotic vehicle capable of navigating urban environments autonomously. In doing so, the vehicle is able to select its own routes, perceive and interact with other traffic, and execute various urban driving skills including lane changes, U-turns, parking, and merging into moving traffic. The vehicle successfully finished and won second place in the DARPA Urban Challenge, a robot competition organized by the U.S. Government.

1 Introduction

Self-driving cars

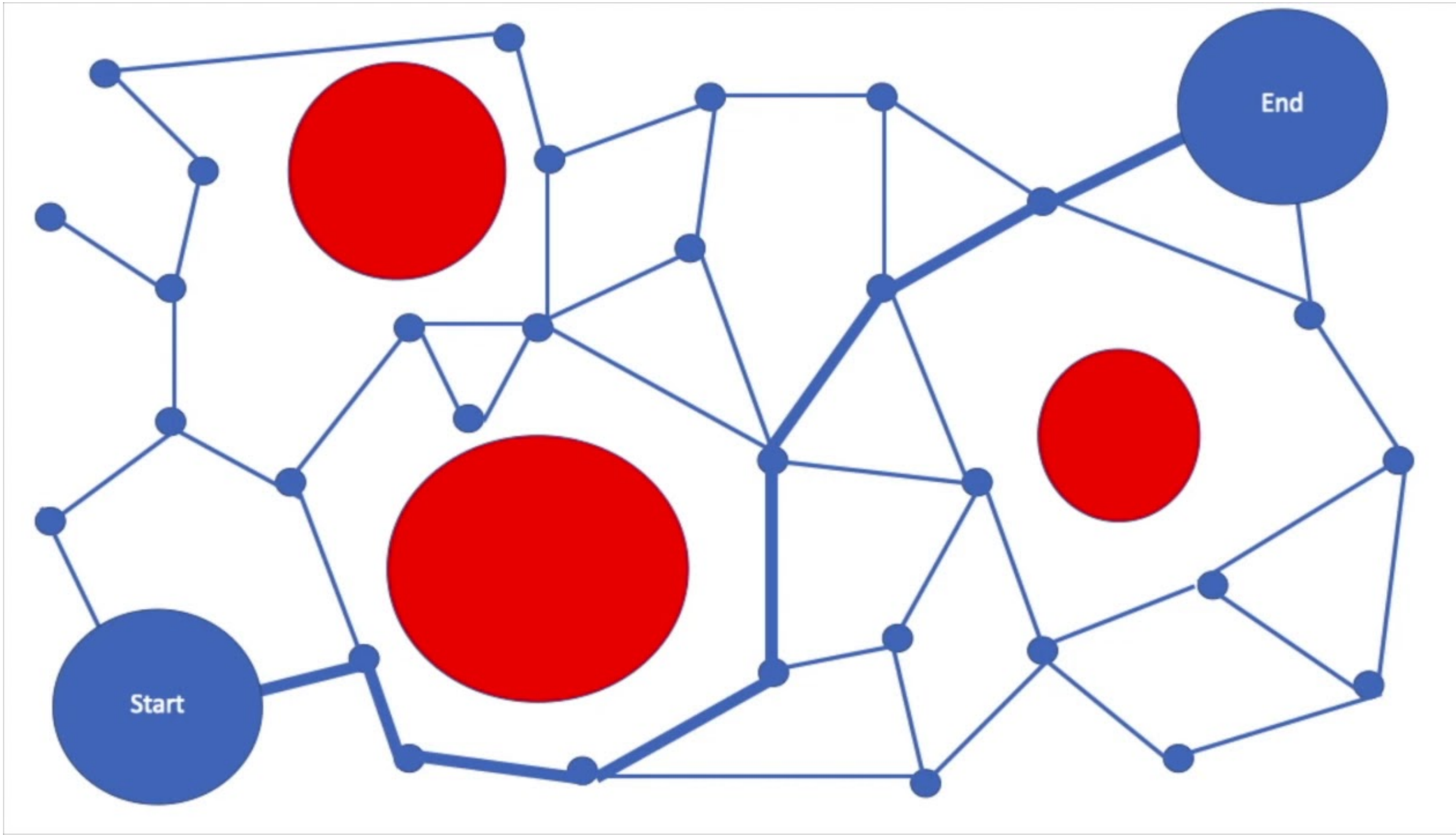
- Both graph search and incremental tree-based
- DARPA urban challenge:
  - **CMU**: lattice graph in 4D (x,y, orientation, velocity); graph search with D*
  - **Stanford**: incremental sparse tree of possible maneuvers, hybrid A*
  - **Virginia Tech**: graph discretization of possible maneuvers, search with A*
  - **MIT**: variant of RRT with biased sampling

- Good read: *A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles, by Brian Paden, Michal Cáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli* https://arxiv.org/pdf/1604.07446.pdf https://www.youtube.com/watch?v=tlFVbHENPCI

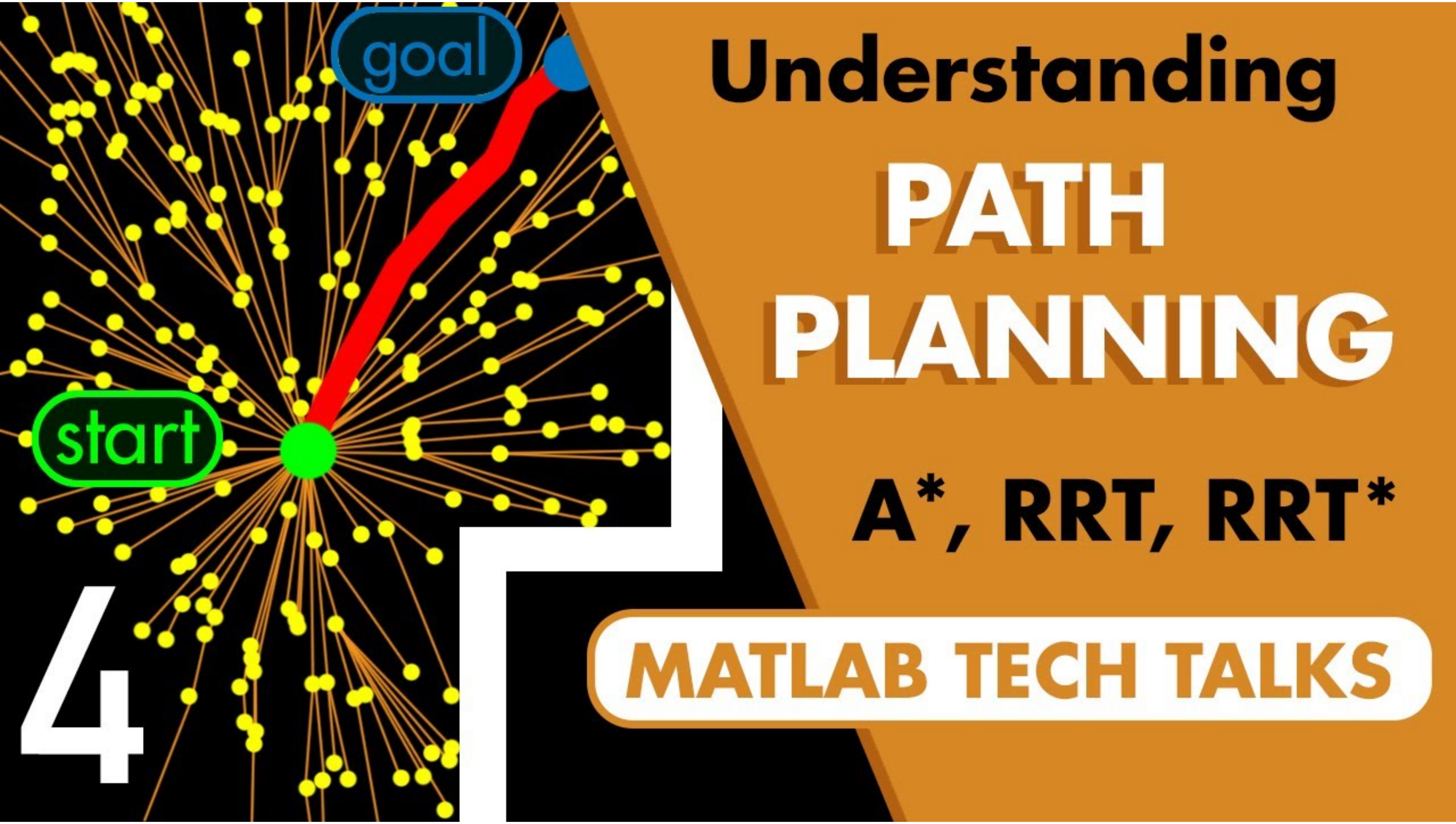

PRM:
Probabilistic
Roadmap
Method
for robotics
robot workspace
robot phase space
path length = 8.09
θ_1
θ_2
θ_3
IRL Lab

Comparison of RRT, PRM (MIT course project)

https://www.youtube.com/watch?v=gP6MRe_IHFo



https://www.youtube.com/watch?v=QR3U1dgc5RE


goal
start
4
Understanding
PATH
PLANNING
A*, RRT, RRT*
MATLAB TECH TALKS

Potential field methods

- Idea [Latombe et al, 1992]
  - Define a potential field
  - Robot moves in the direction of steepest descent on potential function
- Ideally potential function has global minimum at the goal, has no local minima, and is very large around obstacles
- Algorithm outline:
  - place a regular grid over C-space
  - search over the grid with potential function as heuristic

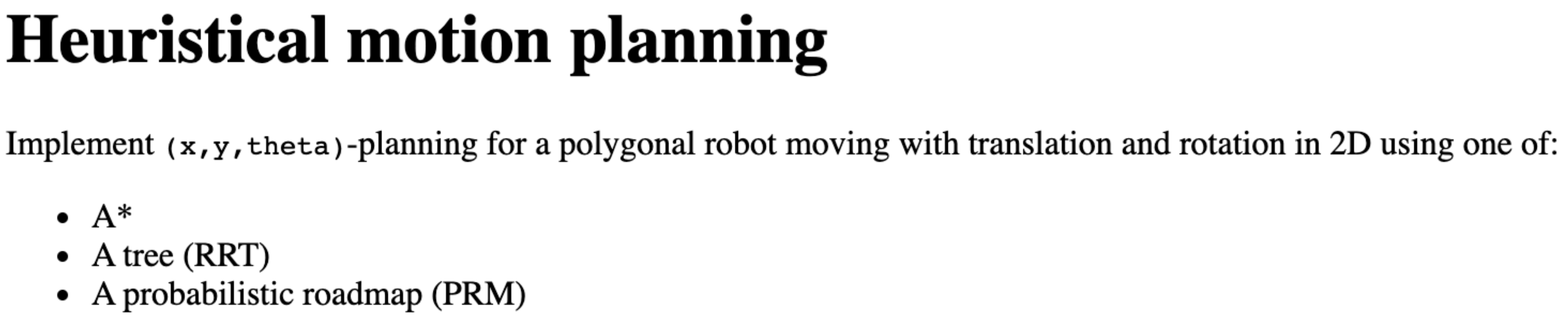
https://www.youtube.com/watch?v=r9FD7P76zJs

Potential field methods

- Pro:
  - Framework can be adapted to any specific scene
- Con:
  - can get stuck in local minima
  - Potential functions that are minima-free are known, but expensive to compute

- Example: RPP (Randomized path planner) is based on potential functions
  - Escapes local minima by executing random walks
  - Succesfully used to
    - performs riveting ops on plane fuselages
    - plan disassembly operations for maintenance of aircraft engines

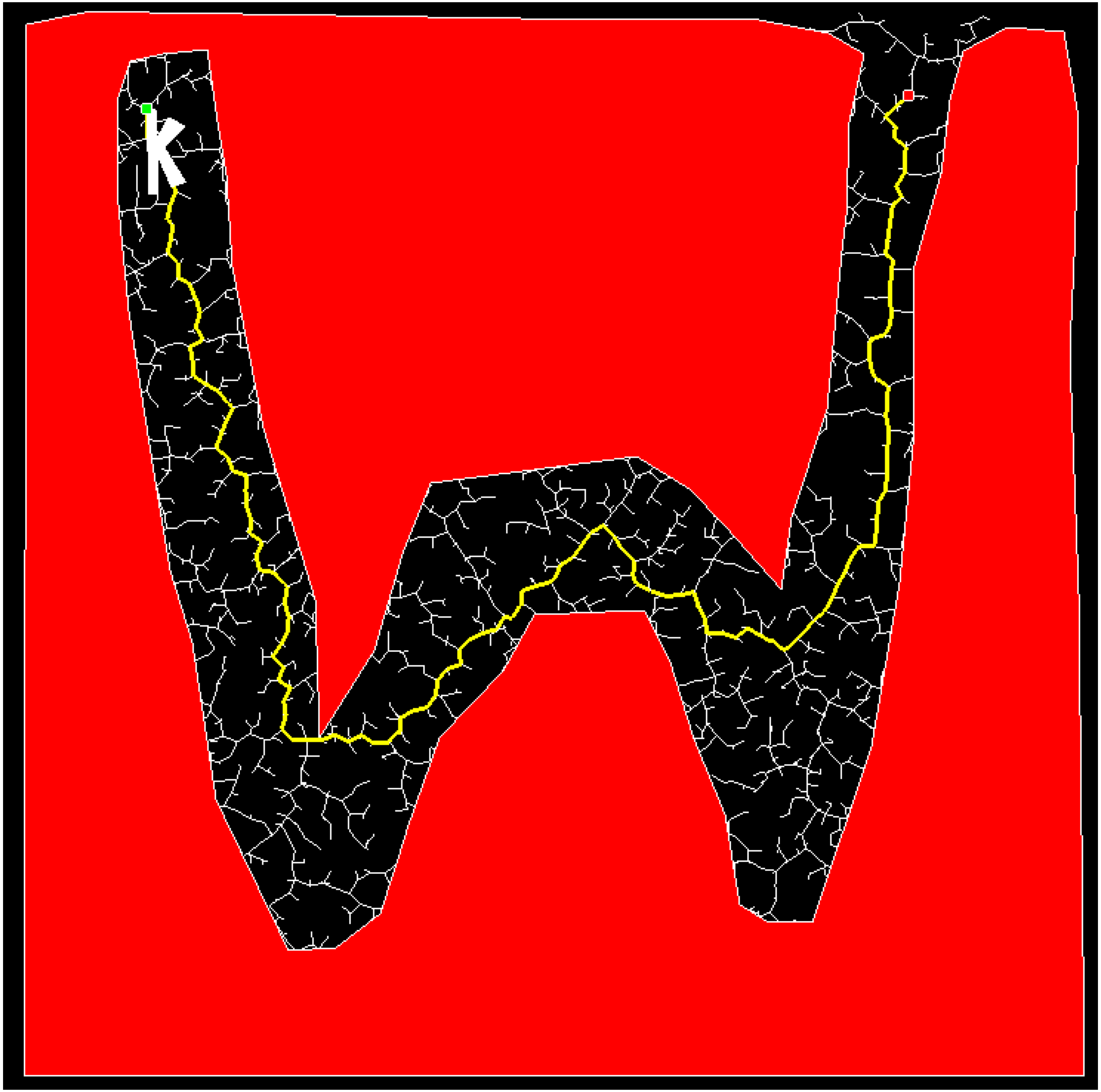
Project 7

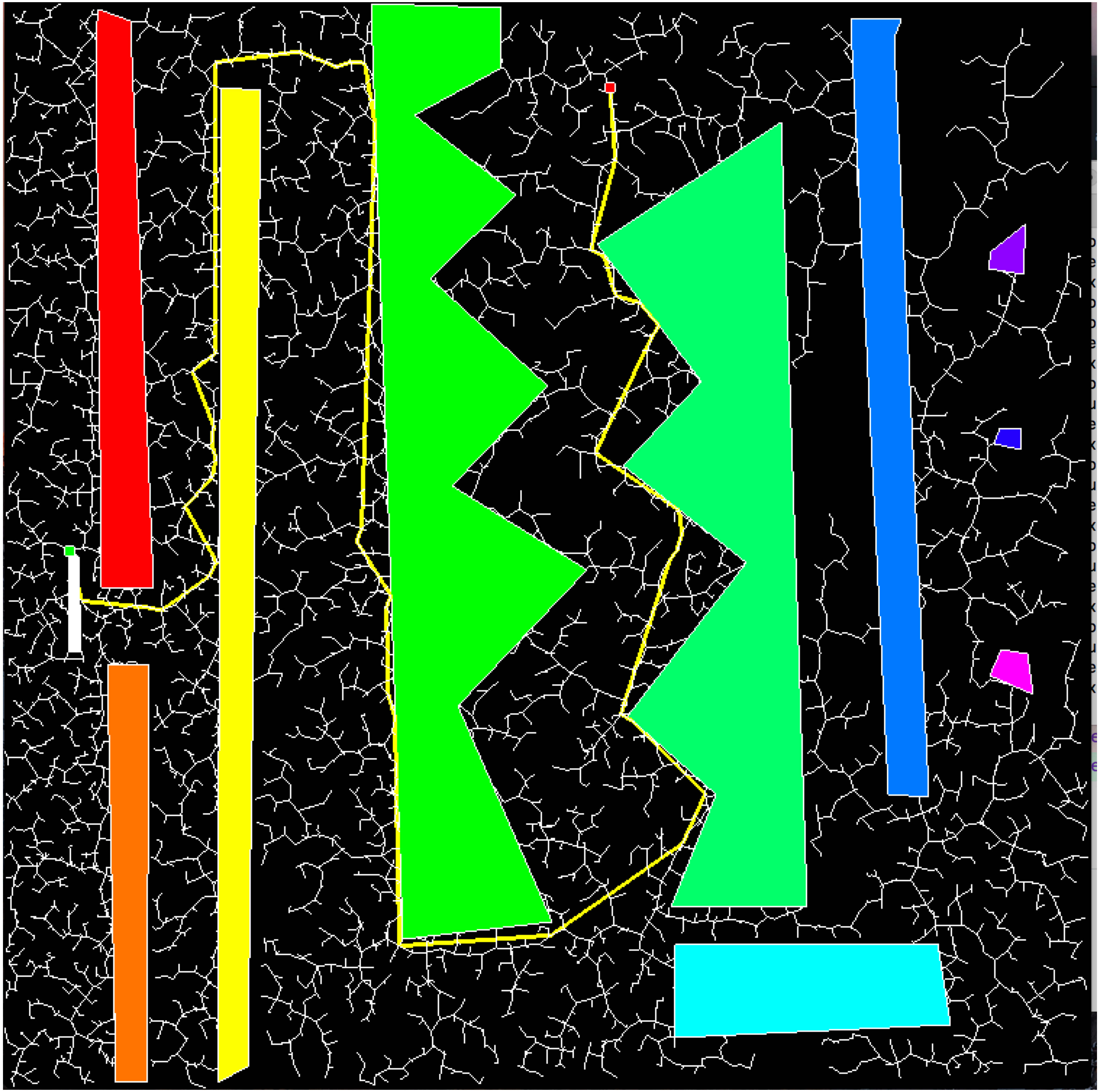


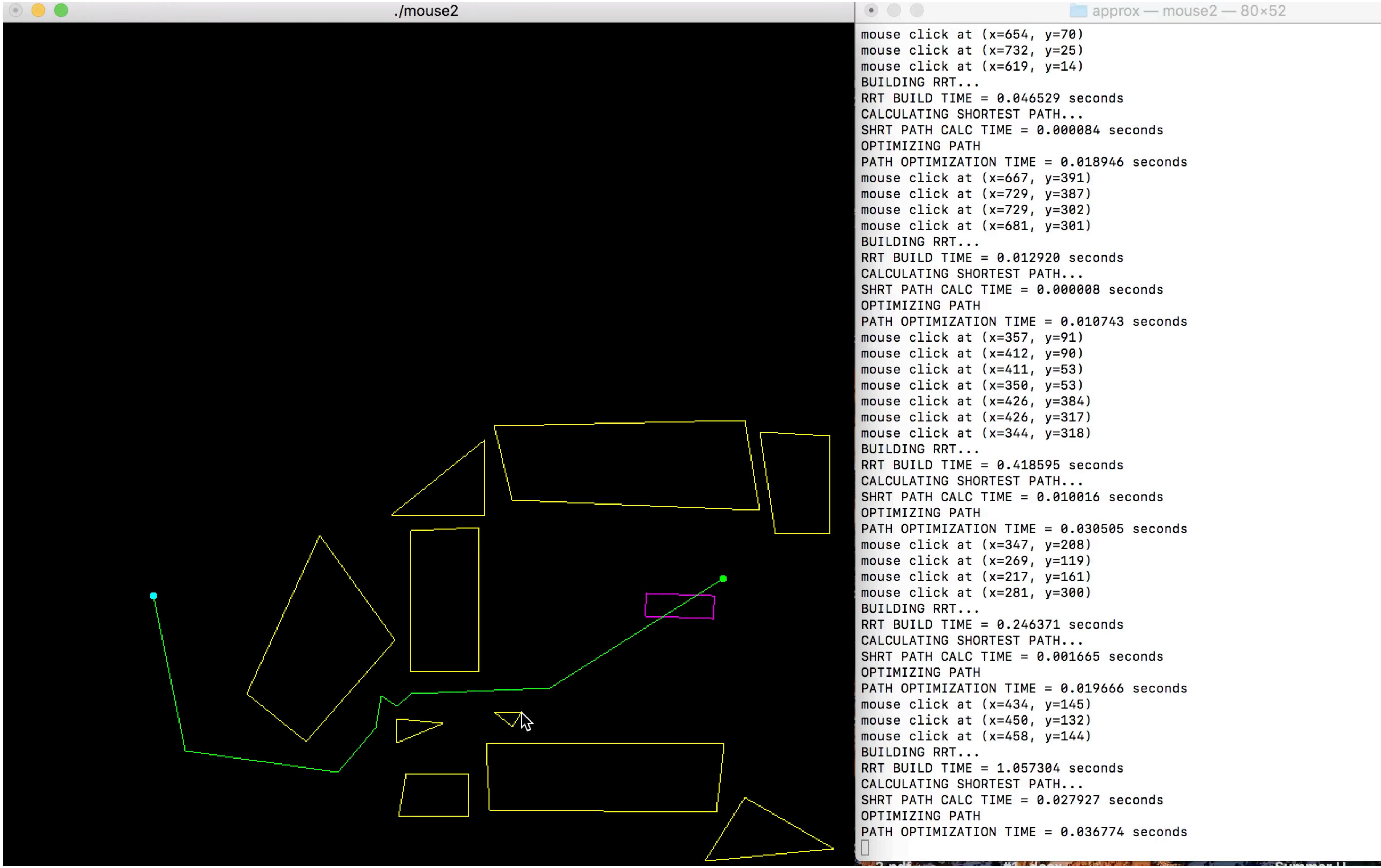
Heuristical motion planning

Implement `(x,y,theta)`-planning for a polygonal robot moving with translation and rotation in 2D using one of:

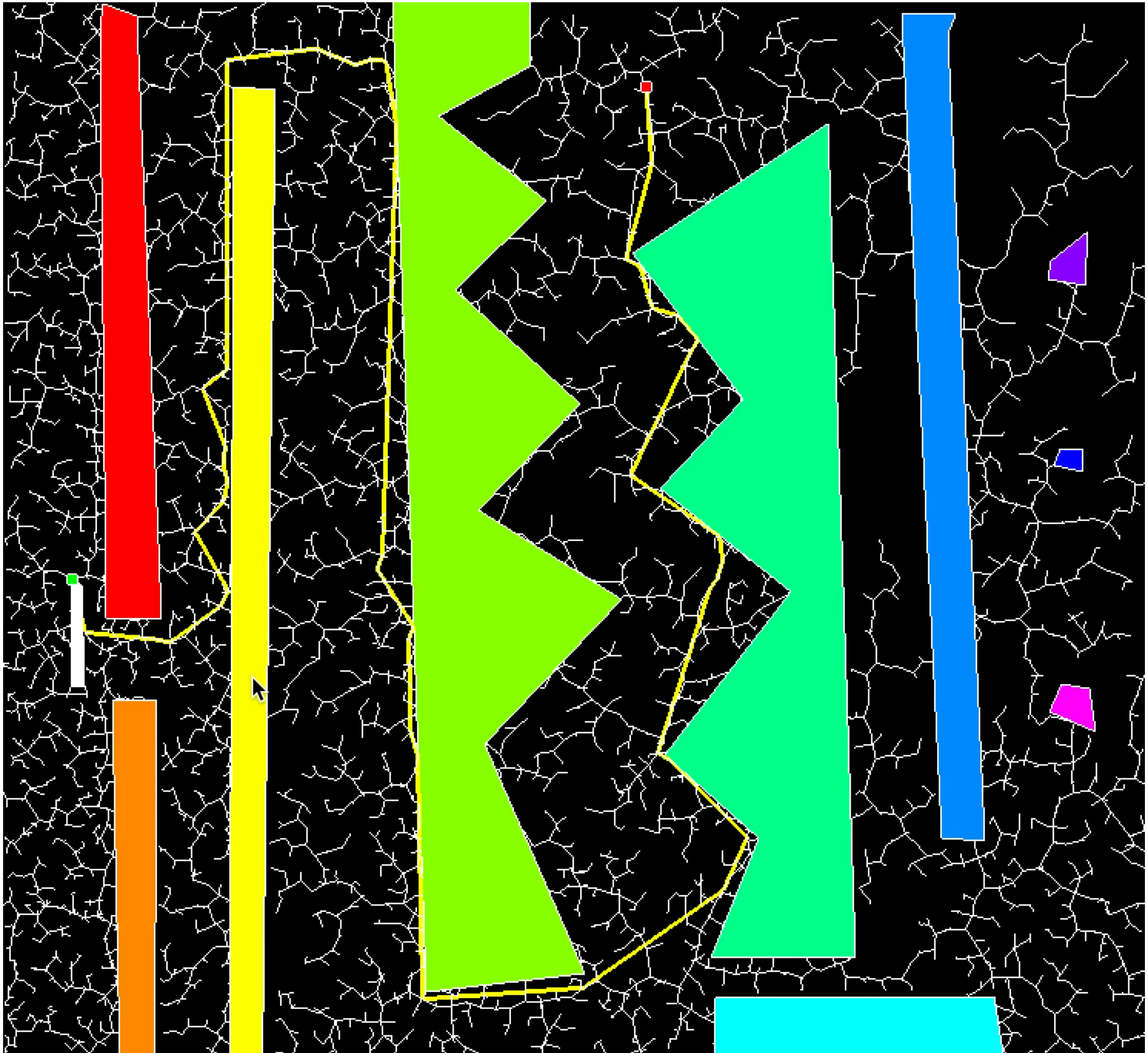
- A*
- A tree (RRT)
- A probabilistic roadmap (PRM)

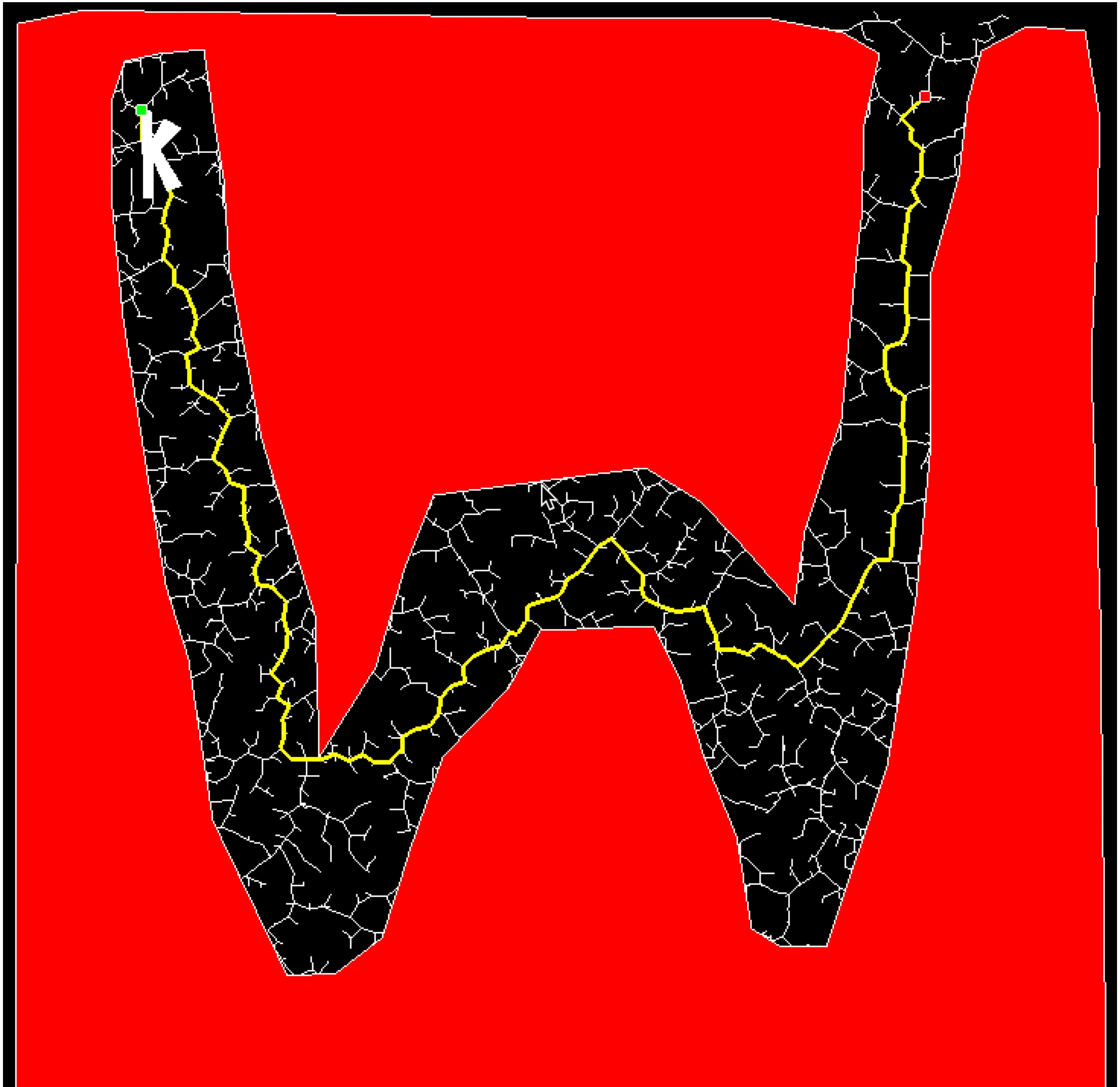

K





```
mouse click at (x=654, y=70)
mouse click at (x=732, y=25)
mouse click at (x=619, y=14)
BUILDING RRT...
RRT BUILD TIME = 0.046529 seconds
CALCULATING SHORTEST PATH...
SHRT PATH CALC TIME = 0.000084 seconds
OPTIMIZING PATH
PATH OPTIMIZATION TIME = 0.018946 seconds
mouse click at (x=667, y=391)
mouse click at (x=729, y=387)
mouse click at (x=729, y=302)
mouse click at (x=681, y=301)
BUILDING RRT...
RRT BUILD TIME = 0.012920 seconds
CALCULATING SHORTEST PATH...
SHRT PATH CALC TIME = 0.000008 seconds
OPTIMIZING PATH
PATH OPTIMIZATION TIME = 0.010743 seconds
mouse click at (x=357, y=91)
mouse click at (x=412, y=90)
mouse click at (x=411, y=53)
mouse click at (x=350, y=53)
mouse click at (x=426, y=384)
mouse click at (x=426, y=317)
mouse click at (x=344, y=318)
BUILDING RRT...
RRT BUILD TIME = 0.418595 seconds
CALCULATING SHORTEST PATH...
SHRT PATH CALC TIME = 0.010016 seconds
OPTIMIZING PATH
PATH OPTIMIZATION TIME = 0.030505 seconds
mouse click at (x=347, y=208)
mouse click at (x=269, y=119)
mouse click at (x=217, y=161)
mouse click at (x=281, y=300)
BUILDING RRT...
RRT BUILD TIME = 0.246371 seconds
CALCULATING SHORTEST PATH...
SHRT PATH CALC TIME = 0.001665 seconds
OPTIMIZING PATH
PATH OPTIMIZATION TIME = 0.019666 seconds
mouse click at (x=434, y=145)
mouse click at (x=450, y=132)
mouse click at (x=458, y=144)
BUILDING RRT...
RRT BUILD TIME = 1.057304 seconds
CALCULATING SHORTEST PATH...
SHRT PATH CALC TIME = 0.027927 seconds
OPTIMIZING PATH
PATH OPTIMIZATION TIME = 0.036774 seconds
```




K