15-780: Graduate AI

Lecture 2. A, Spatial Search*

Geoff Gordon (this lecture)
Ziv Bar-Joseph
TAs Geoff Hollinger, Henry Lin

Admin

- *Slides on web site*
- *Matlab tutorial next Tue (5-6 NSH 1507)*
- *Please send your email address to TA Henry Lin (thlin at cs), who is compiling a class email list*
- *Please check the website regularly for readings (for Lec. 1–2, Ch. 1–4 of RN)*

Review

Topics covered

- *What is AI? (Be able to discuss an example or two)*
- *Types of uncertainty & corresponding approaches*
- *How to set up state space graph for problems like the robotic grad student or path planning*

Topics covered

- *Generic search algorithm & data structures*
- *Search methods: be able to simulate*
  - *BFS, DFS, DFID*
  - *Heuristic search*
- *What are advantages of each?*

Projects

Project ideas

- *Plan a path for this robot so that it gets a good view of an object as fast as possible*

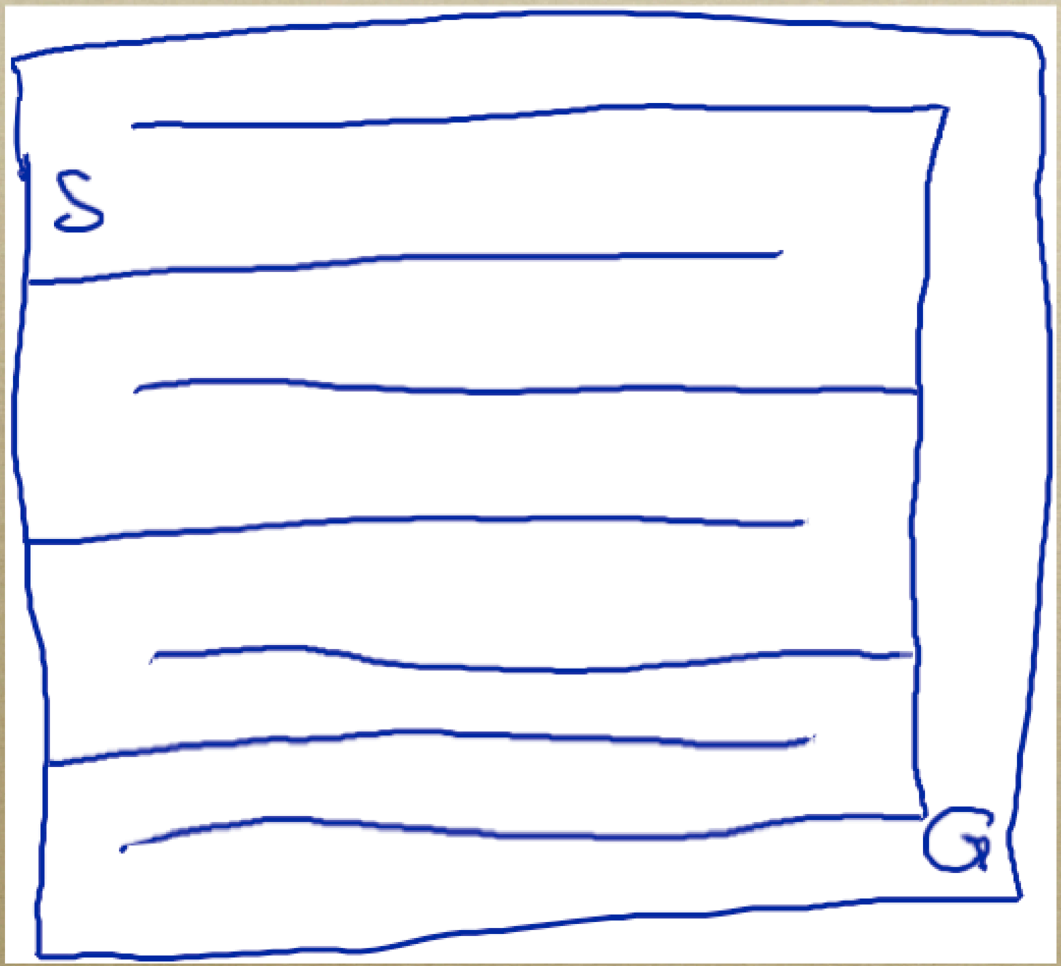
Project ideas

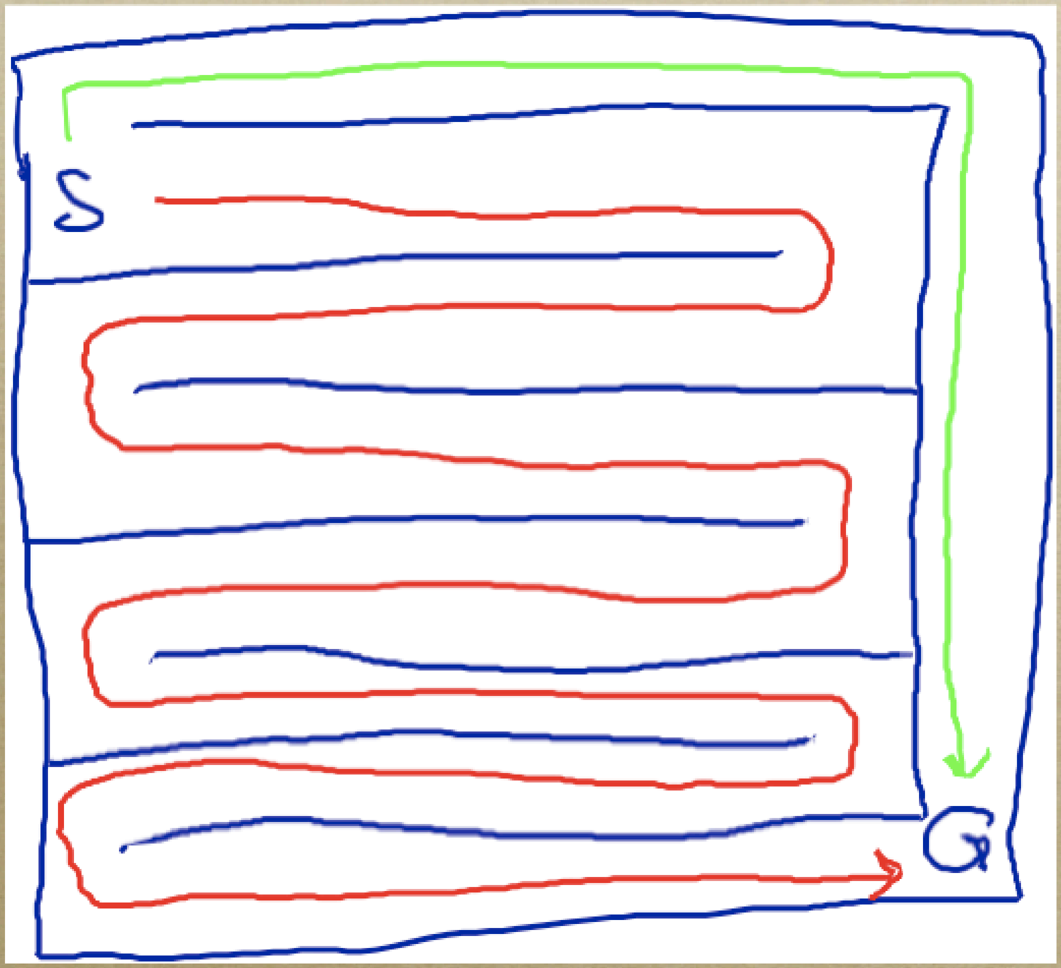
- *Do something cool w/ Lego Mindstorms*
  - *plan footstep placements*
  - *plan how to grip objects*

A* Search

Heuristic search looking bad



Heuristic search looking bad



Generic search

$S = \{\ start\ \}\quad M = \emptyset$

$While\ (S \neq \emptyset)$

$\quad x \leftarrow some\ element\ of\ S,\ \ S \leftarrow S \setminus x$

$\quad CheckSolution(x)$

$\quad For\ y \in neighbors(x) \setminus M$

$\qquad S \leftarrow S \cup \{y\}$

$\quad M = M \cup \{x\}$

A* search: Open list

- *Implement S with priority queue*
  - *S.insert(x, P)*
  - *S.pop()*
  - *and maybe S.test_member(x)*
- *Like heuristic search*
  - *but priority calculated differently (more below)*

A* search: Path costs

- *For both priority and closed list, maintain path cost function $g(x)$*
- *$g(x)$ = best cost to reach x so far*
  - *or ∞ if no path from start to x found yet*
- *When pushing y, set $g(y) = g(x) + c(x,y)$*
  - *if $g(y)$ finite, smaller of old and new values*

A* search: Closed list

- *Implementation of M: use g(x) and S*
- *If g(x) finite, x must be either open or closed*
- *So,* $M = \{\, g(x) \text{ finite} \wedge x \notin S \,\}$
- *This is where we'd use S.test_member(), but it will turn out we can be slightly smarter*

A* search: Priority

- *When calling S.insert(x, P)*
- *Set* $P = f(x) \equiv g(x) + h(x)$
- $h(x)$ *= heuristic estimate of distance from x to goal (just like in heuristic search)*
- $f(x)$ *= estimate of cost of path through x*
- *Idea: focus on nodes that might yield short paths*

Generic search

$S = \{ \textit{ start } \} \quad M = \varnothing$

$\textit{While } (S \neq \varnothing)$

$\quad x \leftarrow \textit{some element of } S, \;\; S \leftarrow S \setminus x$

$\quad \textit{CheckSolution}(x)$

$\quad \textit{For } y \in \textit{neighbors}(x) \setminus M$

$\qquad S \leftarrow S \cup \{y\}$

$\quad M = M \cup \{x\}$

A* search

$S = \{ \text{start} \}$ $\boxed{g(x) = \infty \ (\forall x)}$ *Initialize M*

While $(S \neq \emptyset)$

Remove and return an element of x

$x \leftarrow \boxed{S.pop()}$

CheckSolution(x)

Update M

For $y \in$ *neighbors(x)*

M check

$\boxed{g(y) = \min(g(y), g(x) + c(x,y))}$

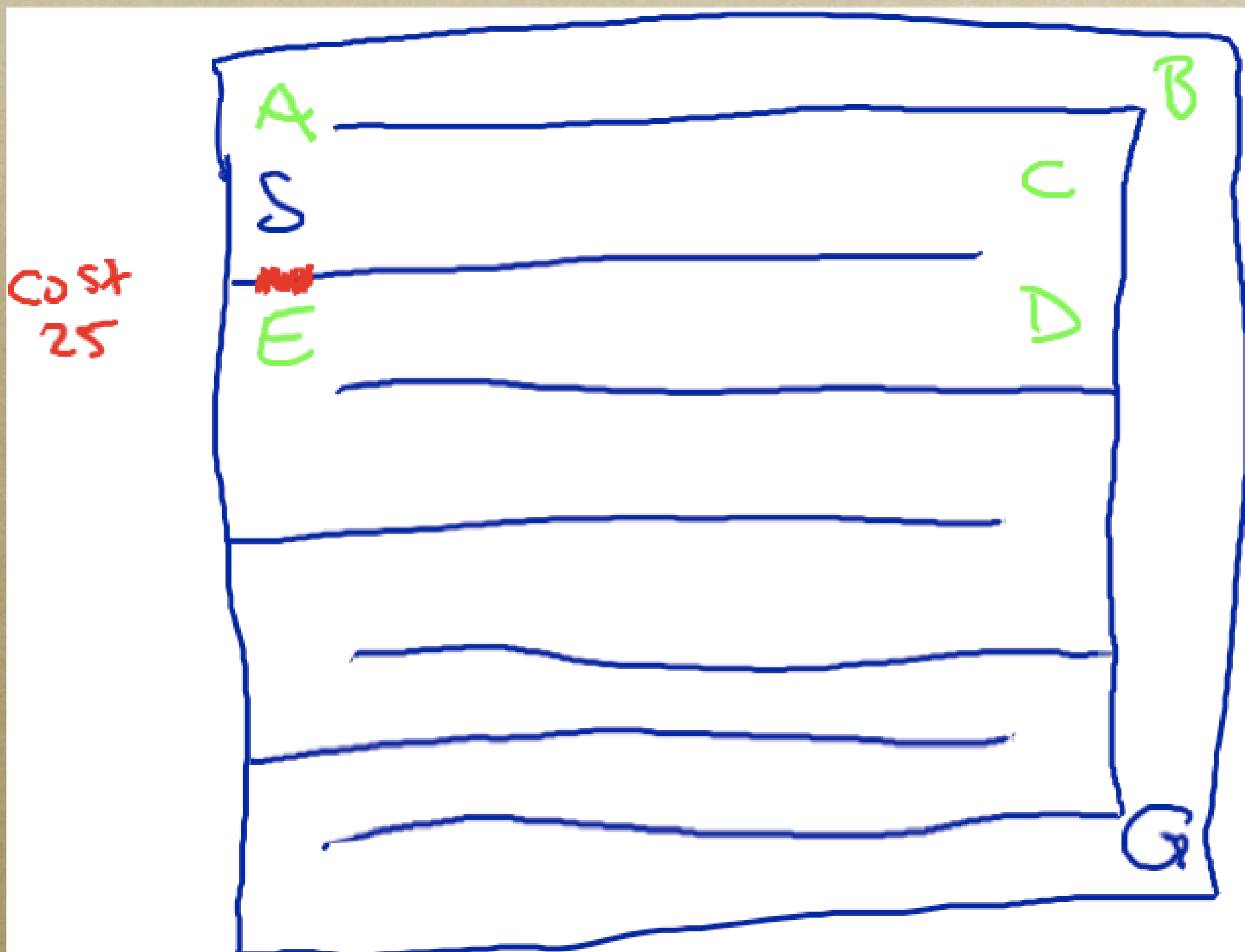
Add to S

$\boxed{\text{If } g(y) \text{ decreased,}}$ $\boxed{S.insert(y, g(y)+h(y))}$

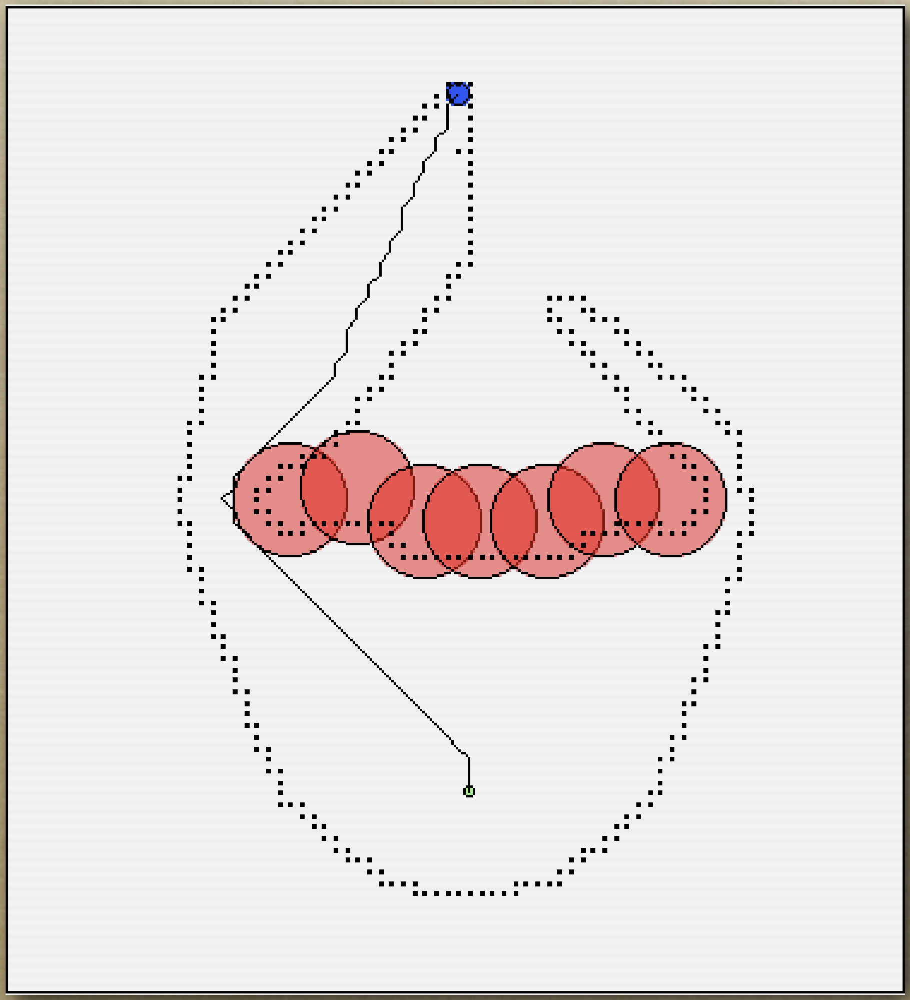
Admissible heuristic

- *A* has nice theoretical properties if h is* ***admissible***
- *That is,* $h(x) \leq$ *true distance from x to goal*
- *E.g., crow-flies distance in a maze*
- *Intuition: make a path look better, we examine it earlier, maybe waste some work. Make it look worse, we might miss it entirely, find a bad solution.*

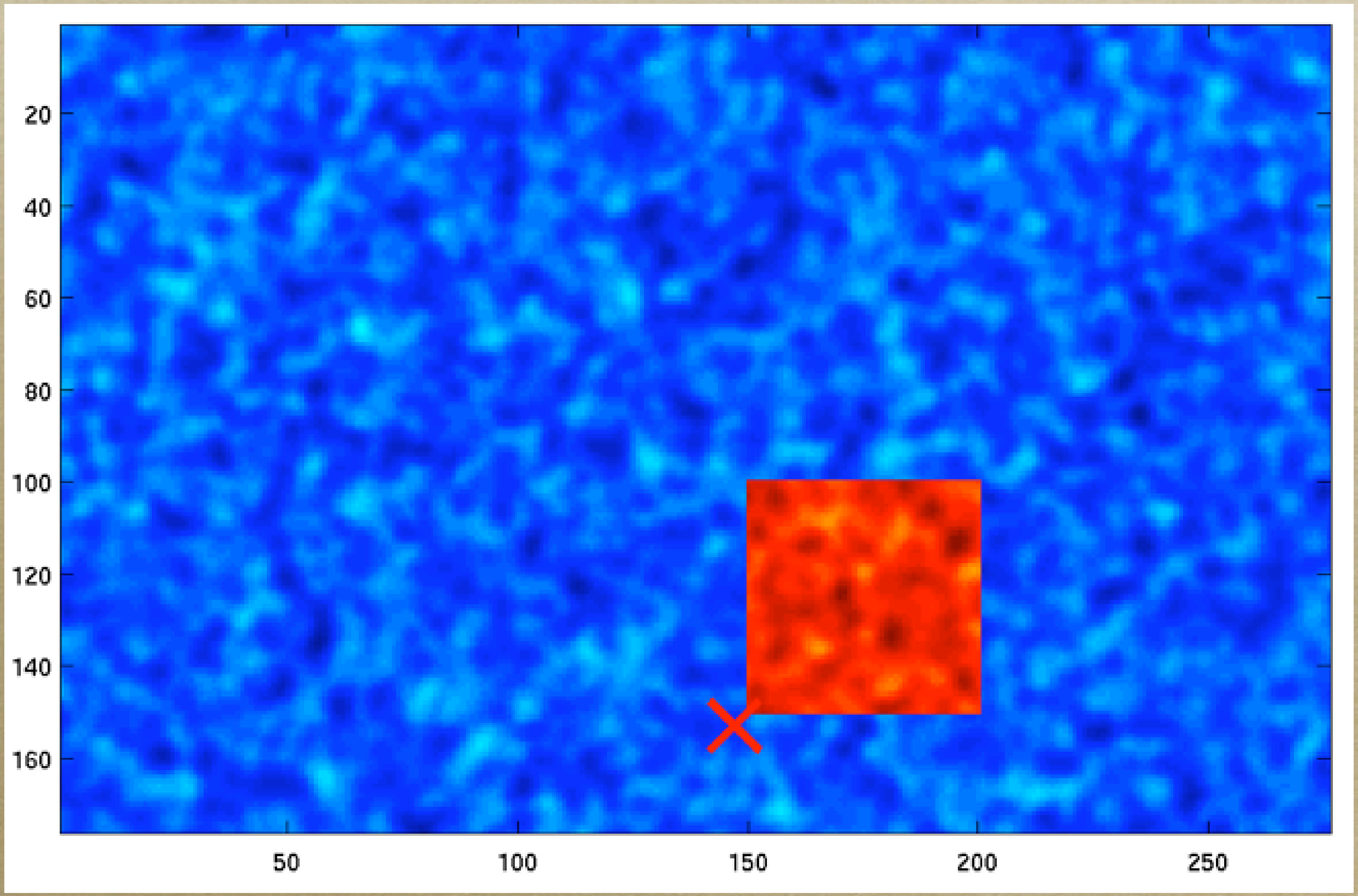
A* example



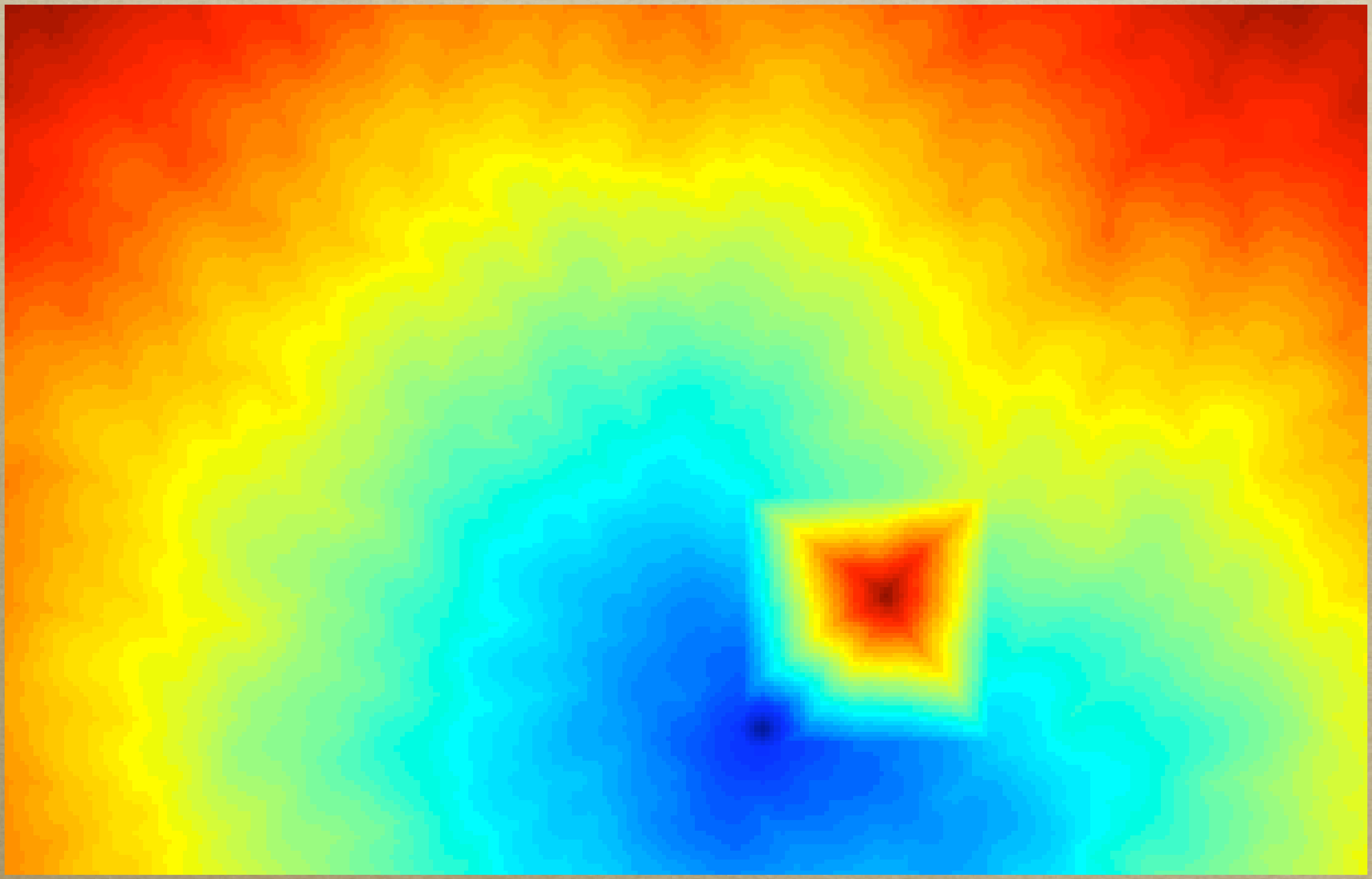
A* example



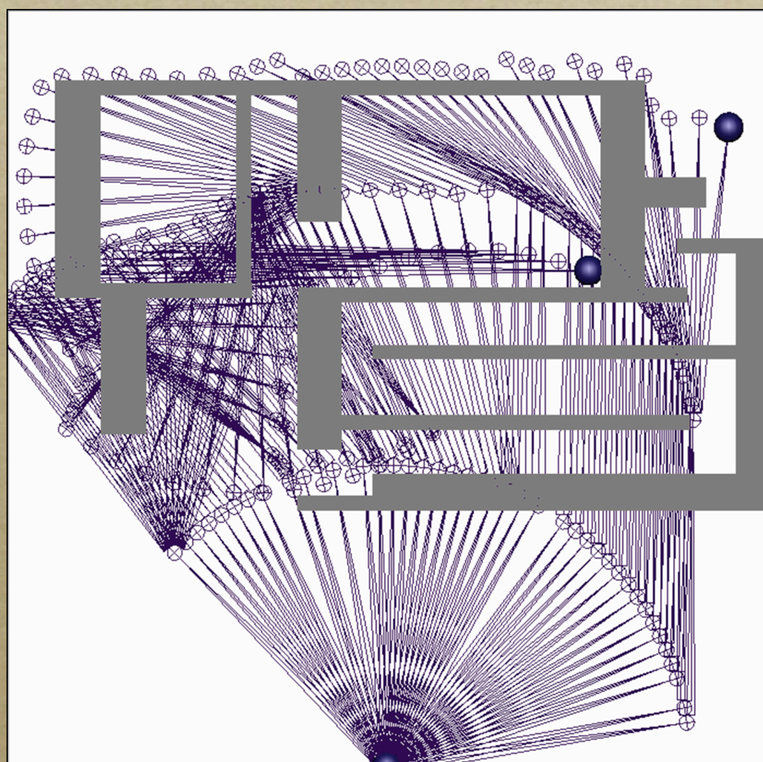
Node costs



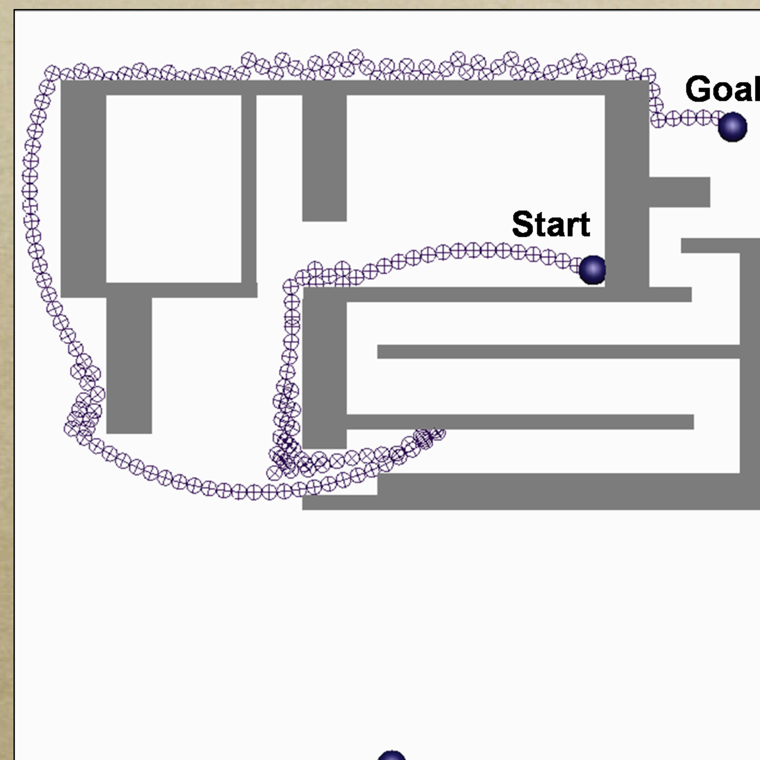
Path costs



More complicated A* example



Optimal Solution



End-effector Trajectory

A* guarantees

- *Write g* for depth of shallowest solution*
- *Assume h() is admissible*
- ***(optimality)** A* finds a solution of depth g**
- ***(efficiency)** A* expands no nodes that have f(node) > g**

A* proof

- *Both optimality and efficiency follow from:*

  Lemma. For any two nodes x and y which have $f(x) < f(y)$, A* expands x before y

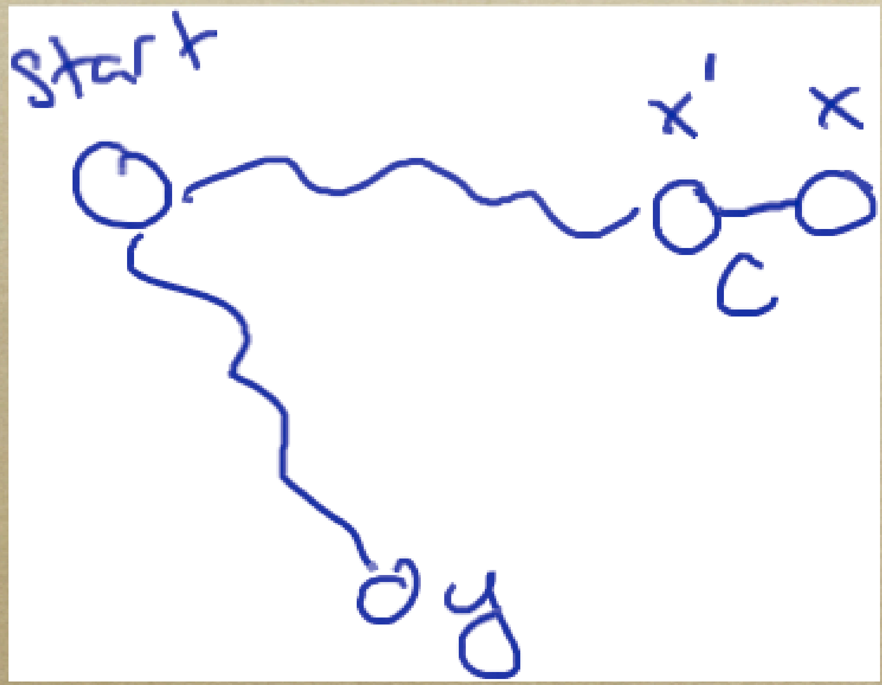
- *To see why optimality and efficiency follow, note goals have $f(x) = g(x)$*
  - *$h(x)$ must be 0*

A* proof

- *Will do a simple case: heuristic satisfies "triangle inequality"*
- *For all neighboring pairs (x, y)*

  $$h(x) \le h(y) + c(x, y)$$

Proof of lemma



- *Suppose f(y) > f(x) (so we want x first)*
- *Consider shortest path from start to x*

Proof cont'd

$$h(x') \le h(x) + c$$

$$\begin{aligned} f(x') &= g(x') + h(x') \\ &\le g(x') + h(x) + c \\ &= g(x) + h(x) \\ &= f(x) \end{aligned}$$

Proof cont'd

- *So, all nodes w on path to x have*

$$f(w) \leq f(x) < f(y)$$

- *At least one such w is always on queue while x has not been expanded (possibly we have w = x)*
- *So if x has not yet been expanded, we must pick w before we expand y — QED*

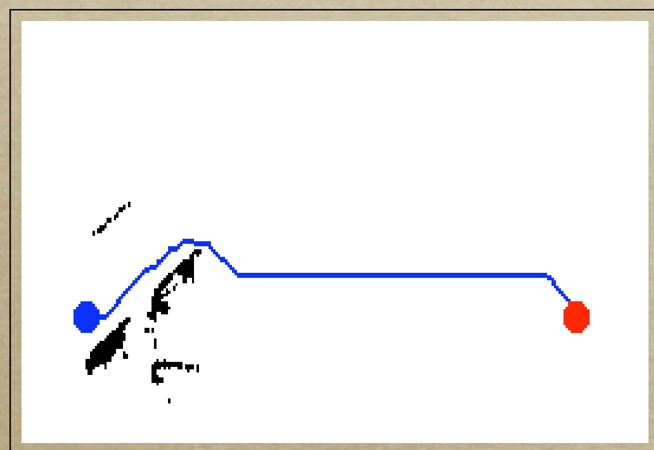
A* extensions

- *Suboptimal: use non-admissible heuristic, lose guarantees but maybe increase speed*
- *Iterative deepening: avoid priqueue*
- *Anytime: start with suboptimal solution, gradually improve it*
- *Dynamic: fast replan if map changes*

IDA*

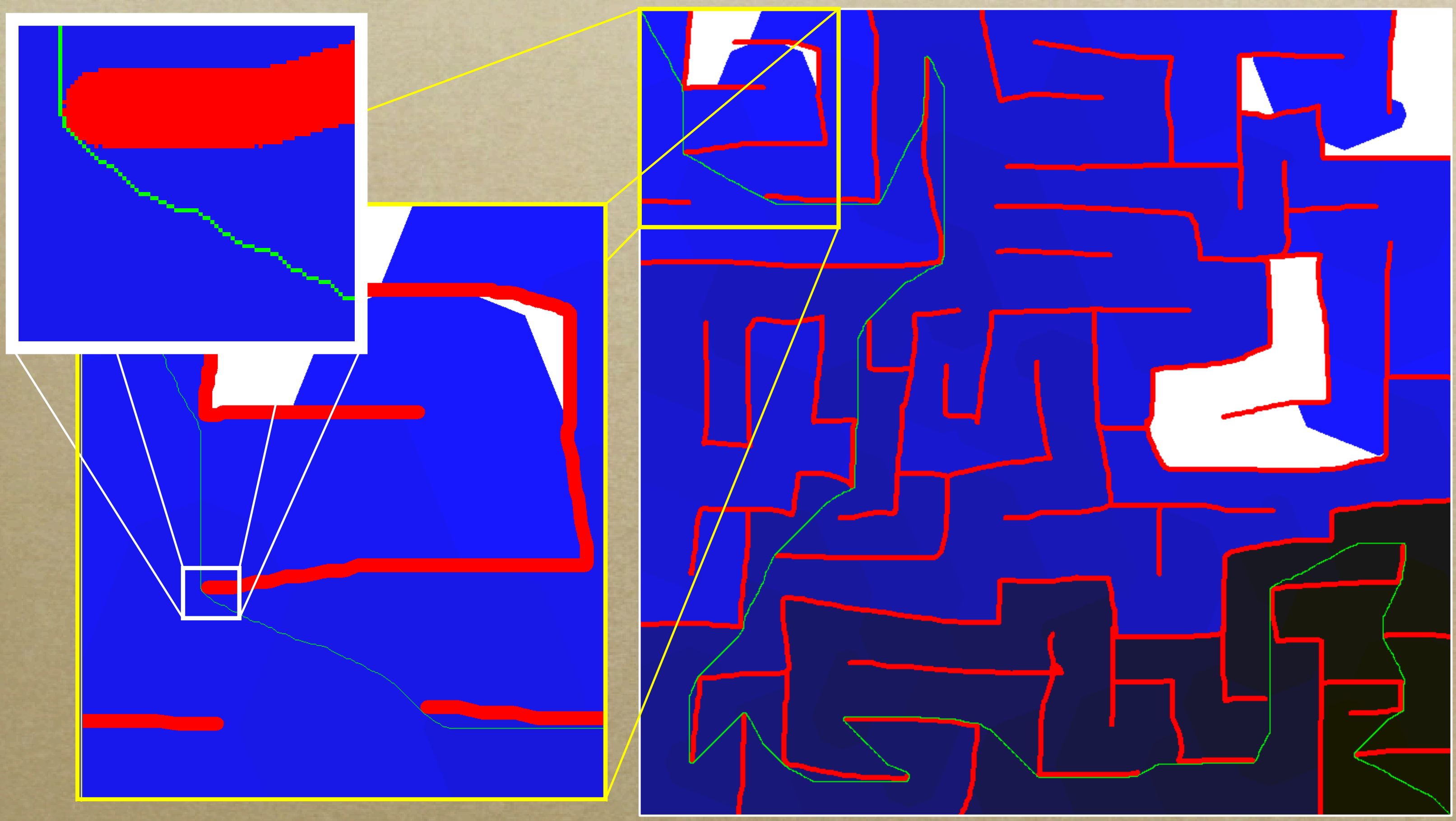
- *Do a DFS of all nodes with $f(node) < k$*
- *If no solution, increment k and try again*
- *Just like DFID, except that instead of a depth bound, bounds $f = g + h$*

Anytime, dynamic planning



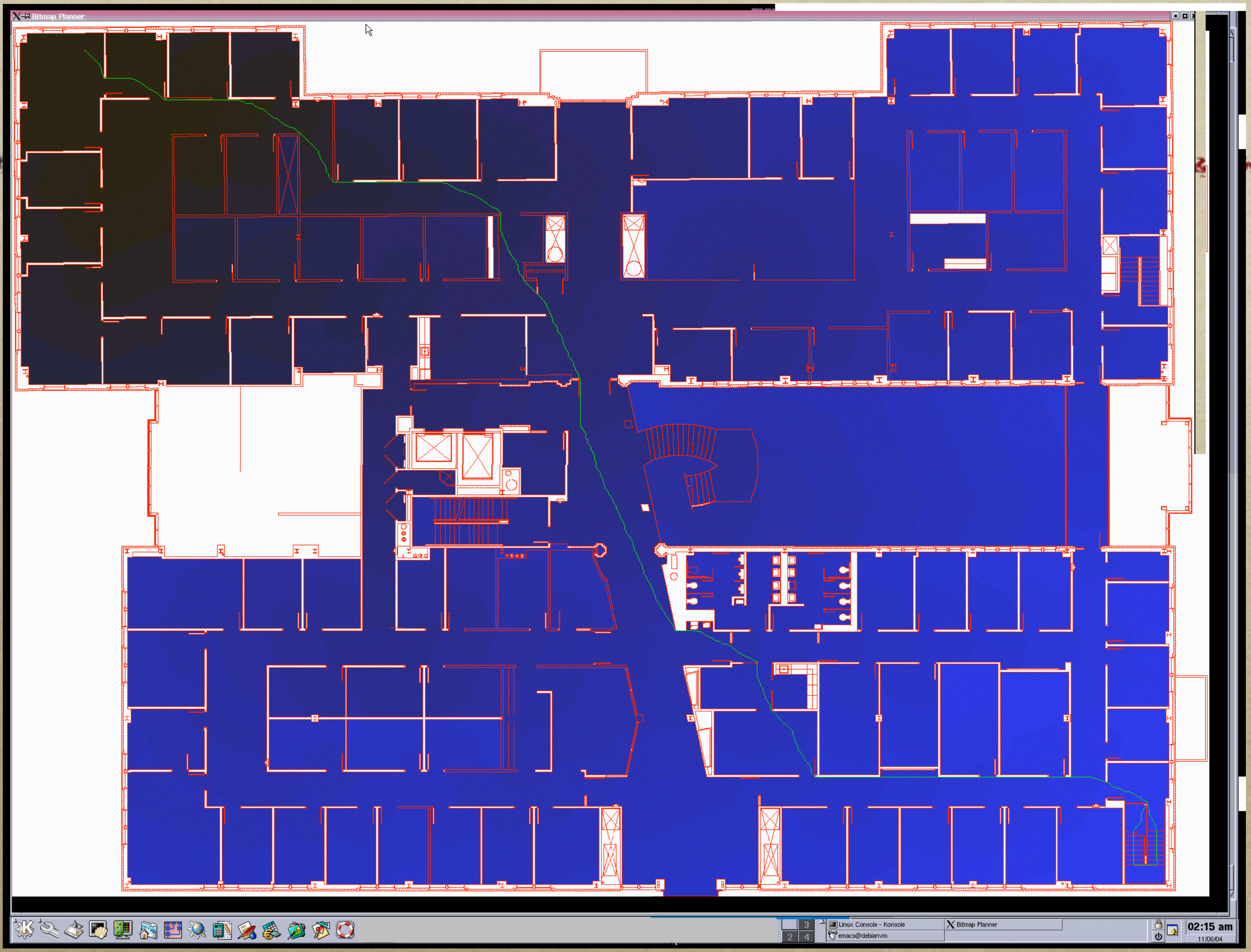

http://www.cs.cmu.edu/~ggordon/likhachev-etal.anytime-dstar.pdf

A* Planning on Big Grids

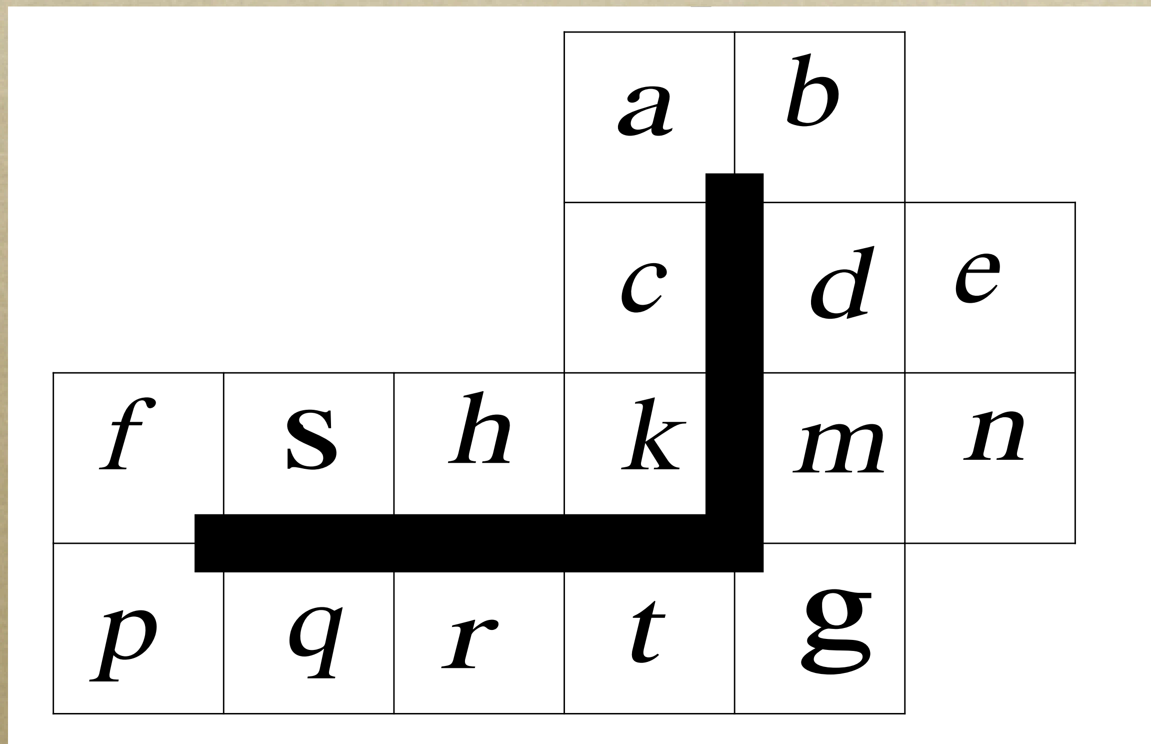


Credit: Kuffner

2D grids: 500,000 nodes = ~ 0.8 sec
10 million nodes = ~ 12 sec


Bitmap Planner
Linux Console - Konsole
emacs@debianvm
Bitmap Planner
02:15 am
11/06/04

Sample exercise



credit: Andrew Moore

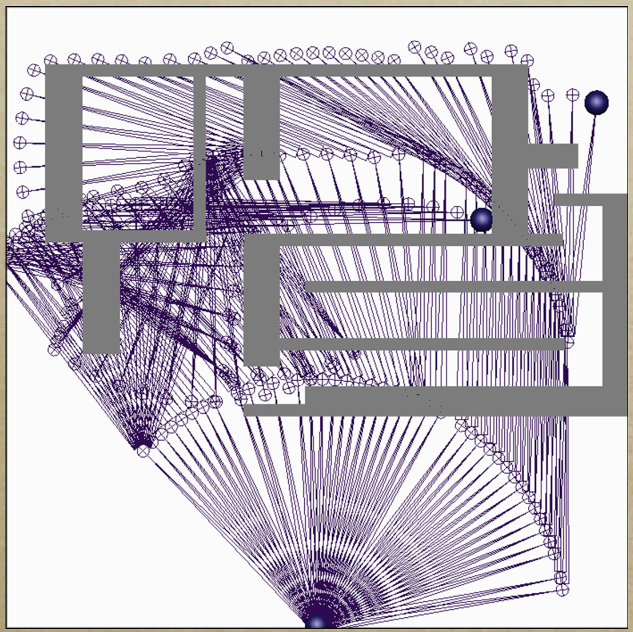
Nodes are connected in 4 cardinal directions, except across dark line

Sample exercise

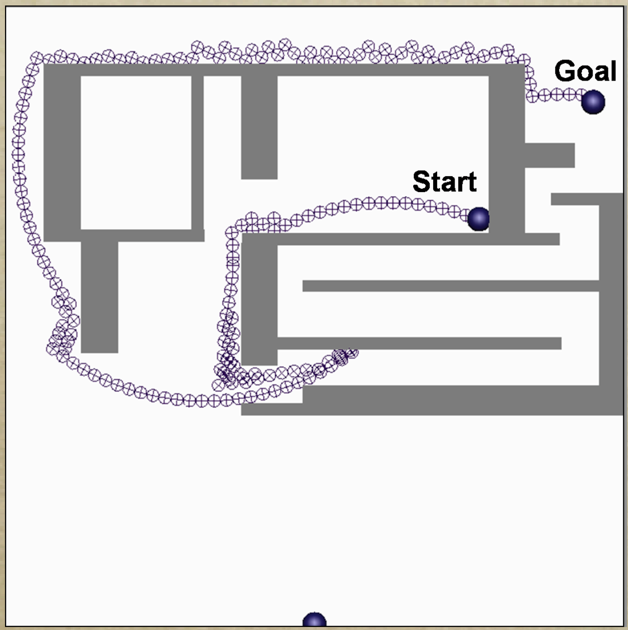
- *In graph on prev page, to find a path from s to g, what is the expansion order for*
  - *DFS, BFS*
  - *Heuristic search using h = Manhattan*
  - *A* using $f = g + h$*
- *Assume we can detect when we reach a node via two different paths, and avoid duplicating it on the queue*

Spatial Planning

Plans in Space…



Optimal Solution



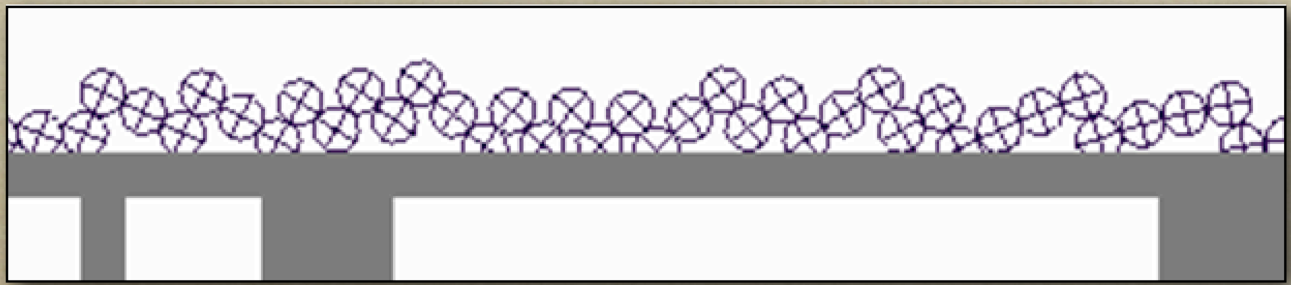
End-effector Trajectory

- *Above, we saw A* for spatial planning (in contrast to, e.g., jobshop scheduling)*

What's wrong w/ A* guarantees?

- ***(optimality)*** *A* finds a solution of cost g**
- ***(efficiency)*** *A* expands no nodes that have f(node) > g**

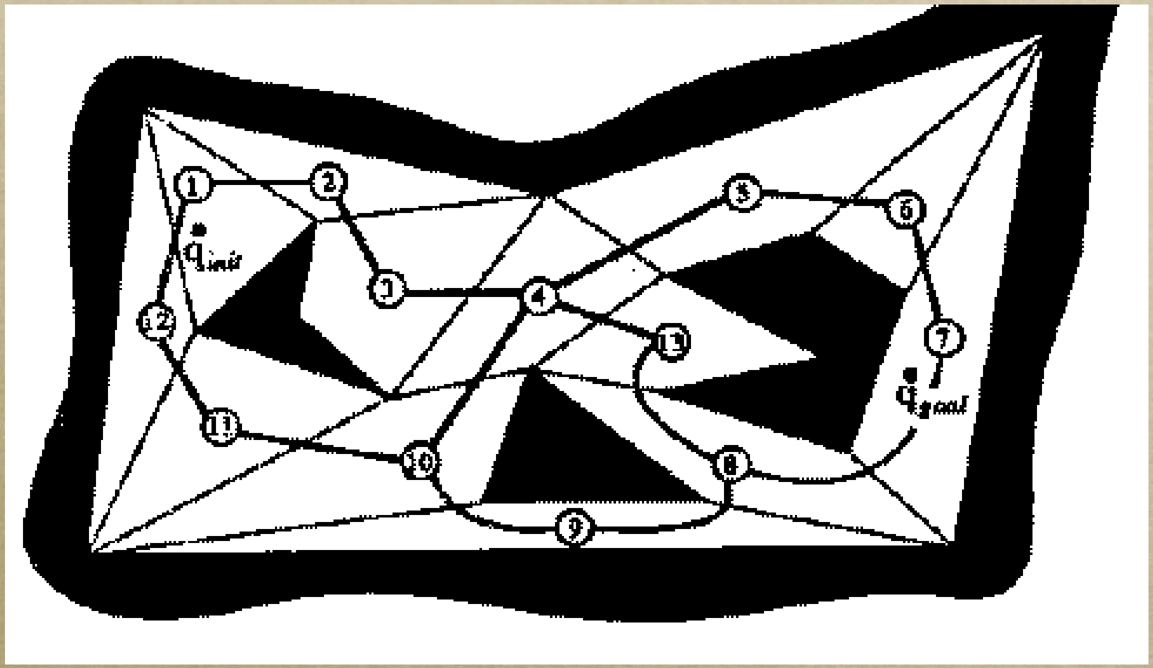
What's wrong with A*?



- *Discretized space into tiny little chunks*
  - *a few degrees rotation of a joint*
  - **Lots** *of states* ⇒ *slow*
- *Discretized actions too*
  - *one joint at a time, discrete angles*
- *Results in jagged paths*

What’s wrong with A*?

Wouldn't it be nice…

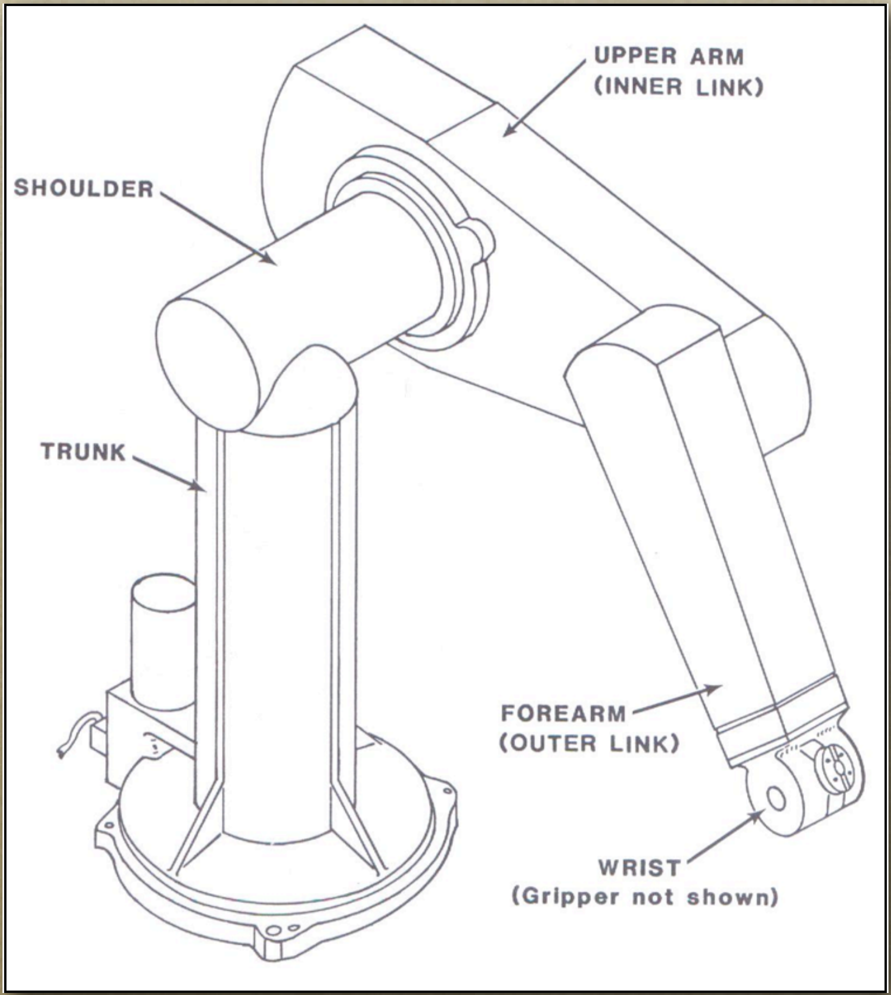


- *… if we could break things up based more on the real geometry of the world?*
- Robot Motion Planning *by Jean-Claude Latombe*

Physical system

- *A moderate number of real-valued coordinates*
- *Deterministic, continuous dynamics*
- *Continuous goal set (or a few pieces)*
- *Cost = time, work, torque, …*

Typical physical system



A kinematic chain

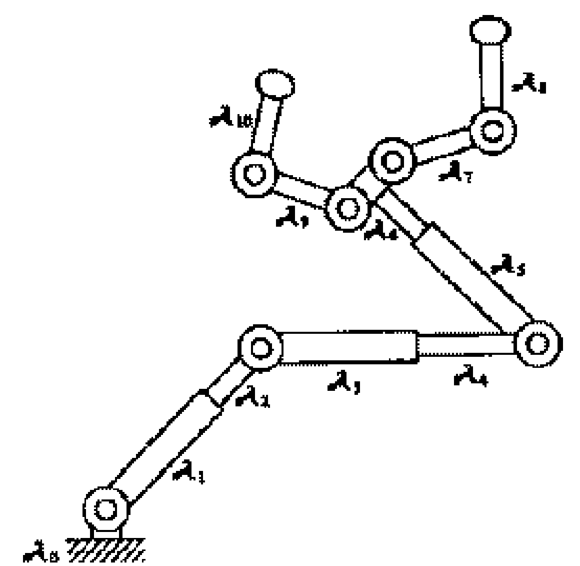


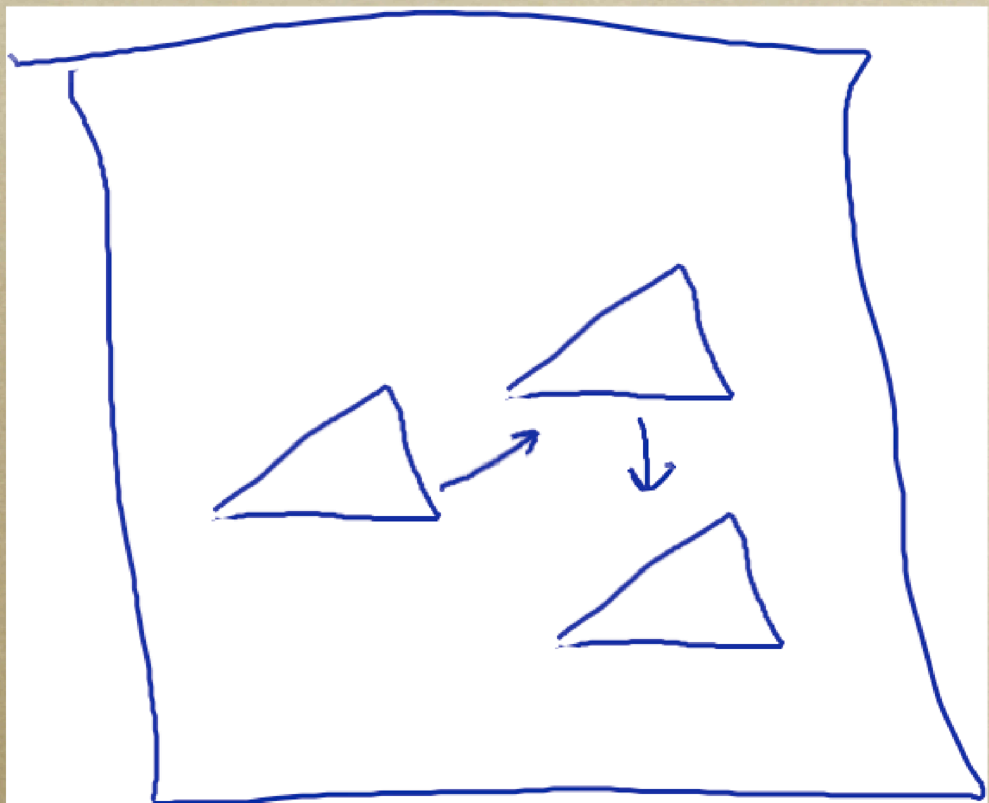
Fig. 11. Structure of the 10-DOF manipulator.

- *Rigid links connected by joints*
  - *revolute or prismatic*
- *Configuration*

  $$\mathbf{q} = (q_1, q_2, \ldots)$$

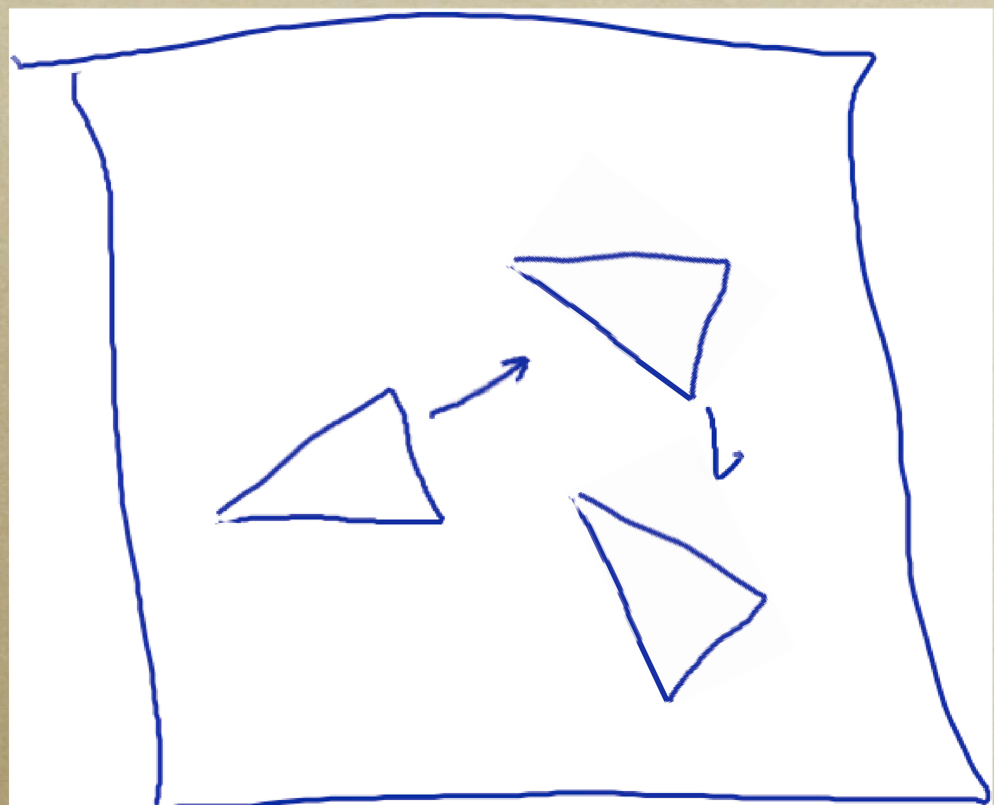
  q_i = *angle or length of joint* i
- *Dimension of* $\mathbf{q}$ = *"degrees of freedom"*

Mobile robots



- *Translating in space = 2 dof*

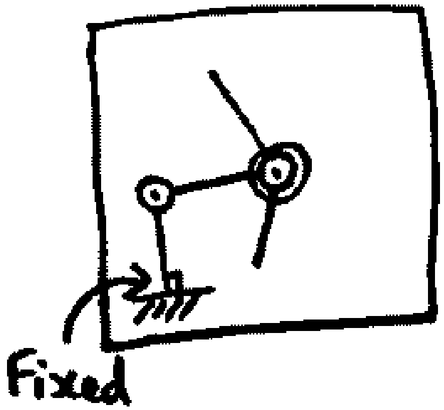
More mobility



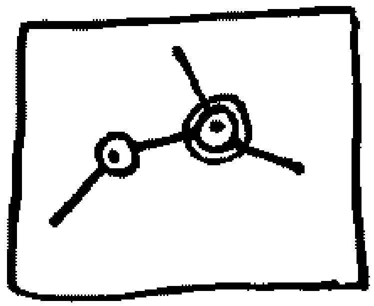
- *Translation + rotation = 3 dof*

Q: How many dofs?

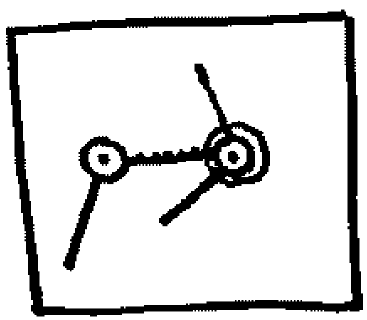
- *3d translation & rotation*



How many dofs?



Free flying
How many dofs?



Midline ——
must always be horizontal.
How many DOFs?

The configuration $\underset{\sim}{q}$ has one real valued entry per DOF.

credit: Andrew Moore

Robot kinematic motion planning

- *Now let's add obstacles*

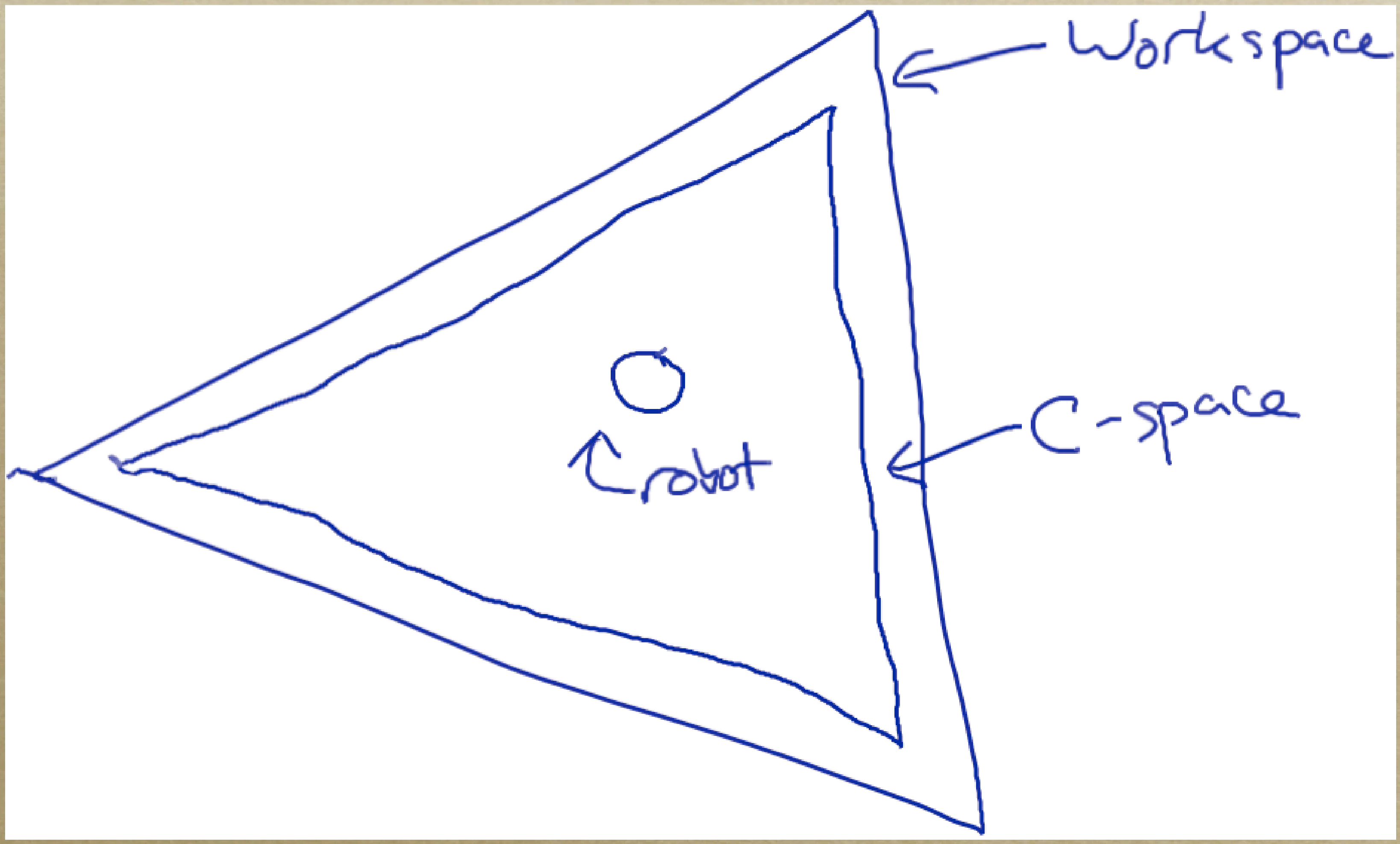
Configuration space

- *For any configuration* $\mathbf{q}$, *can test whether it intersects obstacles*
- *Set of legal configs is "configuration space" C (a subset of* $\mathbb{R}^{dofs}$*)*
- *Path is a continuous function from [0,1] into C with* $q(0) = \mathbf{q_s}$ *and* $q(1) = \mathbf{q_g}$

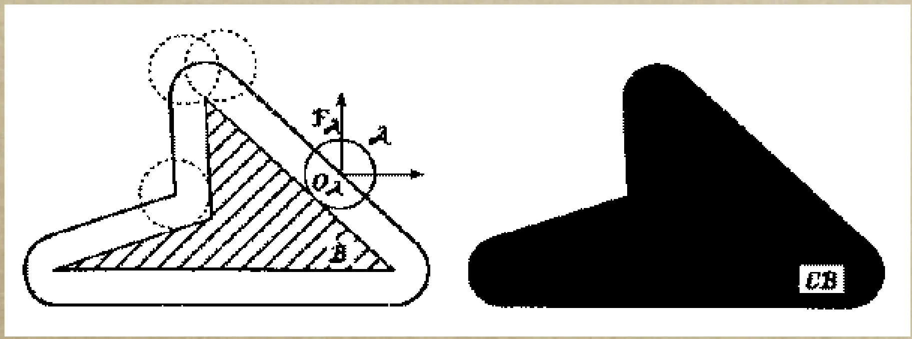
Note: dynamic planning

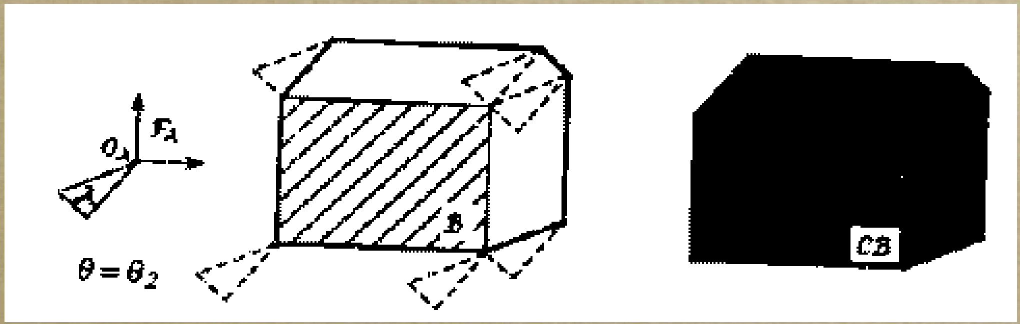
- *Includes inertia as well as configuration*
- $\mathbf{q}, \dot{\mathbf{q}}$
- *Harder, since twice as many dofs*
- *More later…*

C-space example



More C-space examples





Another C-space example

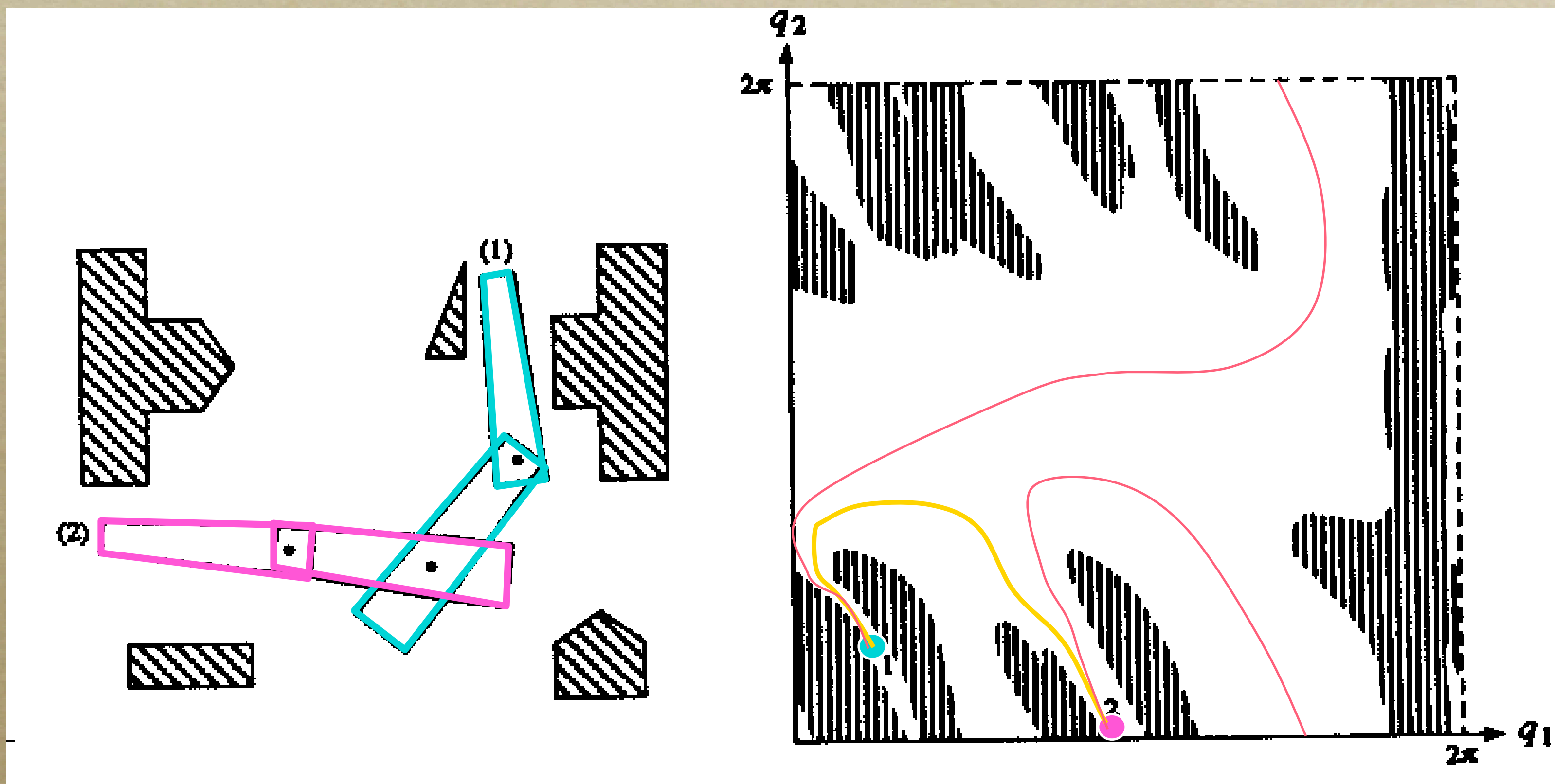
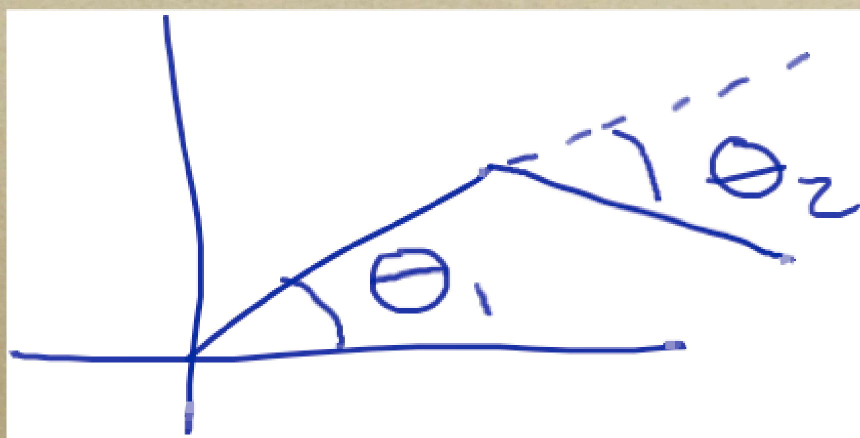


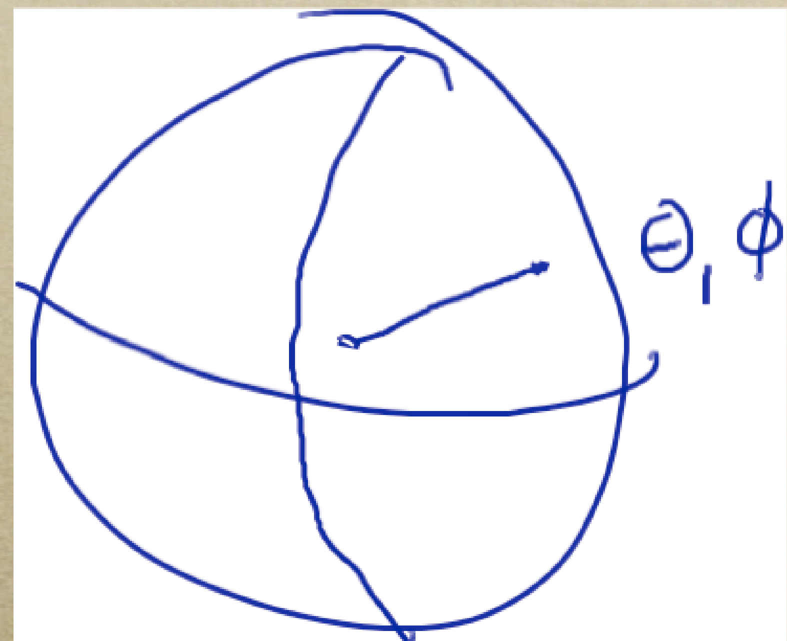
image: J Kuffner

Topology of C-space

- *Topology of C-space can be something other than the familiar Euclidean world*
- *E.g. set of angles = unit circle = SO(2)*
  - *not [0, 2π) !*
- *Ball & socket joint (3d angle) ⊆ unit sphere = SO(3)*

Topology example



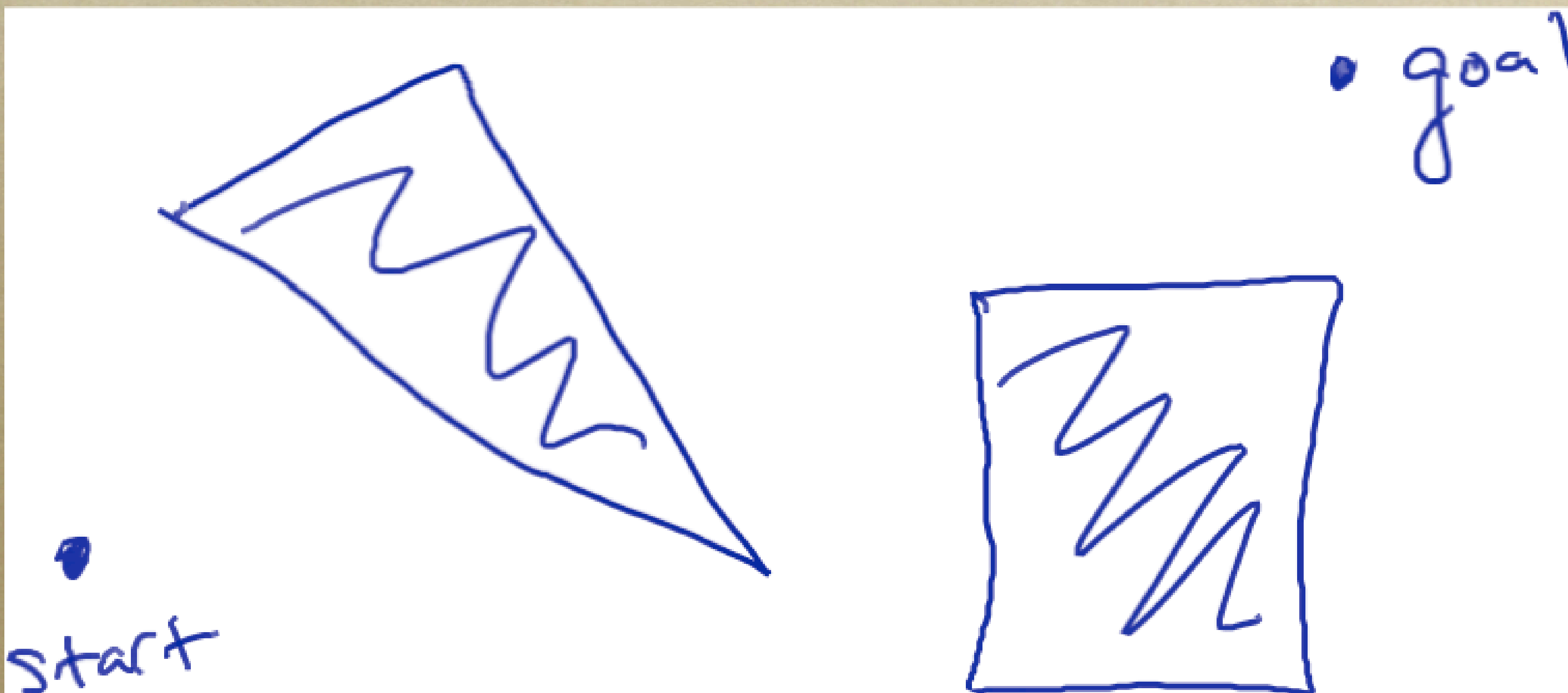


- *Compare L to R: 2 planar angles v. one solid angle — both 2 dof (and neither the same as Euclidean 2-space)*

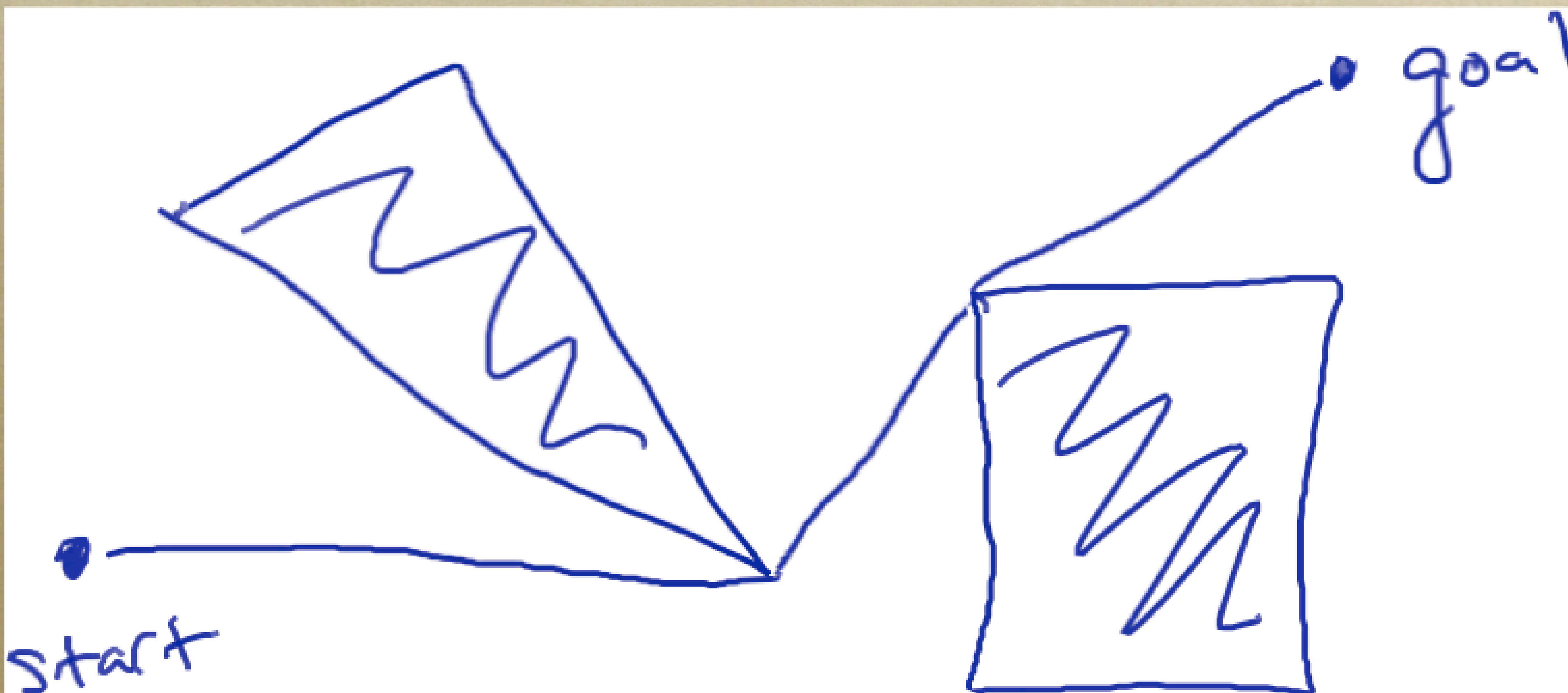
Back to planning

- *Complaint with A* was that it didn't break up space intelligently*
- *How might we do better?*
- *Lots of roboticists have given lots of answers!*

Shortest path in C-space



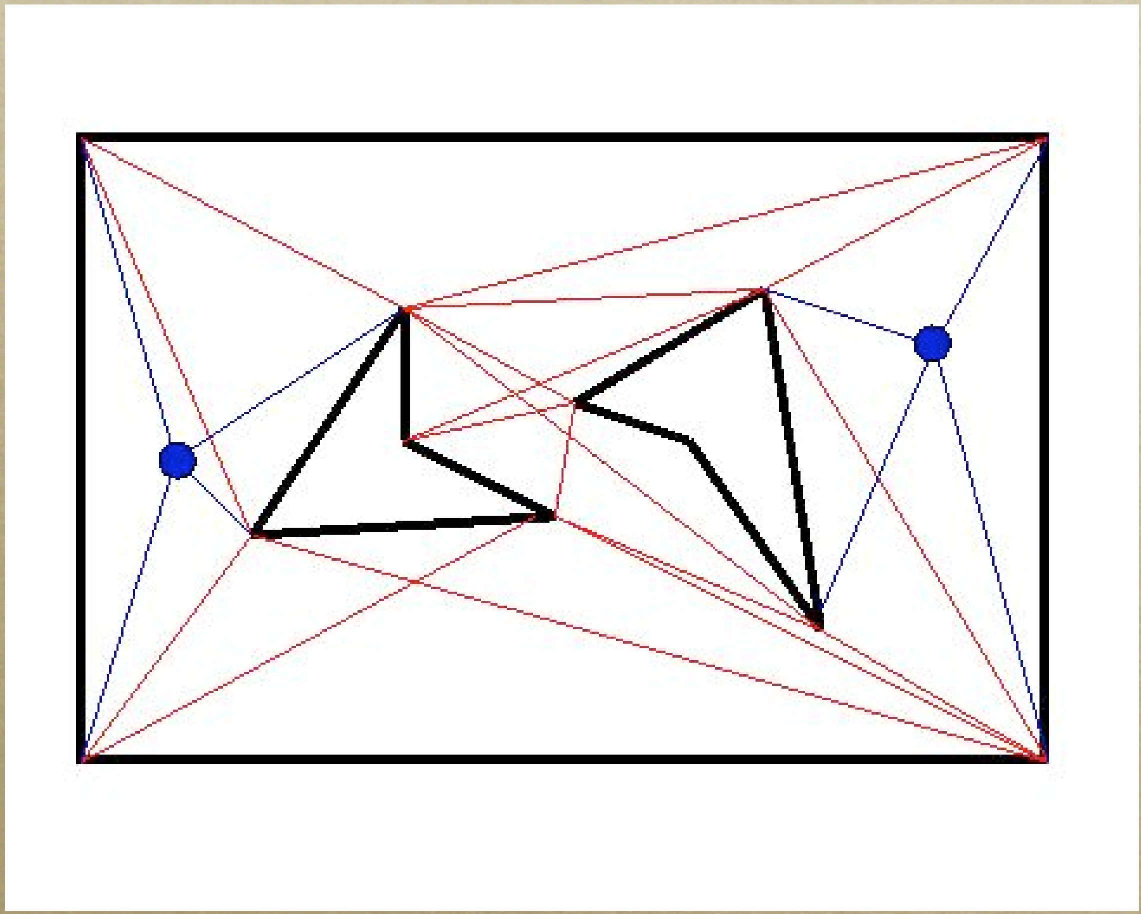
Shortest path in C-space



Shortest path

- *Suppose a planar polygonal C-space*
- *Shortest path in C-space is a sequence of line segments*
- *Each segment's ends are either start or goal or one of the vertices in C-space*
- *In 3-d or higher, might lie on edge, face, hyperface, …*

Visibility graph



http://www.cse.psu.edu/~rsharma/robotics/notes/notes2.html

Naive algorithm

For i = 1 … points

For j = 1 … points

included = t

For k = 1 … edges

if segment ij intersects edge k

included = f

Complexity

- *Naive algorithm is $O(n^3)$ in planar C-space*
- *For algorithms that run faster, $O(n^2)$ and $O(k + n \log n)$, see [Latombe, pg 157]*
  - *k = number of edges that wind up in visibility graph*
- *Once we have graph, search it!*

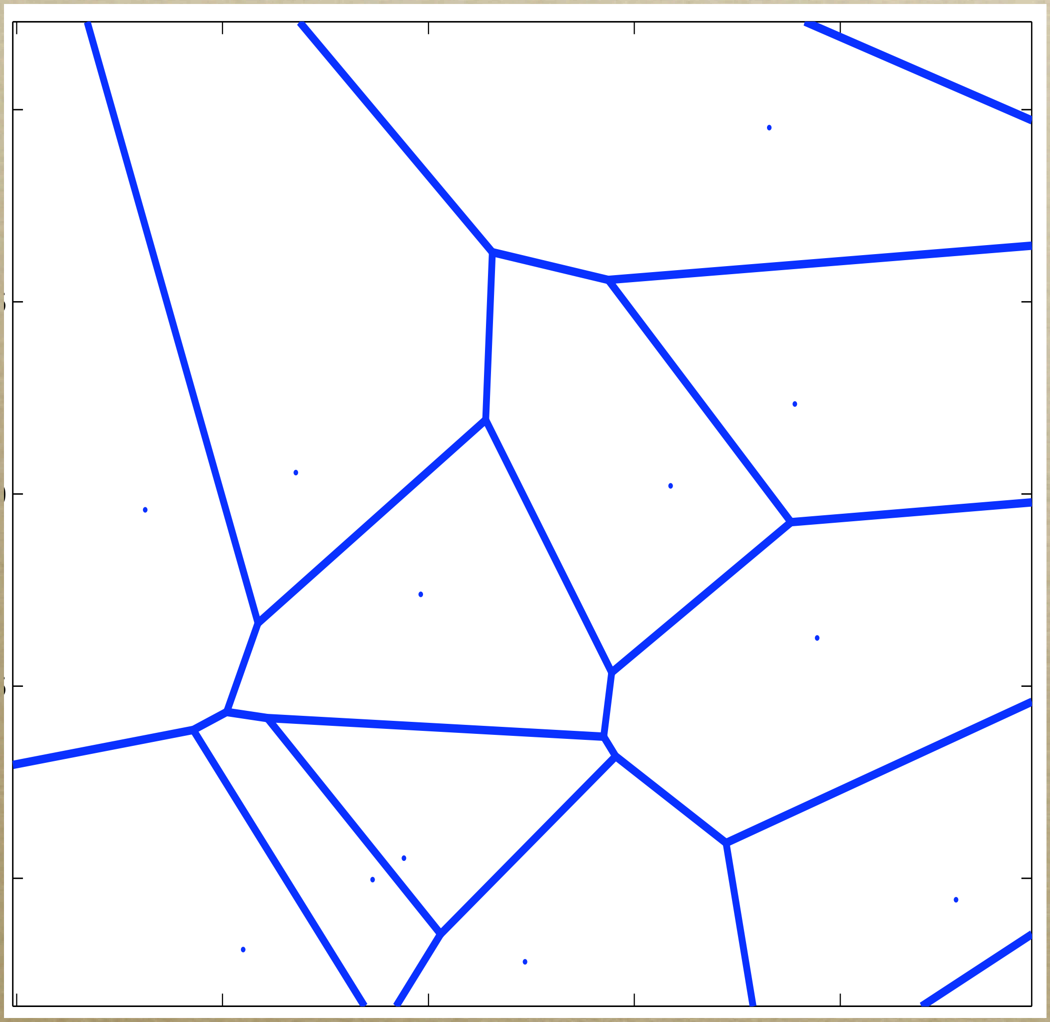
Discussion of visibility graph

- *Good: finds shortest path*
- *Bad: complex C-space yields long runtime, even if problem is easy*
  - *get my 23-dof manipulator to move 1mm when nearest obstacle is 1m*
- *Bad: no margin for error*

Getting bigger margins

- *Could just pad obstacles*
  - *but how much is enough? might make infeasible…*
- *What if we try to stay as far away from obstacles as possible?*

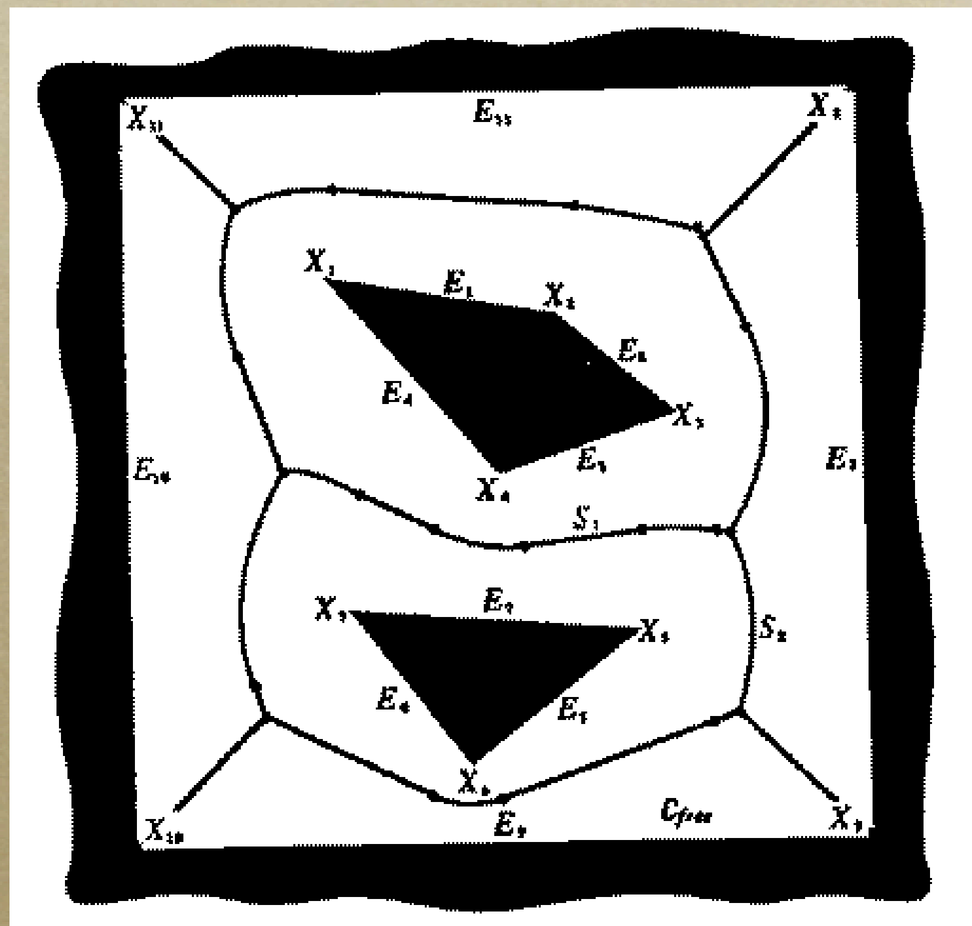
Voronoi graph



Voronoi graph

- *Given a set of point obstacles*
- *Find all places that are equidistant from two or more of them*
- *Result: network of line segments*
- *Called* Voronoi graph
- *Each line stays as far away as possible from two obstacles while still going between them*

Voronoi from polygonal C-space



Voronoi from polygonal C-space

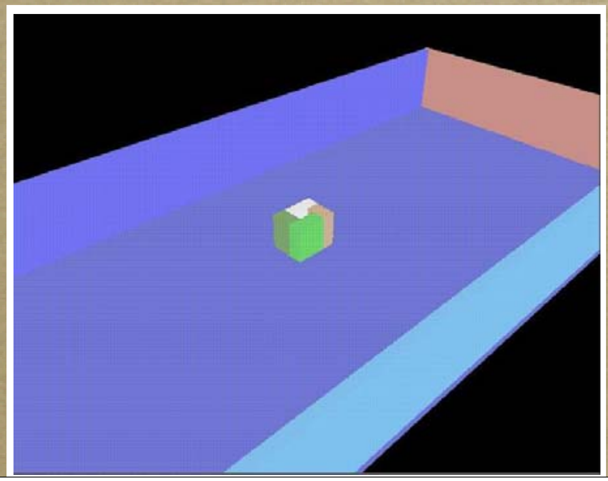
- *Set of points which are equidistant from 2 or more closest points on border of C-space*
- *Polygonal C-space in 2d yields lines & parabolas intersecting at points*
  - *lines from 2 points*
  - *parabolas from line & point*

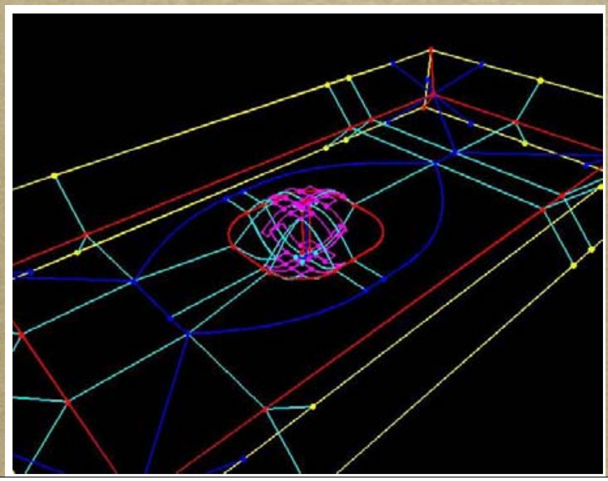
Voronoi method for planning

- *Compute Voronoi diagram of C-space*
- *Go straight from start to nearest point on diagram*
- *Plan within diagram to get near goal (e.g., with A*)*
- *Go straight to goal*

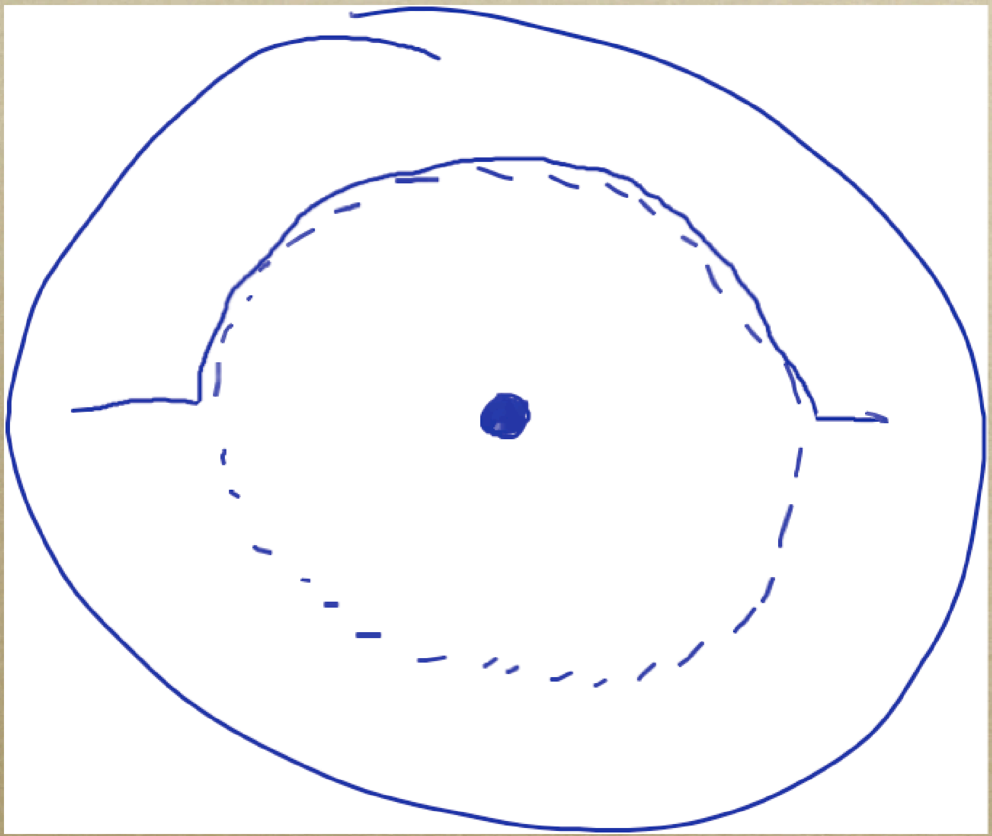
Discussion of Voronoi

- *Good: stays far away from obstacles*
- *Bad: assumes polygons*
- *Bad: gets kind of hard in higher dimensions (but see Howie Choset's web page and book)*



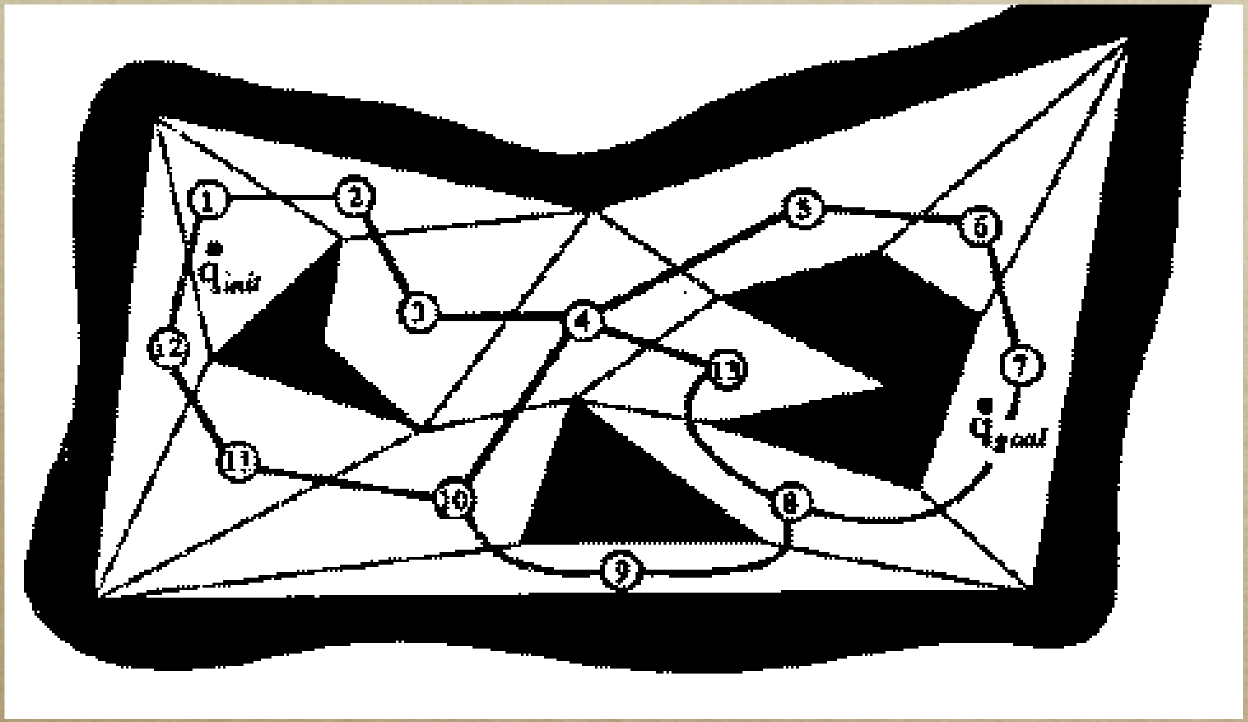


Voronoi discussion



- *Bad: kind of gun-shy about obstacles*

Exact cell decompositions

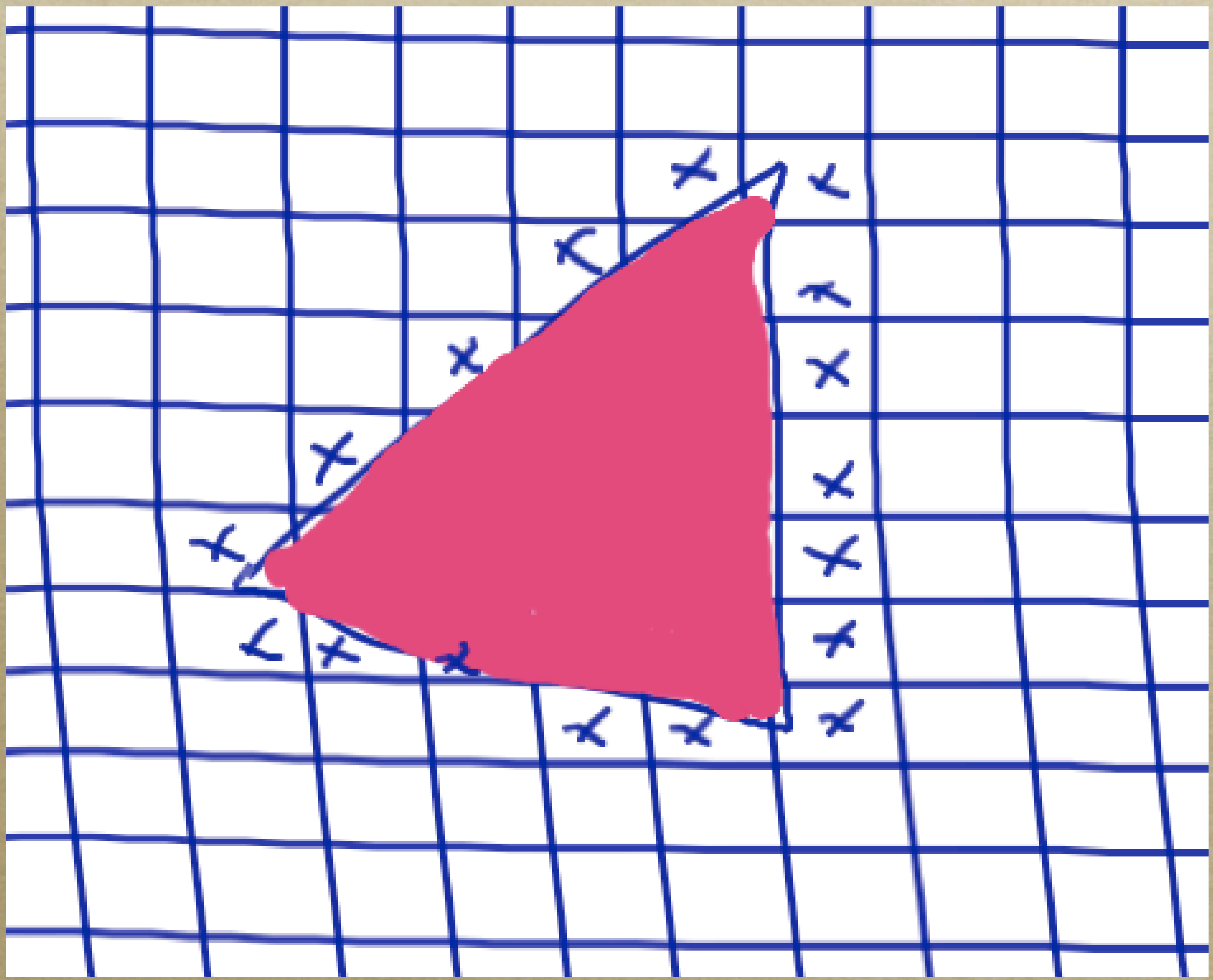


- *We can try to break C-space into a bunch of convex polygons*

Exact cell decompositions

- *Will not discuss how to do*
- *Common approach for video game NPCs*
- *But is also hard in higher than 2d*
- *And can result in wobbly paths*

Approximate cell decompositions



Planning algorithm

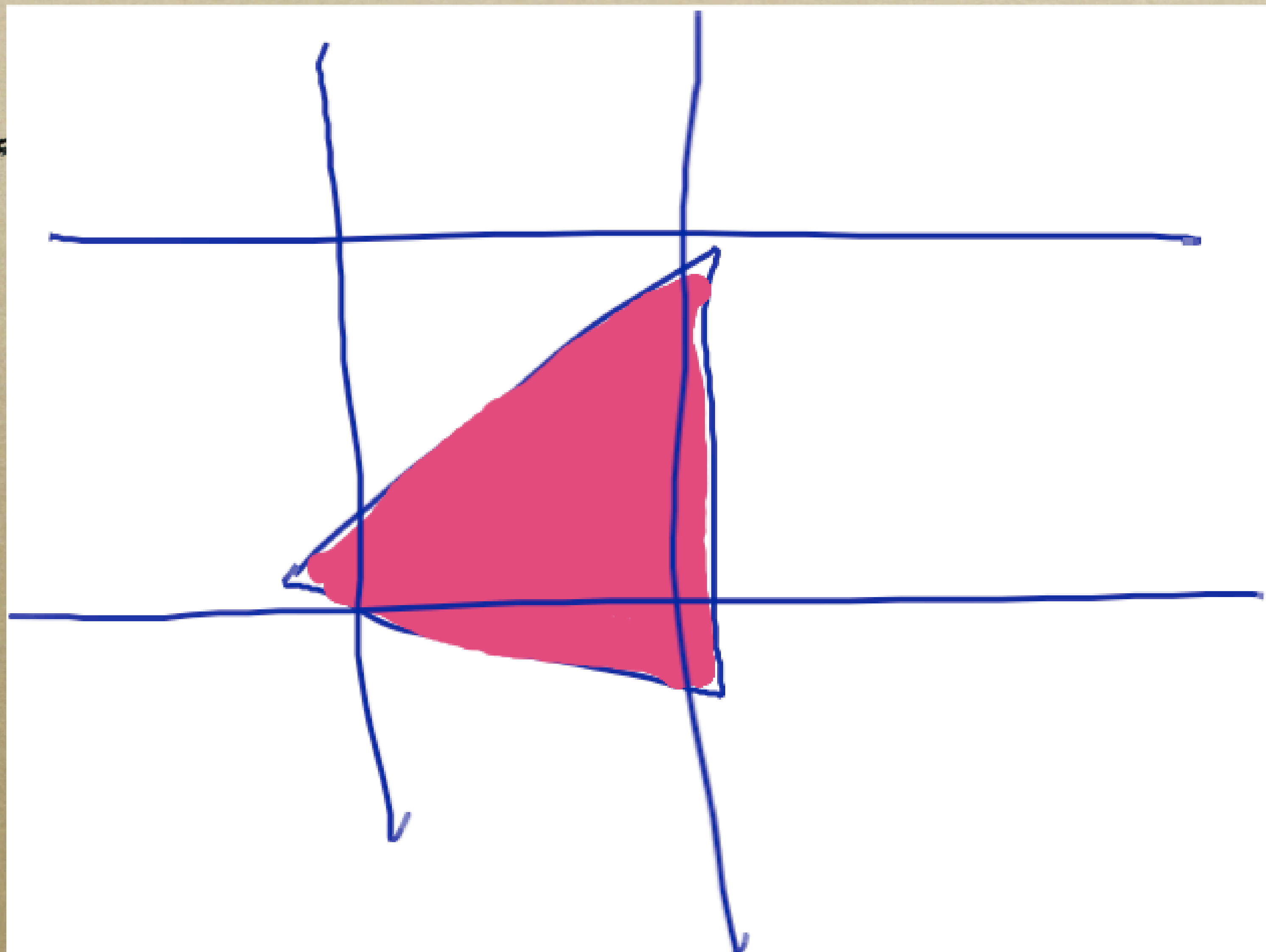
- *Lay down a grid in C-space*
- *Delete cells that intersect obstacles*
- *Connect neighbors*
- *A**
- *If no path, double resolution and try again*
    - *never know when we're done*

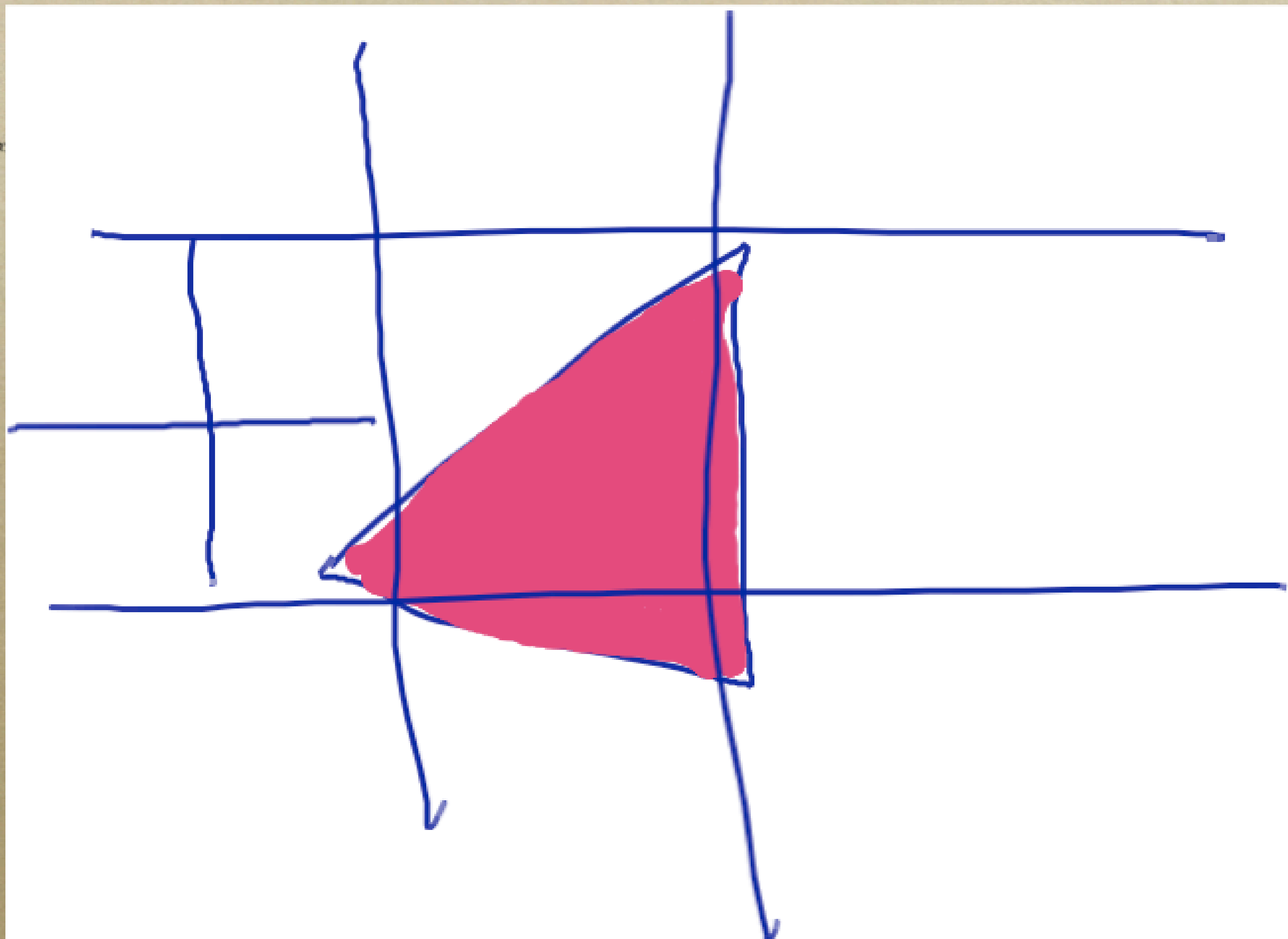
Approximate cell decomposition

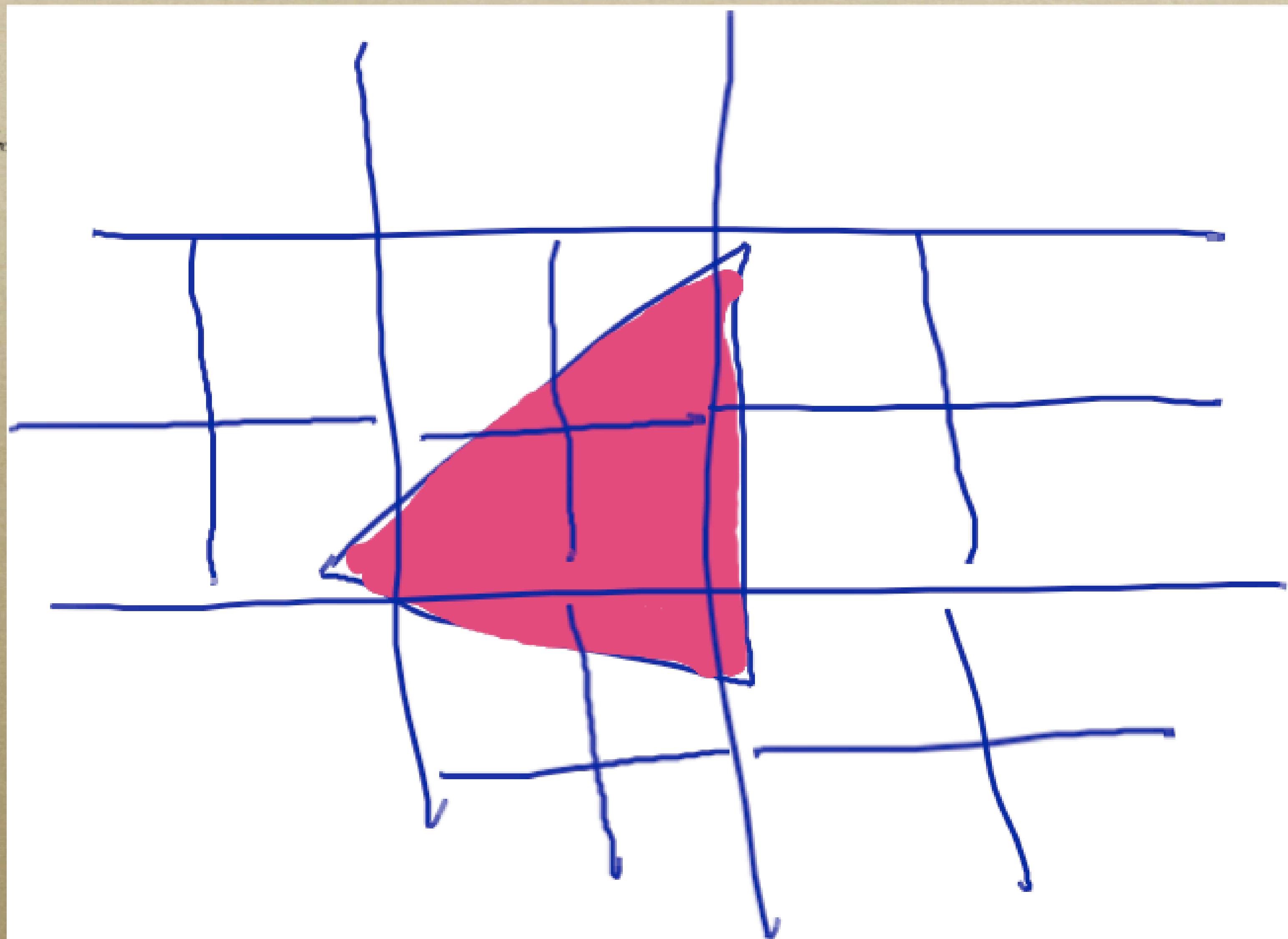
- *This decomposition is what we were using for A* in examples from above*
- *Works pretty well except:*
  - *need high resolution near obstacles*
  - *want low res away from obstacles*

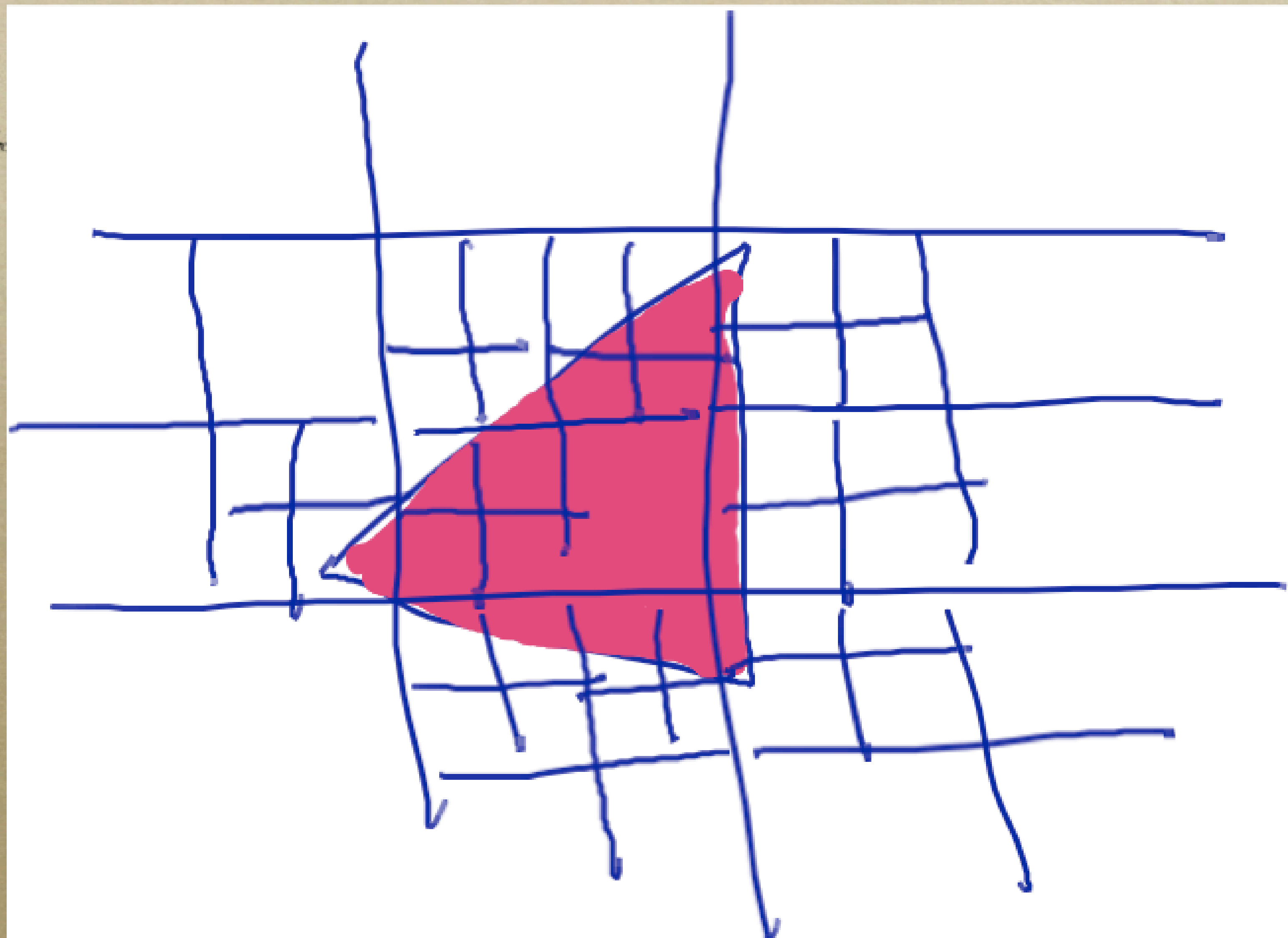
Fix: variable resolution

- *Lay down a coarse grid*
- *Split cells that intersect obstacle borders*
    - *empty cells good*
    - *full cells also don't need splitting*
- *Stop at fine resolution*
- *Data structure: quadtree*









Discussion

- *Works pretty well, except:*
  - *Still don't know when to stop*
  - *Won't find shortest path*
  - *Still doesn't really scale to high-d*

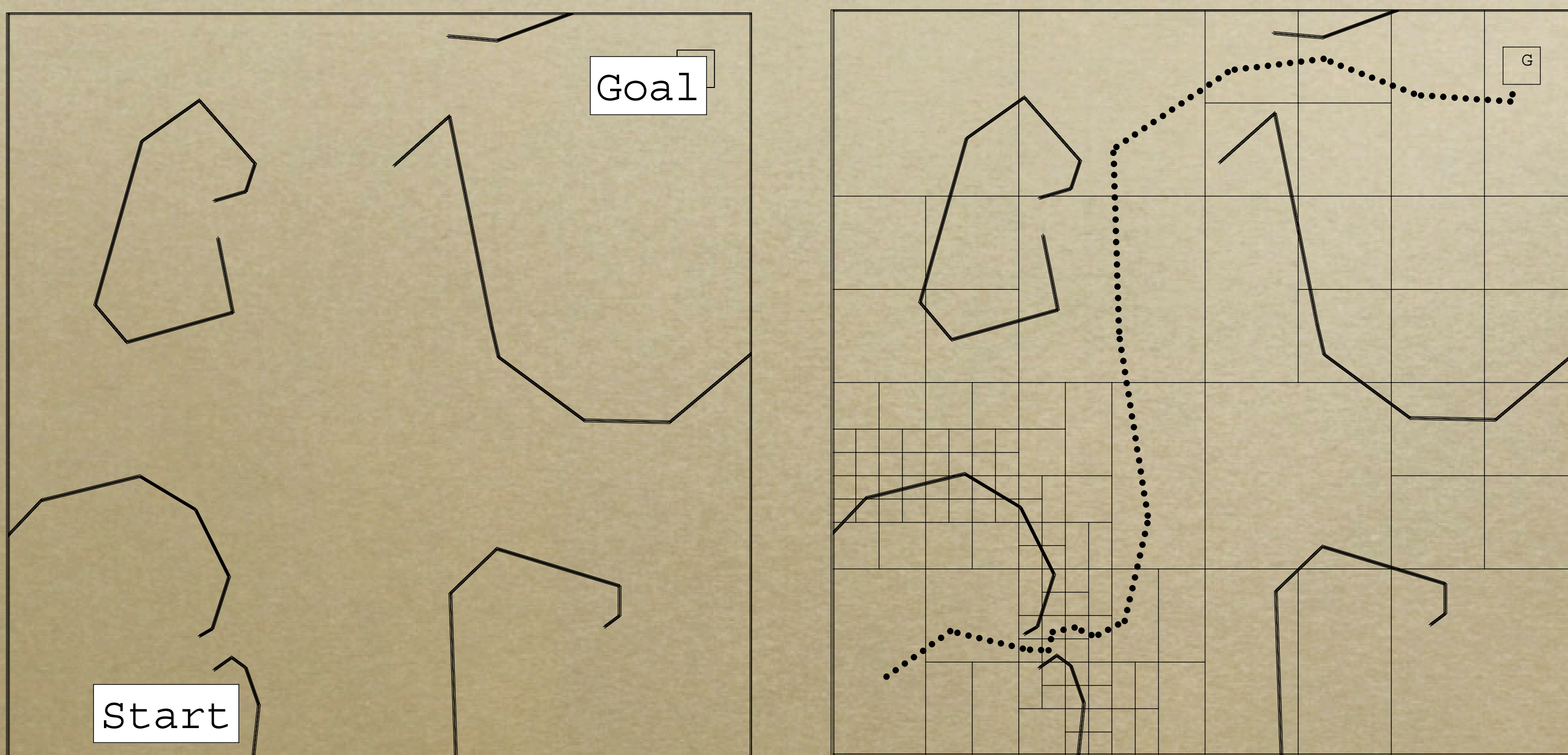
Better yet

- *Adaptive decomposition*
- *Split only cells that actually make a difference*
    - *are on path from start*
    - *make a difference to our policy*

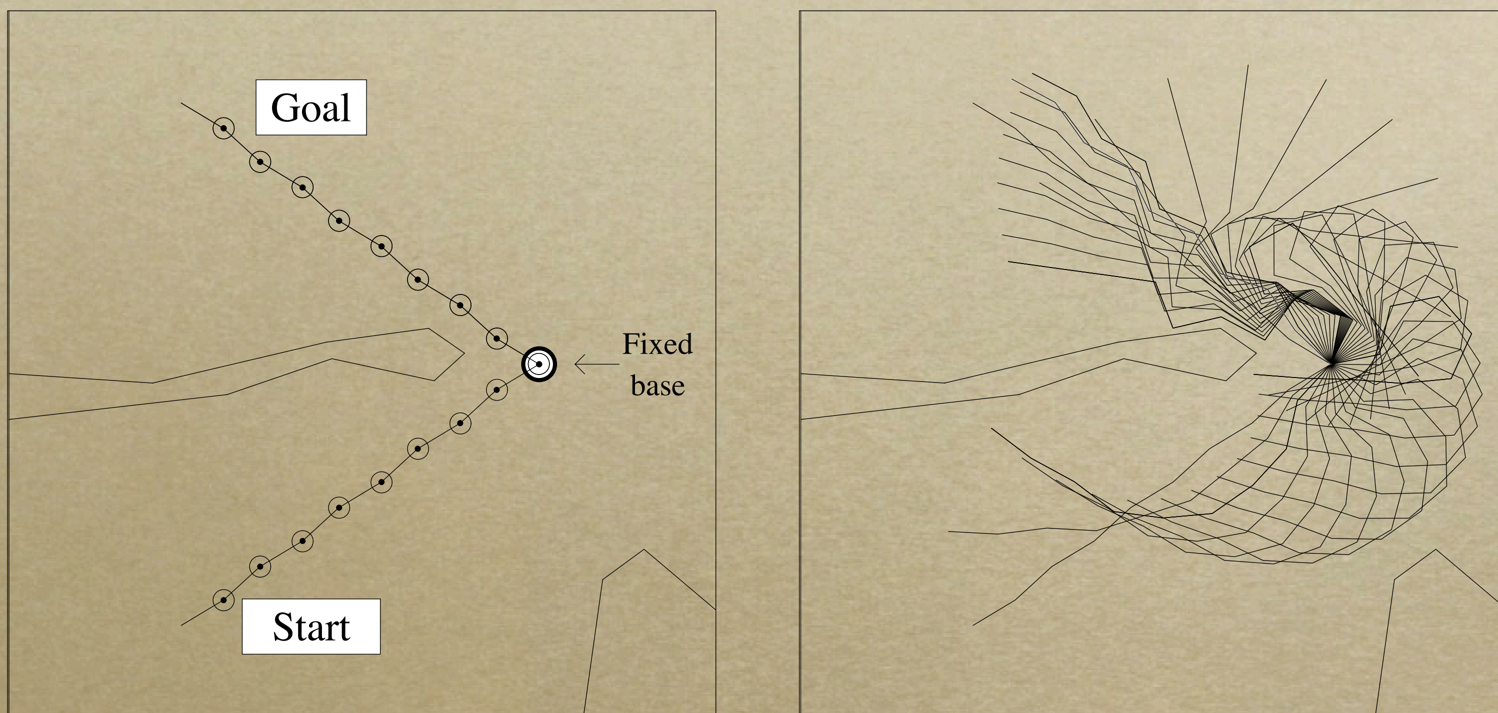
Parti-game algorithm

- *Try actions from several points per cell*
- *Try to plan a path from start to goal*
- *On the way, pretend an opponent gets to choose which outcome happens (out of all that have been observed in this cell)*
- *If we can get to goal, we win*
- *Otherwise we can split a cell*

Parti-game example



9dof planar arm



85 partitions total

Parti-game paper

- *Andrew Moore and Chris Atkeson*. The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces
- *http://www.autonlab.org/autonweb/14699.html*

Randomness in search

Rapidly-exploring Random Trees

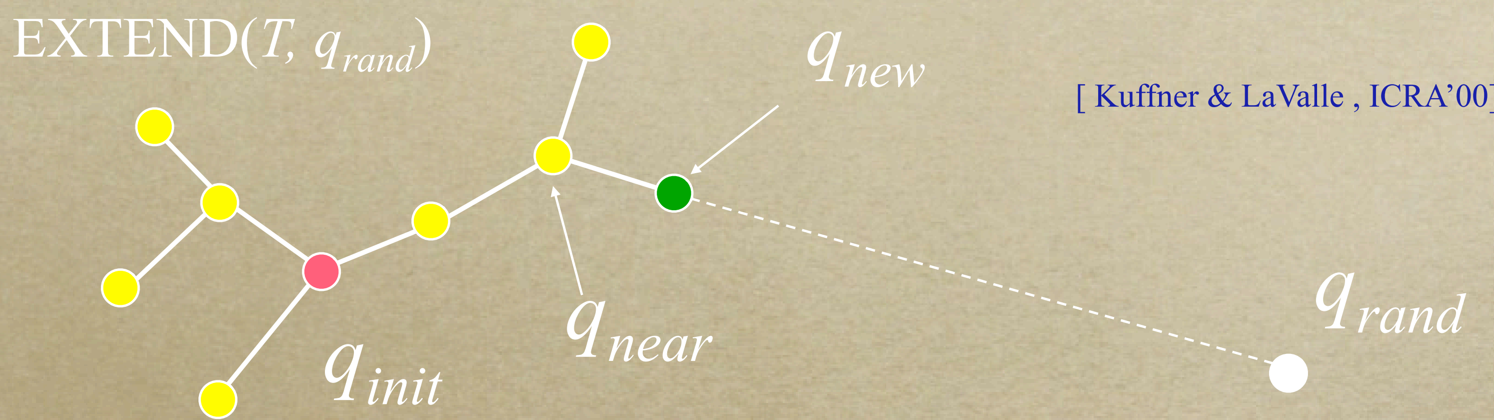
- *Put landmarks into C-space*
- *Break up C-space into Voronoi regions around landmarks*
- *Put landmarks densely only if high resolution is needed to find a path*
- *Will not guarantee optimal path (*)*

RRT assumptions

- *RANDOM_CONFIG*
    - *samples from a distribution on C-space*
- *EXTEND(***q***,* **q***')*
    - *local controller, heads toward* **q***' from* **q**
    - *stops before hitting obstacle*
- *FIND_NEAREST(***q***, Q)*
    - *searches current tree Q for point near* **q**

Path Planning with RRTs

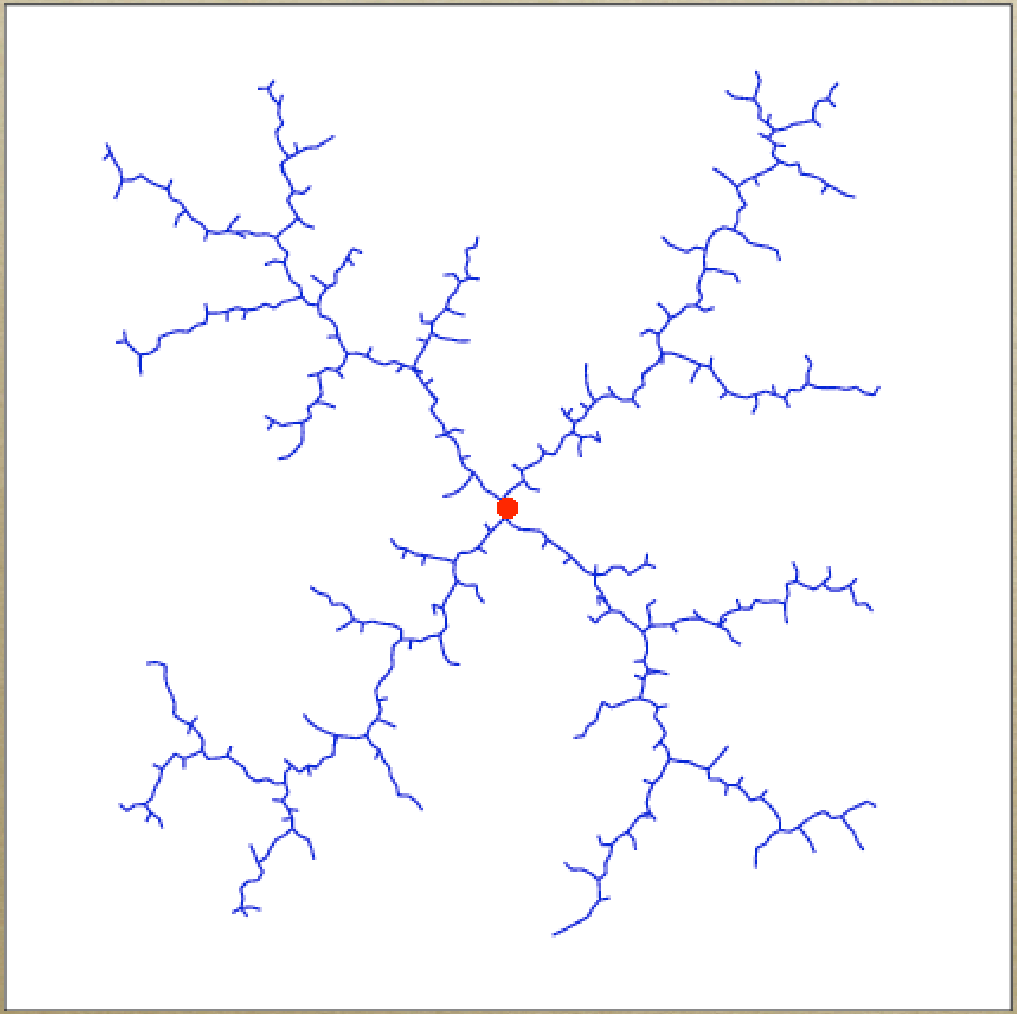
RRT = Rapidly-Exploring Random Tree



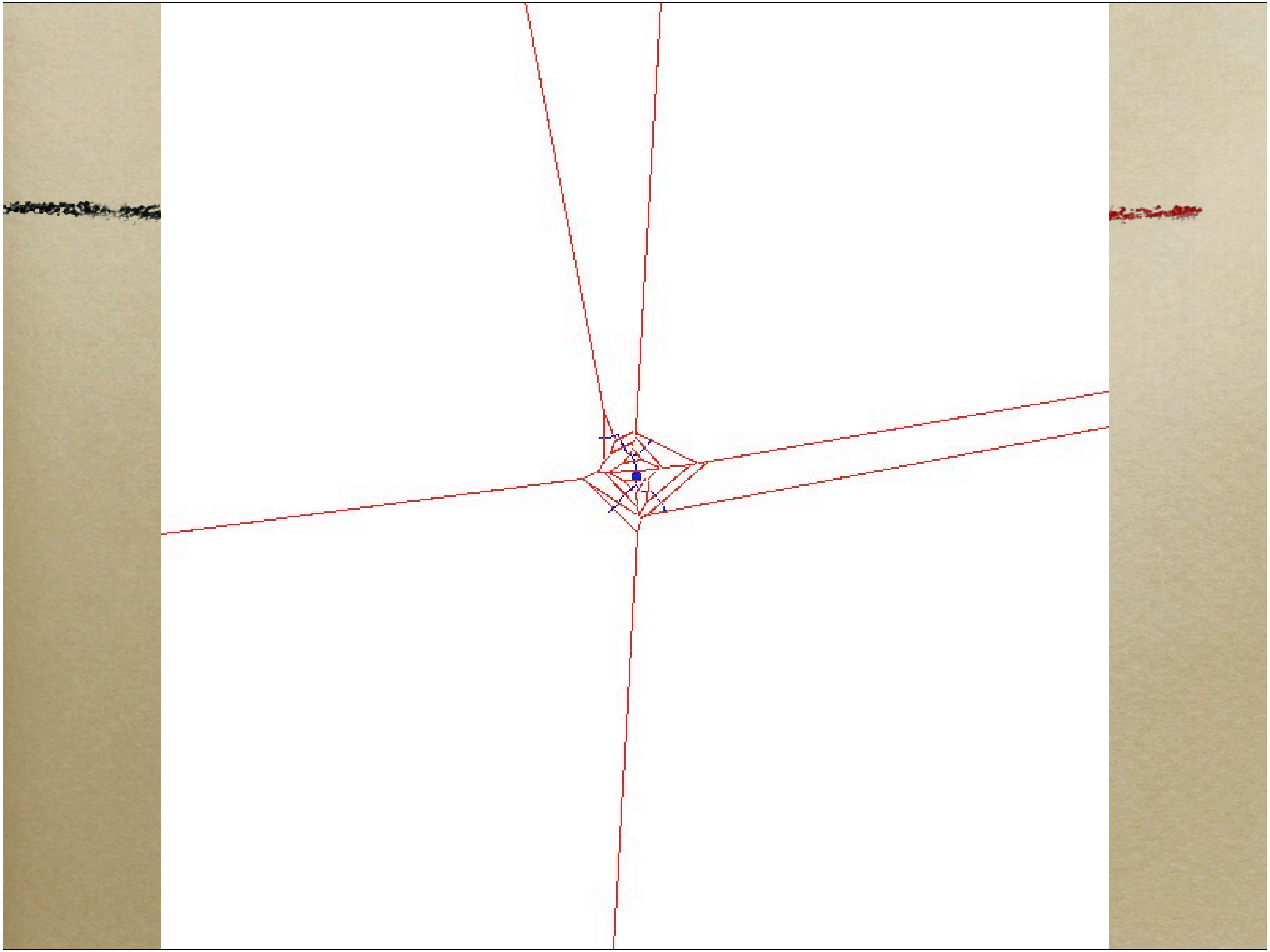

```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

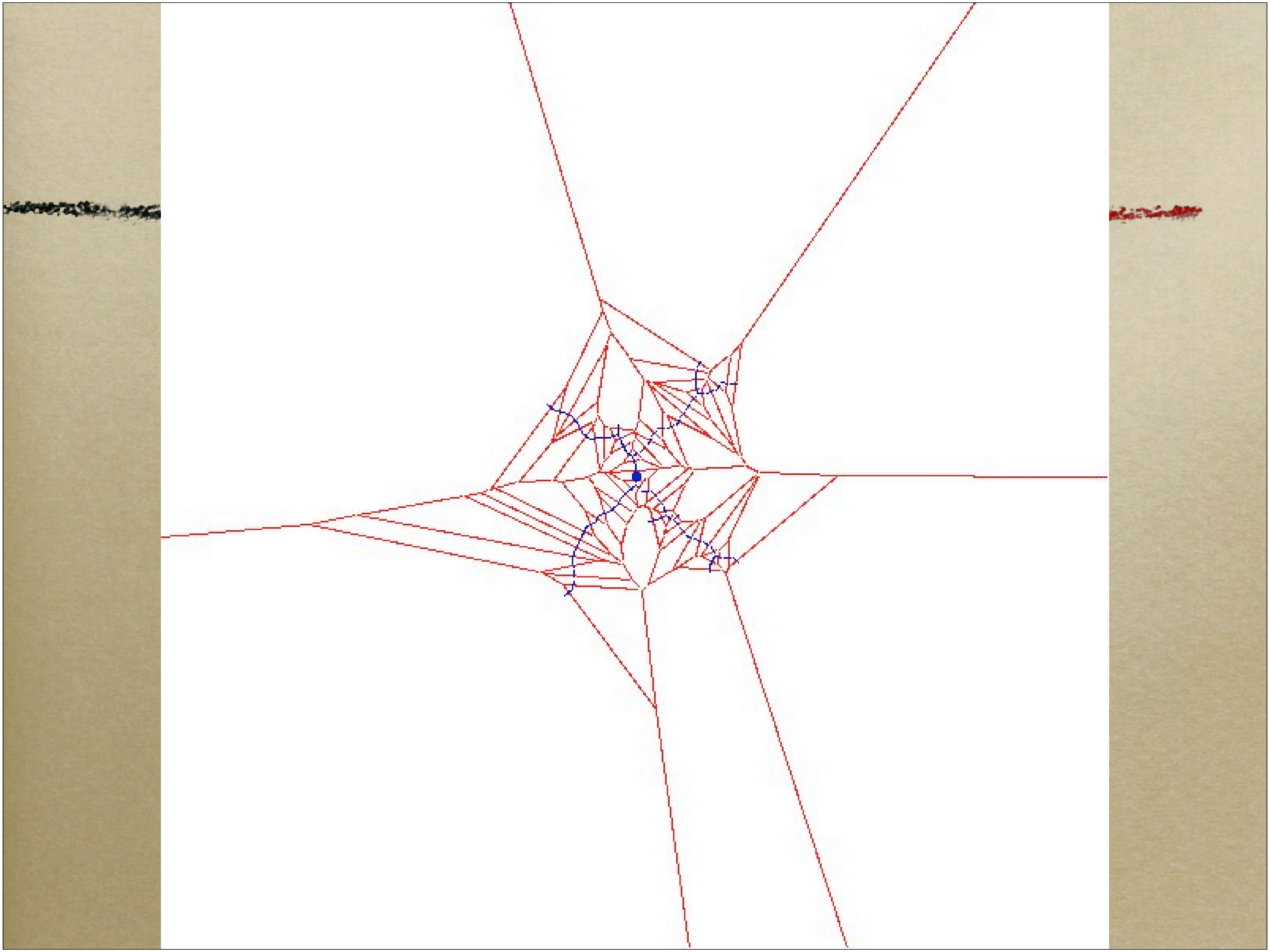
```
EXTEND(T, q) {
    q_near = FIND_NEAREST(q, T)
    q_new = EXTEND(q_near, q)
    T = T + (q_near, q_new)
}
```

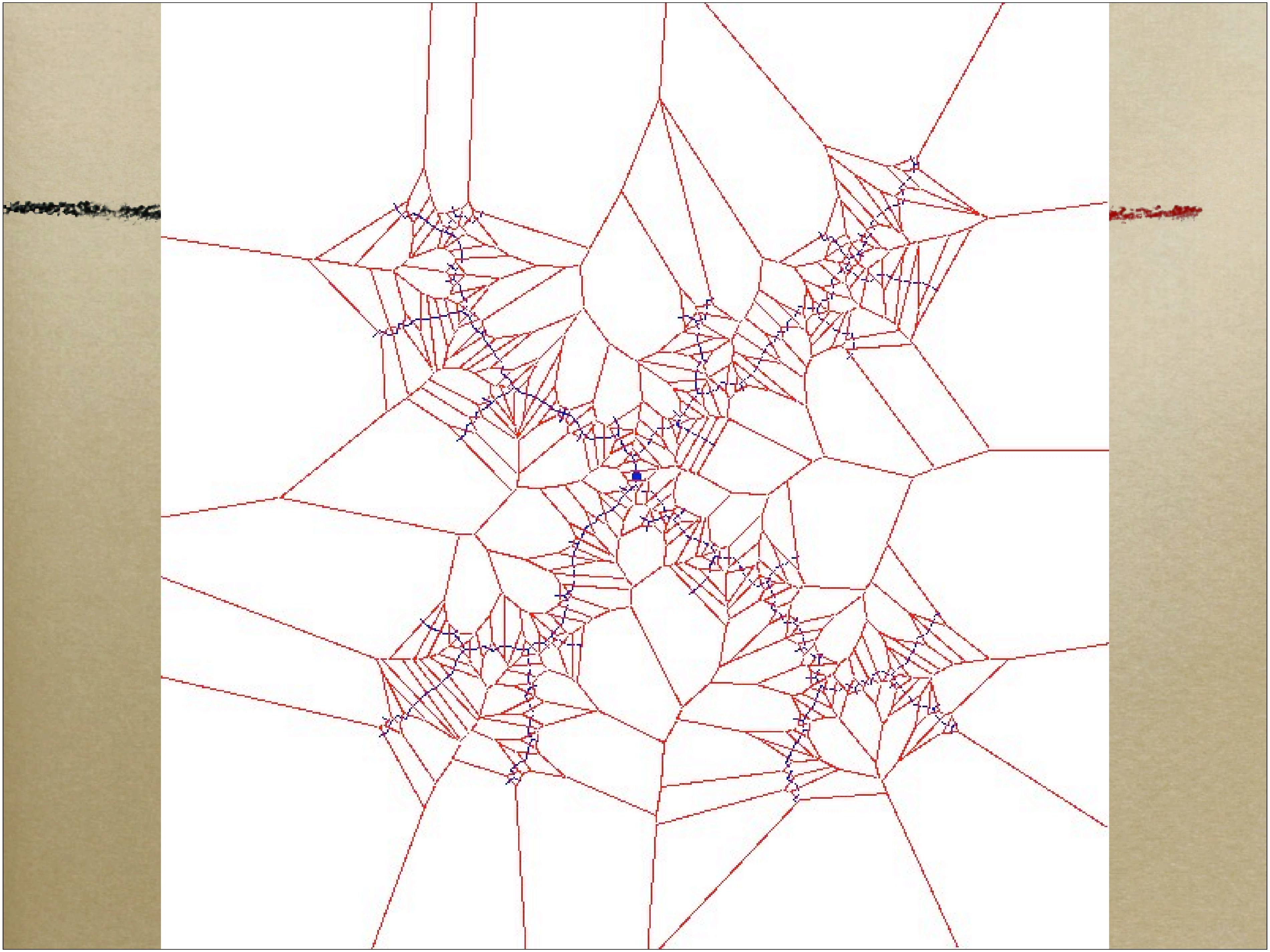
RRT example

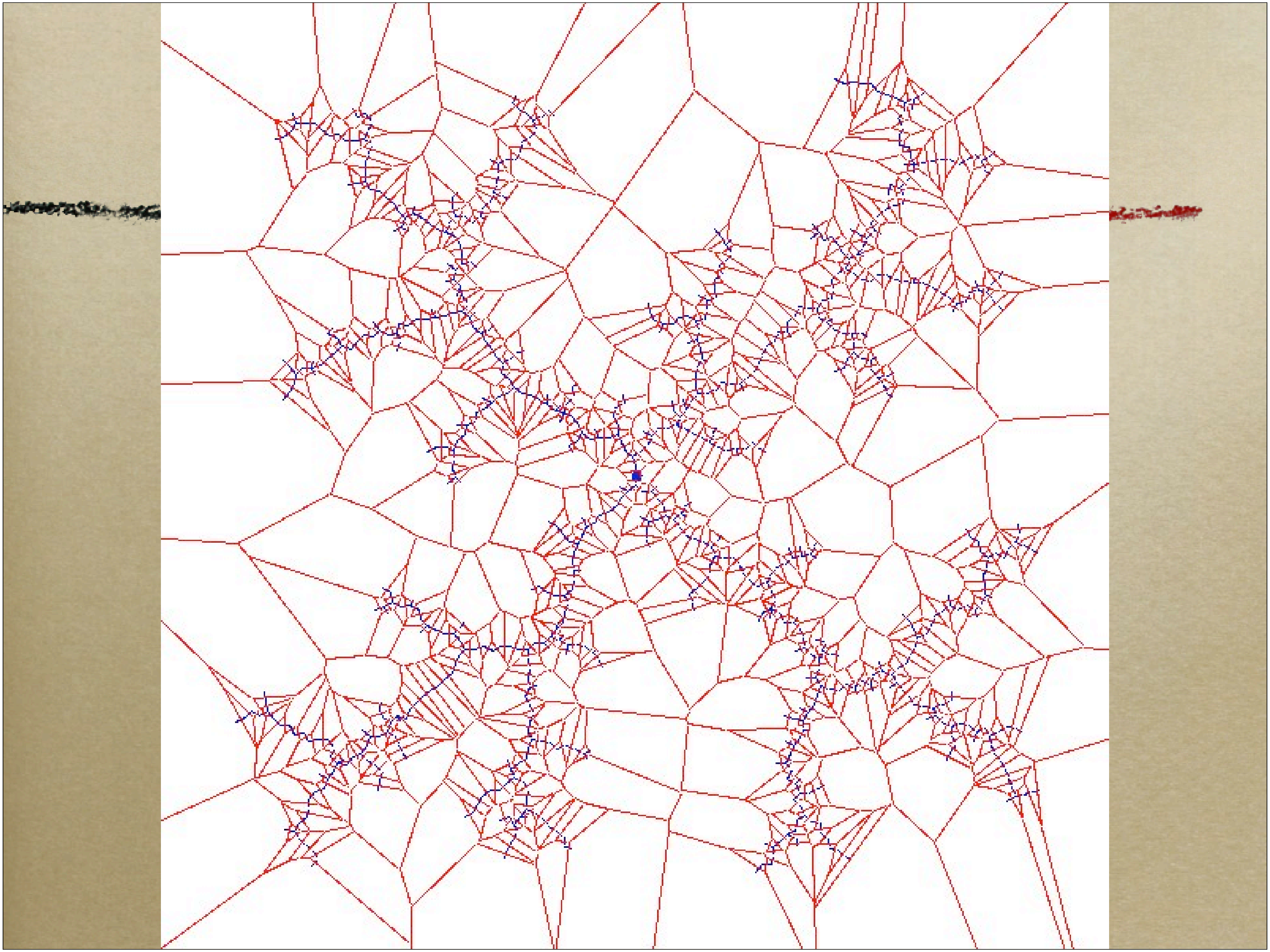


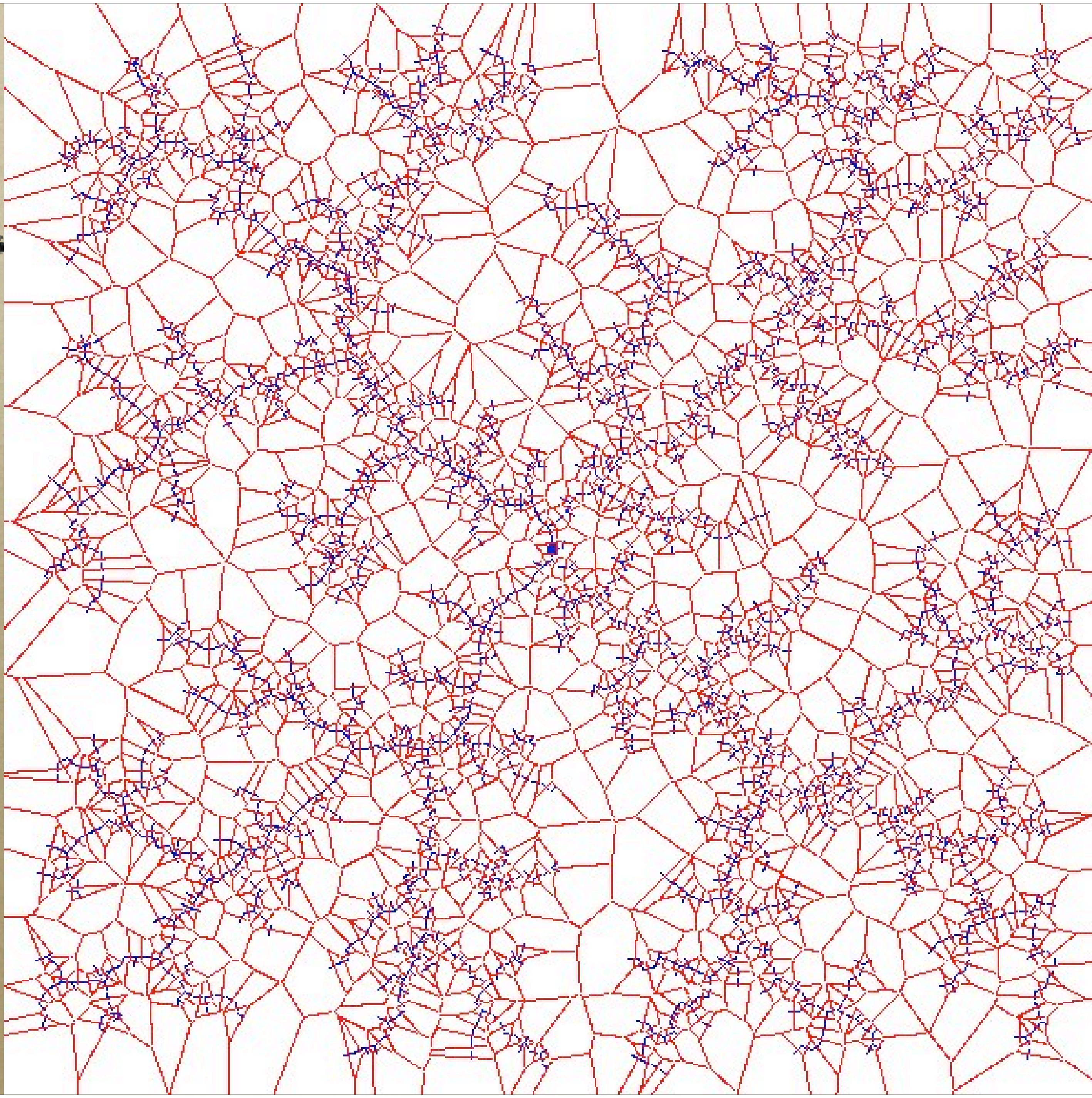
Planar holonomic robot



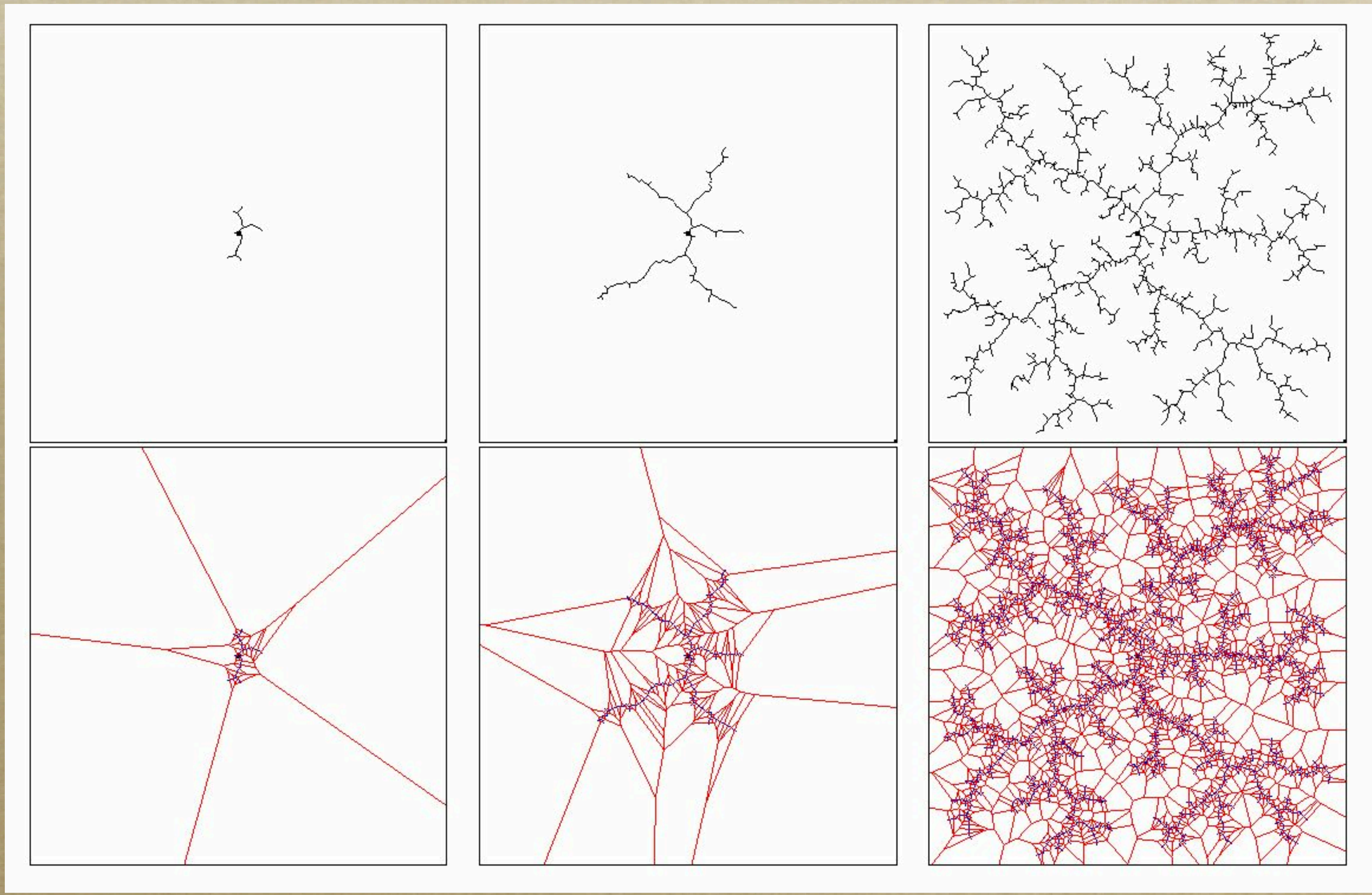




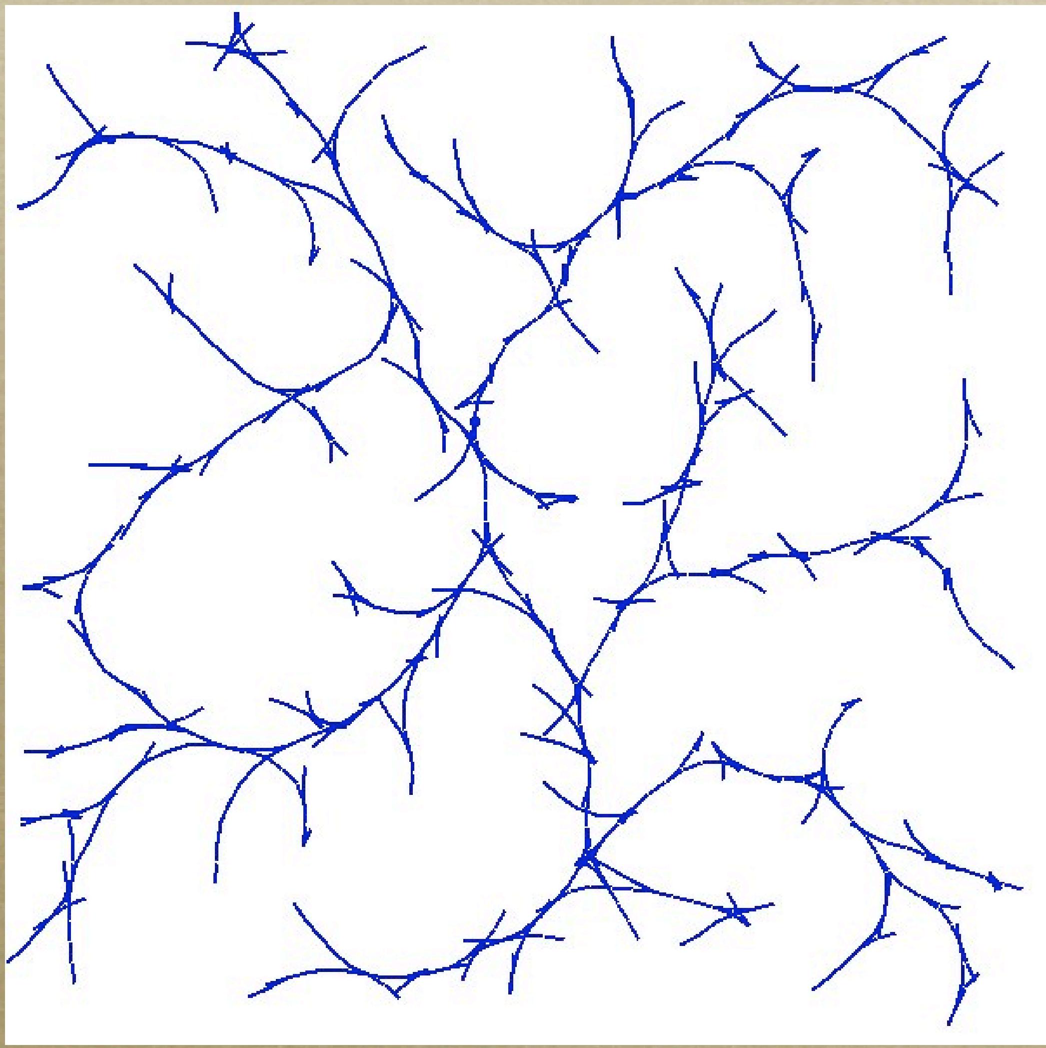




RRT example



RRT for a car (3 dof)

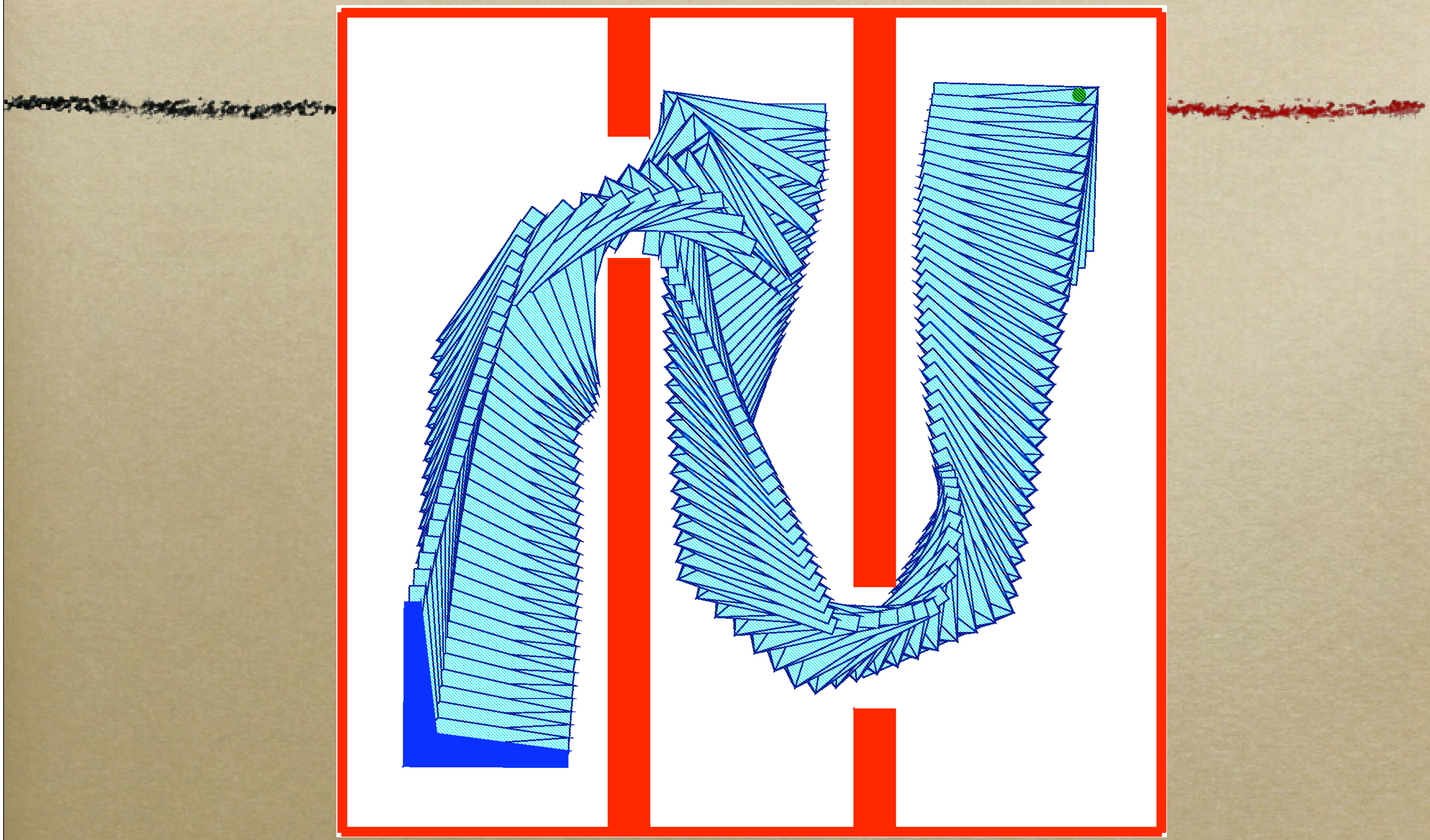


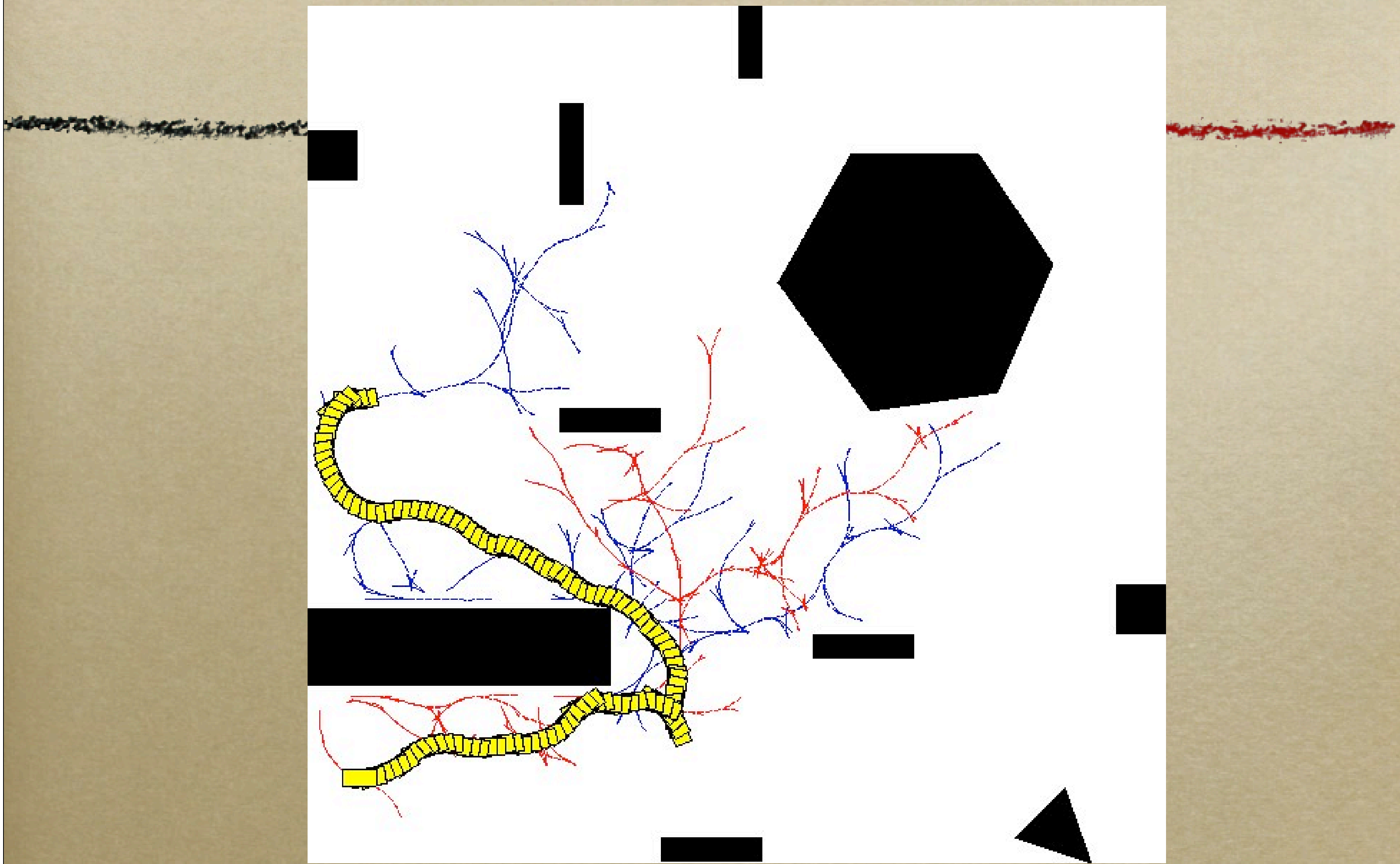
RRTs explore coarse to fine

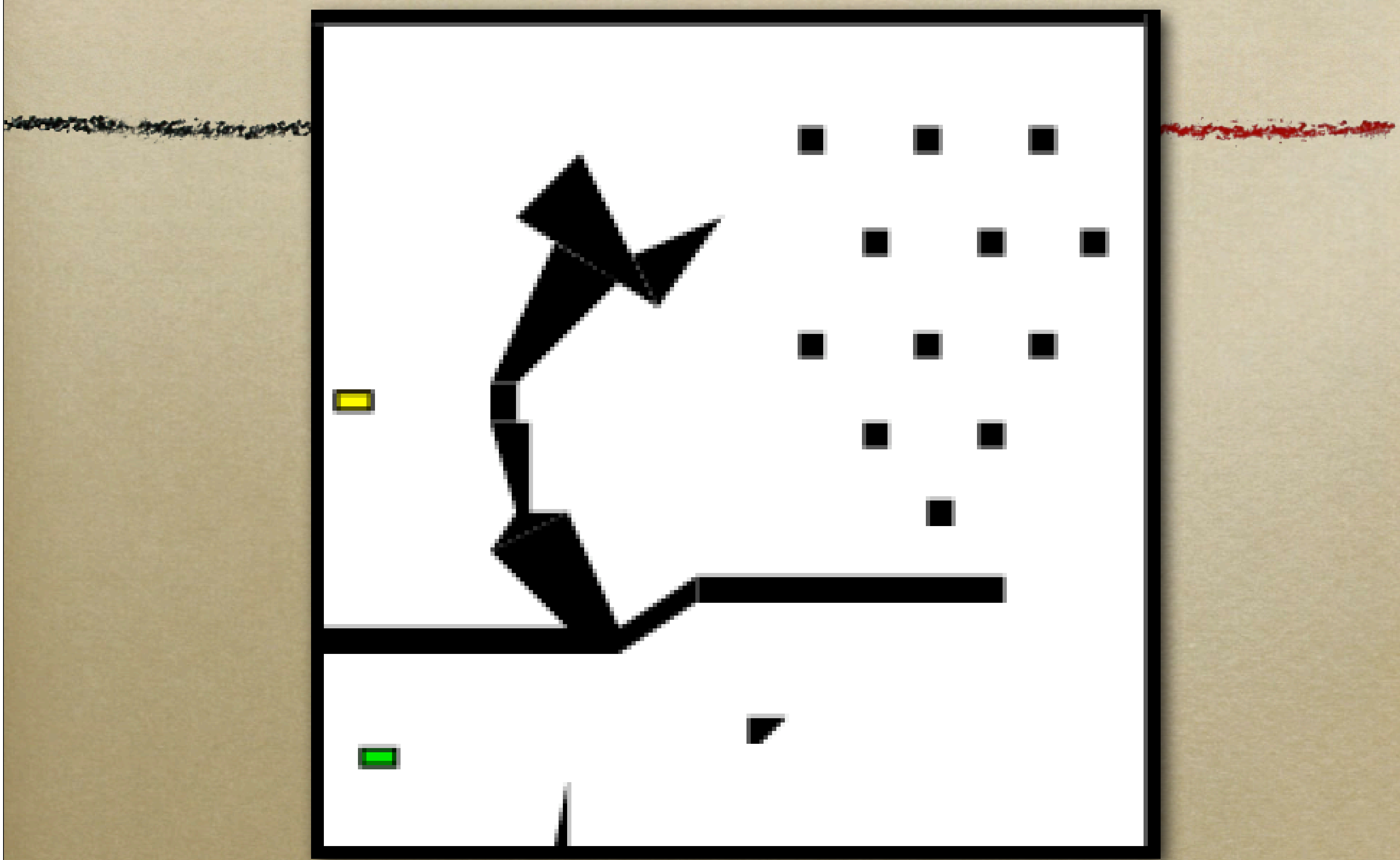
- *Tend to break up large Voronoi regions*
- *Limiting distribution of vertices is RANDOM_CONFIG*
    - *Key idea in proof: as RRT grows, probability that q_{rand} is reachable with local controller (and so immediately becomes a new vertex) approaches 1*

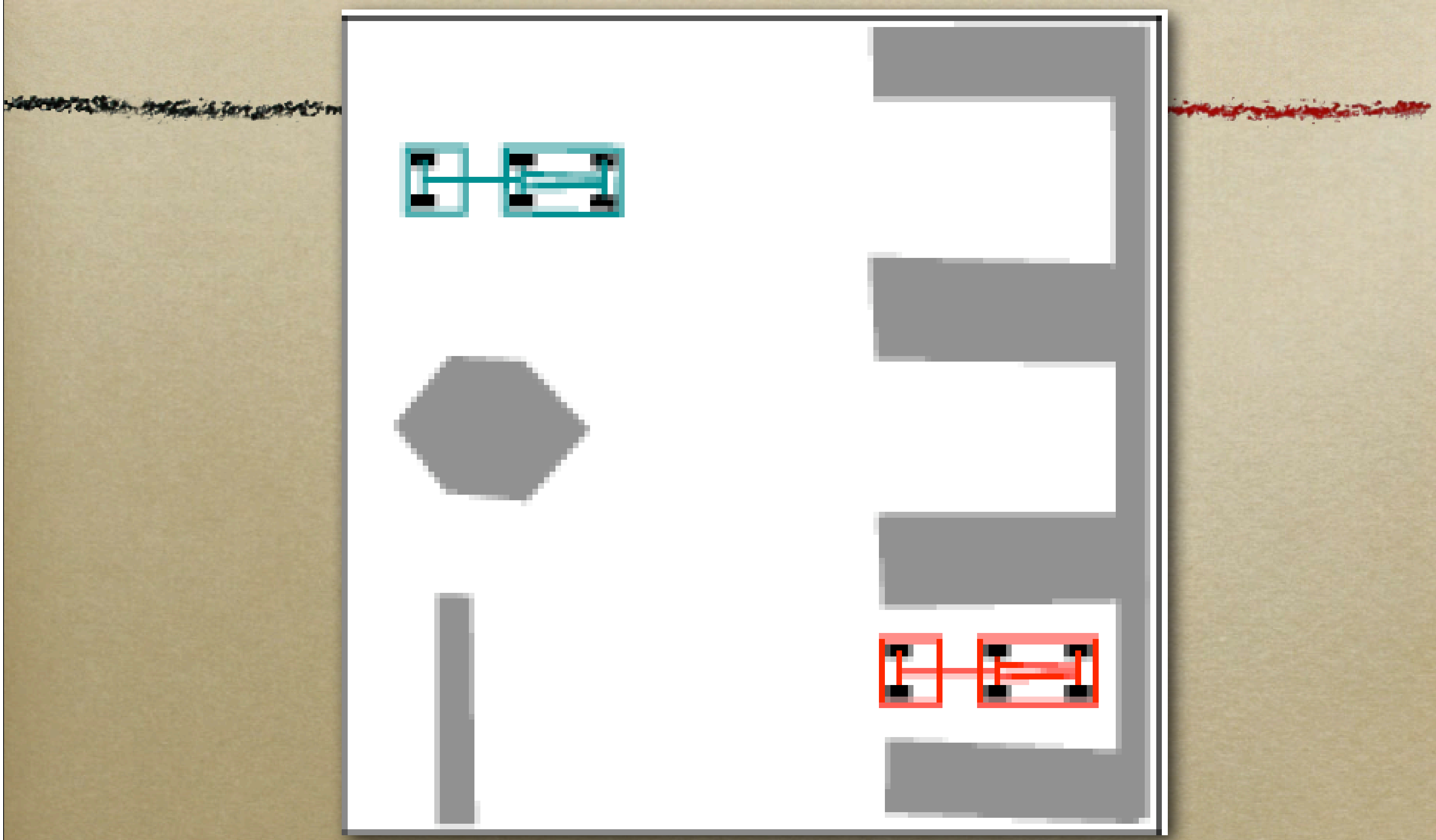
Planning with RRTs

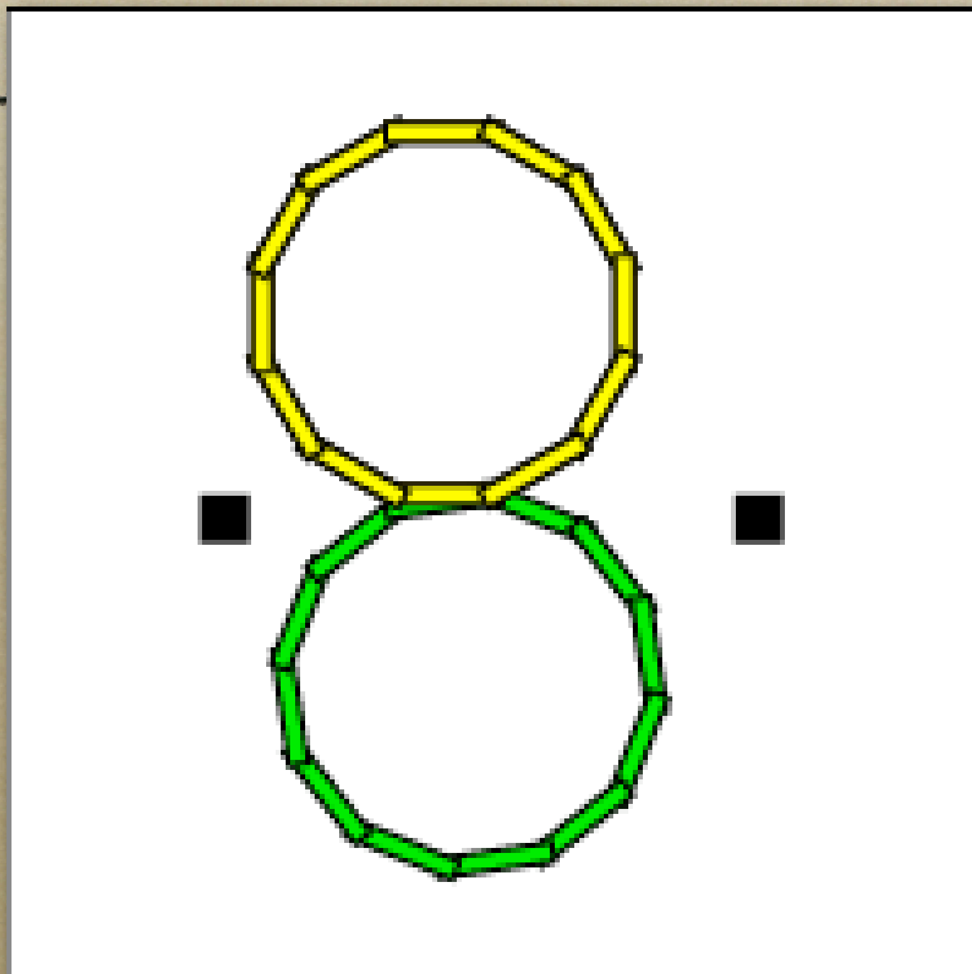
- *Build RRT from start until we add a node that can reach goal using local controller*
- *(Unique) path: root → last node → goal*
- *Optional: cross-link tree by testing local controller, search within tree using A**
- *Optional: grow forward and backward*











What you should know

- *C-space*
- *Ways of splitting up C-space*
  - *Visibility graph*
  - *Voronoi*
  - *Exact, approximate cell decomposition*
  - *Variable resolution or adaptive cells (quadtree, parti-game)*
- *RRTs*