15-780: Grad AI
Lecture 15: Planning

Geoff Gordon (this lecture)
Tuomas Sandholm
TAs Erik Zawadzki, Abe Othman

Review

- Planning algorithms
  - reduce to FOL (complications)
  - or use subset of FOL (e.g., STRIPS)
    - linear planner: add op to end of plan
    - partial-order planner (operators, bindings, partial order, guards, open preconditions): resolve open precond
- STRIPS: (world) state, operator = { preconditions } + { effects }, variable binding, goals

Plan Graphs

Planning & model search

- For a long time, it was thought that SAT-style model search was a non-starter as a planning algorithm
- More recently, people have written fast planners that
  - propositionalize the domain
  - turn it into a CSP or SAT problem
  - search for a model

Plan graph

- Tool for making good CSPs: plan graph
- Encodes a subset of the constraints that plans must satisfy
- Remaining constraints are handled
  - during search (reject solutions that violate them)—needs special-purpose code
  - or by adding extra clauses/constraints

Example

- Start state: have(Cake)
- Goal: have(Cake) $\wedge$ eaten(Cake)
- Operators: bake, eat

- Bake
  - pre: $\neg$have(Cake)
  - post: have(Cake)
- Eat
  - pre: have(Cake)
  - post: $\neg$have(Cake), eaten(Cake)

Propositionalizing

- Note: this domain is fully propositional
- If we had a general STRIPS domain, would have to pick a universe and propositionalize
- E.g., eat(x) would become eat(Banana), eat(Cake), eat(Fred), …

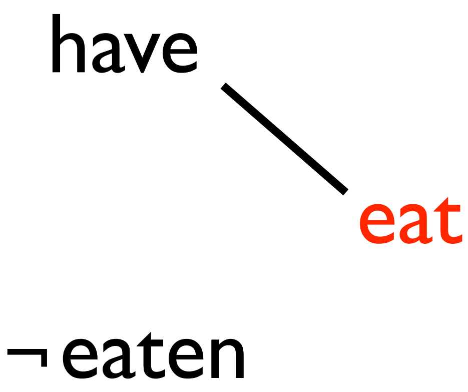
Plan graph

have

¬ eaten

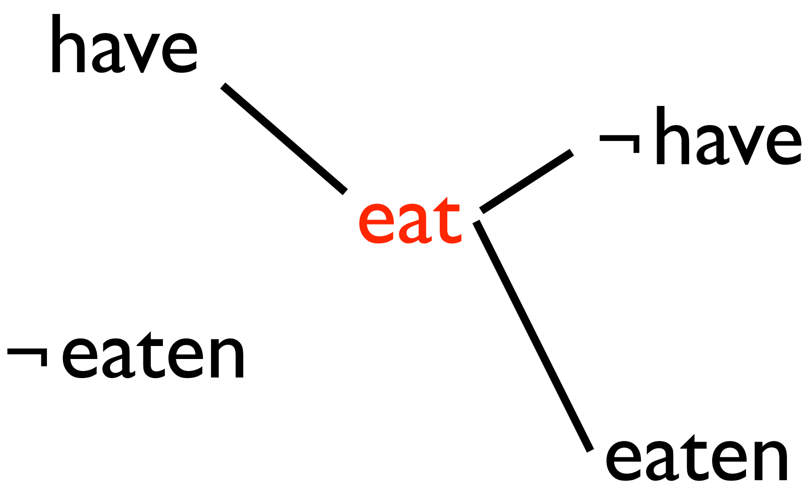
- Alternating levels: states and actions
- First level: initial state

Plan graph



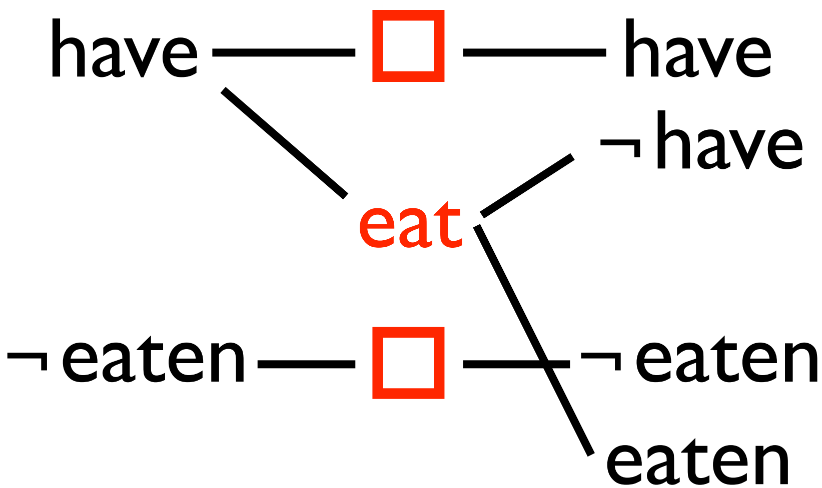
- First action level: all applicable actions
- Linked to their preconditions

Plan graph



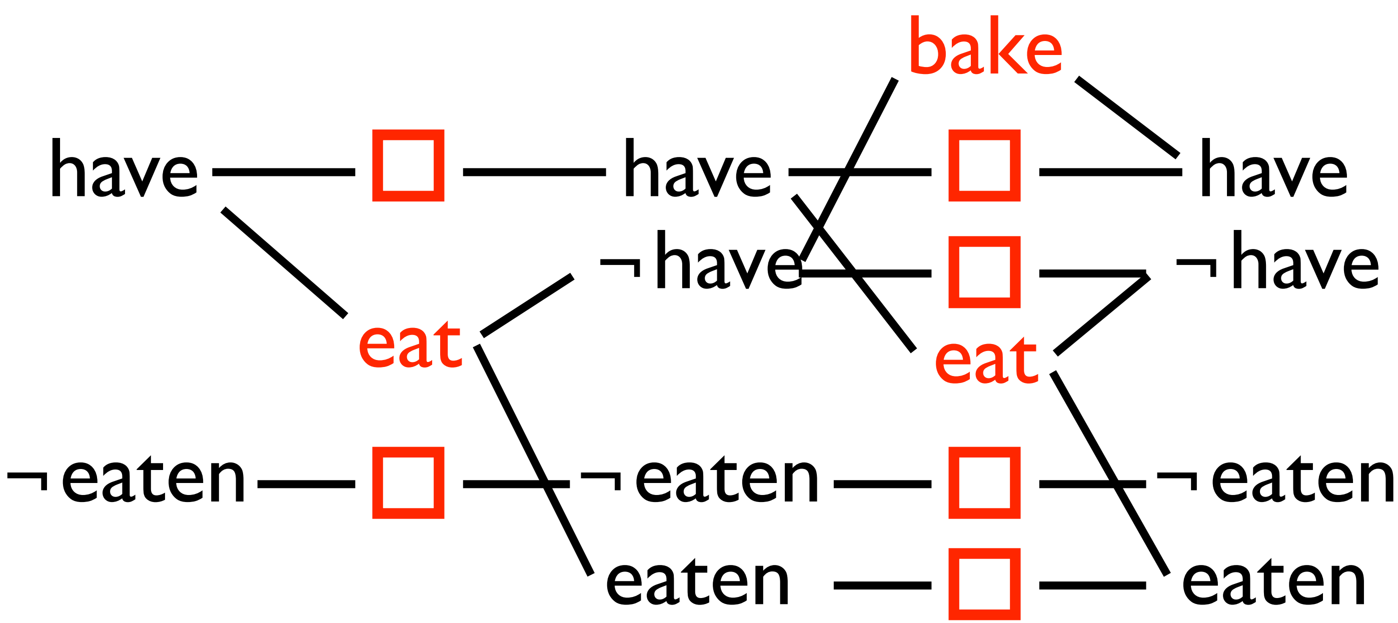
- Second state level: add effects of actions to get literals that could hold at step 2

Plan graph



- Also add **maintenance actions** to represent effect of doing nothing

Plan graph

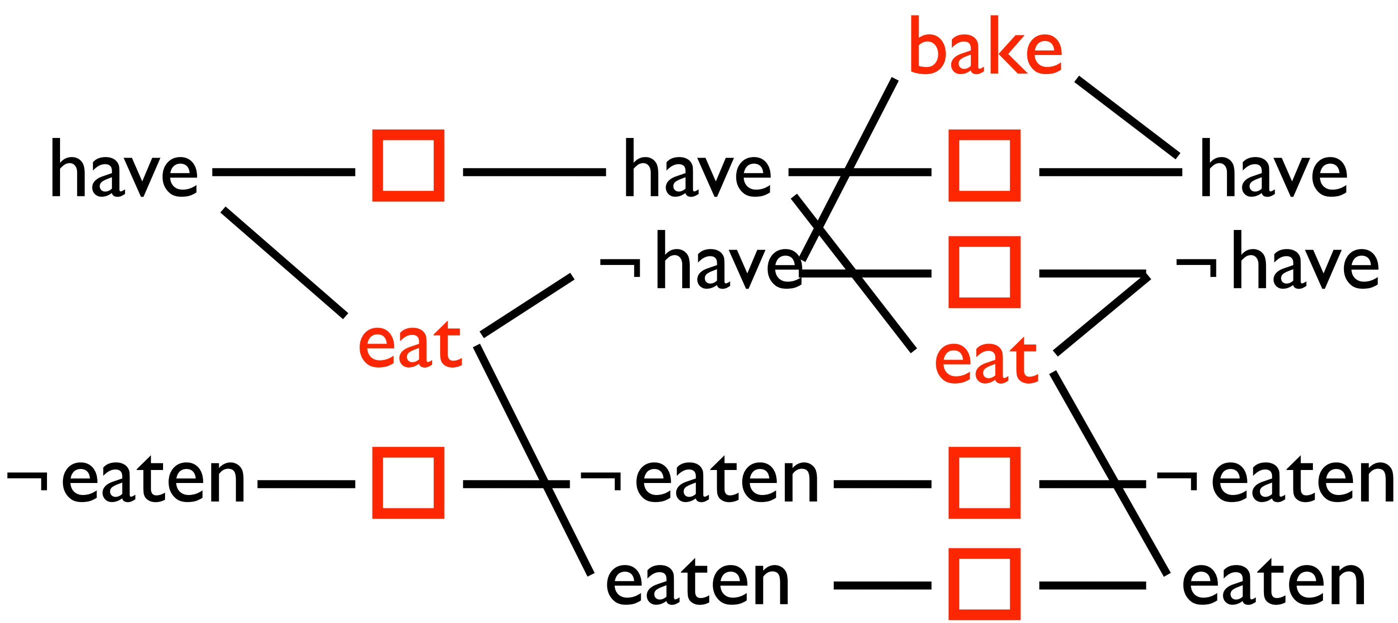


- Extend another pair of levels: now bake is a possible action

Plan graph

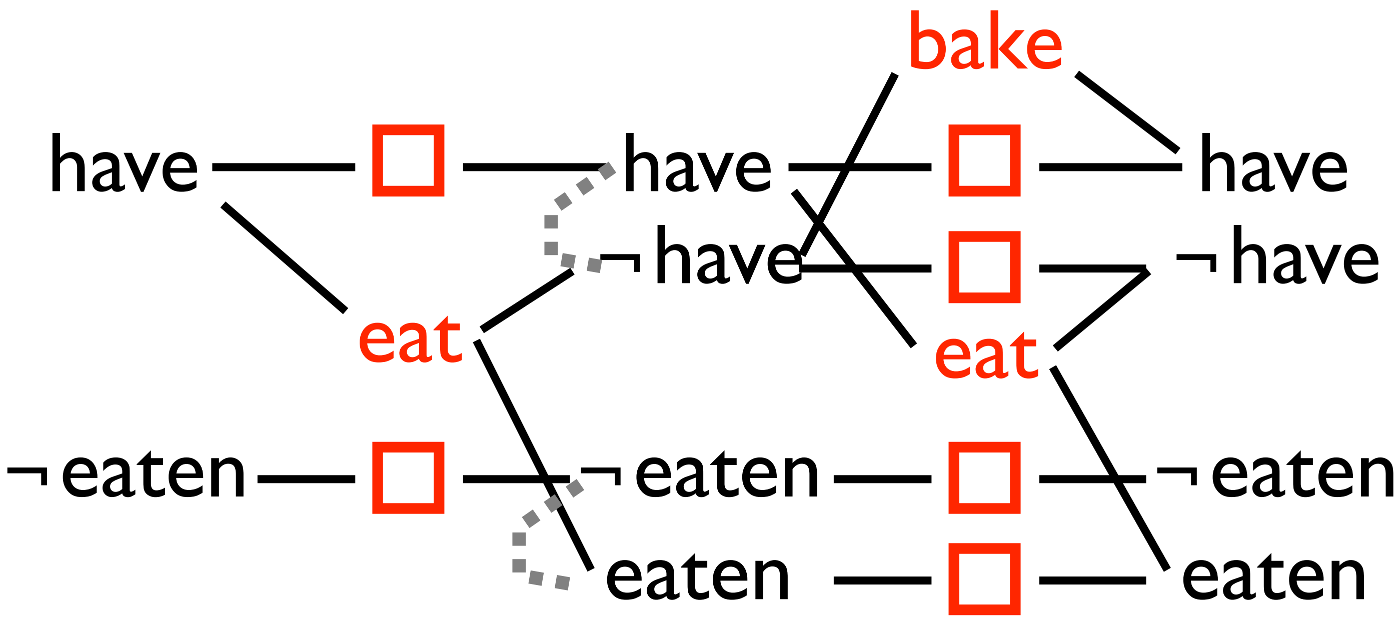
- Can extend as far right as we want
- Plan = subset of the actions at each action level
- Ordering unspecified within a level

Plan graph



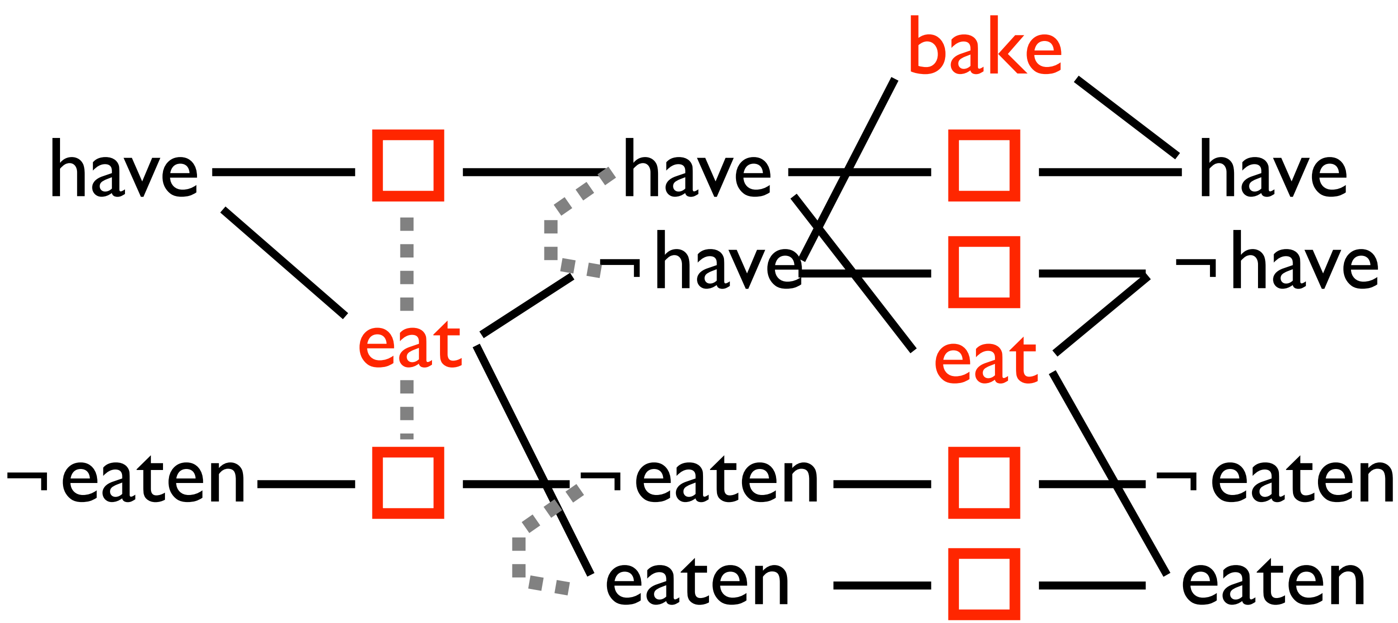
- In addition to the above links, add ***mutex*** links to indicate mutually exclusive actions or literals

Plan graph



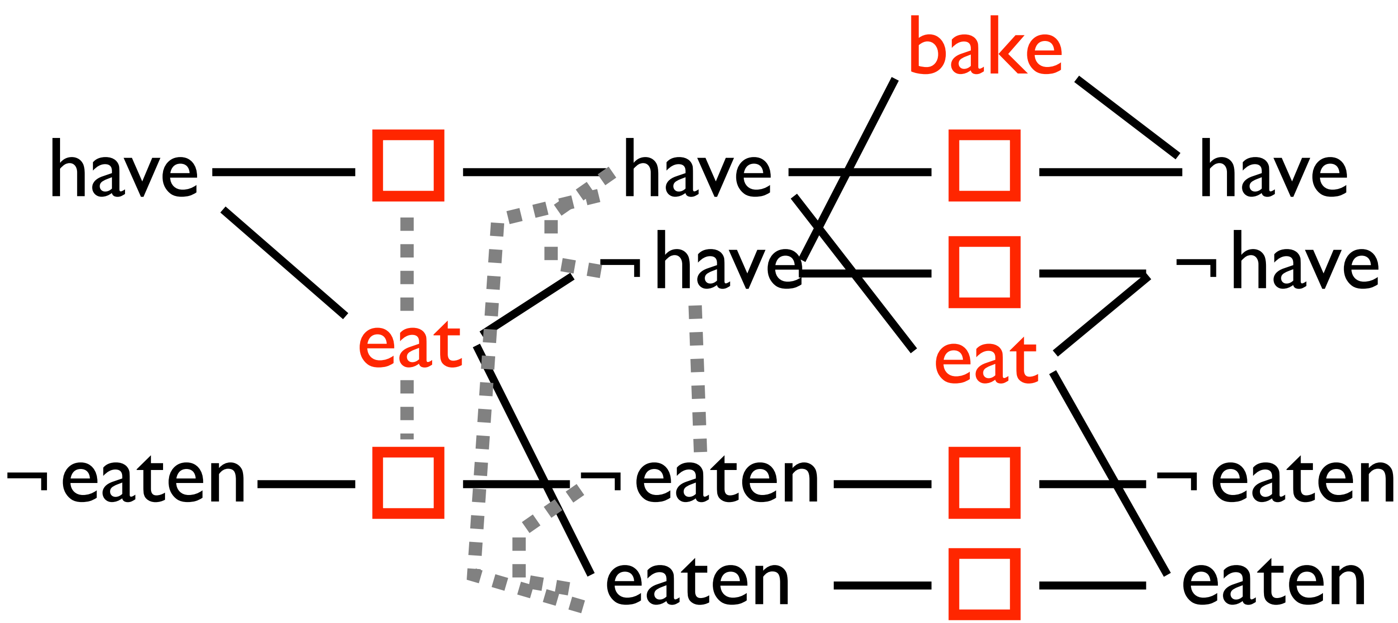
- Literals are mutex if they are contradictory

Plan graph



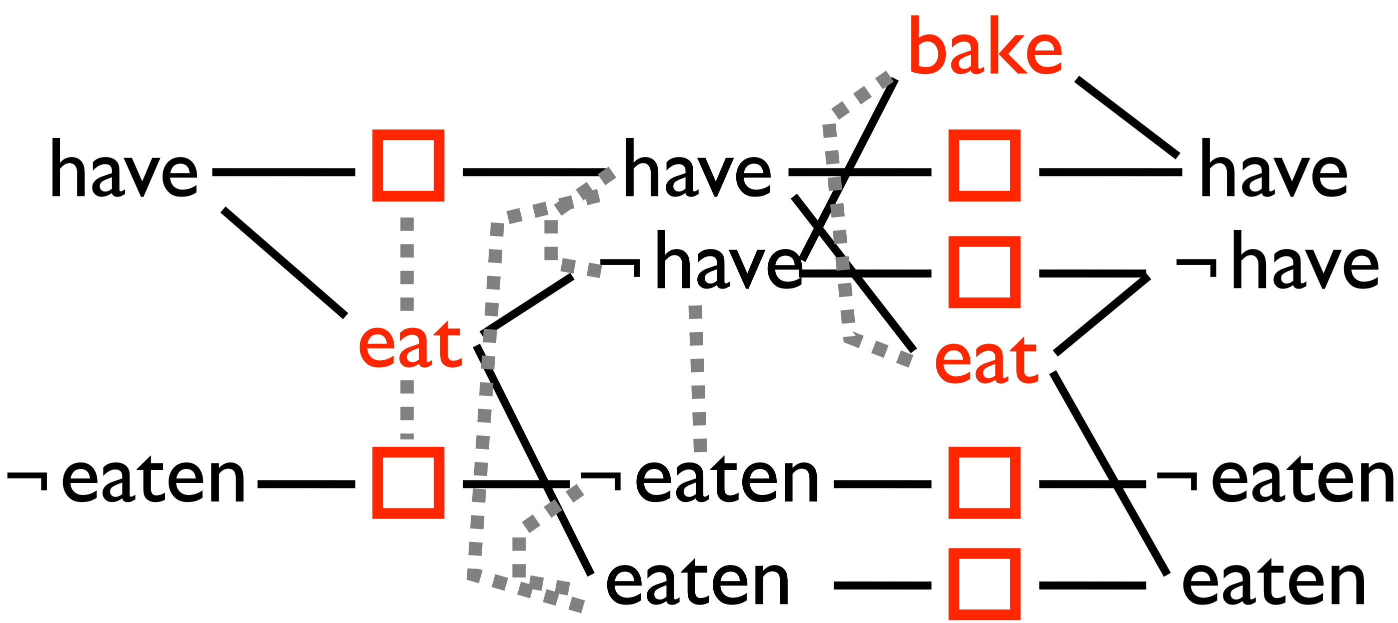
- Actions which assert contradictory literals are mutex (***inconsistent effects***)

Plan graph



- Literals are also mutex if there is no action or non-mutex pair of actions that could achieve both (***inconsistent support***)

Plan graph



- Actions are also mutex if one deletes a precondition of other (***interference***), or if preconditions are mutex (***competition***)

Mutex summary

- For each action level, left to right, check pairs of actions A, B (each check linear in rep'n size):
  - inconsistent effects: check each effect of A vs. effects of B
  - interference: effects of A vs. preconds of B
  - competing preconditions: check mutex links on preconditions of A, B
- Results at action level L tell us (in)consistent support at proposition level L+1

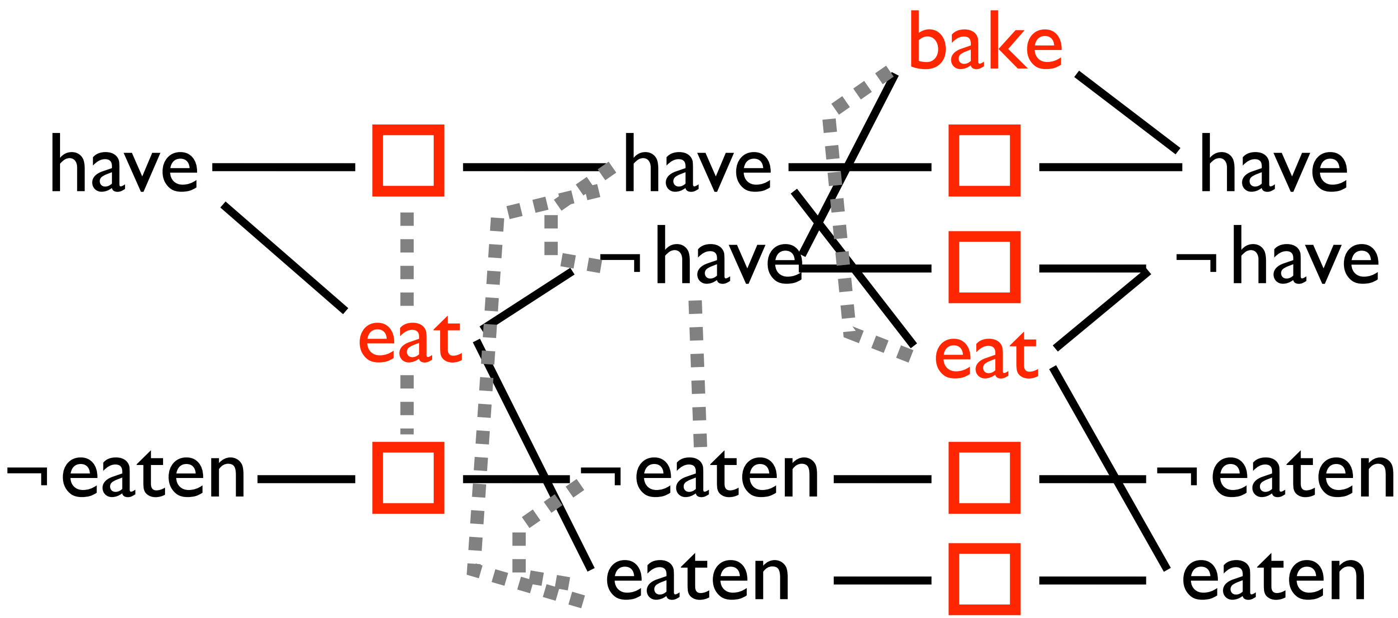
Getting a plan

- Build the plan graph out to some length k
- Search:
  - directly on the graph
  - or by translating to SAT or CSP
- If search succeeds, read off the plan
- If not, increment k and try again
- There is a test to see if k is "big enough"

Plan search

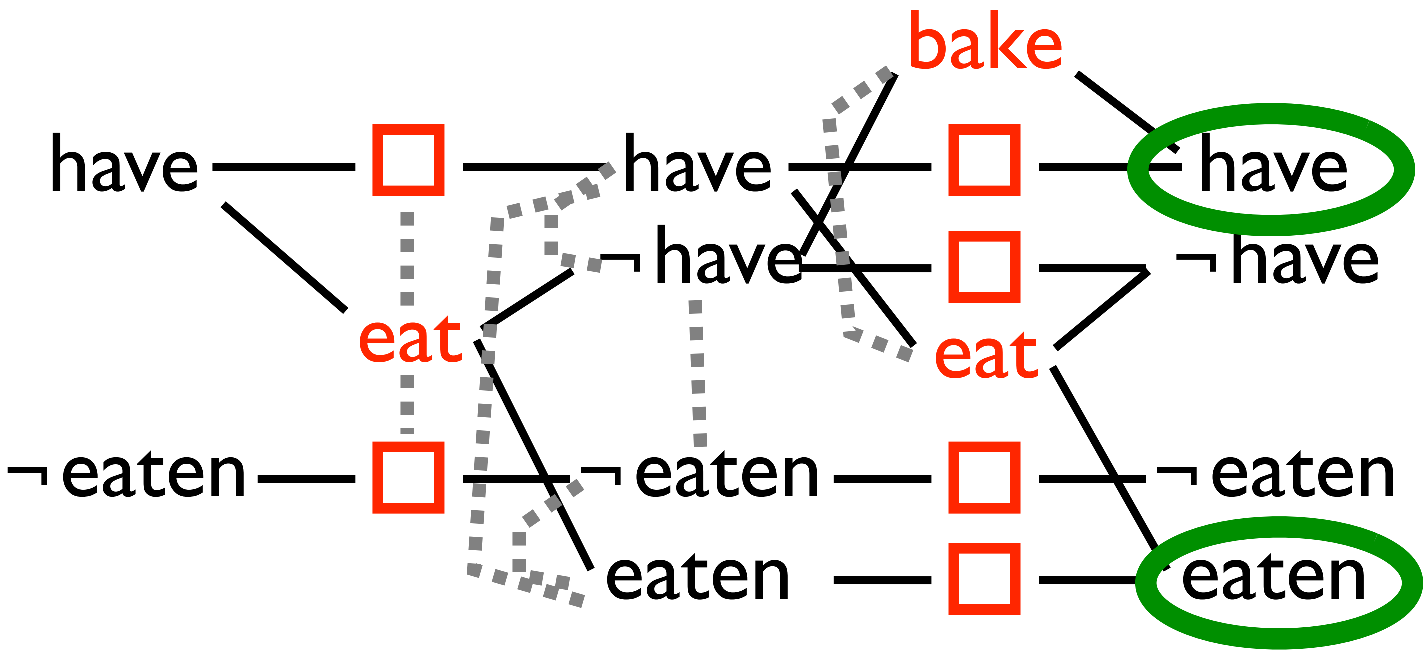
- DFS w/ variable ordering based on plan graph
- Start from last level, fill in last action set, compute necessary preconditions, fill in 2nd-to-last action set, etc.
- If at some level there is no way to do any actions, or no way to fill in consistent preconditions, backtrack

Plan search

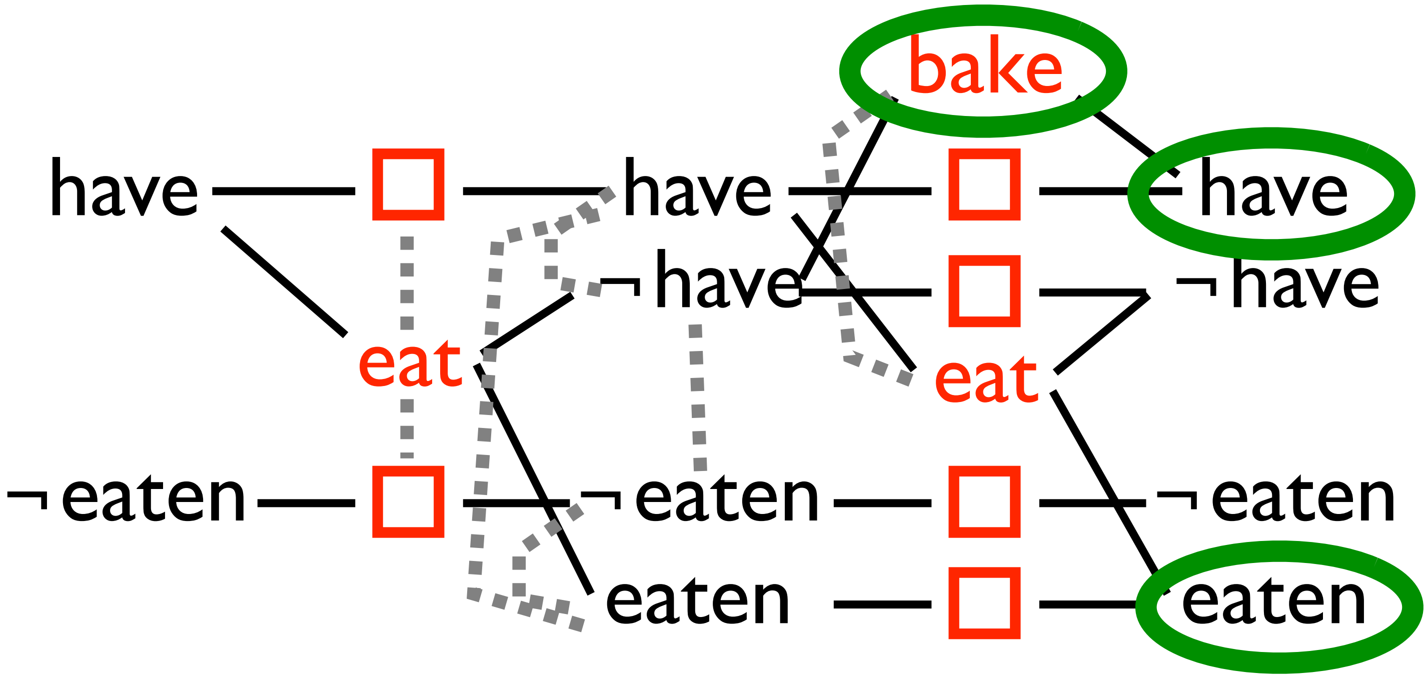


bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Plan search

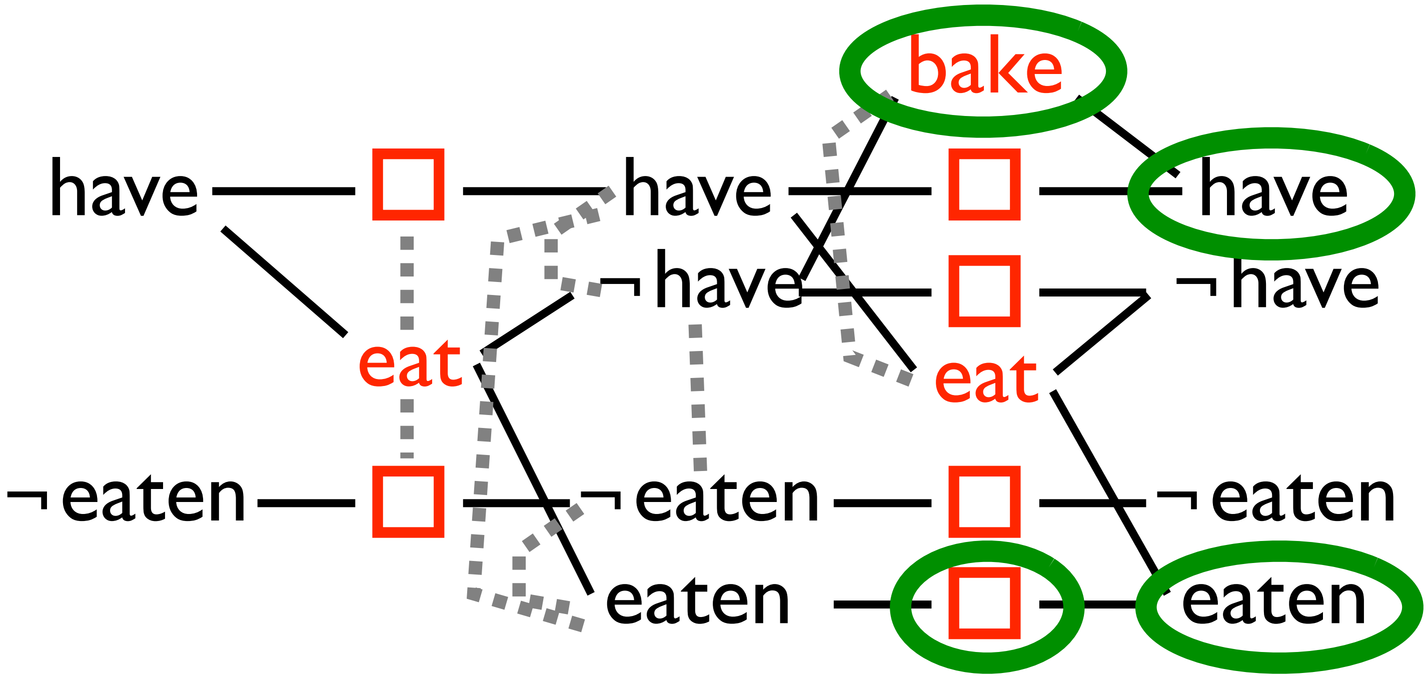

bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Plan search



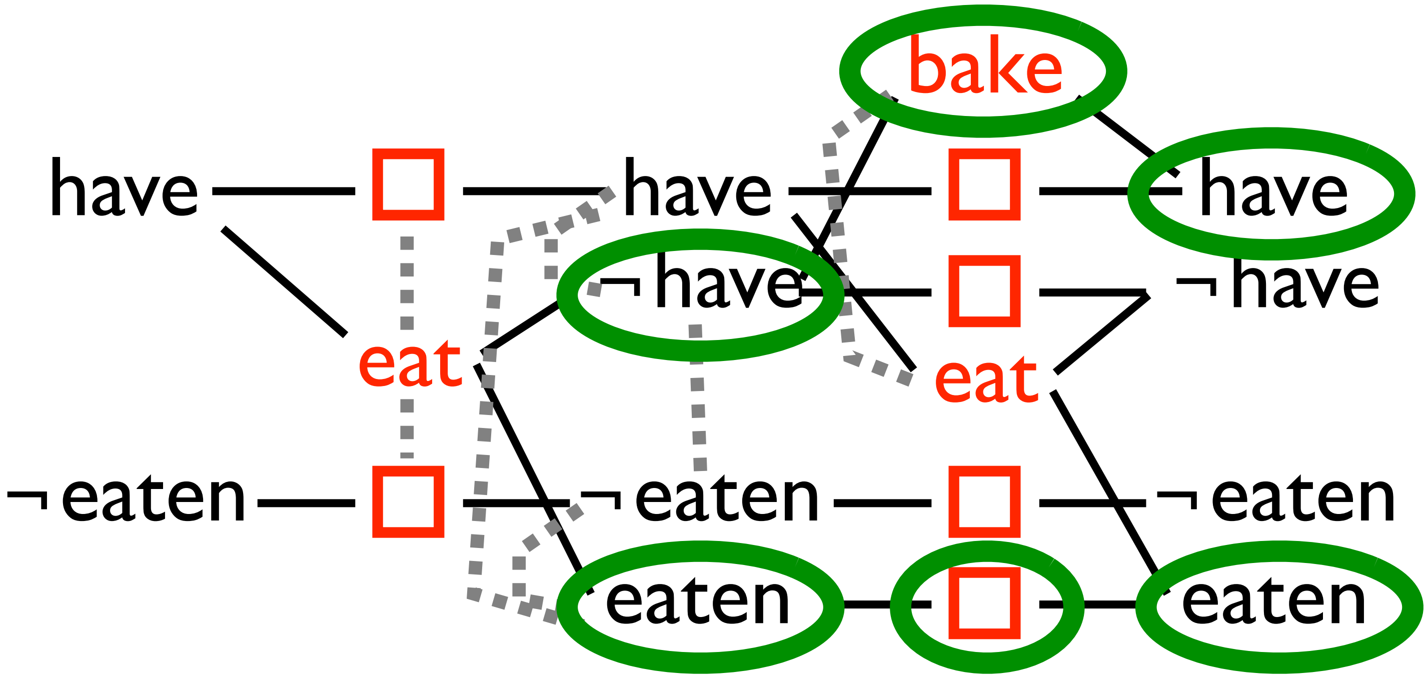
bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten
eaten

Plan search

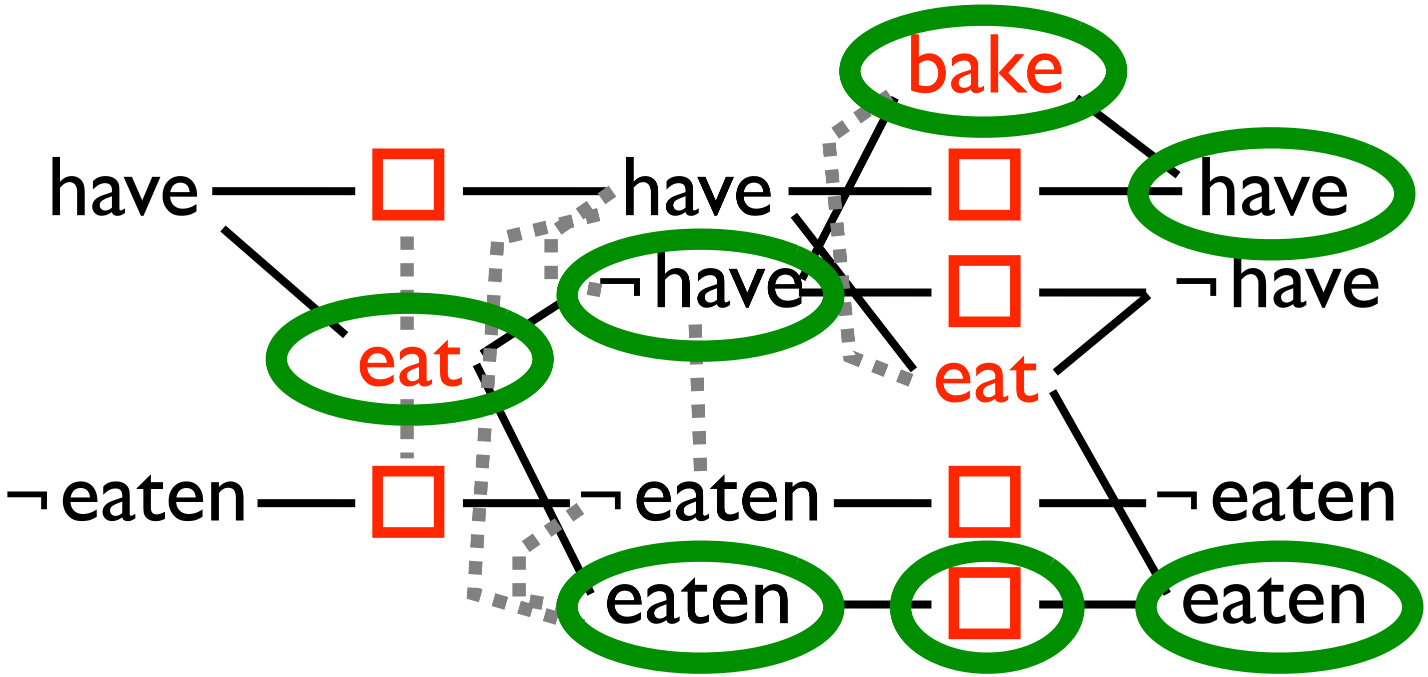


bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Plan search

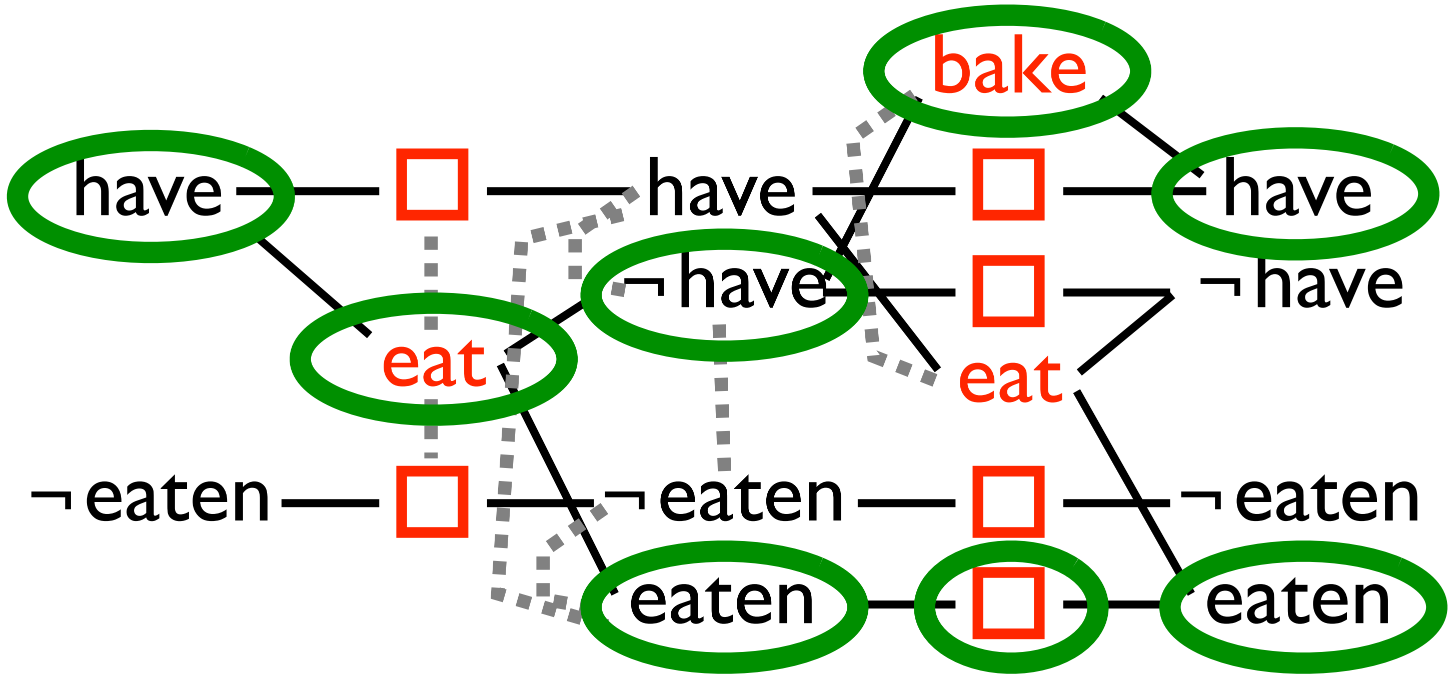

bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Plan search



bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Plan search



bake
have
have
have
¬ have
¬ have
eat
eat
¬ eaten
¬ eaten
¬ eaten
eaten
eaten

Translation to SAT

- One variable for each pair of literals in state levels
- One variable per action in action levels
- Constraints implement STRIPS semantics plus “hints”
- Solution tells us which actions are performed at each action level, which literals are true at each state level

Action constraints

- Each action can only be executed if all of its preconditions are present:

  $act_{t+1} \Rightarrow pre1_t \wedge pre2_t \wedge \ldots$

- If executed, action asserts its postconditions:

  $act_{t+1} \Rightarrow post1_{t+2} \wedge post2_{t+2} \wedge \ldots$

Literal constraints

- In order to achieve a literal, we must execute an action that achieves it
  - $post_{t+2} \Rightarrow act1_{t+1} \vee act2_{t+1} \vee \ldots$
- Might be a maintenance action

Initial & goal constraints

- Goals must be satisfied at end:

  $goal1_T \wedge goal2_T \wedge \ldots$

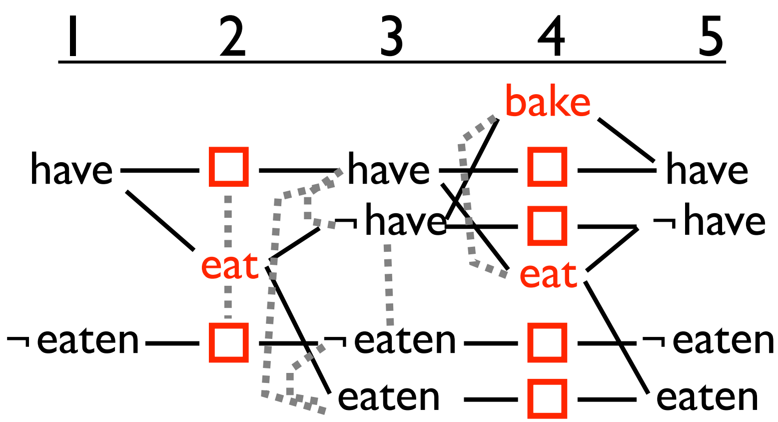
- And initial state holds at beginning:

  $init1_I \wedge init2_I \wedge \ldots$

Mutex constraints

- Mutex constraints between actions or literals: add clause $(\neg x \vee \neg y)$
- Mutexes are redundant, but help anyway

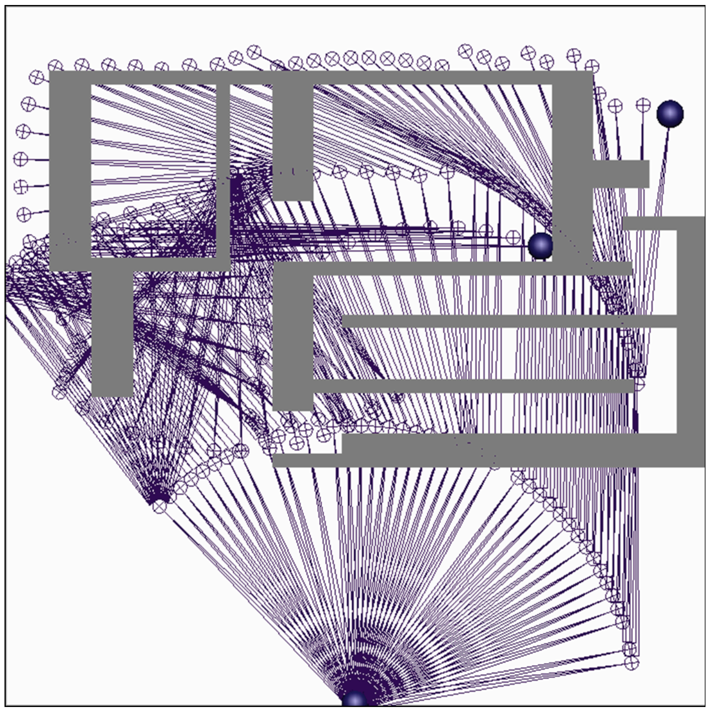
Translation to SAT: example



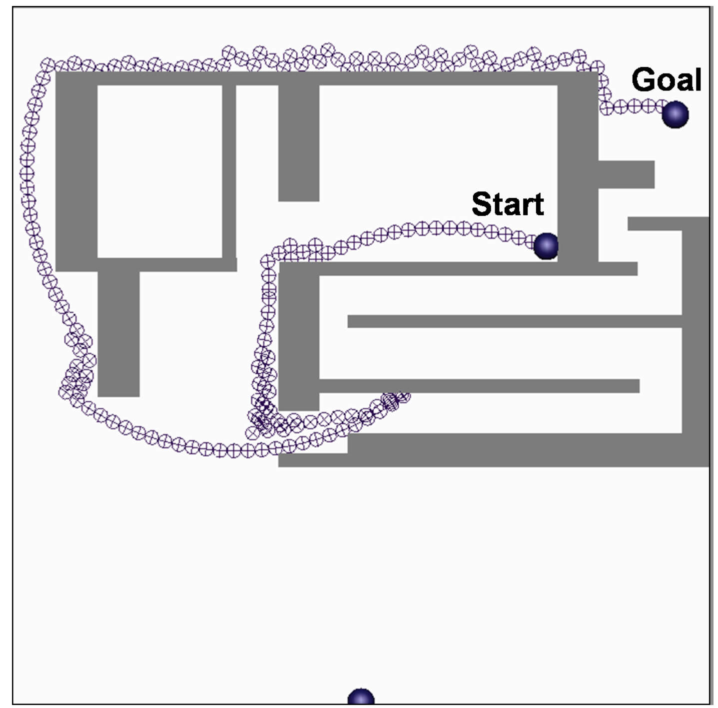
note: haven't drawn all mutexes at levels 4 & 5

Spatial Planning

Plans in Space…



Optimal Solution



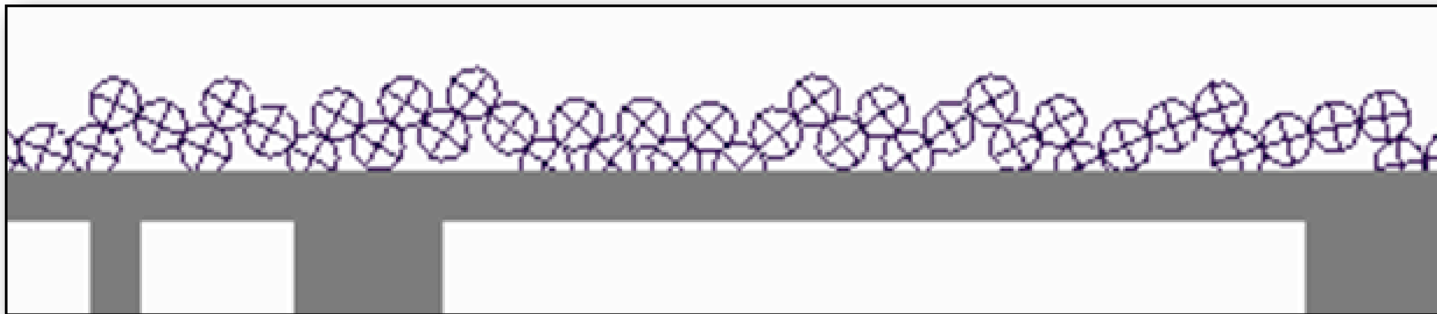
End-effector Trajectory

- A* can be used for many things
- Here, A* for spatial planning (in contrast to, e.g., jobshop scheduling)

What's wrong w/ A*?

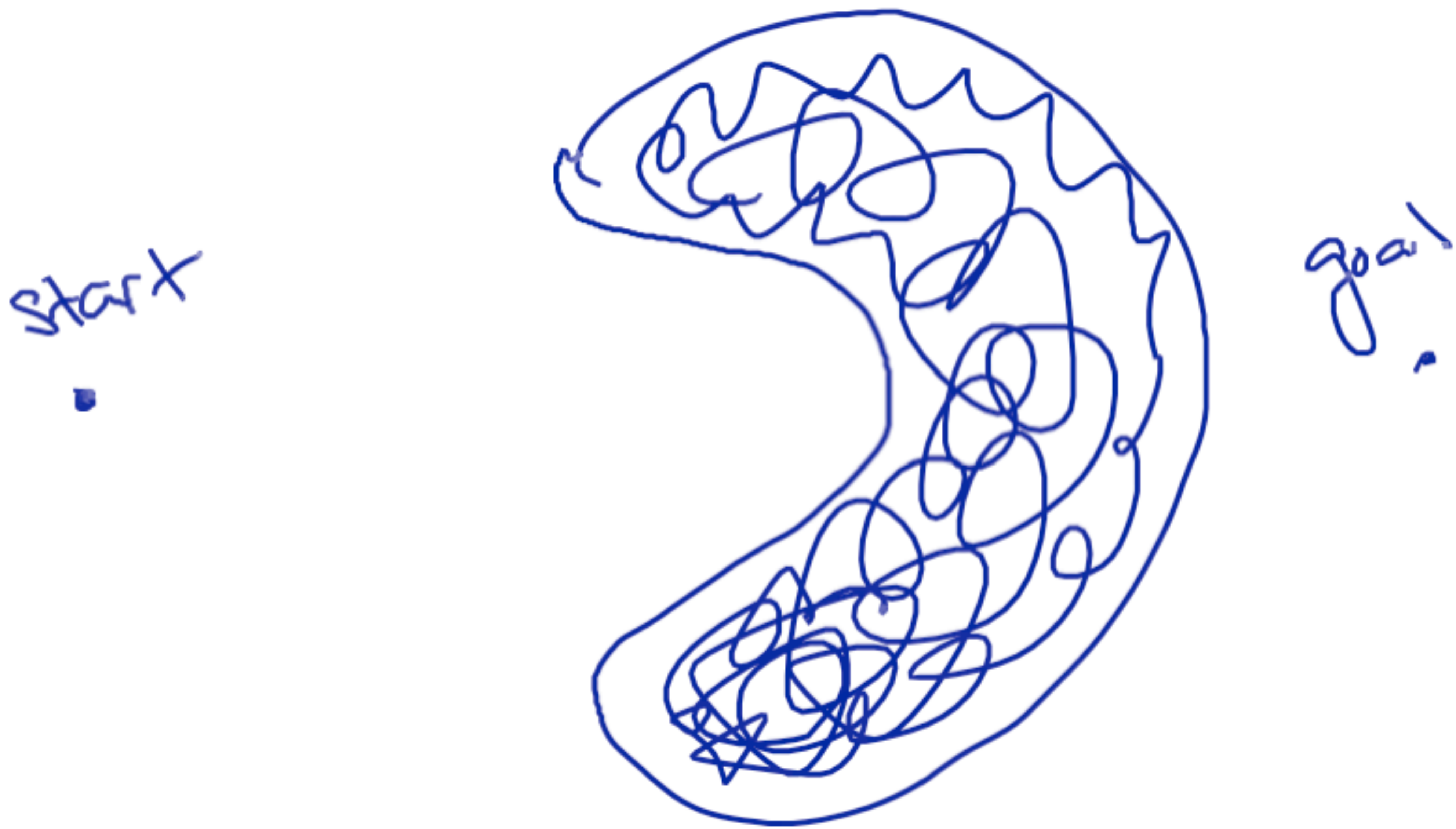
- A* guarantees:
  - ***(optimality)*** A* finds a solution of cost g*
  - ***(efficiency)*** A* expands no nodes that have f(node) > g*

What's wrong with A*?



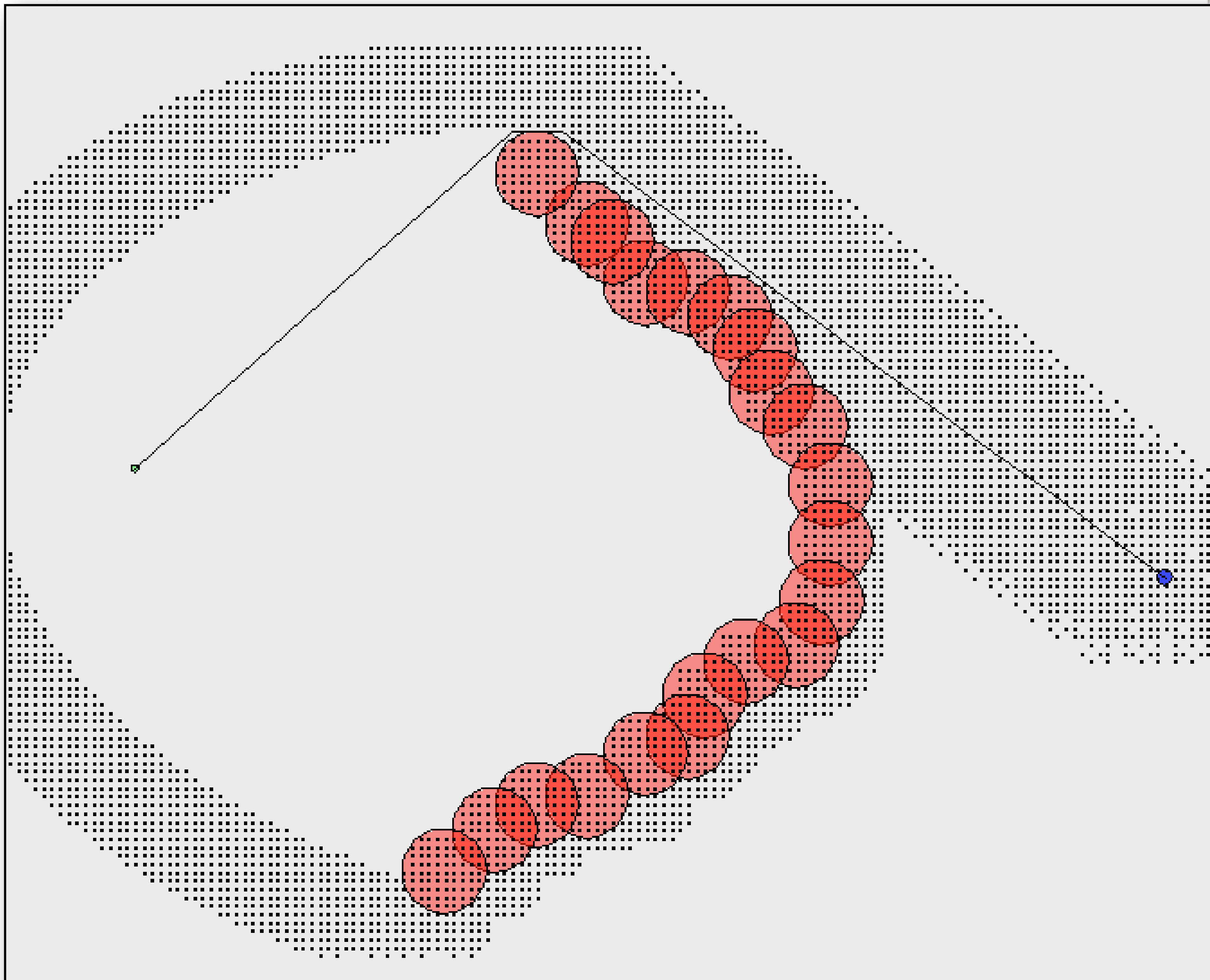
- Discretized space into tiny little chunks
  - a few degrees rotation of a joint
  - ***Lots*** of states ⇒ lots of states w/ $f \leq g^*$
- Discretized actions too
  - one joint at a time, discrete angles
- Results in jagged paths

What's wrong with A*?

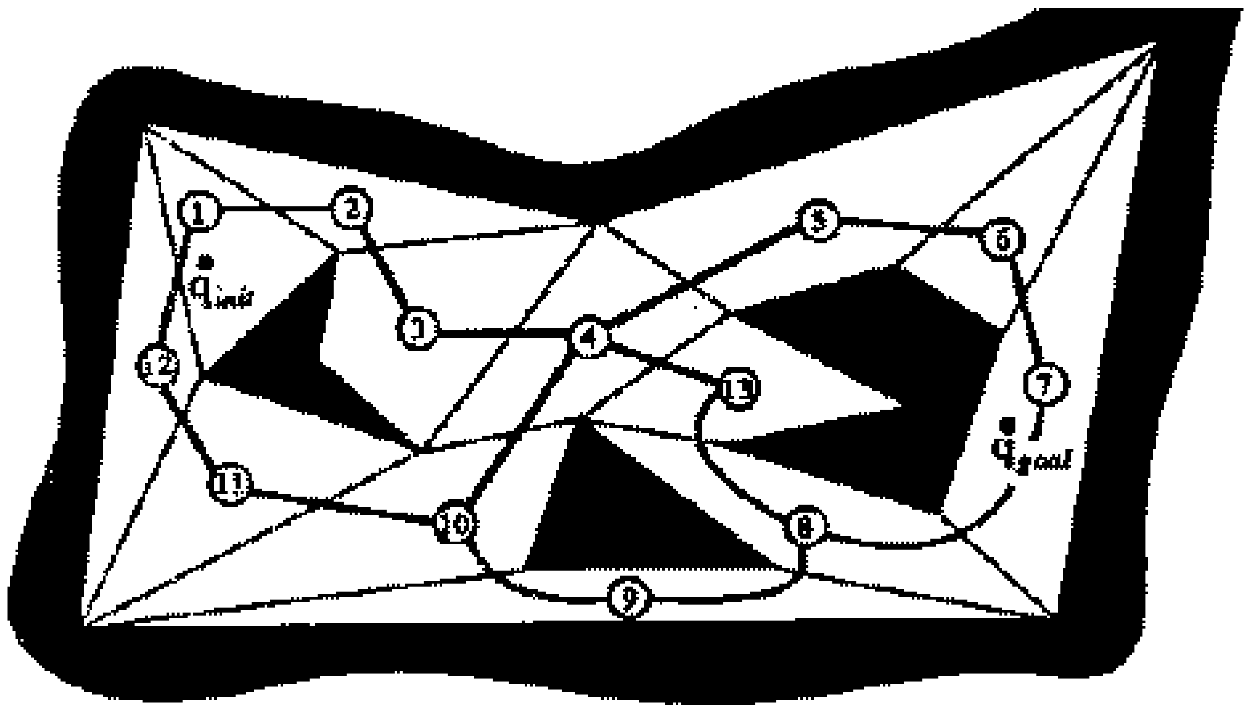


Snapshot of A*

http://www.cs.cmu.edu/~ggordon/PathPlan/



Wouldn't it be nice…

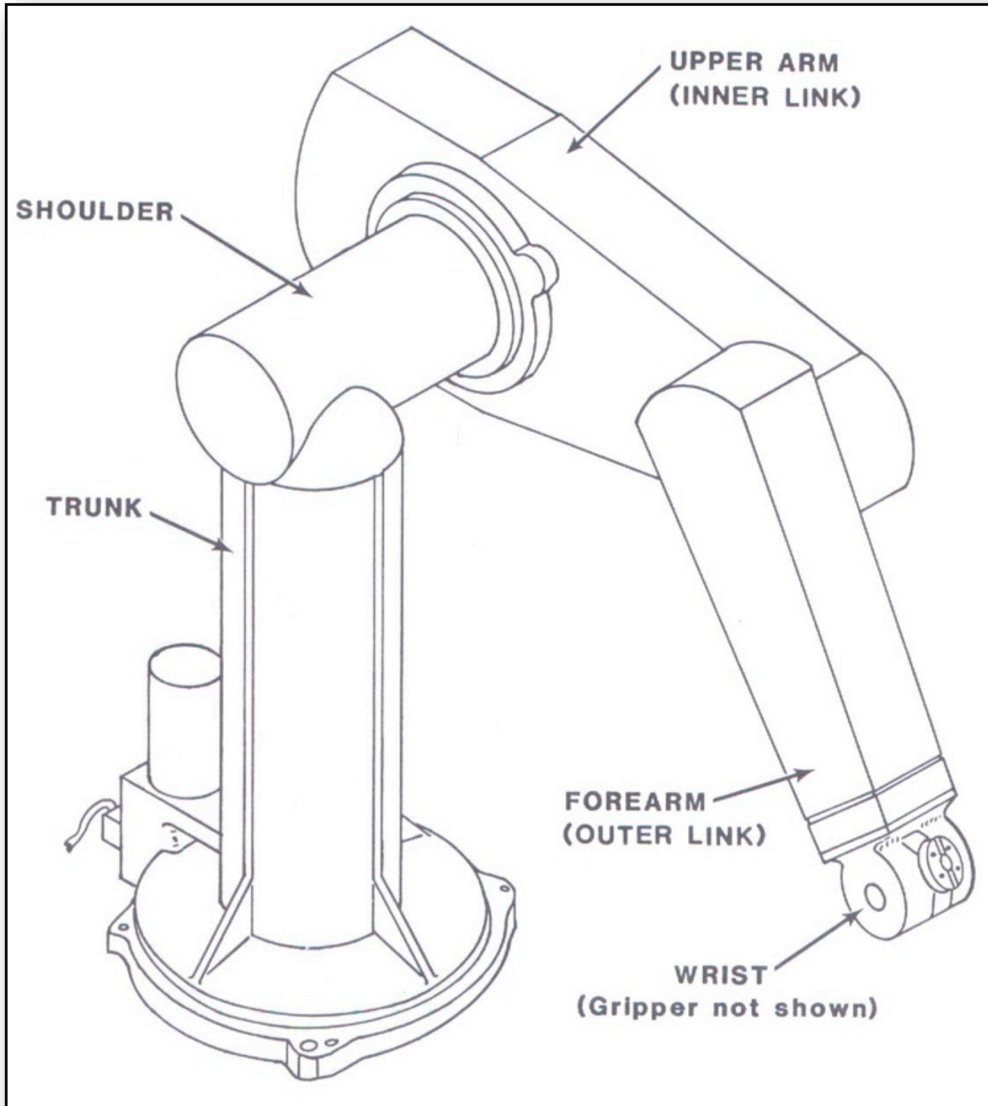


- … if we could break things up based more on the real geometry of the world?
- *Robot Motion Planning*, Jean-Claude Latombe

Physical system

- 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

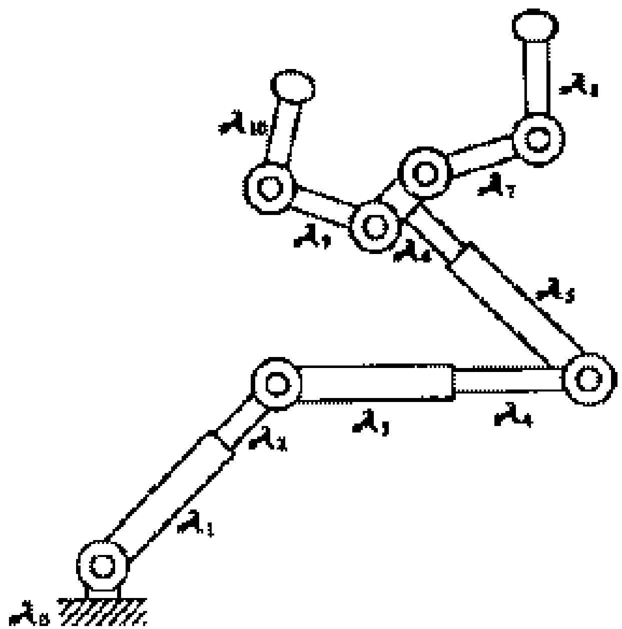


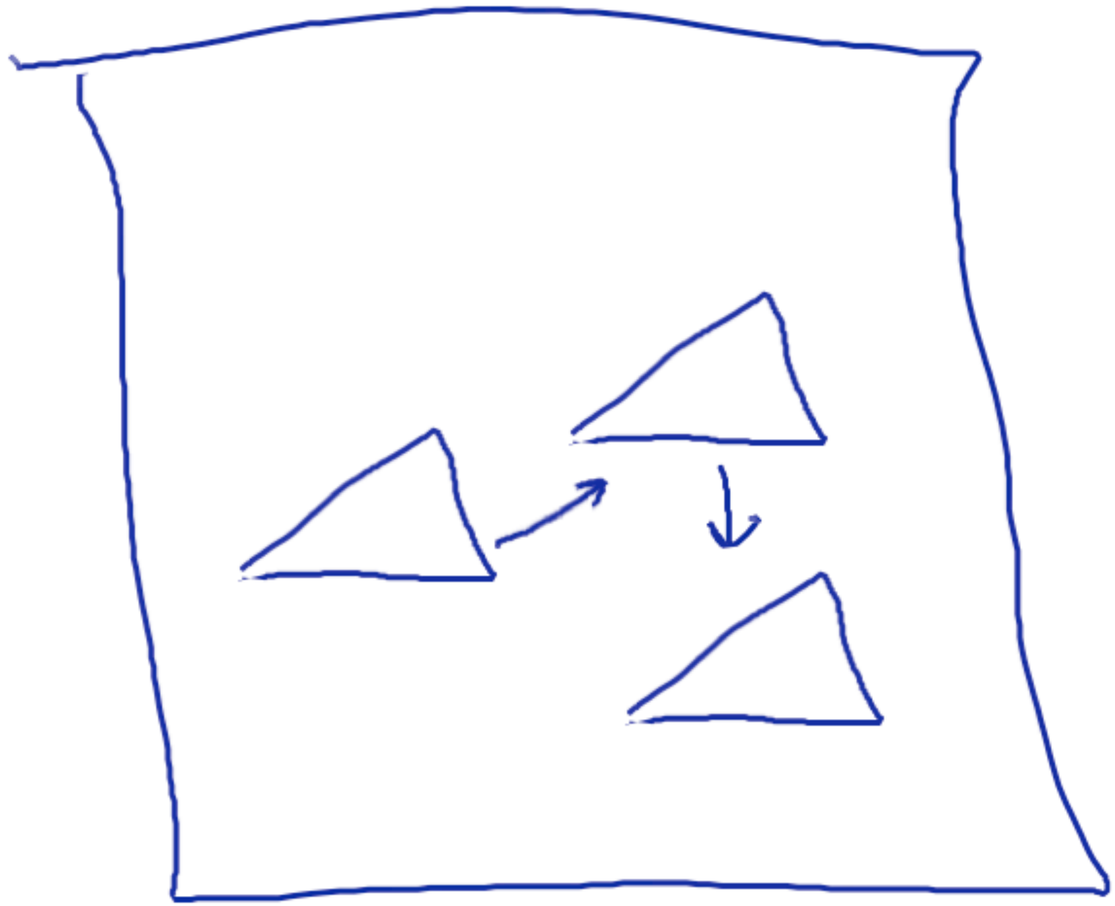
Fig. 11. Structure of the 10-DOF manipulator.

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

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

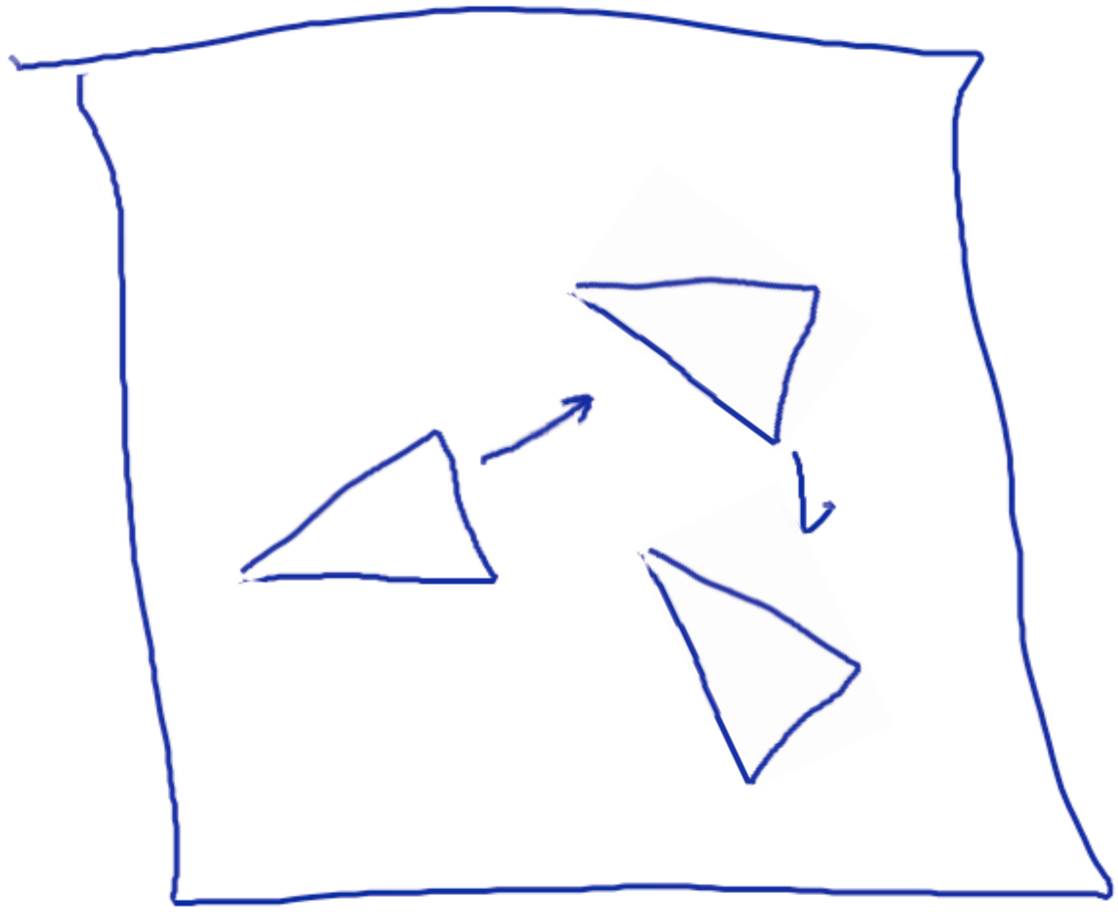
  q_i = angle or length of joint i
- Dimension of $\mathbf{q}$ = “degrees of freedom”

Mobile robots



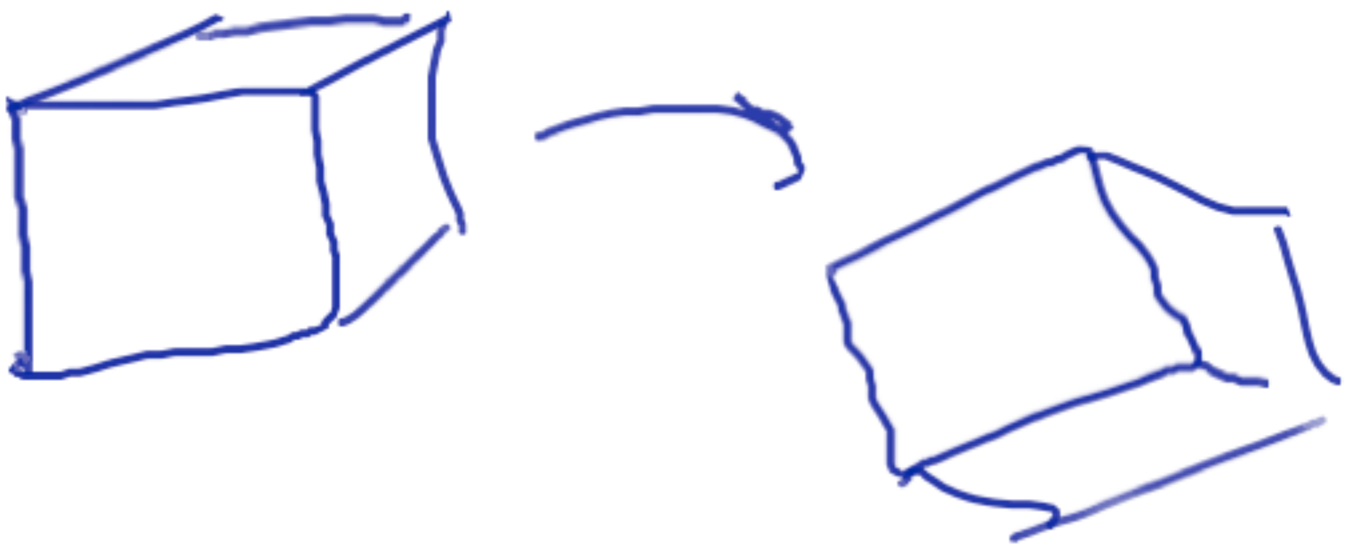
- Translating in space = 2 dof

More mobility

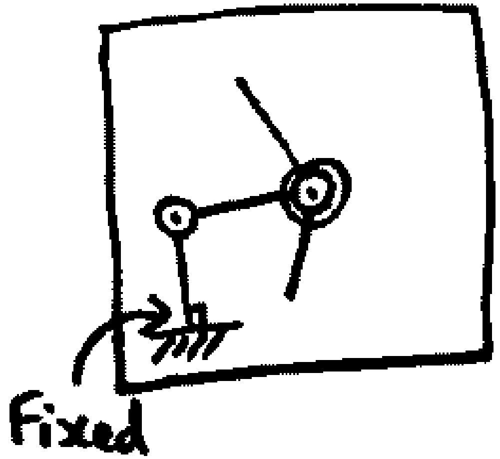


- Translation + rotation = 3 dof

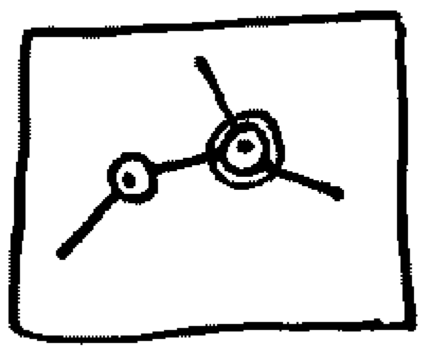
Q: How many dofs?



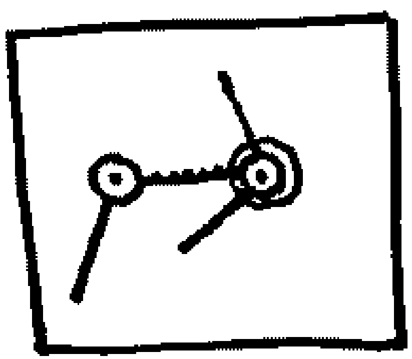
- 3d translation & rotation



How many dofs?



Free flying
How many dofs?

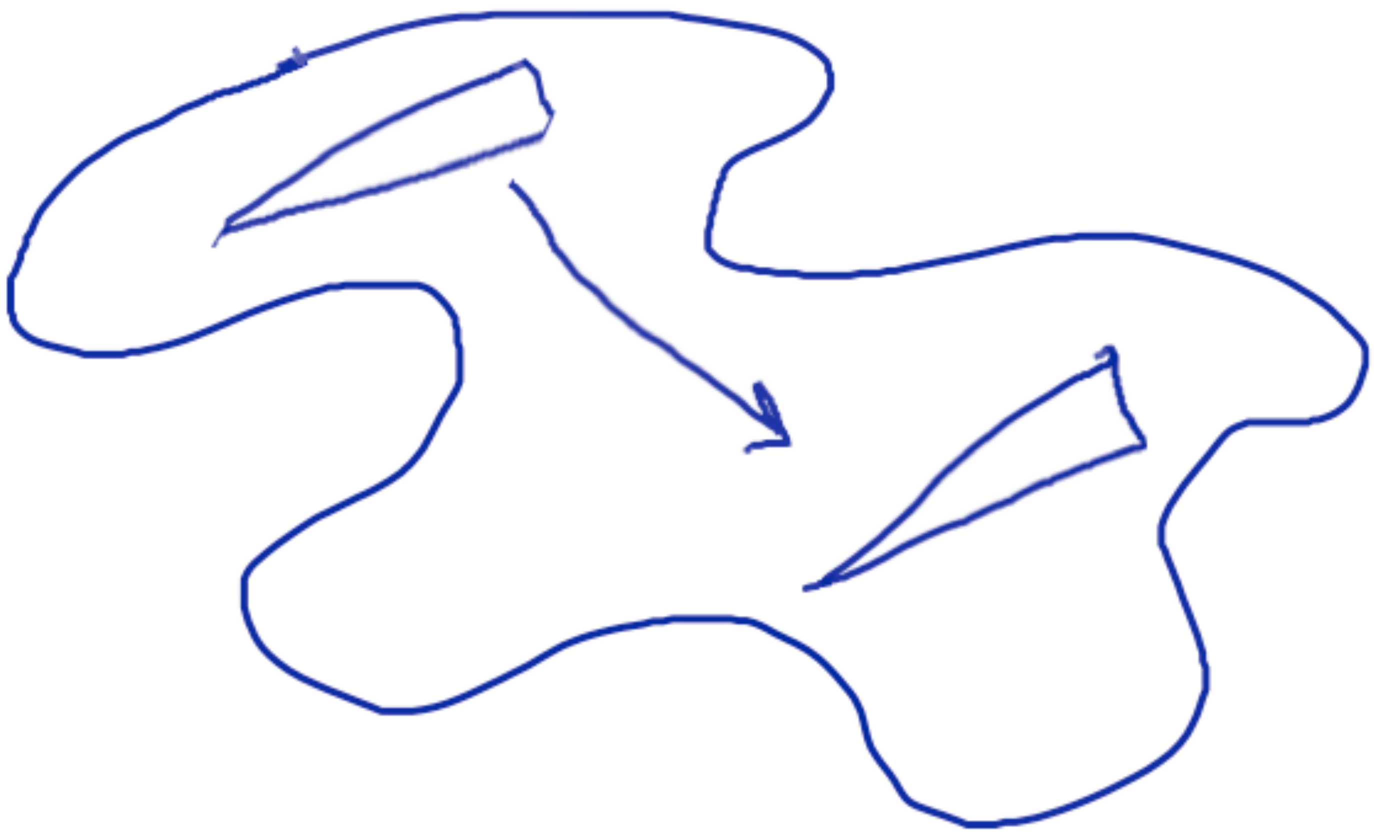


Midline ------ must always be horizontal.
How many DOFs?

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

credit: Andrew Moore

Kinematic motion planning



- Now let's add obstacles

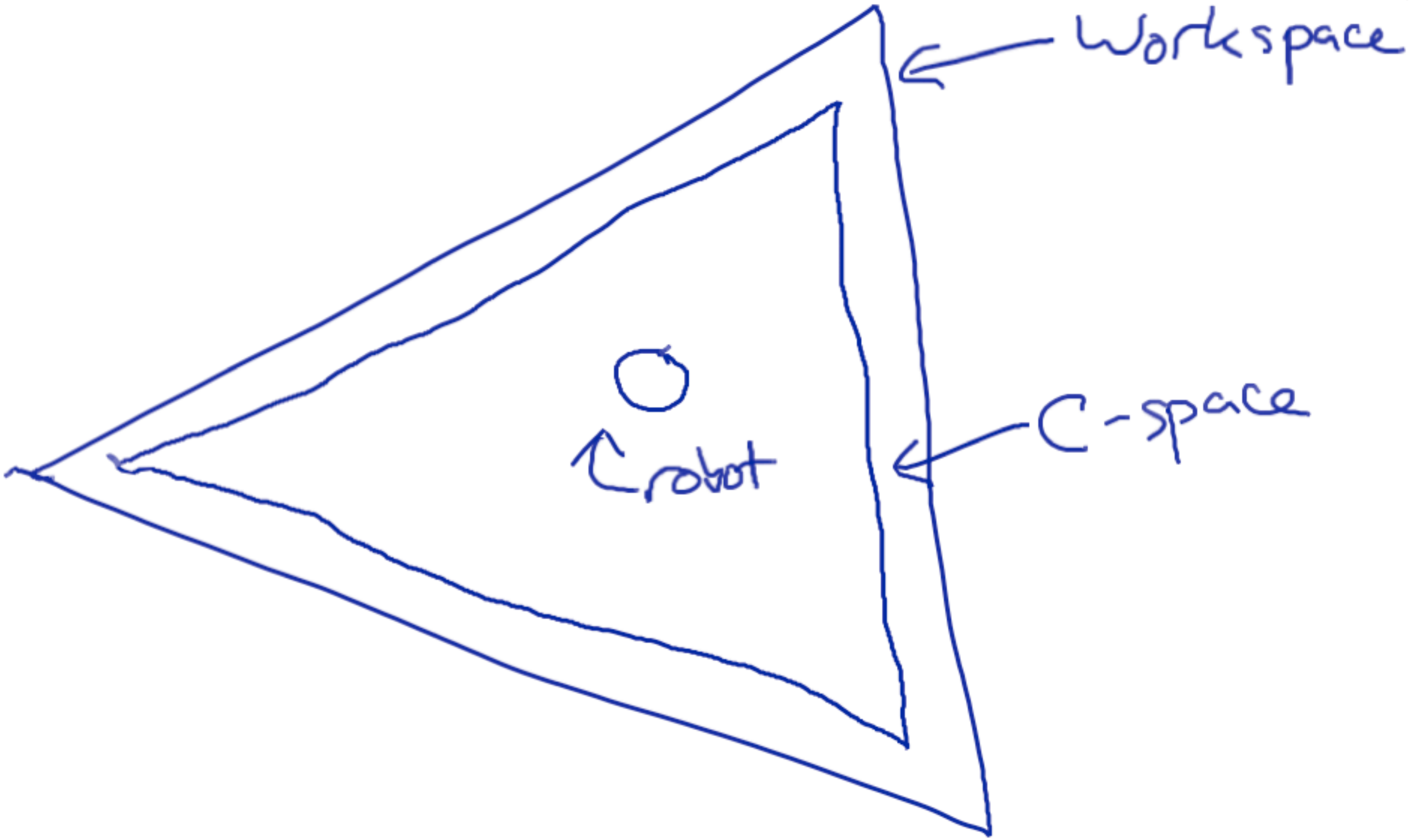
Configuration space

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

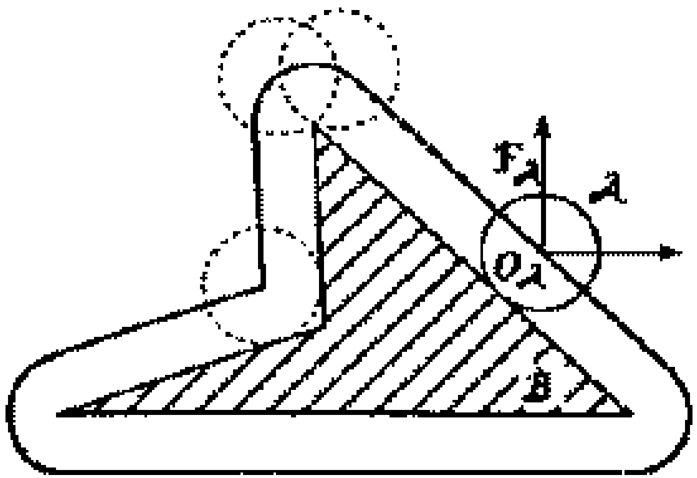
Note: dynamic planning

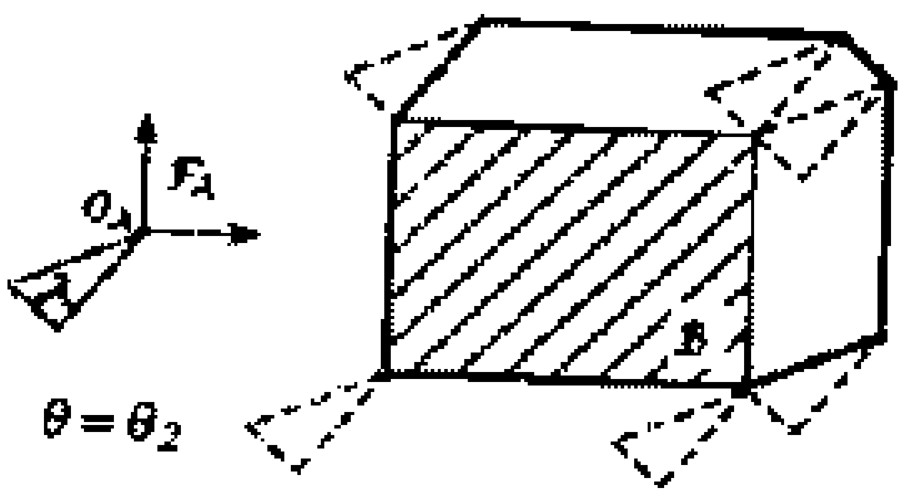
- Includes inertia as well as configuration
  - $\dot{\mathbf{q}}, \mathbf{q}$
- Harder, since twice as many dofs, and typically stronger constraints
- Won't really cover here…

C-space example



More C-space examples





Another C-space example

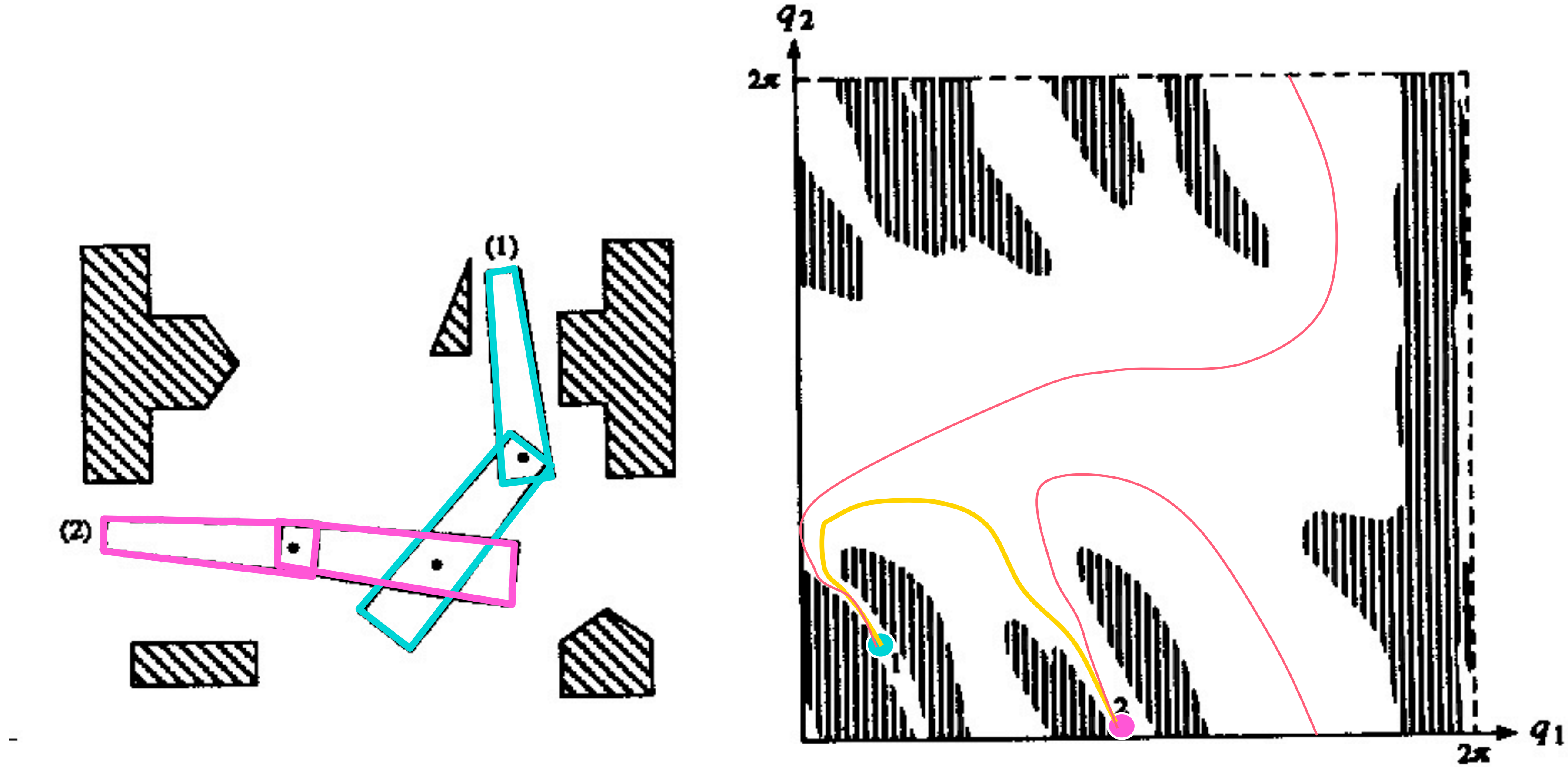


image: J. Kuffner

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 $[0, 2\pi)$!
- Ball & socket joint (3d angle) $\subseteq$ unit sphere = SO(3)

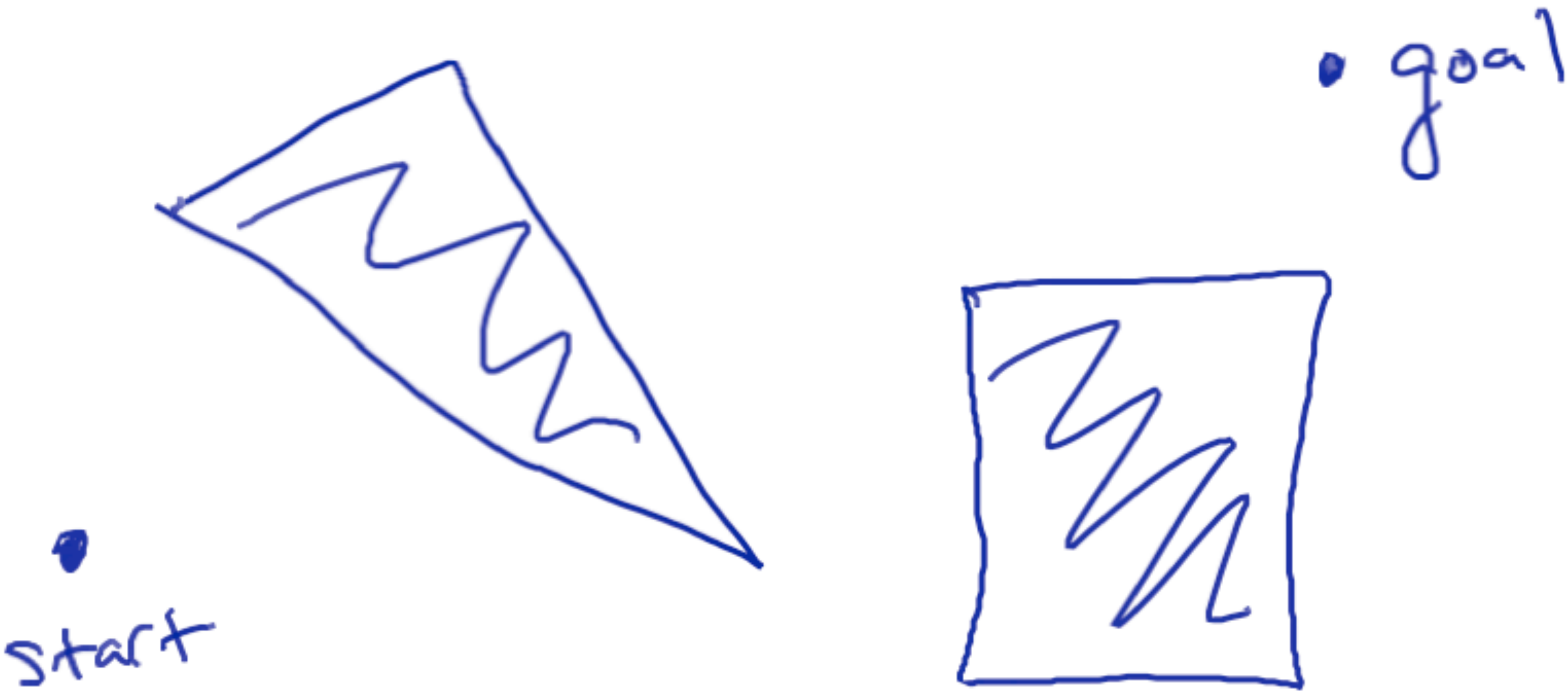
Topology example

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

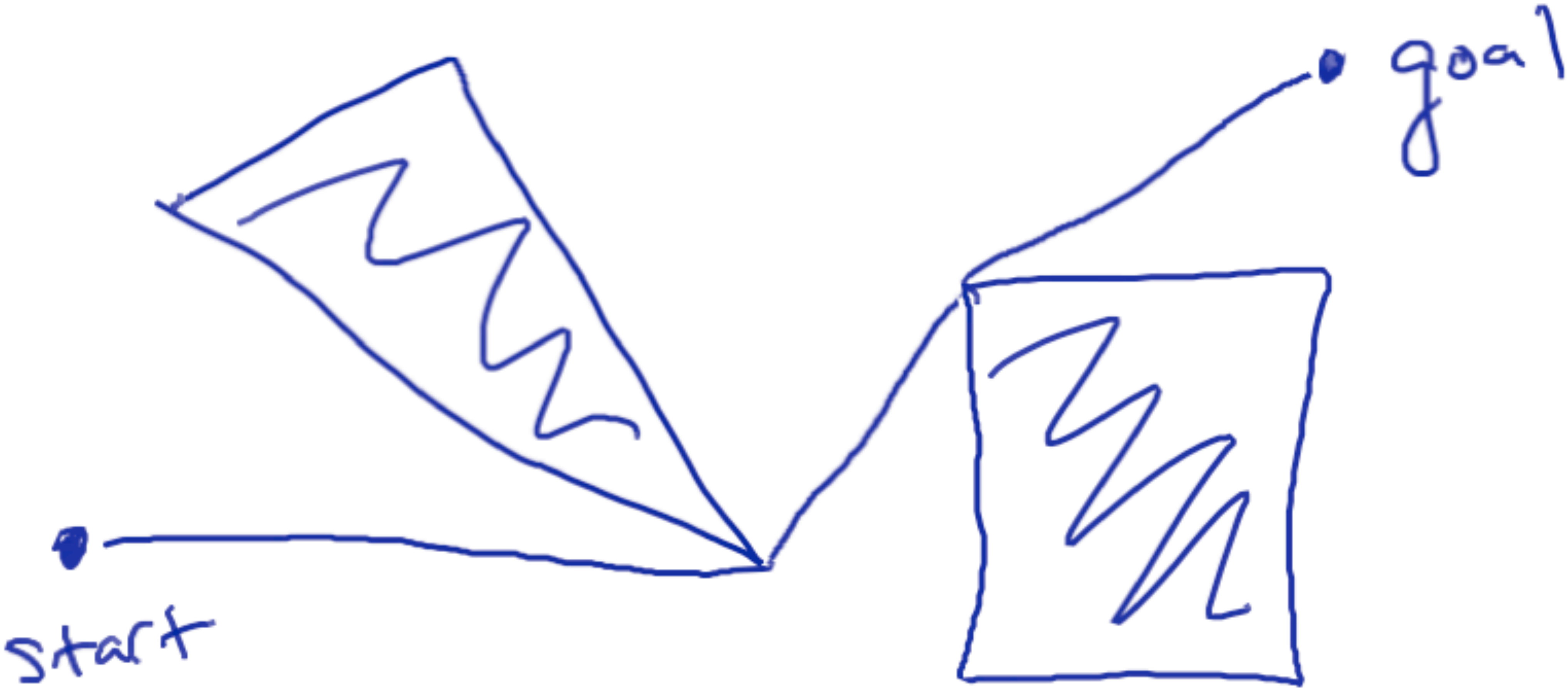
Back to planning

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

Shortest path in C-space



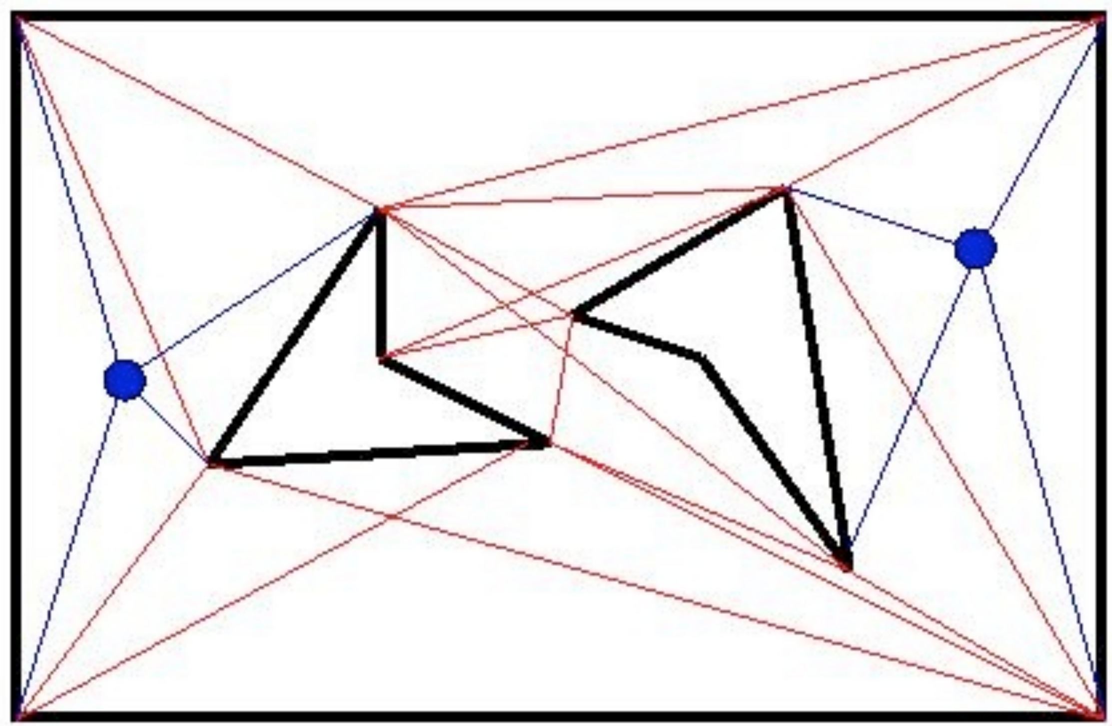
Shortest path in C-space



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 faster algorithms, $O(n^2)$ or $O(k+n \log(n))$, see [Latombe, pg 157]
  - k = number of edges that wind up in visibility graph
  - in dimension d, graph gets much bigger, more complex; speedup tricks stop working
- Once we have graph, search it!

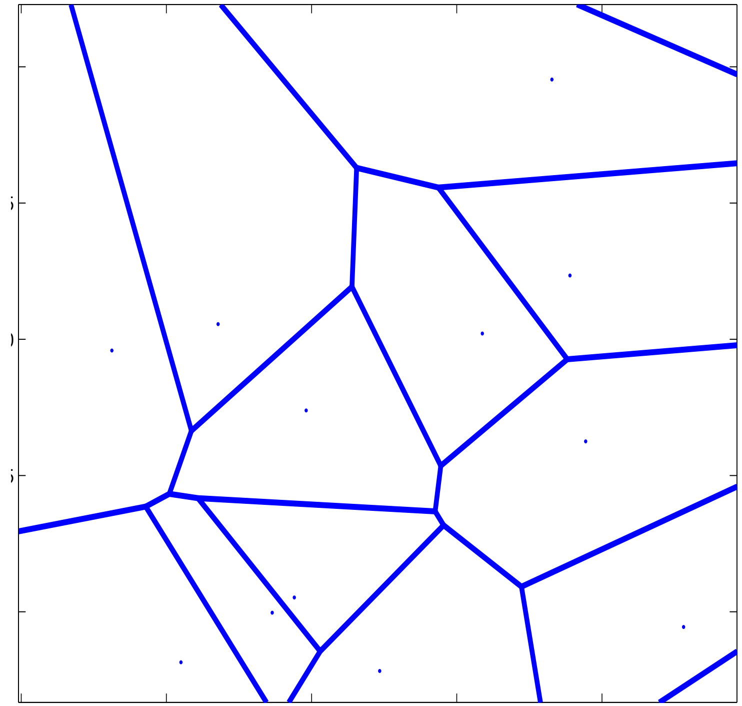
Discussion of visibility graph

- 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

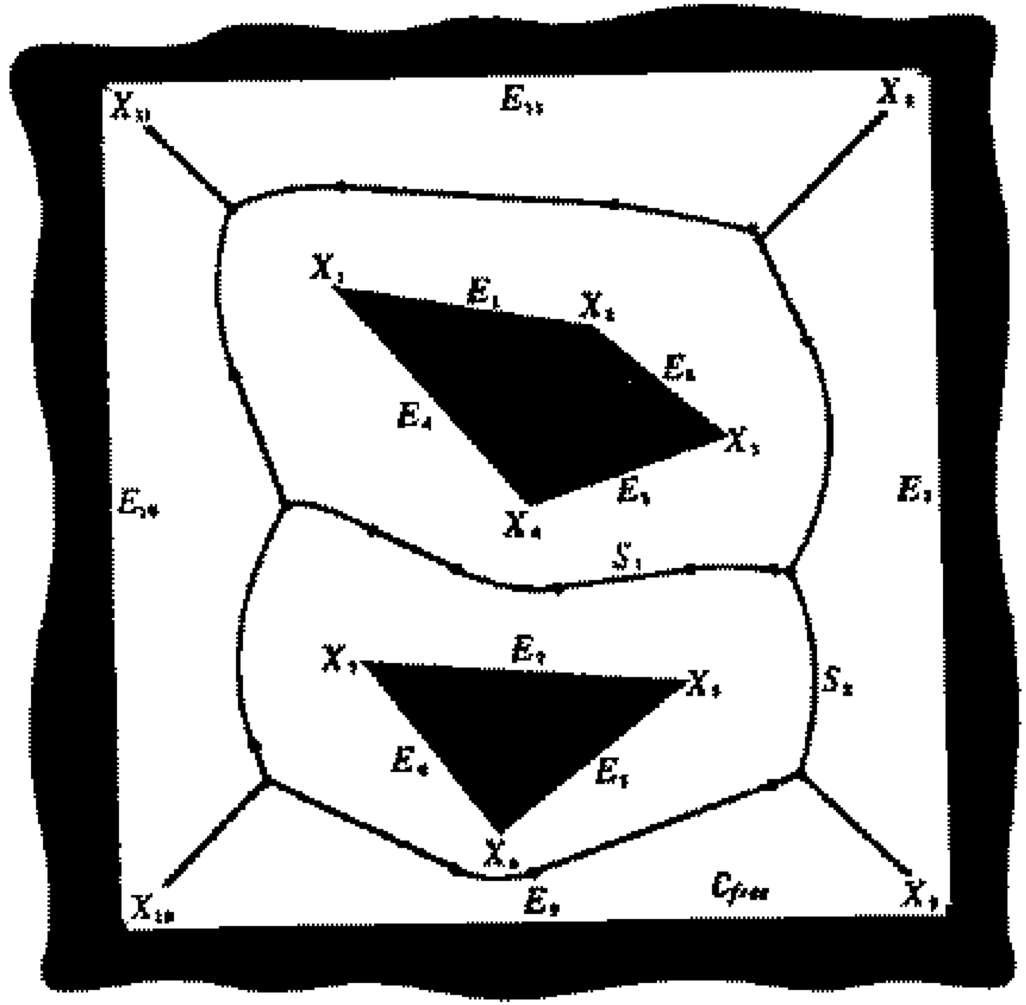
- Could just pad obstacles
  - but how much is enough? might make infeasible…
- What if we try to stay as far away from obstacles as possible?

Voronoi graph



- Set of all places equidistant from two or more obstacles: ***Voronoi graph***
  - point obstacles: network of line segments
  - nonzero extent: graph may include curves

Voronoi w/ polygonal C-space

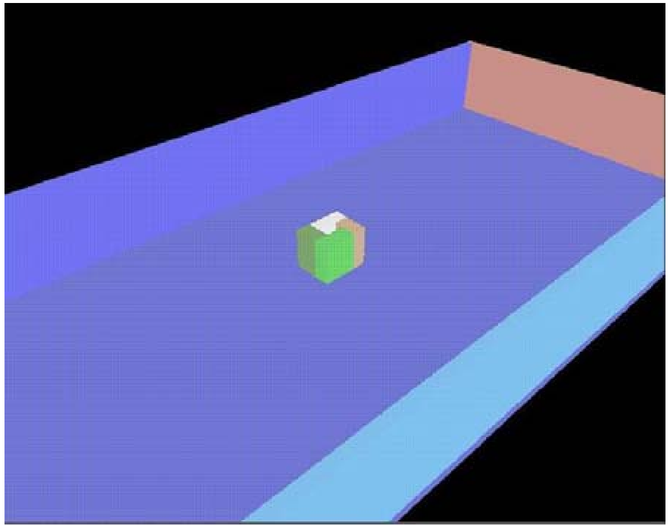


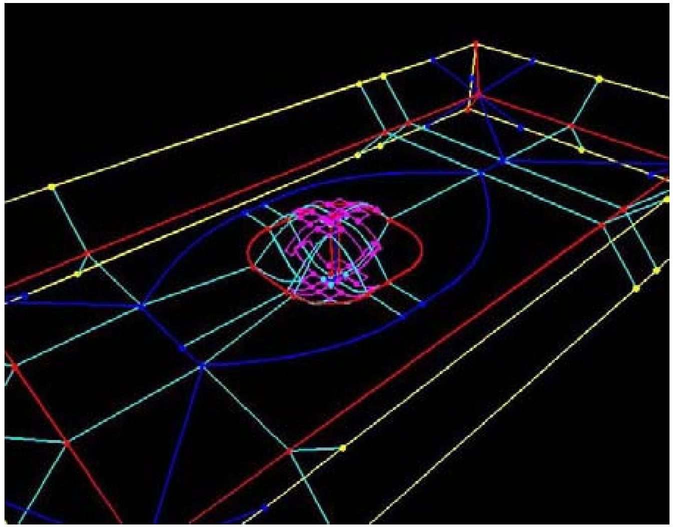
Voronoi method for planning

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

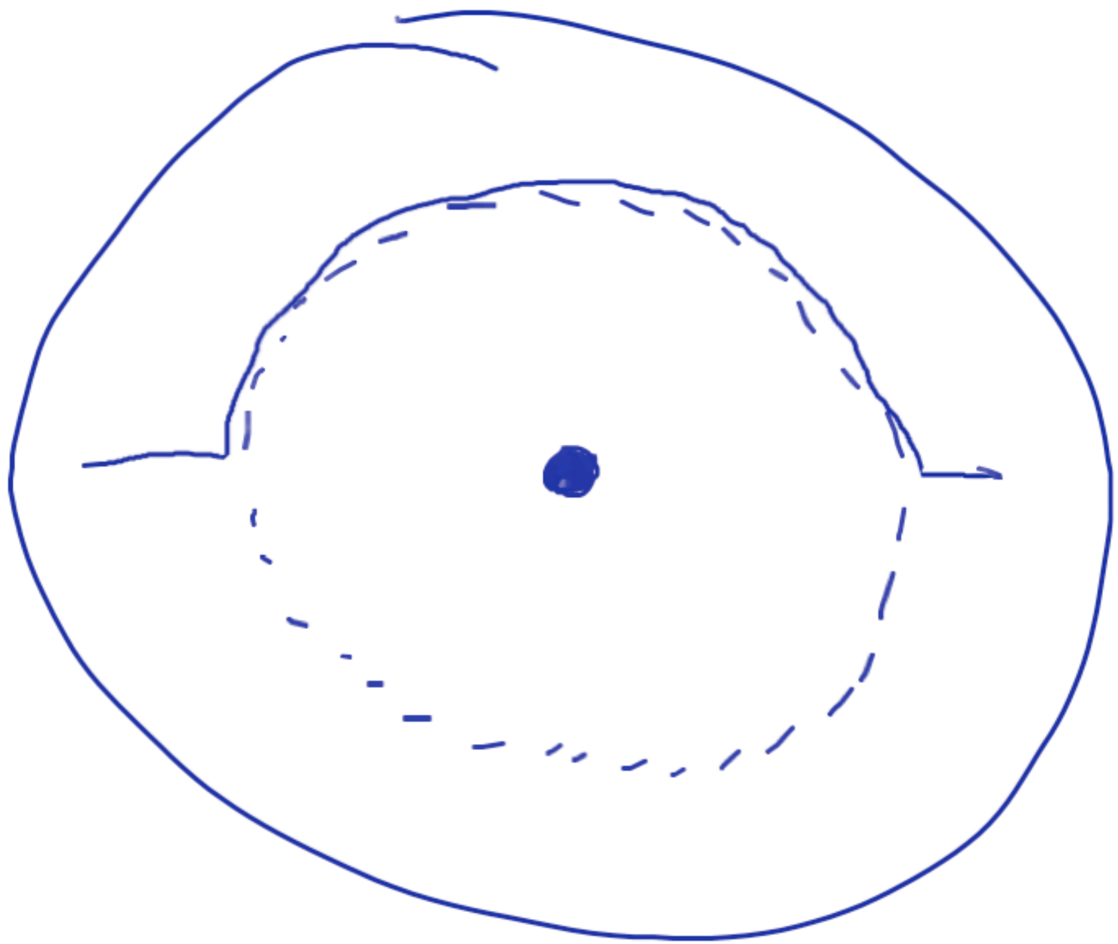
Voronoi discussion

- 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)



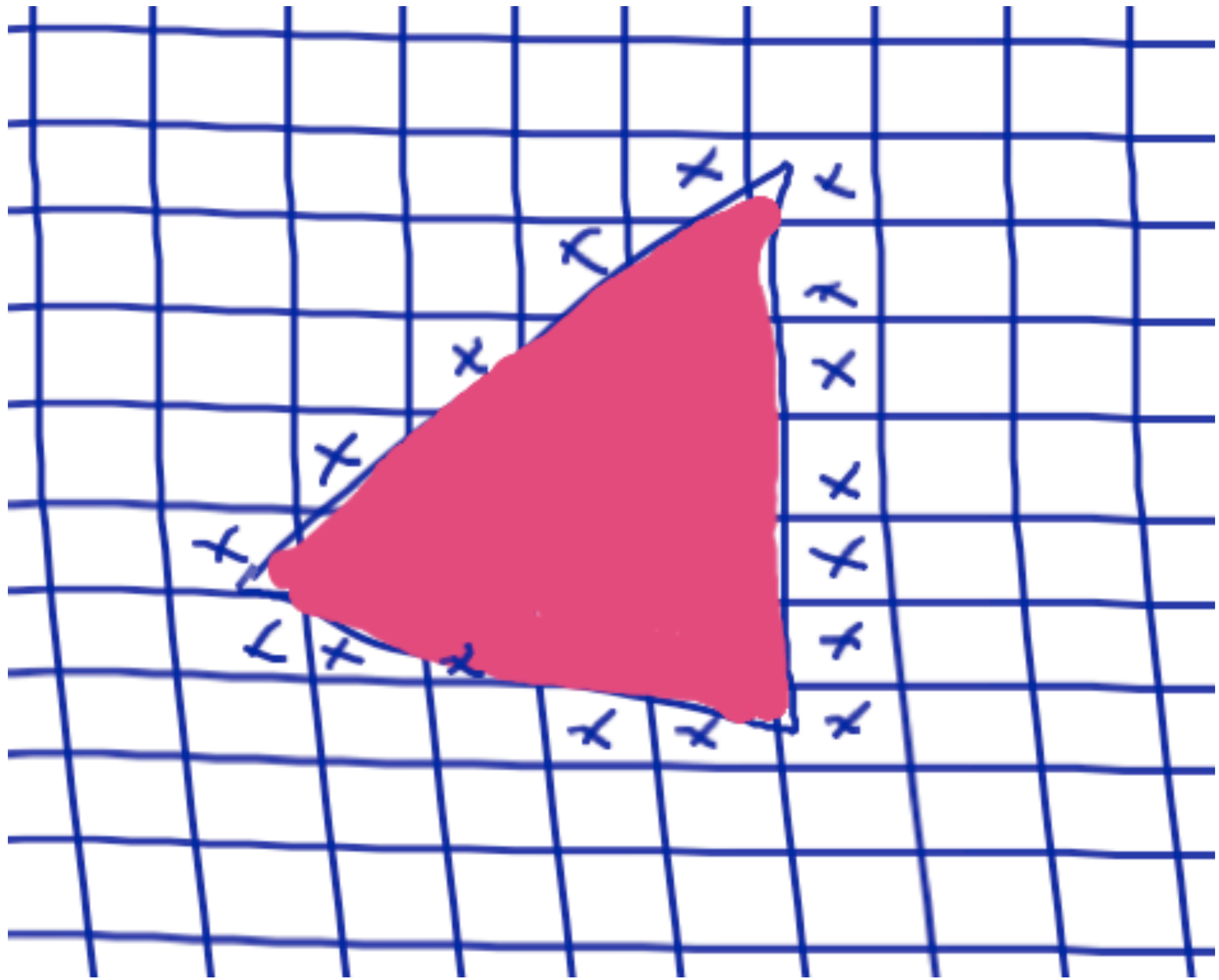


Voronoi discussion



- Bad: kind of gun-shy about obstacles

(Approximate) cell decompositions



Planning algorithm

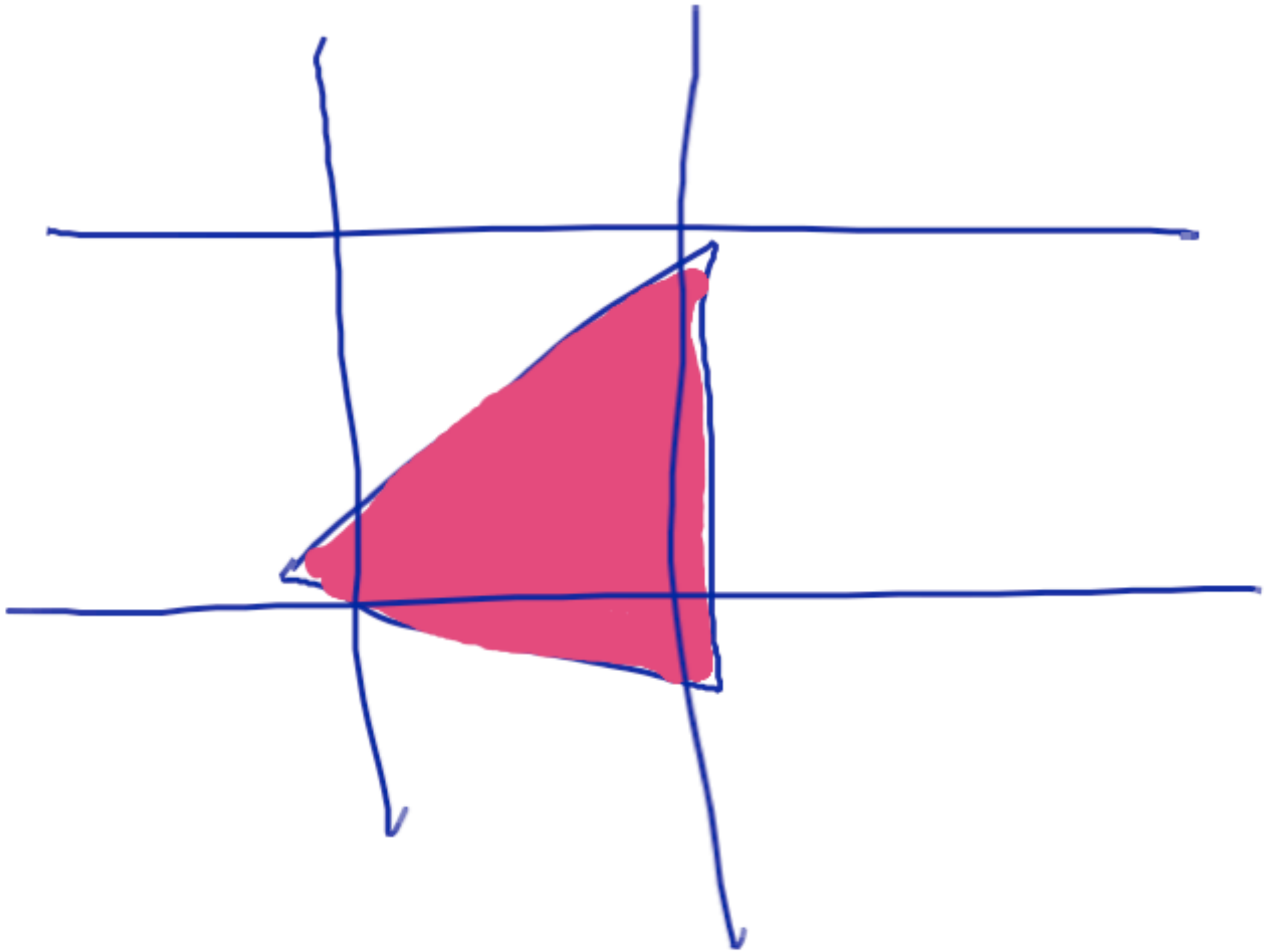
- Lay down a grid in C-space
- Delete cells that intersect obstacles
- Connect neighbors
- A*
- If no path, double resolution and try again
  - never know when we're done

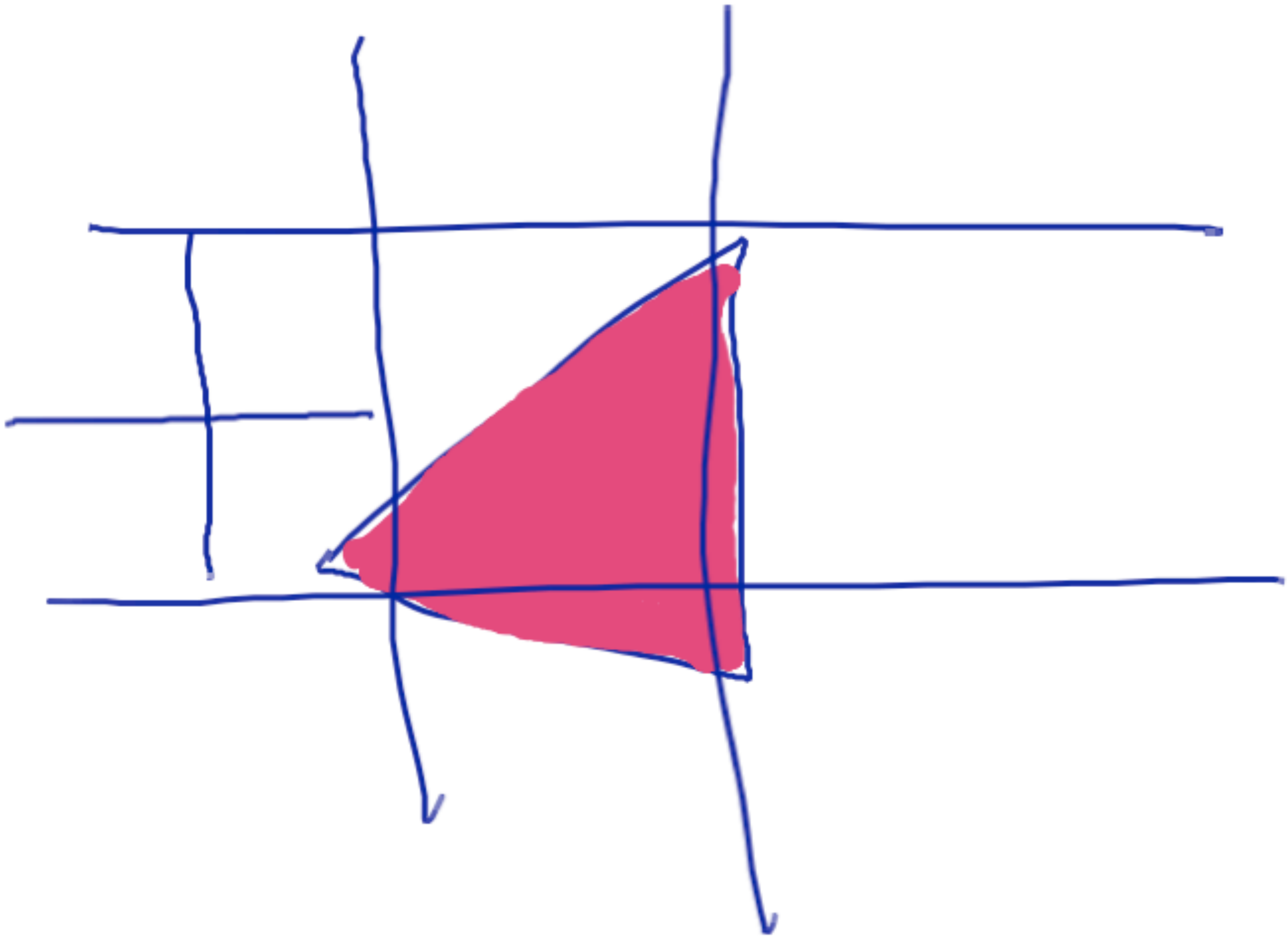
Planning algorithm

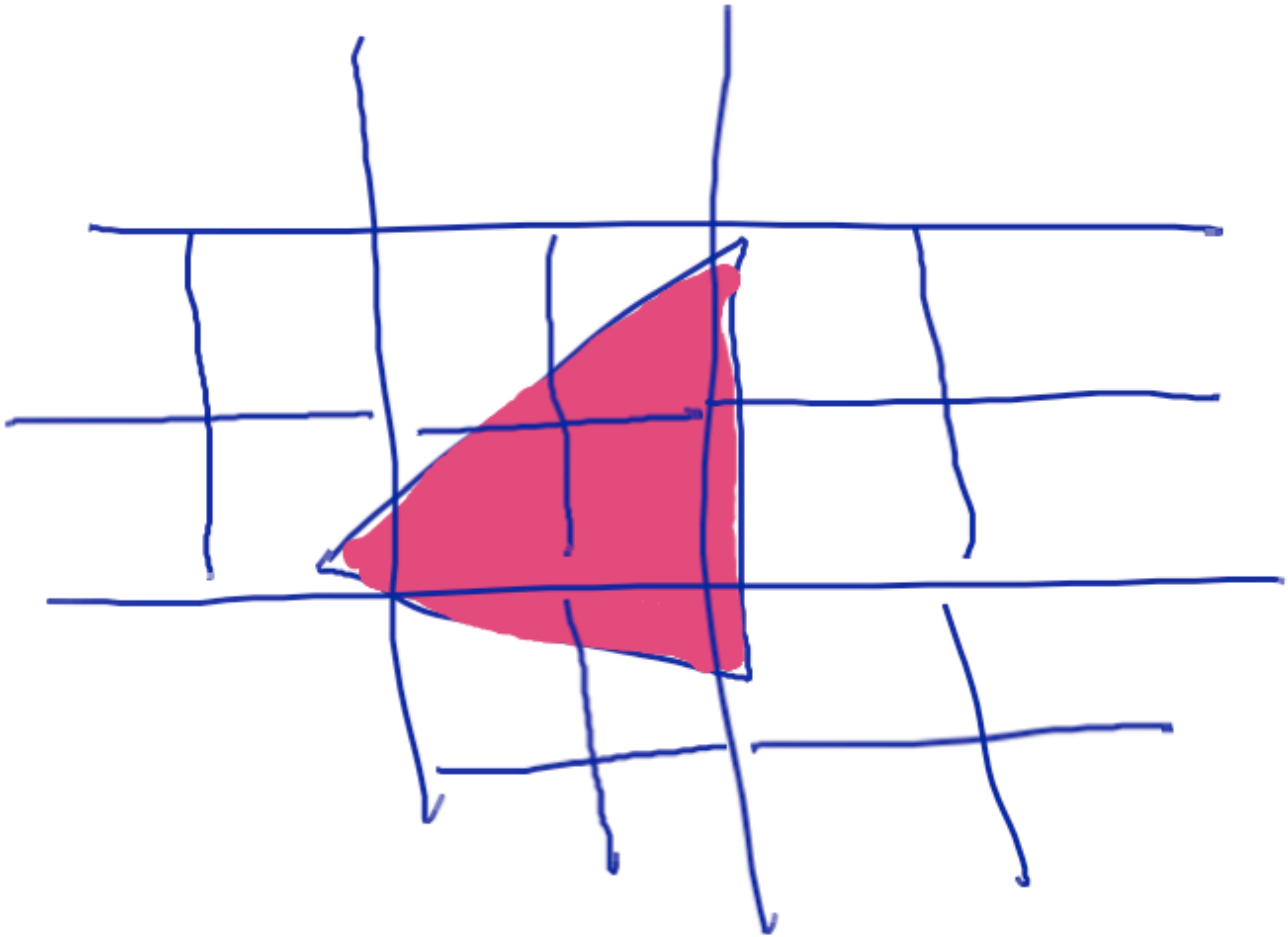
- This method is what we were using in end-effector planning examples above
- Works pretty well except:
  - need high resolution near obstacles
  - want low res away from obstacles

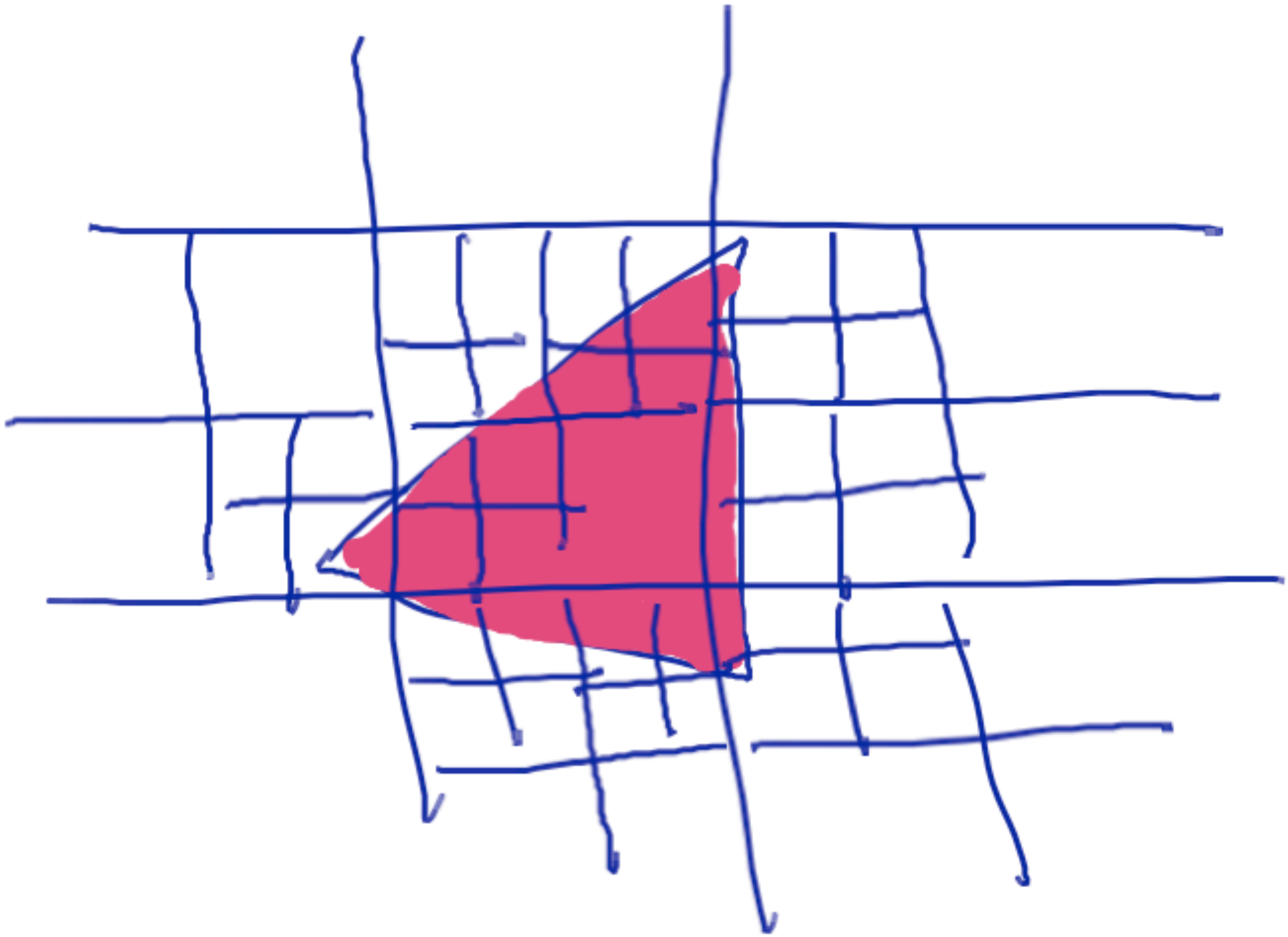
Fix: variable resolution

- 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









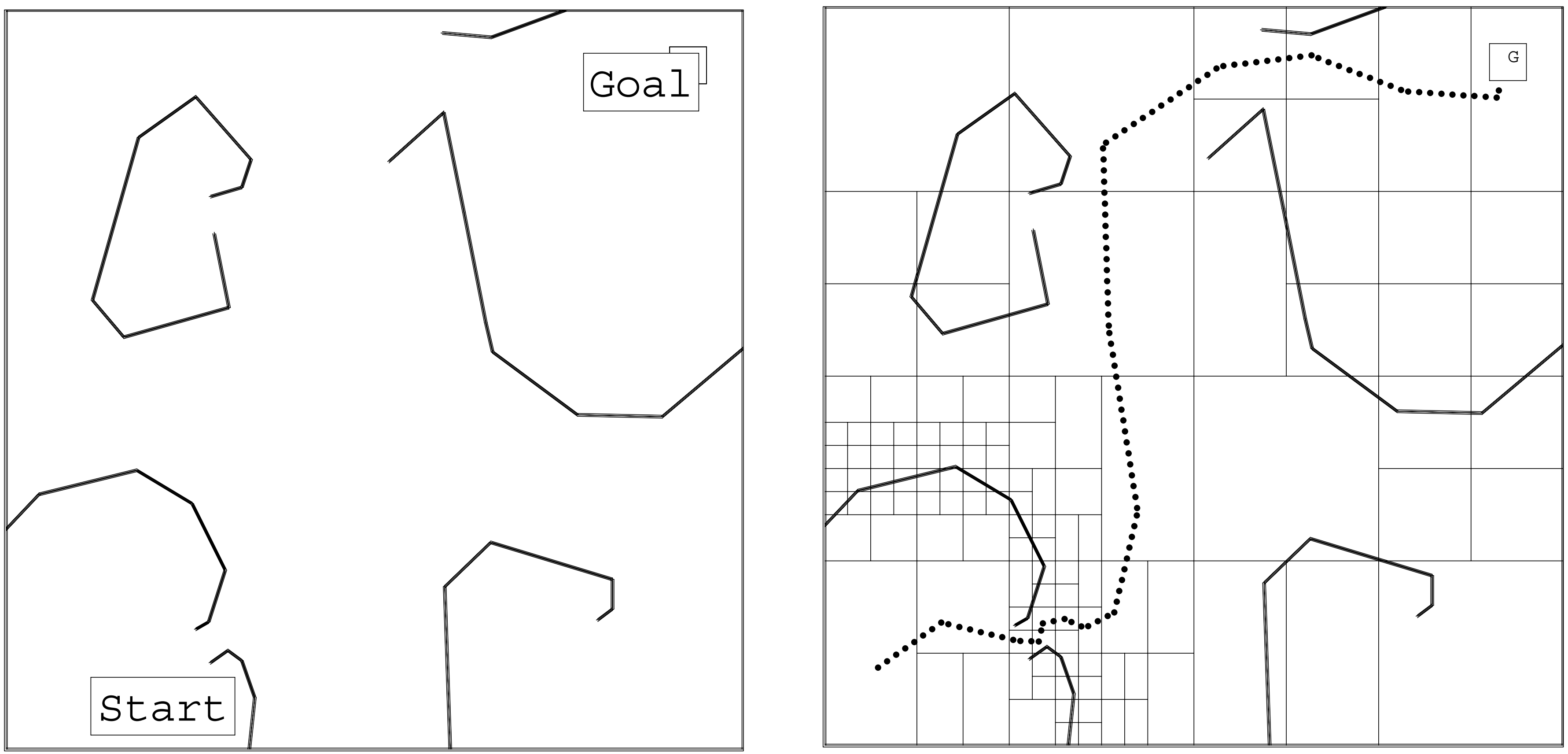
Discussion

- 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

An adaptive splitter: parti-game

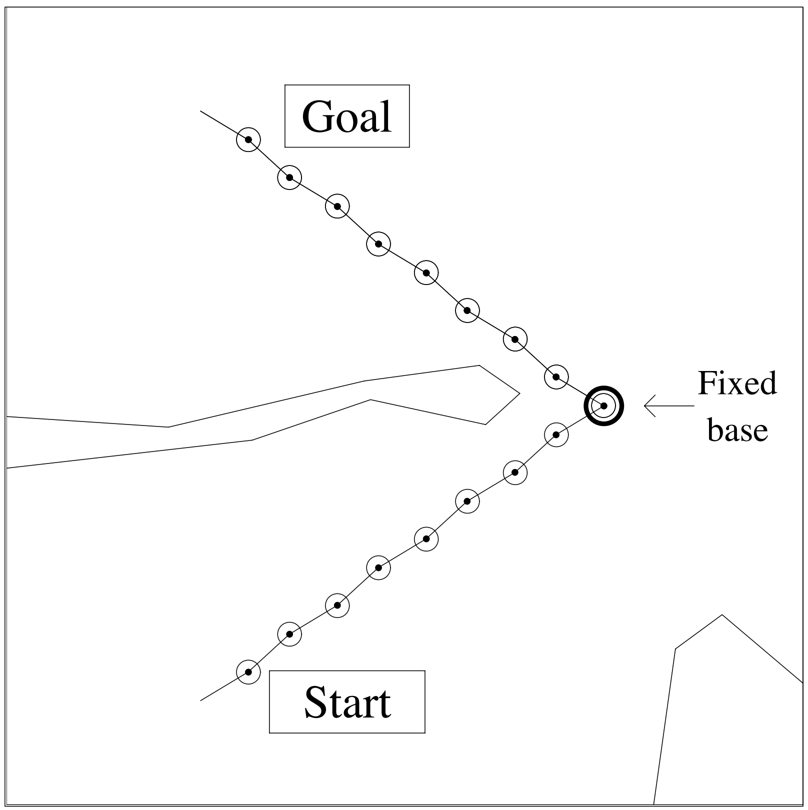


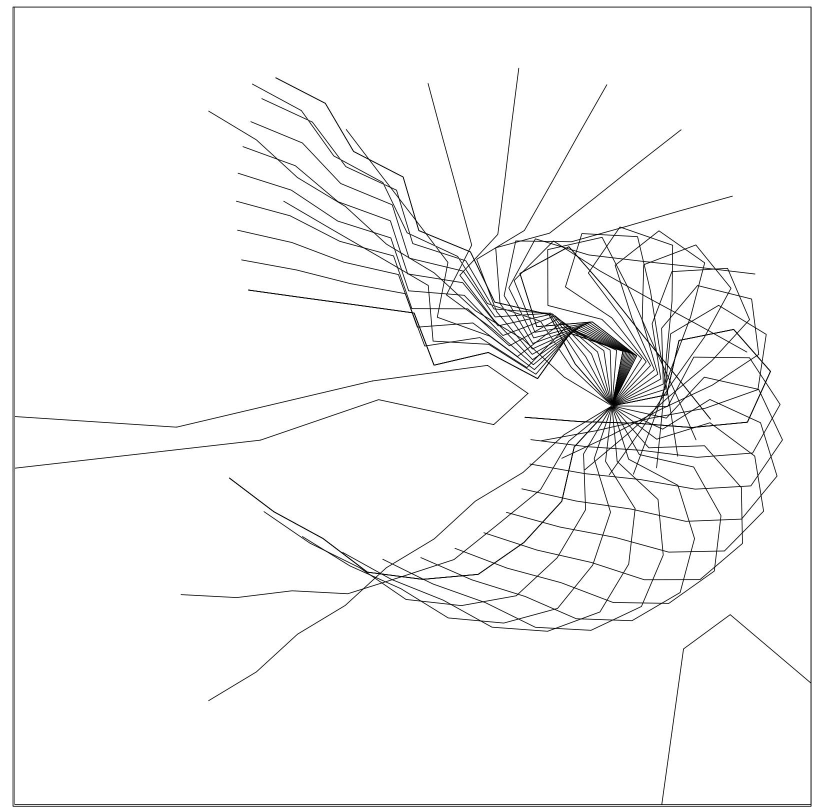
Andrew Moore and Chris Atkeson. *The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces.* http://www.autonlab.org/autonweb/14699.html

Parti-game algorithm

- Sample 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

9dof planar arm





85 partitions total

Randomness in search

Rapidly-exploring Random Trees

- Break up C-space into Voronoi regions around random landmarks
- Invariant: landmarks always form a tree
  - known path to root
- Subject to this requirement, placed in a way that tends to split large Voronoi regions
  - coarse-to-fine search
- Goal: ***feasibility*** not ***optimality*** *(*)*

RRT assumptions

- RANDOM_CONFIG
  - samples from C-space
- EXTEND(**q**, **q**')
  - local controller, heads toward **q**' from **q**
  - stops before hitting obstacle (and perhaps also after bound on time or distance)
- FIND_NEAREST(**q**, Q)
  - searches current tree Q for point near **q**

Path Planning with RRTs

RRT = Rapidly-Exploring Random Tree

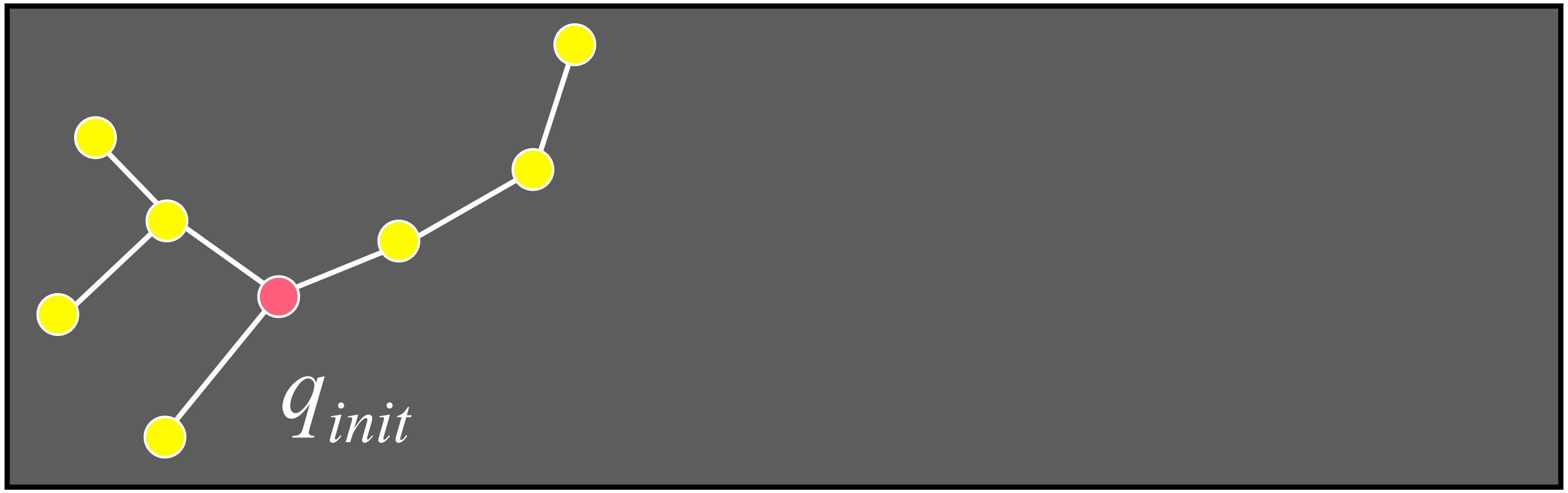
```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

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

[Kuffner & LaValle , ICRA'00]

Path Planning with RRTs

RRT = Rapidly-Exploring Random Tree



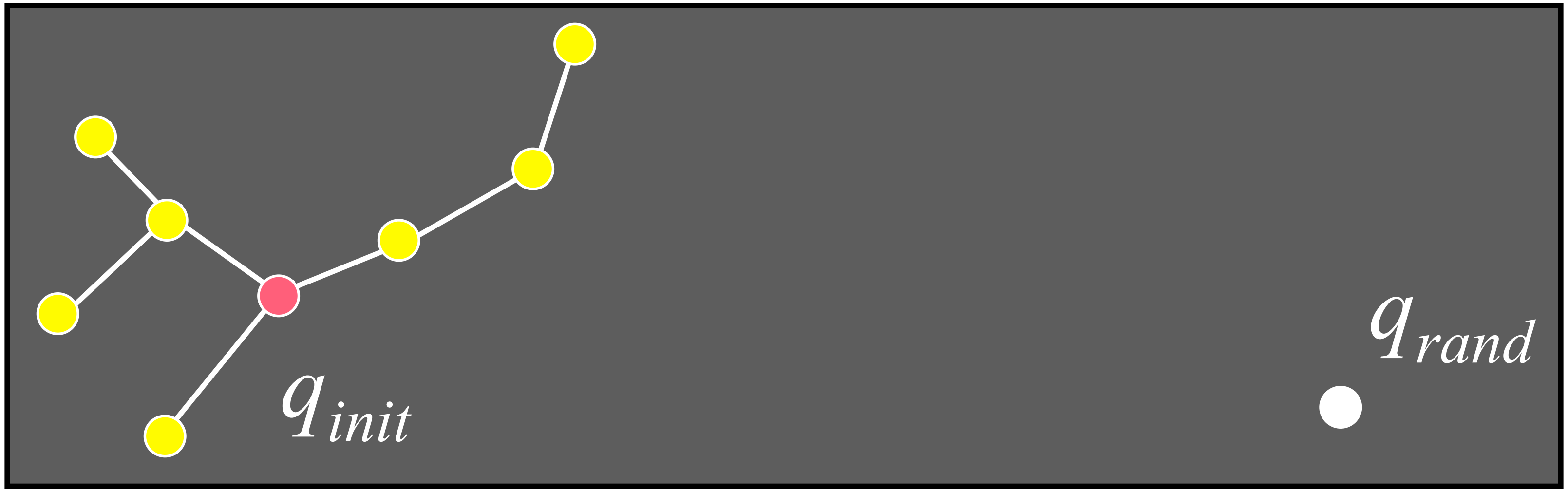

```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

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

[Kuffner & LaValle , ICRA'00]

Path Planning with RRTs

RRT = Rapidly-Exploring Random Tree



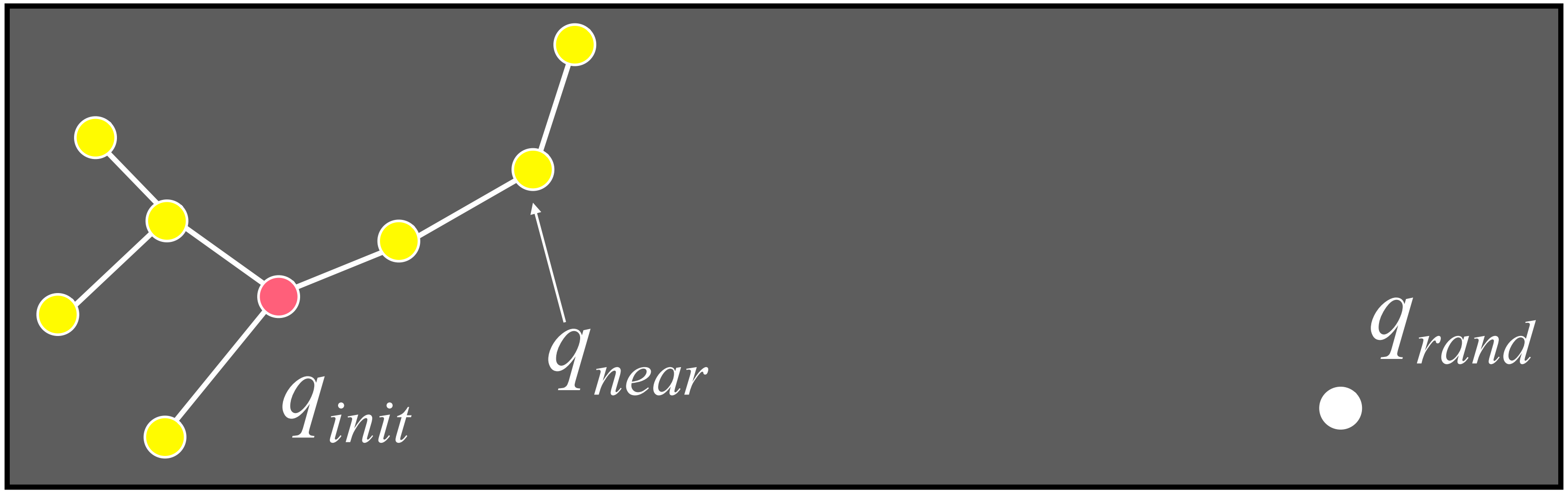

```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

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

[Kuffner & LaValle , ICRA'00]

Path Planning with RRTs

RRT = Rapidly-Exploring Random Tree



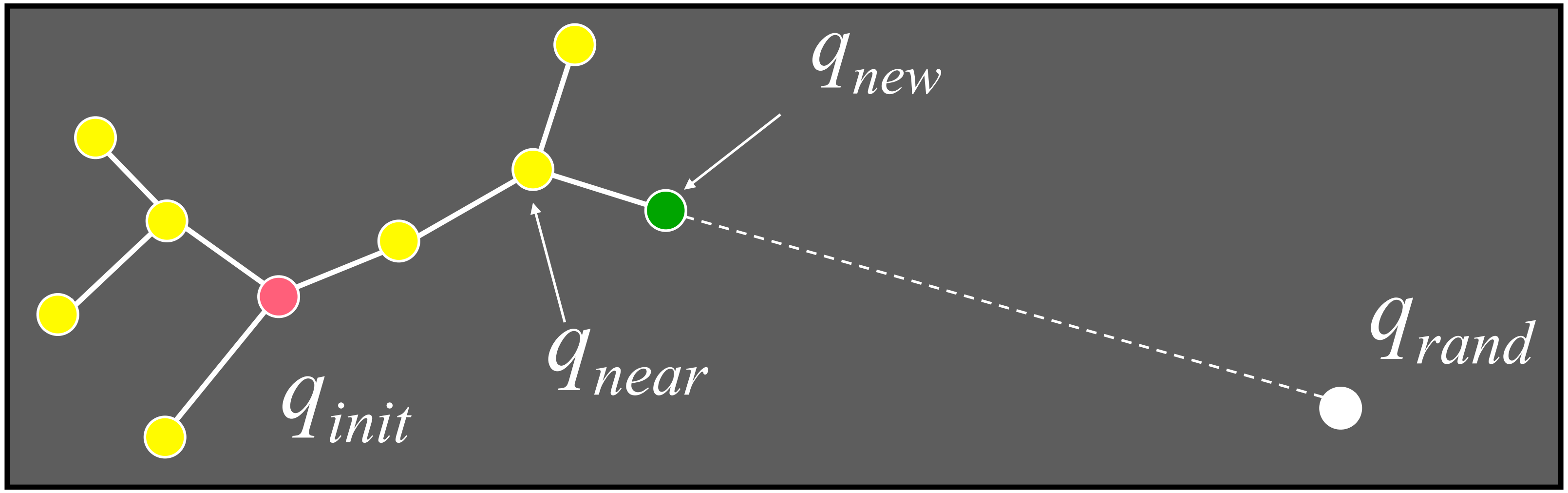

```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

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

[Kuffner & LaValle , ICRA'00]

Path Planning with RRTs

RRT = Rapidly-Exploring Random Tree

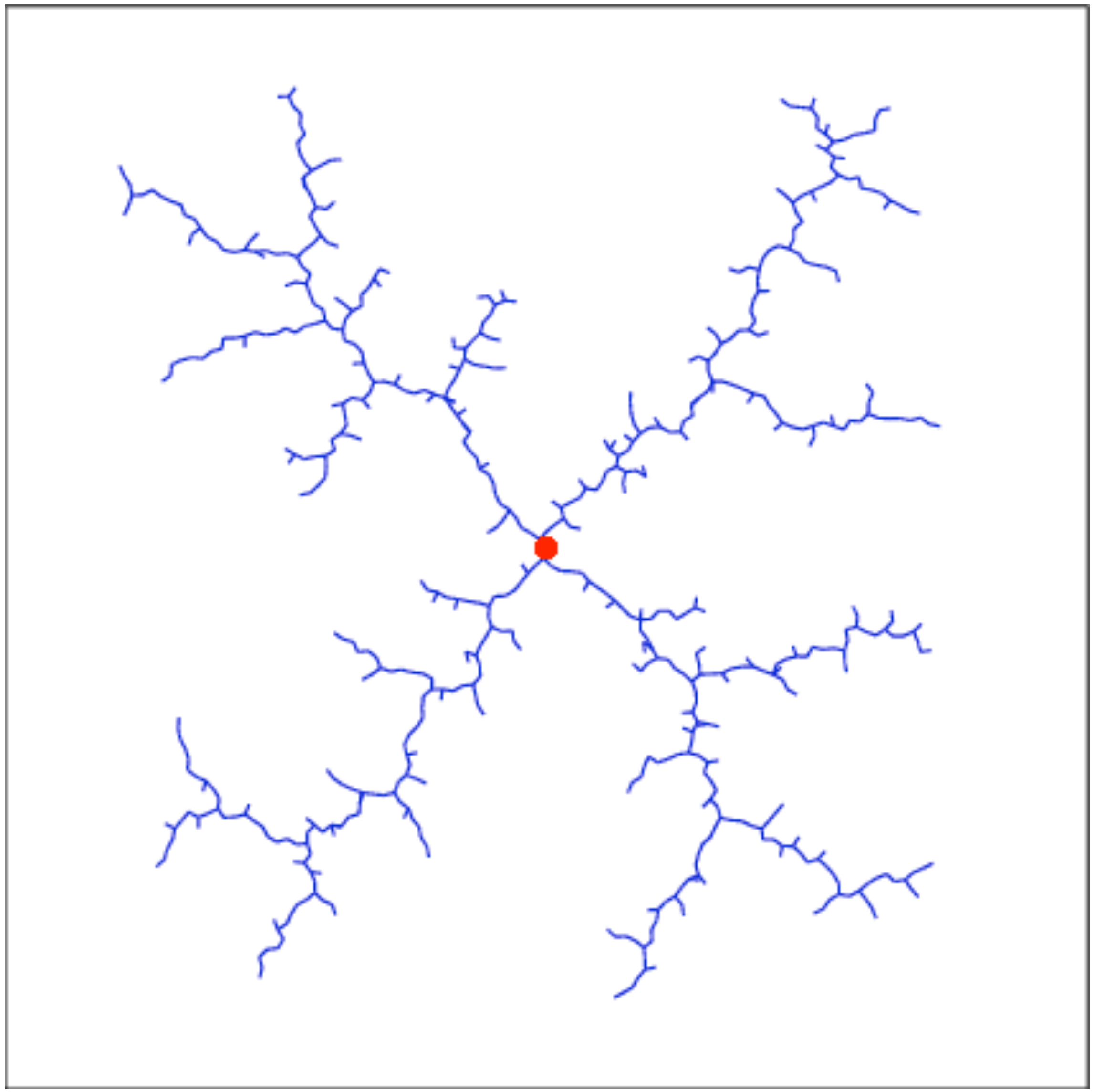



```
BUILT_RRT(q_init) {
    T = q_init
    for k = 1 to K {
        q_rand = RANDOM_CONFIG()
        EXTEND(T, q_rand);
    }
}
```

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

[Kuffner & LaValle , ICRA'00]

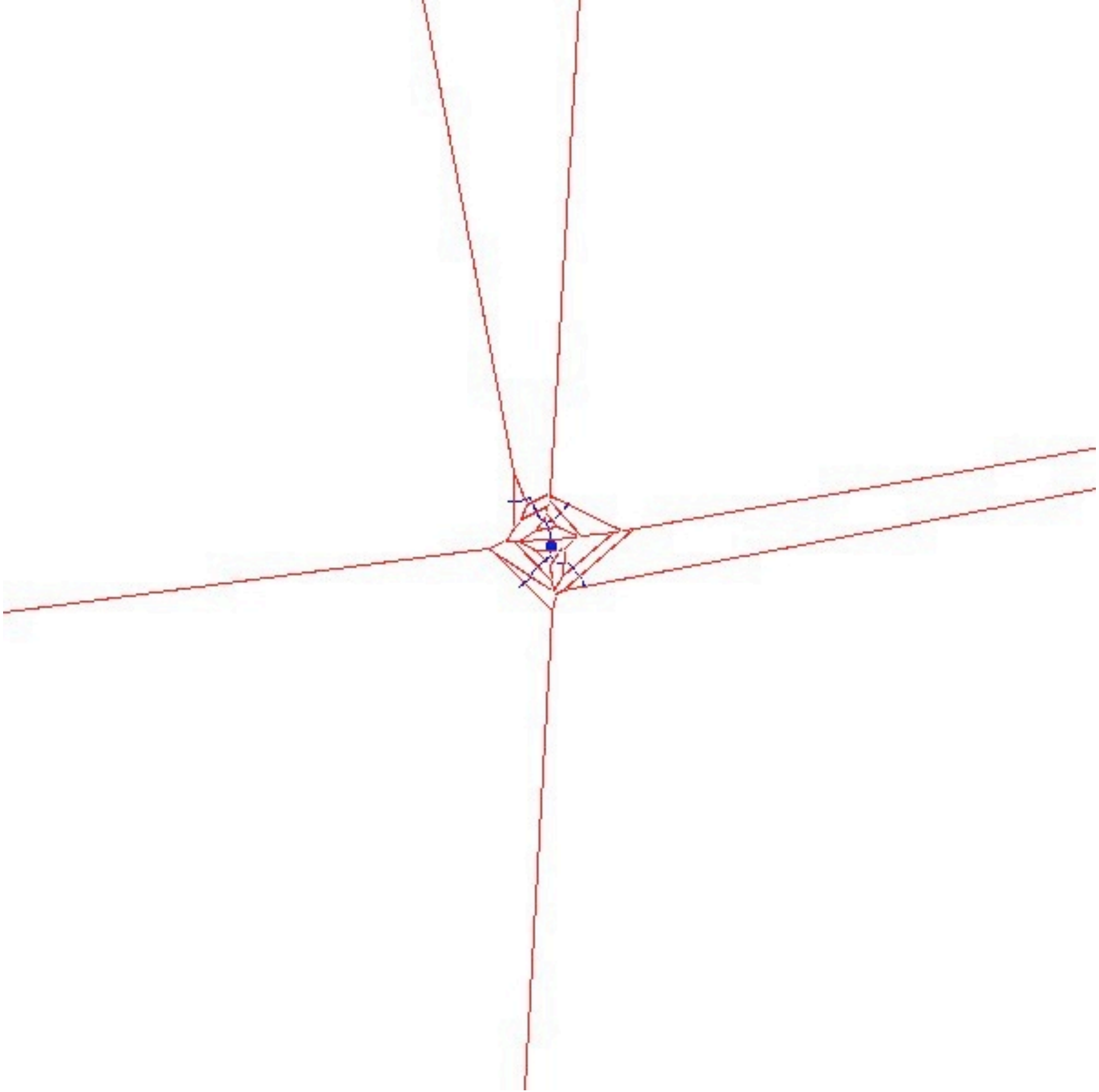
RRT example

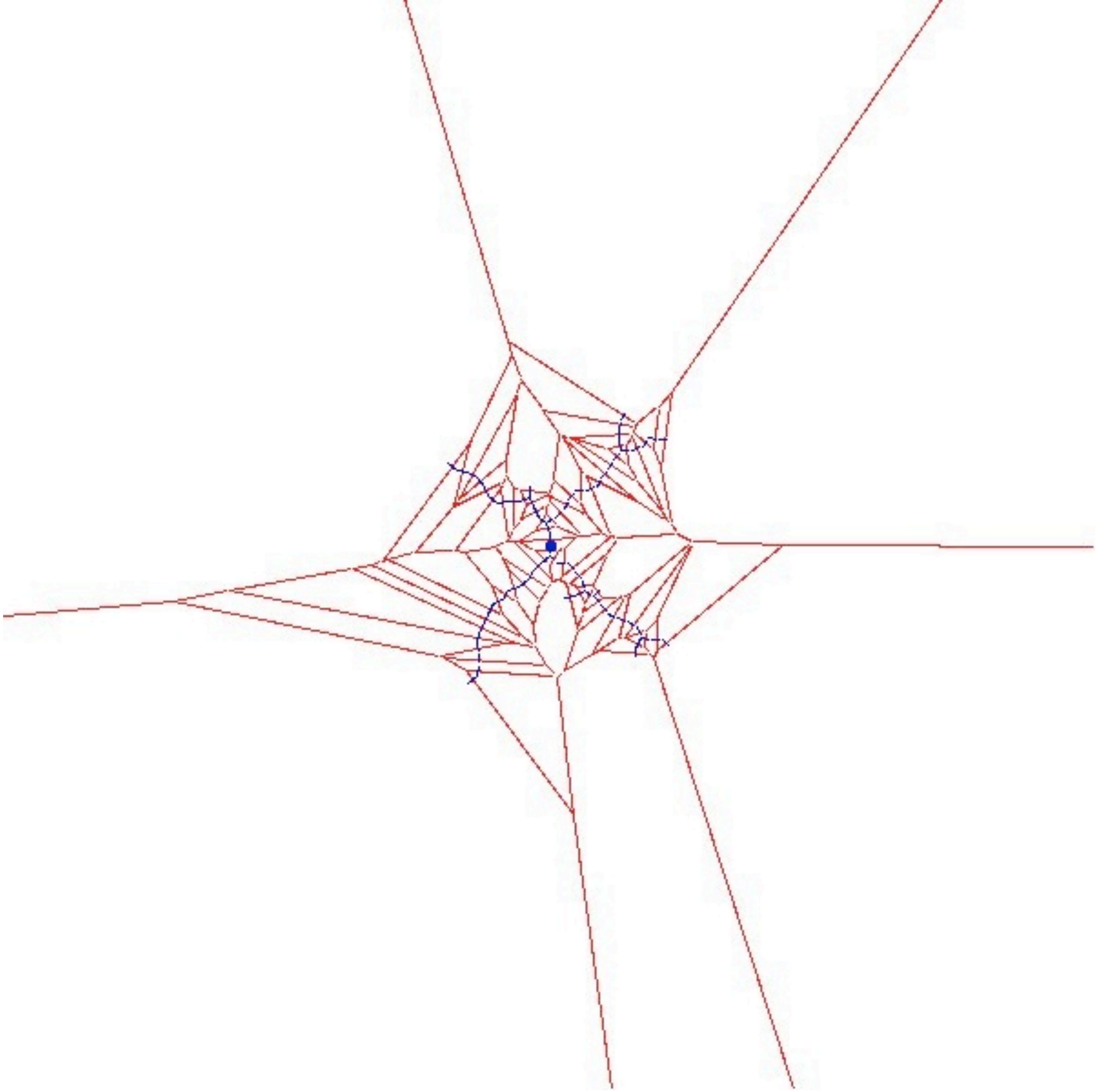


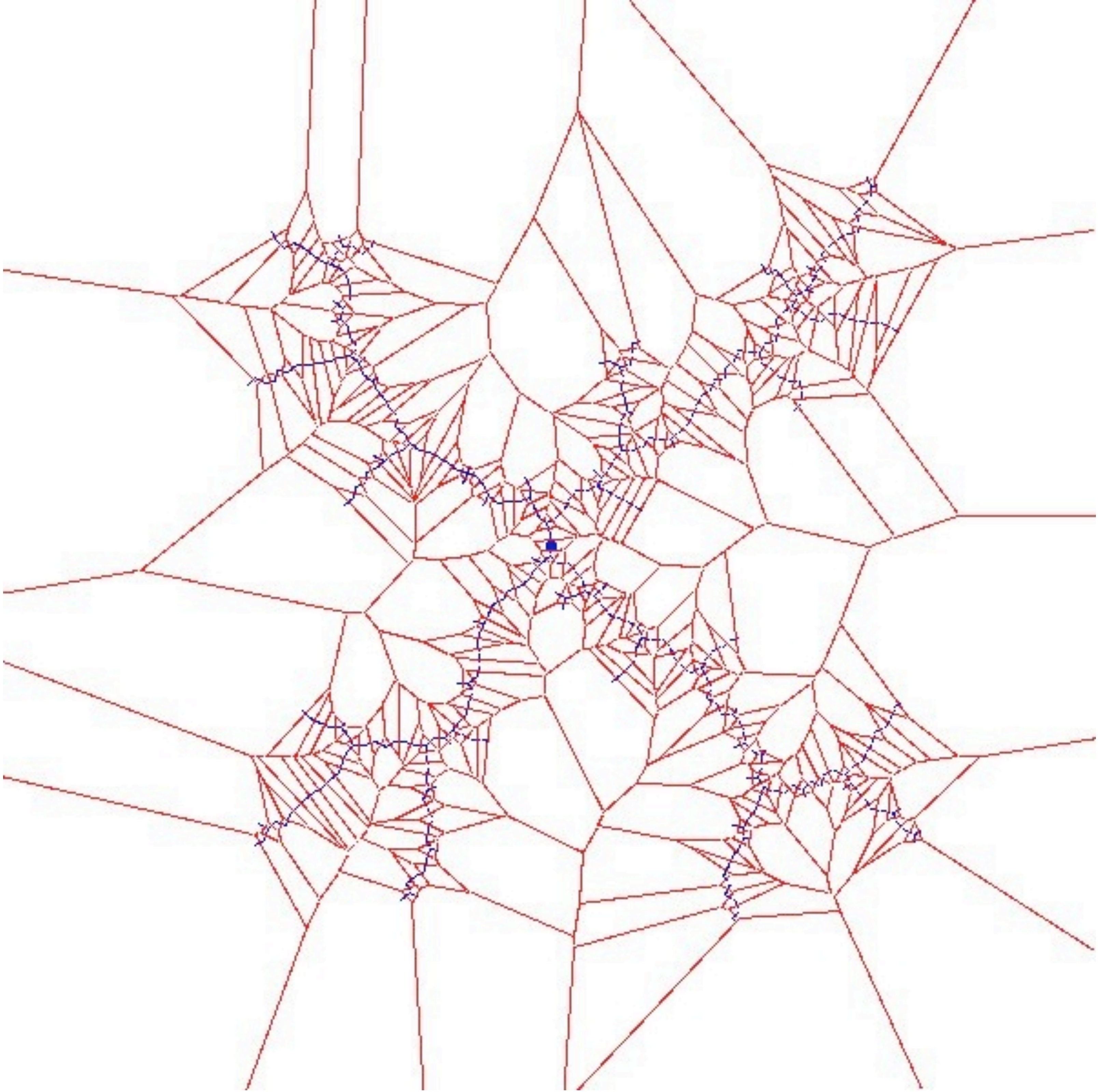
Planar holonomic robot

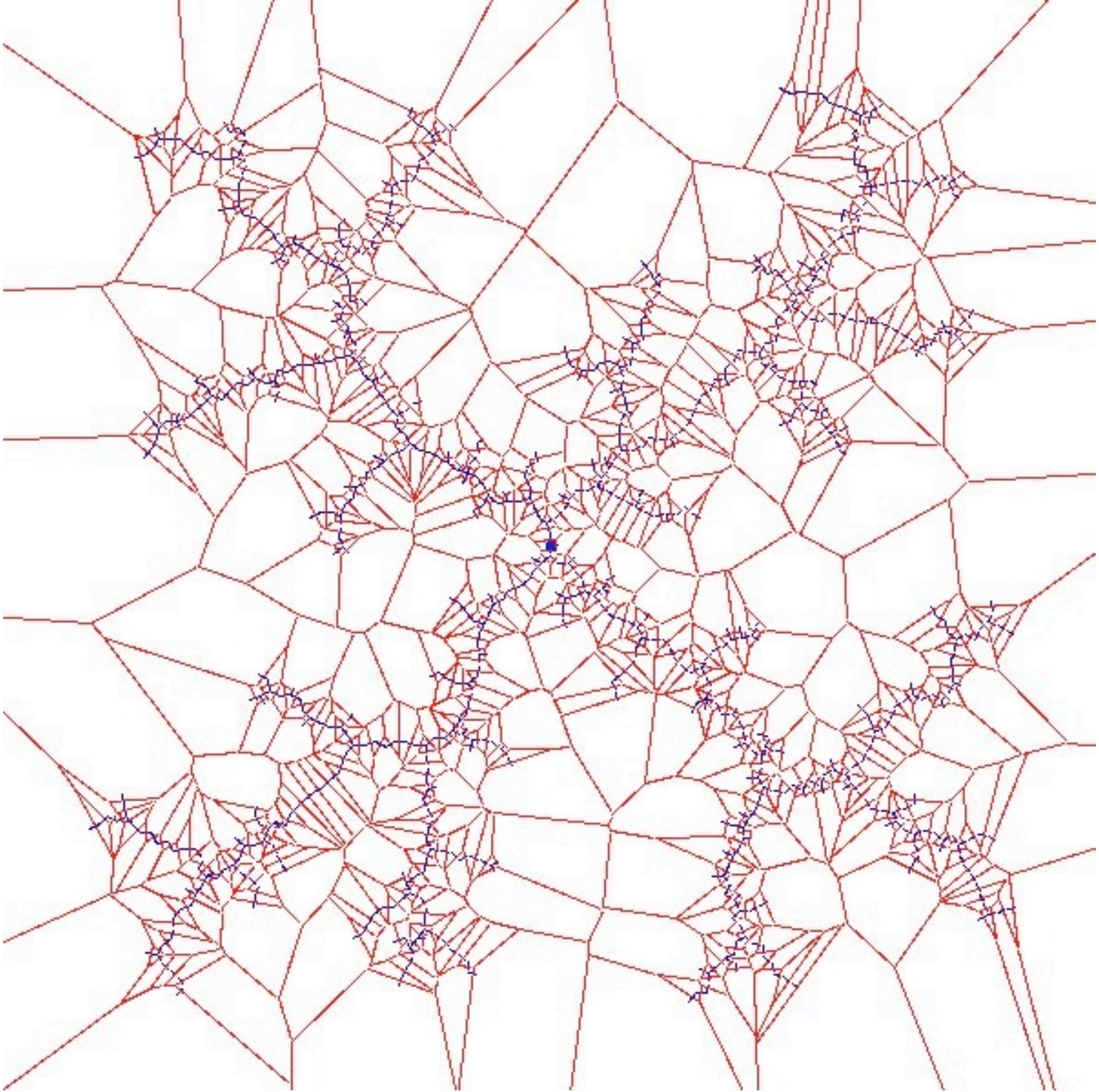
RRTs explore coarse to fine

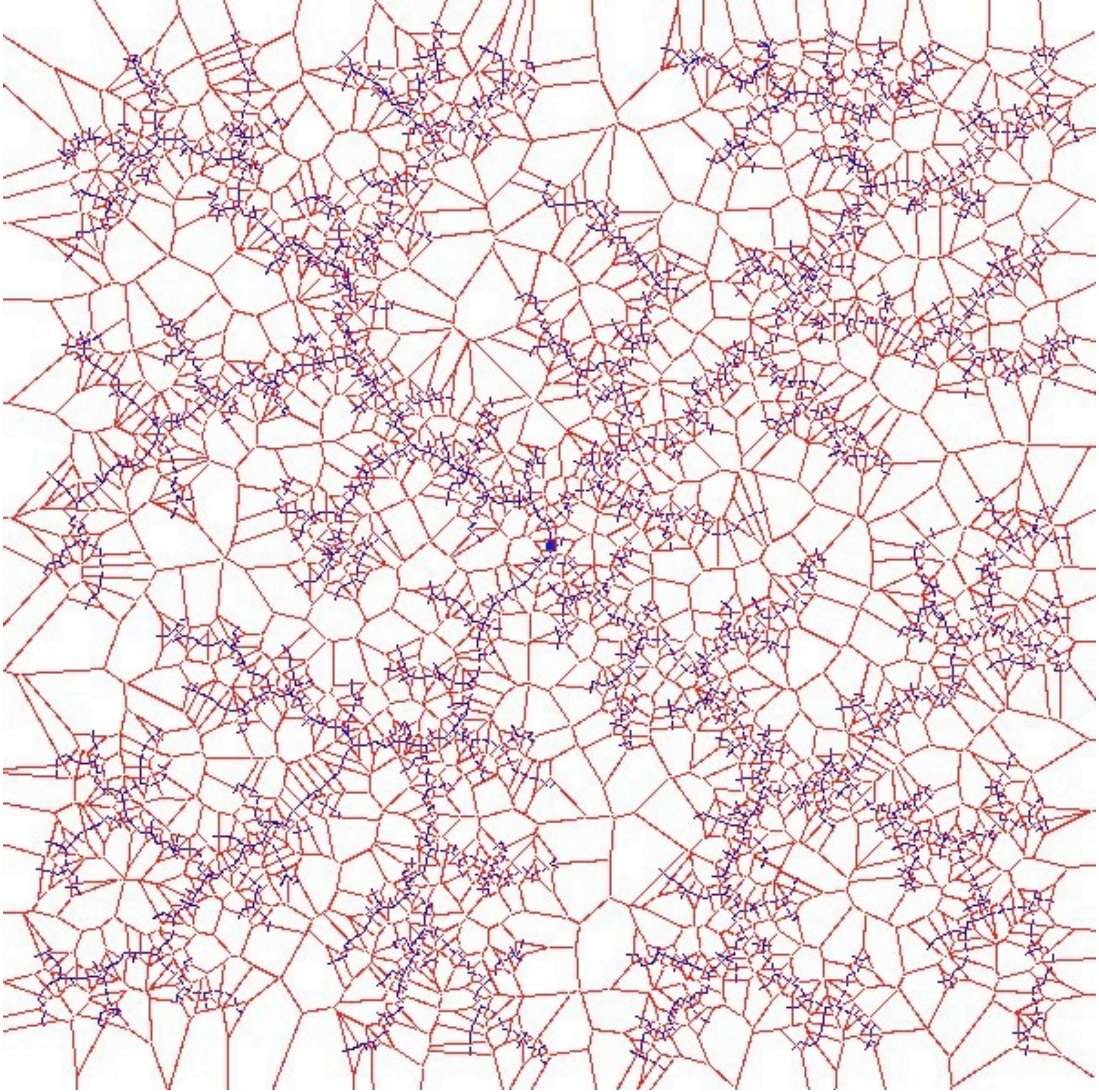
- Tend to break up large Voronoi regions
  - higher probability of q_{rand} being in them
- Limiting distribution of vertices given by RANDOM_CONFIG
  - as RRT grows, probability that q_{rand} is reachable with local controller (and so immediately becomes a new vertex) approaches 1



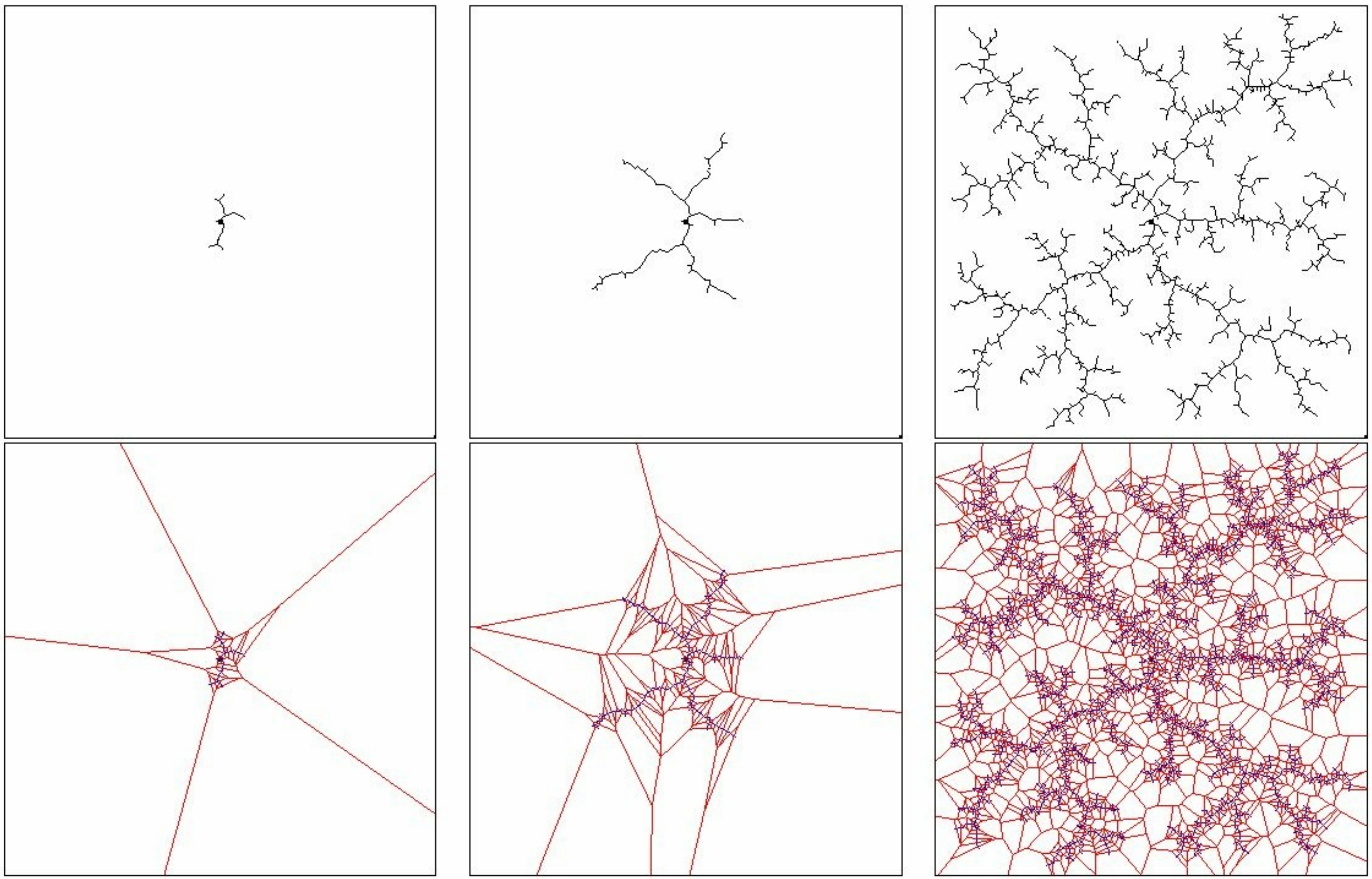




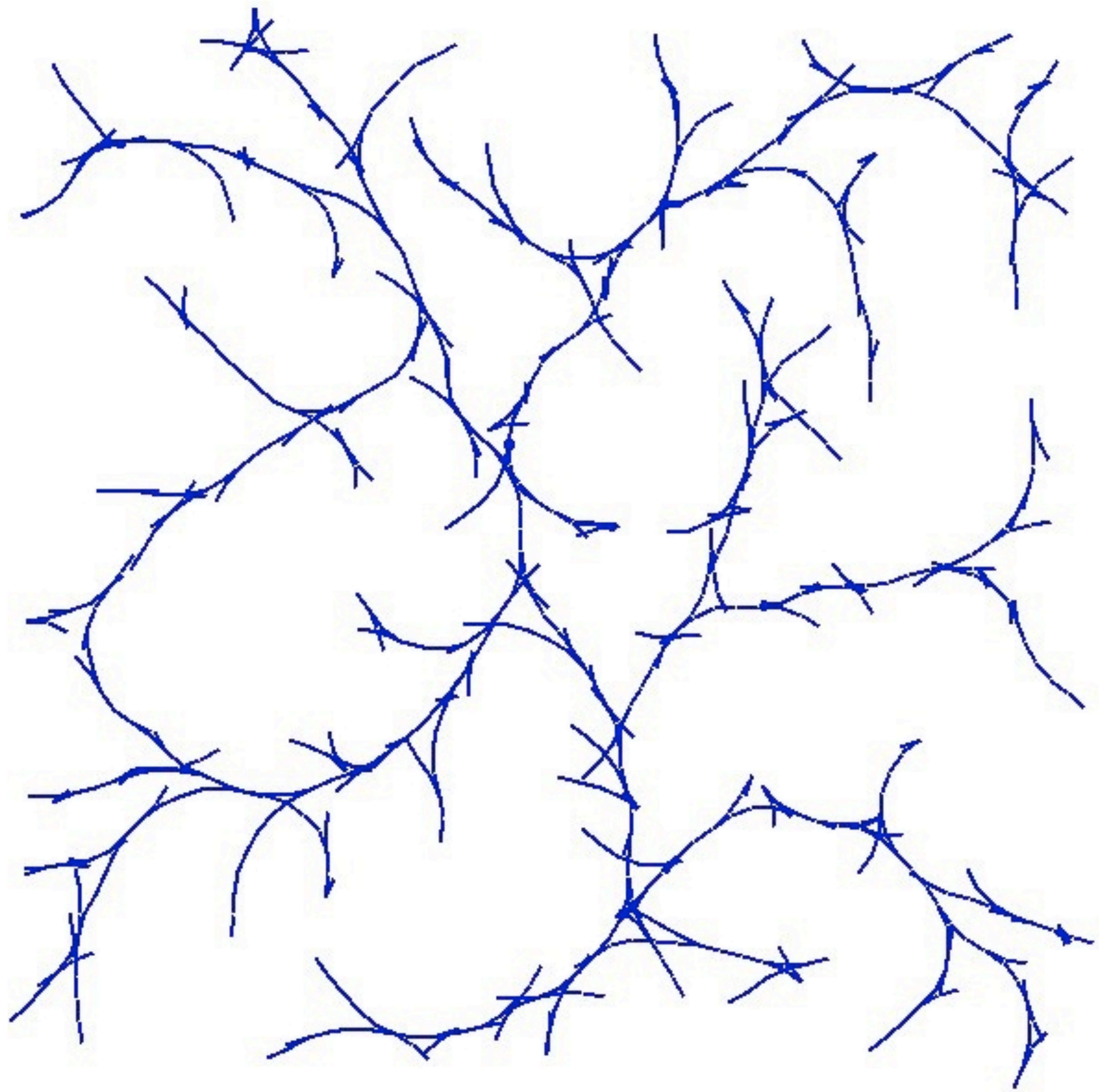




RRT example



RRT for a car (3 dof)



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: "rewire" tree during growth by testing connectivity to more than just closest node
- Optional: grow forward and backward