KTH ROYAL INSTITUTE
OF TECHNOLOGY

Lecture 7: Planning

Petter Ögren

Reminder!

A reminder for everyone to register in Kattis as well as click the "Join the session" button as described here (bottom of the page):

https://canvas.kth.se/courses/28858/pages/assignments?module_item_id=340574

If you do not do this we can not see your submission and you WILL NOT RECEIVE A GRADE!

Today!

Tomorrow at 17:00-18:00 I will go through the registration list and enter everyone that is in there into Canvace by checking the "Hello World - check your Kattis registration" assignment as passed.

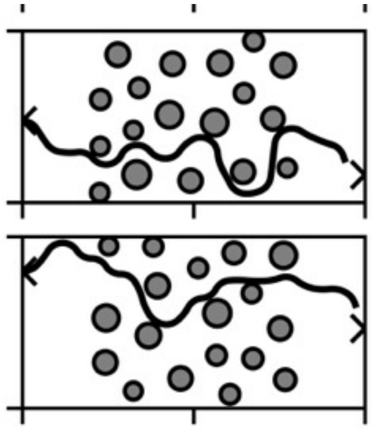
Please register before that so you can get the assurance we can see and grade you for the coming deadlines.

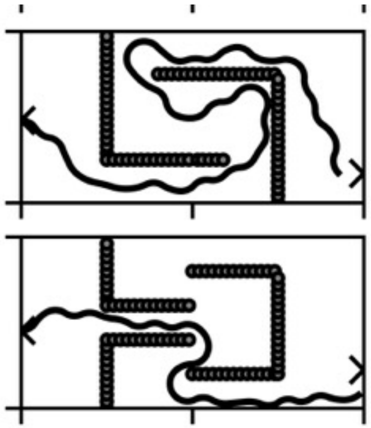
CANVAS

View announcement | Update your notification settings

When does a Robot need to do Planning?

- To go from A to B
- To grasp object O
- To assemble an object
- Note: Planning horizon ←→ Predictability
- In this lecture we assume the world is static





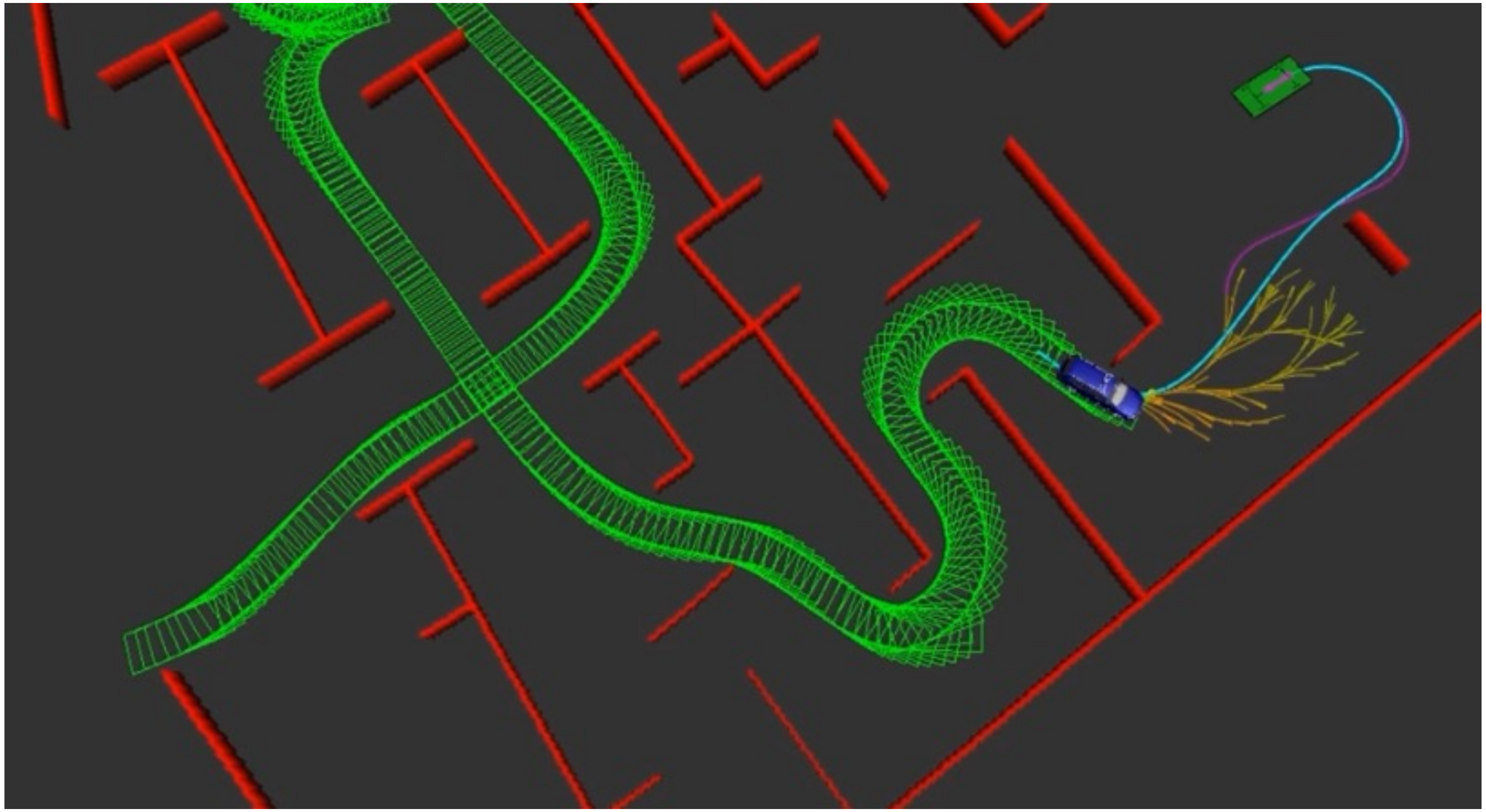
In general: Path planning is hard

- A complete algorithm finds a path if one exists and reports no path exists otherwise.
- Several variants of the path planning problem have been proven to be NP-hard.
- A complete algorithm may take exponential time.
- → We usually have to settle for “Good Enough” algorithms

Planning for Autonomous Driving

How is this done?

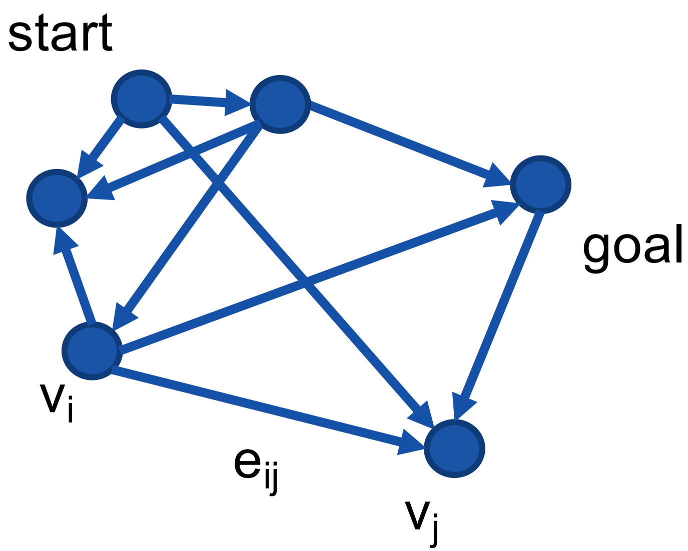
Planning for Autonomous Driving



How is this done?

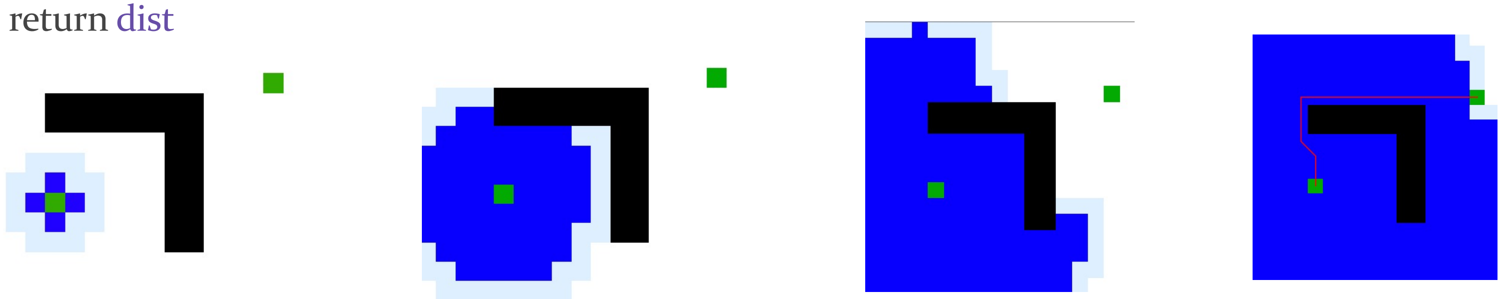
The foundation for many planning algorithms

- Shortest Path in a Graph
- A Graph G=(V,E)
  - Vertices (v_i)
  - Edges $e_{ij}=(v_i, v_j)$
  - Costs c_{ij}
- Solved by
  - Dijkstas algorithm
  - A*



Dijkstras Algorithm (dynamic programming)

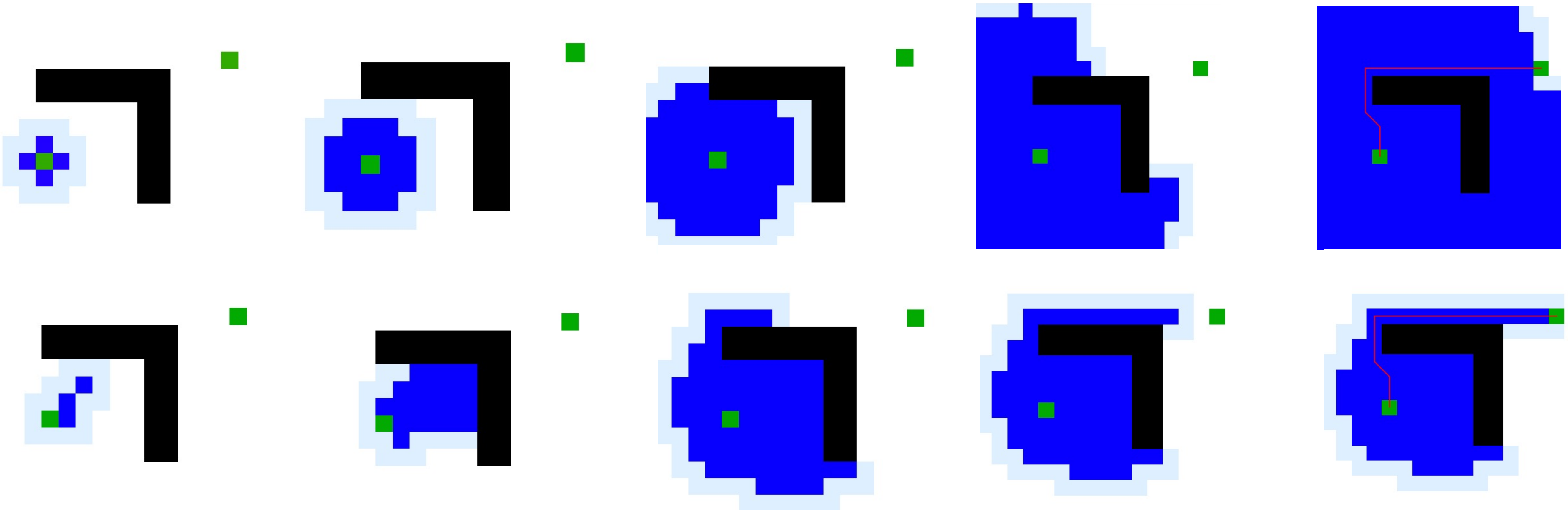
```
dist[s] ←0                                      (distance to source vertex is zero)
for all v ∈ V–{s}
        do dist[v] ←∞                           (set all other distances to infinity)
C←Ø                                             (C, the set of closed vertices is initially empty)
Q←V                                             (Q, the queue initially contains all vertices)
while Q ≠Ø                                      (while the queue is not empty)
do  u ← argmin(Q,dist)                          (select the element of Q with the min. distance)
      C←C∪{u}                                           (add u to list of closed vertices)
      for all v ∈ neighbors[u]
            do if  dist[v] > dist[u] + w(u, v)  (if new shortest path found)
                  then   d[v] ←d[u] + w(u, v)   (set new value of shortest path)

                                                (if desired, add traceback code)

return dist
```



Dijkstras vs A*

The closed set grows
- As a Circle in Dijkstra
- As an ellipsoid in A*

```
dist[s] ←o
for  all v ∈ V–{s}
        do  dist[v] ←∞
C←∅
Q←V
while Q ≠∅
do   u ← argmin(Q,dist + heur(u, start) )
      C←C∪{u}
       for all v ∈ neighbors[u]
              do  if   dist[v] > dist[u] + w(u, v)
                         then      d[v] ←d[u] + w(u, v)
```

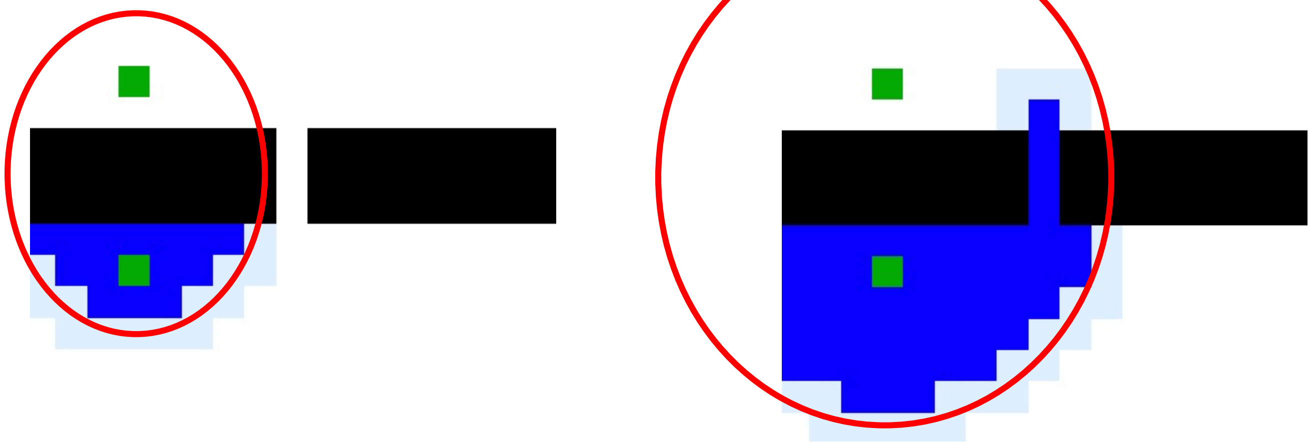


Dijkstras vs A*

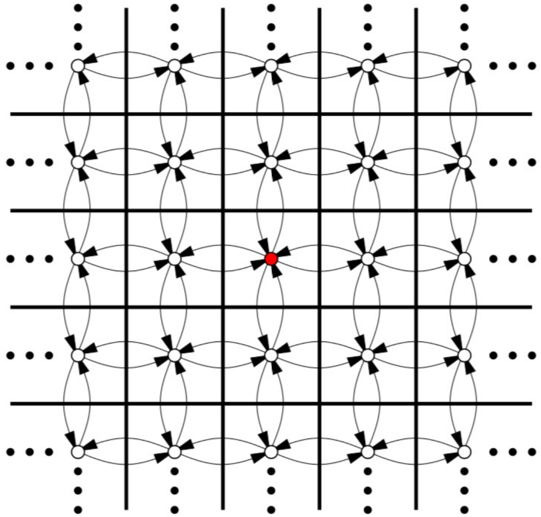
The closed set grows

- As a Circle in Dijkstra
- As an ellipsoid in A*

```
dist[s] ←o                                        (distance
for  all v ∈ V–{s}
       do  dist[v] ←∞                              (set all ot
C←∅                                               (S, the se
Q←V                                               (Q, the q
while Q ≠∅                                        (while th
do   u ← argmin(Q,dist + heur(u, start) )         (select th
     C←C∪{u}                                      (add u to
      for all v ∈ neighbors[u]
            do  if   dist[v] > dist[u] + w(u, v)  (if new sl
                      then      d[v] ←d[u] + w(u, v)  (set new

                                                  (if desire
```

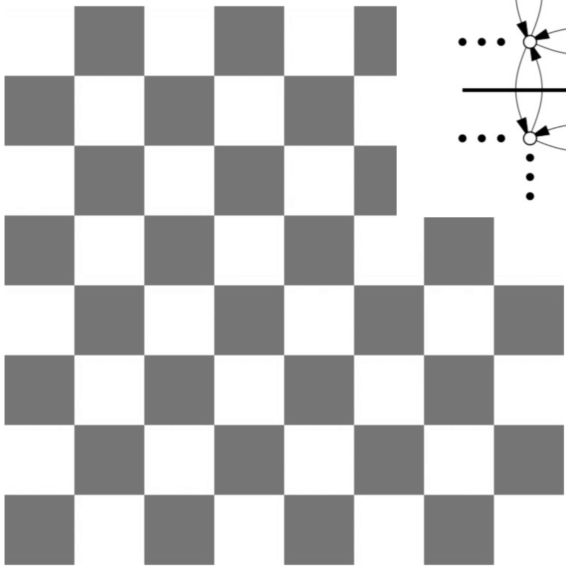


How do we use A*?



- Graph?
- Costs?

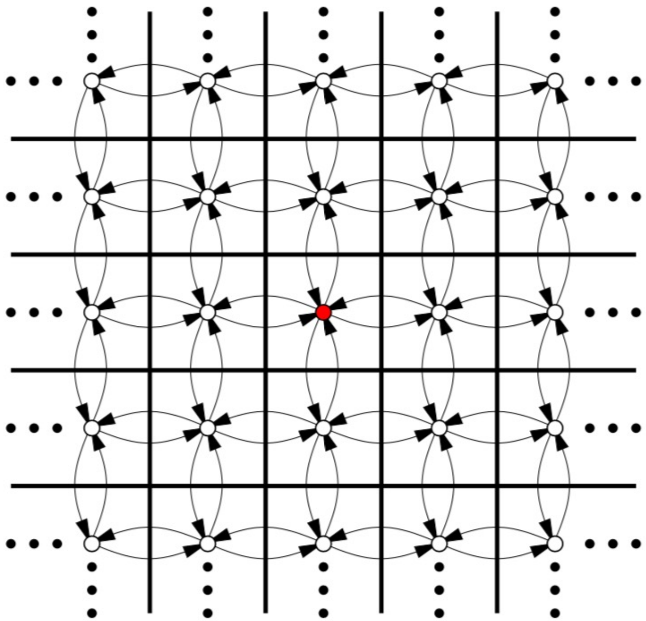
- What chess piece has the graph on the right?



How do we use A*?

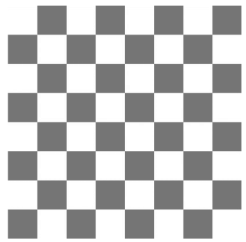
- Graph?
- Costs?

- For a Robot…

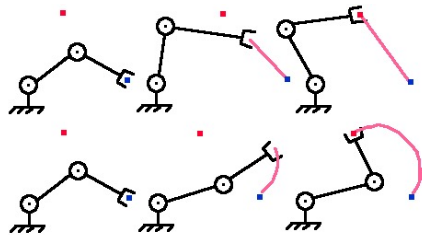


How do we use A*?

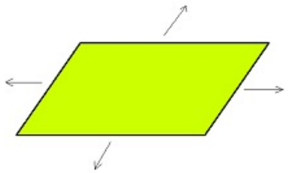
- Graph?
- Costs?

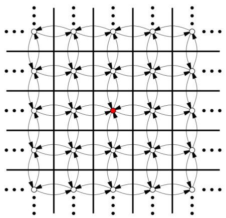


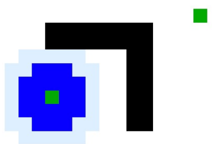
- Problems
  - Robot is not a point (size)
  - Robot does not live on R^2 (manipulator, drone)
  - Motion is restricted (car)

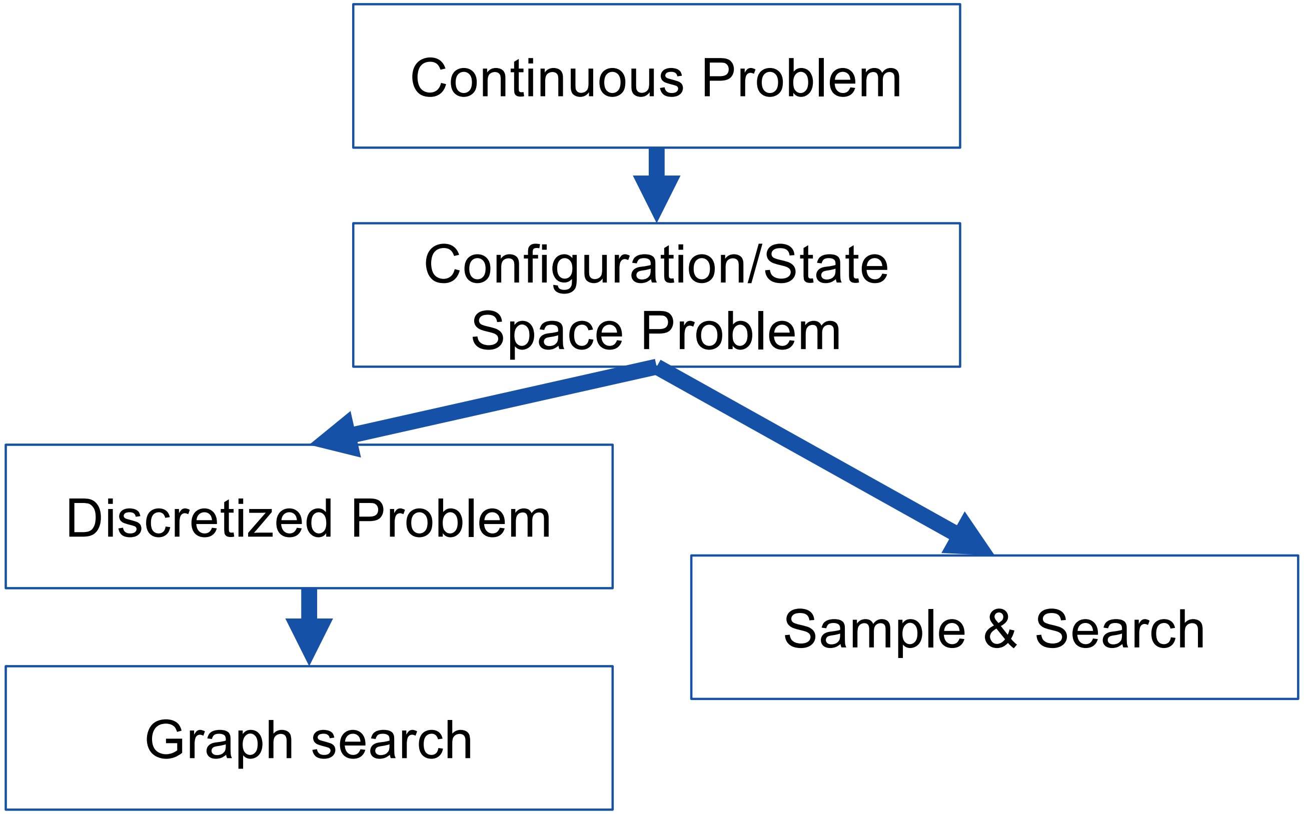


Common Path Planning Approach





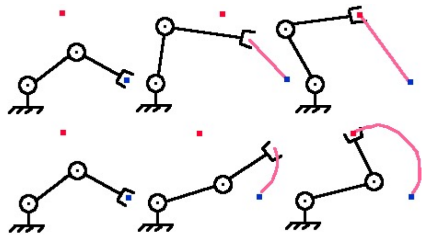




Configuration and State Spaces

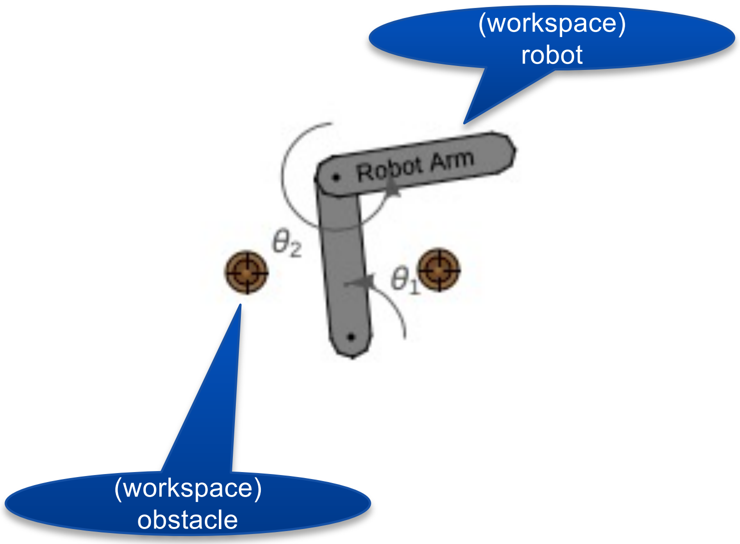
- **Configuration**: A complete specification of **every position** of the system
  - Ex: (x, y, theta) of a car
  - Configuration space (C-space)
    - > *space where conf. lives*
  - Ex: R^3 or R^2

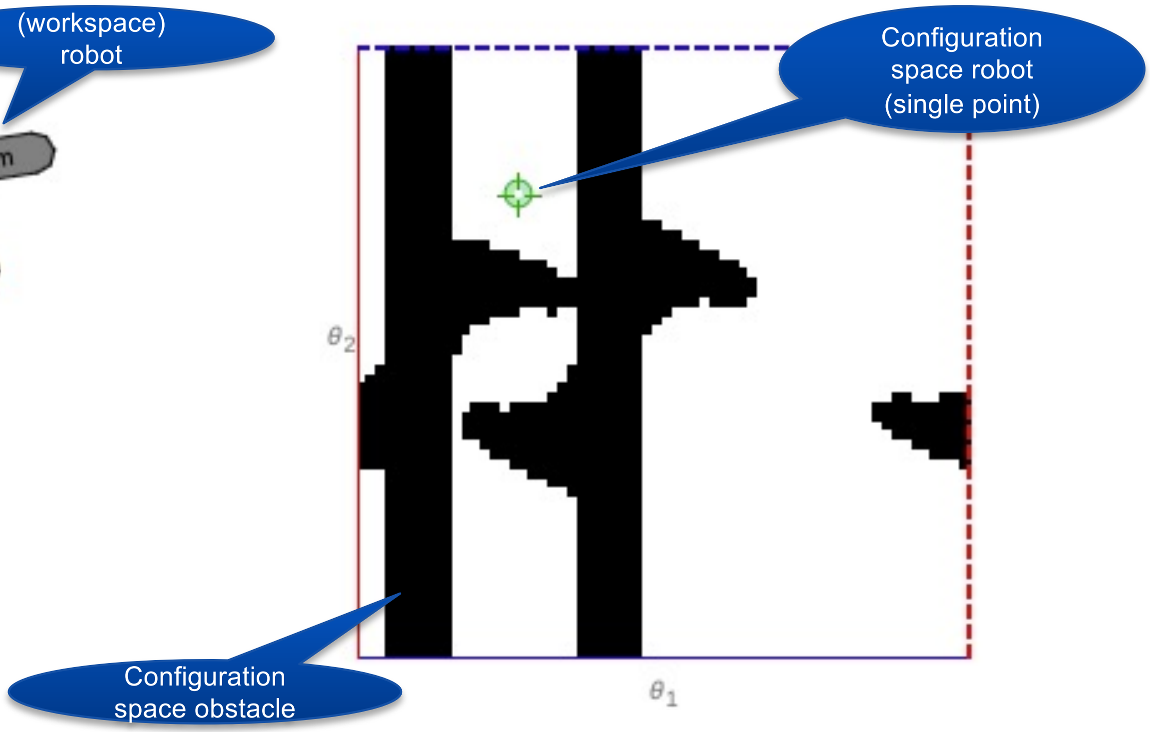
- **State**: A complete specification of the system
  - Ex: (x,y, theta, velocity) of a car
  - Configuration space is subset of State space



2 link manipulator

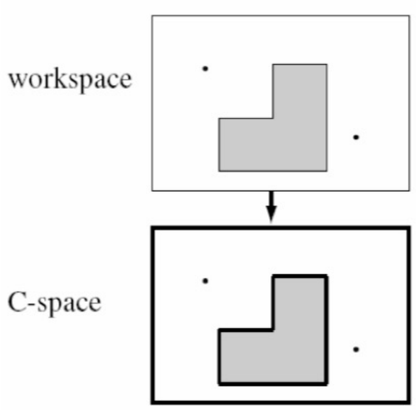
- **Workspace**: 3D space around robot
- **Configuration space**: A complete specification of **every position** of the system





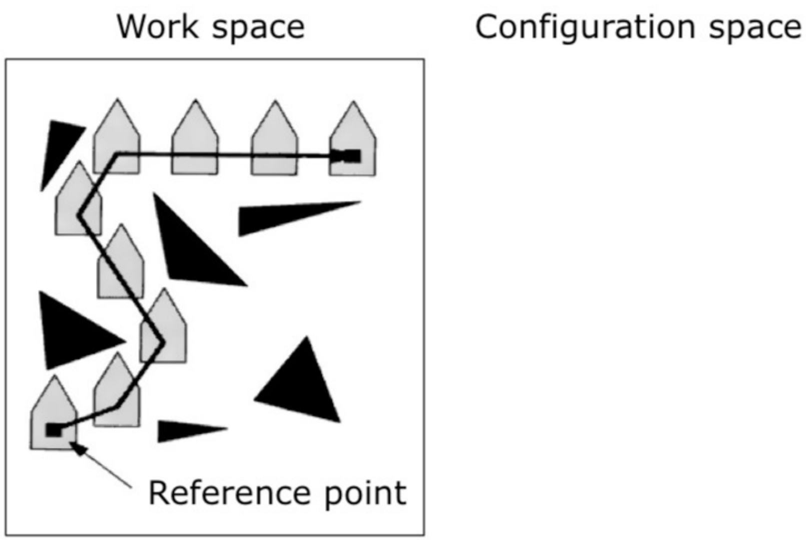
Configuration Space Obstacles (CSO)

- What is a CSO?
- Part of C-space that induces a collision somewhere

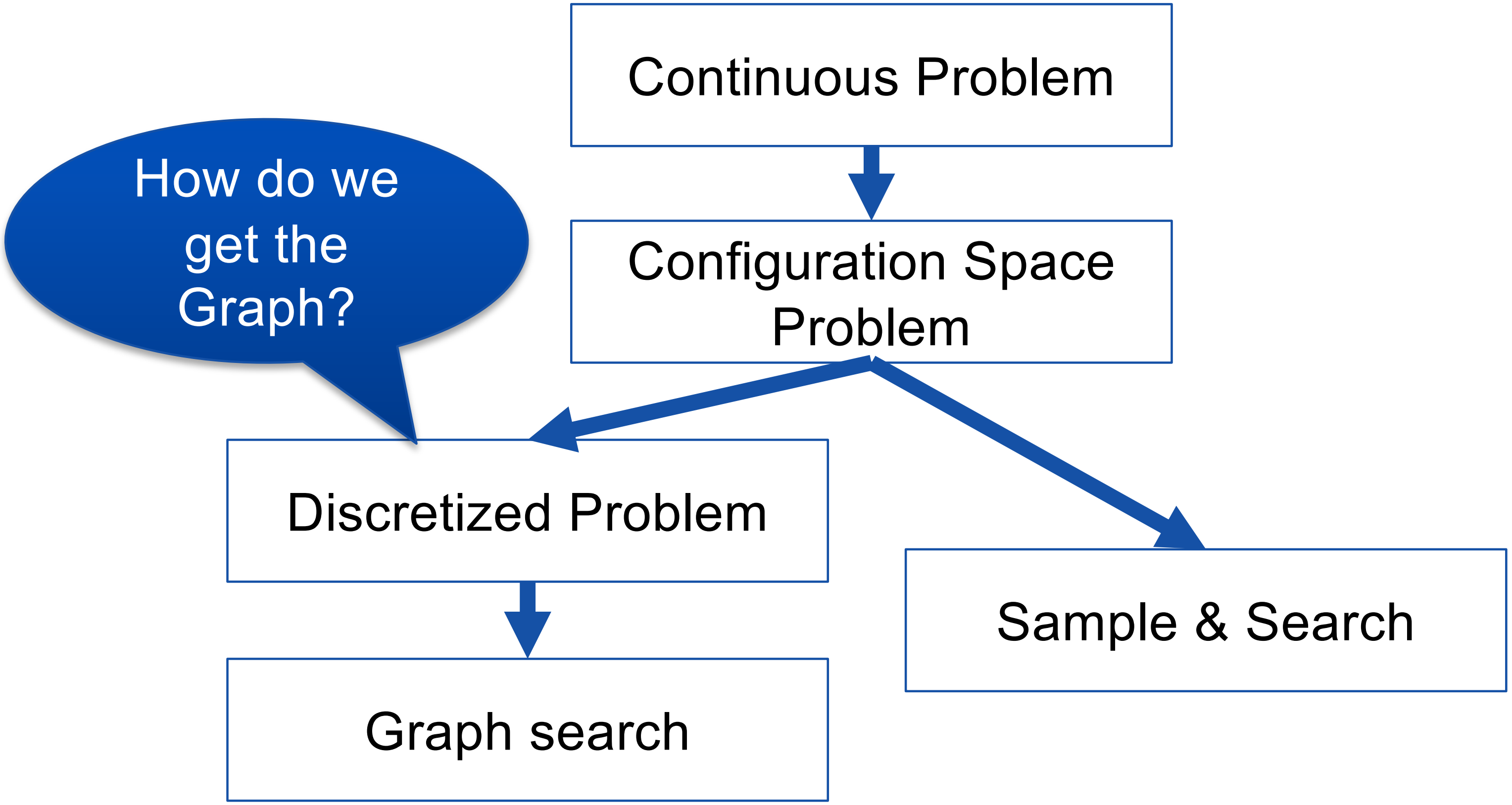


Configuration Space Obstacles (CSO)

- What a CSO?
- Part of C-space that induces a collision somewhere

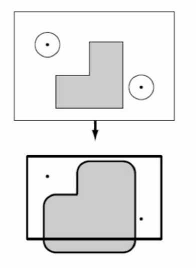


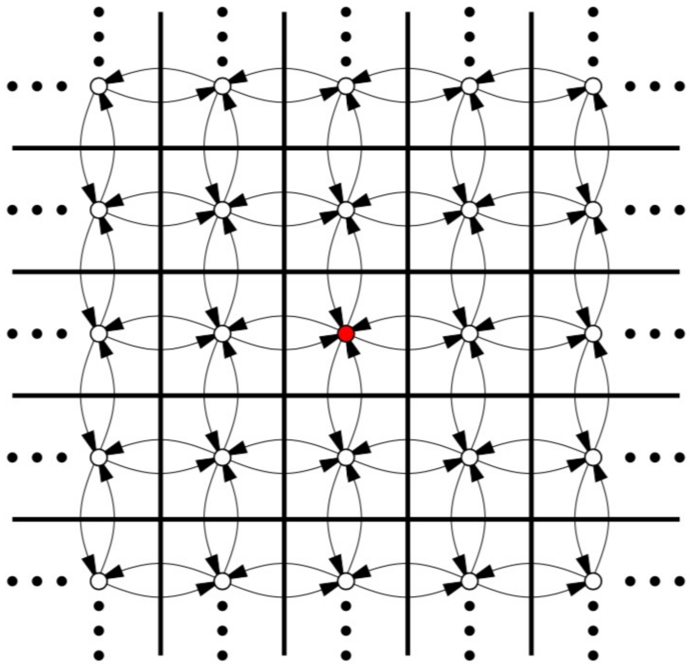
Common Path Planning Approach



How to make a Graph from C-space?

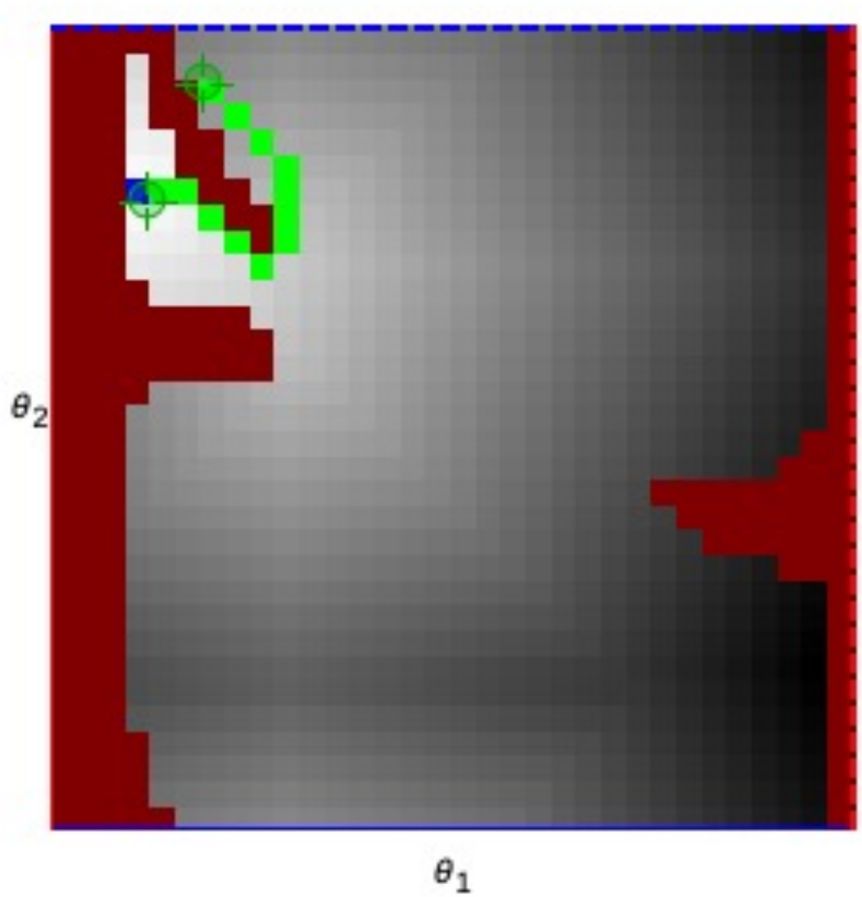
A grid





Solving A* on the Grid Graph

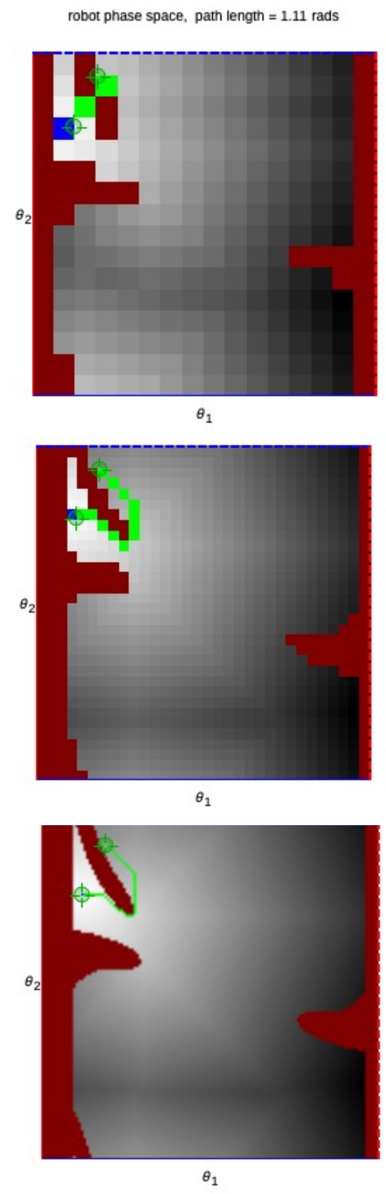
7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15



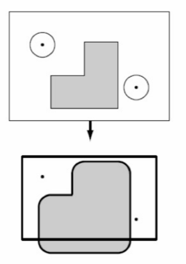
How small should we make the grids?

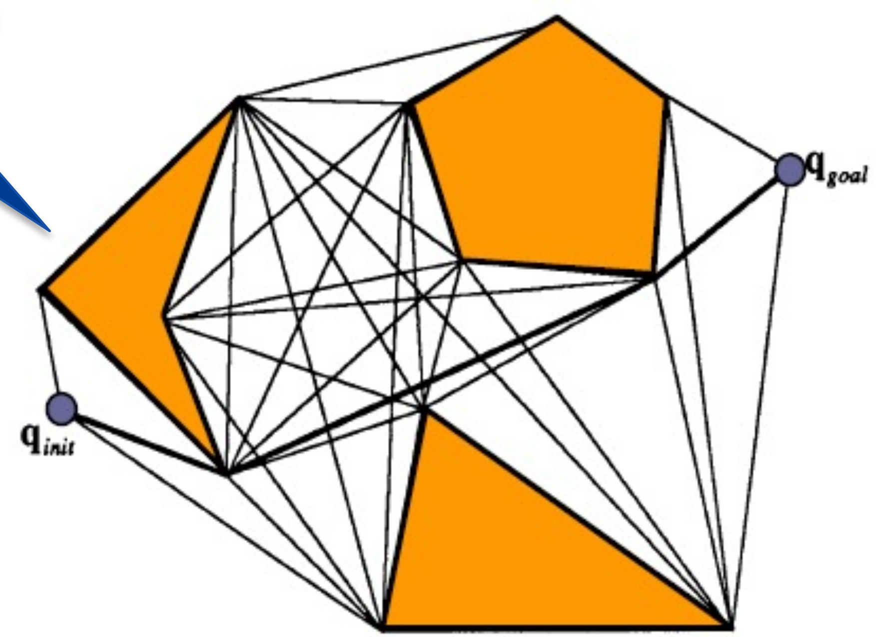
- Tradeoff
  - Reduce Computation (use large grids)
  - Improve Accuracy (use small grids)
    - > *Fake paths appear*
    - > *Real paths disappear*
  - Note:
    - > *Smaller grids do not give near-optimal paths*

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15



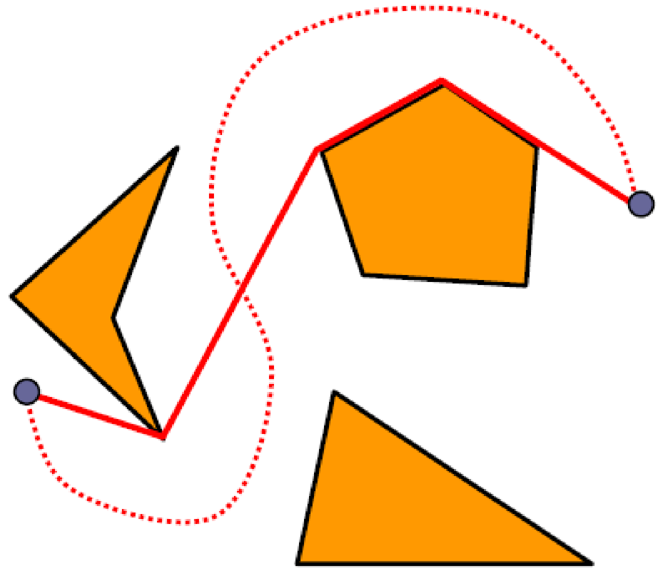
How to make a Graph from C-space?



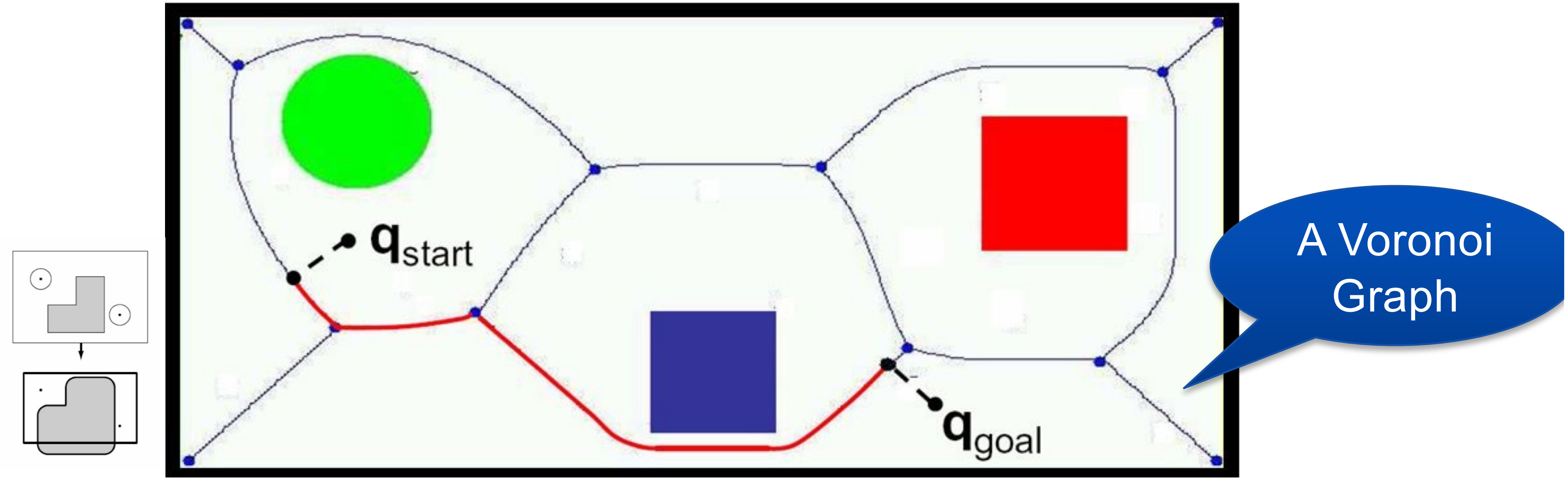


How to make a Graph from C-space?

- Observation:
  - Shortening any path gives a visibility graph path
  - Advantages?
  - Drawbacks?

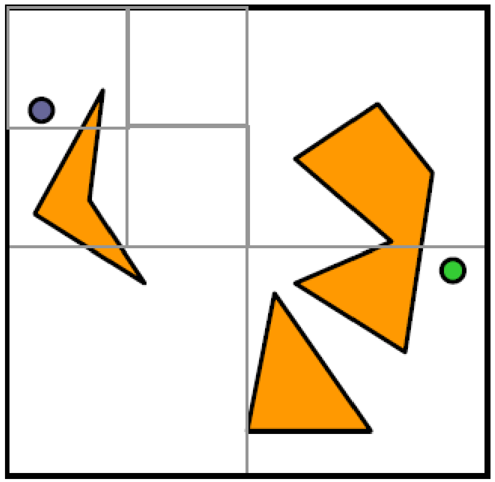


How to make a Graph from C-space?

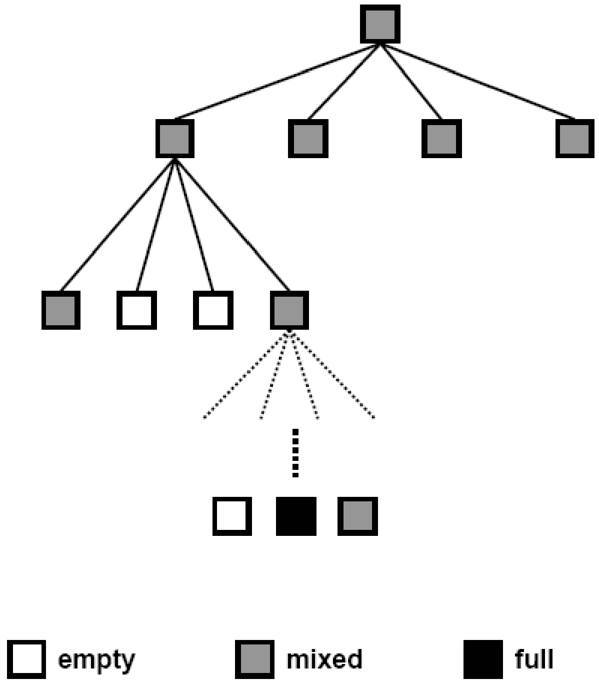


- Points that have equal distance to the two closest obstacles
- Advantages?
- Drawbacks?

High resolution in narrow areas
Low resolution in open areas…

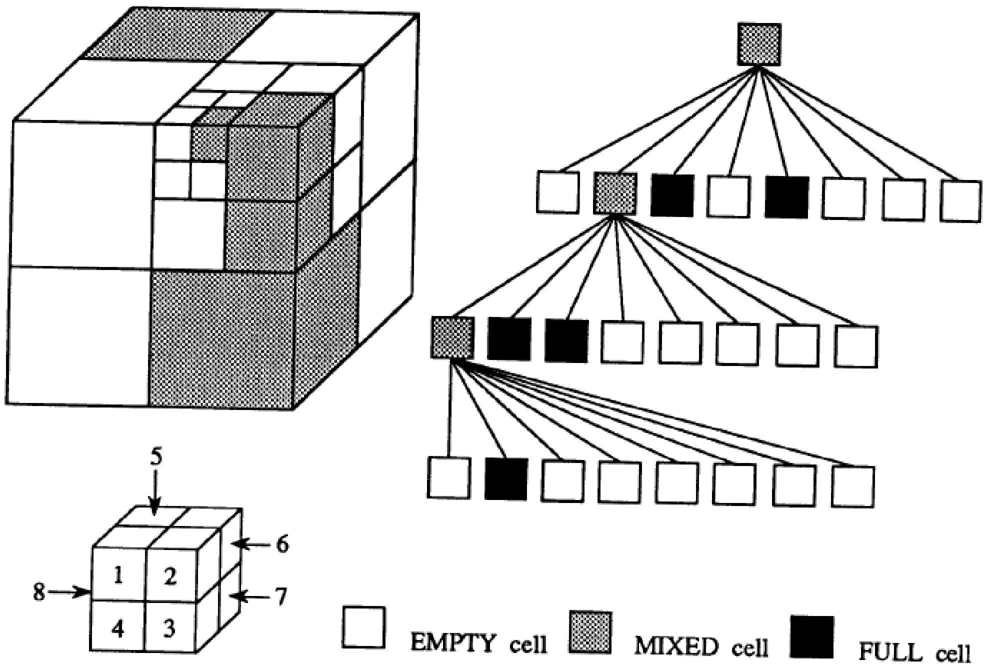


Quadtree
Decomposition



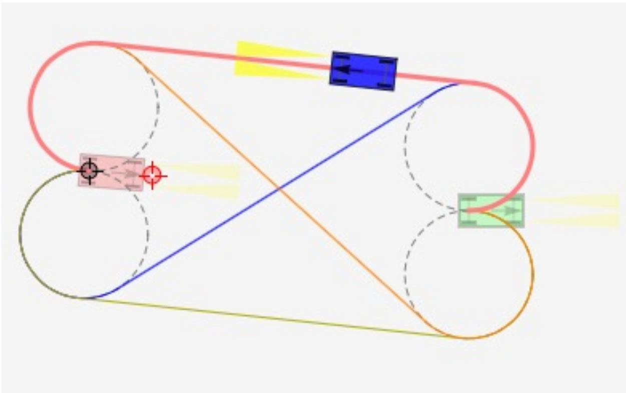
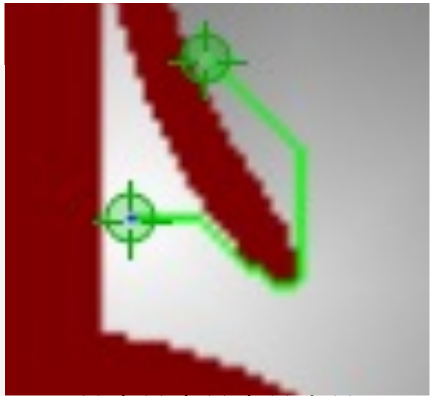
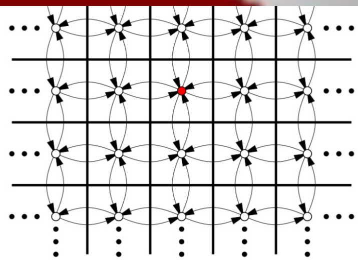
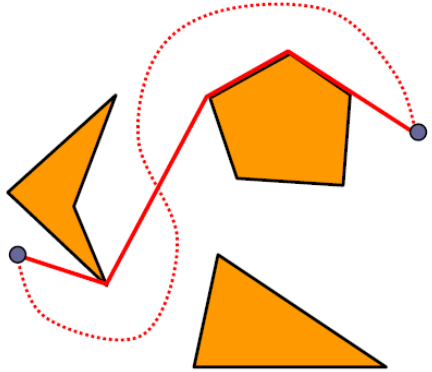
High resolution in narrow areas
Low resolution in open areas…

Octree
Decomposition



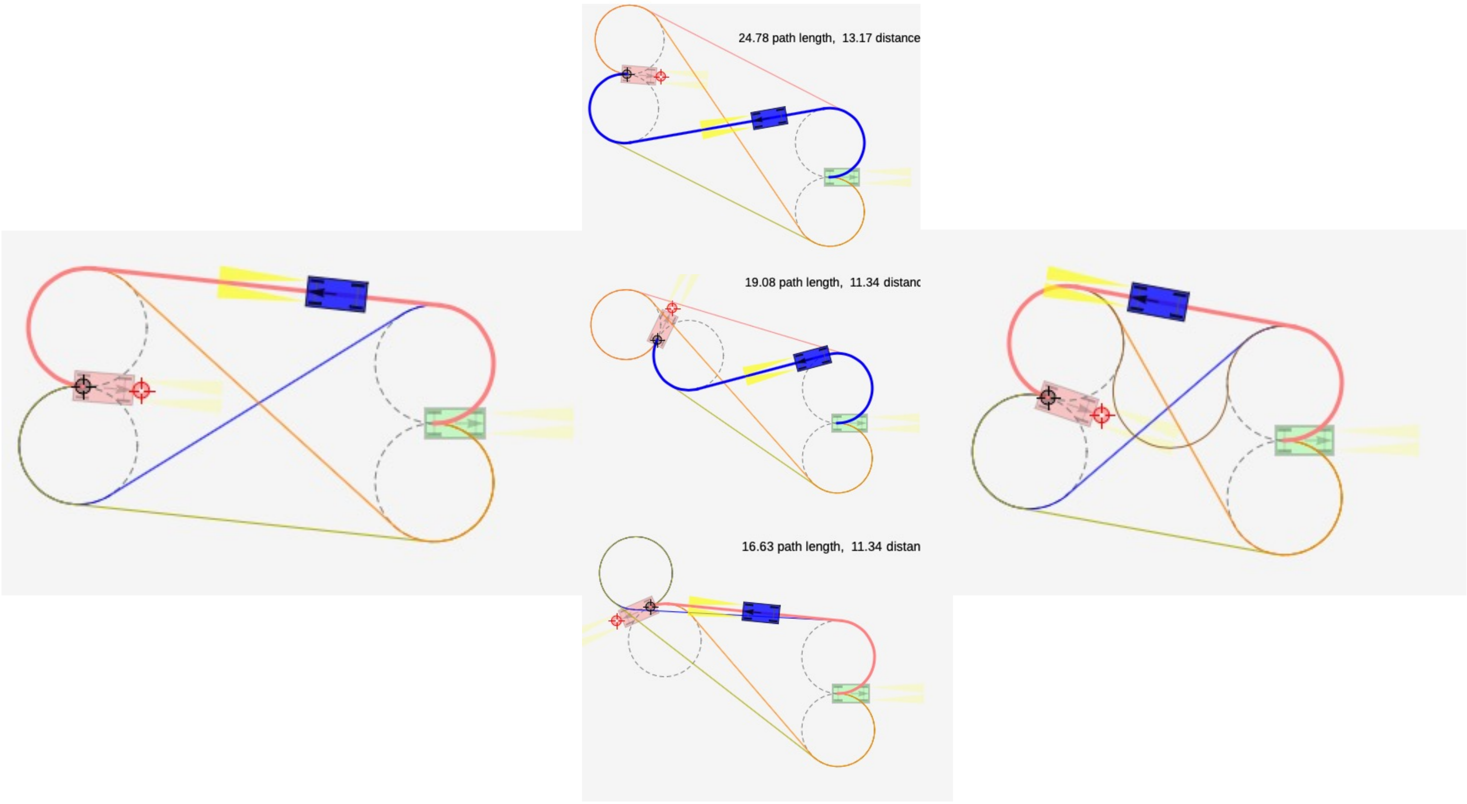
What about undrivable trajectories?

- Can a car drive any path?

Dubins car

- The optimal path for a car (with no obstacles) can be created using at most 3 circles and 1 straight line



Can we fix an undrivable path?

Plan and Transform

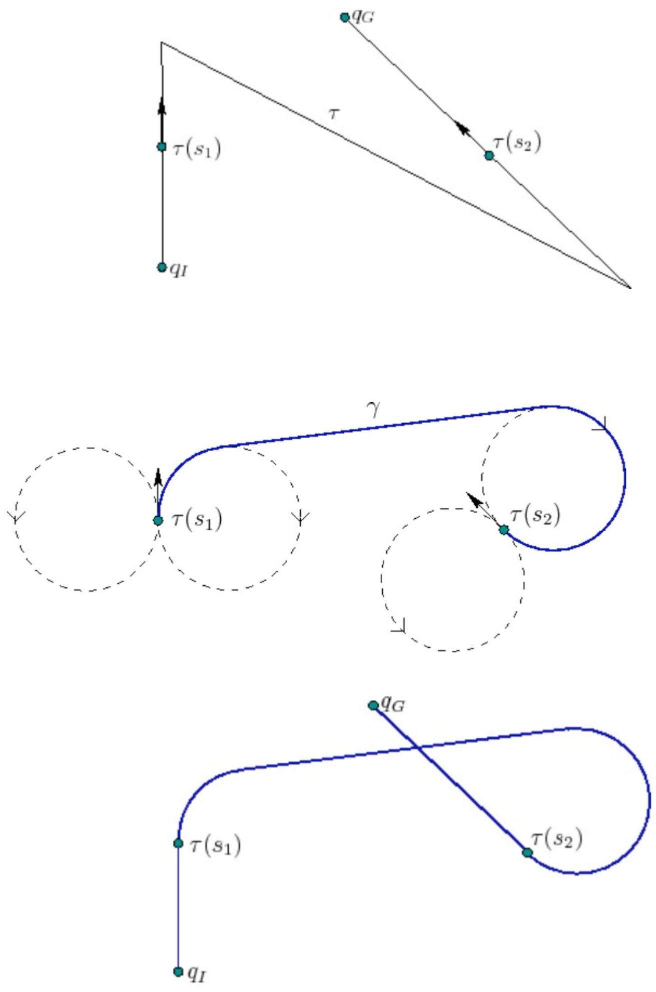
Algorithm

1. Plan a short non-traversable path
2. Pick two points on path
3. Connect with traversable sub-path
4. Iterate from 2, until whole path is traversable

- Not always possible
- Hard to know when to stop
- Can yield very good solutions for visibility graph

not

possible



Common Path Planning Approach

Two Problems:

- How small to make the grids?
- Is the graph drivable?

Continuous Problem

Configuration Space Problem

Discretized Problem

Graph search

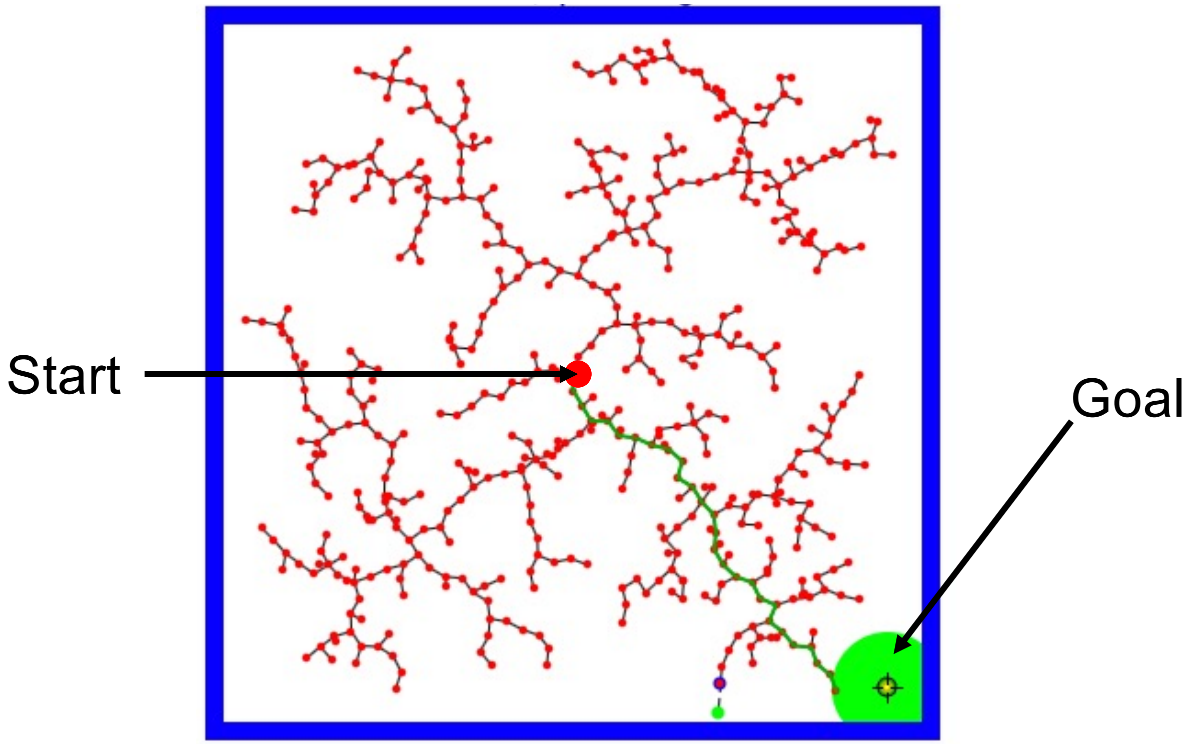
Sample & Search

Can we create the graph and search at the same time?

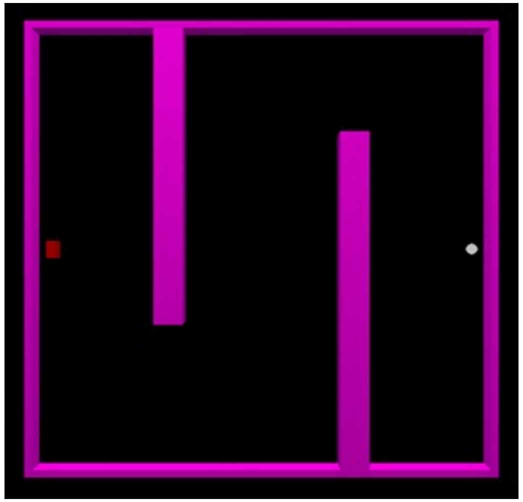
Make the graph traversable!

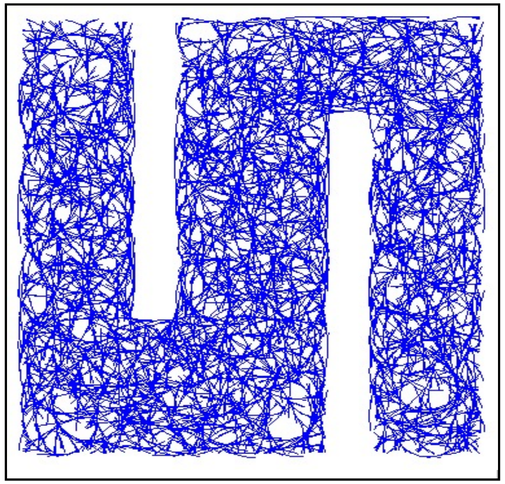
Sample & Search: RRT

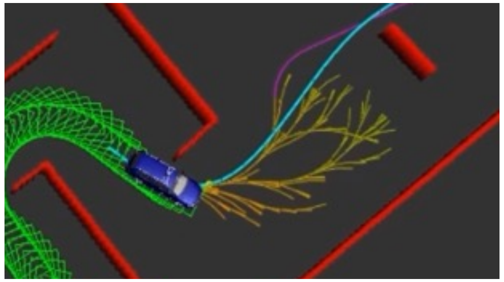
RRT: Rapidly Exploring Random Trees



Example: Simple RRT Planner

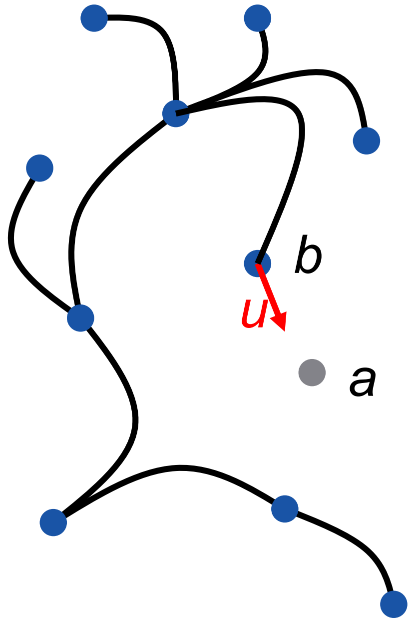






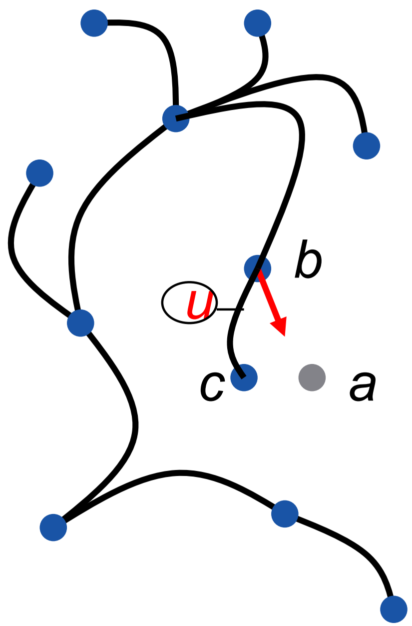
Building an RRT

- To extend an RRT:
  - Pick a **random** point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*



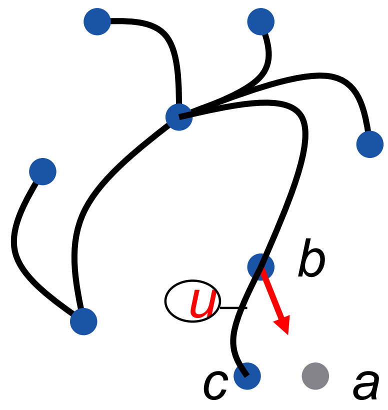
Building an RRT

- To extend an RRT (cont.)
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge



RRT Algorithm

- To extend an RRT
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find **control inputs *u* to steer the robot from *b* to *a***
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge



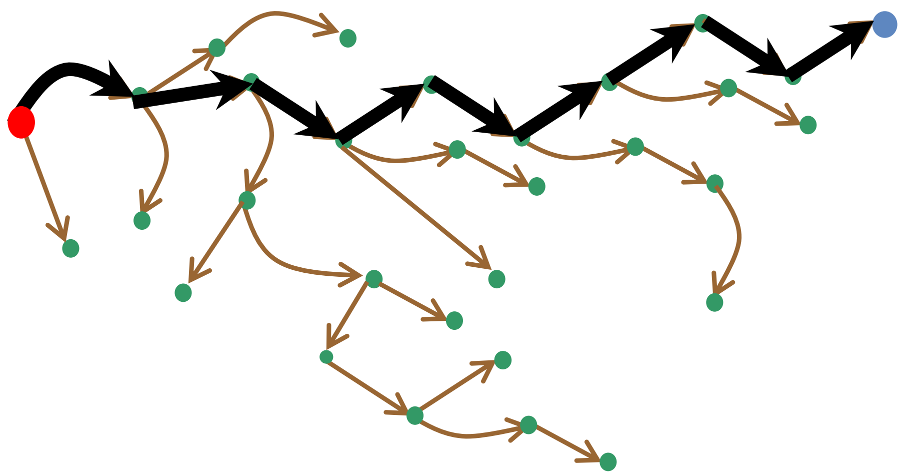
2 Grid Problems:
- How small to make the grids?
- Is the graph drivable?

RRT
- Resolution improves over time
- Drivable by design

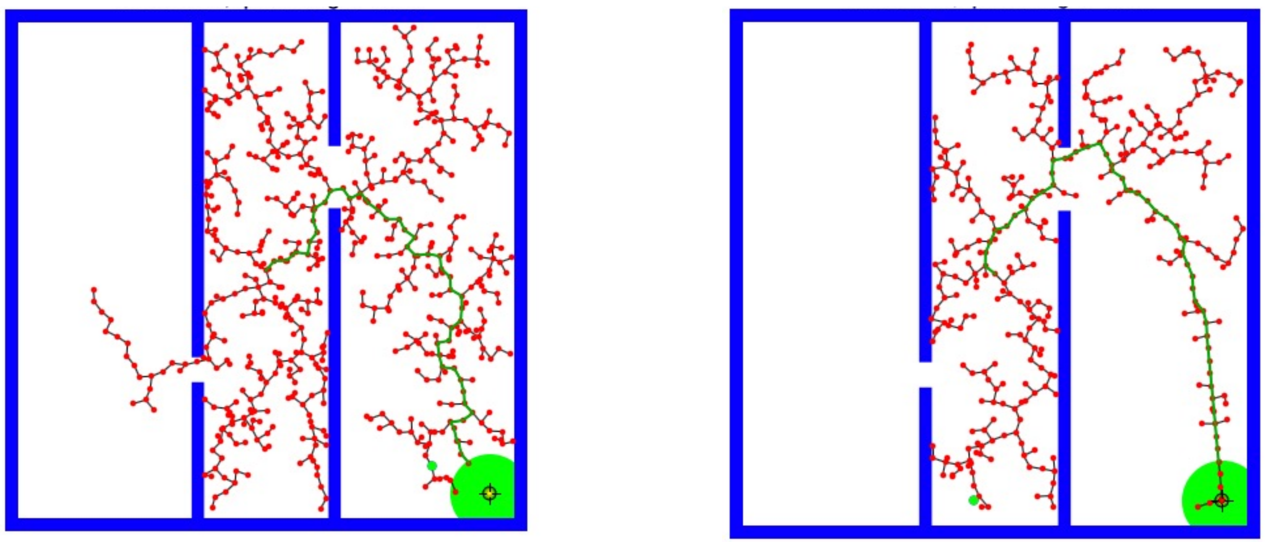
Executing the Path

Once the RRT reaches s_{goal}

- Backtrack along tree to identify edges that lead from s_{start} to s_{goal}
- Drive robot using control inputs stored along edges in the tree



Example: Simple RRT Planner

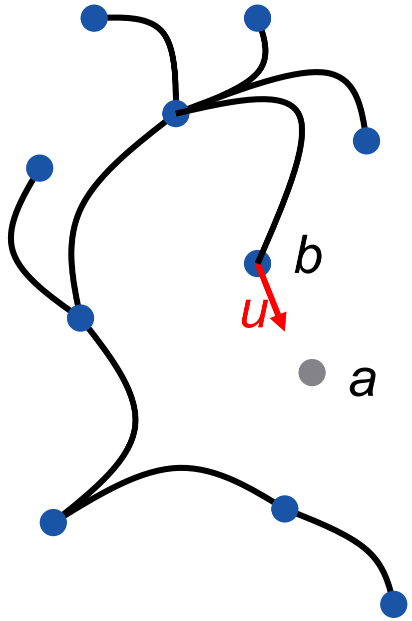


- Problem: ordinary RRT explores *X* uniformly
  - slow convergence
- Solution: bias distribution towards the goal
  - Pick the goal point with X% probability

Building an RRT

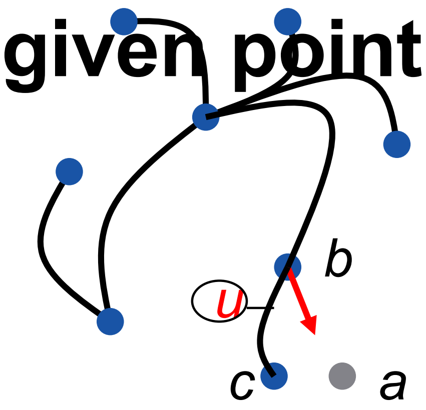
Bias random points towards goal!
I.e. pick the goal every 10th time…

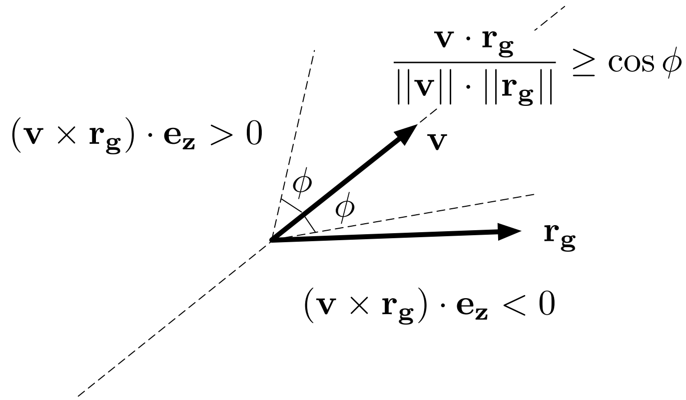
- To extend an RRT:
  - Pick a **random** point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*



Steering a Car towards a given point

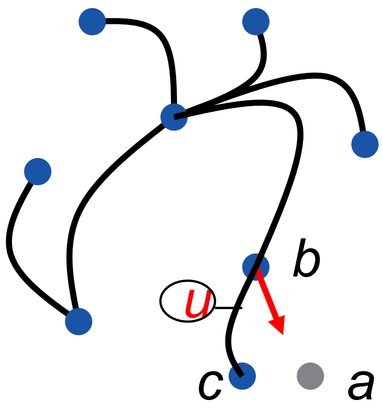
- To extend an RRT
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find **control inputs *u* to steer the robot from *b* to *a***
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge





Things to think about…

- To extend an RRT
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge



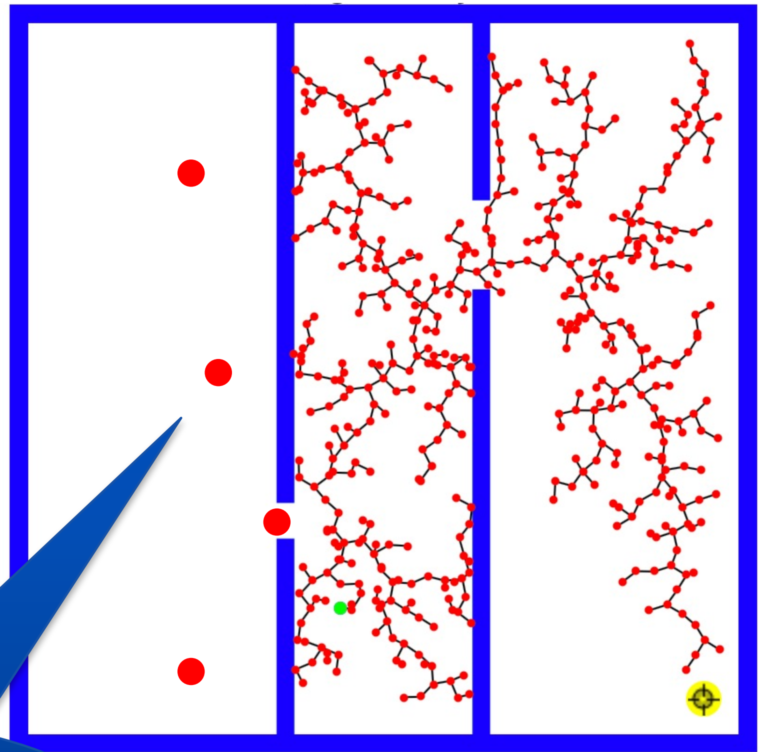
- (x,y)?
- (x,y,theta)?
- (x,y,theta,v)?

Closest in what sense?

Things to think about…

- To extend an RRT
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge

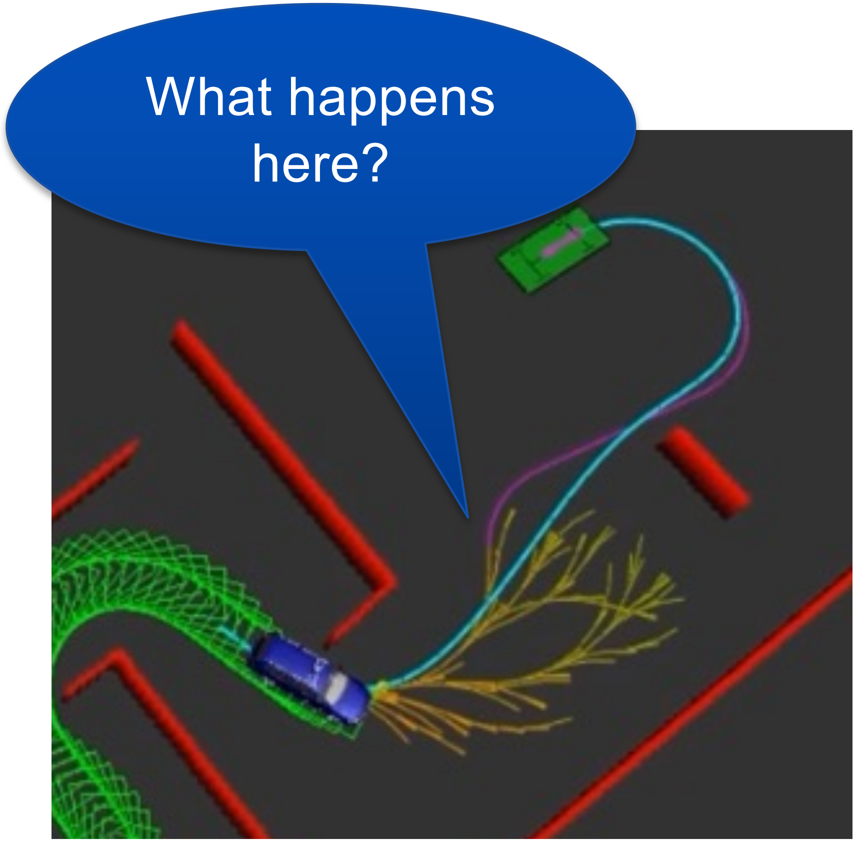
Why are there no Nodes here



Consider sampling bias
- In narrow gaps
- Along optimal grid path
- …

Things to think about…

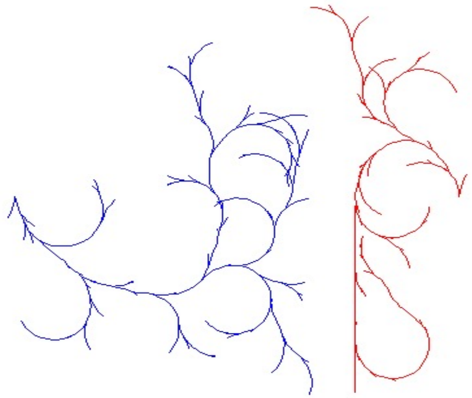
- To extend an RRT
  - Pick a random point *a* in *X*
  - Find *b*, the node of the tree closest to *a*
  - Find control inputs *u* to steer the robot from *b* to *a*
  - Apply control inputs *u* for time δ, so robot reaches *c*
  - If no collisions occur in getting from *a* to *c*, add *c* to RRT and record *u* with new edge



- Check if c can be connected to goal using Dubins Trajectory (purple)
- If so done!
- Or post process to get smooth blue

Additional improvement: Bidirectional Planners

- Build two RRTs, from start and goal state



- Complication: need to connect two RRTs
  - **bias** the distribution, so that the trees meet

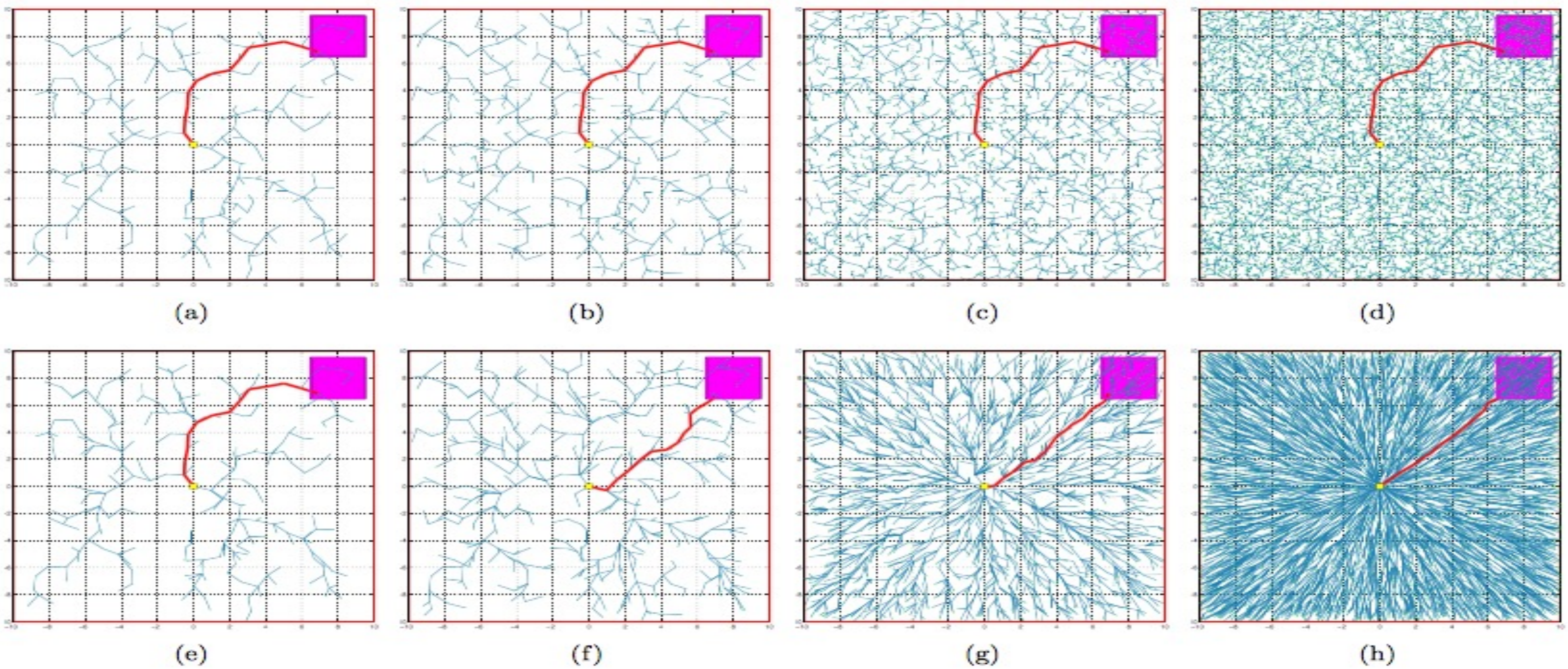
Some notes on RRT

- RRT finds **one** solution with probability →1
  - Quality is not perfect…

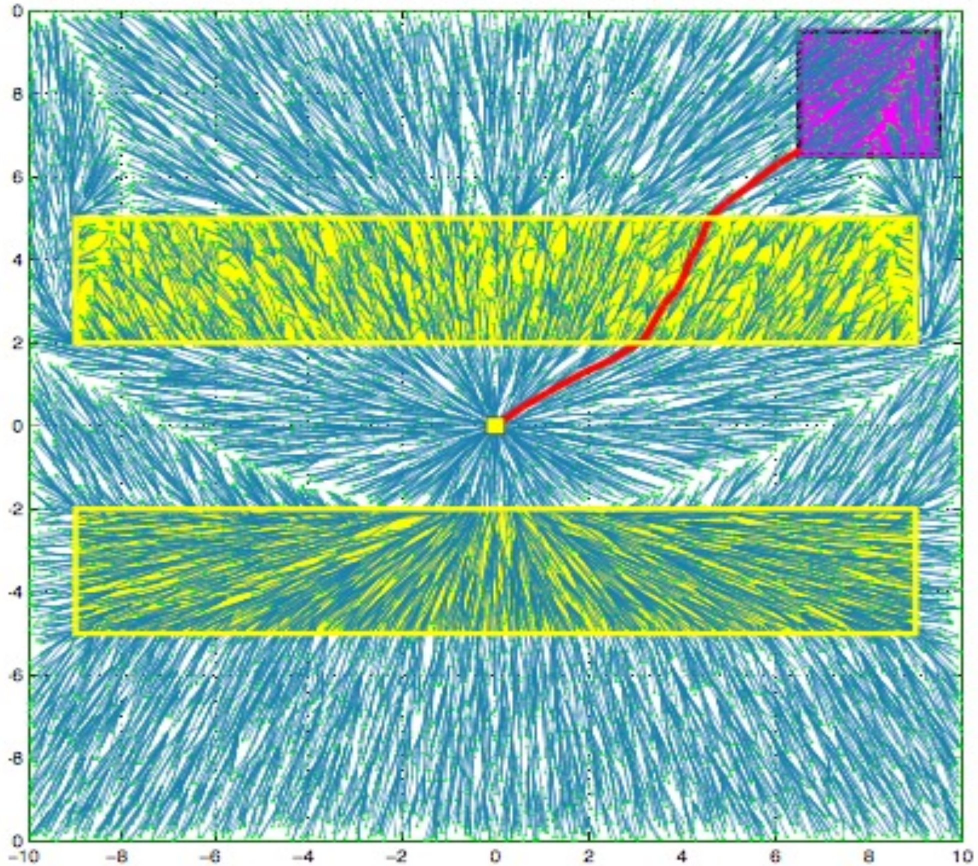
- Brake through in 2011 (Karaman and Frazzoli)
  - RRT*

- RRT* finds **optimal** solution with probability → 1

RRT vs RRT* (Karaman and Frazzoli)



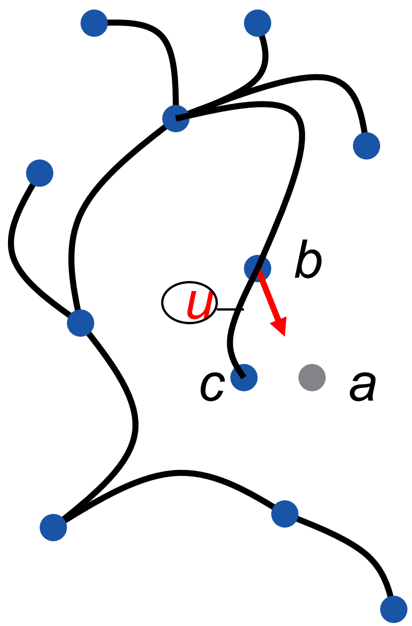
RRT* (High cost and Low cost regions)



How does the RRT* work?

Same start as RRT...

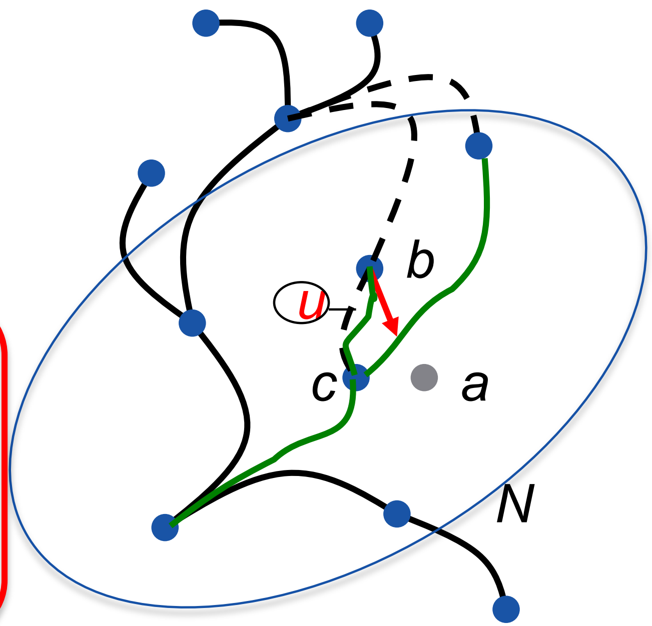
- Pick a **random** point ***a*** in *X*
- Find ***b***, the node of the tree closest to ***a***
- Find control inputs ***u*** to steer the robot from *b* to *a*
- Apply control inputs *u* for time δ, so robot reaches *c*
- If no collisions occur in getting from *a* to *c*, ~~add *c* to RRT and record *u* with new edge~~



How does the RRT* work?

Same start as RRT...

- Pick a **random** point ***a*** in *X*
- Find ***b***, the node of the tree closest to ***a***
- Find control inputs ***u*** to steer the robot from *b* to *a*
- Apply control inputs *u* for time δ, so robot reaches *c*
- If no collisions occur in getting from *a* to *c*
  - > *Find set of Neighbors N of c*
  - > *Choose Best parent!*
  - > *Try to adopt Neighbors (if good)*



RRT* (2011, original)

a

b

c

Neighbors N

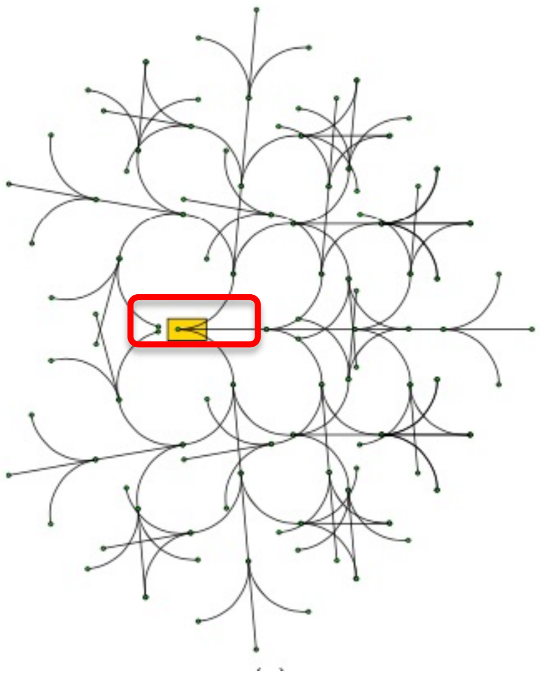
Find best parent

Adopt new children (if improvment)

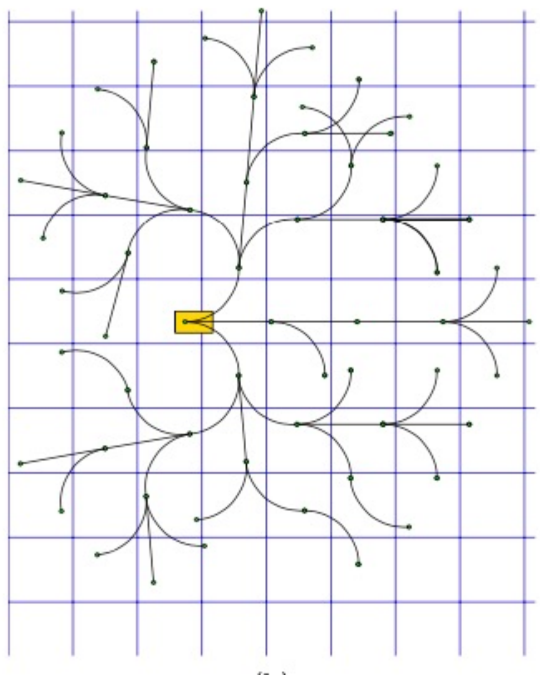
Algorithm 6: RRT*

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
    $V \leftarrow V \cup \{x_{\text{new}}\}$;
    $x_{\text{min}} \leftarrow x_{\text{nearest}}$; $c_{\text{min}} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Connect along a minimum-cost path
      if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\text{min}}$ **then**
        $x_{\text{min}} \leftarrow x_{\text{near}}$; $c_{\text{min}} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
    $E \leftarrow E \cup \{(x_{\text{min}}, x_{\text{new}})\}$;
    foreach $x_{\text{near}} \in X_{\text{near}}$ **do** // Rewire the tree
      if $\texttt{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge \texttt{Cost}(x_{\text{new}}) + c(\texttt{Line}(x_{\text{new}}, x_{\text{near}})) < \texttt{Cost}(x_{\text{near}})$
      then $x_{\text{parent}} \leftarrow \texttt{Parent}(x_{\text{near}})$;
      $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\}$
return $G = (V, E)$;

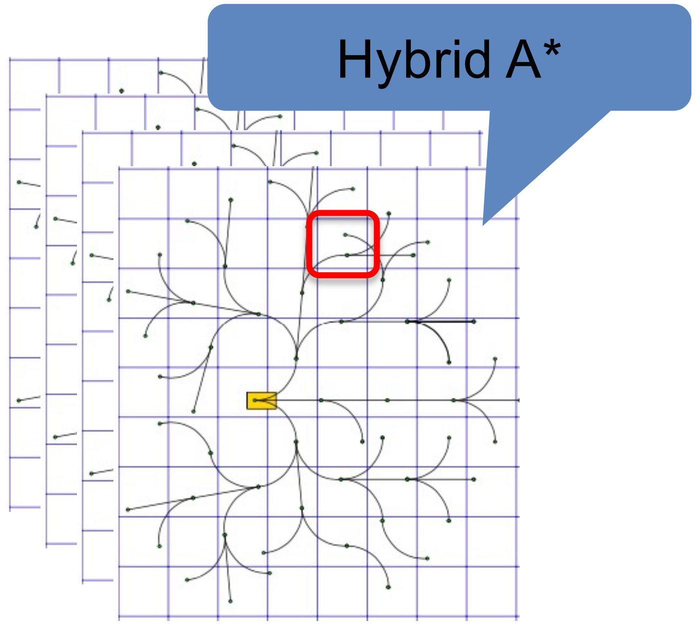
What if we create the graph online in A*?



If we just build a search tree we get copies of same state

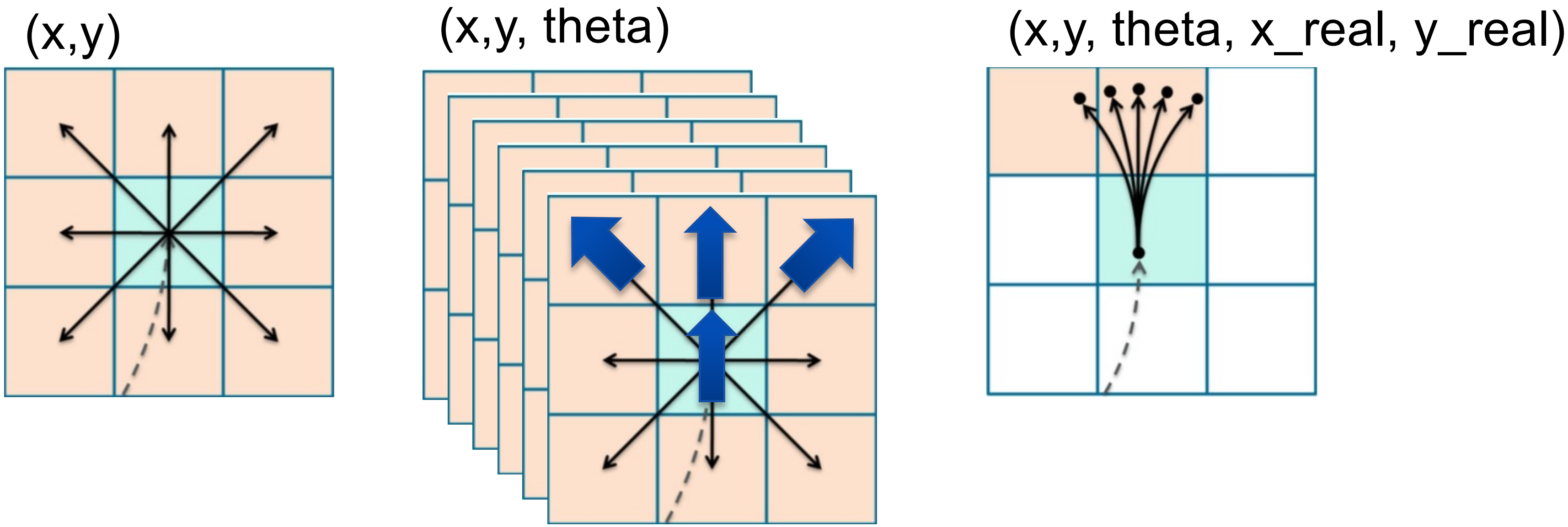


Allowing just one state in each grid



Allowing 4 states in each grid: theta=(0,pi/2,pi,3pi/2)

Hybrid A*



- How to make sure transitions are feasible?
- Allow positions that are not in center of grid -> **Hybrid A***

Hybrid A*

$$n = (\tilde{x}, \tilde{\theta}, x, g, f, n_{\mathrm{p}}) \ .$$

(grid_no (x,theta), actual_pos, cost, tot_cost_estimate, parent_node)

Algorithm 1 Standard version of Hybrid A*

procedure PLANPATH(m, μ, x_{s}, θ_{s}, G)
  $n_{\mathrm{s}} \leftarrow (\tilde{x}_{\mathrm{s}}, \tilde{\theta}_{\mathrm{s}}, x_{\mathrm{s}}, 0, h(x_{\mathrm{s}}, G), \text{-})$
  $O \leftarrow \{n_{\mathrm{s}}\}$
  $C \leftarrow \emptyset$
  while $O \neq \emptyset$ **do**
    $n \leftarrow$ node with minimum f value in O
    $O \leftarrow O \setminus \{n\}$
    $C \leftarrow C \cup \{n\}$
    if $n_x \in G$ **then**
      return reconstructed path starting at n
    else
      UPDATENEIGHBORS(m, μ, O, C, n)
    end if
  end while
  return no path found
end procedure

Standard A*

Key step

Hybrid A*

$$n = (\tilde{x}, \tilde{\theta}, x, g, f, n_{\mathrm{p}}) \ .$$

(grid_no, actual_pos, cost, tot_cost_estimate, parent_node)

for all desired heading changes

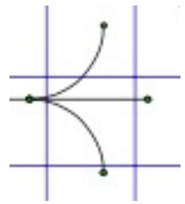
Add to closed if obstacle

If grid is non-empty

Replace node in open if cost improvement

Add to Open

procedure UPDATENEIGHBORS(m, μ, O, C, n)
  for all δ **do**
    $n' \leftarrow$ succeeding state of n using $\mu(n_\theta, \delta)$
    if $n' \notin C$ **then**
      if $m_{\mathrm{o}}(n'_{\tilde{x}}) =$ obstacle **then**
        $C \leftarrow C \cup \{n'\}$
      else if $\exists n \in O : n_{\tilde{x}} = n'_{\tilde{x}}$ **then**
        compute new costs g'
        if $g' < g$ value of existing node in O **then**
          replace existing node in O with n'
        end if
      else
        $O \leftarrow O \cup \{n'\}$
      end if
    end if
  end for
end procedure



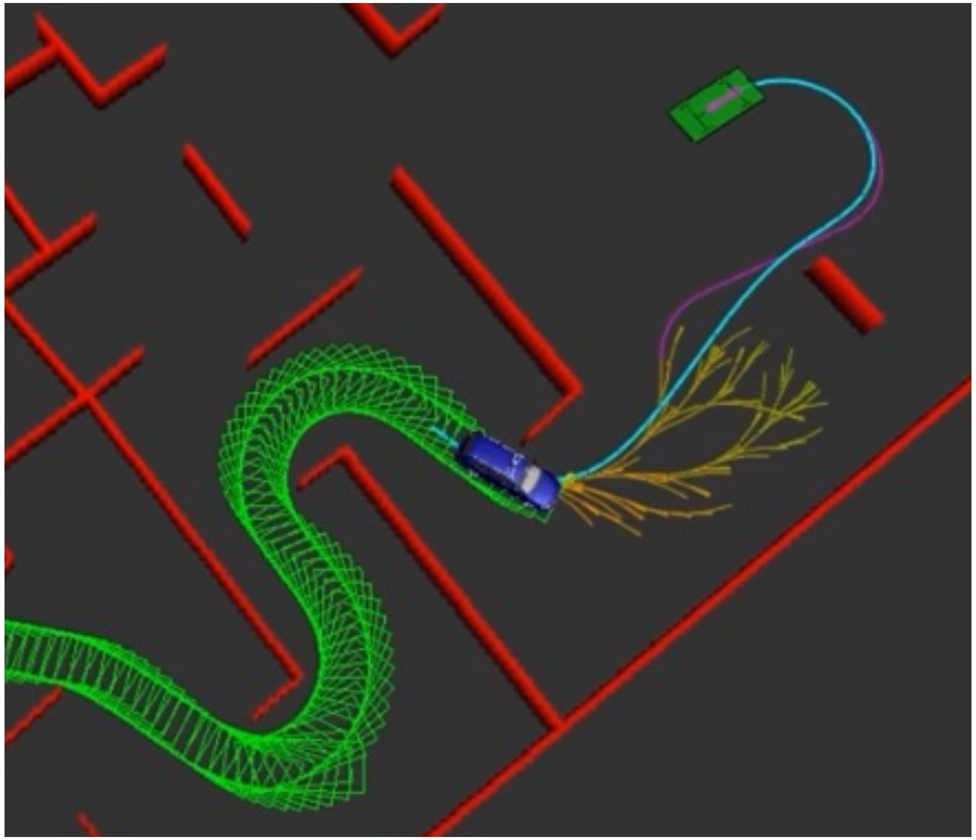
Key Difference

Need way to move between given headings

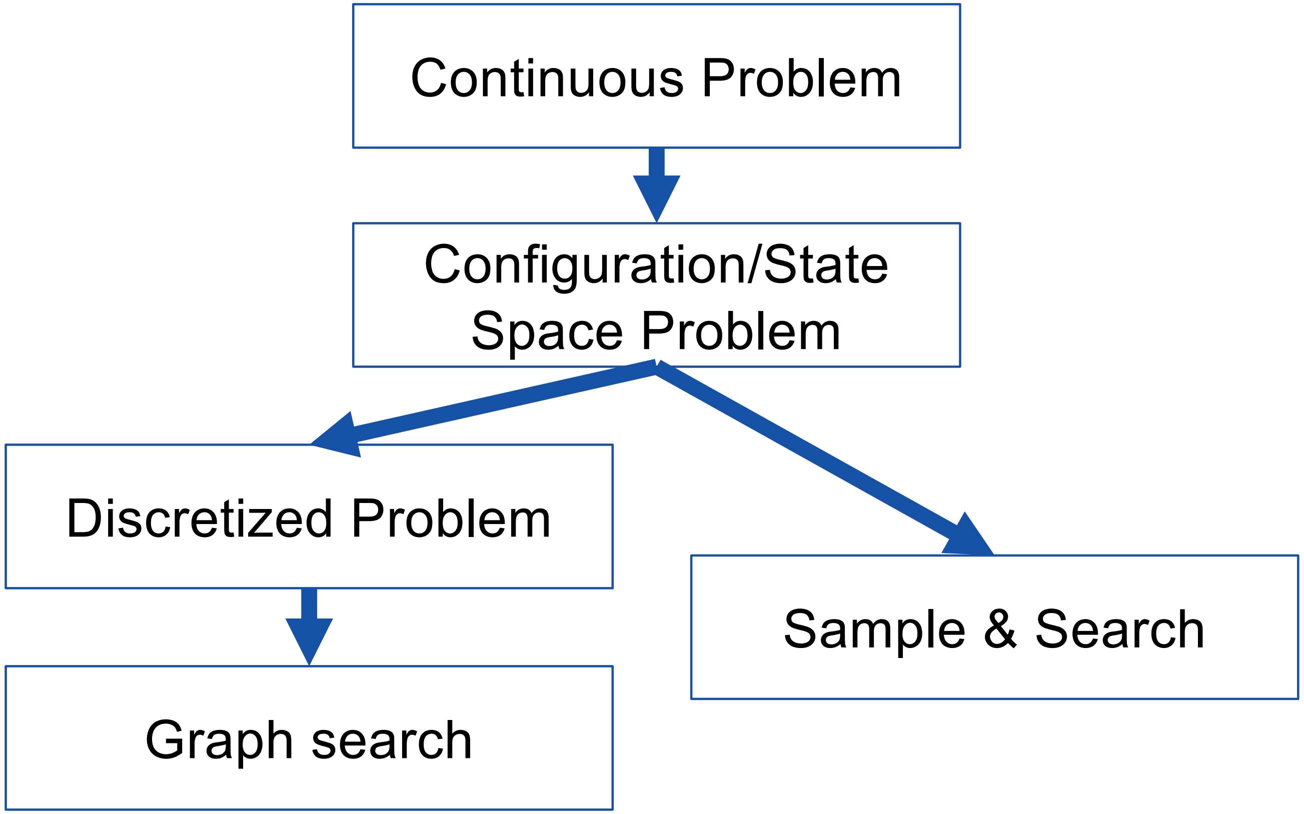
Note: Heading is discretized, only position is allowed to be “free” in cell

Planning for Autonomous Driving

- Orange: Hybrid A*
- Purple: Obstacle free solution (Dubins Car) from orange to goal
- Blue: Smooothed final trajectory



Common Path Planning Approach



The End