Function Approximation

Pieter Abbeel

UC Berkeley EECS

Value Iteration

- Algorithm:
  - Start with $V_0^*(s) = 0$ for all s.
  - For i=1, … , H

    For all states s $\in$ S:

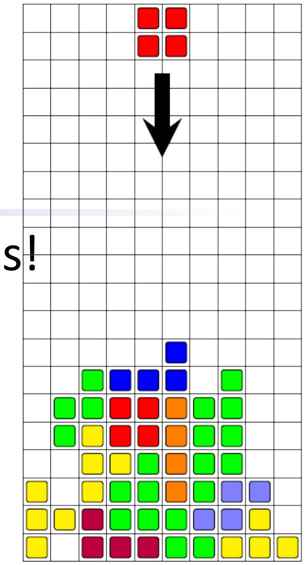
$$V_{i+1}^*(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i^*(s')\right]$$

$$\pi_{i+1}^*(s) \leftarrow \arg\max_{a \in A} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_i^*(s')]$$

Impractical for large state spaces

- $V_i^*(s)$ = the expected sum of rewards accumulated when starting from state s and acting optimally for a horizon of i steps
- $\pi_i^*(s)$ = the optimal action when in state s and getting to act for a horizon of i steps

Example: tetris

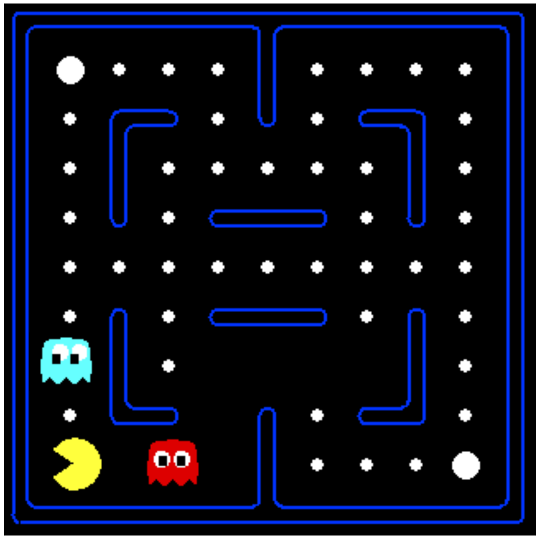


- state: board configuration + shape of the falling piece ~2^{200} states!
- action: rotation and translation applied to the falling piece
- 22 features aka basis functions ϕ_i
  - Ten basis functions, 0*, . . . , 9, mapping the state to the height h[k] of each of the ten* columns.
  - Nine basis functions, 10*, . . . , 18, each mapping the state to the absolute difference* between heights of successive columns: $|h[k+1] - h[k]|$, $k = 1, \ldots, 9$.
  - One basis function, 19, that maps state to the maximum column height: $\max_k h[k]$
  - One basis function, 20, that maps state to the number of 'holes' in the board.
  - One basis function, 21, that is equal to 1 in every state.

$$\hat{V}_\theta(s) = \sum_{i=0}^{21} \theta_i \phi_i(s) = \theta^\top \phi(s)$$

[Bertsekas & Ioffe, 1996 (TD); Bertsekas & Tsitsiklis 1996 (TD); Kakade 2002 (policy gradient); Farias & Van Roy, 2006 (approximate LP)]

Function Approximation



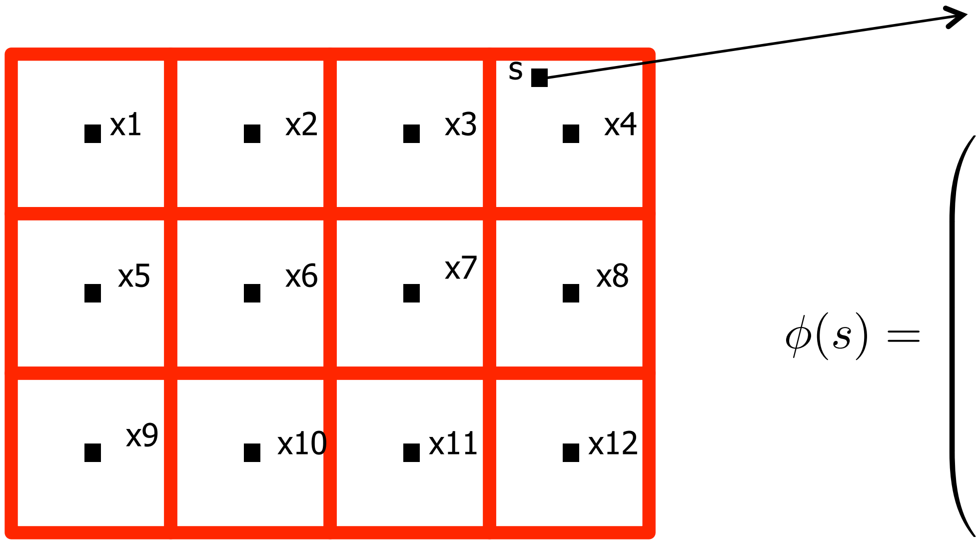
V(s) = θ_0 $+\theta_1$ "distance to closest ghost"
$+\theta_2$ "distance to closest power pellet"
$+\theta_3$ "in dead-end"
$+\theta_4$ "closer to power pellet than ghost is"
$+ \ldots$

$$= \sum_{i=0}^{n} \theta_i \phi_i(s) = \theta^\top \phi(s)$$

Function Approximation

- 0'th order approximation (1-nearest neighbor):

$$\hat{V}(s) = \hat{V}(x4) = \theta_4$$



$$\phi(s) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{pmatrix} \qquad \hat{V}(s) = \theta^\top \phi(s)$$

Only store values for x1, x2, …, x12

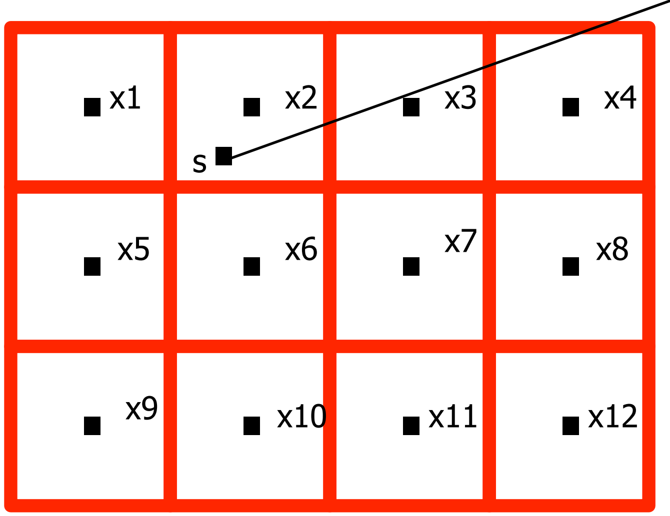
– call these values $\theta_1, \theta_2, \ldots, \theta_{12}$

Assign other states value of nearest "x" state

Function Approximation

- 1'th order approximation (k-nearest neighbor interpolation):

$$\hat{V}(s) = \phi_1(s)\theta_1 + \phi_2(s)\theta_2 + \phi_5(s)\theta_5 + \phi_6(s)\theta_6$$



$$\phi(s) = \begin{pmatrix} 0.2 \\ 0.6 \\ 0 \\ 0 \\ 0.05 \\ 0.15 \\ 0 \\ \ldots \\ 0 \end{pmatrix} \qquad \hat{V}(s) = \theta^\top \phi(s)$$

Only store values for x1, x2, …, x12

– call these values $\theta_1, \theta_2, \ldots, \theta_{12}$

Assign other states interpolated value of nearest 4 "x" states

Function Approximation

- Examples:
  - $S = \mathbb{R}, \quad \hat{V}(s) = \theta_1 + \theta_2 s$
  - $S = \mathbb{R}, \quad \hat{V}(s) = \theta_1 + \theta_2 s + \theta_3 s^2$
  - $S = \mathbb{R}, \quad \hat{V}(s) = \sum_{i=0}^{n} \theta_i s^i$
  - $S, \quad \hat{V}(s) = \log(\frac{1}{1 + \exp(\theta^\top \phi(s))})$

Function Approximation

- Main idea:
  - Use approximation $\hat{V}_\theta$ of the true value function V,
    - θ is a free parameter to be chosen from its domain Θ
    - Representation size: $|S|$ → downto: $|\Theta|$

+ : less parameters to estimate

- : less expressiveness, typically there exist many V for which there is no θ such that $\hat{V}_\theta = V$

Supervised Learning

- Given:
  - set of examples

$$(s^{(1)}, V(s^{(1)}), (s^{(2)}, V(s^{(2)}), \ldots, (s^{(m)}, V(s^{(m)})$$

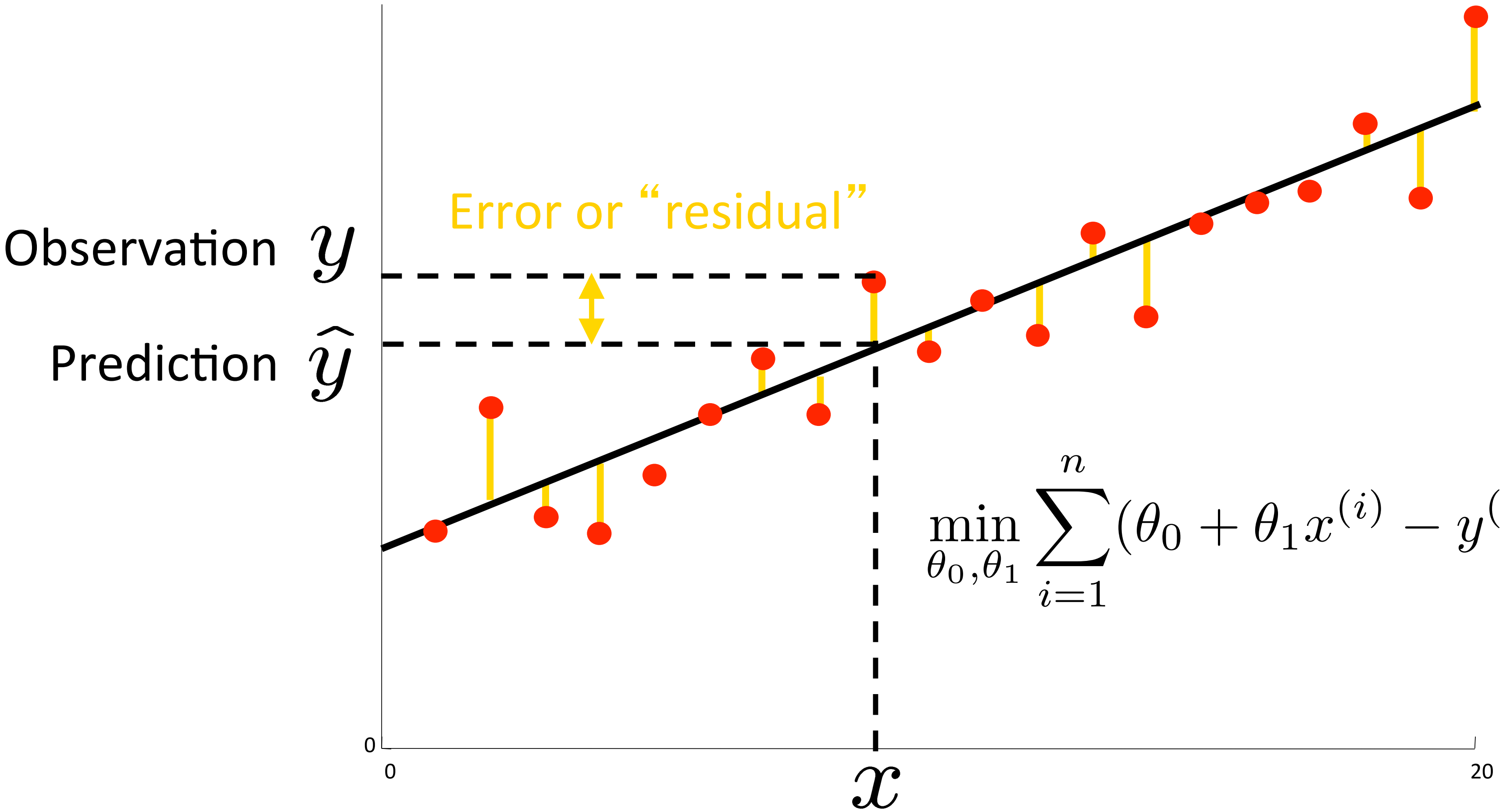
- Asked for:
  - "best" $\hat{V}_\theta$

- Representative approach: find θ through least squares:

$$\min_{\theta \in \Theta} \sum_{i=1}^{m} (\hat{V}_\theta(s^{(i)}) - V(s^{(i)}))^2$$

Supervised Learning Example

- Linear regression



$$\min_{\theta_0,\theta_1} \sum_{i=1}^{n} (\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2$$

Overfitting

- To avoid overfitting: reduce number of features used

- Practical approach: leave-out validation
  - Perform fitting for different choices of feature sets using just 70% of the data
  - Pick feature set that led to highest quality of fit on the remaining 30% of data

Value Iteration with Function Approximation

- Pick some $S' \subseteq S$ (typically $|S'| << |S|$)
- Initialize by choosing some setting for $\theta^{(0)}$
- Iterate for i = 0, 1, 2, …, H:
  - Step 1: Bellman back-ups

$$\forall s \in S': \quad \bar{V}_{i+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \hat{V}_{\theta^{(i)}}(s') \right]$$

  - Step 2: Supervised learning

    find $\theta^{(i+1)}$ as the solution of:

$$\min_\theta \sum_{s \in S'} \left(\hat{V}_{\theta^{(i+1)}}(s) - \bar{V}_{i+1}(s) \right)^2$$

Infinite Horizon Linear Program

$$\min_V \sum_{s \in S} \mu_0(s) V(s)$$

$$\text{s.t.} \quad V(s) \geq \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right], \quad \forall s \in S, a \in A$$

μ_0 is a probability distribution over S, with μ_0(s)> 0 for all s $\in$ S.

Theorem. V^* is the solution to the above LP.

Infinite Horizon Linear Program

$$\min_V \sum_{s \in S} \mu_0(s) V(s)$$

$$\text{s.t.} \quad V(s) \geq \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V(s') \right], \quad \forall s \in S, a \in A$$

- Let: $V(s) = \theta^\top \phi(s)$, and consider S' rather than S:

$$\min_\theta \sum_{s \in S'} \mu_0(s) \theta^\top \phi(s)$$

$$\text{s.t.} \quad \theta^\top \phi(s) \geq \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \theta^\top \phi(s') \right], \quad \forall s \in S', a \in A$$

→ Linear program that finds $\hat{V}_\theta(s) = \theta^\top \phi(s)$

Approximate Linear Program – Guarantees**

$$\min_{\theta} \sum_{s \in S'} \mu_0(s) \theta^\top \phi(s)$$

$$\text{s.t.} \quad \theta^\top \phi(s) \geq \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \theta^\top \phi(s') \right], \quad \forall s \in S', a \in A$$

- LP solver will converge
- Solution quality: [de Farias and Van Roy, 2002]

Assuming one of the features is the feature that is equal to one for all states, and assuming S'=S we have that:

$$\| V^* - \Phi\theta \|_{1,\mu_0} \leq \frac{2}{1-\gamma} \min_{\theta} \| V^* - \Phi\theta \|_{\infty}$$

(slightly weaker, probabilistic guarantees hold for S' not equal to S, these guarantees require size of S' to grow as the number of features grows)

Sampling-Based Motion Planning

Pieter Abbeel

UC Berkeley EECS

Many images from Lavalle, Planning Algorithms

Motion Planning

- **Problem**
  - Given start state x_S, goal state x_G
  - Asked for: a sequence of control inputs that leads from start to goal
- Why tricky?
  - Need to avoid obstacles
  - For systems with underactuated dynamics: can't simply move along any coordinate at will
    - E.g., car, helicopter, airplane, but also robot manipulator hitting joint limits

Solve by Nonlinear Optimization for Control?

- Could try by, for example, following formulation:

$$\begin{aligned} \min_{u,x} \quad & (x_T - x_G)^\top (x_T - x_G) \\ \text{s.t.} \quad & x_{t+1} = f(x_t, u_t) \quad \forall t \\ & u_t \in \mathcal{U}_t \\ & x_t \in \mathcal{X}_t \\ & x_0 = x_S \end{aligned}$$

X_t can encode obstacles

- Or, with constraints, (which would require using an infeasible method):

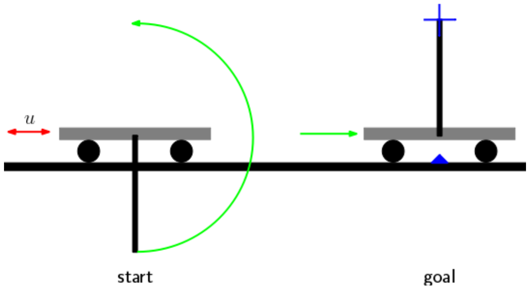
$$\begin{aligned} \min_{u,x} \quad & \|u\| \\ \text{s.t.} \quad & x_{t+1} = f(x_t, u_t) \quad \forall t \\ & u_t \in \mathcal{U}_t \\ & x_t \in \mathcal{X}_t \\ & x_0 = x_S \\ & X_T = x_G \end{aligned}$$

- *Can work surprisingly well, but for more complicated problems with longer horizons, often get stuck in local maxima that don't reach the goal*

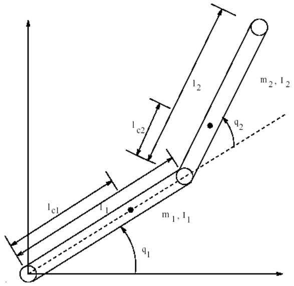
Examples

- Helicopter path planning

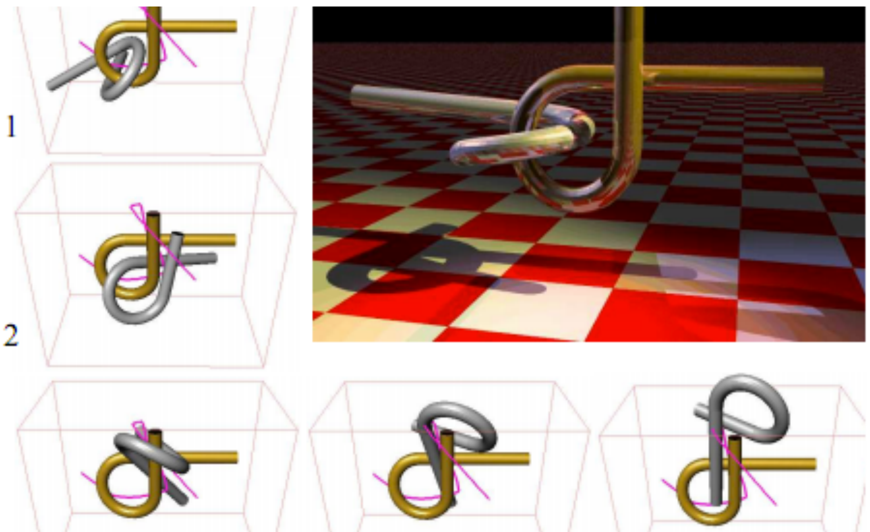
- Swinging up cart-pole

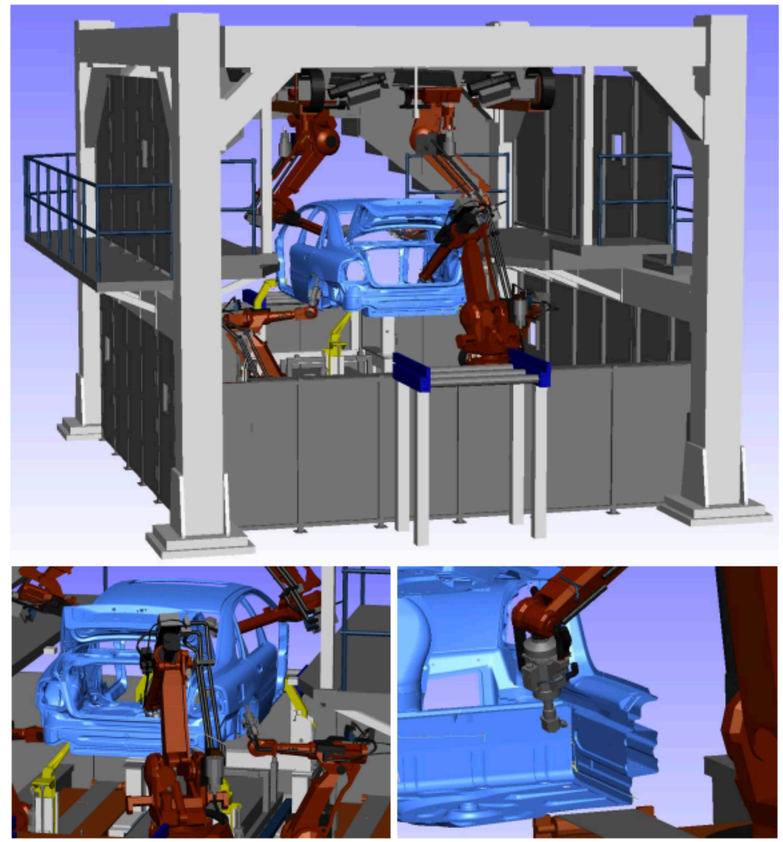


- Acrobot

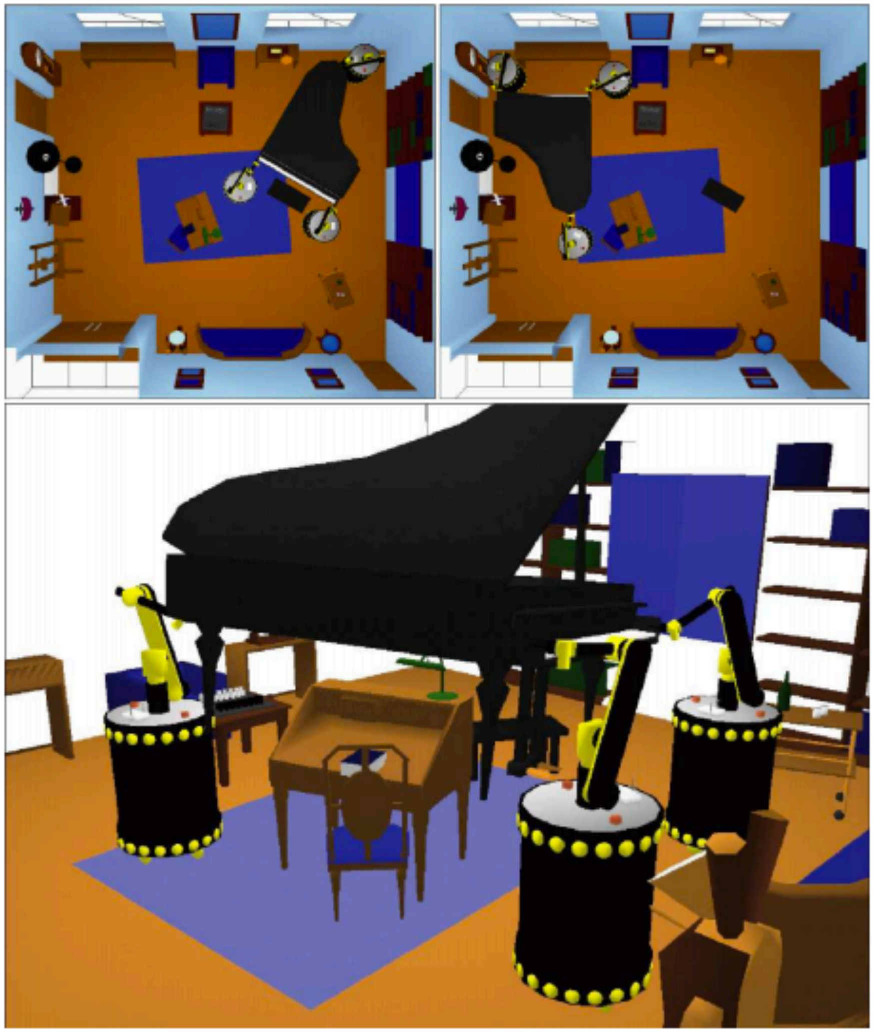


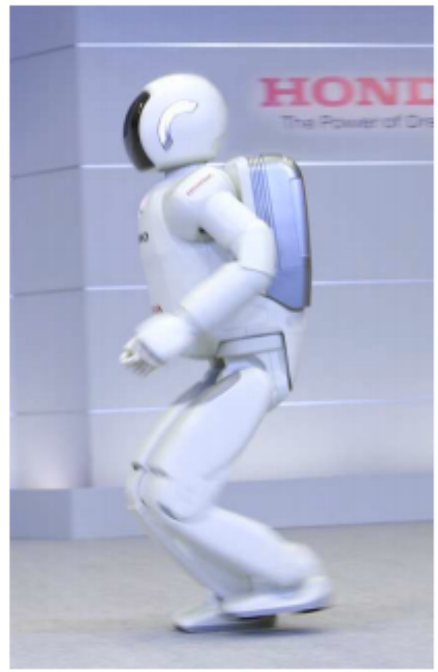
Examples

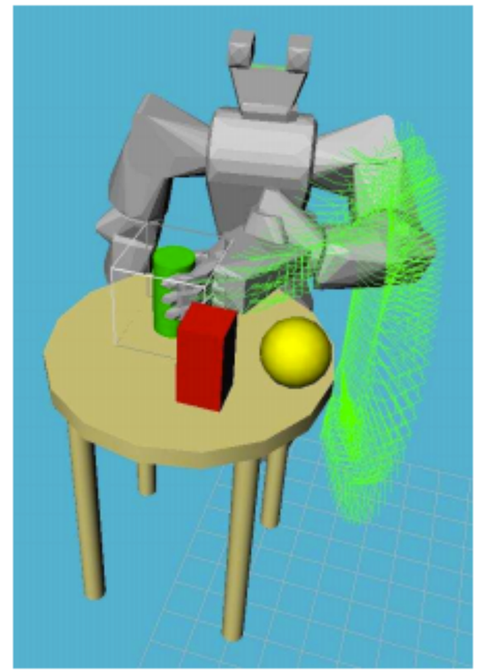




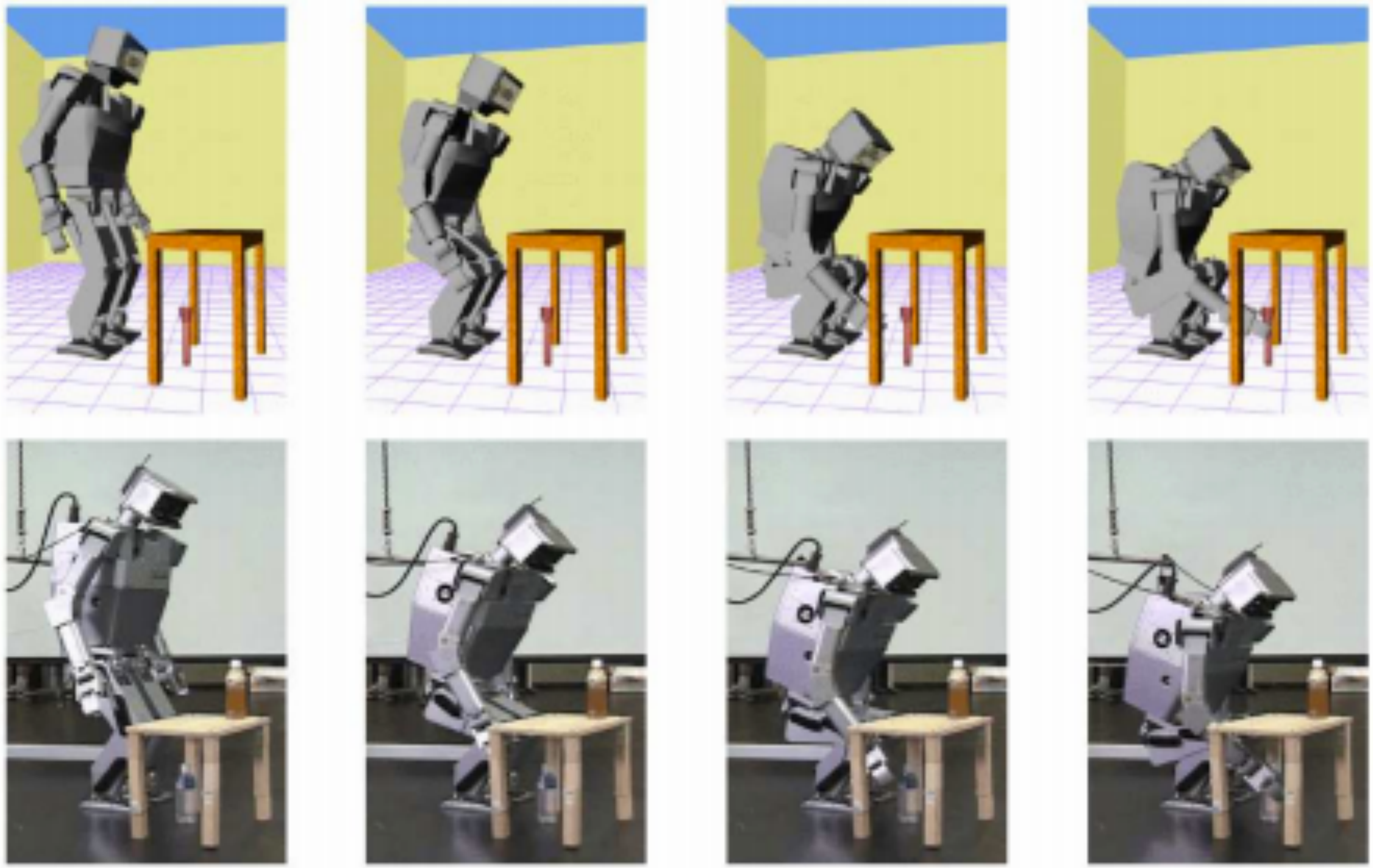
Examples







Examples



Motion Planning: Outline

- Configuration Space
- Probabilistic Roadmap
  - Boundary Value Problem
  - Sampling
  - Collision checking
- Rapidly-exploring Random Trees (RRTs)
- Smoothing

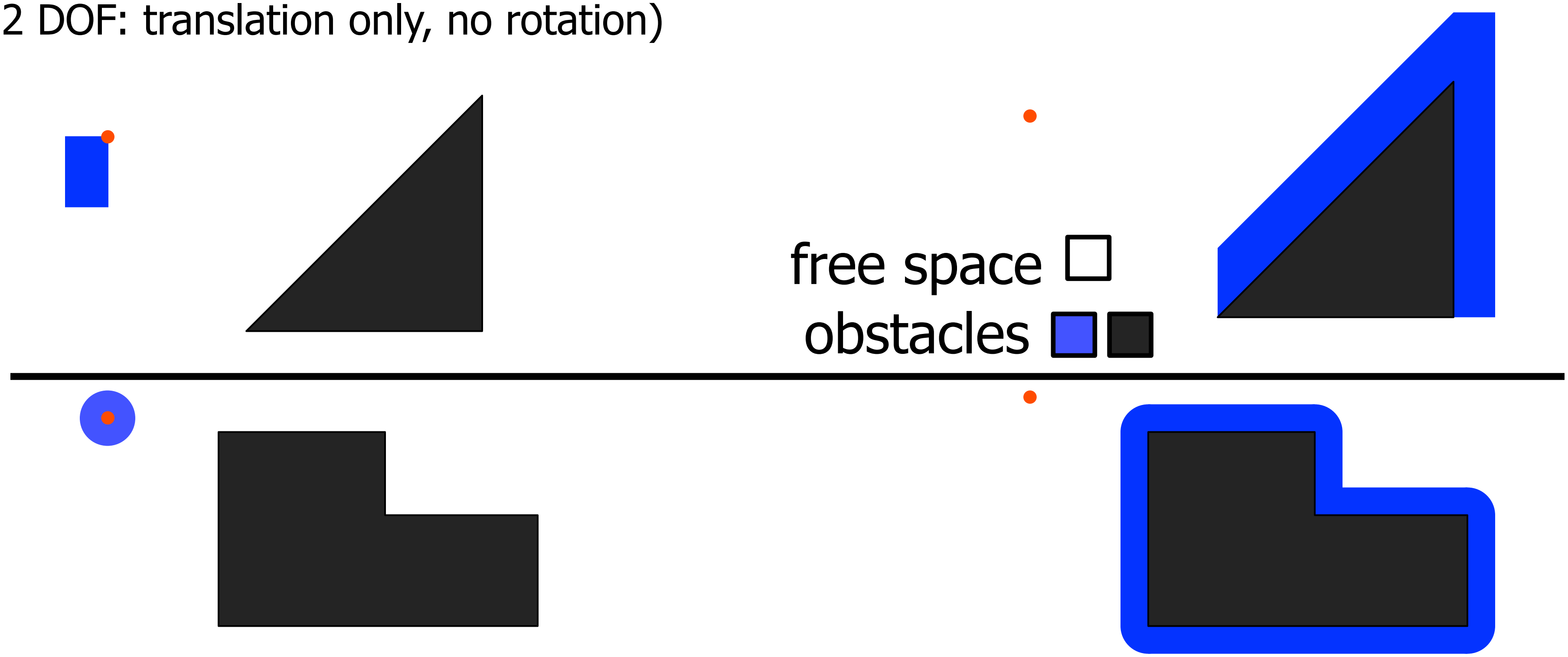
Configuration Space (C-Space)

= { x | x is a pose of the robot}

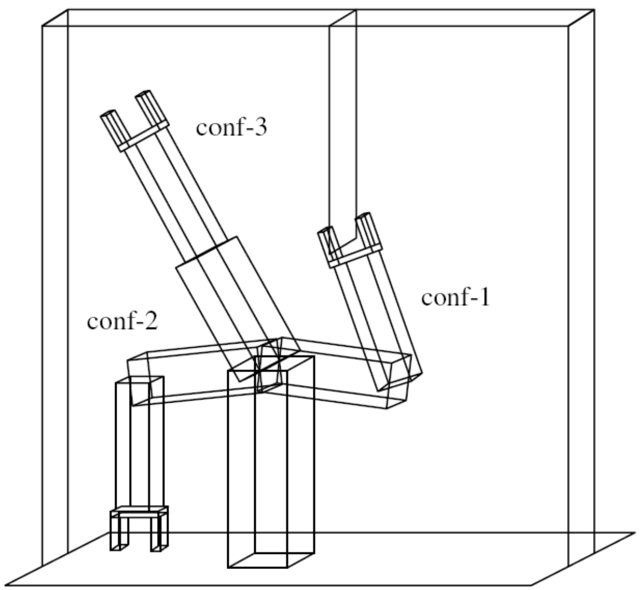
- obstacles → configuration space obstacles

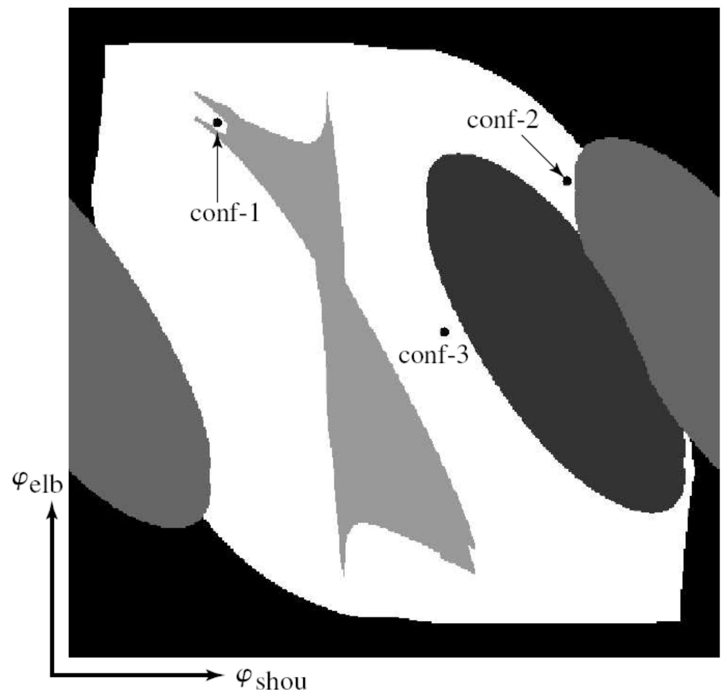
Workspace *Configuration Space*

(2 DOF: translation only, no rotation)

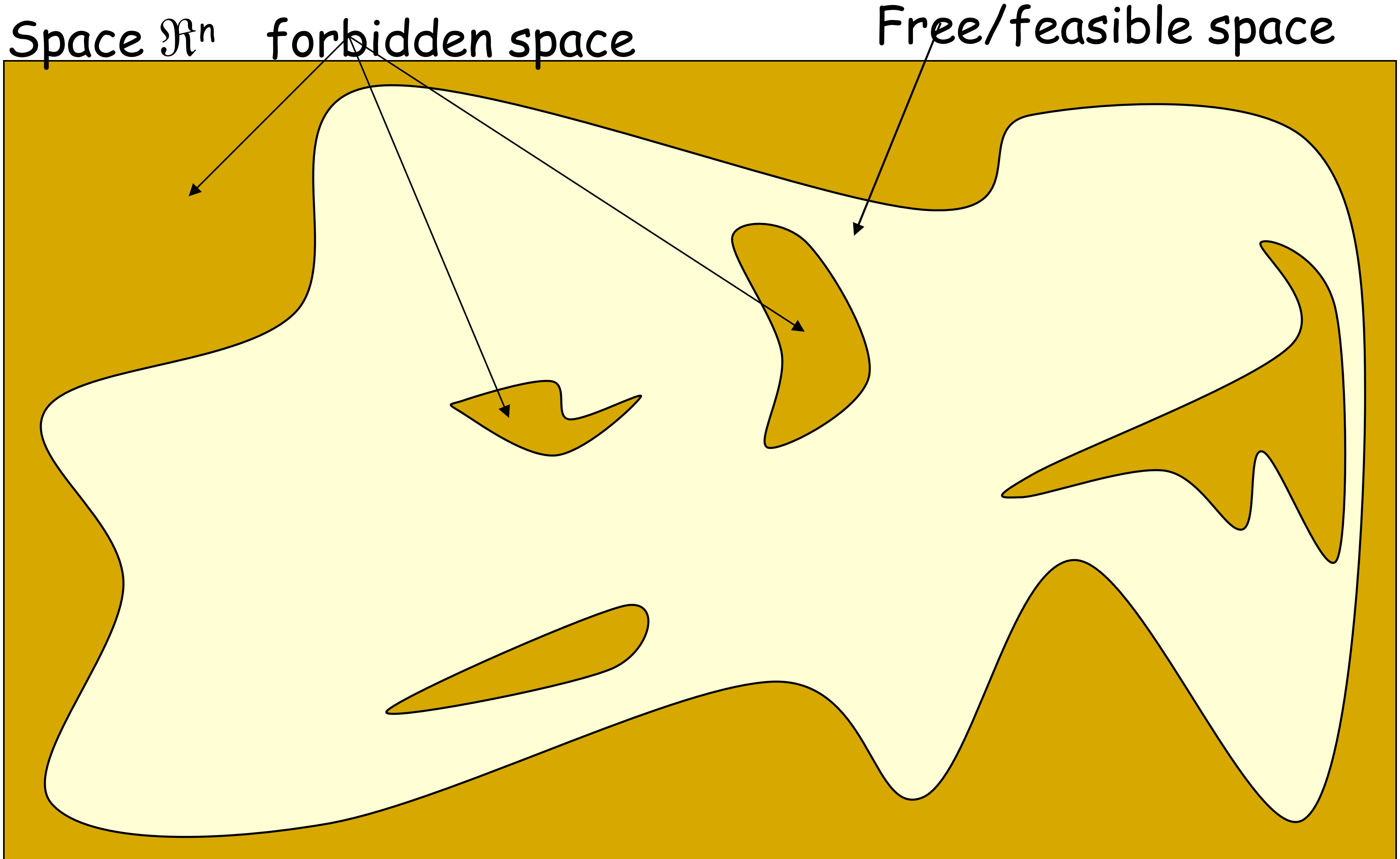


Motion planning



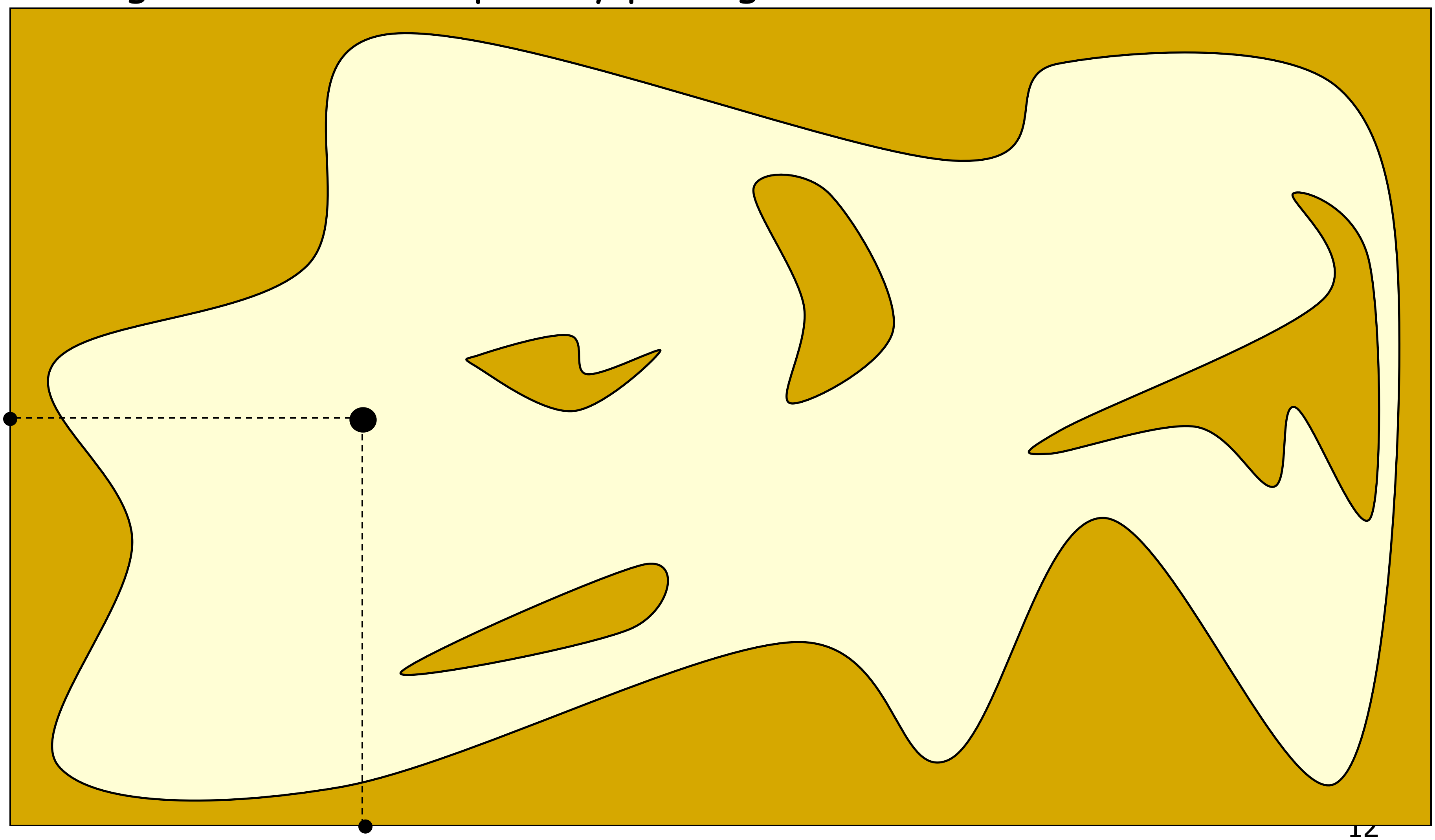


Probabilistic Roadmap (PRM)



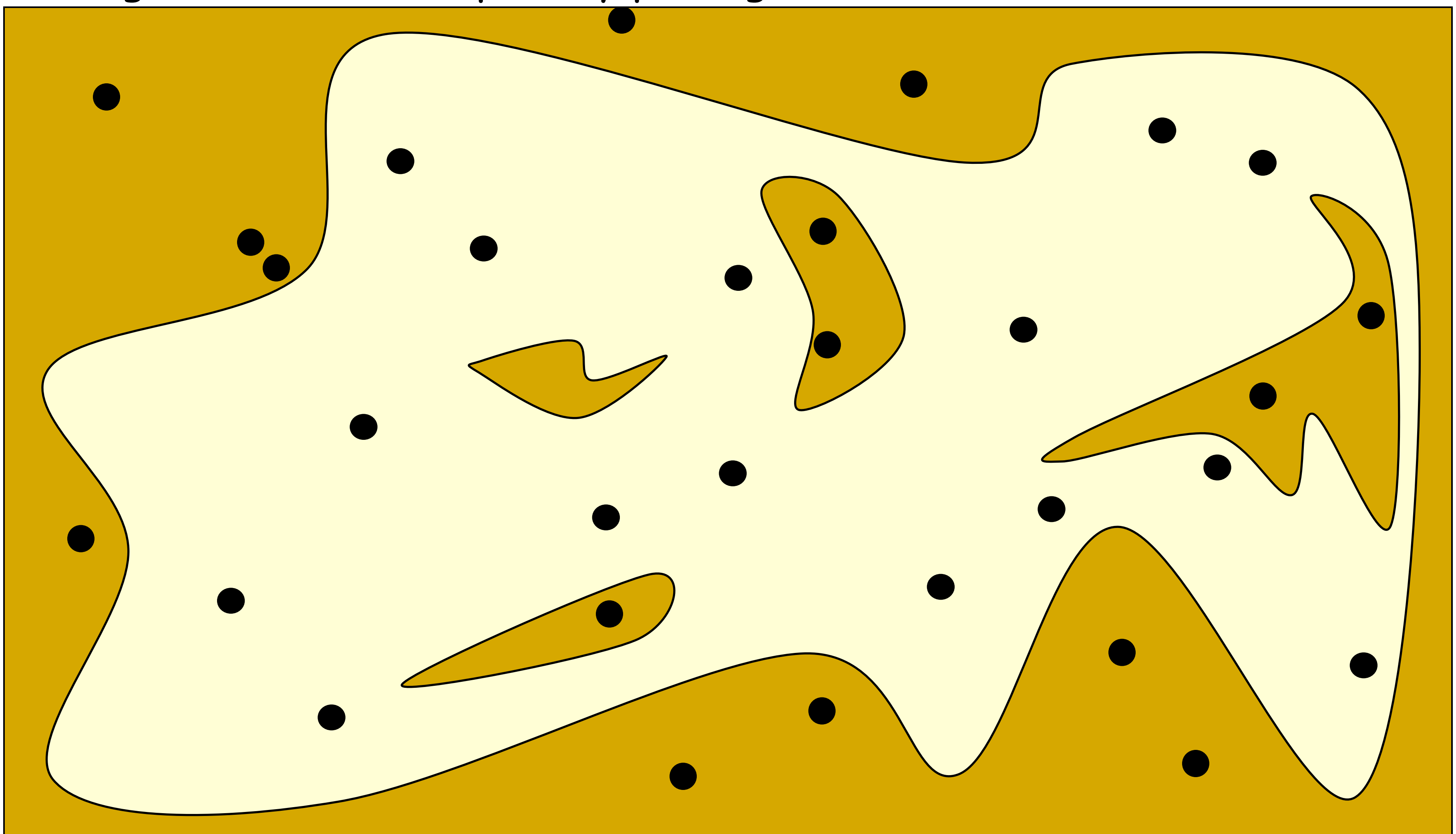
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



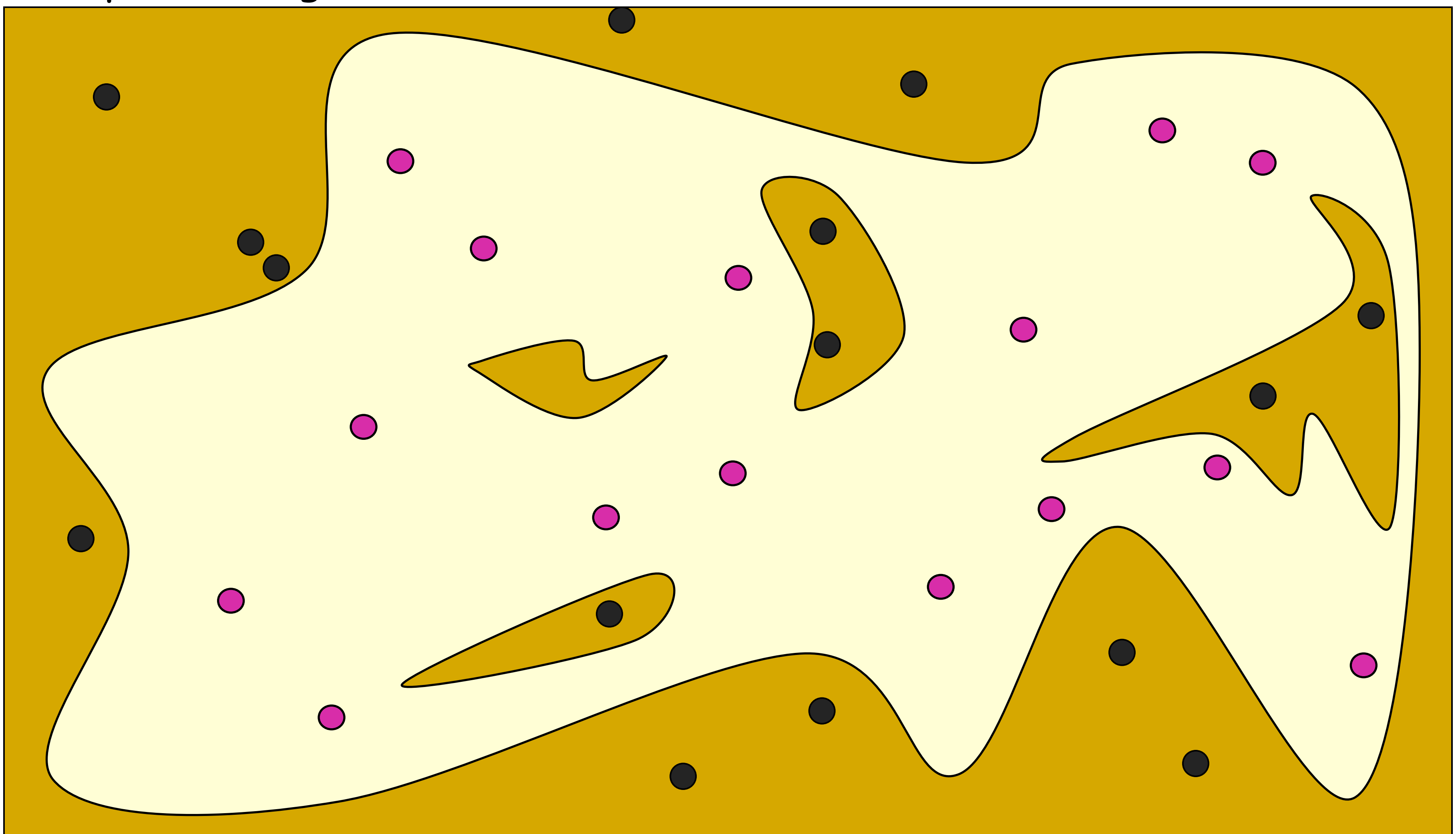
Probabilistic Roadmap (PRM)

Configurations are sampled by picking coordinates at random



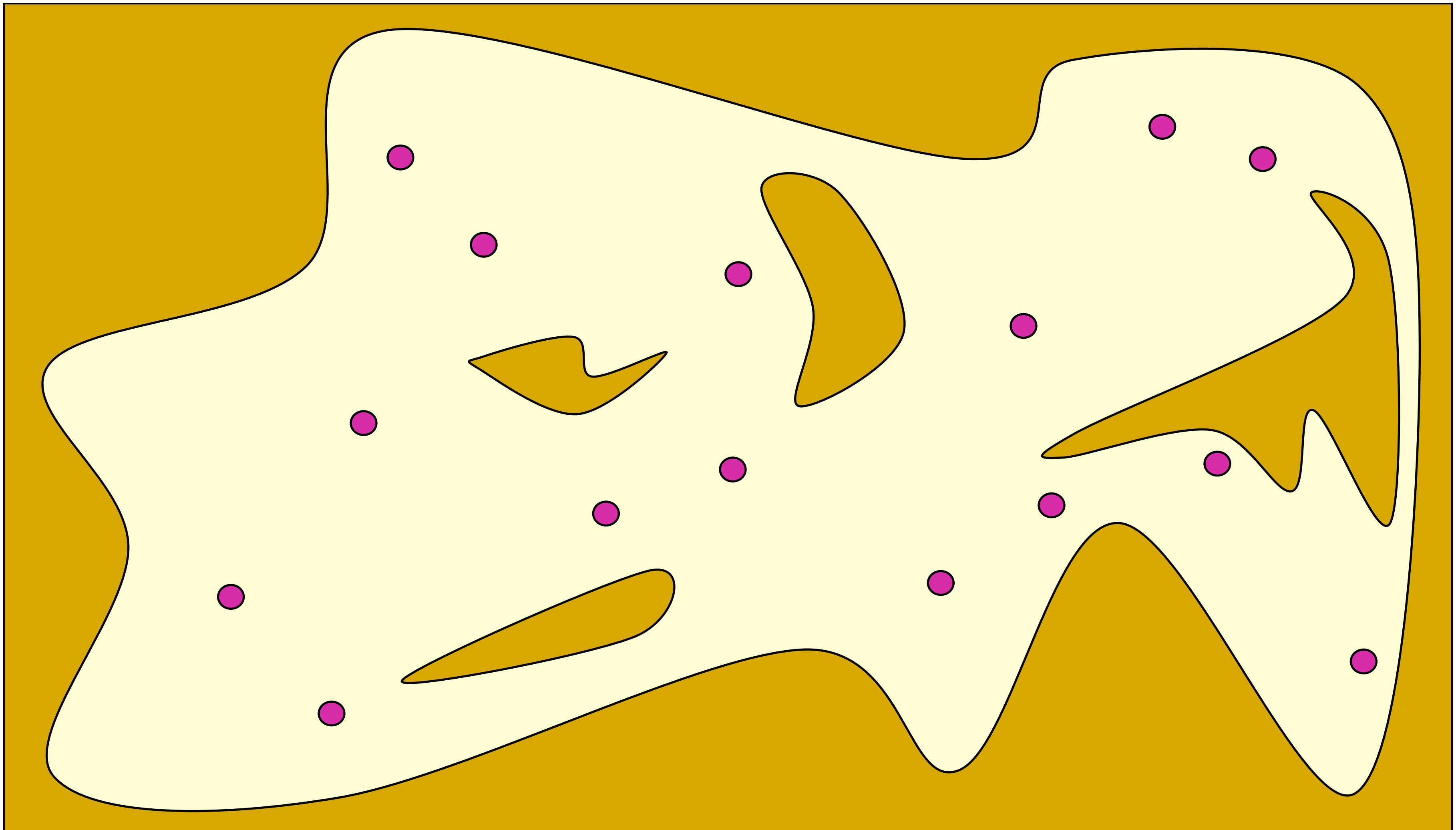
Probabilistic Roadmap (PRM)

Sampled configurations are tested for collision



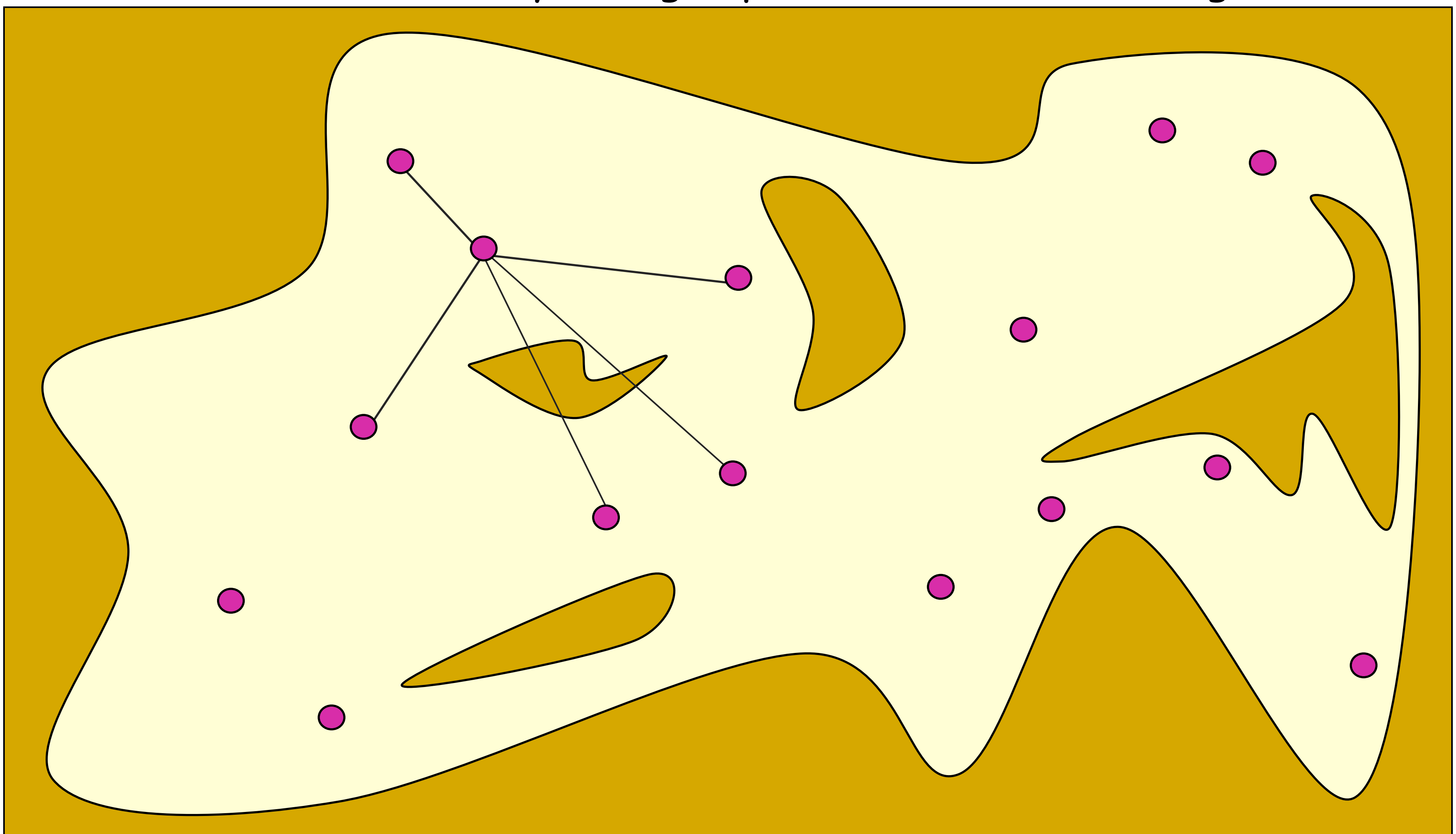
Probabilistic Roadmap (PRM)

The collision-free configurations are retained as milestones



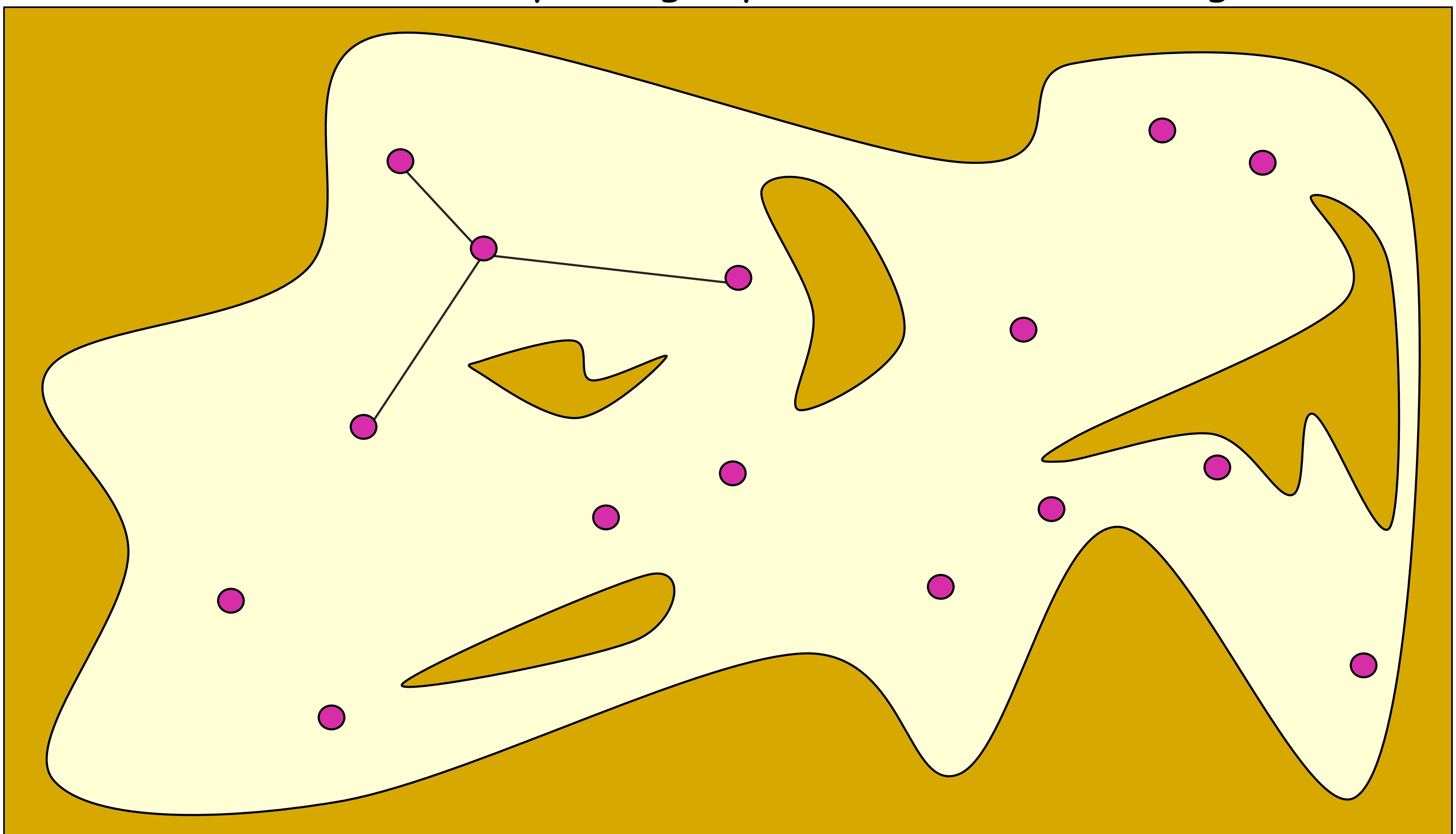
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



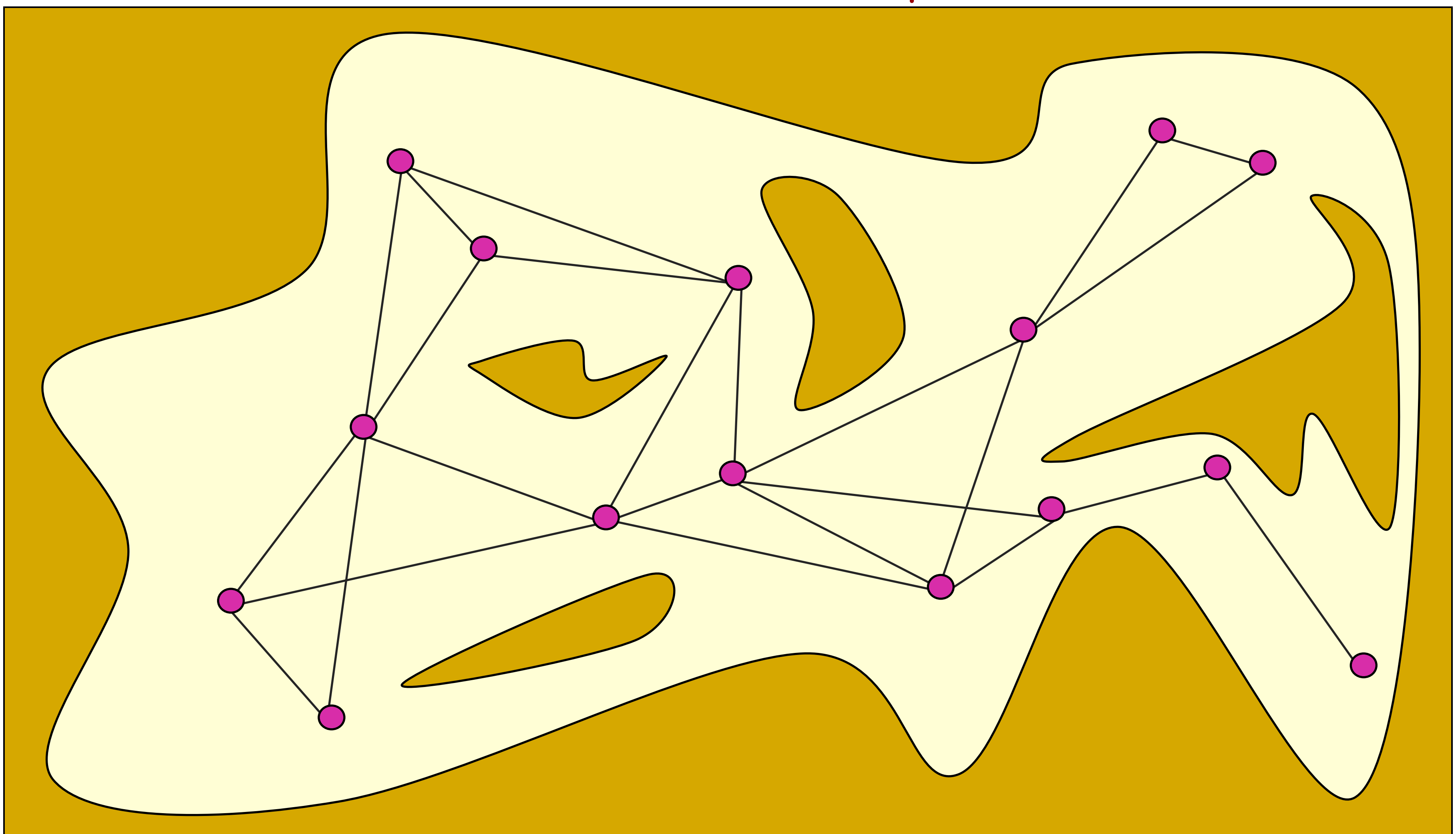
Probabilistic Roadmap (PRM)

Each milestone is linked by straight paths to its nearest neighbors



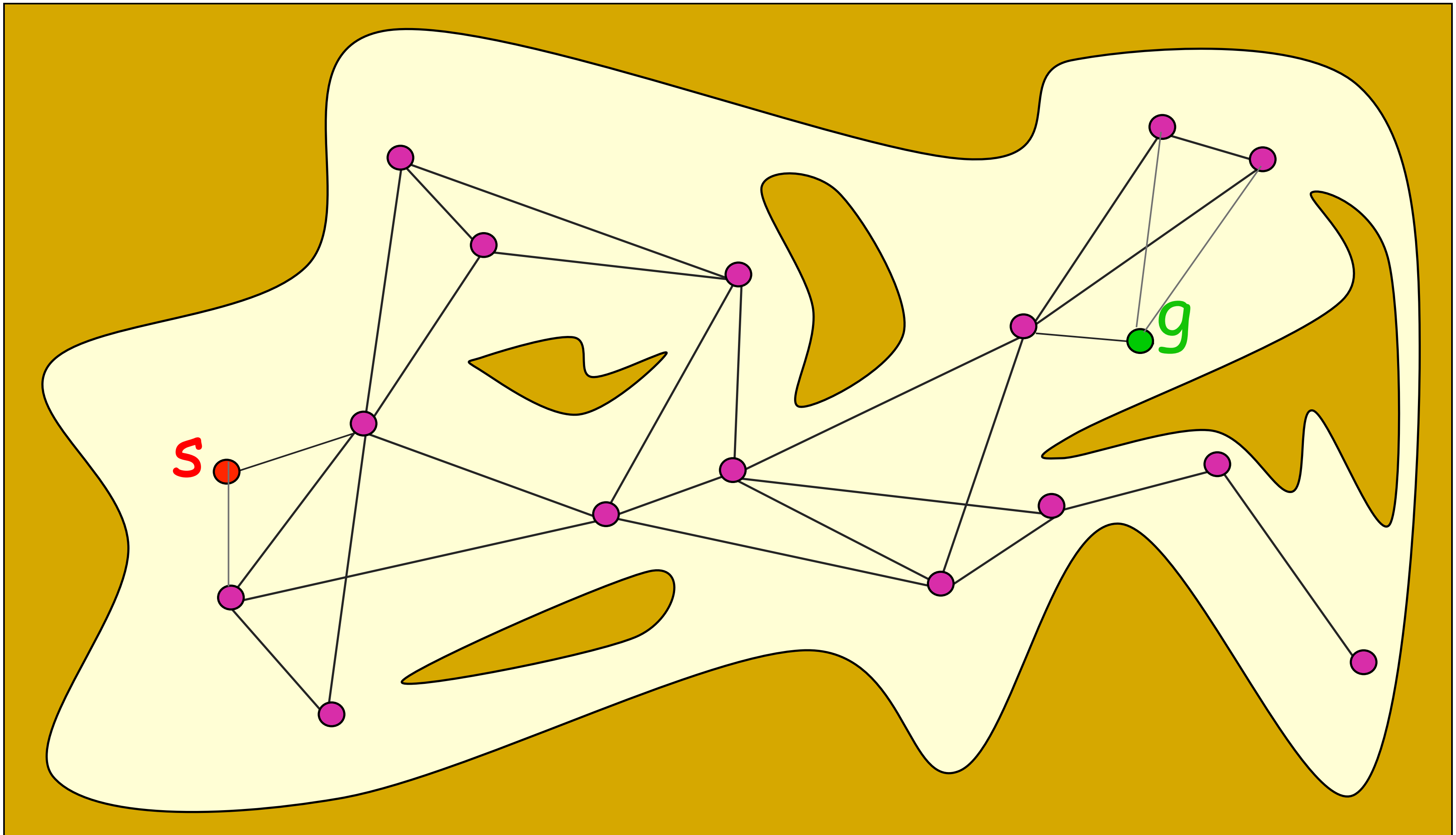
Probabilistic Roadmap (PRM)

The collision-free links are retained as local paths to form the PRM



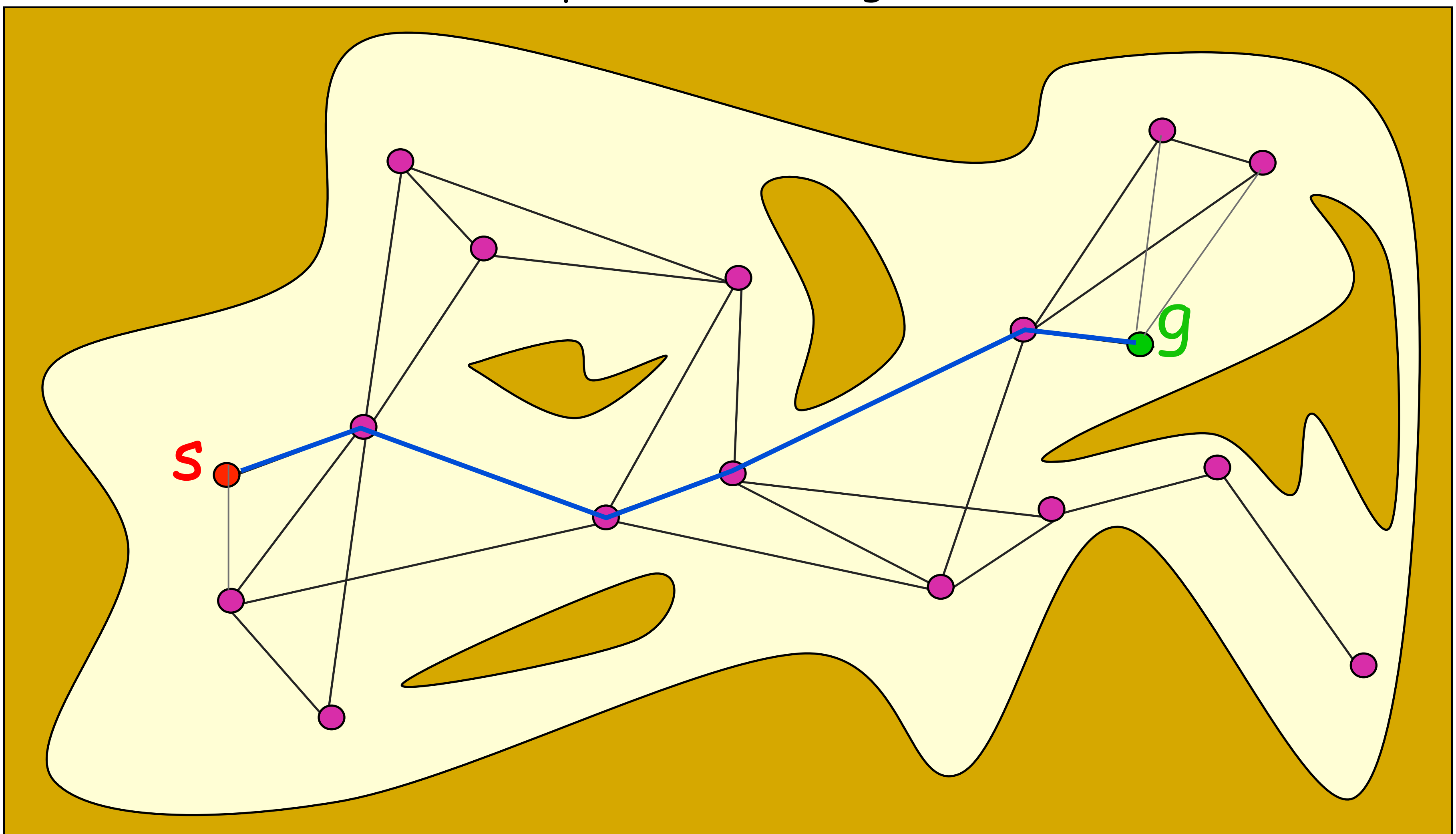
Probabilistic Roadmap (PRM)

The start and goal configurations are included as milestones



Probabilistic Roadmap (PRM)

The PRM is searched for a path from s to g

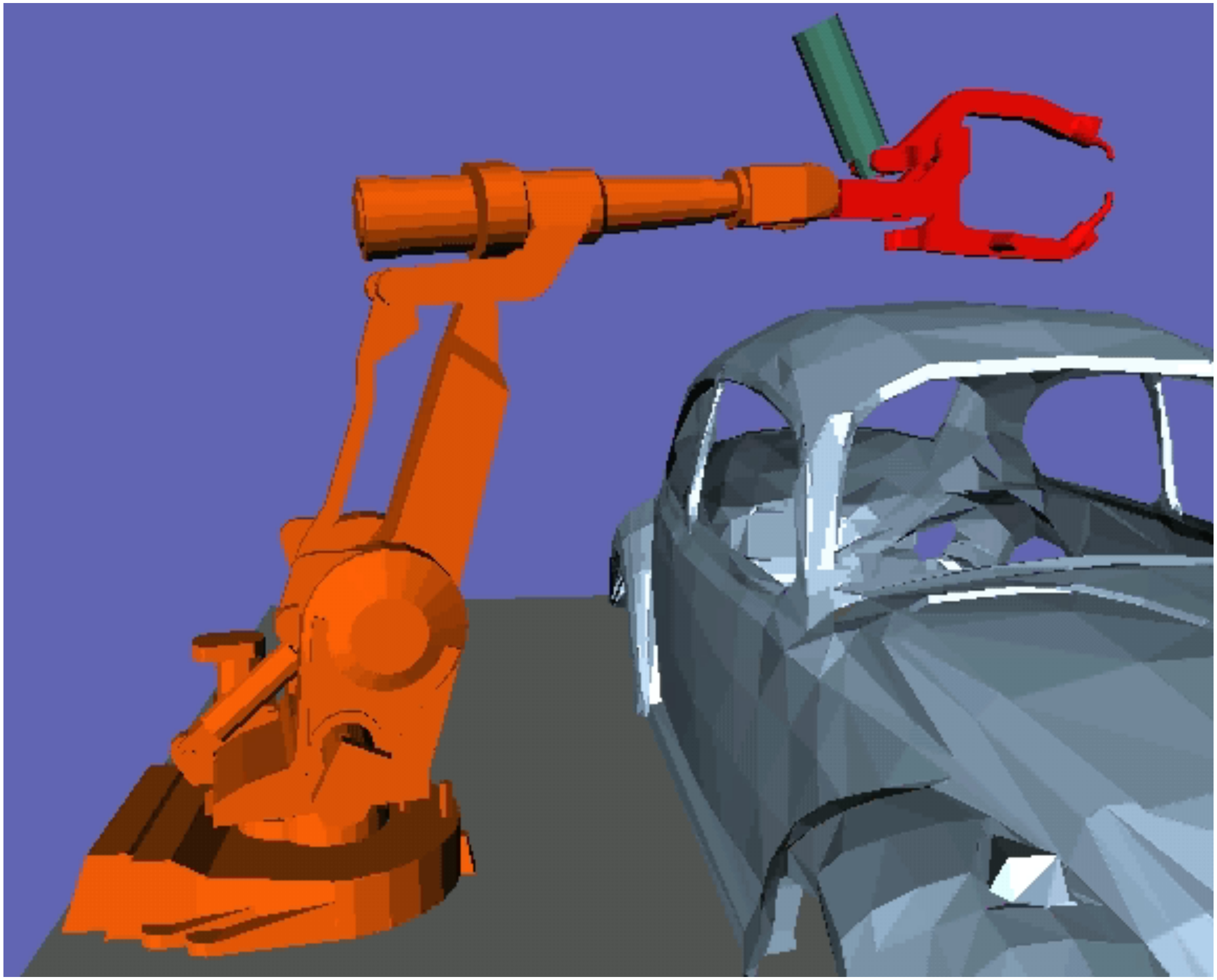


Probabilistic Roadmap

- Initialize set of points with x_S and x_G
- Randomly sample points in configuration space
- Connect nearby points if they can be reached from each other
- Find path from x_S to x_G in the graph

  - Alternatively: keep track of connected components incrementally, and declare success when x_S and x_G are in same connected component

PRM example



PRM example 2

Sampling

- How to sample uniformly at random from $[0,1]^n$?
  - Sample uniformly at random from [0,1] for each coordinate
- How to sample uniformly at random from the surface of the n-D unit sphere?
  - Sample from n-D Gaussian → isotropic; then just normalize
- How to sample uniformly at random for orientations in 3-D?

PRM: Challenges

1. Connecting neighboring points: Only easy for holonomic systems (i.e., for which you can move each degree of freedom at will at any time). Generally requires solving a Boundary Value Problem

$$
\begin{aligned}
\min_{u,x} \quad & \|u\| \\
\text{s.t.} \quad & x_{t+1} = f(x_t, u_t) \quad \forall t \\
& u_t \in \mathcal{U}_t \\
& x_t \in \mathcal{X}_t \\
& x_0 = x_S \\
& X_T = x_G
\end{aligned}
$$

Typically solved without collision checking; later verified if valid by collision checking

2. Collision checking:

Often takes majority of time in applications (see Lavalle)

PRM's Pros and Cons

- Pro:
  - Probabilistically complete: i.e., with probability one, if run for long enough the graph will contain a solution path if one exists.
- Cons:
  - Required to solve 2 point boundary value problem
  - Build graph over state space but no particular focus on generating a path

Rapidly exploring Random Trees

- Basic idea:
  - Build up a tree through generating “next states” in the tree by executing random controls
  - However: not exactly above to ensure good coverage

Rapidly-exploring Random Trees (RRT)

GENERATE_RRT($x_{init}, K, \Delta t$)

$\mathcal{T}$.init(x_{init});

for $k = 1$ **to** K **do**

  $x_{rand} \leftarrow$ RANDOM_STATE();

  $x_{near} \leftarrow$ NEAREST_NEIGHBOR($x_{rand}, \mathcal{T}$);

  $u \leftarrow$ SELECT_INPUT(x_{rand}, x_{near});

  $x_{new} \leftarrow$ NEW_STATE($x_{near}, u, \Delta t$);

  $\mathcal{T}$.add_vertex(x_{new});

  $\mathcal{T}$.add_edge(x_{near}, x_{new}, u);

Return $\mathcal{T}$

RANDOM_STATE(): often uniformly at random over space with probability 99%, and the goal state with probability 1%, this ensures it attempts to connect to goal semi-regularly

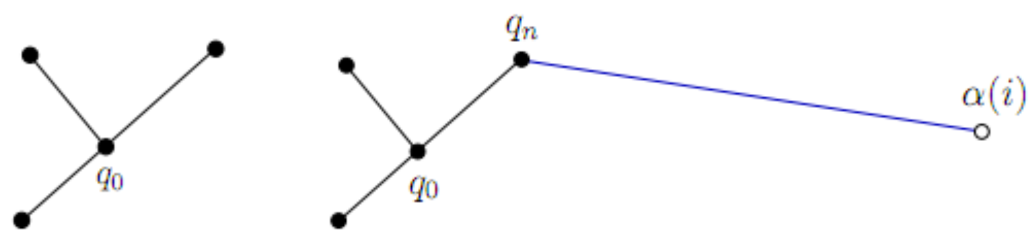
RRT Practicalities

- NEAREST_NEIGHBOR(x_{rand}, T): need to find (approximate) nearest neighbor efficiently
  - KD Trees data structure (upto 20-D) [e.g., FLANN]
  - Locality Sensitive Hashing

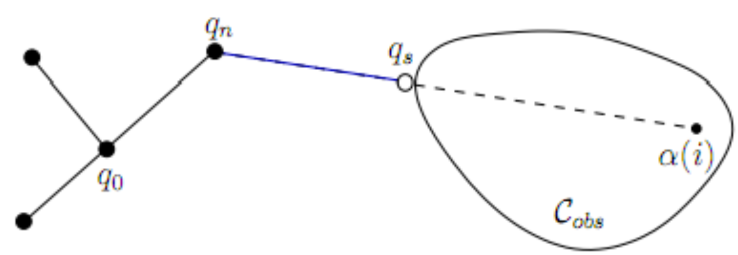
- SELECT_INPUT(x_{rand}, x_{near})
  - Two point boundary value problem
    - If too hard to solve, often just select best out of a set of control sequences. This set could be random, or some well chosen set of primitives.

RRT Extension

- No obstacles, holonomic:



- With obstacles, holonomic:



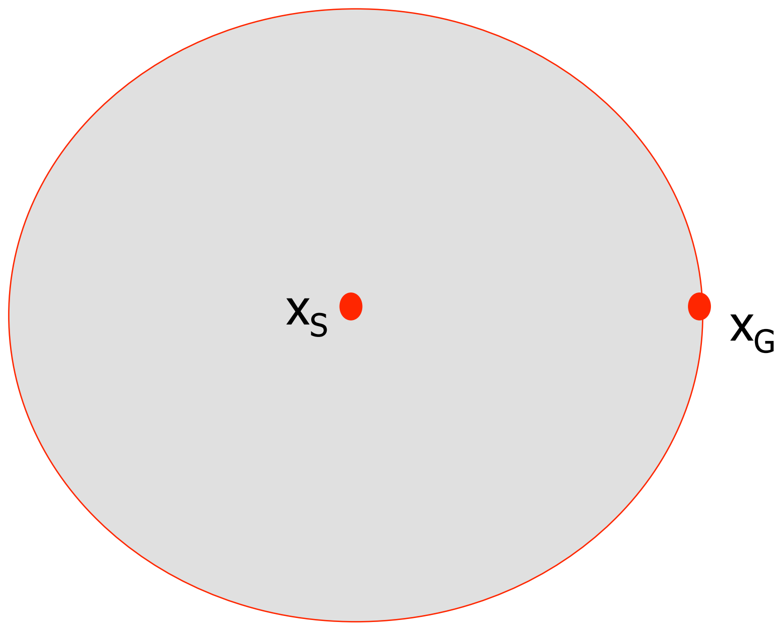
- Non-holonomic: approximately (sometimes as approximate as picking best of a few random control sequences) solve two-point boundary value problem

Growing RRT

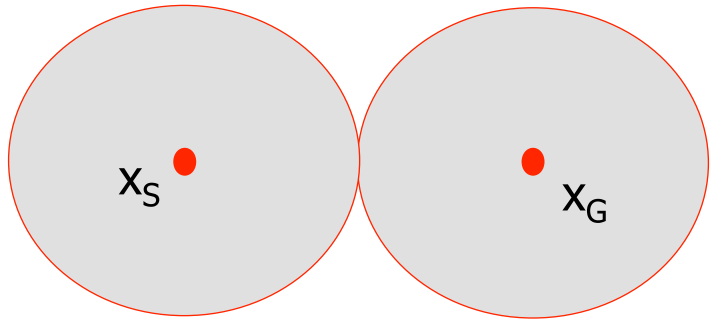
Demo: http://en.wikipedia.org/wiki/File:Rapidly-exploring_Random_Tree_(RRT)_500x373.gif

Bi-directional RRT

- Volume swept out by unidirectional RRT:



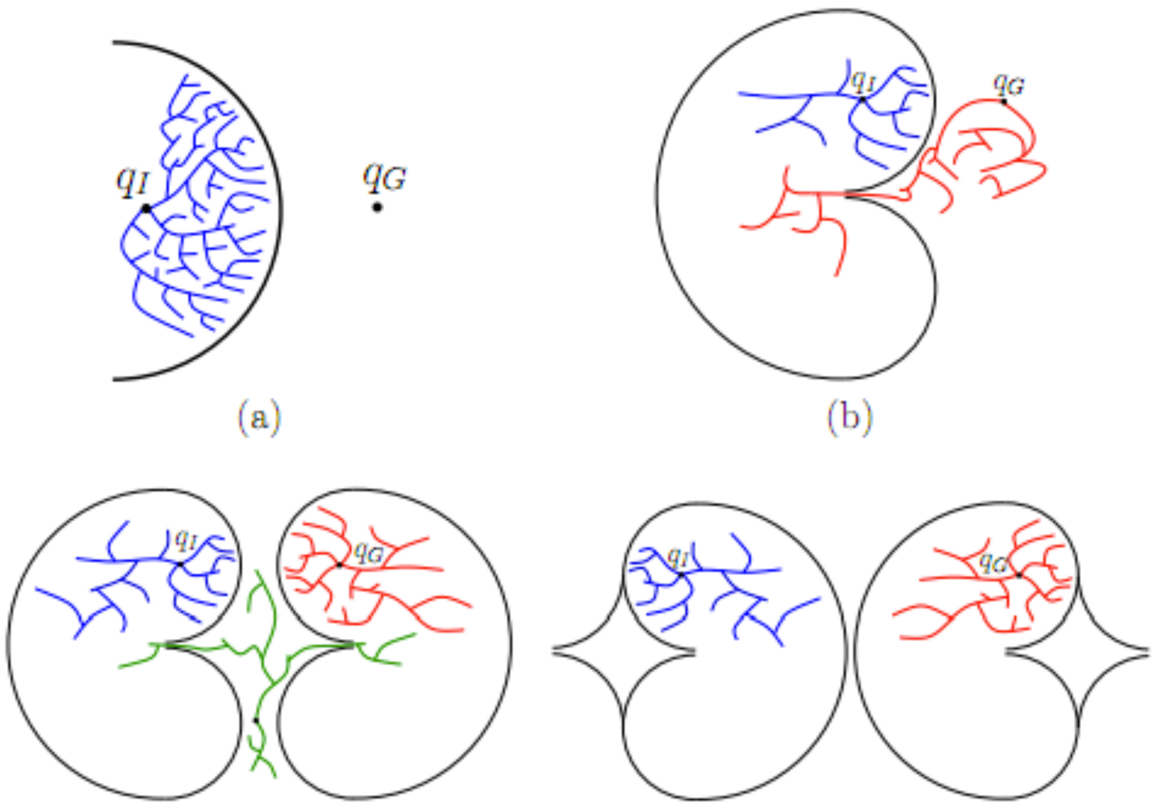
- Volume swept out by bi-directional RRT:



- Difference becomes far more pronounced in higher dimensions

Multi-directional RRT

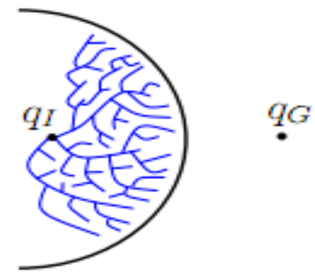
- Planning around obstacles or through narrow passages can often be easier in one direction than the other



(a) (b) (c) (d)

Resolution-Complete RRT (RC-RRT)

- Issue: nearest points chosen for expansion are (too) often the ones stuck behind an obstacle



RC-RRT solution:

- Choose a maximum number of times, m, you are willing to try to expand each node
- For each node in the tree, keep track of its Constraint Violation Frequency (CVF)
- Initialize CVF to zero when node is added to tree
- Whenever an expansion from the node is unsuccessful (e.g., per hitting an obstacle):
  - Increase CVF of that node by 1
  - Increase CVF of its parent node by 1/m, its grandparent $1/m^2$, …
- When a node is selected for expansion, skip over it with probability CVF/m

RRT*

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**
        $X_{\text{near}} \leftarrow \texttt{Near}(G = (V, E), x_{\text{new}}, \min\{\gamma_{\text{RRT}^*}(\log(\text{card}\,(V))/\,\text{card}\,(V))^{1/d}, \eta\})$;
        $V \leftarrow V \cup \{x_{\text{new}}\}$;
        $x_{\min} \leftarrow x_{\text{nearest}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{nearest}}) + c(\texttt{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
        foreach $x_{\text{near}} \in X_{\text{near}}$ **do**   // Connect along a minimum-cost path
            if $\texttt{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\min}$ **then**
                $x_{\min} \leftarrow x_{\text{near}}$; $c_{\min} \leftarrow \texttt{Cost}(x_{\text{near}}) + c(\texttt{Line}(x_{\text{near}}, x_{\text{new}}))$
        $E \leftarrow E \cup \{(x_{\min}, x_{\text{new}})\}$;
        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 $G = (V, E)$;

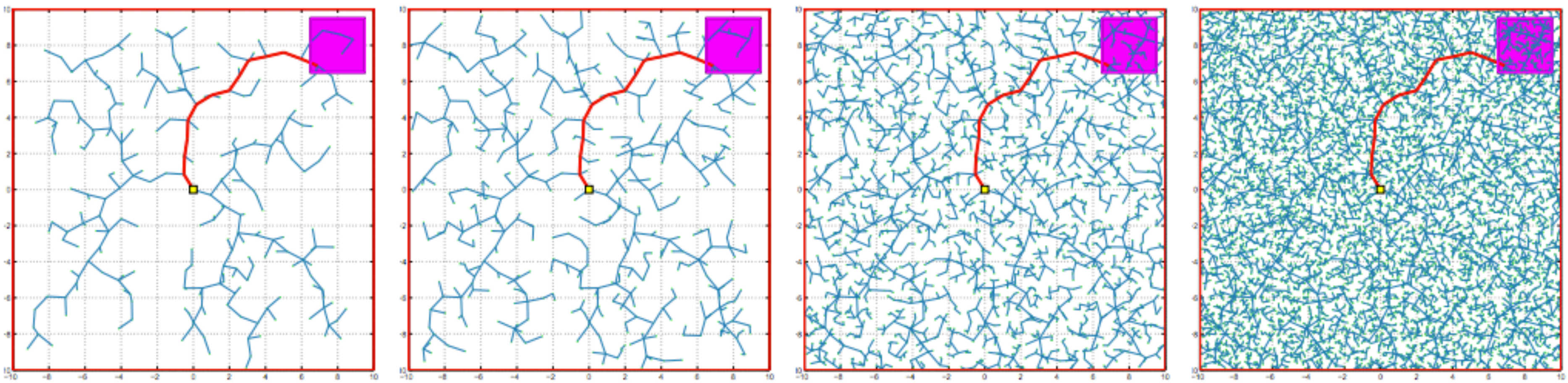
Source: Karaman and Frazzoli

RRT*

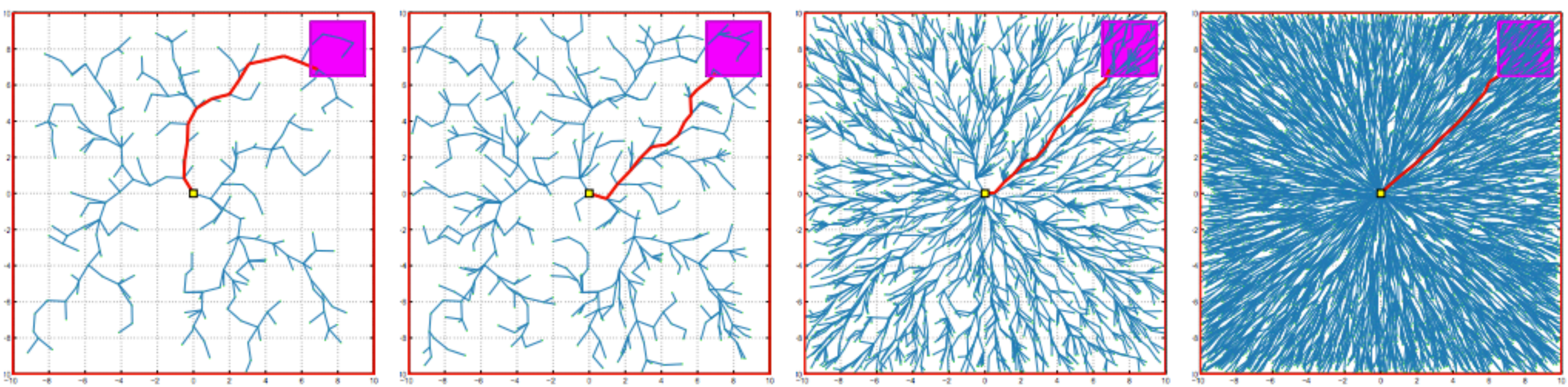
- Asymptotically optimal
- Main idea:
  - Swap new point in as parent for nearby vertices who can be reached along shorter path through new point than through their original (current) parent

RRT*

RRT



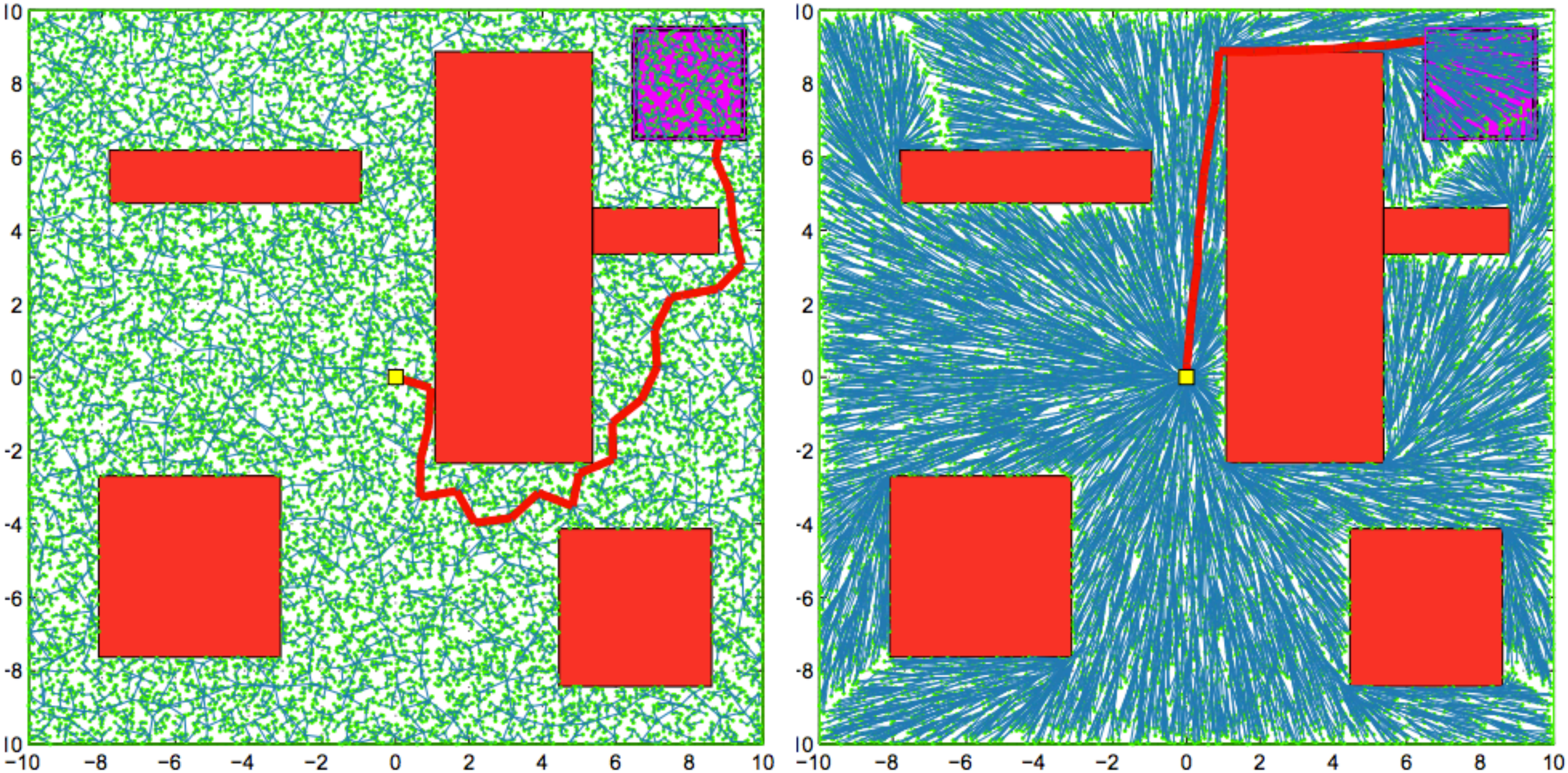
RRT*



Source: Karaman and Frazzoli

RRT*

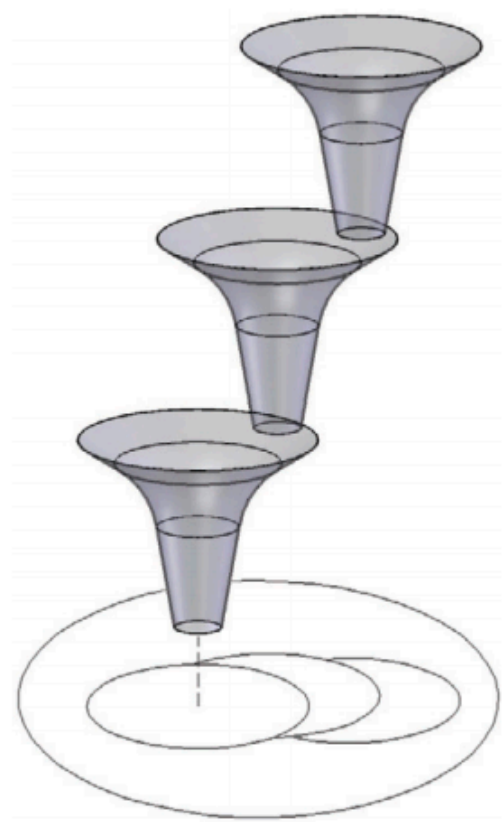
RRT RRT*



Source: Karaman and Frazzoli

LQR-trees (Tedrake, IJRR 2010)

- Idea: grow a randomized tree of stabilizing controllers to the goal
  - Like RRT
  - Can discard sample points in already stabilized region



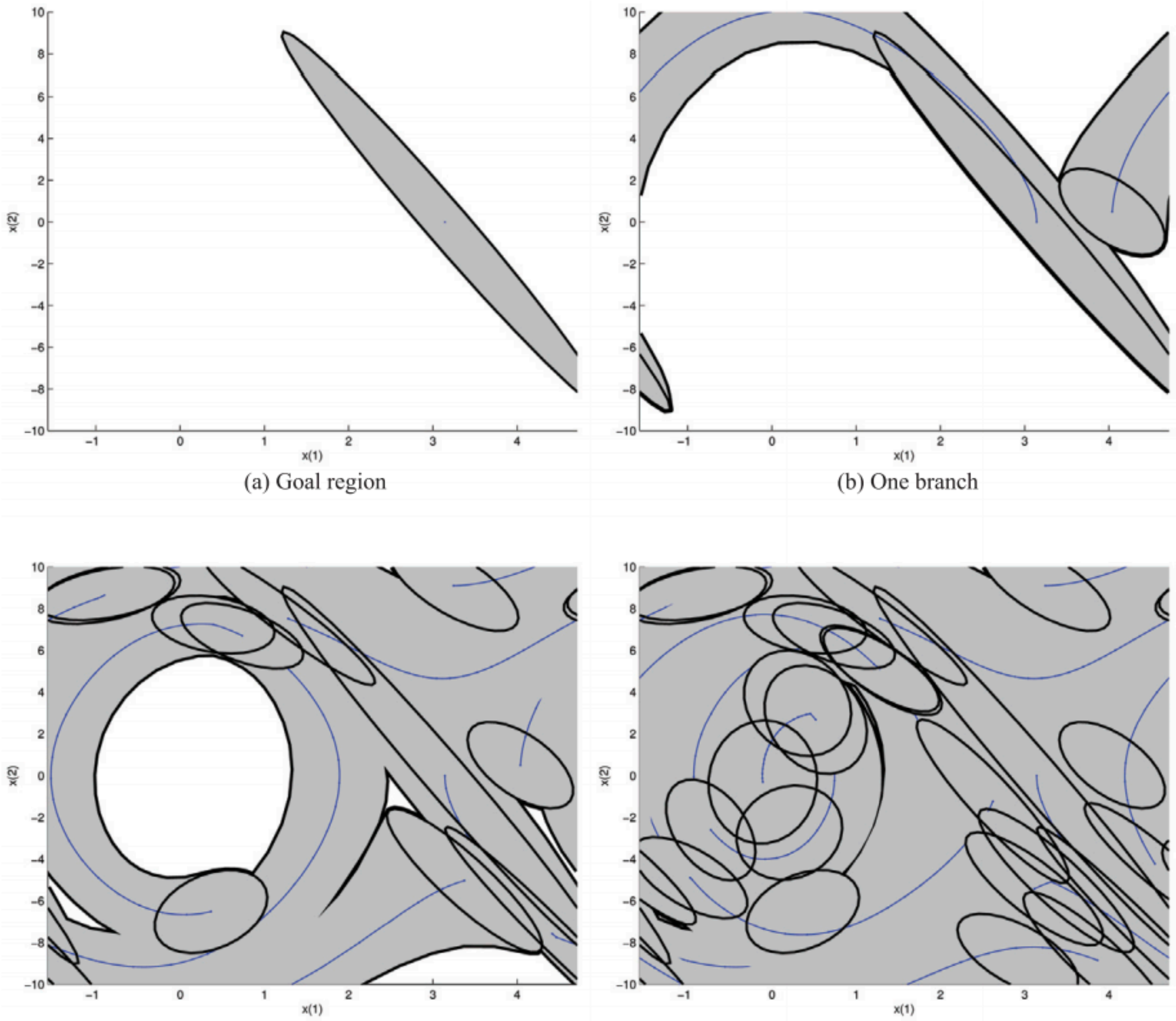
LQR-trees (Tedrake)

Algorithm 1 LQR-tree ($\mathbf{f}, \mathbf{x}_G, \mathbf{u}_G, \mathbf{Q}, \mathbf{R}$)

$[\mathbf{A}, \mathbf{B}] \Leftarrow$ linearization of $\mathbf{f}(\mathbf{x}, \mathbf{u})$ around $(\mathbf{x}_G, \mathbf{u}_G)$
$[\mathbf{K}, \mathbf{S}] \Leftarrow \mathrm{LQR}(\mathbf{A}, \mathbf{B}, \mathbf{Q}, \mathbf{R})$
$\rho_c \Leftarrow$ level set computed as described in §3.1.1
T.init($\{\mathbf{x}_g, \mathbf{u}_g, \mathbf{S}, \mathbf{K}, \rho_c$, NULL$\}$)
for $k = 1$ to K **do**
  $\mathbf{x}_{\text{rand}} \Leftarrow$ random sample
  if $\mathbf{x}_{\text{rand}} \in \mathcal{C}_k$ **then**
    continue
  end if
  $[t, \mathbf{x}_0(t), \mathbf{u}_0(t)]$ from trajectory optimization with a "final tree constraint"
  if $\mathbf{x}_0(t_f) \notin \mathcal{T}_k$ **then**
    continue
  end if
  $[\mathbf{K}(t), \mathbf{S}(t)]$ from time-varying LQR
  $\rho_c \Leftarrow$ level set computed as in §3.1.1
  $i \Leftarrow$ pointer to branch in T containing $\mathbf{x}_0(t_f)$
  T.add-branch($\mathbf{x}_0(t), \mathbf{u}_0(t), \mathbf{S}(t), \mathbf{K}(t), \rho_c, i$)
end for

Ck: stabilized region after iteration k

LQR-trees (Tedrake)



(a) Goal region (b) One branch

(c) Six branches (d) Thirteen branches

Smoothing

Randomized motion planners tend to find not so great paths for execution: very jagged, often much longer than necessary.

→ In practice: do smoothing before using the path

- Shortcutting:
  - along the found path, pick two vertices x_{t1}, x_{t2} and try to connect them directly (skipping over all intermediate vertices)
- Nonlinear optimization for optimal control
  - Allows to specify an objective function that includes smoothness in state, control, small control inputs, etc.