Rapidly Exploring Random Trees

- **Idea:** aggressively probe and explore the C-space by **expanding incrementally** from an initial configuration q_0
- The explored territory is marked by a **tree rooted at** $\boldsymbol{q_0}$

45 iterations q_g

2345 iterations

RRTs

- The algorithm: Given C and q_0

Algorithm 1: RRT

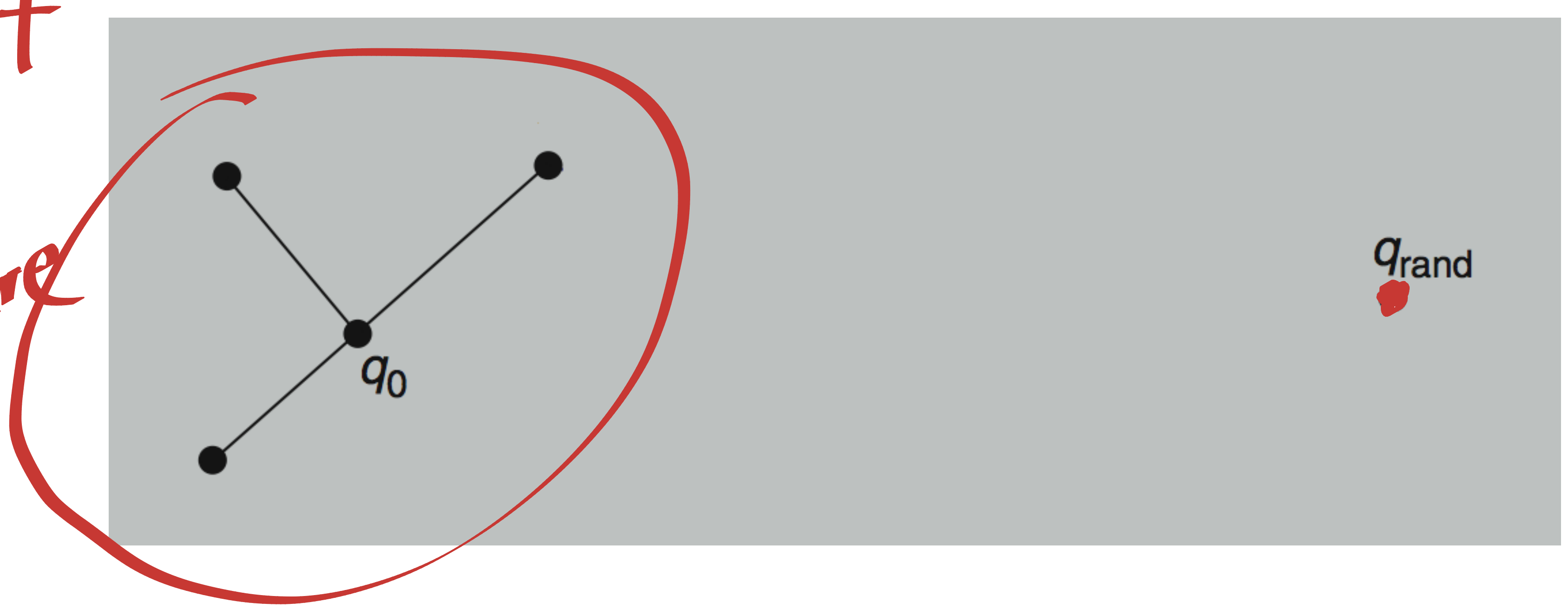
$G.\text{init}(q_0)$ initiate a tree structure with configuration q_0
repeat
  $q_{rand} \rightarrow \text{RANDOM\_CONFIG}(\mathcal{C})$
  $q_{near} \leftarrow \text{NEAREST}(G, q_{rand})$
  $G.\text{add\_edge}(q_{near}, q_{rand})$
until *condition*

Sample from a **bounded region** centered around q_0

E.g. an axis-aligned relative random translation or random rotation

(but recall sampling over rotation spaces problem)

current tree structure G



RRTs

- The algorithm

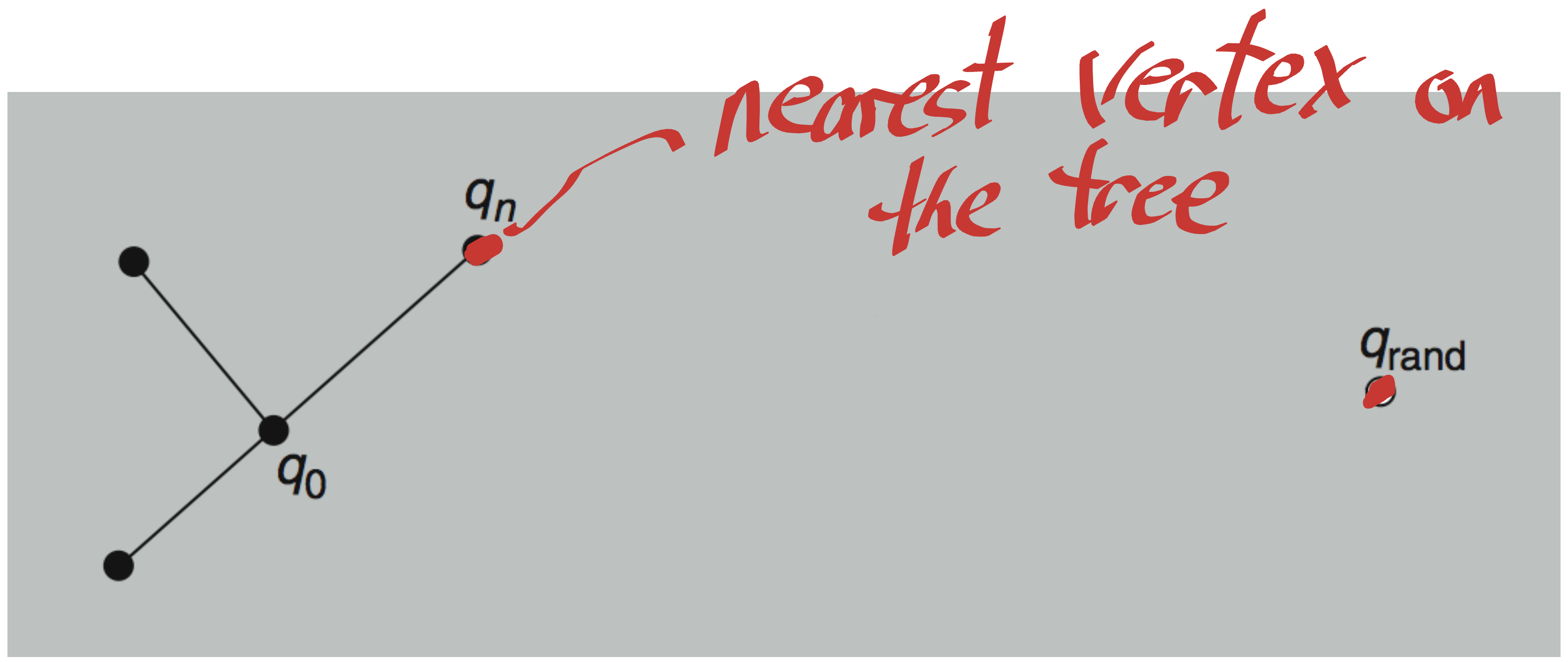
Algorithm 1: RRT

G.init(q_0)
repeat
  $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
  $q_{near} \leftarrow$ NEAREST(G, q_{rand})
  G.add_edge(q_{near}, q_{rand})
until *condition*

Finds closest vertex in G using a **distance function**

$$\rho \; : \; \mathcal{C} \times \mathcal{C} \rightarrow [0, \infty)$$

formally a ***metric*** defined on C



RRTs

- The algorithm

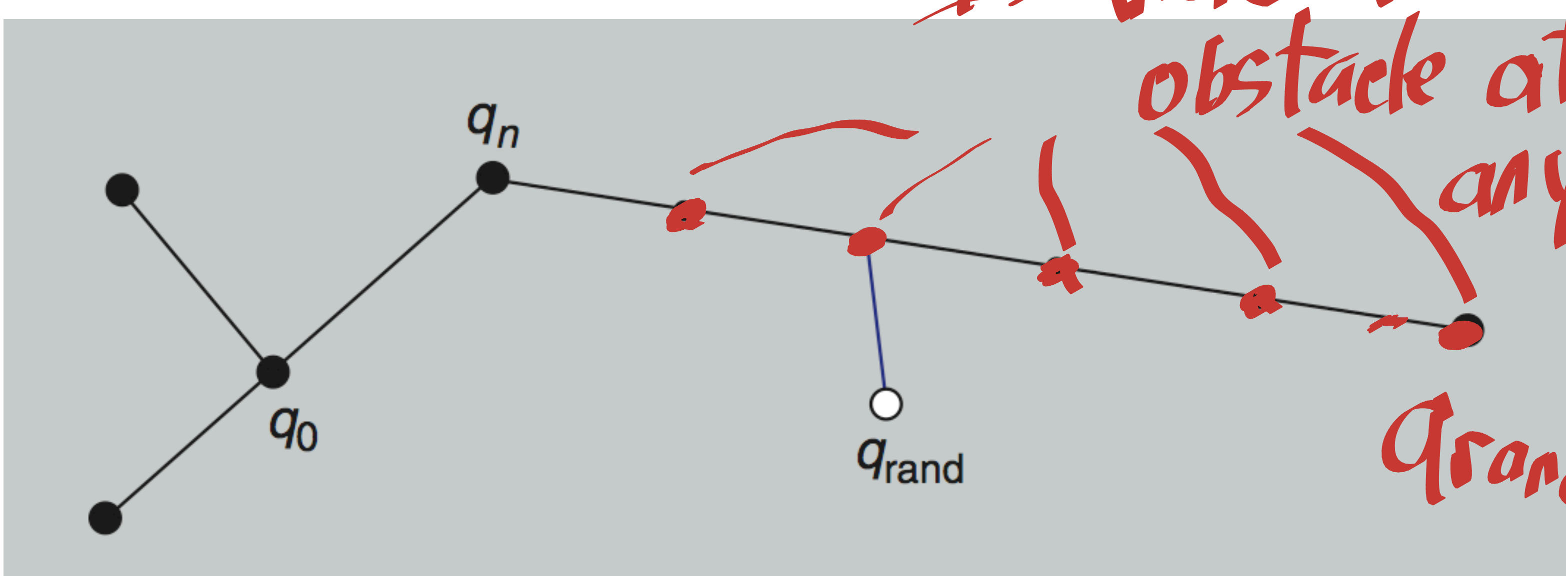
Algorithm 1: RRT

G.init(q_0)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST(G, q_{rand})
    G.add_edge(q_{near}, q_{rand})
until *condition*

Several stategies to find q_{near} given the closest vertex on G:

- Take closest vertex
- Check intermediate points at regular intervals and split edge at q_{near}

Is there an obstacle at any points in between



qrand

RRTs

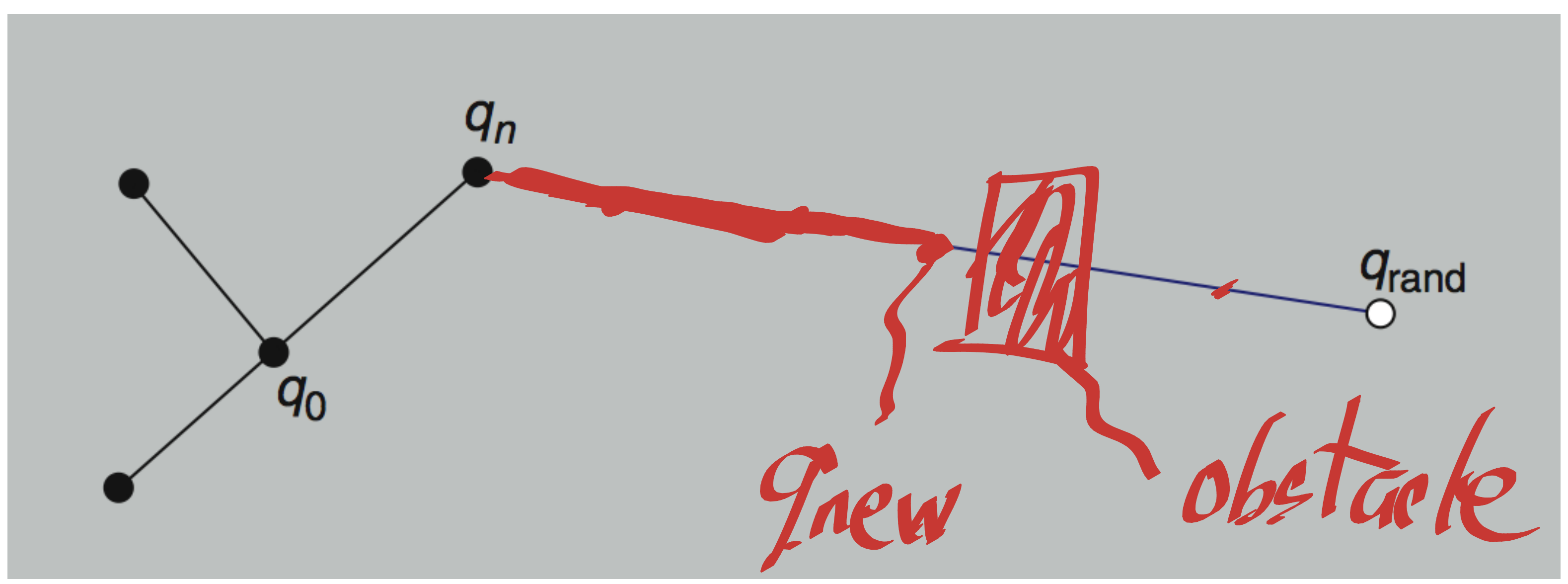
- The algorithm

Algorithm 1: RRT

$G.\text{init}(q_0)$
repeat
    $q_{rand} \rightarrow \text{RANDOM\_CONFIG}(\mathcal{C})$
    $q_{near} \leftarrow \text{NEAREST}(G, q_{rand})$
    $G.\text{add\_edge}(q_{near}, q_{rand})$
until *condition*

Connect nearest point with random point using a **local planner** that travels from q_{near} to q_{rand}

- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}

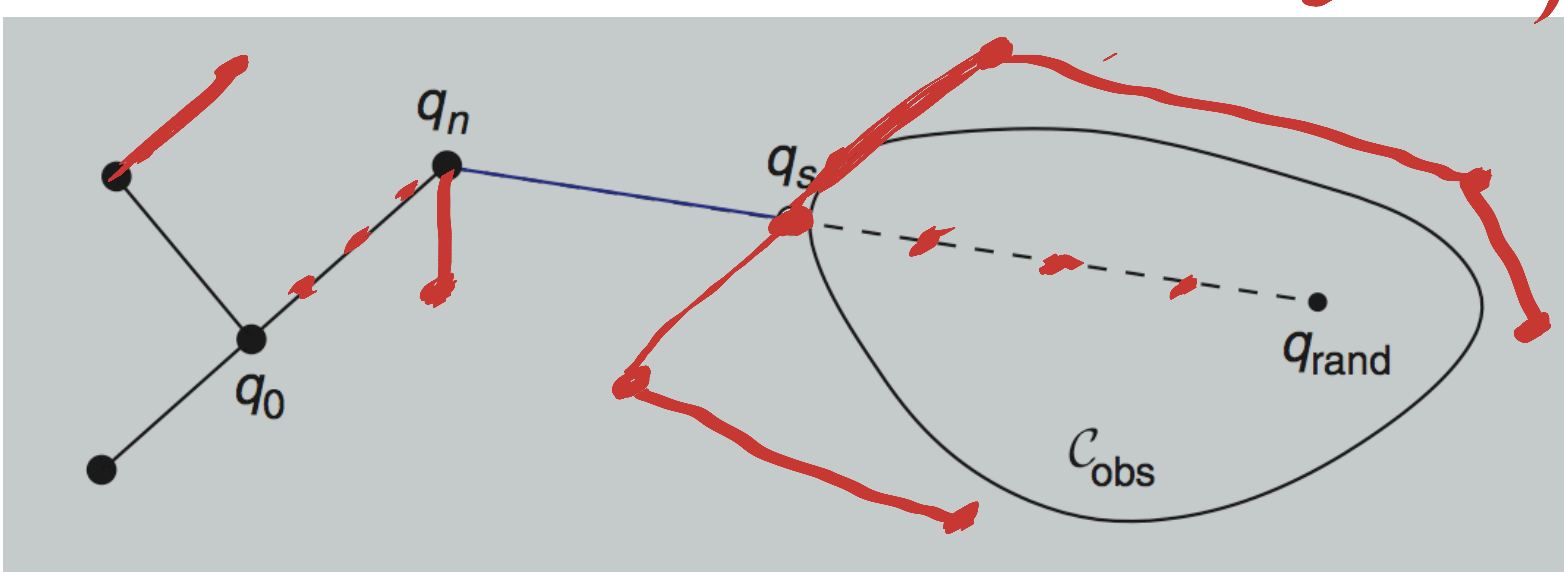


RRTs

- The algorithm

Algorithm 1: RRT

$G.\text{init}(q_0)$
repeat
  $q_{rand} \rightarrow \text{RANDOM\_CONFIG}(\mathcal{C})$
  $q_{near} \leftarrow \text{NEAREST}(G, q_{rand})$
  $G.\text{add\_edge}(q_{near}, q_{rand})$
until *condition* (q_{new} is close to goal)



Connect nearest point with random point using a **local planner** that travels from q_{near} to q_{rand}

- No collision: add edge
- Collision: new vertex is q_i, as close as possible to C_{obs}

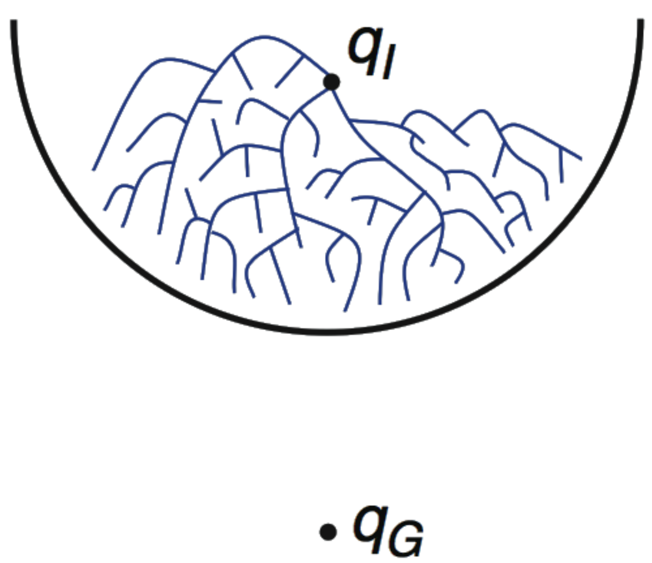
RRTs

- How to perform path planning with RRTs?
  1. Start RRT at q_I
  2. At every, say, 100th iteration, force $q_{rand} = q_G$
  3. If q_G is reached, problem is solved

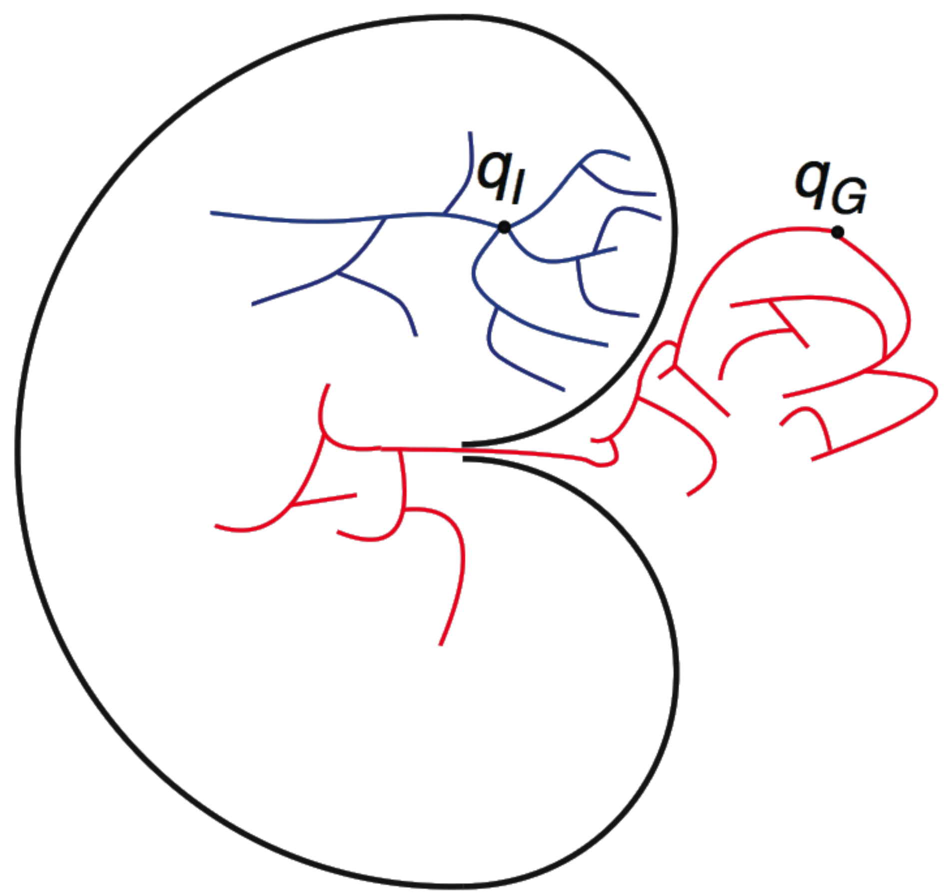
- Why not picking q_G every time?
- This will fail and waste much effort in running into C_{Obs} instead of exploring the space

RRTs

- However, some problems require more effective methods: **bidirectional search**
- Grow **two** RRTs, one from q_I, one from q_G
- In every other step, try to extend each tree towards the newest vertex of the other tree



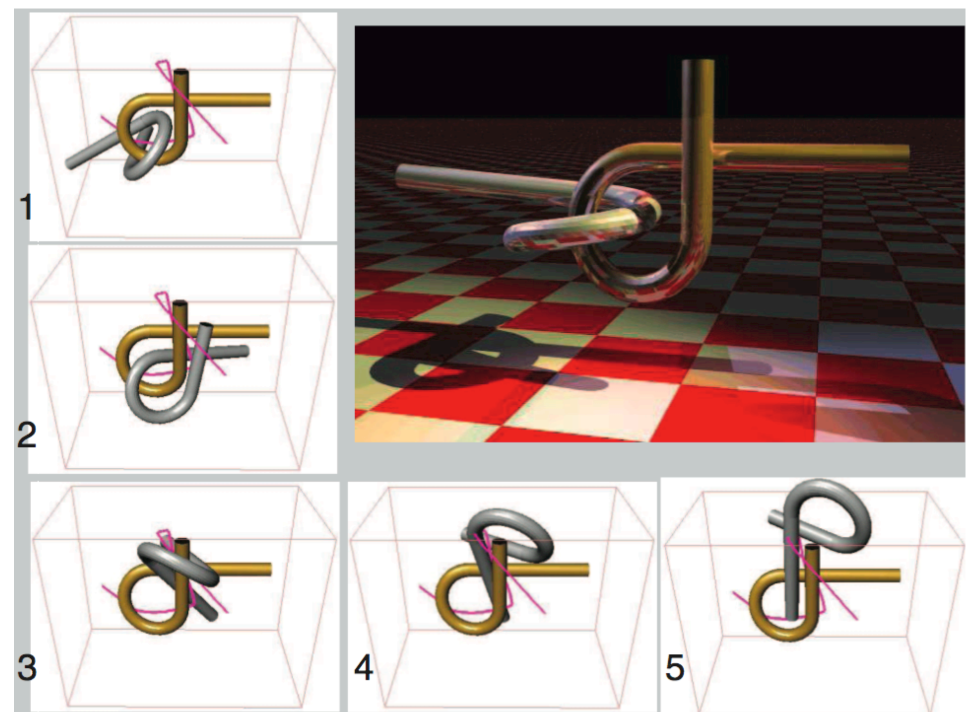
Filling a well



A bug trap

RRTs

- RRTs are popular, many extensions exist: real-time RRTs, anytime RRTs, for dynamic environments etc.
- **Pros:**
  - Balance between greedy search and exploration
  - Easy to implement
- **Cons:**
  - Metric sensivity
  - Unknown rate of convergence



Alpha 1.0 puzzle. Solved with bidirectional RRT

From Road Maps to Paths

- All methods discussed so far **construct a road map** (without considering the query pair q_I and q_G)

- Once the investment is made, the **same road map** can be reused for **all** queries (provided world and robot do not change)

1. **Find** the cell/vertex that contain/is close to q_I and q_G (not needed for visibility graphs)
2. **Connect** q_I and q_G to the road map
3. **Search** the road map for a path from q_I to q_G

Sampling-Based Planning

Wrap Up

- Sampling-based planners are **more efficient** in most **practical problems** but offer weaker guarantees
- They are **probabilistically complete**: the probability tends to 1 that a solution is found if one exists (otherwise it may still run forever)
- Performance degrades in problems with **narrow passages**. Subject of active research
- Widely used. Problems with high-dimensional and complex C-spaces are still computationally hard