# Motion Planning

**Nicolas Perrin-Gilbert**

INSTITUT
DES SYSTÈMES
INTELLIGENTS
ET DE ROBOTIQUE

# One of the reference books:

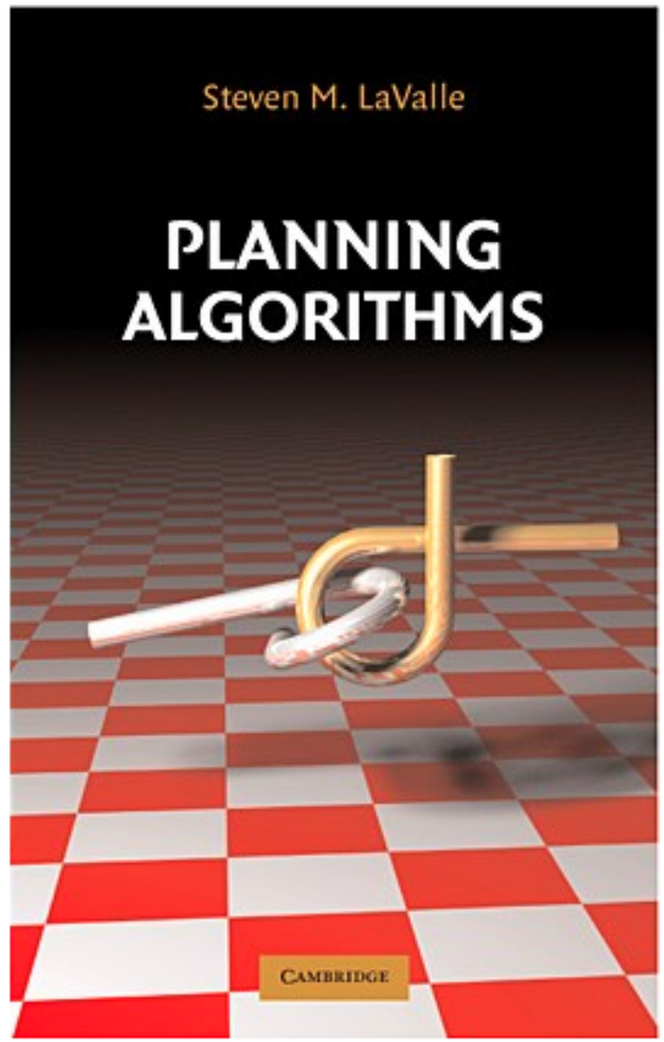
**Planning Algorithms**

Steven M. LaValle

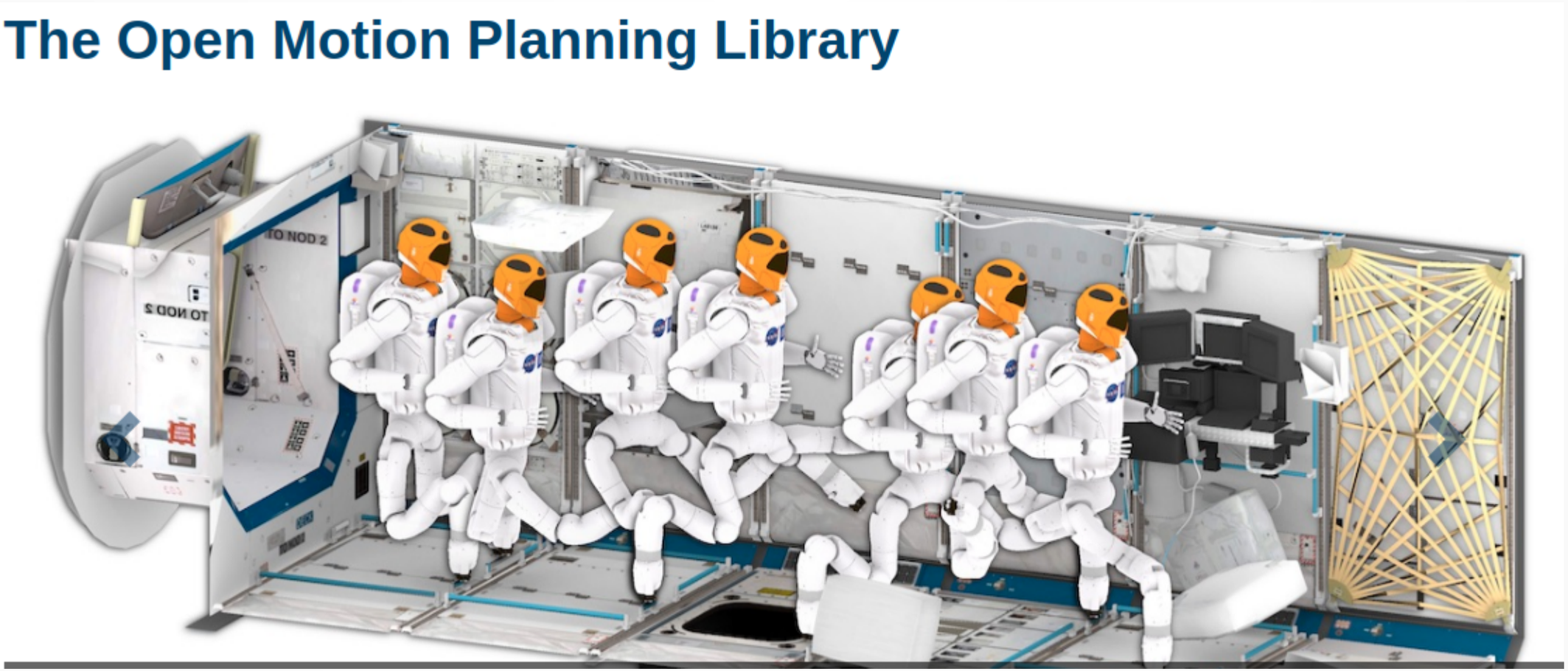
Cambridge University Press

Available at:

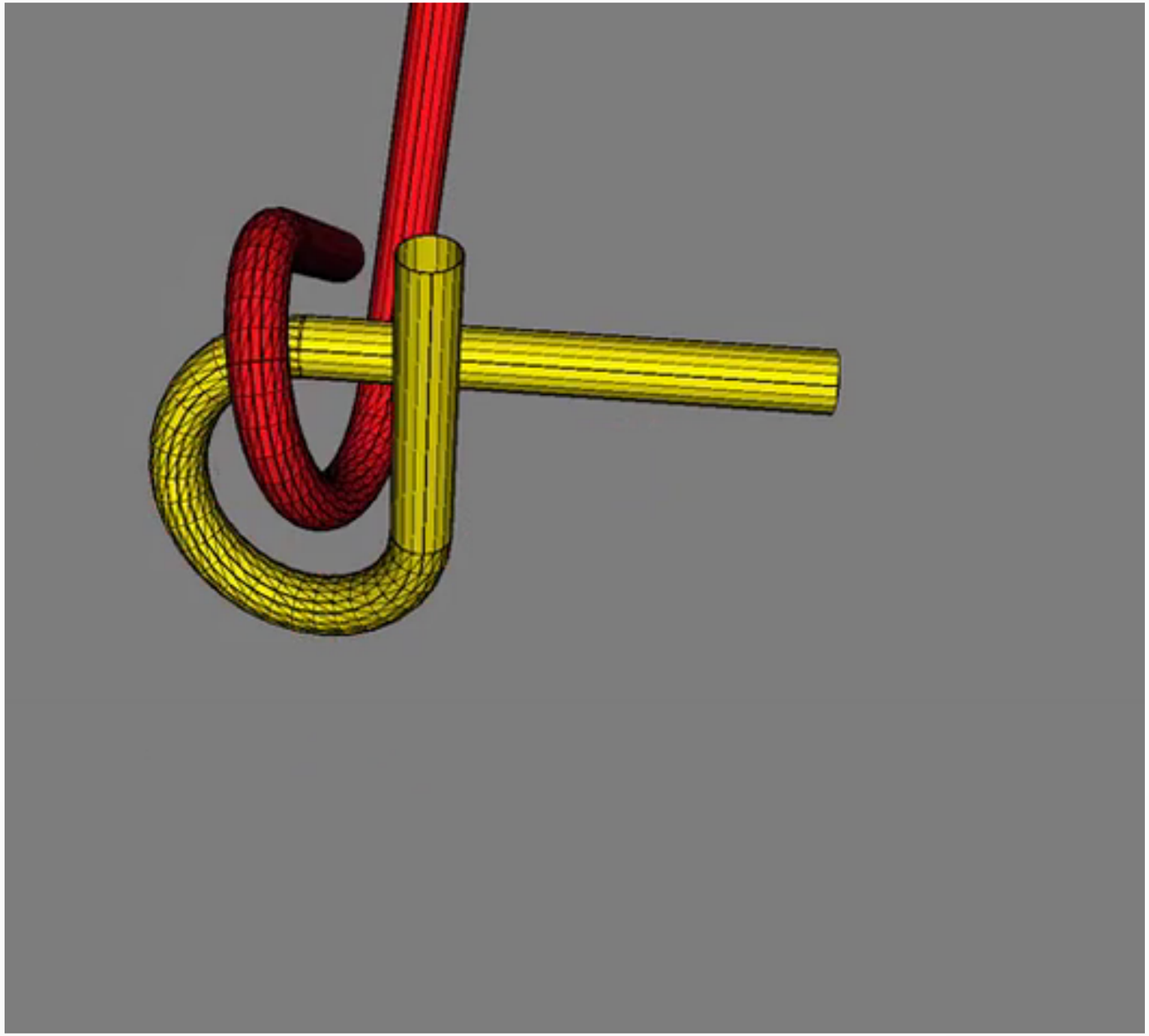
http://lavalle.pl/planning/

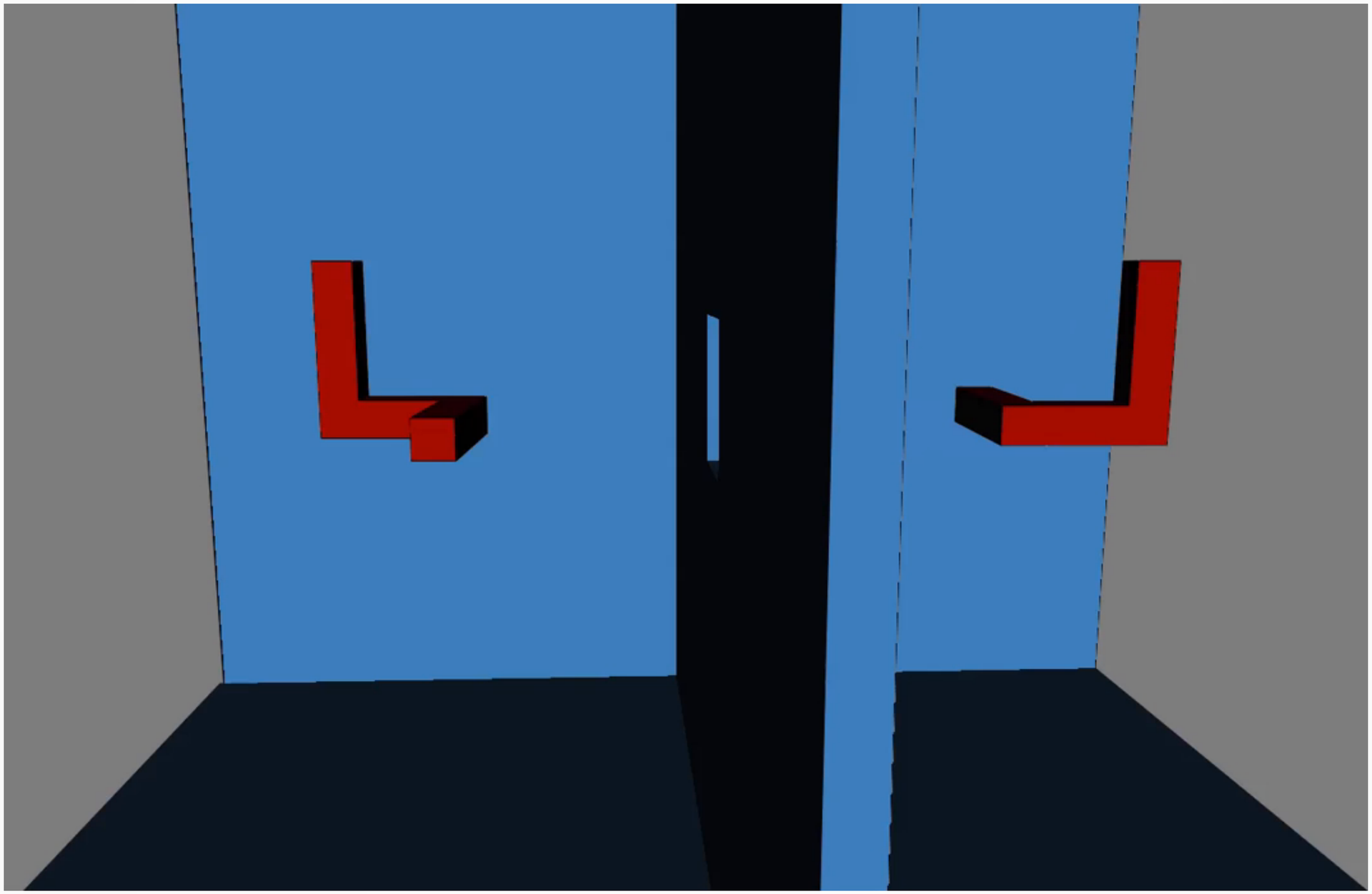


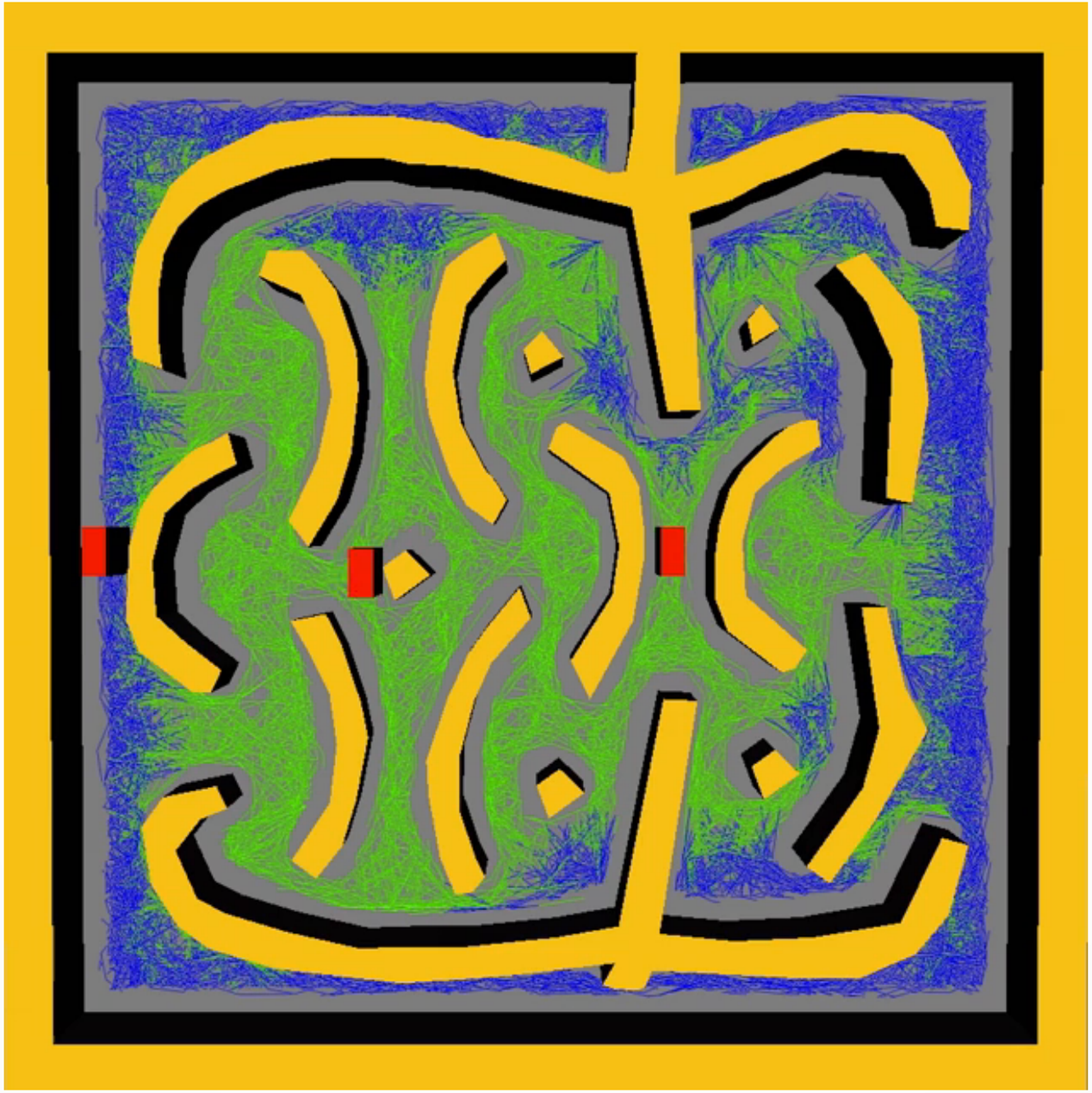
# Reference library (C++/Python):

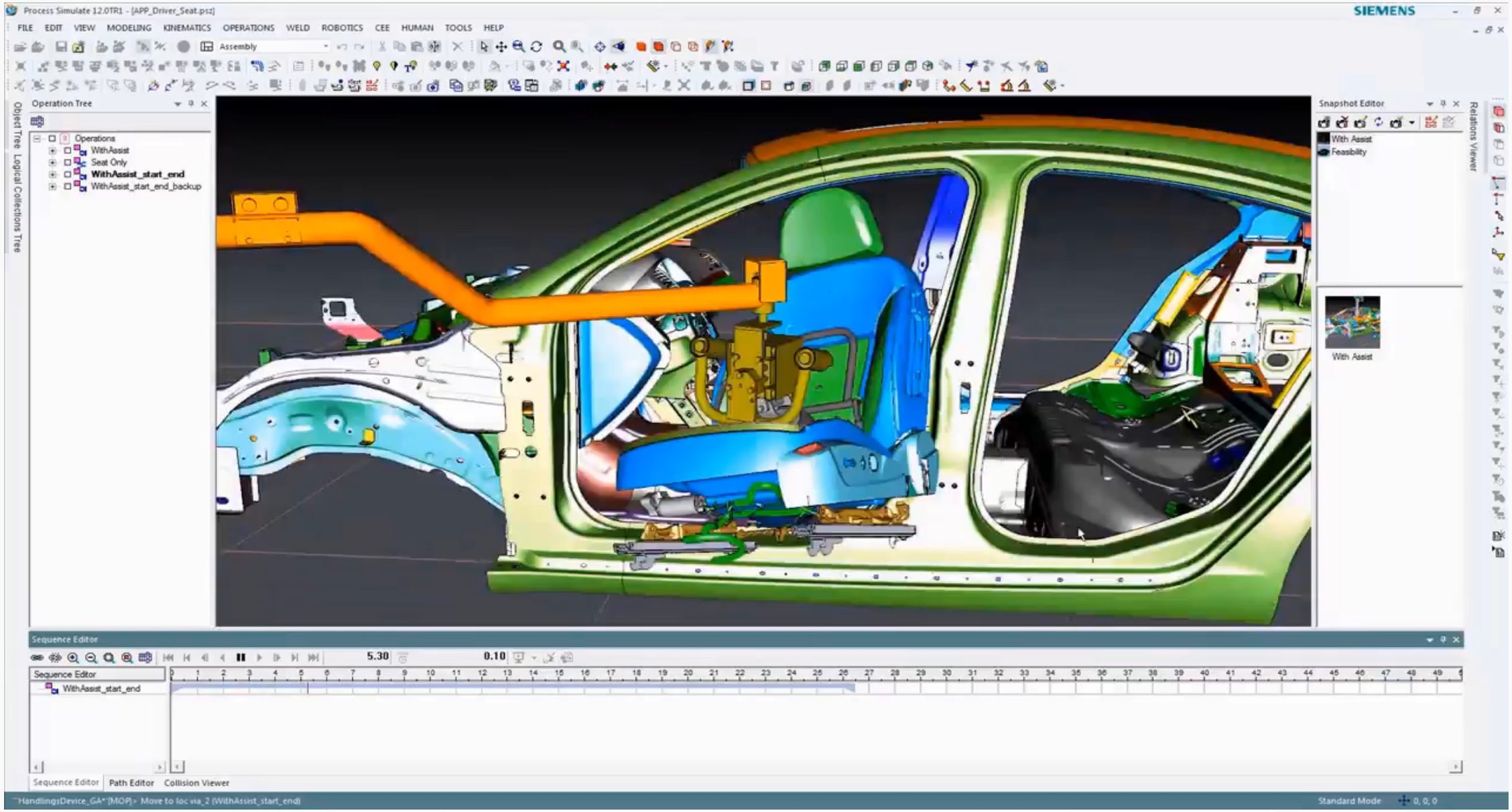



Website: **ompl.kavrakilab.org**







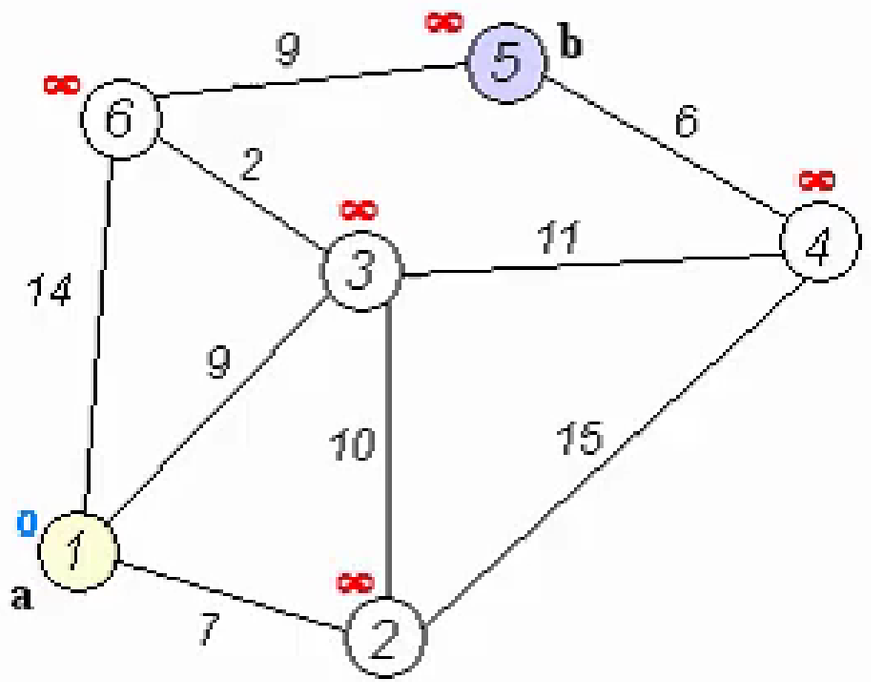

Process Simulate 12.0TR1 - [APP_Driver_Seat.psz]
SIEMENS
FILE EDIT VIEW MODELING KINEMATICS OPERATIONS WELD ROBOTICS CEE HUMAN TOOLS HELP
Assembly
Operation Tree
Operations
WithAssist
Seat Only
WithAssist_start_end
WithAssist_start_end_backup
Snapshot Editor
With Assist
Feasibility
With Assist
Object Tree Logical Collections Tree
Relations Viewer
Sequence Editor
5.30
0.10
Sequence Editor
WithAssist_start_end
Sequence Editor Path Editor Collision Viewer
HandlingsDevice_GA* (MOP)> Move to loc via_2 (WithAssist_start_end)
Standard Mode
0, 0, 0

# Preamble: path planning in a graph

Shortest path: **Dijkstra's algorithm** (1959)

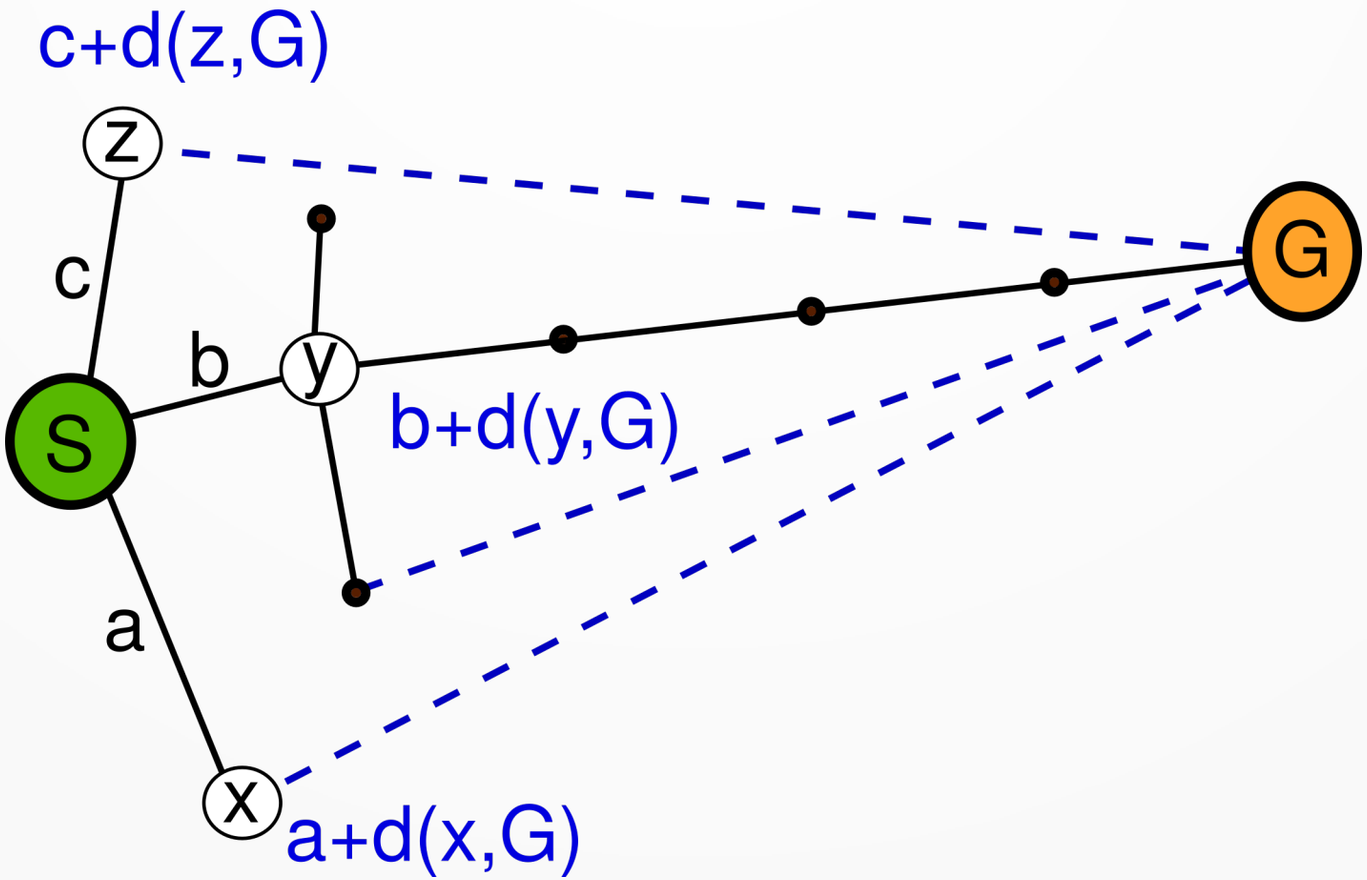
Complexity: O(a+n log n)

→ Slow if there are many vertices, or many edges.



# Preamble: path planning in a graph

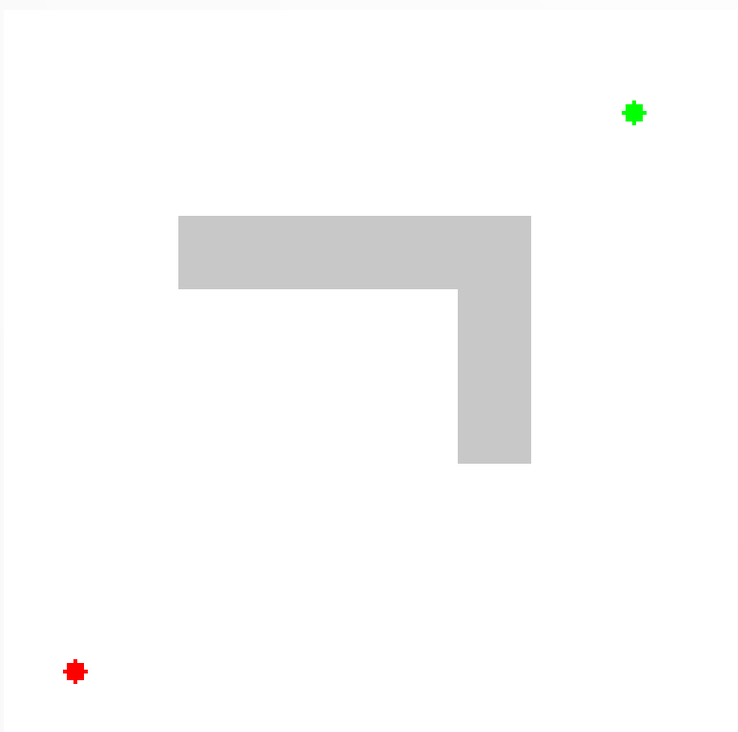
Dijkstra + heuristic: **A* algorithm** *(Hart, Nilsson & Raphael 1968)*

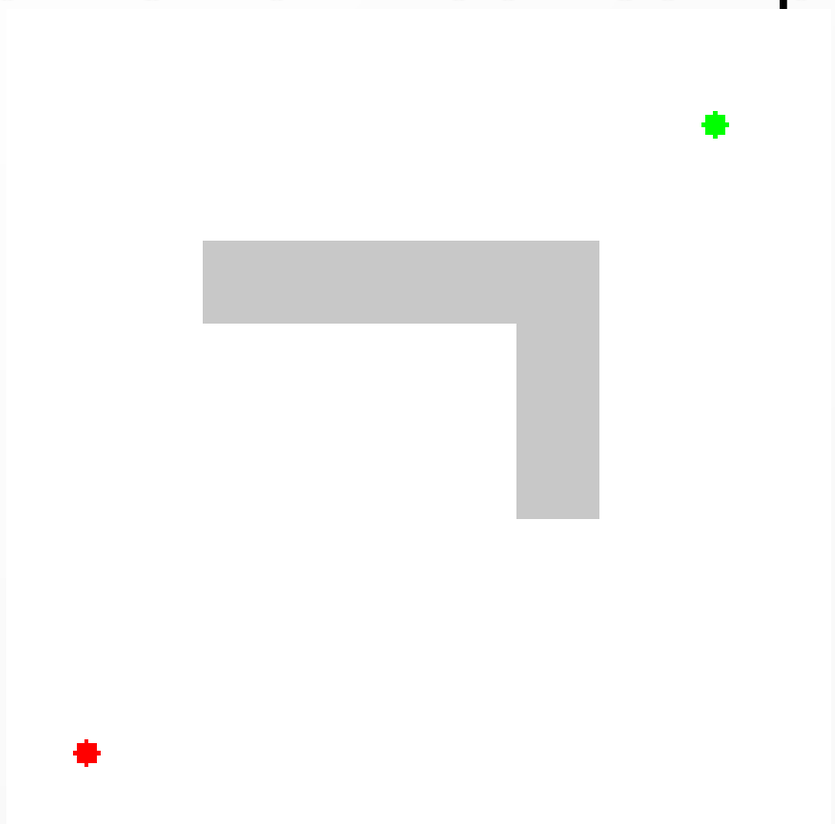


# Preamble: path planning in a graph

**A*:**

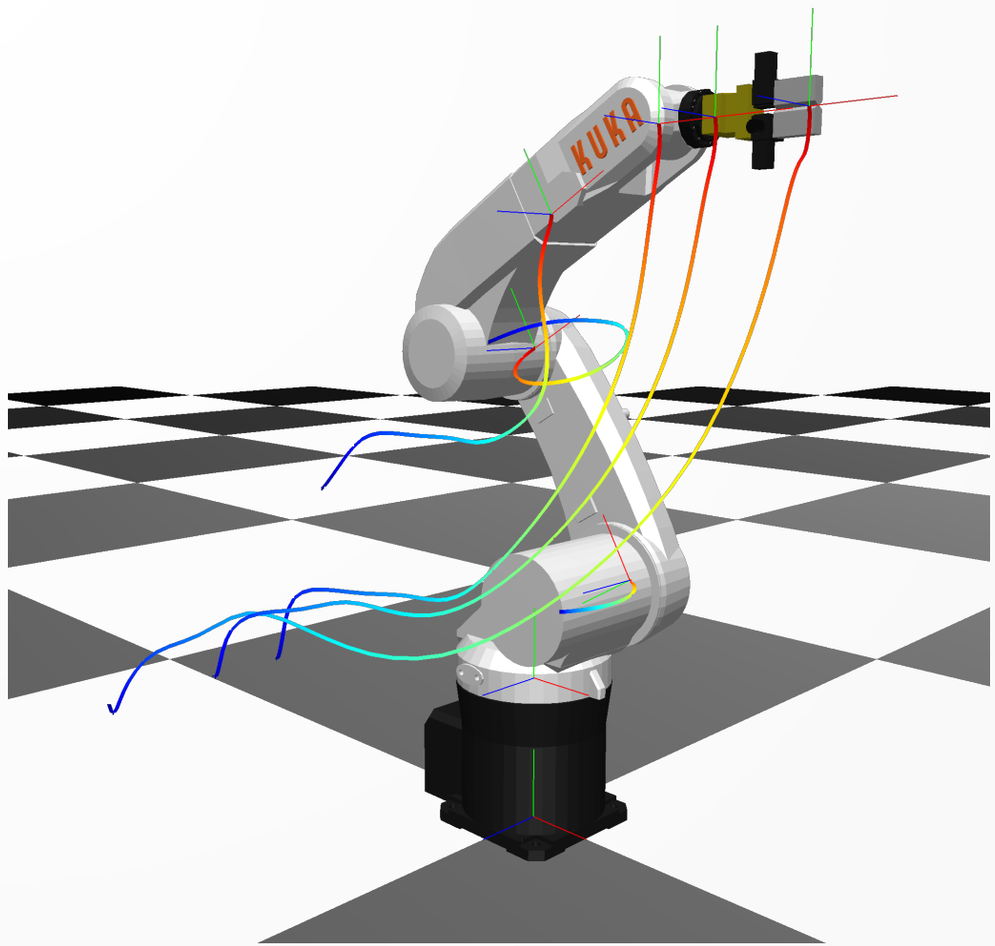
- A good heuristic is not always easy to find.
- Only efficient if the solution is not “too” complex.





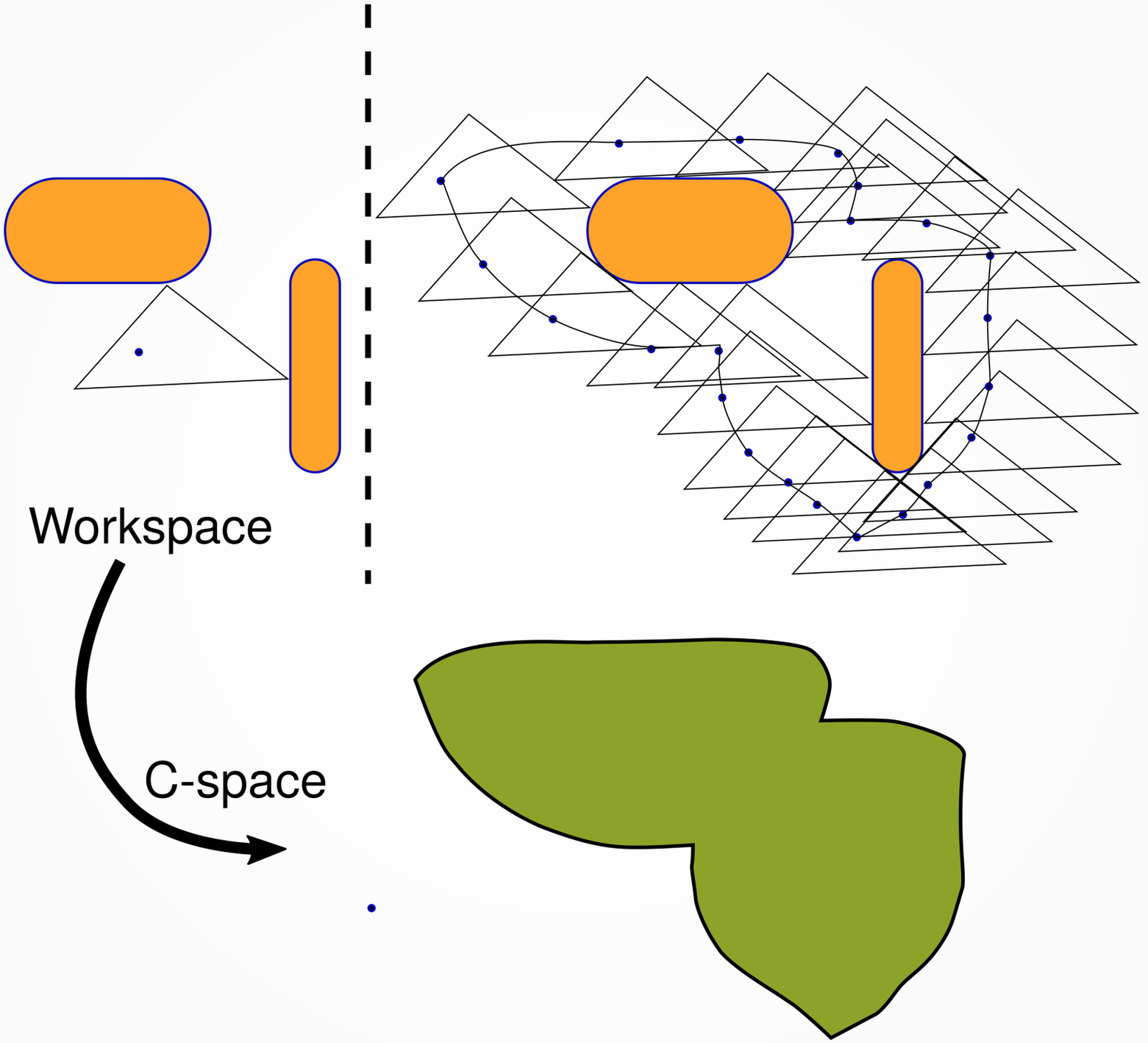
# Challenges

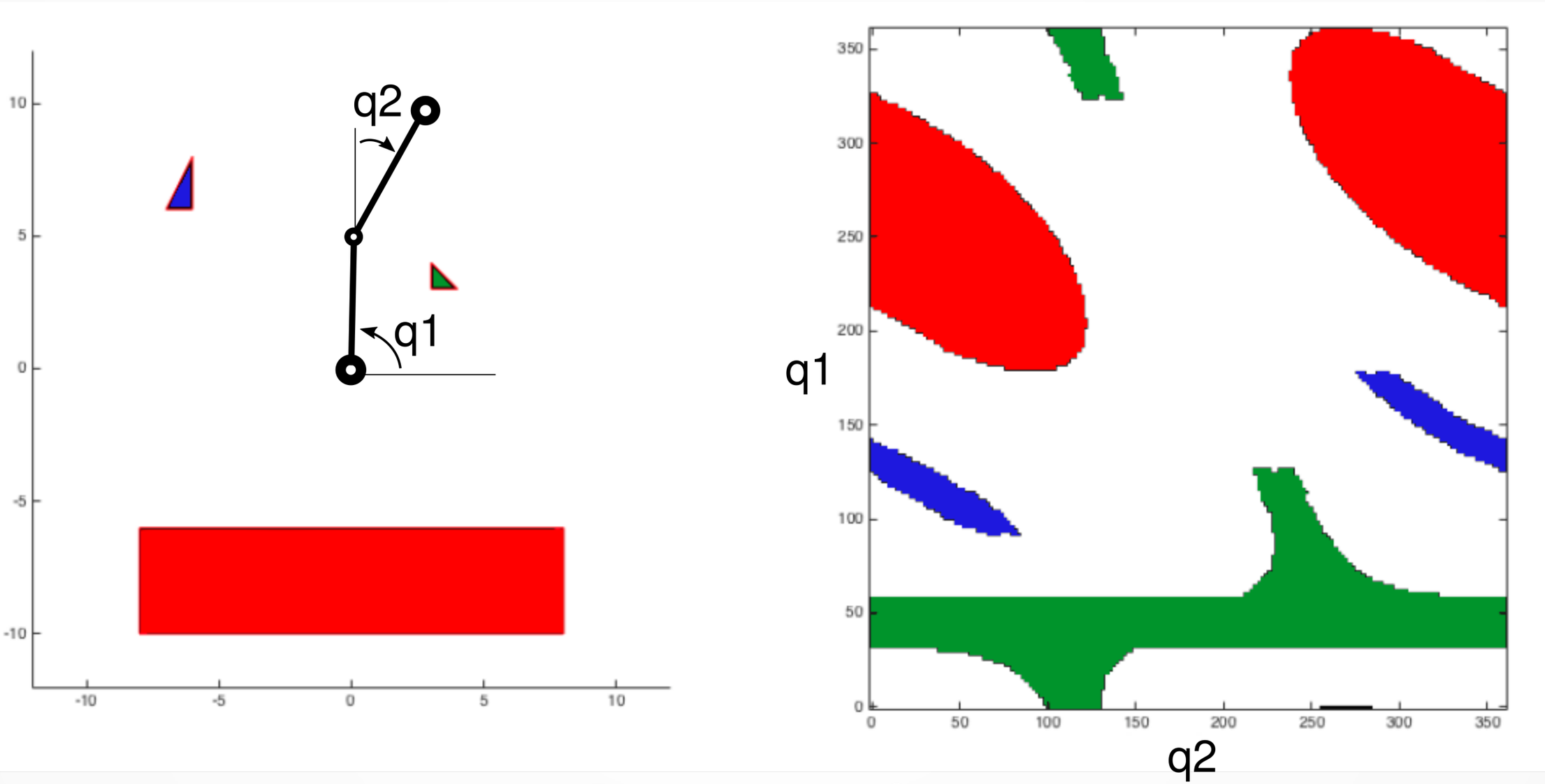
- Continuous, high-dimensional **configuration spaces**
- Continuous, high-dimensional **action spaces**



# Challenges

- The configuration space, or C-space, is a space where, conveniently, the robot is just a point (an element of a vector space). Remark: we, humans, never think about the C-space.
- Obstacles are known in the **workspace**, not directly in the **configuration space**.


Workspace
C-space


q2
q1
10
5
0
-5
-10
-10
-5
0
5
10
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300
350
q1
q2
Workspace
C-space

# Challenges

- No efficient way to know the obstacles in the C-space.

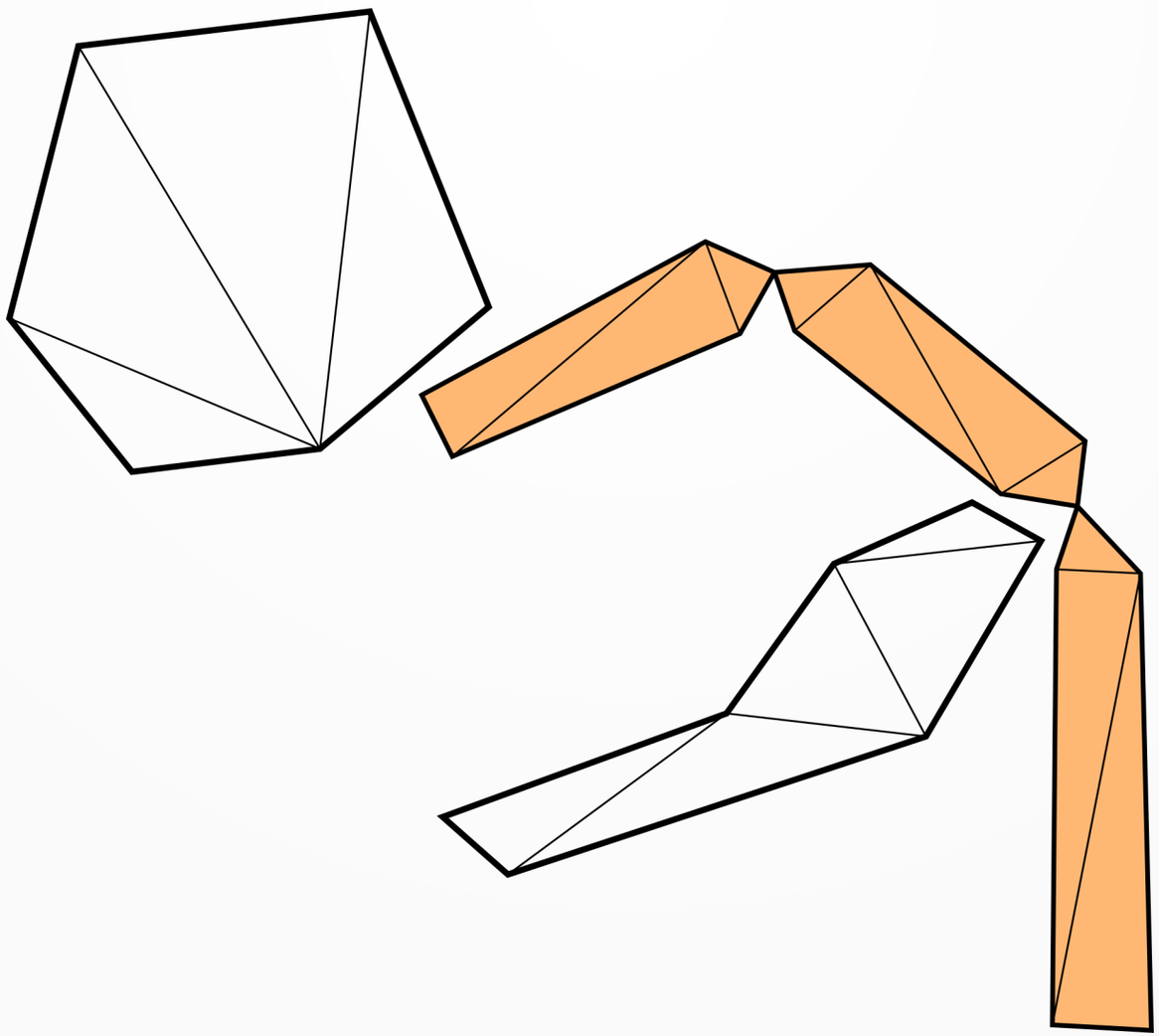
  → Most approaches perform motion planning “blindly” (without starting from the obstacles):

  - Configuration are first tried and then collisions are checked.
  - This leads to **“sampling-based”** motion planning, which relies on two main components:

    (1) **efficient random exploration**

    (2) **efficient collision checking**

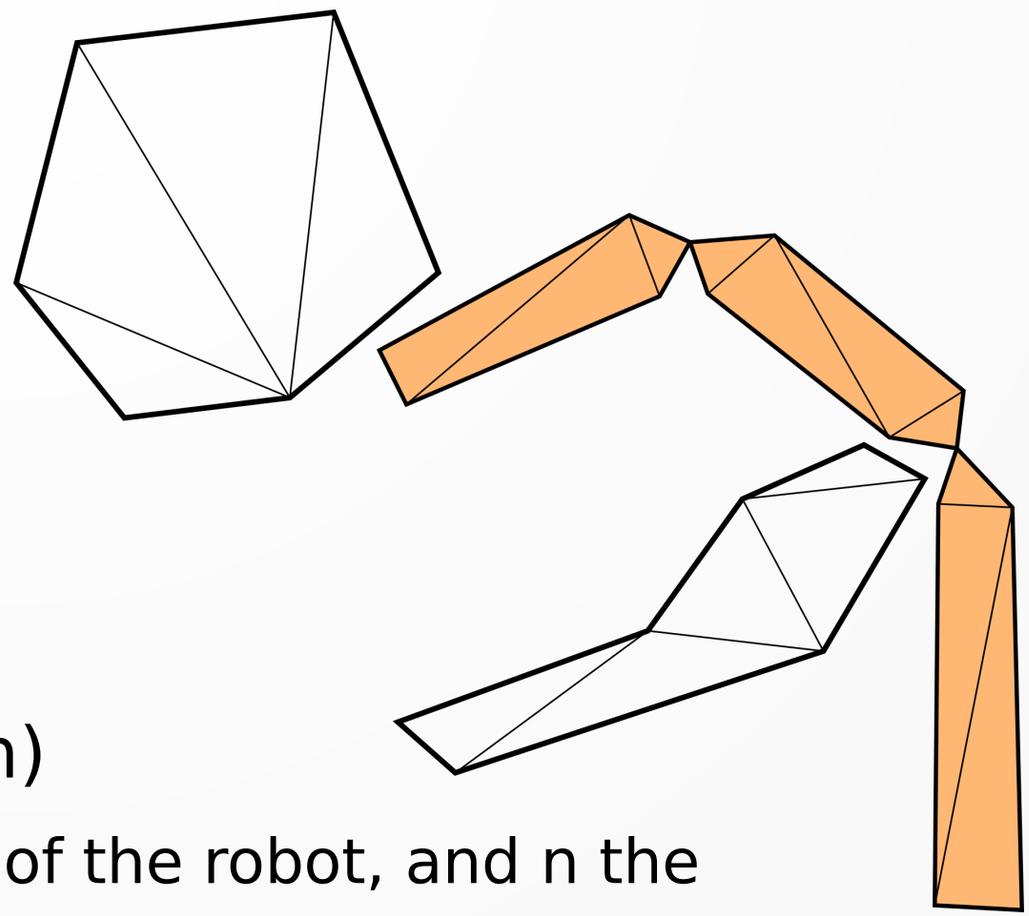
# Collision checking



Triangulation: see for instance Delaunay triangulation (1924)

# Collision checking

- Naive approach:
  - Test intersection for every possible pair of triangles *(that belong to different rigid bodies, one of these rigid bodies being part of the robot)*



- Complexity: close to n x (n+m)

  (with n the number of triangles of the robot, and n the number of triangles in the environment)

# Collision checking

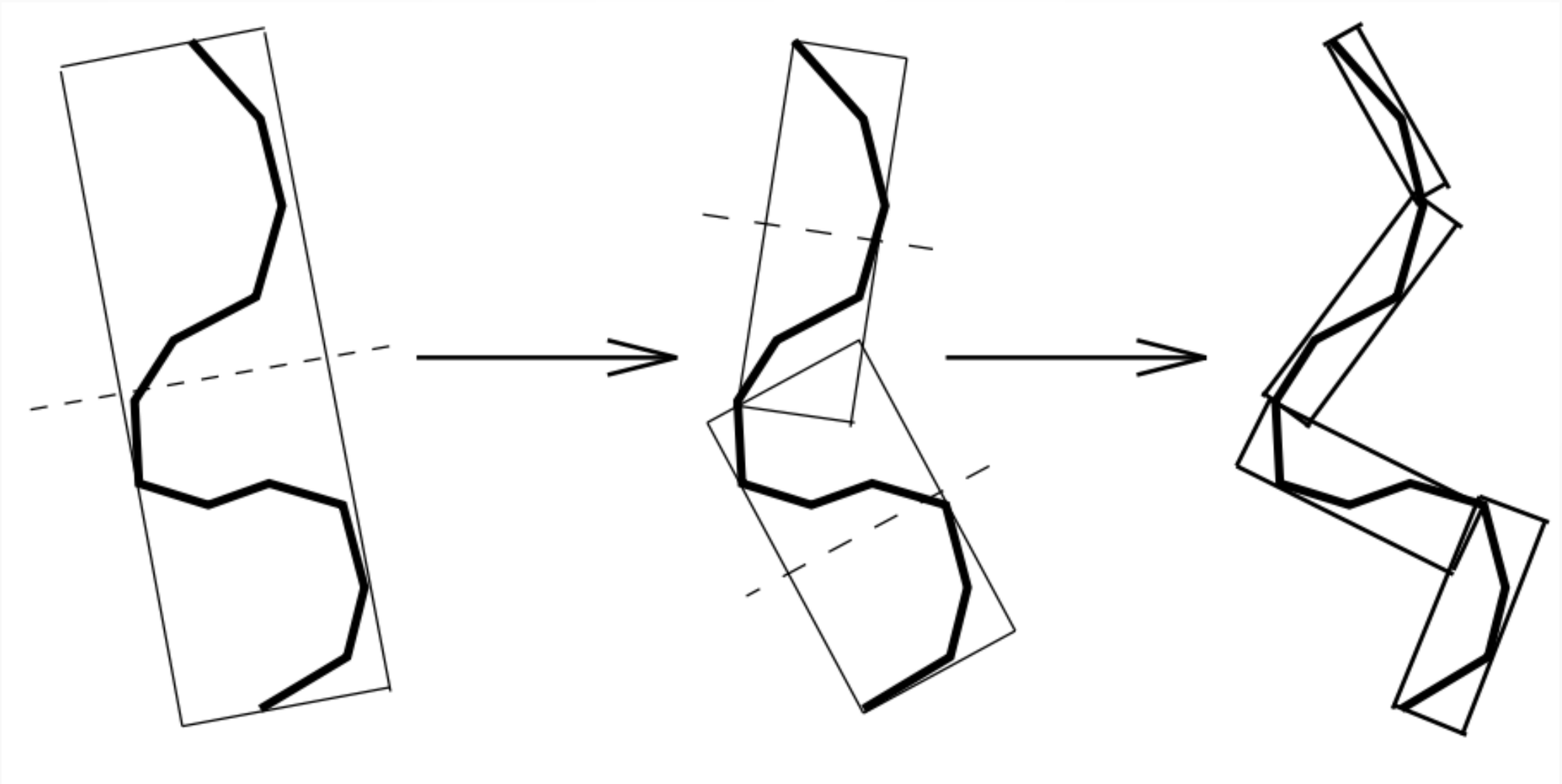
- Better approaches:
  - Incremental methods

    E.g.: **A fast algorithm for incremental distance calculation**, M.C. Lin & J.F. Canny, IEEE ICRA 1990

  - Hierarchical methods

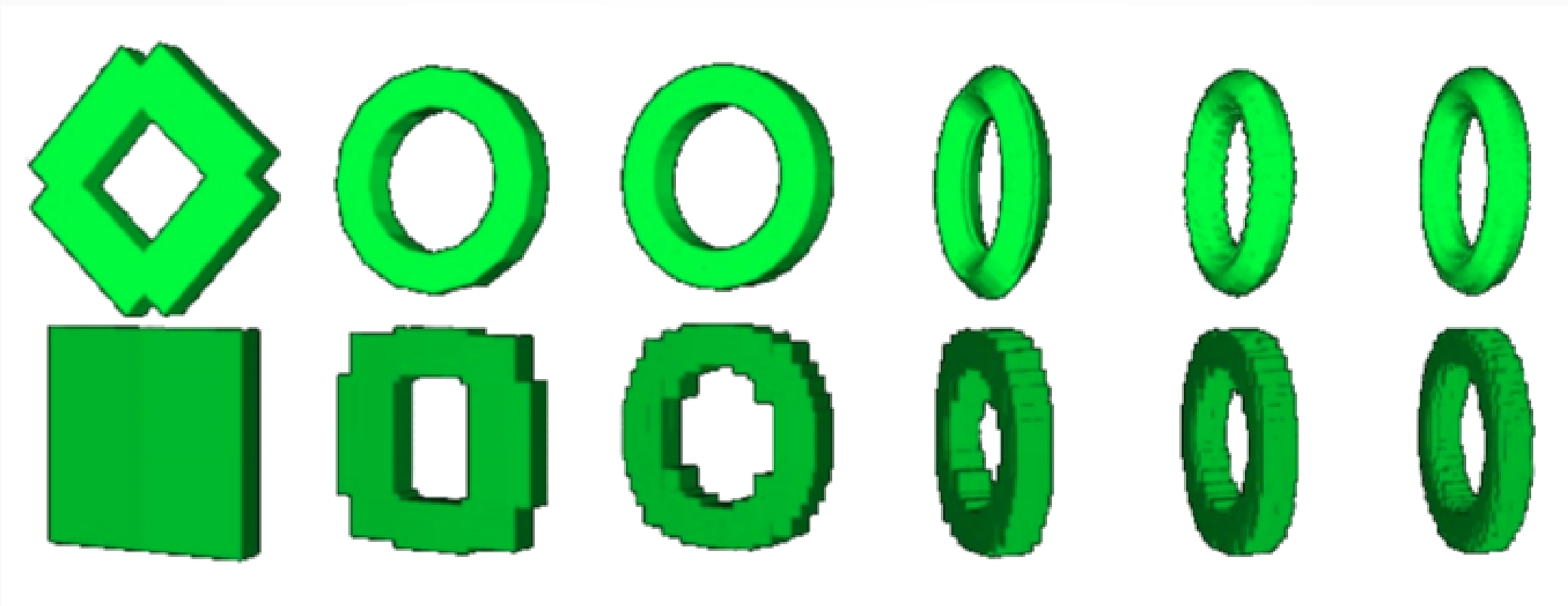
    E.g.: **OBBTree: A Hierarchical Structure for Rapid Inference Detection**, S. Gottschalk, M.C. Lin, D. Manocha, ACM SIGGRAPH 1996

# OBBTree



# OBBTree

OBB vs. AABB (Axis-Aligned Bounding Boxes):

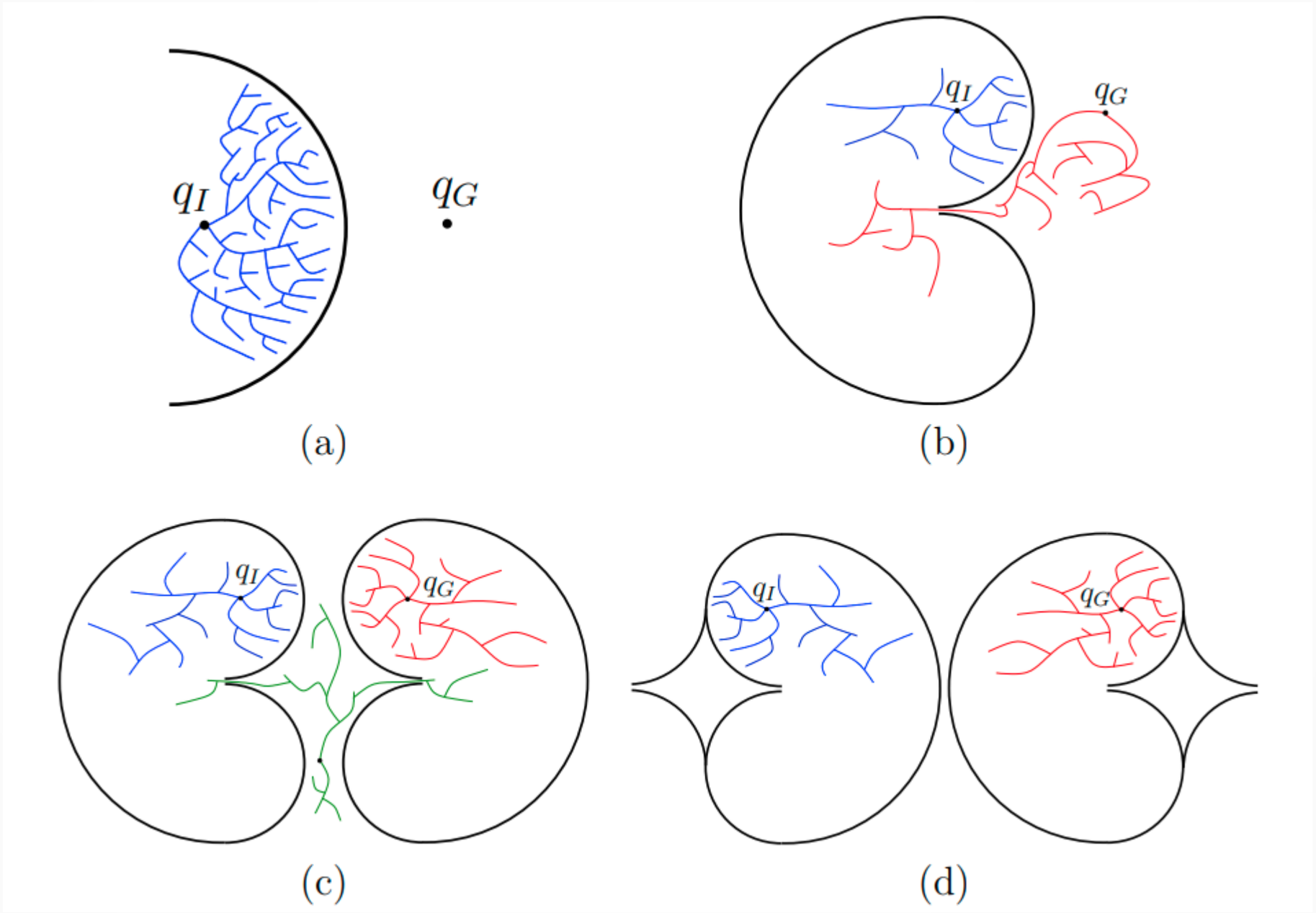


# Sampling-based exploration

- To simply find configurations, one possibility is to use *low-discrepancy sequences*, which cover the search space quickly and evenly (much better than uniformly distributed random samples).
- However, motion planning usually aims at finding paths, not only new configurations. **The standard approach are based on the iterative expansion of random trees.**

# Single-query path planning

- Unidirectional methods: one random tree
- Bidirectional methods: 2 random trees
- Multi-directional methods: more than 2 trees


$q_I$
$q_G$
(a)
$q_I$
$q_G$
(b)
$q_I$
$q_G$
(c)
$q_I$
$q_G$
(d)

# Selection-expansion

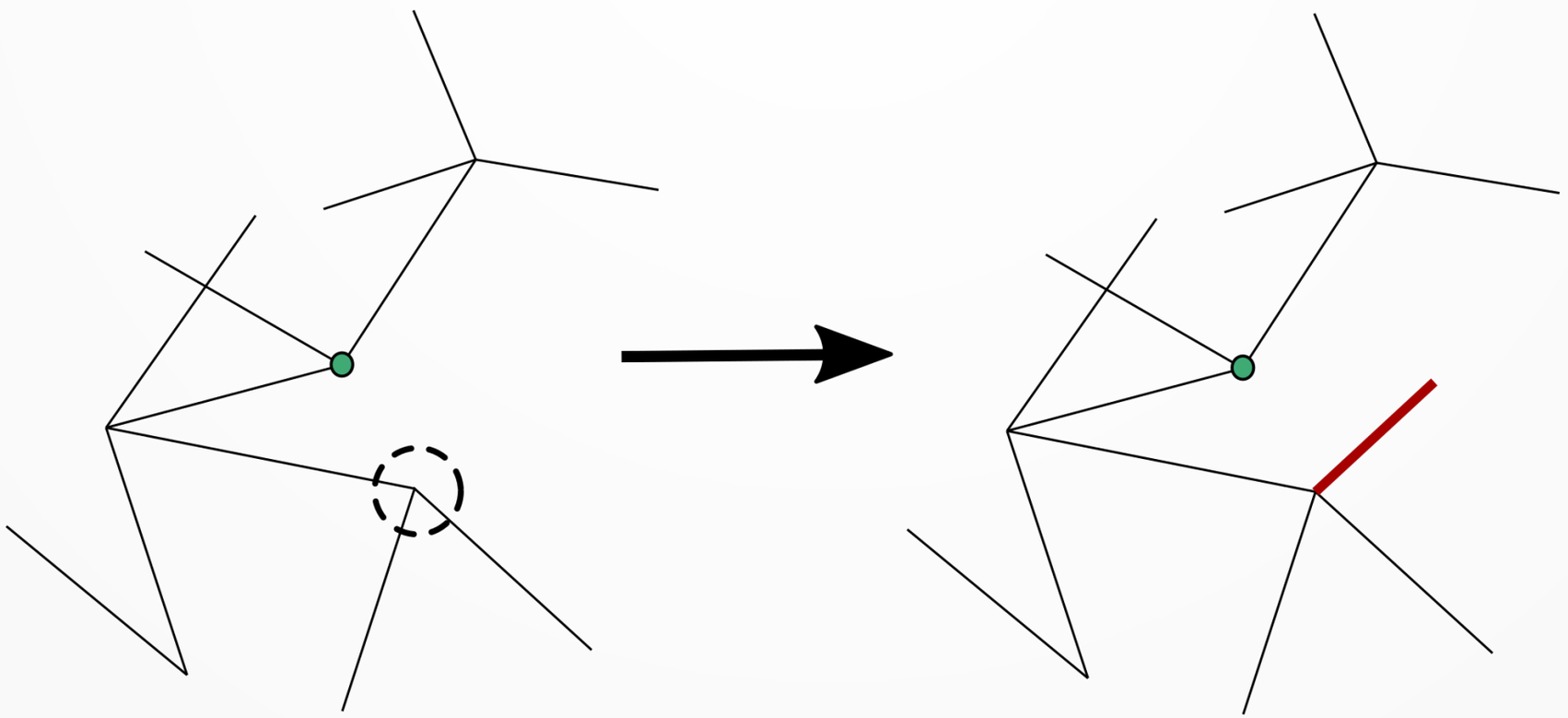
The structure of search algorithms based on trees is basically always the same, it consists in repeating the following operations:

- (1) **Selection:** select a node of the tree based on some criterion.
- (2) **Expansion:** try to “expand” from the selected node, i.e. construct a small collision-free motion from that node to a new node.

  Remark: to check that a transition (edge) from a configuration to another is collision-free, we usually test regularly-spaced configurations along the transition (which requires a hand-tuned parameter: the delta between consecutive configurations).

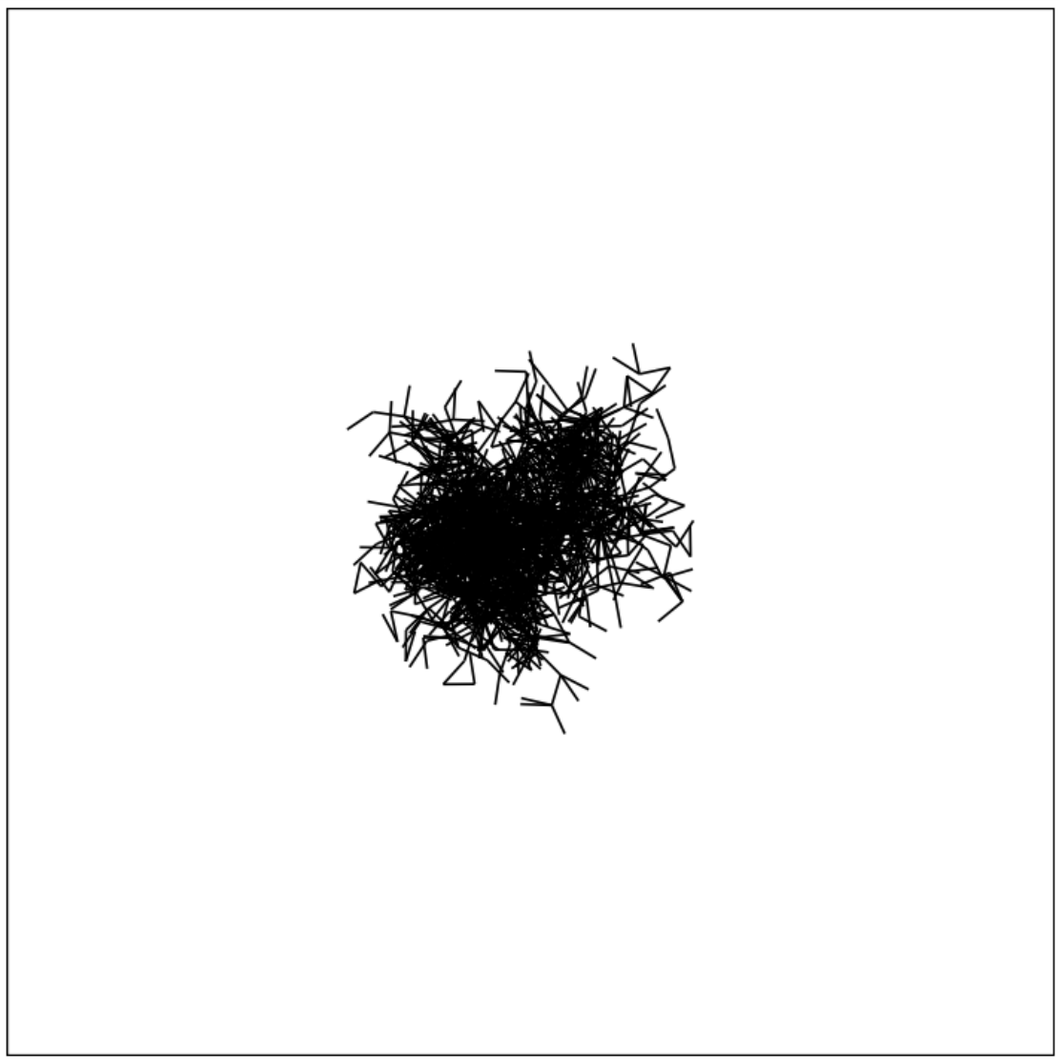
# Naive random tree

Select a node randomly, and then expand it randomly.

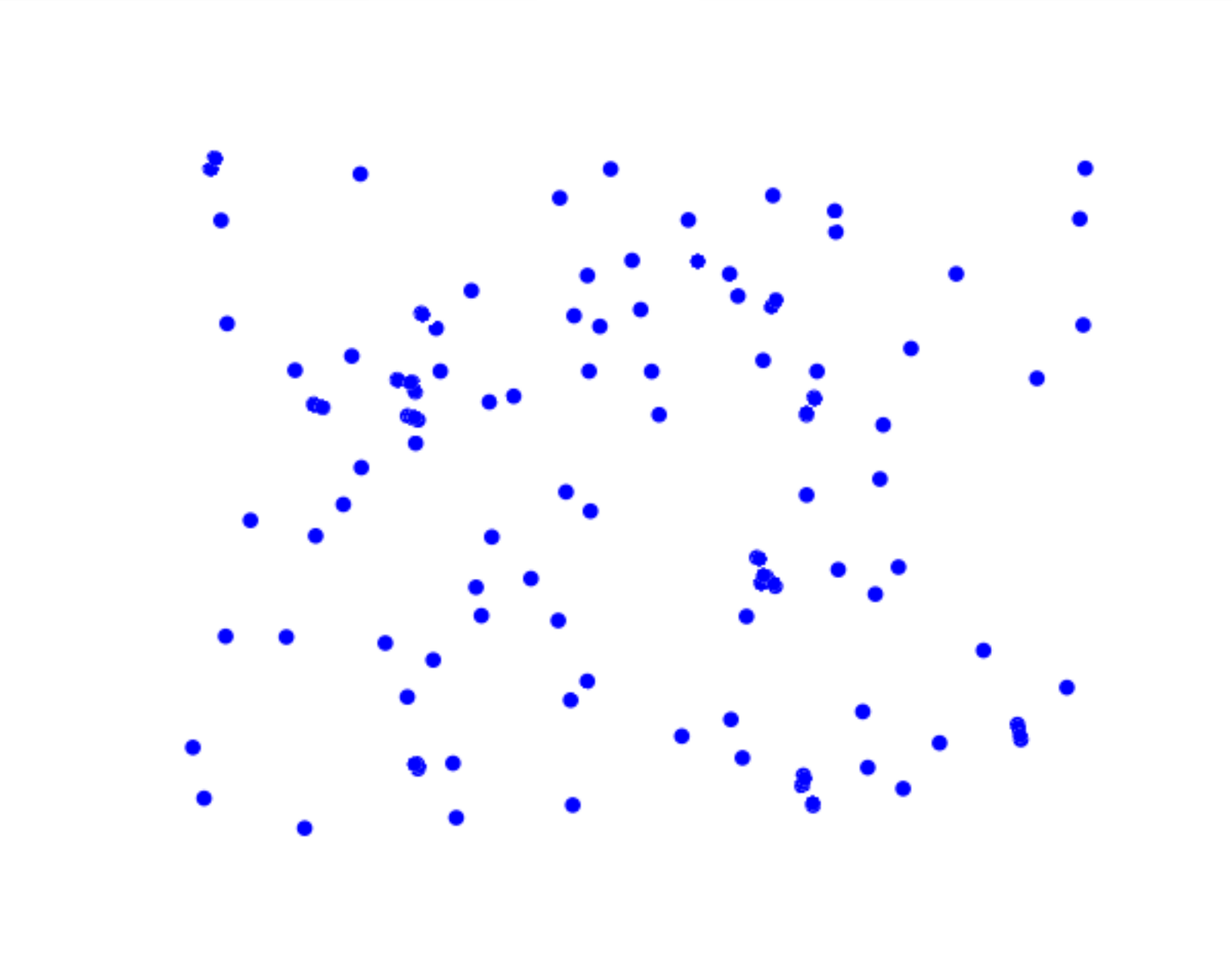


# Naive random tree

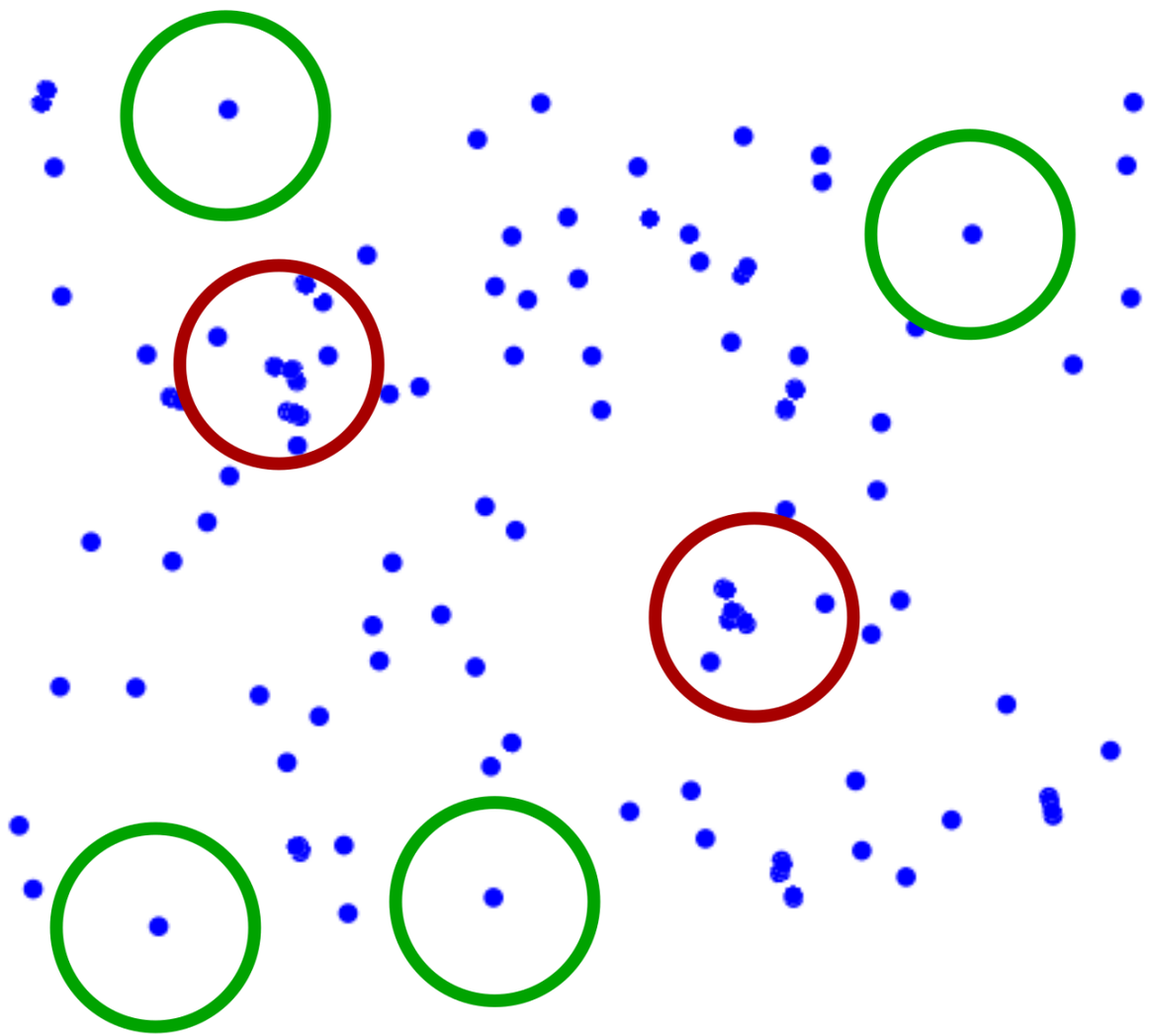
If we iterate, we end up with something like this (very dense):



# Selection based on density



# Selection based on density



# Expansive Spaces Trees (EST)

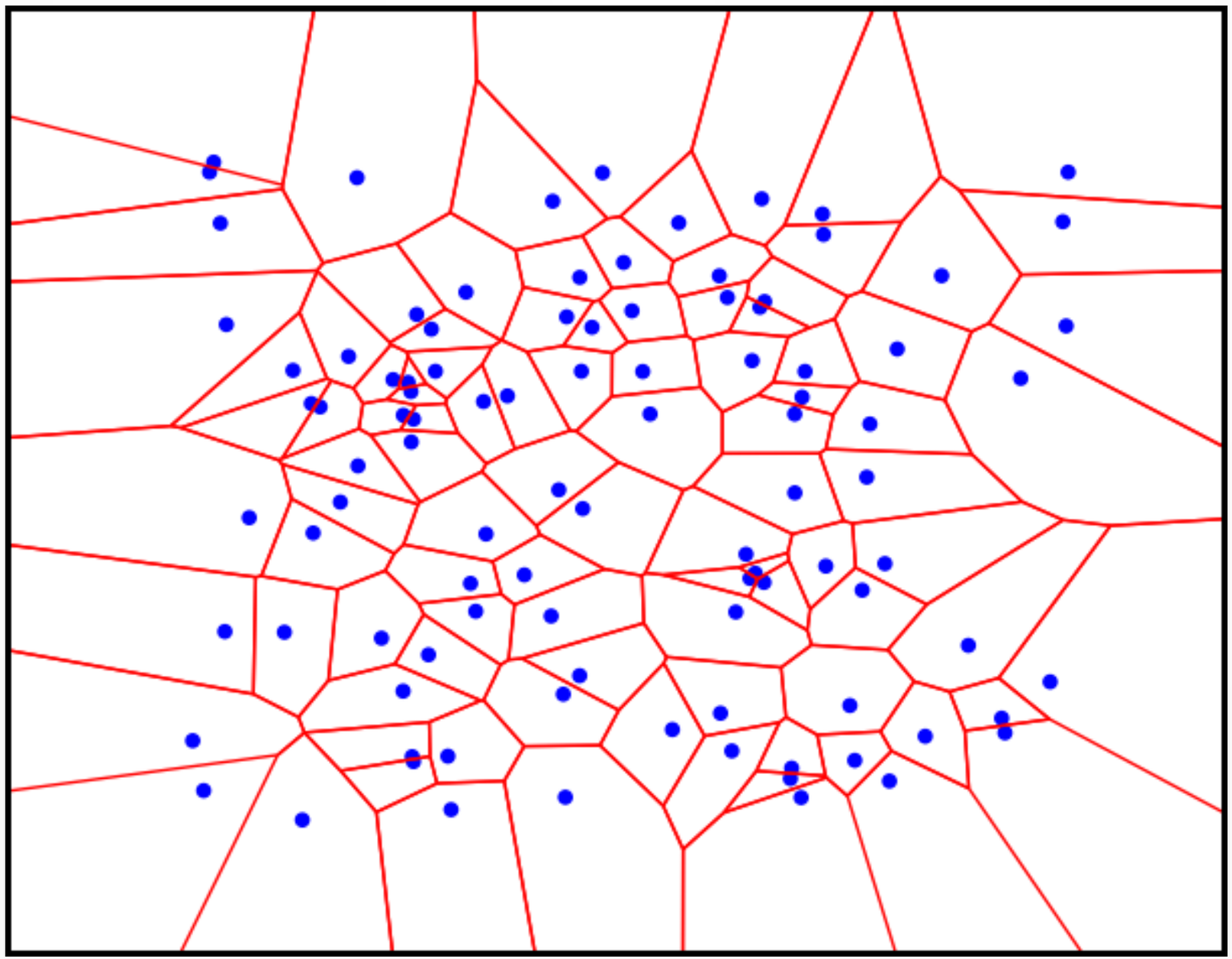
- When adding a new node, count its neighbors within an arbitrarily fixed distance to estimate the “local density” of nodes around it (= the number of nodes found divided by the volume). *Remark: update also the estimated density for the neighbors.*
- Nodes are selected with a probability that is inversely proportional to the estimated local density.

Ref.: **Path Planning in Expansive Configuration Spaces**, D. Hsu, J.-C. Latombe, R. Motwani, IEEE ICRA 1997.

# Limitations of EST

- Requires good **nearest neighbors algorithm**. *The standard way to do this is to use k-d trees. See* **Multidimensional binary search trees used for associative searching**, J. L. Bentley, Communications of the ACM 1975.
- Covering with constant density fills holes, it does not attempt to exand as fast as possible.
- Once nodes are selected, expansions are too random.

# Voronoi Diagrams

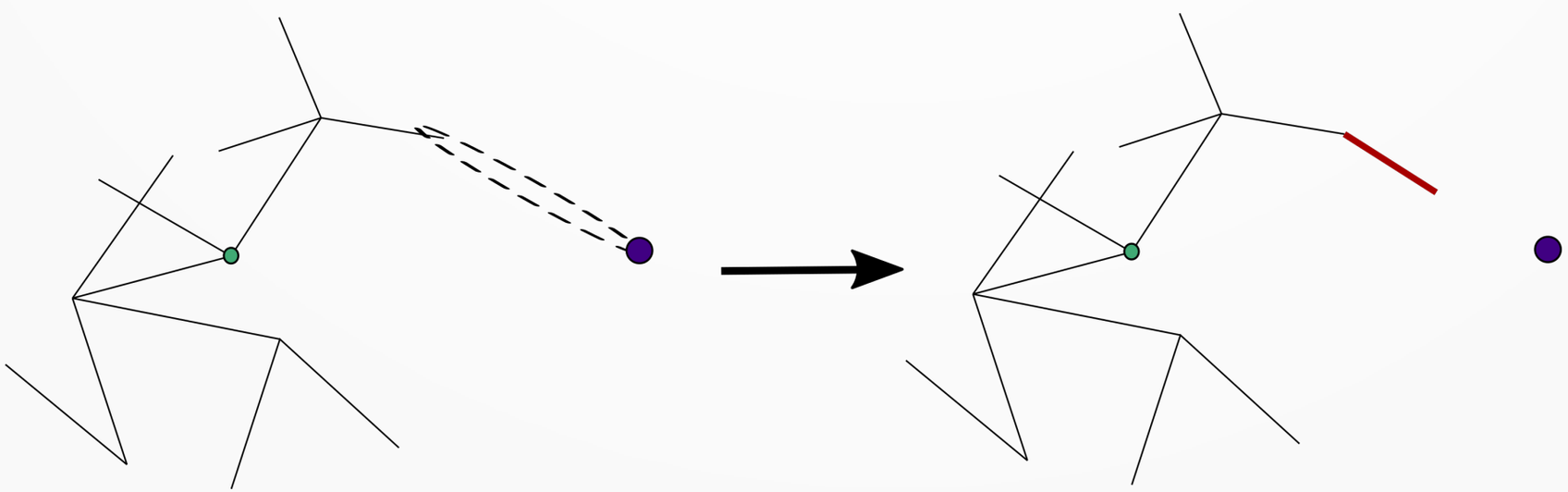


# Voronoi Diagrams

- We would like to expand the node with the largest Voronoi cell, however this is very difficult to compute.
- *But, what if we draw with uniform probability a random sample in the configuration space, and then select the nearest neighbor within the nodes ?* **Doing so, nodes have a probability to be selected that is proportional to the volume of their Voronoi cell.** It results in a sampling of nodes with a bias towards large Voronoi cells.

# Rapidly-exploring Random Trees (RRT)

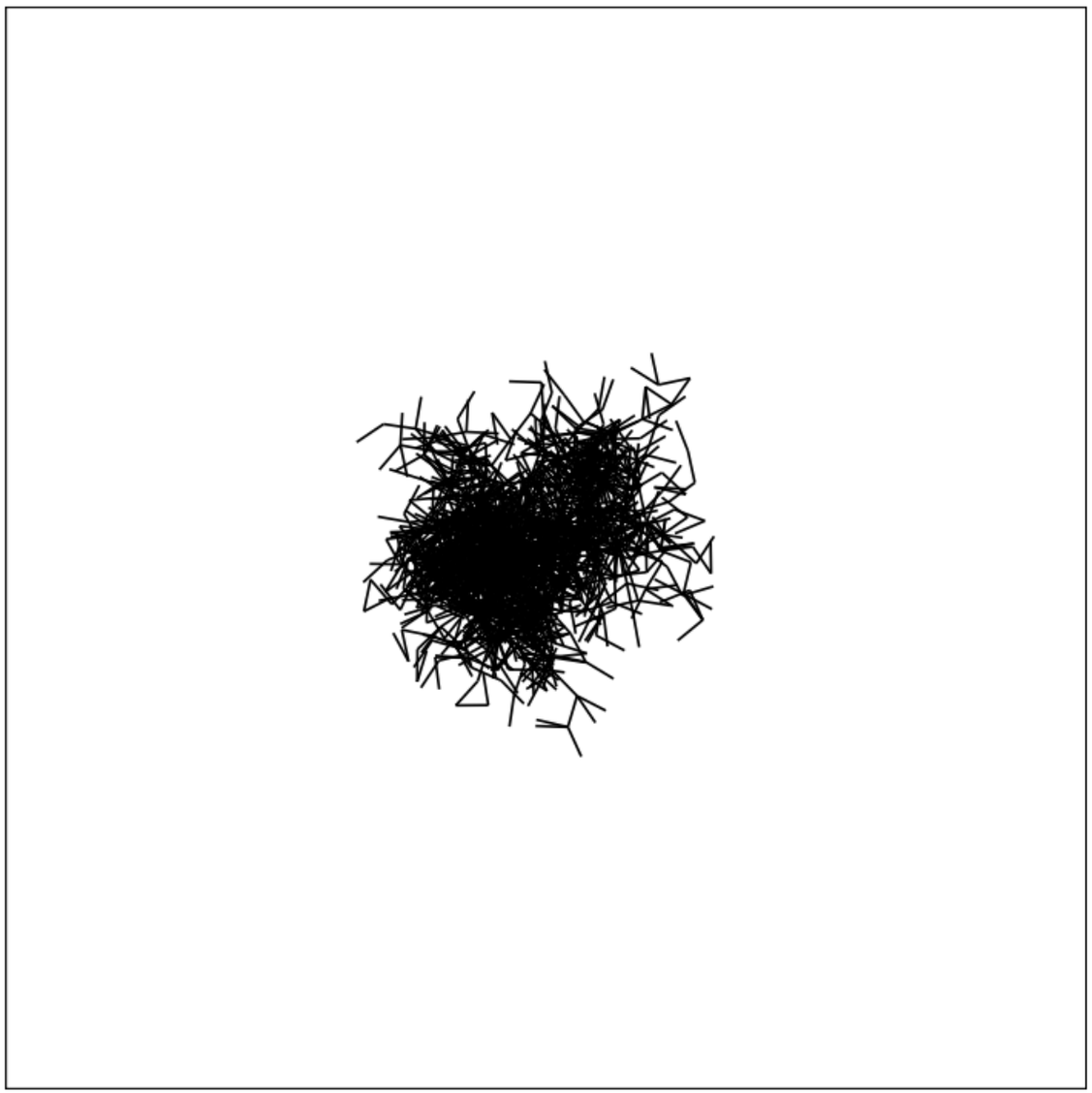
- Draw a random sample with uniform probablity in the configuration space, and select the nearest neighbor within the nodes of the tree.
- Expand the selected node **toward** the randomly sampled configuration.

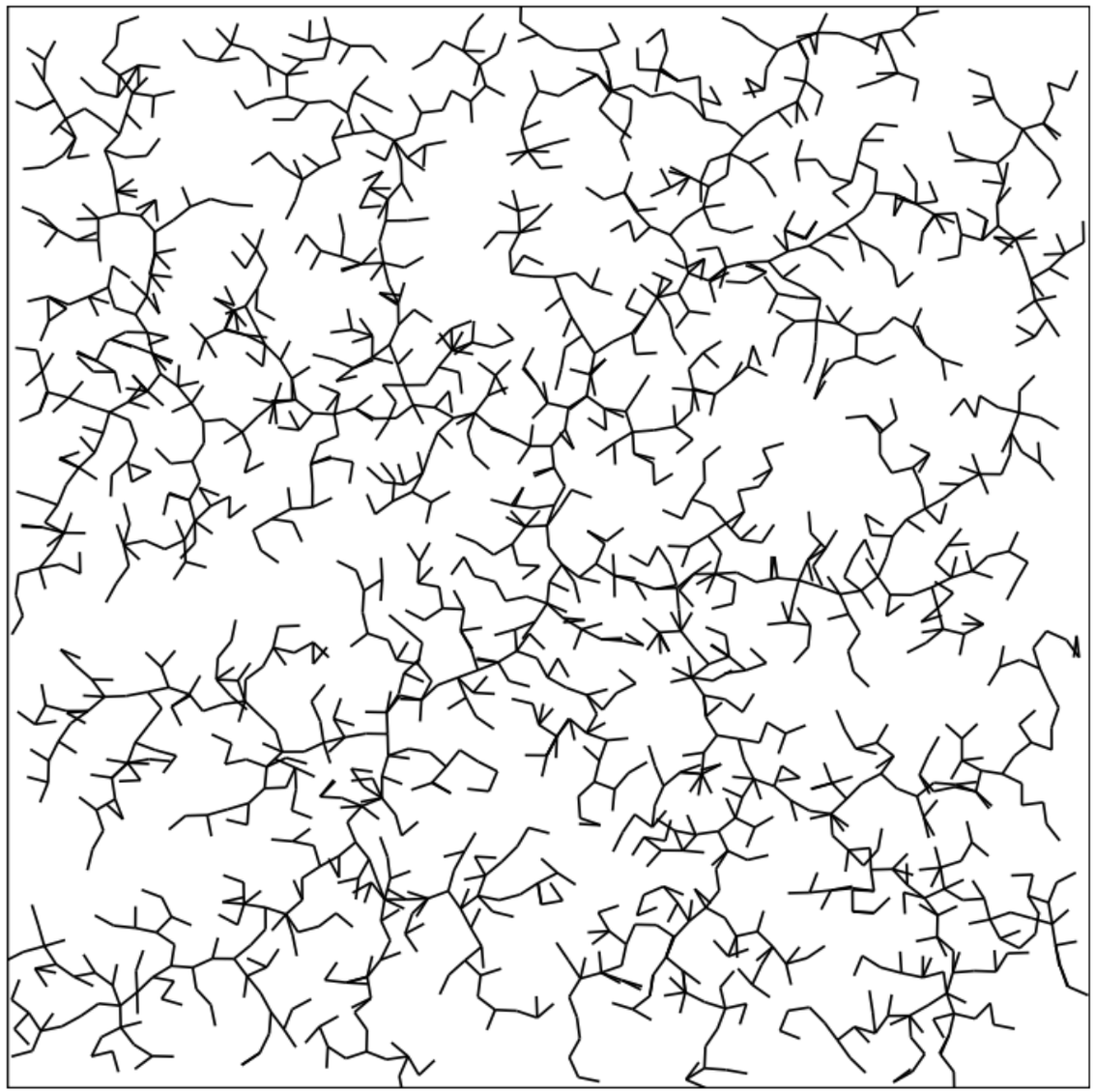


# Rapidly-exploring Random Trees (RRT)

- See **Randomized Kinodynamic Planning**, S.M. LaValle & J.J. Kuffner, International Journal of Robotics Research, 2001.

# Rapidly-exploring Random Trees (RRT)





# Limitations of RRT

- Also requires a lot of nearest neighbor computations.
- Need to know the bounds of the configuration space → In fact, there are simple solutions to that *(example: use bounds ~20% larger than the smallest bounding box of the current nodes)*.
- **What if sampling a random configuration is hard?**
- **What if expanding toward a sample is hard?**
- **It is asymptotically complete, but not asymptotically optimal.**

# Limitations of RRT

- If there exists a path from A to B (B a region of strictly positive volume), then RRT will find it with probability one (*if given enough time*).
- However, it will not find an *optimal path* (for example the straight line if is collision-free).

# RRT*

RRT* is a variant of RRT that includes some “rewiring” of the tree to achieve asymptotic optimality.

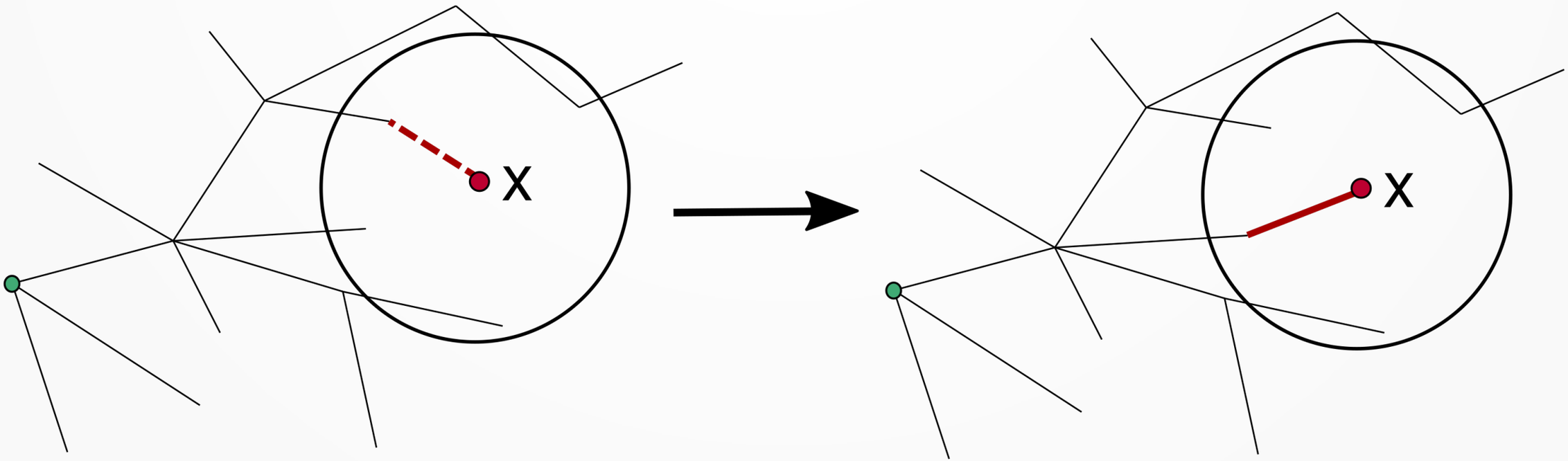
See: **Sampling-based algorithms for optimal motion planning**, S. Karaman & E. Frazzoli, International Journal of Robotics Research, 2011.

# RRT*: the algorithm

(1)

- Run an RRT selection-expansion, it adds a new node x to the tree. Consider the neighbors of x in the tree within a sphere of radius r (*remark: r depends on the current total number of vertices in the tree*)
- Modify the expansion by choosing a parent for x that minimizes the cost of the total path from the root to x.

# RRT*: the algorithm

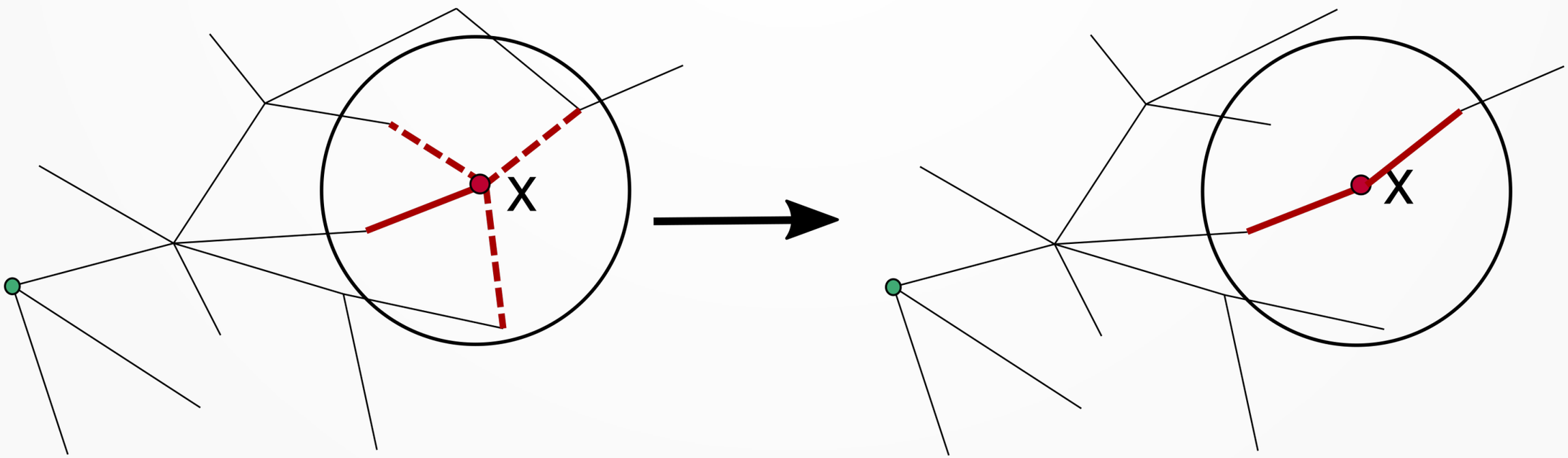


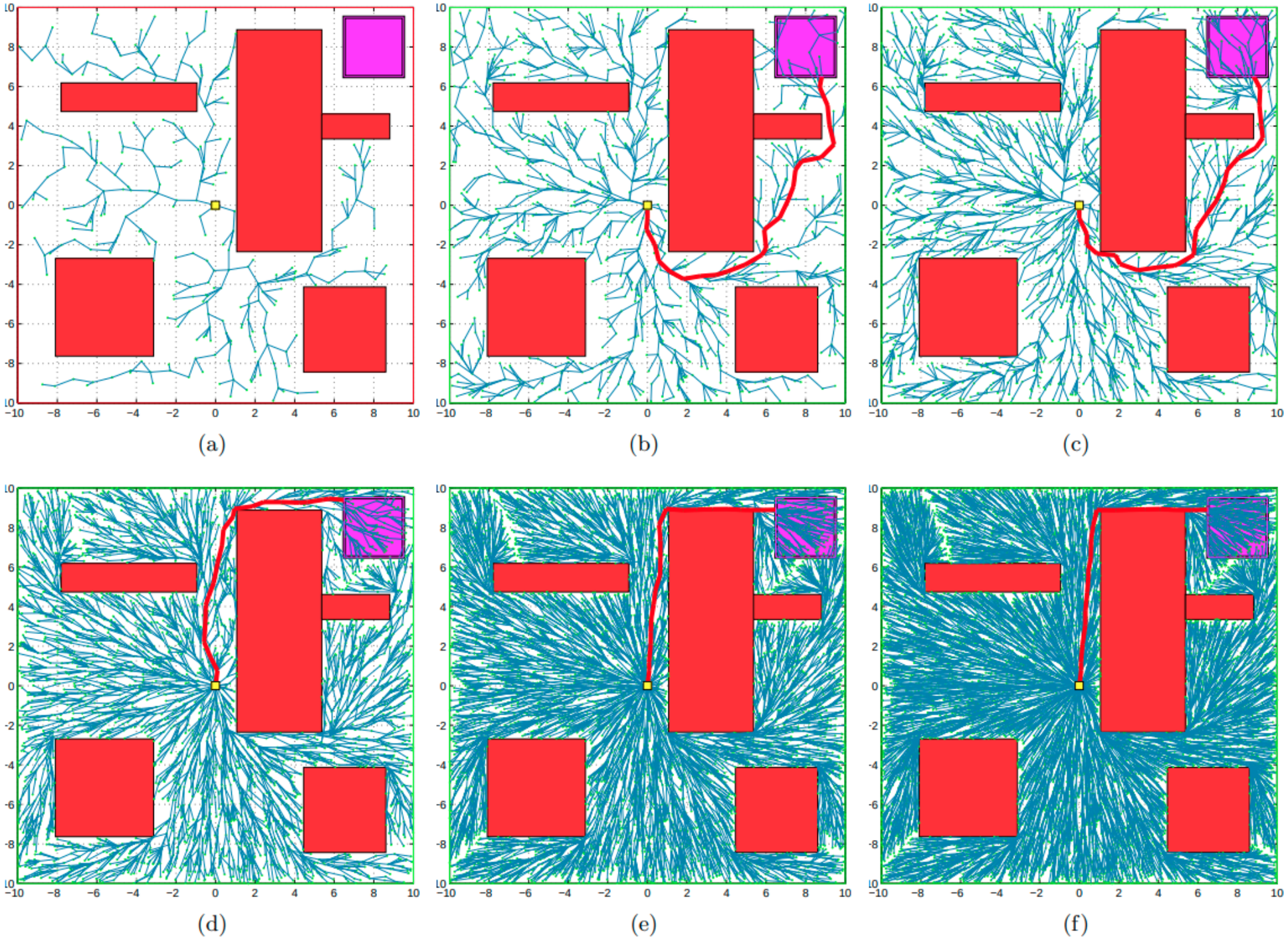
# RRT*: the algorithm

(2)

- Consider edges from x to all the “neighbors” y in the sphere, and add an edge to a neighbor y if it creates a path to y that is shorter than the currently existing path. If an edge to y is added, the edge to y from its previous parent is removed.

# RRT*: the algorithm




(a)
(b)
(c)
(d)
(e)
(f)

# Non-holonomic and kinodynamic motion planning

In motion planning problems with non-holonomic or kinodynamic constraints, not all geometric trajectories are feasible. In such cases, the planning must be done in the state space, not only in the configuration space (remark: typically, the state includes velocities).

*In general, via discretization, the reachable states* **x'** *from a state* **x** *verify :*

**x' = x + f(x, u)**, for some possibly constrained control input u.

# Non-holonomic and kinodynamic motion planning

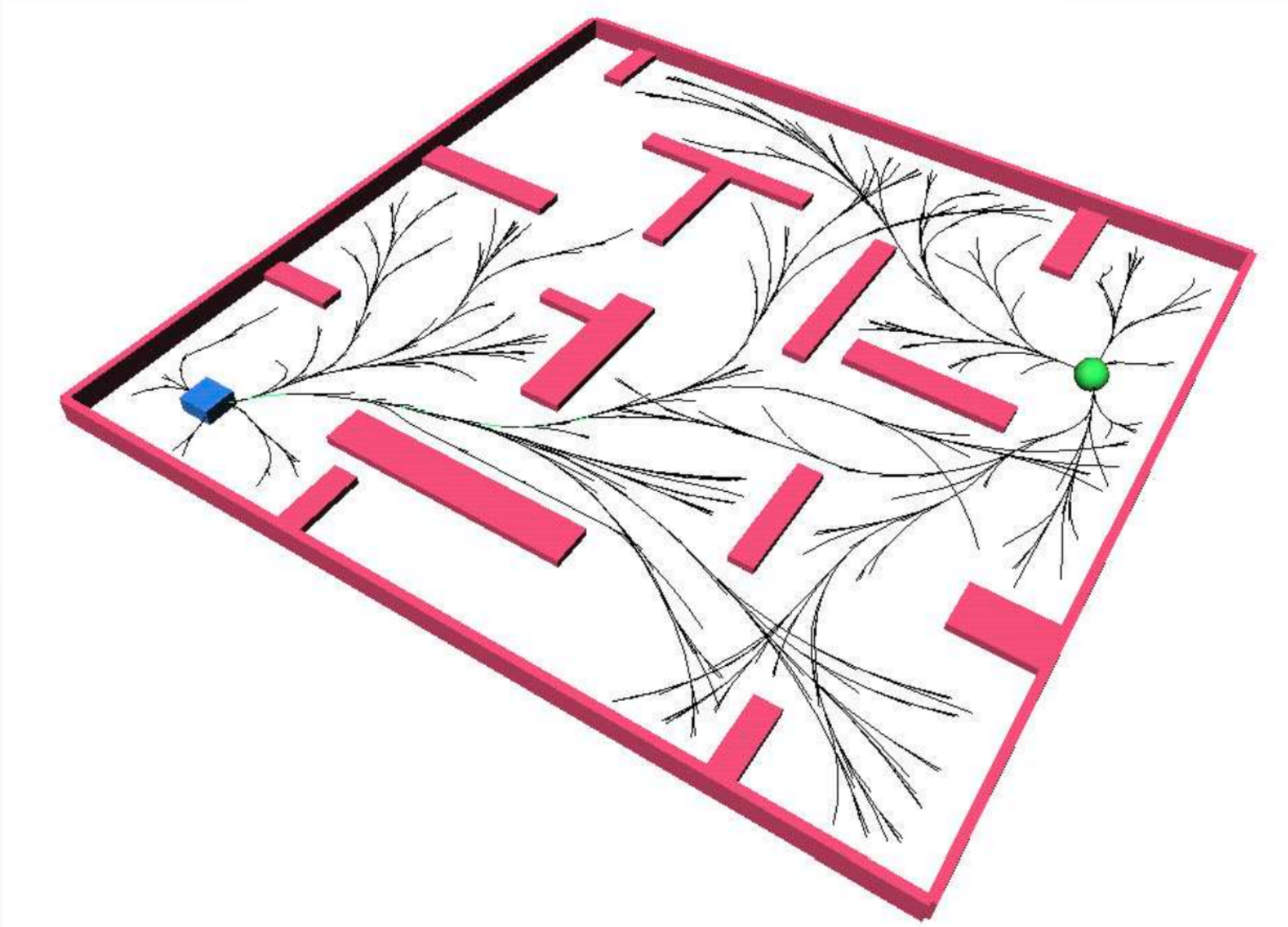
**x’ = x + f(x, u)**

Without knowing anything specific about f, expanding toward a sampled state is not possible.

But RRT can be adapted, *either:*

- *by choosing* **u** *randomly (random expansion), or:*
- *by trying many possible inputs and choosing one that yields a new state as close as possible to the target sample.*

# Kinodynamic RRT works well too:

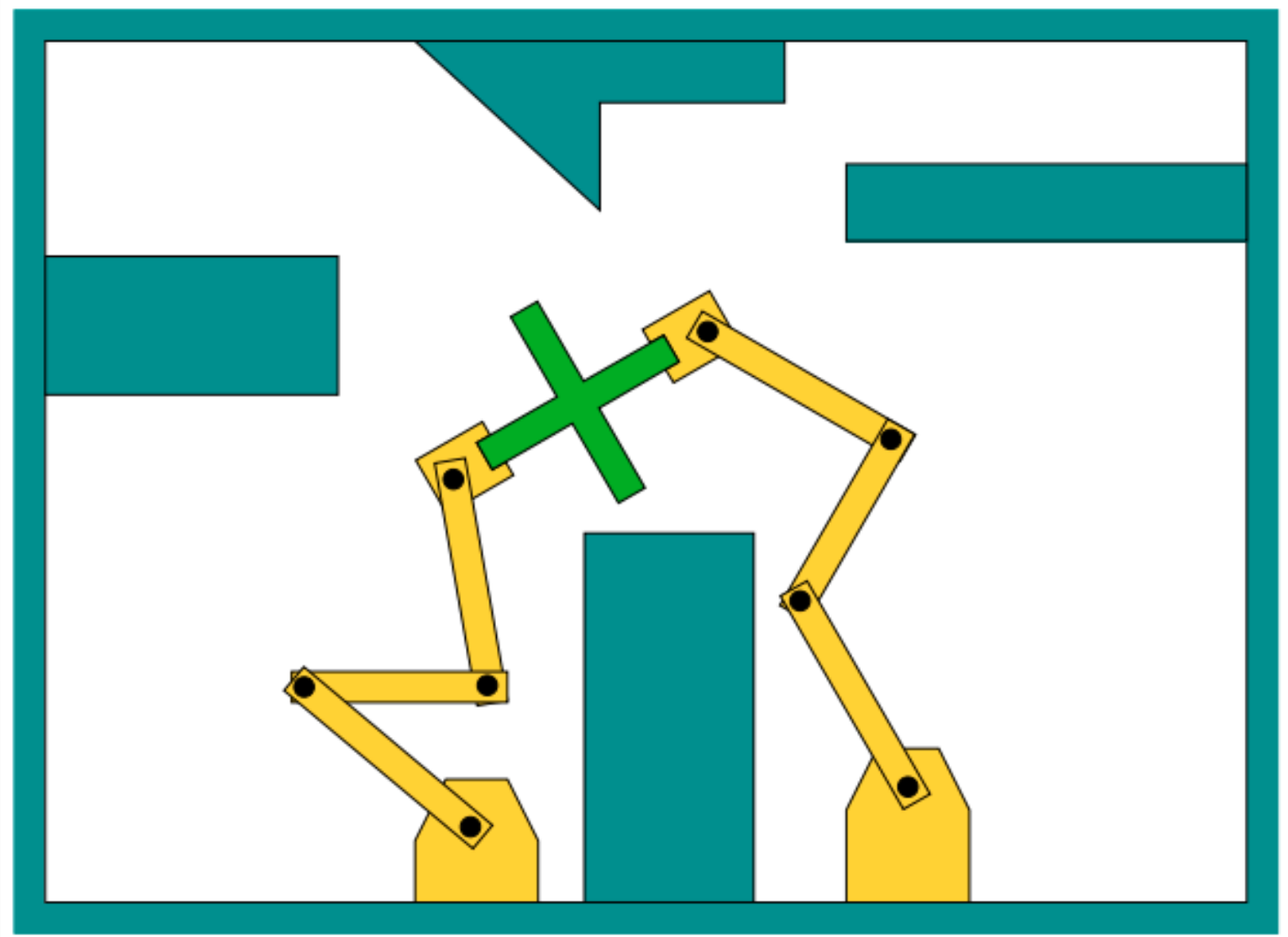


# Kinodynamic RRT

*Remark:* techniques like RRT* cannot be applied to kinodynamic motion planning, therefore asymptotic optimality remains a big challenge for problems with dynamic constraints.

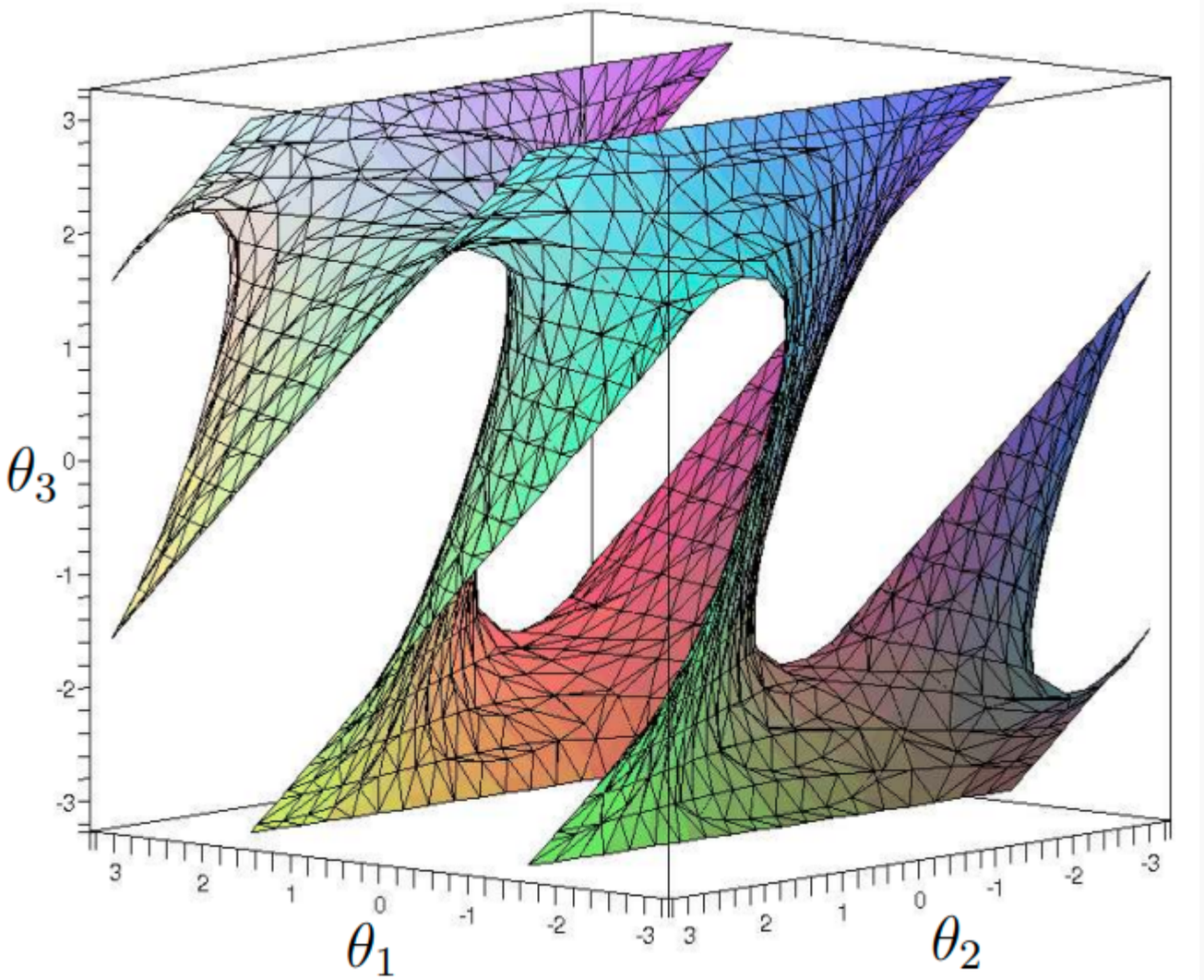
A way to address this issue is to use motion planning to find *initial trajectory candidates*, and then use another technique to incrementally improve the solution (e.g. *reinforcement learning, optimal control, ...*)

# When it is hard to sample configurations: example with closed kinematic chains



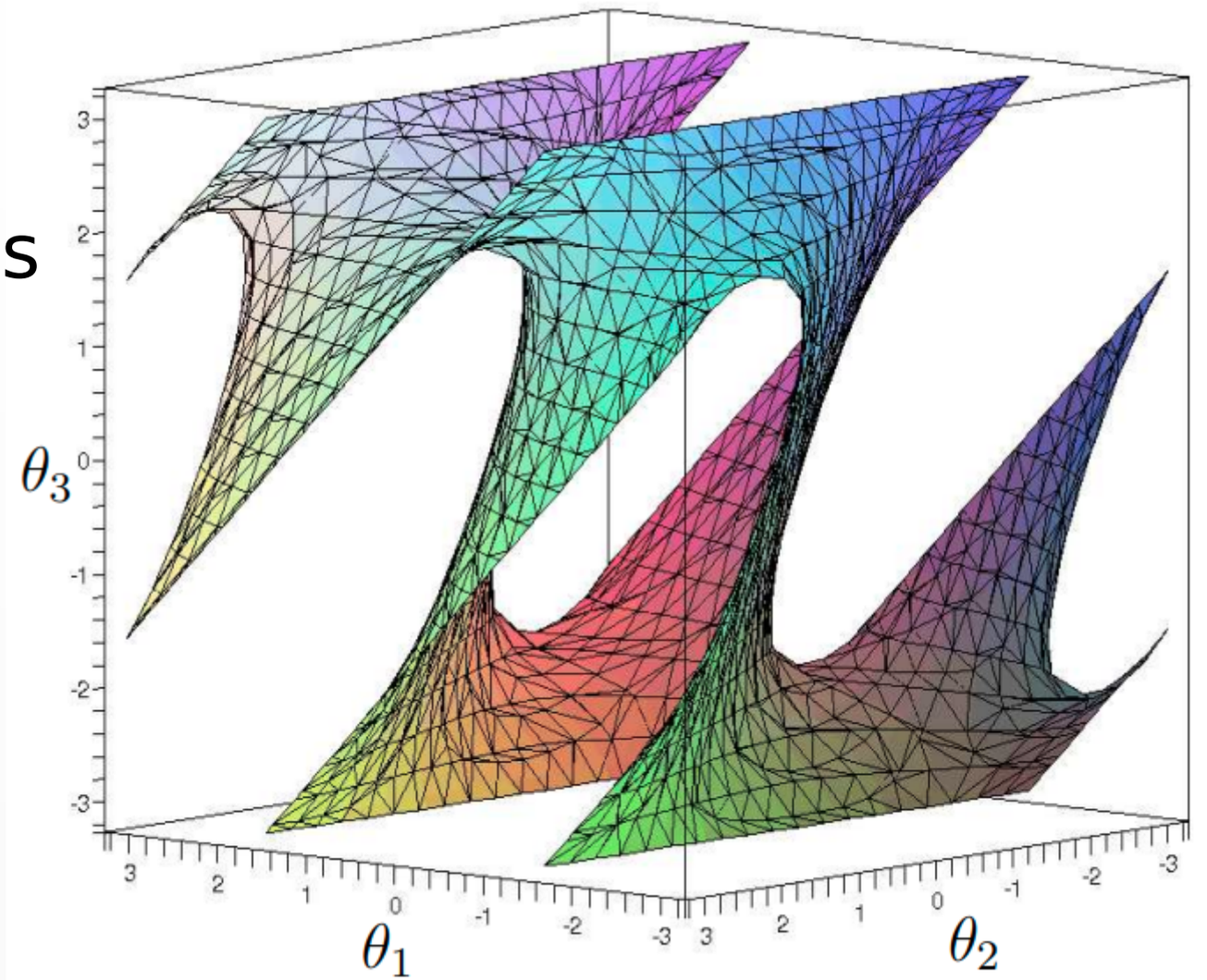
# Closed kinematic chains

Example of configuration space (a submanifold of the joint space) for a 3-link kinematic chain.



# Closed kinematic chains

The motion planning must explore trajectories that remain on the submanifold, but it cannot be ensured using control inputs.



# Closed kinematic chains

There ways to adapt sampling-based motion planning algorithms, usually by solving loop closure equations to project all considered motions on the correct submanifold.

See: **Motion planning algorithms for general closed-chain mechanisms**, J. Cortes, 2005.

# Multiple-query motion planning

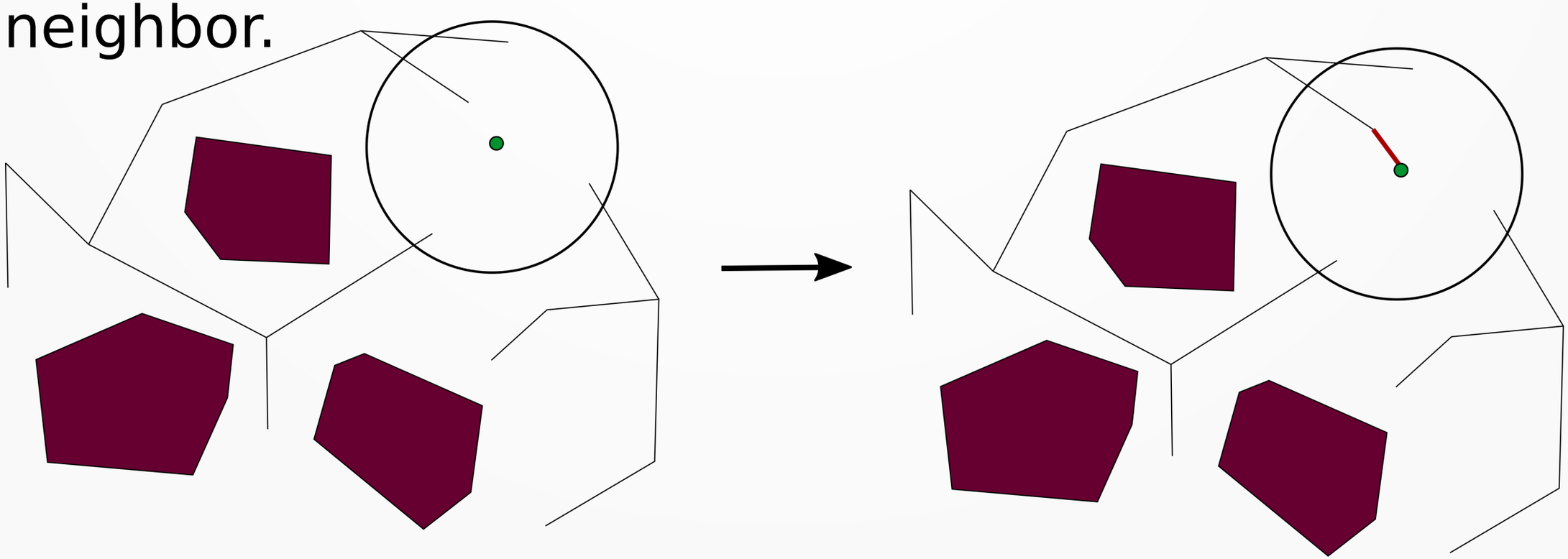
Goal: building a *roadmap*, i.e. a graph of motions that can be useful for quickly answering motion planning queries.

# Probabilistic Roadmaps (PRM)

See: **Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces**, L.E. Kavraki, P. Svestka, J.-C. Latombe, M. H. Overmars, IEEE Transactions on Robotics and Automation, 1996.

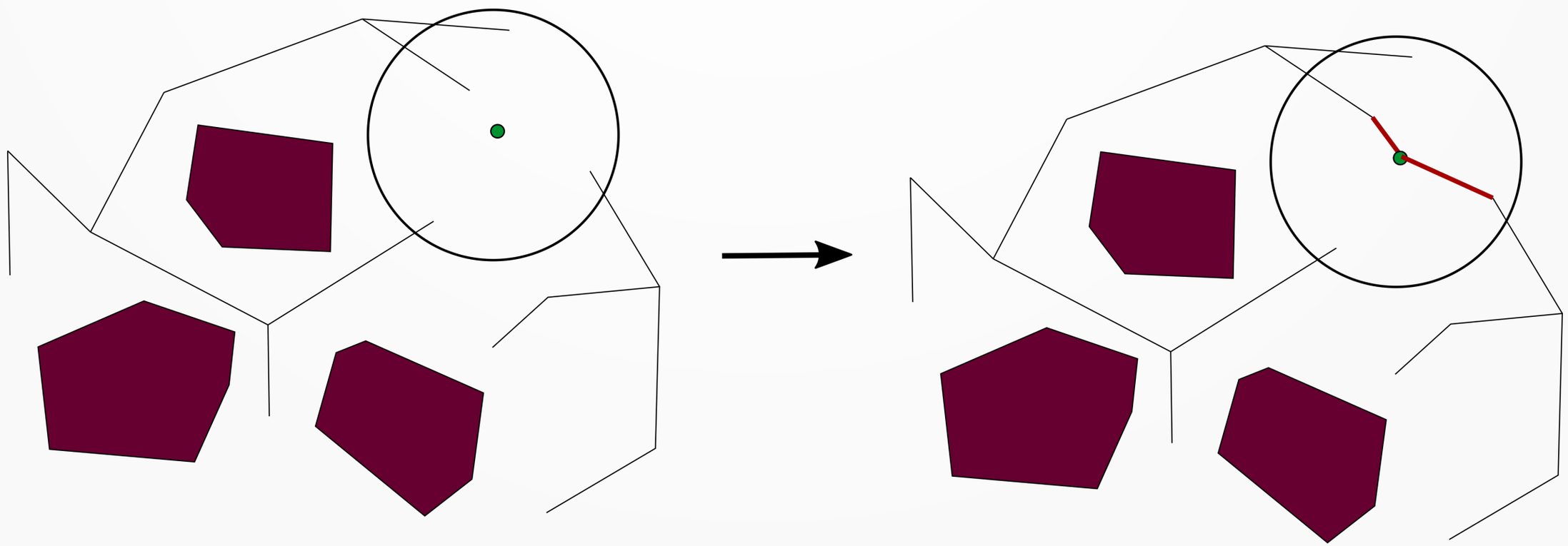
# Probabilistic Roadmaps (PRM)

(1) Randomly sample a configuration, and consider neighbors in the current graph within a sphere of arbitrarily fixed radius. If possible, create a collision-free edge between the new sample and its nearest neighbor.

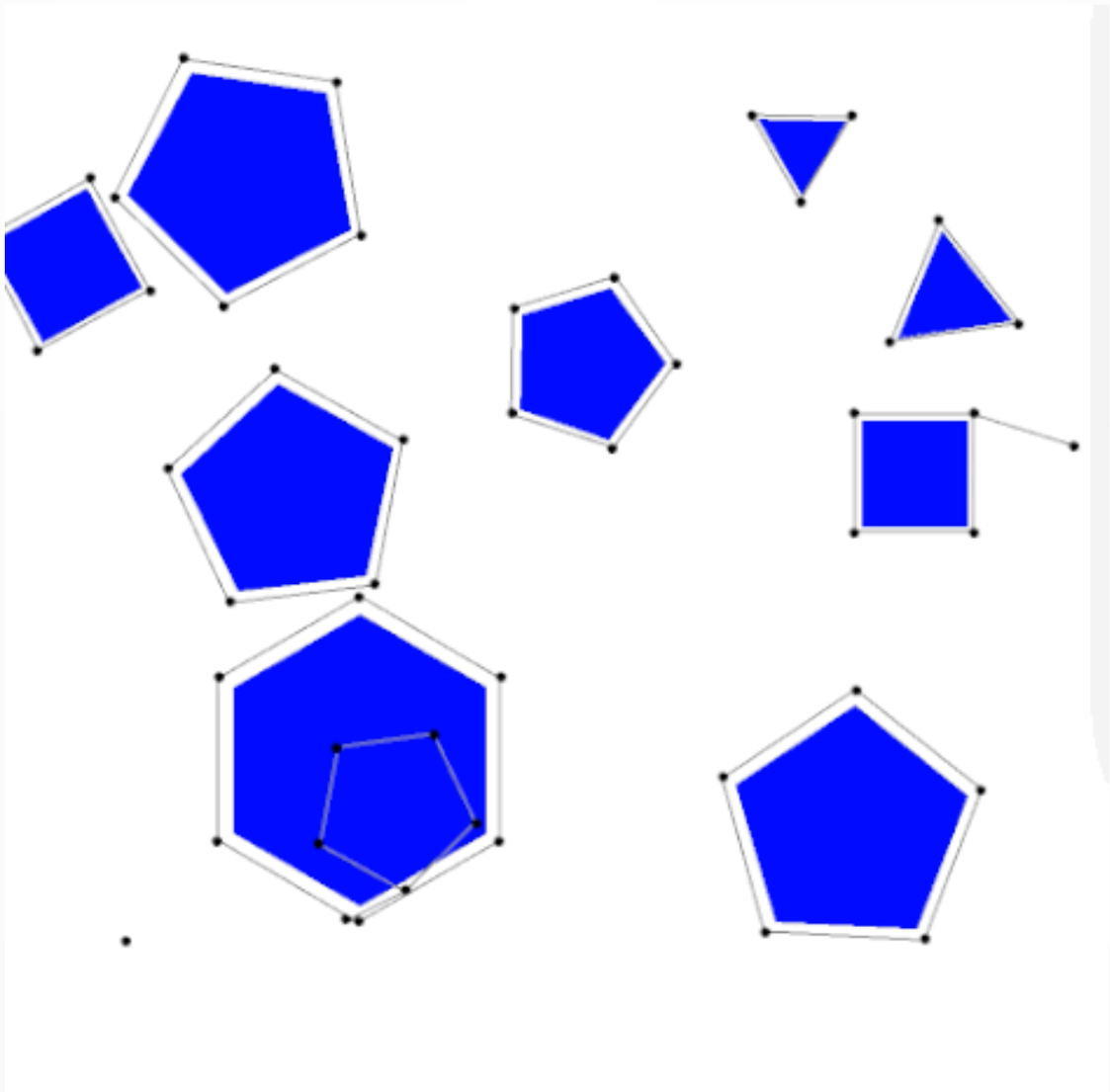


# Probabilistic Roadmaps (PRM)

(2) Consider the other neighbors by order of increasing distance, and add an edge (if collision-free) if and only if it joins two distinct connected components of the graph (roadmap).



# Probabilistic Roadmaps (PRM)



# Probabilistic Roadmaps (PRM)

The query phase relies on a *local planner* to connect the source configuration **S** and goal configuration **G** to the roadmap, and then applies Dijkstra's algorithm to find the shortest path from **S** to **G** via the roadmap.