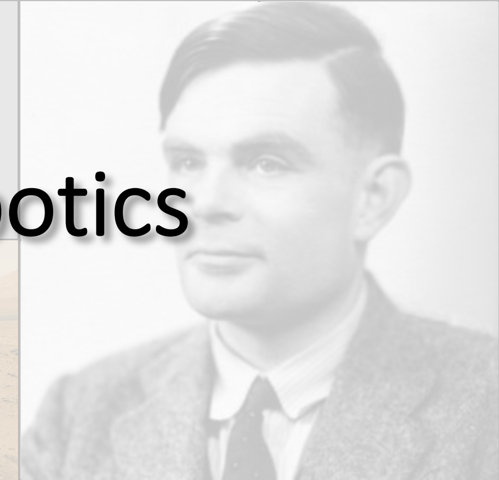
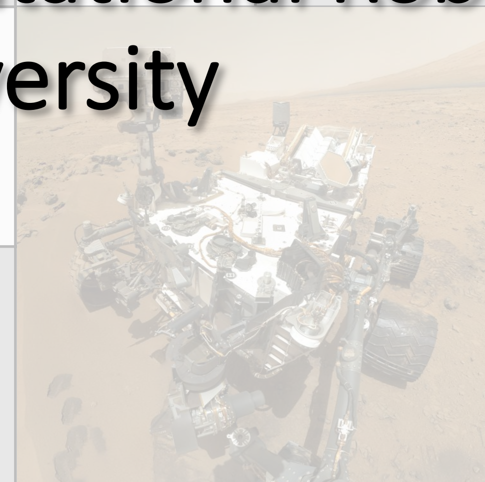
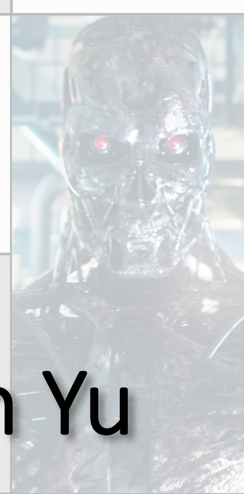
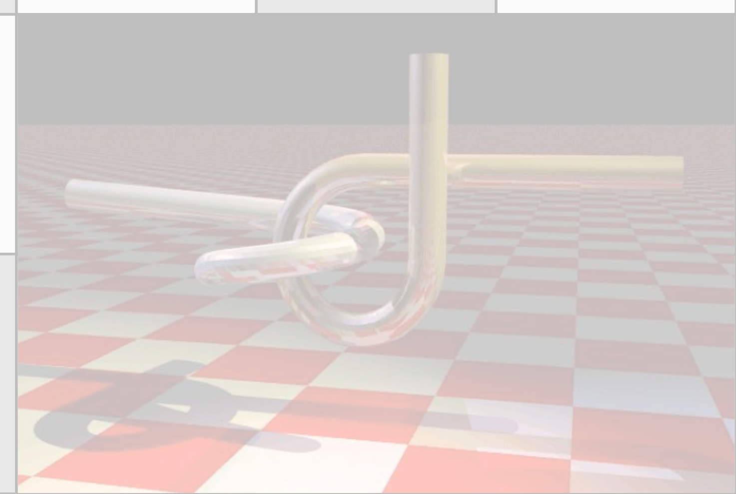
CS 460/560
Introduction to Computational Robotics
Fall 2019, Rutgers University

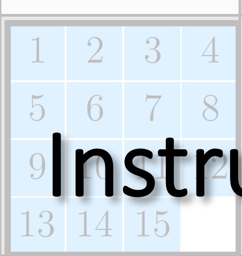




Final Review





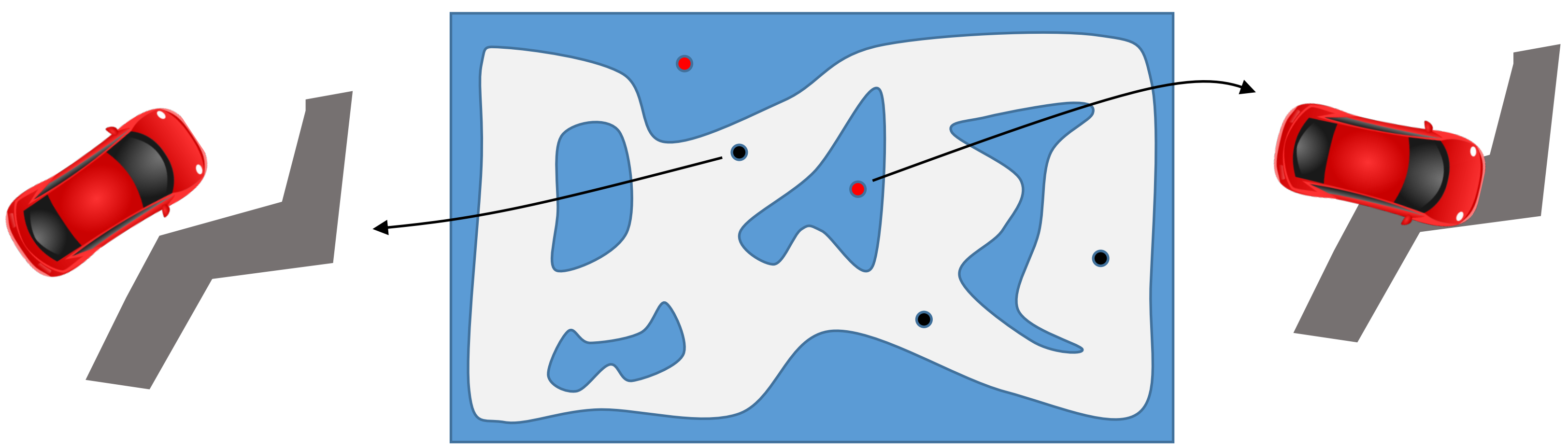


Instructor: Jingjin Yu

Key Components of Sampling-Based Planning

Sampling-based planning requires several important subroutines

- ⇨An **efficient sampling routine** is needed to generate the samples. These samples should **cover** C_{free} well in order to be effective
- ⇨**Efficient nearest neighbor search** is necessary for quickly building the roadmap: for each sample in C_{free} we must find its k-nearest neighbors
- ⇨The neighbor search also requires a **distance metric** to be properly defined so we know the distance between two samples
  - ⇨ This can be tricky for certain spaces, e.g., $SE(3)$
- ⇨**Collision checking** - Note that C_{free} is not computed explicitly so we actually are checking collisions between a complex robot and a complex environment



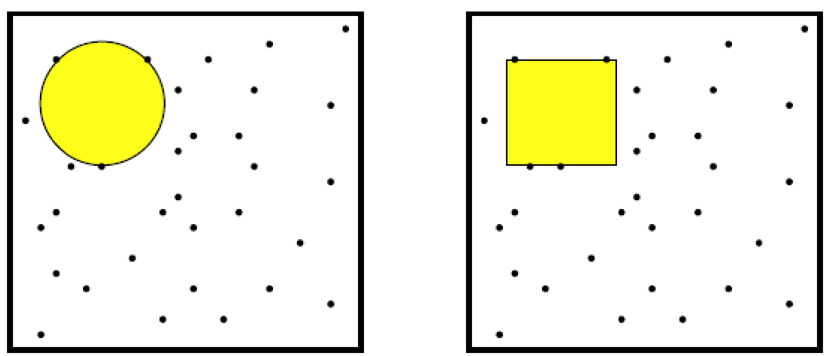
“Goodness” of Samples

The sampling process aims at “covering” C_{free}. How to measure the “goodness” of a set of samples?

Dispersion: the dispersion of a finite set P of samples in a metric space (X, ρ) is

$$\delta(P) = \sup_{x \in X}\{\min_{p \in P}\{\rho(x, p)\}\}$$

Roughly, this means the largest ball that can be fit in the samples without including any sample inside the ball



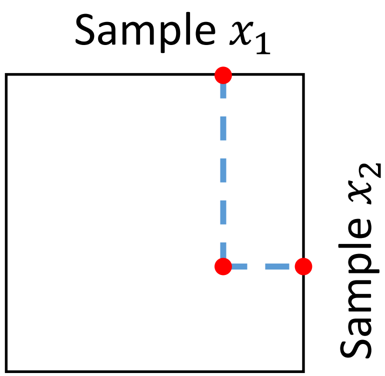
(a) L_2 dispersion (b) L_∞ dispersion

Generally speaking, given $|P|$ samples, a sample set with smaller dispersion $\delta(P)$ is better.

Image source: Planning Algorithms by LaValle

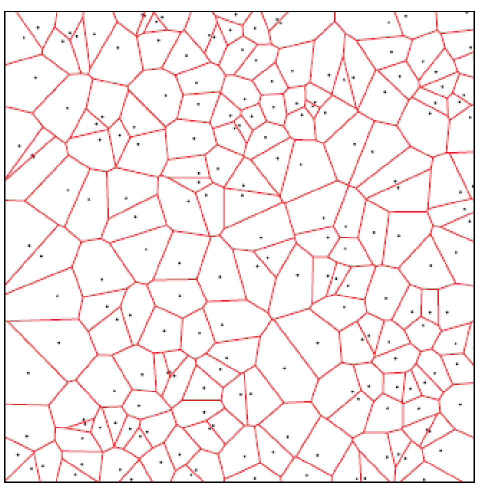
Sampling Routine

The simplest way of achieving this: **uniformly random sampling**

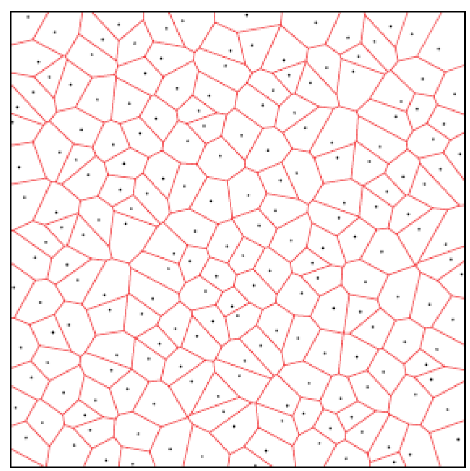


A sample $(x_1, x_2) \in \mathbb{R}^2$

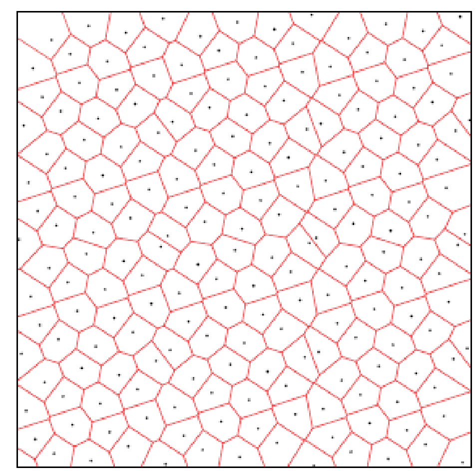
Generally, **incremental, dense** sampling w/ good **dispersion**



(a) 196 pseudorandom samples



(a) 196 Halton points



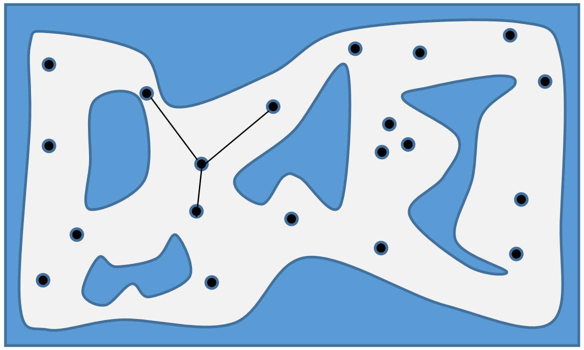
(b) 196 Hammersley points

Nearest Neighbor Search w/ k-d Tree

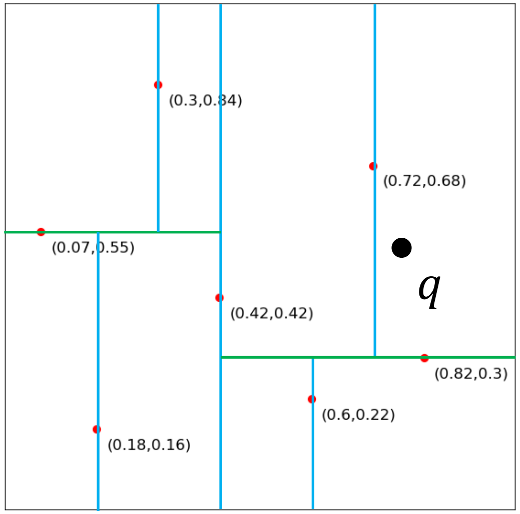
Connecting the samples

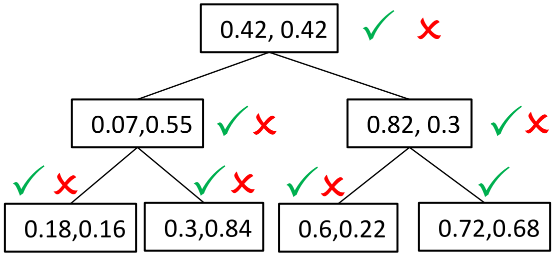
⇨Building the graph requires connecting the samples

⇨Need efficient methods for this



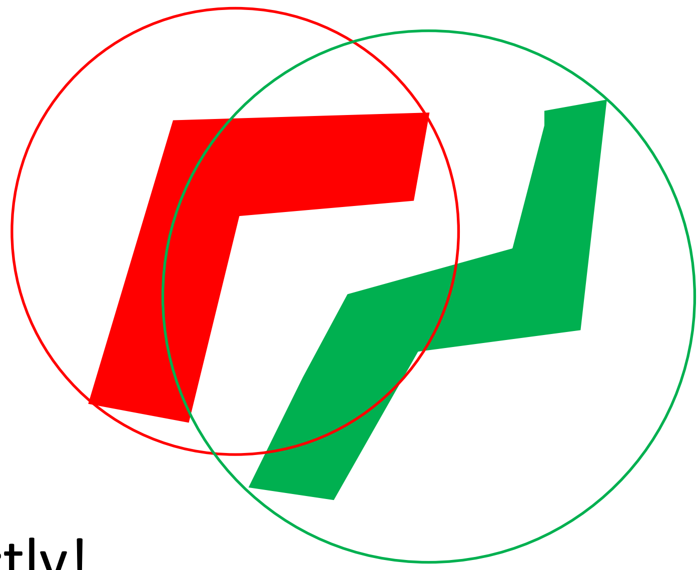
k-d Tree





Bounded Volume Hierarchy (BVH)

Collision checking can be difficult for general objects, e.g.,

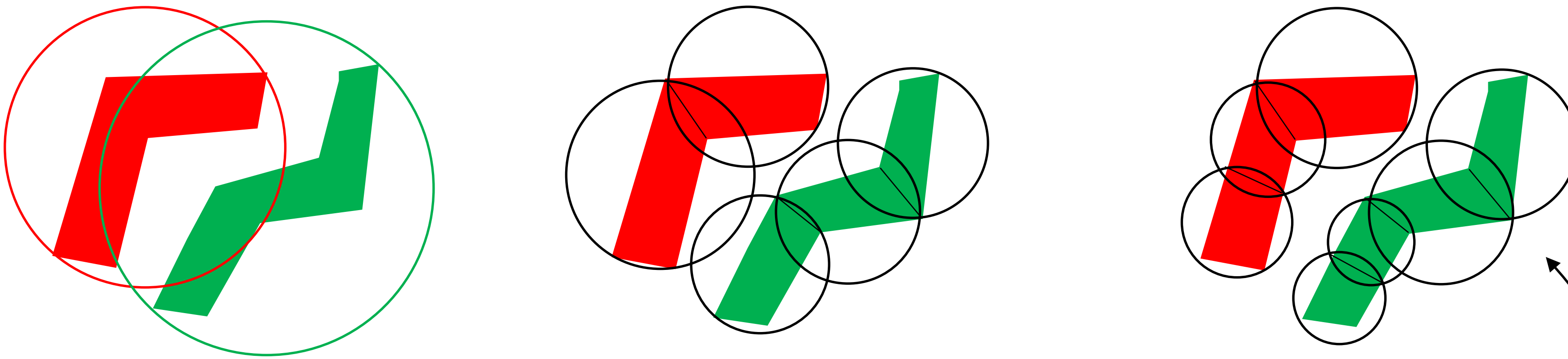


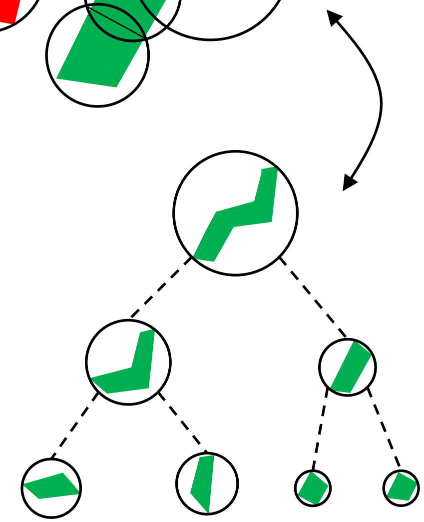
$d(A, B)$ are hard to compute directly!

Often, simpler **bounding volumes** are used to approximate the shapes

- ⇨ However, bounding volumes **over approximate** the shapes
- ⇨ No collision between bounding volumes → no collision between the shapes
- ⇨ Collision between bounding volumes → **possible** collision
- ⇨ Need to refine hierarchically if a possible collision is detected
- ⇨ Such a method is called **bounded volume hierarchy** (BVH)

Bounded Volume Hierarchy (BVH), Continued

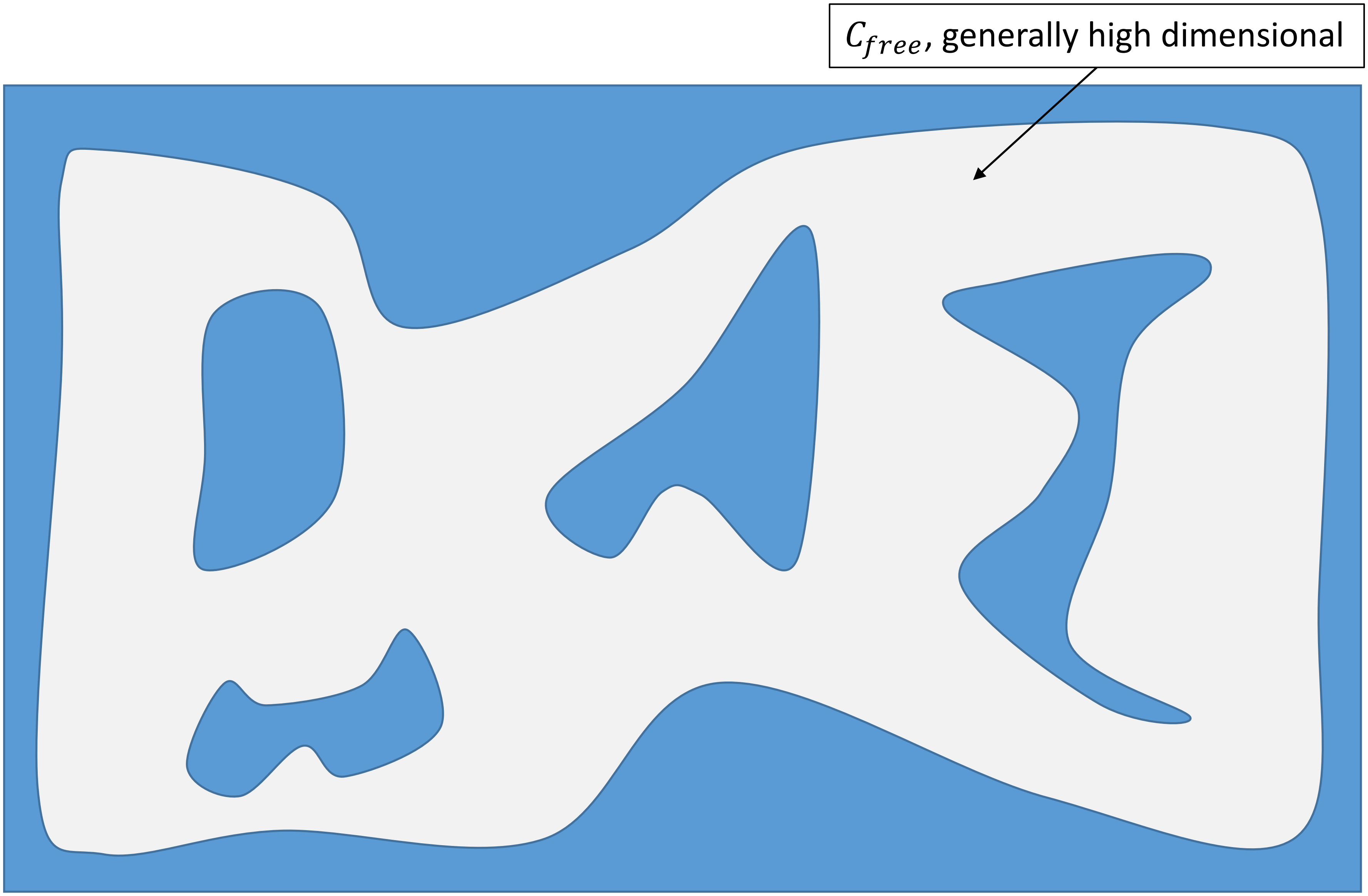




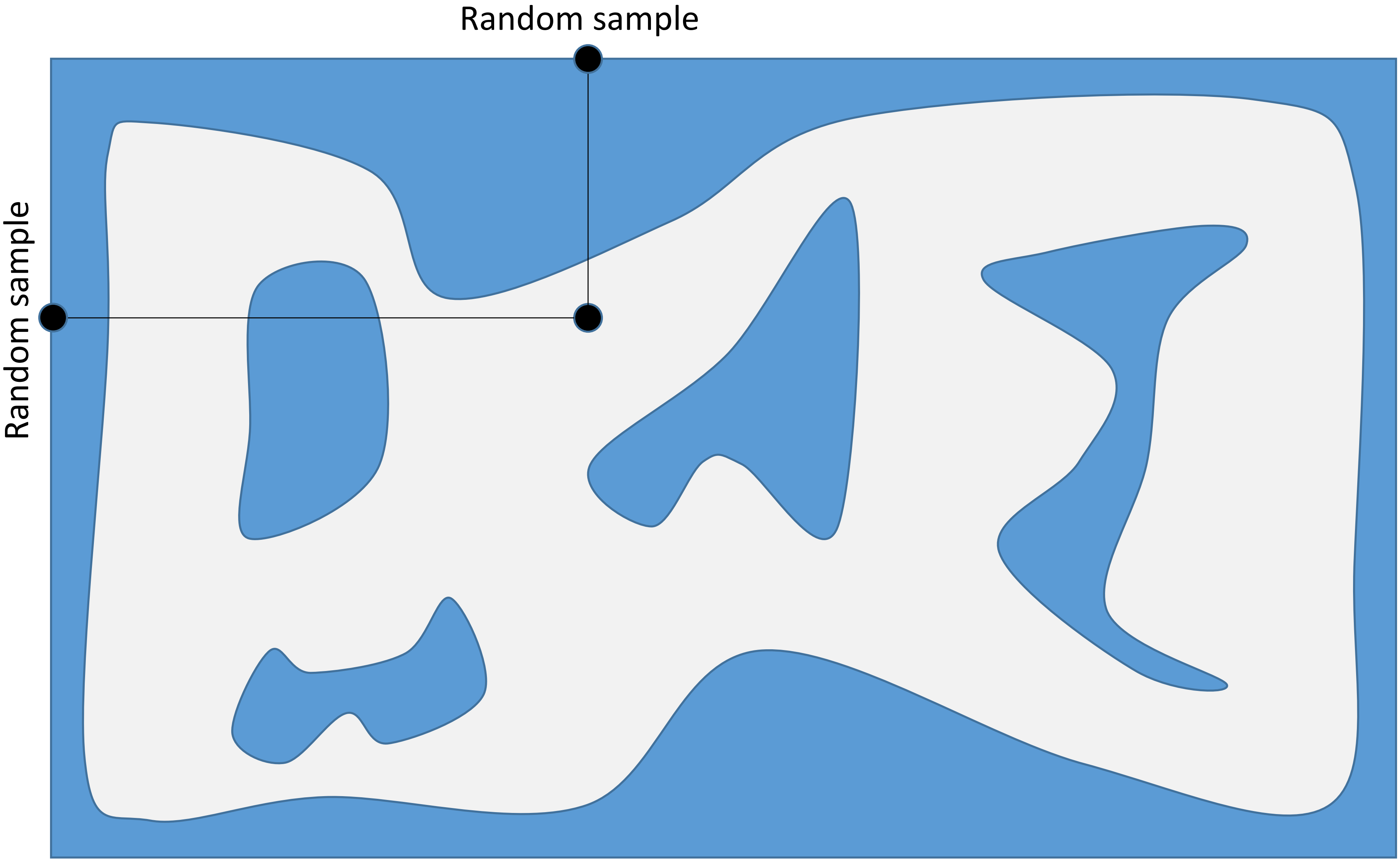
For collision checking, it works with two BVH trees

- ⇨Starting from the roots and check for collision (how?)
  - ⇨No collision → done with the branch
  - ⇨Otherwise, check pairs of children on the trees
- ⇨Recursively call the procedure
- ⇨Traverse down the tree
- ⇨How many possible checks in total (say each object has n pieces)?
  - ⇨At most n^2 checks
  - ⇨Using BVH can save some checks

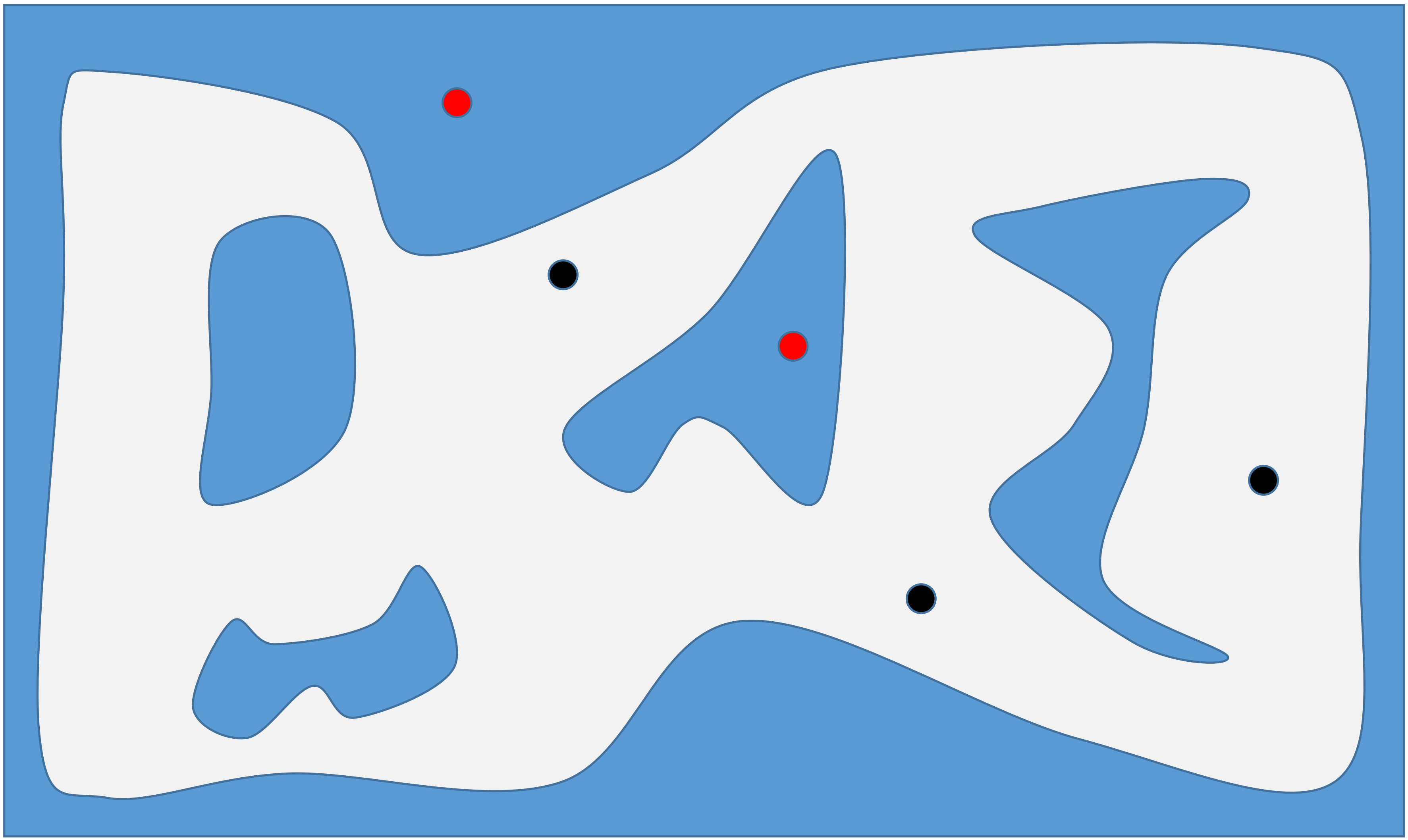
Probabilistic Roadmap in More Detail



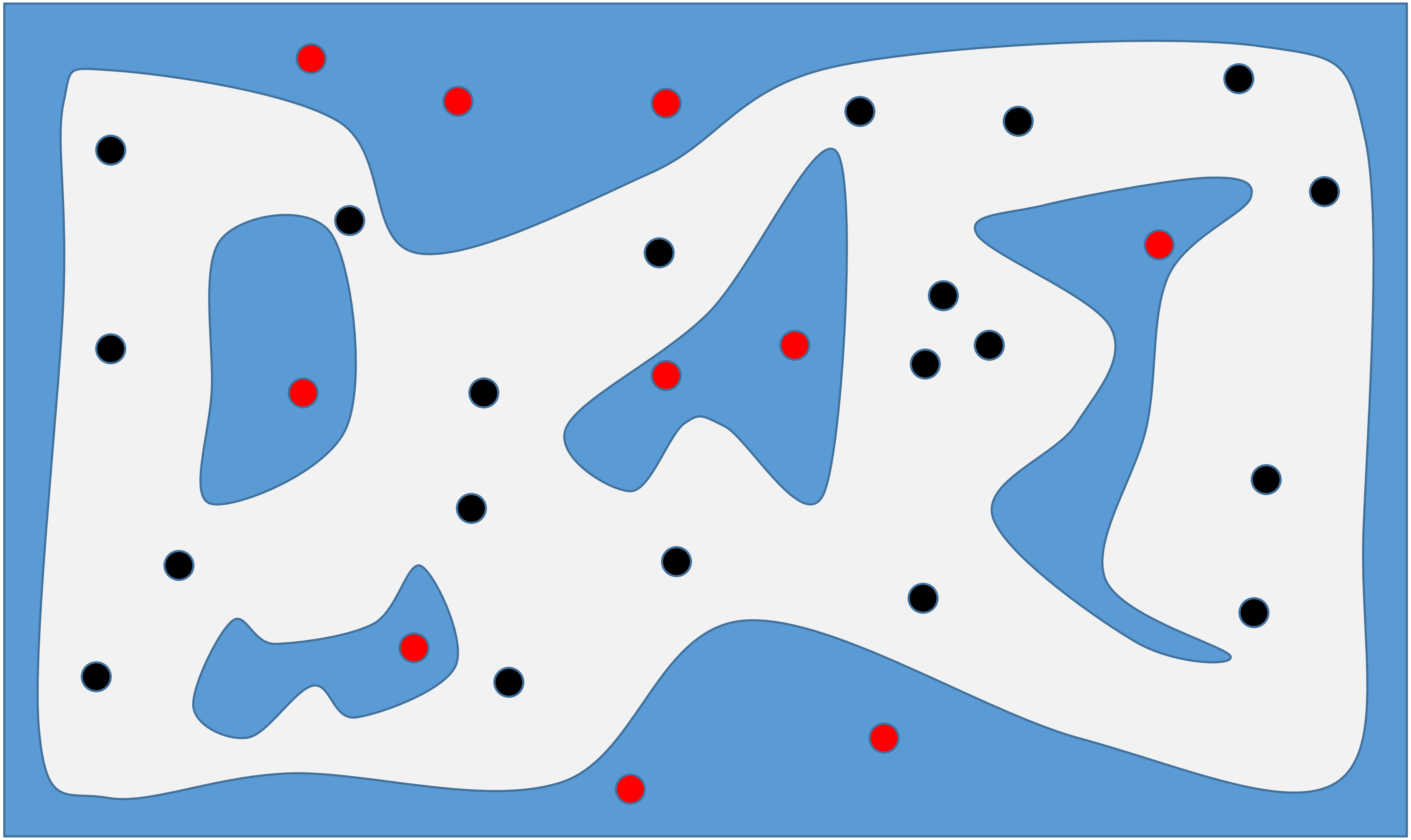
Generating Random Samples



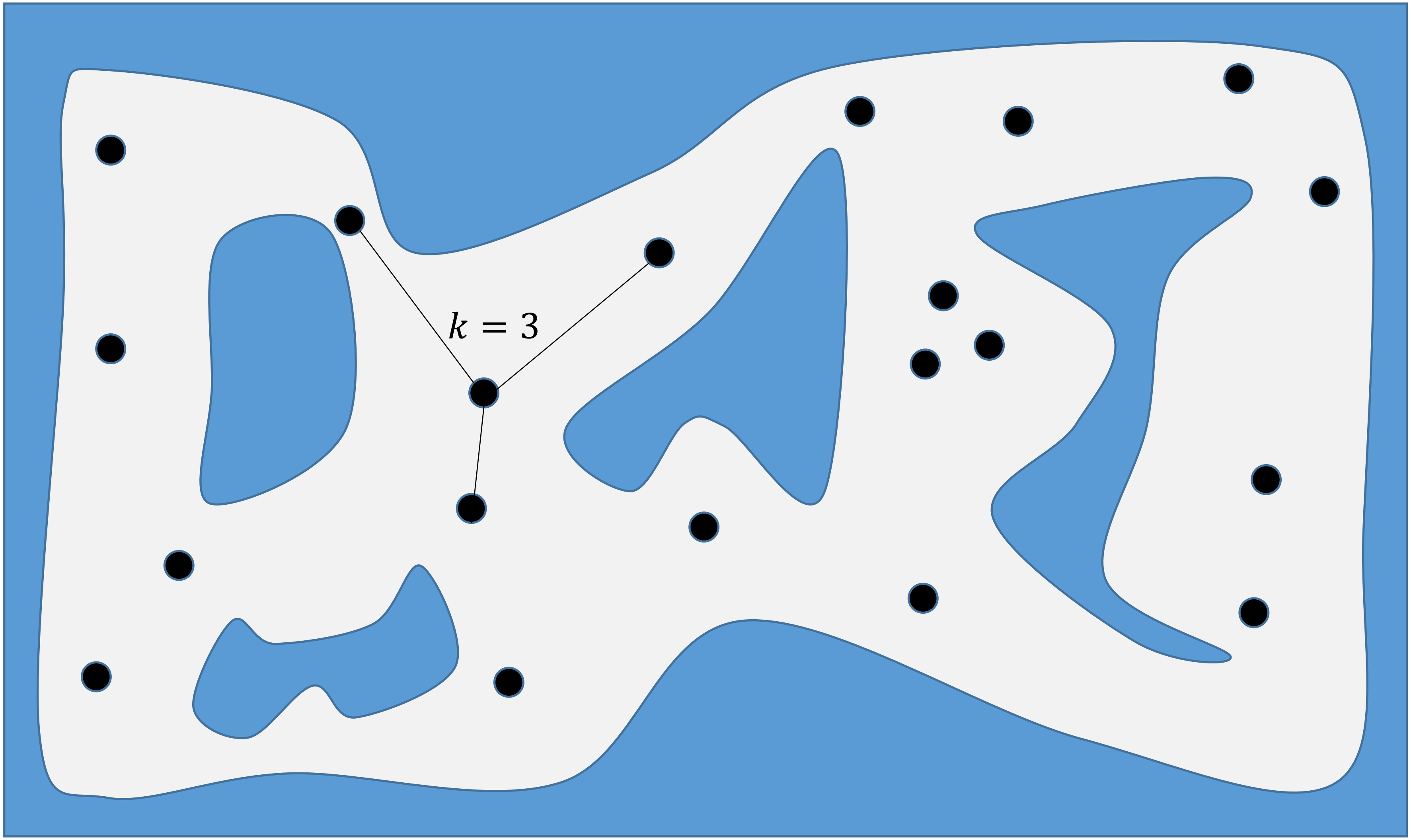
Rejecting Samples Outside C_{free}



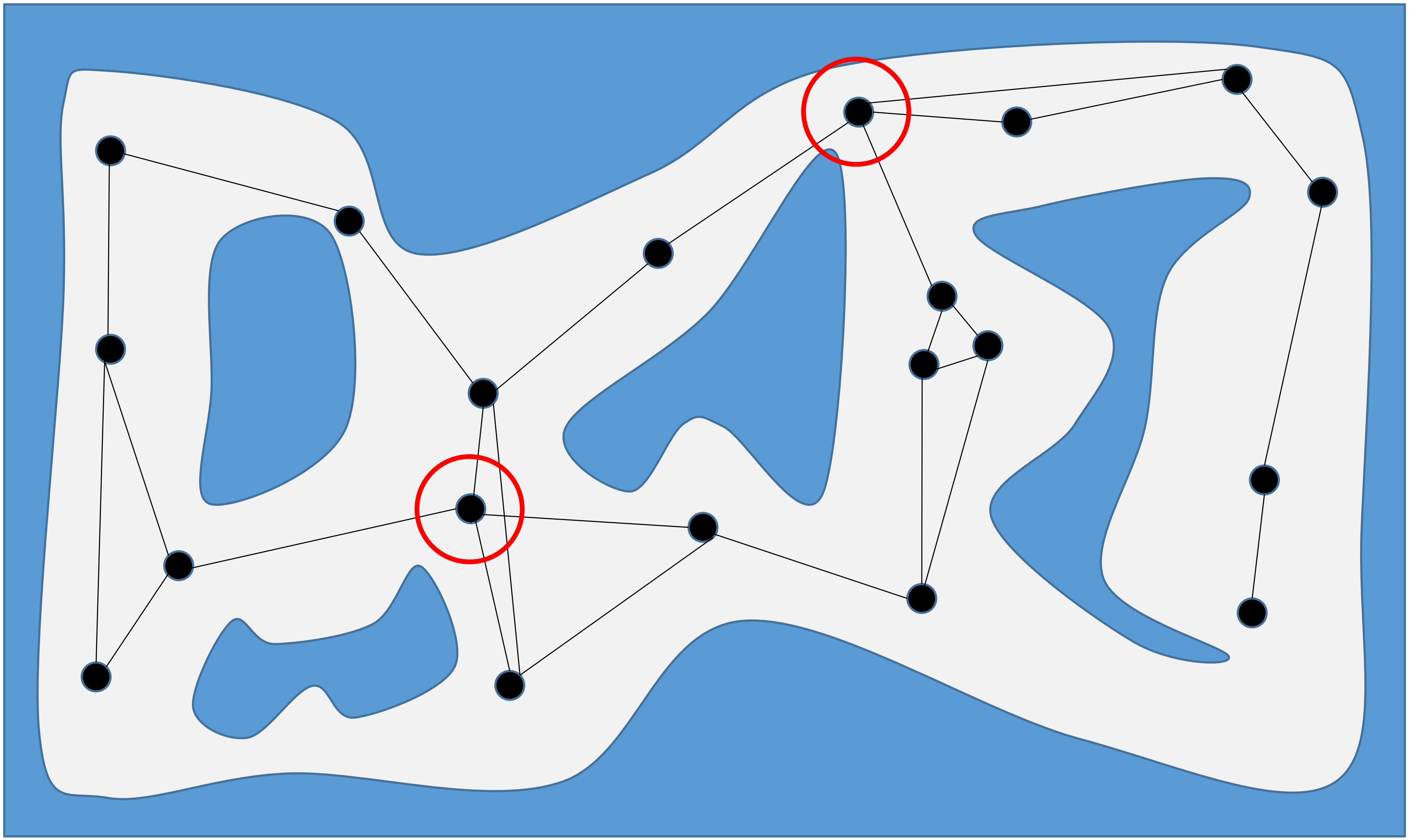
Collecting Enough Samples in C_{free}



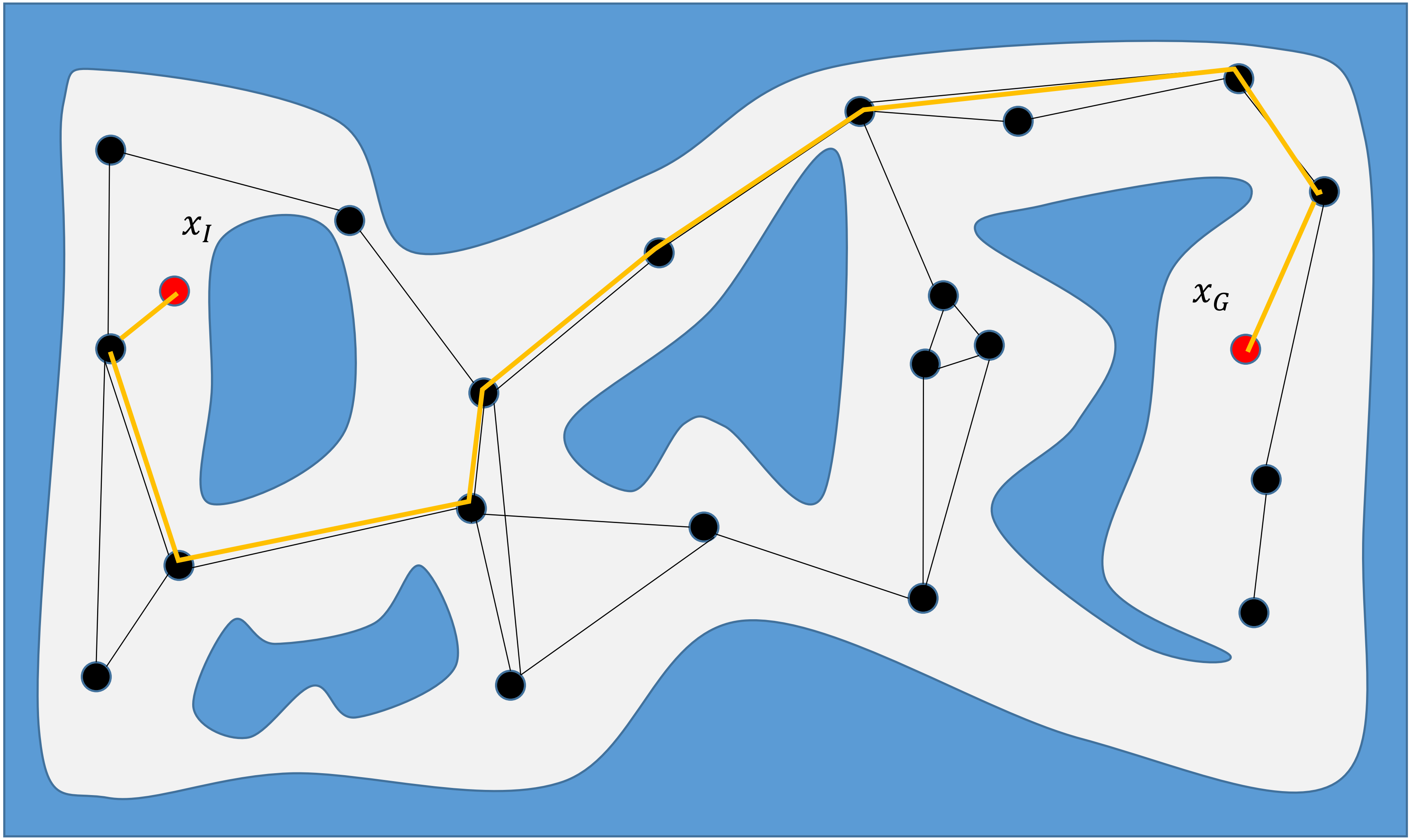
Connect to k Nearest Neighbors (If Possible)



Connect to k Nearest Neighbors (If Possible)



Query Phase

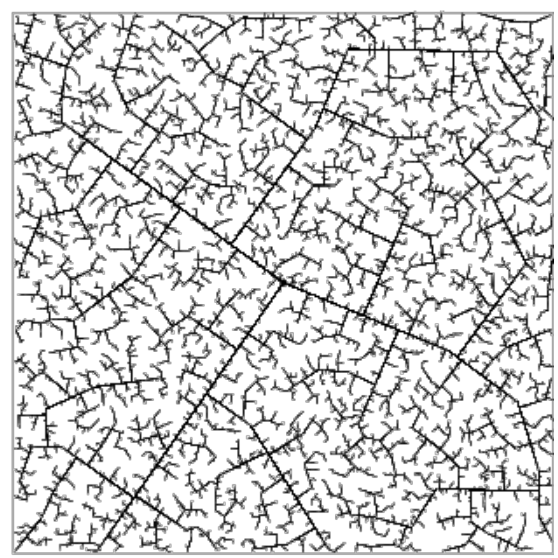


Drawbacks of Multi-Query Methods

PRM is known as a "multi-query" sampling-based method because after initial roadmap is built, multiple queries can be executed on the same roadmap

- ⇨ But, this also means that the roadmap is likely to have a lot of useless information stored if we want to run a single query
- ⇨ People developed **single-query** methods to handle such situations
- ⇨ One method is the rapidly-exploring random trees (RRT, by LaValle & Kuffner)

45 iterations



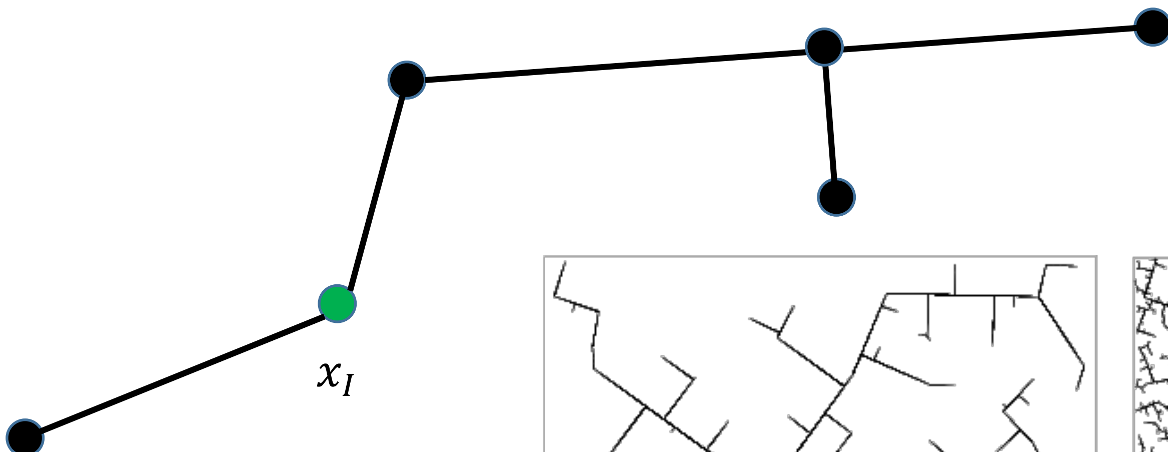
2345 iterations

Image sources: Planning Algorithms by LaValle

Rapidly-Exploring Random Trees w/o Obstacle

RRT without obstacle simply grows a tree from a point

⇨Basically, tries to connect new points to the closest part of the existing tree



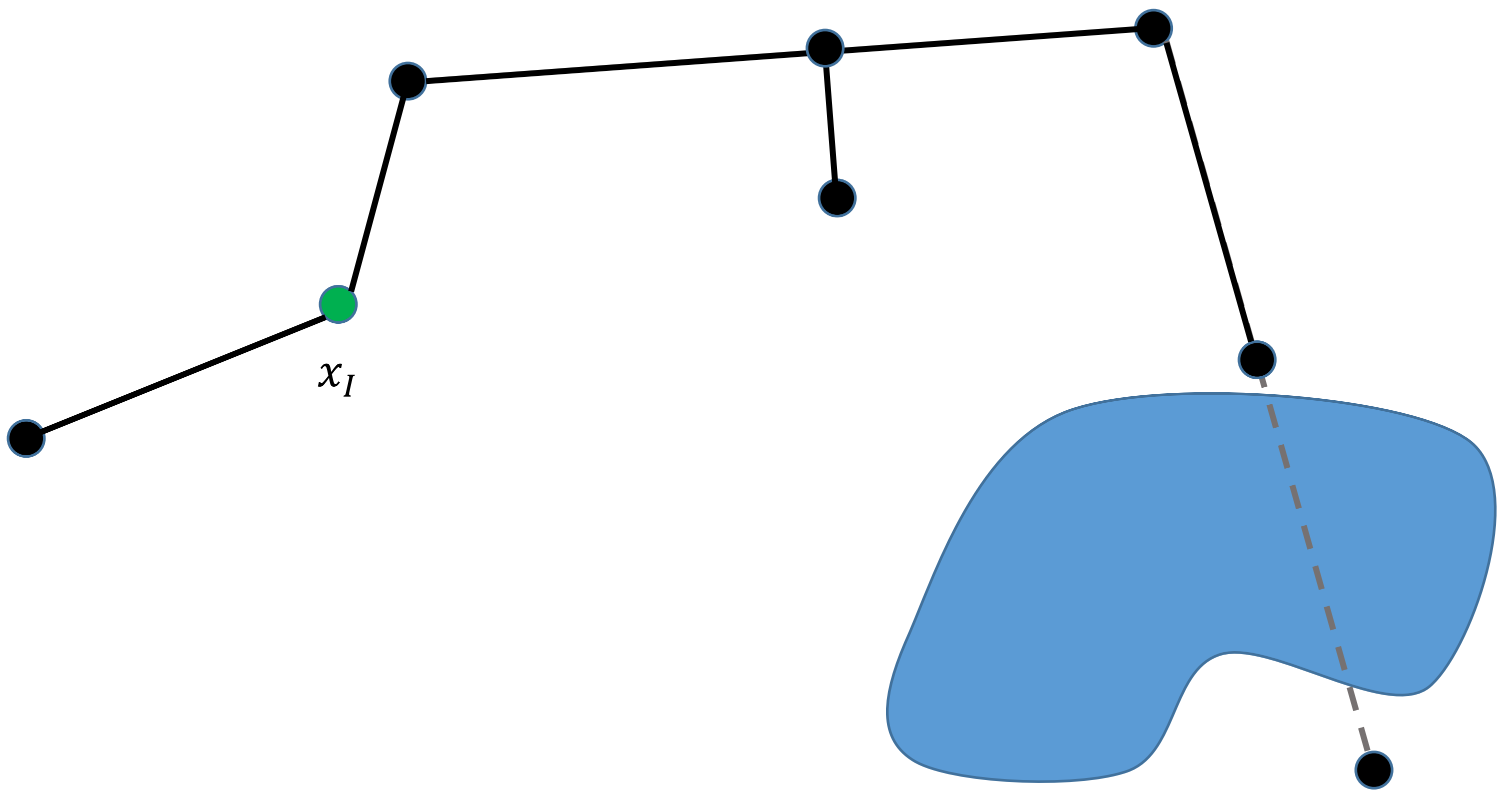
45 iterations

2345 iterations

Image sources: Planning Algorithms by LaValle

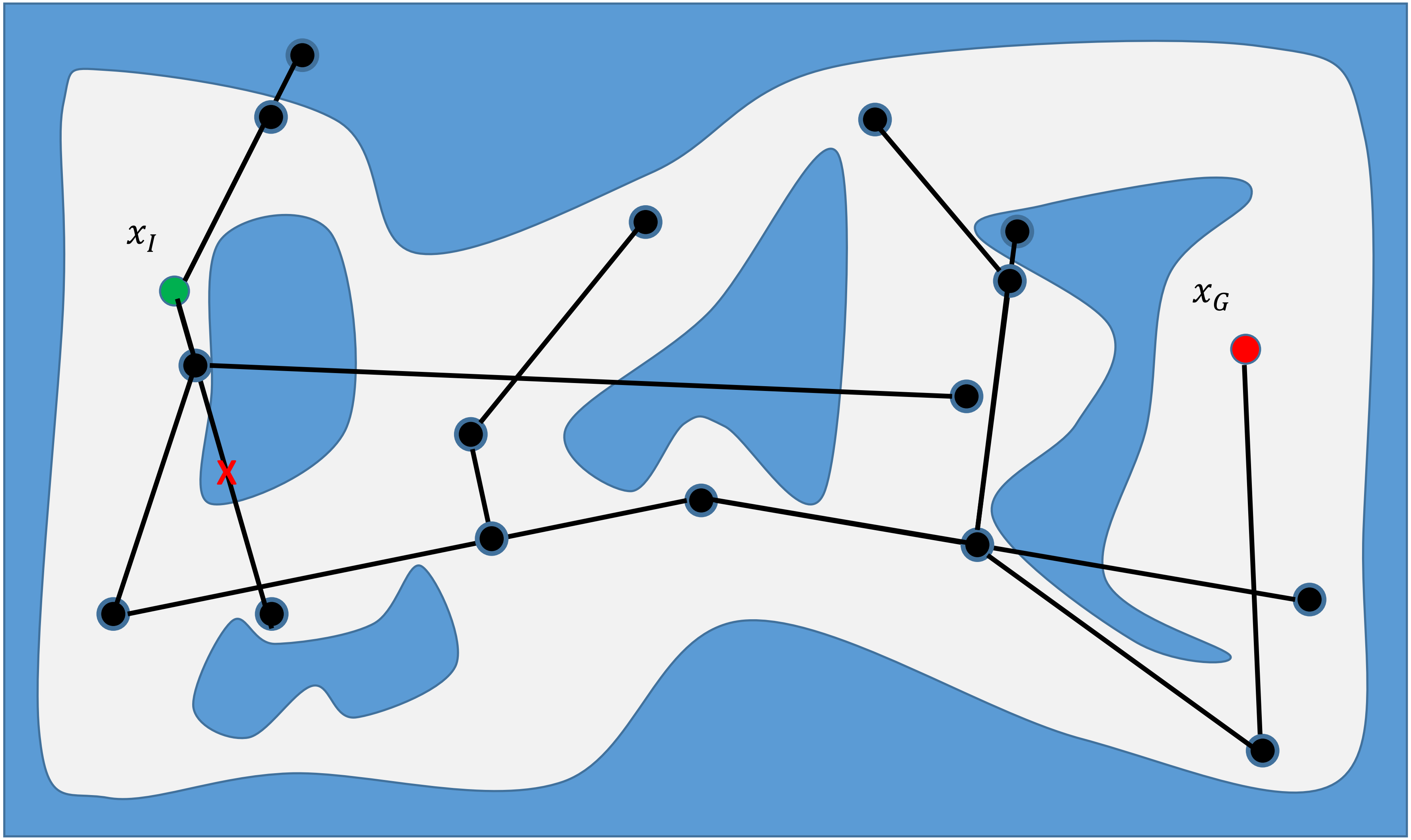
RRT with Obstacles

When there are obstacles, try to extend the tree as much as possible



Same procedure if sample falls inside an obstacle

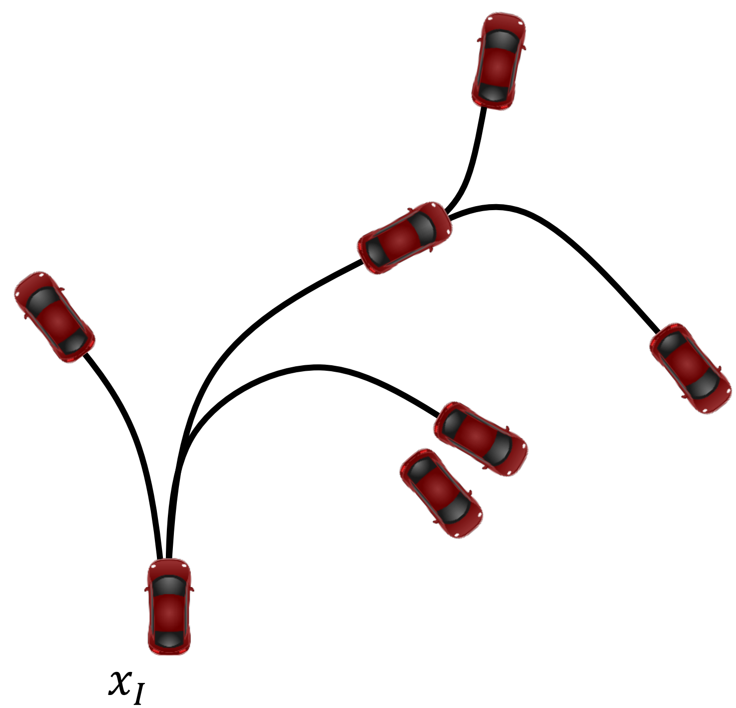
Tree Building Example



Kinodynamic RRT

We can grow an RRT respecting the differential constraints

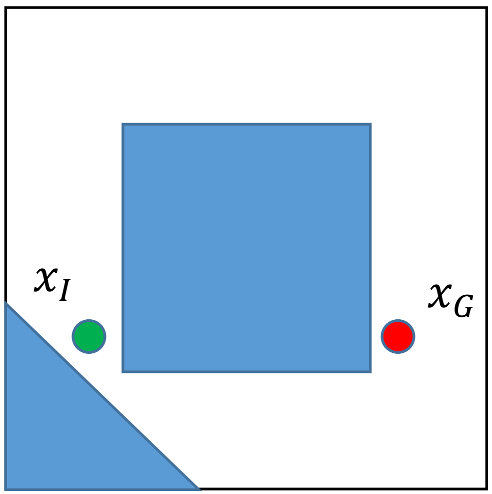
- ⇨Standard PRM and RRT cannot be applied!
- ⇨Need to compute path more carefully
    - ⇨ Needs to solve a boundary value problem (differential equations)
- ⇨Example w/o obstacles



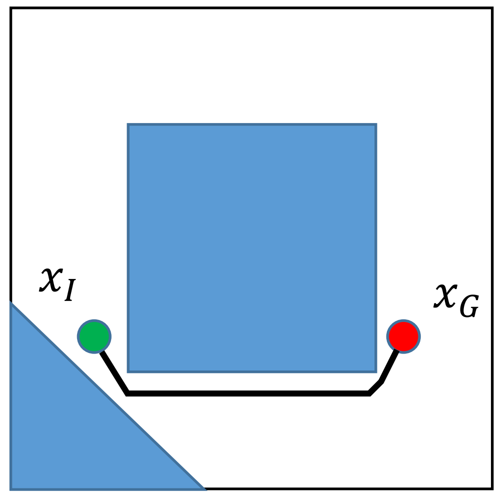
Non-Optimality of PRM and RRT

PRM and RRTs are not optimal

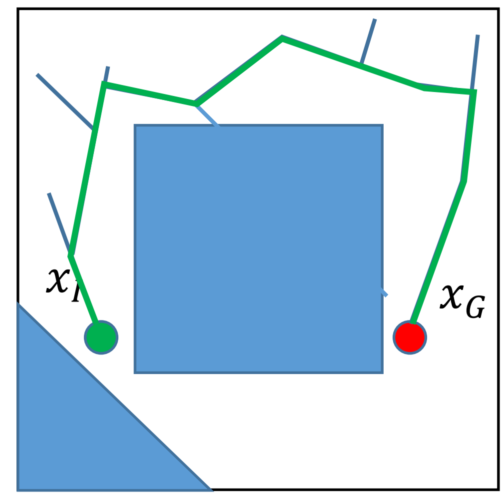
⇨It is possible construct instances to make PRM/RRT produce long paths



Problem



Optimal solution

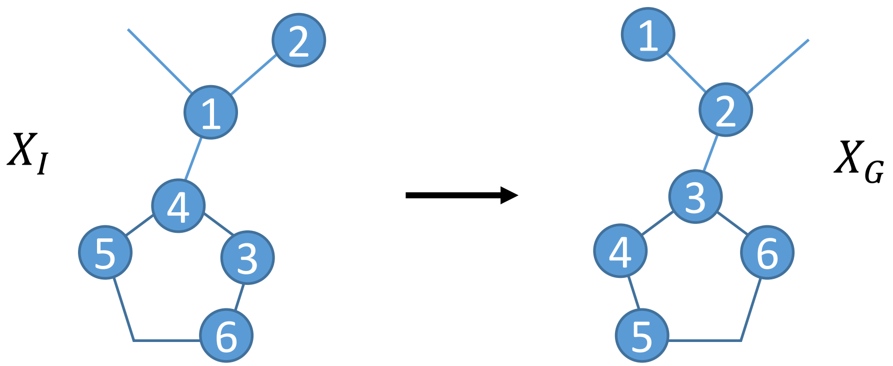


Likely RRT solution

⇨Can we do better?

- ⇨ Need to keep “re-wiring” the graph structure

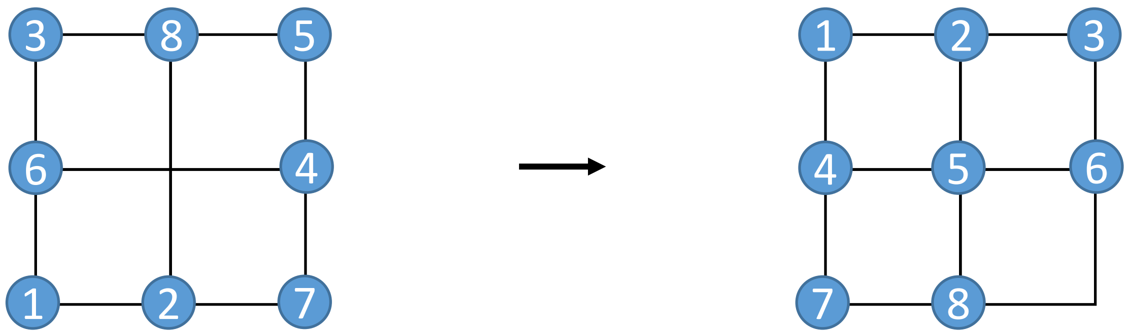
Multi-Robot Path Planning



MPP Problem: (G, X_I, X_G), solution: collision free $P = \{p_1, \ldots, p_n\}$
Optimality objectives (minimization):

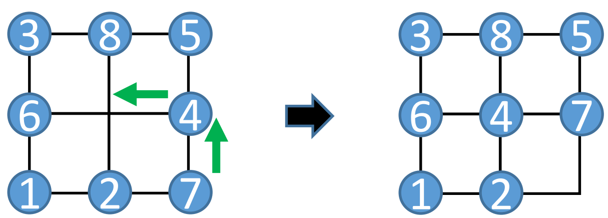
- ⇨ Max time (makespan): $\min_{P \in \mathcal{P}} \max_{p_i \in P} time(p_i)$
- ⇨ Total time: $\min_{P \in \mathcal{P}} \sum_{p_i \in P} time(p_i)$
- ⇨ Max distance: $\min_{P \in \mathcal{P}} \max_{p_i \in P} length(p_i)$
- ⇨ Total distance: $\min_{P \in \mathcal{P}} \sum_{p_i \in P} length(p_i)$

A Simple Method for $N^2 - 1$ Puzzle Feasibility

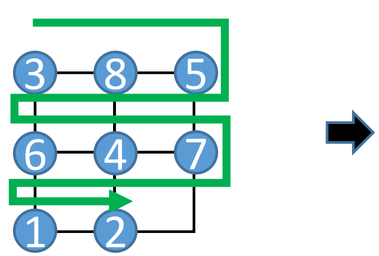


Steps

1. Move the empty spot to the lower right (doesn't matter how you do it)



2. Flatten the square row by row



38564712

3. Bubbling each number from 1 and count number of moves

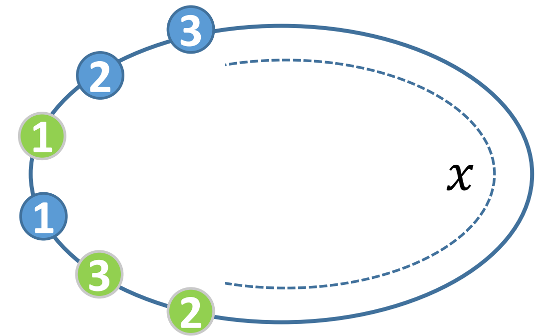
38564712	Moved 1 over 385647, 6 numbers →	13856472
13856472	Moved 2 over 385647, 6 numbers →	12385647
12385647	Nothing to do for 3, 0 number →	12385647
12385647	Moved 4 over 856, 3 numbers →	12348567
12348567	Moved 5 over 8, 1 number →	12345867
12345867	Moved 6 over 8, 1 number →	12345687
12345687	Moved 7 over 8, 1 number →	12345678

4. Sum up $X = 6 + 6 + 0 + 3 + 1 + 1 + 1 = 18$

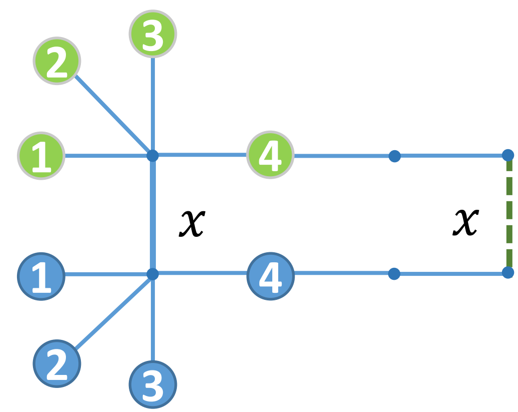
5. Odd – infeasible. Even – feasible. $X = 18$, instance is feasible

Incompatibility of the Formulations

	Makespan	Total time
Clockwise	$x+1$	$2x+3$
Counterclockwise	$x+4$	$x+12$



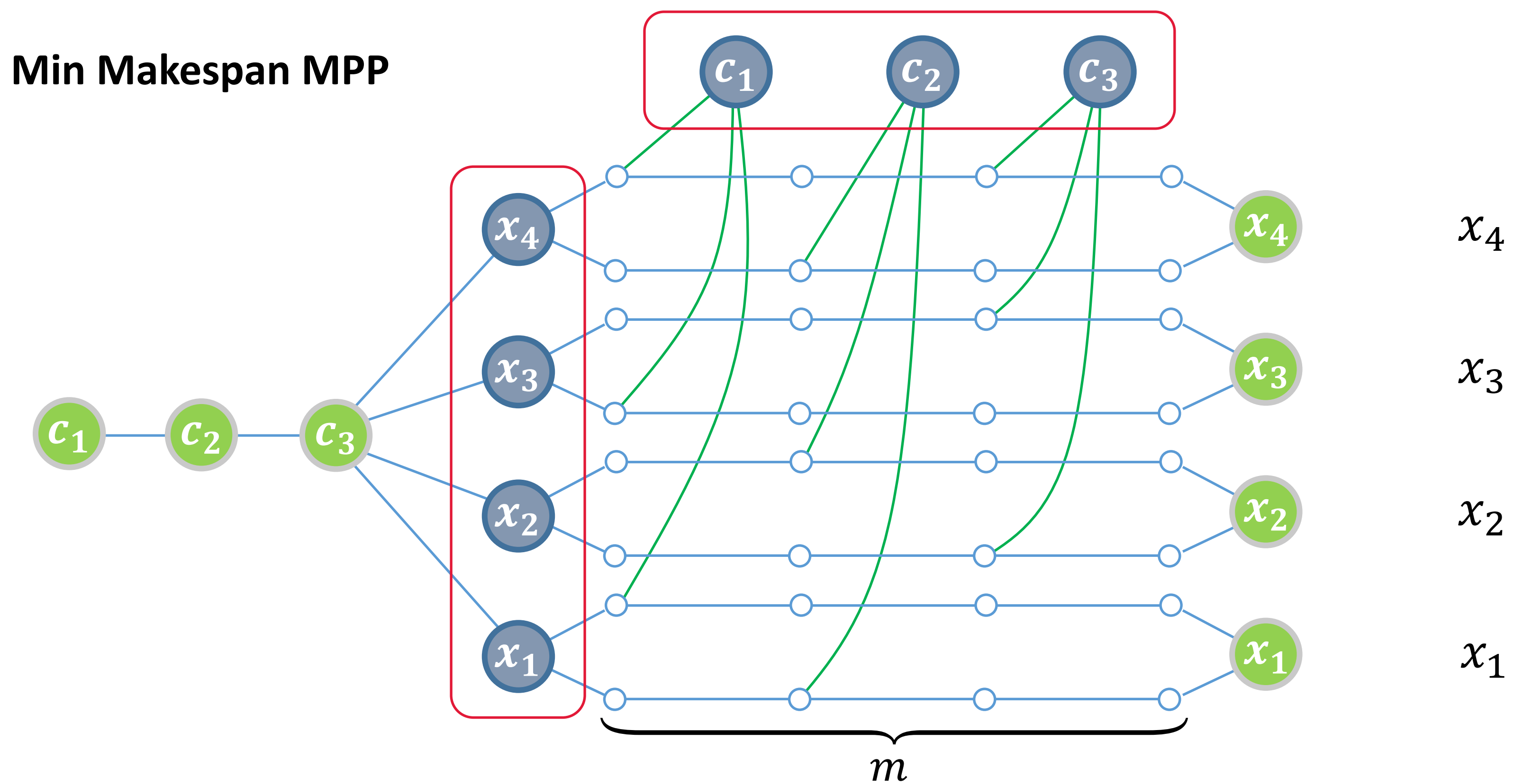
	Total distance	Total time
Left path only	$4x+8$	$4x+14$
Using right path	$4x+10$	$4x+13$



A pair of the four MPP objectives on makespan, total time, max distance, and total distance demonstrates a Pareto-optimal structure.

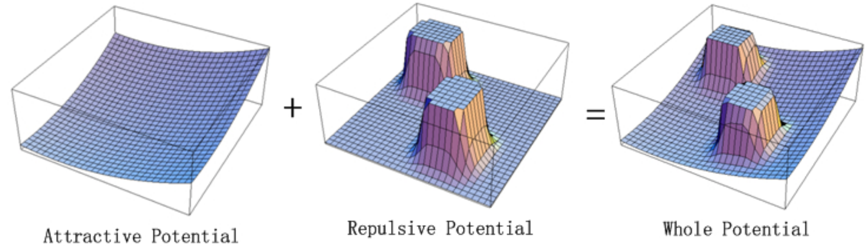
NP-Hardness of Makespan Optimal $\mathrm{MPP_r}$

Min Makespan $\mathrm{MPP_r}$ is NP-hard



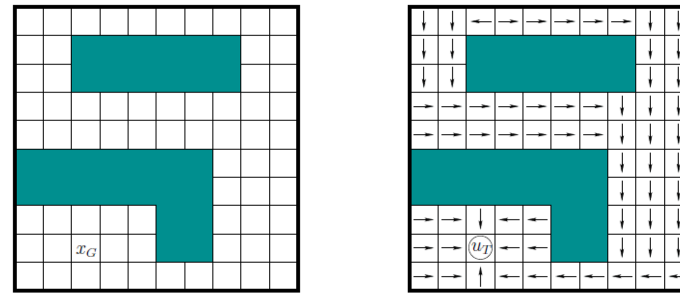
Theorem. MPP is NP-hard when optimizing min makespan, min total time, min max distance, and min total distance.

Other Planning Methods



Potential fields

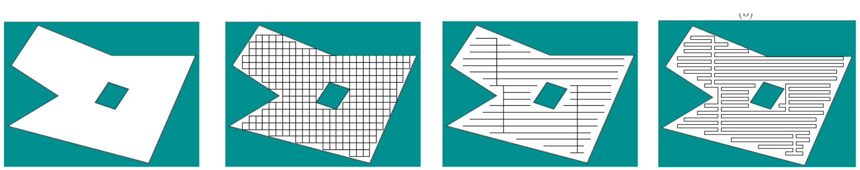
Feedback-based planner

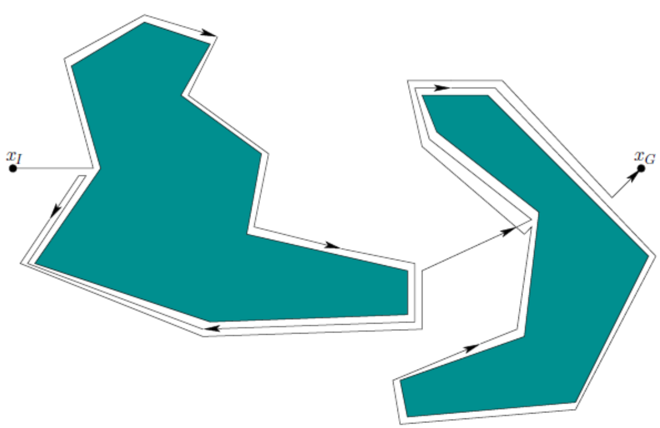


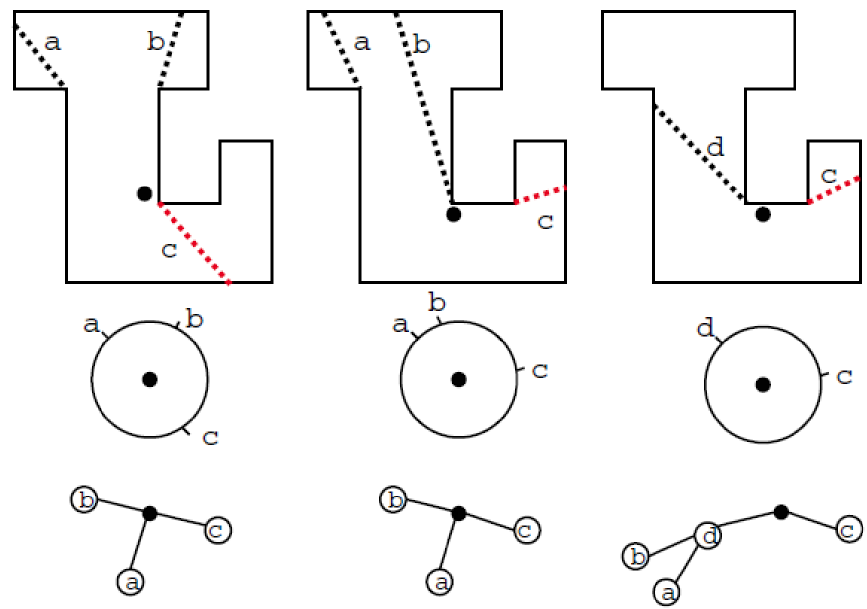
Spanning tree doubling (for coverage)

Bug algorithms

Gap-navigation trees







A Little History on Modern Feedback Control

After steam engine was invented, how to control its running speed is a problem of major interest

A successful design was Watt's flyball governor

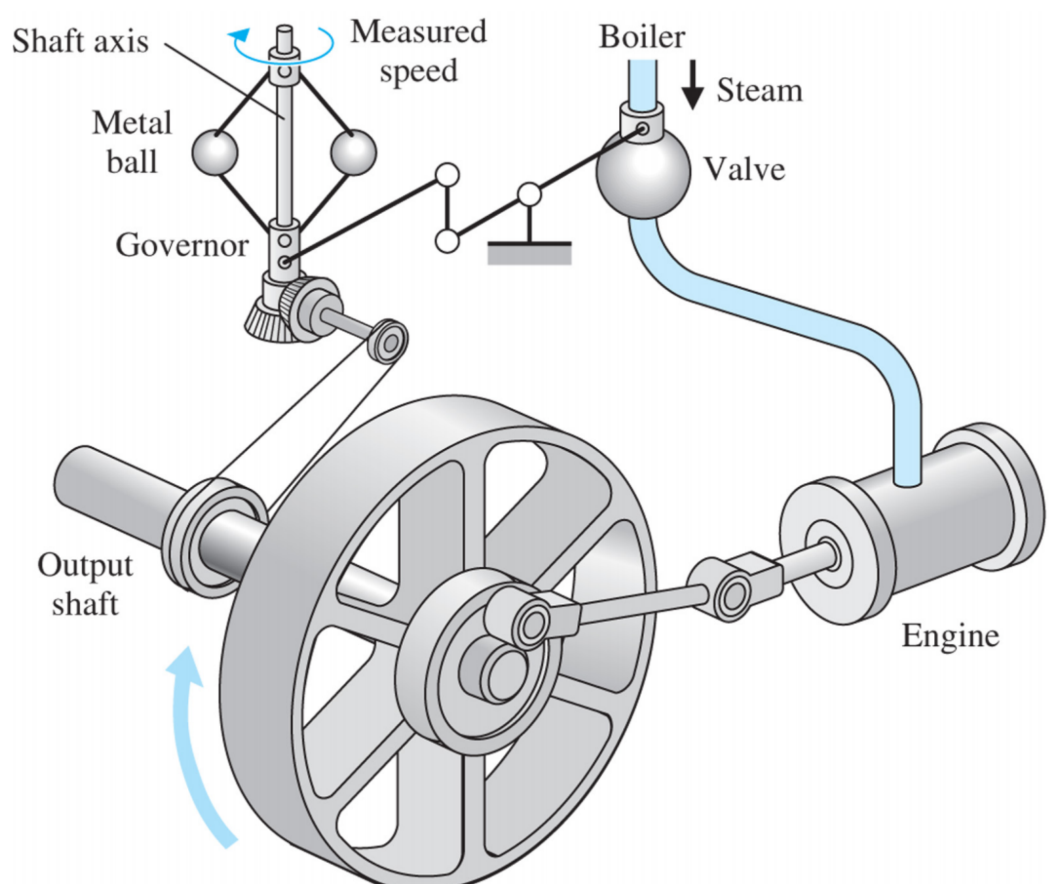


Image source: http://www.ece.mcmaster.ca/~davidson/

PID Controller

PID controller stands for **proportional-integral-derivative controller**

⇨There are many different “theoretical” feedback controllers

⇨However, the final implementation often uses some form of PID control

General form:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

$$e(t) = Set\ Point\ - Current\ Location$$

Block diagram

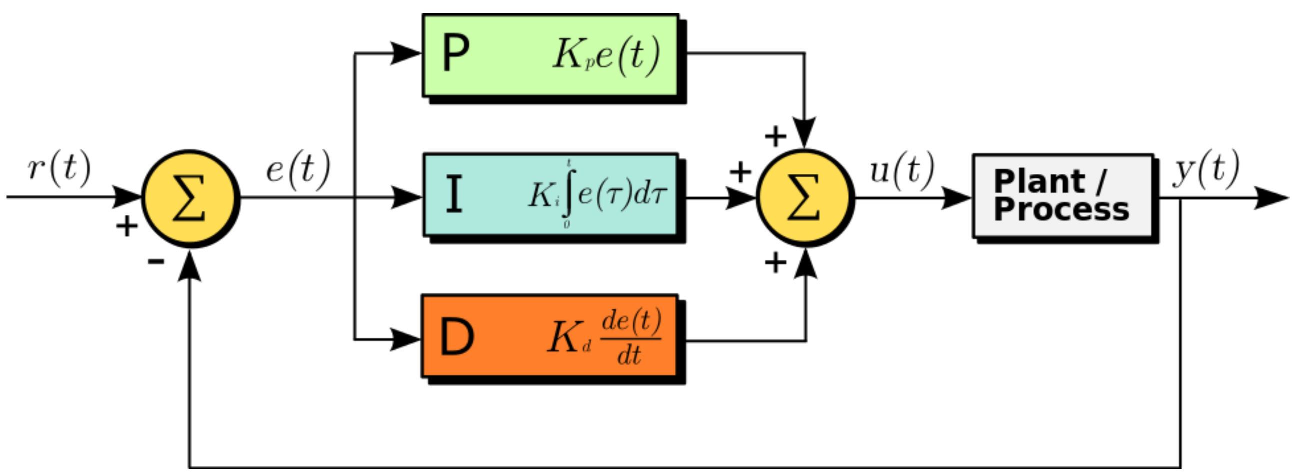


Image source: wikipedia.org

Pure Pursuit for Differential Drive Robots

Most two wheeled robots can be viewed as a **differentially driven robot** (DDR)

⇨ Two wheel inputs in the range of $[-1, 1]$

Pure pursuit path following algorithm

⇨ From the current location of car, locate a waypoint of distance ℓ (some constant) on the desired trajectory

⇨ Compute the required curvature to the waypoint

⇨ Adjust wheel speeds to follow the computed arc

⇨ Note: the car’s direction is tangential to the computed arc