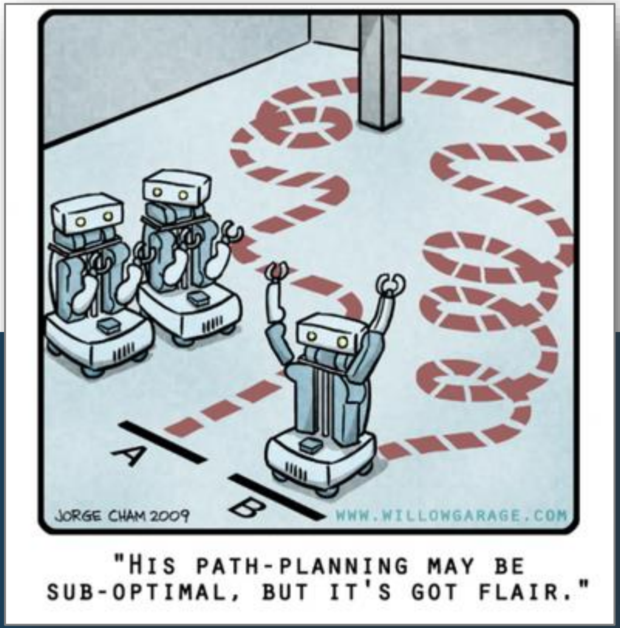


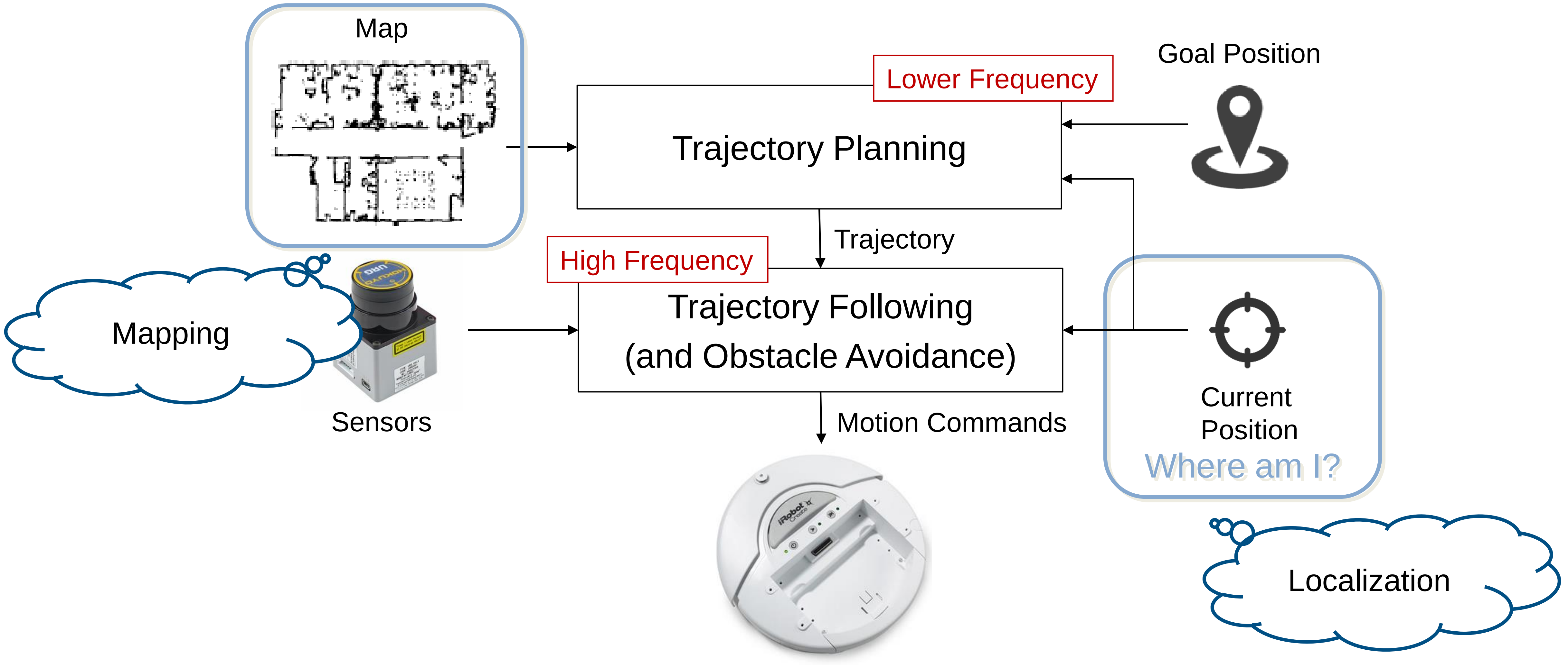
Robotics

Robot Motion Control (and Planning)

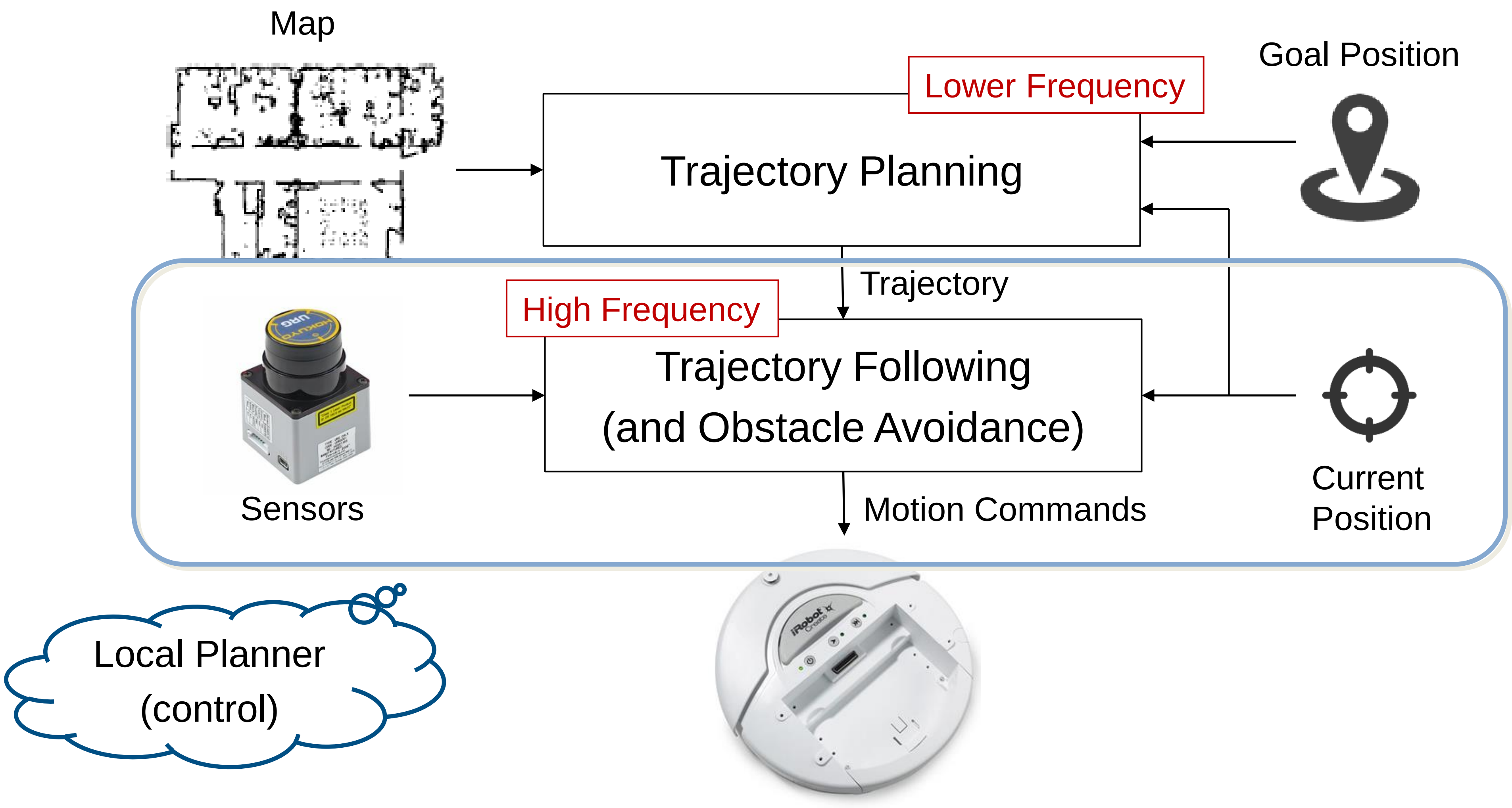
Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

A Simplified Sense-Plan-Act Architecture



A Simplified Sense-Plan-Act Architecture



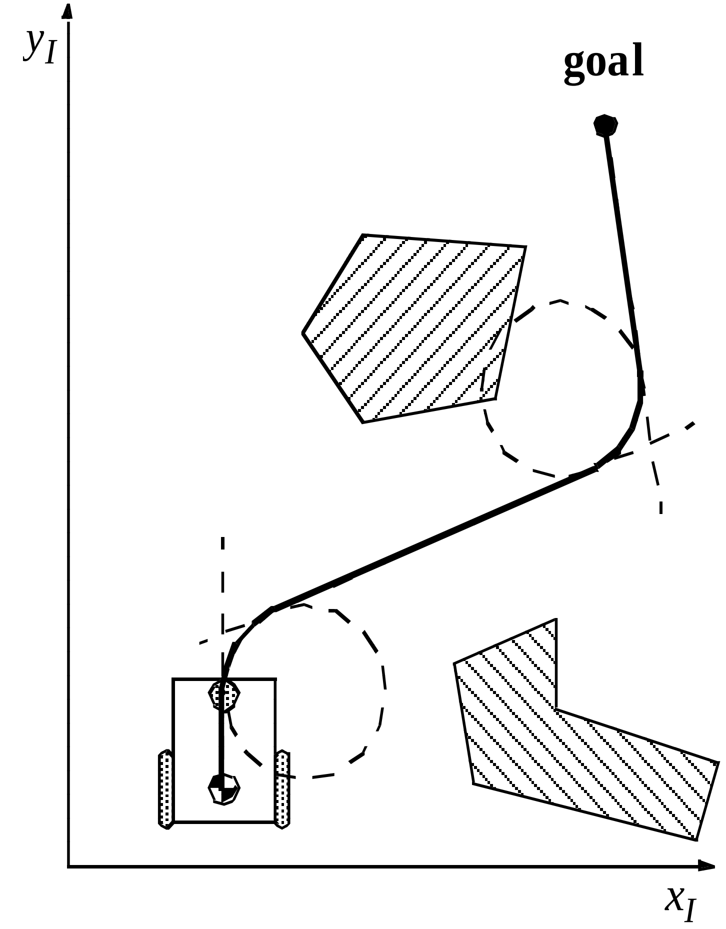
Open loop control

A mobile robot is meant to move from one place to another

- Pre-compute a smooth trajectory based on motion segments (e.g., line or circles) from start to goal
- Execute the planned trajectory till the goal

Disadvantages:

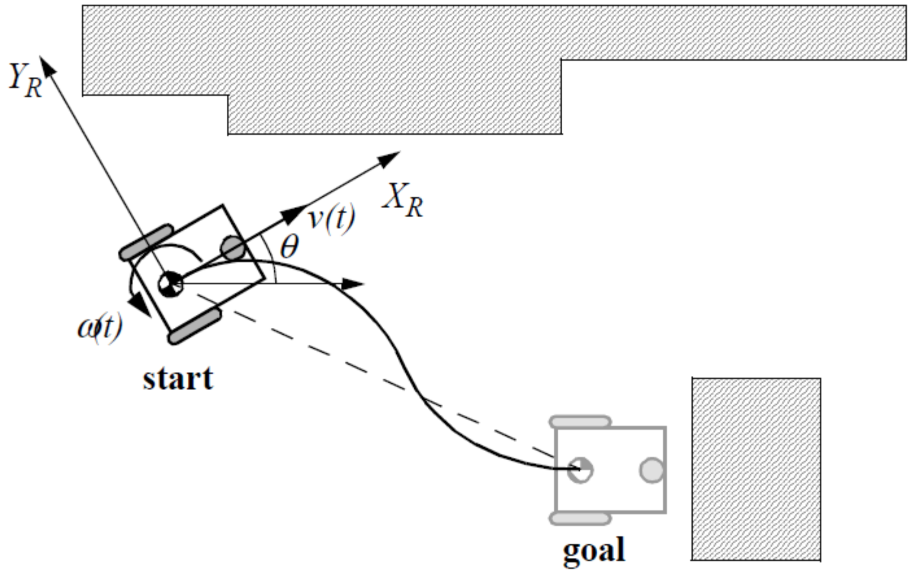
- Not easy to pre-compute a feasible trajectory
- Limitations and constraints of the robots velocities and accelerations
- Does not handle dynamical changes (obstacles)
- No recovery from errors

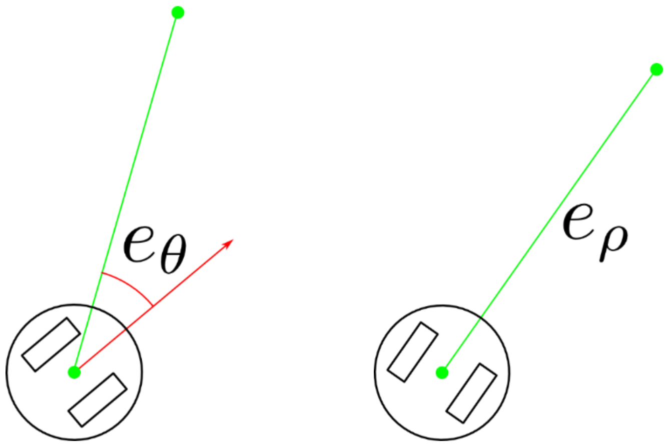


Feedback control (simple diff drive example)

The trajectory is recomputed / adapted online via a simple control schema for path following

- Control orientation acting on angular velocity
- Control distance acting on linear velocity

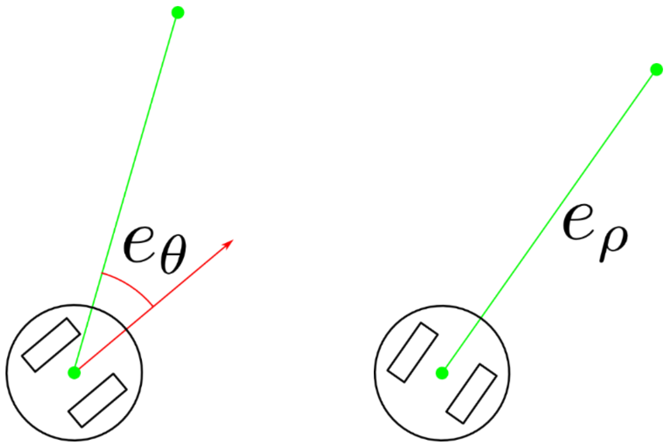


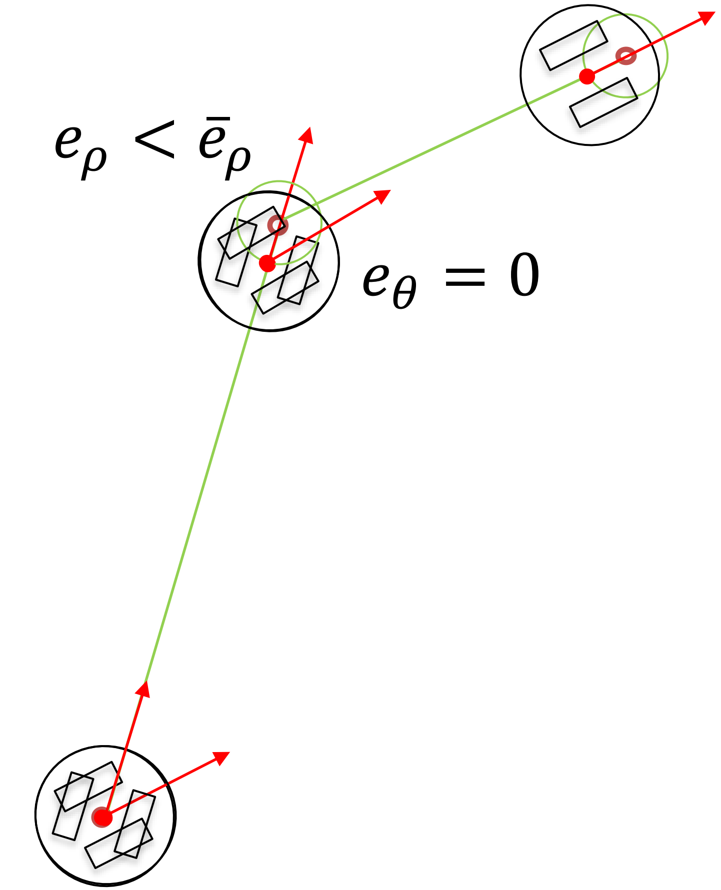


Feedback control (simple diff drive example)

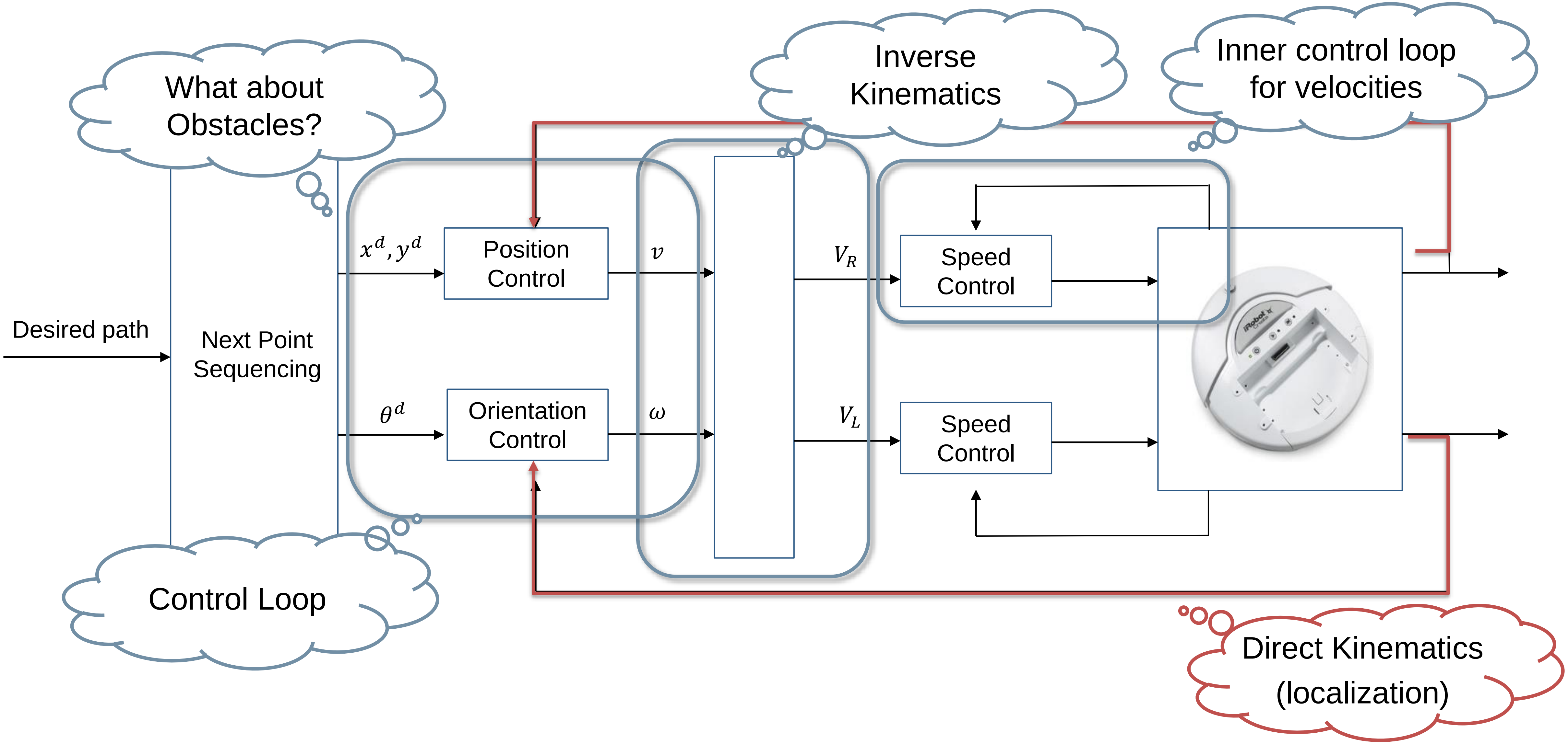
The trajectory is recomputed / adapted online
via a simple control schema for path following

- Control orientation acting on angular velocity
- Control distance acting on linear velocity





Feedback control (simple diff drive example)



Obstacle Avoidance (Local Path Planning)

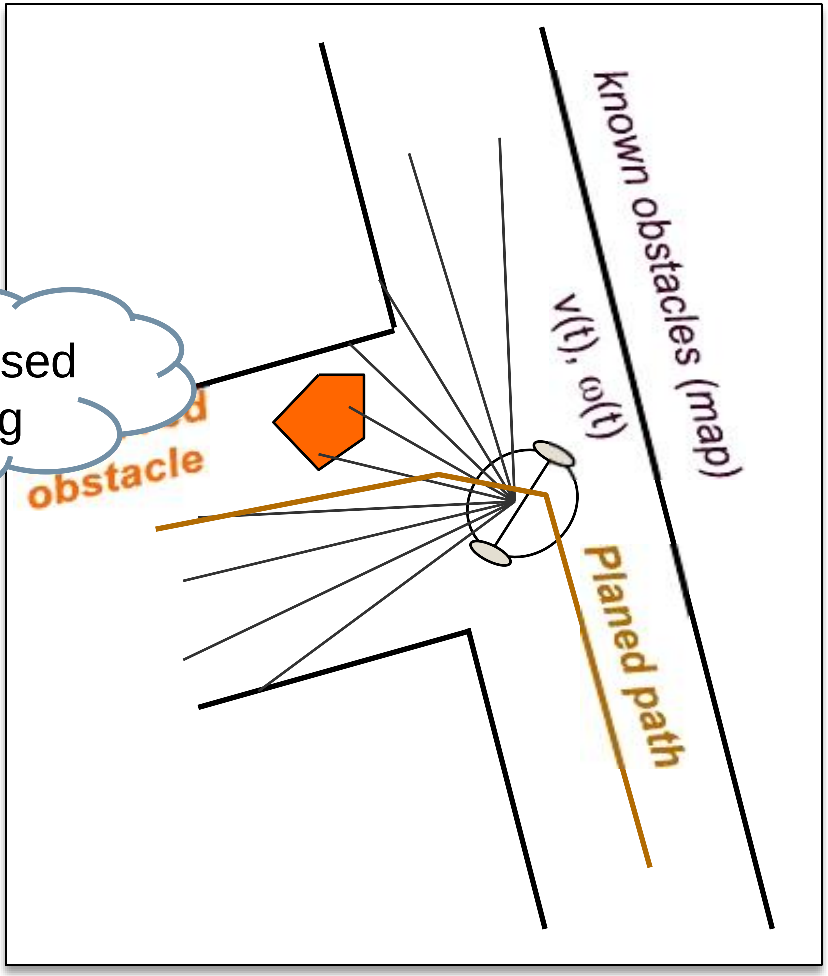
Obstacle avoidance should:

- Follow the planned path
- Avoid unexpected obstacle (i.e., not in the map)

Several proposed methods in the literature

- Potential field methods [Borenstein, 89]
- Vector field histogram [Borenstein, 91, 98, 00]
- Curvature-Velocity [Simmons, 96]
- Nearness diagram [Minguez & Montano, 00]
- Dynamic Window Approach [Fox, Burgard, Thrun, 97]
- …

Sometimes used for planning



The Simplest One …

"Bugs" have little if any knowledge …

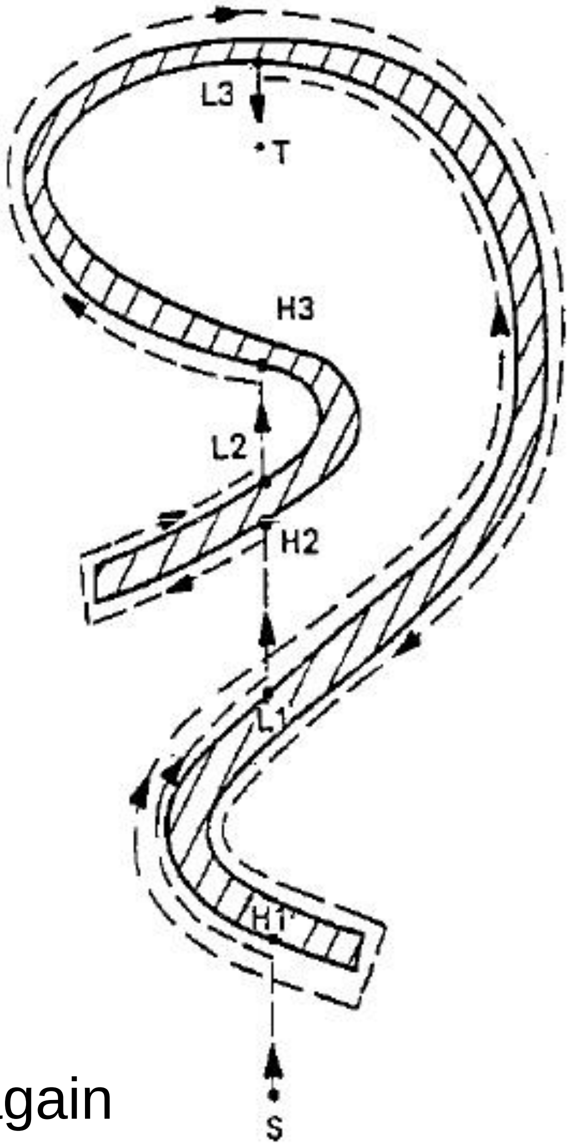
- They known the direction to the goal
- They have local sensing (obstacles + encoders)

… and their world is reasonable!

- Finite obstacles in any finite range
- A line intersects an obstacle finite times

Switch between two basic behaviors

1. Head toward goal
2. Follow obstacles until you can head toward the goal again



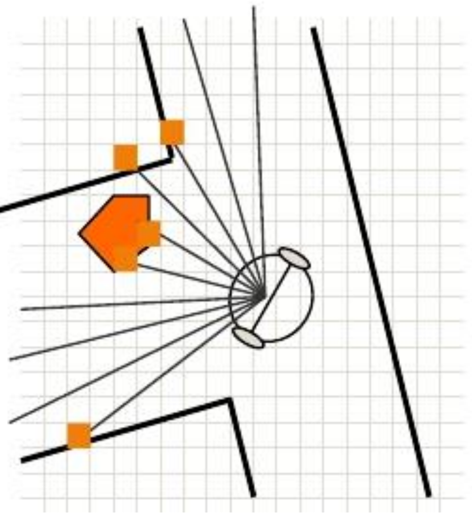
al

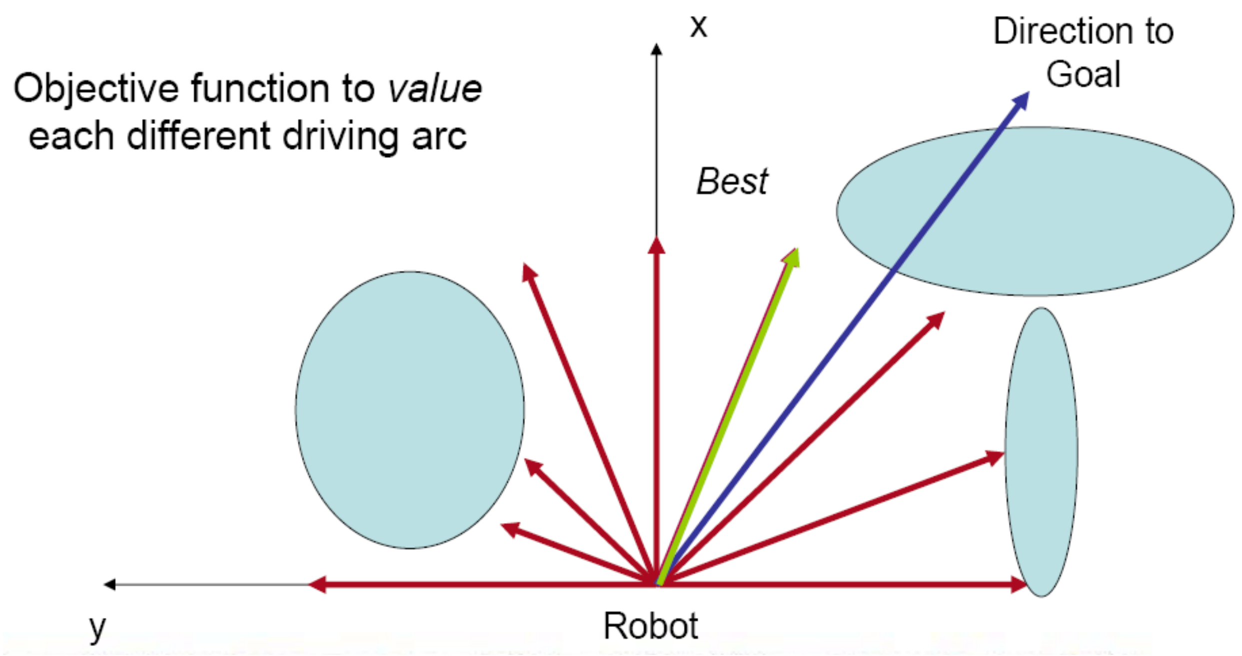
Vector Field Histograms (VFH) *[Borenstein et al. 1991]*

Use a local map of the environment and evaluate the angle to drive towards

- Environment represented in a grid (2 DOF) with local measurements
- All openings for the robot to pass are found

Borenstein et al.

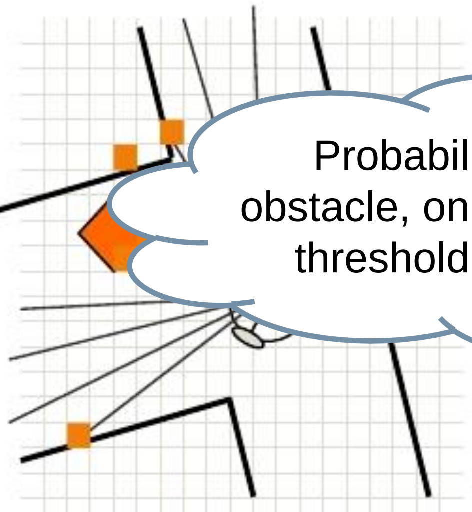




Vector Field Histograms (VFH) *[Borenstein et al. 1991]*

Use a local map of the environment and evaluate the angle to drive towards

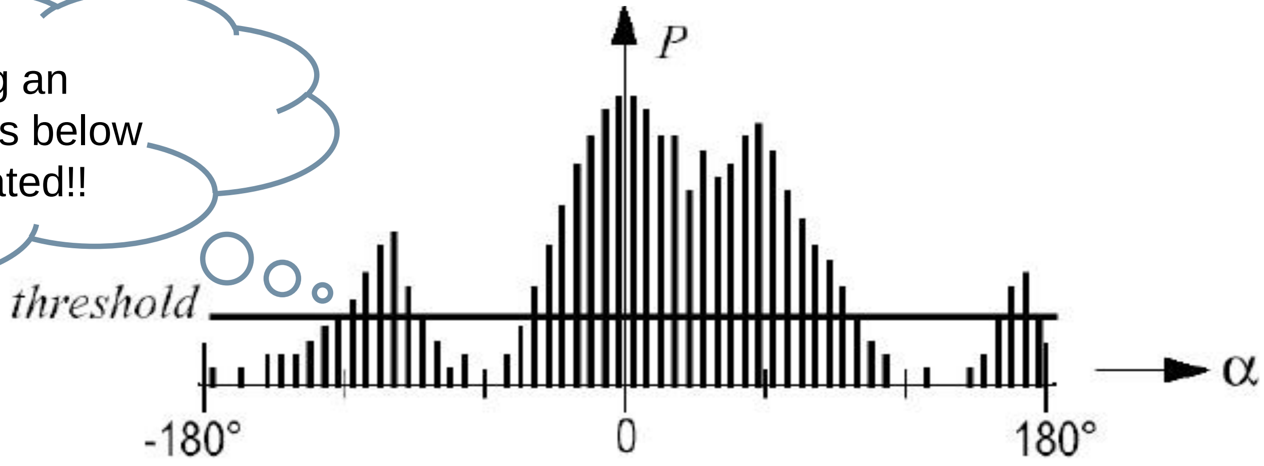
- Environment represented in a grid (2 DOF) with local measurements
- All openings for the robot to pass are found
- The one with lowest cost is selected



Borenstein et al.

$$G = a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_direction}$$

Probability of hitting an obstacle, only directions below threshold are evaluated!!



Vector Field Histograms (VFH) *[Borenstein et al. 1991]*

Use a local map of the environment and evaluate the angle to drive towards

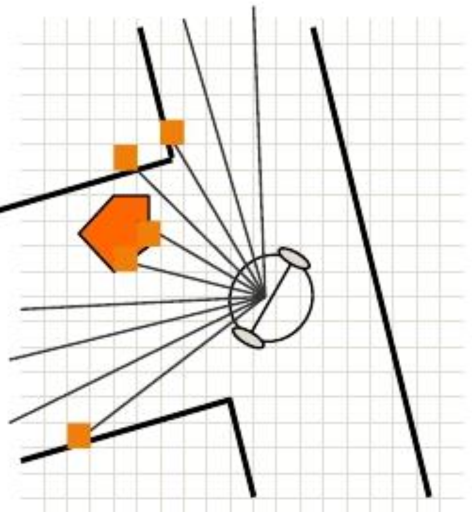
- Environment represented in a grid (2 DOF) with local measurements
- All openings for the robot to pass are found
- The one with

Alignment of the robot path with the goal

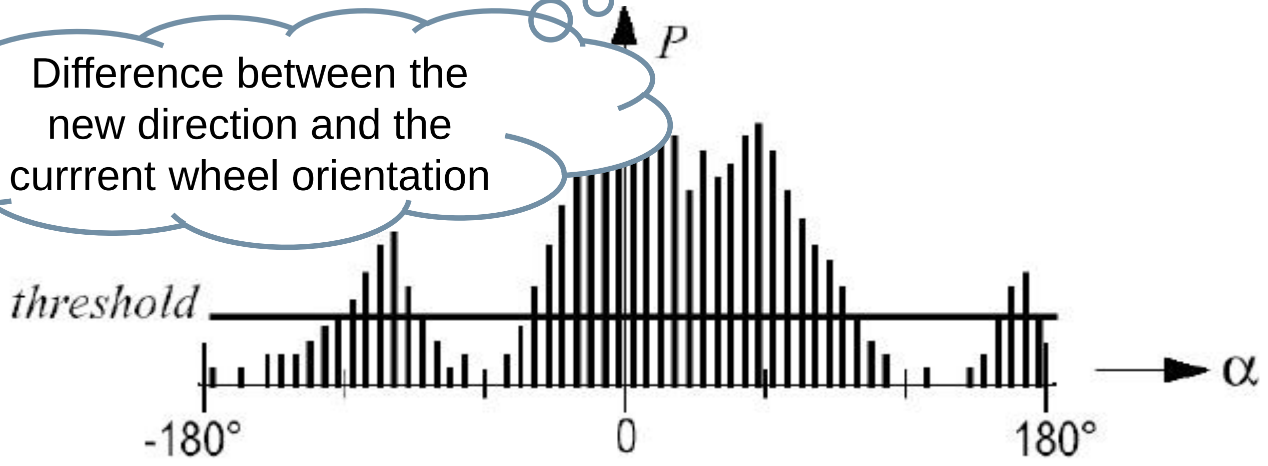
Difference between the previously selected direction and the new direction

$$G = a \cdot \text{target\_direction} + b \cdot \text{wheel\_orientation} + c \cdot \text{previous\_direction}$$

Difference between the new direction and the currrent wheel orientation



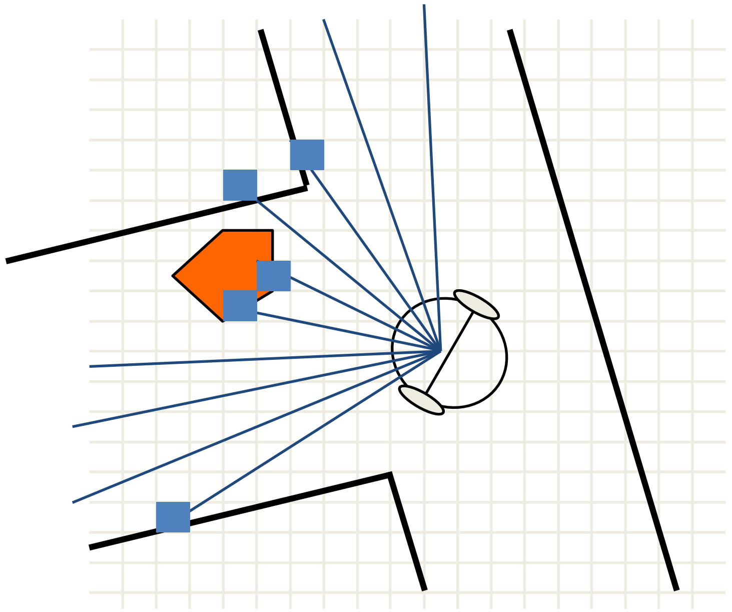
Borenstein et al.

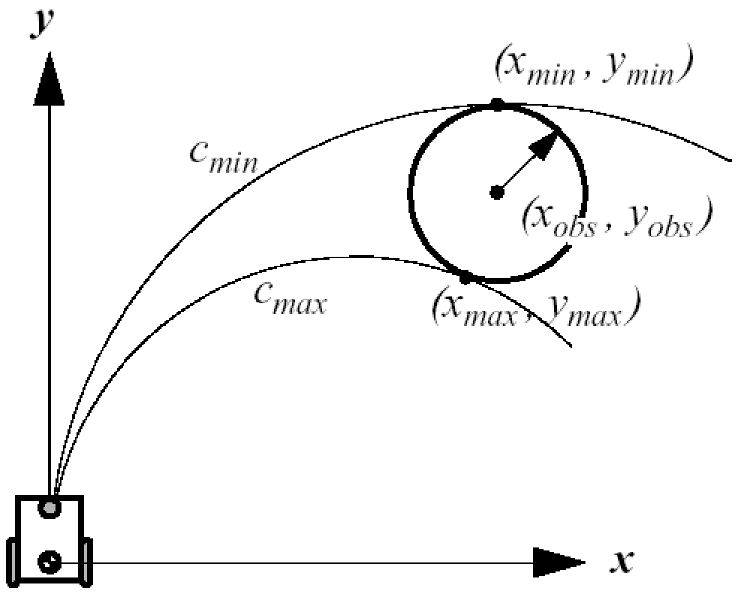


Curvature Velocity Methods (CVM) *[Simmons et al. 1996]*

CVMs add physical constraints from the robot and the environment on (v, w)

- Assumption that robot is traveling on arcs (c= w / v) with acceleration constraints
- Obstacles are transformed in velocity space
- An objective function to select the optimal speed



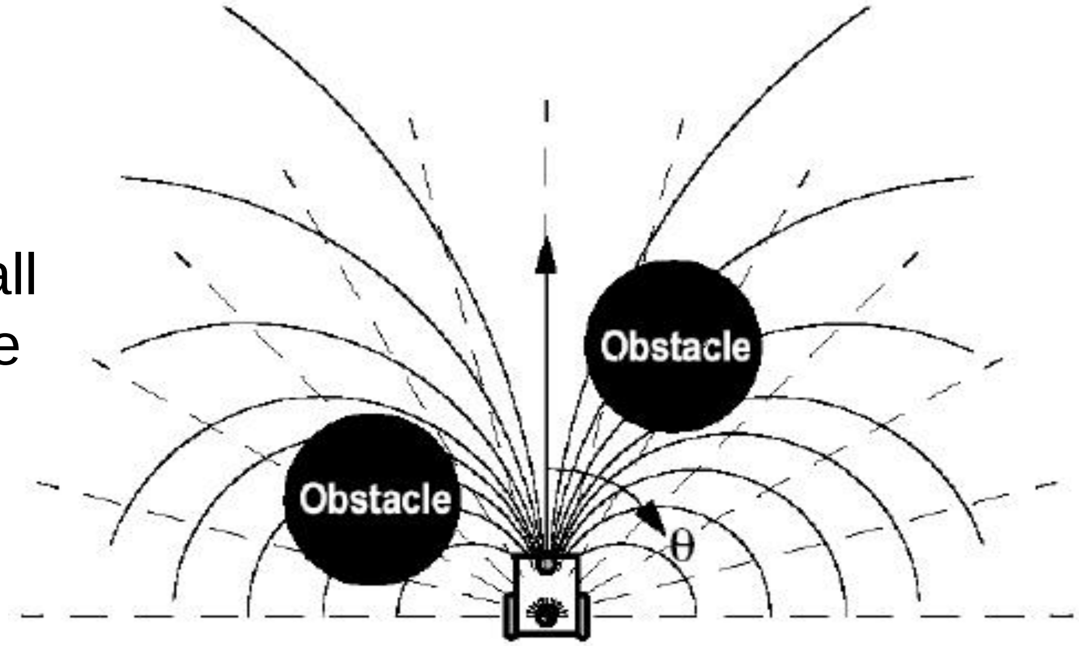


Simmons et al.

Vector Field Histogram+ (VFH+) *[Borenstein et al. 1998]*

VFH+ accounts also for vehicle kinematics

- Robot moving on arcs or straight lines
- Obstacles blocking a given direction blocks all trajectories (arcs) like in an Ackerman vehicle
- Obstacles are enlarged so to account for all kinematically blocked trajectories



Borenstein et al.

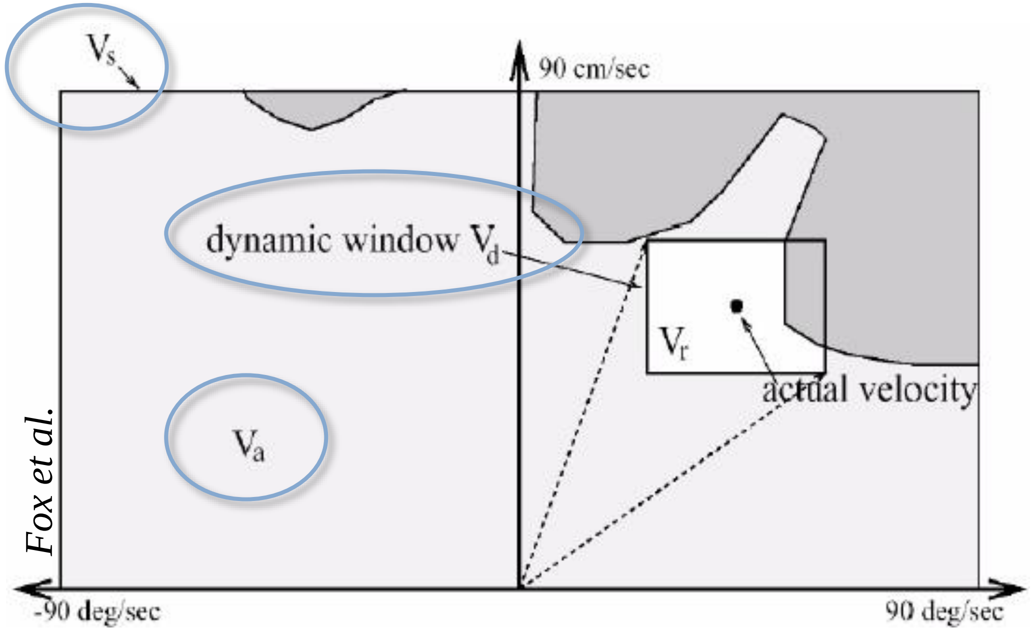
However VFH+ as VFH suffers

- Limitation if narrow areas (e.g. doors) have to be passed
- Local minima might not be avoided
- Reaching of the goal can not be guaranteed
- Dynamics of the robot not really considered

Dynamic Window Approach (DWA) *[Fox et al. 1997]*

The kinematics of the robot are considered via local search in velocity space:

- Consider only circular trajectories via pairs $V_s=(v,\omega)$ of linear and angular speeds
- $V_a=(v, \omega)$ is admissible, if the robot is able to stop before the closest obstacle
- A dynamic window restricts the reachable velocities V_d to those that can be reached within a short time given limited robot accelerations



$$V_d = \begin{cases} v \in [v - a_{tr} \cdot t, v + a_{tr} \cdot t] \\ \omega \in [\omega - a_{rot} \cdot t, \omega + a_{rot} \cdot t] \end{cases}$$

DWA Search Space

$$V_r = V_s \cap V_a \cap V_d$$

How to choose (v,ω)?

Steering commands are chosen maximizing a heuristic navigation function:

- Minimize the travel time by "driving fast in the right direction"
- Planning restricted to V_r space [Fox, Burgard, Thrun '97]

$$G(v,\omega) = \sigma\left(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega)\right)$$

Alignment with target direction

Distance to closest obstacle intersecting with curvature

Forward velocity of the robot

- Global approach [Brock & Khatib 99] in <x,y>-space uses

Forward robot velocity

Follows global path

$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \Delta nf + \delta goal$$

Cost to reach the goal

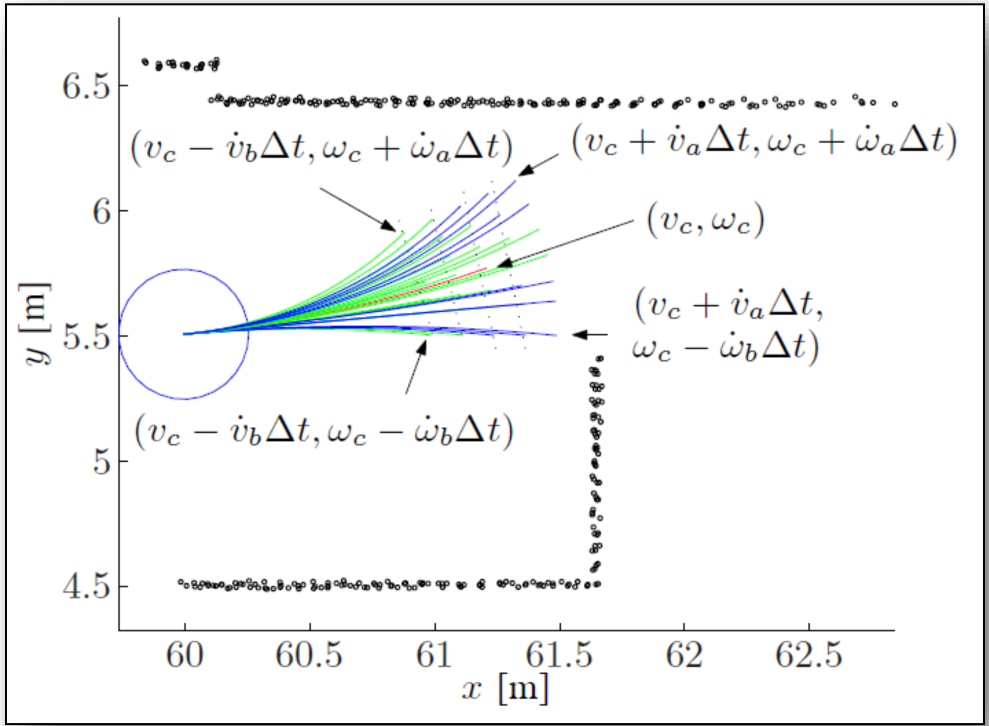
Goal nearness

Navigation Function (NF)

DWA Algorithm (via trajectory rollout)

The basic idea of DWA … but with samples

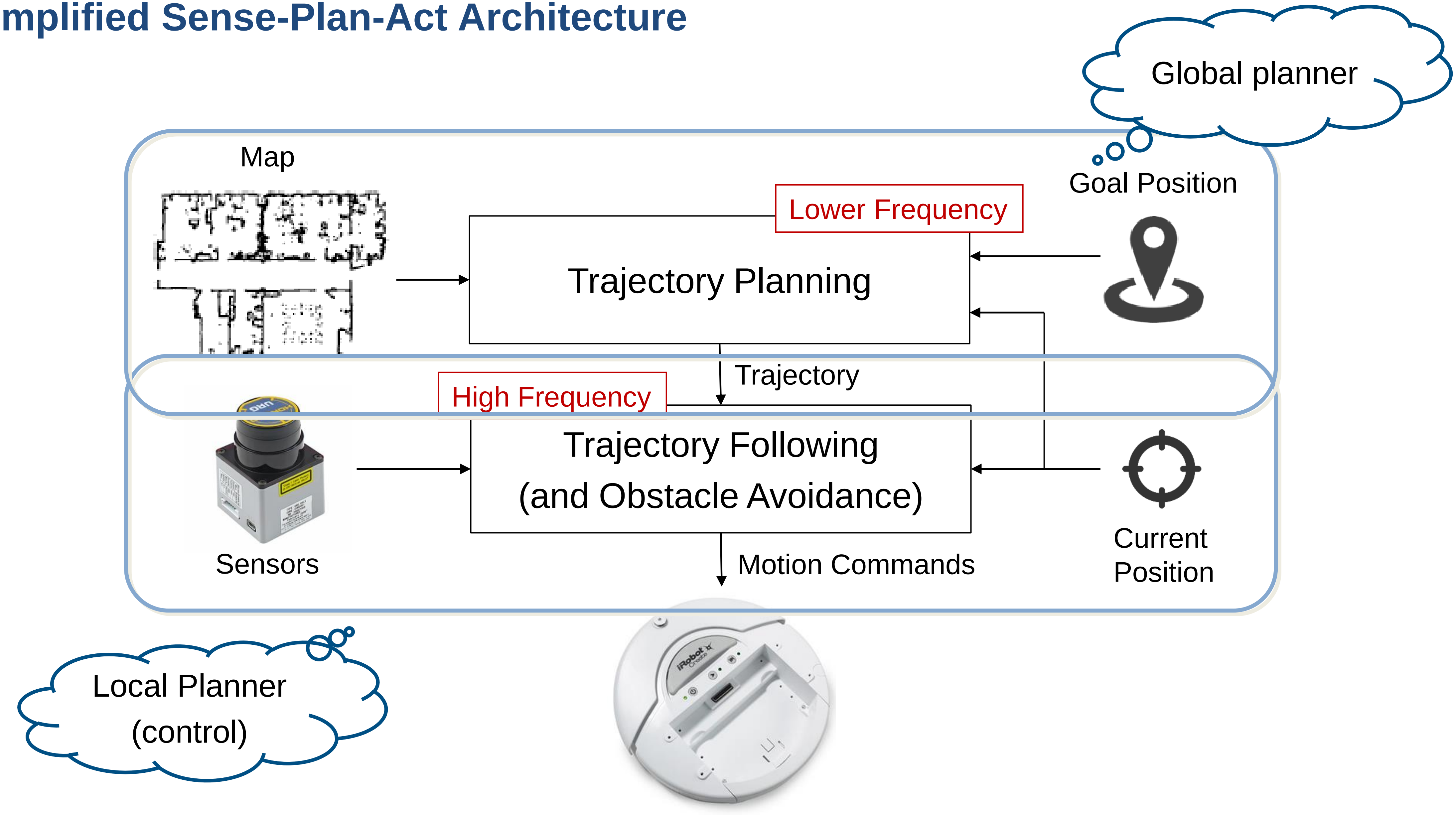
1. Discretely sample robot control space
2. For each sampled velocity, perform forward simulation to predict what would happen if applied for some (short) time.
3. Evaluate (score) each trajectory resulting from the forward simulation
4. Discard illegal trajectories, i.e., those that collide with obstacles, and pick the highest-scoring trajectory



Can handle non circurar trajectories

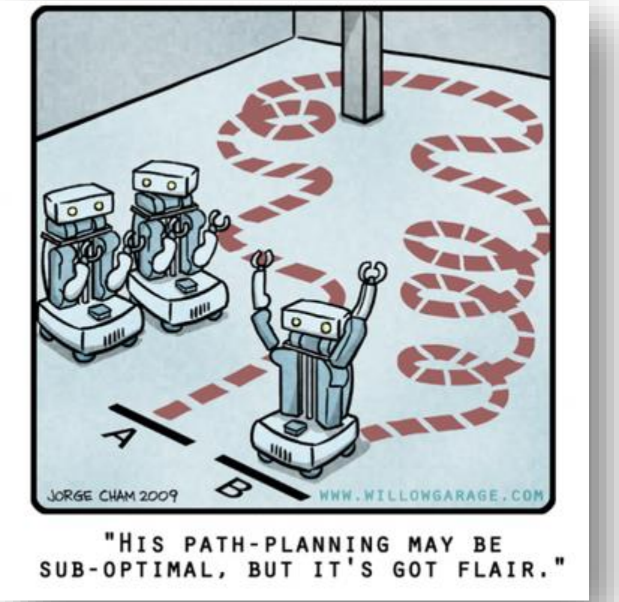
Clothoid: $S(x) = \int_0^x \sin(t^2)\, dt, \quad C(x) = \int_0^x \cos(t^2)\, dt.$

A Simplified Sense-Plan-Act Architecture



Motion planning

> "...eminently necessary since, by definition,
> a robot accomplishes tasks by moving in the real world."
> J.-C. Latombe (1991)



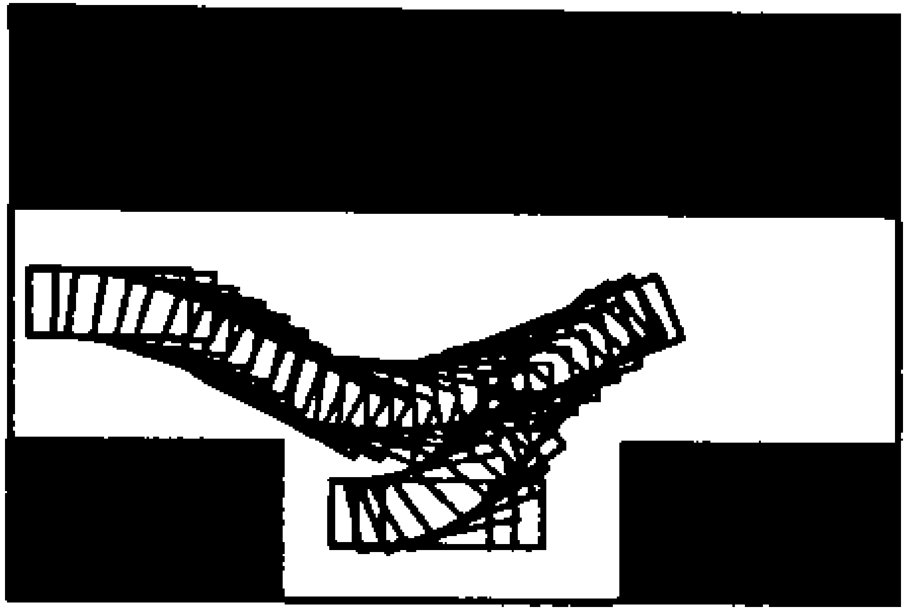
Robot Motion Planning Goals

- Collision-free trajectories
- Robot should reach the goal location as fast as possible (or maximizing an optimality criterion)

Problem statement

Find a collision free path between an initial pose and the goal, taking into account the constraints (geometrical, physical, temporal)

- *Path Planning*: A PATH is a geometric locus of way points, in a given space, where the vehicle must pass
- *Trajectory Generation*: A TRAJECTORY is a path for which a temporal law is specified (e.g., acceleration and velocity at each point)
- *Maneuver Planning*: a MANOUVER is a series of actions or a scheme or plot that the vehicle should execute



Motion planning definition

Given the following notation:

- A: single rigid object (the robot)
- W: Euclidean space where A moves
- B_1, B_2, ..., B_m fixed rigid objects distributed in W (obstacles)

Let assume

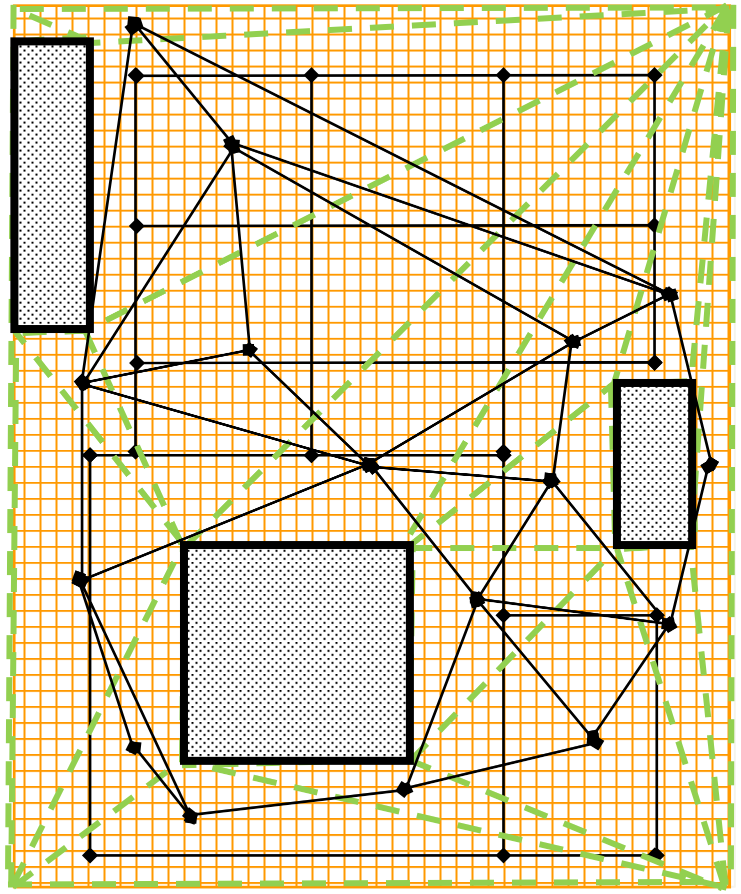
- The geometry of A and B_i is known
- The localization of the B_i in W is accurately known
- There are no kinematic constraints in the motion of A (A is a free-flying object)

Given an initial pose and a goal pose of A in W, generate a continuous sequence of poses of A avoiding contact with the B_i, starting at the initial pose and terminating at the goal pose.

Planning and Maps Representations

Different possible maps representations exist for path planning

- Paths (e.g., probabilistic road maps)
- Free space (e.g., Voronoi diagrams)
- Obstacles (e.g., geometric obstacles)
- Composite (e.g., grid maps)



What a Planner?

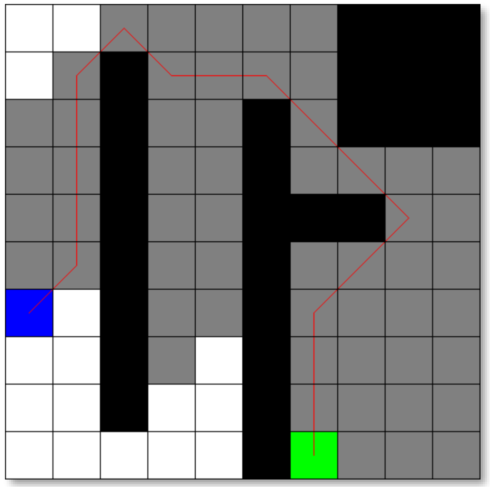
Search Based Planning Algorithms

- A*
- ARA*
- ANA*
- *AD**
- D*
- …

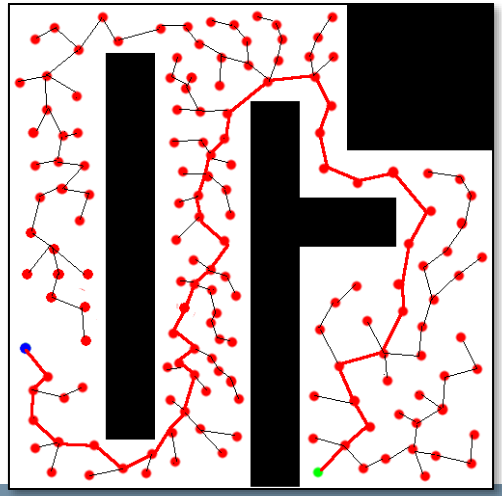
Random Sampling

- PRMs
- RRT
- T-RRT
- SBL
- …

Search Based Planning Library



Open Motion Planning Library



Pros and Cons

	PROS	CONS
Search Based Planning	• Finds the optimal solution • Possible to assign costs • Use of Heuristics • Can state if a solution exists (complete)	• High computational cost
Random Sampling Planning	• Fast in finding a feasible solution	• Hard to assign costs • Only probably complete (cannot be used to test for existance)

Lets have a look at Search Based Methods (SBPL) first because of

- Their simplicity (at least in description)
- The generality of approaches
- Their theoretical guarantees
  (if connectivity assumptions hold)

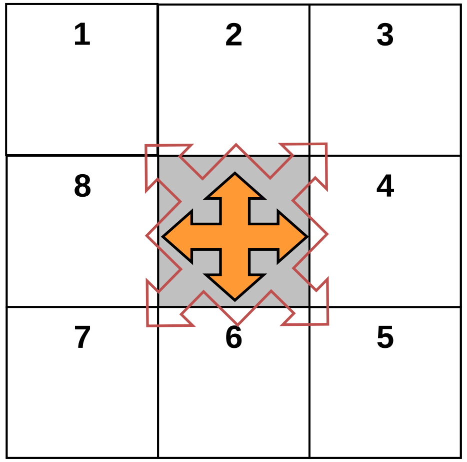
Planning on a Grid

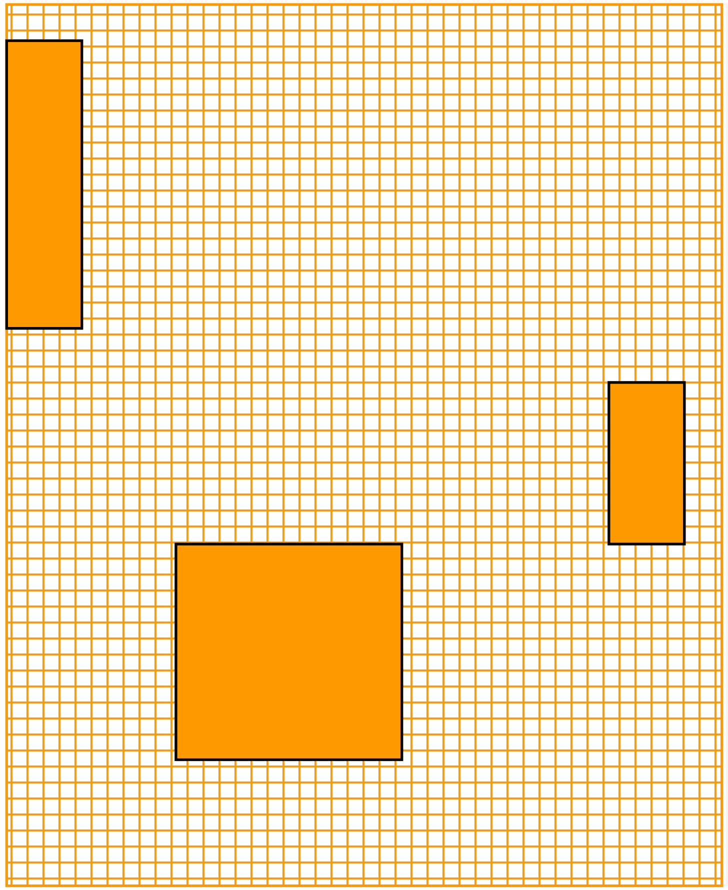
Different possible maps representations exist for path planning

- Paths (e.g., probabilistic road maps)
- Free space (e.g., Voronoi diagrams)
- Obstacles (e.g., geometric obstacles)
- Composite (e.g., grid maps)

Kinematics approximation in grid maps

- 4-orthogonal connectivity
- 4-diagonal connectivity
- 8-connectivity

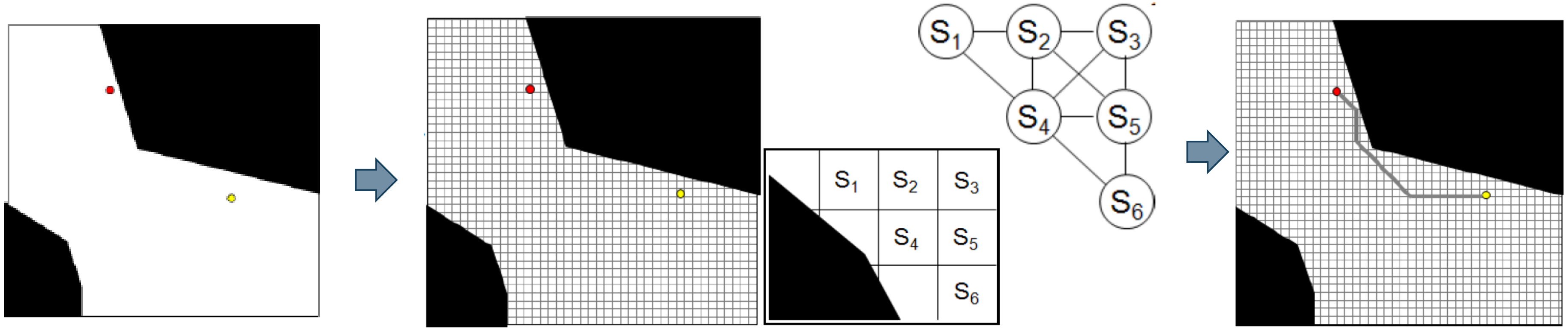




Graph (search) based planning basics

The overall idea:

- Generate a discretized representation of the planning problem
- Build a graph out of this discretized representation (e.g., through 4 neighbors or 8 neighbors connectivity)
- Search the graph for the optimal solution

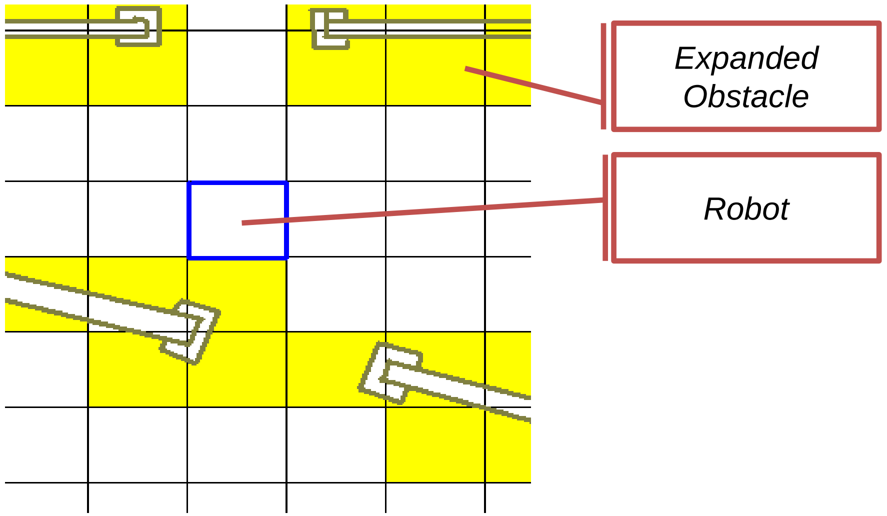


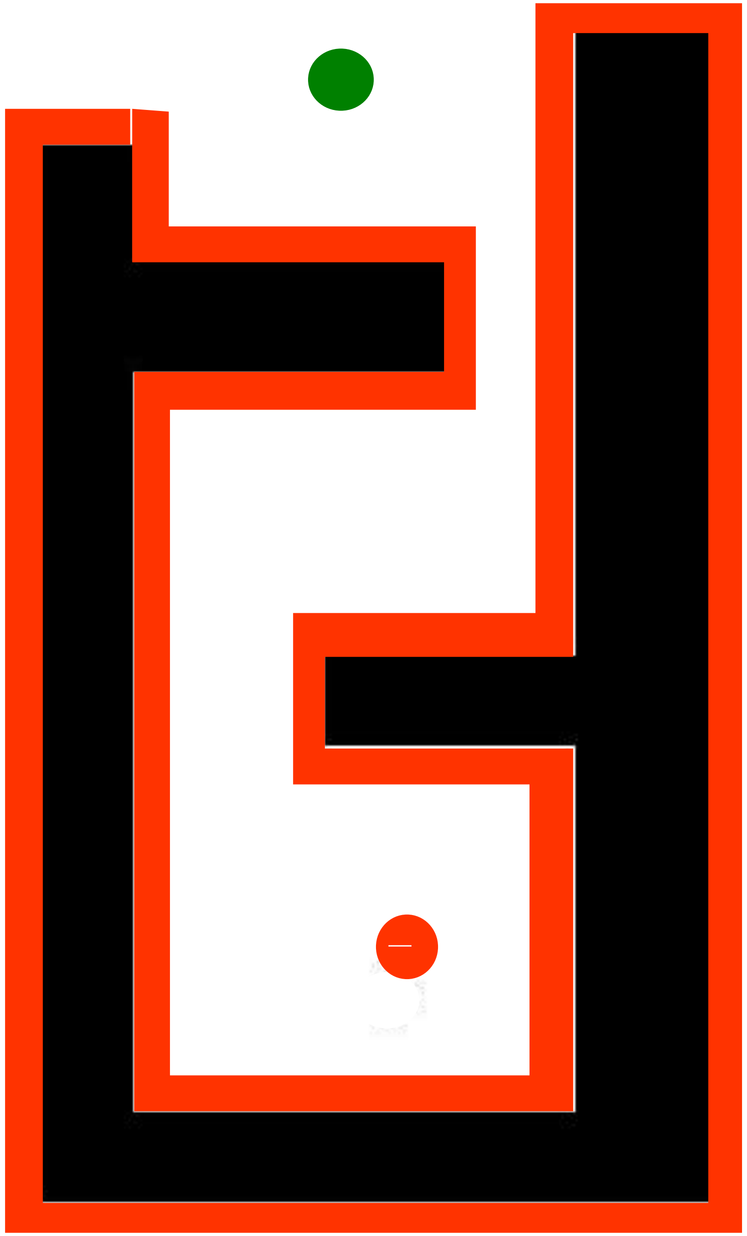
- Can interleave the construction of the representation with the search (i.e., construct only what is necessary)

Robot shape

A real mobile robot should not be modeled as a point; to take into account its shape obstacles are enlarged

This might generate some issues and a trade-off is between memory requirements and performance



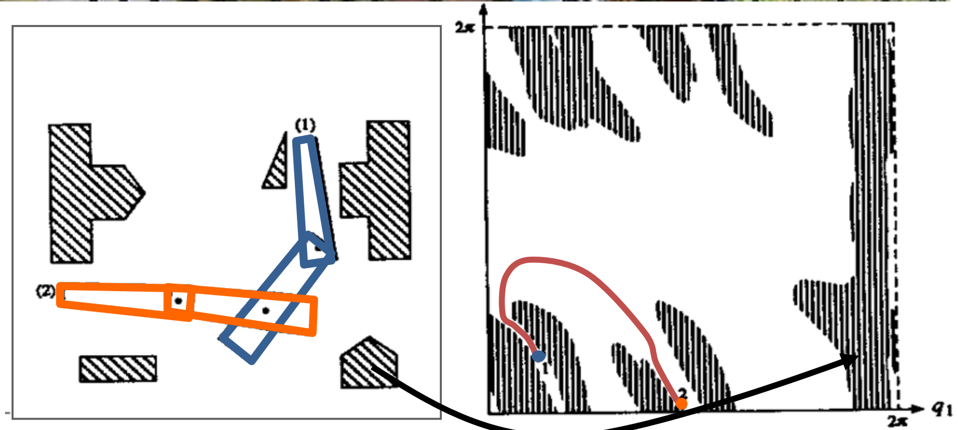


Configuration Space (C-Space)

For an accurate collision detection the Configuration space is used

- A configuration of an object is a point q = (q1, q2,…,qn)
- Point q is free if the robot in q does not collide
- C-obstacle = union of all q where the robot collides
- C-free = union of all free q
- Cspace = C-free + C-obstacle

Planning can be performed in C-Space

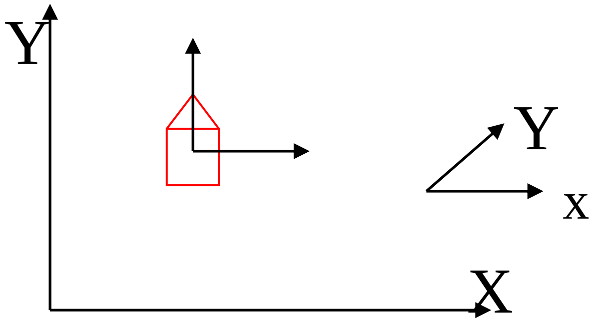


workspace configuration space

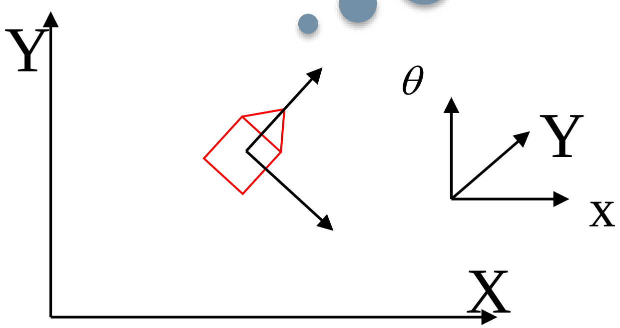
Mobile robot 2D C-Space

Non holonomic constraints can't be $\mathcal{C}$-Space obstacles

A robot can translate in the plane and/or rotate

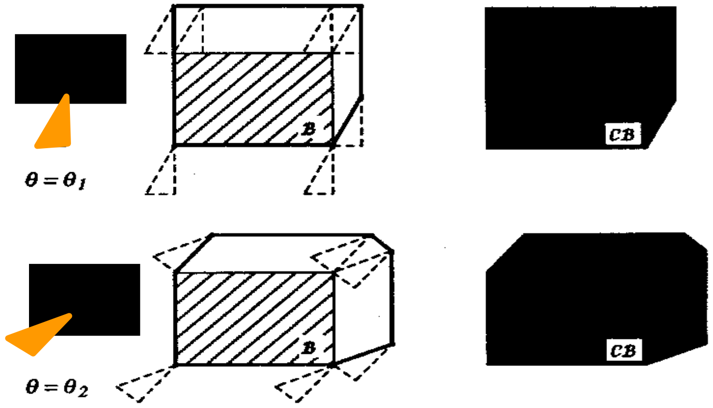


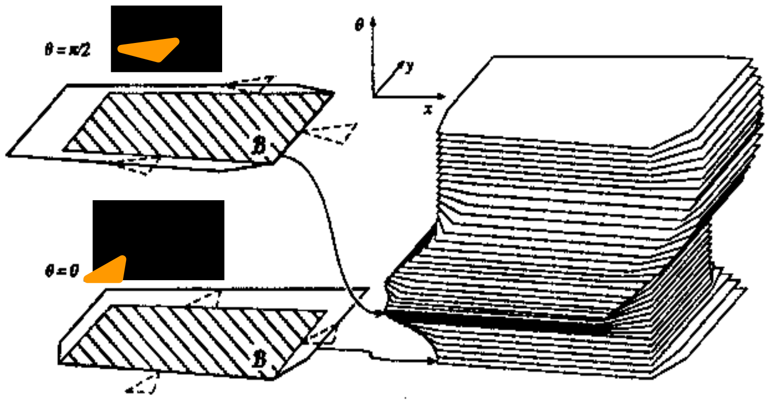
$\mathcal{C}$-space: 2D (x, y)



$\mathcal{C}$-space: 3D (x, y, θ)

Obstacles should be expanded according to the robot orientation

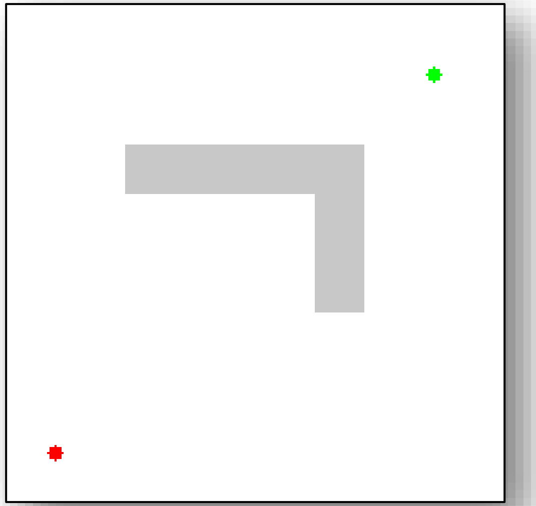




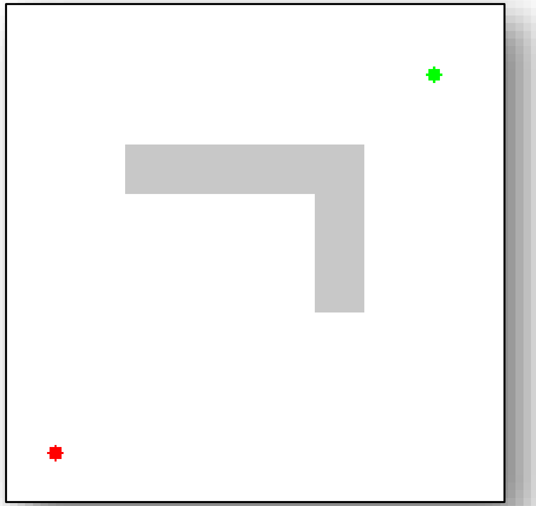
Exact and approximate planning

Different algorithms are available

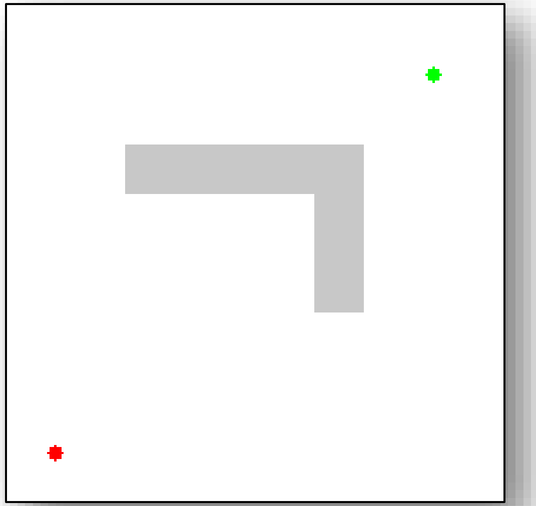
- Returning the optimal path (e.g., Dijstra, A*, …)
- Returning an ε sub-optimal path
  (e.g., weighted A*, ARA*, AD*, R*, D* Lite, ...)



Dijstra



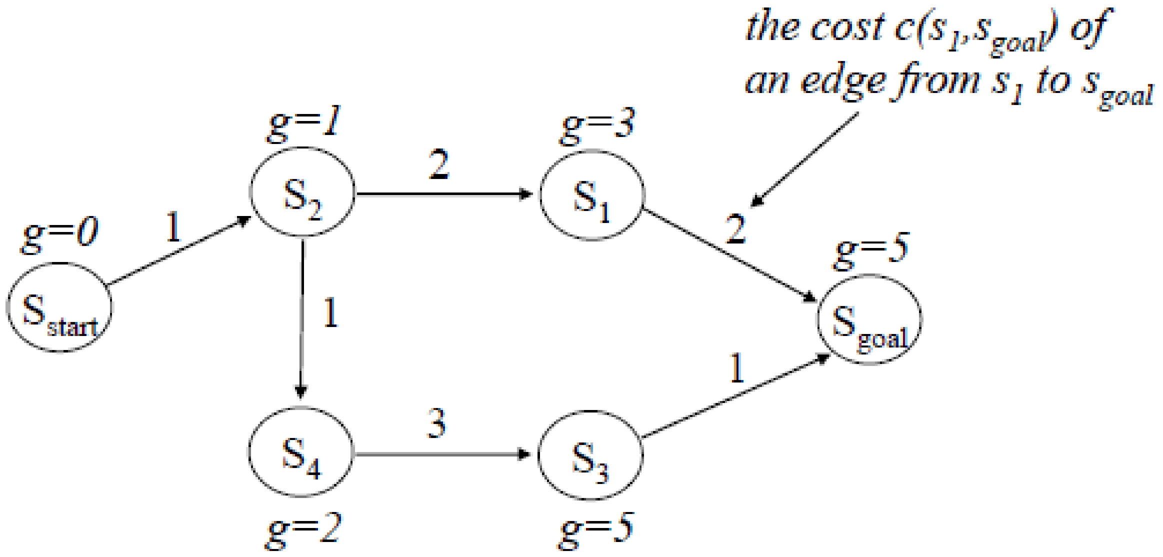
A*



weighted A*

Searching graphs for least cost path

Given a graph search for the path that minimizes costs as much as possible

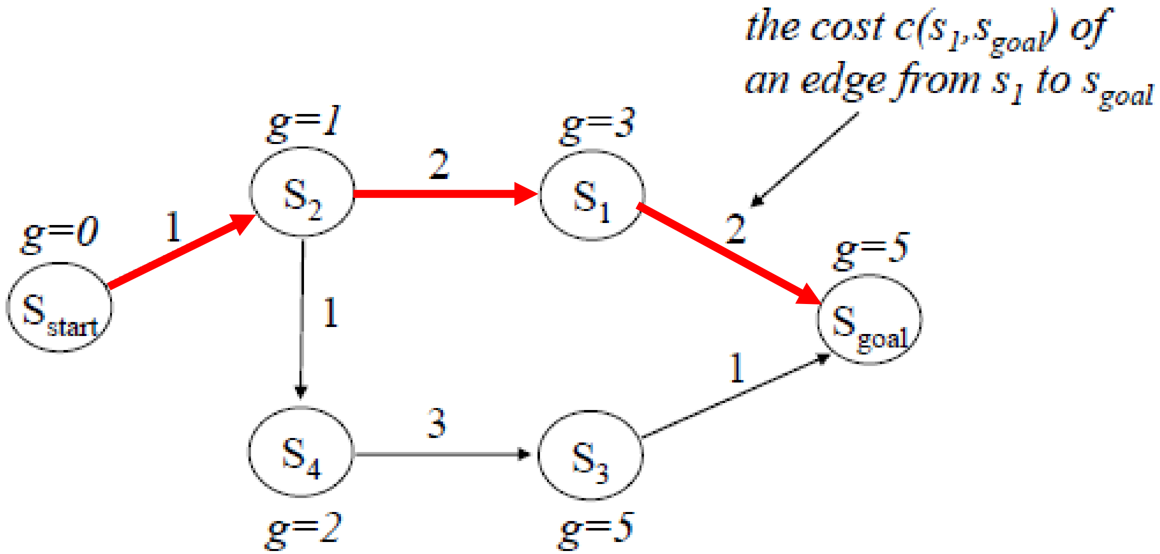


Many search algorithms compute optimal g-values for relevant states

- $g(s)$–an estimate of the cost of a least-cost path from s_{start} to s
- optimal values satisfy: $g(s) = \min_{s'' \text{ in } pred(s)} g(s'') + c(s'',s)$

Searching graphs for least cost path

Given a graph search for the path that minimizes costs as much as possible



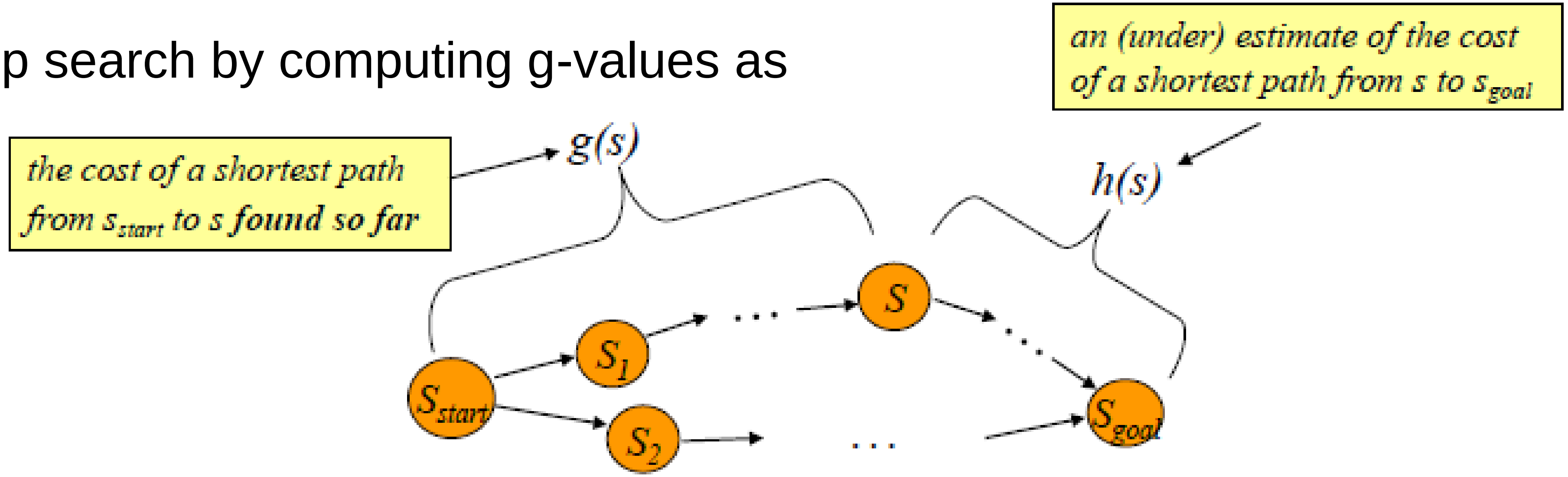
Least-cost path is a greedy path computed by backtracking:

- start with s_{goal} and from any state s move to the predecessor state s' such that

$$s' = \text{argmin}_{s'' \text{ in pred}(s)} (g(s'')+c(s'',s))$$

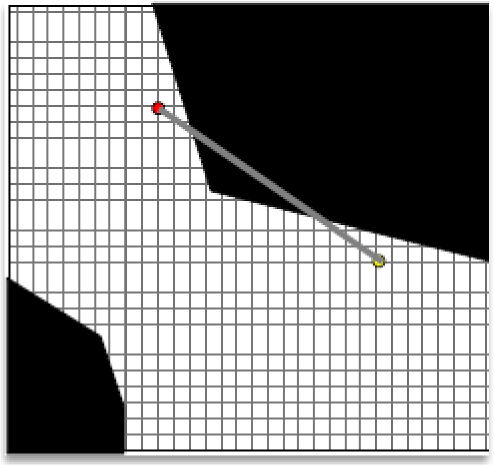
A* search algorithm

A* speeds up search by computing g-values as



Heuristic function must be

- Admissible: for every state s, $h(s) \leq c^*(s,s_{goal})$
- Consistent (satisfy triangle inequality):
  - $h(s_{goal},s_{goal}) = 0$
  - for every $s \neq s_{goal}$, $h(s) \leq c(s,succ(s)) + h(succ(s))$



Admissibility follows from consistency and often viceversa

A* Search Algorithm

Main function

- $g(s_{start}) = 0$; all other g-values are infinite;
- $OPEN = \{s_{start}\}$;
- ComputePath();

Set of candidates for expansion

ComputePath function

For every expanded state g(s) is optimal
(if heuristics are consistent)

- while(s_{goal} is not expanded)
  - remove s with the smallest $[f(s) = g(s)+h(s)]$ from $OPEN$;
  - expand s;

A* Search Algorithm

Main function

- $g(s_{start}) = 0$; all other g-values are infinite;
- $OPEN = \{s_{start}\}$;

Set of candidates for expansion

- ComputePath();

ComputePath function

Set of states already expanded

- while(s_{goal} is not expanded)
  - remove s with the smallest [f(s) = g(s)+h(s)]from OPEN;
  - insert s into CLOSED;
  - for every successor s'of s such that s' not in CLOSED
    - if g(s') > g(s) + c(s,s')
      - g(s') = g(s) + c(s,s');
      - insert s' into OPEN;

Tries to decrease g(s') using the found path from s_{start} to s

A* Search Algorithm

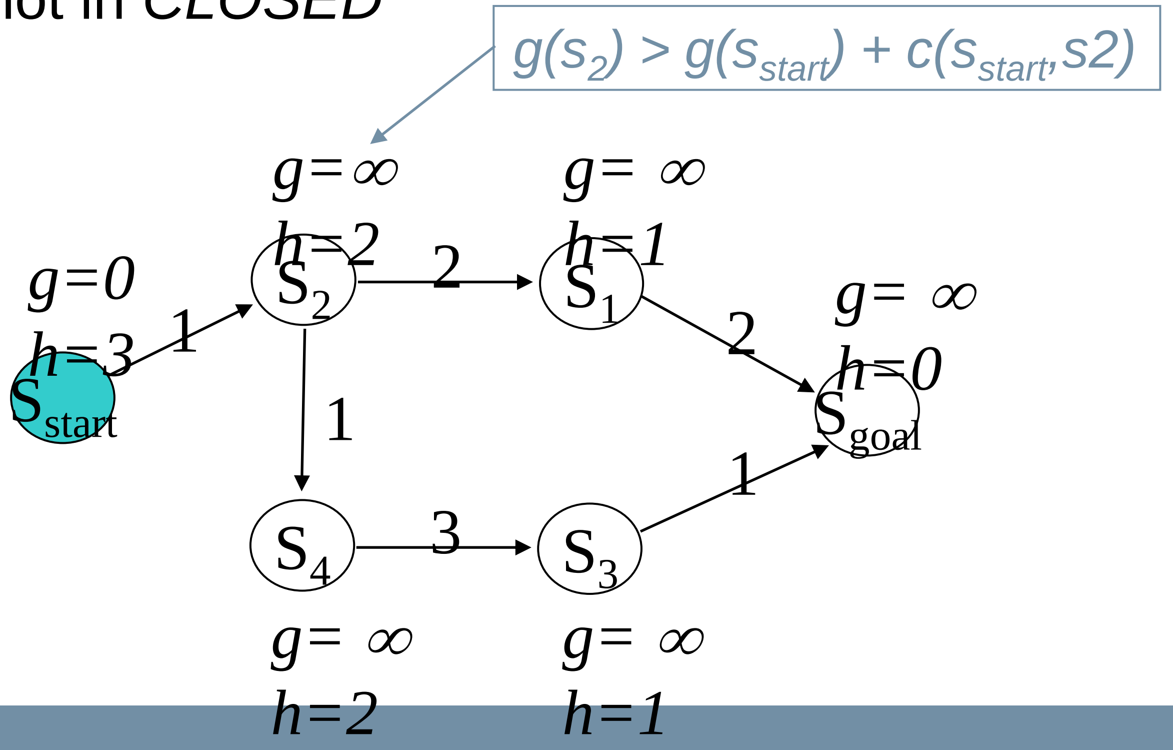
ComputePath function

- while(s_{goal} is not expanded)
- remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
- insert s into *CLOSED*;
- for every successor s' of s such that s' not in *CLOSED*
    - if $g(s') > g(s) + c(s,s')$
        - $g(s') = g(s) + c(s,s')$;
        - insert s' into *OPEN*;

CLOSED = {}

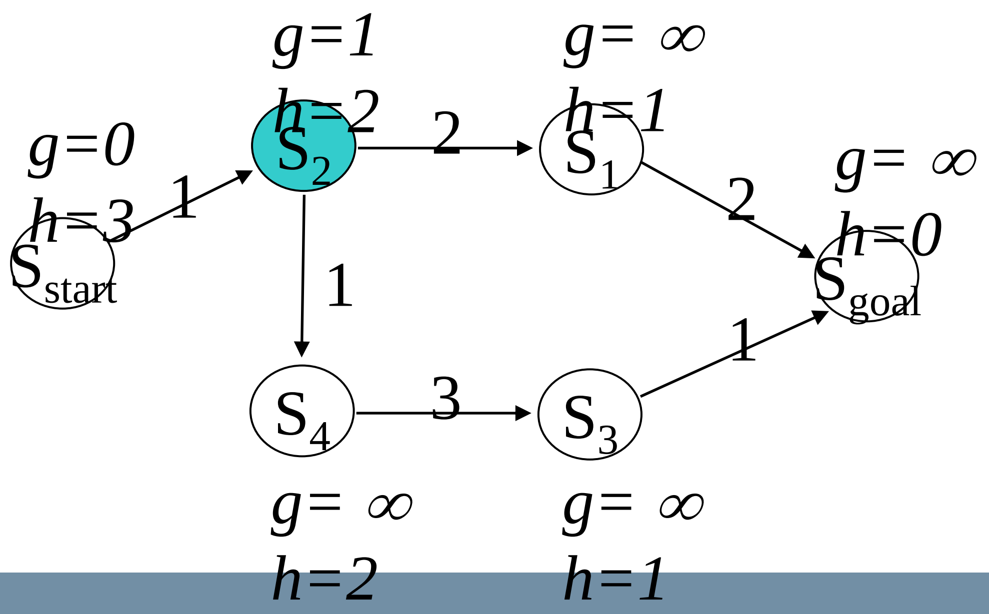
OPEN = $\{s_{start}\}$

next state to expand: s_{start}



ComputePath function

- while(s_{goal} is not expanded)
- remove *s* with the smallest *[f(s) = g(s)+h(s)]*from *OPEN*;
- insert *s* into *CLOSED*;
- for every successor *s'*of *s* such that *s'* not in *CLOSED*
    - if $g(s') > g(s) + c(s,s')$
        - $g(s') = g(s) + c(s,s');$
        - insert *s'* into *OPEN*;

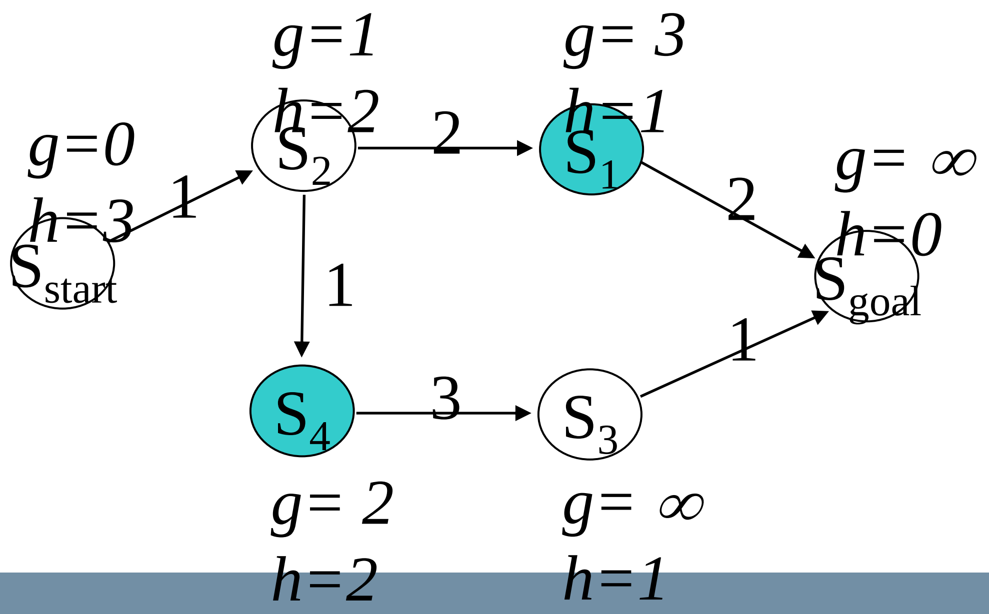


CLOSED = $\{s_{start}\}$
OPEN = $\{s_2\}$
next state to expand: s_2

ComputePath function

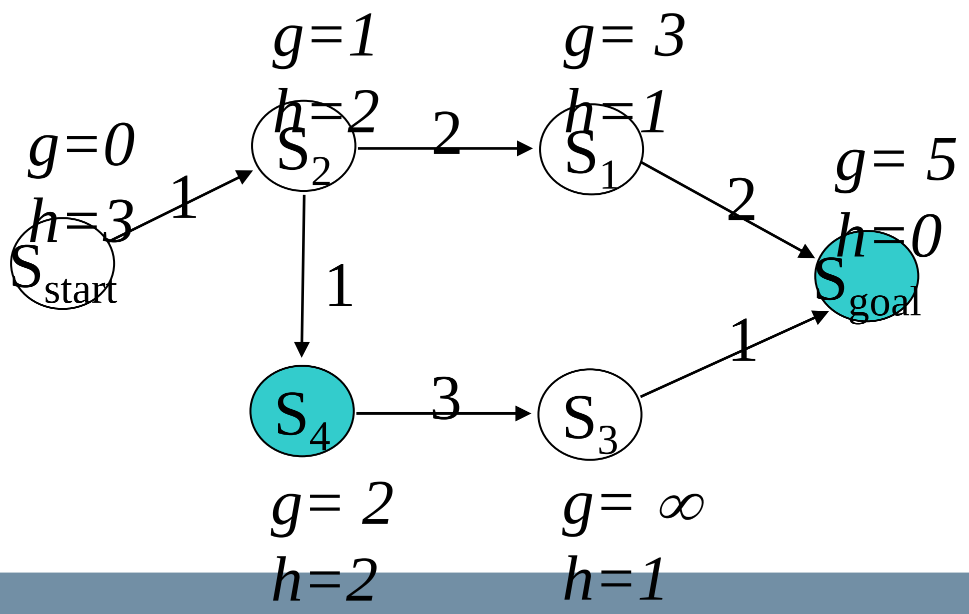
- while(s_{goal} is not expanded)
- remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
- insert s into *CLOSED*;
- for every successor s' of s such that s' not in *CLOSED*
    - if $g(s') > g(s) + c(s,s')$
        - $g(s') = g(s) + c(s,s');$
        - insert s' into *OPEN*;

CLOSED = $\{s_{start}, s_2\}$
OPEN = $\{s_1,s_4\}$
next state to expand: s_1



ComputePath function

- while(s_{goal} is not expanded)
- remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
- insert s into *CLOSED*;
- for every successor s' of s such that s' not in *CLOSED*
    - if $g(s') > g(s) + c(s,s')$
        - $g(s') = g(s) + c(s,s')$;
        - insert s' into *OPEN*;

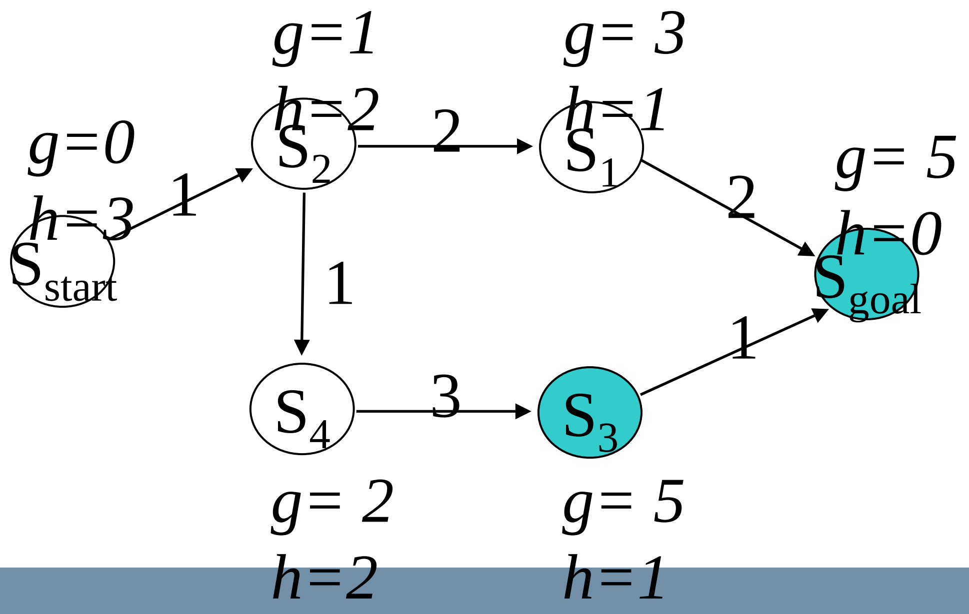


CLOSED = $\{s_{start}, s_2, s_1\}$
OPEN = $\{s_4, s_{goal}\}$
next state to expand: s_4

A* Search Algorithm

ComputePath function

- while(s_{goal} is not expanded)
- remove s with the smallest $[f(s) = g(s)+h(s)]$ from *OPEN*;
- insert s into *CLOSED*;
- for every successor s' of s such that s' not in *CLOSED*
    - if $g(s') > g(s) + c(s,s')$
        - $g(s') = g(s) + c(s,s');$
        - insert s' into *OPEN*;



$CLOSED = \{s_{start}, s_2, s_1, s_4\}$
$OPEN = \{s_3, s_{goal}\}$
next state to expand: s_{goal}

A* Search Algorithm

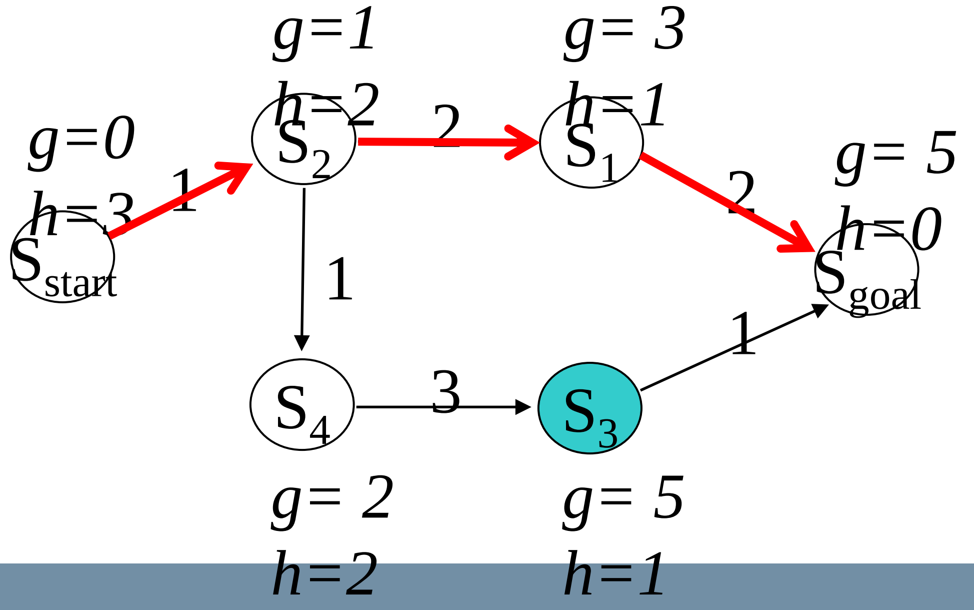
ComputePath function

- while(s_{goal} is not expanded)
- remove *s* with the smallest *[f(s) = g(s)+h(s)]*from *OPEN*;
- insert *s* into *CLOSED*;
- for every successor *s'*of *s* such that *s'* not in *CLOSED*
  - if *g(s') > g(s) + c(s,s')*
    - *g(s') = g(s) + c(s,s');*
    - insert *s'* into *OPEN*;

CLOSED = $\{s_{start}, s_2, s_1, s_4, s_{goal}\}$

OPEN = $\{s_3\}$

<u>*DONE!*</u>



A* Properties

A* is guaranteed to

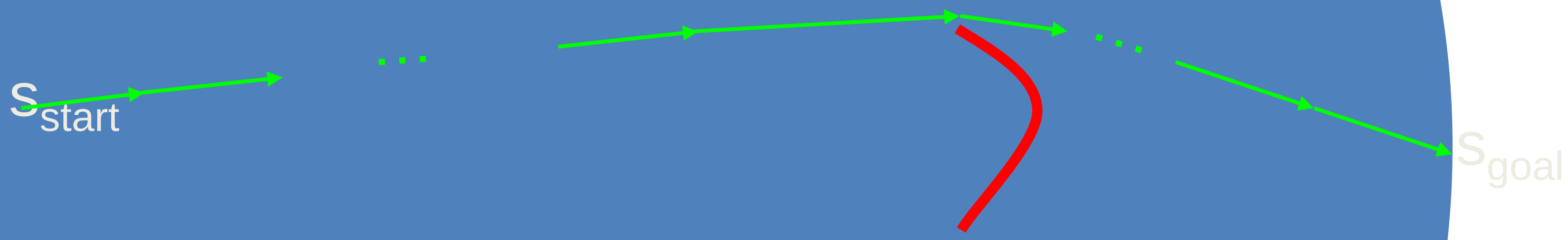
- Return an optimal path in terms of the solution
- Perform provably minimal number of state expansions

Algorithms state expansion:

- Dijkstra's: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values
- Weighted A***:**expands states in the order of $f = g + \varepsilon\, h$ values,
  $\varepsilon > 1$= bias towards states that are closer to goal

Weighted A* Search in many domains, it has been shown to be orders of magnitude faster than A*

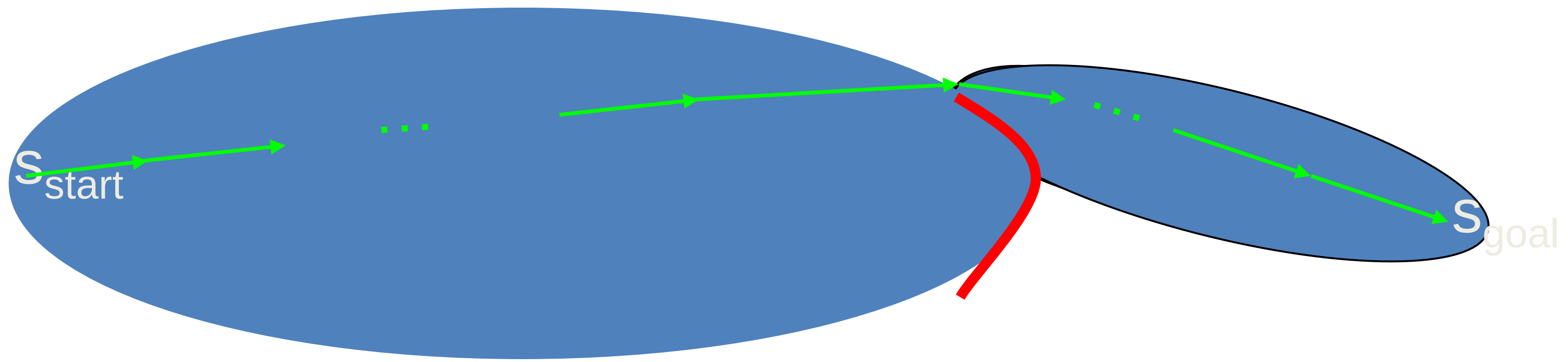
J. Pearl., "Heuristics: Intelligent Search Strategies for Computer Problem Solving", *Addison-Wesley*, 1984 ughly)



A* Properties

Algorithms state expansion:

- Dijkstra's: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values

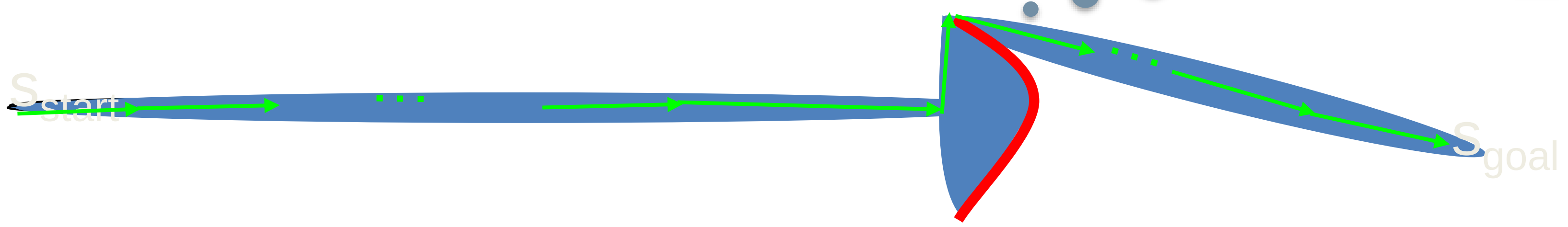


A* Properties

Algorithms state expansion:

- Dijkstra's: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values
- Weighted A*:expands states in the order of $f = g + \varepsilon h$ values, $\varepsilon > 1$= bias towards states that are closer to goal

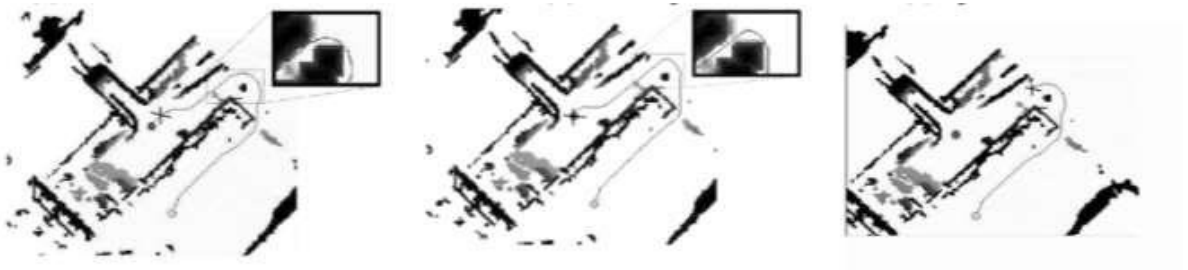
Shallow minima help in finding solution fast.



Other variations of A*

ARA* (Anytime Repairing A*)

- Subsequent queries with decreasing suboptimality factor ϵ
- Fast initial (suboptimal) solution
- Refinement over time



A*: 25s   ARA* (ϵ=2.5): 0.6s.   ARA* (ϵ=1.0): 25s.

Likhachev, M. (2003). "ARA*: Anytime A* with provable bounds on sub-optimality", *Advances in Neural Information Processing Systems*

D*/D*-Light

- Re-use parts of the previous query and only repair solution locally where changes occured

Anytime D* (D* + ARA*)

- Anytime graph-search re-using previous query

Planning problem ingredients

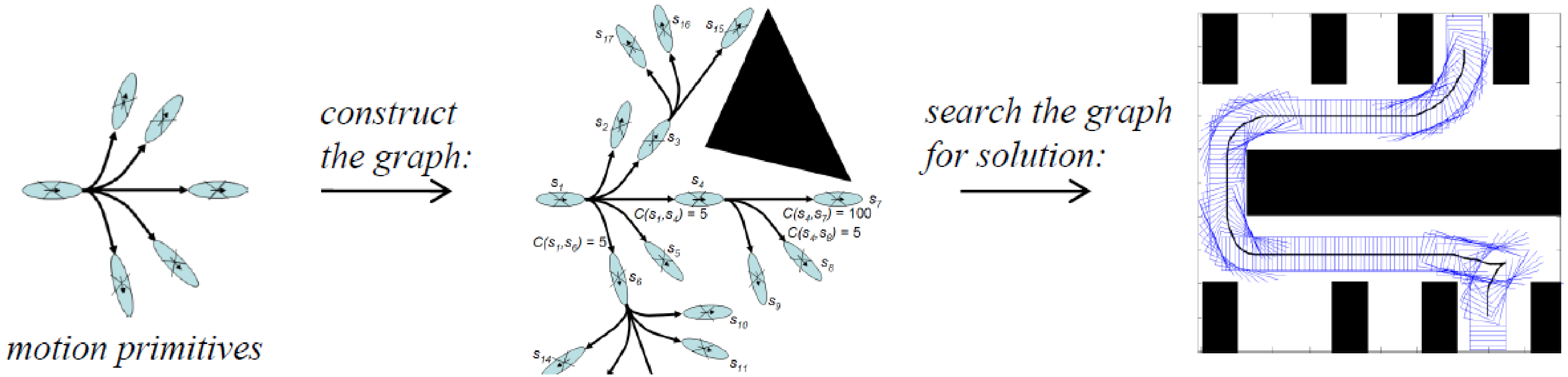
Typical components of a Search-based Planner

- Graph construction (given a state what are its successor states)
- Cost function (a cost associated with every transition in the graph)
- Heuristic function (estimates of cost-to-goal)
- Graph search algorithm (for example, A* search)

Domain dependent

Domain independent

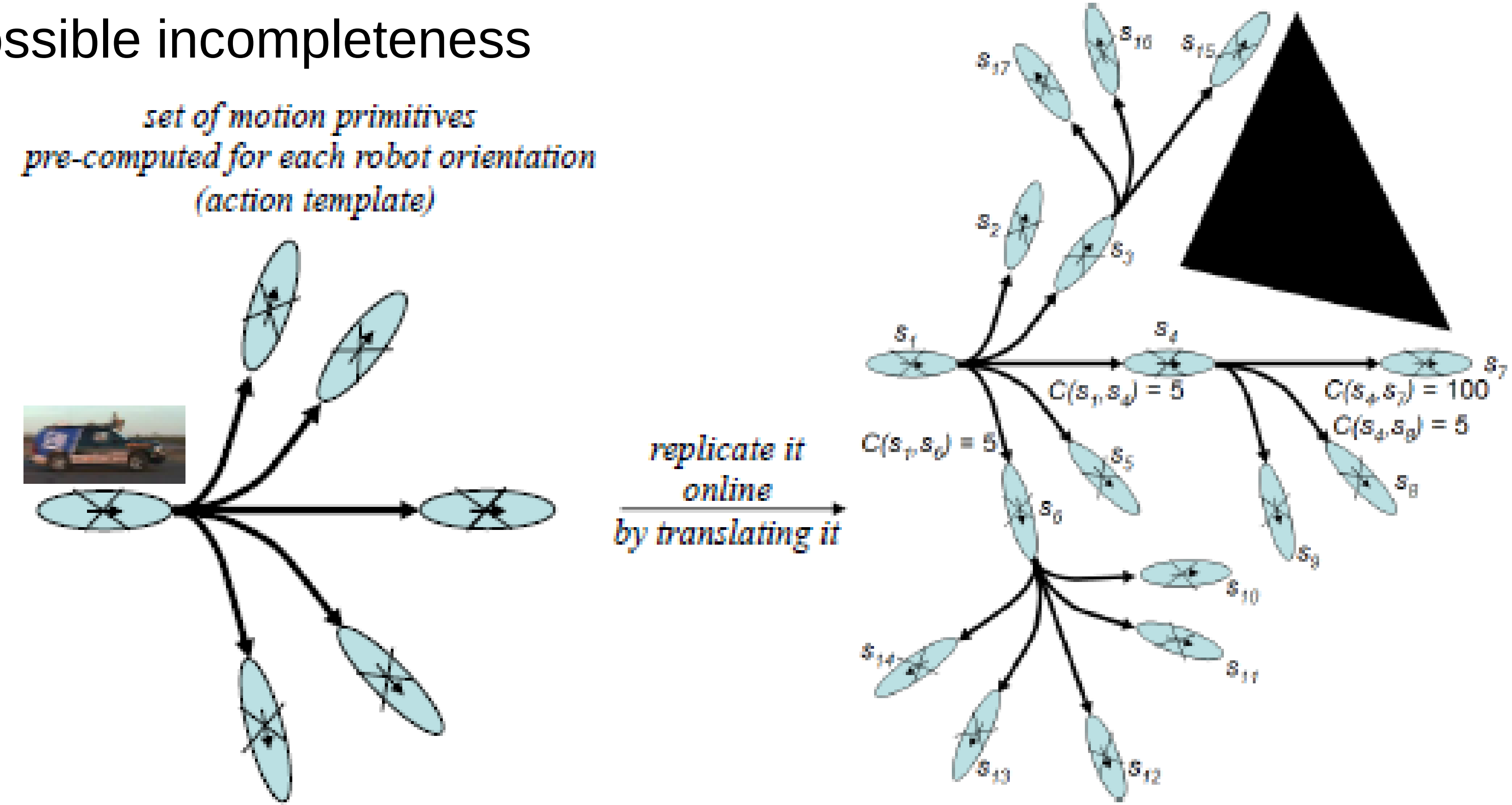
The graph can be built taking into account robot dynamics/kinematics constraints



Planning with graphs

Graph can be constructed by using motion primitives

- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness



Planning with graphs

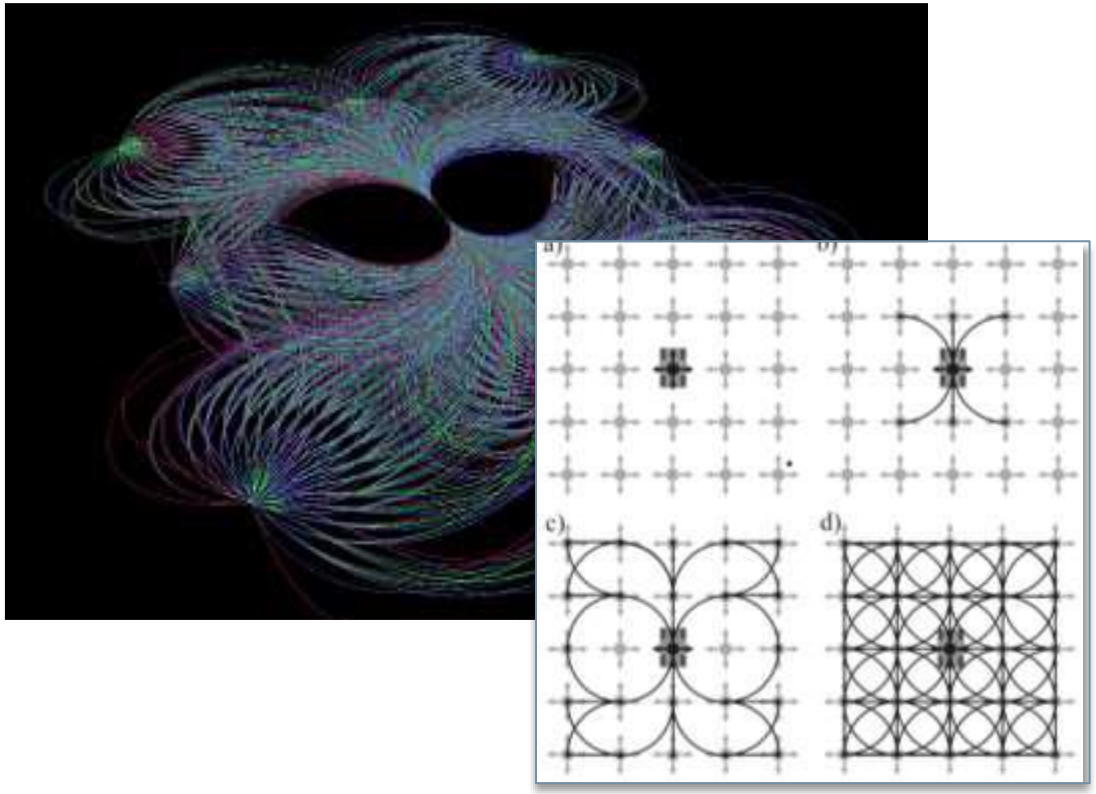
Graph can be constructed by using motion primitives

- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness

*planning on 4D (<x,y,orientation,velocity>) multi-resolution lattice using Anytime D**
[Likhachev & Ferguson, '09]

Motion planning for constrained platforms as a graph search in state-space

- Discretize state-space into a hypergrid (e.g. (x, y, θ, κ))
- Compute neighborhood set by connecting each tuple of states with feasible motions
- Define cost-function/edge-weights
- Run any graph-search algorithm to find lowest-cost path



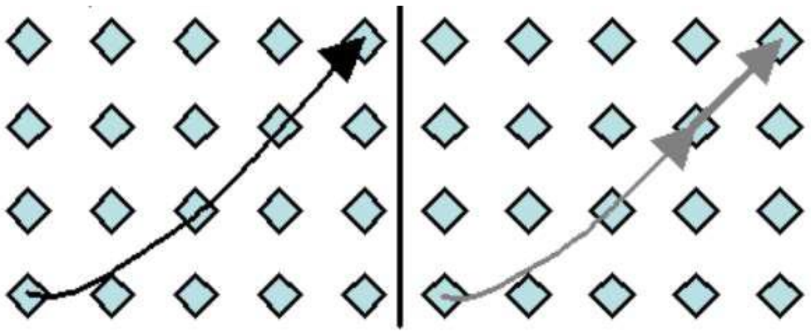
Pivtoraiko, M., & Kelly, A. (2005). Constrained Motion Planning in Discrete State Spaces. In *Field and Service Robotics pp.* 269–280.

State-Lattice planning pros and cons

Pros	Cons
Resolution complete	"Curse of dimensionality". Number of states grows exponentially with dimensionality of state-space
Optimal	State-lattice construction requires solving nontrivial two-state boundary value problem
Offline computations due to regular structure possible	Regular discretization might cause problems in narrow passages, not aligned with the hypergrid
	Discretization causes discontinuities in state variables not considered in the hypergrid thus motion plans are not inherently executable

Design minimal neighborhood sets

- Avoid insertion of edges that can be decomposed with the existing control
- Decomposition "close" in cost-space

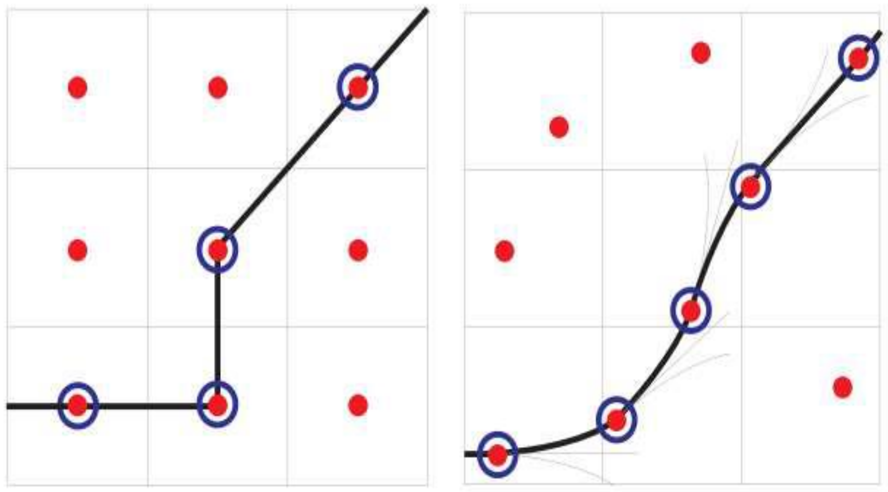


Pivtoraiko, M., & Kelly, A. (2005). Constrained Motion Planning in Discrete State Spaces. In *Field and Service Robotics pp.* 269–280.

Hybrid A*

Generate motion primitives by sampling control space

- No need to solve boundary value problem
- Resulting continuous states are associated with a discrete state in the hypergrid
- Each grid-cell stores a continuous state



Conventional Stale-Lattice

Hybrid A*

No completeness guarantee any more

- Changing reachable statespace
- Pruning of continuous-state branches

Produces inherently driveable paths and above mentioned shortcomings almost never happen in practice

Dolgov, D., et al., “Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments”, *The Int. Journal of Robotics Research*, 2010

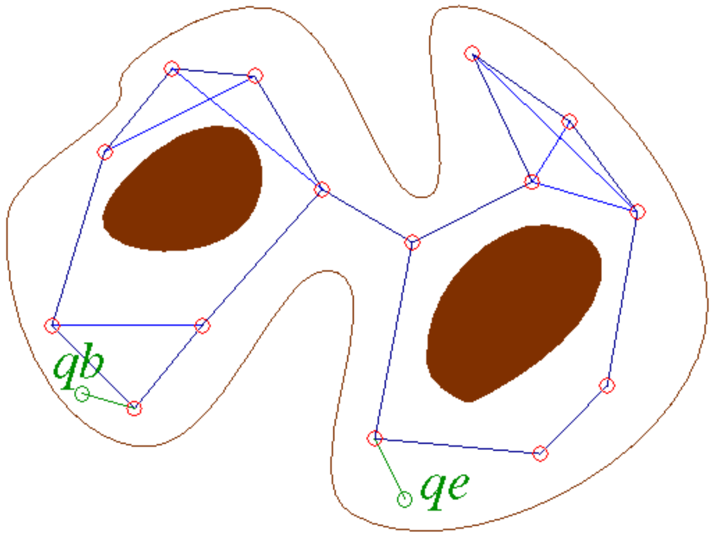
Sampling-Based Motion Planning

Other motivations for sampling:

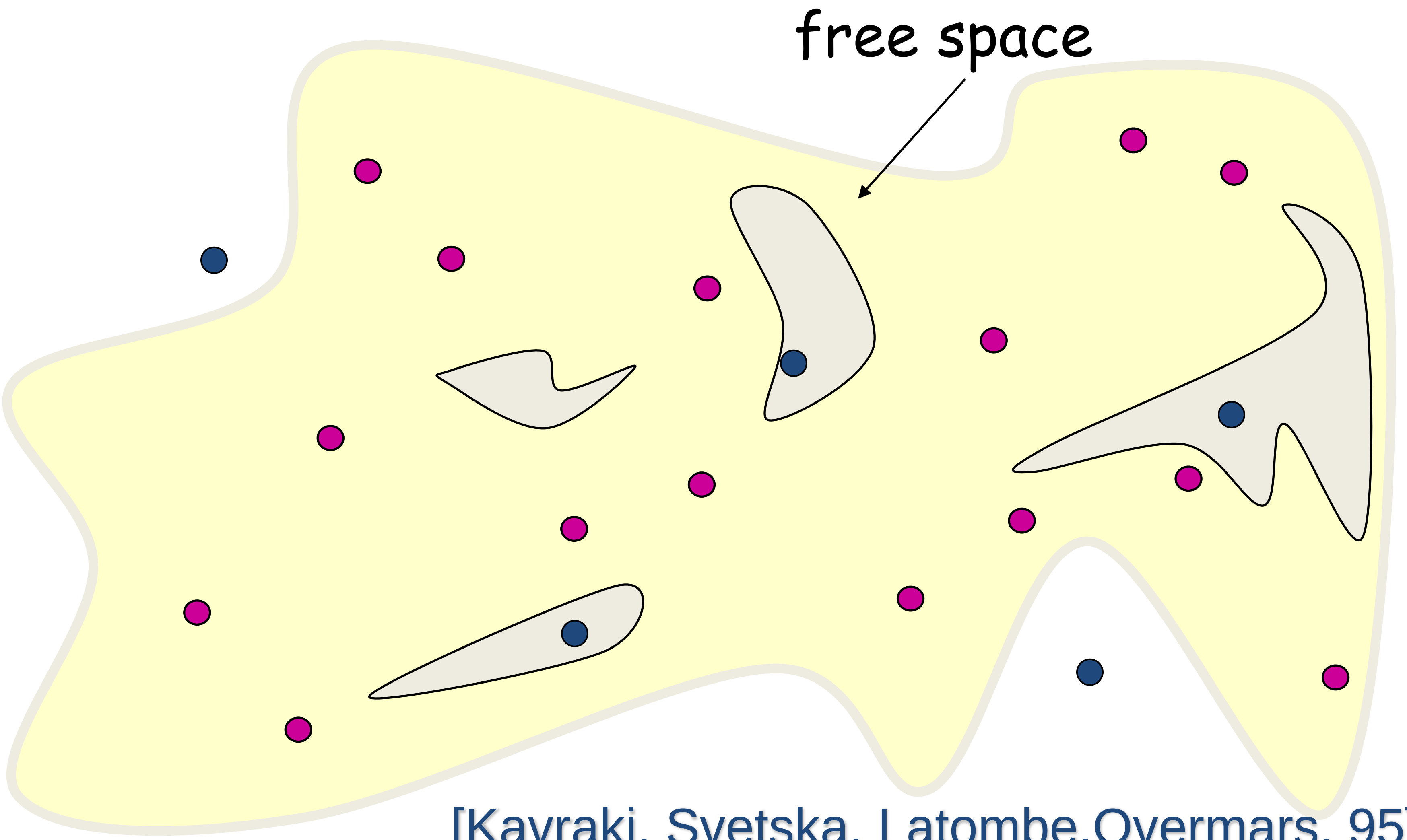
- Computing an explicit representation of collision-free space is extremely time consuming and impractical
- Conversely checking if a position is in free space is fast. There exist fast collision-checking algorithms to test whether any given configuration (or short path) is collision-free or not, in less than 0.001 sec

Basic idea :

- Sample the space of interest
- Connect sampled points by simple paths
- Check if the path is collision free
- Search the resulting graph

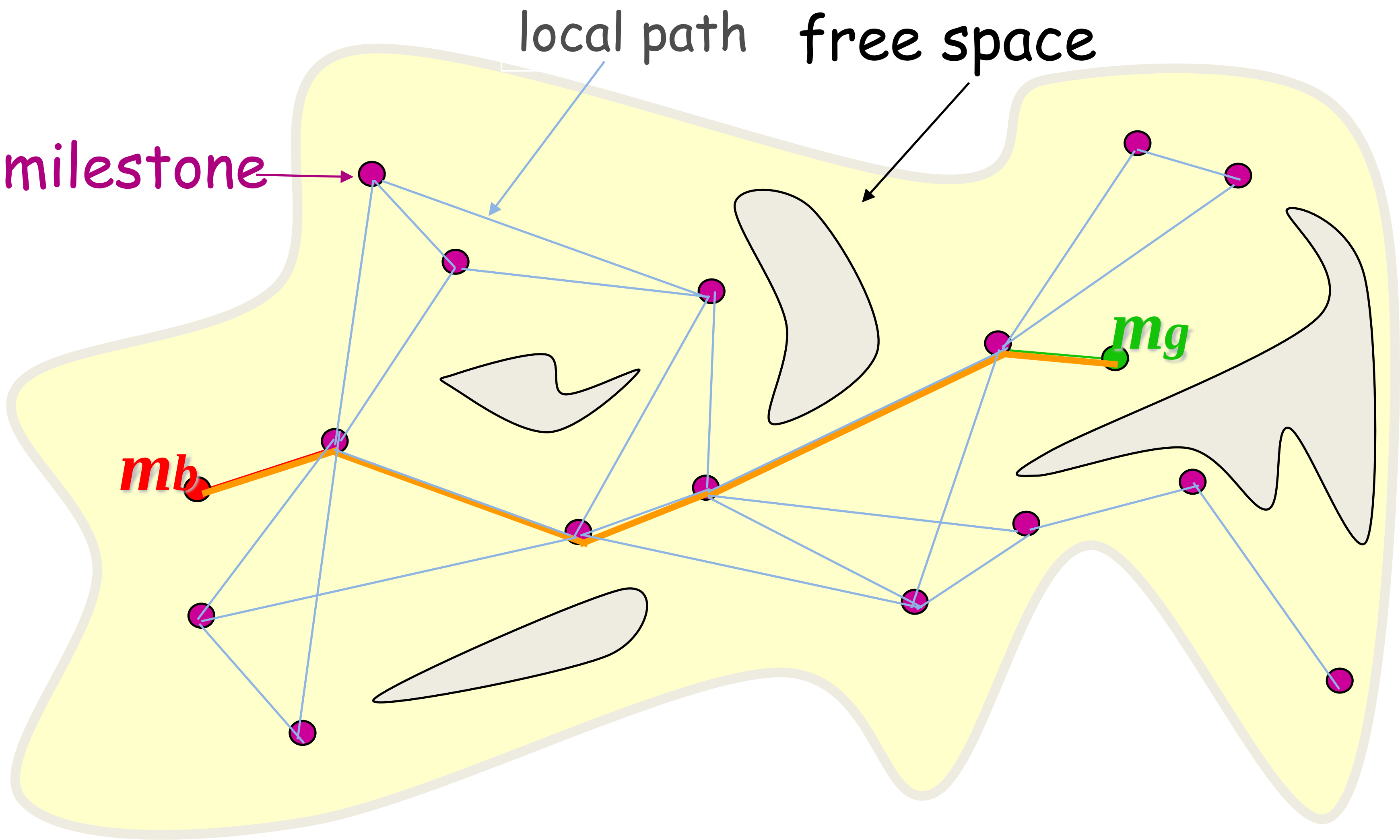


Probabilistic Roadmap (PRM)



[Kavraki, Svetska, Latombe,Overmars, 95]

Probabilistic Roadmap (PRM)



PRM Algorithms

Build roadmap

- Pick uniformly at random s configurations in F and create M, the set of milestones
- Construct the roadmap, i.e., a graph R=(M, L), where L is every pair of milestones that see each other
- Call R the roadmap

PRM Algorithms

Query the graph

- for i={b, e} do

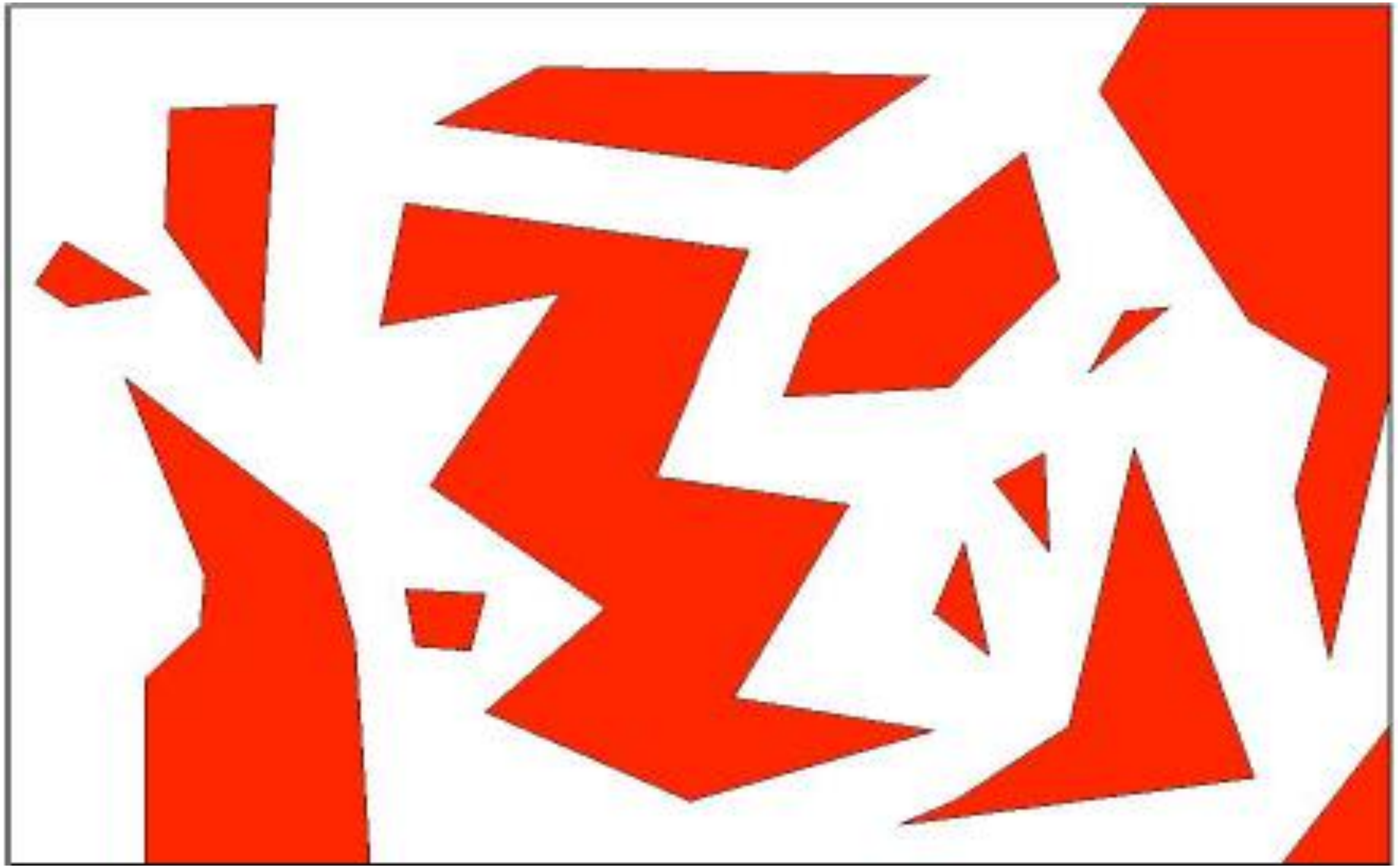
  if there is a milestone m that sees qi then

  mi <= m

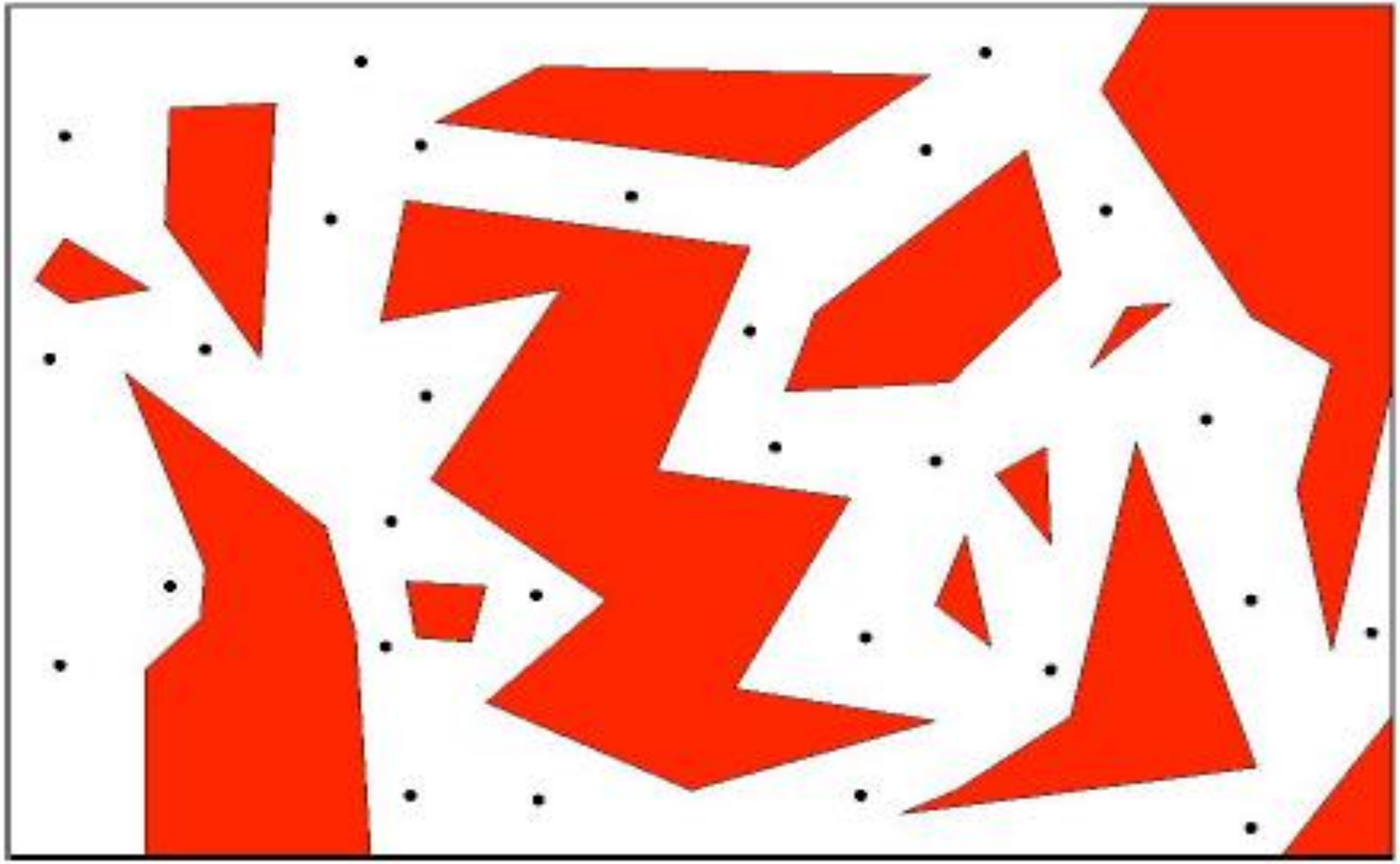
  else

  i. repeat t times: pick a configuration q in F at random near qi until q sees both qi and a milestone m
  ii. if all t trials fail, return FAILURE, else mi<-m
- if m_b and m_e are in the same connected component of the roadmap then return a path connecting them else return NO PATH

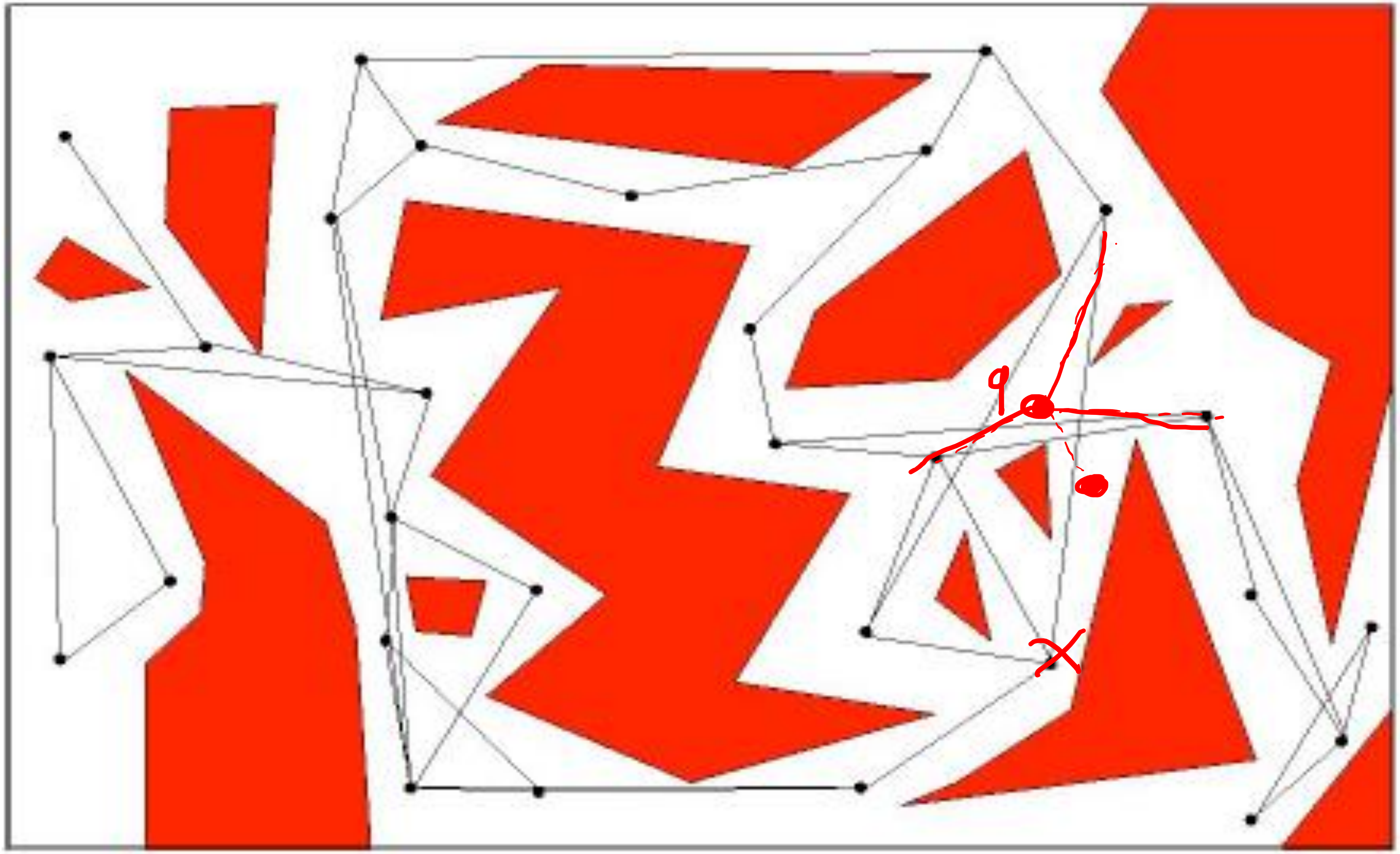
PRM in Short



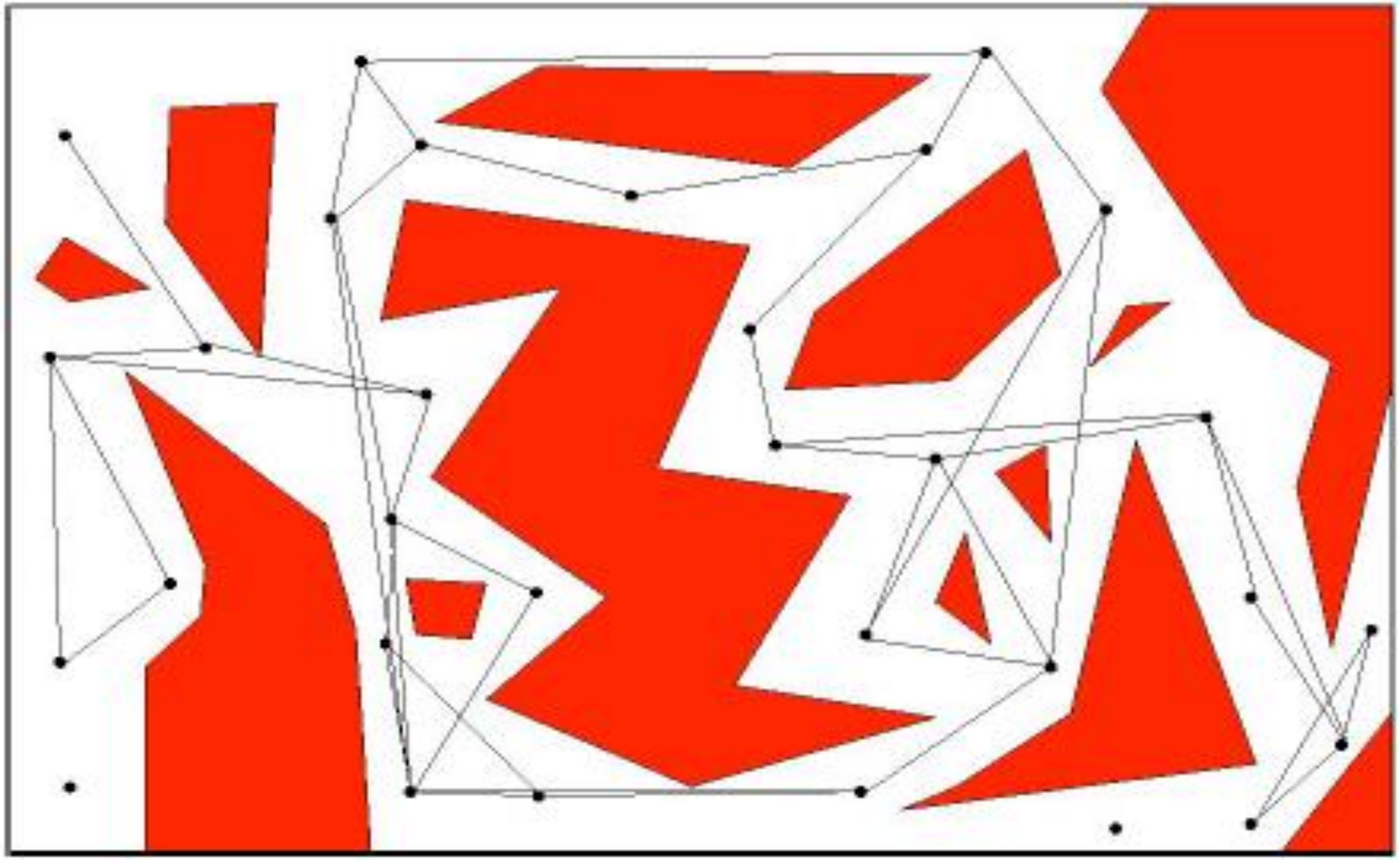
PRM in Short



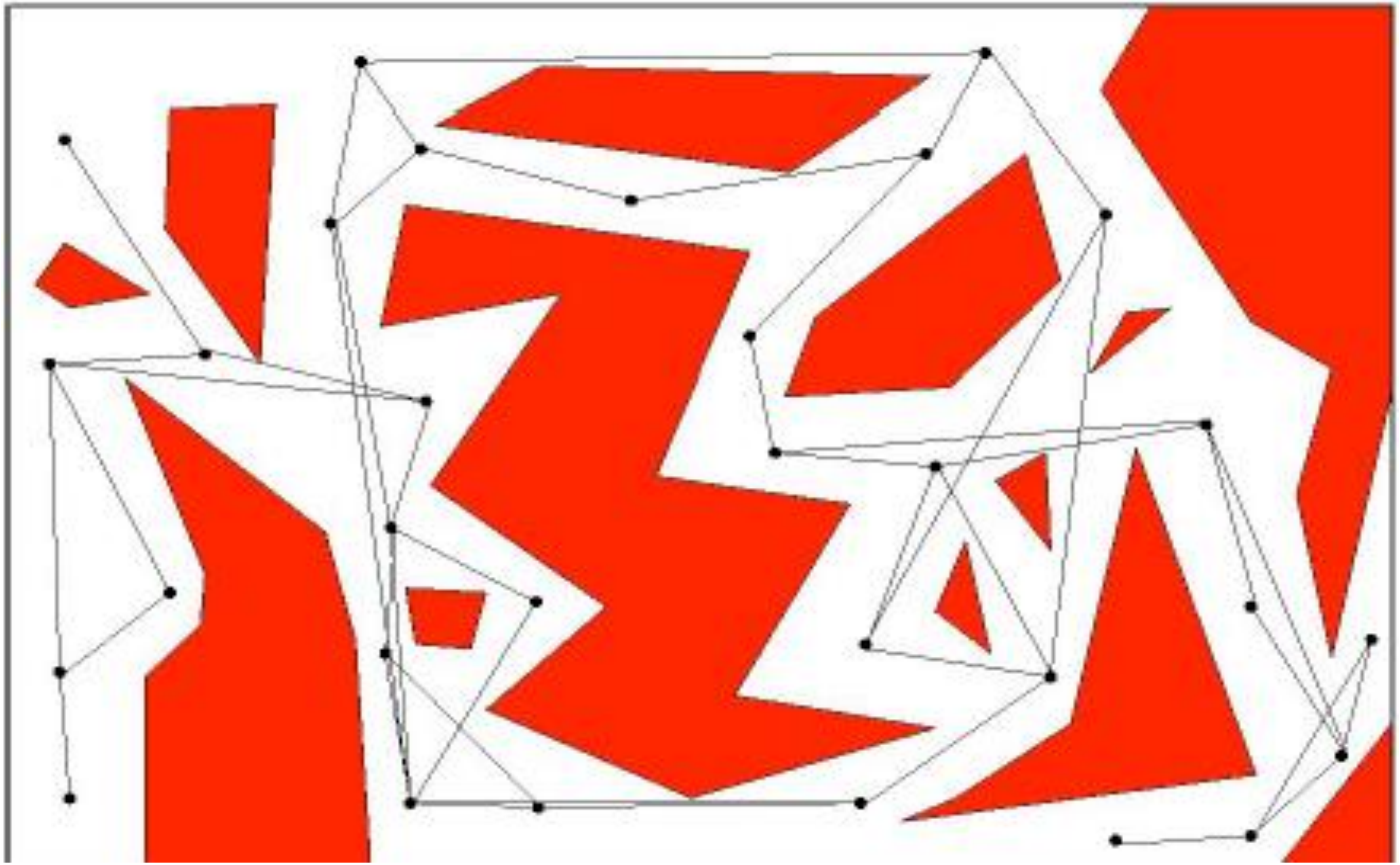
PRM in Short



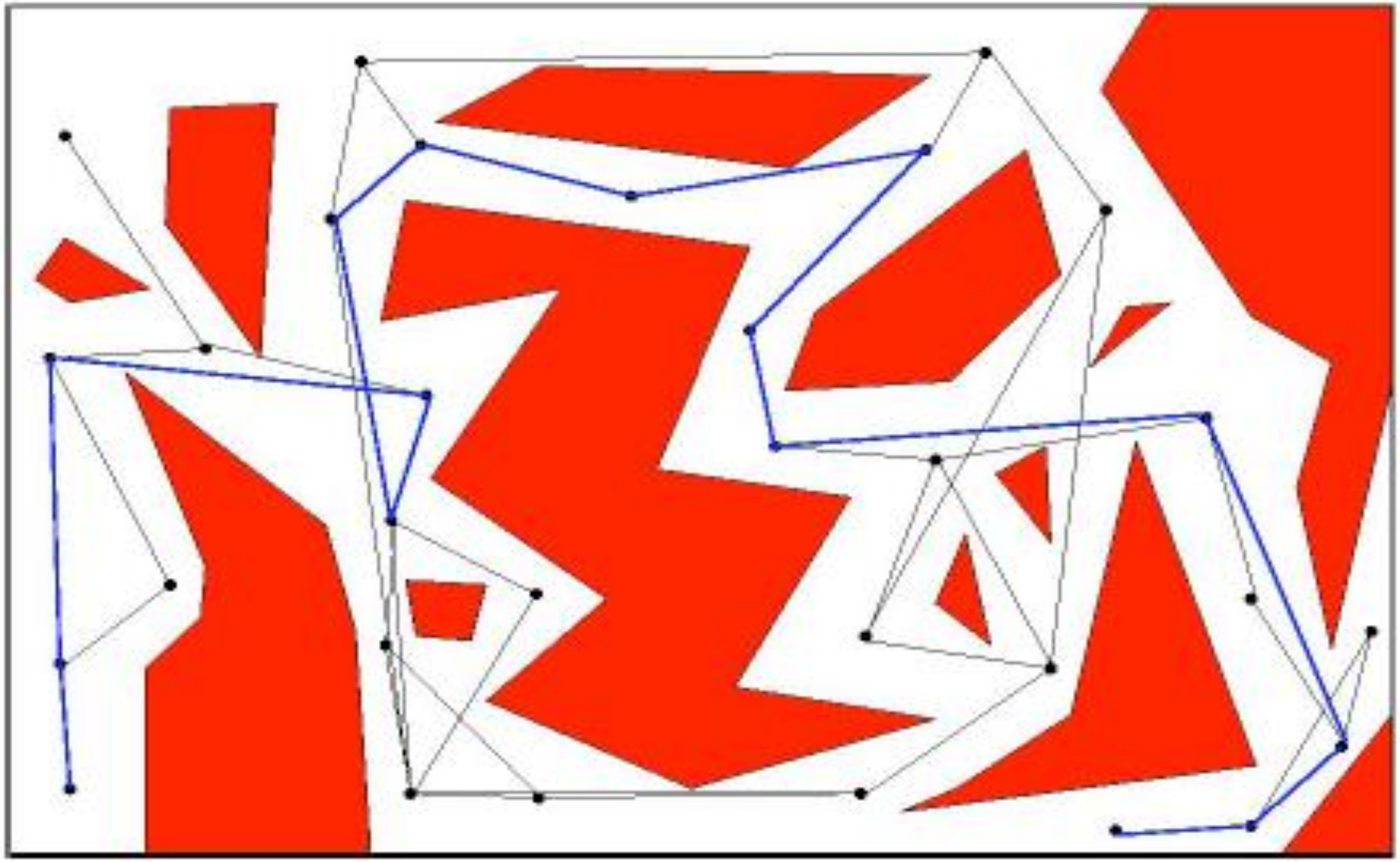
PRM in Short



PRM in Short



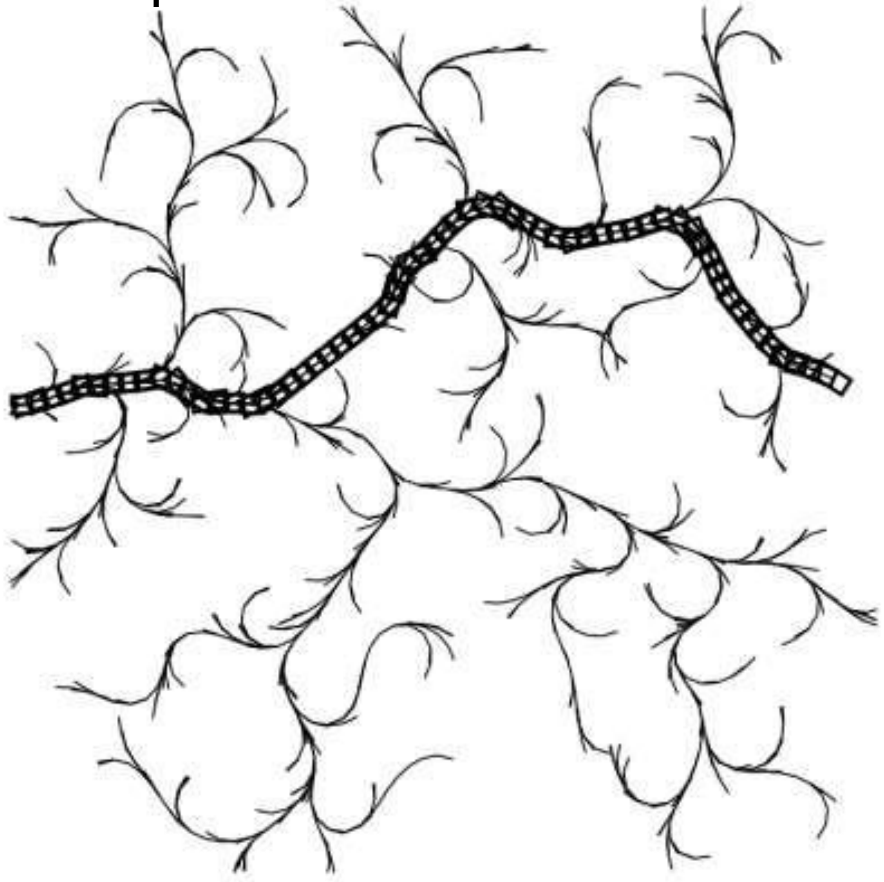
PRM in Short



Sampling based planners

Sample based methods incrementally construct a search tree by gradually improving the resolution and without the need of the roadmap

- Incremental sampling and searching approach without any parameter tuning
- In the limit the tree densely covers the space
- Dense sequence of samples is used as a guide in the construction of the tree



Several version exists:

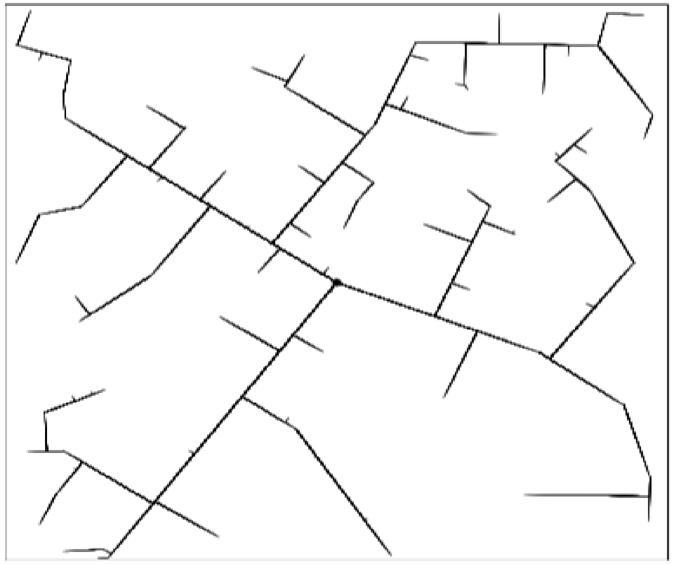
- Rapidly exploring dense tree (RDT)
- Rapidly exploring random tree (RRT – RRT*)

Rapidly exploring dense trees (RDT)

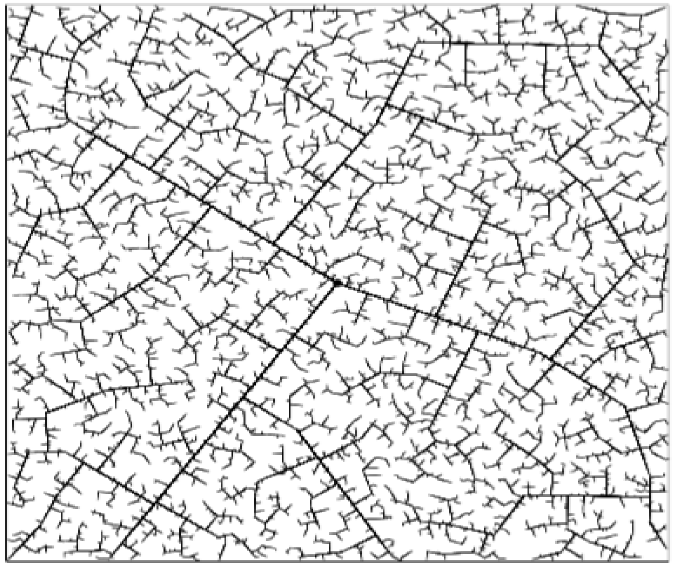
SIMPLE_RDT(q_0)

$\mathcal{G}$.init(q_0);
for $i = 1$ **to** k **do**
  $\mathcal{G}$.add_vertex($\alpha(i)$);
  $q_n \leftarrow$ NEAREST($S(\mathcal{G}), \alpha(i)$);
  $\mathcal{G}$.add_edge($q_n, \alpha(i)$);

α: *D*nse sequence of samples in C
$\alpha(i)$: i^{th} sample of the sample sequence
$G(V, E)$: topological representation of RDT
$S \subset C_{free}$: Set of points reached by G
$S = \bigcup_{e \in E} e([0,1])$ where $e([0,1]) \in C_{free}$



45 iterations



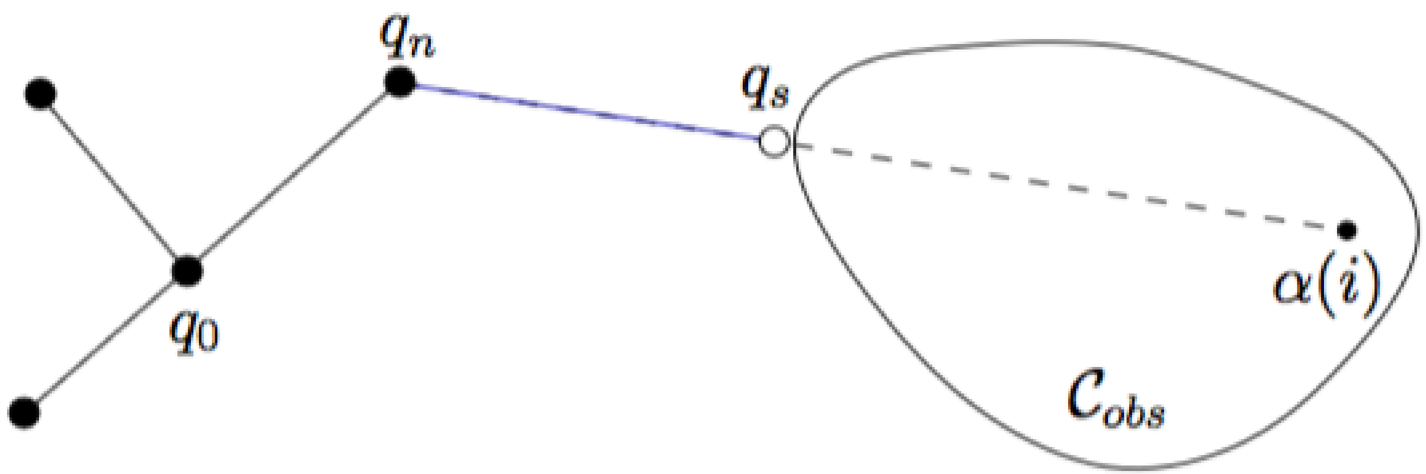
2345 iterations

LaValle, S. (1998). Rapidly-Exploring Random Trees A New Tool for Path Planning.

Rapidly Exploring Dense Trees (RDTs)

RDT(q_0)

$\mathcal{G}$.init(q_0);
for $i = 1$ **to** k **do**
    $q_n \leftarrow$ NEAREST($S, \alpha(i)$);
    $q_s \leftarrow$ STOPPING-CONFIGURATION(q_n,$\alpha(i)$);
    if $q_s \neq q_n$ **then**
        $\mathcal{G}$.add_vertex(q_s);
        $\mathcal{G}$.add_edge(q_n, q_s);



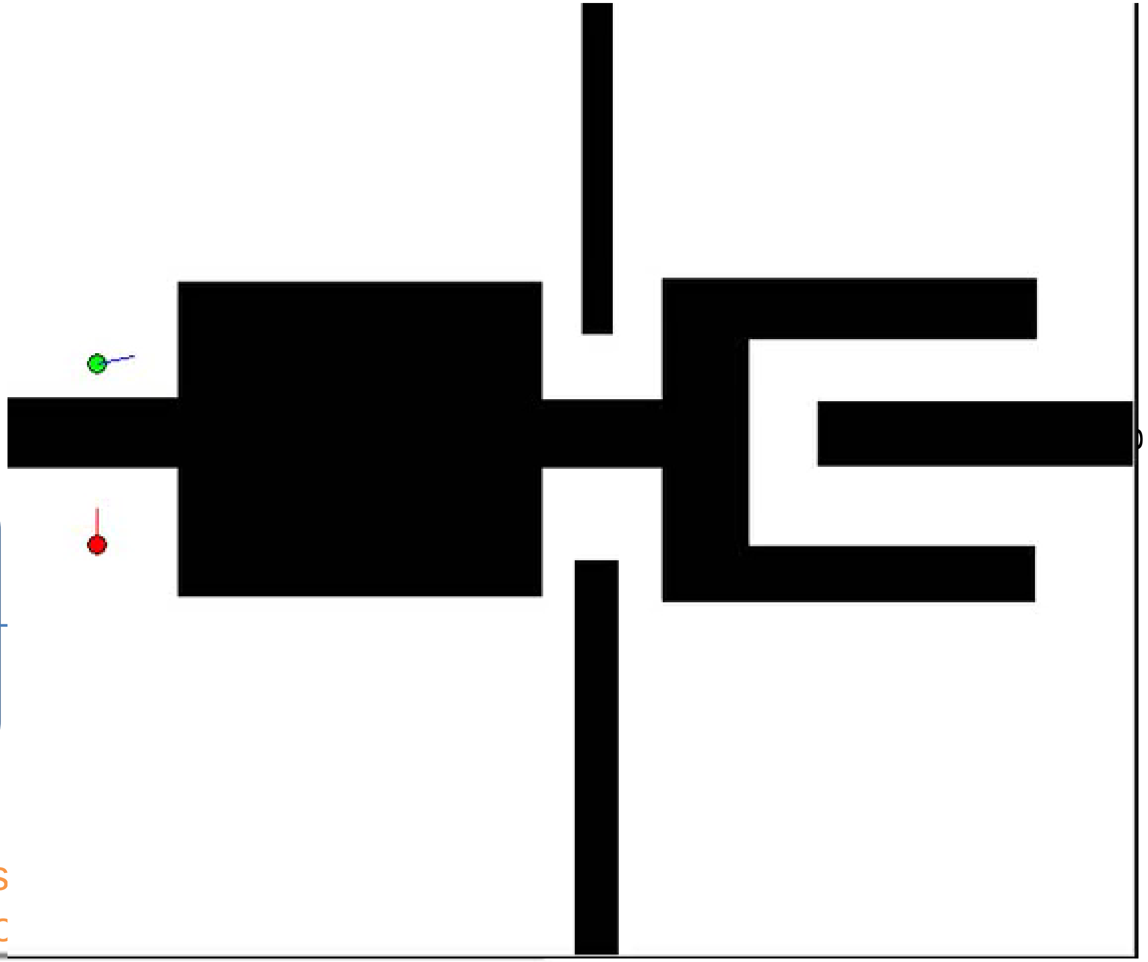
LaValle, S. (1998). Rapidly-Exploring Random Trees
A New Tool for Path Planning.

Rapidly Exploring Dense Trees (RDTs)

RDT_BALANCED_BIDIRECTIONAL(q_I, q_G)

T_a.init(q_I); T_b.init(q_G);
for $i = 1$ **to** K **do**
    $q_n \leftarrow$ NEAREST($S_a, \alpha(i)$);
    $q_s \leftarrow$ STOPPING-CONFIGURATION(q_n,$\alpha(i)$);
    if $q_s \neq q_n$ **then**
        T_a.add_vertex(q_s);
        T_a.add_edge(q_n, q_s);
        $q'_n \leftarrow$ NEAREST(S_b, q_s);
        $q'_s \leftarrow$ STOPPING-CONFIGURATION(q'_n,q_s);
        if $q'_s \neq q'_n$ **then**
            T_b.add_vertex(q'_s);
            T_b.add_edge(q'_n, q'_s);
        if $q'_s = q_s$ **then return** SOLUTION;
    if $|T_b| > |T_a|$ **then** SWAP(T_a, T_b);
return FAILURE

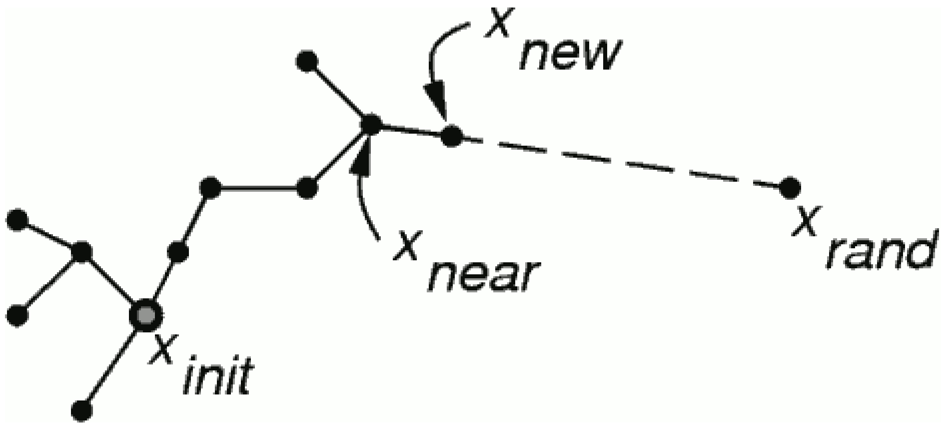
Two trees
than the c



Rapidly Exploring Random Trees (RRT)

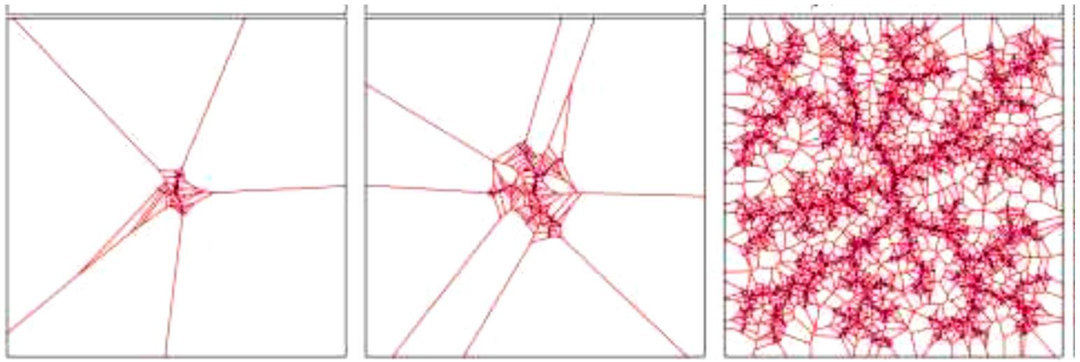
Algorithm 1 $\mathcal{T} = (V, E) \leftarrow \text{RRT}(z_{init})$

$\mathcal{T} \leftarrow \text{InitializeTree}();$
$\mathcal{T} \leftarrow \text{InsertNode}(\emptyset, z_{init}, \mathcal{T});$
for $i = 1$ **to** $i = N$ **do**
  $z_{rand} \leftarrow \text{Sample}(i);$
  $z_{nearest} \leftarrow \text{Nearest}(\mathcal{T}, z_{rand});$
  $(x_{new}, u_{new}, T_{new}) \leftarrow \text{Steer}(z_{nearest}, z_{rand});$
  if $\text{ObstacleFree}(x_{new})$ **then**
    $\mathcal{T} \leftarrow \text{InsertNode}(z_{new}, \mathcal{T});$
  end if
end for
return $\mathcal{T}$



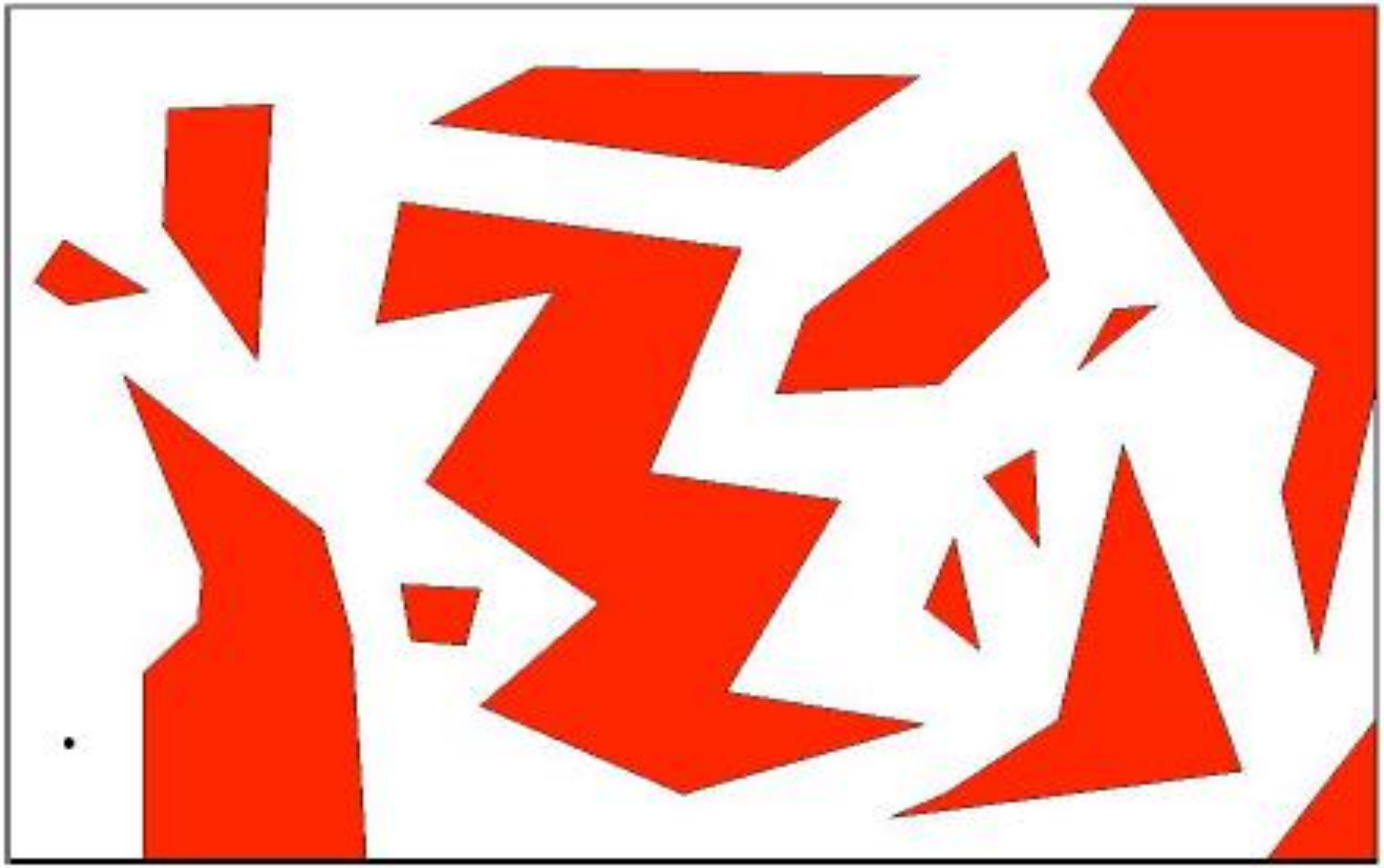
RRT improves on the basic RDT

- Steering the system toward random samples according to kinodinamics
- Bias the tree towards unexplored areas by using a Voronoy bias

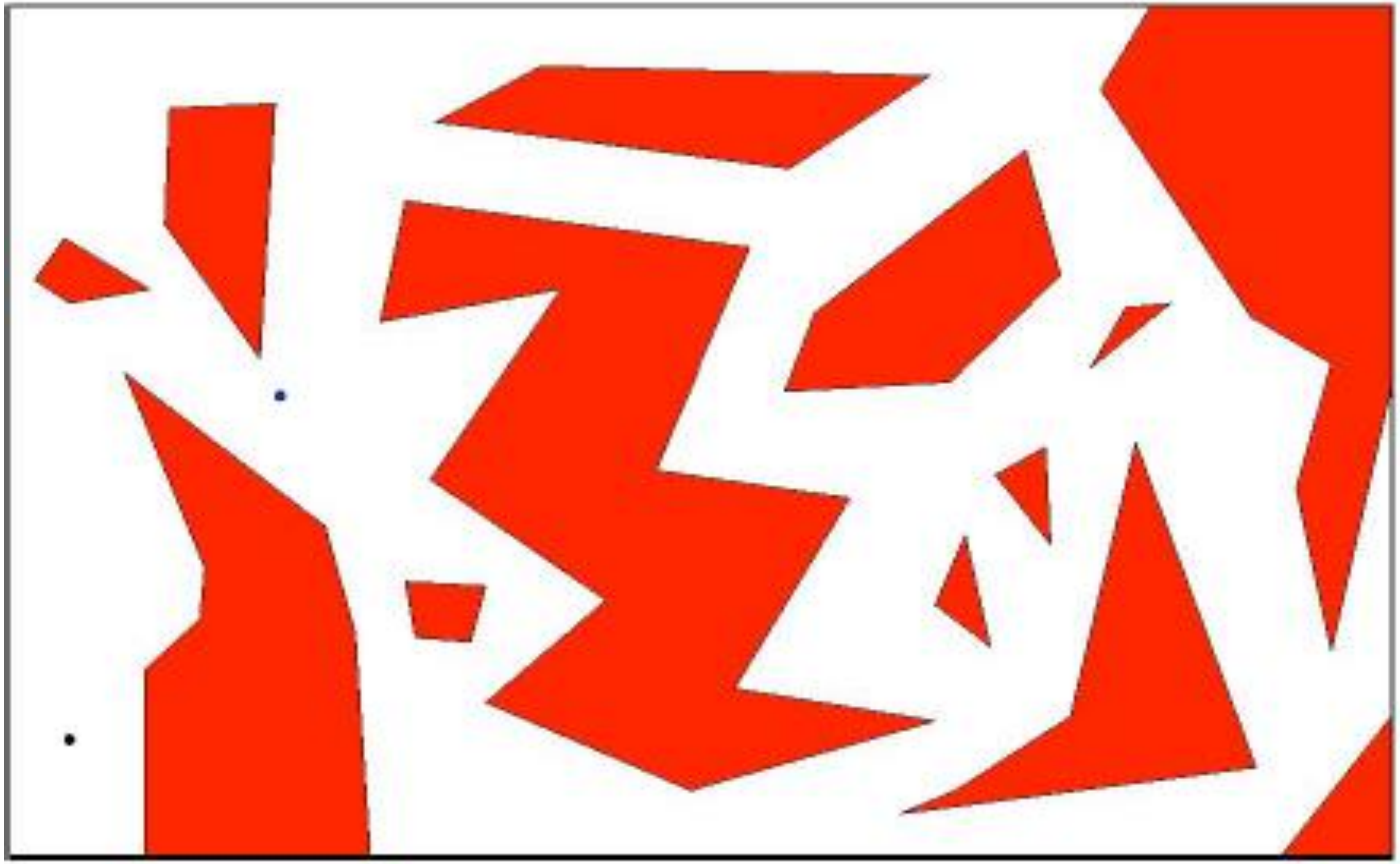


LaValle, S. M., Kuffner, J., "Randomized Kinodynamic Planning", *IEEE International Conference on Robotics and Automatio*, 1999

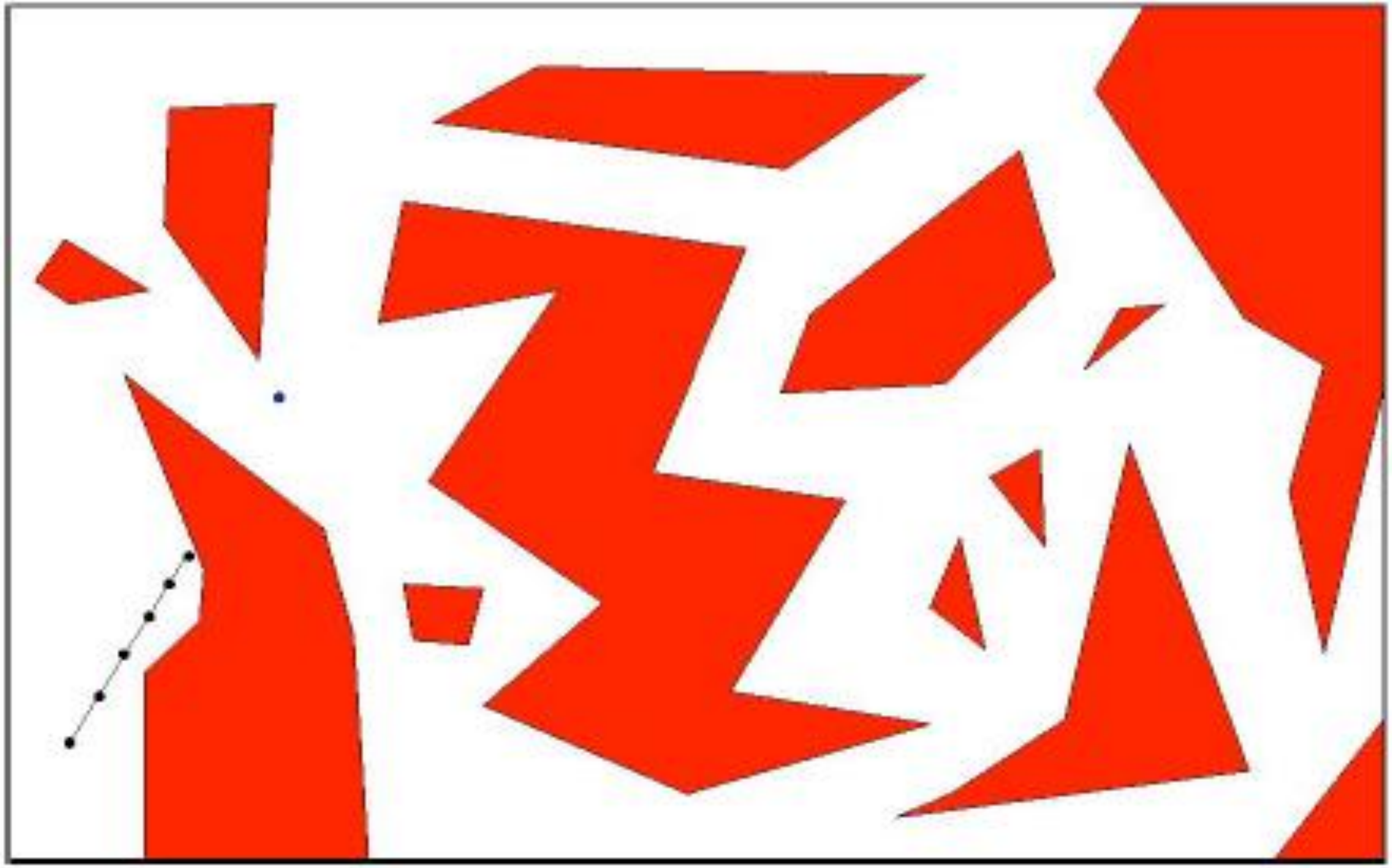
RRT in Short



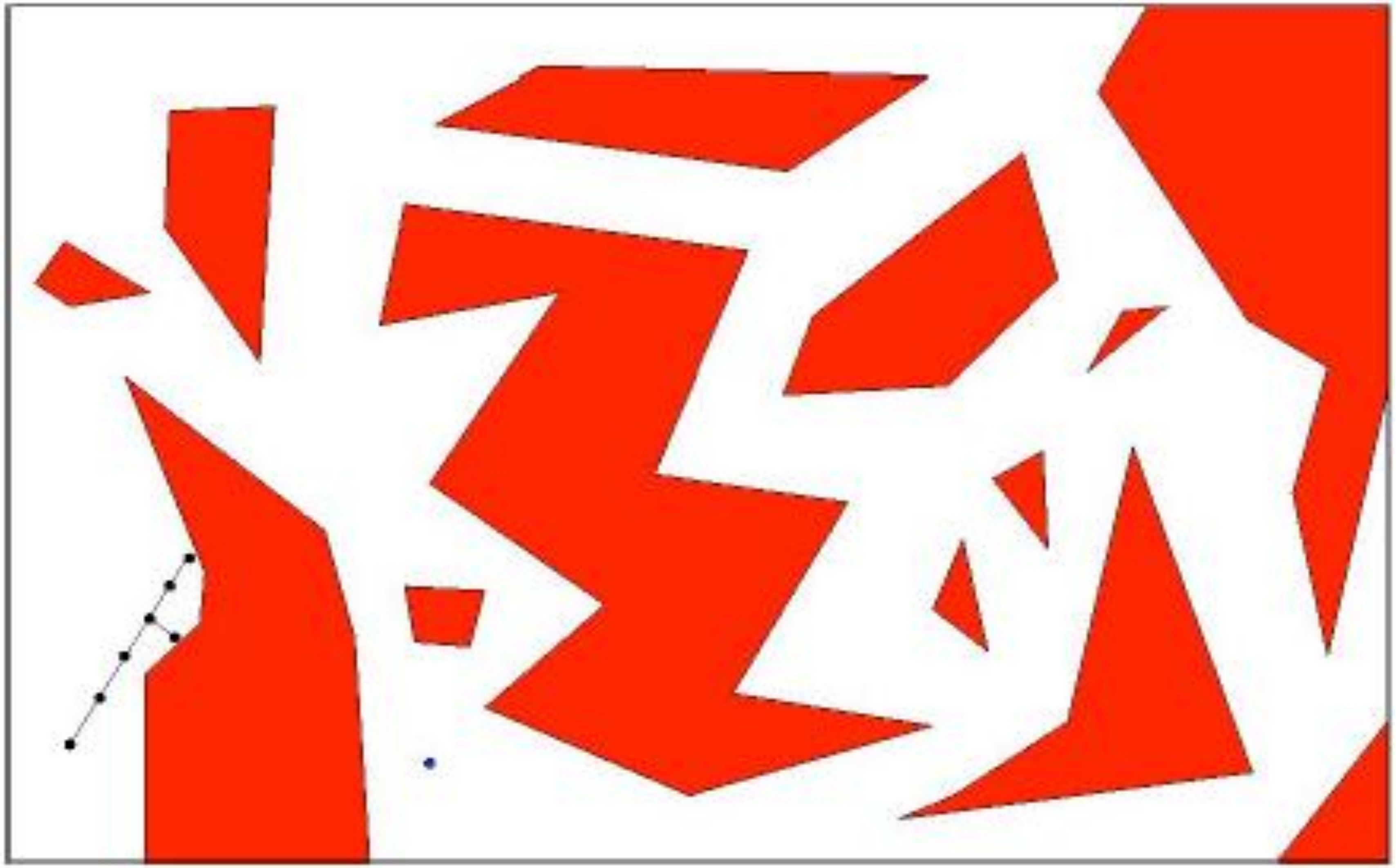
RRT in Short



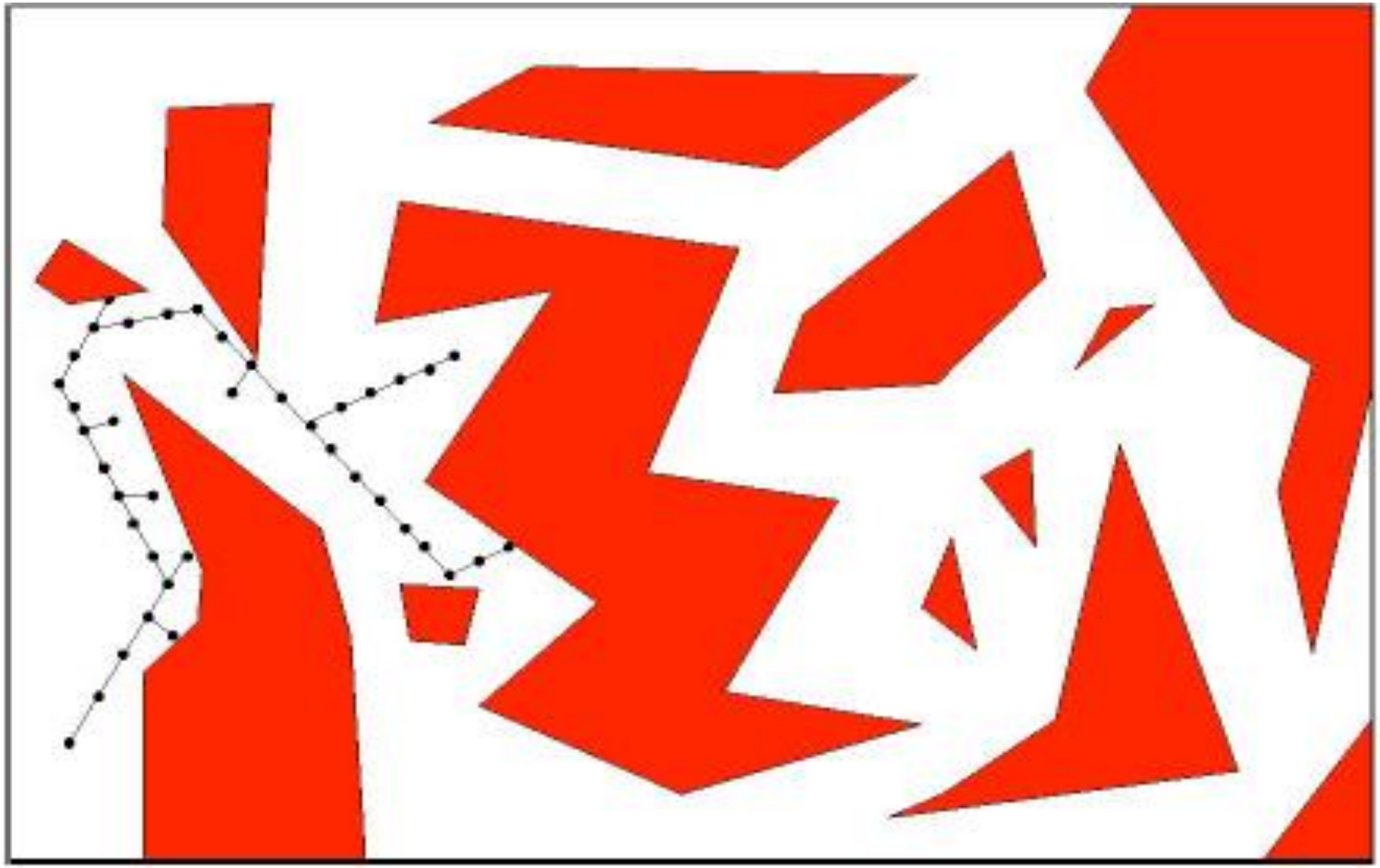
RRT in Short



RRT in Short

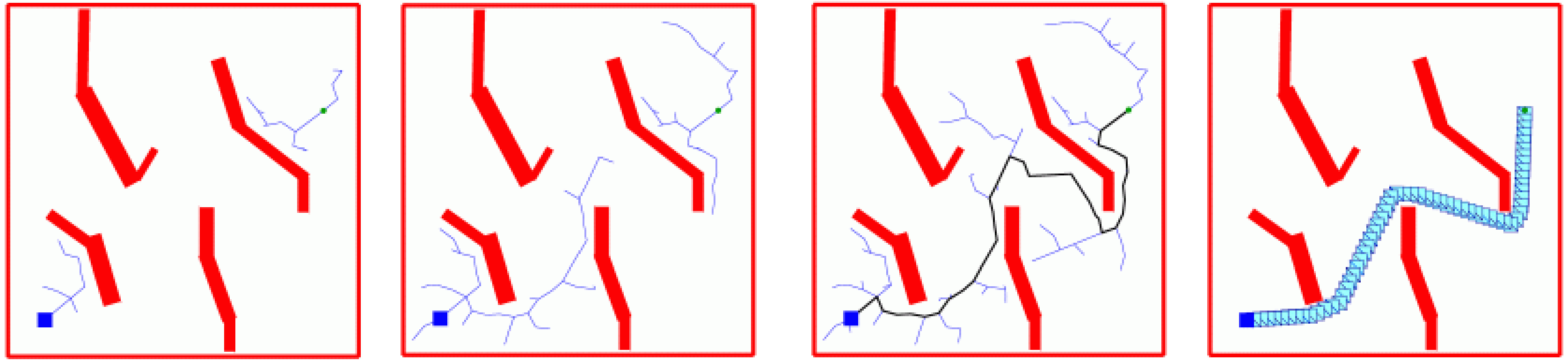


RRT in Short

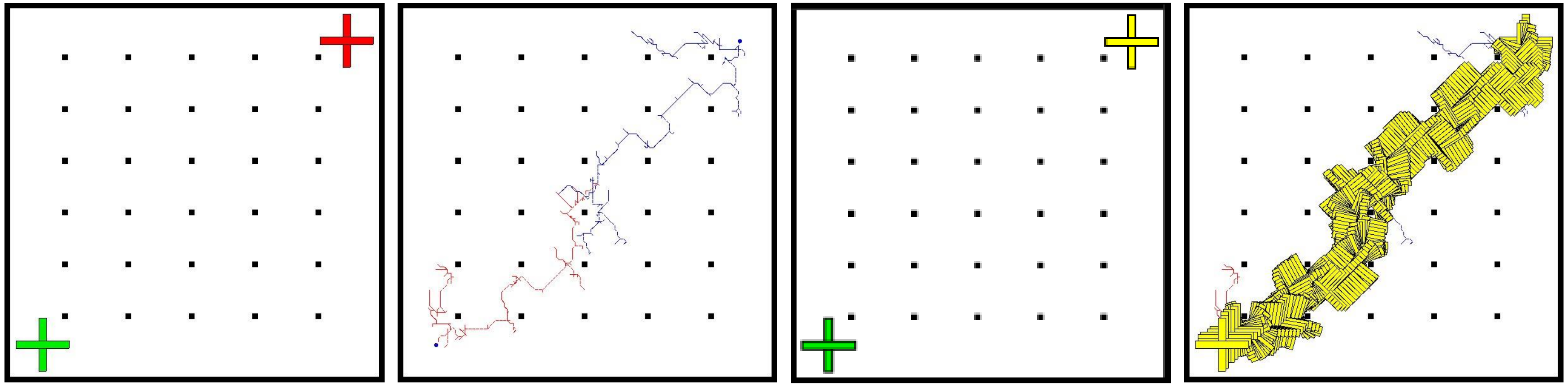


A few examples

Simple object

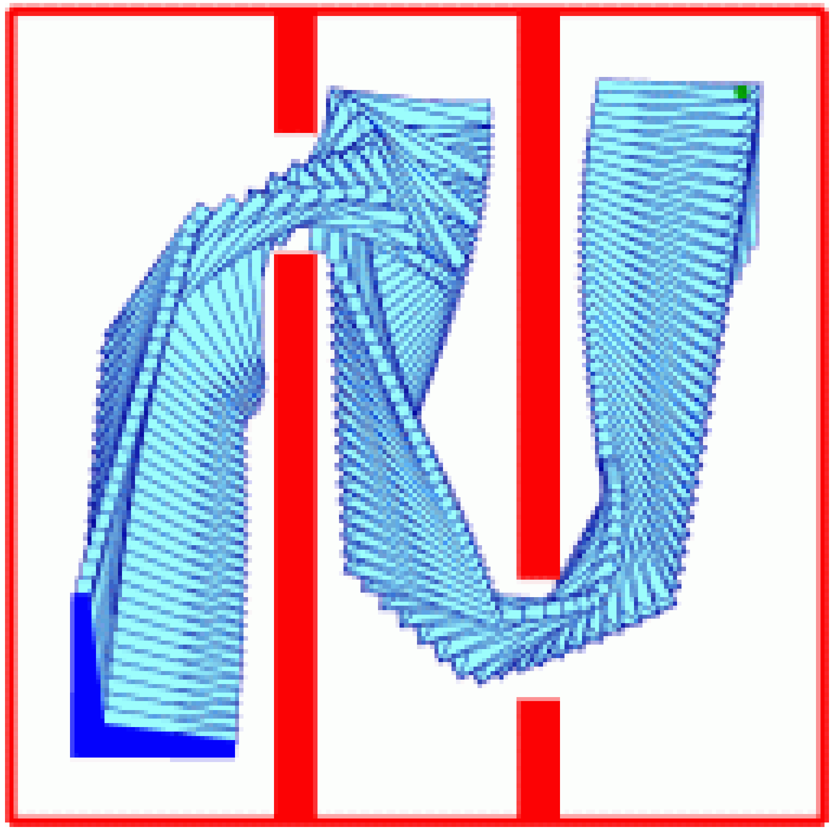


Complex Object



A few examples

More complex object

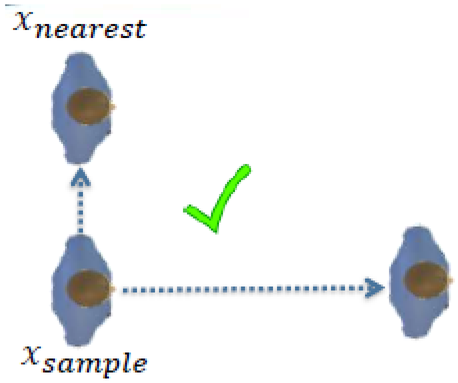


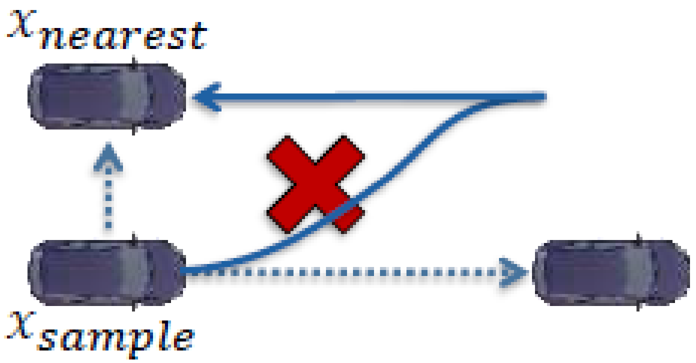
RRT pros and cons

Not AsymptotIcally Optimal !!!

Pros	Cons
Asymptotically complete	No optimality guarantee (!)
Works reasonably well in high dimensional state-spaces	Produces “jerky” paths in finite time
No two-state boundary value solver required	Hardly any offline computations possible
Easy implementation	
Easy to deal with constrained platforms	

RRT exploration quality is sensitive to distance metric and obtaining metrics and obtaining distance metrics for non-holonomic systems is non-trivial

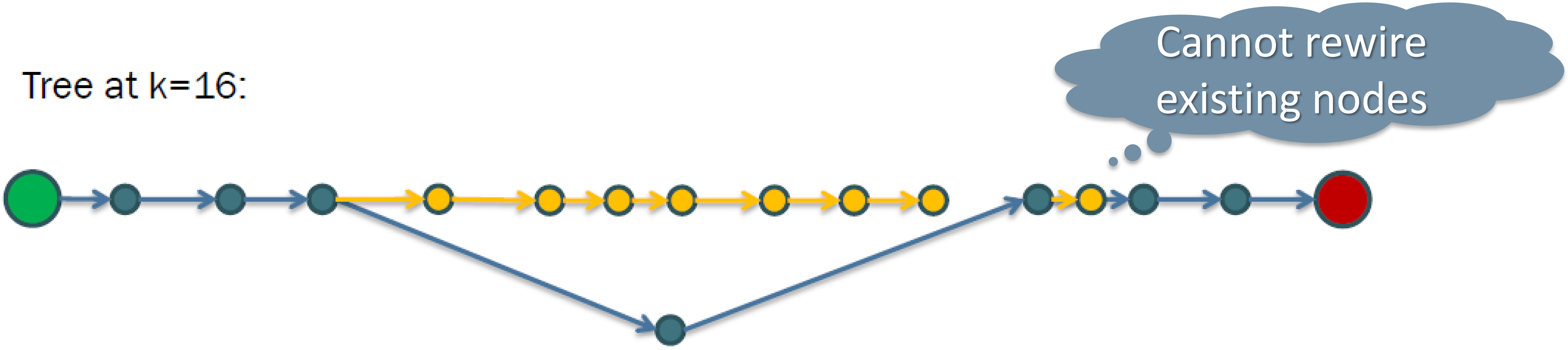




RRT exstensions and RTT*

Few extensions have been proposed to the basic RRT algorithm

- *Bidirectional RRT* grows two trees from start and goal and frequently tries to merge them
- *Goal-biased RRT* samples the goal state every n-th sample to tradeoff exploration and exploitation
- *RRT** Introduces local rewiring step to obtain asymptotic optimality…

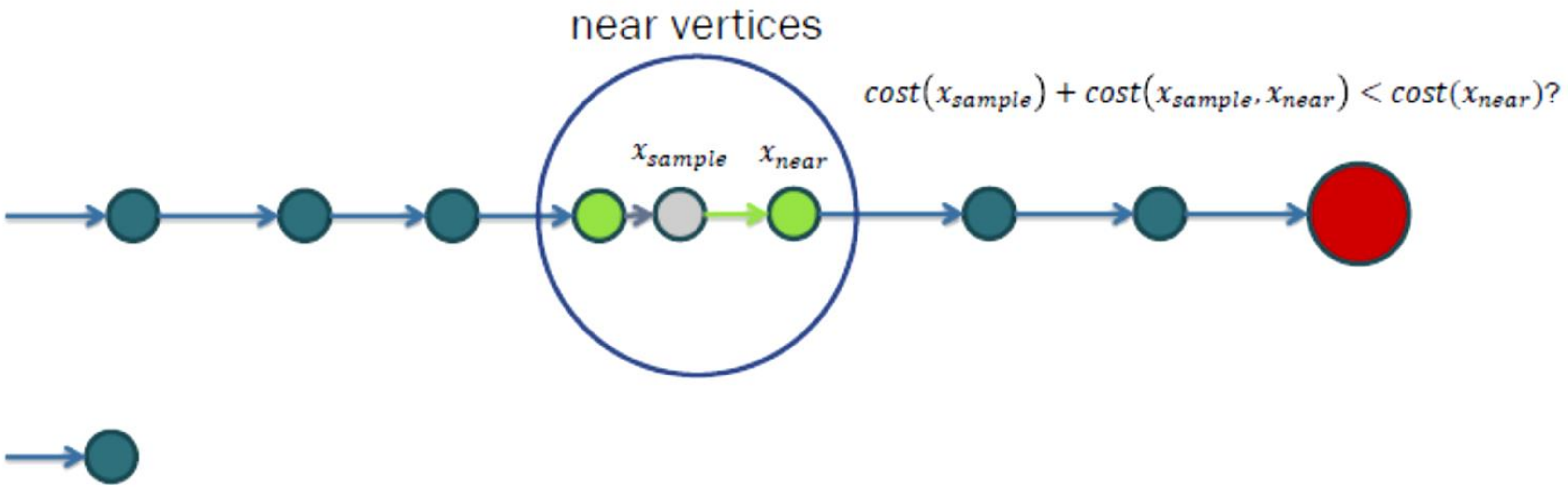


S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning", Robotics: Science and Systems, 2010

RRT exstensions and RTT*

Few extensions have been proposed to the basic RRT algorithm

- *Bidirectional RRT* grows two trees from start and goal and frequently tries to merge them
- *Goal-biased RRT* samples the goal state every n-th sample to tradeoff exploration and exploitation
- *RRT** Introduces local rewiring step to obtain asymptotic optimality…



S. Karaman and E. Frazzoli, “Incremental sampling-based algorithms for optimal motion planning”, Robotics: Science and Systems, 2010

RRT* pros and cons

Pros	Cons
Asymptotically complete	Two-state boundary value solver required for the rewiring
Asymptotically optimal guarantee	Produces “jerky” paths in finite time
Works reasonably well in high dimensional state-spaces	Hardly any offline computations possible
No two-state boundary value solver required	
Easy implementation	
Easy to deal with constrained platforms	