POLITECNICO
MILANO 1863

Unmanned autonomous vehicles in air land and sea

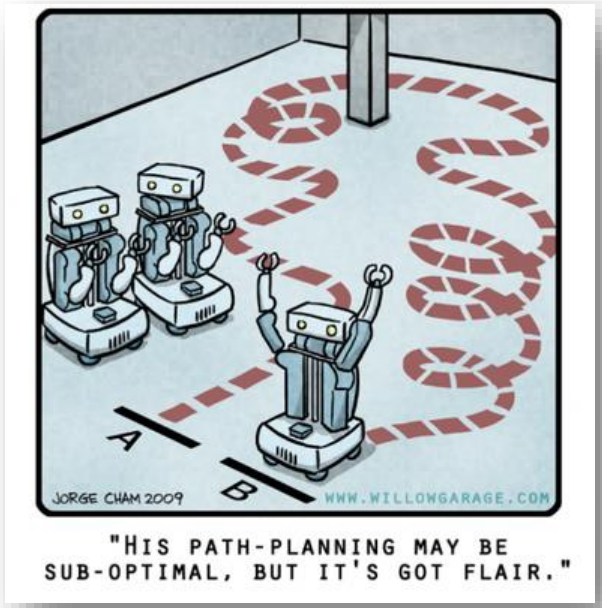
Trajectory planning in land vehicles

Matteo Matteucci
matteo.matteucci@polimi.it

Artificial Intelligence and Robotics Lab - Politecnico di Milano

Motion planning

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



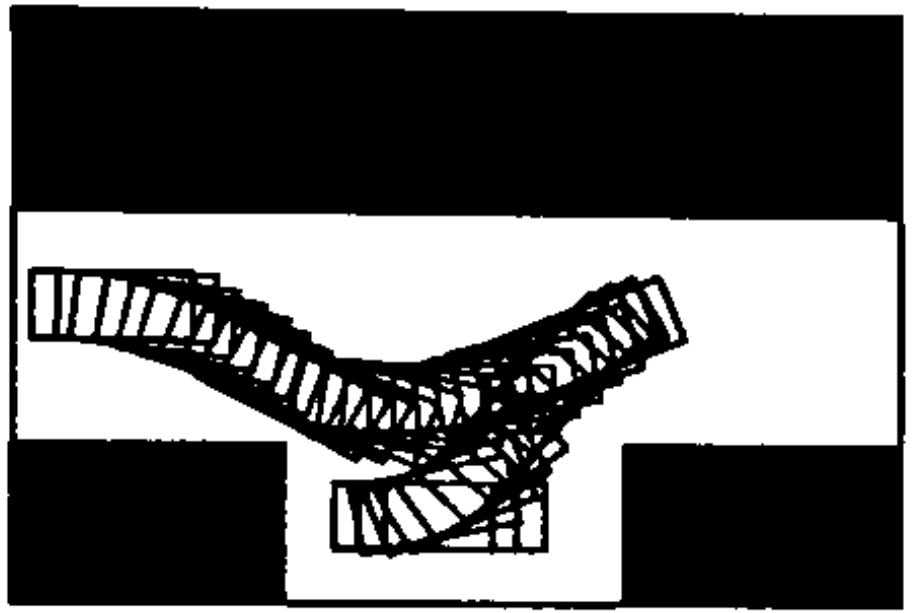
Robot Motion Planning Goals

- 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 vehicle)
- W: Euclidean space where A moves (typically $W = \Re^2 \ \ or \ \ \Re^3$)
- 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)

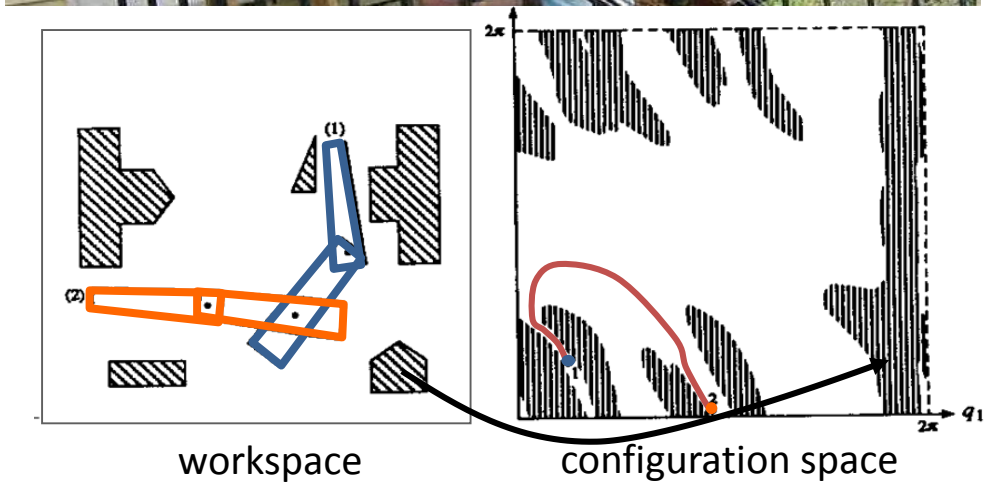
Given an initial pose and a goal pose of A in W, generate a continuous sequence of poses of A avoiding contact with the B_i, starting at the initial pose and terminating at the goal pose.

Configuration Space (C-Space)

To speed up 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

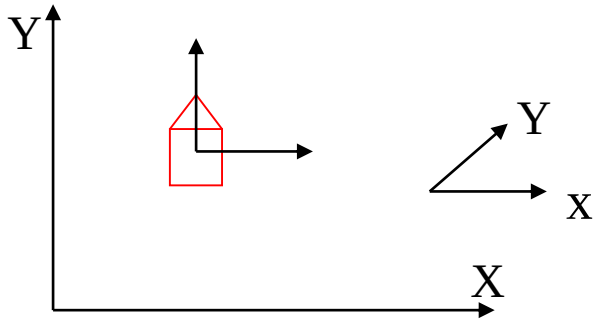
Planning can be performed in C-Space



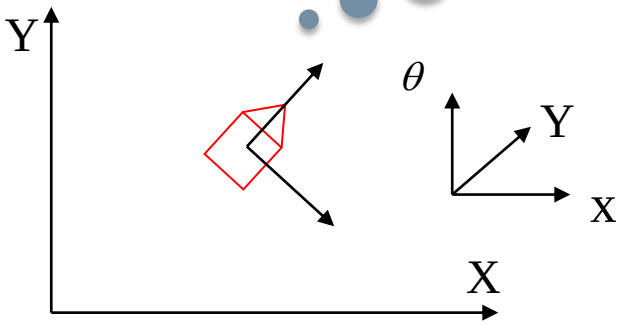
Mobile robot 2D C-Space

Non holonomic constraints can't be C-Space obstacles

A robot can translate in the plane and/or rotate

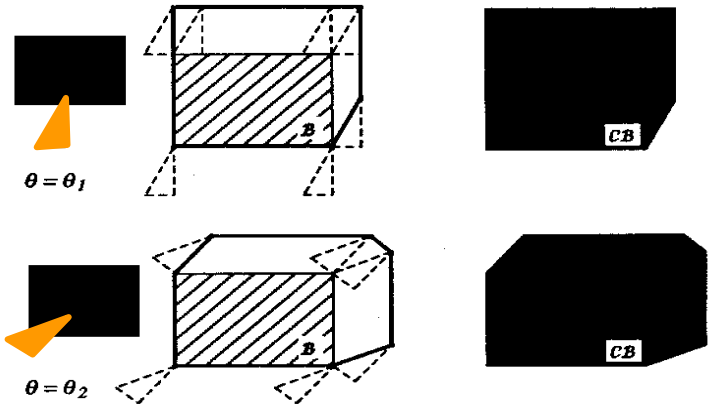


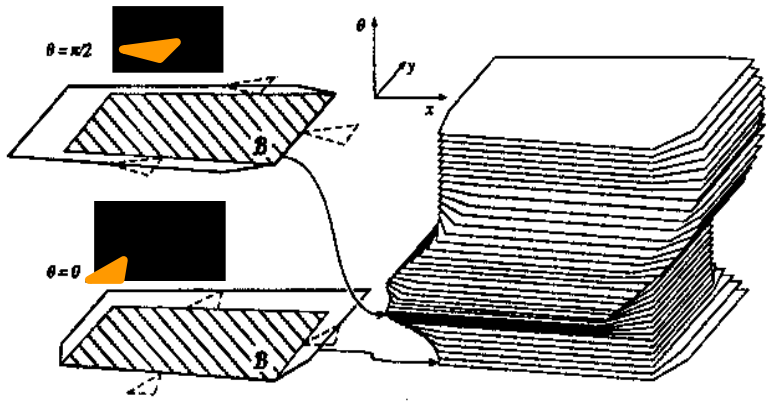
C-space: 2D (x, y)



C-space: 3D (x, y, θ)

Obstacles should be expanded according to the robot orientation

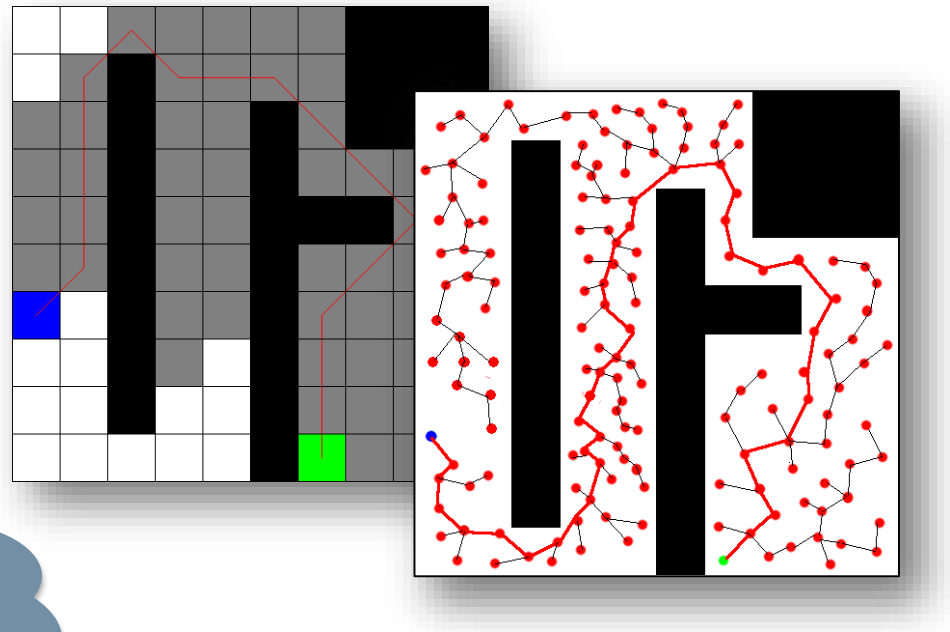




What a planner for an autonomous car?

Several planners have been used for trajectory planning on unmanned vehicles

- The Dynamic Window Approach
- Graph-Search & State-Lattice planning
- Randomized Approaches: Probabilistic Roadmaps, RRT, RRT*
- Potential fields & the Fast Marching algorithm
- …

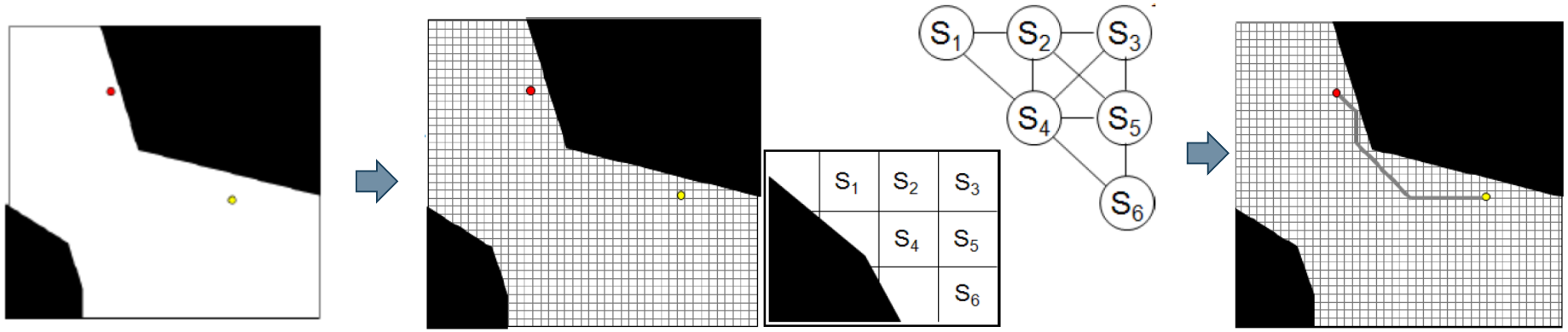


We will se only few of these!

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)

Planning problem ingredients

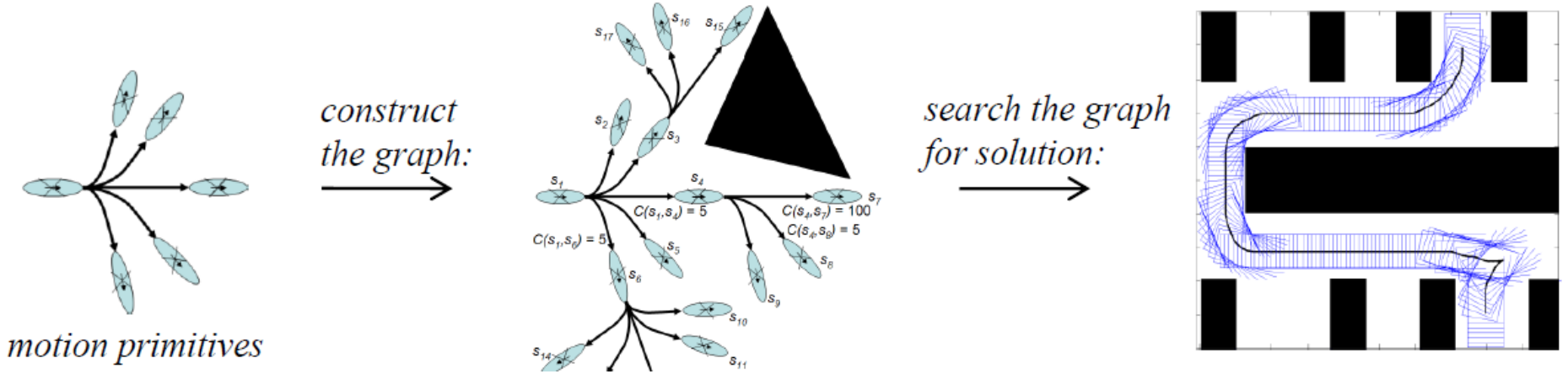
Typical components of a Search-based Planner

- 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

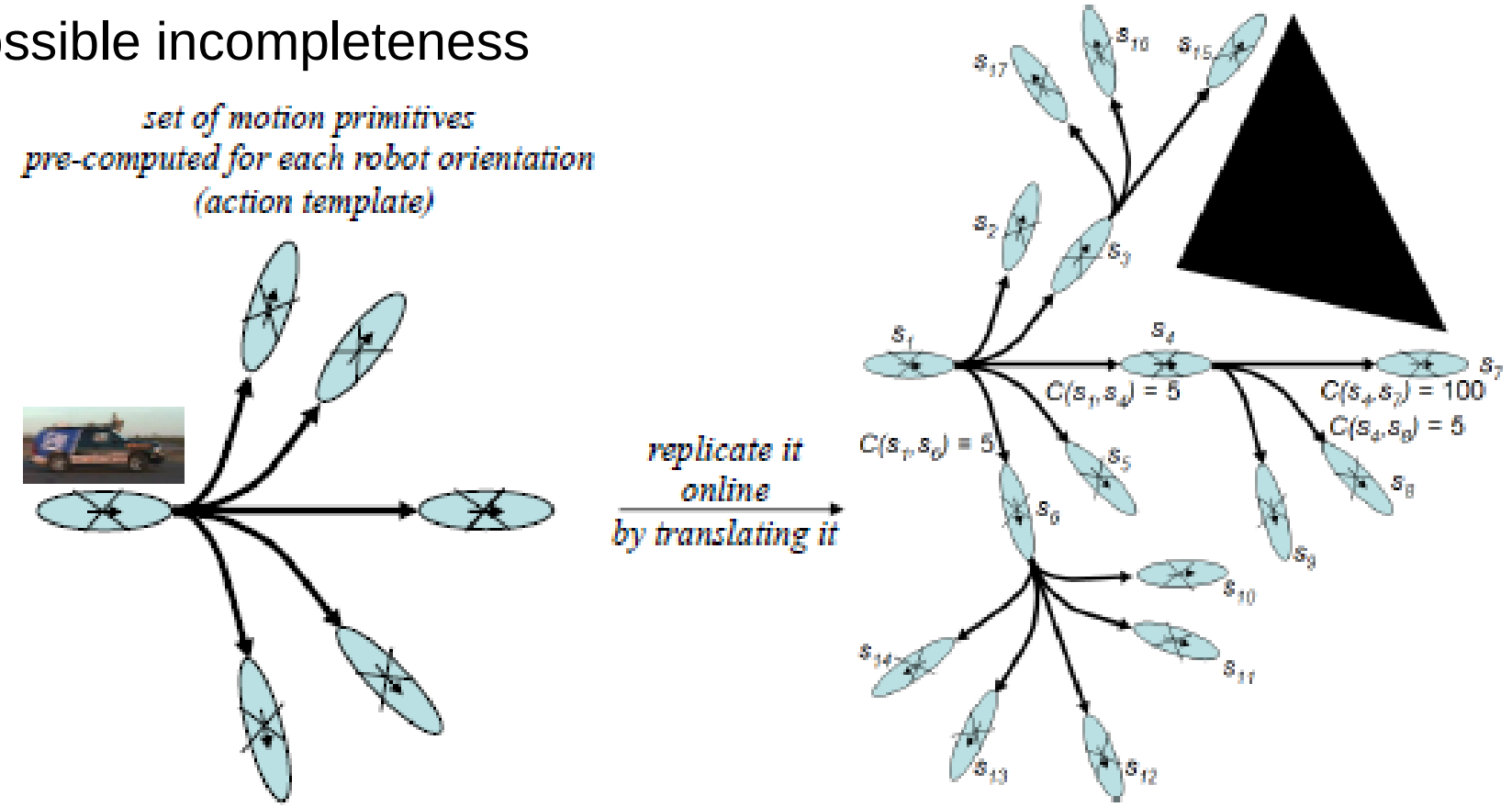
The graph can be built taking into account robot dynamics/kinematics constraints



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

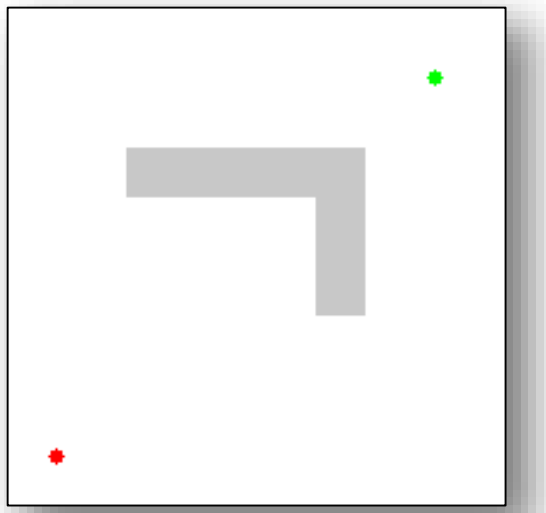
- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness

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

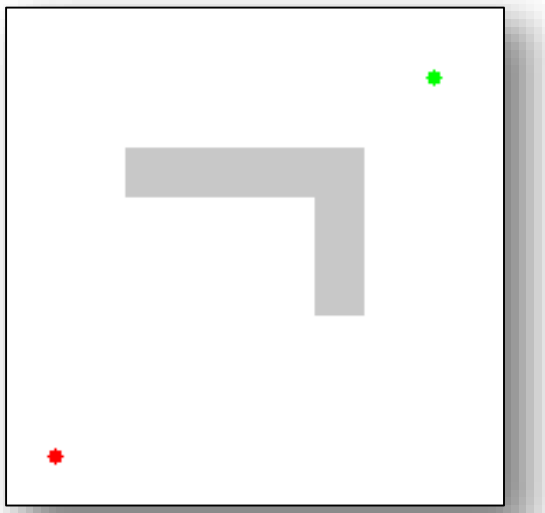
Exact and approximate planning

Different algorithms are available

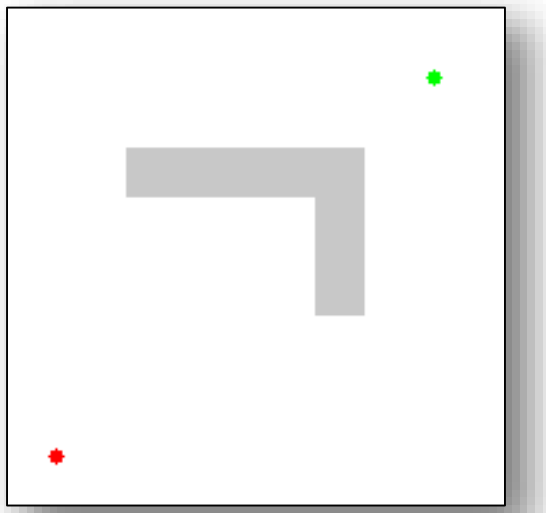
- Returning the optimal path (e.g., Dijstra, A*, …)
- Returning an ε sub-optimal path
  (e.g., weighted A*, ARA*, AD*, R*, D* Lite, ...)



Dijstra



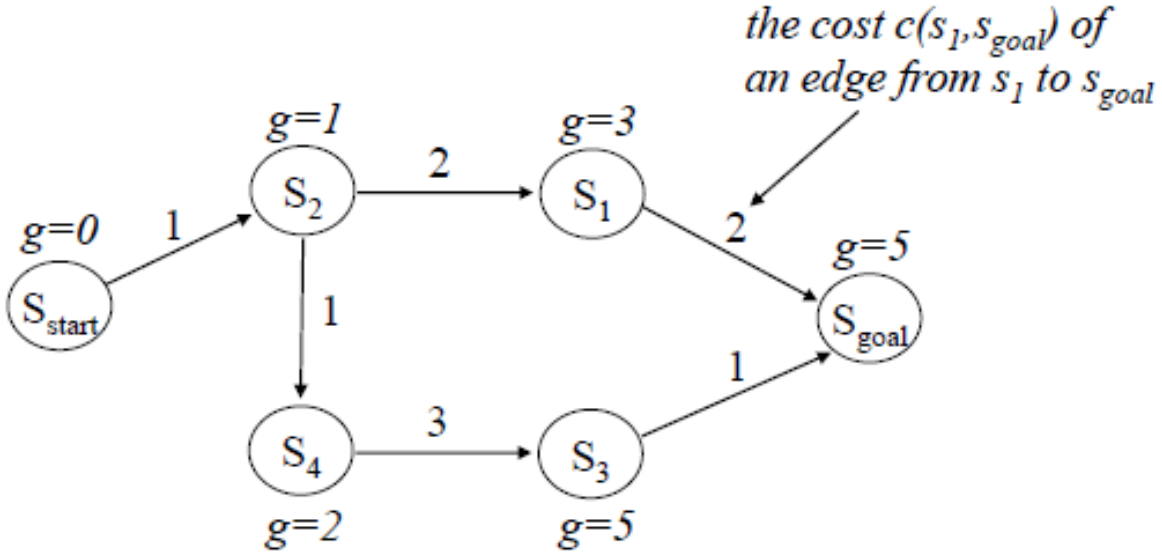
A*



weighted A*

Searching graphs for least cost path

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

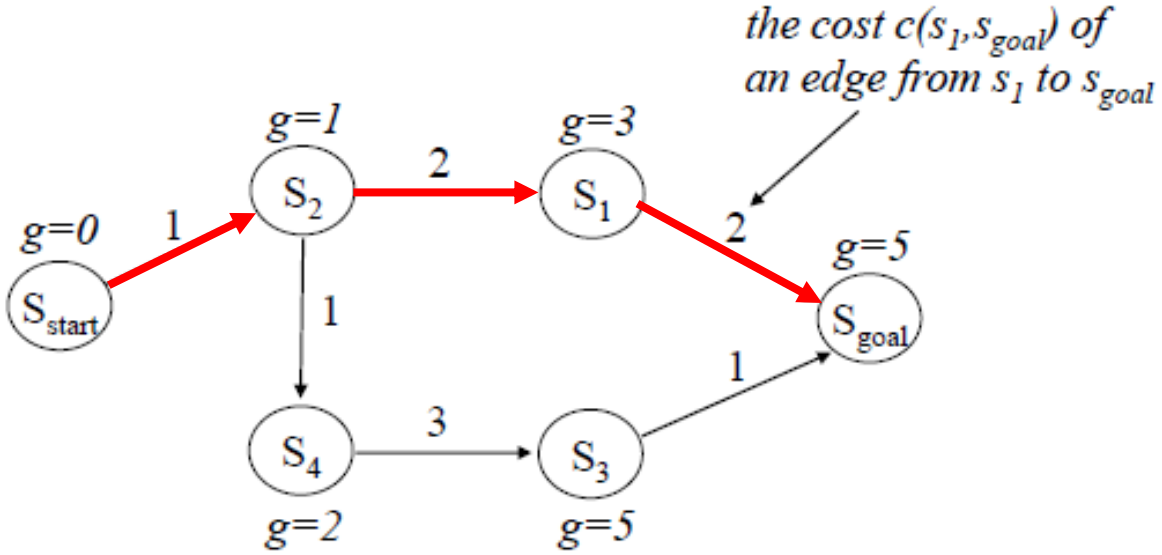


Many search algorithms compute optimal g-values for relevant states

- *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



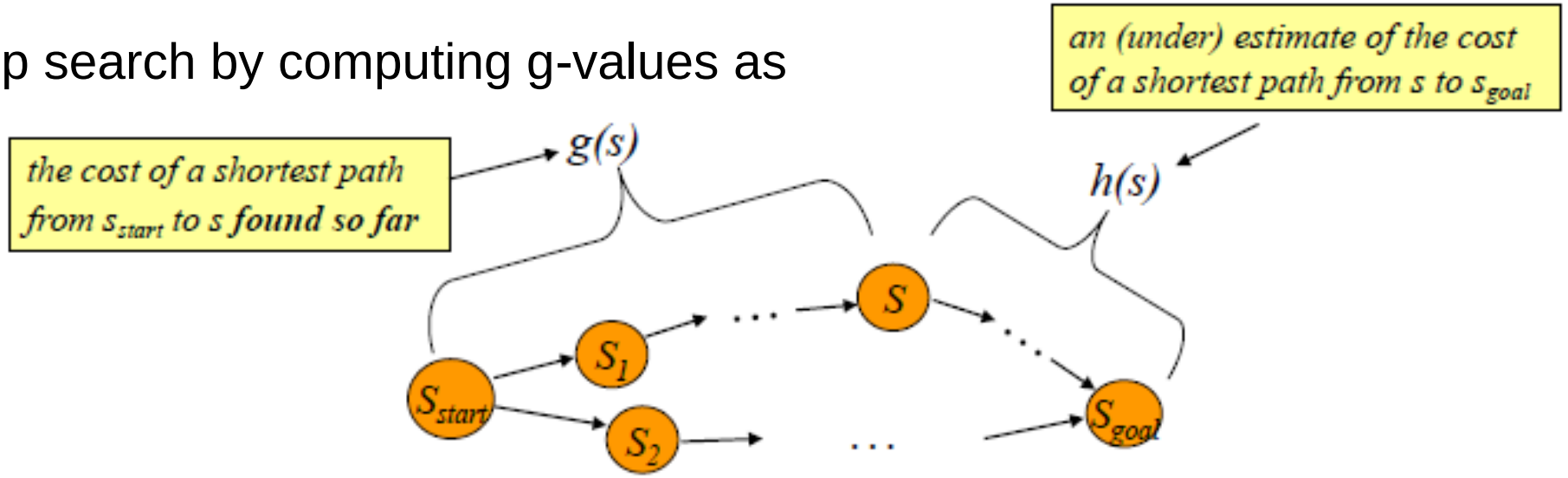
Least-cost path is a greedy path computed by backtracking:

- 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

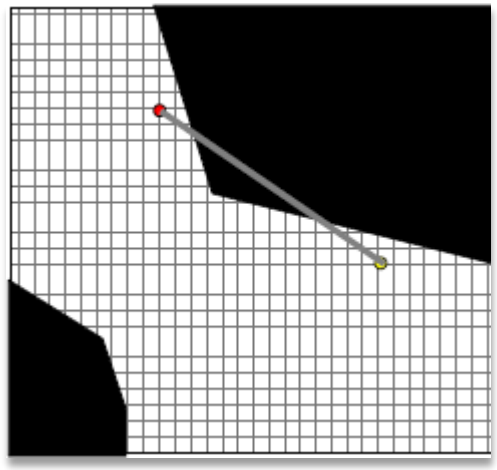
A* speeds up search by computing g-values as



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

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

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

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

- 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;

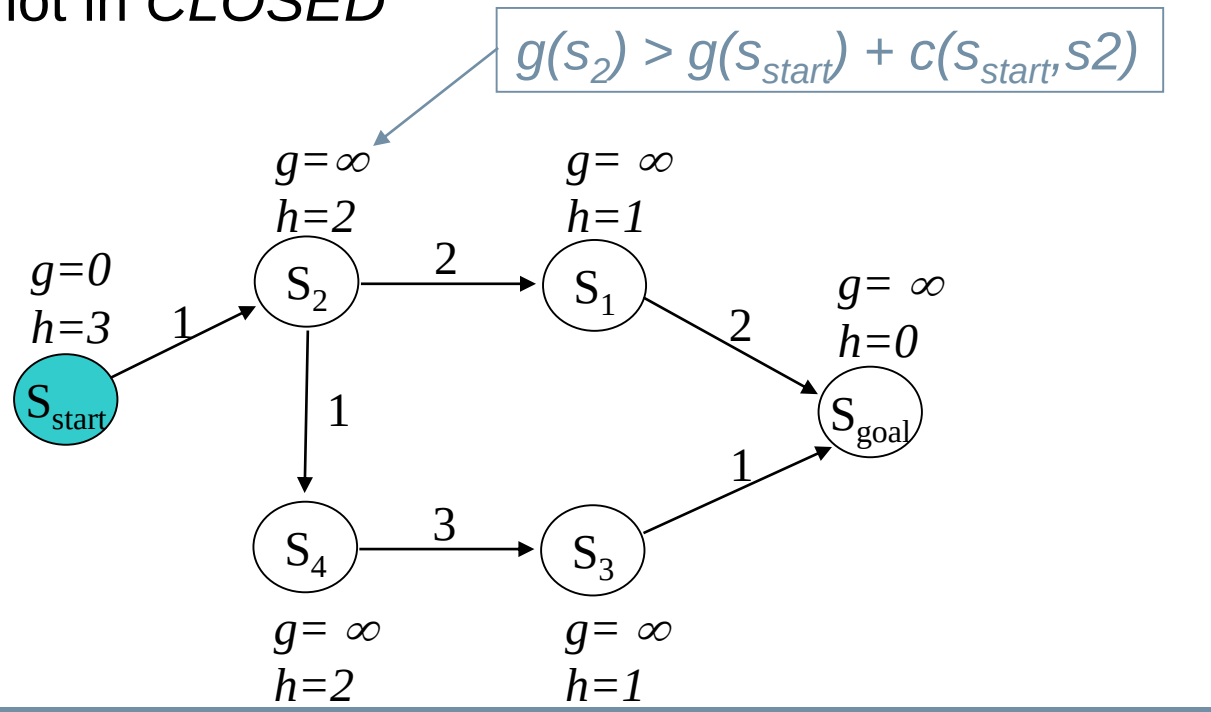
Set of states already expanded

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 = {}

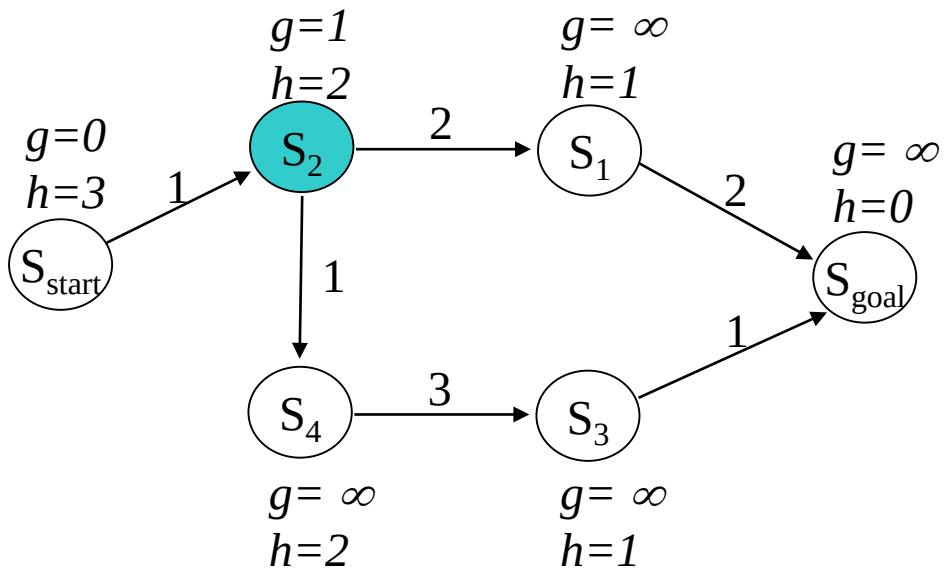
OPEN = {s_{start}}

next state to expand: s_{start}

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*;

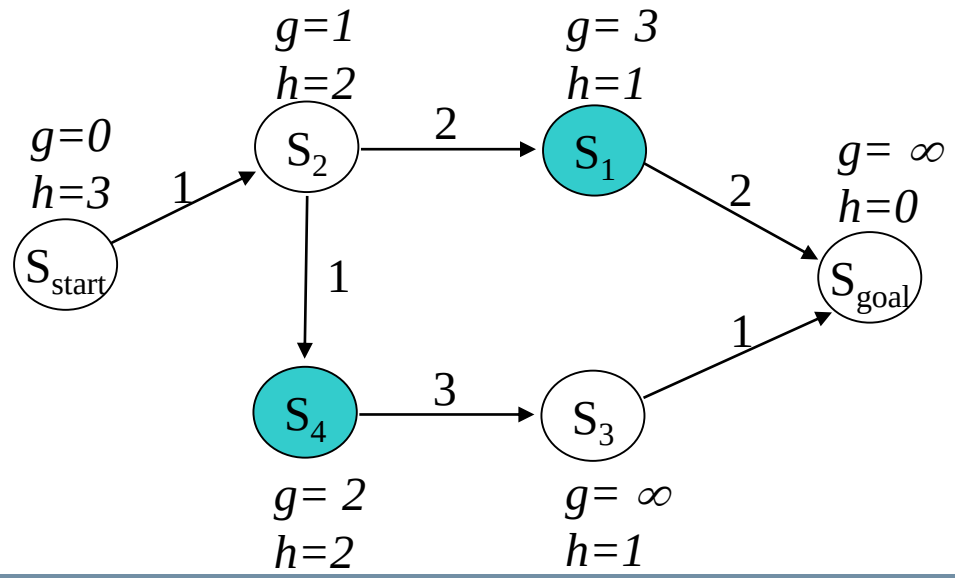


CLOSED = {s_{start}}
OPEN = {s_2}
next state to expand: s_2

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*;

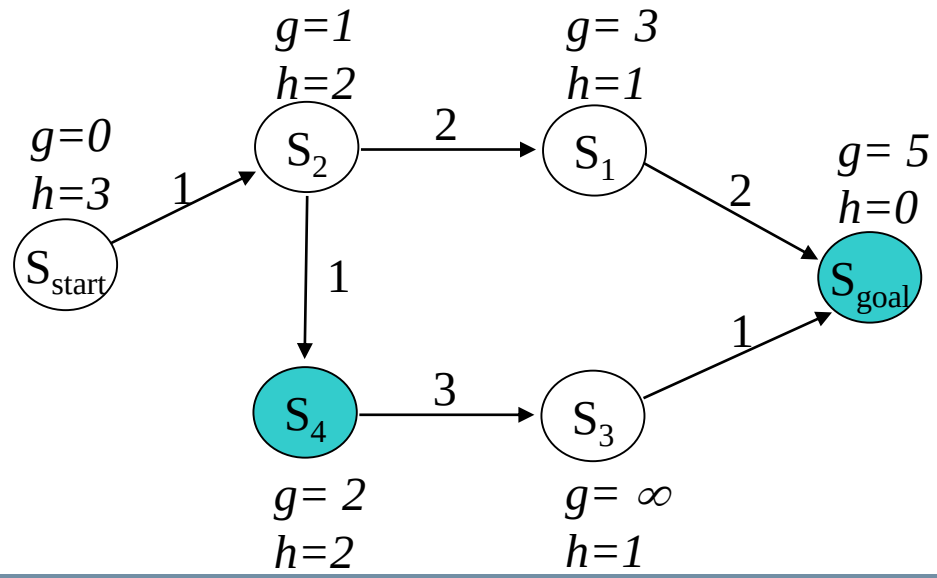


CLOSED = {s_{start}, s_2}
OPEN = {s_1,s_4}
next state to expand: s_1

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*;

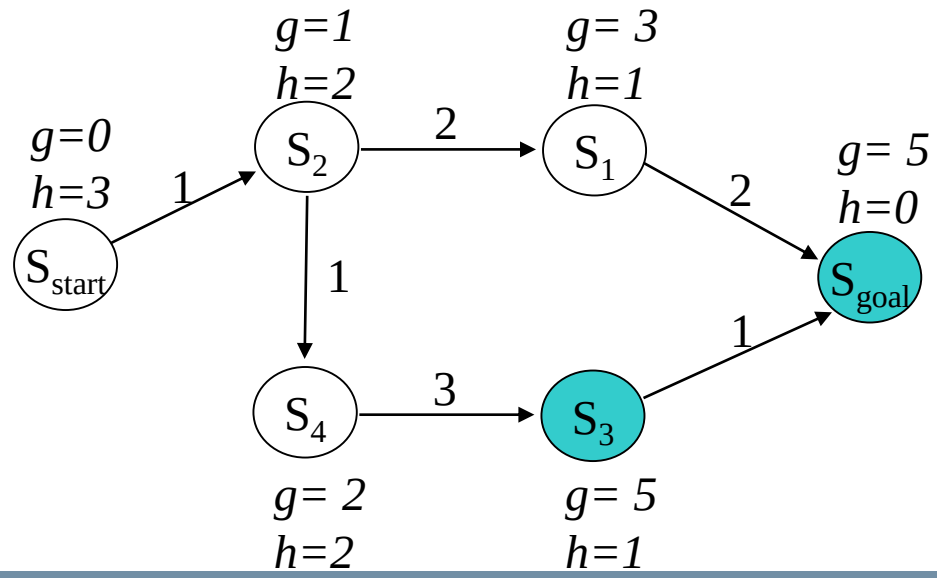


CLOSED = {s_{start}, s_2, s_1}
OPEN = {s_4,s_{goal}}
next state to expand: s_4

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}

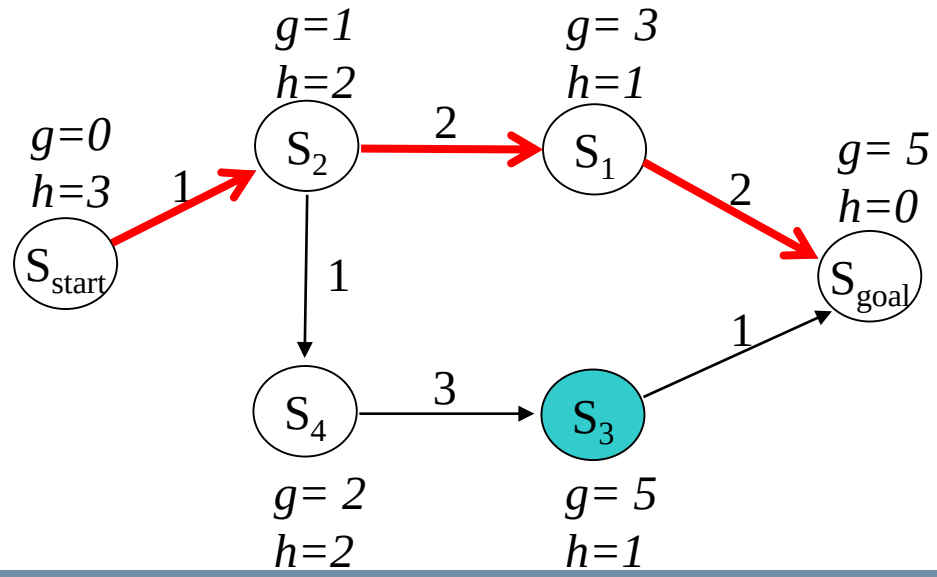
A* Search Algorithm

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}

DONE!



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*

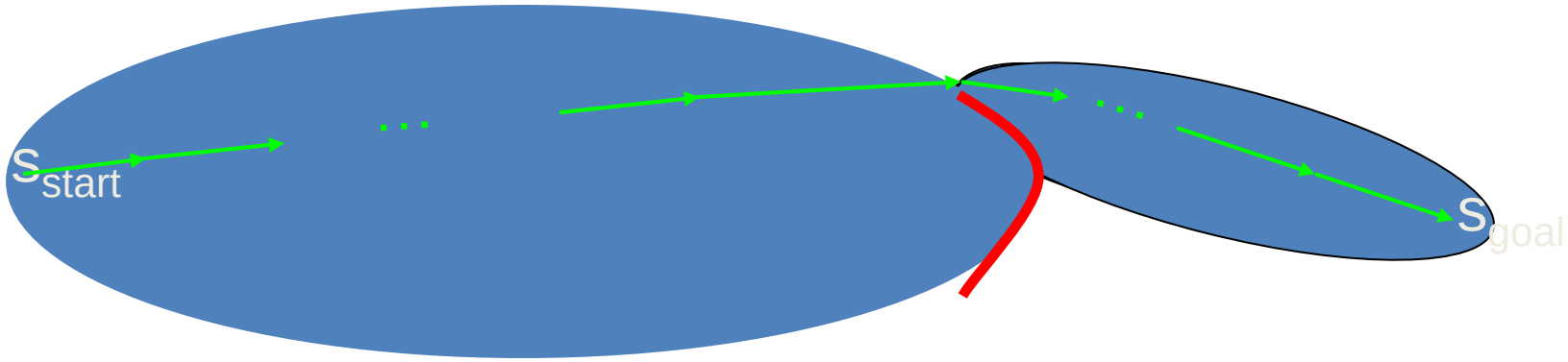
J. Pearl., "Heuristics: Intelligent Search Strategies for Computer Problem Solving", *Addison-Wesley*, 1984

ughly)

A* Properties

Algorithms state expansion:

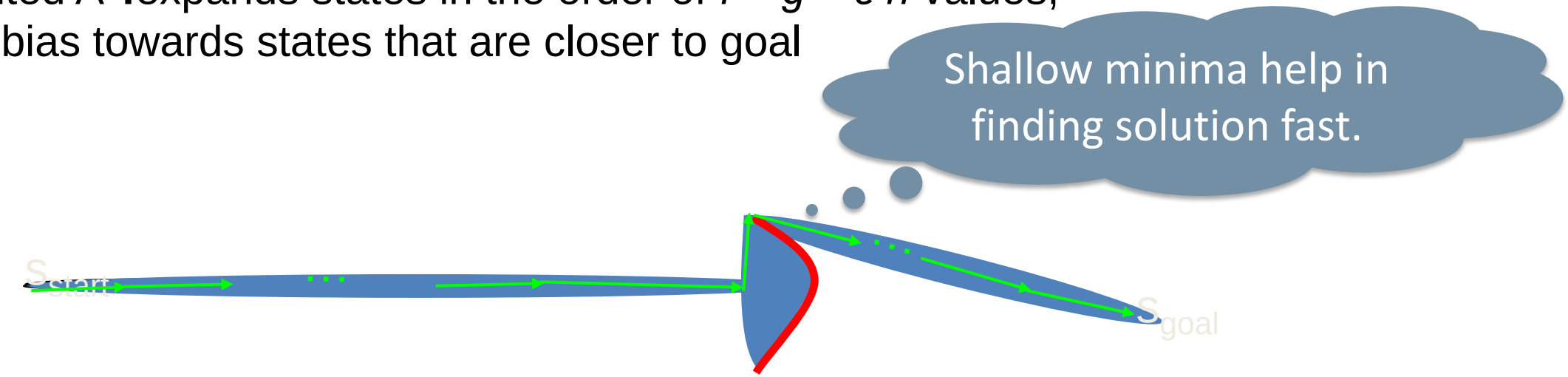
- Dijkstra's: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values



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



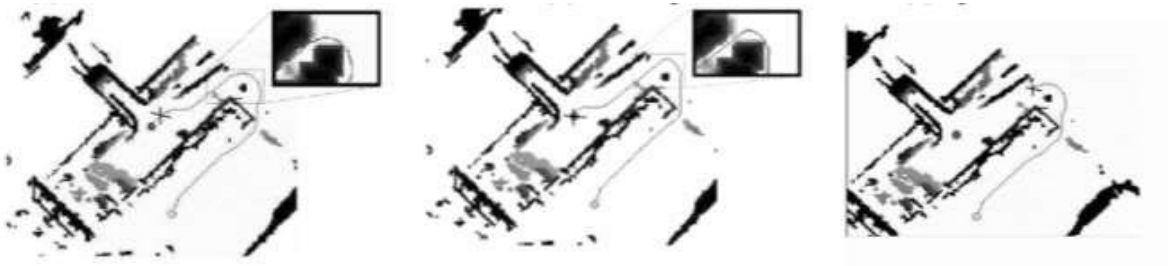
Other variations of A*

ARA* (Anytime Repairing A*)

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

D*/D*-Light

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



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*

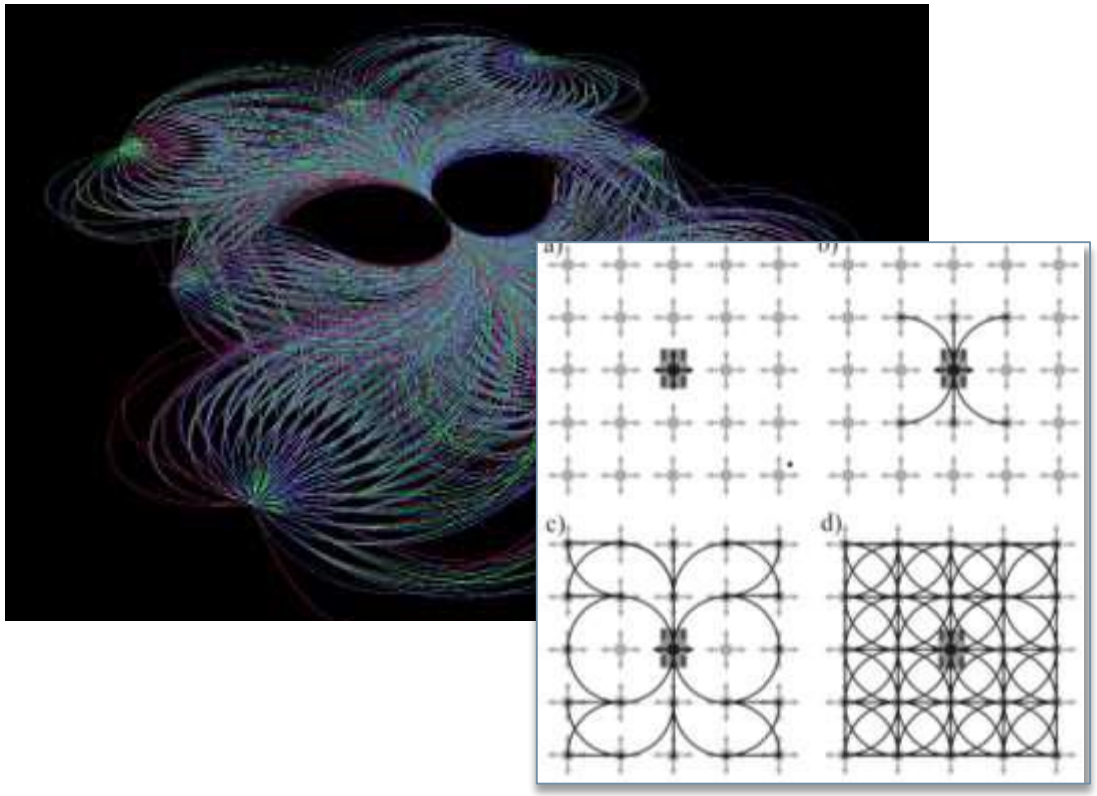
Anytime D* (D* + ARA*)

- Anytime graph-search re-using previous query

State-Lattice planning

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



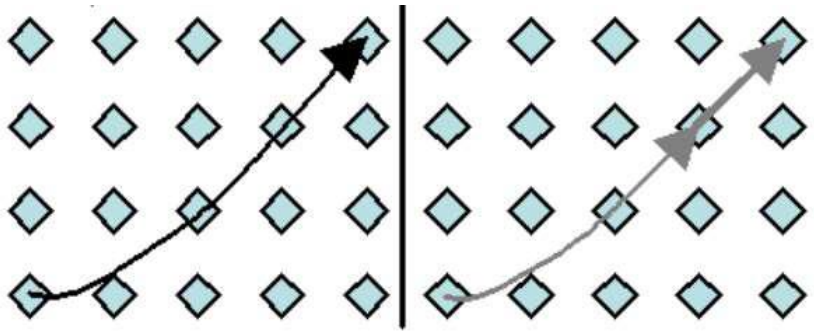
Pivtoraiko, M., & Kelly, A. (2005). Constrained Motion Planning in Discrete State Spaces. In *Field and Service Robotics pp.* 269–280.

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.

An example of State-Lattice «curse of dimensionality»

Discretization of the state-space for an Ackermann vehicle

- x = [-50.0, 50.0], 0.2 m resolution → 501 states
- y = [-50.0, 50.0], 0.2 m resolution → 501 states
- heading θ = [-π, +π], 0.1 rad resolution → ~64 states
- steering angle ϕ = [-0.6, +0.6], 0.1 rad resolution → 13 states

Overall state lattice size

- S = $x \times y \times \theta \times \phi$ = 208′832′832 states
- Impossible to search exhaustively (during online operation)

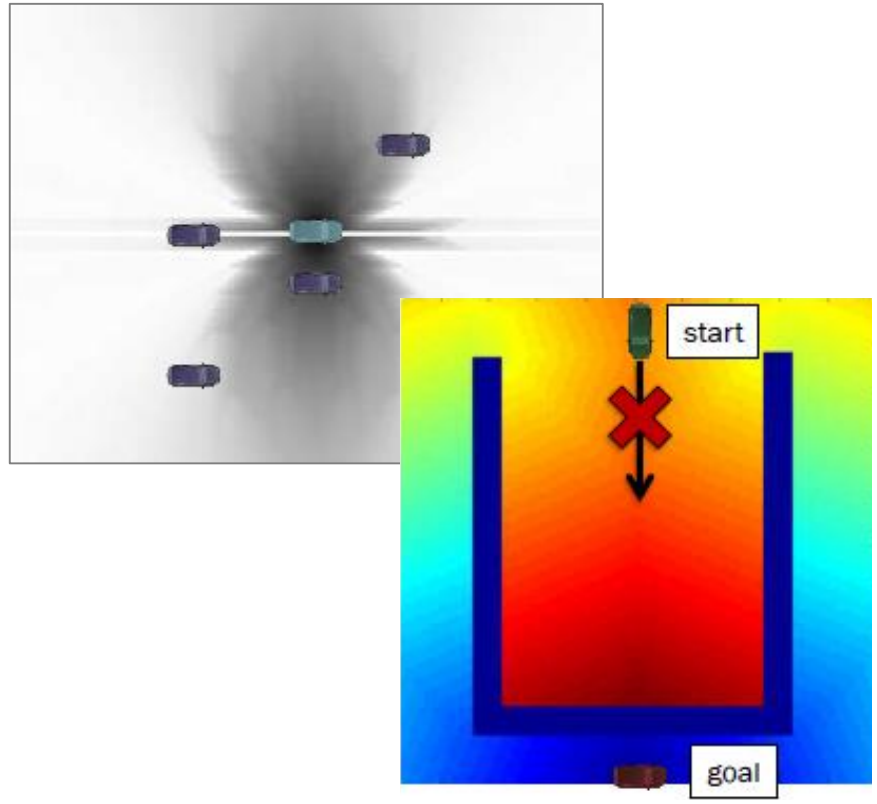
Opportunities for speed-up:

- Well-informed heuristics
- Multi-resolution state-lattices

Well informed heuristics for State-Lattice search

The closer the heuristics estimates the true cost-to-go, the faster the search

- h_1: non-holonomic-without-obstacles
  - Expensive, but can be computed offline
  - Better estimate of true cost in vicinity of goal
- h_2: holonomic-with-obstacles
  - Cheap to compute/online
  - Better estimate of true cost far from the goal
- $h = \max(h_1, h_2)$



Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly

- Define this as an optimization problem

$$u^* = \underset{\boldsymbol{u}}{\operatorname{argmin}} \left(\Phi(\boldsymbol{u}, t_f) + \int_{t_o}^{t_f} L(\boldsymbol{x}, \boldsymbol{u}, t) dt \right)$$

- Convert functional optimization problem into parametric one by restriction to a parametrized function of the inputs

$$p^* = \underset{\boldsymbol{p}}{\operatorname{argmin}} \left(\Phi(\boldsymbol{u}(\boldsymbol{p}), t_f) + \int_{t_o}^{t_f} L(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{p}), t) dt \right)$$

- Subject to: $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$
  $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$

Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly

- Often convenient to parametrize problem over path length

$$p^* = \underset{\boldsymbol{p}}{\operatorname{argmin}} \left(\Phi\big(\boldsymbol{u}(\boldsymbol{p}, s_f), s_f\big) + \int_0^{s_f} L(\boldsymbol{x}, \boldsymbol{u}(\boldsymbol{p}), s) ds \right)$$

- E.g., with polynomial curvature input

$$\boldsymbol{u}(\boldsymbol{p}, s) = \kappa(\boldsymbol{p}, s) = \boldsymbol{p}_0 + \boldsymbol{p}_1 s + \boldsymbol{p}_2 s^2 + \cdots$$

- Subject to: $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$
  $\boldsymbol{x}(t_f) = \boldsymbol{x}_f$

- Design variables for optimization problem $\boldsymbol{q} = \left[\boldsymbol{p}, s_f\right]^T$

Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly

- Cartesian coordinates x(s), y(s) cannot be computed in closed form for an Ackermann system model even for simple inputs

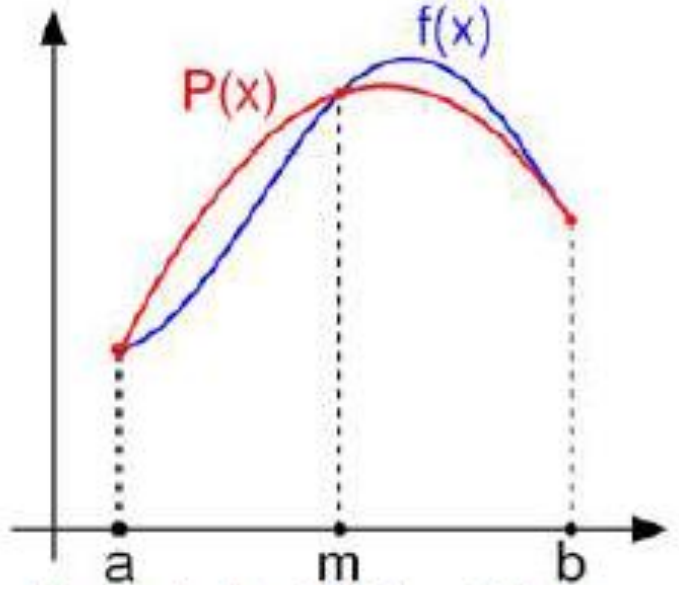
$$x(s) = \int_0^s \cos\big(\theta(\tau)\big)\, d\tau$$

$$y(s) = \int_0^s \sin\big(\theta(\tau)\big)\, d\tau$$

$$\theta(s) = \int_0^s \kappa(\tau)\, d\tau$$

$$\kappa(s) = \boldsymbol{p}_0 + \boldsymbol{p}_1 s + \boldsymbol{p}_2 s^2 + \cdots$$

- Solved by numerical approximation of integrals



Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly

- For 3rd order curvature polynomials, the problem has a unique solution

$$\boldsymbol{q} = \left[\boldsymbol{p}_0 = \kappa_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, s_f\right]^T$$

- Constraints on Δx, Δy, $\Delta\theta$, κ_0, κ_f can be included in the optimization criterion

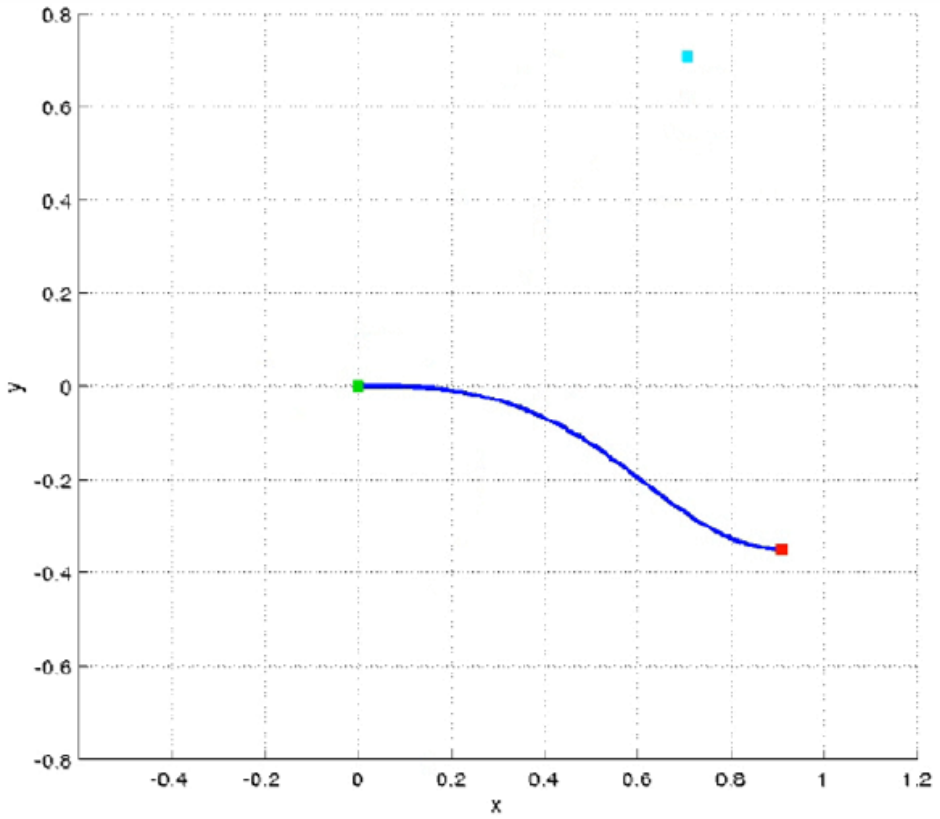
$$\boldsymbol{p}^* = \underset{\boldsymbol{p}}{argmin} \begin{bmatrix} x_f - x(s_f) \\ y_f - y(s_f) \\ \theta_f - \theta(s_f) \\ \kappa_f - \kappa(s_f) \end{bmatrix}$$

- Gradient-based nonlinear optimization to obtain remaining design variables

Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly



initial state $[x = 0, y = 0, \theta = 0, \kappa = 0]^T$

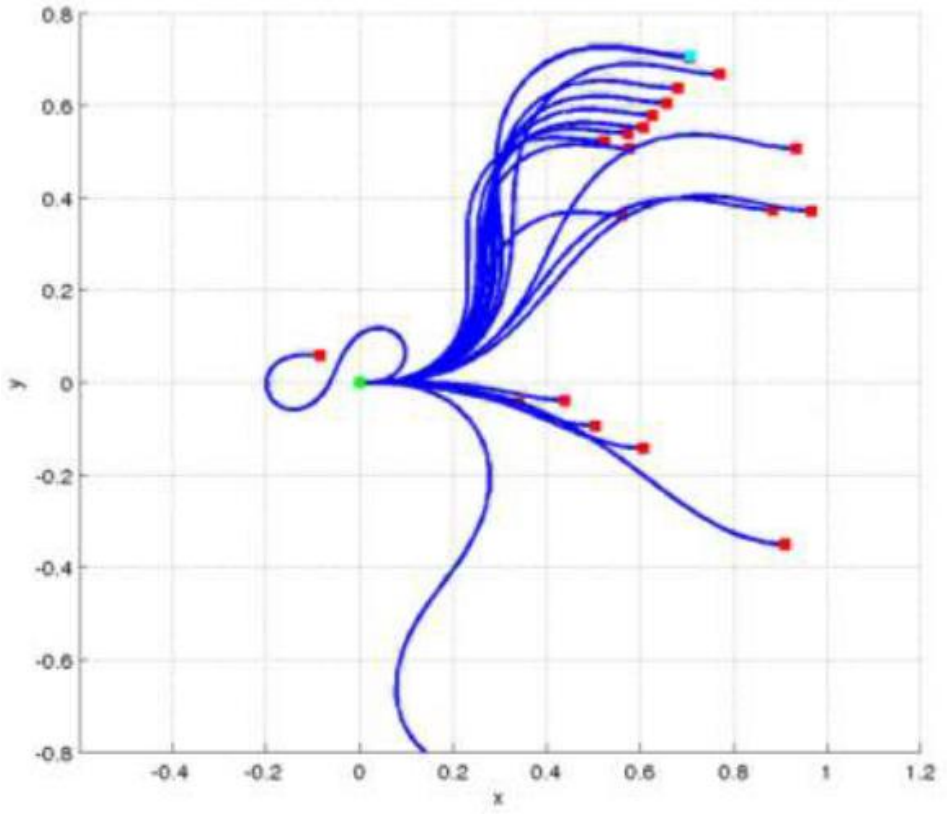
desired state $[x = 4, y = 4, \theta = 0, \kappa = 0.78]^T$

terminal state

Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

Two-state boundary value problem

Control set generation for state-lattice search requires to obtain motion primitives connect two states in state space exactly



initial state $[x = 0, y = 0, \theta = 0, \kappa = 0]^T$

desired state $[x = 4, y = 4, \theta = 0, \kappa = 0.78]^T$

terminal state

Kelly, A., Nagy, B., "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control", *The International Journal of Robotics Research*, 2003

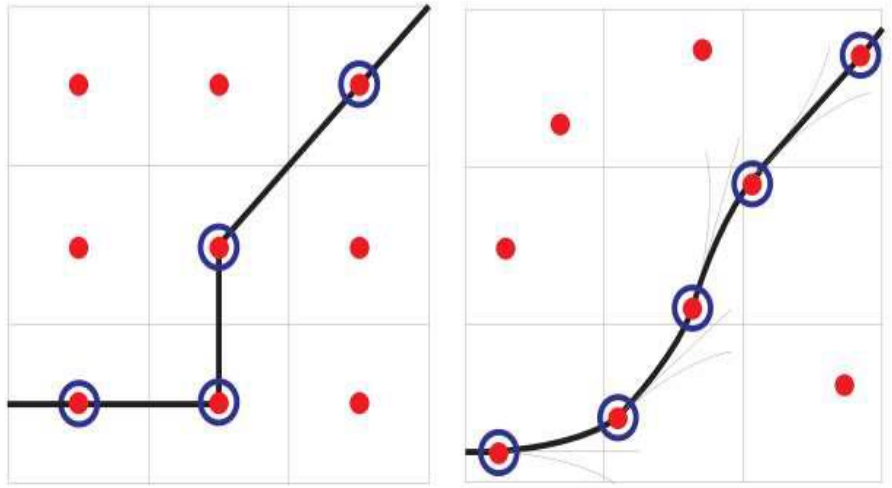
Hybrid A*

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

No completeness guarantee any more

- Changing reachable statespace
- Pruning of continuous-state branches



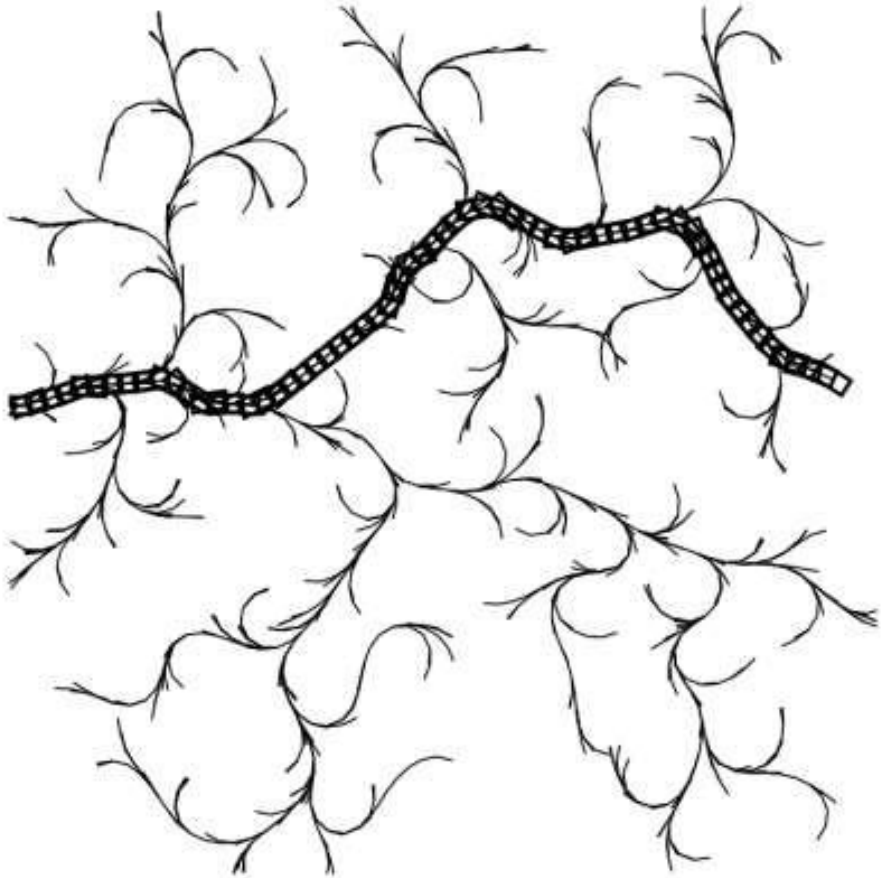
Produces inherently driveable paths and above mentioned shortcomings almost never happen in practice

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

Sampling based planners

Sample based methods incrementally construct a search tree by gradually improving the resolution

- 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)
1  G.init(q_0);
2  for i = 1 to k do
3      G.add_vertex(α(i));
4      q_n ← NEAREST(S(G), α(i));
5      G.add_edge(q_n, α(i));
```

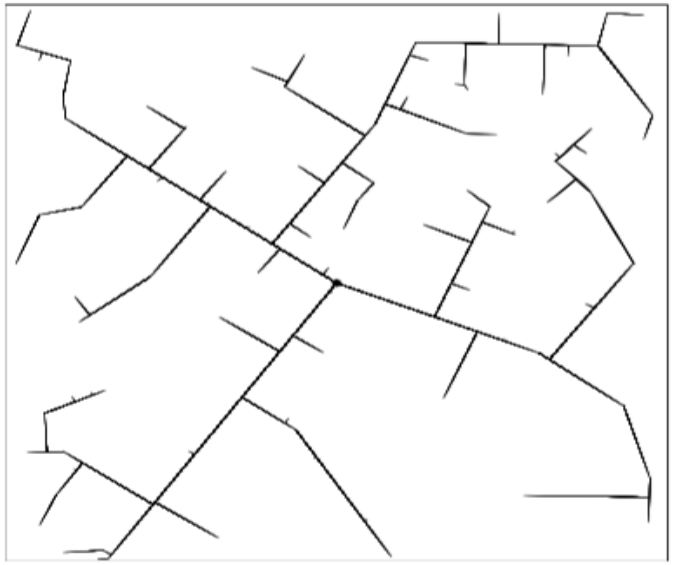
α: *D*nse sequence of samples in C

$\alpha(i)$: i^{th} sample of the sample sequence

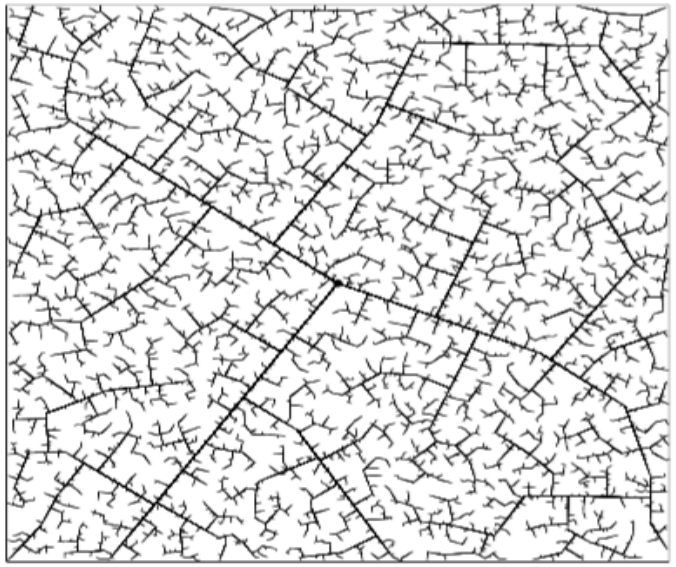
$G(V, E)$: topological representation of RDT

$S \subset C_{free}$: Set of points reached by G

$S = \bigcup_{e \in E} e([0,1])$ where $e([0,1]) \in C_{free}$



45 iterations

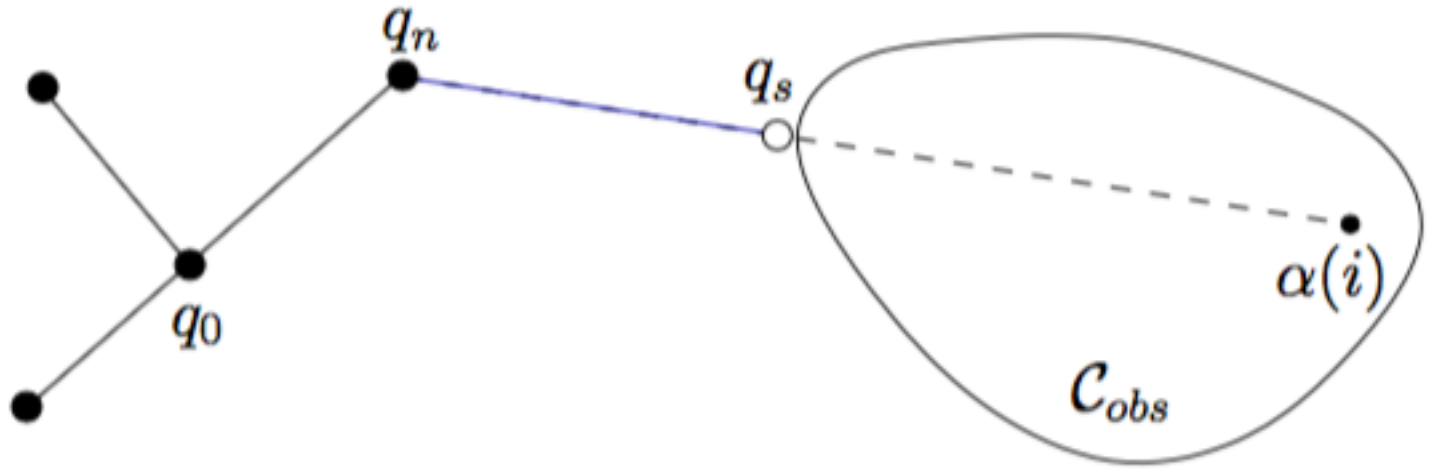


2345 iterations

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

Rapidly Exploring Dense Trees (RDTs)

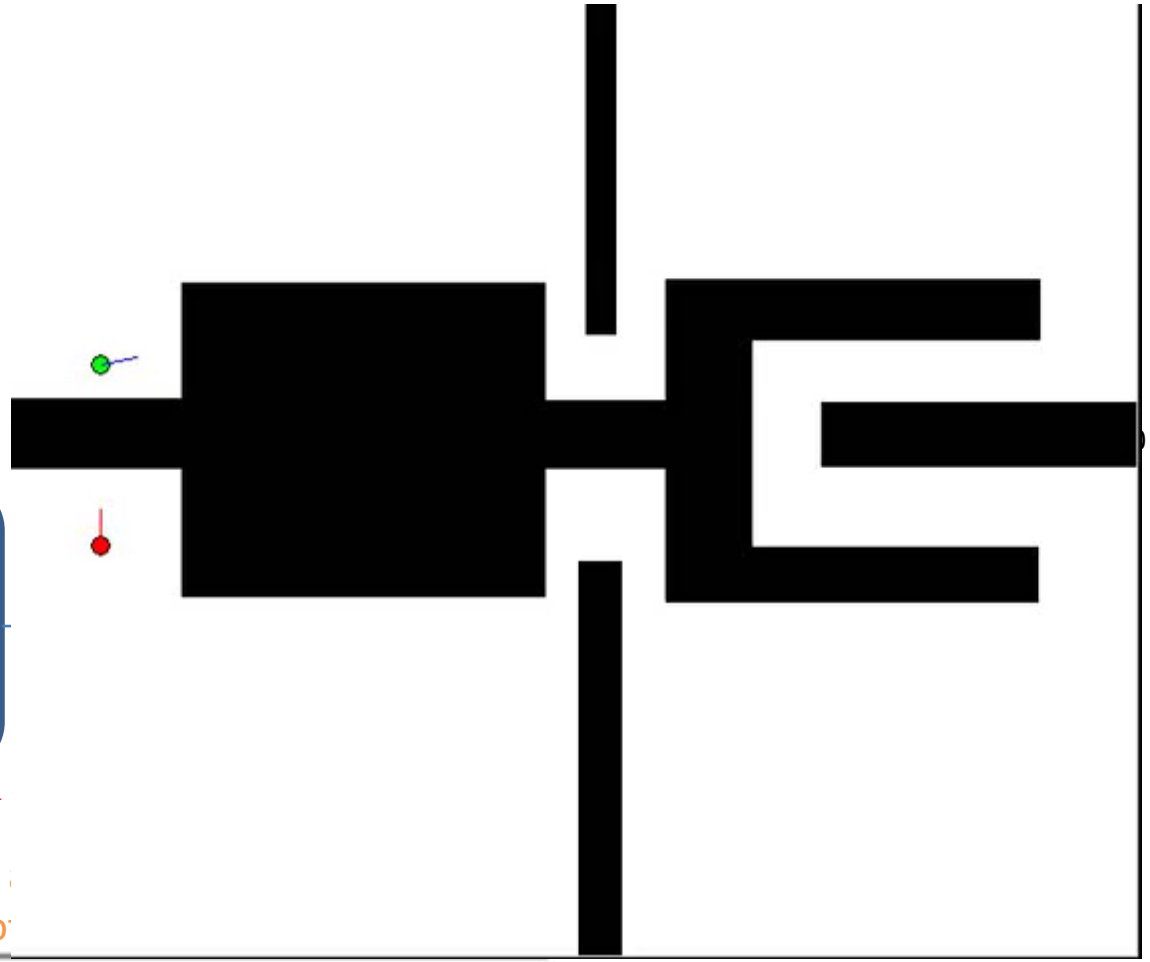
```
RDT(q_0)
1  G.init(q_0);
2  for i = 1 to k do
3      q_n ← NEAREST(S, α(i));
4      q_s ← STOPPING-CONFIGURATION(q_n, α(i));
5      if q_s ≠ q_n then
6          G.add_vertex(q_s);
7          G.add_edge(q_n, q_s);
```



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)
 1   T_a.init(q_I); T_b.init(q_G);
 2   for i = 1 to K do
 3       q_n ← NEAREST(S_a, α(i));
 4       q_s ← STOPPING-CONFIGURATION(q_n, α(i));
 5       if q_s ≠ q_n then
 6           T_a.add_vertex(q_s);
 7           T_a.add_edge(q_n, q_s);
 8           q'_n ← NEAREST(S_b, q_s);
 9           q'_s ← STOPPING-CONFIGURATION(q'_n, q_s);
10           if q'_s ≠ q'_n then
11               T_b.add_vertex(q'_s);
12               T_b.add_edge(q'_n, q'_s);
13           if q'_s = q_s then return SOLUTION;
14       if |T_b| > |T_a| then SWAP(T_a, T_b);
15   return FAILURE
```

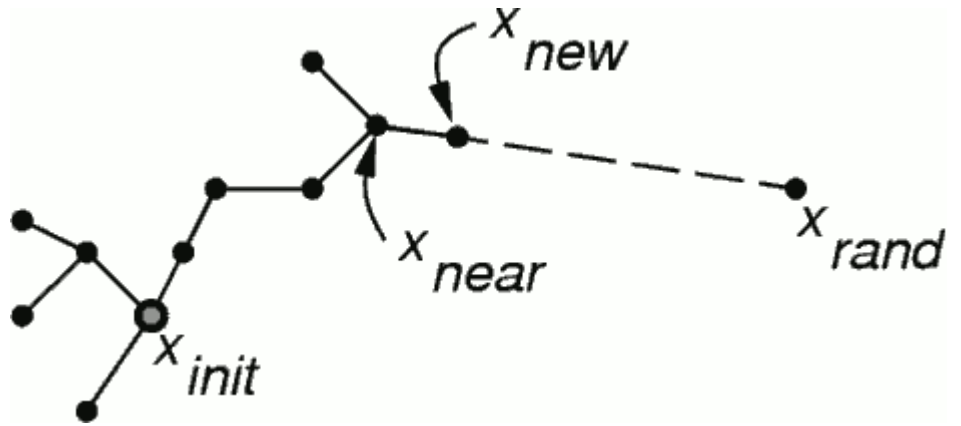


Two trees
than the o

Rapidly Exploring Random Trees (RRT)

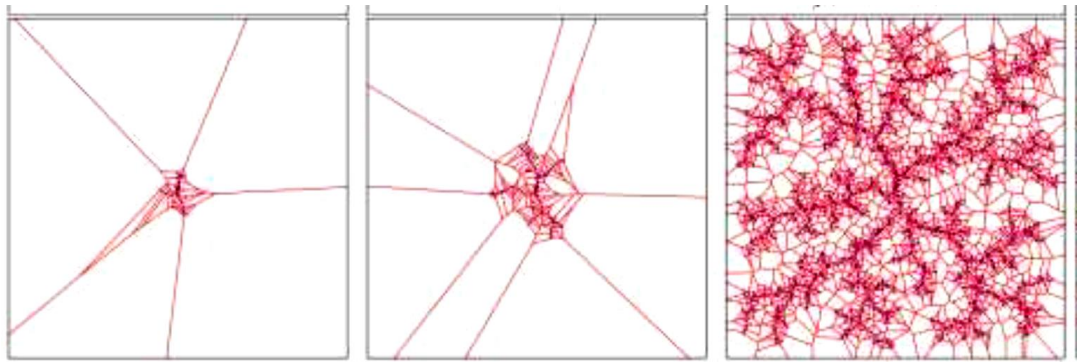
Algorithm 1 $\tau = (V, E) \leftarrow \mathrm{RRT}(z_{init})$

1: $\tau \leftarrow$ **InitializeTree**();
2: $\tau \leftarrow$ **InsertNode**$(\emptyset, z_{init}, \tau)$;
3: **for** $i = 1$ to $i = N$ **do**
4: $\quad z_{rand} \leftarrow$ **Sample**(i);
5: $\quad z_{nearest} \leftarrow$ **Nearest**(τ, z_{rand});
6: $\quad (x_{new}, u_{new}, T_{new}) \leftarrow$ **Steer**$(z_{nearest}, z_{rand})$;
7: $\quad$ **if ObstacleFree**(x_{new}) **then**
8: $\quad\quad \tau \leftarrow$ **InsertNode**(z_{new}, τ);
9: $\quad$ **end if**
10: **end for**
11: **return** τ



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



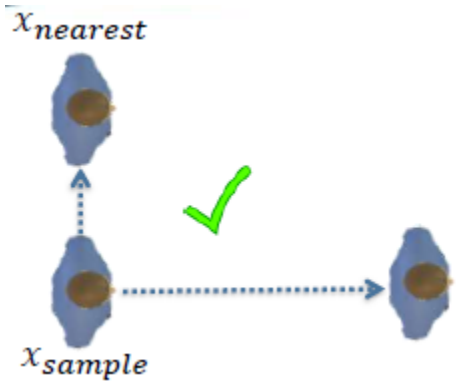
LaValle, S. M., Kuffner, J., "Randomized Kinodynamic Planning", *IEEE International Conference on Robotics and Automatio*, 1999

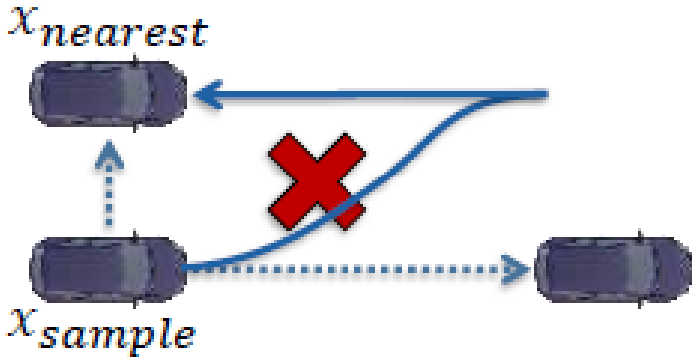
RRT pros and cons

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

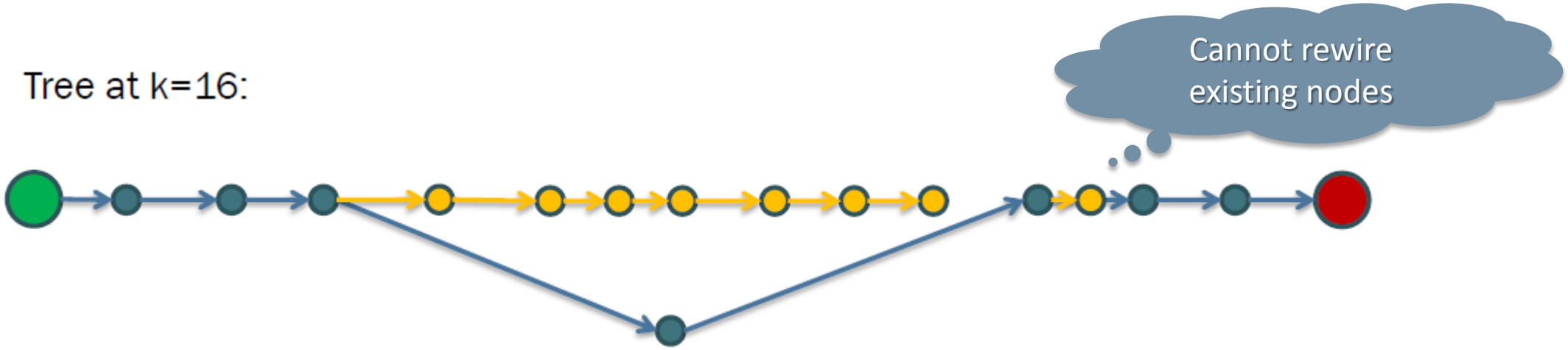




RRT exstensions and RTT*

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…

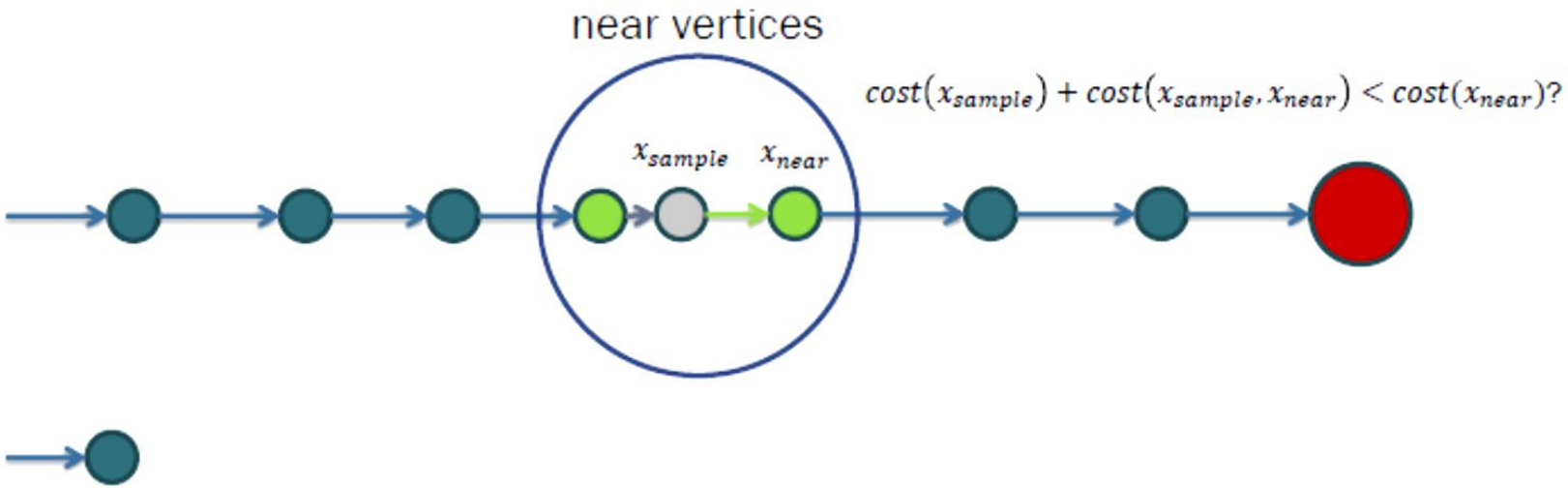


S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning", Robotics: Science and Systems, 2010

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…



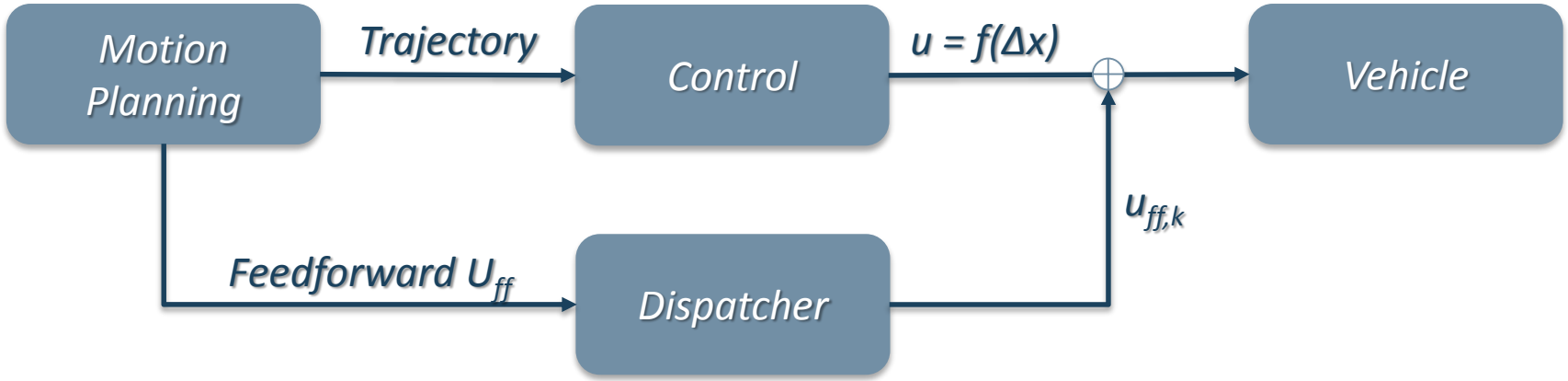
S. Karaman and E. Frazzoli, "Incremental sampling-based algorithms for optimal motion planning", Robotics: Science and Systems, 2010

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	

Motion planning vs control

In unmanned vehicles motion planning and control are highly coupled



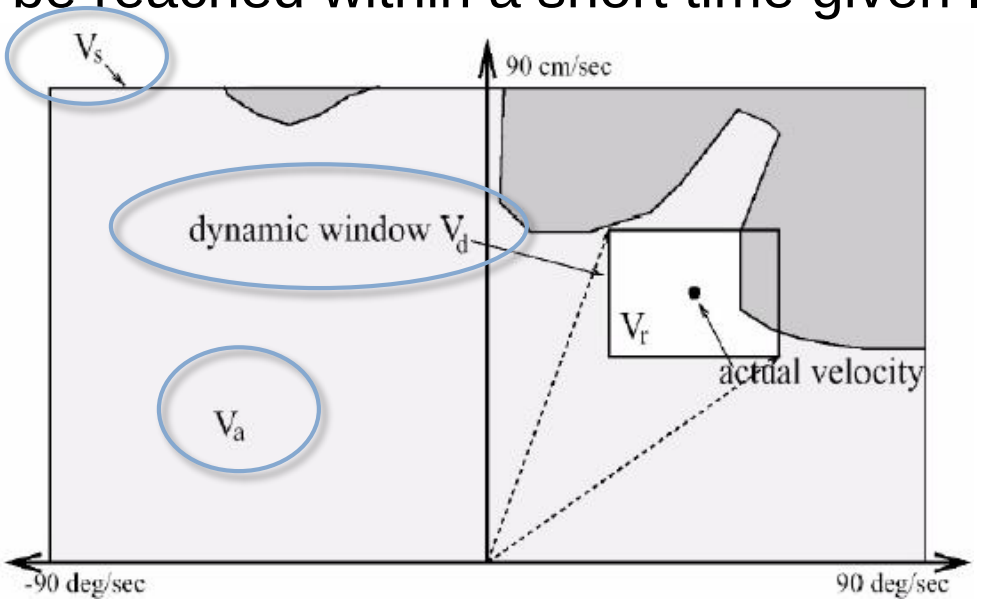
Control mainly responsible to account for errors built up in between planning cycles:

- Cope with modeling inaccuracies in between planning steps
- Regulate vehicle onto trajectory
- Compute valid vehicle control input $\boldsymbol{u}$
- Error states are frequently reset due to re-planning

The Dynamic Window Approach (DWA)

The kinematics of the vehicle is considered via local search in velocity space:

- Consider pairs $V_s=(v,\omega)$ of translational and rotational speeds
- A pair $V_a=(v, \omega)$ is considered *admissible*, if it allows to stop before the closest obstacle on the corresponding curvature.
- 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

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

Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. IEEE *RAM*, *4*(1), 23–33.

How to choose the best (v,ω) in DWA?

Steering commands are chosen maximizing a heuristic navigation function:

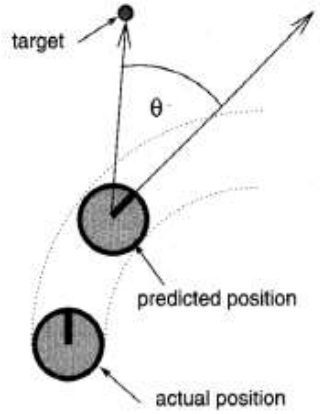
- Minimize the travel time by "driving fast in the right direction"
- Planning restricted to V_r space [Fox, Burgard, Thrun '97]

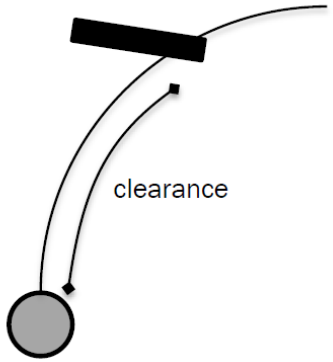
$$G(v,\omega) = \sigma(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot velocity(v,\omega))$$

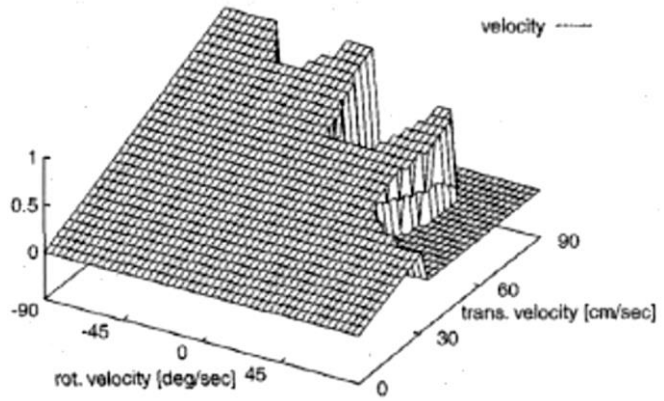
Alignment with target direction

Distance to closest obstacle intersecting with curvature

Forward velocity of the robot







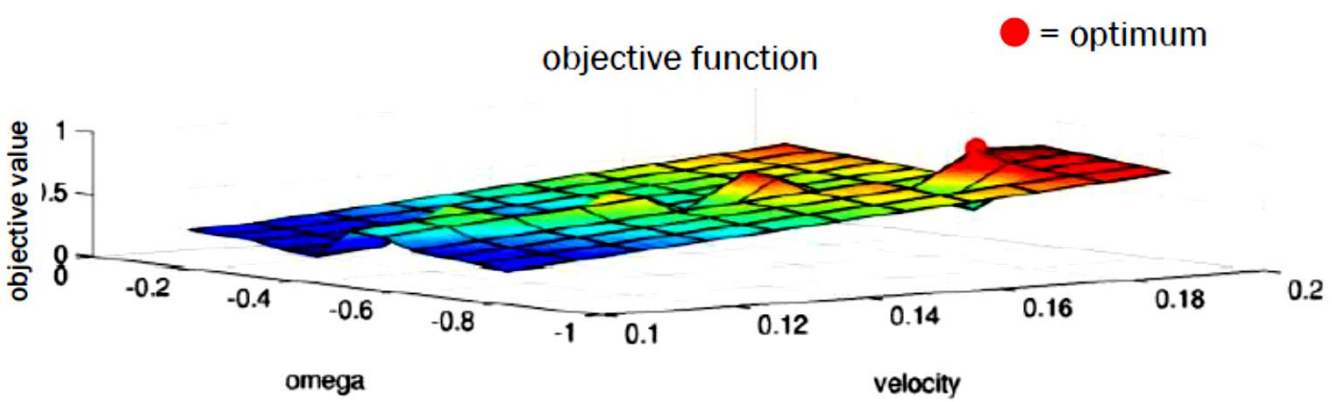
How to choose the best (v,ω) in DWA?

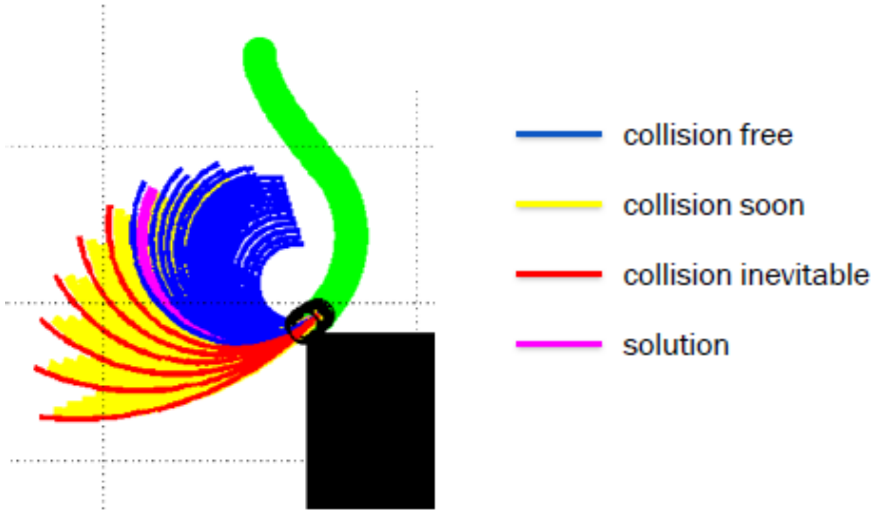
Steering commands are chosen maximizing a heuristic navigation function:

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

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

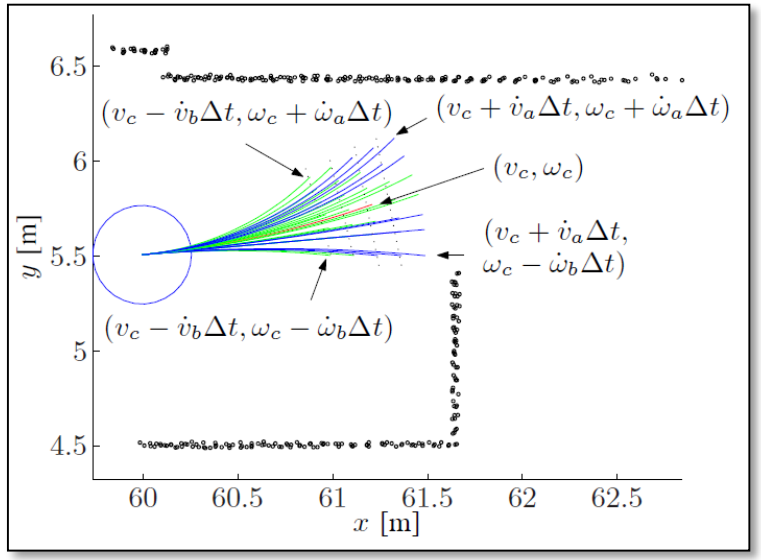
- Solution found by exhaustive evaluation of the discretized search space





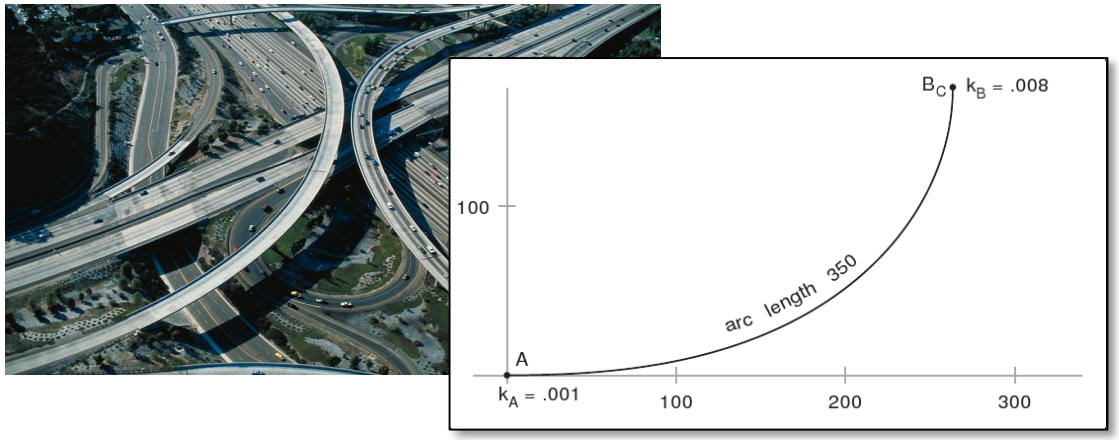
The DWA Algorithm

1. Discretely sample robot control space
2. For each sampled velocity, perform forward simulation from current state 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



What about non circular kinematics?

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



DWA pros and cons

Pros	Cons
Fast to compute (<10ms)	Local DWA can get stuck in local minima
Incorporates vehicle model	Constant input assumption often too simplistic
Intuitive parameter tuning	
The Global DWA deals well with local minima	

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

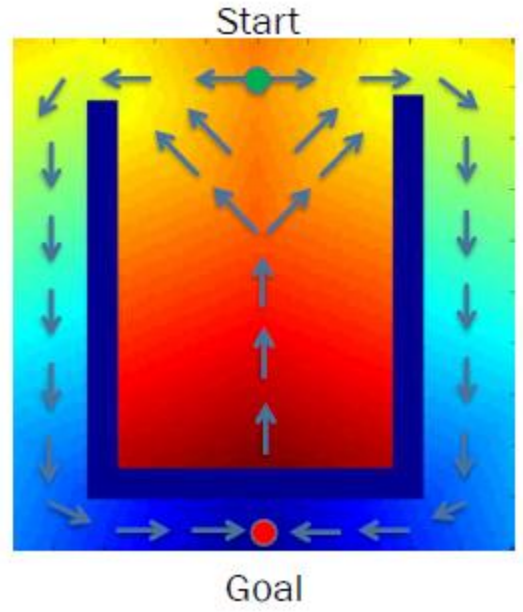
Forward robot velocity

Follows global path

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

Cost to reach the goal

Goal nearness



Wrap-up slide on “Trajectory planning in land vehicles ”

What should remain from this lecture?

- What is trajectory planning
- How graph-based methods such as A* work
- How randomized methods such as RRT work
- How the Dynamic Window Approach navigation method works

References

- LaValle, S. M. (2006). Planning algorithms. Cambridge university press.
- Latombe, J-C (2012). Robot motion planning. Vol. 124. Springer Science & Business Media.
- Fox, D., Burgard, W., & Thrun, S. (1997). The dynamic window approach to collision avoidance. IEEE RAM, 4(1), 23–33.