Introduction to
Mobile Robotics

Robot Motion Planning

Wolfram Burgard, Cyrill Stachniss,
Maren Bennewitz, Kai Arras

UNI FREIBURG

Slides by Kai Arras Last update July 2011
With material from S. LaValle, JC. Latombe, H. Choset et al., W. Burgard

Robot Motion Planning

Contents

- Introduction
- Configuration Space
- Combinatorial Planning
- Sampling-Based Planning
- Potential Fields Methods
- A*, Any-Angle A*, D*/D* Lite
- Dynamic Window Approach (DWA)
- Markov Decision Processes (MDP)

Robot Motion Planning

J.-C. Latombe (1991):

"…eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world."

Goals

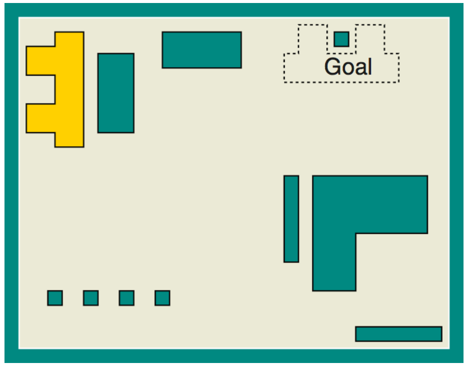
- Collision-free trajectories
- Robot should reach the goal location as fast as possible

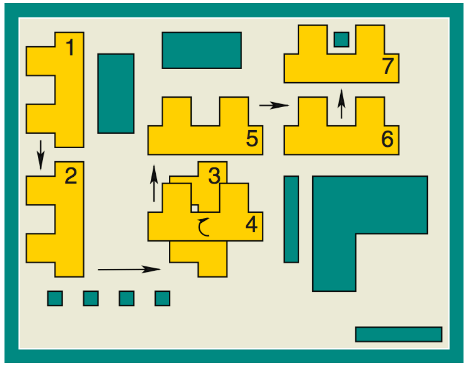
Problem Formulation

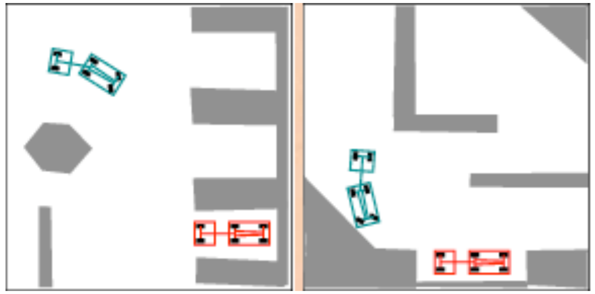
- The **problem of motion planning** can be stated as follows. Given:
  - A **start** pose of the robot
  - A desired **goal** pose
  - A geometric description of the **robot**
  - A geometric description of the **world**

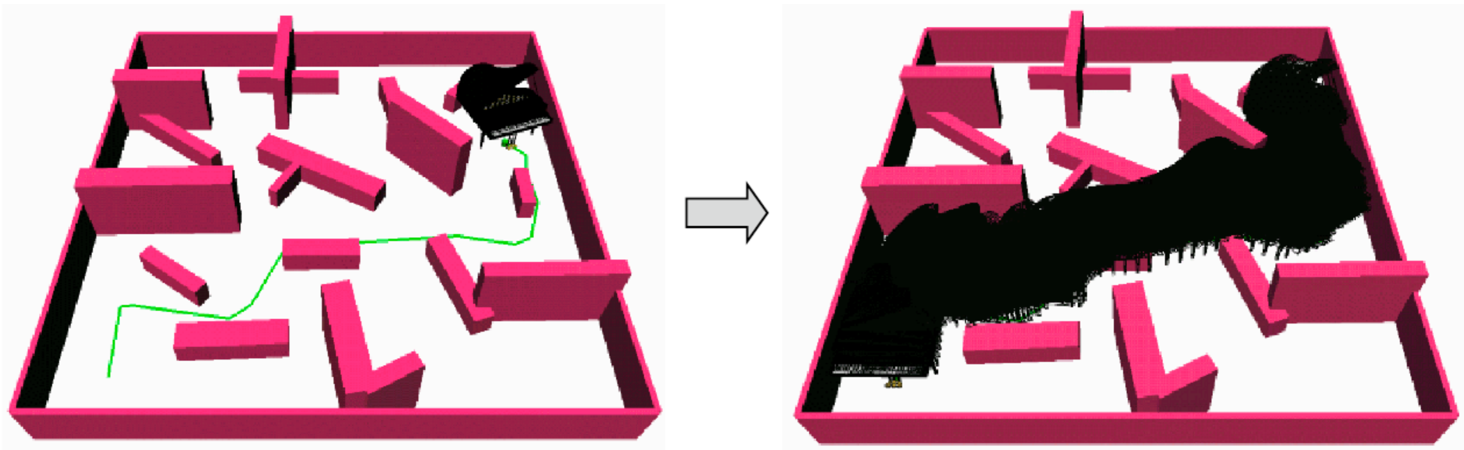
- Find a path that moves the robot gradually from **start** to **goal** while **never touching** any obstacle

Problem Formulation









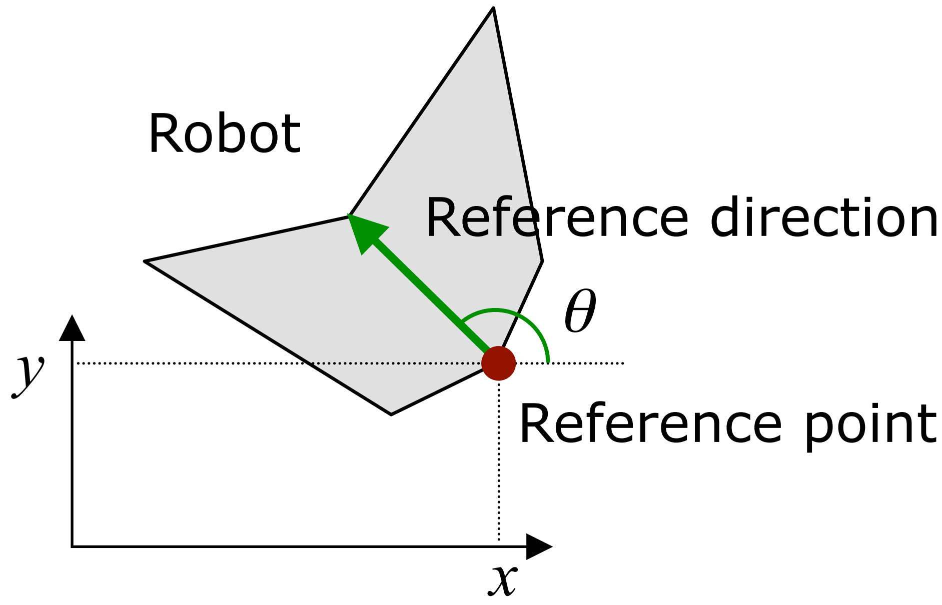
Motion planning is sometimes also called **piano mover's problem**

Configuration Space

- Although the motion planning problem is defined in the regular world, it lives in another space: the **configuration space**
- A robot configuration q is a specification of the positions of all robot points relative to a fixed coordinate system
- Usually a configuration is expressed as a **vector of positions** and **orientations**

Configuration Space

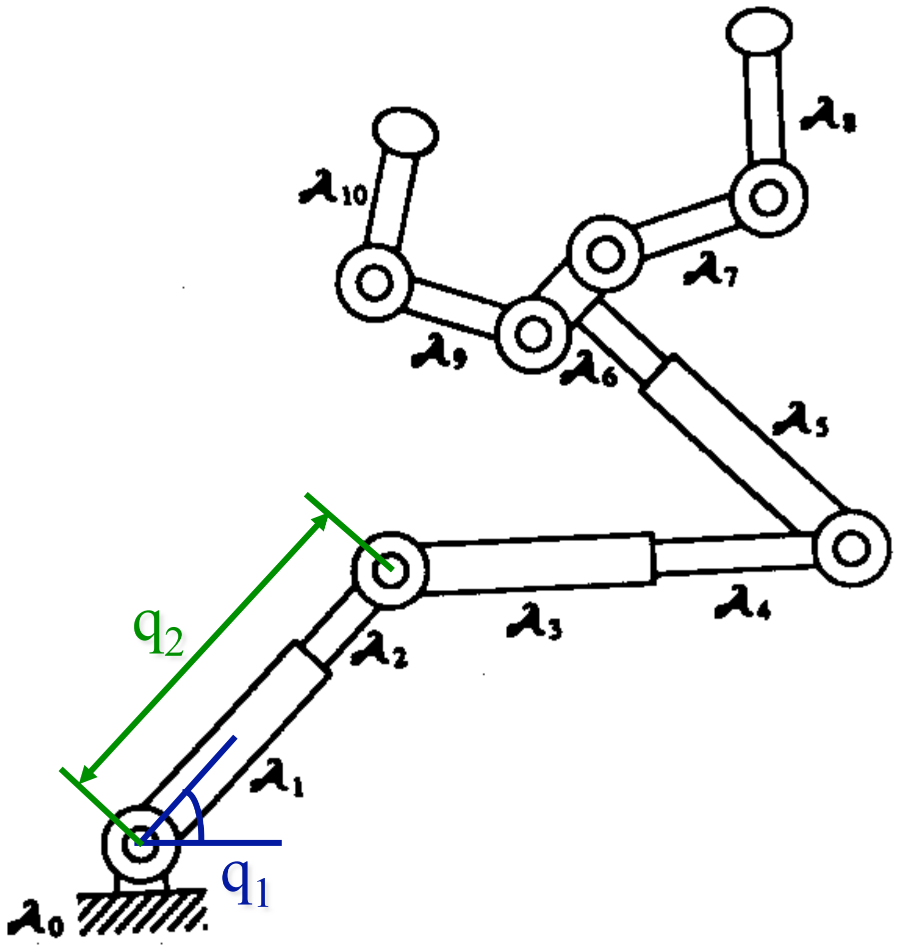
Rigid-body robot example



- 3-parameter representation: $q = (x,y,\theta)$
- In 3D, q would be of the form $(x,y,z,\alpha,\beta,\gamma)$

Configuration Space

Articulated robot example



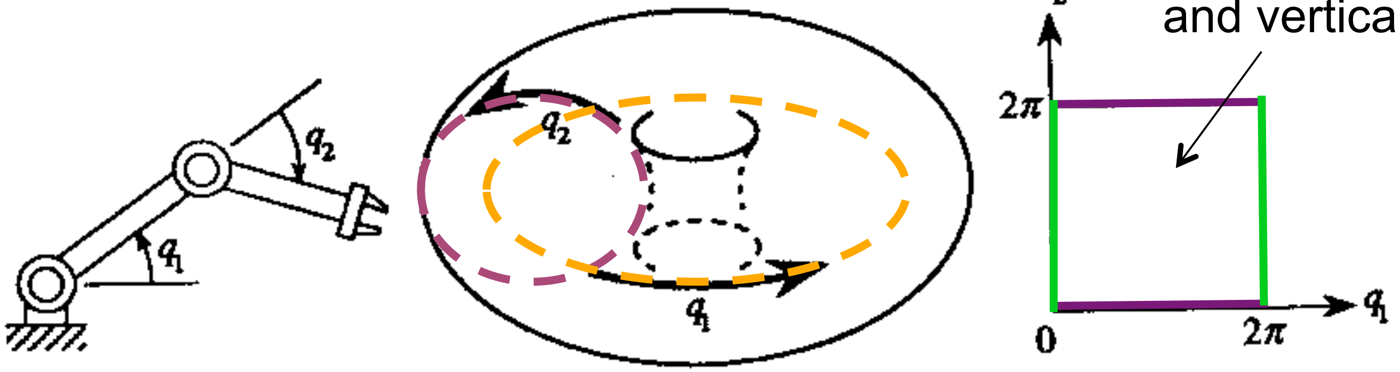
$$q = (q_1, q_2, \ldots, q_{10})$$

Configuration Space

- The configuration space (C-space) is the **space of all possible configurations**
- The topology of this space is usually **not** that of a Cartesian space
- The C-space is described as a **topological manifold**
- Example:

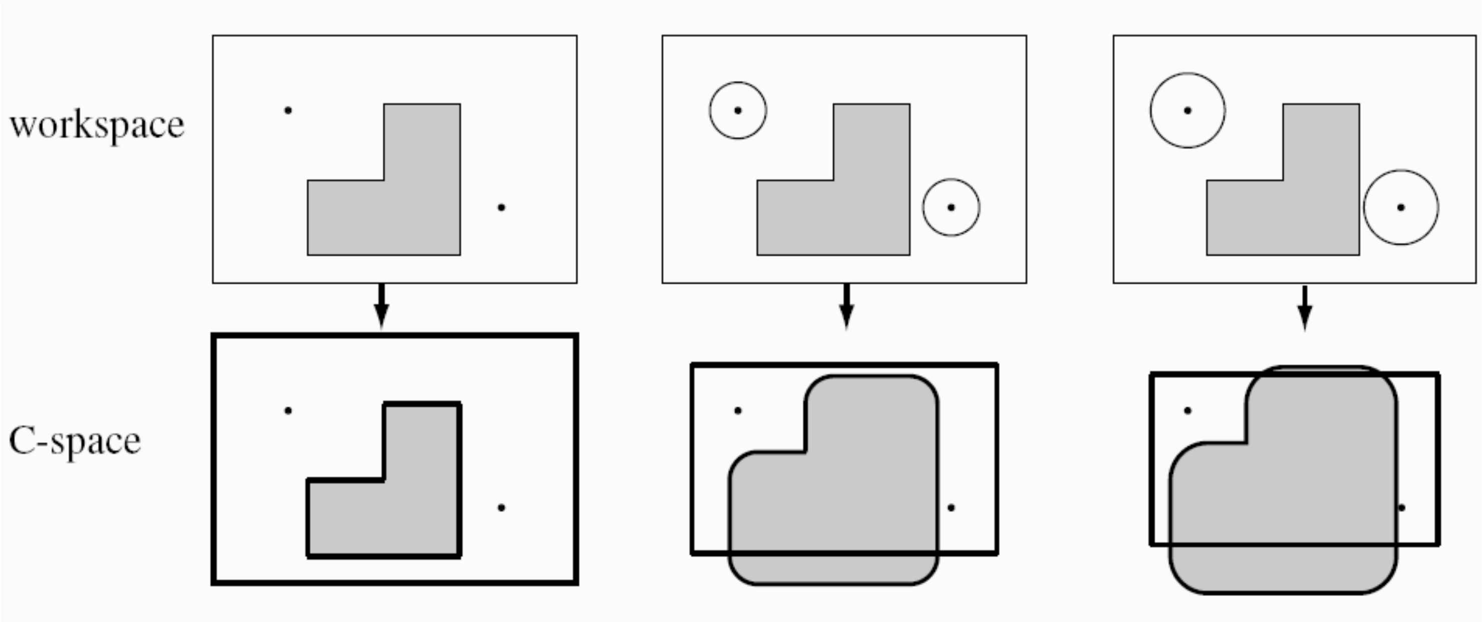
$C = S^1 \times S^1$

wraps horizontally
and vertically!



Configuration Space

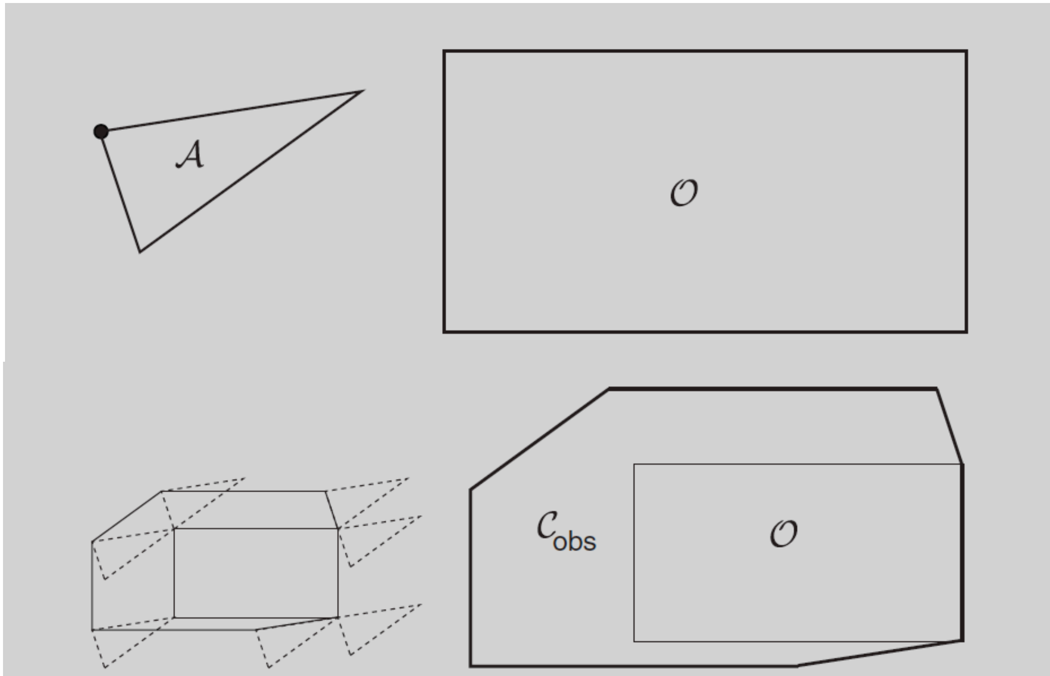
- Example: circular robot



- C-space is obtained by sliding the robot along the edge of the obstacle regions "blowing them up" by the robot radius

Configuration Space

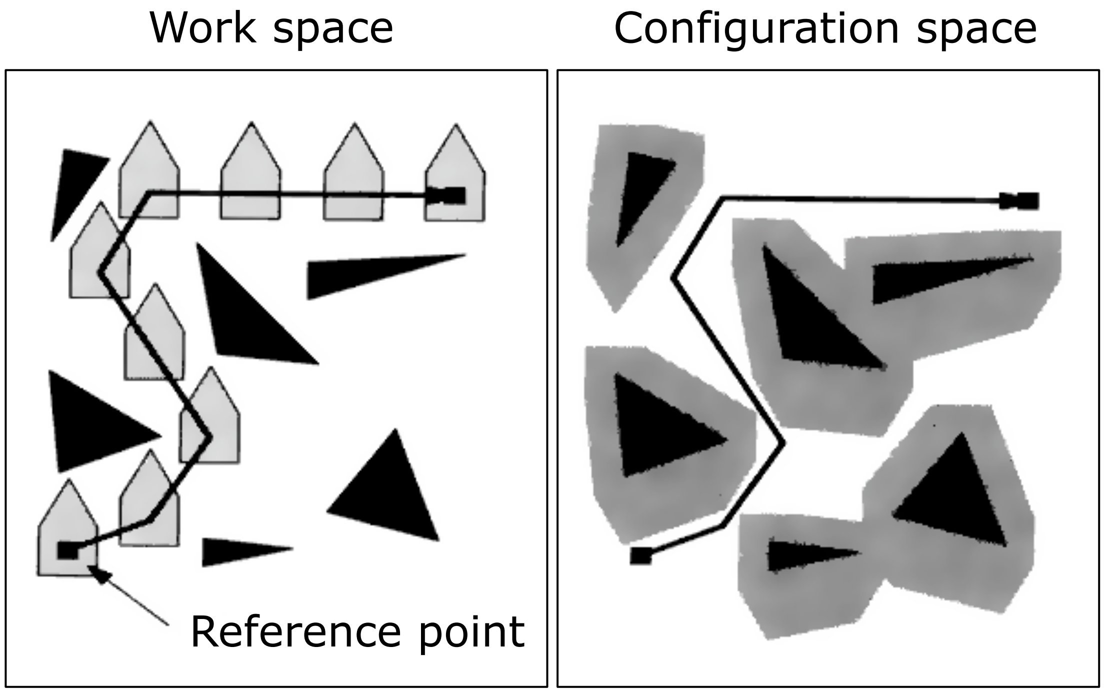
- Example: polygonal robot, translation only



- C-space is obtained by sliding the robot along the edge of the obstacle regions

Configuration Space

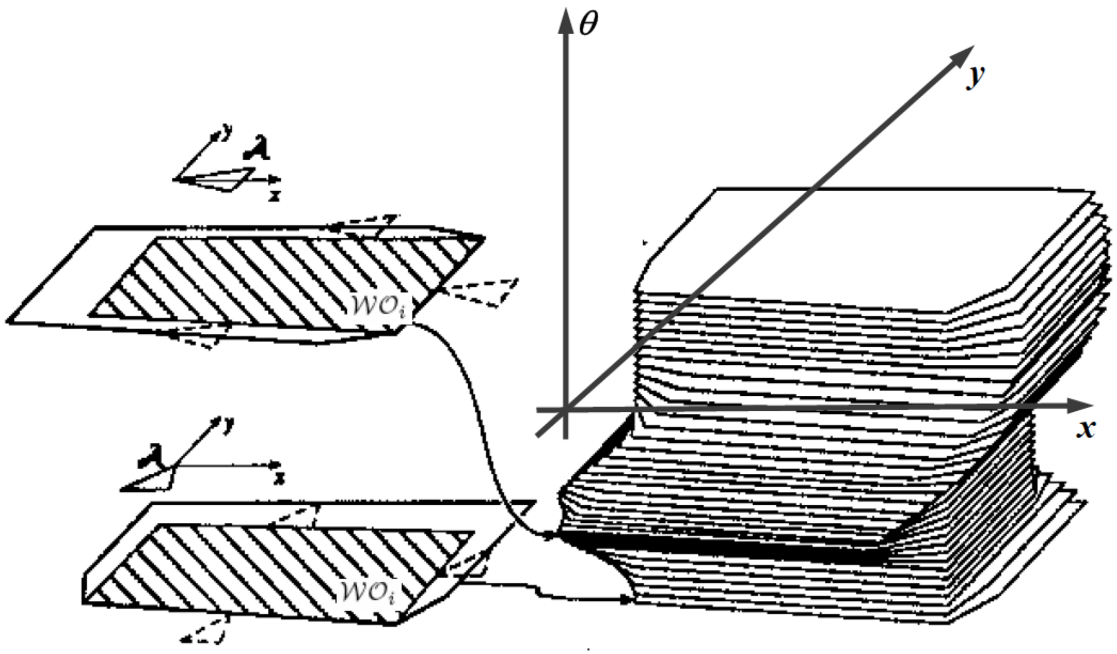
- Example: polygonal robot, translation only



- C-space is obtained by sliding the robot along the edge of the obstacle regions

Configuration Space

- Example: polygonal robot, trans+**rotation**



- C-space is obtained by sliding the robot along the edge of the obstacle regions in all orientations

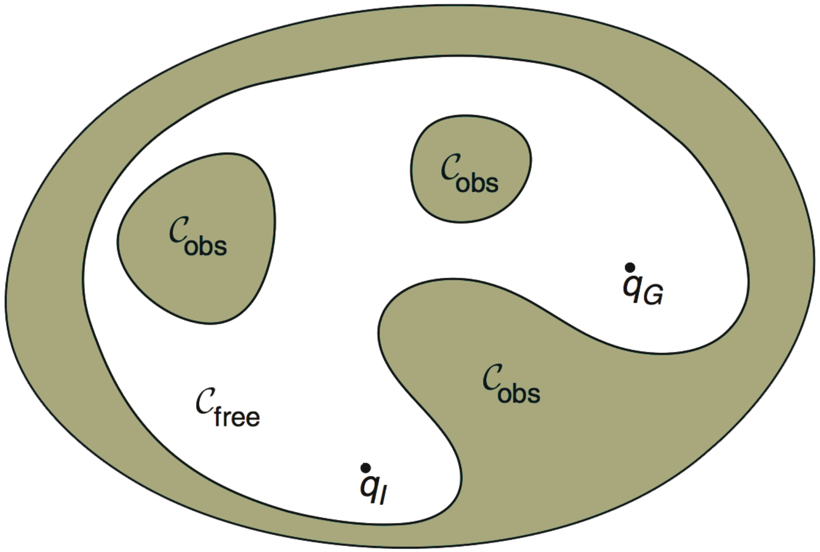
Configuration Space

Free space and **obstacle region**

- With $\mathcal{W} = \mathbb{R}^m$ being the work space, $\mathcal{O} \in \mathcal{W}$ the set of obstacles, $\mathcal{A}(q)$ the robot in configuration $q \in \mathcal{C}$

$$\begin{aligned} \mathcal{C}_{free} &= \{q \in \mathcal{C} \mid \mathcal{A}(q) \cap \mathcal{O} = \emptyset\} \\ \mathcal{C}_{obs} &= \mathcal{C} / \mathcal{C}_{free} \end{aligned}$$

- We further define
  q_I : start configuration
  q_G : goal configuration



Configuration Space

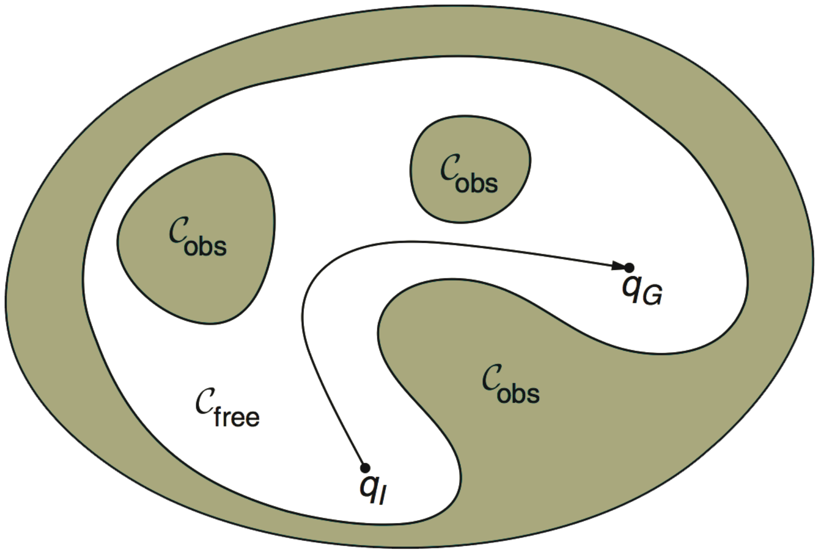
Then, motion planning amounts to

- Finding a continuous path

  $$\tau : [0,1] \rightarrow \mathcal{C}_{free}$$

  with $\tau(0) = q_I \, , \, \tau(1) = q_G$

- Given this setting, we can do planning with the robot being a **point in C-space!**



C-Space Discretizations

- Continuous terrain needs to be **discretized** for path planning
- There are **two general approaches** to discretize C-spaces:
  - **Combinatorial planning**

    Characterizes C_{free} explicitely by capturing the connectivity of C_{free} into a graph and finds solutions using search

  - **Sampling-based planning**

    Uses collision-detection to probe and incrementally search the C-space for solution

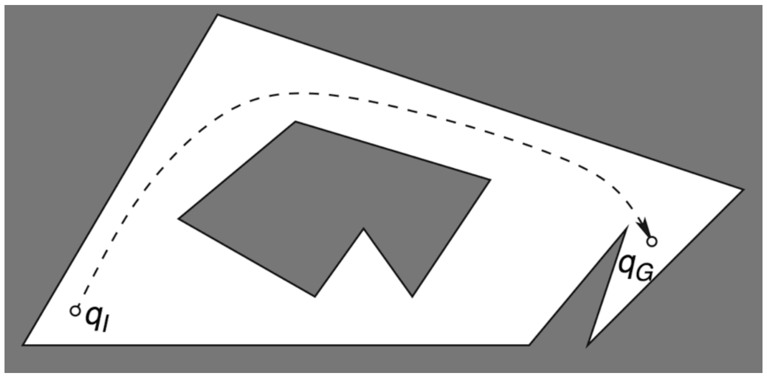
Combinatorial Planning

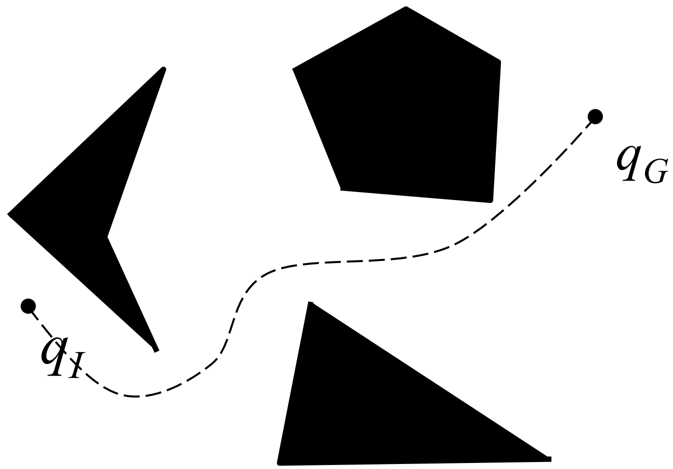
- We will look at four **combinatorial planning techniques**
  - Visibility graphs
  - Voronoi diagrams
  - Exact cell decomposition
  - Approximate cell decomposition

- They all produce a **road map**
  - A **road map** is a **graph in C_{free}** in which each vertex is a configuration in C_{free} and each edge is a collision-free path through C_{free}

Combinatorial Planning

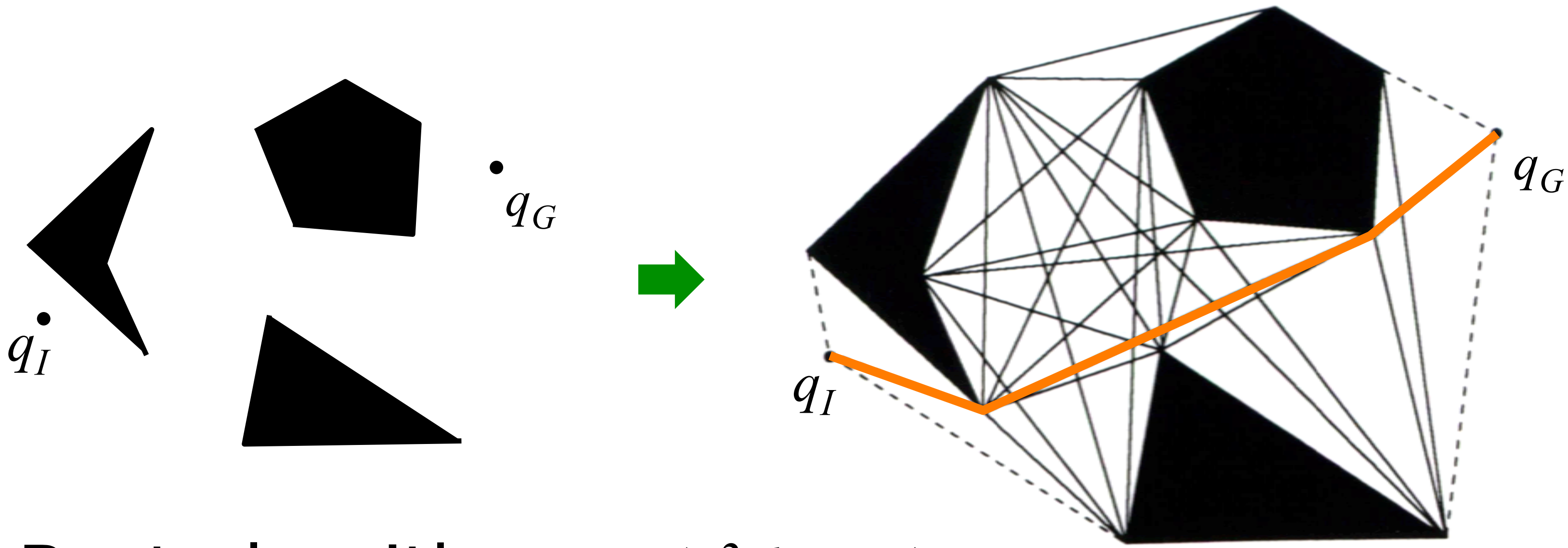
- Without loss of generality, we will consider a problem in $\mathcal{W} = \mathbb{R}^2$ with a **point robot** that cannot rotate. In this case: $\mathcal{C} = \mathbb{R}^2$
- We further assume a **polygonal** world





Visibility Graphs

- **Idea:** construct a path as a polygonal line connecting q_I and q_G through vertices of C_{obs}
- Existence proof for such paths, **optimality**
- One of the earliest path planning methods



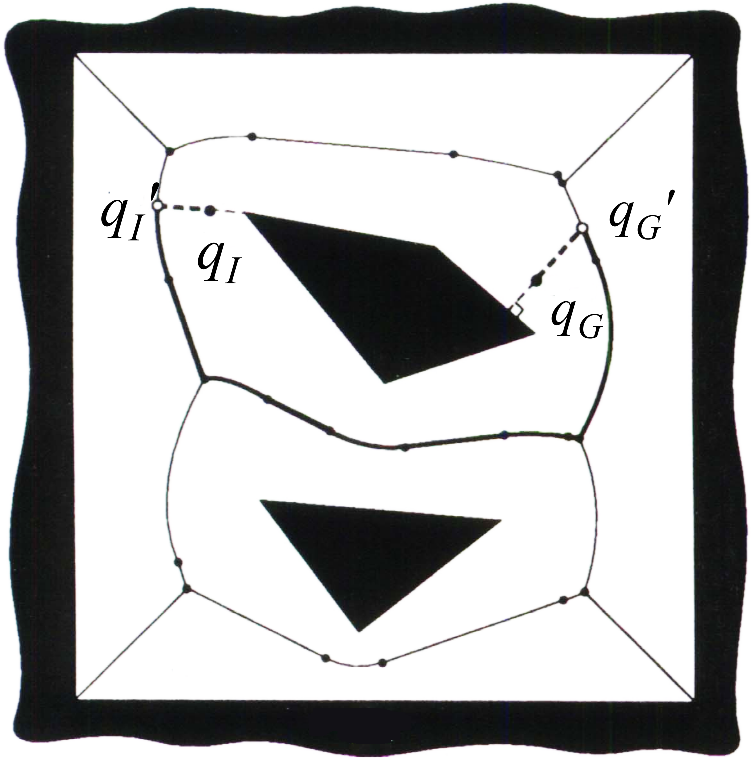
- Best algorithm: $O(n^2 \log n)$

Generalized Voronoi Diagram

- **Defined** to be the set of points q whose cardinality of the set of boundary points of C_{obs} with the same distance to q is greater than 1

- Let us decipher this definition...

- **Informally:** the place with the same **maximal clearance** from all nearest obstacles



Generalized Voronoi Diagram

- **Formally:**

  Let $\beta = \partial \mathcal{C}_{free}$ be the boundary of C_{free}, and $d(p,q)$ the Euclidian distance between p and q. Then, for all q in C_{free}, let

  $$clearance(q) = \min_{p \in \beta} d(p, q)$$

  be the *clearance* of q, and

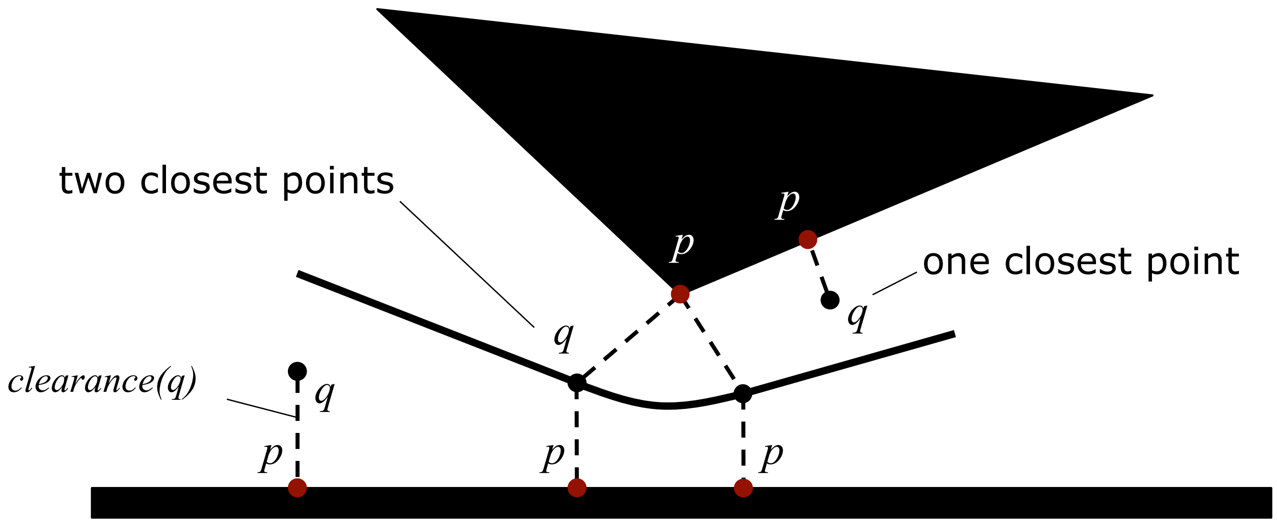
  $$near(q) = \{p \in \beta \mid d(p, q) = clearance(q)\}$$

  the set of "base" points on β with the same clearance to q. The **Voronoi diagram** is then the set of q's with more than one base point p

  $$V(\mathcal{C}_{free}) = \{q \in \mathcal{C}_{free} \mid |near(q)| > 1\}$$

Generalized Voronoi Diagram

- **Geometrically:**



- For a polygonal C_{obs}, the Voronoi diagram consists of (n) lines and parabolic segments
- Naive algorithm: $O(n^4)$, best: $O(n \log n)$

Voronoi Diagram

- Voronoi diagrams have been well studied for (reactive) **mobile robot** path planning
- Fast methods exist to compute and update the diagram in real-time for low-dim. C's
  - **Pros:** maximize clear-ance is a good idea for an uncertain robot
  - **Cons:** unnatural at-traction to open space, suboptimal paths

- Needs extensions

Exact Cell Decomposition

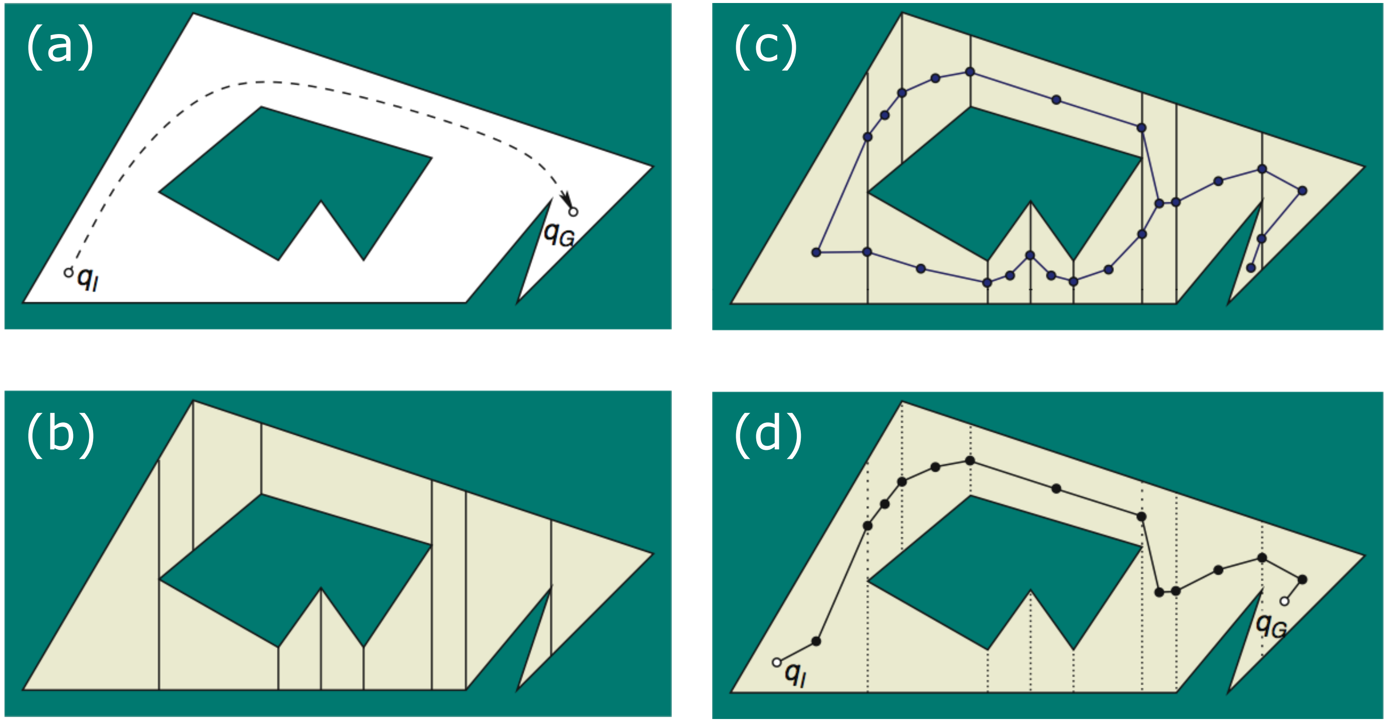
- **Idea:** decompose C_{free} into non-overlapping cells, construct connectivity graph to represent adjacencies, then search

- A popular implementation of this idea:

  1. Decompose C_{free} into **trapezoids** with vertical side segments by shooting rays upward and downward from each polygon vertex
  2. Place one **vertex** in the interior of every **trapezoid**, pick e.g. the centroid
  3. Place one **vertex** in every vertical **segment**
  4. Connect the vertices

Exact Cell Decomposition

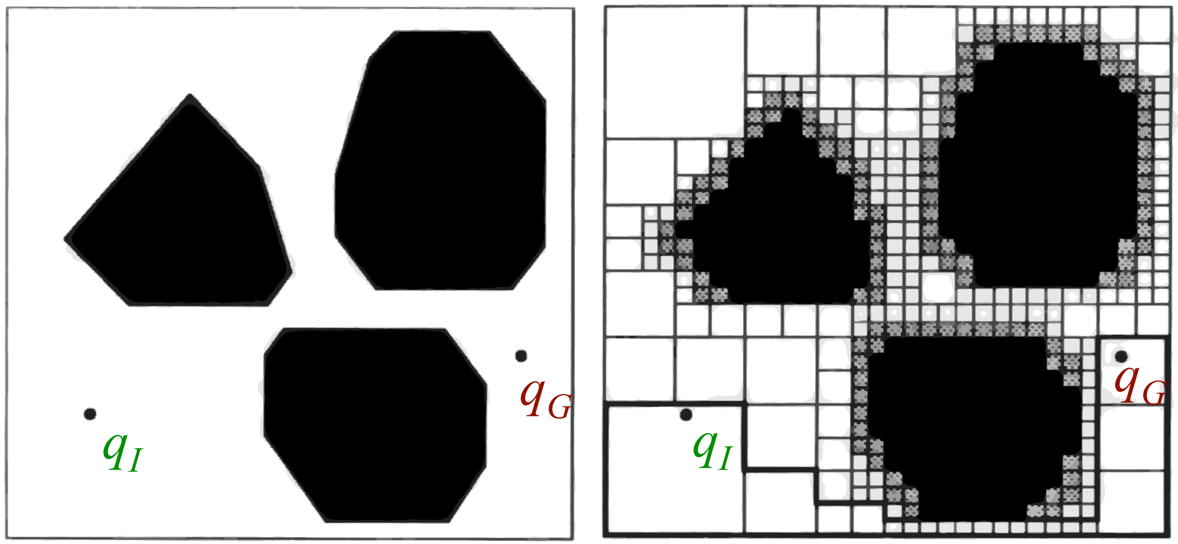
- Trapezoidal decomposition ($\mathcal{C} = \mathbb{R}^3$ max)



- Best known algorithm: $O(n \log n)$ where n is the number of vertices of C_{obs}

Approximate Cell Decomposition

- Exact decomposition methods can be involved and inefficient for complex problems
- Approximate decomposition uses cells with the **same simple predefined shape**



Quadtree decomposition

Approximate Cell Decomposition

- Exact decomposition methods can be involved and inefficient for complex problems
- Approximate decomposition uses cells with the **same simple predefined shape**
- **Pros:**
  - Iterating the **same** simple computations
  - Numerically more **stable**
  - **Simpler** to implement
  - Can be made **complete**

Combinatorial Planning

Wrap Up

- Combinatorial planning techniques are **elegant** and **complete** (they find a solution if it exists, report failure otherwise)
- But: become **quickly intractable** when C-space dimensionality increases (or n resp.)
- Combinatorial **explosion** in terms of **facets** to represent $\mathcal{A}$, $\mathcal{O}$, and $\mathcal{C}_{obs}$, especially when rotations bring in non-linearities and make C a nontrivial manifold

➡ Use **sampling-based planning**
Weaker guarantees but more efficient

Sampling-Based Planning

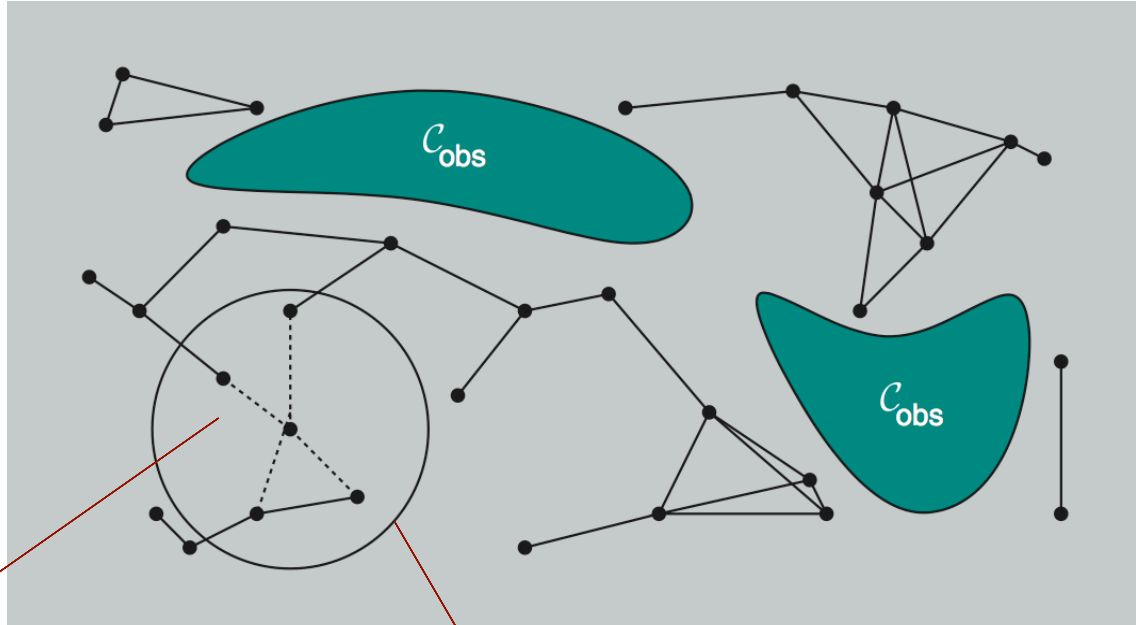
- Abandon the concept of explicitly characterizing C_{free} and C_{obs} and leave the algorithm **in the dark** when exploring C_{free}
- The only light is provided by a **collision-detection algorithm**, that probes C to see whether some configuration lies in C_{free}
- We will have a look at
  - **Probabilistic road maps** (PRM) *[Kavraki et al., 92]*
  - **Rapidly exploring random trees** (RRT) *[Lavalle and Kuffner, 99]*

Probabilistic Road Maps

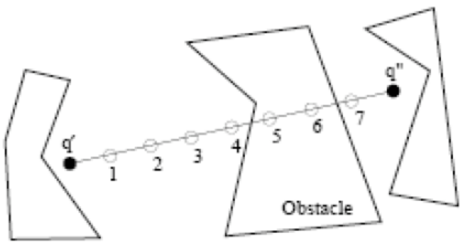
- **Idea:** Take random samples from C, declare them as vertices if in C_{free}, try to connect nearby vertices with local planner
- The **local planner** checks if line-of-sight is collision-free (powerful or simple methods)
- Options for *nearby:* **k-nearest neighbors** or all neighbors within **specified radius**
- Configurations and connections are added to graph until roadmap is **dense enough**

Probabilistic Road Maps

- Example



specified radius



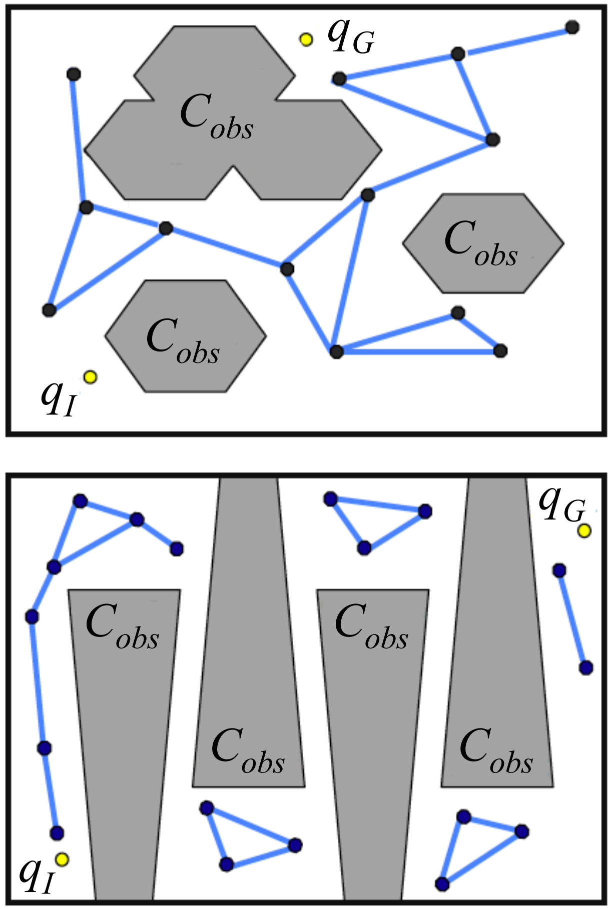
Example local planner

What means "nearby" on a manifold?
Defining a good metric on C is crucial

Probabilistic Road Maps

Good and bad news:

- **Pros:**
  - *Probabilistically complete*
  - Do not construct C-space
  - Apply easily to high-dim. C's
  - PRMs have solved previously unsolved problems
- **Cons:**
  - Do not work well for some problems, narrow passages
  - Not optimal, not complete

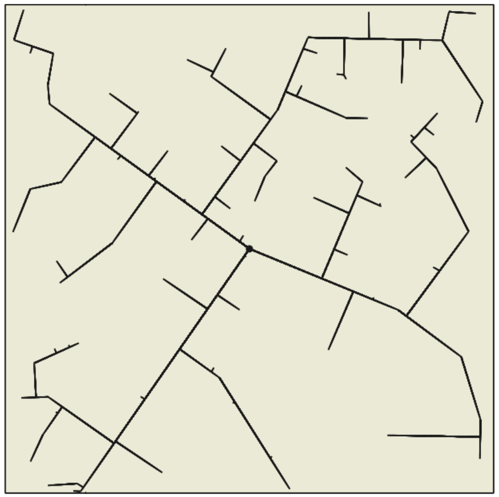


Probabilistic Road Maps

- How to **uniformly sample** C ? This is not at all trivial given its topology
- For example over spaces of rotations: Sampling Euler angles gives samples near poles, not uniform over $SO(3)$. Use quaternions!
- However, PRMs are **powerful, popular** and **many extensions** exist: advanced sampling strategies (e.g. near obstacles), PRMs for deformable objects, closed-chain systems, etc.

Rapidly Exploring Random Trees

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



45 iterations

2345 iterations

RRTs

- The algorithm: Given C and q_0

Algorithm 1: RRT

$$
\begin{array}{l}
G.\mathrm{init}(q_0) \\
\textbf{repeat} \\
\quad q_{rand} \rightarrow \mathrm{RANDOM\_CONFIG}(\mathcal{C}) \\
\quad q_{near} \leftarrow \mathrm{NEAREST}(G, q_{rand}) \\
\quad G.\mathrm{add\_edge}(q_{near}, q_{rand}) \\
\textbf{until}\ \mathit{condition}
\end{array}
$$

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

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

(but recall sampling over rotation spaces problem)

RRTs

- The algorithm

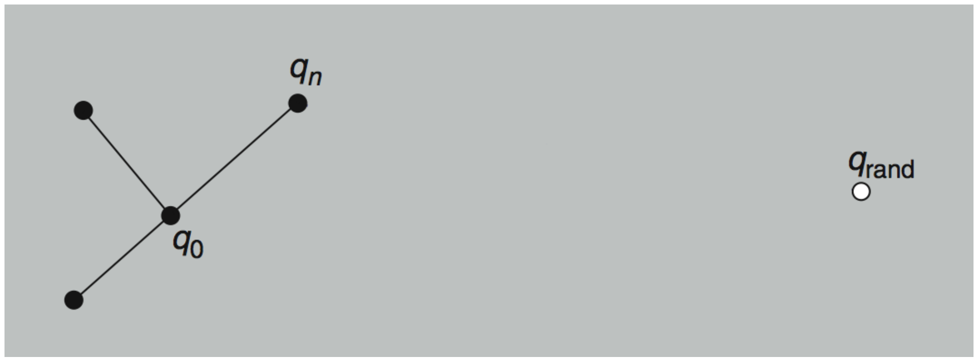
Algorithm 1: RRT

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

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

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

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

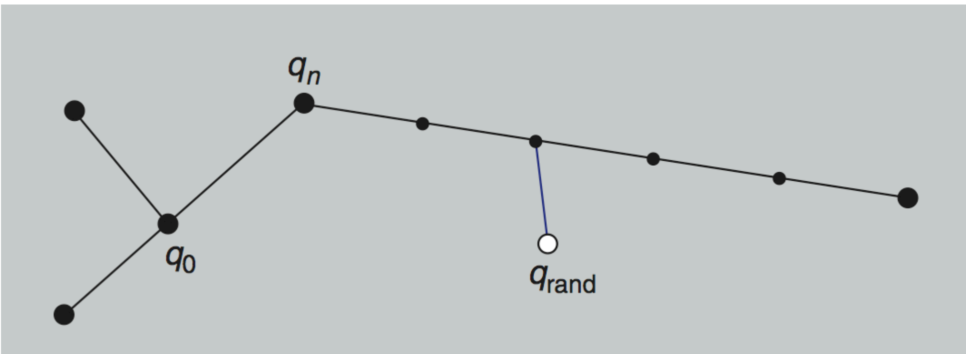


RRTs

- The algorithm

Algorithm 1: RRT

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



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

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

RRTs

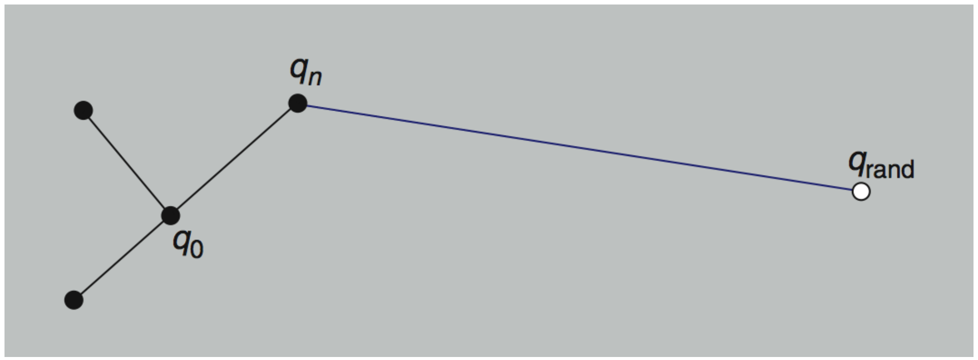
- The algorithm

Algorithm 1: RRT

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

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

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

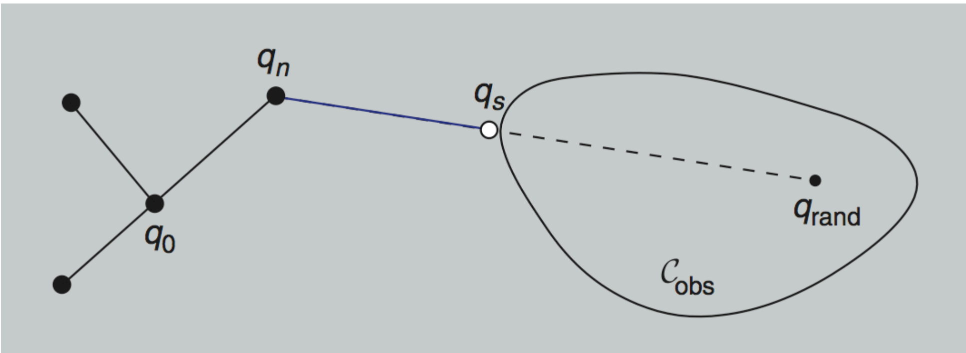


RRTs

- The algorithm

Algorithm 1: RRT

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



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

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

RRTs

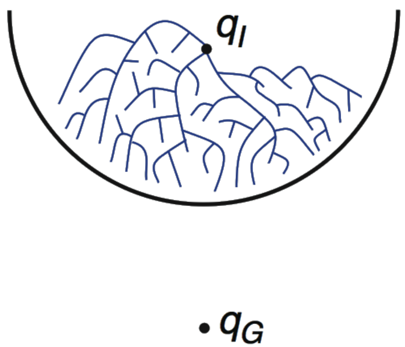
- How to perform path planning with RRTs?

  1. Start RRT at q_I
  2. At every, say, 100th iteration, force $q_{rand} = q_G$
  3. If q_G is reached, problem is solved

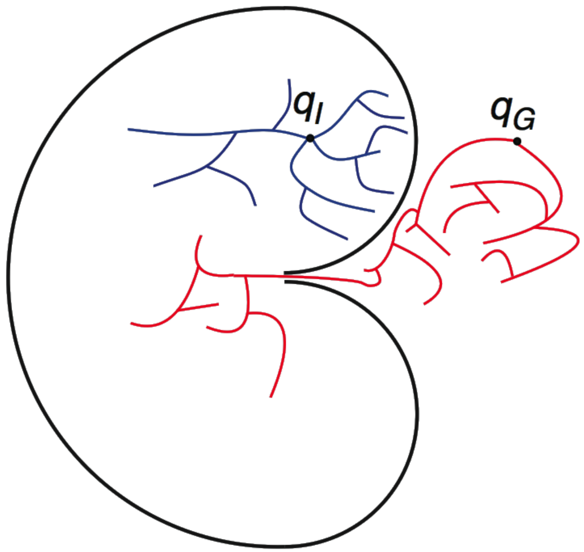
- Why not picking q_G every time?
- This will fail and waste much effort in running into C_{Obs} instead of exploring the space

RRTs

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



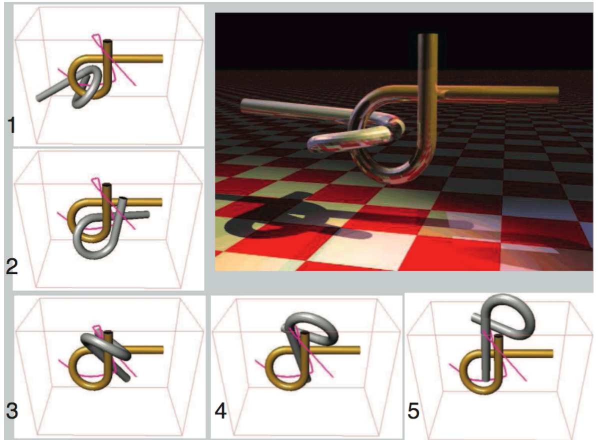
Filling a well



A bug trap

RRTs

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



Alpha 1.0 puzzle.
Solved with
bidirectional RRT

From Road Maps to Paths

- All methods discussed so far **construct a road map** (without considering the query pair q_I and q_G)
- Once the investment is made, the **same road map** can be reused for **all** queries (provided world and robot do not change)

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

Sampling-Based Planning

Wrap Up

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

Potential Field Methods

- All techniques discussed so far aim at capturing the connectivity of C_{free} into a graph
- **Potential Field methods** follow a different idea:

  The robot, represented as a point in C, is modeled as a **particle** under the influence of a **artificial potential field** $\mathbf{U}$

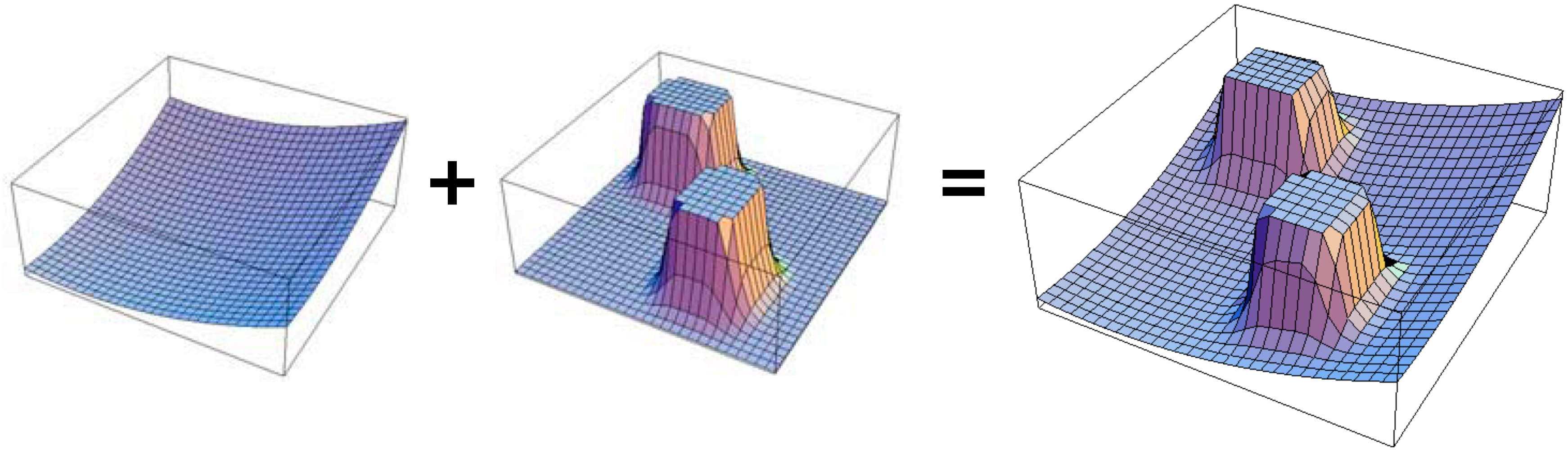
  $\mathbf{U}$ superimposes
  - **Repulsive forces** from obstacles
  - **Attractive force** from goal

Potential Field Methods

- Potential function

$$\mathbf{U}(q) = \mathbf{U}_{att}(q) + \mathbf{U}_{rep}(q)$$

$$\vec{F}(q) = -\vec{\nabla}\mathbf{U}(q)$$



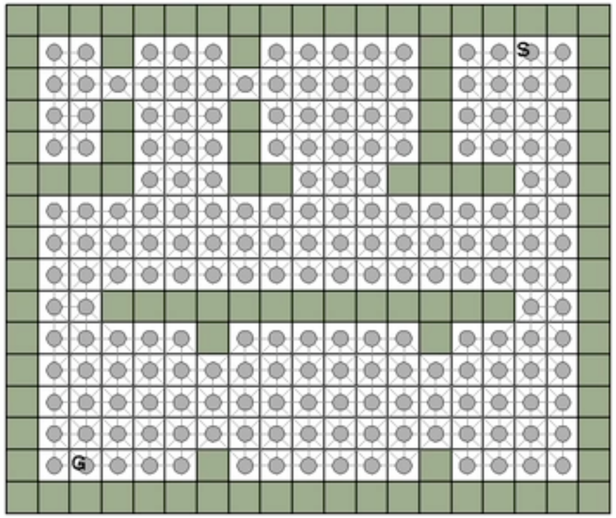
- Simply perform **gradient descent**
- C-pace typically discretized in a grid

Potential Field Methods

- Main problems: robot gets stuck in **local minima**
- Way out: Construct local-minima-free **navigation function** ("NF1"), then do gradient descent (e.g. bushfire from goal)
- The gradient of the potential function defines a **vector field** (similar to a policy) that can be used as **feedback control strategy**, relevant for an uncertain robot
- However, potential fields need to represent C_{free} **explicitely**. This can be too costly.

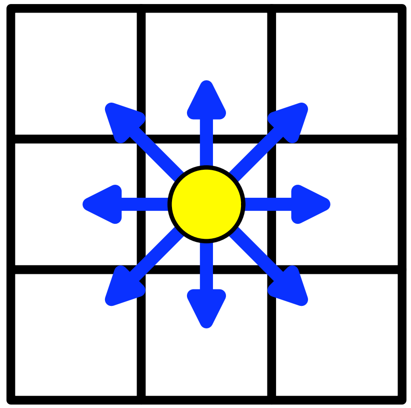
Robot Motion Planning

- Given a road map, let's do **search**!



A* Search

- **A*** is one of the most widely-known informed search algorithms with many applications in robotics
- *Where are we?*
  A* is an instance of an **informed algorithm** for the general problem of **search**
- In robotics: planning on a 2D occupancy grid map is a common approach



Search

The problem of **search:** finding a sequence of actions (a *path*) that leads to desirable states (a *goal*)

- **Uninformed search:** besides the problem definition, no further information about the domain ("blind search")
- The only thing one can do is to expand nodes differently
- Example algorithms: breadth-first, uniform-cost, depth-first, bidirectional, etc.

Search

The problem of **search:** finding a sequence of actions (a *path*) that leads to desirable states (a *goal*)

- **Informed search:** further information about the domain through **heuristics**
- Capability to say that a node is "more promising" than another node
- Example algorithms: greedy best-first search, **A***, many variants of A*, D*, etc.

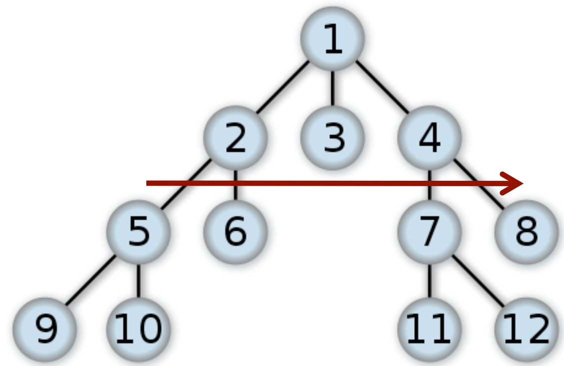
Search

The performance of a search algorithm is measured in four ways:

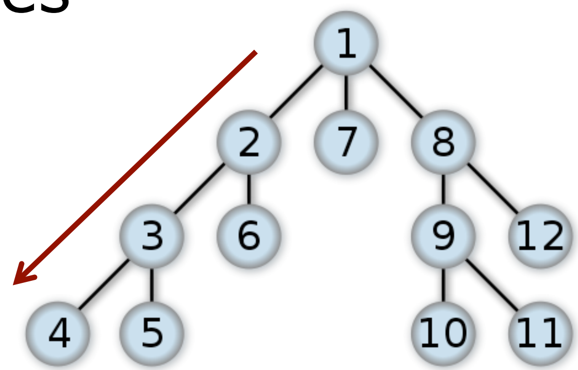
- **Completeness:** does the algorithm find the solution when there is one?
- **Optimality:** is the solution the best one of all possible solutions in terms of path cost?
- **Time complexity:** how long does it take to find a solution?
- **Space complexity:** how much memory is needed to perform the search?

Uninformed Search

- **Breadth-first**
  - Complete
  - Optimal if action costs equal
  - Time and space: $O(b^d)$

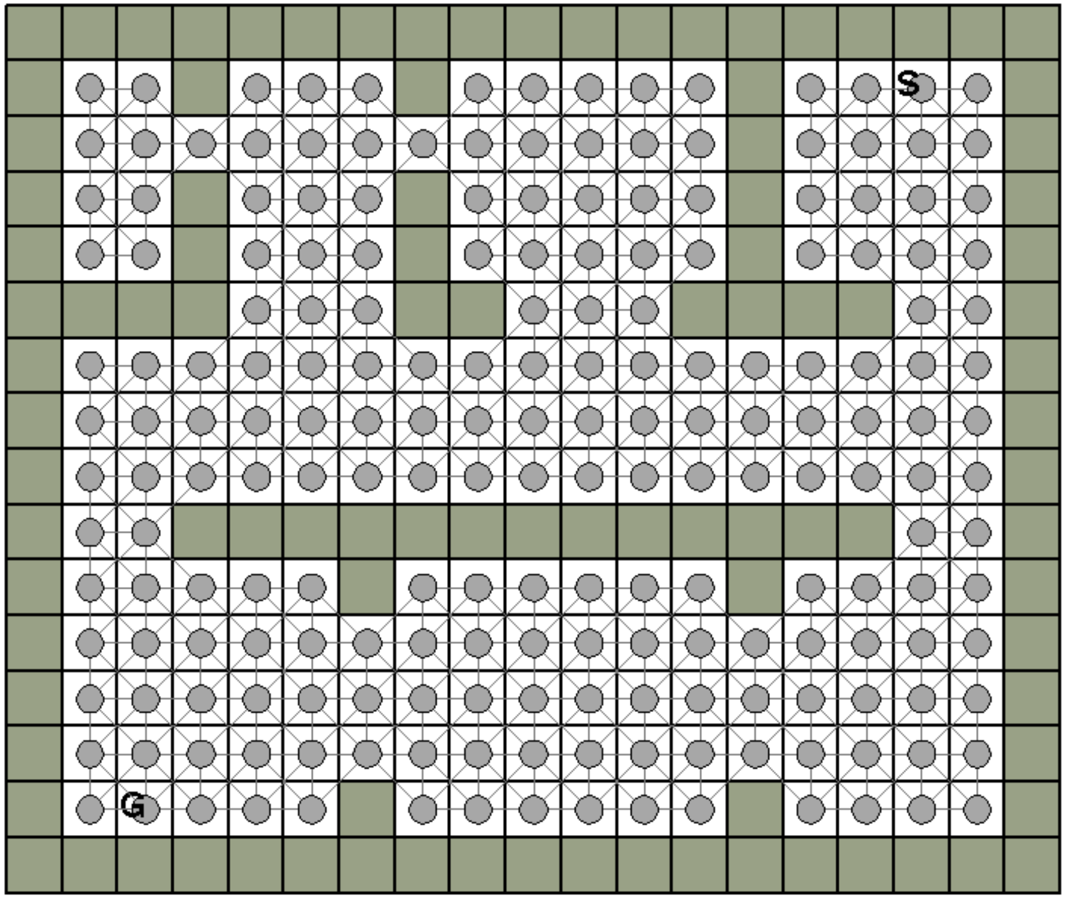


- **Depth-first**
  - Not complete in infinite spaces
  - Not optimal
  - Time: $O(b^m)$
  - Space: $O(bm)$ (can forget explored subtrees)



(b: branching factor, d: goal depth, m: max. tree depth)

Breadth-First Example



Informed Search

- Nodes are selected for expansion based on an **evaluation function** $f(n)$ from the set of generated but not yet explored nodes
- Then select node first with lowest $f(n)$ value

- Key component to every choice of $f(n)$: **Heuristic function** $h(n)$
- Heuristics are most common way to inject domain knowledge and inform search
- Every $h(n)$ is a cost estimate of cheapest path from n to a goal node

Informed Search

- **Greedy Best-First-Search**
  - Simply expands the node closest to the goal

    $$f(n) = h(n)$$

  - Not optimal, not complete, complexity $O(b^m)$

- **A* Search**
  - Combines path cost to n, $g(n)$, and estimated goal distance from n, $h(n)$

    $$f(n) = g(n) + h(n)$$

  - $f(n)$ estimates the cheapest path cost through n
  - If $h(n)$ is *admissible*: **complete** and **optimal**!

Heuristics

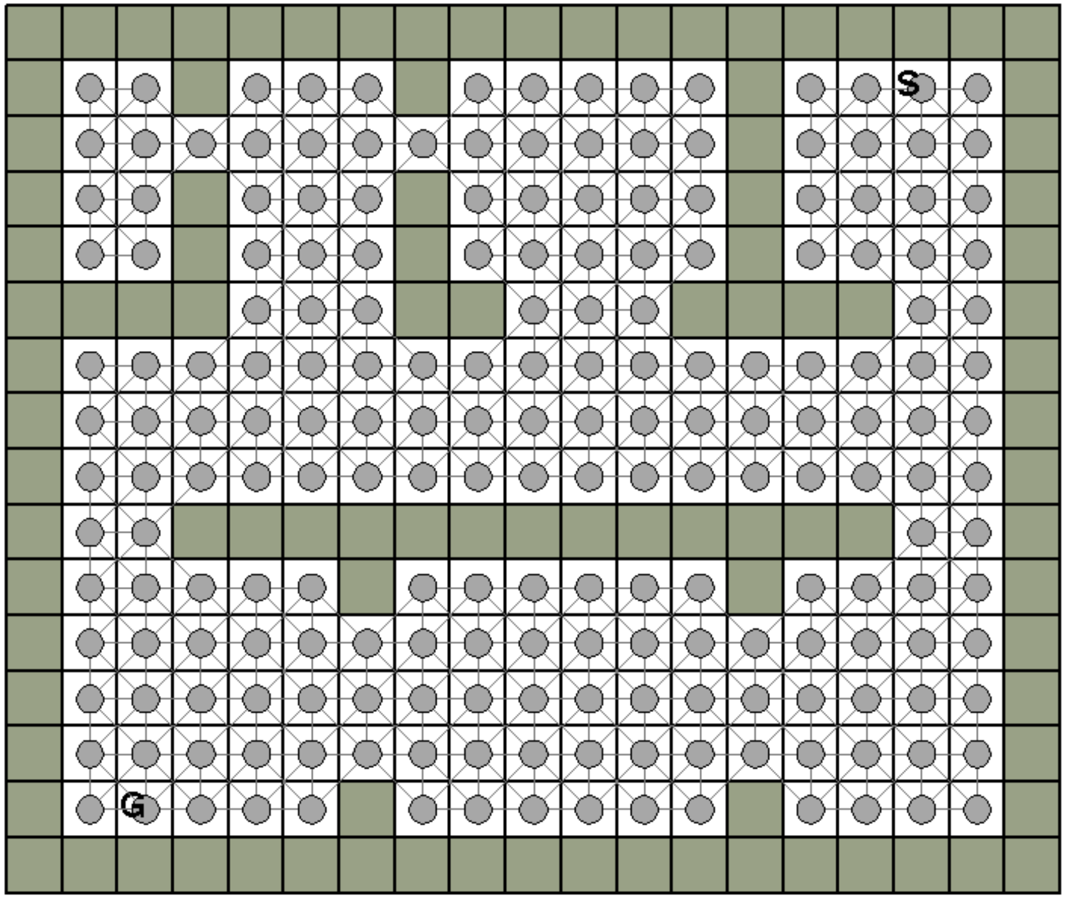
- **Admissible heuristic:**
  - Let $h^*(n)$ be the true cost of the optimal path from n to the goal. Then $h(.)$ is admissible if the following holds for all n:

$$h(n) \leq h^*(n)$$

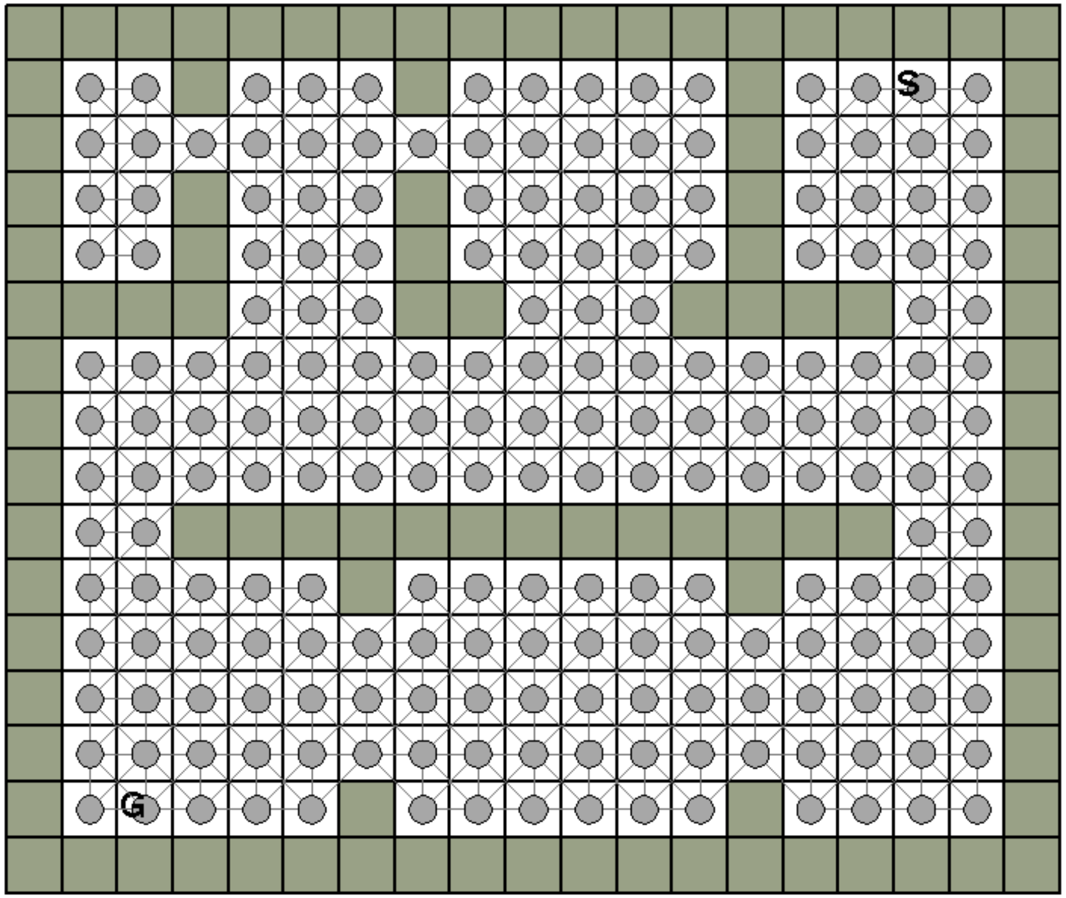
be optimistic, never overestimate the cost

- Heuristics are problem-specific. Good ones (admissible, efficient) for **our task** are:
  - **Straight-line distance** $h_{SLD}(n)$ (as with any routing problem)
  - **Octile distance**: Manhattan distance extended to allow diagonal moves
  - Deterministic **Value Iteration**/Dijkstra $h_{VI}(n)$

Greedy Best-First Example



A* with h_{SLD} Example



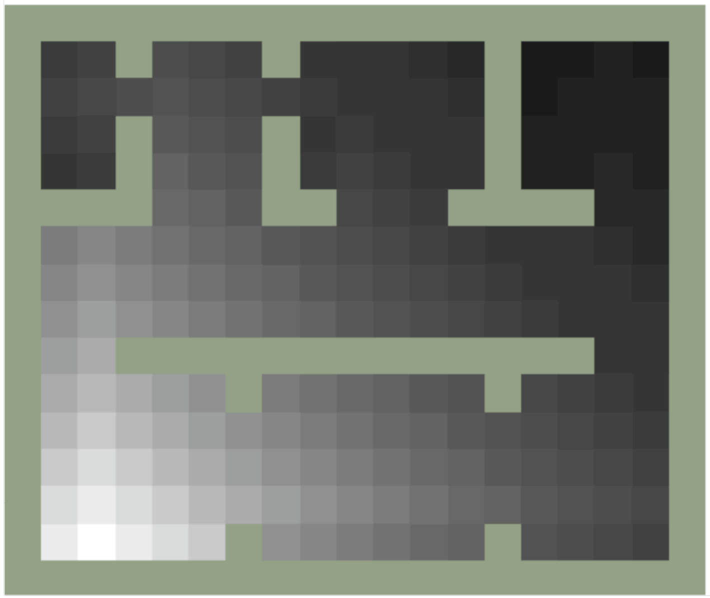
Heuristics for A*

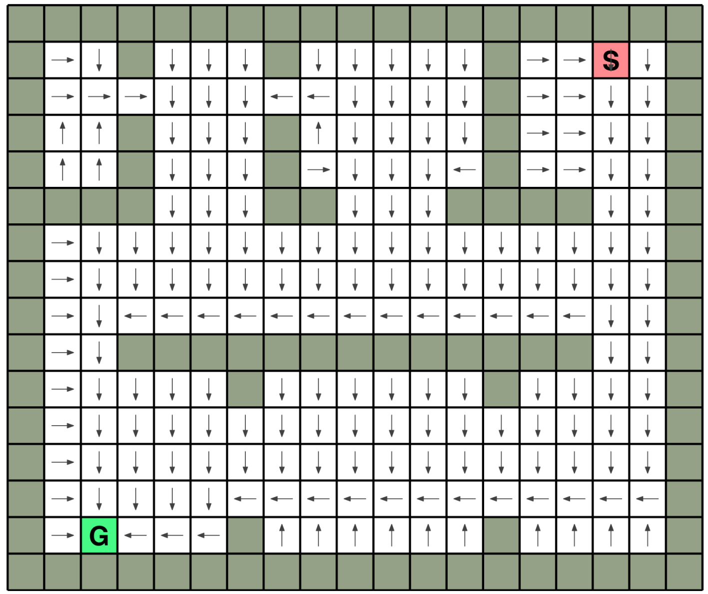
- Deterministic Value Iteration
  - Use Value Iteration for MDPs (later in this course) with rewards -1 and unit discounts
  - Like Dijkstra

- Precompute for dynamic or unknown environments where replanning is likely

Heuristics for A*

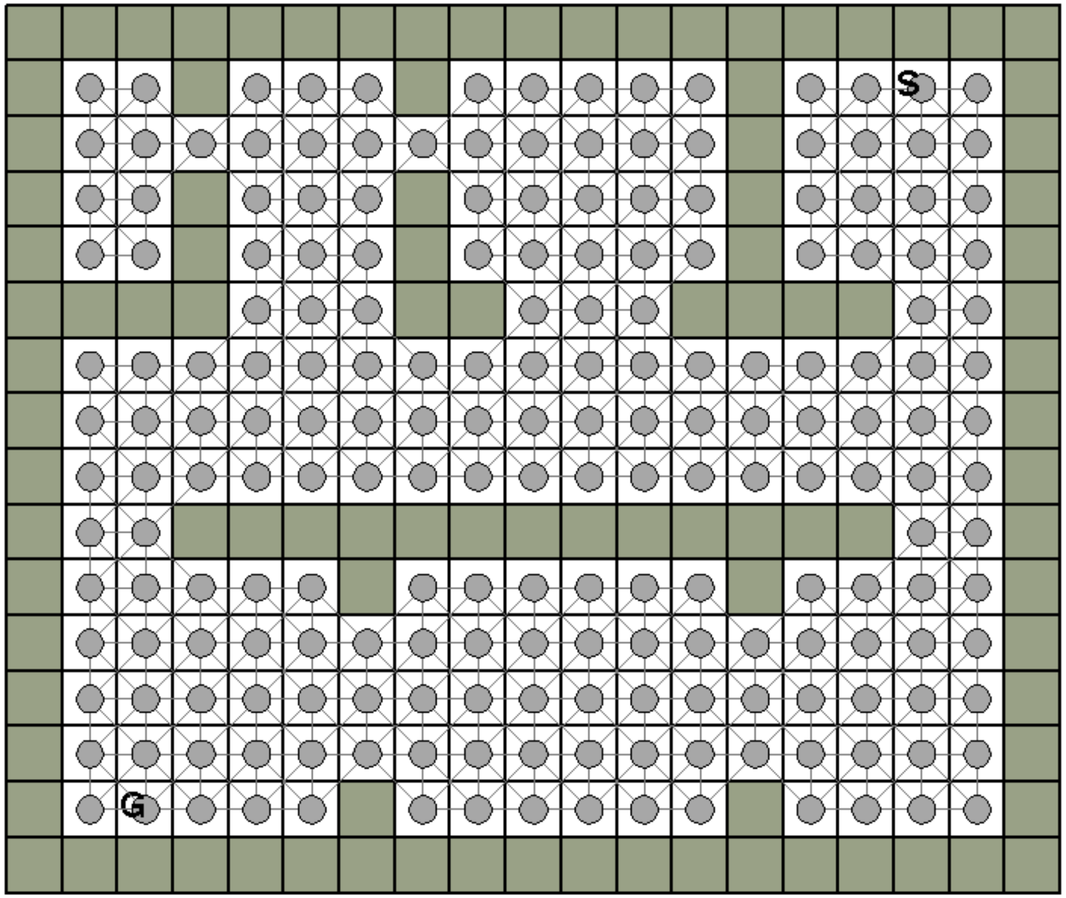
- Deterministic Value Iteration





- Recall vector field from potential functions: allows to implement a **feedback control strategy** for an uncertain robot

A* with h_{VI} Example



Problems with A* on Grids

1. The shortest path is often very **close to obstacles** (cutting corners)
   - Uncertain path execution increases the risk of collisions
   - Uncertainty can come from delocalized robot, imperfect map or poorly modeled dynamic constraints

2. Trajectories **aligned to the grid** structure
   - Path looks unnatural
   - Such paths are longer than the true shortest path in the continuous space

Problems with A* on Grids

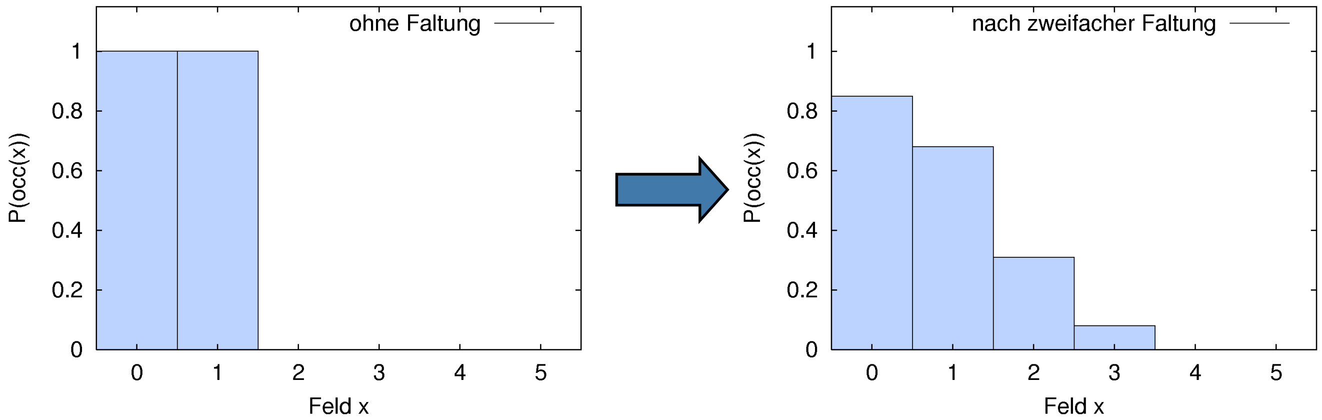
3. When the path turns out to be blocked during traversal, it needs to be **replanned from scratch**
   - In unknown or dynamic environments, this can occur very often
   - Replanning in large state spaces is costly
   - Can we reuse the initial plan?

Let us look at **solutions** to these problems...

Map Smoothing

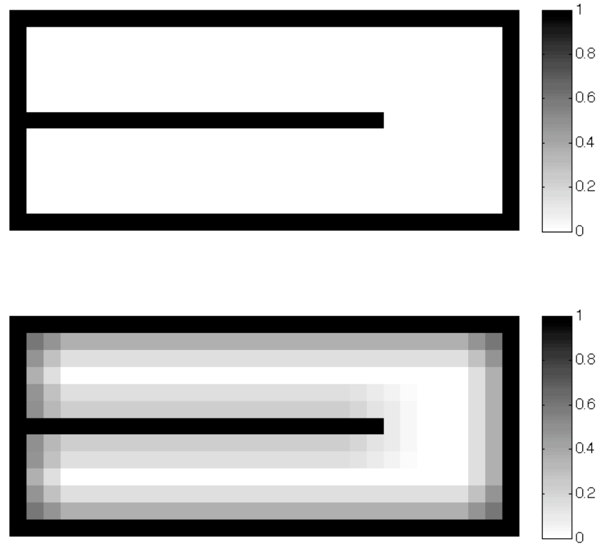
- Given an occupancy grid map
- **Convolution** blurs the map M with kernel k (e.g. a Gaussian kernel)

$$(M * k)[i] = \sum_{j=-\infty}^{\infty} M[j]\, k[i-j]$$



1D example: cells before and after two convolution runs

Map Smoothing



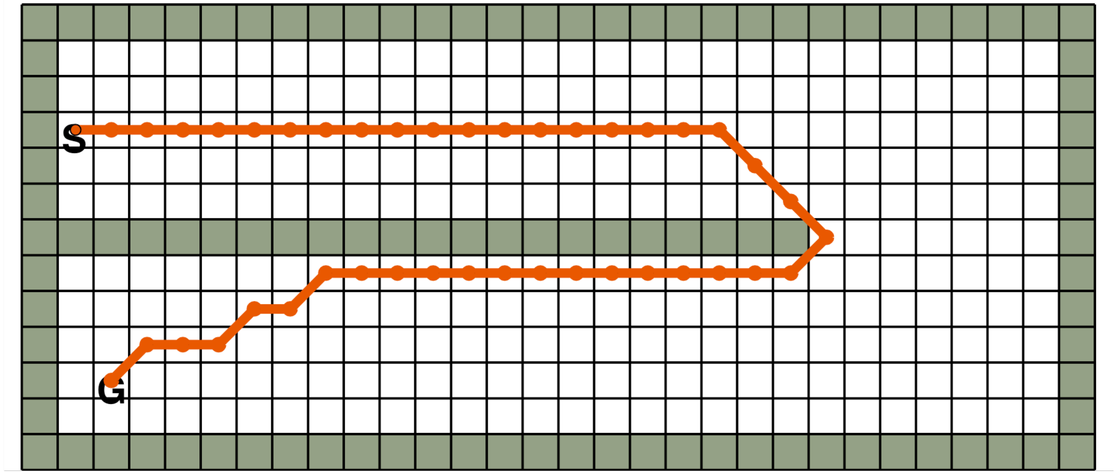
- Leads to above-zero probability areas around obstacles. Obstacles **appear bigger** than in reality
- Perform A* search in **convolved map** with evalution function

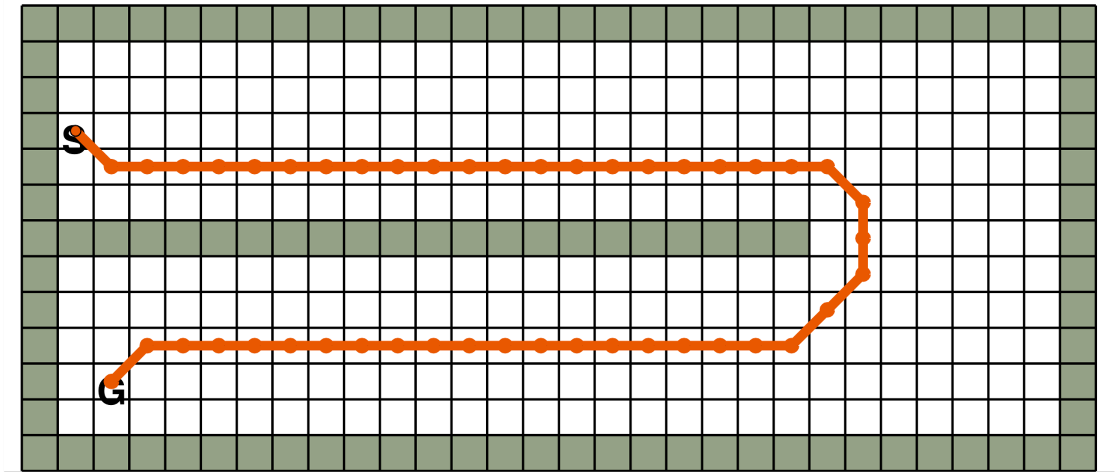
$$f(n) = g(n) \cdot p_{occ}(n) + h(n)$$

$p_{occ}(n)$: occupancy probability of node/cell n

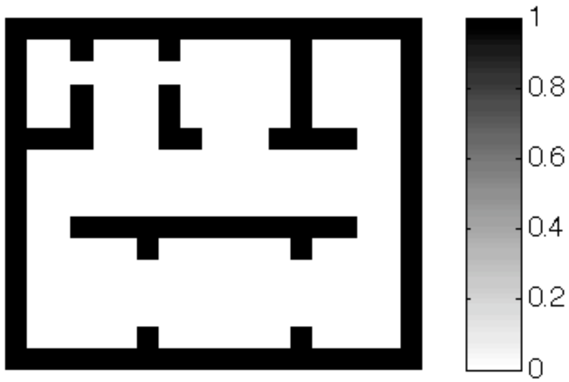
- Could also be a term for cell traversal cost

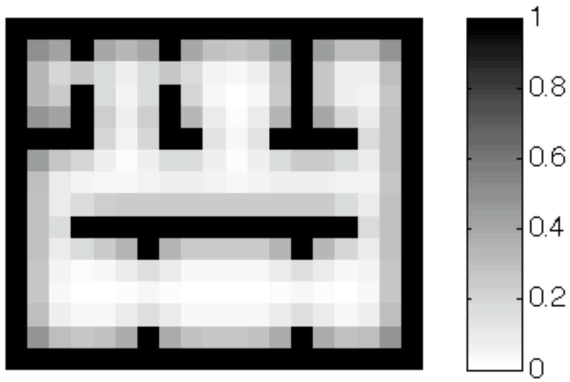
Map Smoothing

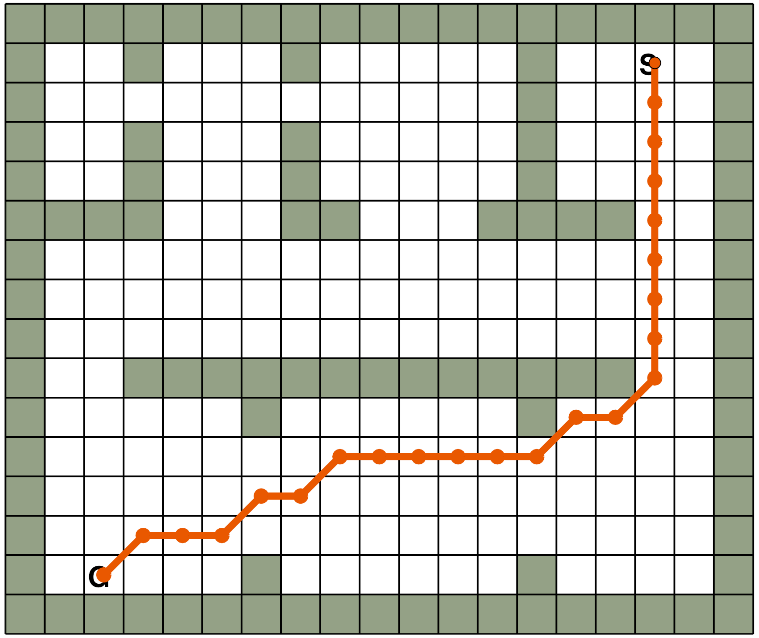


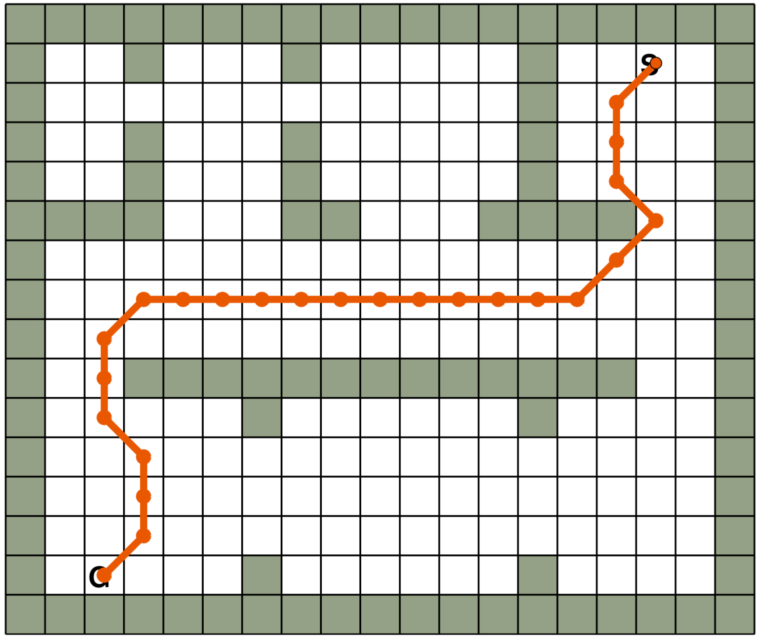


Map Smoothing



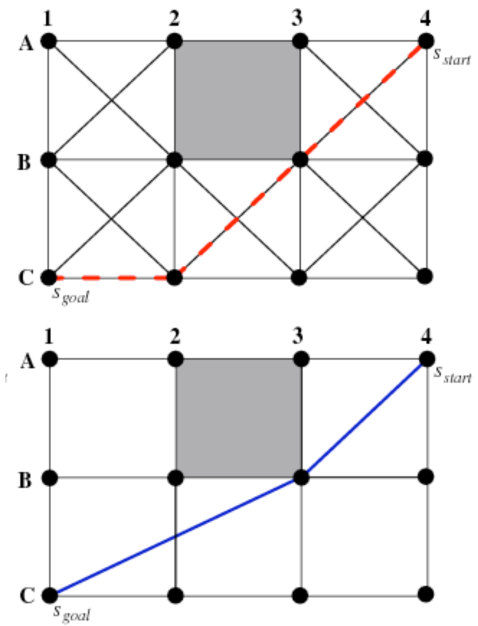


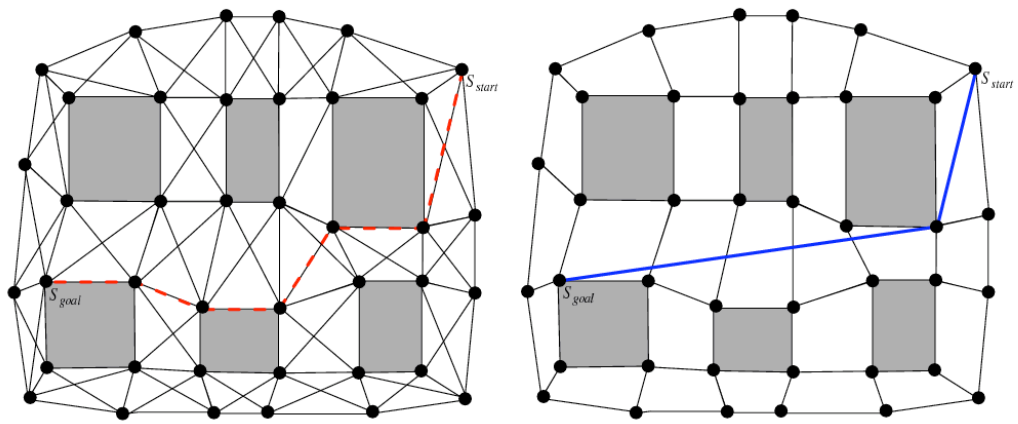




Any-Angle A*

- **Problem:** A* search only considers paths that are **constrained to graph edges**
- This can lead to **unnatural**, grid-aligned, and **suboptimal** paths





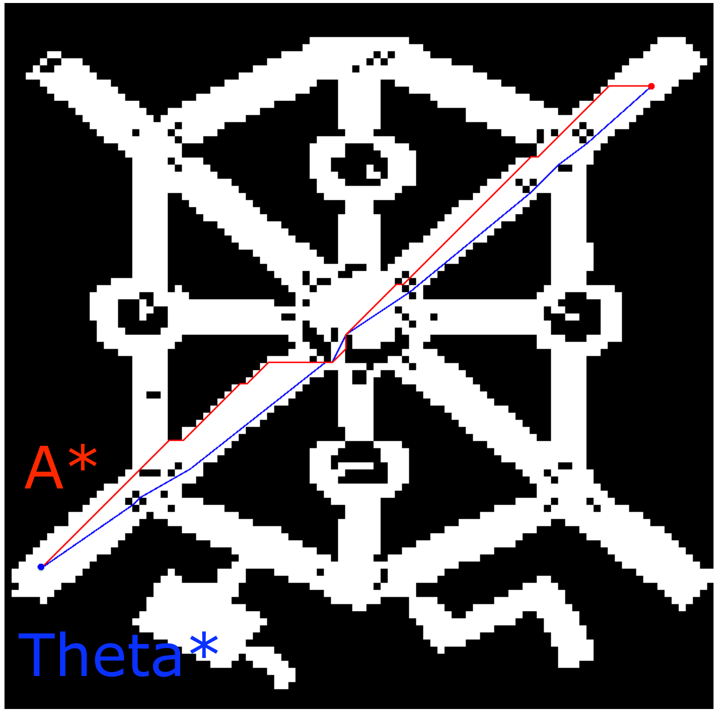
Pictures from *[Nash et al. AAAI'07]*

Any-Angle A*

- Different approaches:
  - **A* on Visibility Graphs**
    Optimal solutions in terms of path length!
  - **A* with post-smoothing**
    Traverse solution and find pairs of nodes with direct line of sight, replace by line segment
  - **Field D*** *[Ferguson and Stentz, JFR'06]*
    Interpolates costs of points not in cell centers. Builds upon D* family, able to efficiently replan
  - **Theta*** *[Nash et al. AAAI'07, AAAI'10]*
    Extension of A*, nodes can have non-neighboring successors based on a line-of-sight test

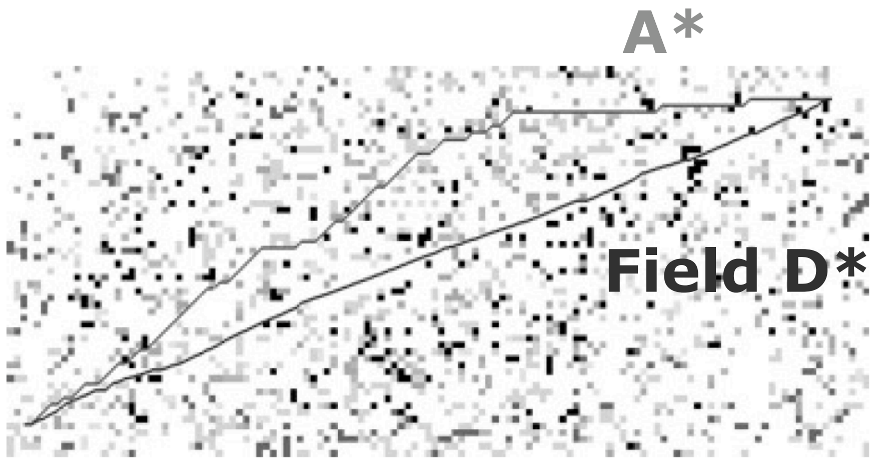
Any-Angle A* Examples

- Theta*



Game environment

- Field D*

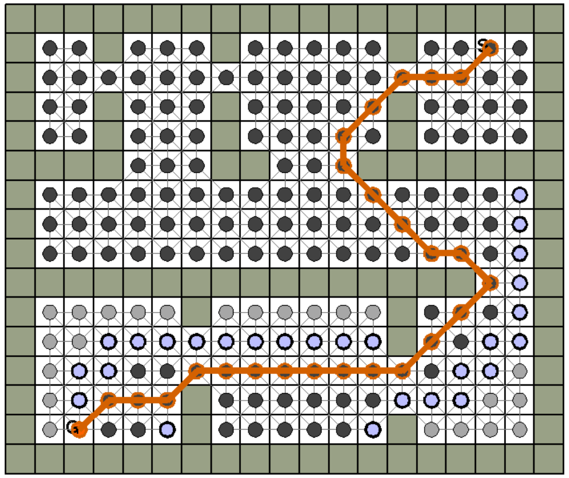


Outdoor environment.
Darker cells have larger **traversal costs**

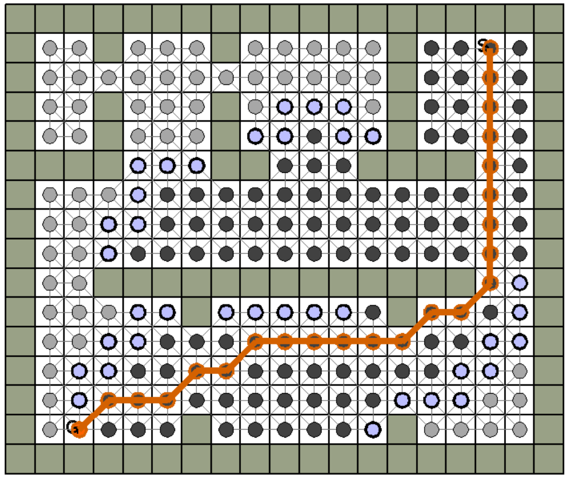
Any-Angle A* Examples

- A* vs. Theta*

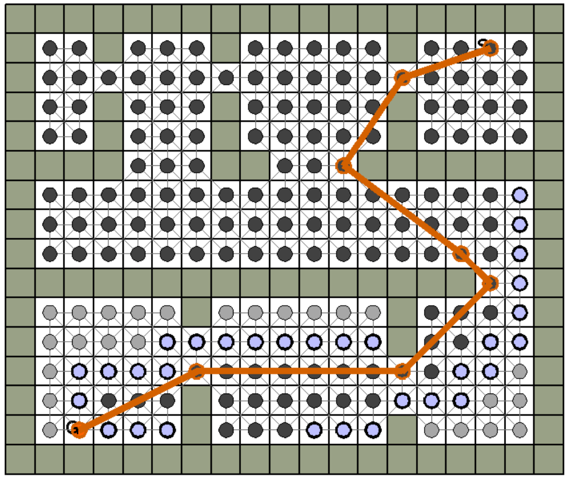
(*len*: path length, *nhead* = # heading changes)



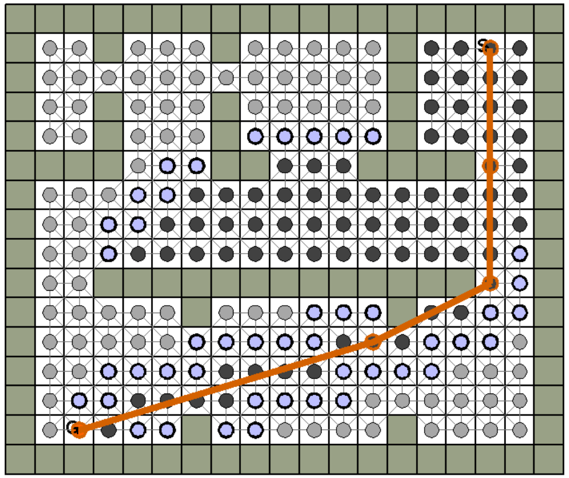
len: 30.0
nhead: 11



len: 24.1
nhead: 9

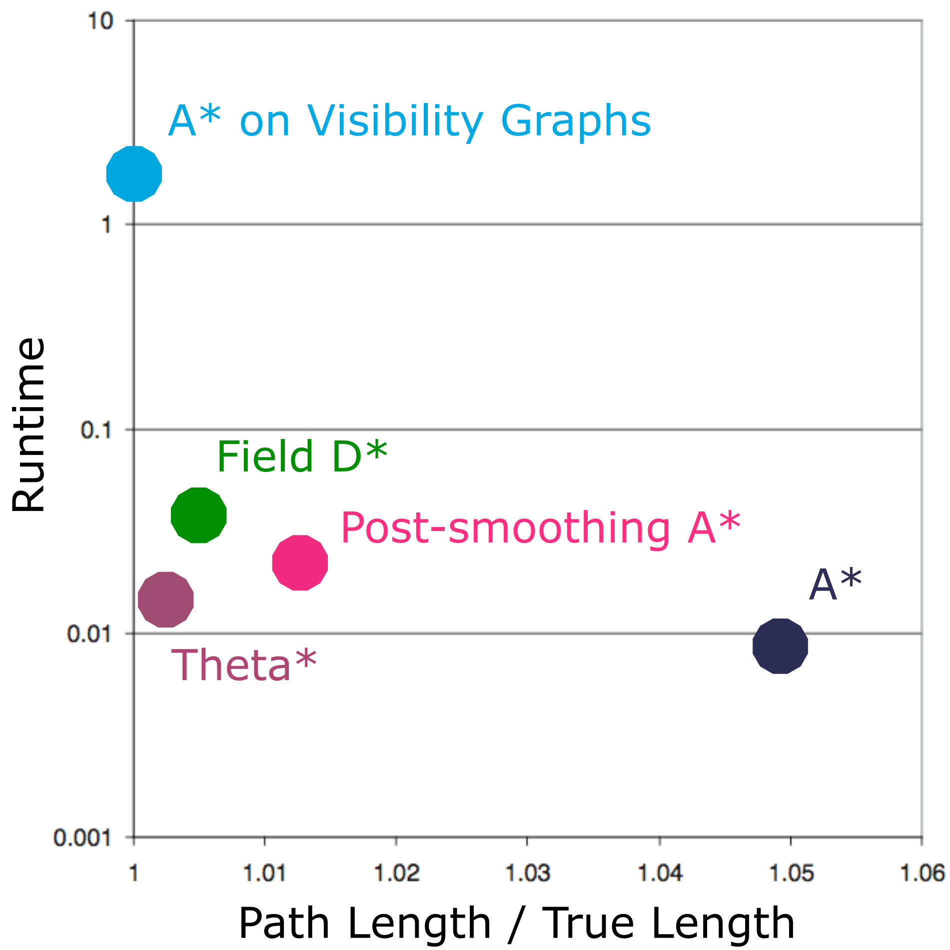


len: 28.9
nhead: 5



len: 22.9
nhead: 2

Any-Angle A* Comparison



[Daniel et al. JAIR'10]

- **A* PS** and **Theta*** provide the best trade off for the problem
- **A* on Visibility Graphs** scales poorly (but is optimal)
- **A* PS** does not always work in nonuniform cost environments. Shortcuts can end up in expensive areas

D* Search

- **Problem:** In unknown, partially known or dynamic environments, the planned path may be blocked and we need to **replan**
- Can this be done efficiently, avoiding to replan the **entire path?**
- **Idea:** Incrementally repair path keeping its modifications local around robot pose
- Several approaches implement this idea:
  - **D*** (Dynamic A*) *[Stentz, ICRA'94, IJCAI'95]*
  - **D* Lite** *[Koenig and Likhachev, AAAI'02]*
  - **Field D*** *[Ferguson and Stentz, JFR'06]*

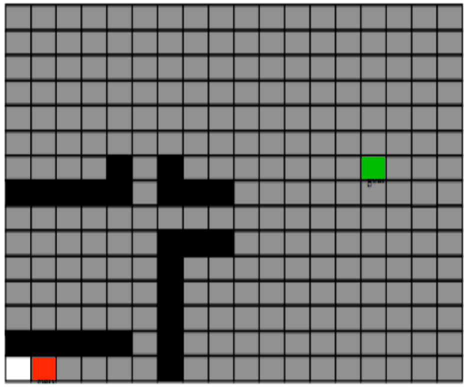
D*/D* Lite

- **Main concepts**
  - **Switched search direction**: search from goal to the current vertex. If a change in edge cost is detected during traversal (around the current robot pose), only few nodes near the goal (=start) need to be updated
  - These nodes are nodes those **goal distances** have changed or not been caculated before AND are **relevant** to recalculate the new shortest path to the goal
  - **Incremental heuristic search** algorithms: able to focus and build upon previous solutions

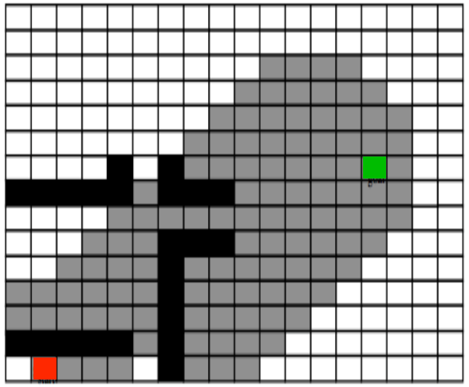
D* Lite Example

- Situation at start

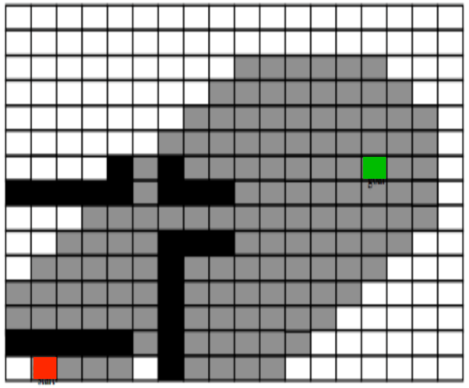
Breadth-First-Search



A*



D* Lite



Start

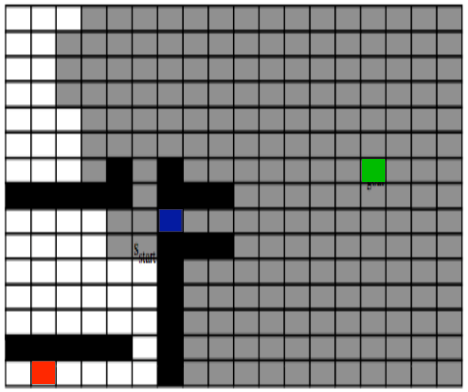
Goal

Expanded nodes (goal distance calculated)

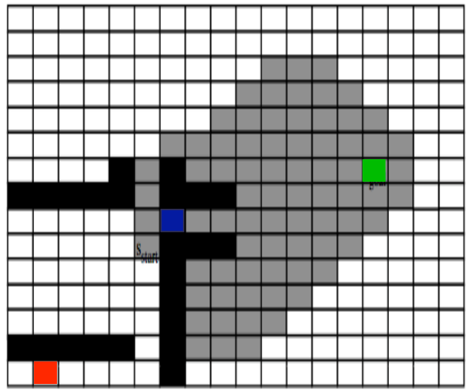
D* Lite Example

- After discovery of blocked cell

Breadth-First-Search



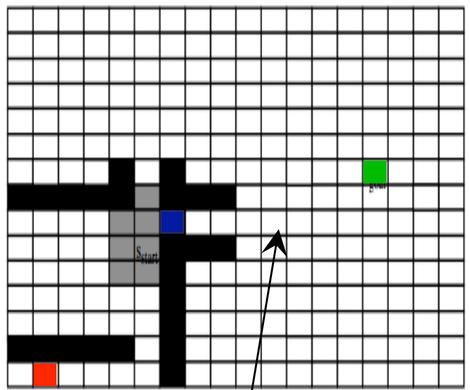
A*



Blocked cell

Updated nodes

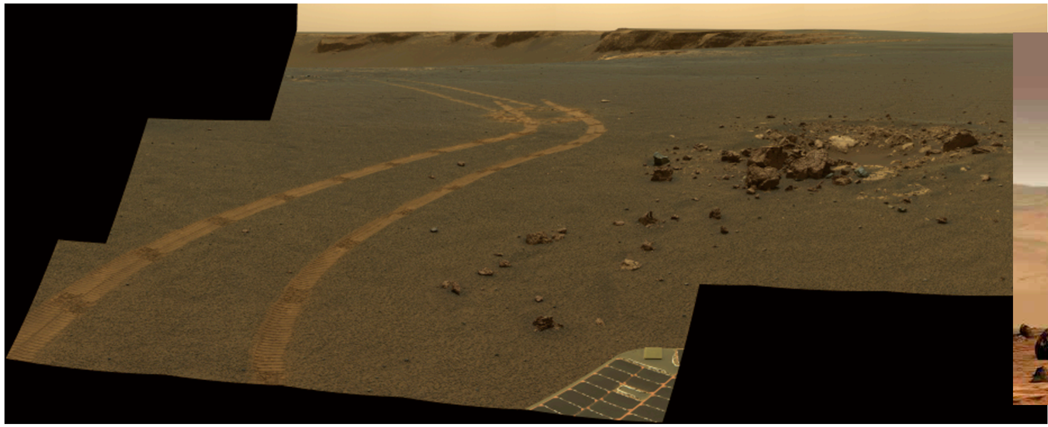
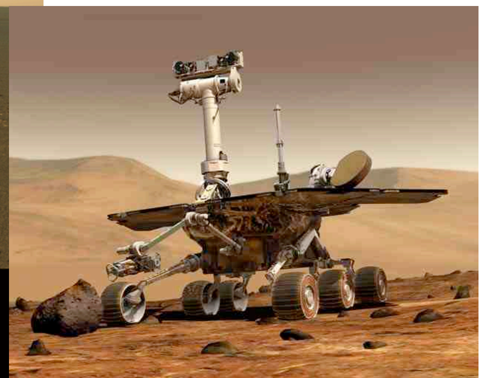
D* Lite



All other nodes remain unaltered, the shortest path can reuse them.

D* Family

- **D* Lite** produces the same paths than D* but is **simpler** and more **efficient**
- D*/D* Lite are **widely used**
- **Field D*** was running on Mars rovers Spirit and Opportunity (retrofitted in yr 3)

Tracks left by a drive executed with Field D*

Still in Dynamic Environments...

- Do we really need to replan the entire path for **each obstacle** on the way?
- What if the robot has to react **quickly** to unforeseen, fast moving obstacles?
  - Even D* Lite can be too slow in such a situation
- Accounting for the **robot shape** (it's not a point)
- Accounting for **kinematic** and **dynamic** vehicle **constraints**, e.g.
  - Decceleration limits,
  - Steering angle limits, etc.

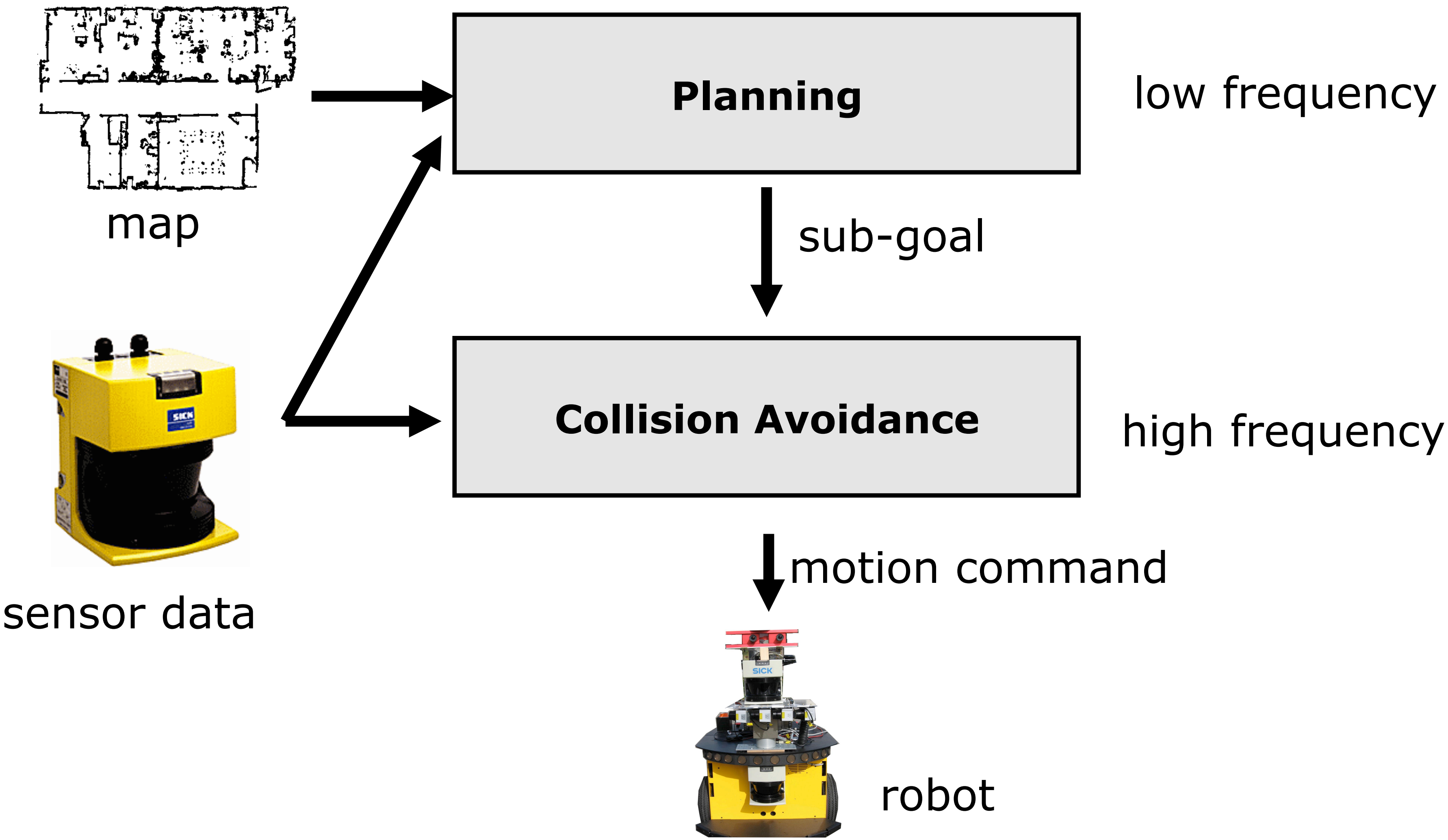
Collision Avoidance

- This can be handled by techniques called **collision avoidance** (obstacle avoidance)
- A well researched subject, different **approaches** exist:
  - Dynamic Window Approaches
    [Simmons, 96], [Fox et al., 97], [Brock & Khatib, 99]
  - Nearness Diagram Navigation
    [Minguez et al., 2001, 2002]
  - Vector-Field-Histogram+
    [Ulrich & Borenstein, 98]
  - Extended Potential Fields
    [Khatib & Chatila, 95]

Collision Avoidance

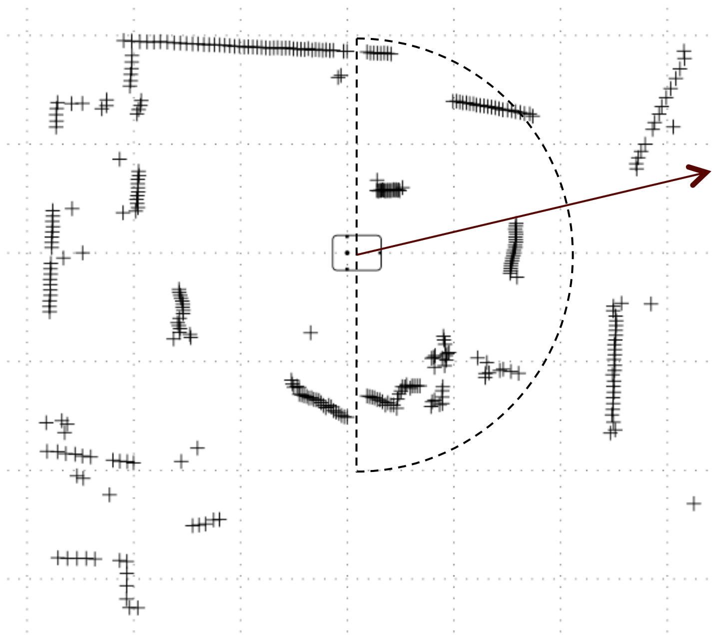
- Integration into general motion planning?
- It is common to subdivide the problem into a global and local planning task:
  - An approximate **global planner** computes paths ignoring the kinematic and dynamic vehicle constraints
  - An accurate **local planner** accounts for the constraints and generates (sets of) feasible local trajectories ("collision avoidance")
- What do we loose? What do we win?

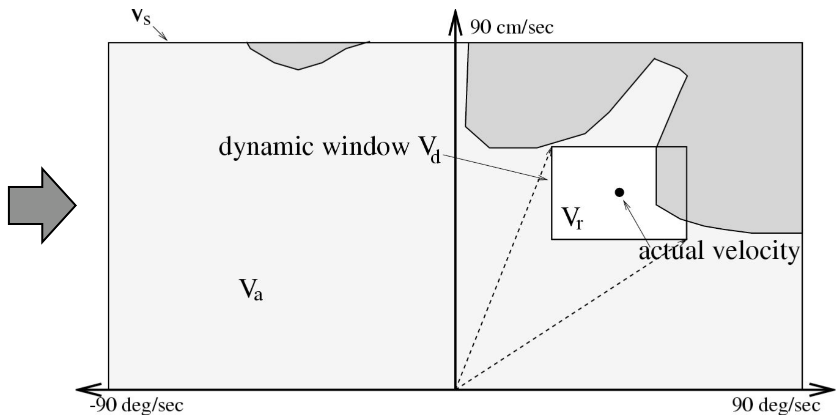
Two-layered Architecture



Dynamic Window Approach

- **Given:** path to goal (a set of via points), range scan of the local vicinity, dynamic constraints
- **Wanted:** collision-free, safe, and fast motion towards the goal





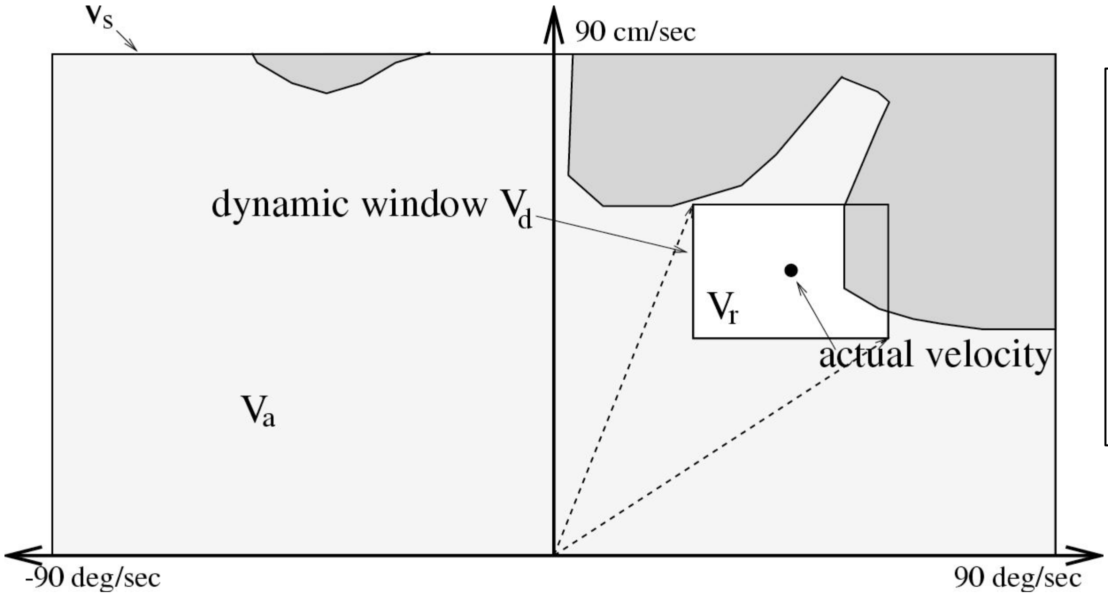
Dynamic Window Approach

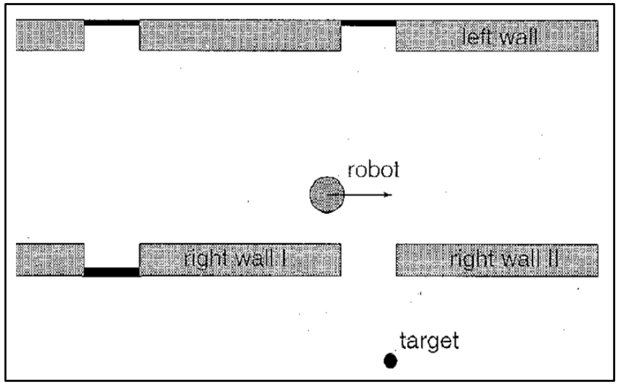
- **Assumption:** robot takes motion commands of the form (v,ω)
- This is saying that the robot moves (instantaneously) on **circular arcs** with radius $r = v / \omega$

- **Question:** which (v,ω)'s are
  - **reasonable**: that bring us to the goal?
  - **admissible**: that are collision-free?
  - **reachable**: under the vehicle constraints?

DWA Search Space

- 2D velocity search space





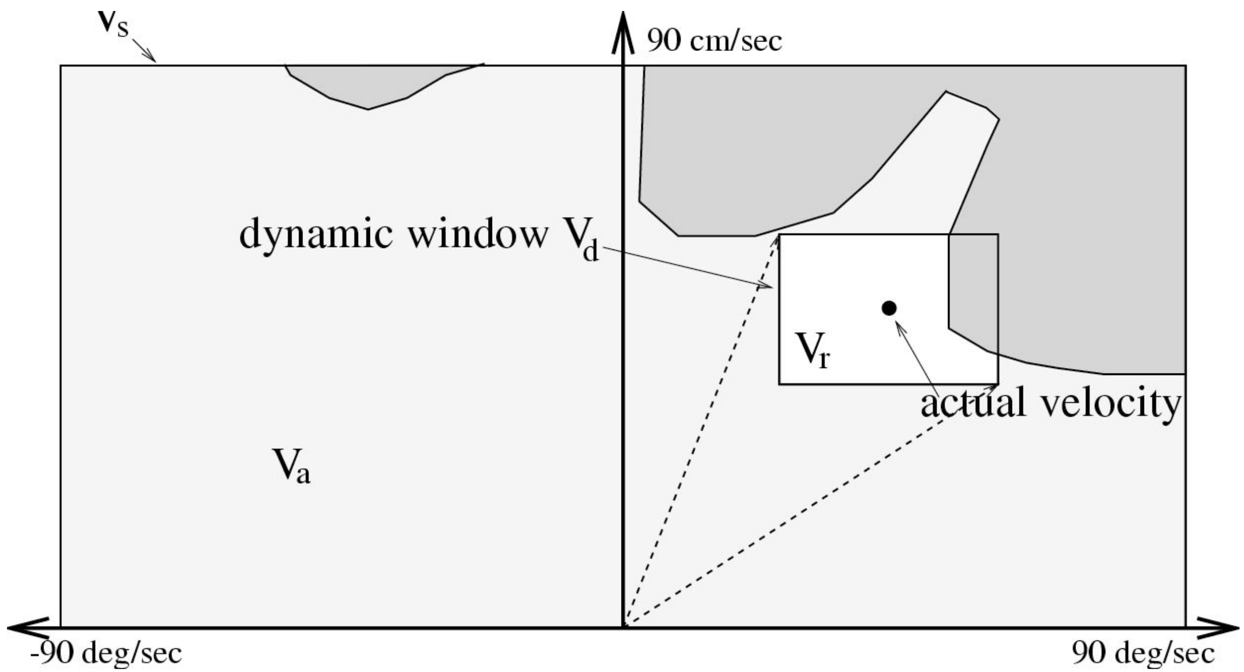
  - V_s = all possible speeds of the robot
  - V_a = obstacle free area
  - V_d = speeds reachable within one time frame given acceleration constraints

$$Space = V_s \cap V_a \cap V_d$$

Reachable Velocities

- Speeds are reachable if

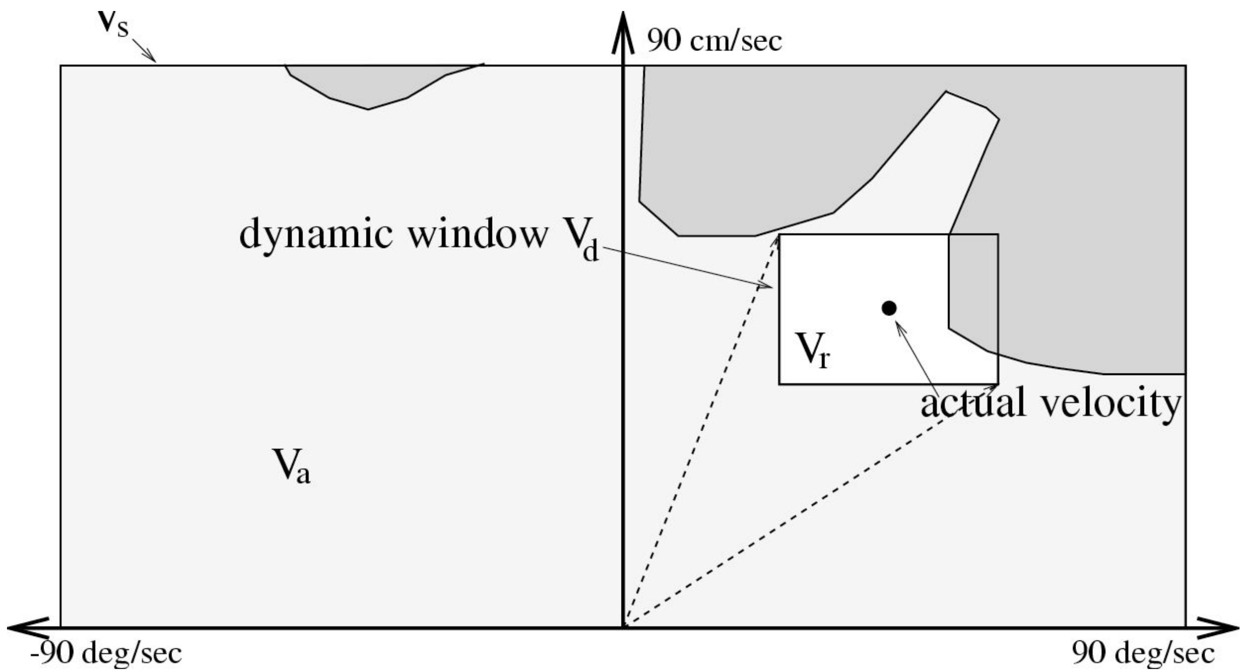
$$V_d = \{(v, \omega) \quad | \quad v \in [v - a_{trans}t, v + a_{trans}t] \quad \wedge \\ \omega \in [\omega - a_{rot}t, \omega + a_{rot}t]\}$$



Admissible Velocities

- Speeds are admissible if

$$V_a = \{(v,\omega) \;\mid\; v \leq \sqrt{2\mathrm{dist}(v,\omega)a_{trans}} \;\wedge\; \omega \leq \sqrt{2\mathrm{dist}(v,\omega)a_{rot}}\}$$



Dynamic Window Approach

- How to choose (v,ω) ?
- Pose the problem as an **optimization problem** of an objective function within the dynamic window, search the maximum
- The objective function is a **heuristic navigation function**
- This function encodes the incentive to minimize the travel time by "driving **fast** and **safe** in the **right direction**"

Dynamic Window Approach

- Heuristic navigation function
- Planning restricted to (v,ω)-space
- Here: assume to have precomputed goal distances from NF1 algorithm

Navigation Function: *[Brock & Khatib, 99]*

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

89

Dynamic Window Approach

- Heuristic navigation function
- Planning restricted to (v,ω)-space
- Here: assume to have precomputed goal distances from NF1 algorithm

Navigation Function: *[Brock & Khatib, 99]*

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

Maximizes velocity

Dynamic Window Approach

- Heuristic navigation function
- Planning restricted to *(v,ω)*-space
- Here: assume to have precomputed goal distances from NF1 algorithm

Navigation Function: *[Brock & Khatib, 99]*

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

Maximizes velocity

Rewards alignment to NF1/A* gradient

Dynamic Window Approach

- Heuristic navigation function
- Planning restricted to (v,ω)-space
- Here: assume to have precomputed goal distances from NF1 algorithm

Navigation Function: *[Brock & Khatib, 99]*

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

Maximizes velocity

Rewards alignment to NF1/A* gradient

Rewards large advances on NF1/A* path

Dynamic Window Approach

- Heuristic navigation function
- Planning restricted to (v,ω)-space
- Here: assume to have precomputed goal distances from NF1 algorithm

Navigation Function: *[Brock &*

Comes in when goal region reached

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

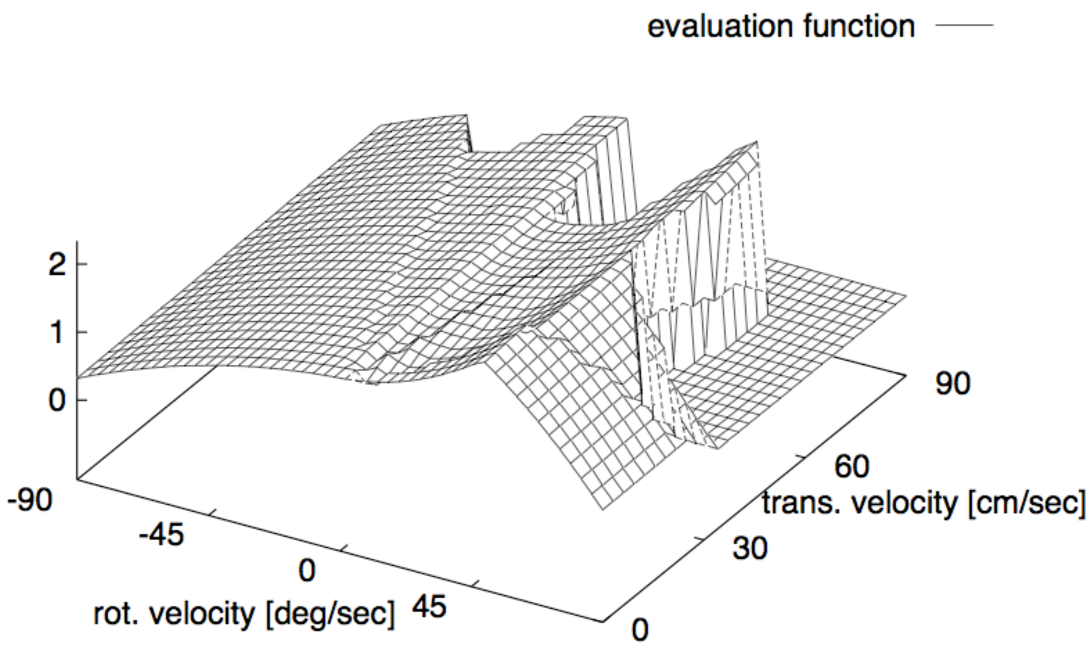
Maximizes velocity

Rewards alignment to NF1/A* gradient

Rewards large advances on NF1/A* path

Dynamic Window Approach

- Navigation function example



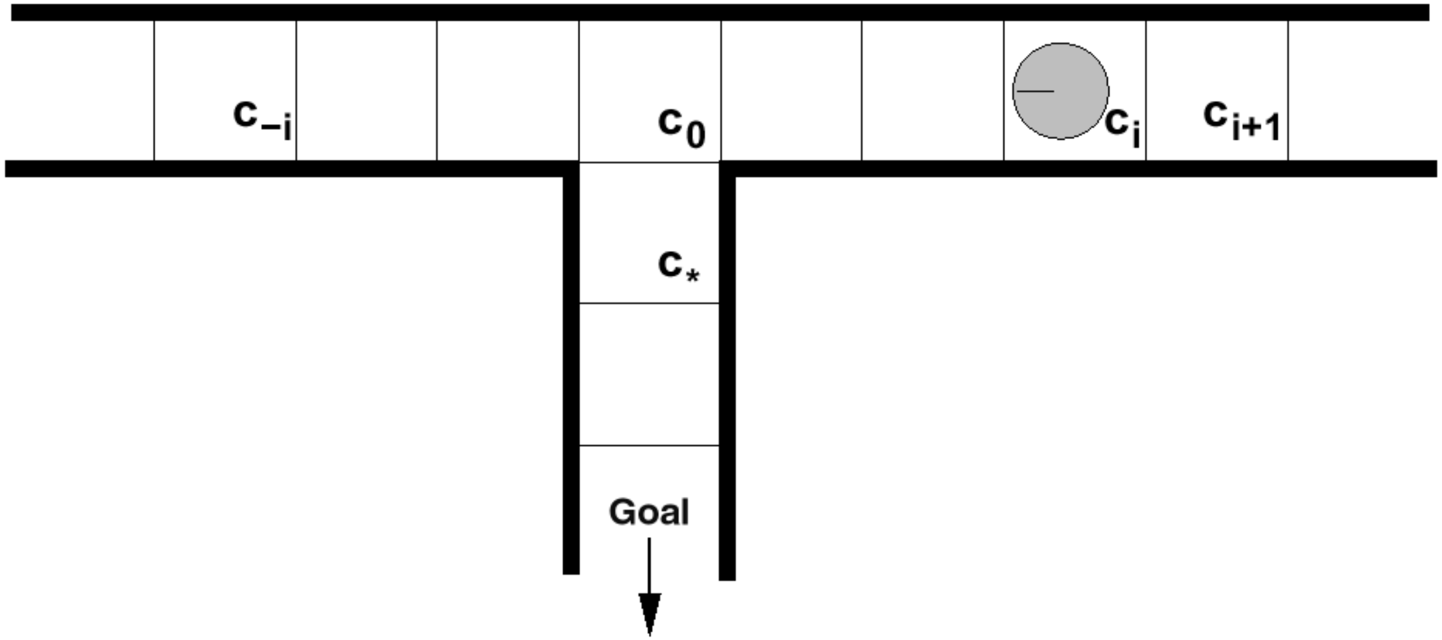
- Now perform search/optimization
- Find maximum

Dynamic Window Approach

- Reacts quickly at low CPU requirements
- Guides a robot on a collision free path
- Successfully used in many real-world scenarios

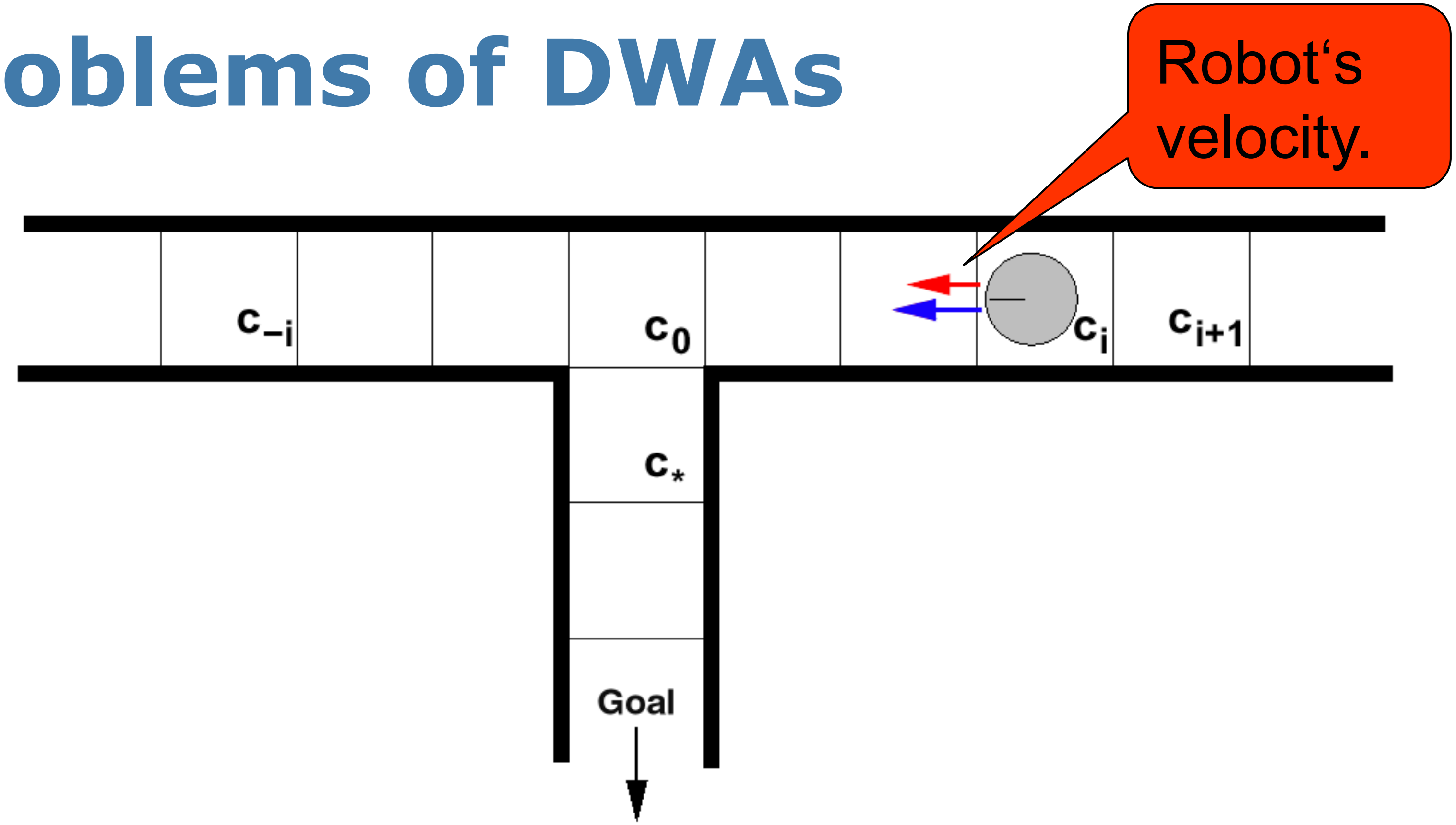
- Resulting trajectories sometimes suboptimal
- Local minima might prevent the robot from reaching the goal location (regular DWA)
- Global DWA with NF1 overcomes this problem

Problems of DWAs



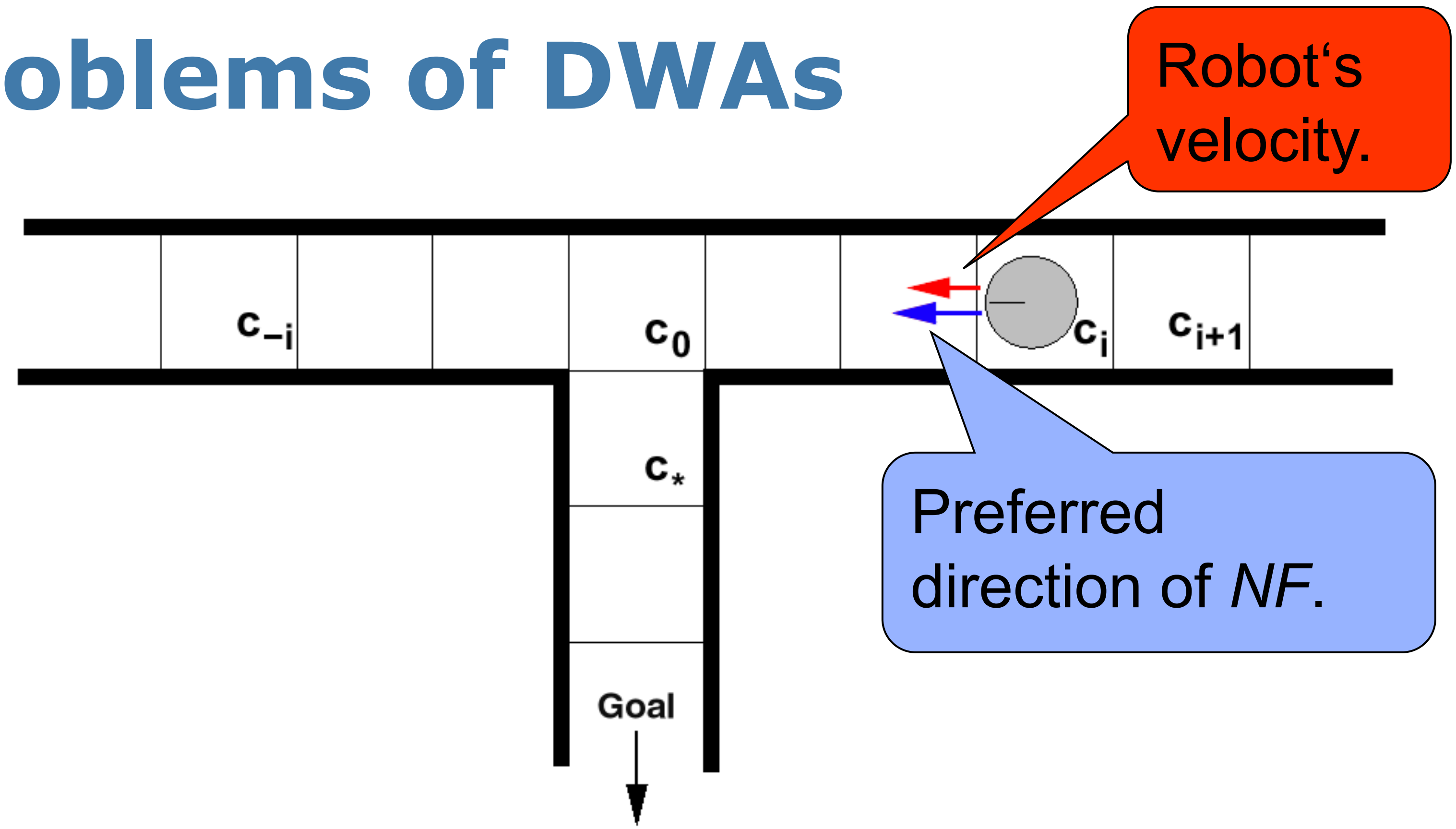
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



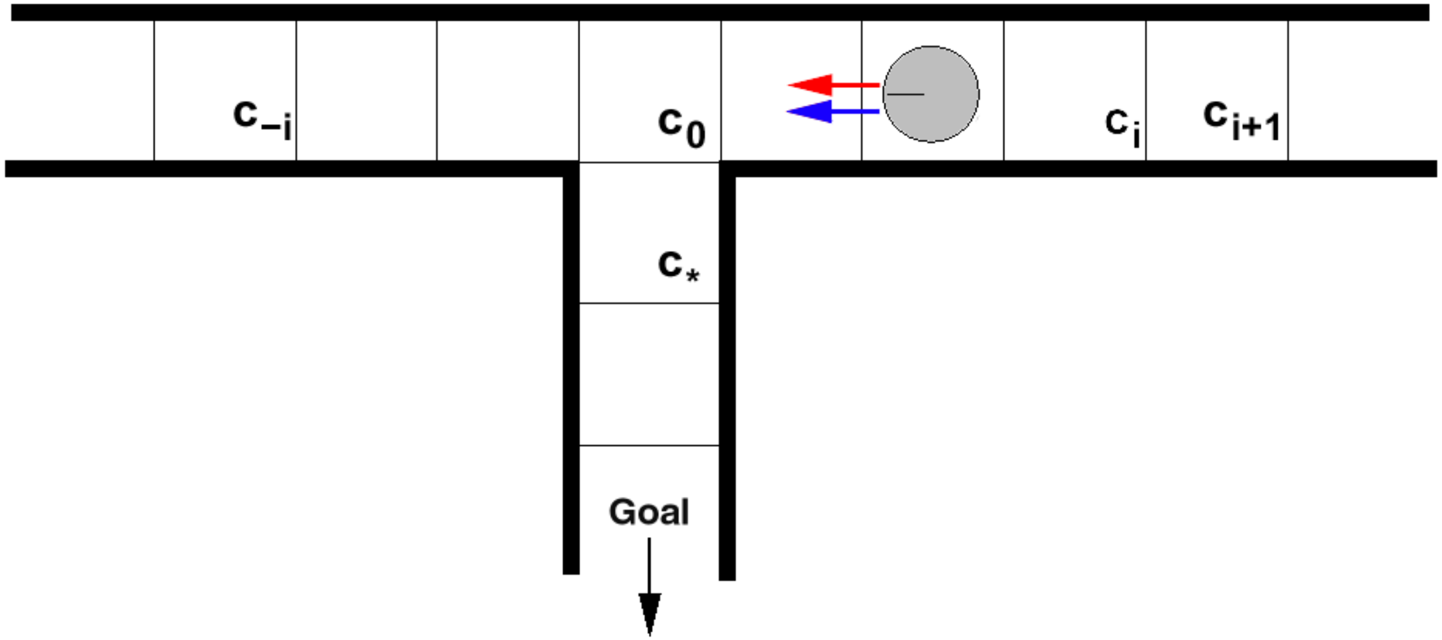
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



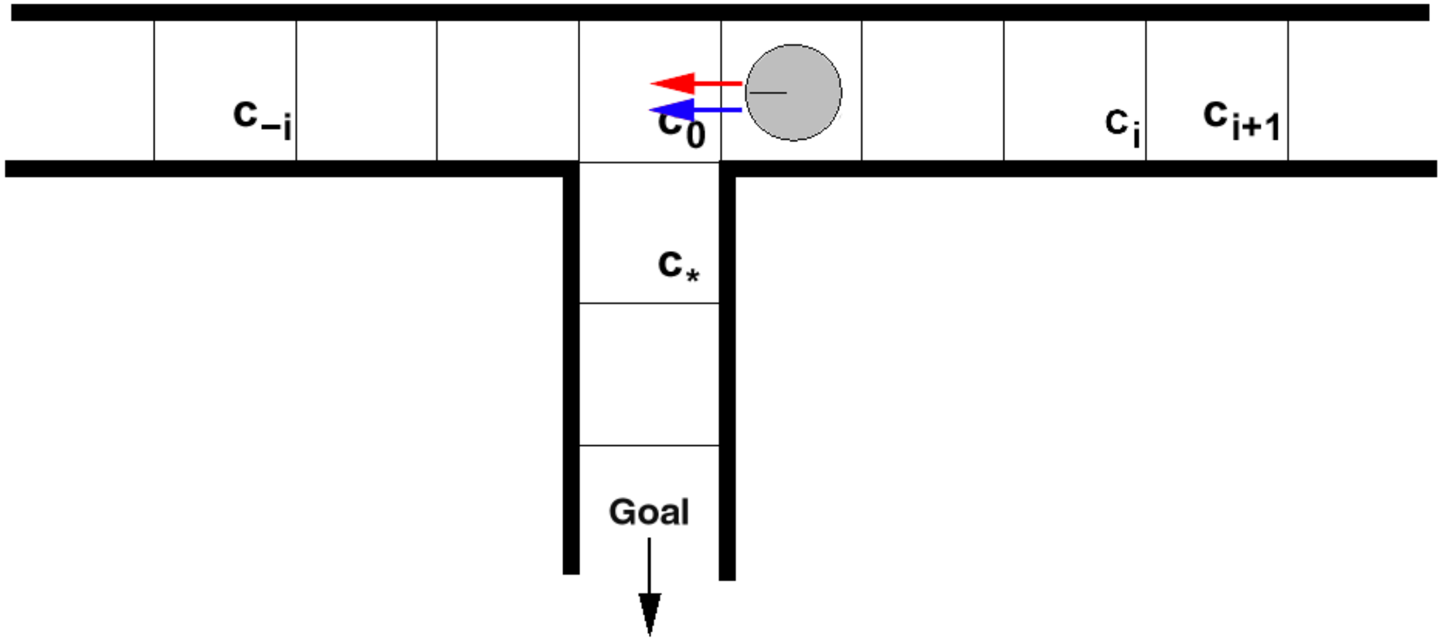
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



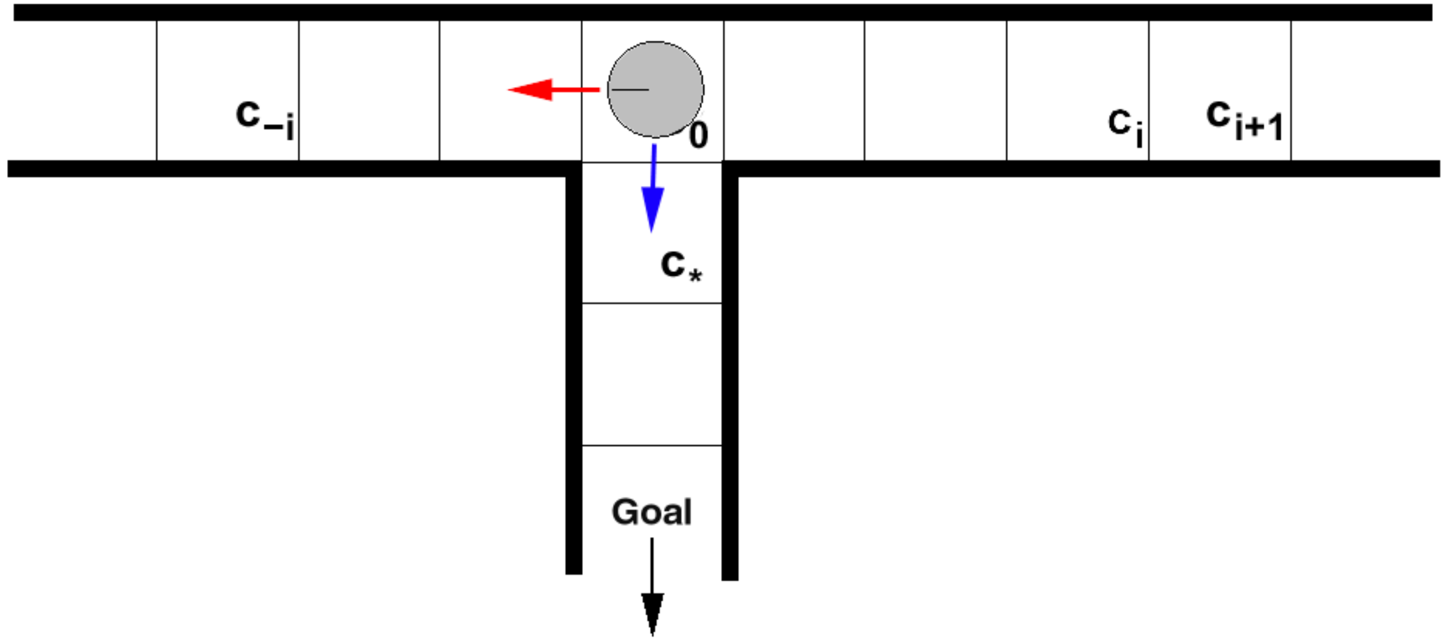
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



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

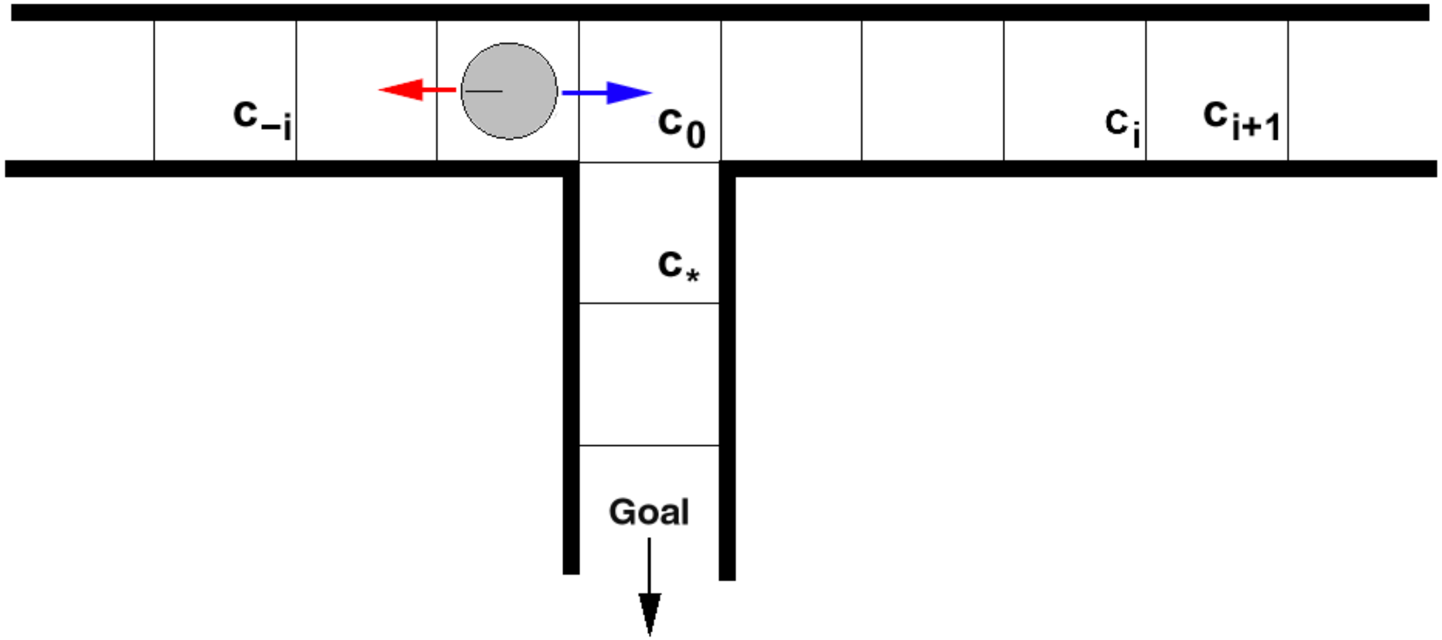
Problems of DWAs



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

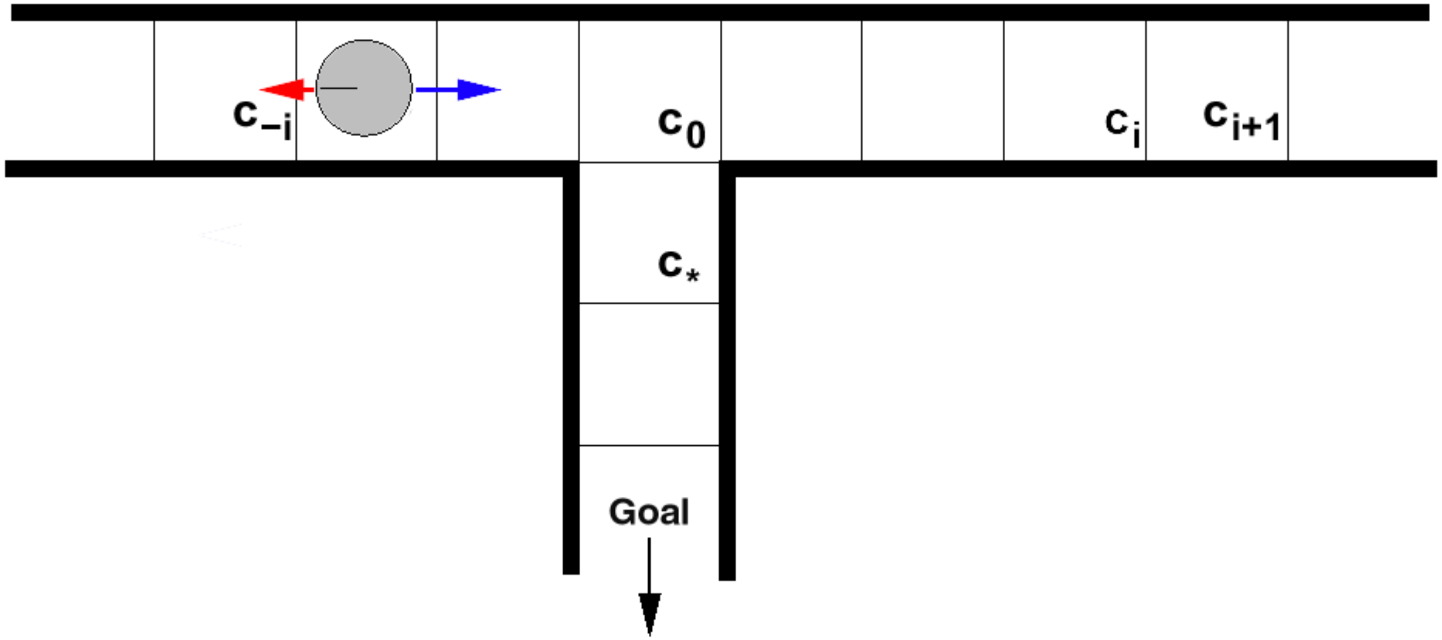
The robot drives too fast at c_0 to enter corridor facing south.

Problems of DWAs



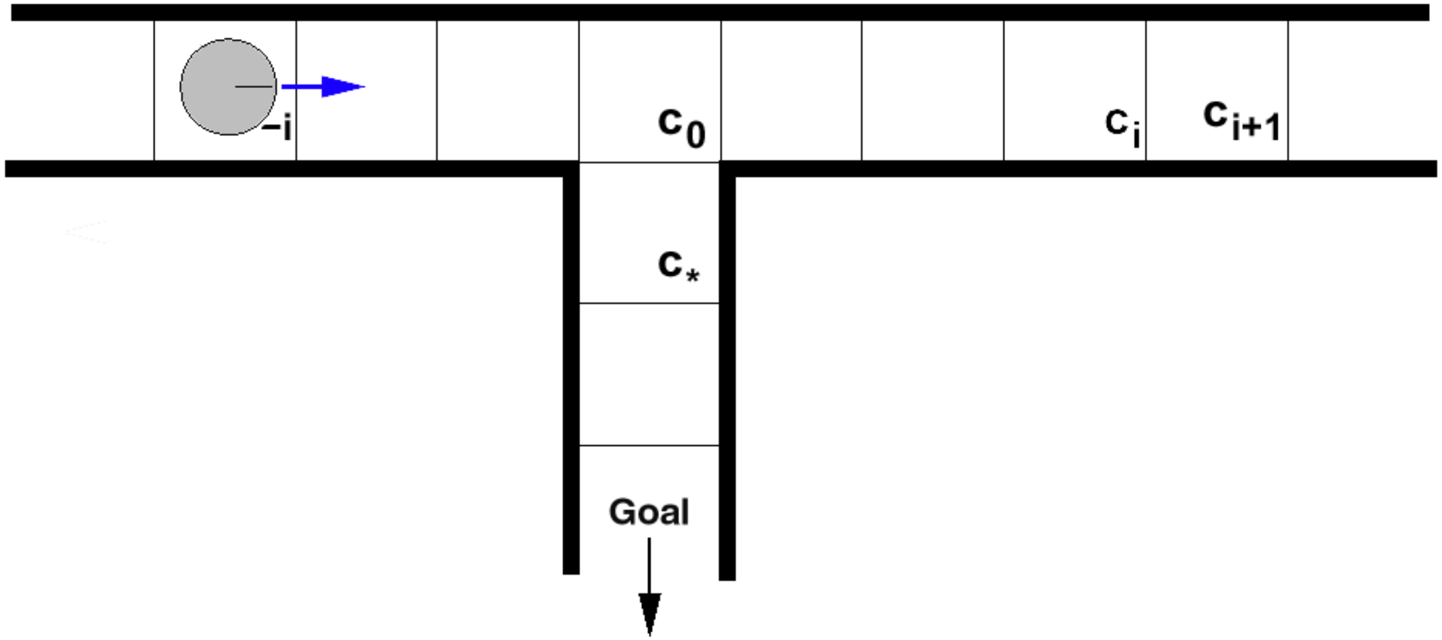
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



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

Problems of DWAs



- Same situation as in the beginning

➔ DWAs have problems to reach the goal

Problems of DWAs

- Typical problem in a real world situation:

- Robot does not slow down early enough to enter the doorway.

Alternative: 5D-Planning

- Plans in the **full** $<x,y,\theta,v,\omega>$-**configuration** space using A*
  - Considers the robot's kinematic constraints

- **Idea:** search in the discretized $<x,y,\theta,v,\omega>$-space

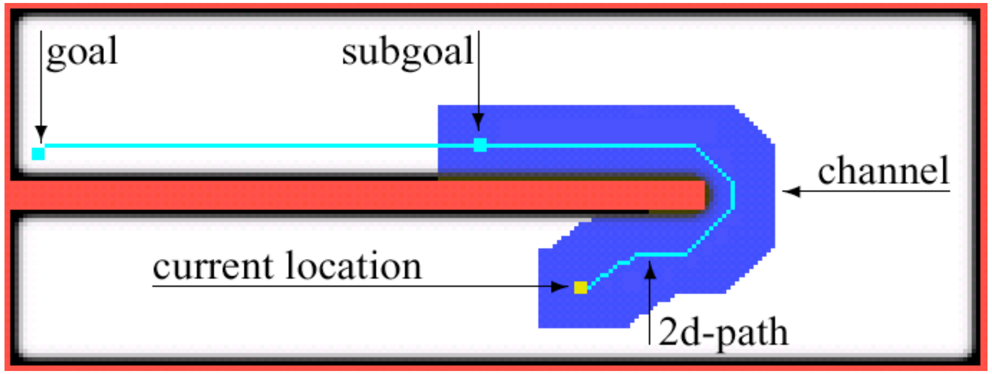
- **Problem:** search space too large to be explored in real-time

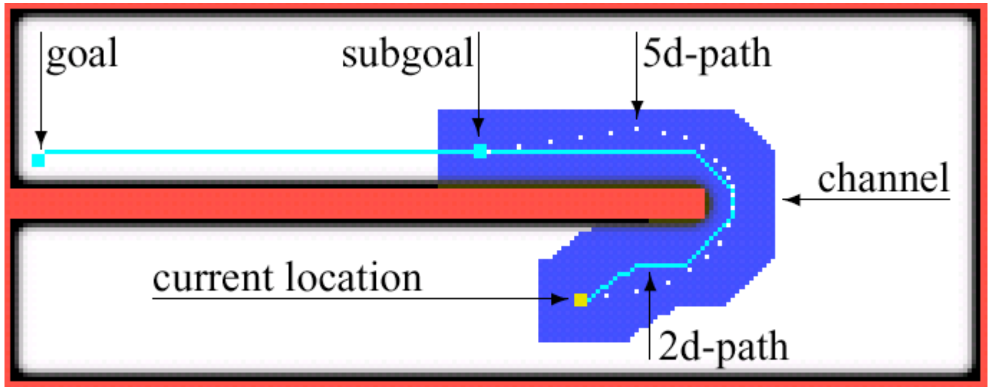
- **Solution:** restrict the full search space to "channels"

5D-Planning

- Use A* to find a trajectory in the **2D** $<x,y>$**-space**
- Choose a **subgoal** lying on the 2D-path within the channel
- Use A^* in the **"channel" 5D-space** to find a sequence of steering commands to reach the subgoal

5D-Planning Example





Summary (1 of 3)

- Motion planning lives in the **C-space**
- **Combinatorial** planning methods scale poorly with C-space dimension and non-linearity but are **complete** and **optimal**
- **Sampling-based** planning methods have weaker guarantees but are more efficient
- They all produce a **road map** that captures the connectivity of the C-space
- For planning on the road map, use **heuristic search** methods such as A*

Summary (2 of 3)

- Deterministic value iteration or Dijkstra yields the **optimal heuristic** for A*. Precompute if on-line replanning is likely
- A* in smoothed grid maps helps to keep the robot **away** from obstacles
- Any-angle A* methods produce **shorter** paths with **fewer** heading changes
- D*/D* Lite **avoids** replanning from **scratch** and finds the (usually few) nodes to be updated for on-line replanning

Summary (3 of 3)

- In highly dynamic environments, reactive **collision avoidance** methods that account for the **kinematic** and **dynamics** vehicle constraints become necessary
- Decoupling into an approximative **global** and an accurate **local** planning problem, integration using a layered architecture
- The Dynamic Window Approach optimizes a navigation function to trade off **feasible**, **reasonable**, and **admissible** motions

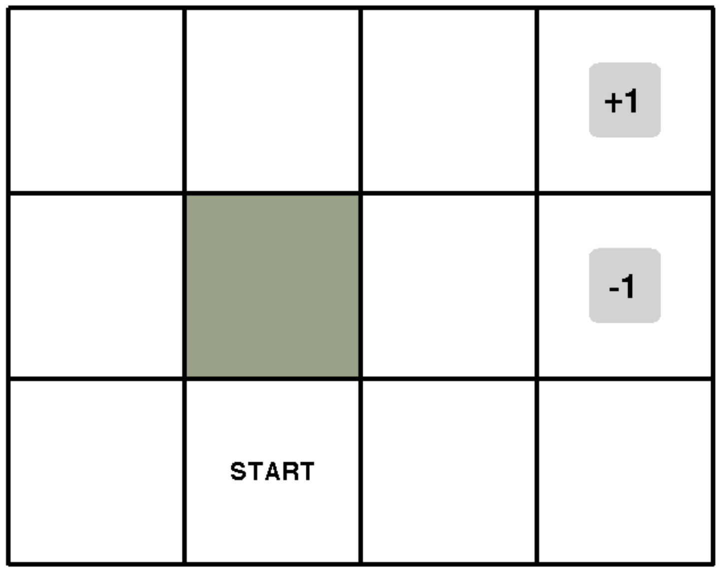
Uncertain Path Execution

- Have you ever **become lost** while trying to follow a **path** (e.g. printed out from Google maps)?
- Problem: **path execution** is inherently **uncertain!**
- Even the best **path** is worthless if the robot is unable to follow it
- Reasons: Underlying trajectoriy controller, DWA, imperfect models of map/dynamics

➔ Instead of a plan, you need a **policy**

Markov Decision Process

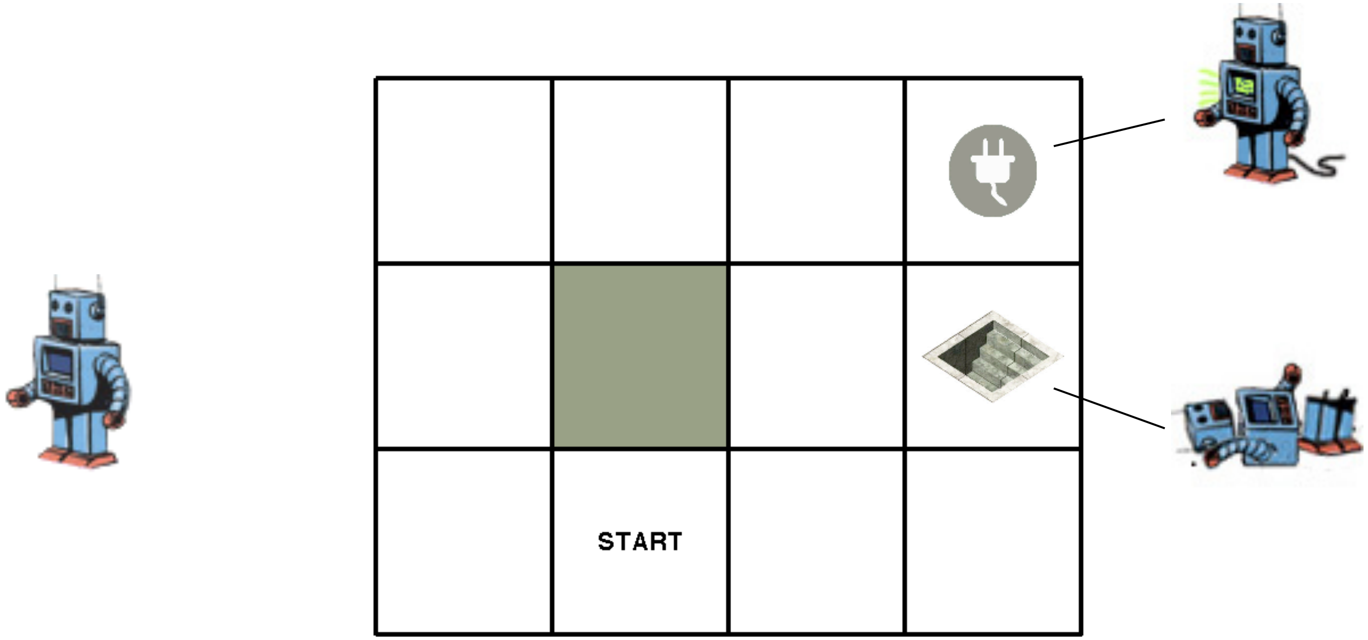
- Consider an agent acting in this environment



- Its mission is to reach the goal marked by +1 avoiding the cell labelled -1

Markov Decision Process

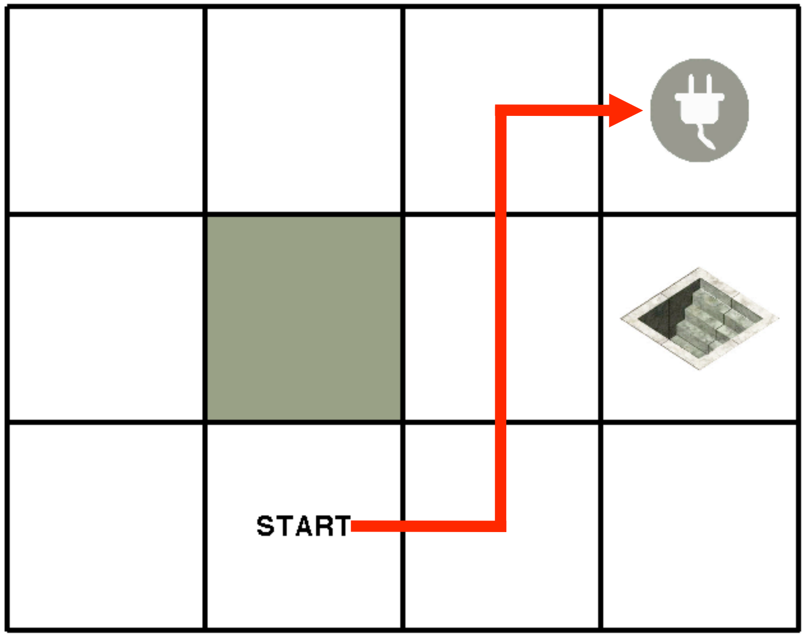
- Consider an agent acting in this environment



- Its mission is to reach the goal marked by +1 avoiding the cell labelled -1

Markov Decision Process

- Easy! Use a search algorithm such as A*



- Best solution (shortest path) is the action sequence *[Right, Up, Up, Right]*

What is the problem?

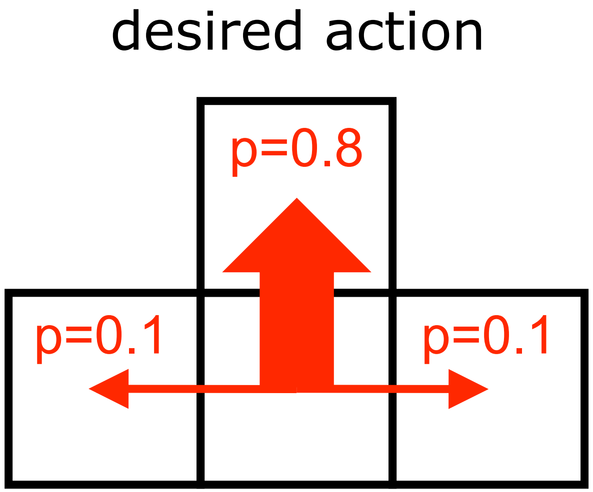
- Consider a non-perfect system in which actions are performed with a **probability less than 1**
- What are the best actions for an agent under this constraint?

- Example: a mobile robot does not *exactly* perform a desired motion
- Example: human navigation

→ Uncertainty about performing actions!

MDP Example

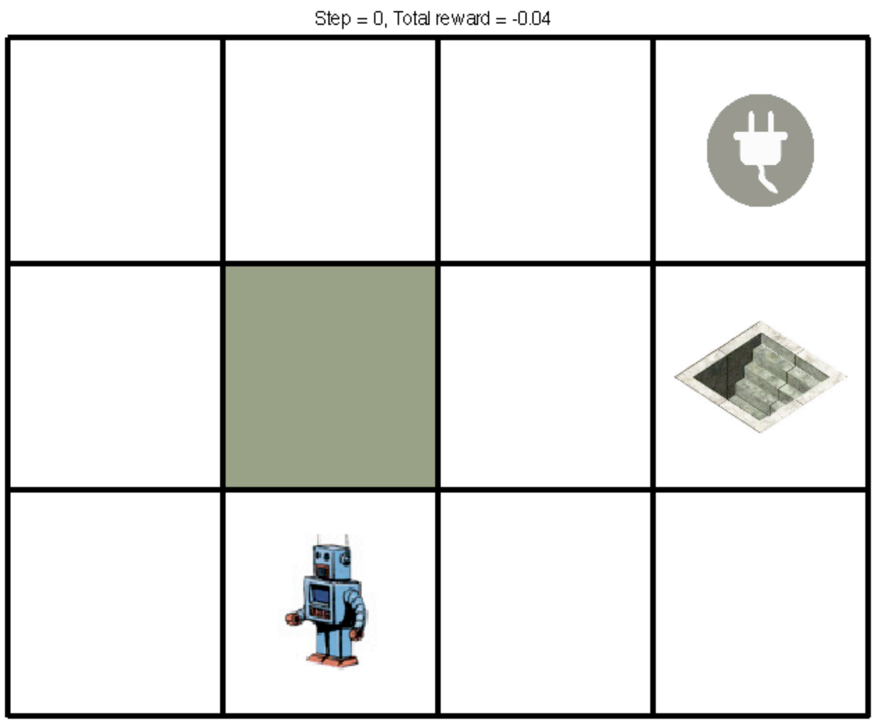
- Consider the **non-deterministic transition model** (N / E / S / W):



- Intended action is executed with p=0.8
- With p=0.1, the agent moves left or right
- Bumping into a wall "reflects" the robot

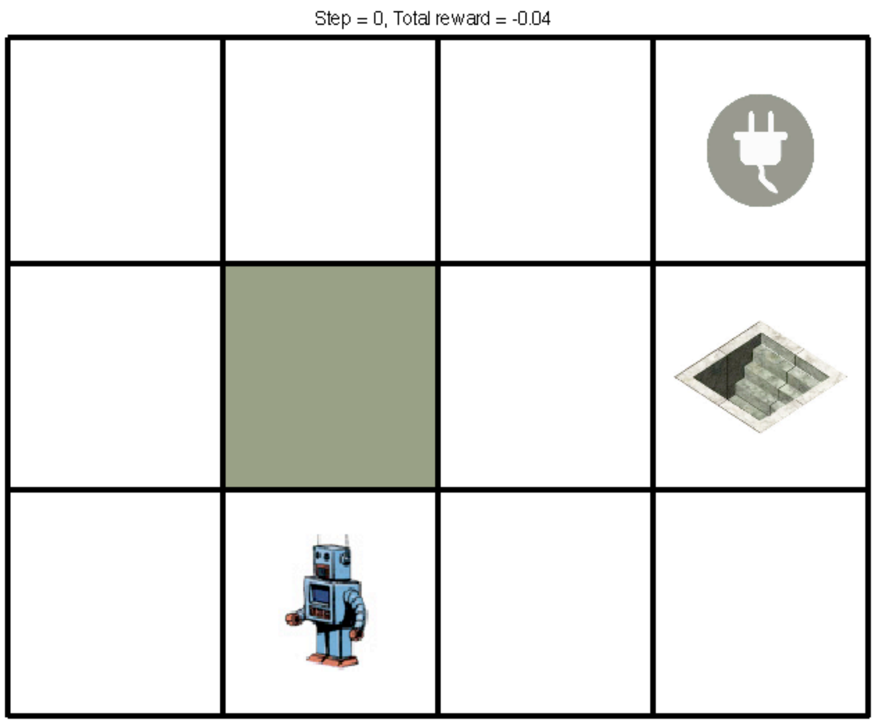
MDP Example

- Executing the **A* plan** in this environment



MDP Example

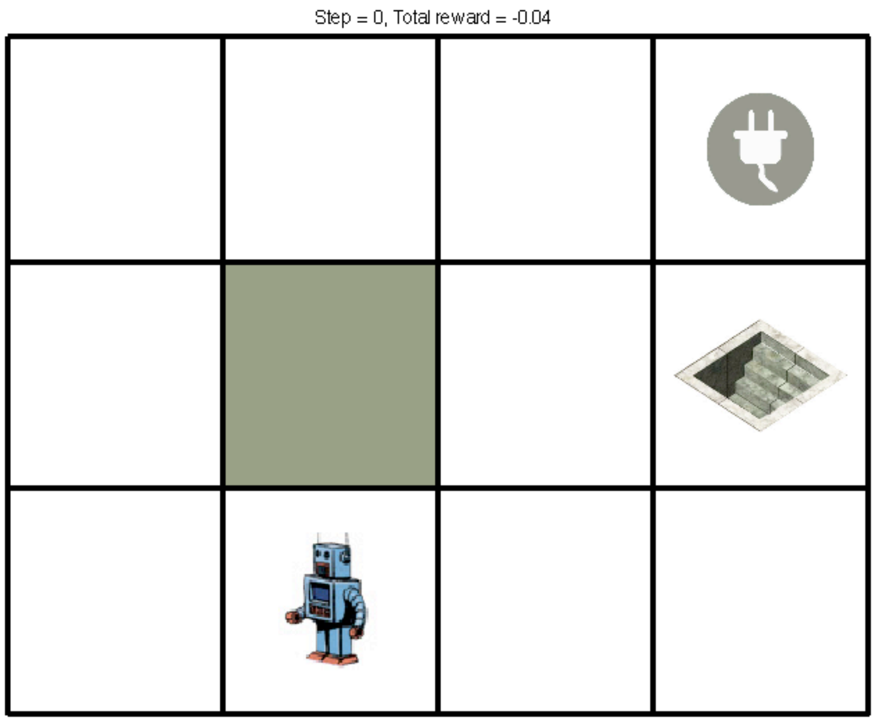
- Executing the **A* plan** in this environment



But: transitions are non-deterministic!

MDP Example

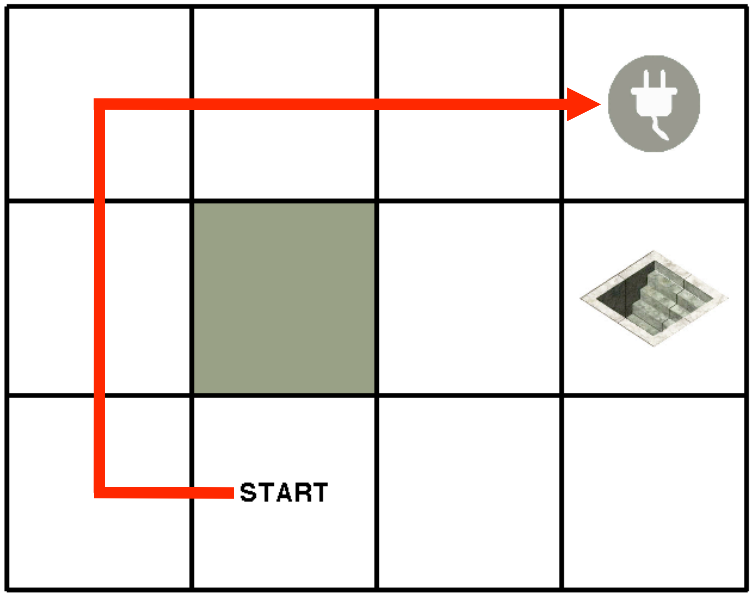
- Executing the **A* plan** in this environment



This will happen sooner or later...

MDP Example

- Use a **longer** path with **lower** probability to end up in cell labelled **-1**



- This path has the **highest overall utility**
- Probability $0.8^6 = 0.2621$

Transition Model

- The probability to reach the next state s' from state s by choosing action a

  $$T(s, a, s')$$

  is called **transition model**

Markov Property:

The transition probabilities from s to s' **depend only on the current state** s and not on the history of earlier states

Reward

- In each state s, the agent receives a **reward** $R(s)$

- The reward may be **positive** or **negative** but must be **bounded**

- This can be generalized to be a function $R(s,a,s')$. Here: consider only $R(s)$, does not change the problem

Reward

- In our example, the reward is **-0.04** in all states (e.g. the cost of motion) except the terminal states (that have rewards **+1**/**-1**)
- A negative reward gives agent an **in-centive to reach the goal quickly**
- Or: "living in this environment is not enjoyable"

-0.04	-0.04	-0.04	+1
-0.04		-0.04	-1
-0.04	-0.04	-0.04	-0.04

MDP Definition

- Given a **sequential decision problem** in a fully observable, stochastic environment with a known Markovian transition model

- Then a **Markov Decision Process** is defined by the components
  - *Set of states:* S
  - *Set of actions:* A
  - *Initial state:* s_0
  - *Transition model:* $T(s, a, s')$
  - *Reward funciton:* $R(s)$

Policy

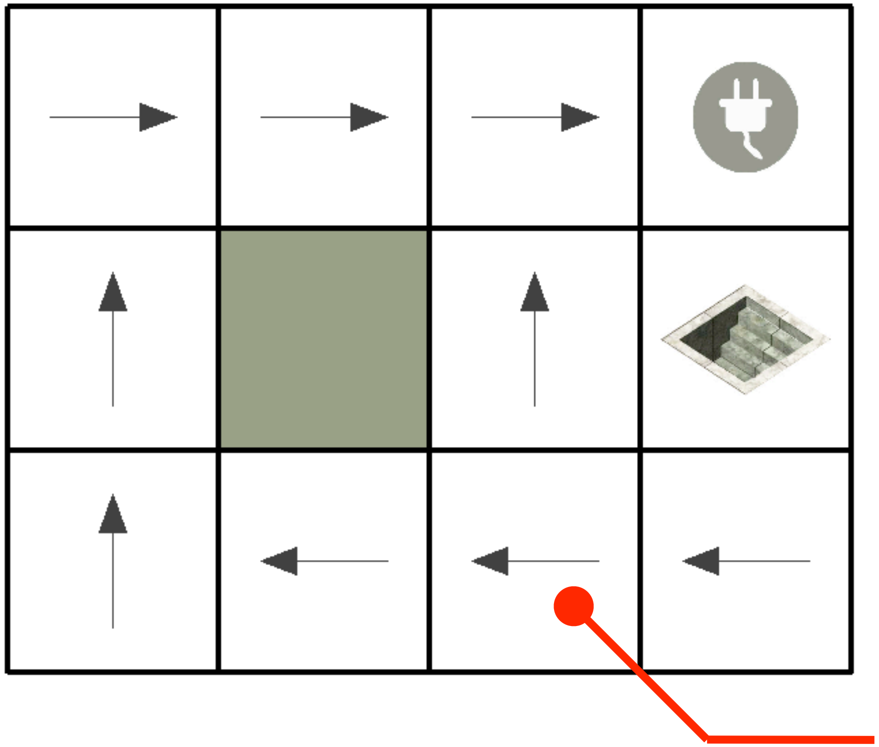
- An MDP solution is called **policy** π
- A policy is a mapping from states to actions

$$policy : States \mapsto Actions$$

- In each state, a policy tells the agent **what to do next**
- Let $\pi(s)$ be the *action* that π specifies for s
- Among the many policies that solve an MDP, the **optimal policy** π* is what we seek. We'll see later what *optimal* means

Policy

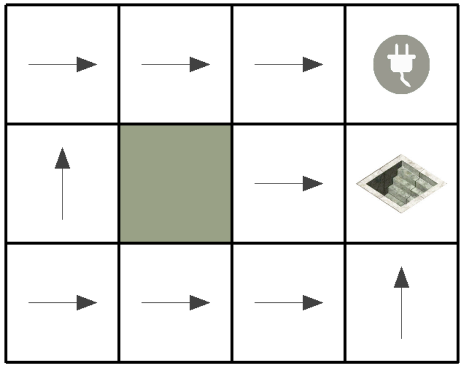
- The optimal policy for our example



Conservative choice
Take long way around as the cost per step of -0.04 is small compared with the penality to fall down the stairs and receive a **-1** reward

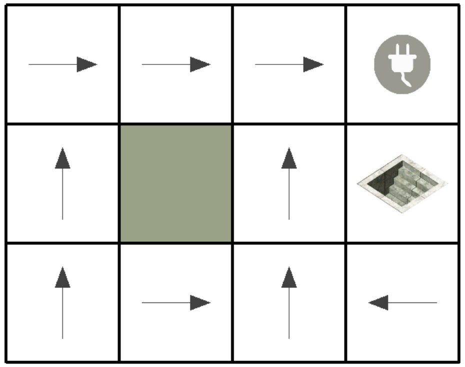
Policy

- When the balance of risk and reward changes, **other policies are optimal**



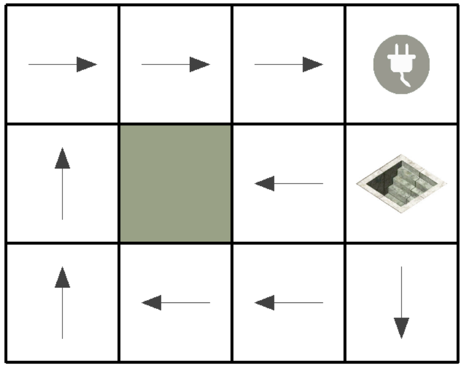
$R < -1.63$

Leave as soon as possible



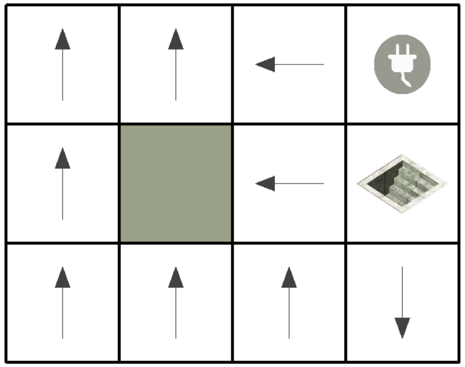
$-0.43 < R < -0.09$

Take shortcut, minor risks



$-0.02 < R < 0$

No risks are taken



$R > 0$

Never leave (inf. #policies)

Utility of a State

- The **utility of a state** $U(s)$ quantifies the **benefit** of a state for the **overall task**
- We first define $U^{\pi}(s)$ to be the **expected utility of all state sequences that start in** *s* given π

$$U^{\pi}(s) \quad = \quad E\left[\sum_{t=0}^{\infty} R(s_t) \mid \pi, s_0 = s\right]$$

- $U(s)$ evaluates (and encapsulates) all possible futures **from** s **onwards**

Utility of a State

- With this definition, we can express $U^{\pi}(s)$ as a **function of its next state** s'

$$
\begin{aligned}
U^{\pi}(s) &= E\left[\sum_{t=0}^{\infty} R(s_t) \mid \pi, s_0 = s\right] \\
&= E\left[R(s_0) + R(s_1) + R(s_2) + \ldots \mid \pi, s_0 = s\right] \\
&= E\left[R(s_0) \mid s_0 = s\right] + E\left[R(s_1) + R(s_2) + \ldots \mid \pi\right] \\
&= R(s) + E\left[\sum_{t=0}^{\infty} R(s_t) \mid \pi, s_0 = s'\right] \\
&= R(s) + U^{\pi}(s')
\end{aligned}
$$

Optimal Policy

- The utility of a state allows us to apply the **Maximum Expected Utility principle** to define the optimal policy π*

- The **optimal policy** π* in s chooses the action a that maximizes the expected utility of s (and of s')

$$\pi^*(s) \quad = \quad \underset{a}{\operatorname{argmax}} \; E\left[U^\pi(s)\right]$$

- Expectation taken over all policies

Optimal Policy

- Substituting $U^{\pi}(s)$

$$
\begin{aligned}
\pi^*(s) &= \underset{a}{\operatorname{argmax}}\ E\left[U^{\pi}(s)\right] \\
&= \underset{a}{\operatorname{argmax}}\ E\left[R(s) + U^{\pi}(s')\right] \\
&= \underset{a}{\operatorname{argmax}}\ E\left[R(s)\right] + E\left[U^{\pi}(s')\right] \\
&= \underset{a}{\operatorname{argmax}}\ E\left[U(s')\right] \\
&= \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')\,U(s')
\end{aligned}
$$

- Recall that $E[X]$ is the weighted average of all possible values that X can take on

Utility of a State

- The **true utility of a state** *U(s)* is then obtained by application of the optimal policy, i.e. $U^{\pi^*}(s) = U(s)$. We find

$$
\begin{aligned}
U(s) &= \max_a E\left[U^\pi(s)\right] \\
&= \max_a E\left[R(s) + U^\pi(s')\right] \\
&= \max_a E\left[R(s)\right] + E\left[U^\pi(s')\right] \\
&= R(s) + \max_a E\left[U(s')\right] \\
&= \underline{R(s) + \max_a \sum_{s'} T(s,a,s')\, U(s')}
\end{aligned}
$$

Utility of a State

- This result is noteworthy:

$$U(s) = R(s) + \max_a \sum_{s'} T(s,a,s')\,U(s')$$

  We have found a direct relationship between the **utility of a state** and the **utility of its neighbors**

- The utility of a state is the immediate reward for that state plus the expected utility of the next state, **provided** the agent chooses the **optimal** action

Bellman Equation

$$\boxed{U(s) = R(s) + \max_a \sum_{s'} T(s,a,s')\, U(s')}$$

- For each state there is a Bellman equation to compute its utility
- There are n **states** and n **unknowns**
- Solve the system using Linear Algebra?
- No! The max-operator that chooses the optimal action makes the system nonlinear
- We must go for an **iterative approach**

Discounting

We have made a **simplification** on the way:

- The utility of a state sequence is often defined as the sum of **discounted** rewards

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \underline{\gamma^t} R(s_t) \mid \pi, s_0 = s\right]$$

  with $0 \leq \gamma \leq 1$ being the *discount factor*

- Discounting says that **future** rewards are **less significant** than **current** rewards. This is a natural model for many domains
- The other expressions change accordingly

Separability

We have made an **assumption** on the way:

- Not all utility functions (for state sequences) can be used
- The utility function must have the **property of separability** (a.k.a. stationarity), e.g. additive utility functions:

$$U([s_0 + s_1 + \ldots + s_n]) = R(s_0) + U([s_1 + \ldots + s_n])$$

- Loosely speaking: the preference between two state sequences is unchanged over different start states

Utility of a State

- The **state utilities** for our example

0.812	0.868	0.918	+1
0.762		0.66	-1
0.705	0.655	0.611	0.388

- Note that utilities are higher closer to the goal as fewer steps are needed to reach it

Iterative Computation

Idea:

- The utility is computed iteratively:

$$U_{i+1}(s) \leftarrow R(s) + \max_a \sum_{s'} T(s, a, s')\, U_i(s')$$

- Optimal utility: $U^* = \lim_{t \to \infty} U_t$
- Abort, if change in utility is below a threshold

Dynamic Programming

- The utility function is the basis for **Dynamic Programming**
- Fast solution to compute n-step decision problems
- Naive solution: $O(\ |A|^n\ )$
- Dynamic Programming: $O(\ n\ |A|\ |S|\ )$
- But: what is the correct value of n?
- If the graph has loops: $n \longrightarrow \infty$

The Value Iteration Algorithm

Algorithm 1: Value Iteration

In: An MDP with
– States and action sets S, A,
– Transition model $T(s, a, s')$,
– Reward function $R(s)$,
– Discount factor γ
Out: The utility of all states U

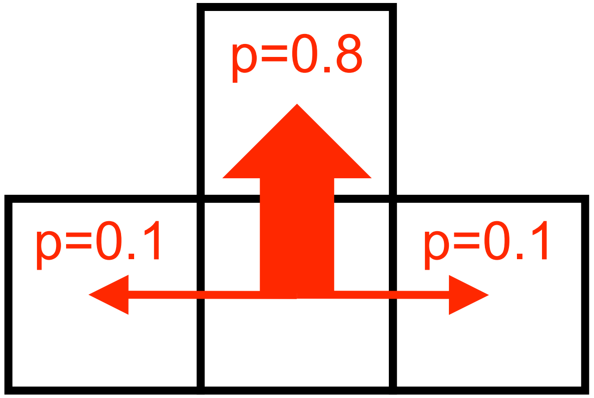
$U' \leftarrow 0$
repeat
  $U \leftarrow U'$
  foreach *state* s *in* S **do**
    $U(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s')$
  end
until *close-enough*(U, U')
return U

Value Iteration Example

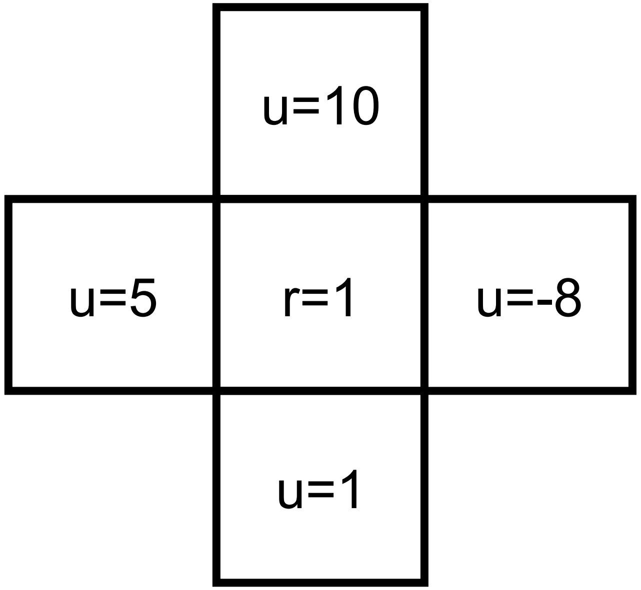
- Calculate utility of the center cell

$$U_{i+1}(s) \leftarrow R(s) + \max_a \sum_{s'} T(s,a,s')\, U_i(s')$$

desired action = *Up*



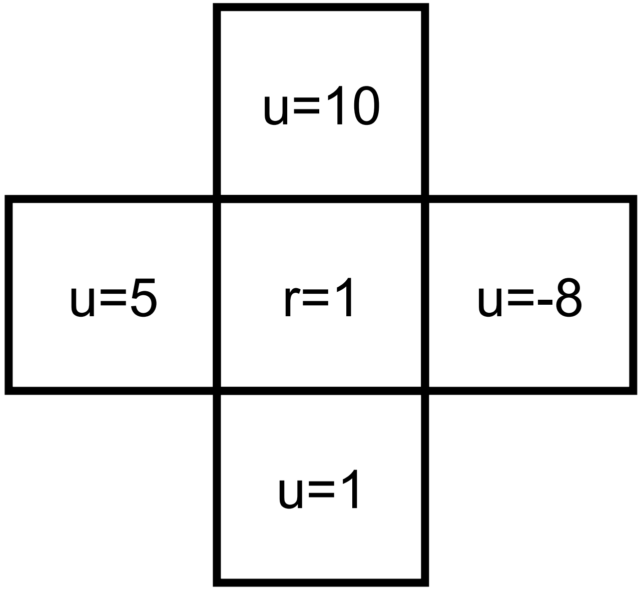
Transition Model



State space
(u=utility, r=reward)

Value Iteration Example

$$U_{i+1}(s) \leftarrow R(s) + \max_a \sum_{s'} T(s,a,s')\, U_i(s')$$



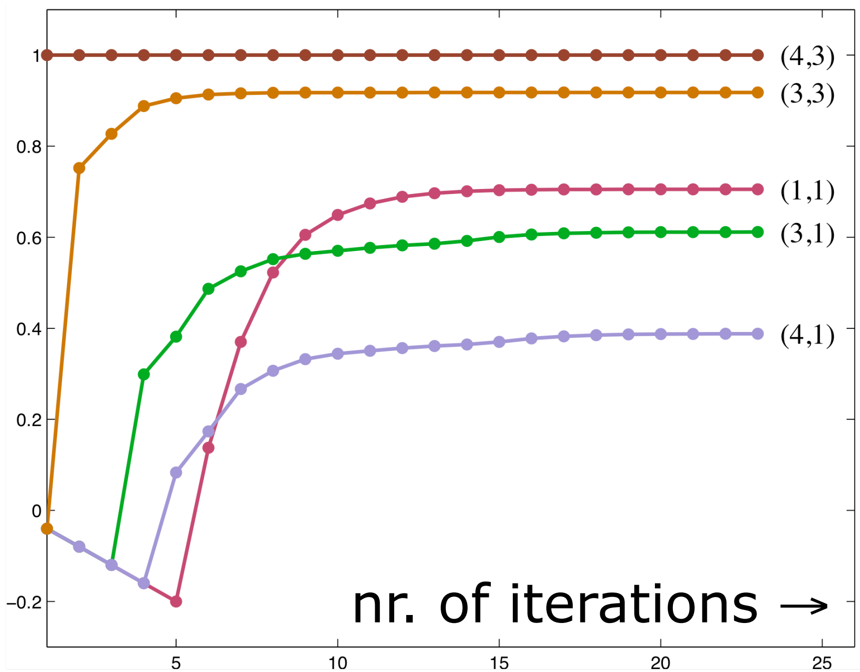
$$\begin{aligned}
&= \mathit{reward} + \mathsf{max}\{ \\
&\quad 0.1 \cdot 1 + 0.8 \cdot 5 + 0.1 \cdot 10 \quad (\leftarrow), \\
&\quad 0.1 \cdot 5 + 0.8 \cdot 10 + 0.1 \cdot -8 \quad (\uparrow), \\
&\quad 0.1 \cdot 10 + 0.8 \cdot -8 + 0.1 \cdot 1 \quad (\rightarrow), \\
&\quad 0.1 \cdot -8 + 0.8 \cdot 1 + 0.1 \cdot 5 \quad (\downarrow)\} \\
&= 1 + \mathsf{max}\{5.1\,(\leftarrow), 7.7\,(\uparrow), \\
&\quad -5.3\,(\rightarrow), 0.5\,(\downarrow)\} \\
&= 1 + 7.7 \\
&= 8.7
\end{aligned}$$

Value Iteration Example

- In our example

0.812	0.868	0.918	+1
0.762		0.66	-1
0.705	0.655	0.611	0.388

(1,1)



- States far from the goal first accumulate negative rewards until a path is found to the goal

Convergence

- The condition $close\text{-}enough(U, U')$ in the algorithm can be formulated by

$$RMS = \frac{1}{|S|}\sqrt{\sum_s (U(s) - U'(s))^2}$$

$$RMS(U, U') < \epsilon$$

- Different ways to detect convergence:
  - RMS error: root mean square error
  - Max error: $\| U - U' \| = \max_s |U(s) - U'(s)|$
  - Policy loss

Convergence Example

- What the agent cares about is **policy loss**: How well a policy based on $U_i(s)$ performs
- Policy loss converges much faster (because of the argmax)

Value Iteration

- Value Iteration finds the **optimal solution** to the Markov Decision Problem!
- **Converges** to the **unique solution** of the Bellman equation system for $\gamma < 1$
- Initial values for U' are arbitrary
- Proof involves the concept of *contraction*.
  $\| B\,U_i - B\,U_i' \| \ \leq \ \gamma \, \| U_i - U_i' \|$ with B being the Bellman operator (see textbook)
- VI propagates information through the state space by means of **local updates**

Optimal Policy

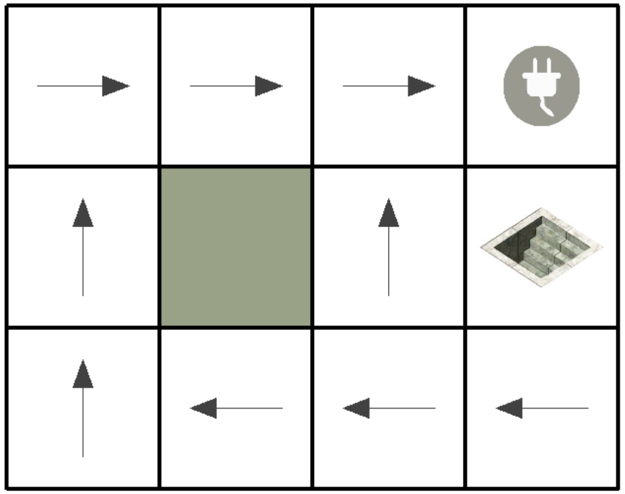
- How to finally compute the **optimal policy**? Can be easily extracted along the way by

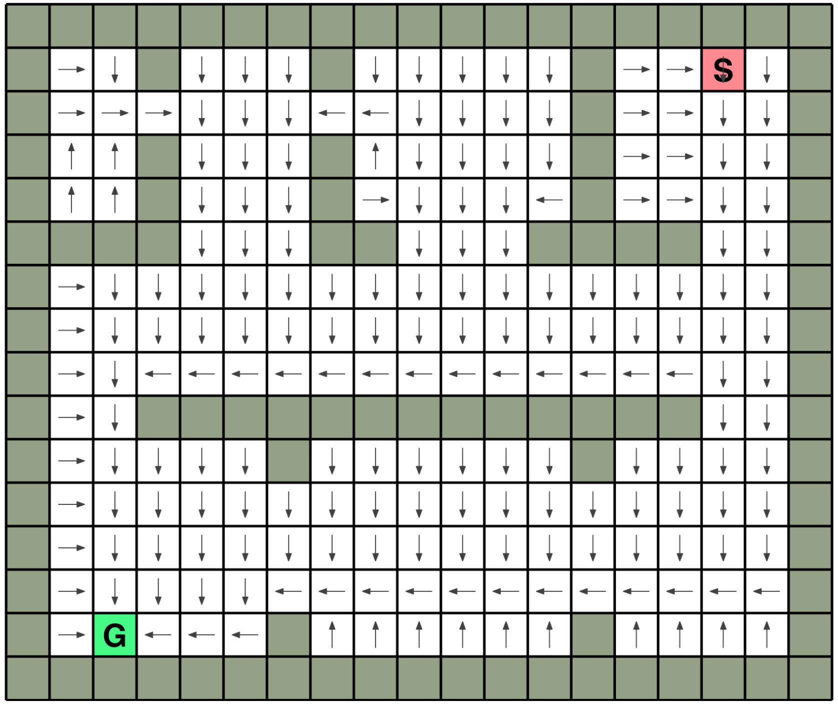
$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} T(s,a,s')\, U(s')$$

- **Note:** $U(s)$ and $R(s)$ are quite different quantities. $R(s)$ is the **short-term** reward for being in s, whereas $U(s)$ is the **long-term** reward **from** s **onwards**

Optimal Policy

- Examples





Summary

- MDPs describe an uncertain agent with a **stochastic transition model**
- The solution is called **policy** that is a mapping from **states to actions**
- Value Iteration is a instance of dynamic programming, converges for lower-than-one discounts or finite horizons
- A policy allows to implement a **feedback control strategy**, the robot can never become lost anymore

What's missing...?

- Good solutions to **jointly** plan the **path under local constraints** that overcome the decoupling of global and local planning
- Good solutions to implement **feasible feedback** control strategies
- Problem: the **curse of dimensionality**
- AI/planning people and control theory people need to **talk more**
- Hence, the robot motion planning problem is not fully solved yet, but **good** solutions for many **practical** problems exist