Introduction to Mobile Robotics

Path and Motion Planning

Wolfram Burgard, Michael Krawez

UTN

Motion Planning

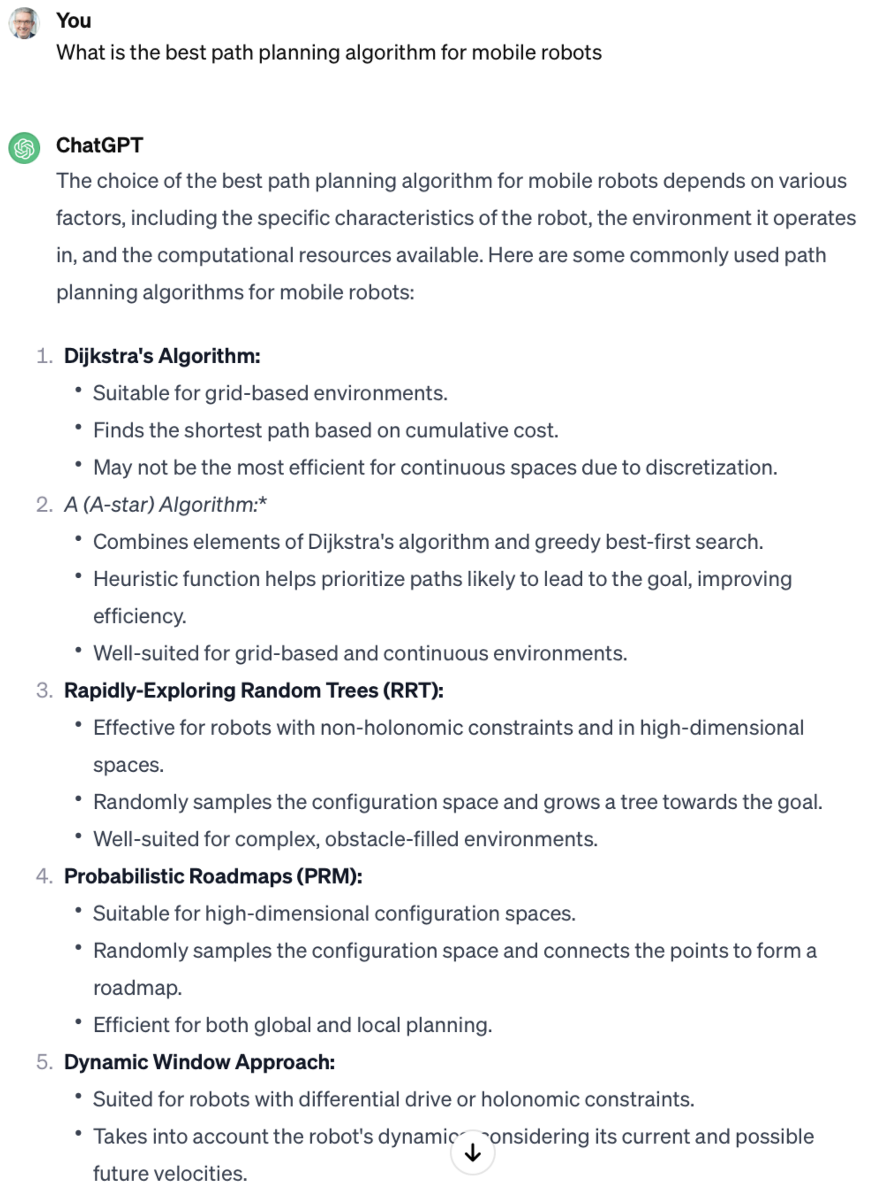
Latombe (1991):

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

Goals:

- Collision-free trajectories
- The robot should reach the goal location "as quickly" as possible

Best Path Planning Algorithms for Mobile Robots



You
What is the best path planning algorithm for mobile robots

ChatGPT
The choice of the best path planning algorithm for mobile robots depends on various factors, including the specific characteristics of the robot, the environment it operates in, and the computational resources available. Here are some commonly used path planning algorithms for mobile robots:

1. **Dijkstra's Algorithm:**
   - Suitable for grid-based environments.
   - Finds the shortest path based on cumulative cost.
   - May not be the most efficient for continuous spaces due to discretization.
2. _A (A-star) Algorithm:*_
   - Combines elements of Dijkstra's algorithm and greedy best-first search.
   - Heuristic function helps prioritize paths likely to lead to the goal, improving efficiency.
   - Well-suited for grid-based and continuous environments.
3. **Rapidly-Exploring Random Trees (RRT):**
   - Effective for robots with non-holonomic constraints and in high-dimensional spaces.
   - Randomly samples the configuration space and grows a tree towards the goal.
   - Well-suited for complex, obstacle-filled environments.
4. **Probabilistic Roadmaps (PRM):**
   - Suitable for high-dimensional configuration spaces.
   - Randomly samples the configuration space and connects the points to form a roadmap.
   - Efficient for both global and local planning.
5. **Dynamic Window Approach:**
   - Suited for robots with differential drive or holonomic constraints.
   - Takes into account the robot's dynamic onsidering its current and possible future velocities.

… in Dynamic Environments

- How to react to unforeseen obstacles?
  - efficiency
  - reliability
- Dynamic Window Approaches
  [Simmons, 96], [Fox et al., 97], [Brock & Khatib, 99]
- Grid-map-based planning
  [Konolige, 00]
- Nearness-Diagram-Navigation
  [Minguez at al., 2001, 2002]
- Vector-Field-Histogram+
  [Ulrich & Borenstein, 98]
- A*, D*, D* Lite, ARA*, …
- Many more (also more recent publications) in the context of robotics and self-driving cars

Challenges

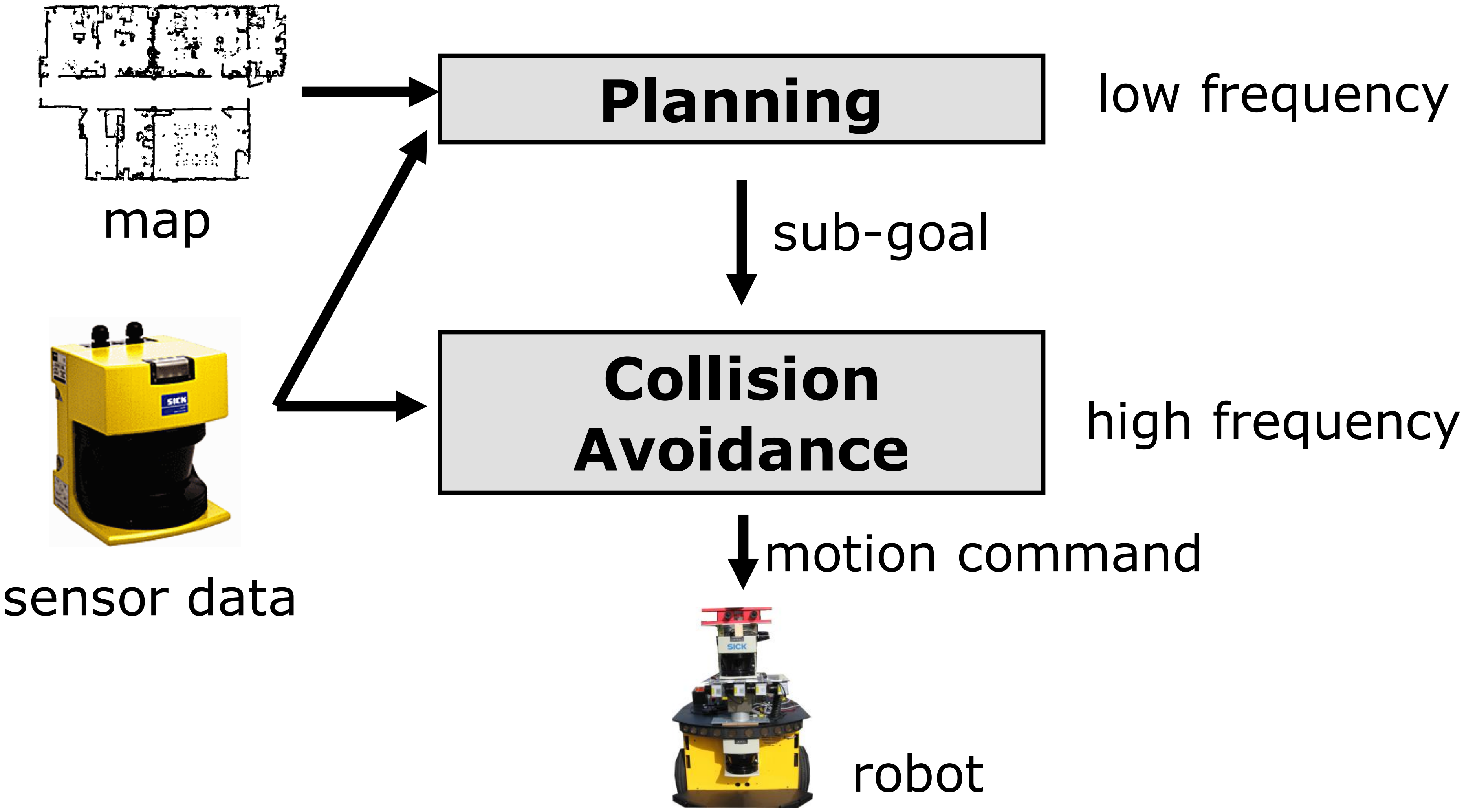
- Calculate the optimal path taking potential uncertainties in the actions (and the state) into account

- Quickly generate actions in the case of unforeseen objects

Further Challenges

- Discrete and continuous spaces
- How do you communicate your intention in mixed environments?
- How do we properly take risks into account?
- How do we perform planning on resource-constrained systems/computers?

Classic Two-layered Architecture



map

sensor data

Planning

low frequency

sub-goal

Collision Avoidance

high frequency

motion command

robot

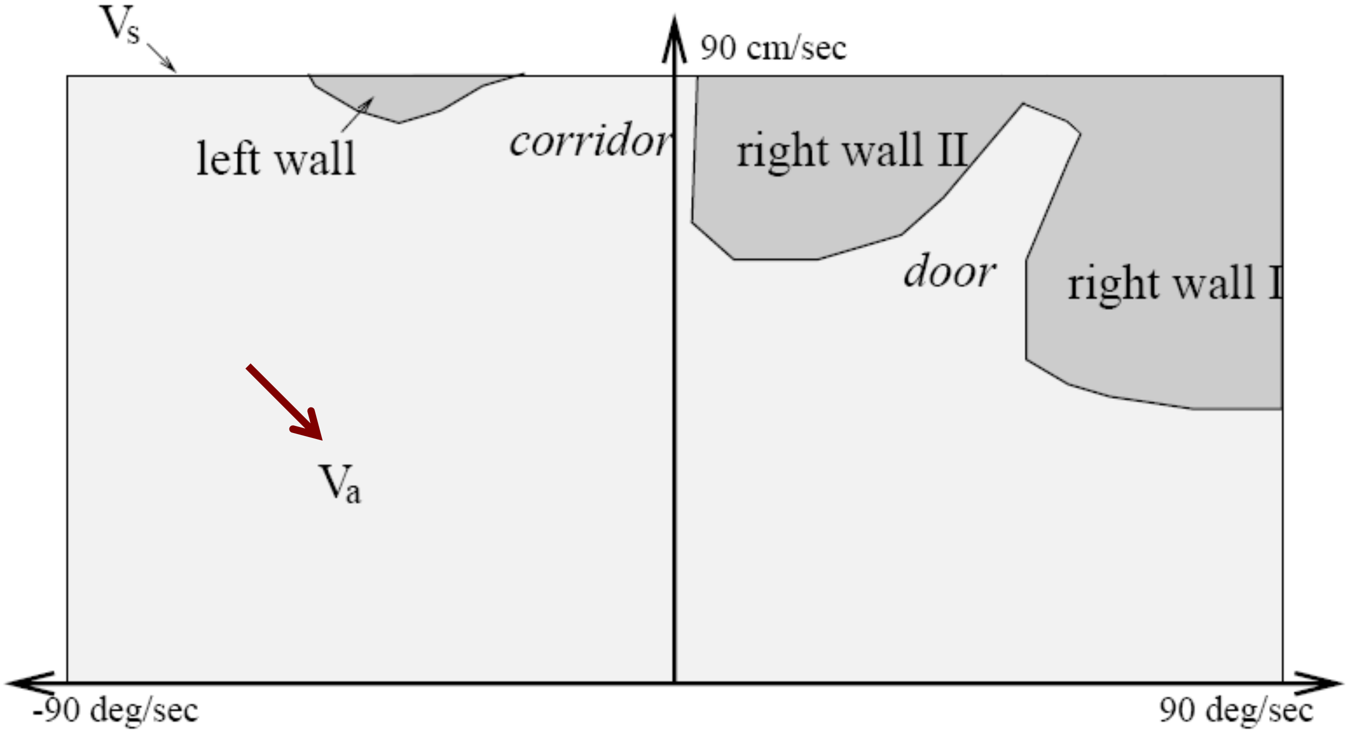
Dynamic-Window Approach to Collision Avoidance

- **Collision avoidance:** Determine collision-free trajectories using geometric operations
- Here: Robot moves on circular arcs
- Motion commands (v,ω)
- Which (v,ω) are admissible and reachable?

Admissible Velocities

- Speeds are admissible if the robot would be able to stop before reaching the obstacle

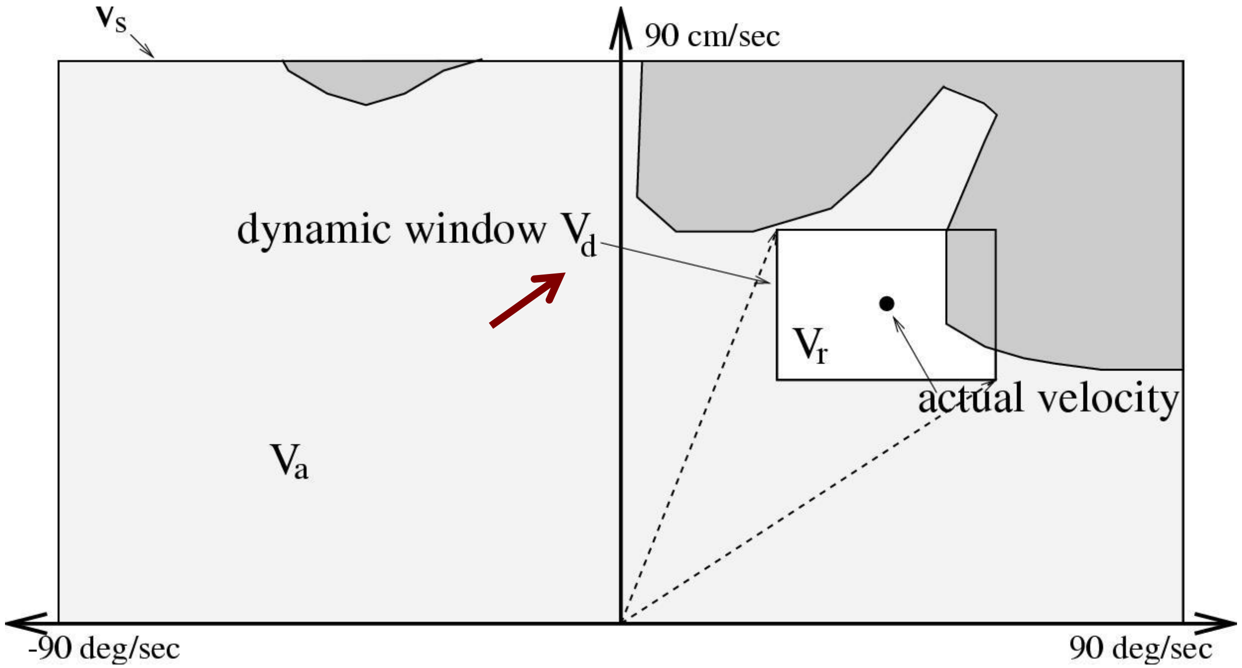
$$V_a = \{(v, \omega) \mid v \leq \sqrt{2\text{dist}(v,\omega)a_{trans}} \wedge \omega \leq \sqrt{2\text{dist}(v,\omega)a_{rot}}\}$$



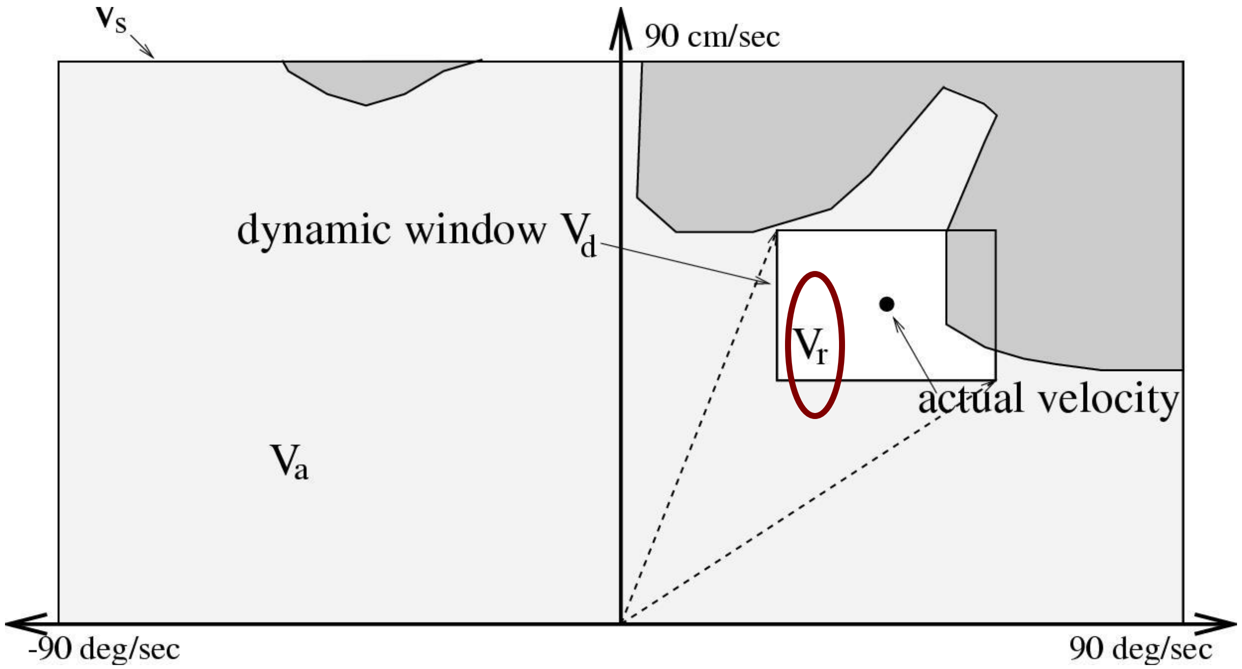
Reachable Velocities

- Speeds that are reachable by acceleration

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



DWA Search Space



- V_s = all possible speeds of the robot.
- V_a = obstacle-free area.
- V_d = speeds reachable within a certain time frame based on possible accelerations.

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

Dynamic-Window Approach

- How to choose <v,ω>?
- Steering commands are chosen by a heuristic navigation function.
- This function tries to minimize the travel-time by "**driving fast** into the **right direction**."

Dynamic-Window Approach

- Heuristic navigation function.
- Planning restricted to <x,y>-space.
- No planning in the velocity space.

Navigation Function: [Brock & Khatib, 99]

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

Dynamic-Window Approach

- Heuristic navigation function.
- Planning restricted to <x,y>-space.
- No planning in the velocity space.

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 <x,y>-space.
- No planning in the velocity space.

Navigation Function: [Brock & Khatib, 99]

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

Maximizes velocity

Considers cost to reach the goal

Dynamic-Window Approach

- Heuristic navigation function.
- Planning restricted to <x,y>-space.
- No planning in the velocity space.

Navigation Function: [Brock & Khatib, 99]

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

Maximizes velocity

Considers cost to reach the goal

Follows grid-based path computed by A*

Dynamic-Window Approach

- Heuristic navigation function.
- Planning restricted to <x,y>-space.
- No planning in the velocity space.

Navigation Function:

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

Maximizes velocity

Considers cost to reach the goal

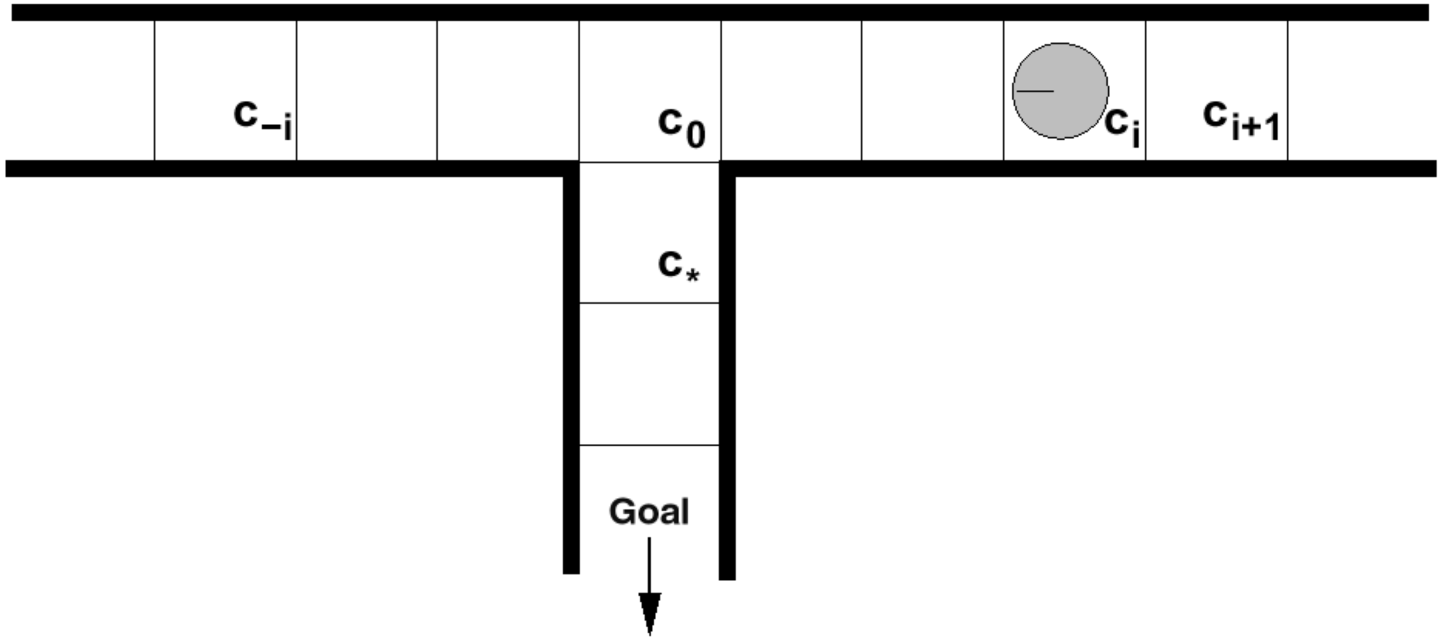
Follows grid-based path computed by A*

Goal nearness

Dynamic-Window Approaches

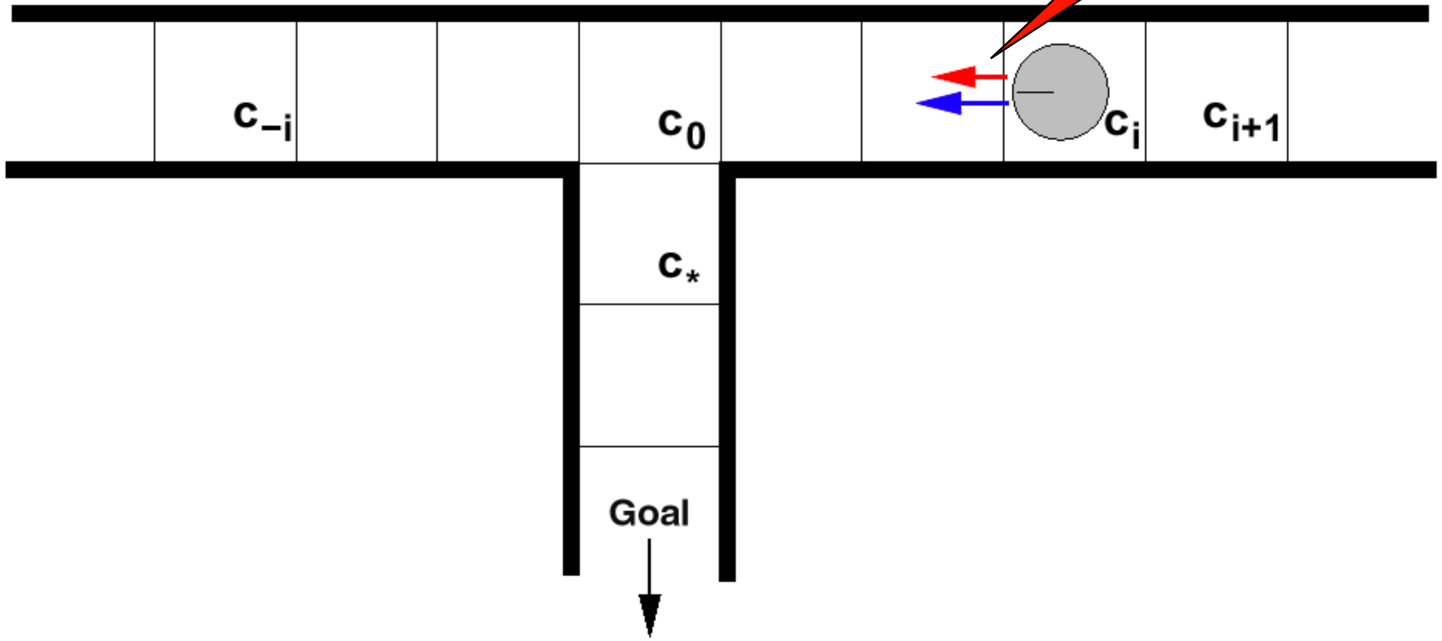
- React quickly
- Low computational requirements
- Guides a robot along a collision-free path
- Successfully used in a lot of real-world scenarios
- The resulting trajectories are sometimes sub-optimal
- Local minima might prevent the robot from reaching the goal location

Problems of DWAs



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

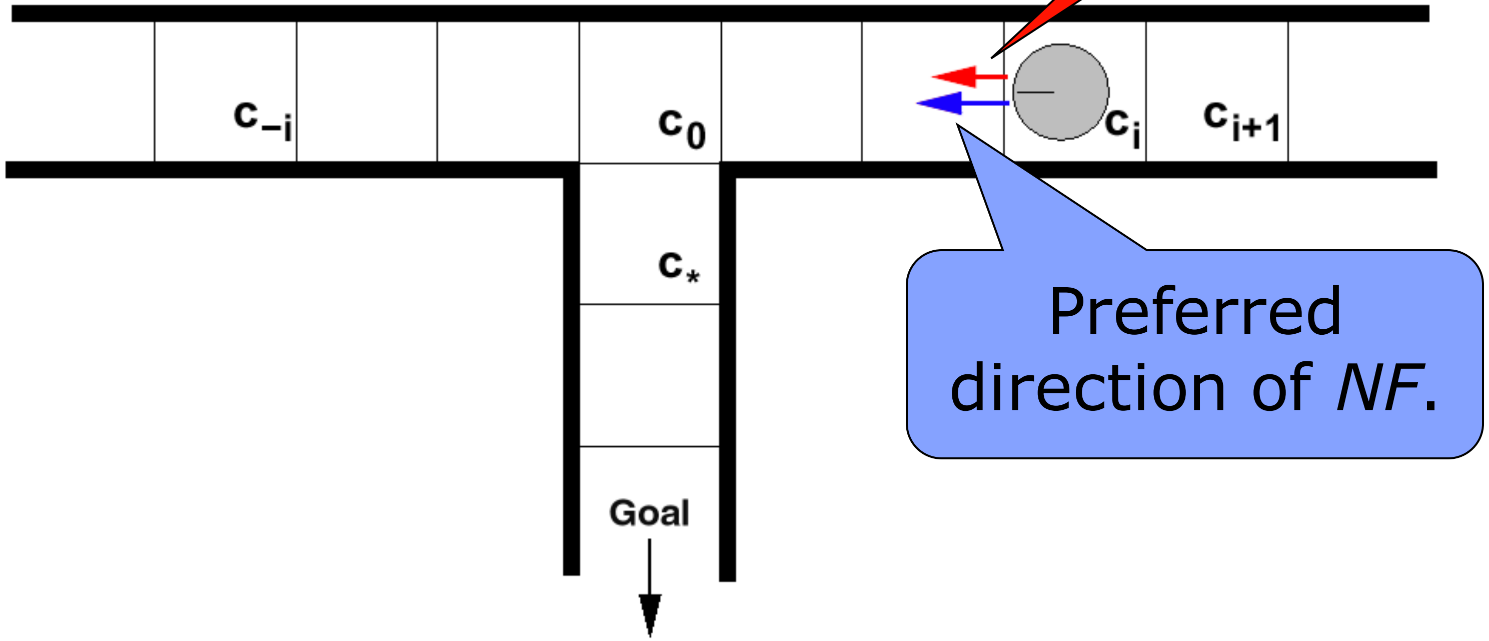
Problems of DWAs



Robot's velocity.

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

Problems of DWAs

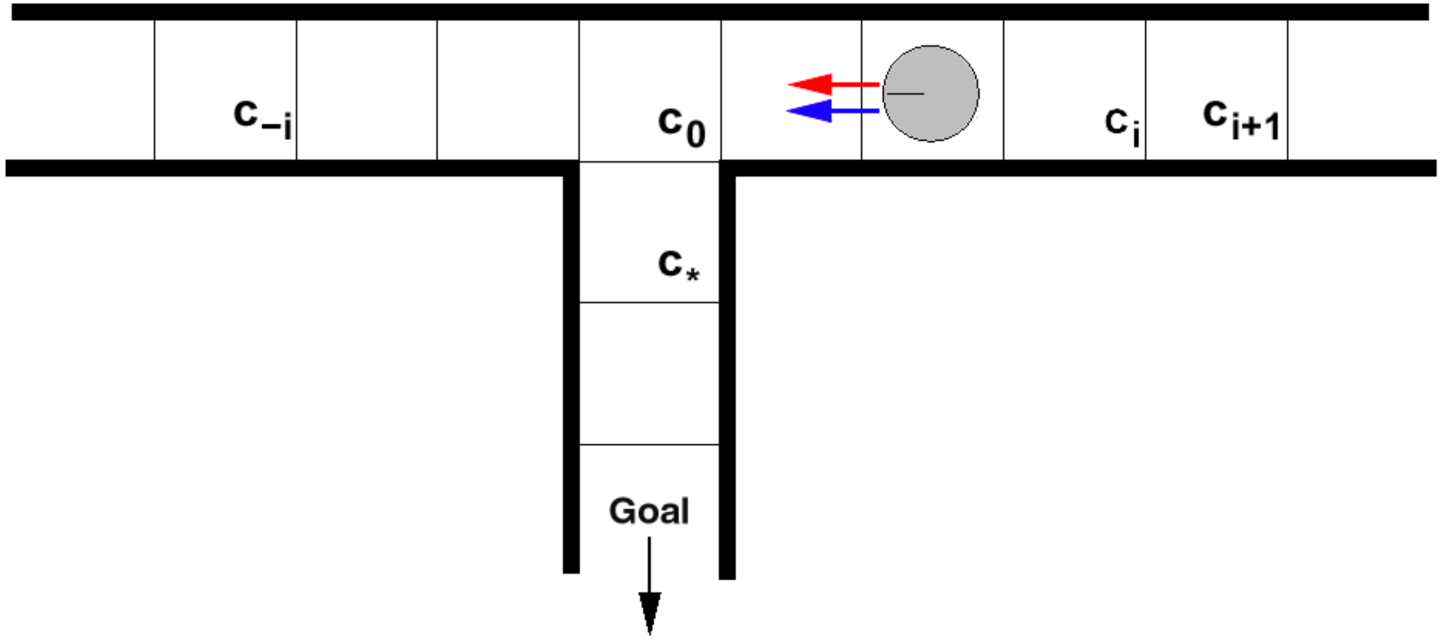


Robot's velocity.

Preferred direction of *NF*.

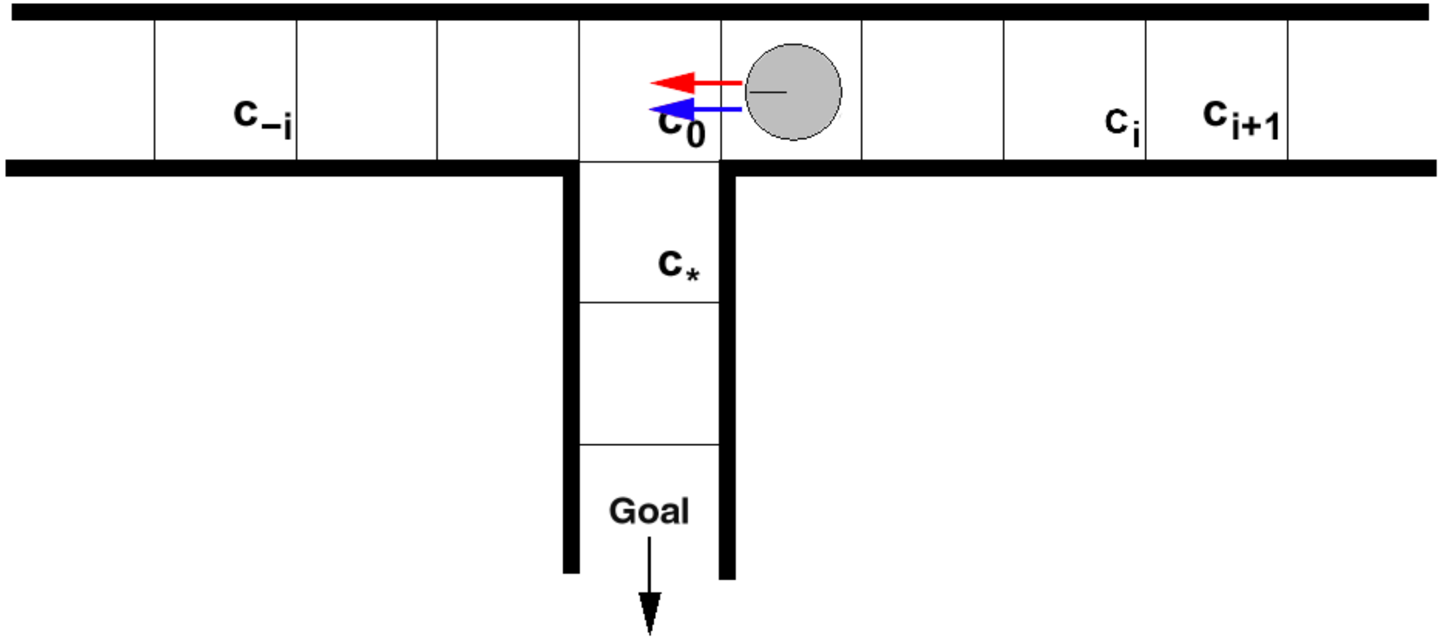
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



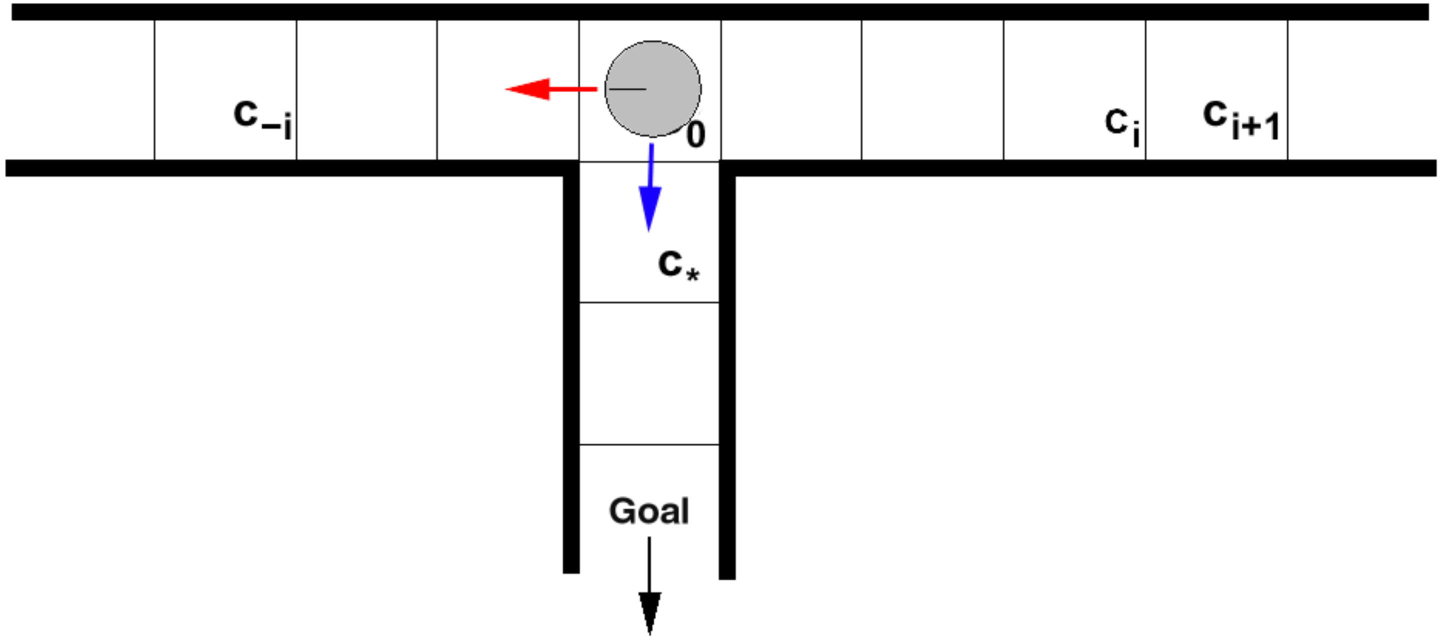
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



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

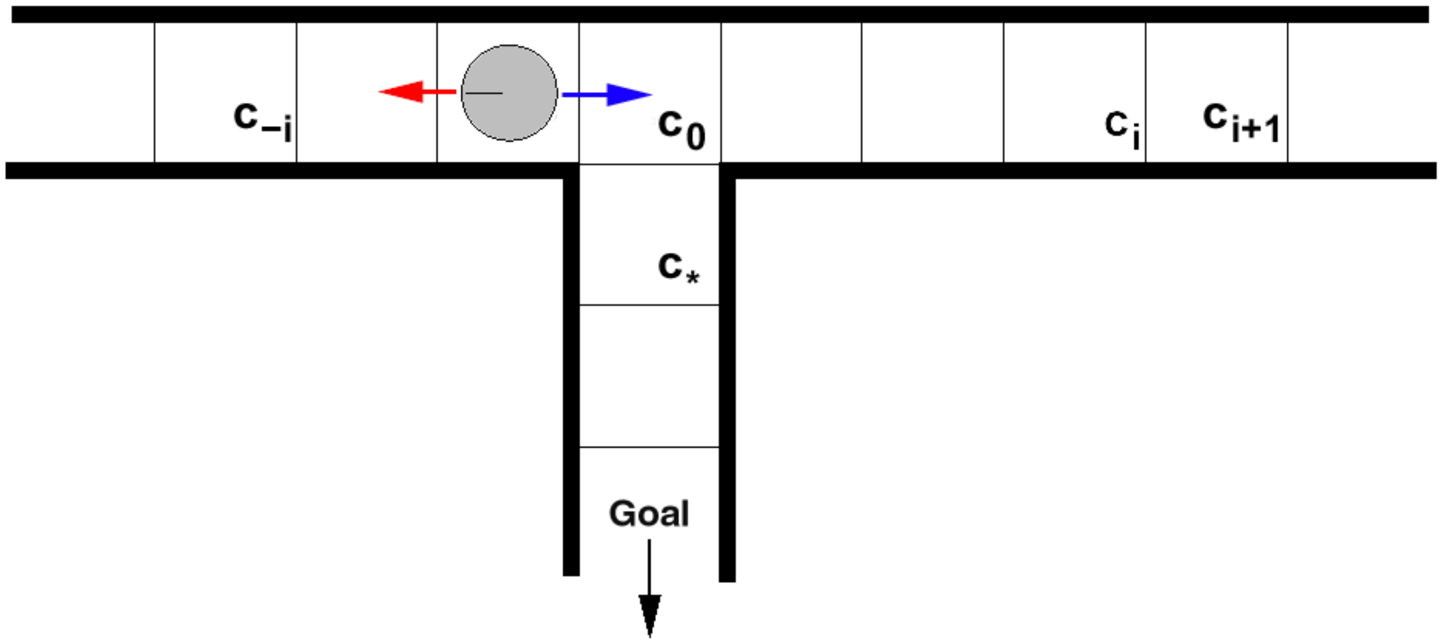
Problems of DWAs



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

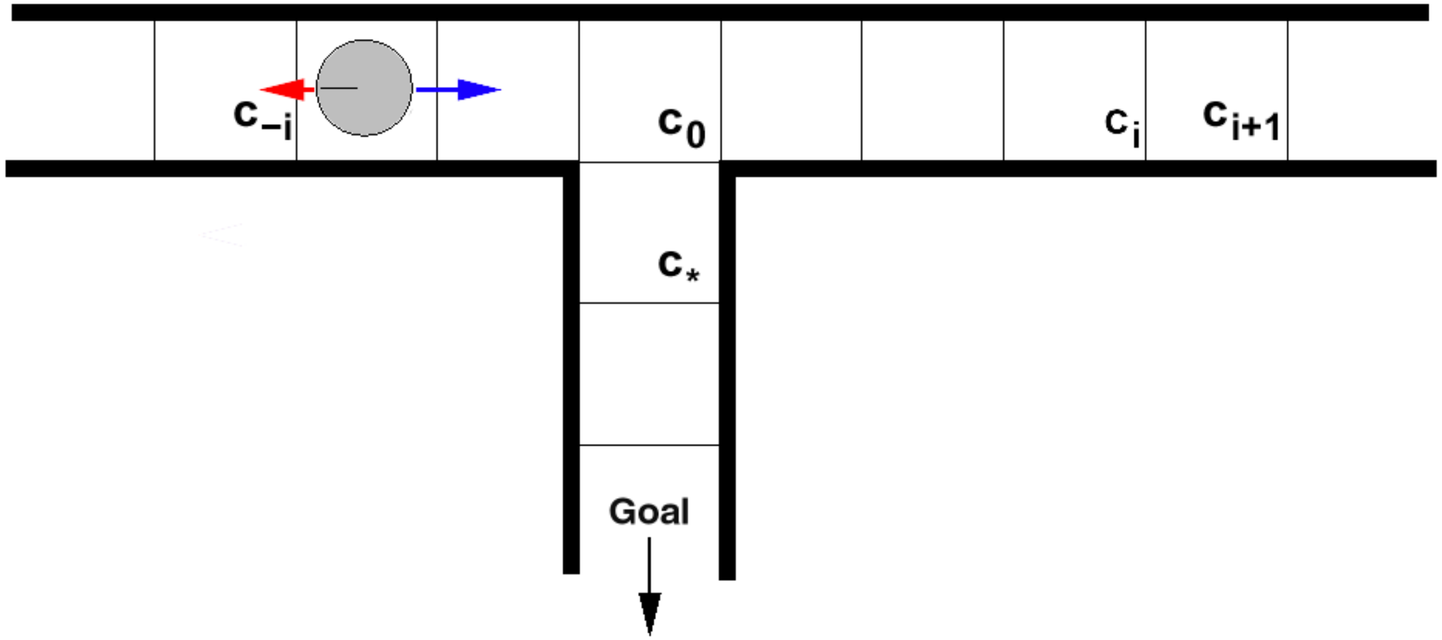
The robot drives too fast at c_0 to enter corridor facing south.

Problems of DWAs



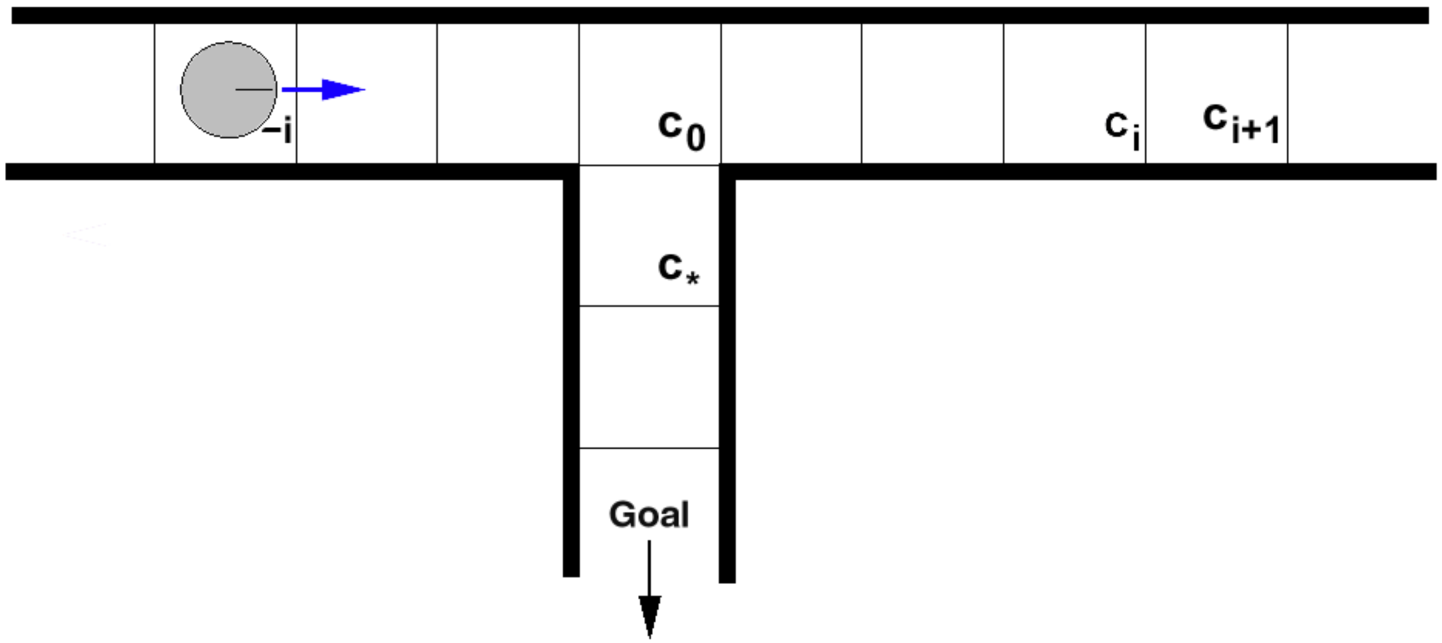
$$NF = \alpha \cdot vel + \beta \cdot nf + \gamma \cdot \Delta nf + \delta \cdot goal$$

Problems of DWAs



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

Problems of DWAs

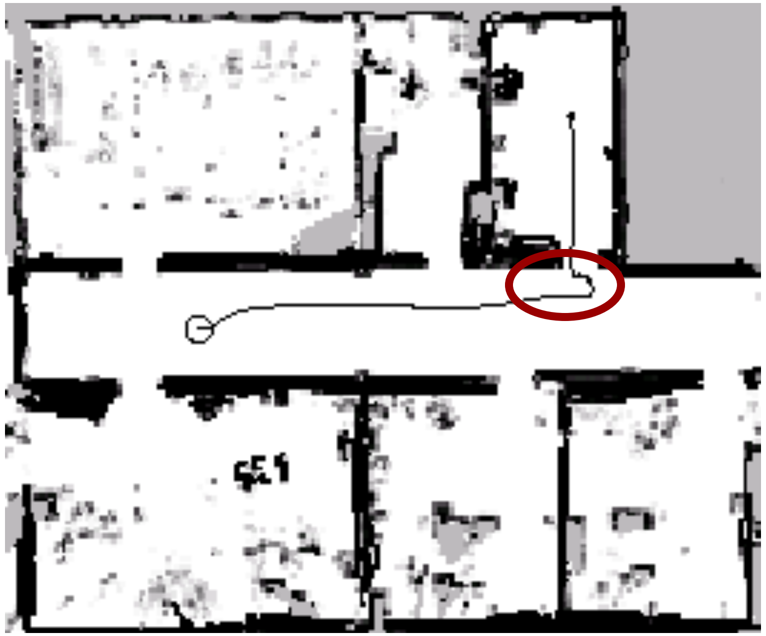


Same situation as in the beginning

➔ DWAs sometimes have problems reaching the goal

Problems of DWAs

- Typical problem in a real-world situation:



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

Motion Planning 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 representation of the **environment**

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

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

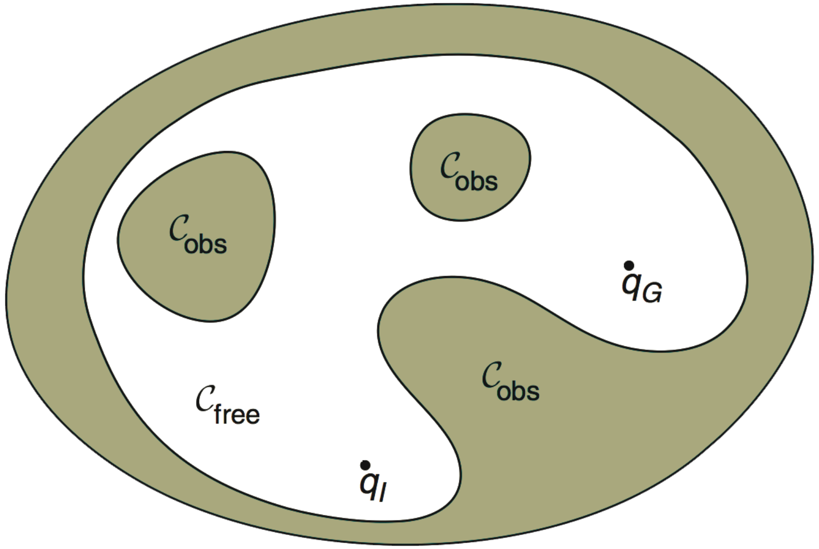
- **Free space** and **obstacle region**
- With $\mathcal{W} = \mathbb{R}^m$ being the workspace, $\mathcal{O} \in \mathcal{W}$ the set of obstacles, $\mathcal{A}(q)$ the robot in configuration $q \in \mathcal{C}$

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

- We further define

q_I : start configuration

q_G : goal configuration



Configuration Space

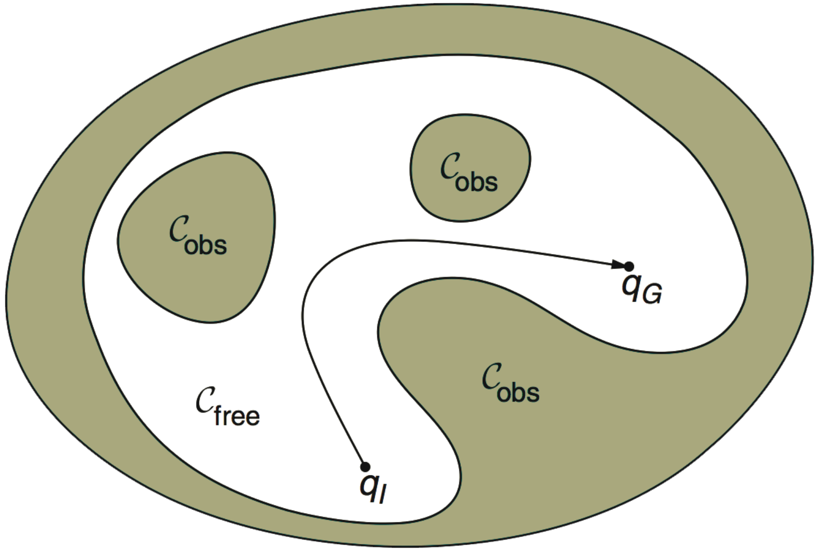
Then, motion planning amounts to

- 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!**


$\mathcal{C}_{\text{obs}}$
$\mathcal{C}_{\text{obs}}$
q_G
$\mathcal{C}_{\text{obs}}$
$\mathcal{C}_{\text{free}}$
q_I

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} explicitly 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 a solution

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*)

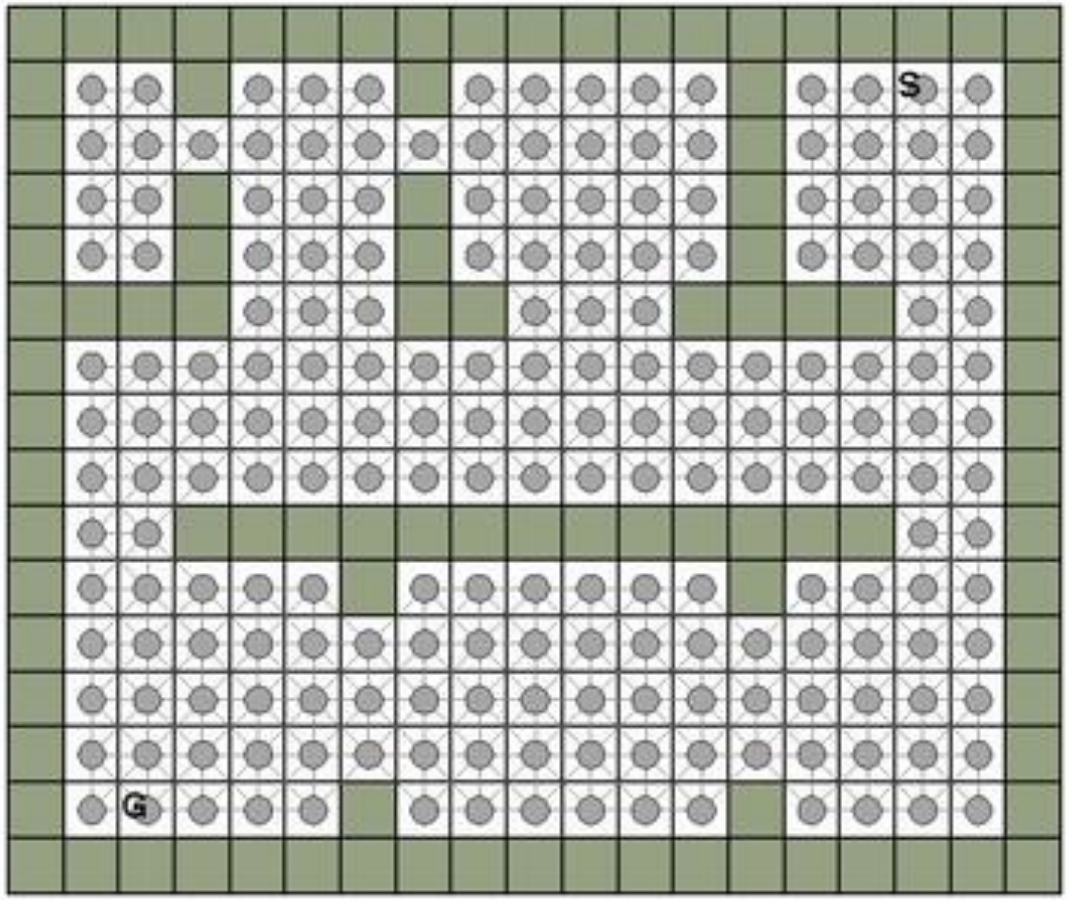
- **Informed search:** further information about the domain through **heuristics**
- Capability to say that a node is “more promising” than another node
- Example algorithms: greedy best-first search, **A***, many variants of A*, D*, etc.

Search

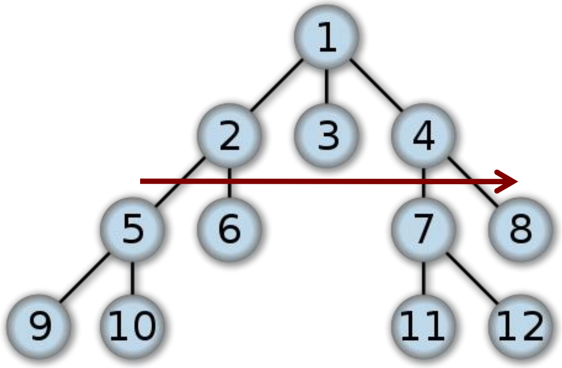
The performance of a search algorithm is often measured in four different ways:

- **Completeness:** does the algorithm find a 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?

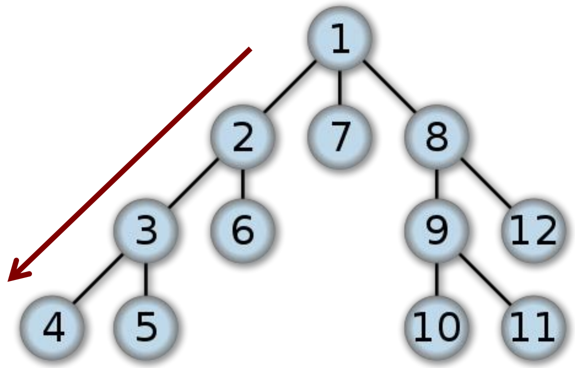
Discretized Configuration Space



Uninformed Search



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

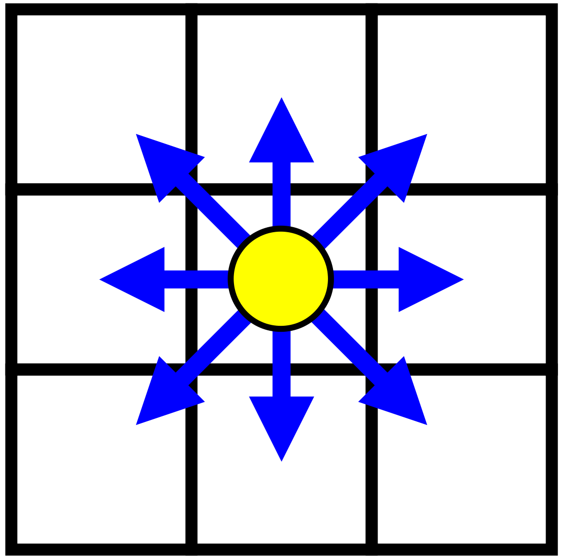


- **Depth-first**
  - Not complete in infinite spaces
  - Not optimal
  - Time: $O(b^m)$
  - Space: $O(bm)$ (can forget explored subtrees)

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

Informed Search: A^*

- What about using A^* to plan the path of a robot?
- Finds the shortest path
- Requires a graph structure
- A limited number of edges
- In robotics: planning on a 2d occupancy grid map



A^*: Minimize the Estimated Path Costs

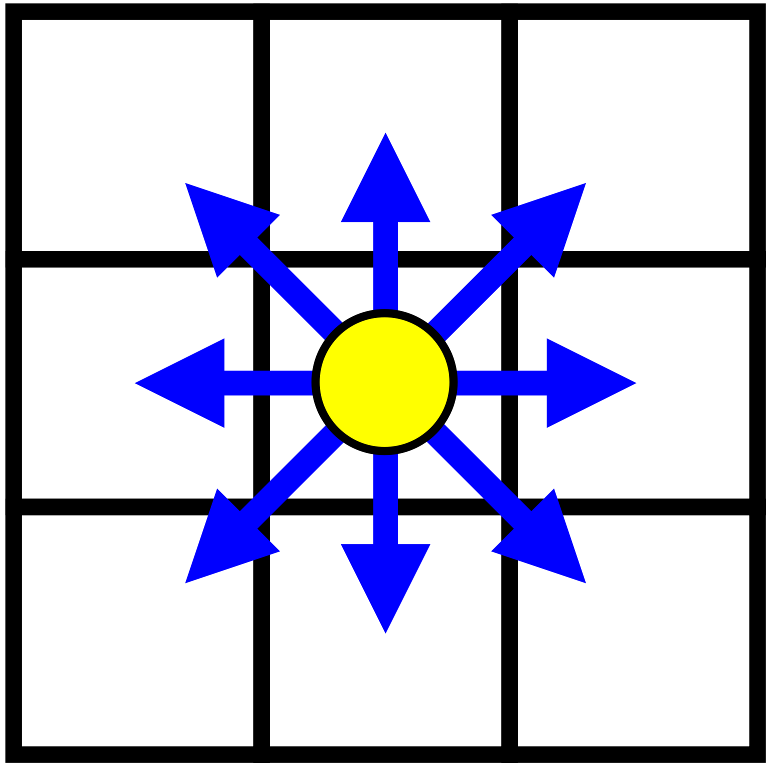
- *g(n)* = actual cost from the initial state to *n*.
- *h(n)* = estimated cost from *n* to the next goal.
- *f(n)* = *g(n)* + *h(n)*, the estimated cost of the cheapest solution through *n*.
- Let $h^*(n)$ be the actual cost of the optimal path from *n* to the next goal.
- *h* is admissible if the following holds for all *n*:

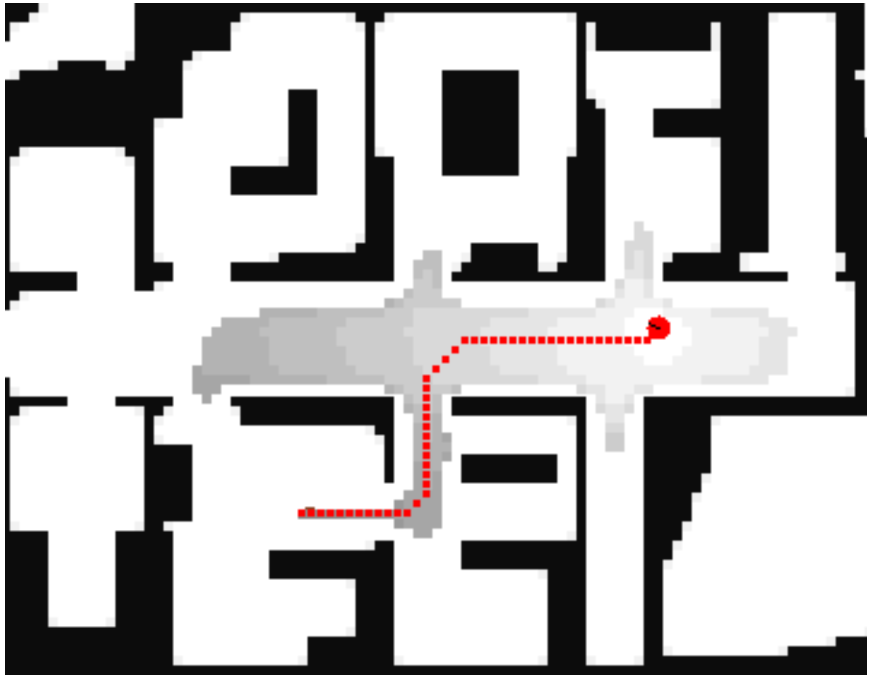
$$h(n) \leq h^*(n)$$

- If *h* is admissible, A^* yields the optimal solution.

Note: The straight-line distance is admissible in the Euclidean space.

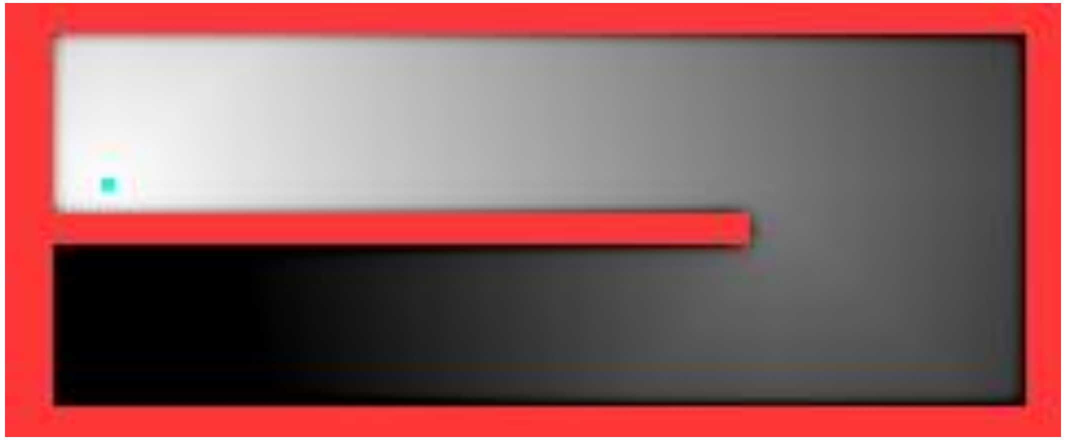
Example: Path Planning for Robots in a Grid-World





Deterministic Value Iteration

- To compute the shortest path from every state to one goal state, use (deterministic) value iteration.
- Very similar to Dijkstra’s Algorithm.
- Such a cost distribution is the optimal heuristic for A*.

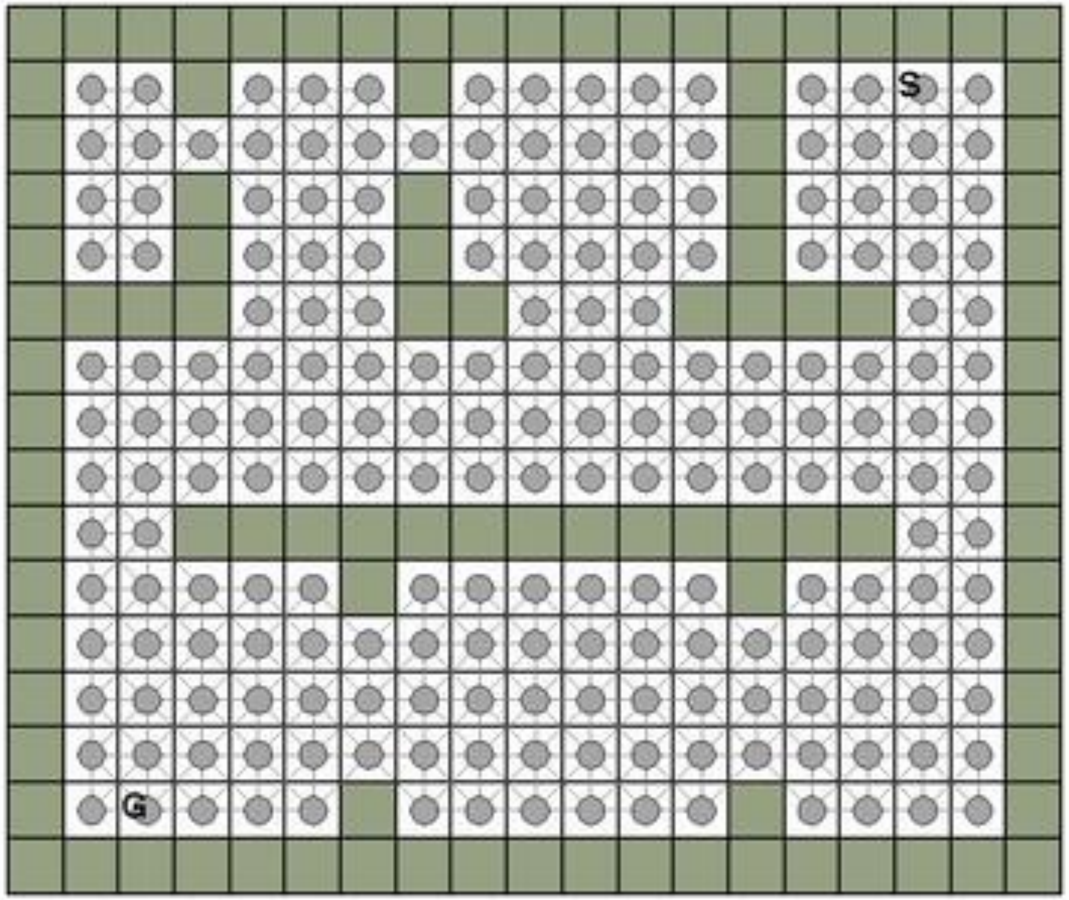


Typical Assumption in Robotics for A* Path Planning

1. The robot's position is assumed to be known
2. The path is computed using an occupancy grid
3. The correct motion commands are executed

Are 1. and 3. always true?

Discretized Configuration Space



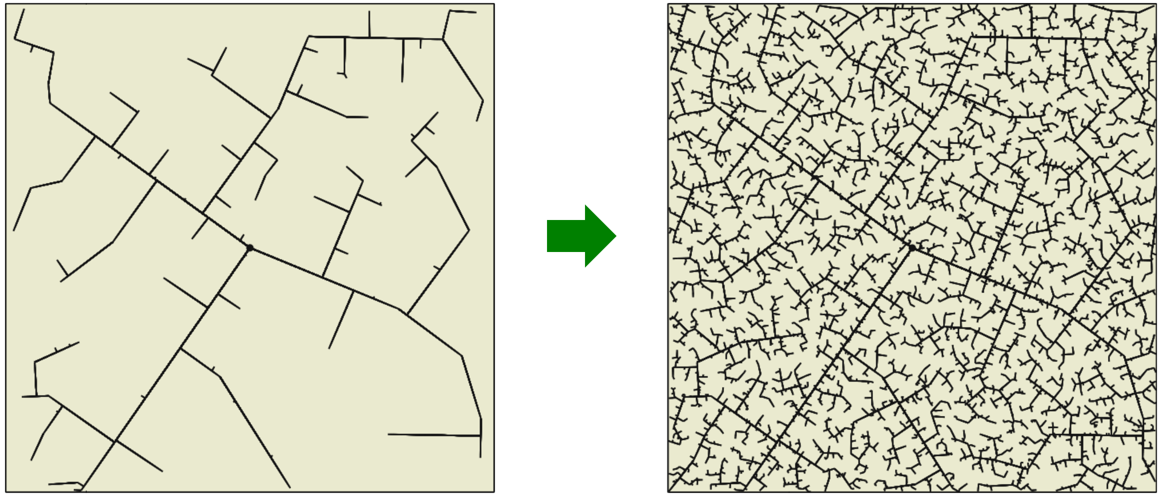
Problems

- What if the robot is (slightly) delocalized?
- Moving on the shortest path often guides the robot along a trajectory close to obstacles.
- Trajectory aligned to the grid structure.

Path Planning

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

2,345 iterations

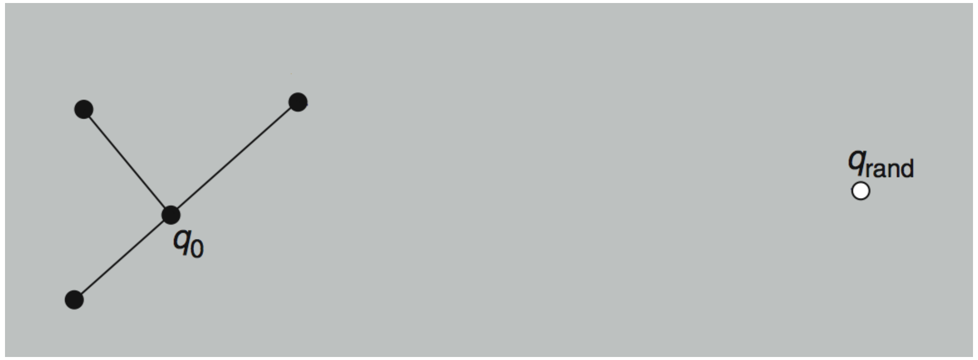
RRTs

The algorithm: Given C and q_0

Algorithm 1: RRT

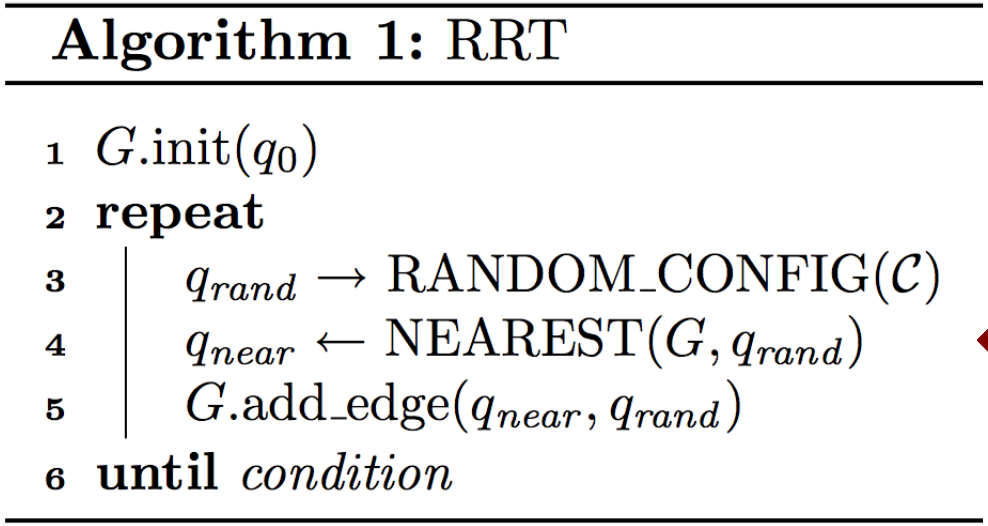
```
G.init($q_0$)
repeat
    $q_{rand} \rightarrow$ RANDOM_CONFIG($\mathcal{C}$)
    $q_{near} \leftarrow$ NEAREST($G, q_{rand}$)
    G.add_edge($q_{near}, q_{rand}$)
until condition
```

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



RRTs

- The algorithm



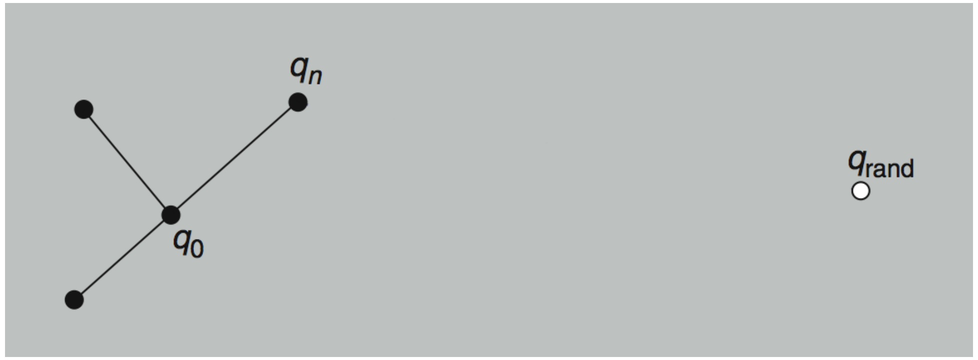
Algorithm 1: RRT

```
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
```

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

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

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

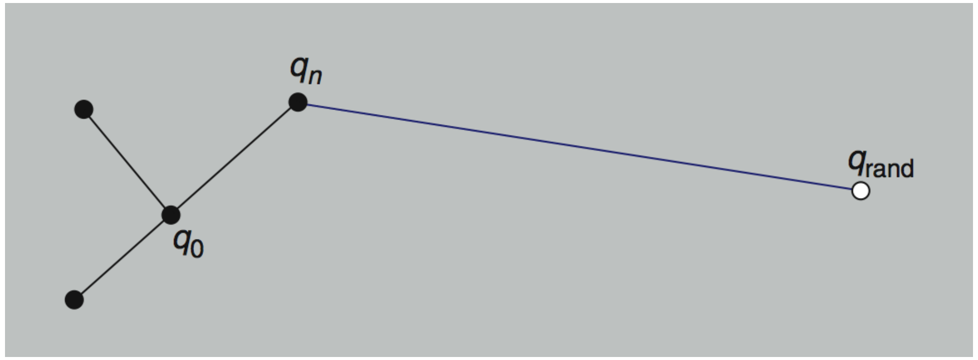


RRTs

- 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
```



Connect nearest point q_{near} with random point q_{rand} using a **local planner** (straight line in the simplest case)

No collision: add edge

RRTs

- The algorithm

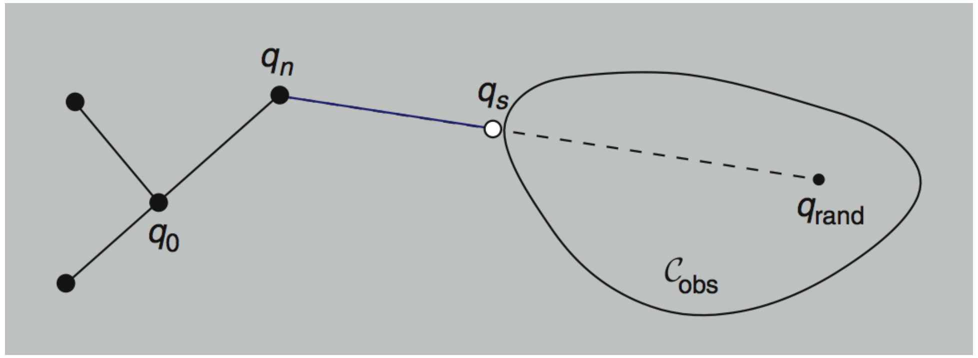
Algorithm 1: RRT

```
$G$.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
```

Connect nearest point q_{near} with random point q_{rand} using a **local planner** (straight line in the simplest case)

No collision: add edge

Collision: new vertex is q_s that is as close as possible to C_{obs}



RRTs

- How to perform path planning with RRTs?

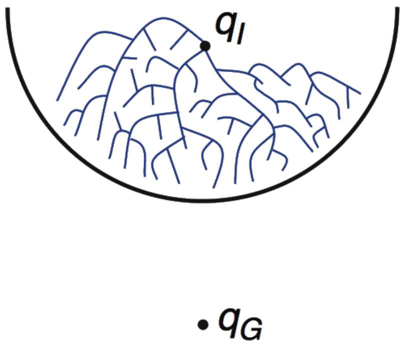
  1. Start RRT at q_I
  2. At every, say, 100th iteration, force $q_{rand} = q_G$
  3. If q_G is reached, problem is solved

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

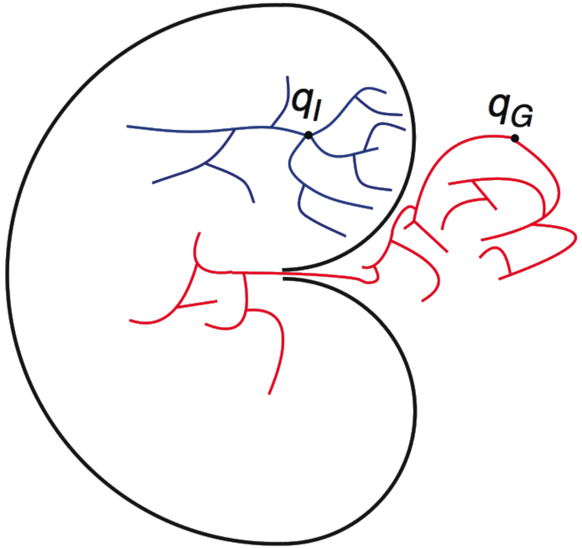
RRTs

- 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:

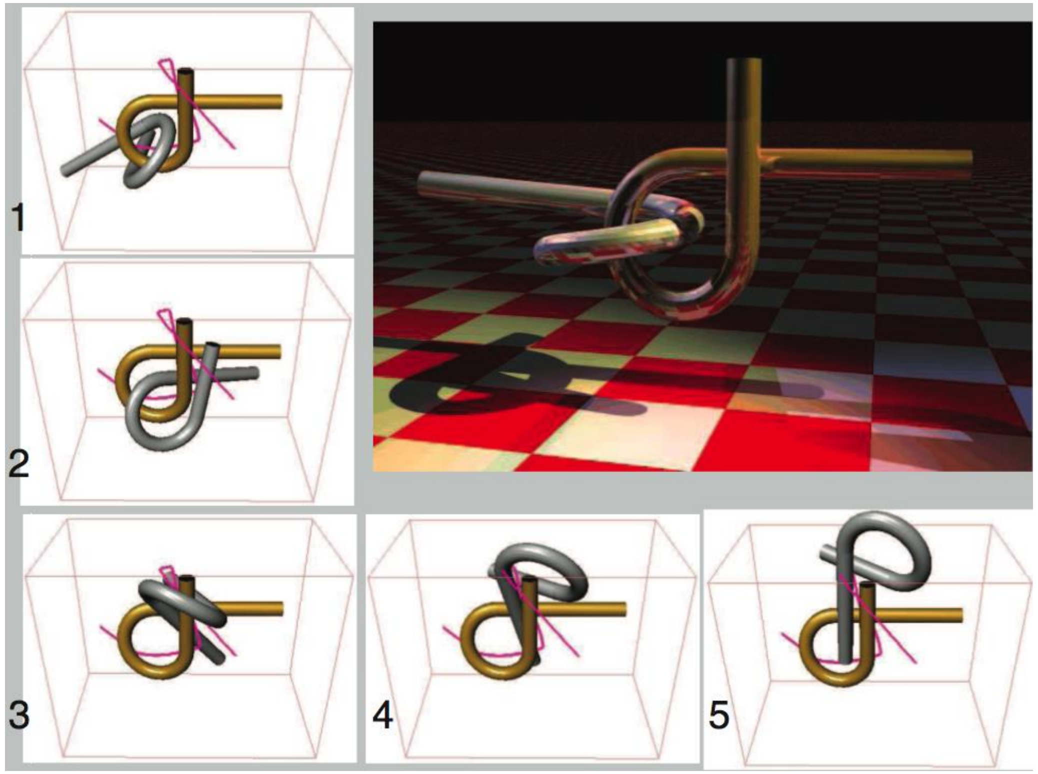


A bug trap:



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 sensitivity
  - Unknown rate of convergence



Alpha 1.0 puzzle
solved with bidirectional RRT

Road Map Planning

- 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}

- Several **planning techniques**
  - Visibility graphs
  - Voronoi diagrams
  - Exact cell decomposition
  - Approximate cell decomposition
  - Randomized road maps

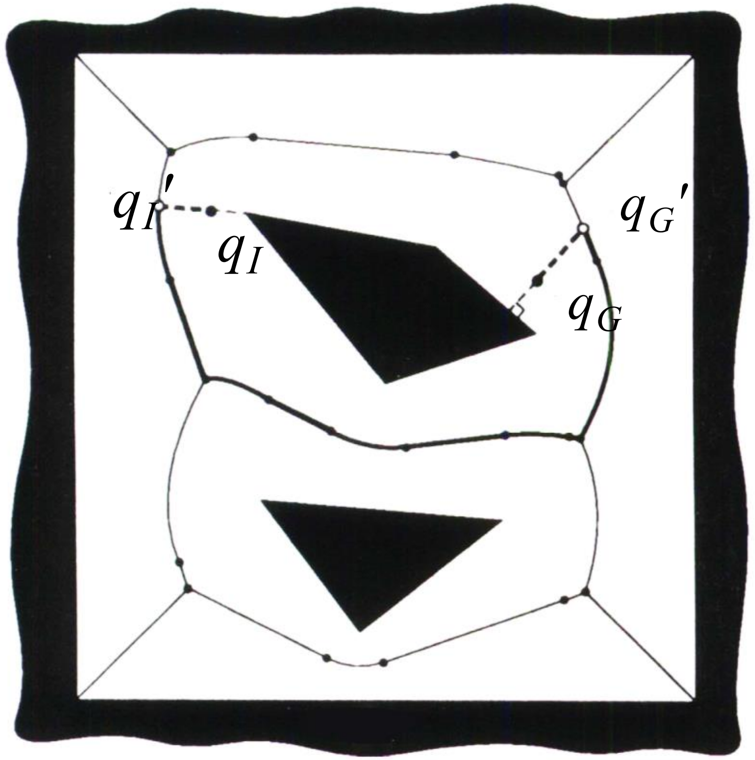
Road Map Planning

- 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}

- Several **planning techniques**
    - Visibility graphs
    - **Voronoi diagrams**
    - Exact cell decomposition
    - Approximate cell decomposition
    - **Randomized road maps**

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

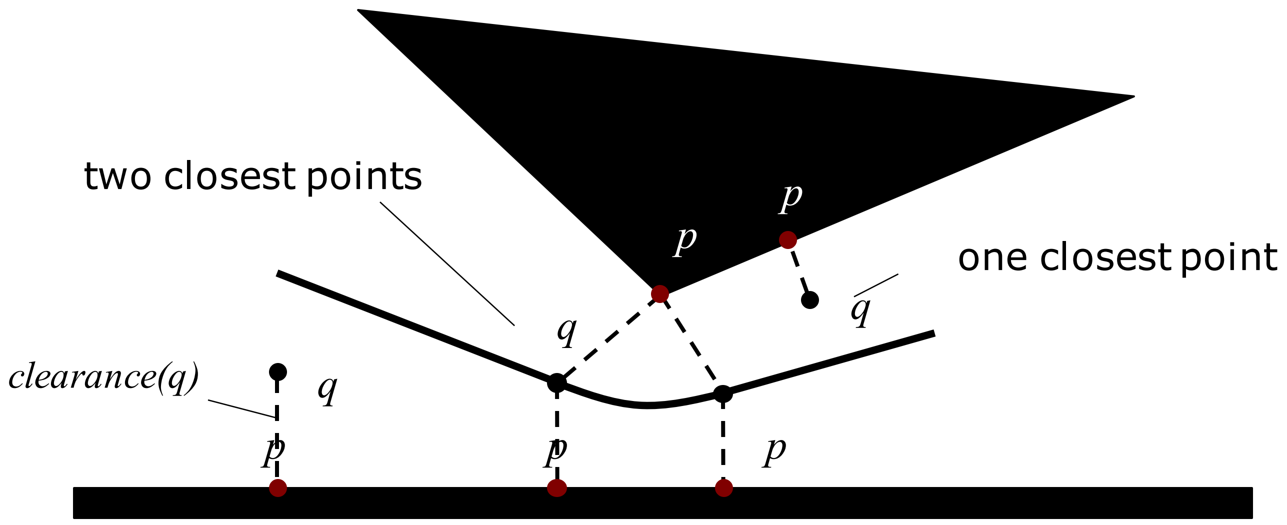
$$near(q) = \{p \in \beta \mid d(p, q) = clearance(q)\}$$

  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

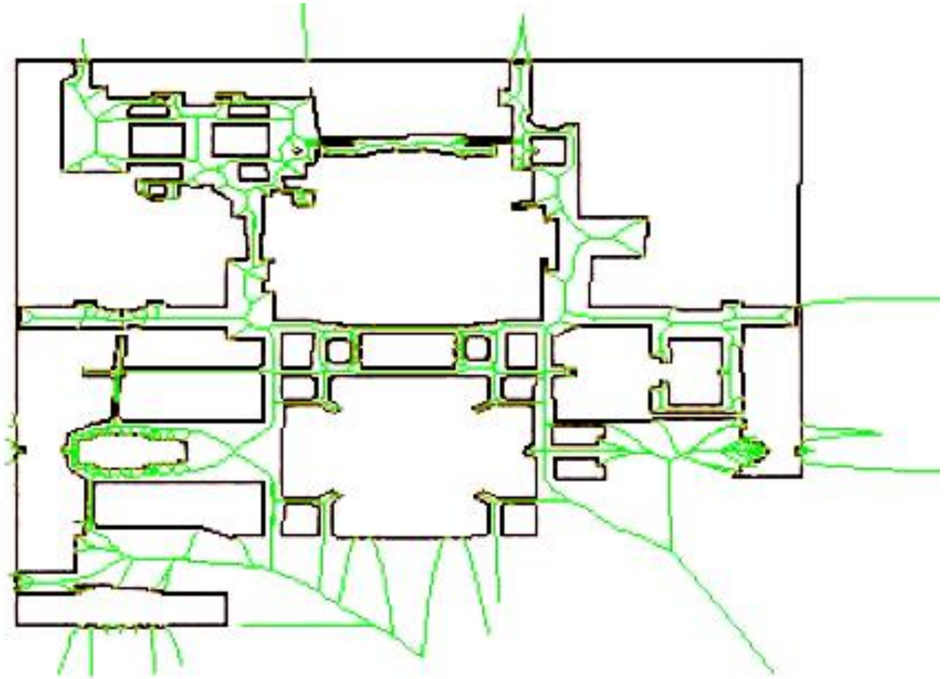
- **Geometrically:**



- 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-dimensional $\mathcal{C}$'s
  - **Pros:** maximizing clearance is a good idea for an uncertain robot
  - **Cons:** unnatural attraction to open space, suboptimal paths



- Needs extensions

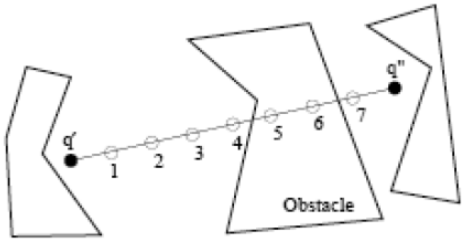
Randomized Road Maps

Also called *Probabilistic Road Maps*

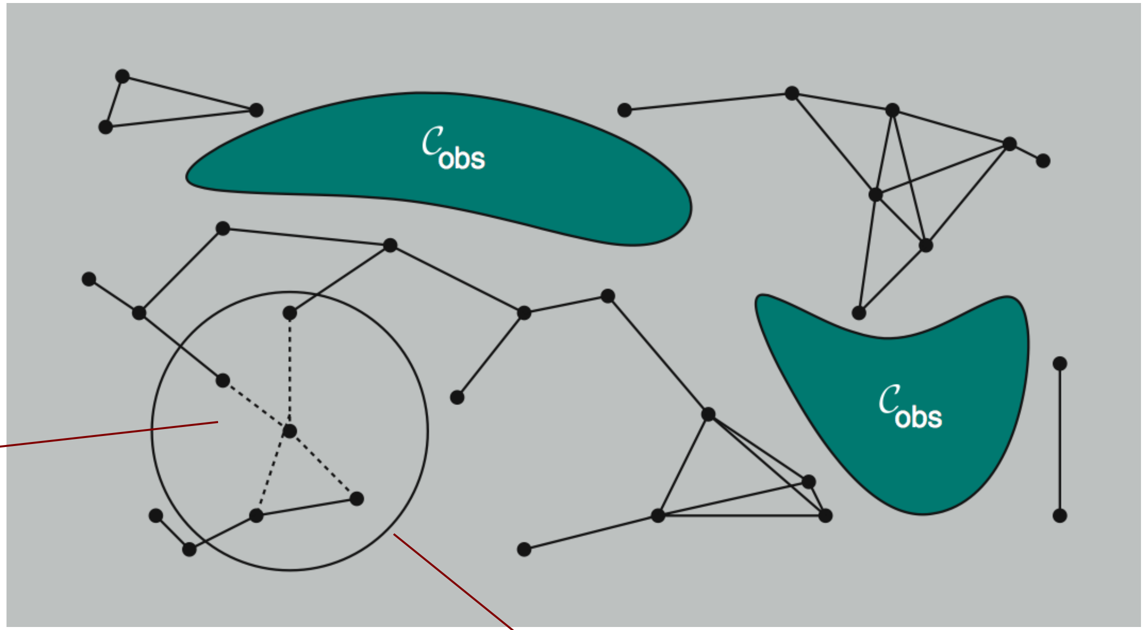
- **Idea:** Take random samples from C, declare them as vertices if in C_{free}, try to connect nearby vertices with local planner
- The **local planner** checks if line-of-sight is collision-free (powerful or simple methods)
- Options for *nearby:* **k-nearest neighbors** or all neighbors within **specified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

Randomized Road Maps

- Example



example local planner



specified radius

- What does “nearby” mean on a manifold?
- Defining a good metric on C is crucial

Randomized 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 more samples near poles, not uniform over $SO(3)$. Use quaternions!
- However, Randomized Road Maps are **powerful, popular** and **many extensions** exist: advanced sampling strategies (e.g., near obstacles), PRMs for deformable objects, closed-chain systems, etc.

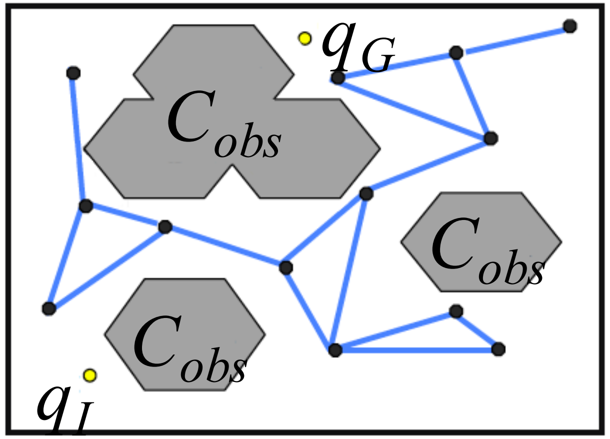
From Road Maps to Paths

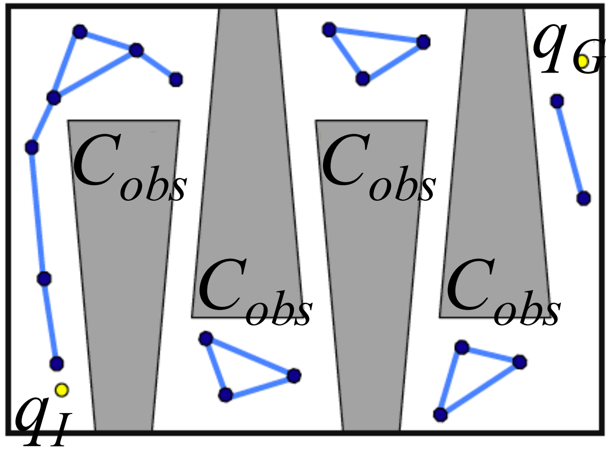
- All methods discussed so far **construct a road map** (without considering the query pair q_I and q_G)
- Once the investment is made, the **same road map** can be reused for **all** queries (provided world and robot do not change)

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

Randomized Road Maps

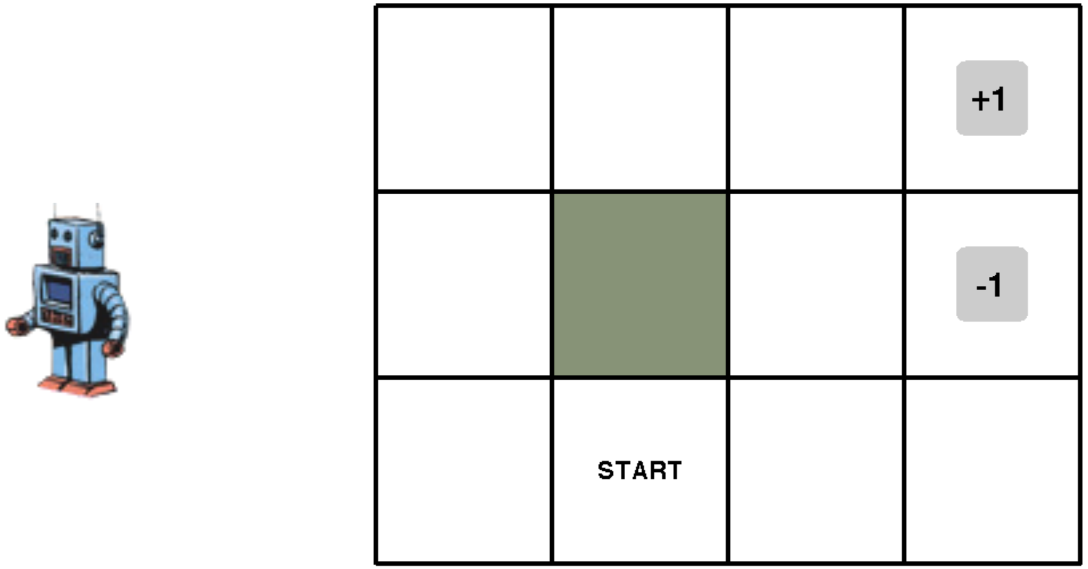
- **Pros:**
  - *Probabilistically complete*
  - Do not construct C-space
  - Apply easily to high dimensional C-spaces
  - Randomized road maps have solved previously unsolved problems
- **Cons:**
  - Do not work well for some problems, e.g., narrow passages
  - Not optimal, not complete





Markov Decision Process

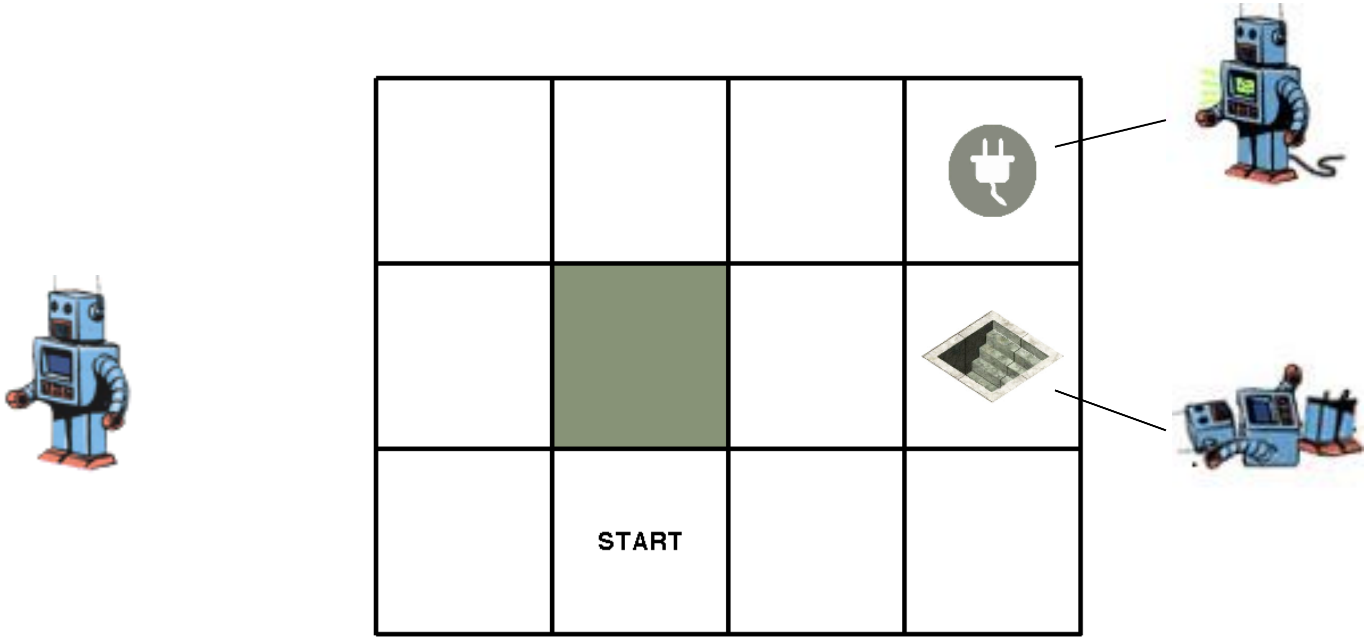
- Consider an agent acting in this environment



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

Markov Decision Process

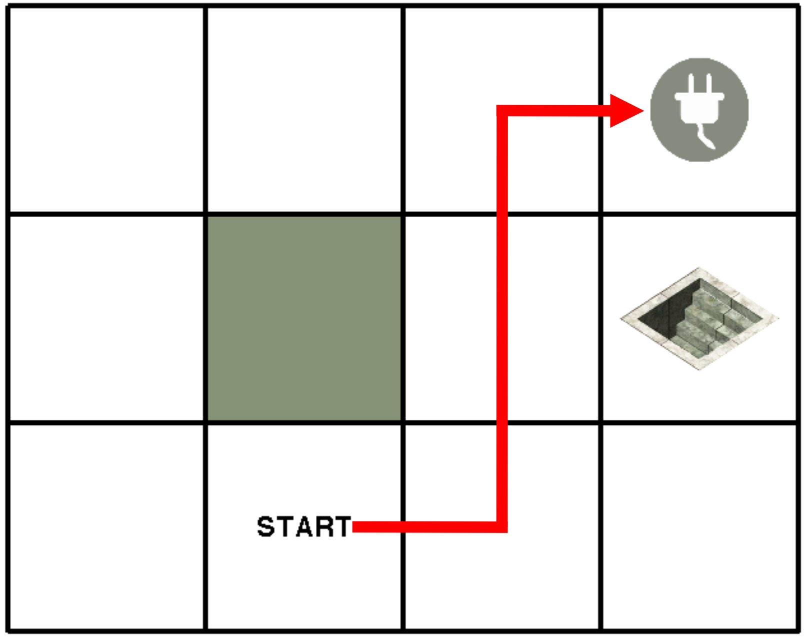
- Consider an agent acting in this environment



- 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?

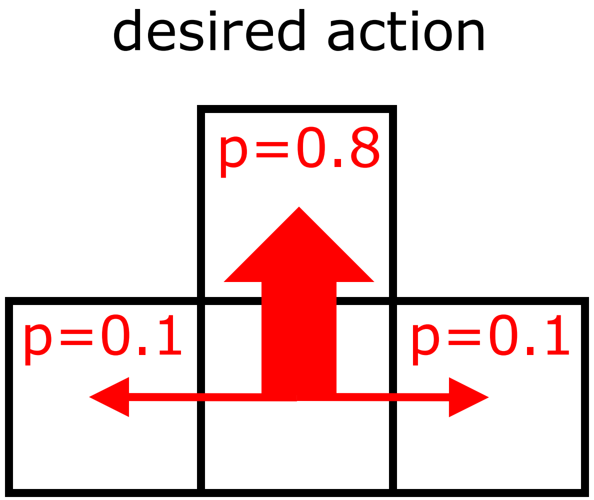
- Example: a mobile robot does not *exactly* perform a desired motion
- Example: human navigation

→ Uncertainty about performing actions!

MDP Example

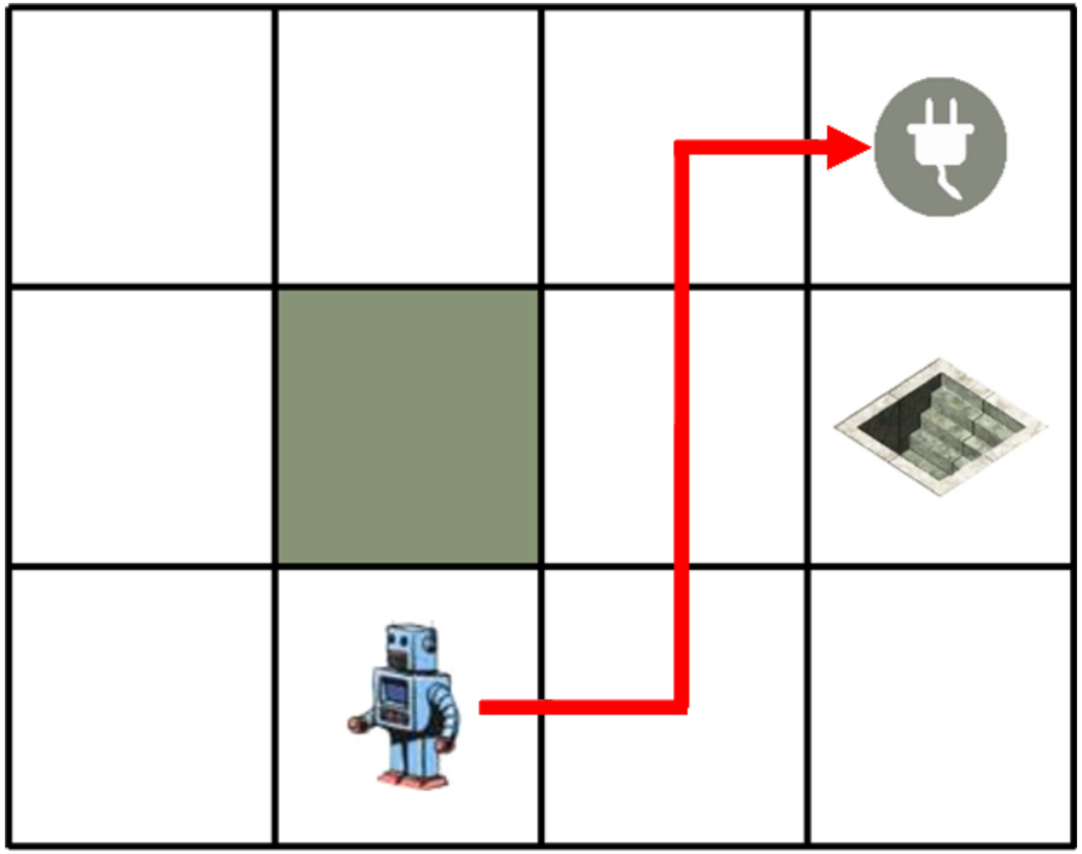
Consider the **non-deterministic transition model** (N / E / S / W):

- 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



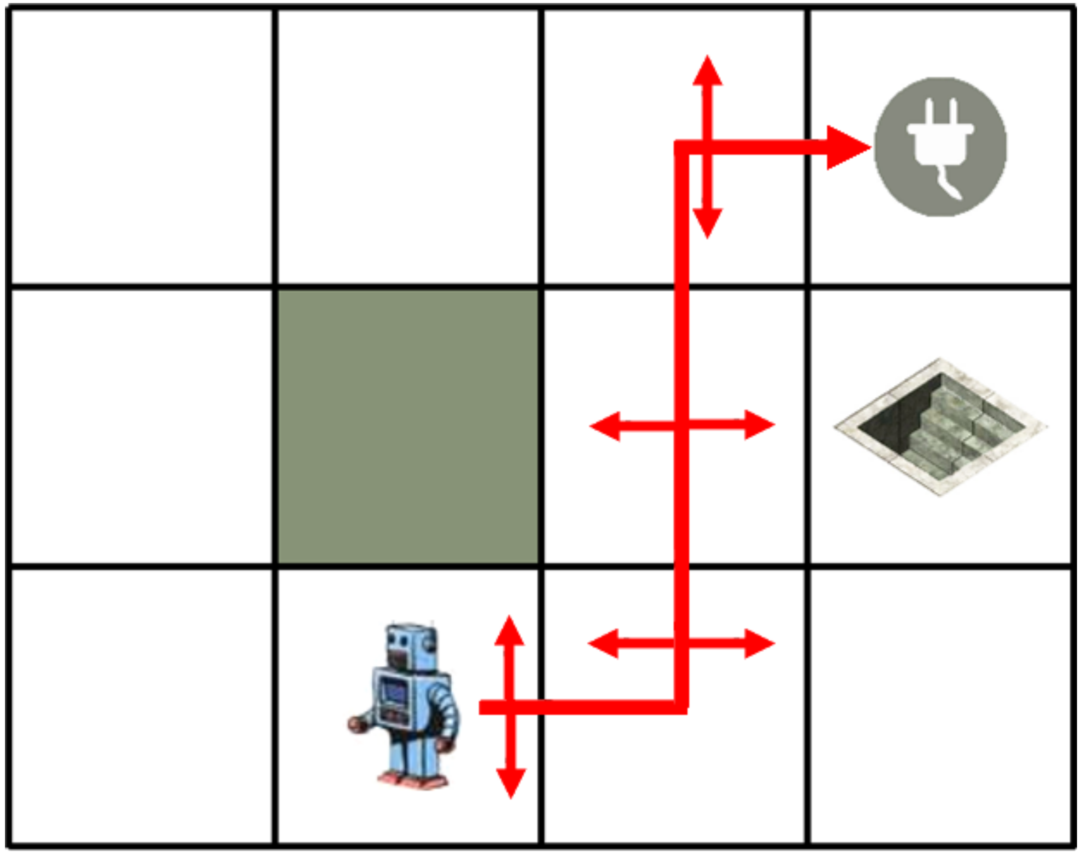
MDP Example

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



MDP Example

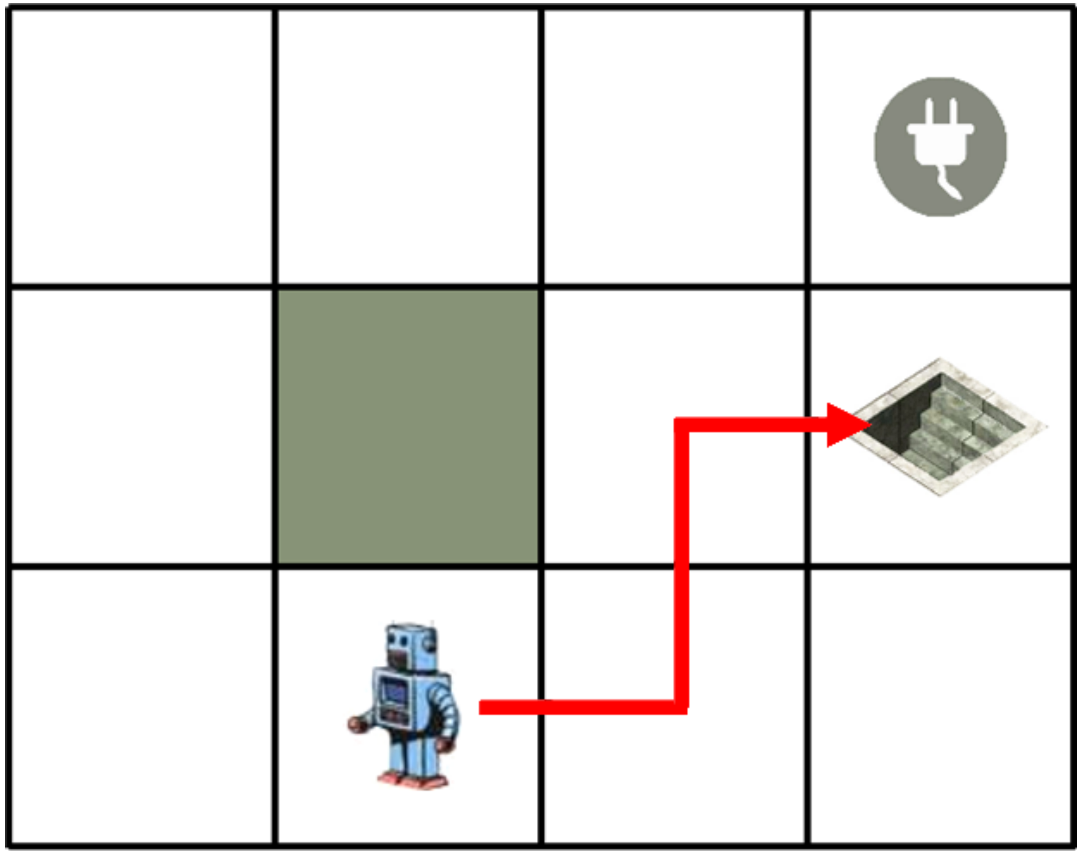
- Executing the **A* plan** in this environment



But: transitions are non-deterministic!

MDP Example

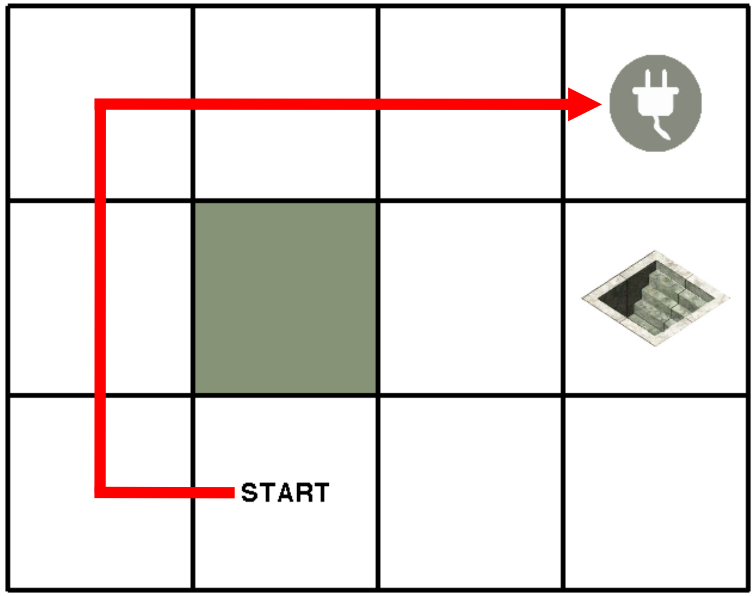
- Executing the **A* plan** in this environment



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:

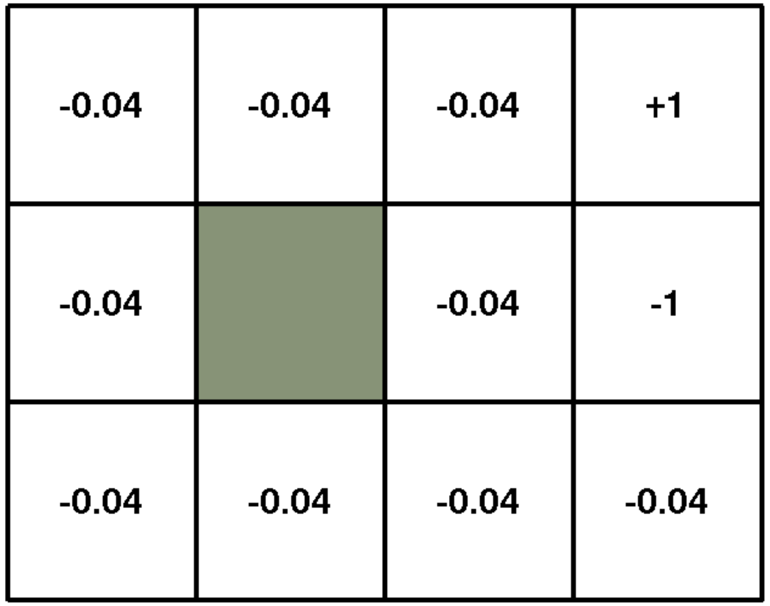
The transition probabilities from s to s' **depend only on the current state** s and not on the history of earlier states

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: considering 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 agents an **incentive 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

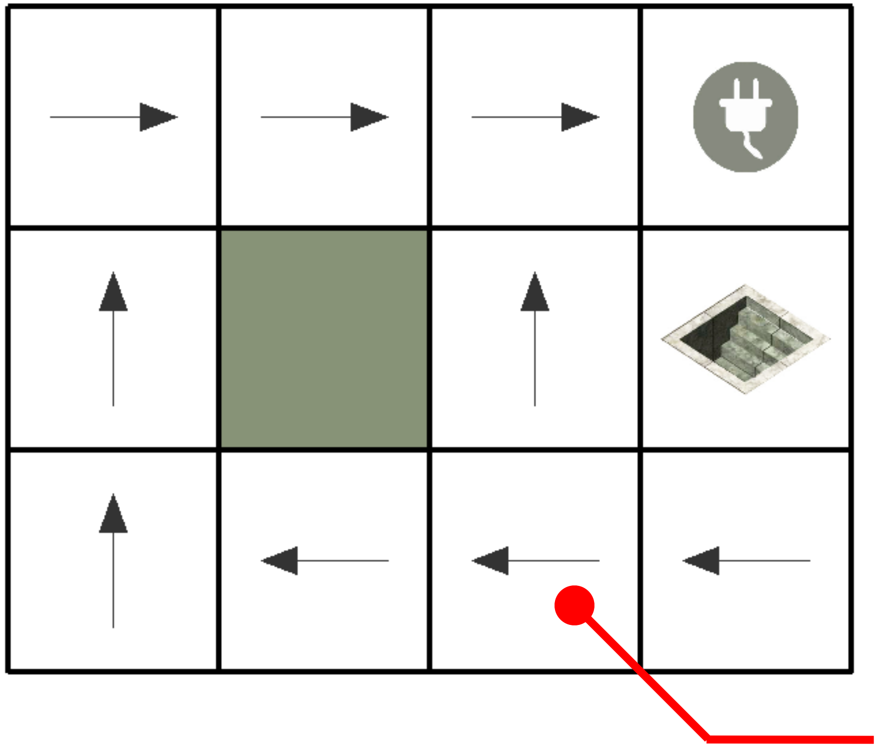
- An MDP solution is called **policy** π
- A policy is a mapping from states to actions

$$policy : States \mapsto Actions$$

- 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

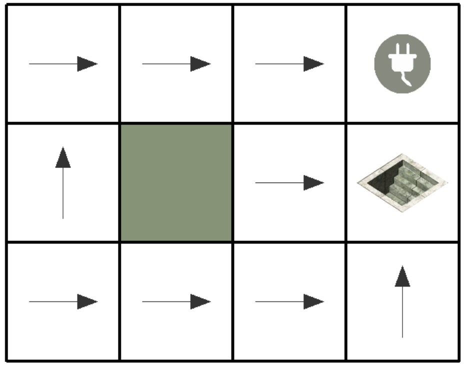
- The optimal policy for our example



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

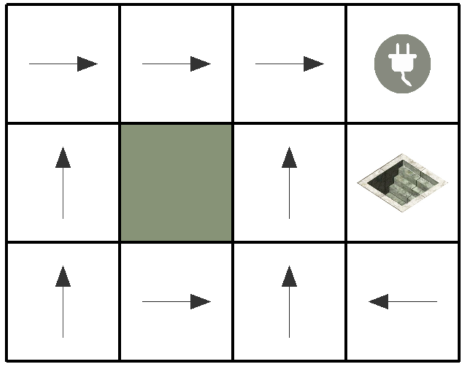
Policy

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



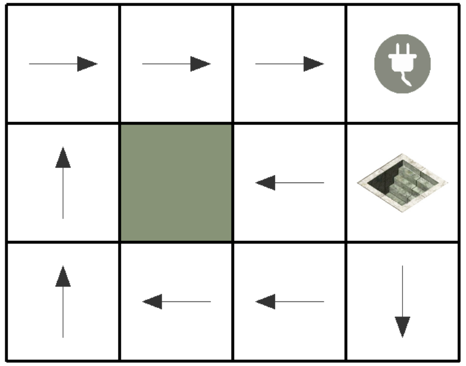
R = -2

Leave as soon as possible



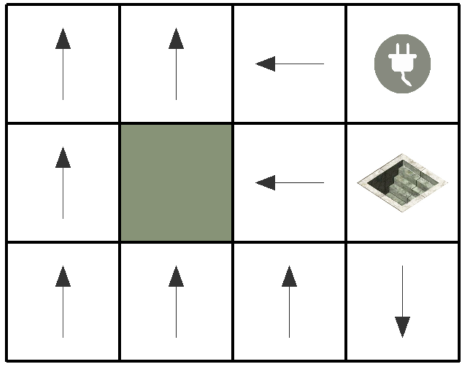
R = -0.2

Take shortcut, minor risks



R = -0.01

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'

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

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')

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

- Expectation taken over all policies

Optimal Policy

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

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

- 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

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

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 $0 \delta \gamma \delta 1$ 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:

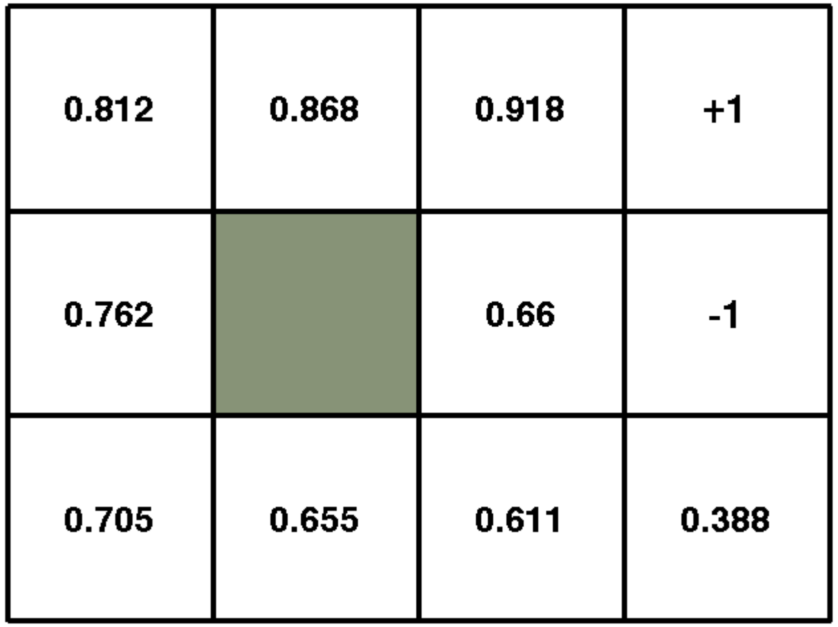
- Not all utility functions (for state sequences) can be used
- The utility function must have the **property of separability** (a.k.a. station-arity), e.g. additive utility functions:

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

- 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:

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

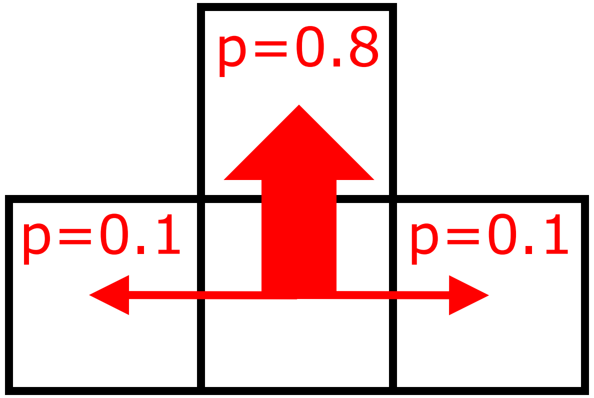
- Optimal utility: $U^* = \lim_{t \to \infty} U_t$
- Abort, if change in utility is below a threshold

Value Iteration Example

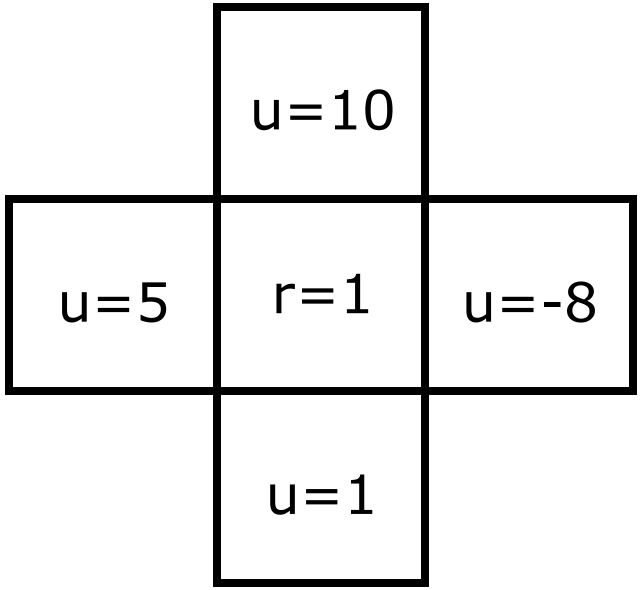
- Calculate utility of the center cell

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

desired action = *Up*



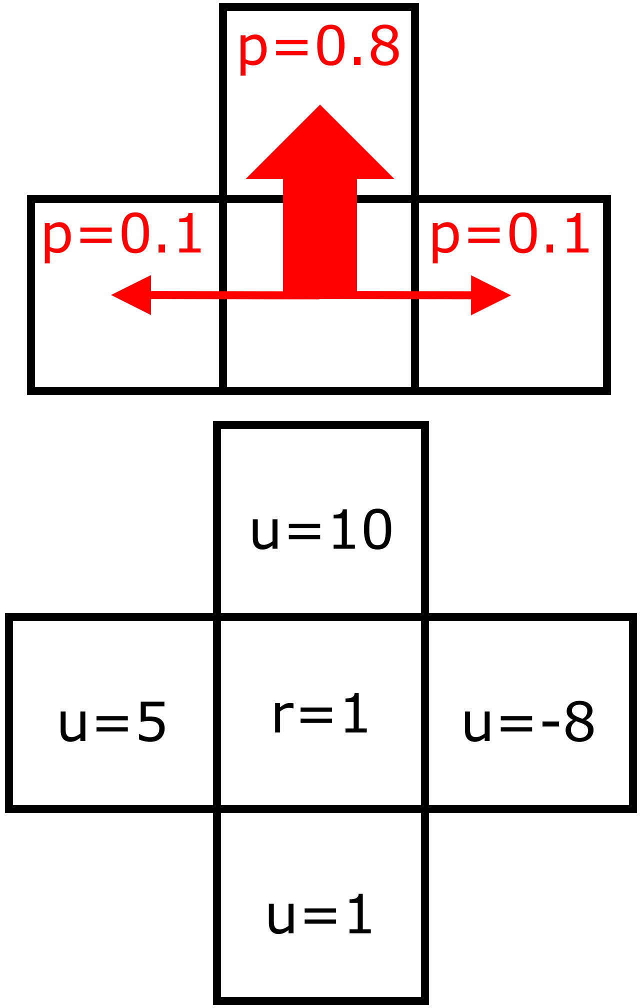
Transition Model



State space
(u=utility, r=reward)

Value Iteration Example

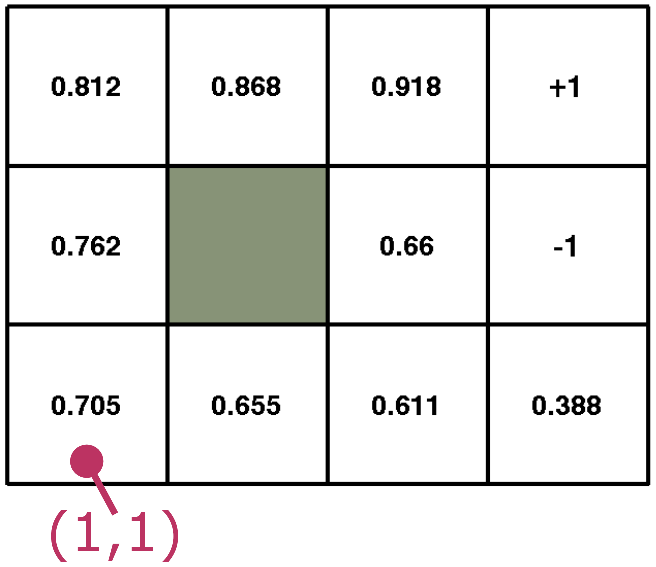
$$U_{i+1}(s) \leftarrow R(s) + \max_a \sum_{s'} T(s,a,s')\, U_i(s')$$



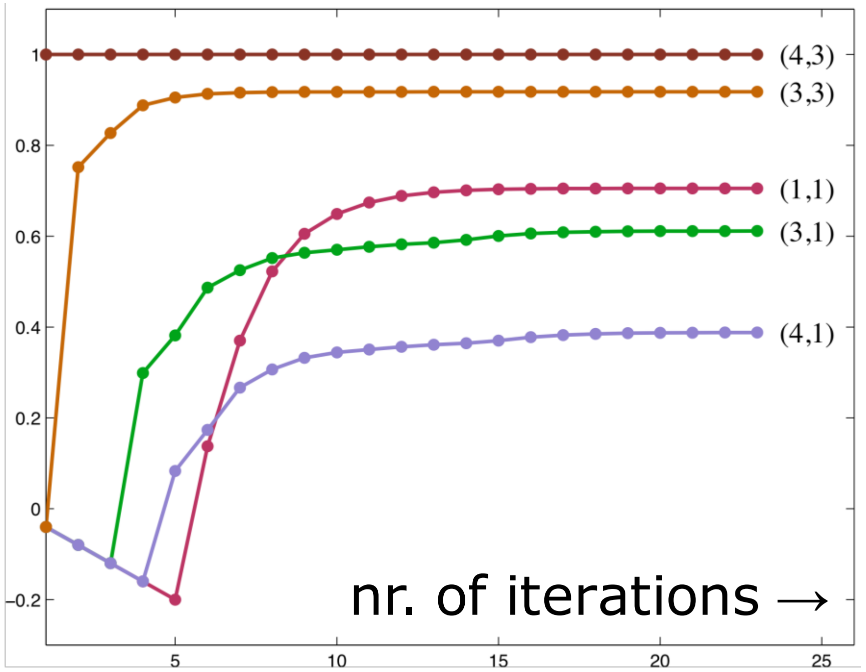
$$\begin{aligned}
&= \mathit{reward} + \max\{\\
&\quad 0.1 \cdot 1 + 0.8 \cdot 5 + 0.1 \cdot 10 \quad (\leftarrow),\\
&\quad 0.1 \cdot 5 + 0.8 \cdot 10 + 0.1 \cdot -8 \quad (\uparrow),\\
&\quad 0.1 \cdot 10 + 0.8 \cdot -8 + 0.1 \cdot 1 \quad (\rightarrow),\\
&\quad 0.1 \cdot -8 + 0.8 \cdot 1 + 0.1 \cdot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

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



- 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

Value Iteration

- Value Iteration finds the **optimal solution** to the Markov Decision Problem!
- **Converges** to the **unique solution** of the Bellman equation system
- Initial values for U' are arbitrary
- Proof involves the concept of *contraction*.
  $\| B\, U_i - B\, U'_i \| \leq \gamma \, \| U_i - U'_i \|$ 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

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

- **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**

Summary

- Robust navigation requires combined path planning & collision avoidance.
- Approaches need to consider robot's kinematic constraints and plans in the velocity space.
- Combination of search and reactive techniques show better results than the pure DWA in a variety of situations.
- Using the 5D-approach the quality of the trajectory scales with the performance of the underlying hardware.
- The resulting paths are often close to the optimal ones.

Summary

- Planning is a complex problem.
- Focus on subset of the configuration space:
  - road maps,
  - grids.
- Sampling algorithms are faster and have a trade-off between optimality and speed.
- Uncertainty in motion leads to the need of Markov Decision Problems.

What’s Missing?

- More complex vehicles (e.g., cars, legged robots, manipulators, …).
- Moving obstacles, motion prediction.
- High dimensional spaces.
- Heuristics for improved performances.
- Learning.