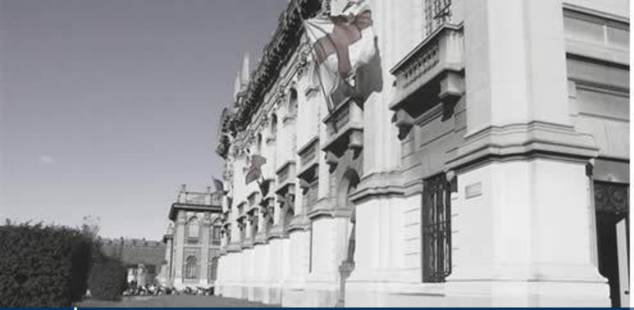


POLITECNICO DI MILANO

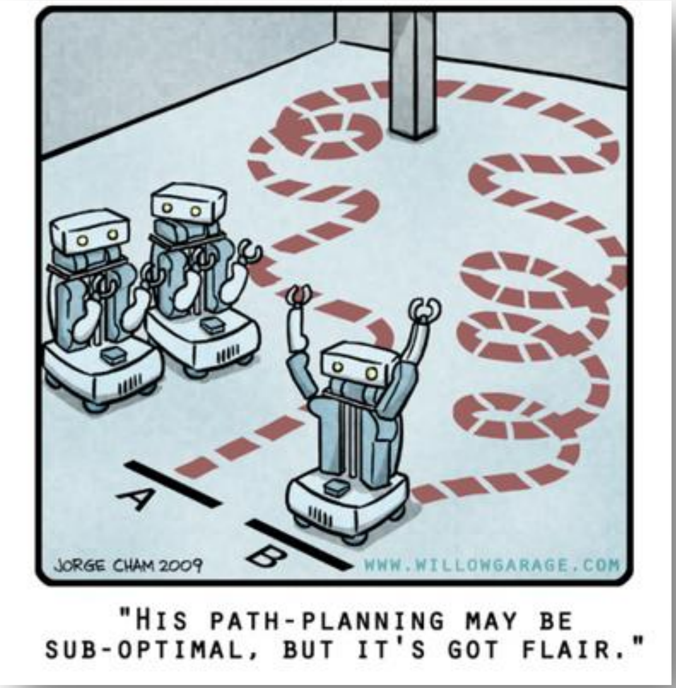
# Cognitive Robotics – Robot Motion Planning

Matteo Matteucci – matteo.matteucci@polimi.it

# Robot Motion Planning

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

J.-C. Latombe (1991)

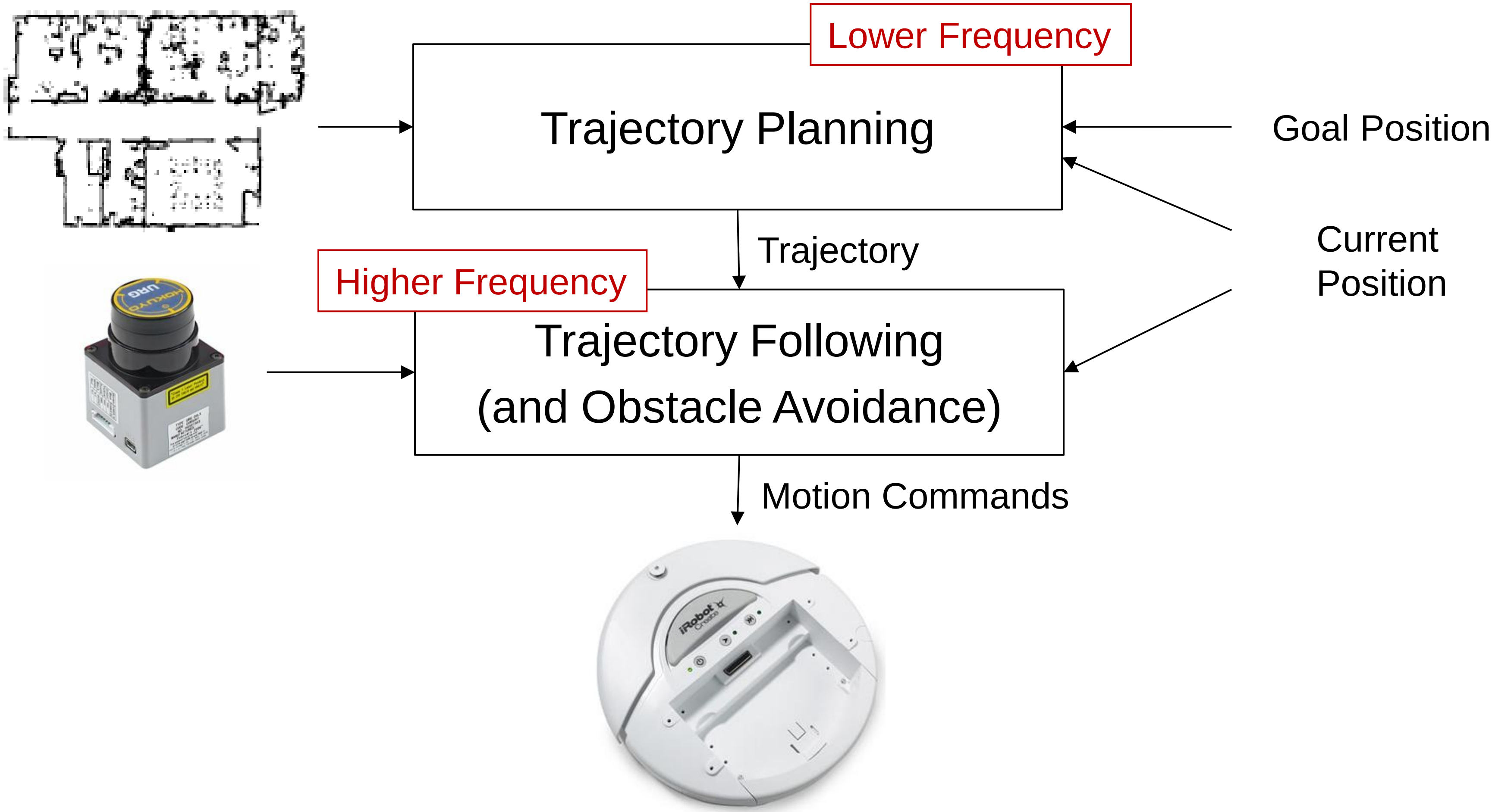



Robot Motion Planning Goals

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

Matteo Matteucci – matteo.matteucci@polimi.it

# A Two Layered Approach





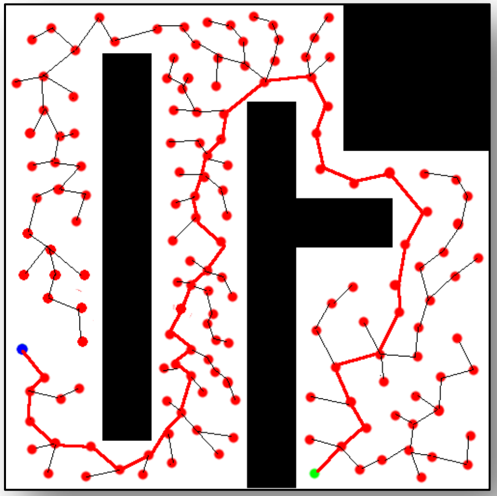
Matteo Matteucci – matteo.matteucci@polimi.it

# What a Planner?

Random Sampling

- PRMs
- RRT
- T-RRT
- SBL
- …

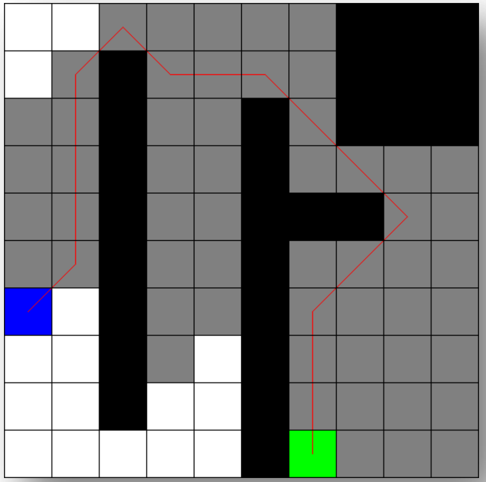
Open Motion Planning Library



Search Based Planning Algorithms

- A*
- ARA*
- ANA*
- *AD**
- D*
- …

Search Based Planning Library





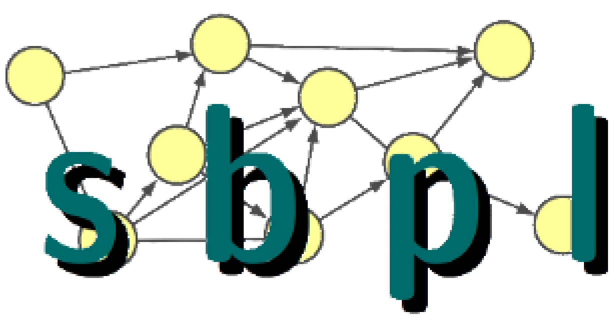
Matteo Matteucci – matteo.matteucci@polimi.it

# Pros and Cons

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

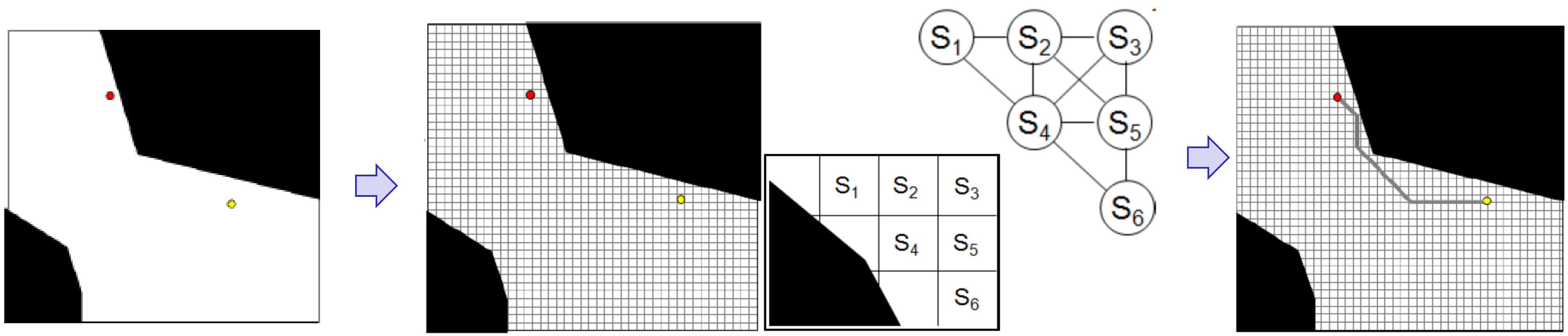
Lets have a look at Search Based Methods (SBPL) because of

- The generality of approaches
- Their theoretical guarantees
- Their simplicity



The overall idea:

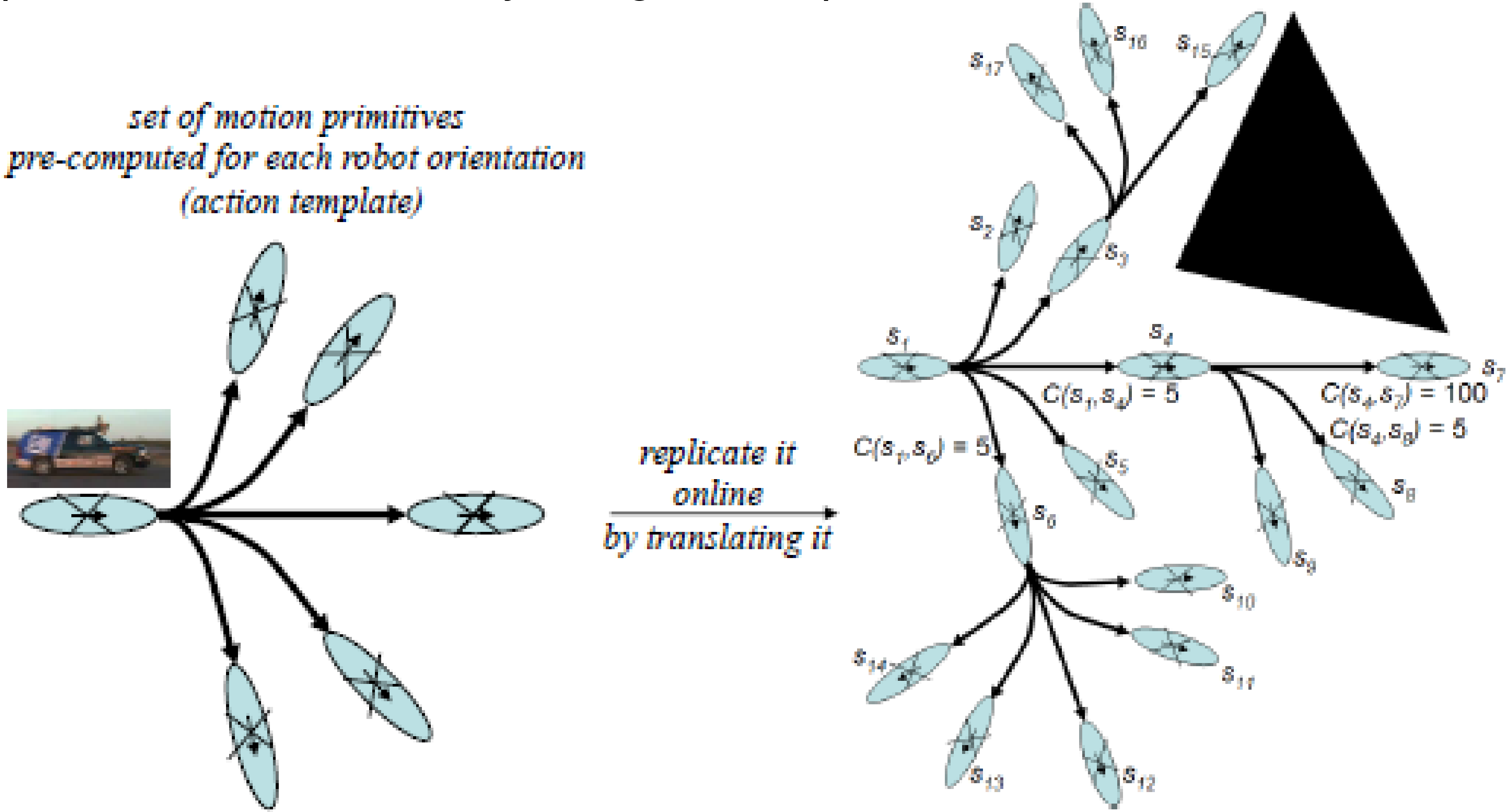
- Generate a discretized representation of the planning problem
- Build a graph out of this discretized representation (e.g., through 4 neighbors or 8 neighbors connectivity)
- Search the graph for the optimal solution




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

# Lattice Based Graphs for Navigation

Graph can be constructed by using motion primitives



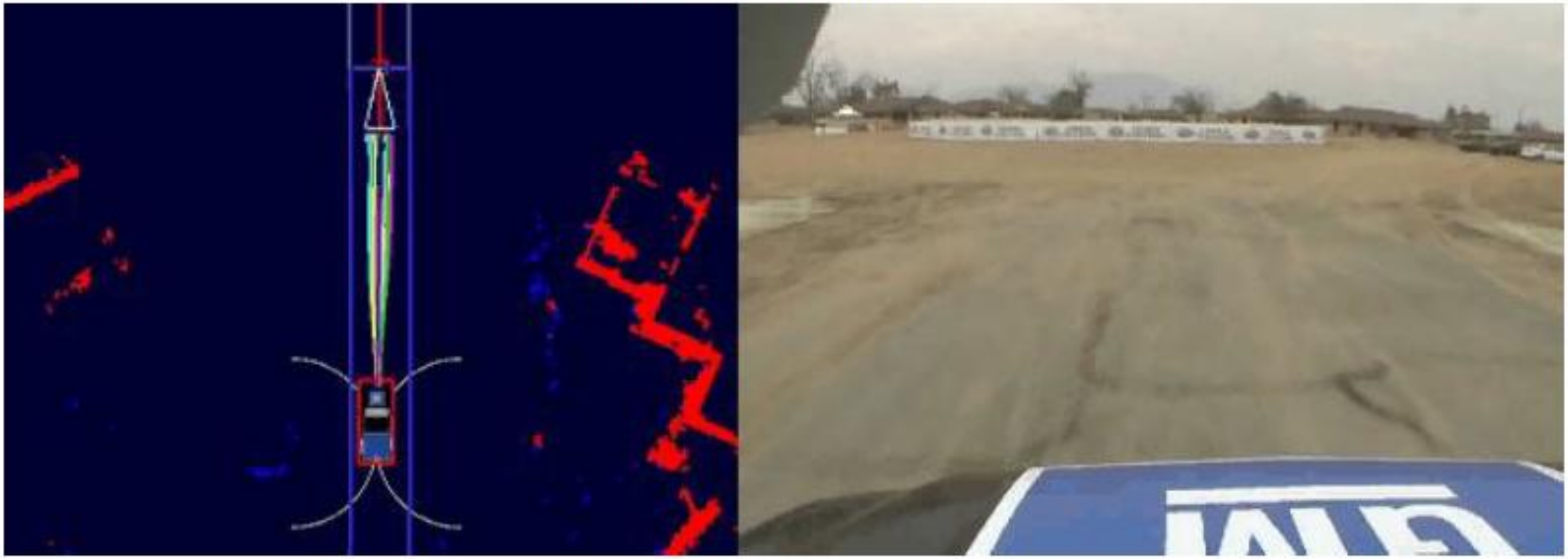

- Pros: sparse graph, feasible path, incorporate a variety of constraints
- Cons: possible incompleteness

# Lattice Based Graphs for Navigation

Graph can be constructed by using motion primitives

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

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



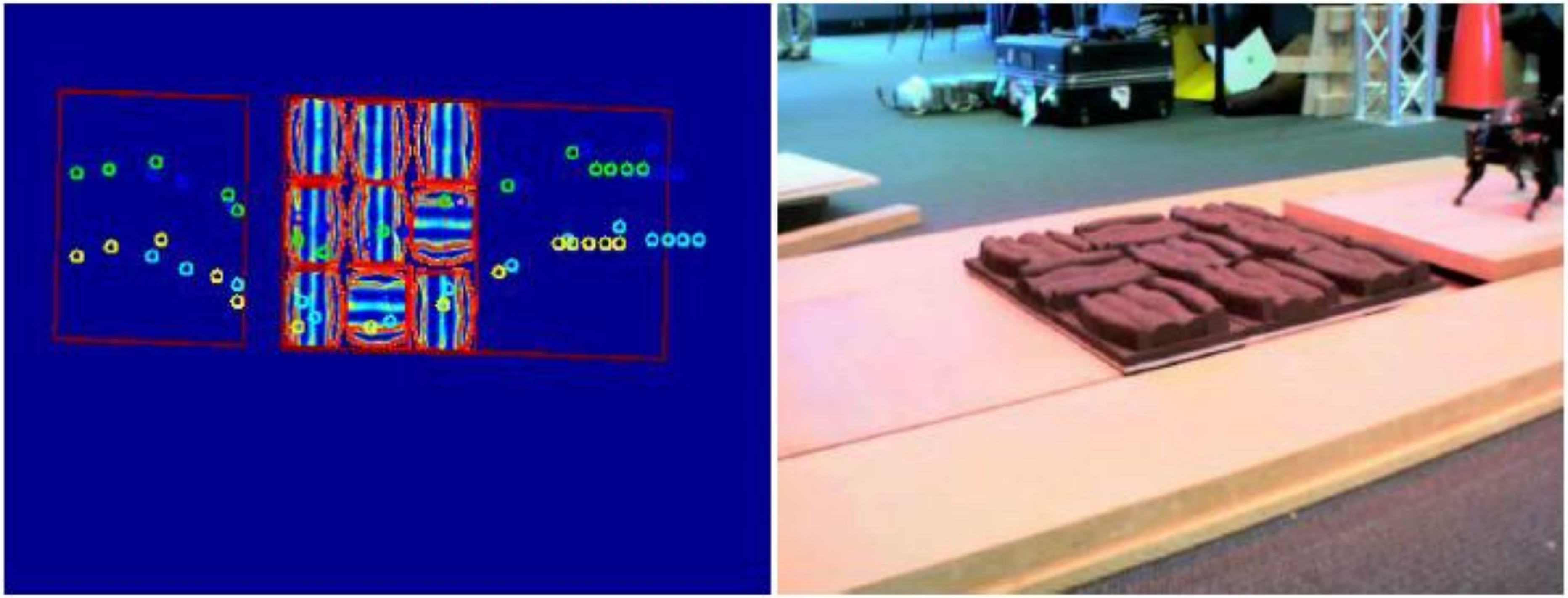

Matteo Matteucci – matteo.matteucci@polimi.it

# Lattice Based Graphs for Navigation

Graph can be constructed by using motion primitives

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

*planning in 8D (foothold planning) lattice-based graph for quadrupeds [Vernaza et al., '09]*
*using R* search [Likhachev & Stentz, '08]*




Matteo Matteucci – matteo.matteucci@polimi.it

# Planning Problem Ingredients

Typical components of a Search-based Planner

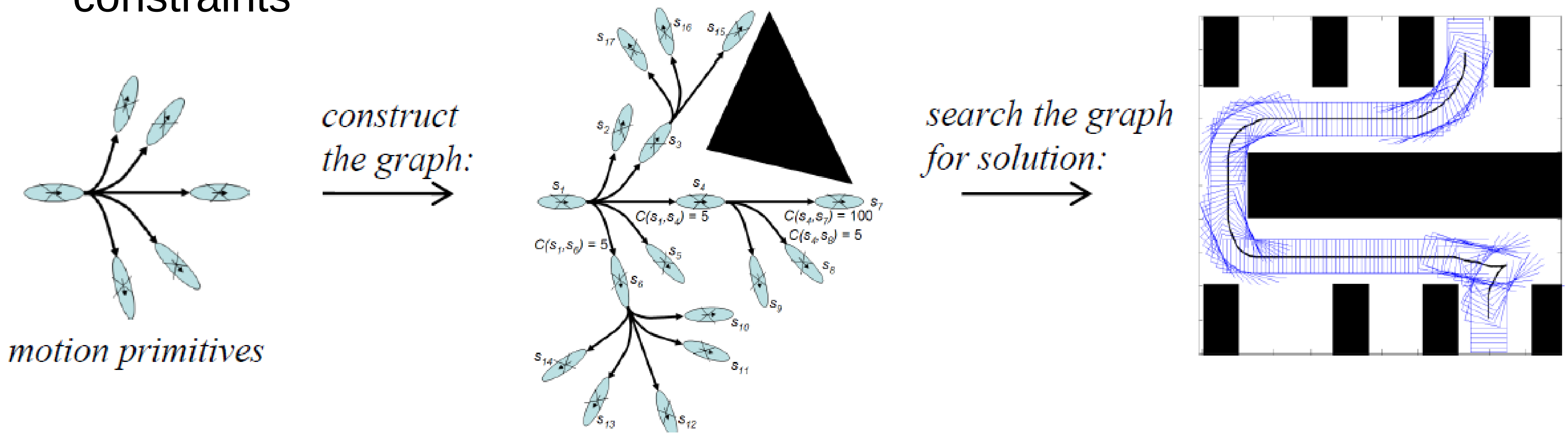
- Graph construction (given a state what are its successor states)
- Cost function (a cost associated with every transition in the graph)
- Heuristic function (estimates of cost-to-goal)

**Domain Dependent**

- Graph search algorithm (for example, A* search)

**Domain Independent**

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



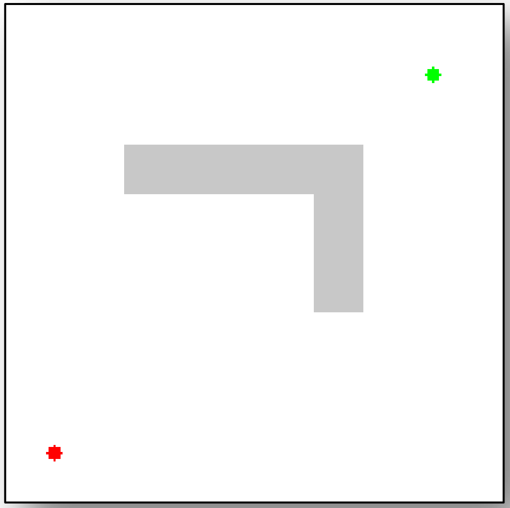


Matteo Matteucci – matteo.matteucci@polimi.it

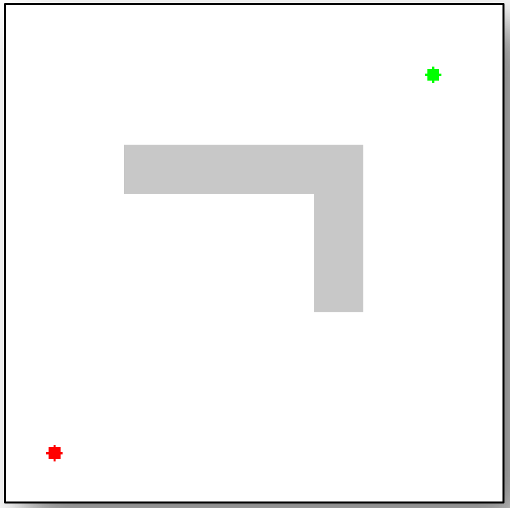
# Exact and Approximate Planning (in SBPL)

Different algorithms are available

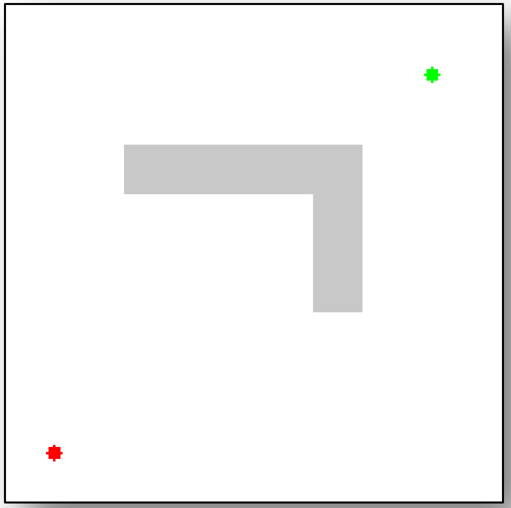
- Returning the optimal path (e.g., Dijstra, A*, …)
- Returning an ε sub-optimal path
  (e.g., weighted A*, ARA*, AD*, R*, D* Lite, ...)



Dijstra



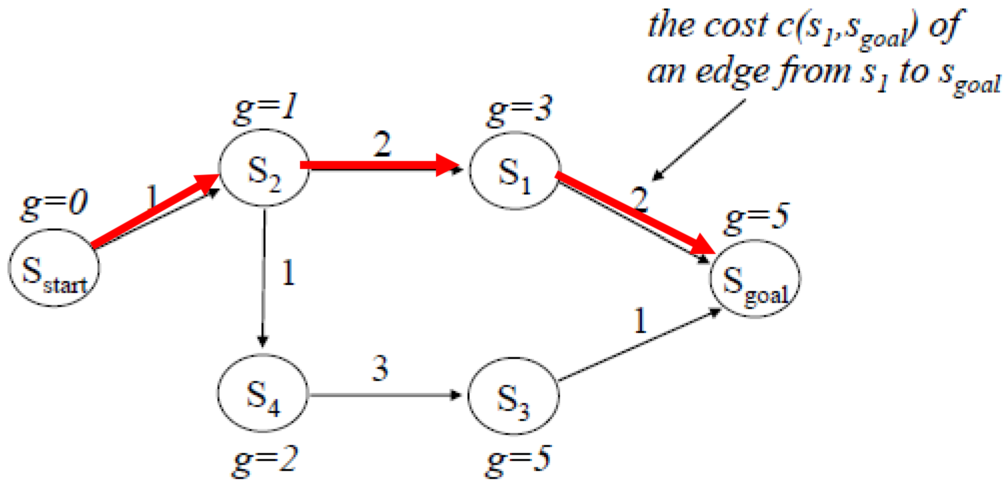
A*



weighted A*

# Searching Graphs for Least Cost Path

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




Many search algorithms compute optimal <u>g-values</u> for relevant states

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

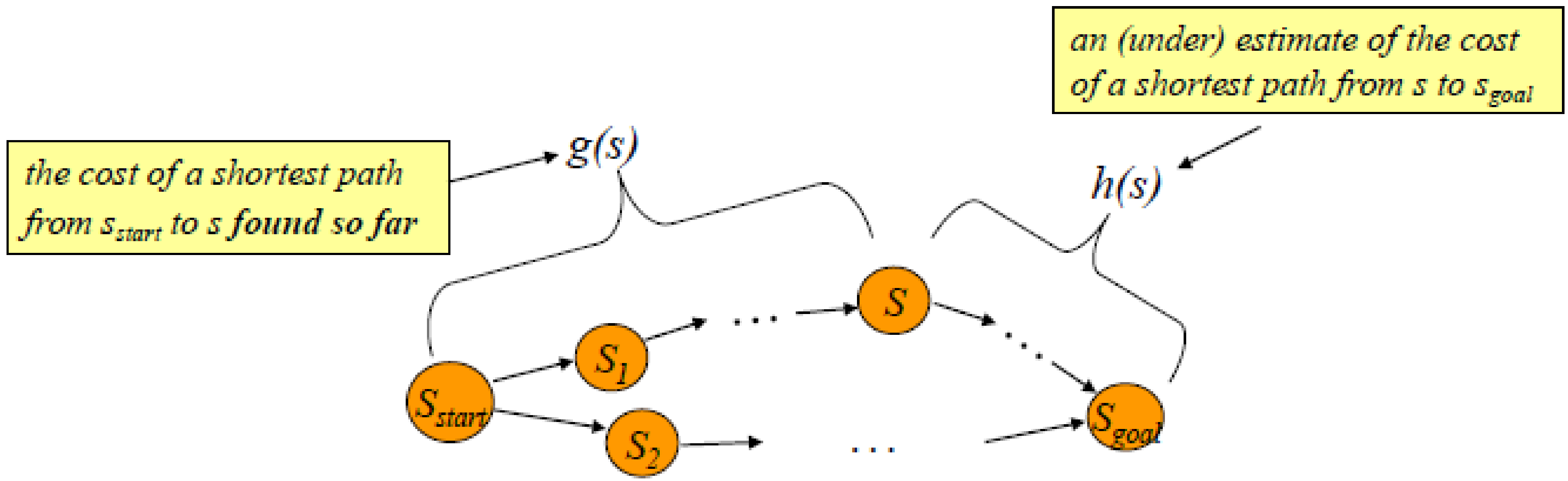
Least-cost path is a greedy path computed by backtracking:

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

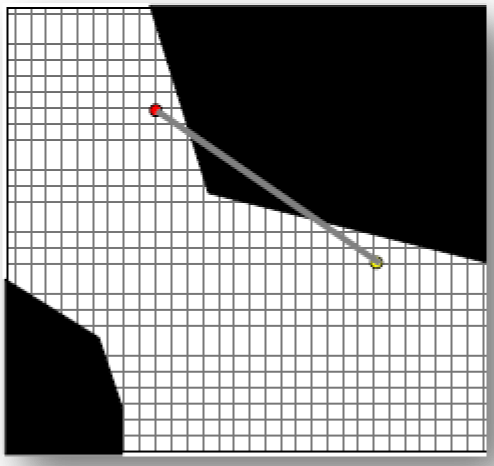
$$s' = \text{argmin}_{s'' \text{ in pred}(s)} (g(s'')+c(s'',s))$$


Matteo Matteucci – matteo.matteucci@polimi.it

A* speeds up search by computing g-values for relevant states as




Heuristic function must be



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

Admissibility follows from consistency and often consistency follows from admissibility

## Main function

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

*Set of candidates for expansion*

## ComputePath function

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

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


Matteo Matteucci – matteo.matteucci@polimi.it

# A* Search Algorithm

**Main function**

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

*Set of candidates for expansion*

**ComputePath function**

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

*Set of states already expanded*

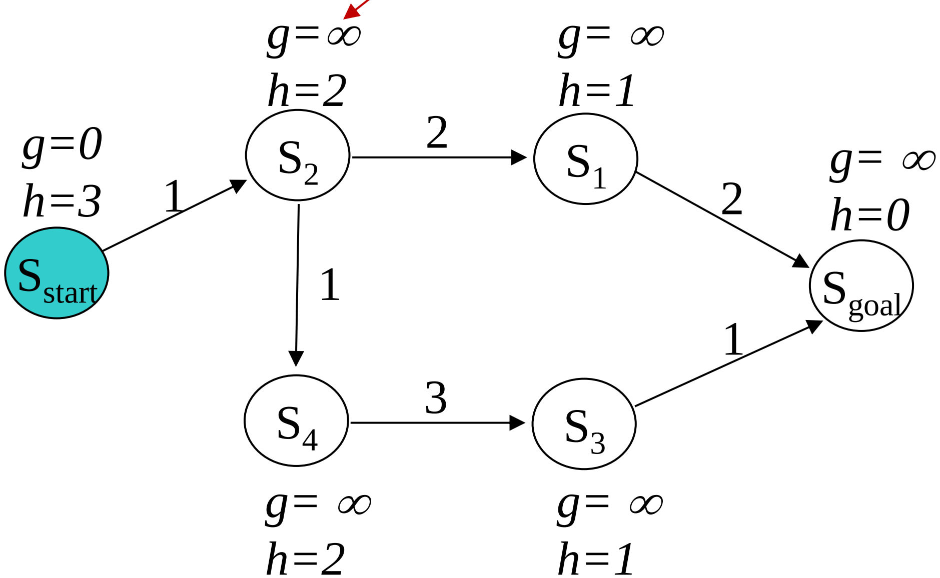
*Tries to decrease g(s') using the found path from $s_{start}$ to s*


Matteo Matteucci – matteo.matteucci@polimi.it

**ComputePath function**

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

$g(s_2) > g(s_{start}) + c(s_{start},s2)$




*CLOSED* = {}

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

*next state to expand:* $s_{start}$

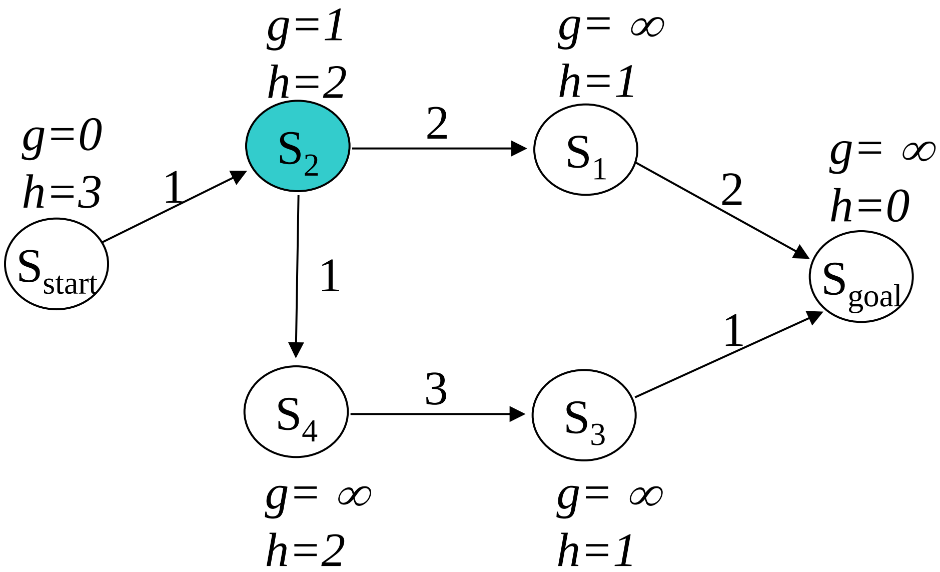
## ComputePath function

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

*CLOSED* = $\{s_{start}\}$

*OPEN* = $\{s_2\}$

*next state to expand:* $s_2$



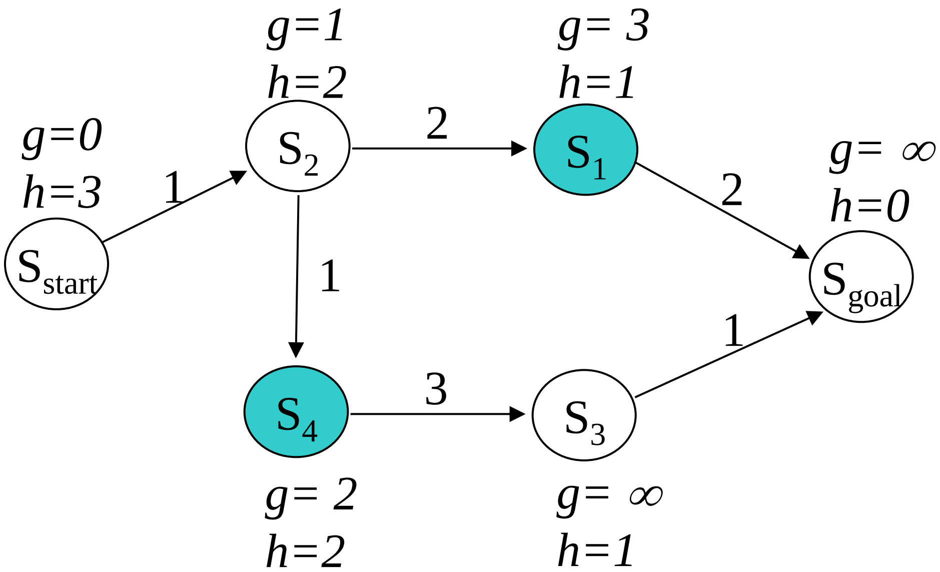
## ComputePath function

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

*CLOSED* = $\{s_{start}, s_2\}$

*OPEN* = $\{s_1, s_4\}$

*next state to expand:* $s_1$



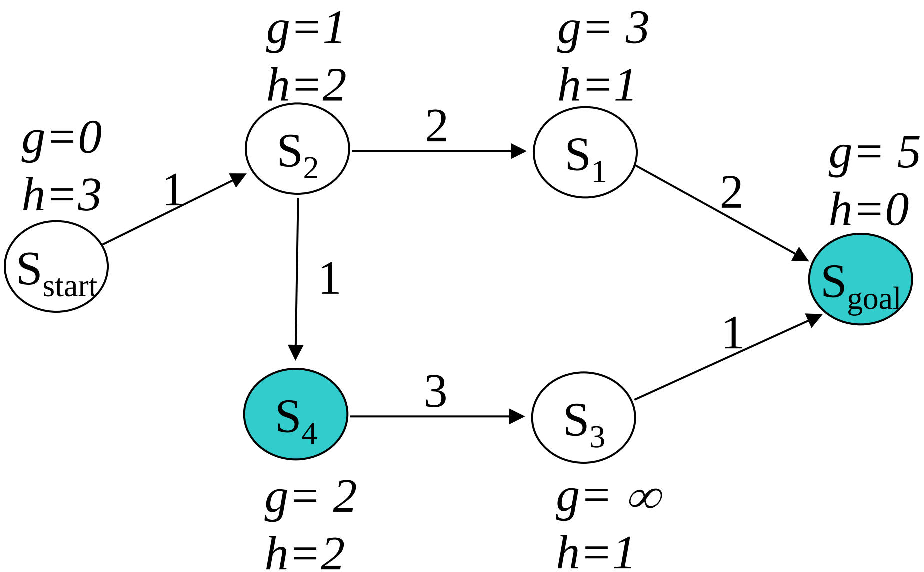
## ComputePath function

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

*CLOSED* = $\{s_{start}, s_2, s_1\}$

*OPEN* = $\{s_4, s_{goal}\}$

*next state to expand:* $s_4$



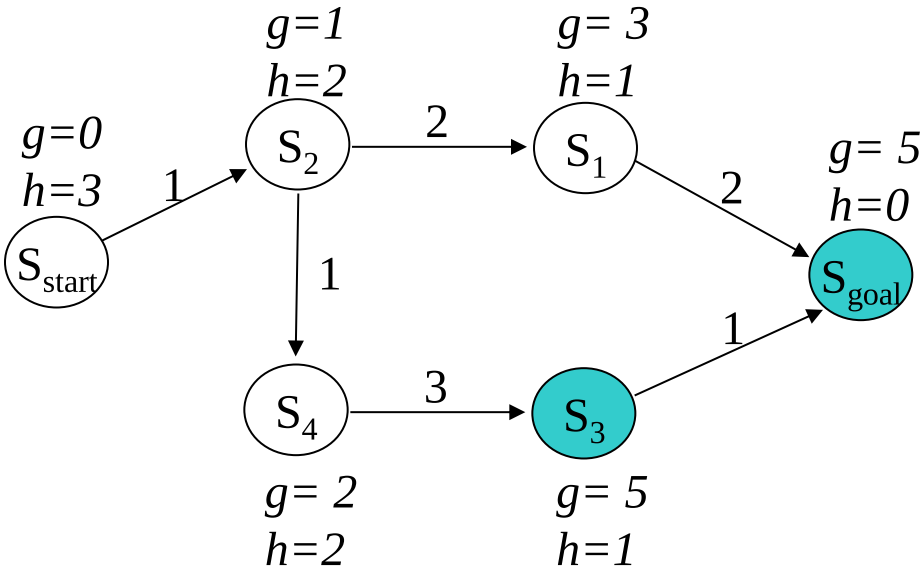
## ComputePath function

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

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

*OPEN* = $\{s_3, s_{goal}\}$

*next state to expand:* $s_{goal}$



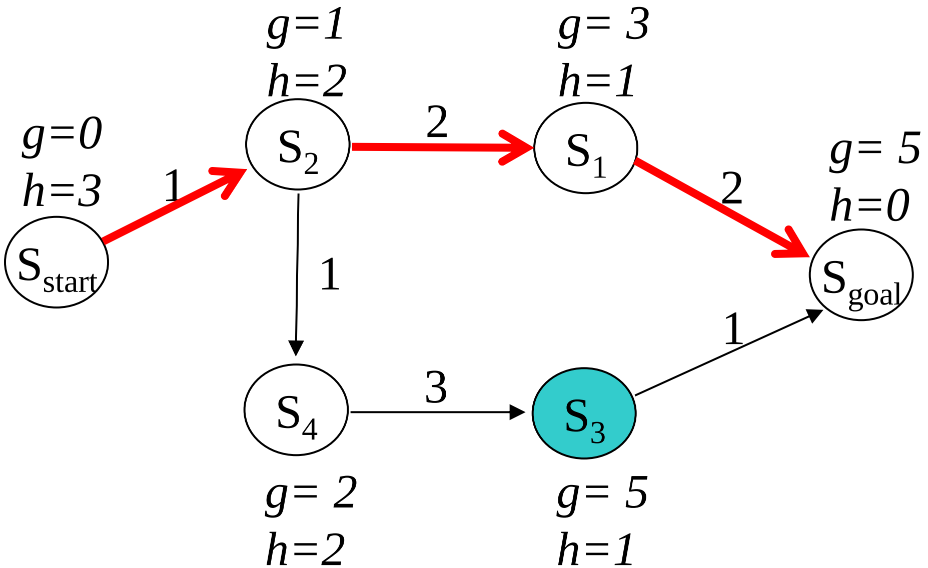
## ComputePath function

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

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

$OPEN = \{s_3\}$

<u>*DONE!*</u>



# A* Properties

A* is guaranteed to

- return an optimal path in terms of the solution
- perform provably minimal number of state expansions

Algorithms state expansion:

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

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


Matteo Matteucci – matteo.matteucci@polimi.it

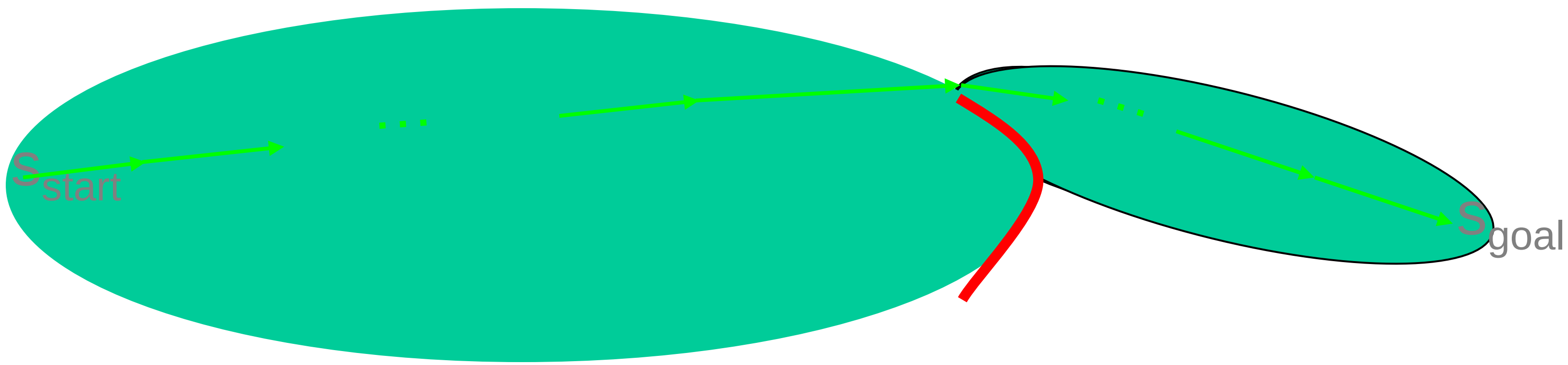
Algorithms state expansion:

- Dijkstra's: expands states in the order of *f* = *g* values (roughly)

# A* Properties

Algorithms state expansion:

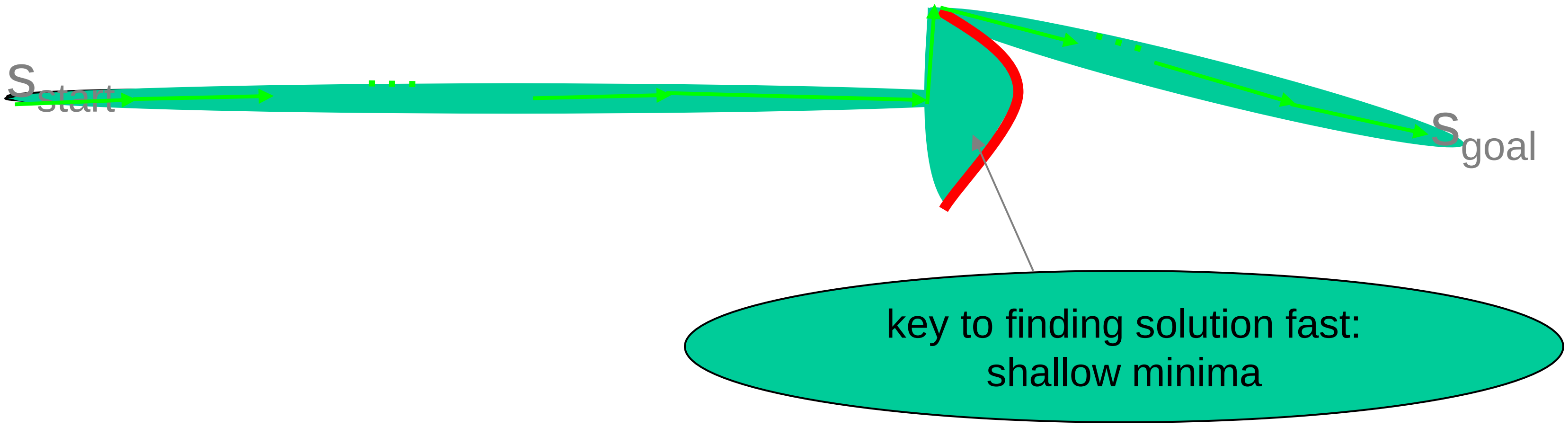
- Dijkstra's: expands states in the order of $f = g$ values (roughly)
- A* Search: expands states in the order of $f = g + h$ values



# A* Properties

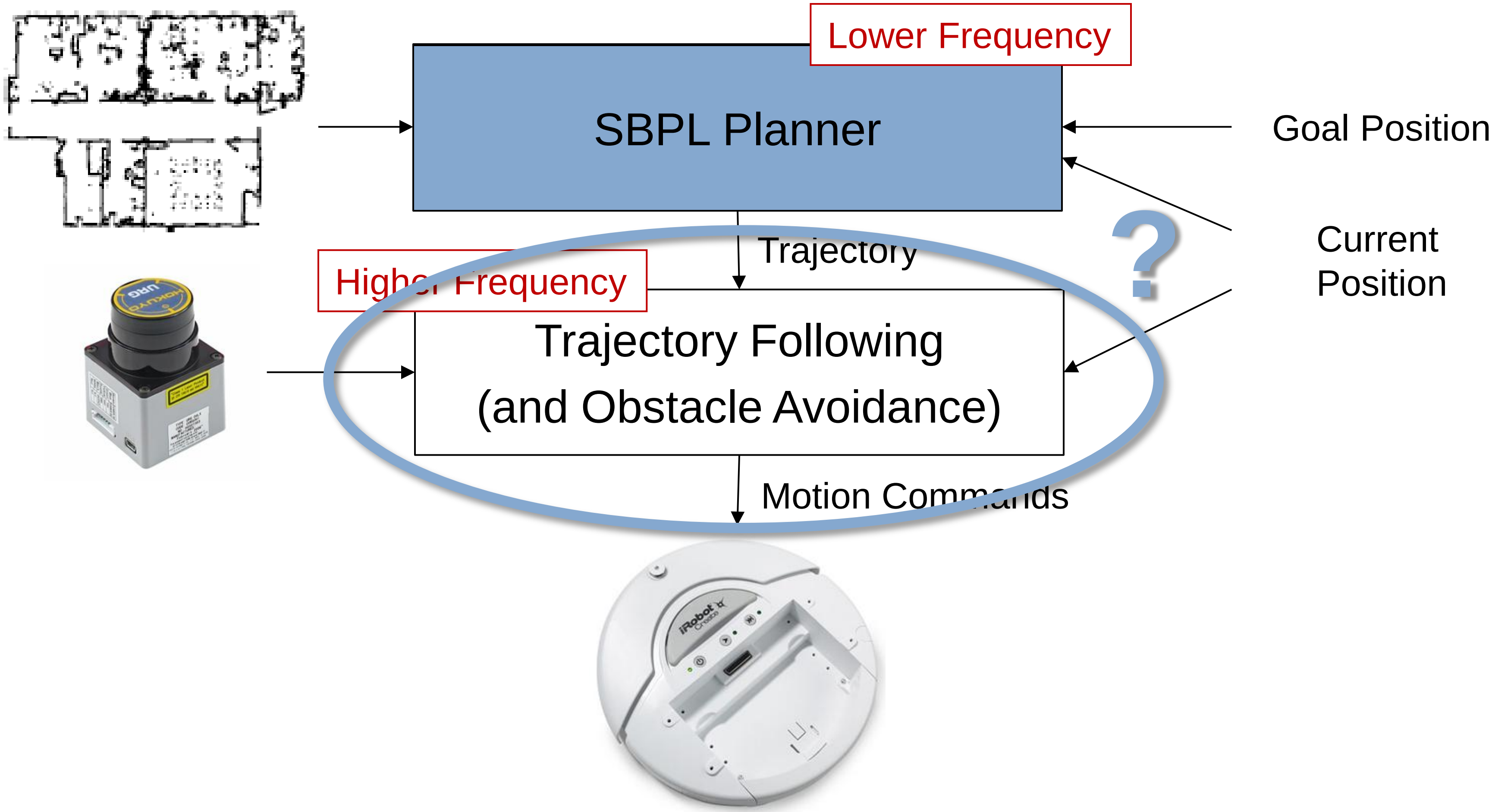
Algorithms state expansion:

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






Matteo Matteucci – matteo.matteucci@polimi.it


Lower Frequency
SBPL Planner
Goal Position
Current
Position
Trajectory
?
Higher Frequency
Trajectory Following
(and Obstacle Avoidance)
Motion Commands

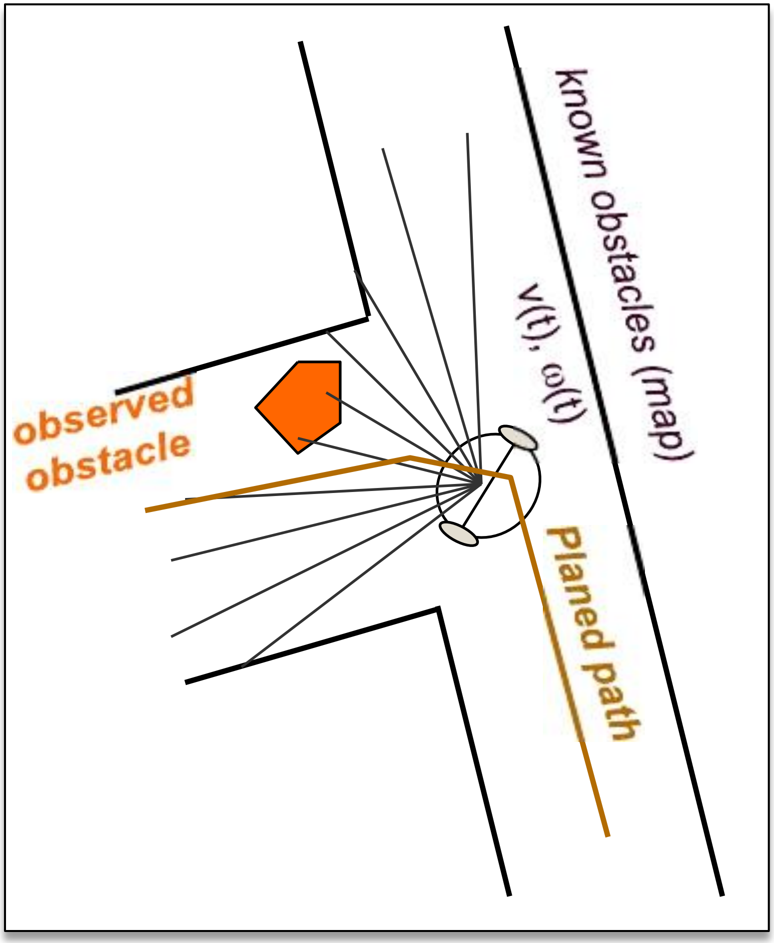
# Obstacle Avoidance (Local Path Planning)

Obstacle avoidance should:

- Follow the planned path
- Avoid unexpected obstacle,
  i.e., those that were not in the map

Several proposed methods in the literature

- Potential field methods
  [Borenstein, 1989]
- Vector field histogram
  [Borenstein, 1991, 1998, 2000]
- Nearness diagram
  [Minguez & Montano, 2000]
- Curvature-Velocity [Simmons, 1996]
- Dynamic Window Approach
  [Fox, Burgard, Thrun, 1997]
- …



# The Simplest One …

"Bugs" have little if any knowledge …

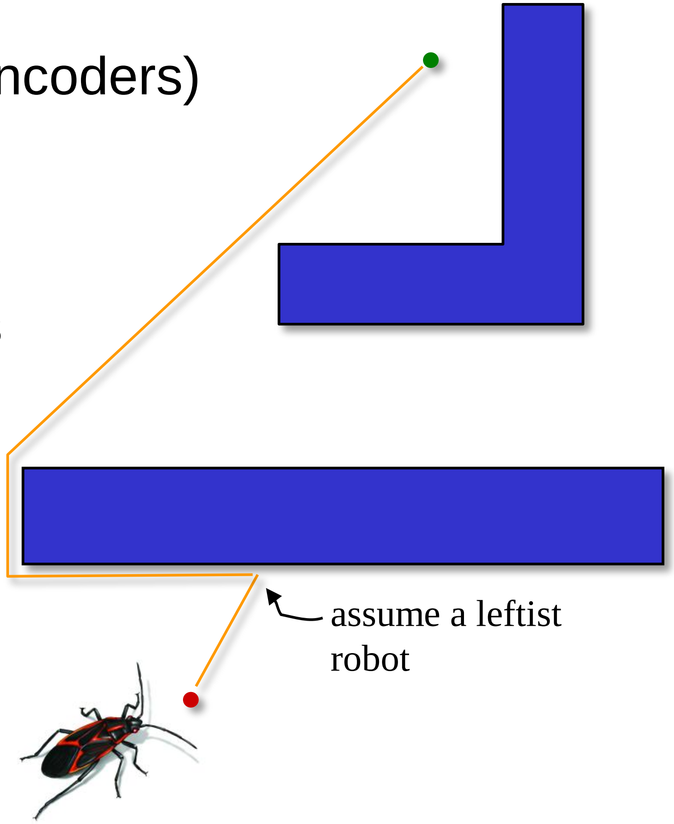
- known direction to the goal
- only local sensing  (walls/obstacles + encoders)

… and their world is reasonable!

- finite obstacles in any finite range
- a line intersects an obstacle finite times

Switch between two basic behaviors

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



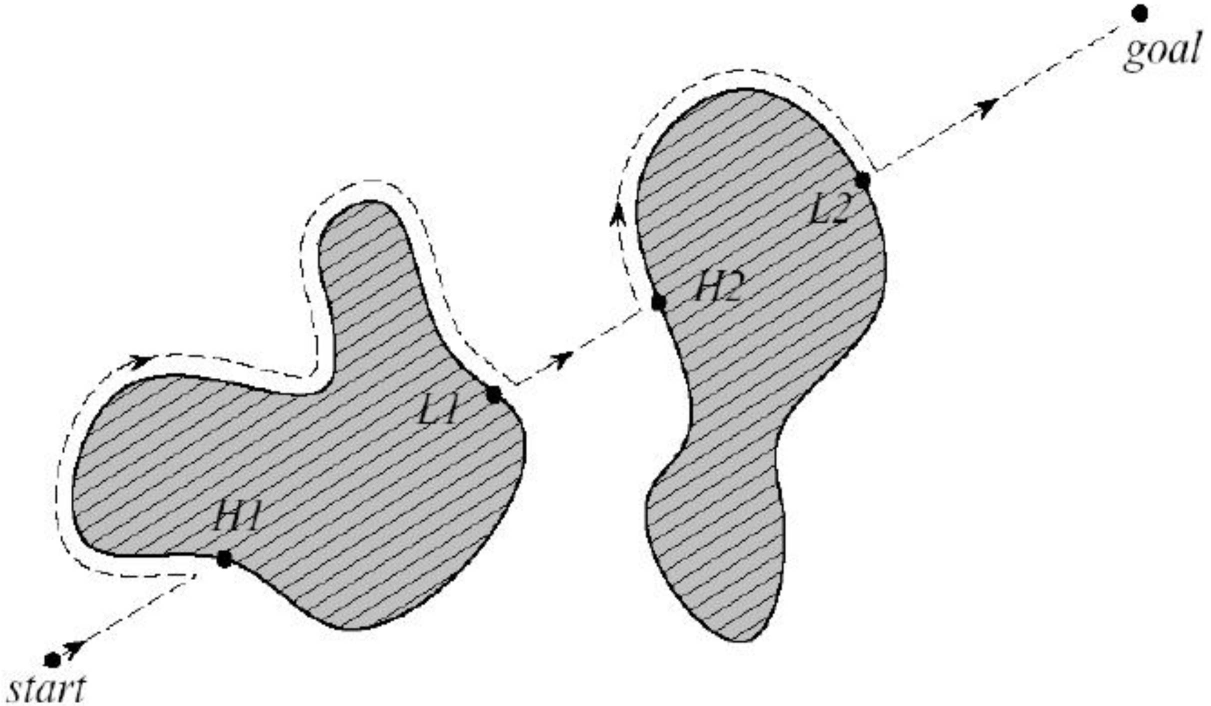


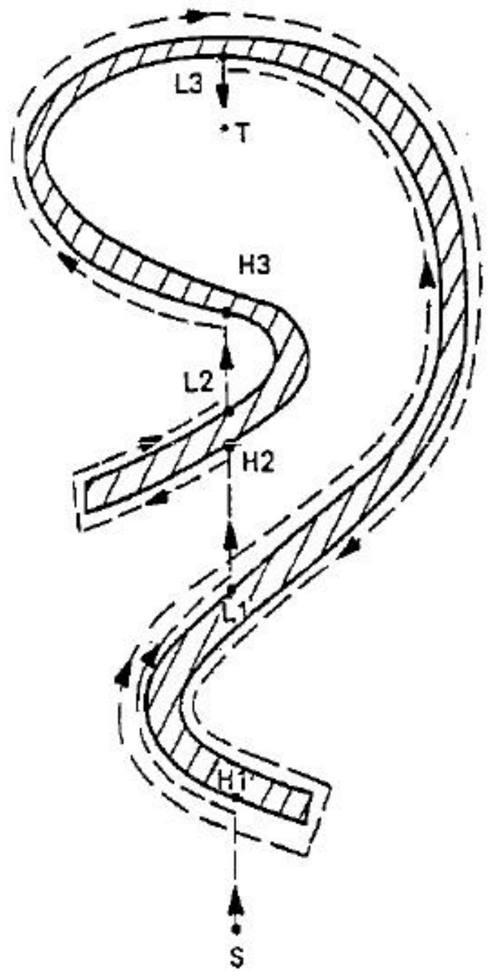
Matteo Matteucci – matteo.matteucci@polimi.it

# Bugs and Features …

Each obstacle is fully circled before it is left at the point closest to the goals

- Advantages
  - No global map required
  - Completeness guaranteed
- Disadvantages
  - Solution are often highly suboptimal



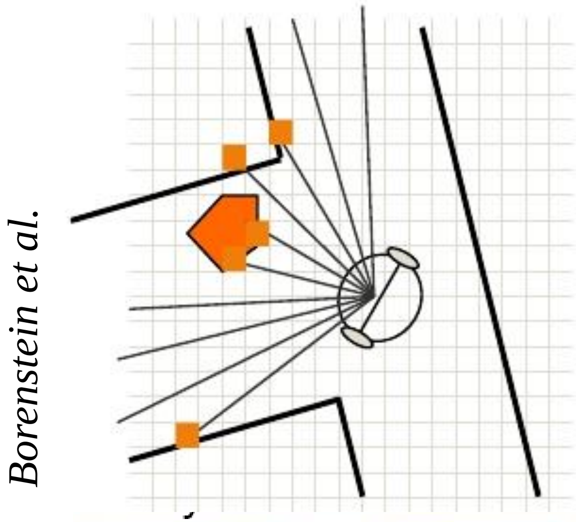
# Vector Field Histograms (VHF) *[Borenstein et al. 1991]*

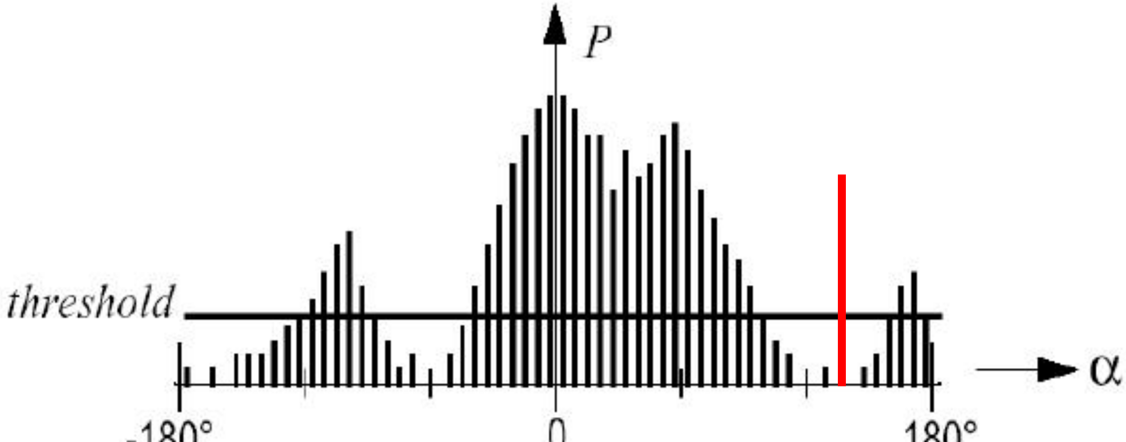
Use a local map of the environment and evaluate the angle to drive towards

- Environment represented in a grid (2 DOF) with
- The steering direction is computed in two steps:
  - all openings for the robot to pass are found
  - the one with lowest cost function *G* is selected

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

target_direction = alignment of the robot path with the goal
wheel_orientation = difference between the new direction and the currrent wheel orientation
previous_direction = difference between the previously selected direction and the new direction



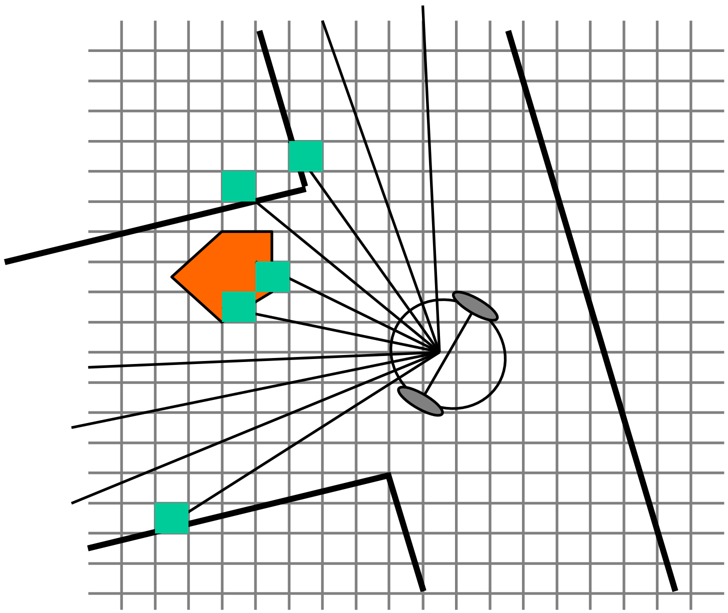


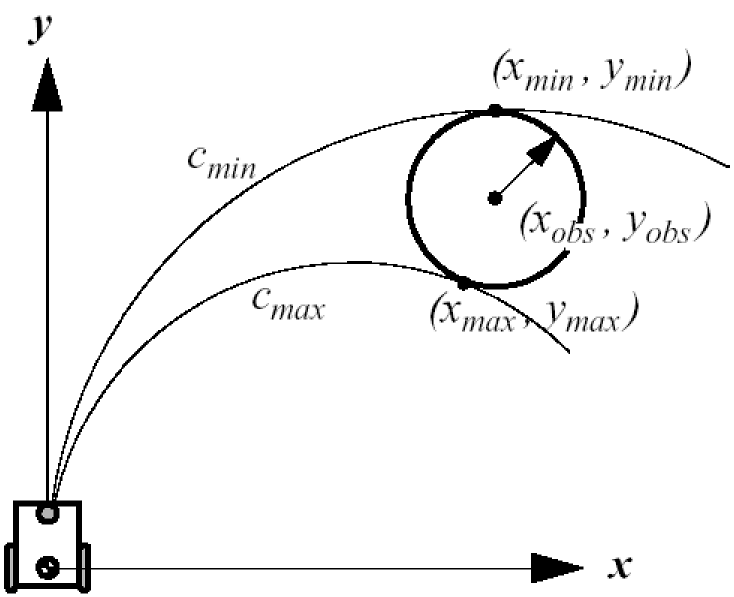
Matteo Matteucci – matteo.matteucci@polimi.it

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

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

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



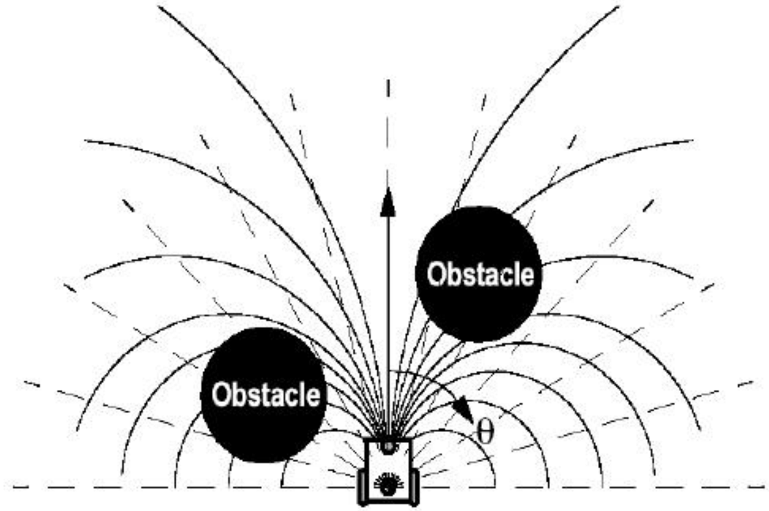



*Simmons et al.*

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

VHF+ accounts also in a very simplified way for vehicle kinematics

- robot moving on arcs or straight lines
- obstacles blocking a given direction also blocks all the trajectories (arcs) going through this direction like in an Ackerman vehicle
- obstacles are enlarged so that all kinematically blocked trajectories are properly taken into account




*Borenstein et al.*

However VHF+ as VHF suffers

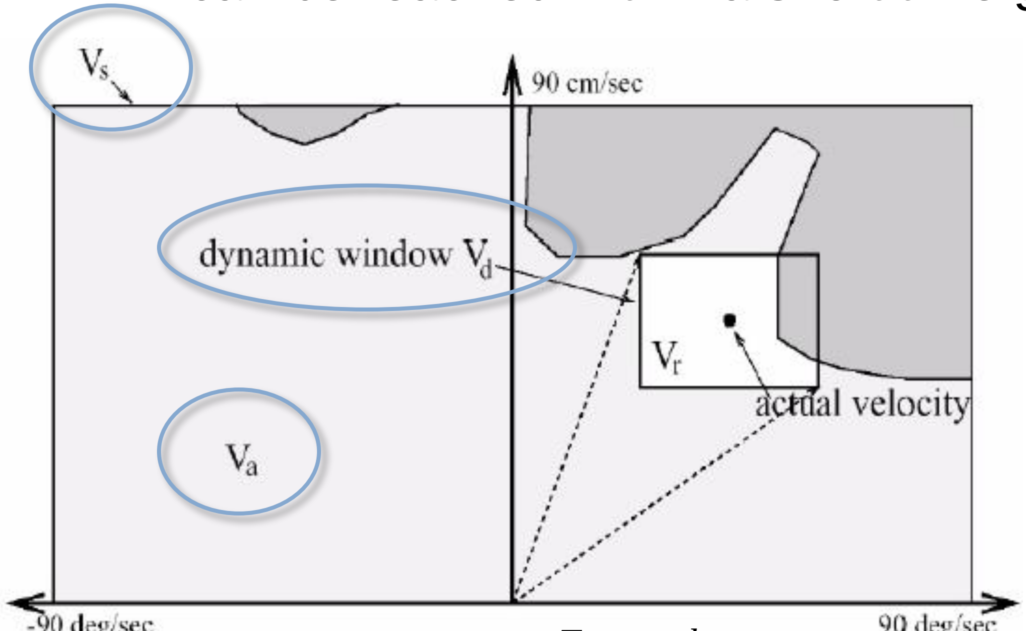
- Limitation if narrow areas (e.g. doors) have to be passed
- Local minima might not be avoided
- Reaching of the goal can not be guaranteed
- Dynamics of the robot not really considered


Matteo Matteucci – matteo.matteucci@polimi.it

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

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

- Consider only <u>circular trajectories</u> determined by pairs $V_s=(v,\omega)$ of translational and rotational speeds
- A pair $V_a=(v, \omega)$ is considered <u>admissible</u>, if the robot is able to stop before it reaches the closest obstacle on the corresponding curvature.
- A <u>dynamic window</u> restricts the reachable velocities $V_d$ to those that can be reached within a short time given limited robot accelerations



*Fox et al.*

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

DWA Search Space

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

# How to choose (v,ω)?

Steering commands are chosen maximizing a heuristic navigation function:

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

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

Alignment with target direction

Distance to closest obstacle intersecting with curvature

Forward velocity of the robot

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

Forward robot velocity

Follows global path

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

Cost to reach the goal

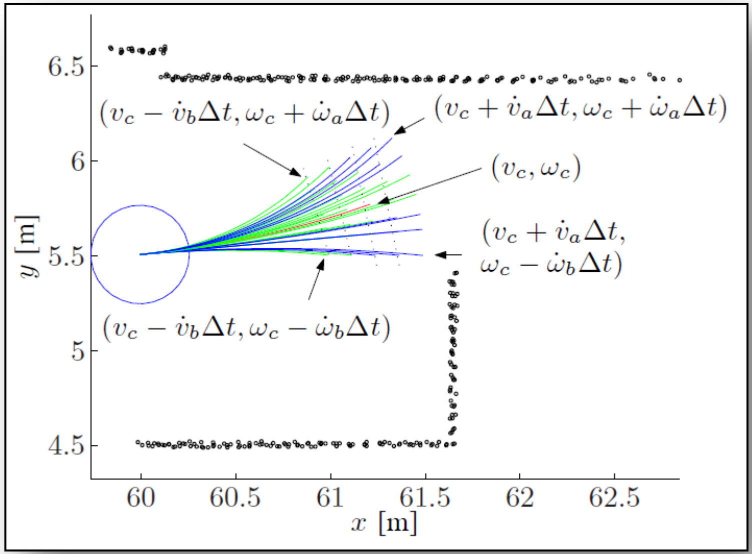
Goal nearness


Matteo Matteucci – matteo.matteucci@polimi.it

# DWA Algorithm (as implemented in ROS movebase)

The basic idea of the Dynamic Window Approach (DWA) algorithm follows …

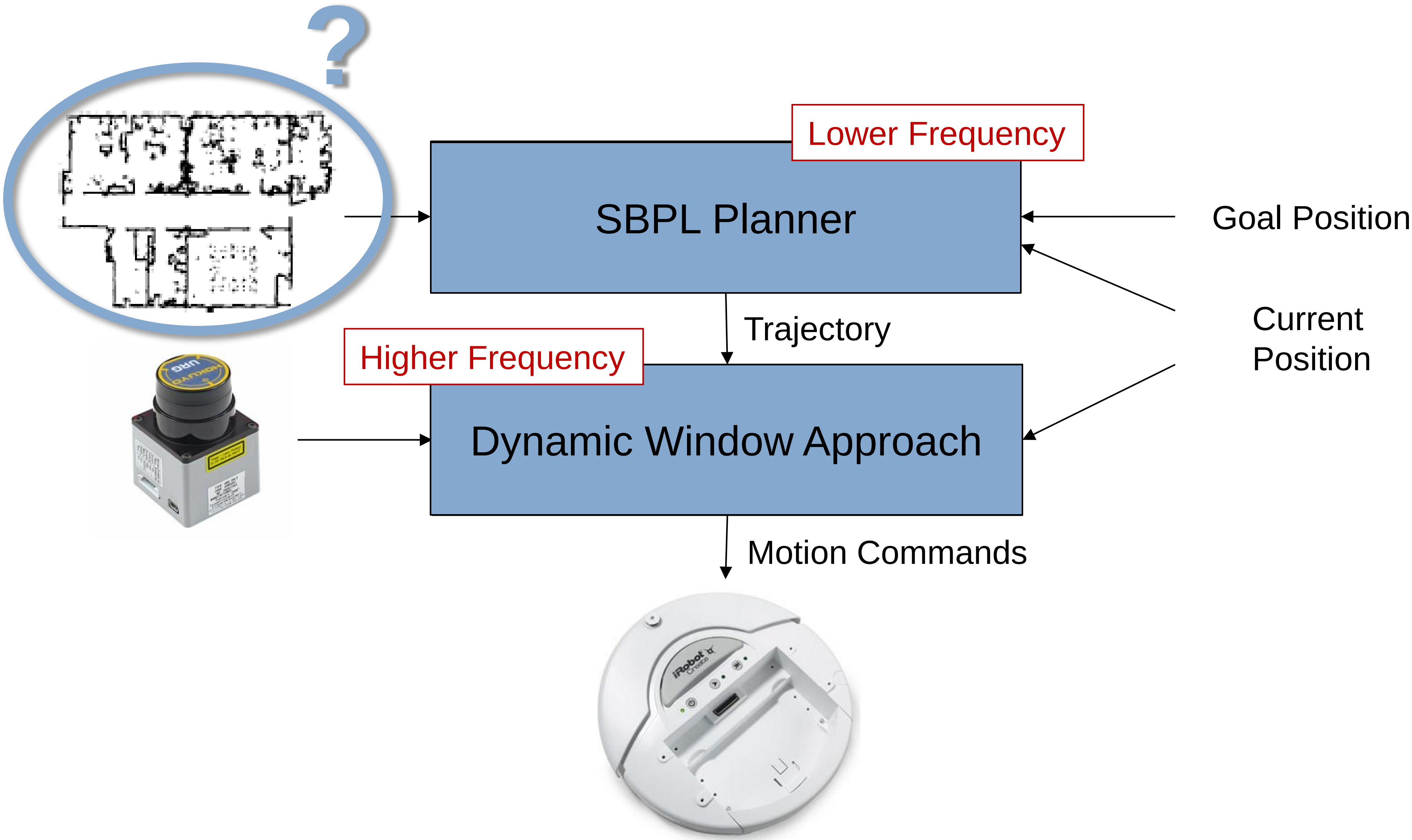
1. Discretely sample robot control space
2. For each sampled velocity, perform forward simulation from current state to predict what would happen if applied for some (short) time.
3. Evaluate (score) each trajectory resulting from the forward simulation
4. Discard illegal trajectories, i.e., those that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




What about non circular kinematics?

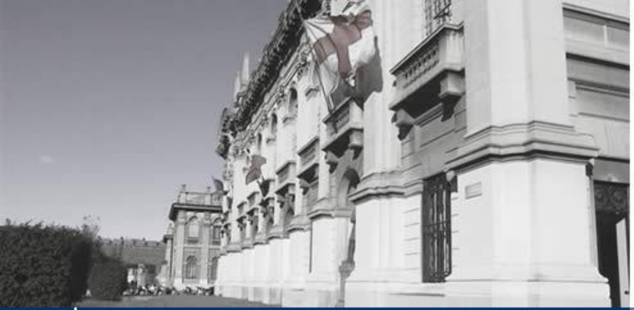
*Clothoid:* $S(x) = \int_0^x \sin(t^2)\,dt, \quad C(x) = \int_0^x \cos(t^2)\,dt.$

# Recall the Two Layered Approach





Matteo Matteucci – matteo.matteucci@polimi.it



POLITECNICO DI MILANO

# Cognitive Robotics – Robot Motion Planning

Matteo Matteucci – matteo.matteucci@polimi.it