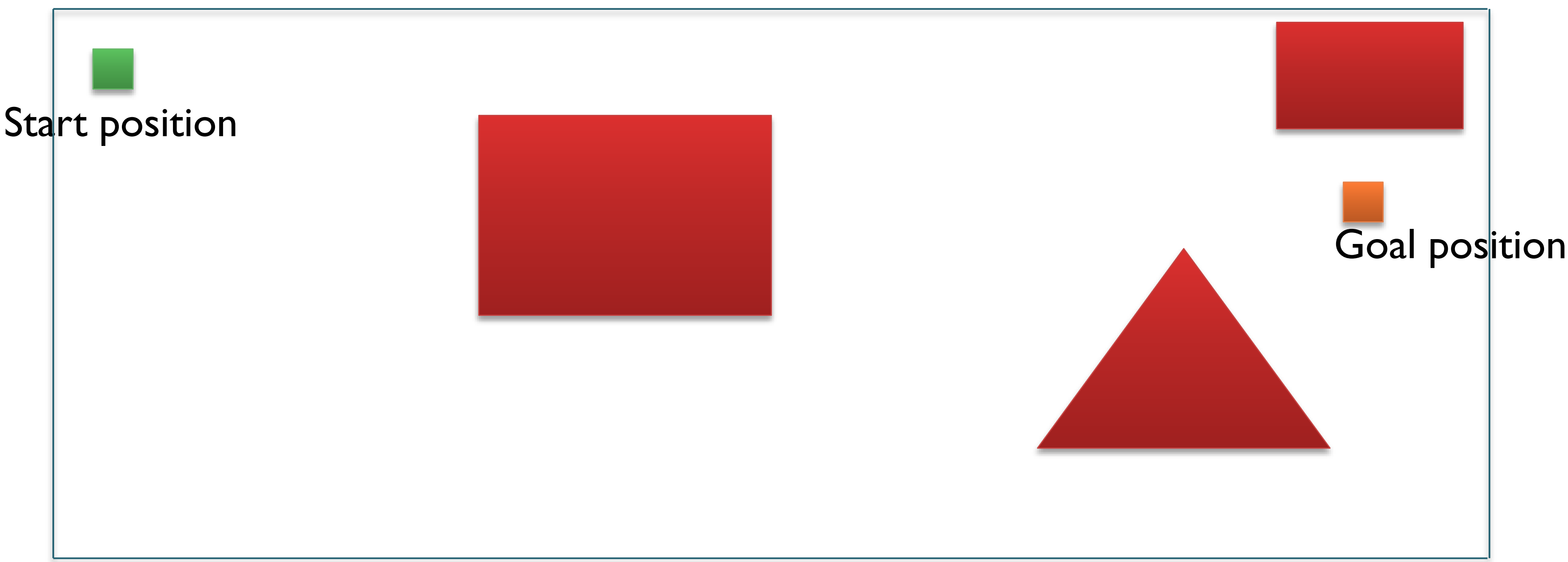
Path/Motion Planning:
An overview

Jonas Kvarnström
Automated Planning and Diagnosis Group
Department of Computer and Information Science
Linköping University

jonas.kvarnstrom@liu.se – 2018

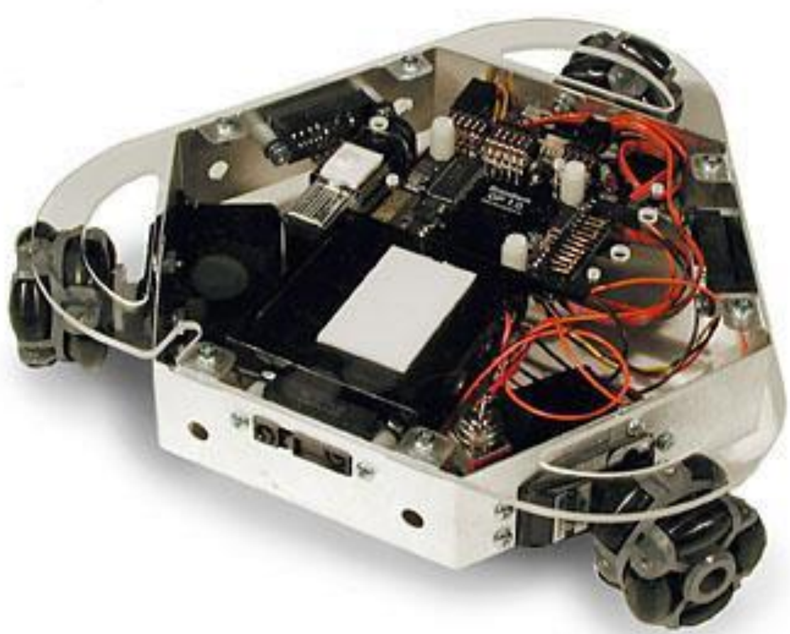
Path/Motion Planning (1)

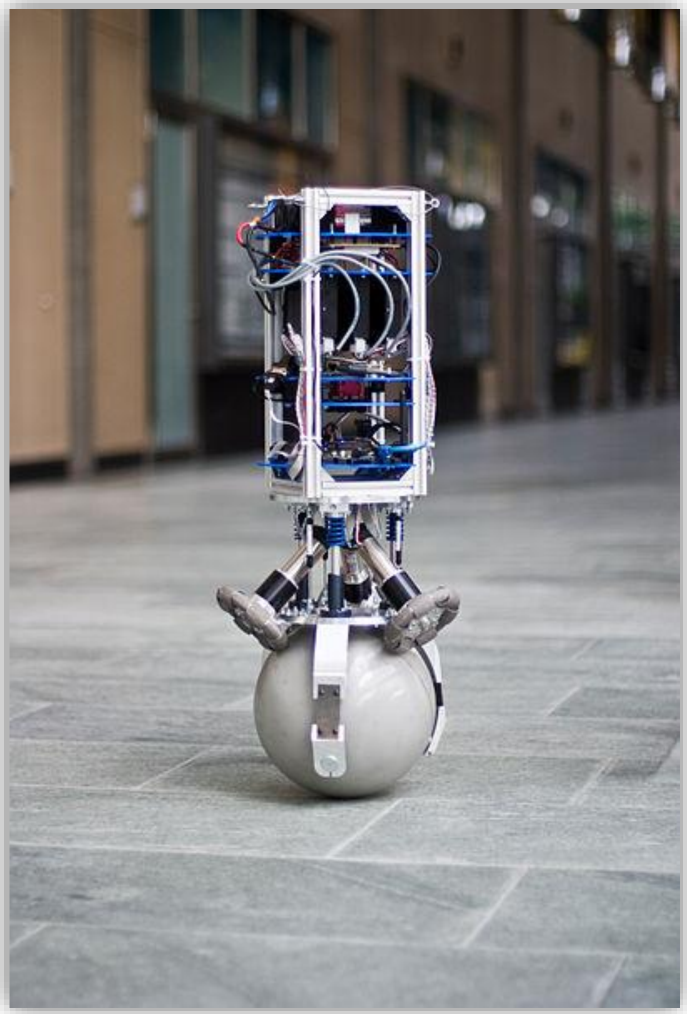
- Perhaps the *easiest* form of **path planning** / **motion planning**:
  - (1) A robot should move in **two dimensions** between start and goal
    - Avoiding known obstacles – or it would be *too* easy…



Path/Motion Planning (2)

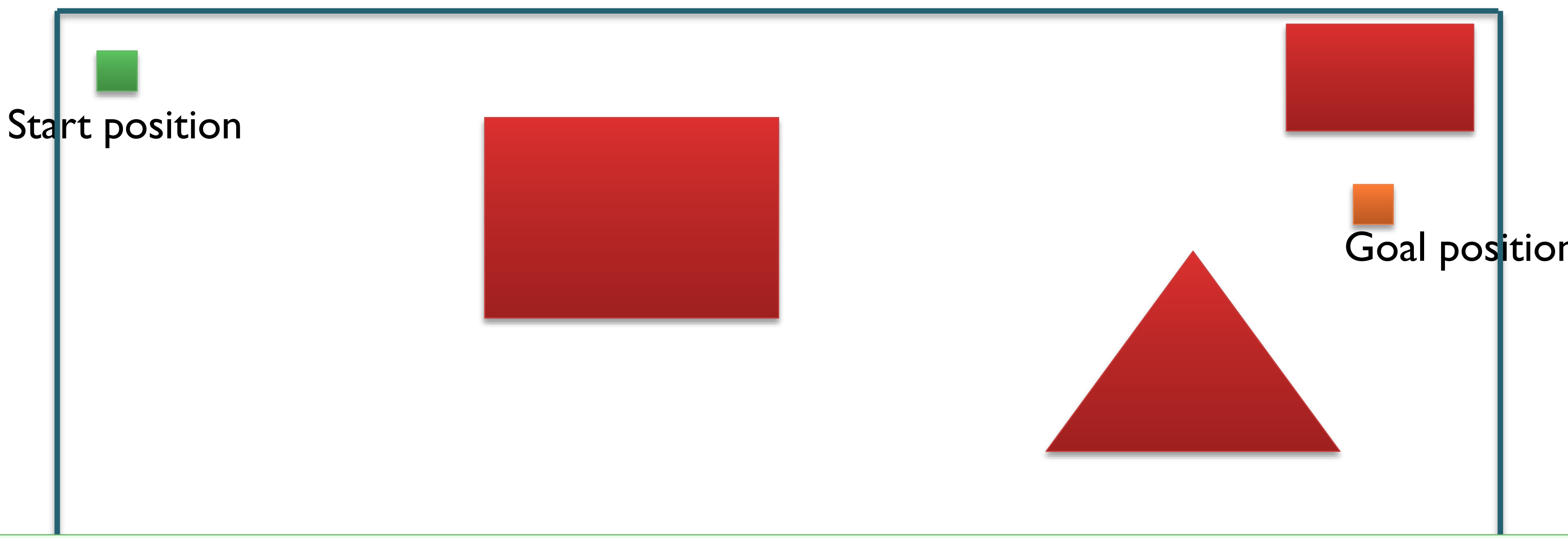
- Perhaps the *easiest* form of **path planning** / **motion planning**:
  - (2) The robot is **holonomic**
    - Informally: Can move in any direction (possibly by first rotating, then moving)





Path/Motion Planning (3)

- Problem: Generating an **optimal continuous path** is hard!
  - Common solution: Divide and conquer
    - **Discretize**: Choose a finite number of **potential waypoints** in the map
    - Assume there exists a robot-specific **local planner** to determine whether one can move **between** two such waypoints (and how)
    - Use **search algorithms** to decide which waypoints to use

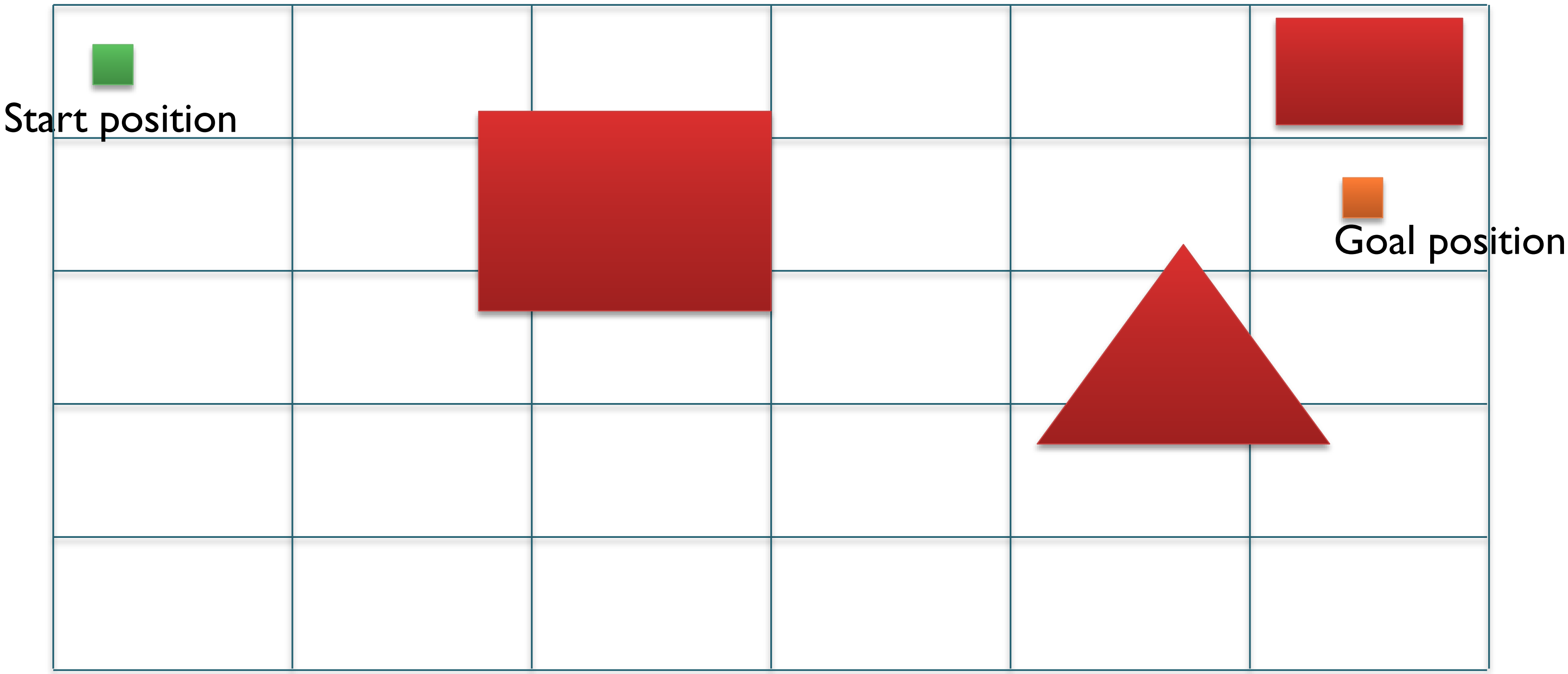


Remaining task: **choosing potential waypoints** + **finding a path** using them

Choosing Potential Waypoints: Grid-Based Methods

Regular 2D Grid

- The simplest type of discretization: A **regular grid**
  - A robot **moves** only *north, east, south* or *west*
    - Details are left to the local planner

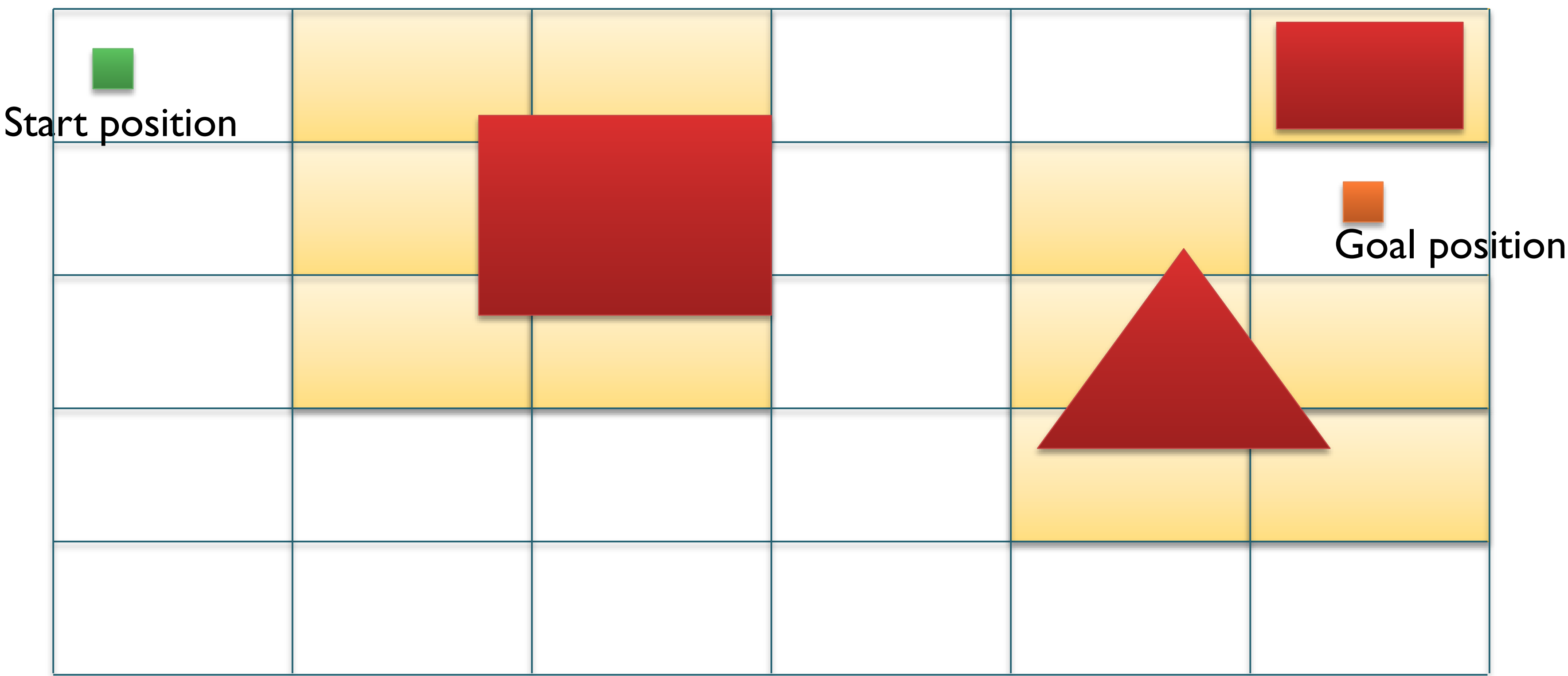


Regular 2D Grid: Real Obstacles

- Real obstacles do not correspond to square / rectangular cells…
  - But we can *cover* them with cells

Partially covered – can't be used

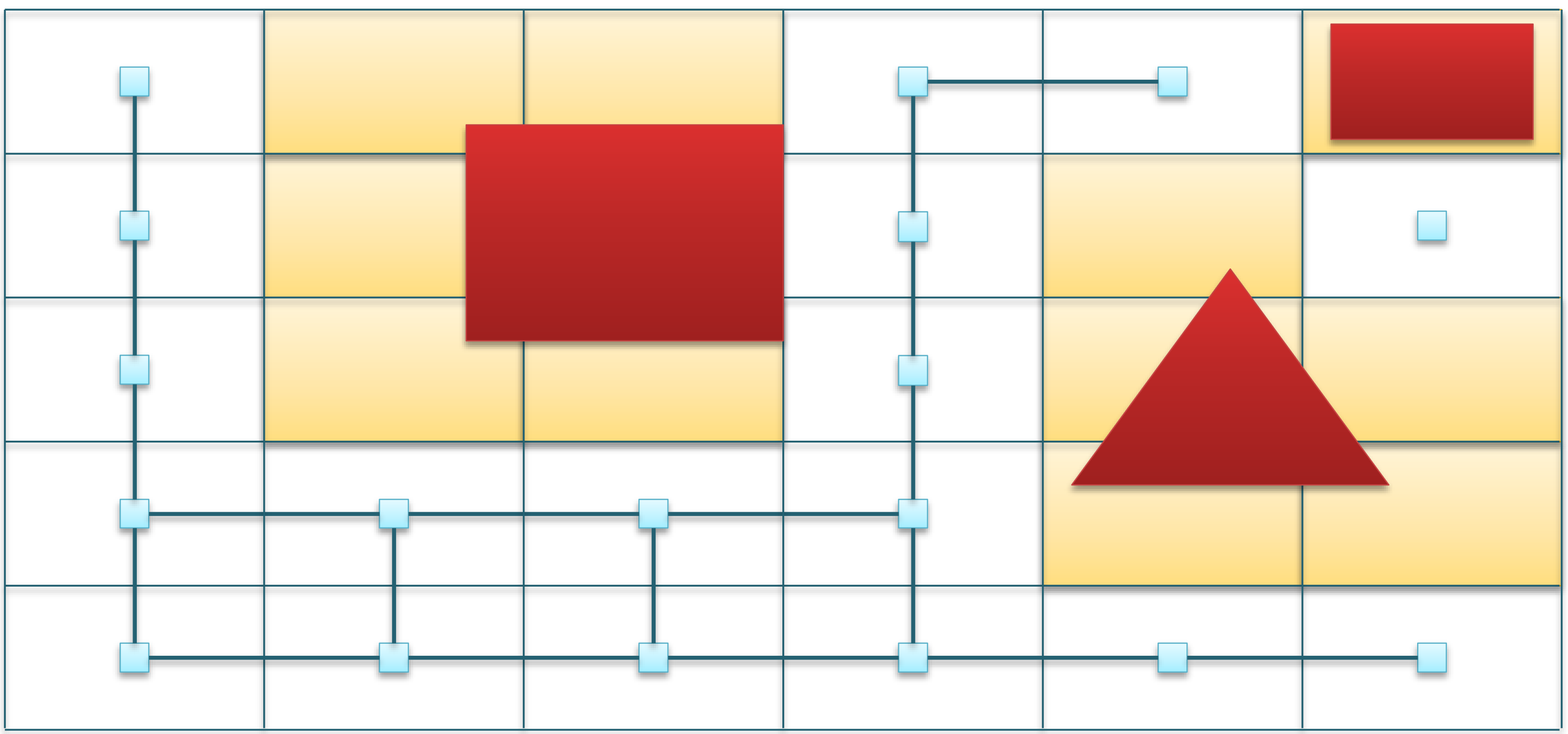
Obstacle



Regular 2D Grid: Discrete Graph

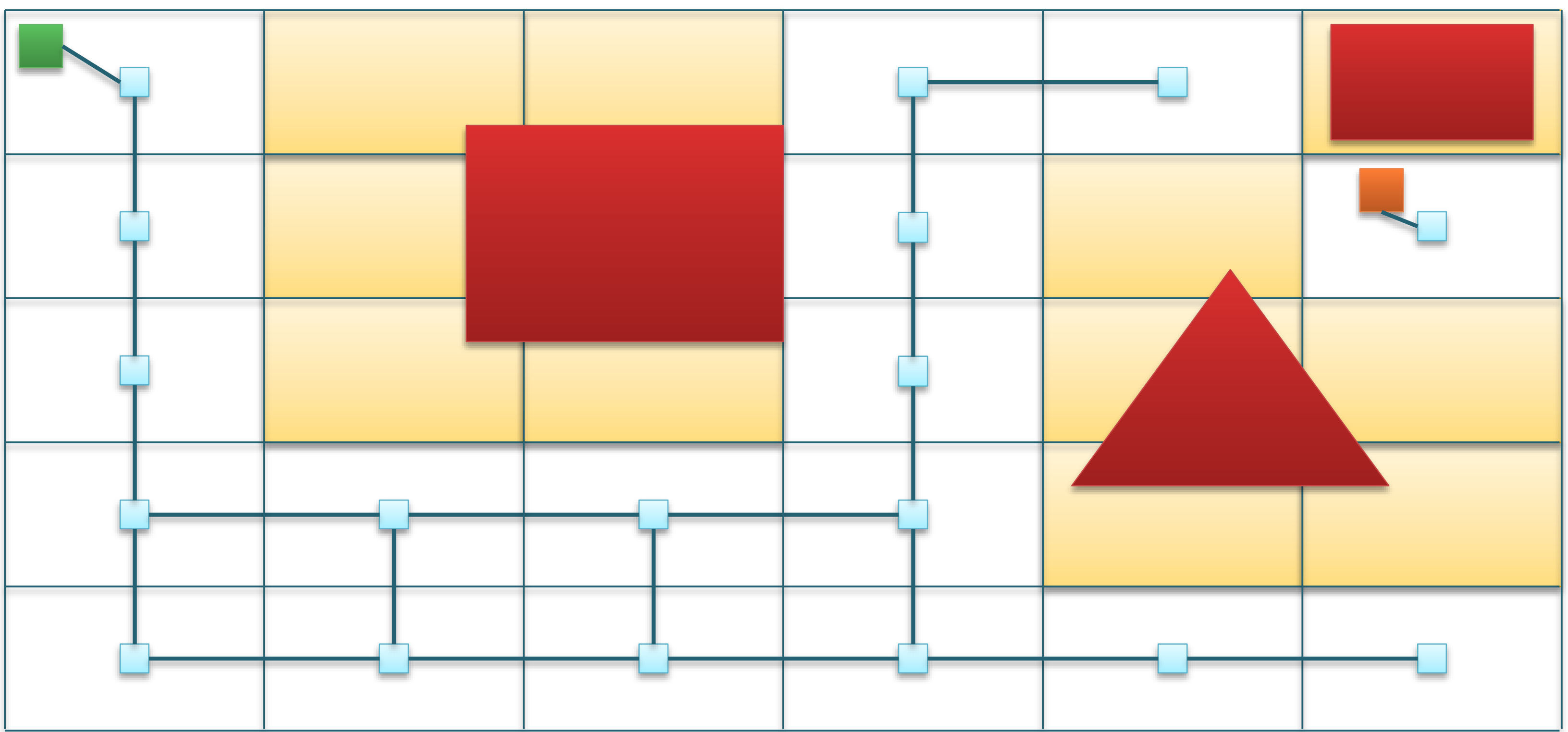
- View the grid **implicitly** as a **discrete graph**
  - Assume the **local** path planner can take us between **any neighboring cells**
    - Between blue nodes
    - No obstacles in the way
    - Sufficient free space to deal with non-holonomic constraints

Simple (trivial?) local planner
Discrete search handles the rest



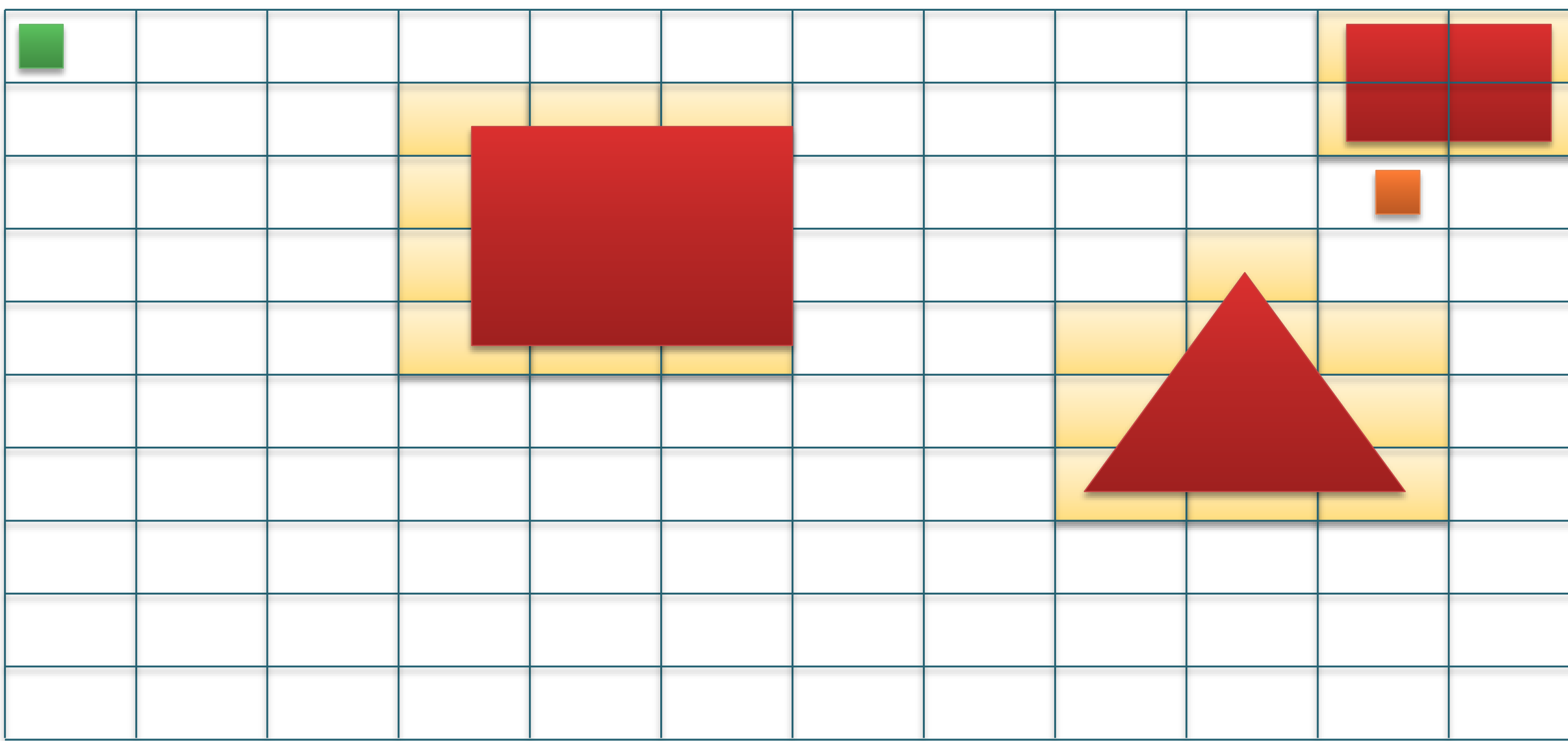
Regular 2D Grid: Discrete Graph (2)

- Connect start/goal configurations to the nodes in their cells
  - Within a cell ➔ no obstacles ➔ can plan a path using local planner
  - Here, the goal is unreachable…



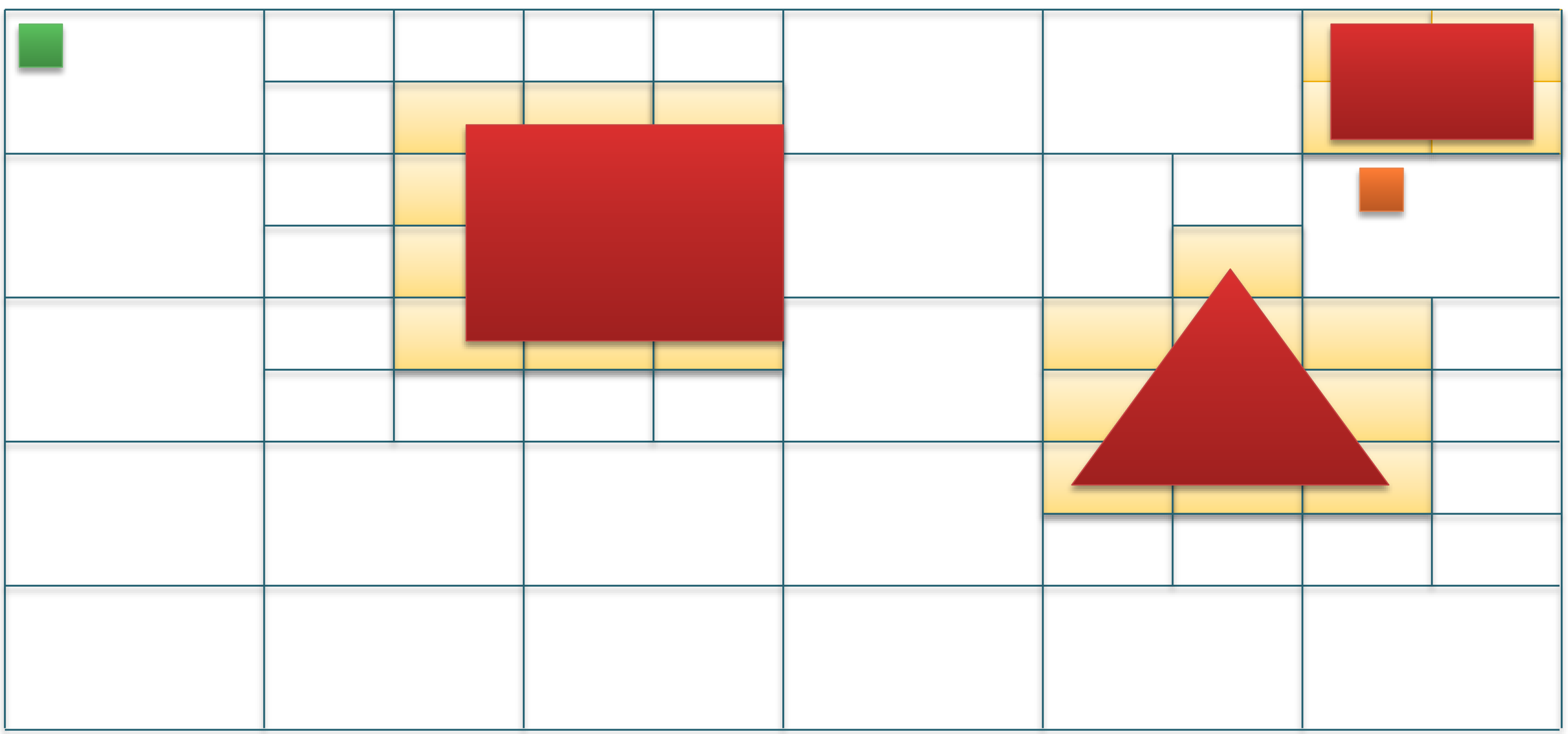
Regular 2D Grid: Grid Density

- **<u>Grid density</u>** matters!
  - Here: 4 times as many grid cells
  - Better approximation of the true obstacles, but many more nodes to search



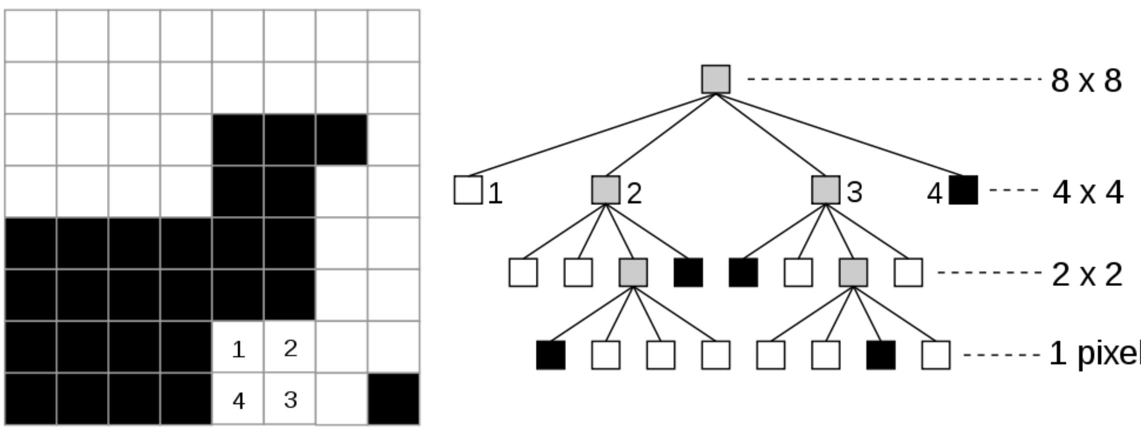
Non-Regular Grids

- Alternative: Use **non-regular** grids
  - For example, denser around obstacles
  - (Or even non-rectangular cells)



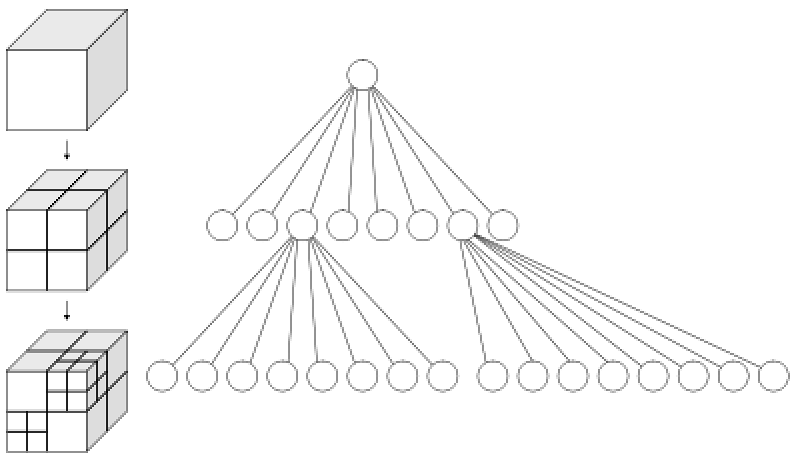
Grid Representations

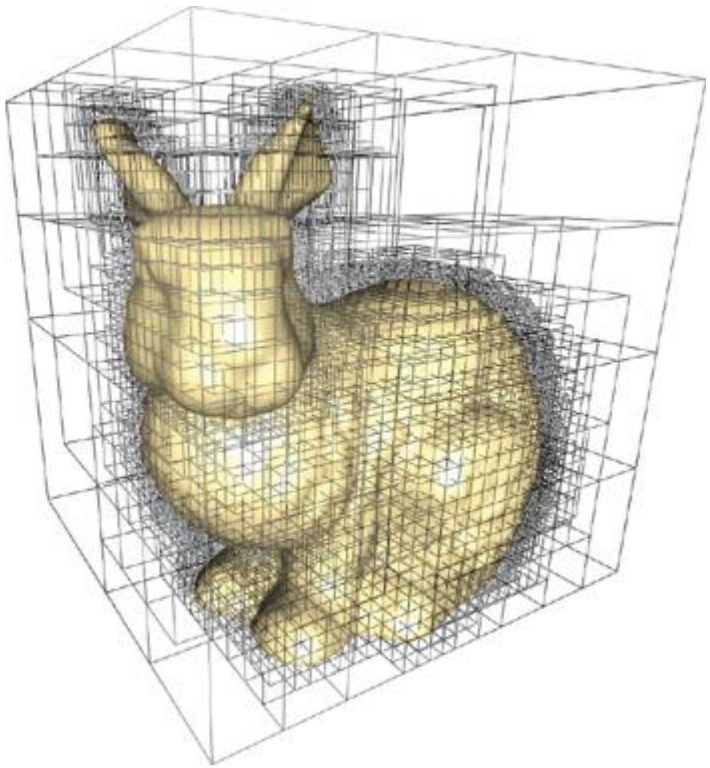
- Space-efficient data structure: **quadtree**
  - Each node keeps track of:
    - Whether it is *completely* covered, *partially* covered or *non-covered*
  - Each non-leaf node has exactly four children



Grid Representations

- Can be generalized to 3D (octree), …





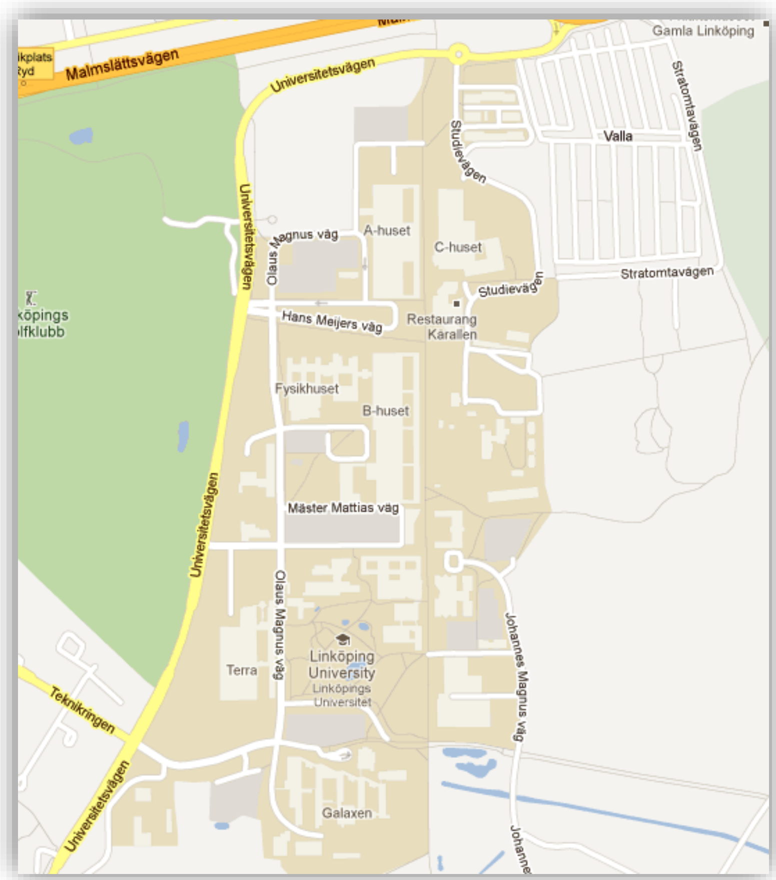
Choosing Potential Waypoints: Geometry-Based Methods

Regular 2D Grid: Grid Density

- Grid-based methods can result in many nodes
  - Even with efficient representation, *searching* the graph takes time
  - Alternative idea: **Place** nodes **depending** on obstacles

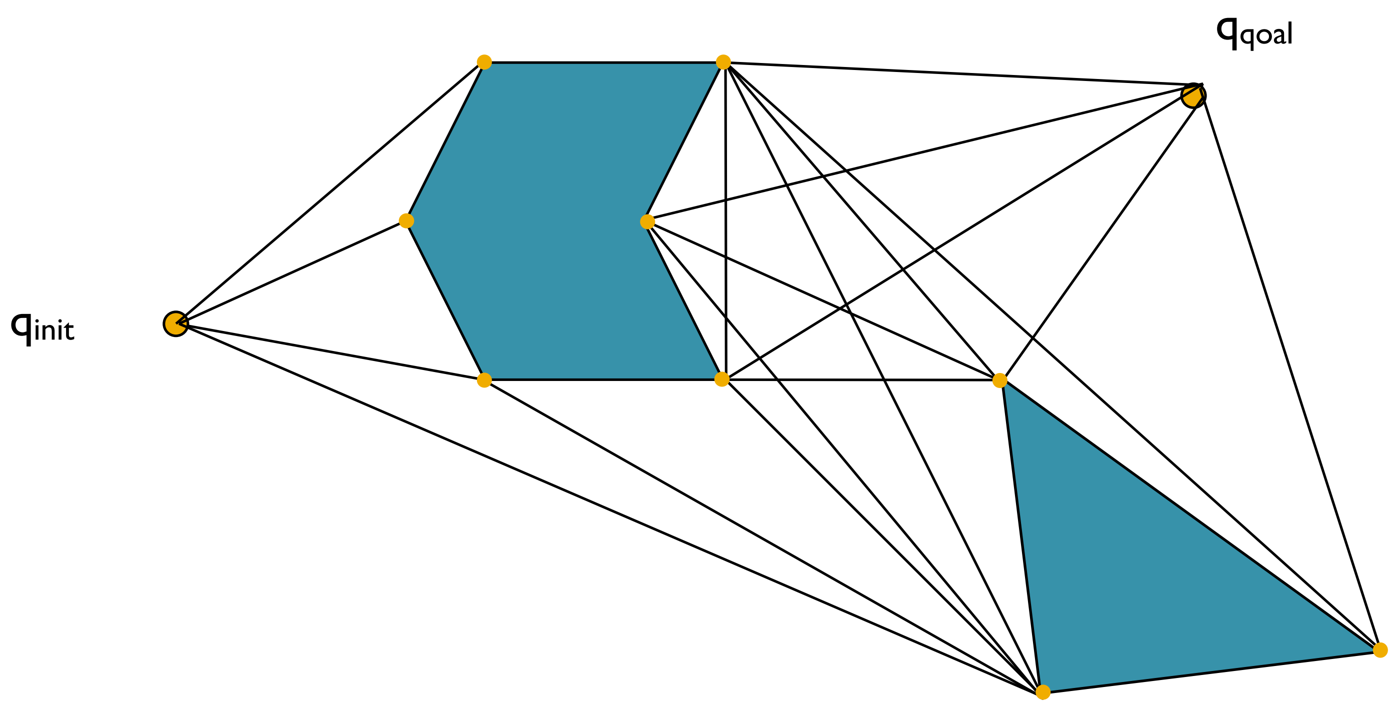
- Simple case: Known road map
  - Model all non-road areas as obstacles, then add a dense **grid**?

  - Or **place** a node in each **intersection**?



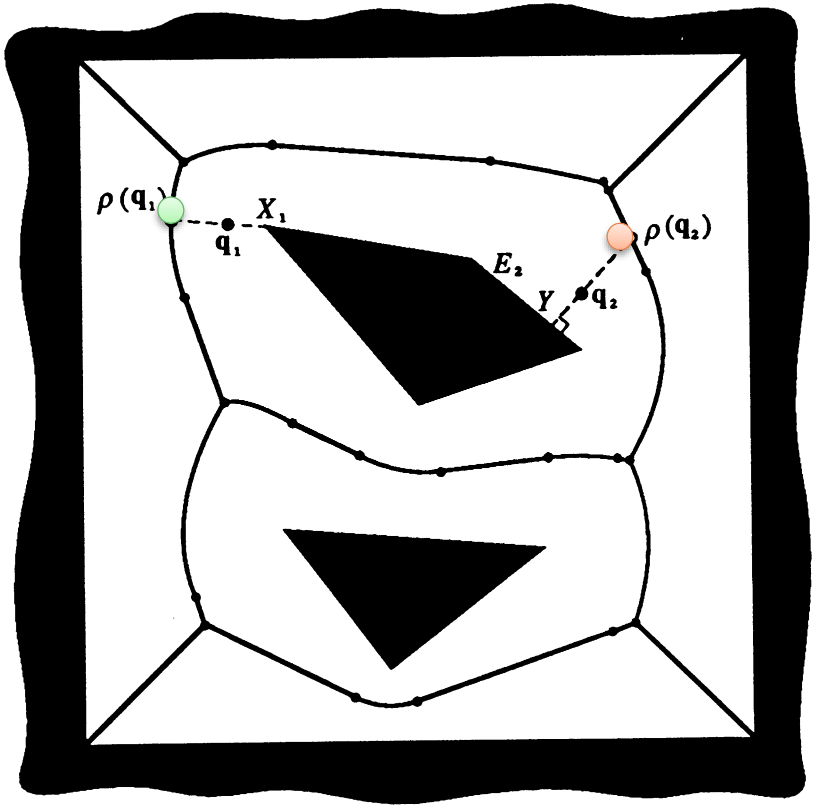
Visibility Graphs

- **Visibility graphs**
  - Applicable to **simple polygons** – straight sides *without intersections*
    - **Nodes** at all polygon corners
    - **Edges** wherever a pair of nodes can be connected using the local planner
  - Mainly interesting in 2D
    - Optimal in 2D, not in 3D



Voronoi Diagrams

- **<u>Voronoi</u>** diagrams
  - Find all points that have the same distance to two or more obstacles
    - Maximizes **<u>clearance</u>** (free distance to the nearest obstacle)
  - Creates unnecessary **<u>detours</u>**
  - Mainly interesting in 2D – does not scale well



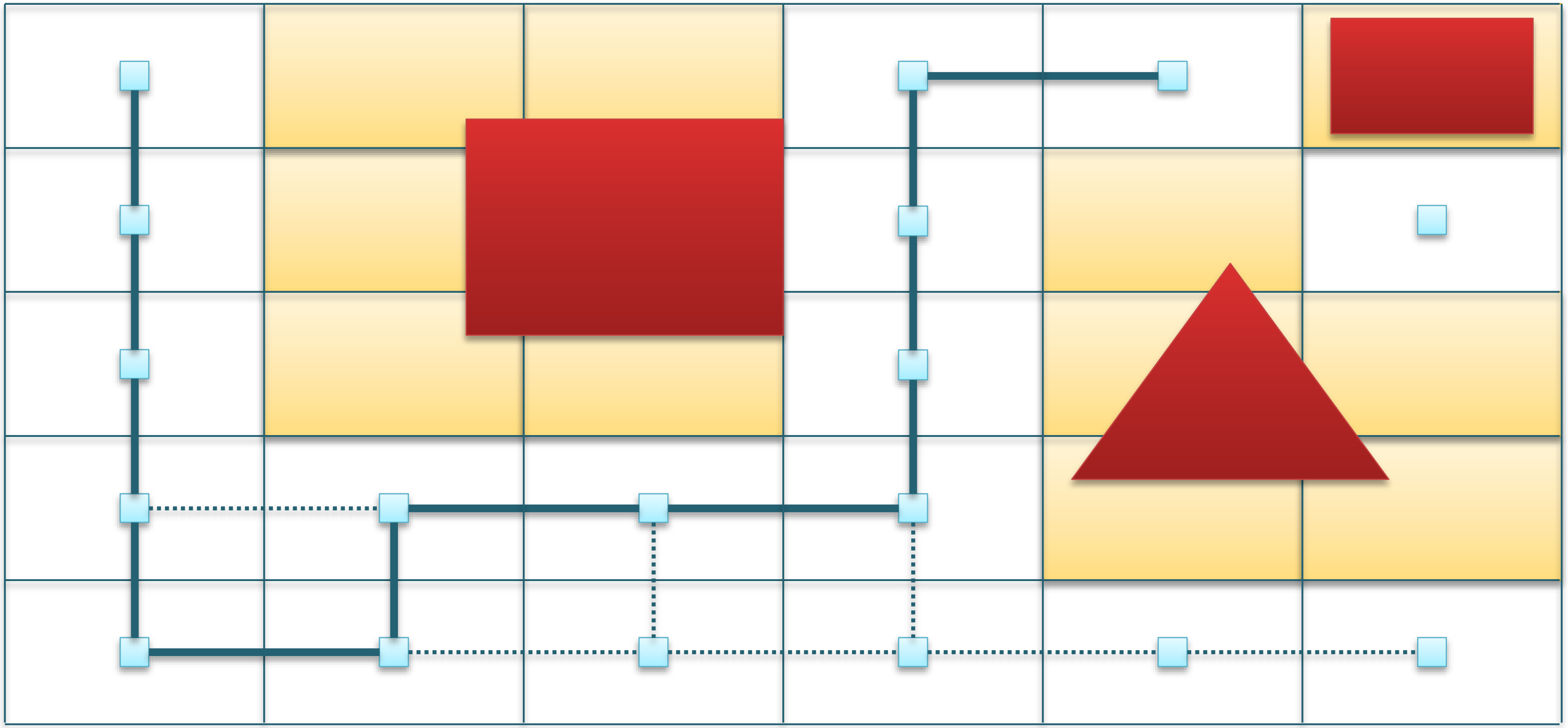
Complex Motion Planning Problems

Introduction

- So far, we implicitly assumed:
  - If we can **draw a line** between two waypoints, the robot can **move** between the waypoints

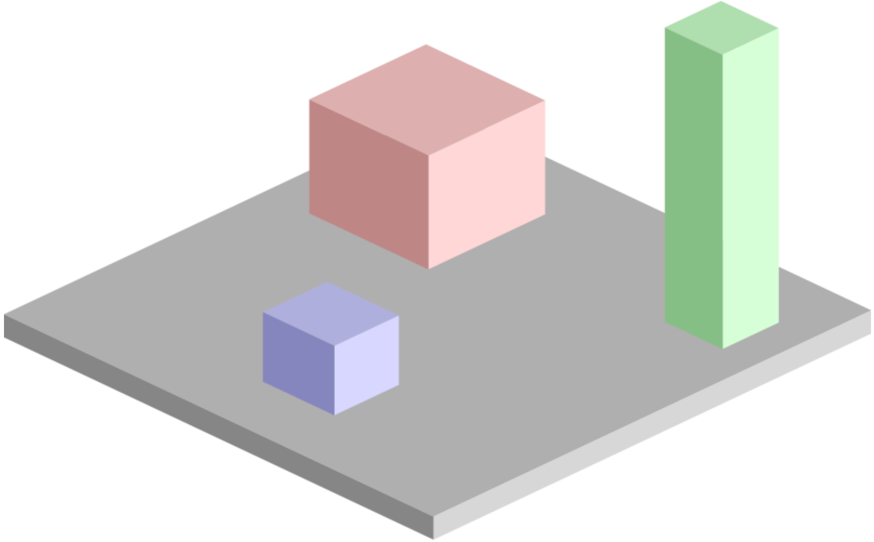
We need to introduce some new concepts…

- But: How does an airplane fly this path?



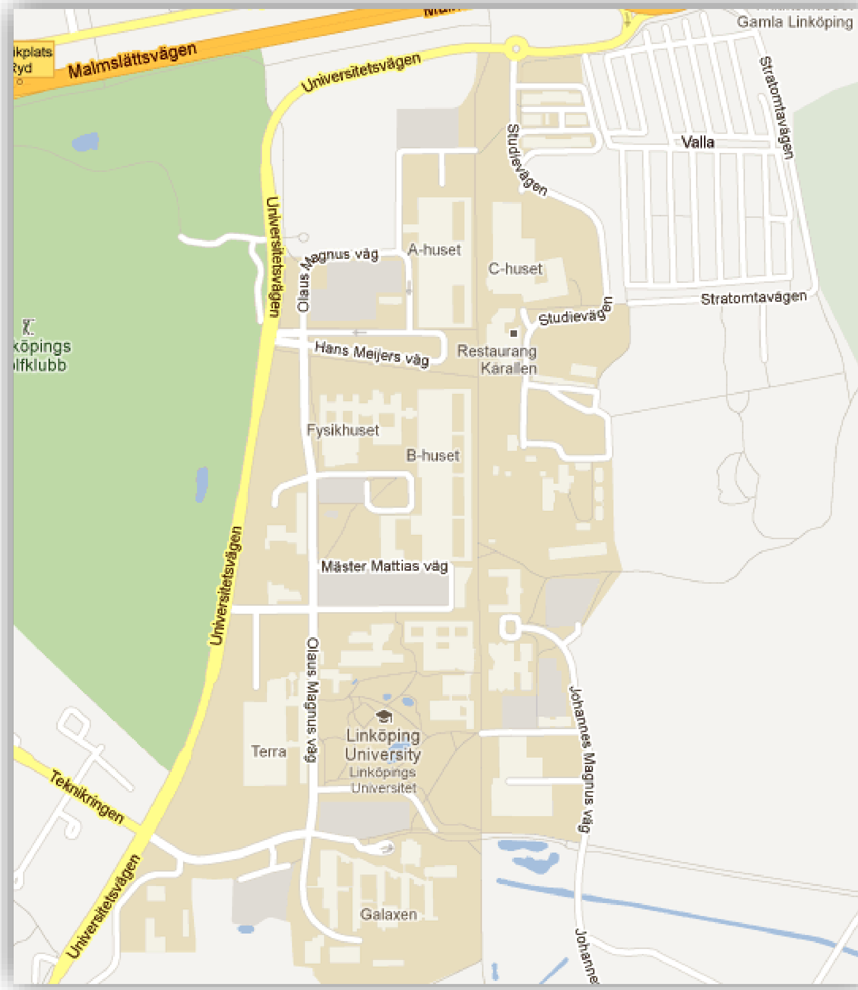
Workspace (1)

- The **workspace** is (1) the **physical space** in which we work…
  - 3 physical dimensions, 3-dimensional coordinates, 3-dimensional obstacles
    - Need full 3D geometry to determine how the helicopter can move



Workspace (2)

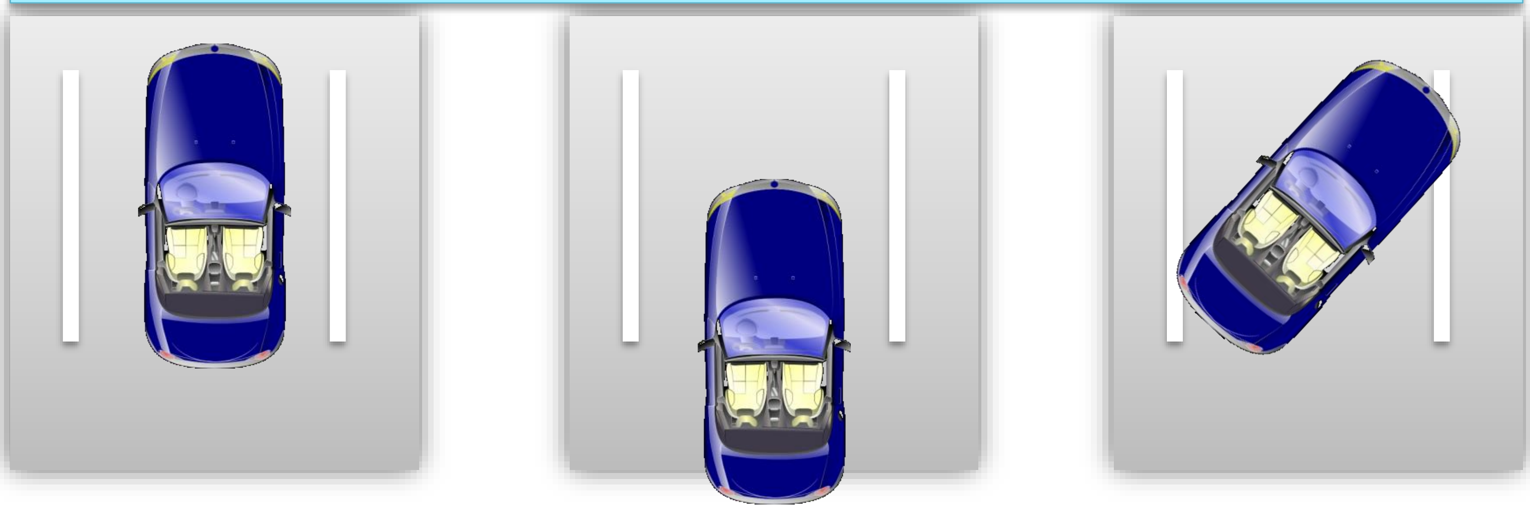
- … or (2) a **2D projection**, in case this is sufficient
  - For a car:
    - Can describe position, rotation in 2D
    - Can describe obstacles in 2D
  - ➔ Workspace can be 2D
    - **Still represents physical locations**



Configuration Space

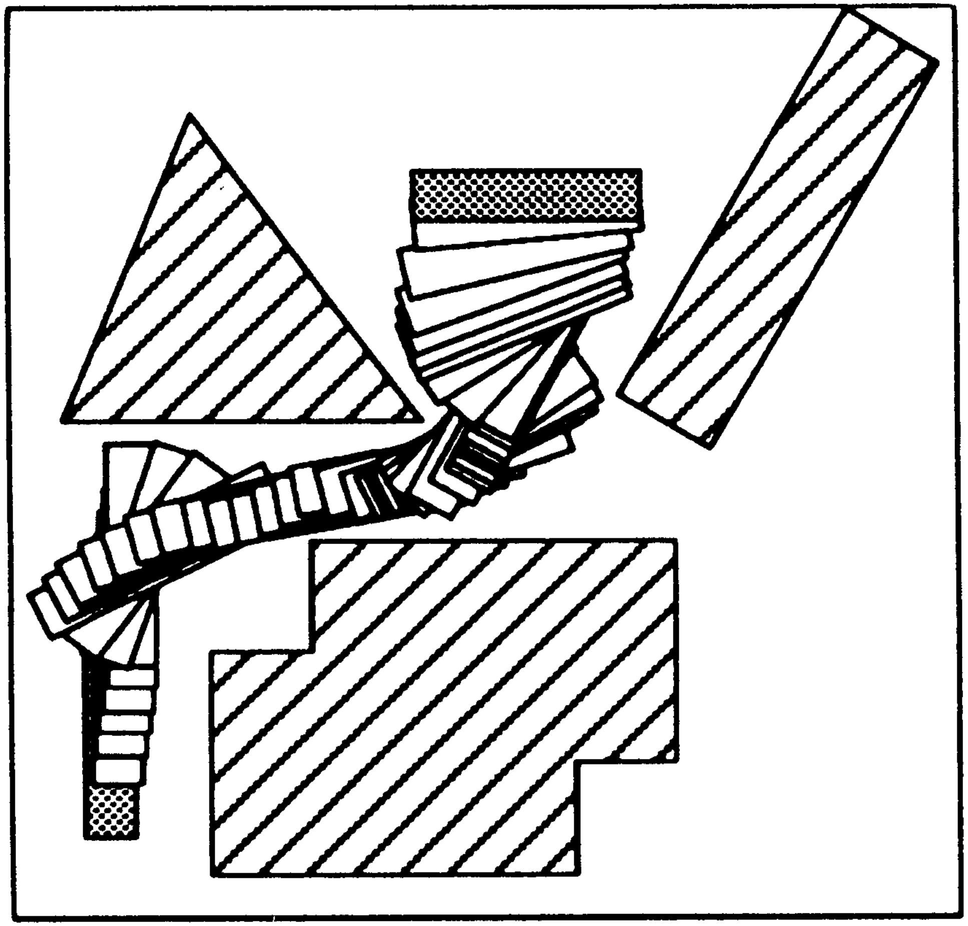
- Even a car has 3 **physical degrees of freedom** (DOF)!
  - The **configuration space** of the car
    - **Location** in the plane (x/y),
    - **Angle** (θ)
  - Each DOF is essential!
    - As part of the *goal* – park at the correct angle
    - As part of the *solution* – must turn the car to get through narrow passages

Motion planning takes place in configuration space:
How do I get from $(200, 200, 12°)$ to $(800, 400, 90°)$?



The Ladder Problem

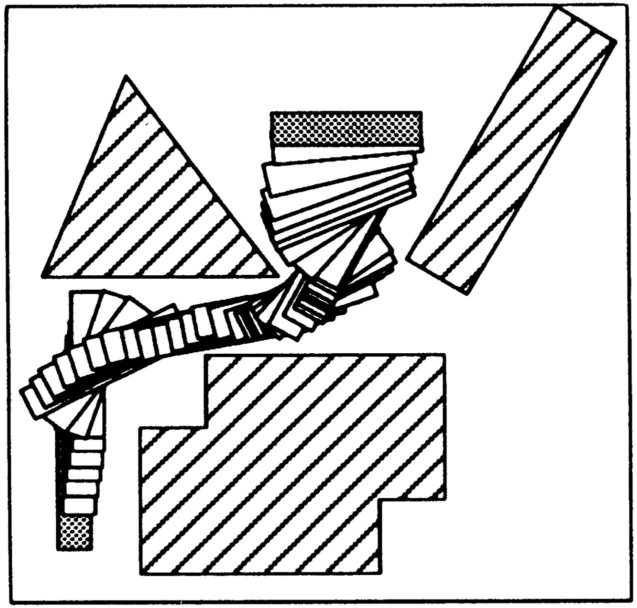
- The **ladder problem** is similar
  - Move a ladder in a 2D workspace , with 3 **physical** DOF
  - Configuration:
    - **Location** in the plane (x/y),
    - **Angle** (θ)



- Again, each DOF is essential:
  - As part of the ***goal***
    - We want the ladder to end up at a specific angle
  - As part of the ***solution***
    - We need to turn the ladder to get it past the obstacles

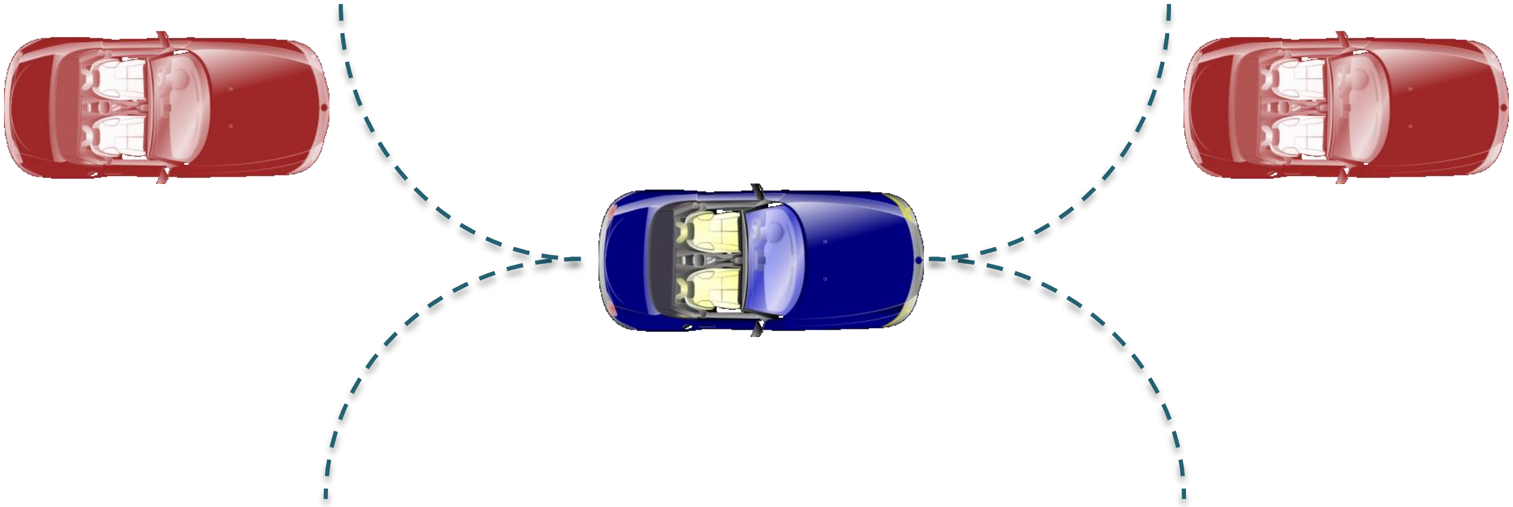
The Ladder Problem: Controllable DOF

- For ladders, each physical DOF is directly controllable!
  - You can:
    - Change x (translate sideways)
    - Change y (translate up/down)
    - Change angle (rotate in place)
  - Therefore:
    - If you want to get from $(200, 200, 12°)$ to $(800, 400, 90°)$, any path **connecting** these 3D points and **going through** free configuration space is sufficient
  - The ladder is **holonomic**!
    - Controllable DOF >= physical DOF



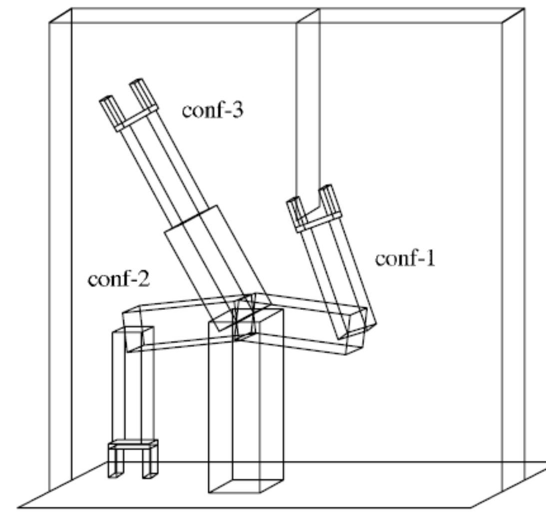
Controllable Degrees of Freedom

- For cars, we can **control** two DOF:
  - Acceleration/breaking
  - Turning (limited)
- In this **parallel parking** example:
  - There **is** free space between current and desired configurations
    - But we can't slide in sideways!
  - Fewer controllable DOF than physical DOF ➔ non-holonomic
    - *Limits possible curves in 3D configuration space!*

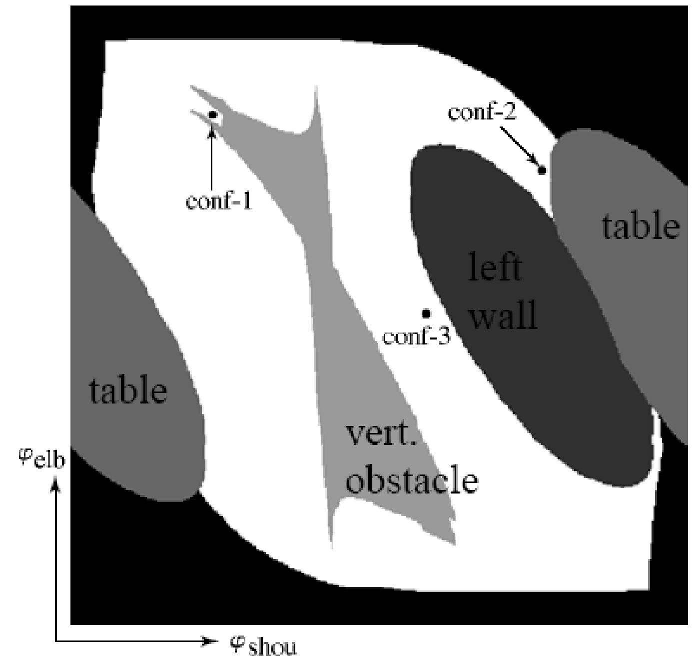


Work Space, Configuration Space

- Summary of important concepts:
  - **Work space**: The physical space in which you move
    - 3-dimensional for this robot arm

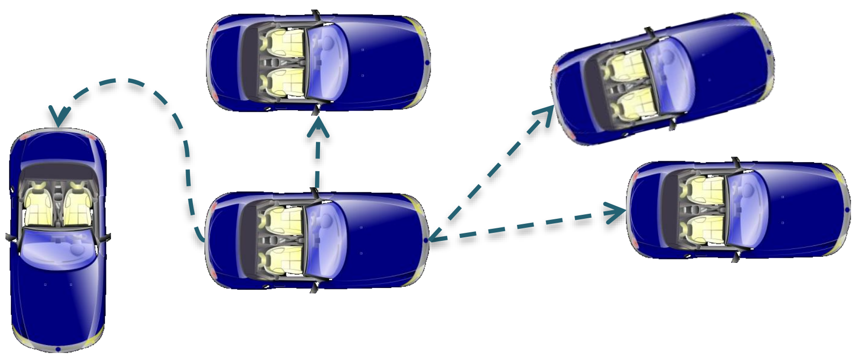


  - **Configuration space**:
    The set of possible configurations of the robot
    - Usually **continuous**
    - Often **many-dimensional**
      (one dimension per physical DOF)
    - Will often be **visualized** in 2D for clarity
  - We have to search
    in the **configuration space**!
    - Connect *configurations*, not *waypoints*

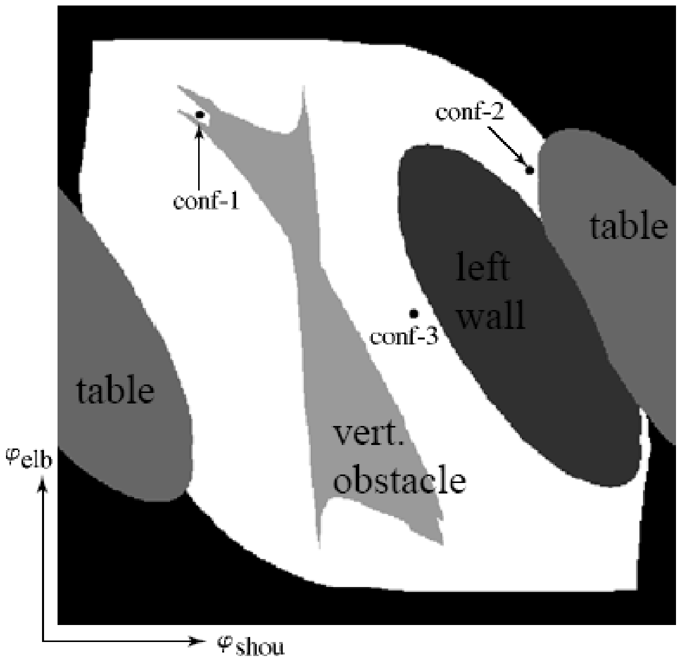


Searching the Configuration Space

- Divide and Conquer!
  - **Local** path planner
    - Determines whether two configurations can be connected with a path, and how
    - Considers vehicle-specific constraints



  - **High-level** path planner
    - Generates configurations
    - Uses plug-in local planner to determine if the configurations can be connected
    - For each specific problem, uses search to determine which intermediate configurations to use



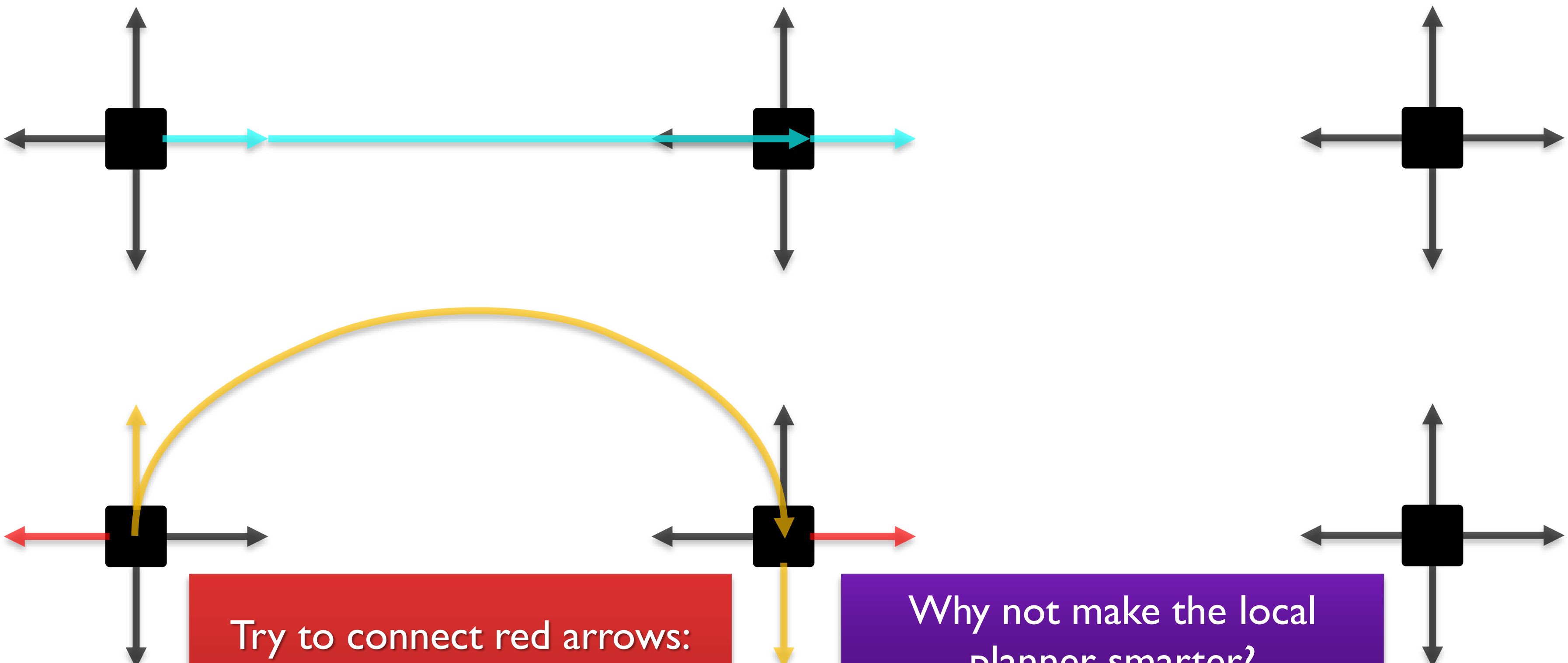
Low-Dimensional Problems

- In low-dimensional problems:
  - The high-level planner *could* use a grid
    - Car: 3-dim configuration space
    - Example: 4 angles considered per spatial location

(0, 0, 0°) (0, 0, 90°) (0, 0, 180°) (0, 0, 270°)	(1, 0, 0°) (1, 0, 90°) (1, 0, 180°) (1, 0, 270°)	(2, 0, 0°) (2, 0, 90°) (2, 0, 180°) (2, 0, 270°)
(0, 1, 0°) (0, 1, 90°) (0, 1, 180°) (0, 1, 270°)	(1, 1, 0°) (1, 1, 90°) (1, 1, 180°) (1, 1, 270°)	(2, 1, 0°) (2, 1, 90°) (2, 1, 180°) (2, 1, 270°)

Local Planner (1)

- Ask local planner: "Can I connect these configurations"?



Try to connect red arrows:

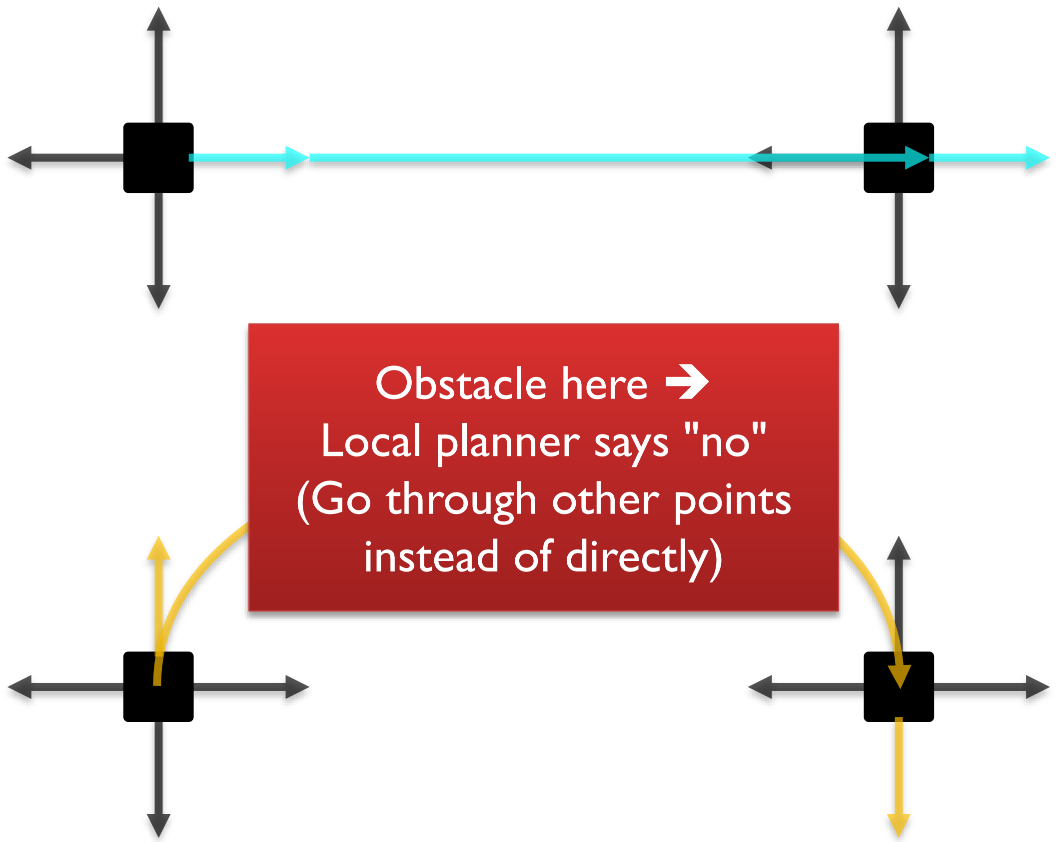
The local planner might say "Sorry, too complex"
➔ have to go through intermediate configs…

Why not make the local planner smarter?

Divide and conquer:
Local planner should be *fast*, the rest is handled through the high-level planner

Local Planner (2)

- Local planner also considers obstacles



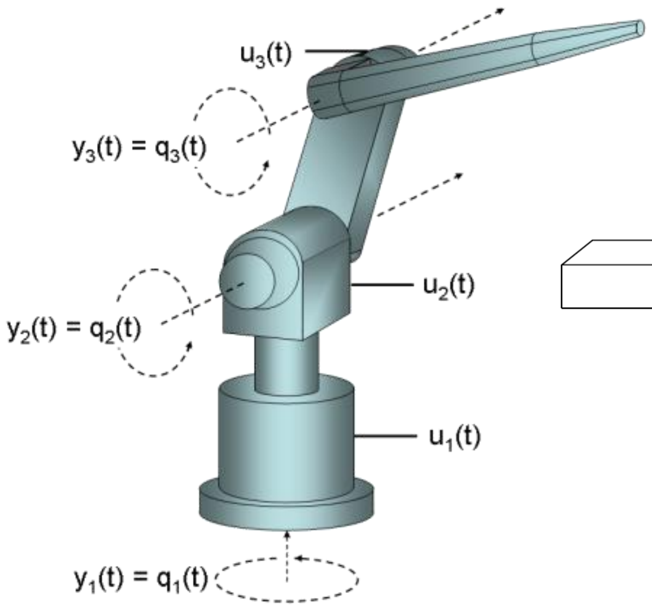
High-Dimensional Problems

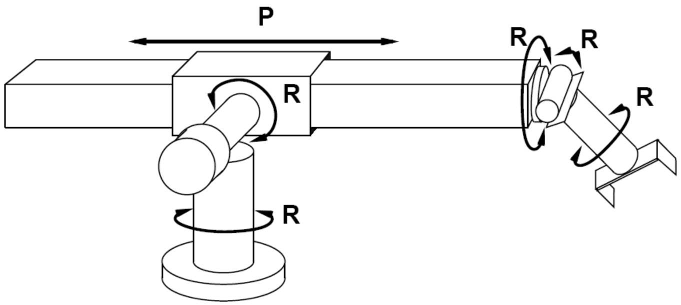
- For an **aircraft**, a configuration could consist of:
  - **location** in 3D space ($x/y/z$)
  - **pitch angle**
  - **yaw angle**
  - **roll angle**

- A **path** is:
  - a continuous **curve** in 6-dimensional configuration space
    avoiding **obstacles**
    and obeying **constraints** on how the aircraft can turn
    - Can make tighter turns at low speed
    - Can't fly at arbitrary pitch angles
    - …

High-Dimensional Problems (2)

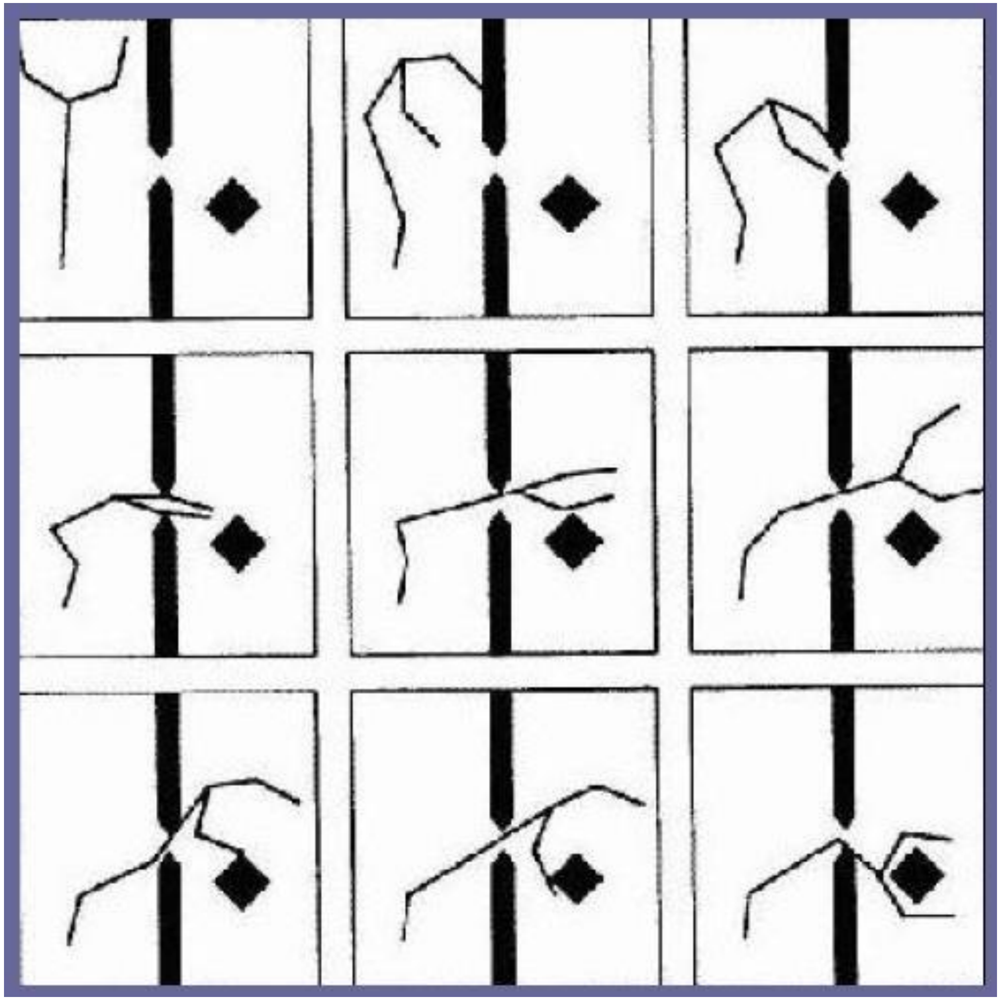
- For a **robot arm**, a configuration could consist of:
  - The position / angle of each joint
- A **path** is a continuous **curve** in n-dimensional configuration space (all joints move continuously to new positions, without "jumping"), avoiding **obstacles** and obeying **constraints** on joint endpoints etc.
- Typical goal: Reach inside the car you are painting / welding, without colliding with the car itself





High-Dimensional Problems (3)

- Moving in tight spaces, again…

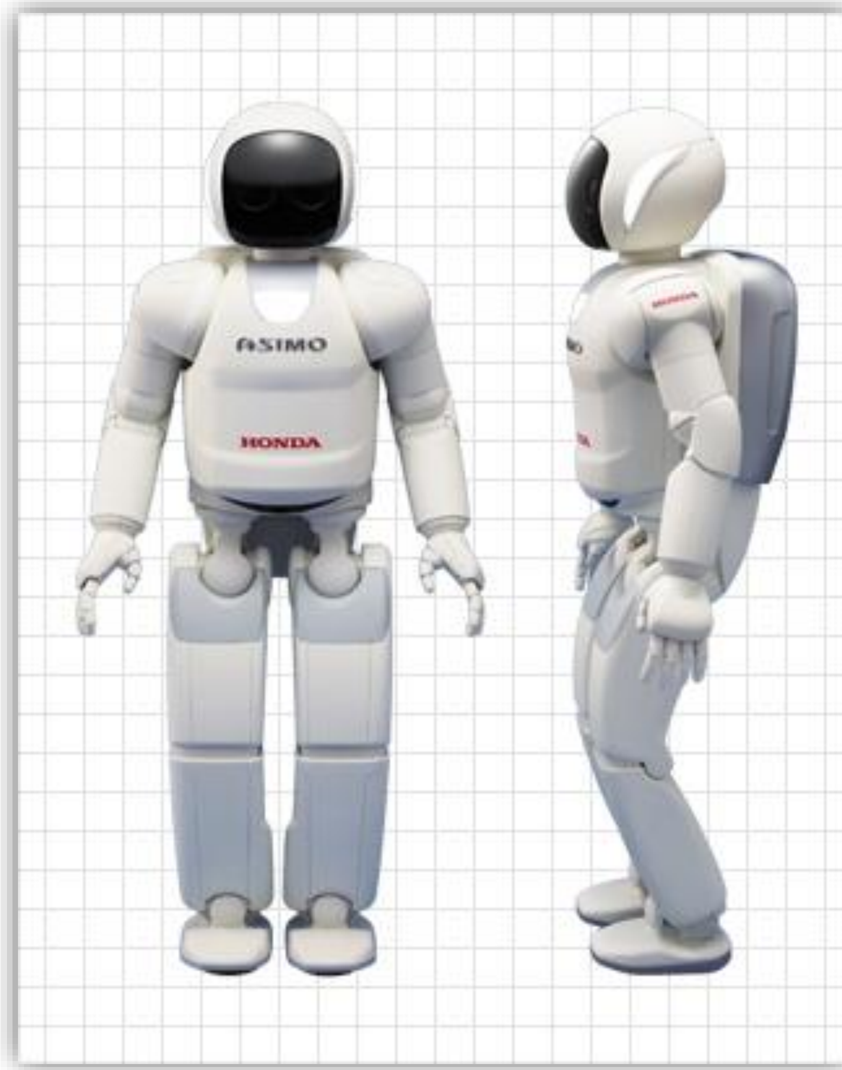


High-Dimensional Problems (4)

- For a **humanoid robot**, a configuration could consist of:
  - Position in x/y space
  - The position of each joint
- The Nao robot:
  - 14, 21 or 25 degrees of freedom depending on model
  - Up to 25-dimensional motion planning!
- Grid methods generally do not scale
  - 25-dimensional configuration space, with 1000 cells in each direction: 10^{75} cells…

High-Dimensional Problems (5)

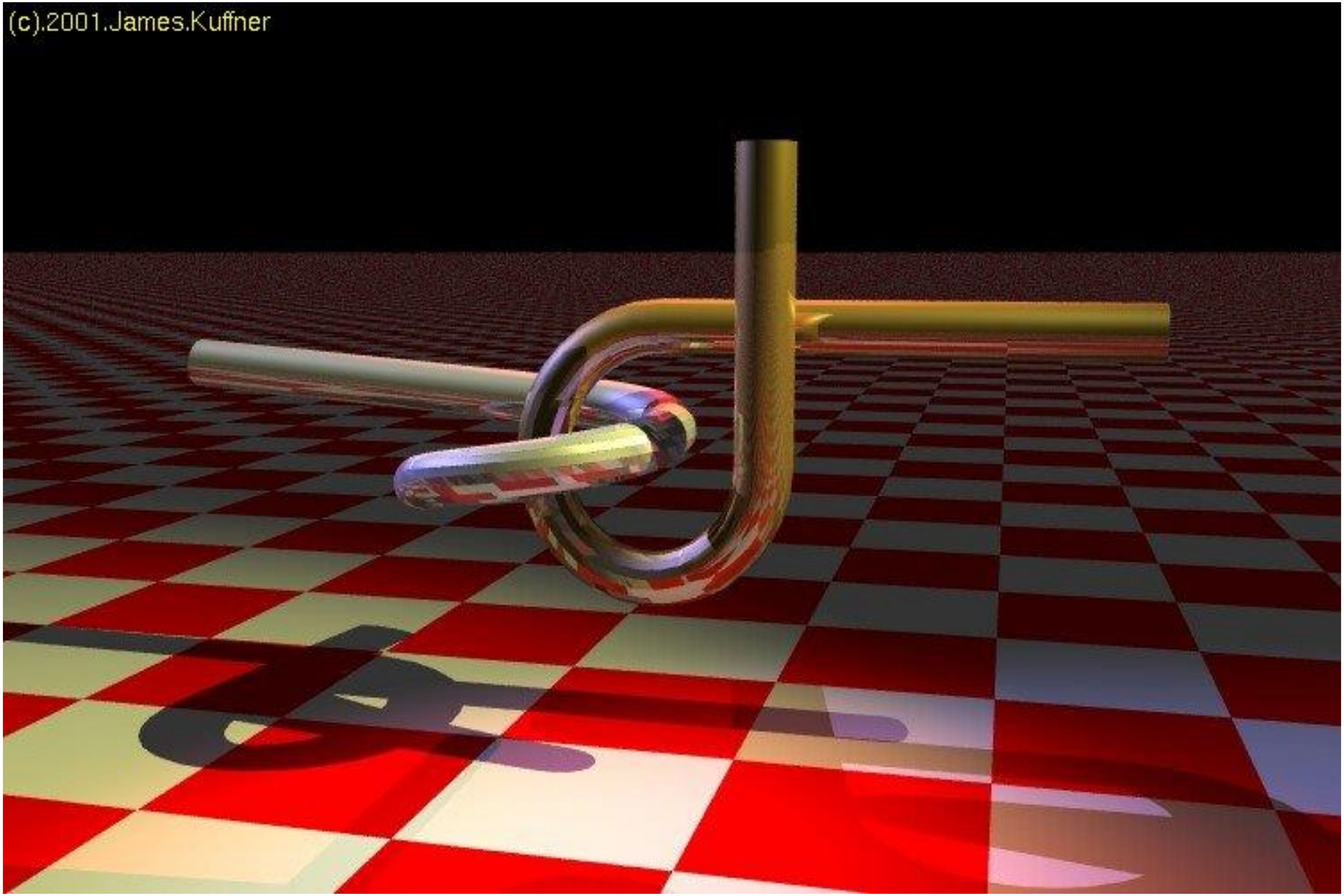
- Honda Asimo: 57 DOF



We can often omit some DOF from planning…

But then we don't use the robot's full capabilities!

Alpha Puzzle: Narrow Passages

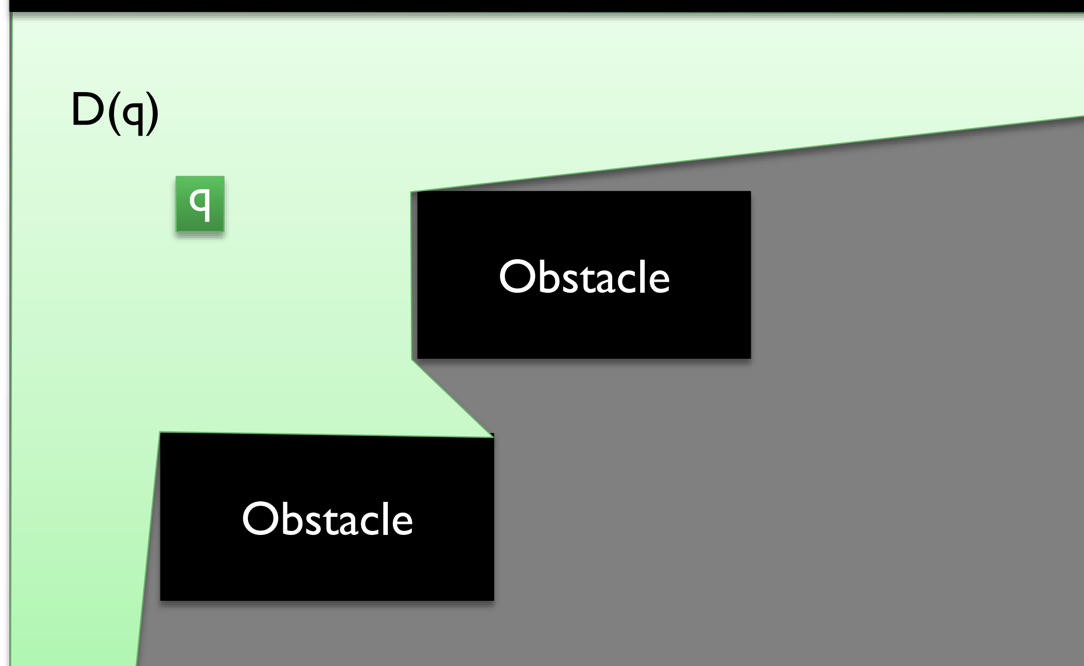


Choosing Potential Configurations: Probabilistic Methods

Preliminaries: Coverage Domain

- Given a **configuration** q in the **free config space**:
  - A particular **local planner** can connect it to a *set of other configs*
  - Called the **coverage domain** $D(q)$ – generally an infinite set

Example: Simple *2D planning*,
local planner uses *straight lines*…

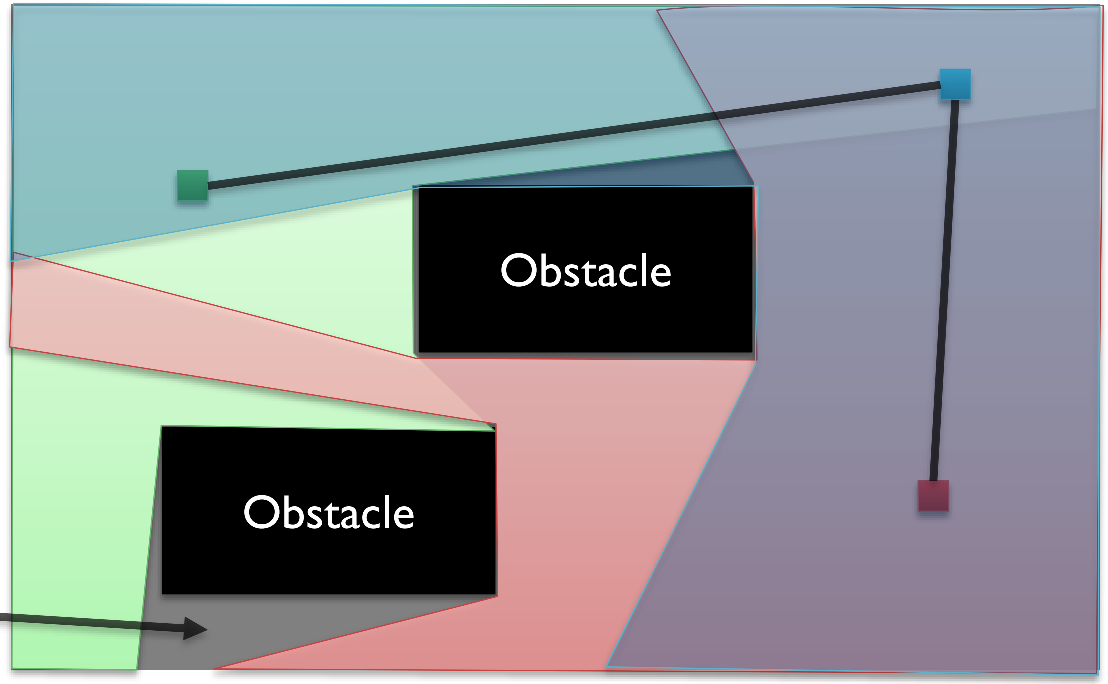


Can connect q to
any config in the green area

Can't connect q to
any other points

Preliminaries: Preprocessing

- **Preprocessing**: *Suppose* we can select configurations so that:
  - Their domains **cover** the entire config space
  - The configs can be **connected**

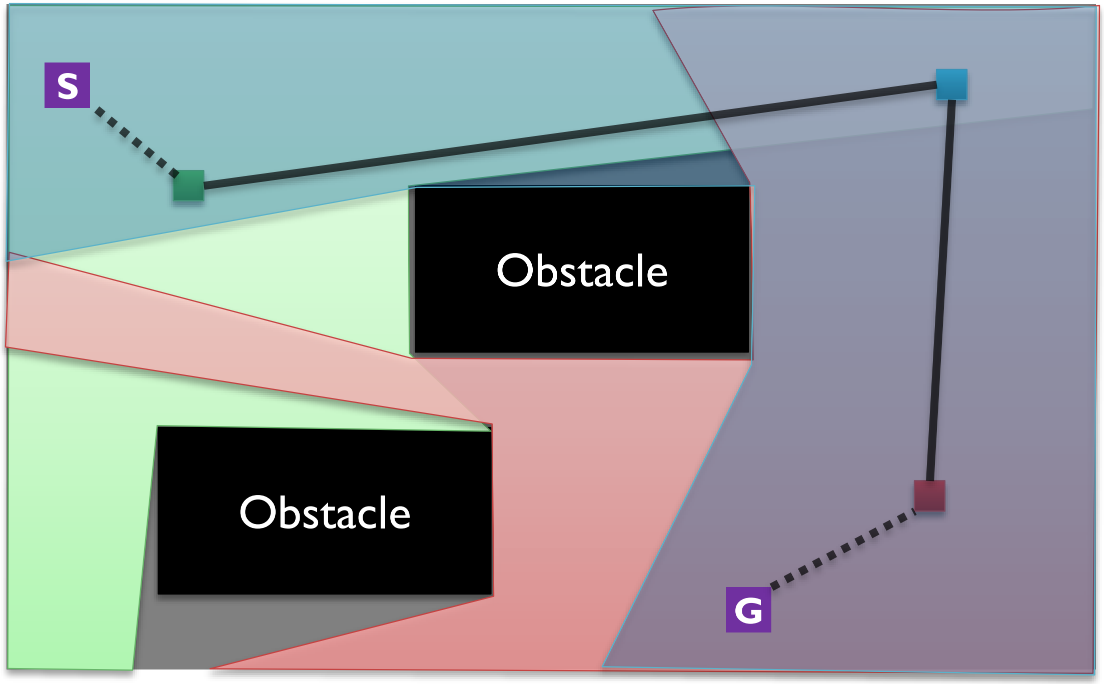


Incomplete so far…

(Imagine many obstacles, hundreds or thousands of configurations, many *dimensions*…)

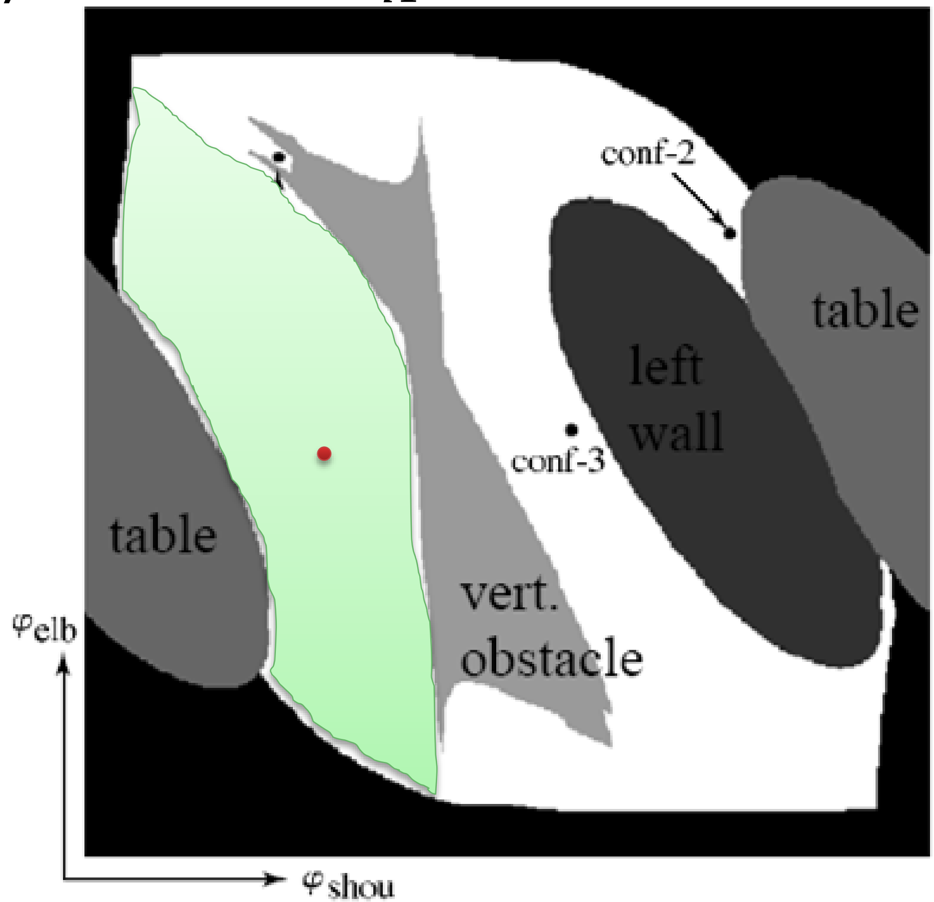
Preliminaries: Solving

- **<u>Solving</u>**: We get…
  - Start configuration q_{start}
    - **Connect** to another configuration
    - Must be possible:
      The ***domains*** of the existing configurations ***covered the entire space***
  - Goal configuration q_{goal}
    - Connect…
  - Find a path
    through the graph!



Preliminaries: Coverage Domains are Implicit

- Problem: We **<u>can't calculate</u>** the coverage domain D(q)
  - Local planner answers "can you connect q_1 with the **<u>specific config</u>** q_2?
  - Computing "all the configurations you can connect q_1 to":
    - High-dimensional spaces (57D???)
    - Complex motion constraints, not just physical obstacles
    - Too computationally complex, even if finite
    - Usually *infinitely* many possibilities



Preliminaries: Probabilistic Methods

- Solution: <u>**Probabilistic methods**</u>
  - Given a set of configurations $Q = \{q_1, \dots, q_n\}$:
    - Don't compute

$$\bigcup_{q \in Q} D(q)$$

    - Directly compute <u>**probability**</u>:

$$P\left(\bigcup_{q \in Q} D(q) \text{ covers entire free configuration space}\right)$$

    - Or:

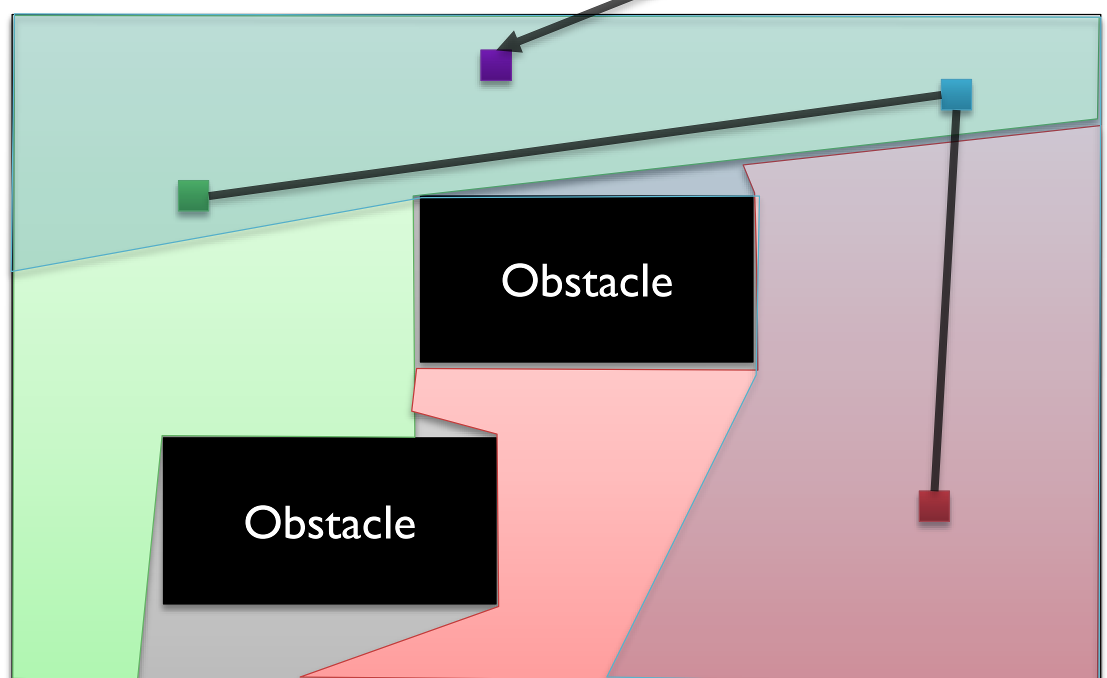
$$P\left(\text{if you pick a random free config, it belongs to} \bigcup_{q \in Q} D(q)\right)$$

    - Add configurations until probability is sufficiently high

Probabilistic Roadmaps

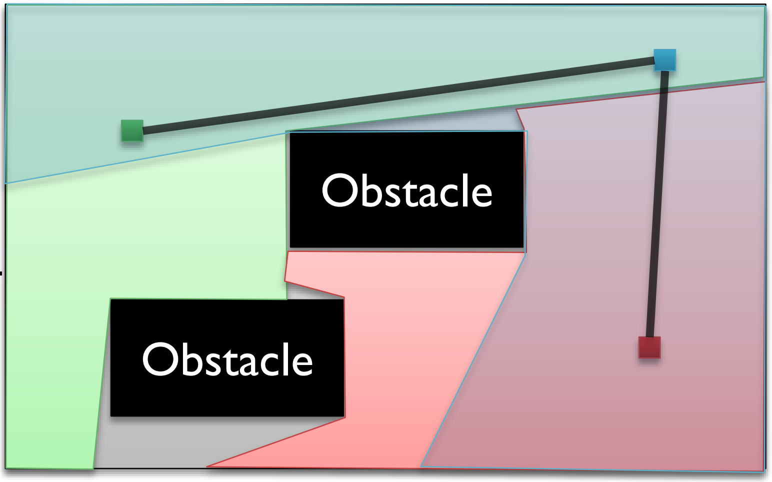
- Probabilistic Roadmaps (PRM): <u>**Construction Phase**</u>
  - M ← empty roadmap
    - **do** {
      randomly generate configuration *q* in free config space
      if (*q* was previously unreachable, so it would extend coverage) {
      add *q* and associated edges to M
      } **else if** (*q* was reachable, but now connects
      two previously unconnected configs) {
      add *q* and associated
      edges to M
      }
      } **until** (sufficient coverage)

A new config here
would *not* be added!



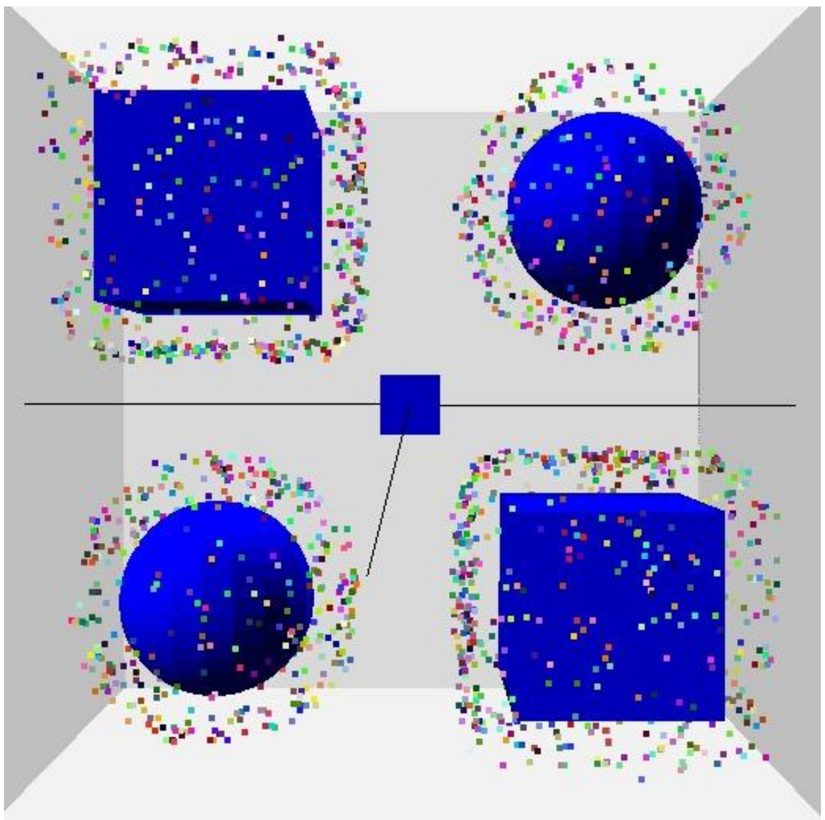
PRM: Sufficient Coverage

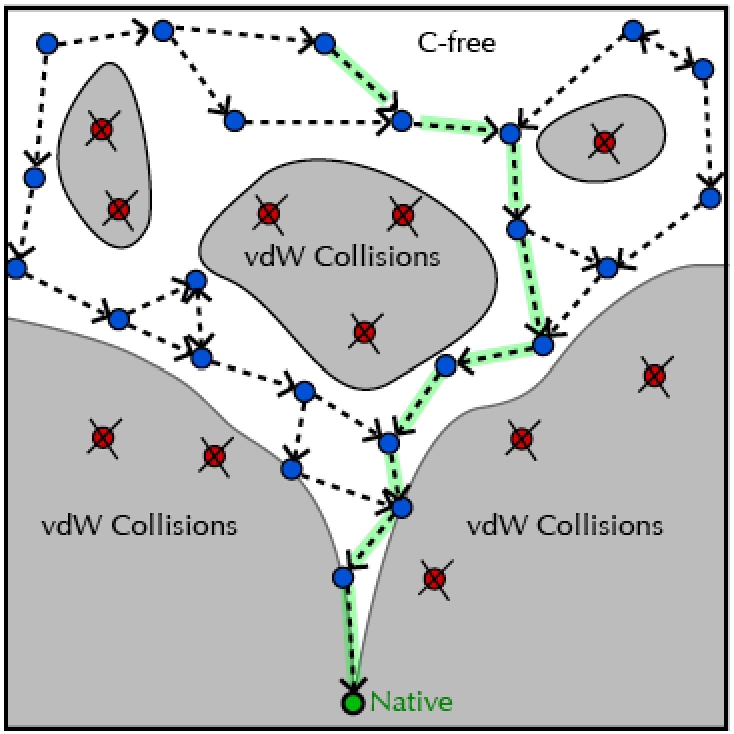
- When do you have **sufficient coverage**?
  - Suppose you have tested n configurations in a row *without being able to add one* to the road map
  - Then the roadmap covers the free config space with probability $1 - \frac{1}{\mathrm{n}}$
    - Example: $n = 1000$ ➔ coverage with 99.9% probability
- Why generate *randomly*? Why don't we ***select** a non-covered config*?
  - How? Many dimensions, complex connectivity, …
  - Random ➔ no need to explicitly calculate coverage domains!
- Construction phase done **in advance**
  - In a sense, a **learning phase**
  - Road map reused for many queries



PRM: Node Placement

- Node placement is *random* but not always *uniform*
  - Can be *biased* towards difficult areas

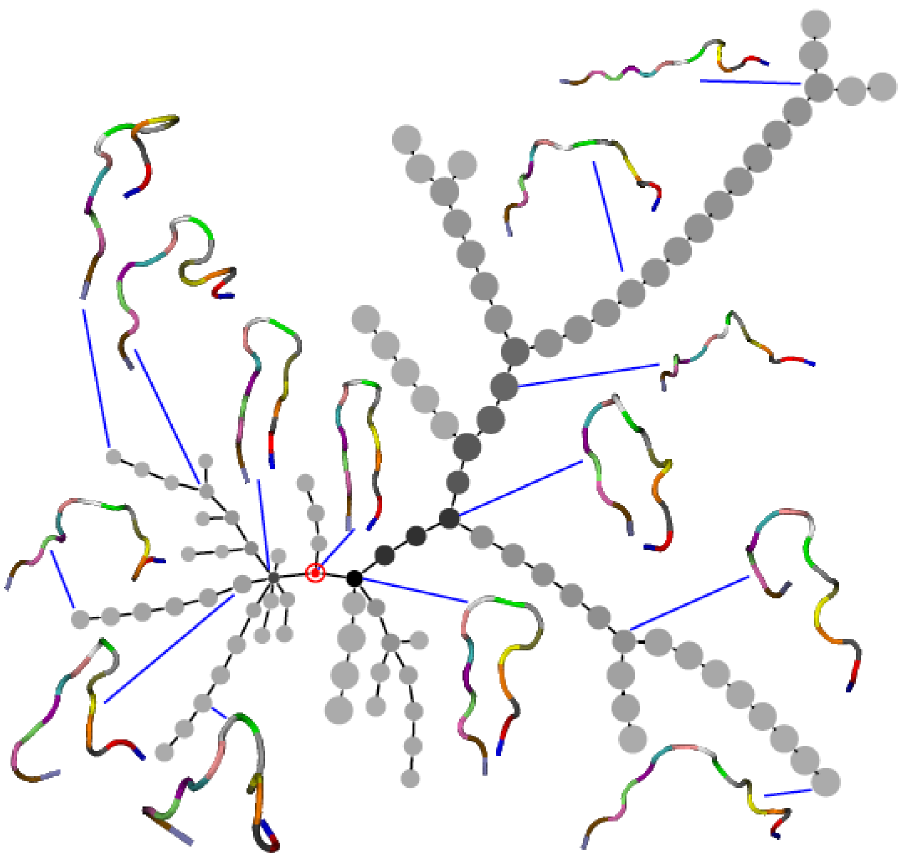




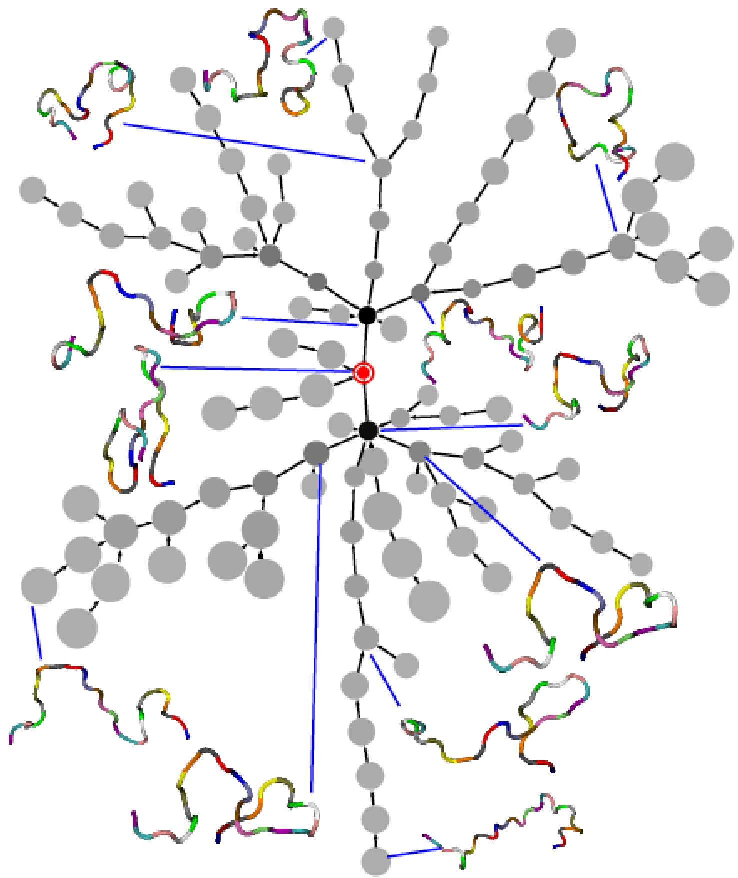
The "obstacles" above are "obstacles" in **configuration space**!

PRM: Protein Folding

- (Second example was from a protein folding application…)



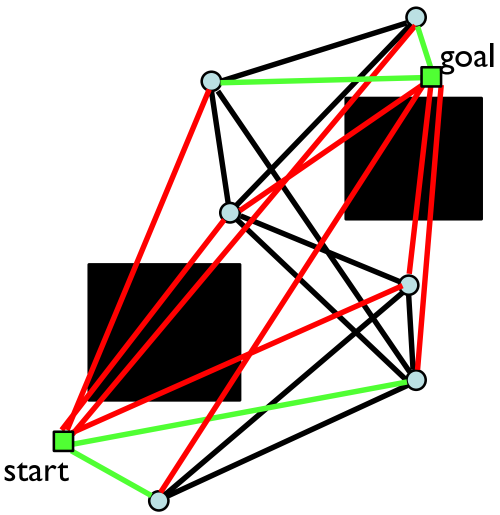
2GB1(16)



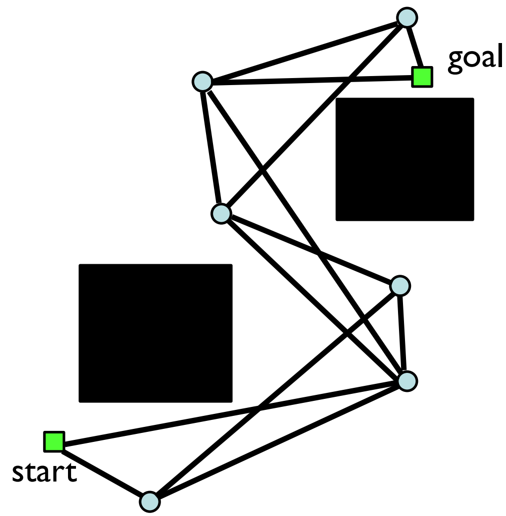
1E0L(28)

PRM: Query Phase

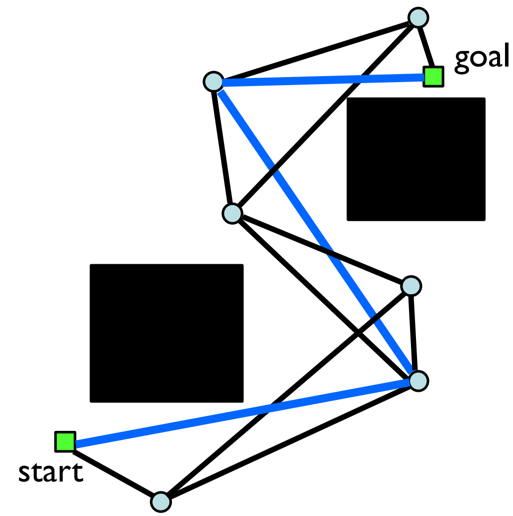
- Query Phase:



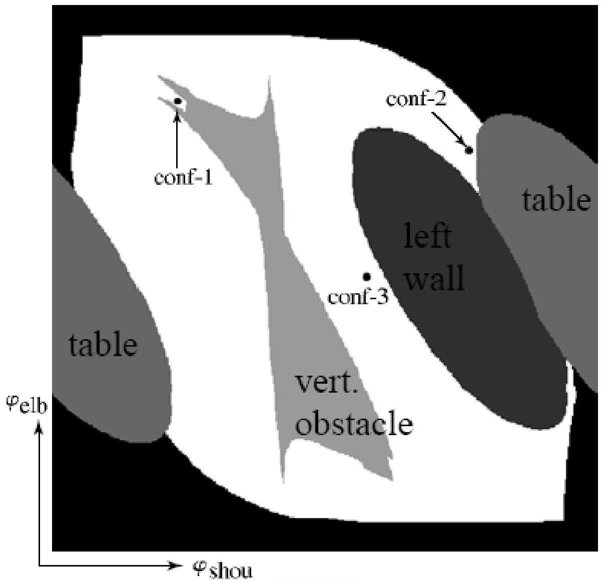
Add and connect start and goal configs to the roadmap (should be possible, as we have good coverage)



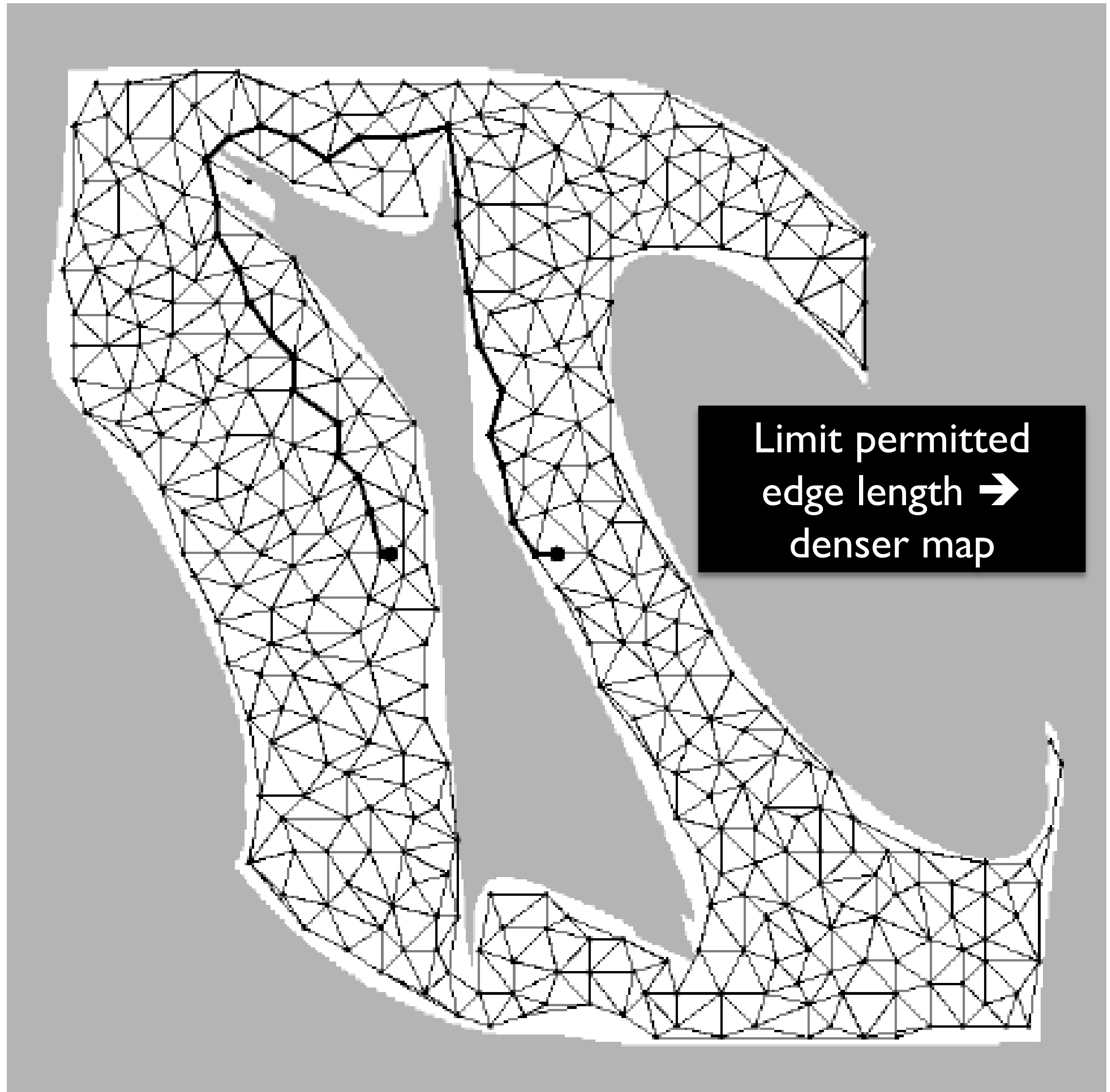
A* search



PRM: Result



Visualized i 2D
Could be 25D
Even in 2D, we have no *closed form description* of the shape – must *sample*!



Limit permitted edge length → denser map

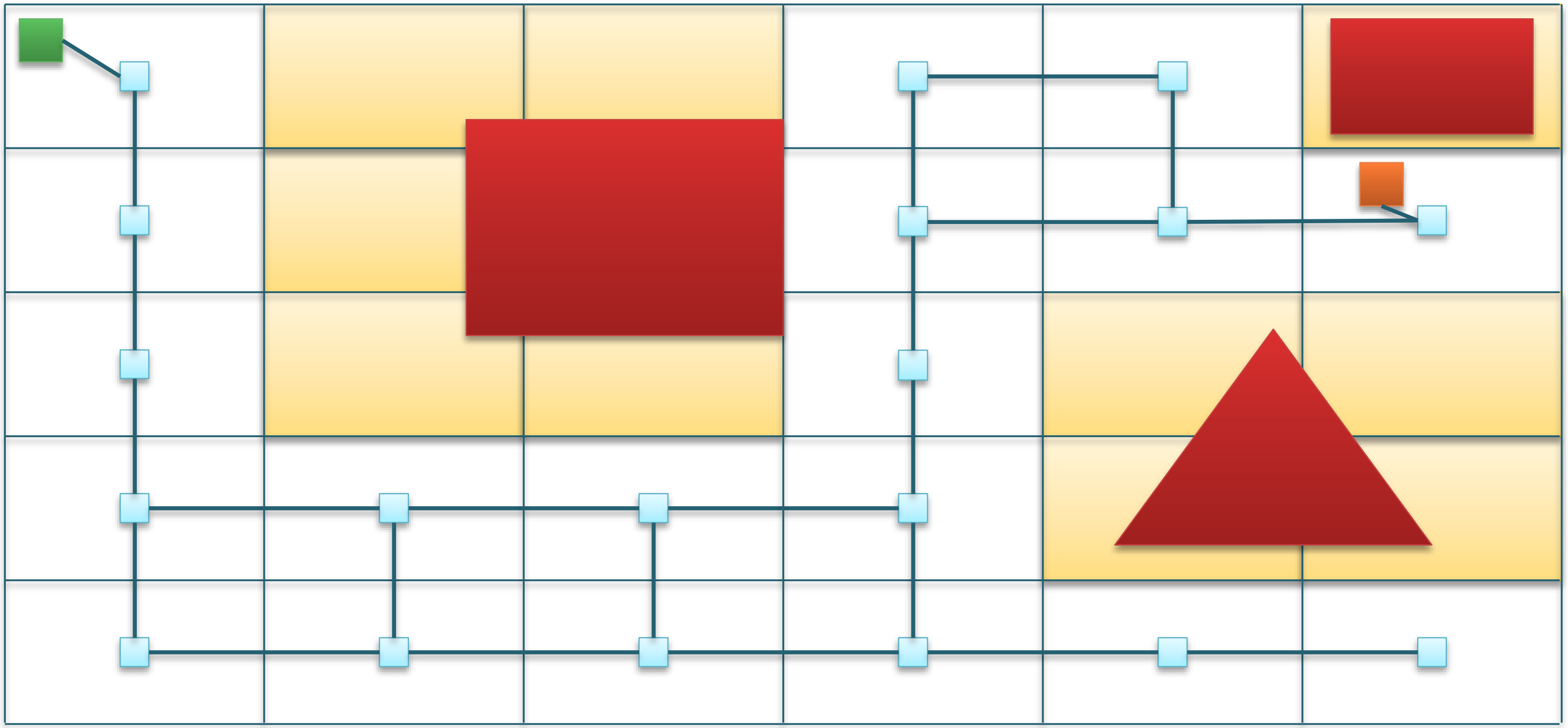
PRM: Properties

- **Properties**:
  - Scales better to higher dimensions
  - Deterministically incomplete, **probabilistically complete**
    - The more configurations you create,
      the greater the probability that a path can be found if one exists
      (approaching 1.0)

Graph Search

Graph Search (1)

- Given a discretization, how do we find a path?
  - One option: Heuristic search using A*
    - Heuristics in **simple geometric paths**: Manhattan distance (4 directions),
      Chebyshev distance (moving in 8 directions),
      Euclidian distance (in general), …
    - Other heuristics in **complex configuration spaces**



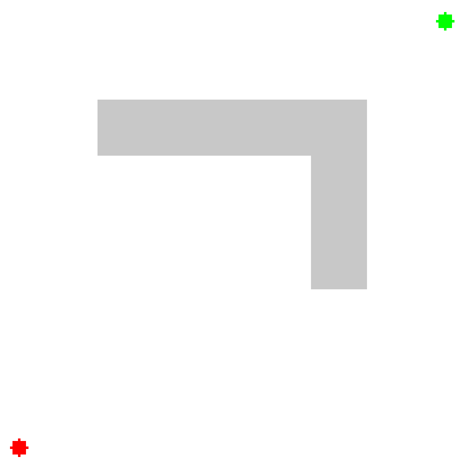
Graph Search (2)

- Suppose **new obstacles** are detected during execution
  - A*: Update map and replan from scratch
    - Inefficient
  - D* (Dynamic A*): Informed **incremental** search
    - First, find a path using information about known obstacles
    - When new obstacles are detected:
      - Affected nodes are returned to the OPEN list, marked as RAISE: More expensive than before
      - Incrementally updates only those nodes whose cost change due to the new obstacles
  - Focused D*:
    - Focuses propagation towards the robot – additional speedup
  - …

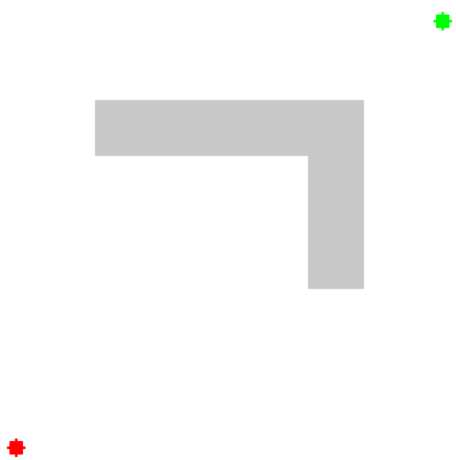
Graph Search (3)

- **<u>Anytime</u>** algorithms:
  - ***Be able to answer whenever I interrupt you!***
  - In practice: ***Create*** some path quickly, then incrementally ***improve*** it
  - "Repeated weighted A*" (standard technique)
    - Run A* with $f(n) = g(n) + \boldsymbol{W} \cdot h(n)$, where $W > 1$: Faster but suboptimal

$w = 1$:
Standard A*



$w = 5$:
Distance from goal is exaggerated
➔
suboptimal



- Decrease W and **<u>repeat</u>**
- But: Has to **<u>redo search</u>** from scratch in each run!

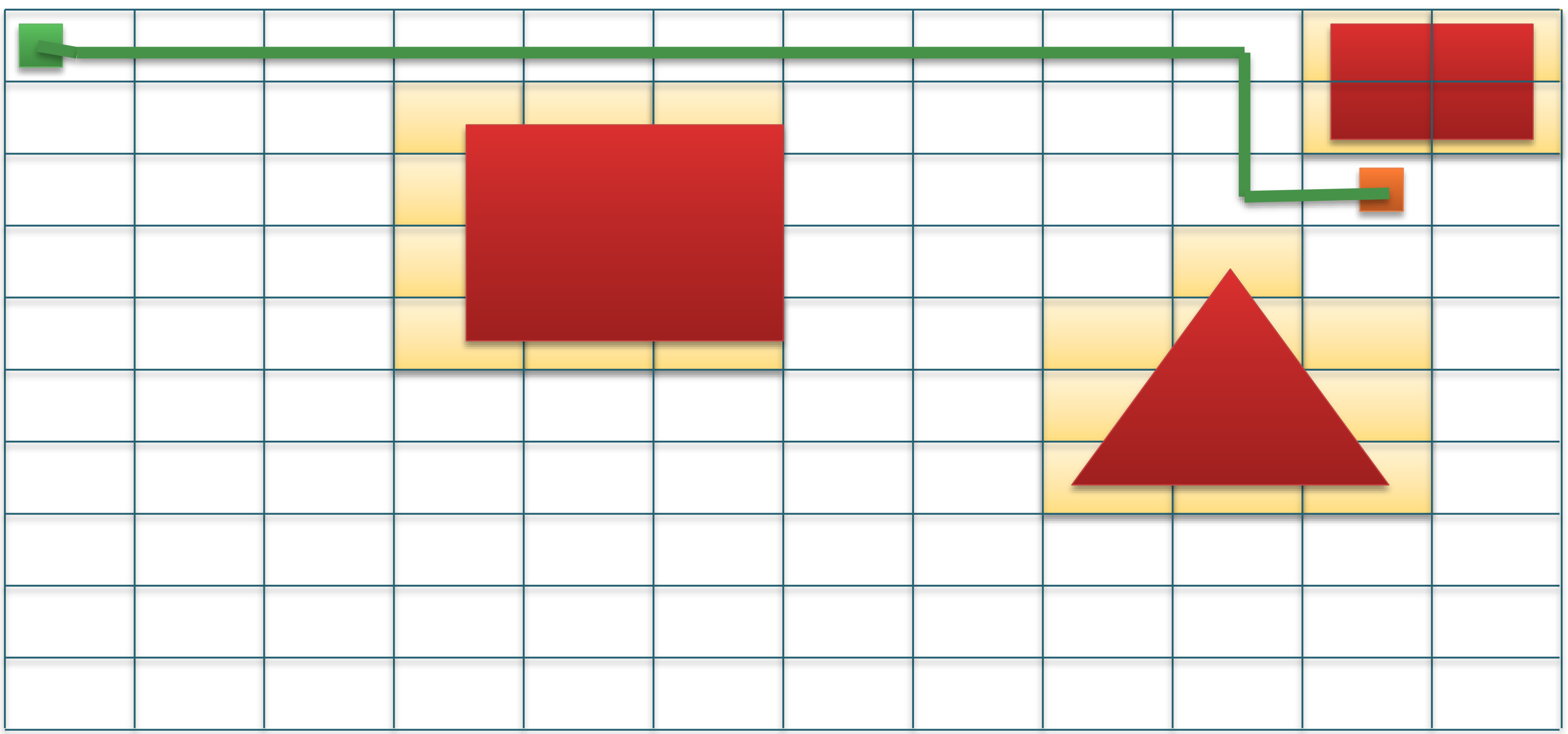
Graph Search (3)

- **Anytime** algorithms:
  - Anytime Repairing A*
    - Like "repeated weighted A*", but reuses search results from earlier iterations
  - Anytime Dynamic A* (AD*)
    - **Both** replanning when problems change and anytime planning
  - …

Path Smoothing

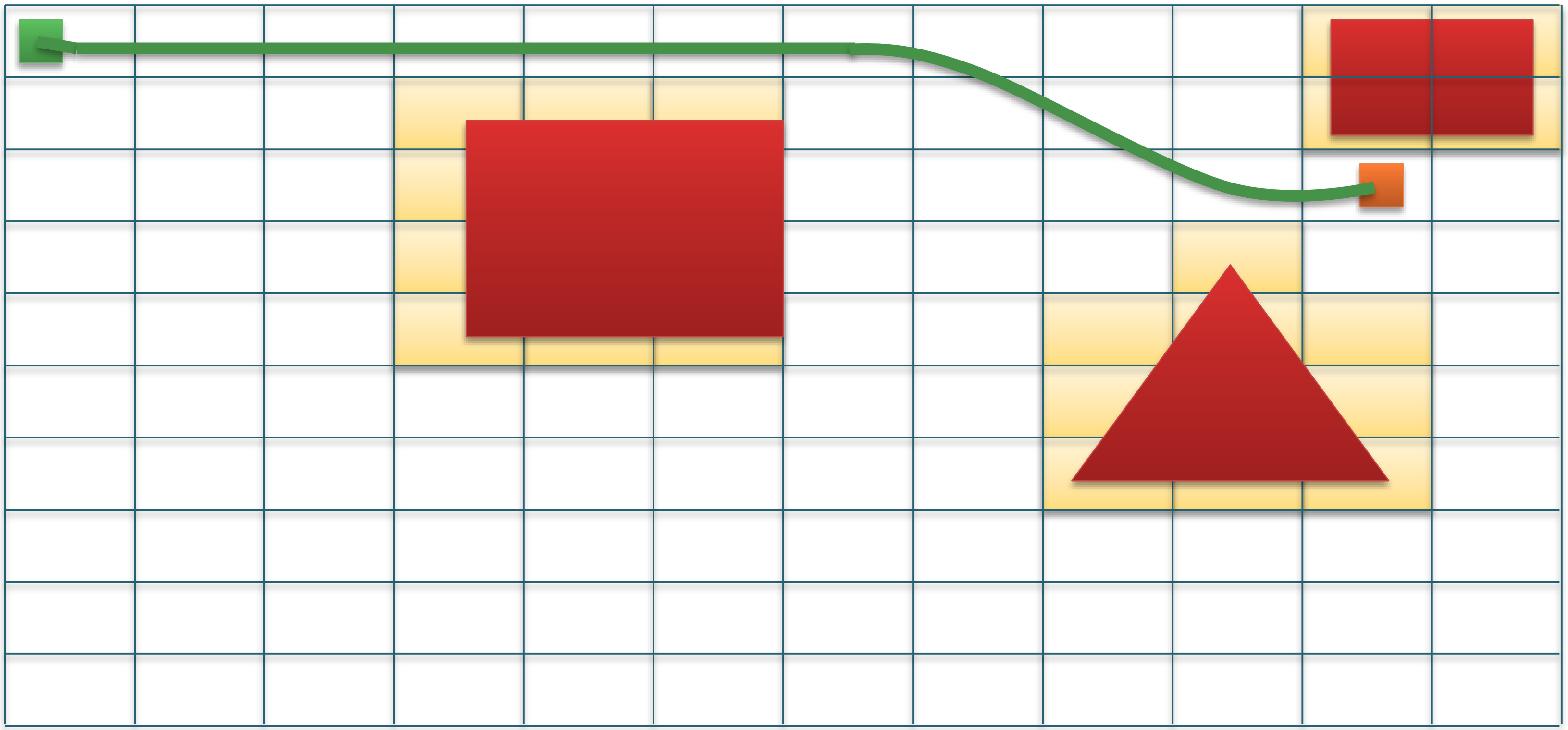
Suboptimal Paths

- Paths are often **suboptimal** in the continuous space
  - Only the chosen points in the cells are used
  - In this example: The midpoints



Smoothing

- Paths can be improved through **smoothing** after generation
  - Still generally does not lead to optimal paths
  - This is just a simple example, where smoothing is easy



Open Motion Planning Library

- Want to experiment?
  - Open Motion Planning Library
  - http://ompl.kavrakilab.org/index.html