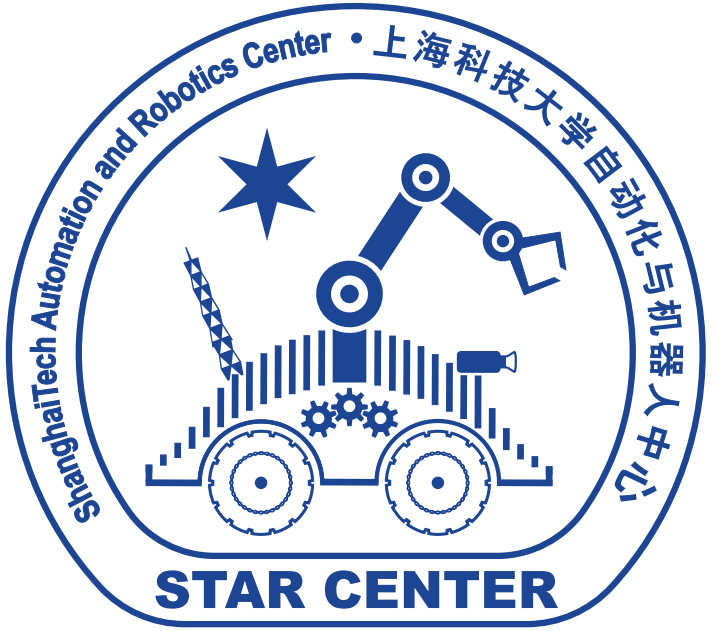
上海科技大学

ShanghaiTech University

CS289: **Mobile Manipulation Fall 2025**

Sören Schwertfeger

ShanghaiTech University



Outline

- Continue Mobile Robotics
- Simulation

SIMULTANEOUS LOCALIZATION AND MAPPING - SLAM

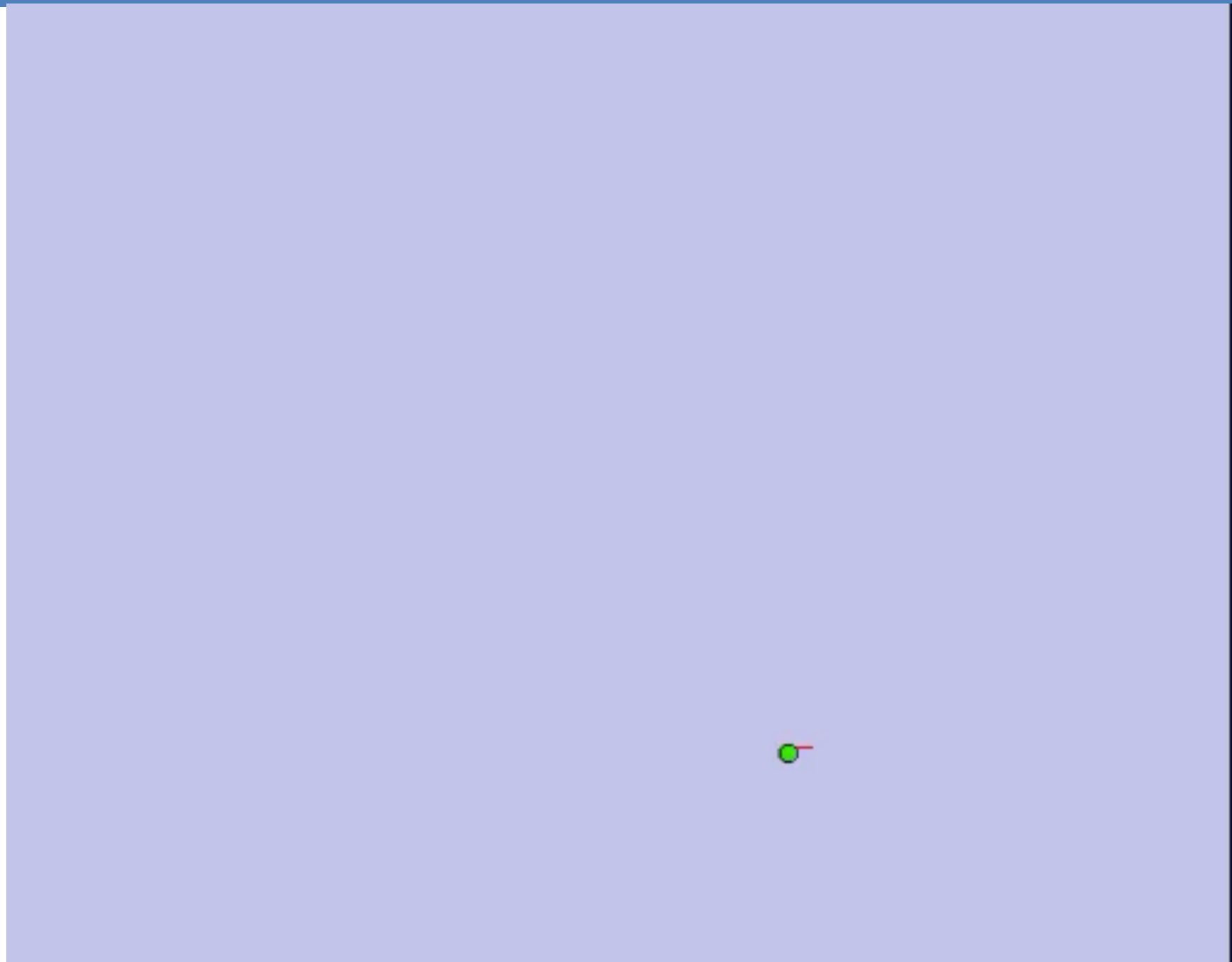
Raw Odometry

- Famous Intel Research Lab dataset (Seattle) by Dirk Hähnel

Courtesy of S. Thrun

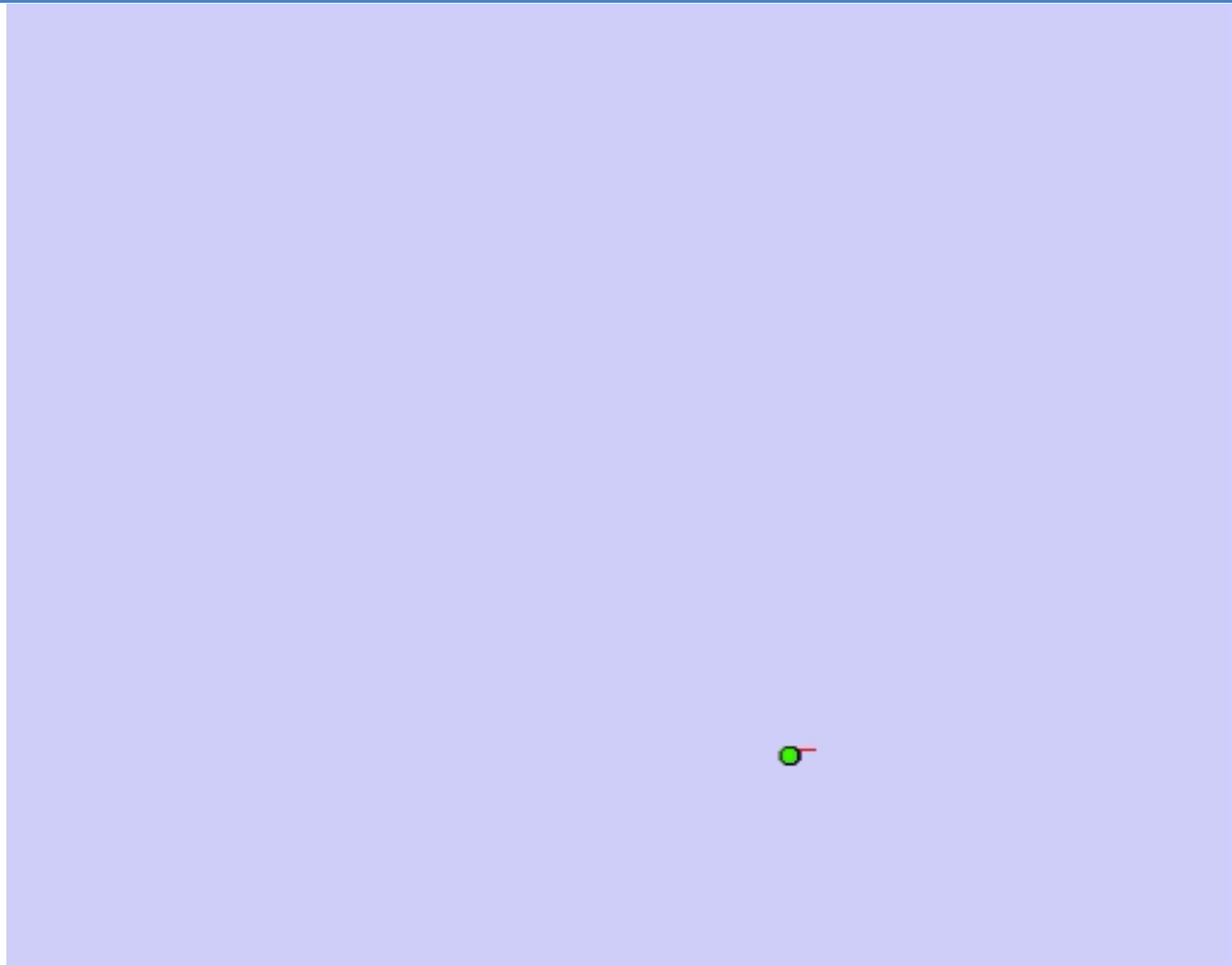
http://robots.stanford.edu/videos.html

Scan Matching: compare to sensor data from previous scan



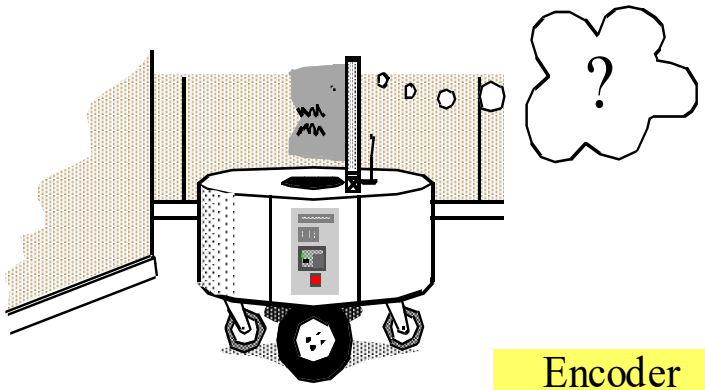
Courtesy of S. Thrun

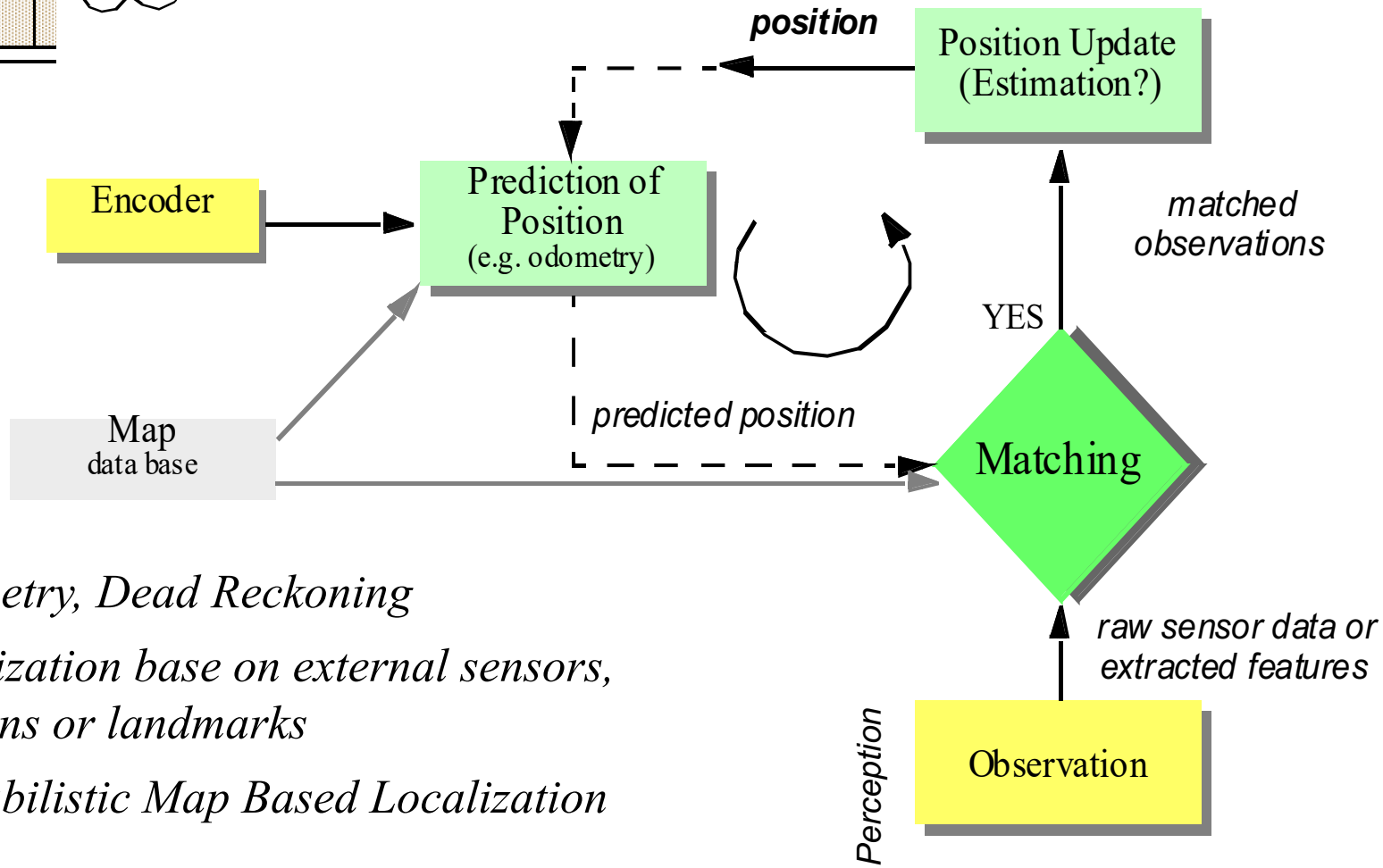
FastSLAM: Particle-Filter SLAM



Courtesy of S. Thrun

Map based localization



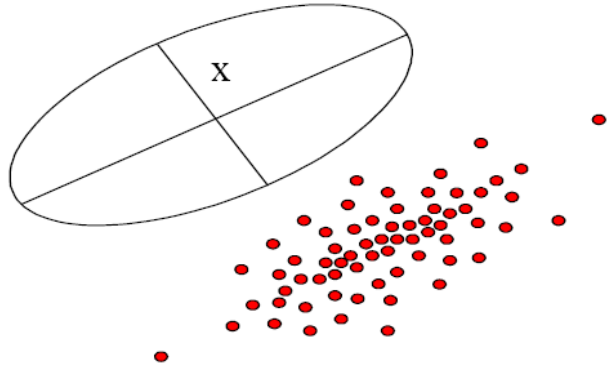


- *Odometry, Dead Reckoning*
- *Localization base on external sensors, beacons or landmarks*
- *Probabilistic Map Based Localization*

Localization

- Based on control commands => Open Loop!
- Wheel odometry
  - Compass, Accelerometer, Gyro => IMU
- Scan Matching of Range Sensors == Registration (rigid => no scaling or shearing)
  - ICP: scan to scan or scan to map
    - Needs good initial guess
  - NDT registration
  - Feature-based registration
  - Direct/ optimization based registration
- Grid-based Localization
- Kalman Filter Based Localization
- Monte-Carlo Localization (MCL) == Particle Filter
  - Adaptative MCL => AMCL
- Visual Odometry (VO)
  - With IMU: Visual Inertial Odometry (VIO)
- SLAM techniques
- 3D Reconstruction
  - Structure from Motion/ Bundle Adjustment
  - Localization is by-product
- Absolute Localization:
  - GPS
  - Markers (e.g. QR code)
  - Landmarks (e.g. ShanghaiTech Tower)

Monte Carlo Localization (MCL)



probability distribution (ellipse) as particle set (red dots)

- Input: Global, known map and laser scan
- Particle filter: set of particles representing a robot state
  - Here: robot pose (position & orientation)
  - Particle filter SLAM (e.g. FastSLAM): also map!
  - Particles are sampled based on probability distribution
- Assign weights (scores) to particles based on how well the scan matches to the map, given this pose
- Markov property: Current state only depends on previous state

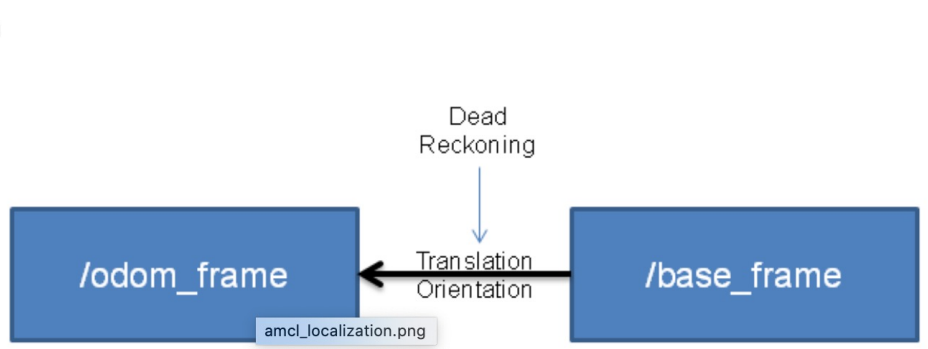
- Algorithm:

1. For all particles:
   1. Apply motion update (e.g. odometry)
   2. Apply the sensor update (scan match) and calculate new weights
2. Re-Sample particles based on their weights

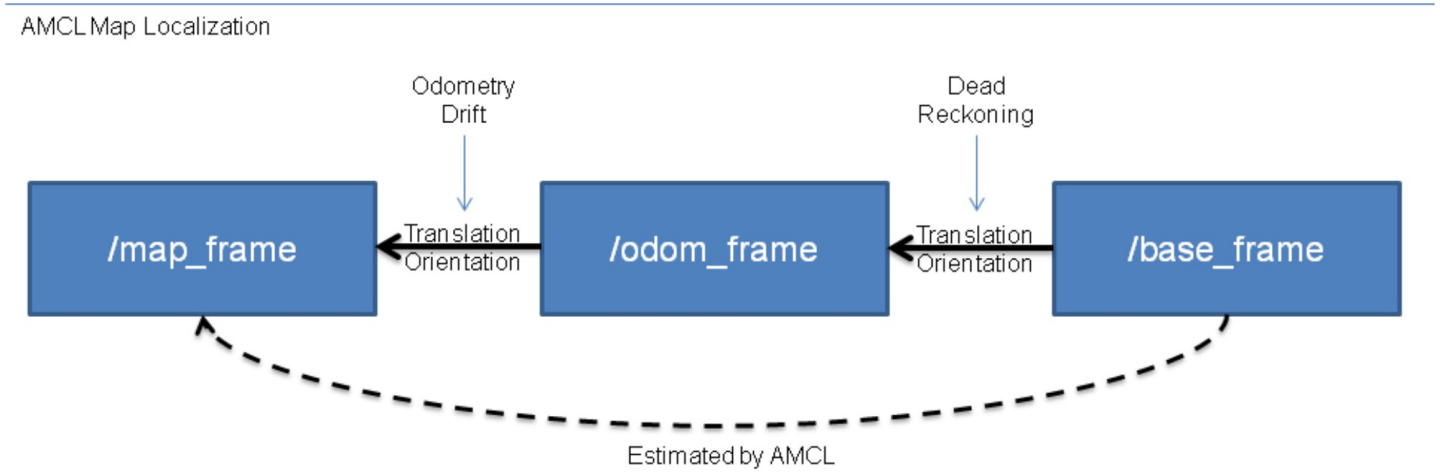
- Can solve the kidnapped robot problem (also wake-up robot problem)
- Problem: Particle of correct pose might not exist…

Adaptive Monte Carlo Localization (AMCL)

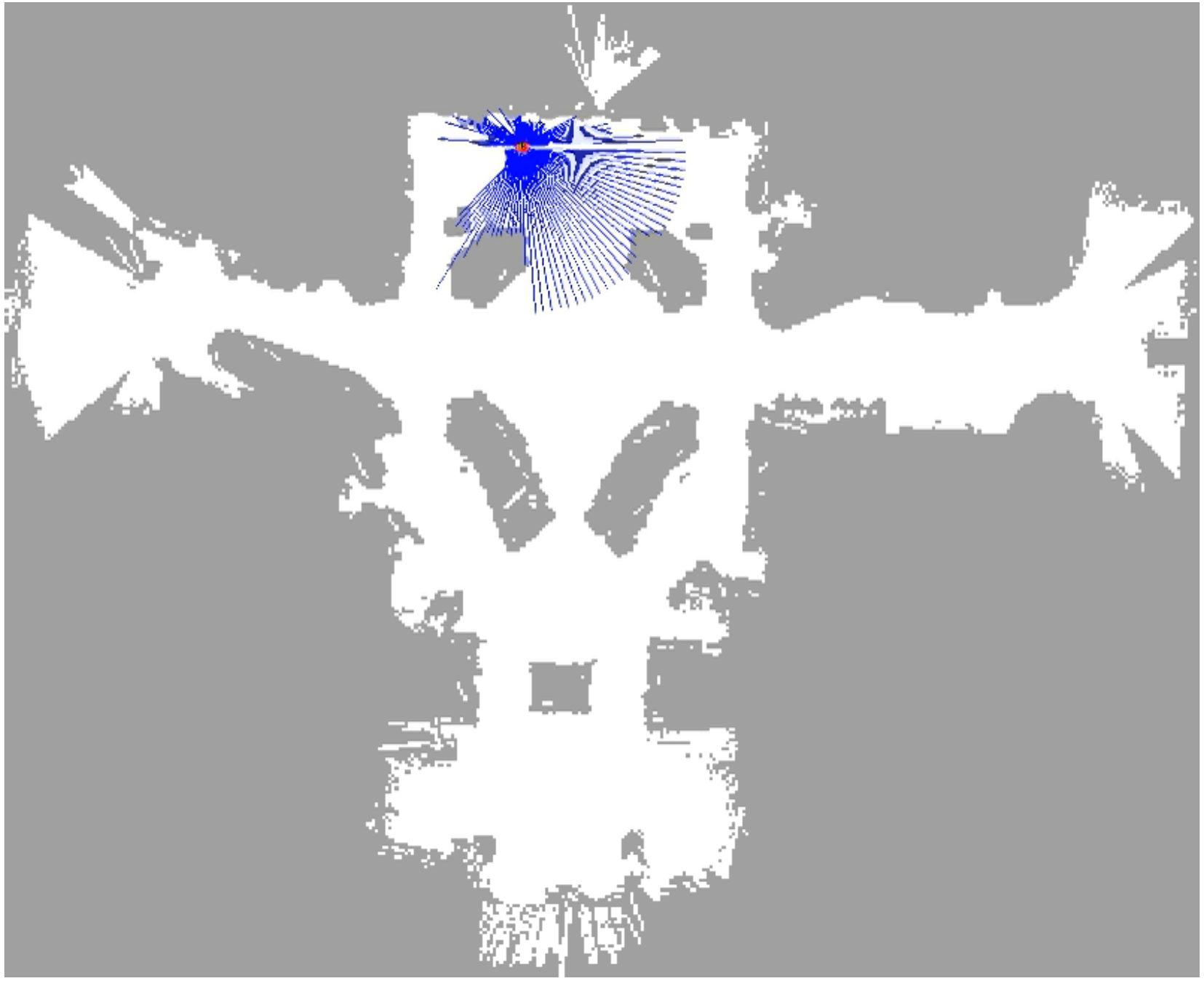
- Sample particles adaptively
  - Based on error estimate
  - Kullback-Leibler divergence (KLD)
  - => when particles have converged, have a fewer number of particles
  - Sample size is re-calculated each iteration



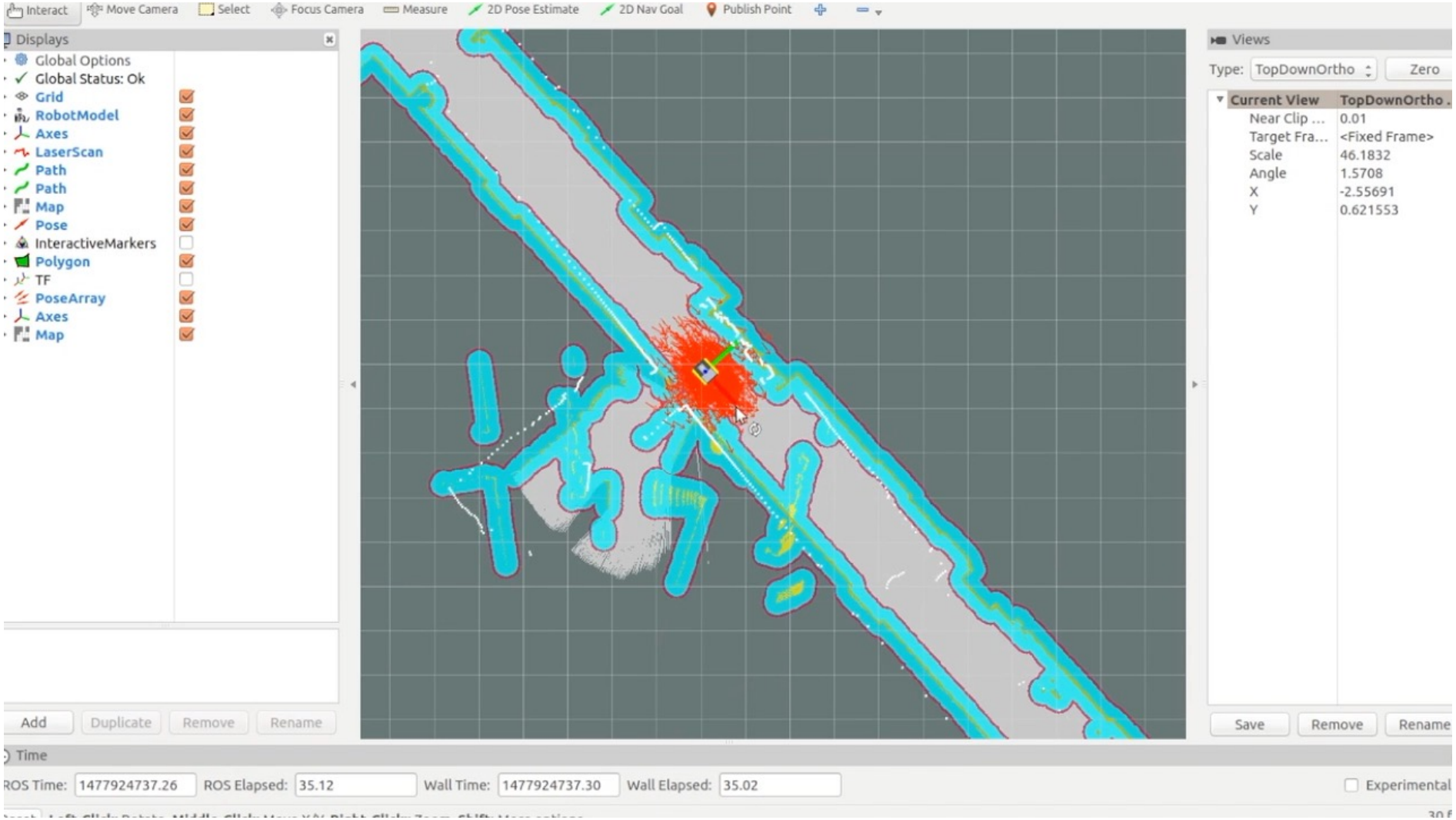
- http://wiki.ros.org/amcl
- Used by the ROS Navigation stack



MCL & Robot Kidnapping



AMCL in ROS


Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Displays
Global Options
Global Status: Ok
Grid
RobotModel
Axes
LaserScan
Path
Path
Map
Pose
InteractiveMarkers
Polygon
TF
PoseArray
Axes
Map
Add
Duplicate
Remove
Rename
Views
Type: TopDownOrtho
Zero
Current View
TopDownOrtho
Near Clip ...
0.01
Target Fra...
<Fixed Frame>
Scale
46.1832
Angle
1.5708
X
-2.55691
Y
0.621553
Save
Remove
Rename
Time
ROS Time:
1477924737.26
ROS Elapsed:
35.12
Wall Time:
1477924737.30
Wall Elapsed:
35.02
Experimental

Overview of SLAM Methods

- Camera
  - Feature-Based Methods
    - MonoSLAM
    - PTAM
    - ORB-SLAM
  - Direct Methods
    - DTAM
    - LSD-SLAM
    - DSO
  - Semi-Direct Methods
    - SVO
  - Others
    - PoseNet
    - CNN-SLAM
    - …
- Laser
  - Pose Graph
    - Cartographer
    - Karto-SLAM
    - Hector-SLAM
    - BLAM
    - LIO
  - Particle Filter
    - FastSLAM
    - Gmapping
  - Extended Kalman Filter
    - EKF-SLAM
    - LINS
  - Others
    - LOAM
    - IMLS-SLAM
    - …

SLAM Front-end & Back-end

- Front-end
  - calculate relative poses between several frames/ to map
    - scan matching
    - image registration
    - …
  - estimate absolute poses
  - construct the local map
- Back-end
  - optimize the absolute poses and mapping
  - only if a loop was closed

https://ww2.mathworks.cn/help/nav/ug/implement-simultaneous-localization-and-mapping-with-lidar-scans.html

THREE SLAM PARADIGMS

The Three SLAM Paradigms

- Most of the SLAM algorithms are based on the following three different approaches:
  - Extended Kalman Filter SLAM: (called EKF SLAM)
  - Particle Filter SLAM: (called FAST SLAM)
  - Graph-Based SLAM

EKF SLAM: overview

- **Extended state vector** y_t : robot pose x_t + position of all the features m_i in the map:

$$y_t = [x_t, m_0, \dots, m_{n-1}]^T$$

- Example: 2D line-landmarks, size of y_t = $3+2n$: three variables to represent the robot pose + $2n$ variables for the n line-landmarks having vector components

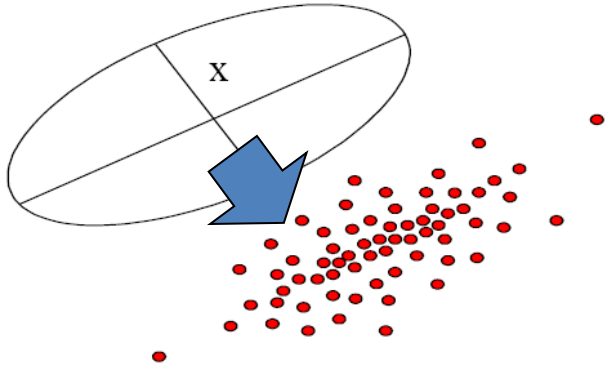
$$(\alpha_i, r_i)$$

$$y_t = [x_t, y_t, \theta_t, \alpha_0, r_0, \dots, \alpha_{n-1}, r_{n-1}]^T$$

- As the robot moves and takes measurements, the state vector and covariance matrix are updated using the standard equations of the extended Kalman filter
- Drawback: EKF SLAM is computationally very expensive.

Particle Filter SLAM: FastSLAM

- **FastSLAM approach**
  - Using particle filters.
  - Particle filters: mathematical models that represent probability distribution as a set of discrete particles that occupy the state space.



probability distribution (ellipse) as particle set (red dots)

- Particle filter update
  - Generate new particle distribution using motion model and controls

a) For each particle:

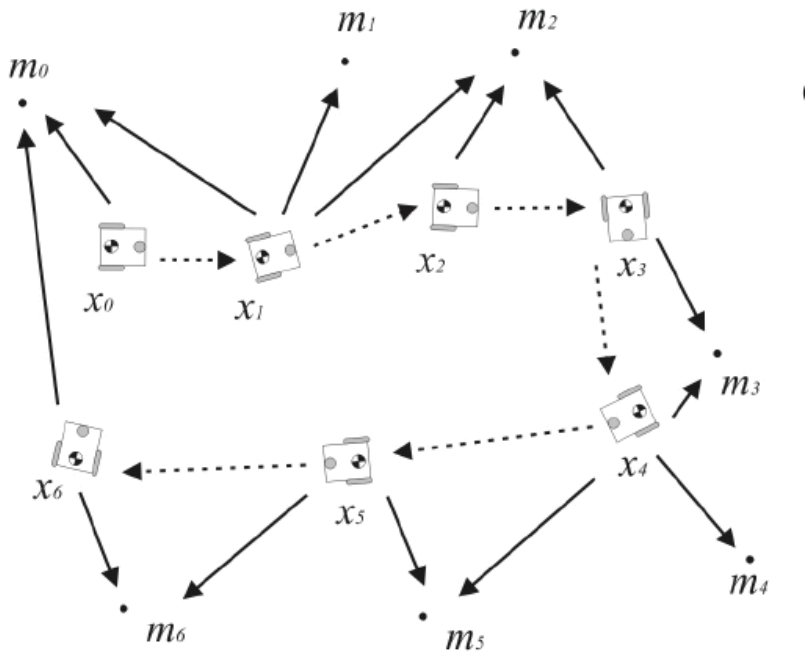
1. Compare particle's prediction of measurements with actual measurements
2. Particles whose predictions match the measurements are given a high weight

b) Filter resample:

- Resample particles based on weight
- Filter resample
  - Assign each particle a weight depending on how well its estimate of the state agrees with the measurements and randomly draw particles from previous distribution based on weights creating a new distribution.

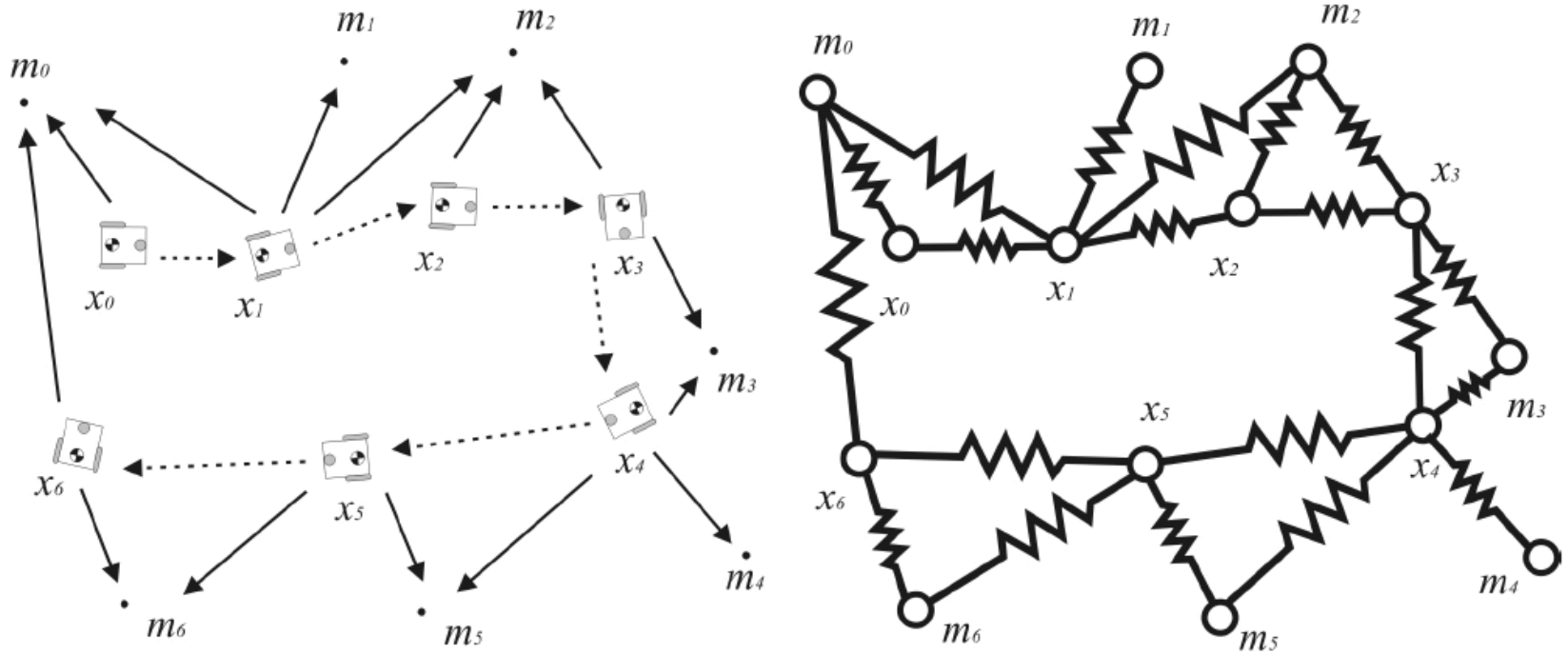
Graph-Based SLAM (1/3)

- SLAM problem can be interpreted as a sparse graph of nodes and constraints between nodes.
- The nodes of the graph are the robot locations and the features in the map.
- Constraints: relative position between consecutive robot poses , (given by the odometry input ***u***) and the relative position between the robot locations and the features observed from those locations.



Graph-Based SLAM (2/3)

- Constraints are not rigid but soft constraints!
- Relaxation: compute the solution to the full SLAM problem =>
  - Compute best estimate of the robot path and the environment map.
  - Graph-based SLAM represents robot locations and features as the nodes of an elastic net. The SLAM solution can then be found by computing the state of minimal energy of this net



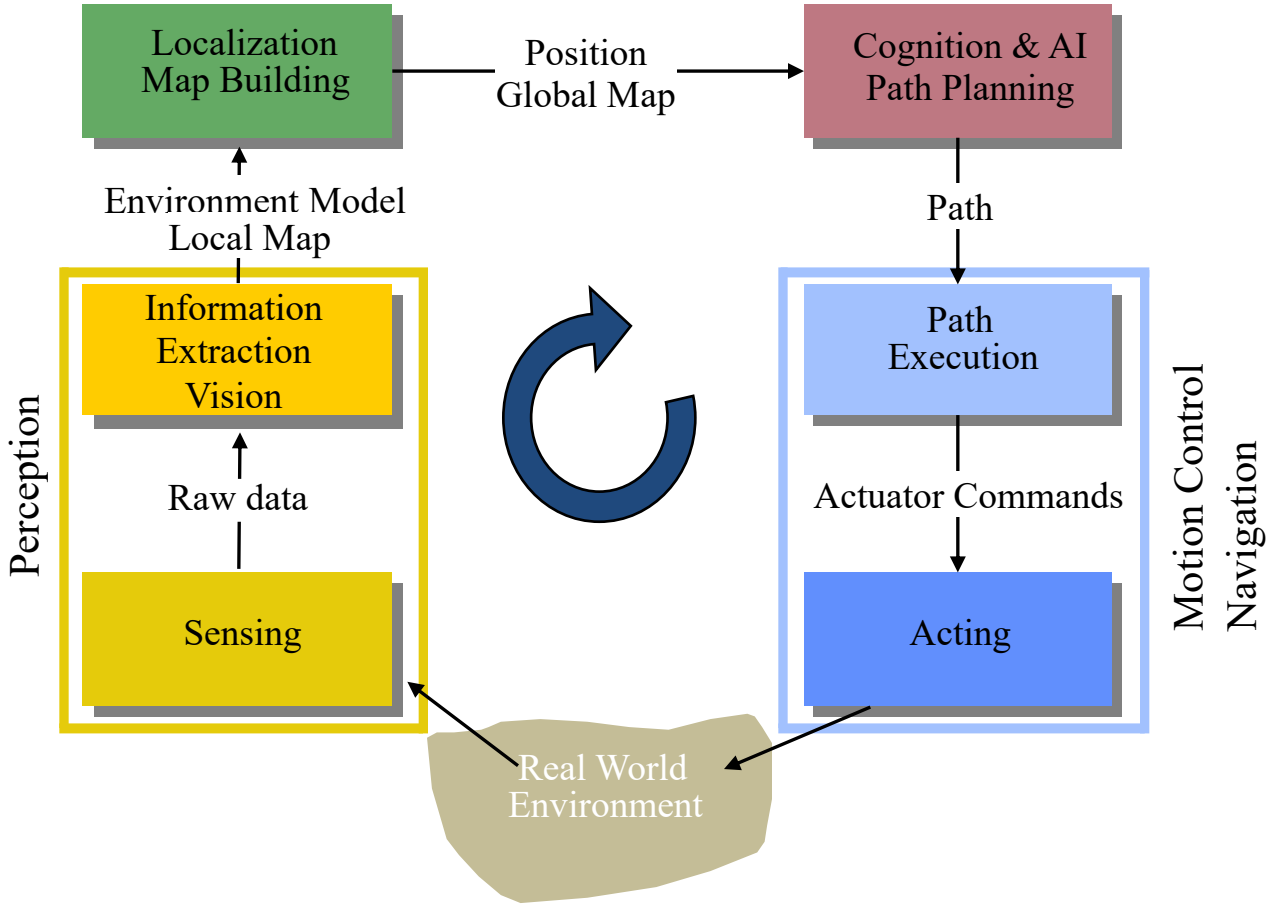
Graph-Based SLAM (3/3)

- Significant advantage of graph-based SLAM techniques over EKF SLAM:
  - EKF SLAM: computation and memory for to update and store the covariance matrix is quadratic with the number of features.
  - Graph-based SLAM: update time of the graph is constant and the required memory is linear in the number of features.
- However, the final graph optimization can become computationally costly if the robot path is long.

- Libraries for graph-based slam: g2o, ceres

PLANNING

General Control Scheme for Mobile Robot Systems

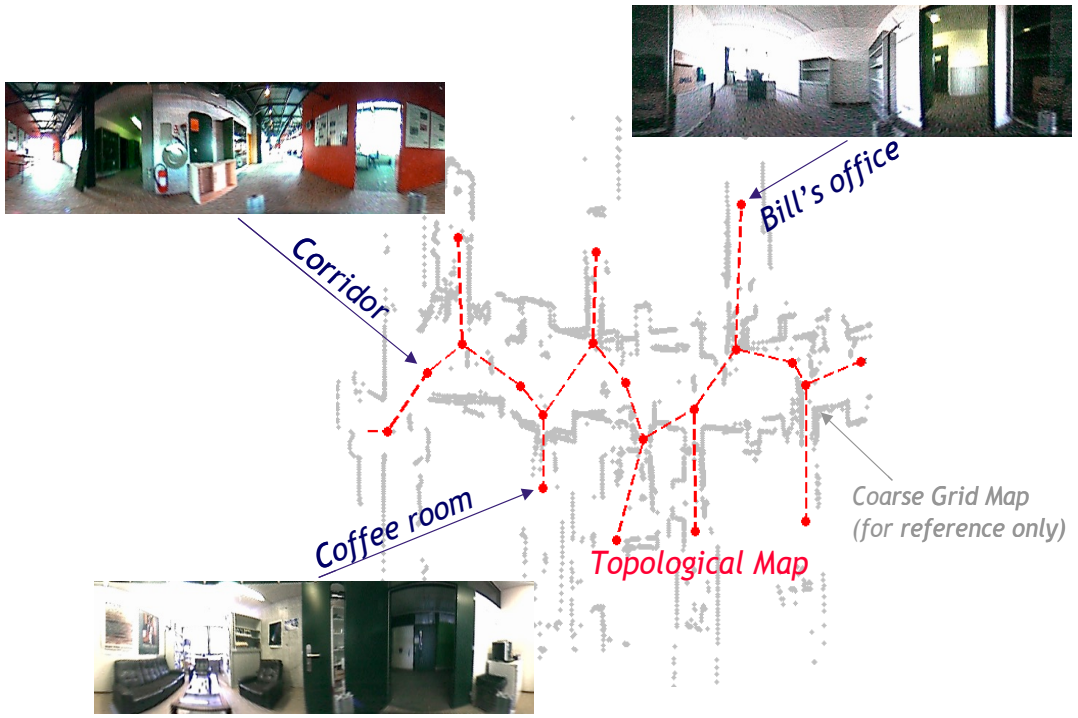


With material from Roland Siegwart and Davide Scaramuzza, ETH Zurich

The Planning Problem

- The problem: find a path in the work space (physical space) from the initial position to the goal position avoiding all collisions with the obstacles

- Assumption: there exists a good enough map of the environment for navigation.



The Planning Problem

- We can generally distinguish between
  - (global) path planning and
  - (local) obstacle avoidance.

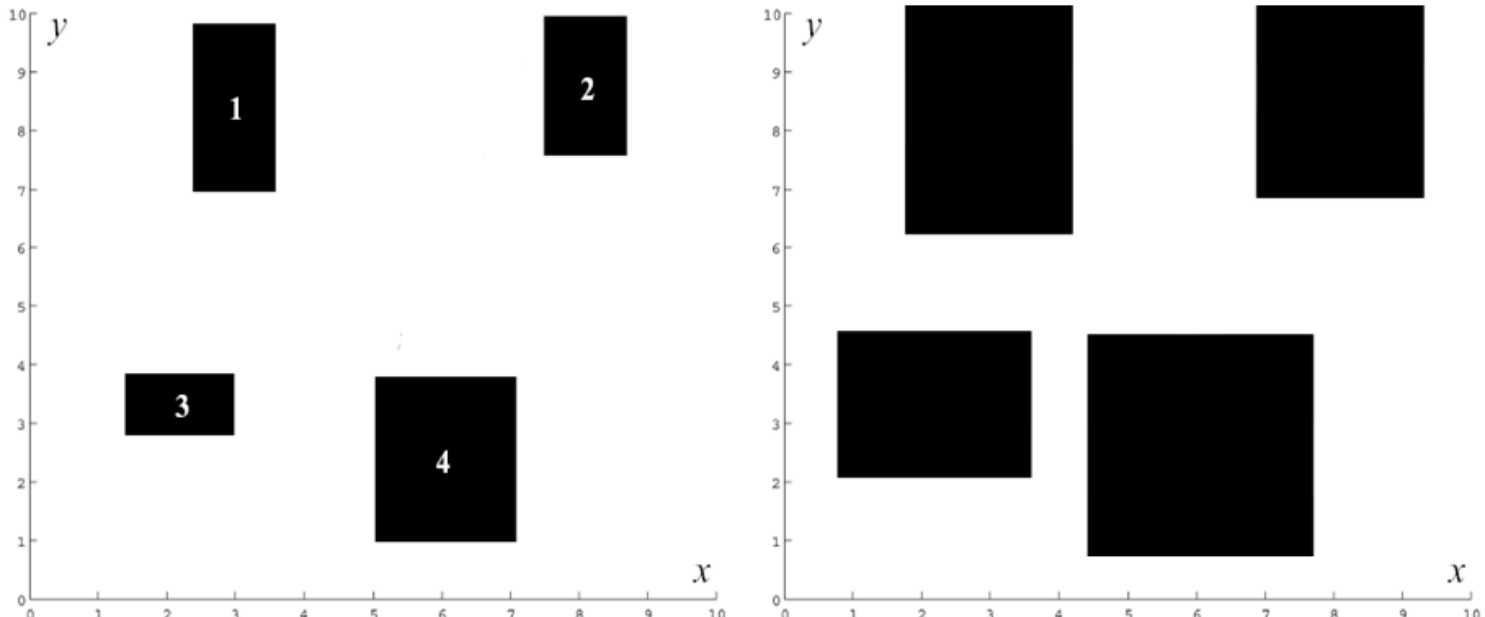
- First step:
  - Transformation of the map into a representation useful for planning
  - This step is planner-dependent

- Second step:
  - Plan a path on the transformed map

- Third step:
  - Send motion commands to controller
  - This step is planner-dependent (e.g. Model based feed forward, path following)

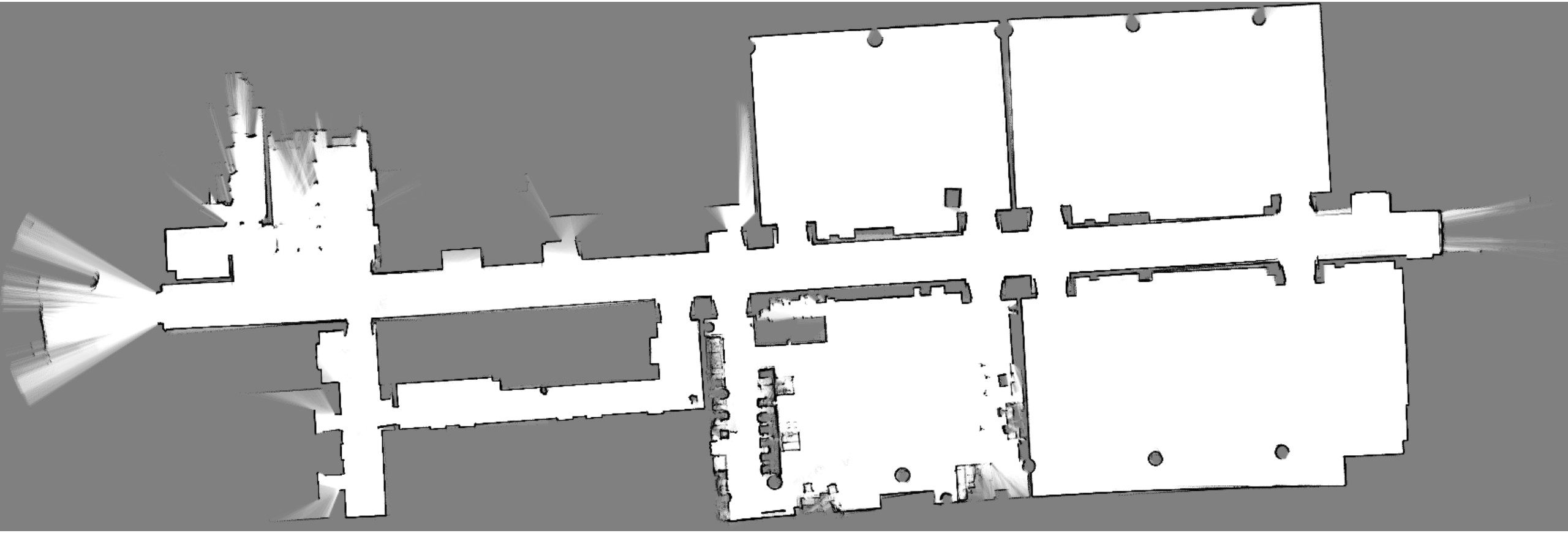
Configuration Space for a Mobile Robot

- Mobile robots operating on a flat ground (2D) have 3 DoF: (x, y, θ)
- Differential Drive: only two motors => only 2 degrees of freedom directly controlled (forward/ backward + turn) => non-holonomic
- Simplification: assume robot is holonomic and it is a point => configuration space is reduced to 2D (x,y)
- => inflate obstacle by size of the robot radius to avoid crashes => obstacle growing



Typical Configuration Space: Occupancy grid

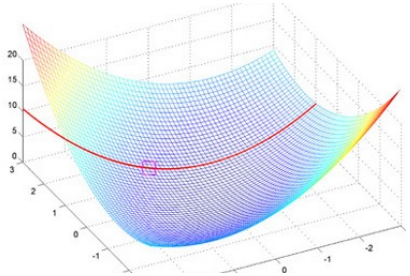
- Fixed cell decomposition: occupancy grid example: STAR Center



Path Planning: Overview of Algorithms

1. Optimal Control

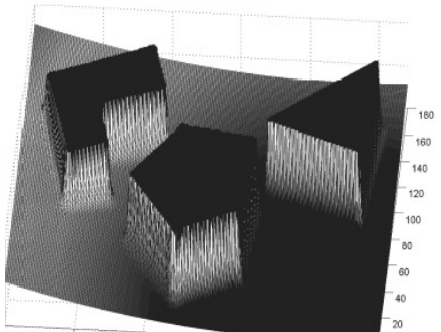
- Solves truly optimal solution
- Becomes intractable for even moderately complex as well as nonconvex problems



Source: http://mitocw.udsm.ac.tz

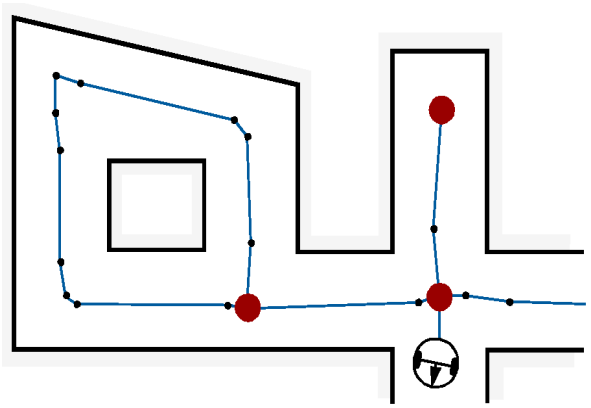
2. Potential Field

- Imposes a mathematical function over the state/configuration space
- Many physical metaphors exist
- Often employed due to its simplicity and similarity to optimal control solutions

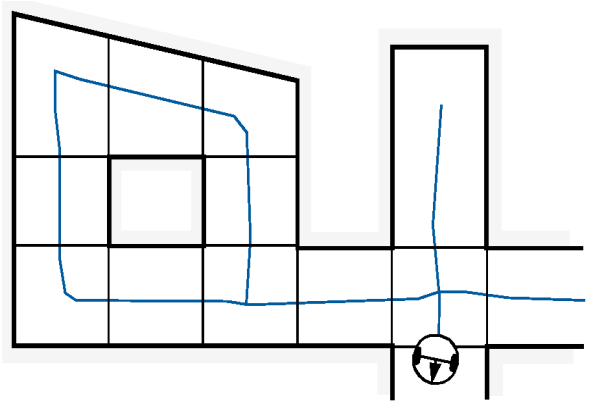


3. Graph Search

- Identify a set edges between nodes within the free space



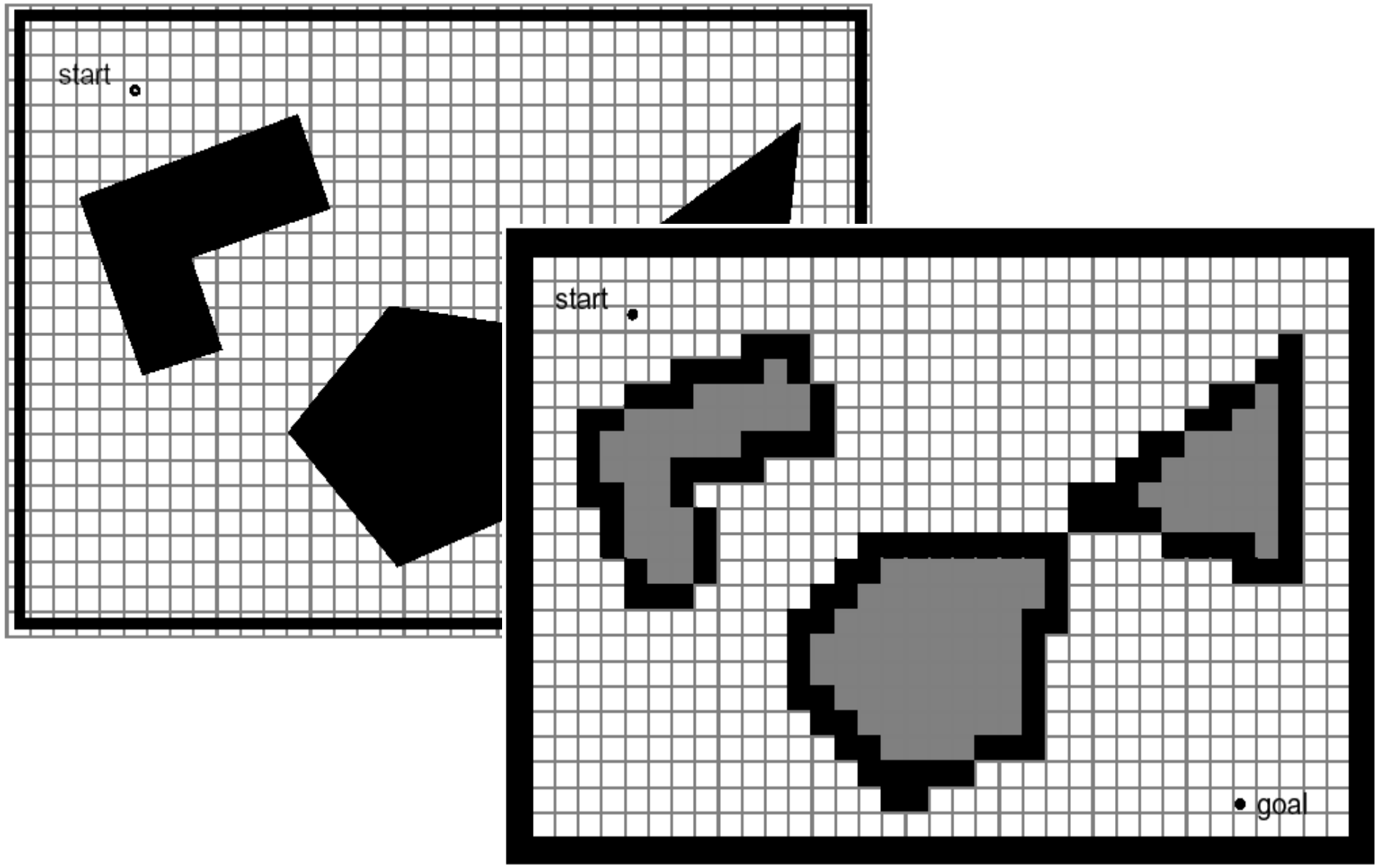
- Where to put the nodes?



Graph Search

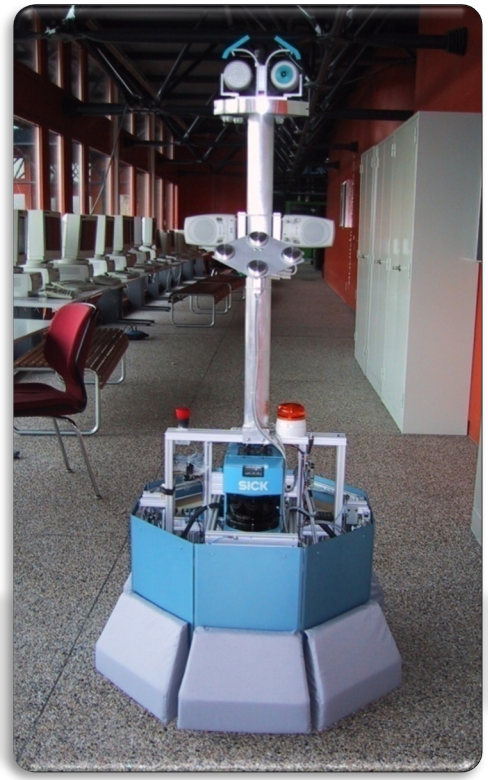
- Overview
  - Solves a least cost problem between two states on a (directed) graph
  - Graph structure is a discrete representation
- Limitations
  - State space is discretized → completeness is at stake
  - Feasibility of paths is often not inherently encoded
- Algorithms
  - (Preprocessing steps)
  - Breath first
  - Depth first
  - Dijkstra
  - A* and variants
  - D* and variants

Graph Construction: Cell Decomposition: Grid Map

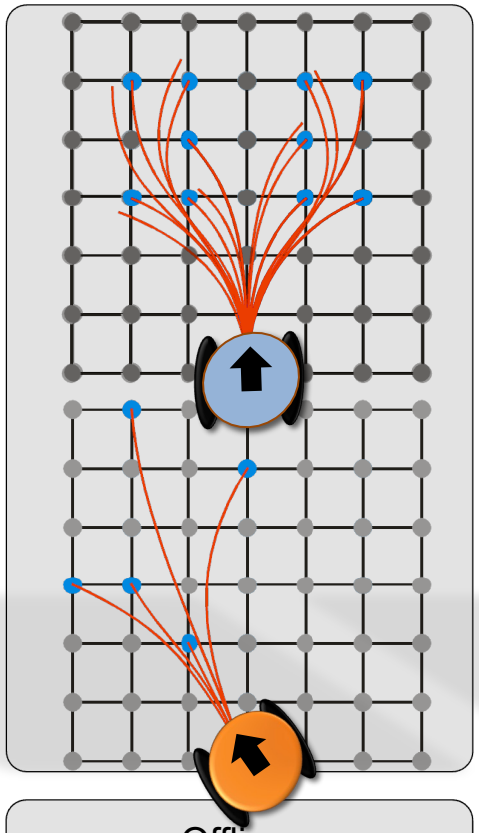


Graph Construction: State Lattice Design (1/2)

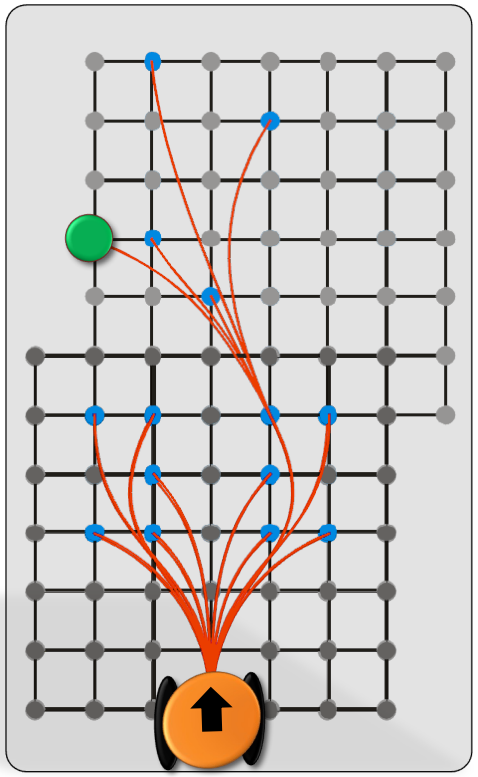
- Enforces edge feasibility
- Popular for Ackerman robots (e.g. cars)



Offline:
Motion Model



Offline:
Lattice Gen.

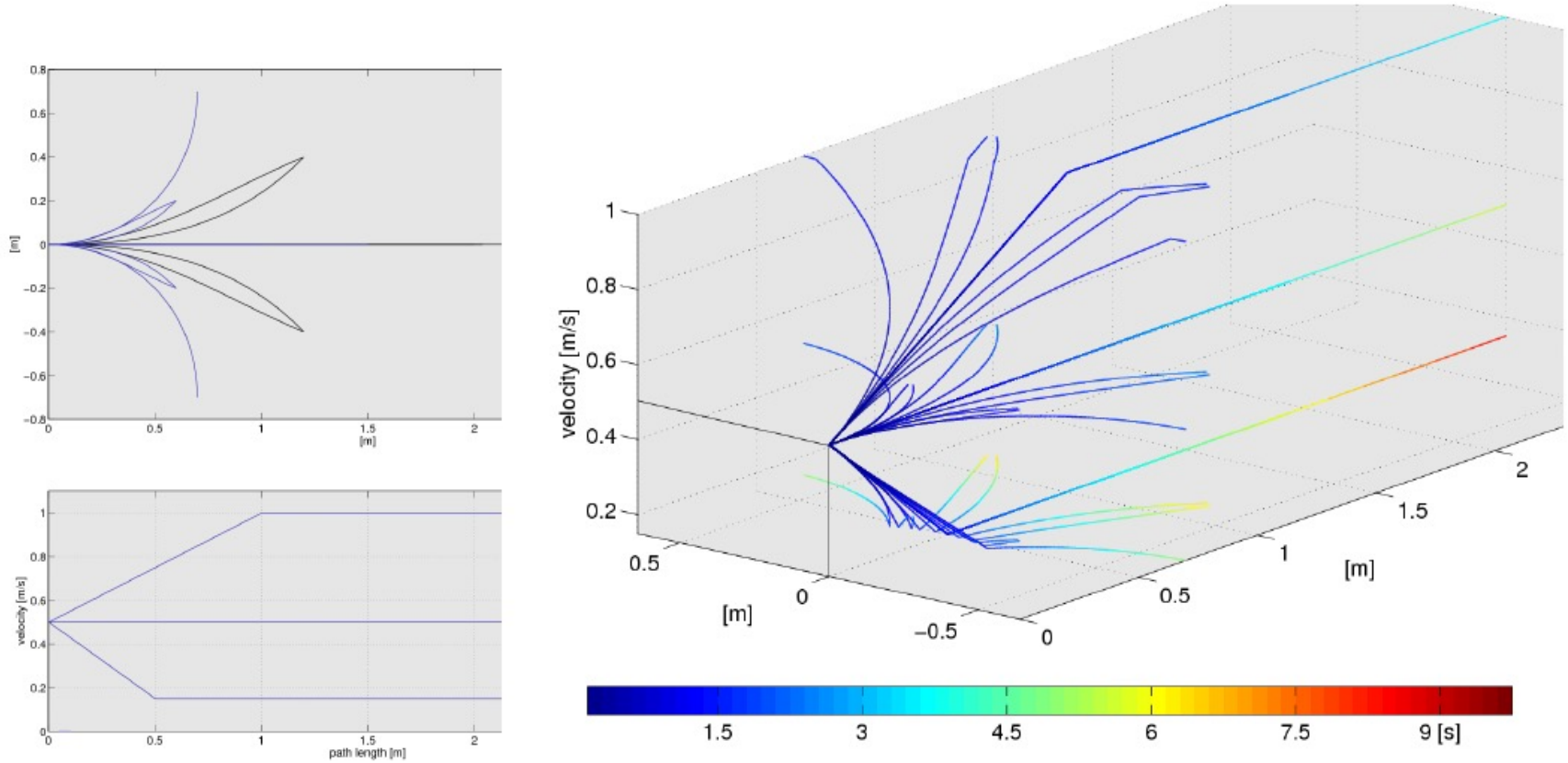


Online:
Incremental Graph
Constr.

Graph Construction: State Lattice Design (2/2)

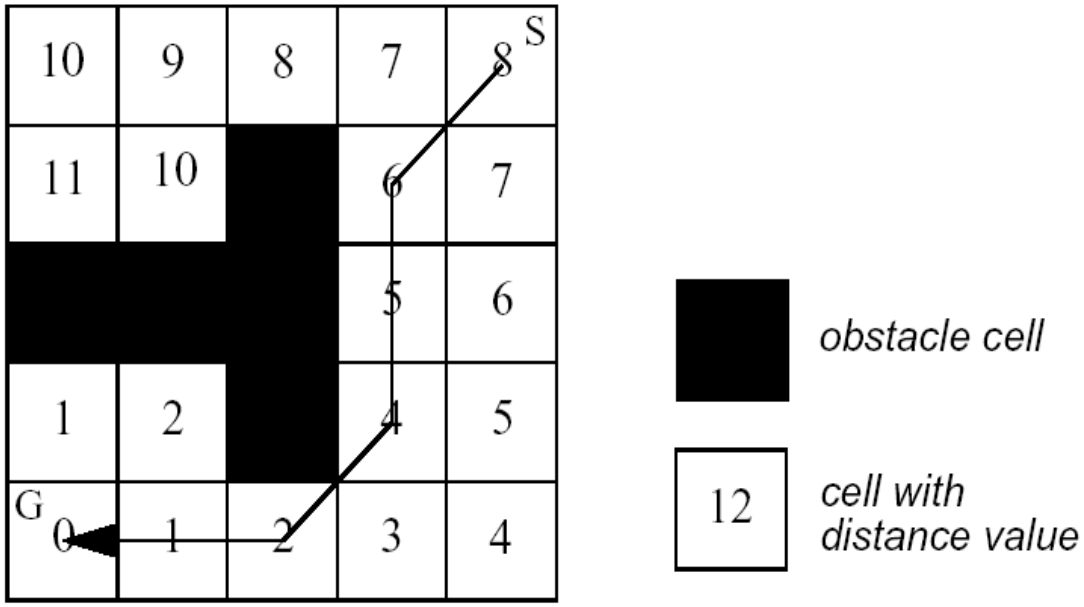
Martin Rufli

- State lattice encodes only kinematically feasible edges



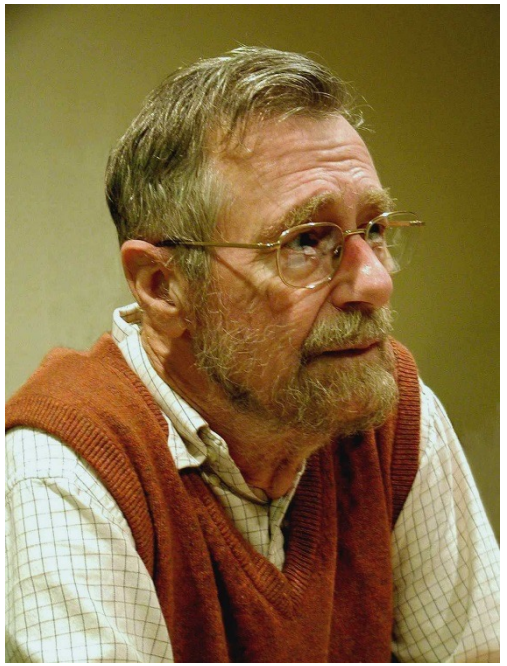
Deterministic Graph Search

- Methods
  - Breath First
  - Depth First
  - **Dijkstra**
  - A* and variants
  - D* and variants
  - ...



DIJKSTRA‘S ALGORITHM

EDSGER WYBE DIJKSTRA



1930 - 2002

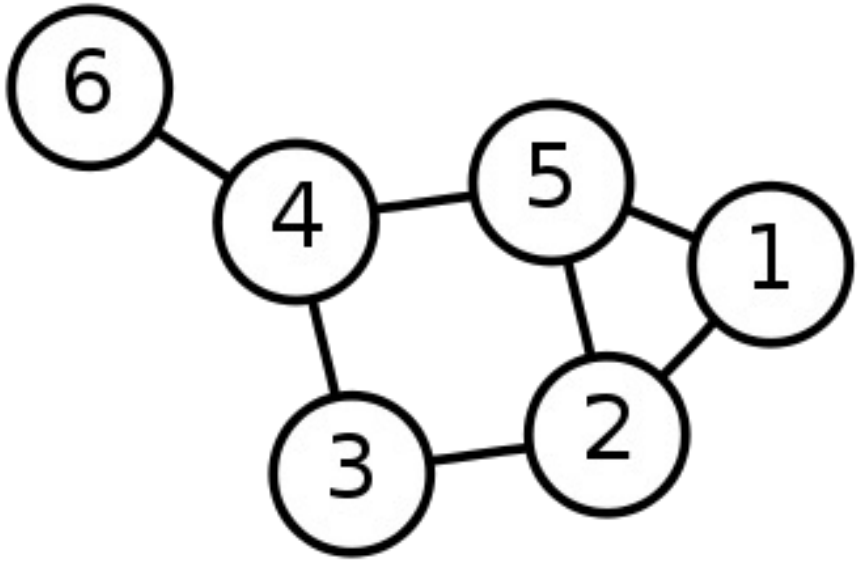
"Computer Science is no more about computers than astronomy is about telescopes."

http://www.cs.utexas.edu/~EWD/

SINGLE-SOURCE SHORTEST PATH PROBLEM

- **Single-Source Shortest Path Problem** - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.

- **Graph**
- Set of vertices and edges
- Vertex:
  - Place in the graph; connected by:
- Edge: connecting two vertices
  - Directed or undirected (undirected in Dijkstra's Algorithm)
  - Edges can have weight/ distance assigned



Dijkstra material from http://www.cs.utexas.edu/~tandy/barrera.ppt

Dijkstra's Algorithm

- Assign all vertices infinite distance to goal
- Assign 0 to distance from start
- Add all vertices to the queue

- While the queue is not empty:
  - Select vertex with smallest distance and remove it from the queue
  - Visit all neighbor vertices of that vertex,
  - calculate their distance and
  - update their (the neighbors) distance if the new distance is smaller

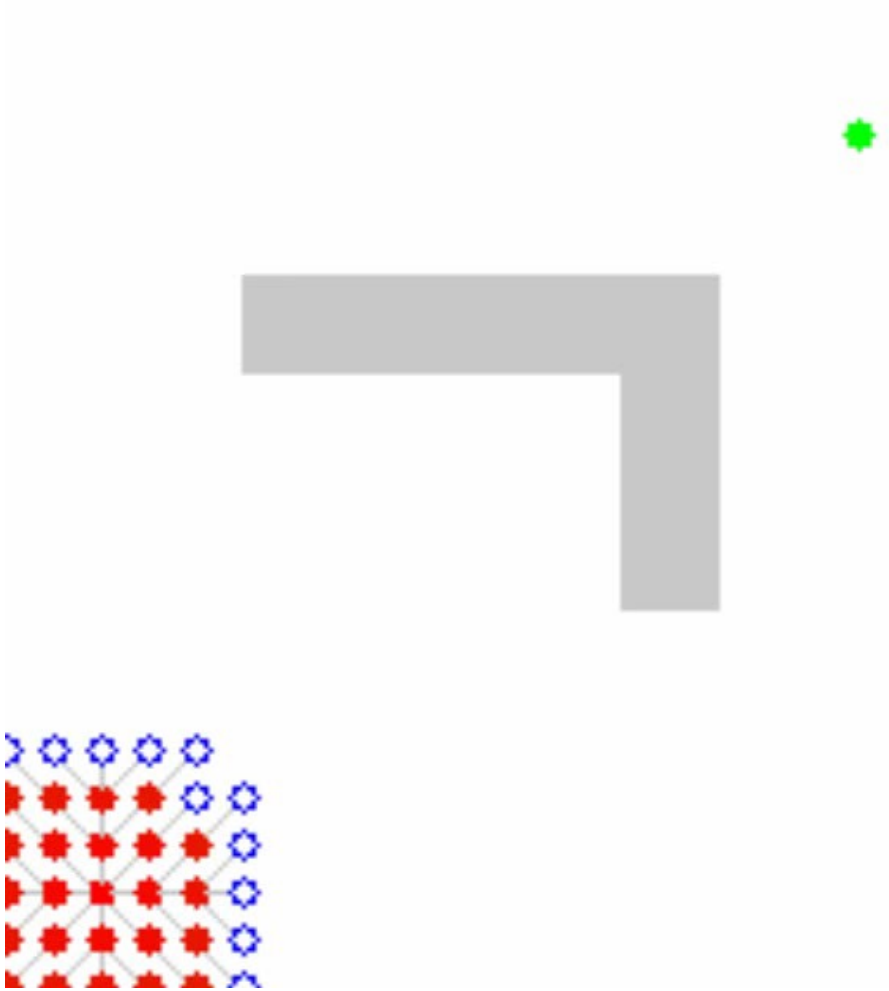
Dijkstra's Algorithm - Pseudocode

```
dist[s] ← 0                                (distance to source vertex is zero)
for all v ∈ V–{s}
      do dist[v] ← ∞                       (set all other distances to infinity)
S ← ∅                                      (S, the set of visited vertices is initially empty)
Q← V                                       (Q, the queue initially contains all vertices)
while Q ≠∅                                 (while the queue is not empty)
do  u ← mindistance(Q, dist)               (select the element of Q with the min. distance)
    S←S∪{u}                                (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do if  dist[v] > dist[u] + w(u, v)        (if new shortest path found)
                then    d[v] ←d[u] + w(u, v)        (set new value of shortest path)
        (if desired, add traceback code)
return dist
```

Dijkstra's Algorithm for Path Planning: Grid Maps

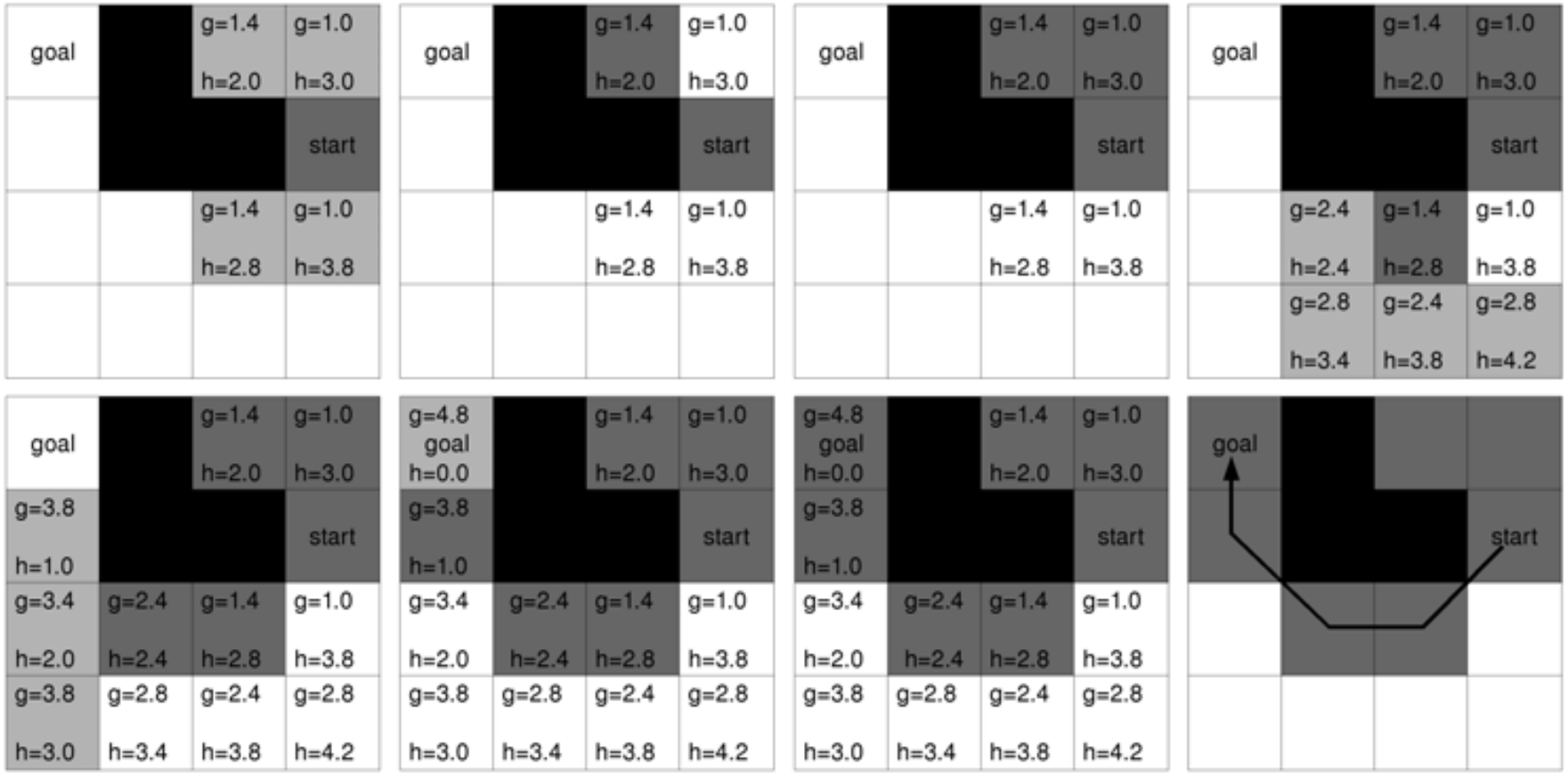
- Graph:
  - Neighboring free cells are connected:
    - 4-neighborhood: up/ down/ left right
    - **8-neighborhood**: also diagonals
  - All edges have weight 1

- Stop once goal vertex is reached
- Per vertex: save edge over which the shortest distance from start was reached => Path



Graph Search Strategies: A* Search

- Similar to Dijkstra‘s algorithm, except that it uses a heuristic function h(n)
- f(n) = g(n) + h(n)



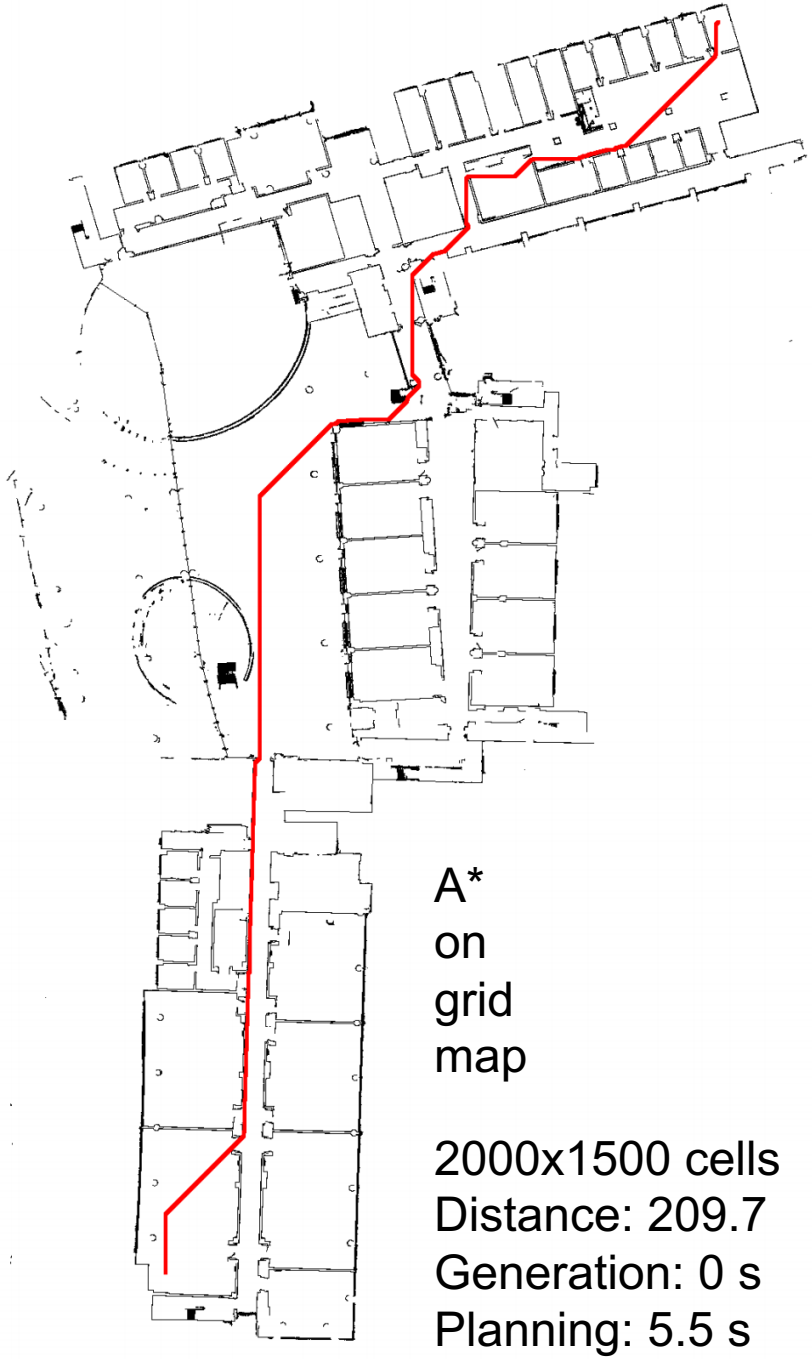
A*

- Developed 1986 as part of the Shakey project!
- Complexity:

Worst-case performance	$O(\|E\|) = O(b^d)$
Worst-case space complexity	$O(\|V\|) = O(b^d)$

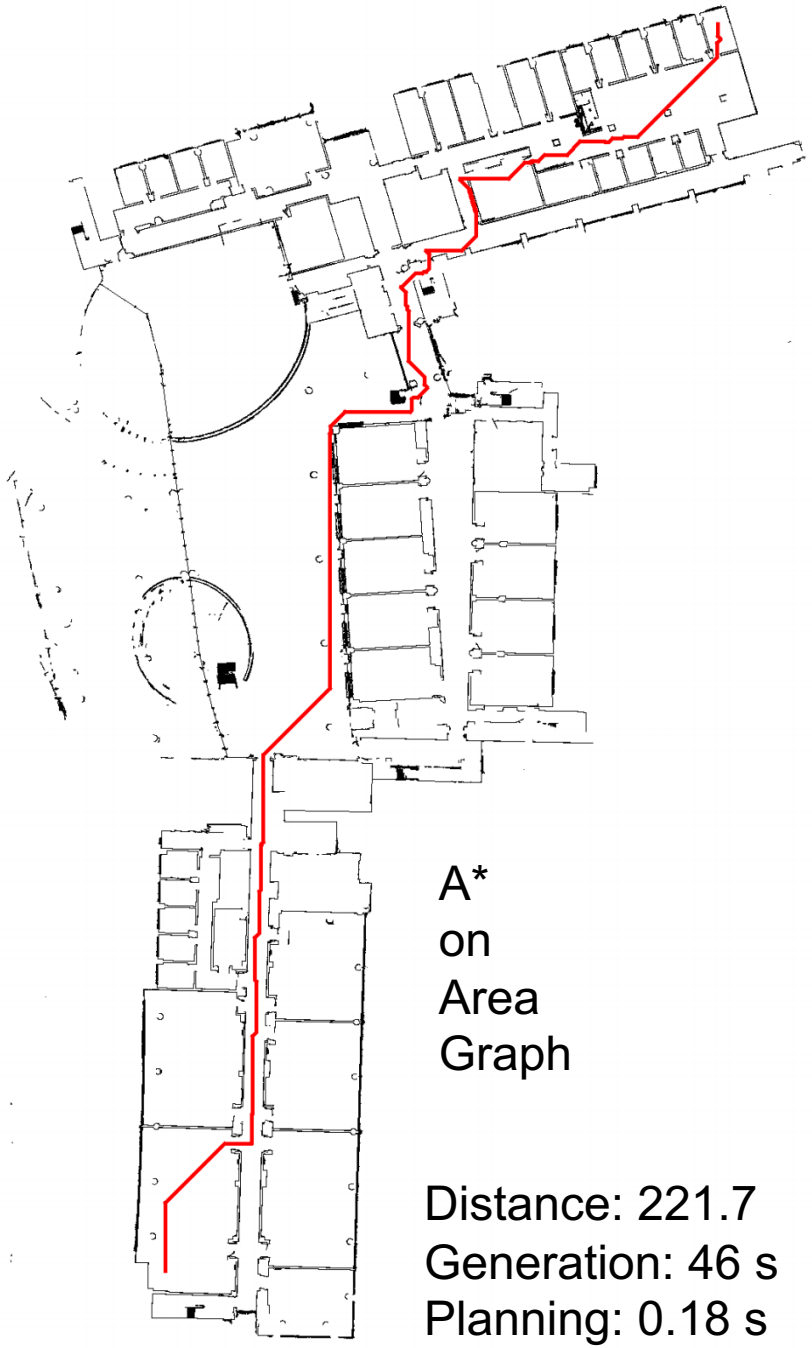
b: branching factor
d: depth

- Good heuristic => small branching factor



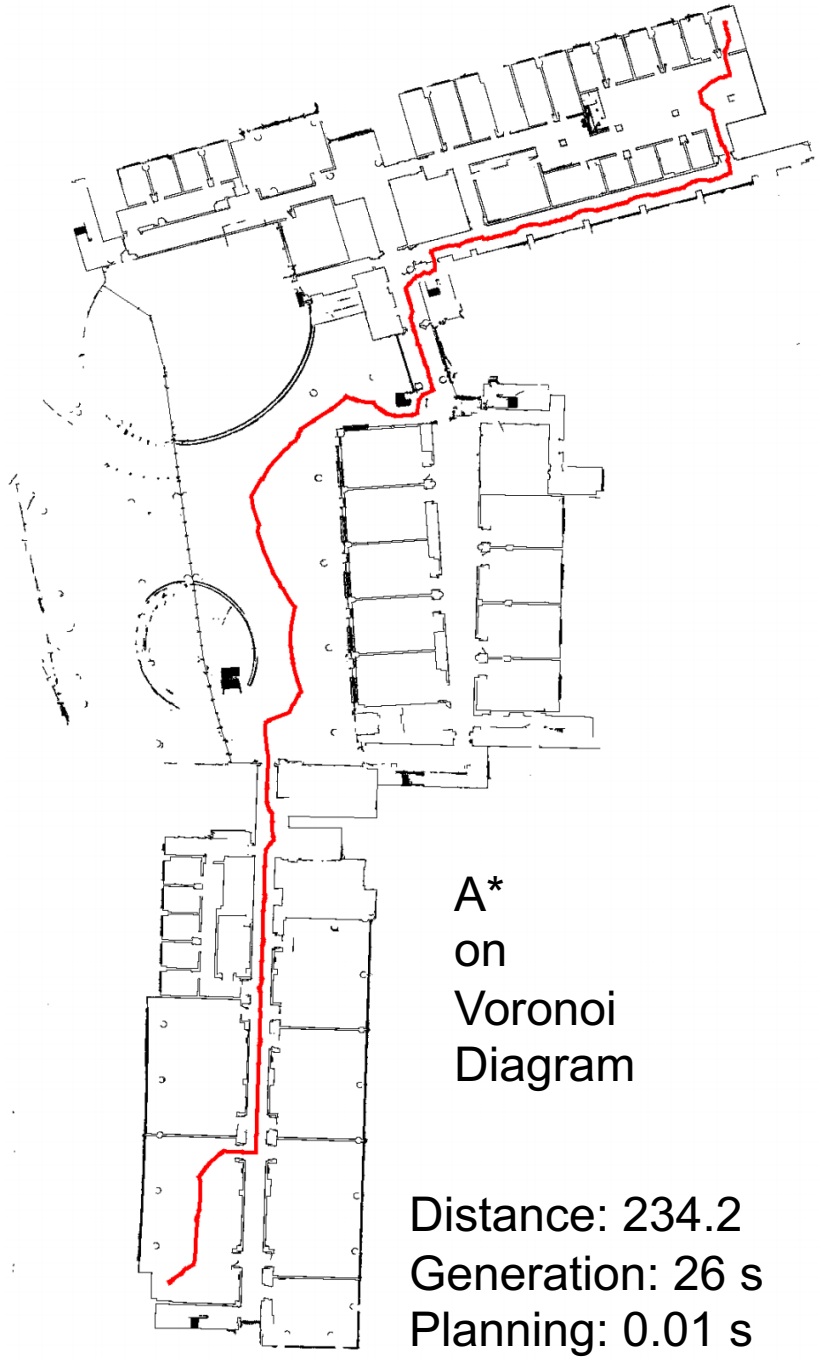
A*
on
grid
map

2000x1500 cells
Distance: 209.7
Generation: 0 s
Planning: 5.5 s



A*
on
Area
Graph

Distance: 221.7
Generation: 46 s
Planning: 0.18 s

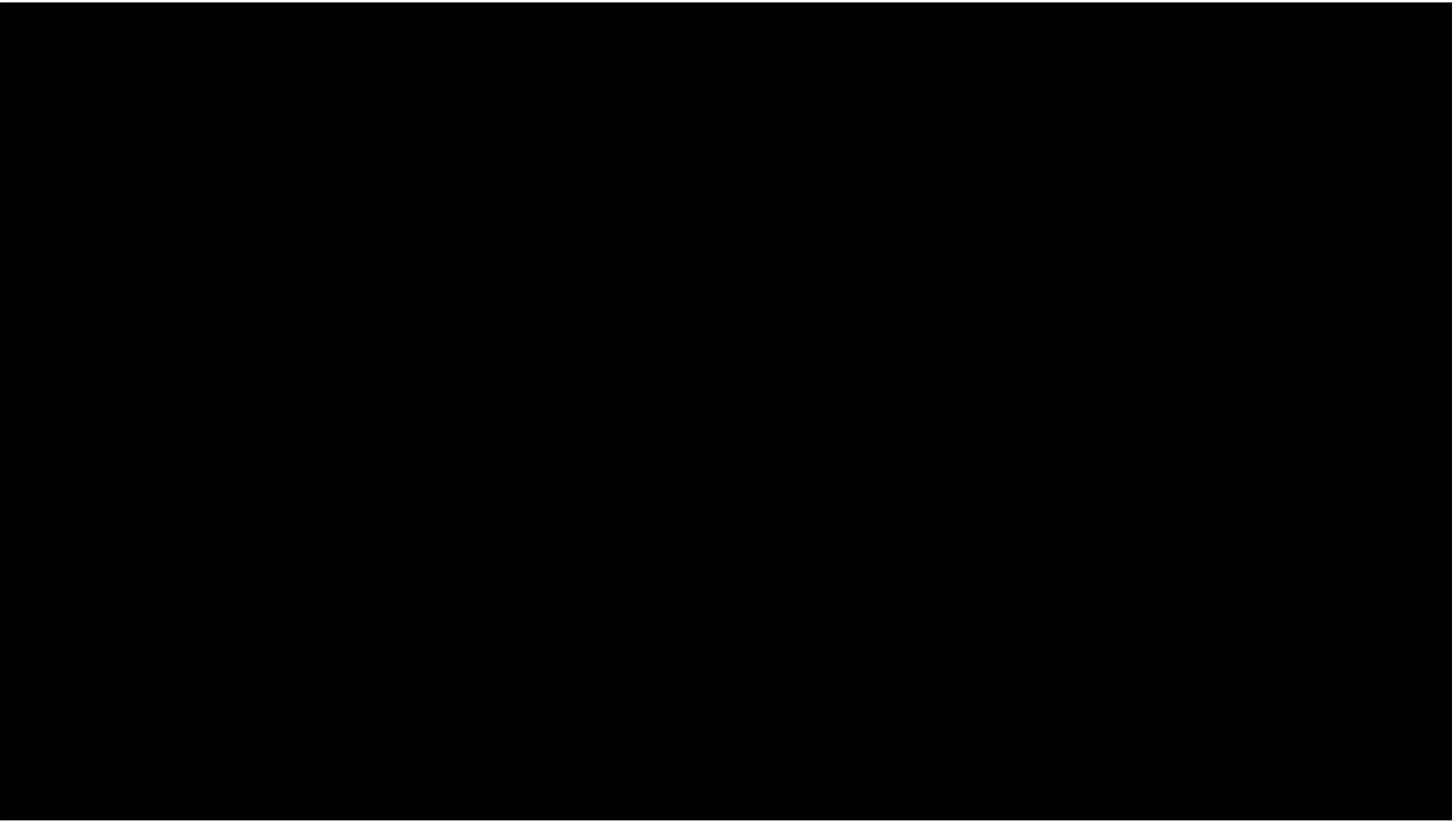


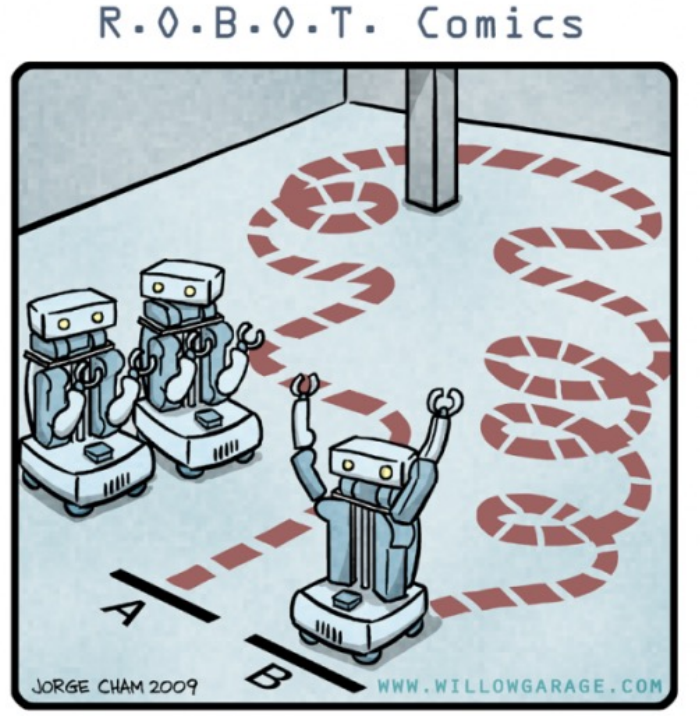
A*
on
Voronoi
Diagram

Distance: 234.2
Generation: 26 s
Planning: 0.01 s

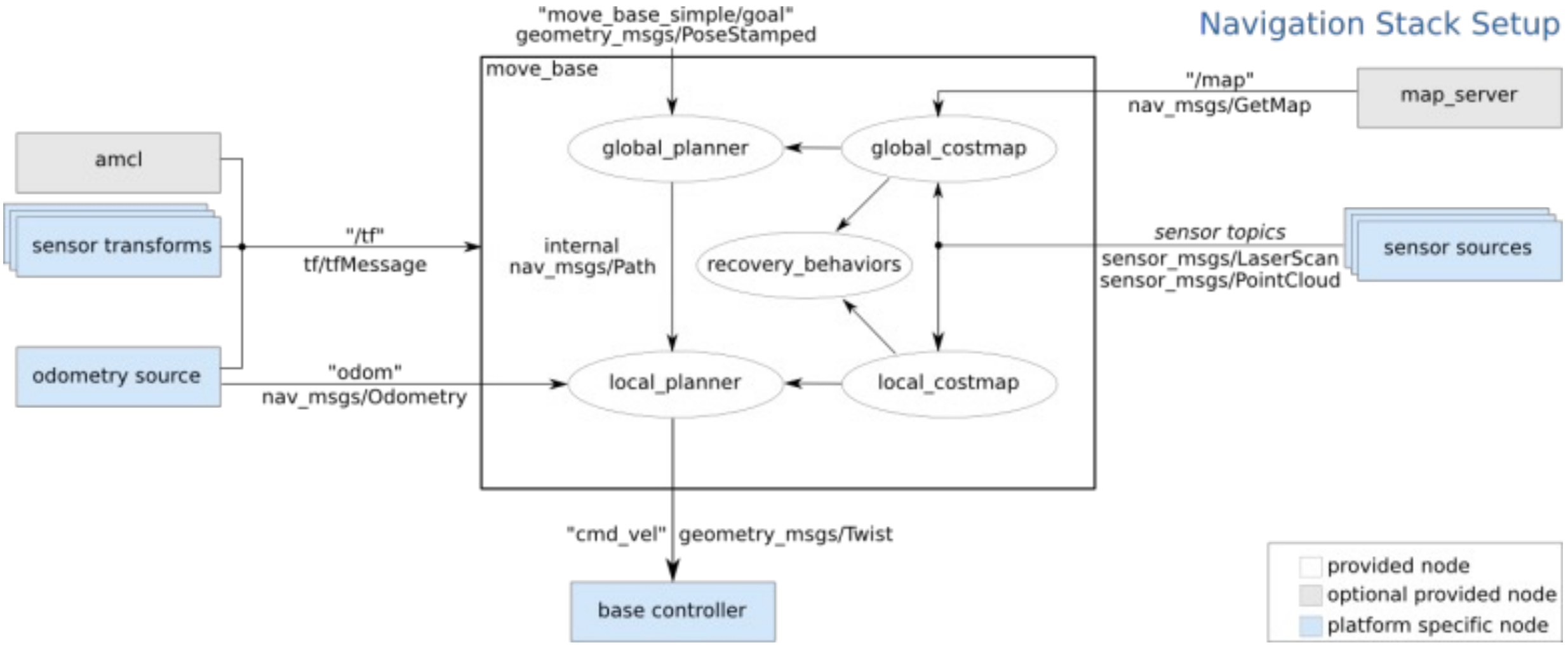
ROS Navigation

- http://wiki.ros.org/navigation





Path Planning in ROS: move_base



technische universität
dortmund

teb_local_planner

An optimal trajectory planner for mobile robots based on Timed-Elastic-Bands

http://wiki.ros.org/teb_local_planner