INFO0948
Control and Navigation of Mobile Robots

Bernard Boigelot
boigelot@montefiore.ulg.ac.be

February 23rd, 2018

These slides are partly based on Chapters 4 and 5 of the book *Robotics, Vision and Control: Fundamental Algorithms in MATLAB* by Peter Corke, published by Springer in 2011, on course material prepared by Renaud Detry in 2016, and on Stéphane Lens's PhD thesis (ULg, 2015).

Control and Navigation

Problem statement: How to drive a mobile robot so as to

- reach a goal,
- as efficiently as possible,
- while satisfying various constraints?

Illustration 1: Solving a maze.

https://www.youtube.com/watch?v=_9Y4ODmweYA

Illustration 2: Skid parking.

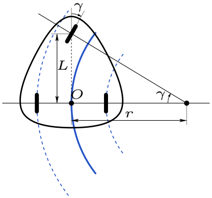
https://www.youtube.com/watch?v=_pi0849uRdI

Controlling a Mobile Robot

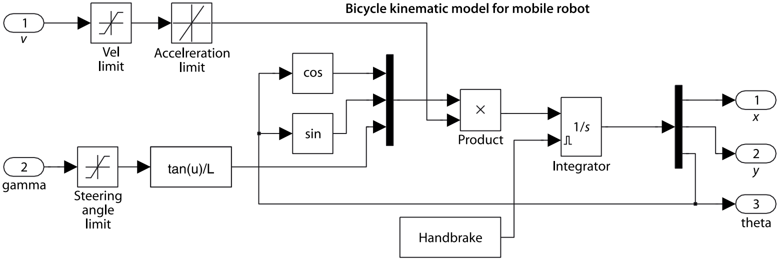
Approach:

1. Develop a *kinematic model* of the mobile robot.
2. Build a *control loop* around this model.

Kinematic Model: Bicycle Drive



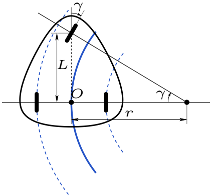
$$
\begin{aligned}
\dot{x} &= v\cos\theta \\
\dot{y} &= v\sin\theta \\
\dot{\theta} &= \frac{v}{L}\tan\gamma
\end{aligned}
$$



(Toolbox model: `Bicycle`.)

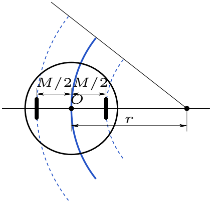
Kinematic Model: Other Platforms

Tricycle drive:



$$
\begin{aligned}
v &= v_S \cos\gamma \quad \Rightarrow \quad \begin{cases} \dot{x} = v_S \cos\gamma \cos\theta \\ \dot{y} = v_S \cos\gamma \sin\theta \end{cases} \\
\dot{\theta} &= \frac{v_S}{L} \sin\gamma
\end{aligned}
$$

Differential drive:



$$
\begin{aligned}
v &= \frac{v_R + v_L}{2} \quad \Rightarrow \quad \begin{cases} \dot{x} = \frac{v_R + v_L}{2} \cos\theta \\ \dot{y} = \frac{v_R + v_L}{2} \sin\theta \end{cases} \\
\dot{\theta} &= \frac{v_R - v_L}{2}
\end{aligned}
$$

Control Loop: Moving to a Point

Control strategy:

- The target velocity is proportional to the distance from the goal:

$$v^* = K_v \sqrt{\left(x^* - x\right)^2 + \left(y^* - y\right)^2}$$

- The steering angle γ is proportional to the angular difference between the direction to the goal and the current orientation θ:

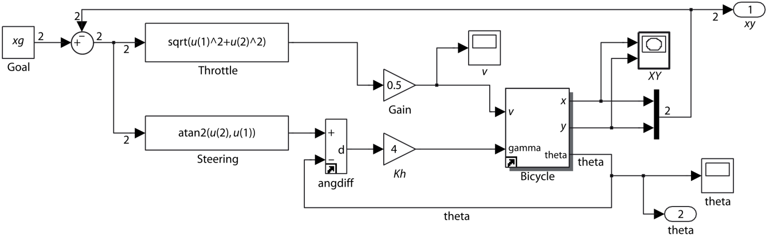
$$\gamma = K_h\left(\theta^* \ominus \theta\right),$$

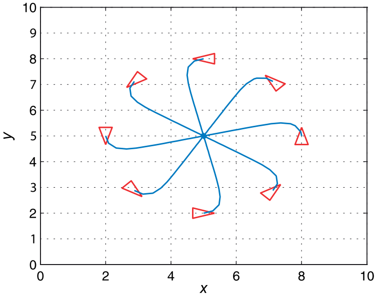
with

$$\theta^* = \arctan \frac{y^* - y}{x^* - x}.$$

(Matlab and toolbox functions: `atan2` for the four-quadrant arctan, `angdiff` for the bounded angle difference $\ominus$.)

Moving to a Point: Simulink Model





(Toolbox model: `sl_drivepoint`.)

Following a Line

Control strategy:

- The goal is to follow the line $ax + by + c = 0$.
- Steering controller 1: Steer towards the line:

  $$\alpha_d = -K_d d$$

  with

  $$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}.$$

- Steering controller 2: Keep our orientation parallel to the line:

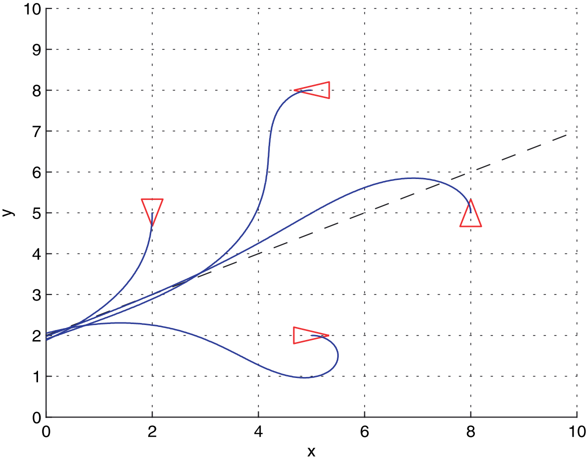
  $$\alpha_h = K_h\big(\theta^* \ominus \theta\big),$$

  with

  $$\theta^* = \arctan \frac{-a}{b}.$$

- Combined steering controller: $\gamma = \alpha_d + \alpha_h$.

Result:



(Toolbox model: `sl_driveline`.)

Following a Path

Control strategy:

- The robot follows a goal (x^*, y^*) that moves along a path.
- The distance d^* between the robot and the moving goal is kept constant by a velocity controller

  $$v^* = K_v e + K_i \int e \, dt$$

  with

  $$e = \sqrt{\left(x^* - x\right)^2 + \left(y^* - y\right)^2} - d^*.$$

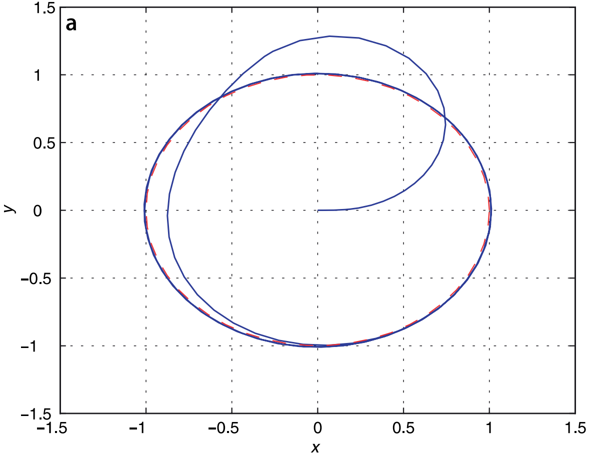
- The steering controller orients the robot towards the moving goal:

  $$\gamma = K_h \left(\theta^* \ominus \theta\right),$$

  with

  $$\theta^* = \arctan \frac{y^* - y}{x^* - x}.$$

Result:



http://www.montefiore.ulg.ac.be/~boigelot/tunnel/bull.mov

(Toolbox model: sl_pursuit.)

Reactive Navigation 1: Braitenberg Vehicles

Principles:

- Direct connection between sensors and actuators.
- No internal memory.
- No internal representation of the environment.

Example:

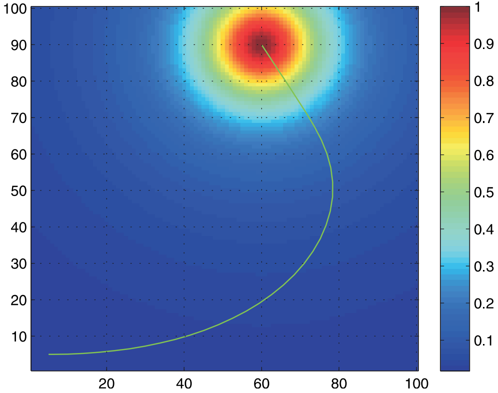
- Two sensors s_R and s_L returning values in $[0, 1]$. The goal is to reach the location where $s_R = s_L = 1$.
- Velocity law:

$$v = 2 - s_R - s_L.$$

- Steering law:

$$\gamma = k(s_L - s_R).$$

Result:



Notes:

- The command strategy remains simple.
- With additional sensors, more complex behaviors can be implemented (e.g., obstacle avoidance).
- Toolbox model: `sl_braitenberg`

Reactive Navigation 2: Simple Automata

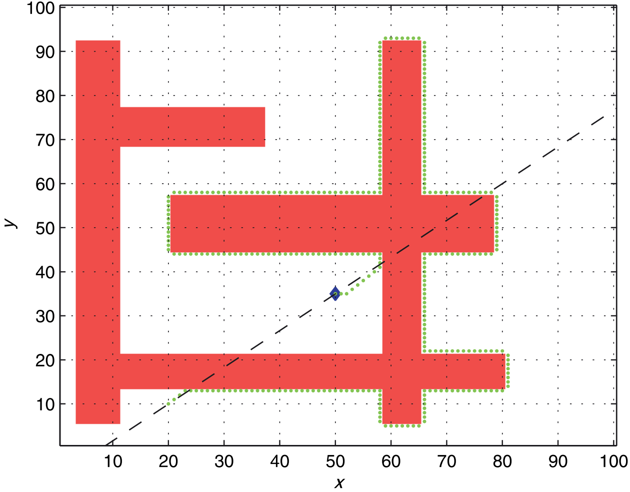
Principles:

- The command logic is implemented as a state machine.
- At each step, the current state and the sensor values determine
  - the immediate motion of the robot, and
  - the next state.

Example:

- Bug robot operating in a grid world.
- The basic mode of operation is to move in a straight line towards the goal.
- If an obstacle is detected, the bug moves around it (counter-clockwise), until it reaches a point on the original line that is closer to the goal.

Result:



This solution is far from being optimal!

(Toolbox model: Bug2.)

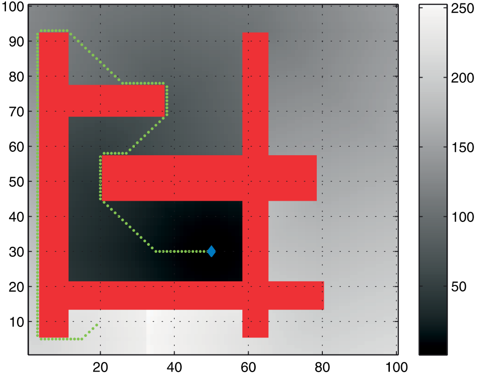
Map-based Planning

- The robot uses a *map* of its environment to plan its paths.
- The main problem is to find a path from a location to another that
  - is physically feasible (in particular, it must avoid obstacles), and
  - minimizes a cost function (traveled distance, time, energy, ...).
- In some applications, the parameters of the problem (initial location, goal, constraints, map contents, ...) may change over time.

Distance Transform

- Simple model where the robot
  - occupies one cell in a grid world,
  - knows precisely its position, and
  - moves in a holonomic way.
- The map labels each cell with its precomputed distance to the goal.
- A simple strategy thus consists in always moving to the neighboring cell for which the distance to the goal is minimal.

Illustration:



- With this solution, the initial location can easily be modified.
- However, a new map has to be computed for every new goal.
- The distance transform computation algorithm implemented in the toolbox (`DXform`) is inefficient, but there exist better solutions.

Graph-based Planning

- The reachable locations are represented by the nodes of a graph (e.g., every free cell in a grid world).
- The graph contains an edge (n, n') whenever n' is directly reachable from n.
- Edges are labeled by the cost of the corresponding move (e.g., 1 for horizontal or vertical neighbors in a grid, $\sqrt{2}$ for diagonal ones).
- The problem is to find a path from an initial node n_0 to a goal n_G that minimizes the total cost of its edges.

Dijkstra's Algorithm

- For each node n, one keeps the current best estimate $g(n)$ of the minimum cost from n_0 to n.
- One maintains a set *OPEN* containing the nodes that still need to be processed.
- Initially:
$$\begin{array}{rcl} g(n_0) & = & 0 \\ g(n) & = & +\infty \text{ for all } n \neq n_0 \\ OPEN & = & \text{the set of all nodes.} \end{array}$$
- While $OPEN \neq \emptyset$:
  1. Remove from $OPEN$ the node n with the smallest $g(n)$.
  2. For each neighbor n' of n, if $g(n) + cost(n, n') < g(n')$, then set $g(n') := g(n) + cost(n, n')$.

Notes:

- Upon completion, $g(n)$ contains the smallest cost from n_0 to n, for every node n. (Thus, changing goals are easily dealt with.)
- In order to compute shortest paths, a simple approach is to keep a *backpointer* in each node, linking to its best predecessor.
- This algorithm runs in $O(N \log N)$ time, where N is the number of nodes, if cleverly implemented (priority queue for the set *OPEN*).

A* Algorithm

- Variant of Dijkstra's, in which one considers at each step the node n with the smallest value of

$$g(n) + h(n),$$

  where $h(n)$ is a *heuristic function* that estimates the cost from n to the goal node n_G.

- If $h(n)$ is always lower than or equal to the true cost of moving from n to n_G, then the algorithm is always able to compute the shortest path from n_0 to n_G, in $O(N \log N)$ time.
- Depending on the quality of the heuristic function h, this computation can be much faster.
- A simple choice for h is to use the Euclidean distance between node locations.

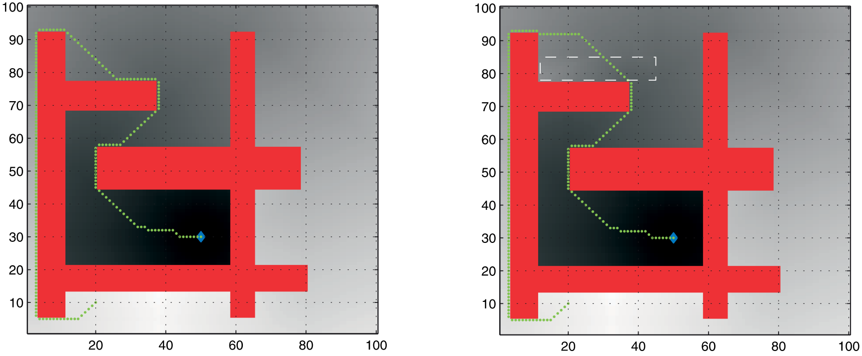
Drawbacks: The A* algorithm cannot easily deal with

- modifying the goal or the initial node.
- changing edge costs.

D* Algorithm

- Yet another variant of Dijkstra's.
- Instead of computing for each node n the best cost from n_0 to n, one computes the smallest cost from n to n_G. (In other, words, the algorithm computes a distance transform.)
- The algorithm supports incremental replanning: The cost of an edge can be modified at any time, leading to propagating the change to the relevant subset of nodes.
- The time cost is $O(N \log N)$ without replanning. Replannings have a worst-case cost of $O(N \log N)$, but are usually much cheaper.
- Toolbox implementation: `Dstar`.

Illustration:



(The cost of the edges in the dashed rectangle have been increased.)

Notes: The D* algorithm

- is still unable to handle changing goals, and
- lacks a heuristic function, and can thus be less efficient than A^* in some cases.

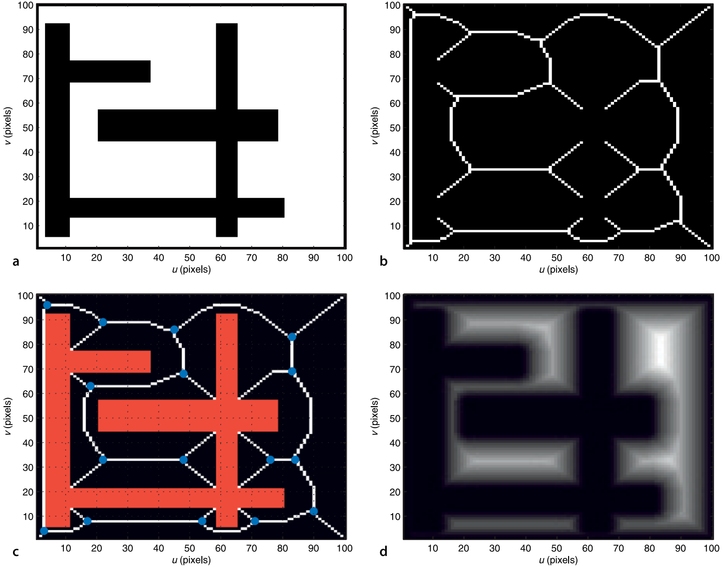
Voronoi Roadmaps

Goal: Handling efficiently queries in which the initial and goal locations are frequently modified.

Idea:

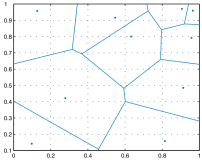
- Precompute a graph of the paths that clear the obstacles at the largest possible distance.
- Connect the initial and goal locations to the nearest nodes in this graph, and compute the shortest path between them.
- Perform local optimization on the resulting path.

Illustration:



Notes:

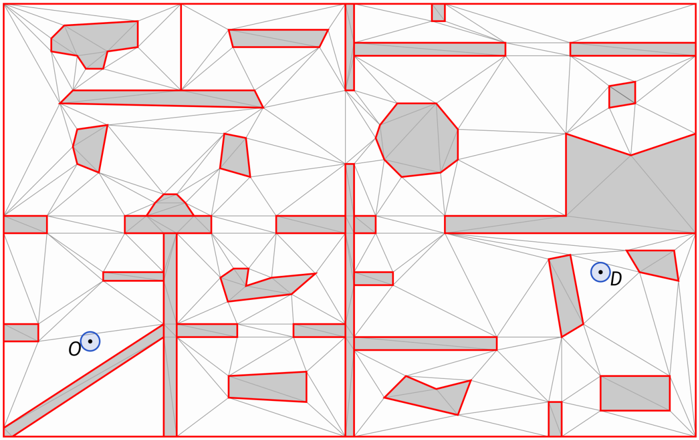
- The main advantage is that the resulting roadmap is much smaller than the graph linking all feasible locations.
- The reference book completely misses the fact that Voronoi roadmaps can be constructed and exploited in a very efficient way:
  - The *Voronoi diagram* of N points can be computed in $O(N \log N)$ time (and is the dual graph of their *Delaunay triangulation*).



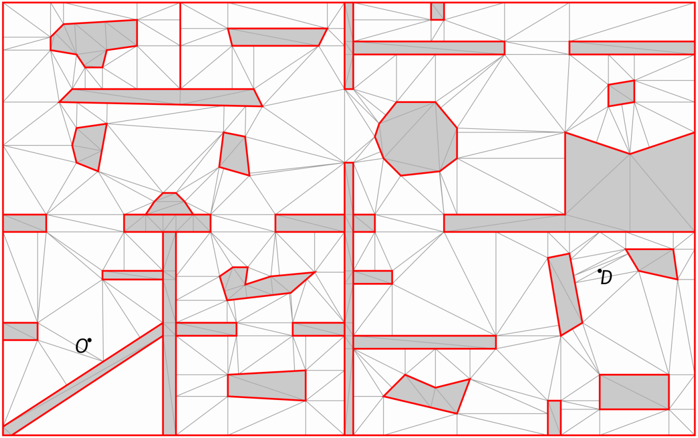
  - For a set of polygonal obstacles, the procedure is a bit more complex, but still runs in $O(N \log N)$ time.
  - Once a shortest path has been extracted from a Voronoi roadmap, it can be simplified in $O(N)$ time into a locally optimal solution.

Illustration (Stéphane Lens's thesis):

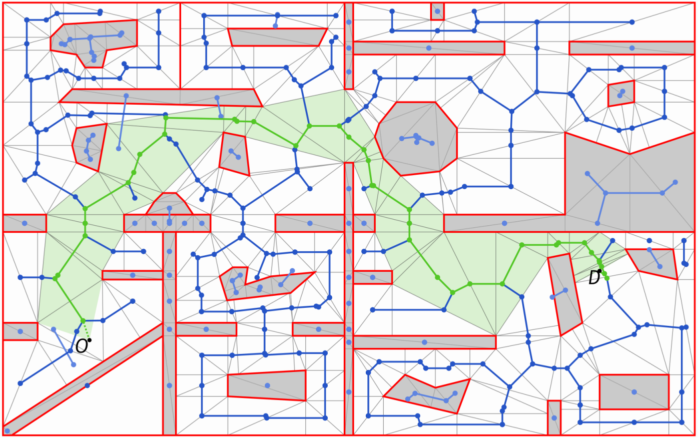
1. Problem statement and initial triangulation.



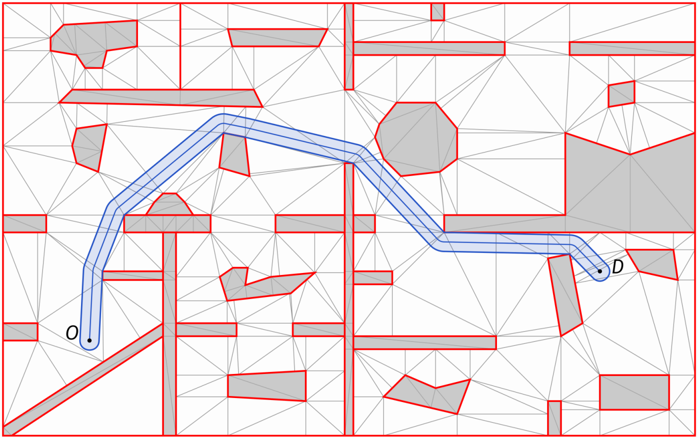
2. Refined triangulation.



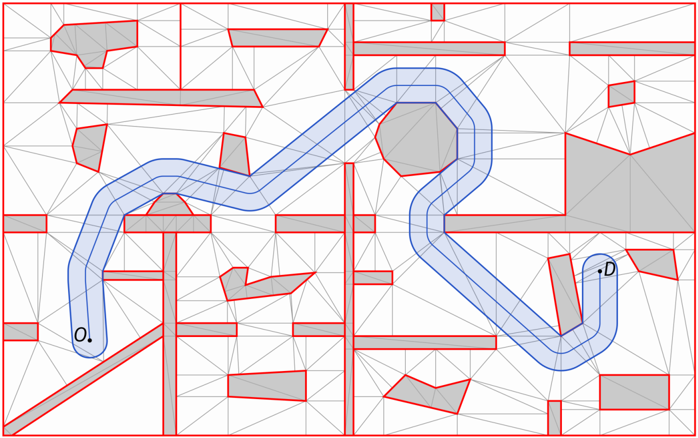
3. Roadmap graph and shortest path.



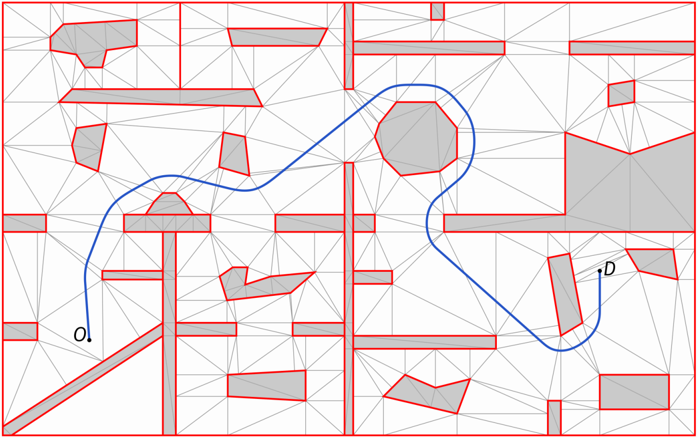
4. Locally optimal solution.



5. Solution with larger clearance.

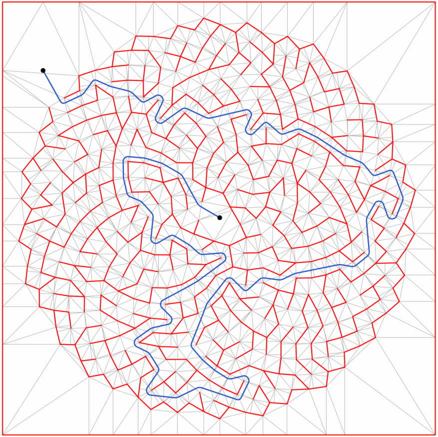


6. Smoothed path.



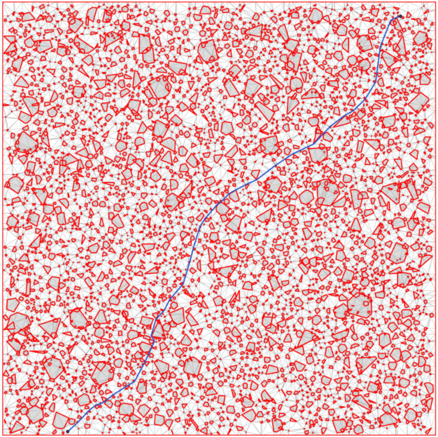
(Total computation time: 1.5 ms.)

A more complex example:



(Total computation time: 4.8 ms.)

Yet another complex example:



(Total computation time: 82.6 ms.)

Example: Wilbur (Eurobot)

`http://www.montefiore.ulg.ac.be/~boigelot/tunnel/wilbur.mov`

Probabilistic Roadmaps

Procedure:

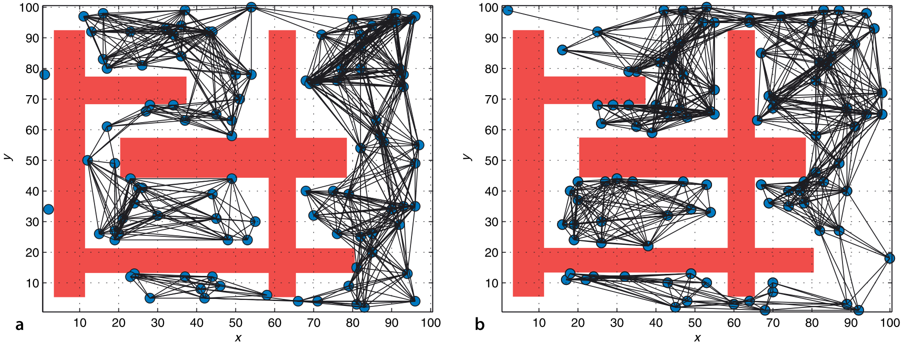
- Build a graph by placing N random points in free space, and connecting them together (making sure that edges do not cross obstacles).
- Set the cost of edges according to the distance between their corresponding nodes.
- Connect the initial and goal locations to their nearest node, and compute the shortest path between them.

Advantage: The method is simple and efficient.

Drawback: It is difficult to ensure that

- all areas of interest are explored, and
- the resulting roadmap is a connected graph.

Illustration:



(Toolbox implementation: PRM.)

Rapidly-exploring Random Trees (RRT)

Procedure: Build a roadmap tree by repeatedly

- placing a random point p' in free space,
- locating the nearest point p in the existing tree, (initially, this tree only contains the initial location),
- simulating a move of the robot from p to p', stopping after a given time or traveled distance,
- connecting p to the reached location p''.

Advantages:

- This approach can take into account constraints on robot motion (e.g., non-holonomicity).
- The procedure is also applicable to n-dimensional planning problems.

Illustration: