Introduction to Planning

Robotics Systems Science 2018

Prof. Yvan Petillot & Yaniel Carreno

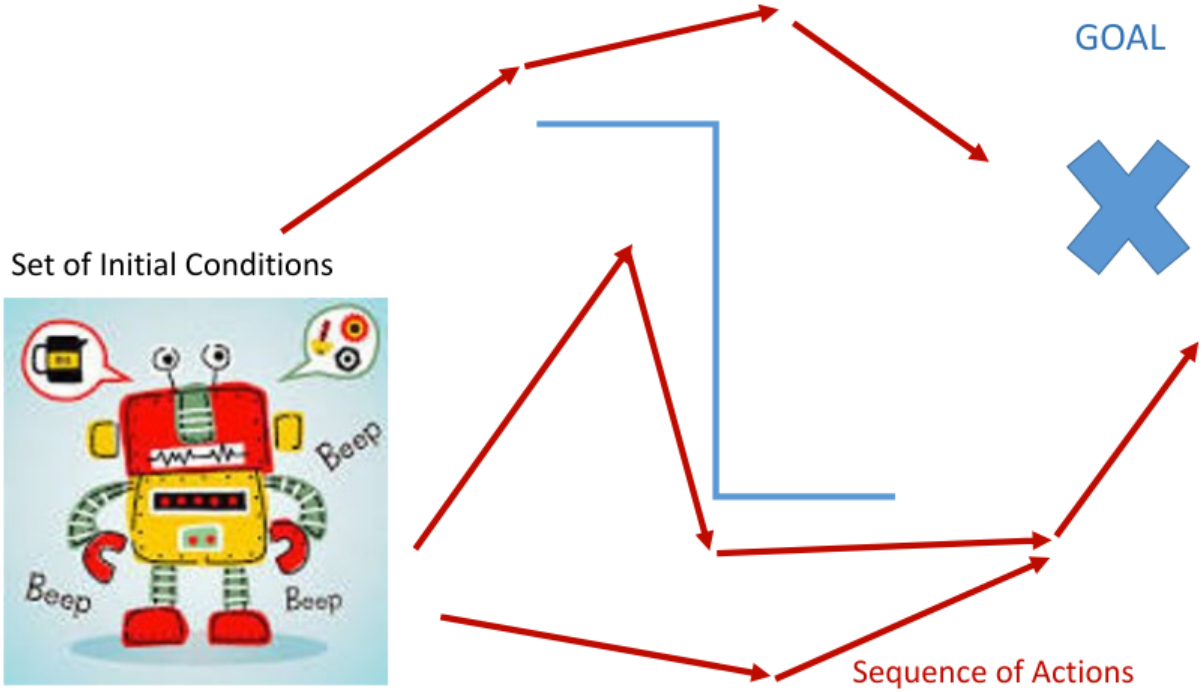
School of Engineering and Physical Sciences
Heriot-Watt University

Lecture Overview

- Planning definition and examples
- Low Level Planning Strategies
  - Motion Planning:
    - Configuration Space
    - Combinatorial Planning
    - Sampling-Based Planning
    - Potential Field (PF)
    - A* & Dijkstra's algorithm
    - Dynamic Window Approach (DWA)
- Task Planners
  - Classical Planners
  - Temporal Planners

Planning Definition

Planning can be defined as the sequence of actions that a robot or a group of robots have to take to achieve particular goals, considering the initial conditions

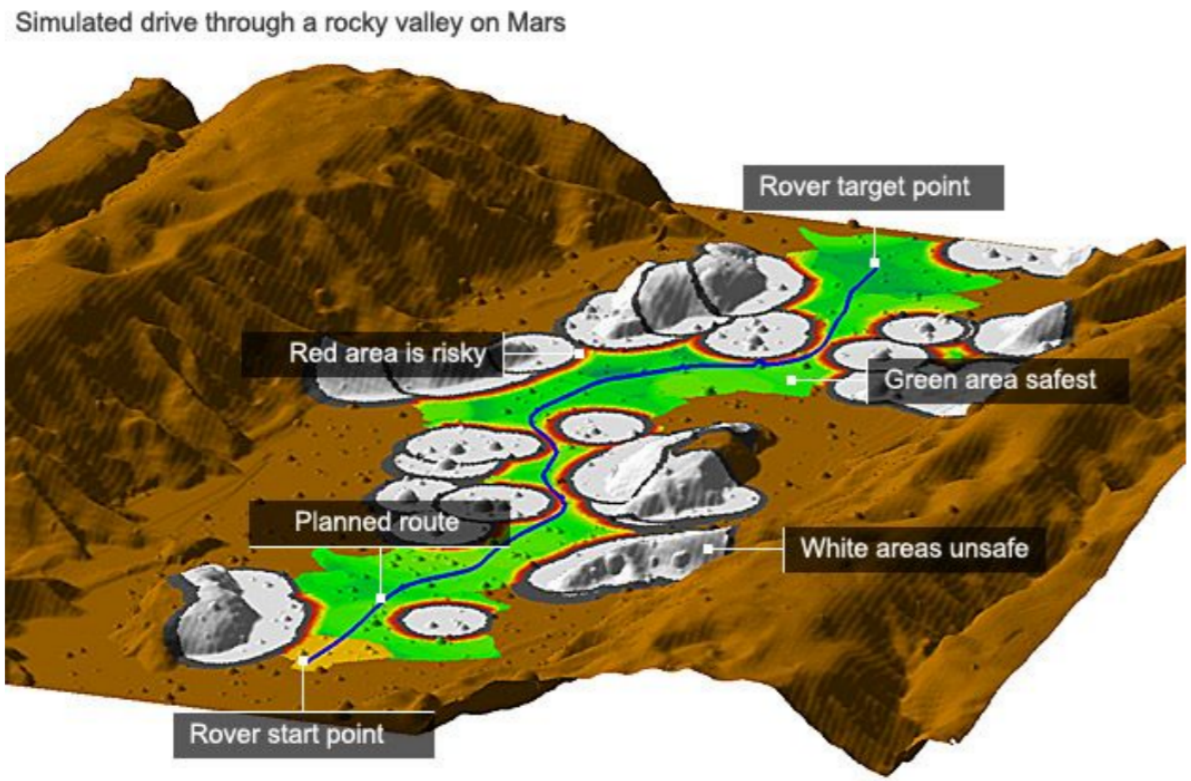


Motion Planning definition

- Goals.
  - Collision-free trajectories.
  - Robot should reach the goal location as fast as possible
- 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 description of the world
- Find a path that moves the robot gradually from start to goal while never colliding with any obstacle.

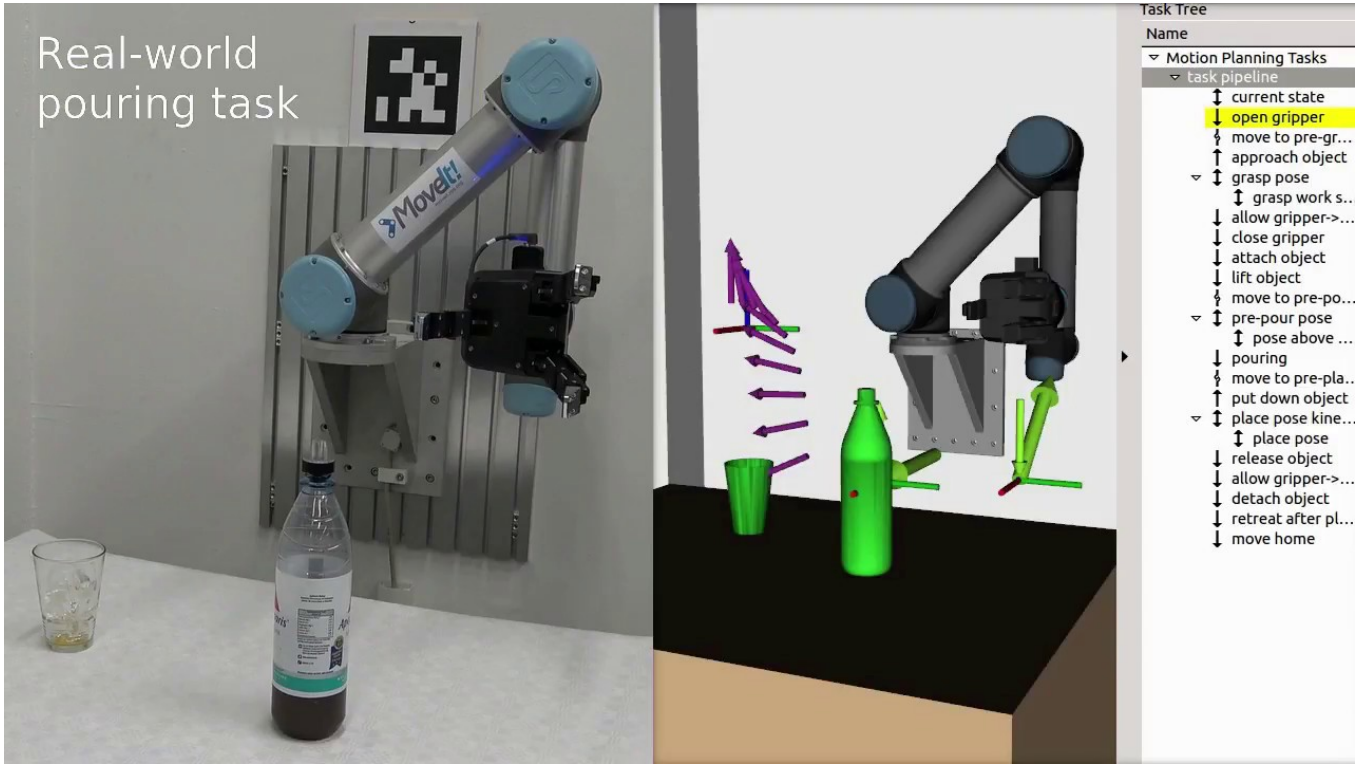
Examples

Mars mission implementation using Rovers



Examples

Grasping and positioning of objects



Examples

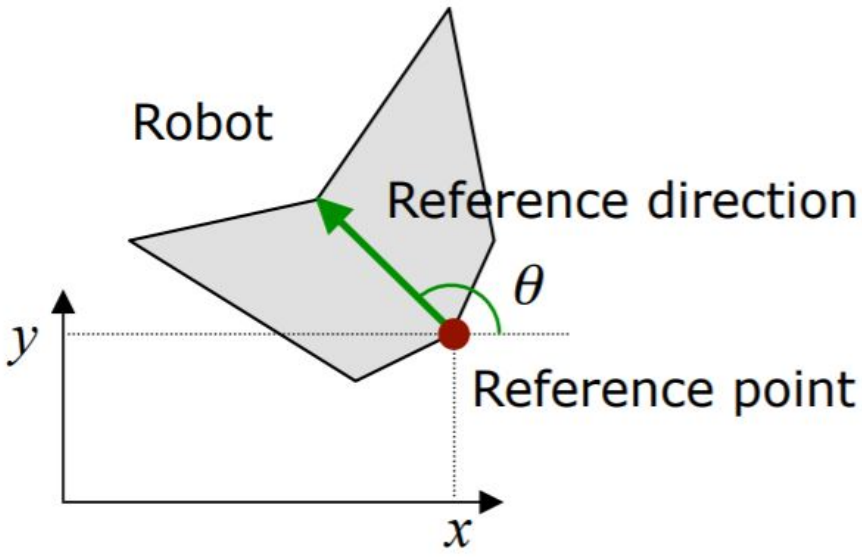
Planning systems in industrial environments such as warehouse

Industrial Example: STAMINA Project

Video From: STAMINA European Project

Configuration Space

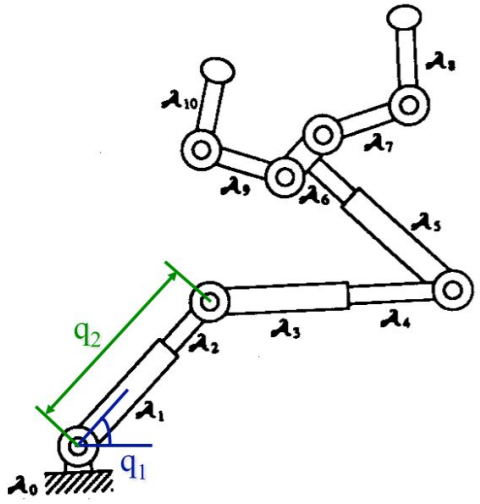
- Although the motion planning problem is defined in the regular world, it lives in another space: the configuration space.
- robot configuration q is a specification of the positions of all robot points relative to a fixed coordinate system.
- Usually a configuration is expressed as a vector of positions and orientations.



- Rigid-body robot example.
- 3-parameter representation: $q = (x, y, \theta)$
- In 3D, q would be of the form $(x, y, z, \alpha, \beta, \gamma)$

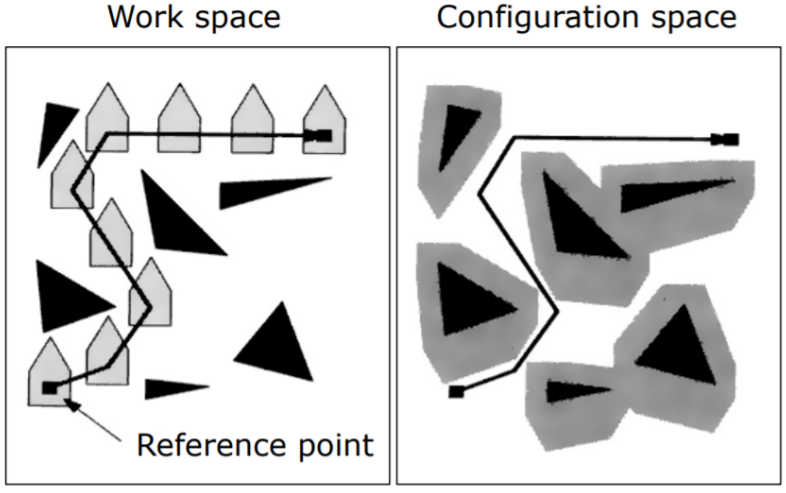
Configuration space Example

Articulated robot example



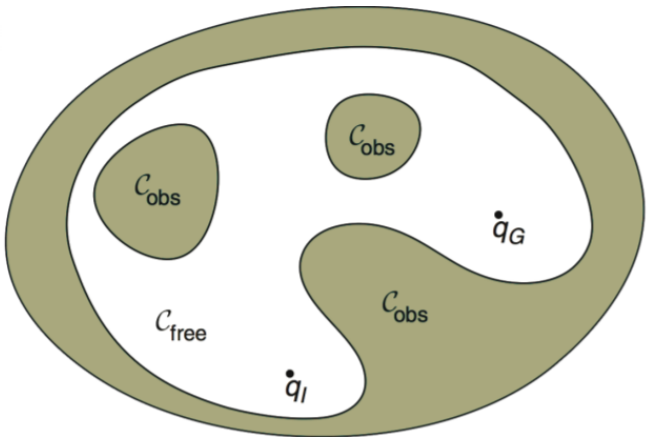
$$q = (q1, q2, \ldots, q10)$$

Polygonal robot, translation only



Configuration space is obtained by sliding the robot along the edge of the obstacle regions.

Configuration Space



- With $W = R^m$ being the work space.
- $O \epsilon W$ the set of obstacles.
- $A(q)$ the robot in configuration $q \epsilon C$.
- $C_{free} = \{q \epsilon C \mid A(q) \cap O = \emptyset\}$
- $C_{obs} = C / C_{free}$
- q_I & q_G are the start and goal configurations.

- For a continuous path $\tau : [0, 1] \rightarrow C_{free}$ with $\tau(0) = q_I, \tau(1) = q_G$
- Given this setting, we can do planning with the robot being a point in configuration space.

Motion Planning

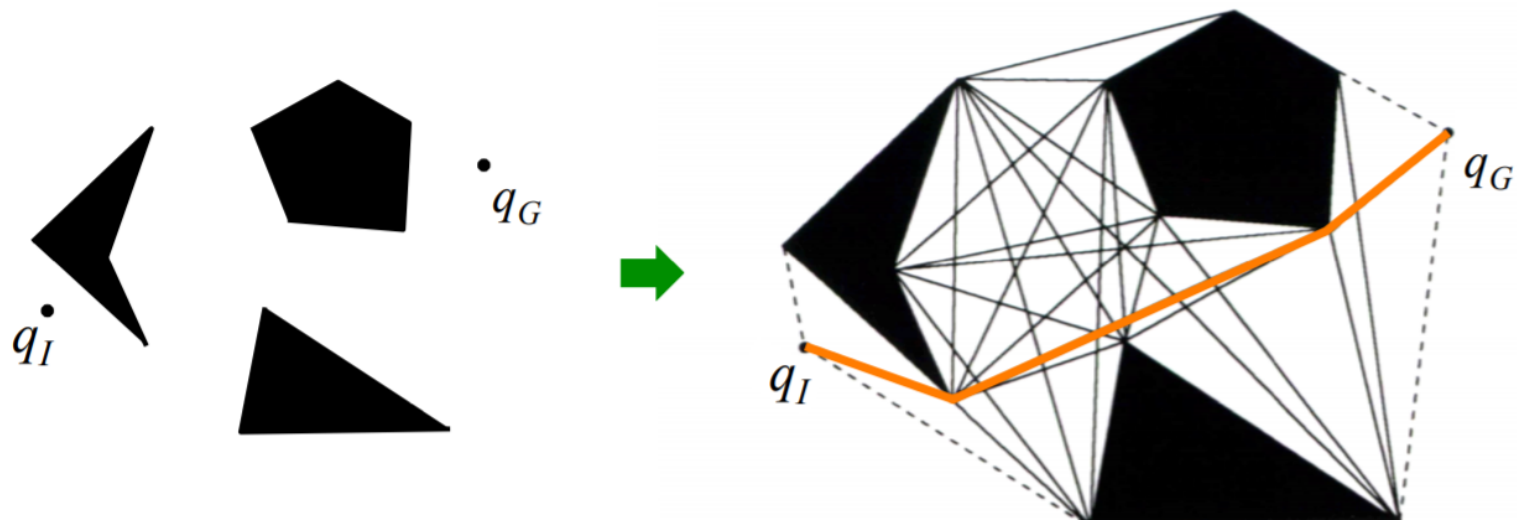
- Continuous terrain needs to be discretised for path planning.
- There are two general approaches to discretise configuration spaces:
  - **Combinatorial planning**: Characterizes C_{free} explicitly by capturing the connectivity of C_{free} into a graph and finds solutions using search.
  - **Sampling-based planning**: Uses collision-detection to probe and incrementally search the configuration space for solution.

Combinatorial planning

- The combinatorial planning techniques produce a road map.
  - 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}
- We will analyse the principal characteristics of two combinatorial planning techniques:
  - **Visibility graphs**
  - **Exact cell decomposition**

Visibility Graphs

- The general idea is to construct a path as a polygonal line connecting q_I and q_G through vertices of C_{obs}.
- Existence proof for such paths, optimality.
- One of the earliest path planning methods.

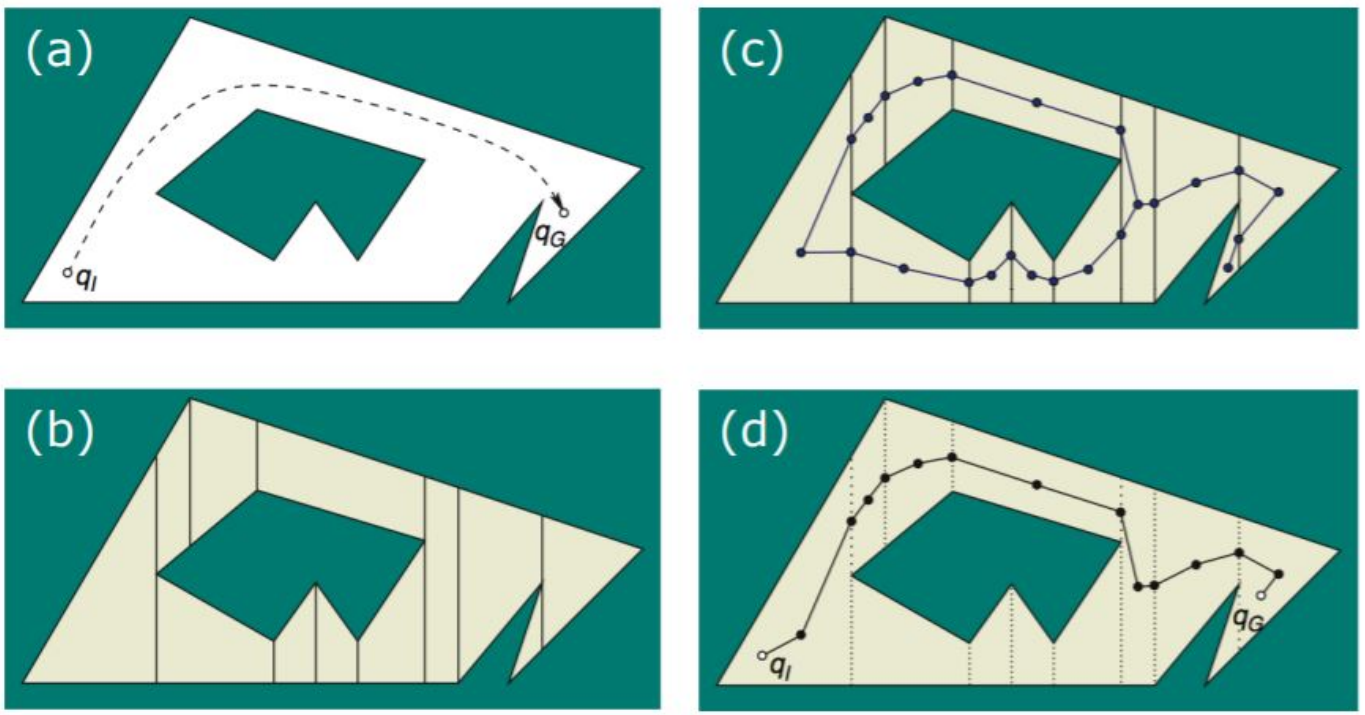


Exact Cell Decomposition

- The main idea of this method is the decomposition of C_{free} into non-overlapping cells, construct connectivity graph to represent adjacencies, then search.
- Steps for the idea implementation:
  - Decompose C_{free} into trapezoids with vertical side segments by shooting rays upward and downward from each polygon vertex.
  - Place one vertex in the interior of every trapezoid, pick e.g. the centroid.
  - Place one vertex in every vertical segment.
  - Connect the vertices.

Exact Cell Decomposition: Trapezoidal decomposition

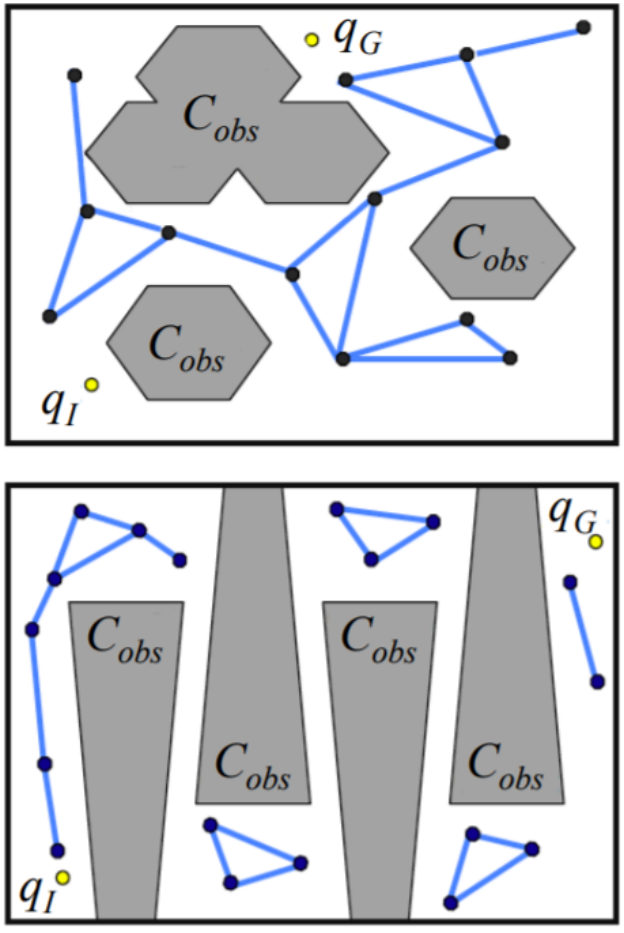
- Best known algorithm: $O(n \log n)$ where n is the number of vertices of C_{obs}



Sampling-Based Planning

- Abandon the concept of explicitly characterizing C_{free} and C_{obs} and leave the algorithm in the dark when exploring C_{free}.
- The only light is provided by a collision detection algorithm, that probes C to see whether some configuration lies in C_{free}.
- Two strategies will be presented:
  - Probabilistic road maps (PRM)
  - Rapidly exploring random trees (RRT)

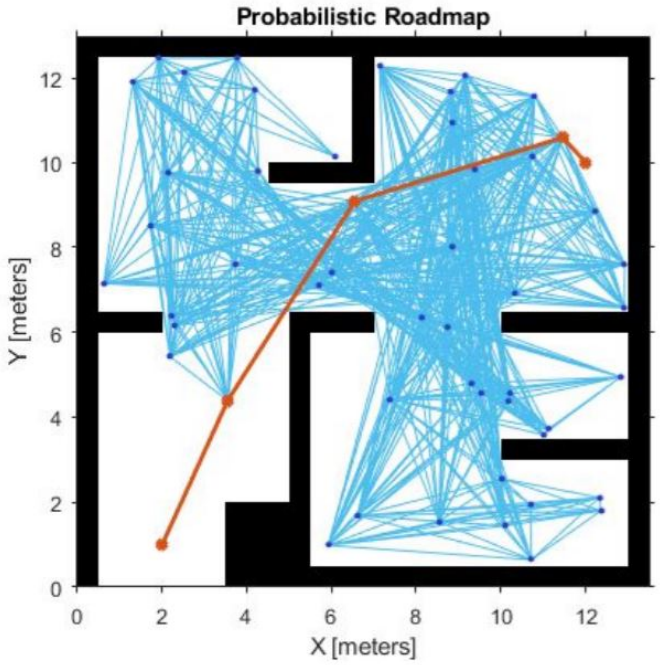
Probabilistic road maps (PRM)

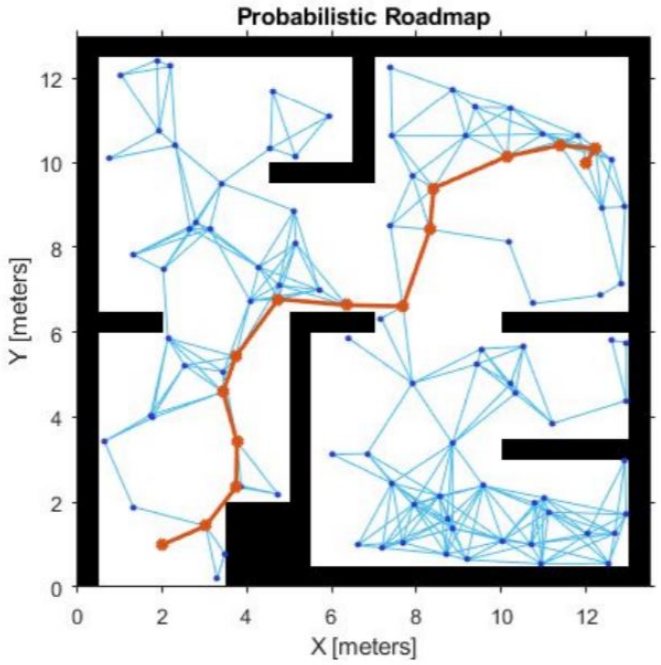


- Take random samples from C, declare them as vertices if in C_{free}, try to connect nearby vertices with local planner.
- The local planner checks if line-of-sight is collision-free (powerful or simple methods).
- Options for nearby: k-nearest neighbors or all neighbors within specified radius.
- Configurations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

PRM

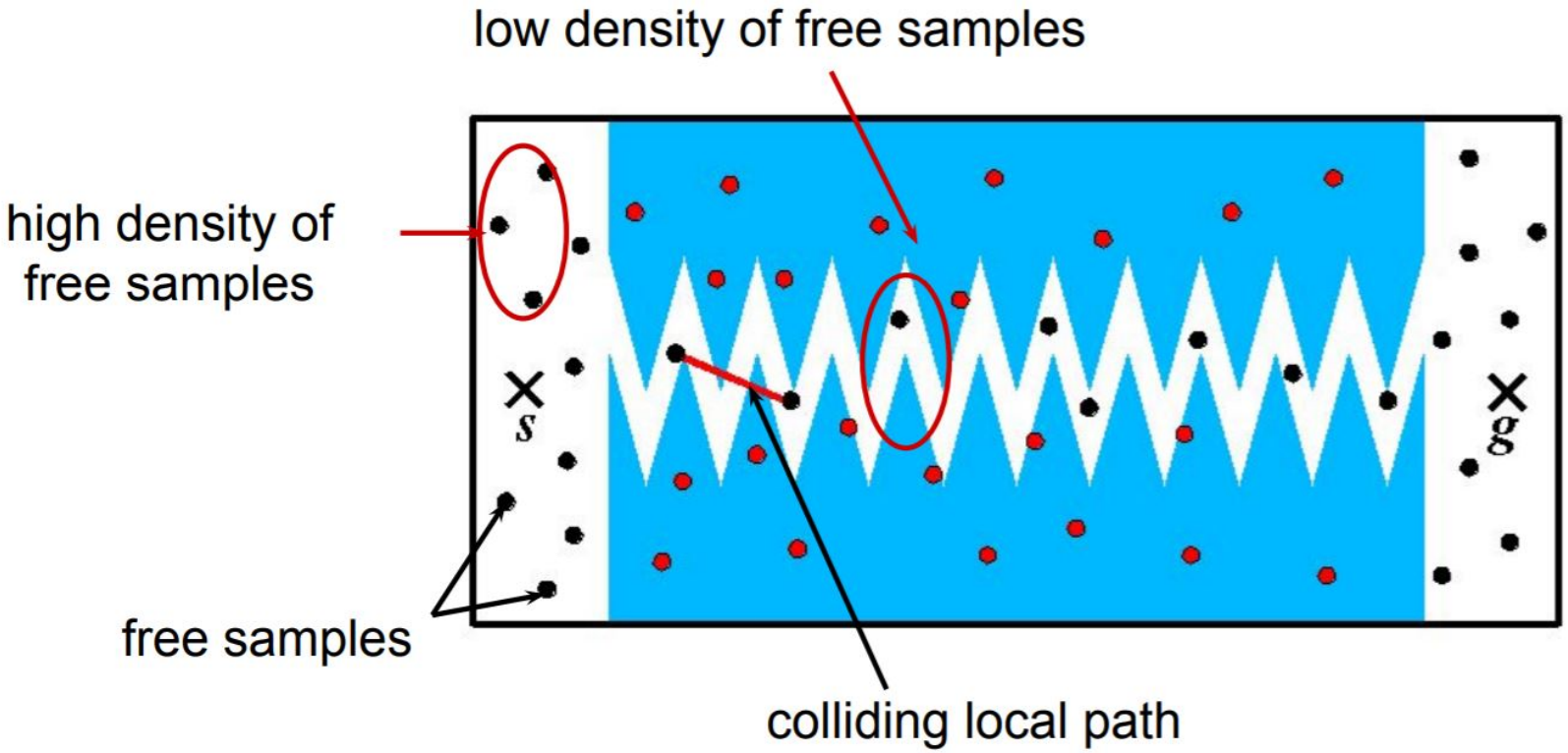
- The neighbourhood distance for connecting each collision-free node is a tunable parameter.
- A smaller connection distance reduces the number of connections, resulting in a simpler map.
- We need to control the dimensionality of configuration space to deal with the complexity of the PRM.





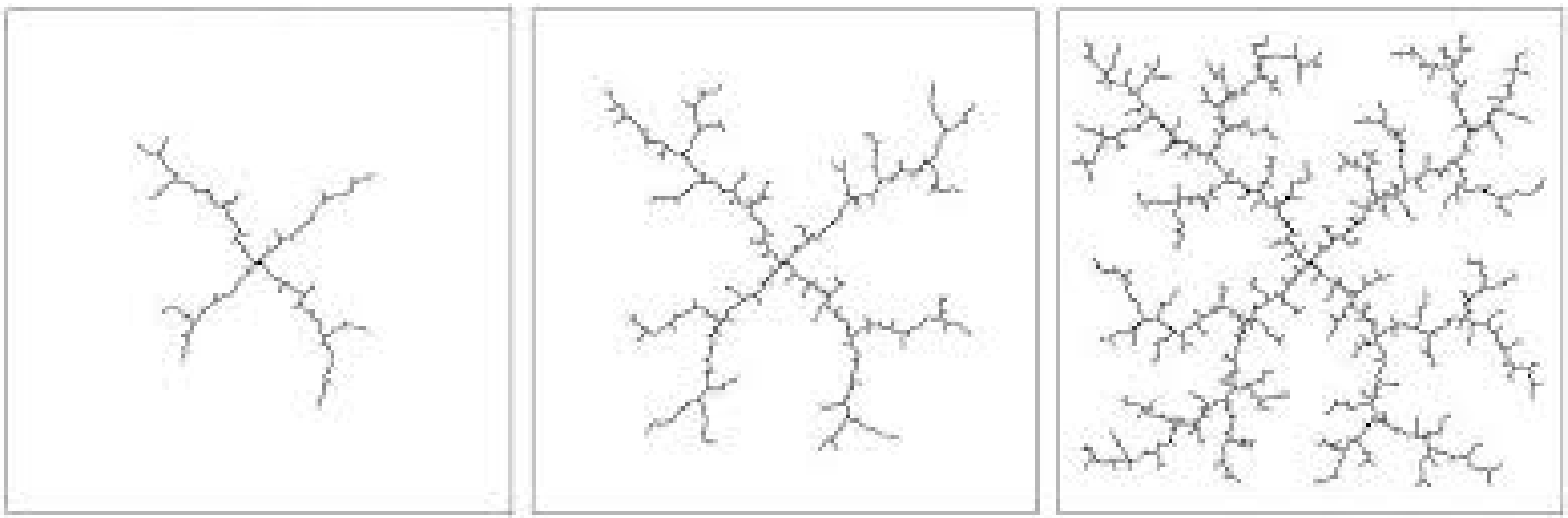
PRM in Narrow Spaces

- It is difficult to capture the free space associated with the narrow passages, because in the narrow passage, volume associated with the free space is relatively low.



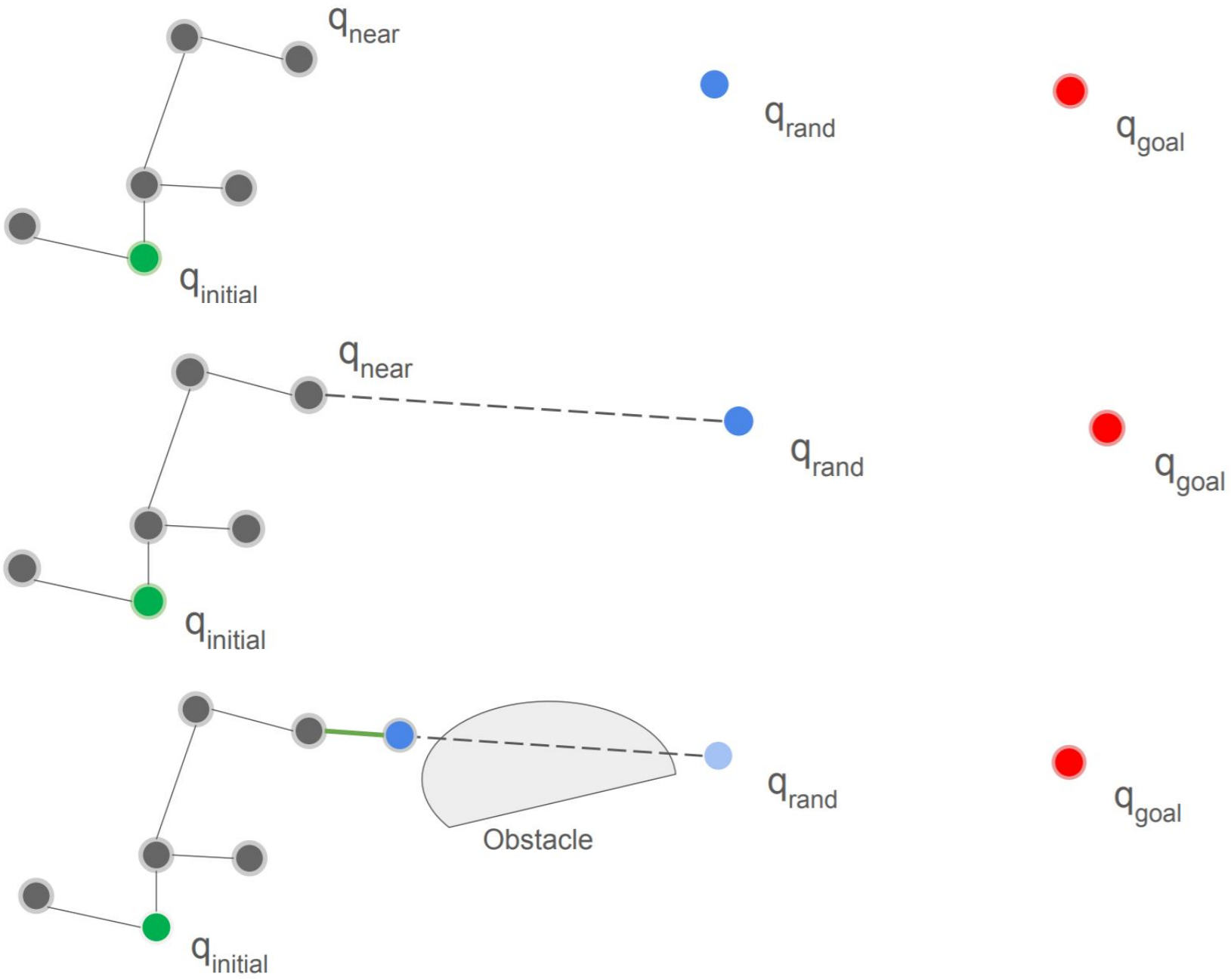
Rapidly Exploring Random Trees

- A single-query planner which makes incremental search; a single set of initial-goal is given to the planning algorithm.
- 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

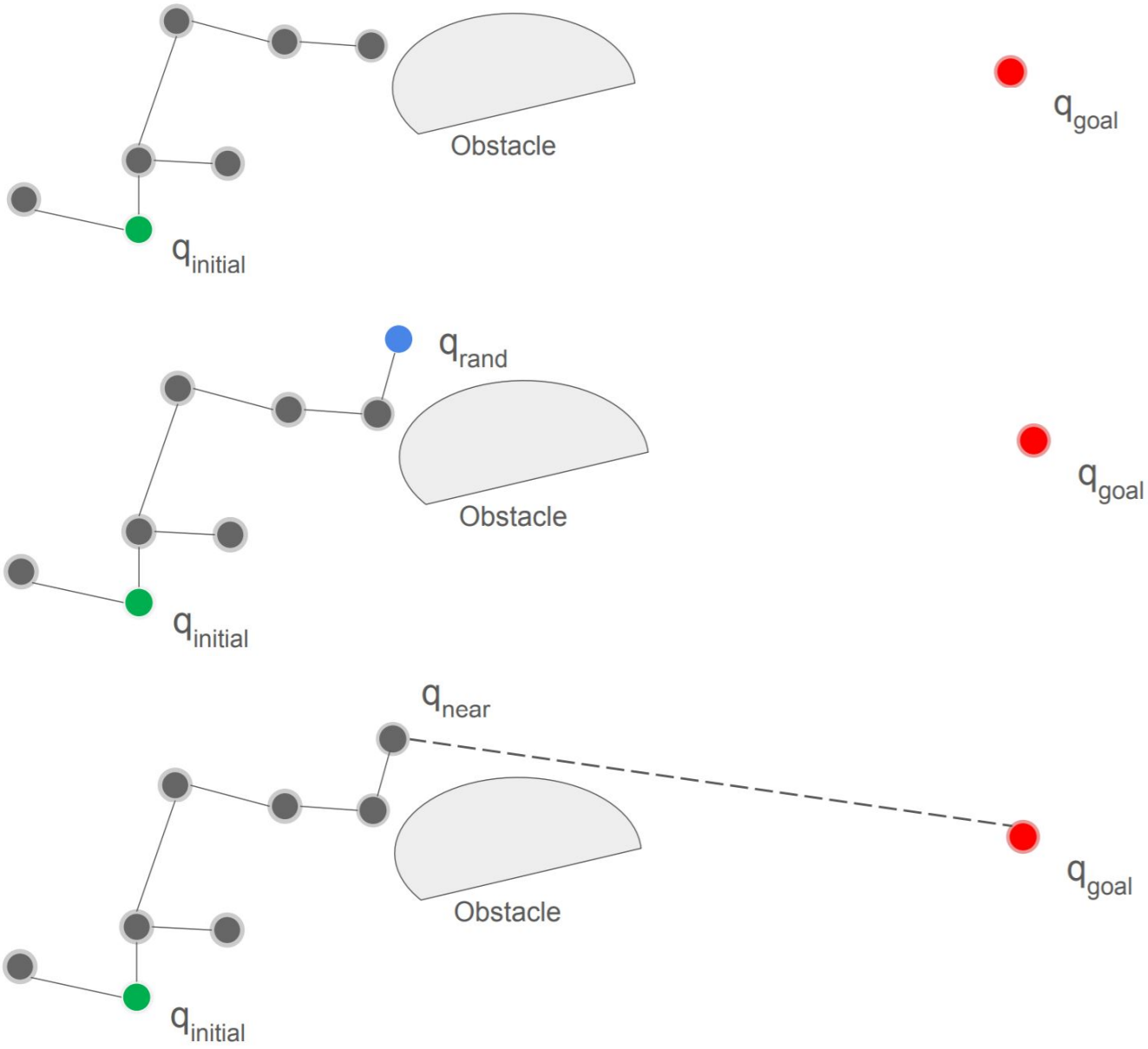


- Path planning with RRTs:
  - Start RRT at q_I.
  - At every, determined number of iterations force $q_{rand} = q_G$.
  - If q_G is reached, problem is solved.
- Choosing q_G every iteration will fail and waste much effort in running into C_{obs} instead of exploring the space.
- Some problems require more effective methods: bidirectional search

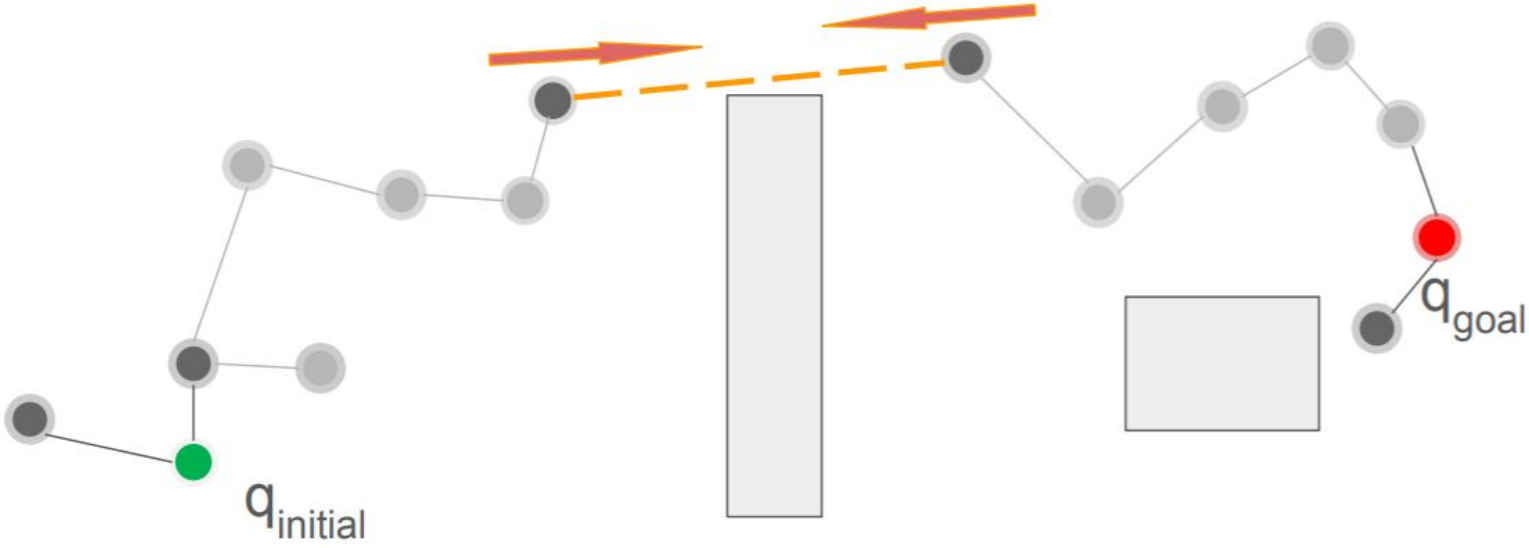
RRT Example


q_{near}
q_{rand}
q_{goal}
$q_{initial}$
q_{near}
q_{rand}
q_{goal}
$q_{initial}$
q_{rand}
Obstacle
q_{goal}
$q_{initial}$

RRT Example


Obstacle
q_{goal}
$q_{initial}$
q_{rand}
Obstacle
q_{goal}
$q_{initial}$
q_{near}
Obstacle
q_{goal}
$q_{initial}$

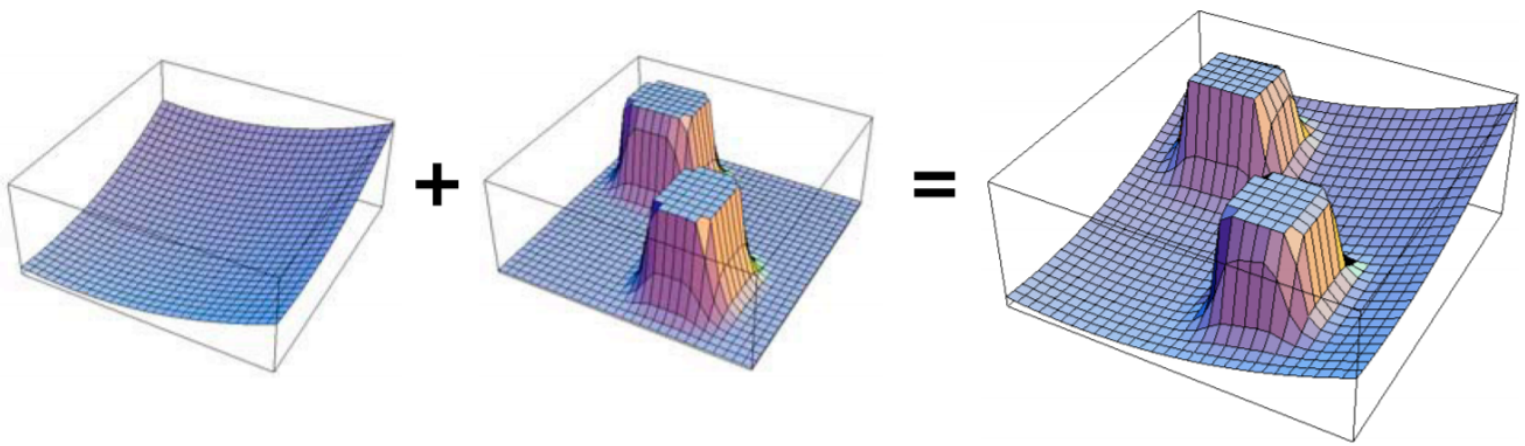
RRT Example: RRT-Connect



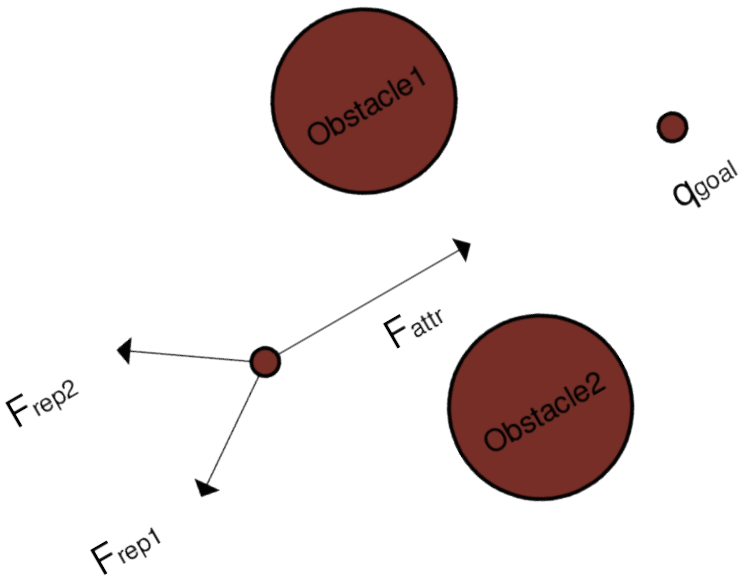
- This approach finds a graph that covers the space nicely that is independent of the query, ie RRT-Connect is still single query planning.
- The idea of constructing search trees from the initial and goal configurations comes from classical AI bi-directional search.

Potential Field

- All techniques discussed so far aim at capturing the connectivity of C_{free} into a graph.
- Potential Field methods follow a different idea: The robot, represented as a point in C, is modelled as a particle under the influence of a artificial potential field U.
- U superimposes:
  - Repulsive forces from obstacles
  - Attractive force from goal.

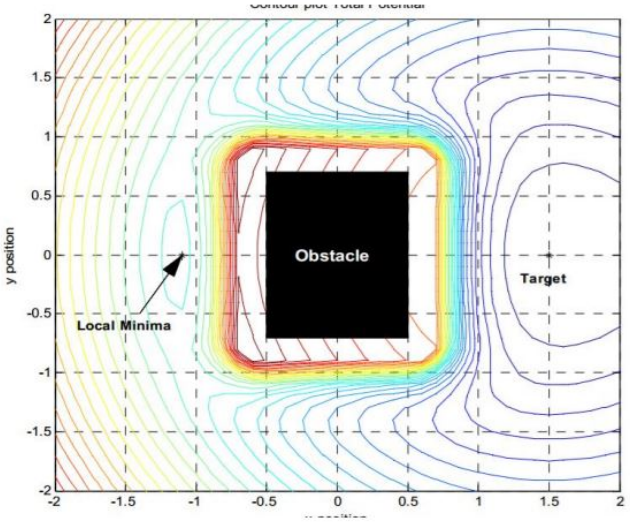


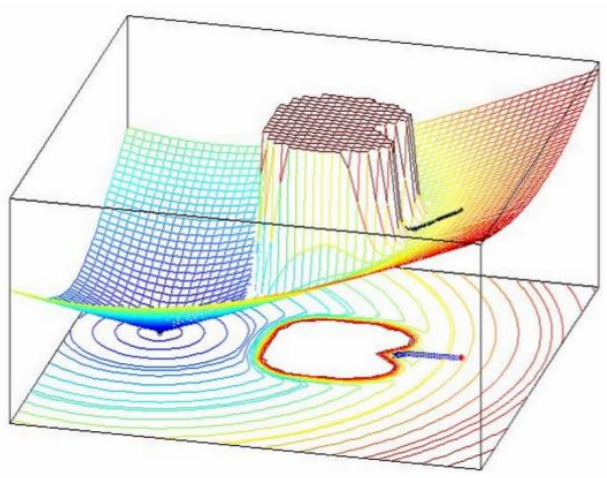
Potential Field

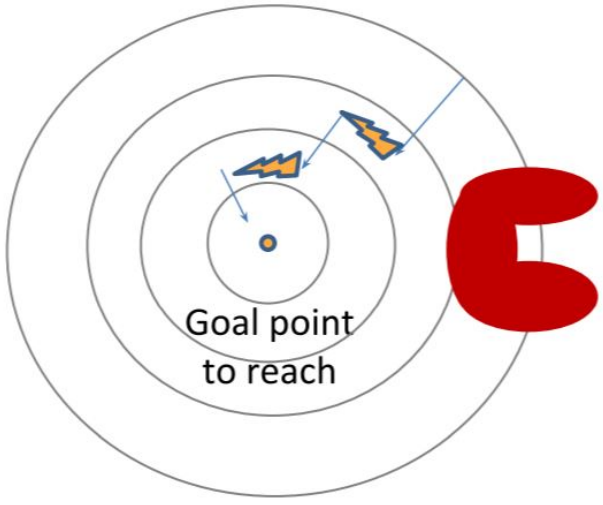


- Potential Field Equations:
  - $U(q) = U_{att}(q) + U_{rep}(q)$
  - $\overrightarrow{F}(q) = -\overrightarrow{\nabla} U(q)$
- Simply perform gradient descent.
- Main problems: robot gets stuck in local minima.
- The gradient of the potential function defines a vector field (similar to a policy) that can be used as feedback control strategy, relevant for an uncertain robot.
- However, potential fields need to represent C_{free} explicitly. This can be too costly.

Potential Field: Local Minima

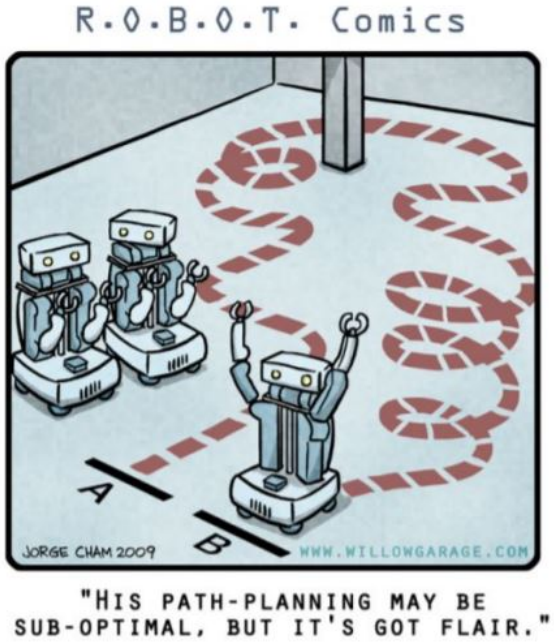


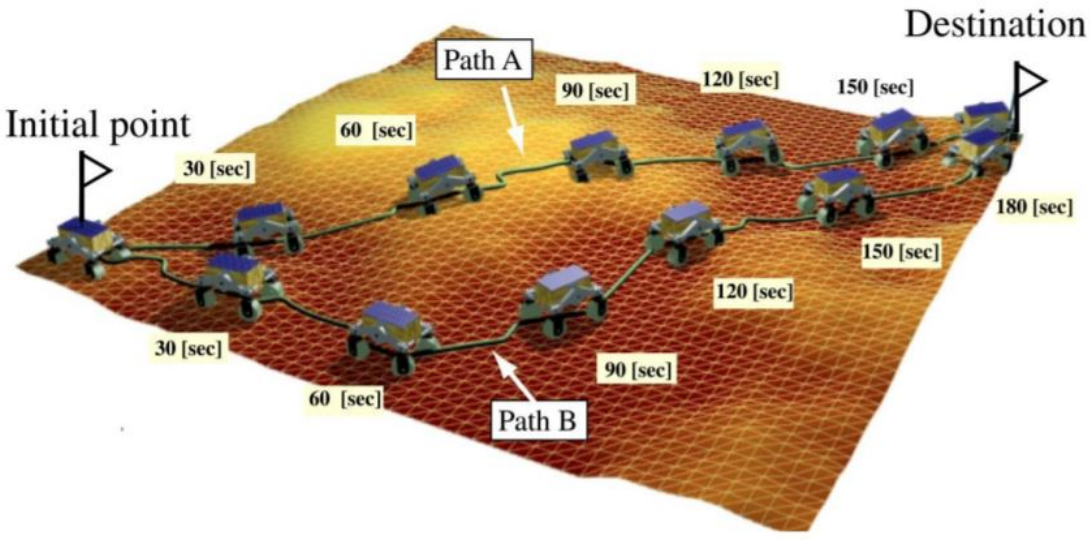




- There are solutions such as adding random walks to get out of local minima, but this problem is generally better resolved by sampling based methods.

Optimality of Motion Planning

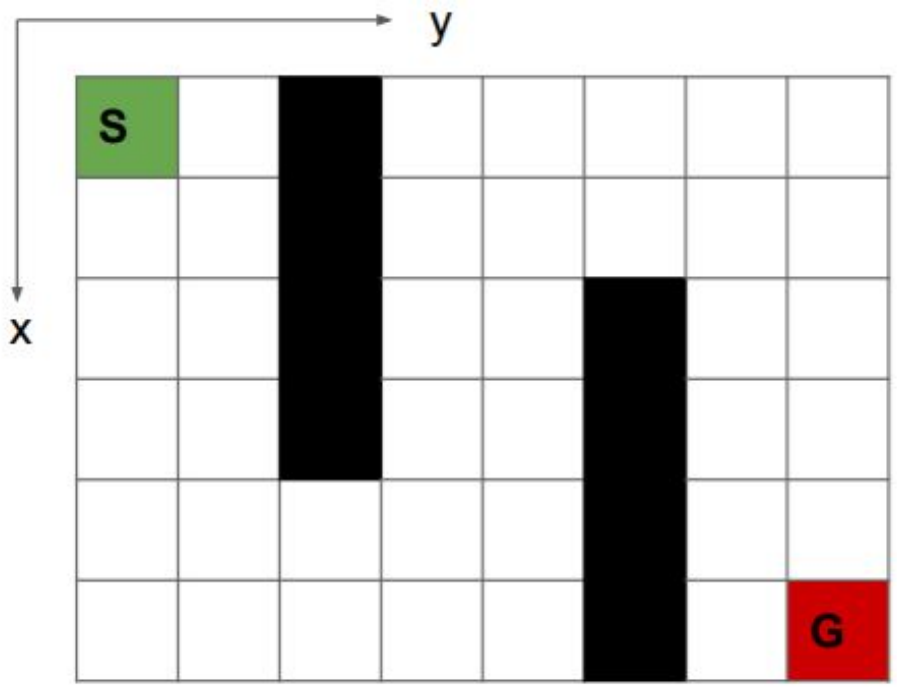




- Optimality: is the solution the best one in terms of path cost?
- Time complexity: how long does it take to find a solution?

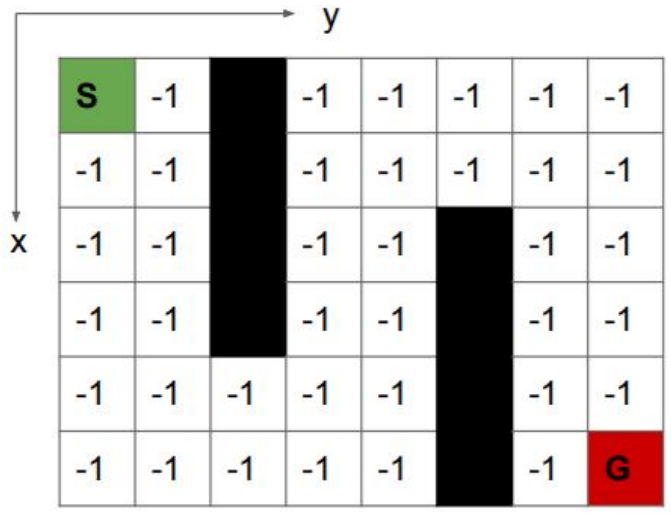
Dijkstra's Search Algorithm

- Dijkstra's Algorithm is a graph search algorithm that solves the shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree.
- Node: n
- G-value: g(n) is the cost of the path from the start node to n.



Dijkstra's Search Algorithm

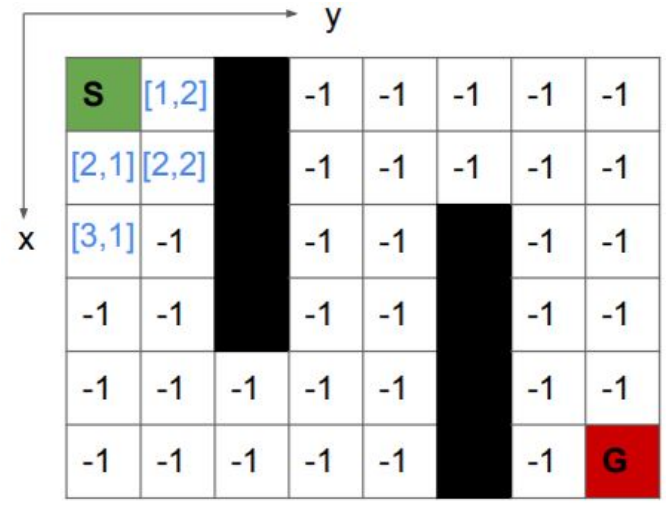
- (i) Set starting point as an initial node, and set other nodes unvisited. Create a set of matrix to store the check for visiting history, eg. unvisited '-1', visited '1'.



- (ii) From initial node, calculate the cost to reach all unvisited neighbour cells. Record the cost and the coordinate of each node. Store this [cost, node] set into a list.

- (iii) Compare all the sets of [cost, node] in the list and pick up a set [cost, node] with a smallest cost.

Dijkstra's Search Algorithm



Checked

{1, [2,1]} {1, [1,2]}

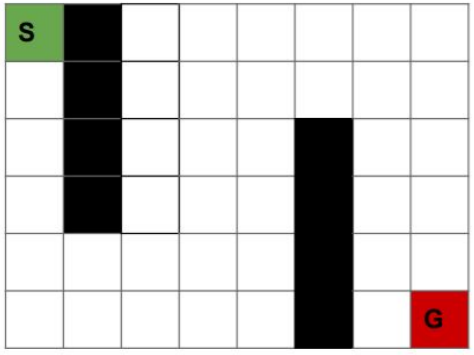
{2 ,[2,2]} {2 ,[3,1]}

Updated *list* :

{1, [2,1]} {1, [1,2]} {2, [2,2]} {2, [3,1]}

- (iv) From this selected set [cost, node], calculate the cost to reach all unvisited neighbour cells. Record the cost and the coordinate of each node. Store this [cost, node] set into a list. A bookkeeping is needed to register if a cell has been visited, as we don't visit a checked node again.
- (v) Repeat step (iii) and (iv) for the rest of unvisited cells, and update the list.
- (vi) If the solution exists, the algorithm will eventually find the goal, and the minimum cost is the optimal path. If the cost is the same for every unit of distance, then the min-cost path is the shortest.

Dijkstra's Search Algorithm: Example




```
grid = [[0, 1, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 1, 0, 0],
        [0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0]]
```

```
Cost map= [0, 0,10,11, 12, 13, 14, 15]
          [1, 0, 9, 10, 11, 12, 13, 14]
          [2, 0, 8,  9, 10,  0, 14, 15]
          [3, 0, 7,  8,  9,  0, 15, 16]
          [4, 5, 6,  7,  8,  0, 16, 17]
          [5, 6, 7,  8,  9,  0, 17, 18]
```

Number of steps: 18
Found goal location: (x, y) = (6, 8)

A* Search Algorithm

- It is an extension of Edsger Dijkstra's 1959 algorithm. A* uses an additional heuristics to guide its search for a better performance, which is more efficient because we don't need to visit every node.
- g(n): the cost of the path from the start node 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.

Dijkstra's algorithm

Iteration list:

[0, 1, -1, 25, 28, 31, 35, 40, 45, 50, 54]
[2, 3, -1, 23, 26, 29, 32, 36, 41, 46, 51]
[4, 5, -1, 20, -1, -1, 37, 42, -1, 52, 55]
[6, 7, -1, 17, 21, 24, -1, 47, 53, 56, 57]
[8, 9, 11, 14, -1, 27, -1, 43, 48, -1, 58]
[10,12,15, 18, -1, 30, 33, 38, -1, 60, 59]
[13,16,19, 22, -1, 34, 39, 44, 49, -1, **61**]

A*

Iteration list:

[0, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
[2, 3, -1, 34, 35, 36, 37,38, 40, 42, 44]
[4, 5, -1, 31, -1, -1, 39, 41, -1, 45, 47]
[6, 7, -1, 20, 21, 22, -1, 43, 46, 48, 49]
[8, 9, 11, 14, -1, 23, -1, 32, 33, -1, 50]
[10,12,15, 17, -1, 24, 25, 27, -1, -1, 51]
[13,16,18, 19, -1, 26, 28, 29, 30, -1, **52**]

Dynamic Window Approach

- It is a technique called to solve collision avoidance problems (obstacle avoidance)
- In collision avoidance problems are common to subdivide the problem into a global and local planning task:
  - An approximate global planner computes paths ignoring the kinematic and dynamic vehicle constraints.
  - An accurate local planner accounts for the constraints and generates (sets of) feasible local trajectories ("collision avoidance")
- For Dynamic Window Approach the path to goal (a set of via points), range scan of the local vicinity, dynamic constraints are given.
- We look for: collision-free, safe, and fast motion towards the goal.

Task Planners

Components of planning task

- **Objects:** Things in the world that interest us.
- **Predicates:** Properties of objects that we are interested in; can be true or false.
- **Functions:** Fluents that return a number.
- **Initial state:** The state of the world that we start in.
- **Goal specification:** Things that we want to be true.
- **Actions/Operators:** Ways of changing the state of the world.

Language:

- PDDL = Planning Domain Definition Language

Planning is only applicable when the system can be modelled as state transitions. The state transition system is defined as

$$\sum = (S, A, E, \gamma) \,, \tag{1}$$

where:

- $S = s_1, s_2 \ldots s_n$ is the set of possible states,
- $A = a_1, a_2 \ldots a_n$ is the set of possible actions,
- $E = e_1, e_2 \ldots e_n$ is the set of exogenous events that are out of the planner's control and can affect the current state, and
- $\gamma : SxAxE$ is the state transition function where the execution of an action can make a transition from s to a next state s'.

Classical & Temporal Planning

Classical Planning: Planning for the restricted model.

Restrictions:

- Finite state space.
- Fully observable: agent has full knowledge of the world.
- Deterministic: taking an action in a given state will cause a transition to only one possible state.
- Static world: the world stays in the same state until the agent takes an action.
- Restricted goals: goals must be defined as desired states to achieve and does not include states to avoid.
- Sequential plans: a solution is a linearly ordered finite sequence of actions.
- Implicit time: actions do not have durations and states transit instantaneously.
- Offline planning: planner assumes initial and goal states do not change while it is searching for a solution.

Temporal Planning: Extends classical planning by removing the assumption of implicit time.

Advantages:

- Allows more realistic modelling of the real-world.
- Extends the applications of planning to domains where temporal constraints must be satisfied.

Planners:

- Forward-Chaining Partial-Order Planning (**POPF**).
- Optimizing Preferences and TIme-dependent Costs (**OPTIC**).
- Temporal Fast Downward (**TFD**)

PDDL

Planning tasks specified in PDDL are separated into two files:

- **Domain file:** for types, predicates, functions and actions.
- **Problem file:** for objects, initial state and goal specification.

```
(define (domain <domain name>)
    (:requirements :typing :action-cost
    (:types <list of types>)
    <PDDL code for predicates>
    <PDDL code for functions>
    <PDDL code for first action>
    [...]
    <PDDL code for last action>
)
```

```
(define (problem <problem name>)
    (:domain <domain name>)
    <PDDL code for objects>
    <PDDL code for initial state>
    <PDDL code for goal specification>
    <PDDL code for metric>
)
```

Implementation Example

Video From: PANDORA European Project

THANK YOU!!!