IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

PROGRAMING OF ROBOTIC SYSTEMS

BOT

Panagiotis PAPADAKIS

CONTENTS

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

CS-4
GRAPH SEARCH (CONTINUED)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

DEPTH-FIRST SEARCH (DFS) – ITERATIVE DEEPENING

Algorithm 1: DFS_planner

```
Input: Initial state x_s, goal state x_g, depth dep
Output: Success, Terminationflag
O.insert_last(x_s)
x.parent=NULL,i=0
while (O ≠ ∅) do
    x=O.remove_last()
    C.insert(x) /* Closed set                    */
    if (x.depth <= dep) then
        for u ∈ U(x) do
            x' = f(x, u)
            x'.depth = x.depth + 1
            x'.parent=x
            if (x' == x_g) then
                return Success
            end
            else if x' ∉ C then
                if x' ∉ O then
                    O.insert_last(x')
                end
            end
        end
    end
end
return (O == ∅) /* Termination flag          */
```

Algorithm 1: DFS-ID planner

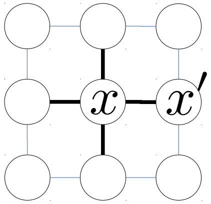
```
Input: Initial state x_s, goal state x_g
Output: Success or Failure
depth = 1
while True do
    res=DFS_planner(x_s,x_g,dep)
    if (res==Success) then
        return Success
    end if
    else if (res==False) then
        /* Max depth reached                  */
        dep = dep + 1
    end if
    else if (res==True) then
        /* All states explored                */
        break
    end if
end while
return Failure
```

Characteristics

- **Good trade-off** between **processing** time and **memory** usage
- Appropriate when **search tree** has **large branching** factor (many actions per state)
- If the search space **has loops** then the depth of a node within a loop depends on the path taken. In this case the original **DFS-ID** is **inappropriate**.

GRAPH SEARCH
CHANGING TRANSITION COSTS

So far, **transitions** to new states where **equally treated**, i.e. there was **no reason** or preference to **select** a particular $u \in U(x)$ in order to arrive to a new state $x' = f(x, u) \in X$:



This is done in the **absence of any information** that could help to **arrive** to the **goal** and hence graph **search** is performed **blindly**.

In the next category of graph search algorithms, the planner can exploit information to help it determine what is the most promising new state to go towards the goal.

DIJKSTRA'S ALGORITHM

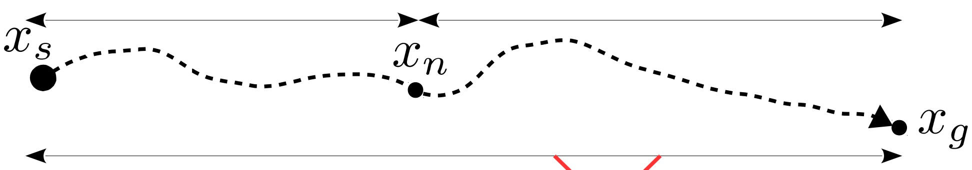
Characteristics

Cost to come

$$\boxed{g(n) := g(x_s, x_n)}$$

Cost to go

$$h(n) := h(x_n, x_g)$$



$$\boxed{f(n) = g(n)} + \cancel{h(n)}$$

Total cost

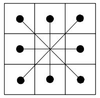
not used

- Dijkstra’s algorithm takes into **account only** the cost to come g(n) (hence h(n) = 0)
- If all edges **costs** are **equal**, **Dijkstra’s** algorithm is **equivalent** to **BFS** algorithm
- Due to the use of the heap (priority queue), **time complexity** of Dijkstra’s algorithm is **higher**

Algorithm 1: Dijsktra’s algorithm

```
Input: Initial state x_s, goal state x_g
Output: Success or Failure
x_s.g = 0
O.insert_sorted(x_s, x_s.g) /* Heap of open states */
x.parent=NULL
while O ≠ ∅ do
    x=O.remove_first()/* Pop element with min cost */
    C.insert(x) /* Set of closed states */
    for u ∈ U(x) do
        x' = f(x, u)
        x'.parent=x
        if (x' == x_g) then
            return Success
        end
        else if x' ∉ C then
            gnew = x.g + edge_cost(x, x')
            if x' ∉ O then
                O.insert_sorted(x', gnew)
            end
            else
                if (gnew < x'.g) then
                    O.remove(x')
                    x'.g = gnew
                    O.insert_sorted(x', gnew)
                end
            end
        end
    end
end
return Failure
```

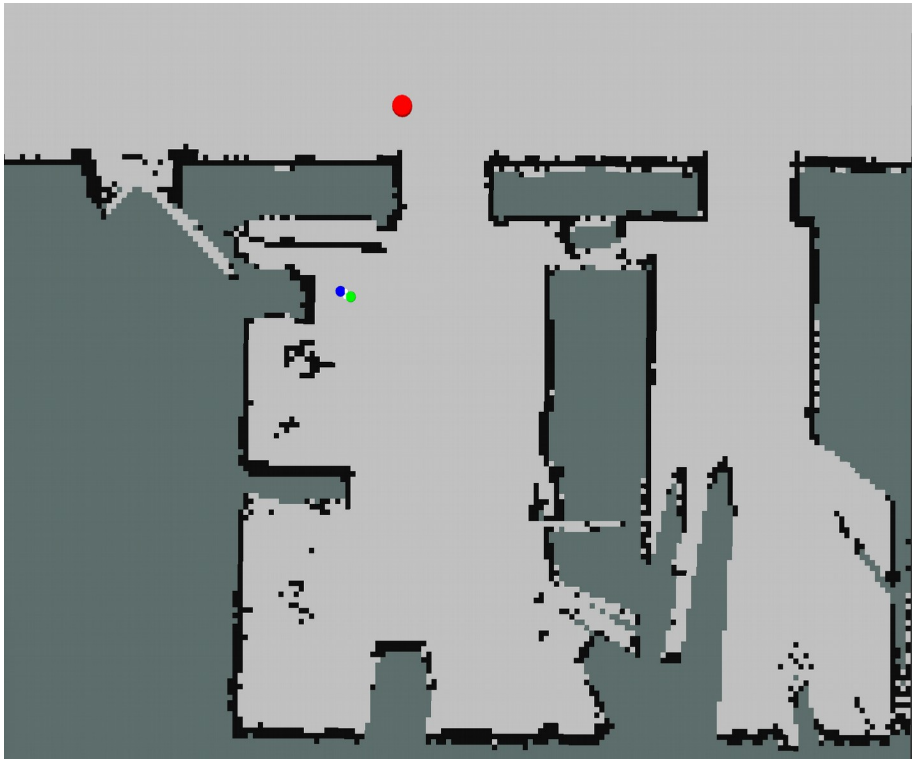
GRAPH SEARCH
DIJKSTRA'S ALGORITHM



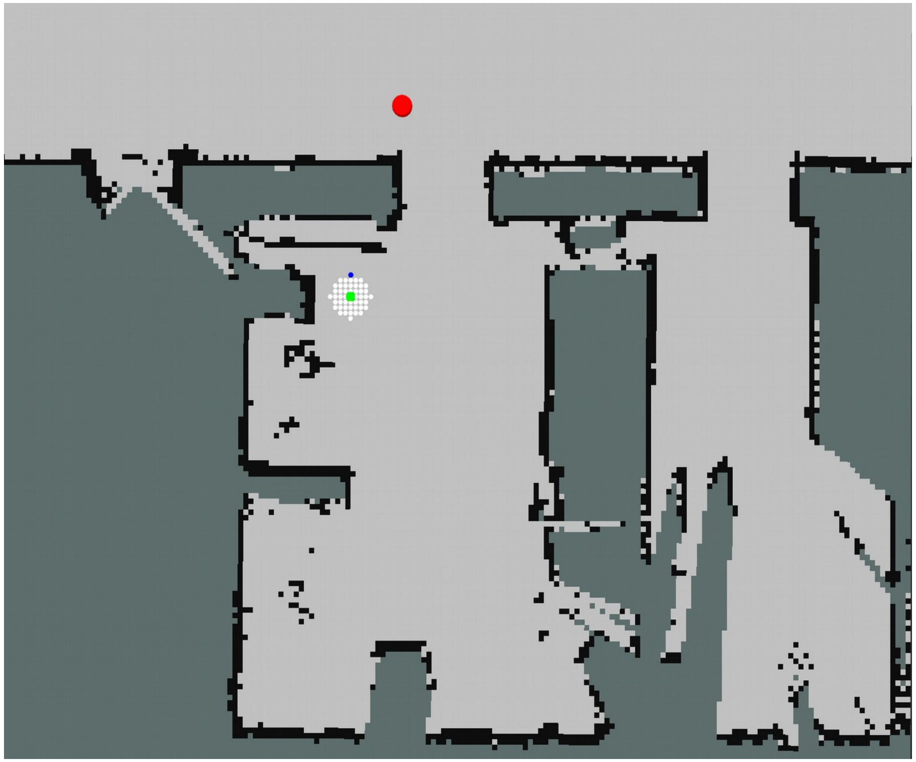
Example

8 possible state transitions, with Euclidean distance as edge cost

#5 iterations



#50 iterations

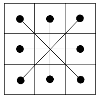


Red point: Goal state

Green point: Initial state

Blue point: Current state

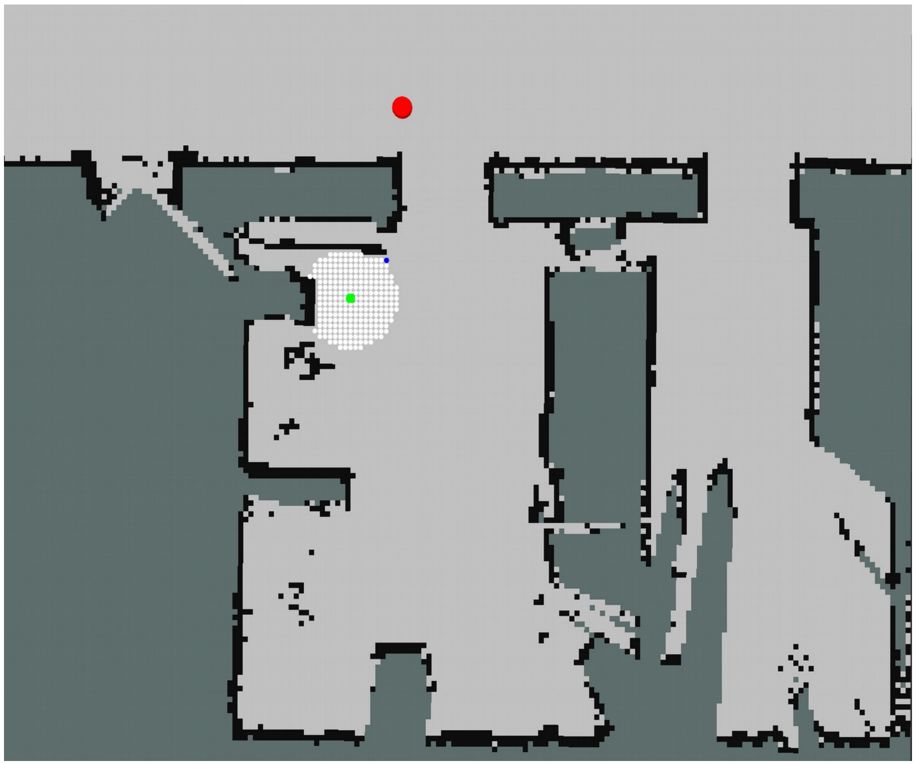
GRAPH SEARCH
DIJKSTRA'S ALGORITHM



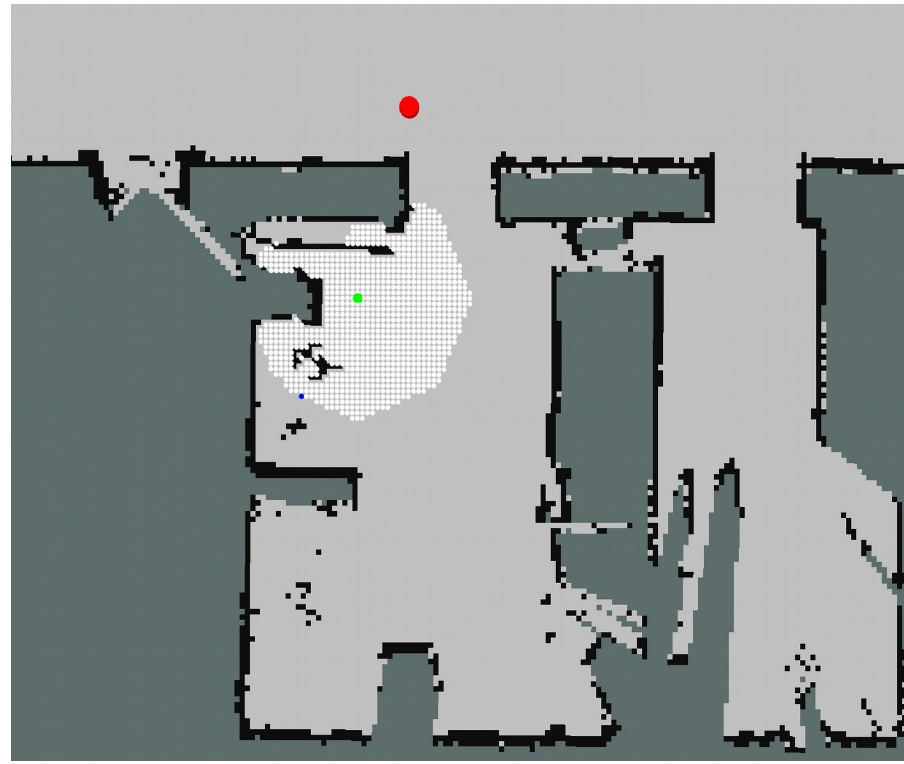
Example

8 possible state transitions, with Euclidean distance as edge cost

#250 iterations



#1000 iterations



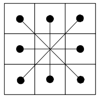
Red point: Goal state

Green point: Initial state

Blue point: Current state

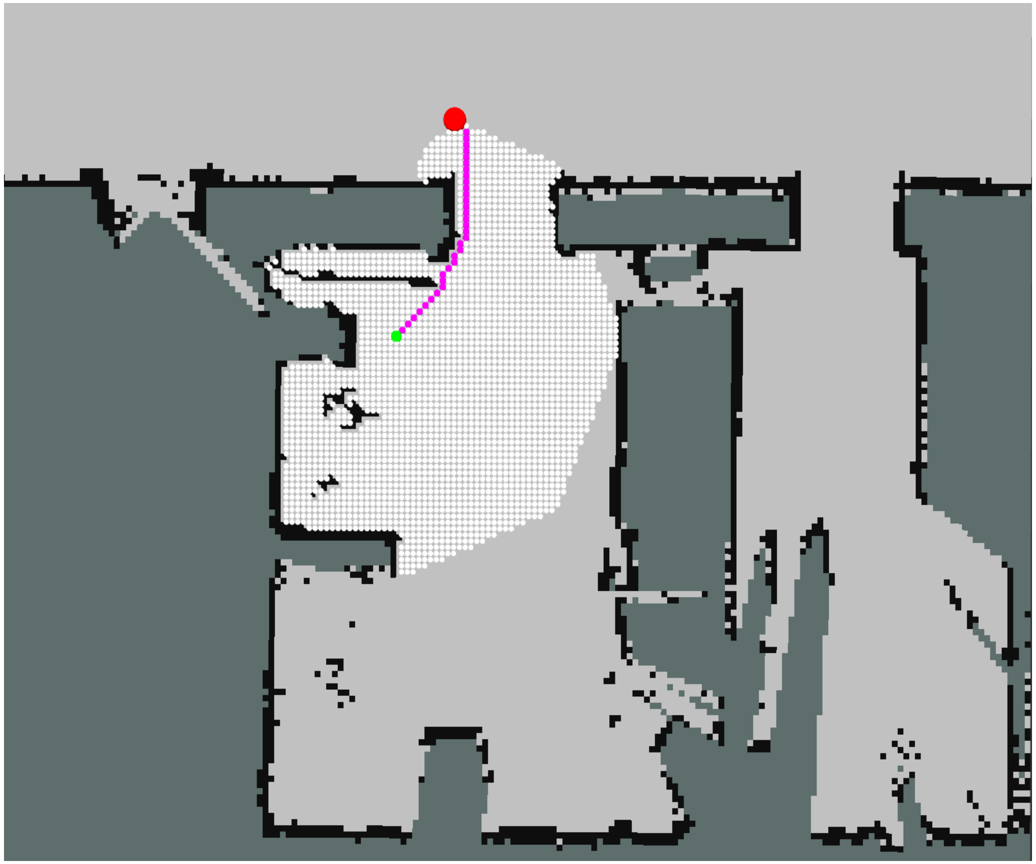
GRAPH SEARCH

DIJKSTRA'S ALGORITHM

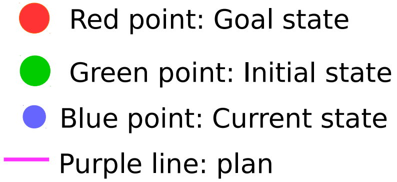


Example

8 possible state transitions, with Euclidean distance as edge cost



#2616 iterations



GRAPH SEARCH
A* ALGORITHM

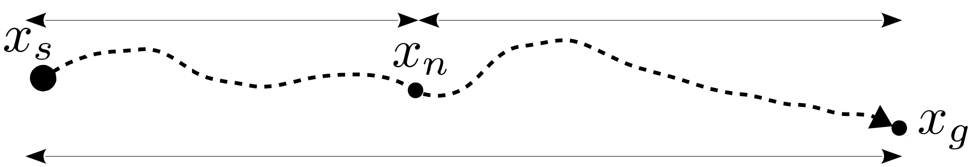
Path planning algorithm developed in the late 70s for « **Shakey the robot** », at Stanford

Cost to come

$$g(n) := g(x_s, x_n)$$

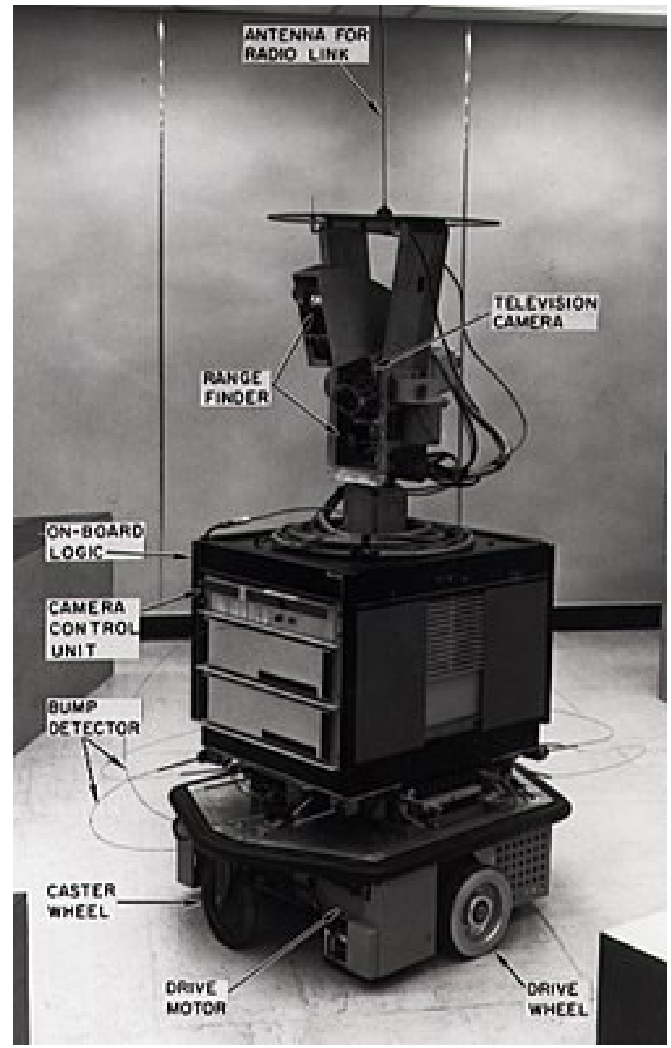
Cost to go

$$h(n) := h(x_n, x_g)$$



Total cost $\boxed{f(n) = g(n) + h(n)}$

- Cost to come is the same as Dijkstra's Algorithm
- Cost to go is a heuristic estimate of the remaining cost to the goal. This estimate should be **admissible for all x** so that A* is **optimal**
- The total cost $f(n)$ is an estimate of the total cost



Shakey the robot
source (wikipedia)

GRAPH SEARCH
A* ALGORITHM

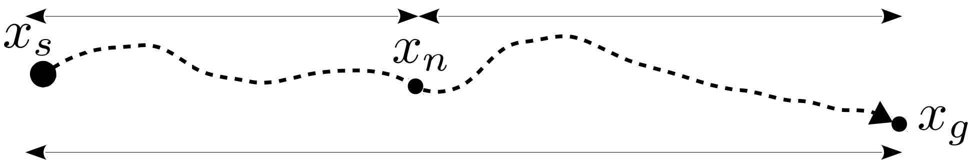
Path planning algorithm developed in the late 70s for « **Shakey the robot** », at Stanford

Cost to come

$$g(n) := g(x_s, x_n)$$

Cost to go

$$h(n) := h(x_n, x_g)$$



Total cost $$f(n) = g(n) + h(n)$$

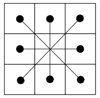
- Cost to come is the same as Dijkstra's Algorithm
- Cost to go is a heuristic estimate of the remaining cost to the goal. This estimate should be **admissible for all x** so that A* is **optimal**
- The total cost $f(n)$ is an estimate of the total cost

- Determining an admissible cost to go function depends on the structure of the state space:
  - *4-connected* neighborhood (**L1** distance)
  - *Full-connected* neighborhood (**L2** distance)

Algorithm 1: A* algorithm

Input: Initial state x_s, goal state x_g
Output: Success or Failure
$x_s.g = 0$
$x_s.f = x_s.g + h(x_s)$
O.insert_sorted(x_s, $x_s.f$) `/* Heap of open states */`
x.parent=NULL
while $O \neq \emptyset$ **do**
  x=O.remove_first() `/* Pop element with min cost */`
  C.insert(x) `/* Set of closed states */`
  for $u \in U(x)$ **do**
    $x' = f(x, u)$
    x'.parent=x
    if $(x' == x_g)$ **then**
      return Success
    end
    else if $x' \notin C$ **then**
      $gnew = x.g + edge\_cost(x, x')$
      $fnew = gnew + h(x')$
      $x'.f = fnew$
      if $x' \notin O$ **then**
        O.insert_sorted(x', $fnew$)
      end
      else
        if $(gnew < x'.g)$ **then**
          O.remove(x')
          $x'.g = gnew$
          O.insert_sorted(x', $fnew$)
        end
      end
    end
  end
end
return Failure

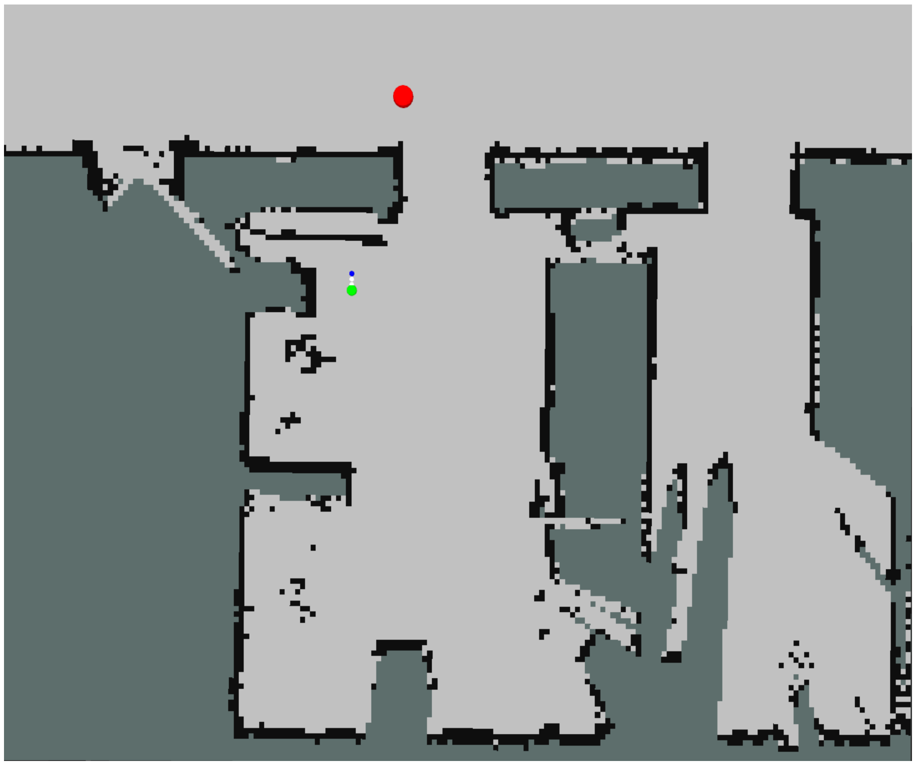
GRAPH SEARCH
A* ALGORITHM



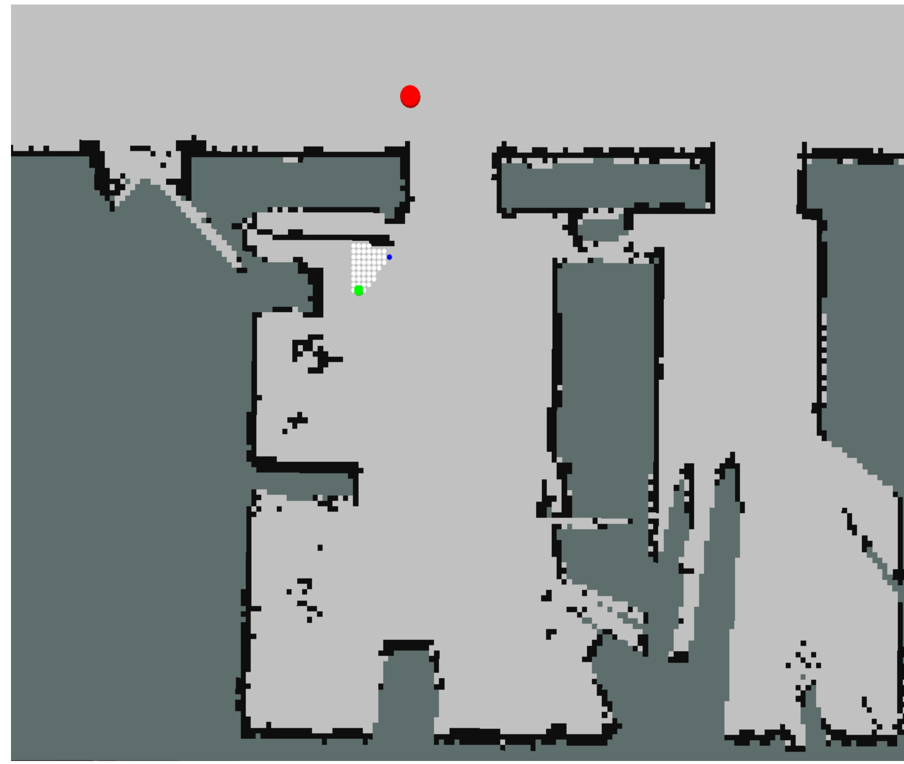
Example

8 possible state transitions, with Euclidean distance as edge cost

#5 iterations



#50 iterations

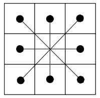


Red point: Goal state

Green point: Initial state

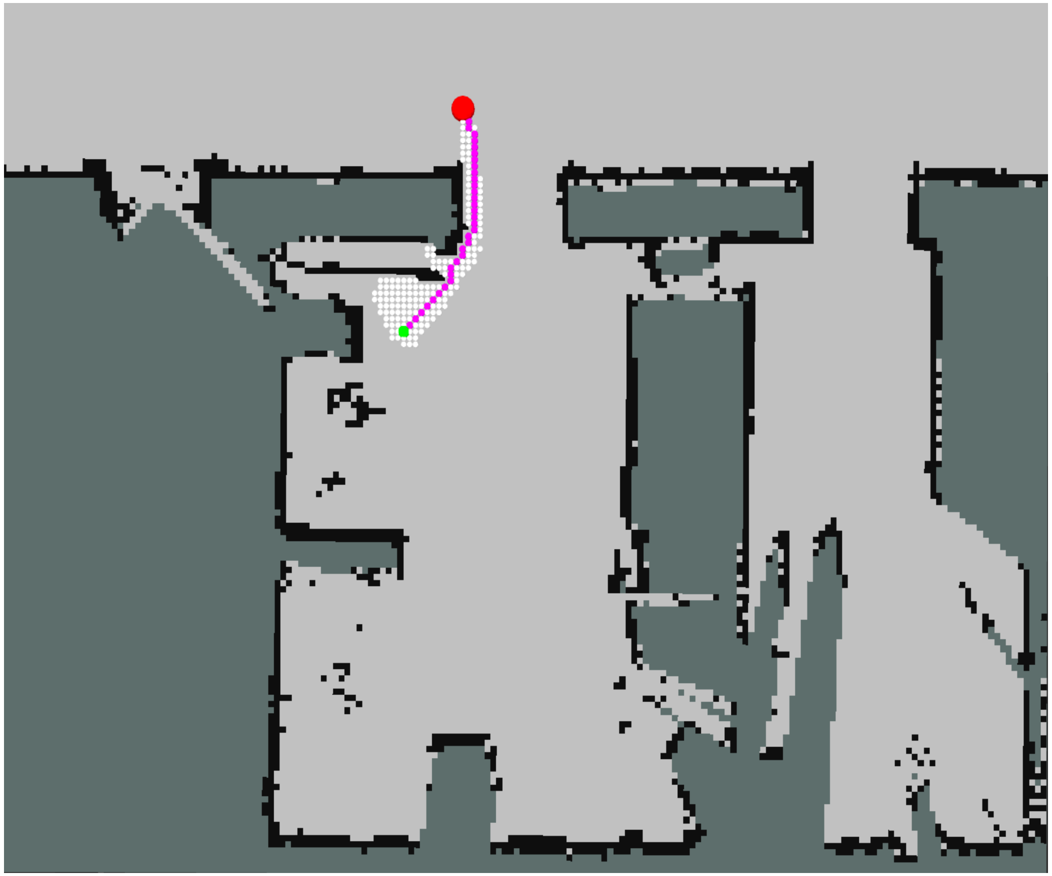
Blue point: Current state

GRAPH SEARCH
A* ALGORITHM

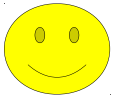


Example

8 possible state transitions, with Euclidean distance as edge cost



#190 iterations



Much more efficient than all the previous !

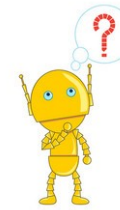
Red point: Goal state

Green point: Initial state

Blue point: Current state

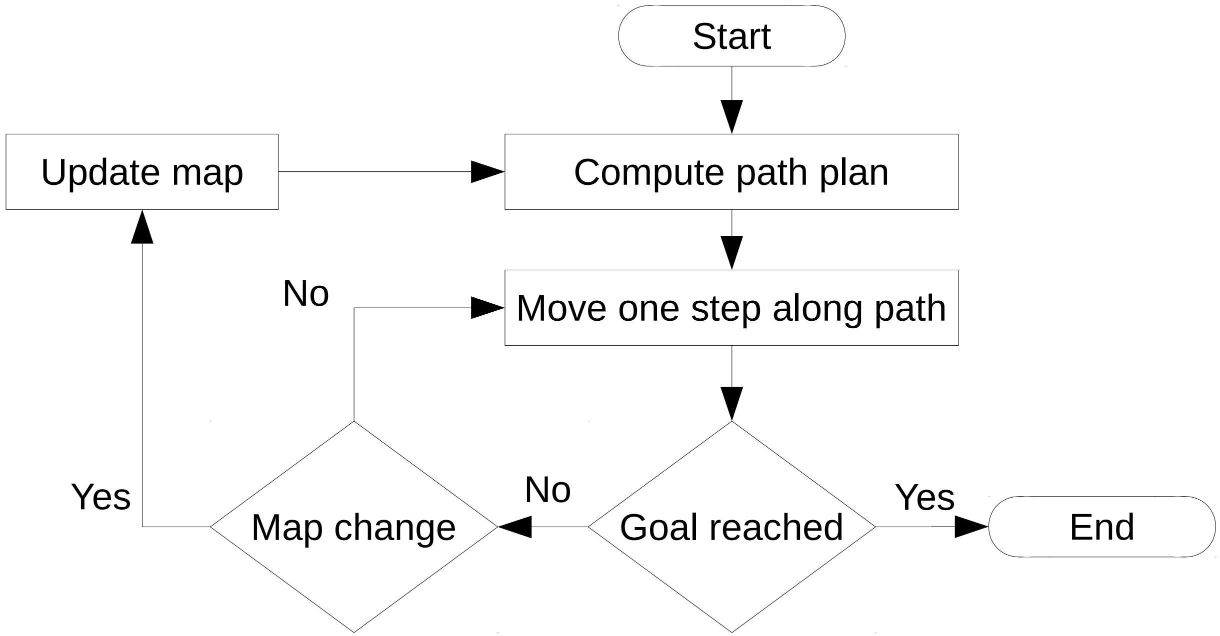
Purple line: plan

GRAPH SEARCH
REPLANNING

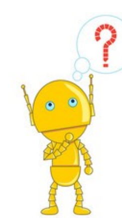


What to do if the **map changes while** the robot **executes** the computed path?

Reapply the **planning** algorithm in the new 2D map



- Not efficient if the robot starts with little information about its environment
- Not efficient when the start and goal state are far away from each other



How can we do better in partially known/dynamic environments?

PLANNING IN DYNAMIC ENVIRONMENTS: D*

The **D*** (a.k.a. Dynamic A*) algorithm[†] aims at **minimizing the cost of replanning** when the map changes.

- It achieves this by **reusing shortest paths** that are **not affected** by the **map change** and **updating** only the **paths** around the **affected** area.

- **Planning** starts in the opposite direction **from** the **GOAL** node **towards** the **START** node.

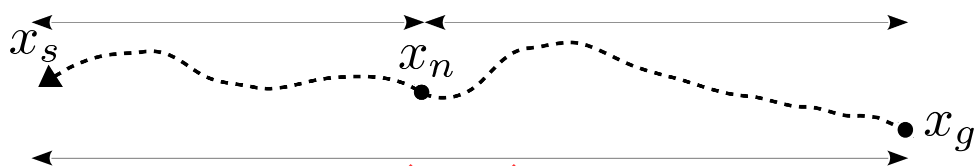
- In terms of results, it gives the **same solutions as Dijkstra’s algorithm** but with higher time efficiency.

Cost to **go**

$$g(n) := g(x_s, x_n)$$

Cost to **come**

$$h(n) := h(x_n, x_g)$$



$$f(n) = \cancel{g(n)} + h(n)$$

not used (hence g(n) = 0)

- The cost to come is no longer a heuristic but <u>the real cost from the goal to the current node</u>

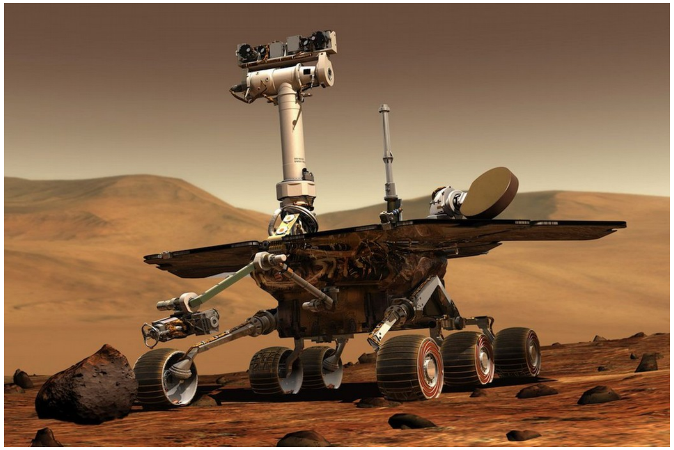
[†]Algorithm details in article : *Choset et al., Principles of Robot Motion Theory, Algorithms, and Implementations, MIT Press*

PLANNING IN DYNAMIC ENVIRONMENTS: D*

Examples of robot vehicles where D* was first used:

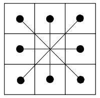
Automated Cross-Country Unmanned Vehicle (XUV)

Crusher



Mars Rover

PLANNING IN DYNAMIC ENVIRONMENTS: D*



Example

8 possible state transitions, with Euclidean distance as edge cost

#5 iterations

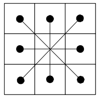
#50 iterations

Red point: Goal state

Green point: Initial state

Blue point: Current state

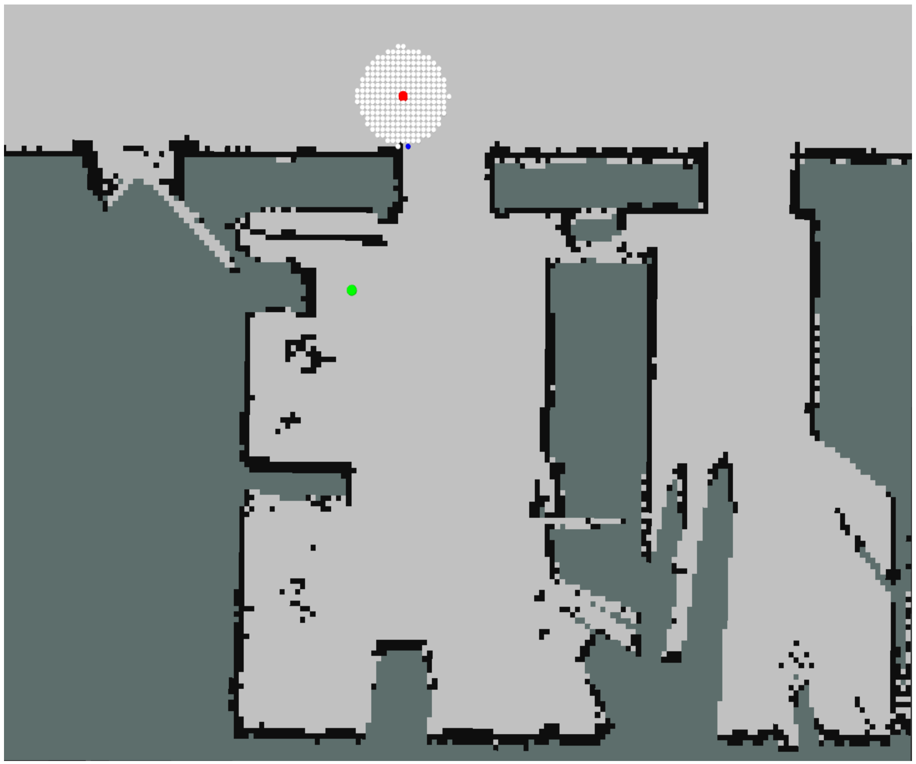
PLANNING IN DYNAMIC ENVIRONMENTS: D*



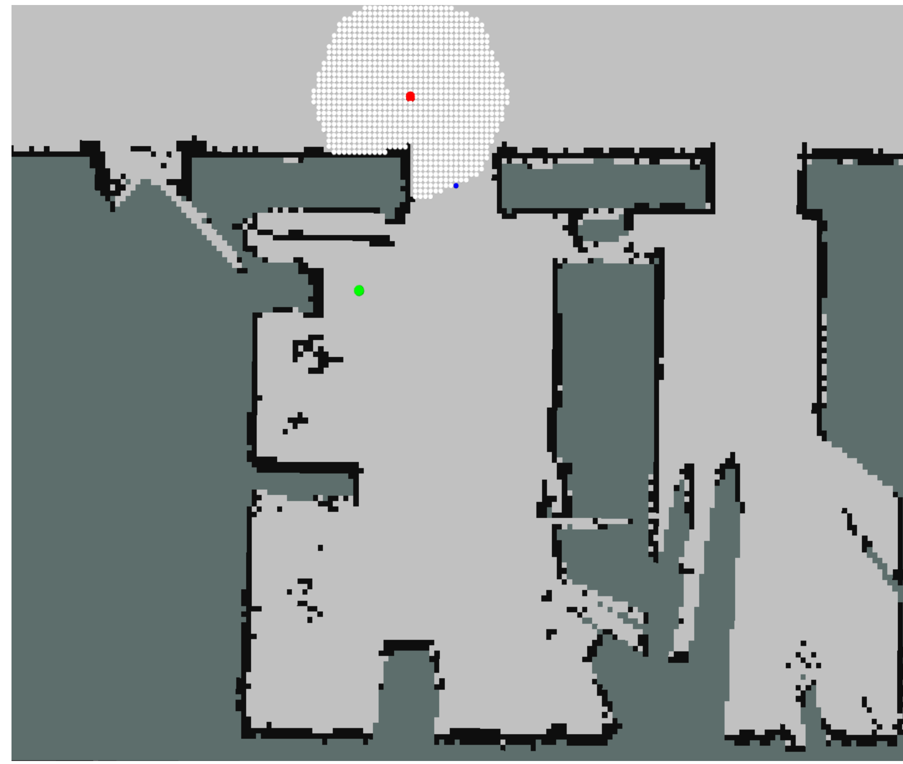
Example (initial planning)

8 possible state transitions, with Euclidean distance as edge cost

#250 iterations



#1000 iterations

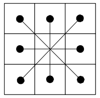


Red point: Goal state

Green point: Initial state

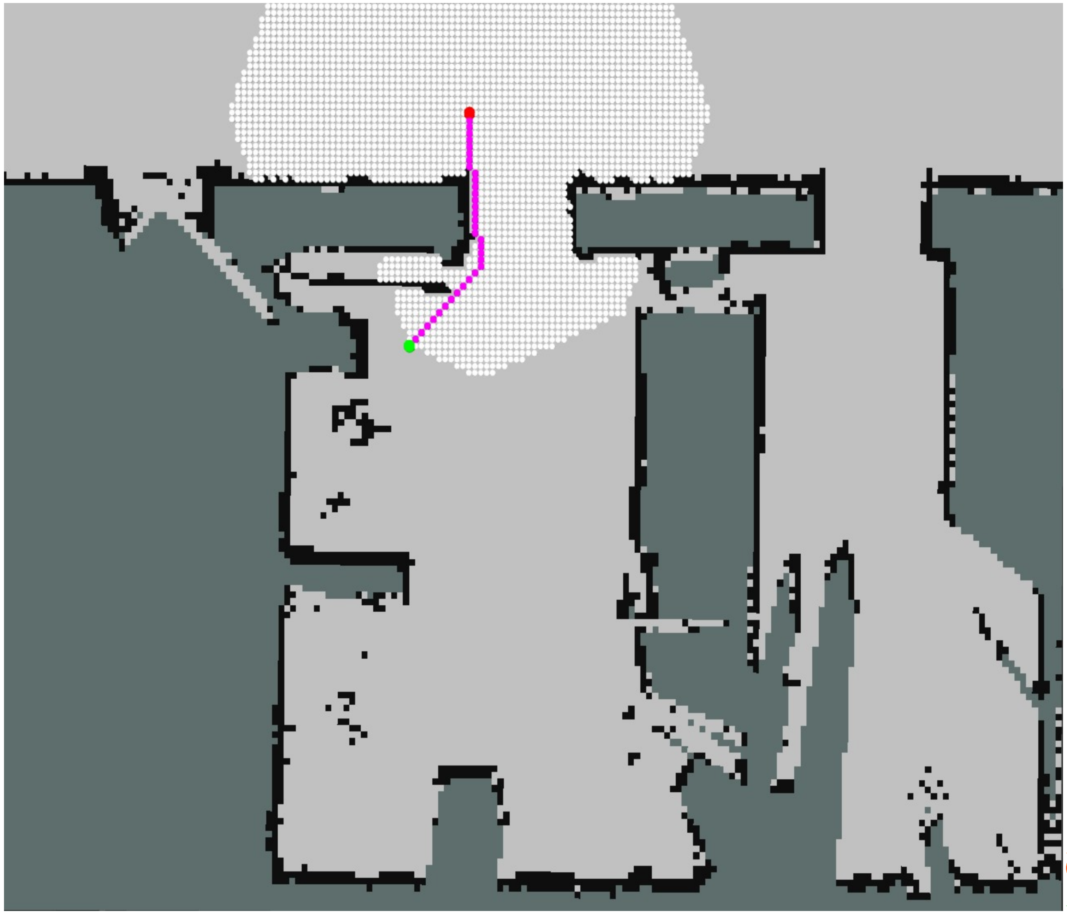
Blue point: Current state

PLANNING IN DYNAMIC ENVIRONMENTS: D*



Example (initial planning)

8 possible state transitions, with Euclidean distance as edge cost



#3512 iterations

Red point: Goal state

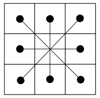
Green point: Initial state

Blue point: Current state

Purple line: plan

GRAPH SEARCH

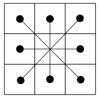
PLANNING IN DYNAMIC ENVIRONMENTS: D*



Example (path execution, step 5)

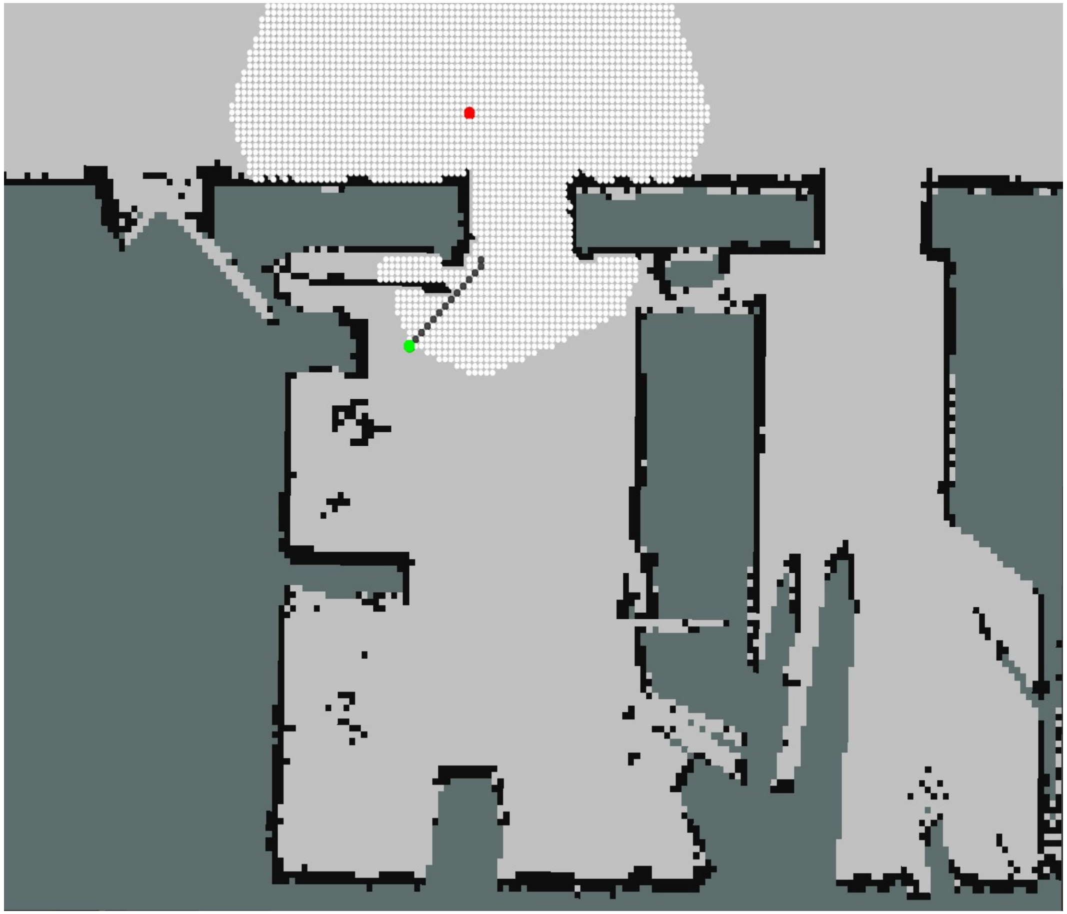
8 possible state transitions, with Euclidean distance as edge cost

PLANNING IN DYNAMIC ENVIRONMENTS: D*



Example (path execution, step 13)

8 possible state transitions, with Euclidean distance as edge cost



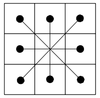
Red point: Goal state

Green point: Initial state

Blue point: Current state

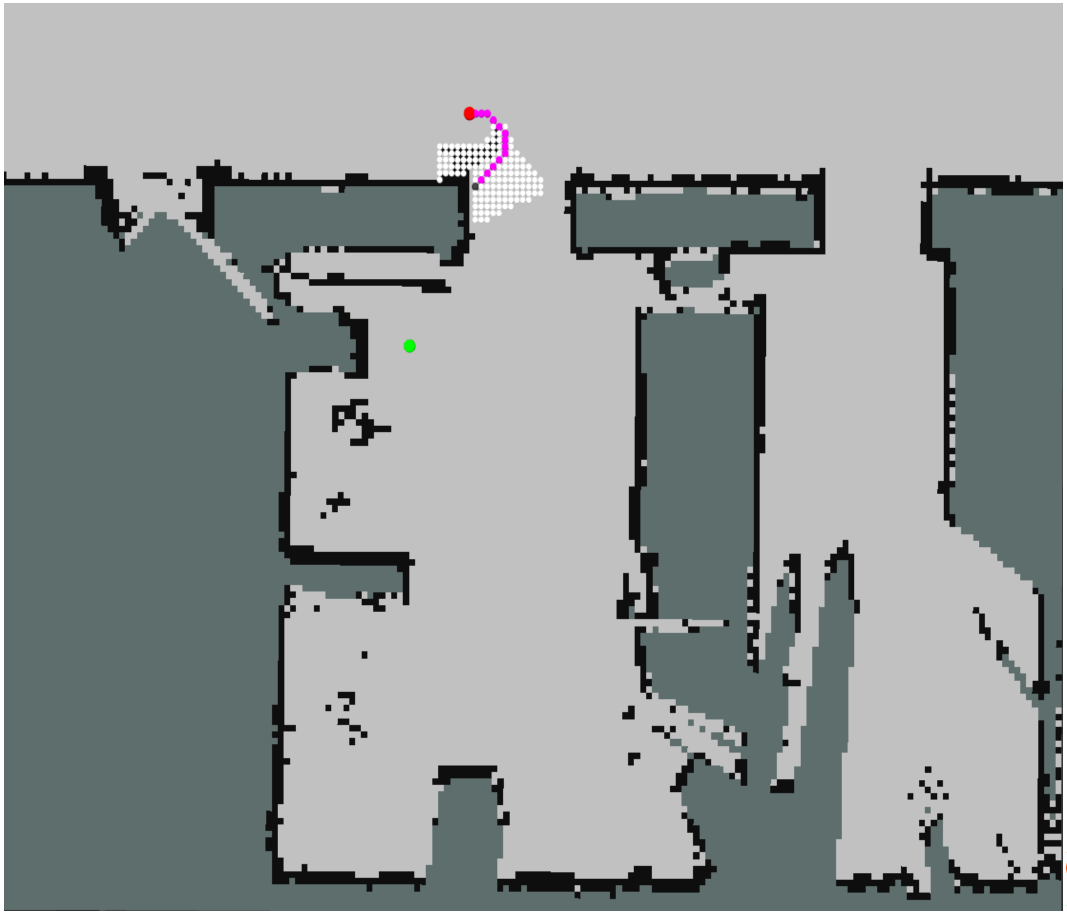
Black line: executed path

PLANNING IN DYNAMIC ENVIRONMENTS: D*



Example (recomputing shortest paths)

8 possible state transitions, with Euclidean distance as edge cost



Red point: Goal state

Green point: Initial state

Blue point: Current state

Purple line: plan

PLANNING IN DYNAMIC ENVIRONMENTS: D*

The success of the **D*** algorithm has inspired several subsequent variations, notably:

- **Focused D***: Anthony Stenz, *The Focused D* Algorithm for Real-Time Replanning*, International Joint Conference on Artificial Intelligence, 1995.

  This algorithm incorporates a heuristic cost-to-go estimate which focuses/biases the graph search towards the goal direction, as is the case for A*

- **D-Lite**: S. Koenig and M. Likhachev, *Fast replanning for navigation in unknown terrain*, IEEE Transactions on Robotics, 2005

  A faster formalization/implementation of the original Focused D* algorithm

CS-4

2. APPLICATION TO ROBOT CAR PARKING

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

APPLICATION TO ROBOT CAR PARKING
PROBLEM DEFINITION

The **task of parking a car** can be formulated as a **discrete planning** problem and solved using graph-search. Accordingly :

1. A non-empty state-space Q, which is a finite or countably infinite set of states

2. For each state $q \in Q$, a finite action space $U(q)$

3. A state transition function f that produces a state $f(q, u) \in Q$ for every $q \in Q$ and $u \in U(q)$, namely, $q' = f(q, u) \in Q$

4. An initial state $q_I \in Q$

5. A goal set $Q_G \subset Q$

Find a finite sequence of actions that, when applied, transforms the initial state $q_I \in Q$ to some state in Q_G

APPLICATION TO ROBOT CAR PARKING
PROBLEM DEFINITION

The **task of parking a car** can be formulated as a **discrete planning** problem and solved using graph-search. Accordingly :

1. A non-empty state-space Q, which is a finite or countably infinite set of states.

OUR JOB

2. For each state $q \in Q$, a finite action space $U(q)$

3. A state transition function f that produces a state $f(q, u) \in Q$ for every $q \in Q$ and $u \in U(q)$, namely, $q' = f(q, u) \in Q$

4. An initial state $q_I \in Q$

5. A goal set $Q_G \subset Q$

Find a finite sequence of actions that, when applied, transforms the initial state $q_I \in Q$ to some state in Q_G

THE JOB OF GRAPH-SEARCH

APPLICATION TO ROBOT CAR PARKING
1. STATE-SPACE DEFINITION

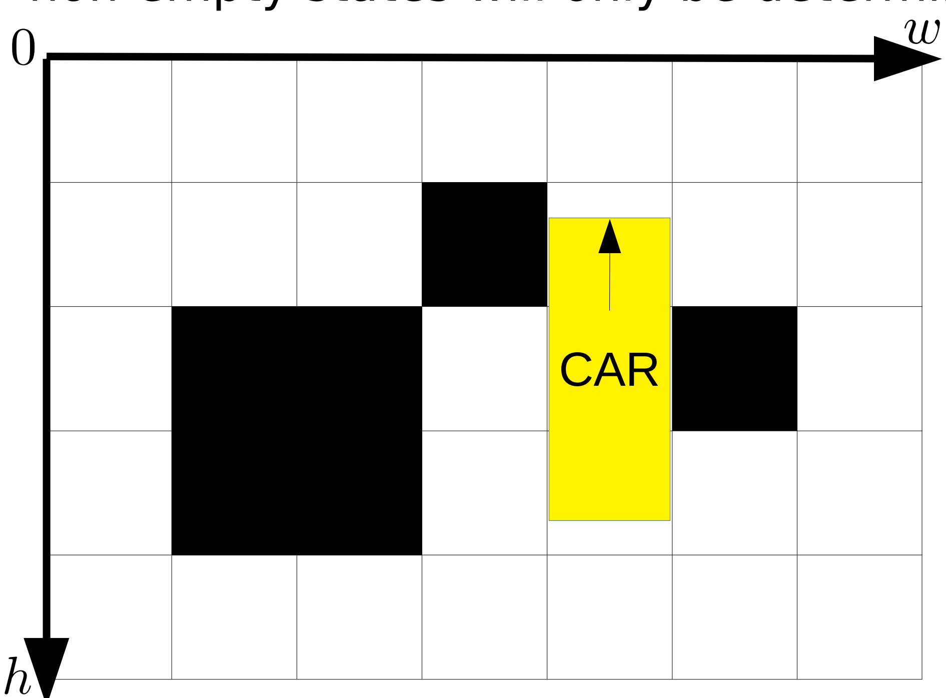
The state of the car has 3DOF and defined as $q = [y, x, \theta]^T$

where $y \in H = \{1, 2, ..., h\}, x \in W = \{1, 2, ..., w\}, \theta \in \Omega = \{0, \omega, ..., 2\pi - \omega\}$

and $\omega = 2\pi/n$. Thus, there can be $w \times h \times n$ possible states

---

Obtaining the entire non-empty state space Q is computationally prohibitive → non-empty states will only be determined <u>on demand</u>

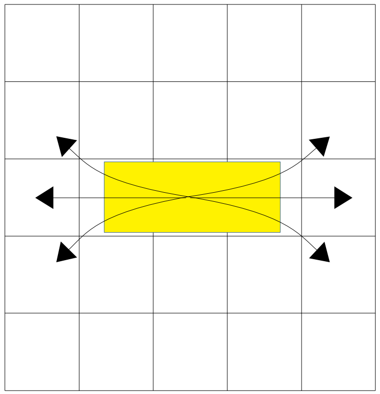


2. ACTION SPACE

A car-like robot can perform linear and circular motions, in forward or backward direction.

- Assuming a fixed linear velocity v_0 and a fixed turning angle ϕ_0, the action space is obtained as:

$$U = \{-v_0, +v_0\} \times \{-\phi_0, 0, +\phi_0\}$$
$$\Rightarrow u \in \{(-v_0, -\phi_0), (-v_0, 0), (-v_0, +\phi_0), (+v_0, -\phi_0), (+v_0, 0), (+v_0, +\phi_0)\}$$

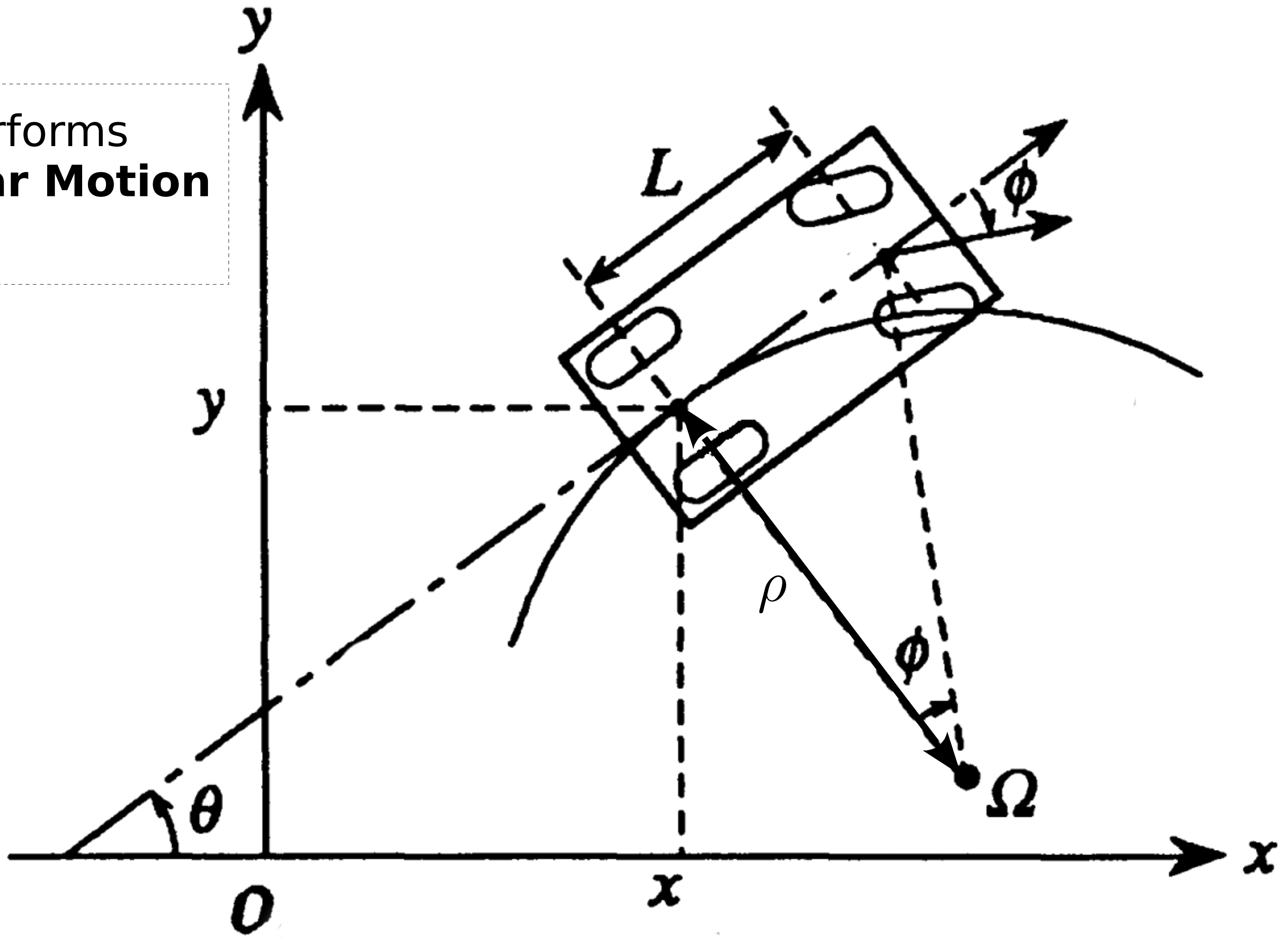


- Motion duration is also fixed at a constant time interval Δt
- Fixing of v_0, ϕ_0 and Δt depends on map resolution (meters/cell side)

3. STATE TRANSITION FUNCTION

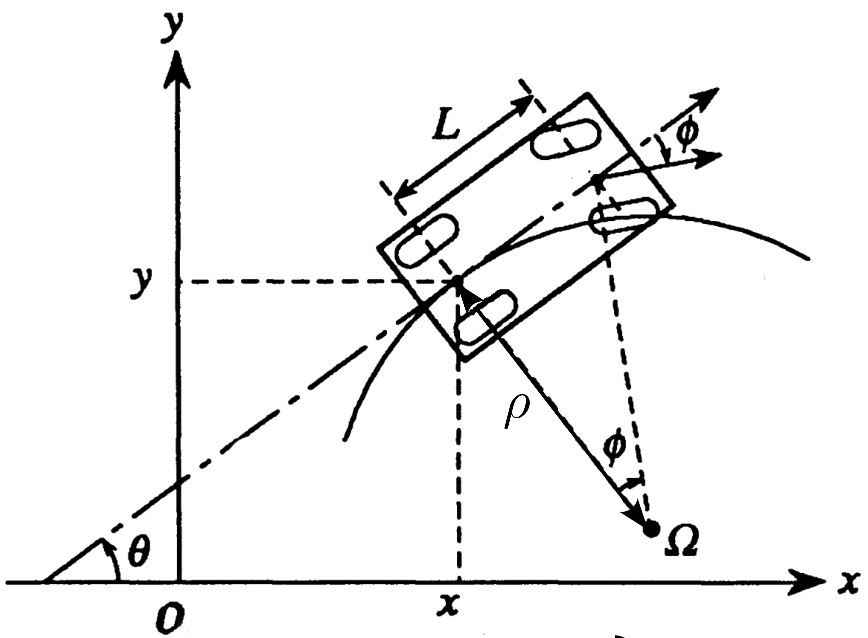
We need to define $f(q_0, u) = q_t = [y_t, x_t, \theta_t]^T = \in Q$

The robot performs
Uniform Circular Motion
(UCM)



3. STATE TRANSITION FUNCTION

We need to define $f(q_0, u) = q_t = [y_t, x_t, \boxed{\theta_t}]^T = \in Q$



$$\theta_t = \theta_0 + \Delta\theta$$

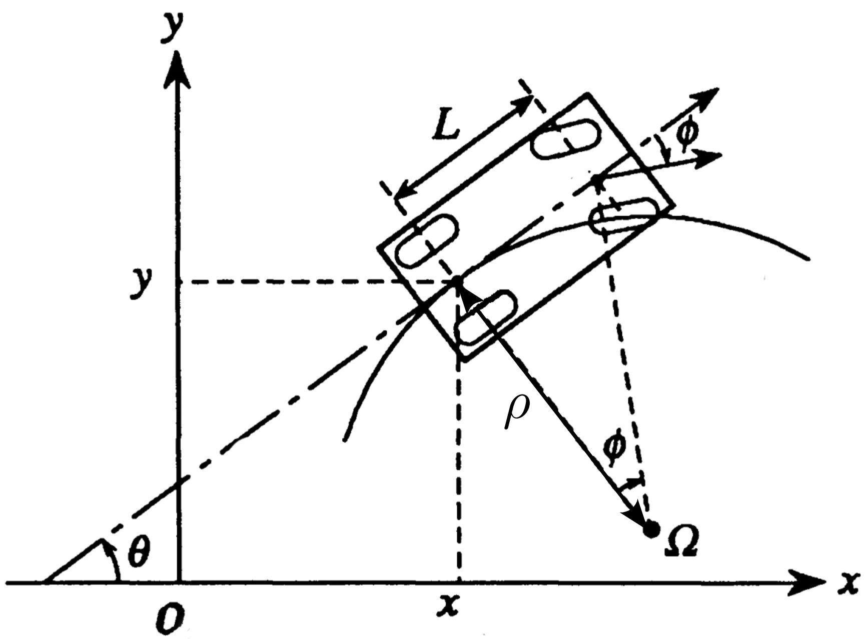
$$\Delta\theta = \int_0^t \dot{\theta} dt \xlongequal[tan\phi = \frac{L}{\rho}]{\dot{\theta} = \frac{v}{\rho}} \int_0^t \frac{v}{L} tan\phi dt = t \cdot \frac{v}{L} tan\phi$$

$$\Rightarrow \boxed{\theta_t} = \theta_0 + t \cdot \frac{v}{L} tan\phi \quad \text{eq. 1}$$

APPLICATION TO ROBOT CAR PARKING

3. STATE TRANSITION FUNCTION

We need to define $f(q_0, u) = q_t = [y_t, \boxed{x_t}, \theta_t]^T = \in Q$



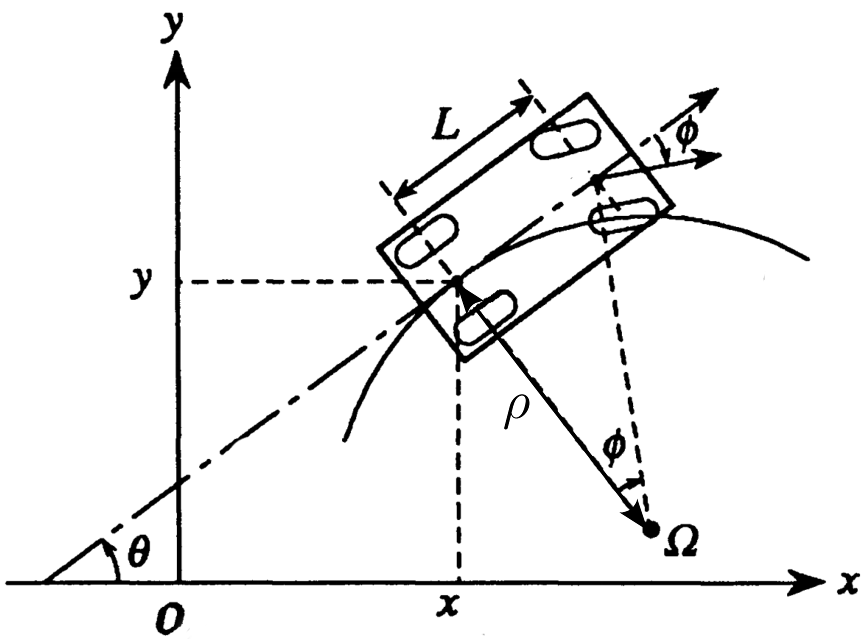
$$x_t = x_0 + \Delta x$$

$$\Delta x = \int_0^t \dot{x} dt = \int_0^t v cos\theta dt \overset{eq.1}{=\!=\!=} \int_0^t v cos(\theta_0 + t \cdot \frac{v}{L} tan\phi) dt =$$

$$\Rightarrow \boxed{x_t} = x_0 + \frac{L}{tan\phi}(sin(\theta_0 + t \cdot \frac{v}{L} tan\phi) - sin\theta_0)$$ eq. 2

APPLICATION TO ROBOT CAR PARKING
3. STATE TRANSITION FUNCTION

We need to define $f(q_0, u) = q_t = [\boxed{y_t}, x_t, \theta_t]^T = \in Q$



$$y_t = y_0 + \Delta y$$

$$\Delta y = \int_0^t \dot{y} dt = \int_0^t v sin\theta dt \overset{eq.1}{=\!=\!=} \int_0^t v cos(\theta_0 + t \cdot \frac{v}{L} tan\phi) dt =$$

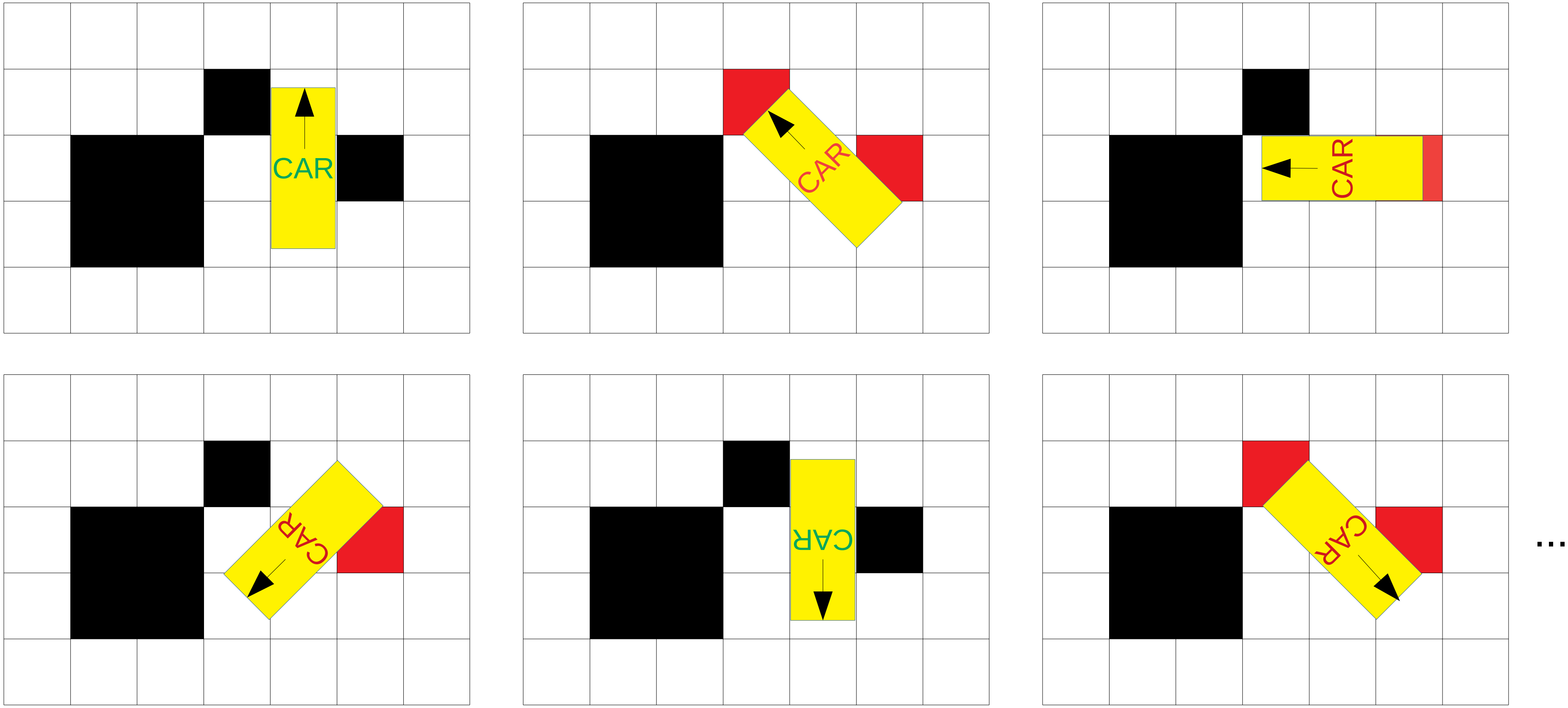
$$\Rightarrow \boxed{y_t} = y_0 - \frac{L}{tan\phi}(cos(\theta_0 + t \cdot \frac{v}{L} tan\phi) - cos\theta_0)$$ eq. 3

3. STATE TRANSITION FUNCTION

To determine whether state q_t is free, **collision checking** is performed between the **robot** car **and** the occupancy **map** M .

Collision checking can be a **costly operation** depending on map and car geometry.

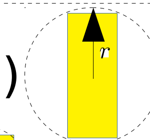
M



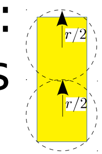
3. STATE TRANSITION FUNCTION

To determine whether state q_t is free, **collision checking** is performed between the **robot** car **and** the occupancy **map** $\mathbf{M}$.

Collision checking can be a **costly operation** depending on map and car geometry. Efficiency can be increased in various ways:

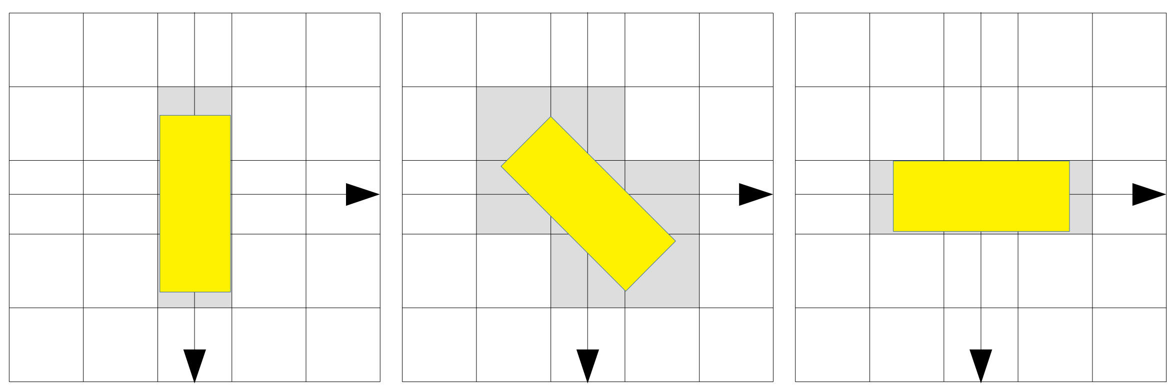


- Assume circular robot of radius r using the bounding sphere (BB)
  If $||(y,x) - \mathbf{M}_{\text{nearest}}(i,j)|| > r$, **then** NO collision. **Otherwise**:
  - Subdivide circular robot in two BBs and recheck collisions



- **Repeat** until no collision found or max. subdivision reached

- Precompute offline, cells occupied by robot for all possible rotations
  - Set $[y,x] = [0,0]$
  - Loop over all θ and determine cells occupied by robot



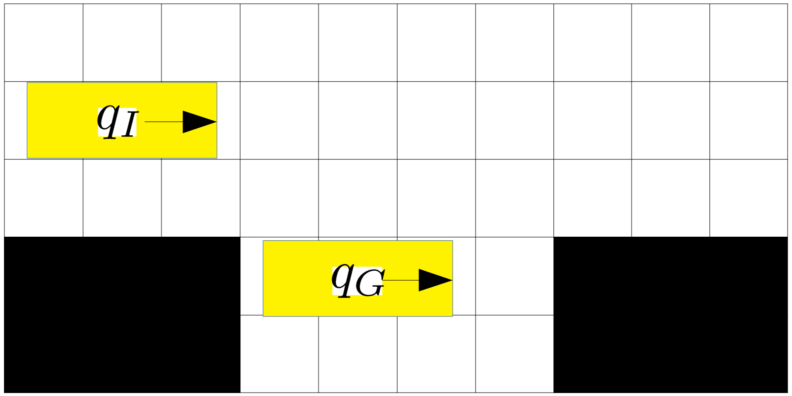
...

4. 5. INITIAL AND GOAL STATE

Initial state: current state of robot car before parking

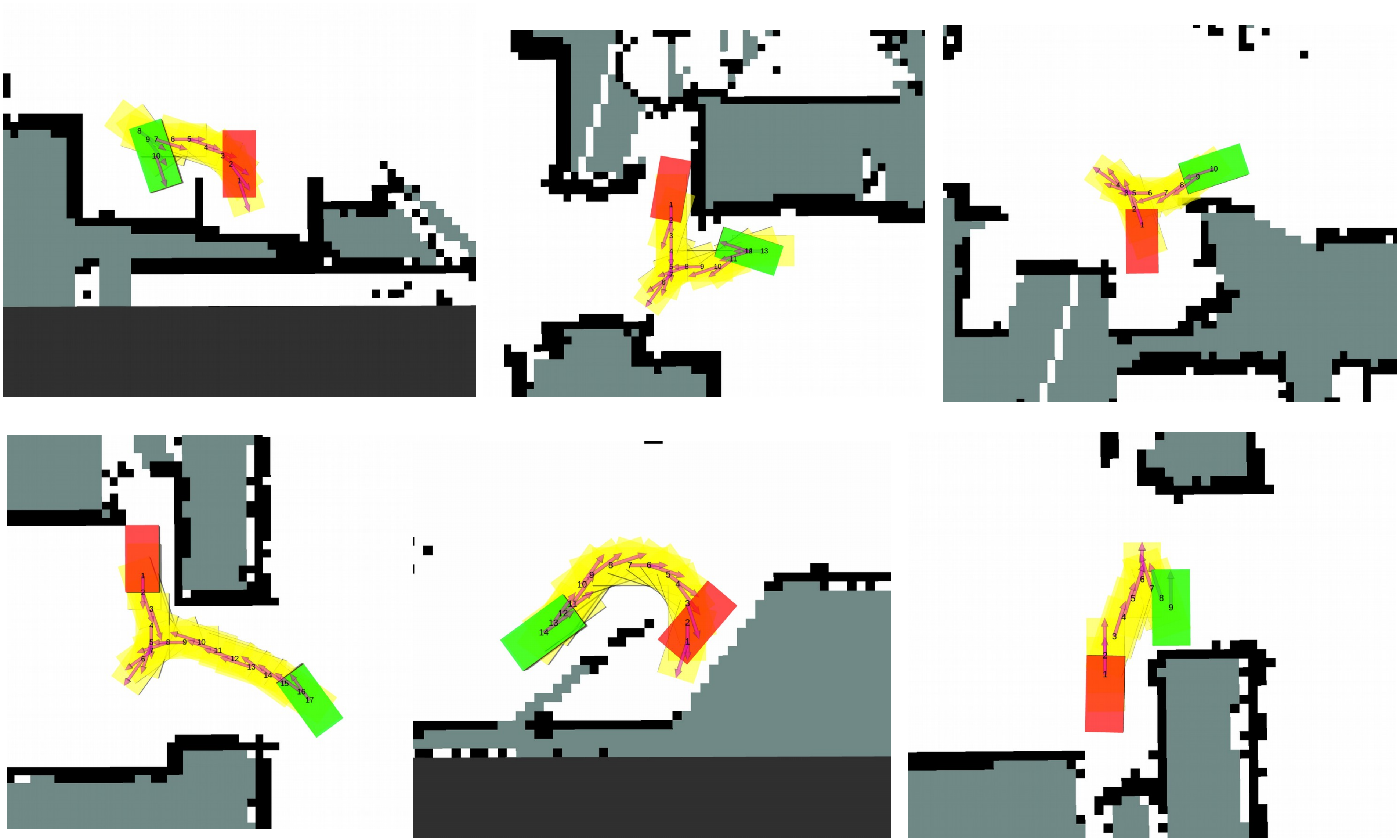
Goal state: desire state for parking of the robot car

Find using graph search, a finite sequence of actions that, when applied, transforms the initial state to the goal state.



APPLICATION TO ROBOT CAR PARKING

EXAMPLES



SUMMARY OF CS 4-B

Upon completion of the CS 4-b, you should be able to:

- Formulate a motion planning problem given a description of the robot and the world it operates in
- Argue on the appropriateness of a search graph design method depending on the problem
- Apply a given graph search algorithm, given the graph, the initial and the goal states

FURTHER READING

- **Tutorial on Motion planning:**
  Motion Planning Part 1: the Essentials (LaValle)

- **Search graph design**
  Mobile Robotics (Kelly), Ch. 10, Sec. 10.1-10.2.1

- **Graph search:**
  Wandering, Systematic Planning, BFS, NF1, DFS, DIJKSTRA, A*, D*****
  - *Mobile Robotics* (Kelly), Ch. 10, Sec. 10.2-10.3

  - *Planning Algorithms* (LaValle), Ch. 2 (Discrete Planning), Sec. 2.1-2.2