

Background on Convex Sets

Robot Planning and its Applications

University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)



Outline

Introduction to Convex Polyhedra and Sets

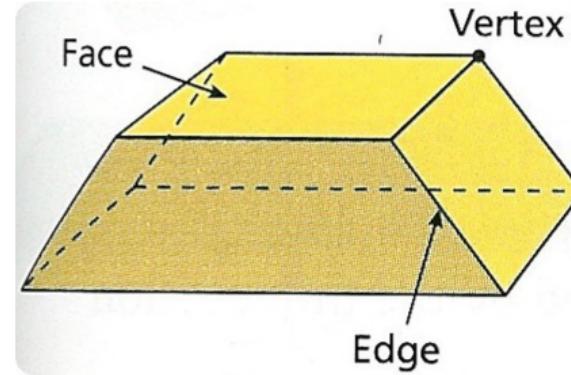
Hyperplanes and friends

Questions and exercises

Convex Polyhedra

- ▶ A **polyhedron** is a solid enclosed by polygonal faces that form a single connected region of space.
- ▶ An **edge** of a polyhedron is a line segment formed by the intersection of two faces.
- ▶ A **vertex** of a polyhedron is a point where edges meet (typically three or more)

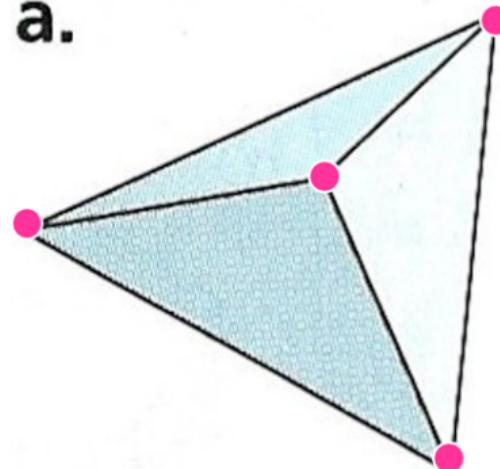
Parts of a Polyhedron



Example (1)

- ▶ How many Vertices?

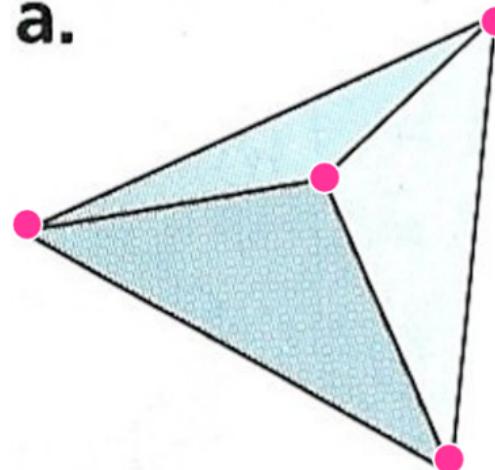
a.



Example (1)

- ▶ How many Vertices?

a.

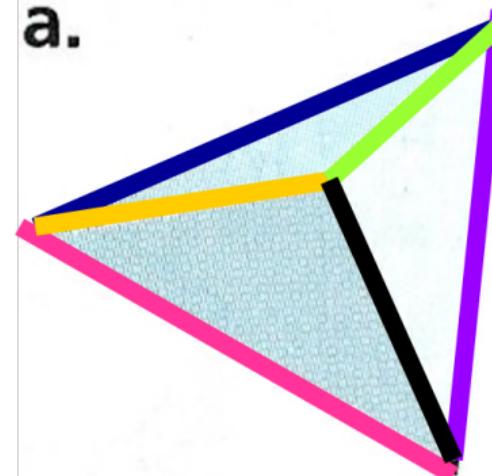


Four vertices.

Example (2)

- ▶ How many Edges?

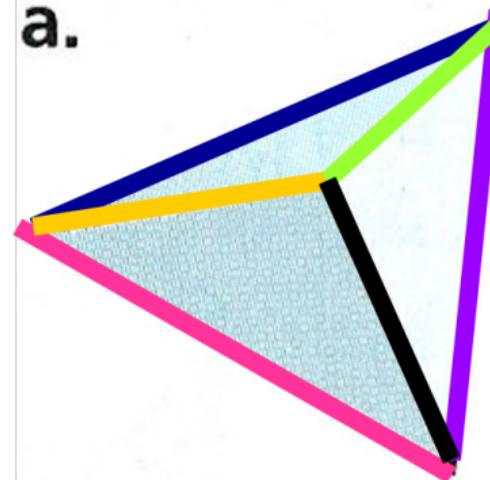
a.



Example (2)

- ▶ How many Edges?

a.

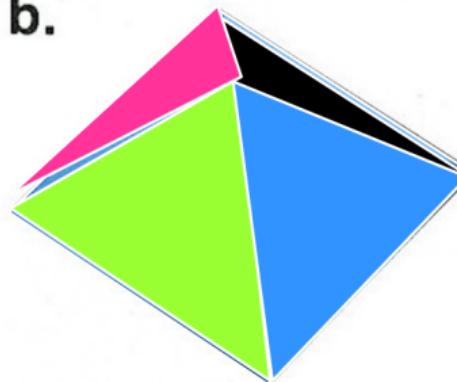


Six edges.

Example (3)

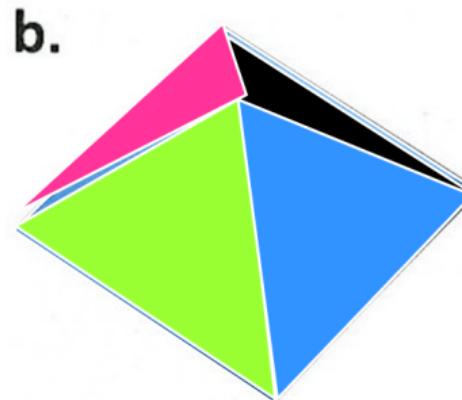
► How many Faces?

b.



Example (3)

- ▶ How many Faces?



Five faces.

A Pattern

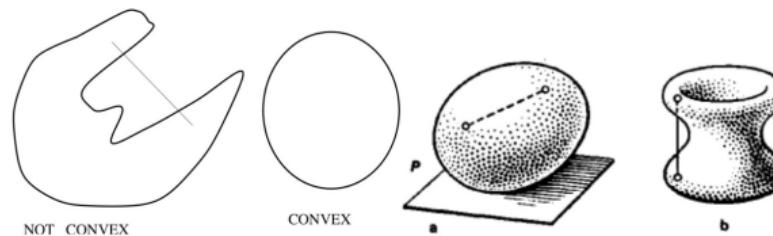
Euler Theorem

The number of faces (F), vertices (V) and edges (E) of a polyhedron are related by Euler's formula:

$$F + V = E + 2$$

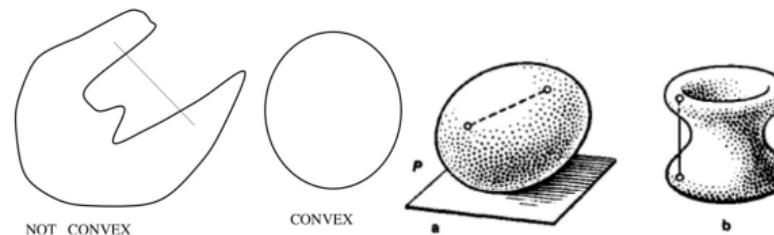
Convex Sets

A set is **convex** if, for any two points in the set, the line segment connecting them is also contained in the set.



Convex Sets

A set is **convex** if, for any two points in the set, the line segment connecting them is also contained in the set.

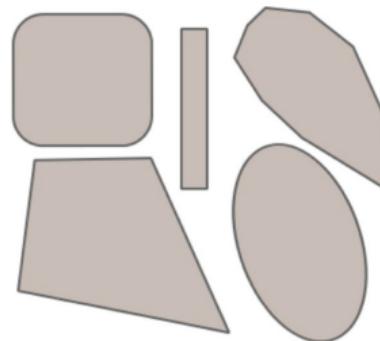


Equation of line segment connecting two points:

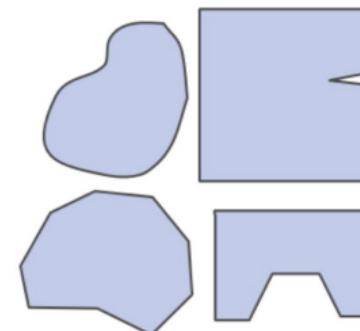
$$\alpha p + (1 - \alpha)q, \alpha \in [0, 1]$$

Examples of Convex Sets

Convex Solids



Non-convex solids





Outline

Introduction to Convex Polyhedra and Sets

Hyperplanes and friends

Questions and exercises

Hyperplanes and Half-Spaces

- ▶ A **hyperplane** is the set of points that satisfy a linear equation:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

- ▶ A **half-space** is the set of points that satisfy a linear inequality:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$

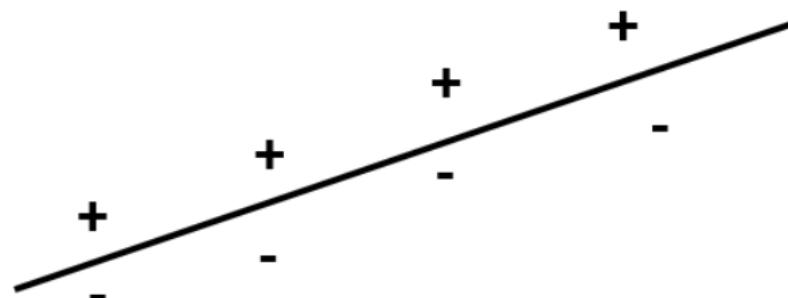
Hyperplanes and Half-Spaces

- ▶ A **hyperplane** is the set of points that satisfy a linear equation:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

- ▶ A **half-space** is the set of points that satisfy a linear inequality:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$



Supporting Hyperplanes and Faces

- ▶ A linear inequality $f(x) \leq \alpha$ is **valid** for a convex set S if every point of S satisfies it.

Supporting Hyperplanes and Faces

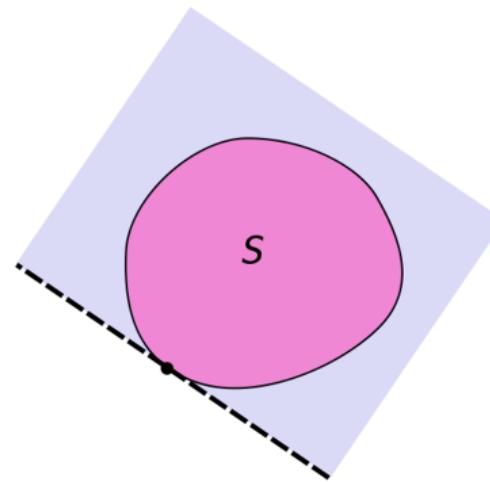
- ▶ A linear inequality $f(x) \leq \alpha$ is **valid** for a convex set S if every point of S satisfies it.
- ▶ A face F of a set S (with $F \subseteq S$) is the set of points where a valid inequality becomes an equality, i.e., $f(x) = \alpha, \forall x \in F$

Supporting Hyperplanes and Faces

- ▶ A linear inequality $f(x) \leq \alpha$ is **valid** for a convex set S if every point of S satisfies it.
- ▶ A face F of a set S (with $F \subseteq S$) is the set of points where a valid inequality becomes an equality, i.e., $f(x) = \alpha, \forall x \in F$
- ▶ A **supporting hyperplane** of S is $f(x) = \alpha$ if S lies within one closed half-space defined by $f(x) \leq \alpha$ and touches S at least at one boundary point.

Example

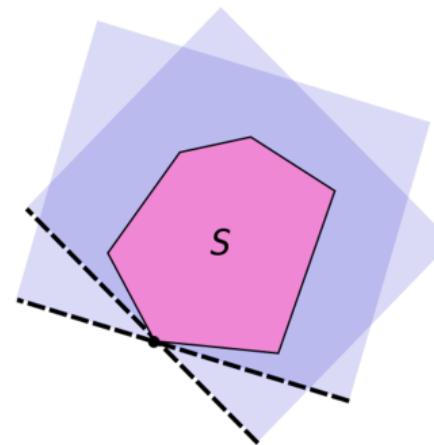
- ▶ Example of a supporting hyperplane



A theorem

Theorem of Supporting Hyperplanes

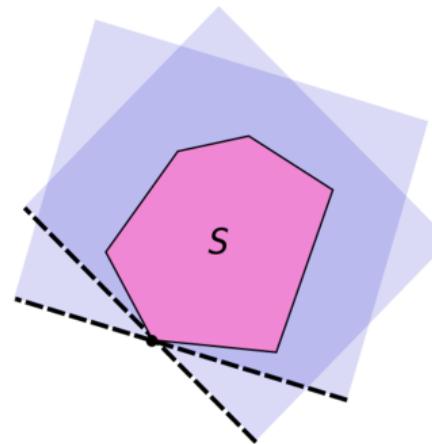
If S is convex and x_0 lies on its boundary, then there exists a supporting hyperplane containing x_0 .



A theorem

Theorem of Supporting Hyperplanes

If S is convex and x_0 lies on its boundary, then there exists a supporting hyperplane containing x_0 .



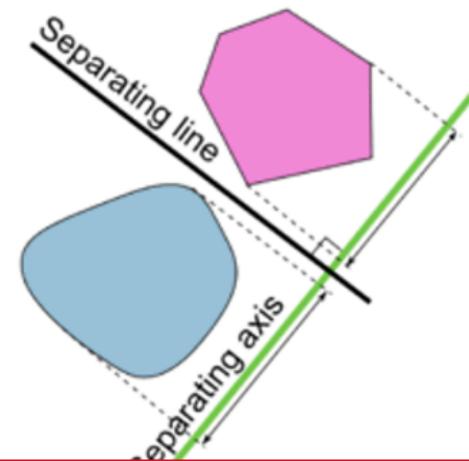
Observation

There can be more than one supporting hyperplane through a boundary point.

Related Results

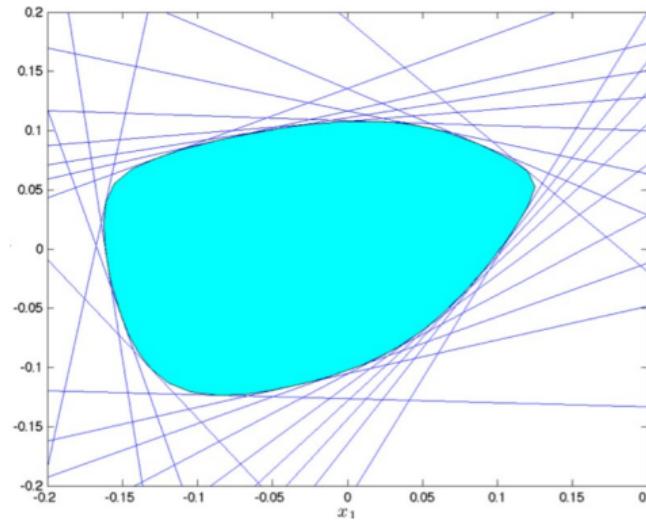
- ▶ If a closed set S has a supporting hyperplane through every boundary point, then S is convex.
- ▶ If S_1 and S_2 are two disjoint convex sets, then there exists a hyperplane $f(x) = \alpha$ that separates them:

$$\begin{array}{ll} f(x) \leq \alpha & \forall x \in S_1 \\ f(x) \geq \alpha & \forall x \in S_2 \end{array}$$



Related Results

- ▶ Any convex set can be expressed as the intersection of its closed supporting half-spaces.



Convex Polyhedra

A convex polyhedron is the intersection of a finite number of half-spaces.



Linear Solvability (Feasibility) Problem

Given a set of linear inequalities, determine whether there exists a feasible solution.

$$a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n \leq b_1$$

$$a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n \leq b_2$$

...

$$a_{p,1}x_1 + a_{p,2}x_2 + \dots + a_{p,n}x_n \leq b_p$$

This is equivalent to checking whether the corresponding polyhedron is empty.

Convex Functions

- ▶ A function $f(x)$ is **convex** if its graph lies below the straight line between any two points: for all $x, y \in \text{dom}f$ and $t \in [0, 1]$:

$$f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y)$$



- ▶ A function $f(x)$ is **concave** if for all $x, y \in \text{dom}f$ and $t \in [0, 1]$:

$$f(tx + (1 - t)y) \geq tf(x) + (1 - t)f(y)$$

Example of convex functions

► Affine functions

$$a_i^T x + b_i$$

are always convex

Example of convex functions

- ▶ **Affine functions**

$$a_i^T x + b_i$$

are always convex

- ▶ **Quadratic functions**

$$x^T Q_i x + a_i^T x + b_i$$

are convex if and only if $Q_i \succeq 0$ (i.e., Q_i is positive semidefinite).

Example of convex functions

► Affine functions

$$a_i^T x + b_i$$

are always convex

► Quadratic functions

$$x^T Q_i x + a_i^T x + b_i$$

are convex if and only if $Q_i \succeq 0$ (i.e., Q_i is positive semidefinite).

Positive semidefinite matrices

A matrix Q is positive semidefinite if $\forall x$

$$x^T Q x \geq 0.$$

It can be shown that a matrix is positive semidefinite iff all its eigenvalues are non-negative.

Convex Optimisation (2)

- ▶ A standard way of expressing convex optimisation problems is

$$\min f(x)$$

$$f_i(x) \leq 0, \quad i = 1, 2, \dots, m$$

$$a_i^T x = b_i \quad i = 1, 2, \dots, p$$

where

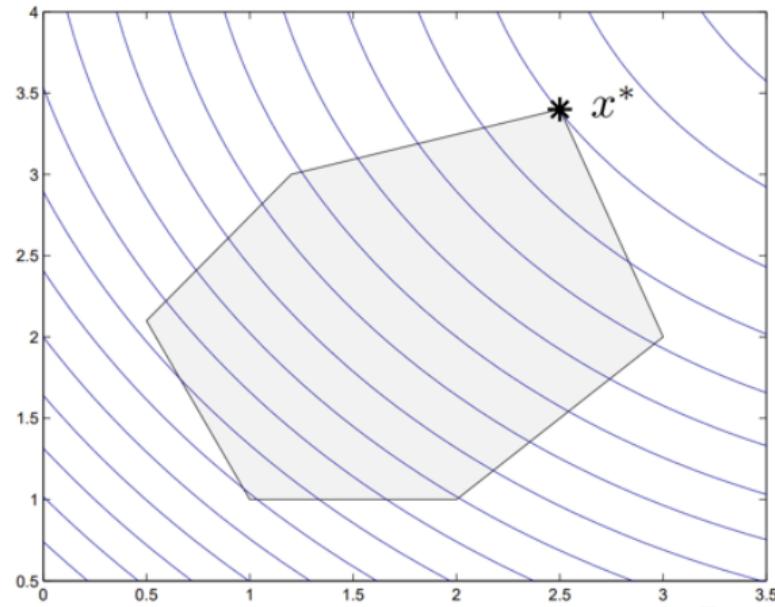
$f_i(x)$ is a convex function

$a_i^T x = b_i$ are equality constraints

- ▶ The feasible domain defined in this way is indeed convex (prove it).

A key fact

- ▶ In convex optimisation problems, any local optimum is also a global optimum.



Example of Convex optimisation problems

$$\min f(x)$$

$$f_i(x) \leq 0, \quad i = 1, 2, \dots, m$$

$$a_i^T x = b_i \quad i = 1, 2, \dots, p$$

Linear Programs (LP): $\rightarrow \begin{cases} f(x) \text{ affine} \\ f_i(x) \text{ affine} \end{cases}$

Quadratic Programs (QP): $\rightarrow \begin{cases} f(x) \text{ quadratic (convex)} \\ f_i(x) \text{ affine} \end{cases}$

Quadratically Constrained Quadratic Programs (QCQP): $\rightarrow \begin{cases} f(x) \text{ quadratic (convex)} \\ f_i(x) \text{ quadratic (convex)} \end{cases}$



Solvers

- ▶ All the problems above are convex: a local optimiser is also a global optimiser
- ▶ Very efficient tools exist for LP, QP and QCQP that produce solutions in polynomial time with respect to the number of variables and constraints.

Linear Programs

► Standard primal form:

$$\begin{aligned}
 & \min c_1x_1 + c_2x_2 + \dots + c_nx_n \quad \text{s.t.} \\
 & a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n}x_n \leq b_1 \\
 & a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n}x_n \leq b_2 \\
 & \quad \dots \\
 & a_{p,1}x_1 + a_{p,2}x_2 + \dots + a_{p,n}x_n \leq b_p
 \end{aligned}$$

which can be written more compactly as:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,N} \\ \dots & & & \\ a_{p,1} & a_{p,2} & \dots & a_{p,N} \end{bmatrix} \quad c = \begin{bmatrix} c_1 \\ \dots \\ c_N \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ \dots \\ b_p \end{bmatrix}$$

► Feasibility problem:

$$\min 0 \quad \text{s.t. } Ax \leq b$$

Solved only to verify if the polyhedron is empty or not

Quadratic Programs

► Standard form:

$$\begin{aligned} \min & x^T Qx + c^T x \\ & Ax \leq b \end{aligned}$$

with $Q \succeq 0$ being symmetric and positive semidefinite, which qualifies the QP as a convex problem.

Quadratic Programs

► Standard form:

$$\begin{aligned} \min & x^T Qx + c^T x \\ & Ax \leq b \end{aligned}$$

with $Q \succeq 0$ being symmetric and positive semidefinite, which qualifies the QP as a convex problem.

- If Q is positive definite it is possible to apply Cholesky decomposition: $Q = R^T R$ and the problem becomes equivalent to the following least-squares minimisation problem.

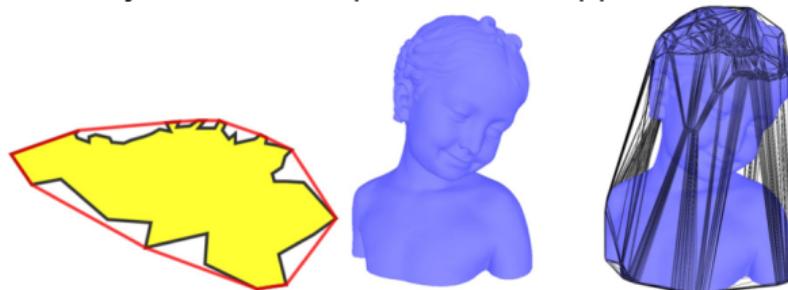
$$\min \frac{1}{2} \|Rx - d\|^2 \quad Ax \leq b$$

with $c = -R^T d$.

Convex Hulls

► Not

all sets are convex, but every set can be represented or approximated by a convex set.



- The **convex hull** of a set S , denoted $\text{conv}(S)$, is the intersection of all convex sets containing S .



An interesting result

Given points x_1, x_2, \dots, x_n , a convex combination is defined as $\sum_{i=1}^n \gamma_i x_i$, with $\gamma_i \geq 0$ and $\sum_{i=1}^n \gamma_i = 1$

An interesting result

Given points x_1, x_2, \dots, x_n , a convex combination is defined as $\sum_{i=1}^n \gamma_i x_i$, with $\gamma_i \geq 0$ and $\sum_{i=1}^n \gamma_i = 1$

Lemma

Given a set of points $A = \{x_1, \dots, x_n\}$. Its convex hull $\text{conv}(A)$ is given by all convex combination of points in A:

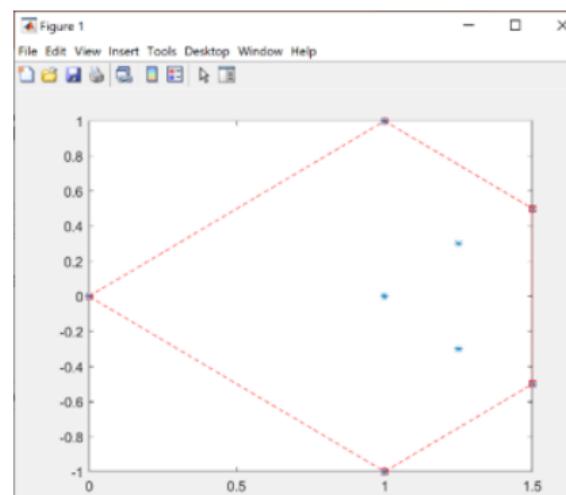
$$\text{conv}(A) = \left\{ \sum_{x_i \in A} \gamma_i x_i, \gamma_i \geq 0, \sum_{i=1}^n \gamma_i = 1 \right\}$$

MATLAB Example: Convex Hull

```
1 P = [0 0; 1 1; 1.5 0.5; 1.5 -0.5; 1.25 0.3; 1 0; 1.25
      -0.3; 1 -1];
2 plot(P(:,1), P(:,2), '*');
3 hold on;
4 [k, av] = convhull(P);
5 plot(P(k,1),P(k,2), 'rs');
```

MATLAB Example: Convex Hull

```
1 P = [0 0; 1 1; 1.5 0.5; 1.5 -0.5; 1.25 0.3; 1 0; 1.25
      -0.3; 1 -1];
2 plot(P(:,1), P(:,2), '*');
3 hold on;
4 [k, av] = convhull(P);
5 plot(P(k,1),P(k,2), 'rs');
```



Example Application 1

- ▶ There are two ways to express a convex polyhedron.
 1. A set of linear inequalities
 2. A set of vertices
- ▶ Example code: given a set of vertices, write a matlab function that expresses the polyhedron in terms of linear inequalities

Possible solution for the 2D case

```
function [A,b] = myPointToMatrix(V, draw)

Vx = mean(V);

k = convhull(V);
if draw==1,
    plot(V(:,1), V(:,2), '*');
    hold on;
end
A = []; b = [];
for i = 1:length(k),
    i1 = max([1,mod(i+1, length(k))]);
    x0 = V(k(i),1); y0 = V(k(i),2);
    x1 = V(k(i1),1);y1 = V(k(i1),2);
    if draw,
        plot([x0, x1], [y0, y1], '--rs');
    end
    ac=[(y1-y0), -(x1-x0)];
    bc = (y1-y0)*x0-(x1-x0)*y0;
    S = sign(bc-ac*Vx');
    A = [A;S*ac]; b = [b;S*bc];
end
```

Example Application 2

- ▶ Find if a point is inside a polyhedron
- ▶ If the polyhedron is given by a set of inequalities, just plug the coordinates and verify that all are satisfied
- ▶ If the polyhedron is given by its vertices
 1. Compute the convex hull (using a standard algorithm)
 2. Find the inequalities associated with each pair of points
 3. Check whether the point satisfies the inequalities

Example Application 2

- ▶ Find if a point is inside a polyhedron
- ▶ If the polyhedron is given by a set of inequalities, just plug the coordinates and verify that all are satisfied
- ▶ If the polyhedron is given by its vertices
 - 1. Compute the convex hull (using a standard algorithm)
 - 2. Find the inequalities associated with each pair of points
 - 3. Check whether the point satisfies the inequalities

Exercise

Write a MATLAB scripts that implements this procedure.



Example Application 3

(For example to see if the robot is colliding with an obstacle (robot and obstacle are represented as Polyhedra)

- ▶ Determine whether two convex Polyhedra intersect
 - ▶ **Strategy 1:** set up a linear feasibility problem putting together the inequalities of the two polyhedra
 - ▶ **Strategy 2:** apply the theorem of separating hyperplanes. If you find an edge of P_1 such that all vertices of P_2 lie outside, we have found a separating hyperplane.

If we have a large number of vertices is better to use a linear problem

Example Application: Strategy 1

```
1 V1 = (rand(20,2)*5 + [3.0,3.0]);  
2 V2 = (rand(20,2)*5 + [-1,-1]);  
3 plot(V1(:,1), V1(:,2), 'r+'); hold on;  
4 plot(V2(:,1), V2(:,2), 'bx');  
5 [A1,b1] = myPointToMatrix(V1,1);  
6 [A2,b2] = myPointToMatrix(V2,1);  
7 [x] = linprog([0,0],[A1;A2],[b1;b2]);
```

Example Application 4

- ▶ Find the distance between a point and a polyhedron.

Example Application 4

- ▶ Find the distance between a point and a polyhedron.
- ▶ Strategy: we can set up a convex QP

$$\min \frac{1}{2} \|x - x_0\|^2$$
$$Ax \leq b$$

x₀ is the point

which can be rewritten as:

Quadratic optimisation problem

$$\min \frac{1}{2} x^T x - x_0^T x$$
$$Ax \leq b$$

Example Application: Distance Point-Polyhedron

```
1 V1 = (rand(20,2)*5 + [3.0 ,3.0]);  
2 xt = (rand(1,2)*5 + [-1,-1]);  
3 plot(V1(:,1), V1(:,2), 'r+' ); hold on;  
4 plot(xt(:,1), xt(:,2), 'bx' );  
5 [A1,b1] = myPointToMatrix(V1,1);  
6 [x] = quadprog(0.5*eye(2,2), -xt', A1, b1);  
7 plot([xt(1),x(1)],[xt(2),x(2)], '-g' );  
8 axis equal;
```

Example Application 5

- ▶ Find the distance between two polyhedra

Example Application 5

- ▶ Find the distance between two polyhedra
- ▶ Strategy: we can set up a convex QP

x is on one polyhedra and y is a point of the other polyhedra

$$\min \frac{1}{2} \|x - y\|^2$$

$$A_1 x \leq b_1$$

$$A_2 y \leq b_2$$

which can be rewritten as:

$$\min \frac{1}{2} [x^T y^T] \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$Ax \leq b$$

Example Application: Distance Between Polyhedra

```
1 V1 = (rand(20,2)*5 + [3.0,3.0]);  
2 V2 = (rand(20,2)*5 + [-1,-1]);  
3 plot(V1(:,1), V1(:,2), 'r+'); hold on;  
4 plot(V2(:,1), V2(:,2), 'bx');  
5 [A1,b1] = myPointToMatrix(V1,1);  
6 [A2,b2] = myPointToMatrix(V2,1);  
7 Q = [1 0 -1 0; 0 1 0 -1];  
8 [z] = quadprog(Q'*Q, zeros(4,1), ...  
9 [A1, zeros(size(A1)); ...  
10 zeros(size(A2)), A2], [b1;b2]);  
11 plot([z(1), z(3)],[z(2),z(4)], '—g');  
12 axis equal;
```



Outline

Introduction to Convex Polyhedra and Sets

Hyperplanes and friends
Questions and exercises

Check Questions (1)

► Convex Sets

- ▶ Give an example of a set that is convex.
- ▶ Give an example of a set that is not convex and explain why.

► Euler's Formula

- ▶ Verify Euler's formula ($F + V = E + 2$) for a cube.

Check Questions (2)

► Supporting Hyperplanes

- ▶ Why does every convex set have a supporting hyperplane at a boundary point?
- ▶ Can a boundary point belong to more than one supporting hyperplane? Give an example.

► Convexity Test

- ▶ Is the set $\{(x, y) \mid x^2 + y^2 \leq 1\}$ convex?
- ▶ Is the set $\{(x, y) \mid x^2 + y^2 = 1\}$ convex?

Level sets of a convex function are convex, but
level curves of a convex function are not
convex

Exercises (1)

► Convex Functions

- ▶ Show that $f(x) = |x|$ is convex.
- ▶ Show that $f(x) = -x^2$ is not convex.

► Polyhedron Membership

- ▶ Given the inequalities $x + y \leq 2$, $x \geq 0$, $y \geq 0$, check if the point $(1, 1)$ belongs to the polyhedron.

Exercises (2)

► MATLAB: Convex Hull

- ▶ Generate 10 random 2D points and plot their convex hull.
- ▶ Extend your code to check if a test point lies inside the hull.

► Distance to a Polyhedron

- ▶ Using quadprog, compute the distance between $(2, 2)$ and the polyhedron $x \geq 0, y \geq 0, x + y \leq 1$.

Advanced Challenge

► Convex Optimisation

- Solve the quadratic program:

$$\min x^2 + y^2 \quad \text{s.t. } x + y \geq 1, \quad x \geq 0, \quad y \geq 0.$$

- Show that the solution is a *global* optimum.



UNIVERSITÀ DEGLI STUDI DI TRENTO

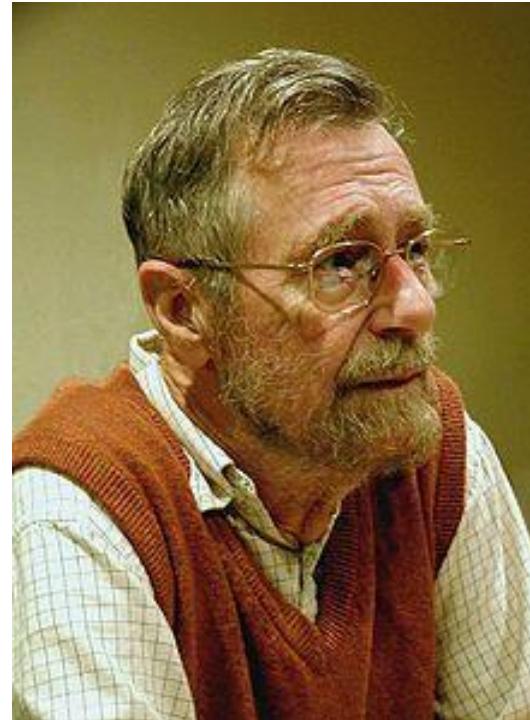
Dipartimento di Ingegneria
e Scienza dell'Informazione

Background: Graph Optimisation

Luigi Palopoli (adapted from Laksman Veeravagu and Luis Barrera et al.)
DISI – UNITN 2025



The author: Edsger Wybe Dijkstra



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

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



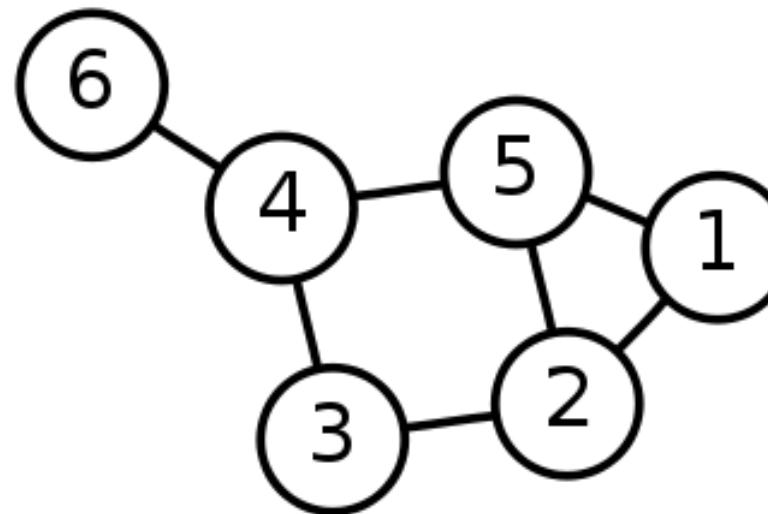
Edsger Wybe Dijkstra

- *1972 A. M. Turing Award (the most prestigious award in computer science).*
- *Held the Schlumberger Centennial Chair of Computer Sciences at the University of Texas at Austin (1984–2000).*
- *Advocated against the GOTO statement in programming, influencing its depreciation.*
- *Renowned for essays on programming and algorithms.*



Single-Source Shortest Path Problem

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





Dijkstra's algorithm

Dijkstra's algorithm - solves the single-source shortest path problem in graph theory.

It works on both directed and undirected graphs, provided that all edges have nonnegative weights.

Approach: Greedy

Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices



Dijkstra's algorithm - Pseudocode

```
dist[s] ← 0  
for all  $v \in V - \{s\}$   
    do dist[v] ←  $\infty$   
 $S \leftarrow \emptyset$   
 $Q \leftarrow V$   
while  $Q \neq \emptyset$   
do  $u \leftarrow \text{mindistance}(Q, \text{dist})$   
     $S \leftarrow S \cup \{u\}$   
    for all  $v \in \text{neighbors}[u]$   
        do if  $\text{dist}[v] > \text{dist}[u] + w(u, v)$            (if a shorter path is found)  
            then  $d[v] \leftarrow d[u] + w(u, v)$    (Update shortest path estimate)  
                  (if desired, add traceback code)  
return dist
```

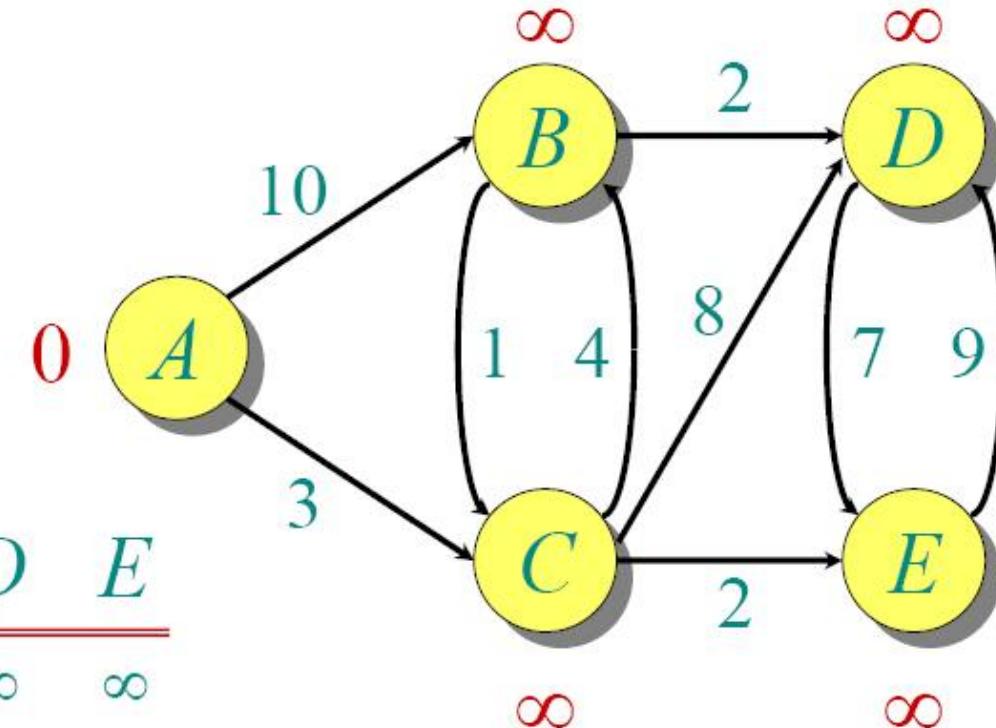


Dijkstra Animated Example

```
dist[s] ← 0
for all  $v \in V - \{s\}$ 
    do dist[v] ←  $\infty$ 
S ←  $\emptyset$ 
Q ← V
while Q ≠  $\emptyset$ 
do    $u \leftarrow \text{mindistance}(Q, \text{dist})$ 
      S ← S ∪ {u}
      for all  $v \in \text{neighbors}[u]$ 
          do if dist[v] > dist[u] + w(u, v)
              then   d[v] ← d[u] + w(u, v)
return dist
```

Initialize:

$Q:$	A	B	C	D	E
	0	∞	∞	∞	∞



$S: \{\}$

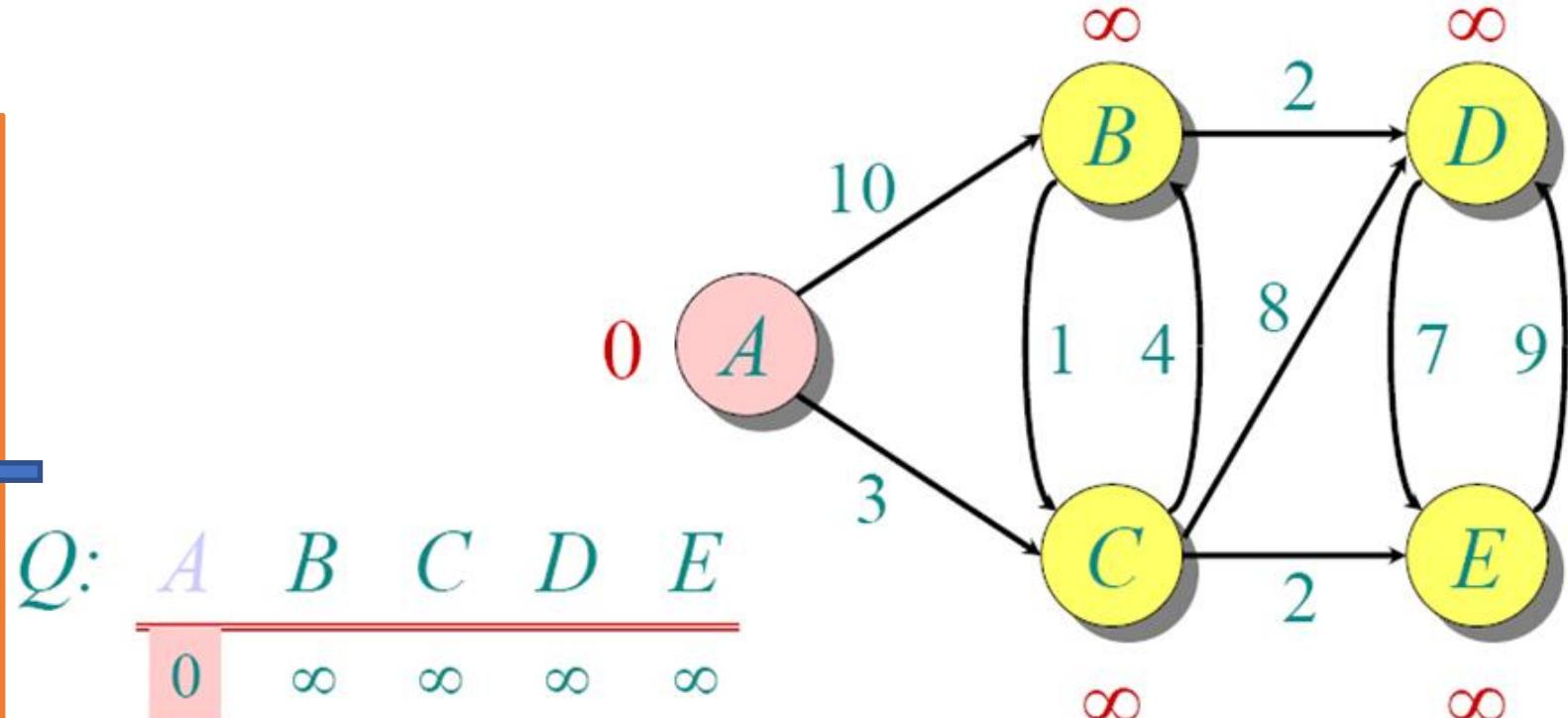
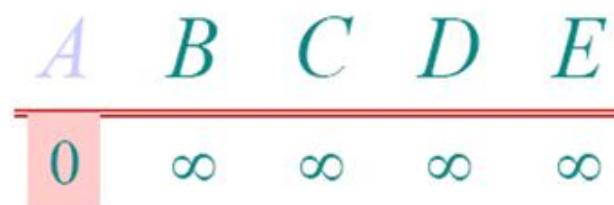


Dijkstra Animated Example

```
dist[s] ← 0
for all  $v \in V - \{s\}$ 
    do dist[v] ←  $\infty$ 
S ←  $\emptyset$ 
Q ← V
while Q ≠  $\emptyset$ 
do    $u \leftarrow \text{mindistance}(Q, \text{dist})$  ←
      S ← S ∪ {u}
      for all  $v \in \text{neighbors}[u]$ 
          do if dist[v] > dist[u] + w(u, v)
              then   d[v] ← d[u] + w(u, v)
```

return dist

$Q:$

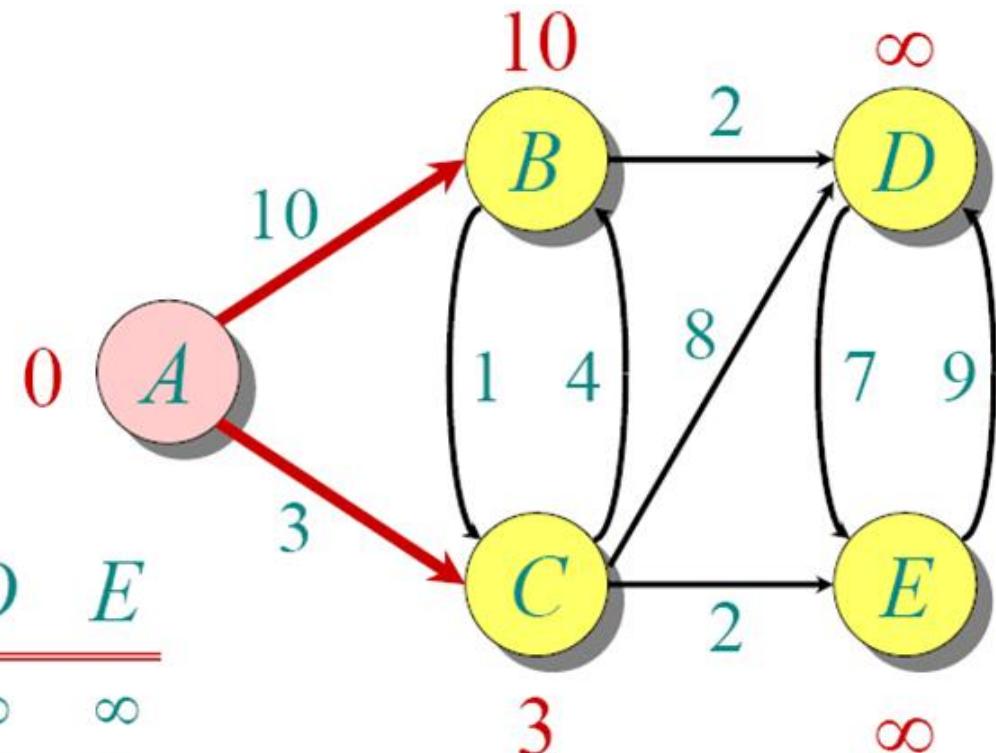




Dijkstra Animated Example

```
dist[s] ← 0
for all  $v \in V - \{s\}$ 
    do dist[v] ←  $\infty$ 
S ←  $\emptyset$ 
Q ← V
while Q ≠  $\emptyset$ 
do  $u \leftarrow \text{mindistance}(Q, \text{dist})$ 
    S ← S ∪ {u}
    for all  $v \in \text{neighbors}[u]$ 
        do if dist[v] > dist[u] + w(u, v)
            then dist[v] ← dist[u] + w(u, v)
return dist
```

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞



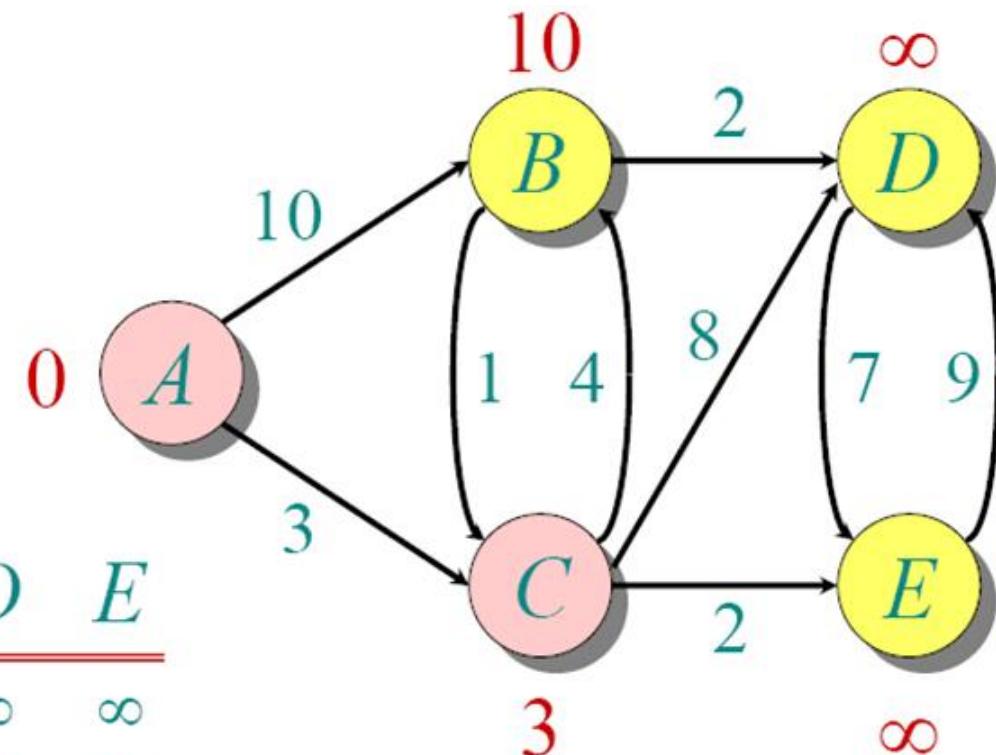
S: { A }



Dijkstra Animated Example

```
dist[s] ← 0
for all  $v \in V - \{s\}$ 
    do dist[v] ←  $\infty$ 
S ←  $\emptyset$ 
Q ← V
while Q ≠  $\emptyset$ 
do    $u \leftarrow \text{mindistance}(Q, \text{dist})$ 
      S ← S ∪ {u}
      for all  $v \in \text{neighbors}[u]$ 
          do if dist[v] > dist[u] + w(u, v)
              then    $d[v] \leftarrow d[u] + w(u, v)$ 
return dist
```

$Q:$	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	∞

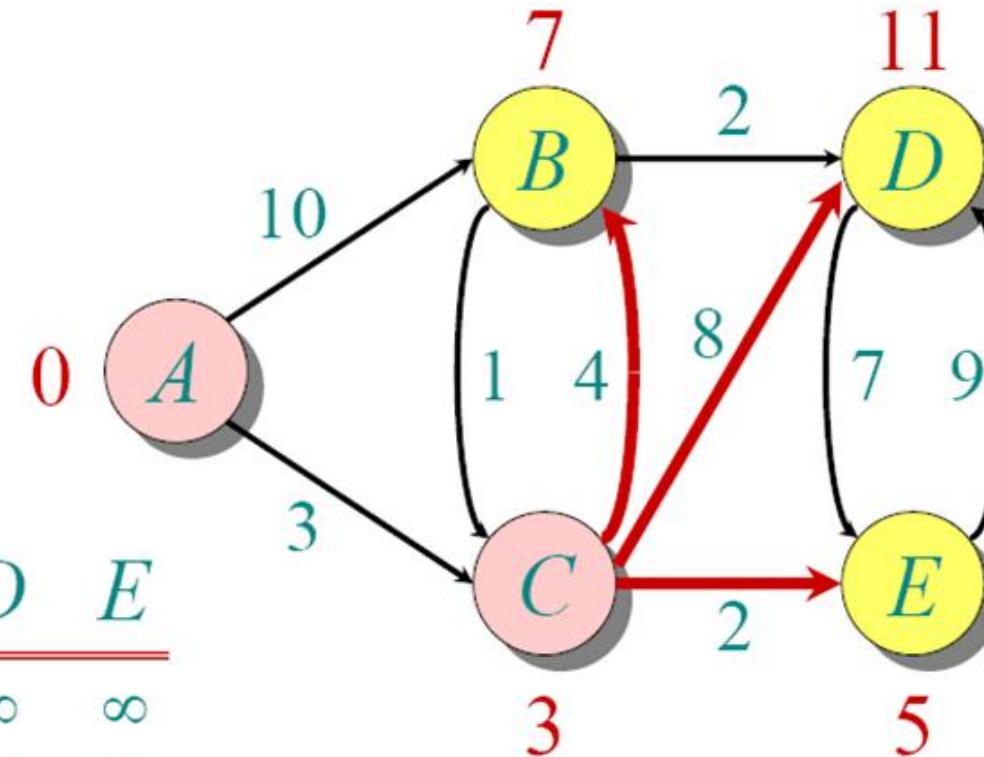


$S: \{ A, C \}$

When we put a node into S , we remove it from Q



Dijkstra Animated Example

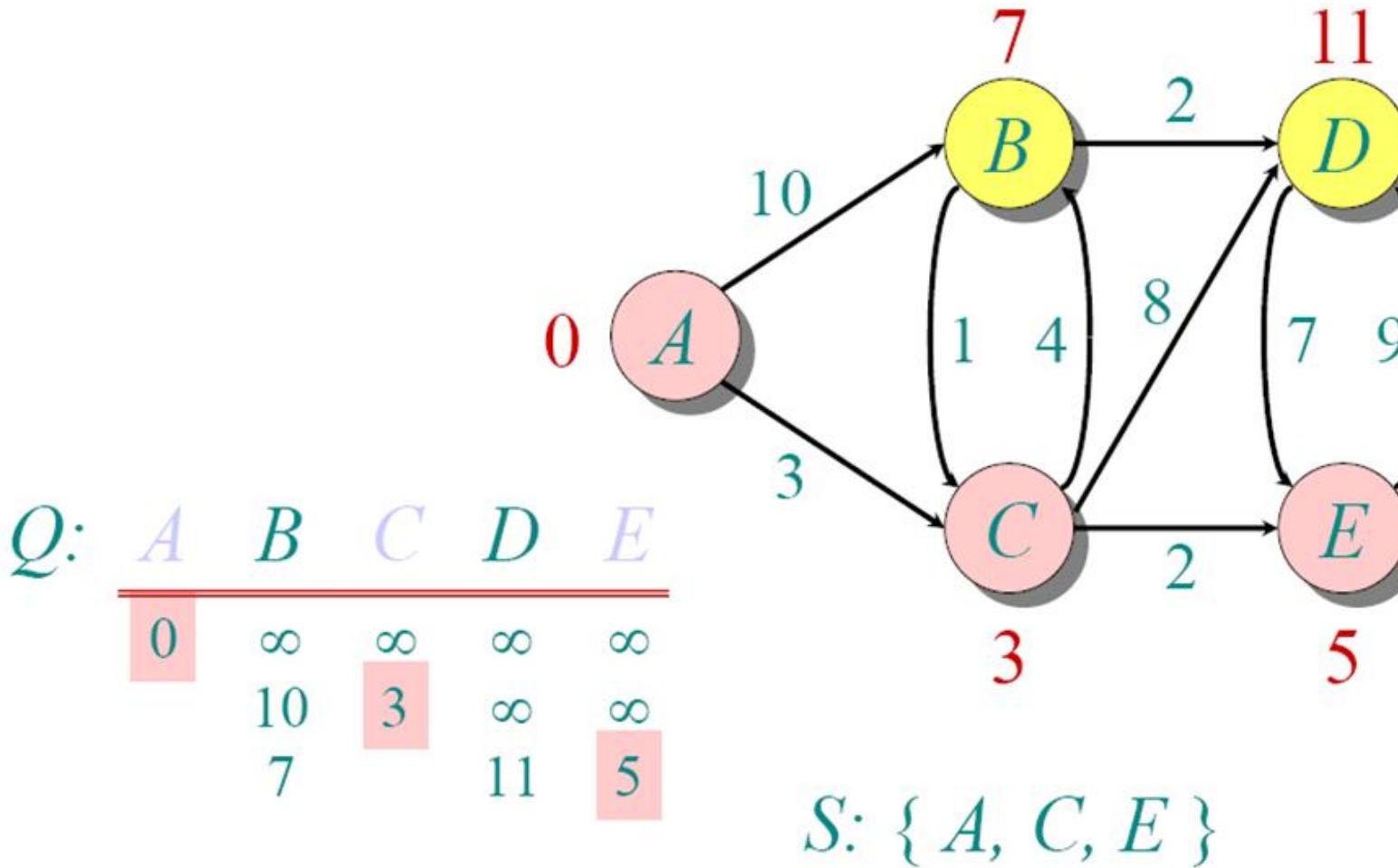


$Q:$	A	B	C	D	E
	0	∞	∞	∞	∞
	10	3	∞	∞	∞
	7		11	5	

$S: \{ A, C \}$

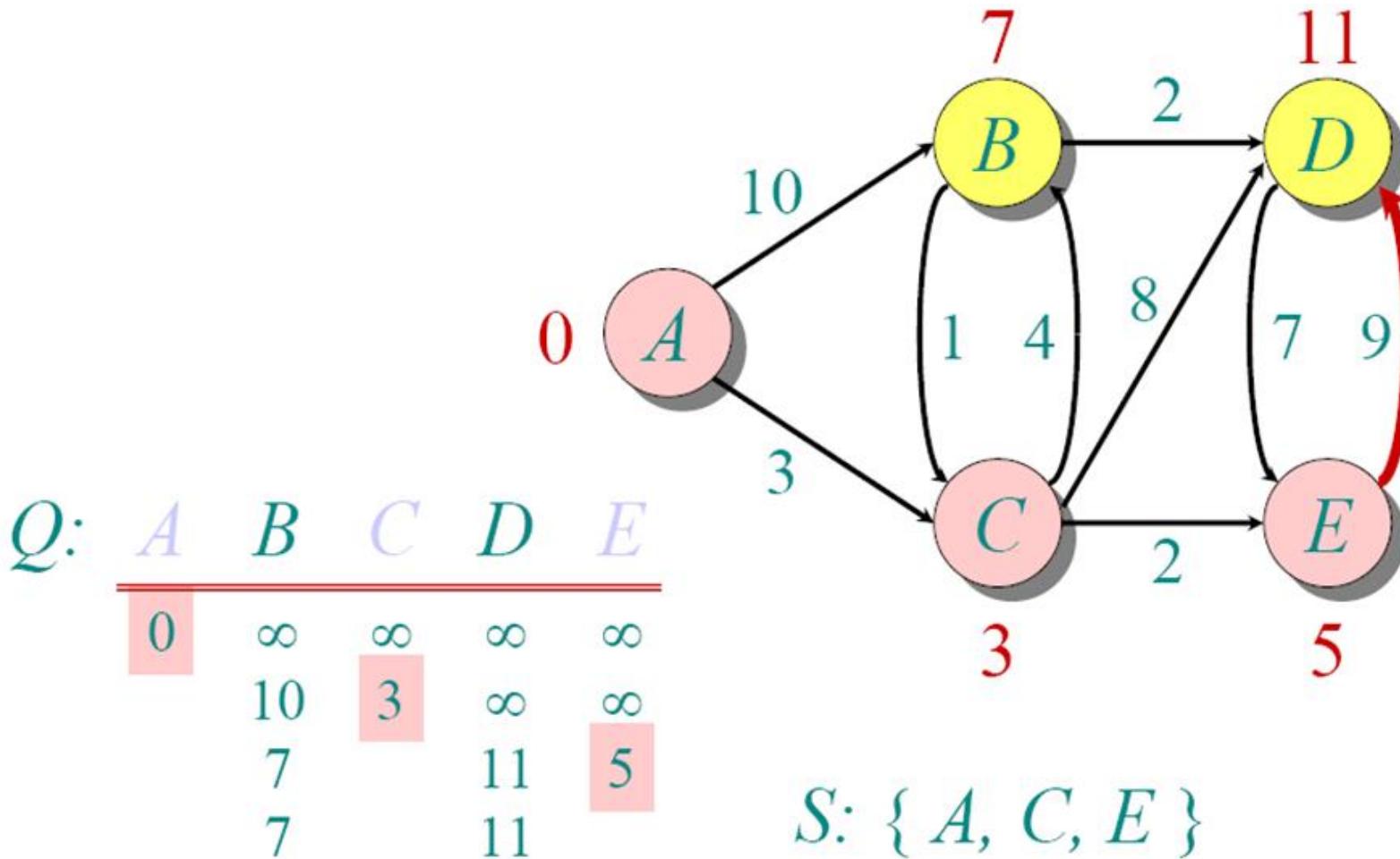


Dijkstra Animated Example



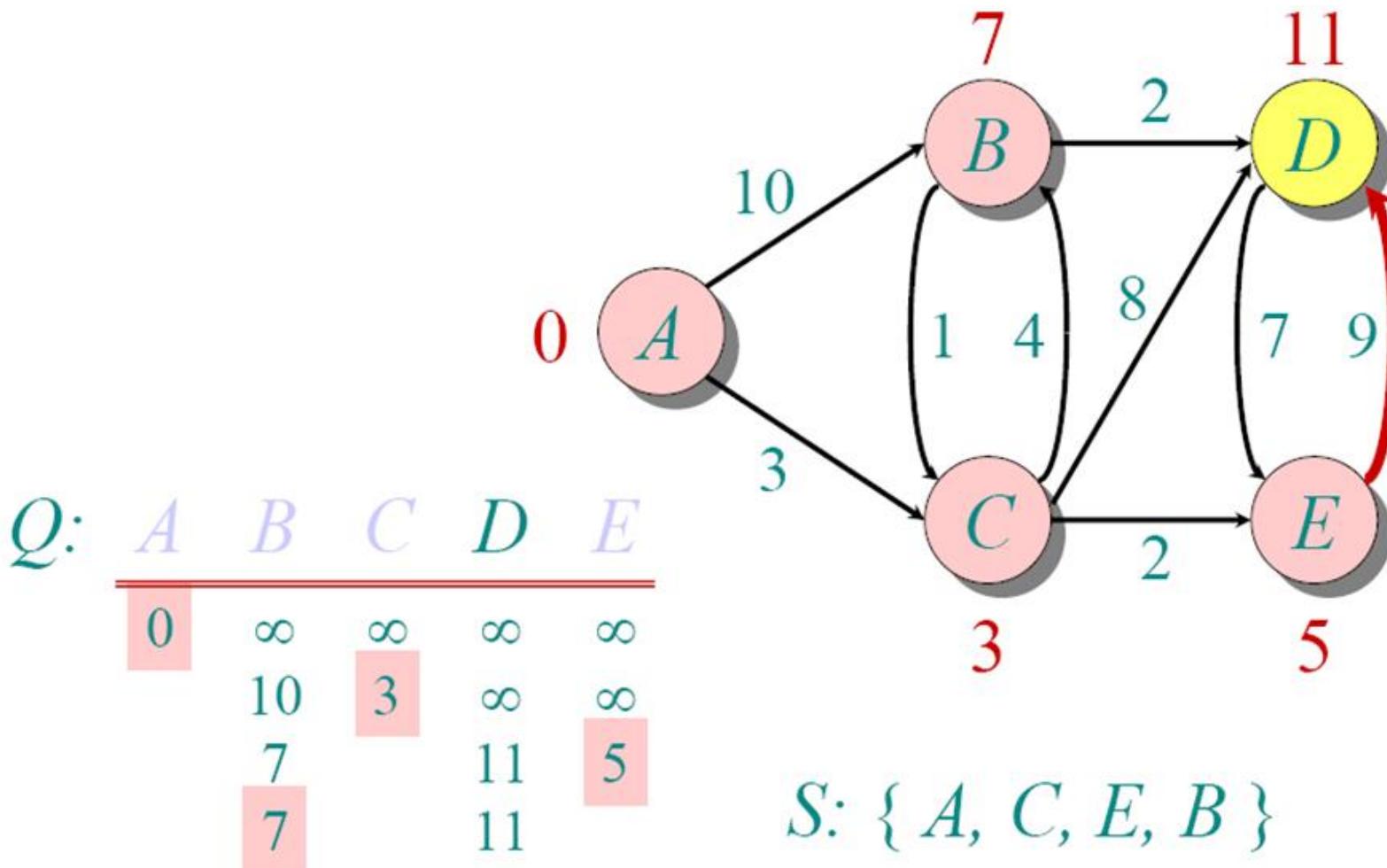


Dijkstra Animated Example



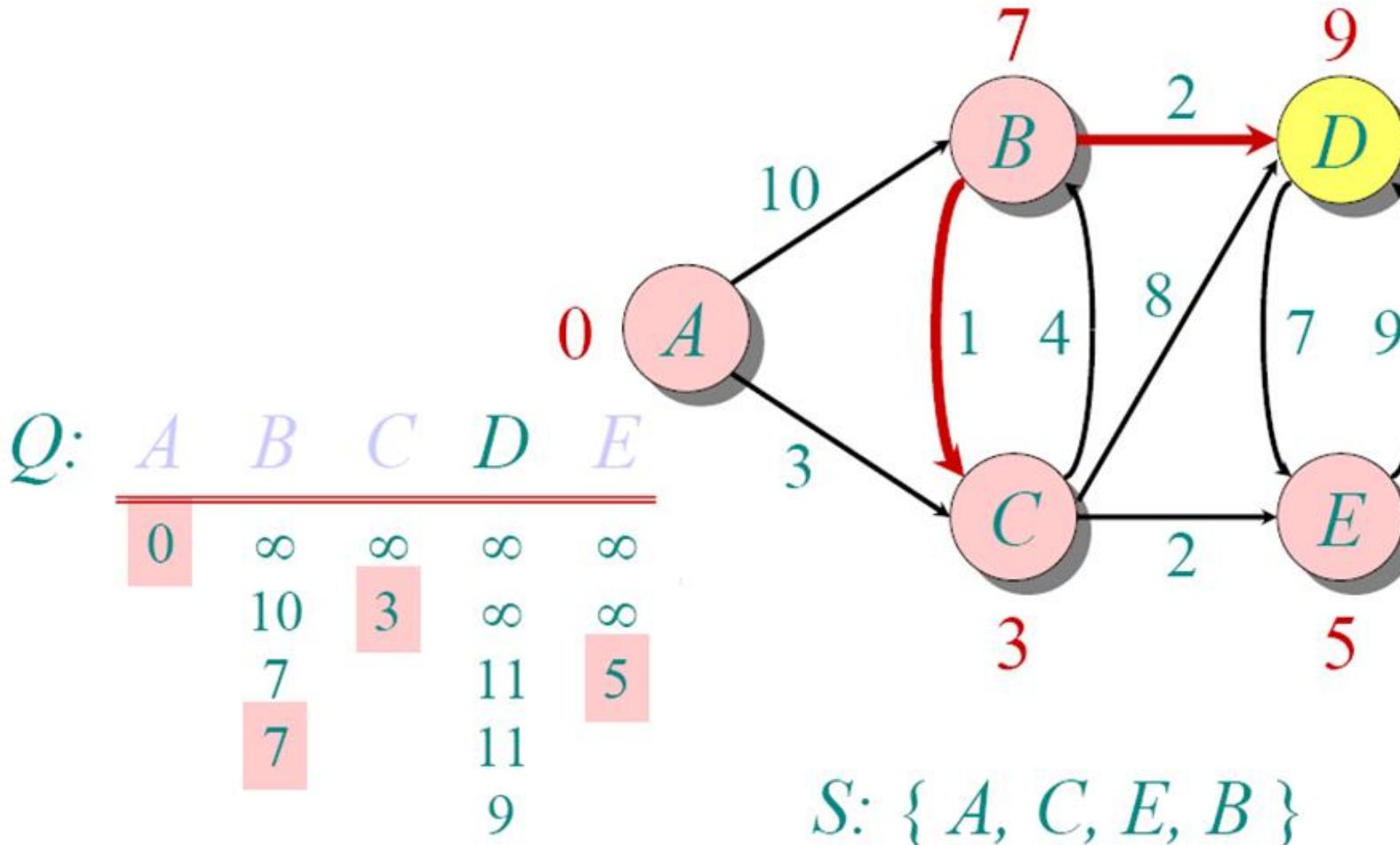


Dijkstra Animated Example



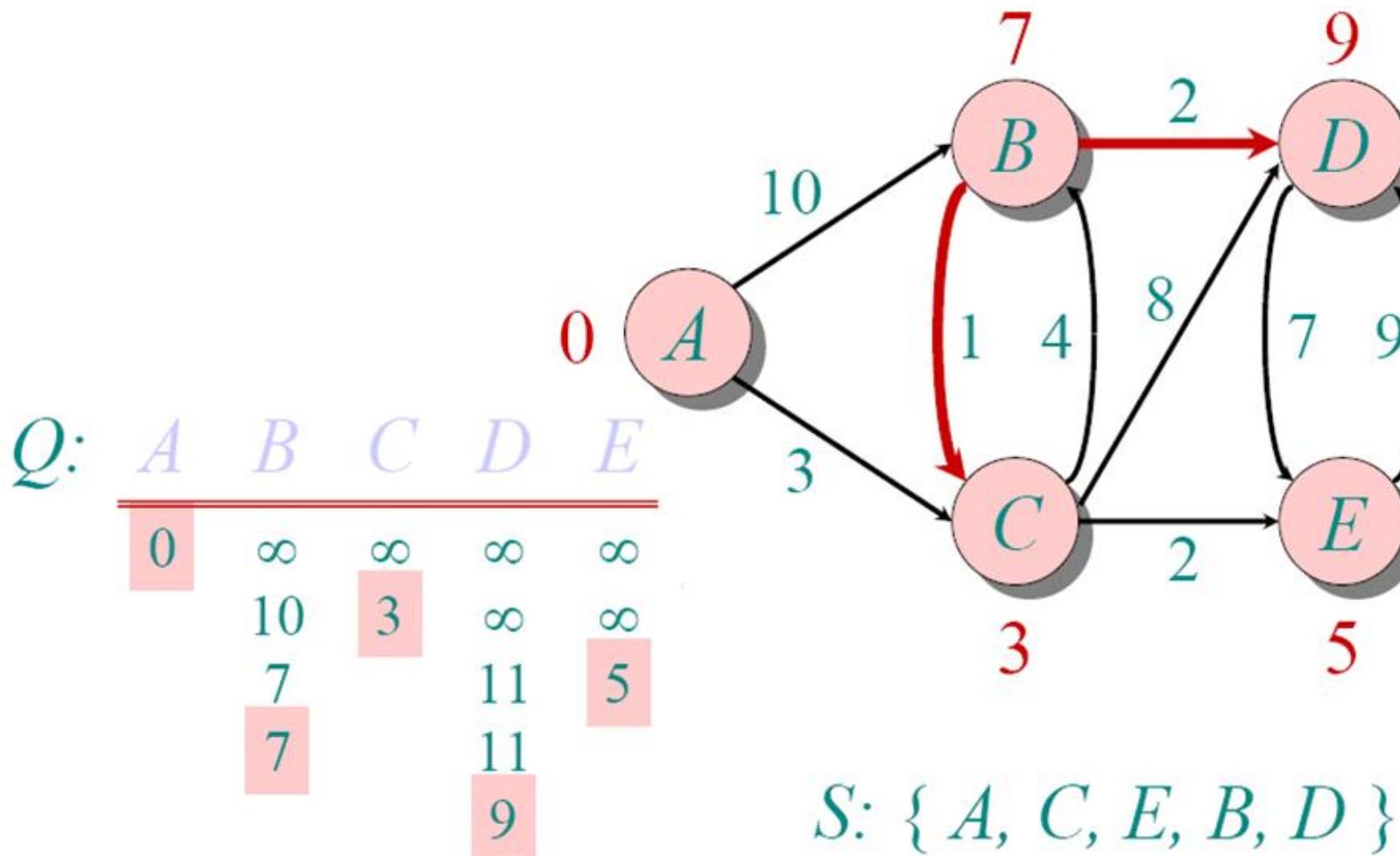


Dijkstra Animated Example





Dijkstra Animated Example





Shortest Path

- The algorithm returns the length of the shortest path to each node.
- If we store, for each node, its immediate predecessor on the minimum-cost path, we can reconstruct the optimal path
- In the example the optimal path from A to D is
 $\{A,C,B,D\}$



Implementations and Running Times

The simplest implementation is to store vertices in an array or linked list. This yields a running time of

$$O(|V|^2 + |E|)$$

For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This yields a running time of

$$O((|E|+|V|) \log |V|)$$



Dijkstra's Algorithm - Why It Works

- As with all greedy algorithms, we must ensure correctness (e.g., that it *always* returns the correct solution given valid input).
- A full proof is beyond the scope of this lecture; please refer to the literature.



A*



Motivations

- The Dijkstra algorithm has advantages
 - It is optimal
 - It is polynomial
- ...but it also has a drawback:
 - It computes the best path from to all nodes (and they can be many)
 - What if we are simply interested in a Source->Goal travel?
 - What if we have a heuristic evaluation of how much it takes to go from any node to the destination?
- For these cases we have got a specialised algorithm (called A*)



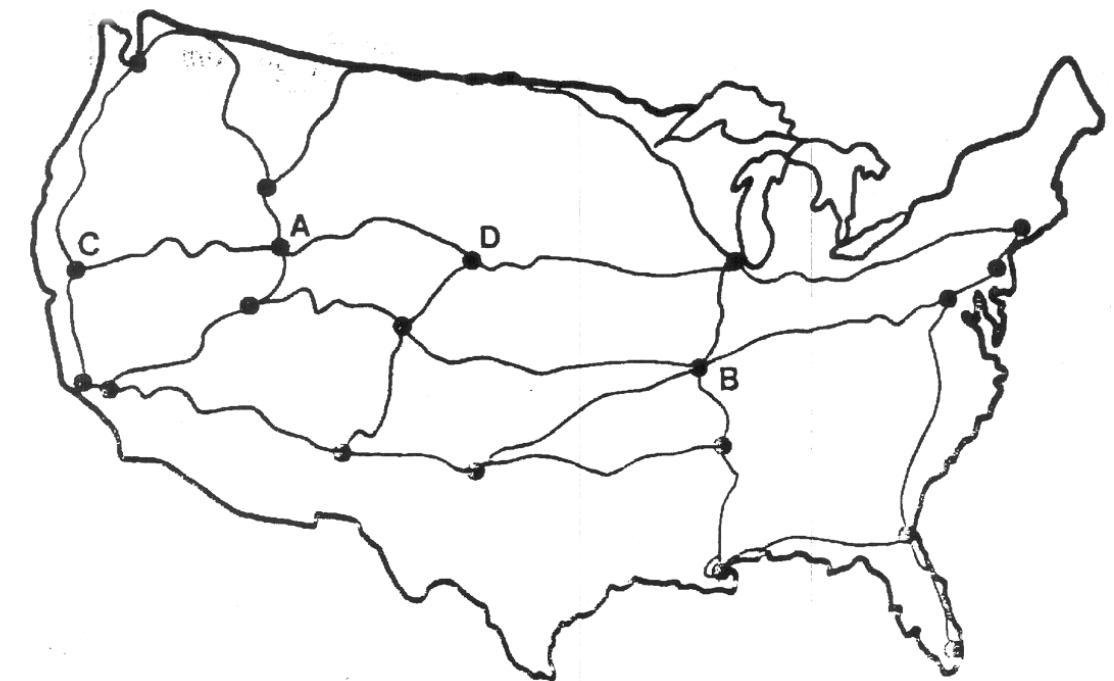
Dijkstra vs. A*

- Dijkstra: compute the optimal solution
- Diskstra: explores a larger search space than A*
- A*: simple
- A*: fast
- A*: “good” result It can deviate from the optimal path if there is not a good heuristic
- A*: use heuristic estimate to eliminate many high cost paths, speeding up process computation of satisfactory “shortest” paths



Example

- Consider the following roadmap
- Find the shortest path from A to B
- Heuristic function:
 - $h(i)$: air distance from any node to B
- We can estimate the distance from A to B as:
 - $d(A,D)+h(D)$



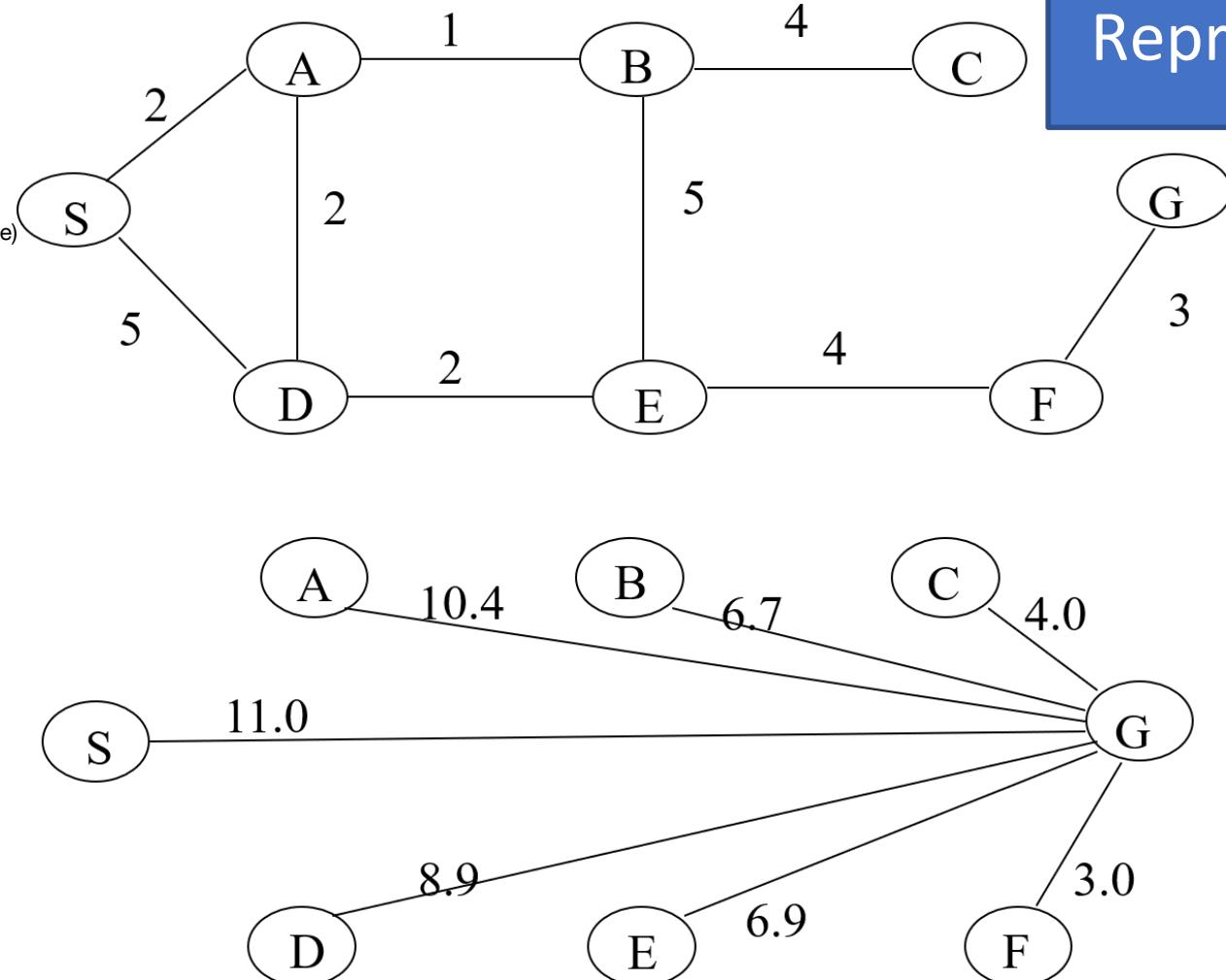


Generally speaking

Graph Representation

- If we have a source node S and a destination node G
(Before we had only a source node)
we are in this condition to estimate the heuristic distance

Heuristic Distance





Algorithm A*

Input: Graph G

Output: Minimum path list

Closed = 0;

Open = G.start();

While (Open != 0) {

 Let n be the node in Open with minimal $f(n)=g(n)+h(n)$;

If (n == G.goal) **return**;

ForEach (v in G.successors(n)) {

 curr_cost = g(n)+w(n,v);

if (v in Open) {

if (g(v) < curr_cost) **continue**; //end of foreach

 } **else if** (v in Closed) {

if (g(v) < curr_cost) **continue**; //end of foreach

 Closed = Closed \ v;

 Open = Open U v;

 } **else** {

 Open = Open U v;

 }

 g(v) = curr_cost;

 v.prev = n

 }

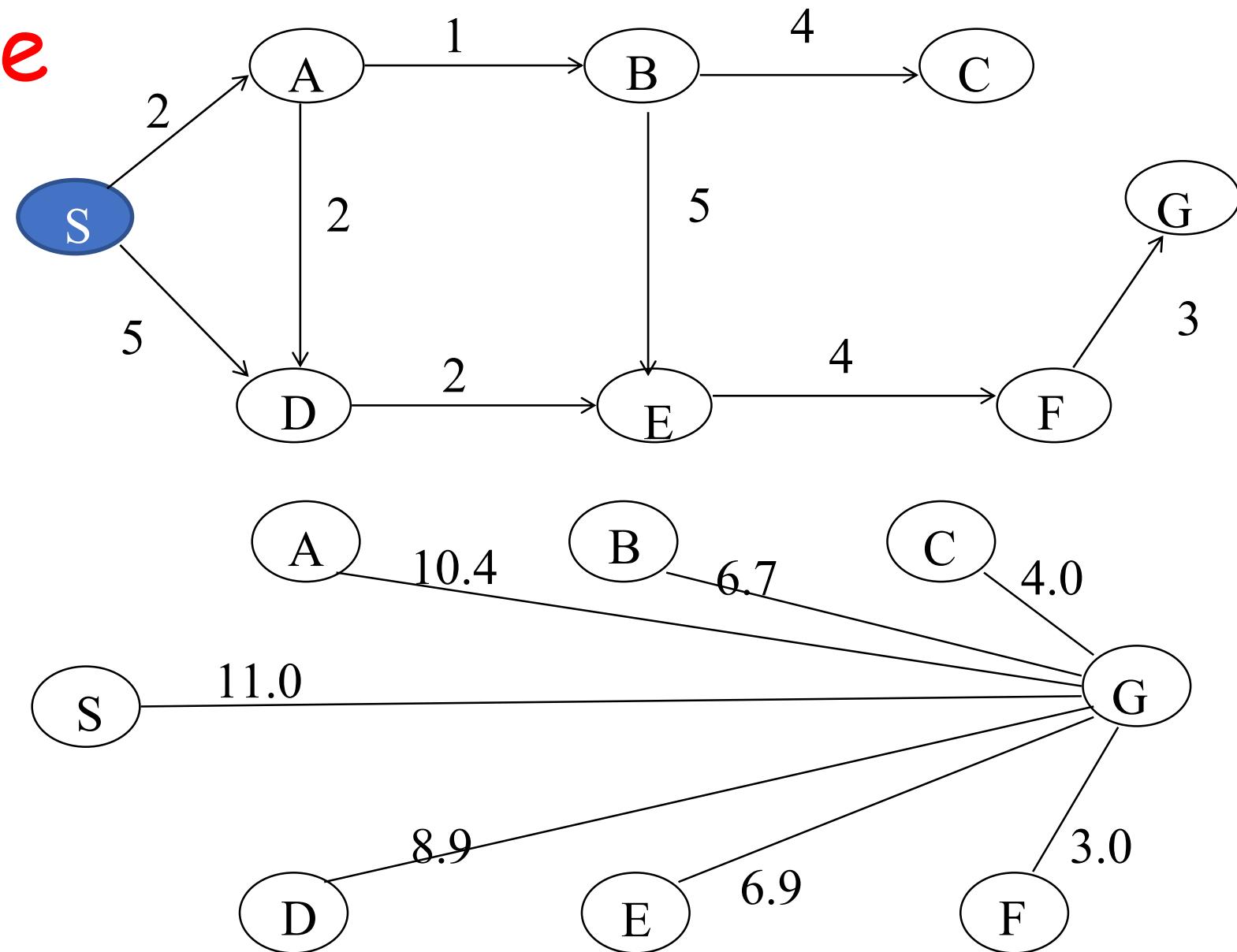
 Closed = Closed U n;

}



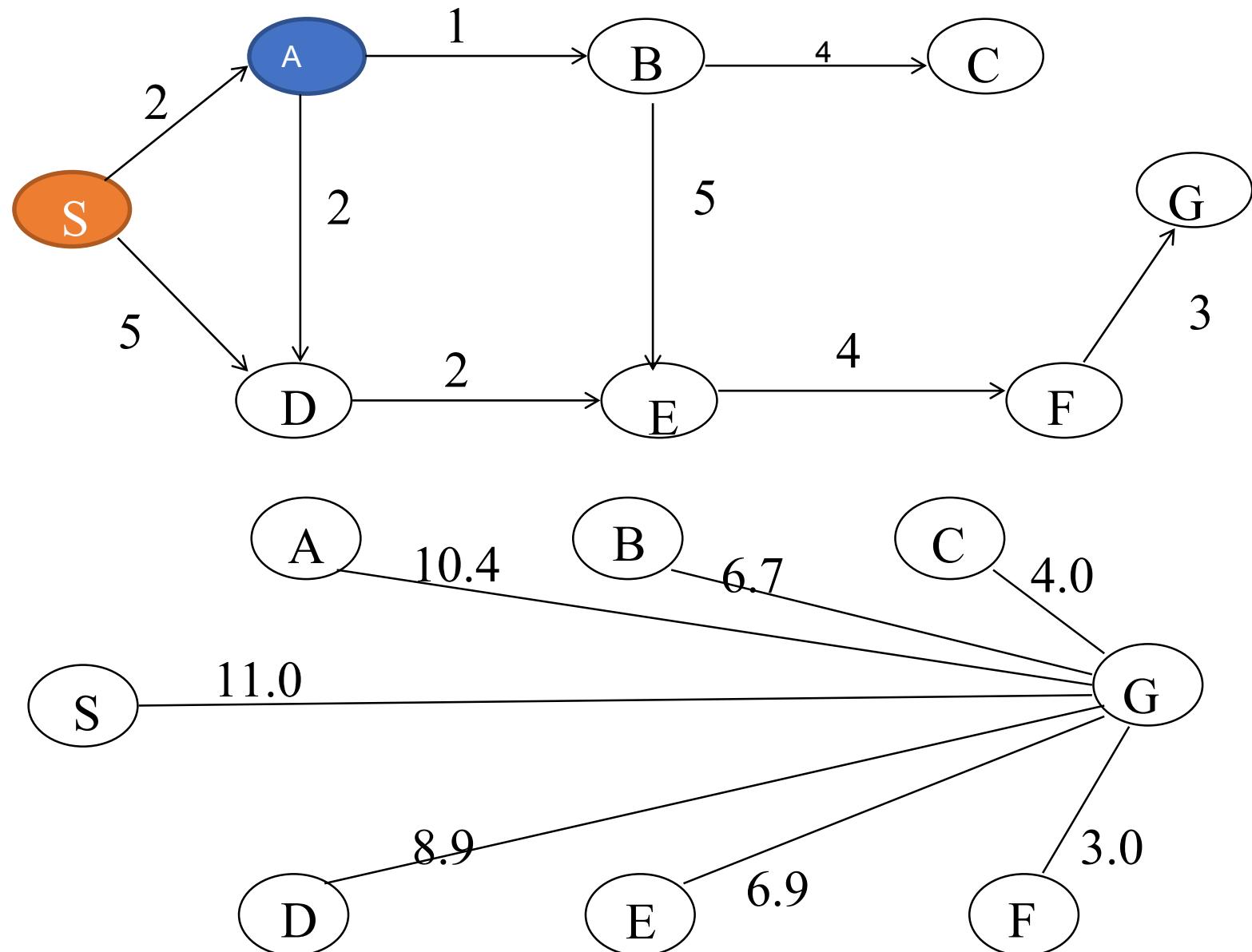
Example

n	v
Open	Closed
S ($g = 0, f = 11$)	



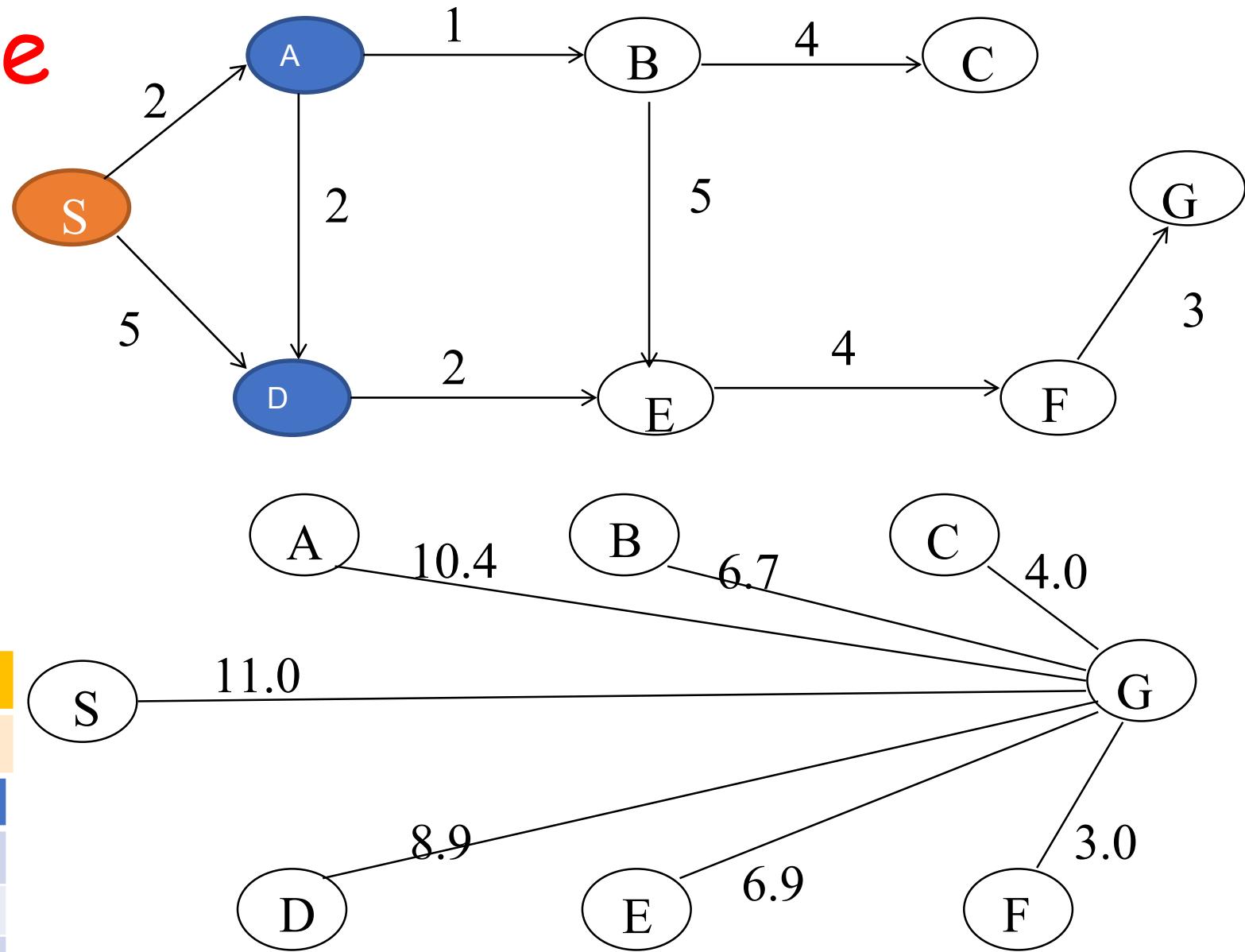


Example



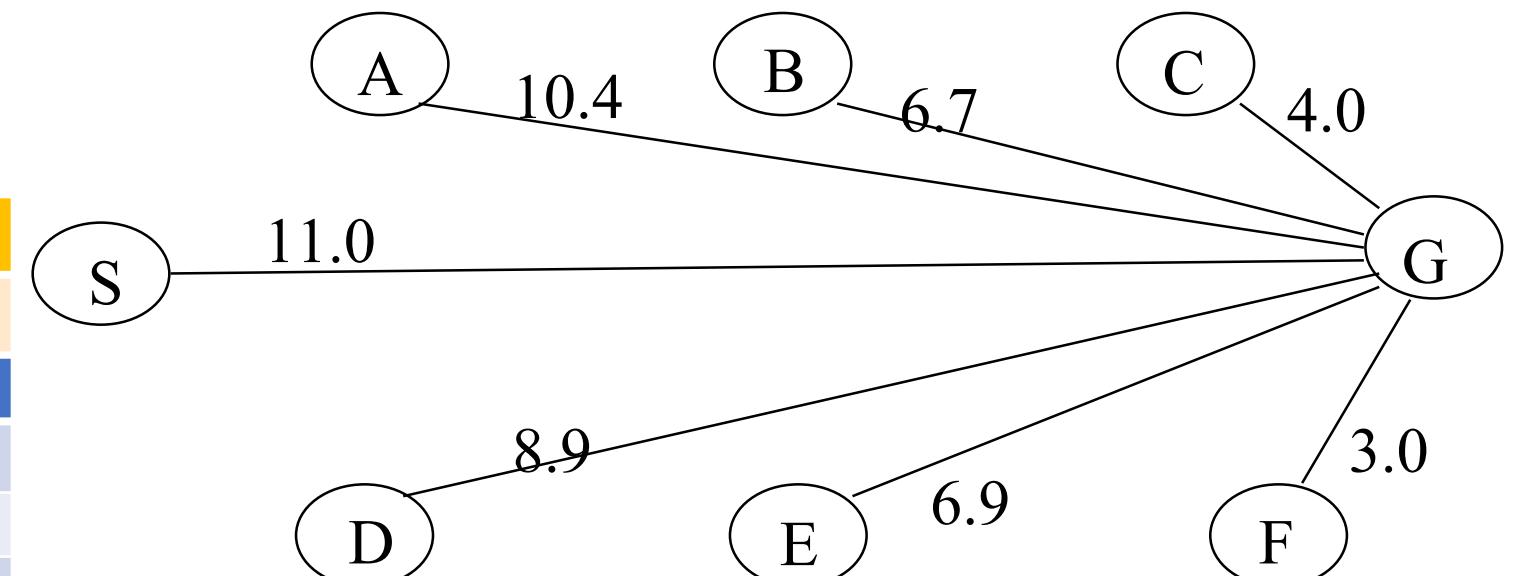
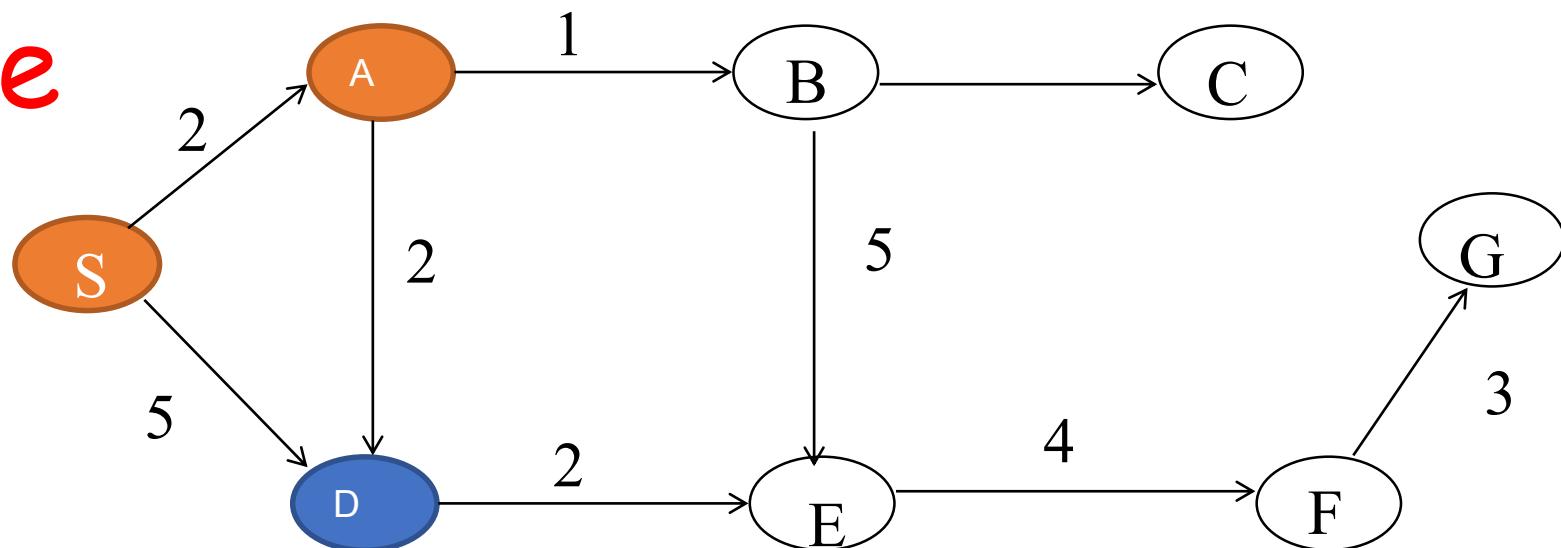


Example





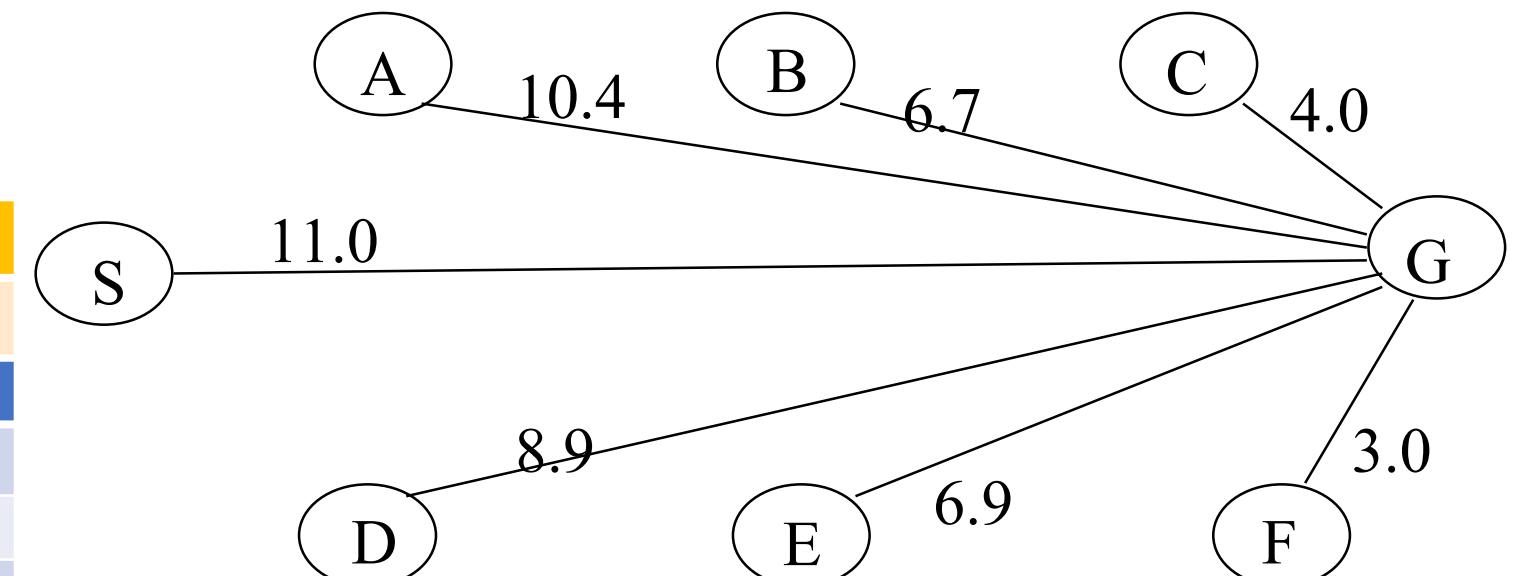
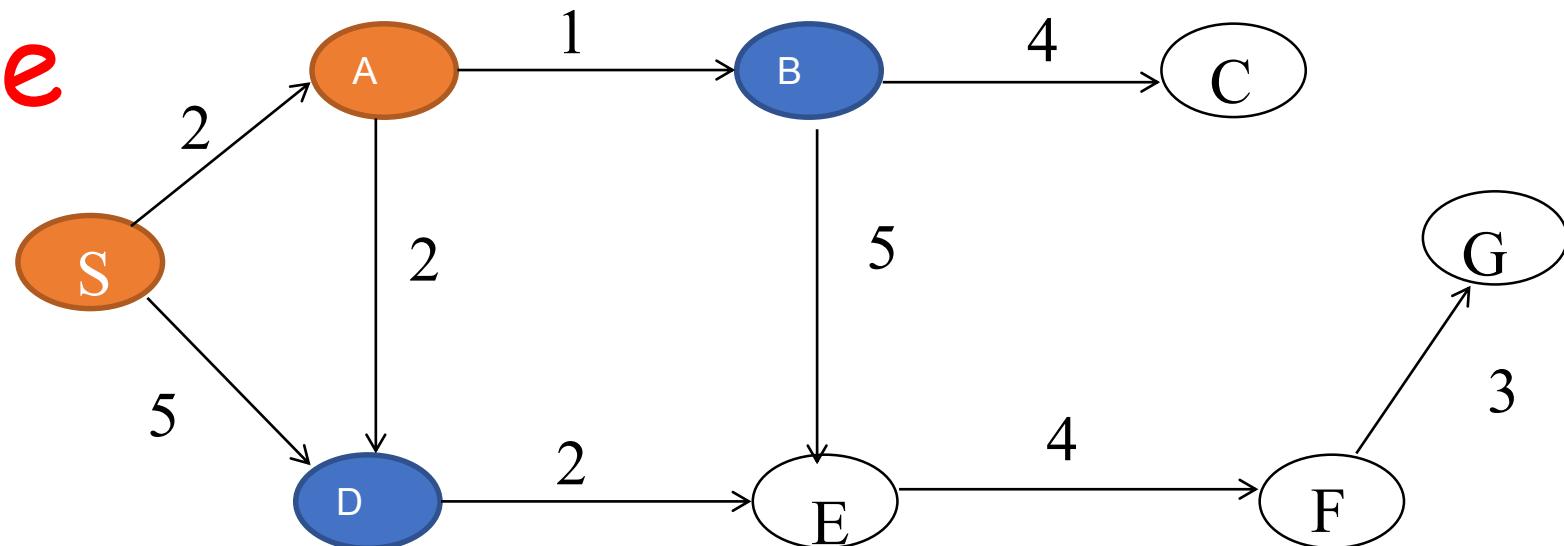
Example



n	v
A	D
Open	Closed
D (g = 5, f = 12.9, prev=A)	S (g = 0, f = 11, prev= Ø)
	A (g=2, f =12.4, prev=S)



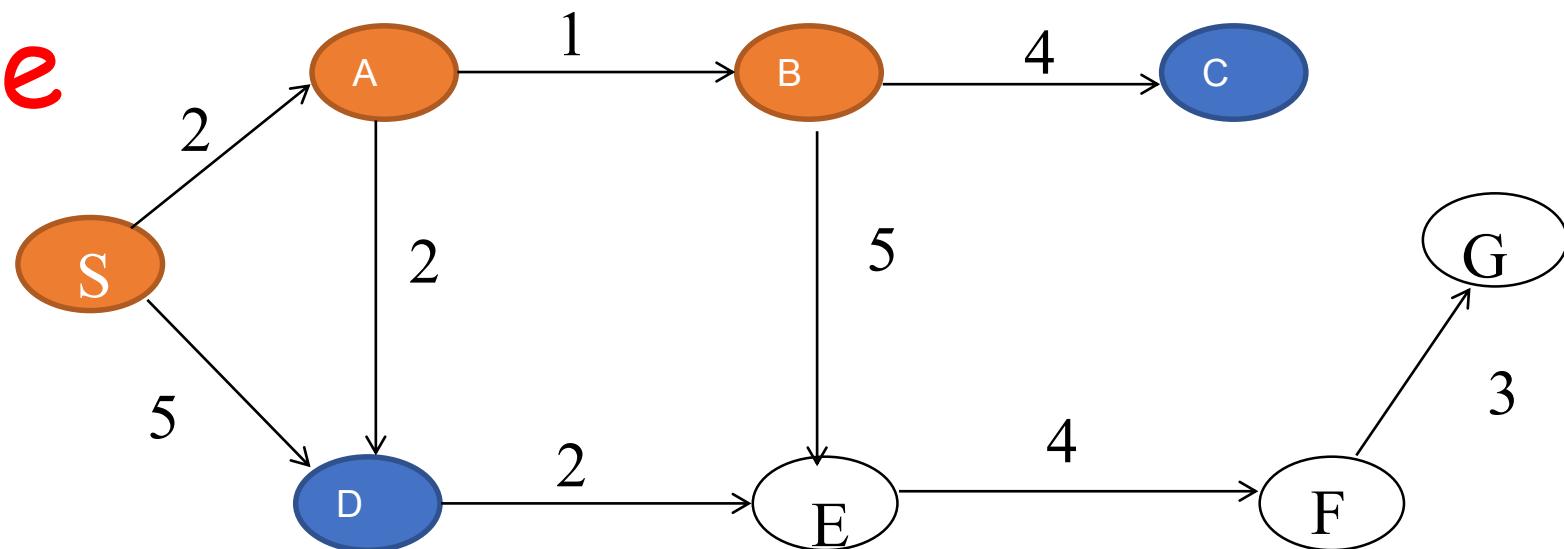
Example



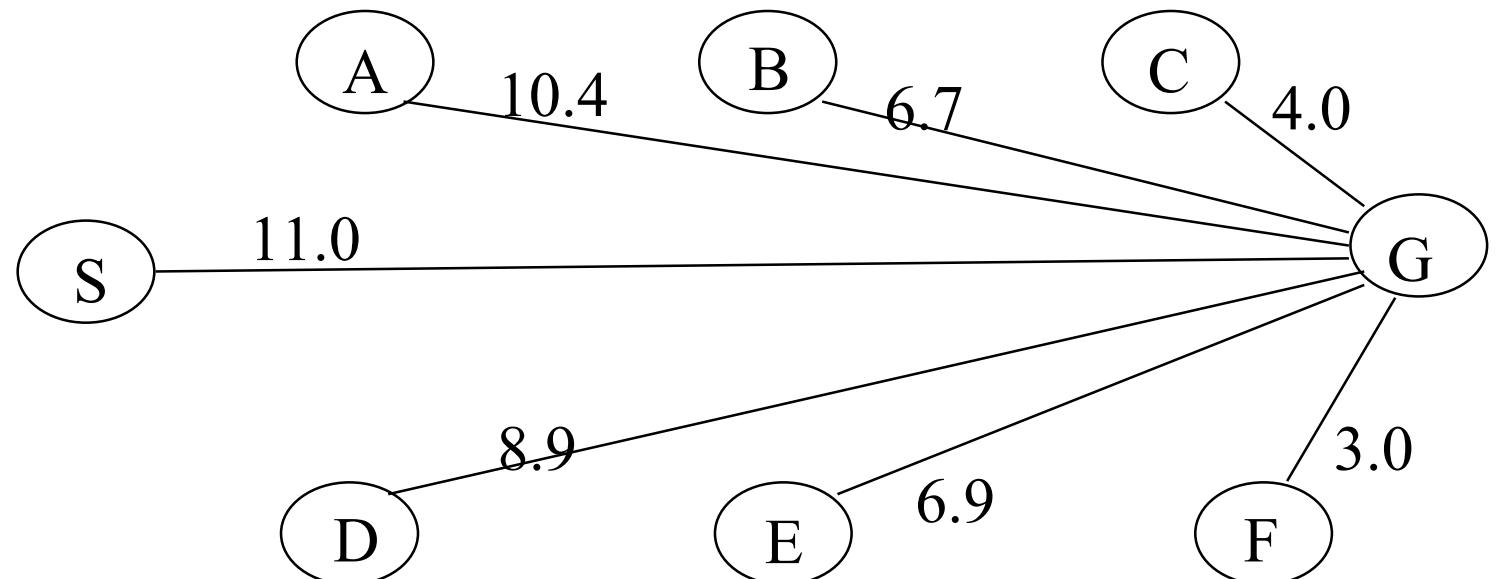
n	v
A	B
Open	Closed
B (g = 3, f = 9.7, prev=A)	S (g = 0, f = 11, prev= Ø)
D (g = 5, f = 12.9, prev=A)	A (g=2, f=12.4, prev=S)



Example

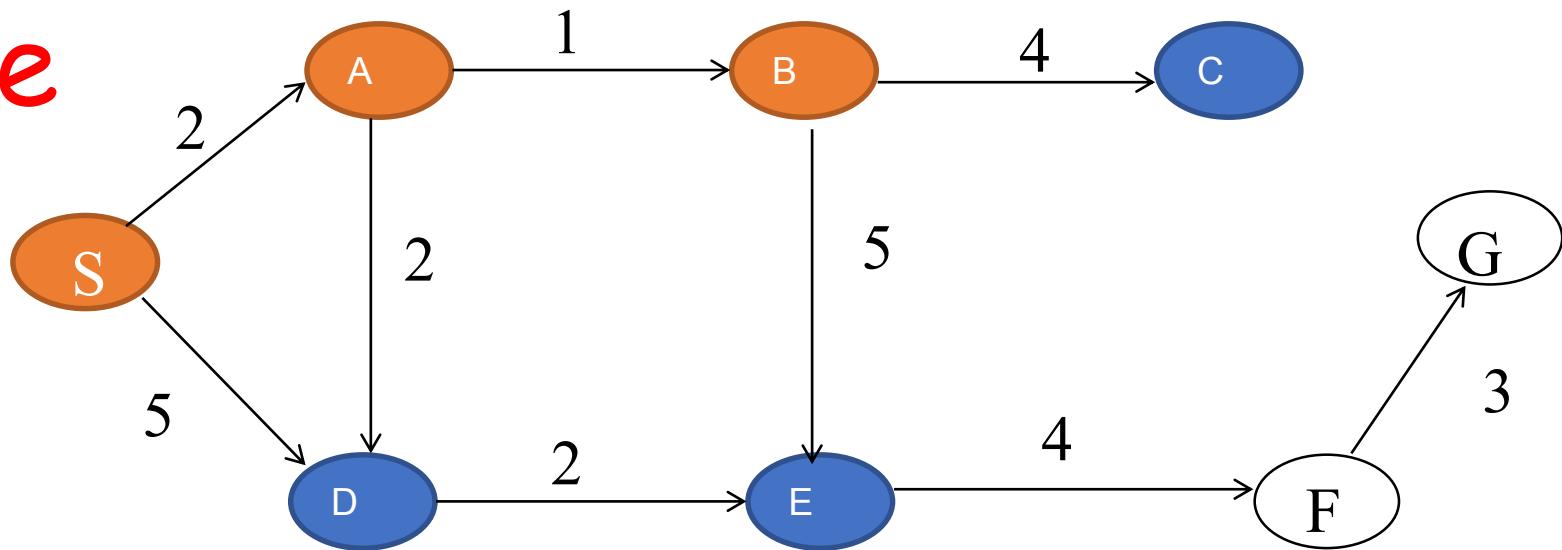


n	v
B	C
Open	Closed
C (g = 7, f = 11, prev=B)	S (g = 0, f = 11, prev= Ø)
D (g = 5, f = 12.9, prev=A)	A (g=2, f =12.4, prev=S)
	B (g = 3, f = 9.7, prev=A)

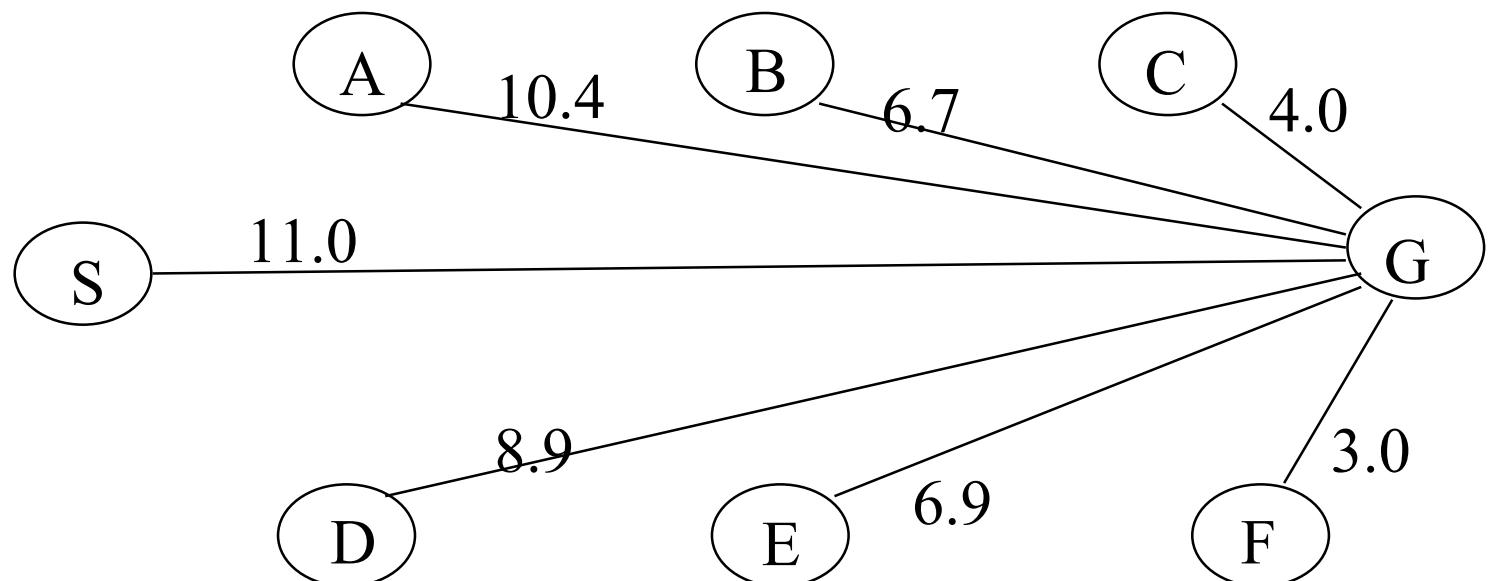




Example

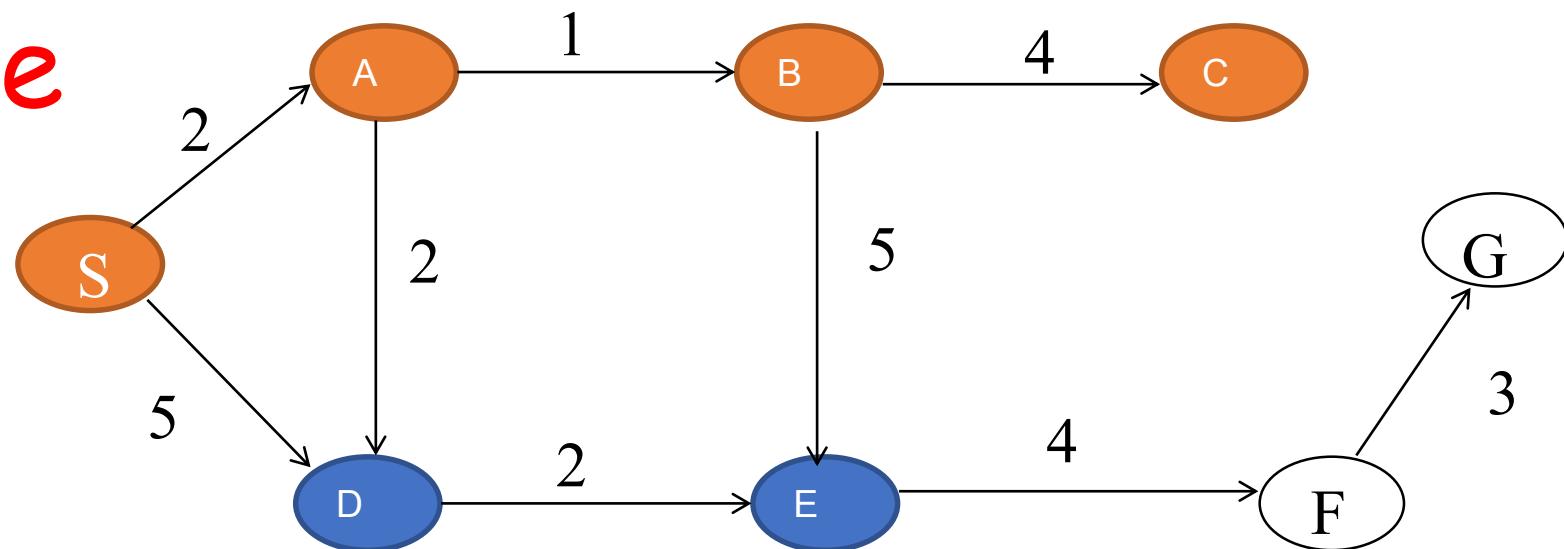


n	v
B	E
Open	Closed
C ($g = 7$, $f = 11$, prev=B)	S ($g = 0$, $f = 11$, prev=∅)
D ($g = 5$, $f = 12.9$, prev=A)	A ($g=2$, $f = 12.4$, prev=S)
E($g=8$, $f = 14.9$, prev=B)	B ($g = 3$, $f = 9.7$, prev=A)

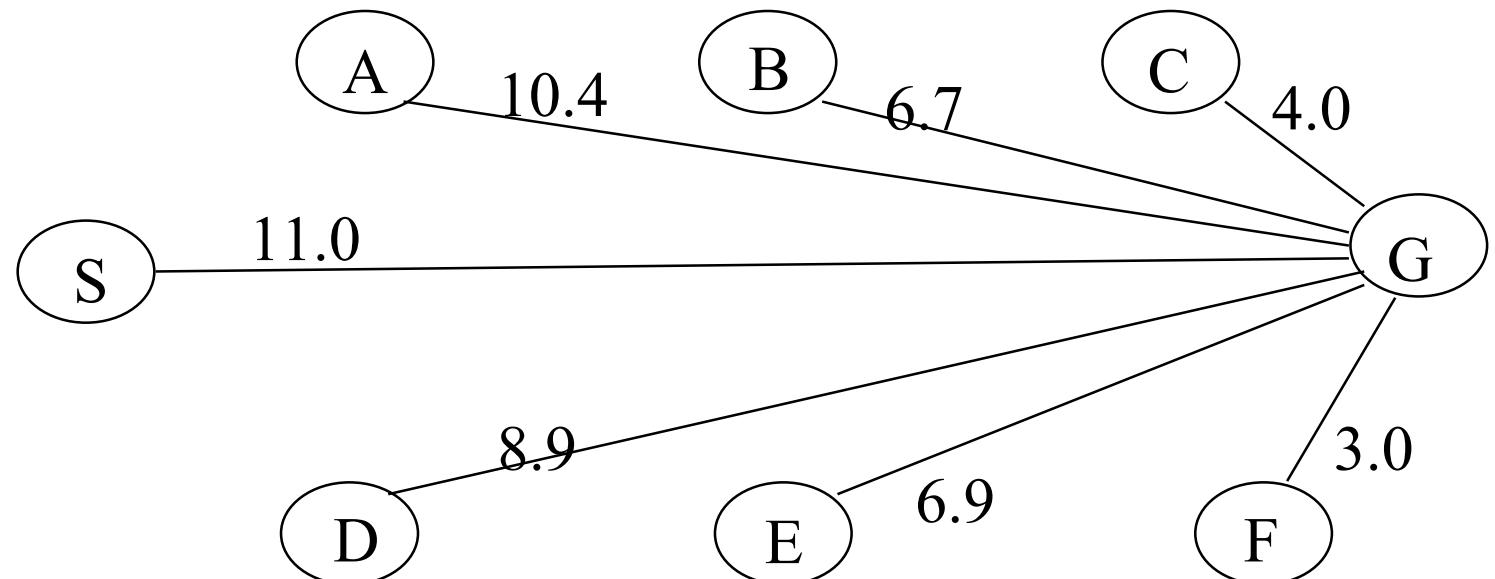




Example

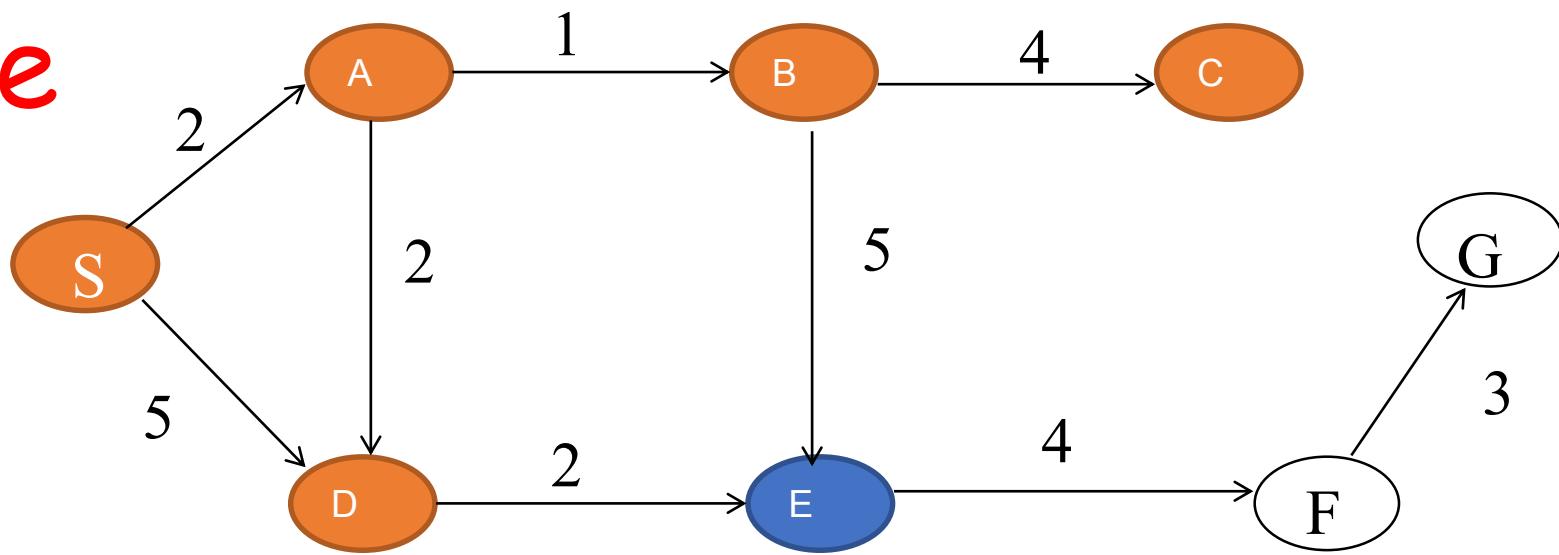


N	V
C	=
Open	Closed
	S ($g = 0, f = 11, \text{prev} = \emptyset$)
D ($g = 5, f = 12.9$, prev=A)	A ($g=2, f = 12.4$, prev=S)
E($g=8, f = 14.9$, prev=B)	B ($g = 3, f = 9.7$, prev=A)
	C ($g = 7, f = 11$, prev=B)

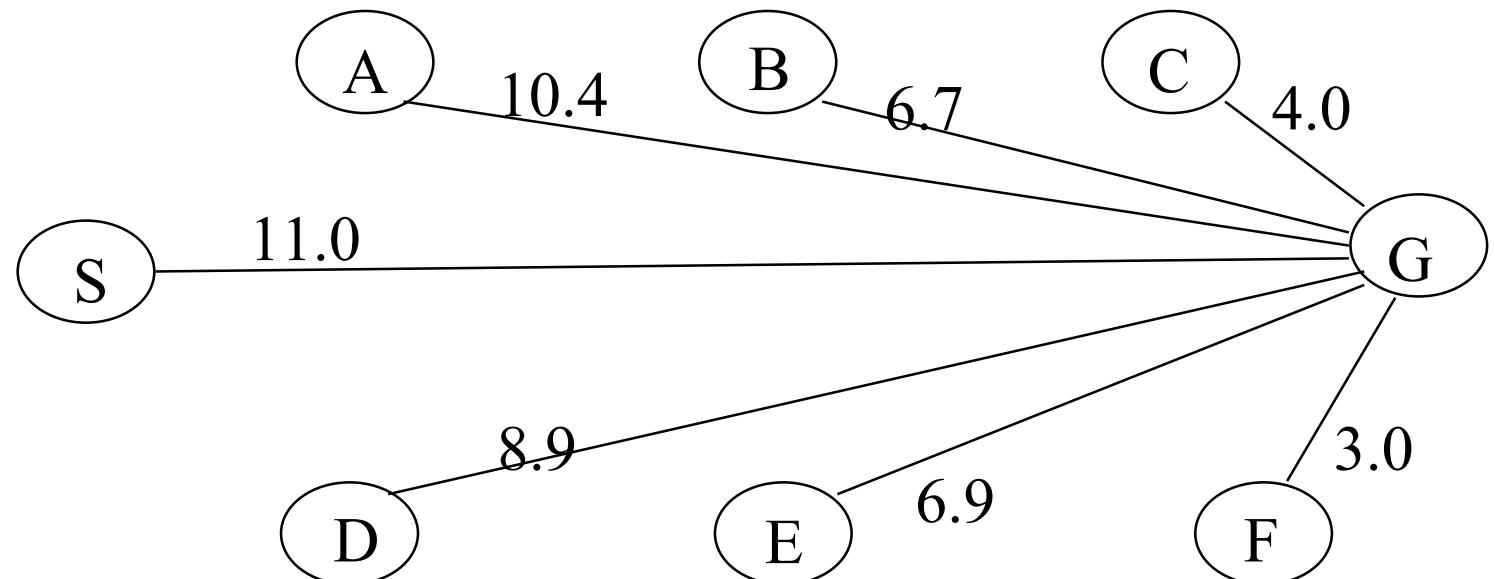




Example

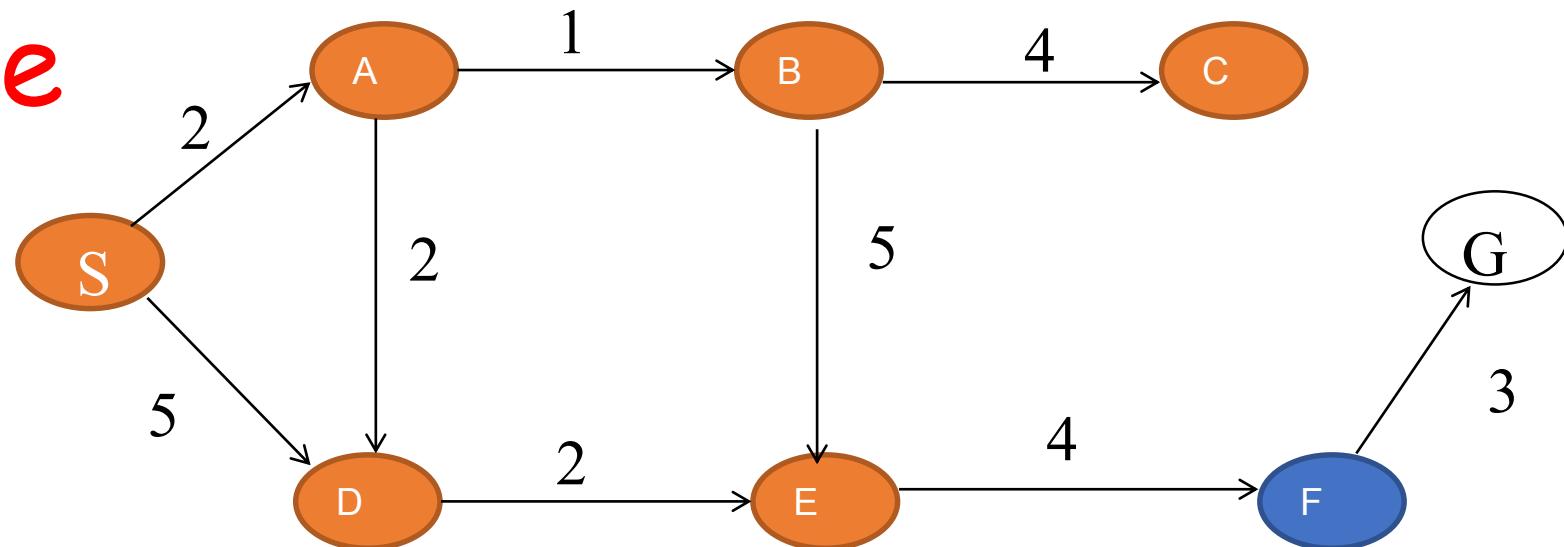


N	V
D	E
Open	Closed
	S ($g = 0, f = 11, \text{prev} = \emptyset$)
	A ($g = 2, f = 12.4, \text{prev} = S$)
E($g = 8, f = 13.9, \text{prev} = D$)	B ($g = 3, f = 9.7, \text{prev} = A$)
	C ($g = 7, f = 11, \text{prev} = B$)
	D ($g = 5, f = 12.9, \text{prev} = A$)

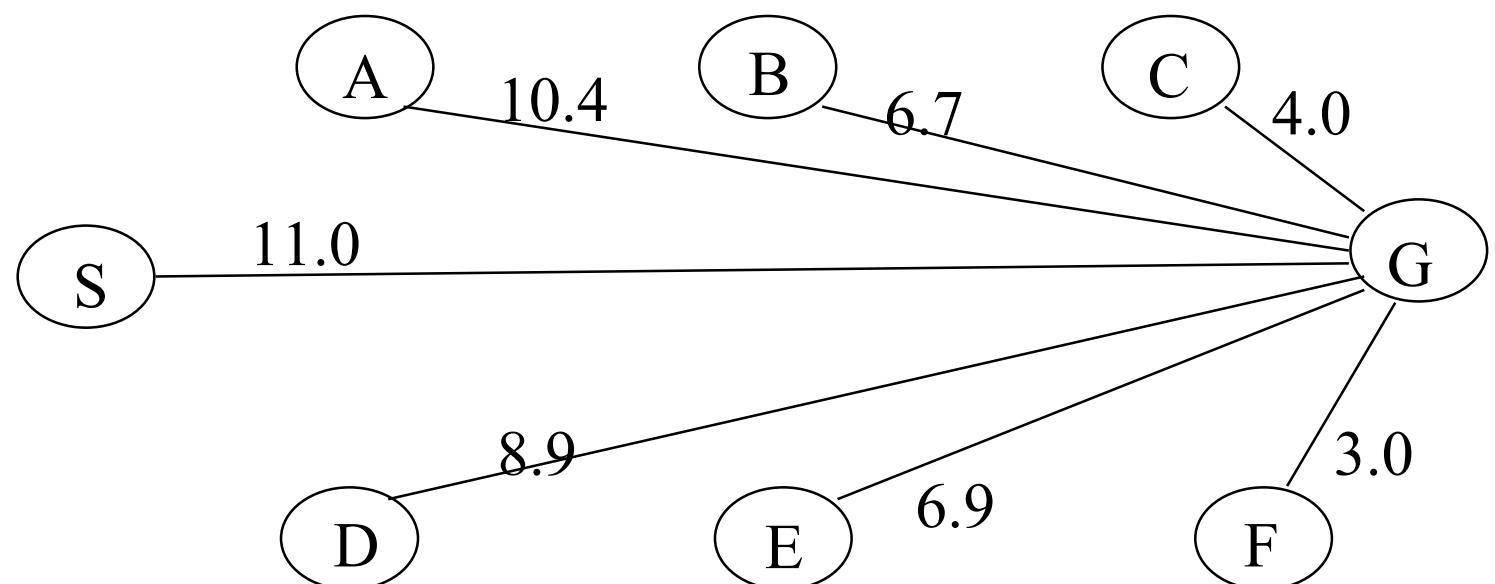




Example

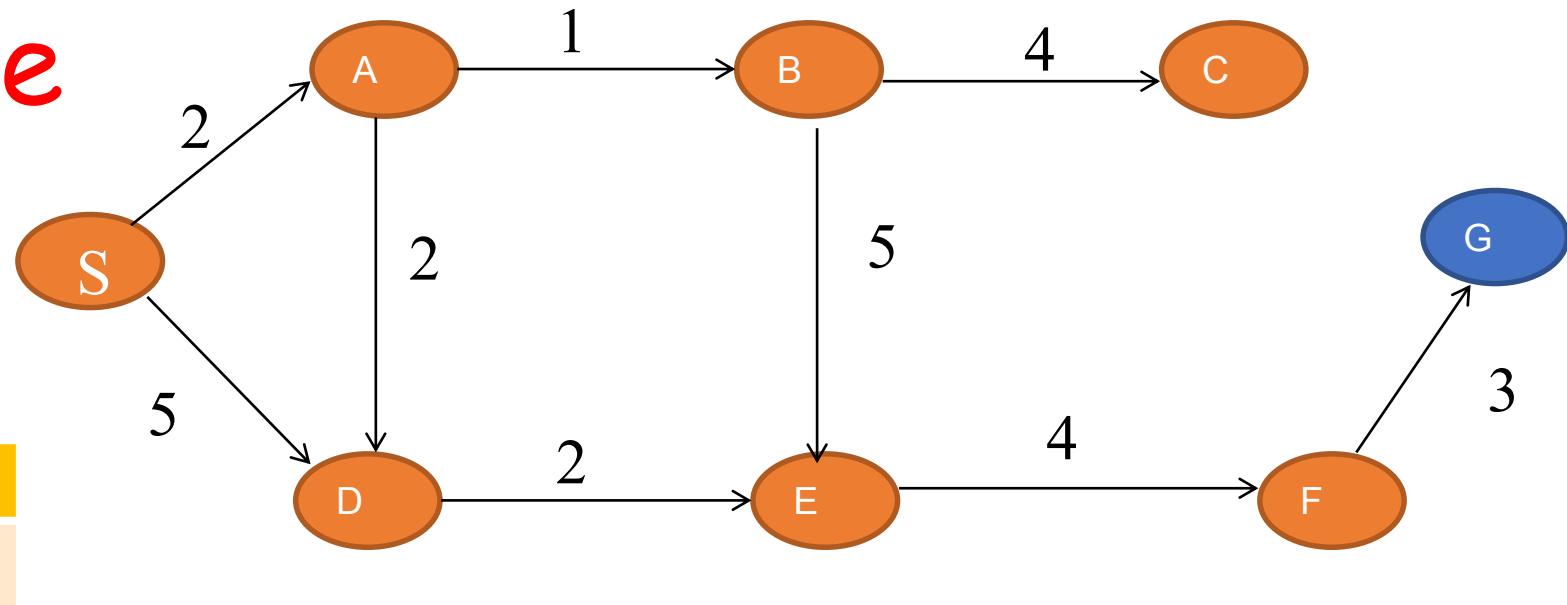


N	V
E	F
Open	Closed
F(g=11,f=14,prev=E)	S (g = 0, f = 11, prev= \emptyset)
	A (g=2, f = 12.4, prev=S)
	B (g = 3, f = 9.7, prev=A)
	C (g = 7, f = 11, prev=B)
	D (g = 5, f = 12.9, prev=A)
	E(g=8, f = 13.9, prev=D)



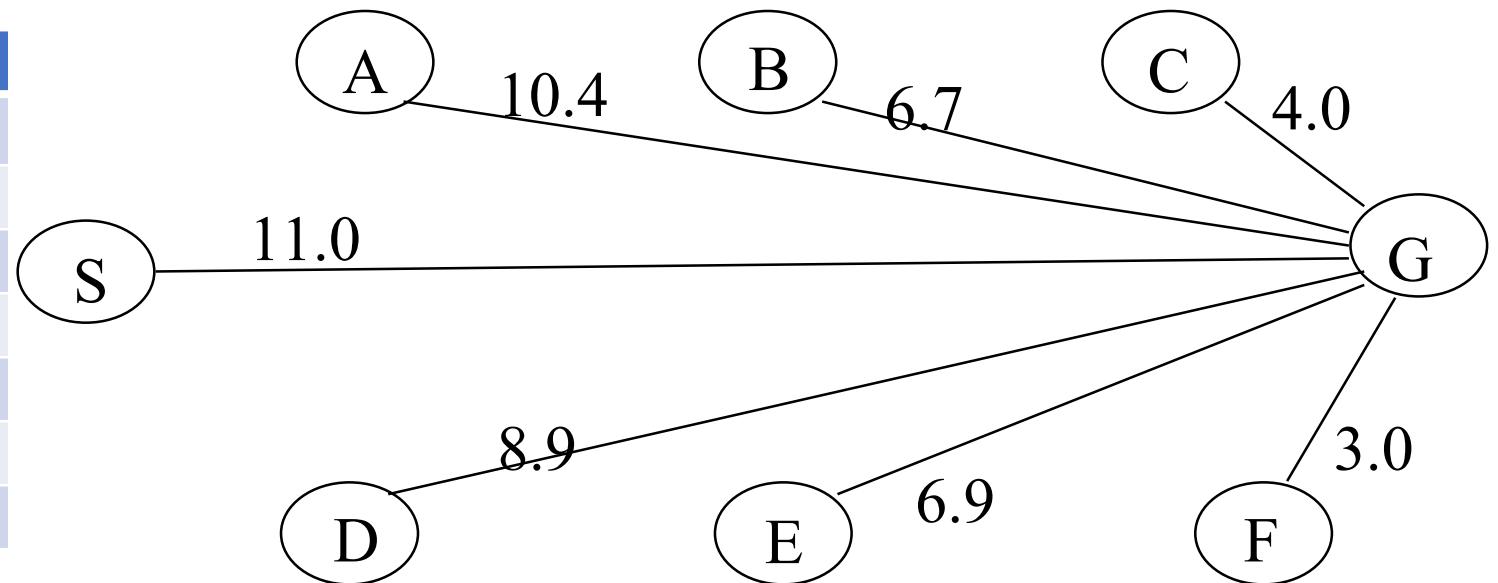


Example



N	V
F	G

Open	Closed
	S (g = 0, f = 11, prev=∅)
	A (g = 2, f = 12.4, prev=S)
	B (g = 3, f = 9.7, prev=A)
	C (g = 7, f = 11, prev=B)
	D (g = 5, f = 12.9, prev=A)
	E(g=8, f = 13.9, prev=D)
	F(g=11,f=14,prev=E)

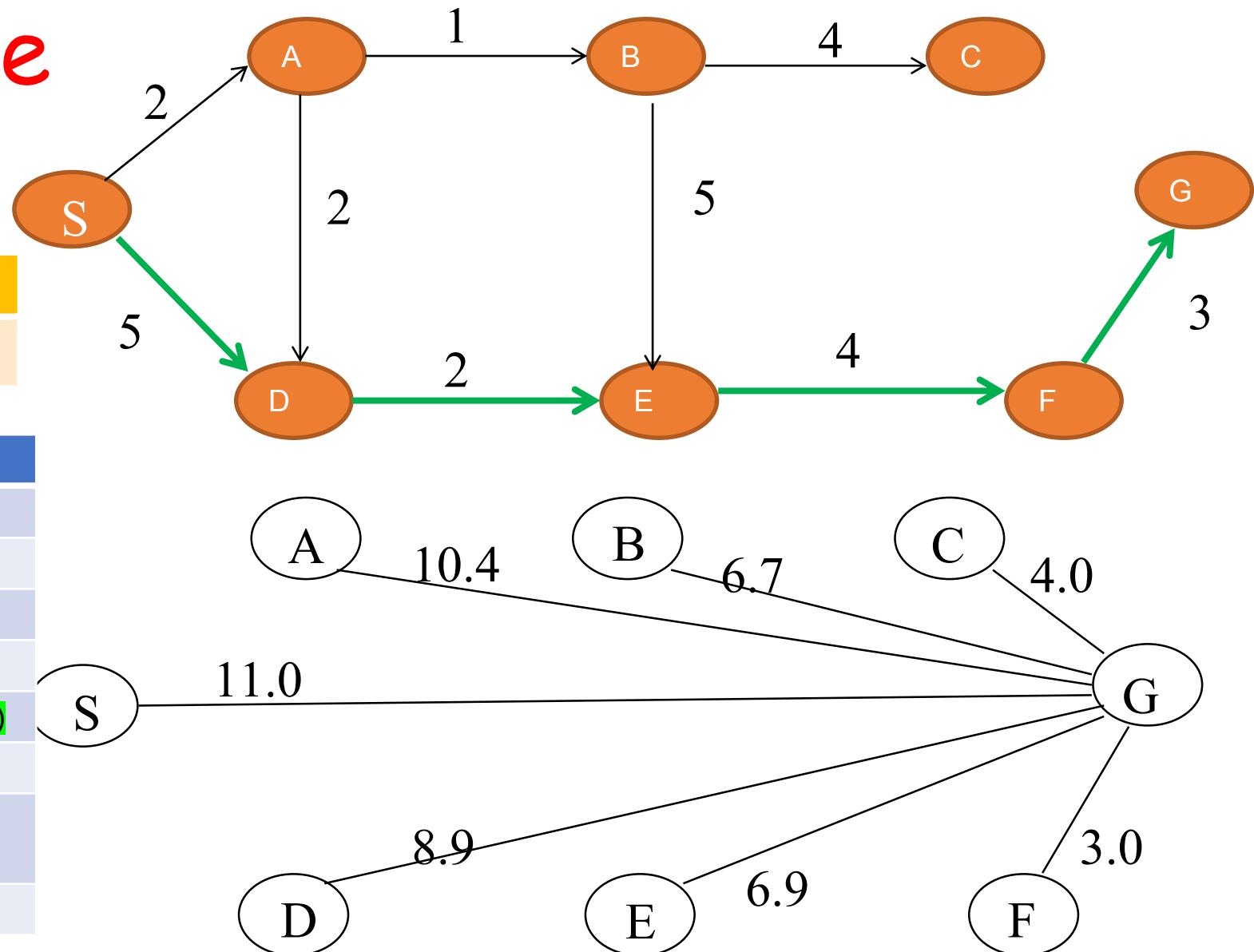




Example

N	V
G	=

Open	Closed
	S ($g = 0, f = 11, \text{prev} = \emptyset$)
	A ($g=2, f=12.4, \text{prev}=S$)
	B ($g = 3, f = 9.7, \text{prev}=A$)
	C ($g = 7, f = 11, \text{prev}=B$)
	D ($g = 5, f = 12.9, \text{prev}=A$)
	E($g=8, f = 13.9, \text{prev}=D$)
	F($g=11,f=14,\text{prev}=E$)
	G($g=14,f=14,\text{prev}=F$)





Observations

- Correctness:
 - If the heuristic $h(n)$ never overestimates the true cost, the algorithm finds the optimal solution.”
- Special choice of heuristic function
 - If we choose $h(n) = 0$ A* reduces to Djikstra
- Both Djikstra and A* can be seen as special cases of a more general framework known as Dynamic Programming



Concept Check

- What is the difference between the single-source shortest path and the single-destination shortest path problem?
- Why does Dijkstra's algorithm require non-negative edge weights?
- What happens if negative weights are present?



Concept Check - Answers

- Single-source: shortest paths from one source to all nodes.
- Single-destination: shortest paths to one target (can be solved by reversing edges).
- Non-negative weights ensure once a node is visited, its shortest path is final.
- With negative weights, Dijkstra may fail; use Bellman–Ford instead.



Quick Questions

- True or False:
- 1. Dijkstra's algorithm always visits nodes in decreasing order of distance.
- 2. A* is guaranteed to find the optimal path if the heuristic never overestimates the true cost.
- 3. Using a priority queue improves Dijkstra's runtime from $O(|V|^2)$ to $O((|E| + |V|) \log |V|)$.



Quick Questions - Answers

- 1. False – Dijkstra always expands the vertex with smallest tentative distance.
- 2. True – admissible heuristics guarantee optimality.
- 3. True – binary heap or priority queue gives $O((|E|+|V|) \log |V|)$.



Coding Practice

- Implement Dijkstra's algorithm in Python or MATLAB using a priority queue.
- Run your implementation on a random sparse graph with 100 nodes.
- Compare runtimes with a naive $O(|V|^2)$ array-based implementation.



Coding Practice - Answers

- Priority queue version is much faster on sparse graphs.
- Naive version: $O(|V|^2)$.
- Priority queue: $O((|E|+|V|) \log |V|)$.



A* Practice

- Design a grid-world (e.g., 10x10 grid with some blocked cells).
- Apply both Dijkstra and A* to find a path from the top-left to the bottom-right corner.
- Compare: number of nodes expanded, computation time, and path quality.



A* Practice - Answers

- Dijkstra explores all nodes until goal reached.
- A* explores far fewer if heuristic is effective.
- Both yield same path length if heuristic is admissible.
- A* usually faster, expands fewer nodes.



Challenge Question

- Prove that if the heuristic $h(n)$ used in A* is admissible (never overestimates) and consistent (monotone), then A* is guaranteed to return an optimal path.
- Show an example where an inadmissible heuristic causes A* to fail to find the shortest path.



Challenge Question - Answers

- Admissible + consistent heuristic $\Rightarrow f(n)=g(n)+h(n)$ is non-decreasing.
- Thus first time goal is expanded, path is optimal.
- Counterexample: If h overestimates, A* may prune true shortest path and return a longer one.



Homework

- Implement both Dijkstra and A* for a weighted graph.
- Experiment with different heuristic functions for A*.
- Report runtime comparison, memory usage, and path length.
- Write a short reflection: When is A* preferable to Dijkstra?



Homework - Answers

- Benchmarks: Dijkstra slower, explores more nodes.
- A* faster with good heuristics, especially in large graphs.
- Both produce same path length with admissible heuristic.
- A* preferable when we care about one goal and have a good heuristic.



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Dynamic Programming

Luigi Palopoli (adapted from Laksman Veeravagu and Luis Barrera et al.)
DISI – UNITN 2022



Algorithmic Paradigms

- **Greedy**: Build a solution myopically optimising some local criterion
- **Divide and Conquer**: Break-up a problem into subproblems, solve each independently and then combine their solutions to find the solution of the original problem
- **Dynamic Programming**: Break-up a problem into a series of overlapping subproblems and build up solutions to larger and larger subproblems
 - Both A* and Dijkstra exploit DP principles, though A* is usually presented in the context of heuristic search.”



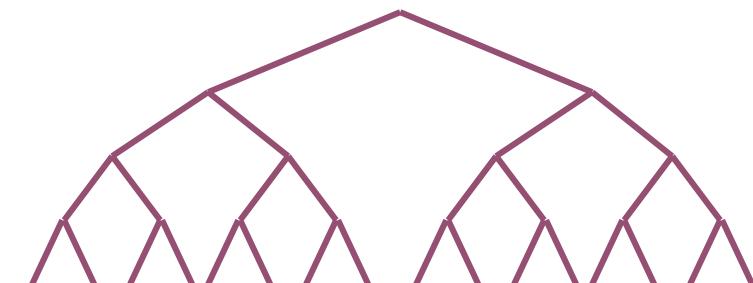
Greedy Algorithms

- Greedy algorithms focus on making the best local choice at each decision point.
- For example, a natural way to compute a shortest path from x to y might be to walk out of x , repeatedly following the cheapest edge until we get to y . Not always correct!!
- In the absence of a correctness proof, greedy algorithms are very likely to fail.



Divide-and-conquer

1. split problem into smaller problems
2. solve each smaller problem recursively
 1. split smaller problem into even smaller problems
 2. solve each even smaller problem recursively
 1. split smaller problem into tiny problems
 2. ...
 3. ...
3. recombine the results
3. recombine the results



should remind you of backtracking

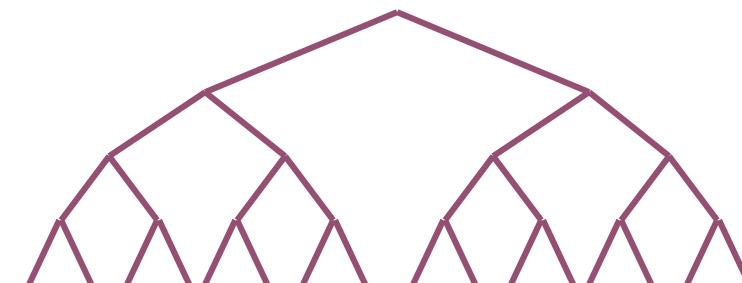


Dynamic programming

Exactly the same as divide-and-conquer ...
but store the solutions to subproblems for possible reuse.

A good idea if many subproblems recur.

There might be $O(2^n)$
nodes in this tree,
but only e.g. $O(n^3)$
different nodes.



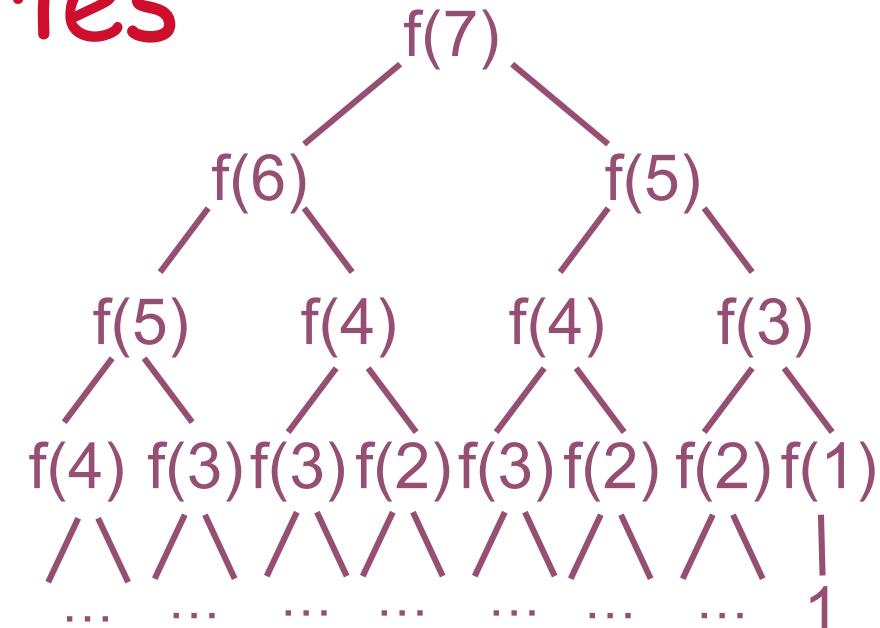
This should remind you of backtracking



Fibonacci series

- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- $f(0) = 0.$
- $f(1) = 1.$
- $f(N) = f(N-1) + f(N-2)$
if $N \geq 2.$

```
int f(int n) {  
    if n < 2  
        return n  
    else  
        return f(n-1) + f(n-2)  
}
```



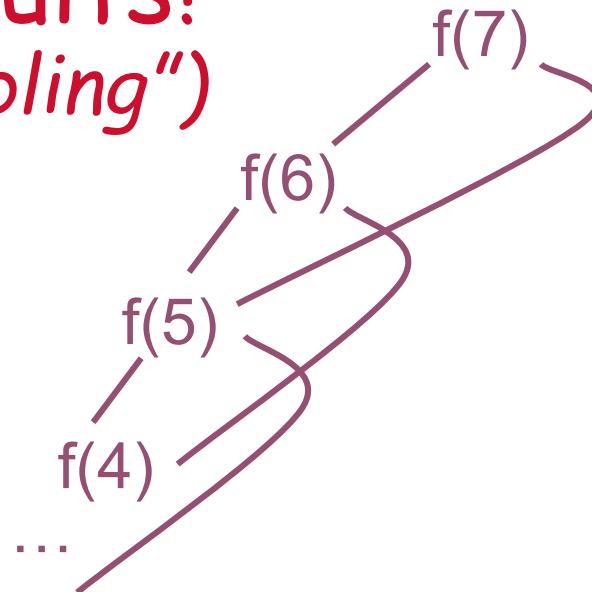
$f(n)$ takes exponential time to compute.

Proof: $f(n)$ takes more than twice as long as $f(n-2)$, which therefore takes more than twice as long as $f(n-4)$...
Don't you do it faster?



Reuse earlier results! ("memoisation" or "tabling")

- 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- $f(0) = 0$.
- $f(1) = 1$.
- $f(N) = f(N-1) + f(N-2)$
if $N \geq 2$.



```
int f(int n) {  
    if n < 2  
        return n  
    else  
        return fmemo(n-1) + fmemo(n-2)  
}
```

Does it matter which of these we call first?

```
int fmemo(int n) {  
    if f[n] is undefined  
        f[n] = f(n)  
    return f[n]  
}
```



1-dimensional DP Problem

- ▶ Problem: given n , find the number of different ways to write n as the sum of 1, 3, 4
- ▶ Example: for $n = 5$, the answer is 6

$$\begin{aligned} 5 &= 1 + 1 + 1 + 1 + 1 \\ &= 1 + 1 + 3 \\ &= 1 + 3 + 1 \\ &= 3 + 1 + 1 \\ &= 1 + 4 \\ &= 4 + 1 \end{aligned}$$



1-dimensional DP Problem

- ▶ Define subproblems
 - Let D_n be the number of ways to write n as the sum of 1, 3, 4
- ▶ Find the recurrence
 - Consider one possible solution $n = x_1 + x_2 + \dots + x_m$
 - If $x_m = 1$, the rest of the terms must sum to $n - 1$
 - Thus, the number of sums that end with $x_m = 1$ is equal to D_{n-1}
 - Take other cases into account ($x_m = 3, x_m = 4$)



1-dimensional DP Problem

- ▶ Recurrence is then

$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$

- ▶ Solve the base cases
 - $D_0 = 1$
 - $D_n = 0$ for all negative n
 - Alternatively, can set: $D_0 = D_1 = D_2 = 1$, and $D_3 = 2$
- ▶ We're basically done!



1-dimensional DP Problem

```
D[0] = D[1] = D[2] = 1; D[3] = 2;  
for(i = 4; i <= n; i++)  
    D[i] = D[i-1] + D[i-3] + D[i-4];
```

- ▶ Very short!



Backward chaining vs. forward chaining

- Recursion is sometimes called “backward chaining”: start with the goal you want, $f(7)$, choosing your subgoals $f(6)$, $f(5)$, ... on an as-needed basis.
 - Reason backwards from goal to facts
(start with goal and look for support for it)
- Another option is “forward chaining”: compute each value as soon as you can, $f(0)$, $f(1)$, $f(2)$, $f(3)$... with the aim of reaching the goal.
 - Reason forward from facts to goal
(start with what you know and look for things you can prove)
- **Either way, you should table results that you'll need later**
- Mixing forward and backward is possible (future topic)



DP: a more formal definition

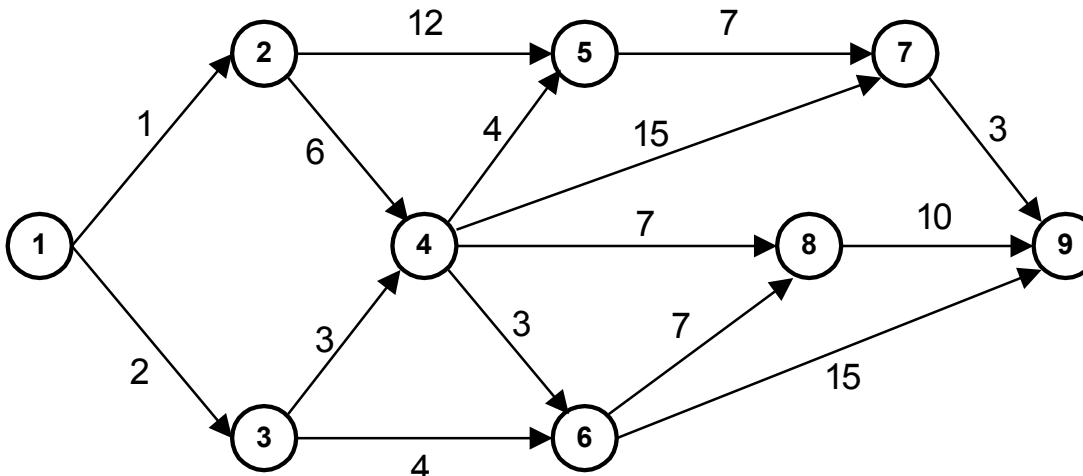
- A specific type of mathematical programming in which the optimal solution of the original problem is found by solving *a chain of subproblems*.
- In dynamic programming, the optimal solution of one subproblem will be used as input to the next subproblem. When the last subproblem is solved, the optimal solution for the *entire* problem is obtained, which necessarily includes the solution to the original problem.
- Linkage between the stages of a DP problem is performed through *recursive computations*. Depending on the nature of the problem at hand, *forward recursive equation* or *backward recursive equation* will be developed for finding solution.



Shortest path (1)

Example 1: Shortest Path Problem

Find the shortest path from node 1 to node 9 of the network:



Define:

f_i : minimum total travel time from node i to node 9.

t_{ij} : travel time through the directed arc (i,j) .



Shortest Path (2)

On an arbitrary arc (i, j) it can be seen that:

$$f_i \leq t_{ij} + f_j \quad i \neq 9, \forall j$$

Hence:

$$f_i \leq \min_j \{t_{ij} + f_j\} \quad i \neq 9$$

However, the shortest path from node i to node 9 should include some intermediate node j (if these intermediate nodes exist). Then,

$$f_i = \min_j \{t_{ij} + f_j\} \quad i \neq 9$$

The above equation is the *recursive equation* (or functional equation) of the shortest path problem in the *backward* form.



Shortest Path (3)

Based on the recursive equation, the optimal solution can be found as follows:

$$f_9 = 0$$

$$f_8 = t_{89} + f_9 = 10 + 0 = 10$$

$$f_7 = t_{79} + f_9 = 3 + 0 = 3$$

$$f_6 = \min \left\{ \begin{array}{l} t_{68} + f_8 \\ t_{69} + f_9 \end{array} \right\} = \min \left\{ \begin{array}{l} 7 + 10 \\ 15 + 0 \end{array} \right\} = 15$$

$$f_5 = t_{57} + f_7 = 7 + 3 = 10$$

$$f_4 = \min \left\{ \begin{array}{l} t_{45} + f_5 \\ t_{46} + f_6 \\ t_{47} + f_7 \\ t_{48} + f_8 \end{array} \right\} = \min \left\{ \begin{array}{l} 4 + 10 \\ 3 + 15 \\ 15 + 3 \\ 7 + 10 \end{array} \right\} = 14$$



Shortest Path (3)

$$f_3 = \min \left\{ \begin{array}{l} t_{34} + f_4 \\ t_{36} + f_6 \end{array} \right\} = \min \left\{ \begin{array}{l} 3 + 14 \\ 4 + 15 \end{array} \right\} = 17$$

$$f_2 = \min \left\{ \begin{array}{l} t_{24} + f_4 \\ t_{25} + f_5 \end{array} \right\} = \min \left\{ \begin{array}{l} 6 + 14 \\ 12 + 10 \end{array} \right\} = 20$$

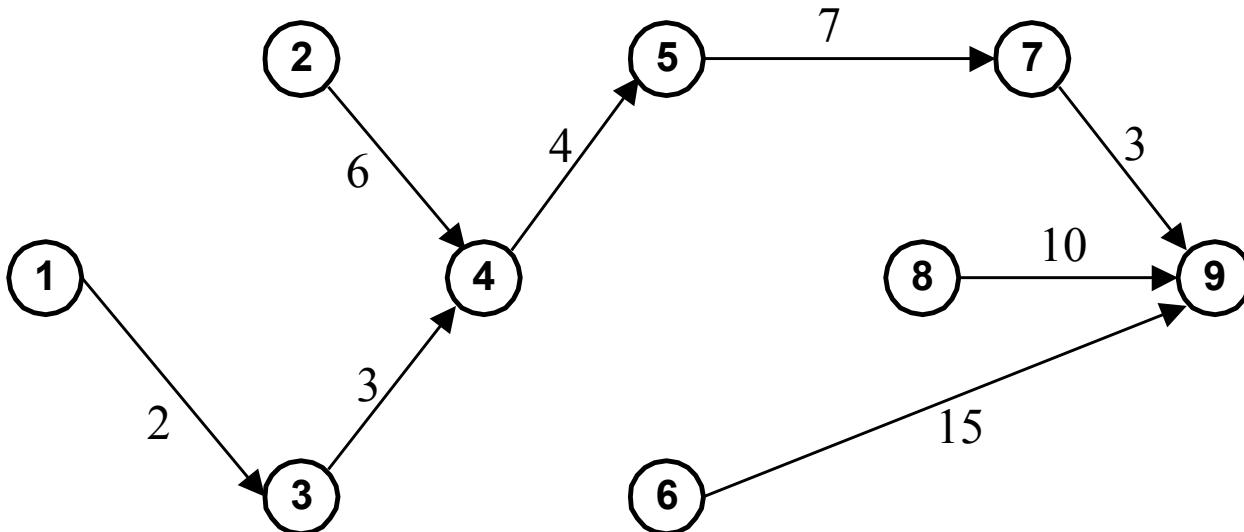
$$f_1 = \min \left\{ \begin{array}{l} t_{12} + f_2 \\ t_{13} + f_3 \end{array} \right\} = \min \left\{ \begin{array}{l} 1 + 20 \\ 2 + 17 \end{array} \right\} = 19$$

Shortest path from node 1 to node 9: 1-3-4-5-7-9 with total travel time = 19.



Observation

It is noted that each subproblem is associated with a network's node and when the shortest path from node 1 to node 9 is determined, we also know the shortest paths from **every node of the network to node 9**.





Forward recursion

The above problem can also be solved by use of *forward* recursive equation as presented below
Define:

f_j : minimum total travel time from node 1 to node j .
 t_{ij} : travel time through the directed arc (i, j) .

Start node

On an arbitrary arc (i, j) , it can be seen that:

$$f_j \leq t_{ij} + f_i \quad j \neq 1, \forall i$$

Hence:

$$f_j \leq \min_i \{t_{ij} + f_i\} \quad j \neq 1$$

However, the shortest path from node 1 to node j should include some intermediate node i (if these intermediate nodes exist). Then,

$$f_j = \min_i \{t_{ij} + f_i\} \quad j \neq 1$$



Forward recursion (1)

Solution can be found recursively as follows:

$$f_1 = 0$$

$$f_2 = t_{12} + f_1 = 1 + 0 = 1$$

$$f_3 = t_{13} + f_1 = 2 + 0 = 2$$

$$f_4 = \min \left\{ \begin{array}{l} t_{24} + f_2 \\ t_{34} + f_3 \end{array} \right\} = \min \left\{ \begin{array}{l} 6 + 1 \\ 3 + 2 \end{array} \right\} = 5$$

$$f_5 = \min \left\{ \begin{array}{l} t_{25} + f_2 \\ t_{45} + f_4 \end{array} \right\} = \min \left\{ \begin{array}{l} 12 + 1 \\ 4 + 5 \end{array} \right\} = 9$$

$$f_6 = \min \left\{ \begin{array}{l} t_{36} + f_3 \\ t_{64} + f_4 \end{array} \right\} = \min \left\{ \begin{array}{l} 4 + 2 \\ 3 + 5 \end{array} \right\} = 6$$



Forward recursion (2)

$$f_7 = \min \left\{ \begin{array}{l} t_{47} + f_4 \\ t_{57} + f_5 \end{array} \right\} = \min \left\{ \begin{array}{l} 15 + 5 \\ 7 + 9 \end{array} \right\} = 16$$

$$f_8 = \min \left\{ \begin{array}{l} t_{48} + f_4 \\ t_{68} + f_6 \end{array} \right\} = \min \left\{ \begin{array}{l} 7 + 5 \\ 7 + 6 \end{array} \right\} = 12$$

$$f_9 = \min \left\{ \begin{array}{l} t_{69} + f_6 \\ t_{79} + f_7 \\ t_{89} + f_8 \end{array} \right\} = \min \left\{ \begin{array}{l} 15 + 6 \\ 3 + 16 \\ 10 + 12 \end{array} \right\} = 19$$

The solution from forward recursive equation gives the shortest paths from **node 1 to every other nodes** of the network, not only to node 9.



Forward recursion (2)

$$f_7 = \min \left\{ \begin{array}{l} t_{47} + f_4 \\ t_{57} + f_5 \end{array} \right\} = \min \left\{ \begin{array}{l} 15 + 5 \\ 7 + 9 \end{array} \right\} = 16$$

$$f_8 = \min \left\{ \begin{array}{l} t_{48} + f_4 \\ t_{68} + f_6 \end{array} \right\} = \min \left\{ \begin{array}{l} 7 + 5 \\ 7 + 6 \end{array} \right\} = 12$$

$$f_9 = \min \left\{ \begin{array}{l} t_{69} + f_6 \\ t_{79} + f_7 \\ t_{89} + f_8 \end{array} \right\} = \min \left\{ \begin{array}{l} 15 + 6 \\ 3 + 16 \\ 10 + 12 \end{array} \right\} = 19$$

The solution from forward recursive equation gives the shortest paths from **node 1 to every other nodes** of the network, not only to node 9.



Observation

Note:

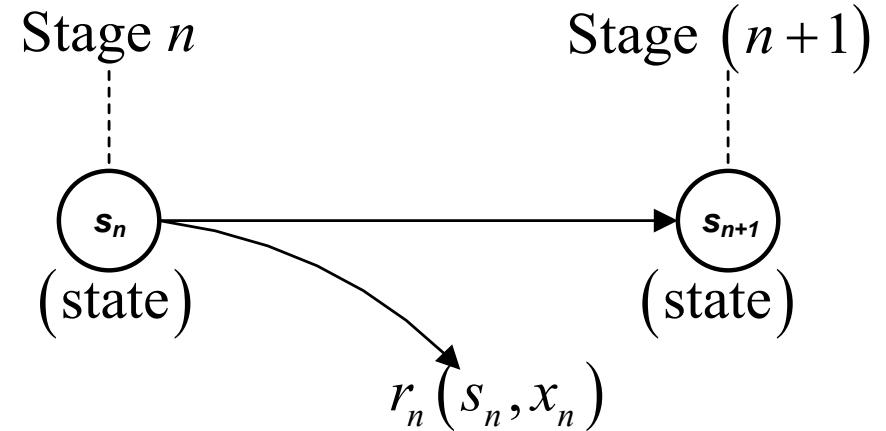
In the example just seen, we could approach the problem both by forward and backward recursion.

- Not all DP programs can be solved by both forward and backward recursive techniques.
- The use of backward recursion or forward recursion depends on the specific structure of the problem under consideration.



Going formal

Structure of a DP problem



- At state s_n in stage n , if decision x_n is taken the current state s_n will be transferred to a new state s_{n+1} in stage $(n + 1)$.
- A revenue $r_n(s_n, x_n)$ will be obtained by decision x_n taken at state s_n
- The new state s_{n+1} is also a function of s_n and x_n , and can be expressed in form of a transformation function: $s_{n+1} = t_n(s_n, x_n)$.



Bellman's Principle of optimality

If the problem has a forward or backward recursive structure, then an optimal policy has the property that whatever the initial state and the initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision

The Bellman's optimality principle helps establish recursive equations when the structure of the problem can be arranged in stages.



Bellman's Principle of optimality: Backward recursion

In case of maximisation problem and backward recursive technique is employed, if we denote $f_n(s_n)$ as the maximum total revenue obtained when the system moves from stage n to stage N (the last stage), given the observed state at stage n is s_n , then:

$$f_n(s_n) = \max_{x_n \in D(s_n)} \left\{ r_n(s_n, x_n) + f_{n+1}(t_n(s_n, x_n)) \right\}$$

In which $D(s_n)$ is the set of all possible decisions of a *given* state s_n at stage n (decision set).



Bellman's Principle of optimality: Forward recursion

Similarly, when the **forward recursive** technique is employed, if we denote $f_n(s_n)$ as the **maximum total revenue** when the system move from stage 1 to stage n , given the observed state at stage n is s_n , then:

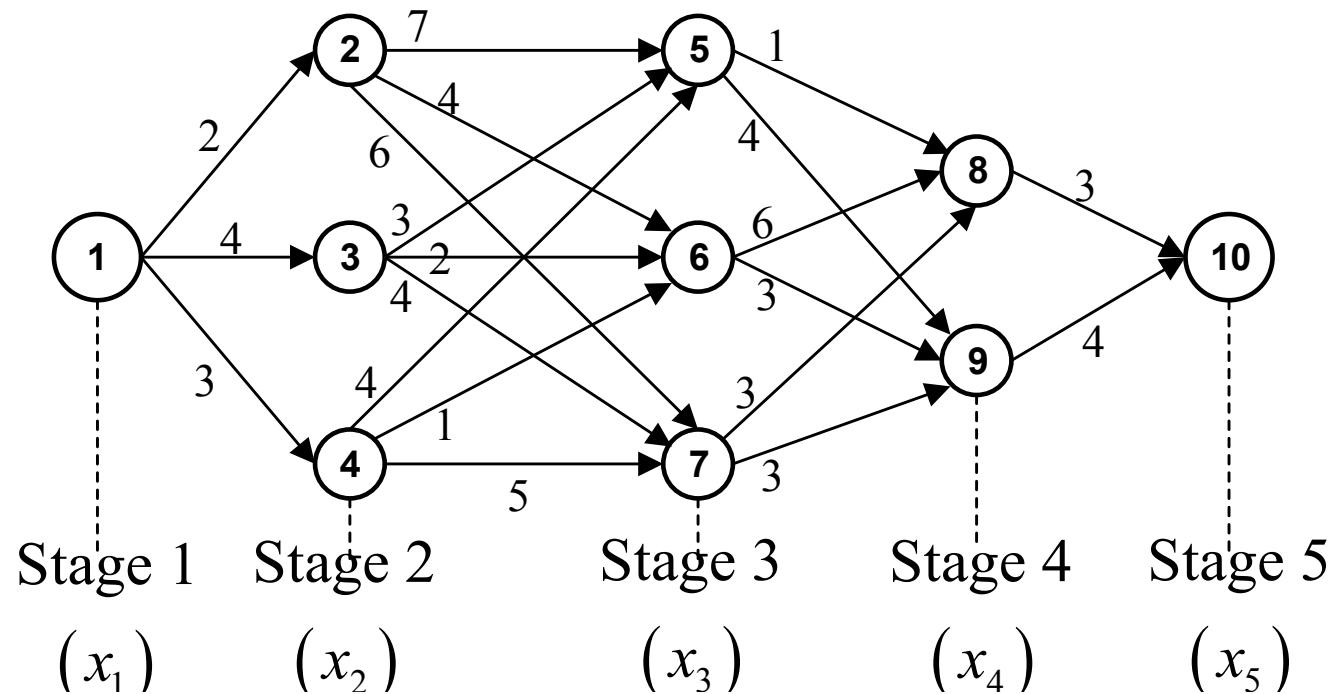
$$f_{n+1}(s_{n+1}) = \max_{x_n \in D(s_{n+1})} \left\{ r_n(s_n, x_n) + f_n(s_n) \right\}$$

In which $D(s_{n+1})$ is the set of all possible decisions x_n at stage n such that these decisions will help to transfer the states s_n 's at stage n to a **predefined** state s_{n+1} in stage $(n + 1)$, i.e., $s_{n+1} = t_n(s_n, x_n)$.



Back to the example

Find the shortest path from node 1 to node 10 of the network





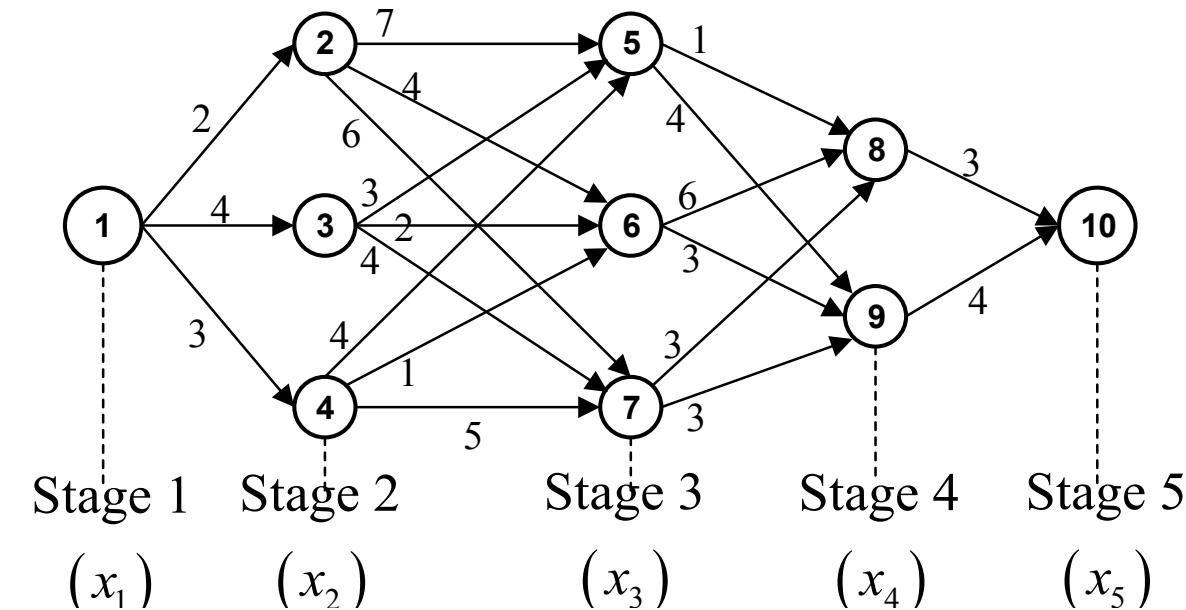
Back to the example

Applying backward recursive approach, the solution can be found as follows:

Stage 4:

x_4	$r_4(s_4, x_4) + f_5(t_4(s_4, x_4))$	$f_4(s_4)$	x_4^*
s_4	10		
Node 8	3	3	10
Node 9	4	4	10

In stage 4, the state s_4 can be *node 8* or *node 9*, and the only decision (i.e., x_4) that can be taken is *go to node 10*.





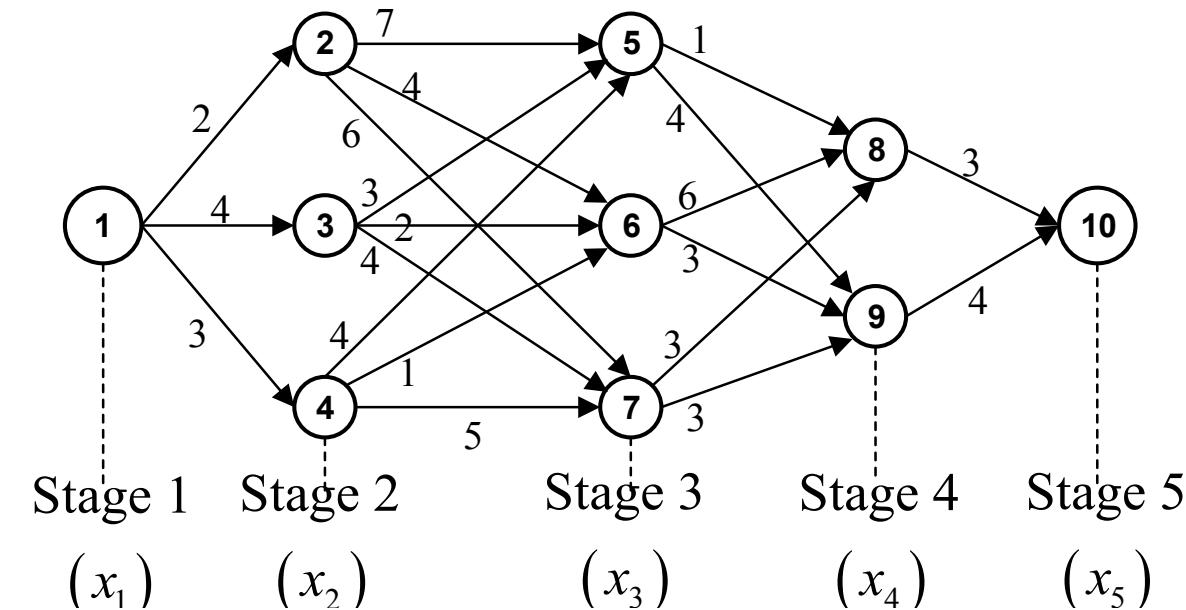
Back to the example

Applying backward recursive approach, the solution can be found as follows:

Stage 4:

x_4	$r_4(s_4, x_4) + f_5(t_4(s_4, x_4))$	$f_4(s_4)$	x_4^*
s_4	10		
Node 8	3	3	10
Node 9	4	4	10

In stage 4, the state s_4 can be *node 8* or *node 9*, and the only decision (i.e., x_4) that can be taken is *go to node 10*.





Back to the example

When the state is *node 8*:

$$r_4(s_4, x_4) = r_4(8, 10) = 3 \quad f_5(t_4(s_4, x_4)) = f_5(10) = 0$$
$$\Rightarrow f_4(s_4) = f_4(8) = 3$$

When the state is *node 9*:

$$r_4(s_4, x_4) = r_4(9, 10) = 4 \quad f_5(t_4(s_4, x_4)) = f_5(10) = 0$$
$$\Rightarrow f_4(s_4) = f_4(9) = 4$$

There exists only one possible decision, so it is the optimal decision.

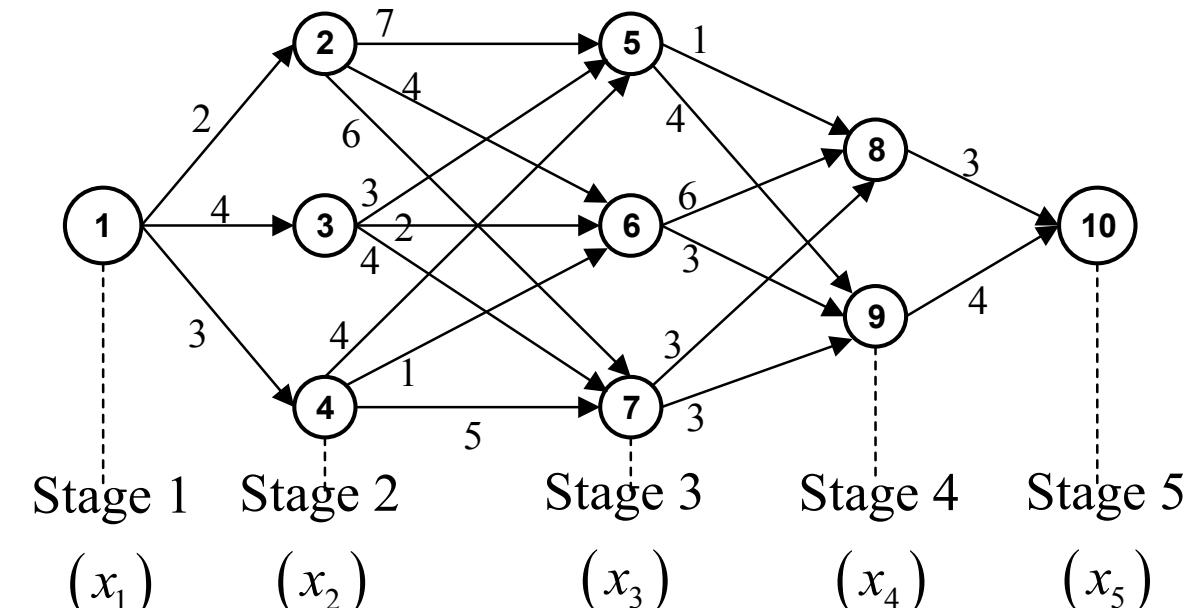


Back to the example

Stage 3:

x_3	$r_3(s_3, x_3) + f_4(t_3(s_3, x_3))$	$f_3(s_3)$	x_3^*
s_3	8	9	
Node 5	$1 + 3 = 4$	$4 + 4 = 8$	4
Node 6	$6 + 3 = 9$	$3 + 4 = 7$	7
Node 7	$3 + 3 = 6$	$3 + 4 = 7$	6

In this stage, the state s_3 can be *node 5*, *node 6* or *node 7*. The decision x_3 can be *go to node 8* or *go to node 9*.





Back to the example

When the state is *node 5*:

If the decision is *go to node 8*:

$$r_3(s_3, x_3) = r_3(5, 8) = 1 \quad f_4(t_3(s_3, x_3)) = f_4(8) = 3$$

If the decision is *go to node 9*:

$$r_3(s_3, x_3) = r_3(5, 9) = 4 \quad f_4(t_3(s_3, x_3)) = f_4(9) = 4$$

$$\Rightarrow f_3(s_3) = f_3(5) = \text{Min}\{1 + 3, 4 + 4\} = 4$$

\Rightarrow Optimal decision: *go to node 8*



Back to the example

When the state is *node 6*:

If the decision is *go to node 8*:

$$r_3(s_3, x_3) = r_3(6, 8) = 6 \quad f_4(t_3(s_3, x_3)) = f_4(8) = 3$$

If the decision is *go to node 9*:

$$r_3(s_3, x_3) = r_3(6, 9) = 3 \quad f_4(t_3(s_3, x_3)) = f_4(9) = 4$$

$$\Rightarrow f_3(s_3) = f_3(6) = \text{Min}\{6 + 3, 3 + 4\} = 7$$

\Rightarrow Optimal decision: *go to node 9*



Back to the Example

When the state is *node 7*:

If the decision is *go to node 8*:

$$r_3(s_3, x_3) = r_3(7, 8) = 3 \quad f_4(t_3(s_3, x_3)) = f_4(8) = 3$$

If the decision is *go to node 9*:

$$r_3(s_3, x_3) = r_3(7, 9) = 3 \quad f_4(t_3(s_3, x_3)) = f_4(9) = 4$$

$$\Rightarrow f_3(s_3) = f_3(7) = \text{Min}\{3 + 3, 3 + 4\} = 6$$

\Rightarrow Optimal decision: *go to node 8*

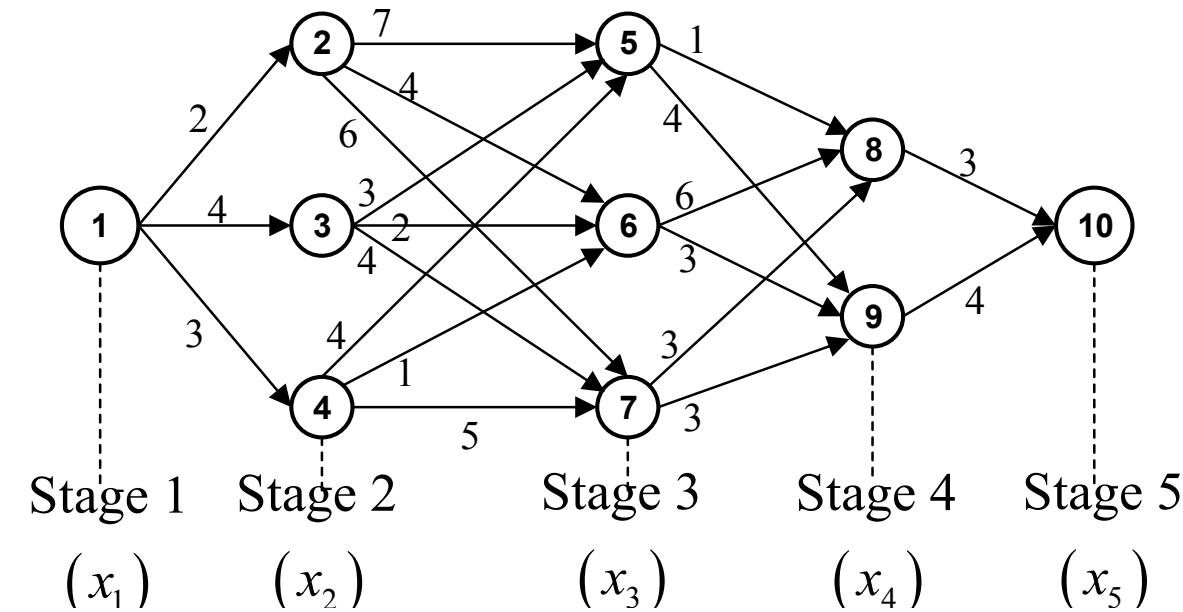


Back to the example

Stage 2:

x_2	$r_2(s_2, x_2) + f_3(t_2(s_2, x_2))$		Min	x_2^*
s_2			$f_2(s_2)$	
Node 2	5	6	7	
Node 2	$7 + 4 = 11$	$4 + 7 = 11$	$6 + 6 = 12$	11
Node 3	$3 + 4 = 7$	$2 + 7 = 9$	$4 + 6 = 10$	7
Node 4	$4 + 4 = 8$	$1 + 7 = 8$	$5 + 6 = 11$	8
				5 or 6

In this stage, the state s_2 can be *node 2*, *node 3* or *node 4*. The decision x_2 can be *go to node 5*, *go to node 6*, or *go to node 7*.





Back to the Example

When the state is *node 2*:

If the decision is *go to node 5*:

$$r_2(s_2, x_2) = r_2(2, 5) = 7$$

$$f_3(t_2(s_2, x_2)) = f_3(5) = 4$$

If the decision is *go to node 6*:

$$r_2(s_2, x_2) = r_2(2, 6) = 4$$

$$f_3(t_2(s_2, x_2)) = f_3(6) = 7$$

If the decision is *go to node 7*:

$$r_2(s_2, x_2) = r_2(2, 7) = 6$$

$$f_3(t_2(s_2, x_2)) = f_3(7) = 6$$

$$\Rightarrow f_2(s_2) = f_2(2) = \text{Min}\{7 + 4, 4 + 7, 6 + 6\} = 11$$

\Rightarrow Optimal decision: *go to node 5 or go to node 6*



Back to the example

When the state is *node 3*:

If the decision is *go to node 5*:

$$r_2(s_2, x_2) = r_2(3, 5) = 3$$

$$f_3(t_2(s_2, x_2)) = f_3(5) = 4$$

If the decision is *go to node 6*:

$$r_2(s_2, x_2) = r_2(3, 6) = 2$$

$$f_3(t_2(s_2, x_2)) = f_3(6) = 7$$

If the decision is *go to node 7*:

$$r_2(s_2, x_2) = r_2(3, 7) = 4$$

$$f_3(t_2(s_2, x_2)) = f_3(7) = 6$$

$$\Rightarrow f_2(s_2) = f_2(3) = \text{Min}\{3 + 4, 2 + 7, 4 + 6\} = 7$$

\Rightarrow Optimal decision: *go to node 5*



Back to the example

When the state is *node 4*:

If the decision is *go to node 5*:

$$r_2(s_2, x_2) = r_2(4, 5) = 4$$

$$f_3(t_2(s_2, x_2)) = f_3(5) = 4$$

If the decision is *go to node 6*:

$$r_2(s_2, x_2) = r_2(4, 6) = 1$$

$$f_3(t_2(s_2, x_2)) = f_3(6) = 7$$

If the decision is *go to node 7*:

$$r_2(s_2, x_2) = r_2(4, 7) = 5$$

$$f_3(t_2(s_2, x_2)) = f_3(7) = 6$$

$$\Rightarrow f_2(s_2) = f_2(3) = \text{Min}\{4 + 4, 1 + 7, 5 + 6\} = 8$$

\Rightarrow Optimal decision: *go to node 5 or go to node 6*

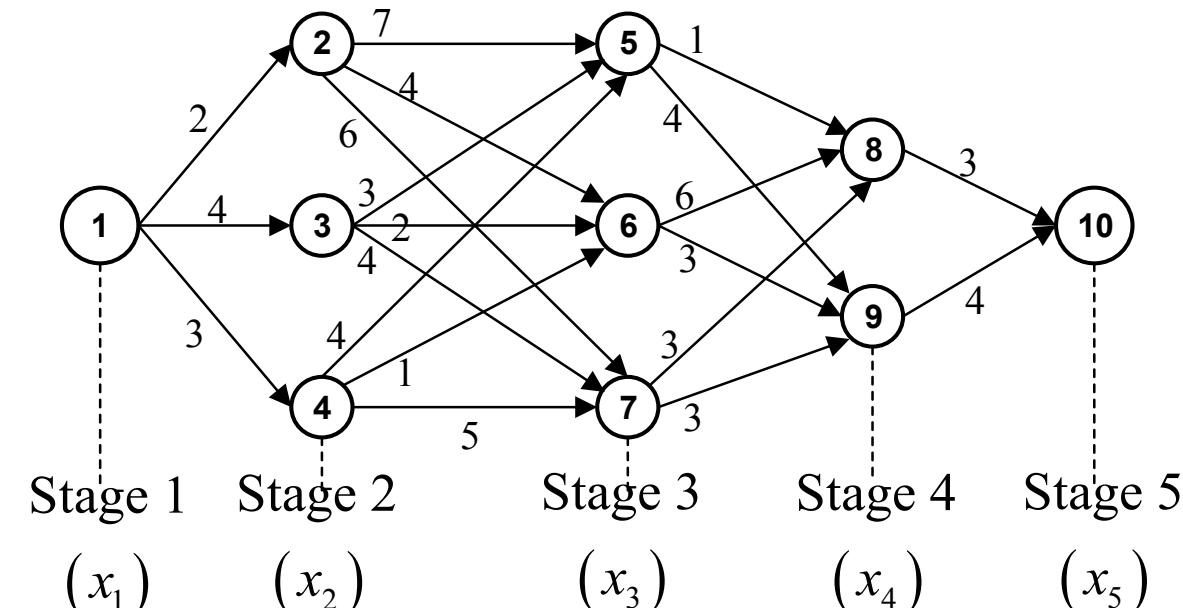


Back to the example

Stage 1:

x_1	$r_1(s_1, x_1) + f_2(t_1(s_1, x_1))$	$f_1(s_1)$	x_1^*
s_1	2 3 4		
Node 1	2 + 11 = 13 4 + 7 = 11 3 + 8 = 11	11	3 or 4

In this stage, the state s_1 is *node 1*. The decisions x_1 can be *go to node 2*, *go to node 3*, or *go to node 4*.





Back to the example

If the decision is *go to node 2*:

$$r_1(s_1, x_1) = r_1(1, 2) = 2 \quad f_2(t_1(s_1, x_1)) = f_2(2) = 11$$

If the decision is *go to node 3*:

$$r_1(s_1, x_1) = r_1(1, 3) = 4 \quad f_2(t_1(s_1, x_1)) = f_2(3) = 7$$

If the decision is *go to node 7*:

$$r_1(s_1, x_1) = r_1(1, 4) = 3 \quad f_2(t_1(s_1, x_1)) = f_2(4) = 8$$

$$\Rightarrow f_1(s_1) = f_1(1) = \text{Min}\{2 + 11, 4 + 7, 3 + 8\} = 11$$

\Rightarrow Optimal decision: *go to node 3 or go to node 4*



Optimal Solution

The optimal solution has been found. There exist three shortage paths from node 1 to node 10:

1-3-5-8-10, 1-4-5-8-10, 1-4-6-9-10



Check Question 1: Algorithmic Paradigms

- What is the main difference between divide & conquer and dynamic programming?



Answer 1

- Divide & Conquer: independent subproblems, then combine results.
- Dynamic Programming: overlapping subproblems, reuse stored solutions.
- Key: DP avoids recomputation by storing results.



Check Question 2: Fibonacci

- When computing $f(6)$ using naïve recursion, how many times is $f(3)$ evaluated?



Answer 2

- Recursive tree expands $f(5) + f(4)$, which call $f(3)$ multiple times.
- $f(3)$ is computed 3 times.
- DP reduces this to once.



Check Question 3: Shortest Path

- What is the recursive equation for the shortest distance from node i to the destination?



Answer 3

- Recursive definition: $f_i = \min_j (t_{ij} + f_j)$, for $i \neq n$
- Base case: $f_n = 0$
- This is the DP recurrence for shortest path.



Check Question 4: Forward vs Backward

- What is the difference between forward and backward recursion in shortest path problems?



Answer 4

- Backward: start from destination, work backwards.
- Forward: start from source, expand forward.
- Both use same recurrence; choice depends on structure.



Check Question 5: Bellman's Principle

- State Bellman's Principle of Optimality in your own words.



Answer 5

- "An optimal solution has the property that any subpath of it is also optimal."
- Meaning: once a step is chosen, the remaining problem is itself optimally solvable.



Check Question 6: Divide & Conquer vs DP

- Suppose a problem can be solved by both divide & conquer and dynamic programming.
- Is DP always the better choice? Why or why not?



Answer 6

- Not always: storing results adds overhead if subproblems are independent.
- Divide & Conquer may be simpler/faster (e.g., merge sort).
- DP is most powerful when subproblems overlap significantly.



Check Question 7: Fibonacci Space Optimisation

- Using bottom-up DP for Fibonacci, we compute values from $f(0)$ to $f(n)$.
- If we only need $f(n)$ once, is it wasteful to compute all values?
- How could this be optimised?



Answer 7

- Yes: storing the entire table uses $O(n)$ space.
- Only last two values are needed at each step.
- Optimisation: reduce space to $O(1)$ with two variables.



Check Question 8: Negative Weights

- Dijkstra's algorithm is often described as a DP algorithm.
- Why does it fail with negative edge weights, while Bellman–Ford succeeds?



Answer 8

- Dijkstra assumes once a node's shortest path is fixed, it never improves.
- Negative edges break this assumption.
- Bellman–Ford systematically relaxes edges via DP recurrence.
- Bellman–Ford handles negatives; Dijkstra cannot.



Check Question 9: Forward vs Backward

- Give an example of a problem that can only be solved via backward recursion.
- Why is forward recursion not applicable?



Answer 9

- Example: stochastic decision problems with uncertain future costs.
- Must reason backward from terminal states.
- Forward recursion cannot apply when future outcomes are unknown.



Check Question 10: Bellman's Principle

- If Bellman's Principle fails for a problem, does that mean it cannot be solved?



Answer 10

- Not necessarily: it means DP is not applicable.
- Other paradigms may work: greedy, branch-and-bound, approximation.
- DP requires optimal substructure and overlapping subproblems.



Check Question 11: True DP?

- Imagine you design a DP algorithm but forget to store solutions to subproblems.
- Does it still count as dynamic programming?



Answer 11

- No: without storage it's just recursion with repeated recomputation (exponential).
- Memoisation/tabulation is essential: storage distinguishes DP from recursion.

Answer 2 (Table): Fibonacci f(6)

Call	Expansion
$f(6)$	$f(5) + f(4)$
$f(5)$	$f(4) + f(3)$
$f(4)$	$f(3) + f(2)$
$f(3)$	$f(2) + f(1)$

Answer 3 (Table): Shortest Path Example

Node i	Computation	f _i
9	Base case	0
8	$t_{89} + f_9 = 10 + 0$	10
7	$t_{79} + f_9 = 3 + 0$	3
6	$\min(t_{68}+f_8, t_{69}+f_9)$	15



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Robots: a Taxonomy

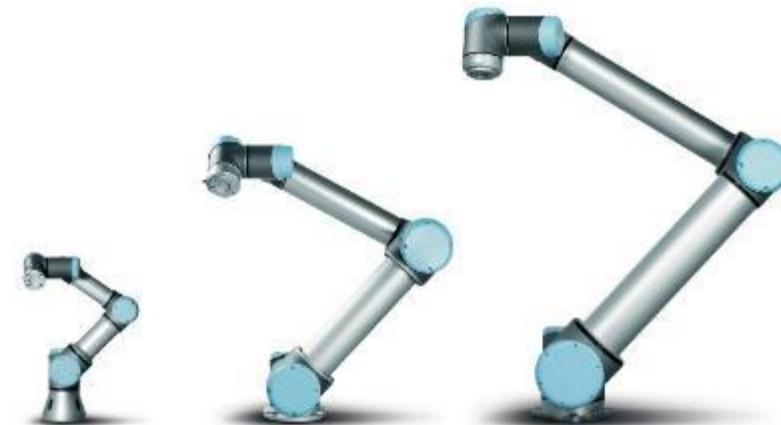
Luigi Palopoli – DISI – UNITN 2022



Some taxonomy and terminology

- A first important distinction is between:

MANIPULATORS: robots with a fixed base



MOBILE ROBOTS: robots with a mobile base





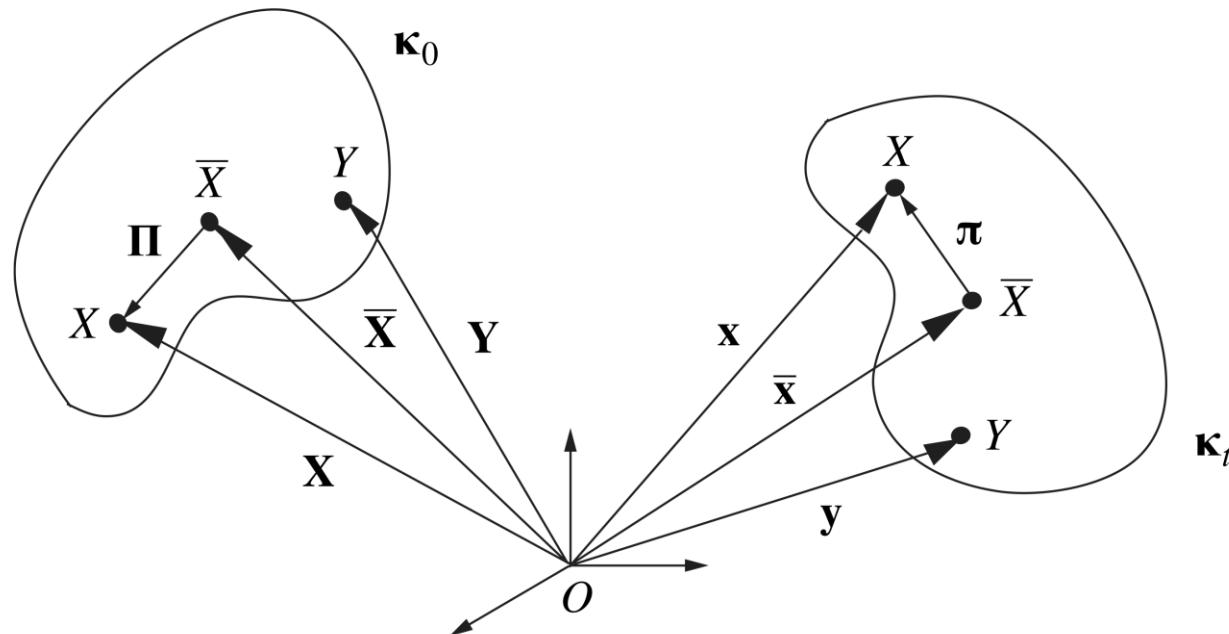
Robotic Manipulators

- A robotic manipulator consists of
 - An arm ensuring mobility
 - A wrist located at the end of the arm enabling dexterous operation
 - An end-effector executing the robot's tasks
- From the geometric point of view, the manipulator is a sequence of rigid bodies (links), connected by mechanical articulations (joints).



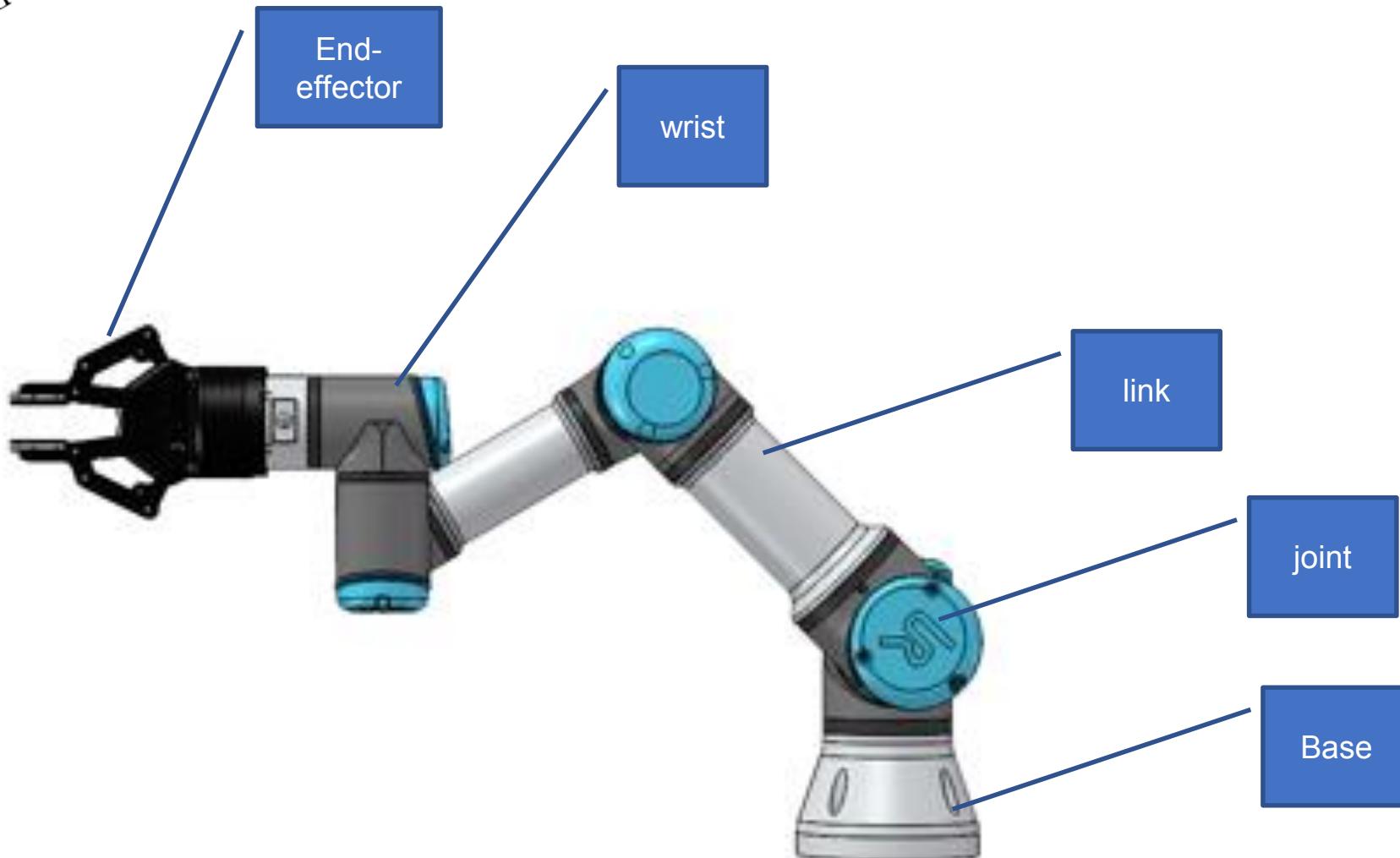
Rigid body

- A rigid body is a collection of material points such that their mutual distance remains constant





Manipulator example



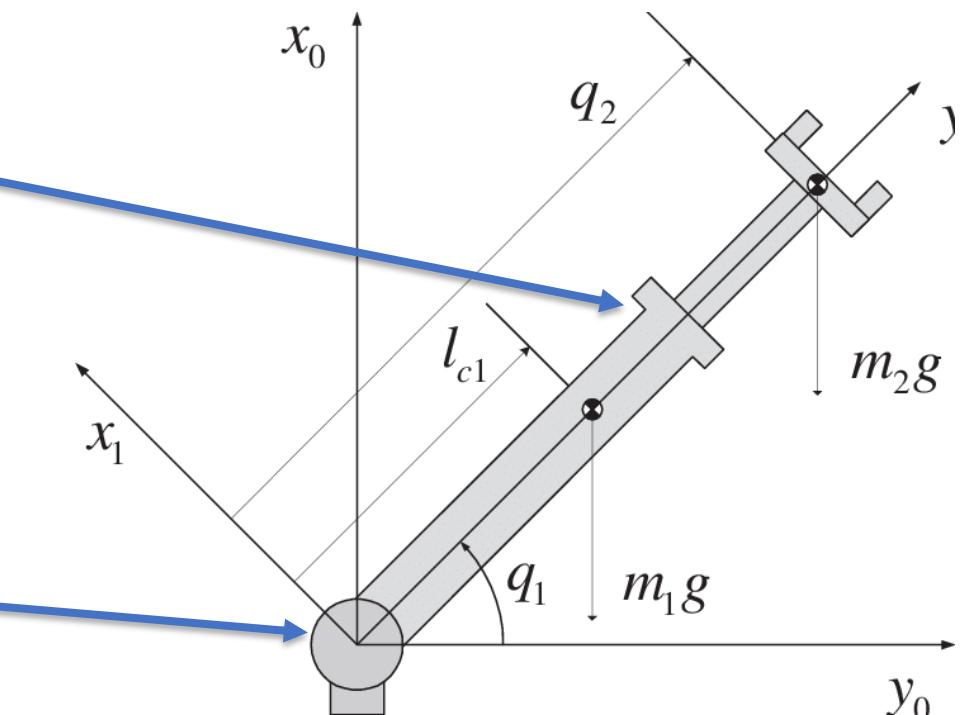


Joints

- Joints provide the structure with its necessary mobility
- Joints can be of two types

Prismatic joints: enable a relative translational motion between the links

Revolute joints: enable a relative rotational motion between the links





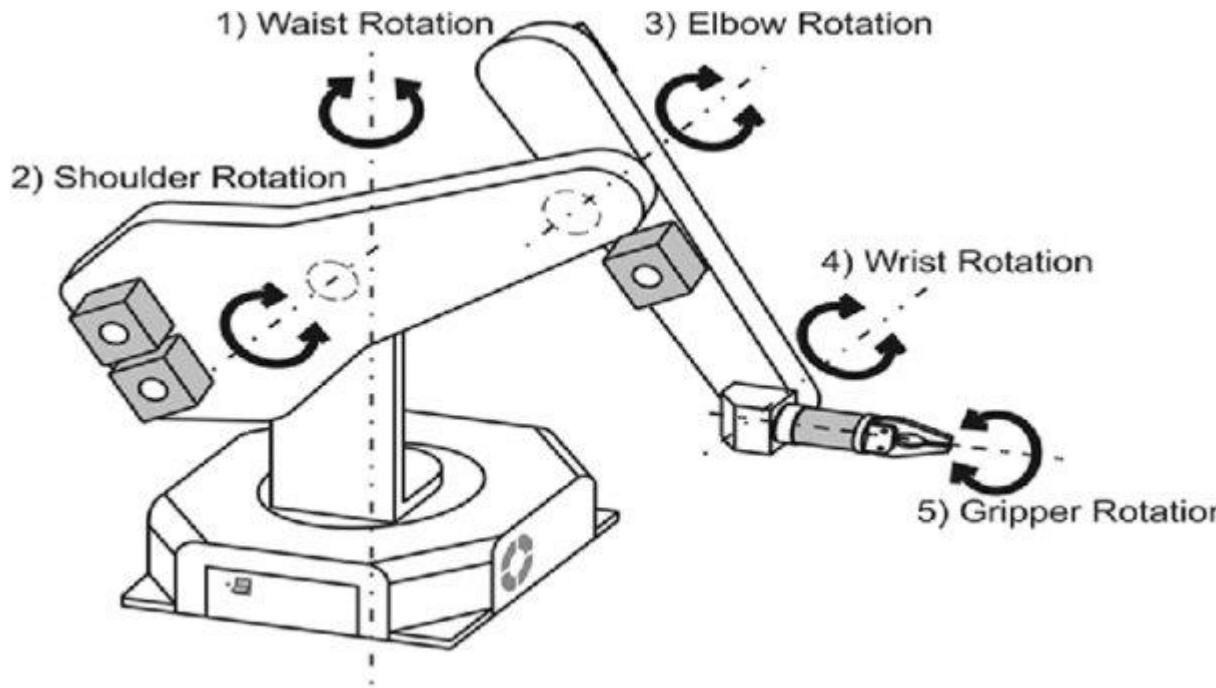
A few definitions

- **Dexterity** can be defined as a robot's ability to cope with a variety of objects and actions.
 - how robots can interact and handle objects and take necessary actions on the objects.
- **Stiffness:** ability of a body to resist deformation
 - In formal terms we can see the stiffness as the amount of force required to induce a motion along a DoF



Degrees of freedom

- In a mechanical structure a degree of freedom (DoF) defines a specific mode in which the robot can move
- Typically each joint is endowed with an actuator (e.g., a brushless, or a linear motor) and provides the structure with one degree of freedom
- The more the degrees of freedom the more flexible is the machine



- The workspace is the area that the end effector can reach
- Its structure very much depends on the number and on the type of the joints



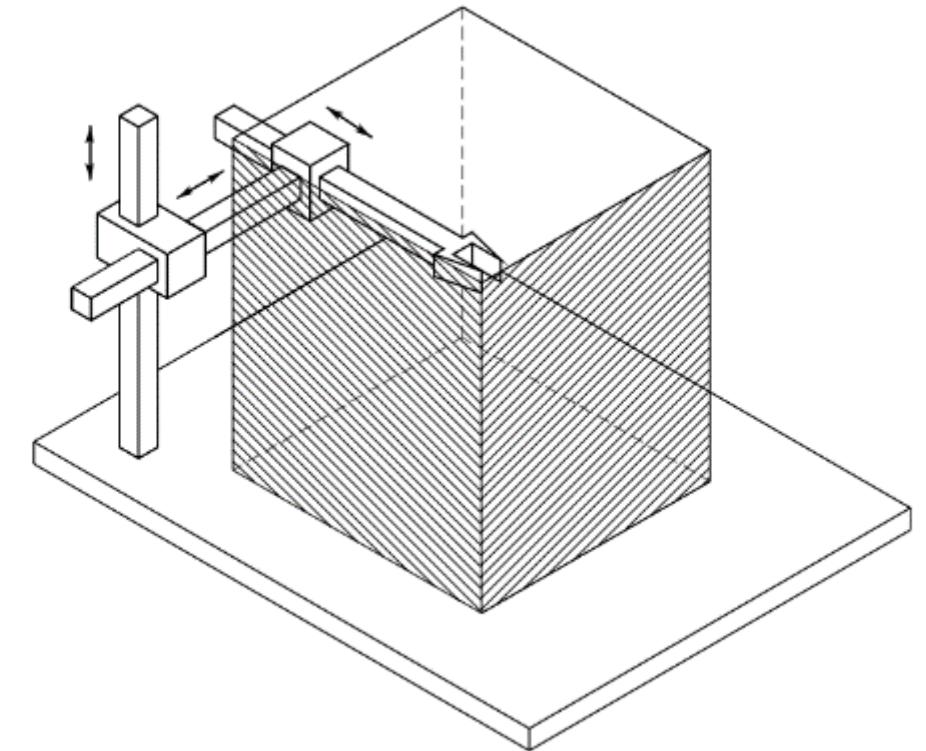
Different type of manipulators

- According to the structure of the different DoF we can distinguish between different types of manipulators
 - Cartesian
 - Cylindrical
 - Spherical
 - SCARA
 - Anthropomorphic



Cartesian

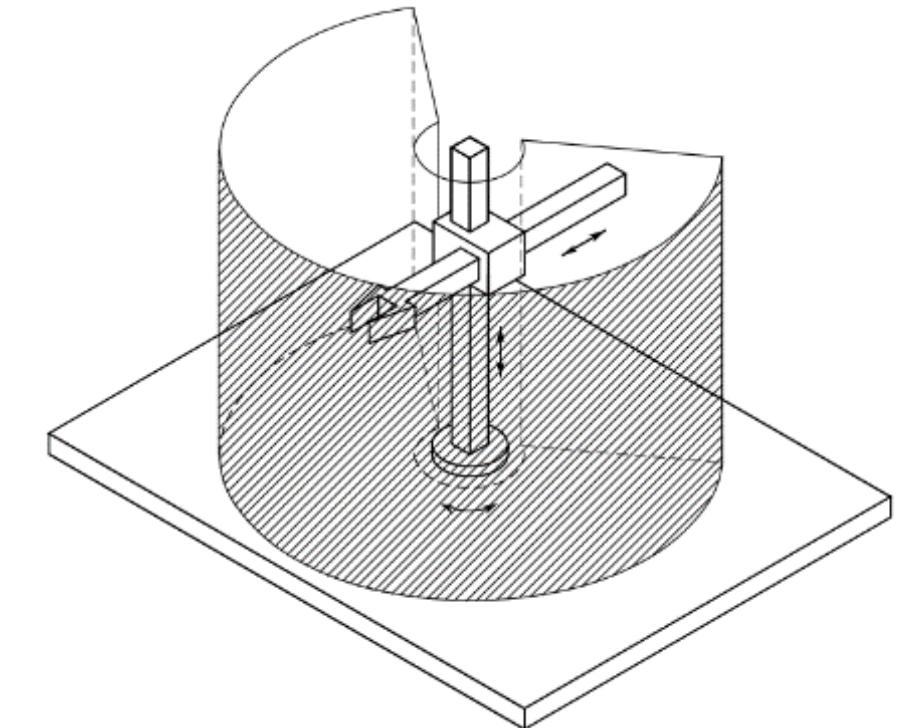
- Cartesian robots are characterised by three prismatic joints with three mutually orthogonal axis
- The workspace is a parallelepiped
- It operates with good mechanical stiffness and accuracy everywhere in its workspace
- The exclusive presence of prismatic joints reduces the structure's dexterity





Cylindrical Manipulators

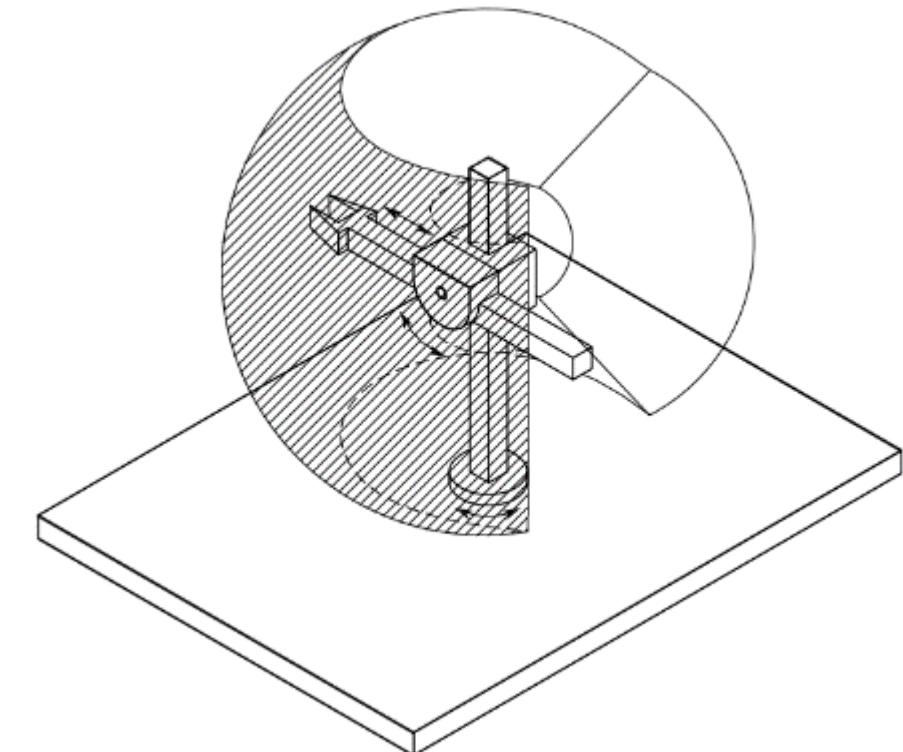
- In a cylindrical manipulators one of the joints is replaced by a revolute joints.
- Wrist accuracy decreases with horizontal stroke
- Good stiffness
- The workspace is a portion of cylinder
- This type of machine is primarily used to move heavy loads within cylindric cavities





Spherical Manipulators

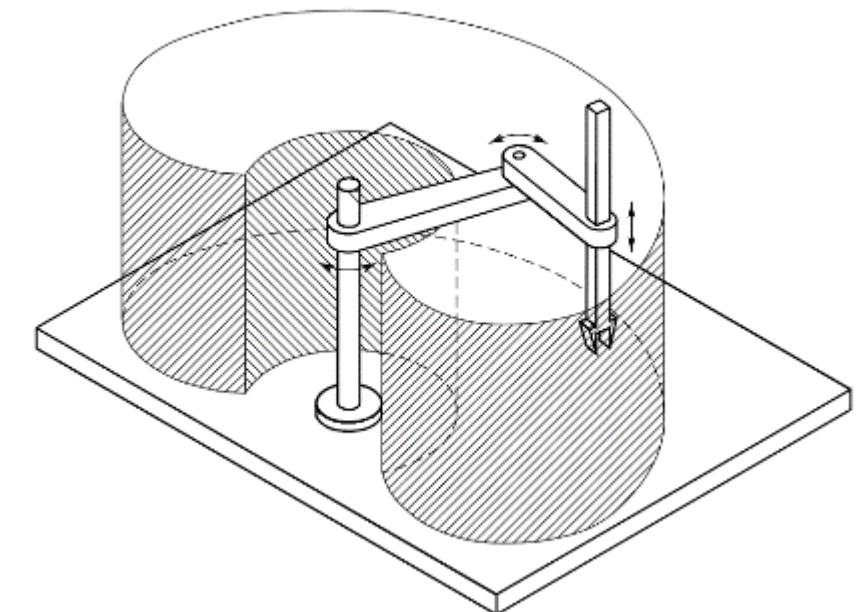
- For a spherical manipulator, we replace two prismatic joints with revolute joints
- The Workspace is a portion of a hollow sphere
- The manipulator has a reduced accuracy when the radial stroke increases
- Primarily used for machining





SCARA manipulators

- For a SCARA (**S**elective **C**ompliance **A**ssembly **R**obot **A**rm), we have two revolute joints and one prismatic joint
- The axis of motion of the SCARA are parallel
- High stiffness to vertical loads, compliance to horizontal loads
- Well suited to vertical assembly tasks and for the manipulation of small objects
- Positioning accuracy decreases with the distance from the first axis

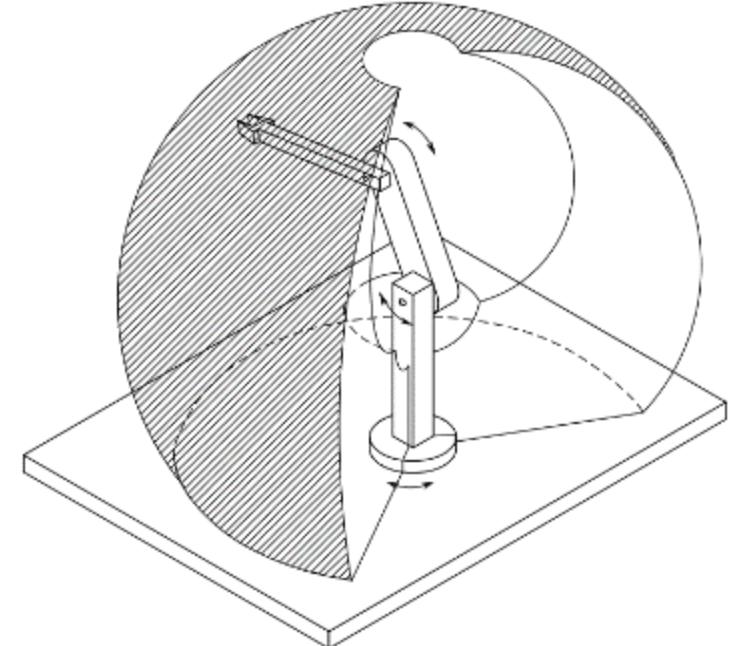




Anthropomorphic manipulators

- In the anthropomorphic manipulator we have three revolute joints.
- The axis of the second and of the third joint are orthogonal to the axis of the first
- The similarity with the human arm is evident
- The second joint is said shoulder and the third is said elbow

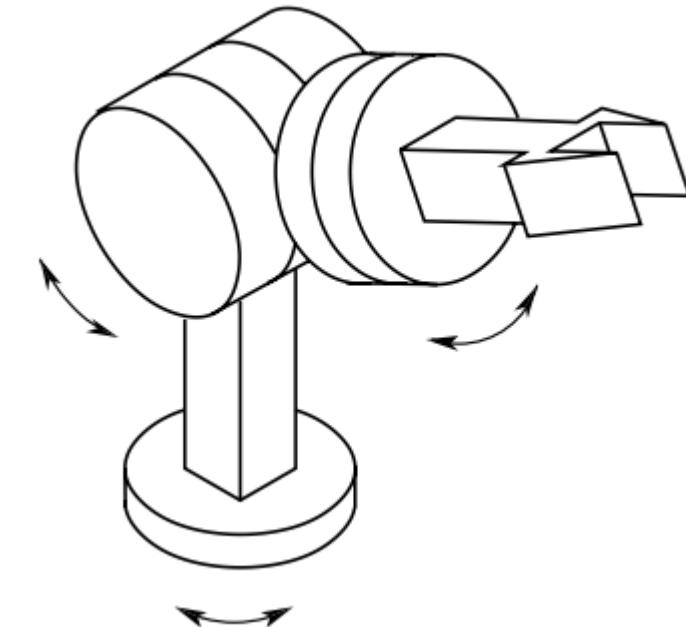
- The three revolute joints maximise dexterity
- The wrist position accuracy is different in the workspace
- This robot is by far the most widespread in industry (59% of the installation)





Wrist

- All different manipulators have an end-effector attached to a wrist
- The end-effector is usually a gripper but can be a different thing according to the task
 - E.g., a welding gun, a spray gun, a screwdriver...
- The configuration of the wrist that maximises dexterity is spherical (three revolute joints)
- Each joint of a wrist provide an additional DoF





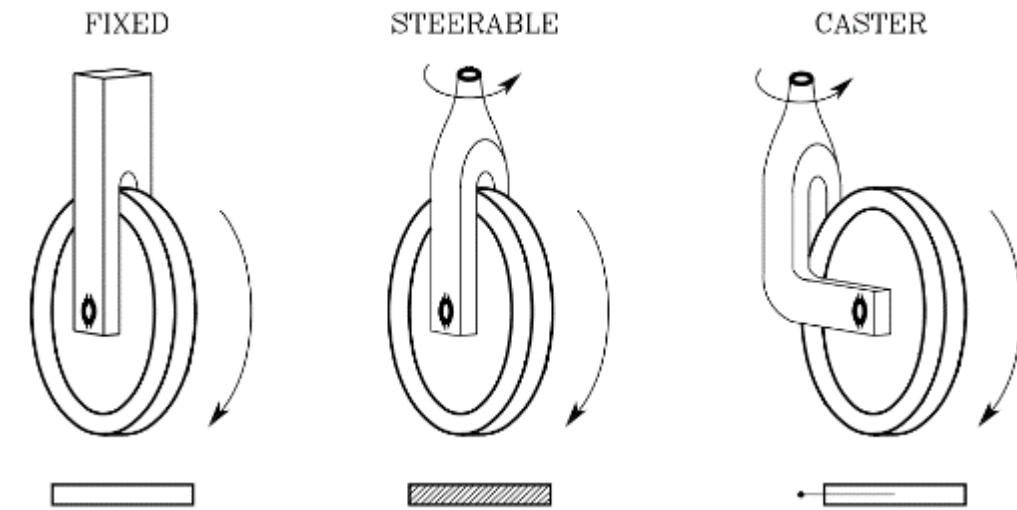
Mobil robot

- A mobile robot is characterised by a mobile base
- We can have two types of mobile robots
 - **Wheeled Robots:** A rigid body (chassis) and a system of wheels that provide motion with respect to the ground
 - Can be connected to a system of trailers connected by revolute joints
 - **Legged Robots:** Multiple rigid bodies connected through revolute (and sometimes) prismatic joints
 - Some of the links form lower limbs and feet that stay in contact with the ground and provide locomotion.
 - We will not consider legged robots in this course



Wheel types

- Wheeled robots use three types of wheels
 - **Fixed**: can only rotate about an axis passing through the centre and orthogonal to the wheel plane
 - **Steerable**: has two axis of rotation: the standard axis typical of all wheels, a vertical axis passing through the centre
 - **Caster wheel**: a caster wheel is similar to a steerable wheel, but its vertical axis has an offset from the centre of the wheel.
 - The wheel swivels around when bending and automatically aligns itself with the direction of motion of the chassis

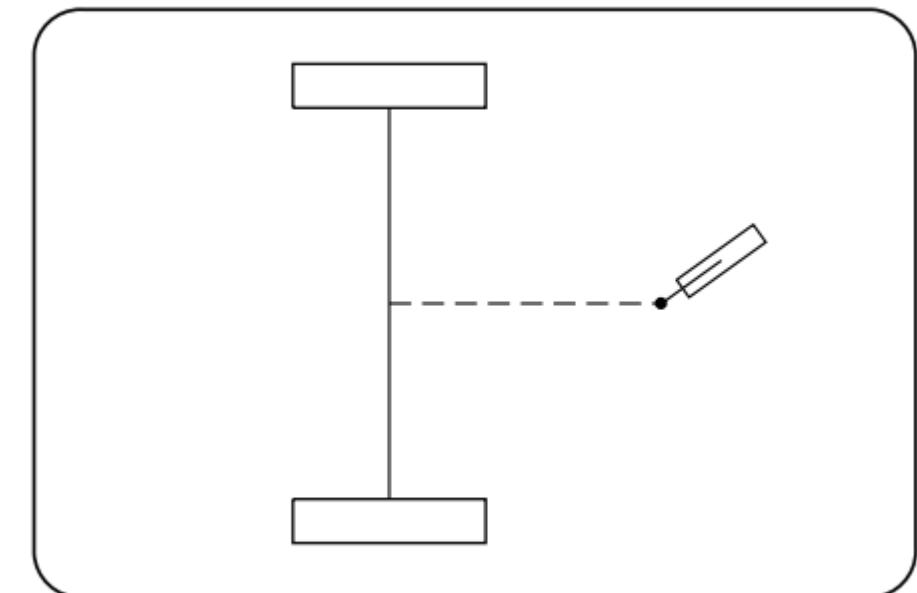


By combining different types of wheels we can obtain different types of kinematic structures



Differential drive vehicle

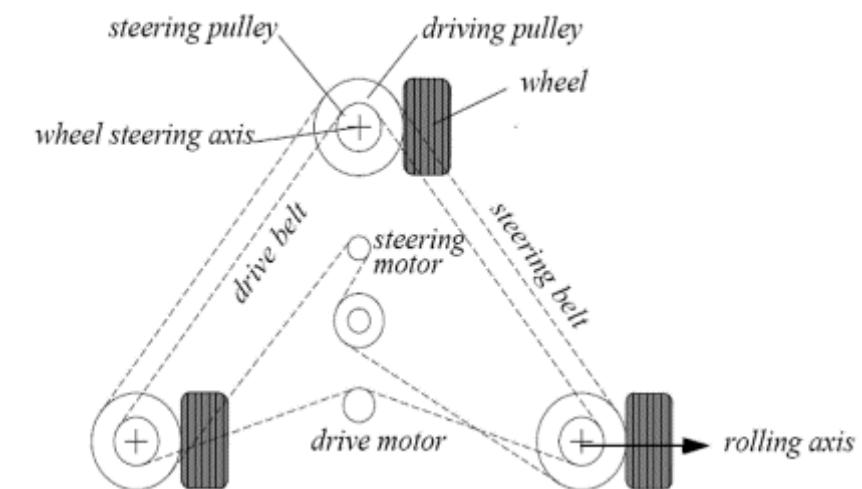
- In a differential drive robot there are
 - Two actuated fixed wheels with the same axis
 - A caster wheel to keep the robot statically balanced
 - The robot can rotate by applying a different velocity to the wheels, or move straight applying the same angular velocity
 - It can rotate on the spot by setting the two speeds to opposite values





Synchro Drive robots

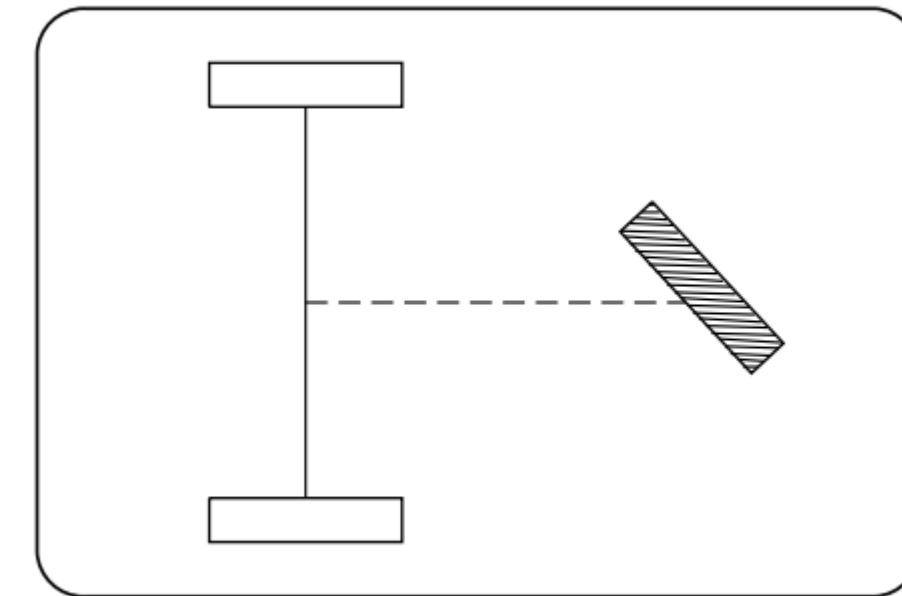
- In the synchro drive configuration there are three steerable wheels connected by a chain
- There are two motors: one rotates the three wheels “synchronously” and the other transmits the motion (synchronously) to all of them
- It can do the same kinematics of the differential drive but it needs only one motor to move straight





Tricycle robots

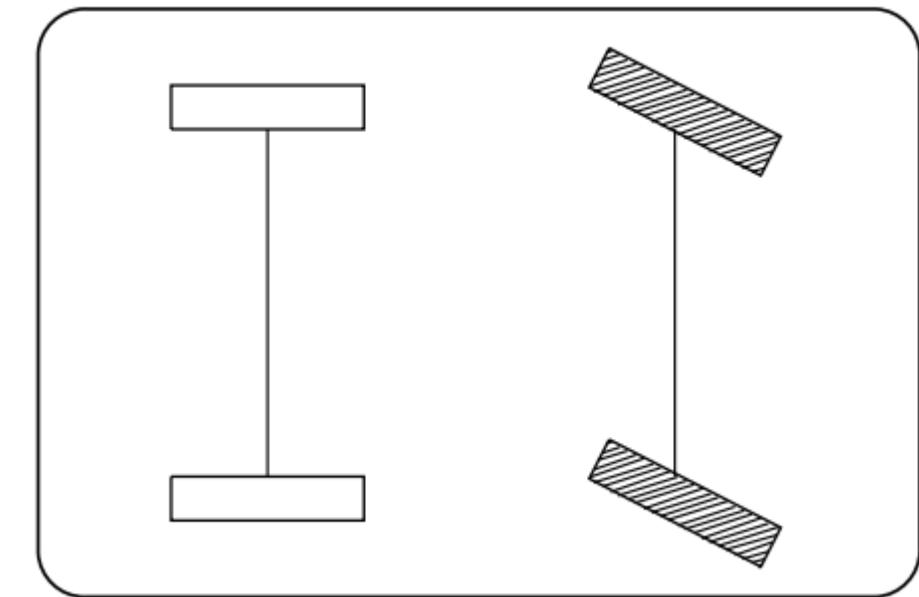
- In a tricycle we have two wheels that are fixed and actuated by a motor
- The third wheel is steerable and its rotation around the vertical axis is governed by another motor
- It is also possible that the two motors operate both on the steering wheel to turn it and to secure locomotion





Carlike

- The carlike robot is very similar to a tricycle except that it has two turning wheels
- Again the traction can be on the fixed wheels (rear-wheel traction) or on the front wheel (front-wheel traction)
- Contrary to a differential drive robot, a car-like (and a tricycle) cannot turn on the spot and have a limited curvature radius
- Besides, the carlike has an interesting problem





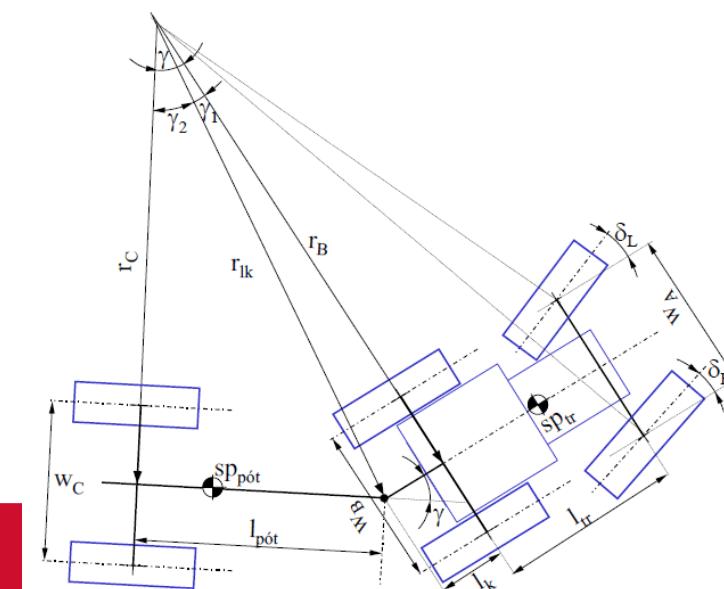
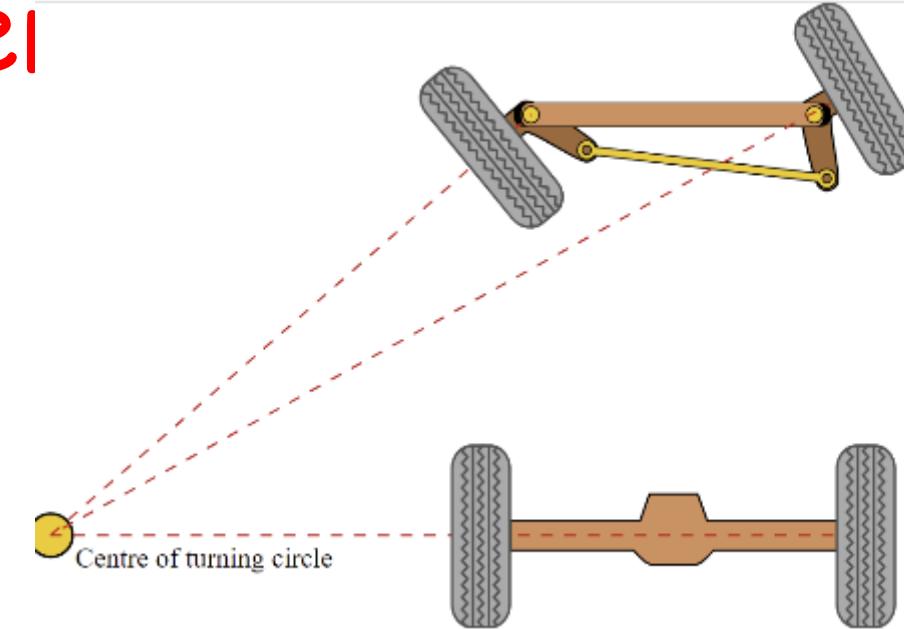
Ackerman steering

- Back in 1758 Erasmus Darwin was driving his horse-drawn carriage.
- The two front wheels turned of the same angle
- One of the wheels (the external one) slipped sideways and it caused the carriage to tip over his driver
- The solution was devised by Lankensperger in Munich and patented by his English agent (Ackerman) in London
- The idea is very simple: the two wheels have to follow the same circular trajectory



Ackerman Steering

- In order to follow the same circle the internal wheel has to turn more than the external one
- This is solved, in modern cars, by the geometry of the axle
- The Ackerman steering can be generalised to the case of trailers.





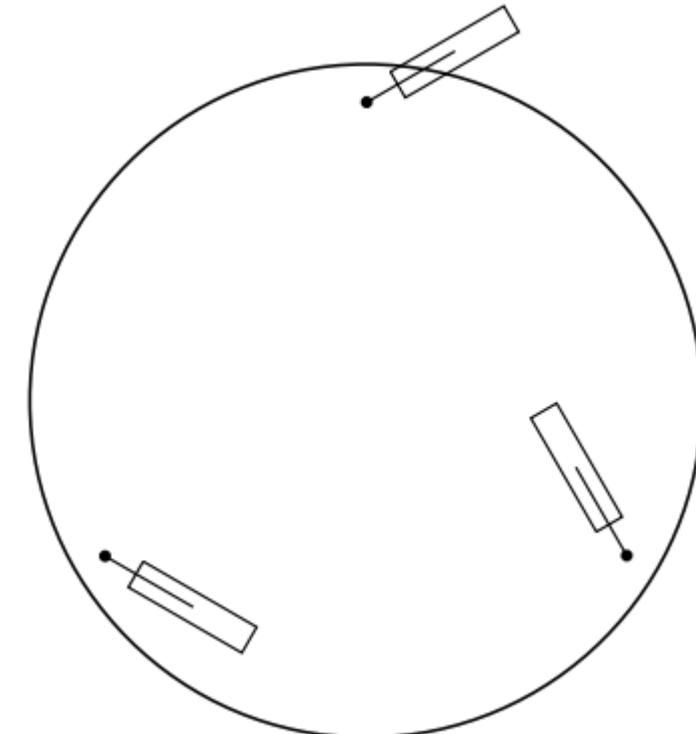
Constraints

- Contrary to manipulators mobile robots do not usually have a limited workspace
 - They can reach any location in the Euclidean space
- However they do have motion constraints
 - A Differential Drive robot is quite flexible, but it cannot move sideways along the axis connecting the actuated wheels
 - This is called a nonholonomic constraint
 - In addition, a carlike has a limited curvature radius



An exception

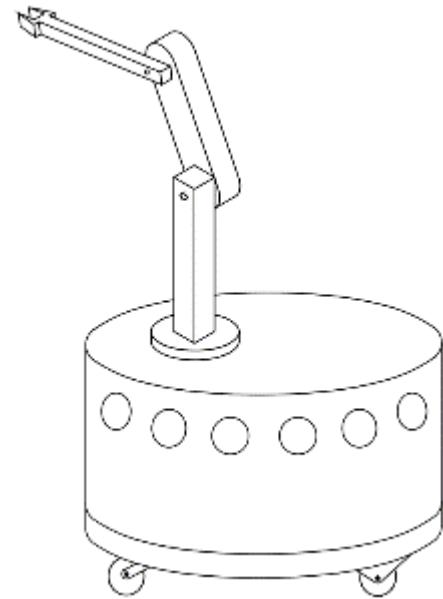
- An exception is an omnibot
- An omnibot is characterised by three autonomously actuated caster-wheels (usually in symmetric positions)
- It can move in *any direction*





Additional types of robots

- Clearly, we can mount an anthropomorphic manipulator and a differential drive mobile base in order to improve its working abilities
- We can improve the dexterity by additional degrees of freedom (7 for the Kuka LWR)





More sophisticated end-effectors

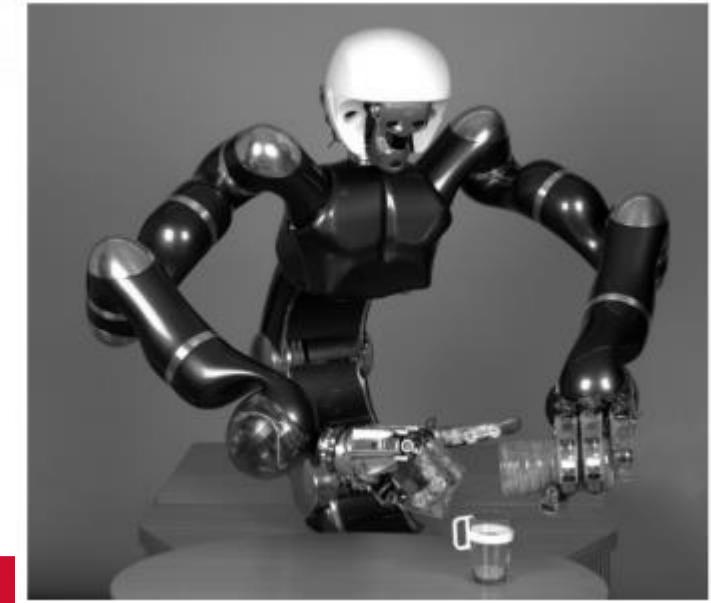
- A very active research area has been on the development of sophisticated anthropomorphic hands
- Such hands can be used for
 - rehabilitation and assistive purposes (e.g., as a replacement for a severed human hands)
 - Creation of sophisticated anthropomorphic robots





A new frontier: Humanoid Robots

- Putting together several types of robots we can create humanoid robots
-but this out of the scope of this course





Modelling: kinematics of manipulators

- **Kinematics** is about the motion of a robotic structure with respect to a generic frame
 - For a manipulator, given the joint position, find position and orientation of the end effector
- **Differential kinematics:**
 - For a manipulator: given the joint positions and motion (and their velocity) find the velocity of the end effector
- **Forward Kinematics:**
 - Find the end effector motion as a function of the joint motion
- **Inverse Kinematics**
 - Find the joint motion as a function of the “desired” end-effector motion and configuration



Modelling: kinematics of mobile robots

- Mobile robots:
 - The robot's kinematics requires a correct understanding of the motion constraints
 - The kinematic model translates the instantaneous motion of the motors into an instantaneous motion of the robot as a whole, which accounts for the robot's current configuration and for its motion constraints.

Mobile Robots

Robot Planning and its Applications

University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)





Outline

Overview

Kinematic Constraints

Manipulators vs Mobile Robots

- ▶ RObots arms are fixed to the ground and are usually composed of a single chain of actuated links
- ▶ The motion of mobile robots is defined through **rolling** and **sliding effects** at the wheel-ground contact points
- ▶ Both have a forward and an inverse kinematics
- ▶ However, for mobile robots the encoder values at the link do not map to a unique pose

Manipulators vs Mobile Robots

I can have different configurations
given the value of the joints

- ▶ Mobile robots can move unbound with respect to their environment
 - ▶ There is no direct (=instantaneous) way to measure the robot's position
 - ▶ Position must be integrated over time, depends on path taken
 - ▶ Leads to inaccuracies of the position (motion) estimate
- ▶ Understanding mobile robot motion starts with understanding wheel constraints placed on the robot's mobility

Non Holonomic Constraints

- ▶ Mobile robots are (typically) non-holonomic.
- ▶ In essence, we cannot integrate the differential equations describing the motion to the final position
- ▶ For a robotic arm this is possible and the final configuration is a function of the travelled “distance” on each joint.
- ▶ For a **non-holonomic system**, the total rotation of the wheels is not sufficient to find the final configuration. We need to know how the movement was executed in time.

$$S_{1R} = S_{2R}, S_{1L} = S_{2L}$$
$$x_1 \neq x_2, y_1 \neq y_2$$

Forward and Inverse Kinematics

- ▶ Forward Kinematics: Transformation from joint to physical space
- ▶ Inverse Kinematics: Transformation from physical to joint space
 - ▶ Needed for motion control
- ▶ Due to the presence of non-holonomic constraints, for mobile robots we will only deal with differential (inverse) kinematics



Outline

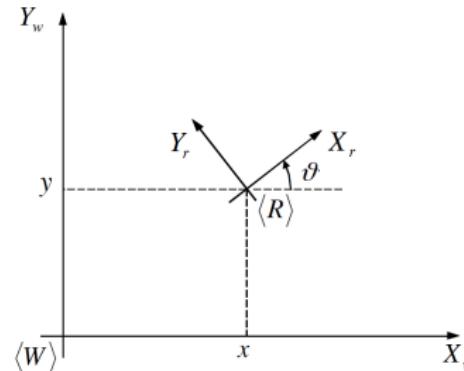
Overview

Kinematic Constraints

Wheeled Mobile Robots

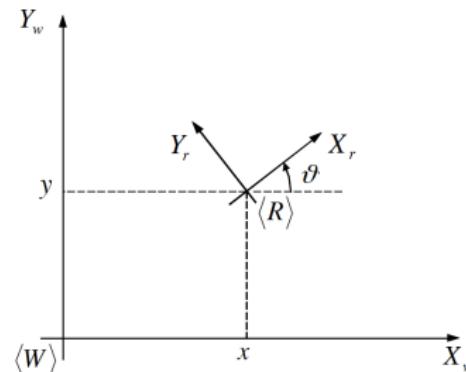
- ▶ Consider a mechanical system whose *configuration* $q \in \mathbb{R}^n$ is described by a vector of *generalised coordinates*, where the space of all possible robot configurations coincides with \mathbb{R}^n .
- ▶ The set of coordinates is *generalised* if they *uniquely* define any possible configuration of the system relative to the reference configuration.
- ▶ The motion of the system, that is represented by the evolution of q over time, may be subject to constraints.
- ▶ Another important characteristic is the *accessible* space, i.e., the workspace of the mobile robot.
- ▶ In the plane, we usually consider the space of the configurations of the vehicle (neglecting trailers) be \mathbb{R}^3 : the position of a point in the plane + the orientation of the rigid body.

An intuitive view



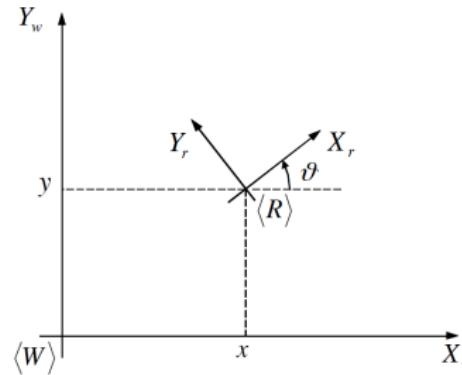
- ▶ Consider the two frames in the figure.
- ▶ Express the position of the frame $\langle R \rangle$ w.r.t. the $\langle W \rangle$ and derive the minimal set of coordinates that determine the relative configuration.

An intuitive view



$$T_R^W = \begin{bmatrix} R_z(\theta) & \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \\ 0 & 1 \end{bmatrix}$$

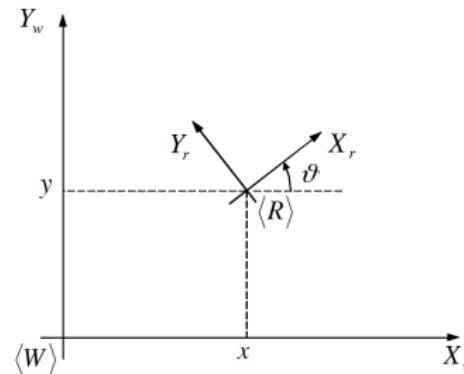
An intuitive view



Differential kinematics.
we can know values at
a certain instant

- ▶ Express the kinematic of the frame $\langle R \rangle$ in terms of the time derivative \dot{x} , \dot{y} and $\dot{\theta}$, assuming the frame $\langle R \rangle$ is moving with a generic linear velocity v expressed in the moving frame and a generic angular velocity $\omega_{\hat{Z}_W} = \omega_{\hat{Z}_R}$.

An intuitive view



$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} [c_{\theta} & -s_{\theta}] \\ [s_{\theta} & c_{\theta}] \\ \omega \hat{Z}_w \end{bmatrix} v$$

An intuitive view

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} [c_{\theta} & -s_{\theta}] \\ [s_{\theta} & c_{\theta}] \\ \omega \hat{z}_w \end{bmatrix} v$$

- ▶ What happens if the reference frame $\langle R \rangle$ is attached to a cart constrained to move on a railway track oriented along the \hat{X}_R axis?
- ▶ What happens if the velocity v is always oriented along the \hat{X}_R axis?

Constraints

- ▶ Generally speaking, the two examples above express motion constraints.
- ▶ Constraints expressed by inequalities are called *unilateral*
↳ Defined by inequalities constraints.
- ▶ Constraints expressed by equalities are called *bilateral* constraints. We will focus only on bilateral constraints

WMR – Bilateral Constraints

- ▶ Consider that a subset of coordinates q_i , with $i = 1, \dots, r \leq n$, is subjected to r bilateral constraints.
- ▶ Constraints that are only function of the positions can be expressed as

$$h(q, t) = 0$$

generalized coordinates that allow
(vector) to identify uniquely
the configuration of
the robot

- ▶ A constraint that depends explicitly on time is called **rheonomic**.
- ▶ We will focus only on **scleronomic** constraints, i.e., time invariant constraints, like

$$h(q) = 0$$

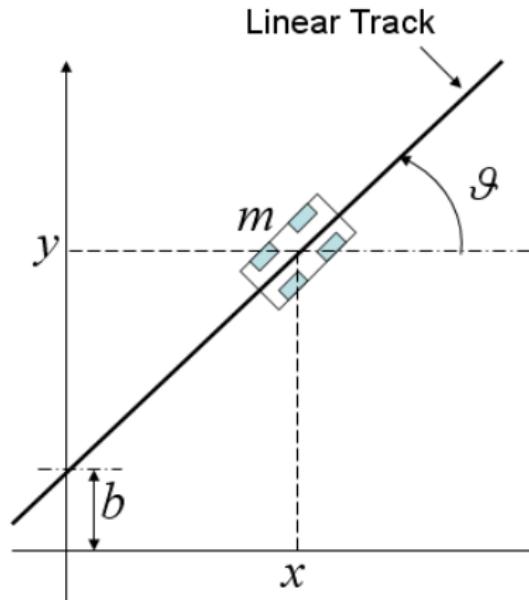
WMR – Bilateral, Scleronomic Constraints

- ▶ If r constraints are defined for the system, $h(q)$ is a vector function with r entries, i.e., $h(q) = [h_1(q), \dots, h_r(q)]^T$, one for each constraint.
- ▶ Such constraints, are called *holonomic* or *integrable*.
- ▶ The effect of holonomic constraints is to reduce the space of accessible configurations to a subset of \mathbb{R}^n with dimension \mathbb{R}^{n-r} (recall the example of the cart).
- ▶ A mechanical system for which all the constraints can be expressed in the holonomic form is called *holonomic*.

WMR – Holonomic Constraints

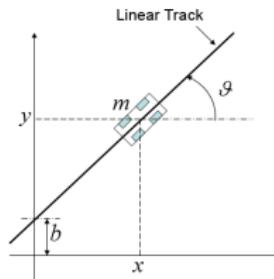
- ▶ For holonomic constraints, the implicit function theorem, or *Dini theorem*, can be used to express r generalised coordinates as a function of the remaining $n - r$, so as to actually eliminate them from the formulation of the problem.
- ▶ Problem: such procedure may introduce singularities.
- ▶ Solution: replace the original generalised coordinates with a reduced set of $n - r$ new coordinates that are directly defined on the accessible subspace.
- ▶ Holonomic constraints are generally the result of mechanical interconnections between the various bodies of the system, prismatic and revolute joints used in robot manipulators are a typical source of holonomic constraints.
 - ▶ The joint variables are an example of reduced sets of coordinates.

Example: Cart Constrained on a Linear Track Railway



Generalised coordinates $\mathbf{q} = [x, y, \theta]^T$.

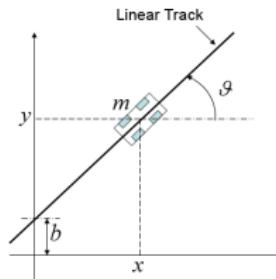
Example: Cart Constrained on a Linear Track



- ▶ First constraint: the linear track has a mathematical description given by $y = x \tan \theta_b + b$.

$$h_1(q) = y - x \tan \theta_b - b = 0$$

Example: Cart Constrained on a Linear Track



- ▶ Second constraint: the equality between the attitude angle of the track and the attitude angle of the cart.
- ▶

$$h_2(q) = \theta - \theta_b = 0$$

WMR – Kinematic Constraints

- ▶ Constraints that involve generalised coordinates and velocities

$$c(\dot{q}, q) = 0,$$

where $c(\dot{q}, q)$ is again a vector function with r entries, are called *kinematic* constraints.

- ▶ Such constraints limit the *instantaneous admissible* motion of the mechanical system by reducing the set of generalised velocities that can be attained at each configuration (recall the example with the constrained v).
- ▶ Kinematic constraints are generally linear in the generalised velocities and hence they can be expressed as $c_i(q)\dot{q} = 0$ and can be expressed in a more compact *Pfaffian form*

$$A(q)\dot{q} = 0$$

WMR – Kinematic Constrained Model

- ▶ $c_i(q)\dot{q}$, $i = 1, \dots, r$, are assumed to be smooth as well as linearly independent, i.e., $A(q)$ is of full rank.
- ▶ The kinematic constraints of the Pfaffian form can be obtained from direct time derivation of holonomic constraints, i.e.,

$$h_i(q) = 0 \Rightarrow \frac{dh_i(q)}{dt} = \frac{\partial h_i(q)}{\partial q} \dot{q} = 0$$

- ▶ For a holonomic mechanical system, *r kinematic constraints correspond to r holonomic constraints*, obtained by integration of the kinematic constraints.

It's not possible
the opposite
↑

WMR – Kinematic Constrained Model

- ▶ Notice that the velocities that belong to the null space of $A(q)$ are feasible, since $A(q)\dot{q} = 0$.
- ▶ The *constrained kinematic model* is defined by

$$\dot{q} = G(q)u,$$

where $G(q)$ is a basis of $\mathcal{N}(A(q))$.

- ▶ The kinematic model expresses velocity that are compatible with the constraints, indeed

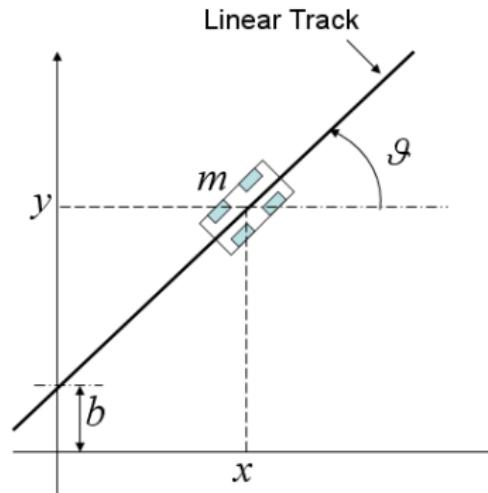
$$A(q)\dot{q} = A(q)G(q)u = 0$$

- ▶ The columns $g_i(q)$ of $G(q)$ are called *input vector fields*.

WMR – Kinematic Constrained Model Remarks

- ▶ $\mathbf{q} \in \mathbb{R}^n$ and $\mathbf{u} \in \mathbb{R}^m$, where $m = n - r$.
- ▶ The kinematic model here derived is *driftless*, because one has $\dot{\mathbf{q}} = 0$ when $\mathbf{u} = 0$.

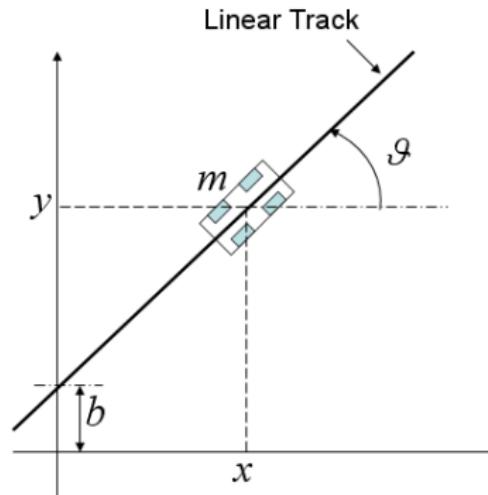
Example: Cart Constrained on a Linear Track



- ▶ First constraint: the linear track has a mathematical description given by $y = x \tan \theta_b + b$.

$$\begin{cases} h_1(q) = y - x \tan \theta_b - b = 0 \\ \frac{d(h_1(q))}{dt} = \dot{y} - \dot{x} \tan \theta_b = 0 \end{cases}$$

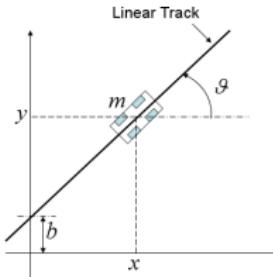
Example: Cart Constrained on a Linear Track



- ▶ Second constraint: the equality between the attitude angle of the track and the attitude angle of the wagon.

$$\begin{cases} h_2(q) = \theta - \theta_b = 0 \\ \frac{d(h_2(q))}{dt} = \dot{\theta} = 0 \end{cases}$$

Example: Cart Constrained on a Linear Track



In Pfaffian form, we have

$$\begin{cases} \frac{d(h_1(q))}{dt} = \dot{y} - \dot{x} \tan \theta_b = 0 \\ \frac{d(h_2(q))}{dt} = \dot{\theta} = 0 \end{cases} \Rightarrow A(q)\dot{q} = \begin{bmatrix} \sin \theta_b & -\cos \theta_b & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = 0$$

Notice that the rank number of the matrix $A(q)$ is always equal to two.

Therefore, the dimension of the null space will be 1.

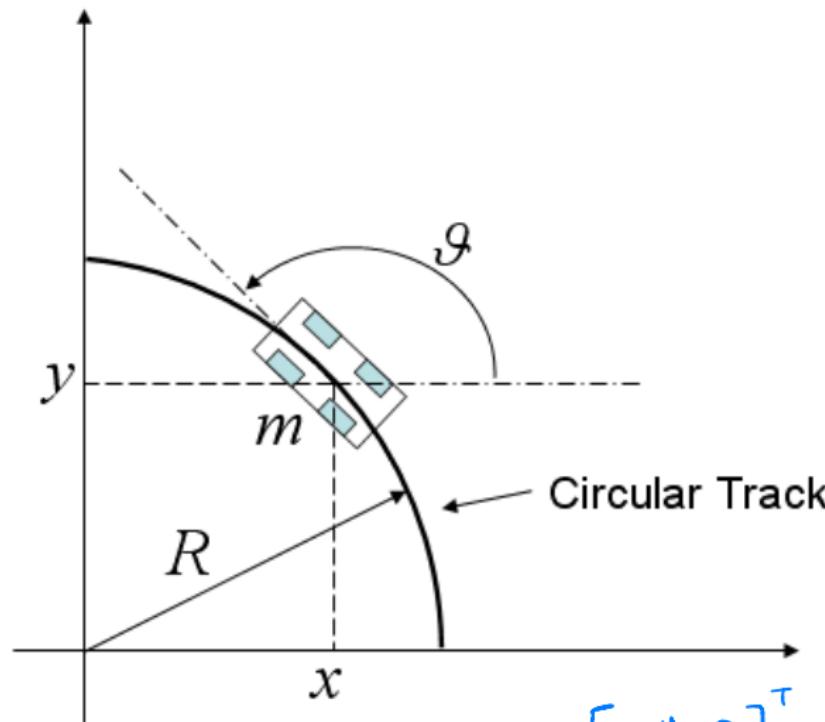
Example: Cart Constrained on a Linear Track

- ▶ The kinematic model is given by

$$\dot{q} = G(q)u = \begin{bmatrix} \cos \theta_b \\ \sin \theta_b \\ 0 \end{bmatrix} u$$

- ▶ The holonomic constraint limits the accessibility of the cart on the plane.
- ▶ Only one variable is independent, i.e., the position on the track
 - ▶ The other two variables may be removed using the implicit function theorem or by changing the generalised coordinates.
- ▶ A similar result has been obtained using the intuitive approach in the first slides.

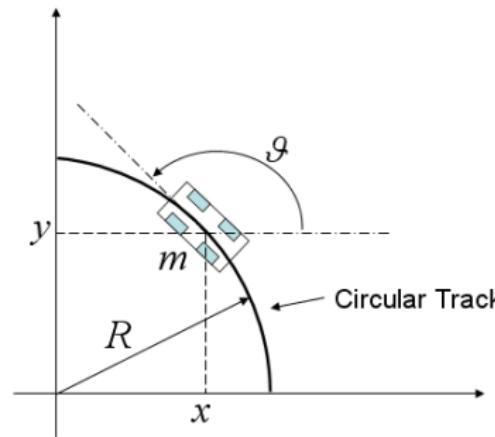
Example: Cart – Circular Track



$$q = [x, y, \theta]^T$$

Generalised coordinates $q = [x, y, \theta]^T$

Example: Cart – Circular Track

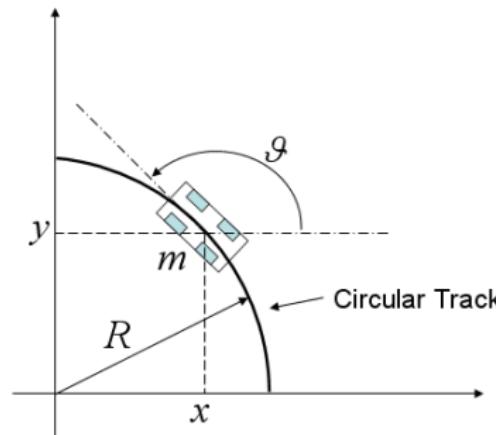


- ▶ First constraint: the mathematical description of the circular track is given by $x^2 + y^2 = R^2$.

$$\begin{cases} h_1(q) = x^2 + y^2 - R^2 = 0 \\ \frac{d(h_1(q))}{dt} = 2x\dot{x} + 2y\dot{y} = 0 \end{cases}$$

The velocity
is tangent to
the radius

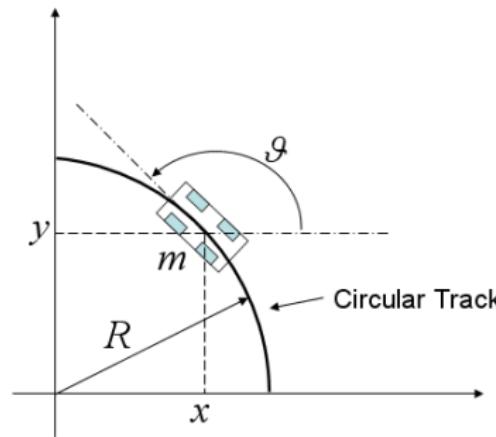
Example: Cart – Circular Track



- ▶ Second constraint: the cart orientation is always tangent to the track.

$$\begin{cases} h_2(q) = \theta - \arctan\left(\frac{y}{x}\right) - \frac{\pi}{2} = 0 \\ \frac{d(h_2(q))}{dt} = \dot{\theta} + \frac{y}{R^2}\dot{x} - \frac{x}{R^2}\dot{y} = 0 \end{cases}$$

Example: Cart – Circular Track



- In Pfaffian form, we have

$$\begin{cases} \frac{d(h_1(q))}{dt} = 2x\dot{x} + 2y\dot{y} = 0 \\ \frac{d(h_2(q))}{dt} = \dot{\theta} + \frac{y}{R^2}\dot{x} - \frac{x}{R^2}\dot{y} = 0 \end{cases} \Rightarrow A(q)\dot{q} = \begin{bmatrix} x & y & 0 \\ y & -x & R^2 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = 0$$

Notice that the rank number of the matrix $A(q)$ is always equal to two. Therefore, the dimension of the null space will be 1.

Example: Cart – Circular Track

A possible choice of the kinematic model is given by

$$G(q) = \begin{bmatrix} -y \\ x \\ 1 \end{bmatrix} \rightarrow \dot{q} = \begin{bmatrix} -y \\ x \\ 1 \end{bmatrix} u$$

- ▶ Summarising, this example presents two holonomic constraints, hence only the position on the circular track is of interest.
- ▶ It is worthwhile to note that u is the forward velocity of the cart on the circular track, while the motion is counter-clockwise.

WMR – Nonholonomic Constraints

- ▶ If the kinematic constraints are *not integrable*, there is not any holonomic constraints that generate them. Hence, the constraints are called *nonholonomic* and the system they refer to is called *nonholonomic*.
- ▶ Beyond the mechanical aspects that may generate a holonomic system rather than a nonholonomic one, the main difference between such systems is related to their *mobility*.
- ▶ Indeed, as it is evident from the examples, for an integrable kinematic constraint, i.e., $c_i(q)\dot{q} = 0$, its integral can be written as $h_i(q) = h_i(q_0)$, where q_0 is the initial robot configuration.
- ▶ Since we have imposed a constraint, the motion of the system will be constrained to a *level surface* of the scalar function $h_i(q)$, defined by $h_i(q_0)$ and of dimension $n - 1$.

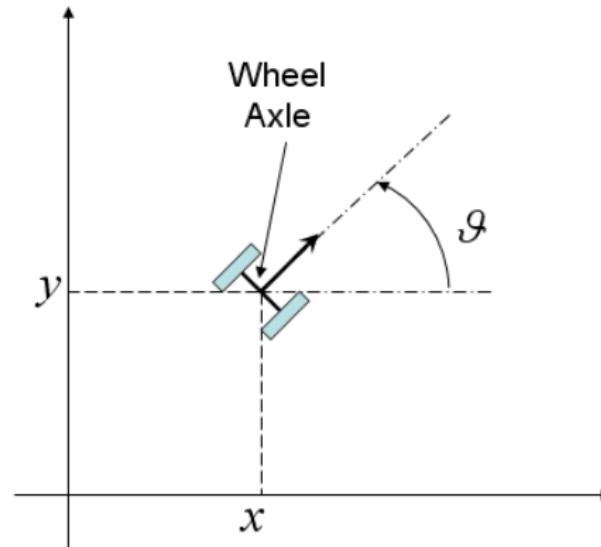
WMR – Nonholonomic Constraints

- ▶ For nonholonomic systems only the velocities are constrained on a subspace of dimension $n - 1$ given by the kinematic constraint. The fact that the constraint is non-integrable means that there is no loss of accessibility to \mathbb{R}^n .
- ▶ In other words, while the number of DOFs decreases to $n - 1$ due to the constraint, the number of generalised coordinates q cannot be reduced, not even locally.
- ▶ This important property can be extended to any number of nonholonomic constraints $r < n$.
- ▶ Summarising, holonomic constraints limit the accessibility of the mechanical structure (e.g., the joints of a manipulator) while nonholonomic constraints impose a limit in the velocity space but preserve the accessibility of the mechanical structure (since nonholonomy constrained feasible \dot{q}).

WMR – Nonholonomic System Models

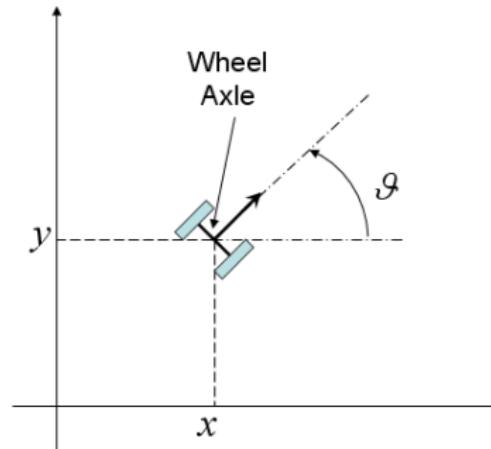
- ▶ Again, the feasible trajectories of a nonholonomic system verify, instantly, the nonholonomic constraints. Hence, only the velocities that belong to the null space of $A(q)$ are feasible, as the case of holonomic ones.
- ▶ The constrained kinematic model represent a very useful tool for the nonholonomic systems analysis.
- ▶ In mobile robotics, indeed, the constructive mechanical simplicity can be derived from nonholonomic constraints.

Example: Unicycle-like Vehicle



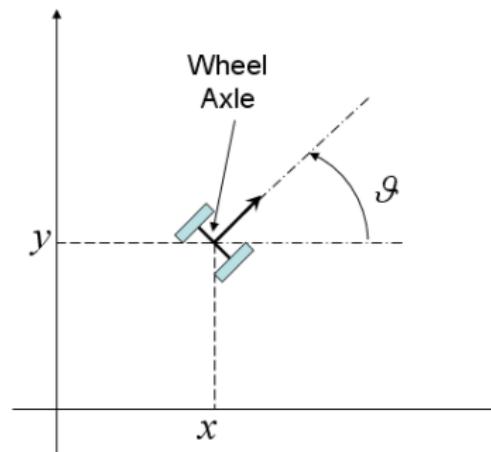
The unicycle-like vehicle is relevant in mobile robotics, since it is widely used in factory automation as well as in academic research.

Example: Unicycle-like Vehicle



- ▶ The generalised coordinate chosen to localise the vehicle on the plan of motion are $q = [x, y, \theta]^T$, where (x, y) represents the midpoint of the traction wheels axle, while θ is the orientation of the vehicle w.r.t. the horizontal *X* axis.
- ▶ The vehicle cannot translate in the direction of the wheel axle.

Example: Unicycle-like Vehicle



- In Pfaffian form, we have

$$c_1(q) = \dot{x} \sin \theta - \dot{y} \cos \theta = 0 \Rightarrow A(q)\dot{q} = \begin{bmatrix} \sin \theta & -\cos \theta & 0 \end{bmatrix} \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = 0$$

Observe the similarity with the first constraint of the linear track.

Example: Unicycle-like Vehicle

- ▶ The kinematic model of the unicycle is then given by

$$G(q) = \begin{bmatrix} \cos \theta & 0 \\ \sin \theta & 0 \\ 0 & 1 \end{bmatrix} \rightarrow \dot{q} = \begin{bmatrix} \cos \theta & 0 \\ \sin \theta & 0 \\ 0 & 1 \end{bmatrix} u$$

- ▶ In this case we have two inputs $u = [u_1, u_2]^T$:

1. u_1 is the forward velocity of the vehicle, since it is directed in the normal direction to the constraint;
2. u_2 is instead the angular velocity, positive counter-clockwise.

Example: Unicycle-like Vehicle

- ▶ From a practical view-point, the vehicle is usually controlled using the rotational velocities (ω_l , ω_r) of the left and right wheel respectively.
- ▶ To obtain the velocities of the kinematic model, the following relations are used

$$\begin{cases} u_1 = \frac{r}{2}(\omega_r + \omega_l) \\ u_2 = \frac{r}{L}(\omega_r - \omega_l) \end{cases}$$

*Angular velocities
of wheel 1 and
wheel 2*

*if $\omega_r = -\omega_l$ (or
viceversa) you rotate
on the spot*

where r is the wheels radius and L is the length of the wheel axle.

- ▶ The kinematic model of the unicycle-like vehicle is the same as the majority of the tracked vehicles, whose steering technique is called *differential drive*, a generalization of the *skid steering* vehicles.

Digging a little deeper (1)

- ▶ Consider the two systems that we have seen

$$\begin{cases} \dot{y} \cos \theta_b - \dot{x} \sin \theta_b = 0 \\ \dot{\theta} = 0 \end{cases}$$

Cart moving on a railway

$$\begin{cases} \dot{y} \cos \theta - \dot{x} \sin \theta = 0 \\ \dot{\theta} = \omega \end{cases}$$

Unicycle

Why are the constraints integrable in one case and not integrable in the other?

Digging a little deeper (1)

- ▶ Consider the two systems that we have seen

$$\begin{cases} \dot{y} \cos \theta_b - \dot{x} \sin \theta_b = 0 \\ \dot{\theta} = 0 \end{cases}$$

Cart moving on a railway

$$\begin{cases} \dot{y} \cos \theta - \dot{x} \sin \theta = 0 \\ \dot{\theta} = \omega \end{cases}$$

Unicycle

Why are the constraints integrable in one case and not integrable in the other?

- ▶ Let us focus on the first constraint. If a function $h(x, y, \theta) = 0$ exists then the constraint has to be expressed as

$$\frac{\partial h}{\partial x} \dot{x} + \frac{\partial h}{\partial y} \dot{y} + \frac{\partial h}{\partial \theta} \dot{\theta} = 0$$

Digging a little deeper (2)

- ▶ The existence of $h(x, y, \theta)$ such that

$$\frac{\partial h}{\partial x} \dot{x} + \frac{\partial h}{\partial y} \dot{y} + \frac{\partial h}{\partial \theta} \dot{\theta} = 0$$

would imply, by comparison with the kinematic constraints, that

$$\dot{y} \cos \theta_b - \dot{x} \sin \theta_b = 0 \rightarrow \begin{cases} \frac{\partial h}{\partial x} = -\sin \theta_b \\ \frac{\partial h}{\partial y} = \cos \theta_b \\ \frac{\partial h}{\partial \theta} = 0 \end{cases}$$

Cart moving on a railway

$$\dot{y} \cos \theta - \dot{x} \sin \theta = 0 \rightarrow \begin{cases} \frac{\partial h}{\partial x} = -\sin \theta \\ \frac{\partial h}{\partial y} = \cos \theta \\ \frac{\partial h}{\partial \theta} = 0 \end{cases}$$

Unicycle

Digging a little deeper (2)

- ▶ The existence of $h(x, y, \theta)$ such that

$$\frac{\partial h}{\partial x} \dot{x} + \frac{\partial h}{\partial y} \dot{y} + \frac{\partial h}{\partial \theta} \dot{\theta} = 0$$

would imply, by comparison with the kinematic constraints, that

$$\dot{y} \cos \theta_b - \dot{x} \sin \theta_b = 0 \rightarrow \begin{cases} \frac{\partial h}{\partial x} = -\sin \theta_b \\ \frac{\partial h}{\partial y} = \cos \theta_b \\ \frac{\partial h}{\partial \theta} = 0 \end{cases}$$

Cart moving on a railway

$$\dot{y} \cos \theta - \dot{x} \sin \theta = 0 \rightarrow \begin{cases} \frac{\partial h}{\partial x} = -\sin \theta \\ \frac{\partial h}{\partial y} = \cos \theta \\ \frac{\partial h}{\partial \theta} = 0 \end{cases}$$

Unicycle

Clairaut's Theorem

Given any smooth (C^2 or higher) function $h(x_1, x_2, \dots, x_n)$, we have

$$\forall x_i, x_j \quad \frac{\partial}{\partial x_i} \frac{\partial h}{\partial x_j} = \frac{\partial}{\partial x_j} \frac{\partial h}{\partial x_i}.$$

Digging a little deeper (3)

- ▶ For a constraint to be integrable, the assignments must be consistent with Clairaut's theorem (mixed partials equal). Let us start with the cart moving on a railway.

$$\begin{cases} \frac{\partial}{\partial x} \frac{\partial h}{\partial \theta} = \frac{\partial}{\partial y} \frac{\partial h}{\partial \theta} = 0 \\ \frac{\partial}{\partial \theta} \frac{\partial h}{\partial x} = -\frac{\partial}{\partial \theta} (\sin \theta_b) = 0 \\ \frac{\partial}{\partial \theta} \frac{\partial h}{\partial y} = \frac{\partial}{\partial \theta} (\cos \theta_b) = 0 \end{cases}$$

which yields

$$\begin{aligned} \frac{\partial}{\partial x} \frac{\partial h}{\partial \theta} &= \frac{\partial}{\partial \theta} \frac{\partial h}{\partial x} \\ \frac{\partial}{\partial y} \frac{\partial h}{\partial \theta} &= \frac{\partial}{\partial \theta} \frac{\partial h}{\partial y} \end{aligned}$$

Hence, the theorem is respected.

Digging a little deeper (4)

- ▶ Let us now check the theorem for the “free” unicycle:

$$\begin{cases} \frac{\partial}{\partial x} \frac{\partial h}{\partial \theta} = \frac{\partial}{\partial y} \frac{\partial h}{\partial \theta} = 0 \\ \frac{\partial}{\partial \theta} \frac{\partial h}{\partial x} = -\frac{\partial}{\partial \theta} (\sin \theta) = -\cos \theta \\ \frac{\partial}{\partial \theta} \frac{\partial h}{\partial y} = \frac{\partial}{\partial \theta} (\cos \theta) = -\sin \theta \end{cases}$$

which yields

$$\begin{aligned} \frac{\partial}{\partial x} \frac{\partial h}{\partial \theta} &\neq \frac{\partial}{\partial \theta} \frac{\partial h}{\partial x} \\ \frac{\partial}{\partial y} \frac{\partial h}{\partial \theta} &\neq \frac{\partial}{\partial \theta} \frac{\partial h}{\partial y} \end{aligned}$$

Hence, the theorem is *not* respected and it is impossible to find a function that is the integral of the constraint.

Digging a little deeper (5)

- ▶ Let us discuss again the consequences.
- ▶ For the cart moving on the railway, the fact that we have two holonomic constraints implies that two generalised coordinates can be expressed as a function of the remaining one.

Digging a little deeper (5)

- ▶ Let us discuss again the consequences.
- ▶ For the cart moving on the railway, the fact that we have two holonomic constraints implies that two generalised coordinates can be expressed as a function of the remaining one.
- ▶ Hence, the system can move only on a one-dimensional manifold (i.e., a curve).

$$y = \tan \theta_b x + c, \theta = \theta_b$$

Digging a little deeper (5)

- ▶ Let us discuss again the consequences.
- ▶ For the cart moving on the railway, the fact that we have two holonomic constraints implies that two generalised coordinates can be expressed as a function of the remaining one.
- ▶ Hence, the system can move only on a one-dimensional manifold (i.e., a curve).

$$y = \tan \theta_b x + c, \theta = \theta_b$$

- ▶ This constraint does not hold for the free unicycle, which can span the entire $SE(2)$ space (all possible x, y and all possible orientations). How is this not in contrast with the kinematic constraint, for which the vehicle cannot skid sideways?

Digging a little deeper (6)

- ▶ How is it possible to reach the position $x = \epsilon, y = 0, \theta = \frac{\pi}{2}$ starting from $x = 0, y = 0, \theta = \frac{\pi}{2}$?

Digging a little deeper (6)

- ▶ How is it possible to reach the position $x = \epsilon, y = 0, \theta = \frac{\pi}{2}$ starting from $x = 0, y = 0, \theta = \frac{\pi}{2}$?
- ▶ Intuitively you cannot do it in one differential step. The possible velocities are given by the null space of $A(q)$:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = v \begin{bmatrix} \cos \theta \\ \sin \theta \\ 0 \end{bmatrix} + \omega \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$= v \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + \omega \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The vector $\begin{bmatrix} \epsilon/dt \\ 0 \\ 0 \end{bmatrix}$ cannot be generated by any assignment of v and ω .

- ▶ However

Digging a little deeper (7)

► Suppose you execute the following steps:

1. Move for a time dt_1 with $v = 0$ and $\omega = \frac{-\pi}{2dt_1}$
2. Move for a time dt_2 with $v = \frac{\epsilon}{dt_2}$ and $\omega = 0$
3. Move for a time dt_1 with $v = 0$ and $\omega = \frac{\pi}{2dt_1}$

Digging a little deeper (7)

- ▶ Suppose you execute the following steps:
 1. Move for a time dt_1 with $v = 0$ and $\omega = \frac{-\pi}{2dt_1}$
 2. Move for a time dt_2 with $v = \frac{\epsilon}{dt_2}$ and $\omega = 0$
 3. Move for a time dt_1 with $v = 0$ and $\omega = \frac{\pi}{2dt_1}$
- ▶ You would end up exactly in the position $(\epsilon, 0, \pi/2)$.

Digging a little deeper (7)

- ▶ Suppose you execute the following steps:
 1. Move for a time dt_1 with $v = 0$ and $\omega = \frac{-\pi}{2dt_1}$
 2. Move for a time dt_2 with $v = \frac{\epsilon}{dt_2}$ and $\omega = 0$
 3. Move for a time dt_1 with $v = 0$ and $\omega = \frac{\pi}{2dt_1}$
- ▶ You would end up exactly in the position $(\epsilon, 0, \pi/2)$.
- ▶ Intuitively, nonholonomy provides two command vectors (degrees of freedom) whose actions do not commute.
- ▶ By alternating them you can generate motions which would not be allowed by the instantaneous constraint.
- ▶ The motion is not constrained to a one-dimensional manifold.



Outline

Overview

Nonholonomic systems

Basic Check Questions

Advanced Questions



Question

What is the main structural difference between robot manipulators and mobile robots?

Answer

Robot manipulators are fixed to the ground and made of a chain of actuated links, while mobile robots move freely in the environment and their motion depends on wheel-ground interactions.



Question

Why can't we determine a mobile robot's position just by reading its encoders?

Answer

Because the robot's position must be integrated over time from velocities, and the path taken affects the final pose. Encoder readings alone are not sufficient.

Question

What does it mean for a system to be nonholonomic?

Answer

It means that the kinematic constraints cannot be integrated into constraints on the configuration variables. They restrict the velocities but not directly the configuration space.



Question

What is the difference between forward and inverse kinematics?

Answer

Forward kinematics: joint space → physical space (pose).

Inverse kinematics: physical space (pose) → joint space.

For mobile robots, we usually deal with differential inverse kinematics.



Question

What distinguishes holonomic from nonholonomic constraints?

Answer

- ▶ Holonomic: constraints are integrable into functions of configuration variables (reduce accessible configuration space).
- ▶ Nonholonomic: constraints apply to velocities only, not integrable, and do not reduce the accessible configuration space.



Question

For a unicycle-like robot, why can't it move sideways?

Answer

Because its wheels constrain motion to be along the wheel orientation. The lateral velocity component must be zero.



Outline

Overview

Nonholonomic systems

Basic Check Questions

Advanced Questions



Question

Why are the constraints of a cart on a railway integrable but those of a unicycle not?

Answer

The cart's constraints satisfy Clairaut's theorem (mixed partial derivatives agree), so they are integrable to holonomic constraints. The unicycle's constraints do not, hence they are nonholonomic.



Question

What role does non-commutativity of control inputs play in nonholonomic robots?

Answer

It allows generation of motions not possible in a single step. By alternating different commands (e.g., rotate then move forward), the robot can reach configurations otherwise forbidden instantaneously.

Question

Why does nonholonomy allow a unicycle to span the full $SE(2)$ space despite its lateral velocity constraint?

Answer

Because although instantaneous motion is constrained, sequences of admissible motions (e.g., rotations and forward moves) can be combined to approximate any displacement in the plane with arbitrary orientation.

Question

In the circular track example, why does the rank of $A(q)$ remain constant, and what does that imply about the system's controllability?

Answer

The rank is always 2 because both constraints are independent everywhere on the track. This means the null space is 1D, so the system can only move along the circle, making it fully constrained to the track.

Question

Consider the kinematic model $\dot{q} = G(q)u$. If $G(q)$ is not full rank at some configuration, what problem could arise?

Answer

The system would experience a singularity: some intended motions cannot be generated at that configuration. This limits controllability and may require reparameterisation of the coordinates.

An introduction to Optimal Control

Robot Planning and its Applications

University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)





Outline

The principle of optimality

The Optimal Control: formulation and examples

The Hamilton-Jacobi-Bellman formulation

The Pontryagin Minimum Principle

Bellman Principle of optimality revisited

We introduced already the Bellman optimality principle:

The Bellman Optimality Principle

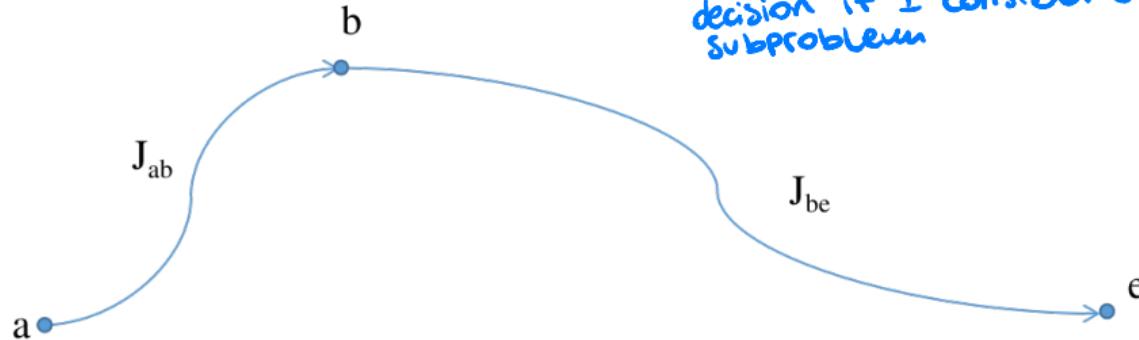
An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

- ▶ This principle was at the heart of Dynamic programming
- ▶ It is useful to think again about its meaning and to try a proof

Bellman Principle of optimality example

Assertion: If $a - b - e$ is the optimal path from a to e then $b - e$ is the optimal path from b to e . The cost of the optimal path is termed J_{ae}^* .

I don't change my decision if I consider a subproblem



Bellman Principle of optimality example

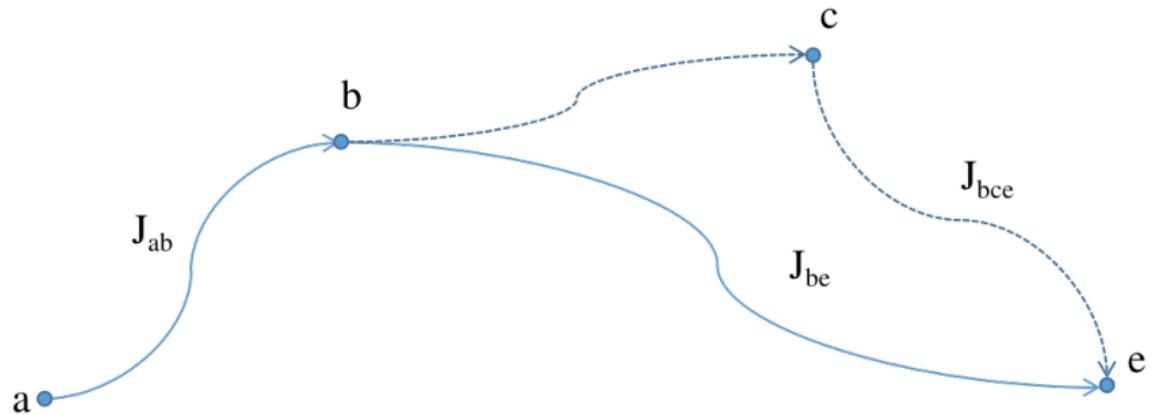
Proof

[BY CONTRADICTION:] Suppose the contrary holds. Then $J_{bce} < J_{be}$. Hence,

$$J_{ab} + J_{bce} < J_{ab} + J_{be} = J_{ae}^*,$$

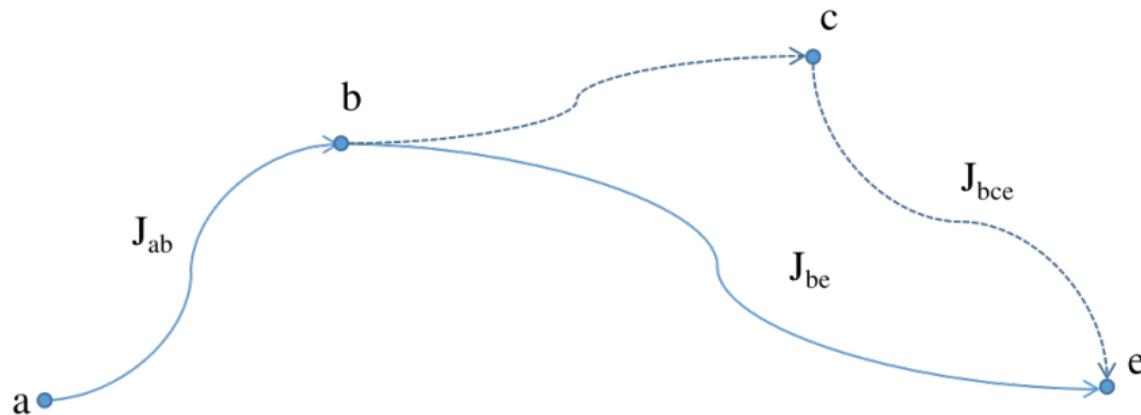
This can be proved
by contradiction

which contradicts the hypothesis that J_{ae}^* is the optimal.



Observation

- ▶ The Bellman principle applies because we are considering a multistage decision process
- ▶ Every choice moves the system into a new states that condenses its past history
- ▶ This means that the optimal choice of the path from a to b is not affected by the decisions taken to go from b to e .





Outline

The principle of optimality

The Optimal Control: formulation and examples

The Hamilton-Jacobi-Bellman formulation

The Pontryagin Minimum Principle

Myth: Queen Dido

- ▶ Elissa, sister of the King of Tyre, killed her husband and fled to Africa carrying his husband's fortune and a band of followers
- ▶ She declined the offer to join the local settlers from Larbus, who mockfully agree to concede as much land as is covered by a bull's hide.
- ▶ She had a bull's hide cut in very thin slices and she had them knit together
- ▶ The rope, covered a large area and became Dido the first Queen of Carthago
- ▶ But, what is the maximum area that can be covered with a rope of fixed length?



Isoperimetric problems

The Dido's problem is actually known as an isoperimetric problem and can be formulated as:

Isoperimetric problem

Find a domain Ω such that:

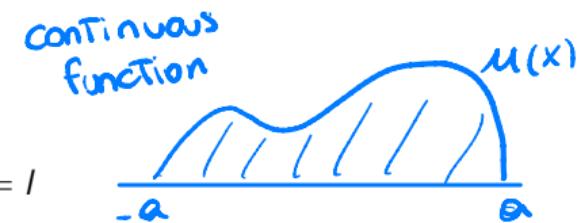
1. the area contained within the domain, given by the double integral $\int \int_{\Omega} dx dy$, is maximal
2. The length of the border $\delta\Omega$, given by the line integral $\oint_{\delta\Omega} ds$, is equal to a fixed length l

Isoperimetric problems

A simpler problem is to find a non negative function $y = u(x)$ that connects two point $[-a, 0]$ and $[a, 0]$ such that the area underneath the graph is maximal and the length of the curve if l . It is easy to see that this problem can be set up as:

$$\max_{u(x)} \int_{-a}^a u(x) dx$$

$$\text{subj. to } \int_{-a}^a \sqrt{1 + \left(\frac{du}{dx}\right)^2} dx = l$$



Functional analysis

This is a strange optimisation problem in which the decision variable is a function $u(x)$this is called *Functional analysis* or *Calculus of variations*. The optimisation cost function is called *functional*.

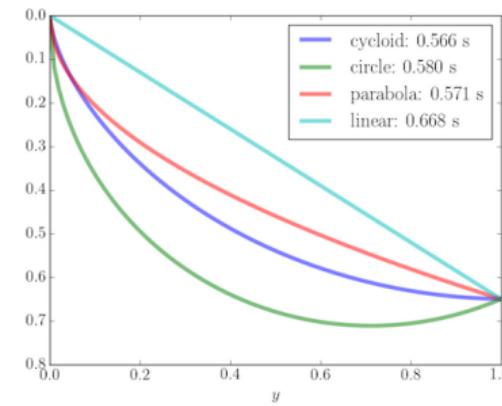
An historical challenge: the brachistocrone curve

In 1696 Johan Bernoulli addresses the most brilliant math minds of his age with a "simple" problem

The brachistocrone

Given two points A and B in a vertical plane, what is the curve traced out by a point acted on only by gravity, which starts at A and reaches B in the shortest time.

- ▶ Solution found by Newton, who submitted an anonymous paper to the Royal Society
- ▶ ...in less than 12 hours (this is what his niece said).
- ▶ The author was immediately recognised: "ex ungue leonem" (as the lion by its claws) said Bernoulli
- ▶ It is not a line but a cycloid.



The brachistocrone curve: a modern solution

- ▶ Let $(0, 0)$ be the initial point and (b, β) be the final point (y axis points downward for simplicity)
- ▶ $u(x)$ is the curve we are looking and $v(x)$ the velocity along the curve
- ▶ The infinitesimal time to go for a tiny piece dx is given by $\sqrt{1 + u'^2} dx/v$.
- ▶ the velocity v can be found by the conservation law $\frac{1}{2}mv^2 - mgu = 0$, which leads to $v = \sqrt{2gu}$

The brachistocrone formalisation

The problem can be formalised again using the calculus of variations

$$\min_{u(x)} \int_0^b \sqrt{\frac{1 + u'^2}{2gu}} dx$$

s.t. $u(0) = 0, u(b) = \beta.$

Optimal control a first example

- ▶ A mass moves on a horizontal railway and it is acted on by a force $u(t)$
- ▶ The force is bounded into a set (maximum and minimum values (e.g., $[-u, u]$)
- ▶ Find the force function to move from a point s_0 (velocity 0) to a point s_f (velocity v_f) in minimum time.

Problem formalisation

We can see the problem as the minimisation of a (trivial) integral cost index

$$\min_{u(t)} \int_0^T dt$$

$$\text{s.t. } s(0) = s_0, s(T) = s_f$$

$$v(0) = 0, v(T) = v_f$$

$$\frac{ds}{dt} = v(t)$$

$$\frac{dv}{dt} = u(t)$$

Optimal control a second example

- ▶ Considering the same point mass system as in the previous example, find the control function that in a time T moves the mass as close as possible to a point s_f spending the minimum possible control energy

Problem formalisation

In this case we can have a functional composed of a terminal part + an integral part

$$\min_{u(t)} \|s(T) - s_f\|^2 + \int_0^T v^2(t) dt$$

$$\text{s.t. } s(0) = s_0$$

$$v(0) = 0$$

$$\frac{ds}{dt} = v(t)$$

$$\frac{dv}{dt} = u(t)$$

Optimal control

- ▶ We can now state the problem in general terms.
- ▶ Consider a system described in state space by:

$$\begin{aligned}\dot{x}(t) &= f(x(t), u(t)), 0 \leq t \leq T \\ x(0) &= x_0\end{aligned}$$

where $x(t)$ is the vector of state variables, $u(t)$ are the command variables, x_0 is the initial state

- ▶ $u(t)$ is feasible if $\forall t \in [0, T]$ it belongs to the admissible set U
- ▶ By choosing $u(t)$ in the $[0, T]$ interval, we decide the state space trajectory $x(t)$ and hence $x(T)$
- ▶ We want to find a feasible function $u(t)$ in $[0, T]$ that optimises the functional

$$J(x(t), u(t)) = h(x(T)) + \int_0^T g(x(t), u(t)) dt$$

Outline

The principle of optimality

The Optimal Control: formulation and examples

The Hamilton-Jacobi-Bellman formulation

The Pontryagin Minimum Principle

Optimal control

- We want now to study the solution to the optimal control problem

$$\min_{u(t), t \in [0, T]} J(x(t), u(t)) = h(x(T)) + \int_0^T g(x(t), u(t)) dt$$

$$\dot{x}(t) = f(x(t), u(t))$$

$$x(0) = x_0$$

$$u(t) \in U$$

HJB - an informal derivation

- ▶ As a first step, let us discretise the horizon $[0, T]$ into N pieces of length $\delta = \frac{T}{N}$
- ▶ Let

$$x_k = x(k\delta), k = 0, 1, 2, \dots, N$$
$$u_k = u(k\delta), k = 0, 1, 2, \dots, N$$

- ▶ We can approximate the system differential equation and the cost functional by:

$$x_{k+1} = x_k + f(x_k, u_k)\delta$$
$$h(x_N) + \sum_{k=0}^{N-1} g(x_k, u_k)$$

- ▶ Let $J^*(t, x)$ represent the optimal cost-to-go at time t from state x and let $\bar{J}^*(t, x)$ represent the discrete-time approximation

HJB - an informal derivation

- ▶ By using the Bellman optimality principle, we can write the following backward recursive Dynamic programming equation

$$\bar{J}^*(N\delta, x) = h(x)$$

$$\bar{J}^*(k\delta, x) = \min_{u \in U} [g(x, u)\delta + \bar{J}^*((k+1)\delta, x + f(x, u)\delta)], k = 0, \dots, N-1$$

- ▶ Assuming differentiability of \bar{J}^* , we can compute the following first order
- ▶ Taylor expansion

$$\begin{aligned}\bar{J}^*((k+1)\delta, x + f(x, u)\delta) &\approx \bar{J}^*(k\delta, x) + \nabla_t \bar{J}^*(k\delta, x) \cdot \delta + \\ &+ \nabla_x \bar{J}^*(k\delta, x)^T f(x, u)\delta\end{aligned}$$

- ▶ Hence

$$\begin{aligned}\bar{J}^*(k\delta, x) &\approx \min_{u \in U} [g(x, u)\delta + \bar{J}^*(k\delta, x) + \nabla_t \bar{J}^*(k\delta, x) \cdot \delta \\ &+ \nabla_x \bar{J}^*(k\delta, x)^T f(x, u)\delta]\end{aligned}$$

HJB - an informal derivation

- ▶ We can now remove $\bar{J}^*(k\delta, x)$ as it is present in both sides of the equation.
- ▶ We can divide by δ and take the limit $\delta \rightarrow 0$.
- ▶ We make the assumption that, with small δ , $\bar{J}^*(k\delta, x)$ converges to its continuous counterpart:

$$\lim_{\delta \rightarrow 0} \bar{J}^*(k\delta, x) = J^*(t, x)$$

- ▶ We obtain the following (Hamilton Jacobi Bellman) Equation:

$$0 = \min_{u \in U} [g(x, u) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)^T f(x, u)]$$

which is a partial Differential Equation, with the boundary condition $J^*(T, x) = h(x)$.

HJB - Sufficient condition

- ▶ The HJB equation stated above should be satisfied for all pairs (t, x) by the cost-to-go function $J^*(t, x)$.
- ▶ The derivation is not rock solid: we assumed $J^*(t, x)$ to be differentiable without knowing it.
- ▶ However, if we were able to solve the HJB computationally or analytically, we could come up with a *sufficient* condition for the solution of the optimal control problem, as stated in the following.

HJB - Sufficient condition

Sufficiency of the HJB solution

Let $V(t, x)$ be a continuously differentiable (\mathcal{C}_1) solution of the HJB equation

$$0 = \min_{u \in U} [g(x, u) + \nabla_t V(t, x) + \nabla_x V(t, x)^T f(x, u)] \quad \forall t \in [0, T], x$$
$$V(T, x) = h(x) \quad \forall x.$$

And let $\mu^*(t, x)$ be the minimiser $\forall t \in [0, T], x$ and $x^*(t)$ the solution of the differential equation $\dot{x} = f(x, u)$, with $x^*(0) = x(0)$ for $u(t) = \mu^*(t, x^*)$. Assume that:

1. The differential equation $\dot{x}^* = f(x^*, \mu^*(t, x^*))$ has a unique solution starting at any pair (t, x) .
2. $\mu^*(t, x^*)$ is piecewise continuous .

Then

- $V(t, x) = J^*(t, x)$
- the control trajectory $u^*(t) = \mu^*(t, x^*)$ is the optimal control for $t \in [0, T]$.

↳ Recupero slide Example

Outline

The principle of optimality

The Optimal Control: formulation and examples

The Hamilton-Jacobi-Bellman formulation

The Pontryagin Minimum Principle

Use of the HJB equation

- ▶ We have seen that the optimal cost-to-go and the optimal control can be found knowing the solution of the HJB equation.

$$0 = \min_{u \in U} [g(x, u) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)^T f(x, u)], \forall t \in [0, T], x$$

$$J^*(T, x) = h(x), \forall x$$

- ▶ The problem is that finding this solution is far from trivial. We will now seek a more usable result.
- ▶ The optimal control is given by:

$$u^*(t) = \arg \min_{u \in U} [g(x^*(t), u(t)) + \nabla_x J^*(t, x^*(t))^T f(x^*(t), u(t))]$$

A useful Lemma

It is useful to state the following.

Lemma

Let $F(t, x, u)$ be a \mathcal{C}_1 function and let $u(t) \in U$ with U being a convex set. Assume that $\mu^*(t, x)$ is a \mathcal{C}_1 function such that:

$$\mu^*(t, x) = \arg \min_{u \in U} F(t, x, u).$$

Then

$$\nabla_t \left\{ \min_{u \in U} F(t, x, u) \right\} = \nabla_t F(t, x, \mu^*(t, x)) \quad \forall t, x$$

$$\nabla_x \left\{ \min_{u \in U} F(t, x, u) \right\} = \nabla_x F(t, x, \mu^*(t, x)) \quad \forall t, x.$$

We will apply this result to the HJB making the assumption (useless for other derivations of the principle) that U is convex.

Use of the HJB equation

Let us start again from the HJB

$$0 = \min_{u \in U} [g(x, u) + \nabla_t J^*(t, x) + \nabla_x J^*(t, x)^T f(x, u)], \forall t \in [0, T], x$$

By computing the gradient with respect to t and x and applying the lemma, we get

$$\begin{aligned} 0 &= \nabla_x g(x, \mu^*(t, x)) + \nabla_{xt}^2 J^*(t, x) + \nabla_{xx}^2 J^*(t, x) f(x, \mu^*(t, x)) + \nabla_x J^*(t, x) \nabla_x f(x, \mu^*(t, x)) \\ 0 &= \nabla_{tt}^2 J^*(t, x) + \nabla_{xt}^2 J^*(t, x)^T f(x, \mu^*(t, x)) \end{aligned} \tag{1}$$

The above apply to any pair (t, x) . In particular, we can specialise along the optimal trajectory

$$\begin{aligned} u^*(t) &= \mu^*(t, x) \\ \dot{x}^*(t) &= f(x^*(t), u^*(t)) \end{aligned}$$

Use of the HJB equation

Observe that:

$$\begin{aligned} \frac{d}{dt} (\nabla_x J^*(t, x(t)) \Big|_{x=x^*}) &= \frac{d}{dt} \nabla_x J^*(t, x) \Big|_{x=x^*} + \nabla_x \nabla_x J^*(t, x(t)) \frac{d}{dt} x(t) \Big|_{x=x^*} = \\ &= \nabla_{tx}^2 J^*(t, x^*(t)) + \nabla_{xx}^2 J^*(t, x^*(t)) f(x^*(t), u^*(t)) \end{aligned}$$

and

$$\begin{aligned} \frac{d}{dt} (\nabla_t J^*(t, x(t)) \Big|_{x=x^*}) &= \frac{d}{dt} \nabla_x J^*(t, x) \Big|_{x=x^*} + \nabla_t \nabla_x J^*(t, x(t)) \frac{d}{dt} x(t) \Big|_{x=x^*} = \\ &= \nabla_{tt}^2 J^*(t, x^*(t)) + \nabla_{xt}^2 J^*(t, x^*(t)) f(x^*(t), u^*(t)) \end{aligned}$$

Therefore, by evaluating Equation 1 in $x = x^*$, we get:

$$\begin{aligned} 0 &= \nabla_x g(x^*(t), u^*(t)) + \frac{d}{dt} (\nabla_x J^*(t, x(t)) \Big|_{x=x^*}) + \nabla_x J^*(t, x) \nabla_x f(x^*(t), u^*(t)) \\ 0 &= \frac{d}{dt} (\nabla_t J^*(t, x(t)) \Big|_{x=x^*}) \end{aligned}$$

Use of the HJB equation

Observe that:

$$\begin{aligned} \frac{d}{dt} (\nabla_x J^*(t, x(t)) \Big|_{x=x^*}) &= \frac{d}{dt} \nabla_x J^*(t, x) \Big|_{x=x^*} + \nabla_x \nabla_x J^*(t, x(t)) \frac{d}{dt} x(t) \Big|_{x=x^*} = \\ &= \nabla_{tx}^2 J^*(t, x^*(t)) + \nabla_{xx}^2 J^*(t, x^*(t)) f(x^*(t), u^*(t)) \end{aligned}$$

and

$$\begin{aligned} \frac{d}{dt} (\nabla_t J^*(t, x(t)) \Big|_{x=x^*}) &= \frac{d}{dt} \nabla_x J^*(t, x) \Big|_{x=x^*} + \nabla_t \nabla_x J^*(t, x(t)) \frac{d}{dt} x(t) \Big|_{x=x^*} = \\ &= \nabla_{tt}^2 J^*(t, x^*(t)) + \nabla_{xt}^2 J^*(t, x^*(t)) f(x^*(t), u^*(t)) \end{aligned}$$

Therefore, by evaluating Equation 1 in $x = x^*$, we get:

$$\begin{aligned} 0 &= \nabla_x g(x^*(t), u^*(t)) + \frac{d}{dt} (\nabla_x J^*(t, x(t)) \Big|_{x=x^*}) + \nabla_x J^*(t, x) \nabla_x f(x^*(t), u^*(t)) \\ 0 &= \frac{d}{dt} (\nabla_t J^*(t, x(t)) \Big|_{x=x^*}) \end{aligned}$$

Use of the HJB equation

Let

$$\lambda(t) = \nabla_x J^*(t, x(t))$$

$$\lambda_0(t) = \nabla_t J^*(t, x(t))$$

We can write equations:

$$0 = \nabla_x g(x^*(t), u^*(t)) + \frac{d}{dt} (\nabla_x J^*(t, x(t))) \Big|_{x=x^*} + \nabla_x J^*(t, x) \nabla_x f(x^*(t), u^*(t))$$

$$0 = \frac{d}{dt} (\nabla_t J^*(t, x(t))) \Big|_{x=x^*}$$

as

$$\begin{aligned}\dot{\lambda}(t) &= -\nabla_x g(x^*(t), u^*(t)) - \nabla_x f(x^*(t), u^*(t)) p(t) \\ \dot{\lambda}_0(t) &= 0\end{aligned}$$

The first differential equation is said *adjoint equation* and it can be seen that it has the terminal constraint $\lambda(T) = \nabla h(x^*(T))$.

Summary of the Discussion

- ▶ If we move along the optimal trajectory $x^*(t)$ using the command $u^*(t)$ for $t \in [0, T]$
 1. the adjoint condition

$$\dot{\lambda}(t) = -\nabla_x g(x^*(t), u^*(t)) - \nabla_x f(x^*(t), u^*(t))\lambda(t)$$

holds with the terminal constraint $\lambda(T) = \nabla h(x^*(T))$

2. the optimal control is given by:

$$u^*(t) = \arg \min_{u \in U} [g(x^*(t), u) + \lambda^T(t)f(x^*(t), u)], \forall t \in [0, T].$$

3. Hamiltonian function

Hamiltonian function

The Hamiltonian function maps triplets (x, u, p) to real numbers and is given by:

$$H(x, u, p) = g(x, u) + \lambda^T f(x, u).$$

The minimum principle (Pontryagin Principle)

Minimum Principle

- ▶ Let $\{u^*(t)|t \in [0, T]\}$ be the optimal control function, and $\{x^*(t)|t \in [0, T]\}$ be the optimal trajectory, i.e.,

$$\dot{x}^*(t) = f(x^*(t), u^*(t)), x^*(0) = x(0).$$

- ▶ Let $p(t)$ be the solution of the adjoint equation:

$$\dot{\lambda}(t) = -\nabla_x H(x^*(t), U^*(t), p(t)), \lambda(T) = \nabla h(x^*(T)),$$

where $h(\cdot)$ is the terminal cost function.

- ▶ Then for all $t \in [0, T]$

1. $u^*(t) = \arg \min_{u \in U} H(x^*(t), u, \lambda(t))$
2. There exist a constant C such that $H(x^*(t), u^*(t), \lambda(t)) = C$.

The last condition derives from the fact that: $H(x^*(t), u^*(t), \lambda(t)) = -\nabla_t J^*(t, x^*) = -\lambda_0(t)$ and $\lambda_0(t)$ is constant.

Observations

- ▶ The derivation of the Pontryagin principle shown above has been made on informal basis.
- ▶ We have made a few assumptions that can be released.
- ▶ The principle gives us a computational procedure to solve a complex variational problem.
- ▶ The optimality conditions are only necessary and need to be complemented with additional conditions (e.g., on the existence and uniqueness of the solution...)
- ▶ But the solution is very effective.



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Optimal control The Dubins maneouvres

Luigi Palopoli

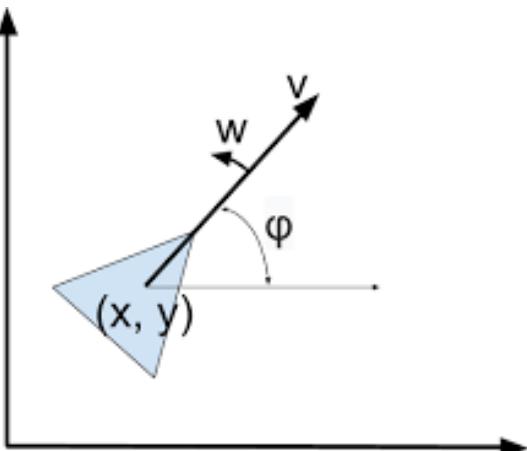


The Dubins problem



The dynamic motion planning problem

- In the following, we will consider a two wheeled unicycle (essentially our robot), its motion is described by the following equations
- We want to reach a final location starting from an initial location in minimum time



$$\dot{x} = v \cos \phi$$

$$\dot{y} = v \sin \phi$$

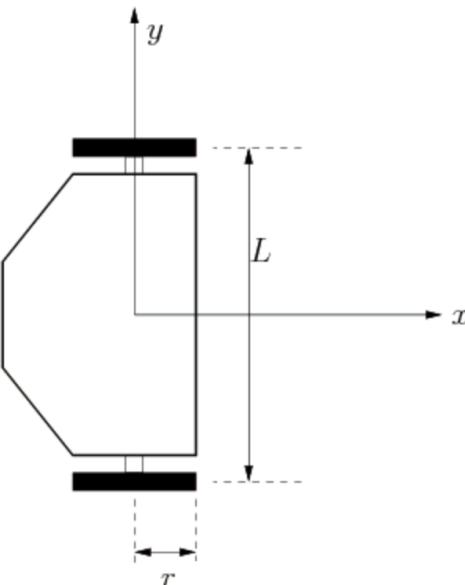
$$\dot{\phi} = \omega$$



The dynamic model

- This is a kinematic model, in which the two control variables are the forward velocity and the angular velocity
- In fact we control the speed of the wheels, but this is a lesser problem

↑
Kinematic Assumption



$$v = \frac{r}{2} (\omega_l + \omega_r)$$

$$\omega = \frac{r}{L} (\omega_r - \omega_l)$$

I'm assuming that I can control INSTANTANEOUS velocities.

In reality we apply torques and the robot Try To reach velocity desired => IT TOOK SOME TIME

Where L is the interaxial length and r is the radius of the wheel



Formulation of the problem

- We can set up the problem as an optimal control problem
- This problem is also called Dubins problem

$$\min_{v(\cdot), \omega(\cdot)} \int_0^T dt$$

Cost Function(minimum time)

$$\dot{x} = v \cos \theta$$

$$\dot{y} = v \sin \theta$$

$$\dot{\theta} = \omega$$

$$x(0) = x_0, x(T) = x_1$$

$$y(0) = y_0, y(T) = y_1$$

$$\theta(0) = \theta_0, \theta(T) = \theta_1$$

$$\mathbf{curv}(0) = \kappa_0, \mathbf{curv}(T) = \kappa_1$$

$$\omega \in [-\kappa_m, \kappa_m]$$

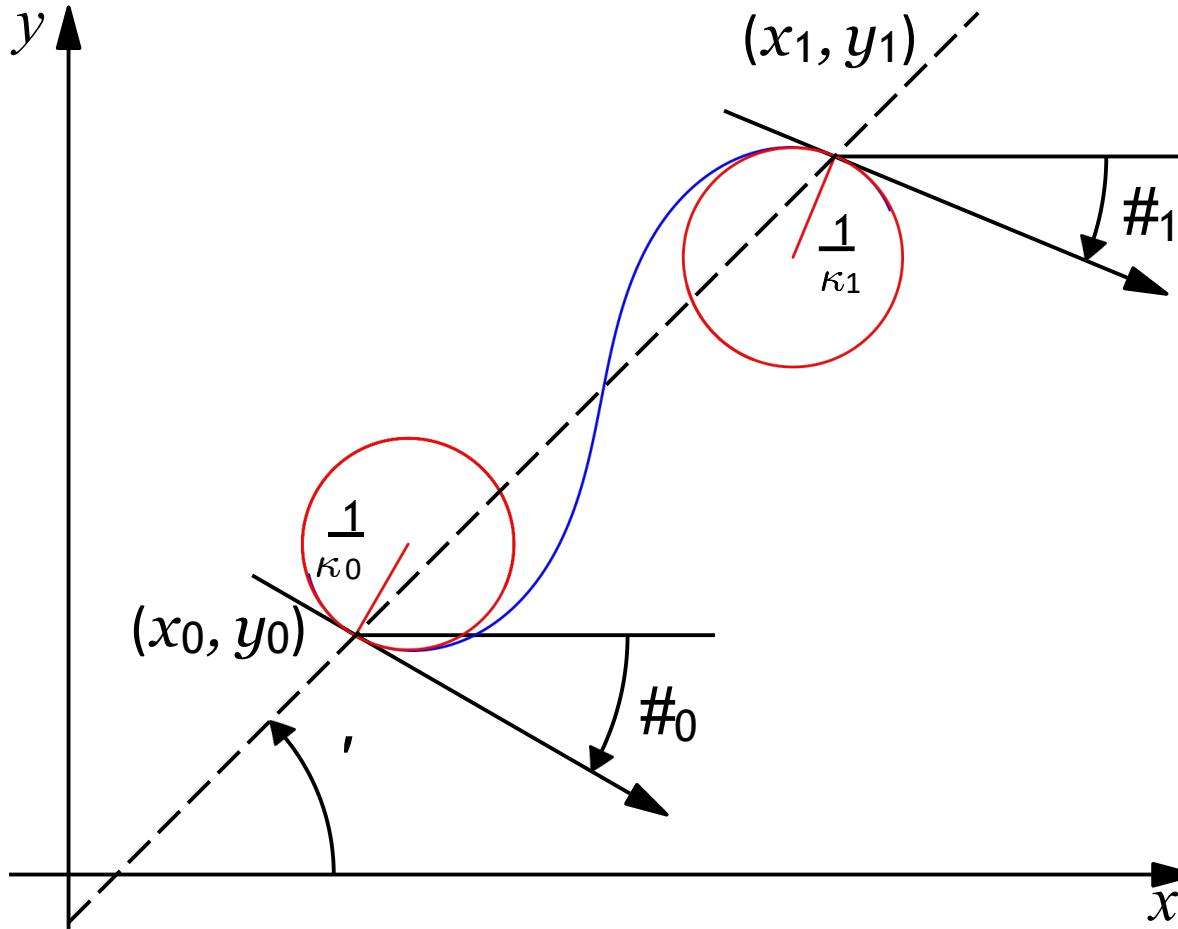
$$v \in [v_m, v_M]$$

Constraints

Optimisation problem in which the decision variables are functions



Graphical Meaning





Our sub-case

- In our project, we will keep a constant velocity

$$\min_{\omega(\cdot)} \int_0^T dt$$

If we are going straight (only v , $\omega=0$, radius = ∞)
curvature is ∞

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

$$\dot{\theta} = \omega$$

$$x(0) = x_0, x(T) = x_1$$

$$y(0) = y_0, y(T) = y_1$$

$$\theta(0) = \theta_0, \theta(T) = \theta_1$$

Define initial and final curvature

$$\text{curv}(0) = \kappa_0, \text{curv}(T) = \kappa_1$$

$$\omega \in [-\kappa_m, \kappa_m]$$

limitation on ω
translate on limitation
 $v = v_c$ on curvature

↳ since we go at constant velocity, short path means also minimum time

In this case the optimisation problem over time becomes finding the minimum length curve that joins two points.

Curvature is defined as the inverse of the radius of the circle tangent to our trajectory



Reformulation the problem using curvilinear abscissa

$\min_{\omega(\cdot)} \int_0^L ds$ *we work with the distance (s), no more with the time*

$$\frac{dx}{ds} = \cos \theta$$

$$\frac{dy}{ds} = \sin \theta$$

$$\frac{d\theta}{ds} = \omega$$

$$x(0) = x_0, x(L) = x_1$$

$$y(0) = y_0, y(L) = y_1$$

$$\theta(0) = \theta_0, \theta(L) = \theta_1$$

$$\omega \in [-\kappa_m, \kappa_m]$$

curvature can be positive or negative
(depending on the orientation of the rotation)

- We have introduced the curvilinear abscissa
- We have normalised with respect to velocity
- We have used the fact that the curvature is given by

$\text{curv}(s) = \omega(s)$ *If $\omega=0$ we're going straight*



Optimal Control

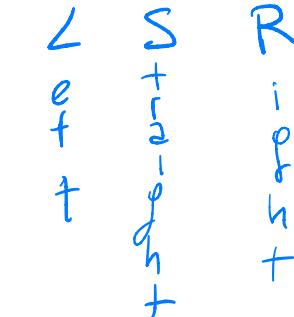
- This is a nasty problem of functional analysis: optimisation problem with the decision variable being a function
- We will solve the problem by applying the Hamilton Jacobi Bellman formulation and on the Pontryagin Principle

$$\begin{pmatrix} x \\ y \\ \theta \end{pmatrix} = \begin{pmatrix} \cos \theta \\ \sin \theta \\ \omega \end{pmatrix}$$

$$H = 1 + \lambda_1 \cos \theta + \lambda_2 \sin \theta + \lambda_3 \omega$$

Optimal solution combine in the best way $\angle S R$

Depend on λ_3





Minimum principle formulation

- In order to solve the problem, we can construct an Hamiltonian function

For a generic problem

$$\min_{\omega(\cdot)} \int_0^T l(x(t), u(t), t) dt$$

$$\dot{x} = f(x, u, t)$$

$$u(t) \in U(x)$$

Where $x \in \mathbb{R}^n$

Hamiltonian Function

$$H(x(t), u(t), \lambda) = \lambda_0 l(x, u) + \lambda(t) f(x, u)$$

λ is called co-state

- In order to solve the problem, we can construct an Hamiltonian function



Two Facts

- For the optimal trajectory $u^*(t)$ we have

$$H(x(t), u^*(t), \lambda(t)) \Big|_{\lambda_0=1} = 0$$

- The optimal action is given by

$$u^*(t) = \operatorname{argmin}_{u \in U(x)} H(x(t), u(t), \lambda(t))$$

- In other words the Hamiltonian is minimised precisely at the optimal action



The missing piece

- The missing piece is to find the expression for the co-state
- It turns out that the costate is the solution of a system of differential equations

$$\dot{\lambda} = g(x, \lambda, u^*)$$

- It can be shown that the g functions can be found differentiating the Hamiltonian function

$$\dot{\lambda}_i = - \frac{\partial H}{\partial x_i}$$



Back to Dubins

$$\min_{\omega(\cdot)} \int_0^L ds$$



Cost Function(minimum time)

$$\frac{dx}{ds} = \cos \theta$$

$$\frac{dy}{ds} = \sin \theta$$

$$\frac{d\theta}{ds} = u$$

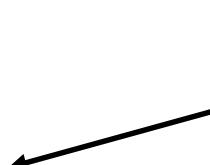
$$x(0) = x_0, x(L) = x_1$$

$$y(0) = y_0, y(L) = y_1$$

$$\theta(0) = \theta_0, \theta(L) = \theta_1$$

$$u \in [-\kappa_m, \kappa_m]$$

Constraints



Hamiltonian Function

$$\mathcal{H} = \lambda_0 + \lambda_1 \cos \vartheta + \lambda_2 \sin \vartheta + \lambda_3 u$$



Multipliers

- The two multipliers λ_1, λ_2 are constant
- Indeed
$$\frac{d\lambda_1}{ds} = -\frac{\partial H}{\partial x} = 0$$
$$\frac{d\lambda_2}{ds} = -\frac{\partial H}{\partial y} = 0$$
- Hence we can write

$$\begin{aligned}\mathcal{H} &= \lambda_0 + \lambda_1 \cos \theta + \lambda_2 \sin \theta + \lambda_3 u \\ &= \lambda_0 + \rho \cos(\theta(s) - \phi) + \lambda_3 u(s) \\ \rho &:= \sqrt{\lambda_1^2 + \lambda_2^2}, \quad \tan \phi := \frac{\lambda_2}{\lambda_1}\end{aligned}$$



Optimal Control

- The optimal control function can be found by minimising the Hamiltonian

$$\begin{aligned} u^* &= \arg \min_u \mathcal{H} = \arg \min_u \lambda_3 u(s) \\ &= \begin{cases} a = \kappa_m & \text{if } \lambda_3(s) < 0 \\ -a = -\kappa_m & \text{if } \lambda_3(s) > 0 \\ ? & \text{if } \lambda_3(s) = 0 \text{ singular case} \end{cases} \end{aligned}$$

- The switching points are given by the derivatives

$$\frac{d\lambda_3}{ds} = -\frac{\partial H}{\partial \theta} = \rho \sin(\theta(s) - \phi)$$



Extremal solutions

- Our optimal solution is once again piecewise given by extremal manouevre. Indeed

$$\lambda_3 \neq 0 \rightarrow u(s) = \pm a$$

Path made of a circle arc with the minimum curvature

$$\lambda_3 = 0 \forall s \in [s_1, s_2] \rightarrow \frac{d\lambda_3}{ds} = 0 \forall t \in [s_1, s_2] \quad \vartheta(s) \text{ constant and the curve is a segment}$$

- The optimal solution is given by three types of curves: L (curve left with minimum radius), R (curve right), S (straight line).



Theorem

- It is possible (but not easy) to show the following

Theorem: For the Dubins problem the optimal solution is always made of three extreme manoeuvres

- Not all combinations are optimal. It is possible to show that the optimal manoeuvres can be **CSC,CS,SC,S,CCC,CC,C**
 - **Where C is a curve and S is a straight**
 - Therefore the optimal manoeuvres are 15
 - **LSL, LSR, RSL, RSR, LS, RS, SL, SR, S, LRL, RLR, LR, RL, L, R**

TEST with each scenario
with our initial and final condition

There can be more than one solution
↳ PICK the one with minimum cost



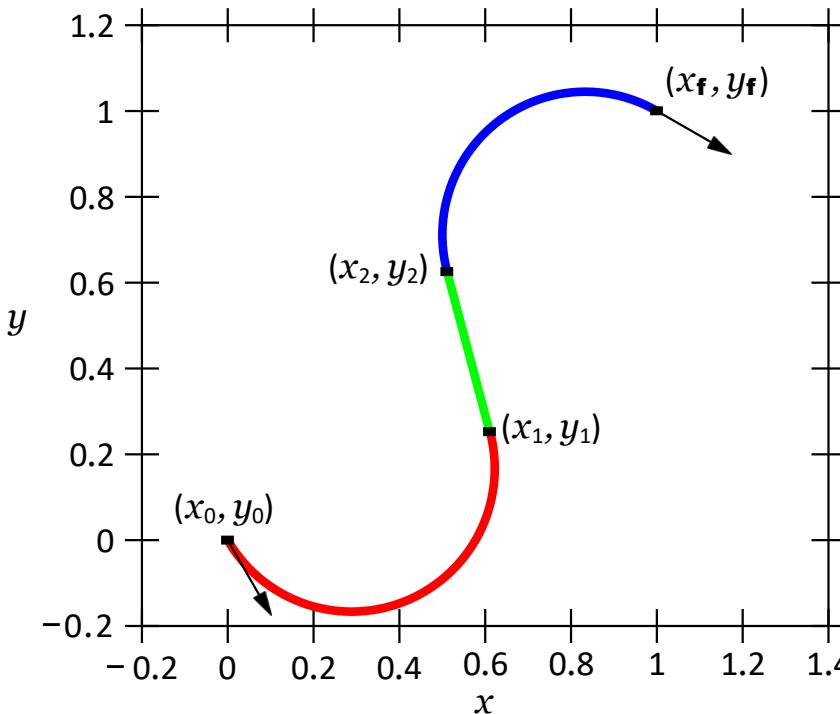
Optimal control synthesis

- Given an initial configuration and a final configuration (x , y , ϑ)
- Given a maximum curvature κ_{\max} or equivalently a minimum turning radius
- Compute all the feasible combinations and choose the optimal one

$LSL, LSR, RSL, RSR, LS, RS, SL, SR, S, LRL, RLR, LR, RL, L, R$



General setting



- Transcendental equations.
- We have to setup a similar system for all possible manoeuvres
 - Good news: it is possible to solve these equations in closed form.

Example equations for LSR

$$x_1 = x_0 - [\sin \vartheta_0 - \sin(\kappa_0 s_1 + \vartheta_0)]/\kappa_0$$

$$y_1 = y_0 + [\cos \vartheta_0 - \cos(\kappa_0 s_1 + \vartheta_0)]/\kappa_0$$

$$= x_1 + s_2 \cos(\kappa_0 s_1 + \vartheta_0)$$

$$x_2 = y_1 + s_2 \sin(\kappa_0 s_1 + \vartheta_0)$$

$$x_f = x_2 - [\sin(\kappa_0 s_1 + \vartheta_0) - \sin(\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]/\kappa_1$$

$$y_f = y_2 + [\cos(\kappa_0 s_1 + \vartheta_0) - \cos(\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]/\kappa_1$$

$$\vartheta_f = \kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0$$

The Dubins Problem

Robot Planning and its Applications

University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)





Outline

Application: The Dubins Manoeuvres

Self-Check Questions

The Dynamic Motion Planning Problem

- ▶ **System:** A two-wheeled unicycle (robot).
- ▶ **Goal:** Reach a final location (position and orientation) from an initial location in minimum time.

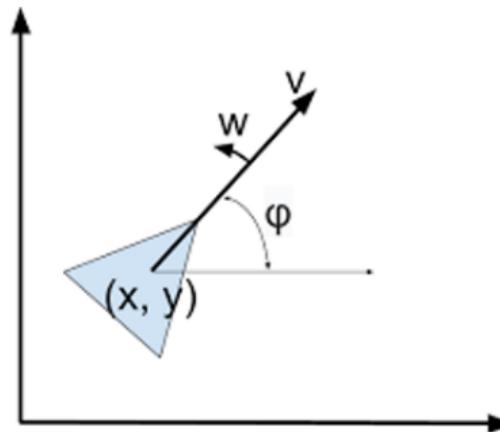


Figure: Unicycle kinematic model variables.

The Dynamic Motion Planning Problem

► Kinematic Model (State Equations):

$$\dot{x} = v \cos \phi$$

$$\dot{y} = v \sin \phi$$

$$\dot{\phi} = \omega$$

► State Vector: $\mathbf{x} = [x, y, \phi]^T$.

► Control Variables: v (forward velocity) and ω (angular velocity).

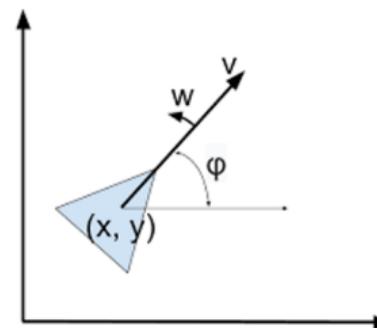


Figure: Unicycle kinematic model variables.

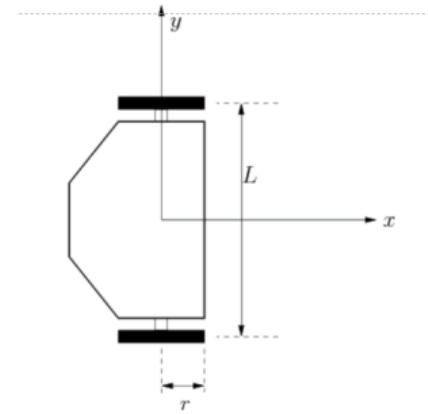
Kinematic Model to Control Inputs

- ▶ The variables v and ω are the control variables.
- ▶ They relate to the wheel speeds (ω_l, ω_r) by:

$$v = \frac{r}{2}(\omega_l + \omega_r) \quad \text{and} \quad \omega = \frac{r}{L}(\omega_r - \omega_l)$$

(Where L is the interaxial length and r is the wheel radius.)

- ▶ The optimisation problem is often called the **Dubins Problem**.



Formulation of the Dubins Problem (Minimum Time)

► Cost Function (Minimum Time):

$$\min_{v(\cdot), \omega(\cdot)} \int_0^T dt$$

► Constraints (Dynamics):

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta, \quad \dot{\theta} = \omega$$

► Boundary Conditions:

$$x(0) = x_0, x(T) = x_1$$

$$y(0) = y_0, y(T) = y_1$$

$$\theta(0) = \theta_0, \theta(T) = \theta_1$$

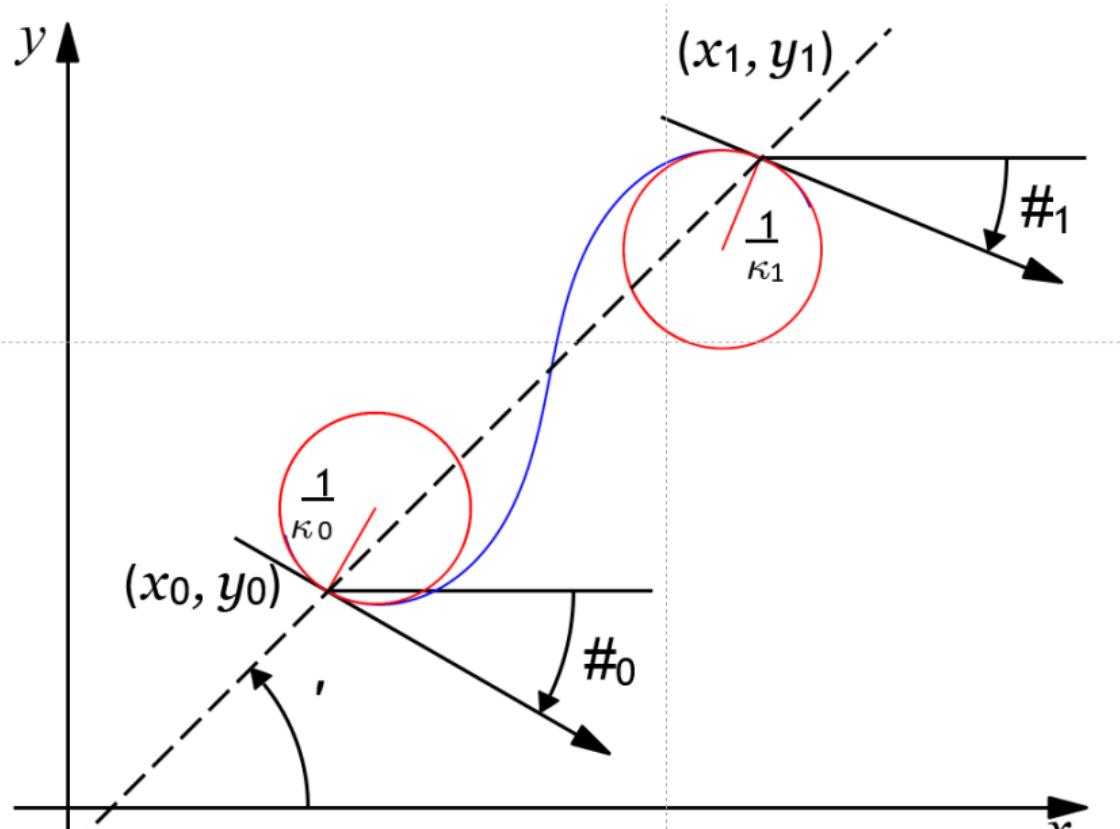
$$\text{curv}(0) = \kappa_0, \text{curv}(T) = \kappa_1$$

► Control Bounds:

$$\omega \in [-\omega_m, \omega_m]$$

$$v \in [v_m, v_M]$$

Graphical Meaning



Our Sub-Case: Constant Velocity

- ▶ Cost Function (Minimum Time):

$$\min_{\omega(\cdot)} \int_0^T dt$$

we consider vehicle that moves only forward (not backward)

- ▶ Constraints (Dynamics):

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta, \quad \dot{\theta} = \omega$$

- ▶ Boundary Conditions:

$$x(0) = x_0, x(T) = x_1$$

$$y(0) = y_0, y(T) = y_1$$

$$\theta(0) = \theta_0, \theta(T) = \theta_1$$

- ▶ Control Bounds:

$$\omega \in [-\omega_m, \omega_m], v = v_c$$

Boundaries on curvature

Constant velocity and constraints on the angular velocity imply that the maximum curvature is fixed to $\frac{\omega_m}{v_c}$.

Our Sub-Case: Constant Velocity

- ▶ For our project sub-case, we assume a **constant velocity** $v = v_c$.
- ▶ The cost function remains minimum time: $\min_{\omega(\cdot)} \int_0^T dt$.
- ▶ With v constant, minimizing time $T = \frac{L_{path}}{v_c}$ is equivalent to **finding the minimum length curve** (L_{path}) that joins the two configurations.

Our Sub-Case: Constant Velocity

- ▶ For our project sub-case, we assume a **constant velocity** $v = v_c$.
- ▶ The cost function remains minimum time: $\min_{\omega(\cdot)} \int_0^T dt$.
- ▶ With v constant, minimizing time $T = \frac{L_{path}}{v_c}$ is equivalent to **finding the minimum length curve** (L_{path}) that joins the two configurations.

Reformulation using Curvilinear Abscissa s (Arc Length):

- ▶ By normalising with respect to velocity, we use arc length s as the new time-like parameter.
- ▶ The optimisation is $\min_{\omega(\cdot)} \int_0^L ds$, where L is the path length.

$$\frac{dx}{ds} = \cos \theta, \quad \frac{dy}{ds} = \sin \theta, \quad \frac{d\theta}{ds} = \omega$$

curvature

- ▶ The curvature is given by $\kappa(s) = \frac{d\theta}{ds} = \omega(s)$, bounded by $\omega \in [-\kappa_m, \kappa_m]$.

Reformulation using curvilinear abscissa

► **Cost Function (Minimum Time):** $\min_{\omega(\cdot)} \int_0^L ds$

► **Constraints (Dynamics):**

$$\frac{dx}{ds} = \cos \theta, \quad \frac{dy}{ds} = \sin \theta, \quad \frac{d\theta}{ds} = \omega$$

► **Boundary Conditions:**

$$x(0) = x_0, x(L) = x_1$$

$$y(0) = y_0, y(L) = y_1$$

$$\theta(0) = \theta_0, \theta(L) = \theta_1$$

► **Control Bounds:**

$$\omega \in [-\kappa_m, \kappa_m] \quad (\text{Maximum Curvature})$$

Minimum Principle Formulation

- ▶ In order to solve the problem, we can construct an **Hamiltonian function**.

For a generic problem:

$$\min_{u(\cdot)} \int_0^T l(x(t), u(t), t) dt \quad \text{subject to } \dot{x} = f(x, u, t) \text{ and } u(t) \in U(x)$$

Where $x \in \mathbb{R}^n$. **Hamiltonian Function:**

$$H(x(t), u(t), \lambda) = \lambda_0 l(x, u) + \lambda(t) f(x, u)$$

λ is called **co-state**

Two Facts

- ▶ For the optimal trajectory $u^*(t)$ we have $H(x(t), u^*(t), \lambda(t))\lambda_0 = 1$ (or 0).
- ▶ The optimal action is given by

$$u^*(t) = \arg \min_{u \in U(x)} H(x(t), u(t), \lambda(t))$$

- ▶ In other words the Hamiltonian is **minimised precisely at the optimal action.**

The Missing Piece

- ▶ The missing piece is to find the expression for the **co-state**.
- ▶ It turns out that the co-state is the solution of a system of differential equations

$$\dot{\lambda} = \mathbf{g}(x, \lambda, u^*)$$

- ▶ It can be shown that the **g** functions can be found by differentiating the Hamiltonian function:

$$\dot{\lambda}_i = -\frac{\partial H}{\partial x_i}$$

Back to Dubins

Cost Function (minimum length):

$$\min_{\omega(\cdot)} \int_0^L ds$$

Constraints (using u for ω):

$$\frac{dx}{ds} = \cos \theta$$

$$\frac{dy}{ds} = \sin \theta$$

$$\frac{d\theta}{ds} = u$$

$$x(0) = x_0, \quad x(L) = x_1 \quad y(0) = y_0, \quad y(L) = y_1 \quad \theta(0) = \theta_0 \quad \theta(L) = \theta_1$$

$$u \in [-\kappa_m, \kappa_m]$$

Back to Dubins

Cost Function (minimum length):

$$\min_{\omega(\cdot)} \int_0^L ds$$

Constraints (using u for ω):

$$\frac{dx}{ds} = \cos \theta$$

$$\frac{dy}{ds} = \sin \theta$$

$$\frac{d\theta}{ds} = u$$

$$x(0) = x_0, \quad x(L) = x_1 \quad y(0) = y_0, \quad y(L) = y_1 \quad \theta(0) = \theta_0 \quad \theta(L) = \theta_1$$

$$u \in [-\kappa_m, \kappa_m]$$

Hamiltonian Function ($\lambda_0 = 1$ for the minimum time/length problem):

$$\mathcal{H} = \lambda_0 + \lambda_1 \cos \theta + \lambda_2 \sin \theta + \lambda_3 u$$

Multipliers

- ▶ The two multipliers λ_1, λ_2 are **constant**. → Infatti la derivata è zero
- ▶ Indeed:

$$\frac{d\lambda_1}{ds} = -\frac{\partial \mathcal{H}}{\partial x} = 0 \quad \text{and} \quad \frac{d\lambda_2}{ds} = -\frac{\partial \mathcal{H}}{\partial y} = 0$$

- ▶ Hence we can write the Hamiltonian:

$$\begin{aligned}\mathcal{H} &= \lambda_0 + \lambda_1 \cos \theta + \lambda_2 \sin \theta + \lambda_3 u \\ &= \lambda_0 + \rho \cos(\theta(s) - \phi) + \lambda_3 u(s)\end{aligned}$$

where

$$\rho := \sqrt{\lambda_1^2 + \lambda_2^2}, \quad \tan \phi := \frac{\lambda_2}{\lambda_1}$$

Optimal Control

- ▶ The optimal control function u^* can be found by minimising the Hamiltonian:

$$u^* = \arg \min_u \mathcal{H} = \arg \min_u \lambda_3 u(s)$$

- ▶ Given the constraint $u \in [-\kappa_m, \kappa_m]$, the optimal control u^* is a bang-bang control:

$$u^* = \begin{cases} \kappa_m & \text{if } \lambda_3(s) < 0 \\ -\kappa_m & \text{if } \lambda_3(s) > 0 \\ ? & \text{if } \lambda_3(s) = 0 \end{cases} \quad (\text{singular case})$$

- ▶ The switching points are governed by the derivative of λ_3 :

$$\frac{d\lambda_3}{ds} = -\frac{\partial \mathcal{H}}{\partial \theta} = \rho \sin(\theta(s) - \phi)$$

Extremal Solutions

- ▶ Our optimal solution is once again piecewise given by **extremal manouevres**.
- ▶ **Indeed:**

$$\lambda_3 \neq 0 \rightarrow u(s) = \pm a$$

This corresponds to a **circle arc with the minimum curvature** ($a = \kappa_m$).

$$\lambda_3 = 0 \quad \forall s \in [s_1, s_2] \rightarrow \frac{d\lambda_3}{ds} = 0 \quad \forall s \in [s_1, s_2]$$

This implies $\rho \sin(\theta(s) - \phi) = 0$. Since $\rho \neq 0$ (otherwise $H = 1$, trivial case), $\theta(s) - \phi = k\pi$. Since ϕ is constant, $\theta(s)$ is constant. This corresponds to a **straight line segment**.

- ▶ The optimal solution is given by three types of curves: **L** (curve left with minimum radius), **R** (curve right with minimum radius), **S** (straight line).

Theorem

- ▶ It is possible (but not easy) to show the following Theorem: For the Dubins problem the **optimal solution is always made of three extreme manoeuvres**.
- ▶ Not all combinations are optimal.
- ▶ It is possible to show that the optimal manoeuvres can be of the types **CSC, CS, SC, S, CCC, CC, C**.
- ▶ Where **C** is a curve (L or R) and **S** is a straight line.
- ▶ Therefore the optimal manoeuvres are **15 types**:

LSL, LSR, RSL, RSR

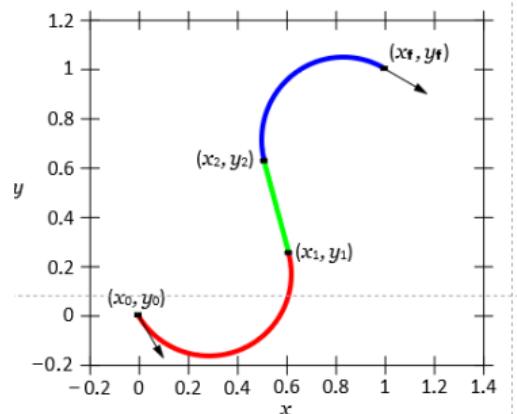
LS, RS, SL, SR, S

LRL, RLR, LR, RL, L, R

Optimal Control Synthesis

- ▶ Given an initial configuration and a final configuration (x_0, y_0, ϑ_0) and (x_f, y_f, ϑ_f) .
- ▶ Given a maximum curvature κ_{\max} or equivalently a minimum turning radius $R_{\min} = 1/\kappa_{\max}$.
- ▶ **Compute all the feasible combinations** (LSL, LSR, RSL, RSR, LS, RS, SL, SR, S, LRL, RLR, LR, RL, L, R) and **choose the optimal one** (minimum length).

General Setting and Example Equations for LSR



*curve
(κ_0 is the
radius)*

$$x_1 = x_0 - \frac{1}{\kappa_0} [\sin \vartheta_0 - \sin(\kappa_0 s_1 + \vartheta_0)]$$

$$y_1 = y_0 + \frac{1}{\kappa_0} [\cos \vartheta_0 - \cos(\kappa_0 s_1 + \vartheta_0)]$$

$$x_2 = x_1 + s_2 \cos(\kappa_0 s_1 + \vartheta_0)$$

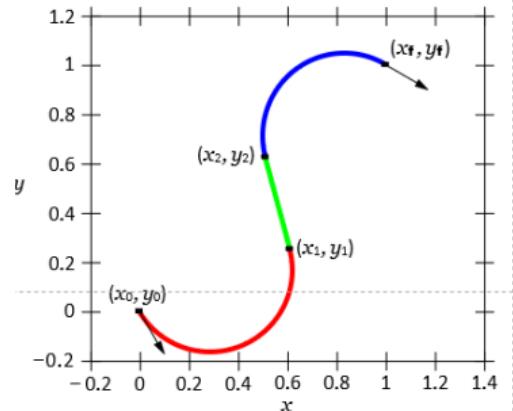
$$y_2 = y_1 + s_2 \sin(\kappa_0 s_1 + \vartheta_0)$$

$$x_f = x_2 - \frac{1}{\kappa_1} [\sin(\kappa_0 s_1 + \vartheta_0) - \sin(-\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]$$

$$y_f = y_2 + \frac{1}{\kappa_1} [\cos(\kappa_0 s_1 + \vartheta_0) - \cos(-\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]$$

$$\vartheta_f = -\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0$$

General Setting and Example Equations for LSR



$$x_1 = x_0 - \frac{1}{\kappa_0} [\sin \vartheta_0 - \sin(\kappa_0 s_1 + \vartheta_0)]$$

$$y_1 = y_0 + \frac{1}{\kappa_0} [\cos \vartheta_0 - \cos(\kappa_0 s_1 + \vartheta_0)]$$

$$x_2 = x_1 + s_2 \cos(\kappa_0 s_1 + \vartheta_0)$$

$$y_2 = y_1 + s_2 \sin(\kappa_0 s_1 + \vartheta_0)$$

$$x_f = x_2 - \frac{1}{\kappa_1} [\sin(\kappa_0 s_1 + \vartheta_0) - \sin(-\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]$$

$$y_f = y_2 + \frac{1}{\kappa_1} [\cos(\kappa_0 s_1 + \vartheta_0) - \cos(-\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)]$$

$$\vartheta_f = -\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0$$

You put initial and final conditions and you have the solution

- ▶ These are **transcendental equations**.
- ▶ We have to set up a similar system for all possible manoeuvres.
- ▶ **Good news:** it is possible to solve these equations in **closed form**.



Outline

Application: The Dubins Manoeuvres

Self-Check Questions



Dynamic Motion Planning Problem — Easy

Question:

What is the objective of the dynamic motion-planning problem for a two-wheeled robot?



Dynamic Motion Planning Problem — Answer

Answer:

To move from a given initial position and orientation to a desired final one in the minimum possible time.



Dynamic Motion Planning Problem — Trickier

Question:

Identify the state and control variables in the unicycle model.

Dynamic Motion Planning Problem — Answer

Answer:

State vector: $\mathbf{x} = [x, y, \phi]^T$

Control variables: linear velocity v and angular velocity ω .

Kinematic Model to Control Inputs — Easy

Question:

How are v and ω related to the wheel angular speeds ω_l and ω_r ?

Kinematic Model to Control Inputs — Answer

Answer:

$$v = \frac{r}{2}(\omega_l + \omega_r), \quad \omega = \frac{r}{L}(\omega_r - \omega_l).$$



Kinematic Model to Control Inputs — Trickier

Question:

Why is the optimisation problem called the *Dubins problem*?



Kinematic Model to Control Inputs — Answer

Answer:

Because it seeks the shortest feasible path for a forward-moving vehicle with bounded curvature, a problem first solved by L.E. Dubins (1957).



Dubins Problem Formulation — Easy

Question:

Write the cost function for the minimum-time Dubins problem.

Dubins Problem Formulation — Answer

Answer:

$$\min_{v(\cdot), \omega(\cdot)} \int_0^T dt = T.$$

Dubins Problem Formulation — Trickier

Question:

Why must v and ω be bounded?

Dubins Problem Formulation — Answer

Answer:

Because real vehicles have limited speed and turning capability. These bounds keep the optimisation physically meaningful: $\omega \in [-\omega_m, \omega_m]$, $v \in [v_m, v_M]$.

Constant Velocity Case — Easy

Question:

If $v = v_c$ is constant, what does minimising time reduce to?



Constant Velocity Case — Answer

Answer:

Minimising time is equivalent to minimising the total path length L_{path} .

Constant Velocity Case — Trickier

Question:

Express the maximum curvature κ_{\max} in terms of ω_m and v_c .

Constant Velocity Case — Answer

Answer:

$$\kappa_{\max} = \frac{\omega_m}{v_c}.$$



Minimum Principle — Easy

Question:

What is the purpose of the Hamiltonian function in optimal control?



Minimum Principle — Answer

Answer:

It combines dynamics and cost in one expression whose minimisation yields necessary conditions for the optimal control law.



Minimum Principle — Trickier

Question:

How is the co-state λ related to the Hamiltonian?



Minimum Principle — Answer

Answer:

It satisfies $\dot{\lambda} = -\frac{\partial H}{\partial x}$ and acts as a Lagrange multiplier enforcing the system dynamics.



Hamiltonian and Multipliers — Easy

Question:

Write the Hamiltonian for the Dubins minimum-length problem.



Hamiltonian and Multipliers — Answer

Answer:

$$H = \lambda_0 + \lambda_1 \cos \theta + \lambda_2 \sin \theta + \lambda_3 u.$$



Hamiltonian and Multipliers — Trickier

Question:

Why are λ_1 and λ_2 constant along s ?

Hamiltonian and Multipliers — Answer

Answer:

Because $\frac{\partial H}{\partial x} = \frac{\partial H}{\partial y} = 0$, hence $\frac{d\lambda_1}{ds} = \frac{d\lambda_2}{ds} = 0$.

Optimal Control Law — Easy

Question:

What type of control arises when $u \in [-\kappa_m, \kappa_m]$?



Optimal Control Law — Answer

Answer:

A *bang-bang* control: $u = \pm \kappa_m$ except at switching points.



Optimal Control Law — Trickier

Question:

What condition determines switching between left and right turns?

Optimal Control Law — Answer

Answer:

Switches occur when the sign of $\lambda_3(s)$ changes, since $\frac{d\lambda_3}{ds} = \rho \sin(\theta - \phi)$.



Extremal Solutions — Easy

Question:

What are the three basic motion primitives in the optimal path?



Extremal Solutions — Answer

Answer:

Left-turn arc (L), right-turn arc (R), and straight line (S).

Extremal Solutions — Trickier

Question:

Why does $\lambda_3 = 0$ imply a straight segment?



Extremal Solutions — Answer

Answer:

If $\lambda_3 = 0$, then $\rho \sin(\theta - \phi) = 0 \Rightarrow \theta$ constant \Rightarrow zero curvature (straight line).



Dubins Theorem — Easy

Question:

According to Dubins' theorem, what is the structure of any optimal path?



Dubins Theorem — Answer

Answer:

Every optimal path consists of at most three extremal segments (L, R, S).



Dubins Theorem — Trickier

Question:

List the main types of extremal path structures.



Dubins Theorem — Answer

Answer:

CSC, CS, SC, S, CCC, CC, and C, where C is a curved (L or R) segment and S is straight.



Optimal Control Synthesis — Easy

Question:

What information must be known to compute an optimal Dubins path?

Optimal Control Synthesis — Answer

Answer:

Initial and final configurations (x_0, y_0, θ_0) , (x_f, y_f, θ_f) , and the maximum curvature κ_{\max} or minimum radius $R_{\min} = 1/\kappa_{\max}$.



Optimal Control Synthesis — Trickier

Question:

What is the basic method for finding the optimal path?

Optimal Control Synthesis — Answer

Answer:

Enumerate all feasible patterns (e.g., LSL, RSR, RLR, etc.), compute their total lengths, and select the shortest one.



Example Equations — Easy

Question:

Why are the LSR equations called *transcendental*?



Example Equations — Answer

Answer:

They include trigonometric terms that make them nonlinear and not solvable by simple algebraic manipulation.



Example Equations — Trickier

Question:

What does it mean that the LSR equations have a *closed-form* solution?



Example Equations — Answer

Answer:

It means explicit analytical formulas for segment lengths (s_1, s_2, s_3) can be derived without iterative numerical computation.

Dubins Curves

Robot Planning and its Applications

University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)
Courtesy M. Frego, P. Bevilacqua



Dubins Curve

Given:

- ▶ initial (x_0, y_0, ϑ_0) and final (x_f, y_f, ϑ_f) configuration
- ▶ bound on maximum path curvature κ_{\max} , or equivalently minimum on the turning radius $\rho_{\min} = 1/\kappa_{\max}$
- ▶ a constraint that the vehicle can only move forward *(no backward motion)*

The Dubins curve is the shortest path that satisfies all these constraints.

Dubins Curve

- ▶ composed of at most three arcs
- ▶ each arc is either a circular arc or a straight line segment
- ▶ each arc is denoted using one of the following symbols:

L: left turn

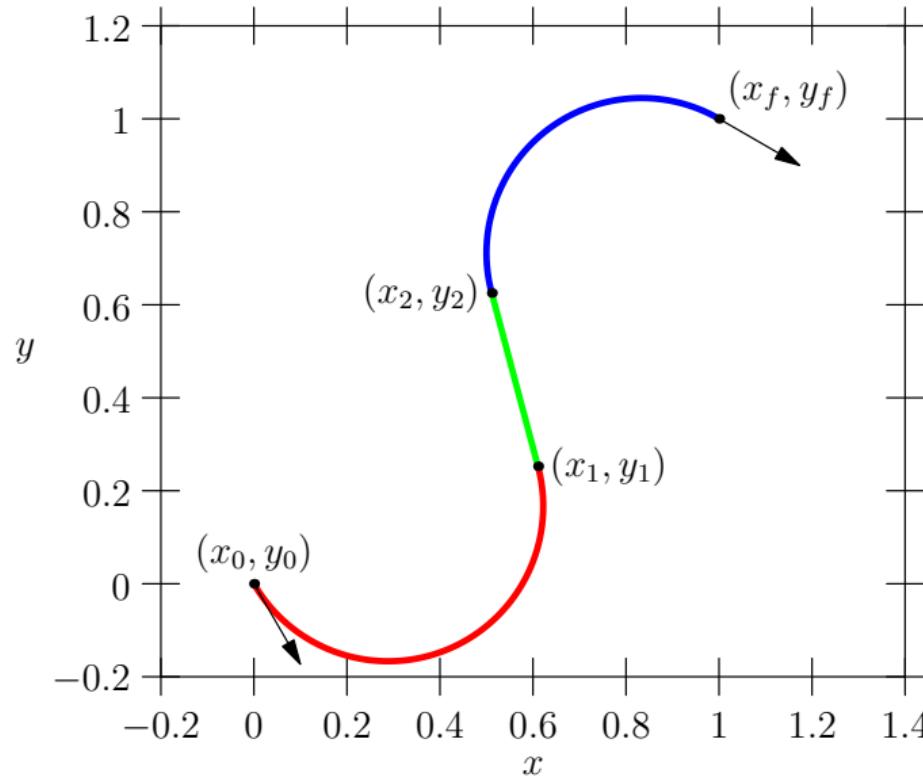
R: right turn

S: straight segment

Dubins Curve

- ▶ only the following combinations can yield optimal solutions: **LSL, LSR, RSL, RSR, LRL, RLR**
- ▶ other possible solutions are composed by less than 3 arcs: **LS, RS, SL, SR, LR, RL, L, R, S**
- ▶ These degenerate solutions can be represented using the six basic primitives, when one or two segments have zero length.

General Setting



General Setting - Example equations

CSC case

$$x_1 = x_0 - [\sin \vartheta_0 - \sin(\kappa_0 s_1 + \vartheta_0)] / \kappa_0$$

$$y_1 = y_0 + [\cos \vartheta_0 - \cos(\kappa_0 s_1 + \vartheta_0)] / \kappa_0$$

$$x_2 = x_1 + s_2 \cos(\kappa_0 s_1 + \vartheta_0)$$

$$y_2 = y_1 + s_2 \sin(\kappa_0 s_1 + \vartheta_0)$$

$$x_f = x_2 - [\sin(\kappa_0 s_1 + \vartheta_0) - \sin(\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)] / \kappa_1$$

$$y_f = y_2 + [\cos(\kappa_0 s_1 + \vartheta_0) - \cos(\kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0)] / \kappa_1$$

$$\vartheta_f = \kappa_1 s_3 + \kappa_0 s_1 + \vartheta_0$$

General Setting

A closed-form solution is possible, but it involves many terms.

- ~~ How can we simplify the expression?
- ~~ Transform the general setting into a standard one, i.e.:

- ▶ invertible transform
- ▶ (x_0, y_0) fixed to $(-1, 0)$
- ▶ (x_f, y_f) fixed to $(1, 0)$
- ▶ adjust κ_{\max} to κ'_{\max}

} On horizontal axis



The bipolar transform reduces the number of parameters from seven to three: ϑ'_0 , ϑ'_f and κ'_{\max} .

Bipolar transform \mathbf{T}

It is a composition of rotation, translation and scaling:

$$\Delta x = x_f - x_0 \quad \Delta y = y_f - y_0$$

$$\varphi = \text{atan2}(\Delta y, \Delta x)$$

$$\lambda = \frac{1}{2} \text{hypot}(\Delta x, \Delta y) = \frac{1}{2} \sqrt{\Delta x^2 + \Delta y^2}$$

$$\vartheta'_0 = \vartheta_0 - \varphi \quad \vartheta'_f = \vartheta_f - \varphi$$

$$\kappa'_{\max} = \lambda \kappa_{\max}$$

Bipolar transform

$$\mathbf{T} \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{\lambda} \left(\begin{bmatrix} \cos \varphi & \sin \varphi \\ -\sin \varphi & \cos \varphi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} \right),$$
$$\mathbf{T}^{-1} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{bmatrix} \cos \varphi & -\sin \varphi \\ \sin \varphi & \cos \varphi \end{bmatrix} \left(\lambda \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \bar{x} \\ \bar{y} \end{bmatrix} \right).$$

where φ is the unique angle in $[-\pi, \pi)$ satisfying

$$\bar{x} = x_0 \cos \varphi + y_0 \sin \varphi + \lambda,$$

$$\bar{y} = -x_0 \sin \varphi + y_0 \cos \varphi.$$

The transformation \mathbf{T} maps the original coordinates into the standard frame.

Standard Setting

- ▶ We can now consider the Dubins problem, without loss of generality, only for points fixed at $(-1, 0)$ and $(1, 0)$ with variable angles ϑ_0 and ϑ_1 and a fixed κ_{\max} , and with a slight abuse of notation, forget the prime symbol ' in the variables.
- ▶ The sub-problems are solved if the lengths $s_i \geq 0$ of the curve segments are specified.
- ▶ By direct substitution, the seven equations of the previous slide can be reduced to 3 equations in s_1, s_2, s_3 .

The Three Cardinal Equations

On a solution the following three equations must hold

$$\begin{aligned}
 2 = & s_1 \operatorname{sinc} \left(\frac{1}{2} \kappa_0 s_1 \right) \cos \left(\vartheta_0 + \frac{1}{2} \kappa_0 s_1 \right) + \\
 & + s_2 \operatorname{sinc} \left(\frac{1}{2} \kappa_1 s_2 \right) \cos \left(\vartheta_0 + \kappa_0 s_1 + \frac{1}{2} \kappa_1 s_2 \right) + \\
 & + s_3 \operatorname{sinc} \left(\frac{1}{2} \kappa_2 s_3 \right) \cos \left(\vartheta_0 + \kappa_0 s_1 + \kappa_1 s_2 + \frac{1}{2} \kappa_2 s_3 \right) \\
 0 = & s_1 \operatorname{sinc} \left(\frac{1}{2} \kappa_0 s_1 \right) \sin \left(\vartheta_0 + \frac{1}{2} \kappa_0 s_1 \right) + \\
 & + s_2 \operatorname{sinc} \left(\frac{1}{2} \kappa_1 s_2 \right) \sin \left(\vartheta_0 + \kappa_0 s_1 + \frac{1}{2} \kappa_1 s_2 \right) + \\
 & + s_3 \operatorname{sinc} \left(\frac{1}{2} \kappa_2 s_3 \right) \sin \left(\vartheta_0 + \kappa_0 s_1 + \kappa_1 s_2 + \frac{1}{2} \kappa_2 s_3 \right) \\
 \vartheta_f = & \kappa_0 s_1 + \kappa_1 s_2 + \kappa_2 s_3 + \vartheta_0
 \end{aligned}$$

Auxiliary functions

- ▶ To verify if the computed values are a valid solution, you can simply evaluate the three equations of the next slide,

$$\kappa_i = \{0, +\kappa_{\max}, -\kappa_{\max}\}, \quad i = 0, 1, 2.$$

- ▶ The first case corresponds to a straight line segment (zero curvature), the second to a left turn (positive curvature) and the third is to a right turn (negative curvature).
- ▶ We also need a special function for the stable solution presented in the next slide. This is the sinc function, defined as follows

$$\text{sinc}(x) \equiv \begin{cases} 1 & x = 0 \\ \frac{\sin x}{x} & x \neq 0 \end{cases}$$

Implementation of sinc

The sinc function is implemented in a numerically stable way, so that it remains numerically stable for small values of x :

$$\text{sinc}(x) \equiv \begin{cases} 1 - x^2 \left(\frac{1}{6} - \frac{x^2}{120} \right) & |x| < 0.002 \\ \frac{\sin x}{x} & |x| \geq 0.002 \end{cases}$$

Another useful function is mod2pi to normalize the angles in the range $[0, 2\pi)$.

1. LSL

closed form
solution

Let

$$C = \cos \vartheta_f - \cos \vartheta_0, \quad S = 2\kappa + \sin \vartheta_0 - \sin \vartheta_f$$

- . The segment lengths are given by:

$$s_1 = \frac{1}{\kappa} \text{mod2pi} [\text{atan2}(C, S) - \vartheta_0]$$

$$s_2 = \frac{1}{\kappa} \sqrt{2 + 4\kappa^2 - 2 \cos(\vartheta_0 - \vartheta_f) + 4\kappa(\sin \vartheta_0 - \sin \vartheta_f)}$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_f - \text{atan2}(C, S)]$$

2. RSR

Let

$$C = \cos \vartheta_0 - \cos \vartheta_f, \quad S = 2\kappa - \sin \vartheta_0 + \sin \vartheta_f$$

The segment lengths are given by:

$$s_1 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_0 - \text{atan2}(C, S)]$$

$$s_2 = \frac{1}{\kappa} \sqrt{2 + 4\kappa^2 - 2 \cos(\vartheta_0 - \vartheta_f) - 4\kappa(\sin \vartheta_0 - \sin \vartheta_f)}$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\text{atan2}(C, S) - \vartheta_f]$$

3. LSR

Let

$$C = \cos \vartheta_0 + \cos \vartheta_f, \quad S = 2\kappa + \sin \vartheta_0 + \sin \vartheta_f$$

The segment lengths are given by:

$$s_2 = \frac{1}{\kappa} \sqrt{-2 + 4\kappa^2 + 2 \cos(\vartheta_0 - \vartheta_f) + 4\kappa(\sin \vartheta_0 + \sin \vartheta_f)}$$

$$s_1 = \frac{1}{\kappa} \text{mod2pi} [\text{atan2}(-C, S) - \text{atan2}(-2, \kappa s_2) - \vartheta_0]$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\text{atan2}(-C, S) - \text{atan2}(-2, \kappa s_2) - \vartheta_f]$$

4. RSL

Let

$$C = \cos \vartheta_0 + \cos \vartheta_f, \quad S = 2\kappa - \sin \vartheta_0 - \sin \vartheta_f$$

The segment lengths are given by:

$$s_2 = \frac{1}{\kappa} \sqrt{-2 + 4\kappa^2 + 2 \cos(\vartheta_0 - \vartheta_f) - 4\kappa(\sin \vartheta_0 + \sin \vartheta_f)}$$

$$s_1 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_0 - \text{atan2}(C, S) + \text{atan2}(2, \kappa s_2)]$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_f - \text{atan2}(C, S) + \text{atan2}(2, \kappa s_2)]$$

5. RLR

Let

$$C = \cos \vartheta_0 - \cos \vartheta_f, \quad S = 2\kappa - \sin \vartheta_0 + \sin \vartheta_f$$

The segment lengths are given by:

$$s_2 = \frac{1}{\kappa} \text{mod2pi} \left[2\pi - \arccos \left(\frac{1}{8} (6 - 4\kappa^2 + 2 \cos(\vartheta_0 - \vartheta_f) + 4\kappa(\sin \vartheta_0 - \sin \vartheta_f)) \right) \right]$$

$$s_1 = \frac{1}{\kappa} \text{mod2pi} \left[\vartheta_0 - \text{atan2}(C, S) + \frac{1}{2}\kappa s_2 \right]$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_0 - \vartheta_f + \kappa(s_2 - s_1)]$$

6. LRL

Let

$$C = \cos \vartheta_f - \cos \vartheta_0, \quad S = 2\kappa + \sin \vartheta_0 - \sin \vartheta_f$$

The segment lengths are given by:

$$s_2 = \frac{1}{\kappa} \text{mod2pi} \left[2\pi - \arccos \left(\frac{1}{8} (6 - 4\kappa^2 + 2 \cos(\vartheta_0 - \vartheta_f) - 4\kappa(\sin \vartheta_0 - \sin \vartheta_f)) \right) \right]$$

$$s_1 = \frac{1}{\kappa} \text{mod2pi} \left[-\vartheta_0 + \text{atan2}(C, S) + \frac{1}{2}\kappa s_2 \right]$$

$$s_3 = \frac{1}{\kappa} \text{mod2pi} [\vartheta_f - \vartheta_0 + \kappa(s_2 - s_1)]$$

Remaining cases

You have 2 closed-form solutions
↓
Try all
↓
choose the best

The remaining combinations with fewer than three segments can be obtained from the above relations, when one of the arcs has length equal to zero. For instance, the case LR can be obtained from RLR with $s_1 = 0$ or from LRL with $s_3 = 0$.

Example - [Kaya 2017 - ex. 3]

Problem: Find the Dubins path connecting the following configurations.

$$\hat{x}_0 = 0$$

$$\hat{x}_f = 4$$

$$\hat{y}_0 = 0$$

$$\hat{y}_f = 0$$

$$\hat{\vartheta}_0 = -\frac{\pi}{2}$$

$$\hat{\vartheta}_f = -\frac{\pi}{2}$$

with a maximum curvature of $\hat{\kappa}_{\max} = 1$ or (equivalently) $\hat{\rho}_{\min} = 1/\hat{\kappa}_{\max} = 1$.
The hat notation refers to the original (unscaled) reference frame.

Example - solution

- ▶ The first step is to map the problem into the standard setting

$$x_0 = -1$$

$$y_0 = 0$$

$$x_f = 1$$

$$y_f = 0$$

Example - solution

- ▶ The first step is to map the problem into the standard setting

$$x_0 = -1$$

$$x_f = 1$$

$$y_0 = 0$$

$$y_f = 0$$

- ▶ We need to find φ and λ to transform ϑ_0 , ϑ_f and κ_{\max} . We have that

$$\Delta x = \hat{x}_f - \hat{x}_0 = 4, \quad \Delta y = \hat{y}_f - \hat{y}_0 = 0.$$

Example - solution

- ▶ The first step is to map the problem into the standard setting

$$x_0 = -1$$

$$x_f = 1$$

$$y_0 = 0$$

$$y_f = 0$$

- ▶ We need to find φ and λ to transform ϑ_0 , ϑ_f and κ_{\max} . We have that

$$\Delta x = \hat{x}_f - \hat{x}_0 = 4, \quad \Delta y = \hat{y}_f - \hat{y}_0 = 0.$$

- ▶ The resulting $\varphi = \text{atan2}(\Delta y, \Delta x) = 0$.

Example - solution

- ▶ The scaled angles are given by

$$\vartheta_0 = \hat{\vartheta}_0 - \varphi = -\frac{\pi}{2}, \quad \vartheta_f = \hat{\vartheta}_f - \varphi = -\frac{\pi}{2}$$

- ▶ The scaling factor λ is given by

$$\lambda = \frac{1}{2} \sqrt{\Delta x^2 + \Delta y^2} = \frac{1}{2} \sqrt{16} = 2$$

- ▶ The scaled maximum curvature is $\kappa_{\max} = \hat{\kappa}_{\max} \lambda = 2$.

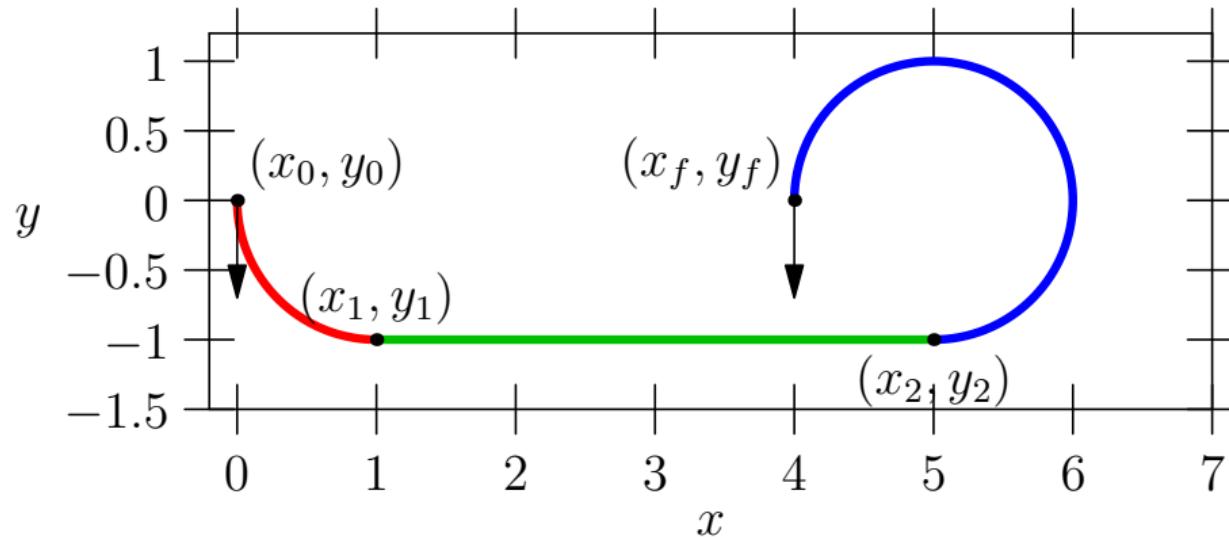
Example - solution

It is now possible to evaluate the six manoeuvres:

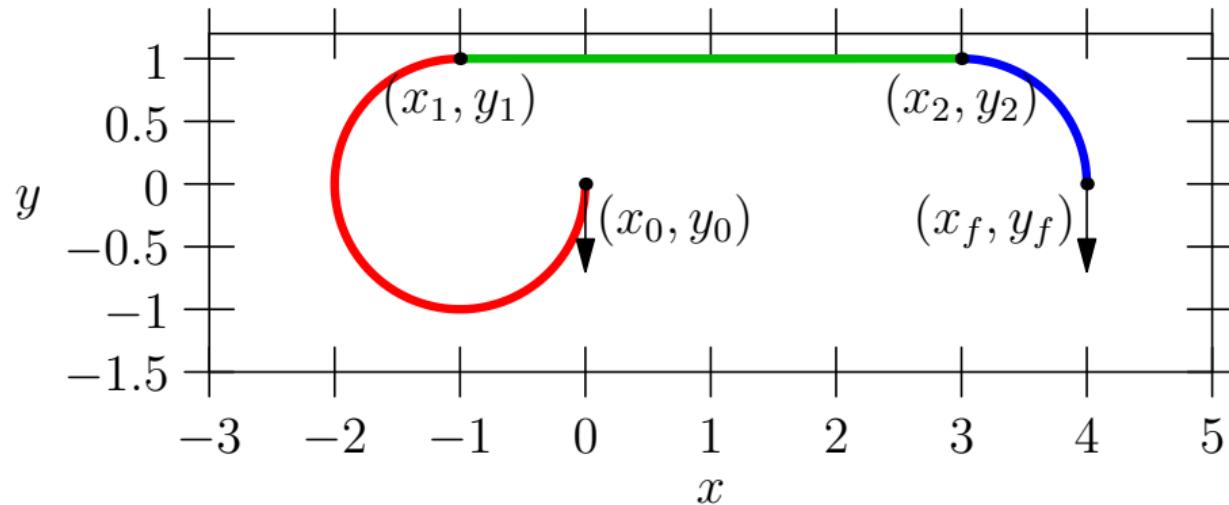
1. LSL $L = 10.28, s_1 = 1.57, s_2 = 4, s_3 = 4.71$
2. RSR $L = 10.28, s_1 = 4.71, s_2 = 4, s_3 = 1.57$
3. LSR $L = 6.28, s_1 = 3.14, s_2 = 0, s_3 = 3.14$
4. RSL $L = 15.76, s_1 = 5.05, s_2 = 5.66, s_3 = 5.05$
5. RLR $L = 6.28, s_1 = 0, s_2 = 3.14, s_3 = 3.14$
6. LRL $L = 6.28, s_1 = 3.14, s_2 = 3.14, s_3 = 0$

The optimal solution correspond to a degenerate (2 arcs) LR manoeuvre.

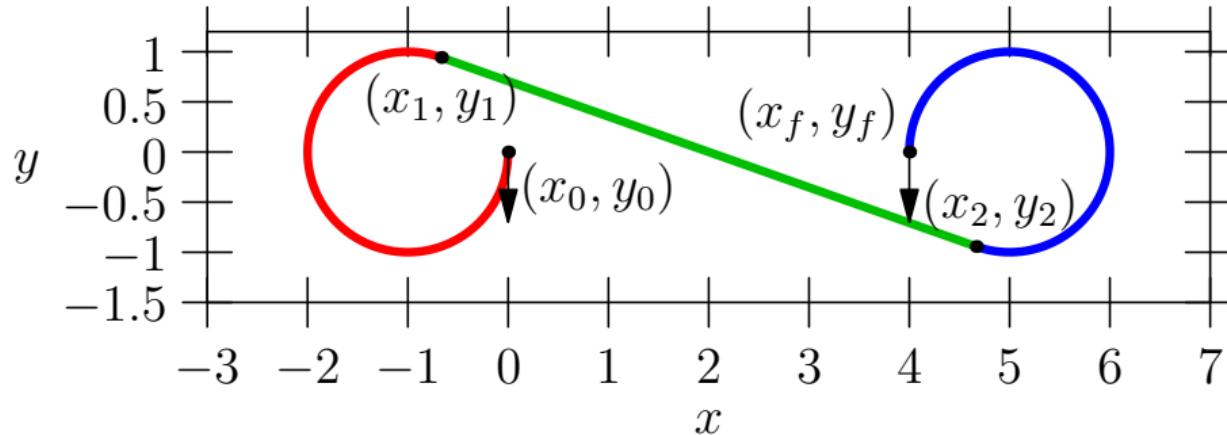
Example - solution LSL



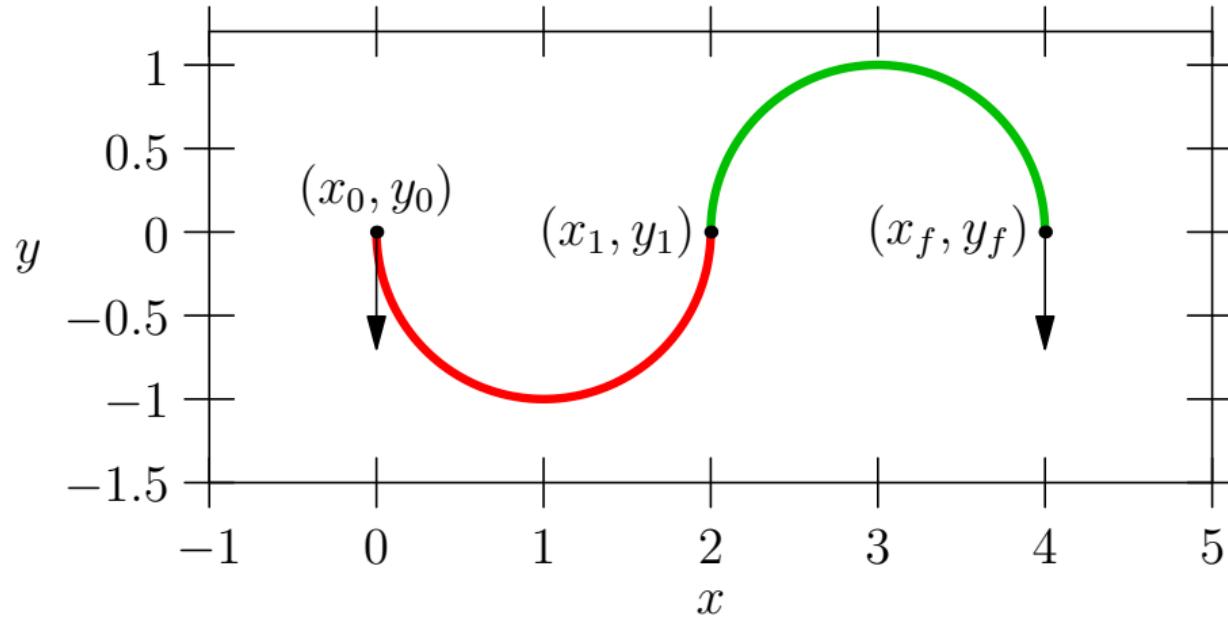
Example - solution RSR



Example - solution RSL



Example - solution LR - optimal





Outline

Self-check questions



Dubins Curve – Definition

What is a Dubins curve and what are its basic constraints?



Answer

A Dubins curve is the shortest path between two configurations (x_0, y_0, ϑ_0) and (x_f, y_f, ϑ_f) with bounded curvature and forward-only motion.



Path Composition

What are the three types of segments that can form a Dubins path?



Answer

Left turns (L), right turns (R), and straight segments (S).



Optimal Combinations

Why are only six combinations (LSL, LSR, RSL, RSR, LRL, RLR) considered optimal?



Answer

These six represent all minimal three-segment sequences that satisfy curvature and continuity constraints; all other valid paths are degenerate forms.



Transforming the Problem

Why is it useful to transform the Dubins problem into a standard setting?

Answer

The transformation simplifies the problem by fixing the start and end points, reducing the number of free parameters.



Bipolar Transform

Which transformations are combined in the bipolar transform to obtain the standard frame?



Answer

It combines rotation, translation, and scaling to map the start to $(-1, 0)$ and the goal to $(1, 0)$, reducing parameters to angles and curvature.



Scaling Factor

What does the scaling factor λ represent in the bipolar transform?



Answer

$\lambda = \frac{1}{2} \sqrt{(x_f - x_0)^2 + (y_f - y_0)^2}$ is half the distance between start and goal points, serving as a scaling factor.



Angle Transformation

How are the transformed angles ϑ'_0 and ϑ'_f related to the originals?

Answer

They are rotated by the reference angle $\phi = \text{atan2}(\Delta y, \Delta x)$: $\vartheta'_0 = \vartheta_0 - \phi$,
 $\vartheta'_f = \vartheta_f - \phi$.



Cardinal Equations

How many independent equations must a Dubins path satisfy?



Answer

Three: two ensure positional continuity and one enforces the final orientation constraint.



Sinc Function

What is the mathematical definition of the sinc function?



Answer

$\text{sinc}(x) = \frac{\sin x}{x}$ for $x \neq 0$, and equals 1 for $x = 0$.



Numerical Stability

Why is a special implementation of $\text{sinc}(x)$ needed for small $|x|$ values?



Answer

For small x , direct computation of $\sin x/x$ can lose precision, so a series expansion ensures numerical stability.



Path Types

Name two Dubins path types that start and end with the same turning direction.



Answer

LSL and RSR.



Example Problem

What are the initial and final headings in the example problem (Kaya 2017)?



Answer

Both initial and final headings are $-\pi/2$, meaning the vehicle faces downward in both configurations.



Scaling and Curvature

How does the scaling factor λ affect the maximum curvature?

Answer

It scales the curvature as $\kappa_{\max} = \lambda \kappa_{\max}$, increasing the effective curvature with larger λ .

Optimal Path

Which path type was found to be optimal in the example?



Answer

The LR (Left–Right) manoeuvre, a degenerate two-arc solution.



Visual Identification

How can you visually recognise the shortest Dubins path among alternatives?



Answer

It is the trajectory with the smallest total arc length that still meets curvature and orientation constraints.

Multipoint Interpolation using Dubins curves

Robot Planning and its Applications
University of Trento

Luigi Palopoli (luigi.palopoli@unitn.it)
Courtesy M. Frego, P. Bevilacqua





Outline

Problem Definition

Problem Definition

Iterative Dynamic Programming Solution

Refinement

Examples

Numerical Results



Outline

Problem Definition

Problem Definition

Iterative Dynamic Programming Solution

Refinement

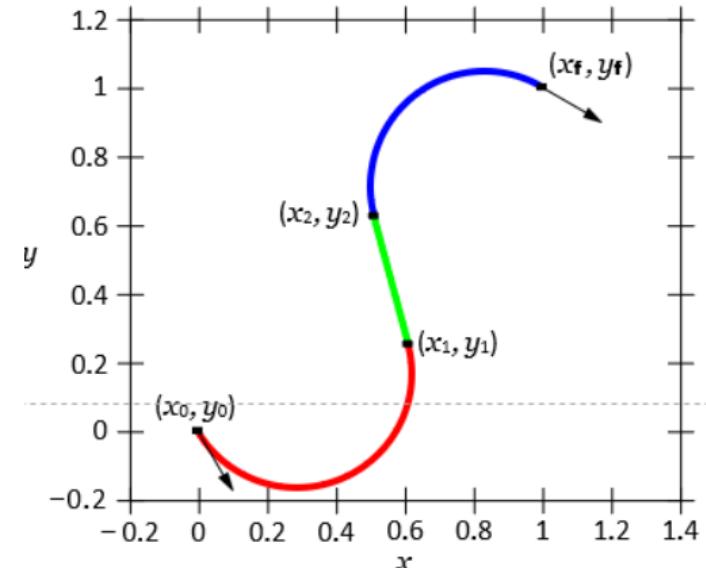
Examples

Numerical Results

Classic Markov-Dubins Problem

The Classic Markov-Dubins Path:

- ▶ The shortest C^1 and piecewise C^2 curve $\gamma : [0, L] \rightarrow \mathbb{R}^2$ between two assigned configurations: initial (P_i, ϑ_i) and final (P_f, ϑ_f) .
- ▶ It is constrained by a minimum turning radius: the absolute value of the curvature of the path γ at almost every point is not greater than $\kappa > 0$.



Multipoint Markov-Dubins Problem (MPMDP)

- ▶ The **Multipoint Markov-Dubins Problem (MPMDP)** generalises the classic problem to a sequence of assigned planar points P_0, \dots, P_n (where $P_0 = P_i$ and $P_n = P_f$) using classic Dubins interpolation.
- ▶ Even for a small number of points, the brute-force method of trying all cases for each pair of consecutive points becomes infeasible because of their **exponential number**.
- ▶ Moreover, the joining angles are new unknowns of the problem.



Outline

Problem Definition

Problem Definition

Iterative Dynamic Programming Solution

Refinement

Examples

Numerical Results

Iterative Dynamic Programming solution

- ▶ Let P_0, \dots, P_n be the sequence of assigned points to be interpolated with a Markov-Dubins curve.
- ▶ Define the function $D_j(\vartheta_j, \vartheta_{j+1})$ as the length of the optimal solution of the two points Markov-Dubins problem.
- ▶ $D_j(\vartheta_j, \vartheta_{j+1})$ is the length of the curve connecting P_j with P_{j+1} with initial and final angles $\vartheta_j, \vartheta_{j+1}$.
- ▶ For the explicit formulas for all the possible cases, we refer the reader to the previous lectures.

Iterative Dynamic Programming solution

Minimisation Problem

The solution to the Multipoint Markov-Dubins Problem is given by the optimal angles $\vartheta_0, \dots, \vartheta_n$ that minimise the total length L of the path:

$$L^* = \min_{\vartheta_0, \dots, \vartheta_n} \sum_{j=0}^{n-1} D_j(\vartheta_j, \vartheta_{j+1}).$$

- ▶ The above formulation considers the angles ϑ_j in the continuous interval $[0, 2\pi)$.
- ▶ A practical numerical implementation adopts a *discrete subset* $\Theta_j \subset [0, 2\pi)$.

Naïve bruteforce approach

- ▶ It is possible to enumerate exhaustively all the combinations of intermediate angles.
- ▶ Then, solve the associated two points problem between each pair of consecutive points and angles, and pick the shortest solution.

Computational Complexity

This has complexity $\mathcal{O}(k^n)$, where $k \in \mathbb{N}$ is the number of discretised angles in Θ_j , supposed to be the same for all j .

Iterative Dynamic Programming solution

Sub-problem Definition

Define the function $L(j, \vartheta_j)$ as the length of the solution of the sub-problem, from point P_j , with angle ϑ_j , onwards:

$$L(j, \vartheta_j) := \min_{\vartheta_{j+1}, \dots, \vartheta_n} (D_j(\vartheta_j, \vartheta_{j+1}) + \dots + D_{n-1}(\vartheta_{n-1}, \vartheta_n)).$$

Iterative Dynamic Programming solution

Recursive Relation

The recursive nature of the problem is suitable for **IDP**:

$$L(j, \vartheta_j) = \min_{\vartheta_{j+1}} (D_j(\vartheta_j, \vartheta_{j+1}) + L(j + 1, \vartheta_{j+1}))$$

which is defined piecewise as:

$$L(j, \vartheta_j) = \begin{cases} \min_{\vartheta_n} D_{n-1}(\vartheta_{n-1}, \vartheta_n) & \text{for } j = n - 1 \\ \min_{\vartheta_{j+1}} (D_j(\vartheta_j, \vartheta_{j+1}) + L(j + 1, \vartheta_{j+1})) & \text{otherwise} \end{cases}$$

- ▶ This is computed for $j = n - 1, \dots, 0$ with $L(n, \vartheta_n) = 0$ (i.e., the length of the portion of the path after the last point).

Iterative Dynamic Programming solution

- ▶ The first step requires to compute $|\Theta^{n-1}| \cdot |\Theta^n| = k^2$ two points Markov-Dubins problems, which can be processed also by **parallel computation**.
- ▶ Then, iteratively, compute $D_j(\vartheta_j, \vartheta_{j+1}) + L(j+1, \vartheta_{j+1})$ for $j = n-2, n-3, \dots, 2, 1, 0$.
- ▶ These computations require the solution of k^2 two points Markov-Dubins manoeuvres each.

Overall Computational Complexity

The overall computational complexity of this method is $\mathcal{O}(nk^2)$.



Outline

Problem Definition

Problem Definition

Iterative Dynamic Programming Solution

Refinement

Examples

Numerical Results

Refinement step

- ▶ To keep the computational time contained, the number k of angle discretisations is kept small, which leads to a first **rough suboptimal solution**.
- ▶ To recover the optimal solution up to the desired approximation error, it is possible to apply an **iterative refinement** of the solution.

Refinement step

- ▶ An improved solution is given by updating the subsets Θ_j with new values sampled around the previous solution ϑ_j .
- ▶ Let $h = 2\pi/k$ be the granularity between the values in the first discretisation.
- ▶ The heuristic choice is to sample k angles uniformly around ϑ_j on the range $[\vartheta_j - \frac{3}{2}h, \vartheta_j + \frac{3}{2}h]$.
- ▶ This refinement step can be further applied several times, until the desired accuracy of the angles is achieved.

Refinement step

- ▶ The value of h is updated at every refinement.
- ▶ After m steps, the new granularity is $h = \frac{2\pi}{3^m(2k)^m} \rightarrow 0$ for $m \rightarrow \infty$.

Global Computational Complexity

The global computational complexity is $\mathcal{O}(nmk^2)$, where n is the number of points, k the granularity of the angle discretisation and m the number of refinement steps.



Outline

Problem Definition

Problem Definition

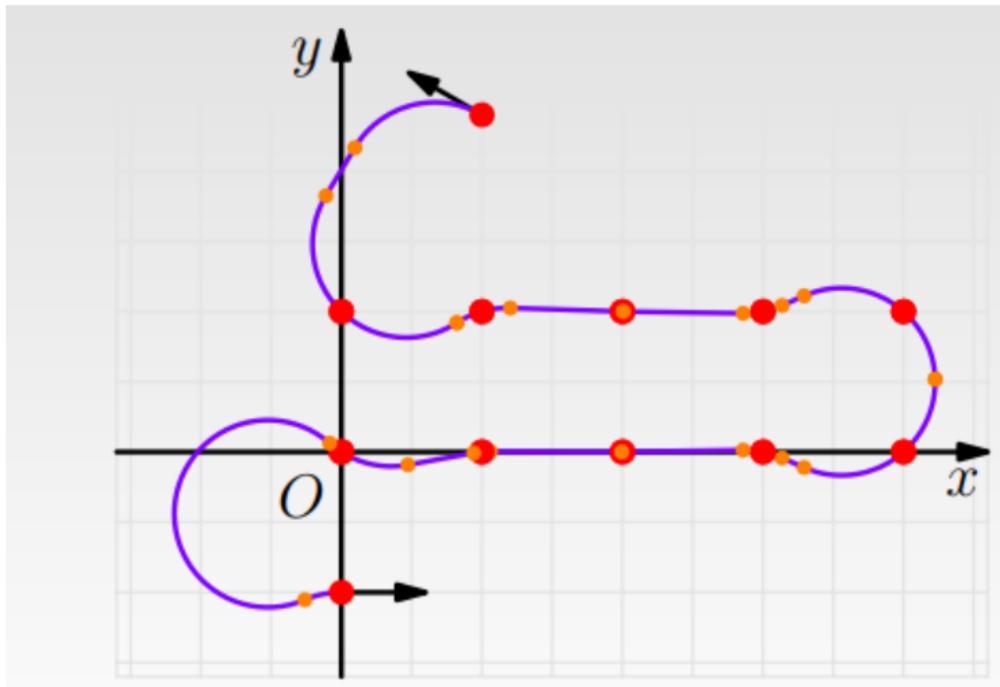
Iterative Dynamic Programming Solution

Refinement

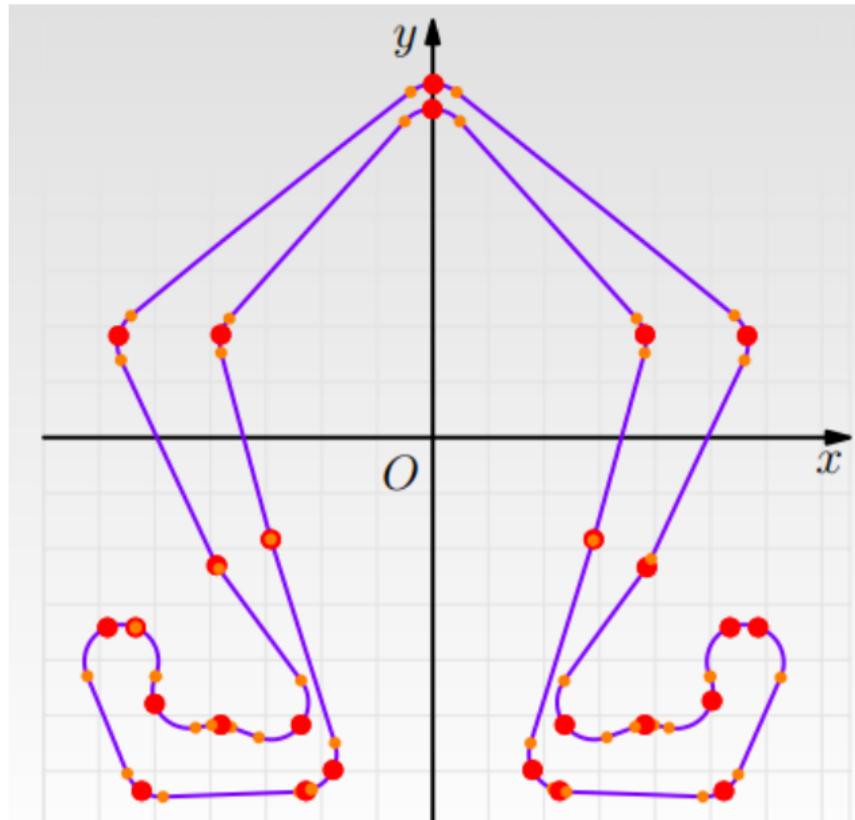
Examples

Numerical Results

Example 4: $n = 12$ points and $\kappa = 3$



Example Ω : $n = 27$ points and $\kappa = 3$



Example F1 track of Spa: $n = 701$ points and $\kappa = 3$





Outline

Problem Definition

Problem Definition

Iterative Dynamic Programming Solution

Refinement

Examples

Numerical Results

Numerical Results - Time in ms, Error in mm

Table of Results: Time in ms, Error in mm. k is the granularity of the angle discretisation and m is the number of refinement steps.

k	m	Ex. 4		Ex. Ω		Ex. Spa	
		Error	Time	Error	Time	Error	Time
4	1	10^3	0.74	$2 \cdot 10^3$	0.82	90	12
4	4	0.69	1.7	10^3	2.7	1.2	27
4	8	0.01	2.8	140	4.0	$6 \cdot 10^{-4}$	51
4	16	0.01	5.1	140	7.0	$5 \cdot 10^{-8}$	94
16	1	2.3	2.0	$2 \cdot 10^3$	3.7	2.7	67
16	4	0.49	4.6	10^3	7.7	10^{-6}	170
16	8	0.49	8.0	10^3	13	$2 \cdot 10^{-8}$	310
16	16	0.49	15	10^3	24	$2 \cdot 10^{-8}$	560
90	1	$5 \cdot 10^{-3}$	21	2.8	49	$2 \cdot 10^{-3}$	103
90	4	$2 \cdot 10^{-3}$	49	1.2	120	$5 \cdot 10^{-9}$	$3 \cdot 10^3$
90	8	$2 \cdot 10^{-3}$	91	1.2	220	$5 \cdot 10^{-9}$	$5 \cdot 10^3$
90	16	$2 \cdot 10^{-3}$	170	1.2	420	$5 \cdot 10^{-9}$	10^4
360	1	$7 \cdot 10^{-5}$	250	0.12	680	$9 \cdot 10^{-6}$	10^4
360	4	$2 \cdot 10^{-9}$	640	$3 \cdot 10^{-9}$	10^3	0	$4 \cdot 10^4$
360	8	0	10^3	0	$3 \cdot 10^3$	0	$8 \cdot 10^4$
360	16	0	$2 \cdot 10^3$	0	$5 \cdot 10^3$	0	10^5



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Collision --- Detection

Luigi Palopoli



Collision Detection

- Fundamental task in many motion planning algorithms and tools (e.g., sampling-based planning, path smoothing).
- Collision checks are repeated a large number of times during exploration of the configuration space.
- In most planning applications, the majority of computation time is spent on collision checking.



Therefore, developing efficient collision detection modules is extremely important for motion planning



Collision Detection - Formalities

- Collision detection can be modeled as the implementation of a function that determines whether two bodies occupy intersecting regions in space.

$$\phi : C \rightarrow \{\text{TRUE}, \text{FALSE}\}$$

C : The domain is the configuration space

$$\phi(q) = \begin{cases} \text{TRUE} & q \in C_{\text{obs}} \\ \text{FALSE} & \text{otherwise} \end{cases}$$

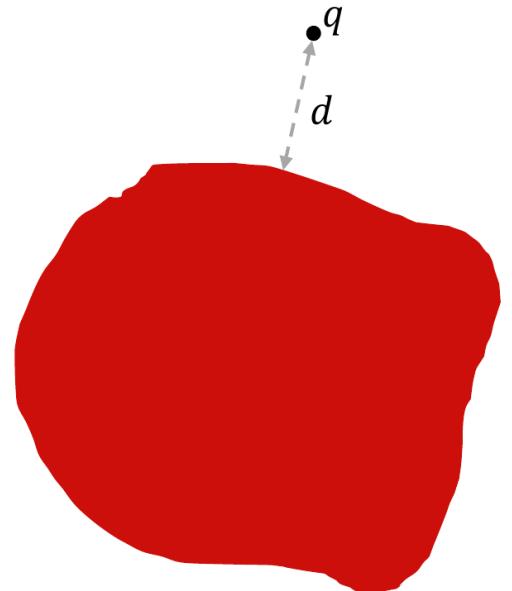


Distance

- In many motion planning applications, it is also important to know the distance between the robot and the closest obstacle.
- A distance function providing this information can be defined as:

$$d_A : C \rightarrow [0, +\infty)$$

It maps each point to a number indicating the distance between the point and the closest point in the set A





Computation of the distance

- Generally, computing the distance between two sets A and B requires solving an optimization problem.
- If the two sets are convex polyhedra, this leads to a quadratic convex optimization problem. However, in practice, simpler and more efficient methods are often sufficient.

$$\min_{x,y} \|x - y\|^2$$

$$x \in A$$

$$y \in B$$

If the two sets are convex Polyhedra, we have the following quadratic convex optimisation problem

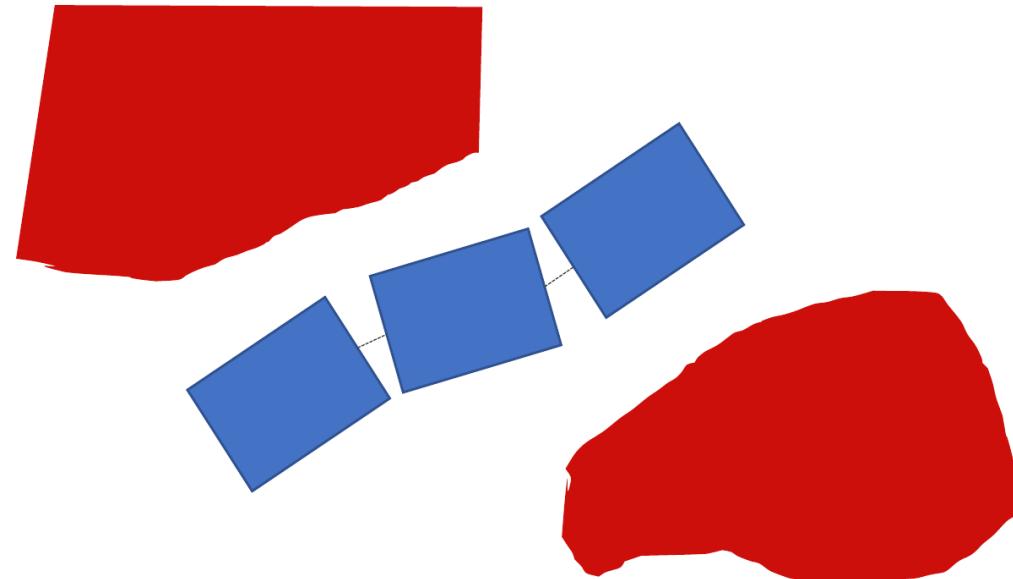
$$\left\{ \begin{array}{l} \min_{x,y} \frac{1}{2} \|x - y\|^2 \\ A_1 x \leq q_1 \\ A_2 y \leq q_2 \end{array} \right. \rightarrow \left\{ \begin{array}{l} \min_{x,y} \frac{1}{2} [x^T y^T] \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}^T \begin{bmatrix} x \\ y \end{bmatrix} \\ A_1 x \leq q_1 \\ A_2 y \leq q_2 \end{array} \right.$$

Often we can get away with much less effort



Two-phases Collision Detection

- In motion planning, the robot is typically modeled as a sequence of links $\underbrace{A_1, A_2, \dots, A_m}_{\text{Different Sets}}$, and the environment as a set of obstacles O_1, O_2, \dots, O_p





Two-phase collision detection

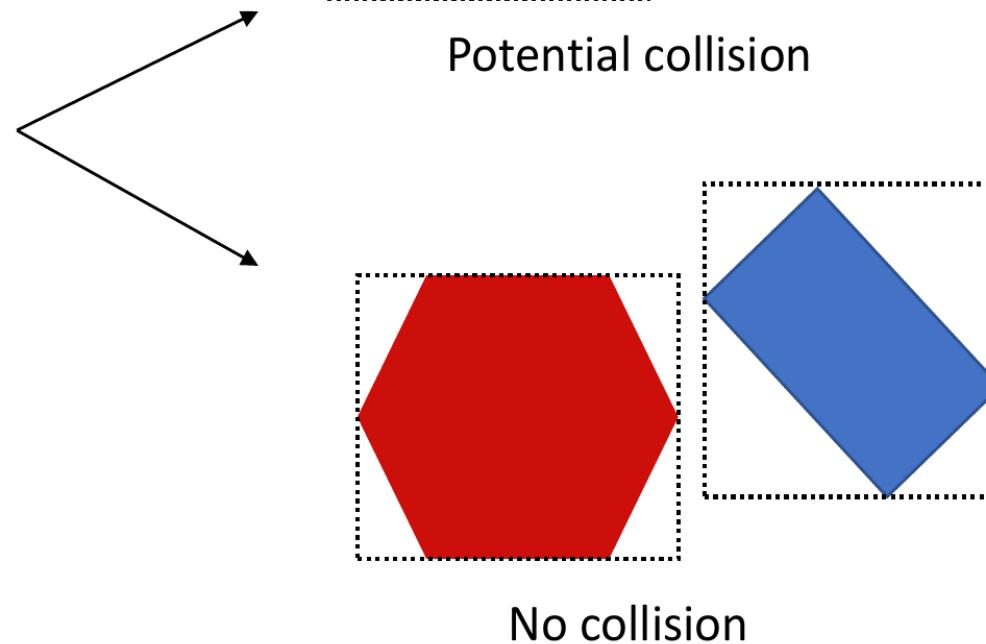
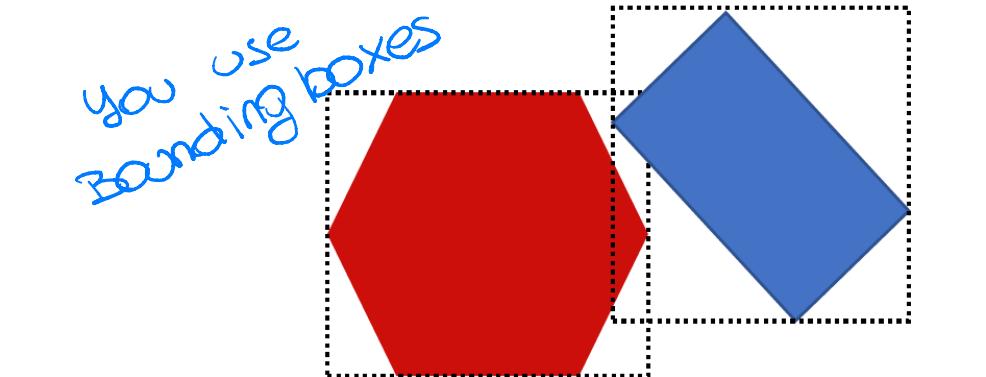
To handle complex scenarios efficiently, collision detection is often divided into two phases:

1. Broad Phase

- Efficient but approximate (conservative) identification of body pairs that may be in collision.
- These pairs are passed to the next phase for detailed checking.

2. Narrow Phase

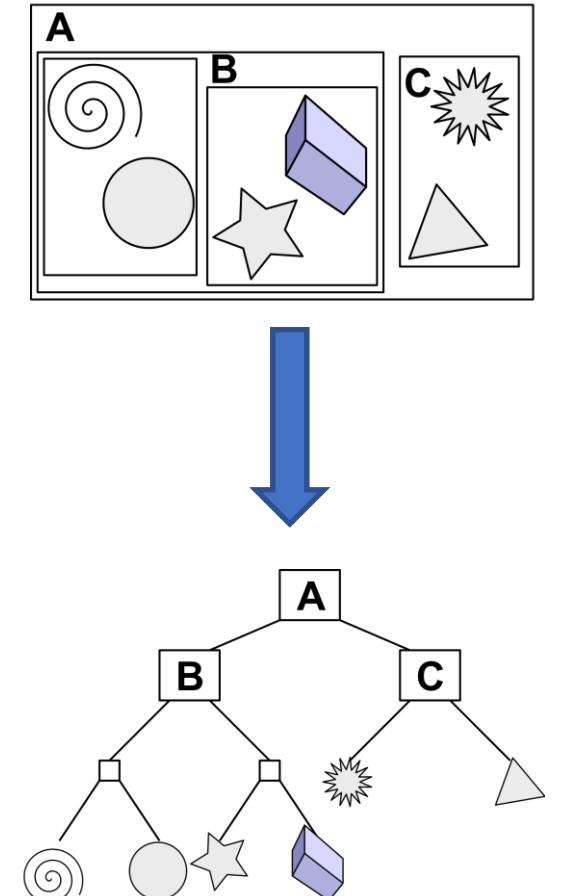
- accurate collision tests between the pairs identified in the Broad Phase.





Hierarchical Methods

- Assume two complicated bodies E, F to check for collision
- Each body is a subset of \mathbb{R}^2 (or \mathbb{R}^3) defined using any kind of geometric primitive (e.g. triangles)
- Hierarchical methods decompose each body into a tree
- Each vertex of the tree represents a bounding region containing some subset of the body
- The bounding region of the root vertex contains the whole body





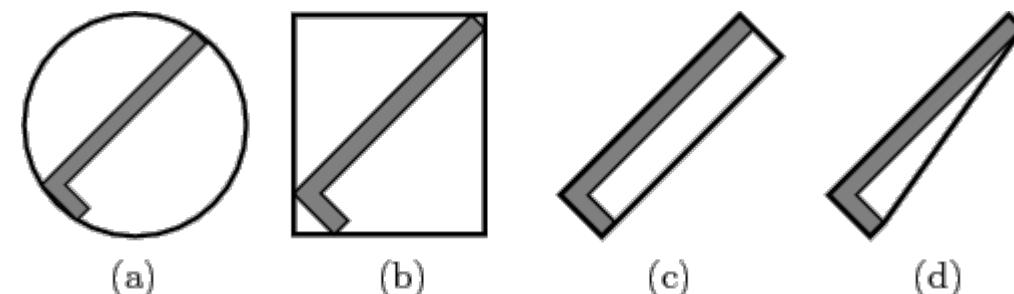
Hierarchical Methods

When selecting the type of bounding region, two opposing criteria must be balanced:

- Fit the body shape as tightly as possible.
- Maximise the efficiency of the intersection test between regions.

Common choices:

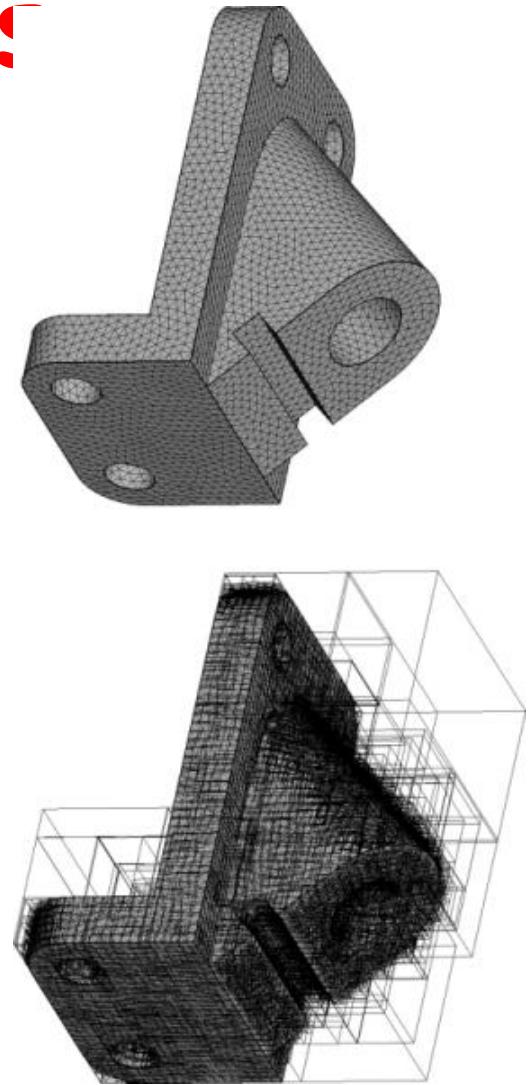
- Sphere
- Axis-Aligned Bounding Box (AABB)
- Oriented Bounding Box (OBB)
- Convex Hull





Hierarchical Method:

- Recursive construction of the tree for a body B :
 - Consider for each vertex the set X of all points of B contained in the corresponding bounding region
 - Split the set into two smaller bounding regions whose union covers X
 - The split partitions the space into two regions of similar size
 - A bounding region is then computed for each children, and the procedure is repeated recursively, until very simple sets are obtained (e.g. composed by a single primitive)





Hierarchical Methods

- Collision Detection with Hierarchical Methods:
 - *Problem: determine if two bodies E and F are in collision*
 - Suppose that the trees T_e and T_f have been constructed for E and F
 - If the bounding regions of root vertices of T_e and T_f do not intersect, then E and F are not in collision, and the procedure stops
 - Otherwise, the procedure must be repeated by trying to intersect the children of T_e with the children of T_f , and the process continues recursively



Hierarchical Methods

- Collision Detection with Hierarchical Methods:
 - As long as the bounding regions overlap, lower levels of the trees are traversed, until the leaves are reached
 - When we need to check for collision two leaf nodes, the narrow phase, more accurate collision detection algorithm is applied
 - Notice that, while trees are traversed, if a bounding region from the vertex v_1 of T_e does not intersect the bounding region from a vertex v_2 of T_f , then no children of v_1 have to be compared to children of v_2
 - In general, this approach drastically reduces the number of comparisons relative to a naive approach that tests all pairs of primitives for intersection

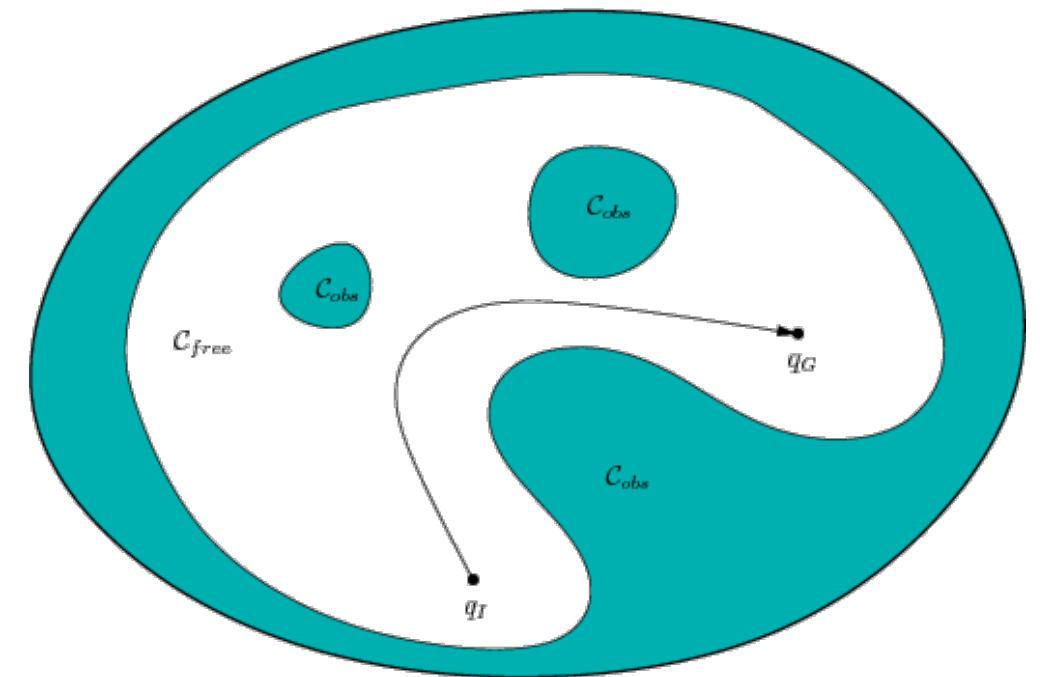


Checking a Path Segment

- Collision detection algorithms can determine whether
$$q \in C_{\text{free}}$$
- Motion planning algorithms require that an entire path maps into C_{free}
- Often the motion planner requires the validation of a (short) path segment,

Given

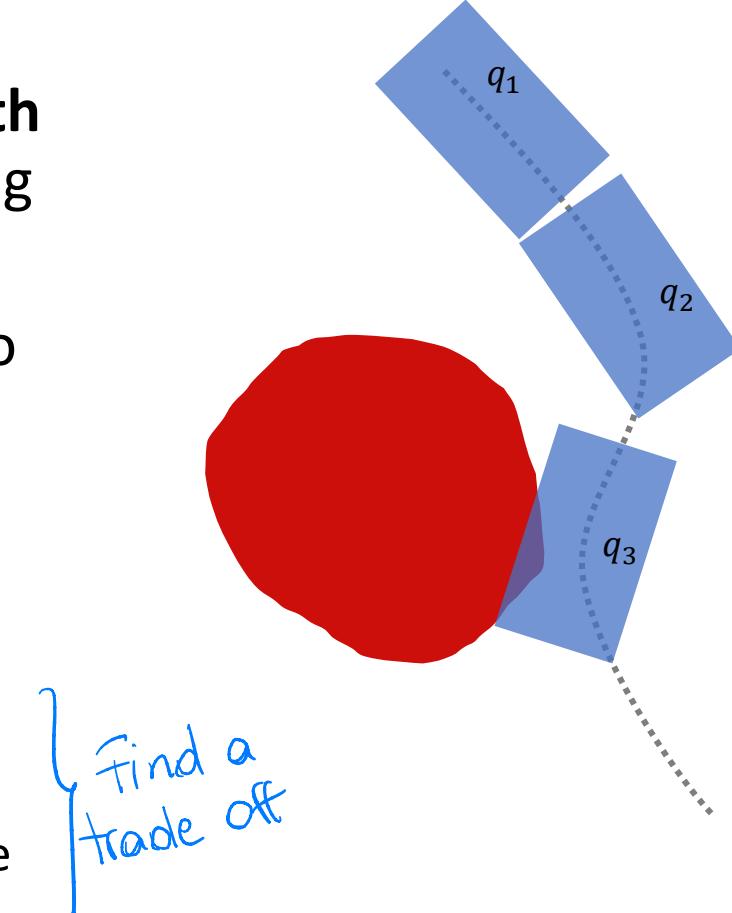
$\tau : [0, 1] \rightarrow C$
determine whether
 $\tau([0, 1]) \subset C_{\text{free}}$





Checking a Path Segment

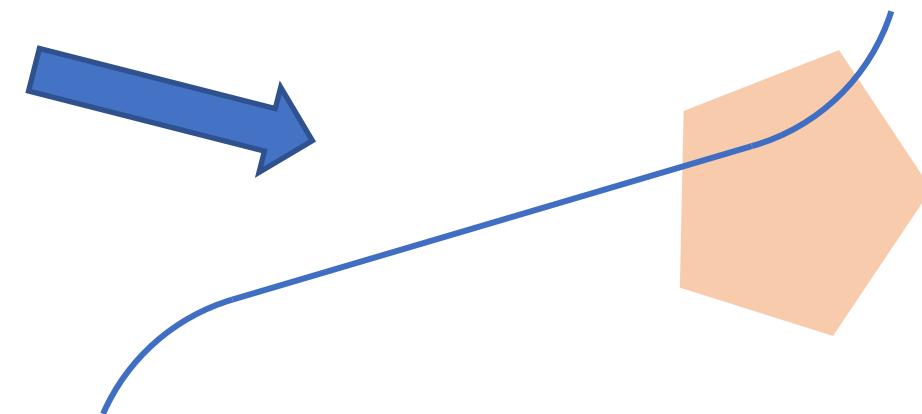
- Collision detection can also be applied to a **path segment** to determine whether any point along the path intersects an obstacle.
- In general scenarios, a common approach is to sample the interval $[0,1]$ and call the collision checker only on the samples
- The step size Δq is often determined experimentally:
 - Too small values increase considerably the computation time spent on collision checking
 - Too large values may produce wrong results, where the path is allowed to jump through thin obstacles





Example Scenario: Dubins Curves

- A Dubins curve is a sequence of line segments and arcs of circle
- **Task:** Check whether a point moving along a Dubins curve collides with any polygonal obstacles in the map.
- The narrow phase of this test for intersection requires to check for intersection between:
 1. Pairs of segments
 2. A segment and an arc of circle





Example Scenario: Dubins Curves

- Detect intersections between pairs of line segments.

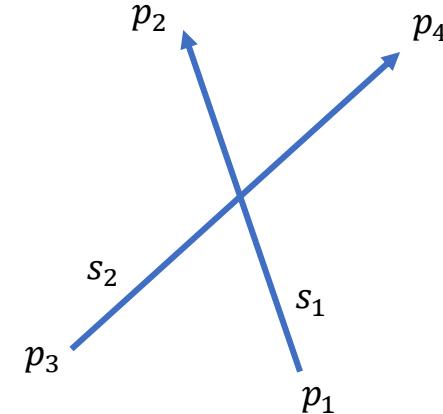
$$p_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \quad p_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

$$p_3 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix}, \quad p_4 = \begin{bmatrix} x_4 \\ y_4 \end{bmatrix}$$

$$s_1 : p_1 + t(p_2 - p_1), \quad t \in [0, 1]$$

$$s_2 : p_3 + u(p_4 - p_3), \quad u \in [0, 1]$$

$$s_1 = s_2 \rightarrow \begin{cases} x_1 + t(x_2 - x_1) = x_3 + u(x_4 - x_3) \\ y_1 + t(y_2 - y_1) = y_3 + u(y_4 - y_3) \end{cases} \rightarrow \begin{bmatrix} (x_4 - x_3) & -(x_2 - x_1) \\ (y_4 - y_3) & -(y_2 - y_1) \end{bmatrix} \begin{bmatrix} u \\ t \end{bmatrix} = \begin{bmatrix} x_1 - x_3 \\ y_1 - y_3 \end{bmatrix}$$





Example Scenario: Dubins Curves

- Detect intersections between pairs of line segments.

$$p_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \quad p_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \quad p_3 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} \quad p_4 = \begin{bmatrix} x_4 \\ y_4 \end{bmatrix}$$

$$\begin{bmatrix} x_4 - x_3 & -(x_2 - x_1) \\ y_4 - y_3 & -(y_2 - y_1) \end{bmatrix} \begin{bmatrix} u \\ t \end{bmatrix} = \begin{bmatrix} x_1 - x_3 \\ y_1 - y_3 \end{bmatrix}$$



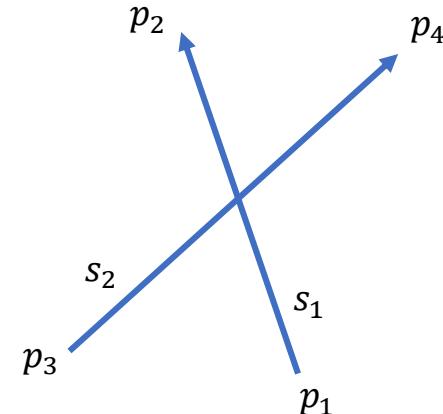
$$\det = \begin{vmatrix} x_4 - x_3 & -(x_2 - x_1) \\ y_4 - y_3 & -(y_2 - y_1) \end{vmatrix} = (x_4 - x_3)(y_1 - y_2) - (x_1 - x_2)(y_4 - y_3)$$

$$t = \frac{(y_3 - y_4)(x_1 - x_3) + (x_4 - x_3)(y_1 - y_3)}{\det}$$

$$u = \frac{(y_1 - y_2)(x_1 - x_3) + (x_2 - x_1)(y_1 - y_3)}{\det}$$

Actual intersection when $t \in [0,1]$, $u \in [0,1]$ (i.e. within segment boundaries)

when $\det = 0$, the segments are collinear





Example Scenario: Dubins Curves

- Segment-Circle intersection:

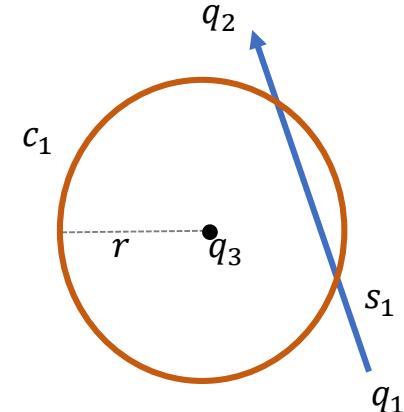
$$q_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \quad q_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \quad q_3 = \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

$$s_1 : q_1 + t(q_2 - q_1), \quad t \in [0,1]$$



$$\begin{aligned} s_{1x} &: x_1 + t(x_2 - x_1) \\ s_{1y} &: y_1 + t(y_2 - y_1) \end{aligned}$$

$$c_1 : (x - x_c)^2 + (y - y_c)^2 = r^2$$



find t such that $s_1 = c_1$



Example Scenario: Dubins Curves

We have to solve the second degree equation

$$at^2 + 2bt + c$$

where

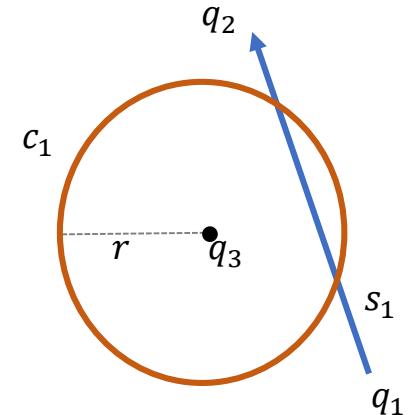
$$a = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

$$b = (x_2 - x_1)(x_1 - x_c) + (y_2 - y_1)(y_1 - y_c)$$

$$c = (x_1 - x_c)^2 + (y_1 - y_c)^2 - r^2$$

and

$$t_{1,2} = \frac{-b \pm \sqrt{b^2 - ac}}{a}$$



Actual intersections are when $t_i \in [0, 1]$

There can be **0, 1, or 2** intersections between a segment and a circle, depending on their relative geometry.



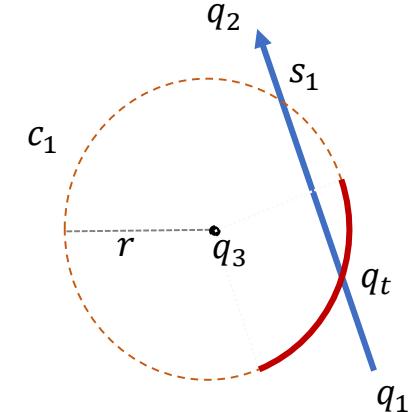
Example Scenario: Dubins Curves

- Segment-Circle Arc intersection:

$$q_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

$$q_2 = \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

$$q_3 = \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

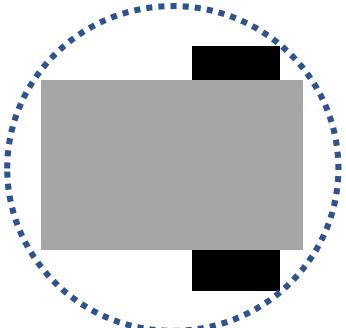


- Possible to use Segment-Circle intersection
- When an intersection $q_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}$ is found:
 - Determine angle θ_t of q_t with respect to center: $\text{atan2}(y_t - y_c, x_t - x_c)$
 - check if point at angle θ_t is part of the circle arc

FINE

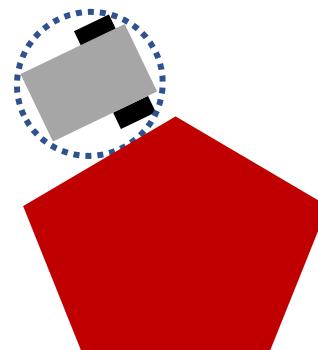


Polygon Offsetting



The robot geometry can be approximated by its **circumscribing circle**.

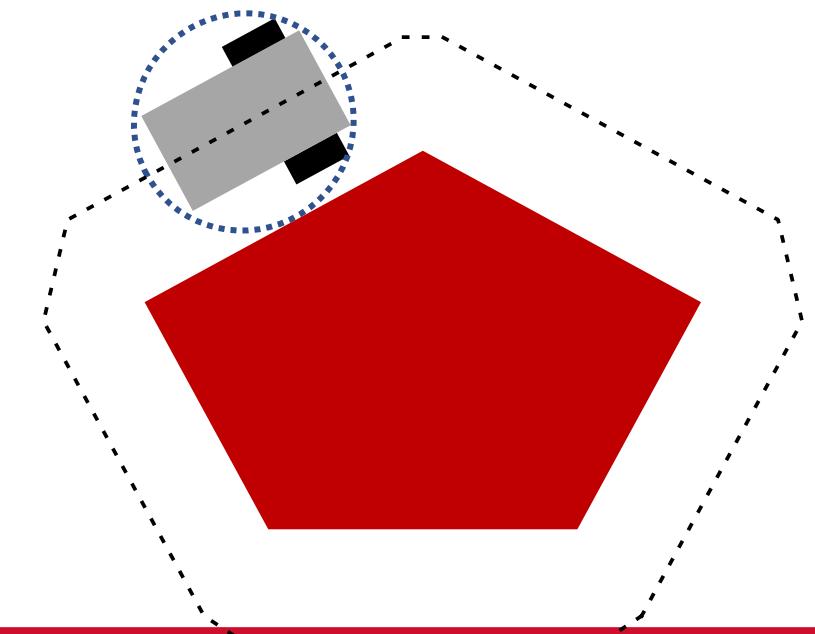
The robot *touches* a polygonal obstacle when the distance from the circle's center to the nearest edge equals the circle's radius.





Polygon Offsetting

- An effective method for collision detection in this case is **polygon offsetting**:
 1. Expand (offset) the obstacles by a size equal to the robot radius
 2. Treat the robot as a **point** moving along the path (The robot's geometry is already accounted for in the offset polygons.)

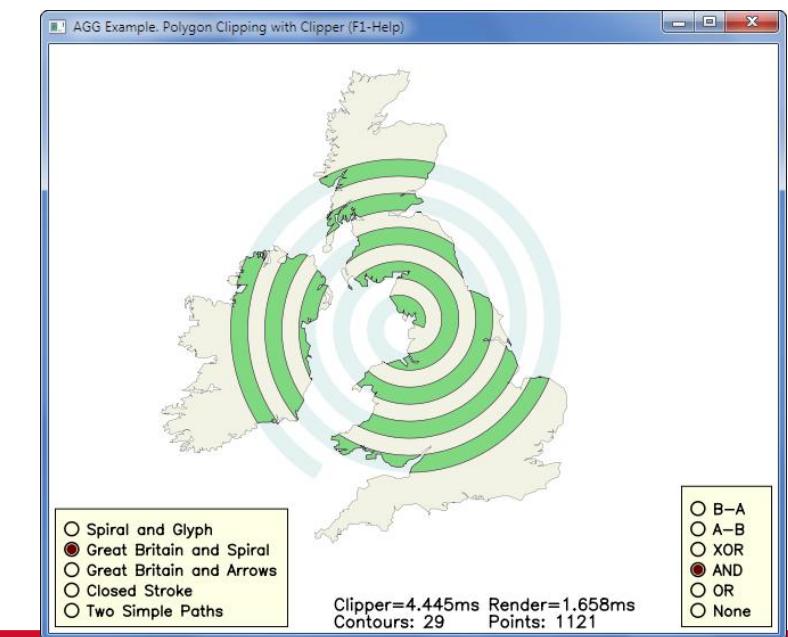




Polygon Offsetting

- A robust implementation of polygon offsetting is provided by the **Clipper library**:
<http://www.angusj.com/delphi/clipper.php>

- Open source library for clipping and offsetting lines and polygons
- Support for different operations: intersection, union, difference, exclusive-or
- Supported languages: Delphi, C++, C#, ...





Using Clipper

- Include the Clipper headers.
- A **Path** represents a polyline or polygon.
- A **Paths** object represents a collection of Path objects.
- Add points to a Path using the `<<` operator.
(Points are represented with integer coordinates.)



Polygon Offsetting

```
#include "clipper/clipper.hpp"           → Include Clipper headers  
  
ClipperLib::Path srcPoly;                → A Path represents a polyline or a polygon  
  
ClipperLib::Paths newPoly;               → Class Paths represent a collection of Path objects  
  
  
const double INT_ROUND = 1000.;  
for (size_t i = 0; i<pointsX.size(); ++i) {  
    int x = pointsX[i] * INT_ROUND;  
    int y = pointsY[i] * INT_ROUND;  
    srcPoly << ClipperLib::IntPoint(x, y);  
}
```

Add the list of points to the Path object using operator <<

Points are represented with Integer coordinates



Polygon Offsetting

```
ClipperLib::ClipperOffset co;  
if (pointsX.size() == 3) {  
    co.AddPath(srcPoly, ClipperLib::jtSquare, ClipperLib::etClosedLine);  
} else {  
    co.AddPath(srcPoly, ClipperLib::jtSquare, ClipperLib::etClosedPolygon);  
}
```

A ClipperOffset object provides the methods to offset a given (open or closed) path

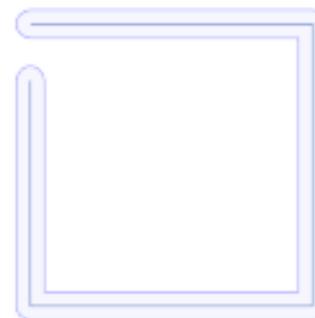
Supported EndTypes



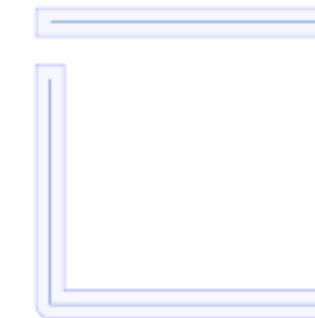
etClosedPolygon



etClosedLine



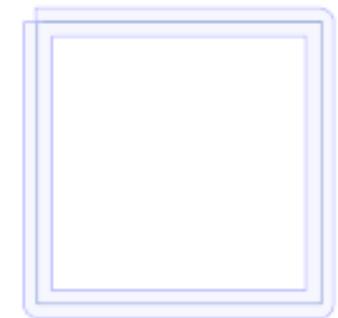
etOpenRound



etOpenSquare



etOpenButt



etOpenButt



Polygon Offsetting

```
co.Execute(newPoly, OFFSET);
```

→ Apply the offsetting with the given size *OFFSET*, and store the resulting Paths into *newPoly*

```
for (const ClipperLib::Path &path: newPoly) {  
    // Obstacle obst = create data structure for current obstacle...  
    for (const ClipperLib::IntPoint &pt: path) {  
        double x = pt.X / INT_ROUND;  
        double y = pt.Y / INT_ROUND;  
        // Add vertex (x,y) to current obstacle...  
    }  
    // Close and export current obstacle...  
}
```

→ Iterate over the resulting Paths, and construct the new Obstacle data structures accordingly

To distinguish outer and hole polygons produced as solutions of the Execute method, it is possible to check their orientation (counter-clockwise vs clockwise). There is a method of the library, i.e.

```
bool Orientation(const Path &poly);
```

returning true for outer polygons and false for hole polygons.



Self-Check Q&A



Question 1

- What is the main purpose of collision detection in motion planning?



Answer 1

- Collision detection determines whether a robot's configuration causes it to intersect with any obstacle in its environment — ensuring that planned motions are feasible and safe.



Question 2

- Why does collision detection often dominate computation time in motion planning algorithms?



Answer 2

- Because collision checks are performed repeatedly — often thousands of times — during exploration of the configuration space, making them the most frequent and costly operation.



Question 3

- What is the difference between collision detection and distance computation?



Answer 3

- Collision detection checks whether two bodies intersect, while distance computation measures how far apart they are when not intersecting. Both are essential for safe and efficient path planning.



Question 4

- What are the two main phases of a typical collision detection process?



Answer 4

- 1. Broad Phase – Quickly identifies pairs of objects that might collide (approximate and conservative).
- 2. Narrow Phase – Performs precise intersection tests only on those pairs.



Question 5

- What are two opposing criteria when choosing a bounding region in hierarchical methods?



Answer 5

- 1. Tight Fit: The bounding volume should closely approximate the object's shape.
- 2. Efficiency: The intersection test between volumes should be computationally simple.



Question 6

- How can hierarchical bounding volumes improve collision detection performance?



Answer 6

- They organise objects into a tree of nested bounding volumes. Large regions can be quickly excluded, so only small, potentially colliding parts are checked in detail — greatly reducing computation time.



Question 7

- What types of intersections must be checked when testing Dubins curves for collisions?



Answer 7

- Segment–segment intersections and segment–circular arc intersections. These determine whether any part of the Dubins curve passes through obstacles.



Question 8

- Why can a segment–circle intersection have 0, 1, or 2 solutions?



Answer 8

- Because the line segment may miss the circle (0 intersections), touch it tangentially (1 intersection), or cross it at two points (2 intersections).



Question 9

- How does polygon offsetting simplify collision detection for circular robots?



Answer 9

- By expanding each obstacle outward by the robot's radius, the robot can be treated as a point during collision checks. The robot's geometry is already encoded in the offset obstacles.



Question 10

- What is the purpose of the Orientation() function in the Clipper library when processing offset polygons?



Answer 10

- `Orientation()` determines whether a polygon is outer (counter-clockwise) or a hole (clockwise). This distinction is necessary for correctly reconstructing the obstacle map after offsetting.



Motion Planning

Given:

- an initial configuration of the robot (e.g. 2D pose)
- a goal configuration
- a model of the robot (geometry, constraints)
- a map of the environment (obstacles, ...)



find an admissible, collision-free path that moves the robot from the start to the goal configuration



Motion Planning

Plan criteria:

- *Feasibility*: find a plan moving the robot to the goal configuration (regardless of efficiency)
- *Optimality*: in addition, the found feasible plan must be optimal w.r.t. some cost function (length, smoothness, time, ...)
 - In general, this problem is much harder than feasibility



Motion Planning

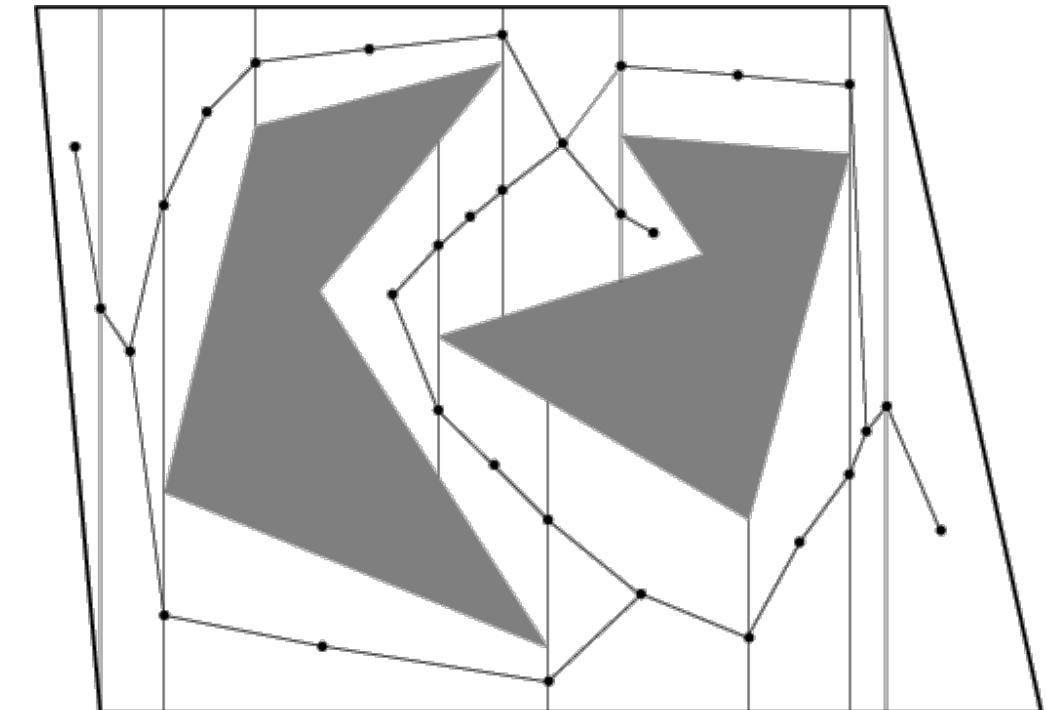
- Two different approaches to explore and discretize C -spaces:
 - *Combinatorial planning*: the connectivity of C_{free} is modelled as a graph (roadmap), used to solve the motion planning queries
 - *Sampling-based planning*: randomly and incrementally explore C_{free} , and construct a graph of sampled configurations



Combinatorial Motion Planning

Roadmap:

- all combinatorial motion planning algorithms are based on the construction of a roadmap
- Concise representation of C_{free} in form of a topological graph G , capturing its connectivity
- Each vertex v is a configuration of C_{free} , and each edge e is a collision-free path through C_{free}





Combinatorial Motion Planning

Roadmap Properties:

Let $S \subset C_{free}$ be the swath, i.e. the set of all points reached by G :

$$S = \bigcup_{e \in E} e([0,1])$$

1. Accessibility:

for every $q \in C_{free}$, simple and efficient to compute a path $\tau: [0,1] \rightarrow C_{free}$, such that $\tau(0) = q$ and $\tau(1) = s$, where $s \in S$

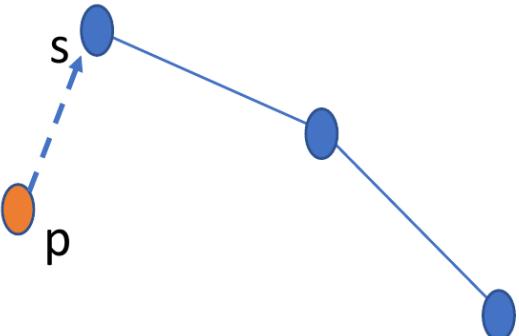
2. Connectivity-preserving:

[Using the first condition, it is always possible to connect some $q_I, q_G \in C_{free}$ to some $s_1, s_2 \in S$.]

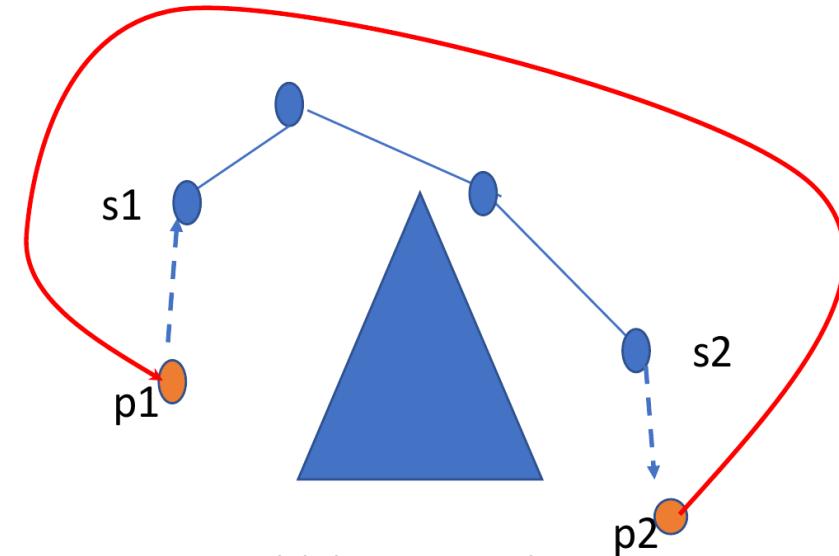
If there exists a path $\tau: [0,1] \rightarrow C_{free}$, such that $\tau(0) = q_I$ and $\tau(1) = q_G$, then there exists a path $\tau': [0,1] \rightarrow S$, such that $\tau'(0) = s_1$ and $\tau'(1) = s_2$



Intuition



Roadmap have to satisfy
these two properties



Accessibility:

From any point I can reach a point
in the roadmap

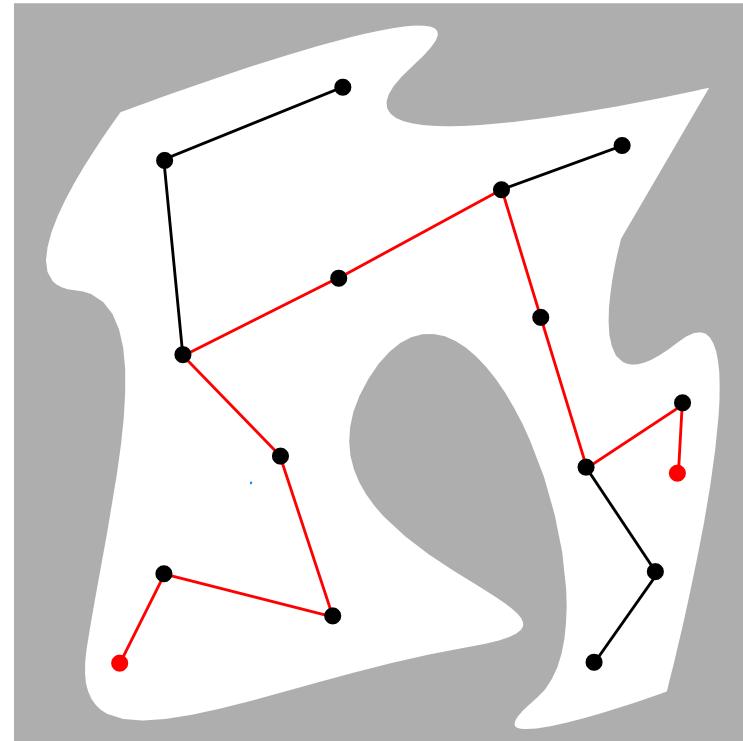
Connectivity Preserving:

If there is a free path between
two points I can also find one
based on the connectivity of
the graph

Combinatorial Motion Planning

Planning on roadmaps:

- Given a roadmap, a planner can plan paths between configurations using graph-based search





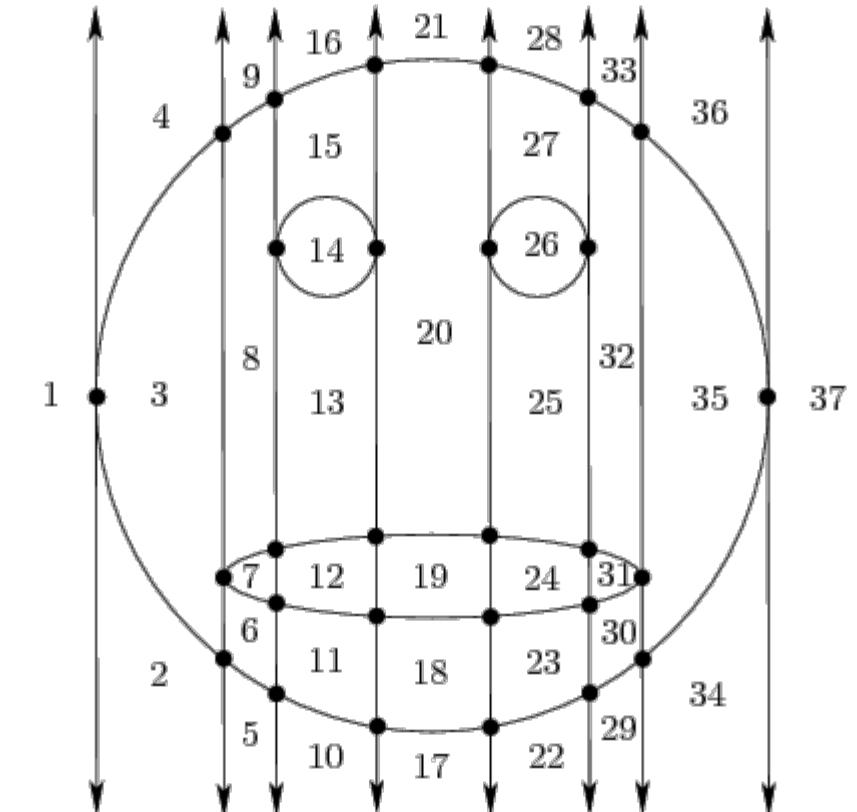
Combinatorial Motion Planning

- Main techniques:
 - Exact Cell decomposition
 - Approximate Cell decomposition
 - Maximum clearance



Exact Cell Decomposition

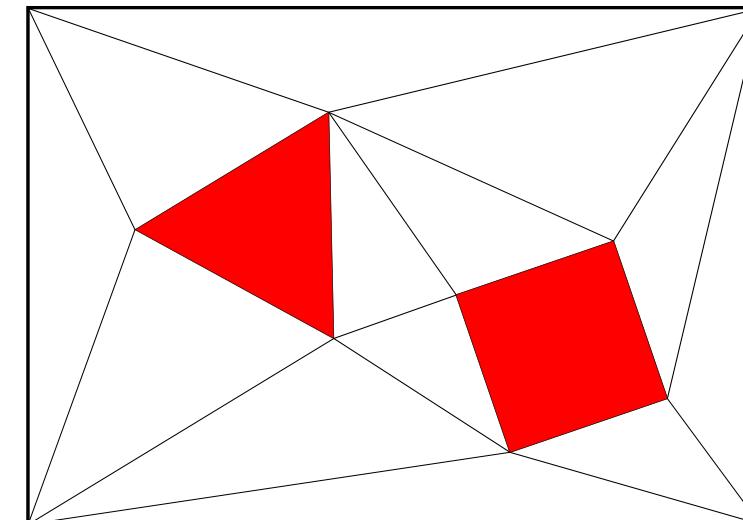
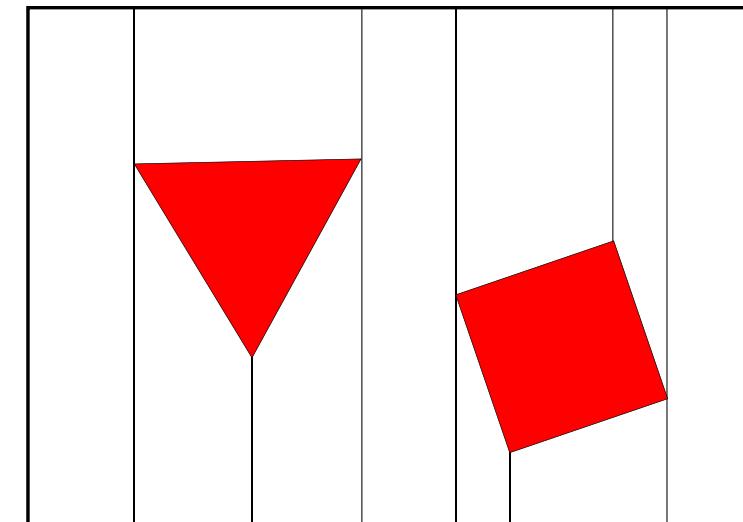
- Represent C_{free} as a collection of non-overlapping cells (connected regions of C_{free})
- The union of all the cells exactly covers C_{free} (There are no points left behind)
- Things are much simpler if we can rely on a polygonal geometry for the objects





Exact Cell Decomposition

- Cell geometry should be simple:
 - Simple to compute a path between configurations within a cell
 - Simple to test whether two cells are adjacent (share a boundary)
 - Simple to find which cell a given configuration belongs to
 - Simple to find a path crossing the shared boundary of two adjacent cells

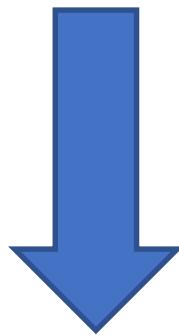




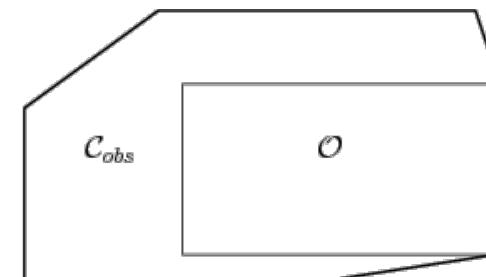
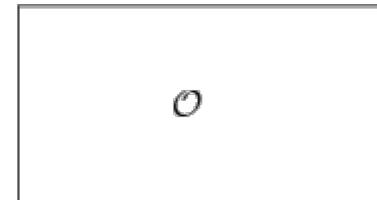
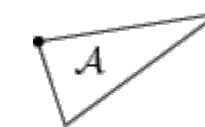
Exact Cell Decomposition

Simple Scenario:

- Assumption: robot A and all obstacles O_i are polygons, and C_{free} is bounded



c_{obs} polygonal region

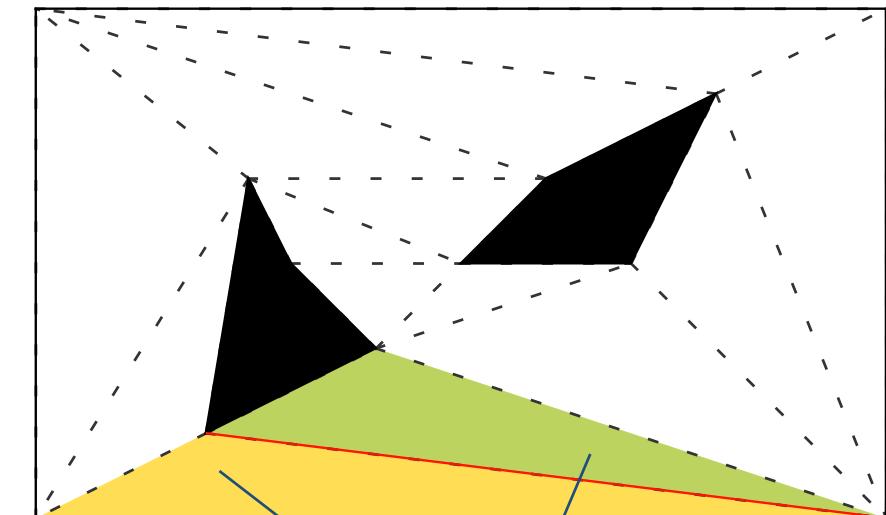




Exact Cell Decomposition

Definitions:

- A *convex polygonal decomposition* K of C_{free} is a finite collection of convex polygons (cells) such that the interiors of any two cells do not intersect and the union of all cells is C_{free}
- Two cells $k, k' \in K$ are *adjacent* iff $k \cap k'$ is a line segment of non-zero length



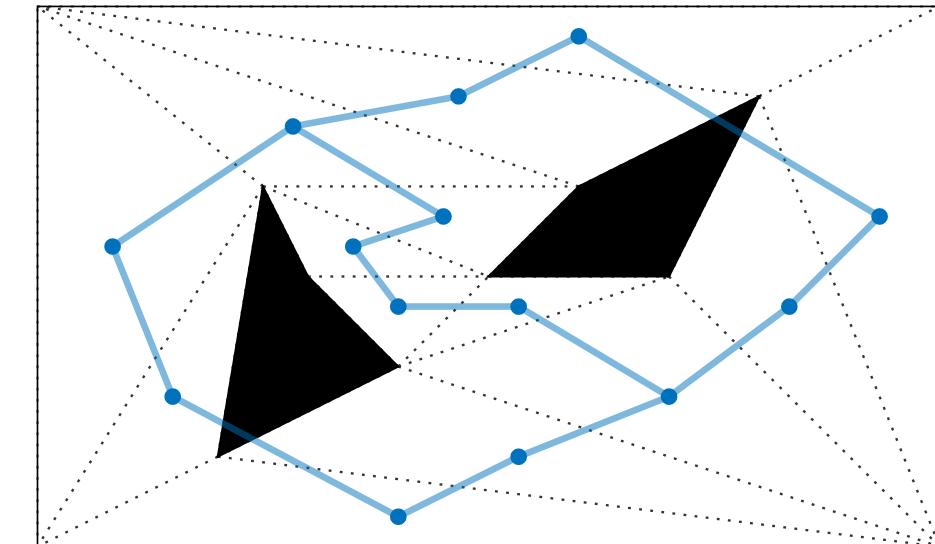
Adjacent
cells



Exact Cell Decomposition

Definitions:

- *Connectivity graph* associated with convex polygonal decomposition K of C_{free} : undirected graph where nodes correspond to cells in K , and edges between nodes in G represent the adjacency between pairs of cells in K

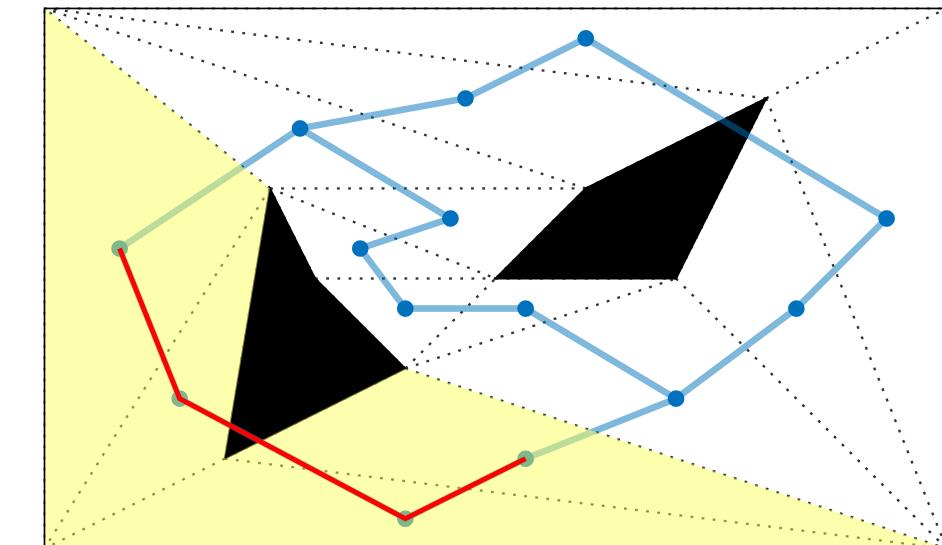




Exact Cell Decomposition

Definitions:

- *Channel* in C_{free} : sequence of adjacent cells of C_{free} , corresponding to a path in the connectivity graph





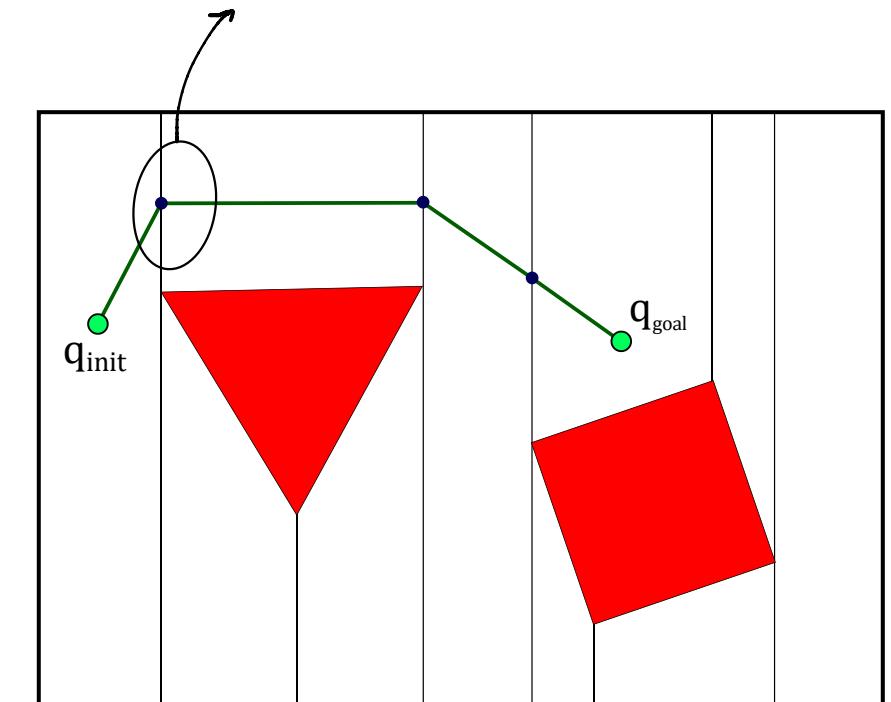
Exact Cell Decomposition

Path Planning Algorithm:

- Input: q_{init} , q_{goal} and C_{obs} (polygonal region)
 - Output: a path in C_{free} connecting q_{init} and q_{goal}
- Difficult part
1. Build K , the convex polygonal decomposition of C_{free}
 2. Construct the connectivity graph G of K
 3. Locate the cells k_{init} and k_{goal} in K containing q_{init} and q_{goal}
 4. Find a path in G between the nodes corresponding to k_{init} and k_{goal} (sequence of cells forming a channel in C_{free})
 5. Find a free path from q_{init} to q_{goal} inside the channel

→ Obstacles (red areas)

Usually are used mid points of cell's edges



(We assume that all obstacles are polygons, if its not the case we need to use convex hulls)



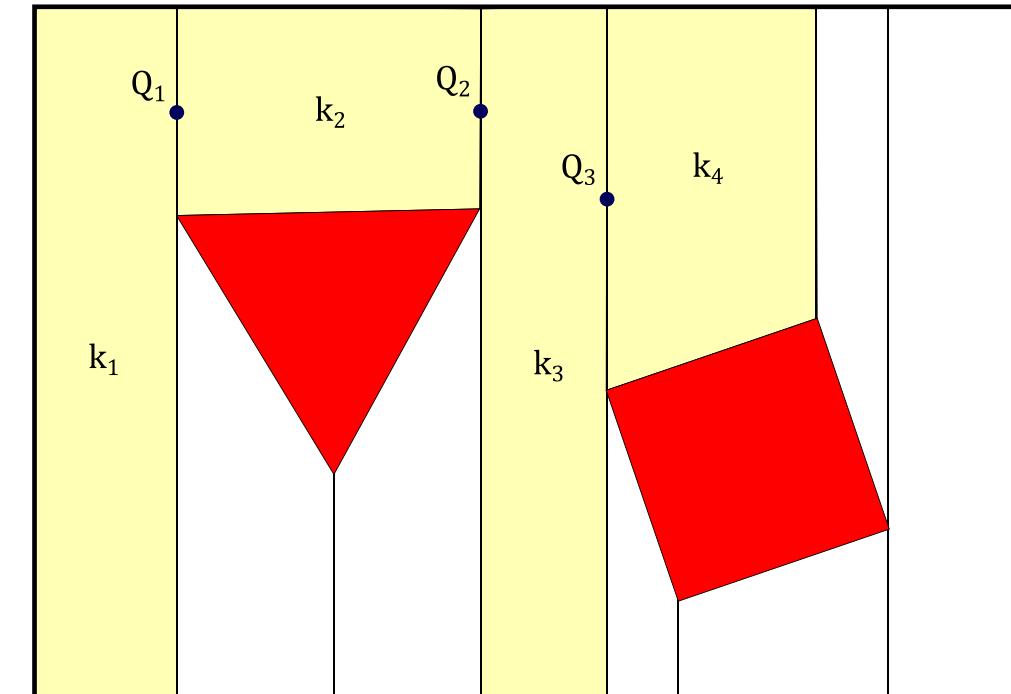
Exact Cell Decomposition

Finding a free path inside the channel:

- Let k_1, k_2, \dots, k_p denote the channel in C_{free} where $k_1 = k_{init}$ and $k_p = k_{goal}$
- Problem: find a free path from q_{init} to q_{goal} that is contained in the interior of the channel
- Notice that, since each k_i is convex, the straight line between any two configurations in k_i lies in k_i



Use midpoints of cell boundaries as crossing points between cells

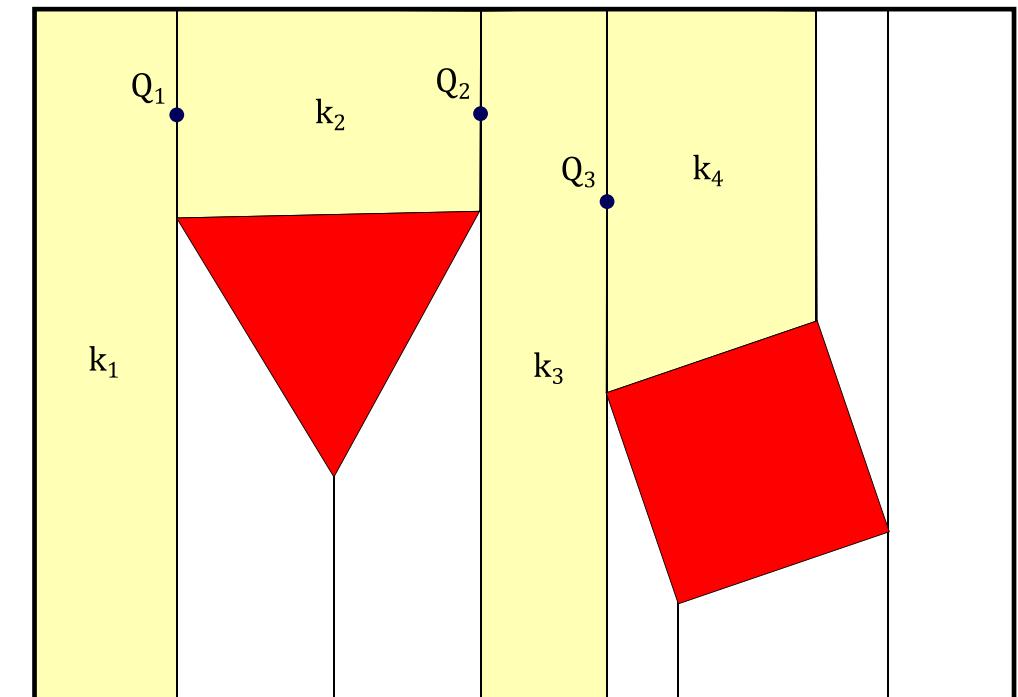




Exact Cell Decomposition

Finding a free path inside the channel:

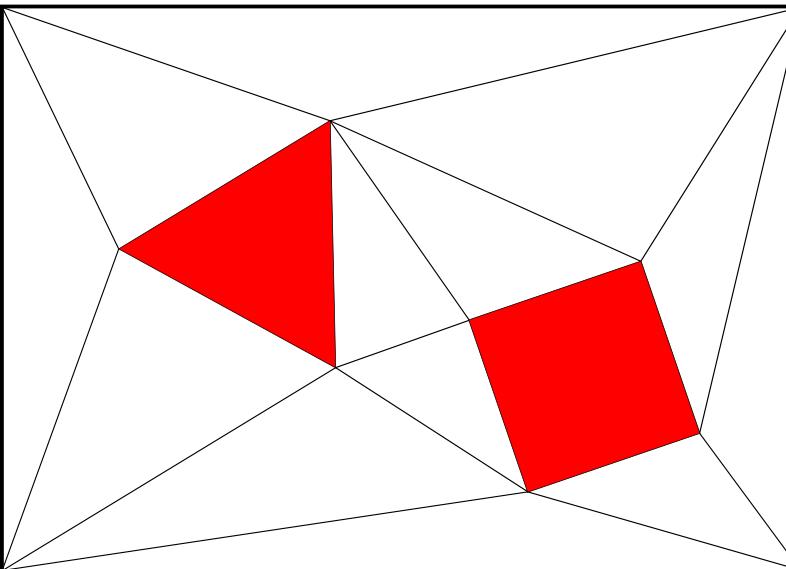
- Let Q_i denote the midpoint of line segment common to both k_i and k_{i+1} in the channel, i.e. Q_i is the midpoint of $\delta k_i \cap \delta k_{i+1}$
- Then $q_{init}, Q_1, Q_2, \dots, Q_{p-1}, q_{goal}$ is a path inside the channel



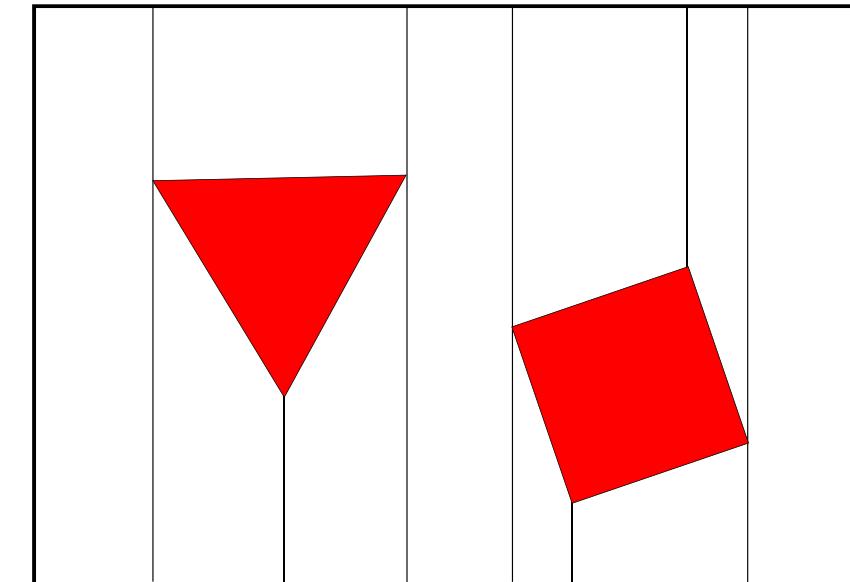


Exact Cell Decomposition

- Different possible ways to decompose C_{free} and generate the connectivity graph



Triangular decomposition



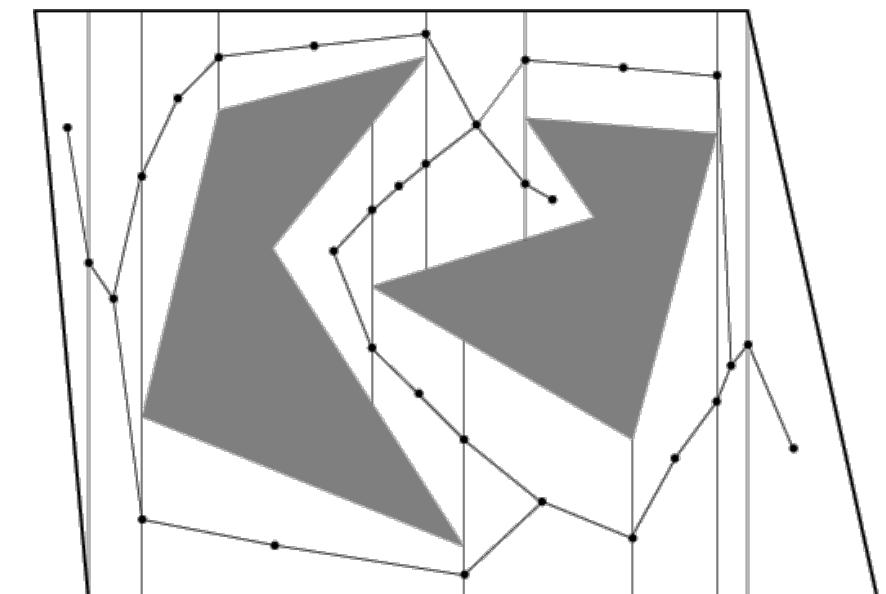
Trapezoidal decomposition

- Difficult task: NP-hard to generate minimum number of convex components for polygons with holes



Exact Cell Decomposition

- A well-known approach is based on the trapezoidal decomposition:
 1. Decompose C_{free} into trapezoids with vertical side segments by shooting rays in C_{free} upward and downward from each polygon vertex, until it touches C_{obs} or the boundary of C_{free}
 2. Place one vertex at the midpoint of every vertical segment
 3. (In addition) Place one vertex in the interior of every trapezoid, pick e.g. the centroid (avoid paths lying on the boundaries)
 4. Connect the vertices to form the adjacency graph

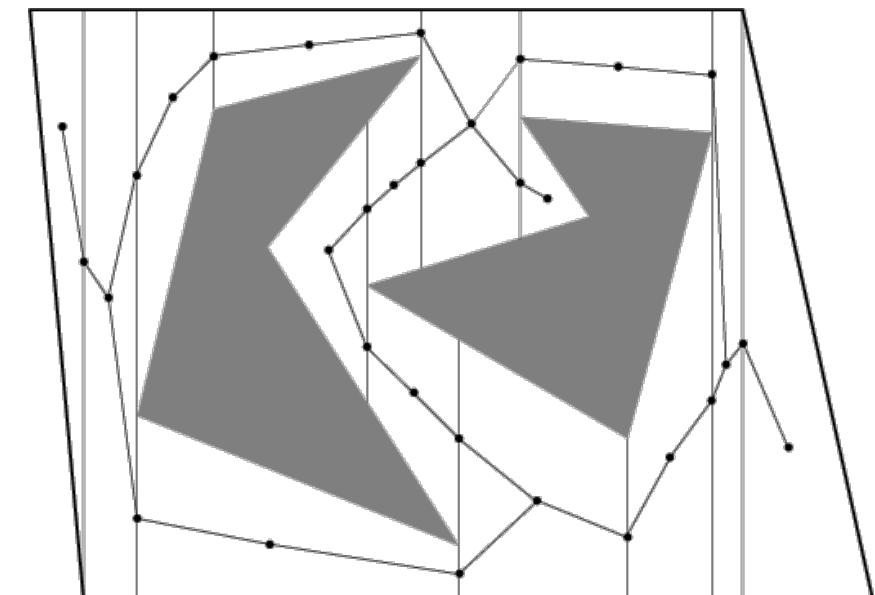




Exact Cell Decomposition

Trapezoidal Decomposition

- Brute force, naive approach:
 - For each ray (i.e. obstacle vertex), compute the intersection with all the other obstacles
 - Complexity: at least $O(n^2)$
- Line-sweep algorithm:
 - Use suitable data structures to maintain "sorted" geometric information
 - Complexity: $O(n \log n)$

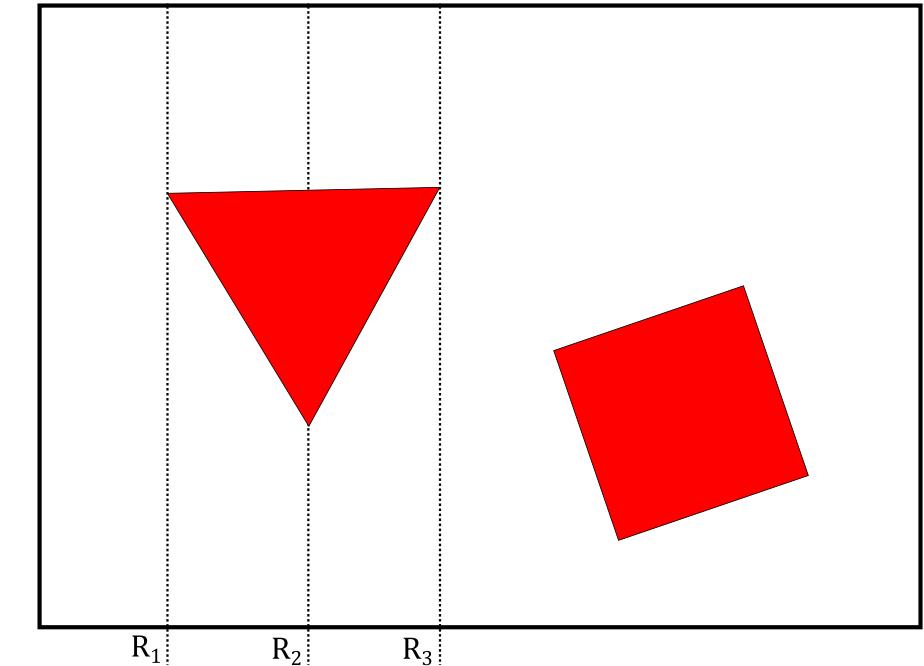




Exact Cell Decomposition

Line-sweep algorithm:

- Inspired from line-sweep principle from computational geometry (which is a generalisation of sorting in the geometric domain)
- Sort the set $P \subset \mathbb{R}^2$ of vertices of C_{obs} by increasing x coordinate
- "sweep" a vertical ray R from $x = -\infty$ to $x = \infty$, stopping at each vertex v in P

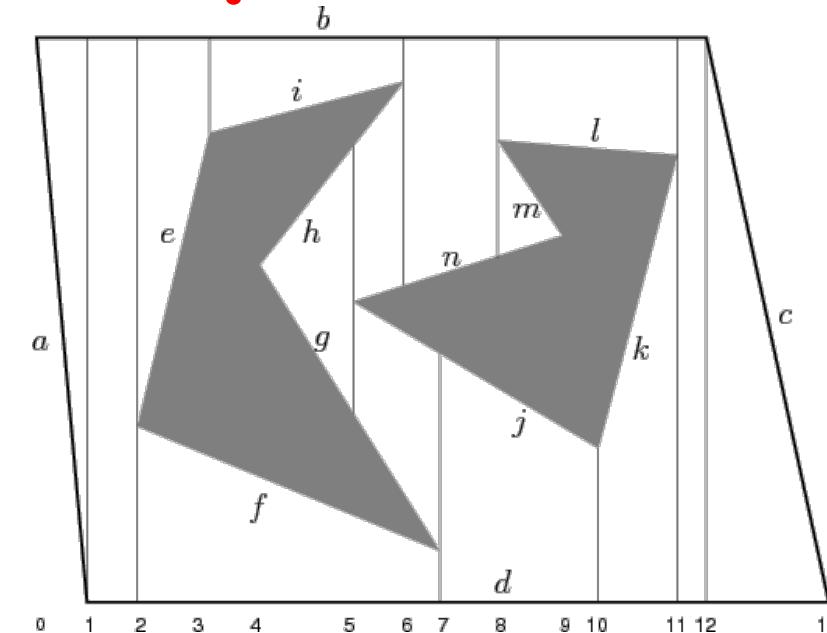




Exact Cell Decomposition

Line-sweep algorithm:

- The visit to each vertex is called an "event"
- Keep a sorted list L of edges of C_{obs} currently intersected by R
- The sorting of L corresponds to the order (from lower to higher) in which the edges are currently intersected by R



Event	Sorted Edges in L	Event	Sorted Edges in L
0	{a, b}	7	{d, j, n, b}
1	{d, b}	8	{d, j, n, m, l, b}
2	{d, f, e, b}	9	{d, j, l, b}
3	{d, f, i, b}	10	{d, k, l, b}
4	{d, f, g, h, i, b}	11	{d, b}
5	{d, f, g, j, n, h, i, b}	12	{d, c}
6	{d, f, g, j, n, b}	13	{}



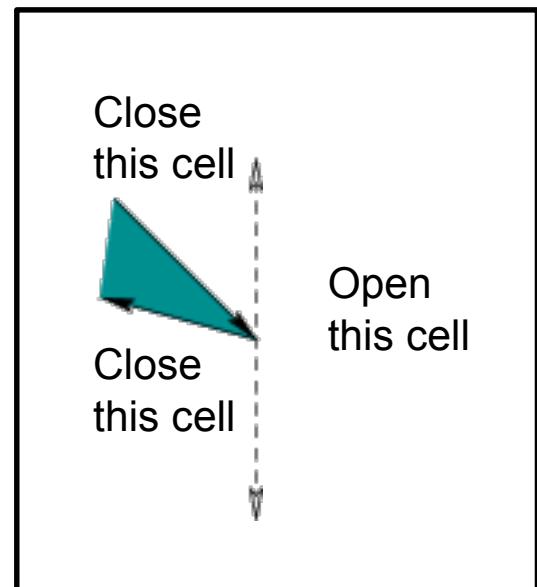
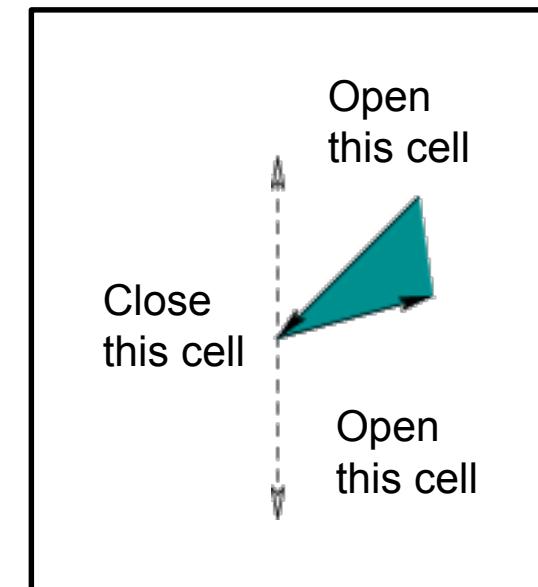
Exact Cell Decomposition

Line-sweep algorithm:

- Events can be classified into 4 different cases:

Case 1 left [right]:

- two rays extending from ν
- two cells are opened [closed]
- two edges are inserted into [deleted from] L



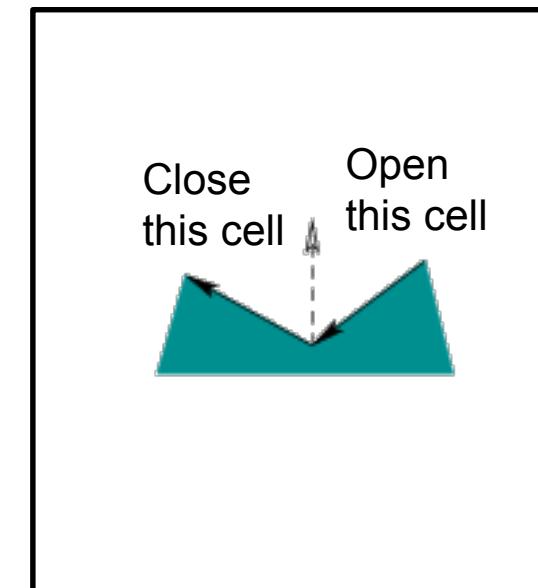


Exact Cell Decomposition

Line-sweep algorithm:

Case 2:

- one ray extending from v
- one cell is opened
- one cell is closed
- one edge is inserted into L and one is deleted



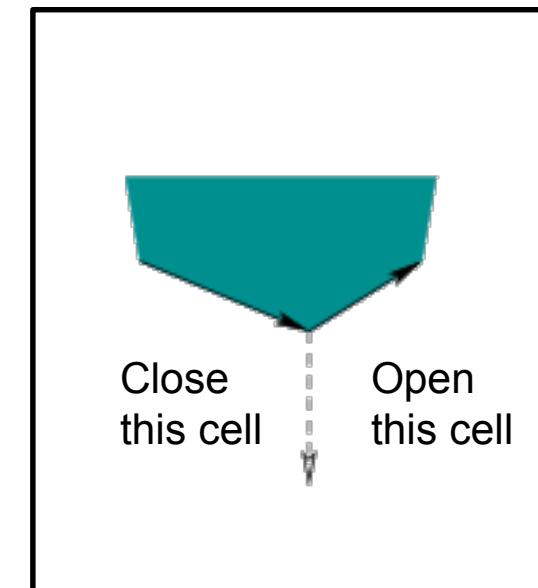


Exact Cell Decomposition

Line-sweep algorithm:

Case 3:

- one ray extending from v
- one cell is opened
- one cell is closed
- one edge is inserted into L and one is deleted



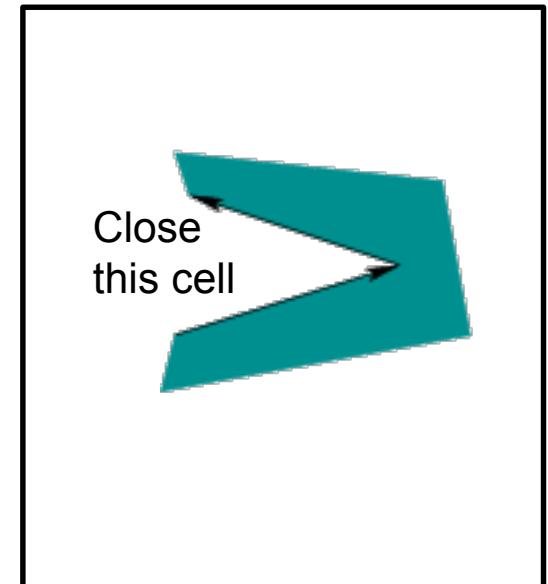
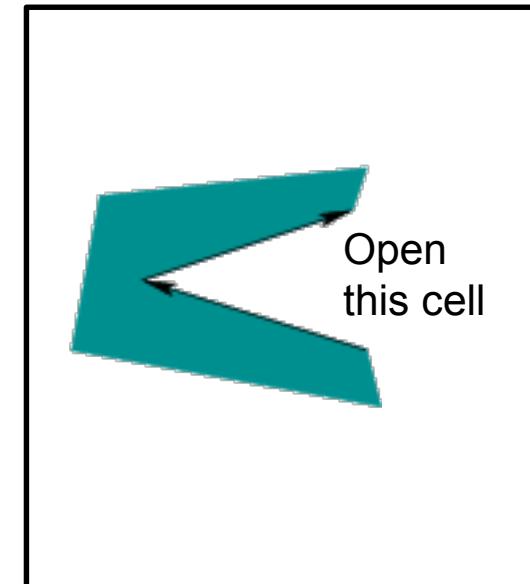


Exact Cell Decomposition

Line-sweep algorithm:

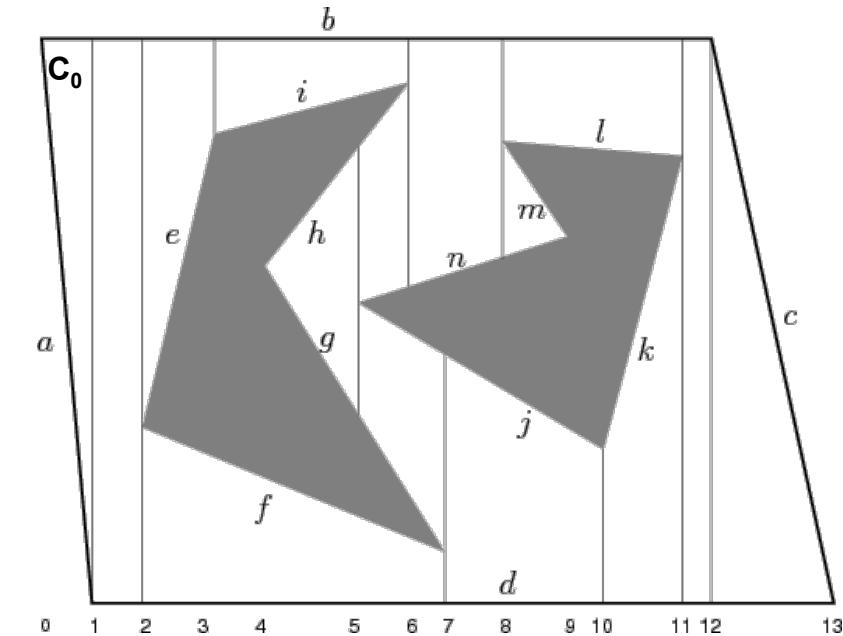
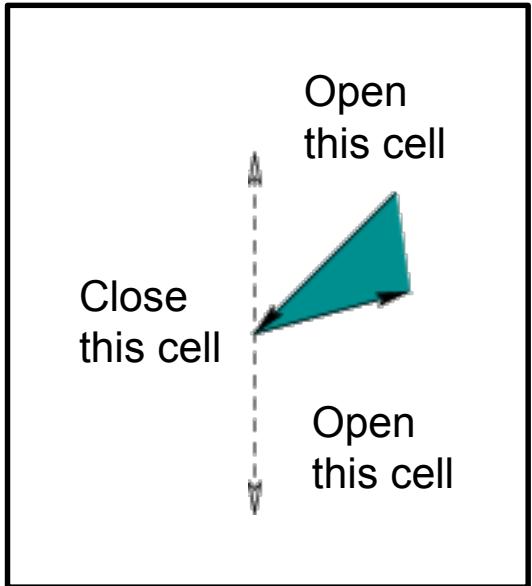
Case 4 left [right]:

- no ray extending from v
- one cell is opened [closed]
- two edges are inserted into [deleted from] L





Example - EVENT 0



Case 1 left [right]:

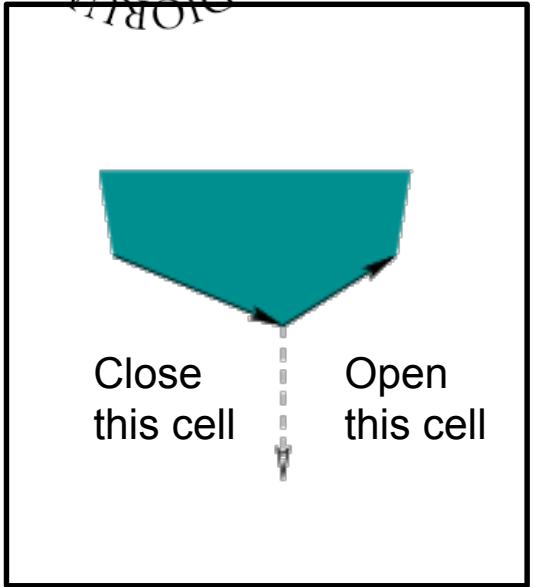
- two rays extending from ν
- two cells are opened [closed]
- two edges are inserted into [deleted from] L

$$L = \{a, b\}$$

Opened Cells: C_0

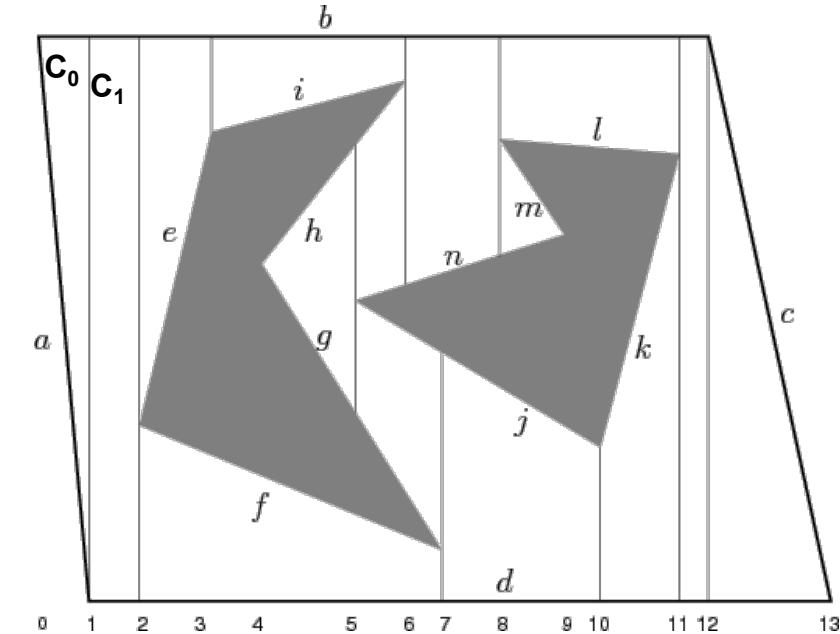


Example - EVENT 1



Case 3:

- one ray extending from ν
- one cell is opened
- one cell is closed
- one edge is inserted into L and one is deleted

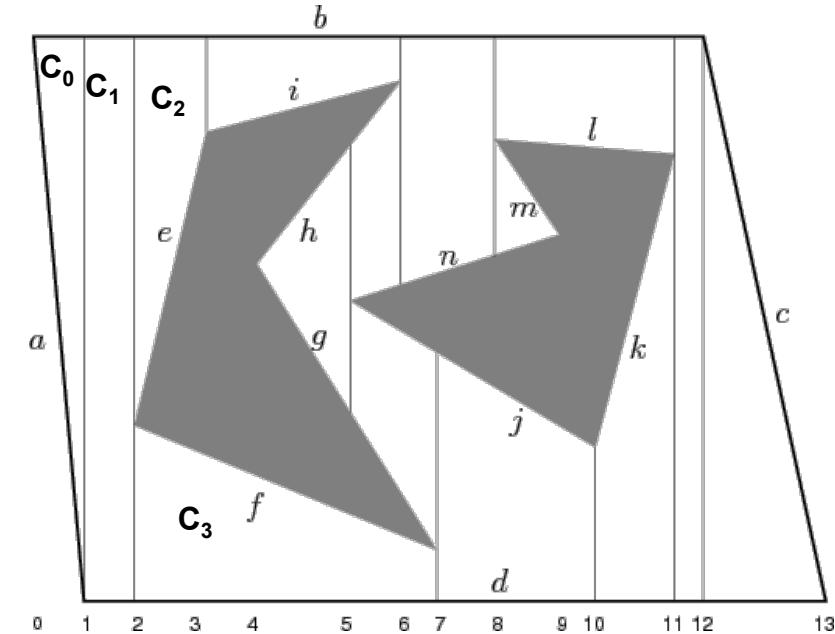
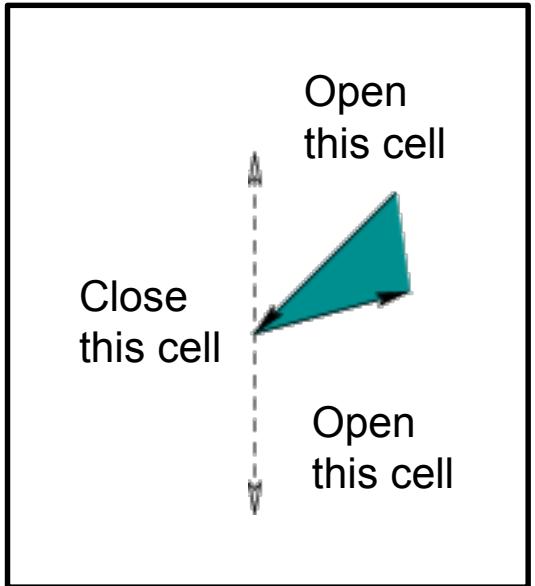


$L = \{a, b\}$
Opened Cells: C_1

$L = \{d, b\}$
Opened Cells: C_1
Closed Cells: C_0



Example - EVENT 2



Case 1 left [right]:

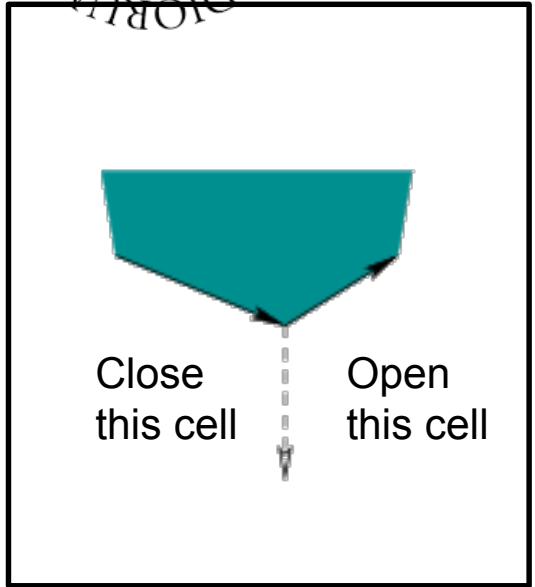
- two rays extending from ν
- two cells are opened [closed]
- two edges are inserted into [deleted from] L

$L = \{d, b\}$
 Opened Cells: C_1
 Closed Cells: C_0

$L = \{d, f, e, b\}$
 Opened Cells: C_2, C_3
 Closed Cells: C_0, C_1

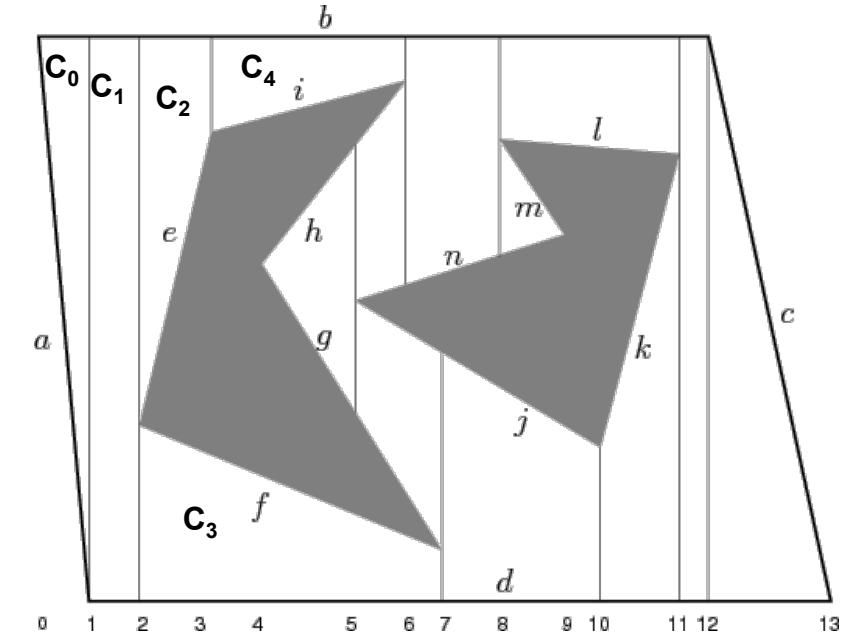


Example - EVENT 3



Case 3:

- one ray extending from ν
- one cell is opened
- one cell is closed
- one edge is inserted into L and one is deleted



$$L = \{d, f, e, b\}$$

Opened Cells: C2, C3

Closed Cells: C0, C1

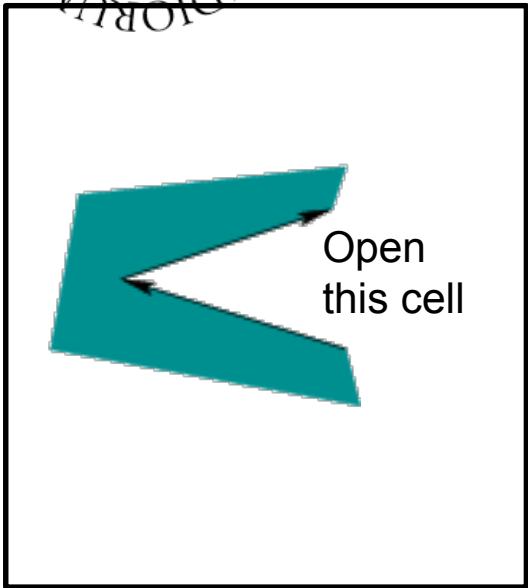
$$L = \{d, f, i, b\}$$

Opened Cells: C3, C4

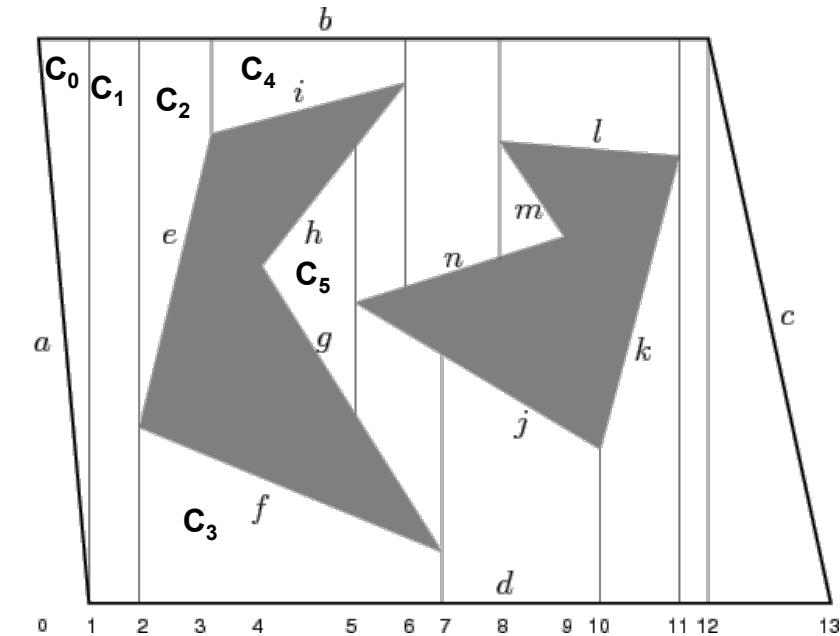
Closed Cells: C0, C1, C2



Example - EVENT 4



Open
this cell



Case 4 left:

- no ray extending from v
- one cell is opened
- two edges are inserted into L

$$L = \{d, f, g, h, i, b\}$$

Opened Cells: C_3, C_4

Closed Cells: C_0, C_1, C_2

$$L = \{d, f, i, b\}$$

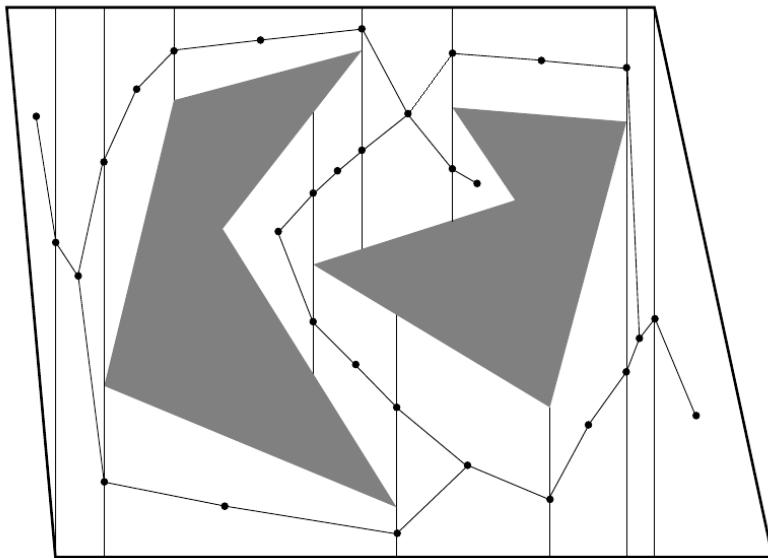
Opened Cells: C_3, C_4, C_5

Closed Cells: C_0, C_1, C_2

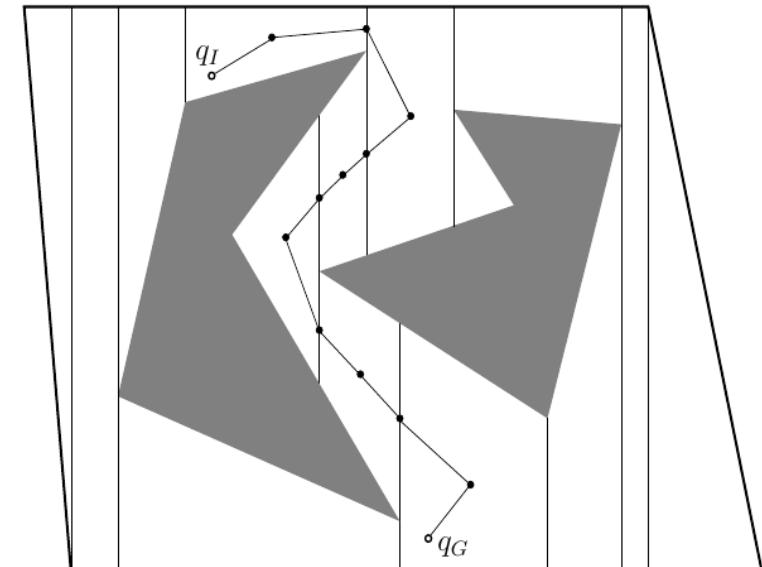


Roadmaps

- Once the cell decomposition is found, the definition of the roadmap is straightforward (exploiting the convexity of the cells).
- We can take the centroid of each cell and midpoint of any face



Roadmap

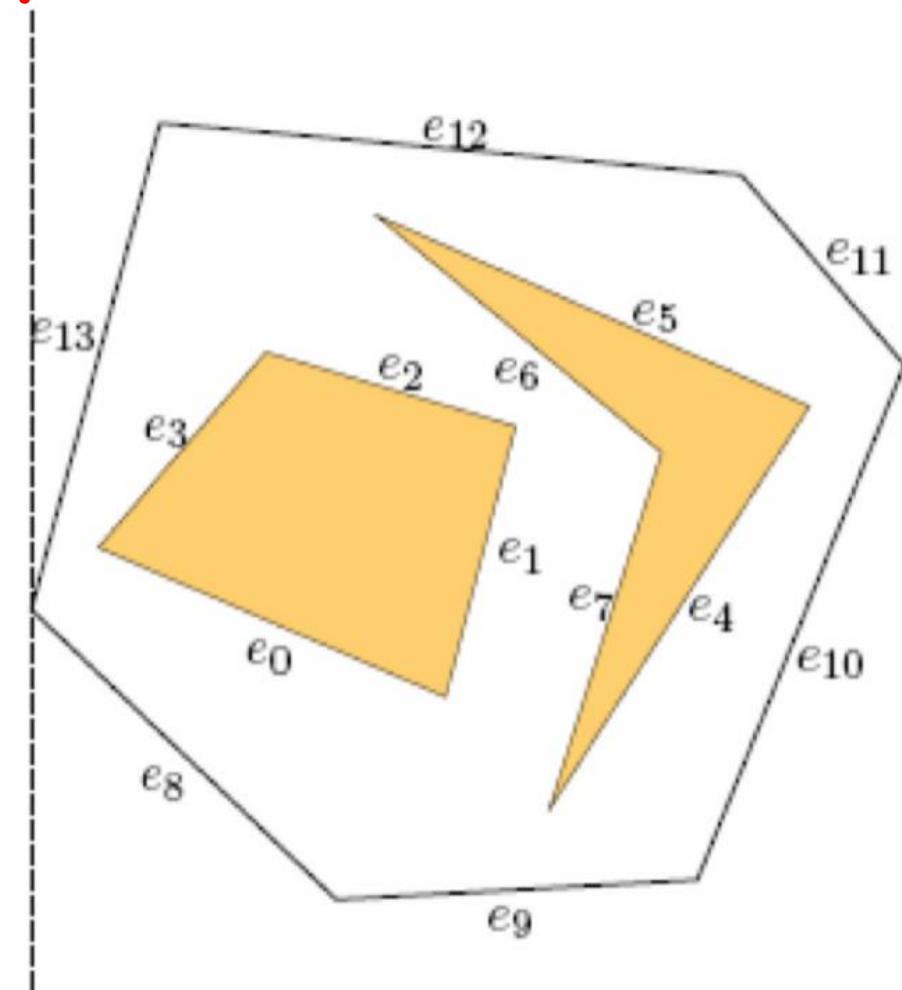


Example Path



Another example

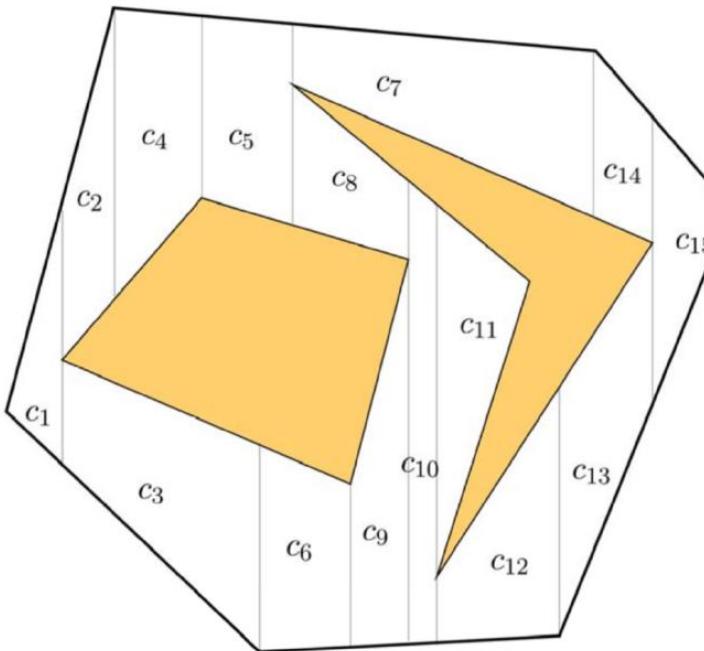
- Try this yourself.....



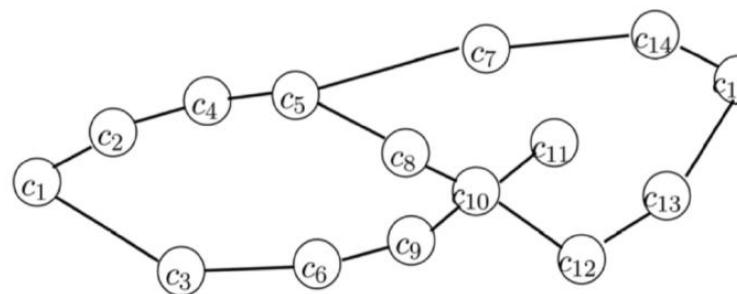


Result

Another example



Cell Partitioning



Connectivity Graph



Exact Cell Decomposition

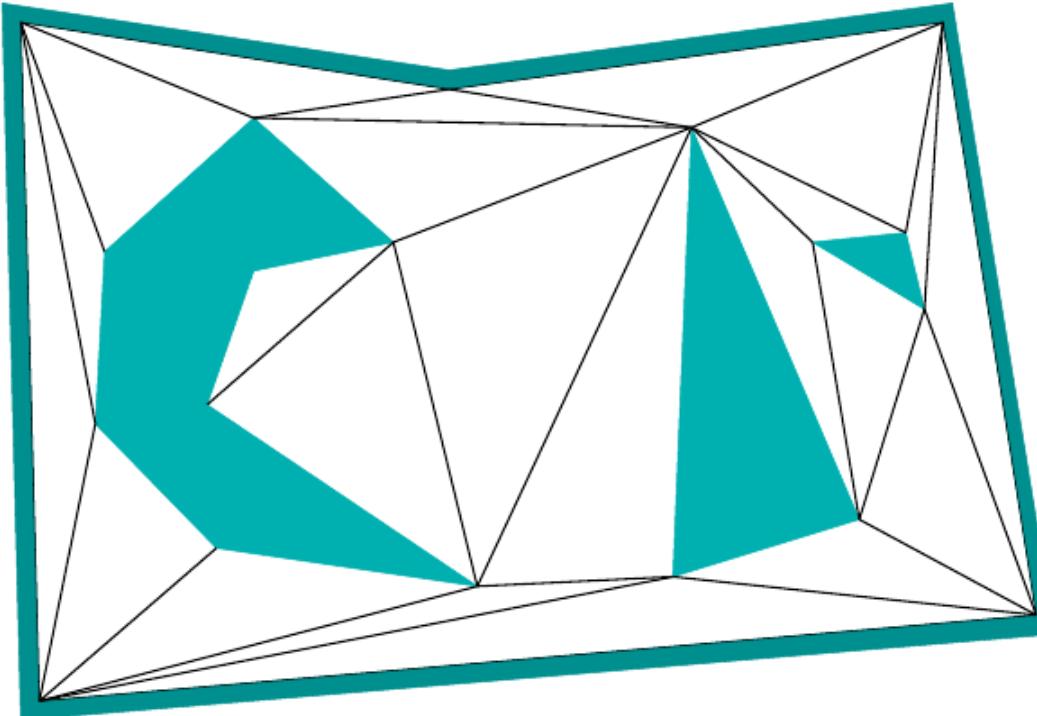
Line-sweep algorithm:

- By using a balanced binary search tree to store L , the insertion/deletion of an edge can be performed in $O(\log n)$ time
- Depending on the case, different updates to the current cells must be performed
 - local update: need to update only the items adjacent to the current vertex
- The roadmap G can be built incrementally at each event (by sampling points along cell boundaries and cell centroids and keeping track of adjacent cells)

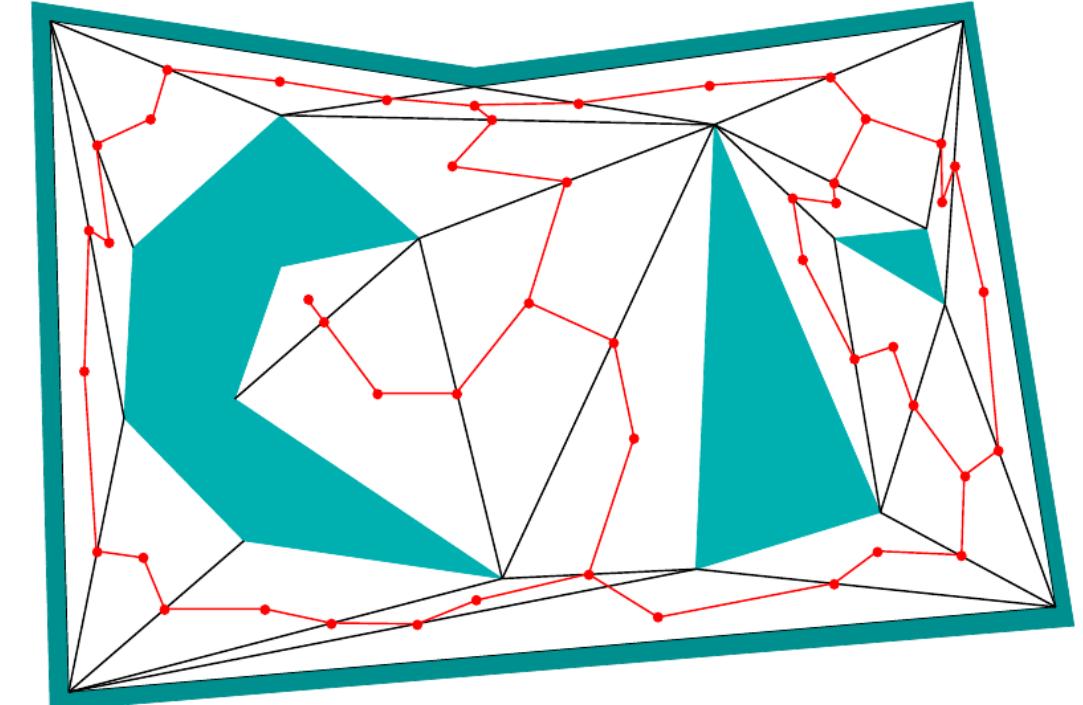


Triangulation

An alternative way to partition into cells is triangulation



Triangulation

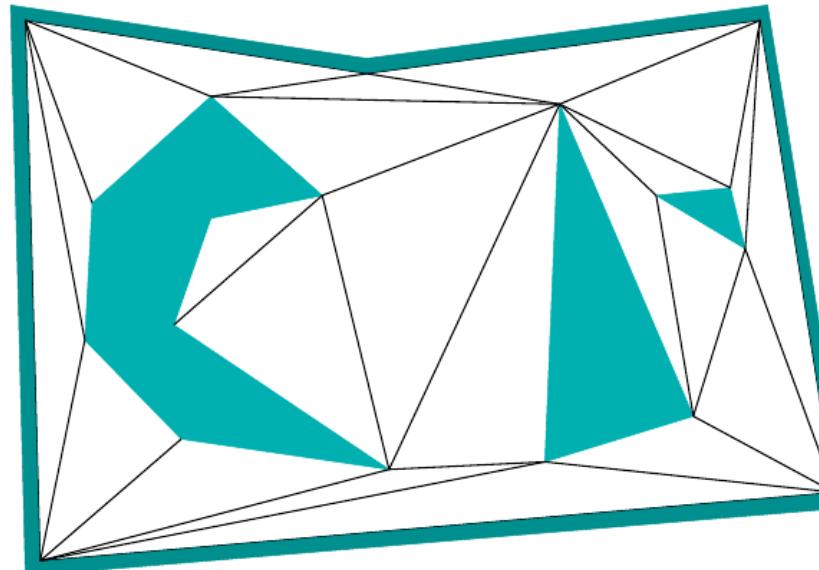


Roadmap



Triangulation

- A simple way to obtain triangulation is to consider the vertices pairwise.
 - If the segment does not intersect any obstacle, then the split is admissible
- This is a $O(n^3)$ solution.
- There are better ways ($O(n^2 \log(n))$) using radial sweep

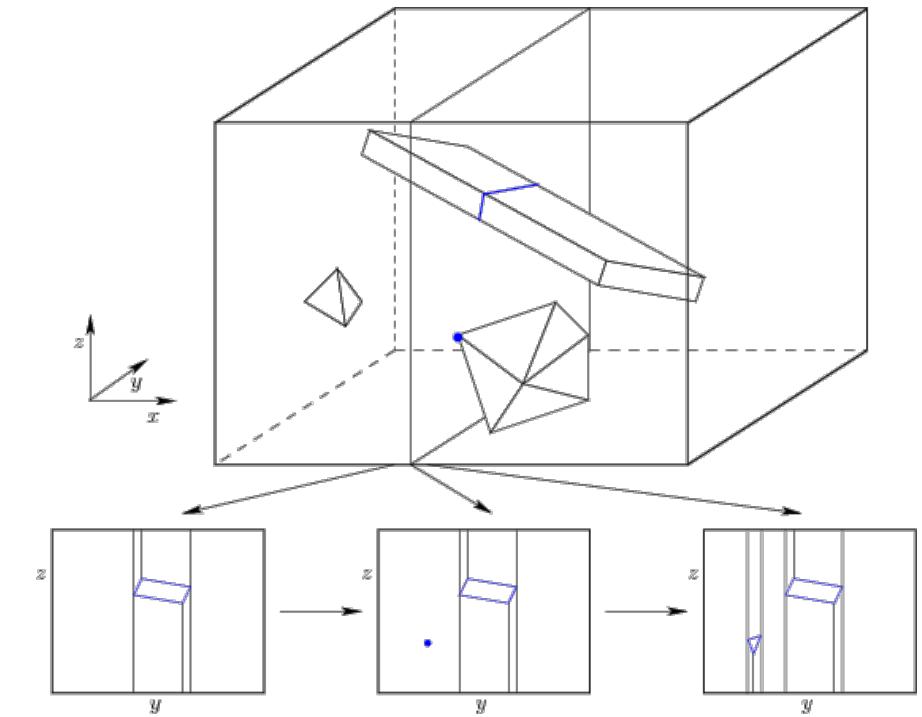




Exact Cell Decomposition

Generalisation and higher dimensions

- The vertical decomposition can be extended to higher dimensions by recursively applying the sweeping idea
- This method, however, works only when C_{obs} is piecewise linear
- E.g. in 3D, a plane is swept along the x axis, and each event produces a 2D polygonal slice of C_{obs}
 - Like in the 2D case, cells can critically change only at some event (i.e. plane sweeping some vertex)

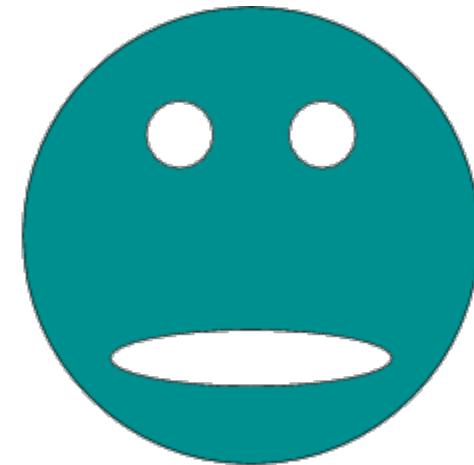




Exact Cell Decomposition

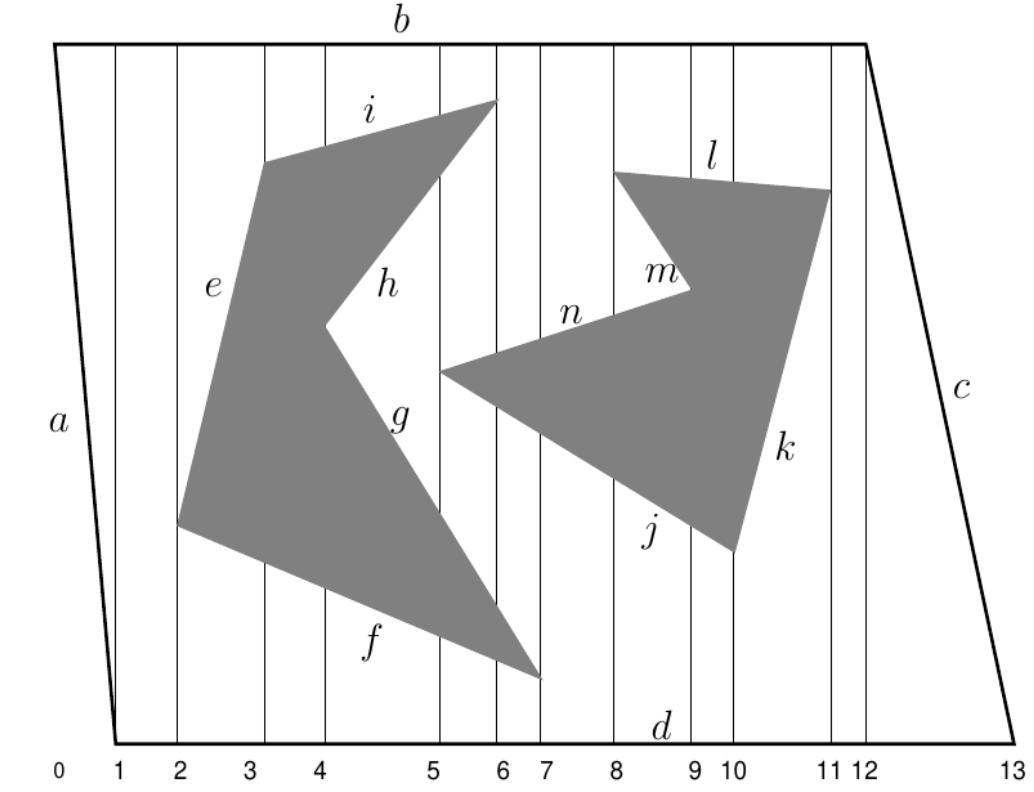
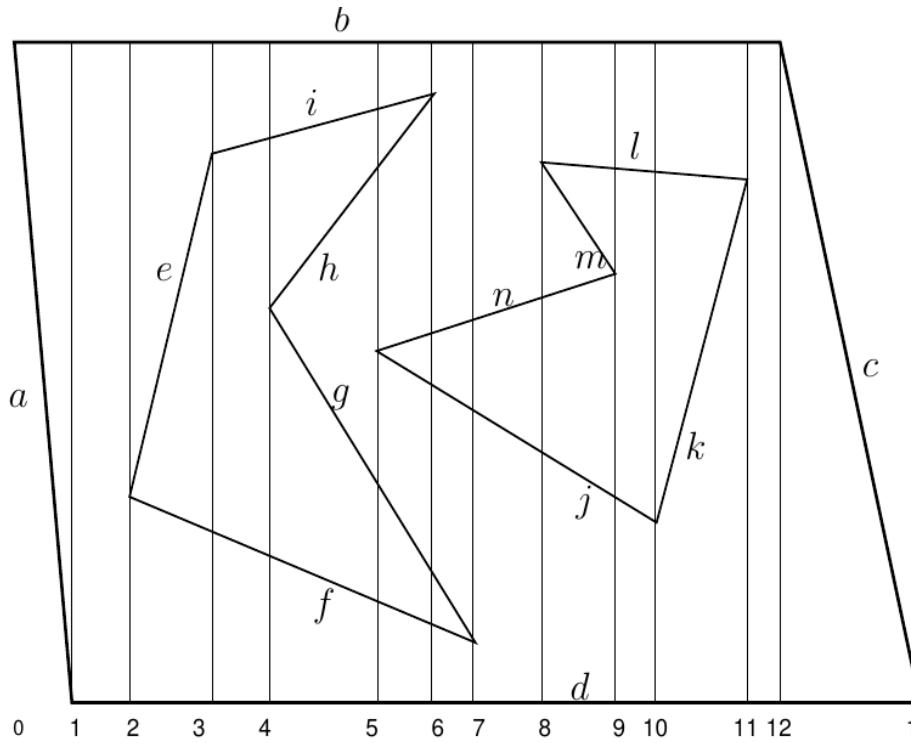
Generalisation and higher dimensions

- It is possible to generalize cell decomposition by using a *cylindrical algebraic decomposition*.
- The only constraint is that both A and O can be described as algebraic sets (using collections of polynomials)



Example

- The cylindrical decomposition is understood going back to our example with vertical decomposition
- Essentially for each event we draw a vertical line pbtaining a cylinder that contains cells interleaved with obstacles.
- For polyhedral obstacles this is inefficient, but it can be generalised to arbitrary shapes and dimensions

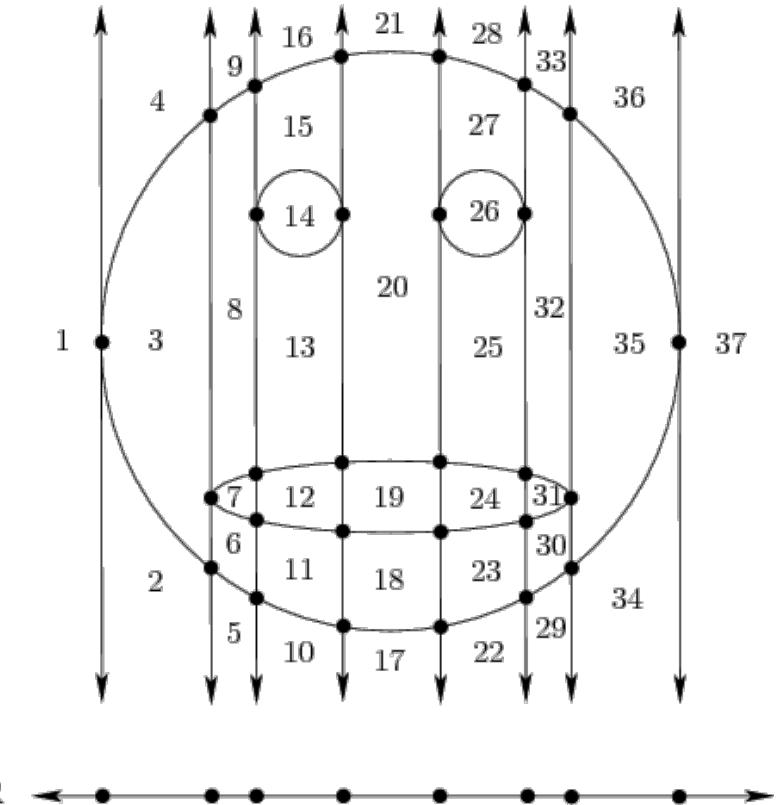




Exact Cell Decomposition

Generalisation and higher dimensions

- Recursive solution:
 - Build m -dimensional cylinders by projecting to $(m - 1)$ -dimensional space, solving (recursively) and then lifting the obtained cells back to a m -dimensional space
- Number of cells doubly exponential in m :
 $O(2^{2^m})$

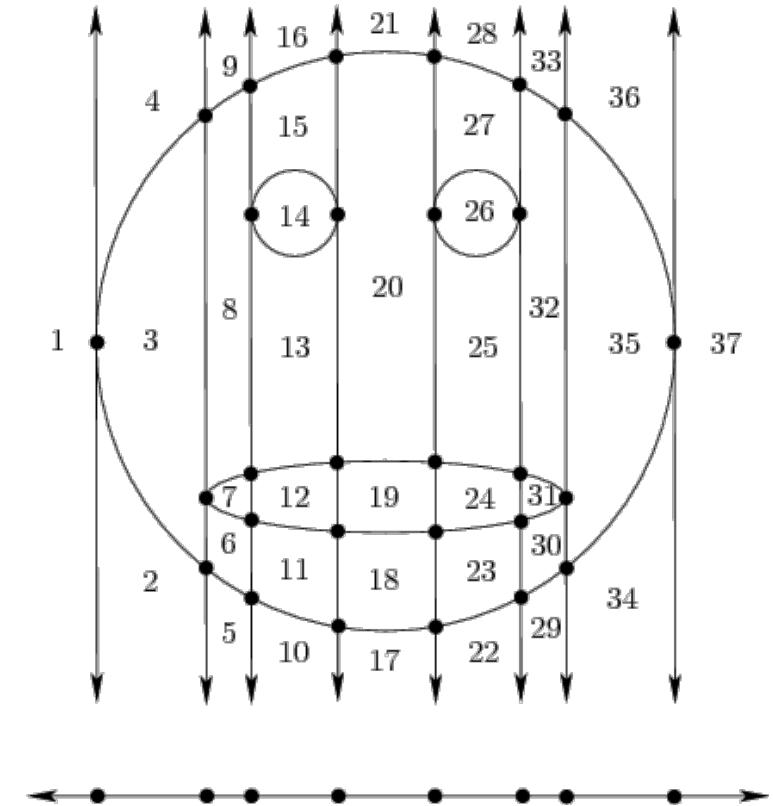




Exact Cell Decomposition

Generalisation and higher dimensions

- Fundamental theoretical result proving the existence of a general (exact, complete) motion planning method
- Not useful in practice for complex scenarios, since it is extremely difficult to implement and has high computational complexity

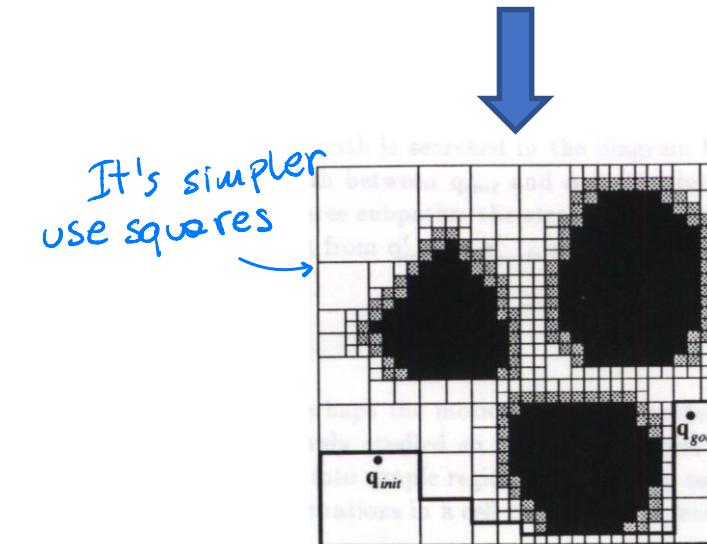
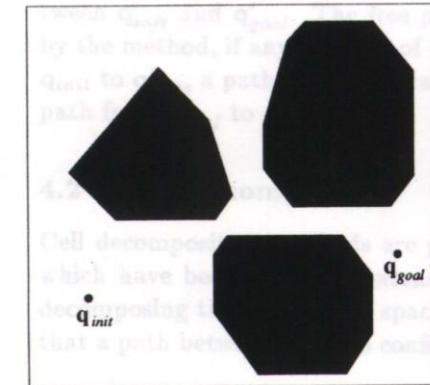




Approximate Cell Decomposition

Give up completeness to increase simplicity

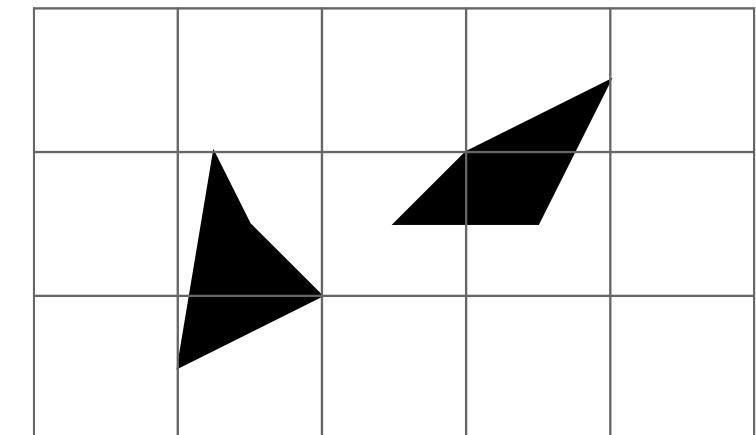
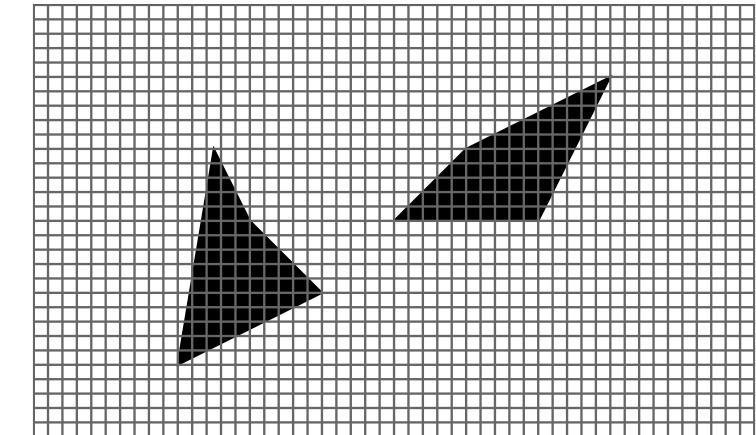
- Represent C_{free} as a collection of non-overlapping cells (like for Exact Cell Decomposition)
- Each cell has a prespecified shape
- These cells cannot represent free space exactly (in general)
- A conservative approximation of C_{free} is constructed





Approximate Cell Decomposition

- The adoption of simple cell shapes determines a simple implementation and lower computational times
- A fixed cell size presents fundamental problems:
 - Large cell size would prevent free paths to be found
 - Small cell size would significantly increase the computational time
- Therefore, usually the cell size is locally adapted to the geometry of C_{obs}





Approximate Cell Decomposition

Definitions:

- A *rectangoloid* designates a closed region of \mathbb{R}^n defined as follows:
$$\{(x_1, \dots, x_n) \mid x_1 \in [x'_1, x''_1], \dots, x_n \in [x'_n, x''_n]\}$$
- The differences $x''_i - x'_i$, for $i = 1, \dots, n$, are called the *dimensions* of the rectangoloid (always greater than zero)



Approximate Cell Decomposition

Definitions:

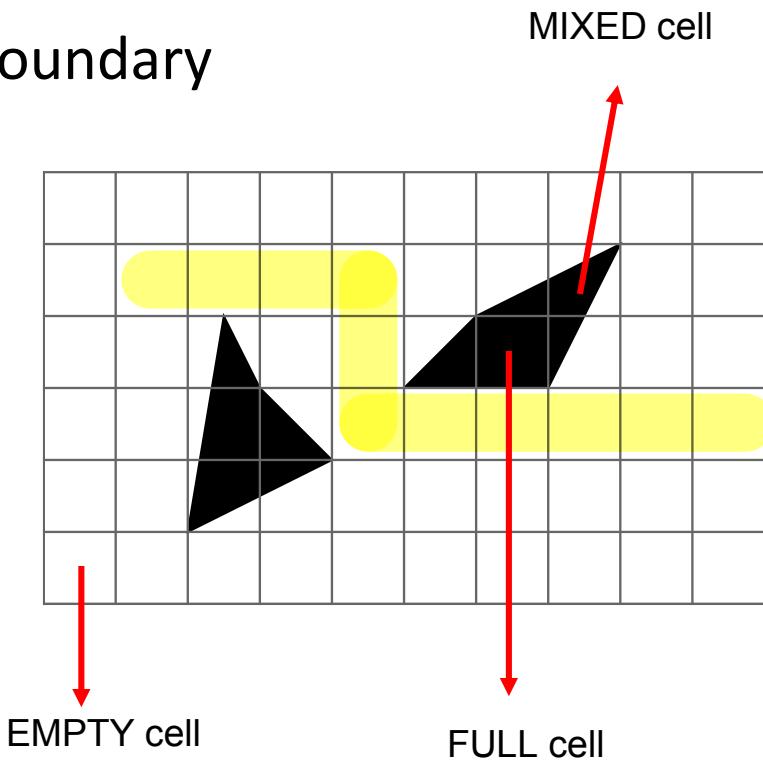
- A *rectangoloid decomposition* P of a rectangoloid $\Omega \subset \mathbb{R}^n$ is a finite collection of rectangoloids $\{k_i\}_{i=1,\dots,r}$ such that:
 - $\Omega = \bigcup_{i=1}^r k_i$, i.e. Ω is equal to the union of k_i , i.e.
 - The interiors of the k_i 's do not intersect



Approximate Cell Decomposition

Definitions:

- Two cells are adjacent if they share a common boundary
- A cell k_i is classified as:
 - *EMPTY*, if it does not intersect C_{obs}
 - *FULL*, if it is entirely contained in C_{obs}
 - *MIXED*, otherwise





Approximate Cell Decomposition

Definitions:

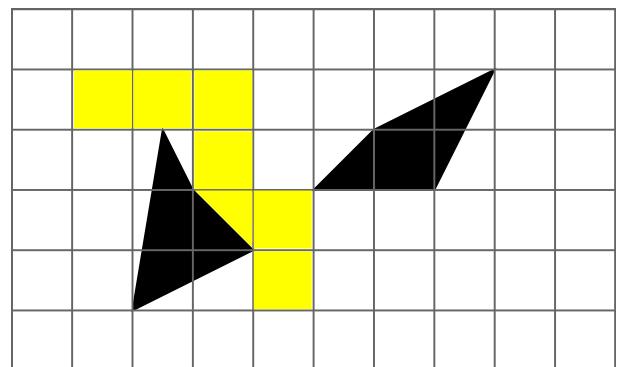
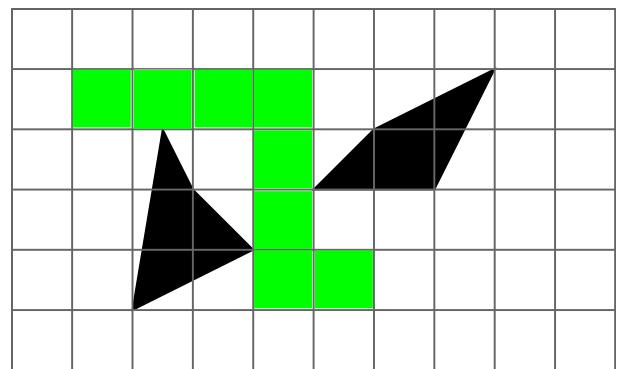
- The connectivity graph associated with a decomposition P of Ω is the undirected graph G defined as follows:
 - The nodes of G are the EMPTY and MIXED cells of P
 - Two nodes of G are connected by a link if and only if the corresponding cells are adjacent



Approximate Cell Decomposition

Definitions:

- Given a rectangoloid decomposition of P of Ω , a channel is defined as a sequence of EMPTY and MIXED cells such that any two consecutive cells are adjacent
- A channel containing only EMPTY cells is called an E-channel
- A channel that contains at least one MIXED cell is called an M-channel

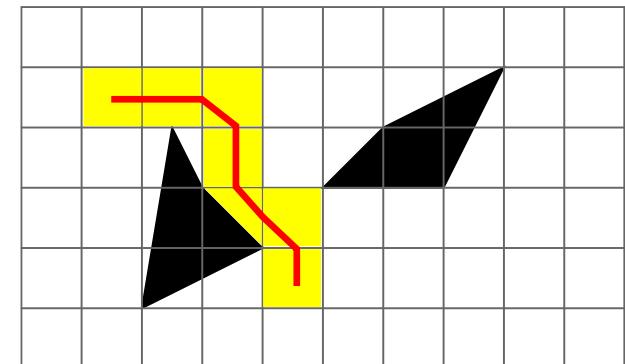




Approximate Cell Decomposition

Definitions:

- For every E-channel, any path lying in its interior a free path
- If K is an M-channel, there may exist a free path lying in its interior connecting two configurations in k_{init} and k_{goal} , but there is no guarantee



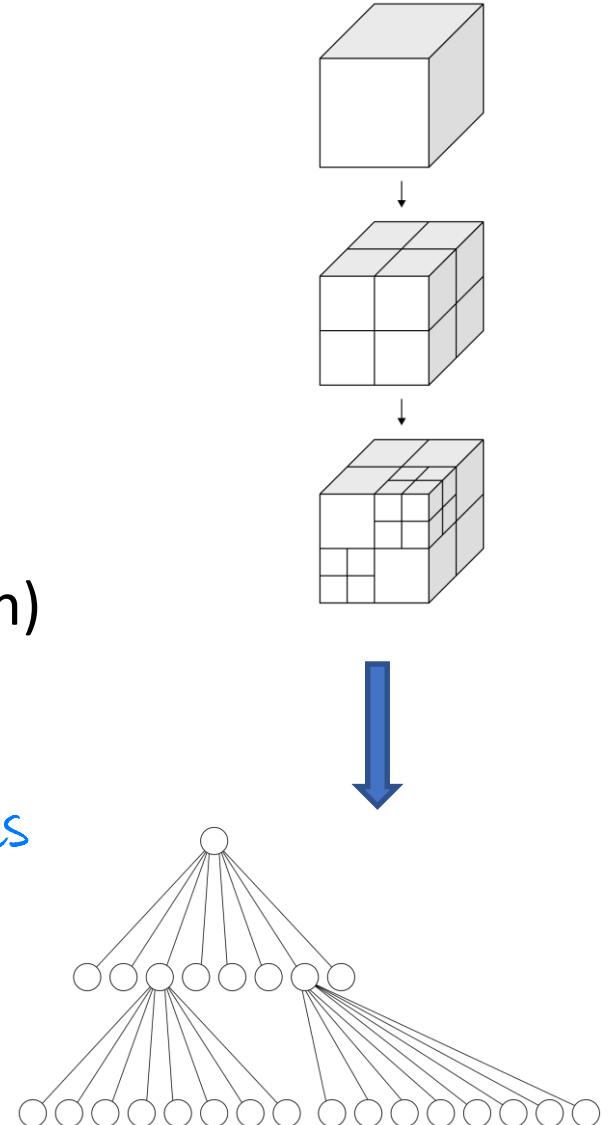


Approximate Cell Decomposition

Hierarchical Decomposition:

- 2^m -tree decomposition (m is the dimension of the configuration space)
- Tree of degree 2^m (i.e. each non-leaf node has exactly 2^m children)
- Each node is a rectangoloid cell labeled EMPTY, FULL (or MIXED)
- The root of the tree is Ω
- Only mixed cells may have children

In order to obtain EMPTY cells





Approximate Cell Decomposition

Hierarchical Decomposition:

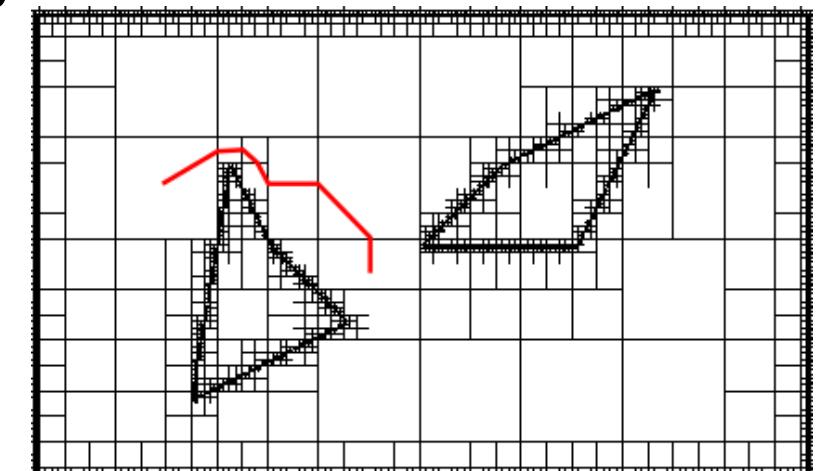
- All the children of a cell k have the same dimensions and are obtained by cutting each edge of k into two segments of equal length
- The depth of a node (number of arcs from the root) determines the dimensions of the corresponding cell relative to Ω
- The height h of the tree (depth of deepest node) determines the resolution of the decomposition of Ω (size of smallest cells in the decomposition)



Approximate Cell Decomposition

Path Planning:

- Given an initial configuration $q_{init} \in C_{free}$ and a goal configuration $q_{goal} \in C_{free}$, the problem is to generate an E-channel from k_{init} to k_{goal}
- Hierarchical approach:
 - Apply an initial, coarse decomposition
 - Locally refine it until a free path is found or the decomposition becomes too fine





Approximate Cell Decomposition

Path Planning:

- Generate an E-channel by iteratively decomposing Ω and searching the associated connectivity graphs
- Let P_i denote the successive decompositions of Ω
 - Each decomposition P_i is obtained from the previous, P_{i-1} , by decomposing one or several MIXED cells
 - Whenever a decomposition P_i is computed, the associated connectivity graph G_i is searched for a channel connecting q_{init} to q_{goal}



Approximate Cell Decomposition

Path Planning:

1. Compute a rectangoloid decomposition P_1 of Ω
2. Search the corresponding connectivity graph for a channel
3. If the output is an E-channel, the search is completed
4. If it is an M-channel, proceed to the next step, otherwise return failure
5. Let Π_i be the M-channel
6. Set P_{i+1} to P_i
7. For every MIXED cell k in Π_i compute a rectangoloid decomposition P^k of k and set P_{i+1} to $[P_{i+1} \setminus \{k\}] \cup P^k$



Approximate Cell Decomposition

- Approximated cell decomposition is a general method and could be applied to the basic motion planning problem in its generality
- In practice, the time and space grows quickly with the dimension m of the configuration space
- Realistically applicable only when the dimension is small (e.g. $m \leq 4$)



Maximum Clearance Roadmap

- Tries to stay as far as possible from C_{obs}
- Good solution for various robotics applications:
 - Sometimes it is difficult to measure and control the precise position of a mobile robot
 - Traveling along the maximum-clearance roadmap reduces the risk of collision due to these uncertainties
- Known also as Generalized Voronoi Diagram or Retraction Method

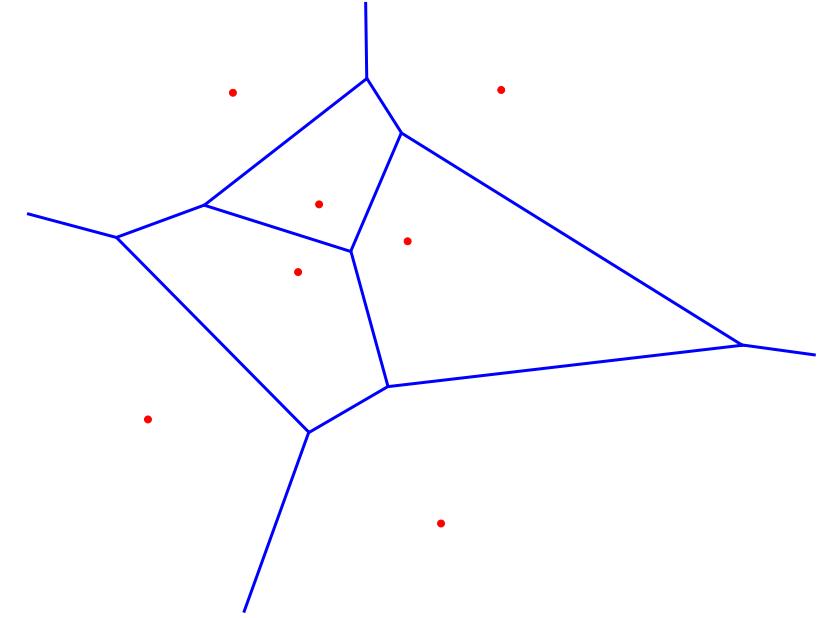


Maximum Clearance Roadmap

- Regular Voronoi diagrams are defined for point obstacles only. The resulting diagram is a collection of line segments (in 2D)
- Set of points where the clearance to the closest obstacles is the same



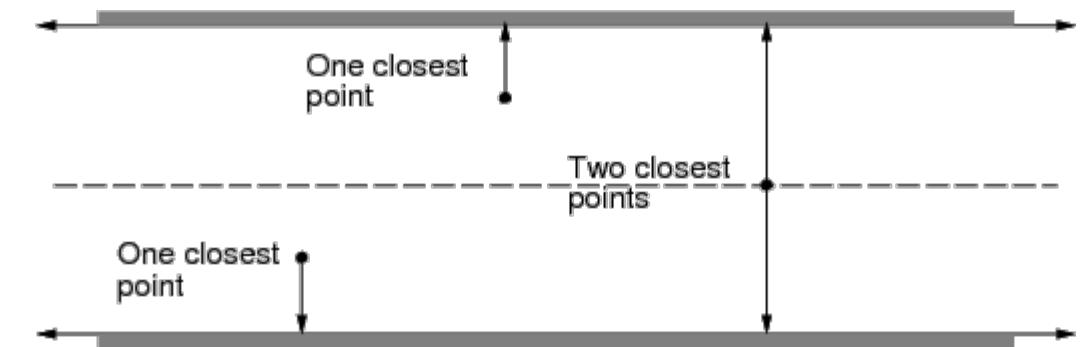
Generalisation of Voronoi Diagram from single, isolated points to polygons





Maximum Clearance Roadmap

- Each point along a roadmap edge is equidistant from two points on the boundary of C_{obs}
- Each roadmap vertex corresponds to the intersection of two or more roadmap edges and is therefore equidistant from three or more points along the boundary of C_{obs}





Maximum Clearance Roadmap

Formal Definition:

- $\beta = \partial C_{free}$: boundary of C_{free}
- $d(p, q) = \|q - p\|$: euclidean distance between p and q
- For all q in C_{free} , let

$$\text{clearance}(q) = \min_{p \in \beta} d(p, q)$$

be the clearance of q , and

$$\text{near}(q) = \{p \in \beta \mid d(p, q) = \text{clearance}(q)\}$$

be the set of base/foot points on β with the same clearance to q



Maximum Clearance Roadmap

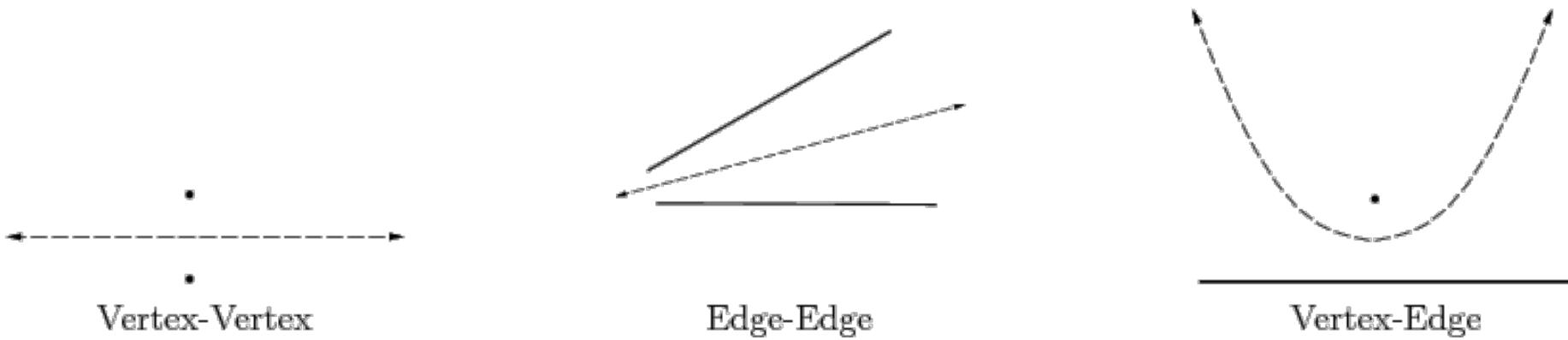
Formal Definition:

- The Generalized Voronoi Diagram is then the set of q 's with more than one base point p , i.e:

$$V(C_{free}) = \{q \in C_{free} \mid |\text{near}(q)| > 1\}$$



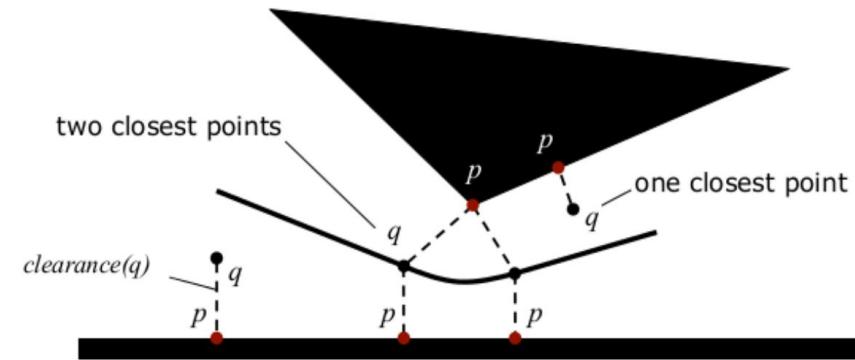
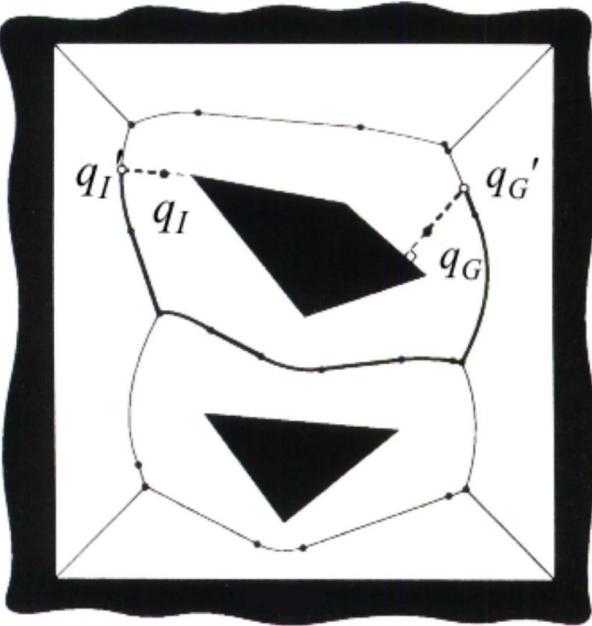
Maximum Clearance Roadmap



For polygonal C_{obs} , the diagram consists of lines and parabolic edges



Maximum Clearance Roadmap



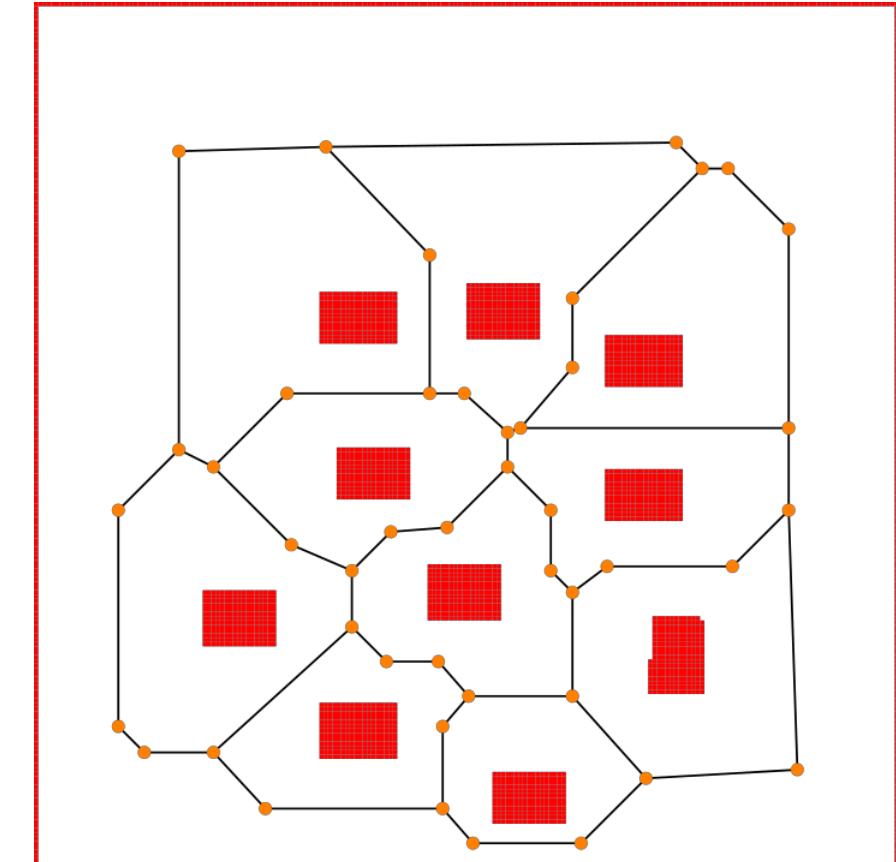
For polygonal C_{obs} , the diagram consists of lines and parabolic edges



Maximum Clearance Roadmap

- With n being the number of vertices on β , the best algorithm has $O(n \log n)$ complexity (based on a sweep-line method)
- For robot motion planning GVD can be used to find clear routes which are furthest from obstacles
- Implemented in several open-source libraries, e.g. Boost:

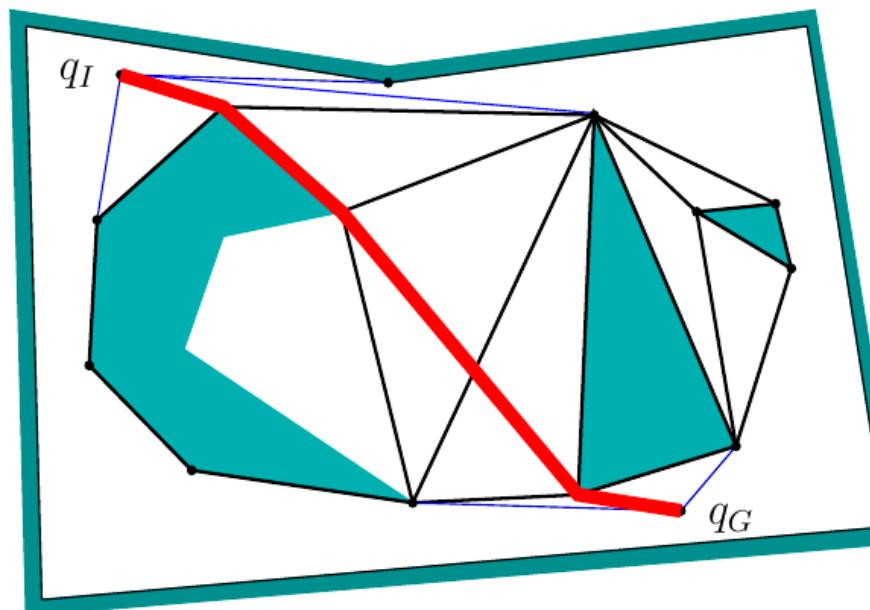
https://www.boost.org/doc/libs/1_67_0/libs/polygon/doc/voronoi_main.htm





Shortest Path Design

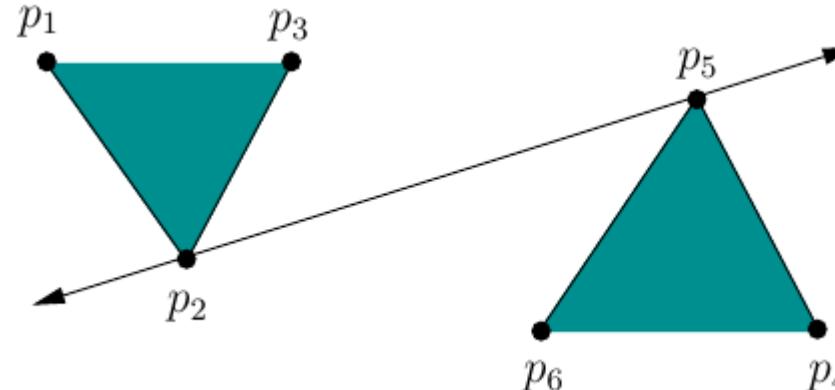
- Maximum clearance is designed to maximise safety
- We could as well seek the solution that “grazes” the obstacles





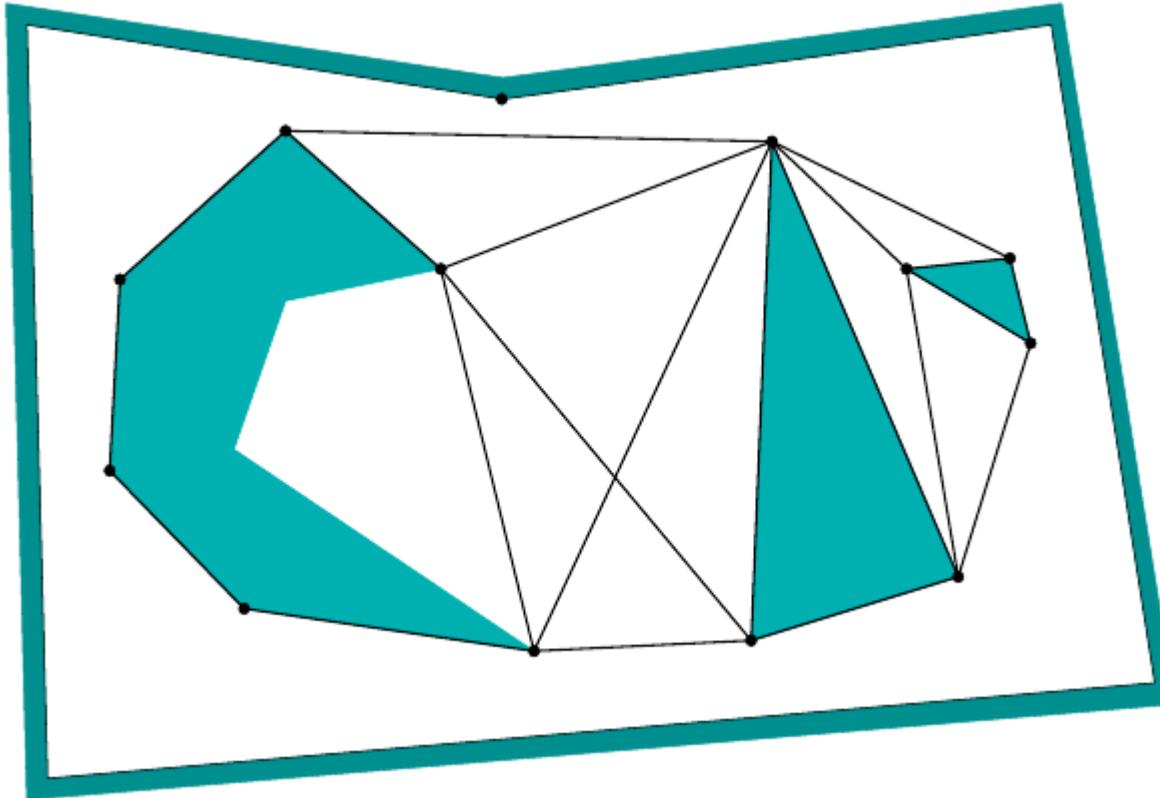
Shortest Path Construction

- Let a *reflex vertex* be a vertex for which the area in C_{free} is greater than π
 - All vertices of a convex polygon are reflex vertices
- The Graph G is made of reflex vertices
- Consecutive Reflex Vertices
 - If two vertices are the end-point of an edge of C_{obs} then an edge between them is made in the graph
- If a bitangent line can be found between two reflex vertices then it corresponds to an edge in v
 - a bitangent line does not poke into the interiors of C_{obs} and the vertices must be mutually visible





Example

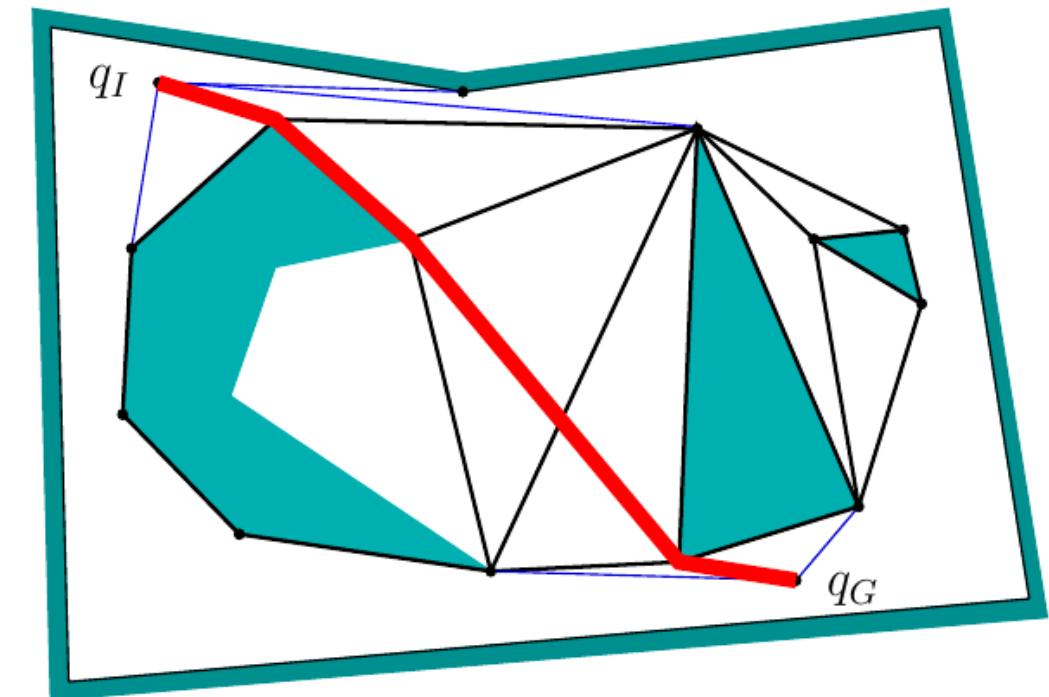
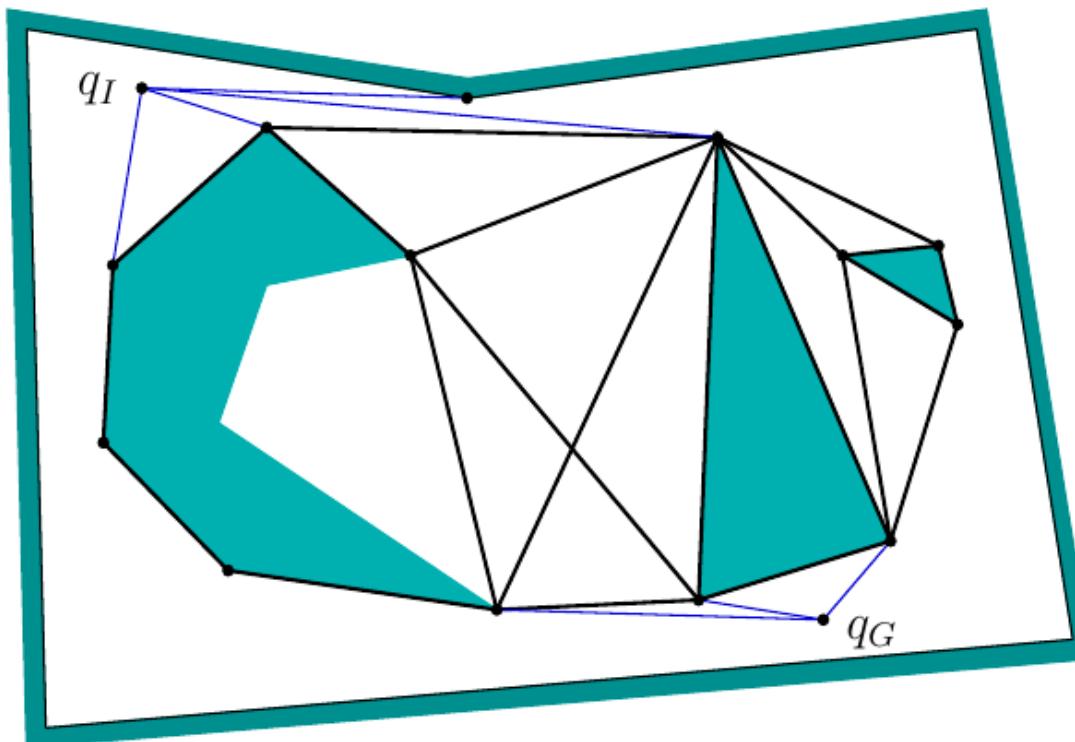


Shortest Path Roadmap



Queries

- Queries on shortest roadmaps are made on graphs





Question 1

- What are the main inputs to a motion planning problem?



Answer 1

- Initial configuration, goal configuration, robot model, and map of the environment (obstacles).



Question 2

- What distinguishes an optimal motion plan from a feasible one?



Answer 2

- An optimal plan minimizes a cost (e.g., length, smoothness, or time), while a feasible plan only ensures collision-free motion.



Question 3

- What is a roadmap in motion planning?



Answer 3

- A roadmap is a graph representing the connectivity of the free configuration space (C_{free}).



Question 4

- Explain the accessibility and connectivity-preserving properties of a roadmap.



Answer 4

- Accessibility: any point in C_{free} can connect to the roadmap.
Connectivity-preserving: if a free path exists, a roadmap path also exists.



Question 5

- What is the goal of exact cell decomposition?



Answer 5

- To partition C_{free} into simple, non-overlapping cells whose union exactly equals C_{free} .



Question 6

- Why is convexity important in exact cell decomposition?



Answer 6

- Because any two points in a convex cell can be connected by a straight line that lies entirely inside the cell.



Question 7

- What is the basic idea behind trapezoidal decomposition?



Answer 7

- Shoot vertical rays from each vertex until hitting an obstacle or boundary, forming trapezoidal cells.



Question 8

- What is the main advantage of using a line-sweep algorithm for trapezoidal decomposition?



Answer 8

- It reduces complexity from $O(n^2)$ to $O(n \log n)$ by maintaining a sorted edge list during the sweep.



Question 9

- How can vertical decomposition be extended to 3D?



Answer 9

- By sweeping a plane through the 3D space and updating 2D slices as events occur.



Question 10

- What is the significance of cylindrical algebraic decomposition?



Answer 10

- It provides a complete theoretical method for motion planning in any dimension, but is computationally infeasible (doubly exponential in dimension).



Question 11

- How does approximate cell decomposition differ from exact?



Answer 11

- It uses a predefined cell shape (e.g., rectangular) that only approximates C_{free} .



Question 12

- What is the role of hierarchical decomposition in approximate cell decomposition?



Answer 12

- It adaptively refines MIXED cells into smaller ones, using a 2^m -tree, until a path is found or resolution is too fine.



Question 13

- What is the purpose of a maximum-clearance roadmap?



Answer 13

- To keep the robot as far as possible from obstacles, reducing collision risk due to uncertainty.



Question 14

- How does the Generalized Voronoi Diagram define roadmap edges?



Answer 14

- Each point on an edge is equidistant from two or more obstacle boundaries; vertices are equidistant from three or more.



Question 15

- What is a reflex vertex?



Answer 15

- A vertex where the interior angle in C_free is greater than 180° , typically at obstacle corners.



Question 16

- What conditions make a bitangent line valid in a shortest-path roadmap?



Answer 16

- It must connect two reflex vertices that are mutually visible and not intersect any obstacle interior.

CHOMP: Covariant Hamiltonian Optimisation for Motion Planning

Luigi Palopoli - taken from Zucker et al.

University of Trento

October 20, 2025



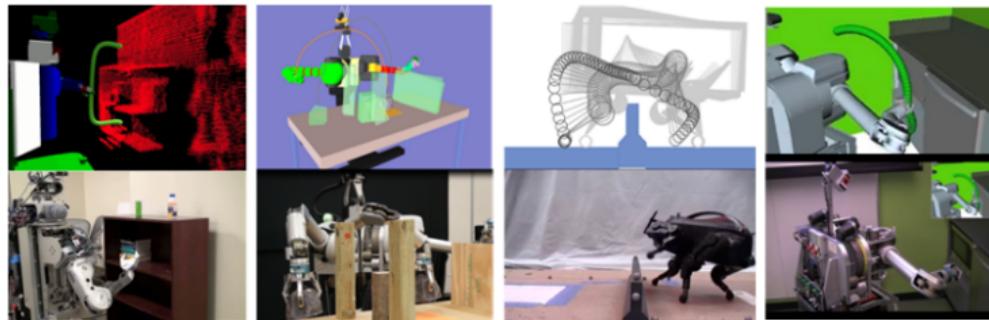
Covariant Hamiltonian Optimisation for Motion Planning (CHOMP)

Introduction: The Challenge of High-Dimensional Planning

- ▶ The increasing complexity of robots, such as 7 DOF arms or 34 DOF humanoids like Atlas, demands **high-dimensional motion planning**.
- ▶ Efficient motion planning is crucial for these robots to perform tasks while respecting motion constraints and **avoiding collisions**.
- ▶ **Trajectory optimisation** is fundamental in optimal control, solving for a sequence of states and controls that optimises a given objective subject to constraints.
- ▶ Traditionally, trajectory optimisation faced challenges in motion planning due to obstacles requiring the solution of a **non-convex, constrained optimisation problem**.

CHOMP: a first recent success in optimisation-based planning

- ▶ CHOMP was developed at CMU and sparked a new interest in optimisation based techniques by demonstrating effectiveness on various robotic platforms.





CHOMP: The Foundational Approach

CHOMP (Covariant Hamiltonian Optimization for Motion Planning) is a method for trajectory optimisation that is **invariant to reparametrisation**.

CHOMP builds on two core ideas:

- 1. Gradient Information is Inexpensive:** Functional **gradient techniques** are used to iteratively improve the trajectory quality. This information allows obstacles to 'push' the robot away from collisions, finding solutions even from **infeasible initial trajectories**.
- 2. Invariance to Parametrisation:** The trajectory is treated as a geometric object (a point in a Hilbert space), **ensuring identical behaviour regardless of the representation used** (e.g., spline vs. waypoints).

The CHOMP Objective Functional

The objective functional $U[\xi]$ measures two complementary aspects of the motion planning problem:

$$U[\xi] = F_{obs}[\xi] + \lambda F_{smooth}[\xi]$$

- ▶ ξ : The trajectory, mapping time $t \in [0, \dots, 1]$ to configurations $q \in C \subset \mathbb{R}^d$.
- ▶ F_{smooth} : Encourages smooth trajectories (dynamical criteria). *→ limiting $\ddot{\alpha}$*
- ▶ F_{obs} : Encourages collision-free trajectories by penalising proximity to obstacles.
- ▶ λ : A trade-off parameter (often managed via **weight scheduling**).

The Smoothness Term F_{smooth}

F_{smooth} penalises trajectories based on dynamical criteria, such as velocity or acceleration, to encourage smooth motion.

Smoothness Objective (Squared Velocity Example)

$$F_{smooth}[\xi] = \frac{1}{2} \int_0^1 \left\| \frac{d}{dt} \xi(t) \right\|^2 dt$$

you can also have penalty on the curve for the example

- ▶ This formulation generalises prior notions of trajectory smoothness (e.g., springs and dampers models).
- ▶ By defining F_{smooth} in terms of a metric in the space of trajectories, it can include higher-order derivatives (e.g., accelerations or jerks).
- ▶ It governs the **timing** along the path.

The Obstacle Term F_{obs}

F_{obs} accumulates cost encountered by the robot's body points as it sweeps across the trajectory.

Time is Fixed

Obstacle Objective (Arc-Length Parametrisation)

$$F_{obs}[\xi] = \int_0^1 \int_B c(x(\xi(t), u)) \left\| \frac{d}{dt} x(\xi(t), u) \right\| du dt$$

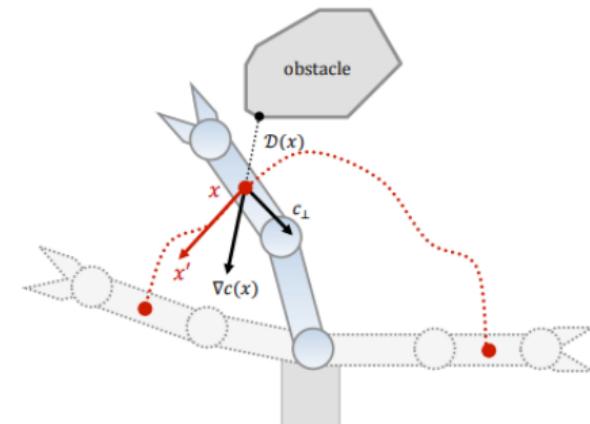
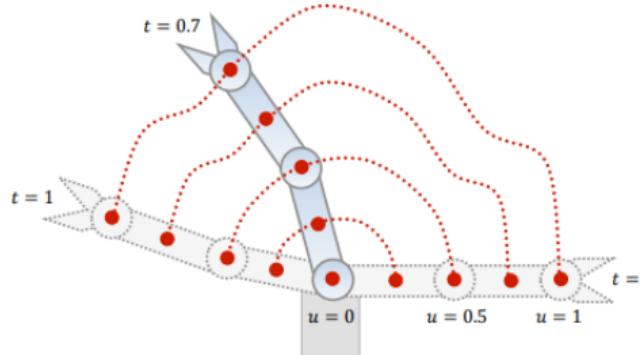
different parts of the body that can collide with the obstacle

- ▶ $x(q, u)$: Forward kinematics mapping configuration q and body point u to workspace position.
- ▶ c : Workspace cost function penalising proximity to obstacles.
- ▶ Crucially, the cost is multiplied by the **norm of the workspace velocity** $\left\| \frac{d}{dt} x(\xi(t), u) \right\|$, ensuring the objective is **invariant to re-timing**.
- ▶ F_{obs} governs the **shape** of the path.

se mi muovo lentamente, il tempo a contatto con l'ostacolo e' maggiore rispetto a quello che sarebbe se mi muoressi velocemente

The Obstacle Term F_{obs}

Meaning of the different quantities.



$D(x)$ is the distance from the object. A single point $x(\xi(t), u)$ is identified by the position on the trajectory and inside the body (u).

Obstacle Cost Function $c(x)$ and Distance Fields

The workspace cost $c(x)$ is typically defined in terms of the **Euclidean Signed Distance** $D(x)$ to the boundaries of obstacles.

Signed Distance Field (SDF)

- ▶ $D(x)$ is **negative** if x is inside an obstacle, **positive** if outside, and **zero** on the boundary.
- ▶ SDFs are often computed using Euclidean Distance Transform (EDT) algorithms.
- ▶ The robot model is simplified using geometric primitives (spheres, capsules).

The general **obstacle cost function** $c(x)$ uses a **hinge-like penalty** (shown below), dropping smoothly to zero when distance $D(x)$ exceeds an allowable threshold ε .

$$c(x) = \begin{cases} -D(x) + \frac{1}{2\varepsilon}, & \text{if } D(x) < 0 \\ \frac{1}{2\varepsilon}(D(x) - \varepsilon)^2, & \text{if } 0 < D(x) \leq \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

MATLAB Code Example

Listing 1: Distance field - Part 1

```
1 x1 = [50 150 150 50];
2 y1 = [50 50 150 150];
3 x2 = [200 300 250];
4 y2 = [100 100 200];
5 % Define the size of the output bitmap (rows x columns)
6 rows = 300;
7 cols = 400;
8 % 2. Convert each polygon to a binary mask using '
9 %     poly2mask'
10 mask1 = poly2mask(x1, y1, rows, cols);
11 mask2 = poly2mask(x2, y2, rows, cols);
12 % Combine the masks (logical OR operation)
13 combined_mask = mask1 | mask2;
```

MATLAB Code Example

Listing 2: Distance field - Part 2

```
1 % 3. Display and Save the binary image array
2 figure;
3 imshow(combined_mask);
4 title('Binary Mask of Polygons');
5 % D = bwdist(BW) computes the Euclidean distance
6 % transform (default)
7 D = bwdist(combined_mask);
8 D1 = bwdist(~combined_mask);
9 figure
10 H = D-D1;
11 imshow(H,[]);
12 figure;
13 [C,h]=contour(D-D1,20);
14 clabel(C,h);
```

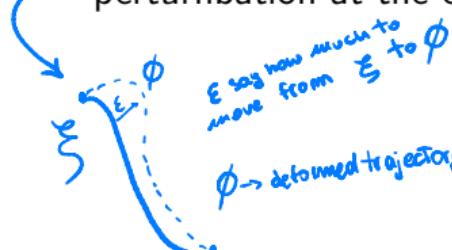
MATLAB Code Example

Listing 3: Distance field - Part 3

```
1     eps = 0.1;
2     c = zeros(size(H));
3     for i = 1:rows
4         for j = 1:cols
5             if H(i,j)<0
6                 c(i,j) = -H(i,j)+eps/2;
7             else
8                 if ((H(i,j) > 0)&(H(i,j)<=eps))
9                     c(i,j) = (1/(2*eps))*(H(i,j)-eps)^2;
10                end
11            end
12        end
13    end
```

Functional Gradients (1)

- ▶ Gradient methods are natural candidates for optimising because they leverage crucial first-order information about the shape of the objective functional.
- ▶ In our case, we consider the functional gradient, which generalises the steepest descent in the functional domain.
- ▶ $\bar{\nabla} U$ is the perturbation $\phi : [0, 1] \rightarrow \mathbb{R}^d$ that maximises $\bar{\nabla} U[\xi + \epsilon\phi]$ as $\epsilon \rightarrow 0$,
- ▶ Following [1], let $U[\xi] = \int v(\xi(t), \dot{\xi}(t)) dt$. Consider a perturbation ϕ such that the new trajectory $\bar{\xi} = \xi + \epsilon\phi$. Assume that $\phi(0) = \phi(1) = 0$ (there is no perturbation at the extremes of the trajectory).



$$\frac{df}{d\epsilon} = \int \left(\frac{\partial v}{\partial \bar{\xi}} \frac{\partial \bar{\xi}}{\partial \epsilon} + \frac{\partial v}{\partial \bar{\xi}'} \frac{\partial \bar{\xi}'}{\partial \epsilon} \right) dt$$

[1] Quinlan, S., Stanford University Computer Science Department. (1994). Real-time modification of collision-free paths.

Functional Gradients (2)

- ▶ Integrating by part and exploiting the assumptions

$$\frac{df}{d\epsilon} = \int \left(\frac{\partial v}{\partial \bar{\xi}} - \frac{d}{dt} \frac{\partial v}{\partial \bar{\xi}'} \right) \frac{\partial \bar{\xi}}{\partial \epsilon} dt$$

and

$$\frac{df}{d\epsilon} \Big|_{\epsilon=0} = \int \left(\frac{\partial v}{\partial \bar{\xi}} - \frac{d}{dt} \frac{\partial v}{\partial \bar{\xi}'} \right) \phi dt$$

- ▶ The perturbation ϕ that maximises this integral is aligned with $\frac{\partial v}{\partial \bar{\xi}} - \frac{d}{dt} \frac{\partial v}{\partial \bar{\xi}'}$

$$\nabla U[\xi] = \frac{\partial v}{\partial \bar{\xi}} - \frac{d}{dt} \frac{\partial v}{\partial \bar{\xi}'}$$

Method to compute the gradient of that functional (function of function)

Functional Gradients (3)

- We can exploit the functional gradient expression to perform a gradient descent:

$$\xi_{i+1} = \xi - \eta_i \bar{\nabla} U[\xi]$$

and halt when we reach a local minimum.

- Remember that our functional is composed of two parts:

$$U[\xi] = F_{obs}[\xi] + \lambda F_{smooth}[\xi]$$

$$\bar{\nabla} U[\xi] = \bar{\nabla} F_{obs}[\xi] + \lambda \bar{\nabla} F_{smooth}[\xi]$$

$$F_{smooth}[\xi] = \frac{1}{2} \int_0^1 \left\| \frac{d}{dt} \xi(t) \right\|^2 dt$$

$$F_{obs}[\xi] = \int_0^1 \int_B c(x(\xi(t), u)) \left\| \frac{d}{dt} x(\xi(t), u) \right\| du dt$$

The Smoothness Functional Gradient ∇F_{smooth}

Using the squared velocity term $F_{smooth}[\xi] = \frac{1}{2} \int_0^1 \|\xi'(t)\|^2 dt$, the corresponding gradient is straightforward:

$$\nabla F_{smooth}[\xi](t) = -\frac{d^2}{dt^2}\xi(t) \quad [33]$$

maximising

- In the continuous domain, ~~minimising~~ smoothness means minimising acceleration (second derivative).

The Obstacle Functional Gradient ∇F_{obs} (1)

The obstacle gradient is more complex involving integration over the body points B , the Jacobian J , velocity \mathbf{x}' , cost \mathbf{c} , and curvature κ :

$$\nabla F_{obs}[\xi] = \int_B J^T \|\mathbf{x}'\| \left[(I - \hat{\mathbf{x}}' \hat{\mathbf{x}}'^T) \nabla c - c \kappa \right] du$$

B :

Body Points

$\mathbf{x}', \mathbf{x}''$

Velocity, Acceleration

$\hat{\mathbf{x}}'$:

Normalised velocity vector

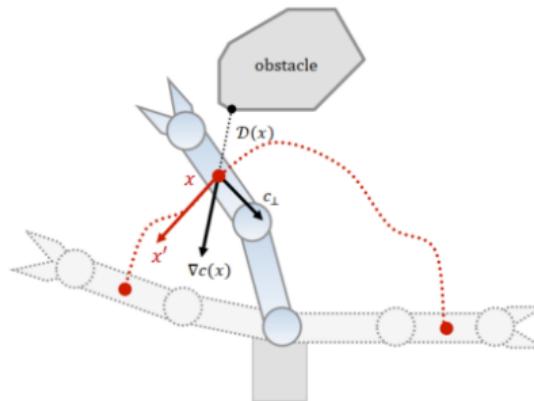
c :

Cost

$$\kappa = \|\mathbf{x}'\|^{-2} (I - \hat{\mathbf{x}}' \hat{\mathbf{x}}'^T) \mathbf{x}'' :$$

Curvature

The Obstacle Functional Gradient ∇F_{obs} (2)



$$\nabla F_{obs}[\xi] = \int_B J^T \|x'\| \left[\left(I - \hat{x}' \hat{x}'^T \right) \nabla c - c \kappa \right] du$$

- ▶ $\left(I - \hat{x}' \hat{x}'^T \right)$ is a projection matrix that projects workspace gradients **orthogonally** to the trajectory's direction of motion.
- ▶ The use of J^T maps the workspace forces back into the configuration space.



Covariant Gradient Descent: The Role of the Metric

- ▶ The definition of the CHOMP functional is independent from the trajectory representation and only related to the geometric properties of the path.
- ▶ However, the derivation mentioned above introduces a dependence on the representation because the functional gradient is derived from a Euclidean norm.
- ▶ CHOMP removes this dependence by properly defining a norm on trajectory perturbations, $\|\xi\|_A^2$, based on dynamical quantities. In plain words, we define a perturbation larger in norm if it differs for a large velocity (or acceleration).

I want to deform the trajectory
minimizing the acceleration needed for
this deformation



Covariant Gradient Descent: The Role of the Metric

- ▶ This norm is defined using an operator A :

$$\|\xi\|_A^2 = \int \sum_{n=1}^k \alpha_n (D^n \xi(t))^2 dt$$

, where α_n is a constant, and ξ is the perturbation.

- ▶ For instance, if $k = 1$, We have:

$$\|\xi\|_A^2 = \int \sum_{n=1}^k \alpha_n \xi'(t)^2 dt$$

- ▶ Functional gradients computed under this invariant norm ($\nabla_A U$) are **covariant** to re-parametrisation. This means that if we change the parametrisation, we have a well-defined rule to find the change in the norm.
- ▶ The relationship is given by the inverse of A :

$$\bar{\nabla}_A U[\xi] = A^{-1} \nabla U[\xi].$$

Covariant Update Insight: $A^{-1}\nabla U[\xi]$

The key to CHOMP's efficacy is the use of A^{-1} (the inverse of the smoothness Hessian) to condition the gradient update.

$$\bar{\nabla}_A U[\xi] = A^{-1} \nabla U[\xi].$$

- ▶ If A measures acceleration, applying A^{-1} to the gradient ∇U transforms a large, local force (from a collision) into a **smooth, global adjustment** of the entire trajectory].
- ▶ This adjustment adds only a **small amount of additional acceleration** to the overall trajectory.
- ▶ The benefit is computational: it is expected to be $O(n)$ times faster to converge than a standard Euclidean gradient method, as it adjusts large parts of the trajectory due to a single local impact.
- ▶ We will discuss numeric ways of defining A .

Practical Implementation: Waypoint Parametrisation (1)

Numerically, CHOMP works with a **uniform discretisation** of the trajectory into N waypoints $\xi \approx (q_1^T, \dots, q_N^T)^T$ over equal time steps Δt .

- ▶ The start (q_0) and end (q_{N+1}) configurations are typically fixed initially.
- ▶ The smoothness objective F_{smooth} is written as a series of finite differences:

$$F_{smooth}[\xi] = \frac{1}{2} \sum_{t=1}^{n+1} \left\| \frac{q_{t+1} - q_t}{\Delta t} \right\|^2$$

which can be written as:

$$F_{smooth}[\xi] = \frac{1}{2} \|K\xi + e\|^2$$

where:

$$K = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ -1 & 1 & 0 & \dots & 0 & 0 \\ 0 & -1 & 1 & \dots & 0 & 0 \\ \vdots & & \vdots & & \vdots & \\ 0 & 0 & 0 & \dots & -1 & 1 \\ 0 & 0 & 0 & \dots & 0 & -1 \end{bmatrix} \otimes I_{m \times m}, e = \begin{bmatrix} -q_0 \\ 0 \\ \vdots \\ 0 \\ q_{N+1} \end{bmatrix}$$

Practical Implementation: Waypoint Parametrisation (2)

Hence,

$$F_{smooth}[\xi] = \frac{1}{2} \|K\xi + e\|^2 = \frac{1}{2}\xi^T A\xi + \xi^T b + c$$

where

$$A = K^T K$$

$$b = k^T e$$

$$c = \frac{e^T e}{2}$$

Practical Implementation: Waypoint Parametrisation (3)

In a similar way, we can compute the gradient of the obstacle cost. Let us consider the time discretisation $\mathbf{t} = \{t_0, t_1, \dots, t_T\}$ and let the body points B be given by a set of primitives $B = \{u_1, \dots, u_b\}$. The workspace velocity of a body point i at time j can be approximated as $x'_{i,j} = \frac{x(u_i, t_j) - x(u_i, t_{j-1})}{\Delta T}$.

$$\begin{aligned}\overline{\nabla} F_{obs}[\xi] &= \int_B J^T \| \mathbf{x}' \| \left[\left(I - \hat{\mathbf{x}}' \hat{\mathbf{x}}'^T \right) \nabla c - c \kappa \right] du \\ &\approx \sum_{u_i \in B} J_{i,j}^T \left(\| \mathbf{x}'_{i,j} \| \left(\left(I - \hat{\mathbf{x}}'_{i,j} \hat{\mathbf{x}}'^T_{i,j} \right) \nabla c_{i,j} - c_{i,j} \kappa \right) \right)\end{aligned}$$



Gradient descent (1)

- ▶ The algorithmic idea of CHOMP is to set up an optimisation problem in which we seek to maximise decrease in our objective function by making small steps between ξ_i and ξ_{i+1} , where small means making small changes in the average acceleration (rather than in a representation specific parameters space).
- ▶ We first make a first order approximation of U

$$U[\xi] \approx U(\xi_i) + (\xi - \xi_i)^T \nabla U[\xi_i]$$

Gradient descent (1)

- ▶ The algorithmic idea of CHOMP is to set up an optimisation problem in which we seek to maximise decrease in our objective function by making small steps between ξ_i and ξ_{i+1} , where small means making small changes in the average acceleration (rather than in a representation specific parameters space).
- ▶ We first make a first order approximation of U

$$U[\xi] \approx U(\xi_i) + (\xi - \xi_i)^T \bar{\nabla} U[\xi_i]$$

- ▶ The optimisation problem we set up is

Acceleration

$$\xi_{i+1} = \arg \min_x iU[\xi_i] + (\xi - \xi_i)^T \bar{\nabla} U[\xi_i] + \frac{\eta}{2} \|\xi - \xi_i\|_M^2$$

Gradient descent (2)

- ▶ The term $\|\xi - \xi_i\|_M^2$ quantifies deviation. Since we want to keep the acceleration small we choose $M = A$
- ▶ By differentiating the above, we found the minimum at:

$$\bar{x}_{i+1} = \xi_i - \frac{1}{\eta} \nabla U[\xi_i]$$

which is our update rule-

- ▶ We will see a Matlab implementation.



Limitations of CHOMP

CHOMP inherits typical problems of optimisation on non-convex cost functions:

- ▶ **Local Optima:** It may converge to high-cost local minima, particularly in complex environments with 'narrow passages' [95].
- ▶ **Initialization Dependence:** Performance heavily relies on a good initial trajectory (hence the need for multiple initialisations or HMC) [47, 95].
- ▶ **Cost Function Alignment:** The current cost structure may not align with desired human preferences (e.g., a short collision path might be rated better than a longer, safer path) .
- ▶ **Constraints:** Handling trajectory-wide **hard constraints** is computationally time-consuming, while **soft constraints** (penalties) can lead to poor solutions .
- ▶ **Parameter Tuning:** Requires tuning of parameters like the trade-off coefficient λ and step size schedules [98].

Conclusion

CHOMP (Covariant Hamiltonian Optimization for Motion Planning) is an effective trajectory optimisation technique for high-dimensional robotic systems

- ▶ Key features include: **reparametrisation invariant** trajectory costs, reliance on **pre-computed signed distance fields**, and **pre-conditioned gradient descent** ($A^{-1}\nabla U$).
- ▶ The covariant update allows obstacles to efficiently 'push' the trajectory out of collision by generating smooth, global adjustments [8, 38].
- ▶ CHOMP is best suited for quickly solving 'simple' motion planning problems and plays a crucial **complementary role** by smoothing and improving trajectories generated by other techniques [99].
- ▶ Extensions (HMC, Goal Sets, Weight Scheduling) enhance its robustness and performance across various robotic domains

The source code for CHOMP and related work (TrajOpt) is often made freely available to the research community



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria
e Scienza dell'Informazione

Sampling-Based Motion Planning

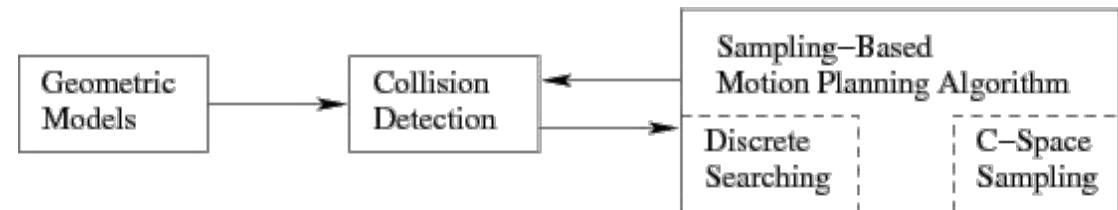
Luigi Palopoli (Credits Paolo Bevilacqua)



Sampling-Based Motion Planning

we don't know the map, we explore gradually.
we sample the space and check if there is an obstacle.

- Avoid explicit representation of C_{obs}
- Search and explore C-space using some sampling scheme
- "Black box" collision detection module, accounting for the geometric model of the robot and obstacles



you can decouple collision detection and space sampling



Sampling-Based Motion Planning

General Framework - Incremental sampling and searching:

1. Initialize an empty graph $G = (V, E)$
2. *Sample* a random configuration c and add to V if it lies in C_{free}
3. Try to connect the new configuration c with the *nearest neighbours* in V using the *local planner*
 - If local path is *collision free*, add the corresponding new edge to E
4. Repeat steps 2 and 3 until termination



Sampling-Based Motion Planning

Components:

- **Sampling:** sample configurations from C . Usually samples are drawn uniformly, but some different strategies may be applied for some specific, difficult problems
- **Collision detection:** a collision detection module to detect whether a given configuration or local path lies in C_{free} (but the check is performed in the workspace, by checking whether the robot geometry $A(q)$ intersects some obstacle)



Sampling-Based Motion Planning

Components:

- **Nearest neighbour:** need an efficient strategy to find the k nearest neighbours (or neighbours within a certain radius) in V for a given new point in C (some specific spatial data structures, e.g. *kd-trees*, may be used to speed up the search)
- **Local planning:** need a function to connect two given configurations. The simplest possible solution, applicable to all holonomic robots, is based on the direct connection of the two configurations with a straight line segment. More complex local planners are required for robots with non-holonomic constraints (e.g. Dubins car)

Instead of connecting points with
straight lines you can use Dubins manouvers





Sampling

For example
periodic numbers
→ you cannot express as a
finite fractional number

- C is uncountably infinite
- Planning algorithm can consider only a finite, (small) number of samples
- Need careful consideration on the sampling strategy
- The order in which samples are chosen is critical, since only a small subset of the (infinite) sequence of samples will be actually used



Sampling

Denseness:

- The ideal (infinite) sequence of samples should reach every point in C
- Not possible, since C is uncountably infinite
- However, it is possible to construct a sequence getting arbitrarily close to every element of C (assuming $C \subseteq \mathbb{R}^m$):

Let U, V be subsets of a topological space

- U is *dense* in V if $\text{cl}(U) = V$ (in simple words, adding the boundary points to U produces V)
- E.g. $(0,1) \subset \mathbb{R}$ is dense in $[0,1] \subset \mathbb{R}$, \mathbb{Q} is countable and dense in \mathbb{R}
- A sequence is dense if the underlying set is dense



Sampling

- A random sequence is dense with probability one
- Simple solution: use (uniform) random sampling to select points in C
- Independent random samples extend easily to Cartesian Products
- i.e. if $X = X_1 \times X_2$, and x_1, x_2 are uniform random samples from X_1, X_2 , then (x_1, x_2) is a uniform random sample for X
- Difficult to combine samples generated with nonrandom (i.e. deterministic) methods over Cartesian product



Sampling

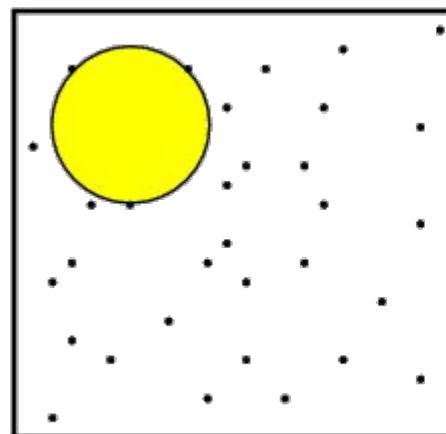
(Deterministic) Low-Dispersion sampling:

- Optimize dispersion, i.e. try to minimize the radius of the largest uncovered area

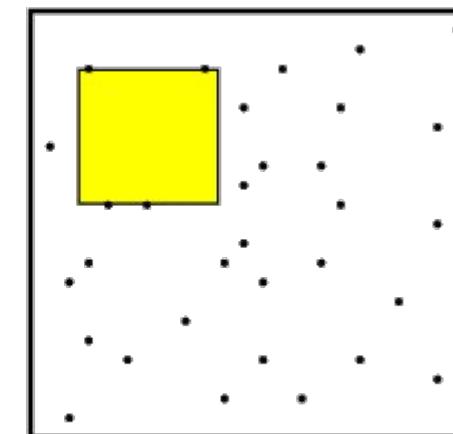
$$\delta(P) = \sup_{x \in X} \left\{ \min_{p \in P} \{\rho(x, p)\} \right\}$$

$\rho \rightarrow$ notion of distance
 $X \rightarrow$ free space
 $p \rightarrow$ point where i am

where P is a finite set of samples in a metric space (X, ρ)



(a) L_2 dispersion



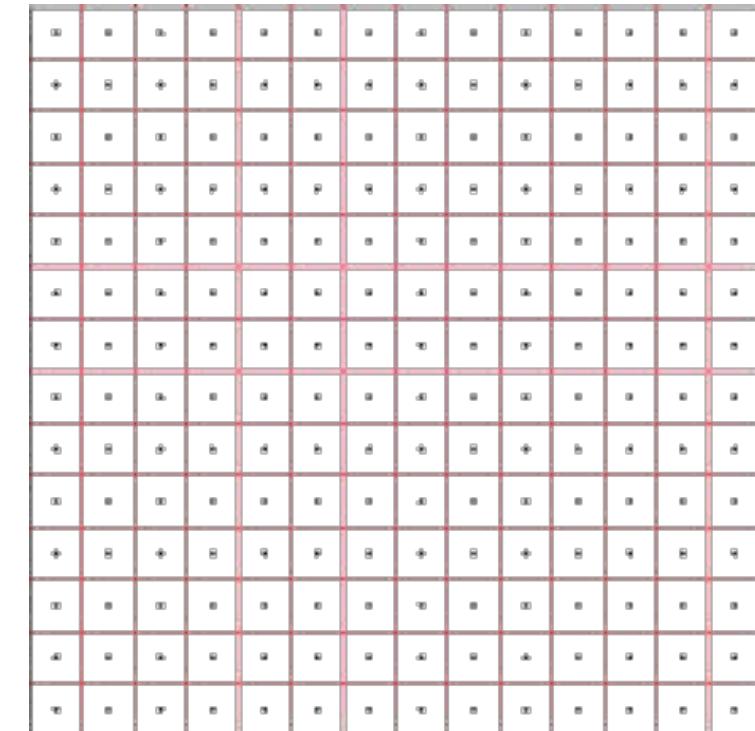
(b) L_∞ dispersion



Sampling

(Deterministic) Low-Dispersion sampling:

- Sukharev grid: given $X = [0,1]^n$ and the number of samples k , partition X into a grid of cubes and use the center of each cube as sample point
- The number of cubes per axis is $\lfloor k^{1/n} \rfloor$
- If $k^{1/n}$ is not integer, the leftover points may be placed anywhere (dispersion is not affected)

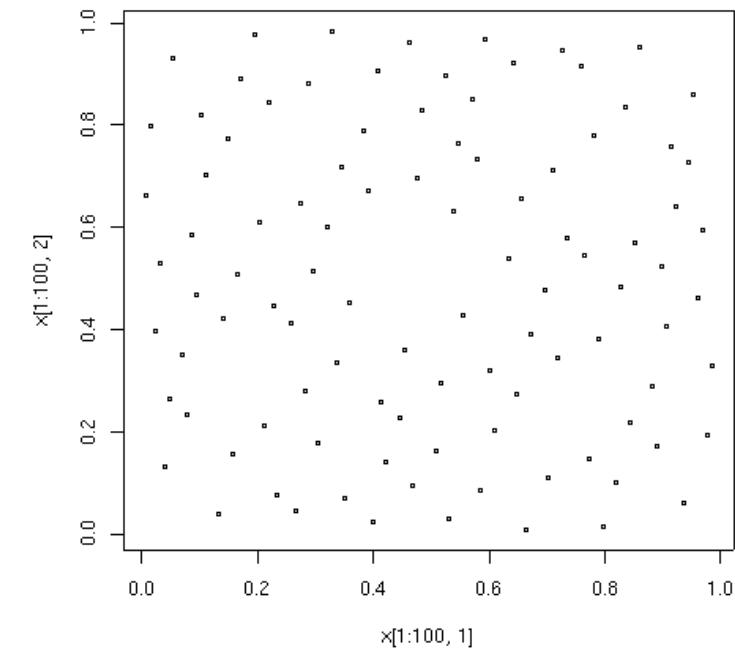




Sampling

Low-Discrepancy sampling:

- Points aligned with coordinate axis may not be suitable for motion planning (e.g. presence of corridors in C_{free})
- Development of sampling algorithms to reduce these alignments, and generate sequences of samples more uniformly distributed

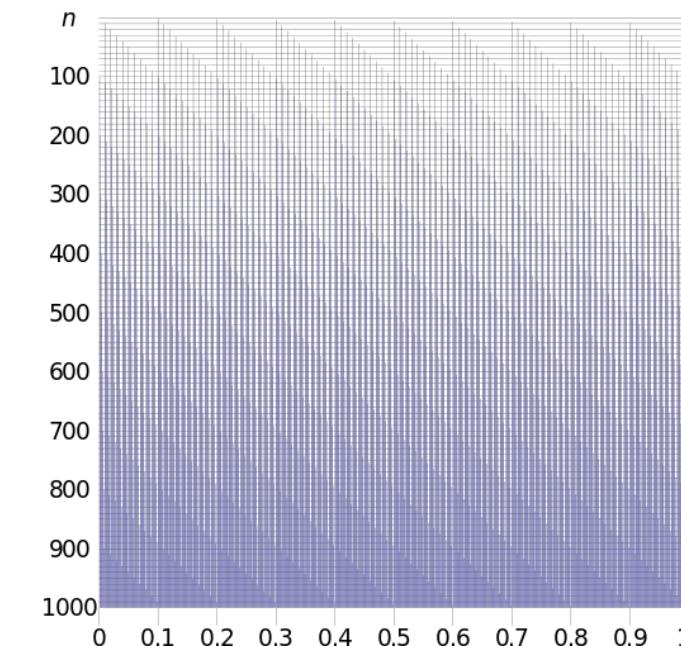




Sampling

Low-Discrepancy sampling:

- Van der Corput sequence (1D, $C = [0,1]$)





Example

- Suppose we divide our $[0, 1]$ segment into 16 buckets



- We start by point 0, which covers the two extremes



Dispersion = 1



Example

- The first point is $1/16$.
 - In binary it is 0.0001
 - If we revert the bit we get: $0.1000 = 1/2$

Dispersion = 0.5



- The second point is $2/16$.
 - In binary it is 0.0010
 - If we revert the bit we get: $0.0100 = 1/4$

Dispersion = 0.5





Example

- The third point is $3/16$.
 - In binary it is 0.0011
 - If we revert the bit we get: $0.1100 = 3/4$

Dispersion = 0.25



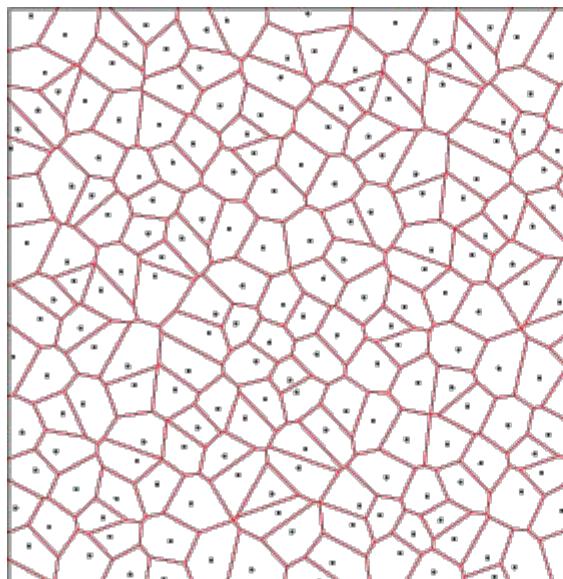
- ...and so on....
- As we can see, at each step we minimise the dispersion



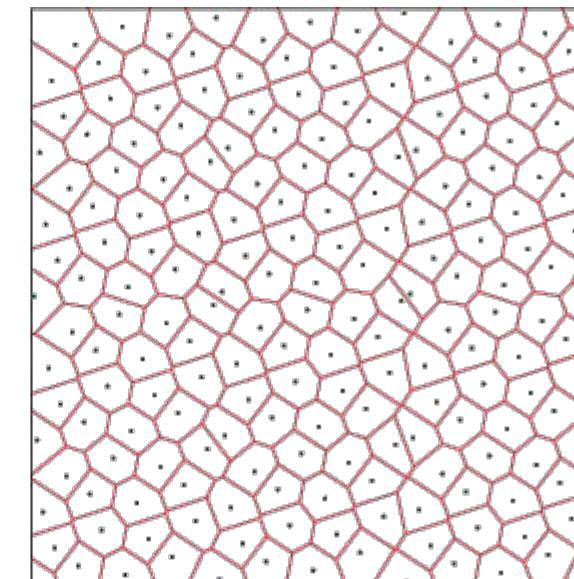
Sampling

Low-Discrepancy sampling:

- Halton/Hammersley: extension of Van der Corput sequence to n-dimensional spaces



(a) 196 Halton points



(b) 196 Hammersley points



Nearest Neighbours

- To perform nearest-neighbour search, sampling-based algorithms require a distance function over C to measure the distance between two configurations
- In general, a metric space is used
- A metric space (X, ρ) is a topological space X equipped with a function $\rho: X \times X \rightarrow \mathbb{R}$ with the following properties (for $a, b, c \in X$):
 - **Nonnegativity** $\rho(a, b) \geq 0$
 - **Reflexivity** $\rho(a, b) = 0$ if and only if $a = b$
 - **Symmetry** $\rho(a, b) = \rho(b, a)$
 - **Triangle inequality** $\rho(a, b) + \rho(b, c) \geq \rho(a, c)$



Nearest Neighbours

- The most important family of metrics over \mathbb{R}^n is given by

$$\rho(x, x') = \left(\sum_{i=1}^n |x_i - x'_i|^p \right)^{\frac{1}{p}}$$

- For each value of p , this metric is called an L_p metric
- The most common cases are:
 - L_2 : the Euclidean metric (the familiar distance in \mathbb{R}^n)
 - L_1 : the Manhattan metric (i.e. distance along axis-aligned grid)
 - L_∞ : limit for $p \rightarrow \infty$

$$L_\infty(x, x') = \max_{1 \leq i \leq n} \{|x_i - x'_i|\}$$



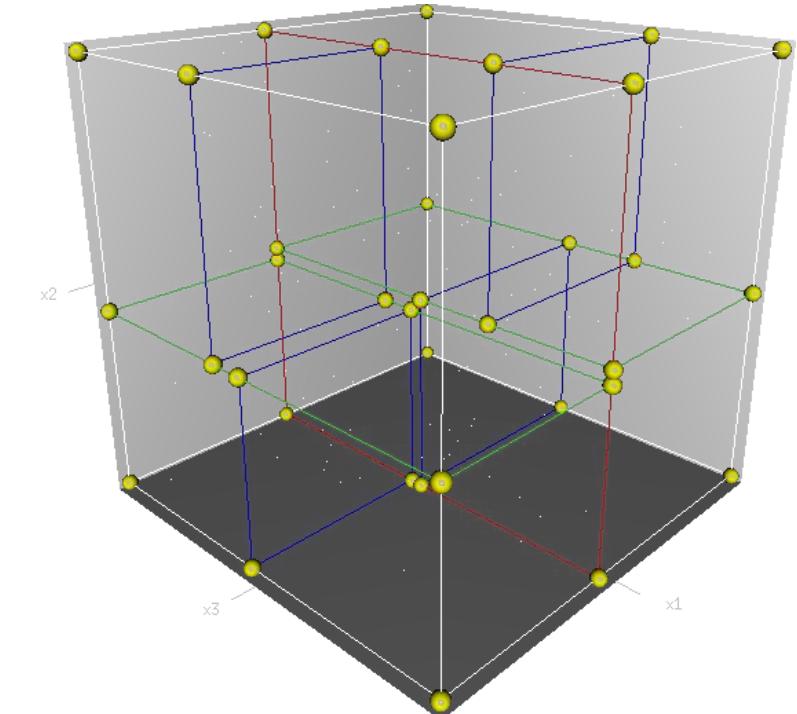
Nearest Neighbours

- Sometimes planning algorithms use distance functions that fail to satisfy some of the metric axioms
- These are called *pseudometrics*
- Sometimes the pseudometric is not symmetric, e.g. it may take longer to move back from q_2 to q_1 than moving forward from q_1 to q_2 (like in the case of the Dubins car)



Nearest Neighbours

- The explicit computation of the nearest neighbours may become computationally expensive when the number of nodes of the graph becomes large (i.e. several thousands)
- In this case, a suitable, efficient data structure for nearest neighbour search should be adopted
- A common approach is based on the adoption of KD-trees



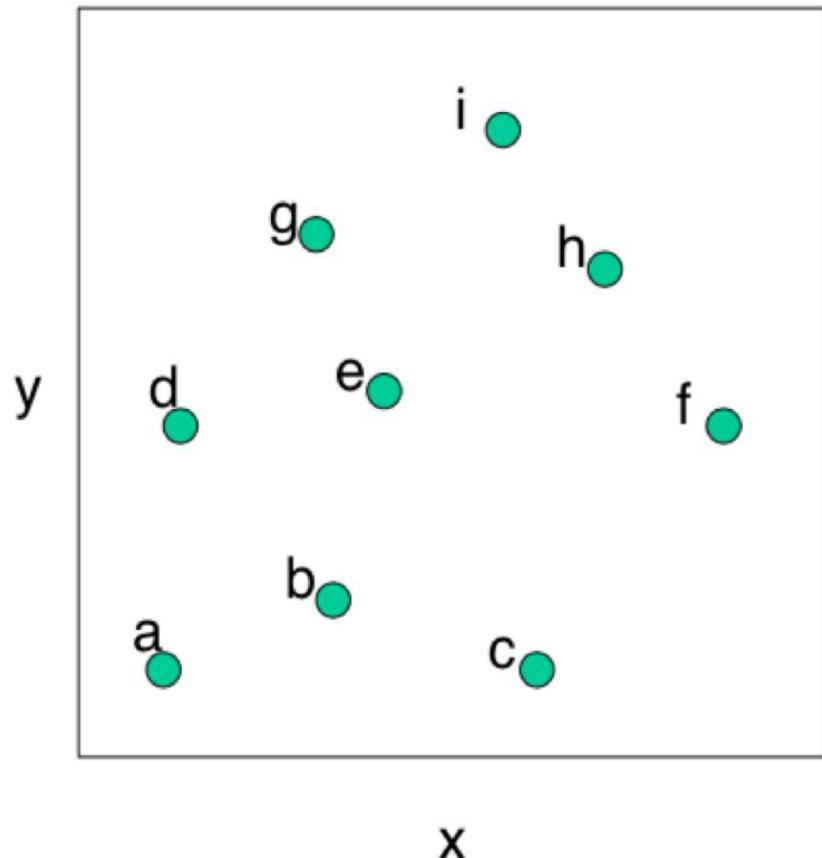


KD Trees Example

- K-D Tree construction
 - If there is just one point, form a leaf with that point.
 - Otherwise, divide the points in half by a line perpendicular to one of the axes.
 - Recursively construct k-d trees for the two sets of points.
 - Division strategies
 - - divide points perpendicular to the axis with widest spread.
 - - divide in a round-robin fashion



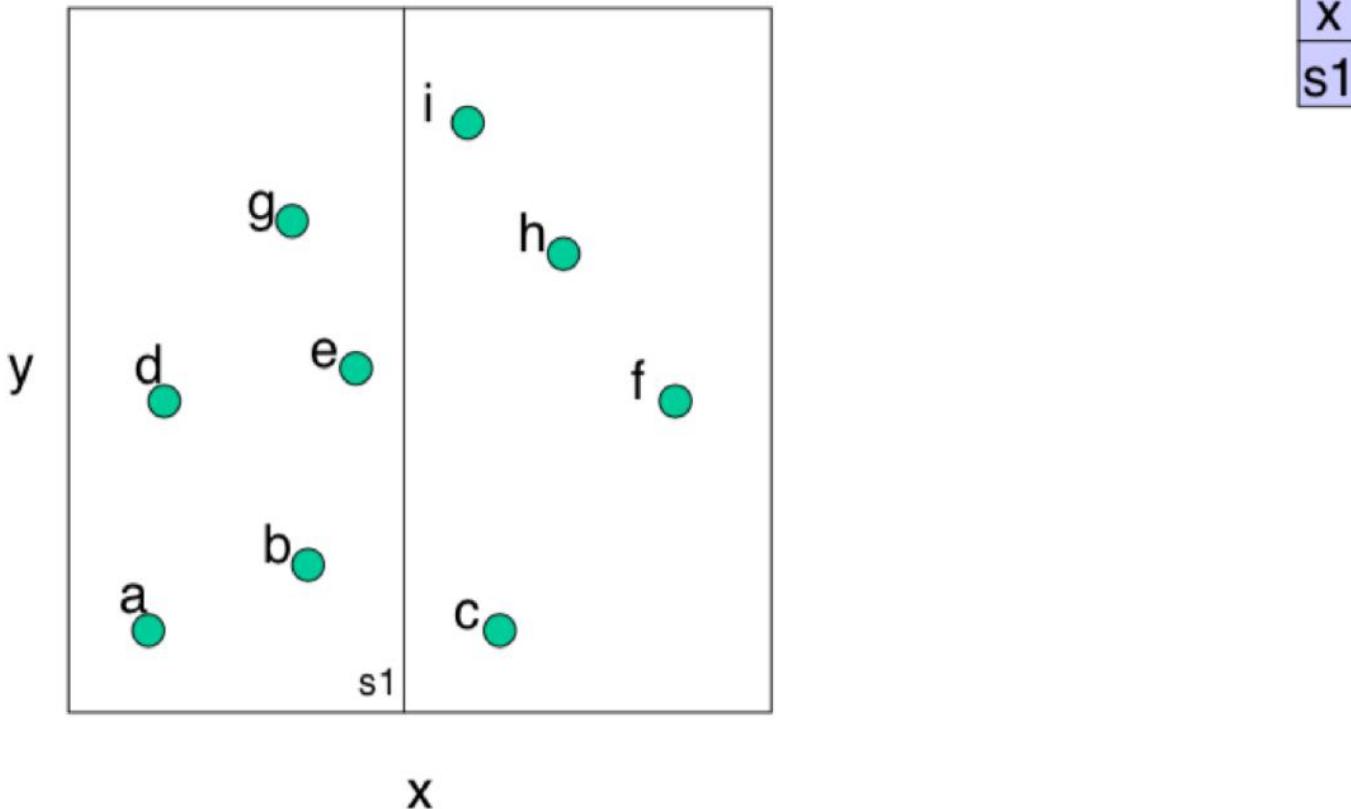
Example



Split orthogonally to the widest spread

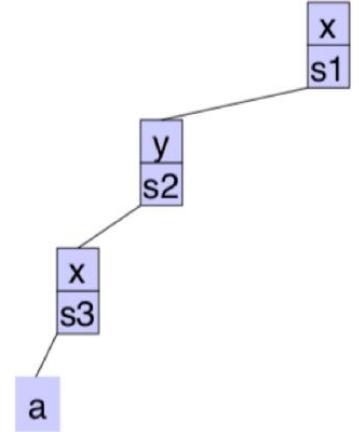
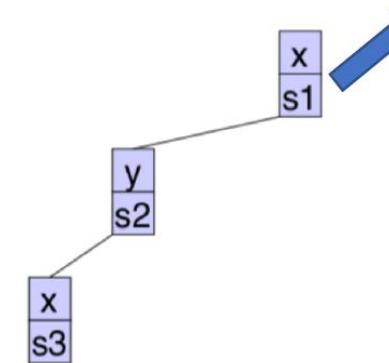
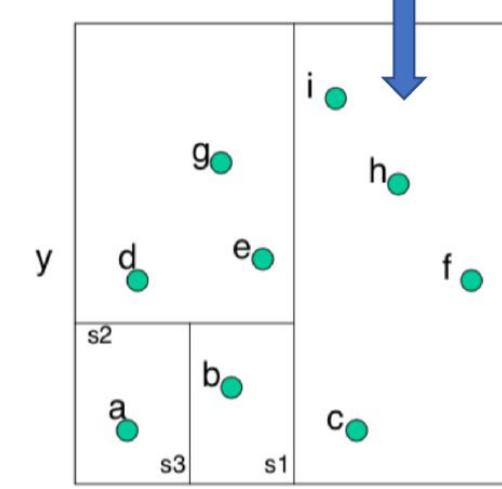
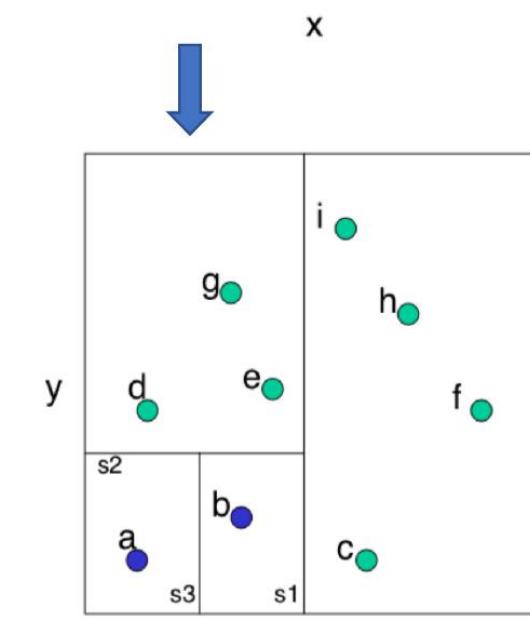
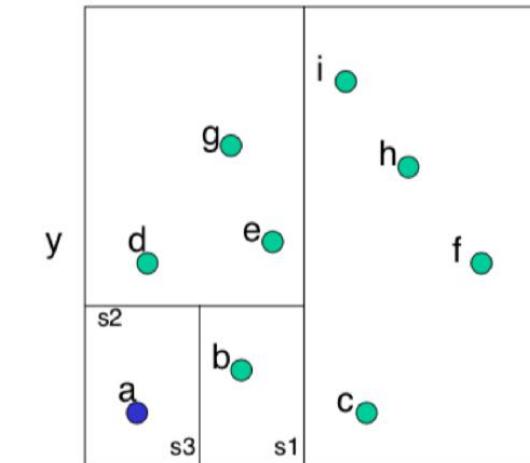
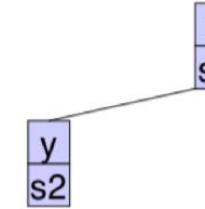
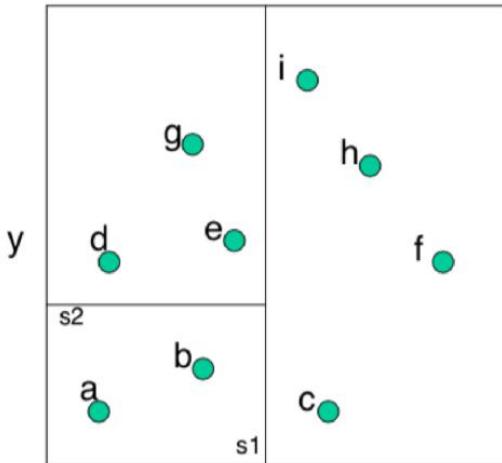


Example



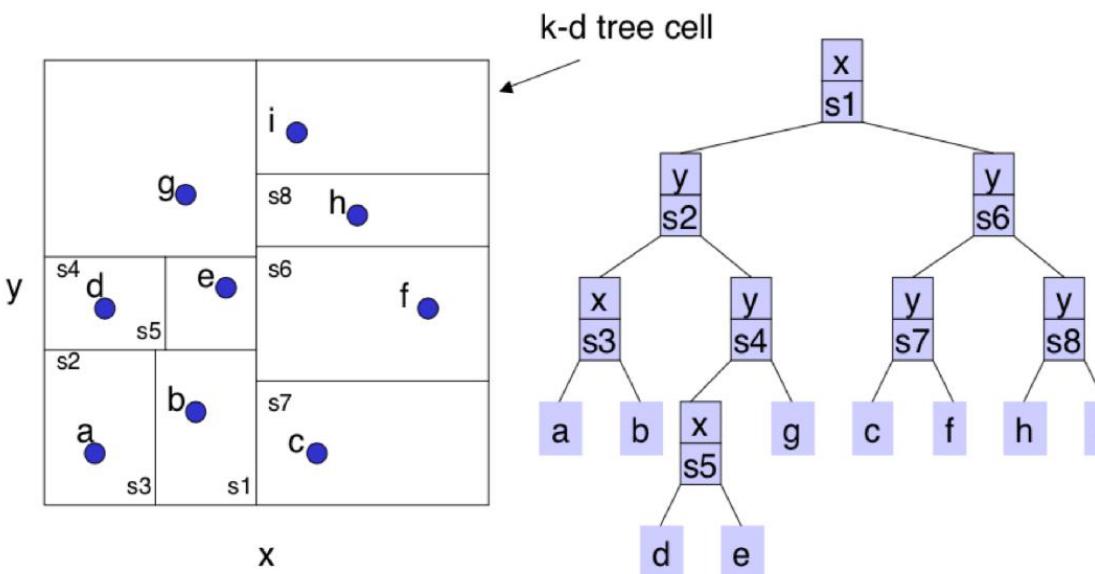


Example





Final Result



The use of the median
Produces a balanced tree



Node Structure

- A node has 5 fields
 - axis (splitting axis)
 - value (splitting value)
 - left (left subtree)
 - right (right subtree)
 - point (holds a point if left and right children are null)

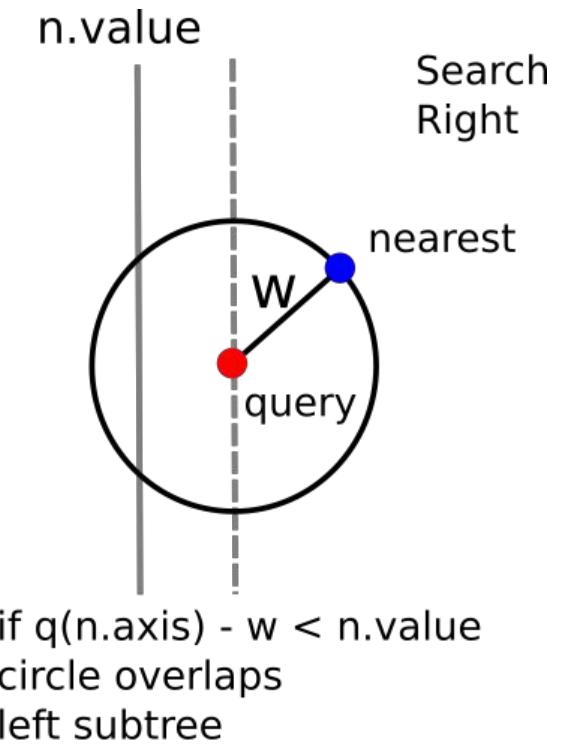
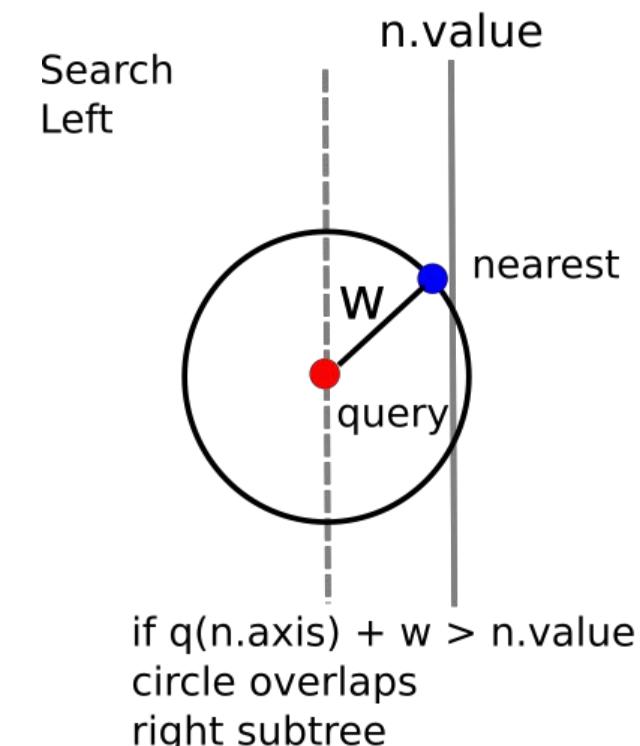


KD Tree nearest neighbour search

- Search recursively to find the point in the same cell as the query.
- On the return, search each subtree where a closer point than the one you already know about might be found.

```
NNS(q: point, n: node, p: ref point w: ref distance)
if n.left = n.right = null then {leaf case}
    w' := ||q - n.point||;
    if w' < w then w := w'; p := n.point;
else
    if q(n.axis) ≤ n.value then
        search_first := left;
    else
        search_first := right;
    if (search_first == left)
        NNS(q, n.left, p, w);
        if q(n.axis) + w > n.value then NNS(q, n.right, p, w);
    else // search_first == right
        NNS(q, n.right, p, w);
        if q(n.axis) - w ≤ n.value then NNS(q, n.left, p, w);
```

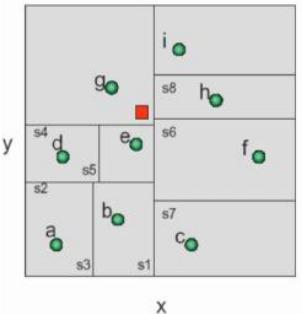
Initial Call: NNS(q, root, p, infinity)



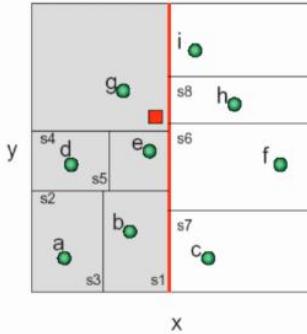


Example

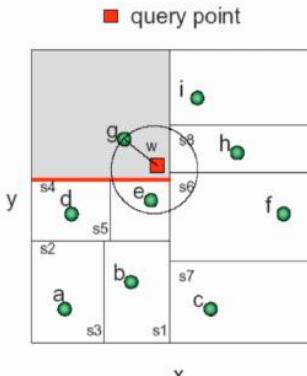
■ query point



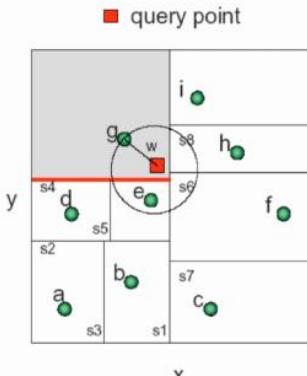
■ query point



■ query point



■ query point

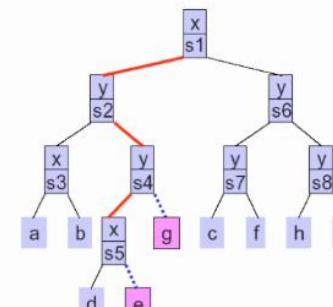
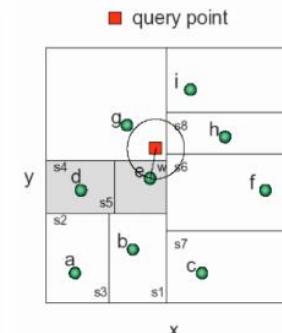
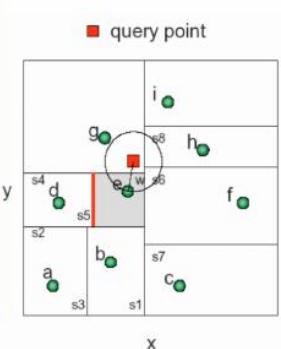
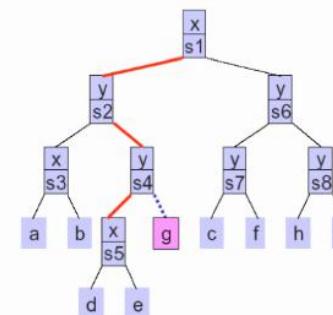
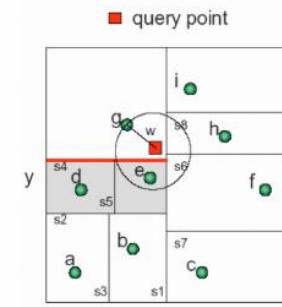
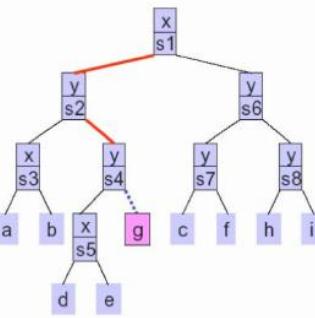
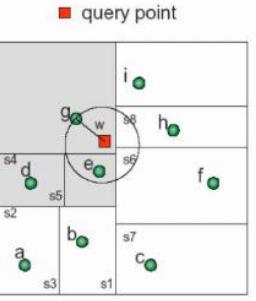


Leaf found. We set the distance to a finite value



Example

Distance from g overlap nearby trees. We have to continue with the search on the right



We check the distance from the border of the cell. If it overlaps other cells we have to continue

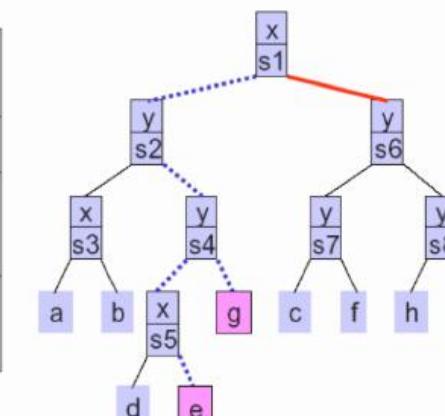
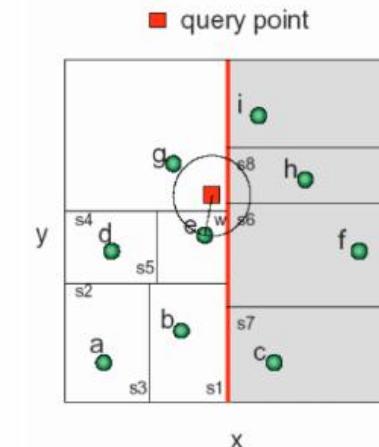
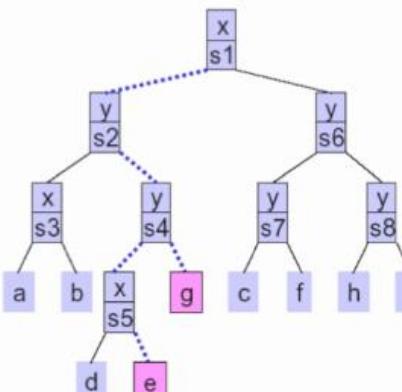
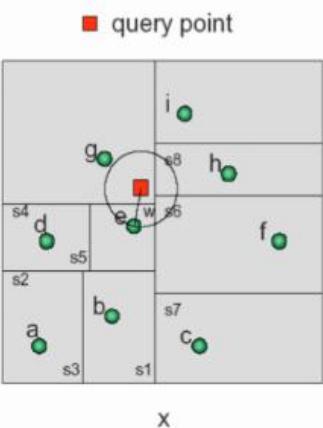
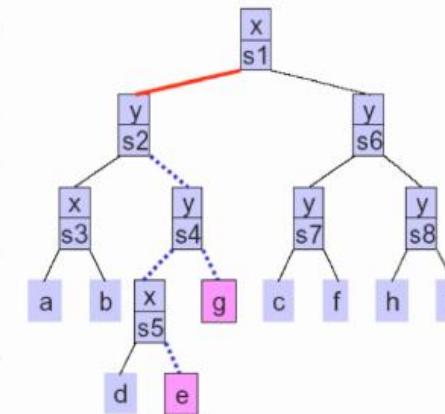
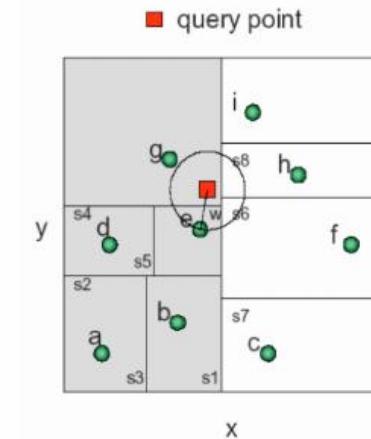
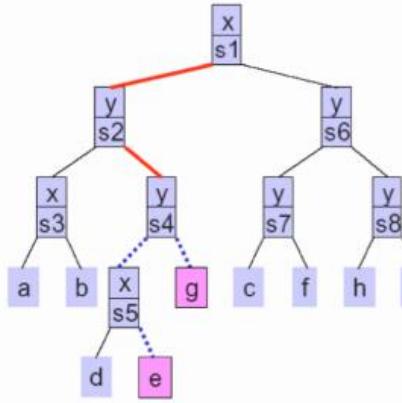
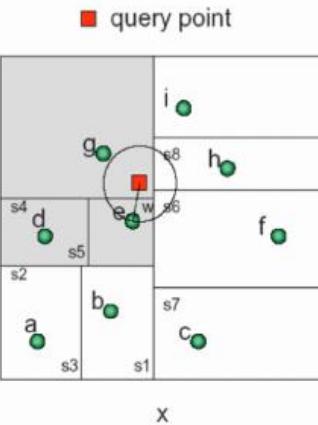
```

NNS(q: point, n: node, p: ref point w: ref distance)
if n.left = n.right = null then {leaf case}
  w' := ||q - n.point||;
  if w' < w then w := w'; p := n.point;
else
  if q(n.axis) ≤ n.value then
    search_first := left;
  else
    search_first := right;
  if (search_first == left)
    if q(n.axis) - w ≤ n.value then NNS(q, n.left, p, w);
    if q(n.axis) + w > n.value then NNS(q, n.right, p, w);
  else// search_first == right
    if q(n.axis) + w > n.value then NNS(q, n.right, p, w);
    if q(n.axis) - w ≤ n.value then NNS(q, n.left, p, w);

initial call NNS(q, root, p, infinity)
  
```



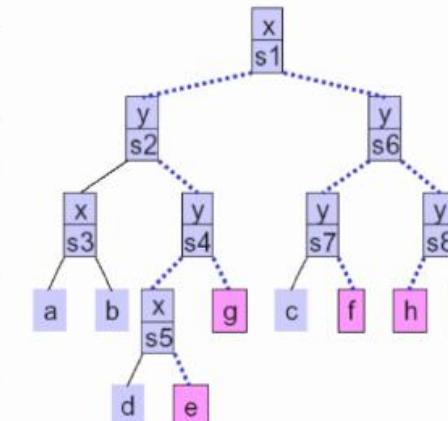
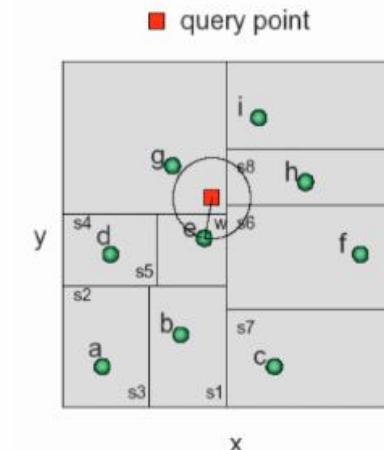
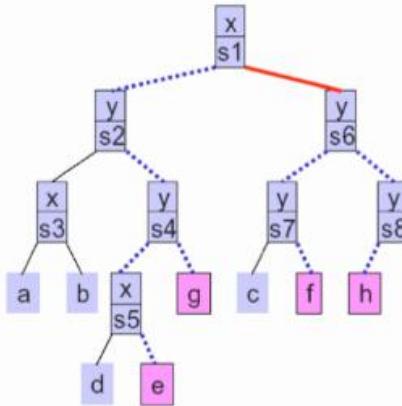
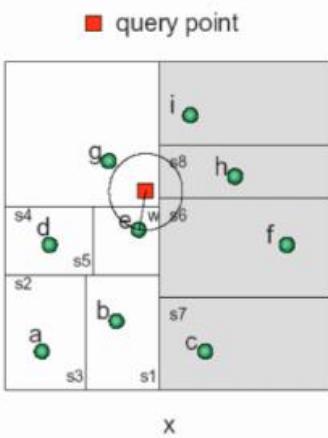
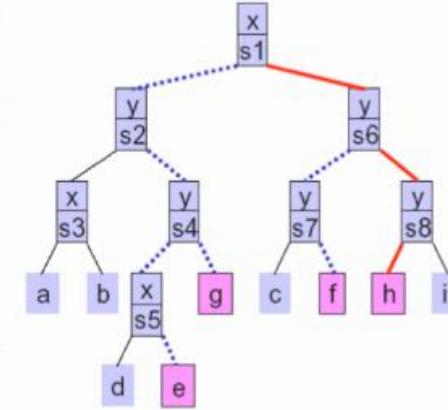
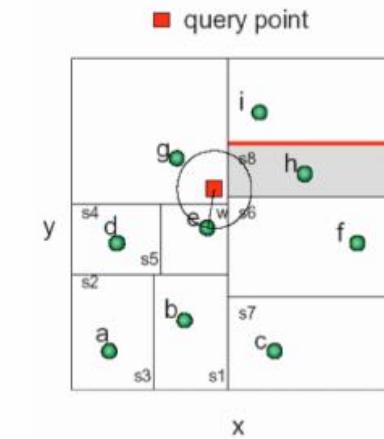
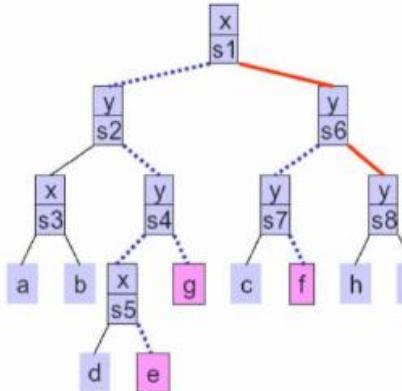
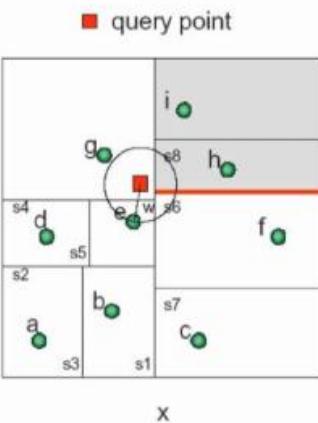
Example



...and so on and
so forth...



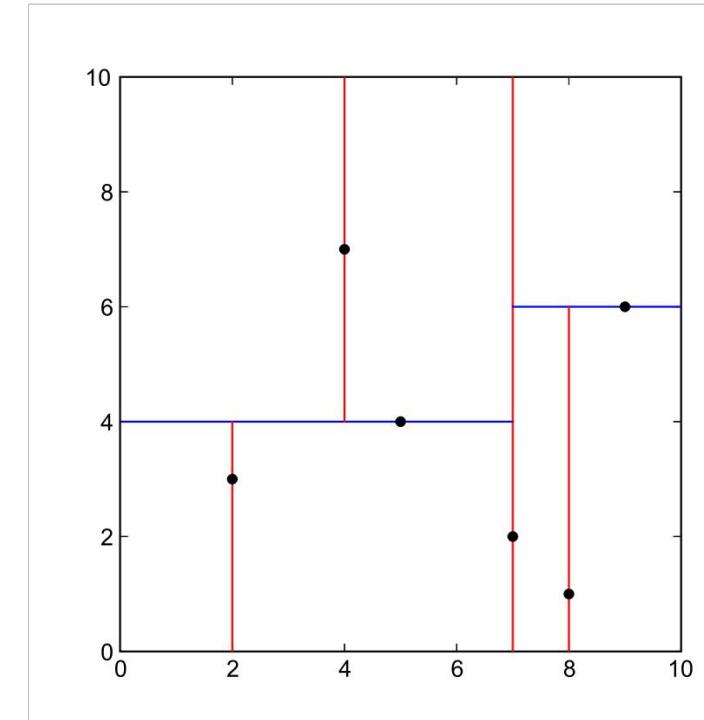
Example - Final Steps





Nearest Neighbours

- A KD-tree partitions the space by recursively subdividing the set of points with alternating axis-aligned hyperplanes
- Construction time $O(dn \log n)$
- Memory $O(dn)$
- Query time logarithmic in n , but exponential in d (suitable up to 10D)
- Many libraries available, e.g. FLANN





Implementation Example



Single Query vs Multiple Query

Single-query model:

- (q_{init}, q_{goal}) given once per robot and obstacle set
- No advantages in precomputation
- Sampling-based motion planning to search a feasible path within C

Multiple-query model:

- Many queries with different (q_{init}, q_{goal}) pairs, but with the same robot and obstacle set
- Convenient to preprocess the environment, and construct a sampling-based roadmap modelling the topology of the environment
- Sampling-based motion planning to explore and create a model for C



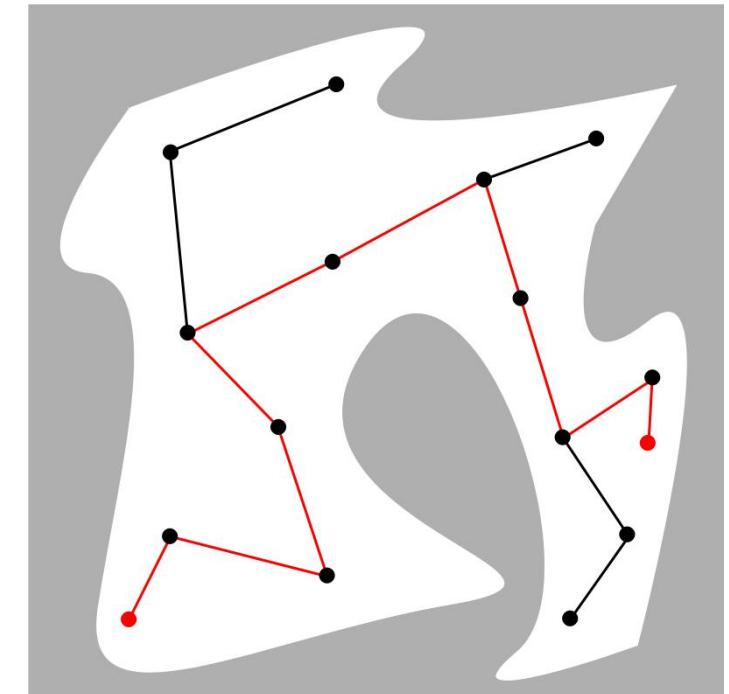
Single Query Model (incremental Sampling)

- 1) **Initialization:** initialize $G(V, E)$, by inserting at least one vertex (usually q_{init} , q_{goal} or both)
- 2) **Vertex Selection:** choose a vertex $q_{cur} \in V$ for expansion
- 3) **Local Planning:** for some $q_{new} \in C_{free}$, try to construct a path τ_s lying in C_{free} and connecting q_{cur} to q_{new}
- 4) **Vertex/Edge Insertion:** insert q_{new} into V and τ_s into E
- 5) **Solution Validity:** if G contains a valid solution from q_{init} to q_{goal} terminate the search.
- 6) **Termination Criteria:** If some termination criteria has been reached (e.g. maximum number of iterations) report failure, otherwise repeat the algorithm by selecting a new vertex.



Single Query Model

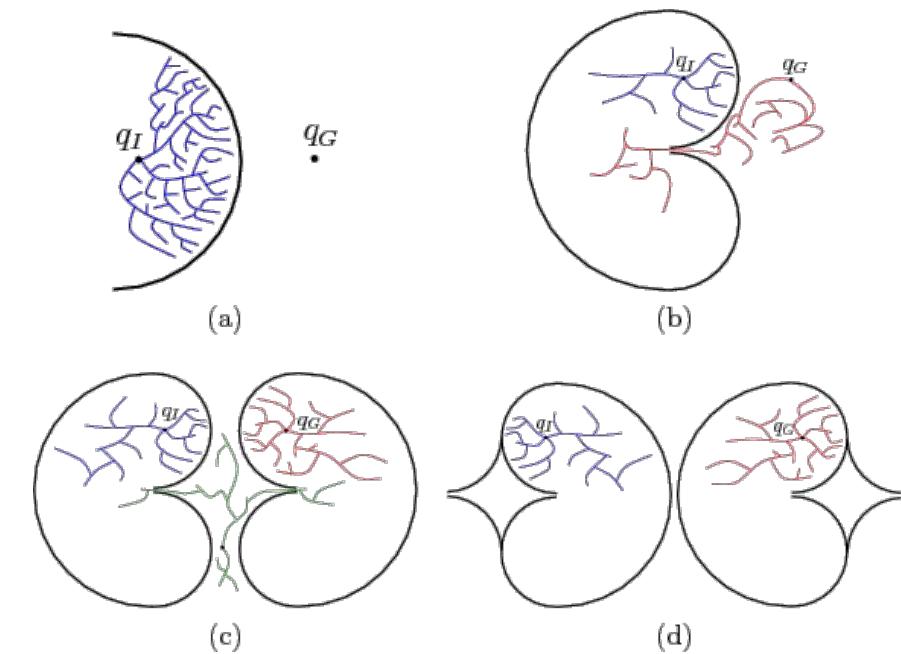
- Different variants of single-query sampling based algorithms, depending on the strategy adopted to implement and solve the different steps
- The role of the *Local Planning Method* is to generate, short, simple collision-free path segments to be inserted into the graph





The Bug Trap

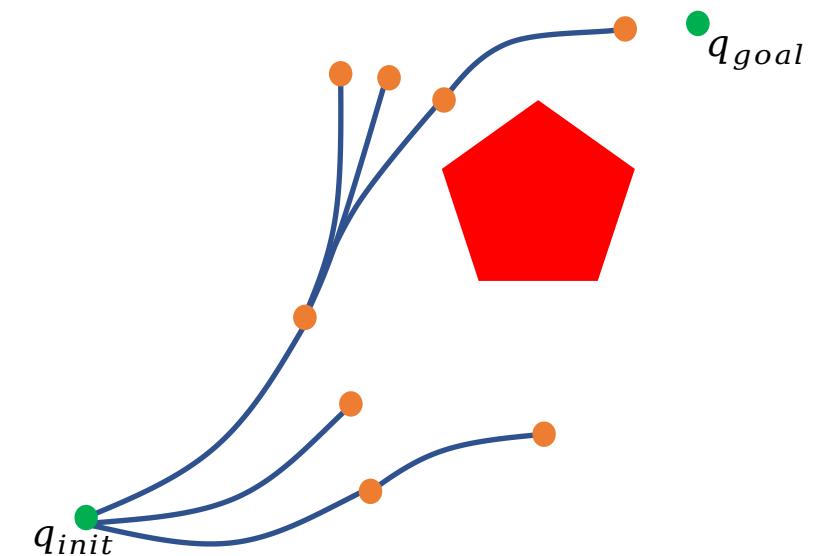
- A common problem for sampling based algorithms based on a forward search is the *bug trap*
- Several variants based on the growth of more than a tree:
 - Bidirectional method: grow two trees, one from q_{init} and one from q_{goal}
 - Vertices to expand are selected alternately from the two trees
 - Local planning is used both to explore new parts of C_{free} and to generate paths connecting the two trees
 - Possible to generalize and extend to more than two trees





Randomly Exploring Dense Trees

- Incremental construction of a search tree that gradually improves the resolution of the discretization
- In the limit, the tree densely covers the space
- Use a dense sequence of samples to guide the incremental construction of the tree
- If the sequence of samples is generated randomly, the algorithm is called Randomly Exploring Random Tree (RRT)
- In general, this family of trees, both using a random or deterministic sequence of samples, is called Rapidly Exploring Dense Tree





RDT Exploration

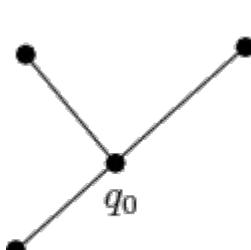
- Goal: to get as close as possible to every configuration, starting from q_{init}
- Let α denote an infinite, dense sequence of samples in C
- $\alpha(i)$ denotes the i^{th} sample
- $S = \bigcup_{e \in E} e([0,1])$ denotes the swath of the graph, i.e. the set of all points reached by G



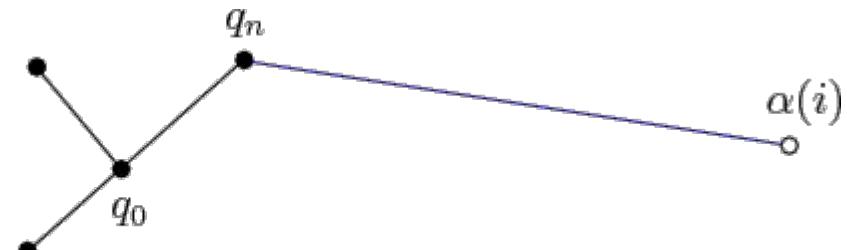
RDT Exploration

Construction of a dense tree in C (without any obstacle)

```
SIMPLE_RDT( $q_0$ )
1  $\mathcal{G}.\text{init}(q_0);$ 
2 for  $i = 1$  to  $k$  do
3    $\mathcal{G}.\text{add\_vertex}(\alpha(i));$ 
4    $q_n \leftarrow \text{NEAREST}(S(\mathcal{G}), \alpha(i));$ 
5    $\mathcal{G}.\text{add\_edge}(q_n, \alpha(i));$ 
```



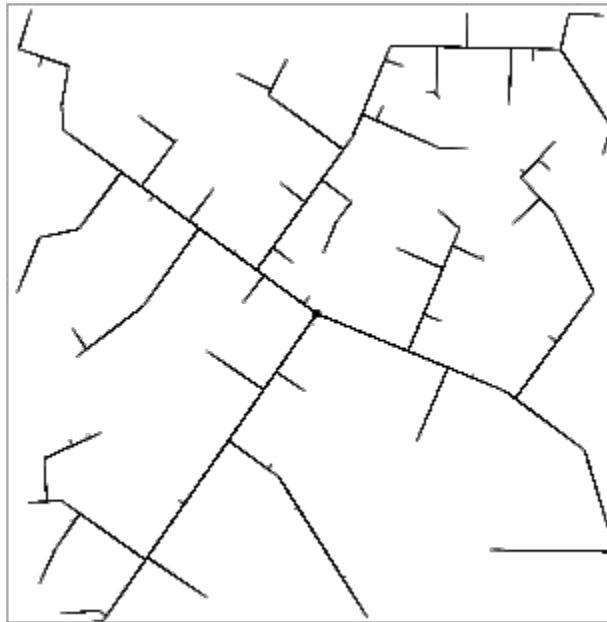
(a)



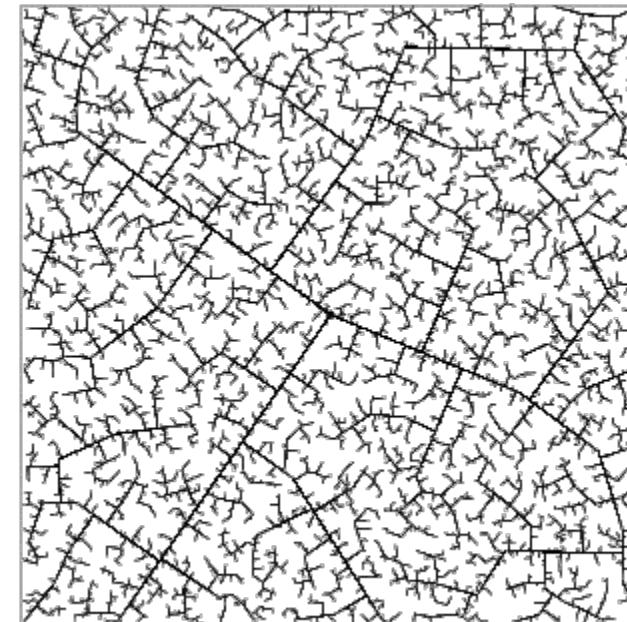
(b)



RDT Exploration



45 iterations



2345 iterations

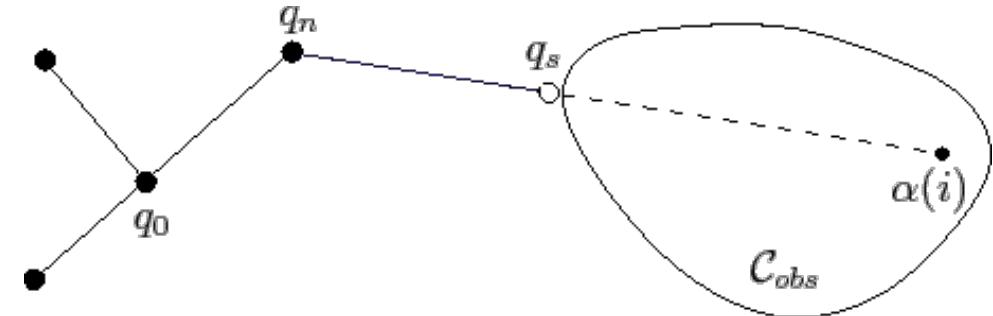
- The resolution (and the covered area) gradually increases as the iterations continue
- Ideal behaviour for sampling based motion planning



RDT Exploration (with obstacles)

RDT(q_0)

```
1  $\mathcal{G}.\text{init}(q_0);$ 
2 for  $i = 1$  to  $k$  do
3    $q_n \leftarrow \text{NEAREST}(S, \alpha(i));$ 
4    $q_s \leftarrow \text{STOPPING-CONFIGURATION}(q_n, \alpha(i));$ 
5   if  $q_s \neq q_n$  then
6      $\mathcal{G}.\text{add\_vertex}(q_s);$ 
7      $\mathcal{G}.\text{add\_edge}(q_n, q_s);$ 
```



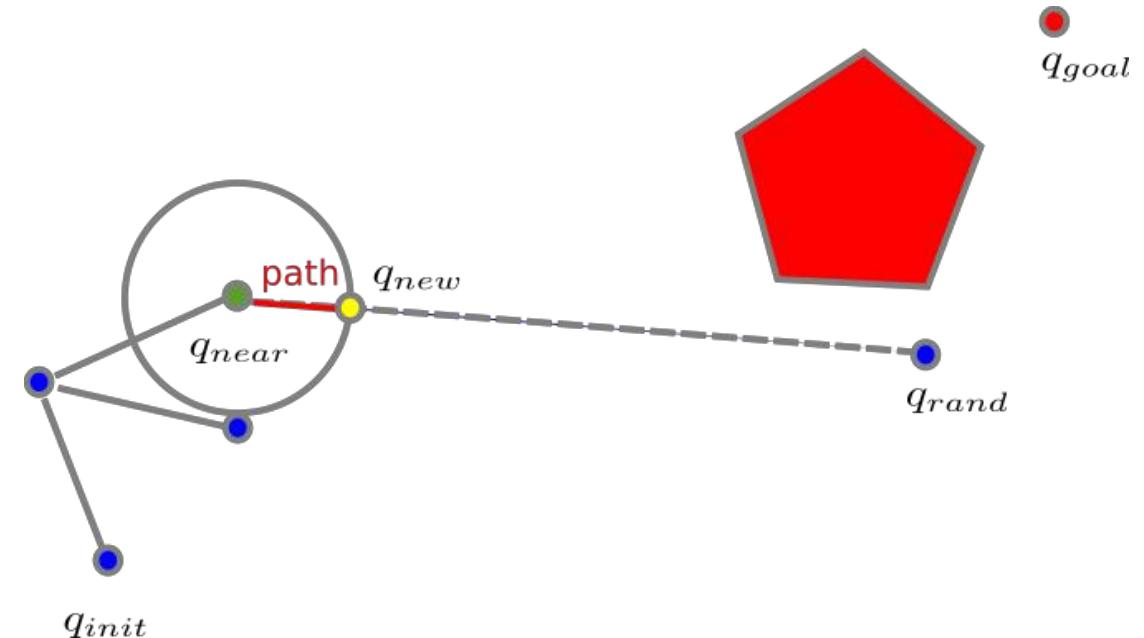
- Sampling based exploration considering an environment with obstacles
- The new edge may not reach $\alpha(i)$ (stop it before hitting an obstacle)
- The configuration q_s corresponding to the end of the edge is inserted into G



Rapidly Exploring Random Tree - RRT

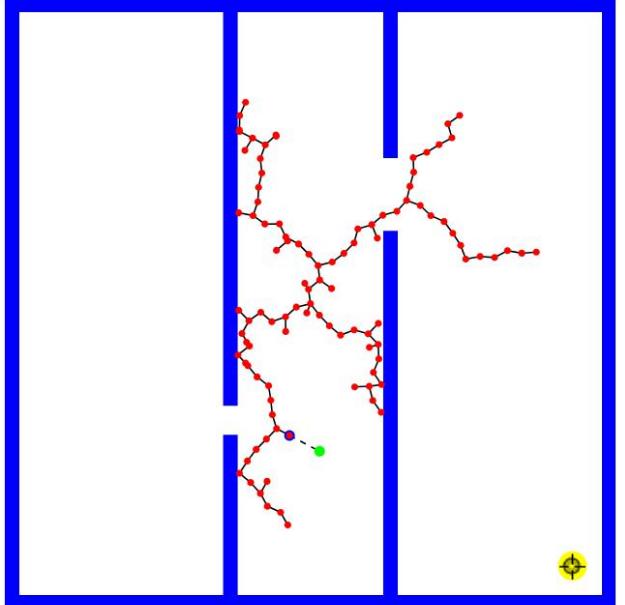
RRT

```
1:  $G.addVertex(q_{init})$ 
2: while  $q_{goal}$  not reached do
3:    $q_{rand} = \text{nextSample}()$ 
4:   if  $\text{CLEAR}(q_{rand})$  then
5:      $q_{nearest} = \text{NEAREST}(G, q_{rand})$ 
6:      $(q_{new}, path) = \text{EXTEND}(q_{nearest}, q_{rand})$ 
7:     if  $\neg\text{COLLISION}(path)$  then
8:        $G.addVertex(q_{new})$ 
9:        $G.addEdge(q_{nearest}, q_{new}, path)$ 
10:    end if
11:   end if
12: end while
```

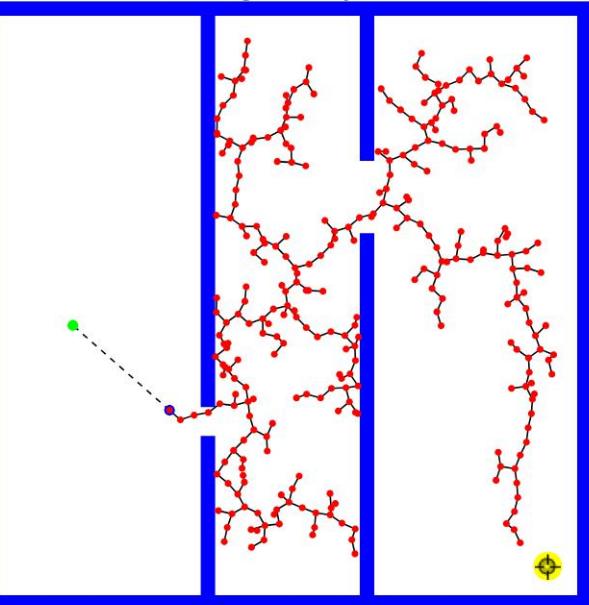


- At each step, sample one point from C_{free} and try to connect it to the closest vertex in the tree
- Probabilistically complete:
 - If a solution exists, it will be found with probability 1

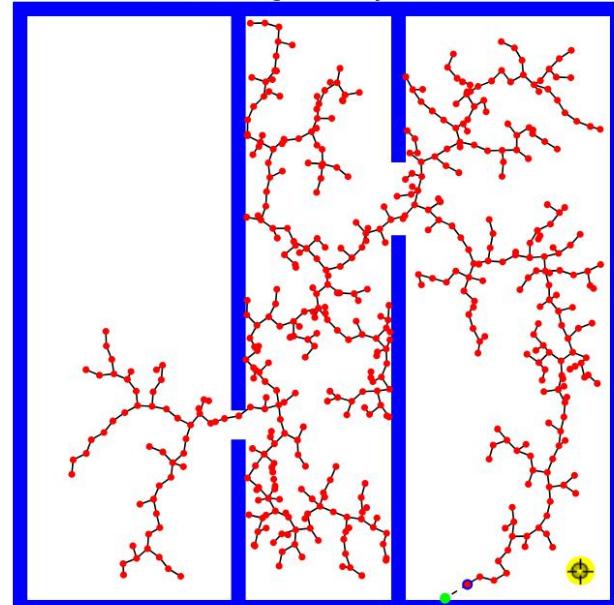
101 nodes, goal not yet reached



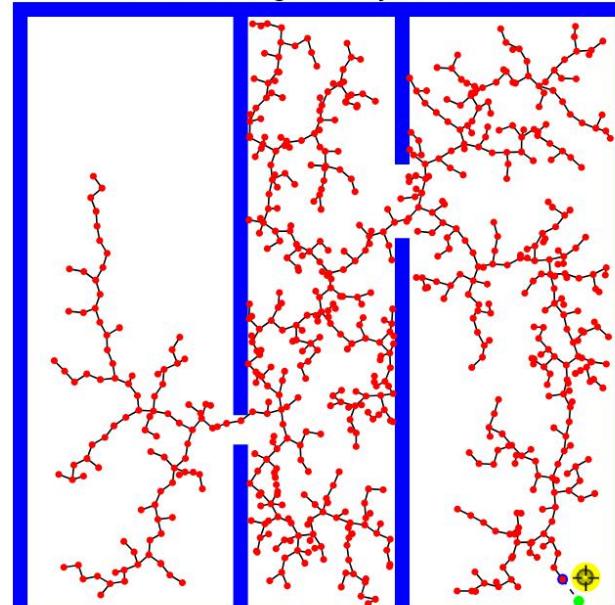
300 nodes, goal not yet reached



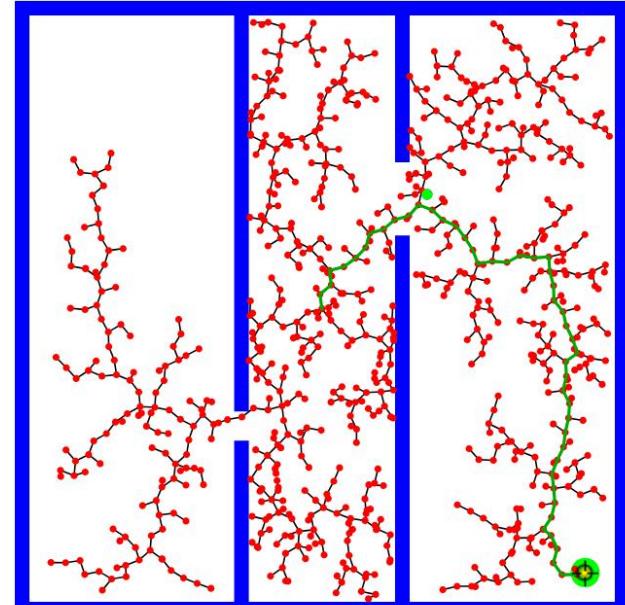
501 nodes, goal not yet reached



701 nodes, goal not yet reached

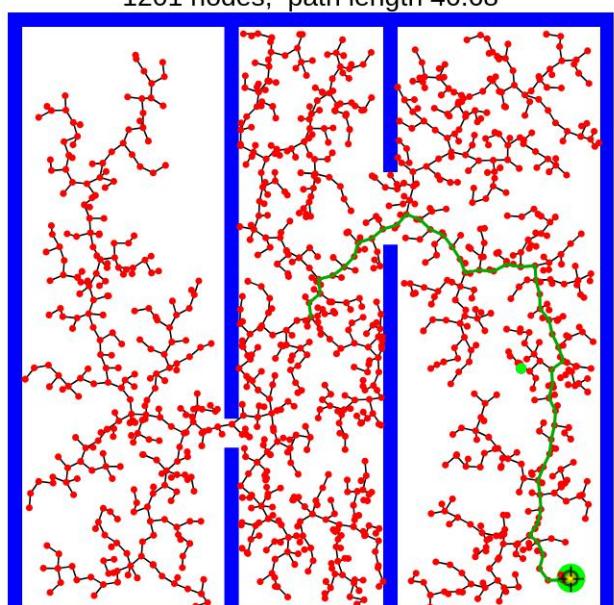


797 nodes, path length 46.68

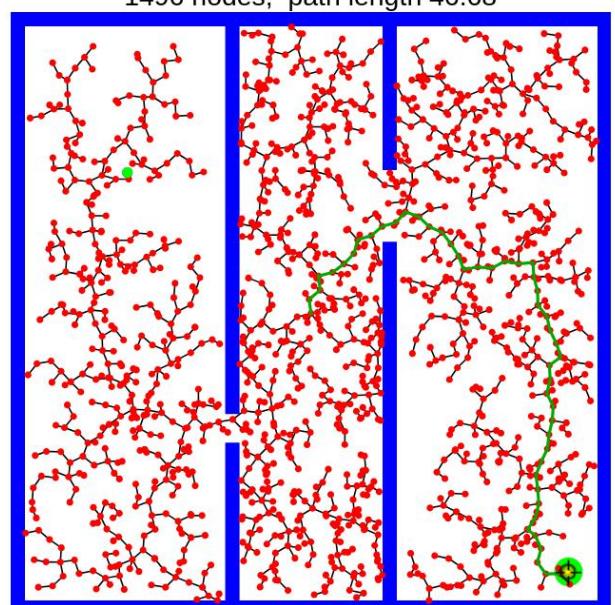


Goal
Reached

1201 nodes, path length 46.68



1496 nodes, path length 46.68



No further
improvement,
path length
unchanged



RRT*

- RRT algorithm is great at finding feasible trajectories quickly
- However, it is not suitable to find *good* trajectories
- Indeed, it has been proved that RRT converges to a sub-optimal solution almost surely, i.e.

$$\Pr[Y_\infty^{RRT} > c^*] = 1$$

- RRT «traps» itself by disallowing new better paths to emerge
- A substantial adaptation of RRT is required to ensure (asymptotic) optimality RRT*[1]

[1] S. Karaman and E. Frazzoli Optimal kinodynamic motion planning using incremental sampling-based methods
49th IEEE Conference on Decision and Control (CDC), pp. 7681-7687, 2010



RRT*

- Based on same principles of RRT
- Adds the possibility of «rewiring» parts of the tree when better paths are discovered
- After rewiring, the cost has to be propagated along the rewired subtree
- RRT* inherits the rapid exploration properties of RRT
- But differently from RRT, it is asymptotically optimal, i.e.

$$\Pr[Y_\infty^{RRT^*} = c^*] = 1$$



RRT*

RRT*

```

1:  $G.addVertex(q_{init})$ 
2: while  $q_{goal}$  not reached do
3:    $q_{rand} = \text{nextSample}()$ 
4:   if  $\text{CLEAR}(q_{rand})$  then
5:      $q_{nearest} = \text{NEAREST}(G, q_{rand})$ 
6:      $(q_{new}, path) = \text{EXTEND}(q_{nearest}, q_{rand})$ 
7:     if  $\neg\text{COLLISION}(path)$  then
8:        $Q_{near} = \text{NEAR}(G, q_{new}, \min(\gamma_{RRT*}(\log(|V|)/|V|)^{1/d}, \eta))$ 
9:        $G.addVertex(q_{new})$ 
10:       $q_{min} = q_{nearest}$ 
11:       $c_{min} = \text{COST}(q_{nearest}) + \text{COST}(path)$ 
12:       $p_{min} = path$ 
13:      for  $q_{near} \in Q_{near}$  do
14:         $p_{near} = PATH(q_{near}, q_{new})$ 
15:        if  $\neg\text{COLLISION}(p_{near})$  then
16:           $c_{near} = \text{COST}(q_{near}) + \text{COST}(p_{near})$ 
17:          if  $c_{near} < c_{min}$  then
18:             $q_{min} = q_{near}$ 
19:             $c_{min} = c_{near}$ 
20:             $p_{min} = p_{near}$ 
21:          end if
22:        end if
23:      end for
24:       $G.addEdge(q_{min}, q_{new}, p_{min})$ 
25:      for  $q_{near} \in Q_{near}$  do
26:         $p_{new} = PATH(q_{new}, q_{near})$ 
27:        if  $\neg\text{COLLISION}(p_{new})$  then
28:           $c_{new} = \text{COST}(q_{new}) + \text{COST}(p_{new})$ 
29:          if  $c_{new} < \text{COST}(q_{near})$  then
30:             $G.rewire(q_{near}, q_{new})$ 
31:          end if
32:        end if
33:      end for
34:    end if
35:  end if
36: end while

```



$$\begin{aligned}
& Q_{near} = \text{NEAR}(G, q_{new}, \min(\gamma_{RRT*}(\log(|V|)/|V|)^{1/d}, \eta)) \\
& G.addVertex(q_{new}) \\
& q_{min} = q_{nearest} \\
& c_{min} = \text{COST}(q_{nearest}) + \text{COST}(path) \\
& p_{min} = path \\
& \mathbf{for} \ q_{near} \in Q_{near} \ \mathbf{do} \\
& \quad p_{near} = PATH(q_{near}, q_{new}) \\
& \quad \mathbf{if} \ \neg\text{COLLISION}(p_{near}) \ \mathbf{then} \\
& \quad \quad c_{near} = \text{COST}(q_{near}) + \text{COST}(p_{near}) \\
& \quad \quad \mathbf{if} \ c_{near} < c_{min} \ \mathbf{then} \\
& \quad \quad \quad q_{min} = q_{near} \\
& \quad \quad \quad c_{min} = c_{near} \\
& \quad \quad \quad p_{min} = p_{near} \\
& \quad \quad \mathbf{end if} \\
& \quad \mathbf{end if} \\
& \mathbf{end for} \\
& G.addEdge(q_{min}, q_{new}, p_{min})
\end{aligned}$$

- **Parent selection:** the parent node is not necessarily $q_{nearest}$, but the one that can be reached with minimum cost
- After the new configuration q_{new} has been determined, its nearest neighbour node q_{min} is selected as parent



RRT* Rewiring

RRT*

```
1: G.addVertex( $q_{init}$ )
2: while  $q_{goal}$  not reached do
3:    $q_{rand} = \text{nextSample}()$ 
4:   if  $\text{CLEAR}(q_{rand})$  then
5:      $q_{nearest} = \text{NEAREST}(G, q_{rand})$ 
6:     ( $q_{new}, path$ ) =  $\text{EXTEND}(q_{nearest}, q_{rand})$ 
7:     if  $\neg\text{COLLISION}(path)$  then
8:        $Q_{near} = \text{NEAR}(G, q_{new}, \min(\gamma_{RRT^*}(\log(|V|)/|V|))$ 
9:       G.addVertex( $q_{new}$ )
10:       $q_{min} = q_{nearest}$ 
11:       $c_{min} = \text{COST}(q_{nearest}) + \text{COST}(path)$ 
12:       $p_{min} = path$ 
13:      for  $q_{near} \in Q_{near}$  do
14:         $p_{near} = \text{PATH}(q_{near}, q_{new})$ 
15:        if  $\neg\text{COLLISION}(p_{near})$  then
16:           $c_{near} = \text{COST}(q_{near}) + \text{COST}(p_{near})$ 
17:          if  $c_{near} < c_{min}$  then
18:             $q_{min} = q_{near}$ 
19:             $c_{min} = c_{near}$ 
20:             $p_{min} = p_{near}$ 
21:          end if
22:        end if
23:      end for
24:      G.addEdge( $q_{min}, q_{new}, p_{min}$ )
25:      for  $q_{near} \in Q_{near}$  do
26:         $p_{new} = \text{PATH}(q_{new}, q_{near})$ 
27:        if  $\neg\text{COLLISION}(p_{new})$  then
28:           $c_{new} = \text{COST}(q_{new}) + \text{COST}(p_{new})$ 
29:          if  $c_{new} < \text{COST}(q_{near})$  then
30:            G.rewire( $q_{near}, q_{new}$ )
31:          end if
32:        end if
33:      end for
34:    end if
35:  end if
36: end while
```

```
for  $q_{near} \in Q_{near}$  do
   $p_{new} = \text{PATH}(q_{new}, q_{near})$ 
  if  $\neg\text{COLLISION}(p_{new})$  then
     $c_{new} = \text{COST}(q_{new}) + \text{COST}(p_{new})$ 
    if  $c_{new} < \text{COST}(q_{near})$  then
      G.rewire( $q_{near}, q_{new}$ )
    end if
  end if
end for
```

- **Rewiring:** after a new node q_{new} has been inserted into the tree, the algorithm tries to perform some rewirings
- For all the nearest neighbour nodes q_{near} of q_{new} , a new path reaching q_{near} through q_{new} is generated
- If the cost of the new path is lower than the current cost of reaching q_{near} , a rewiring is applied and the new parent of q_{near} is set to q_{new} (and the subtree rooted at q_{near} is updated accordingly)

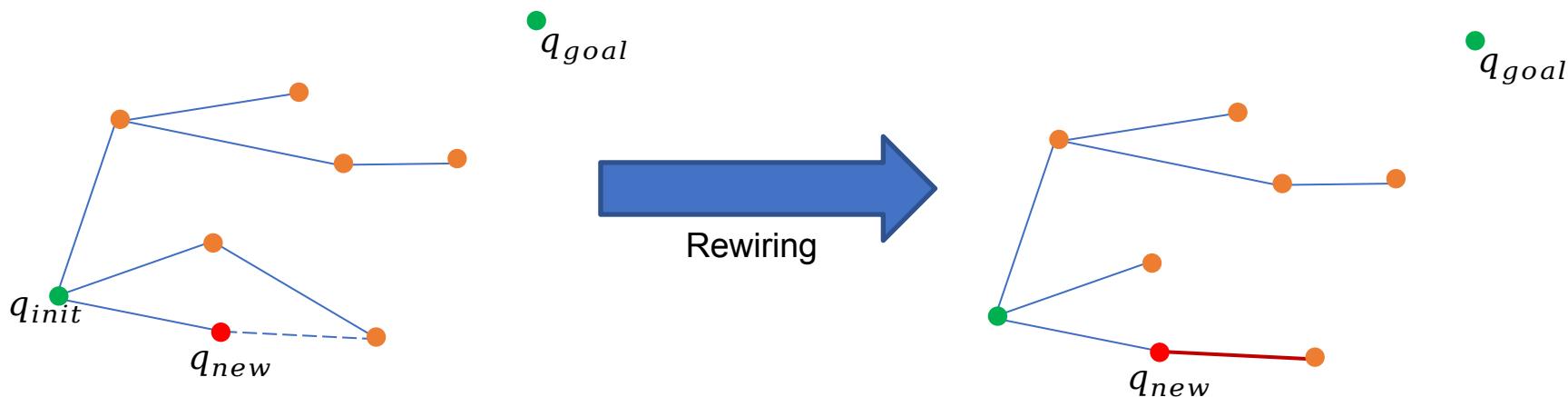
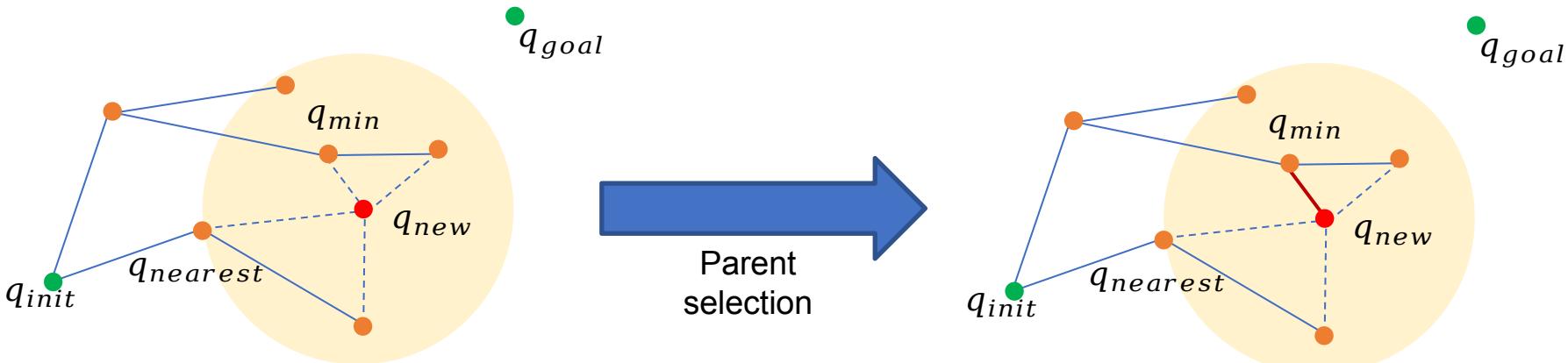


RRT* VS RRT

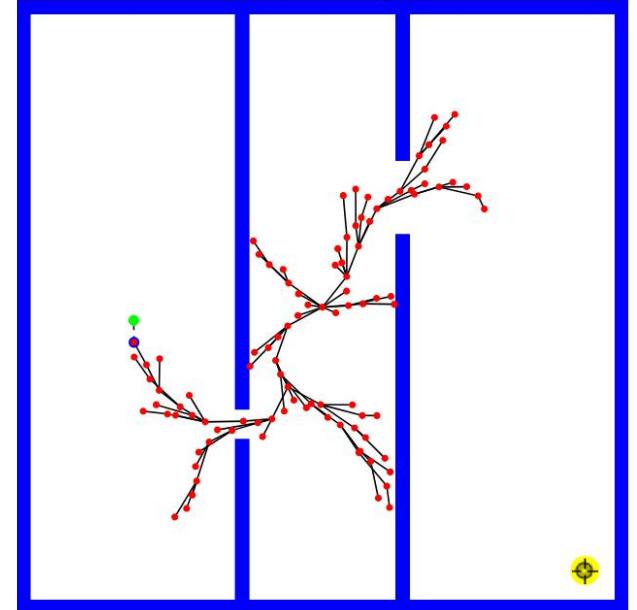
- Instead of adding q_{new} right away I add an edge to the node “closest” in terms of costs to q_{init}
- This creates a tree where most branches are pointing to the start node
- This results in paths with less sharp turns, with no need of straighten the path



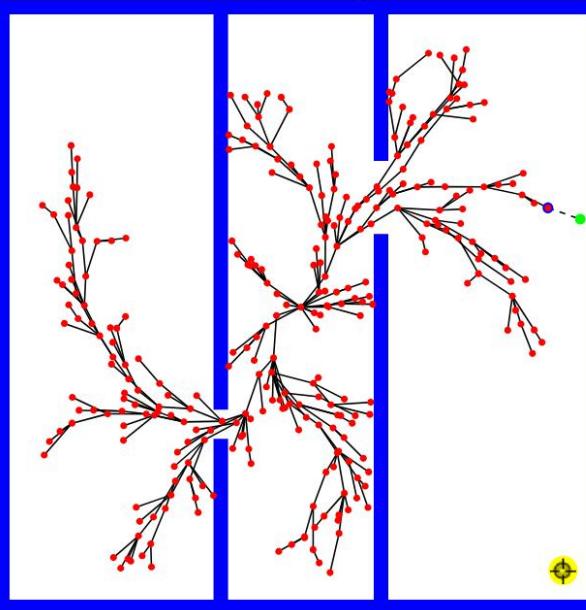
RRT*



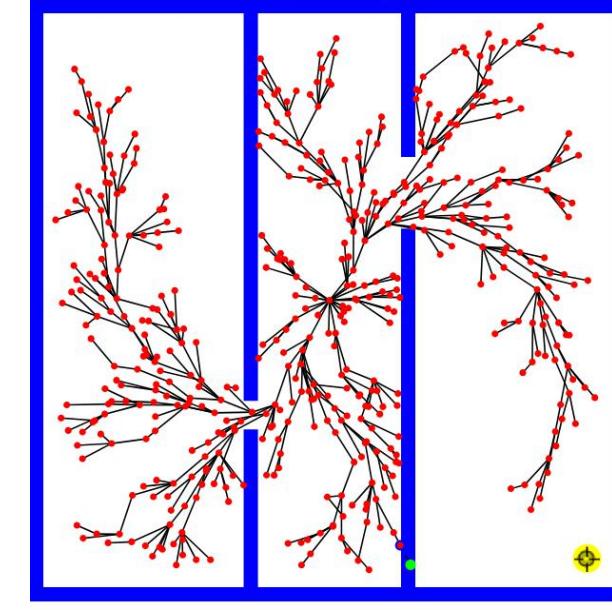
101 nodes, goal not yet reached



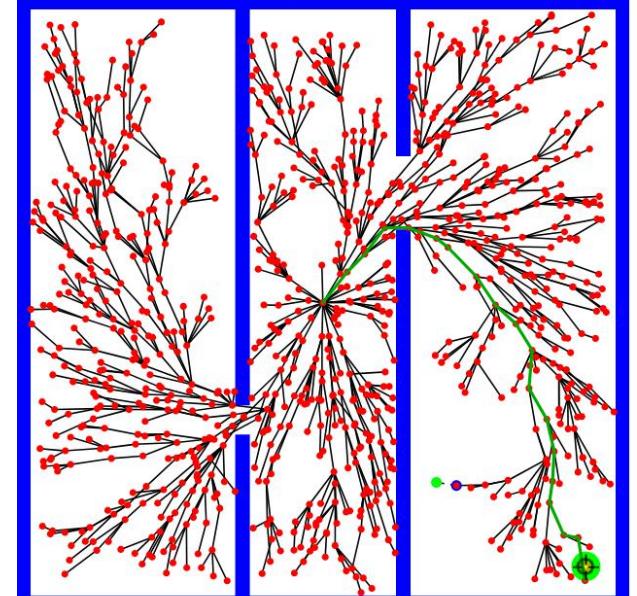
299 nodes, goal not yet reached



501 nodes, goal not yet reached

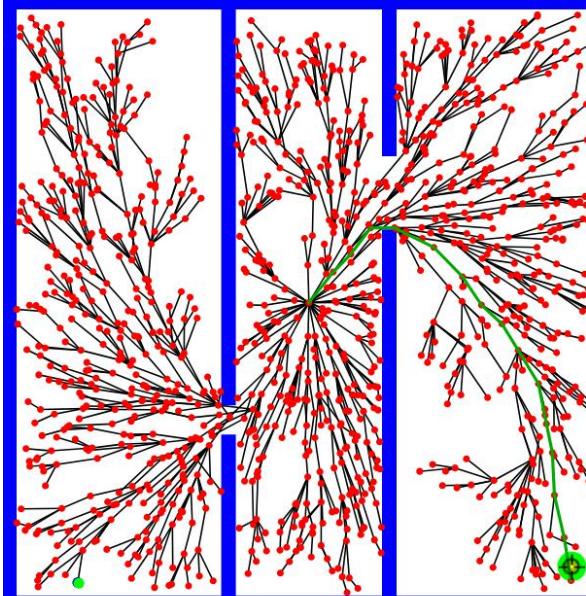


998 nodes, path length 37.66

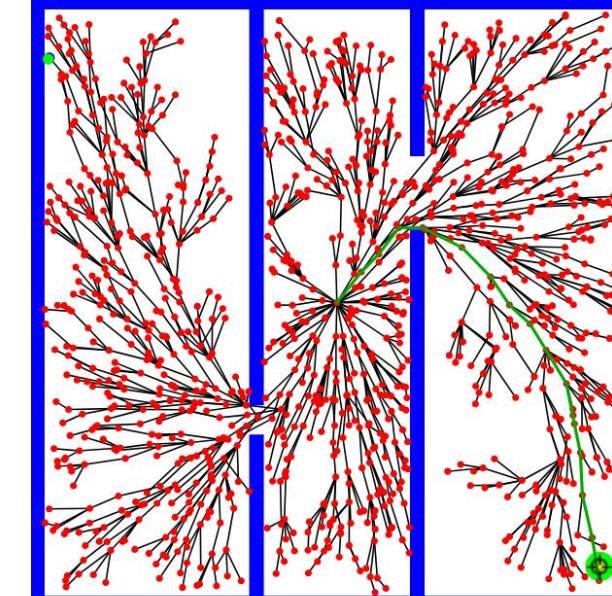


Goal
Reached

1199 nodes, path length 36.75



1201 nodes, path length 36.75

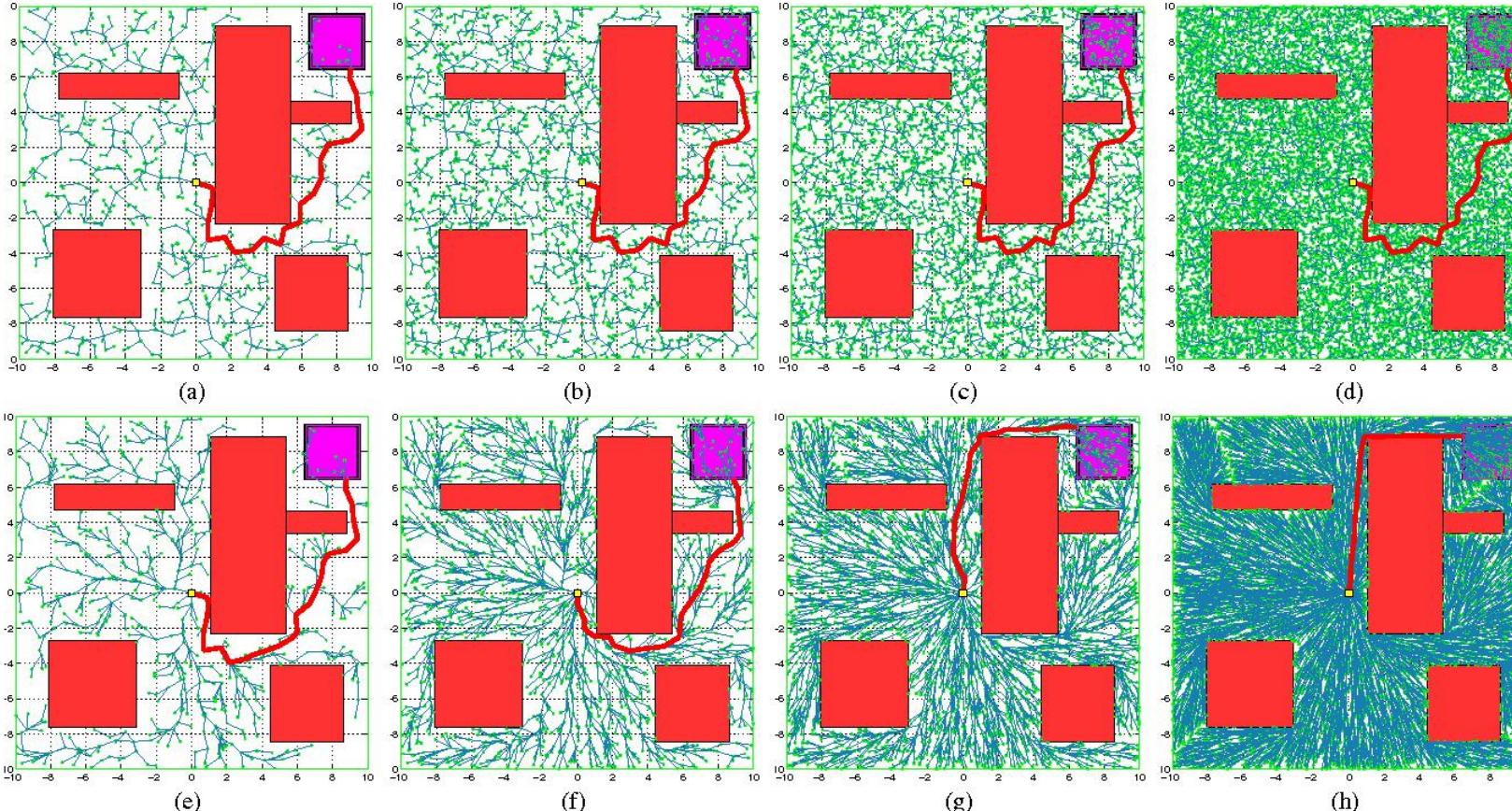


Refinements:
Adding new nodes
the path length
is shortening

Asymptotically
you get close to the
optimal **shortest** path



RRT vs RRT*



Trees grown by RRT (a-d)
and RRT* (e-h) after 1000,
2500, 5000, 15000
iterations

Source: Karaman, Sertac, and Emilio Frazzoli. "Incremental sampling-based algorithms for optimal motion planning." *Robotics Science and Systems VI* 104 (2010): 2.



Multiple Queries Model

Two phase computation:

1. Preprocessing phase: construct a roadmap G that can be used to quickly answer future queries (i.e. should be accessible from every part of C_{free})
2. Query phase:
 - a pair (q_{init}, q_{goal}) is given
 - both configurations are (easily) connected to G using a local planner
 - a discrete graph search is performed to find a sequence of edges from q_{init} to q_{goal} (e.g. using Dijkstra)



Probabilistic Roadmap: PRM

PRM

```
1: for  $i = 1..n$  do
2:   repeat
3:      $q = \text{nextSample}()$ 
4:   until  $\text{CLEAR}(q)$ 
5:    $G.\text{addVertex}(q)$ 
6: end for
7: for  $q \in V$  do
8:    $q_{near} = \text{KNEAR}(G, q, k)$ 
9:   for  $q' \in q_{near}$  do
10:    if  $q' \neq q \wedge \langle q, q' \rangle \notin E$  then
11:       $p = \text{PATH}(q, q')$ 
12:      if  $\neg \text{COLLISION}(p)$  then
13:         $G.\text{addEdge}(q, q', p)$ 
14:      end if
15:    end if
16:   end for
17: end for
```

- Construct the vertices of the roadmap by sampling n points from C_{free}
- For each vertex q , find its K-nearest-neighbours q_{near}
- Try to connect q to each vertex in $q' \in q_{near}$ (using the local planner):
 - if the resulting path is collision-free, add an edge from q to q' to the roadmap
- Parameters to tune (depending on specific problem, environment, ...): n , K and the parameters related to the local planner



Path Smoothing

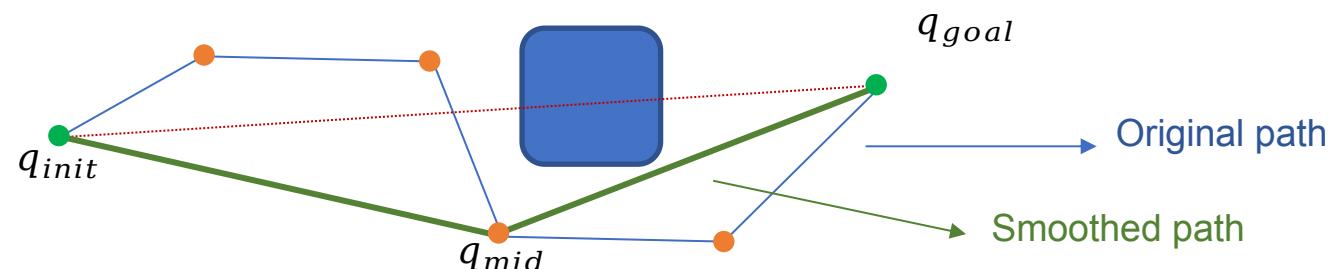
- The paths generated by motion planners (especially sampling based) tend to be jagged and longer than necessary
- In practice, usually to the output of the motion planner is applied some post-processing refinement to improve its quality and smoothness
- ***Shortcutting:*** iteratively sample pairs of random configurations along the current path (chosen randomly or with some appropriate heuristic), and try to connect them directly to «shortcut» the original path



Path Smoothing

Shortcutting:

- A possible approach is based on the hierarchical binary partitioning of the path and application of the shortcuttering to each subpath:
 - Try to connect directly q_{init} with q_{goal}
 - If the resulting path is collision free, we are done
 - Otherwise, split the path in two parts of equal length, and apply recursively the procedure to the subpath q_{init} - q_{mid} and q_{mid} - q_{goal}





Combinatorial vs Sampling Based

- *Combinatorial Motion Planning:*

1. Elegant and complete solutions (find a solution if it exists, otherwise report failure)
2. Intractable when C -space dimensionality grows
3. Difficult to implement (in general scenarios)



Combinatorial vs Sampling Based

- *Sampling Based Motion Planning:*

- Only probabilistic guarantees (if a solution exists, it will be found with probability 1, but if a solution does not exist the algorithm may run forever)
- More efficient in most practical problems w.r.t. combinatorial algorithms
- Probabilistic asymptotic optimality (using RRT*) at the price of higher overhead
- Performance degradation in the presence of narrow passages (e.g. bug-trap)
- Possibility to handle non-holonomic, complex constraints
- Simpler to implement (in general scenarios)

Robot Control

Robot planning

University of Trento

TEACHER: Michele Focchi





Outline

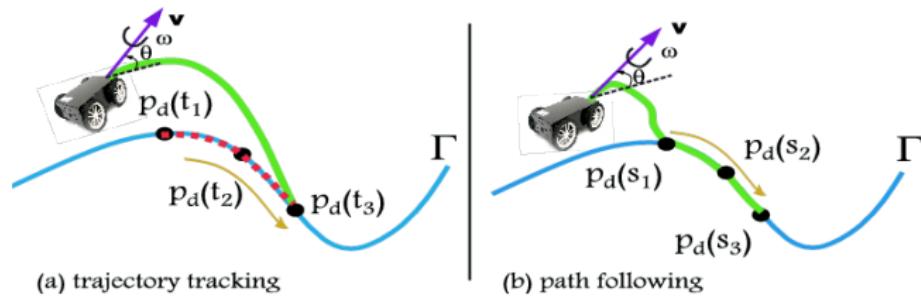
Guiding the Robot

Guiding a mobile robot

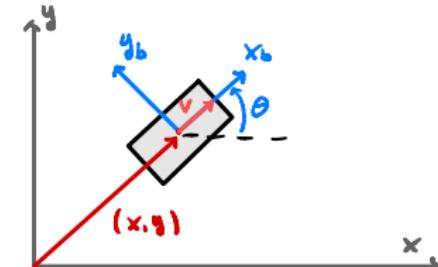
- ▶ We have seen how to decide an optimal sequence of actions to drive the robot between two configurations
- ▶ We have a problem: we are living a real life in a real world.
 - ▶ We do not know exactly where the robot is (localisation error)
 - ▶ The robot has a dynamics and does not respond instantaneously to our commands.
 - ▶ Sensors are affected by noise and actuators are not 100% accurate (I could ask for 0.3rad/s and get 0.28 rad/s or 0.32rad/s)
- ▶ The robot deviates from the "ideal" trajectory and the error accumulates over time.
- ▶ After some time, it could be far away from where I expected it to be

Control algorithms

- ▶ In the realm of mobile robotics we distinguish between trajectory tracking (tracking an ideal robot while it moves along the trajectory) and path tracking (following a geometric path)
- ▶ We will concentrate on the first type of control problem



Problem Formulation



Given a *reference* robot that moves according to the model:

$$\dot{\hat{x}} = \begin{bmatrix} \dot{\hat{x}} \\ \dot{\hat{y}} \\ \dot{\hat{\theta}} \end{bmatrix} = \begin{bmatrix} \hat{v} \cos(\hat{\theta}(t)) \\ \hat{v} \sin(\hat{\theta}(t)) \\ \hat{\omega} \end{bmatrix}$$

with known $\hat{x}(0), \hat{v}(t), \hat{\omega}(t)$; given an actual robot with state evolution:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v \cos(\theta(t)) \\ v \sin(\theta(t)) \\ \omega \end{bmatrix}$$

Find a control law $\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = k(x(t), \hat{x}(t), \hat{v}(t), \hat{\omega}(t))$

such that the state $x(t)$ converges to the state $\hat{x}(t)$

Lyapunov Stability

The formulation above is quite imprecise. There is a much better defined notion called Lyapunov stability. This notion is usually referred to equilibrium points but it easily generalises to trajectories.

Lyapunov Stability - An informal definition

An equilibrium is:

- ▶ locally stable if starting from a small enough neighbourhood of the equilibrium (i.e., with a small perturbation), the state of the system will remain close to the equilibrium,
- ▶ locally asymptotically stable if it is stable and eventually it converges to the equilibrium,
- ▶ unstable if even a small perturbation will move the state far away from the equilibrium
- ▶ globally asymptotically stable if the state converges to the equilibrium starting from any initial point.

Lyapunov Stability

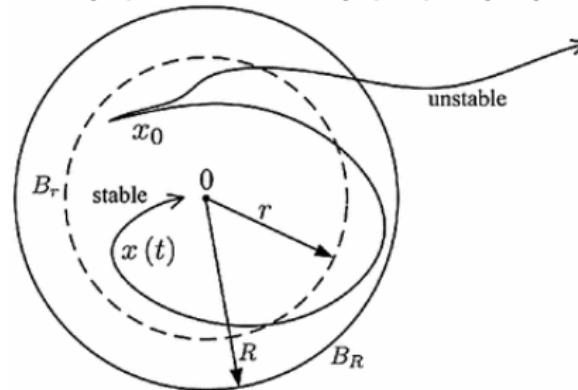
We can be a little more precise (boring but useful 😊) Let
 $\mathbb{B}_\rho(\mathbf{x}_0) = \{\mathbf{x} \in \mathbb{R}^n \text{ s.t. } \|\mathbf{x} - \mathbf{x}_0\| \leq \rho\}$

Let $\dot{\mathbf{x}} = f(\mathbf{x})$ be a dynamical system. We will use the following definitions:

- ▶ \mathbf{x}_{eq} is an equilibrium point if $f(\mathbf{x}_{\text{eq}}) = 0$.
- ▶ The equilibrium is *locally* stable if for all $R > 0$ there exists $r > 0$ if $\mathbf{x}(0) \in \mathbb{B}_r(\mathbf{x}_{\text{eq}})$ then $\mathbf{x}(t) \in \mathbb{B}_R(\mathbf{x}_{\text{eq}})$ for all t .
- ▶ The equilibrium is *asymptotically locally* stable if it is locally stable and if $\mathbf{x}(0) \in \mathbb{B}_r(\mathbf{x}_{\text{eq}}) \rightarrow \lim_{t \rightarrow \infty} \|\mathbf{x}(t) - \mathbf{x}_{\text{eq}}\| = 0$
- ▶ The equilibrium is globally asymptotically stable (GAS) if $\forall \mathbf{x}(0) \in \mathbb{R}^n : \lim_{t \rightarrow \infty} \|\mathbf{x}(t) - \mathbf{x}_{\text{eq}}\| = 0$

Lyapunov Stability

We can represent the Lyapunov Stability property by the following plot:



With this definition in mind our control problem can be rephrased as:

Find a control law $\begin{bmatrix} \mathbf{v}(t) \\ \omega(t) \end{bmatrix} = k(\mathbf{x}(t), \hat{\mathbf{x}}(t), \hat{u}(t), \hat{\omega}(t))$

s.t. $\hat{\mathbf{x}}(t)$ is a globally asymptotically stable trajectory for the system.

Some definitions

In order to understand how to design a controller for such a system, we need a few more technical details. A function $V(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$

- ▶ is positive (negative) definite if
 $V(\mathbf{x}) > 0 (V(\mathbf{x}) \leq 0) \forall \mathbf{x} \in \mathbb{R}^n \setminus \{0\}$ and $V(\mathbf{x}) = 0$ iff $\mathbf{x} = 0$.
- ▶ is positive (negative) semi-definite if
 $V(\mathbf{x}) \geq 0 (V(\mathbf{x}) \leq 0) \forall \mathbf{x} \in \mathbb{R}^n \setminus \{0\}$ and $V(\mathbf{x}) = 0$ iff $\mathbf{x} = 0$.
- ▶ is radially unbounded if $\|\mathbf{x}\| \rightarrow \infty \implies V(\mathbf{x}) \rightarrow \infty$
- ▶ has sub-level sets defined by $\{\mathbf{x} \in \mathbb{R}^n \text{ s.t. } V(\mathbf{x}) \leq \alpha\}$ for $\alpha \in \mathbb{R}$.

Lyapunov Theorem

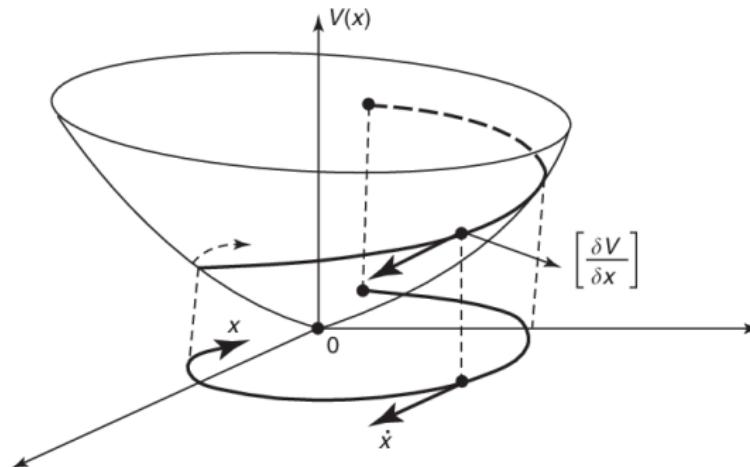
Lyapuov Theorem

Let \mathbf{x}_e be an equilibrium point. Let $V : \mathbb{R}^n \rightarrow \mathbb{R}$ be a positive definite and continuously differentiable function in a neighbourhood of \mathbf{x}_e .

- ▶ if $\dot{V} : \mathbb{R}^n \rightarrow \mathbb{R}$ is negative semi-definite then the equilibrium is locally stable.
- ▶ if $\dot{V} : \mathbb{R}^n \rightarrow \mathbb{R}$ is negative definite then the equilibrium is locally asymptotically stable.

This theorem is easy to generalise to a trajectory $\hat{\mathbf{x}}$. If \mathbf{x} is the state, we can consider a new system with state given by the difference $e(t) = \hat{\mathbf{x}} - \mathbf{x}$ which has an equilibrium in 0.

Lyapunov Theorem - the intuition



Since $\dot{V} = \nabla_x V \cdot \dot{x}$ we are actually requiring that the derivative vector \dot{x} has a negative component along the gradient $\nabla_x V$. We can think of $V(x)$ as a generalisation of an energy function. In essence, we are saying that the system dissipates energy everywhere except for the equilibrium point.

Lasalle's theorem

We can generalise the theorem above as follows.

Lasalle's theorem

Let x_e be an equilibrium point. Let $V : \mathbb{R}^n \rightarrow \mathbb{R}$ be a positive definite and continuously differentiable function $\forall x \in \mathbb{R}^n$. Assume that $V(x)$ is radially unbounded and that its sub-level sets are bounded. Then the equilibrium is **globally** asymptotically stable if $\dot{V}(x)$ is negative semi-definite and the only solution $x(t)$ such that $\dot{x} = f(x)$ with $\dot{V}(x) = 0$ is $x(t) = x_e$ for all t .

Lyapunov-based control design

We can use Lyapunov functions to construct feedback control functions that guarantee “by construction” the convergence to a desired trajectory.
Let us go back to our problem:

The control problem

Given a *reference* robot that moves according to the model:

$$\dot{\hat{x}} = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{\theta} \end{bmatrix} = \begin{bmatrix} \hat{v} \cos(\hat{\theta}t) \\ \hat{v} \sin(\hat{\theta}t) \\ \hat{\omega} \end{bmatrix}$$

with known $\hat{x}(0)$, $\hat{v}(t)$, $\hat{\omega}(t)$; given an actual robot with state evolution:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v \cos(\theta t) \\ v \sin(\theta t) \\ \omega \end{bmatrix}$$

Find a control law $\begin{bmatrix} v(t) \\ \omega(t) \end{bmatrix} = k(x(t), \hat{x}(t), \hat{v}(t), \hat{\omega}(t))$

such that the state $x(t)$ converges to the state $\hat{x}(t)$

Lyapunov-based control design

Let $\mathbf{e} = \mathbf{x} - \hat{\mathbf{x}}$ be the error state.

The function

$$\begin{aligned}V &= \frac{1}{2} \|\mathbf{e}\|^2 = \\&= \frac{1}{2} (x - \hat{x})^2 + \frac{1}{2} (y - \hat{y})^2 + \frac{1}{2} (\theta - \hat{\theta})^2 \\&= \frac{1}{2} e_x^2 + \frac{1}{2} e_y^2 + \frac{1}{2} e_\theta^2\end{aligned}$$

is positive definite, is globally continuous and differentiable, is radially unbounded and has bounded sub-level sets. So it could be a perfect fit if we found a control function \mathbf{u} such that \dot{V} is negative definite.

Lyapunov-based control design

The derivative of the Lyapunov function is given by:

$$\begin{aligned}\dot{V}(e(t)) &= e_x \dot{e}_x + e_y \dot{e}_y + e_\theta \dot{e}_\theta \\ &= e_x(v \cos(\theta) - \hat{v} \cos(\hat{\theta})) + e_y(v \sin(\theta) - \hat{v} \sin(\hat{\theta})) + e_\theta(\omega - \hat{\omega}).\end{aligned}$$

By choosing $v = \hat{v} + \delta_v$ and $\omega = \hat{\omega} + \delta_\omega$, we can write:

$$\begin{aligned}\dot{V} &= \hat{v} \left(e_x (\cos(\theta) - \cos(\hat{\theta})) + e_y (\sin(\theta) - \sin(\hat{\theta})) \right) + \\ &\quad + \delta_v (e_x \cos(\theta) + e_y \sin(\theta)) + \\ &\quad + e_\theta \delta_\omega.\end{aligned}$$

Let $e_{xy} = \sqrt{e_x^2 + e_y^2}$, and $\psi = \text{atan2}(e_y, e_x)$. Clearly $e_x = e_{xy} \cos(\psi)$ and $e_y = e_{xy} \sin(\psi)$.

Lyapunov-based control design

Let $\alpha = \theta + \hat{\theta}$. By applying some simple trigonometric formulas:

$$\cos(\theta) - \cos(\hat{\theta}) = -2 \sin\left(\frac{e_\theta}{2}\right) \sin\left(\frac{\alpha}{2}\right)$$
$$\sin(\theta) - \sin(\hat{\theta}) = 2 \sin\left(\frac{e_\theta}{2}\right) \cos\left(\frac{\alpha}{2}\right)$$

which leads to

$$\left(e_x (\cos(\theta) - \cos(\hat{\theta})) + e_y (\sin(\theta) - \sin(\hat{\theta})) \right) =$$
$$2e_{xy} \sin\left(\frac{e_\theta}{2}\right) \left(-\sin\left(\frac{\alpha}{2}\right) \cos \psi + \cos\left(\frac{\alpha}{2}\right) \sin(\psi) \right) =$$
$$2e_{xy} \sin\left(\frac{e_\theta}{2}\right) \sin\left(\psi - \frac{\alpha}{2}\right).$$

and

$$(e_x \cos(\theta) + e_y \sin(\theta)) = e_{xy} (\cos(\psi) \cos(\theta) + \sin(\psi) \sin(\theta)) =$$
$$= e_{xy} \cos(\theta - \psi).$$

Lyapunov-based control design

Overall we can write:

$$\begin{aligned}\dot{V}(\mathbf{e}) &= \hat{v} \left(e_x (\cos(\theta) - \cos(\hat{\theta})) + e_y (\sin(\theta) - \sin(\hat{\theta})) \right) + \\ &\quad + \delta_v (e_x \cos(\theta) + e_y \sin(\theta)) + \\ &\quad + e_\theta \delta_\omega = \\ &= 2\hat{v} e_{xy} \sin\left(\frac{e_\theta}{2}\right) \sin\left(\psi - \frac{\alpha}{2}\right) + \delta_v e_{xy} \cos(\theta - \psi) + e_\theta \delta_\omega\end{aligned}$$

Now, if we choose

$$\begin{aligned}\delta_v &= -k_p e_{xy} \cos(\theta - \psi) \\ \delta_\omega &= -\frac{2}{e_\theta} \hat{v} e_{xy} \sin\left(\frac{e_\theta}{2}\right) \sin\left(\psi - \frac{\alpha}{2}\right) - k_\theta e_\theta =\end{aligned}$$

$$-\hat{v} \text{sinc}\left(\frac{e_\theta}{2}\right) \sin\left(\psi - \frac{\alpha}{2}\right) - k_\theta e_\theta$$

$$\text{Where } \text{sinc}(x) = \frac{\sin(x)}{x}.$$

Therefore, we have

$$\dot{V}(\mathbf{e}) = -k_p e_{xy}^2 \cos(\theta - \psi)^2 - k_\theta e_\theta^2$$

Lyapunov-based control design

- ▶ The function $-k_p e_{xy}^2 \cos(\theta - \psi)^2 - k_\theta e_\theta^2$ is positive semi-definite.
- ▶ It becomes 0 when $e_{xy} = 0, e_\theta = 0$ (which is the equilibrium) or when $\theta = \psi \pm \frac{\pi}{2}$, which means that the orientation is orthogonal to the trajectory error (i.e., the robot moves parallel to the virtual robot).
- ▶ When $\dot{V} = 0$, we have $e_\theta = 0$ ($\theta = \hat{\theta}$) and

$$\begin{aligned}\delta_v &= -k_p e_{xy} \cos(\theta - \psi) = 0 \\ \delta_\omega &= -e_{xy} \hat{v} \operatorname{sinc}\left(\frac{e_\theta}{2}\right) \sin\left(\psi - \frac{\alpha}{2}\right) - k_\theta e_\theta \\ &= -e_{xy} \hat{v} \sin\left(\psi - \frac{\alpha}{2}\right) = \\ &= -e_{xy} \hat{v} \sin\left(\theta \pm \frac{\pi}{2} - \frac{\theta + \hat{\theta}}{2}\right) \\ &= -e_{xy} \hat{v}.\end{aligned}$$

- ▶ Hence the only case in which we reach equilibrium ($\delta_v = 0, \delta_\omega = 0$) is when $e_{xy} = 0$ and $e_\theta = 0$.

Controller Exercise

- ▶ Implement the controller and test in the loco_nav simulation framework or in Matlab.
- ▶ Set initial error (desired initial state different than the initial state) and appreciate the evolution of the controller. What happens if I change the values of k_p and k_θ ?
- ▶ What happens if you set higher desired speeds \hat{v} , $\hat{\omega}$?

The orienteering problem

Robot Planning and its Applications

University of Trento

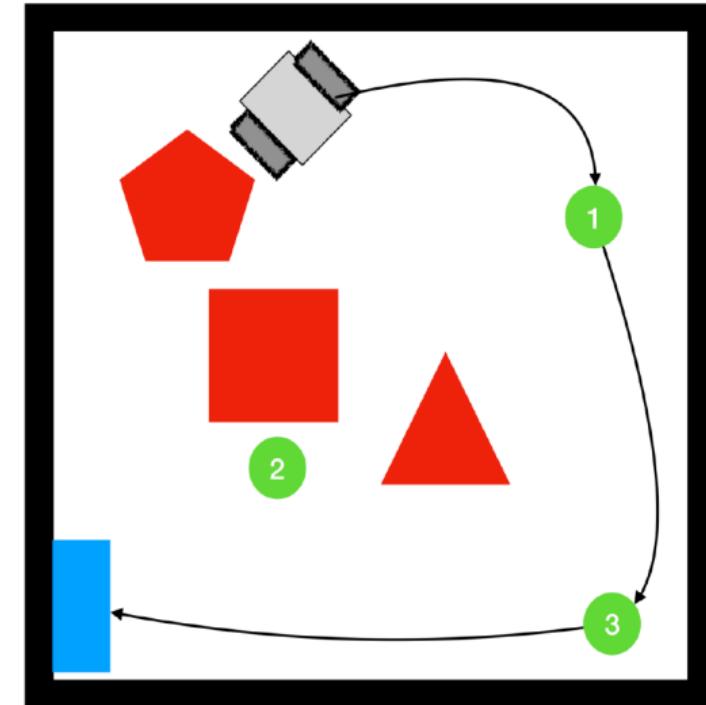
Luigi Palopoli (luigi.palopoli@unitn.it)



Project 1

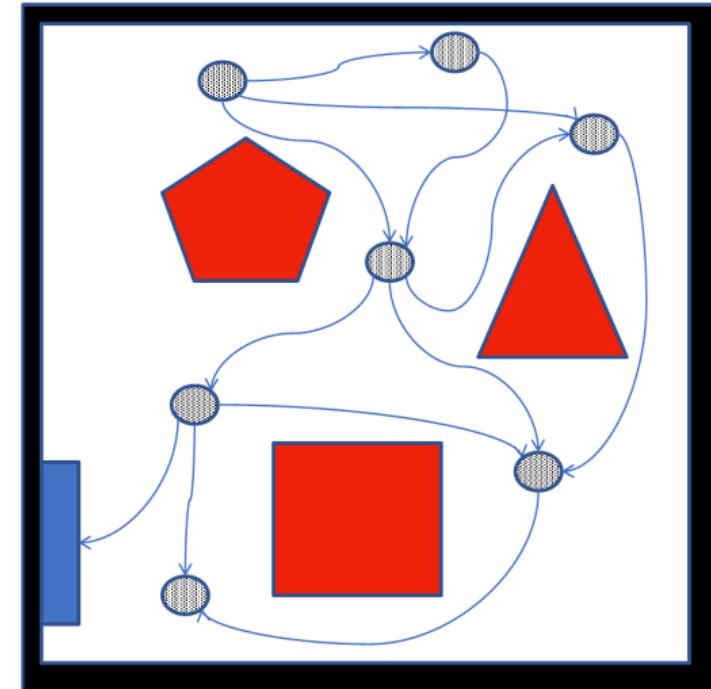
Find a route that moves the robot from its initial position to the gate in the minimum time and collecting as many "victims" as possible.

- ▶ The problem is multi-objective.
- ▶ We aim to minimise time, but at the same time we want to visit as many location as possible.
- ▶ We can model this thing, assuming that on each target location visited we subtract some time to the total.



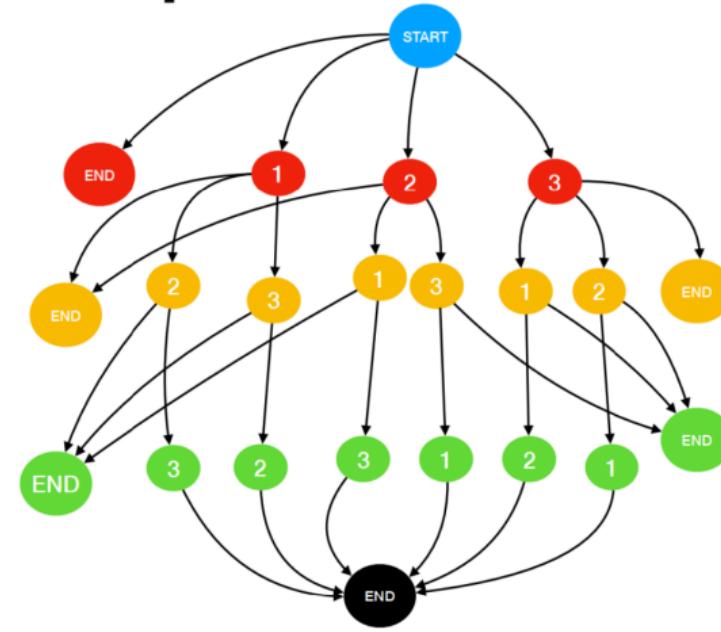
Roadmap discretisation

- ▶ We have seen that possible strategy is to first create a roadmap discretising the problem.
- ▶ Then we have a classical path optimisation problem on a graph.
- ▶ It is not the only way (see "Dubins Orienteering Problem" by Penicka et al., "Traveling Salesperson problems for the Dubins Vehicle" by Savla et al.)



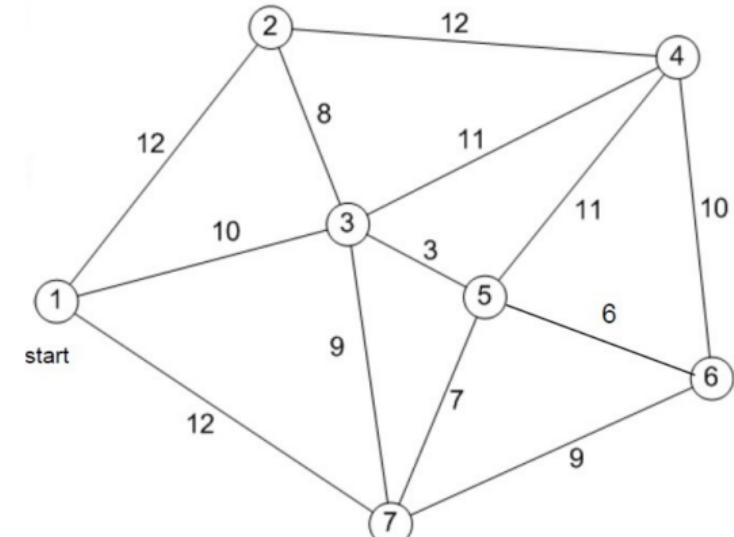
Brute force solution

- ▶ After discretising the problem, an easy solution (but hardly a scalable one) can be to make a brute force search over all possible alternatives.
- ▶ Luckily there are better alternatives



A simpler problem: the travelling Salesperson Problem

- ▶ It's one of the most famous combinatorial optimization problems — NP-hard, with applications in logistics, robotics, chip design, DNA sequencing, and more.
- ▶ Starting from city 1 the TSP has to touch all the cities before returning to base.
- ▶ We have a distance associated with each link (for simplicity the same in both directions)
- ▶ The TSP wants to minimise the total travel time.



A first easy formulation:

- ▶ Let $d_{i,j} \geq 0$ be a parameter encoding the distance (or cost) of travelling from node i to node j . Typically $d_{i,i} = 0$ and $d_{i,j} = d_{j,i}$, if symmetric.
- ▶ The cities form a complete weighted graph $G=(V,E)$
- ▶ Introduce a decision variable $x_{i,j}$:

$$x_{i,j} = \begin{cases} 1 & \text{if the optimal path goes from node } i \text{ to node } j \\ 0 & \text{Otherwise} \end{cases}$$

The variable $x_{i,j}$ is defined only for the pairs i, j for which there is an edge.

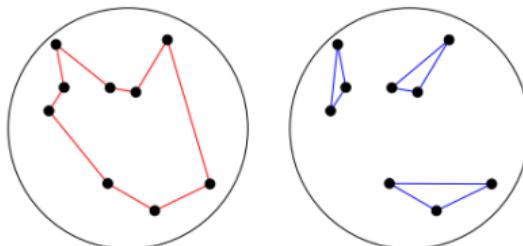
- ▶ The total distance (or cost) of a path will be given by: $\sum_{i=1}^N \sum_{j=1, j \neq i}^N d_{i,j} x_{i,j}$
This sums the distances of all the selected edges.
- ▶ We have to have one incoming arc and one outgoing arc for each node. How?

$$\sum_{i=1, i \neq j}^N x_{i,j} = 1, \quad \forall j = 1, \dots, N$$

$$\sum_{i=1}^N x_{i,j} = 1, \quad \forall i = 1, \dots, N$$

Subtour elimination constraint

- ▶ The formulation above allows us to find a solution that touches all cities.
- ▶ However, the problem can produce disconnected subtours
- ▶ We can solve the problem by introducing an additional constraint:



$$\sum_{i \in S} \sum_{j \in S} x_{i,j} \leq |S| - 1,$$

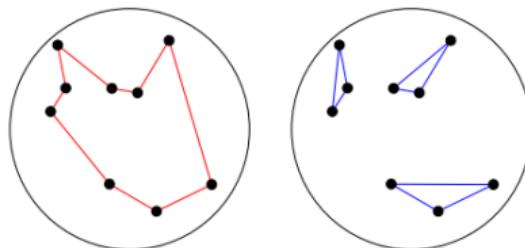
$$S \subset V,$$

$$2 \leq |S| \leq N - 1$$

where S are the subsets of the vertices V

The problem of subtours: MTZ constraint formulation

- ▶ An alternative strategy can be to introduce additional variable u_i , one for each node, such that $u_i > u_j$ if i is visited after j .
- ▶ We can impose this by the constraints:



$$u_i - u_j + Nx_{i,j} \leq N - 1, \quad \forall i \neq j \quad i, j \in \{2, \dots, N\}$$
$$u_1 = 1 \quad (\text{fix reference})$$
$$2 \leq u_i \leq N \quad \forall i = 2, \dots, N \quad (\text{order bounds})$$

- ▶ When $x_{i,j} = 0$ the constraint above simply reduces to $u_i - u_j \leq N - 1$ which is obviously true. when $x_{i,j} = 1$ the above becomes $u_j \geq u_i + 1$
- ▶ imposing progressive ordering prevents sub-tours, because if a smaller cycle existed, the ordering would become inconsistent (it would require a city to have both higher and lower order numbers simultaneously).

The complete TSP formulation

$$\min \sum_{i=1}^N \sum_{j=1, j \neq i}^N d_{i,j} x_{i,j}$$

$$x_{i,j} \in \{0, 1\} \quad i, j = 1 \dots N$$

$$\sum_{i=1, i \neq j}^m x_{i,j} = 1, \quad j = 1, \dots, N$$

$$\sum_{j=1, i \neq j}^m x_{i,j} = 1, \quad i = 1, \dots, N$$

$$u_1 = 1$$

$$2 \leq u_i \leq N \quad \forall i = 2, \dots, N$$

$$u_i - u_j + Nx_{i,j} \leq N - 1, \quad \forall i \neq j \quad i, j \in \{2, \dots, N\}$$

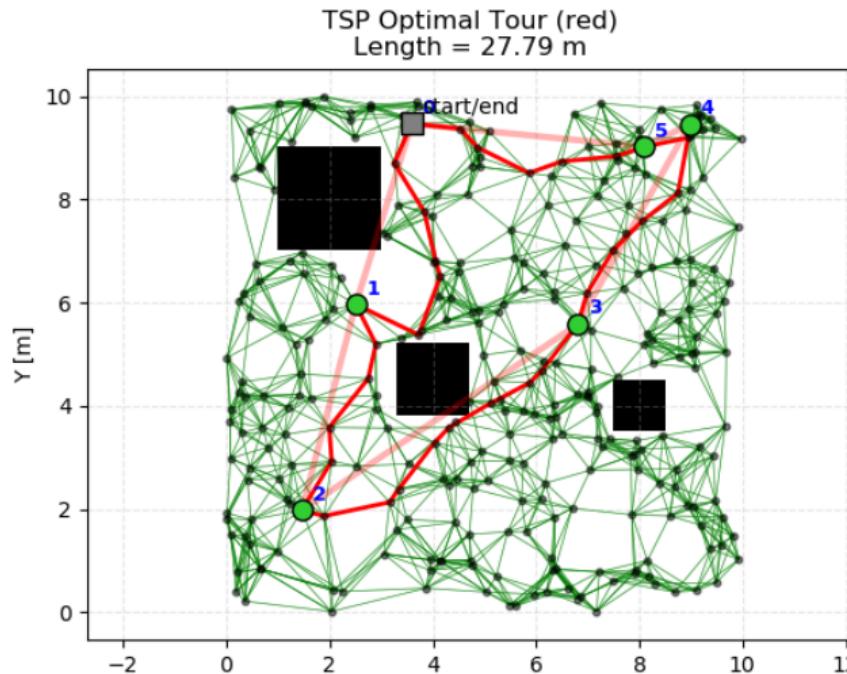
We solve the MILP problem, to reconstruct the optimal path, we start from the start node and iteratively follow the successor arc checking which $x_{i,j} = 1$.

Complexity

- ▶ Number of constraints: $O(N^2)$
- ▶ Number of variables: $O(N^2)$ binary + $O(N)$ continuous
- ▶ Still NP-hard, but practical for small/medium N with MILP solvers.

Results

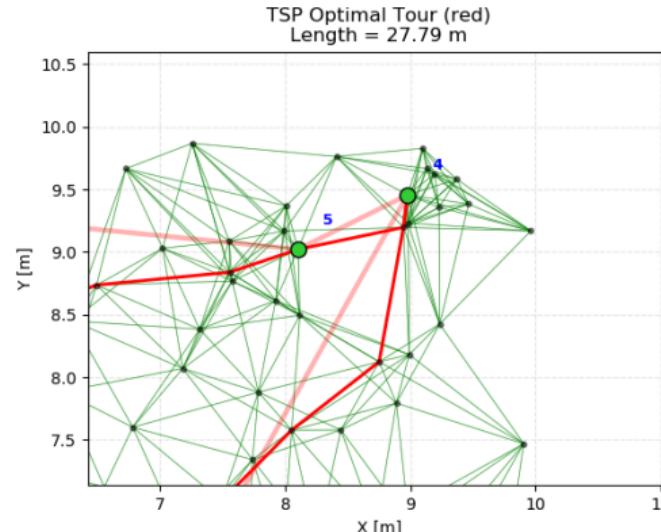
- ▶ cities: (4, 9), (8, 9),(1.5, 2.2) , (6.7, 5.4), (3, 6.34),(9.1, 9.5)
- ▶ PRM: 300 samples (N), nearest neighbors (k): 10



Results

- ▶ If any city has no finite shortest-path distance to at least one other city, your $\text{in}/\text{out-degree} == 1$ constraints make the model infeasible.
- ▶ Before building the MILP problem, verify that for every city there's at least one other city with finite distance (outgoing and incoming).
- ▶ either increase PRM density/connectivity
- ▶ Still the actual formulation could allow some edges of the PRM to be used twice...

Results



- ▶ MTZ constraints work on city-level arcs $x_{i,j}$ but not on the PRM local topology.
- ▶ Instead of solving TSP over cities, you can solve it directly over the PRM nodes
- ▶ Heavier computationally (many more variables) but conceptually correct:

The Orienteering Problem

- ▶ The Orienteering Problem is a generalisation of the TSP that has received a lot of attention in the past few decades due to its relevance in various routing and resource allocation applications.
- ▶ The term “Orienteering Problem (OP)” was first introduced by Golden, Levy, and Vohra (1987).
- ▶ Differently from TSP we have a limited budget of time (or cost or distance), not all available nodes can be visited due to the limited time budget
- ▶ We want to visit as many customer as possible, each customer's visit produces a different value s_i
- ▶ In the Orienteering Problem (OP), a set of vertices is given, each associated with a score or profit. The objective is to determine a path (or cycle) of limited length that visits a subset of these vertices so as to maximize the total collected score and the given time budget is not exceeded
- ▶ It is a combination of node selection and determining the shortest path between the selected nodes.
- ▶ can be seen as a combination between two classical combinatorial problems, the Knapsack Problem and the TSP

OP VS TSP

- ▶ we have two opposite criteria: one that motivates the salesperson/rescuer to travel and another that imposes a constraint **in the route length (time)**
- ▶ our example: a rescue robot needs to rescue victims whose value depends on their survival chance
- ▶ Determining the shortest path between the selected vertices will be helpful to visit as many vertices as possible **in the available time**
- ▶ Can you try to model it yourself?

Preliminaries

- ▶ Let N be the number of nodes (victims). The OP can also be defined with the aid of a graph $G=(V,E)$ where $V = v_1, \dots, v_N$ is the vertex set and E is the arc set.
- ▶ In this definition the nonnegative score s_i is associated with each vertex $v_i \in V$ and the travel time/cost d_{ij} to travel from vertex i to j is known for all vertices and is associated with each arc.
- ▶ In most cases, the OP is defined as a path to be found between distinct vertices, it is always possible to add a dummy arc between end and start vertex to turn a path problem into a circuit problem (not your case).
- ▶ The starting point (vertex 1) and the end point (gate) (vertex N) are fixed.
- ▶ Available time budget d_{max}

The Orienteering Problem

- Once again, the OP can be formulated as an integer programming model with the following decision (binary) variables: $x_{i,j} = 1$ if we travel from customer i to customer j and 0 otherwise.
- The cost function?

$$\max \sum_{i=2}^{N-1} \sum_{j=2}^N s_i x_{i,j}$$

- We have to start from node 1 and end into node N

$$\sum_{j=2}^N x_{1,j} = \sum_{i=1}^{N-1} x_{i,N} = 1$$

- Each node must be visited at least once and the path must be connected

$$\sum_{i=1}^{N-1} x_{i,k} = \sum_{j=2}^N x_{k,j} \leq 1, \quad \forall k = 2, \dots, N-1$$

- The total time should not exceed the budget

$$\sum_{i=1}^{N-1} \sum_{j=2}^N d_{i,j} x_{i,j} \leq d_{\max}$$

- Constraint for subtours elimination

$$u_1 = 1$$

$$2 \leq u_i \leq N$$

$$u_i - u_j + N x_{i,j} \leq N-1,$$

$$\forall i = 2, \dots, N$$

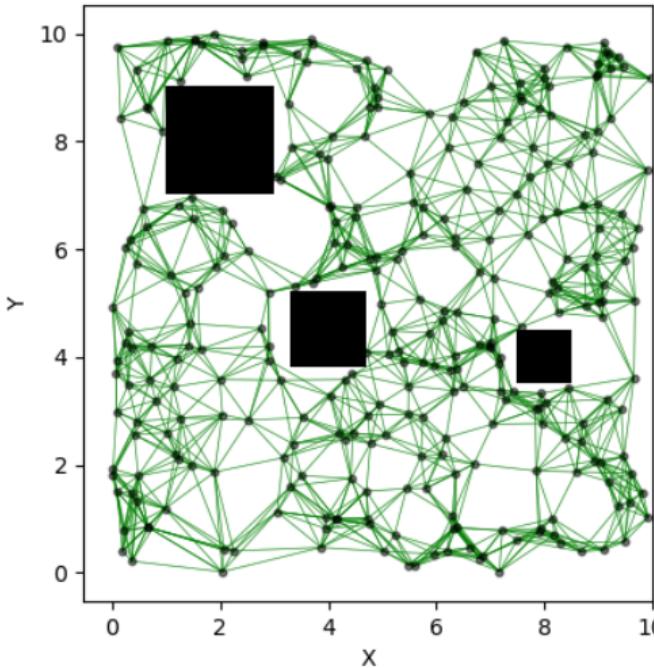
$$\forall i \neq j \quad i, j \in \{2, \dots, N\}$$

- Note: scores are counted only on intermediate nodes, while all travel costs, including the first edge from the start, are considered in the budget constraint.

Difficulties

- ▶ G.Golden et al. (1987) prove that the OP is NP-hard; no polynomial time algorithm has been designed, or is expected to be designed, to solve this problem to optimality.
- ▶ This implies that exact solution algorithms are very time consuming and for practical applications heuristics will be necessary
- ▶ heuristics may direct the algorithm in undesirable directions, they do not efficiently explore the whole solution space
- ▶ determining a (shortest) path between the selected vertices becomes more complicated when the number of vertices increases.
- ▶ The score of a vertex and the time to reach the vertex are independent and often contradictory to each other

Implementation

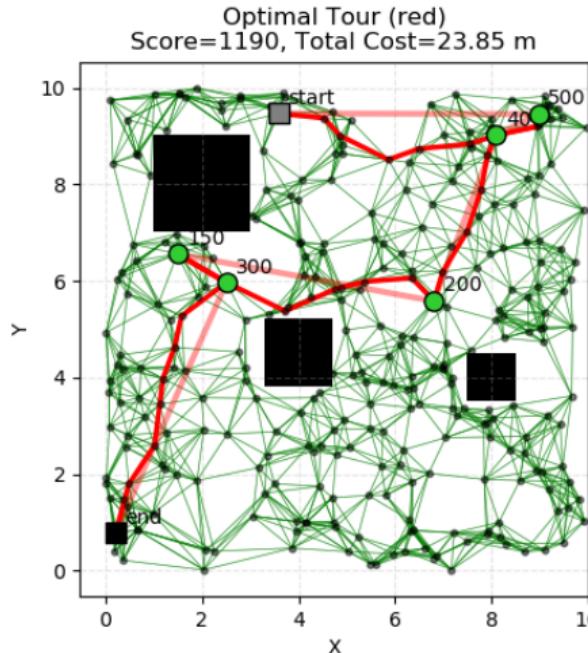


- ▶ we create a 10×10 m map with the following squared obstacles:
 - ▶ obs1: center=(2,8), side=2m
 - ▶ obs2: center=(4,4.5), side=1.5m
 - ▶ obs3: center=(8, 4), side=1m
- ▶ we generate a visibility graph by the PRM algorithm, with 300 samples and connectivity 10

Implementation

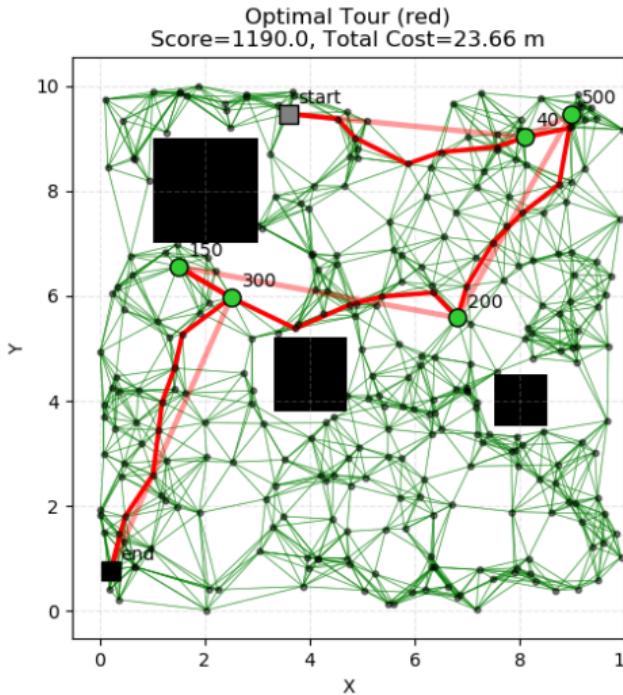
- ▶ we set the following start and goal location: start = (4, 9), goal = (0,1)
- ▶ We set the following victims location inside the 10x10 m map: [(9.1, 9.5), (6.7, 5.4), (1.36, 6.34), (3, 6.34), (8,9)]
- ▶ We need to approximate the positions of the victims/start/goal to the closest vertices in the PRM.
- ▶ we associate the following costs to the victims: [500, 200, 150, 300, 40]
- ▶ we set a time/distance/cost budget of $d_{max} = 25m$

Results: Brute force



- ▶ Precompute DJKSTRA between all the nodes and store into a list
- ▶ I consider all the possible combinations of subset of victims
- ▶ for each possible sequence (start+subset of victims+goal) compute the total path length and the score
- ▶ the optimum is the sequence with maximum score for which the total length is lower than $d_{max} = 25m$
- ▶ total score: 1190 (maximum), Total cost = 23.85m, num. visited victims: 5/5, run time: 0.017 s

Results: MILP (Pulp)



- ▶ Alternatively, we can leverage Pulp library as a modeling layer to build the Mixed Integer Linear Program variables, constraints and objectives
- ▶ subsequently we pass it to the solver: e.g. Gurobi, CBC, CPLEX, GLPK.
- ▶ total score: 1190 (maximum), Total cost = 23.66m, num. visited victims: 5/5, run time: 0.042s
- ▶ same score with lower cost

Comparison with 10 victims

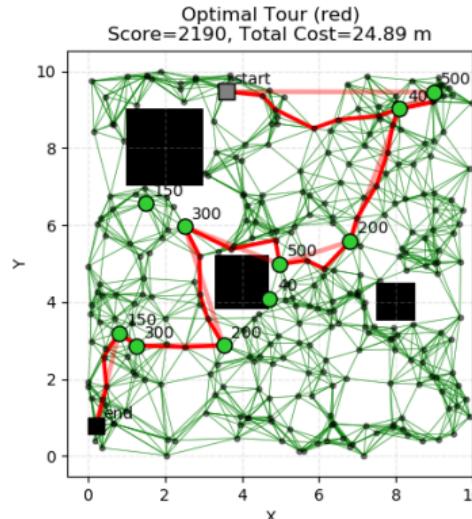


Figure: Brute Force

- ▶ total score: 2190
- ▶ Total cost = 24.89m
- ▶ num. visited victims: 8/10,
- ▶ run time: 23.6s

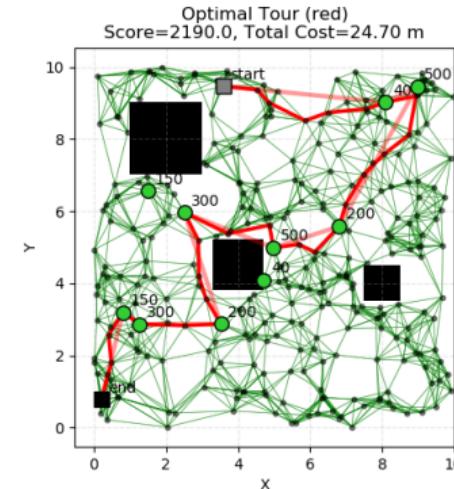


Figure: Pulp

- ▶ total score: 2190
- ▶ Total cost = 24.7m
- ▶ num. visited victims: 8/10,
- ▶ run time: 0.33s

Comparison with $d_{max} = 15m$

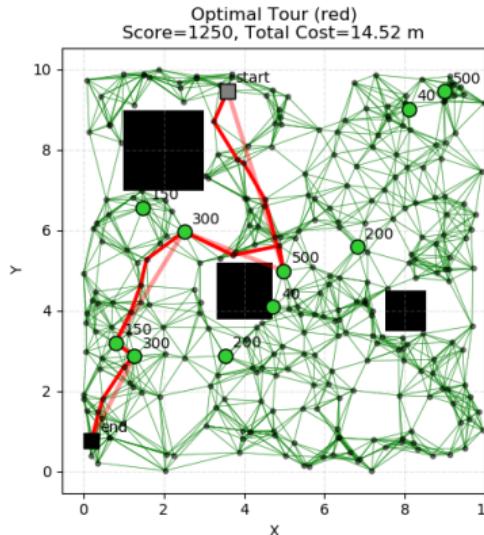


Figure: Brute Force

- ▶ total score: 1250
- ▶ Total cost = 14.52m
- ▶ num. visited victims: 4/10,
- ▶ run time: 25.6s

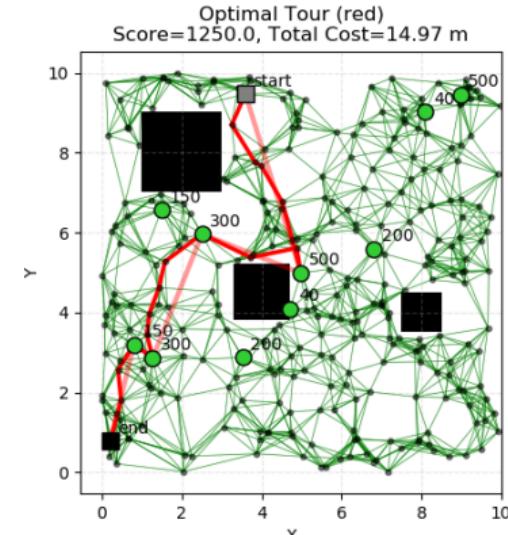


Figure: Pulp

- ▶ total score: 1250
- ▶ Total cost = 14.97m
- ▶ num. visited victims: 4/10,
- ▶ run time: 0.07s

Some notes on pursuit evasion game

Robot Planning and its Applications
University of Trento

Luigi Palopoli & Michele Focchi





Outline

Introduction

Modelling

Project 3

Two robots: pursuer and evader.

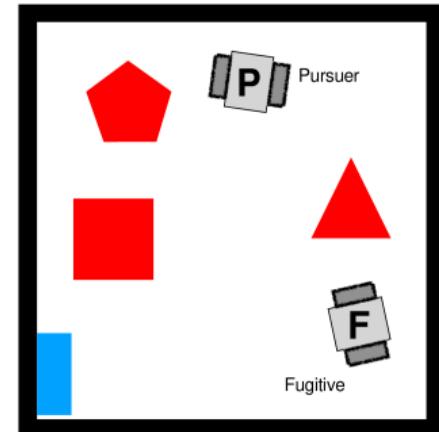
- ▶ The evader tries to reach the exit
- ▶ The pursuer wants to catch the fugitive before it runs away.

Pursuit–evasion problems have a long and rich history in:

- ▶ Differential games (Isaacs, Differential Games, 1965),
- ▶ Graph theory (the “cops and robbers” game — Nowakowski & Winkler 1983),

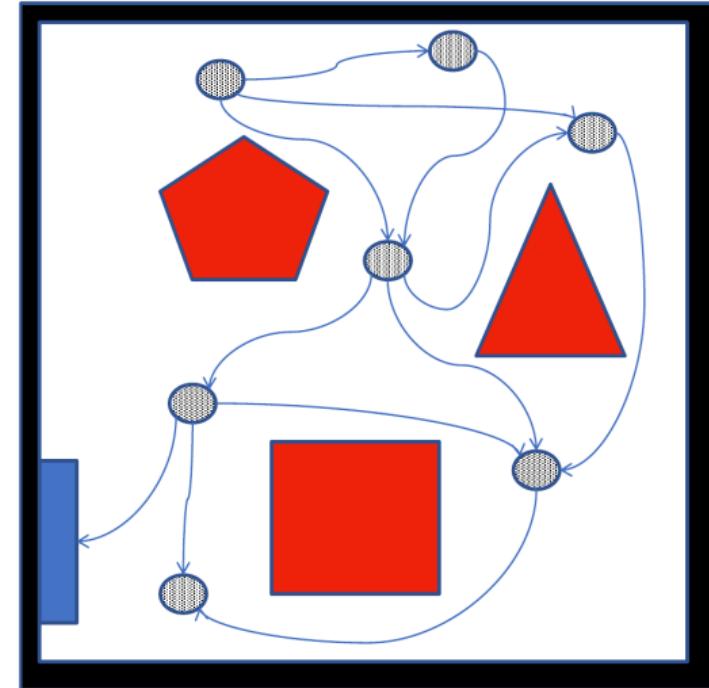
In robotics, these problems appear in:

- ▶ Surveillance and security (tracking intruders),
- ▶ Rescue robotics (locating a moving target),
- ▶ Autonomous driving (avoiding pursuers/collisions),
- ▶ Multi-robot strategies.



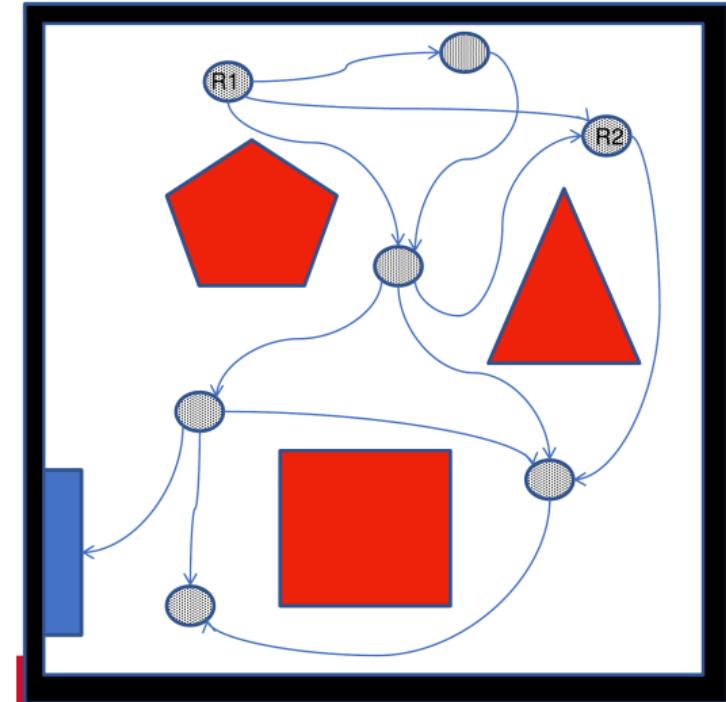
Roadmap discretisation

- ▶ We have seen that possible strategy is to first create a roadmap discretising the problem.
- ▶ Discretization converts a continuous pursuit–evasion game into a graph game, where:
 - ▶ The state is a pair of graph nodes (pursuer position, evader position).
 - ▶ Actions are moves along edges.
- ▶ This allows you to use discrete game-theoretic algorithms such as minimax.

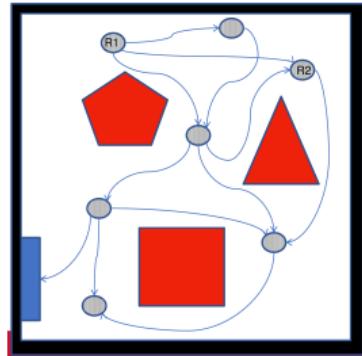


Synchronous hypotheses

- ▶ We will assume that the game evolves in turns.
- ▶ Every turn corresponds to the motion of the robot between two different locations.
- ▶ A turn ends when all robots reach their chosen node
- ▶ synchronous turns simplify the problem to a finite-state game tree
- ▶ we have a **competitive** environment: two agents have conflicting goals, → adversarial search problems

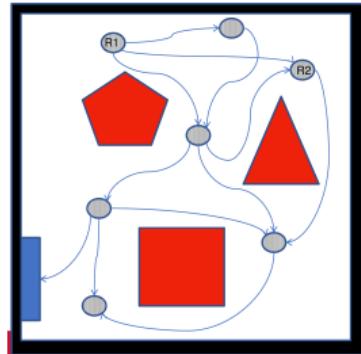


Example execution

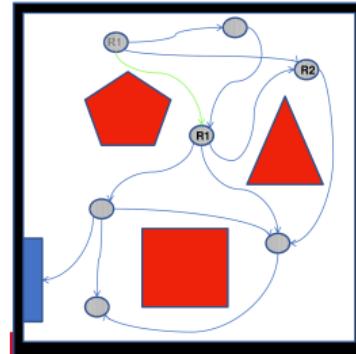


(1)

Example execution

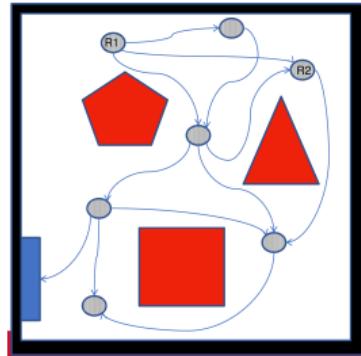


(1)

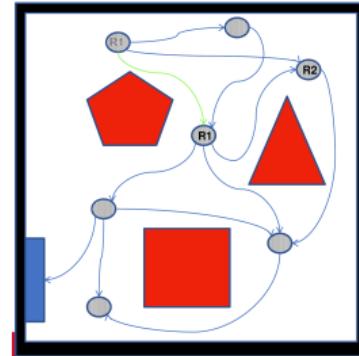


(2)

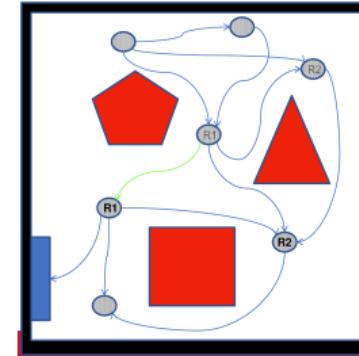
Example execution



(1)

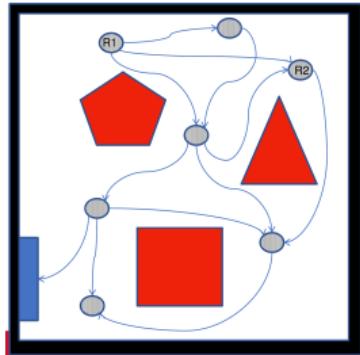


(2)

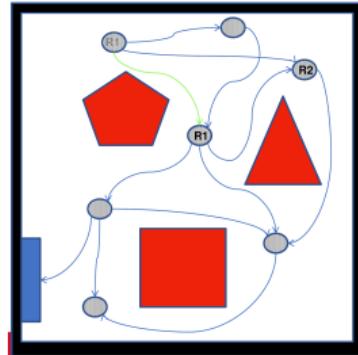


(3)

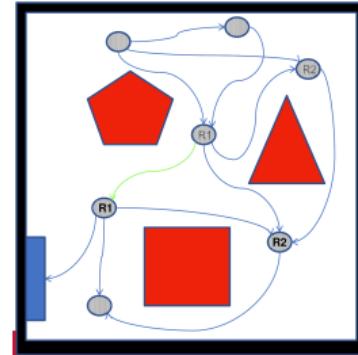
Example execution



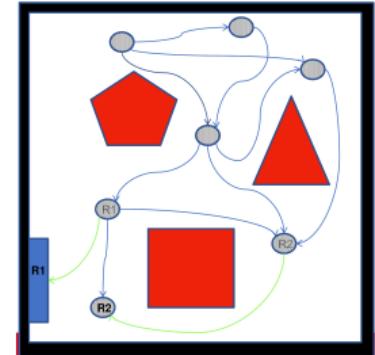
(1)



(2)

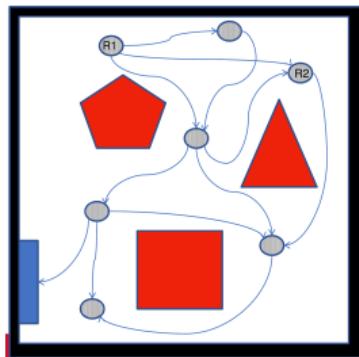


(3)

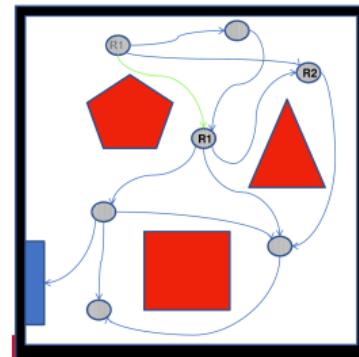


(4)

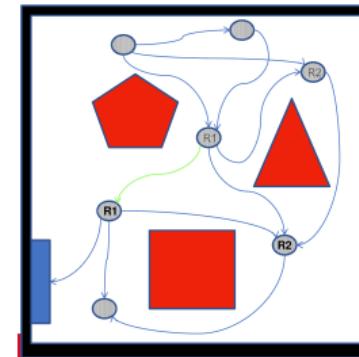
Example execution



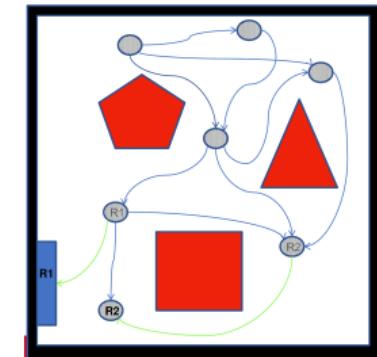
(1)



(2)



(3)



(4)

Result of the game

The evader wins!



Outline

Introduction

Modelling



Games

- ▶ We have seen an example of a deterministic, turn-taking, perfect information, zero-sum game.
- ▶ the state of a game is easy to represent, agents are usually restricted to a small number of actions whose effects are defined by precise rules.
- ▶ Deterministic and turn-taking are obvious attributes of our game
- ▶ *Perfect information* means that the two players have a state and that they know everything about each other's state (full observability)
- ▶ *Zero-sum* means that what one gains, the other one loses. There is no possibility of a win-win outcome.
 - ▶ *Utility*: Our utility in this can be:

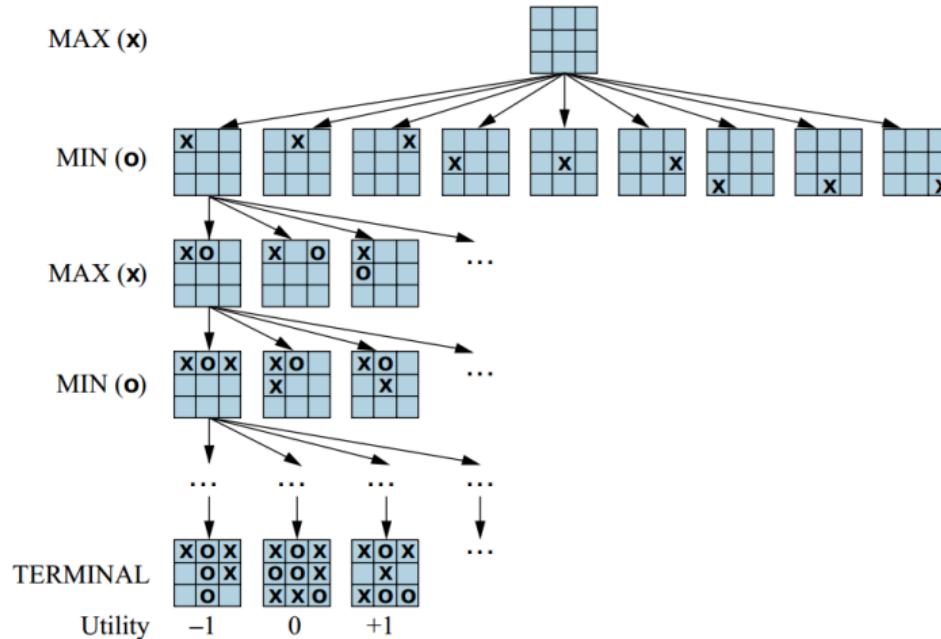
$$\text{Utility} = \begin{cases} +1 & \text{Pursuer wins} \\ -1 & \text{Evader wins} \\ 0 & \text{Draw (e.g., stalemate)} \end{cases}$$

The Game Formal definitions

The game just described can be modelled by the following elements

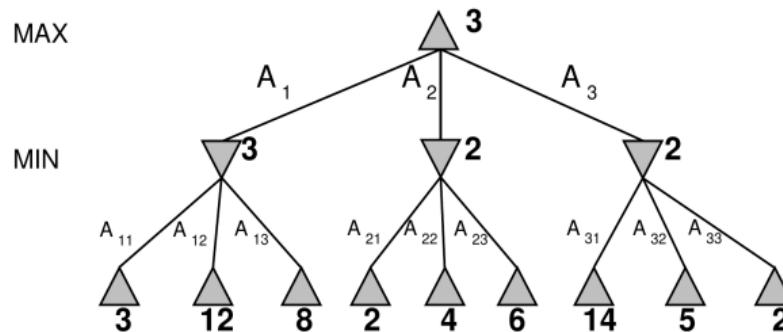
- ▶ S_0 : *Initial state*, how the game is set up.
- ▶ $\text{TO-MOVE}(s)$: the player that must move at state s
- ▶ $\text{ACTIONS}(s)$: set of legal moves at state s
- ▶ $\text{RESULT}(s, a)$: the new state resulting from the applications of action a at state s
- ▶ $\text{IS-TERMINAL}(s)$: returns true if the state is associated with a terminal condition (e.g., the evader escapes or is caught):
- ▶ $\text{UTILITY}(s, p)$: returns the utility for the player p when it reaches the terminal state s
- ▶ ACTIONS function, and RESULT function define the state space graph: Actions (edges) \rightarrow Moves, States (vertices) \rightarrow positions
- ▶ superimpose a **search tree** over part of that graph to determine what move to make
- ▶ **game tree**: a search tree that follows every sequence of moves all the way to a terminal state.

Example 1: Tic-tac-toe Game



- ▶ Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three squares in a row or all the squares are filled

Example 2: A two-ply game tree



- ▶ The possible moves for MAX at the root node are labeled A1, A2, and A3
- ▶ The possible replies to A1 for MIN are A11, A12, A13, and so on
- ▶ This particular game ends after one move each by MAX and MIN.
- ▶ The utilities of the terminal states in this game range from 2 to 14

Optimal decisions in Games

- ▶ The pursuer wants to maximise its utility (MAX player), the evader wants to minimise its utility (MIN player).
- ▶ MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves.
- ▶ The best move is found by an algorithm called **minimax search**
- ▶ Given a game tree, the optimal strategy can be determined by working out the Minimax value of each state in the tree

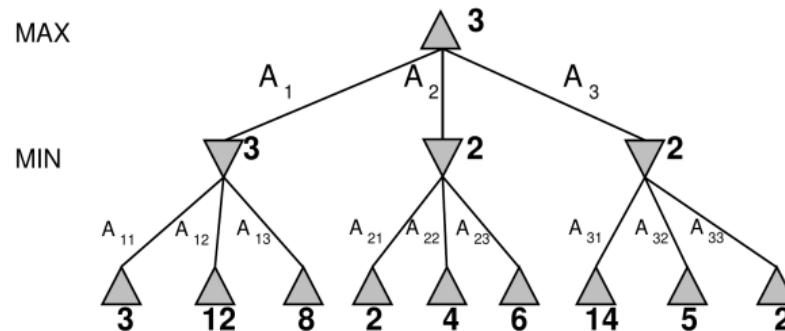
Minimax value

The minimax value associated with a state s - called $\text{MINIMAX}(s)$ - is the utility for player MAX assuming that both players optimally from state s to the end of game. For a terminal state is just its utility.

The minimax value can be computed recursively, as in the the following definition:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if IS-TERMINAL}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

Minmax with the two-ply game



- ▶ The first MIN node, has three successor states with values 3, 12, and 8, so its minimax value is 3
- ▶ the other two MIN nodes have minimax value 2
- ▶ The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3
- ▶ at the root the action is the optimal choice for MAX because it leads to the state with the highest minimax value (3).

The minimax search algorithm

Finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value.

```
procedure MINIMAX-SEARCH(game, state) return an action
    if game.TO-MOVE(state) = MAX then
        move ← arg maxa ∈ ACTIONS(state) MAX-VALUE(RESULT(state, a))
    else
        move ← arg maxa ∈ ACTIONS(state) MIN-VALUE(RESULT(state, a))
    end if
    return move
end procedure

procedure MAX-VALUE(state) return an utility value
    if IS-TERMINAL(state) then
        return UTILITY(state)
    end if
    v ← -∞
    for a ∈ ACTIONS(state) do
        v ← max(v, MIN-VALUE(RESULT(state, a)))
    end for
    return v
end procedure

procedure MIN-VALUE(state) return an utility value
    if IS-TERMINAL(state) then
        return UTILITY(state)
    end if
    v ← +∞
    for a ∈ ACTIONS(state) do
        v ← min(v, MAX-VALUE(RESULT(state, a)))
    end for
    return v
end procedure
```

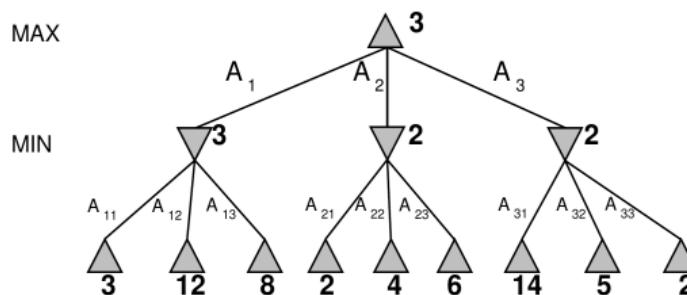
- ▶ The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

The minimax algorithm

- ▶ The minimax algorithm performs a depth-first search of the game tree.
- ▶ If the depth of the tree is m and there are b moves available at each step:
 - ▶ Time complexity: $O(b^m)$
 - ▶ Space complexity $O(bm)$ for an algorithm that generates all actions at once
- ▶ For chess games we have: $b \approx 35$, $m \approx 80$ the time complexity is $O(35^{80}) = O(10^{123})$ states to search.
- ▶ The minimax algorithm is impractical for large games

Pruning techniques

- ▶ The exponential nature of the minimax algorithm cannot be tamed
- ▶ However, it is possible to simplify the search, pruning trees that I already know not produce any useful information.



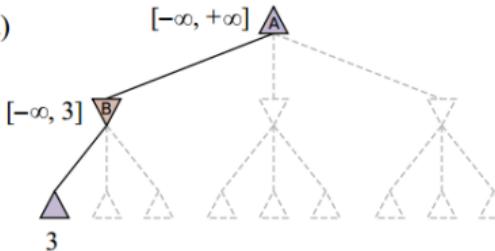
- ▶ When I explore the first subtree I already know now that min will be greater than 3
- ▶ As soon as I reach the leaf 2 in the second subtree, I know it is pointless to continue

$$\begin{aligned} \text{MINIMAX}(\text{root}) = \\ \max(\min(3, 12, 18), \min(2, x, y), \min(14, 15, 2)) \end{aligned}$$

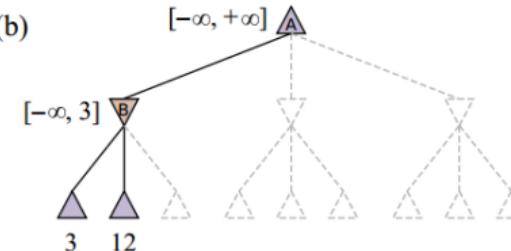
- ▶ (x, y) have no influence on the result
- ▶ This technique is called alpha-beta pruning

Alpha beta pruning Example

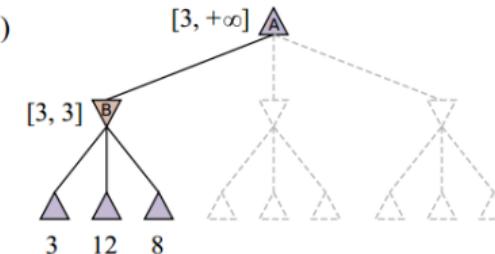
(a)



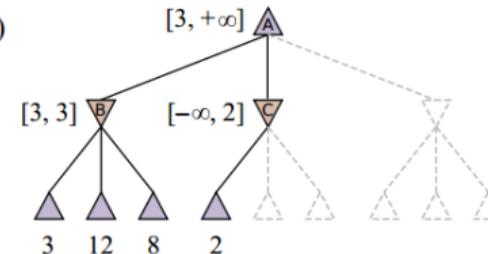
(b)



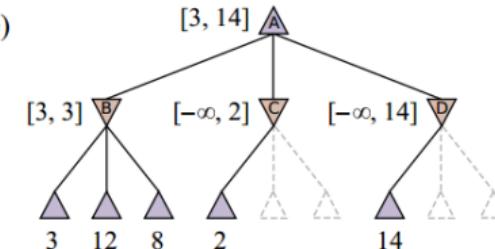
(c)



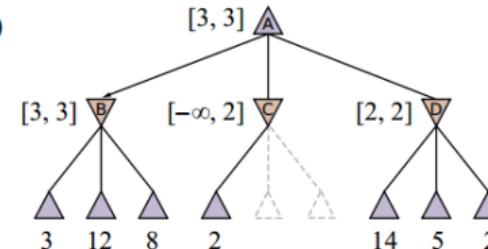
(d)



(e)



(f)



Alpha beta pruning Example

- ▶ α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: α = "at least."
- ▶ β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: β = "at most".
- ▶ Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively

Multi-Agent Path Finding

Luigi Palopoli, Enrico Saccò



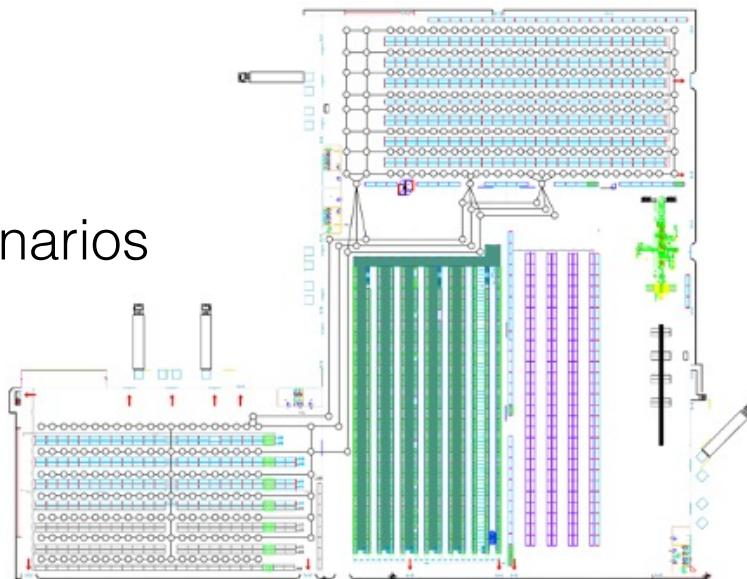
UNIVERSITÀ
DI TRENTO

What?

- The standard Multi-Agent Path Finding (MAPF) problem [1] consists in:
given a map and N agents, finding the best *feasible joint plan* Π such that each agent moves from its initial position to its final position minimizing an *objective function*
- It's a combinatorial problem
- In the standard definition, the map is usually a **grid**
- Solution minimizes objective function --> time, space, resource, etc.
- **Centralized approach**

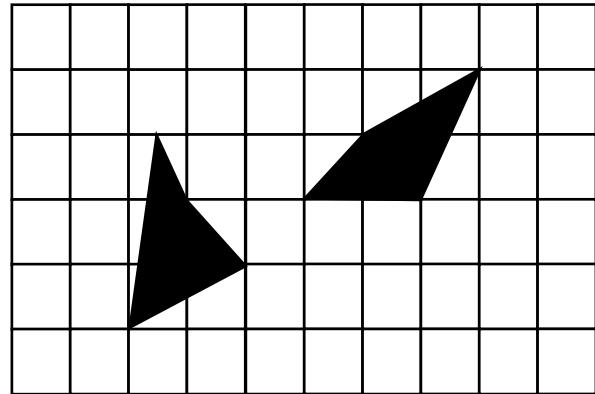
Why?

- Robotics is a main topic in the Industrial Revolution 4.0 and 5.0
- Used in an increasing number of scenarios
- Challenging problem [2]
- Algorithms can be applied to other scenarios



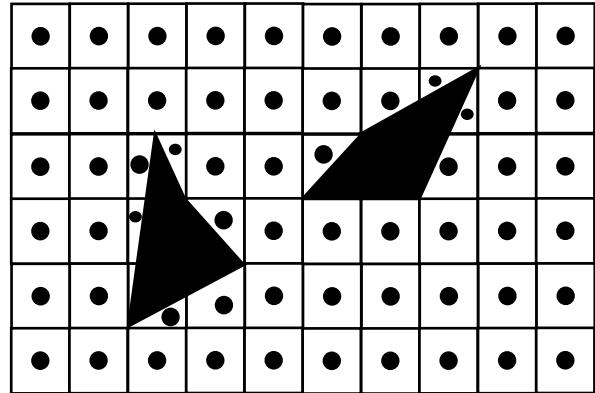
Map Decomposition

- How do I deal with obstacles and map borders?
- Many algorithms to partition the map in cells:
 - Exact cell decomposition
 - Approximate cell decomposition
 - Maximum clearance
 - Morse decomposition
 - Brushfire decomposition
- Each cell is a node --> connectivity graph



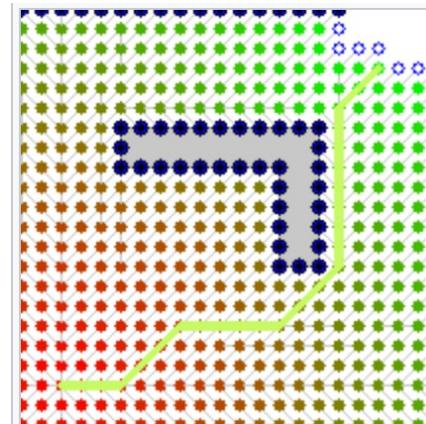
Map Decomposition

- How do I deal with obstacles and map borders?
- Many algorithms to partition the map in cells:
 - Exact cell decomposition
 - Approximate cell decomposition
 - Maximum clearance
 - Morse decomposition
 - Brushfire decomposition
- Each cell is a node --> connectivity graph



Single-Agent Path Finding (SAPF)

- Given a graph $G = (V, E)$, find the *best feasible* plan π_i to go from an initial position to a final position
 - The problem usually consists in computing the shortest path between two nodes on a graph
 - Deterministic algorithms, e.g., Dijkstra's
 - Heuristic algorithms, e.g., A*

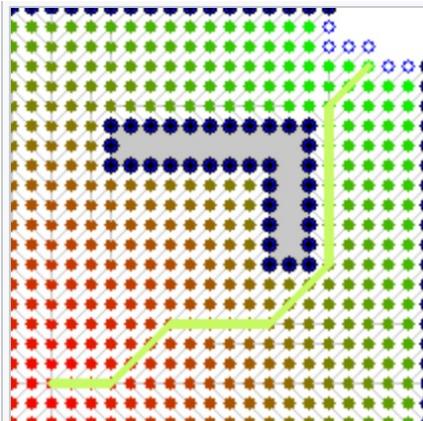


Single-Agent Path Finding (SAPF)

Dijkstra's

- Complete
- Optimal
- Evolution of BFS
- Cost function:

$$f(x) = g(x)$$

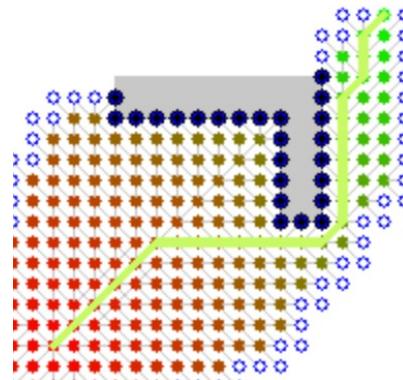


A*

- Complete?
- Optimal?
- Admissible heuristic $h(x)$:

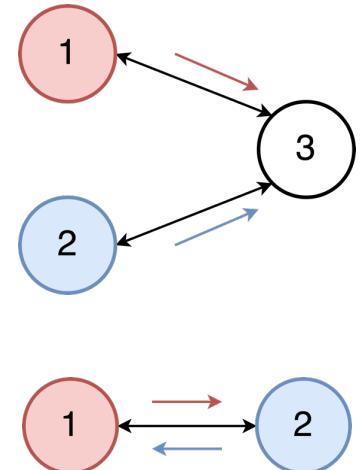
$$f(x) = g(x) + h(x)$$

$$h(x) \leq d(x, y) + h(y)$$

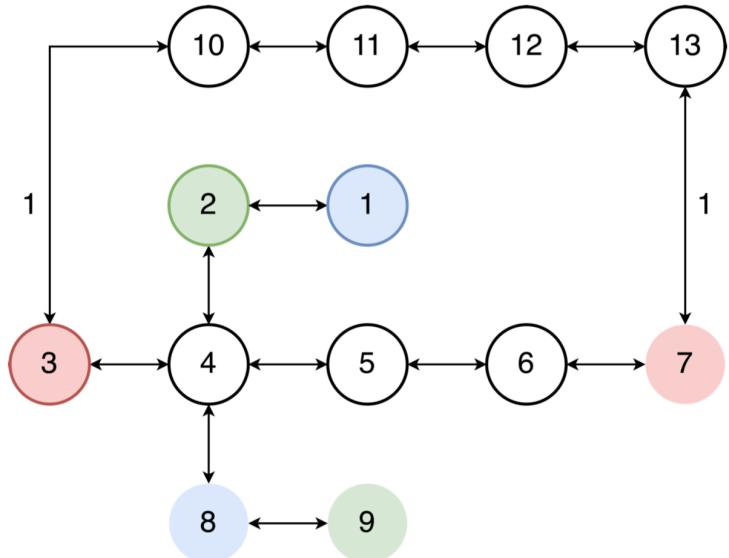


Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- A joint plan is a set of single plans: $\Pi = \{\pi_1, \dots, \pi_k\}$
- A path if feasible if no conflict arises [1]:
 - Vertex conflicts
 - Edge conflicts
 - Swap conflicts
- Objective functions:
 - Makespan (MKS)
 - Sum of individual costs (SIC)



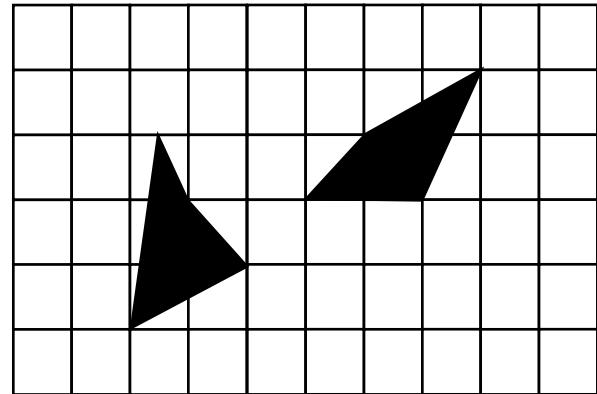
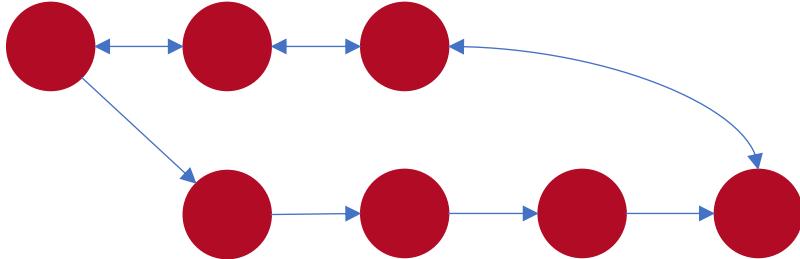
Multi-Agent Path Finding Cost Functions



Π_i	$SIC(\Pi_i)$	$MKS(\Pi_i)$
$\Pi_1 = \begin{cases} \pi_1 = \{3, 10, 11, 12, 13, 7\} \\ \pi_2 = \{2, 4, 8, 9\} \\ \pi_3 = \{1, 2, 4, 8\} \end{cases}$	14	6
$\Pi_2 = \begin{cases} \pi_1 = \{3, 4, 5, 6, 7\} \\ \pi_2 = \{2, 2, 4, 8, 9\} \\ \pi_3 = \{1, 1, 2, 4, 8\} \end{cases}$	15	5

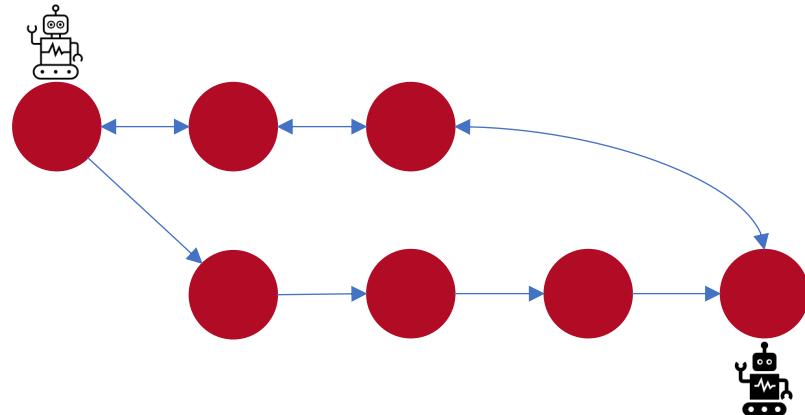
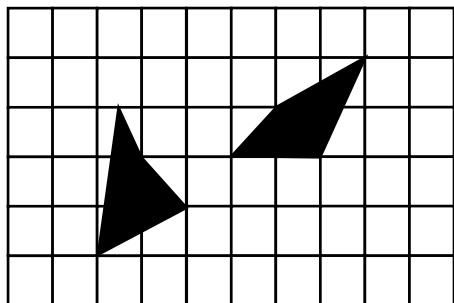
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation



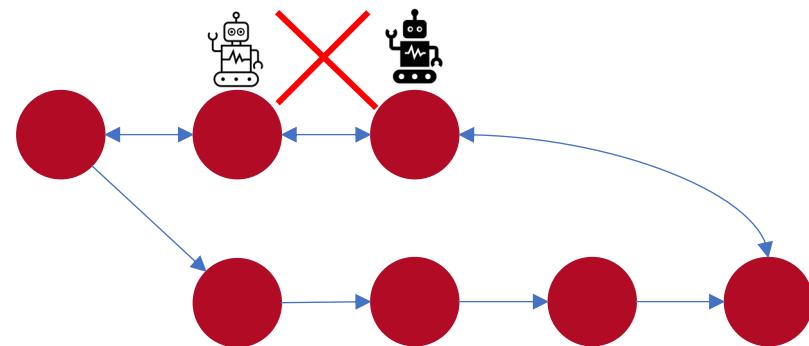
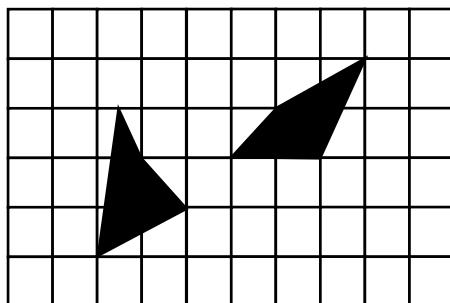
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs



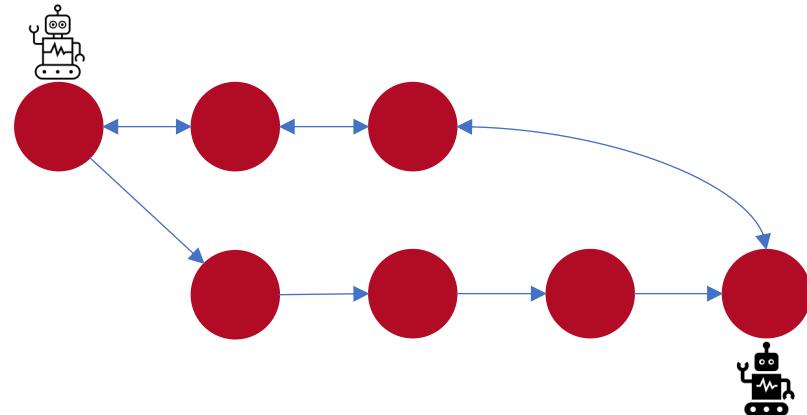
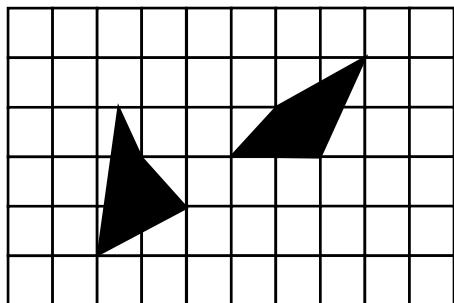
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
 - Time is **discretized**
 - Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
 - Edges have **unitary** costs



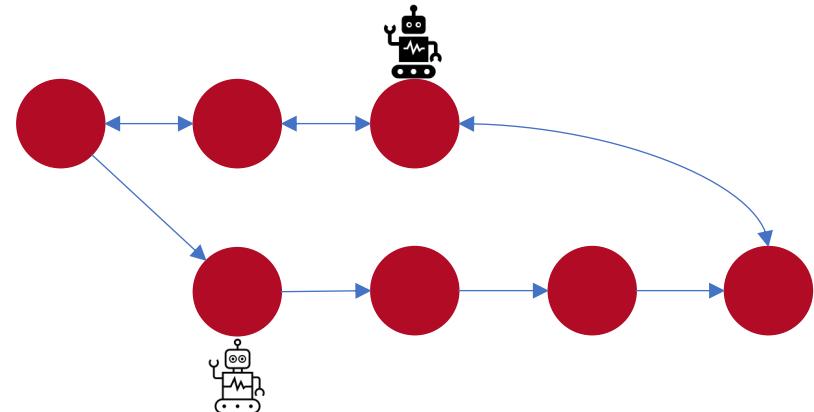
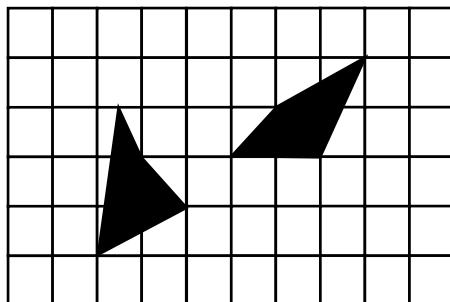
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs



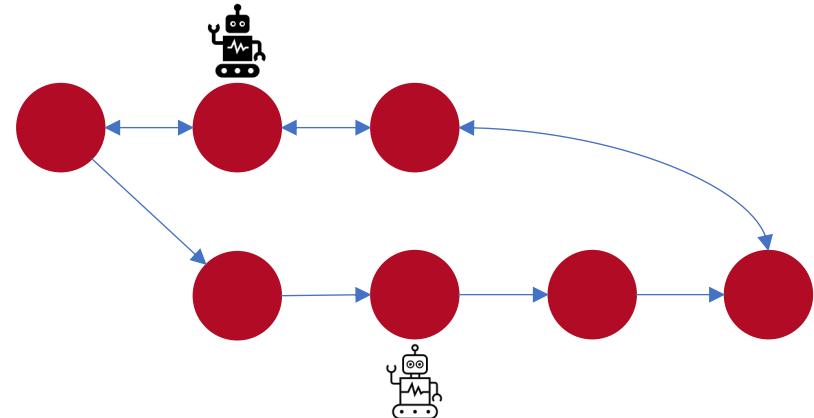
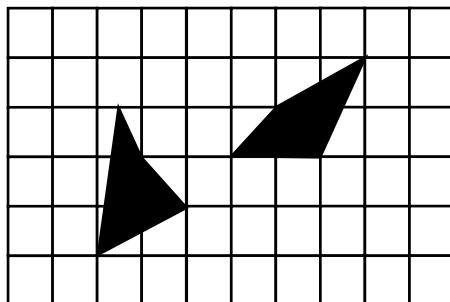
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs



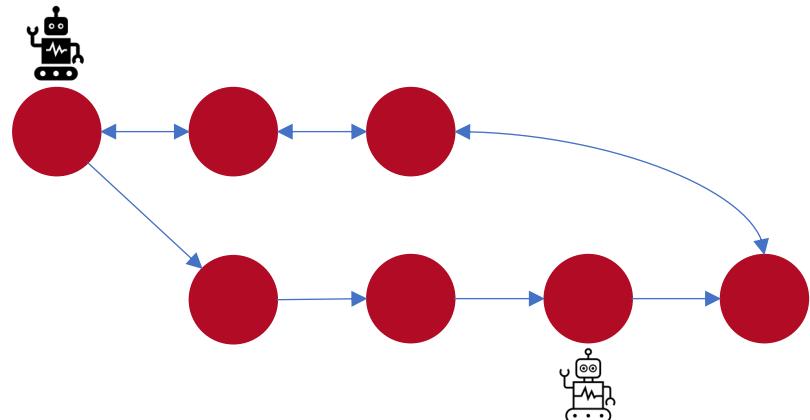
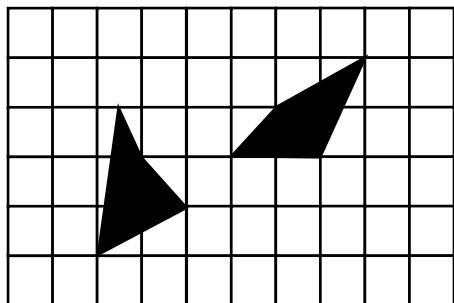
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs



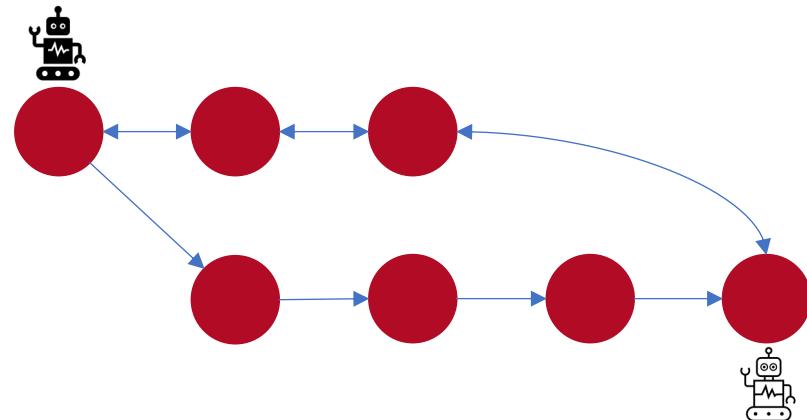
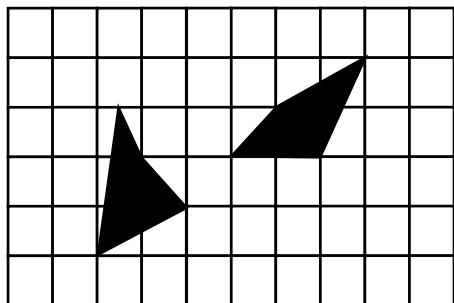
Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs



Multi-Agent Path Finding Overview

- Given a graph $G = (V, E)$ and k agents, find the *best feasible joint* plan Π such that each agent moves from its initial position to its final position minimizing an objective function
- Time is **discretized**
- Each agent can either
 - move to an adjacent cell; or
 - stay on the same cell --> variation
- Edges have **unitary** costs

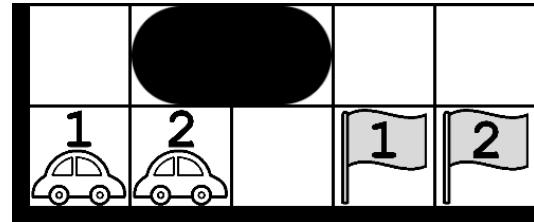
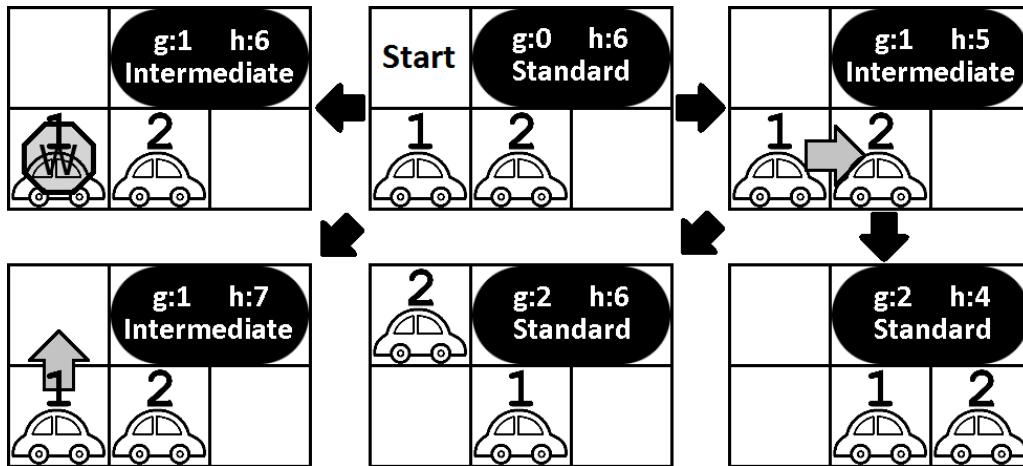


Enhanced Versions of A* [3]

- Instead of considering one position, considers a tuple
- At each timestep:
 - the current state space contains the position of all N agents
 - the next state space has to consider all possible movements of the agents
- This gets bad pretty fast
- **Operator decomposition (OD)**
- **Simple independence detection (SID)**

Enhanced Versions of A* – OD

- Do not consider N agents per each time step
- Considers 1 agent at a time and it requires N operations to advance 1 timestep
- The order in which the agents are chosen is fixed
- It's not complete or optimal without the correct heuristic



Enhanced Versions of A* – SID

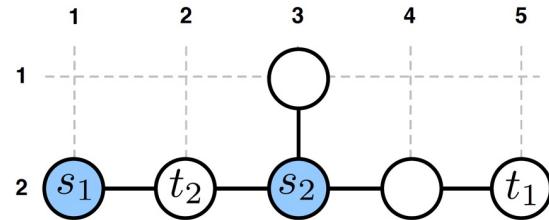
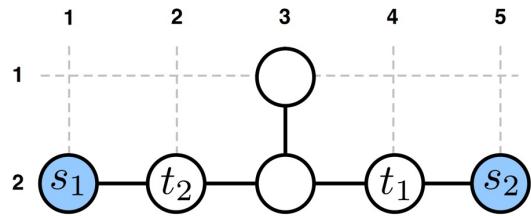
- Also in OD, the search space is still exponential in the number of agents
- Agents whose path *does not collide*, are in **independent** group and should be considered separately
- The algorithm follow these steps:
 - Start with the optimal paths as if the agents were alone
 - If there are conflicts, divide the agents in conflict groups
 - Solve the conflicts in the group
 - Repeat

Priority Planning (PP) [4]

- Each agent has a fixed priority
- The higher the priority, the first the agent's path is computed
- Pros:
 - This allows for keeping the complexity small
- Cons:
 - Not complete [5]
 - Not optimal
- Many works, focus on using ML or DL approaches to learning the priorities

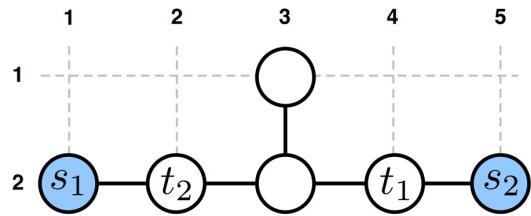
Priority Planning (PP)

- There are some instances in which priority planning cannot solve the problem [5]

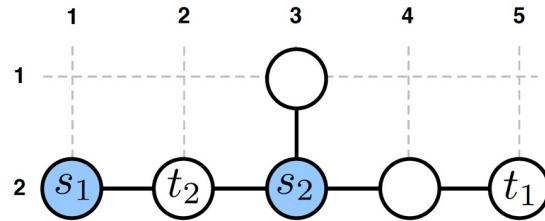


Priority Planning (PP)

- There are some instances in which priority planning cannot solve the problem [5]



Not solvable



Solvable

- A MAPF instance must be *well-formed* to be solvable
 - Agents can wait for any amount of time on the initial or final node without blocking other agents

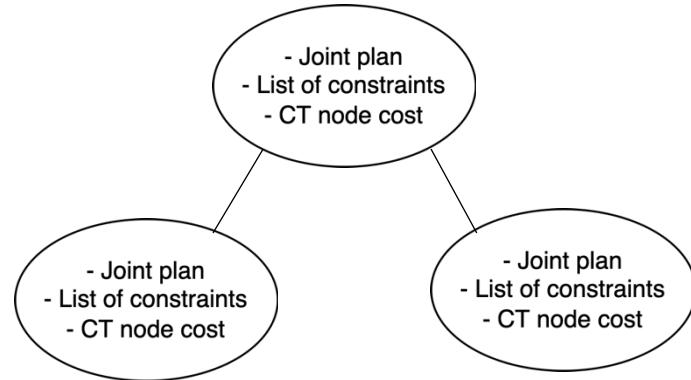
Constraint Based Search (CBS)

- Proposed in 2015 by G. Sharon, R. Stern, A. Felner and N. R. Sturtevant [6]
- Optimal algorithm divided in two phases:
 1. High-level search → manages conflicts
 2. Low-level search → SAPF problem

Constraint Based Search (CBS)

1. High-level search → manages conflicts

- Creates a *constraint tree*
- When a conflict is found, creates two children:
 - Agent a_i cannot be on node n at time t
 - Agent a_j cannot be on node n at time t
- The search continues until a joint plan without conflicts is found
- Nodes to be explored are chosen based on their joint cost

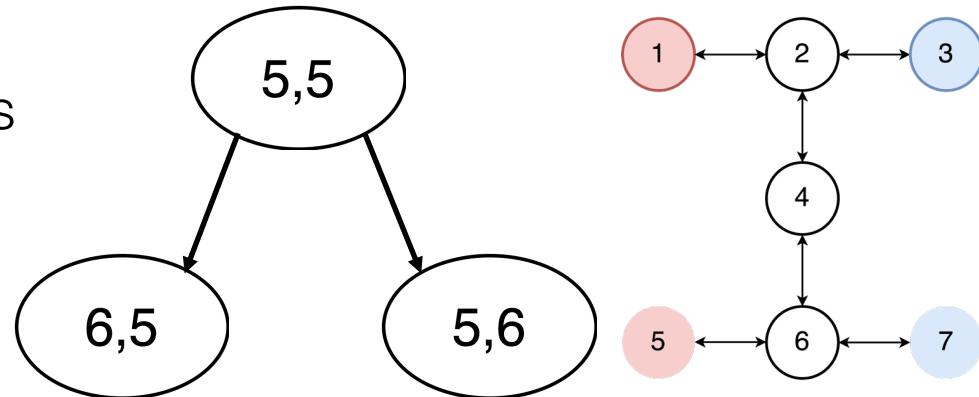


CBS Implementation – High-Level

- Start from a root node with:
 - No constraints
 - Joint plan computed as SAPF
- Iterate until a feasible solution is found
 - If one or more vertex conflicts were found, then create two new nodes:
 - Child 1: agent a_i cannot be on node n at time t
 - Child 2: agent a_j cannot be on node n at time t
 - If one or more swap conflicts were found, then create two new nodes:
 - Child 1: agent a_i cannot move from node n_1 to node n_2 at time t
 - Child 2: agent a_j cannot move from node n_2 to node n_1 at time t
- Conflicts are checked by comparing the positions of the solutions
- A joint plan is updated only for the new constraint's agent

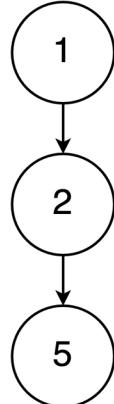
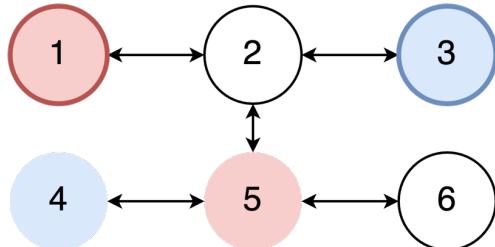
Increasing Cost Tree Search (ICTS)

- It was proposed in 2013 by G. Sharon, R. Stern, M. Goldenberg and A. Felner and it is optimal [7]
- Similarly to CBS, ICTS is divided in two searches: high-level and low-level
- It uses an Increasing Cost Tree
- For each new level, k new nodes are created
- The search continues until a feasible solution is found

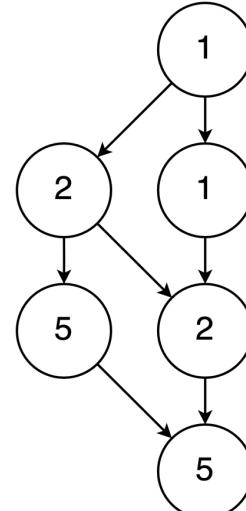


ICTS – Low-Level Search

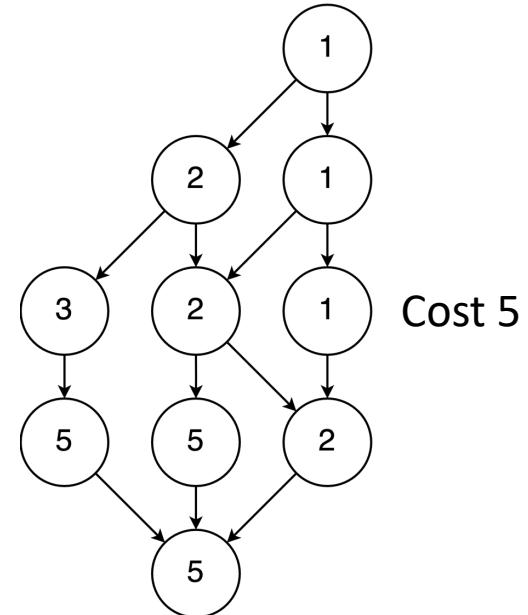
- The low-level search is implemented using Multi-value Decision Diagrams (MDDs) [8]
- An MDD contains all the paths for an agent going from its initial position to their goal with a certain cost



Cost 3



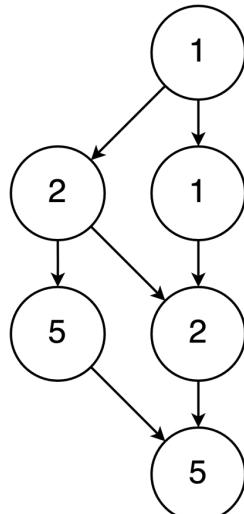
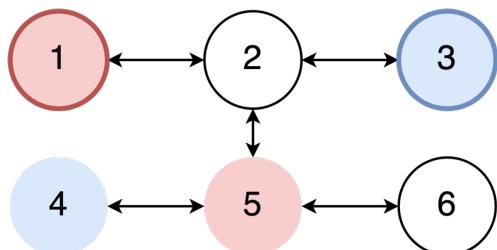
Cost 4



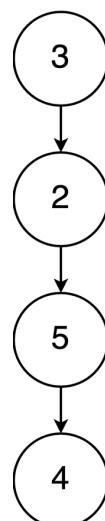
Cost 5

ICTS – Low-Level Search

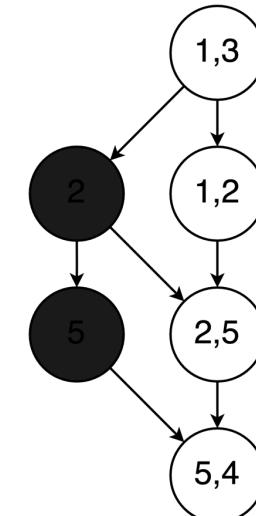
- We can merge the MDDs of different agents to check for possible solutions
- The branches that have conflicts are removed



Red agent



Blue agent



Merged MDD

Constraint Programming (CP) [9]

- MAPF has been proven to be NP-Hard [10] → it can be reduced to SAT and MILP
- Constraint programming is a mathematical modeling paradigm in which some constraints are placed over some variables.
- Some constraints are:
 - Agents must be only on one vertex at each time step;
 - A node can be occupied by at most one agent at a time;
 - Agents start from their initial position and must be on their arrival position at the end;
 - Agents must move along edges.

Constraint Programming – Implementation

- Bartak et al use Picat
- IBM's CPLEX for performance or Google's or-tools
- Decision variables:

$X[n_steps][n_nodes][n_agents]$ $movement[n_agents][n_steps]$
 $goal_points[n_steps][n_nodes][n_agents]$ $edges[n_agents][n_steps]$

- An agent a can be only on one node n at a time s :

$$\forall s \in S, \forall a \in A, \sum_{n \in N} X[s][n][a] = 1$$

- C++:

```
FOREACH(s, steps) {  
    FOREACH(a, agents) {  
        IloExpr expr(env);  
        FOREACH(n, nodes) {  
            expr += x[s][n][a];  
        }  
        model.add(x: expr <= 1);  
    }  
}
```

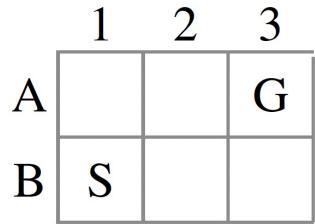
Constraint Programming – Constraints

- Examples of constraints are:
 - Agents cannot be on more than node each time step
 - A node cannot be occupied by more than one agent at time
 - An agent must occupy a neighbor of the node it is on at time $t + 1$ or stay on the same node
 - Agents start on their initial positions and end on their final positions
 - At a certain time, an edge cannot be used in more than one direction
 - The movement cost of an agent at a given time is the cost of the edge it is traversing
 - The agent must go through all the goals and only once before reaching the final position

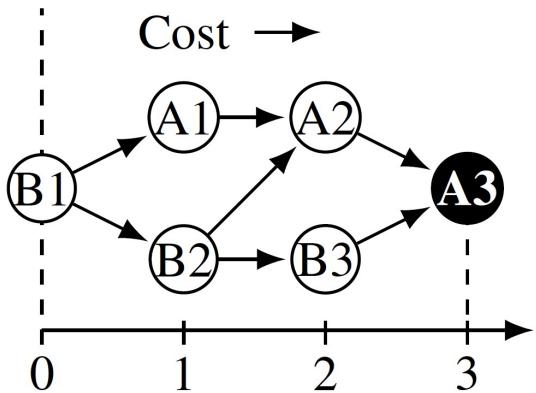
Extended ICTS [11]

- Standard MAPF considers edges with unit costs
- Assumptions on agents' movements:
 - Wait on the center of the node
 - Moves in a straight line
 - Collision is the overlapping in an instant of time
- Two problem:
 - Partial time overlap conflict detection
 - Detect conflicts
 - Partial time overlap successor generation
 - Generate successive states

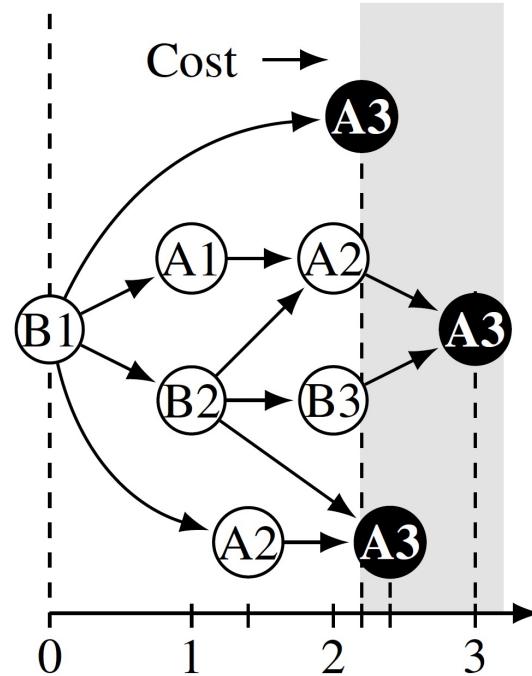
Extended ICTS



(a)



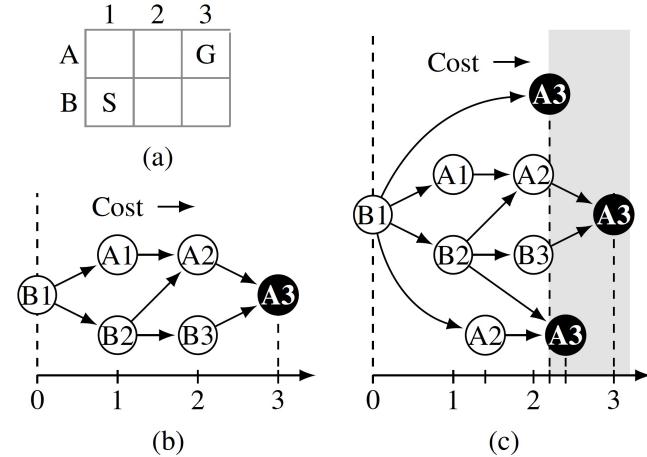
(b)



(c)

Extended ICTS – Optimal

- In ICTS, the MDD had one root and one sink
- The next child is not obtained with an increment of 1, but we need to set an increment value δ
 - If δ is small --> the depth of the ICT may become too big
 - If δ is large --> search is reduced, but the solution may not be optimal
- The ICT nodes now contain intervals:
 - Lower bound is the solution minimum
 - Higher bound is the solution maximum
- The low-level is changed from a satisfactory problem to an optimal one: find the solution with the minimum cost in the interval



Extended ICTS – Heuristics

- ϵ -ICTS: considers the low-level a satisfactory problem
 - This allows the algorithm to find a solution that is bounded sub-optimal
- w -ICTS: we can bound the sub-optimality for the generation of the next step to values of δ by adding a weight value w

CBS Heuristics

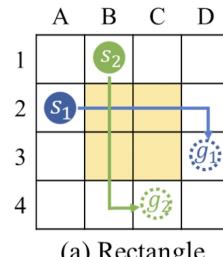
- CBS has been the start of the show with many improvements:
 - Bypassing conflicts [12]
 - Prioritizing conflicts [13]
 - Symmetry reasoning [14]
- And also a number of different heuristics:
 - ECBS [15]
 - EECBS [16]
 - EEEEECBS (Saccon et al., 2025)
 - EEEEEEEEEEECBS (Saccon et al., 2030).

CBS Heuristics

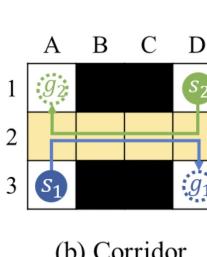
- CBS has been the start of the show with many improvements:
 - Bypassing conflicts [12]
 - Prioritizing conflicts [13]
 - Symmetry reasoning [14]
- And also a number of different heuristics:
 - ECBS [15]
 - EECBS [16]

CBS Heuristics

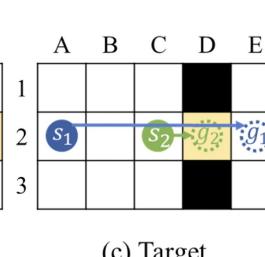
- Bypassing conflicts:
 - Do not split the node every time a conflict is found
 - When analyzing a conflict, modify the agents' path
 - If the cost of the new solution is the same as before and the number of conflicts is reduced --> do not split, but substitute
- Prioritizing conflicts:
 - A conflict is cardinal iff by solving the cost of both child CT nodes increases
 - A conflict is semi-cardinal iff the cost of only one child increases
 - A conflict is non-cardinal iff neither children's cost increased
- Symmetry reasoning



(a) Rectangle



(b) Corridor



(c) Target

Enhanced CBS (ECBS)

- Instead of using A* uses Focal search [17]
 - Bounded suboptimal algorithm
 - Uses OPEN and FOCAL sets
 - FOCAL contains all those nodes that have a weights suboptimal cost
 - The values in FOCAL are sorted using a function to estimate the cost-to-go
- ECBS implements focal search both for the low-level search and the high-level search:
 - The low-level is not sped up --> given an agent and a CT node, it returns
 - the path that minimizes the number of conflicts with other agents
 - the cost of the shortest path
 - The high-level search gets the costs of the shortest paths and can use focal search to speed up the analyses of the tree

Explicit Estimation Search

- Two main problems with ECBS high-level search:
 - It considers only the cost-to-go --> solution cost may be greater than the sub-optimality bound
 - At each time, the number of CT nodes with a similar cost is large --> FOCAL is rarely emptied
- The world is full of heuristic searches!
- Explicit Estimation Search (EES) [18] is a bounded-suboptimal search algorithm --> uses one more heuristic to overcome said focal behavior

Explicit Estimation Search (EES)

- EES is a bounded-suboptimal search algorithm --> uses one more heuristic to overcome said focal behavior
- It uses \hat{h} and \hat{d} to estimate the cost-to-go and the distance-to-go
- It keeps track of:
 - $best_f$, the node minimizing $f(n) = g(n) + h(n)$ from the FOCAL list
 - $best_{\hat{f}}$, the lowest predicted solution cost
 - $best_{\hat{d}}$, the node between the w admissible ones that appears closer to the target
- The node to explore is chosen based on
 1. $\hat{f}(best_{\hat{d}}) \leq w \cdot f(best_f) \rightarrow best_{\hat{d}}$ --> chose the node nearest to the goal
 2. $\hat{f}(best_{\hat{f}}) \leq w \cdot f(best_f) \rightarrow best_{\hat{f}}$ --> chose the node with the best path
 3. $best_f \rightarrow$ trust A*

Explicit Estimation CBS (EECBS)

- EECBS improves on ECBS by using EES on the *high-level* search
- It maintains 3 lists of CT nodes:
 - CLEANUP: regular list of A* sorted by the lower bound
 - OPEN: regular list of A* sorted by a *potentially inadmissible* function
 - FOCAL: nodes with cost bounded by w sorted by the distance-to-go
- The choice of the node to expand is similar to EES:
 - $\text{cost}(\text{best}_{h_c}) \leq w \cdot \text{lb}(\text{best}_{lb}) \rightarrow \text{best}_{h_c}$
 - $\text{cost}(\text{best}_{\hat{f}}) \leq w \cdot \text{lb}(\text{best}_{lb}) \rightarrow \text{best}_{\hat{f}}$
 - best_{lb}

Anytime Solvers

- We have seen that:
 - optimal algorithms do not scale well for the problem, but
 - sub-optimal algorithms may return a solution that is too inadequate
- Here come anytime solvers!
- The idea is:
 - return a sub-optimal solution in the shortest amount of time possible;
 - refine said solution in the remaining time available
- Many algorithms focus on intersections --> nodes with two or more neighbors
- We will see:
 - MAPF-LNS [19]
 - X* [21]

MAPF-LNS

- The algorithm uses Large-Neighborhood Search (LNS) [20]
 - The idea is to take a solution, remove an area, consider what remains as good and replan only on the sub-area which is a sub-problem of the initial
- It starts by finding an initial sub-optimal solution with either
 - EECBS, PP, or heuristics on PP
- The important aspect is how to extract a neighborhood
 - Agent-based
 - Map-based
 - Random-based

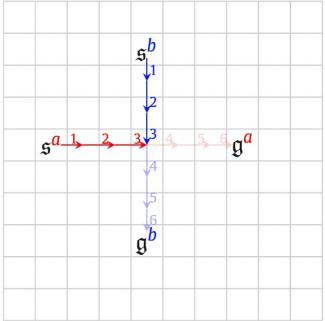
MAPF-LNS

- Agent-based neighborhood:
 1. Extract those agents that are not following the shortest path they could
 2. For each, compute a shorter random path
 3. Find agents that are colliding
 4. Replan groups of agents
- Map-based neighborhood:
 1. Identify the intersections --> higher probability of collision
 2. Identify agents moving through intersection, or in the area
 3. Change order in which agents pass through intersection
- Random neighborhood:
 1. Randomly choose N agents to replan for
 2. Replan

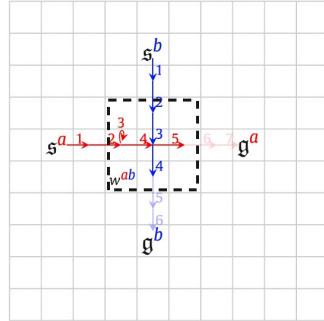
X*

- They introduce a concept called *window*
 - Identify agents and states around a conflict and repair the conflict
 - Each window has a successor, which basically is a larger window --> windows can grow
 - Two windows can also be merged together
- How does it work?
 1. Plan each agent individually
 2. Then starting by time t_0 it looks for conflicts
 3. For each conflict it creates a window and tries to solve it locally to the window
 4. The fixes are optimal within the window
- By enlarging windows and merging them, it can produce an optimal solution
- By preventing changes to following windows, it can produce a sub-optimal solution

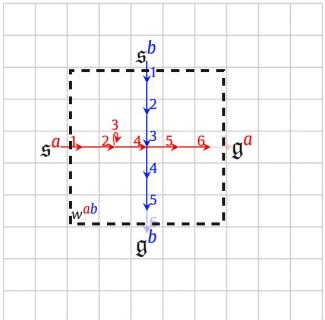
X^* – Example



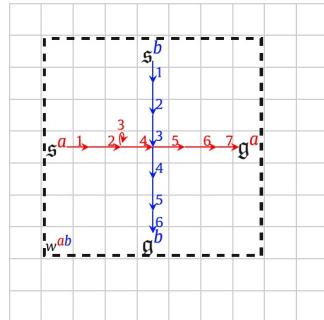
(a) Individually planned paths for each agent from s to g are used to form a global path. An agent-agent collision occurs in the path between a and b at $t = 3$.



(b) Collision between a and b is repaired by jointly planning inside w^{ab} . The global path is now guaranteed to be valid, but not guaranteed to be optimal.



(c) w^{ab} is grown and a new repair is generated for a and b . The window does not yet encapsulate the search from s^{ab} and g^{ab} , so the repaired global path is not yet guaranteed to be optimal.



(d) w^{ab} is grown and a new repair is generated. The repair search is from s^{ab} to g^{ab} and unimpeded by w^{ab} , thus allowing w^{ab} to be removed and the global path returned as optimal.

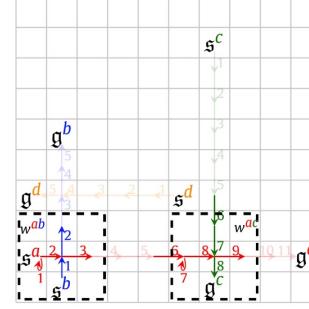
X*- Merging



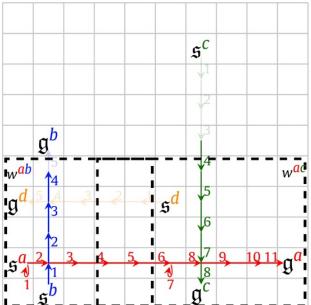
(a) Individually planned paths for each agent from s to g are used to form a global path. An agent-agent collision occurs between a and b at $t = 1$.



(b) Collision between a and b is repaired by jointly planning inside w^{ab} . The repair creates a collision between a and c at $t = 7$.



(c) Collision between a and c is repaired by jointly planning inside w^{ac} . No collisions exist, thus producing a valid global path.



(d) All windows are grown in order to improve repair quality.

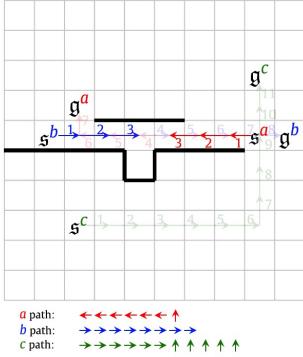


(e) As they overlap in agent set and states, w^{ab} and w^{ac} are merged to form w^{abc} , and a new repair is generated and inserted into the global path.

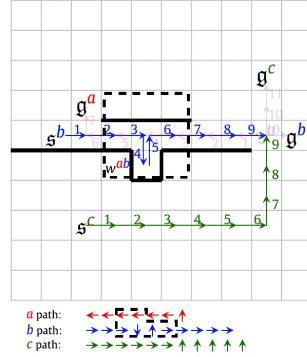


(f) w^{abc} is repeatedly grown and searched until the search of w^{abc} takes place from s^{abc} to g^{abc} unimpeded, thus allowing w^{abc} to be removed and the global path returned as optimal.

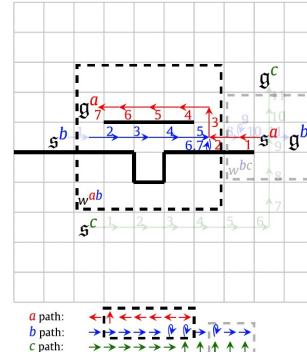
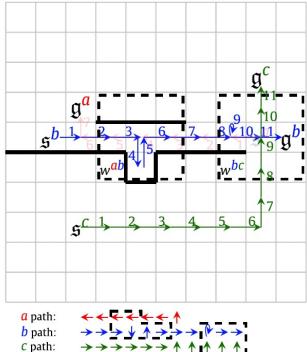
X^* – Example



(a) Individually planned paths for each agent from s to g are used to form a global path. An agent-agent collision occurs in the path between a and b between $t = 3$ and $t = 4$. Walls are depicted by thick black lines.



(b) Collision between a and b is repaired by jointly planning inside w^{ab} . b now side steps into the slot to allow a to pass, but this repair causes a collision with c at $t = 9$. The region of the paths repaired by w^{ab} is surrounded by dashed lines.

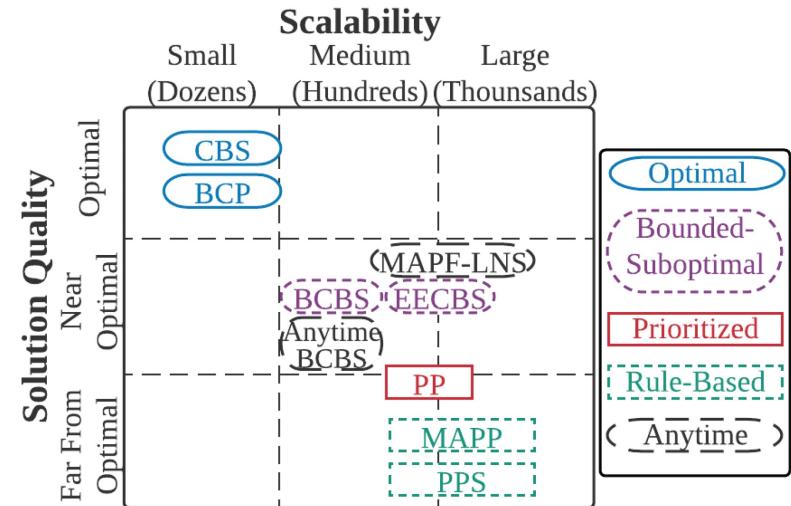


MAPF – Variants

- Different types of conflicts
- Different behaviors of agents when reaching goal
- MAPF with agents of different sizes
- Lifelong MAPF
- MAPF with non discrete time
- Multi-Objective MAPF
- MAPF in combination with task planning

Recap

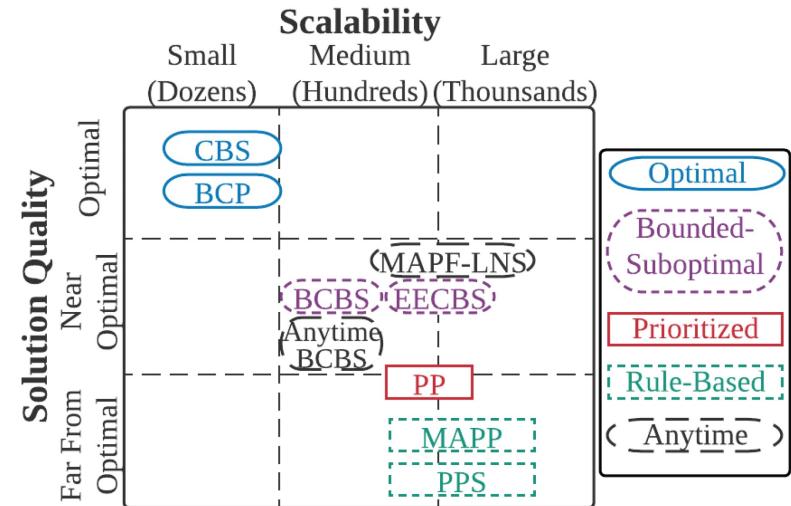
- Optimal:
 - Enhanced A* [2] (complete with correct heuristic)
- Complete and optimal:
 - CBS [6]
 - ICTS [7]
 - Constraint/logic programming [9]
- Fast:
 - PP [4]
- Suboptimal:
 - ϵ -ICTS and w -ICTS
 - CBS with improvements
 - ECBS
 - EECBS
- Anytime solvers:
 - MAPF-LNS
 - X^*



Recap

- Optimal:
 - Enhanced A* [2] (complete with correct heuristic)
- Complete and optimal:
 - CBS [6]
 - ICTS [7]
 - Constraint/logic programming [9]
- Fast:
 - PP [4]
- Suboptimal:
 - ϵ -ICTS and w -ICTS
 - CBS with improvements
 - ECBS
 - EECBS
- Anytime solvers:
 - MAPF-LNS
 - X^*

[MAPF Bible](#)



References

- [1] Stern, Roni, et al. "Multi-agent pathfinding: Definitions, variants, and benchmarks." *Proceedings of the International Symposium on Combinatorial Search*. Vol. 10. No. 1. 2019.
- [2] <https://www.leagueofrobotrunners.org/>
- [3] Standley, T. (2010). Finding Optimal Solutions to Cooperative Pathfinding Problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1), 173-178. <https://doi.org/10.1609/aaai.v24i1.7564>
- [4] Silver, D. (2021). Cooperative Pathfinding. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 1(1), 117-122. <https://doi.org/10.1609/aiide.v1i1.18726>
- [5] Ma, H., Harabor, D., Stuckey, P. J., Li, J., & Koenig, S. (2019). Searching with Consistent Prioritization for Multi-Agent Path Finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01), 7643-7650. <https://doi.org/10.1609/aaai.v33i01.33017643>
- [6] Sharon, Guni, et al. "Conflict-based search for optimal multi-agent pathfinding." *Artificial Intelligence* 219 (2015): 40-66.
- [7] Sharon, Guni, et al. "The increasing cost tree search for optimal multi-agent pathfinding." *Artificial intelligence* 195 (2013): 470-495.
- [8] A. Srinivasan, T. Ham, S. Malik and R. K. Brayton, "Algorithms for discrete function manipulation," *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, Santa Clara, CA, USA, 1990, pp. 92-95, doi: 10.1109/ICCAD.1990.129849.
- [9] R. Barták, N. -F. Zhou, R. Stern, E. Boyarski and P. Surynek, "Modeling and Solving the Multi-agent Pathfinding Problem in Picat," *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, Boston, MA, USA, 2017, pp. 959-966, doi: 10.1109/ICTAI.2017.00147.
- [10] Yu, J., & LaValle, S. (2013). Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1), 1443-1449. <https://doi.org/10.1609/aaai.v27i1.8541>
- [11] Walker, Thayne T., Nathan R. Sturtevant, and Ariel Felner. "Extended Increasing Cost Tree Search for Non-Unit Cost Domains." *IJCAI*. 2018.

References

- [12] Boyrasky, Eli, et al. "Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding." *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 25. 2015.
- [13] Boyarski, Eli, et al. "Icbs: The improved conflict-based search algorithm for multi-agent pathfinding." *Proceedings of the International Symposium on Combinatorial Search*. Vol. 6. No. 1. 2015.
- [14] Li, Jiaoyang, et al. "New techniques for pairwise symmetry breaking in multi-agent path finding." *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020.
- [15] Barer, Max, et al. "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem." *Proceedings of the International Symposium on Combinatorial Search*. Vol. 5. No. 1. 2014.
- [16] Li, J., Ruml, W., & Koenig, S. (2021). EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14), 12353-12362. <https://doi.org/10.1609/aaai.v35i14.17466>
- [17] Pearl, Judea, and Jin H. Kim. "Studies in semi-admissible heuristics." *IEEE transactions on pattern analysis and machine intelligence* 4 (1982): 392-399.
- [18] Thayer, Jordan Tyler, and Wheeler Ruml. "Bounded suboptimal search: A direct approach using inadmissible estimates." *IJCAI*. Vol. 2011. 2011.
- [19] Li, Jiaoyang, et al. "Anytime multi-agent path finding via large neighborhood search." *International Joint Conference on Artificial Intelligence 2021*. Association for the Advancement of Artificial Intelligence (AAAI), 2021.
- [20] Shaw, Paul. "Using constraint programming and local search methods to solve vehicle routing problems." *International conference on principles and practice of constraint programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998
- [21] Vedder, Kyle, and Joydeep Biswas. "X*: Anytime multi-agent path finding for sparse domains using window-based iterative repairs." *Artificial Intelligence* 291 (2021): 103417.

