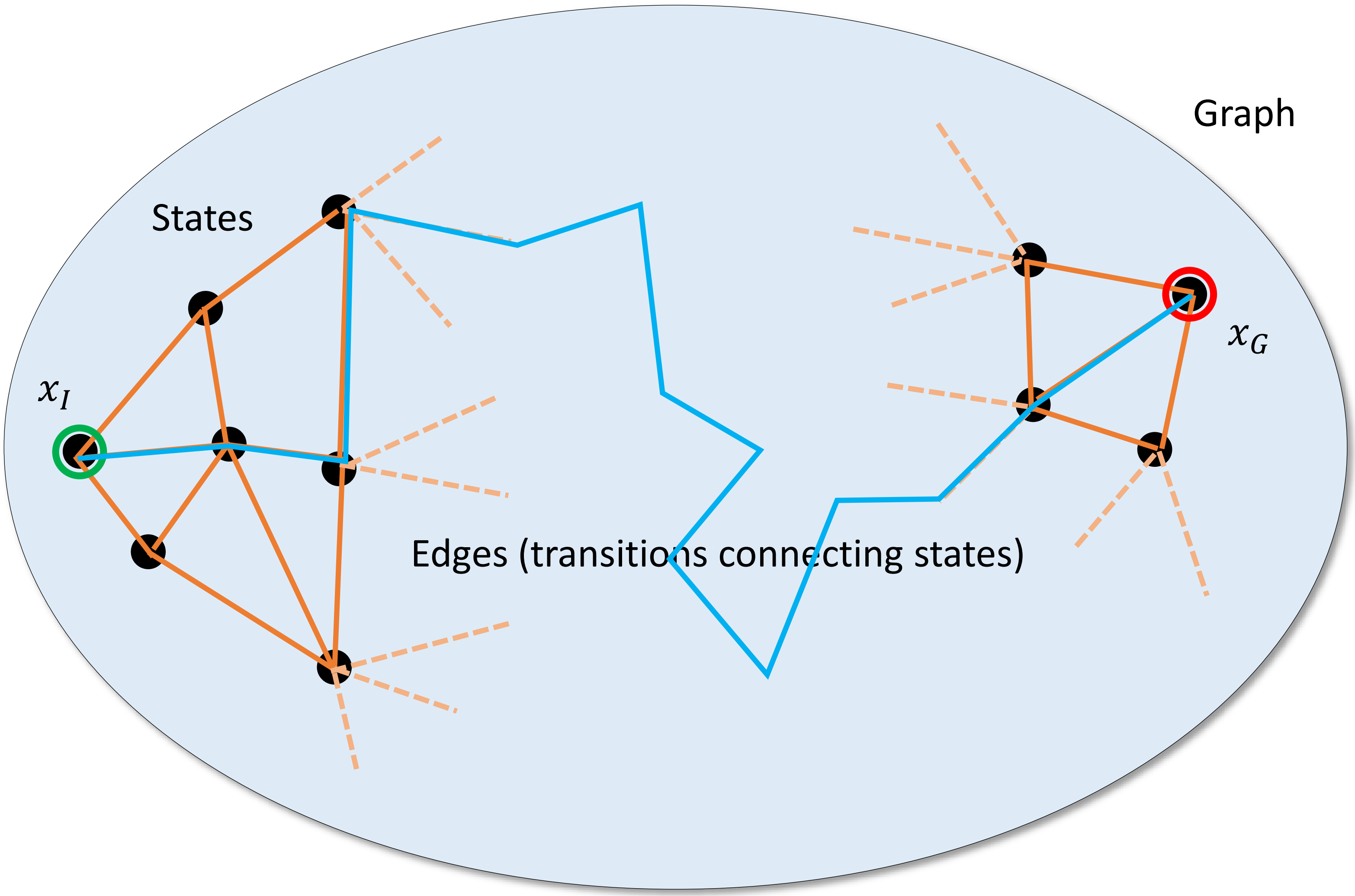
Review: A Generic Graph Search Algorithm

```
input: G = (V, E), x_I, x_G
AddToQueue(x_I, Queue);  // Add x_I to a queue of nodes to be expanded
while(!IsEmpty(Queue))
    x ← Front(Queue);                  // Retrieve the front of the queue
    if(x.expanded == true) continue;     // Do not expand a node twice
    x.expanded = true;                 // Mark x as expanded
    if(x == x_G) return solution;      // Return if goal is reached
    for each neighbor n_i of x // Add all neighbors of to the queue
        if(n_i.expanded == false) AddToQueue(n_i, Queue)
return failure;
```

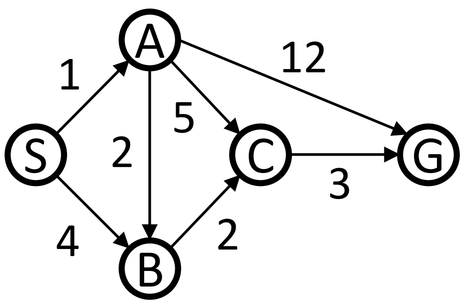
Different graph search algorithms (breadth first, depth-first, uniform-cost, …) differ at the function `AddToQueue`

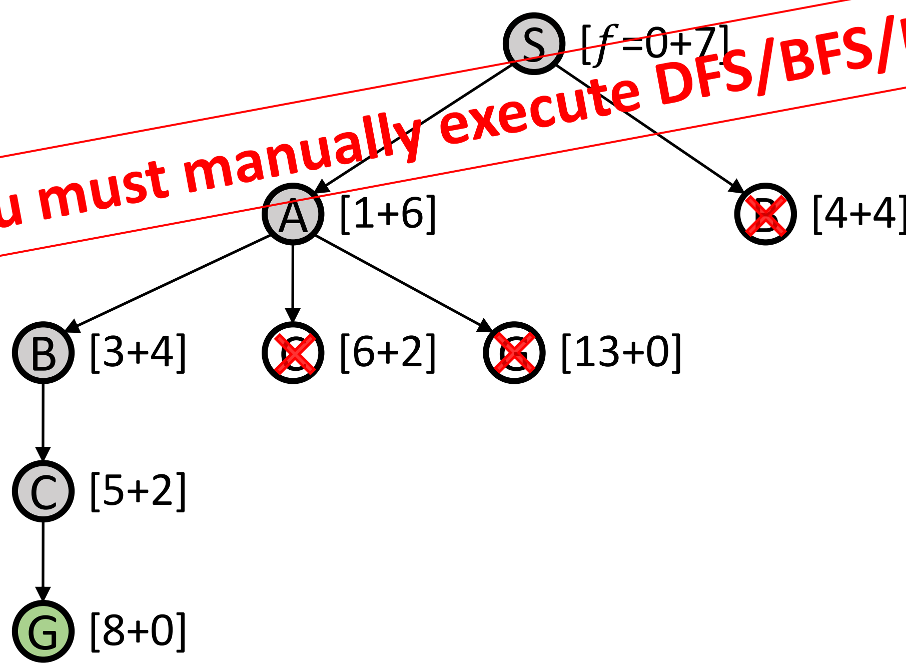
To retrieve the actual path, use **back pointers**

Review: Graph View of Search



Very Important: Manual Execution





For a good understanding, you must manually execute DFS/BFS/UC/A*!

State	$h(x)$
S	7
A	6
B	4
C	2
G	0

CS 460/560
Introduction to Computational Robotics
Fall 2019, Rutgers University

Lecture 11
Combinatorial Planning: In the Plane

Instructor: Jingjin Yu

Outline

Holonomic robots

Convex shapes, revisited

Combinatorial planning

- ⇨Computing intersection and visibility
- ⇨Vertical cell decomposition
- ⇨Shortest-path roadmaps
- ⇨Maximum clearance roadmaps

Holonomic Robots

Formally, a robot is **holonomic** if its constraints are of the form

$$f(x_1, \dots, x_n, t) = 0$$

Important: constraints cannot contain $\dot{x}_i, \ddot{x}_i$, and so on

Essentially, a **holonomic robot** has its controllable DOFs (i.e., number of independent directions along which it can move) equal to its configuration space dimension

Examples

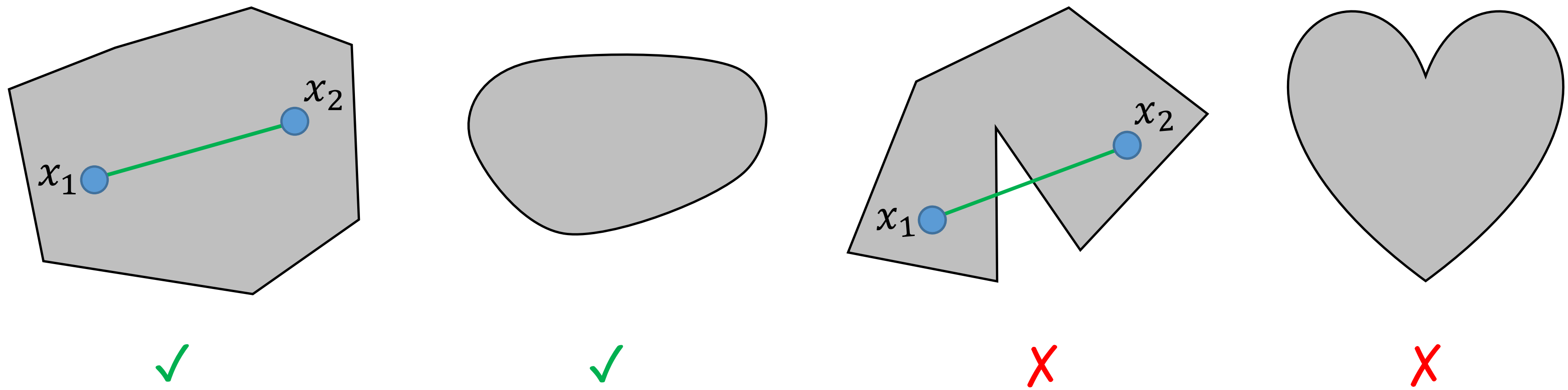
- ⇨ Holonomic robot: a vehicle with Mecanum wheels
  - ⇨ Configuration space: $SE(2)$, 3 dimensions: x, y, θ
  - ⇨ Such robots can move and rotate at the same time
- ⇨ Non-holonomic: a regular car
  - ⇨ Configuration space: $SE(2)$, 3 dimensions: x, y, θ
  - ⇨ Can only move along its heading direction: two controllable degrees

For this week, we work with holonomic robots unless otherwise stated

Image source: https://en.wikipedia.org/wiki/Holonomic_(robotics)

Convex Shapes, Revisited

Recall: in a Euclidean space, a set X is **convex** if given any $x_1, x_2 \in X$, all points on the straight-line segment $x_1 x_2$ belong to X.



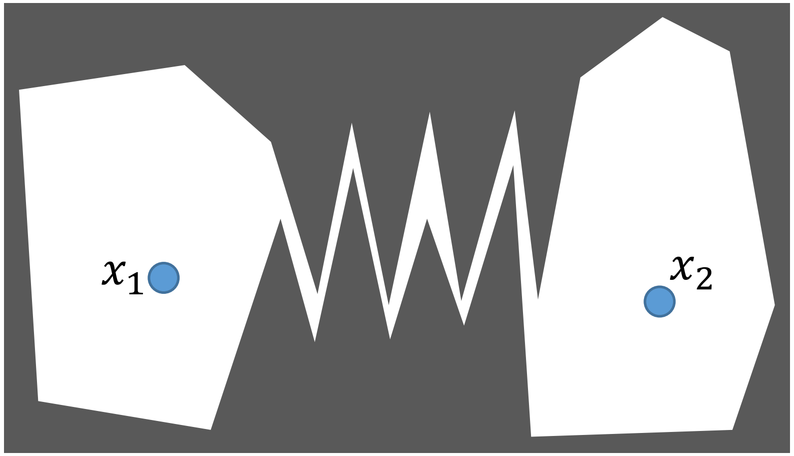
For holonomic robots, planning inside a convex shape is easy

⇨Why?

⇨Simply connect the two points!

⇨Not true for general robots (e.g., cars)

It's not as easy in non-convex shapes



Remember: we do not have a "God's view" in computer!

Combinatorial Planning

Recall that the C-space abstraction “shrinks” the robot into a point

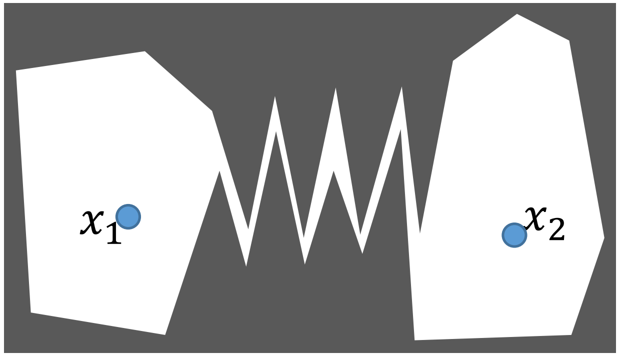
With C_{free} computed and the robot shrunk into a point, we can attempt path planning from x_I to x_G

⇨Now we are looking for a **simple path** connecting x_I and x_G

⇨No need to consider the geometry of the robot anymore!

Challenge: the configuration space may be **non-convex**

⇨So the path is not a straight line in C_{free}



Classical (2D) methods

⇨Cell decomposition

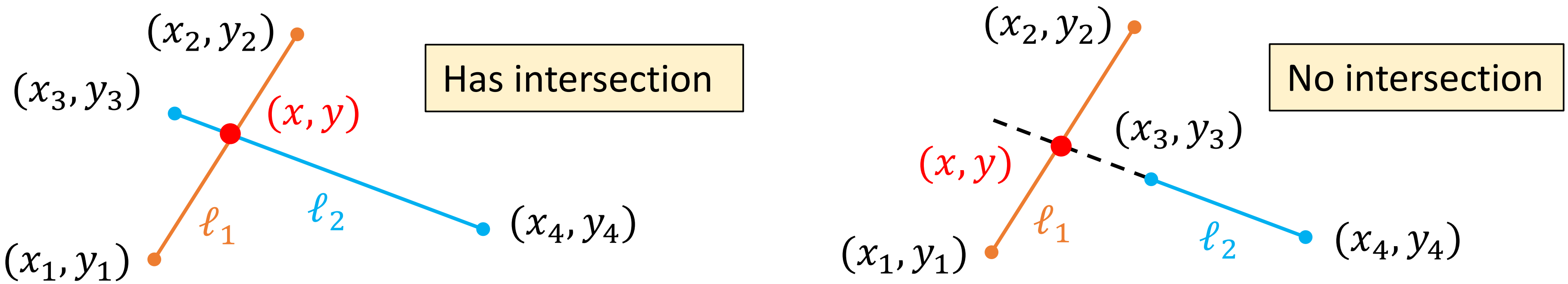
⇨Shortest-path roadmaps

⇨Maximum clearance roadmaps

Finding Intersection and Checking Visibility

These methods need to **find intersections** and **check visibility**

Finding intersection of two line segments

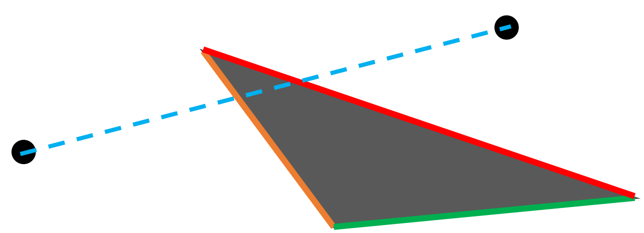


⇨Equations: $\ell_1\colon a_1x + b_1y + c_1 = 0$, $\ell_1\colon a_2x + b_2y + c_2 = 0$

⇨Two equations, two unknowns x, y, can solve for (x, y)

⇨Check whether (x, y) belongs to both ℓ_1 and ℓ_2

Checking visibility is similar



Cell Decomposition Methods

Cell decomposition breaks down C_{free} into simpler pieces that allow graph search to be carried out

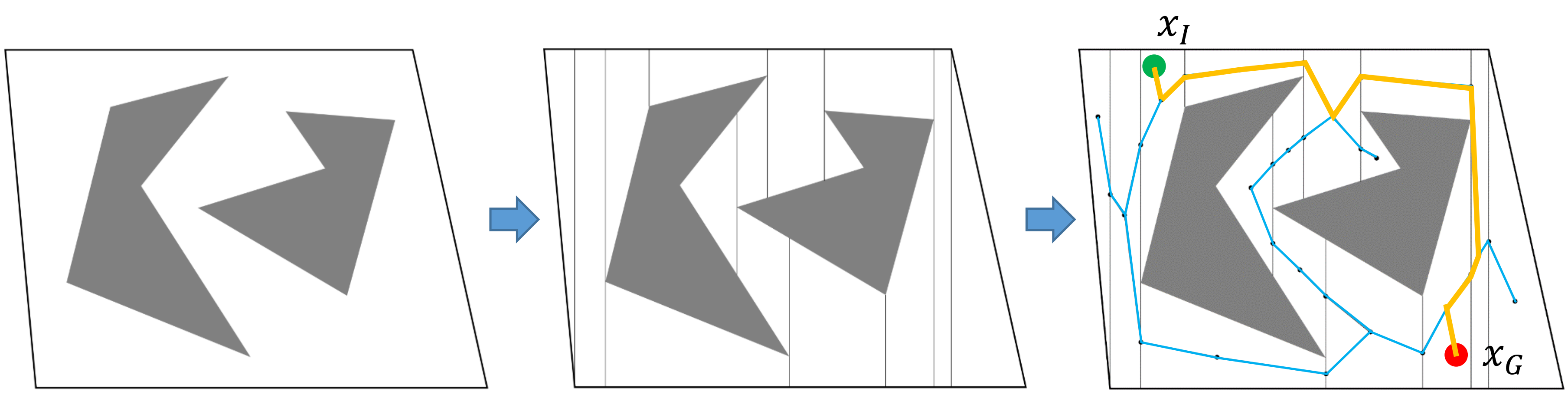
Example: vertical cell decomposition

⇨Break the free space into **convex** pieces (cells)

⇨Build a roadmap (graph) connecting the cells

⇨Plug in x_I and x_G and connect to cell centers

⇨Search over the roadmap for a path

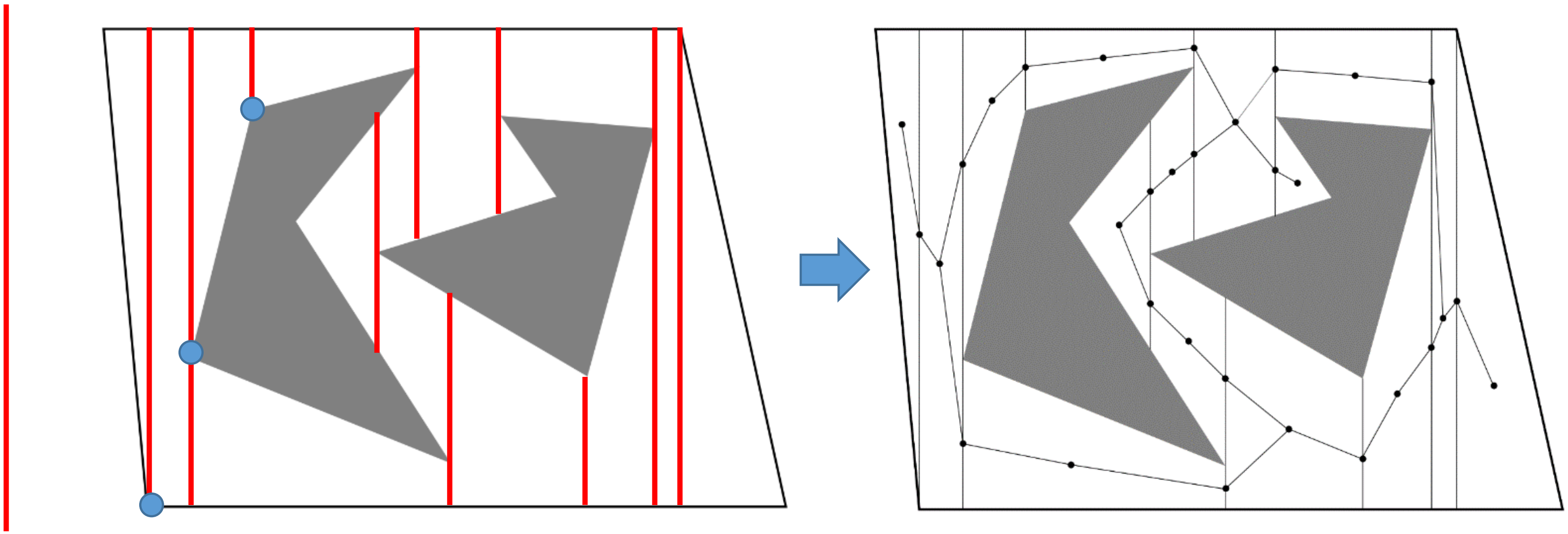


Many different types of cell decomposition methods

Image sources: http://planning.cs.uiuc.edu/ch6.pdf

Vertical Cell Decomposition in More Detail

Vertical cell decomposition uses a “vertical sweep-line” for doing the decomposition

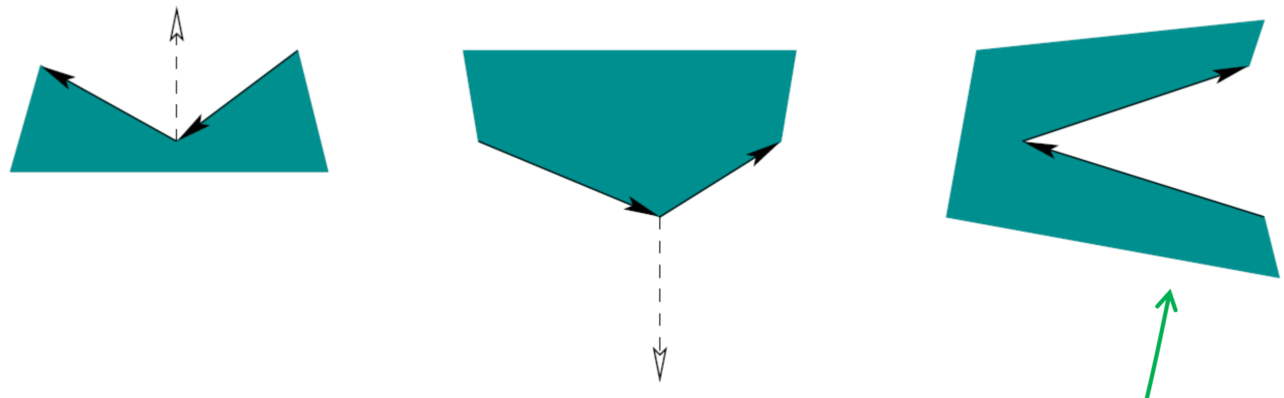


- ⇨Move the sweep-line from left to right
- ⇨For each vertex it touches, extend from the vertex along the line
- ⇨Stop when the line hits other features of the environment
- ⇨Compute the centers and boundary centers and connect them

Image sources: http://planning.cs.uiuc.edu/ch6.pdf

Vertical Cell Decomposition in More Detail

There are four types of vertices



The second type is actually not needed

⇨With it, we get trapezoids

⇨Otherwise, we get convex polygons which is still OK

The last type does not generate any vertical lines

Image sources: http://planning.cs.uiuc.edu/ch6.pdf

Vertical Cell Decomposition in More Detail

Algorithm details

- A simple algorithm
  - Sort the x coordinates of input points - $O(n \log n)$
  - Order the points from left to right
  - For each point, check whether the vertical scan line intersects each line segment of the environment, pick the closest intersection point(s) - $O(n)$
  - Overall computation time is $O(n^2)$
- Smarter algorithm
  - If we take more care to track the line segments in the environment we can cut down the computation time to $O(n \log n)$

The algorithm comes from **plane-sweep** or **line-sweep** algorithms from computational geometry

- Can be found in many computational geometry textbooks

Shortest-Path Roadmaps

Vertical cell decomposition is simple but is suboptimal – it does not provide shortest paths connecting x_I and x_G

Shortest-path roadmaps solve the problem (also known as the **reduced visibility graph**)

The black structure (including the isolated vertex) is the shortest-path roadmap

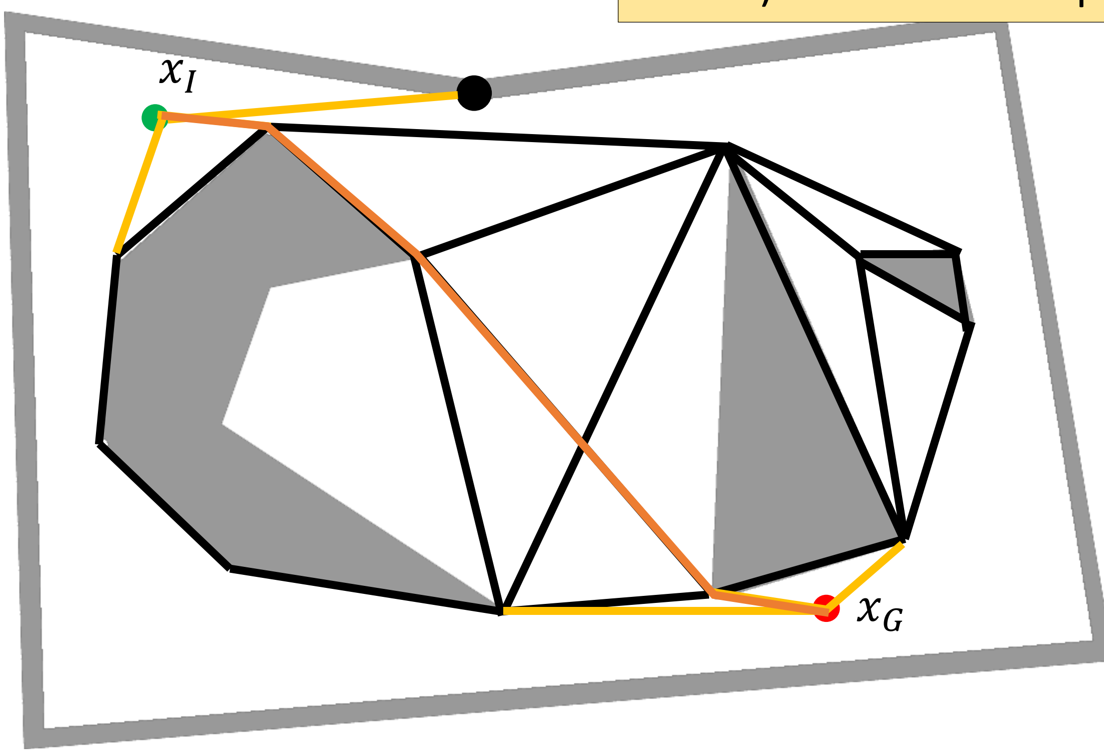
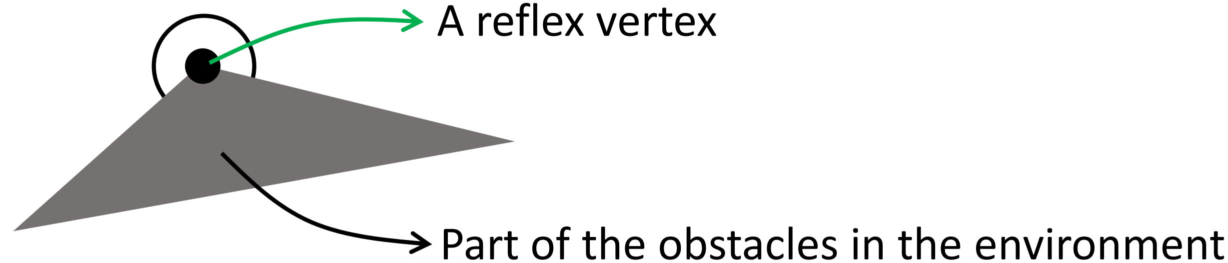


Image sources: http://planning.cs.uiuc.edu/ch6.pdf

Building Shortest-Path Roadmap

Reflex vertices

⇨ A **reflex vertex** is one where the angle through the vertex in the environment is greater than π

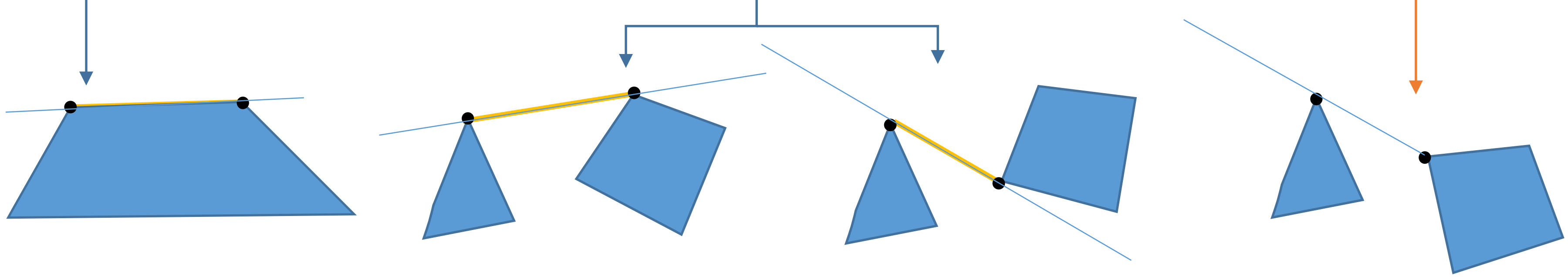


Building the roadmap

⇨ Add all reflex vertices to the roadmap

⇨ For two reflex vertices that are visible to each other, add an edge if

- ⇨ Consecutive reflex vertices on the same obstacle
- ⇨ The two vertices yield a **bi-tangent**
- ⇨ But not this types of setup



Search on the Shortest-Path Roadmaps

To search for a path from x_I and x_G on the roadmap

- ⇨Add x_I and x_G to the roadmap
- ⇨Connect x_I to visible roadmap vertices; same for x_G
- ⇨Searching for a shortest path through the connected roadmap containing x_I and x_G

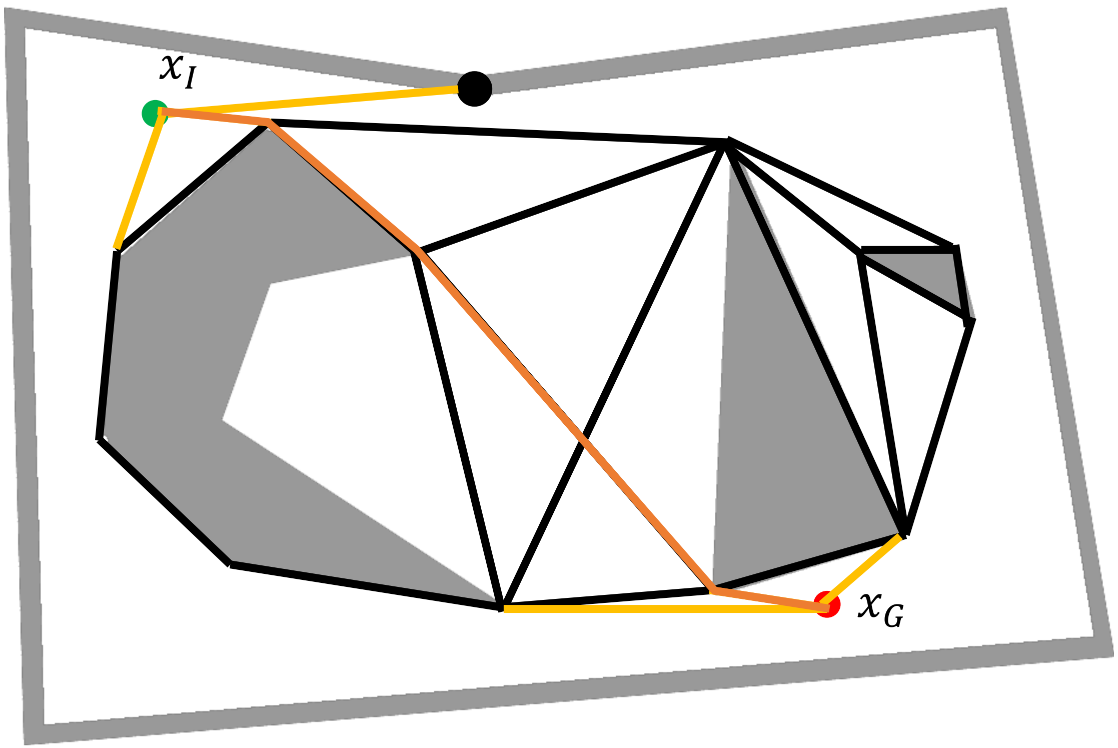
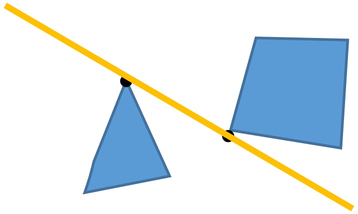


Image sources: http://planning.cs.uiuc.edu/ch6.pdf

Running Time for Roadmap Building



Naïve algorithm

- ⇨There are $O(n)$ reflex vertices so n^2 pairs to check for bi-tangents
- ⇨For each pair, check whether the pair is visible - $O(n)$
- ⇨This yields an $O(n^3)$ algorithm

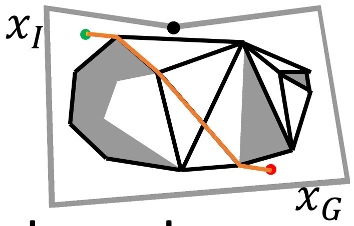
Radial sweep algorithm

- ⇨For a reflex vertex, compute all bi-tangents by sweeping from 0 to 2π radially
- ⇨This allows the computation of all bi-tangents from the vertex in $O(n \log n)$ time
- ⇨Total complexity is $O(n^2 \log n)$

Even faster algorithms

- ⇨Output sensitive algorithm $O(n \log n + m)$ where m is the number of edges in the output roadmap

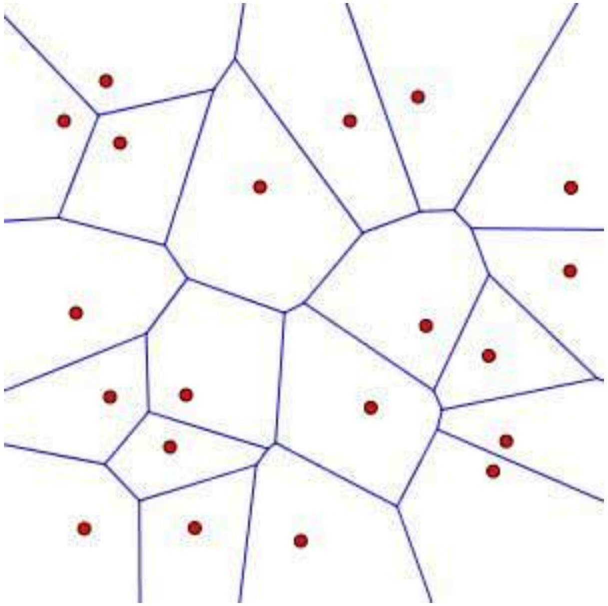
Maximum Clearance Roadmap



The shortest-path roadmap requires robots to go along obstacle boundaries, which can be unsafe

Maximum clearance roadmap does the opposite

⇨Based on the idea of Voronoi diagrams

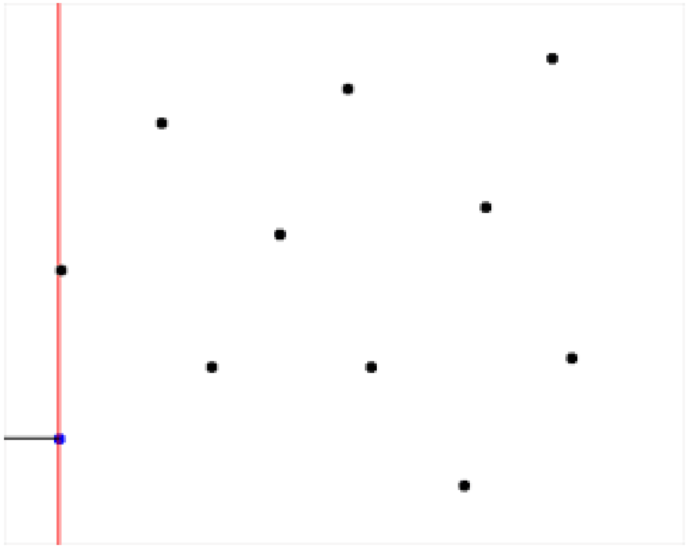


⇨In a Voronoi diagram, the lines are as far away from all points as possible

Computing the Voronoi Diagram

Voronoi diagram can be computed in the plane in $O(n \log n)$ time using Fortune's algorithm

- ⇨Also a sweep-line algorithm!
- ⇨Maintains a "beach front"



For higher dimensions, Bowyer-Watson computes a Delaunay triangulation which can be turned into a Voronoi diagram

Image source: https://en.wikipedia.org/wiki/File:Fortunes-algorithm.gif

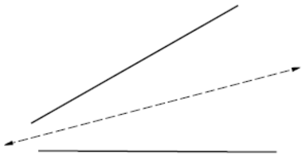
Building Maximum Clearance Roadmap

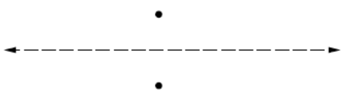
Instead of vertex-vertex interactions, now with edge interactions

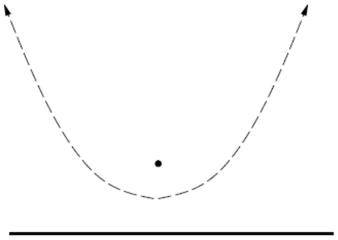
⇨Edge-edge

⇨Vertex-vertex

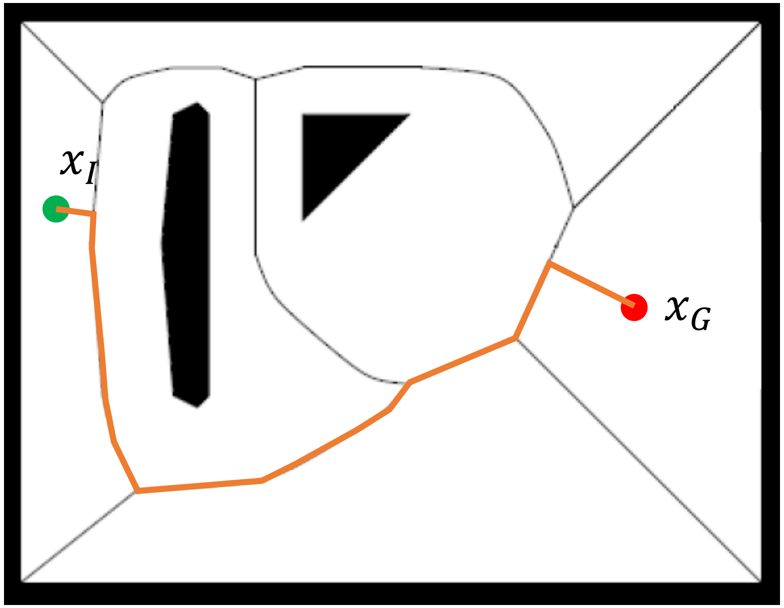
⇨Vertex-edge







Example



Running Time for Roadmap Building

Naïve algorithm

- ⇨For each pair of features, compute the line as shown above - n^2 of these
- ⇨For each pair of these lines, compute their intersections – $O(n^4)$

Using a better algorithm, can get the running time to $O(n \log n)$

- ⇨This can be done by modifying the sweep line algorithm