CS 3630

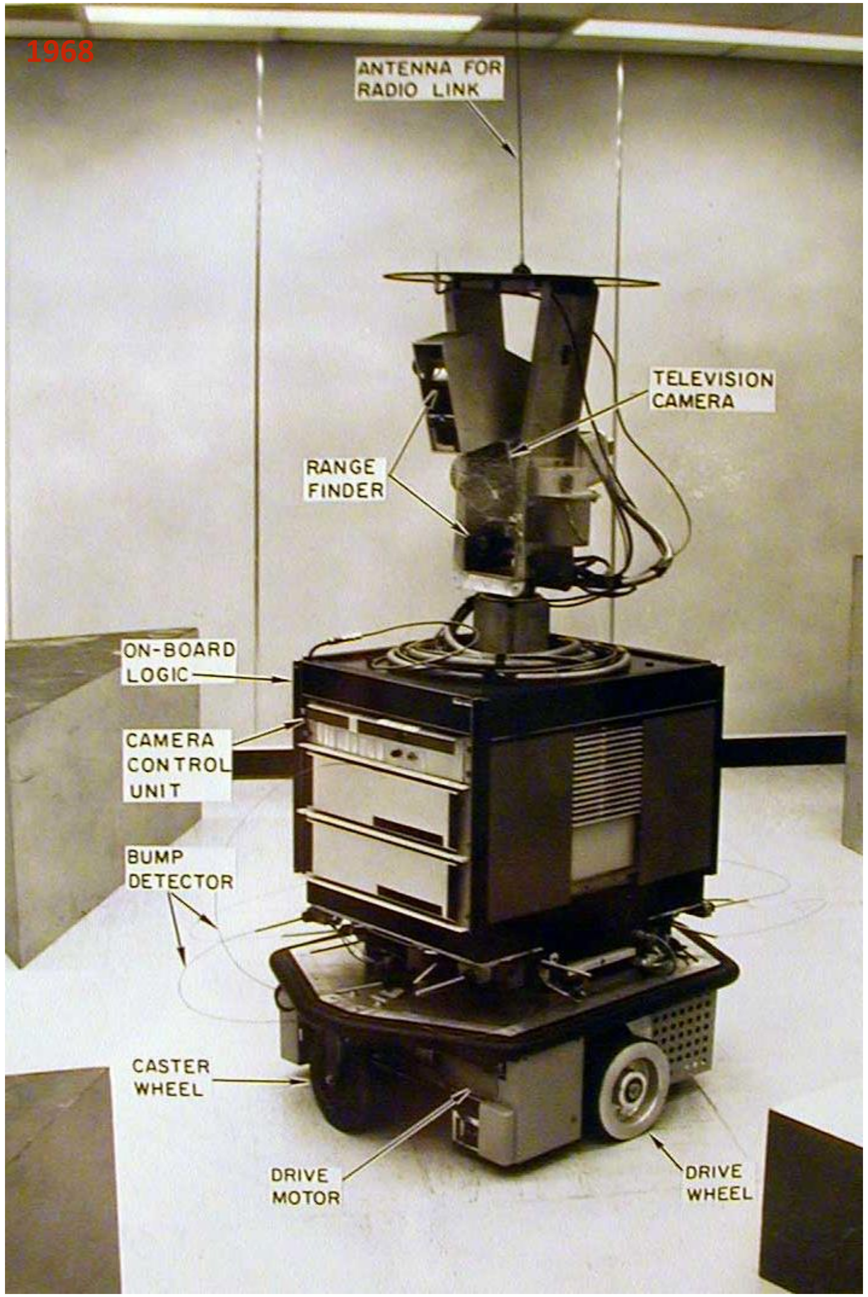
Differential Drive Robots

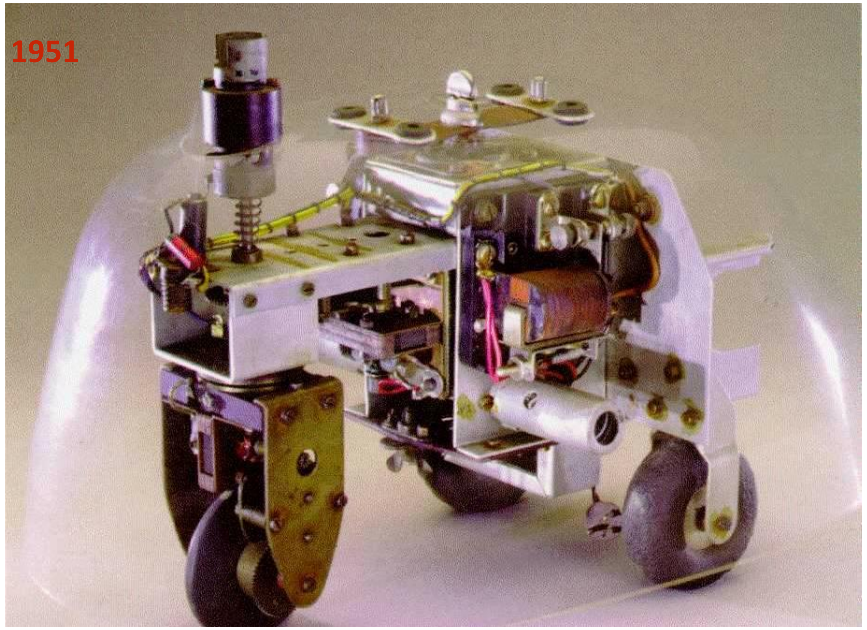
Mobile Robots

- There are many kinds of wheeled mobile robots.
- In this class, we primarily study *differential drive robots.*
- The Duckiebot is a differential drive robot.

Mobile Robot Kinematics

- Relationship between input commands (e.g., wheel velocity) and pose of the robot, not considering forces. *If the wheels turn at a certain rate, what is the resulting robot motion?*
- No direct way to measure pose (unless we sensorize the environment), but we can integrate velocity (odometry) to obtain a good estimate.





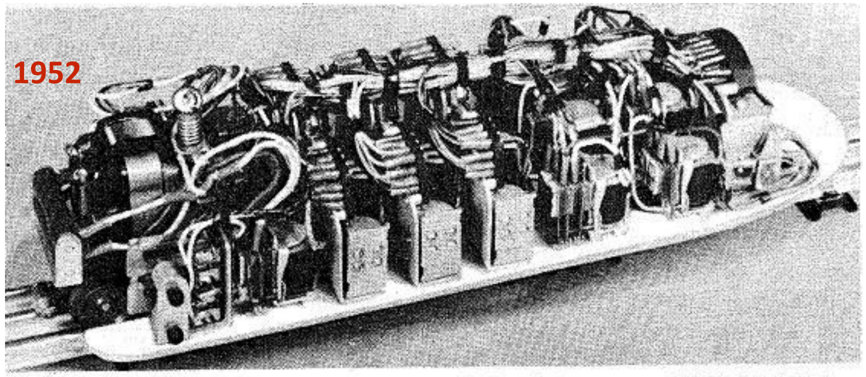
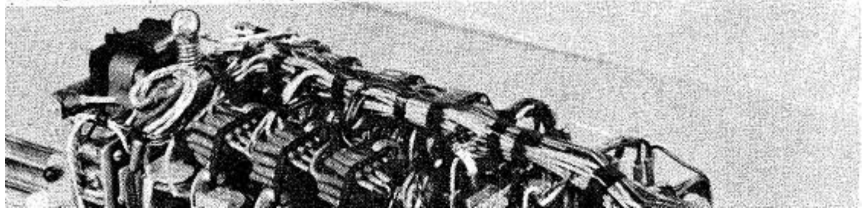
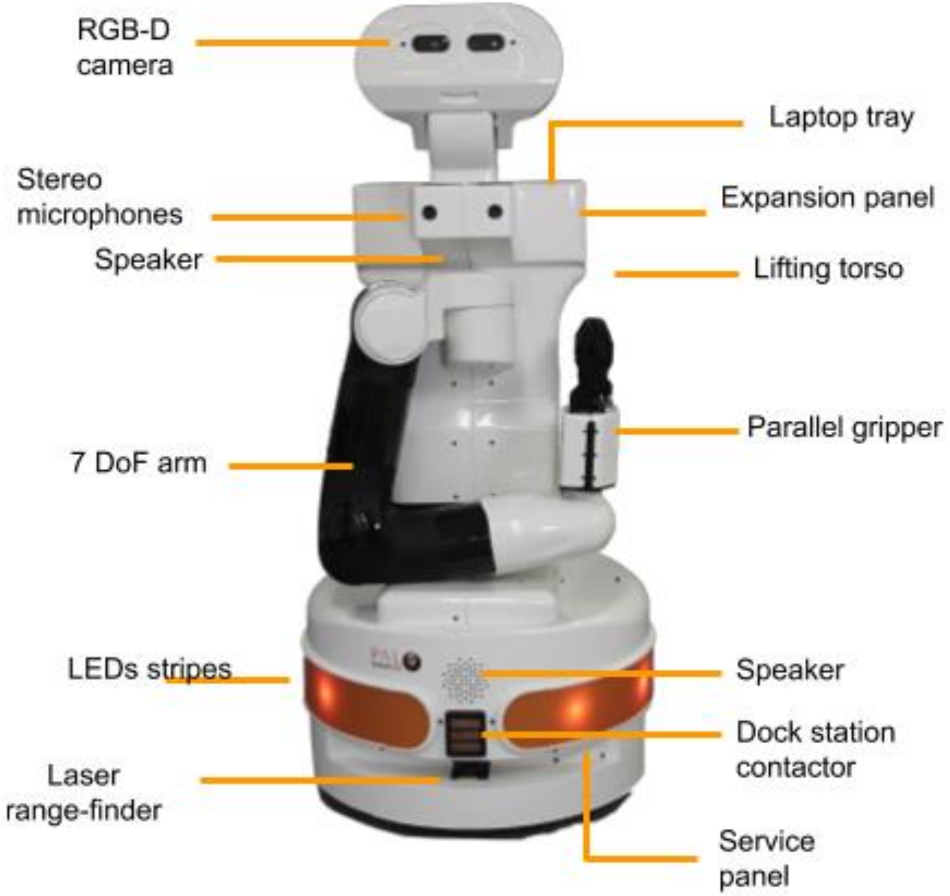


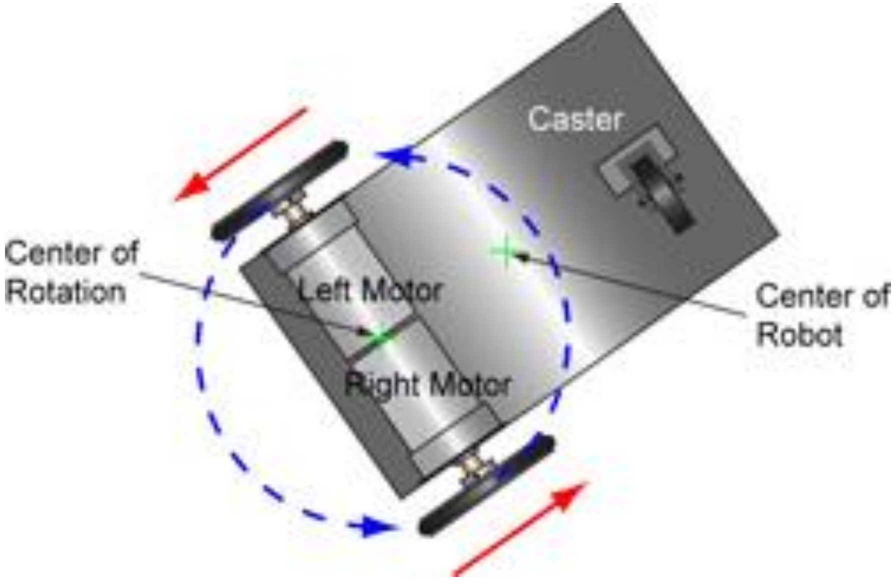
FIGURE I. THE MAZE SOLVING COMPUTER.

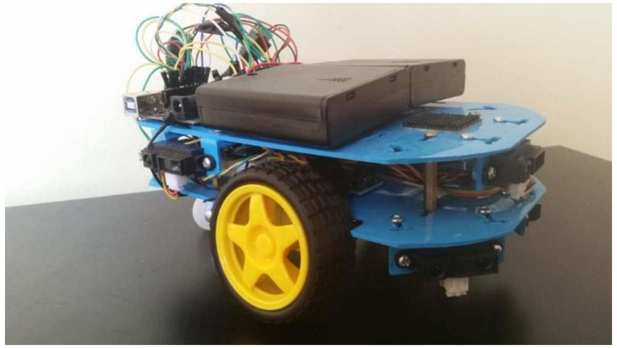


More Modern AGVs

Differential Drive Robots

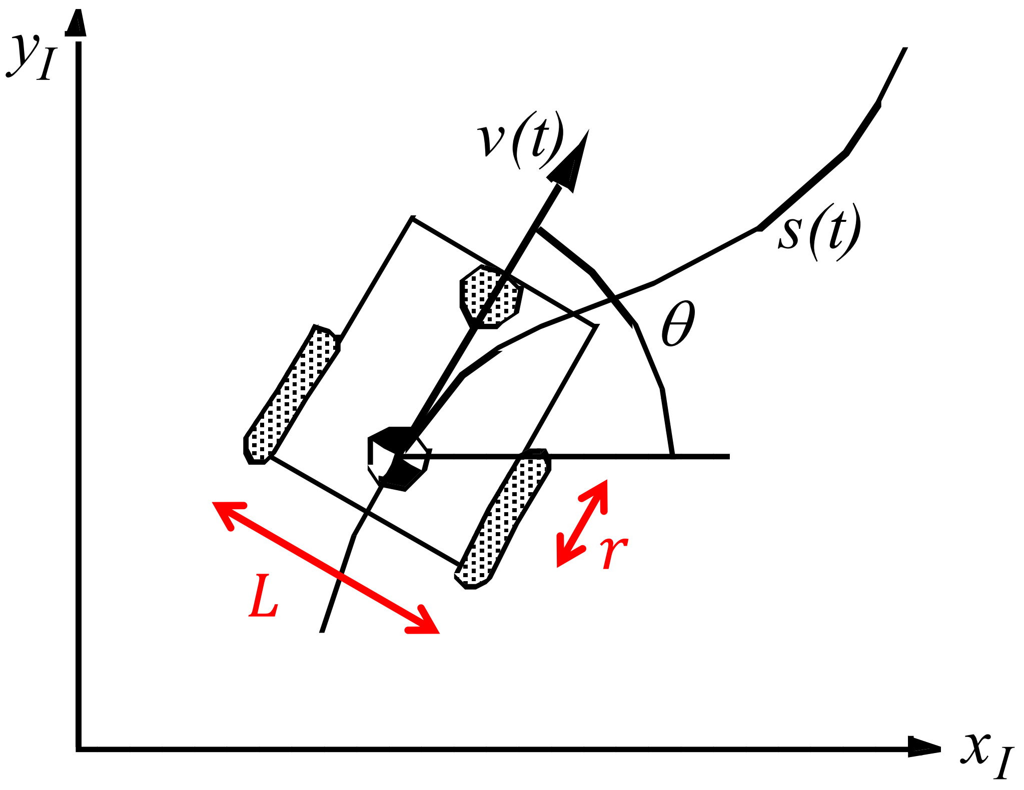






Two wheels with a common axis, and that can spin independently

Differential Drive Robots



The configuration of the robot can be specified by

$$q = (x, y, \theta)$$

At any moment in time, the instantaneous velocity of the robot is given by

$$v(t) = \begin{bmatrix} v_x \\ v_y \end{bmatrix}, \qquad \dot{\theta} = \omega$$

This robot cannot move instantaneously in the direction perpendicular to the forward velocity: $v_y = 0$

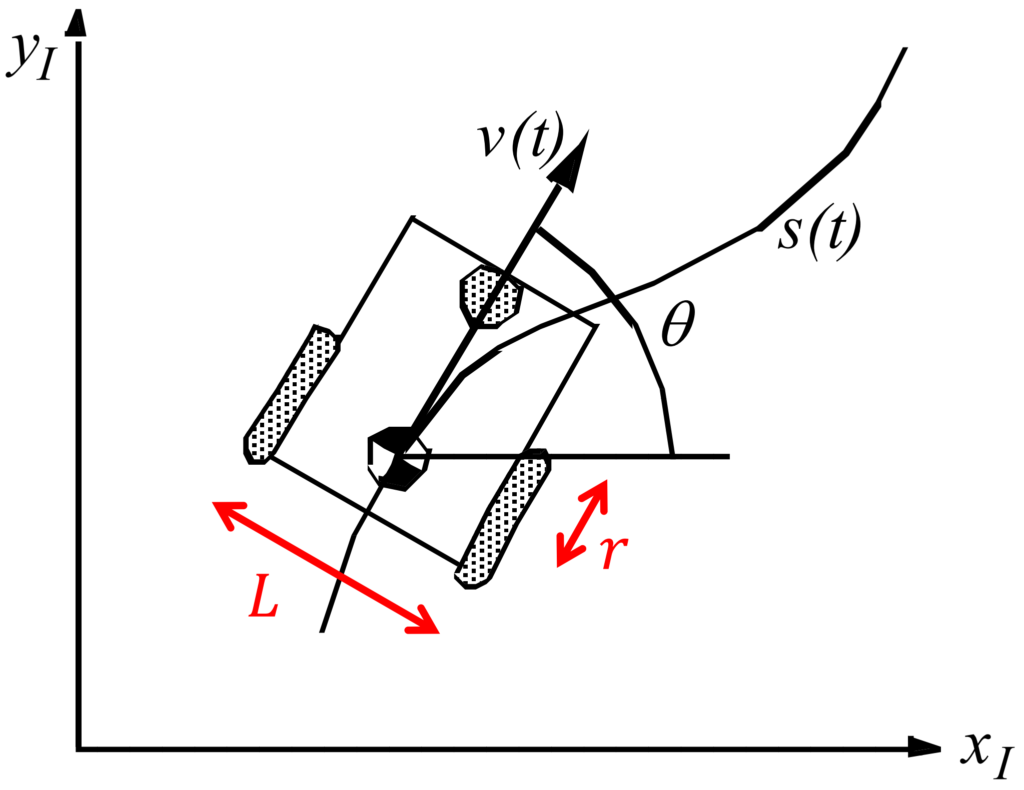
Wheel radius is r

Baseline distance between wheels is L

NOTE: These velocities are specified w.r.t. the robot's coordinate frame.

Differential Drive Robots

$\dot{\phi}$ = speed of wheel rotation



When both wheels turn with the same velocity and same direction, we have pure forward motion:

$$\dot{\phi}_R = \frac{v_x}{r}, \qquad \dot{\phi}_L = \frac{v_x}{r}$$

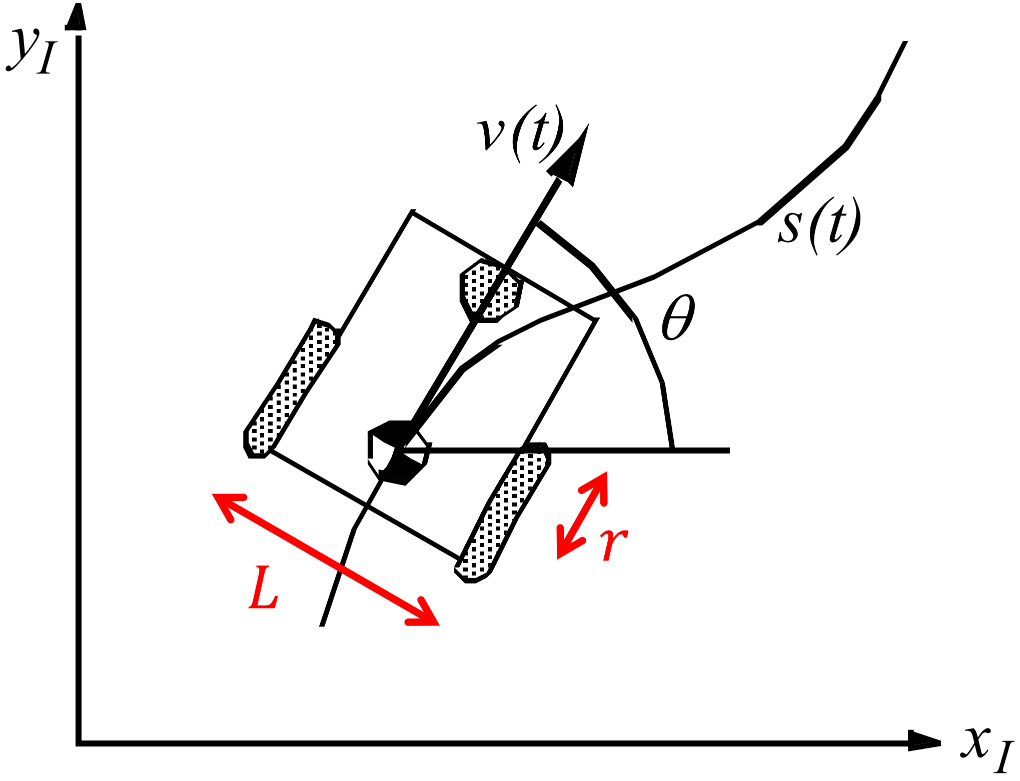
When the wheels turn in opposite directions with the same velocity, we have pure rotation:

$$\dot{\phi}_R = \frac{\omega L}{2r}, \qquad \dot{\phi}_L = -\frac{\omega L}{2r}$$

Combining the two (velocities are linear, so superposition applies) we obtain:

$$\dot{\phi}_R = \frac{\omega L}{2r} + \frac{v_x}{r}, \qquad \dot{\phi}_L = -\frac{\omega L}{2r} + \frac{v_x}{r}$$

Differential Drive Robots



We have equations that define wheel angular velocity in terms of linear and angular velocity of the robot:

$$\dot{\phi}_R = \frac{\omega L}{2r} + \frac{v_x}{r}, \qquad \dot{\phi}_L = -\frac{\omega L}{2r} + \frac{v_x}{r}$$

A bit of algebra gives the desired relationship between input (wheel velocity) and output (linear and angular velocity of the robot):

$$\frac{r}{2}(\dot{\phi}_R + \dot{\phi}_L) = v_x, \qquad \frac{r}{L}(\dot{\phi}_R - \dot{\phi}_L) = \omega = \dot{\theta}$$

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2}(\dot{\phi}_R + \dot{\phi}_L) \\ 0 \\ \frac{r}{L}(\dot{\phi}_R - \dot{\phi}_L) \end{bmatrix}$$

Motion relative to the world frame

We transform the robot velocity to world coordinates using our usual coordinate transformation:

$$v^0 = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} v_x \\ 0 \end{bmatrix} = \begin{bmatrix} v_x \cos\theta \\ v_x \sin\theta \end{bmatrix}$$

$$\begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{r}{2}(\dot{\phi}_R + \dot{\phi}_L) \\ 0 \\ \frac{r}{L}(\dot{\phi}_R - \dot{\phi}_L) \end{bmatrix}$$

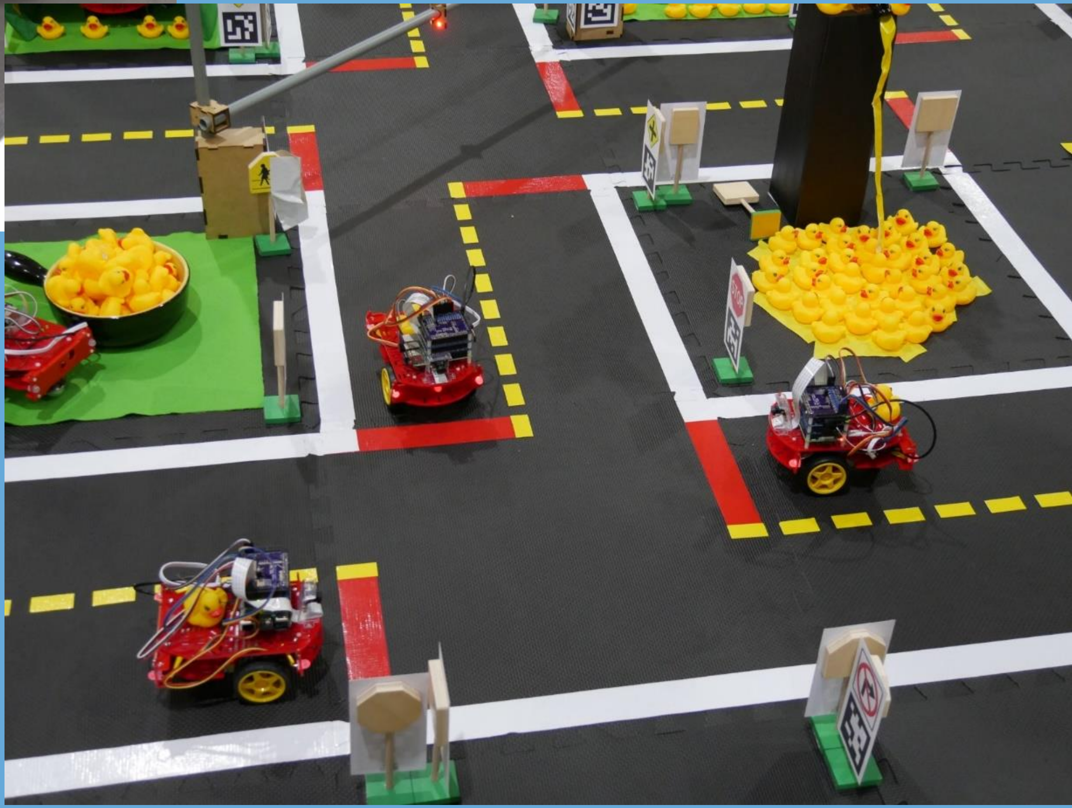
$$\dot{\theta} = \omega$$

We typically think of the robot as a device with linear and angular velocity input, rather than think about wheel RMPs.

We typically write the equations of motion as:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v_x \cos\theta \\ v_x \sin\theta \\ \omega \end{bmatrix} \quad \text{or as} \quad \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

CS 3630
Motion Planning in the Plane



With lots of slides and ideas from:

Howie Choset

Greg Hager

Zack Dodds

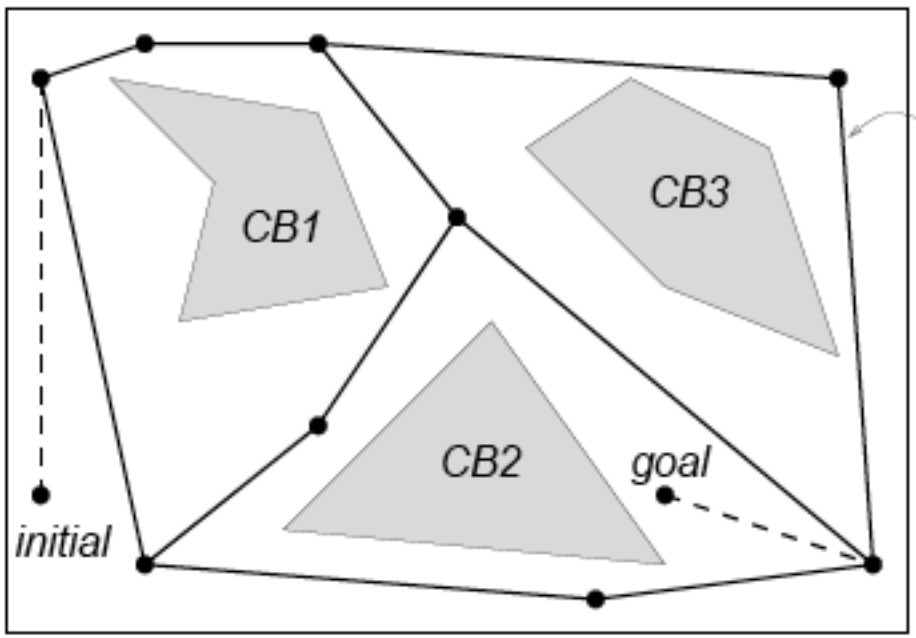
Nancy Amato

Mobile Robots

- In general, motion planning is intractable.
- For certain special cases, efficient algorithms exist.
- Mobile robots that move in the plane are much simpler than robot arms, mobile manipulators, humanoid robots, etc.
- The main simplifying property is that we can often treat path planning as a two-dimensional problem for a point moving in the plane, $x \in \Re^2$.
- Today --- path planning algorithms for such robots.

Roadmap methods

Capture the connectivity of the free space by a graph or network of paths.



Roadmaps

A roadmap, RM, is the union of one-dimensional curves such that for all x_{start} and x_{goal} that can be connected by a collision- free path:

- **Accessibility**: There is a collision-free path connecting x_{start} to some point $x_1 \in RM$.
- **Departability**: There is a collision-free path connecting x_{goal} to some point $x_2 \in RM$.
- **Connectivity**: There is a path in RM connecting x_1 and x_2.

If such a roadmap exists, then a free path from x_{start} to x_{goal}can be constructed from these three sub-paths, and the path planning problem can be reduced to finding the three sub-paths.

RoadMap Path Planning

1. Build the roadmap
   a) nodes are points in the free space or its boundary
   b) two nodes are connected by an edge if there is a free path between them

2. Connect start end goal points to the road map at point x_1 and x_2 , respectively

3. Find a path on the roadmap between x_1 and x_2

The result is a path from start to goal

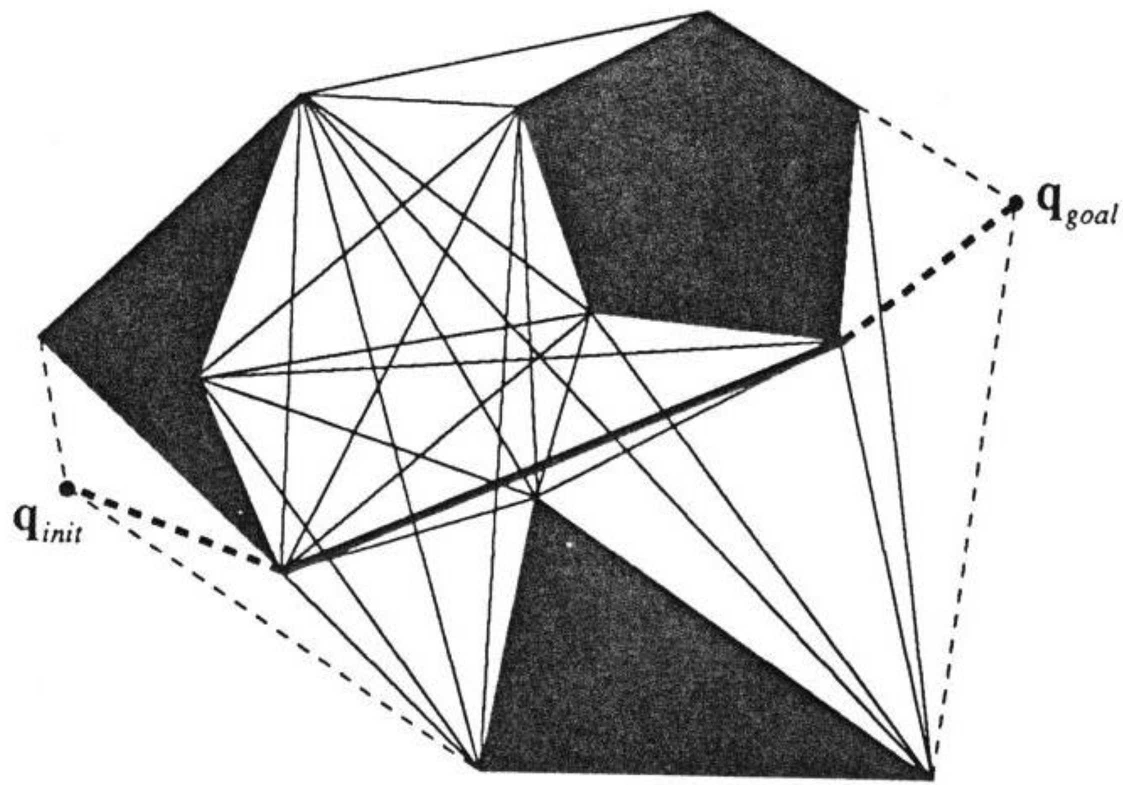
Shortest, But Possibly Dangerous Paths

The Visibility Graph

Visibility Graph methods

- Defined for polygonal obstacles
- Nodes correspond to vertices of obstacles
- Nodes are connected if
  - they are connected by an edge on an obstacle

  OR
  - the line segment joining them is in free space

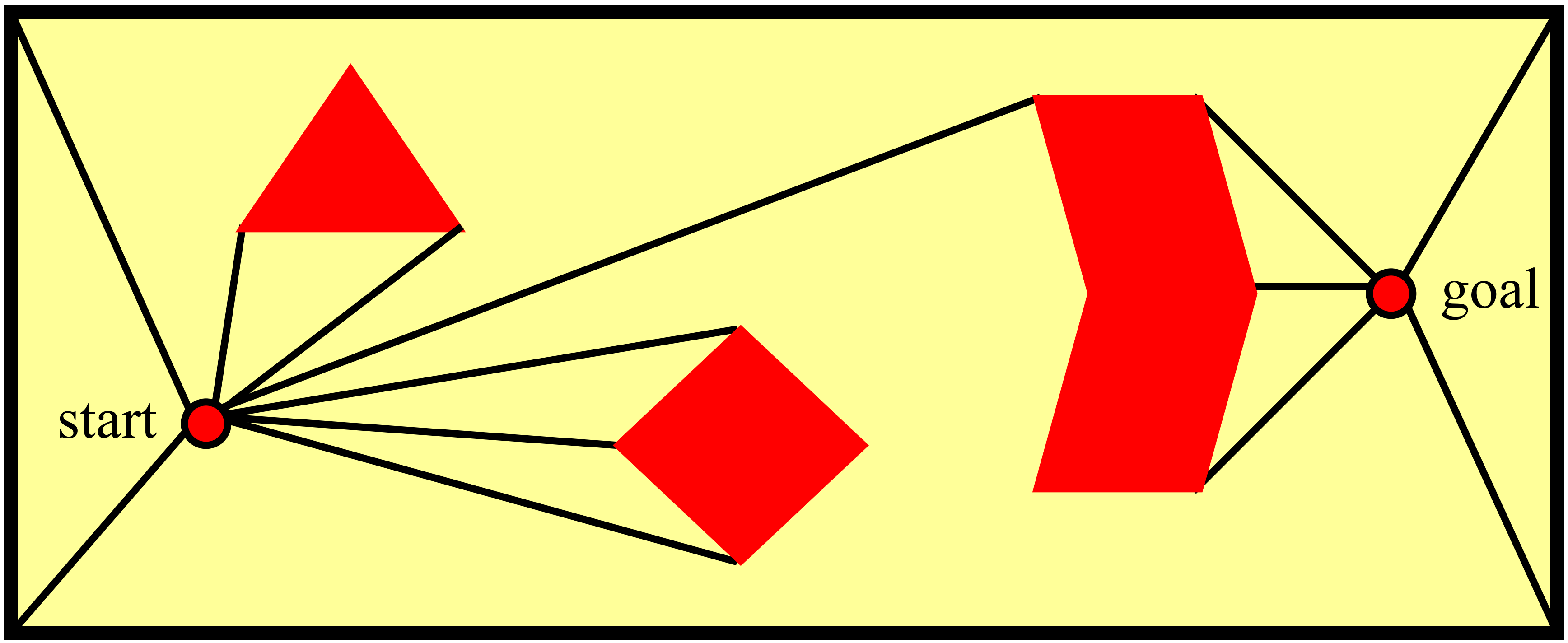


- If there is there a path, then the *shortest* path is in the visibility graph
- If we include the start and goal nodes, they are automatically connected
- Algorithms for constructing them can be efficient
  - $O(n^3)$ brute force (i.e., naïve)
  - $O(n^2 \log n)$ if clever

The Visibility Graph in Action (Part 1)

- First, draw lines of sight from the start and goal to all "visible" vertices and corners of the world.

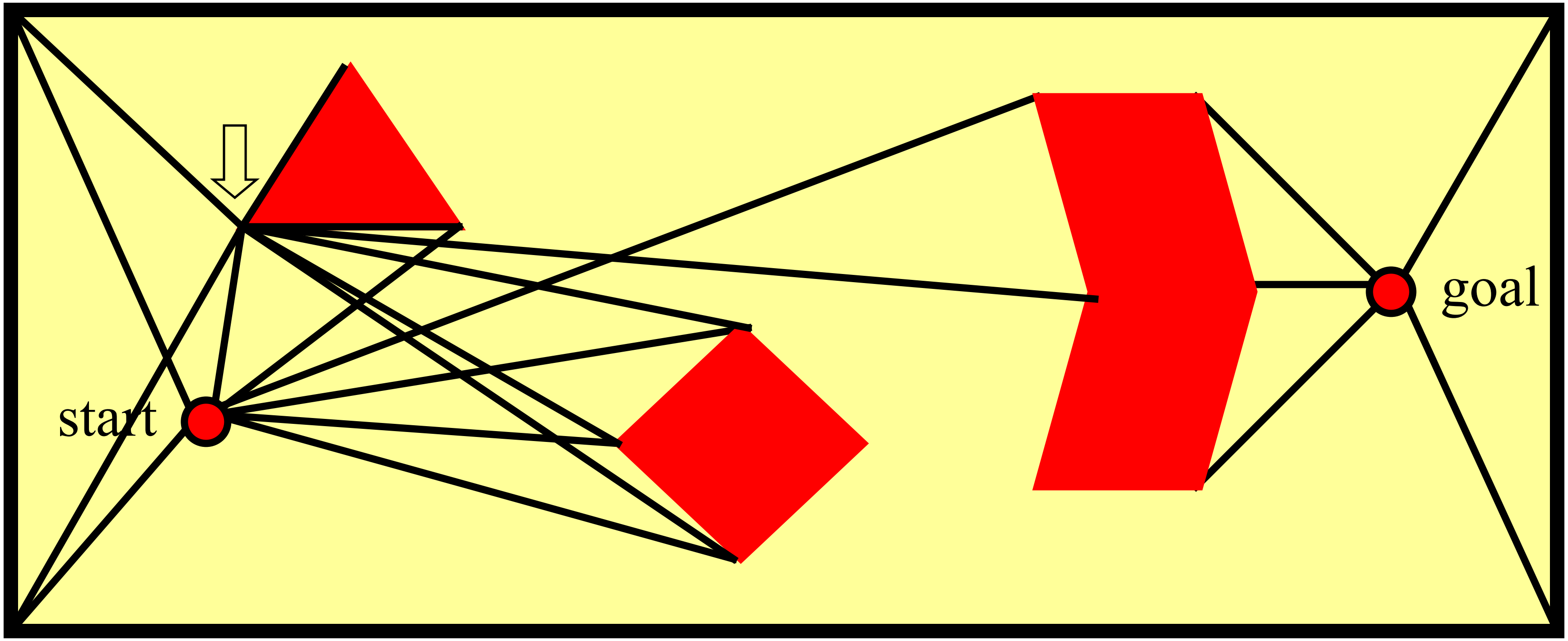
$$e_{ij} \neq \emptyset \iff sv_i + (1-s)v_j \in \mathrm{cl}(\mathcal{Q}_{\mathrm{free}}) \;\; \forall s \in (0,1)$$



The Visibility Graph in Action (Part 2)

- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.

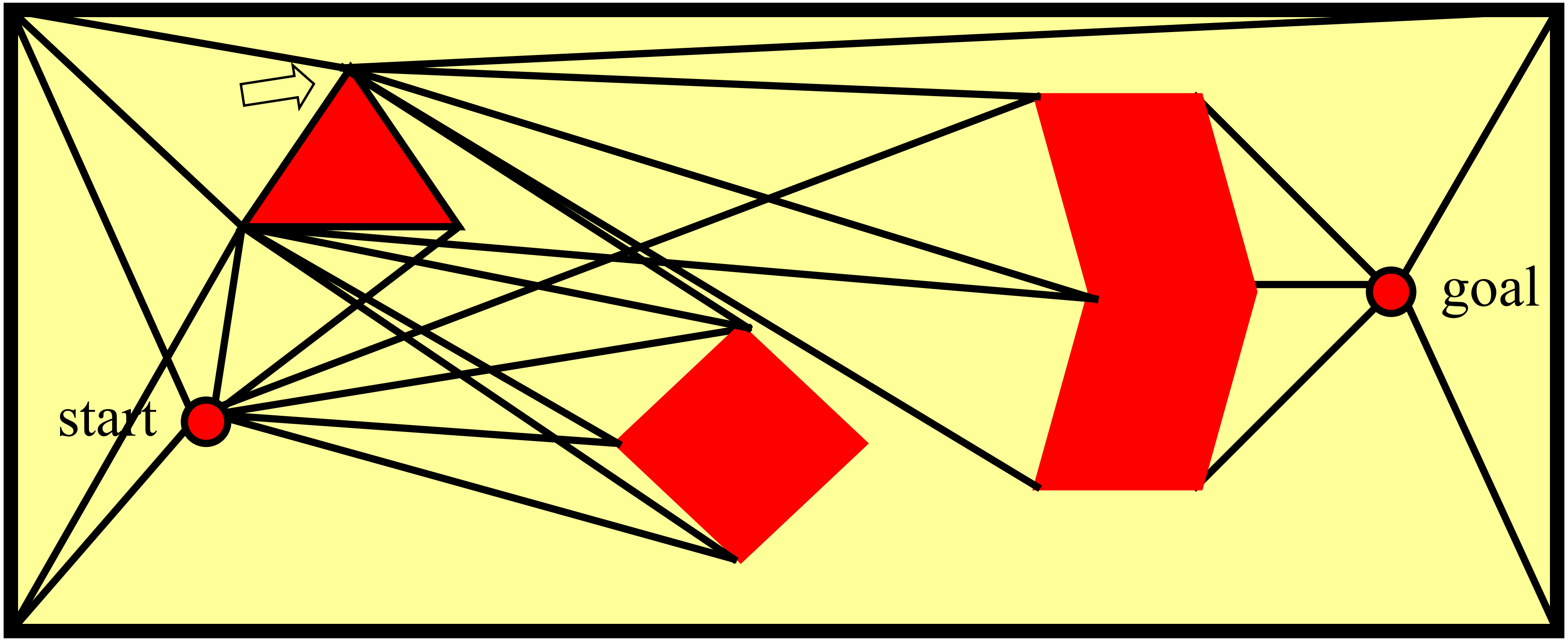
$$e_{ij} \neq \emptyset \iff sv_i + (1-s)v_j \in \mathrm{cl}(\mathcal{Q}_{\mathrm{free}}) \quad \forall s \in (0,1)$$



The Visibility Graph in Action (Part 3)

- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.

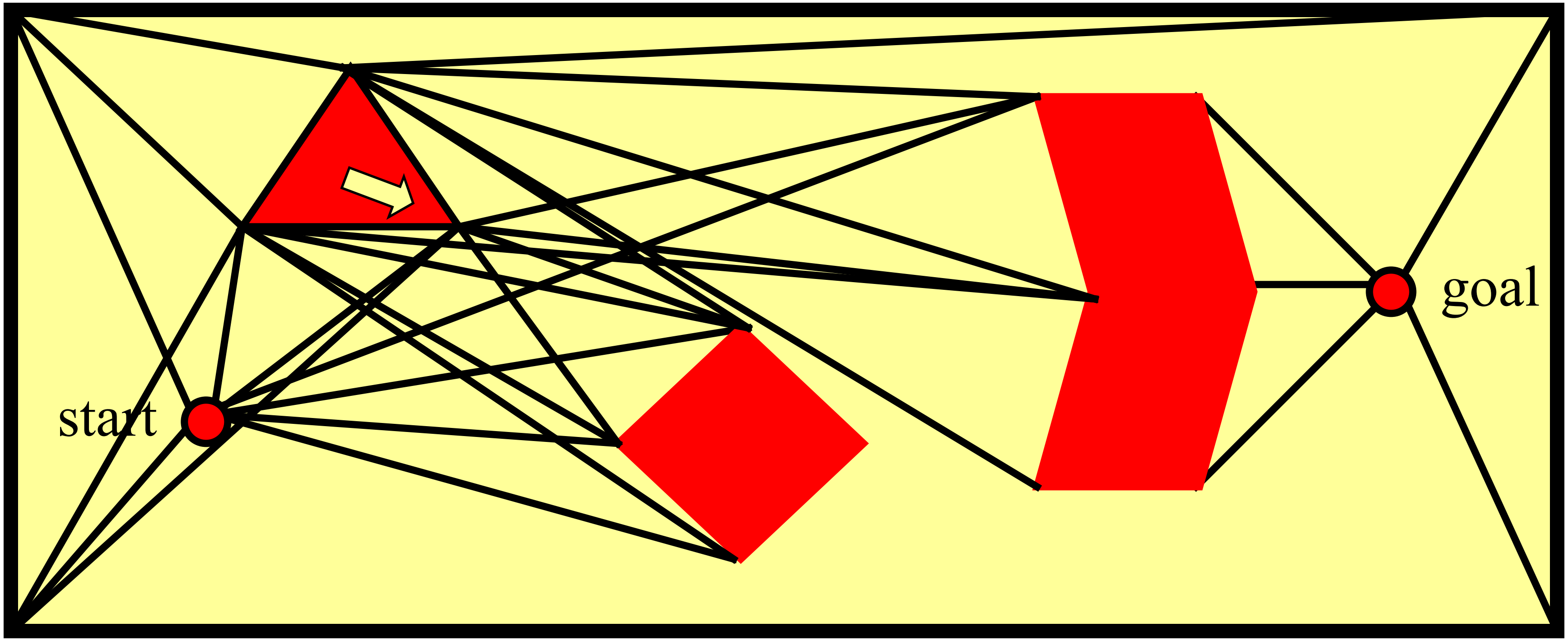
$$e_{ij} \neq \emptyset \iff sv_i + (1-s)v_j \in \mathrm{cl}(\mathcal{Q}_{\mathrm{free}}) \quad \forall s \in (0,1)$$



The Visibility Graph in Action (Part 4)

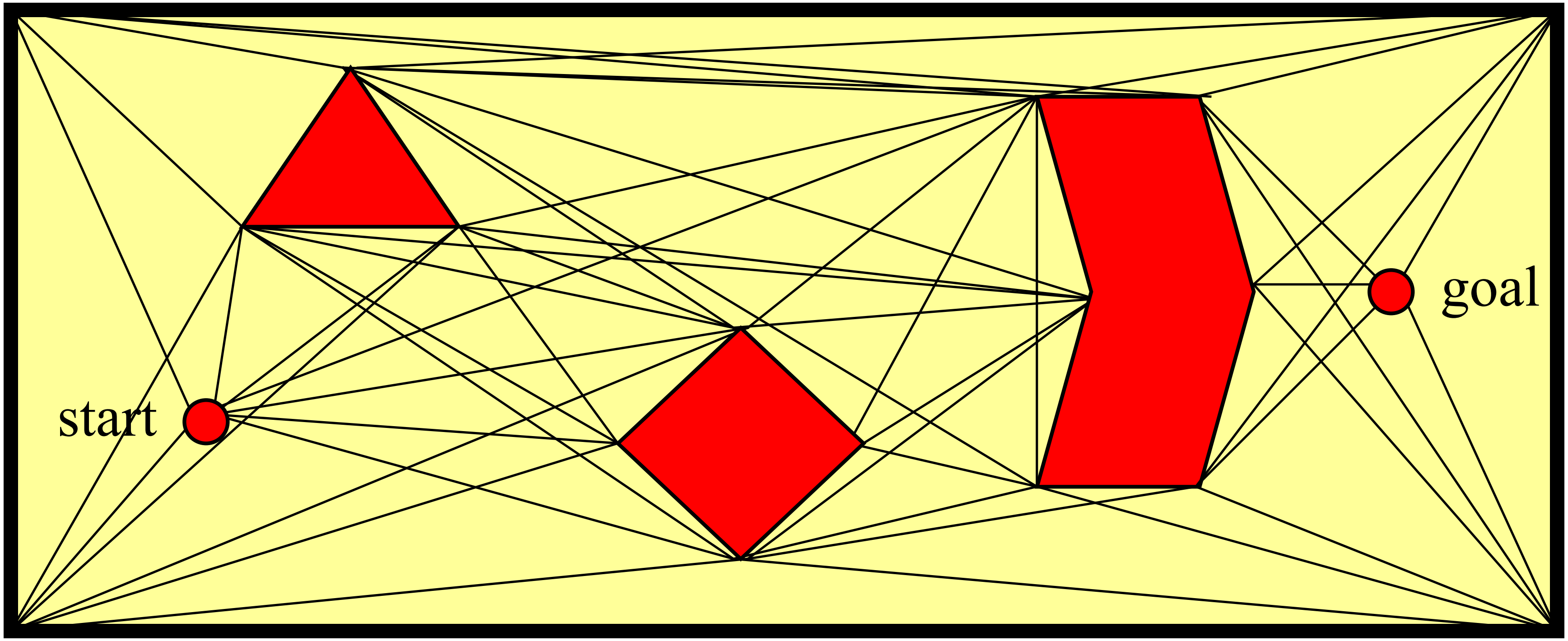
- Second, draw lines of sight from every vertex of every obstacle like before. Remember lines along edges are also lines of sight.

$$e_{ij} \neq \emptyset \iff sv_i + (1-s)v_j \in \mathrm{cl}(\mathcal{Q}_{\mathrm{free}}) \quad \forall s \in (0,1)$$



The Visibility Graph (Done)

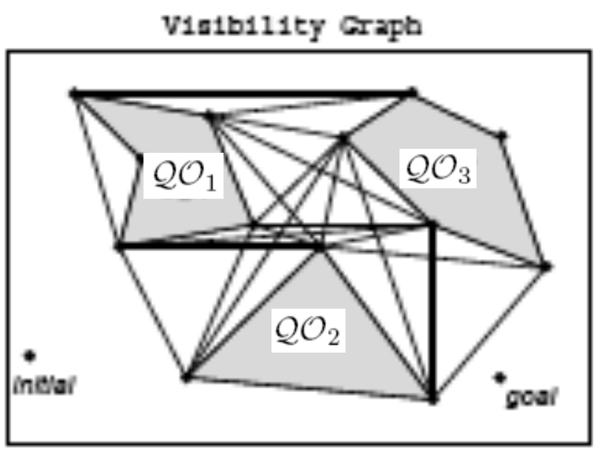
- Repeat until you're done.
- If there are n vertices, then there are $O(n^2)$ edges in the visibility graph – this is a bound, not the exact number of edges.

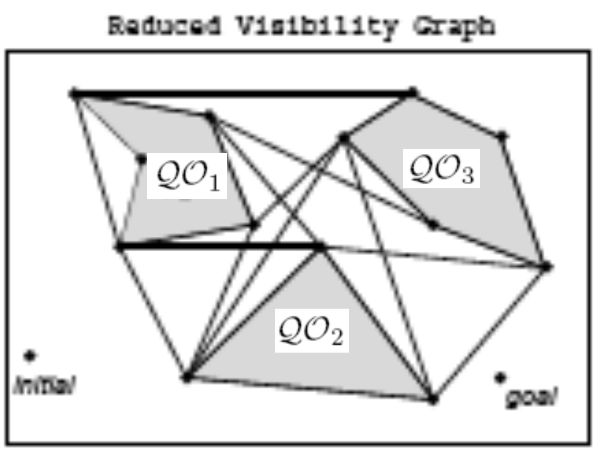


Reduced Visibility Graphs

- The current graph as too many edges
  - lines to concave vertices
  - lines that “head into” the object

- A reduced visibility graph consists of
  - Vertices that are convex
  - Edges that are “tangent” (i.e. do not head into the object at either endpoint)





interestingly, this all only works in $\Re^2$

A Sweepline Algorithm:

Initially:

calculate the angle α_i of segment v-v_i and sort vertices by this creating list E

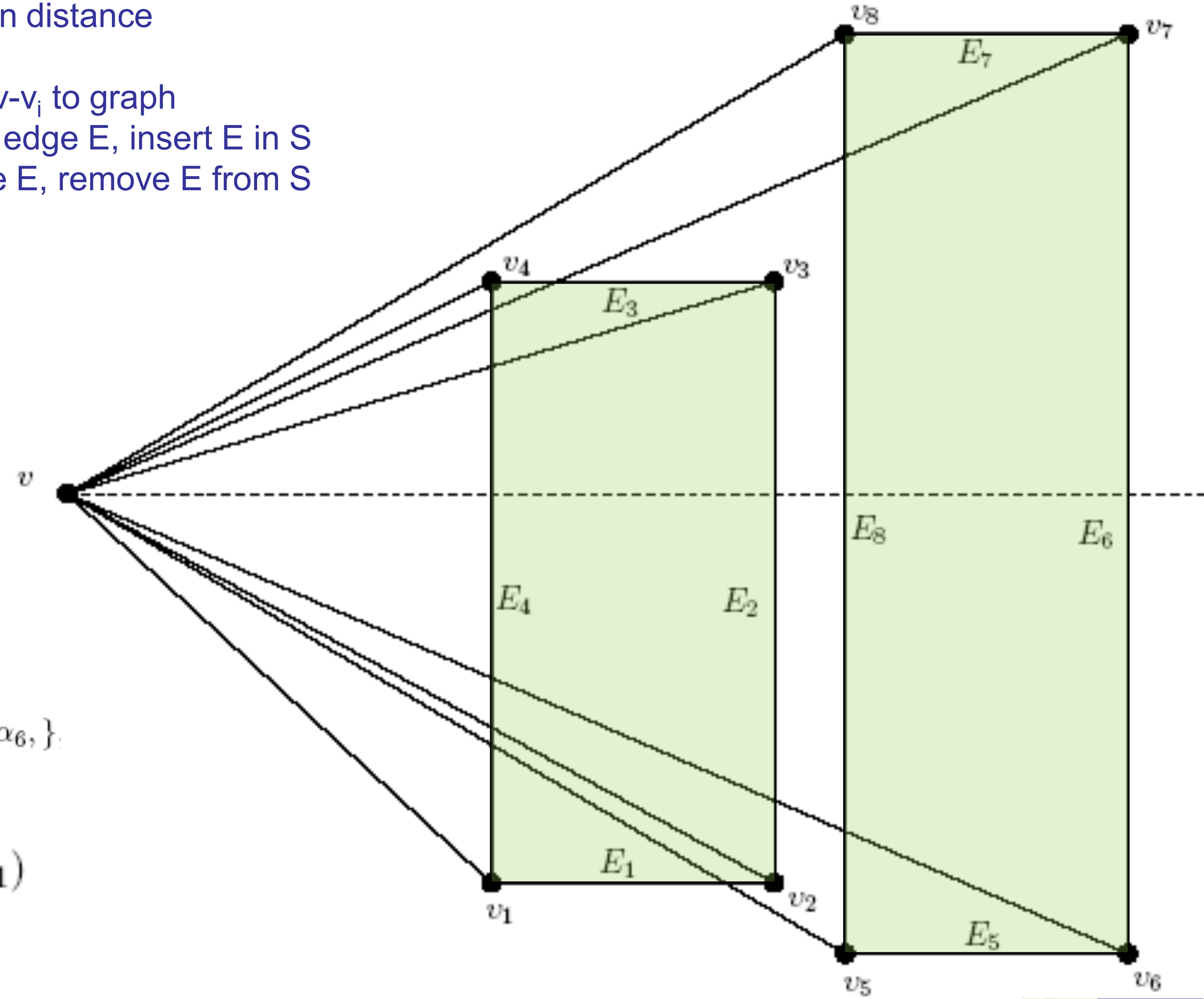
create a list of edges that intersect the horizontal from v sorted by intersection distance

For each α_i

if v_i is visible to v then add v-v_i to graph

if v_i is the "beginning" of an edge E, insert E in S

if v_i is the "end" of and edge E, remove E from S



$$\mathcal{E} = \{\alpha_3, \alpha_7, \alpha_4, \alpha_8, \alpha_1, \alpha_5, \alpha_2, \alpha_6, \}$$

$$(v, v_4),\ (v, v_8),\ \text{and}\ (v, v_1)$$

The Sweepline Algorithm

For each vertex v_i, calculate α_i, the angle from the horizontal axis to the line segment $\overline{vv_i}$. $O(n)$

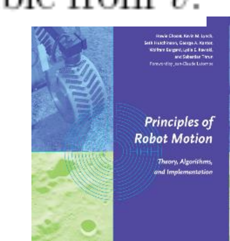
Create the vertex list $\mathcal{E}$, containing the α_i's sorted in increasing order. $O(n \log n)$

Create the active list $\mathcal{S}$, containing the sorted list of edges that intersect the horizontal half-line emanating from v. $O(n \log n)$

```
for all α_i do                                          O(n log n)   n times (once for each vertex)
    if v_i is visible to v then
        Add the edge (v, v_i) to the visibility graph.
    end if
    if v_i is the beginning of an edge, E, not in S then
        Insert the E into S.                            O(log n)
    end if
    if v_i is the end of an edge in S then
        Delete the edge from S.                         O(log n)
    end if
end for
```

If the line segment $\overline{vv_i}$ does not intersect the closest edge in $\mathcal{S}$, and if l does not lie between the two edges incident on v then v_i is visible from v.

Analysis: For a vertex, n log n to create initial list, log n for each α_i
Overall: n log (n) (or n^2 log (n) for all n vertices



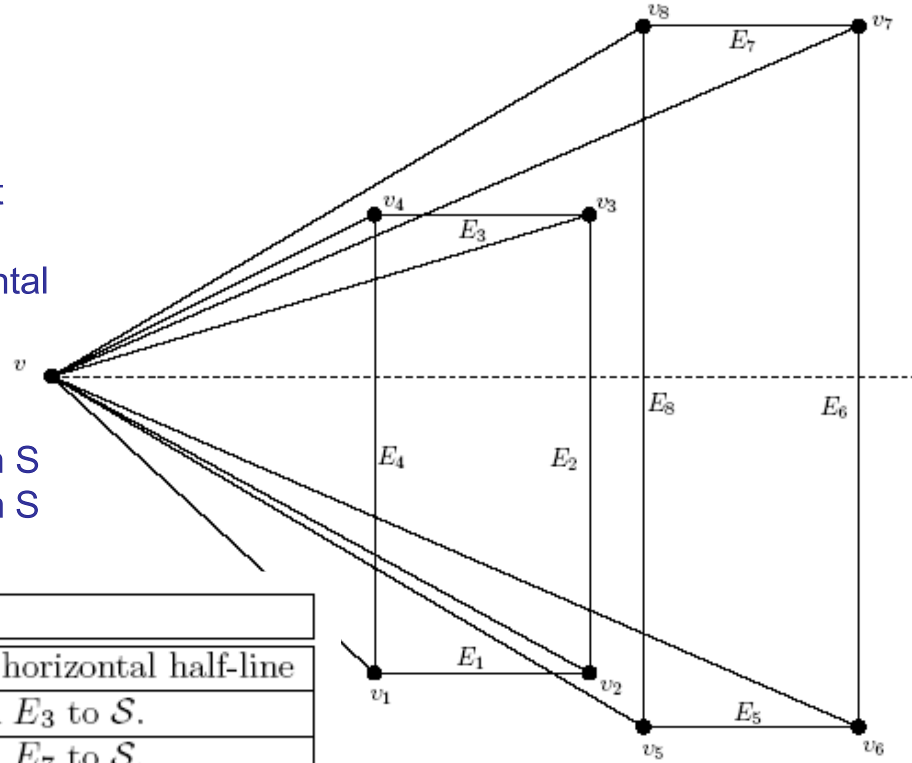
Algorithm:

<u>Initially:</u>

- calculate the angle α_i of segment v-v_i and sort vertices by this creating list E
- create a list of edges that intersect the horizontal from v sorted by intersection distance

<u>For each α_i</u>

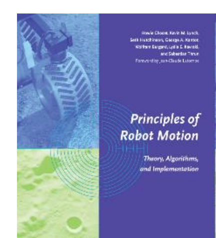
- if v_i is visible to v then add v-v_i to graph
- if v_i is the "beginning" of an edge E, insert E in S
- if v_i is the "end" of and edge E, remove E from S



Vertex	New $\mathcal{S}$	Actions
Initialization	$\{E_4, E_2, E_8, E_6\}$	Sort edges intersecting horizontal half-line
α_3	$\{E_4, E_3, E_8, E_6\}$	Delete E_2 from $\mathcal{S}$. Add E_3 to $\mathcal{S}$.
α_7	$\{E_4, E_3, E_8, E_7\}$	Delete E_6 from $\mathcal{S}$. Add E_7 to $\mathcal{S}$.
α_4	$\{E_8, E_7\}$	Delete E_3 from $\mathcal{S}$. Delete E_4 from $\mathcal{S}$. ADD (v, v_4) to visibility graph
α_8	$\{\}$	Delete E_7 from $\mathcal{S}$. Delete E_8 from $\mathcal{S}$. ADD (v, v_8) to visibility graph
α_1	$\{E_1, E_4\}$	Add E_4 to $\mathcal{S}$. Add E_1 to $\mathcal{S}$. ADD (v, v_1) to visibility graph
α_5	$\{E_4, E_1, E_8, E_5\}$	Add E_8 to $\mathcal{S}$. Add E_5 to $\mathcal{S}$.
α_2	$\{E_4, E_2, E_8, E_5\}$	Delete E_1 from $\mathcal{S}$. Add E_2 to $\mathcal{S}$.
α_6	$\{E_4, E_2, E_8, E_6\}$	Delete E_5 from $\mathcal{S}$. Add E_6 to $\mathcal{S}$.
Termination		

$$\mathcal{E} = \{\alpha_3, \alpha_7, \alpha_4, \alpha_8, \alpha_1, \alpha_5, \alpha_2, \alpha_6,\}$$

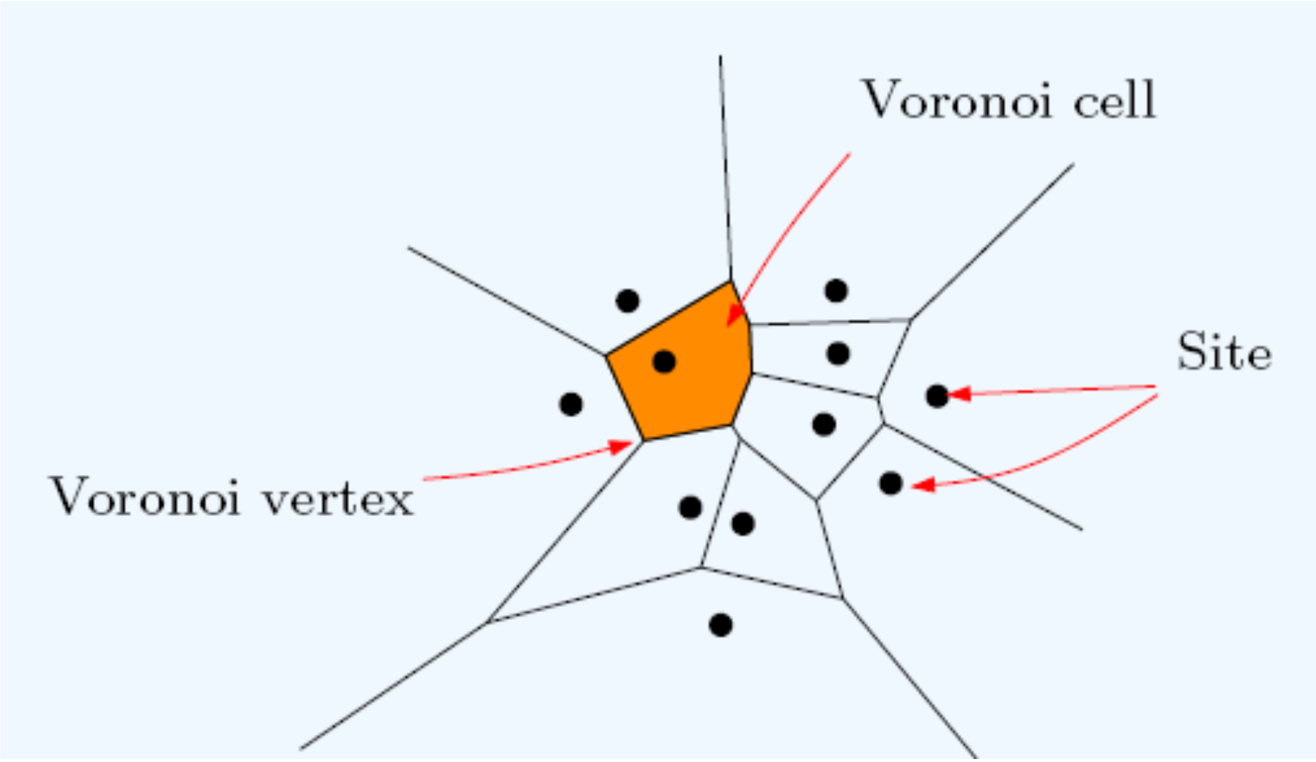
(v, v_4), (v, v_8), and (v, v_1)



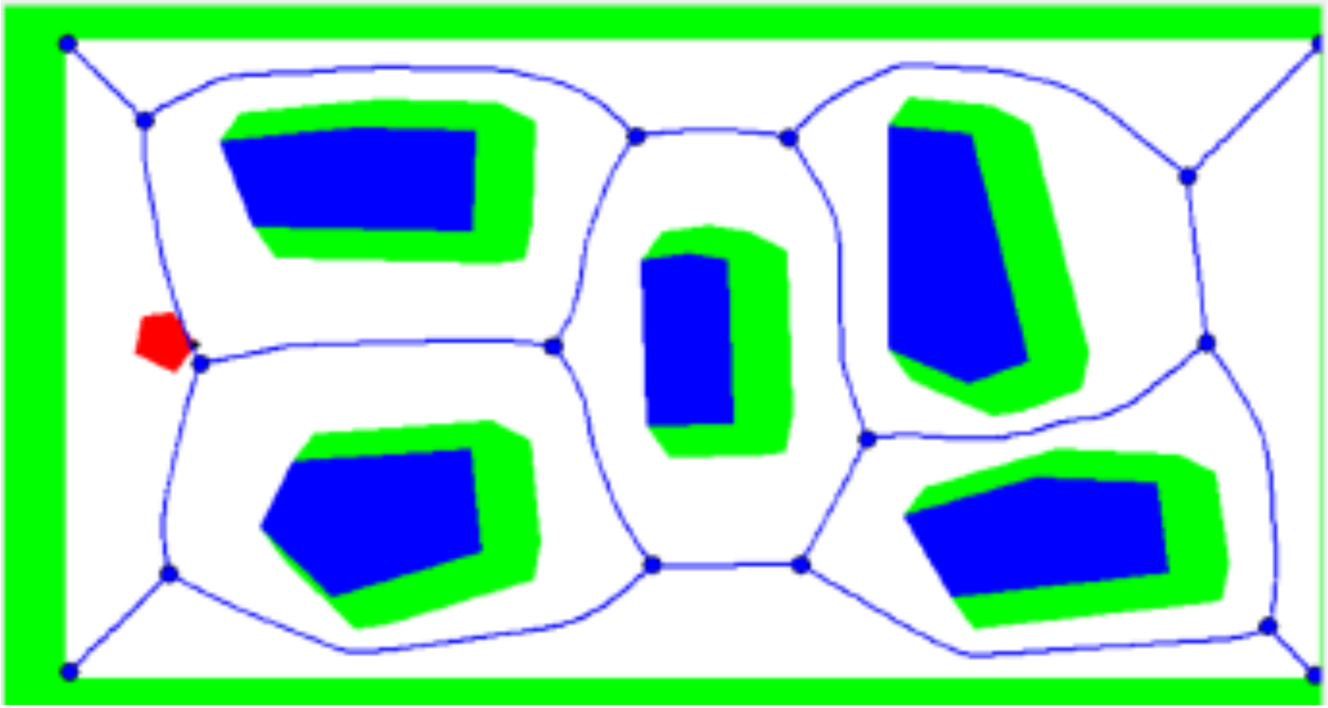
Safe Paths that Have Large Clearance to Obstacles

The Generalized Voronoi Diagram

Voronoi Diagrams

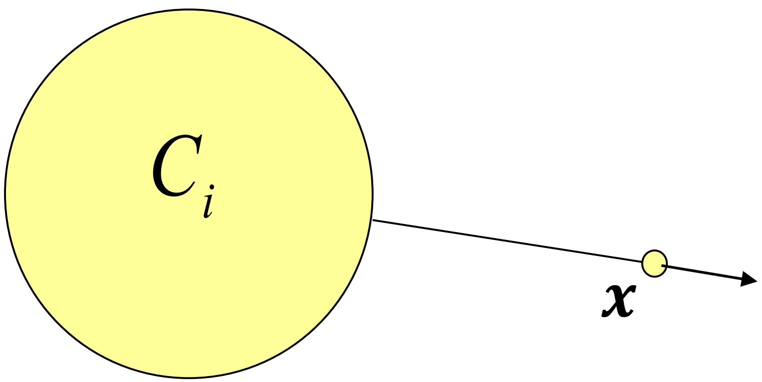


Generalized Voronoi Diagrams



Beyond Points: Basic Definitions

$d_i(x)$ is the distance from the point x to the nearest point that belongs to an obstacle.



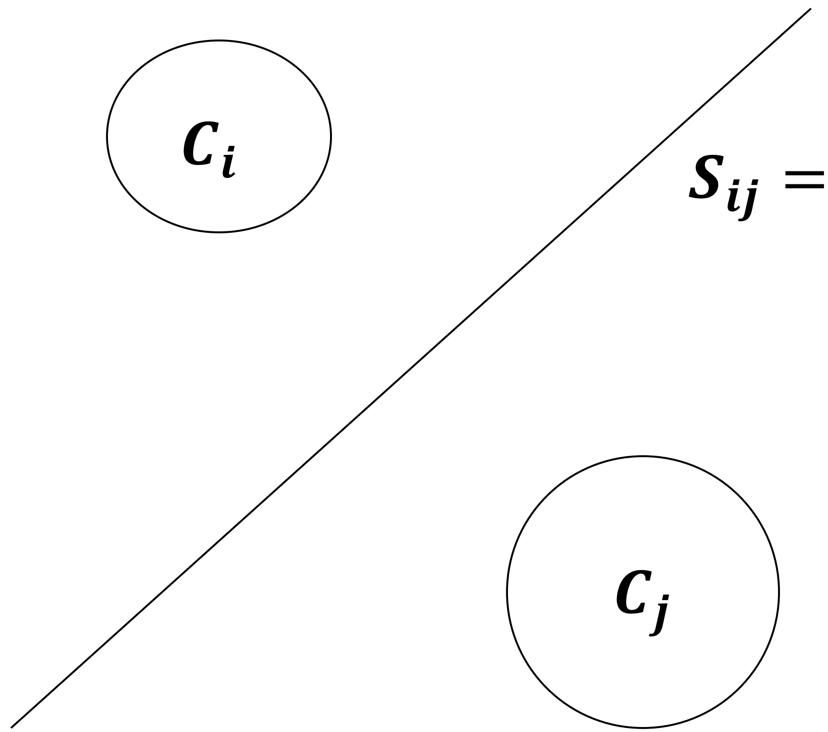
$$\nabla \mathrm{d_i(x)} = \frac{x-c}{||x-c||}$$

we'll use this later…

$$\mathrm{d_i(x)} = \min_{\mathrm{c} \in \partial C_i} \mathrm{d(x,c)}$$

Two-Equidistant

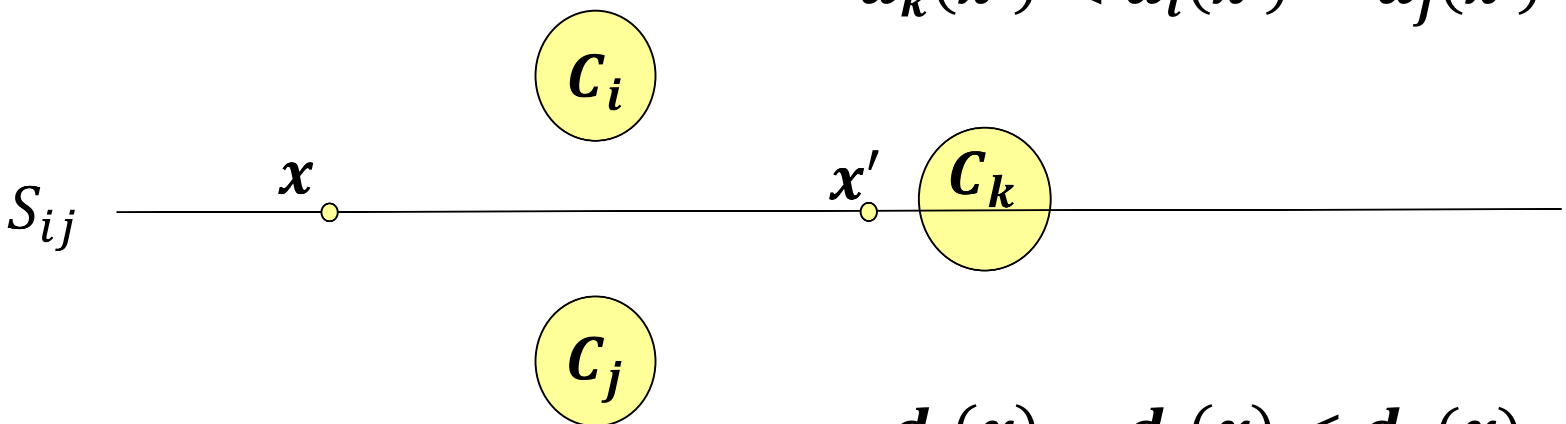
A Two-equidistant surface is the set of points equally distant to two obstacles.



$$\boldsymbol{S_{ij} = \{x \mid d_i(x) - d_j(x) = 0|}$$

More Frugal Definition

$$\boldsymbol{d_k(x') < d_i(x') = d_j(x')}$$



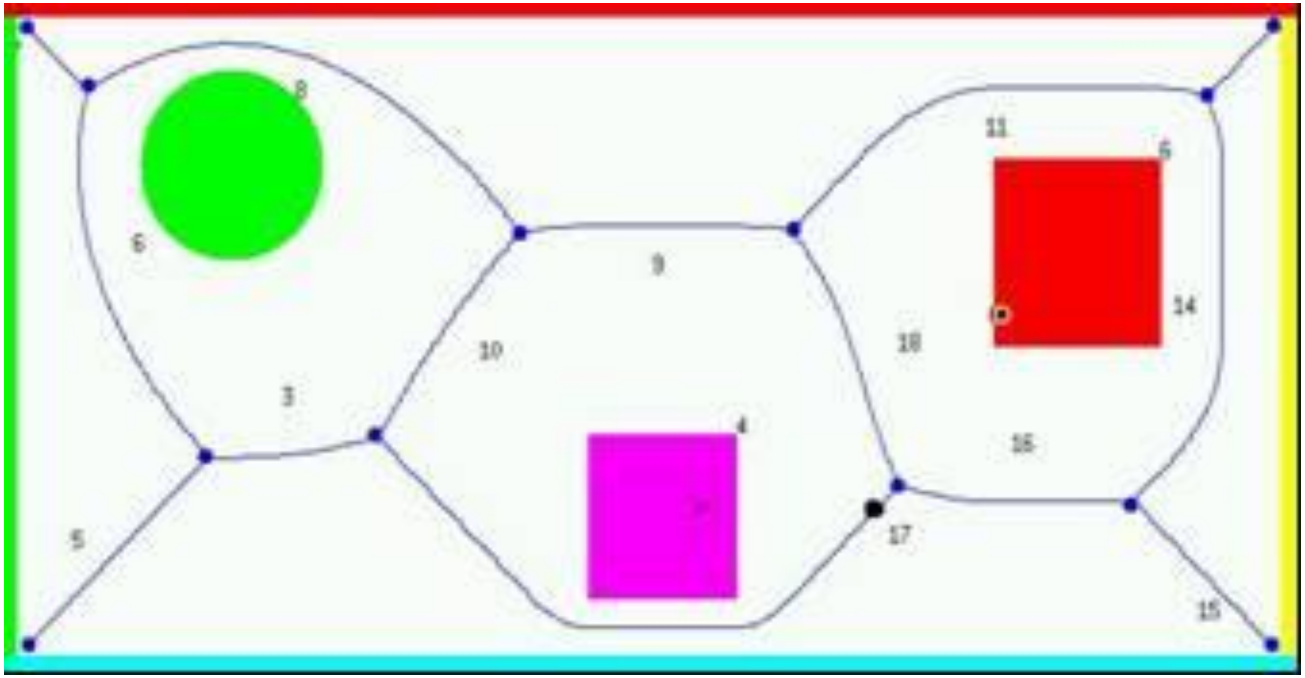
$$\boldsymbol{d_i(x) = d_j(x) < d_k(x)}$$

Two-Equidistant *Face*

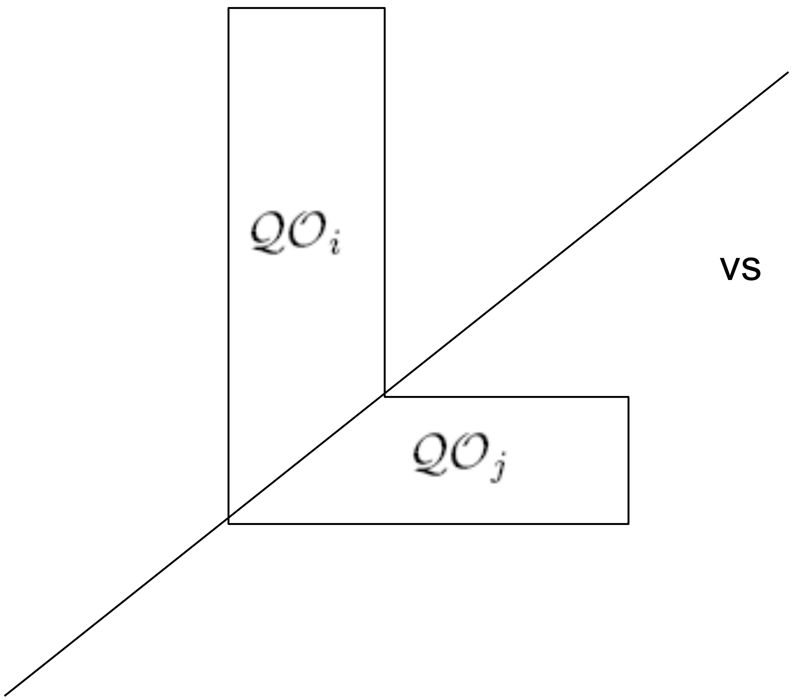
$$F_{ij} = \{ \mathrm{x} \in S_{ij} \mid d_i(x) = d_j(x) < d_k(x), \text{for all } h \neq i, j$$

General Voronoi Diagram

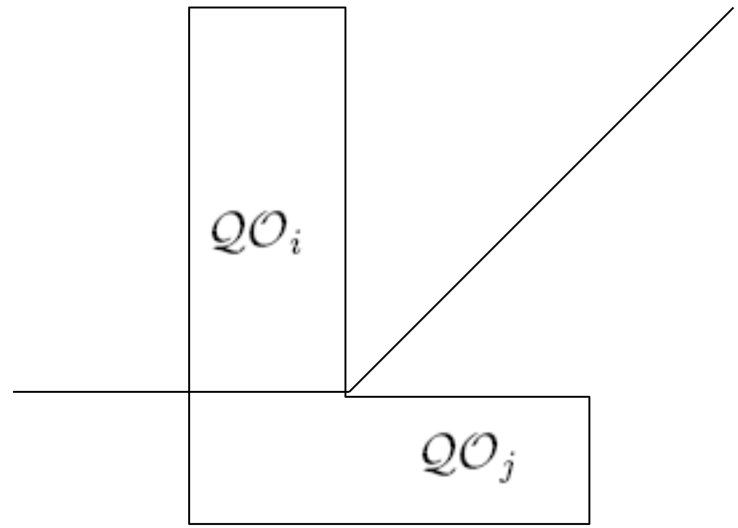
$$\mathrm{GVD} = \bigcup_{i=1}^{n-1} \bigcup_{j=i+1}^{n} F_{ij}$$



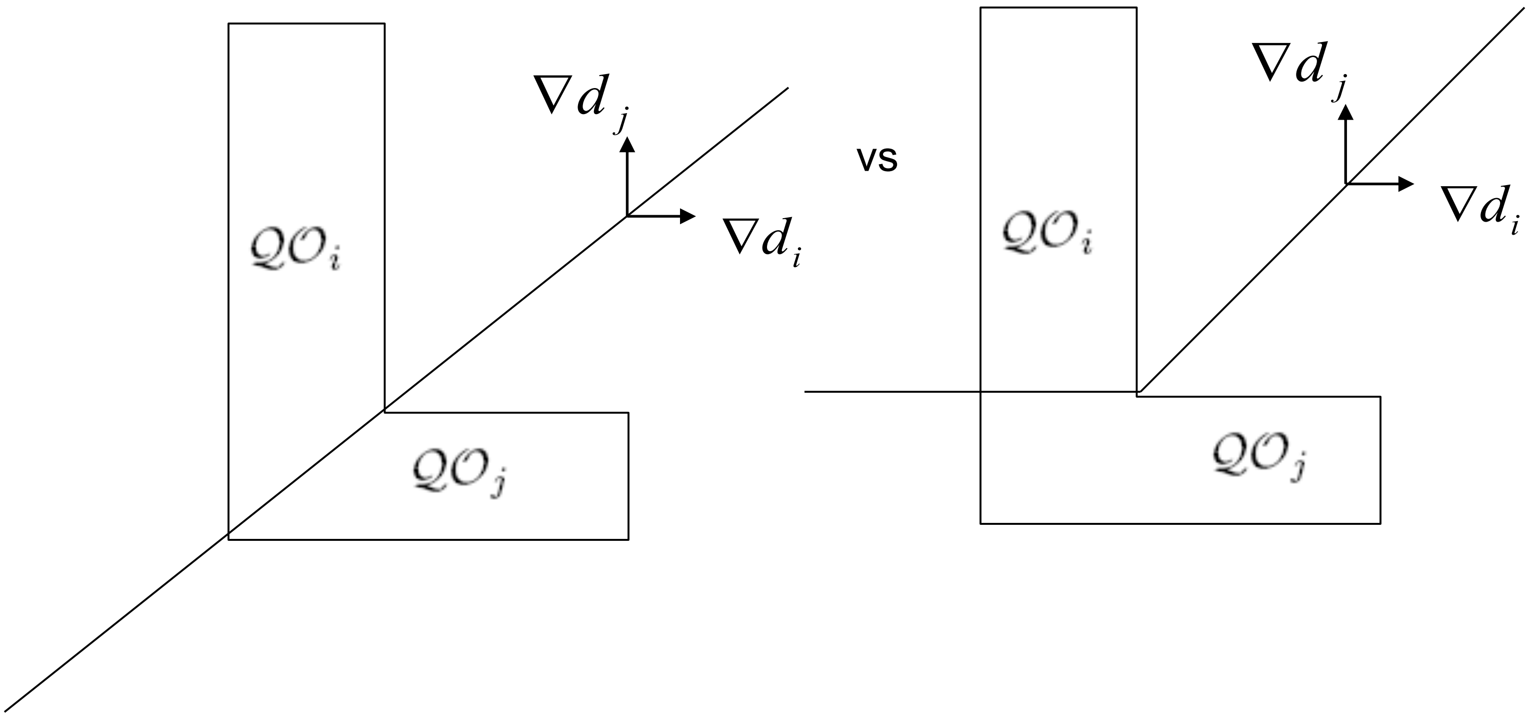
What about concave obstacles?



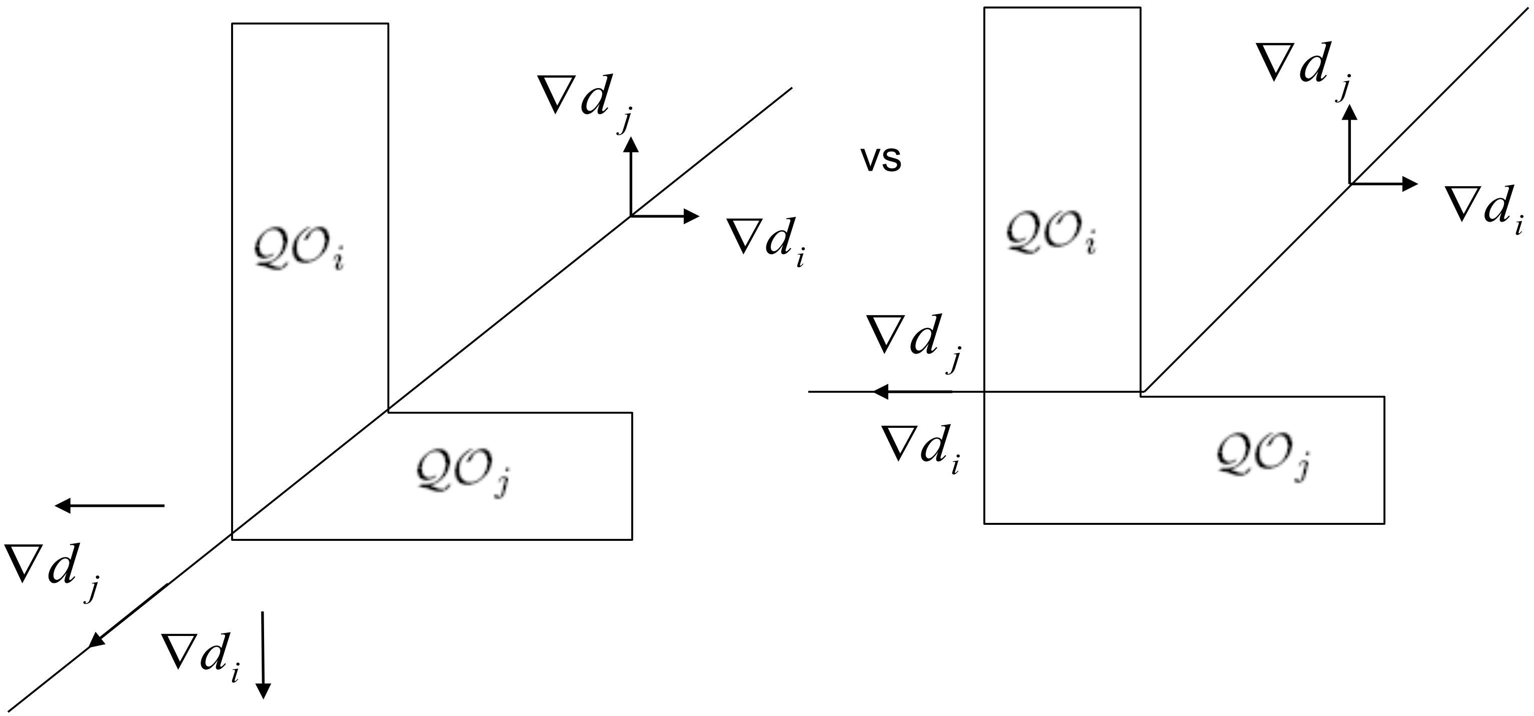
vs



What about concave obstacles?



What about concave obstacles?



Two-Equidistant

- *Two-equidistant surface*

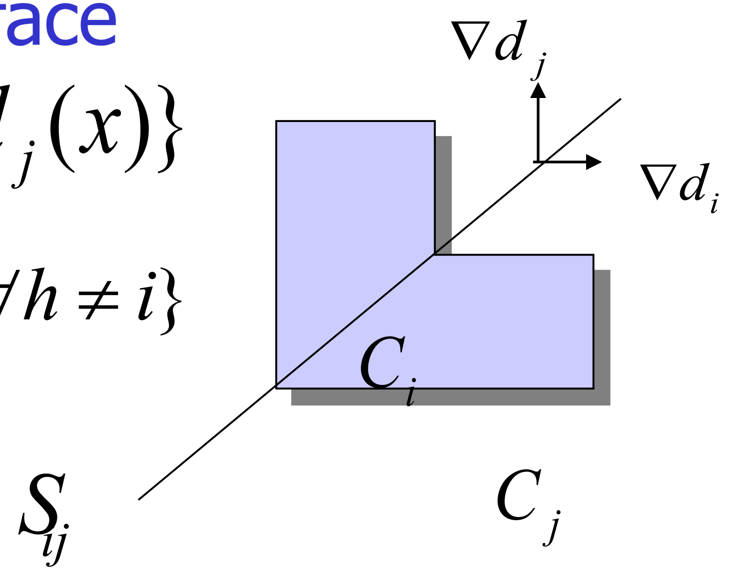
$$S_{ij} = \{x \in Q_{\text{free}} : d_i(x) - d_j(x) = 0\}$$

- Two-equidistant surjective surface

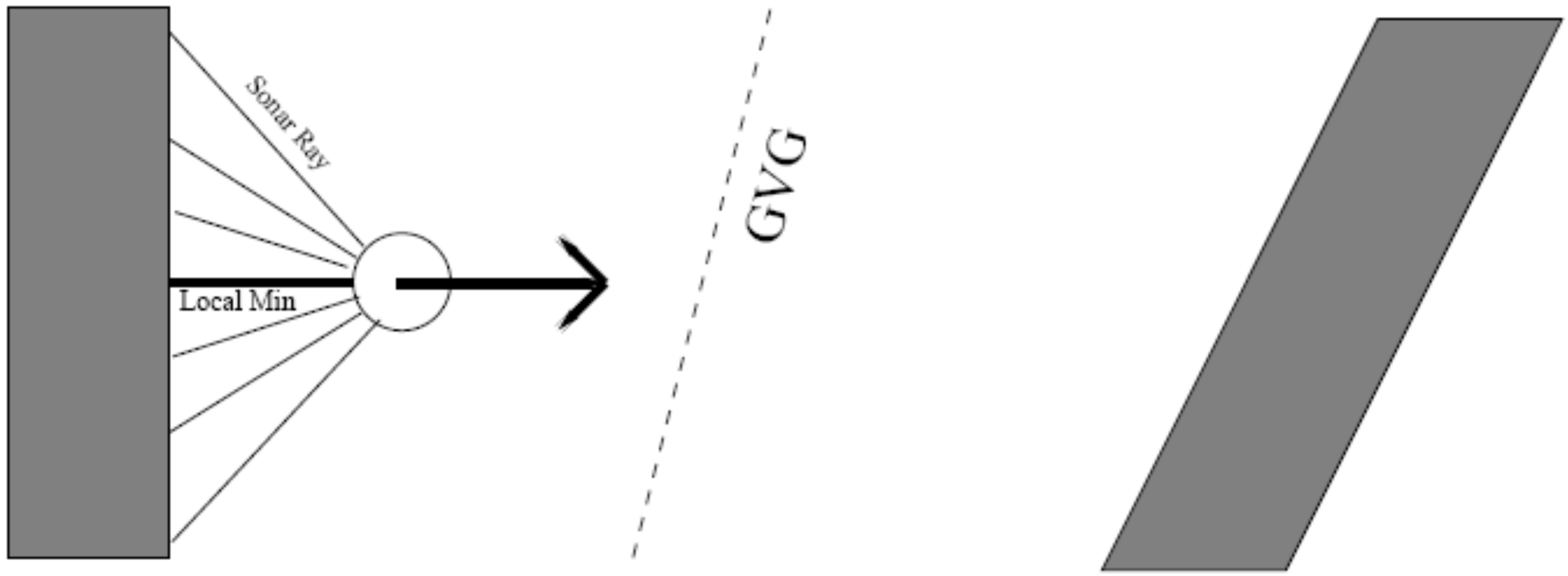
$$SS_{ij} = \{x \in S_{ij} : \nabla d_i(x) \neq \nabla d_j(x)\}$$

$$F_{ij} = \{x \in SS_{ij} : d_i(x) \leq d_h(x), \forall h \neq i\}$$

$$\text{GVD} = \bigcup_{i=1}^{n-1} \bigcup_{j=i+1}^{n} F_{ij}$$



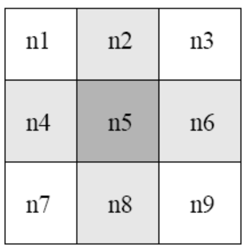
Accessibility (in the Plane)



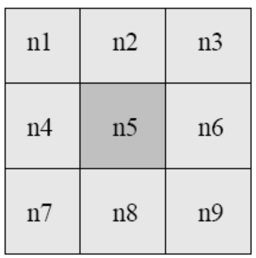
Follow the gradient of the distance function until another obstacle is equally close.

A Discrete Version of the Generalized Voronoi Diagram

- use a discrete version of space and work from there
  - The Brushfire algorithm is one way to do this
    - need to define a grid on space
    - need to define connectivity (4/8)
    - obstacles start with a 1 in grid; free space is zero



4



8

Brushfire Algorithm

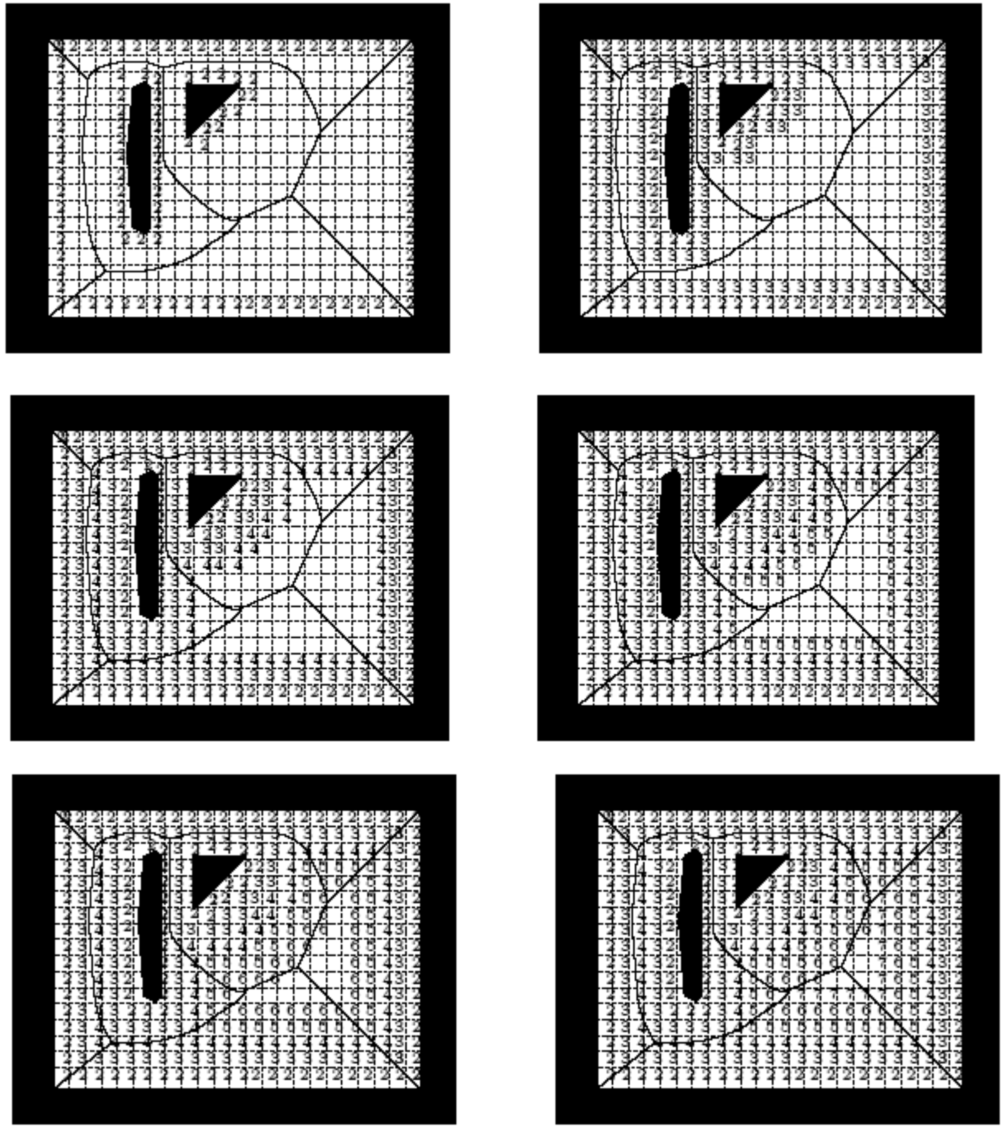
- Initially: create a queue L of pixels on the boundary of all obstacles, set $d(t) = 0$ for each non-boundary grid cell t

- While $L \neq \emptyset$
  - pop the top element t of L
  - if $d(t) \ = \ 0$
    - $d(t) \leftarrow 1 + \min_{t' \in N(t), d(t') \neq 0} d(t')$
    - $L \leftarrow L \cup \{t' \ \in N(t) \mid d(t) = 0\}$ /* add unvisited neighbors to L

The result is a distance map d where each cell holds the minimum distance to an obstacle.

Local maxima of d define the cells at which "wave fronts" cross, and these lie on the discrete Generalized Voronoi Diagram.

Brushfire example



Note that the curves here are not at all perfect…

Path Planning for Large Empty Spaces

Cell Decomposition

Cell Decomposition

- Don’t explicitly build a 1-D Roadmap.
- The “Roadmap” corresponds to the adjacency graph of the cellular decomposition.
- Nodes in the adjacency graph correspond to free cells.
- Arcs in the adjacency graph connect nodes that correspond to adjacent cells.

Definition

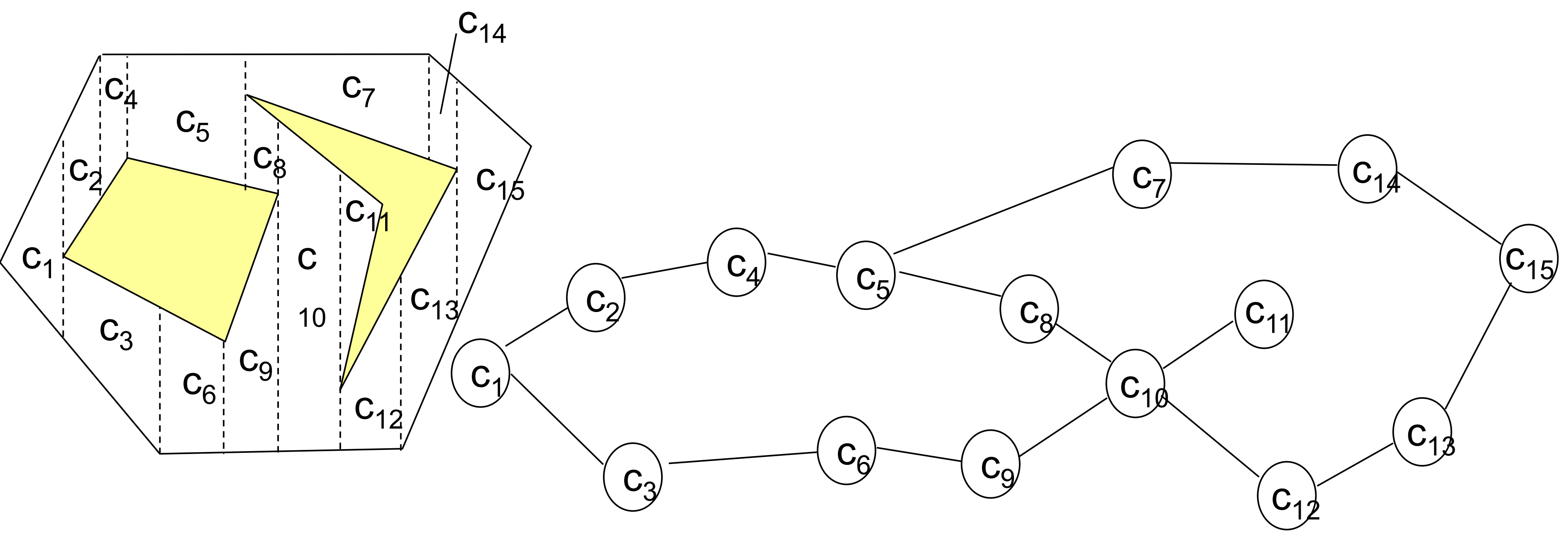
Exact Cellular Decomposition

- ν_i is a cell
- $\mathrm{int}(\nu_i) \cap \mathrm{int}(\nu_j) = \emptyset$ if and only if $i \neq j$
- $Q_{free} \cap (\mathrm{cl}(\nu_i) \cap \mathrm{cl}(\nu_j)) \neq \emptyset$ if ν_i and ν_j are adjacent cells
- $Q_{free} = \cup_i (\nu_i)$

Adjacency Graph

- Node correspond to a cell
- Edge connects nodes of adjacent cells

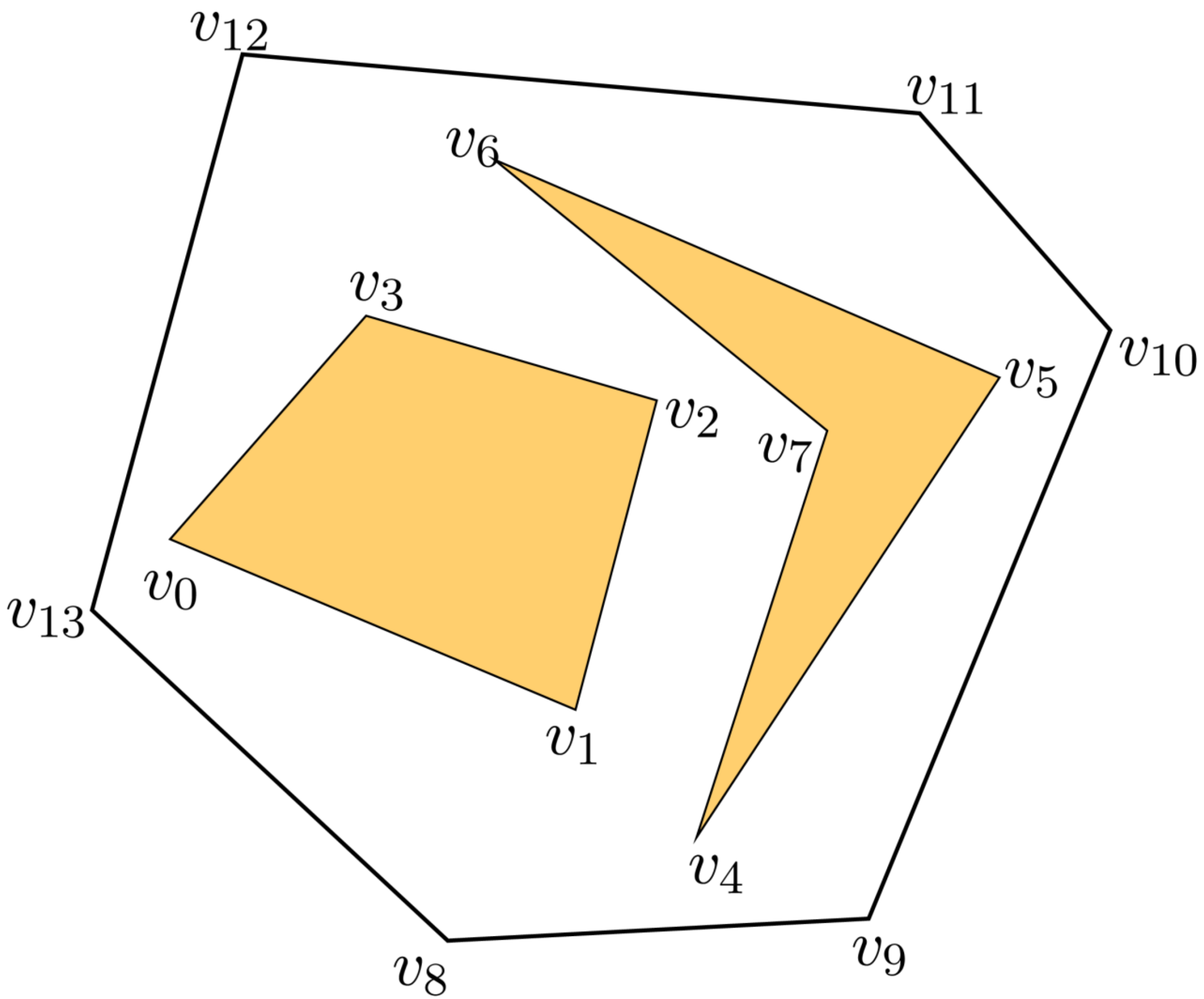
- Two cells are *adjacent* if they share a common boundary



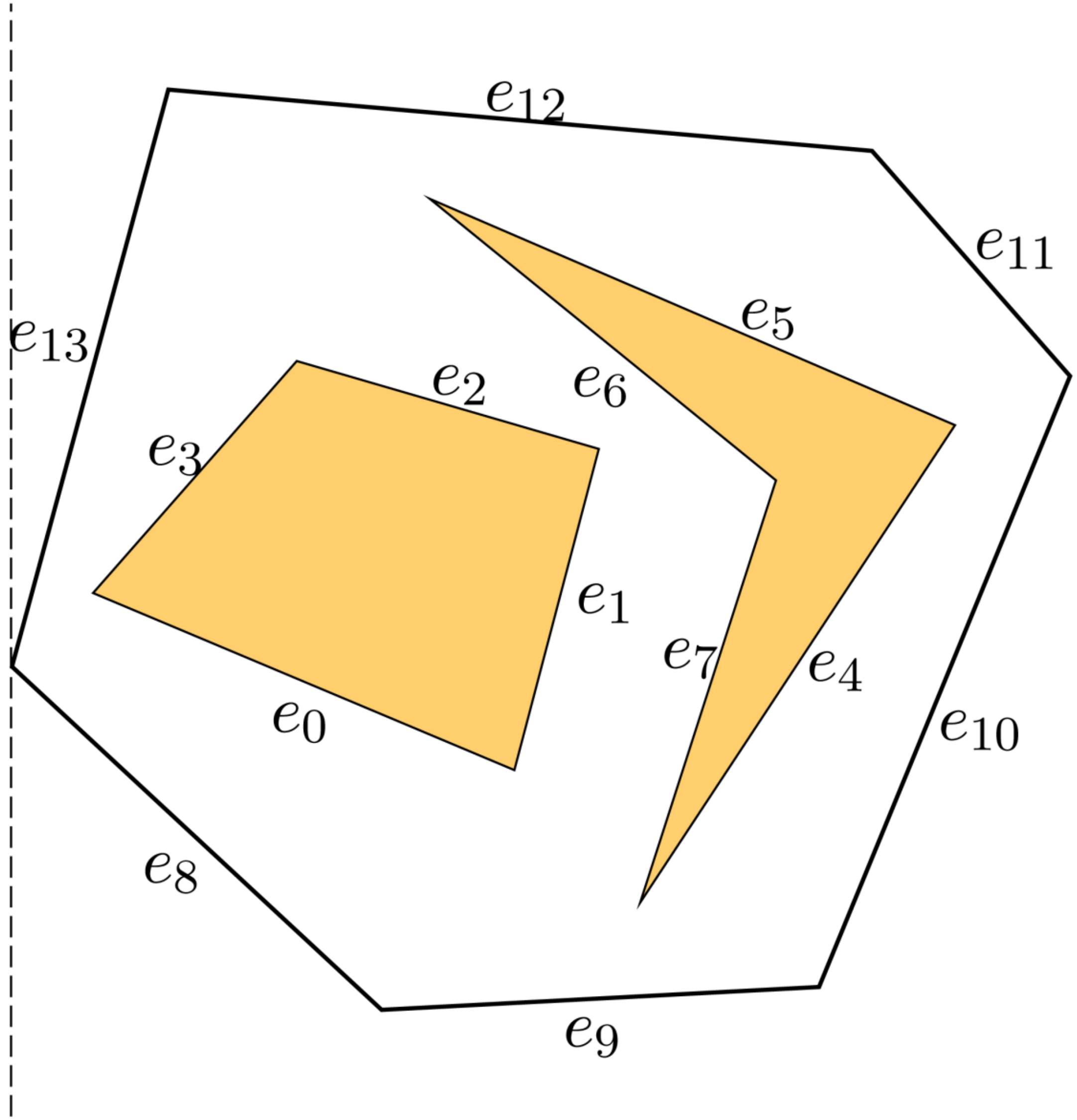
Path Planning

- Path Planning in two steps:
  - Planner determines cells that contain the start and goal
  - Planner searches for a path within adjacency graph

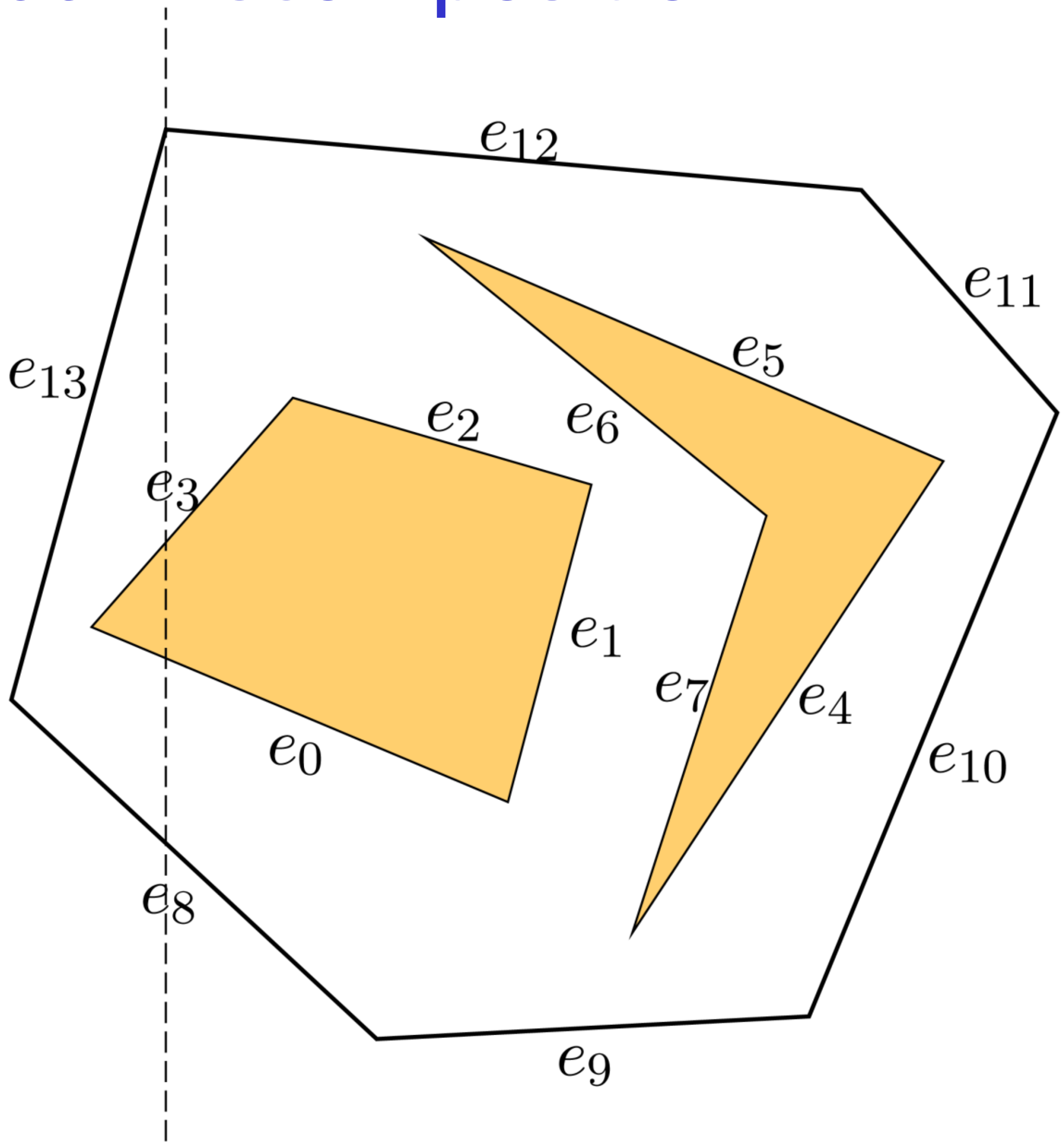
Trapezoidal Decomposition



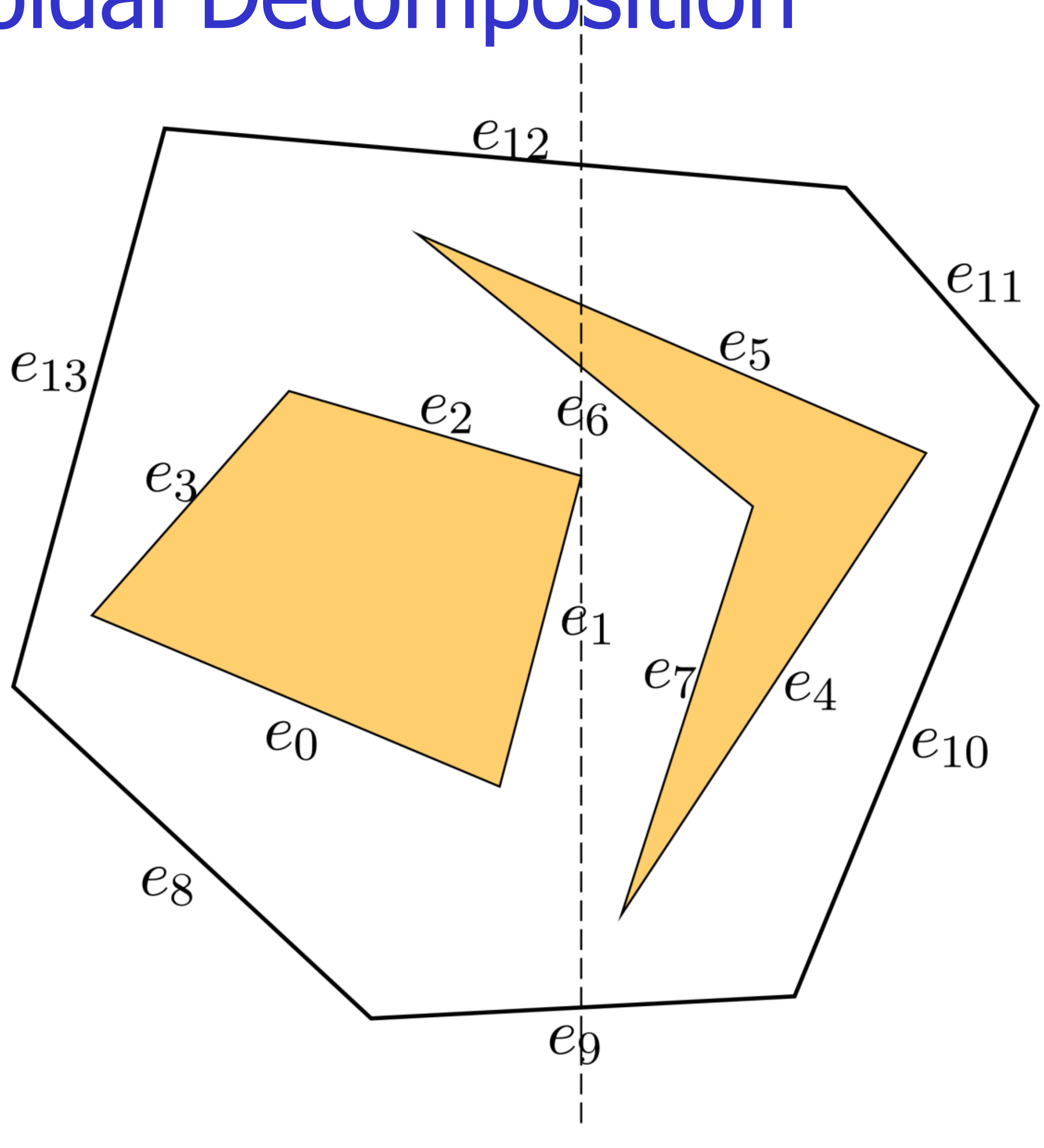
Trapezoidal Decomposition



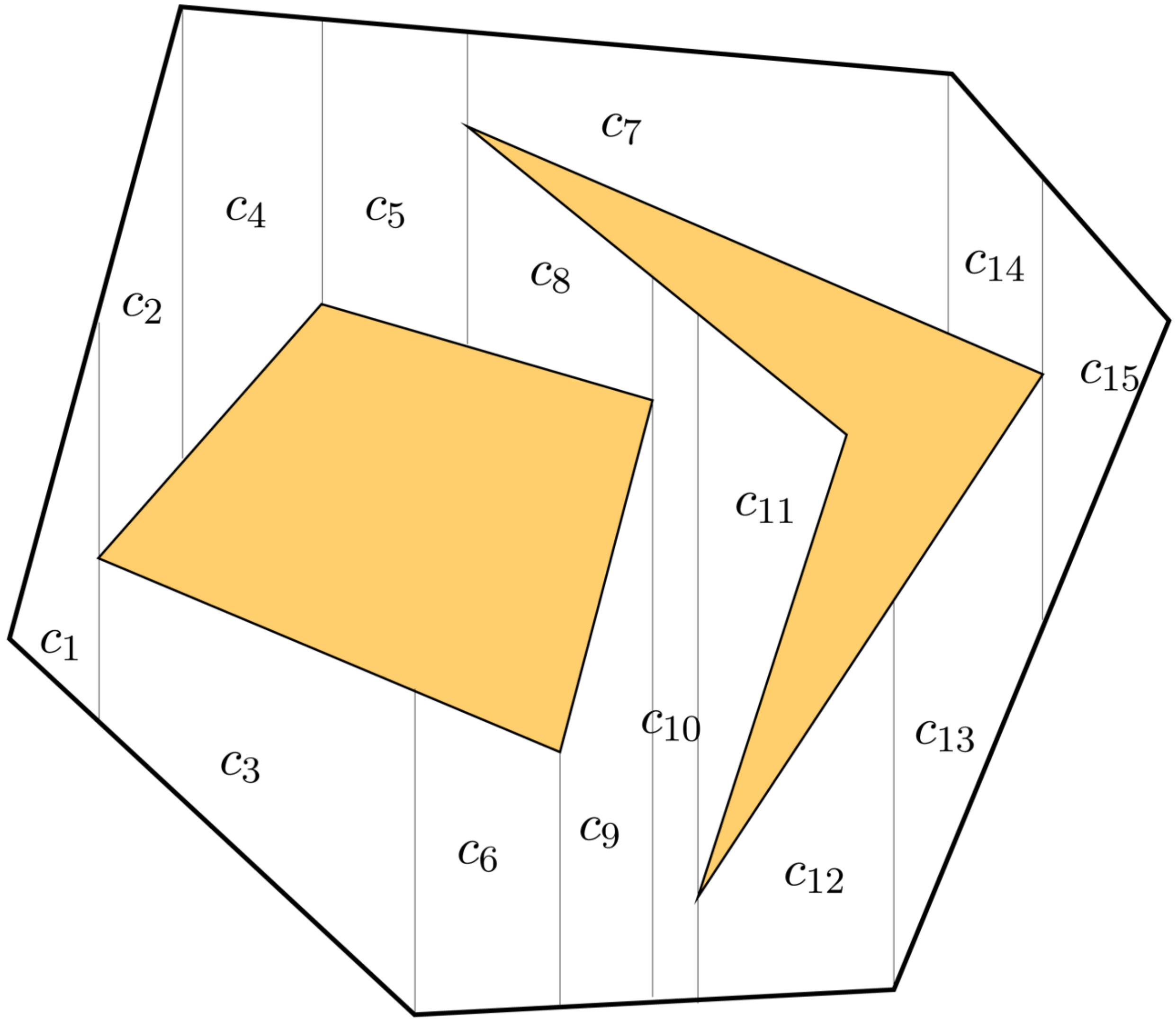
Trapezoidal Decomposition



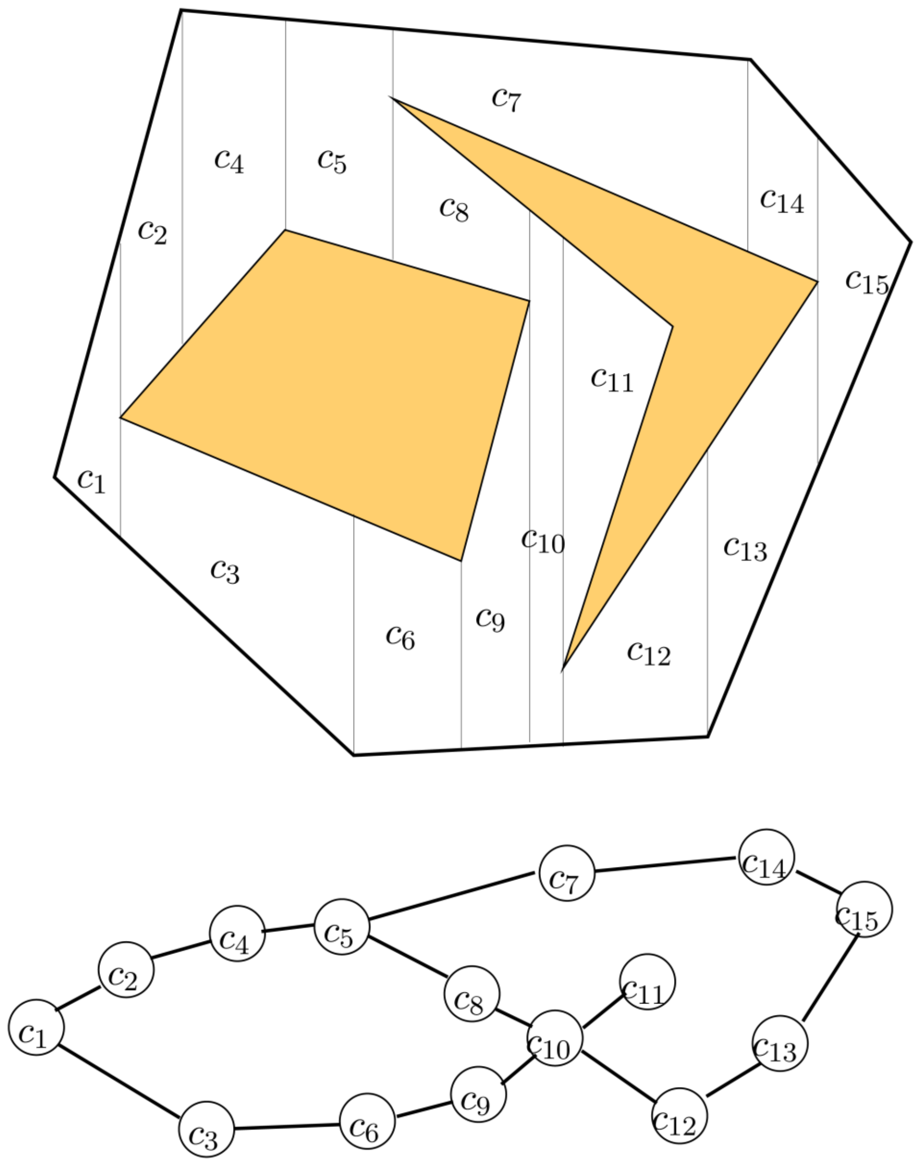
Trapezoidal Decomposition



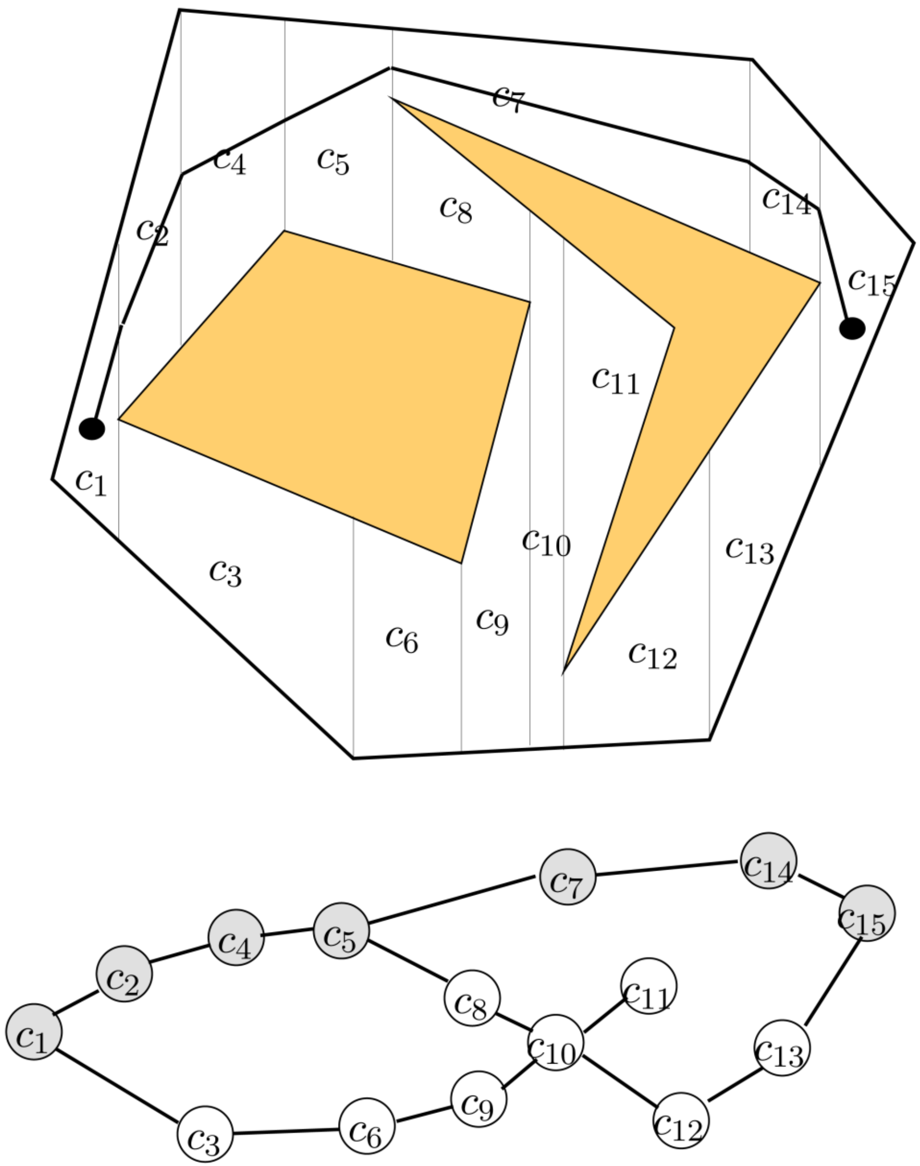
Trapezoidal Decomposition



Trapezoidal Decomposition



Trapezoidal Decomposition Path



Implementation

- Input is vertices and edges
- Sort n vertices O(n logn)
- Determine vertical extensions
  - For each vertex, intersect vertical line with each edge – O(n) time
  - Total O(n^2) time

Sweep line approach

Sweep a line through the space stopping at vertices which are often called events

Maintain a list L of the current edges the slice intersects

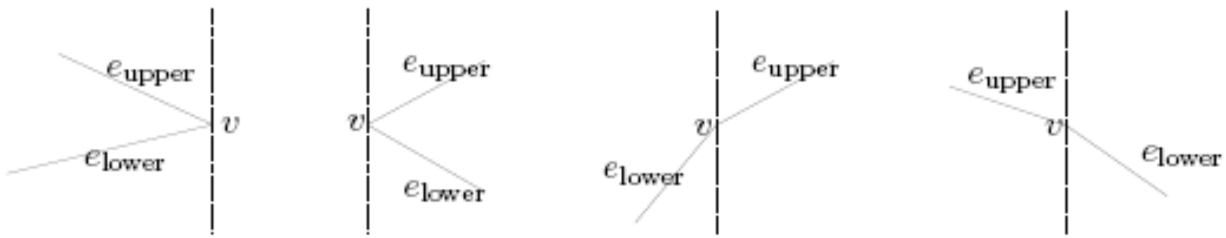
Determining the intersection of slice with L requires O(n) time but with an efficient data structure like a balanced tree, perhaps O(log n)

Really, determine between which two edges the vertex or event lies These edges are e_{LOWER} and e_{UPPER}

So, really maintaining L takes O(n log n) – log n for insertions, n for vertices

Events

"other" vertex of e_{lower} has a y-coordinate lower than the "other" vertex of e_{upper}



Out

e_{lower} and e_{upper} are both to the left of the sweep line

- delete e_{lower} and e_{upper} from the list
- $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$
  $(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots)$

Middle

e_{lower} is to the left and e_{upper} is to the right of the sweep line

- delete e_{lower} from the list and insert e_{upper}
- $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{UPPER}}, \ldots)$
  $(\ldots, e_{\text{LOWER}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

In

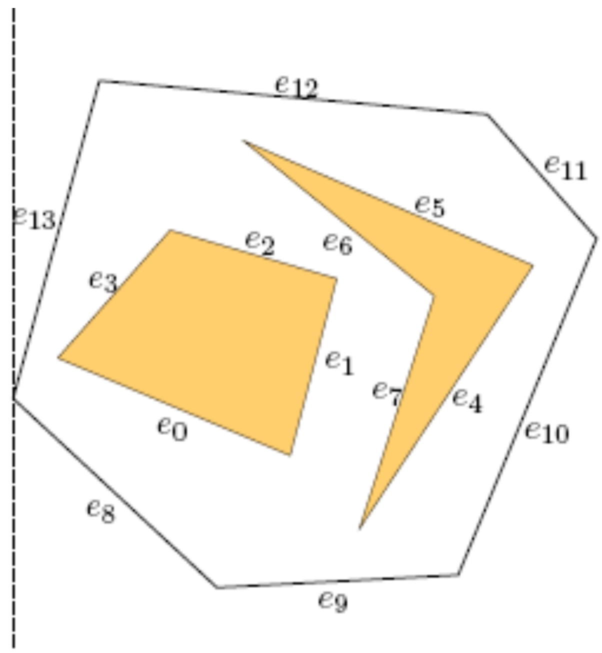
e_{lower} and e_{upper} are both to the right of the sweep line

- insert e_{lower} and e_{upper} into the list
- $(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

e_{lower} is to the right and e_{upper} is to the left of the sweep line

- delete e_{upper} from the list and insert e_{lower}
- $(\ldots, e_{\text{LOWER}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$
  $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{UPPER}}, \ldots)$

Example



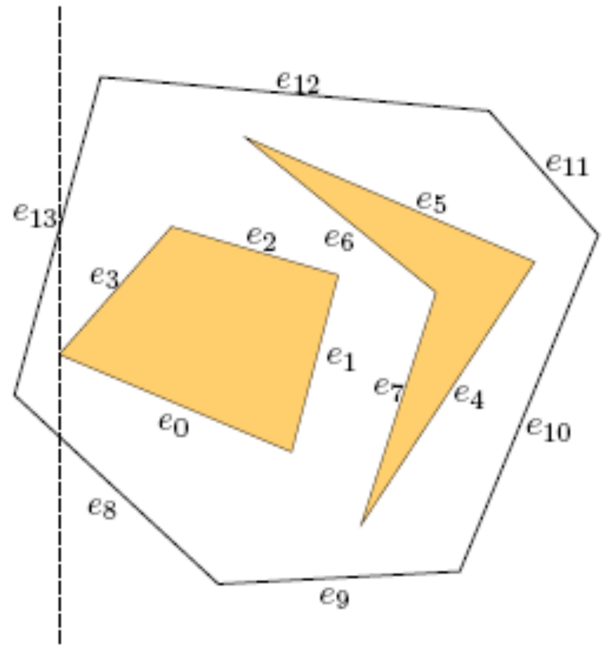
$$L : \emptyset \to \{e_8, e_{13}\}$$

e_{lower} and e_{upper} are both to the right of the sweep line

- insert e_{lower} and e_{upper} into the list
- $(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots) \to (\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

Example

Each insertion or deletion requires $O(\log n)$ time

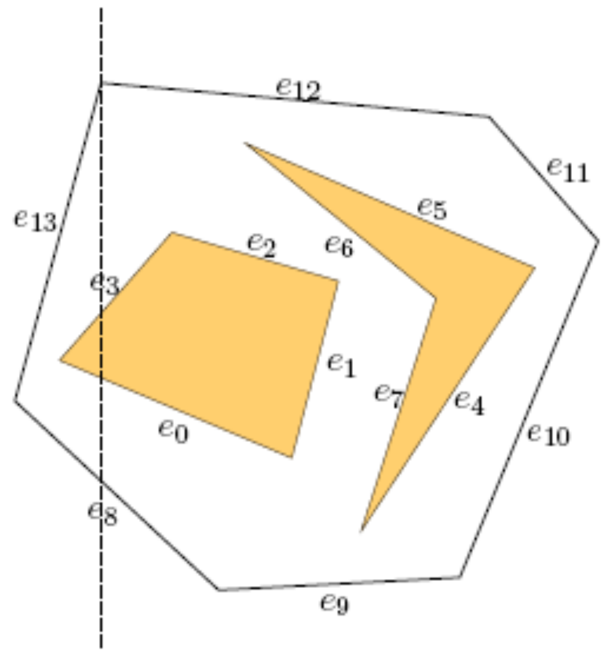


$$L : \{e_8, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{13}\}$$

e_{lower} and e_{upper} are both to the right of the sweep line

- insert e_{lower} and e_{upper} into the list
- $(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots) \rightarrow (\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

Example

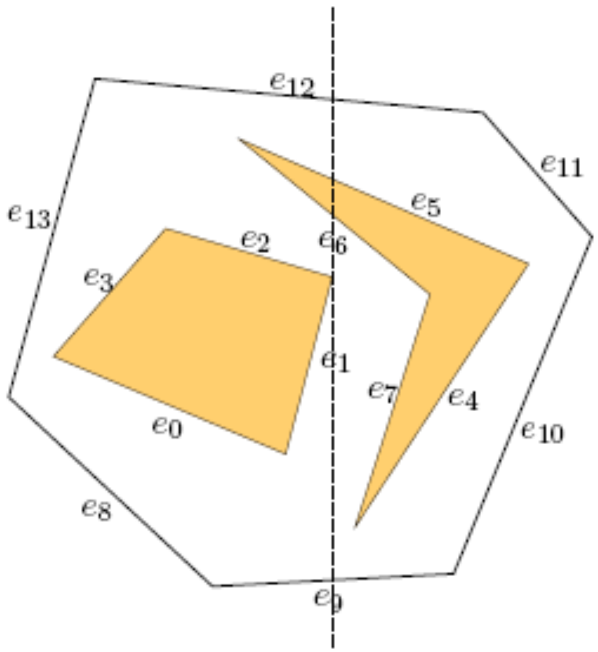


$$L: \{e_8, e_0, e_3, e_{13}\} \rightarrow \{e_8, e_0, e_3, e_{12}\}$$

e_{lower} is to the left and e_{upper} is to the right of the sweep line

- delete e_{lower} from the list and insert e_{upper}
- $(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{UPPER}}, \ldots)$
  $(\ldots, e_{\text{LOWER}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$

Example



$$\{e_9, e_1, e_2, e_6, e_5, e_{12}\} \to \{e_9, e_6, e_5, e_{12}\}.$$

delete e_{lower} and e_{upper} from the list

$(\ldots, e_{\text{LOWER}}, e_{\text{lower}}, e_{\text{upper}}, e_{\text{UPPER}}, \ldots)$
$(\ldots, e_{\text{LOWER}}, e_{\text{UPPER}}, \ldots)$

Trapezoidal Decomposition