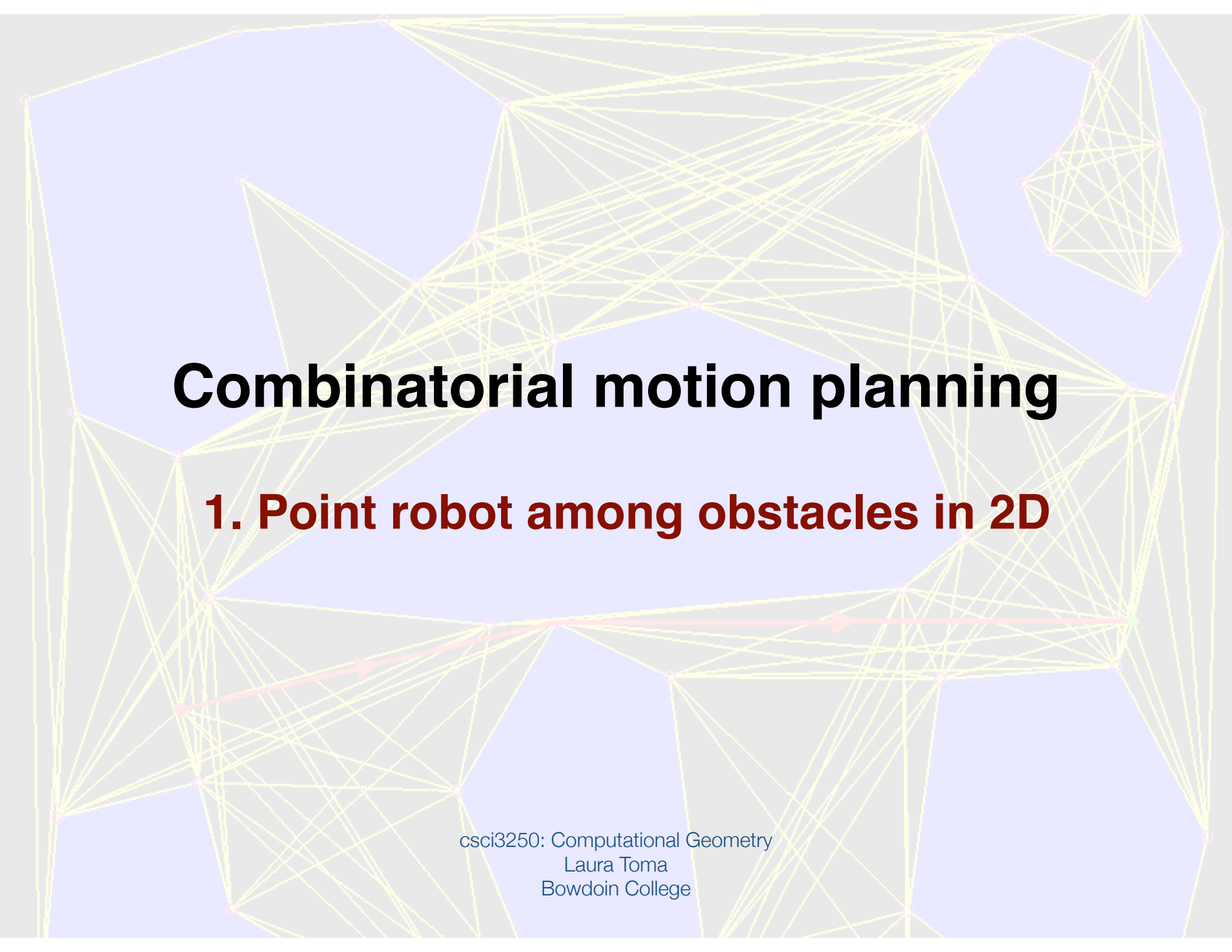
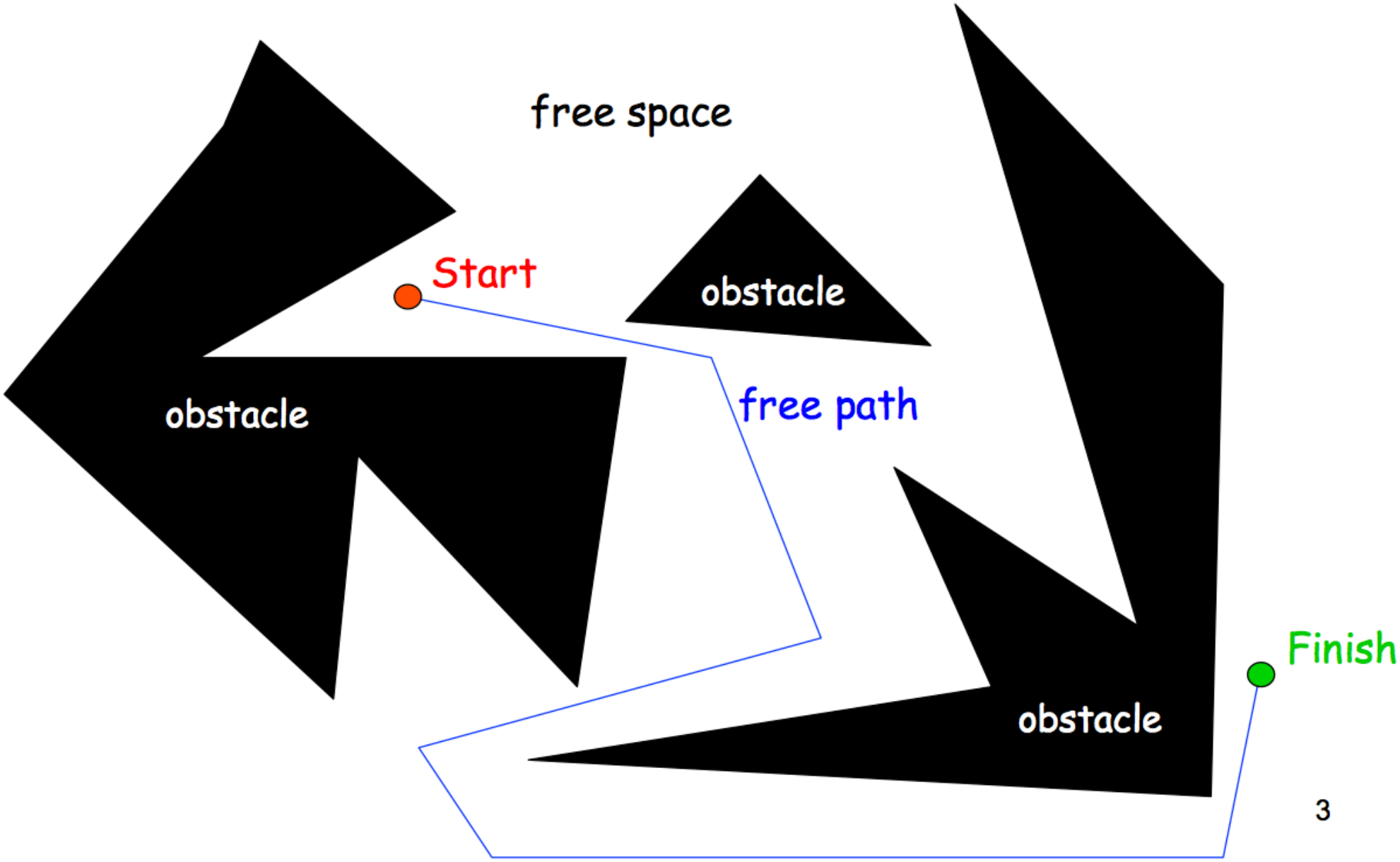


# Combinatorial motion planning

## 1. Point robot among obstacles in 2D

csci3250: Computational Geometry
Laura Toma
Bowdoin College

# Motion planning






screenshot from: ai.stanford.edu/~latombe/cs26n/2012/slides/**point-robot**-bug.ppt

# Motion Planning

Input:

- a robot R
- start and end position
- a set of obstacles S = {$O_1$, $O_2$,…}

Find a path from start to end (that optimizes some objective function).

Parameters:

- physical space (2d, 3d)
- geometry of obstacles (polygons, disks, convex, non-convex, etc)
- geometry of robot (point, polygon, disc)
- robot movement —how many degrees of freedom (dof);  2d, 3d
- objective function to minimize (euclidian distance, nb turns, etc)
- static vs dynamic environment
- exact vs approximate path planning
- known vs unknown map

# Motion Planning

algorithm that finds a path

Ideally we want a **planner** to be complete and optimal.

- A planner is complete: it always finds a path when a path exists

- A planner is optimal: it finds an optimal path (wrt an objective function)

# Path planning problems


- point robot moving among (arbitrary) polygons in 2D
- polygonal robot moving among (arbitrary) polygons in 2D
    - translation only
    - translation+rotation
- ….
- robot with arms and articulation moving in 3D

# Approaches

- Combinatorial (exact)
  - Used for path planning in 2D
  - Idea:
    - Compute an exact representation of free space as a graph
    - Find a path using the graph
- Approximate
  - Used for higher dimensional planning
  - Idea:
    - sample and approximate free space

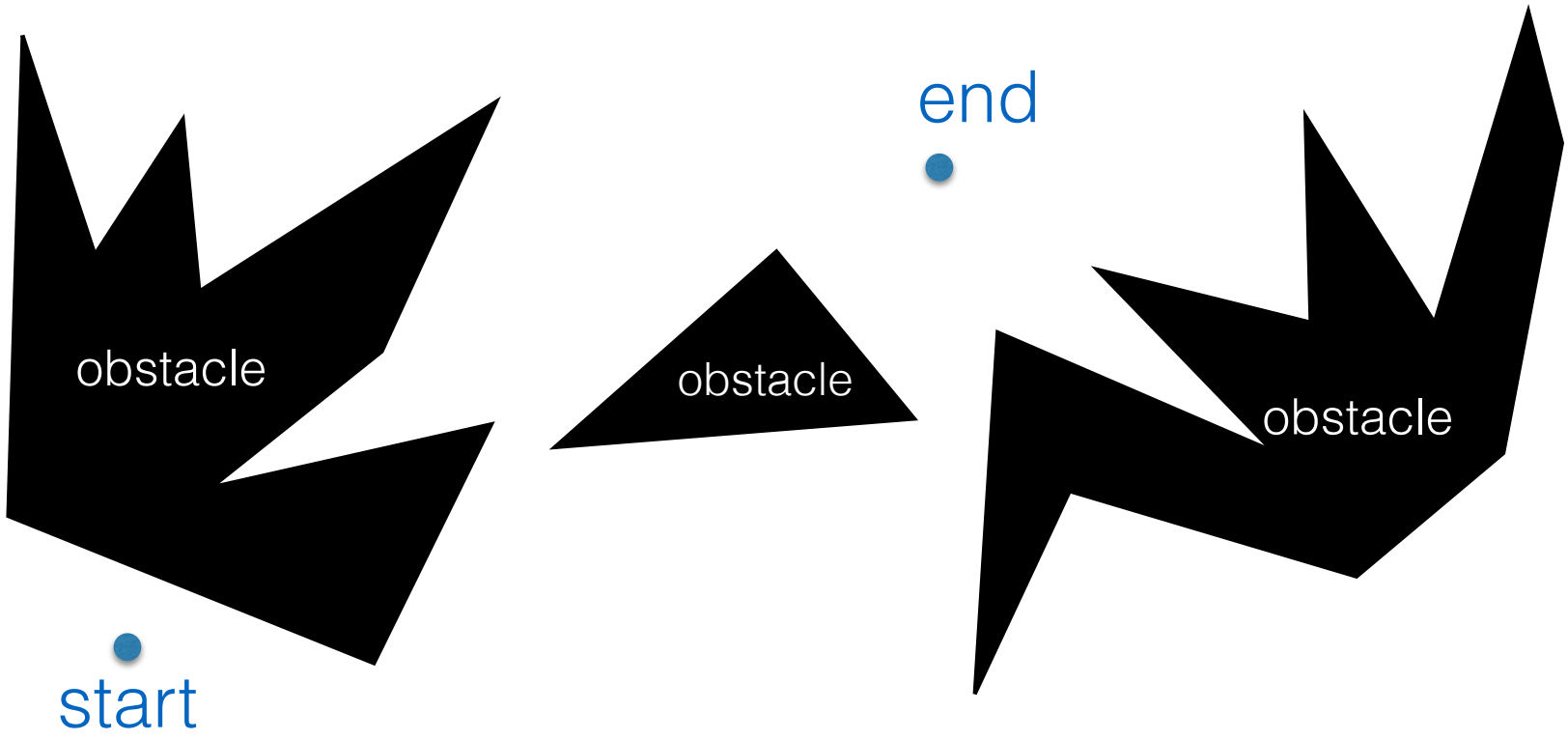
# 1. Point robot moving in 2D

# Point robot in 2D

Input:

- start and end position
- a set of polygonal obstacles $S = \{O_1, O_2, \ldots\}$

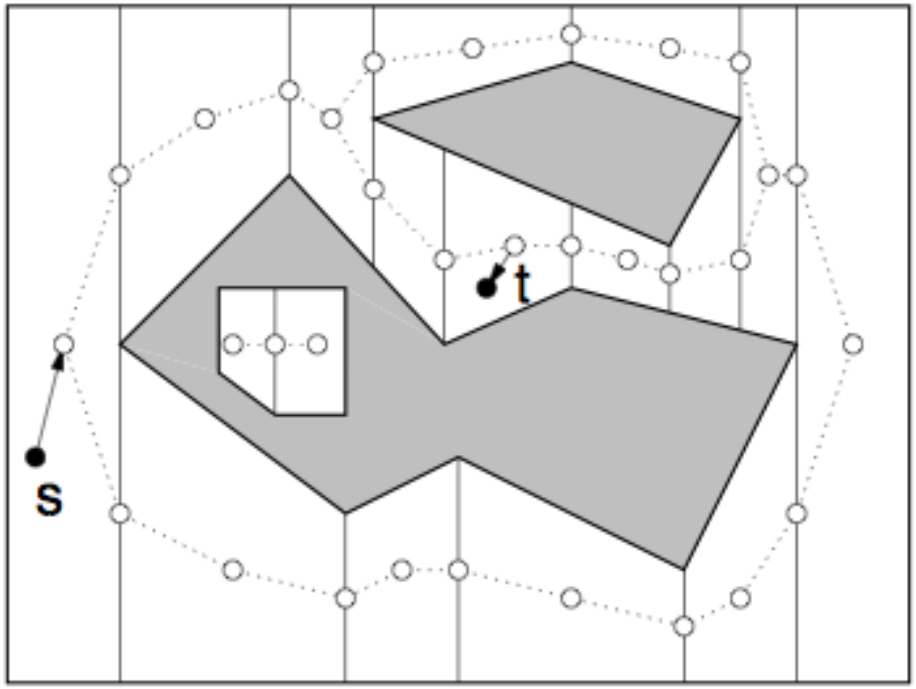
Find a path from start to end.



# Point robot in 2D

General idea

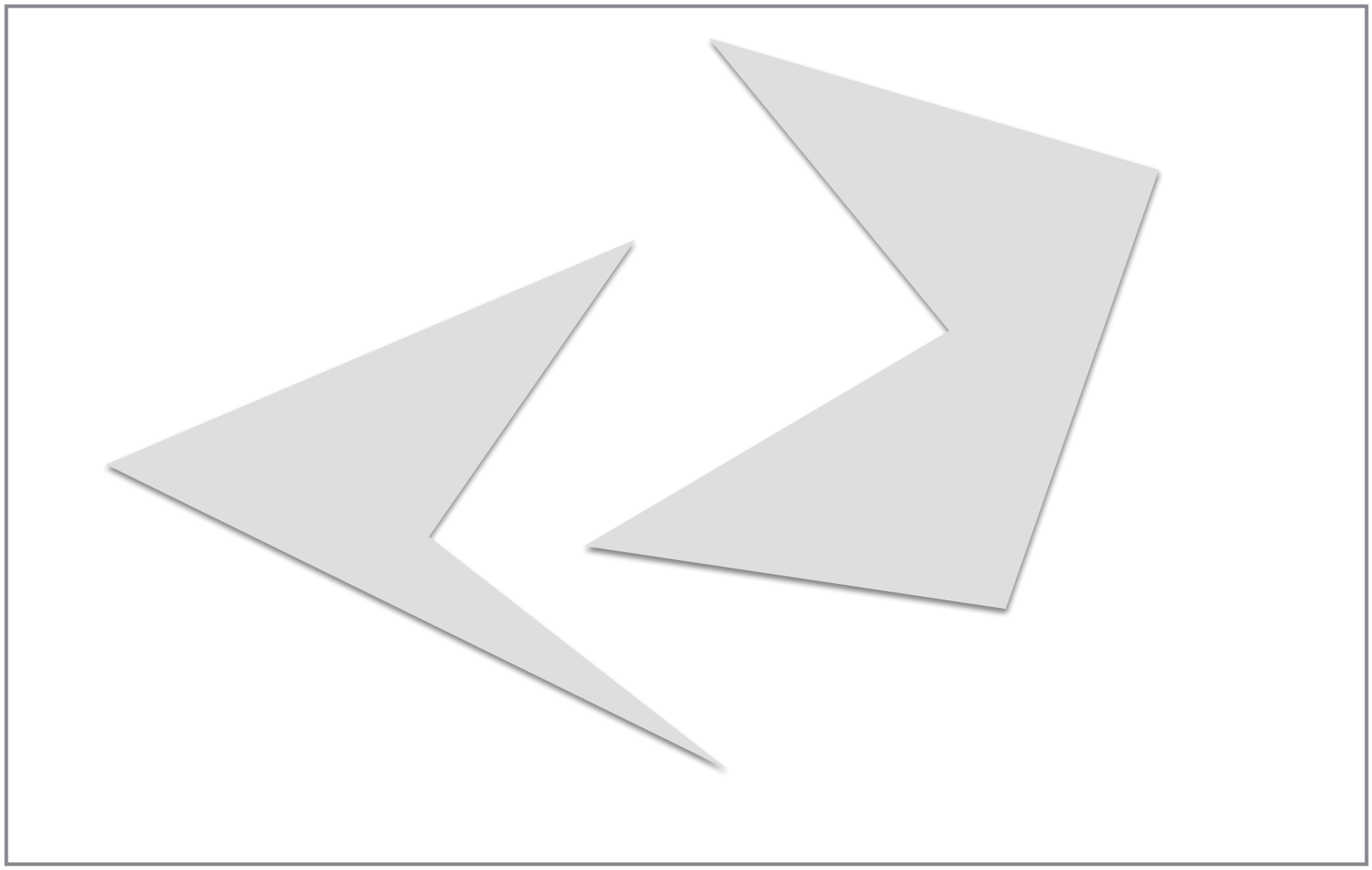
- Build a graph that represents movement through the free space
    - based on trapezoid decomposition of free space
- Search graph to find path




(screenshot from O'Rourke)

- Questions: How? How long? Size?

Let’s consider the following scene. Show a trapezoid decomposition of free space and the corresponding graph (“roadmap”).



# Point robot in 2D

$n$ = complexity of obstacles (total number of edges)

- Compute a trapezoid partition of free space ← Has size $O(n)$ and can be computed in $O(n \lg n)$ time
- Build graph of free space ← Has size $O(n)$ and can be computed in $O(n)$ time
- Search graph to find path ← BFS or DFS in $O(n)$ time

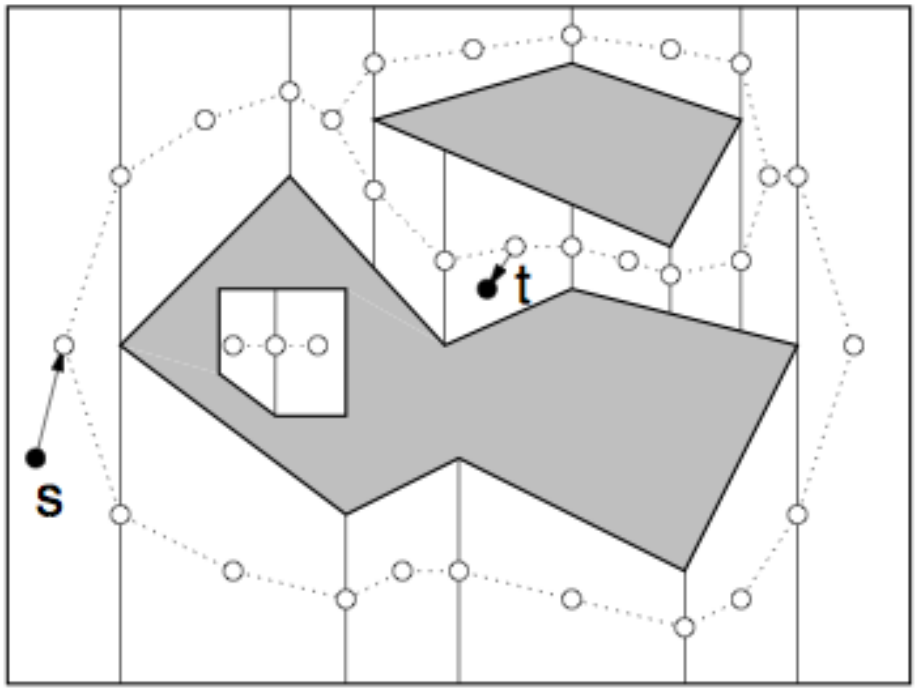
**Result**: Let R be a point robot moving among a set of polygonal obstacles in 2D with n edges in total. We can pre-process the scene in O(n lg n) expected time such that, between any start and goal position, a collision-free path for R can be computed in O(n) time, if it exists.

- Big idea: Path planning for point robot in 2D reduces to graph search in the "free space" graph

# Point robot in 2D

Is this complete?

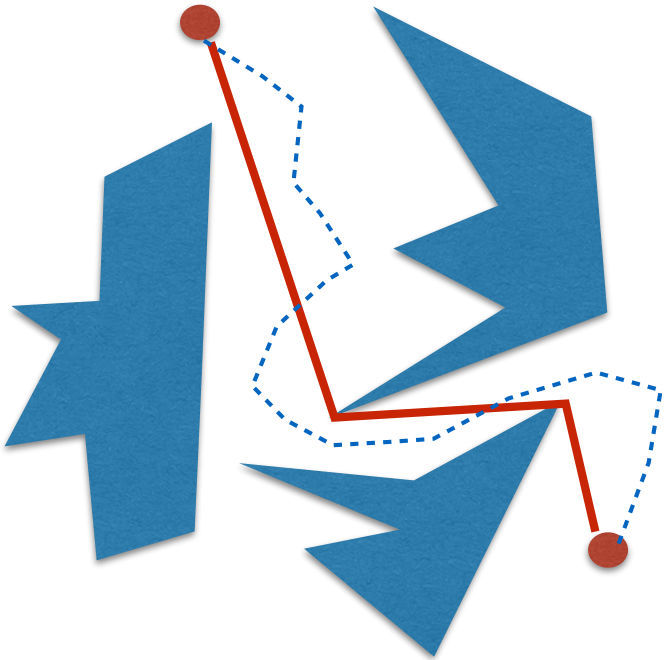
YES

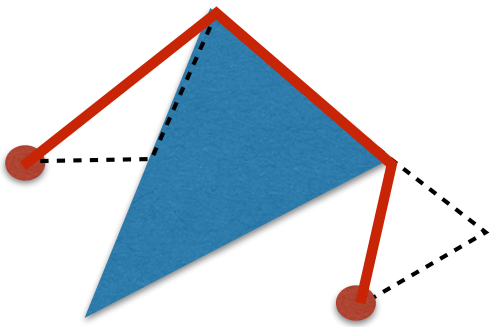



Is this optimal?

No

# What if we wanted an **optimal** path?

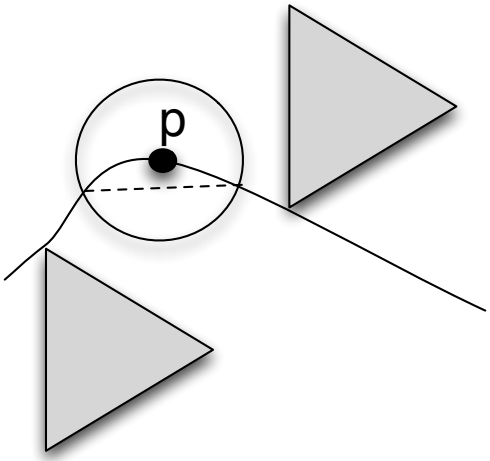


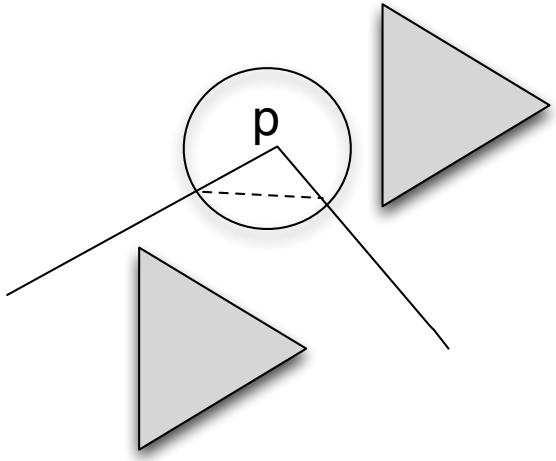


**Theorem**:

Any shortest path among a set S of disjoint polygonal obstacles:

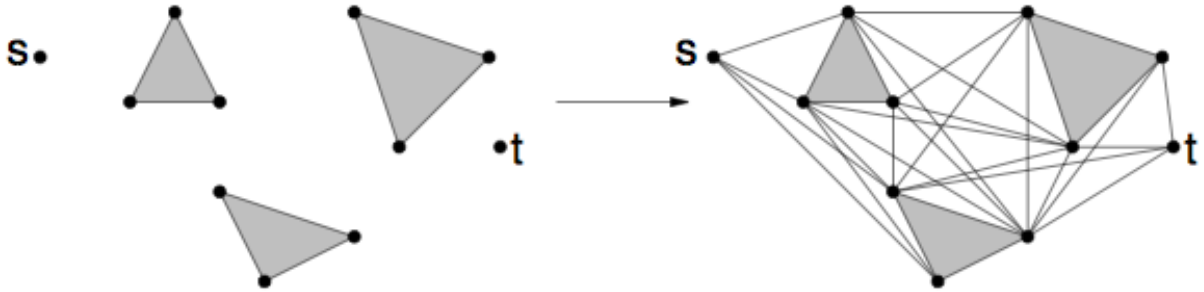
1. is a polygonal path (that is, not curved)
2. its vertices are the vertices of S.



# Visibility graph

- Idea: Since the vertices of any shortest path are the vertices of S, build a graph that represents all possible ways to travel between the vertices of the obstacles
  - V = {set of vertices of obstacles + $p_{start}$ + $p_{end}$}
  - E = {all pairs of vertices $(v_i, v_j)$ such that $v_i v_j$ are visible to each other (and not inside a polygon)
- Claim: any shortest path must be a path in the VG

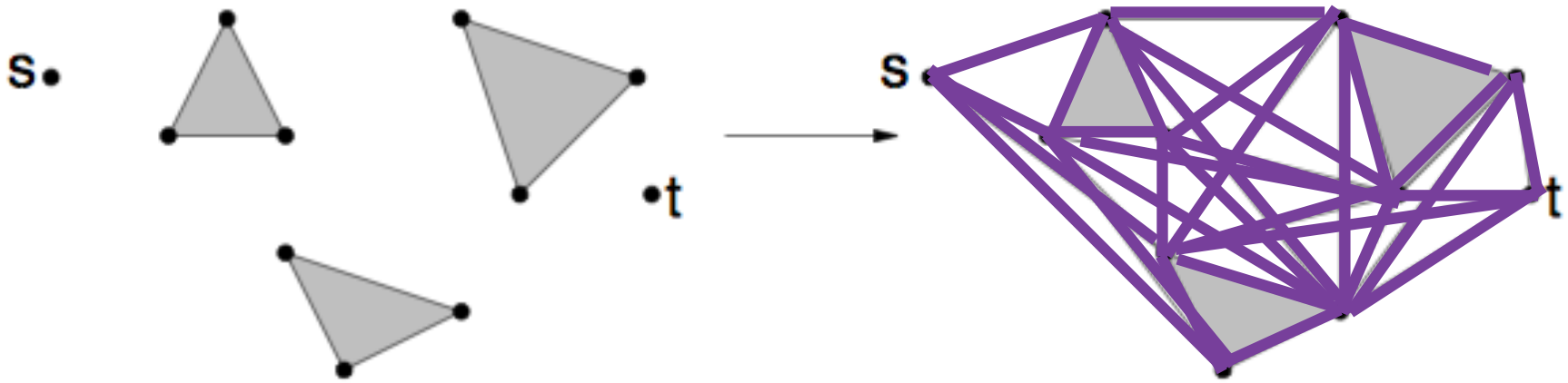


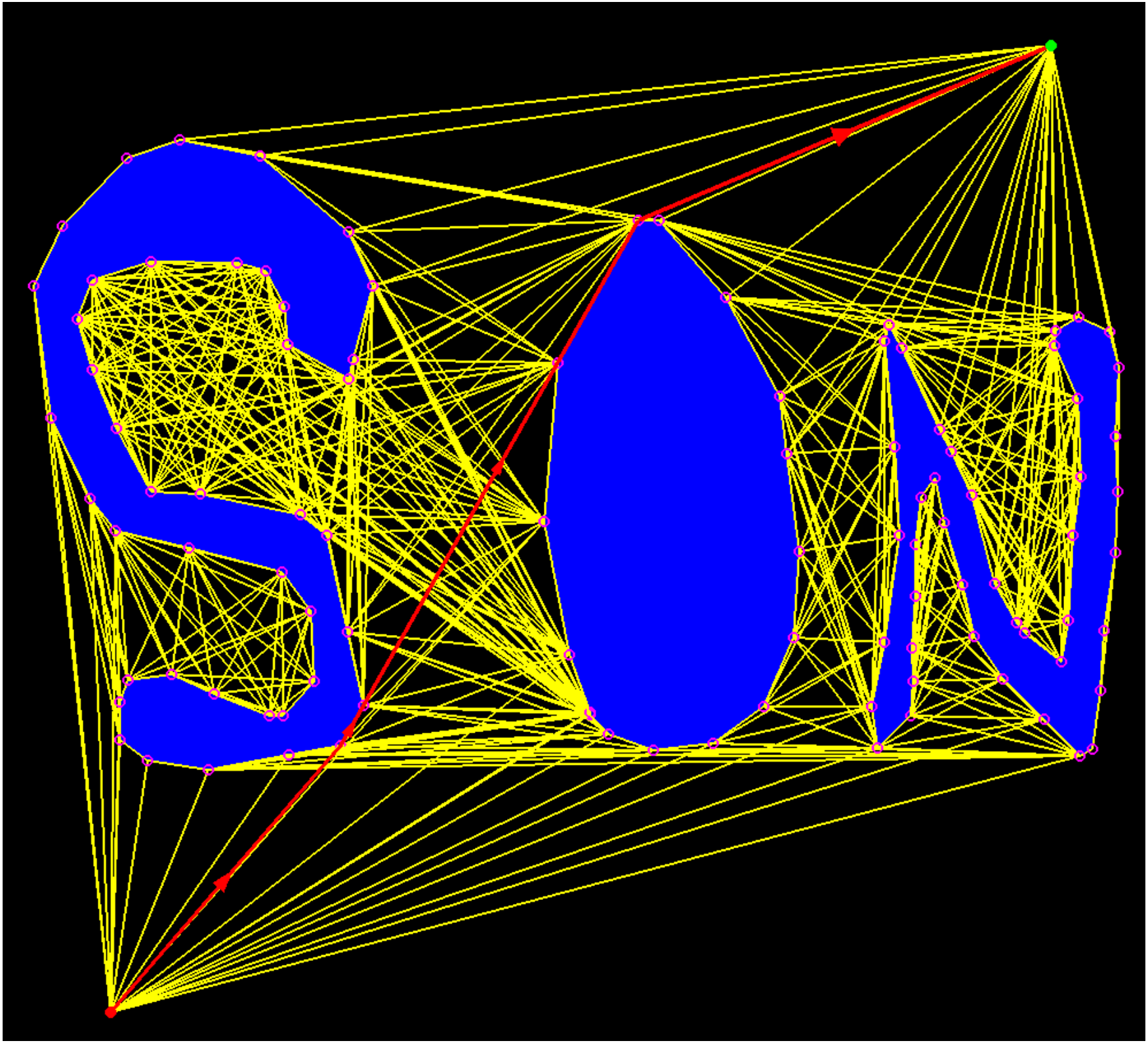

Path planning:

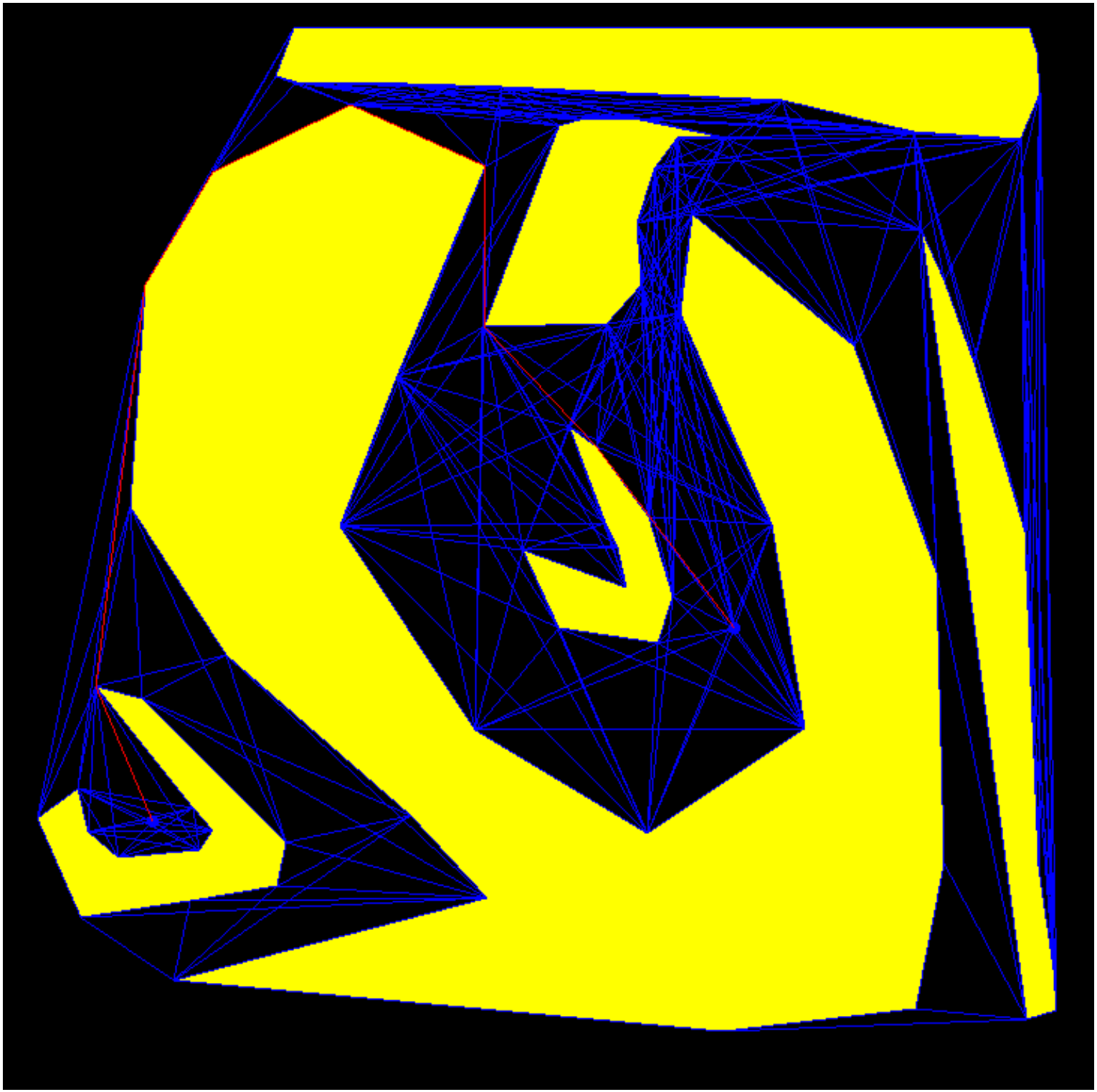
- Compute visibility graph
- SSSP (Dijkstra) in VG from $p_{start}$ to $p_{end}$

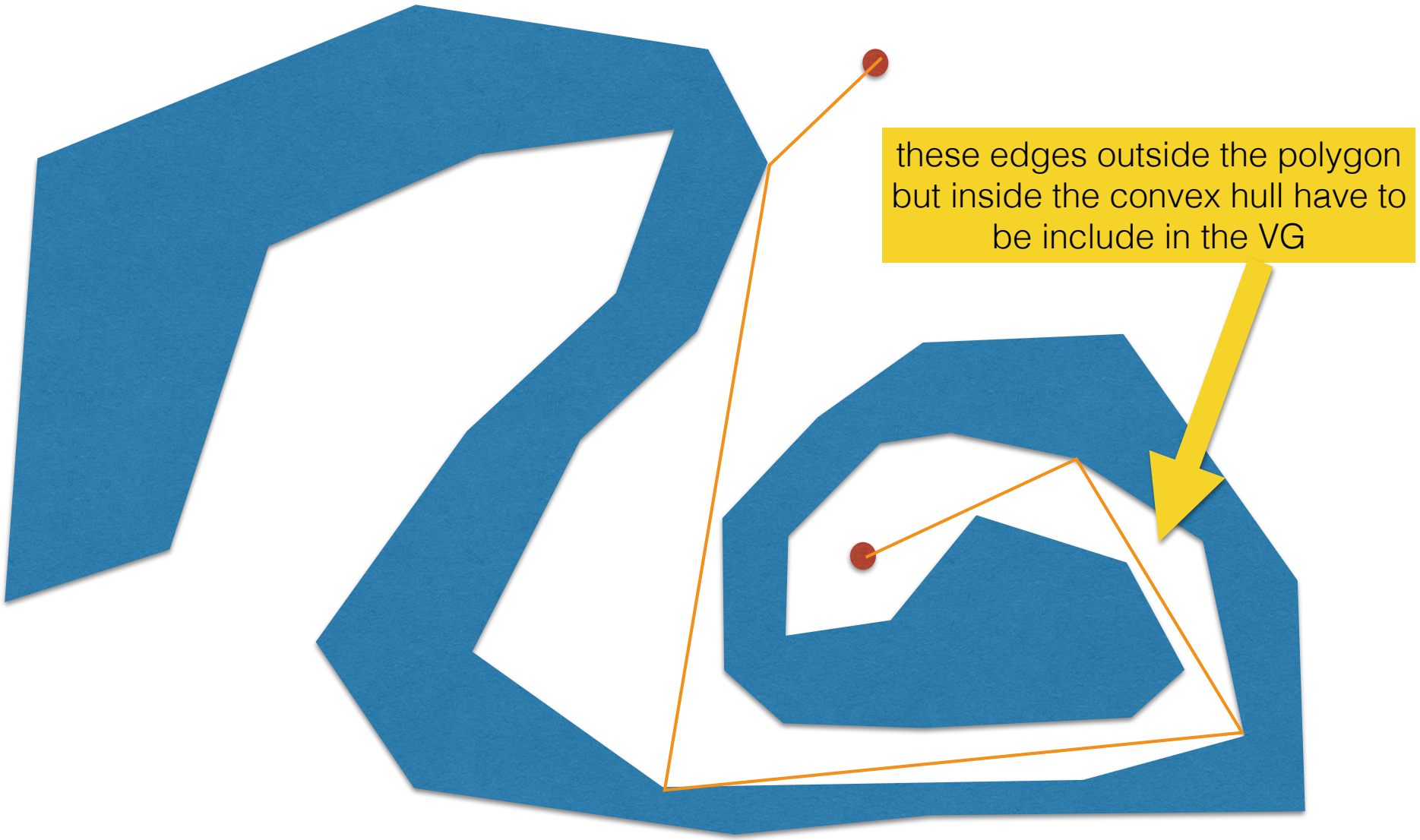
# Visibility graph

- V = {set of vertices of obstacles + $p_{start}$ + $p_{end}$}
- E = {all pairs of vertices $(v_i, v_j)$ such that $v_i v_j$ are visible to each other (and not inside a polygon)








these edges outside the polygon but inside the convex hull have to be include in the VG

$n$ = complexity of obstacles (total number of edges)

# Computing the Visibility Graph

- Straightforward:
  - V = {set of vertices of obstacles + $p_{start}$ + $p_{end}$}
  - for each vertex $u$:
    - for each vertex $v$:
      - if segment $uv$ does not intersect any edges of the polygon properly AND $uv$ is NOT interior to a polygon: add $uv$ as an edge

- Notes:
  - the edges of the polygons must be in the VG
  - interior edges: use `inCone`$(a, b)$ to determine if $b$ is in the cone of $a^-aa^+$

- Running time: $O(n^3)$
- Size of visibility graph:
  - nb of vertices $V$ : $n + 2 = \Theta(n)$
  - nb of edges: $\Omega(n), O(n^2)$

$n$ = complexity of obstacles
(total number of edges)

# Optimal planning for point robot in 2D

Path planning:

- Compute visibility graph ← can have quadratic size
- SSSP (Dijkstra) in VG from $p_{start}$ to $p_{end}$

- Computing the visibility graph
  - $O(n^3)$ straightforward
  - $O(n^2 \lg n)$ improved
- Dijkstra in VG
  - $O(|E| \lg n)$

- Data structures
  - PQ of $(u, dist[u])$ with decreaseKey()
  - for all vertices $u$: $dist[u]$, *pred[u]*, $done[u]$

Dijkstra(vertex s)

- initialize
  - dist[v] = ∞ , pred[v] = null for all v, dist[s] = 0
  - for all v: PQ.insert(<v, dist[v]>)
- while PQ not empty
  - $(u, dist[u])$ = PQ.deleteMin()
  - done[u] = true ← //claim: dist[u] is the shortest path from s to u
  - for each edge (u,v), if v not done:
    - alt = dist[u] + edge(u,v)
    - if alt < dist[v]
      - dist[v] = alt, pred[v] = u, PQ.decreasePriority(v, dist[v])

requires a structure that can search, or a PQueue with additional book-keeping

not all pqueues support it

**Improvement/simplification**

PQ of $(u, dist[u])$ without decreaseKey()

Dijkstra(vertex s)

- initialize
  - dist[v] = ∞ , pred[v] = null for all v, dist[s] = 0
  - PQ.insert(<s, dist[s]>)
- while PQ not empty
  - (u, dist[u]) = PQ.deleteMin()
  - if u not done, for each edge (u,v), if v not done
    - alt = dist[u] + edge(u,v)
    - if alt < dist[v]
      - dist[v] = alt, perd[v]=u, PQ.insert(<v, dist[v]>)
  - done[u]=true

1: insert only the start

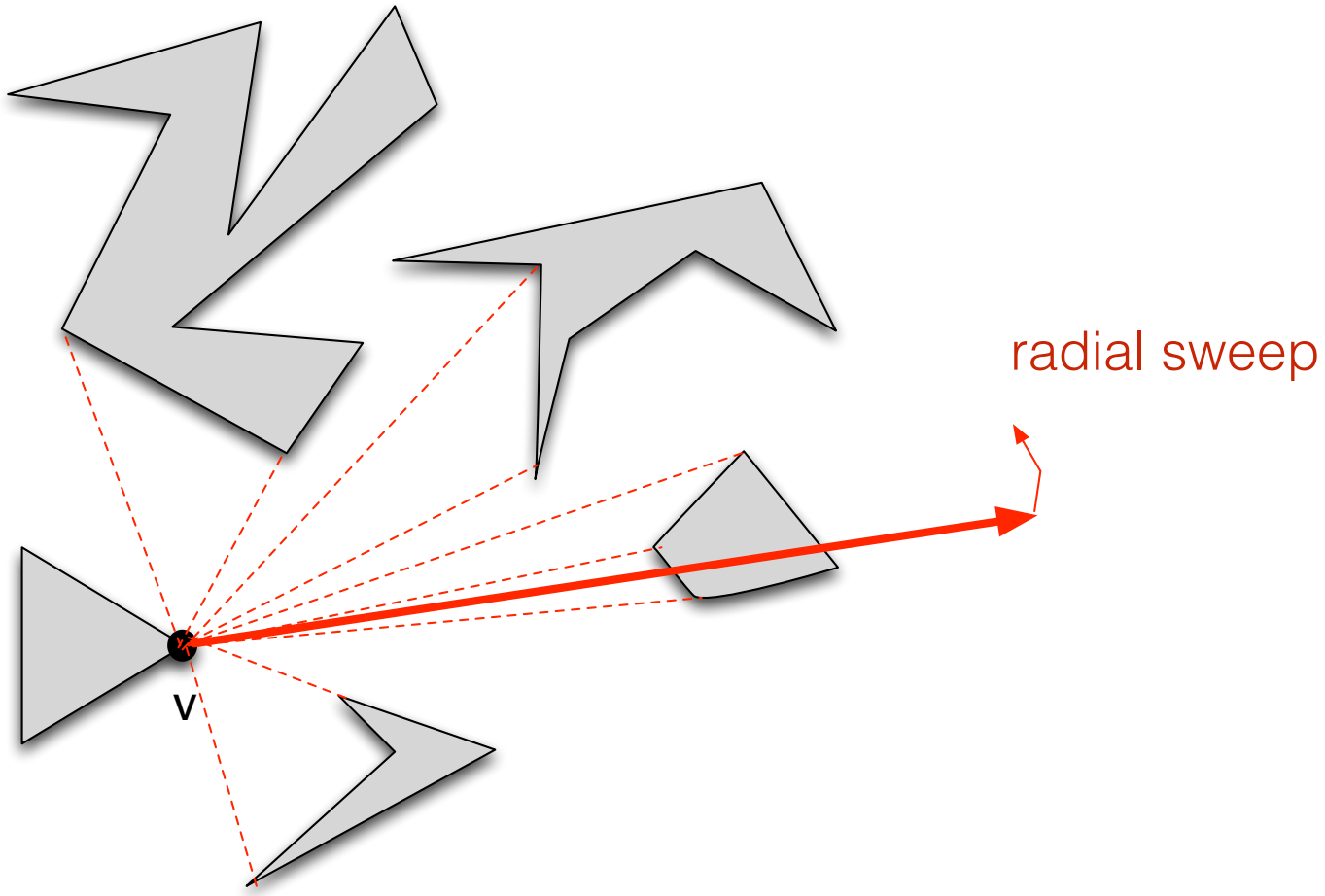
Because we don't decreaseKey, PQ may contain the same vertex with different dist[]. We process u only the first time we see it

2. insert it
(even if it's already there)

Computing the visibility graph in $O(n^2 \lg n)$

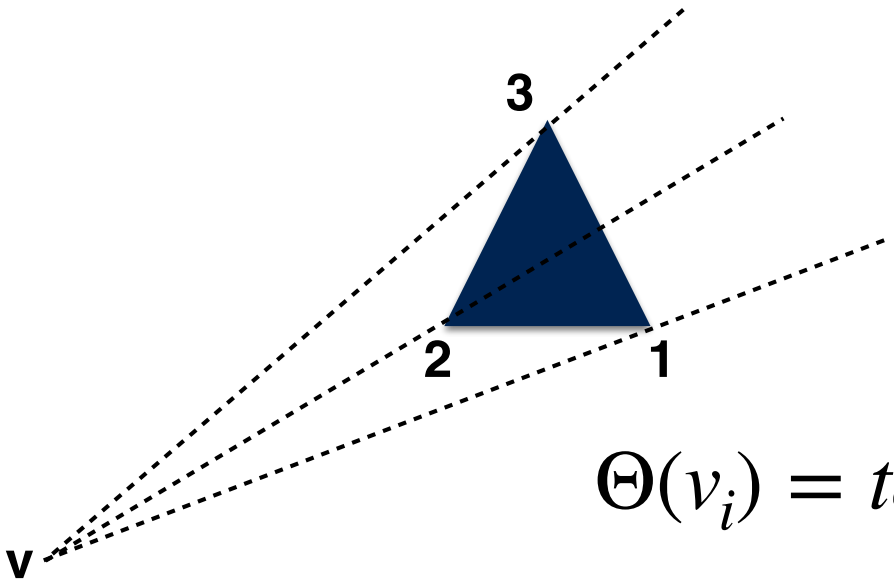
# Improved computation of VG

- For every vertex v:  compute all vertices visible from v in O(n lg n)



# Improved computation of VG

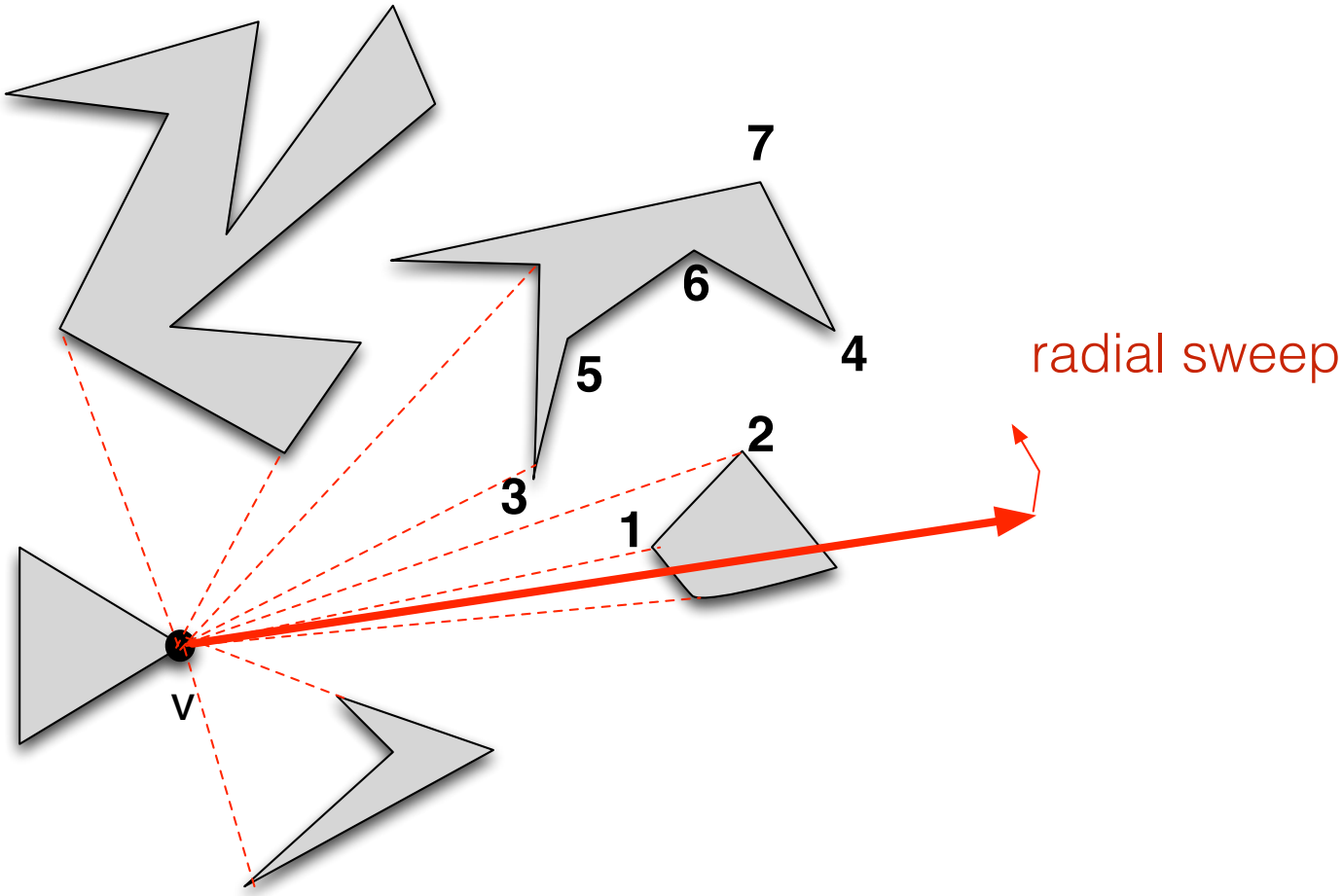
- Radial sweep: rotate a ray centered at v
- Events: vertices of polygons (obstacles) sorted in radial order
    - events of equal angle, sorted by distance from v



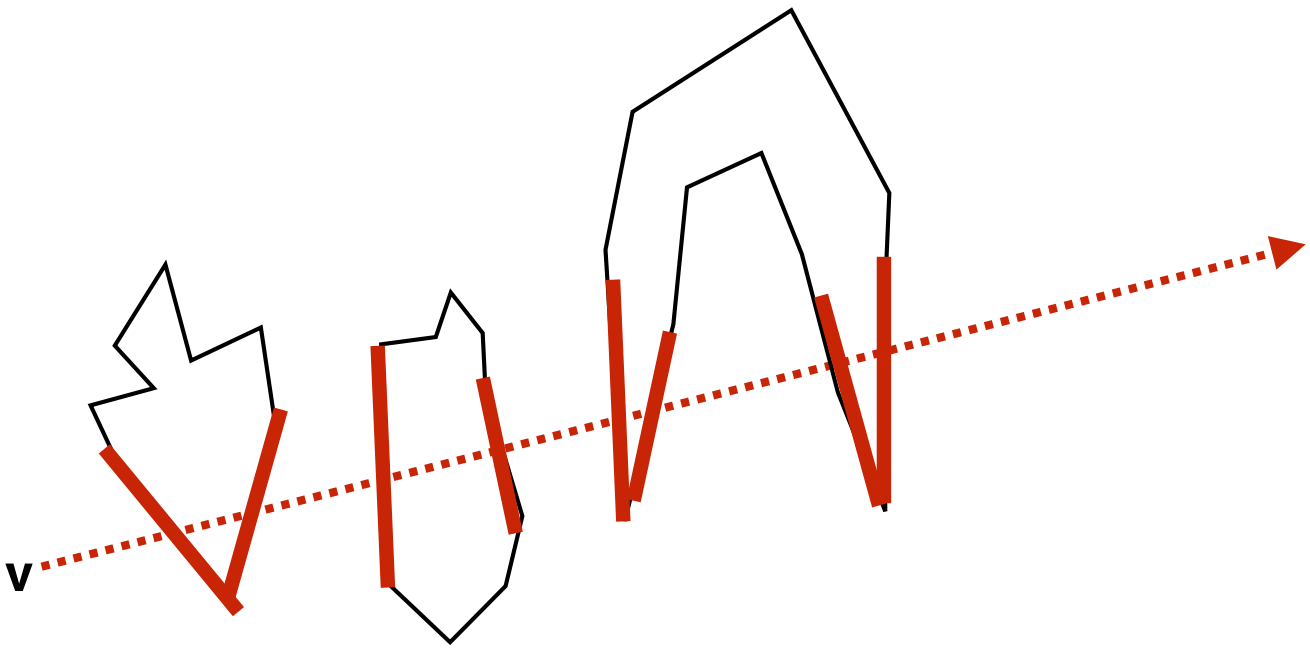

$$\Theta(v_i) = tan^{-1}\frac{y_i - y_v}{x_i - x_v}$$

# VG via line sweep

- Radial sweep: rotate a ray centered at v
- Events: vertices of polygons (obstacles) sorted in radial order
    - events of equal angle, sorted by distance from v



# VG via line sweep




Active structure (AS) stores all the edges that intersect the sweep line, ordered by distance from v

# VG via line sweep

//find all vertices visible from a vertex p

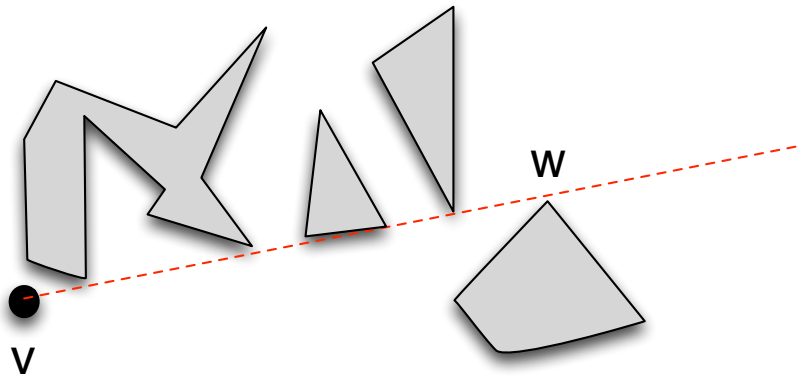
RadialSweep(polygon vertices V, vertex p)

- sort V radially from p, and secondarily by distance from p
- initialize AS with all edges that intersect the horizontal ray from p
- For each vertex v in sorted order:
    - use AS to determine if v is visible from p
    - figure out if the edges incident to v are above/below the sweep line.
      If above -> insert edge in AS. If below => delete edge from AS

Runs in $O(n \lg n)$ time

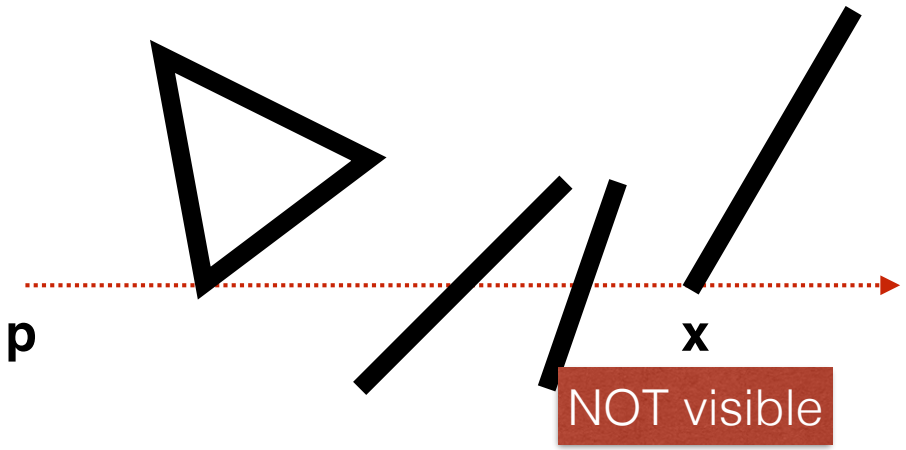
Repeat for all vertices p ==> $O(n^2 \lg n)$

# VG via line sweep

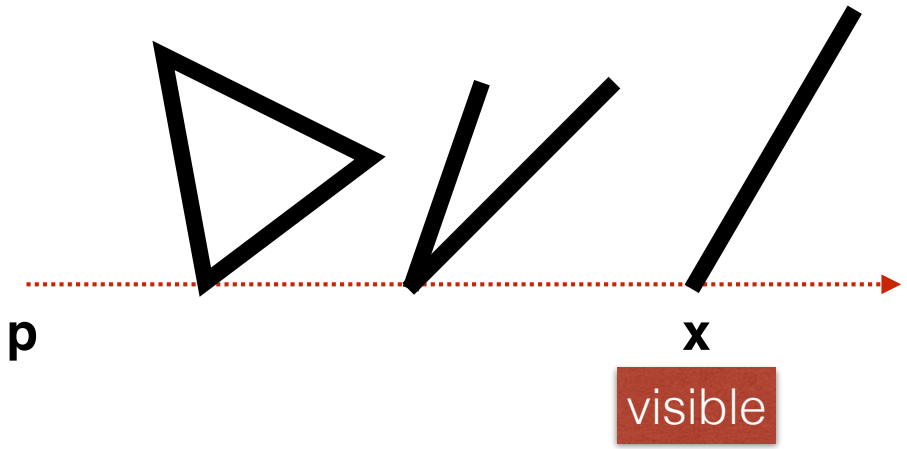



w visible if vw does not intersect the interior of any obstacle

# Is vertex x visible from p? some cases







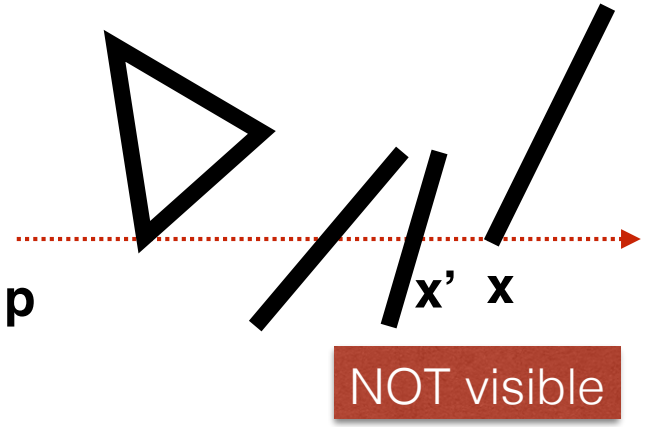

x is NOT visible:

If there is any edge in AS left of x, whose interior intersects the line

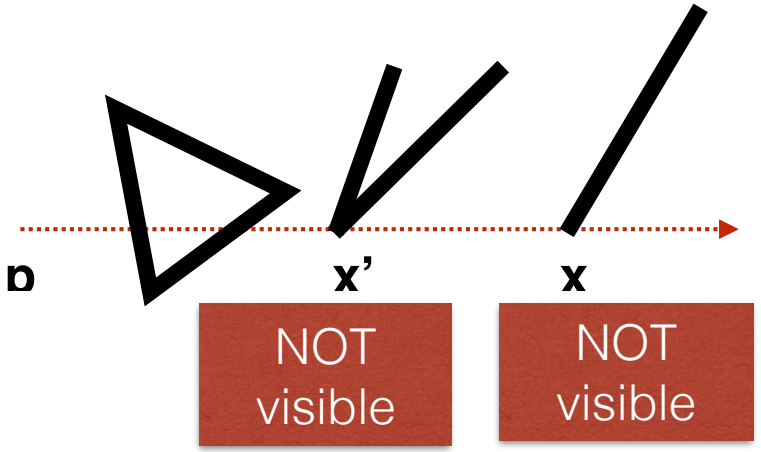
# Is vertex x visible from p?

Let x' be the edge just before x in the AS, x' = AS.predecessor(x)

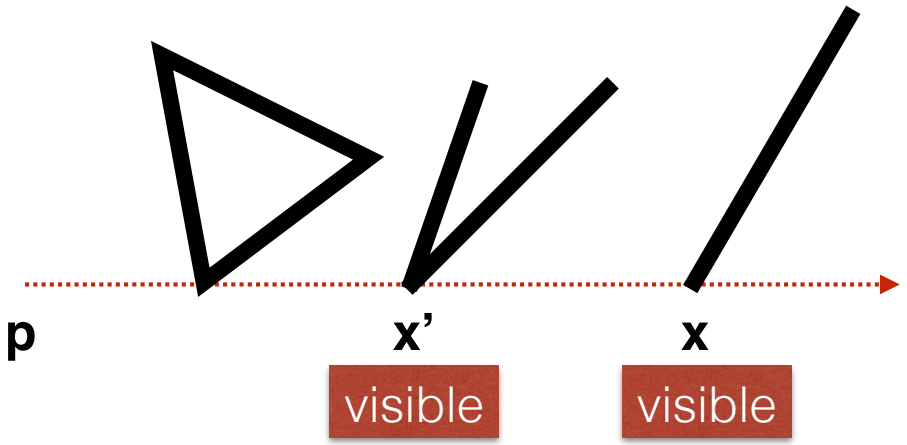
x' interior intersects line




x'is a vertex and it's not visible




x'is a vertex and it's visible

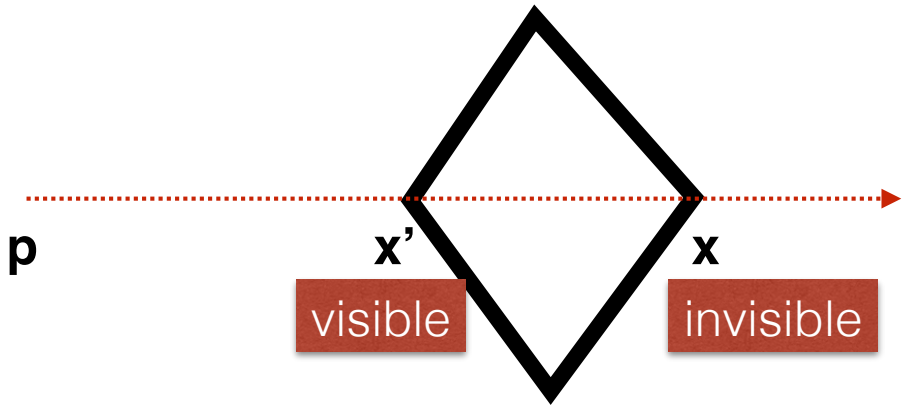


# Is vertex x visible from p?

Let x' be the edge just before x in the AS, x' = AS.predecessor(x)

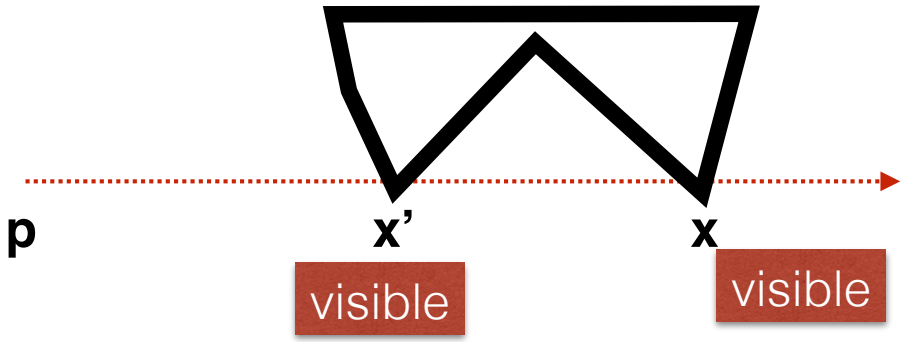
x'is a vertex and it's visible

x, x' part of same polygon




x'is a vertex and it's visible

x, x' part of same polygon



# Is vertex x visible from p?

- check the event just before x in AS. Call this x', x'=AS.predecessor(x)
- if x' is an edge whose interior intersects sweep line => x is not visible
- if x' has a vertex on the sweep line then:
    - if x' is not visible => x not visible
    - if x' is visible => x visible, unless they are both on the same polygon (a few cases to check)

Runs in $O(\lg n)$ time

Computing the visibility graph in $O(n^2 \lg n)$

END

# Recap: Point robot in 2D

- Complete, not optimal
  - Compute the trapezoid decomposition of free space and a graph that represents it in $O(n \lg n)$ time
  - BFS in this graph in $O(n)$ time

- Complete and optimal
  - Compute visibility graph in $O(n^2 \lg n)$
  - Dijkstra in VG in $O(E_{VG} \lg n)$

+ Any shortest path must be a path in VG

+ VG needs to be computed only once, so we can think of it as pre-processing

- VG may be large, so this approach is doomed to $\Omega(n^2)$

# Point robot in 2D

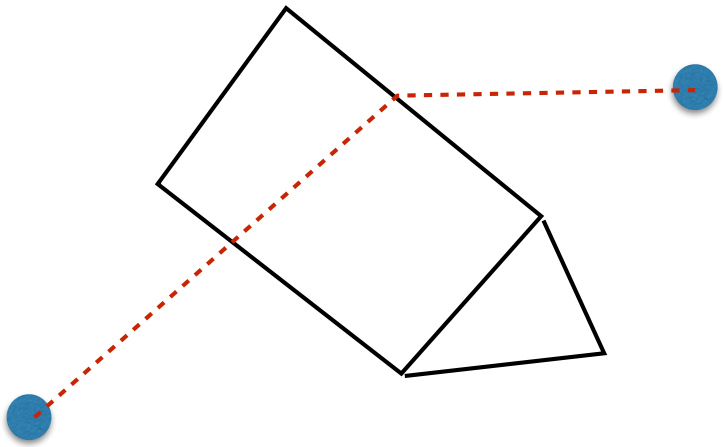
Long history of research and results

- $O(E_{VG} \lg n) = O(n^2 \lg n)$
- Improved to $O(n^2)$
- Quadratic barrier broken by Joe Mitchell: shortest path for a point robot moving in 2D can be computed in $O(n^{1.5+\epsilon})$
- Continuous Dijkstra approach: SP of a point robot moving in 2D can be computed in $O(n \lg n + k)$ [Hershberger and Suri 1993]
- Special cases can be solved faster:
    - e.g. SP inside a simple polygon w/o holes: $O(n)$ time

# Point robot in 3D

**Visibility graph does not generalize to 3D**

- Inflection points of SP are not restricted to vertices of S, can be inside edges



- Shortest paths in 3D much harder
  - Computing 3D shortest paths among polyhedral obstacles is NP-complete
  - Complete and optimal planning in 3D is hopeless

# Path planning in 2D

- point robot moving among arbitrary polygons in 2D


- polygonal robot moving among arbitrary polygons in 2D
  - translation only
  - translation+rotation
  -