Roadmap-Based Path Planning

Chapter 7

Objectives

- Understand the definition of a Road Map
- Investigate techniques for roadmap-based goal-directed path planning in 2D environments
  - geometry-based algorithms that decompose the environment into regions between which robot can travel
  - sampling-based algorithms that choose fixed or random locations in the environment and then interconnect them to form potential paths.
- To understand some issues in applying these algorithms to real robots

What's in Here ?

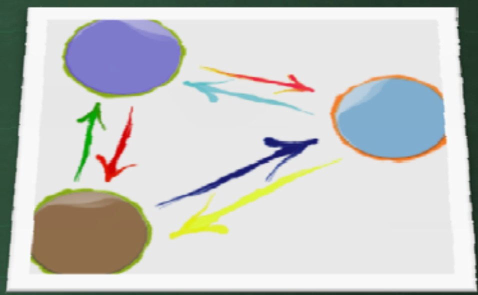
- Road Maps
- Geometry-Based Road Maps
  - **Visibility Graph Paths**
    - Shortest Paths in 2D Among Obstacles
    - Real Robot Shortest Paths
    - Shortest Paths in a Grid
  - **Triangulation Dual Graph Paths**
  - **Generalized Voronoi Diagram Paths**
  - **Cell Decomposition Paths**
    - Trapezoidal Decomposition
    - Boustrophedon Decomposition
    - Canny's Silhouette Algorithm
- Sampling-Based Road Maps
  - **Grid Based Sampling**
  - **Probabilistic Road Maps**
  - **Rapidly Exploring Random Tree Maps**

Road Maps

- A *Road Map* is:
  - a kind of topological map
  - represents a set of paths (or roads) between two points in the environment that the robot can travel on without collision

- Road Maps assume that global knowledge of the environment is available.
- They are commonly used to compute pre-planned paths.
  - i.e., the first step towards goal-directed path planning

Road Maps



- Usually, the set of paths are stored as:
  - a graph of nodes and edges
  - a raster grid (graph is implied by cell arrangement)
- Usually, the graph is pre-computed ahead of time without knowledge of start/goal locations.
  - start/goal locations are given later as a query.
  - a few edges are sometimes added to graph to answer query
- In all cases, the graph is searched to find an efficient (e.g., shortest) path to the goal.
  - usually Dijkstra's algorithm, A* or something similar

Road Map Algorithms

- They are categorized into two main categories:
  - *Geometry-based* algorithms
  - *Sampling-based* algorithms

- Geometry-based algorithms use computational geometry methods to compute nodes and graph edges based on various constraints.
- Sampling-based algorithms select random robot configurations (e.g., points) as nodes and then interconnect them based on some constraints.

Geometry-Based Road Maps

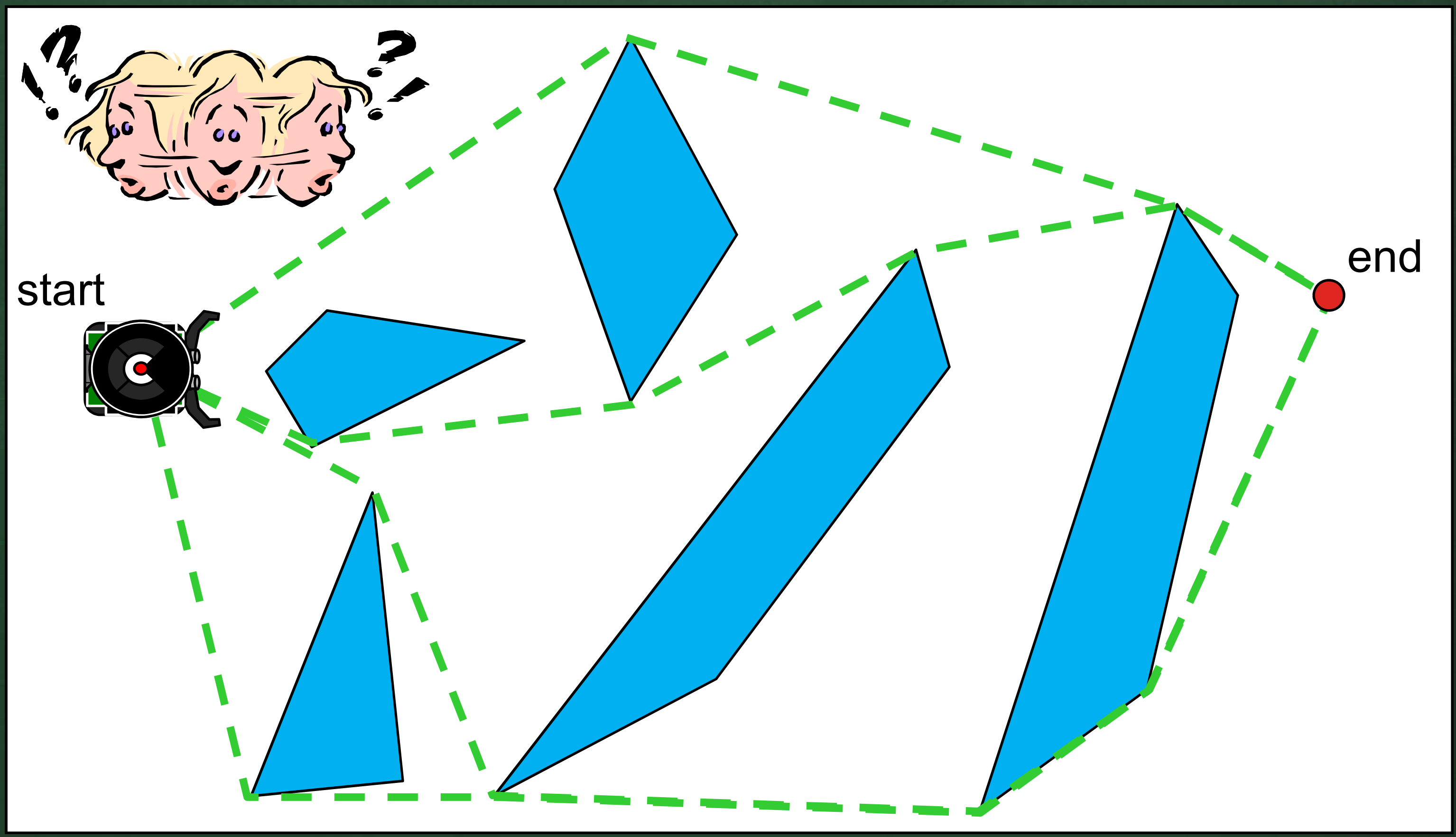
Geometry-Based Road Maps

- There are a few that we will look at based on:
  - Visibility graphs
  - Triangulations dual graphs
  - Generalized Voronoi diagrams
  - Cell decompositions
    - Trapezoidal decompositions
    - Boustrophedon decompositions
    - Canny's Silhouette algorithm

Visibility Graph Paths

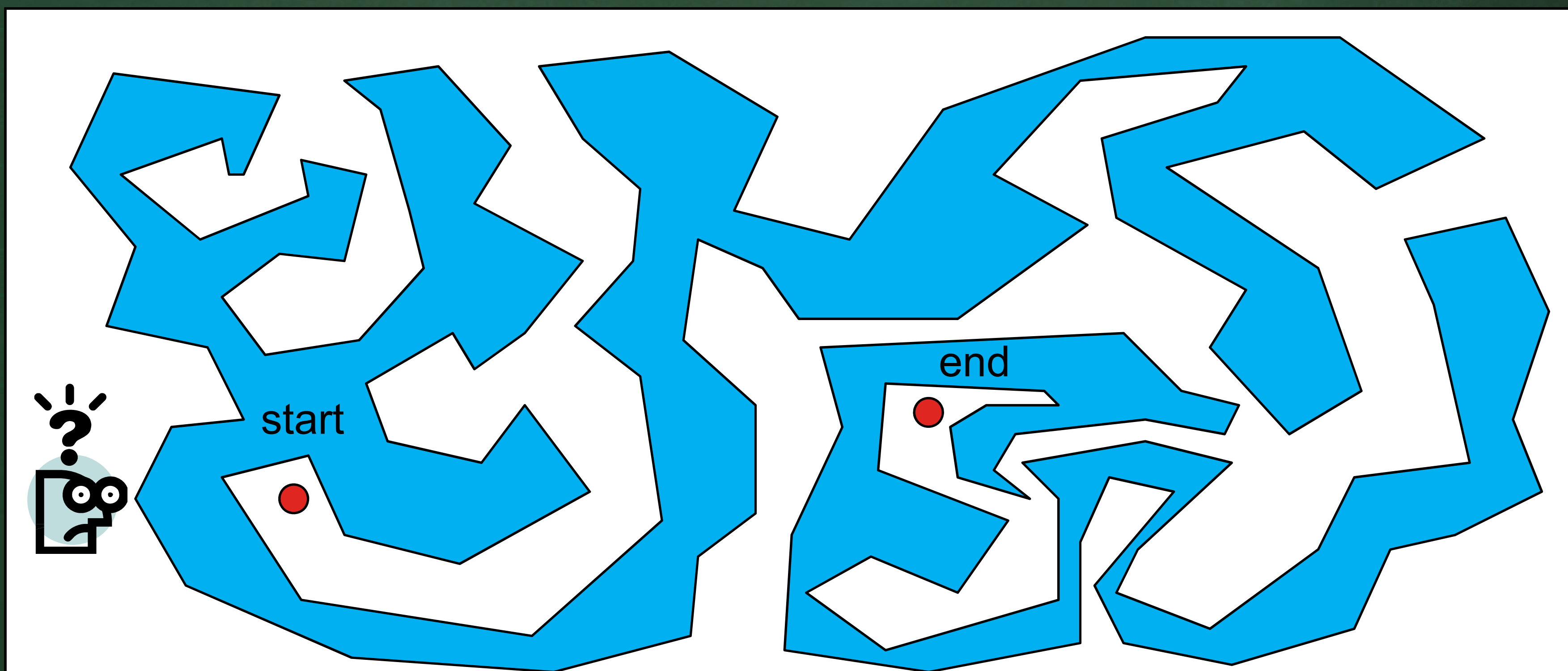
Shortest Path Problem

- How do we get a robot to move efficiently without collisions from one location to another ?



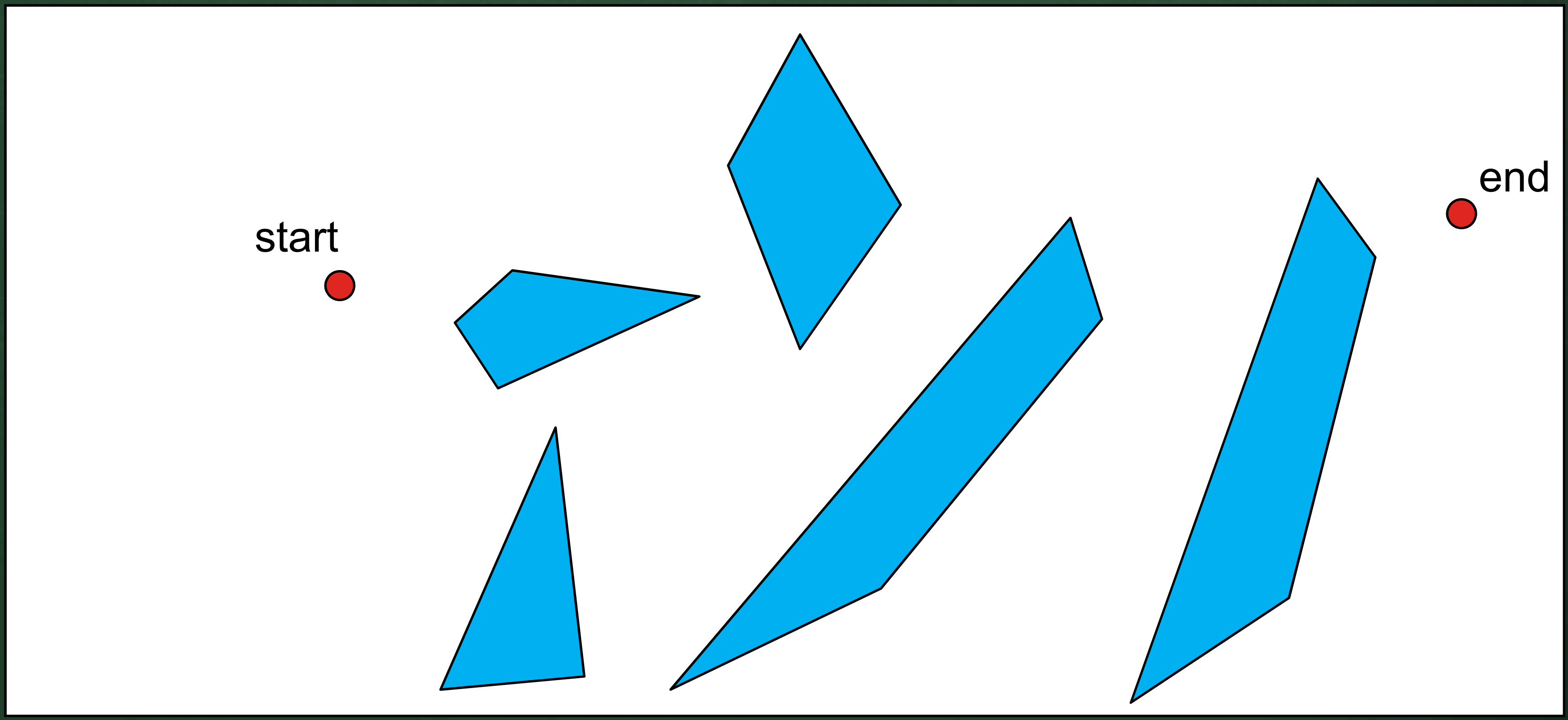
Shortest Path Problem

- Moving without collisions is simple with adequate sensors, but how do we direct it towards a goal ?
- What if the environment is complex ?



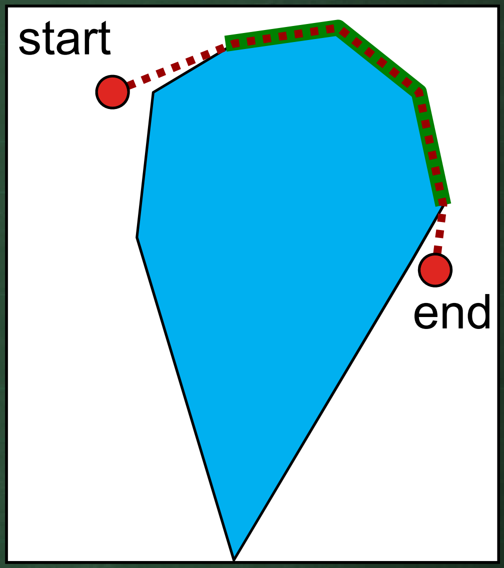
Shortest Path Problem

- Need to examine the map and plan a path
  - Consider simpler problem where robot is a point & obstacles are convex.

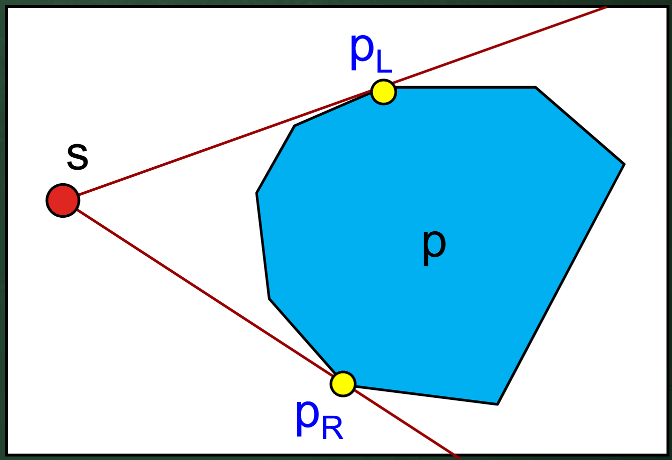


Shortest Path Properties

- Shortest path will travel around obstacles, touching boundaries.



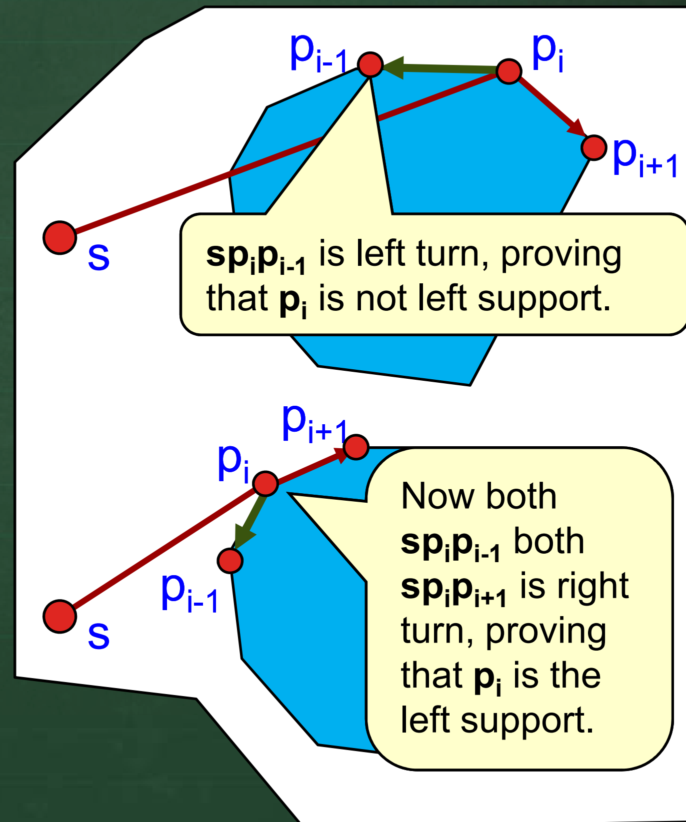
- Consider the robot standing at point *s*.
- Determine support lines of polygon *p*:
  - A *support line* is a line intersecting *p* such that *p* lies completely on one side of that line.
  - exactly two called *left support* and *right support* lines.
  - defined by 2 vertices of *p* called left & right *support vertices* (p_L & p_R)



Shortest Path Properties

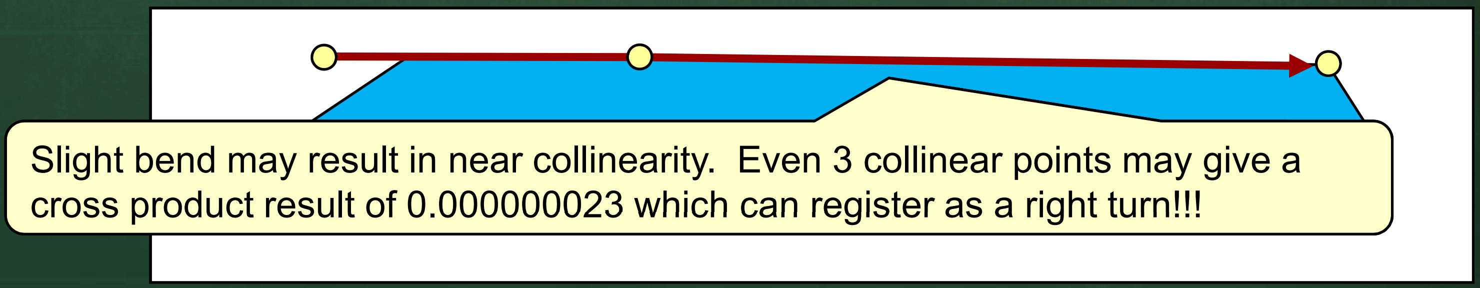
- Can find p_L and p_R by checking each vertex using left/right turn test:
  - For convex polygons, $p_i = p_L$ (resp. p_R) if both sp_ip_{i-1} and sp_ip_{i+1} are right (resp. left) turns.
  - Just compute:

```
t1 = (pix-sx)(pi+1y-sy)-(piy-sy)(pi+1x-sx)
t2 = (pix-sx)(pi-1y-sy)-(piy-sy)(pi-1x-sx)
IF ((t1 < 0) AND (t2 < 0)) THEN pL = pi
IF ((t1 > 0) AND (t2 > 0)) THEN pR = pi
```



Shortest Path Properties

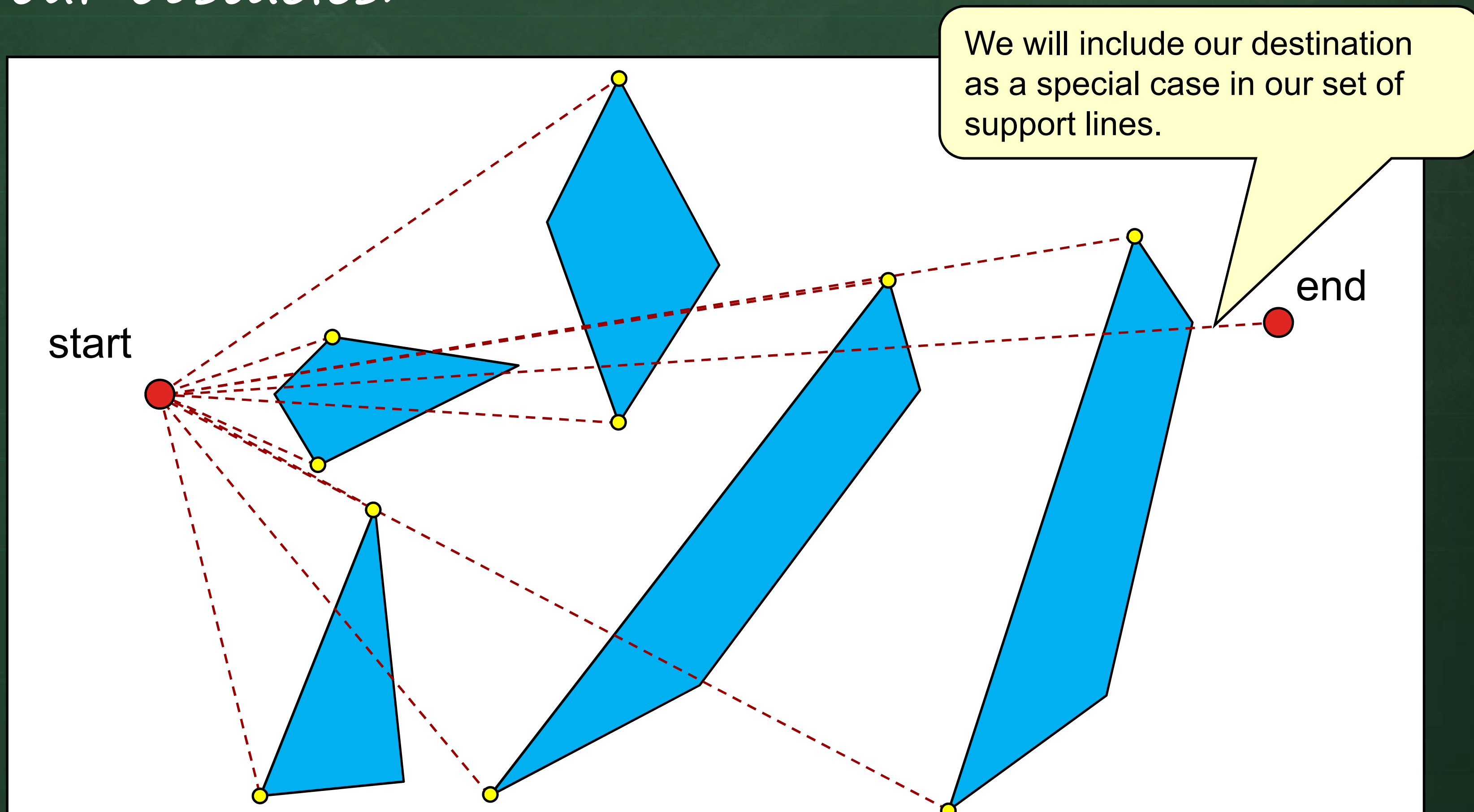
- This support-finding algorithm can take O(n) time but it is practical for small polygons.
- A more efficient algorithm can use a binary search for the left/right support vertices in O(log n) time. Can YOU do this ?
- There are some numerical issues with collinearity:



- May have to allow for computational margins.

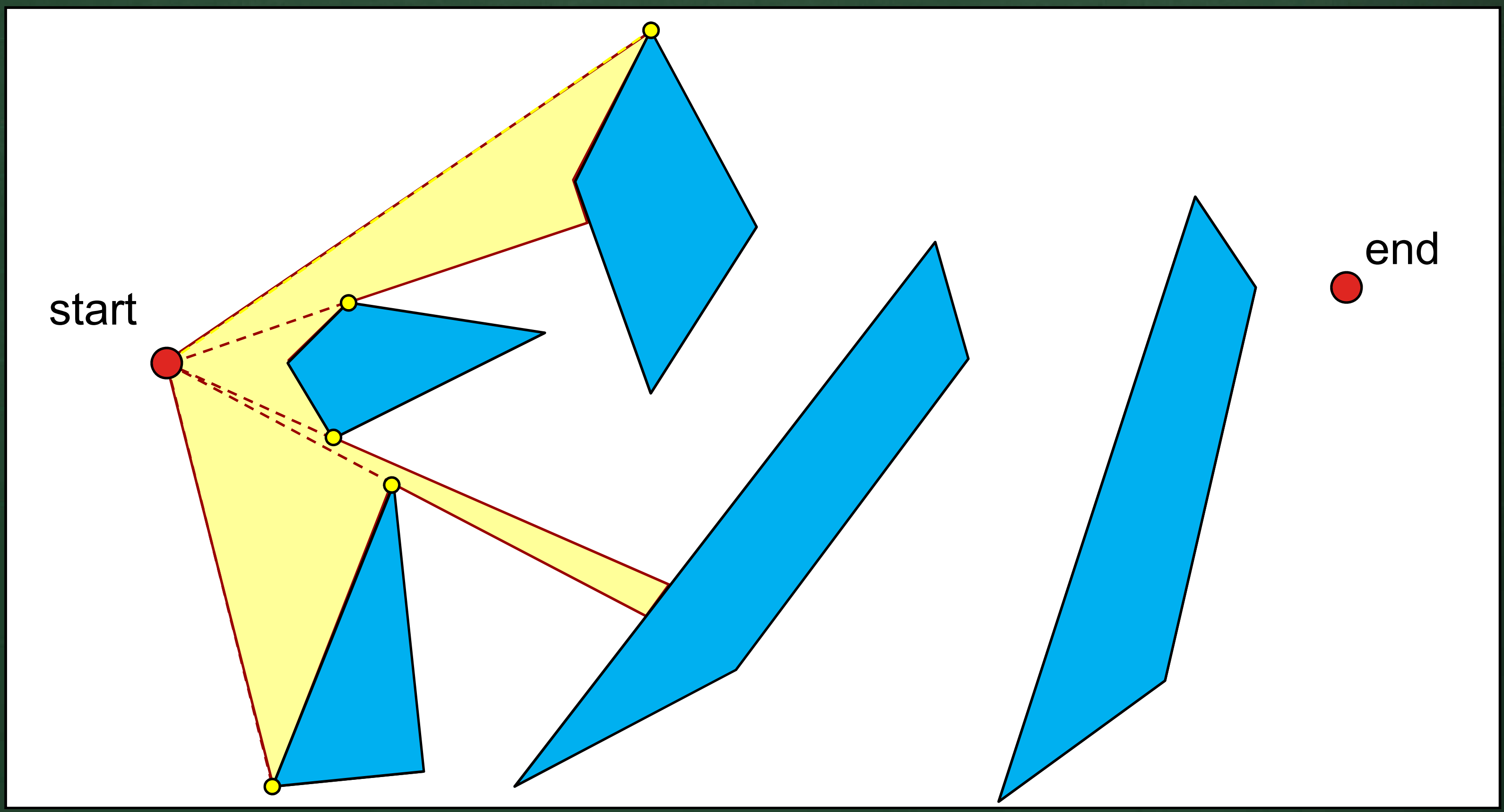
Shortest Path Algorithm

- We can now apply this by finding all support vertices of our obstacles:



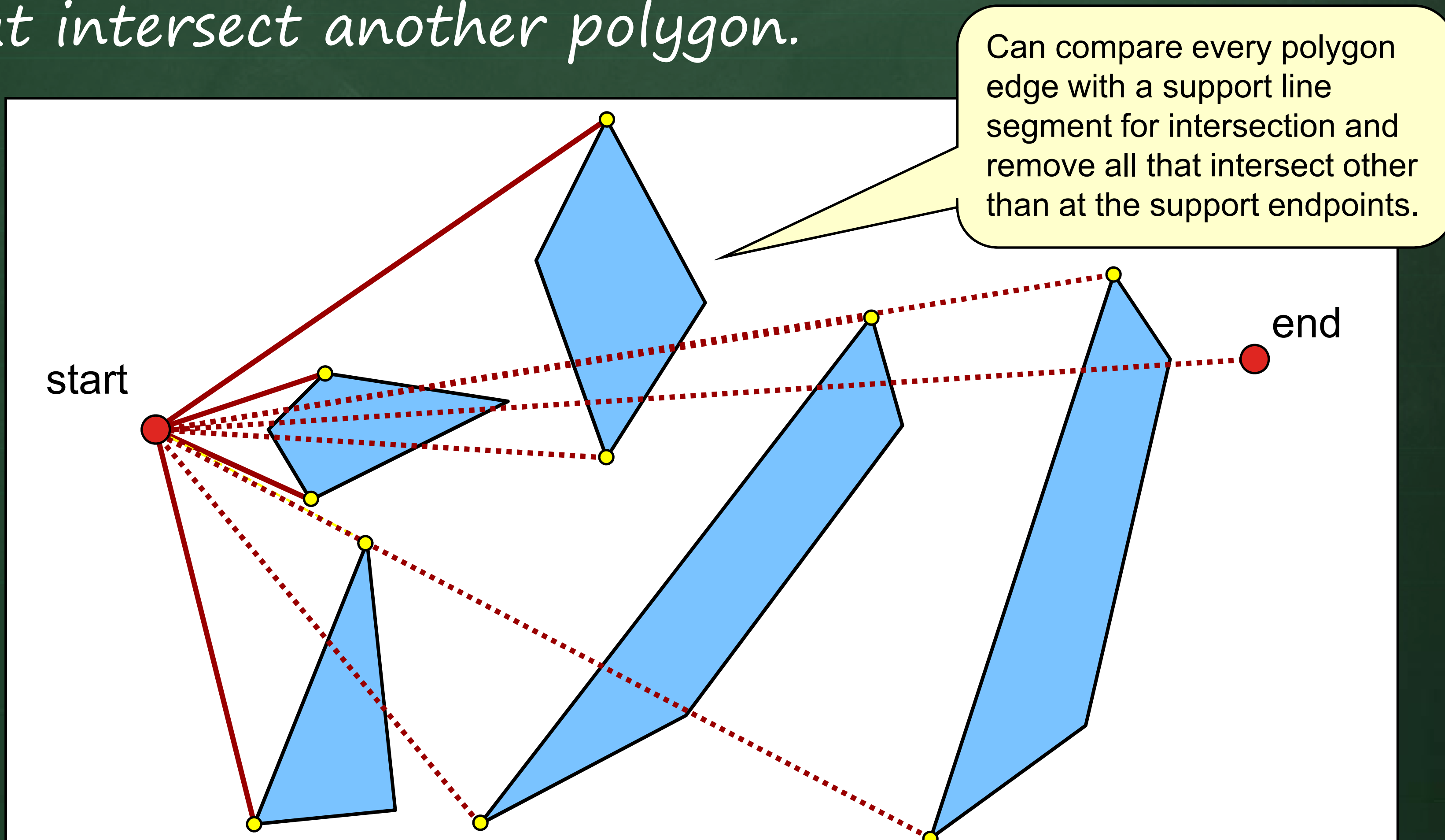
Shortest Path Algorithm

- Now determine which support lines represent valid paths for the robot to travel (i.e., the *visible* support vertices):



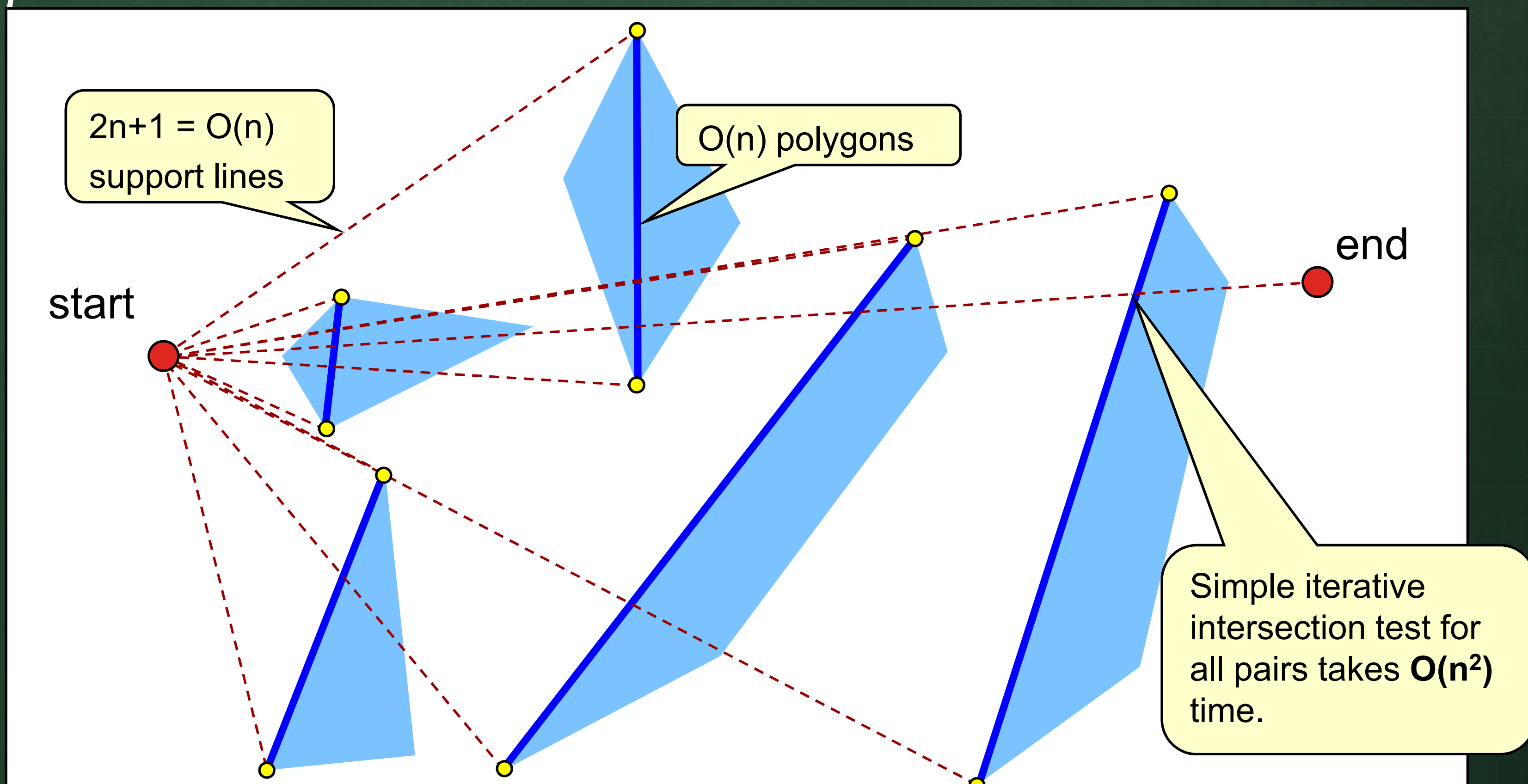
Shortest Path Algorithm

- Do this by eliminating any support line segments that intersect another polygon.



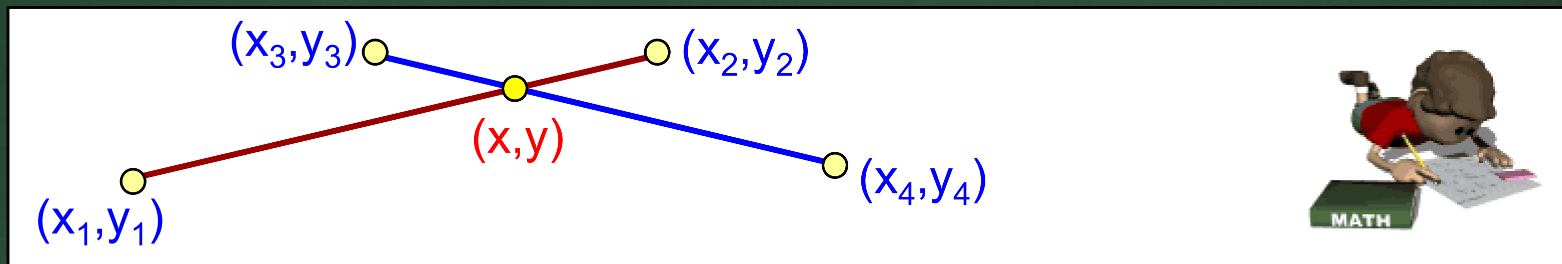
Shortest Path Algorithm

- Since obstacles are convex, it is enough to compare support lines against line segments joining polygon support vertices:



Line Intersection test

- How do we check for line-segment intersection ?



- Can use well-known equation of a line:

$$y = m_a x + b_a$$

$$y = m_b x + b_b$$

where

$$m_a = (y_2 - y_1) \,/\, (x_2 - x_1)$$

$$m_b = (y_4 - y_3) \,/\, (x_4 - x_3)$$

$$b_a = y_1 - x_1 m_a$$

$$b_b = y_3 - x_3 m_b$$

Must handle special case where lines are vertical. (i.e., $\mathbf{x_1} = \mathbf{x_2}$ or $\mathbf{x_3} = \mathbf{x_4}$)

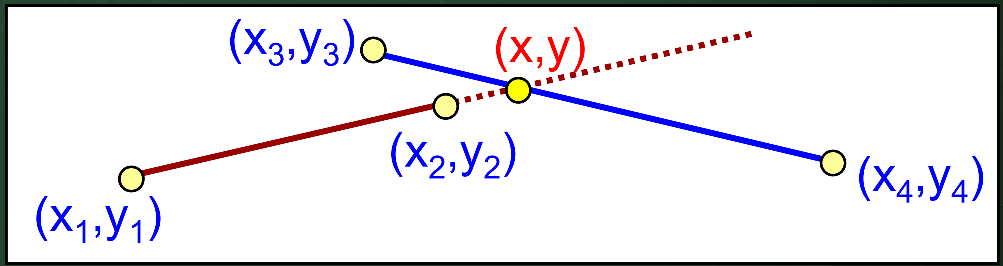
Line Intersection test

- Intersection occurs when these are equal:

  $m_a x + b_a = m_b x + b_b$

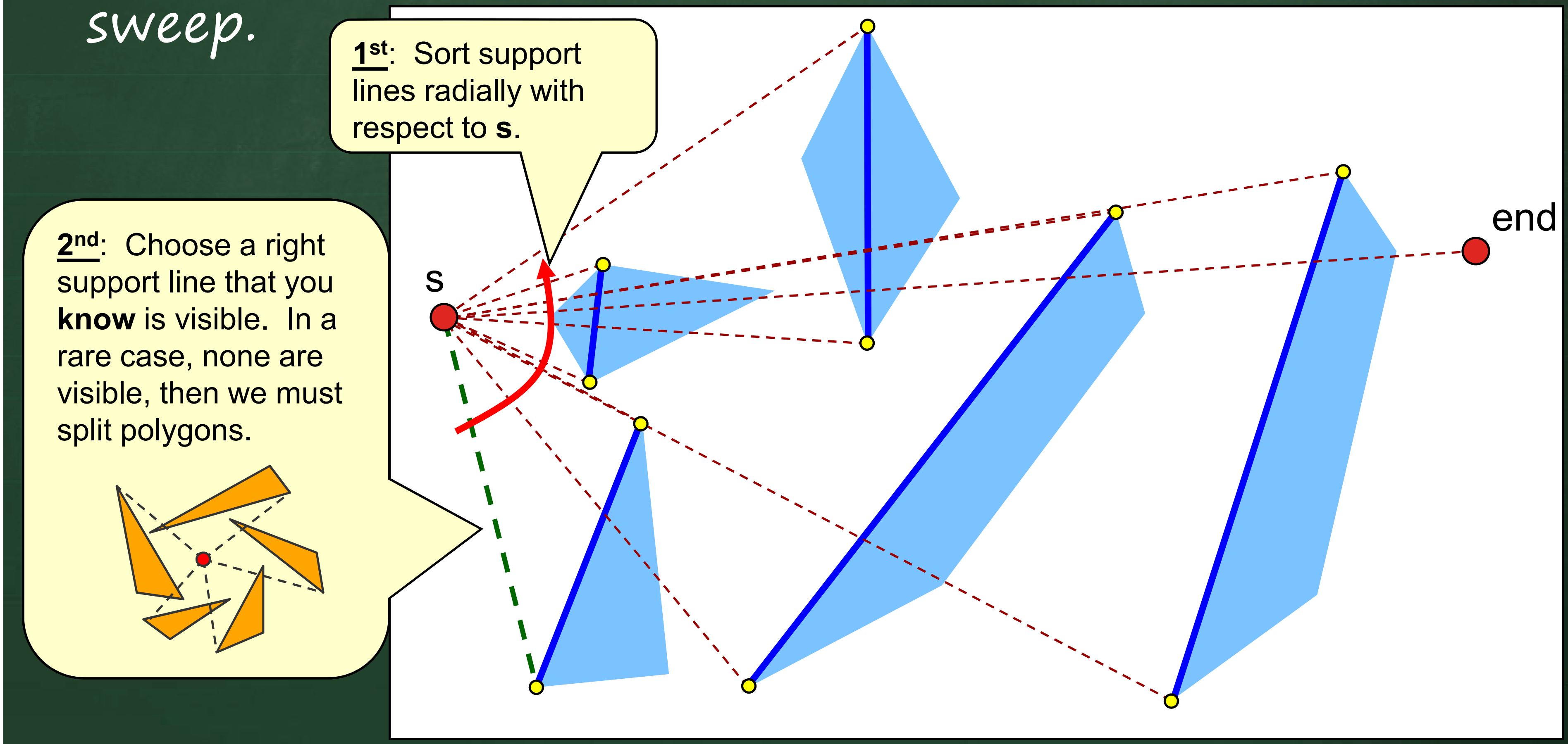
  $\rightarrow x = (b_b - b_a) / (m_a - m_b)$

- If ($m_a = m_b$) the lines are parallel and there is no intersection
- Otherwise solve for x, plug back in to get y.
- Final test is to ensure that intersection (x, y) lies on line segment … just make sure that each of these is true:
  - $\max(x_1, x_2) \geq x \geq \min(x_1, x_2)$
  - $\max(x_3, x_4) \geq x \geq \min(x_3, x_4)$



More Efficient Approach – Radial Sweep

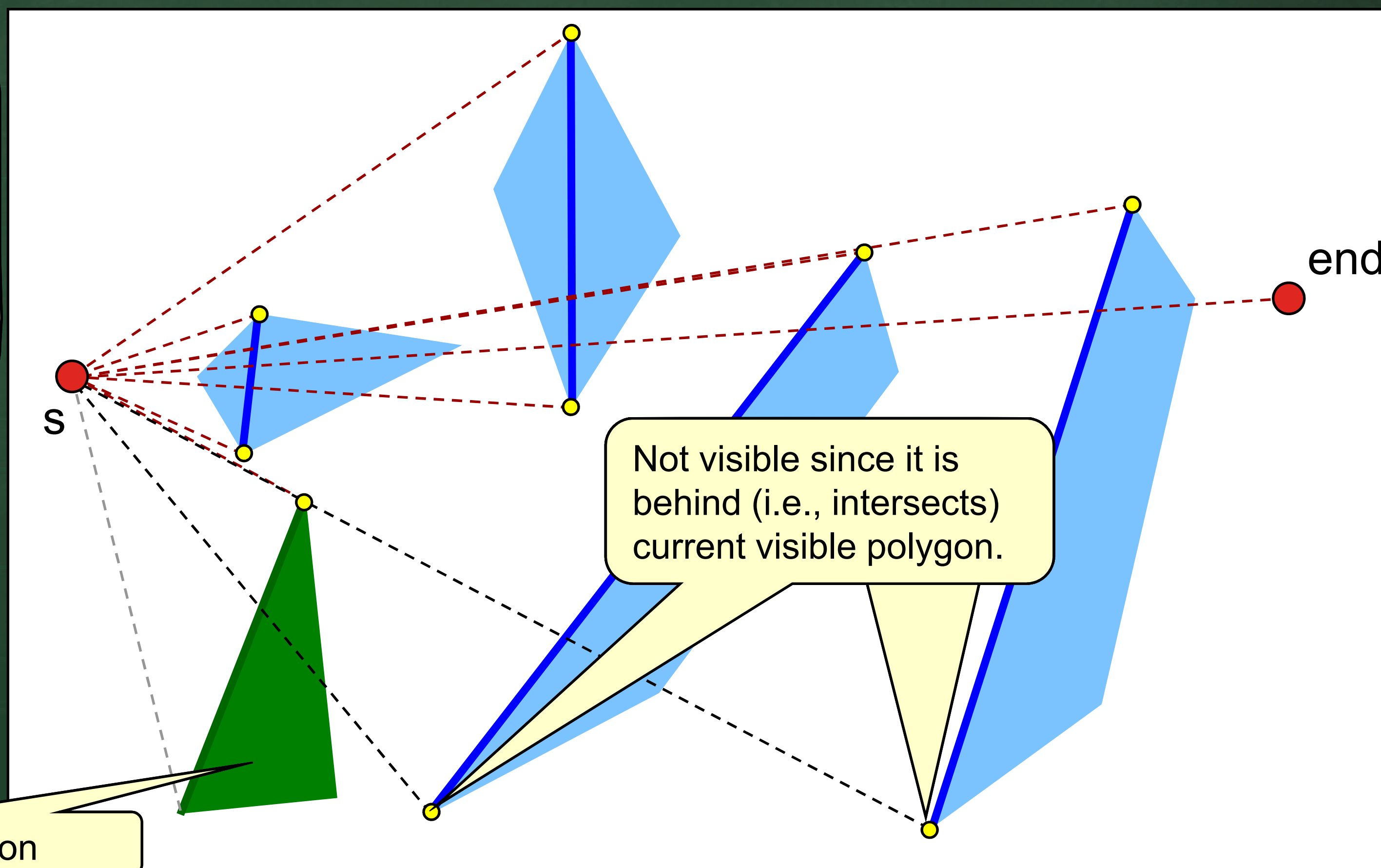
- More efficient approach can compute and remove all intersections in O(n log n) time by using a radial sweep.



More Efficient Approach – Radial Sweep

- 3rd: Do a radial sweep, keeping track of the closest (visible) polygon.

When reaching a right support vertex, check if it is in front or behind the current polygon.

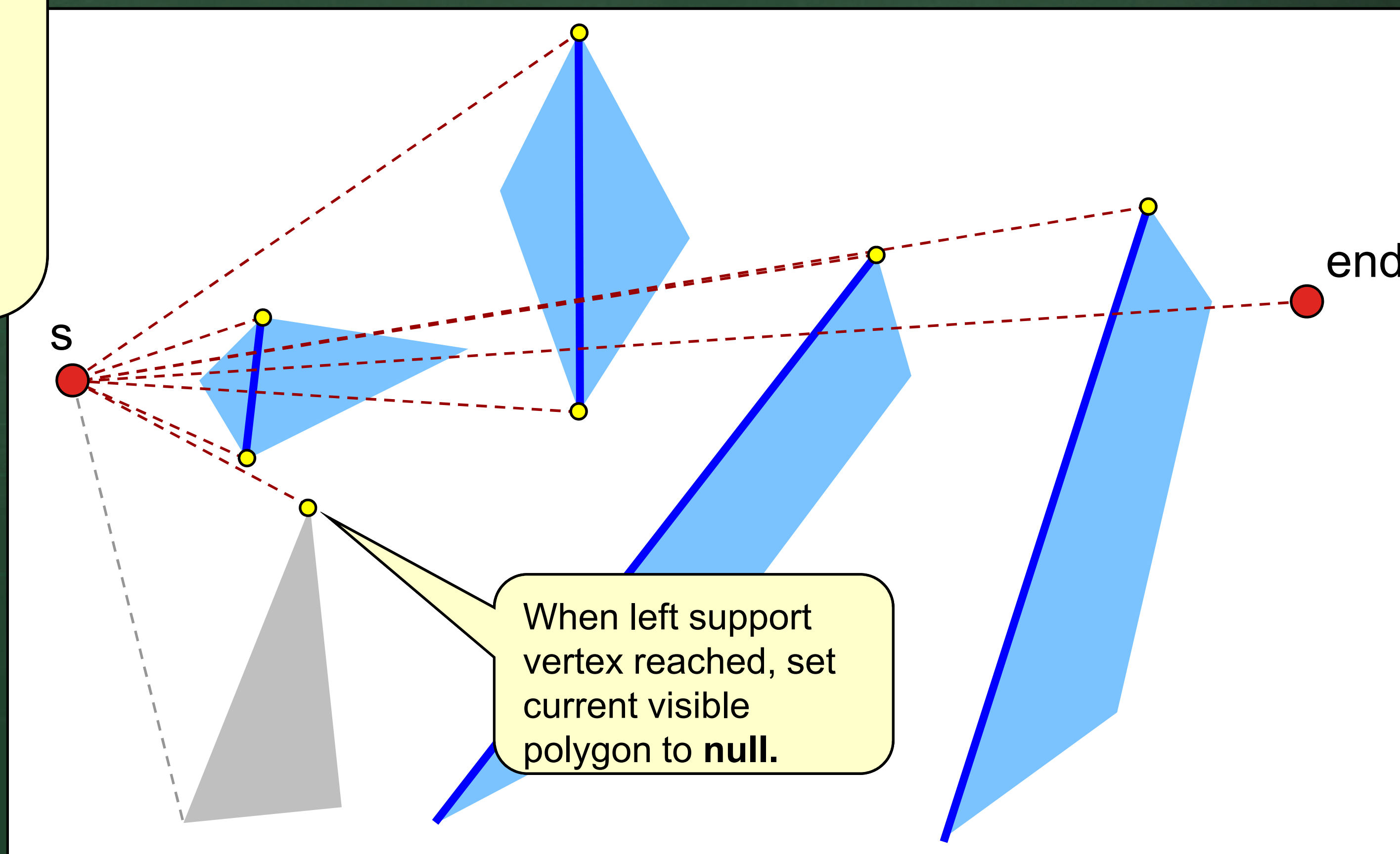


Current visible polygon

More Efficient Approach – Radial Sweep

- When left support vertex encountered:

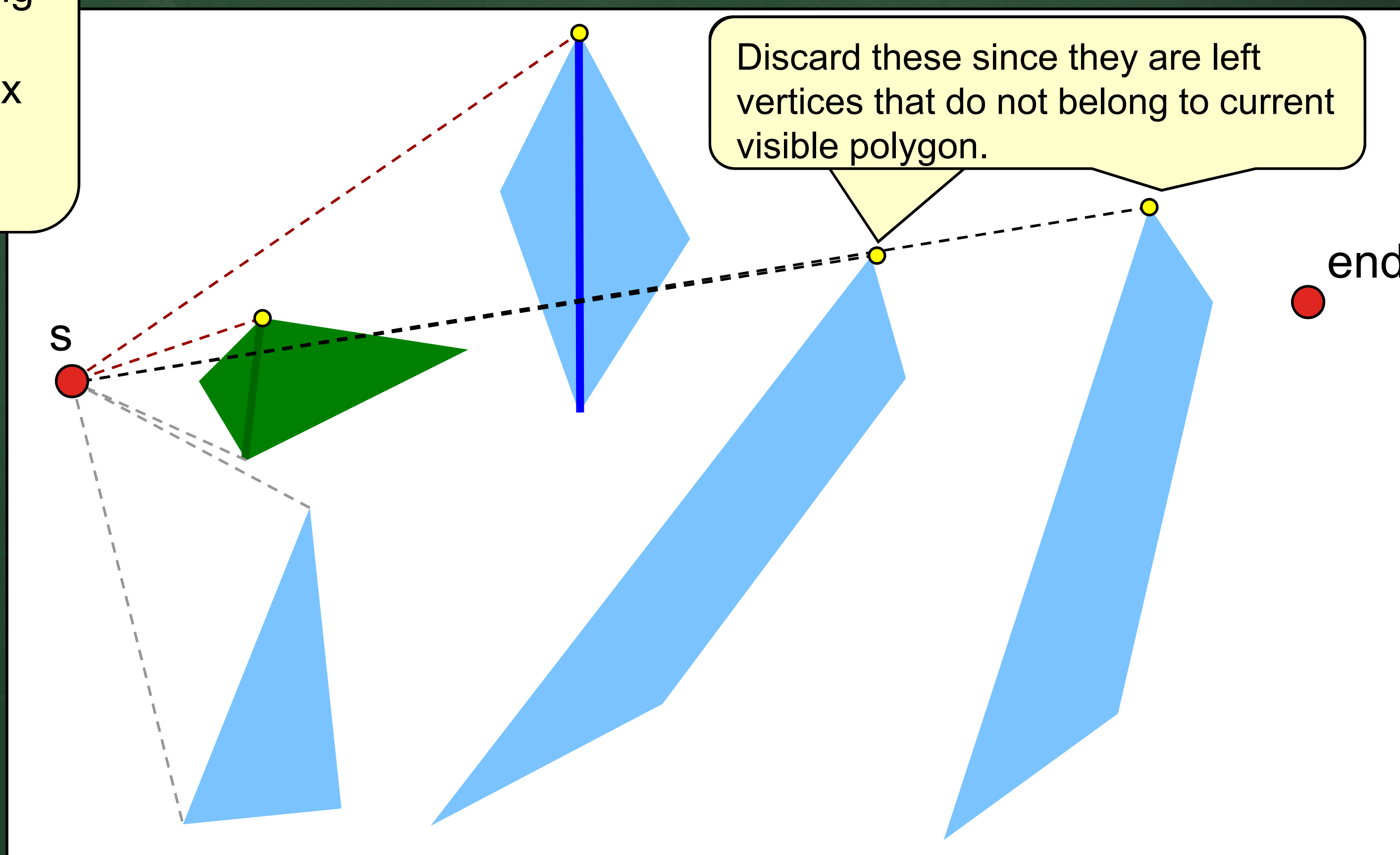
If it belongs to current visible polygon, mark it as visible and then set current polygon to **null**.



More Efficient Approach – Radial Sweep

- When left support vertex encountered:

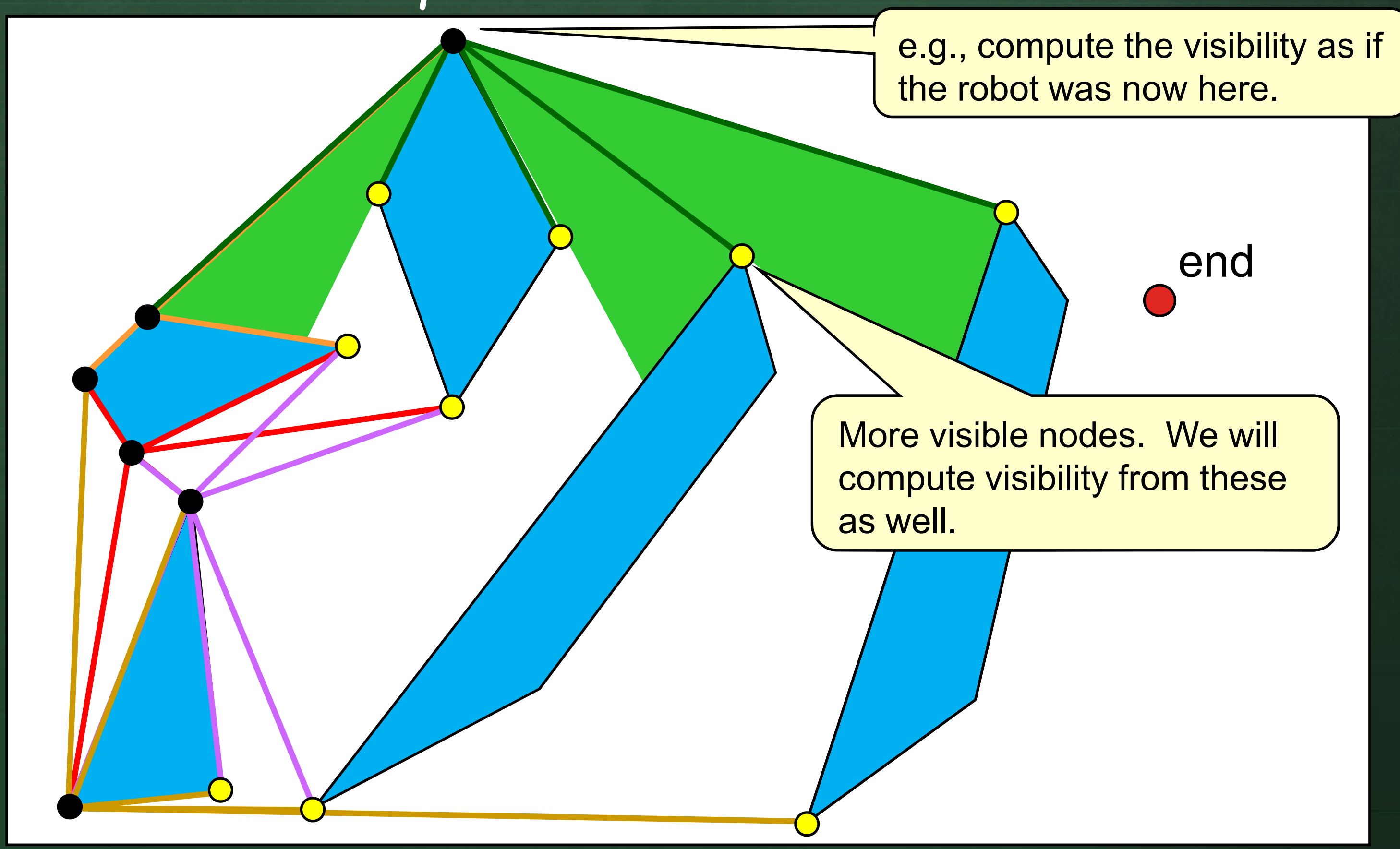
If it does not belong to current visible polygon, the vertex is not visible, so discard it.



Shortest Path Algorithm (Continued)

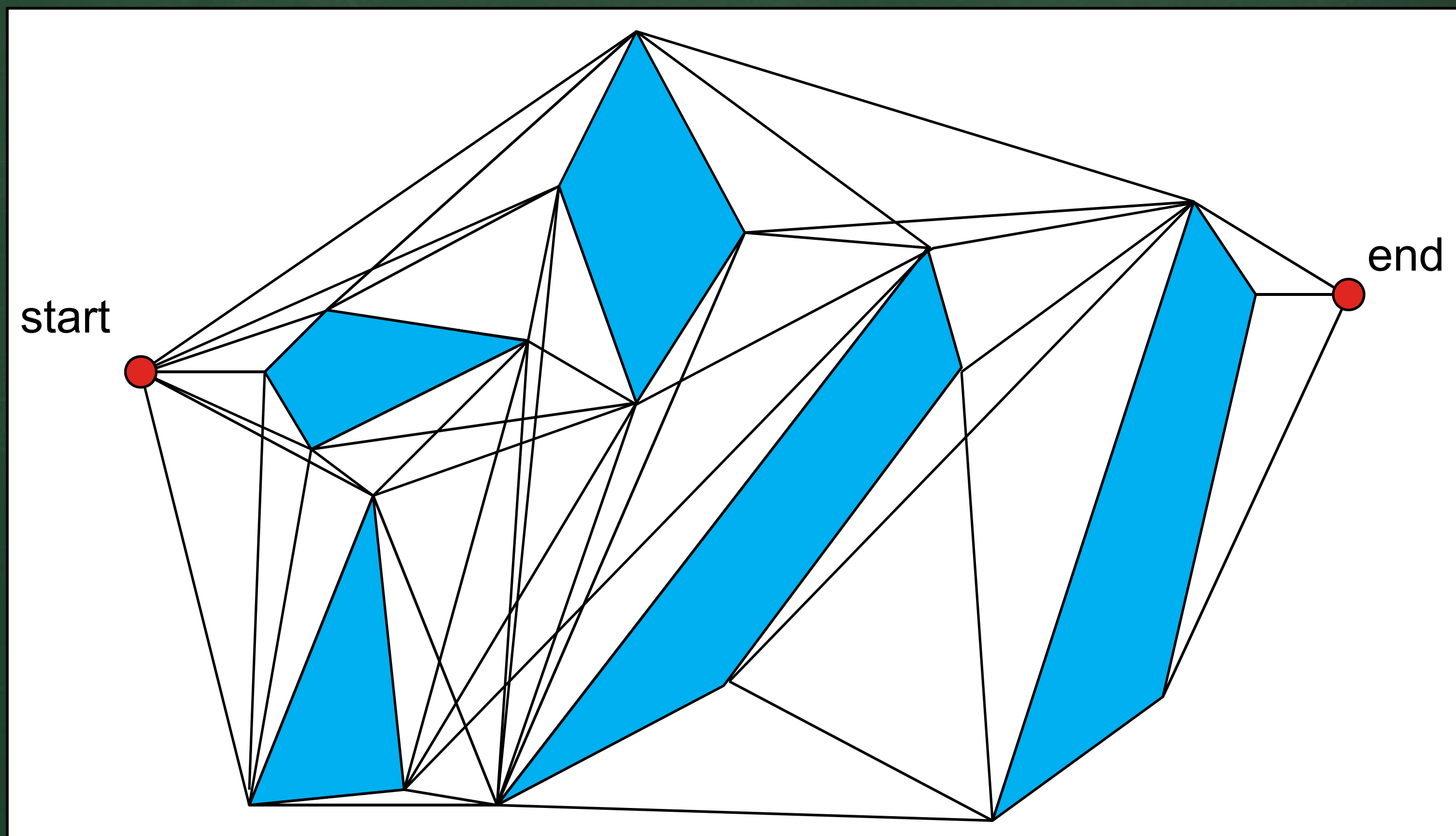
- Repeat this process iteratively by letting each visible support vertex become point s

(i.e., assume robot moved there):



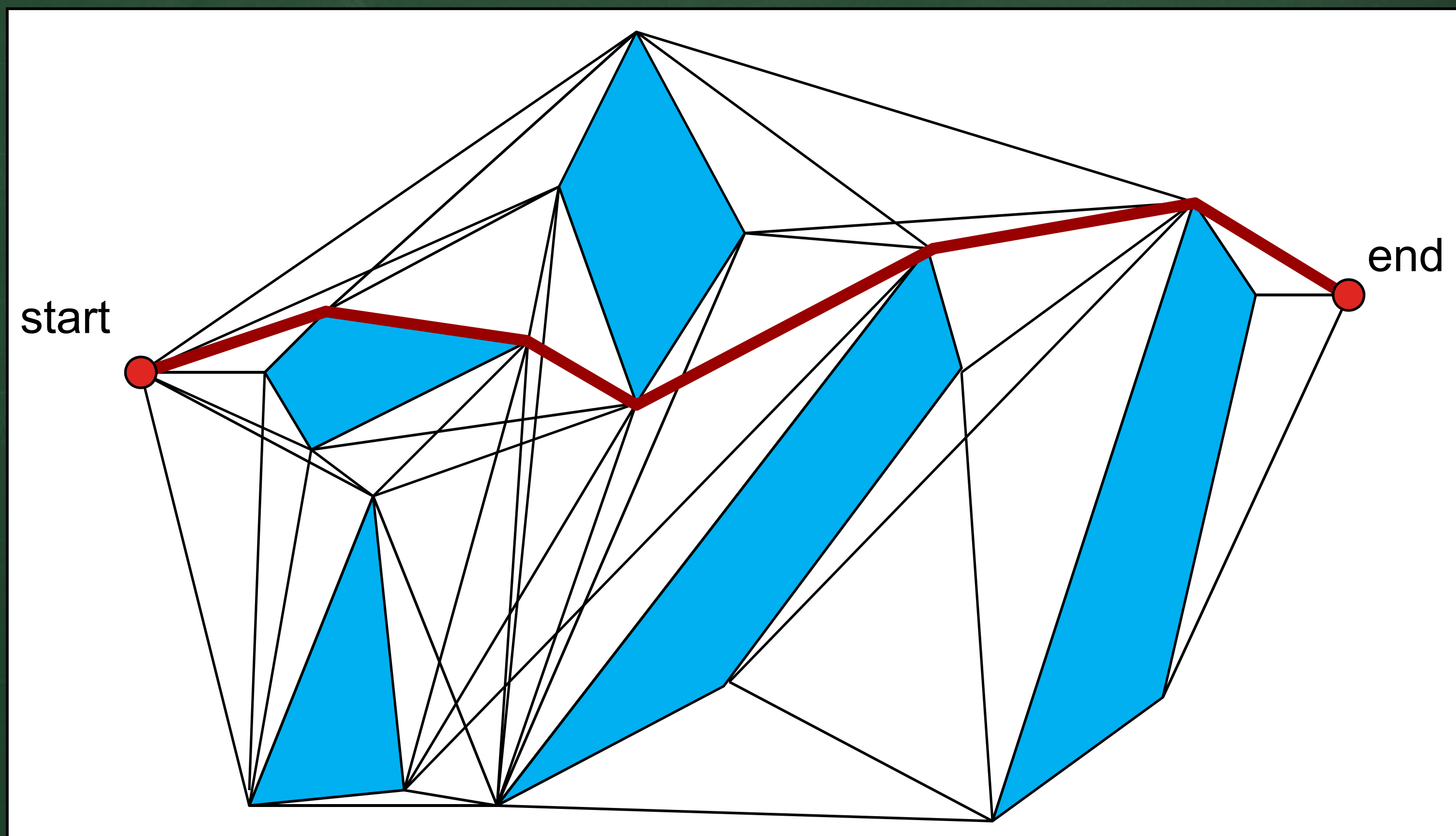
Shortest Path Algorithm (Continued)

- By appending all these visible segments together, a visibility graph is obtained:



Shortest Path Algorithm (Continued)

- We can then search this visibility graph for the shortest path from the start to the goal:



Shortest Path in Graph

- We can use Dijkstra's shortest path algorithm to compute the shortest path in this graph.
  - Takes O(V log V + E) time for a V-vertex / E-edge graph

```
function Dijkstra(G, s, t)
    for (each vertex v in V[G]) do
        d[v] = infinity
        previous[v] = undefined
    d[s] = 0
    Q = queue of all vertices
    while (Q is not empty) do
        u = Extract-Min(Q)
        if (u == t) then return;
            for each edge (u,v) outgoing from u do
                if (d[v] > d[u] + dist(u,v)) then
                    d[v] = d[u] + dist(u,v)
                    previous[v] = u
                    Q = Update(Q)
```

Initialize all distances to vertices (i.e., d[v]) to ∞, except start which has distance 0

Remembers how we got to this node.

Priority queue sorted by distances from s.

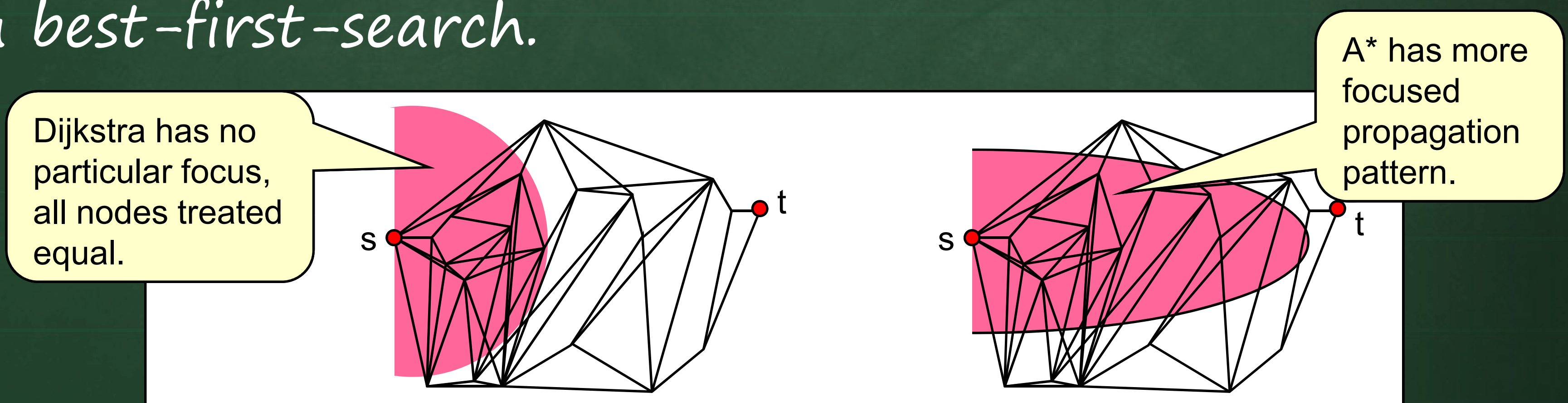
Get the next closet unprocessed vertex. If it is the destination, we are done.

Relax all edges from u to v, by updating (reducing) the cost d[v] at each v if can be reached quicker from u.

Re-sort the queue since priorities may have changed.

Shortest Path in Graph

- Alternatively, we can use the A* algorithm.
  - employs "heuristic estimate" that ranks each node by an *estimate* of the best route that goes through that node.
- Dijkstra employs breadth-first-search, while A* does a best-first-search.



- Can be just as bad as Dijkstra in worst case but often much quicker in practice.

Shortest Path in Graph

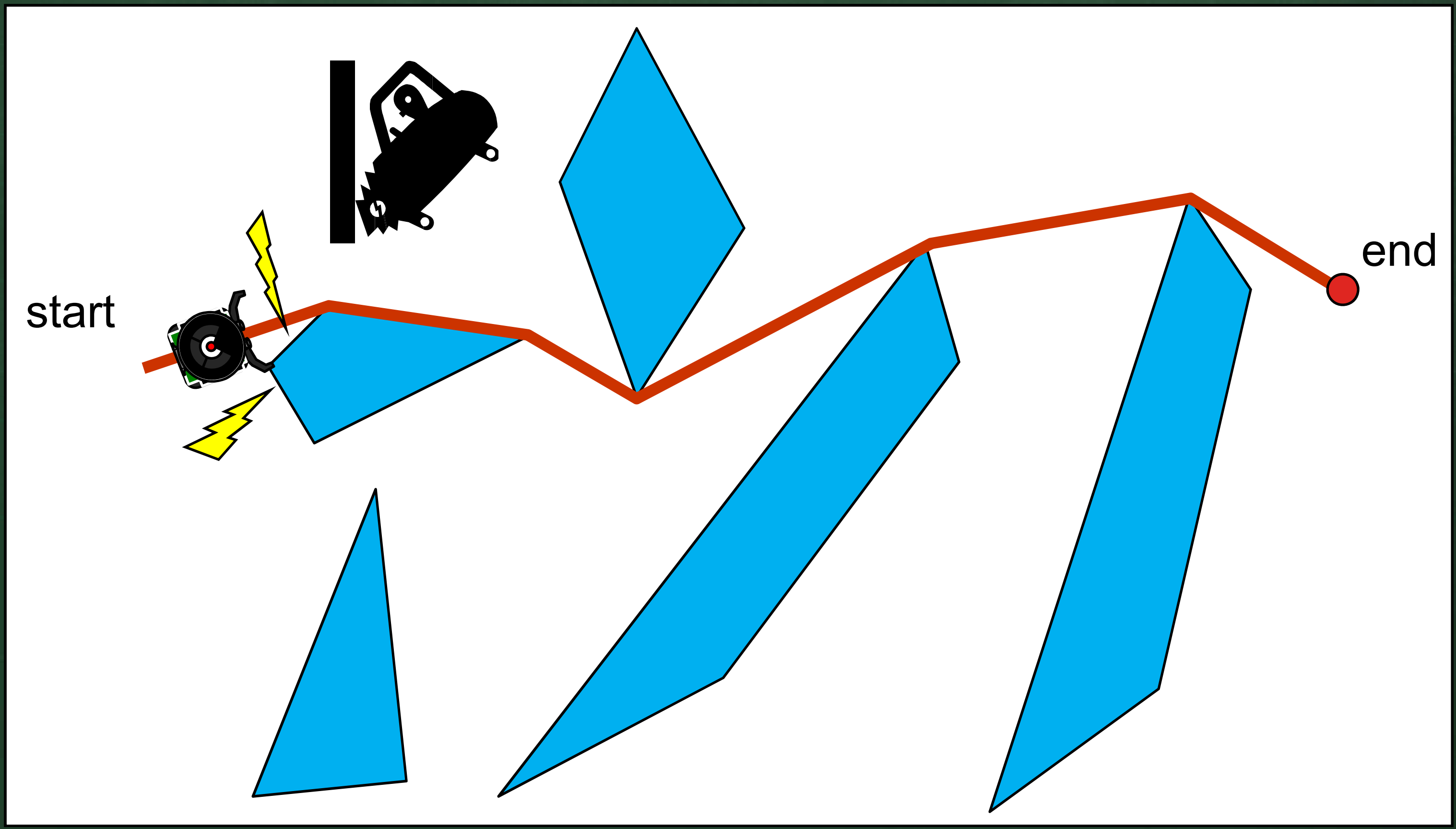
- A* algorithm is the same as Dijkstra's except that:
  - In Dijkstra's alg., nodes v_i in queue are sorted by $d[v_i]$
  - In A*, they are sorted by $d[v_i] + E[v_i,t]$ where $E[v_i,t]$ is an ***underestimate*** of the distance from v_i to t.
  - Only the Extract-Min(Q) function of line 8 is affected.
- E is often simply the straight line distance from v_i's coordinate to t's coordinate.
  - This is always an underestimate since the real cost from v_i to t can never be greater than the straight line cost.

Shortest Path Traversal

- From this produced path, we have a set of points and angles.
- We can then apply inverse kinematics for our robot to move it along this path.
- But wait a minute! Our robot is not a point, it's a rectangle. We cannot simply hug along the obstacle boundaries!

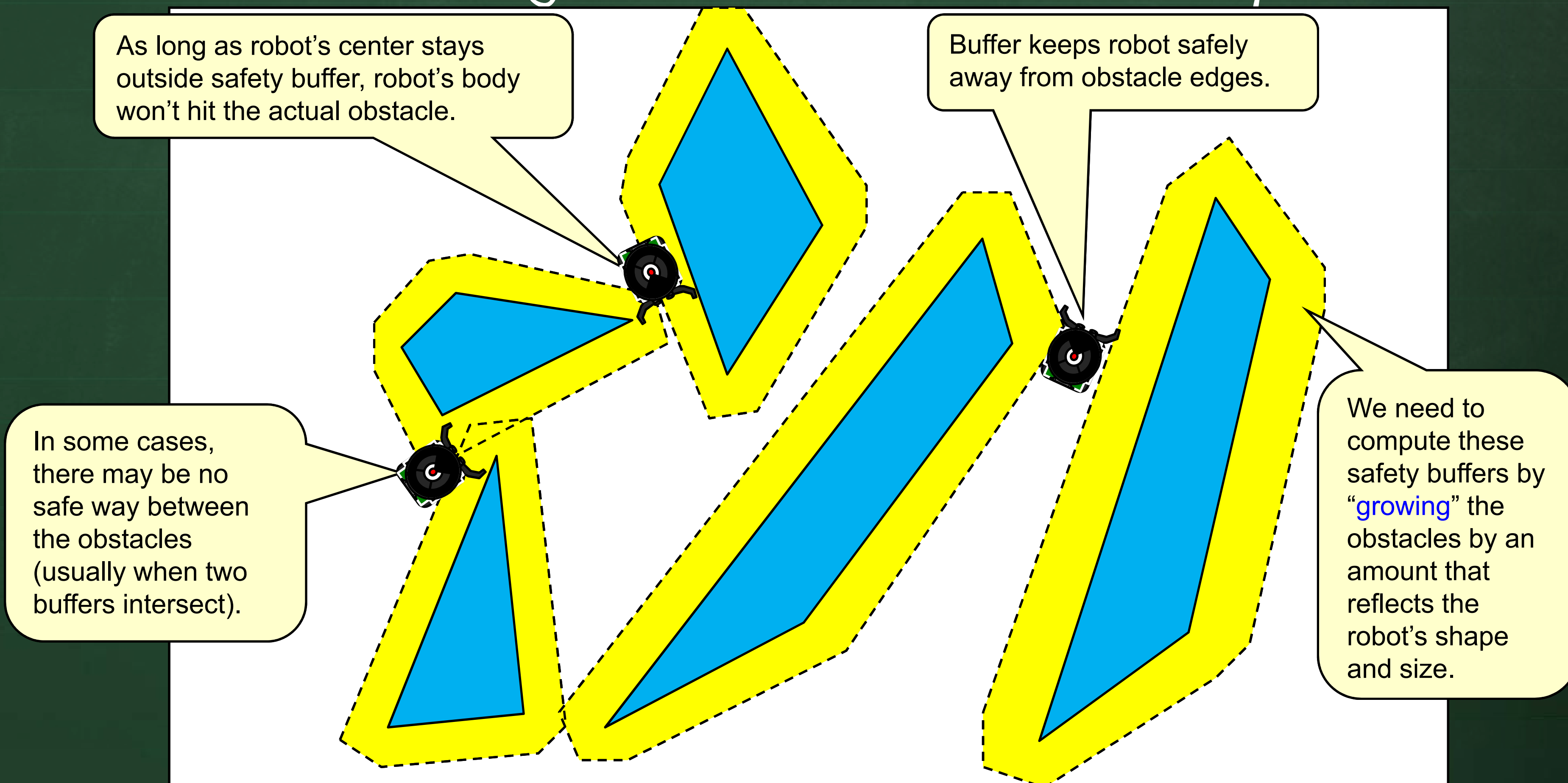
Real Robot Shortest Path

- Our real-robot will collide with obstacles if it travels along a path computed as we described:



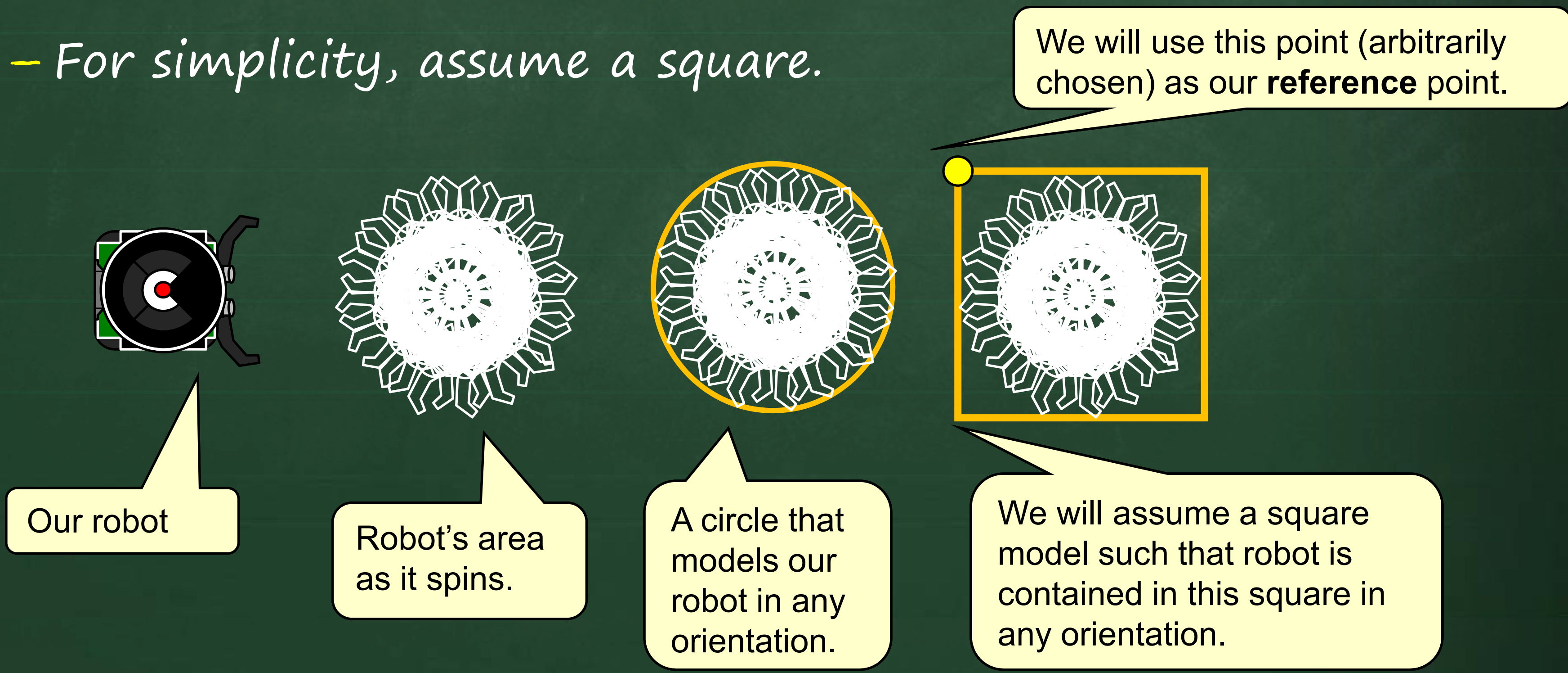
Real Robot Shortest Path

- We need a kind of "safety buffer" around each obstacle according to the robot's size & shape:



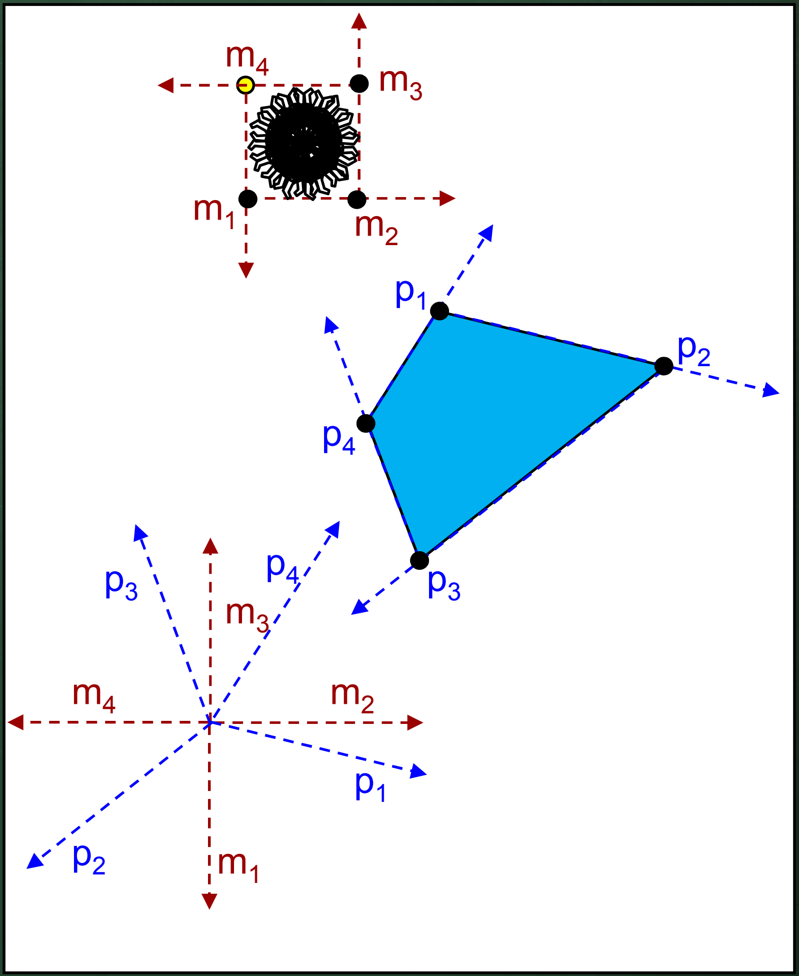
Real Robot Shortest Path

- If robot is symmetrical in all directions, we can still work with our same algorithm.
  - Only circles are symmetrical in all directions.
  - For simplicity, assume a square.



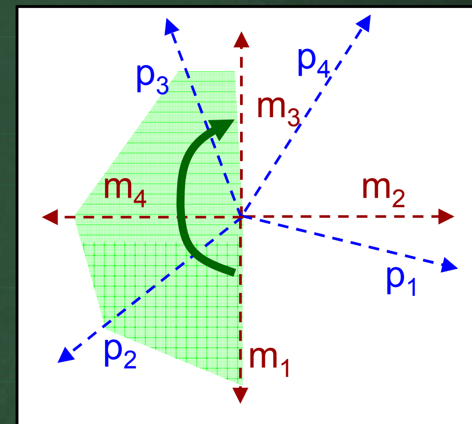
Real Robot Shortest Path

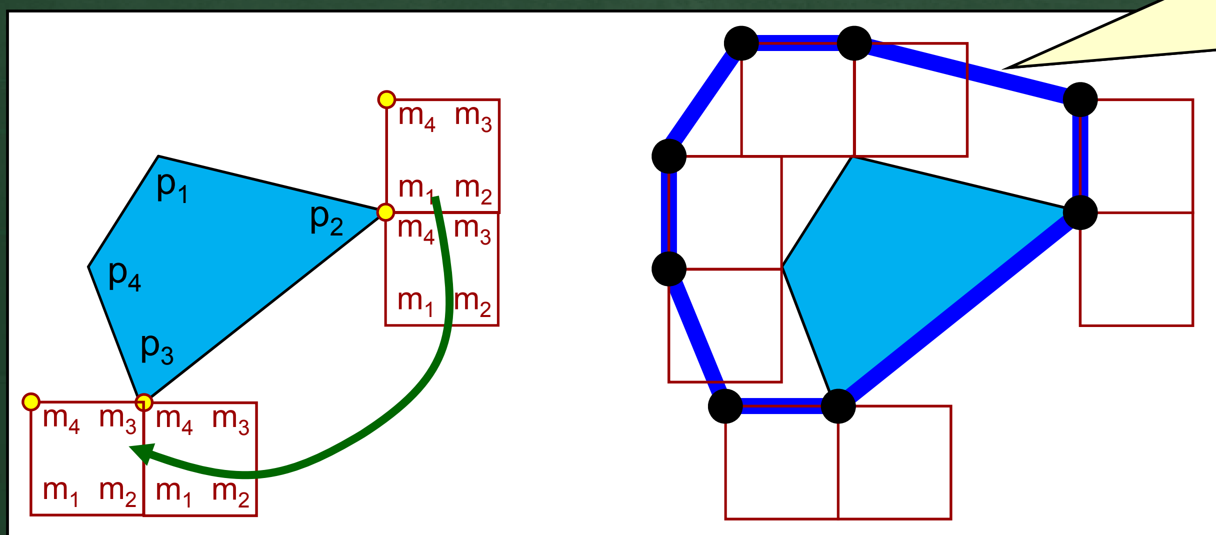
- Can apply a "growing" procedure to each obstacle:
  - Determine edge vectors along model in CCW order:
  - Determine edge vectors along polygon in CW order:
  - Sort combined edge vectors by angle



Real Robot Shortest Path

- Traverse the vectors radialy clockwise starting at m_1.
  - When sweeping between model vector m_i and polygon vector p_i, translate the model such that $m_i = p_i$

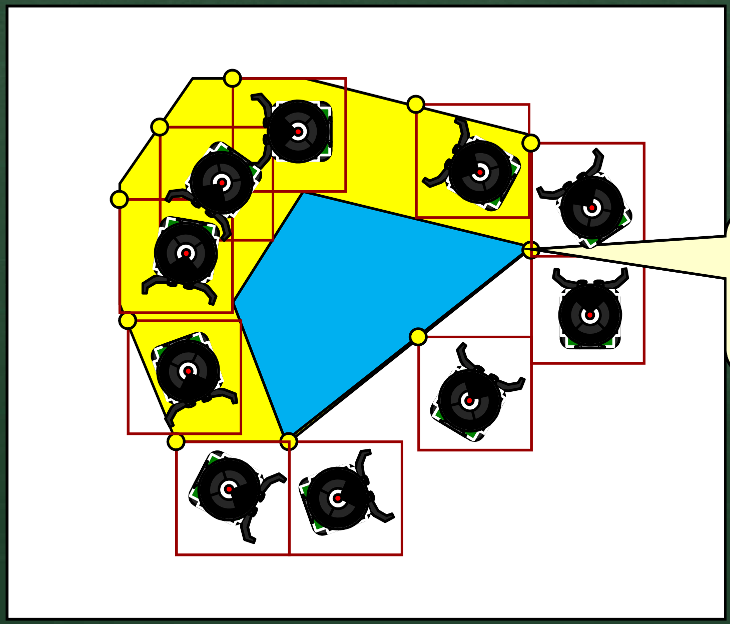




Connect reference points of all model translations to form the ***grown obstacle*** which will have at most **n+4** vertices.

Real Robot Shortest Path

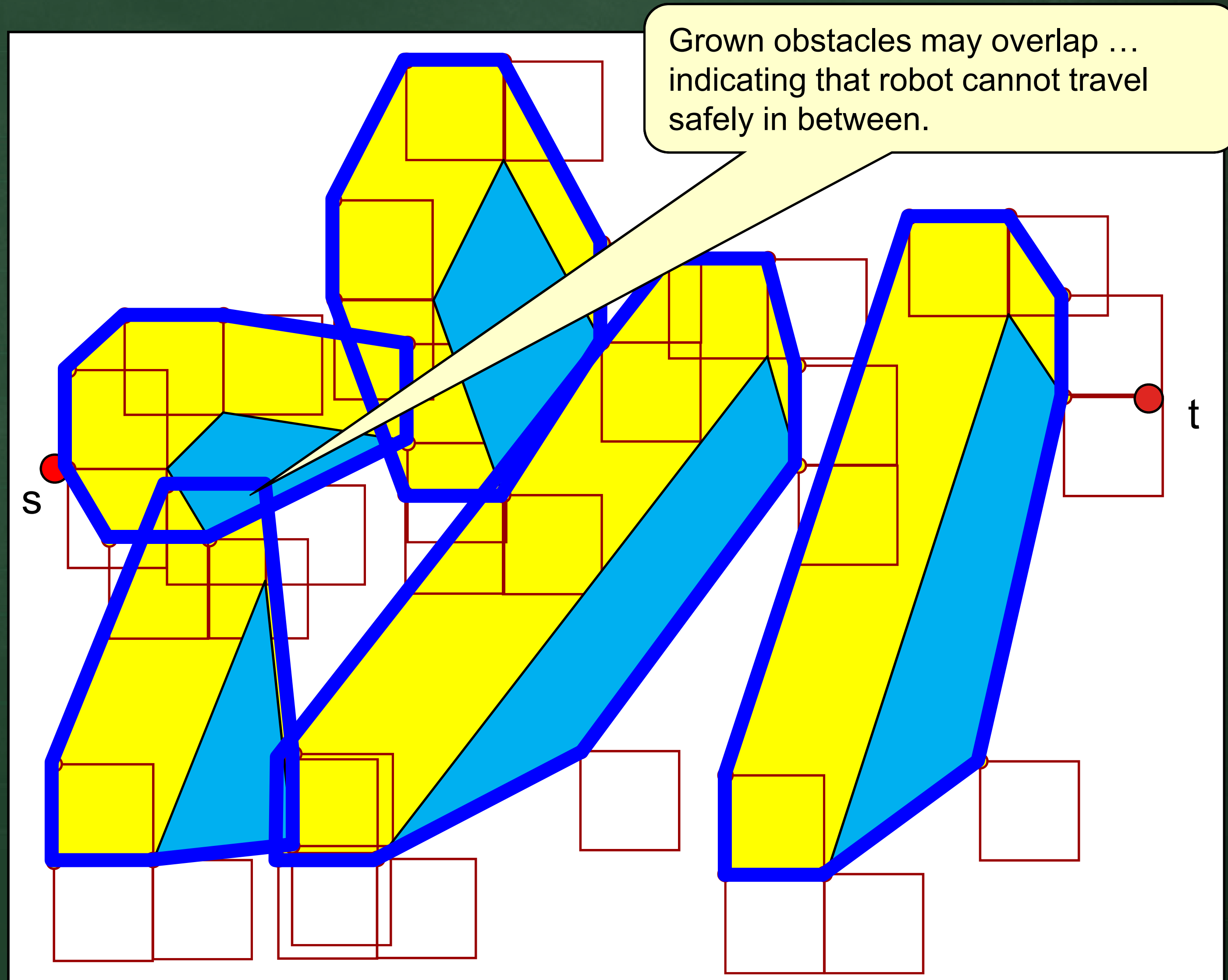
- It is easy to see that as long the reference point of our model lies on our outside the grown obstacle, then the robot will not collide with the real obstacle.



Reference point lies on or outside grown obstacle.

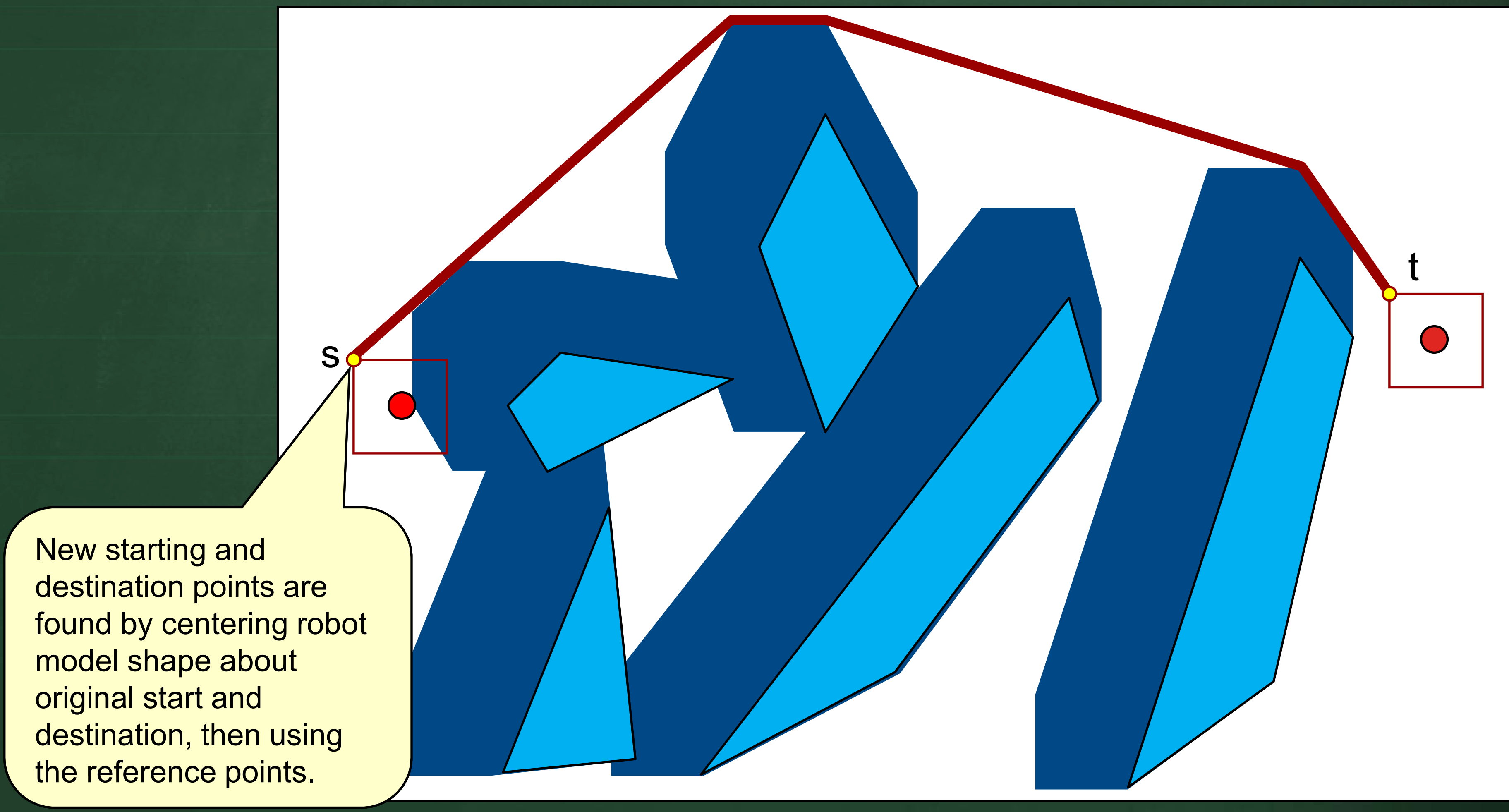
Real Robot Shortest Path

- Apply this to all obstacles to obtain the *grown obstacle space.*



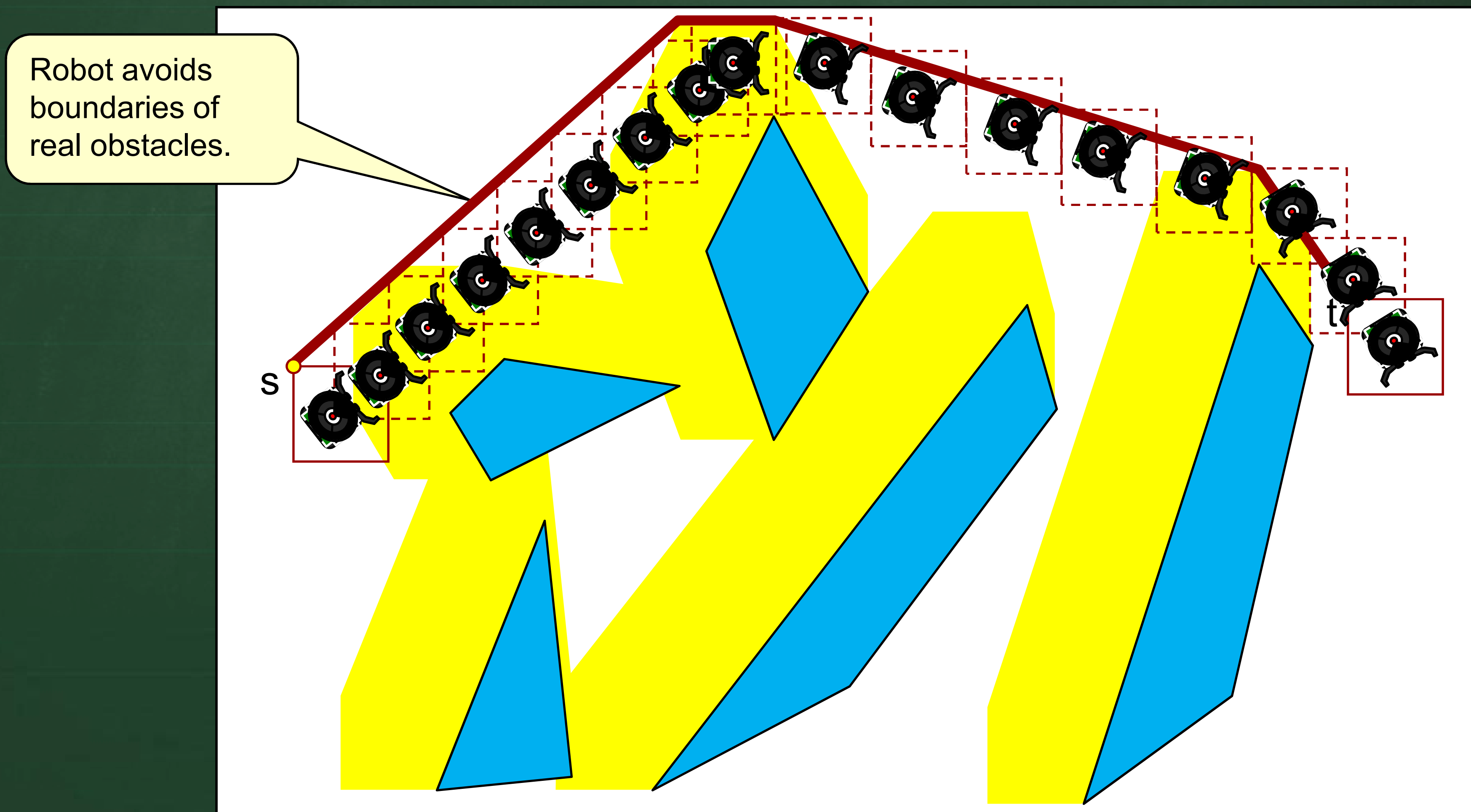
Real Robot Shortest Path

- Now apply previous point-robot algorithm.



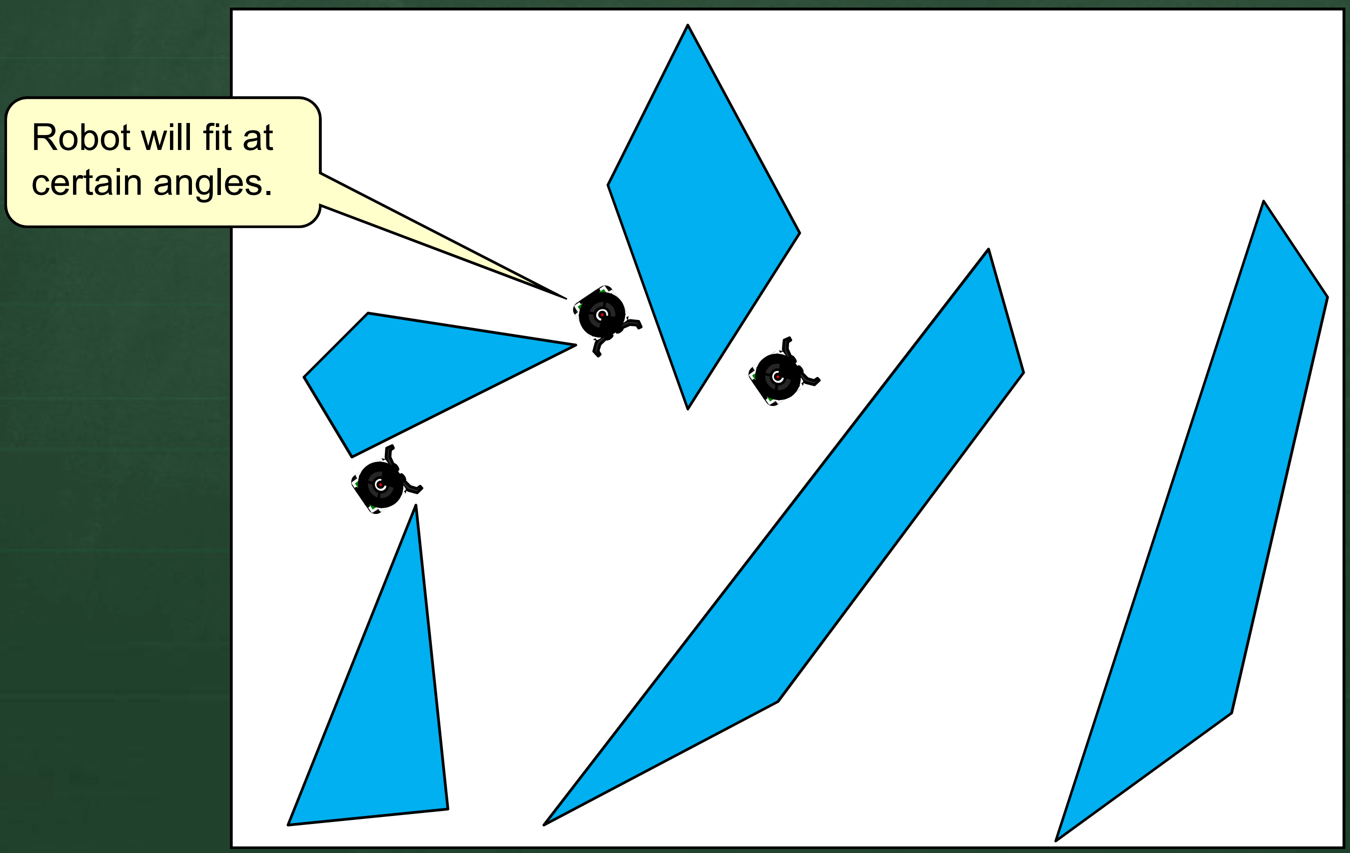
Real Robot Shortest Path

- Robot now moves safely along path:



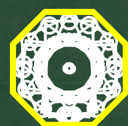
Real Robot Shortest Path

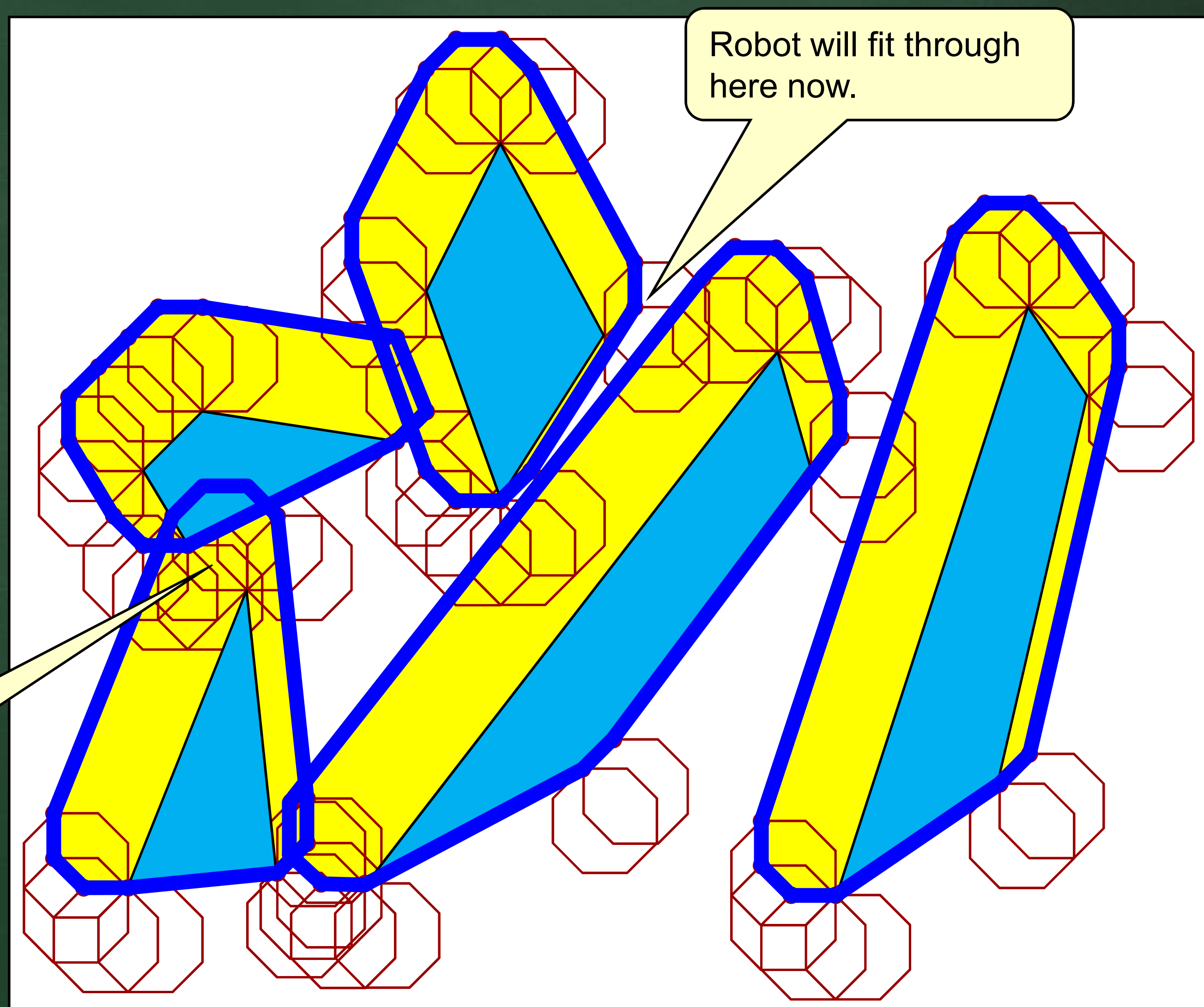
- But our robot should be able to fit in between those obstacle. Why doesn't our solution allow this ?



Real Robot Shortest Path

- Use a more accurate model to produce a more accurate path.

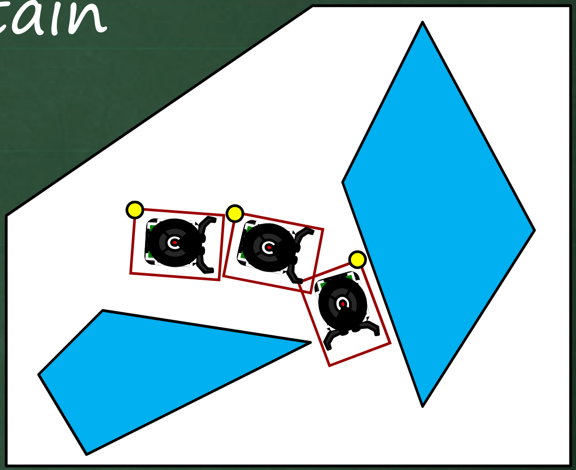




Robot still cannot fit through some places.

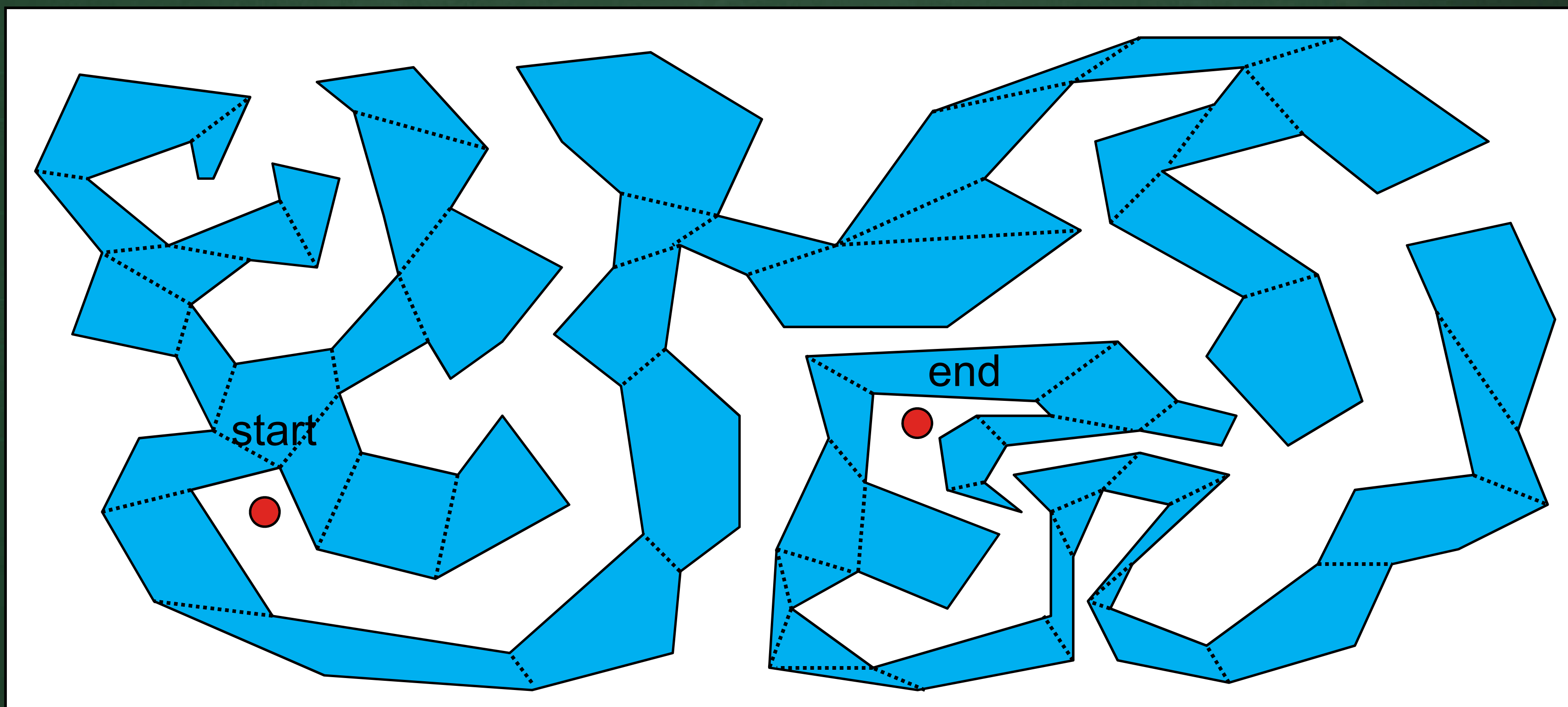
Real Robot Shortest Path

- We could use a more complicated approach that allows the robot to pass through certain areas only in specific directions.
  - can shrink the model .
  - must allow model to rotate.
  - Too complicated for us in this course, but can be done.
  - Realistically, robot sensors are not reliable enough nor accurate enough to ensure safe travel within areas that require a small margin of error.



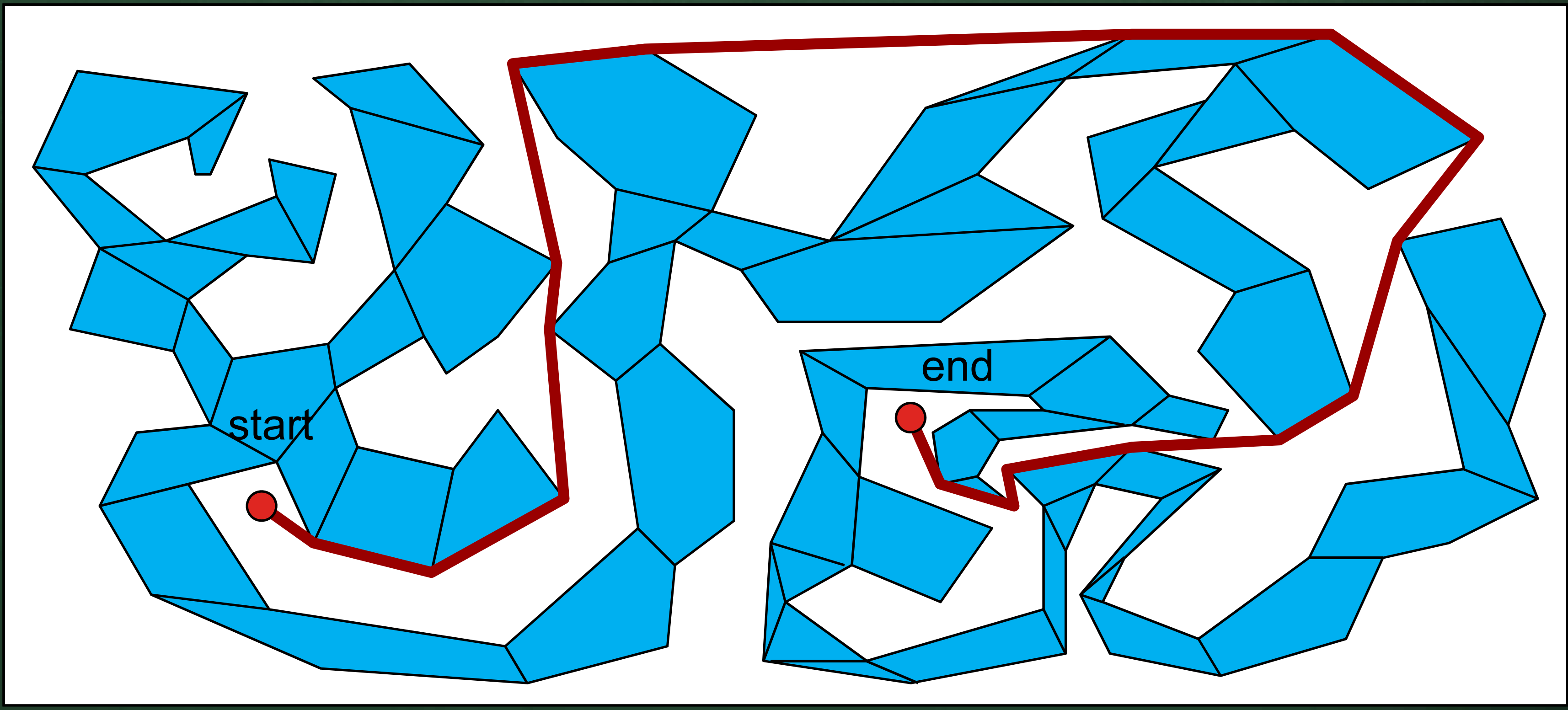
Non-Convex Obstacles

- What about non-convex obstacles ?
  - Can divide them into convex polygons and then apply the same algorithms (although better solutions exist).



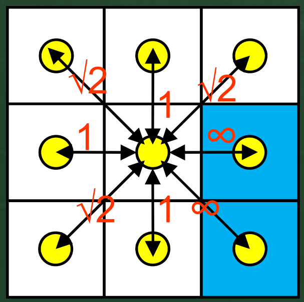
Non-Convex Obstacles

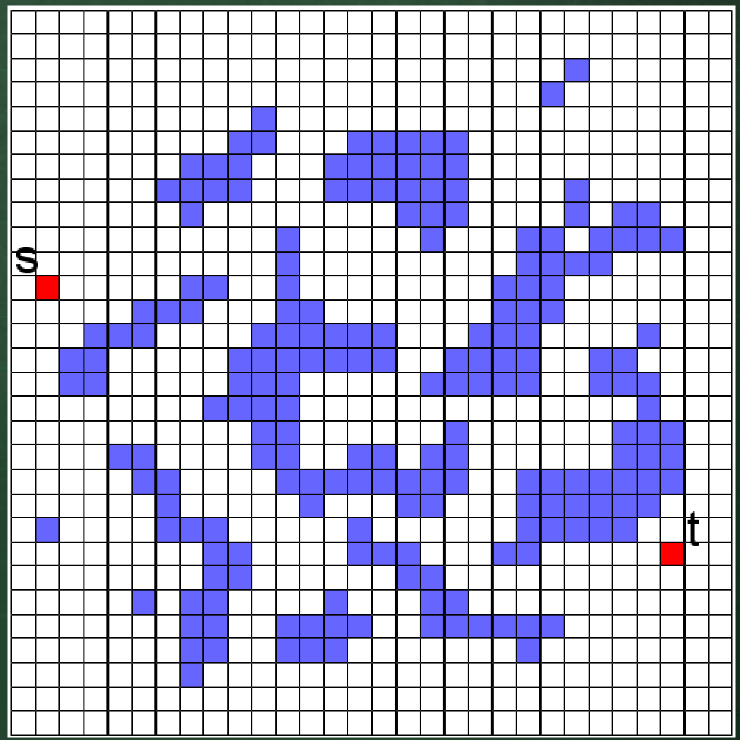
- What about non-convex obstacles ?
  - Can divide them into convex polygons and then apply the same algorithms (although better solutions exist).



Shortest Paths in Grids

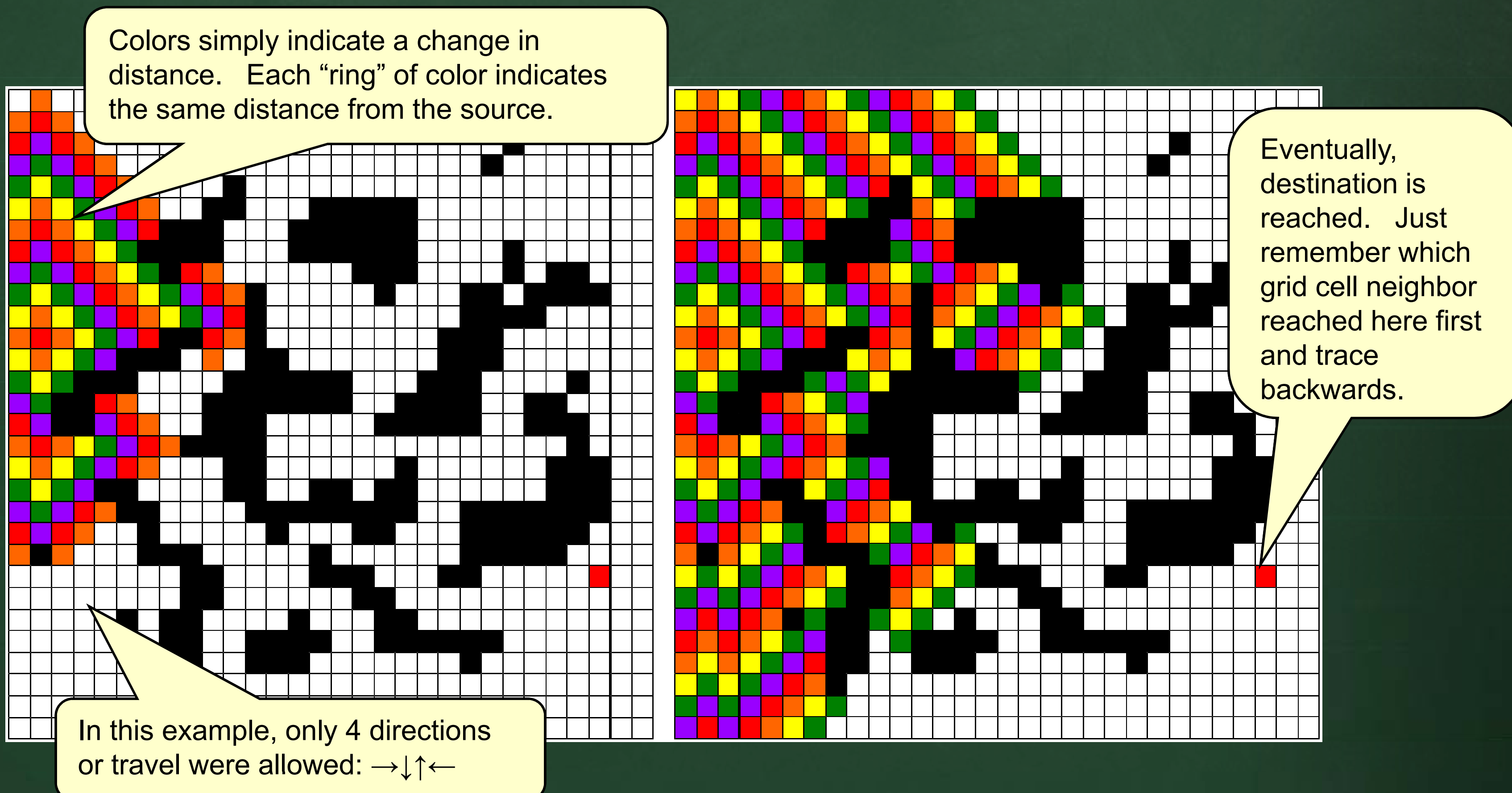
- How do we find the shortest path in a binary grid ?
  - Can apply Dijkstra's algorithm by creating an "implicit" graph from the grid.
  - Assign weights to nodes according to realistic distance





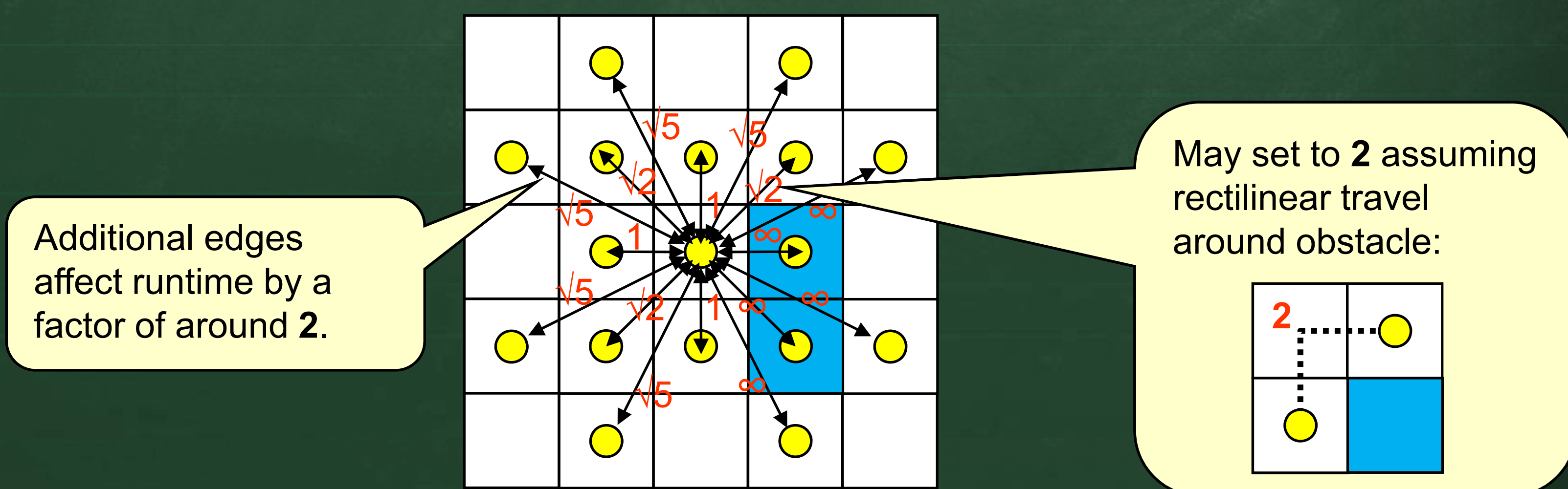
Shortest Paths in Grids

- As cells are processed in order of distance from source, a *wavefront* propagates through the grid:



Shortest Paths in Grids

- For an $M \times N$ grid, the graph has $O(MN)$ vertices and $O(MN)$ edges.
- Algorithm thus takes $O(MV \log MN)$ runtime.
- More accurate paths can be produced if we increase the number of edges:

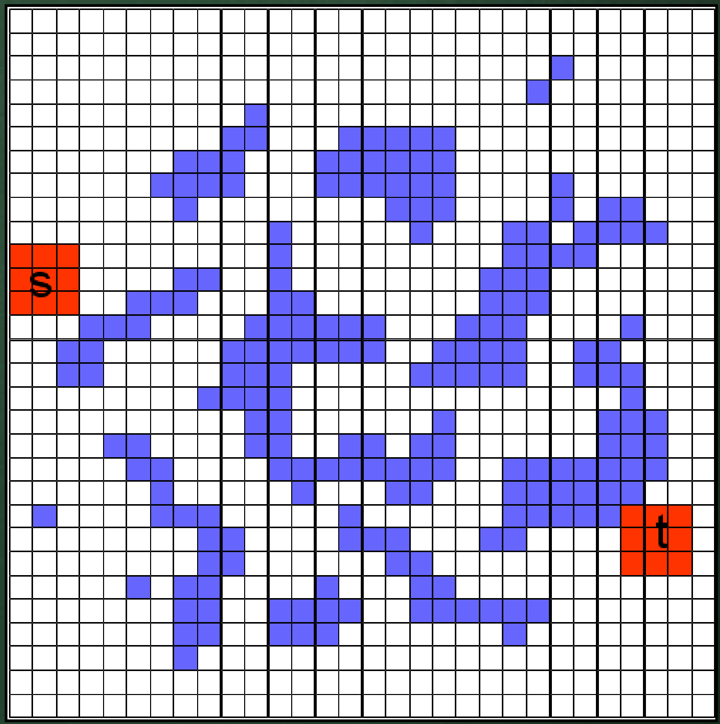


Shortest Paths in Grids

- It takes no effort to handle complicated obstacles since algorithm merely concentrates on moving from one grid unit to another.

- What about non-point robots ?

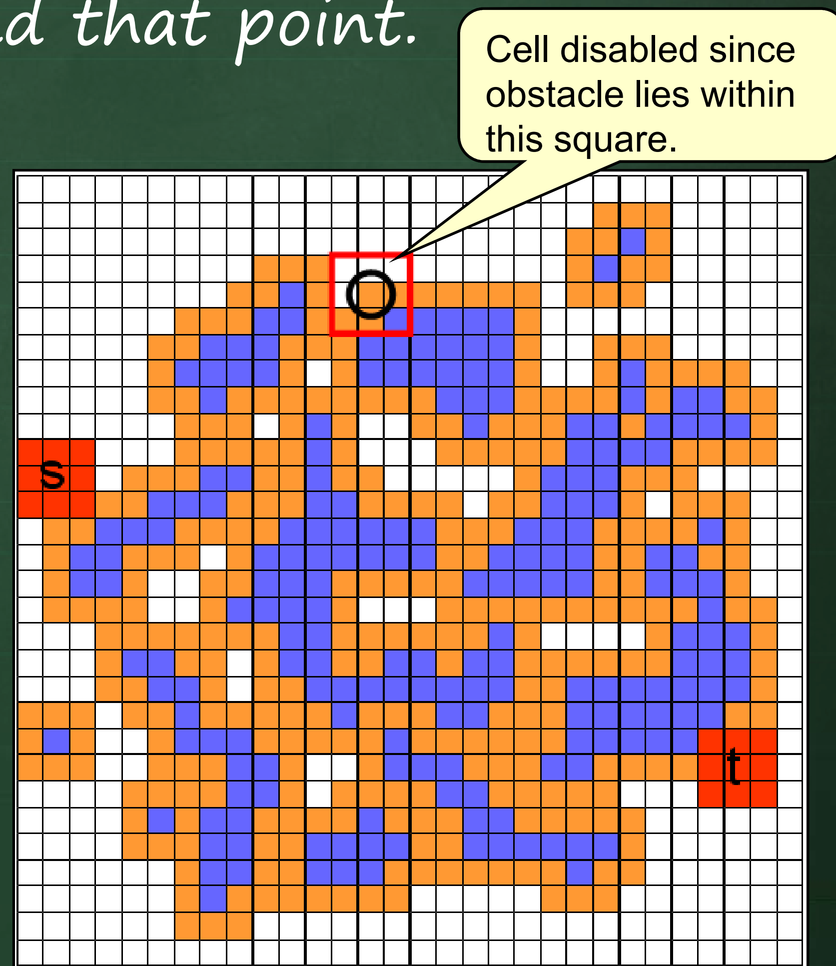
- Since robot shape is known ahead of time, we can adjust the weights of adjacent nodes in the grid accordingly.



Shortest Paths in Grids

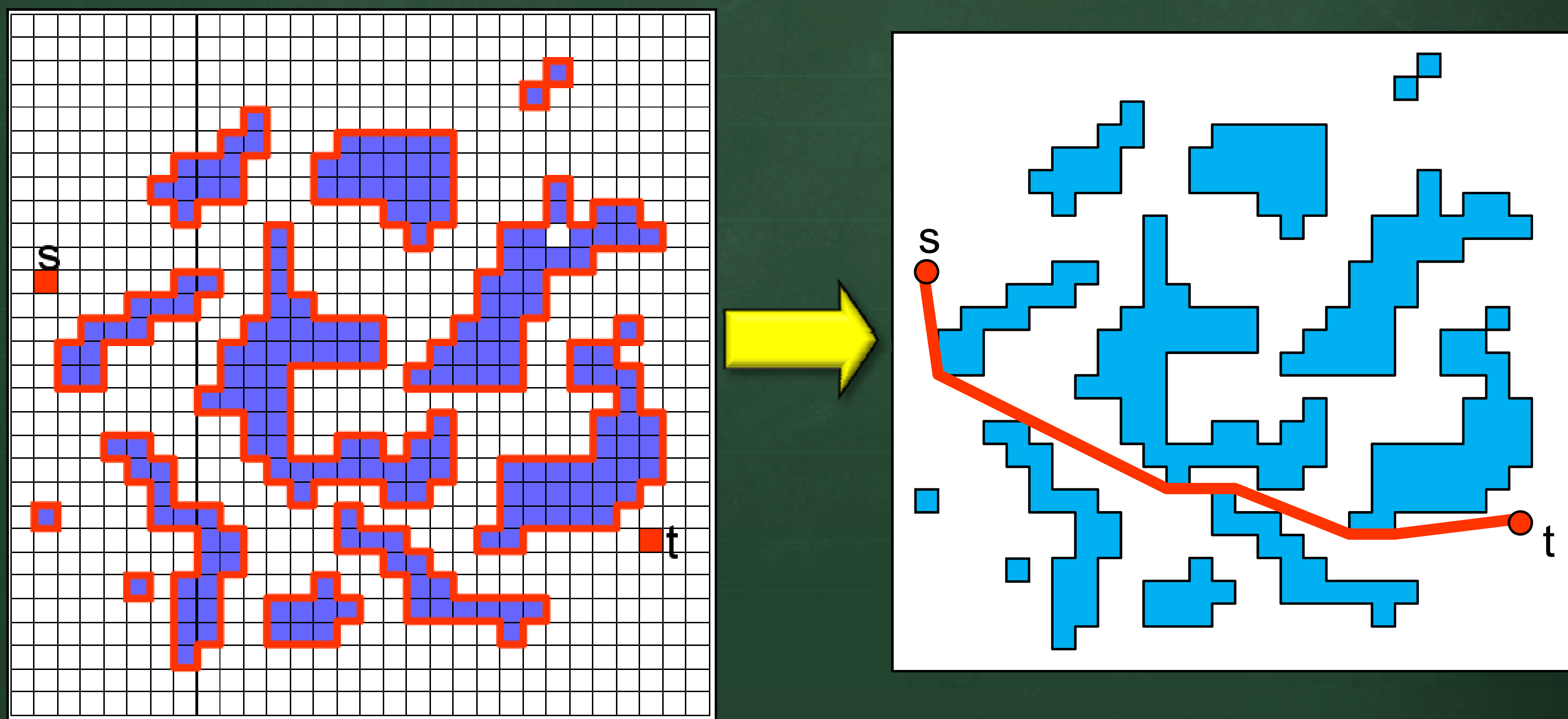
- For each grid location, center robot model (i.e., a collection of grid cells) around that point.

- If any obstacle locations intersect it, disable this grid location either by:
  - removing the node from the graph entirely
  - setting the weights of edges going in and out of it to ∞.



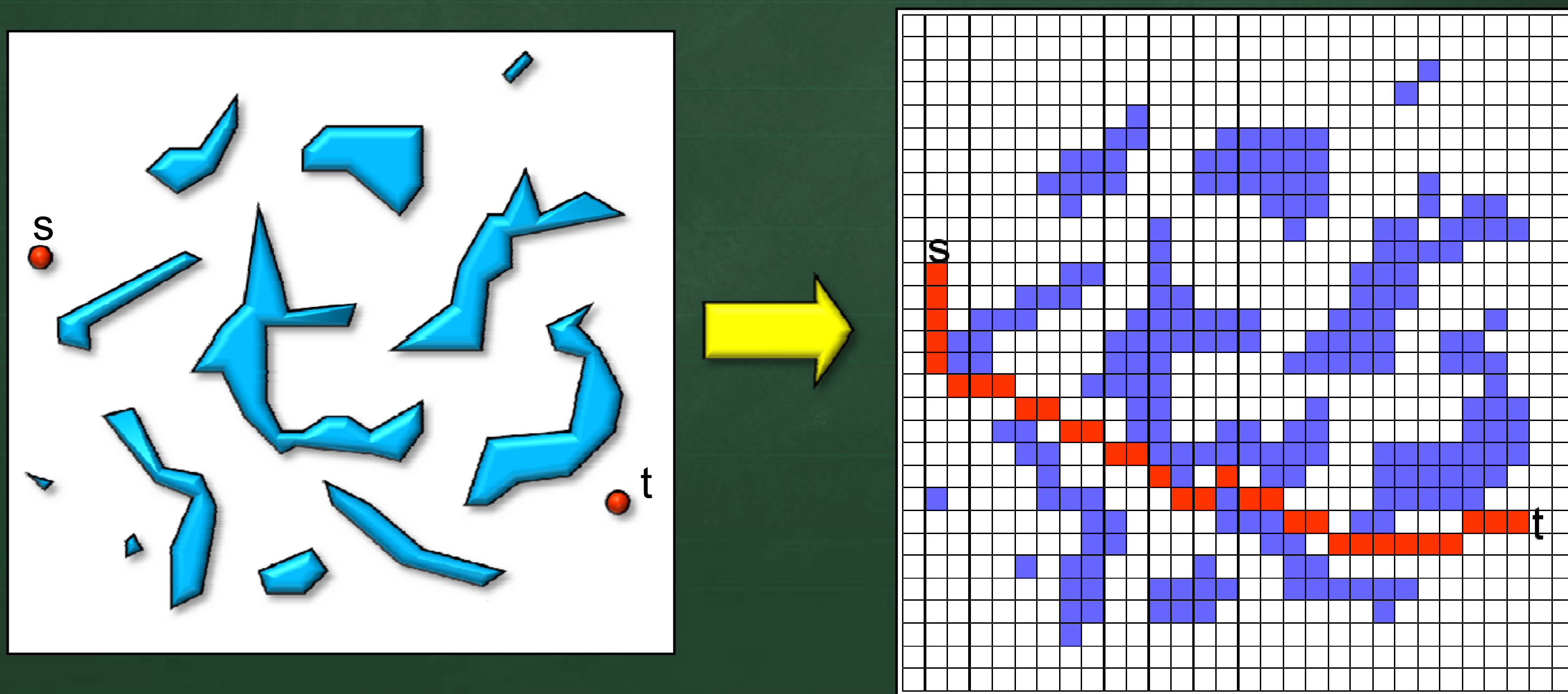
Shortest Paths in Grids

- Another solution to the grid shortest path problem is to convert the grid into vector obstacles, then apply the vector-based algorithm:



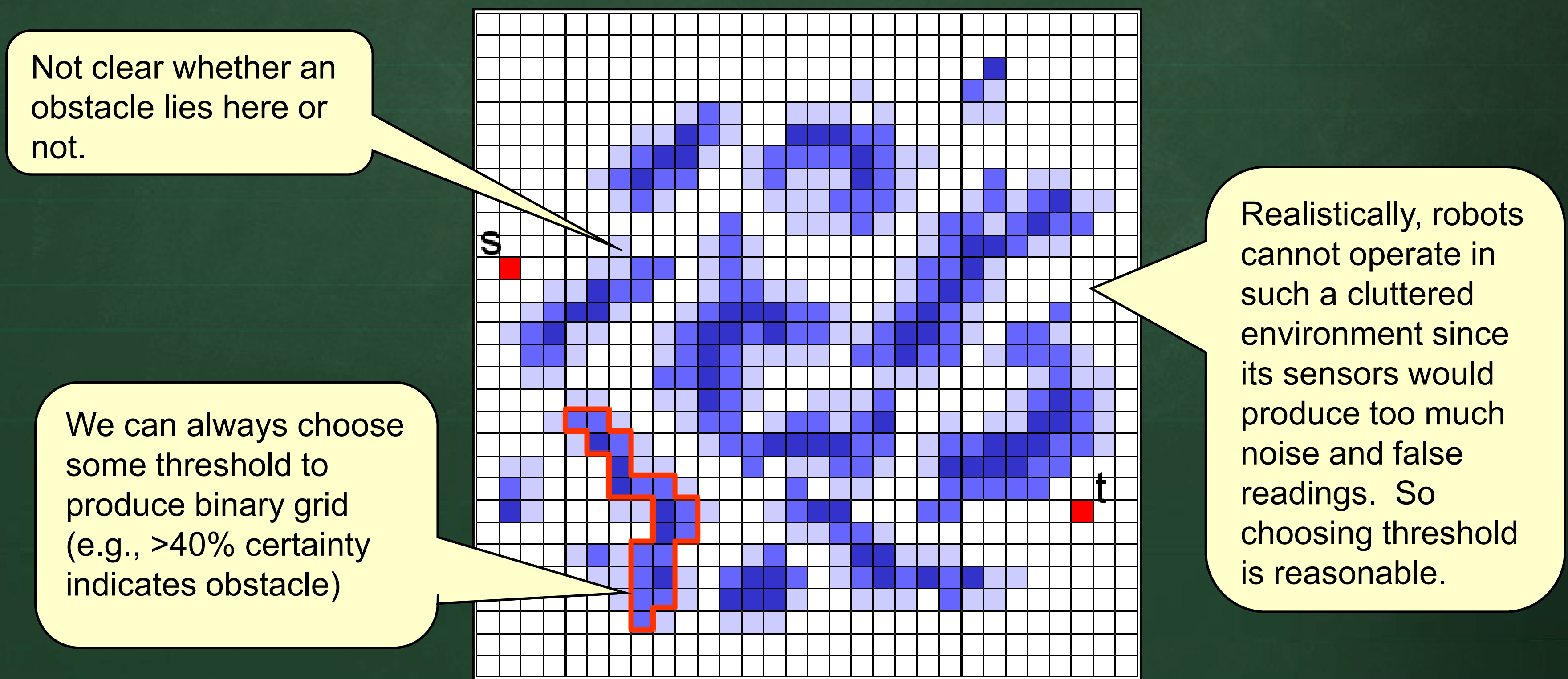
Shortest Paths in Grids

- Of course we can even do the reverse if we prefer to work with grids (i.e., convert vector to grid):



Shortest Paths in Grids

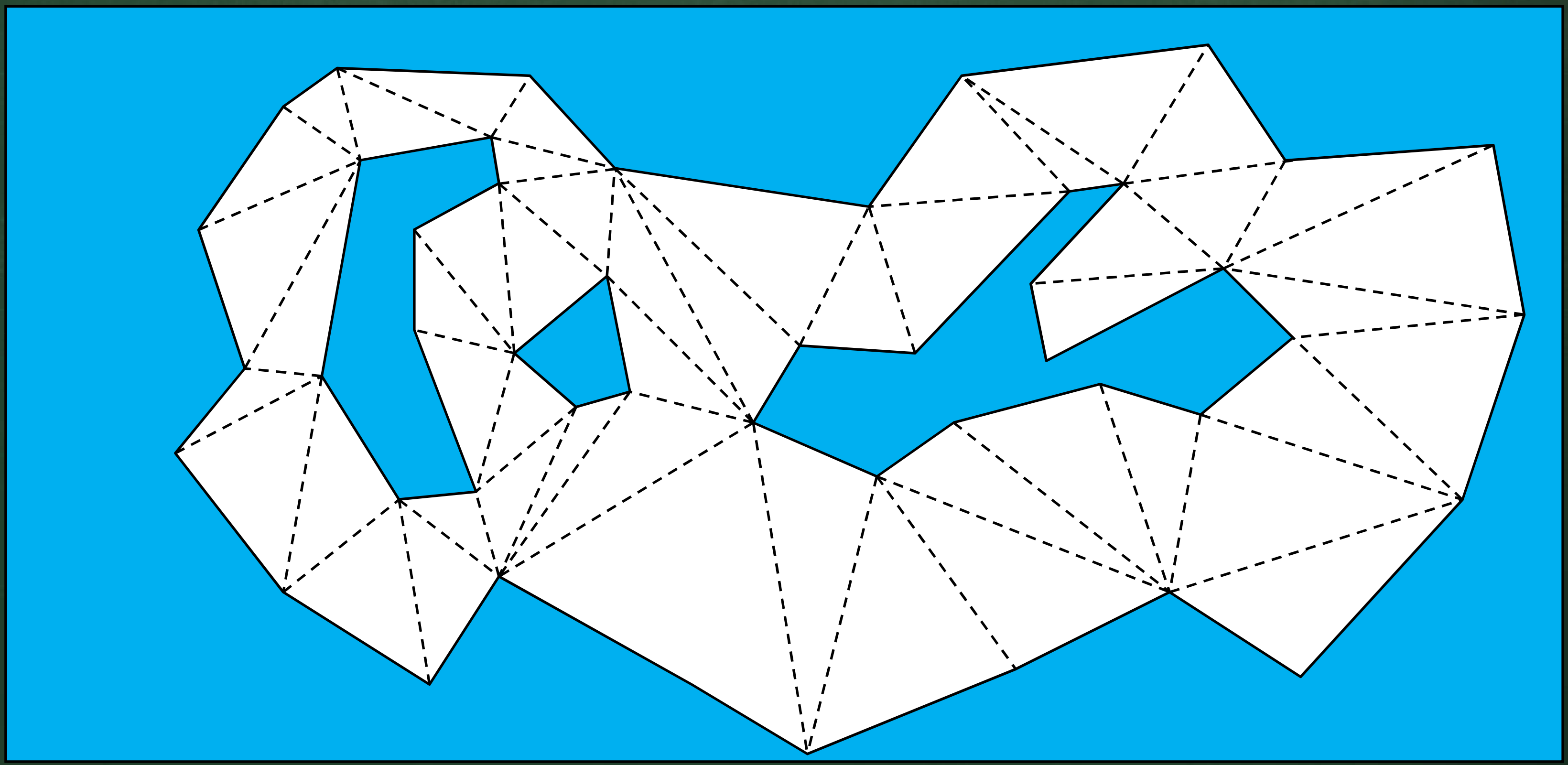
- A problem does arise however in the more realistic maps (i.e., certainty grids) since sensor data is noisy and we no longer have binary values.



Triangulation
Dual Graph Paths

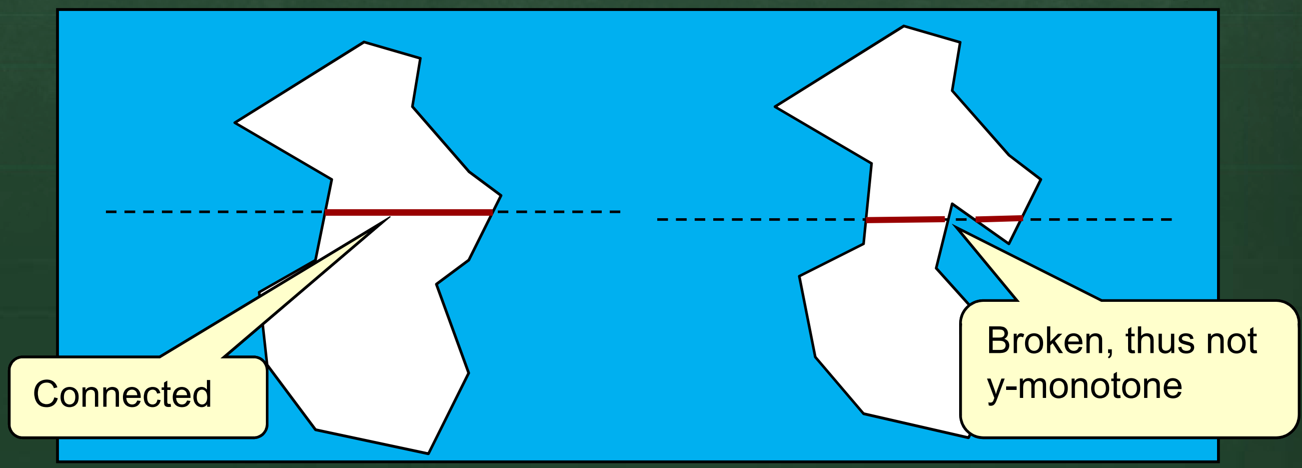
Triangulation

- A geometric strategy is based on computing a ***triangulation*** of the environment:
  - Decompose into triangular free-space regions



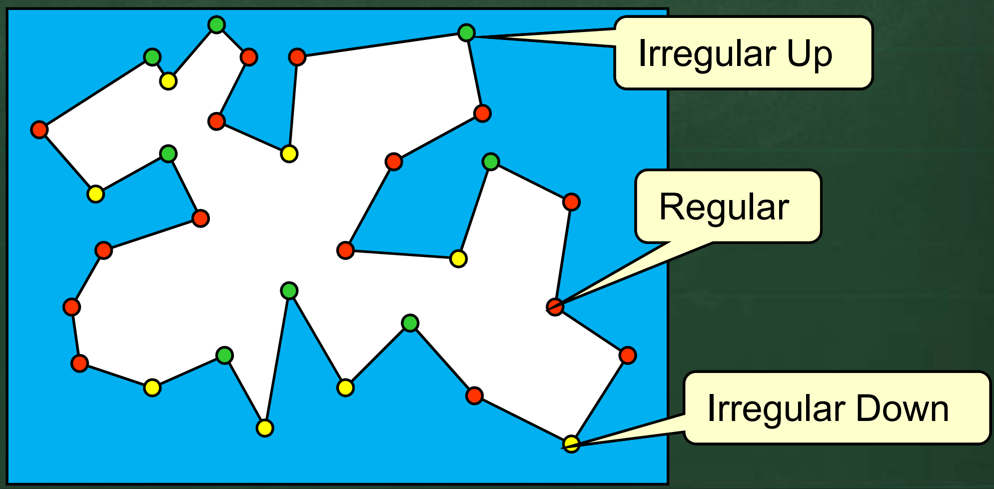
Triangulation

- There are MANY such triangulations and also many algorithms for obtaining them.
- One approach is to start by decomposing the free-space region into *y-monotone* polygons.
  - A simple polygon is called y-monotone if any horizontal line is connected.



Triangulation

- We need to understand different types of vertices:
  - A *regular vertex* is a vertex that is adjacent (connected to) at least one vertex with a larger y-coordinate and one with a smaller y-coordinate.
  - A *irregular up vertex* is a vertex that is not connected to any vertices with a larger y-coordinate.
  - A *irregular down vertex* is a vertex that is not connected to any vertices with a smaller y-coordinate.



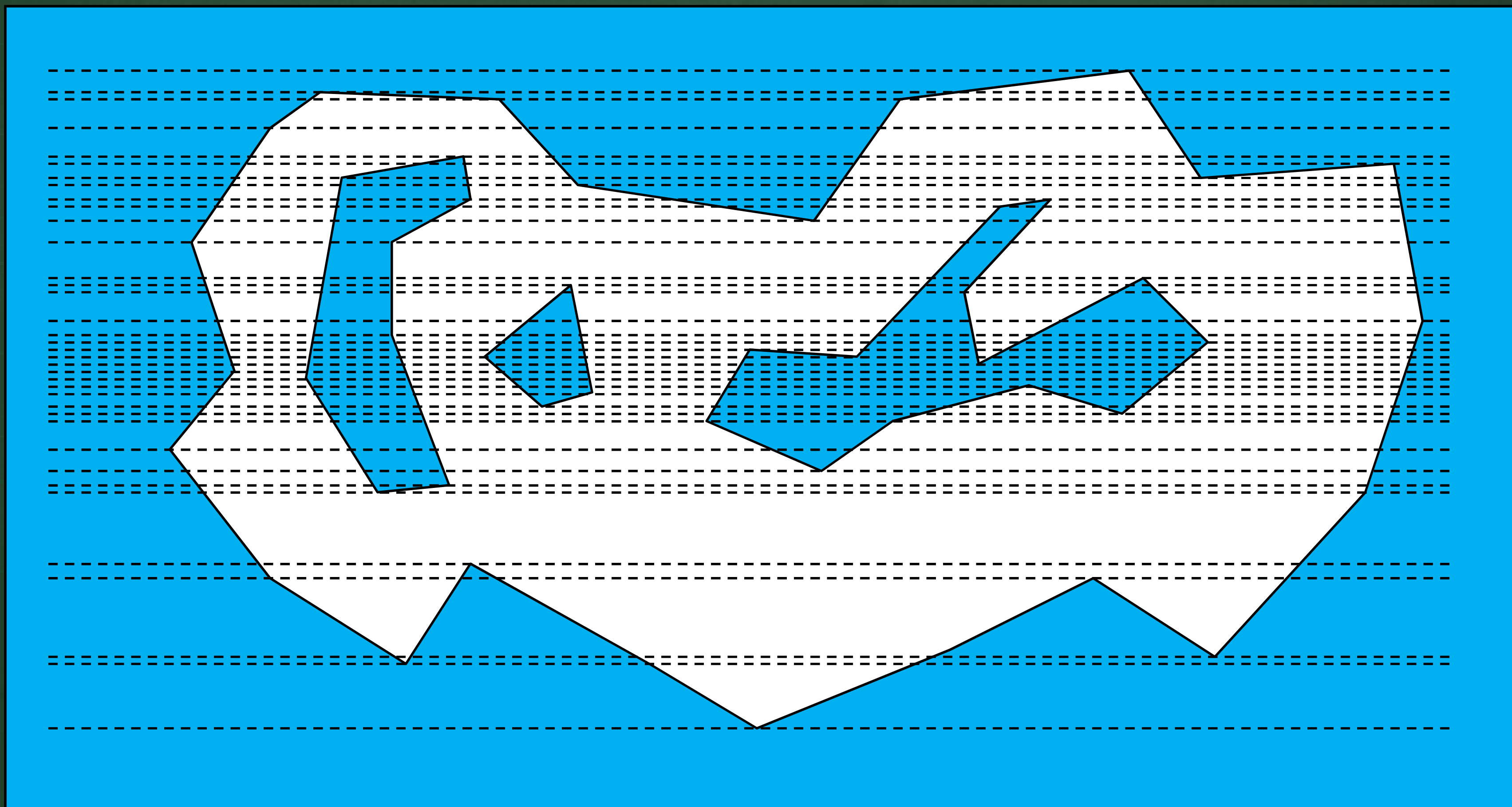
Triangulation

- We need to *regularize* the polygon with holes:
  - Break it into a subgraph such that all vertices are regular except the most extreme vertices in the y direction.

- Result is a decomposition into monotone pieces.
- Basic idea:
  - Vertical sweep from top to bottom regularizing vertices that are irregular down
  - Vertical sweep from bottom to top, regularizing vertices that are irregular up.

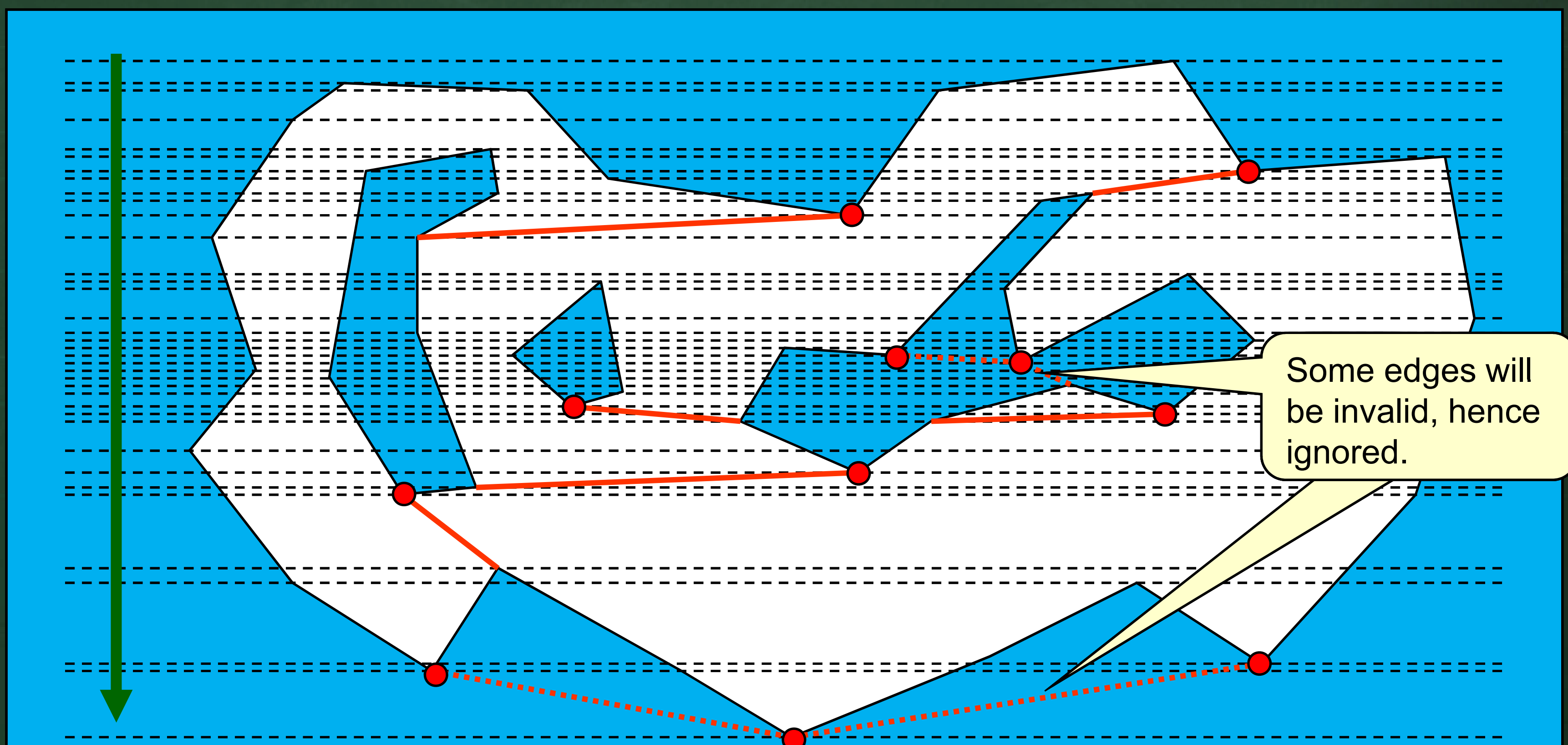
Triangulation

- Can do this by first sorting vertices in vertical order:



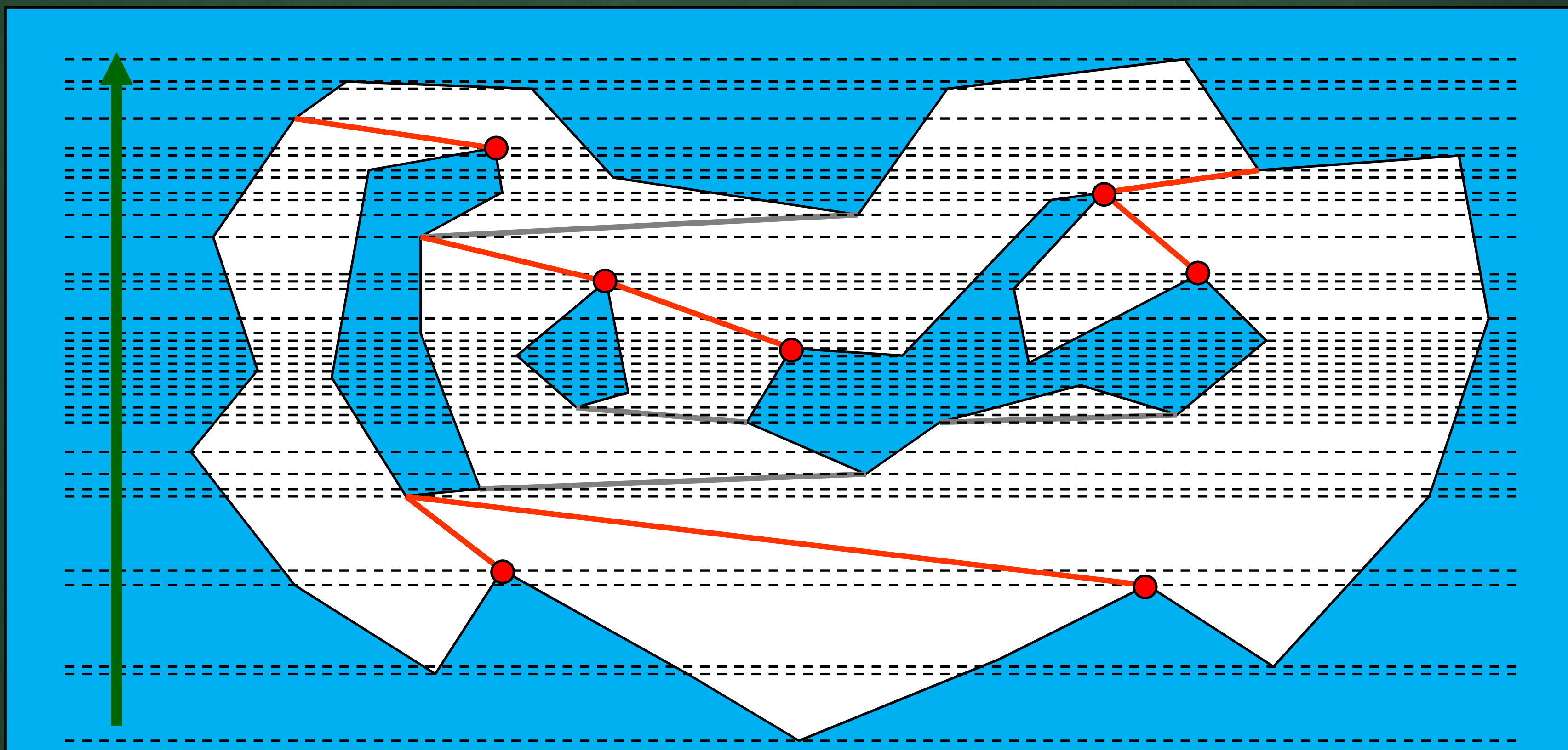
Triangulation

- Perform vertical sweep downwards connecting irregular down vertices to the next nearby vertex:



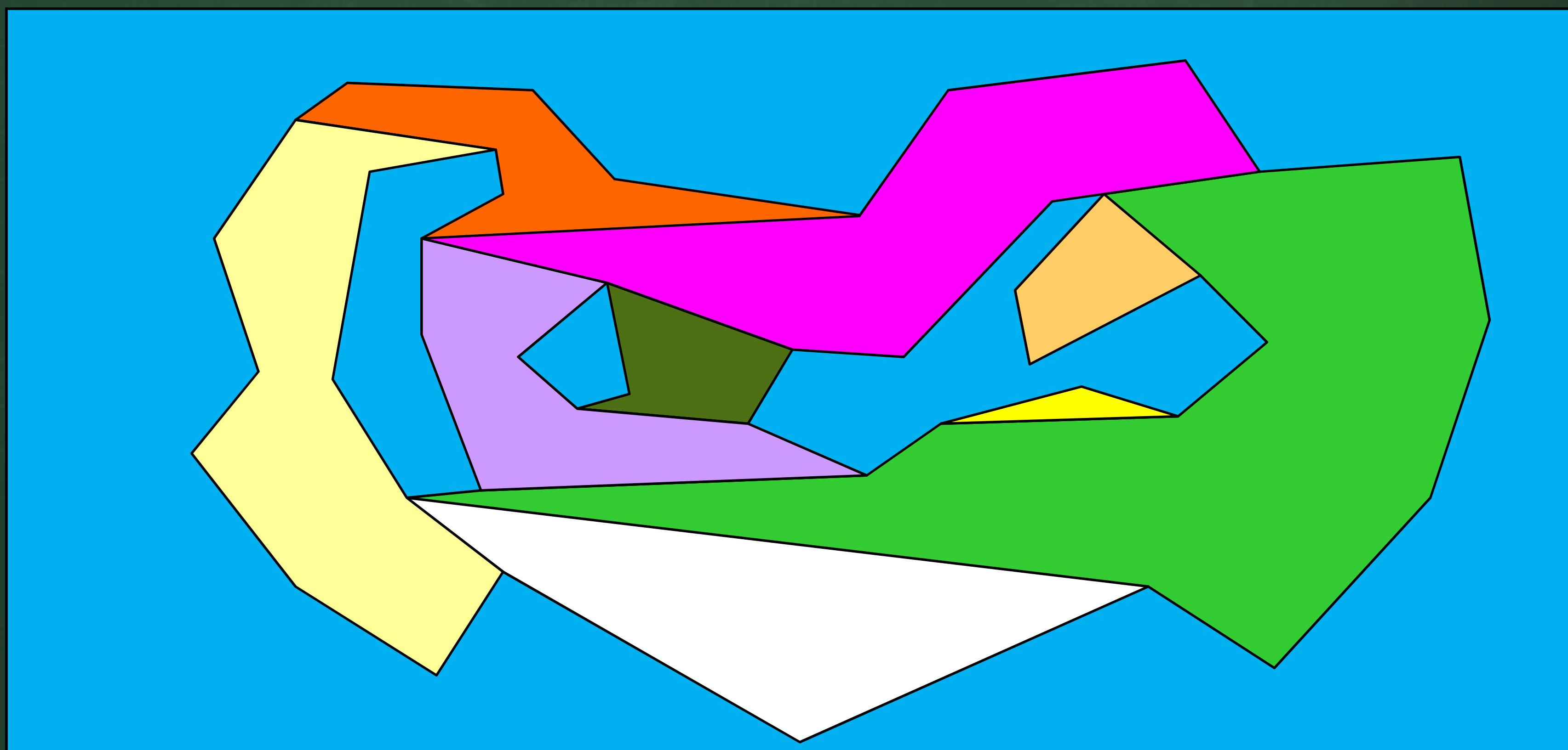
Triangulation

- Perform vertical sweep upwards connecting irregular up vertices to the next nearby vertex:



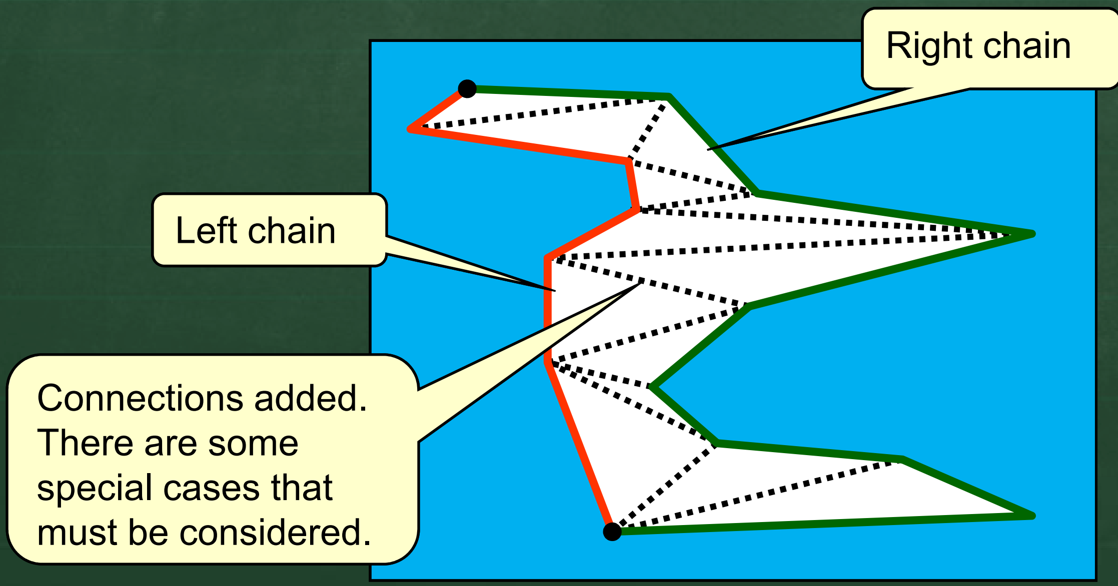
Triangulation

- Some details have been left out, but result is a set of monotone polygons:



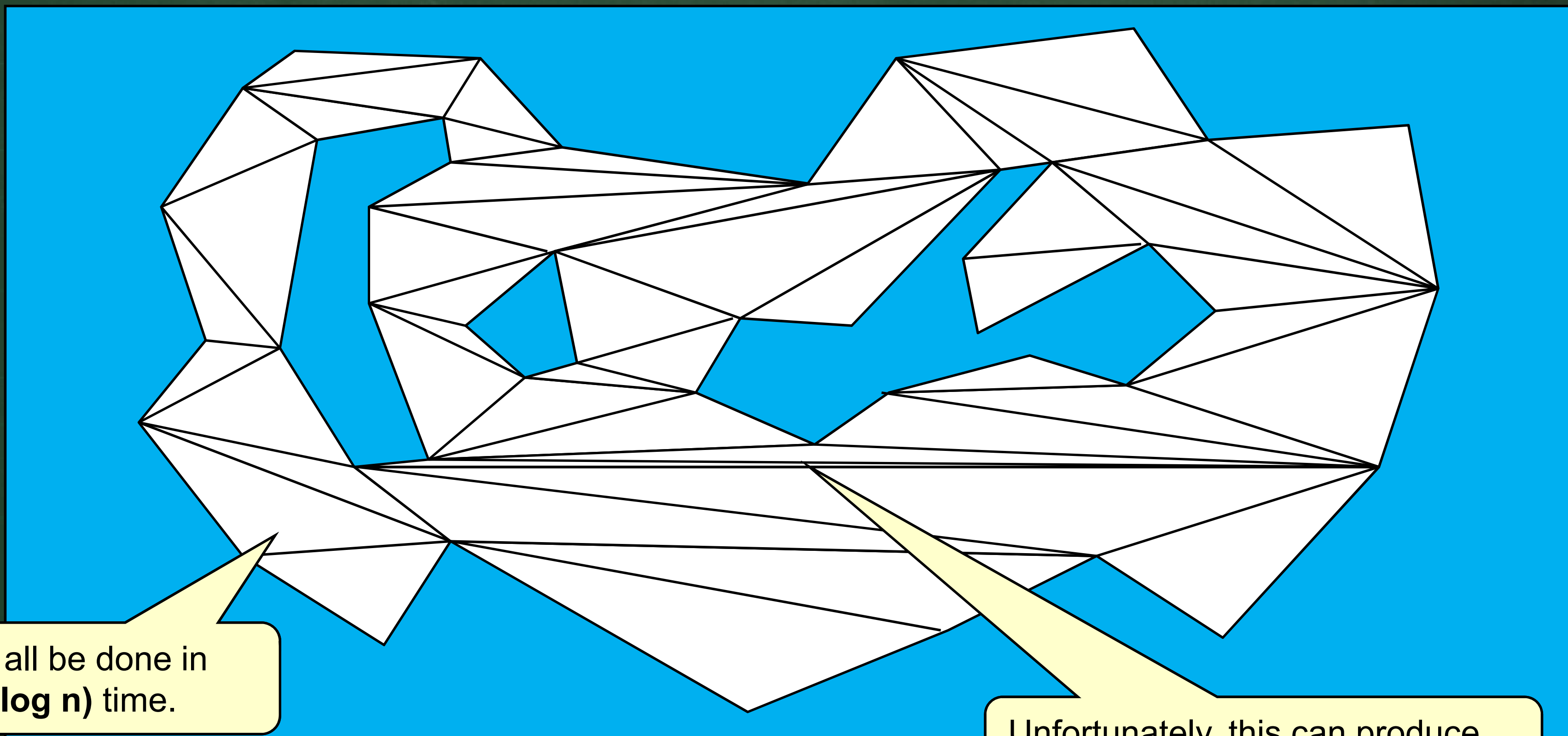
Triangulation

- Monotone pieces each triangulated separately:
  - Do vertical sweep downward, connect vertices from left monotone chain to right



Triangulation

- Monotone pieces each triangulated separately, then results are joined together:

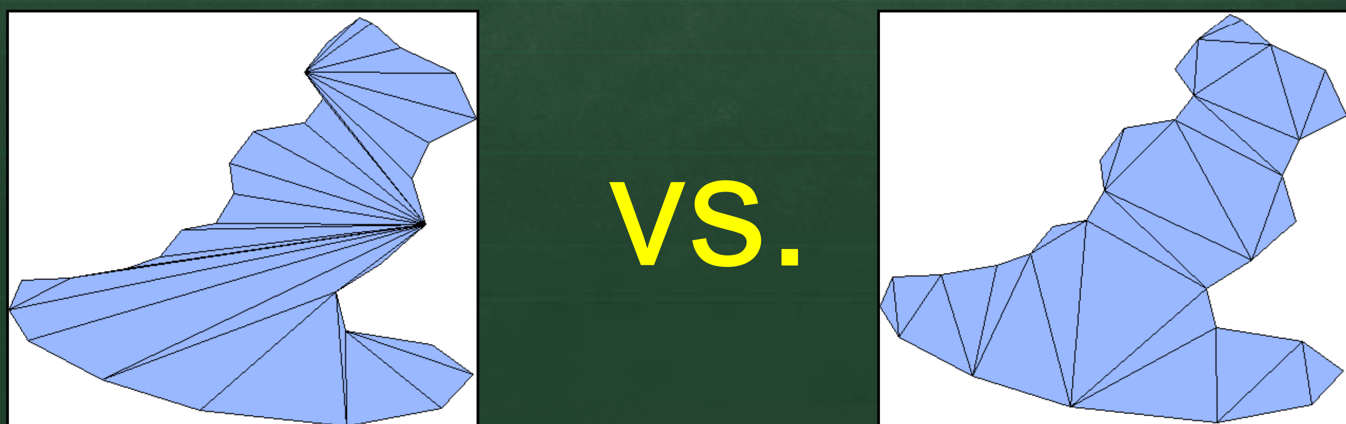


Can all be done in **O(n log n)** time.

Unfortunately, this can produce long/thin triangles.

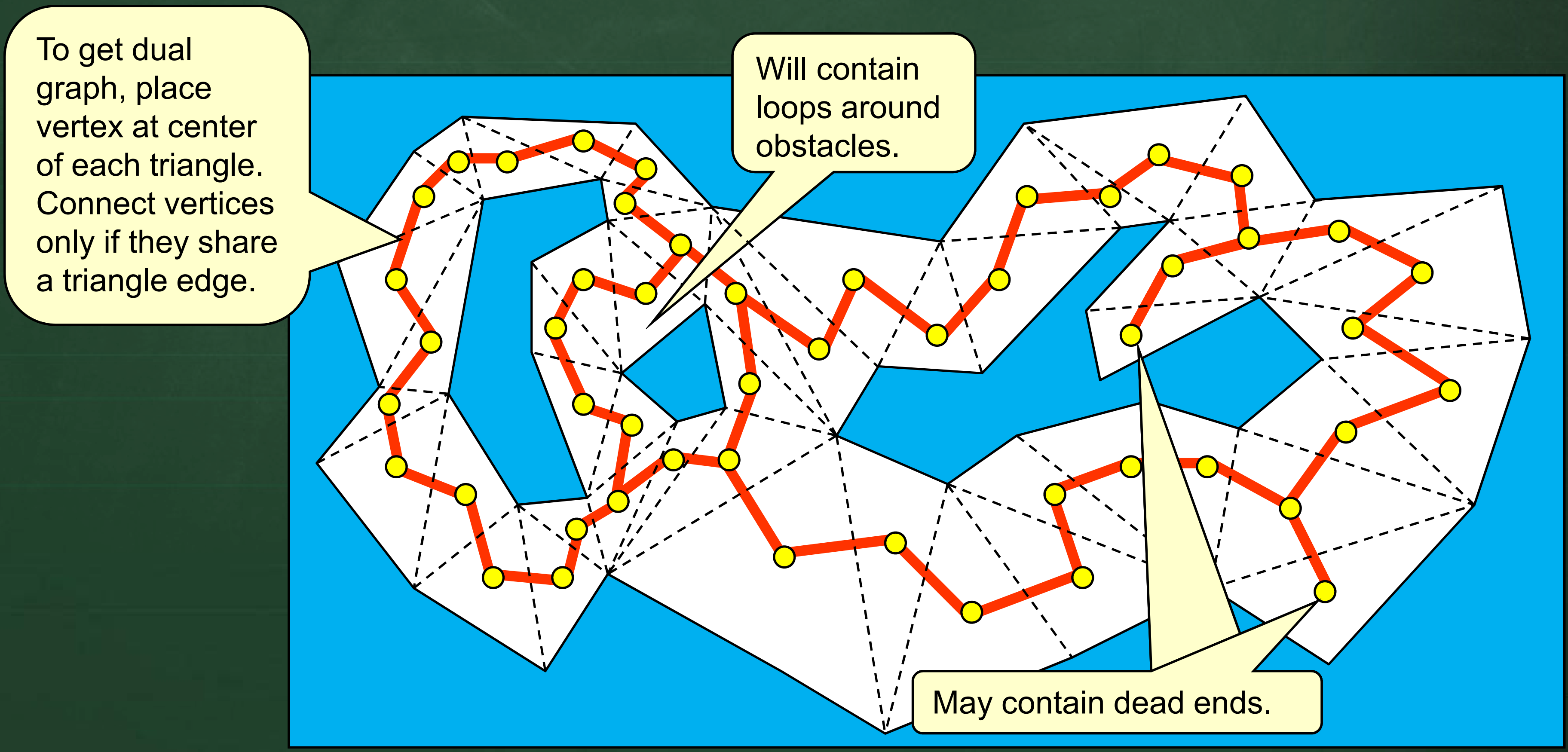
Triangulation

- Better algorithm is a *Constrained Delaunay Triangulation*
  - Produces "fatter" triangles
  - Nicer looking decomposition
  - More complicated, but still practical
  - Beyond the scope of <u>this</u> course



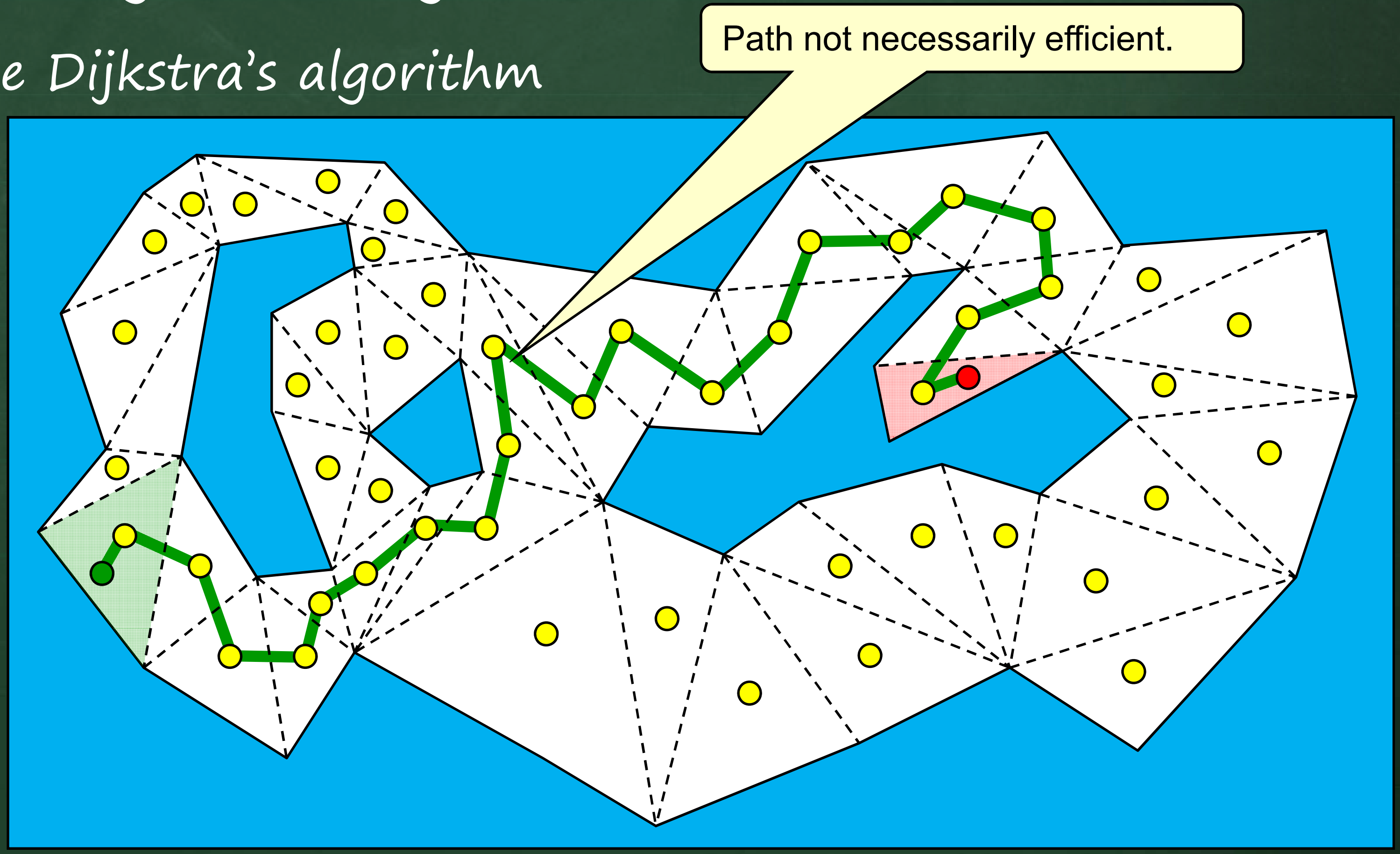
The Dual Graph

- Compute the *dual graph* which gives a rough idea as to the paths that the robot may travel.



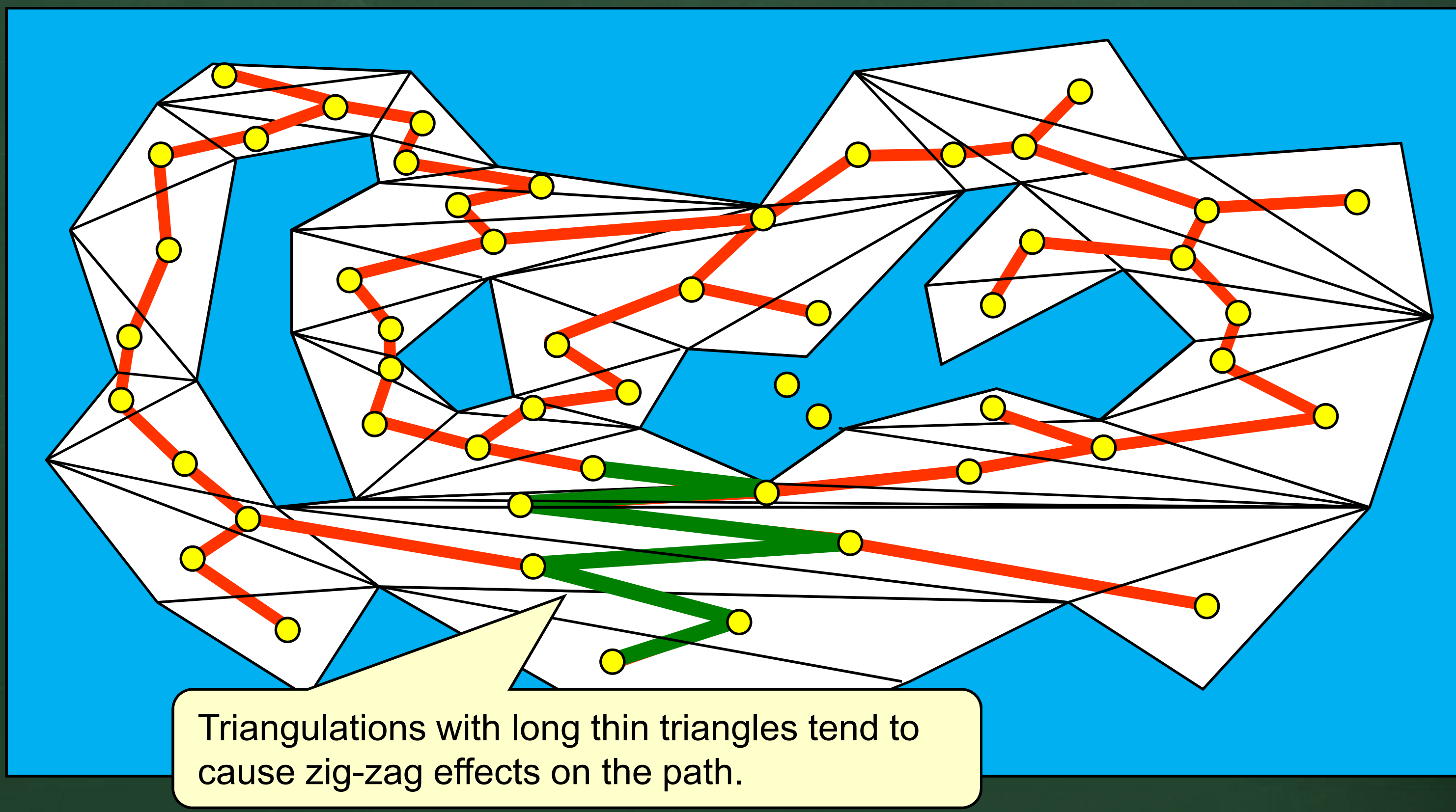
Computing a Path

- Robot can compute path in dual graph from start triangle to goal triangle:
  - Can use Dijkstra's algorithm



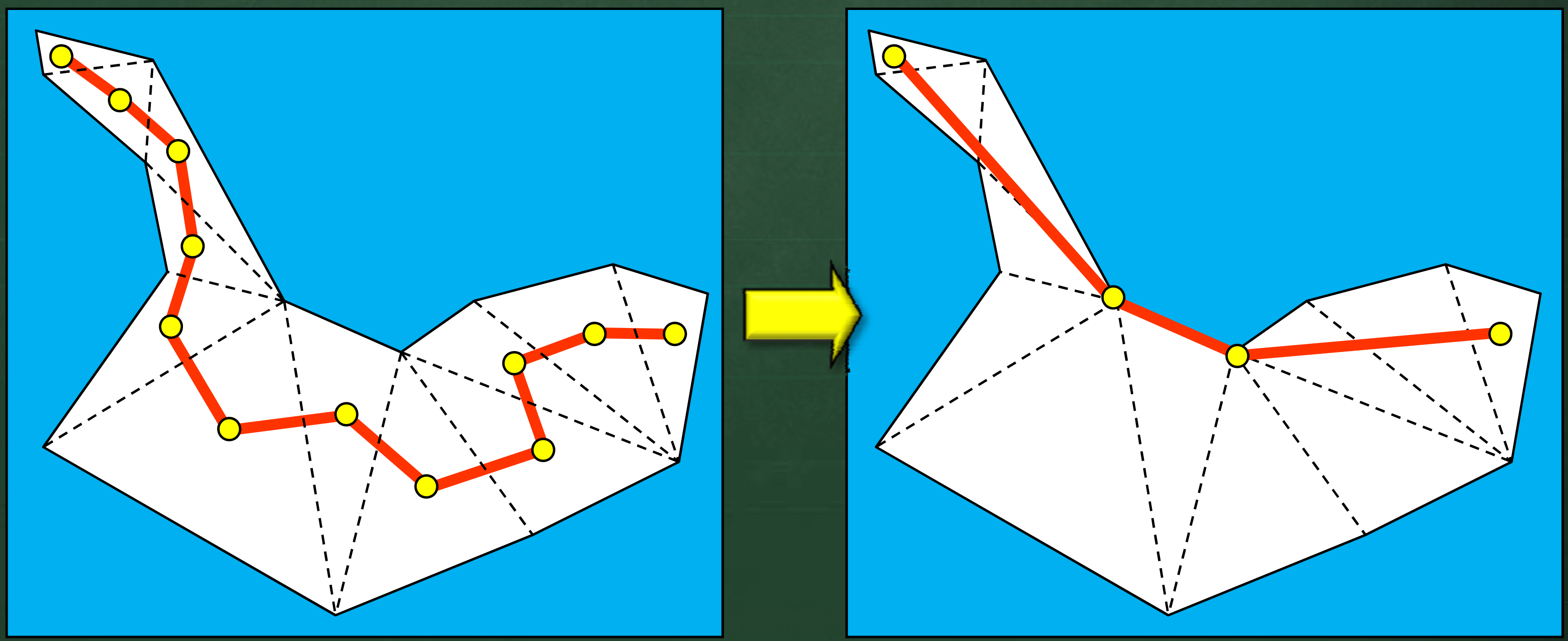
Computing a Path

- The efficiency of the path solutions are highly dependant on the triangulation:



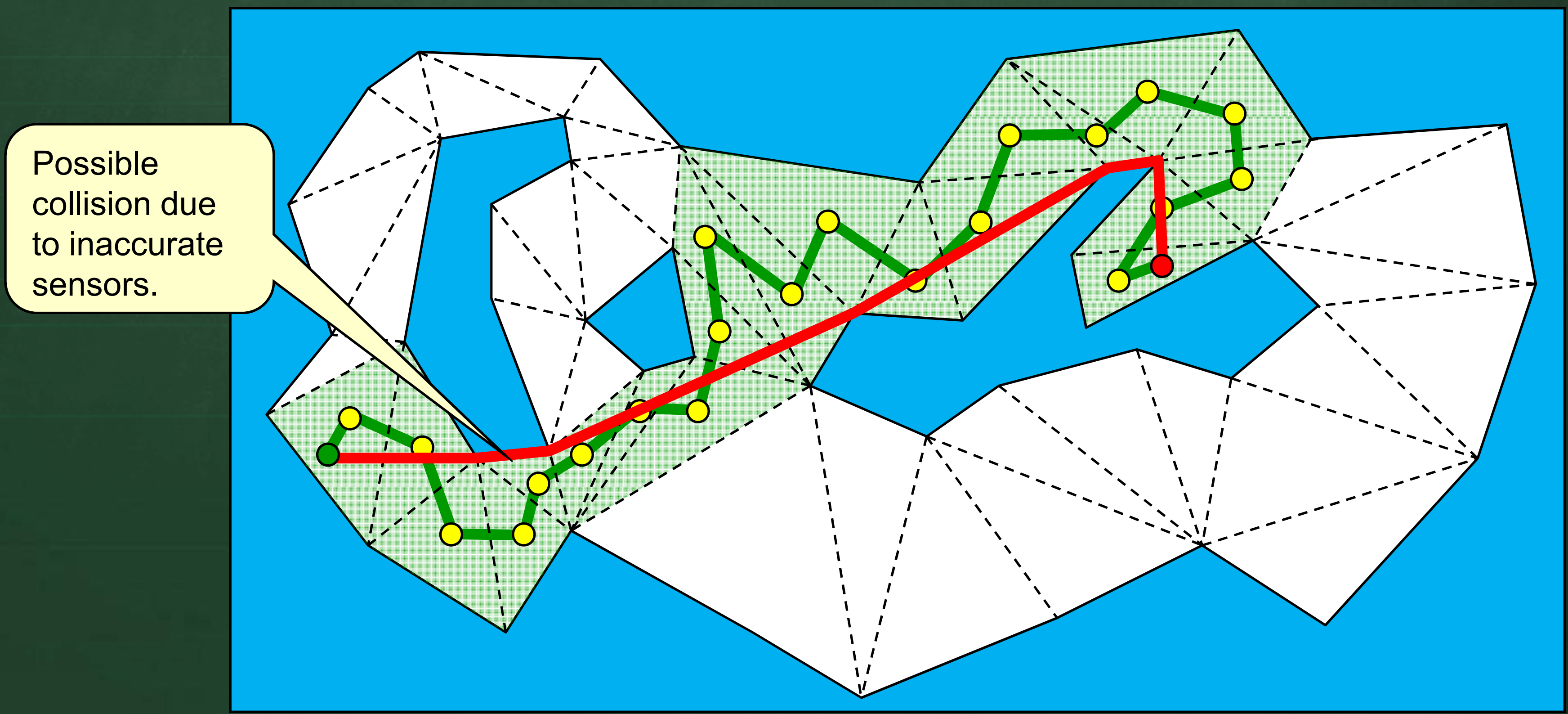
Refining the Path

- Can also simplify (i.e., refine) any path in the dual graph by computing the shortest path in the ***sleeve*** formed by connected triangles:



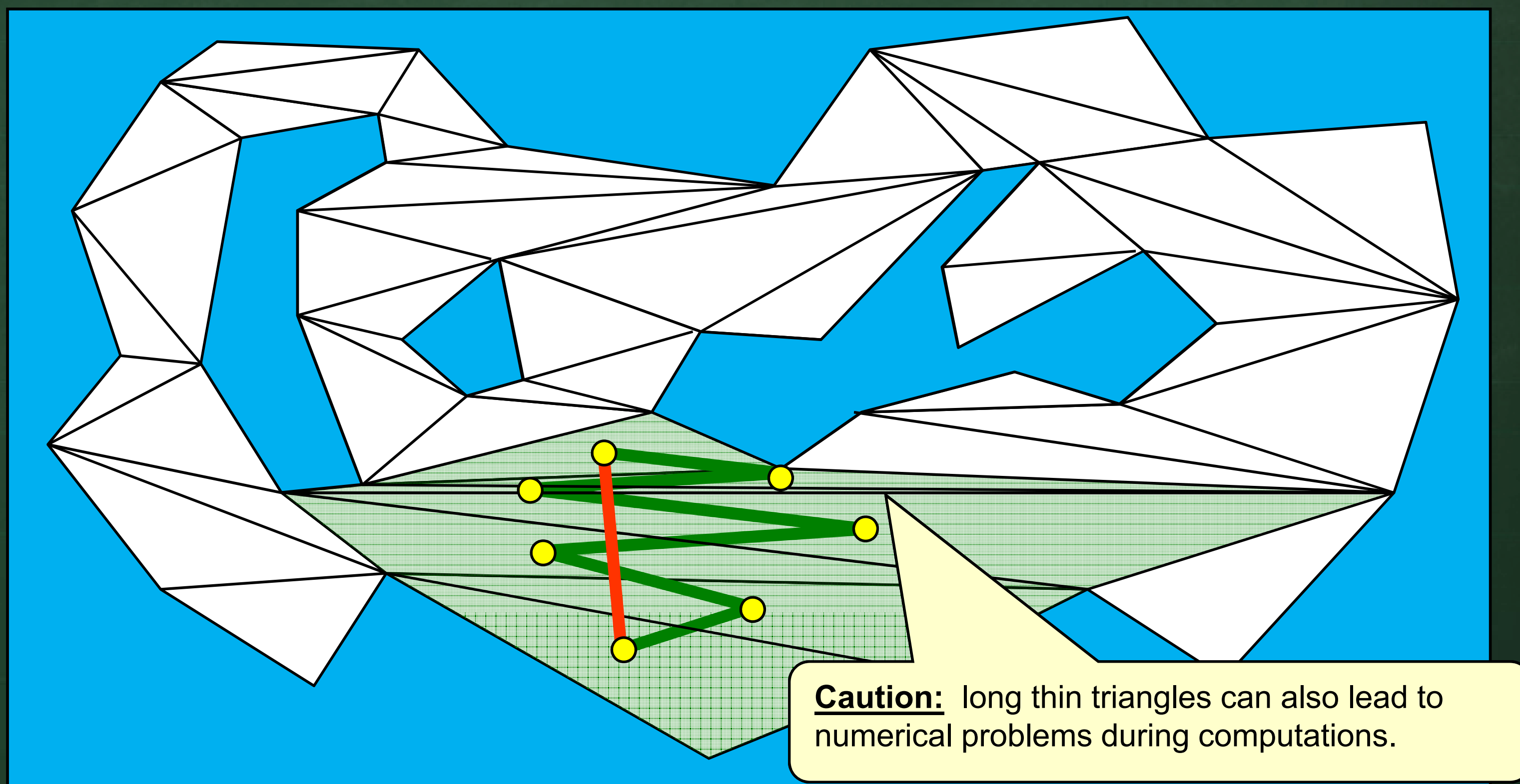
Refining the Path

- As a result, the computed path will be more efficient in terms of length
  - But path will generally travel close to boundaries.



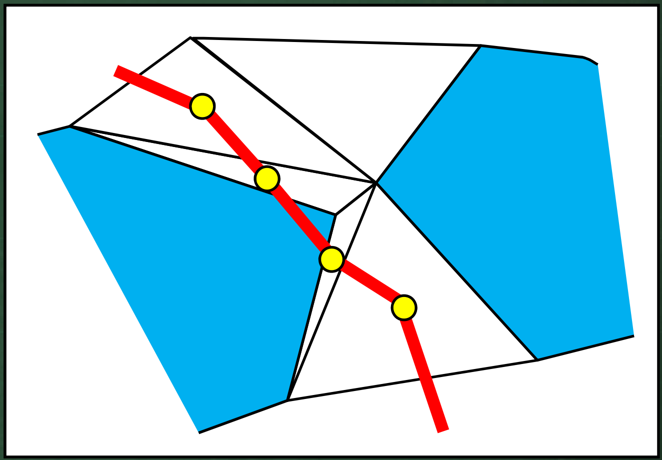
Refining the Path

- The zig-zag effect essentially disappears when path is refined.

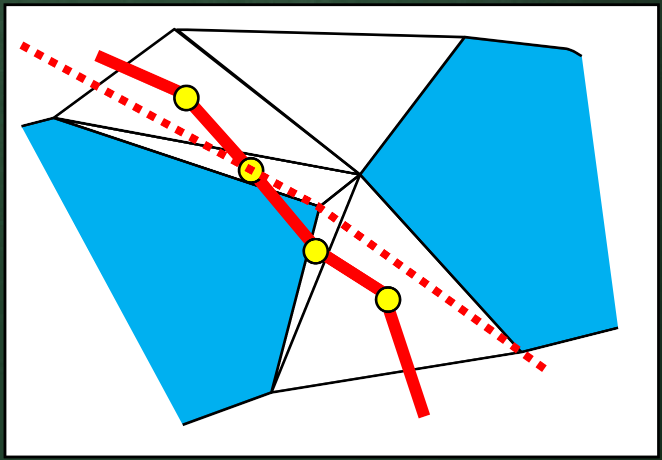


Problems

- If traveling between centers of triangles, this could be dangerous, for thin triangles:

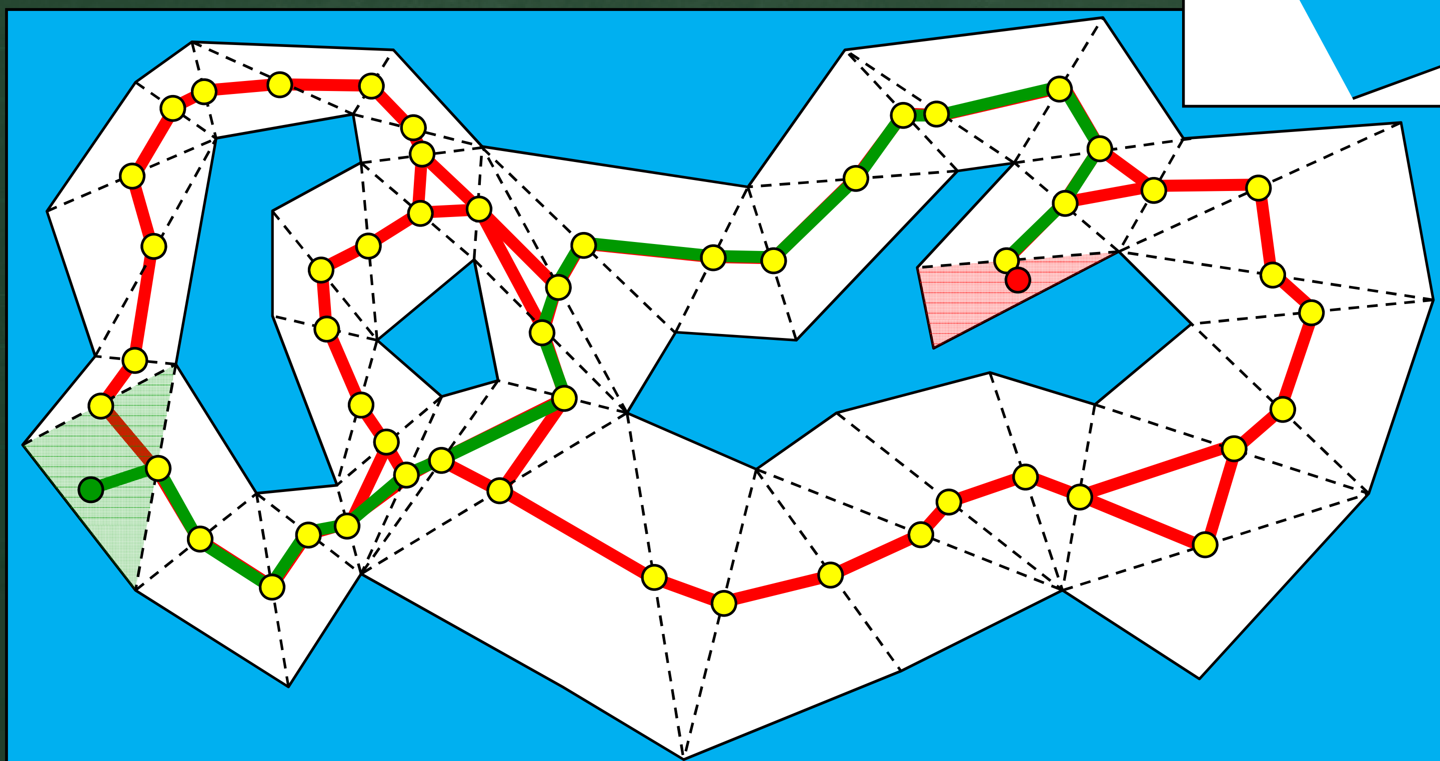


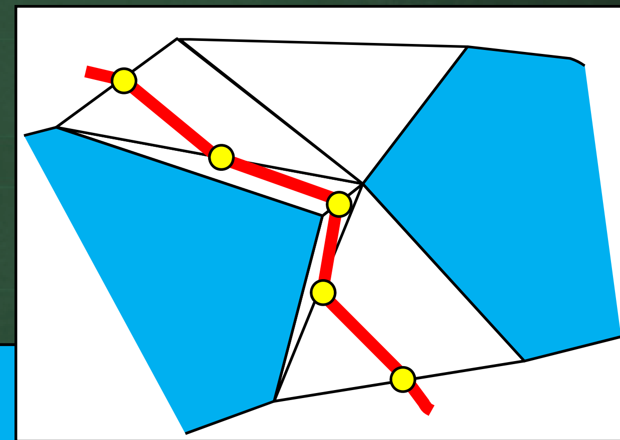
- Computing refined path will always correct this:



Problems

- Alternatively, we can connect vertices at midpoints of triangulation edges:





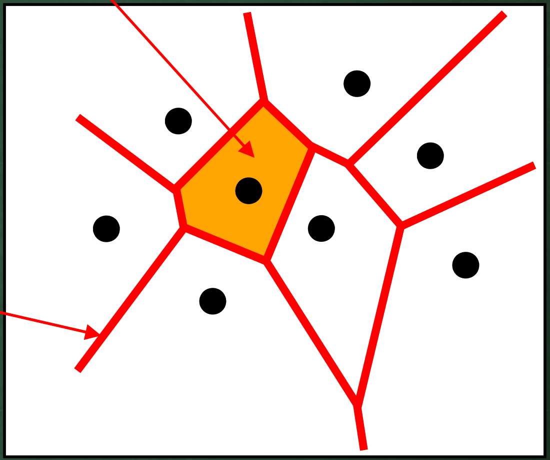
Generalized Voronoi Diagram Paths

Voronoi Road Maps

- A ***Voronoi road map*** is a set of paths in an environment that represent maximum clearance between obstacles.
- They are sometimes preferred in robotics since they reduce the chance of collisions because sensors are often inaccurate and prone to error.
- Other names for this roadmap are ***generalized Voronoi diagram*** and ***retraction method***.
- It is considered as a generalization of the Voronoi diagram for points.

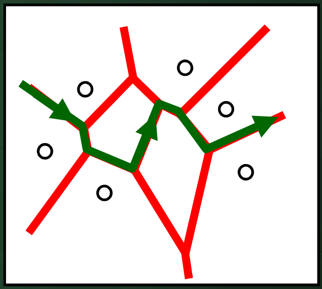
Voronoi Diagram

- Let *S* be a set of *n* points. For each point *p* of *S*, the ***Voronoi cell*** of *p* is the set of points that are closer to *p* than to any other points of *S*.



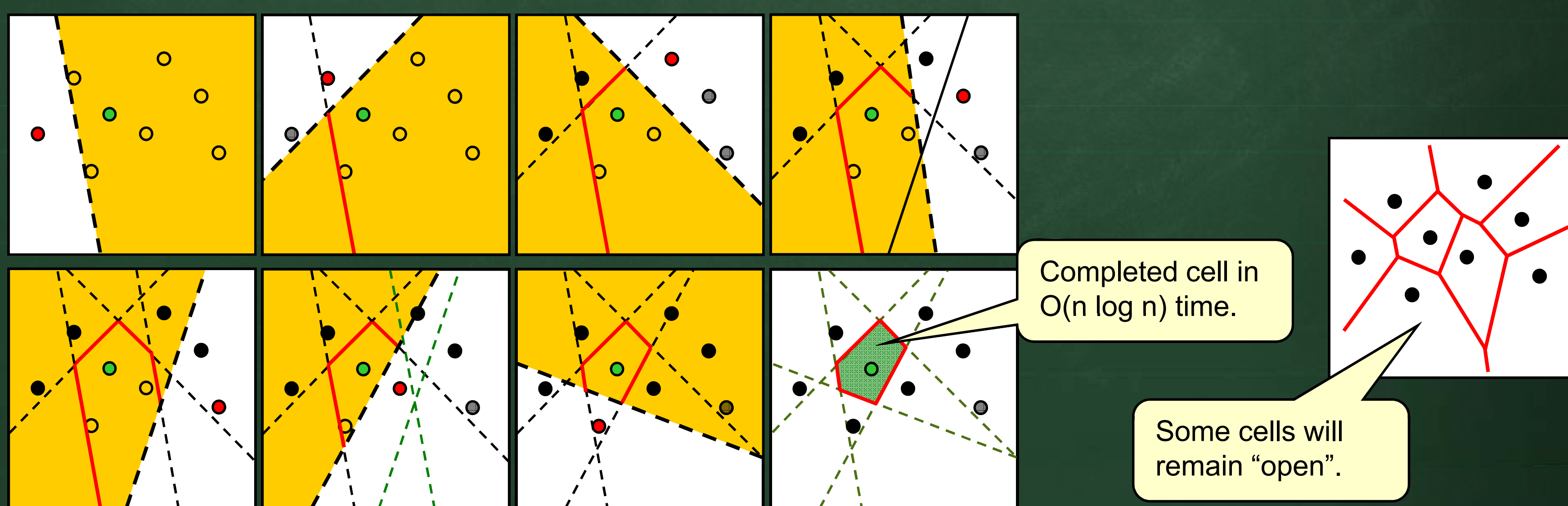
- The ***Voronoi diagram*** is the space partition induced by Voronoi cells.

- If the points were obstacles, a robot would travel along the edges of a Voronoi diagram if it wanted to keep maximum distance away from the obstacles.



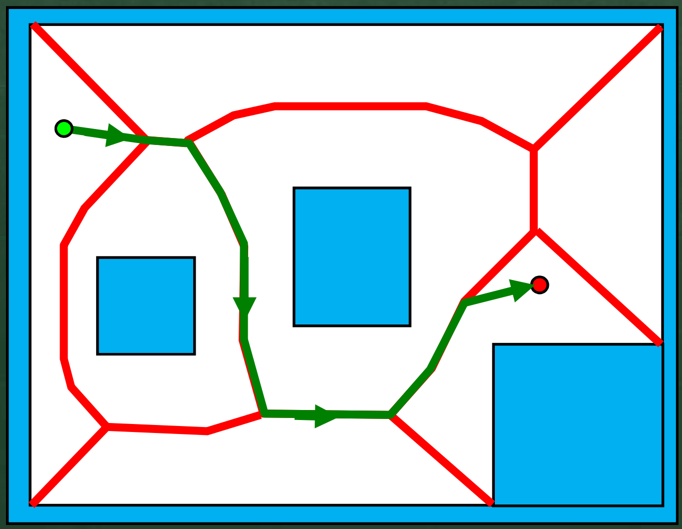
Voronoi Diagram

- Multiple ways of computing a Voronoi Diagram.
  - We consider the simplest, and leave the more advanced algorithms for a computational geometry course.
- Basically, we compute each Voronoi cell as the intersection of a set of half-planes: $O(n^2 \log n)$ time.



Generalized Voronoi Diagram

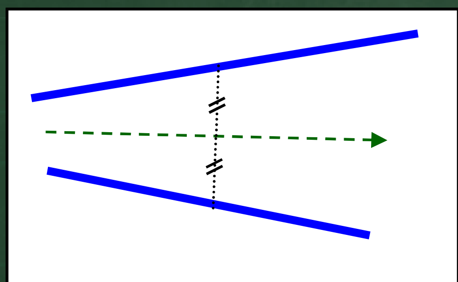
- What if the obstacles are polygons ?
- Now we compute the *Generalized Voronoi Diagram* in which the edges forming it maintain maximal distance between edges of the environment, as opposed to just points.



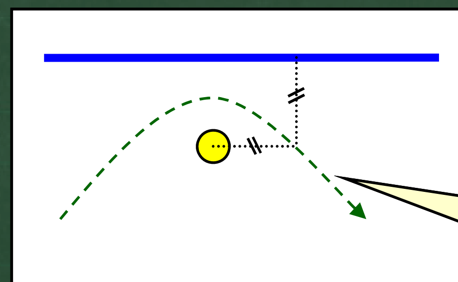
Generalized Voronoi Diagram

- Edges formed based on three types of interaction:

Edge-Edge

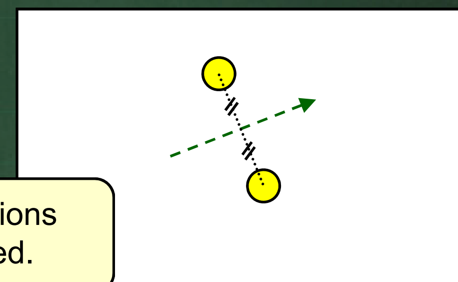


Edge-Vertex



Certain portions will be curved.

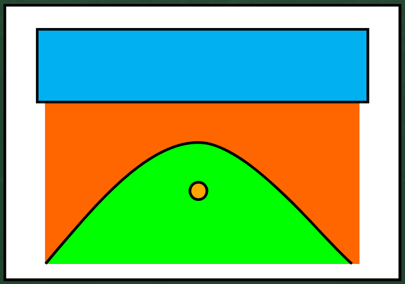
Vertex-Vertex



- There are different ways of computing this diagram:
  - Exact computation
  - Approximation – Discretize Obstacles
  - Approximation – Discretize Space

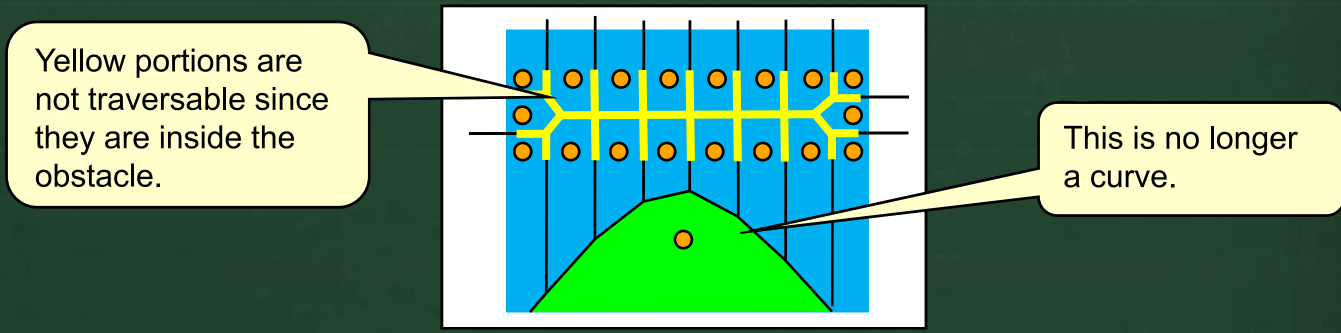
Computing the GVD

- Exact computation
  - Based on computing analytic boundaries
  - Boundaries may be composed of high-degree curves and surfaces and their intersections.
  - Complex and difficult to implement
  - Robustness and accuracy problems



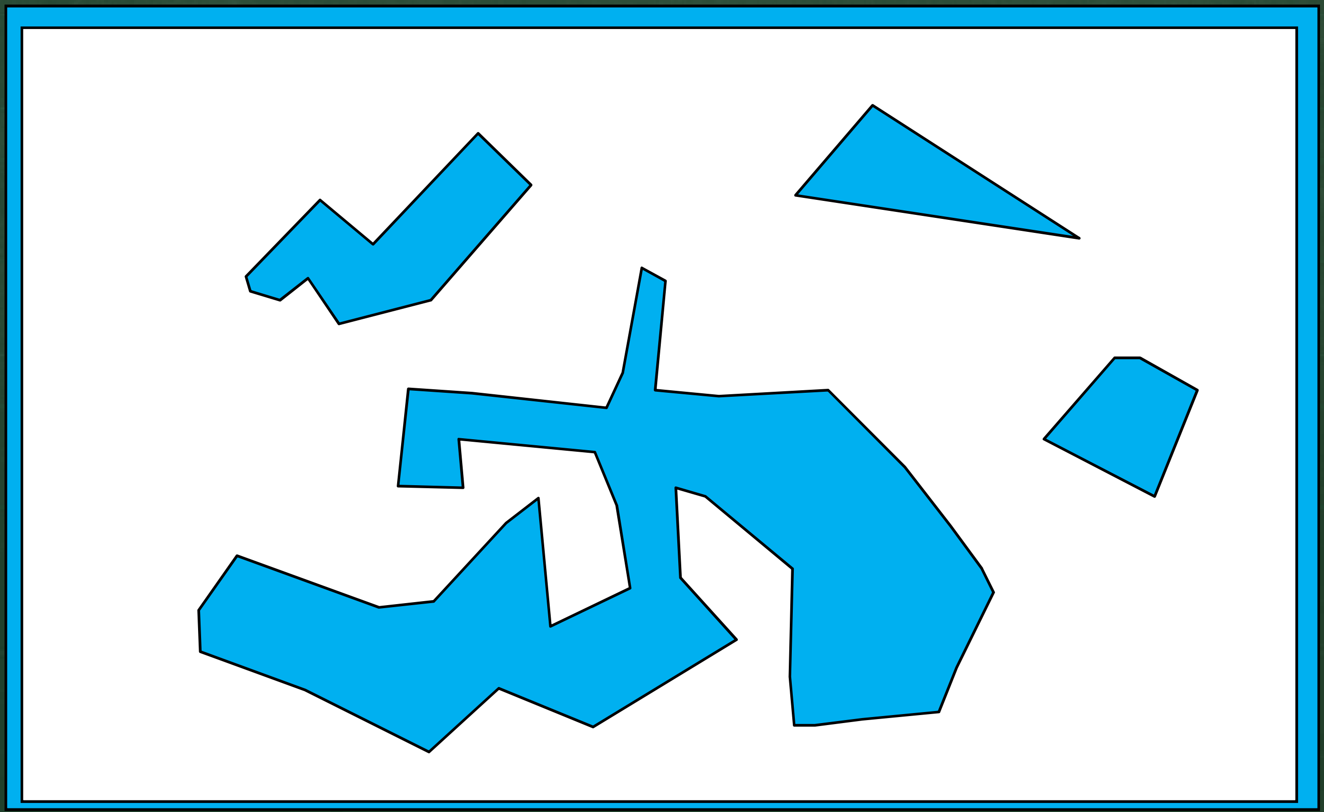
GVD Approximation – Method 1

- Approximation – Discretize Obstacles
  - Convert each obstacle into a set of points by selecting samples along boundaries.
  - Compute regular Voronoi diagram on resulting point sets.
  - Will produce some diagram edges that are not traversable. Must prevent travel along these portions.
  - Can be slow to compute, depending on samples.



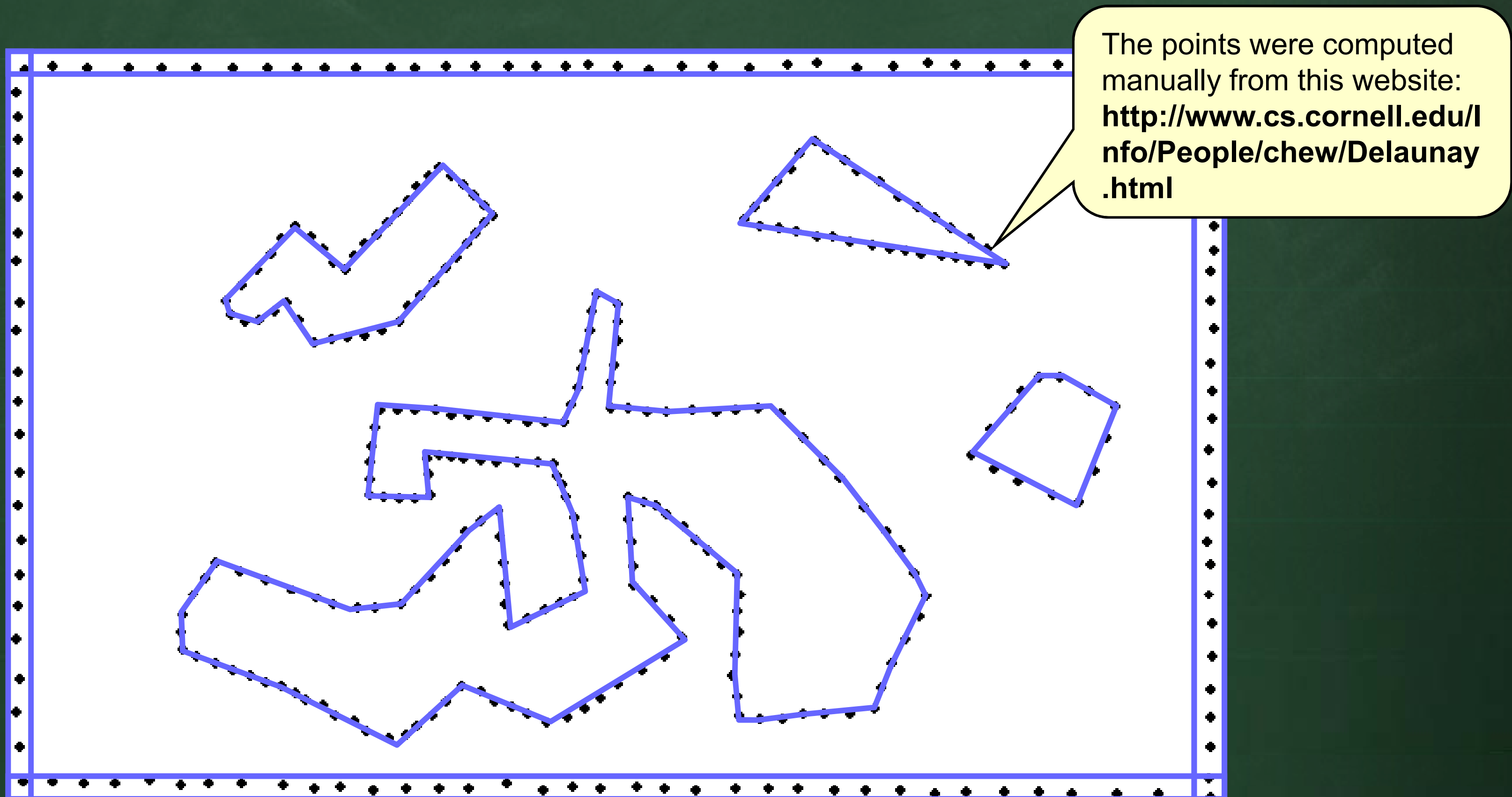
GVD Approximation – Method 1

- Approximation – Discretize Obstacles (continued)
  - Consider computing the GVD for the following example:



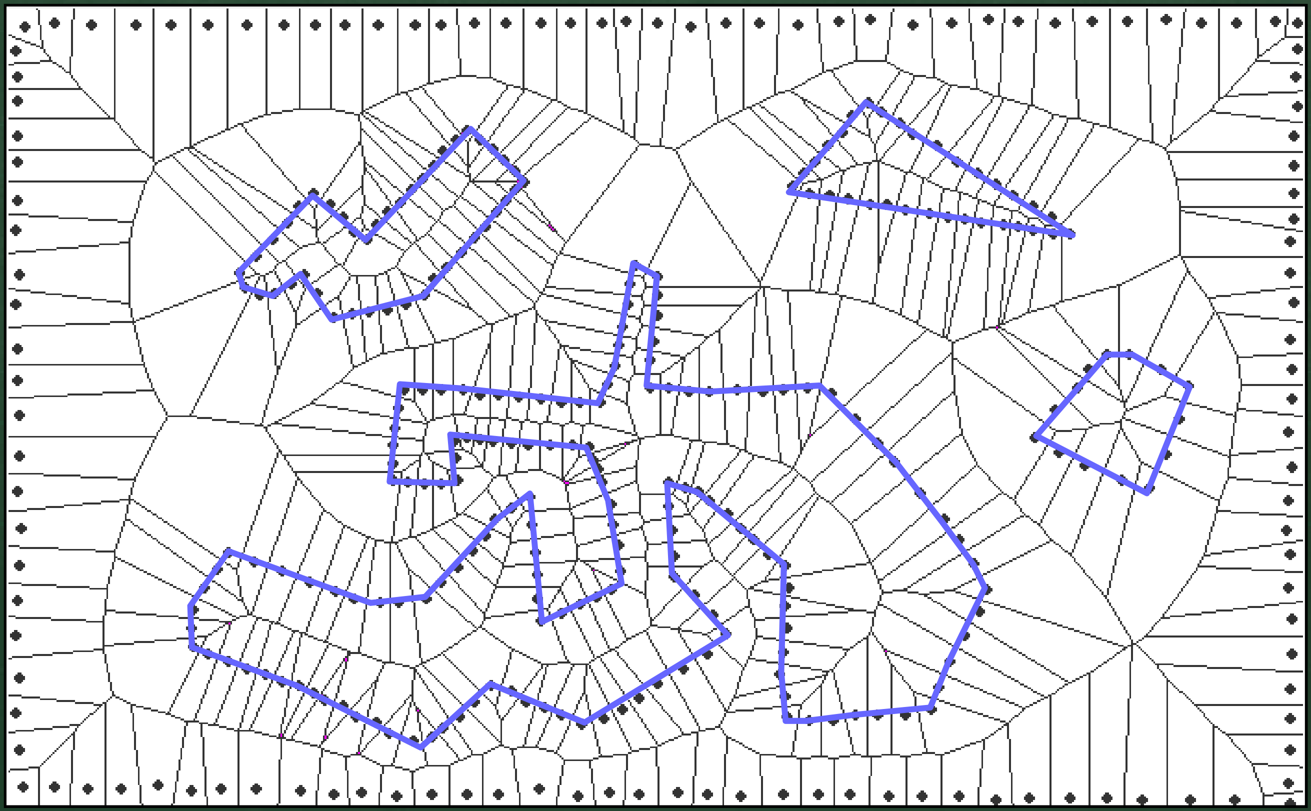
GVD Approximation – Method 1

- Approximation – Discretize Obstacles (continued)
  - Compute sample points along obstacle border.



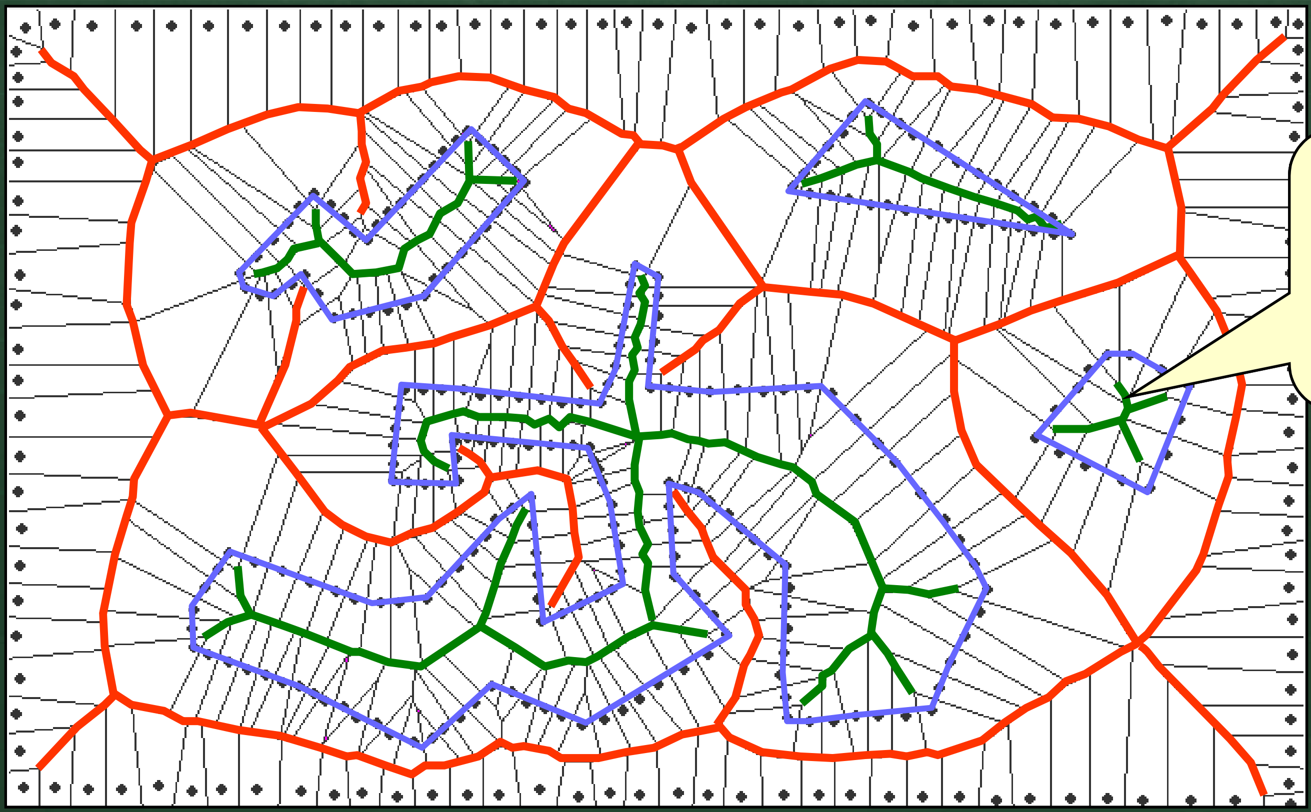
GVD Approximation – Method 1

- Approximation – Discretize Obstacles (continued)
  - Here is the Voronoi Diagram for the point set:



GVD Approximation – Method 1

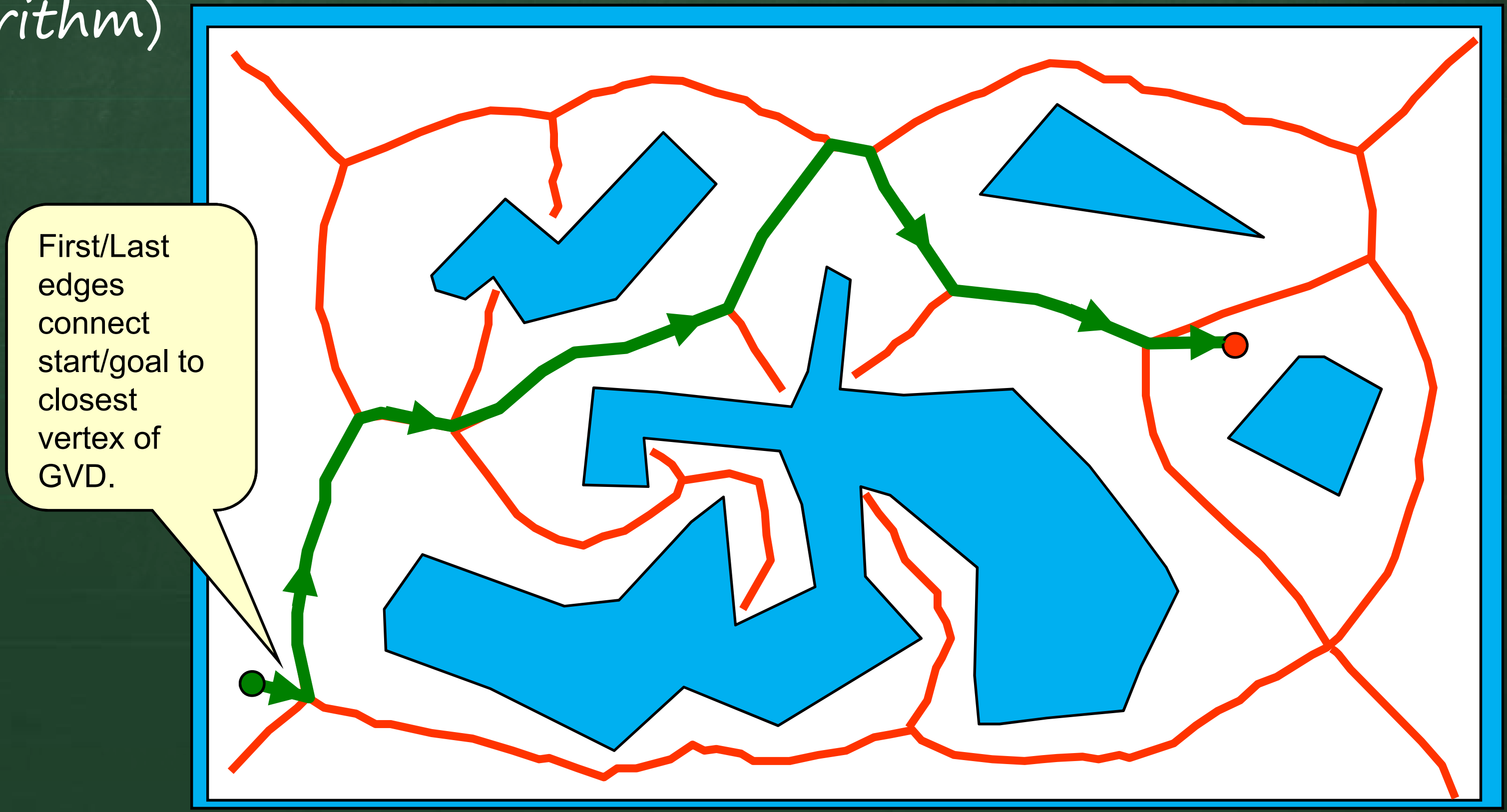
- Approximation – Discretize Obstacles (continued)
  – Can discard (ignore) all edges of GVD that are defined by two consecutive points from the same obstacle:



Can also discard all edges that lie completely interior to any obstacle (i.e., green ones here).

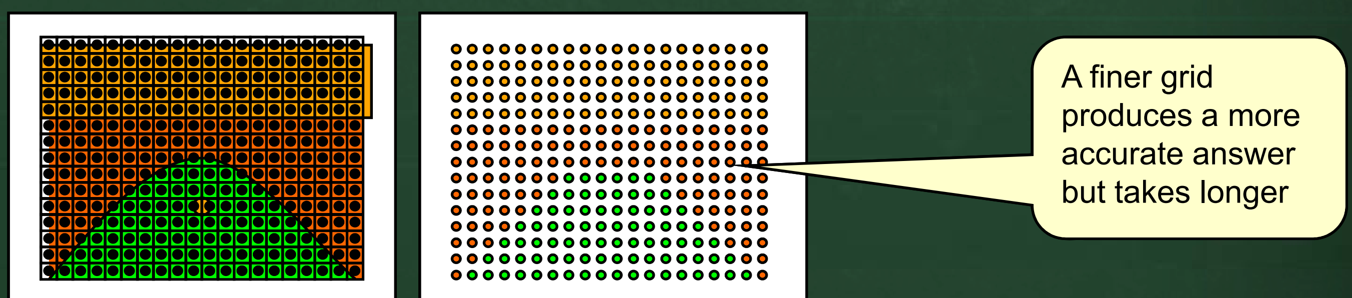
GVD Approximation – Method 1

- Approximation – Discretize Obstacles (continued)
  - Resulting GVD edges can be searched for a path from start to goal (e.g., store GVD as graph, run Dijkstra's shortest path algorithm)



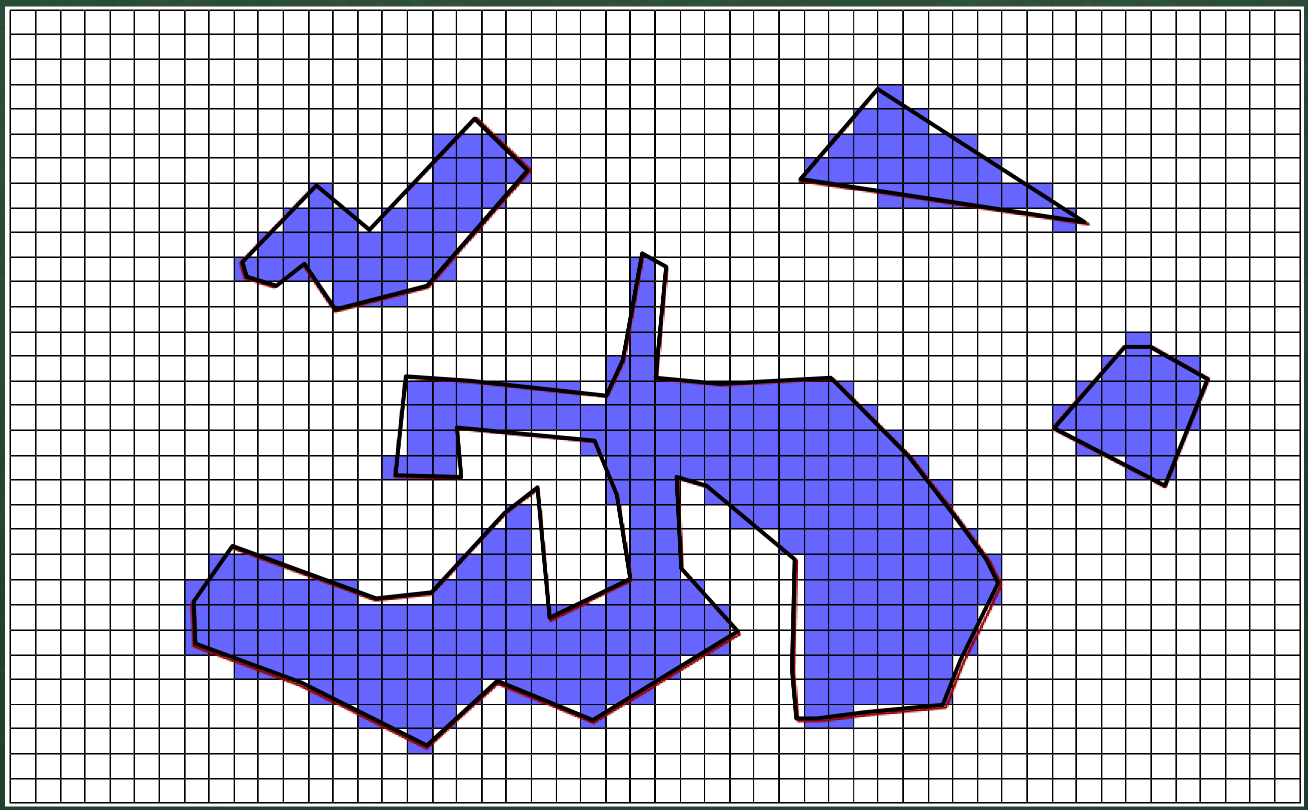
GVD Approximation – Method 2

- Approximation – Discretize Space
  - Convert the environment into a grid.
  - Compute the Voronoi diagram on resulting grid by propagating shortest paths from each obstacle point.
  - Remember which obstacle point the shortest path came from for each non-obstacle grid cell.
  - Can be slow to compute, depending on samples.



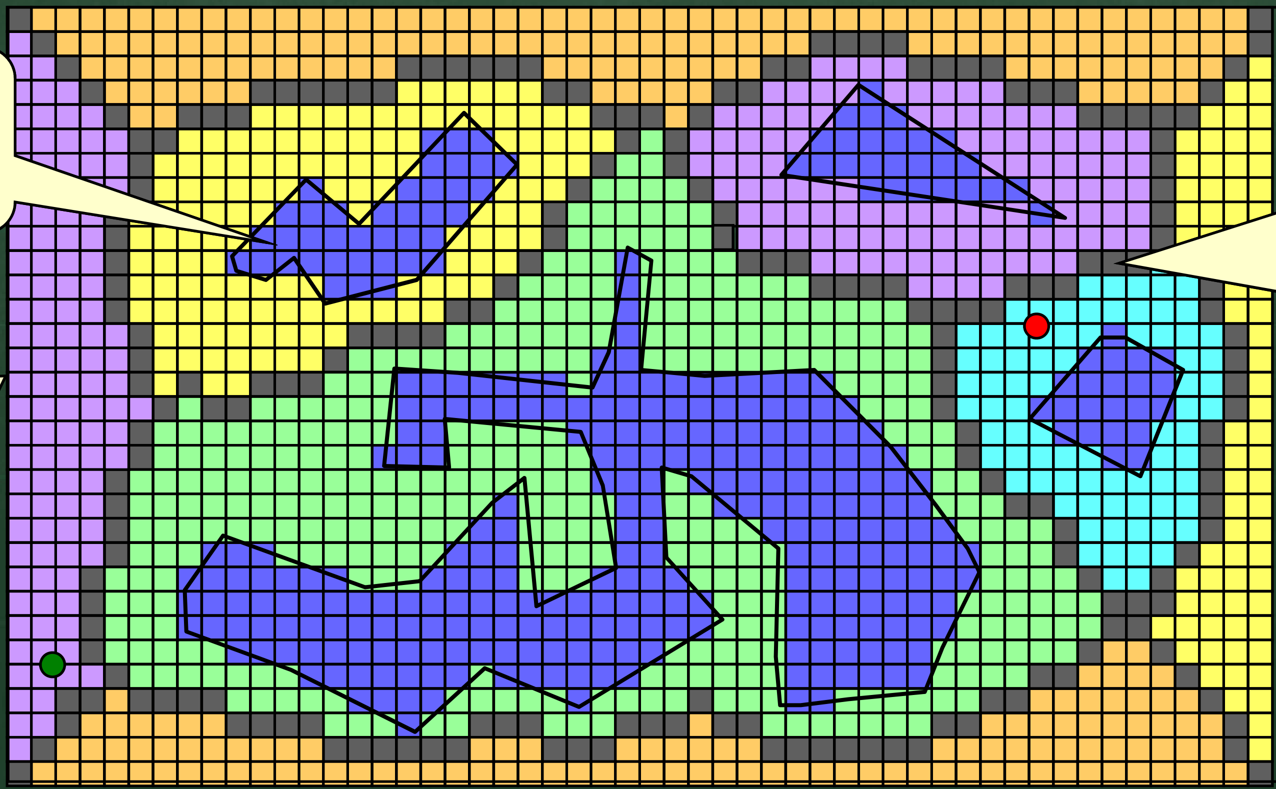
GVD Approximation – Method 2

- Approximation – Discretize Space (continued)
  - Create a grid from the environment



GVD Approximation – Method 2

- Approximation – Discretize Space (continued)
  - Compute the Voronoi diagram by running a grid shortest path, setting each obstacle cell as a source



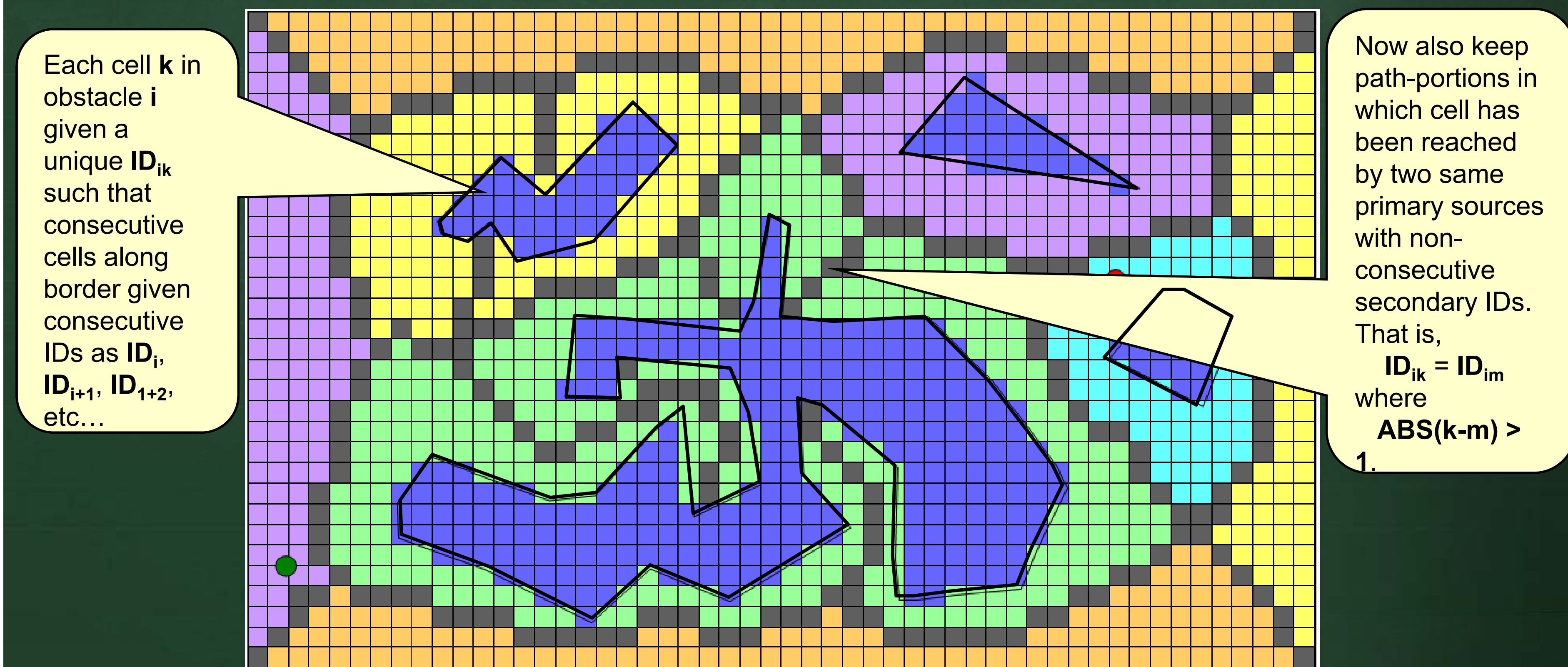
All cells in each obstacle **i** given unique **ID**$_i$.

Environment border included as obstacle.

Roadmap paths found when point is equally reachable by two different sources: That is, **ID**$_i$ **==** **ID**$_j$ where **i ≠ j**.

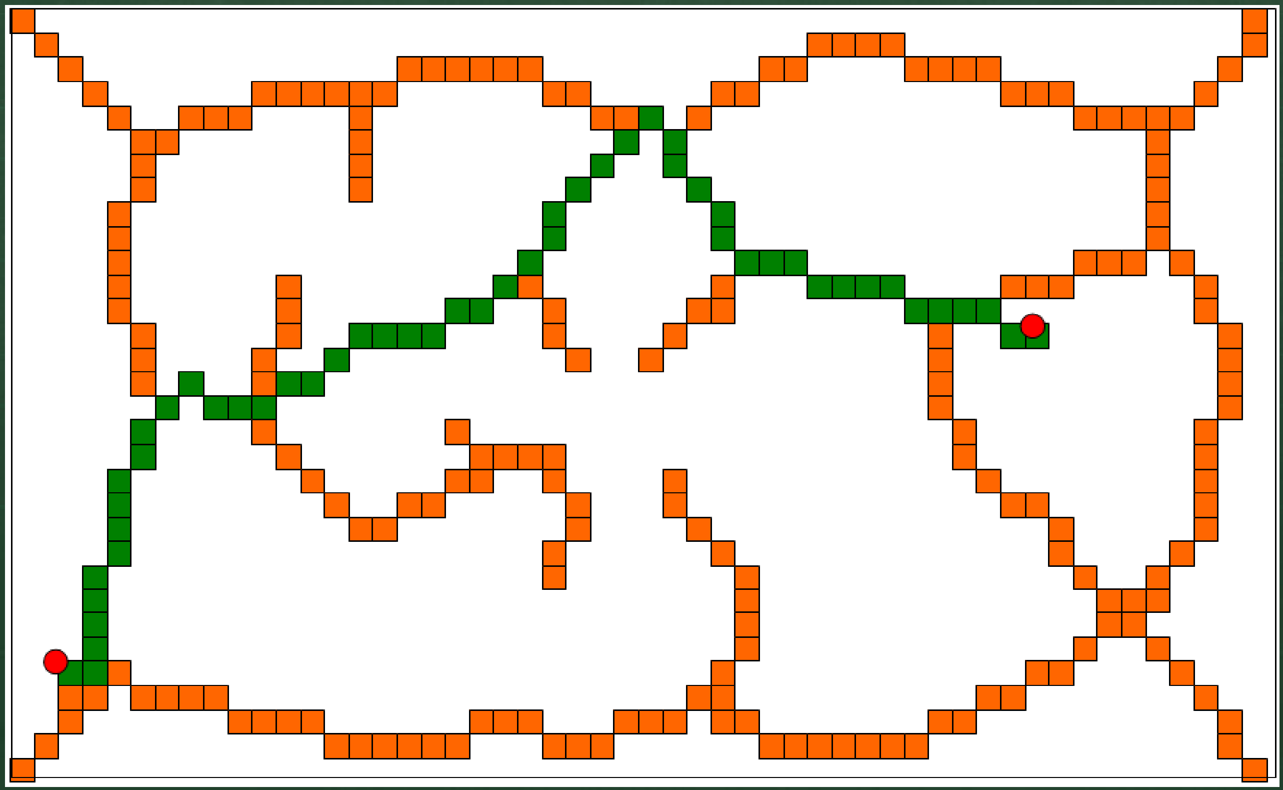
GVD Approximation – Method 2

- Approximation – Discretize Space (continued)
  - Use secondary-ID's to get path portions in between areas of non-convex obstacles.



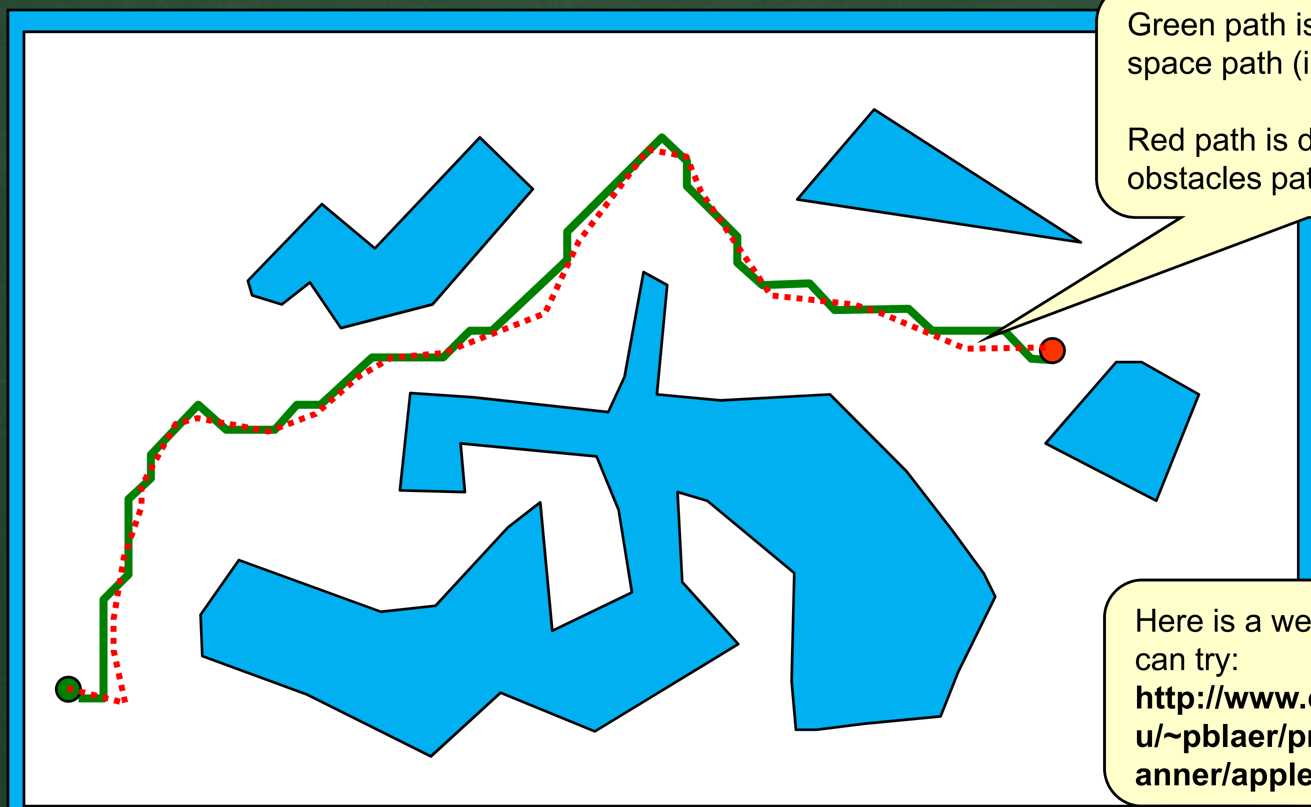
GVD Approximation – Method 2

- Approximation – Discretize Space (continued)
  - Compute a path in the Voronoi diagram



GVD Approximation – Method 2

- Approximation – Discretize Space (continued)
  - Resulting path is pretty good too:



Green path is discretized space path (i.e., grid).

Red path is discretized obstacles path (i.e., previous).

Here is a website that you can try: **http://www.cs.columbia.edu/~pblaer/projects/path_planner/applet.shtml**

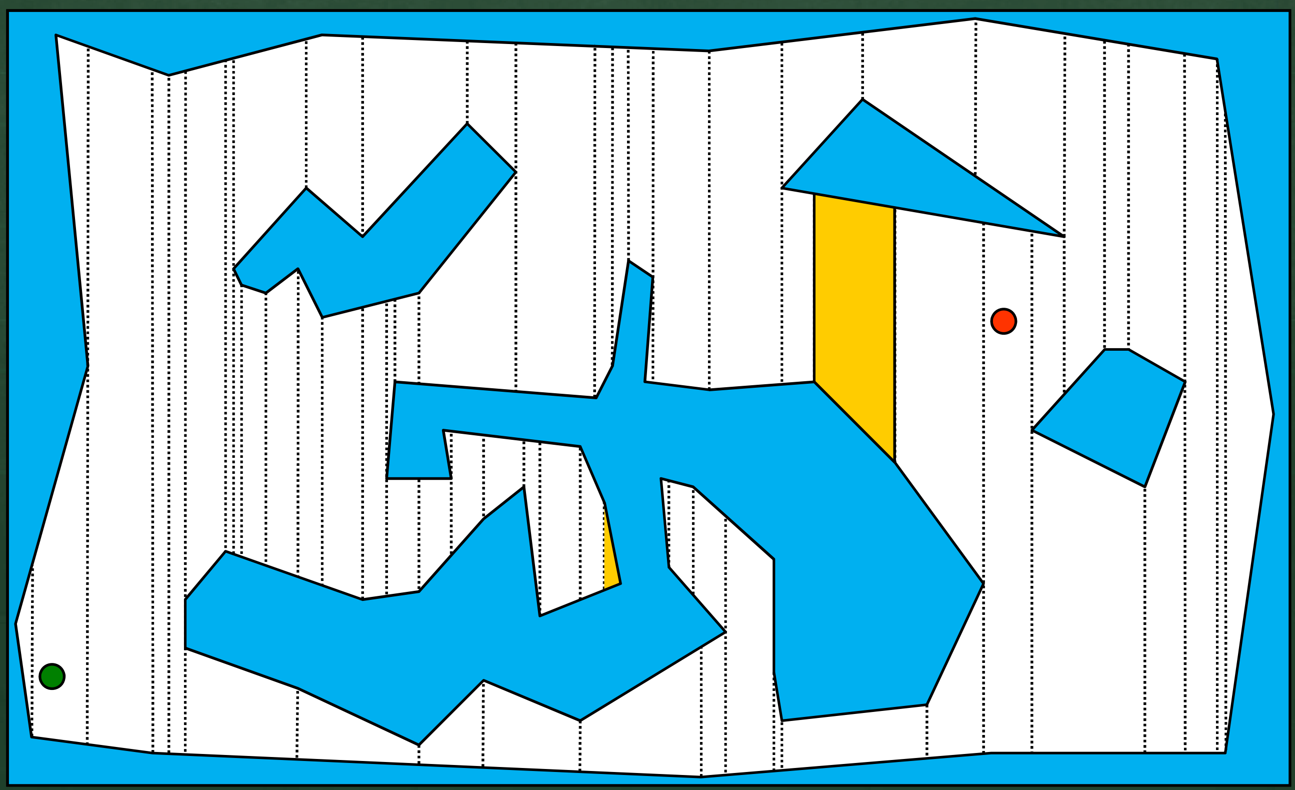
Cell Decomposition Paths

Cell Decomposition

- There are various ways to decompose (i.e., split up) the environment into cells.
  - We have already looked at grid-based methods, which are based on the same idea
- Now we will look at how to geometrically break up the environment into small-sized polygonal regions called ***cells***.
- We will then see how to determine a path through these cells.

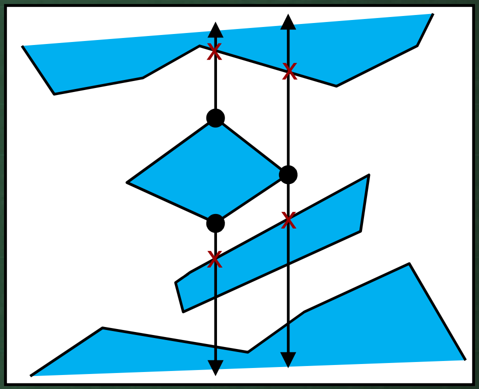
Trapezoidal Decomposition

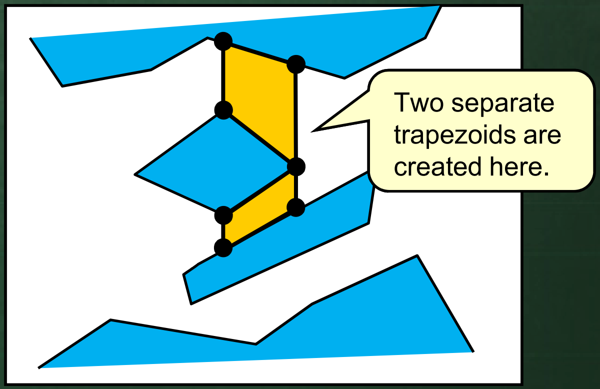
- Perhaps a simpler way to compute paths is to decompose the environment into simpler vertical cells in the form of trapezoids or triangles:



Trapezoidal Decomposition

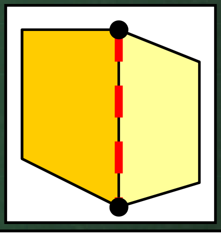
- How do we make the trapezoids ?
  - extend rays vertically directed up and down from each vertex of each obstacle and (including outer boundary).
  - when rays intersect obstacles (or boundary), the ray stops, becoming a trapezoid edge
  - need to compute intersections of each ray with all other obstacles.
    - can be done efficiently using a plane sweep technique, assuming vertices of all obstacles are sorted in x direction.



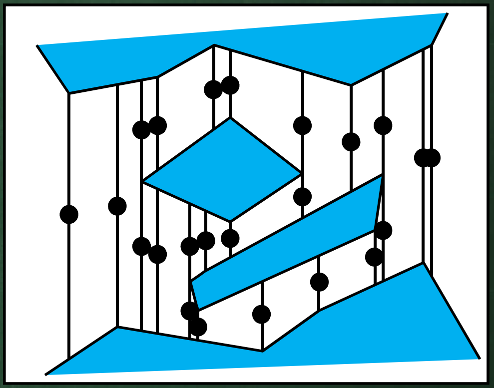


Trapezoidal Decomposition

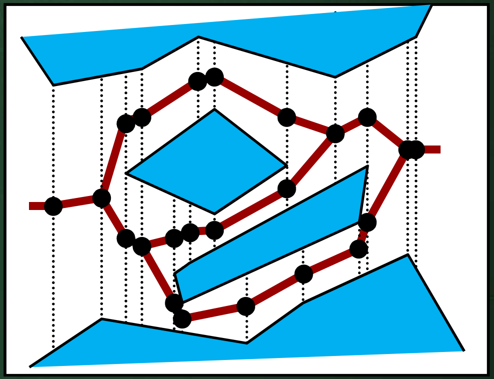
- While doing this, maintain which trapezoids are adjacent (i.e., beside) one another.



  - Adjacent trapezoids will share an edge with the exact same endpoints.

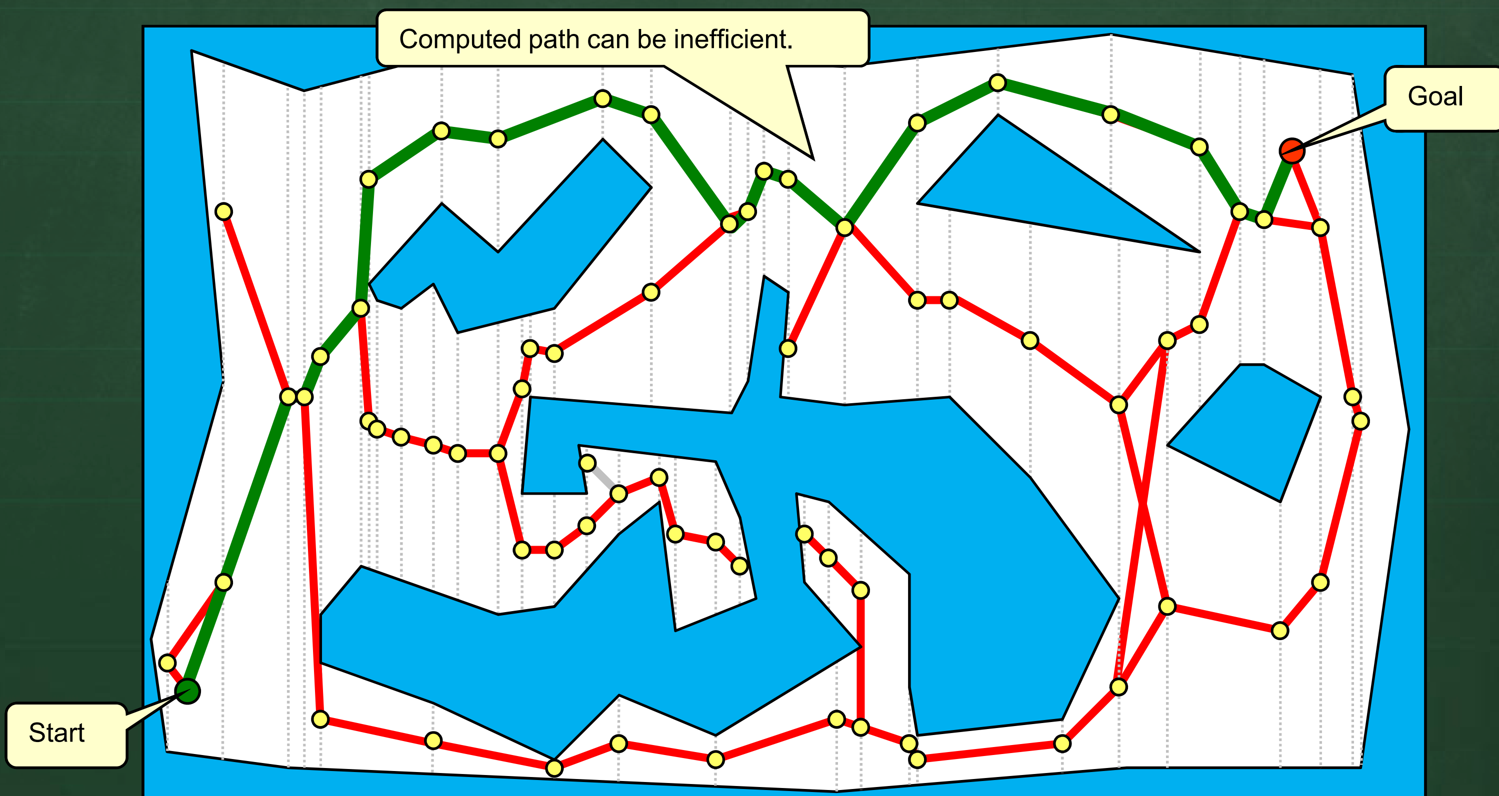


  - Determine midpoint of each trapezoid edge (except polygon/boundary edges).
  - Form a graph where
    - the nodes are the midpoints of the trapezoidal edges and two nodes are connected if they represent midpoints of edges belonging to the same trapezoid



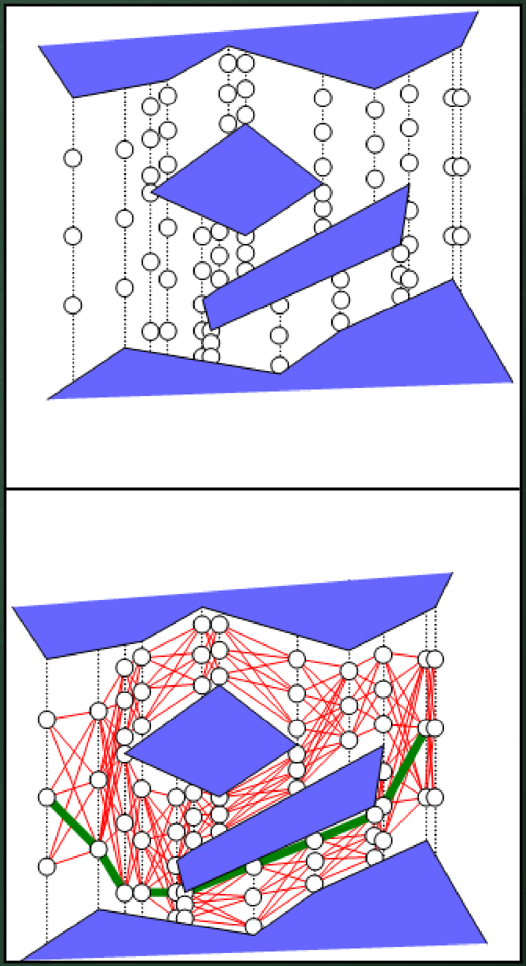
Computing a Path

- Easy to compute path now in the resulting graph:
  - Just determine which trapezoid contains start/goal and connect the start/goal to each node of that trapezoid.



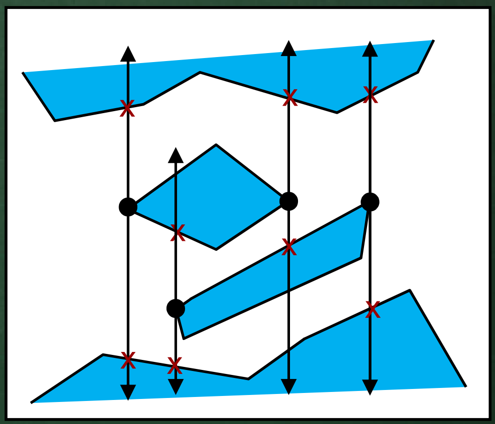
Improving the Path

- Can we make the computed path more efficient ?
  - Add more points (not just midpoint):
    - fixed number per edge, or
    - fixed distance between points
- As a result, the path:
  - may take different path around obstacles
  - will be more efficient
  - may travel closer to boundaries



Boustrophedon Cell Decomposition

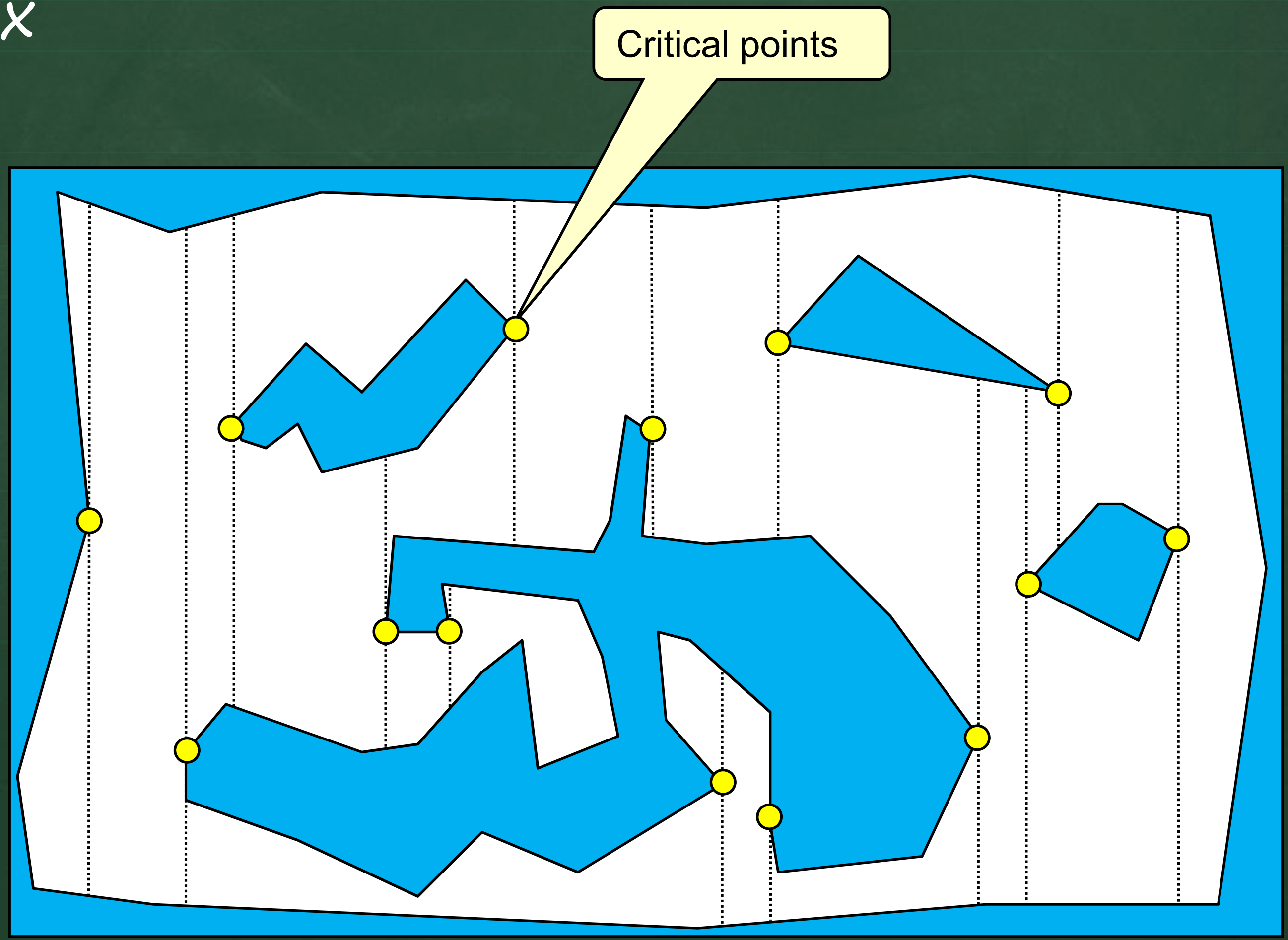
- Boustrophedon cell decomposition considers only critical points.
  - *critical points* are obstacle vertices from which a ray can be extended both upwards and downwards through free space.
  - Connect midpoints of formed line segments as with the trapezoidal decomposition technique.



- Cells, in general, are no longer trapezoids or triangles

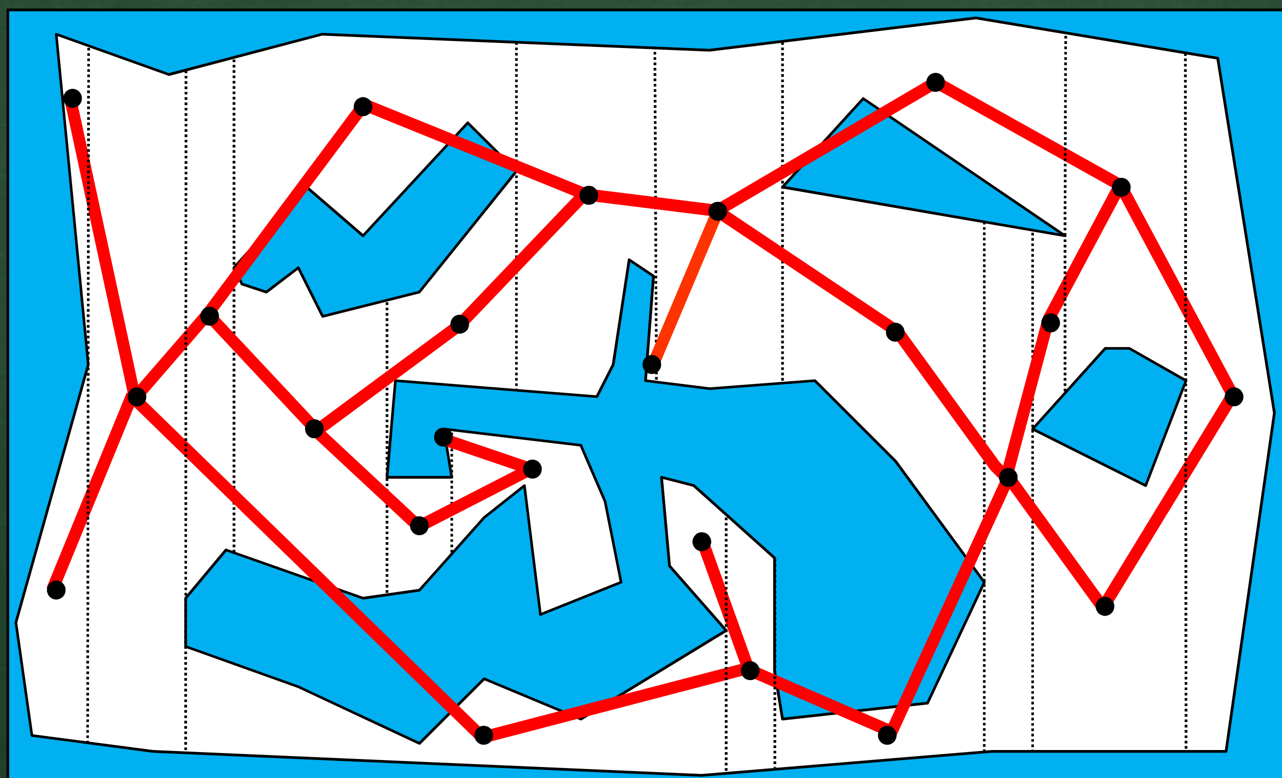
Boustrophedon Cell Decomposition

- Now less cells than trapezoidal, but cells are more complex



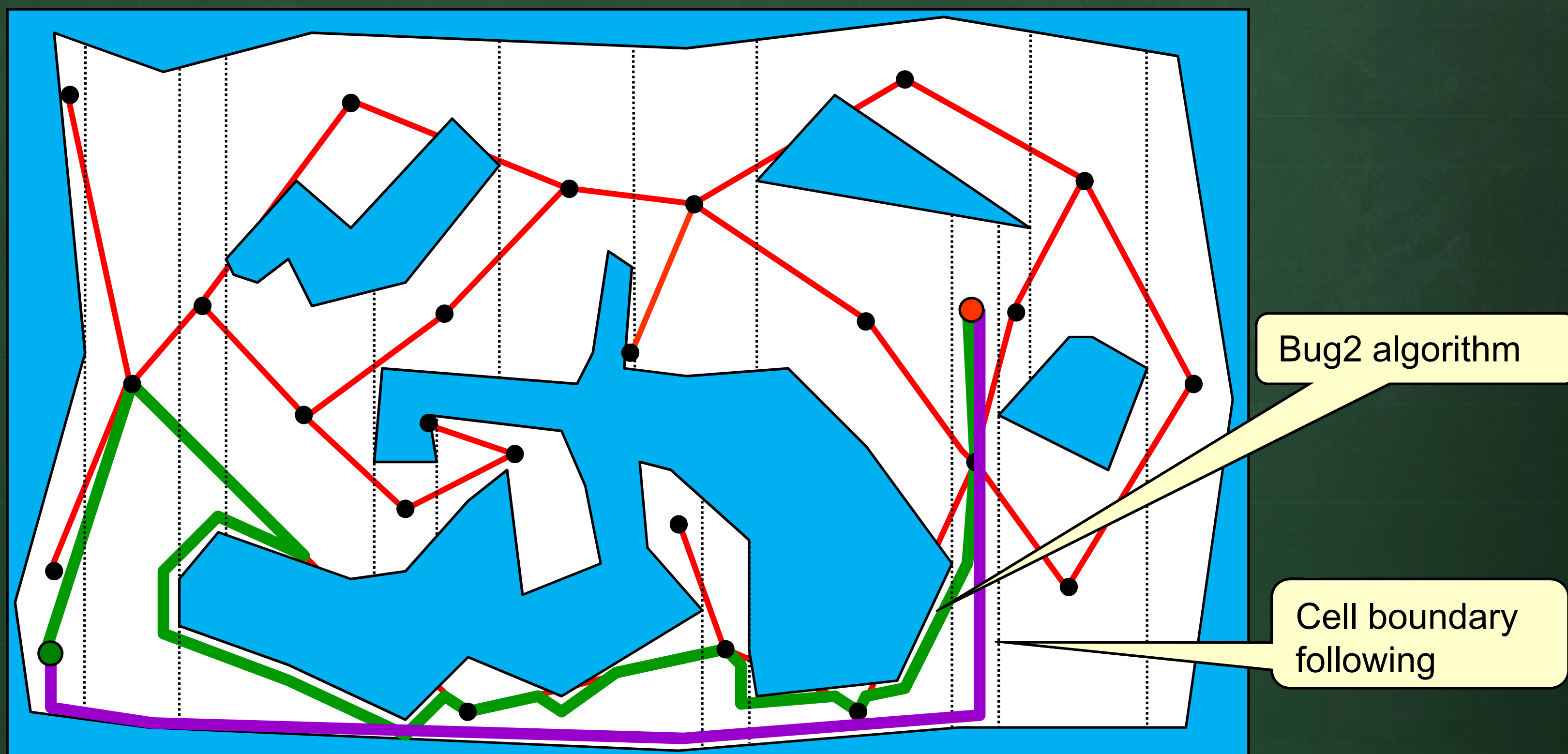
Boustrophedon Cell Decomposition

- Can interconnect cells, but connections are topological, not actual valid paths:



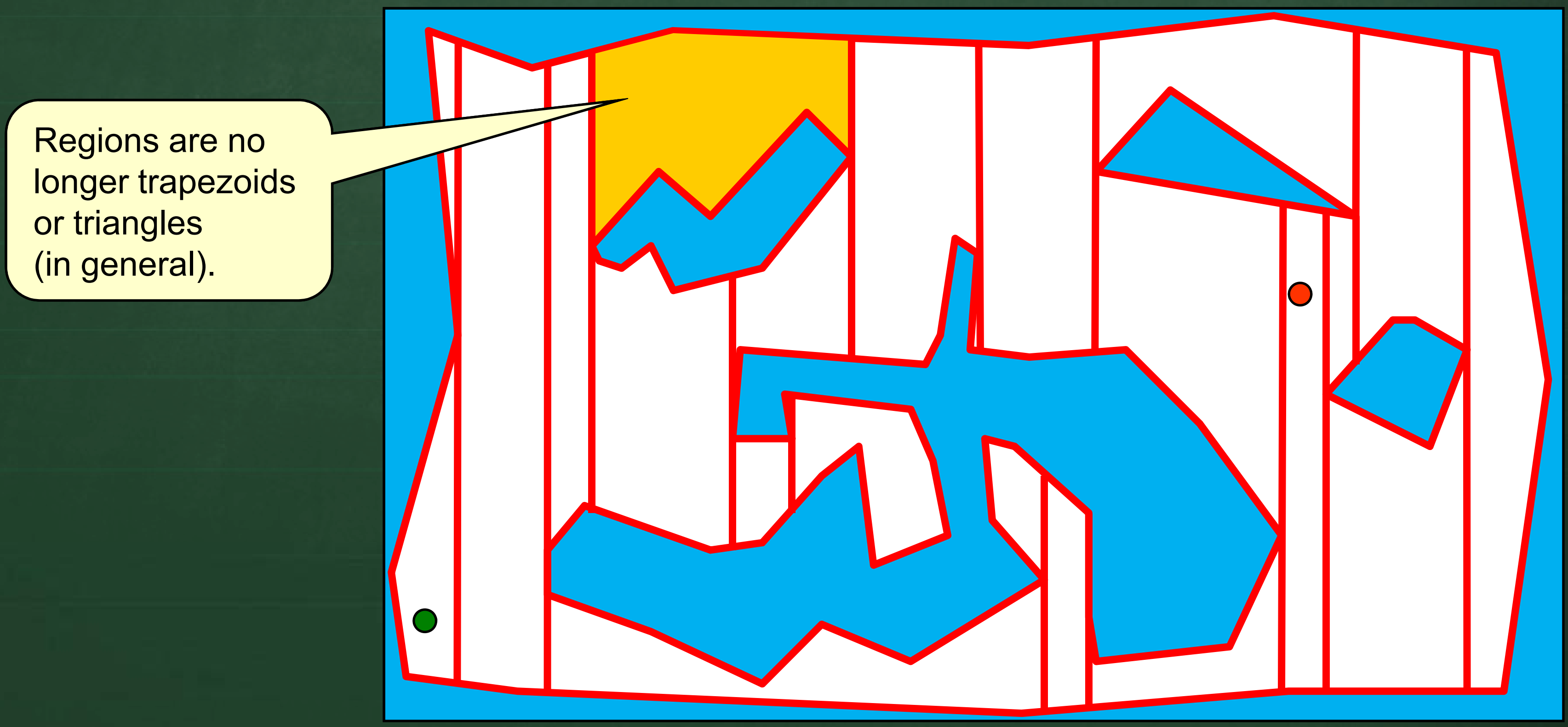
Boustrophedon Cell Decomposition

- To find a path now, we can use various strategies:
  - Bug algorithm, cell boundary following etc…



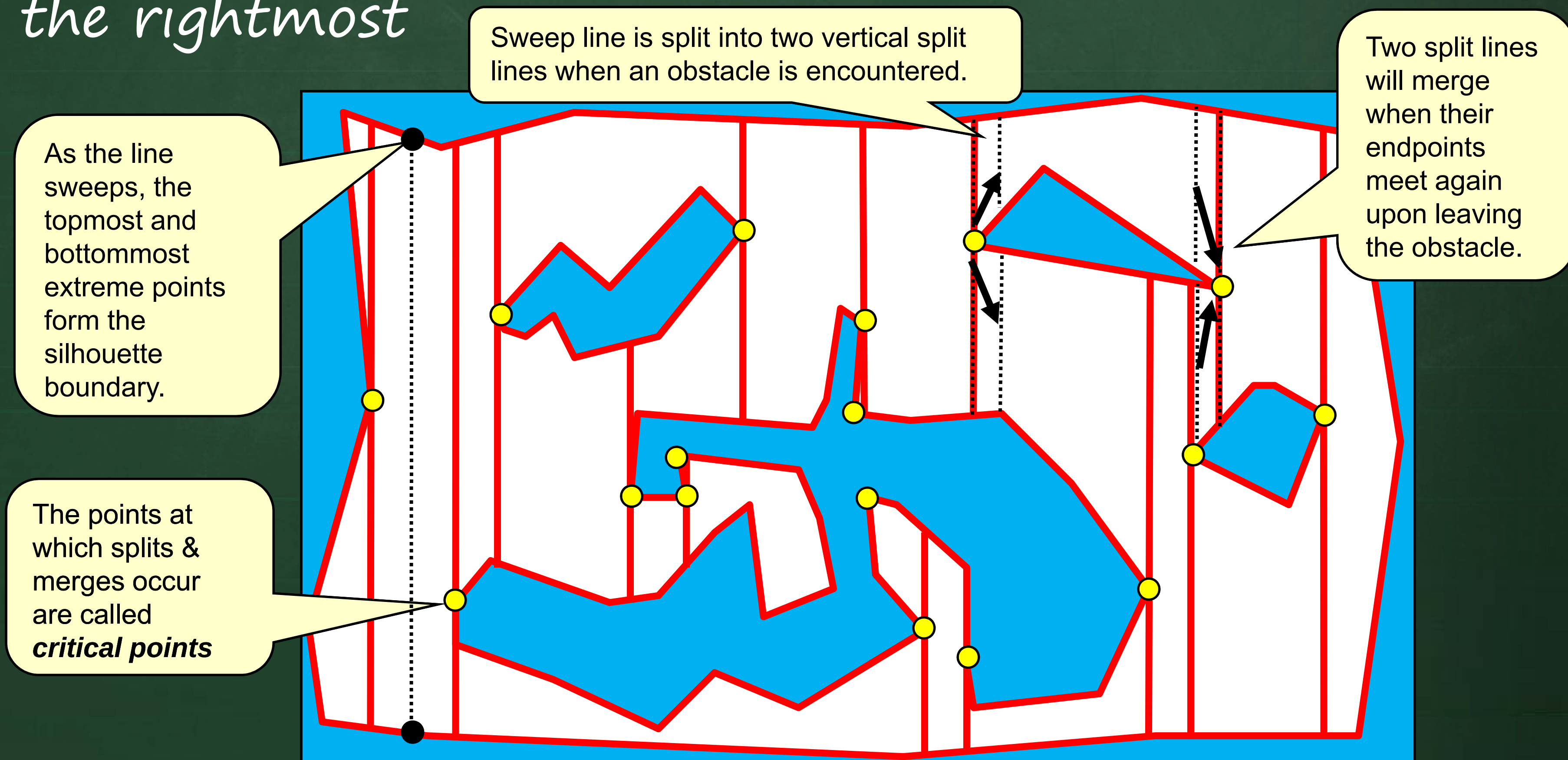
Canny's Silhouette Algorithm

- Another approach is to decompose the environment into silhouette curves which represent borders of the obstacles:



Canny's Silhouette Algorithm

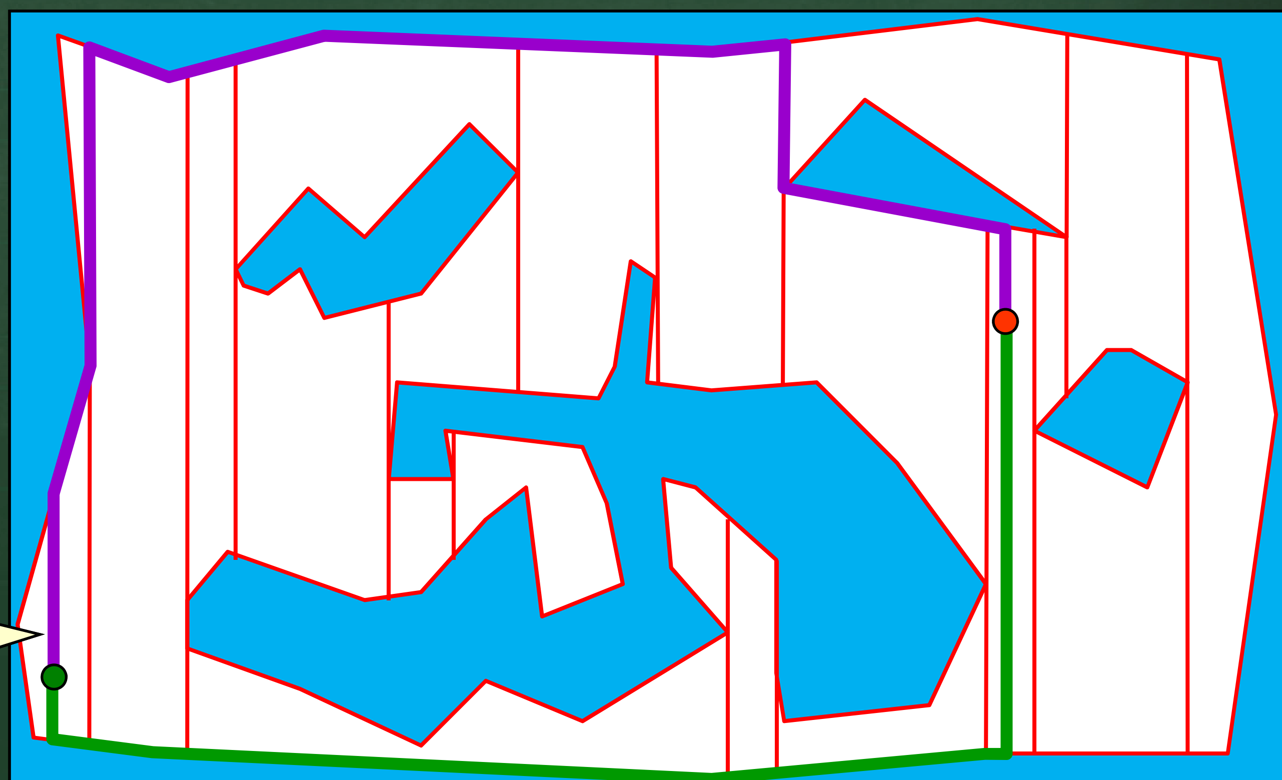
- To do this, consider a vertical line sweeping horizontally from the leftmost environment vertex to the rightmost



Canny's Silhouette Algorithm

- Compute a path by determining extreme points of vertical line passing through start/goal and then following silhouette path:

Here are two solutions … one going upwards, the other downwards.



Sampling-Based Road Maps

Sampling-Based Road Maps

- There are a few that we will look at based on:
  - Fixed Grid sampling
  - Probabilistic sampling
  - Random Tree expansion

- Such algorithms work by choosing fixed or random valid robot positions and then interconnecting them based on close proximity to form a graph of valid paths.

Grid-Based Sampling

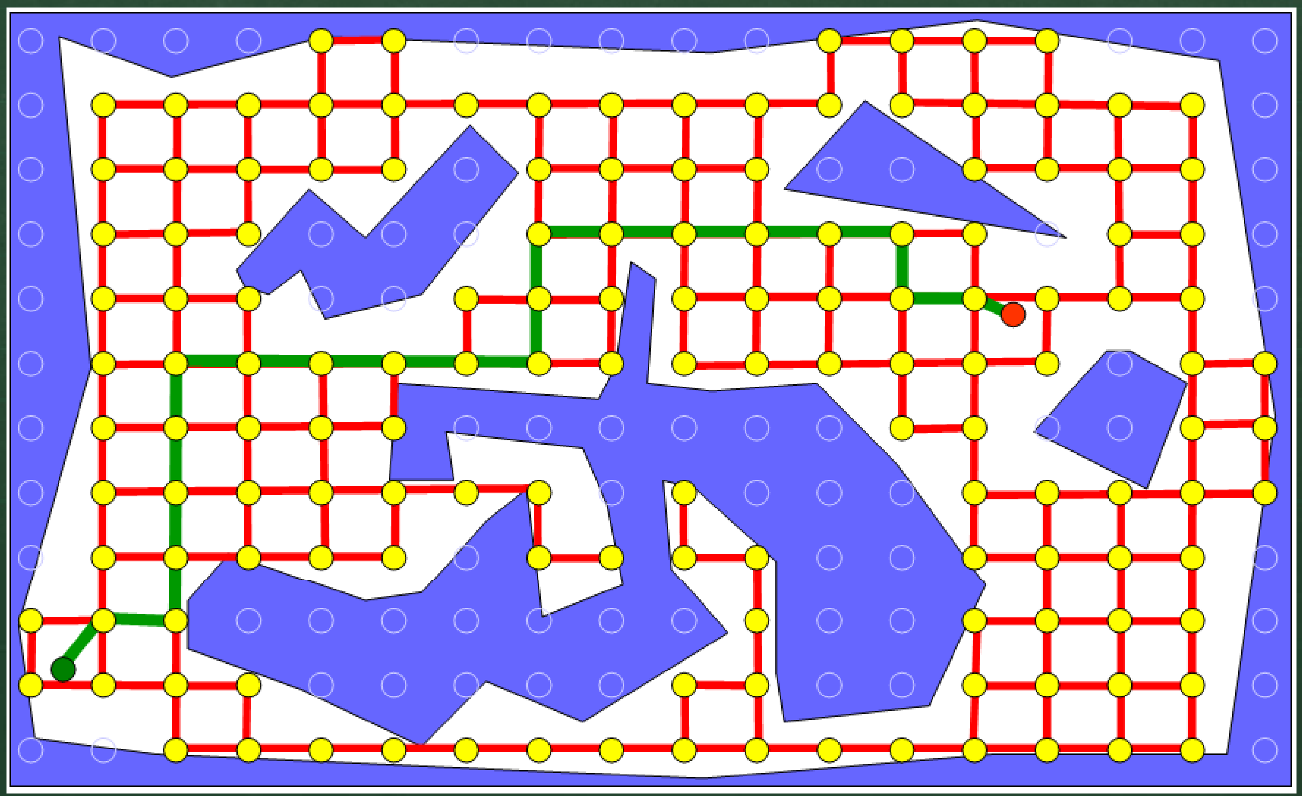
Grid-Based Sampling

- Grid-based sampling is perhaps the simplest technique based on overlaying a grid of vertices and connecting adjacent ones.
  - Accuracy and feasibility of resulting path depends on granularity of grid.

- We already looked at this strategy in terms of grid maps.
- Only interconnect vertices that do not intersect obstacle boundaries
- Multiple ways of interconnecting…

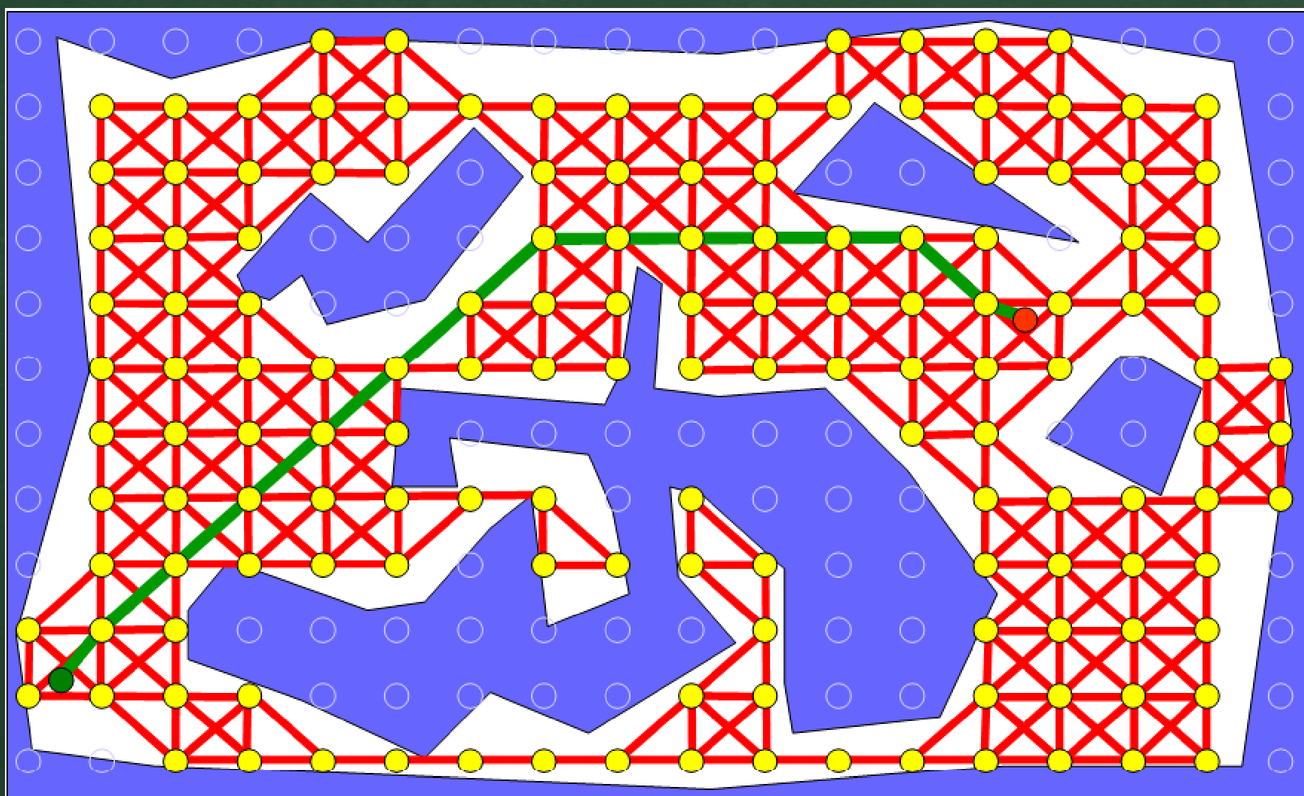
Grid-Based Sampling

- Here is a straight forward 4-connectivity grid.
  - Compute path from start to goal using graph search:

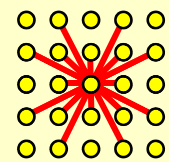


Grid-Based Sampling

- With additional "neighbor" connections, the graph allows more efficient paths…at a cost of increased space and slower computation time.

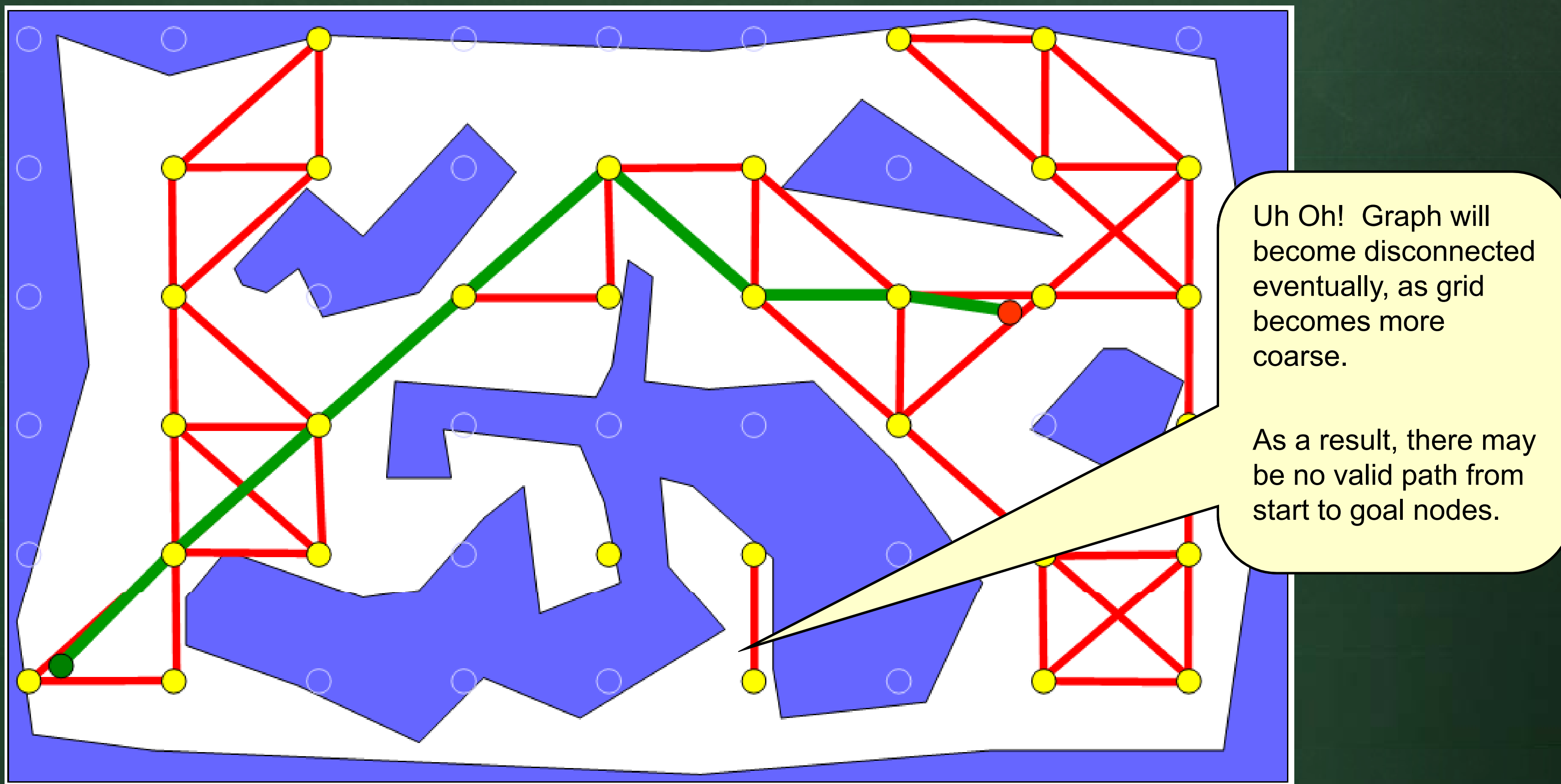


Can use a variety of neighbor interconnection strategies per node:



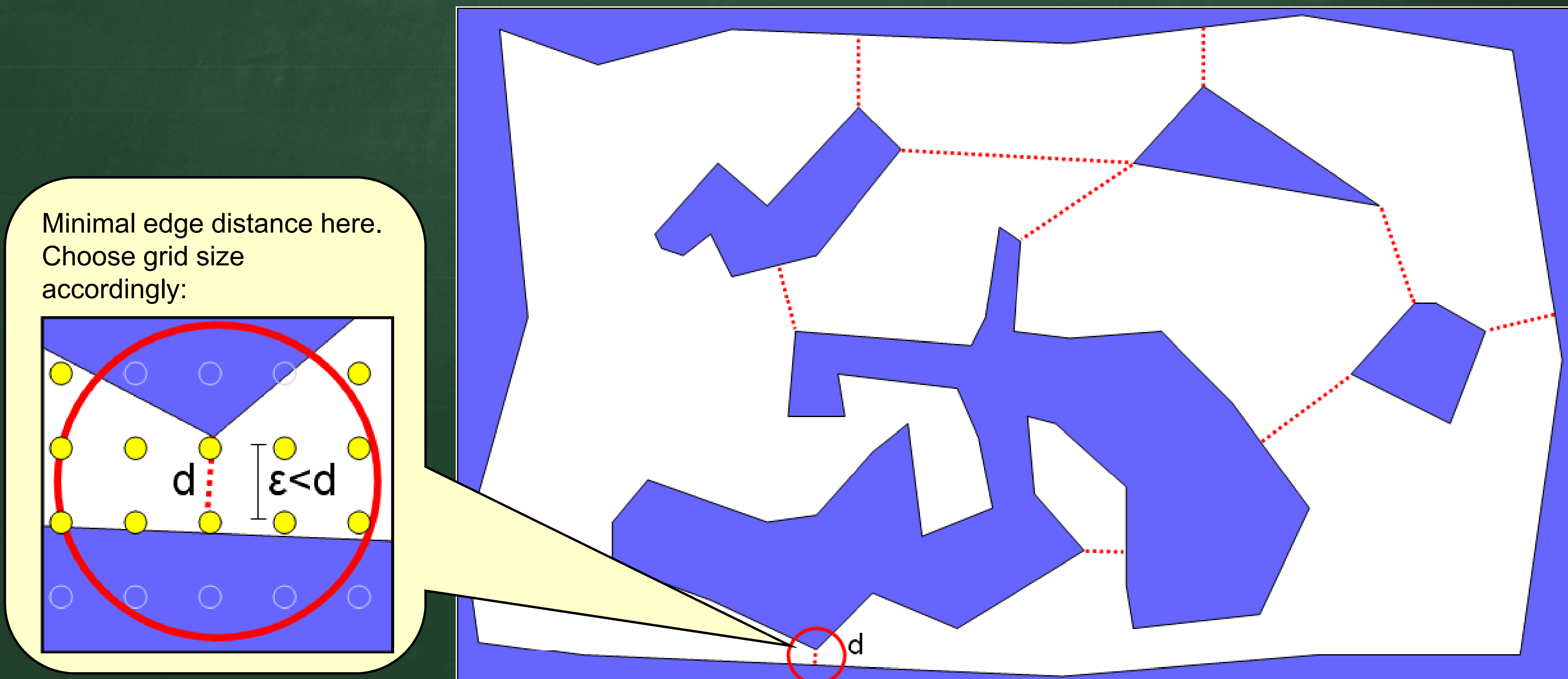
Grid-Based Sampling

- Here is the result with a reduced-size sample set (i.e., more coarse grid)



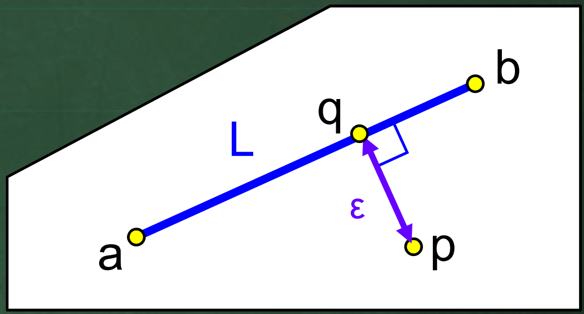
Setting the Grid Size

- We can ensure that a path exists:
  - choose grid size (i.e., width between connected nodes) to be smaller than minimum distance between any two obstacle edges that do not share a vertex:



Setting the Grid Size

- How do we determine the shortest distance between two line segments L_1 and L_2 ?
- Consider first the distance from a point p to a line L:



  - p will intersect line L at a right angle, say at point q
  - let u be the distance of q along L from a to b

$$u = \frac{(x_p - x_a)(x_b - x_a) + (y_p - y_a)(y_b - y_a)}{(x_b - x_a)^2 + (y_b - y_a)^2}$$

  - the coordinates of q are:

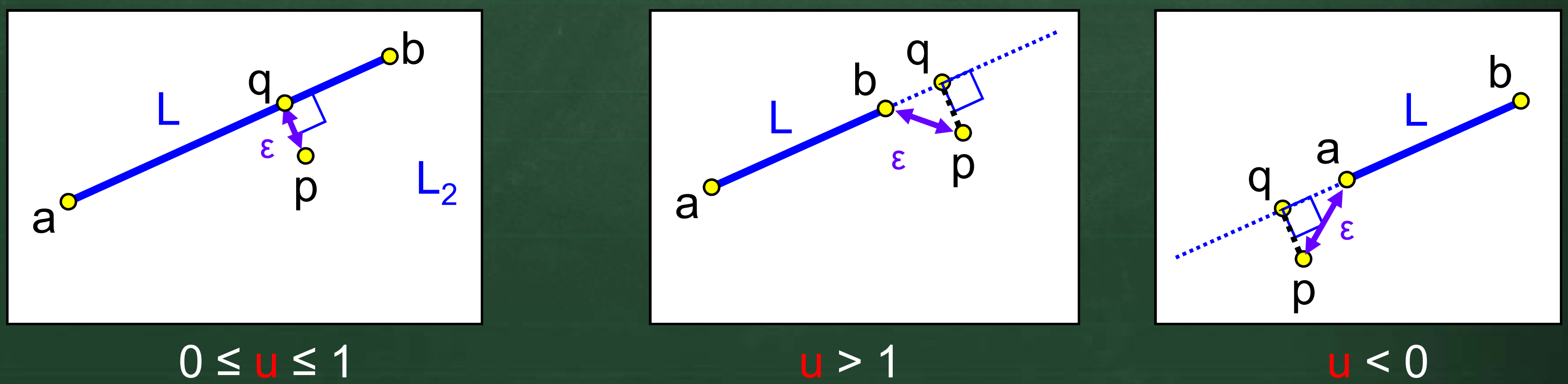
$x_q = x_a + u(x_b - x_a)$ and $y_q = y_a + u(y_b - y_a)$

ε is then just the distance between **p** and **q**.

Setting the Grid Size

- We then need to determine whether or not $q=(x_q,y_q)$ lies on the segment $L = ab$.

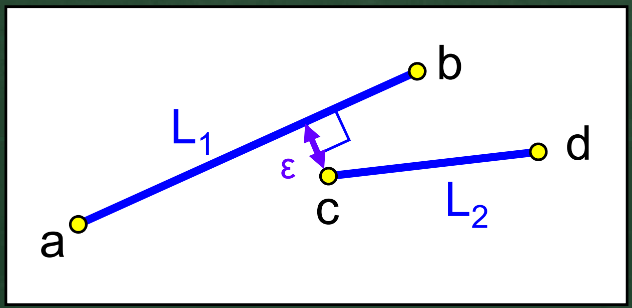
- If $0 \leq u \leq 1$ then q lies on the segment and therefore $\varepsilon = |\overline{pq}|$ else $\varepsilon = \min(|\overline{pa}|, |\overline{pb}|)$



Setting the Grid Size

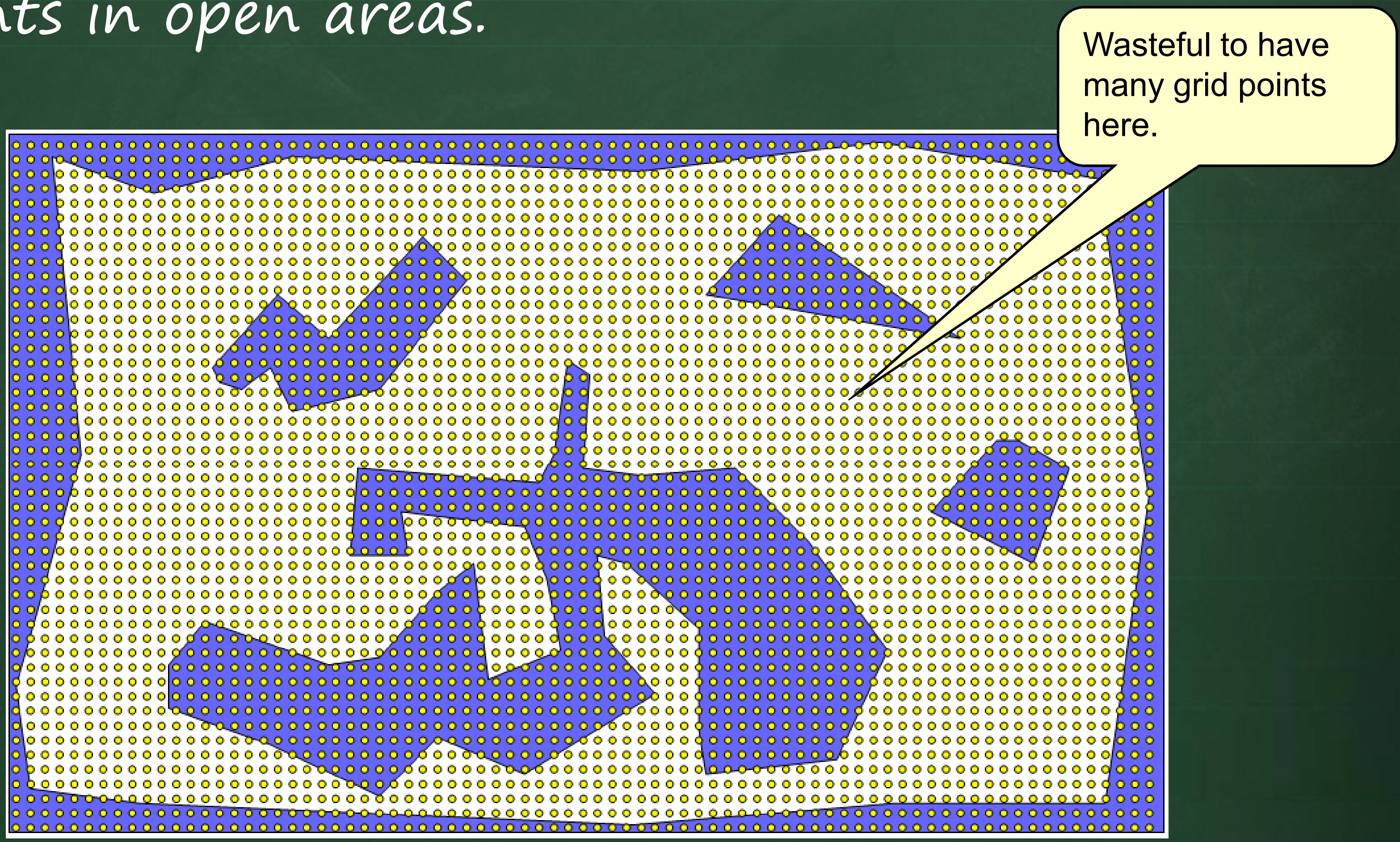
- Let $\varepsilon = \delta(p, L)$ be the shortest distance function from a point p to a segment L.
- We can use this to find the minimum distance between two segments L_1 and L_2 as:

$$\varepsilon = Min(\delta(a, L_2), \delta(b, L_2), \delta(c, L_1), \delta(d, L_1))$$



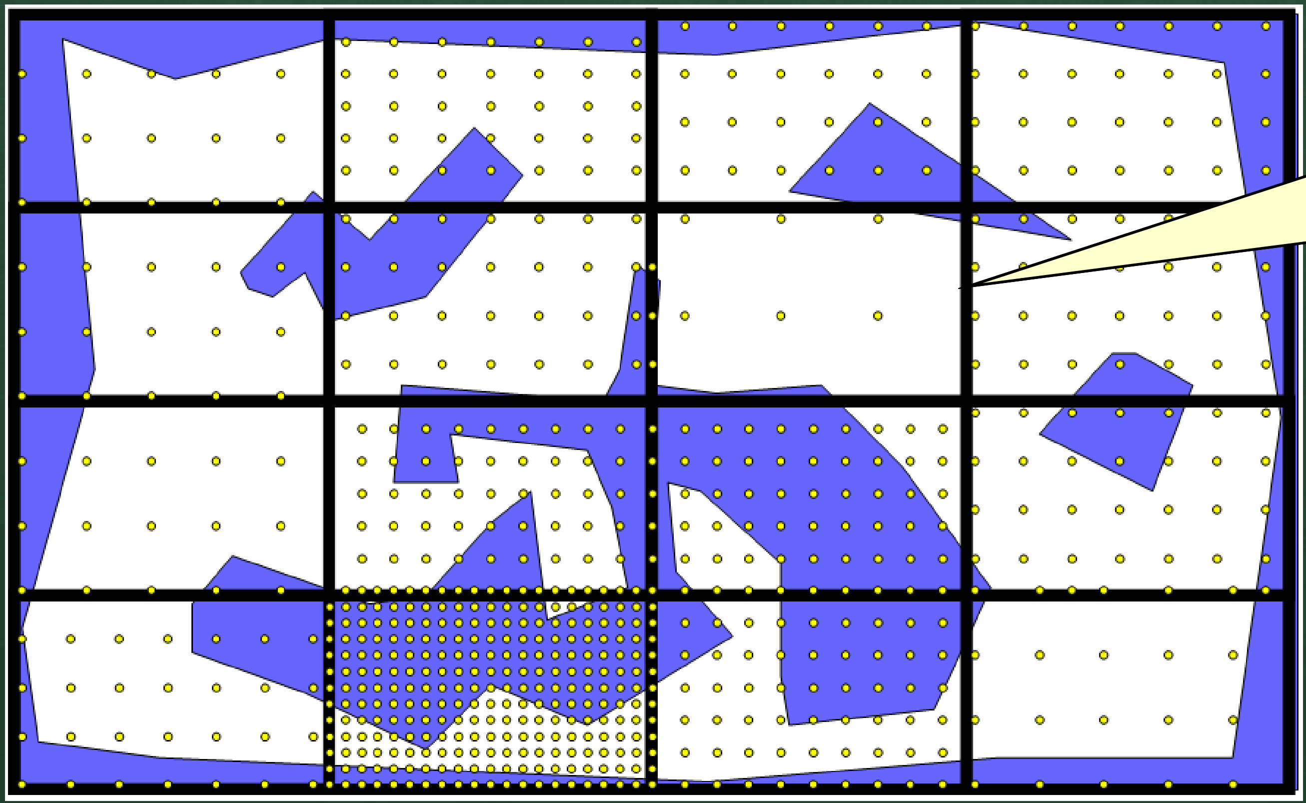
Grid-Based Sampling

- The main problem here is it causes too many grid points in open areas.



Grid-Based Sampling

- Can always do a quad-tree-like decomposition, determining the smallest gaps within certain areas, recursively.

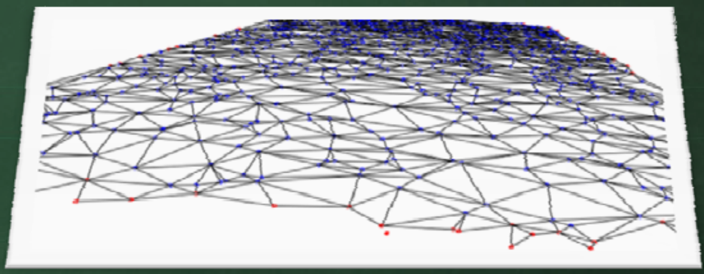


Details have been left out as to how to connect at borders.

Probabilistic Road Maps

Probabilistic Road Maps

- *Probabilistic Road Maps* (PRM) are sampling-based mapping strategies.



  - They are created by selecting random points (i.e., samples) from the environment and interconnecting points that represent valid short path lengths.
  - They perform fairly well, but are best for situations in which robot configurations are more complex than a single point robot.
  - Solution depends on how many nodes are used and how much interconnectivity there is between nodes.

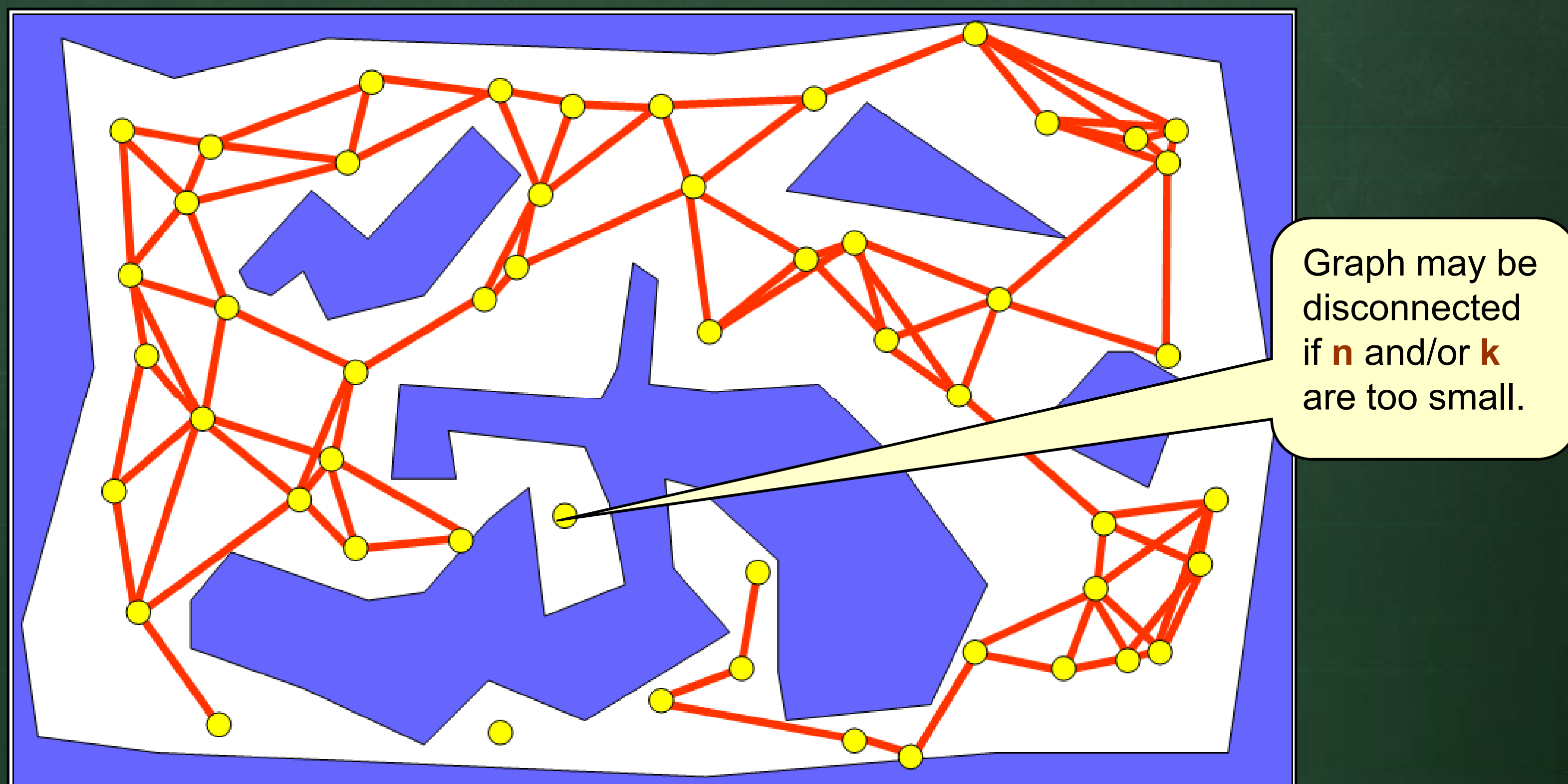
Probabilistic Road Maps

- Algorithm produces a graph $G=(V,E)$ as follows:

```
LET V and E be empty.
REPEAT
   Let v be a random robot configuration (i.e., random point)
   IF (v is a valid configuration) THEN       // i.e., does not intersect obstacles
        add v to V
UNTIL V has n vertices
FOR (each vertex v of V) DO
   Let C be the k closest neighbors of v      // i.e., the k closest vertices to v
   FOR (each neighbor ci in C) DO
        IF (E does not have edge from v to ci) AND (path from v to ci is valid)
   THEN
            Add an edge from v to ci in E
   ENDFOR
ENDFOR
```

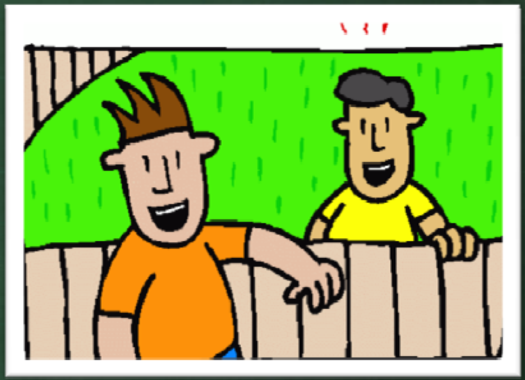
Probabilistic Road Maps

- Here is an example of randomly added nodes and their interconnections (roughly, **n** = 52 and **k** = 4):



Probabilistic Road Maps

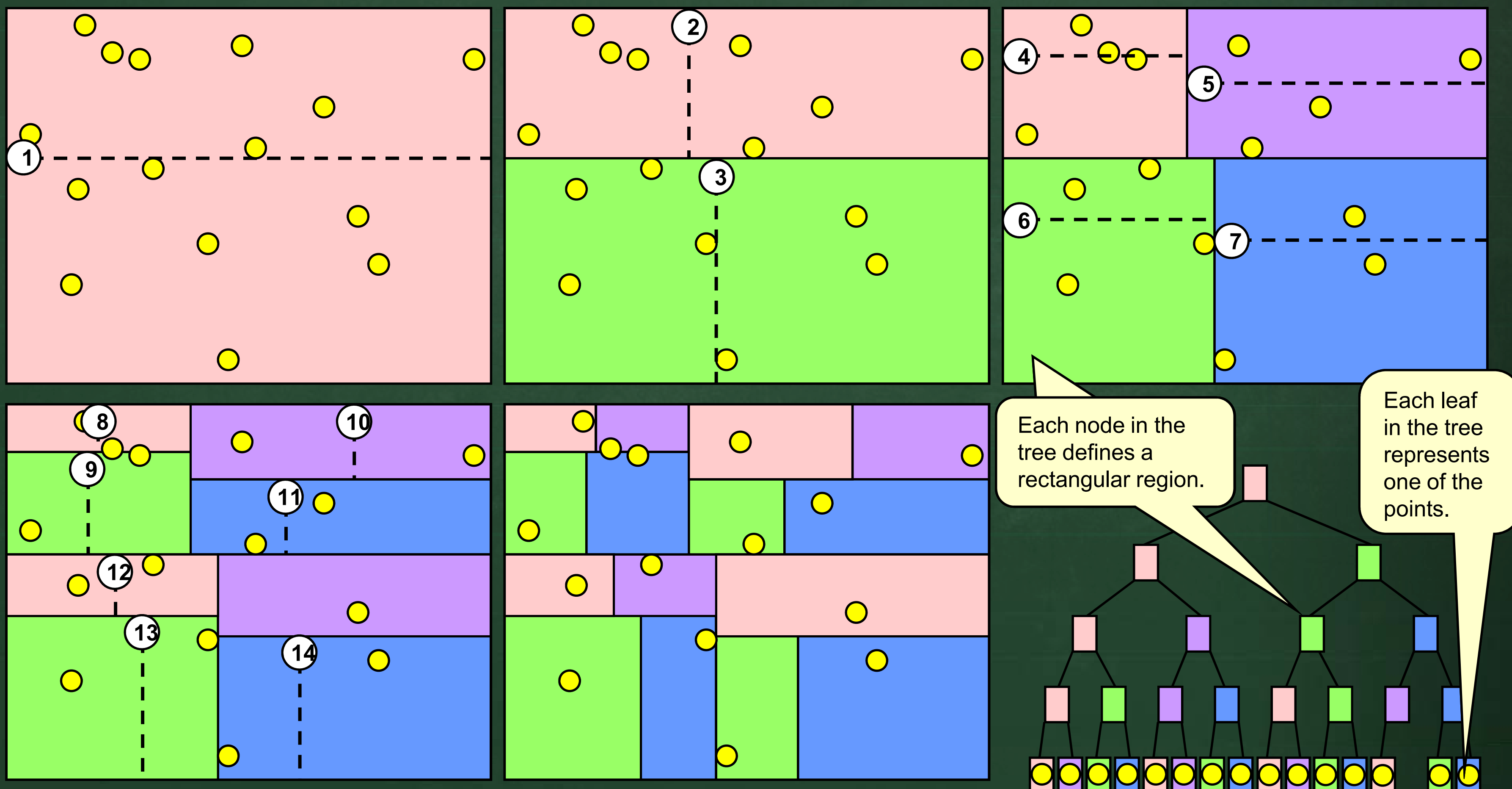
- How do we find the k-nearest neighbors ?
- Multiple strategies:
  - "Brute Force" (check everything $O(n^2 \log n)$)
  - KD-Trees — Most popular
  - R-Trees
  - VP-Trees



- The KD tree is the most popular since it is relatively straight forward to implement.
- Basically, divides recursively the sets of points in half…alternating with vertical/horizontal cuts.

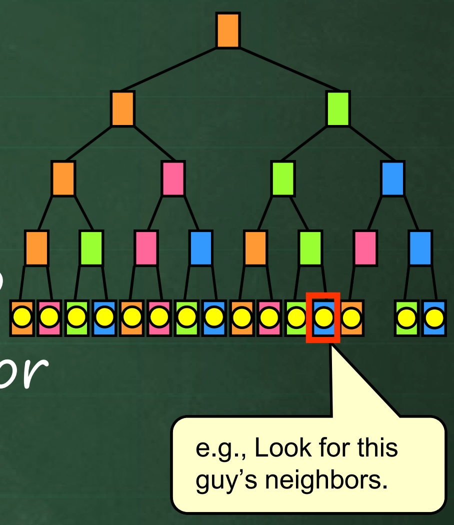
K-D Trees

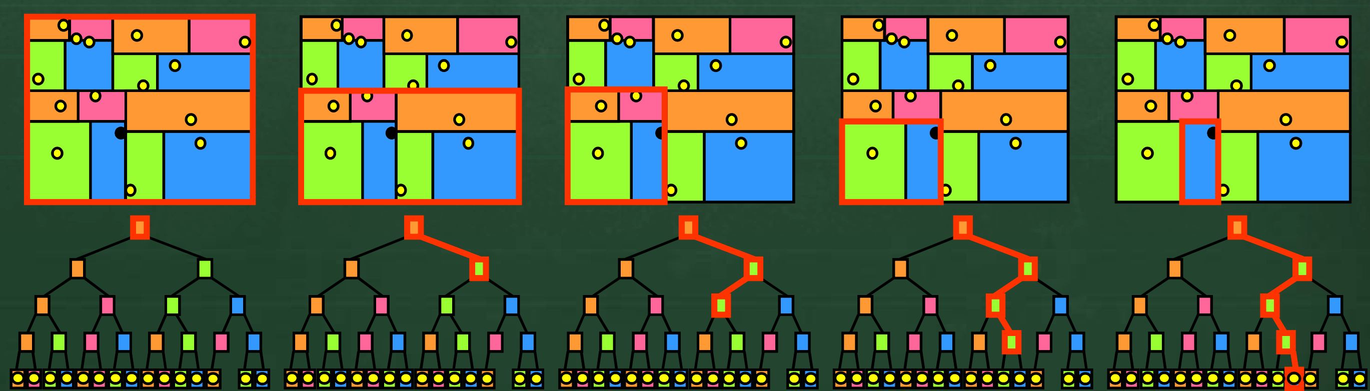
- Here is how to a KD-Tree is constructed:



K-D Trees

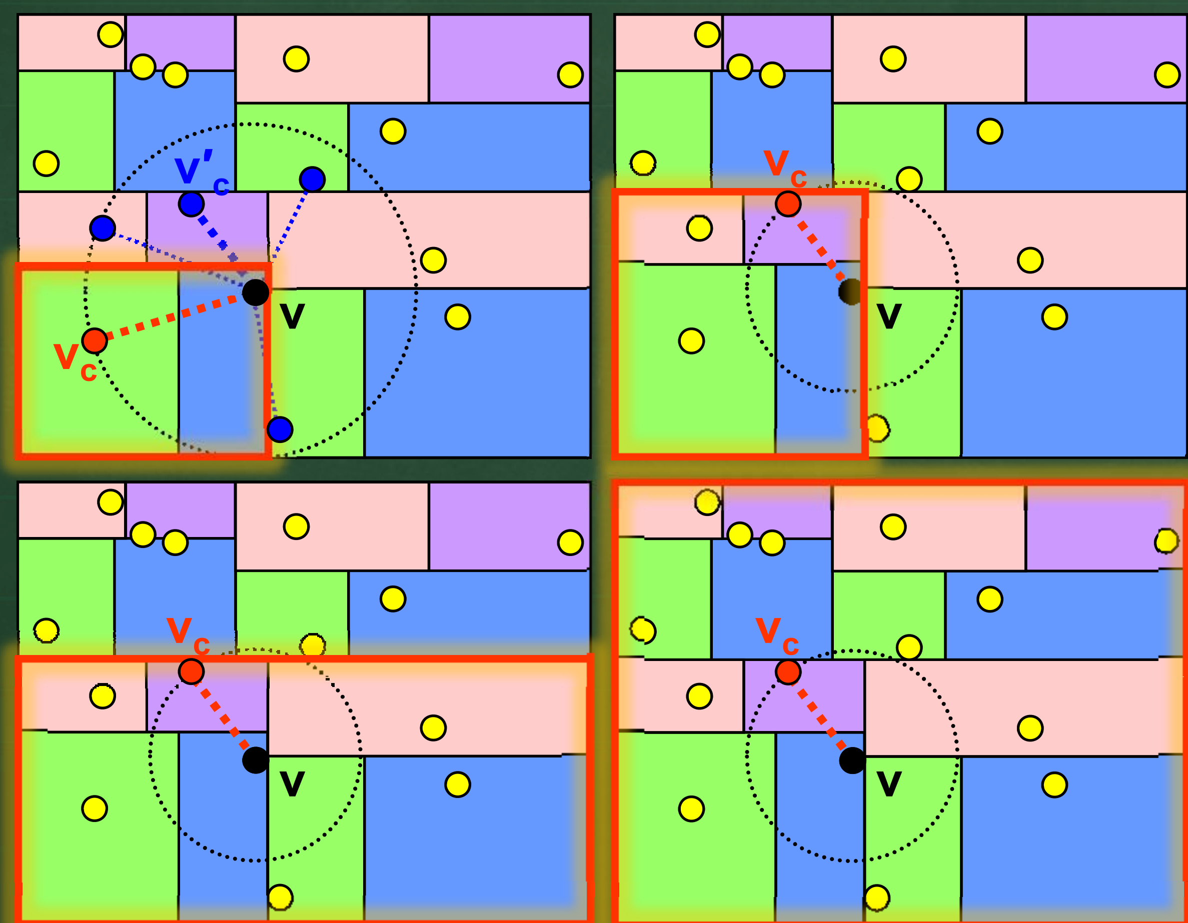
- Once constructed, we find the k-nearest neighbors of a leaf.
  - Start by recursively searching down the tree to find the rectangle that contains the vertex v (for which we are trying to find its neighbors)





K-D Trees

- Compute closest neighbor on way back from recursion:



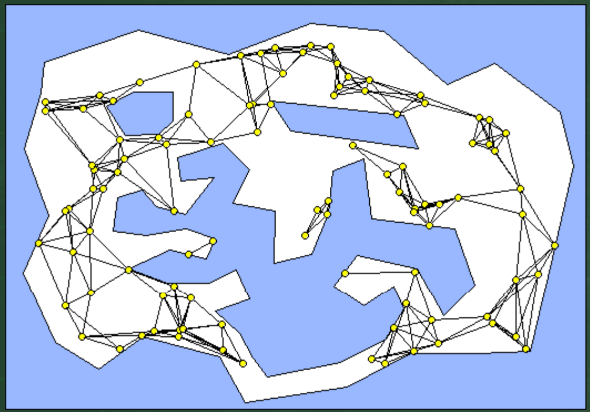
We can find the **k** neareast neighbors as follows:

1. Let closest neighbor $\mathbf{v_c}$ be the point in the first window on way back from recursion.
2. Compute a circle with radius $\mathbf{vv_c}$.
3. Check vertices in all rectangles that intersect the circle for a better neighbor.
4. If a better neighbor $\mathbf{v'_c}$ is found, shrink the circle to a smaller radius defined by $\mathbf{vv'_c}$.
5. Continue in this way until the root is reached.

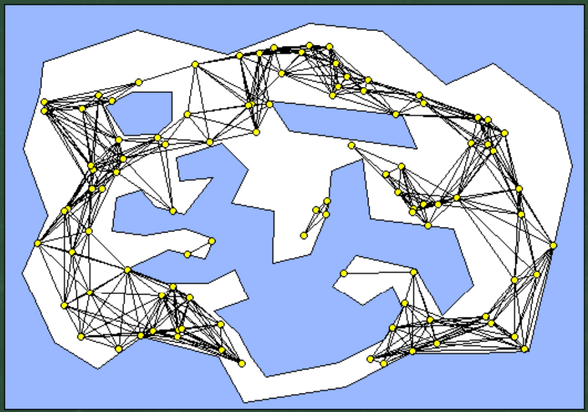
Repeat the above procedure **k** times…making sure to flag the closest neighbor each time so that it is not found again.

Probabilistic Road Maps

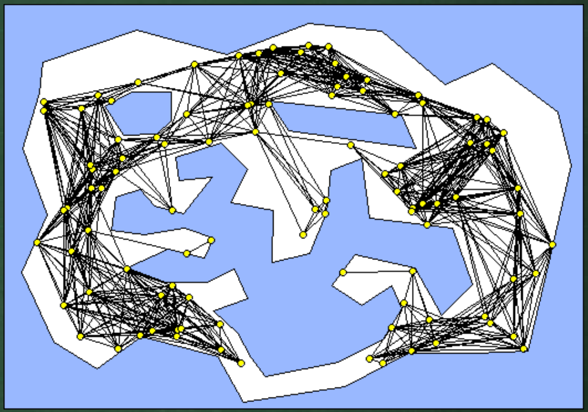
- Here are some maps for various **n** and **k** values:



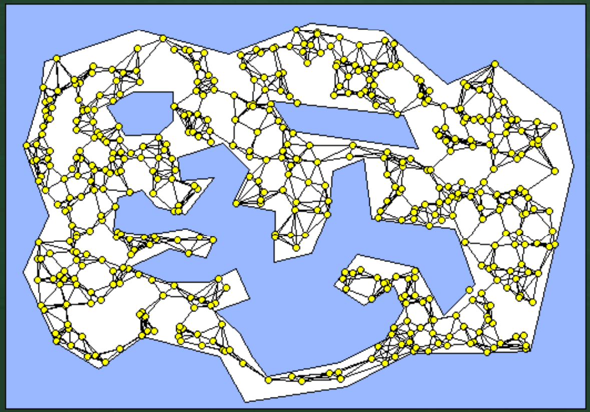
n=100, k=5



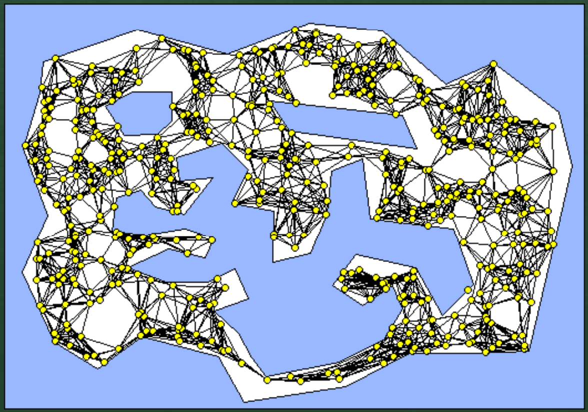
n=100, k=10



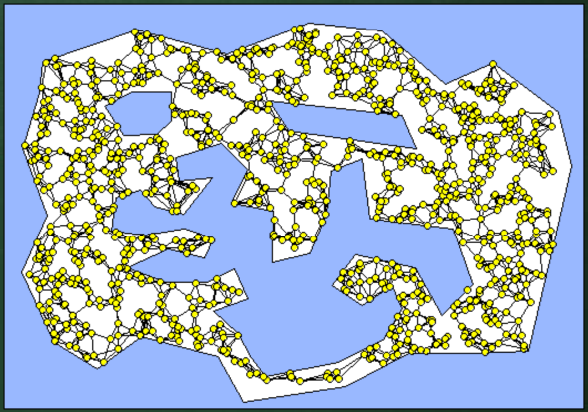
n=100, k=20



n=500, k=5



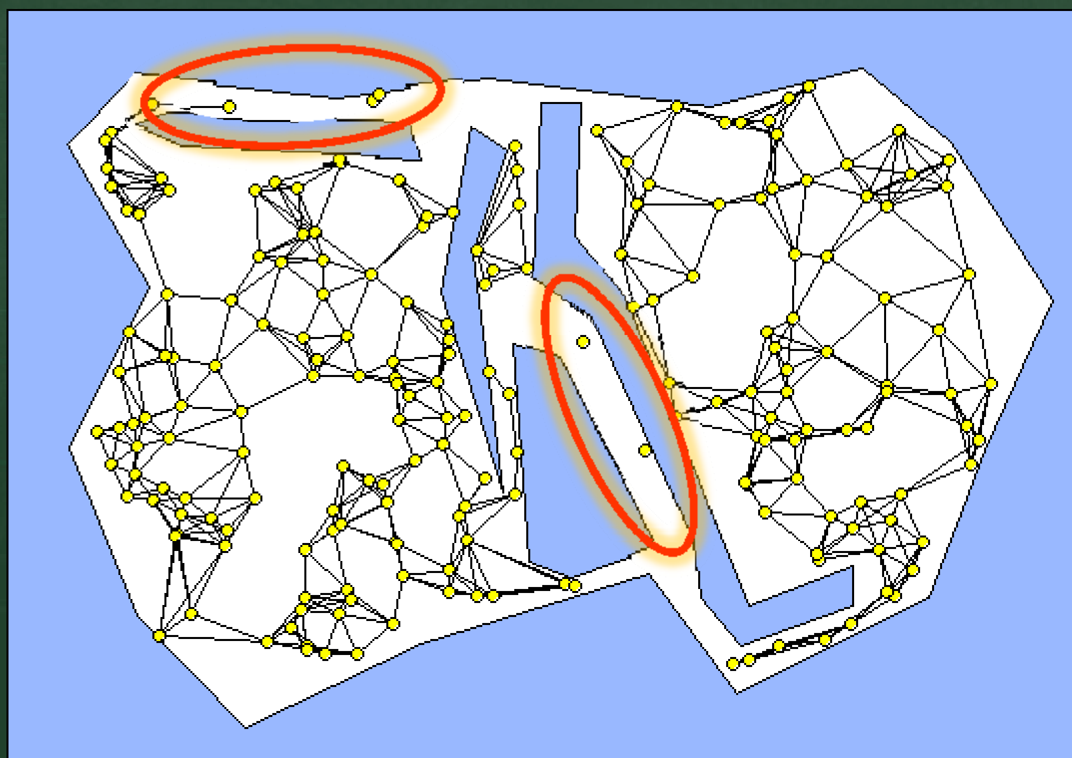
n=500, k=10



n=1000, k=5

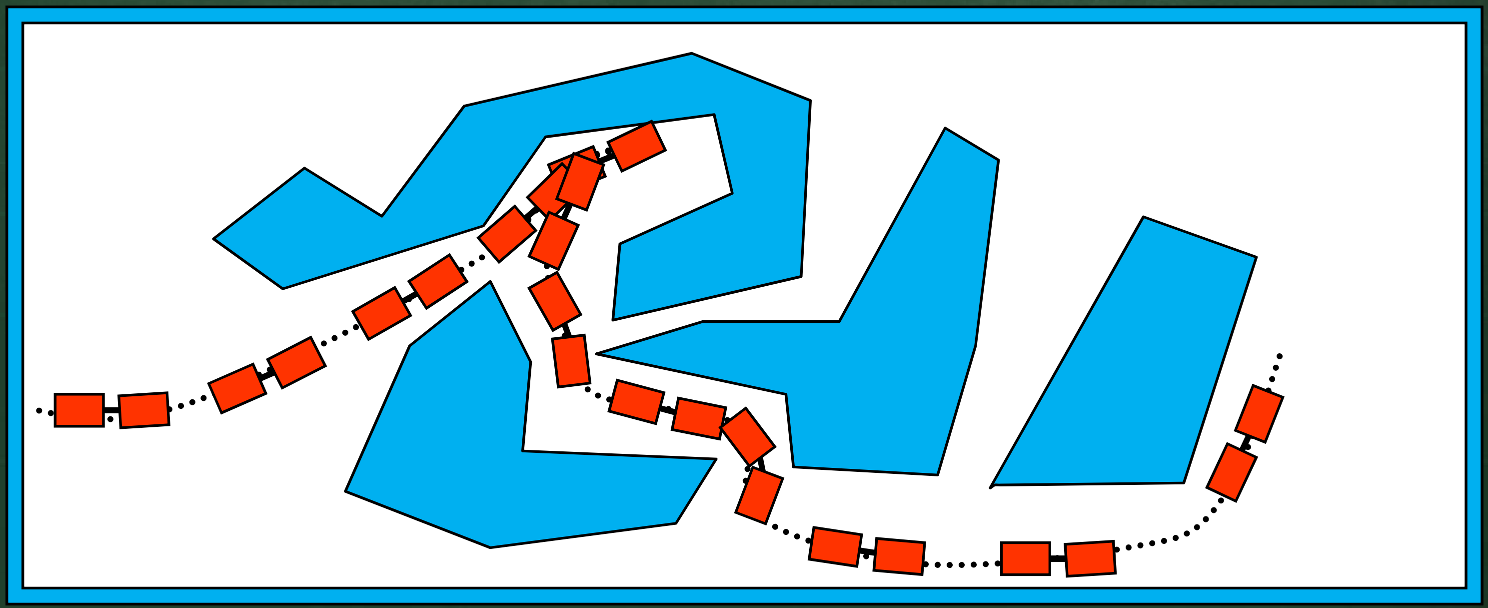
Probabilistic Road Maps

- PRMs perform well in practice, but are susceptible to missing vertices in narrow passages
  - Could lead to disconnected graphs and no solution:



Probabilistic Road Maps

- PRMs perform well when the robot configurations are more complex
  - when robots are not just points, but different shapes in different positions.
  - performs very well for robot arm kinematics



Rapidly-Exploring Random Tree Maps

Rapidly Exploring Random Trees

- *Rapidly Exploring Random Trees* (RRTs):
  - each node represents random robot configuration (i.e., point representing valid robot location in environment).
  - single query planner which covers the space between the start/goal locations quickly
  - root starts at the current robot position.
  - grows outwards from the start either completely randomly or somehow biased towards the goal location.
  - input parameters are the number of nodes to be used in the tree and the length (i.e., step size) of edges to add.

RRT Algorithm

- The algorithm produces a tree G=(V,E) as follows:

LET V contain the start vertex and E be empty.

REPEAT

  LET q be a random valid robot configuration (i.e., random point)

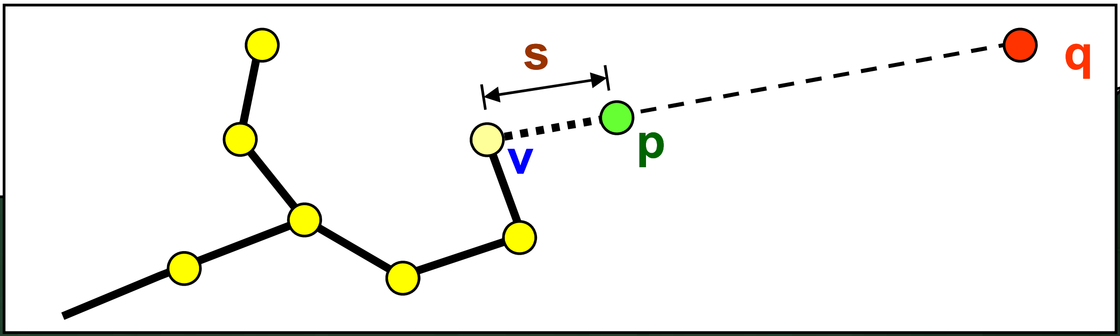
  LET v be the node of V that is closest to q

  LET p be the point along the ray from v to q that is at distance s from v.

  IF (vp is a valid edge) **THEN** *// i.e., does not intersect obstacles*

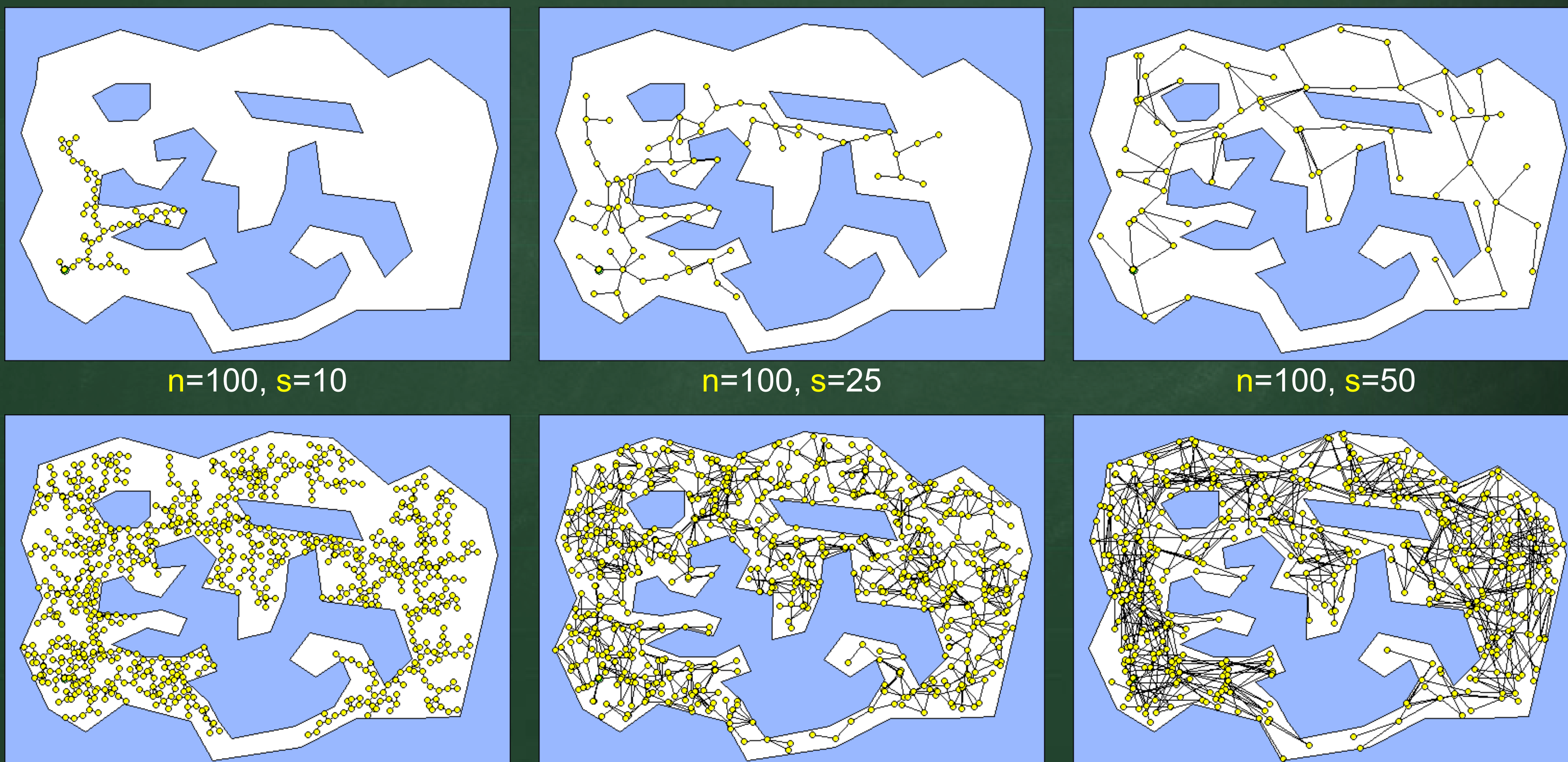
    add new node p to V with parent v *// i.e., add edge from v to p in E*

UNTIL V has n vertices



RRT Maps

- Here are some maps for various **n** and **s** values:

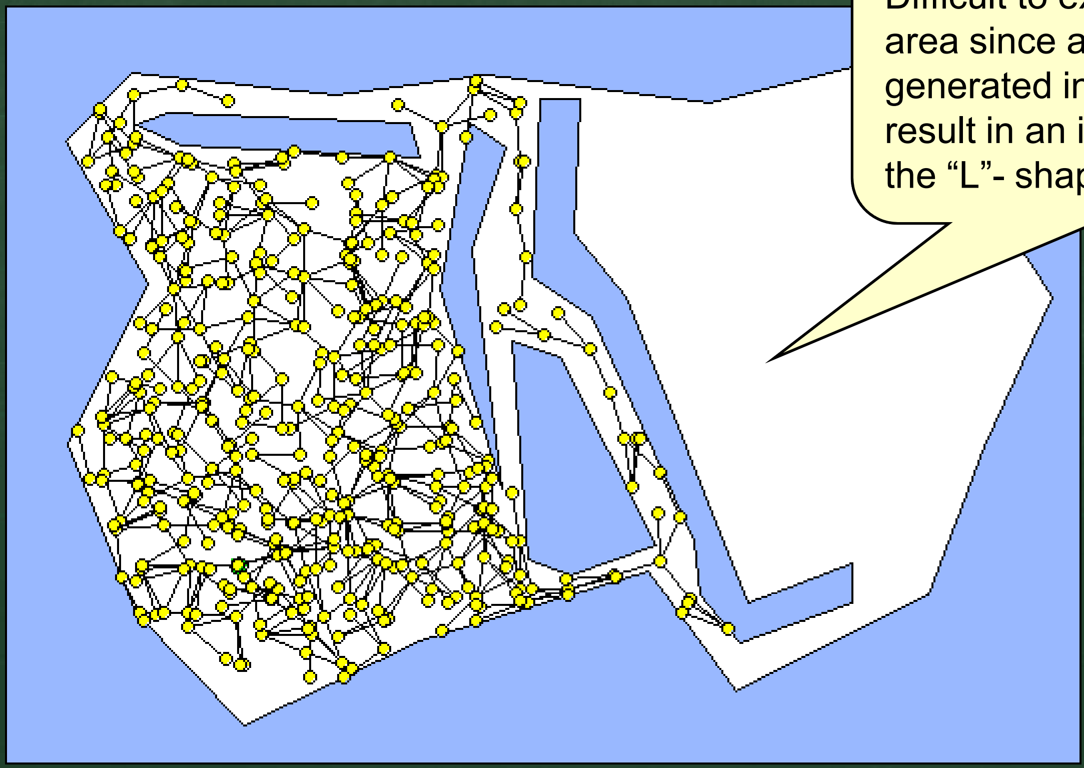


n=100, s=10 | n=100, s=25 | n=100, s=50

n=1000, s=10 | n=1000, s=25 | n=1000, s=50

RRT Problems

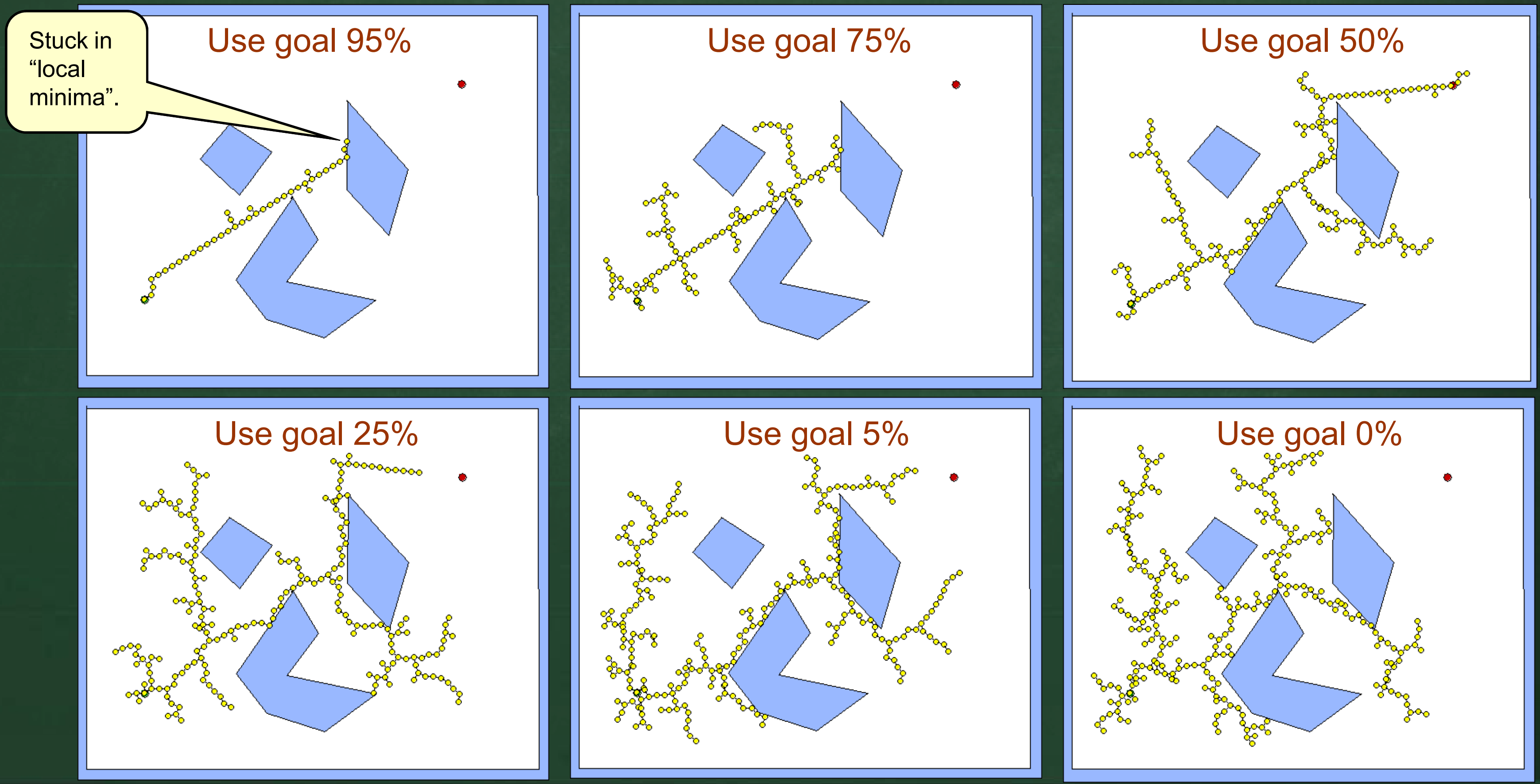
- RRTs have problems expanding through narrow passages and getting around obstacles:



Difficult to expand into this area since any random points generated in this area tend to result in an intersection with the “L”- shaped obstacle.

RRT Guiding

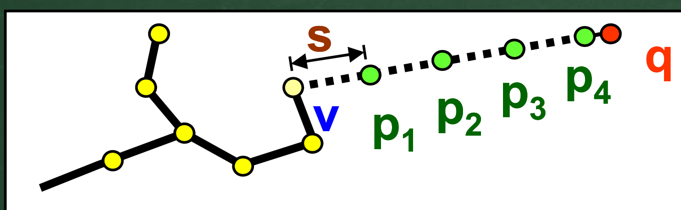
- Can we bias the results to head towards the goal ?
  - Use goal point for expand direction instead of random



Greedy RRTs

- A greedy approach to the RRT growth is to allow the tree to expand beyond the step size ***s***:

Without greedy

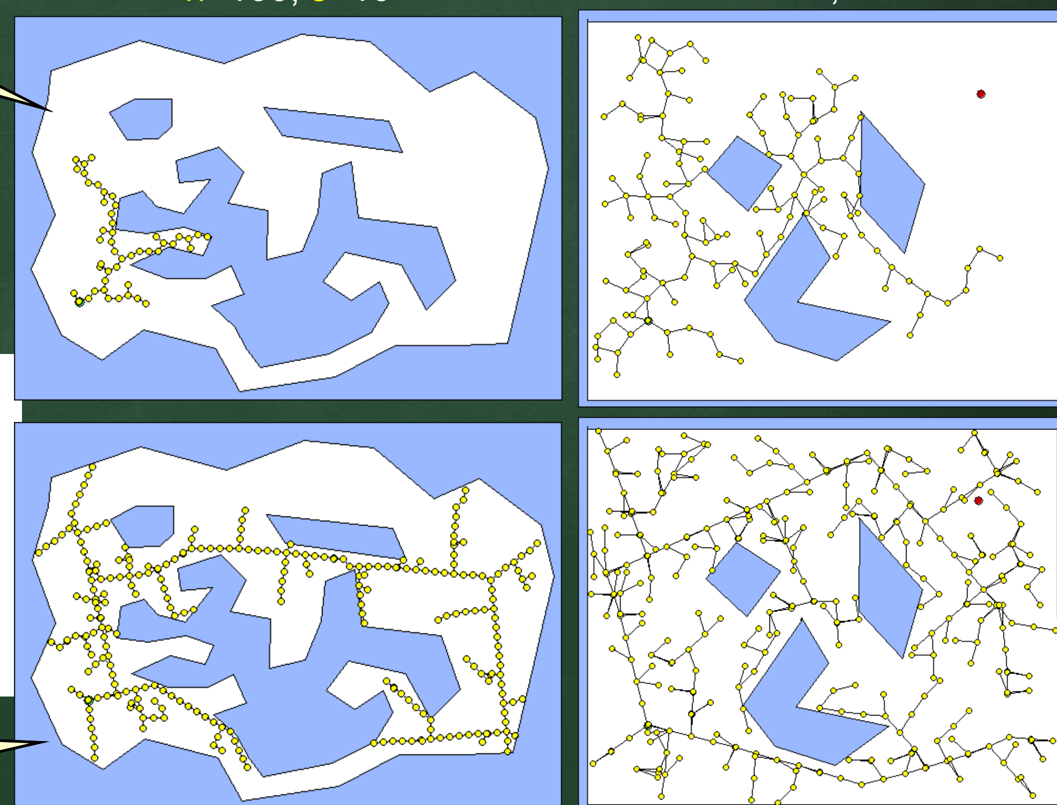



```
LET V contain the start vertex and E be empty.
REPEAT
  LET q be a random valid robot configuration
  REPEAT
    LET v be the node of V that is closest to q.
    LET p be point along vq at distance s from v.
    IF (vp is a valid edge) THEN
      add new node p to V with parent v
  UNTIL (p is invalid) OR (p is within distance s from q)
UNTIL V has n vertices
```

Just need to add this REPEAT loop.

With greedy

n=100, s=10 n=200, s=25



Reaching the Goal

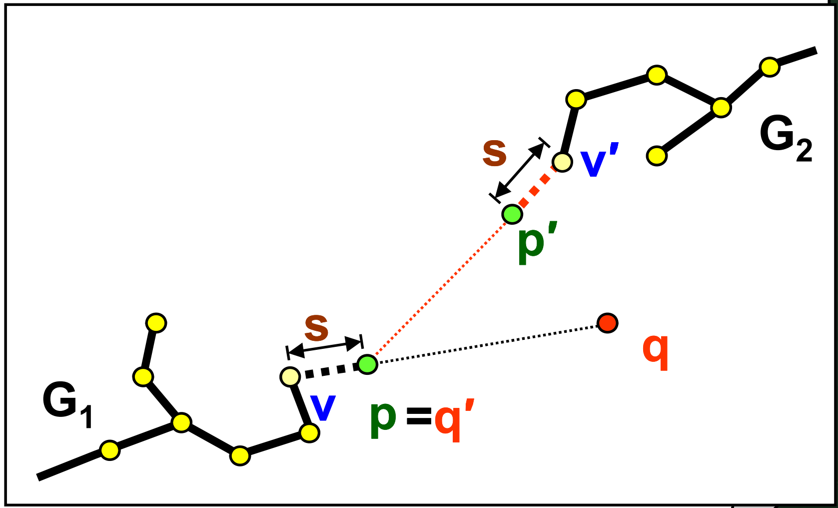
- We have yet to see how to stop the growth when the goal is reached.
- Make the following changes to the algorithm:

```
LET V contain the start vertex and E be empty.
REPEAT
    LET q be a random valid robot configuration (i.e., random point)
    LET v be the node of V that is closest to q.
    IF (distance from v to goal < s) THEN
        p = goal
    ELSE
        LET p be the point along the ray from v to q that is at distance s from v.
    IF (vp is a valid edge) THEN              // i.e., does not intersect obstacles
            add new node p to V with parent v // i.e., add edge from v to p in E
UNTIL V has n vertices
```

Dual Trees

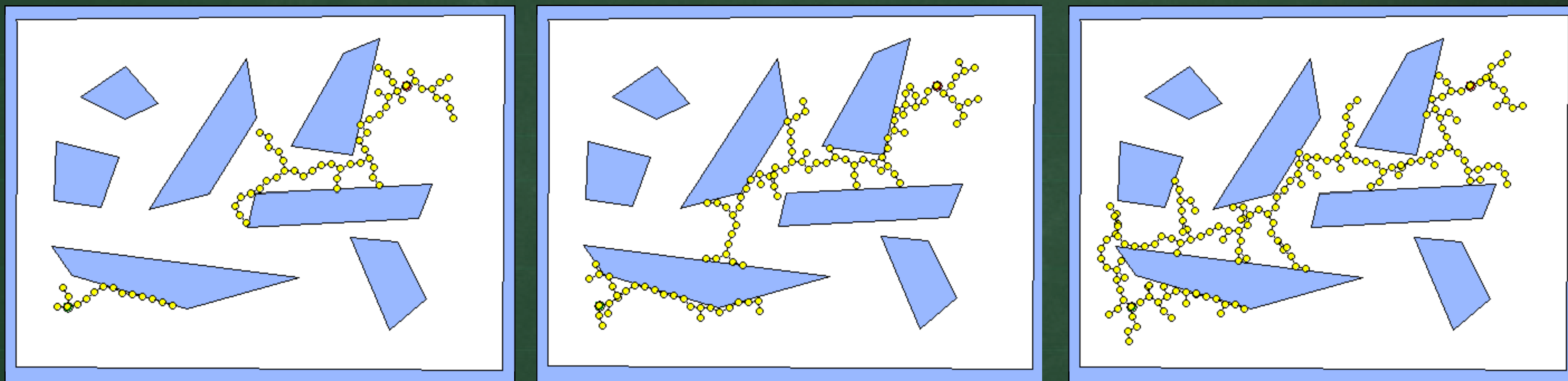
- It is more beneficial (faster) to maintain two trees $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$

LET V_1 contain the start vertex, V_2 contain the goal vertex, **LET** E_1 and E_2 be empty.
REPEAT
  LET q be a random valid robot configuration (i.e., random point)
  LET v be the node of V_1 that is closest to q.
  LET p be the point along the ray from v to q that is at distance s from v.
  IF (p is a valid configuration) **THEN**
    add new node p to V_1 with parent v
    LET q' be p
    LET v' be the node of V_2 that is closest to q'.
    LET p' be the point along the ray from v' to q'
      that is at distance s from v'.
    IF (p' is a valid configuration) **THEN**
      add new node p' to V_2 with parent v'
    Swap G_1 and G_2
  ENDIF
UNTIL V_1 and V_2 have n vertices in total



Merging Trees

- As a result, the trees grow towards each other and eventually (hopefully) merge:

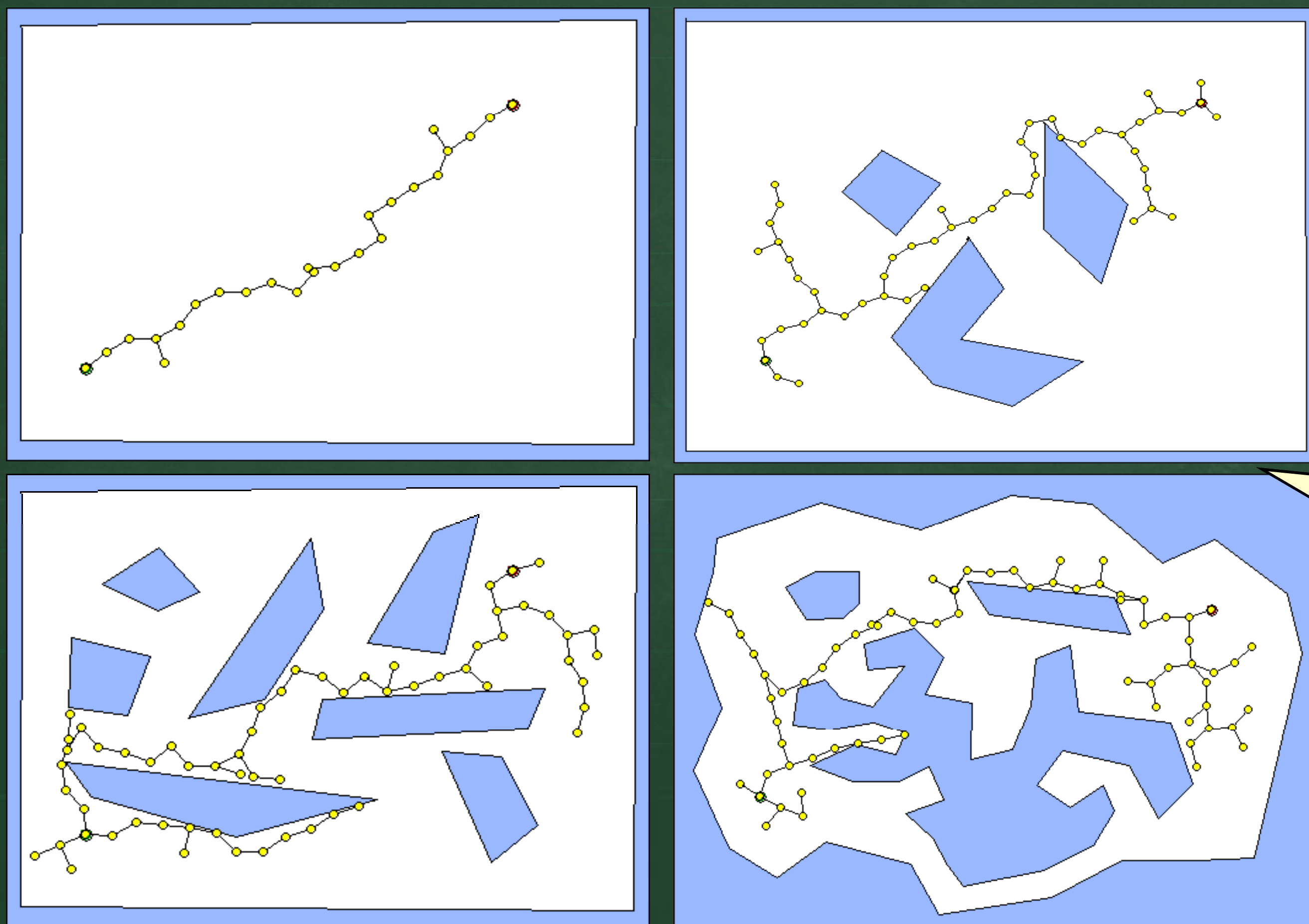


n=100, s=10   n=200, s=10   n=300, s=10

- Trees remain separate graphs, but merge when a node from one tree is within distance **s** from the other tree.

Merging Trees

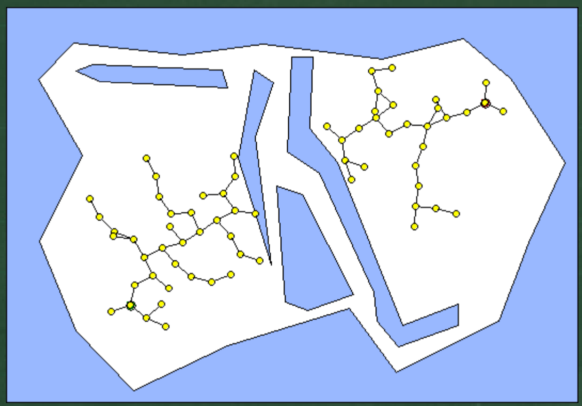
- A variety of environments work using this strategy:



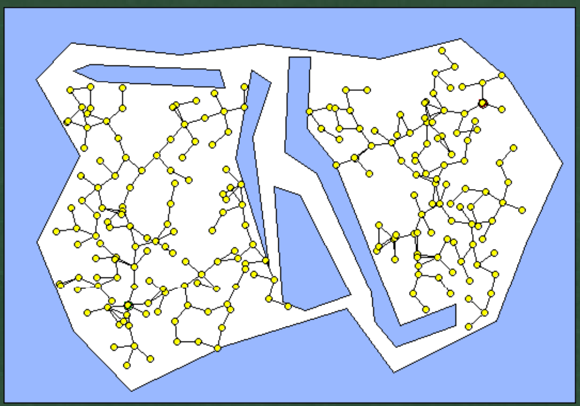
Each of these results have
n = 100
s = 20

Merging Trees

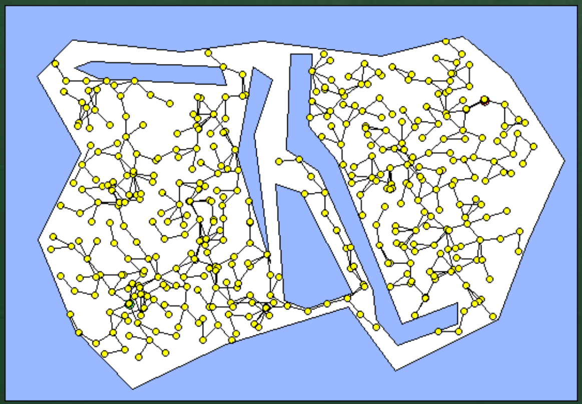
- Sometimes, it takes a while to get them to merge:



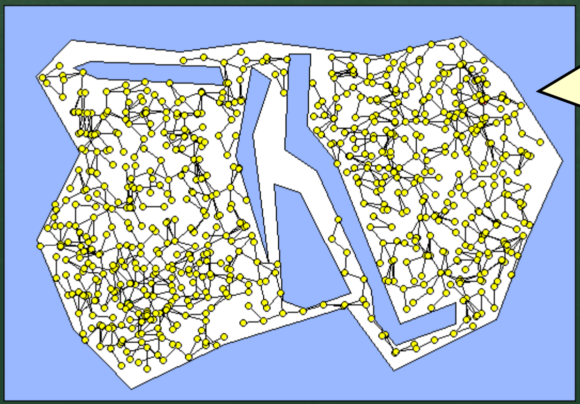
n=100, s=20



n=500, s=20



n=1000, s=20



n=10000, s=20

There are not 10,000 points here…the path was found before that.

Summary

- You should now understand:
  - How to efficiently plan the motion of a robot from one location to another in a 2D environment.
  - Various techniques for computing planned paths.
  - How to "grow" obstacles to accommodate real robot solutions.
  - How to combine what we've learned here with what we learned in robot position estimation and navigation to fully control a robot's position at all times.