# Coverage and Search Algorithms

Chapter 10

# Objectives

- To investigate some simple algorithms for covering **the area** in an environment
- To understand how to break down an environment into simple convex pieces
- To understand how to consider **searching** environments with a limited range and limited direction sensor.

# What’s in Here ?

- Complete Coverage Algorithms
  - Difficulties and Issues
  - Boustrophedon Coverage
  - Other Coverage Ideas

- Search Algorithms
  - Searching and Visibility
  - Guard Placement
  - Traveling Salesman Problem
  - Visibility Search Paths
  - Searching With Limited Range Visibility

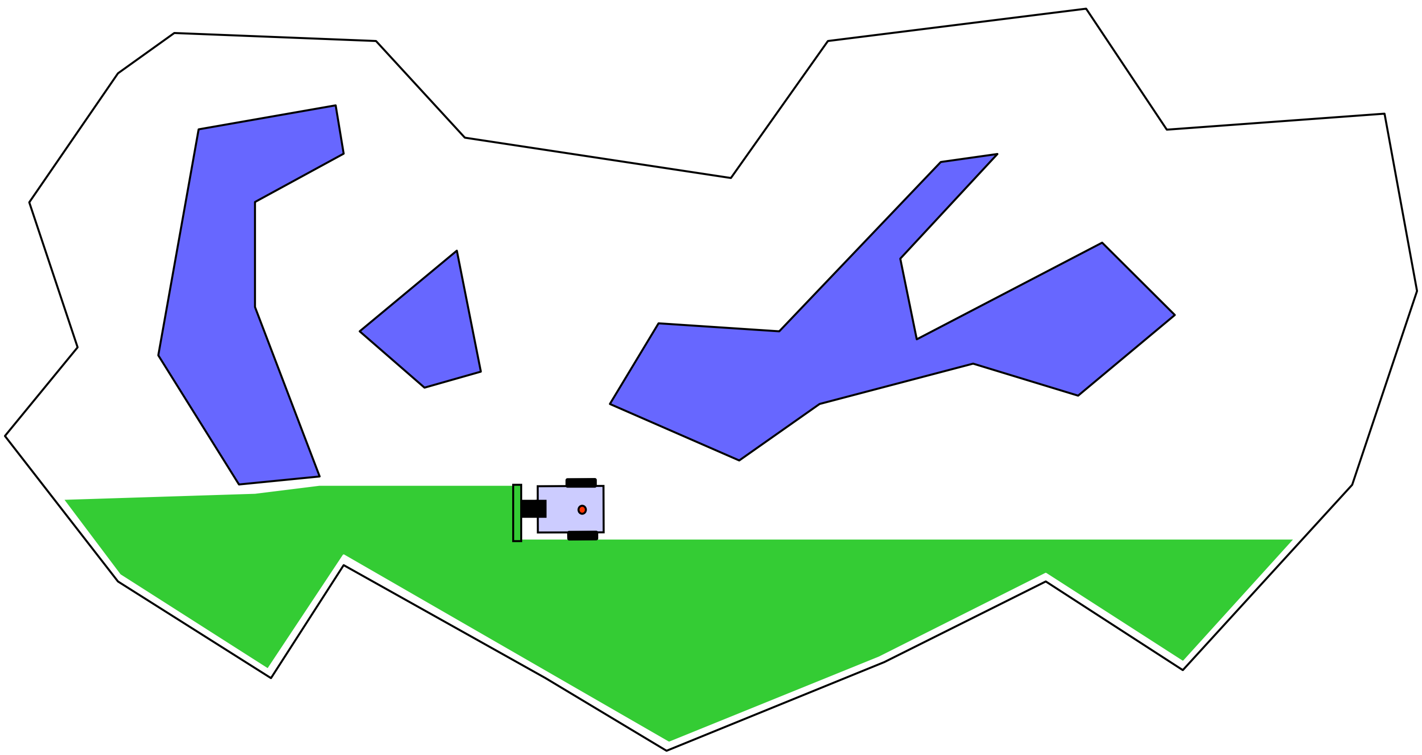
# Complete Coverage Algorithms

# Coverage Algorithms

- A *complete coverage algorithm* produces a path that a robot must travel on in order to "cover" or travel over the entire surface of its environment.
- Applications include:
  - vacuum and sweeping
  - painting
  - searching
  - security patrolling
  - map verification
  - etc…

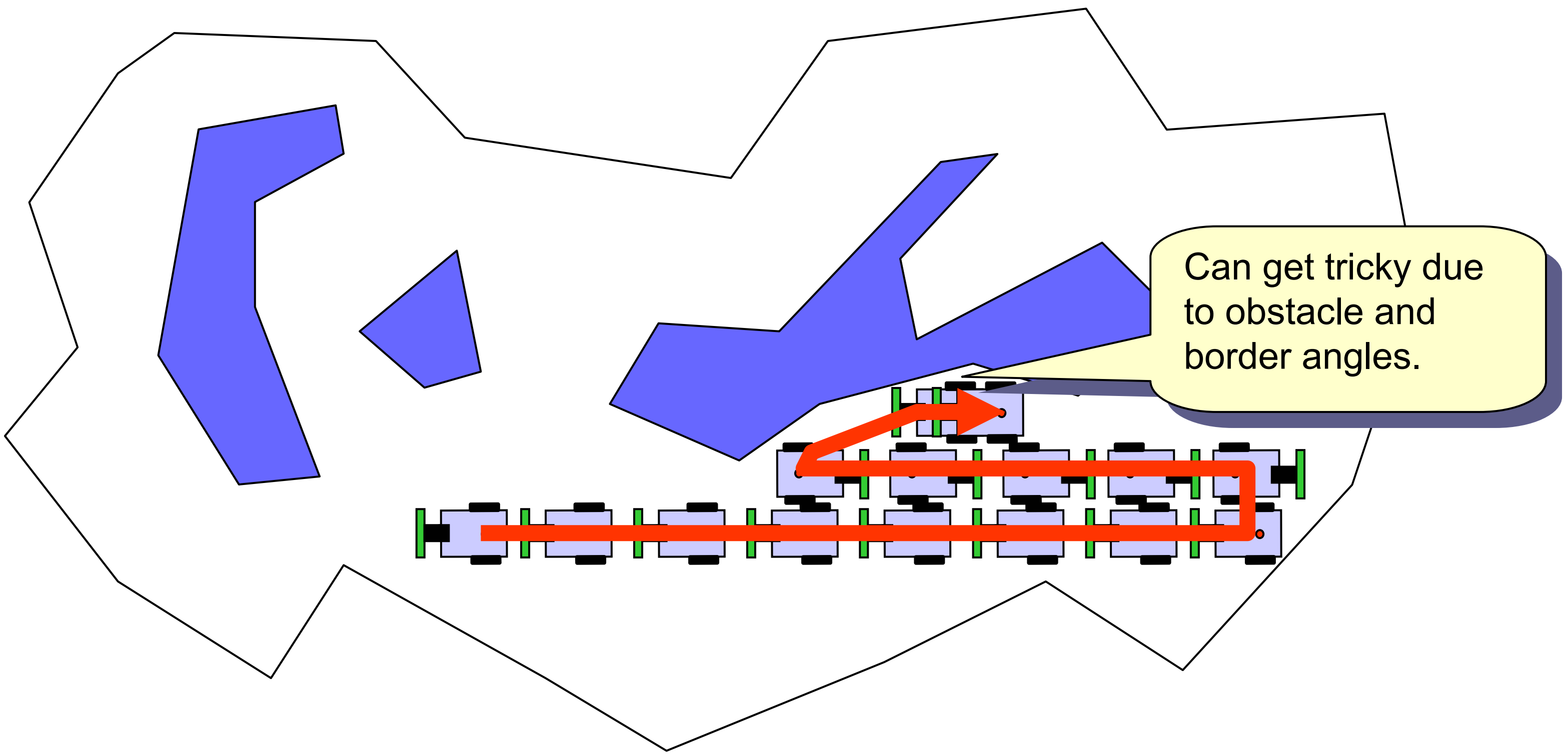
# Coverage Algorithms

- How can we determine a **valid** path that the robot can take to cover the whole environment ?



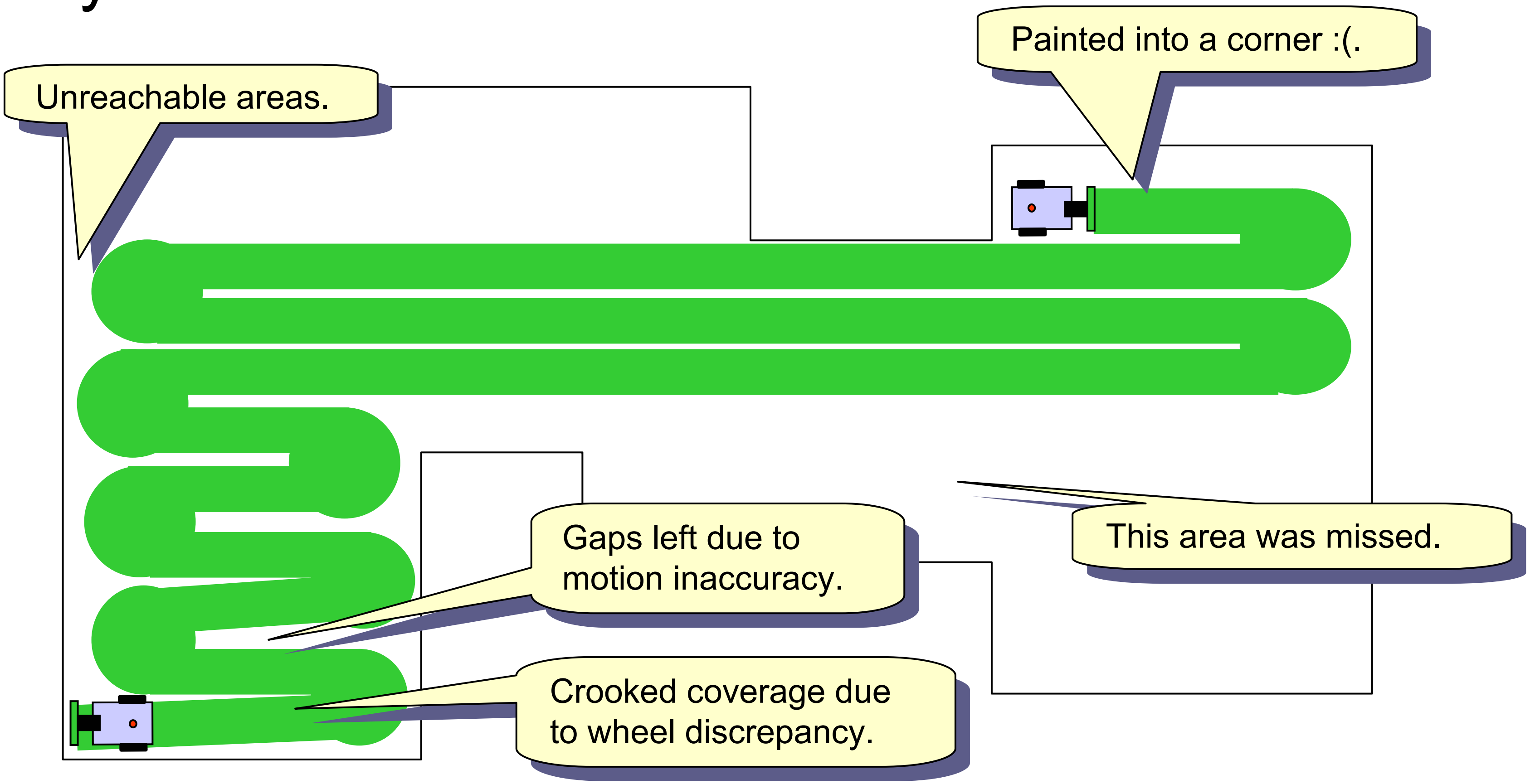
# Coverage Algorithms

- One approach is to simply travel in some fixed direction (e.g., North) until an obstacle is encountered, then turn around…cover in strips:



# Coverage Algorithms

- Even in rectilinear environments, many problems may arise:

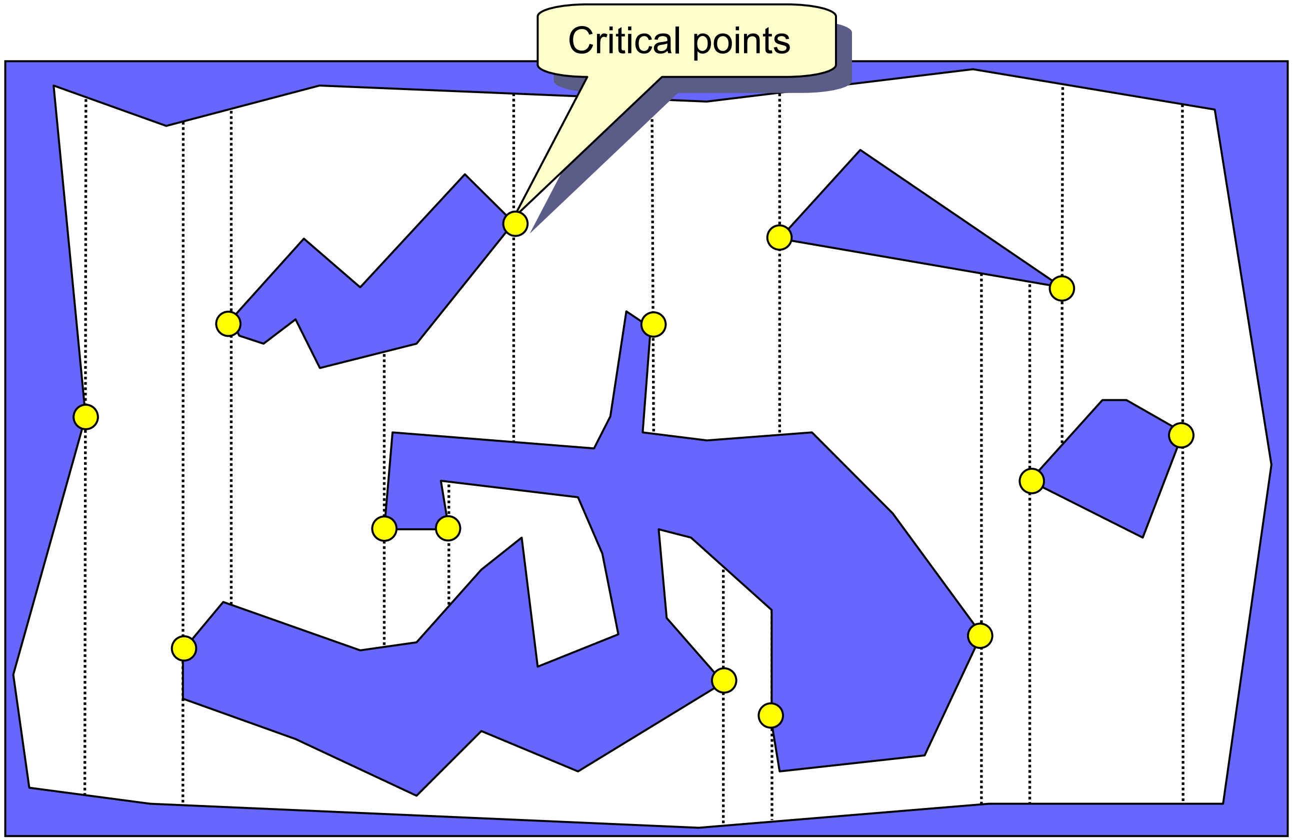


# Coverage Algorithms

- Is there any hope ?
  - there will always be some error in terms of coverage.
    - may still miss close to edges and in corners
  - allowing overlapping coverage will help
  - dividing environment into smaller "chunks" will help
- For most applications (not painting the floor) being "close enough" to the obstacles is sufficient.
  - sensors can "pick-up"/detect from a certain distance away.
  - sometimes, a rough coverage is enough.

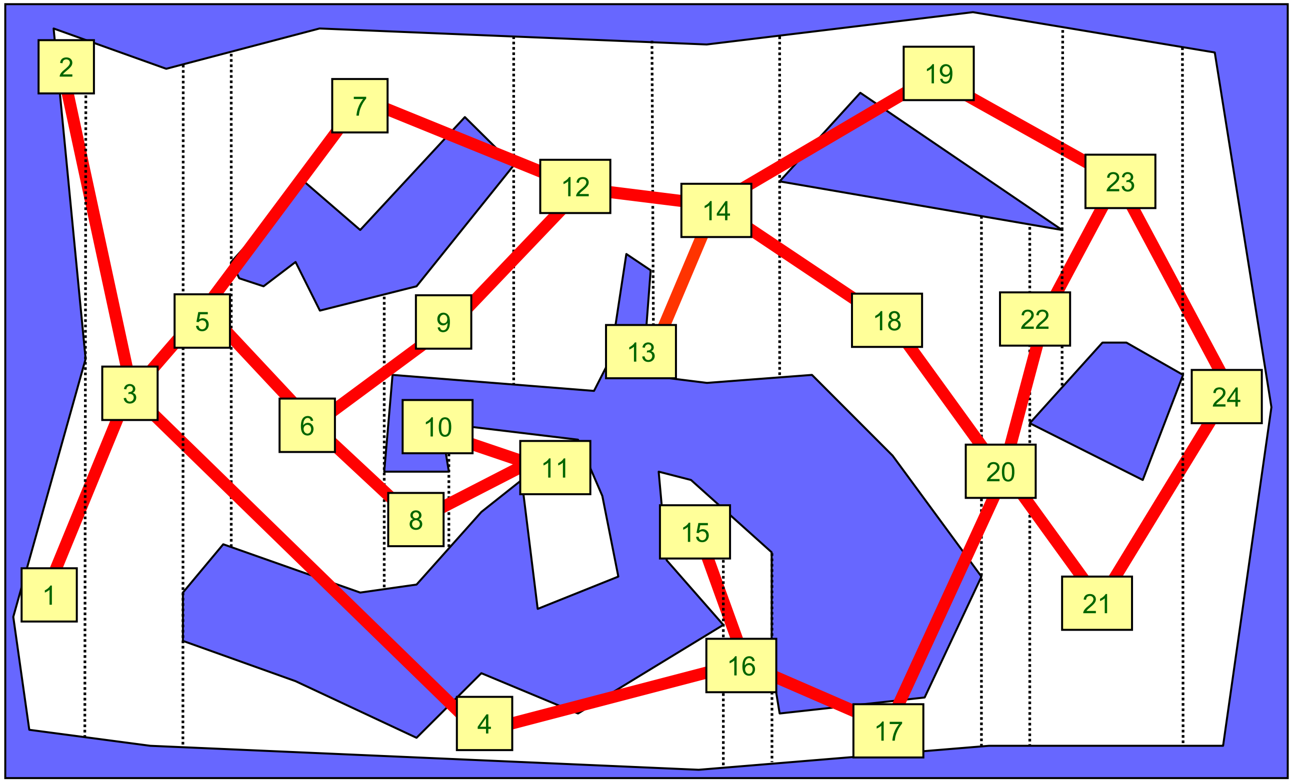
# Boustrophedon Coverage

- Recall the Boustrophedon cell decomposition of a polygonal environment:



# Boustrophedon Coverage

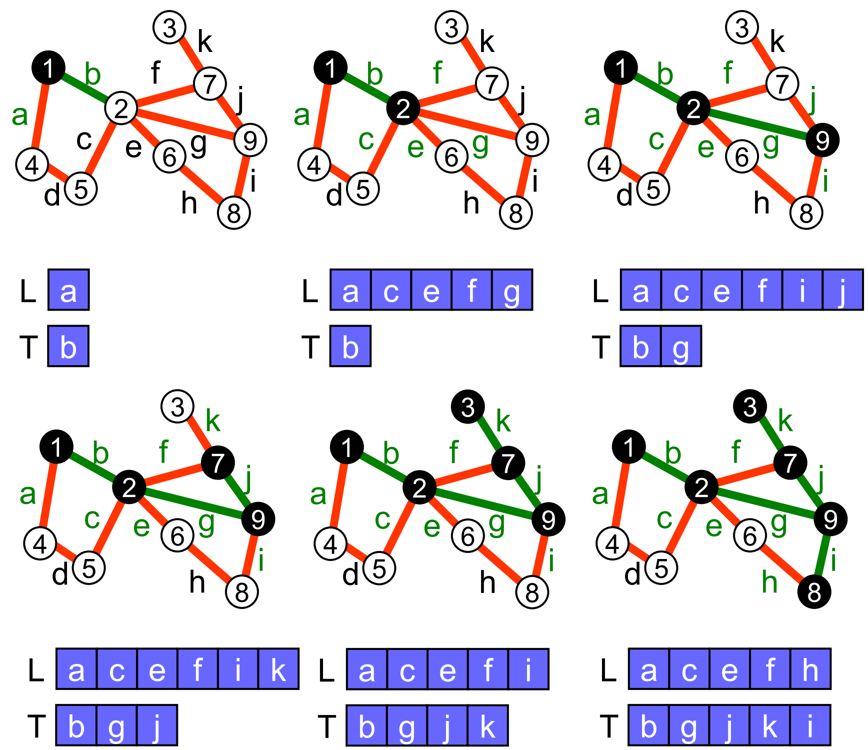
- Now connect adjacent cells to form a graph and consider an arbitrary ordering of the cells:
  - (e.g., from left to right)



# Finding a Path

- Perform a depth-first-search (DFS) on the graph to determine an exhaustive walk through the cells:

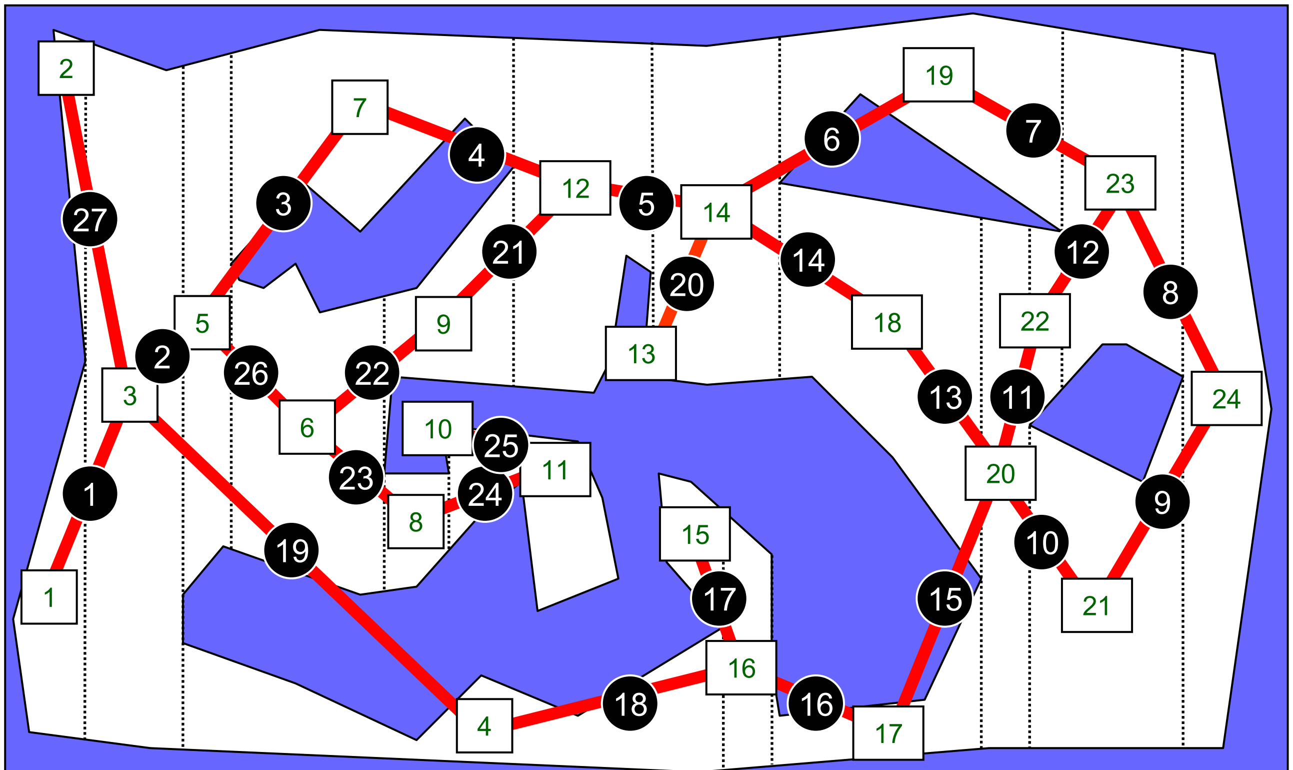
```
dfs(G) {
  list L = empty
  tree T = empty
  choose a starting vertex x
  search(x)
  WHILE (L nonempty) DO
    remove edge (v,w) from end of L
    IF (w not yet visited) THEN
      add (v,w) to T
      search(w)
}

search(vertex v) {
  visit(v)
  FOR (each edge (v,w)) DO
    add edge (v,w) to end of L
}
```




etc…

# Finding a Path

- Here is what the DFS ordering may have produced in our example:

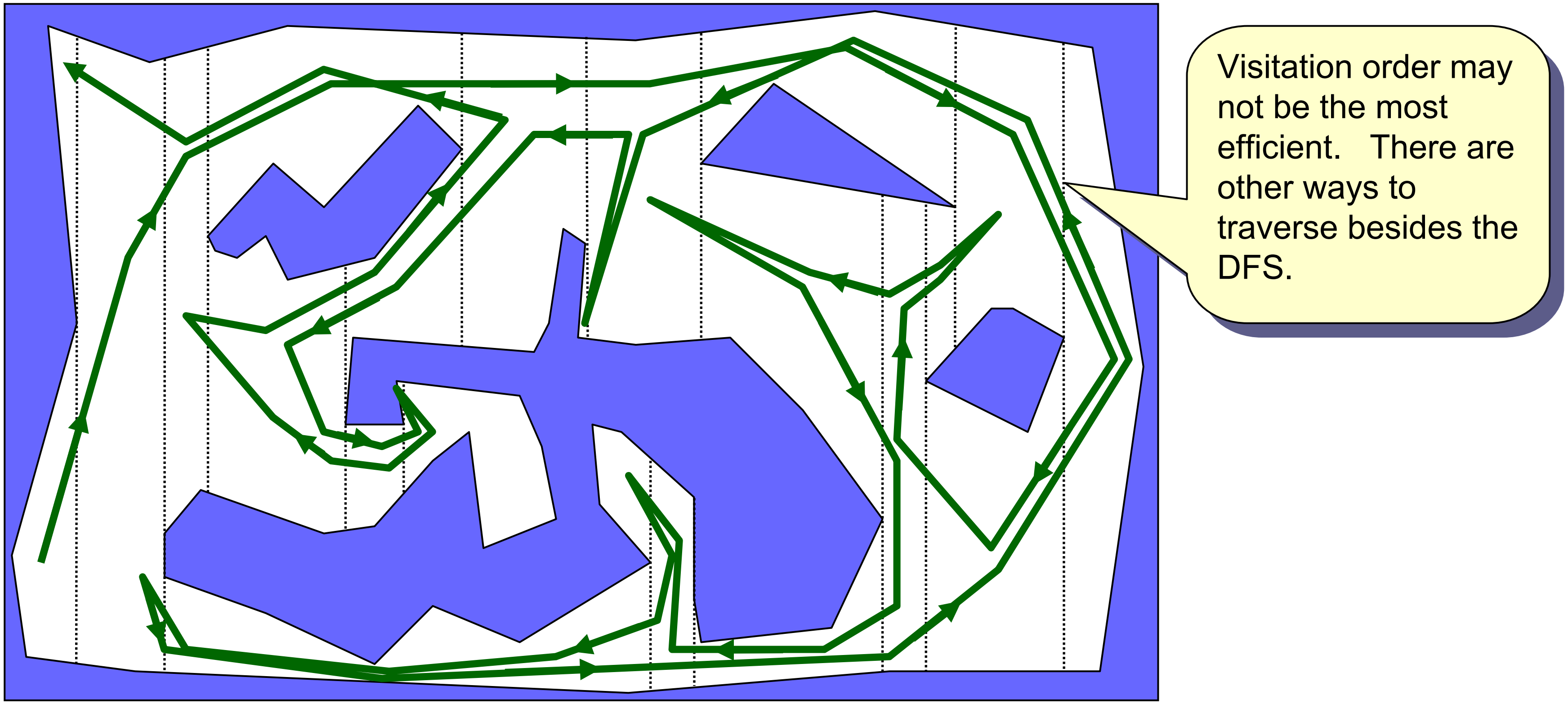


Cells visited in the following order (blue numbers indicate backtracking):

**1-3-5-7-12-14-19-23-24-21-20-22-23-22-20-18-14-18-20-17-16-15-16-4-3-4-16-17-20-21-24-23-19-14-13-14-12-9-6-8-11-10-11-8-6-5-6-9-12-7-5-3-2**

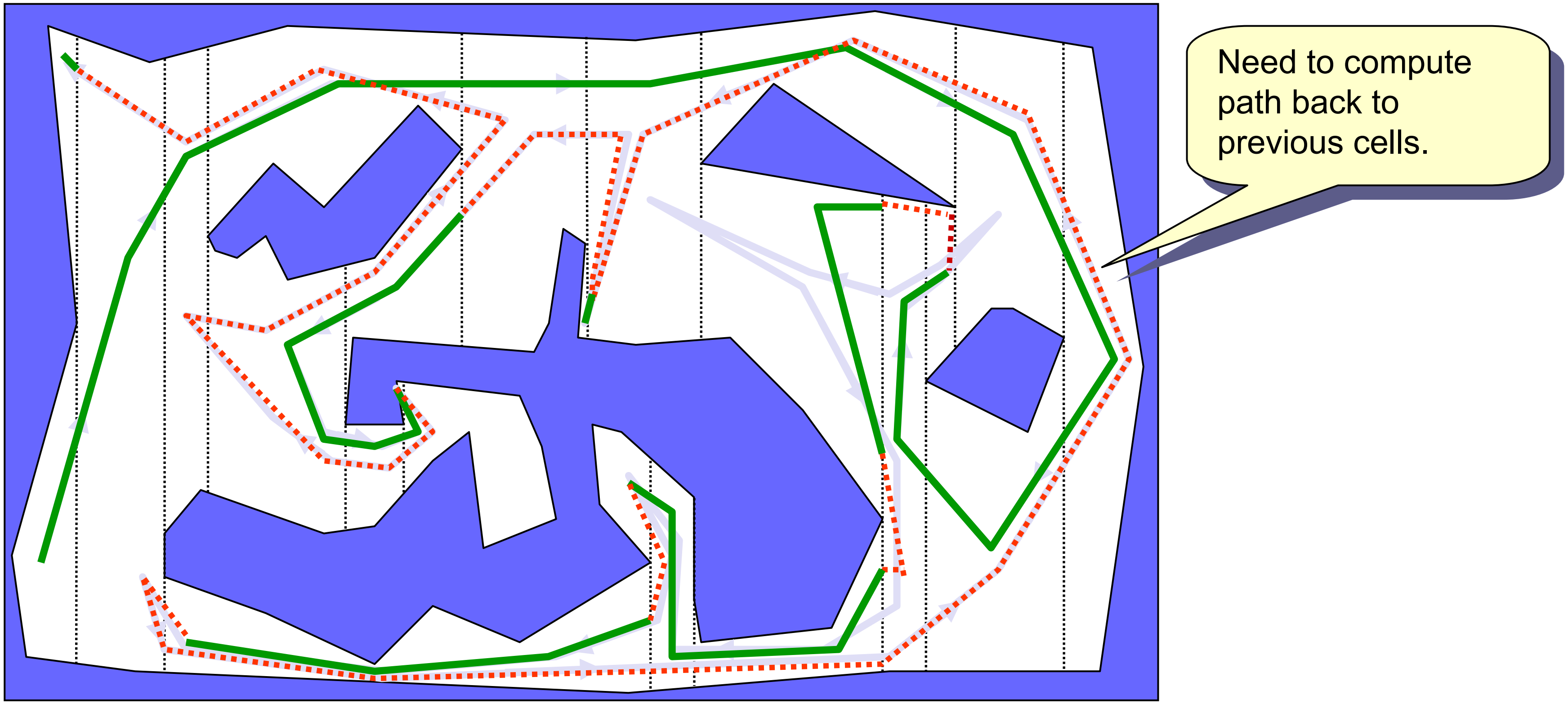
# Coverage Along a Path

- Once a path is found, the robot visits all of these cells in that order:



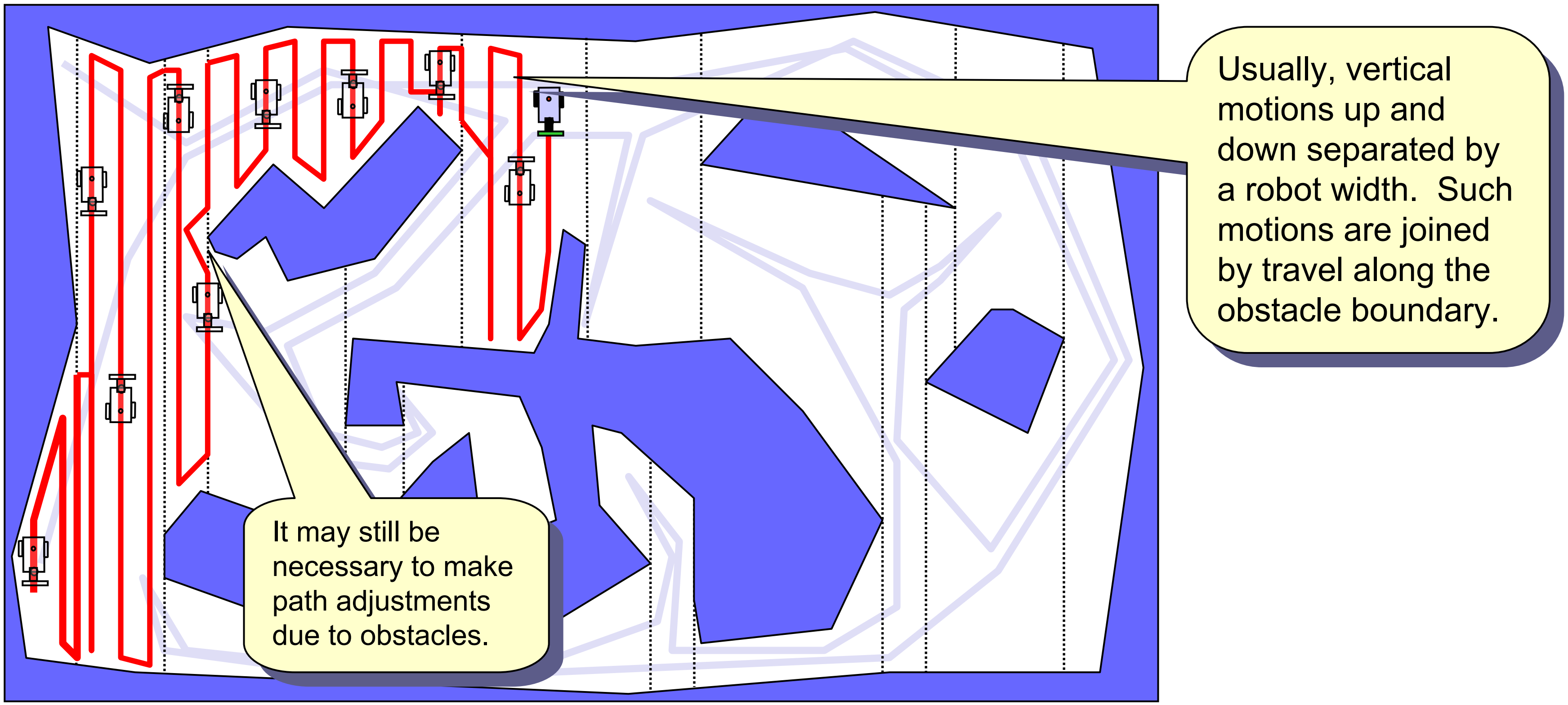
# Coverage Along a Path

- When coming back to cells already visited, it is not necessary to re-cover the cell again:



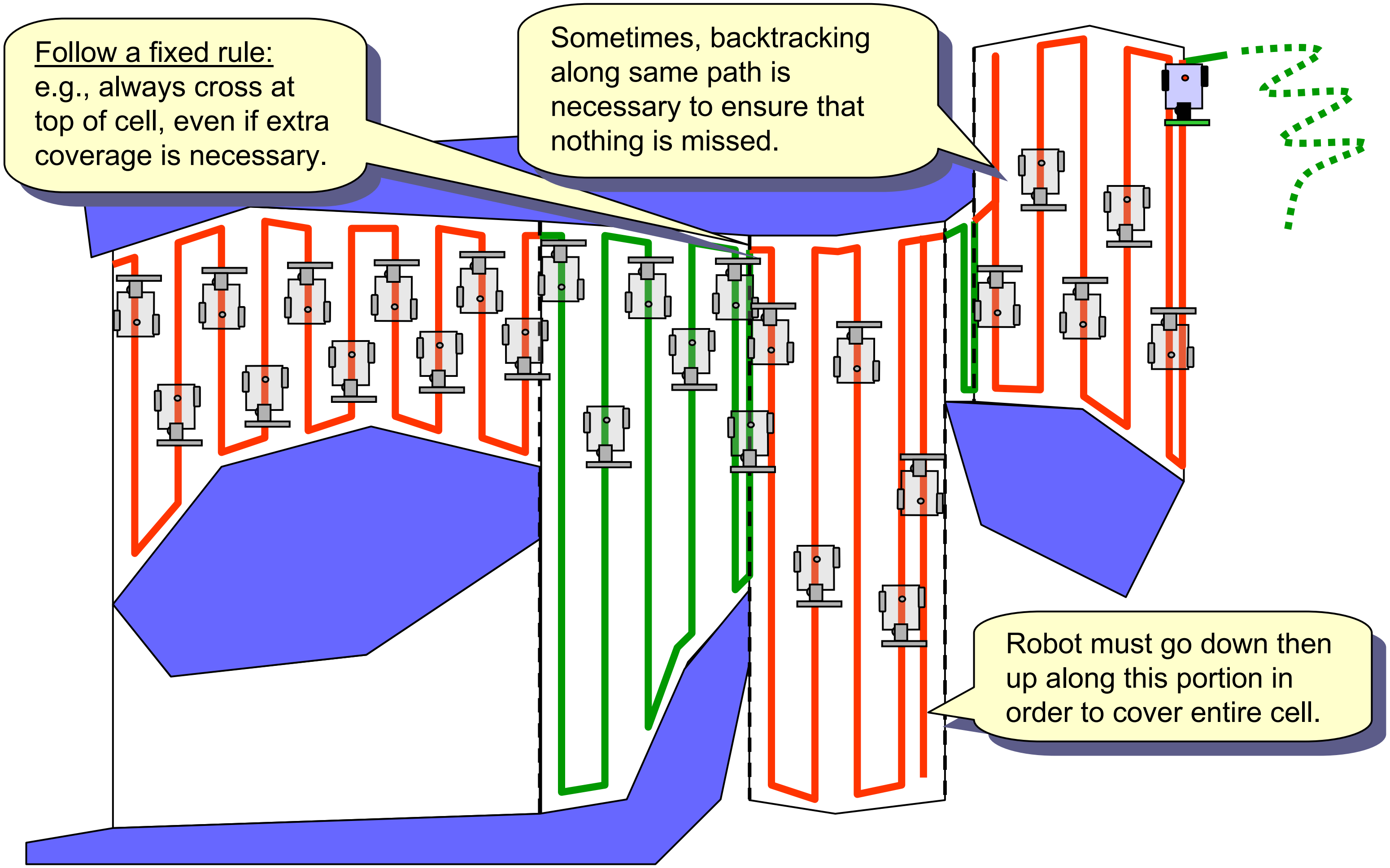
# Coverage Along a Path

- When entering a cell, the robot performs some simple maneuvers to cover the cell's entire area:



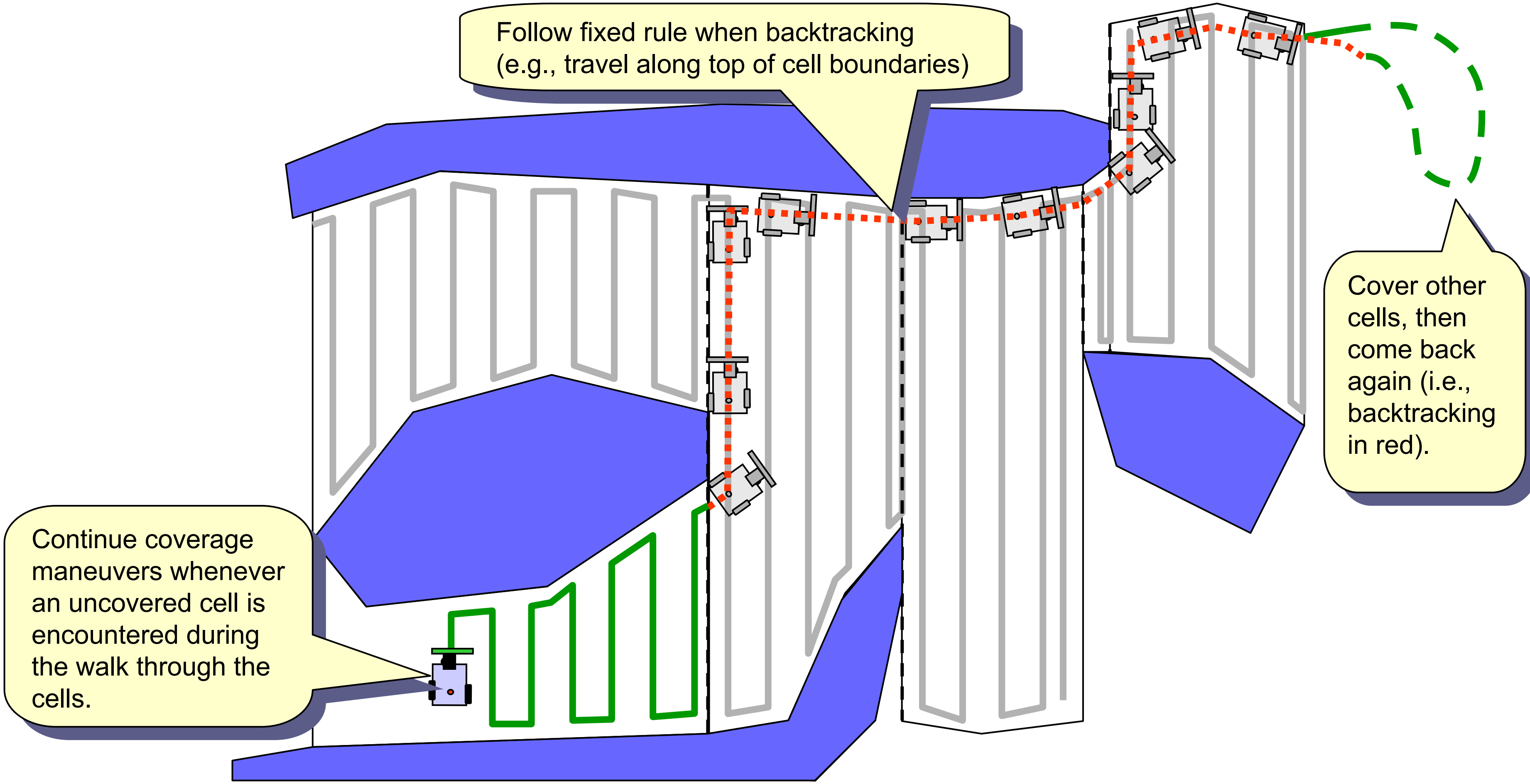
# Coverage Along a Path

- Must take care when crossing one cell to another:



# Coverage Along a Path

- When backtracking, follow along cell boundaries:



# Other Coverage Ideas

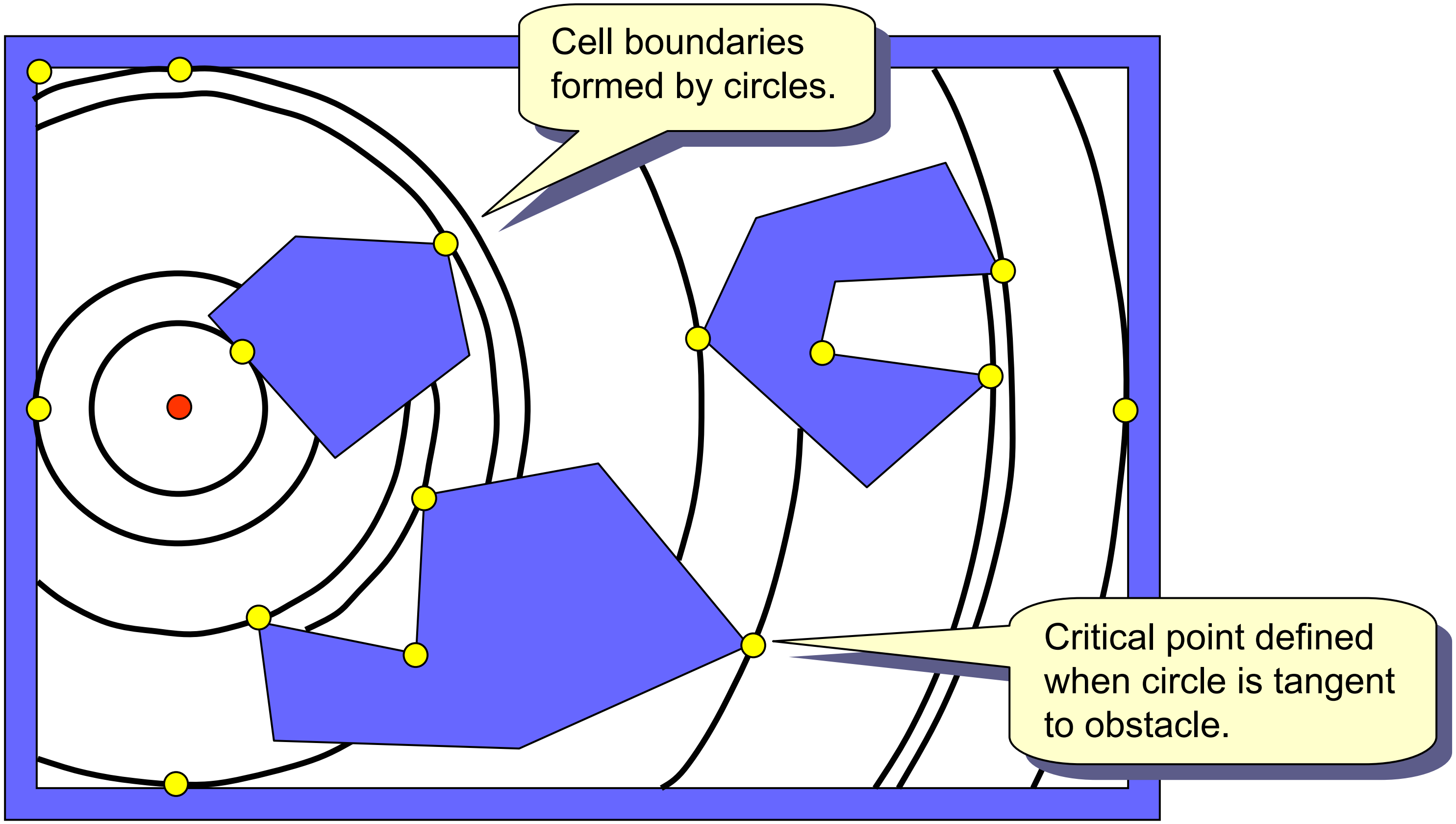
- There are other ways to decompose the environment into cells and compute a coverage path. For example:
  - circular or diamond-shaped spiral cells
  - spike cells
  - brushfire decomposition cells (like GVD)

We will look very briefly at these two

- Each of these, however, may require different traversal techniques.
- Their choice should depend on the robot's sensor characteristics.

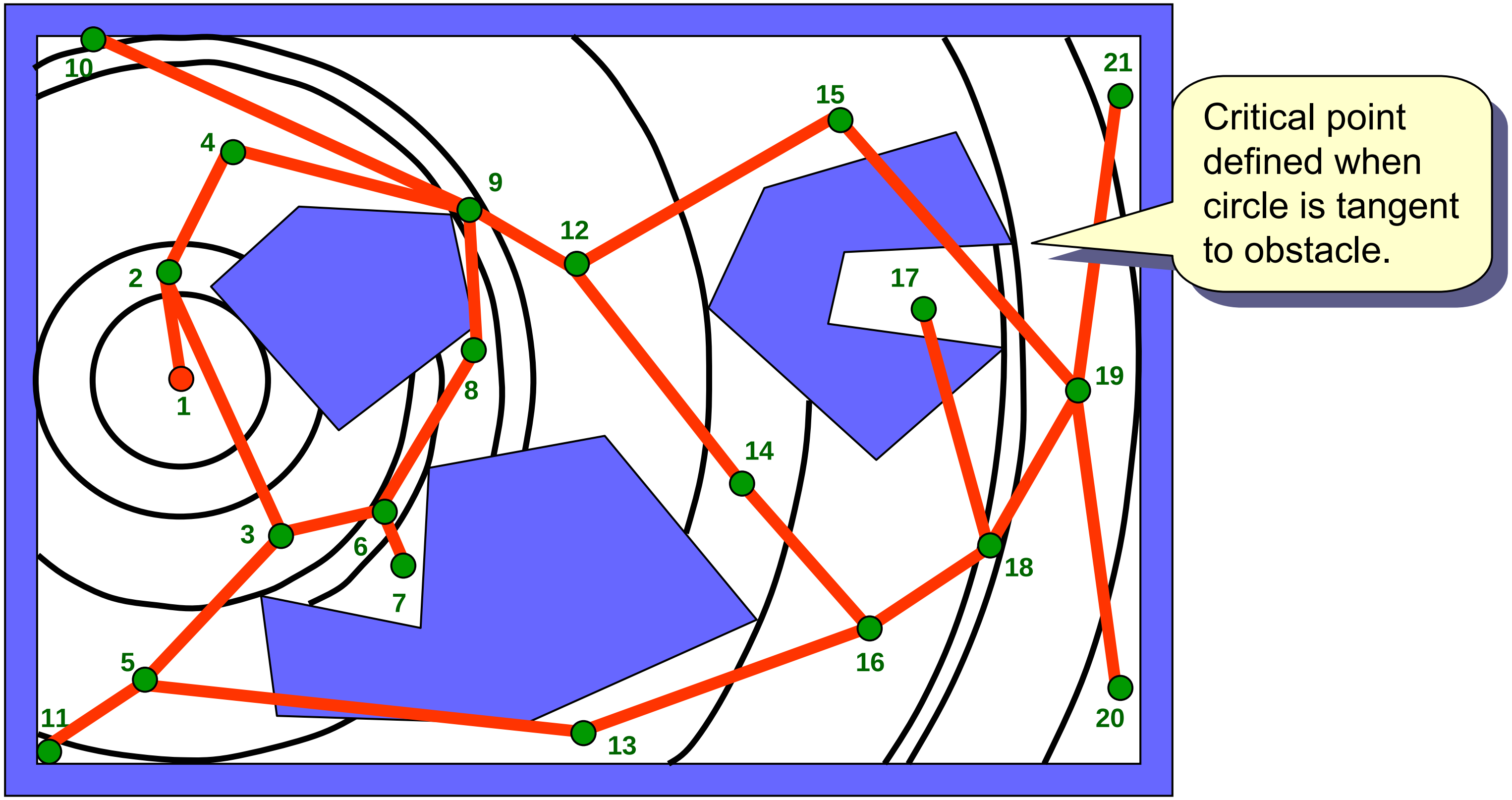
# Circular Coverage Patterns

- We can alternatively create circular cells defined by circles extending outwards from the start location:



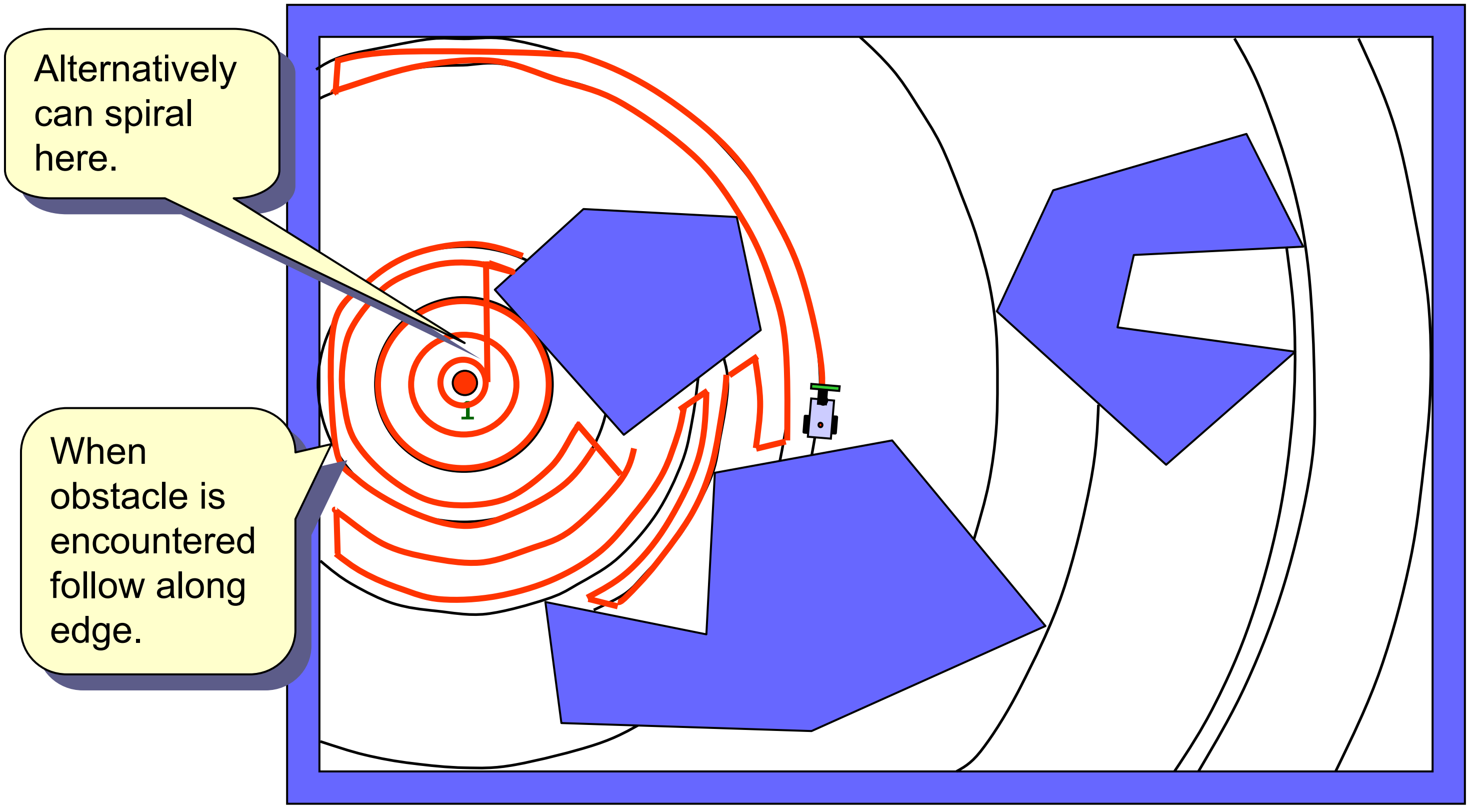
# Circular Coverage Patterns

- Once again, interconnect cells and do DFS to find path in graph:



# Circular Coverage Patterns

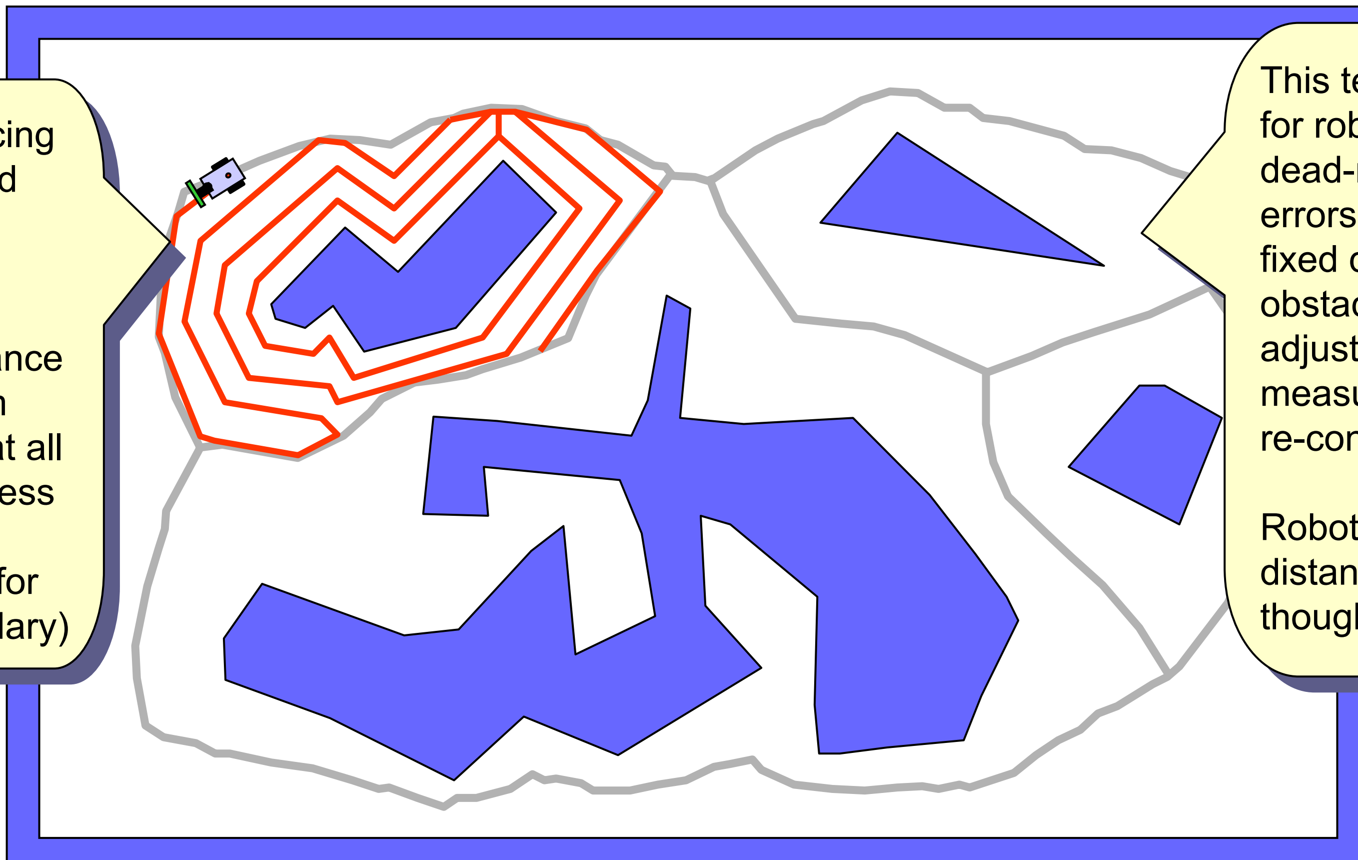
- Traverse each cell by making "laps" around the cell where each lap is separated by the robot width:



# Brushfire Decomposition

- Can even break down into regions based on GVD and then traverse cells around obstacles:

When tracing cell around obstacle, robot maintains fixed distance away from obstacle at all times (unless when adjusting for cell boundary)



This technique is best for robots that have dead-reckoning errors. By maintaining fixed distance from obstacle, robot can re-adjust its measurements and re-confirm its position.

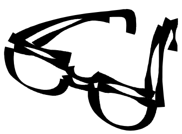
Robot needs long distance range sensor though.

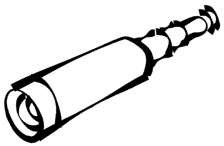
# Search Algorithms

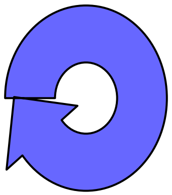
# Searching

- Consider covering an environment for the purpose of searching for other robots, fire, intruders, any identifiable object etc…

- Robot is equipped with one or more search sensors of some kind which have either:
  - unlimited or limited detection range
  - omni-directional (i.e, 360°) or limited direction detection capabilities

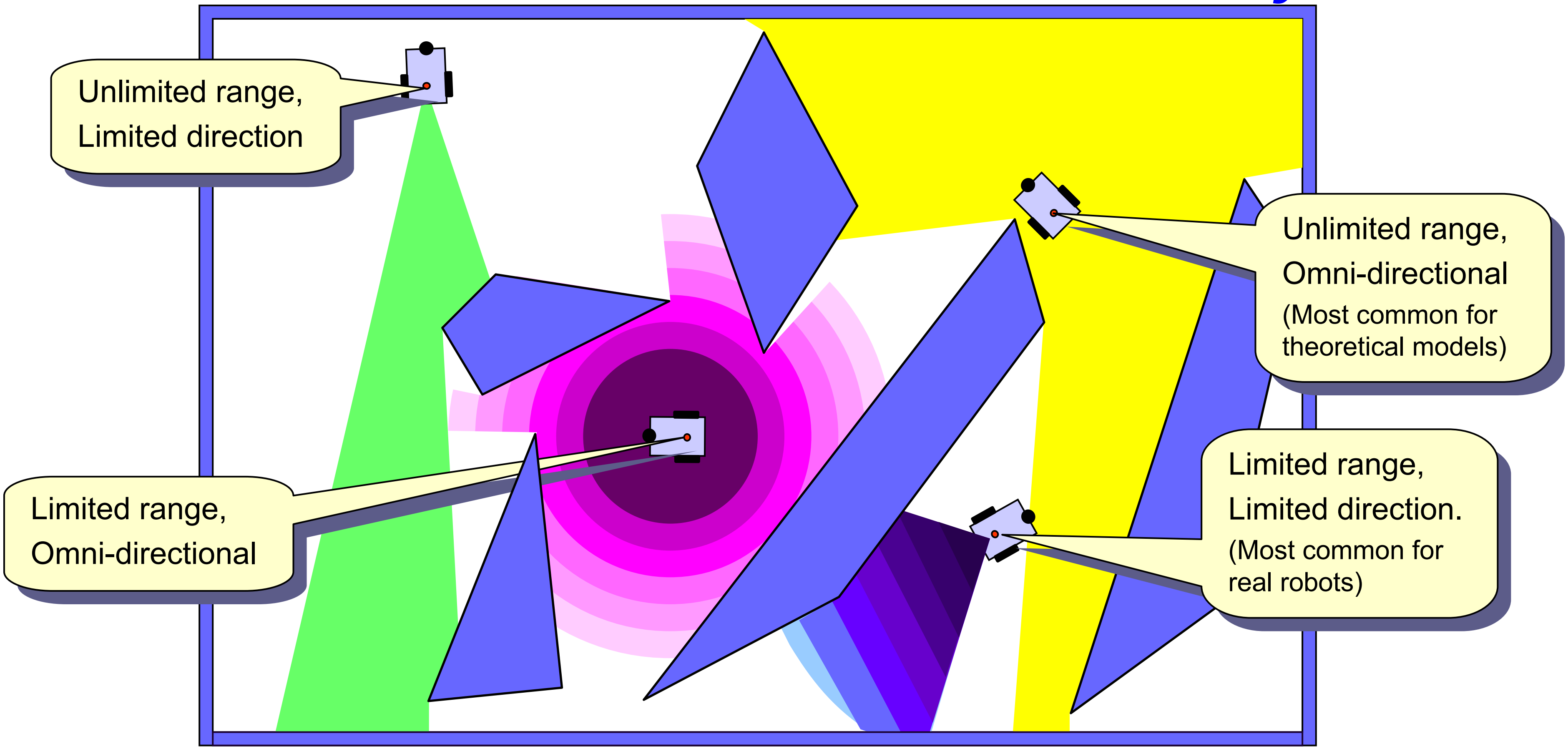






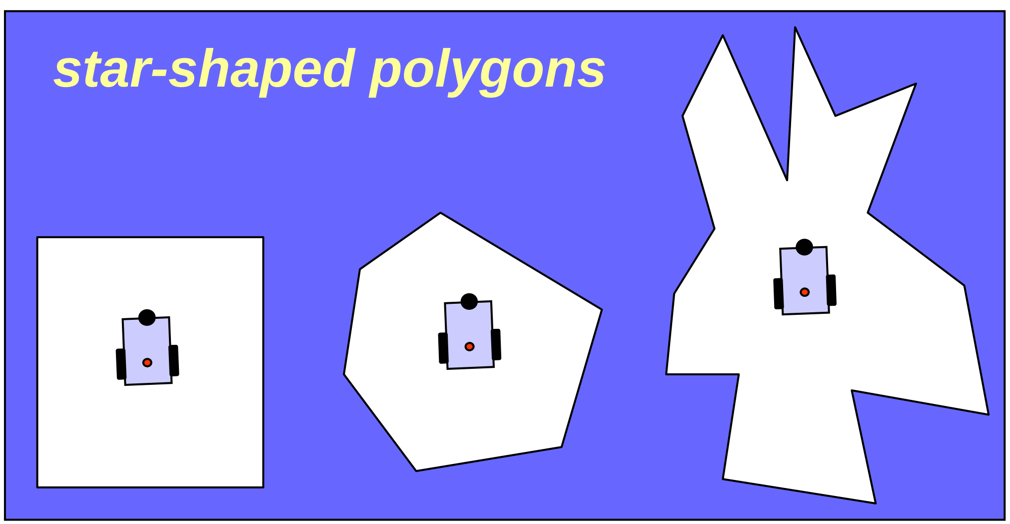
# Searching

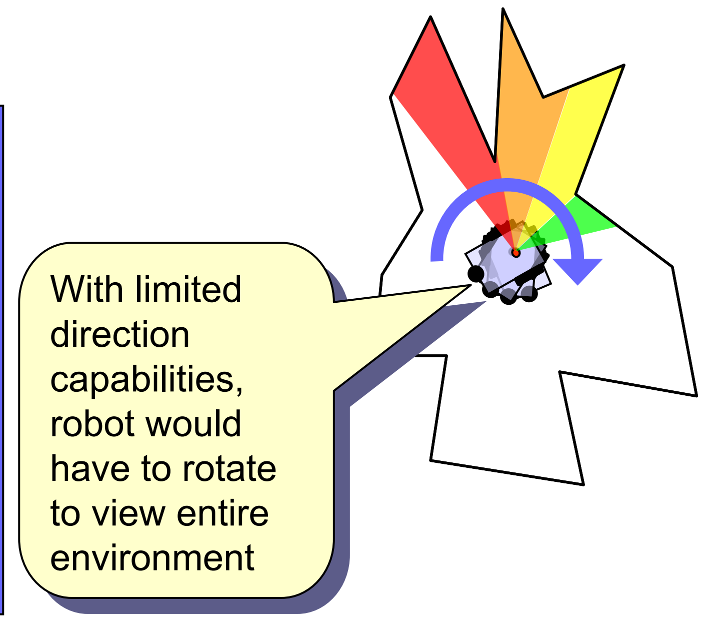
- As the robot moves around in the environment, it is able to search based on its current ***visibility***:



# Visibility

- Consider a simple environment with no obstacles and a robot with omni-directional sensing with unlimited range capabilities.
- Which environments can it search (i.e., see) completely without moving ?

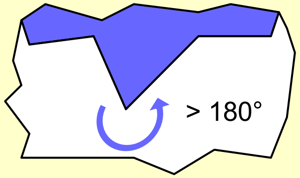


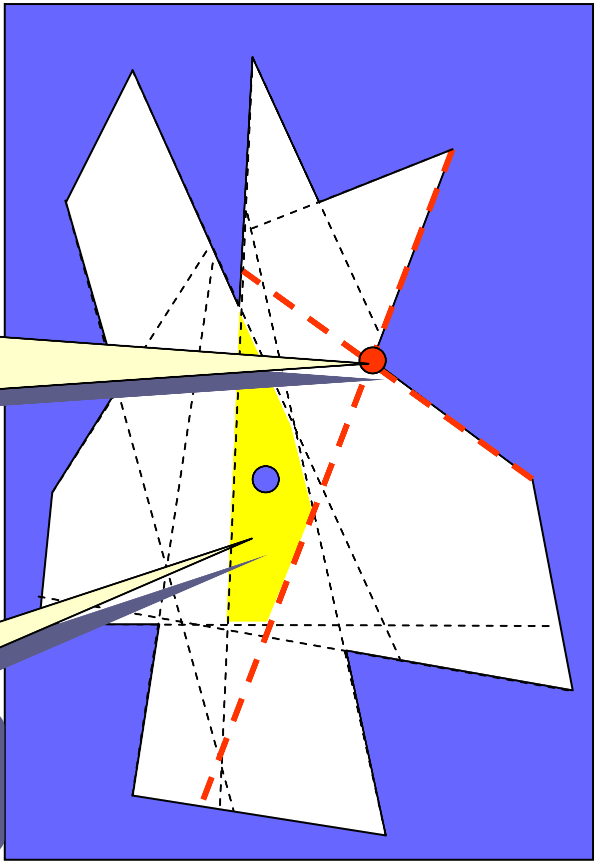



# Visibility

- The ***kernel*** of a star-shaped polygon is the area of the polygon from which the robot can “see” the entire boundary of the environment:

Extend lines from each ***reflex*** vertex parallel to edges containing that vertex. A ***reflex*** vertex is one which forms an inside angle > 180°.

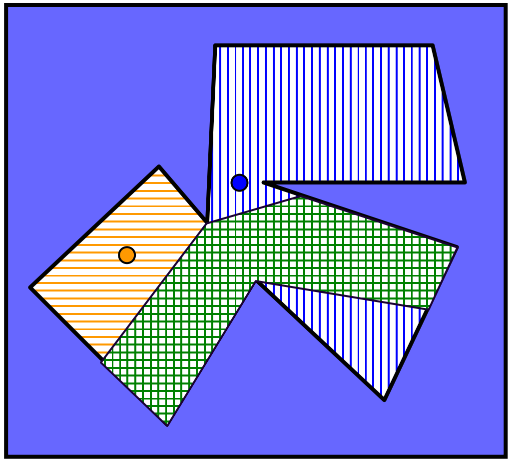


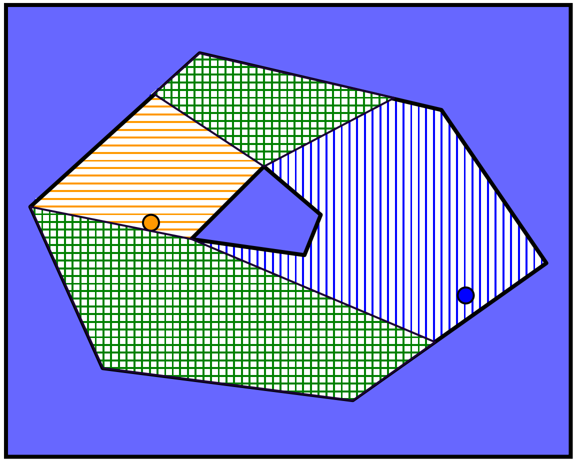



Kernel formed as intersection of the resulting half planes.

# Visibility

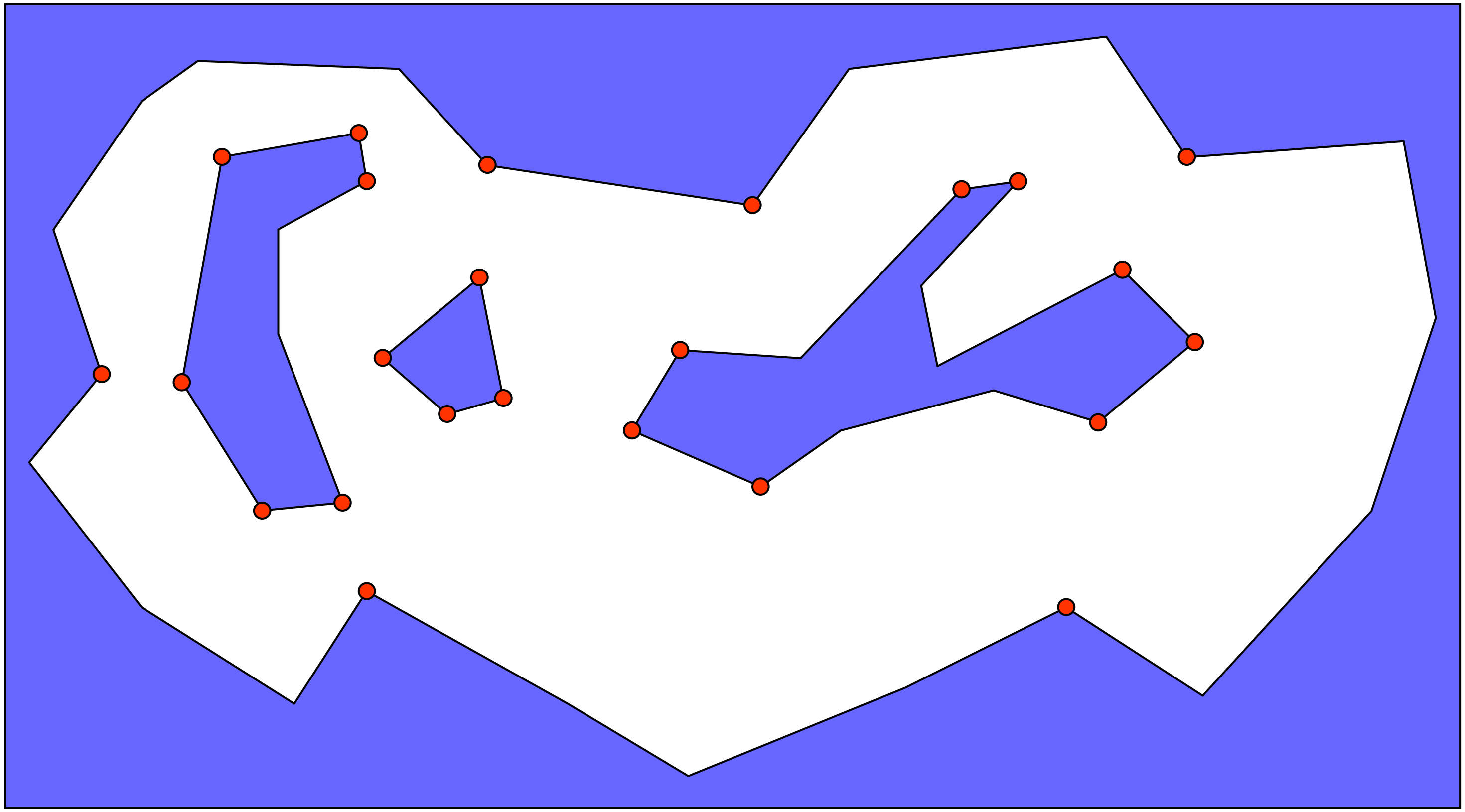
- What if environment is not star-shaped or has obstacles ?
  - kernel is empty (i.e., can’t see whole environment from one location)
  - need to determine a set of locations (i.e., view points) that cover the entire environment





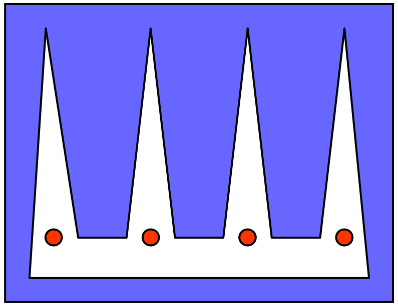
# Visibility

- Placing robot at each reflex vertex will ensure complete visibility coverage.   Do you know why ?



# Guard Placement

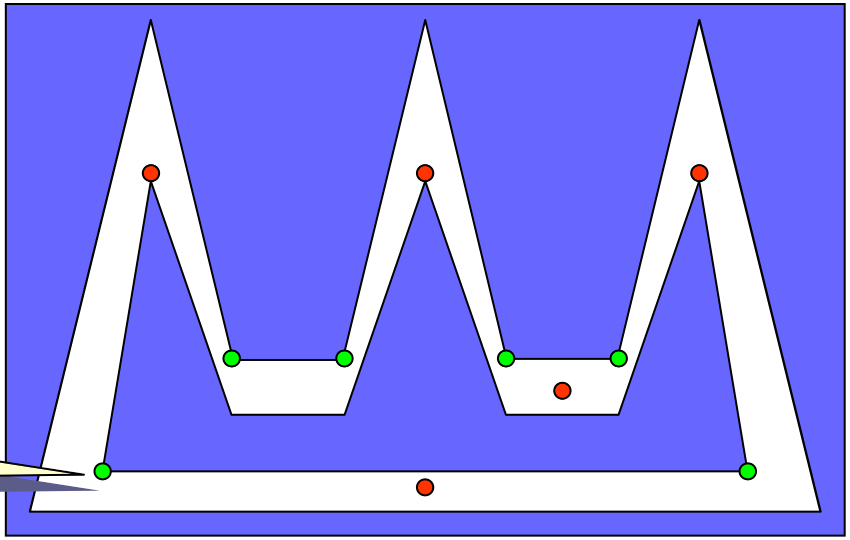
- Can we cover with less locations ?
- This problem is called the ***Guard Placement*** problem or ***Art Gallery*** problem.
- For a simple polygon environment with $n$ vertices:
  - $\lfloor n/3 \rfloor$ locations are occasionally necessary and always sufficient to have every point in the polygon visible from at least one of the locations:
  - e.g., $n$ = 12 and 12 // 3 = 4 locations are necessary and sufficient



# Guard Placement

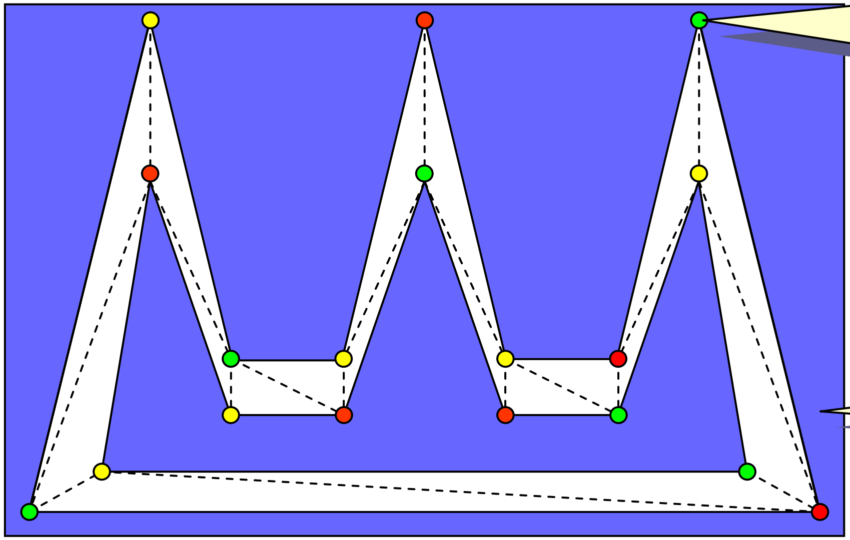
- When the environment contains **h** obstacles and has **n** edges (including obstacle edges), it can be shown that $\lfloor (n+h)/3 \rfloor$ locations are sometimes necessary and always sufficient to cover the entire environment:
  - e.g., n = 19, h = 1
    then (19+1)//3 = 20//3 = 6
    locations are necessary
    and sufficient.

There are many possible placements, here are two …



# Guard Placement

- How do we compute these locations ?
- Can do a ***3-coloring*** of the triangulation:
  - color each vertex of the triangulation with one of 3 colors
  - no two vertices sharing a triangulation edge should have the same color



Coloring is done through a DFS, but in some cases the straight forward approach does not always work…it can be tricky.

Each color here indicates a possible set of robot locations.

# Search Paths

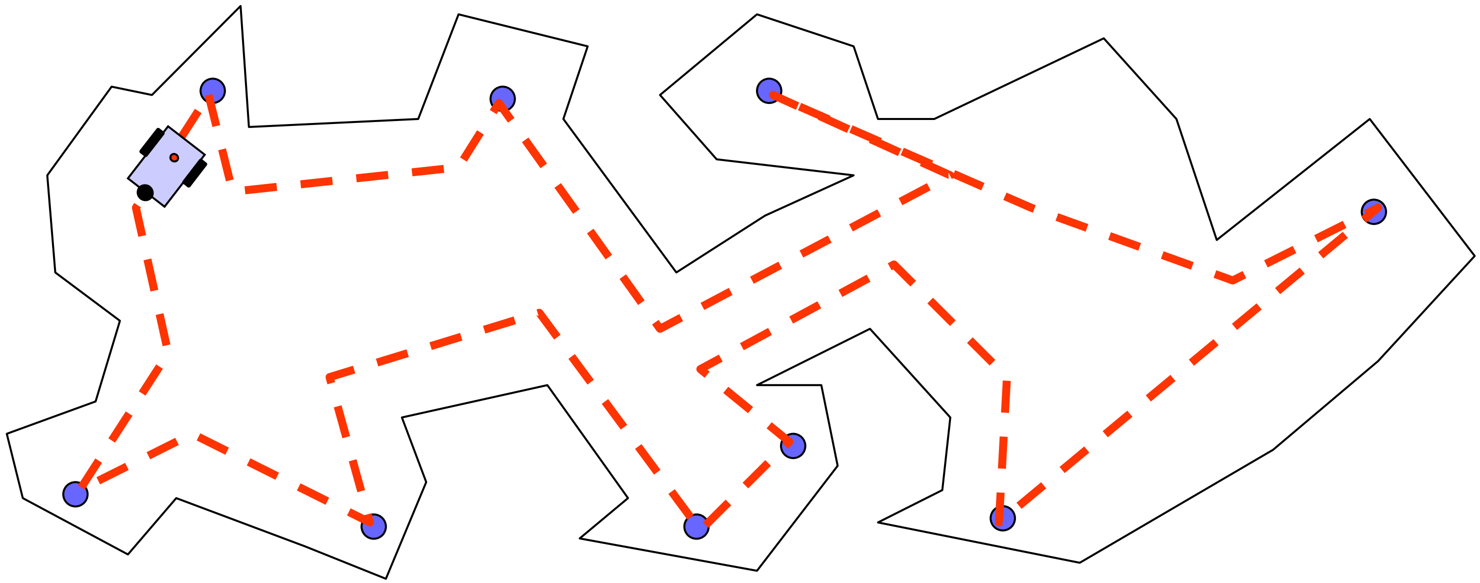
- To perform an exhaustive search, the robot must move around in the environment
  - ***shortest watchman route*** – the shortest possible path in the environment such that the robot covers (i.e., sees) all areas in the environment.
  - difficult to find exact solution, approximations are usually simpler and acceptable

- Can solve this problem by finding guard placement locations and then connect them with an efficient path (i.e., travel between multiple goal locations).

# Traveling Salesman Problem

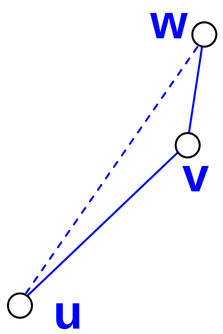
# Traveling Salesman Problem

- Given a number of locations that the robot must travel to, what is the cheapest round-trip route that visits each location once and then returns to the starting location ?
  - (e.g., visiting stations in a building for security checks).



# Traveling Salesman Problem

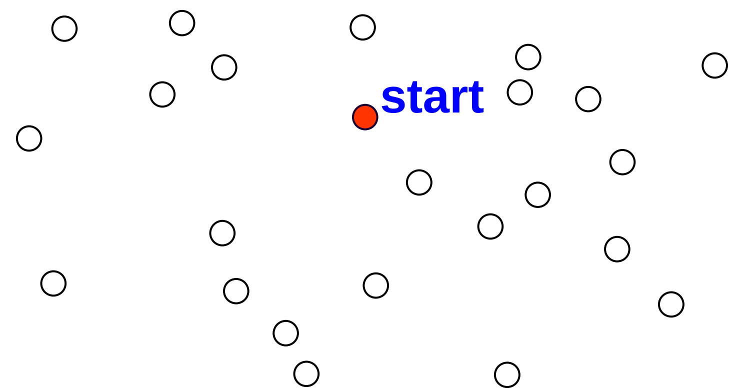
- Most direct solution:
  - try all permutations (ordered combinations) and see which one is cheapest
  - number of permutations is $n!$ for n locations … impractical !!
- There are many approaches to this problem
  - many use heuristics and approximations



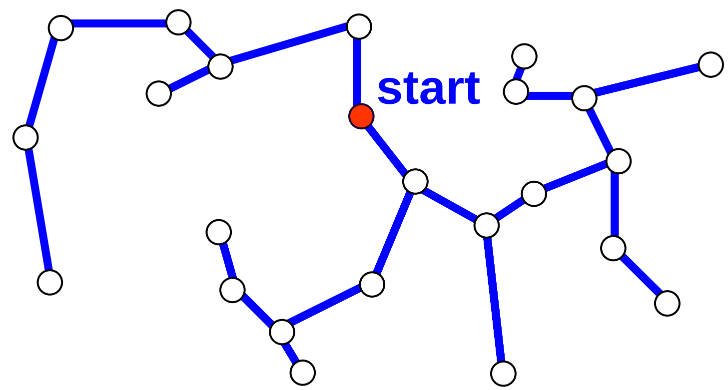

- If we don't need the "optimal" path, we can compromise for some simpler algorithms.
- Assume triangle inequality holds: $|\overline{uw}| \leq |\overline{uv}| + |\overline{vw}|$

# Traveling Salesman Problem

- Consider the locations that robot must travel to.




- Approximate tour is based on ***minimum spanning tree*** from the start location:




Use well-known **Prim's algorithm** or other favorite.

# Traveling Salesman Problem

- Consider a complete graph of the locations
  - i.e., each location connects to every other location
- The minimum spanning tree is a subset of the complete graph's edges that forms a tree that includes every location, where the total length of all the edges in the tree is minimized.

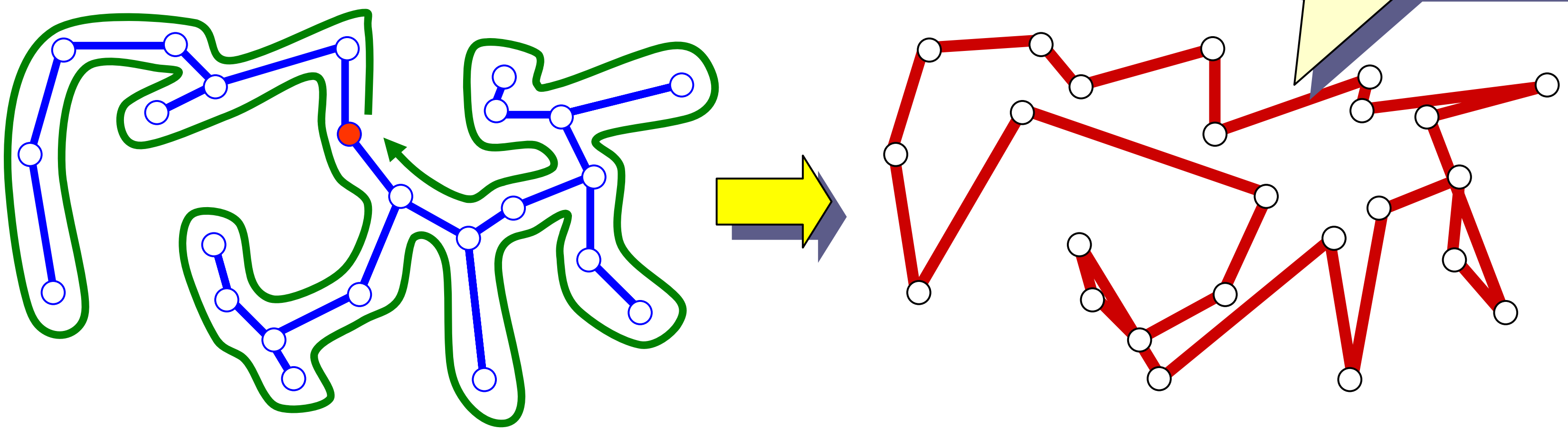
1. Create a tree containing a single arbitrarily chosen location
2. Create a set **S** containing all the edges in the graph
3. WHILE (any edge in **S** does not connect two locations in the tree) DO
4. Remove the shortest edge from **S** that connects a location in the tree with a location not in the tree
5. Add that edge to the tree

Use simple heap data structure.

# Traveling Salesman Problem

- From the root of the minimum spanning tree perform a pre-order traversal of the tree.
- Connect nodes in order visited.

Solution can be at most twice the best path … but is usually not so bad.

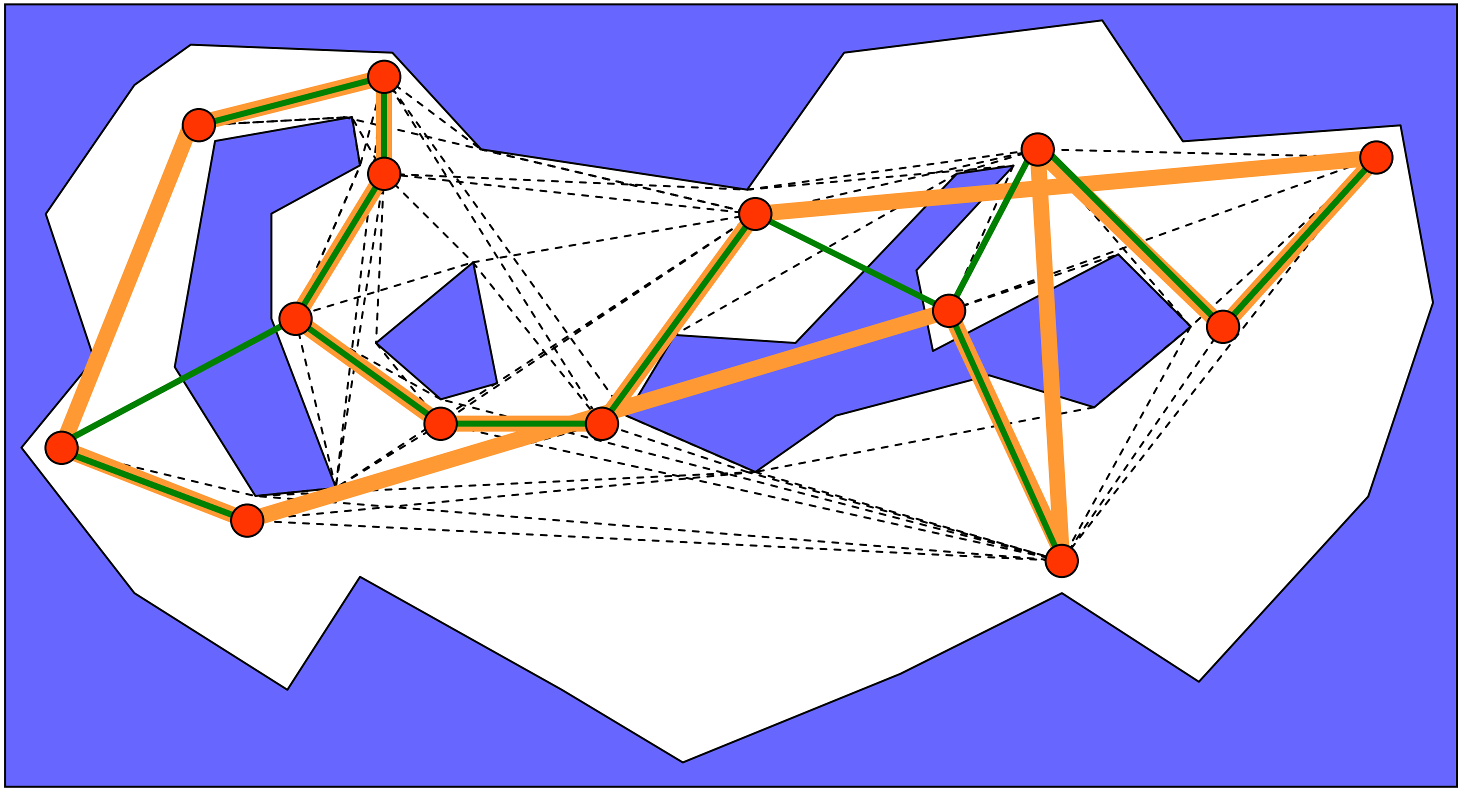


# Traveling Salesman Problem

- Running time is $O(n^2)$ for $n$ locations.
- There are other variations of this problem … we could spend a whole course discussing these types of problems.
- Can we use this algorithm practically ?
  - the triangle inequality may not hold since obstacles are often in the way.
  - can still do a minimum spanning tree, but must replace straight line paths with weighted shortest path links.

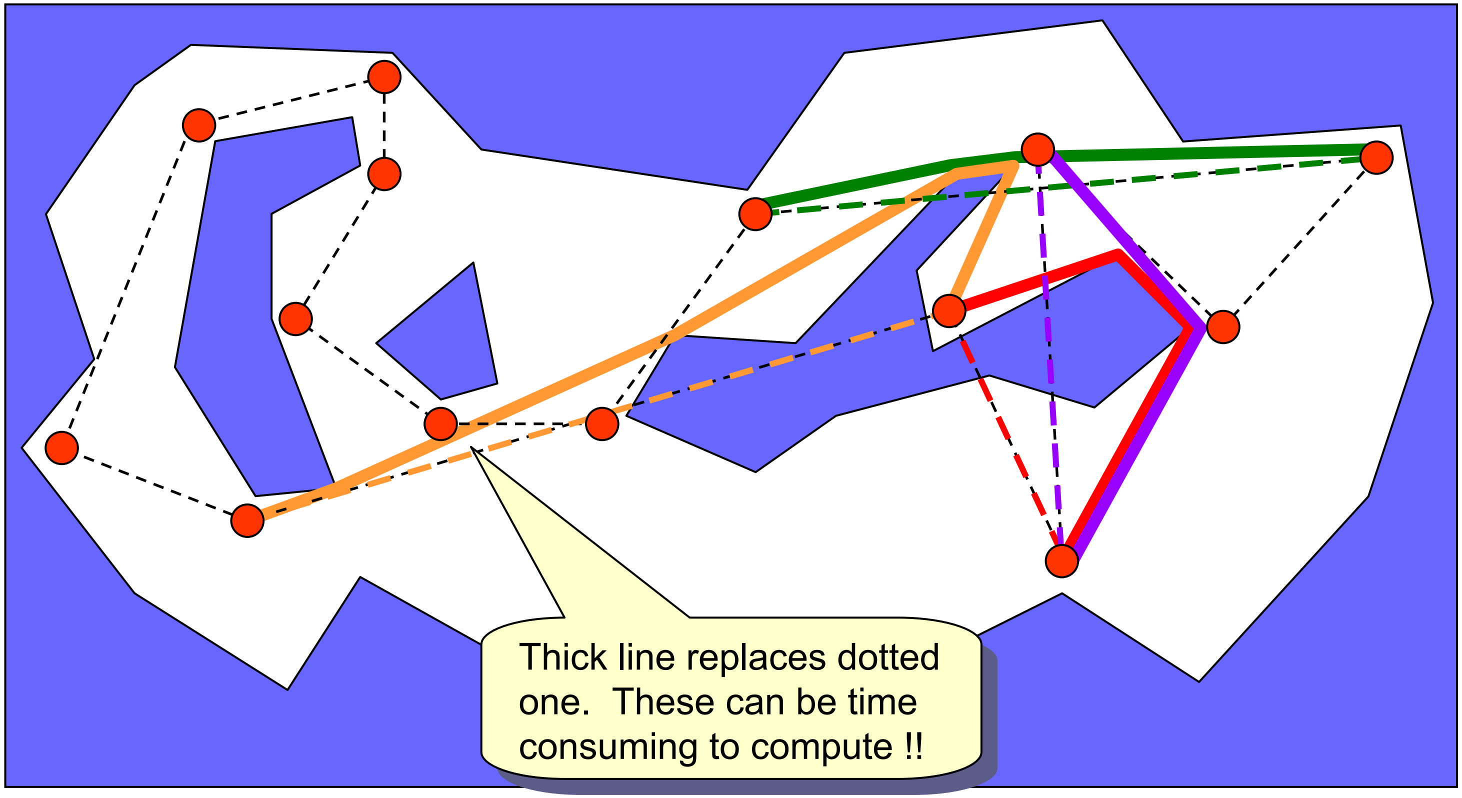
# Traveling Salesman Problem

- Solution to TSP may yield invalid paths.
  - would have to replace point-to-point costs with shortest path costs



# Traveling Salesman Problem

- Can replace invalid segments with shortest path segments:



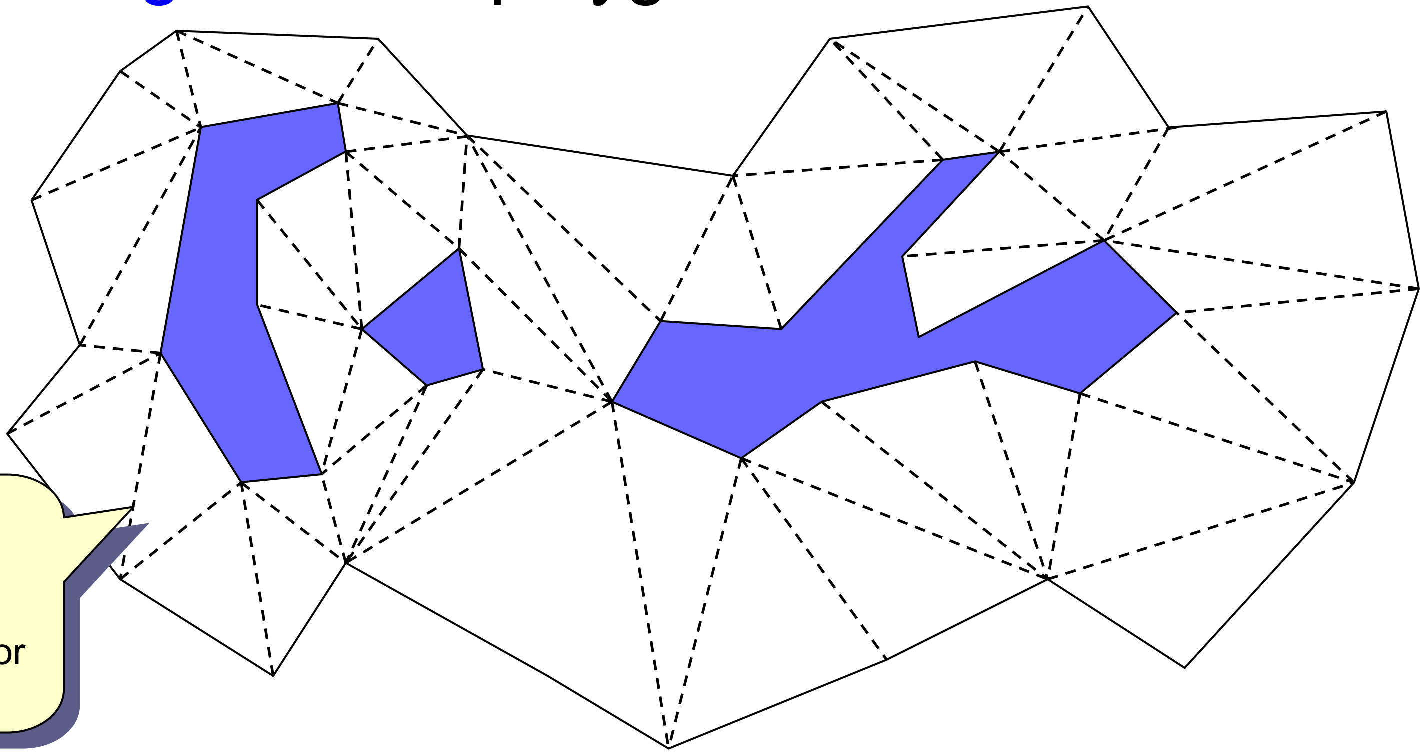
# Traveling Salesman Problem

- The solution to the traveling salesman problem does not directly apply to our problem since paths may be invalid.
- Often a simpler, more practical, approach is a better one.
  - \+ easier to compute
  - \+ can ensure complete coverage
  - \- may end up with longer path
- Simplest, most practical approach is to use the dual graph of the triangulation.

# Visibility Search Paths

# Dual Paths

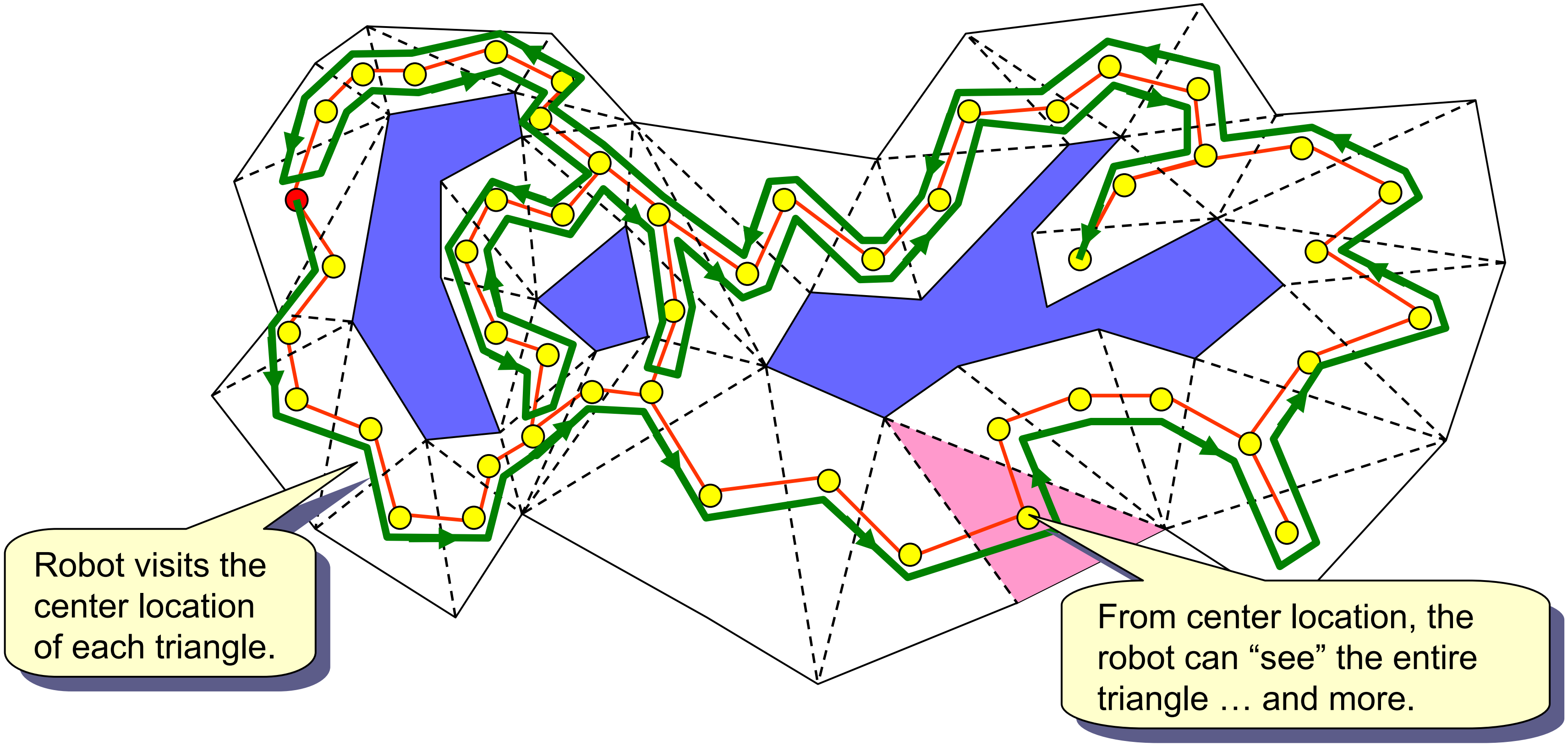
- Consider a robot with an unlimited range, omni-directional sensor.
- First, triangulate the polygon with holes:



Use a constrained Delauney Triangulation for best results.

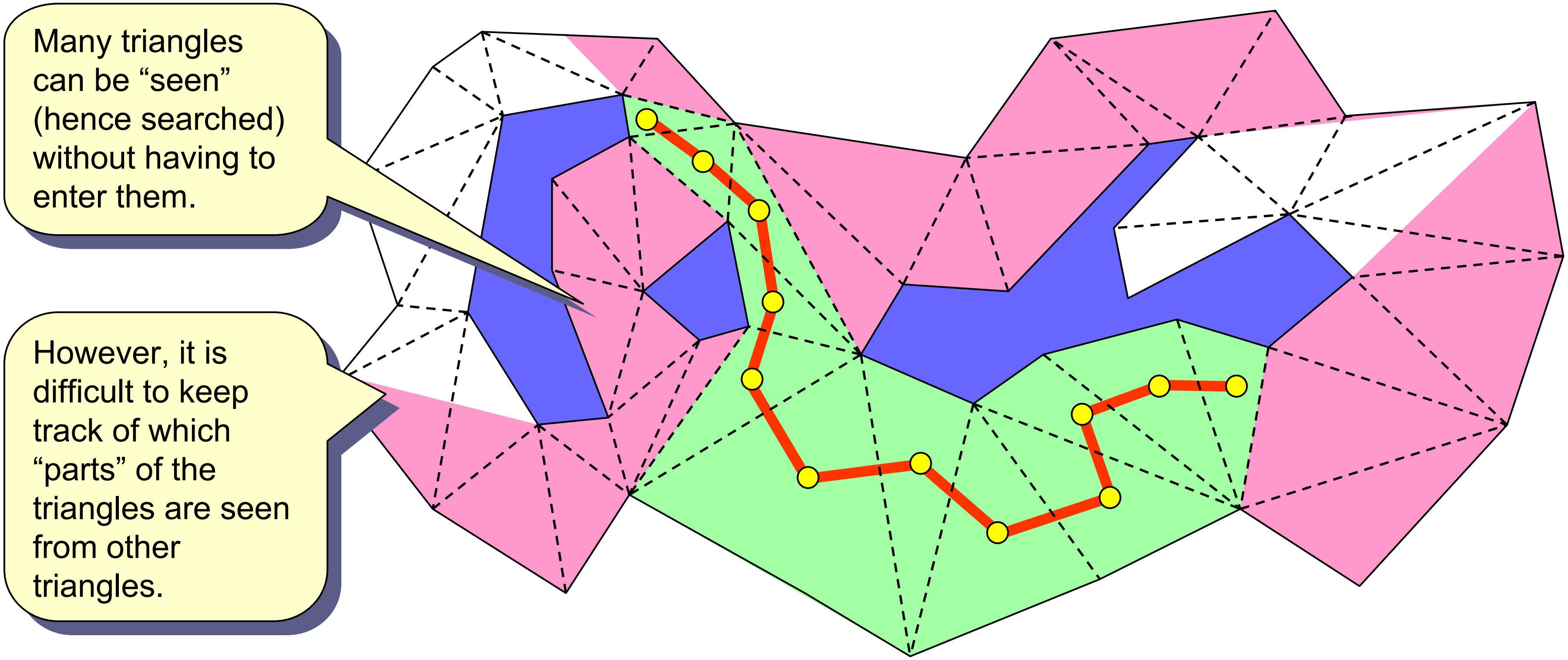
# Dual Paths

- We can traverse the dual graph (using a Depth First Search) as a rough path around all obstacles:



# Visibility Path

- As the robot travels along the dual graph, it can actually "see" (i.e., search) a much larger area than the triangles it passes through:

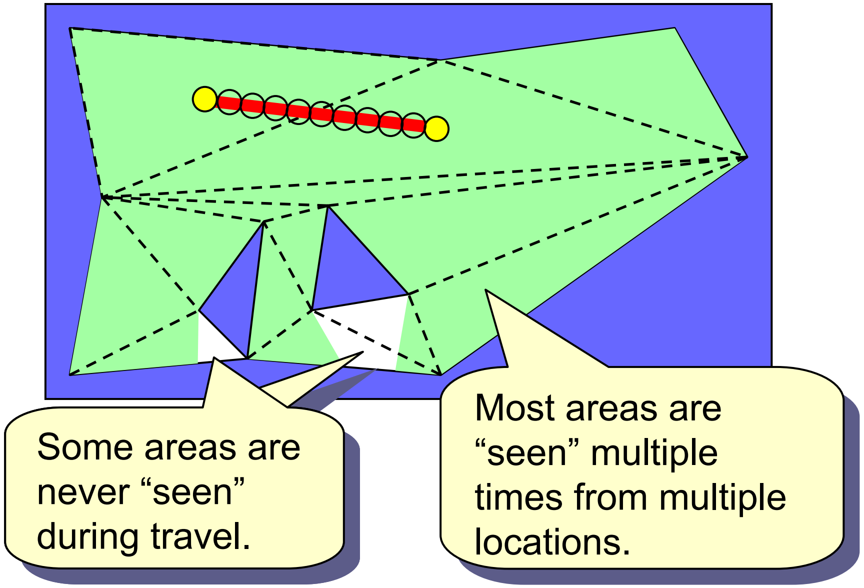


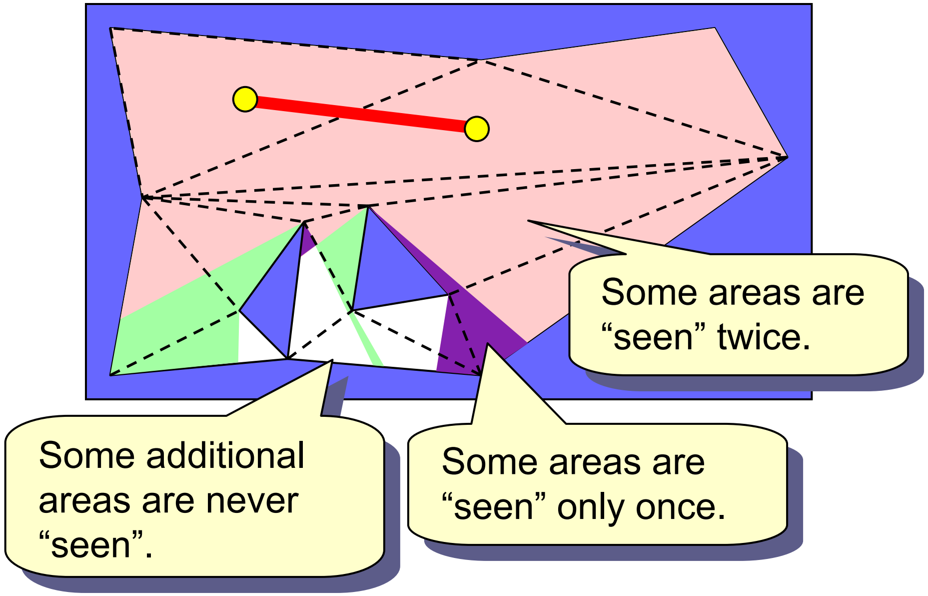
# Visibility Path

- When robot travels between triangles it can:

  - search along the way while traveling

  - search only when arriving at the triangle centers



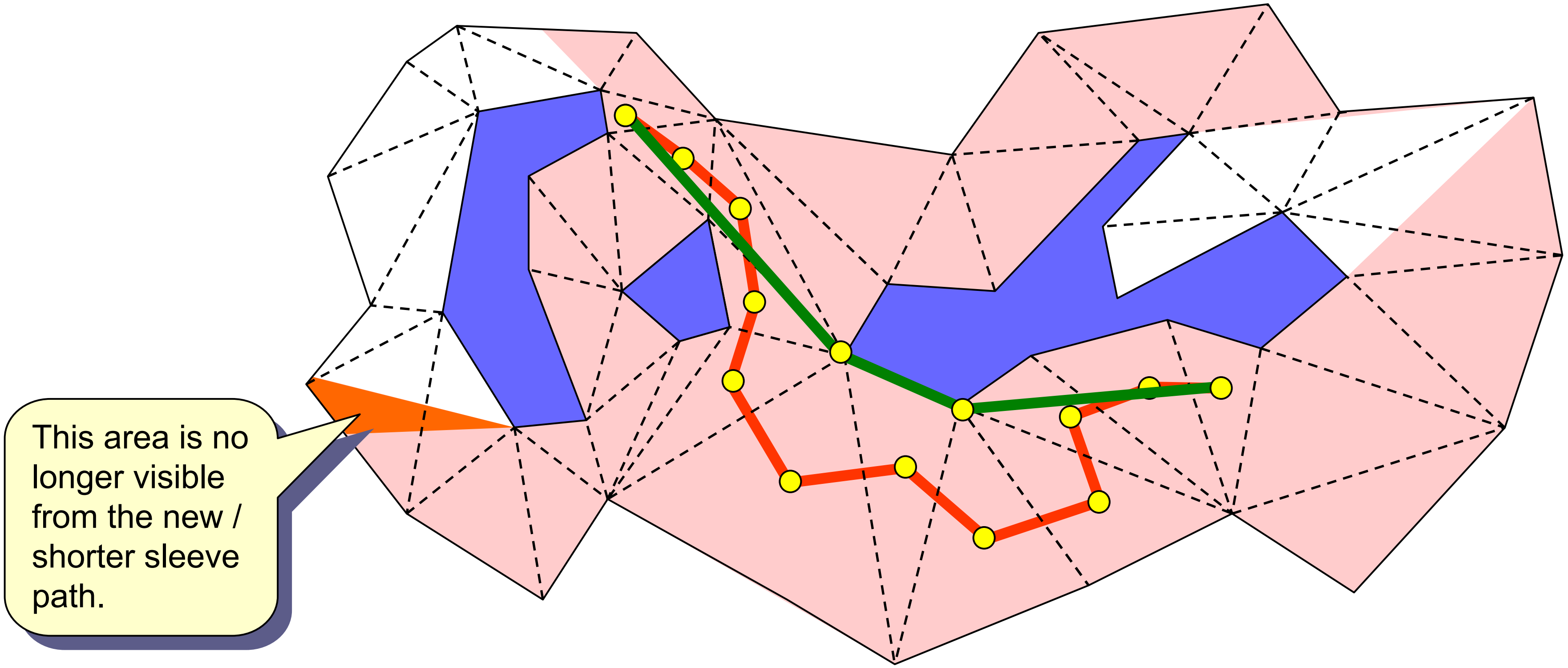




+ better coverage
- more computation

+ less computation
- less coverage

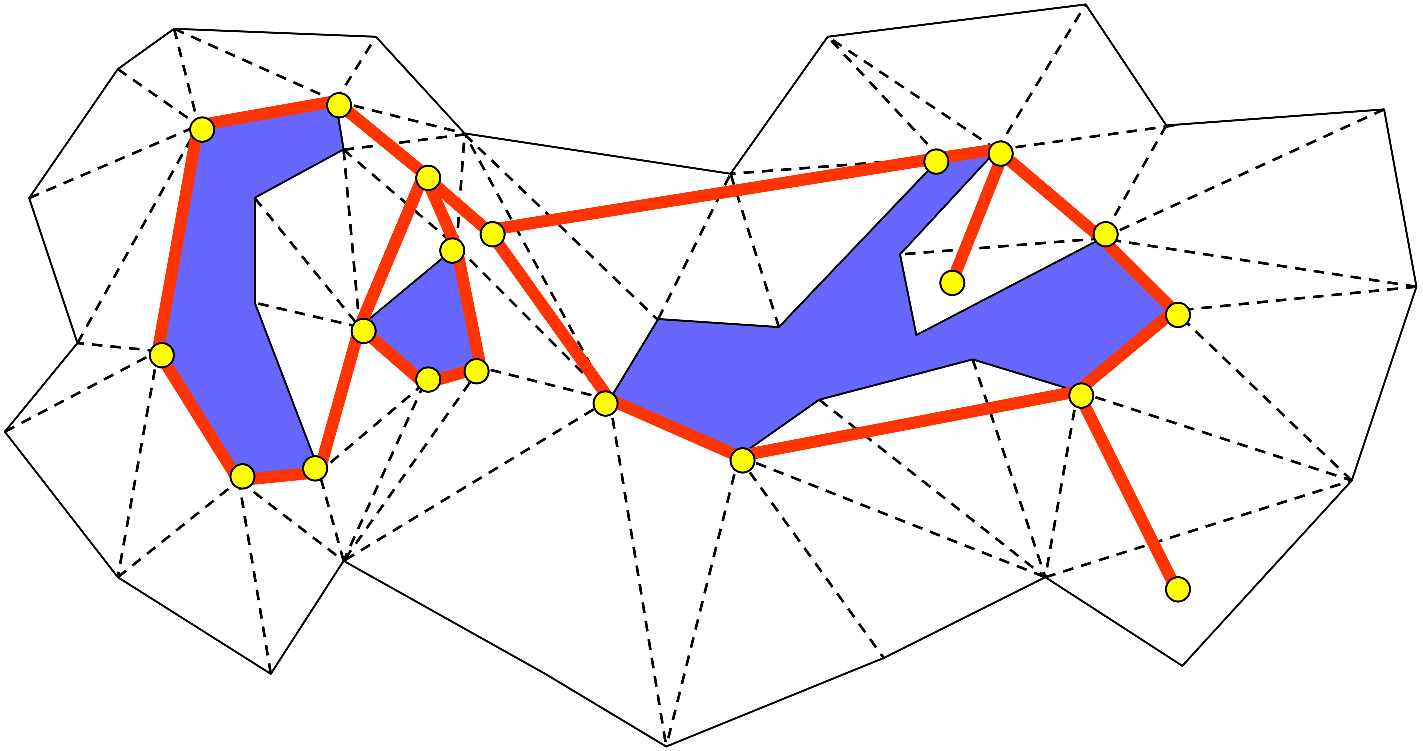
# Refining Paths

- Recall that dual graph paths can be refined by computing a shortest sleeve path:
  - results in a slightly modified area coverage



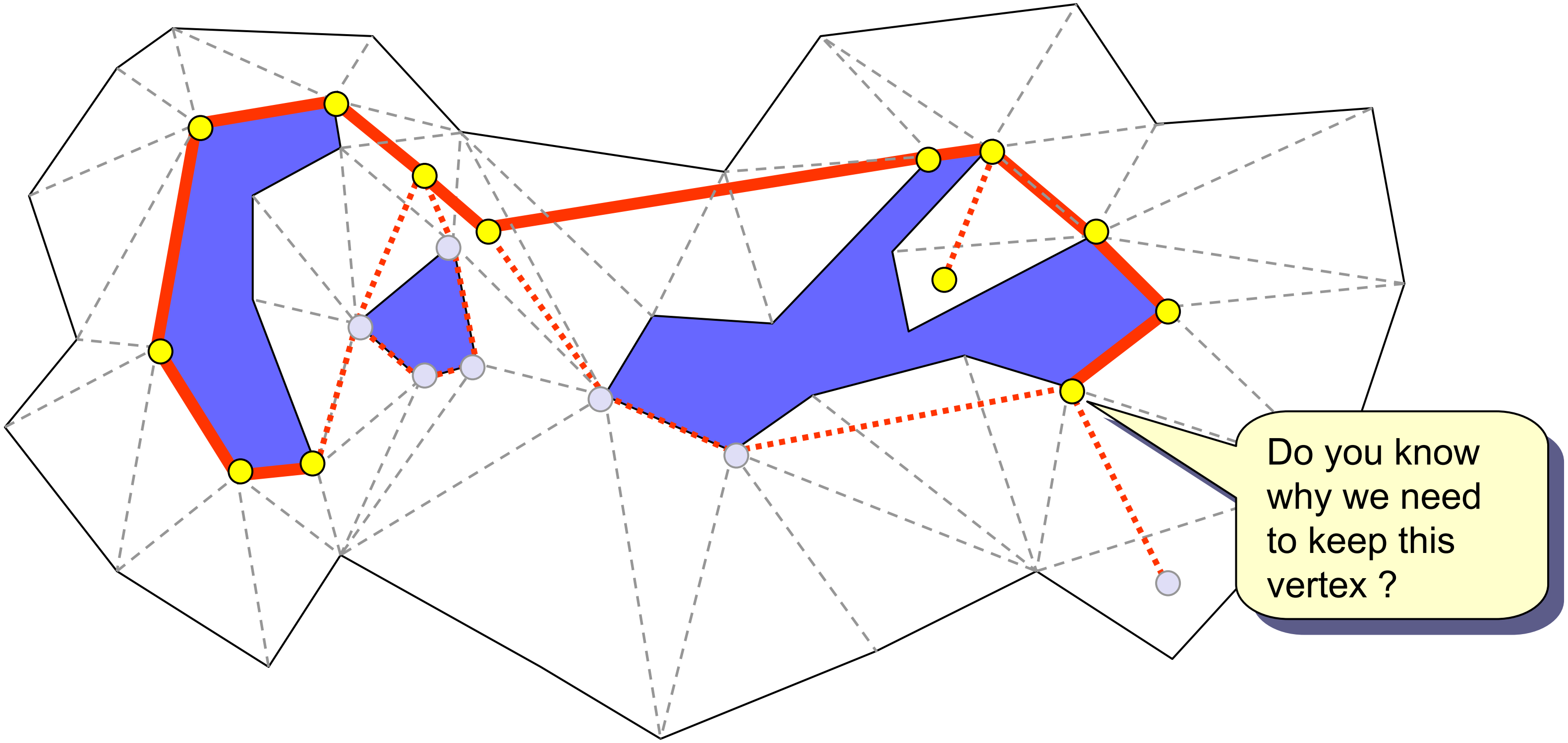
# Refining Paths

- Combining all such ***refined paths*** leads to an efficient path that will guarantee visibility of the entire environment:



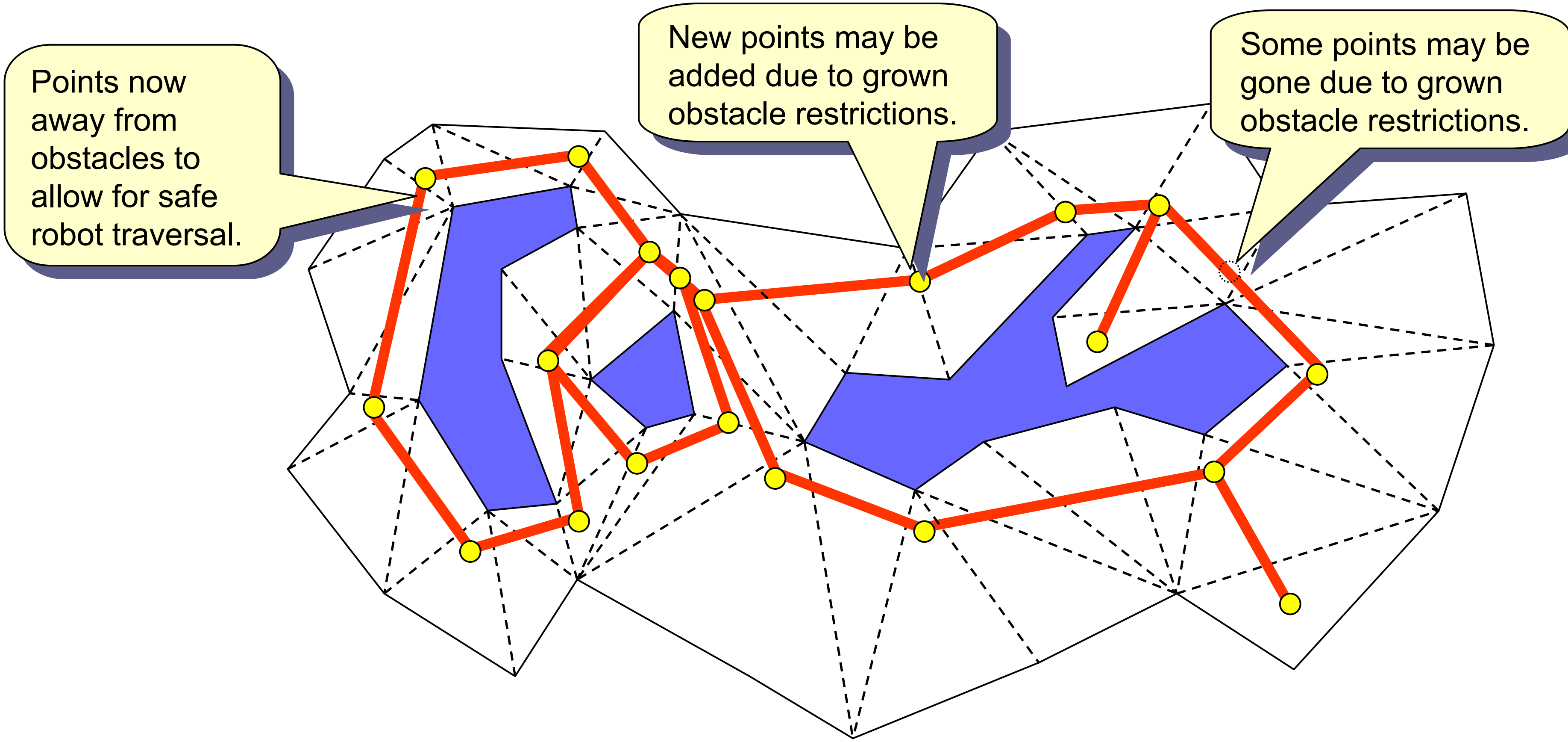
# Refining Paths

- Can even ***trim*** (i.e., remove edges from) the path:
  - walk through the path, keeping track of which triangles are completely covered along the way. Eliminate edges/vertices that do not add to the path's coverage.



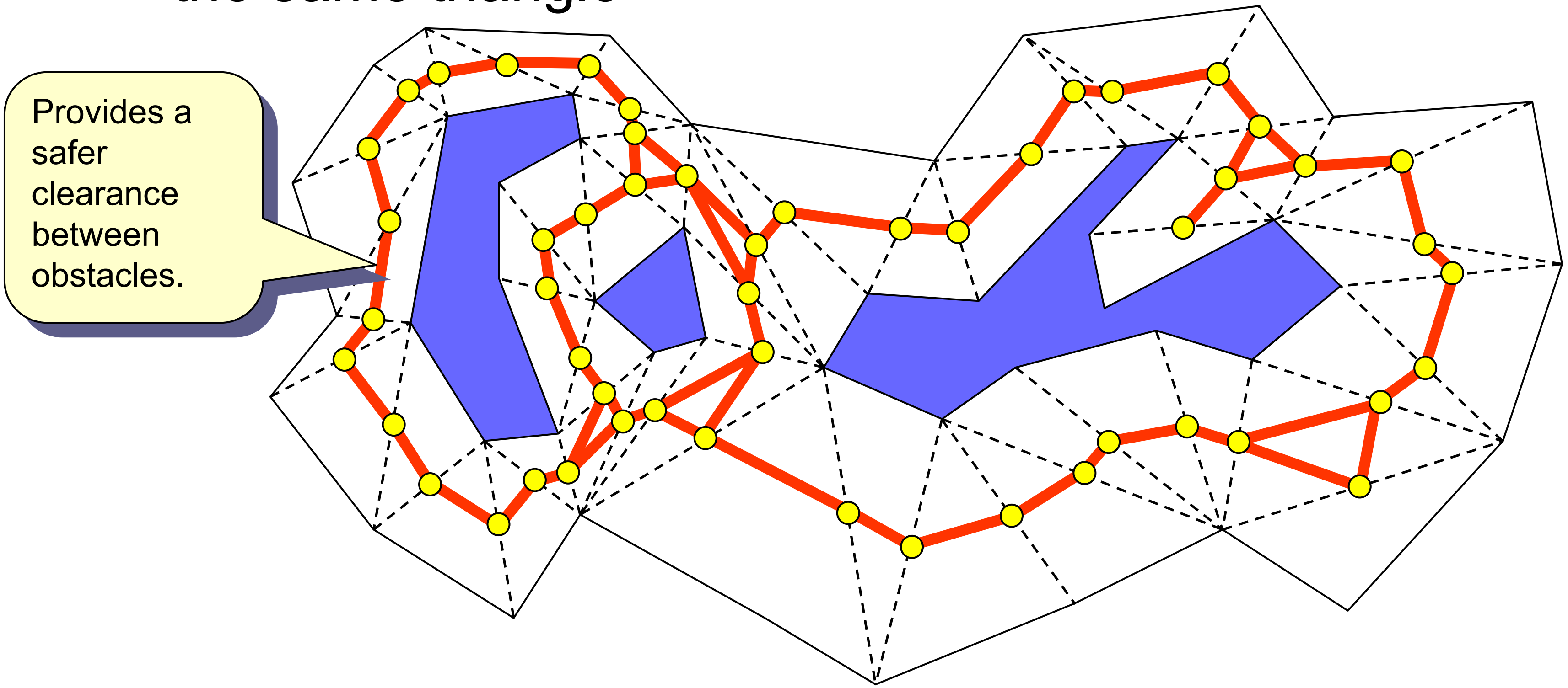
# Safer Paths

- For safety, we can first grow obstacles according to robot model to allow valid paths that do not collide:



# Safer Paths

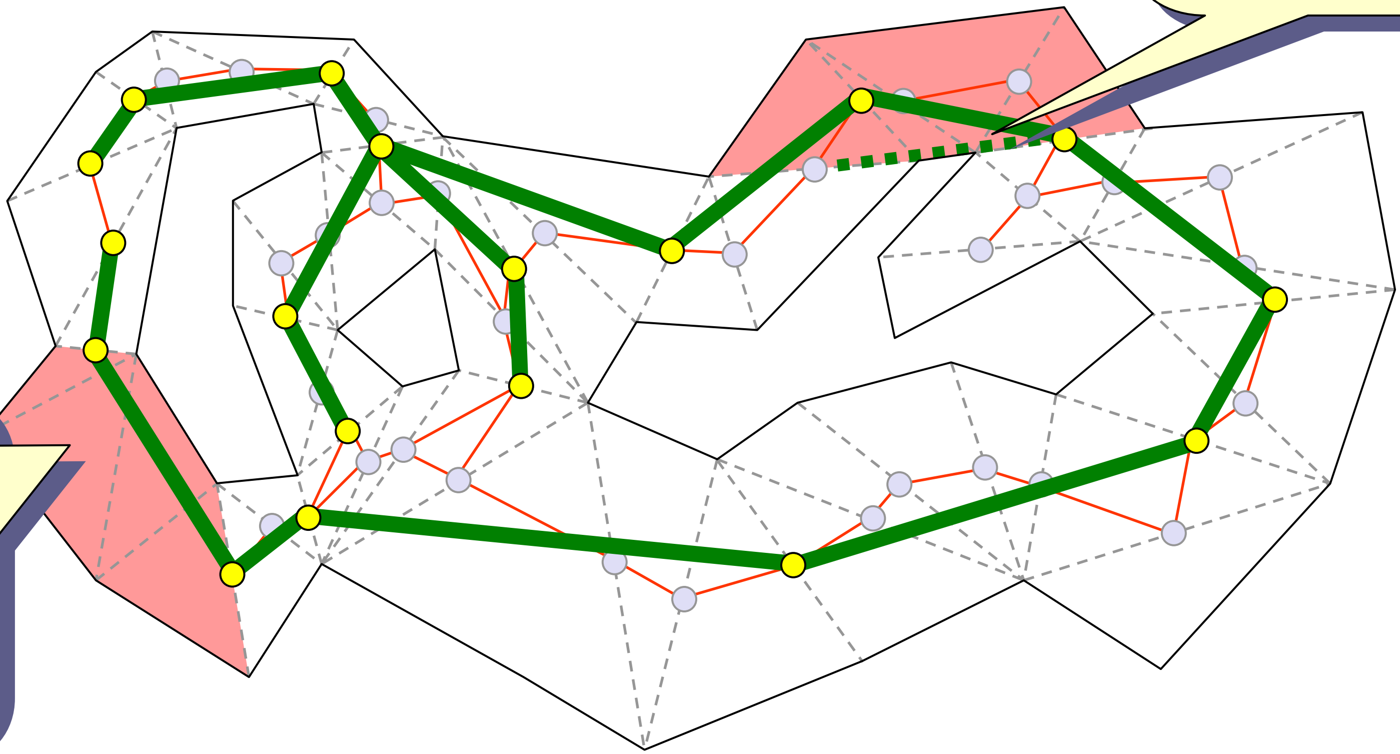
- A safer/simpler approach:
  - place view locations at midpoints of triangulation diagonal edges and connect viewpoints from edges on the same triangle



# Safer Paths

- Once again, trim edges by removing ones that do not add to the coverage:

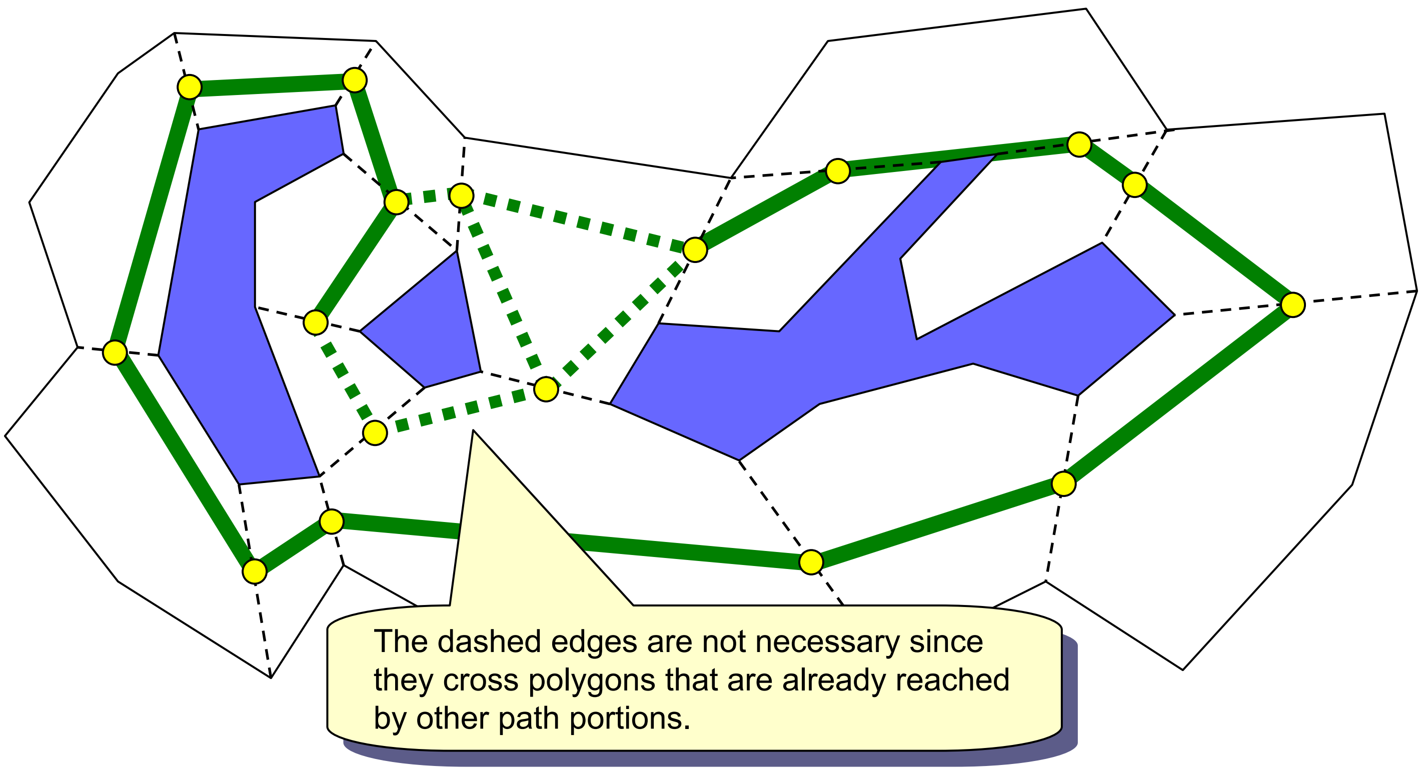
May also put constraint that edge must have certain clearance from obstacles.



Merge triangles that form convex polygons since entire convex polygon is visible from any point inside it.

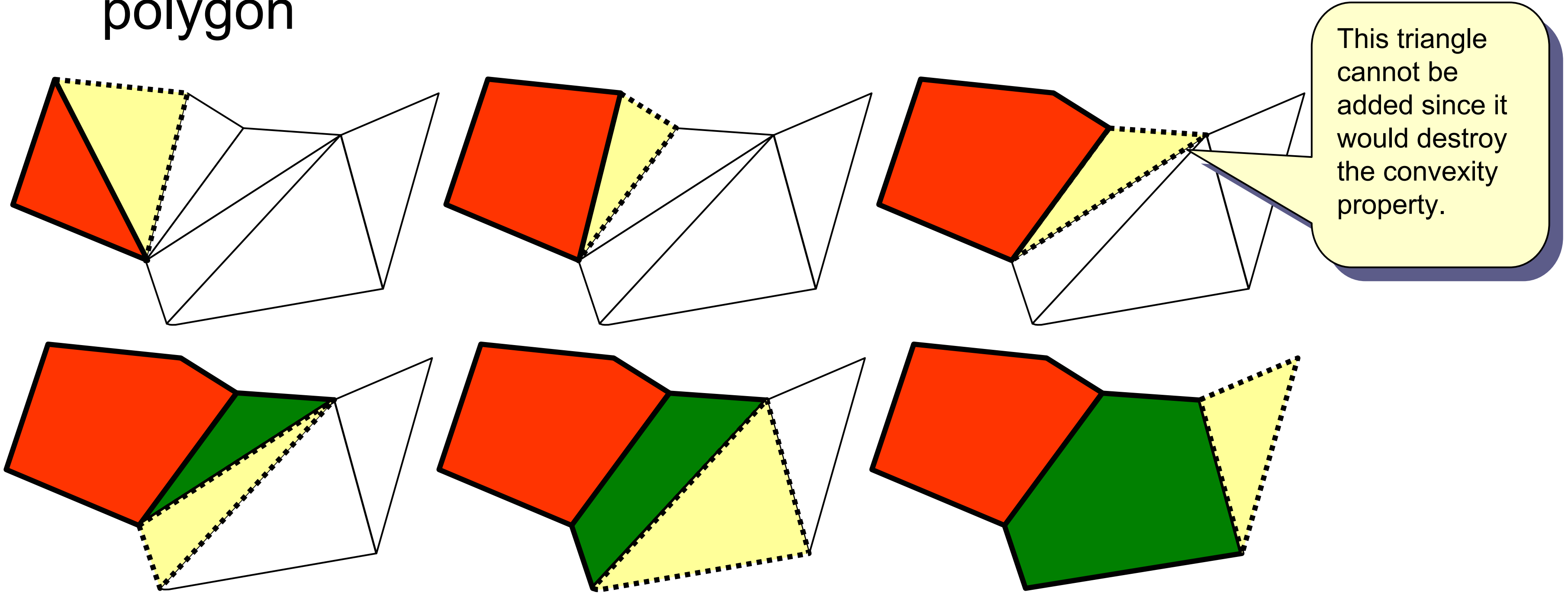
# Convex Pieces

- Of course, we can always merge triangles to form convex areas before we search the graph
  - reduces need to trim off many edges later



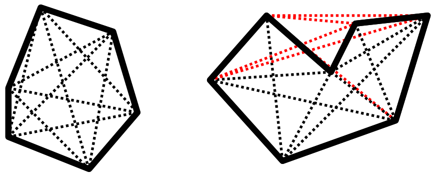
# Convex Pieces

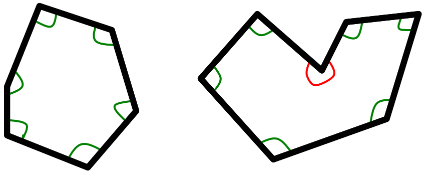
- How do we merge triangles into convex pieces ?
  - traverse dual graph using DFS.
  - build up convex polygon by adding new triangles one at a time … if a new triangle “ruins” convexity, start a new polygon

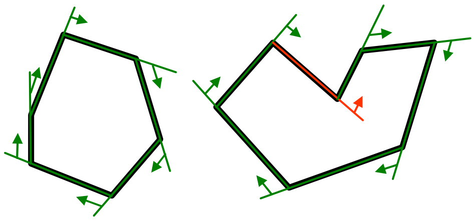


# Convex Pieces

- How do we determine if a polygon is convex ?
- There are a variety of ways:
  - check that the line segment between each pair of non-adjacent vertices does not intersect any polygon edge.
  - check that each pair of consecutive edges forms an interior angle ≤180°.
  - traverse the polygon CW and make sure that each consecutive edge makes a right turn.

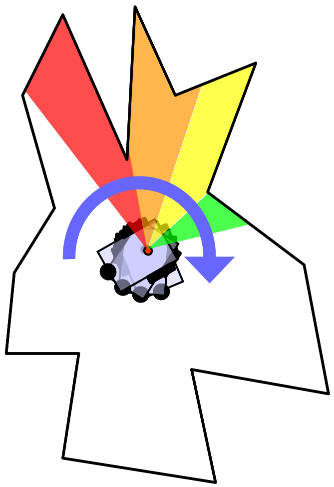




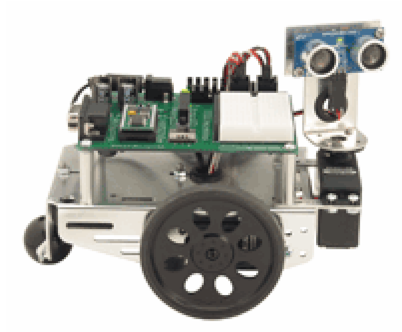


# Limited Direction Visibility

- What if the robot cannot sense omni-directionally ?
- Recall that robot can turn at each search point:
  - can be time consuming
  - try to minimize search locations



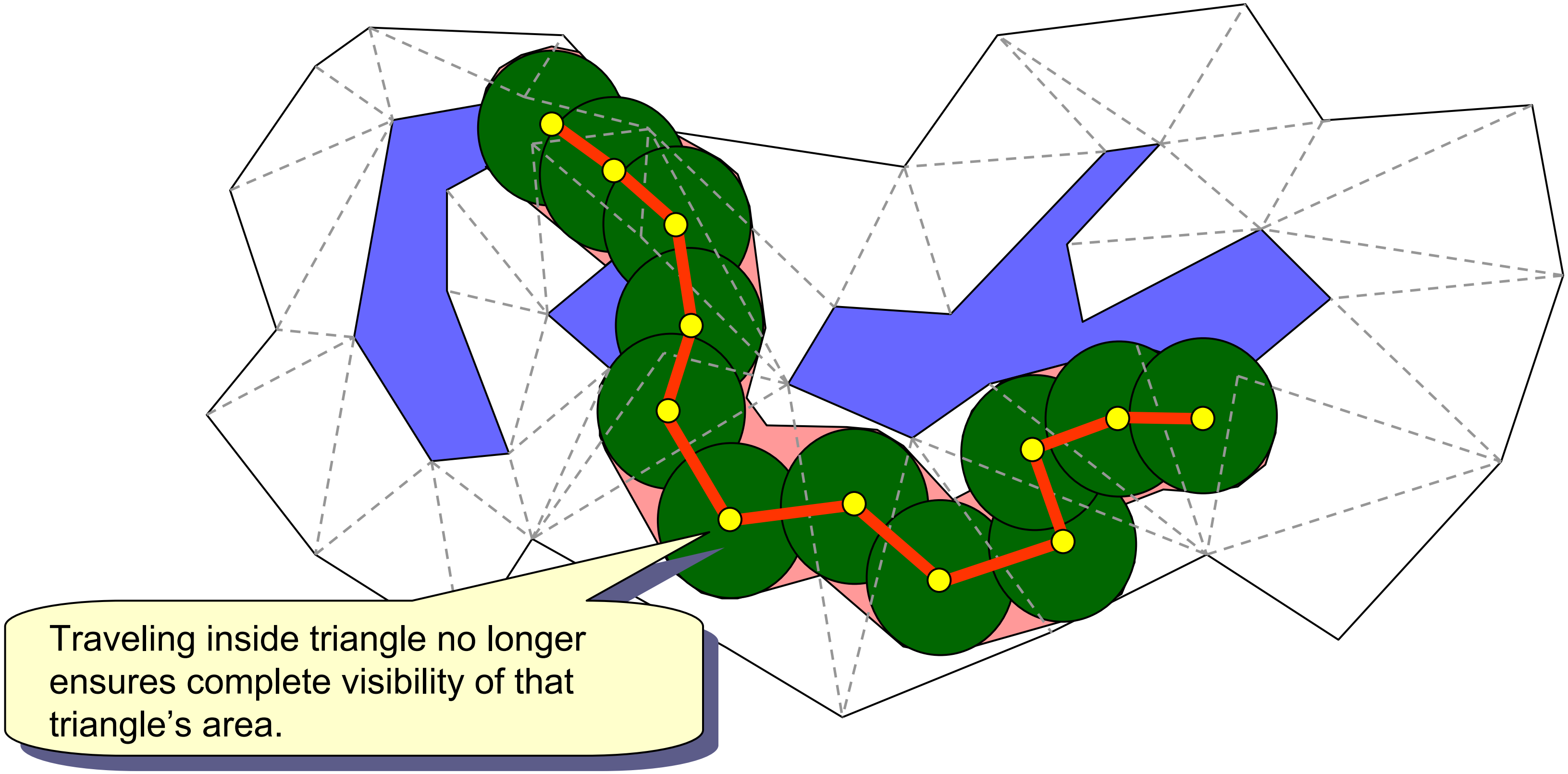
- Alternatively, some robots are equipped with ***head turrets*** that can turn 360°.



# Searching With Limited Visibility Range

# Limited Range Visibility

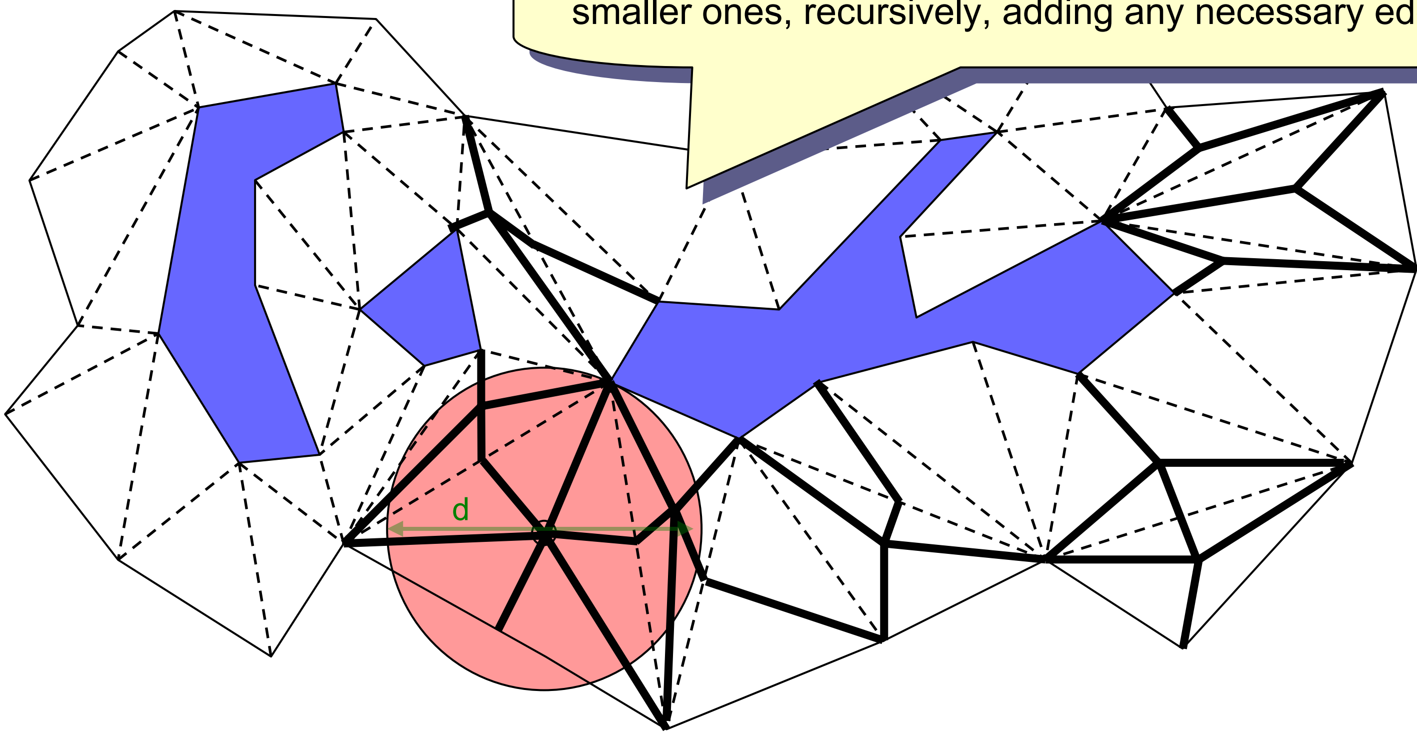
- The problem changes when the robot has limited sensing range:



# Recursively Decompose

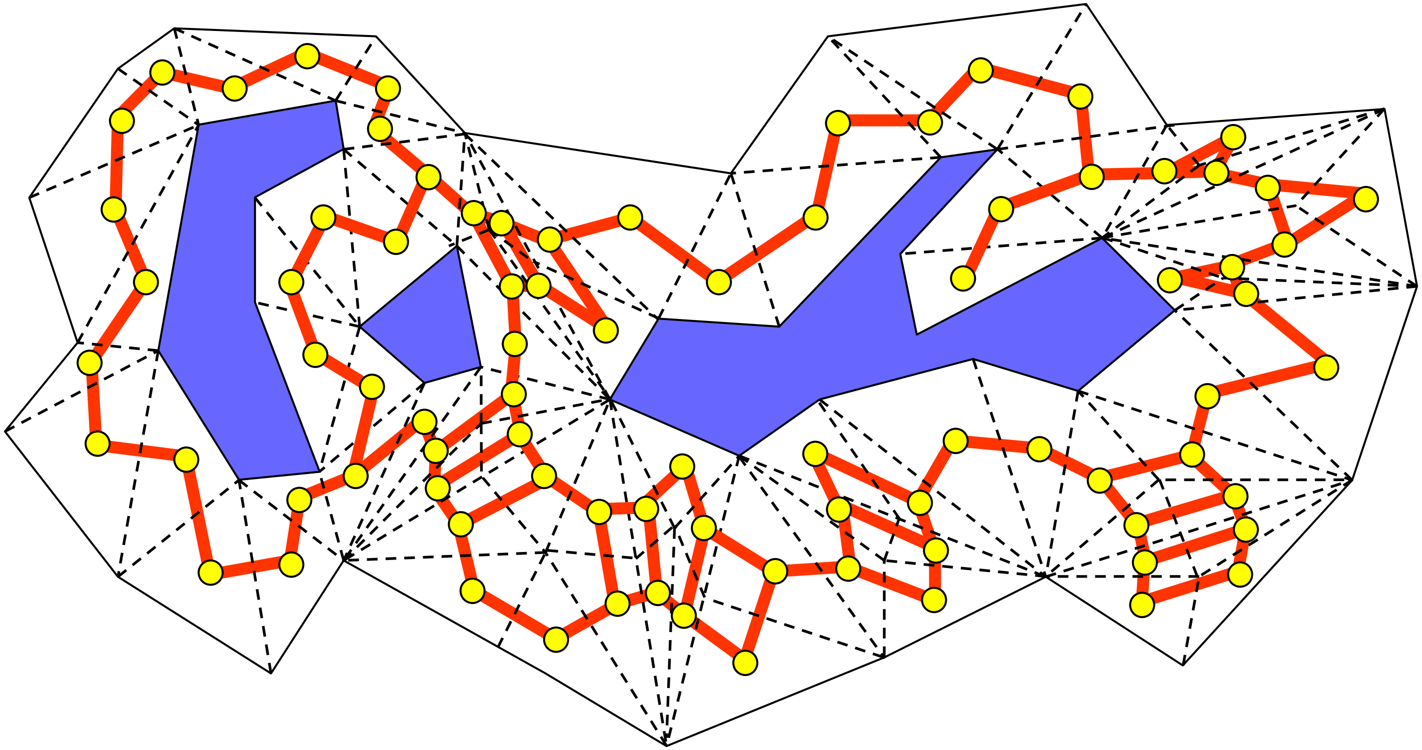
- One option is to ensure that each triangle is small enough to be covered by the robot's range:

For each triangle that is not covered from its center (based on robot's viewing range **d**), split triangle into smaller ones, recursively, adding any necessary edges.



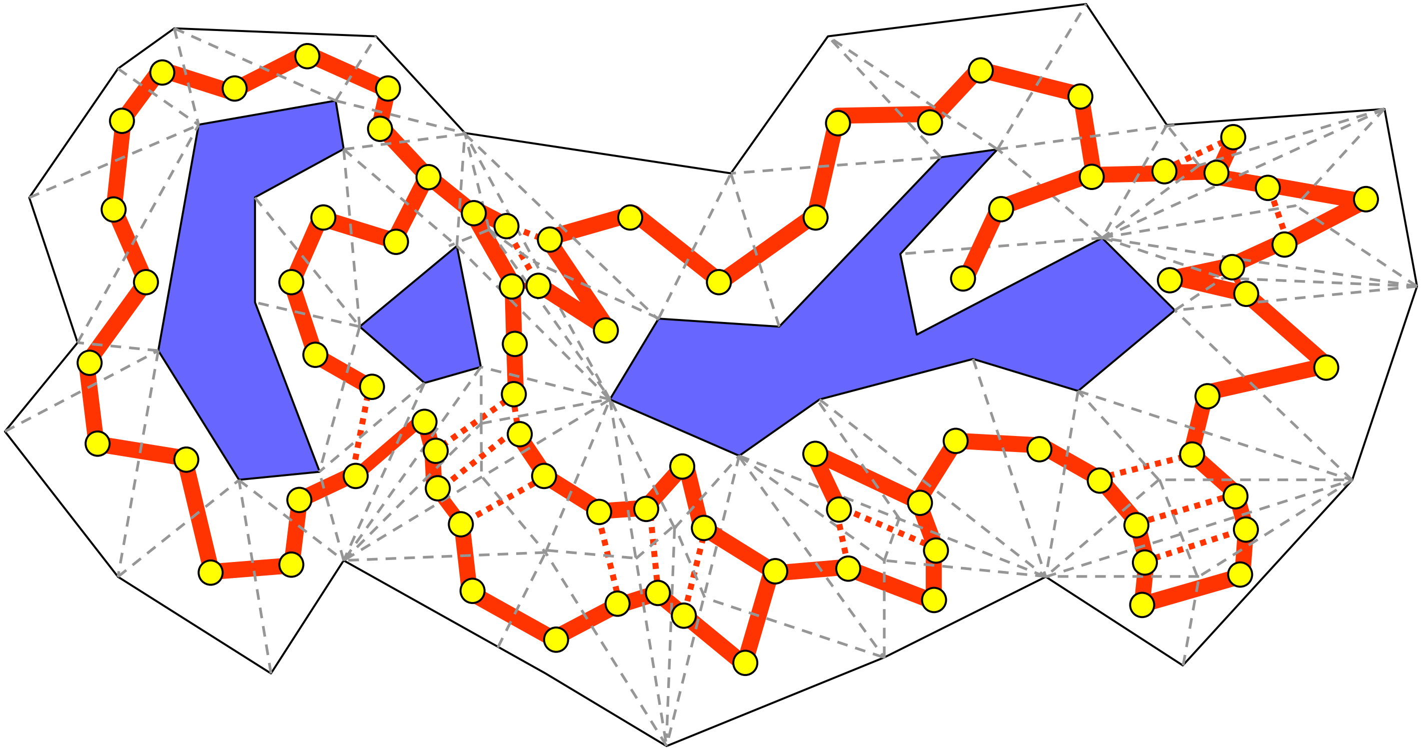
# Limited Range Paths

- Again, form path from dual graph:
  - more loops now
  - cannot trim vertices now, only edges.



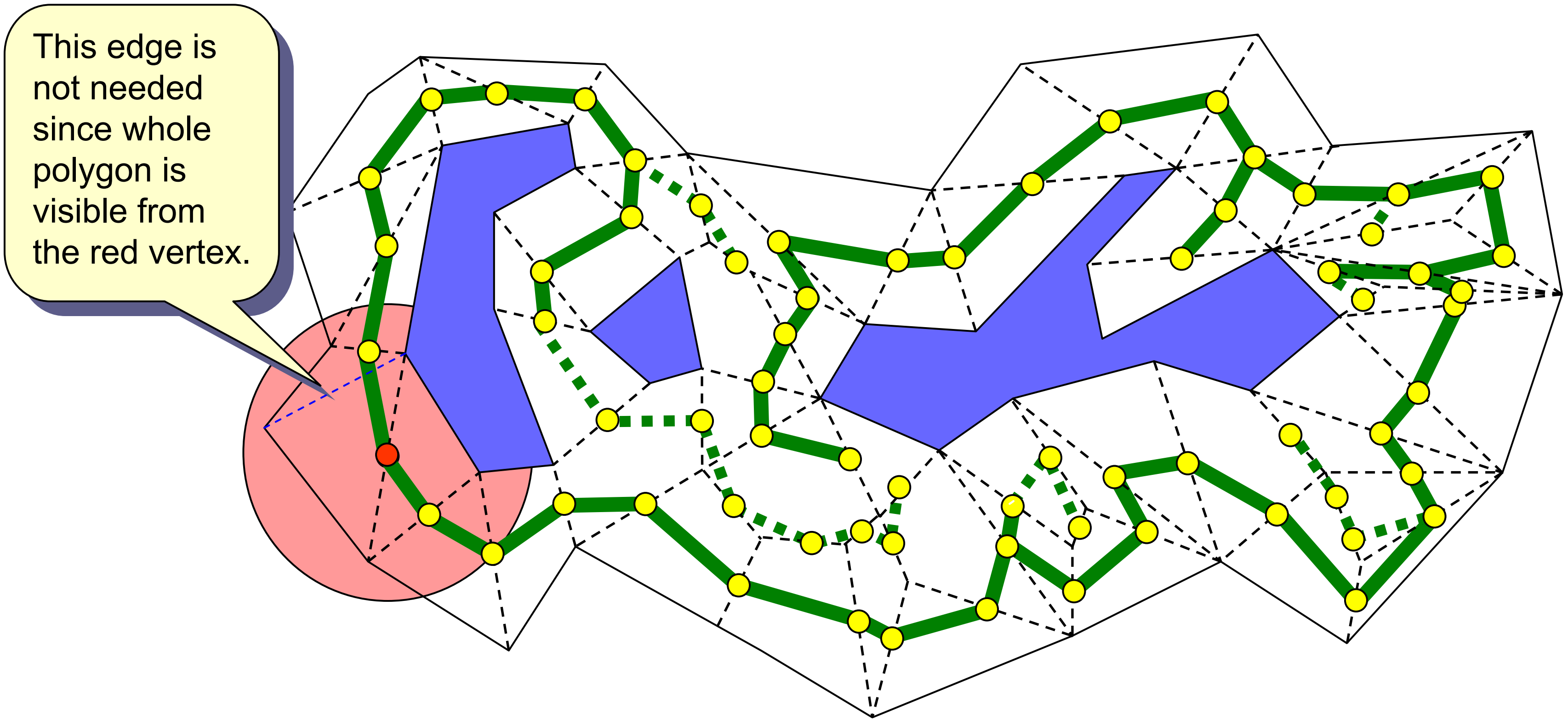
# Refining Limited Range Paths

- May trim as many edges as possible, provided that the removal of the edge does not disconnect the graph.



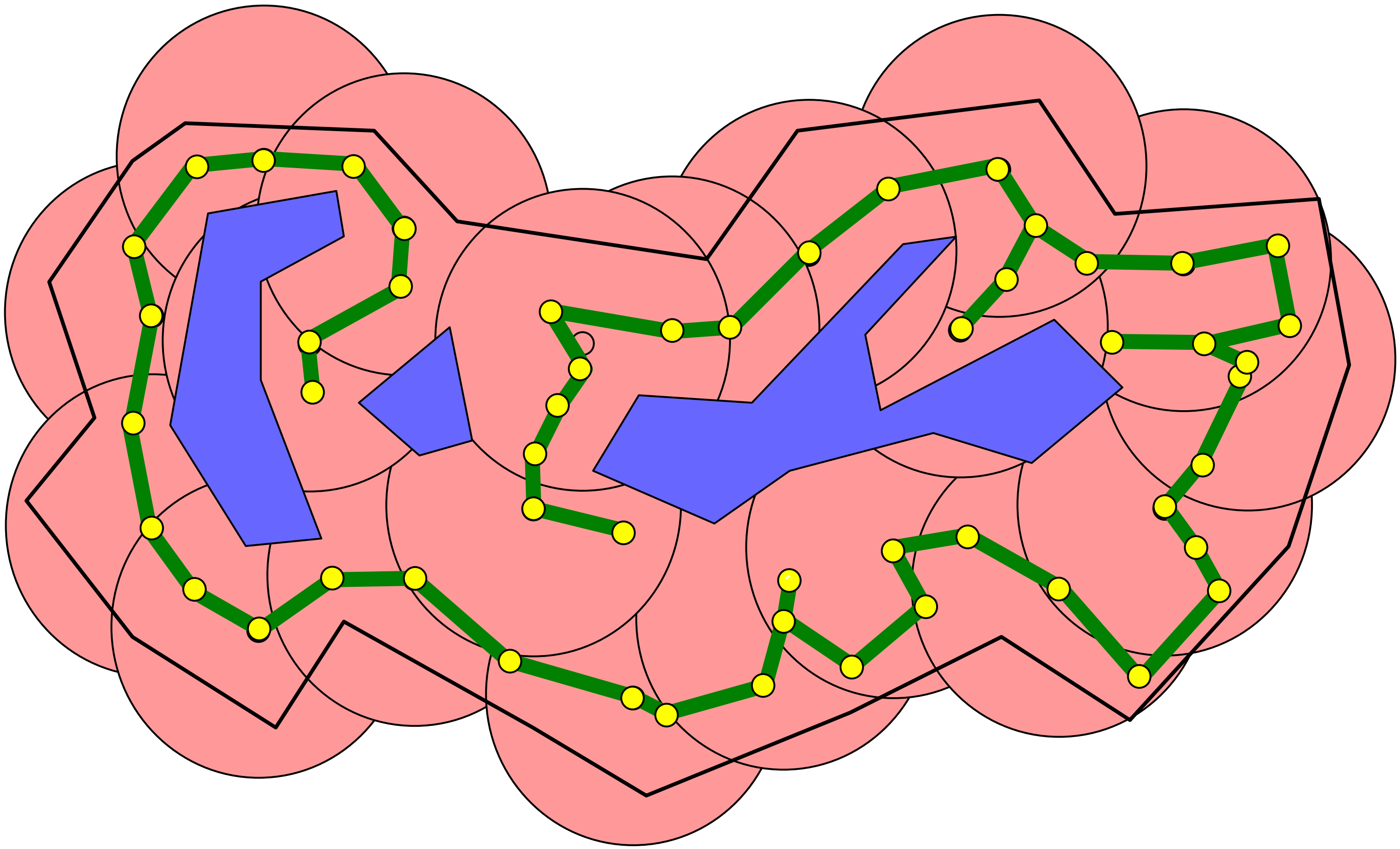
# Convex Pieces

- Again, we can merge into convex polygons first, provided that the convex polygons are fully visible from each edge:



# Limited Range Visibility Coverage

- Result is that entire area is covered:



# Summary

- You should now understand:
  - How to compute paths that cover an environment
  - Different ways of covering an environment
  - How to compute a set of robot locations that see the entire environment
  - A simple way to search an environment with robots that have sensors with unlimited or limited range as well as omni-directional or limited direction.