Goal-Directed Navigation

Chapter 6

Objectives

- Investigate techniques for navigating a robot towards a goal location
- Examine algorithms that work in environments with
  - known obstacle locations
  - unknown obstacle locations
- Understand various ways of representing maps
  - Investigate 1 way to navigate using feature-based maps
  - Investigate 1 way to navigate using potential fields
- Understand how robot can make a "best" choice decision based on only local sensor information.

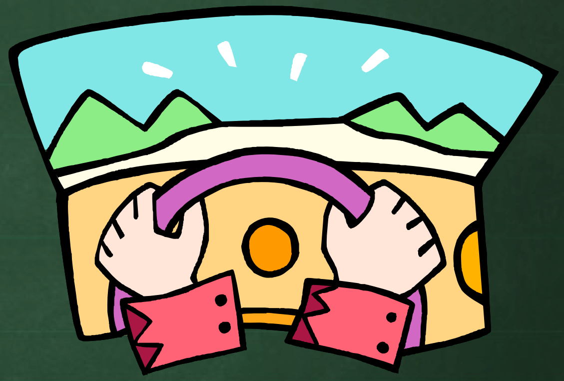
What's in Here ?

- Goal Directed Navigation
  - Navigation and Path Planning
  - Goal-Directed Navigation
- Navigation in Unknown Environments
  - Bug Algorithms (Bug1, Bug2, Tangent Bug)
  - Vector Field Histograms
- Navigation in Known Environments
  - Map Representation and Storage
  - Map Accuracy and Map Hierarchy
  - Feature-Based Maps
  - Feature Map Navigation
    - Creating Feature Maps
    - Navigating Feature Maps
  - Potential Field Navigation
    - Potential Fields
    - Corridor Fields

Goal-Directed Navigation

Navigation

- In robotics, ***navigation*** is the act of moving a robot from one place to another in a collision-free path.



- When navigating, robots either:
  - *head towards goal* location(s), or
  - follow a *fixed path* (known in advance)
- When heading toward goal, robot usually relies on local sensor information and updates its location/direction according to the "best" choice that will lead to the goal.

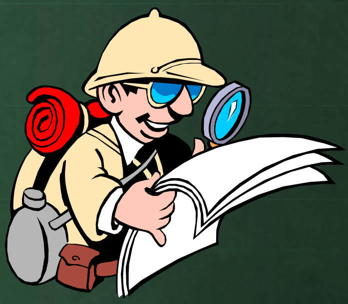
Path Planning

- When a fixed path is provided on which to navigate, the path is usually computed (i.e., planned) beforehand.

- *Path planning* is the act of examining known information about the environment and computing a path that satisfies one or more conditions:
  - avoids obstacles, shortest, least turns, safest etc…
- Key to path planning is efficiency
  - in real robots, optimal solution is not always practical
  - approximate solutions are often sufficient and desired.

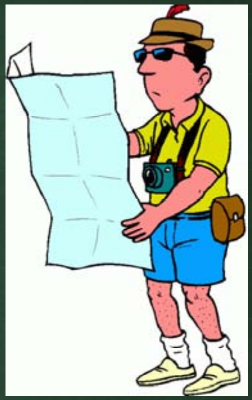
Path Planning

- To accomplish complicated tasks, a mobile robot usually MUST pre-plan it's paths.
- Many interesting problems are solved that make use of planned motion of the robot:
  - Efficient collision-free travel (e.g., shortest paths)
  - Environment coverage (e.g., painting, cleaning)
  - Guarding and routing (e.g., security monitoring)
  - Completion of various tasks etc…



- We will look first at ***goal-directed*** navigation in which the robot is trying to reach a goal location.

Goal-Directed Navigation



- Approaches to goal-directed navigation vary depending on two important questions:
  - Are robot & goal locations (i.e., coordinates) known ?
    - if available, goal position would be given as a coordinate, otherwise the problem becomes one of searching.
    - robot would either maintain its own location as it moves (e.g., dead reckoning which is inaccurate) or have this information provided externally (e.g., GPS system).
  - Are obstacles (i.e., locations and shape) known ?
    - if available, coordinates of all polygonal obstacle vertices would be given and known to the robot.
    - if unavailable, robot must be able to sense obstacles (sensing is prone to error and inaccuracies).

Goal-Directed Navigation

- Here is a summary of algorithms for navigating towards a goal location under various conditions:
  - **Goal Position Unknown**
    - Obstacles Unknown
      1. Behaviors (wandering, light seeking, wall following etc…) } Already discussed
    - Obstacles Known
      1. Search algorithms
      2. Coverage algorithms } Discussed later
  - **Goal Position Known**
    - Obstacles Unknown (i.e., local sensing information only)
      1. Reactive Navigation
    - Obstacles Known (i.e., global information available)
      1. Feature-Based Navigation
      2. Potential Field Navigation } Discussed here
      3. Roadmap-Based Planning } Discussed later

Navigation in
Unknown Environments

Four Strategies

- We will consider 4 strategies for navigating to a goal location when obstacle locations are unknown.
  - Robot must be able to sense obstacles within its vicinity
  - Robot's use a kind of reactive navigation, in that they simply keep moving toward the goal and update their path as they move.
- The strategies are:
  - Bug 1 algorithm
  - Bug 2 algorithm
  - Tangent Bug algorithm
  - Vector Field Histograms (VFH)

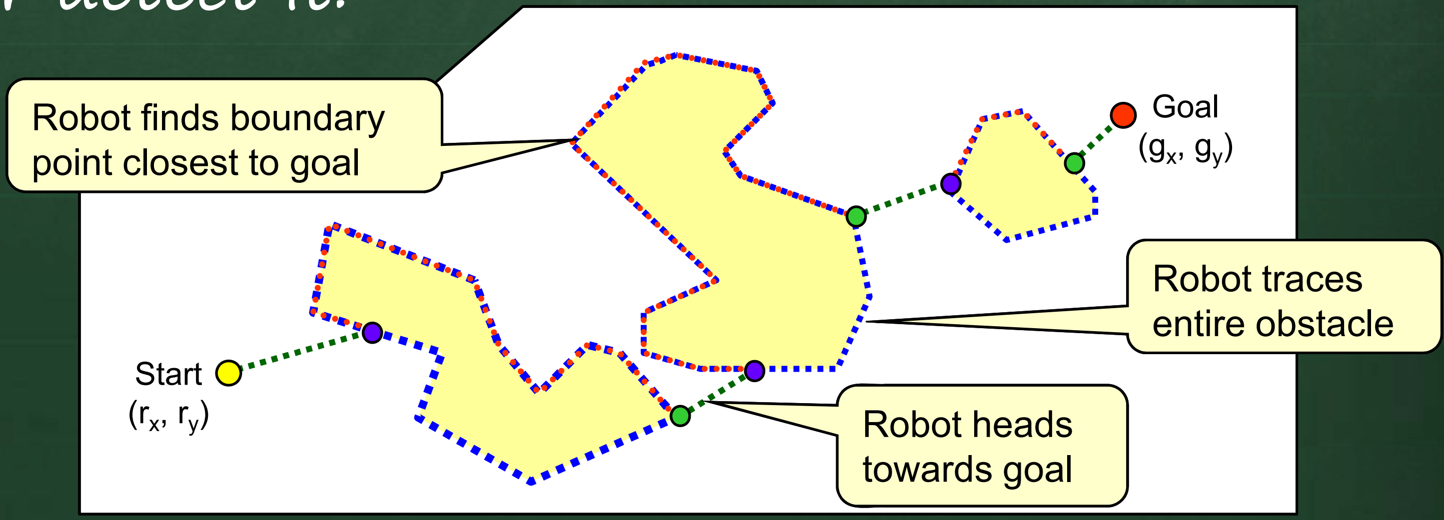
Bug Algorithms

Bug Algorithms

- First consider the following situation:
  - the goal location is known but obstacles are unknown
  - a GPS is available at any time which provides the robot with its location within the environment, or the robot has efficient on-board dead-reckoning abilities.
  - the robot has sensors to detect and follow obstacle boundaries
- There are three simple algorithms for this scenario:
  - Bug1
  - Bug2
  - Tangent Bug

The Bug1 Algorithm

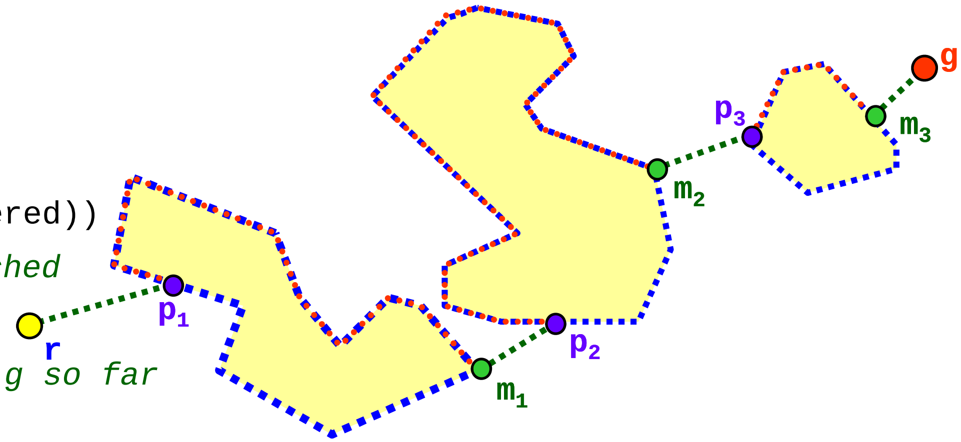
- Bug1 Strategy:
  - Move toward goal unless obstacle encountered, then go around obstacle and find its closest point to the goal.
  - Travel back to that closest point and move towards goal.
- Assumes robot knows goal location but is unable to see or detect it.

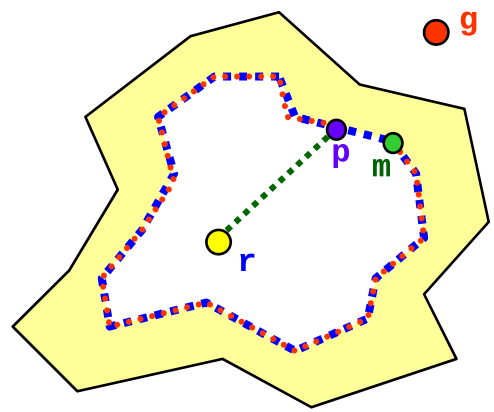


The Bug1 Algorithm

- *Here is the pseudo code for the algorithm:*

```
WHILE (TRUE)
   REPEAT
         Move from r towards g
         r = robot's current location
   UNTIL ((r == g) OR (obstacleIsEncountered))
   IF (r == g) THEN quit     // goal reached
   LET p = r       // contact location
   LET m = r       // location closest to g so far
   REPEAT
         Follow obstacle boundary
         r = robot's current location
         IF ((distance(r,g) < distance(m,g)) THEN m = r
   UNTIL ((r == g) OR (r == p))
   IF (r == g) THEN quit     // goal reached
   Move to m along obstacle boundary
   IF (obstacleIsEncountered at m in direction of g)
         THEN quit   // goal not reachable
ENDWHILE
```





The Bug1 Algorithm

- This algorithm:
  - always finds goal location (if it is reachable).
  - performs an exhaustive search for the "best" point to leave the obstacle and head towards the goal.
- If we denote the perimeter of an obstacle Obj_i as $perimeter(Obj_i)$, then the robot may travel a distance of:

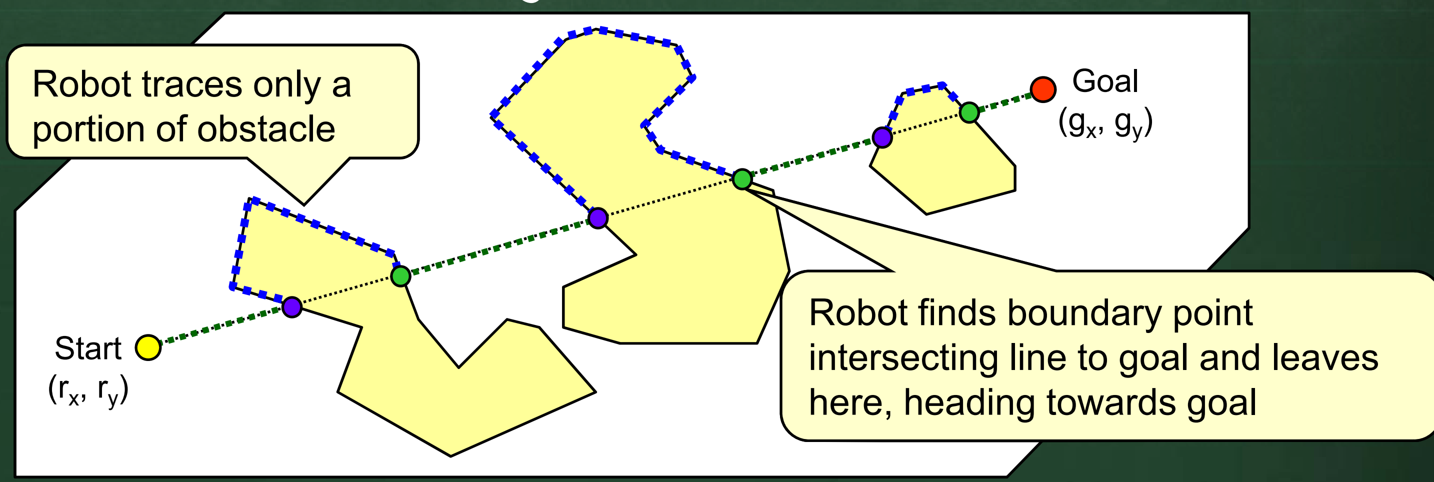
$$|\overline{sg}| + 1.5\,(perimeter(Obj_1) + perimeter(Obj_2) + \ldots + perimeter(Obj_n))$$

s = start location

Once around the obstacle to determine best position to leave from and up to ½ times around to get back to that position (since we can take the shorter of the two choices).

The Bug2 Algorithm

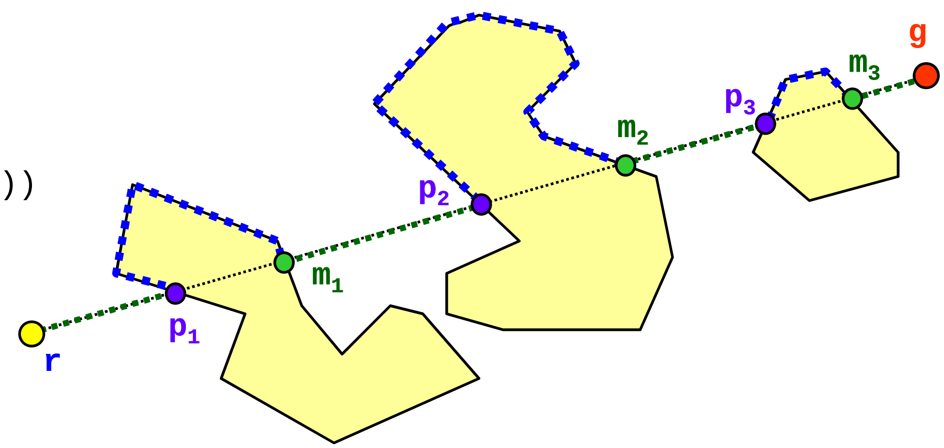
- A variation to this algorithm will allow the robot to avoid traveling ALL the way around the obstacles.
- Bug2 Strategy:
  - Move toward goal unless obstacle encountered, then go around obstacle. Remember the line from where the robot encountered the obstacle to the goal and stop following when that line is encountered again.



The Bug2 Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
   LET L = line from r to g
   REPEAT
        Move from r towards g
        r = robot’s current location
   UNTIL ((r == g) OR (obstacleIsEncountered))
   IF (r == g) THEN quit     // goal reached
   LET p = r       // contact location
   REPEAT
        Follow obstacle boundary
        r = robot’s current location
        LET m = intersection of r and L
   UNTIL (((m is not null) AND (dist(m,g) < dist(p,g)) OR (r == g) OR (r == p))
   IF (r == g) THEN quit     // goal reached
   IF (r == p) THEN quit     // goal not reachable
ENDWHILE
```



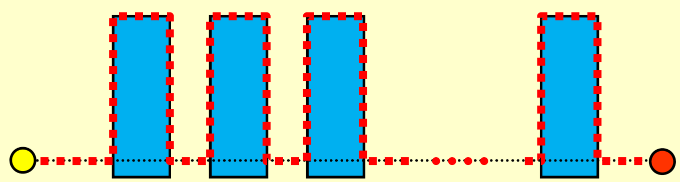
The Bug2 Algorithm

- This algorithm:
  - also always finds goal location (if it is reachable).
  - performs a "greedy" search for the "best" point to leave the obstacle and head towards the goal.

- The robot may travel a distance of:

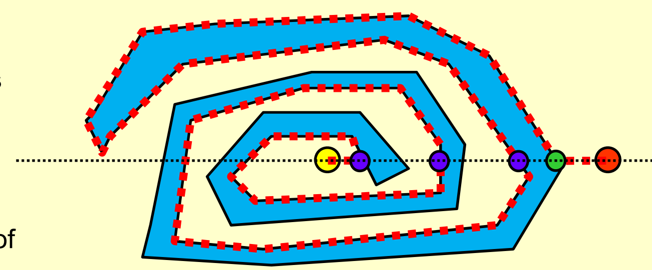
$$|\overline{sg}| + O(\text{perimeter}(obj_1) + \text{perimeter}(obj_2) + \ldots + \text{perimeter}(obj_n))$$

In worst case, we choose the "wrong way to go around the obstacle, leading to almost a full perimeter traversal:



Amount of perimeter travel will depend on choice of left/right edge following:

What happens if robot follows on its left side instead of right ?

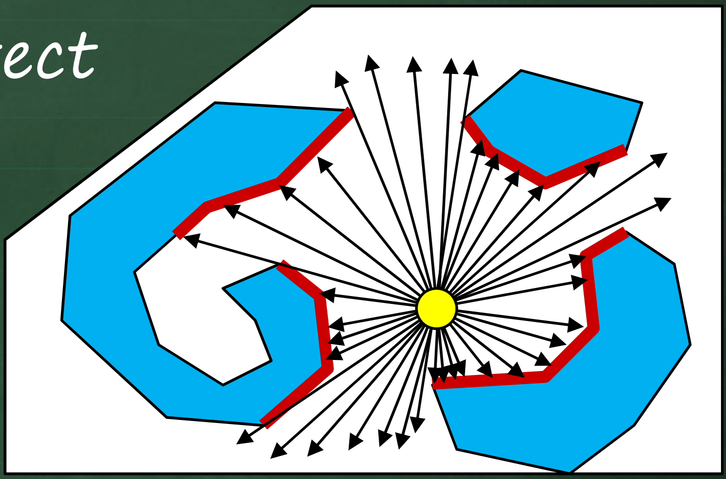


The Bug2 Algorithm

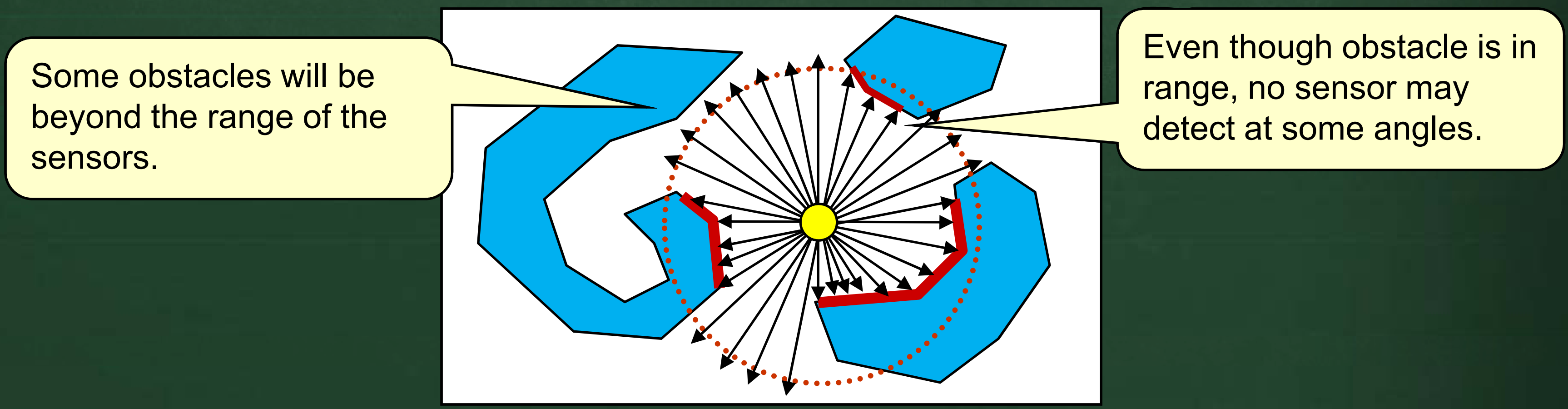
- Bug2 algorithm is quicker than Bug1.
- These two algorithms assumed that the robot could only detect the presence of an obstacle upon contact (or close proximity).
- We can improve the algorithm when the robot is equipped with a 360° range sensor that determines distances to obstacles around it.
- We will look now at the Tangent Bug algorithm

Tangent Bug Algorithm

- Assumes robot has sensors to detect distances to obstacles around it:

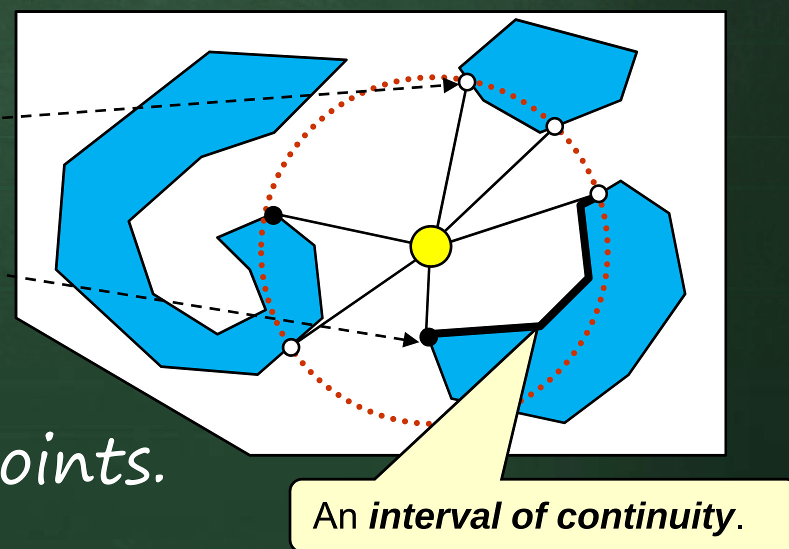


- In practice:
  - number of detectable angles is fixed (e.g., every 5°)
  - operating range of sensors is limited



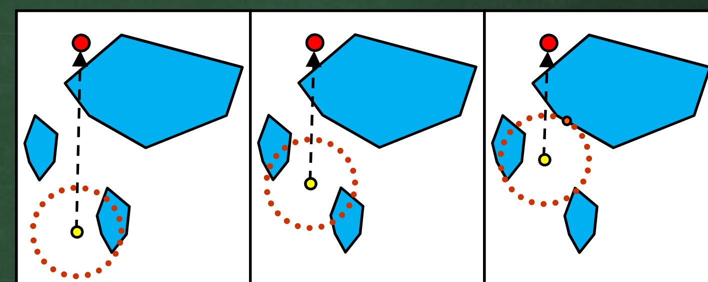
Tangent Bug Algorithm

- For purposes of explanation, assume infinite angular resolution and a finite binary detection range defined as a circle around the robot.
- Define a *discontinuity point* as a point in which sensor readings are lost (during a radial sweep) due to:
  - obstacle being out of range
  - obstacle being obscured
- An *interval of continuity* is defined by two discontinuity points.

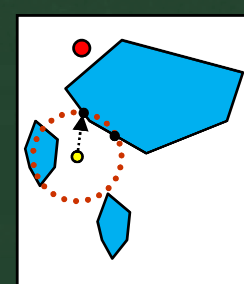


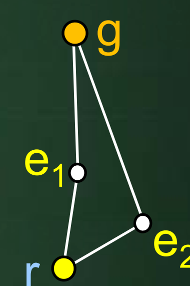
Tangent Bug Algorithm

- Algorithm is similar to the Bug2 algorithm:
  - robot moves towards goal until it senses an object directly between it and the goal.
    - in this case, the line from the robot to the goal intersects an interval of continuity.



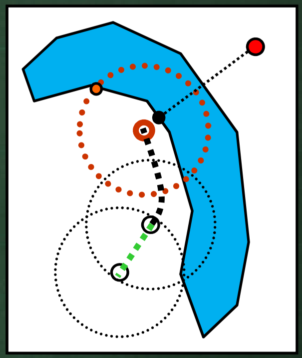
  - Robot then moves to the discontinuity point (e_1 or e_2) of the interval that maximally decreases some heuristic estimate to the goal.
    - e.g., MIN($|\overline{re_1}|+|\overline{e_1g}|$, $|\overline{re_2}|+|\overline{e_2g}|$)



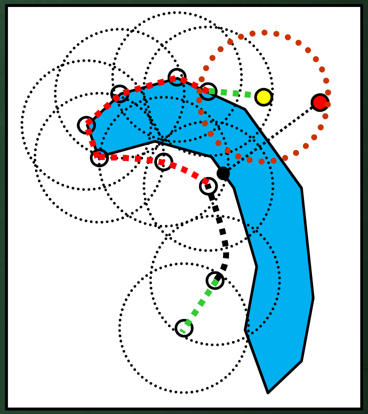


Tangent Bug Algorithm

- The robot continues heading towards the point of discontinuity until it can no longer decrease the heuristic estimate to the goal.
  - i.e., it reaches a local minimum



- The robot then follows the boundary, by heading towards the discontinuity point in the same direction.
- It then leaves the boundary by heading towards the goal again.



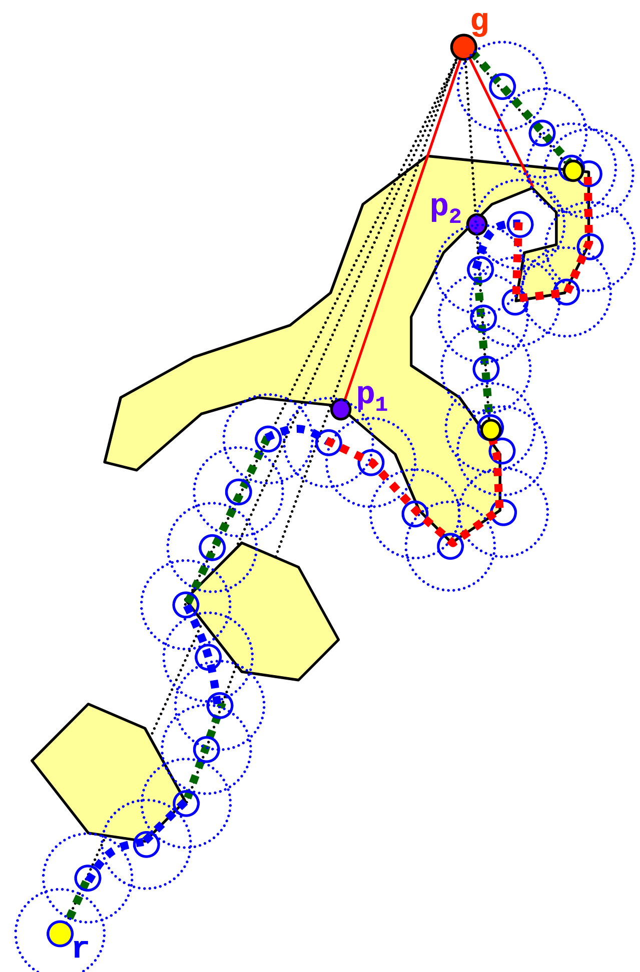
Tangent Bug Algorithm

- Here is the pseudo code for the algorithm:

```
WHILE (TRUE)
    LET w = g
    REPEAT
      r' = r   // robot's previous location
      update r by moving towards w
      IF (no obstacle detected in direction w) THEN w = g
      ELSE
         LET eL and eR be discontinuity points
         IF ((dist(r, eL)+dist(eL,g)) < (dist(r, eR)+dist(eR,g)))
            THEN w = eL ELSE w = eR
    UNTIL ((r == g) OR (dist(r',g) < dist(r,g)))
    IF (r == g) THEN quit      // goal reached

    LET p = m = r // contact location
    LET dir = direction of continuity point (L or R)
    REPEAT
      LET w = the discontinuty point in direction dir
      IF (dist(r,g) < dist(m,g)) THEN m = r
      update r by moving towards w
    UNTIL ((dist(r,g) < dist(m,g)) OR (r == g) OR (r == p))

    IF (r == g) THEN quit      // goal reached
    IF (r == p) THEN quit      // goal not reachable
ENDWHILE
```



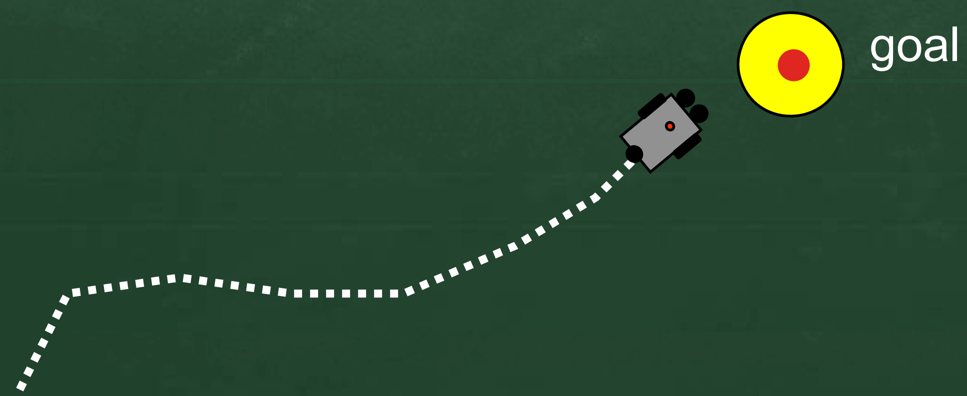
Bug Algorithms

- Bug path planning algorithms have advantages:
  - \+ Simple and intuitive
  - \+ Easy implementation
  - \+ Guaranteed (theoretically) to reach goal (when possible)

- The algorithms do have practical problems:
  - – Assumes perfect positioning (not really possible)
  - – Assumes error-free sensing (not ever possible)
  - – Real robots have limited angular resolution

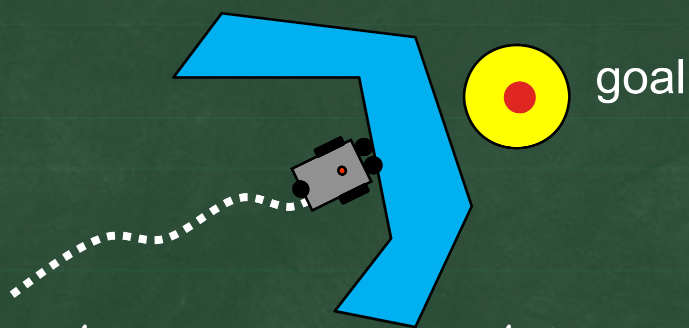
Unknown Goal Navigation

- Consider now the situation in which the location of the robot and the goal is also unknown:
  - robot must be able to detect (sense) goal location (e.g., light/energy source, sound, image, etc…)
  - this "goal" sensor MUST have way of determining direction to the source.

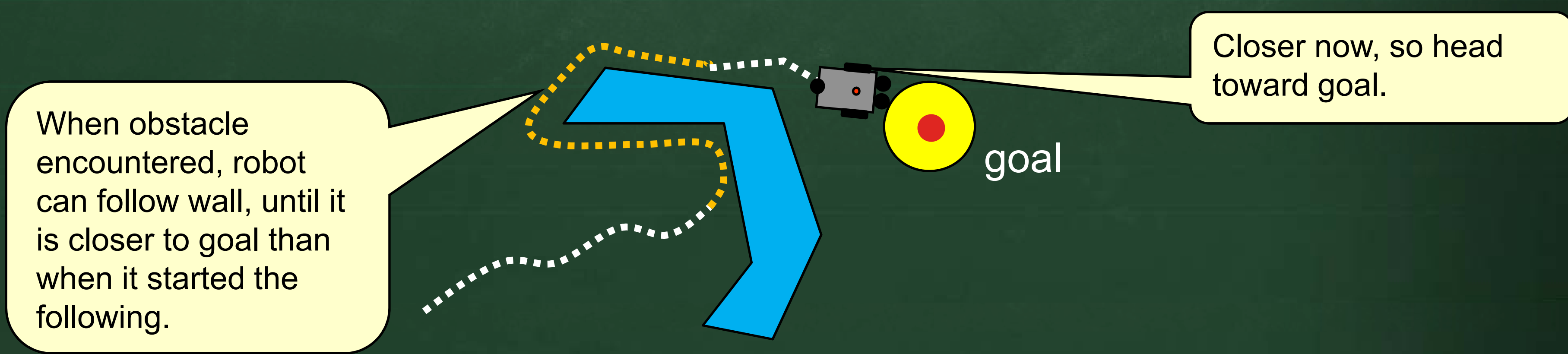


Unknown Goal Navigation

- without knowledge (or estimate) of location, obstacles can easily prevent robot from reaching its goal.

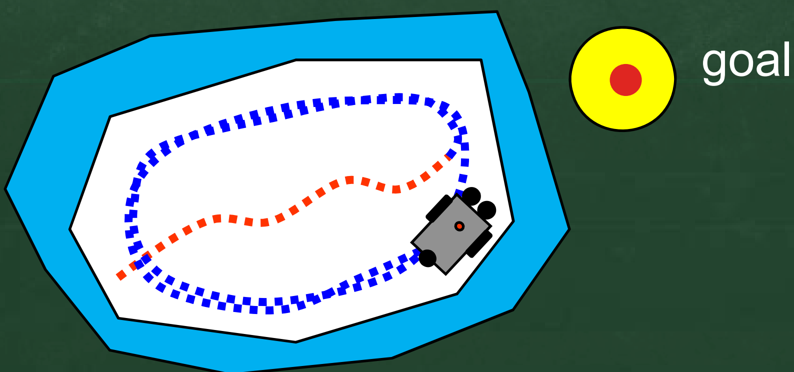


- ability to sense "closeness" to goal (e.g., intensity) can help.



Unknown Goal Navigation

- With the ability to detect closeness to the goal, a similar strategy to the Bug2 algorithm can be applied to work around obstacles.
- Problems occur of course with multiple ambiguous goals (e.g., multiple light sources).
- Also, cannot detect cases where goal is unreachable:



Vector Field Histograms

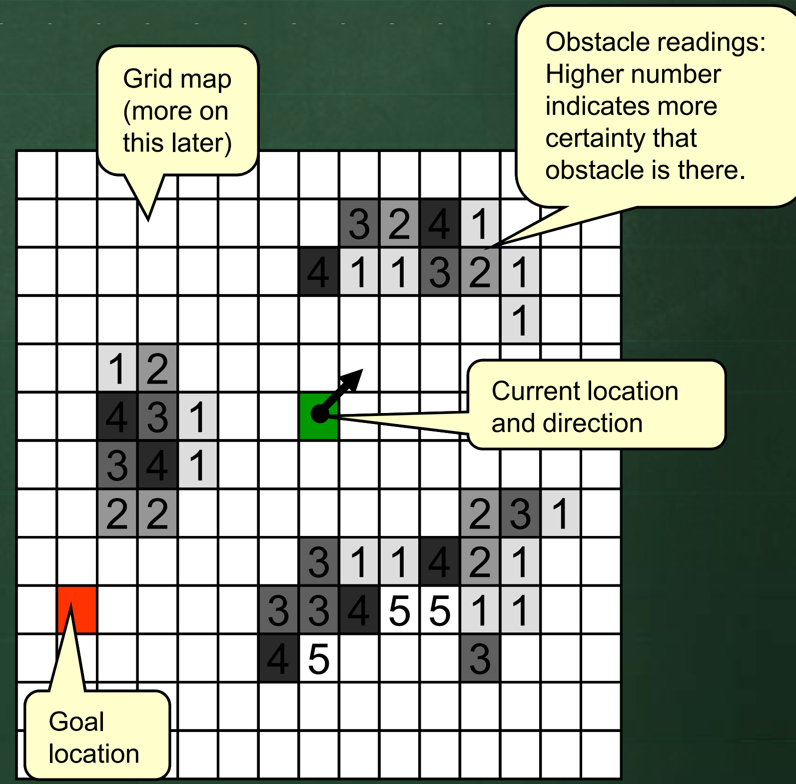
Vector Field Histograms

- *Vector Field Histograms* are used to quickly navigate around obstacles.

- This technique steers in the "best" direction that leads to the goal.
  - Best is defined in terms of "most likely" to avoid obstacles
- Each location in the grid map presents a vector, based on the current sensor readings.
- Vectors are combined to produce an overall direction vector which is used to steer the robot.

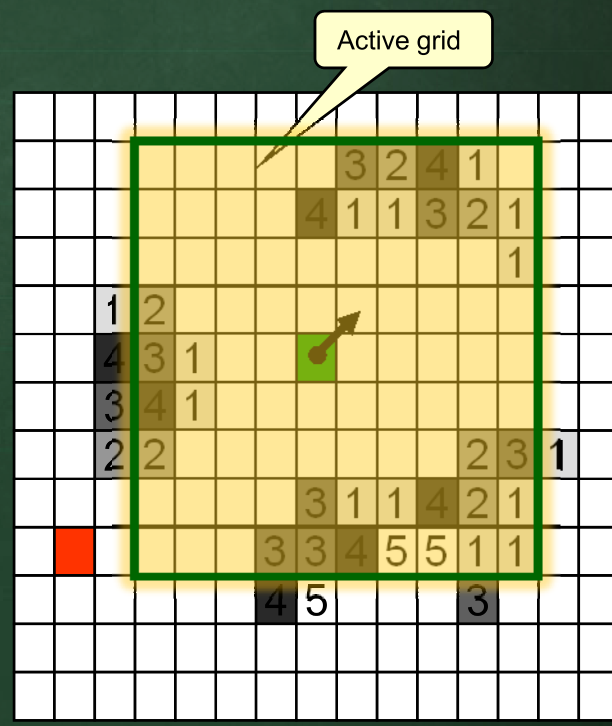
Vector Field Histograms

- Also incorporates uncertainty in obstacle locations by making decisions based on an uncertainty map (we will discuss maps later).
- This 2D grid is then reduced to a histogram from which the steering angle and velocity controls for the robot are computed.



Vector Field Histograms

- In order to simplify calculations, in practice only a portion (i.e., window) of the whole grid is used, called the *active grid*.
  - may be function of sensor range as well.
  - changes over time as robot moves.
- The histogram will be computed for the cells in this window only.
  - Robot only examines local info., not global as with potential fields (discussed later).

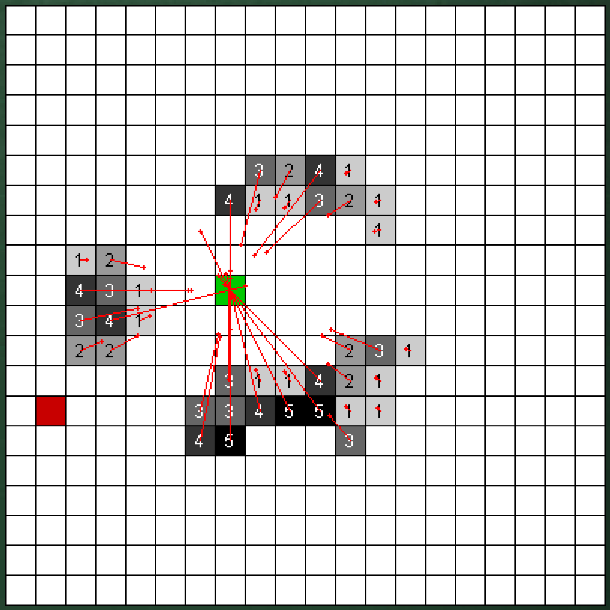


Vector Field Histograms

- For each cell, a vector is computed
  - directed from the cell towards robots current location
  - magnitude corresponds to the certainty of obstacle (i.e., the cell's current value)

- These vectors represent a desire to "push" away from the obstacle.

- Need to incorporate a steering towards the goal as well.



Calculating Vectors

- Let (c^{ij}_x, c^{ij}_y) be the location of cell at position (i, j) in the grid with certainty value c^{ij}_{val}.
- Let (r_x, r_y) be the robot's current location
- Compute vector v^{ij} for cell c^{ij} as follows:

$$v^{ij}_{mag} = (c^{ij}_{val})^2(\phi - \sigma\ dist((c^{ij}_x, c^{ij}_y), (r_x, r_y)))$$

Constant used for normalizing. (e.g., 12) [ϕ]

Constant used for normalizing. (e.g., 1.5) [σ]

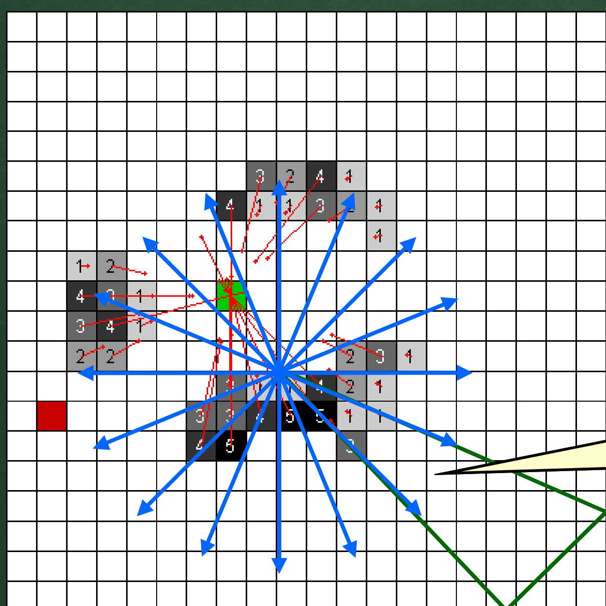
Distance from cell's center to robot's center [$dist$]

$$v^{ij}_{dir} = atan((c^{ij}_y - r_y) / (c^{ij}_x - r_x))$$

Need to also add 180° when $c^{ij}_x - r_x >= 0$

Calculating Sectors

- Given an angular resolution α, the vectors are grouped into a specified number of angular regions (i.e., sectors).
  - e.g., if α = 24, then each sector represents angles within a 15° wedge. If α = 16 then wedges are 22.5° etc…



Sector s^{ij} of cell c^{ij} computed as:

$$s^{ij} = (\text{int})\ v^{ij}_{dir} / (360° / \alpha)$$

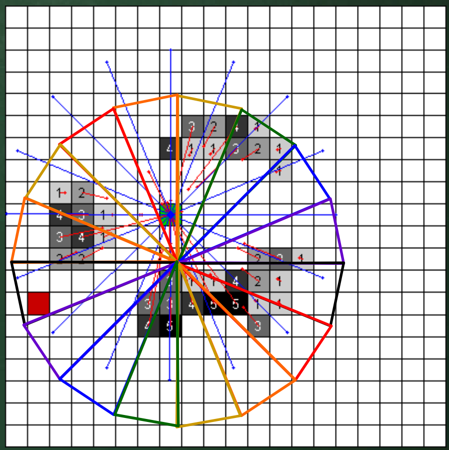
Multiple vectors are grouped within this 22.5° sector.

Calculating the Histogram

- Compute a histogram with α bins (i.e., one bin for each sector).
- Compute the value h_k of a sector ($0 \leq k < \alpha$), called the ***polar obstacle density***, as the sum of the magnitudes of all vectors with the same s^{ij}.

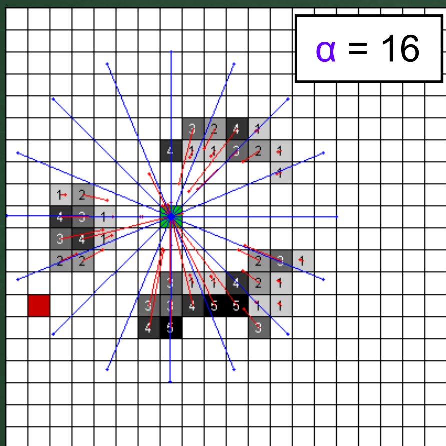
$$h_k = \sum_{i,j} (v^{ij}_{mag} \mid s^{ij} = k)$$

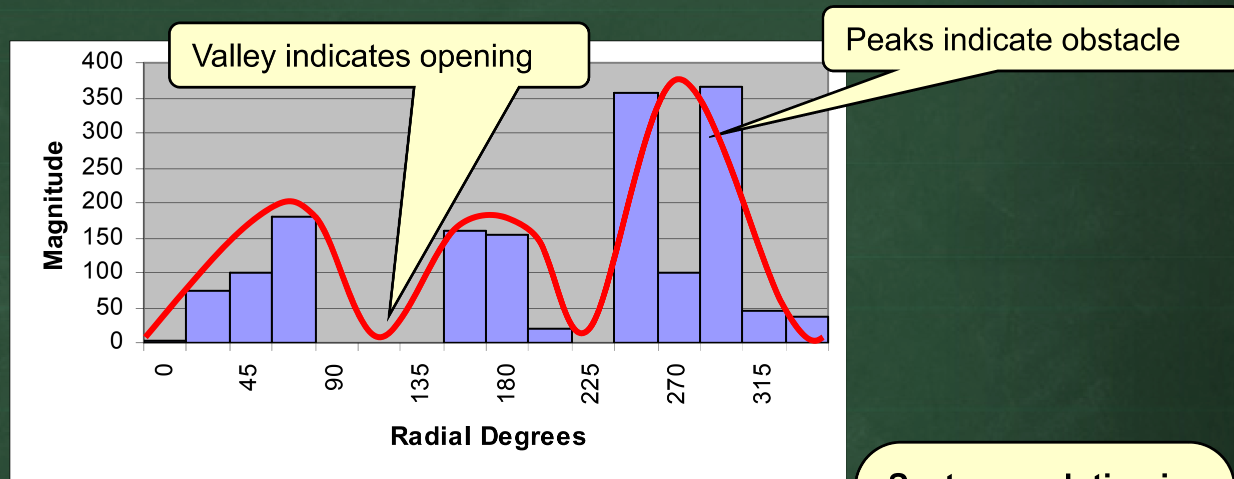
- All cell vectors v^{ij} are thus distributed into the appropriate sector.

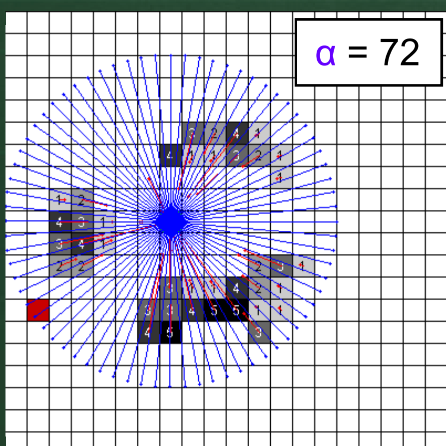


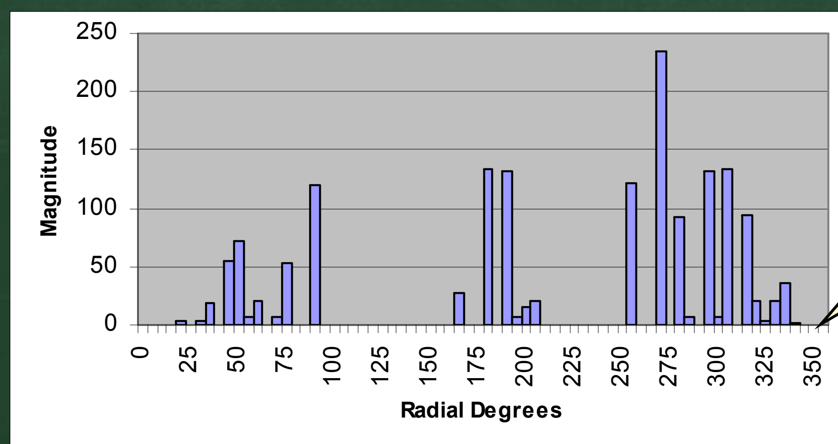
Calculating the Histogram

- A histogram can then be generated:





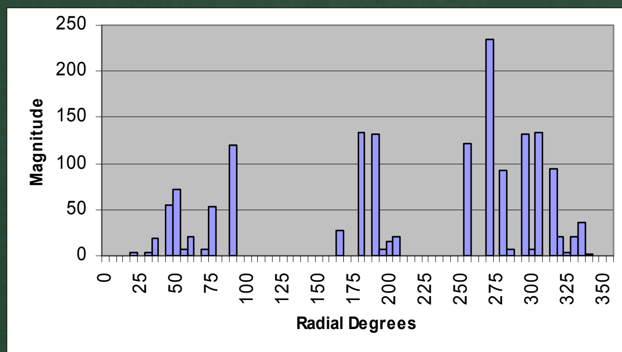




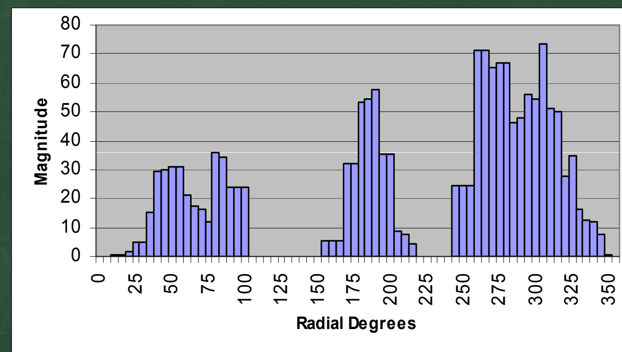
Sector resolution is better:
Can now easily distinguish opening between these two obstacles

Calculating the Histogram

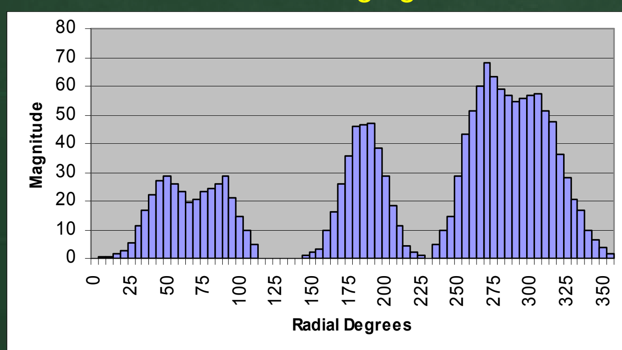
- Good idea to average the histogram by setting a histogram value to be the average of the nearby values before and after it.



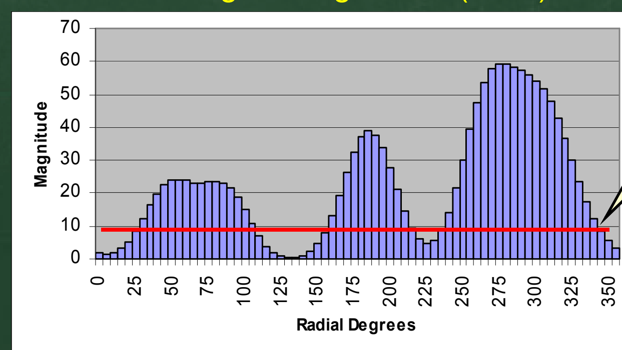
No averaging



Average 4 neighbours (once)



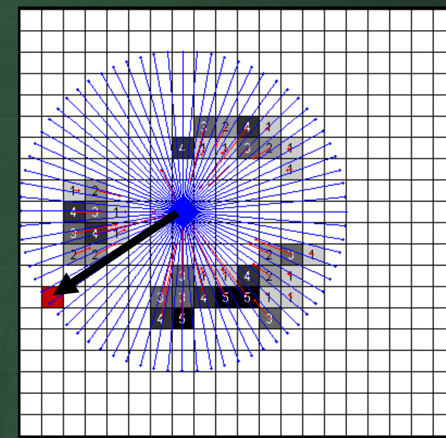
Average 4 neighbours (twice)



Average 4 neighbours (four times)

Need to vary threshold of what constitutes an "opening" between the obstacles.

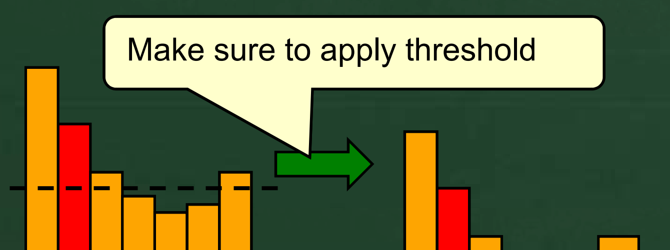
Using the Histogram

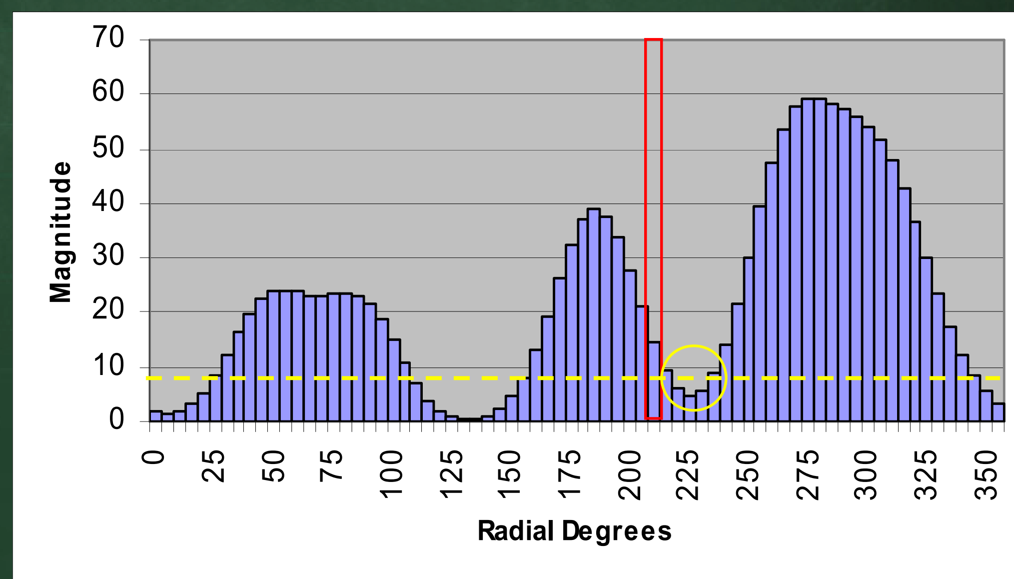


- Once histogram is created, need to determine sector that contains vector from robot location to goal location:

$$s^g = (int)\ (atan((g_y - r_y) / (g_x - r_x)) / (360^\circ / \alpha))$$

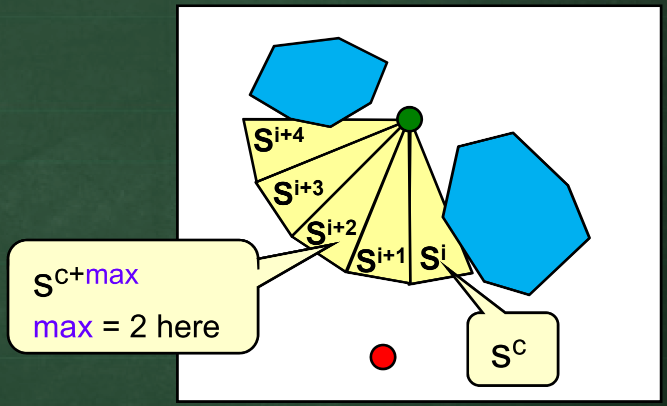
- Then find the valley $s^i, s^{i+1}, \ldots, s^k$ that is closest to sector s^g.





Using the Histogram

- Note that all consecutive sectors s^i, s^{i+1}, …, s^{i+k} of a valley must have magnitude < threshold value ε.
- Let s^c be either sector s^i or s^{i+k}, whichever is closest to sector s^g.



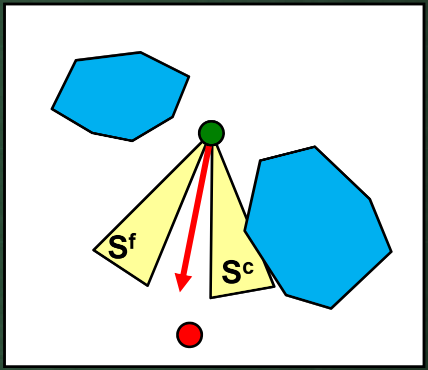
- Can classify valleys as:
  - ***wide***: if k > some threshold *max*

    Let s^f = sector $s^{c \pm max}$ (whichever lies in valley)
  - ***narrow***: if k ≤ *max*

    Let s^f = either s^i or s^{i+k} whichever is not s^c

Using the Histogram

- The robot then moves in the direction $\theta = (s^f + s^c)/2$
- Note that robot does not simply head towards the goal if there is an opening.



- Instead, algorithm causes robot to stick close to the obstacles so as to have a better chance of navigating around them.
- Successful navigation is accomplished by tweaking threshold values.

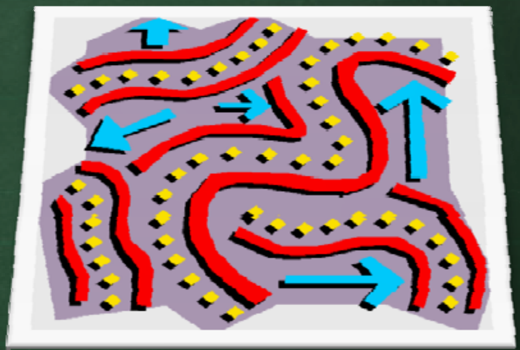
Issues

- If ε threshold is set too high
  - robot may get too close to obstacles
  - may move too quickly to be able to prevent a collision
- If ε threshold is set too low
  - robot may miss out on some valid candidate valleys
- Generally, the threshold is only vital when the robot is moving quickly through tightly packed obstacles.

Map Representation and Storage

Maps

- An robot's environment may change over time.
- A *map* is a stored representation of an environment at some particular time.
- Allows robot to:
  - plan navigation strategies
  - avoid obstacle collisions during travel
  - identify changes in the environment
  - identify accessible/inaccessible areas
  - verify its own position in the environment



- We will only consider 2-D maps in this course

Maps

- Realistically, maps are only estimates
  - often imprecise

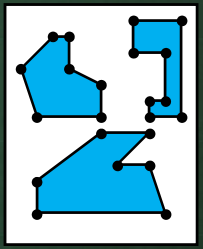
- Robot must also rely on its sensors to avoid collisions since maps may be inaccurate or simply wrong.
- For now, we will consider our maps to be accurate, readily available and fixed (i.e., static).
- We will look more later on how to create maps in unknown environments.

Map Representation

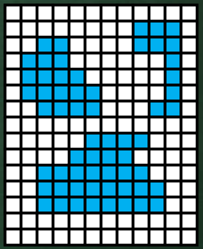
- Maps can be represented as various types:
  - Topological maps
    - Keeps relations between obstacles (or free space) within env.
  - Obstacle maps
    - Keeps locations of obstacles and inaccessible locations in env.
  - Free-space maps
    - Keeps locations that robot is able to safely move to within env.
  - Path maps
    - Keeps set of paths that robot can travel along safely in env.
    - Usually used in industrial applications to move robots along known safe paths.

Map Representation

- Maps are stored in one of two main ways:
  - **Vector**
    - stored as collection of line segments and polygons
    - usually represents obstacle boundaries



  - **Raster**
    - storage in terms of fixed 2D grid of cells
    - each cell stores probability of occupancy (i.e., obstacle)

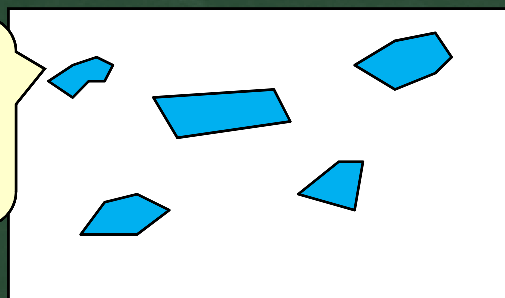


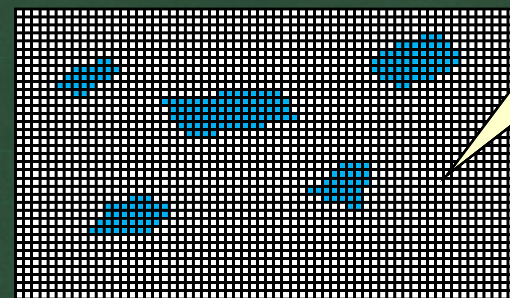
- Main differences lie in:
  - storage space requirements
  - algorithm complexity and runtime

Map Storage Space

- Large environments with few and simple obstacles take less space to store as vector:

Only need to store a few vertex coordinates and edge connections.





Both "occupied" and "empty" regions take up storage space.

- Smaller, obstacle-dense environments may be better stored as raster/grid:

Storing many vertices and edges may require more space than storing a small course grid.

Map Storage Space

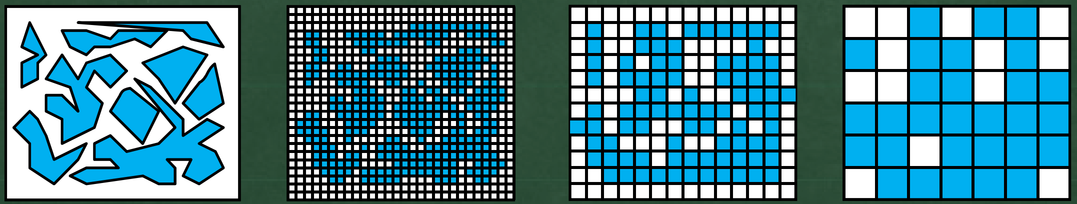
- Vector maps require the following storage space:
  - m obstacles with n vertices each requires storage of (x,y) vertex coordinates as well as edges (e.g., stored as linked list pointers)
  - Storage = $(m * n)*2 + 2*(m * n) = O(mn)$

Optionally, edges don't have to be stored explicitly, but same time complexity.

- Raster maps require storage space that varies according to grid size (i.e., according to desired resolution):
  - a grid of size $M \times N$ takes $O(MN)$
- If $m,n \ll M,N$ then vector maps are more efficient

Map Storage Space

- Of course, much varies according to the resolution of the raster maps (i.e., depends on M & N).
- Resolution depends on desired accuracy.   Notice the difference that it can make on the map:

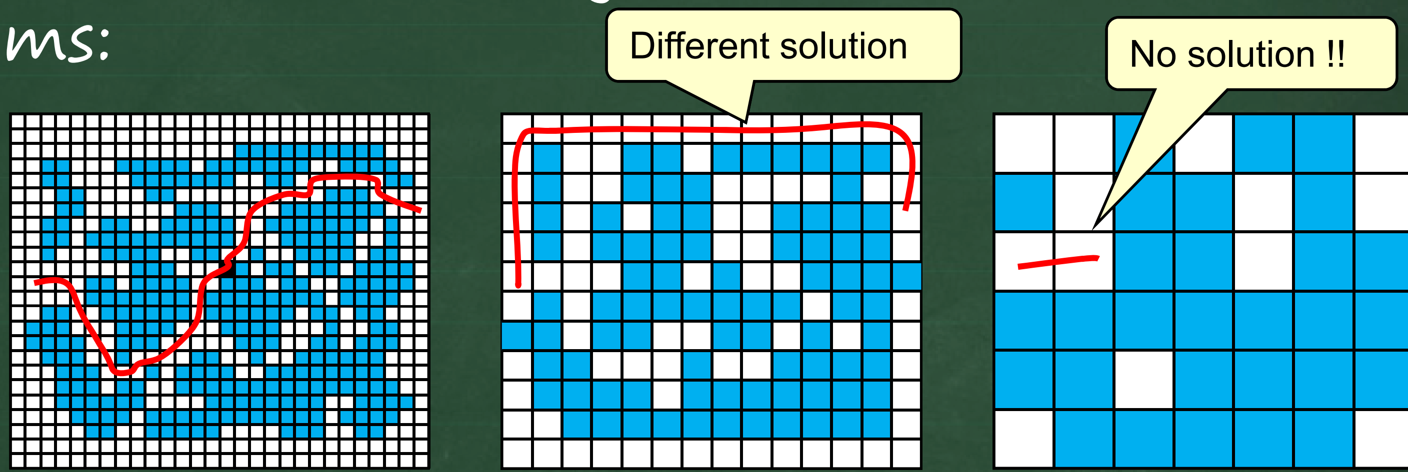


actual   28 x 24   14 x 12   7 x 6

- As resolution decreases:
  - storage requirements are reduced
  - representation of "true" environment is compromised.

Map Accuracy

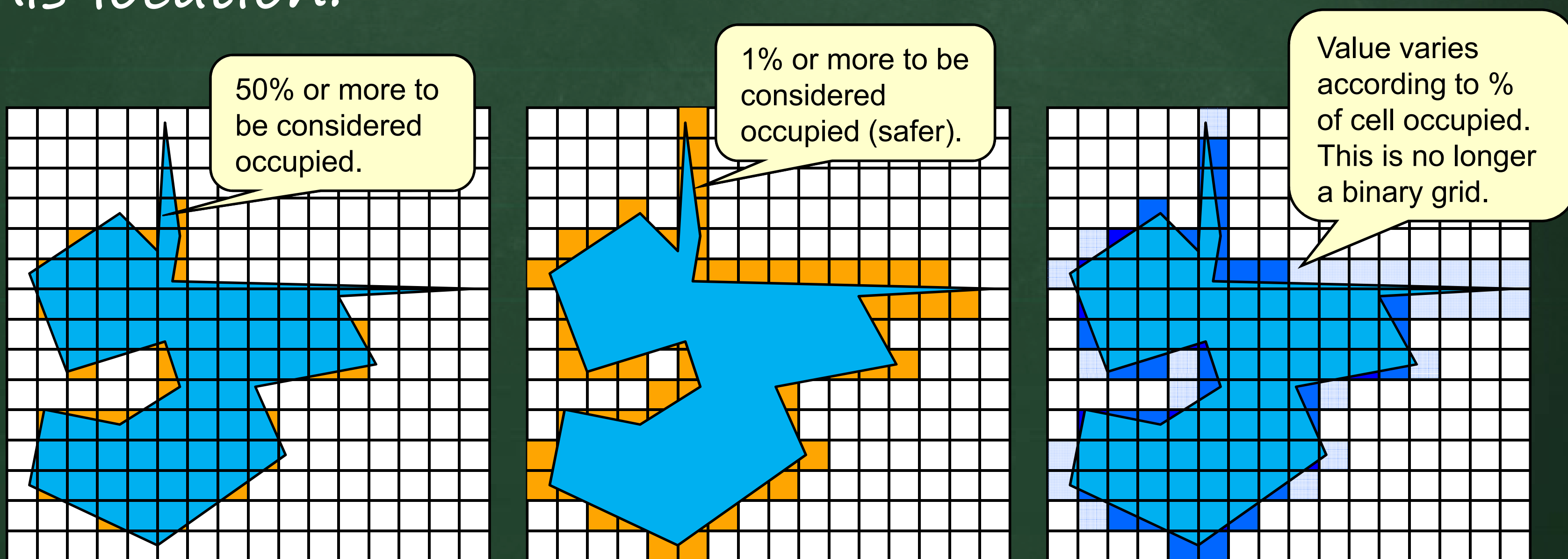
- This decrease in accuracy can affect solutions to problems:



- With vector maps, solution does not depend on storage resolution, but instead on numerical precision:

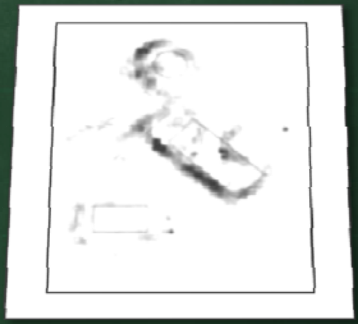
Map Accuracy

- Other issues in raster map creation is in regards to robot safety.
- Occupancy of grid cells can depend on some threshold indicating "*certainty*" that obstacle is at this location:



Map Accuracy

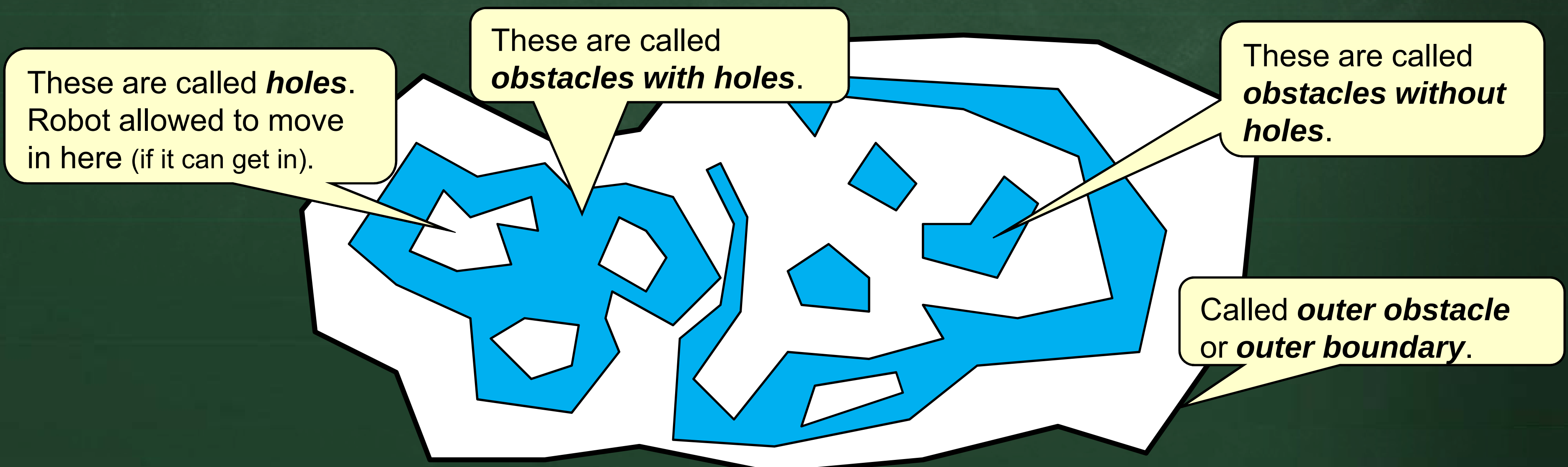
- In practice, many robots use such raster maps because they allow for "fuzziness" in terms of obstacle position.
  - They are commonly called *occupancy grids* (or *certainty grids* or *evidence grids*).
- Still useful since most maps are constructed based on sensor data (which is already uncertain).
- The cell's occupancy value indicates the probability that an obstacle is at that location.



Map Hierarchy

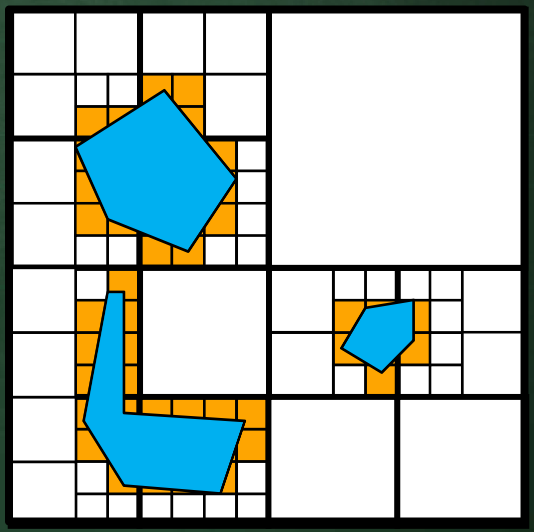
- Maps may also be *hierarchical*
  - store relationship between groups of obstacles or cells.
  - often called *topological* since indicates connectivity between nodes or areas

- Vector maps can have holes within obstacles.



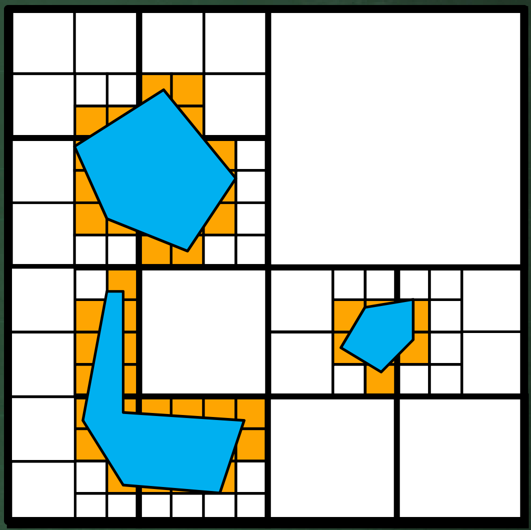
Map Hierarchy

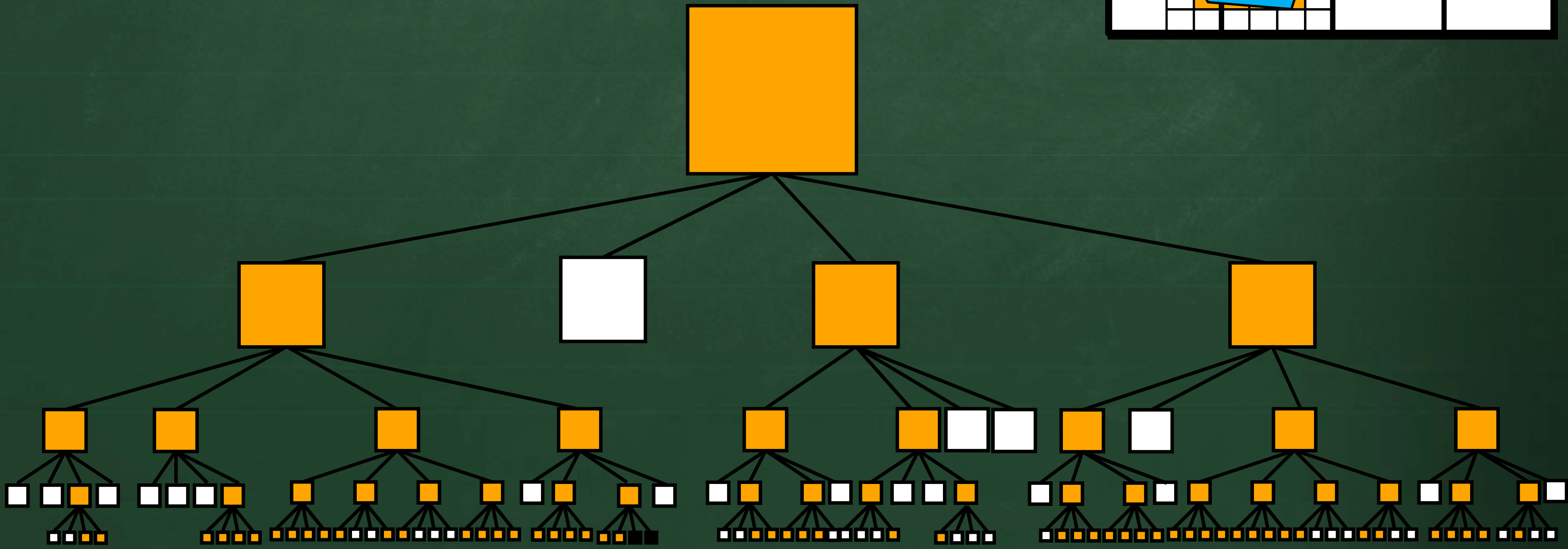
- Raster maps can store holes as well, this simply appears as free space cells.
- To reduce storage requirements, rasters are sometimes stored using a *quadtree* data structure:
  - environment is recursively divided into half along horizontal and vertical directions to produce 4 sub-areas
  - divide only if obstacle exists in area
  - resolution varies in certain areas according to density of obstacles



Map Hierarchy

- Result is a tree-like representation where siblings are ordered left to right in this order:



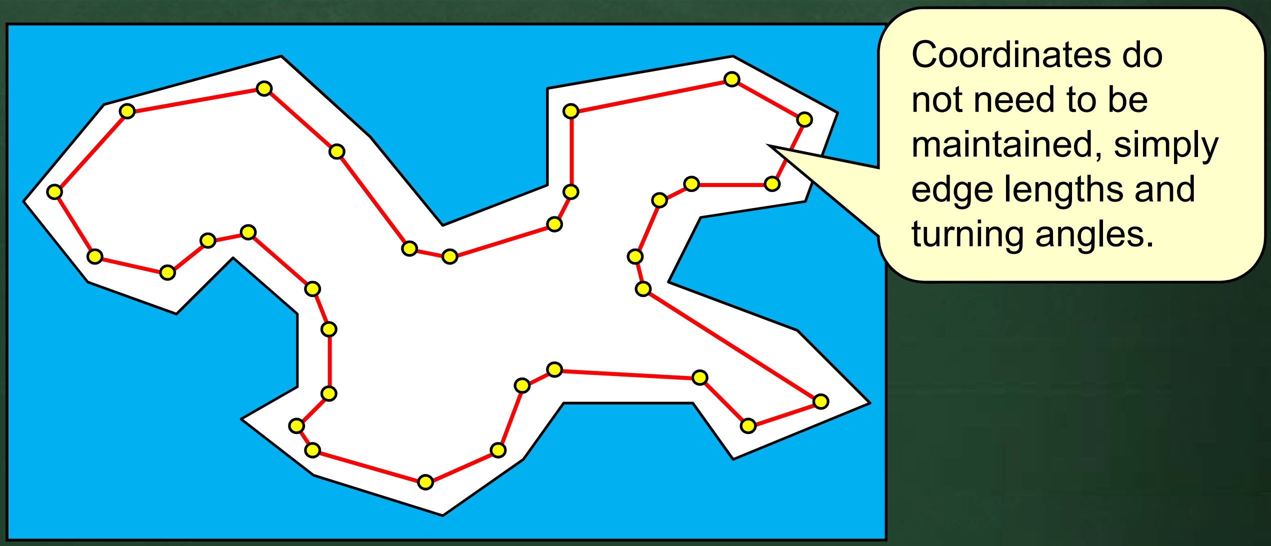


Map Hierarchy

- Quadtrees:
  - can dramatically reduce storage space requirements in sparse (not obstacle-dense) environments.
  - require trickier coding in algorithms due to the different cell sizes and shared boundaries of regions.
- Not overly popular storage choice in practice when using certainty grids:
  - as robot sensor data arrives, certain regions need to expand or collapse according to new obstacle certainties.
  - sensor noise and small fluctuations cause most levels to become expanded.

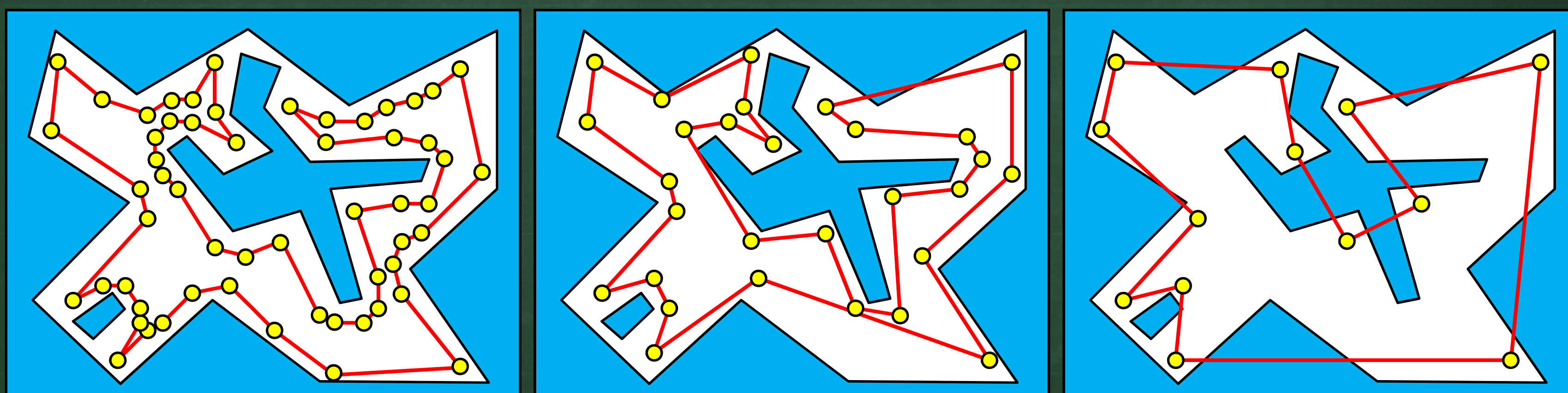
Feature (Landmark) Maps

- One type of topological map is that of a feature map … which stores *features* of an environment
  - Features may be free space, landmarks or boundaries
- Consider a map obtained from a tracing out of an obstacle boundary, recording edge and corner features:



Feature (Landmark) Maps

- The map produced depends on the accuracy of the robot as well as some parameters such as the minimum turn angle that is considered to be a vertex:

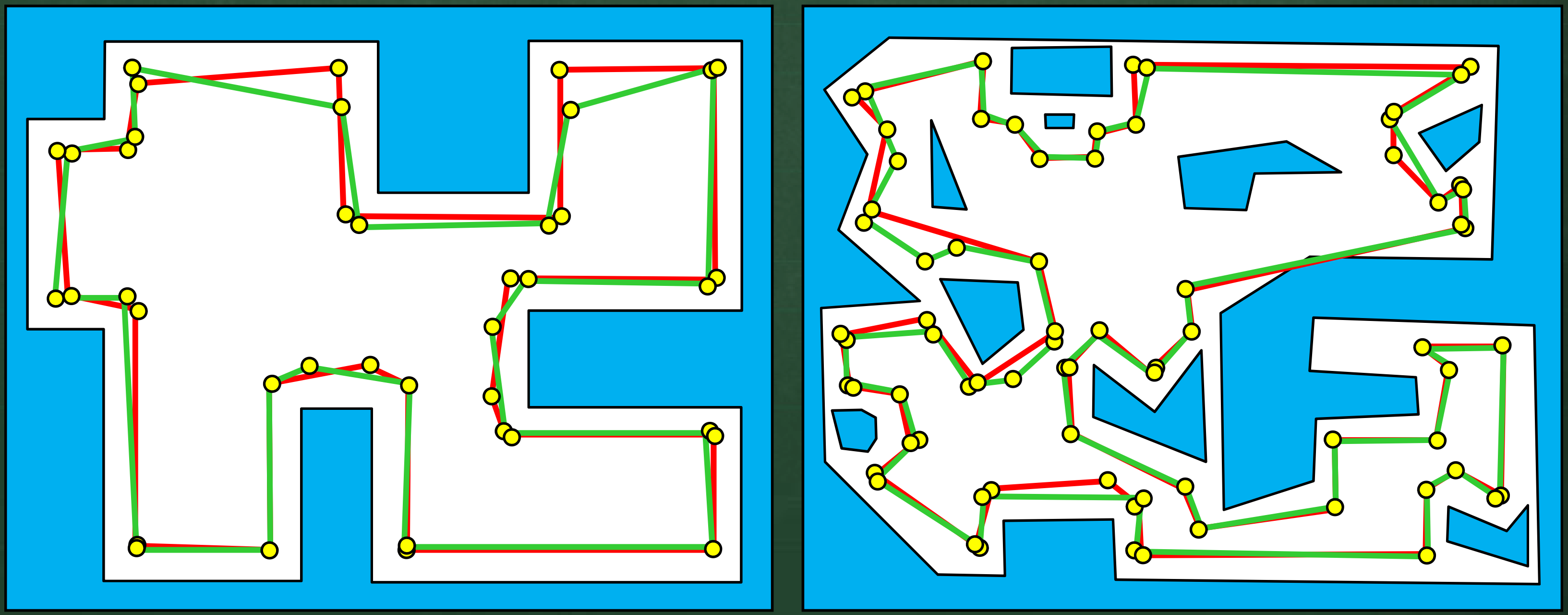


15° turn threshold 45° turn threshold 60° turn threshold

- Hence, the map can be made coarse or fine.

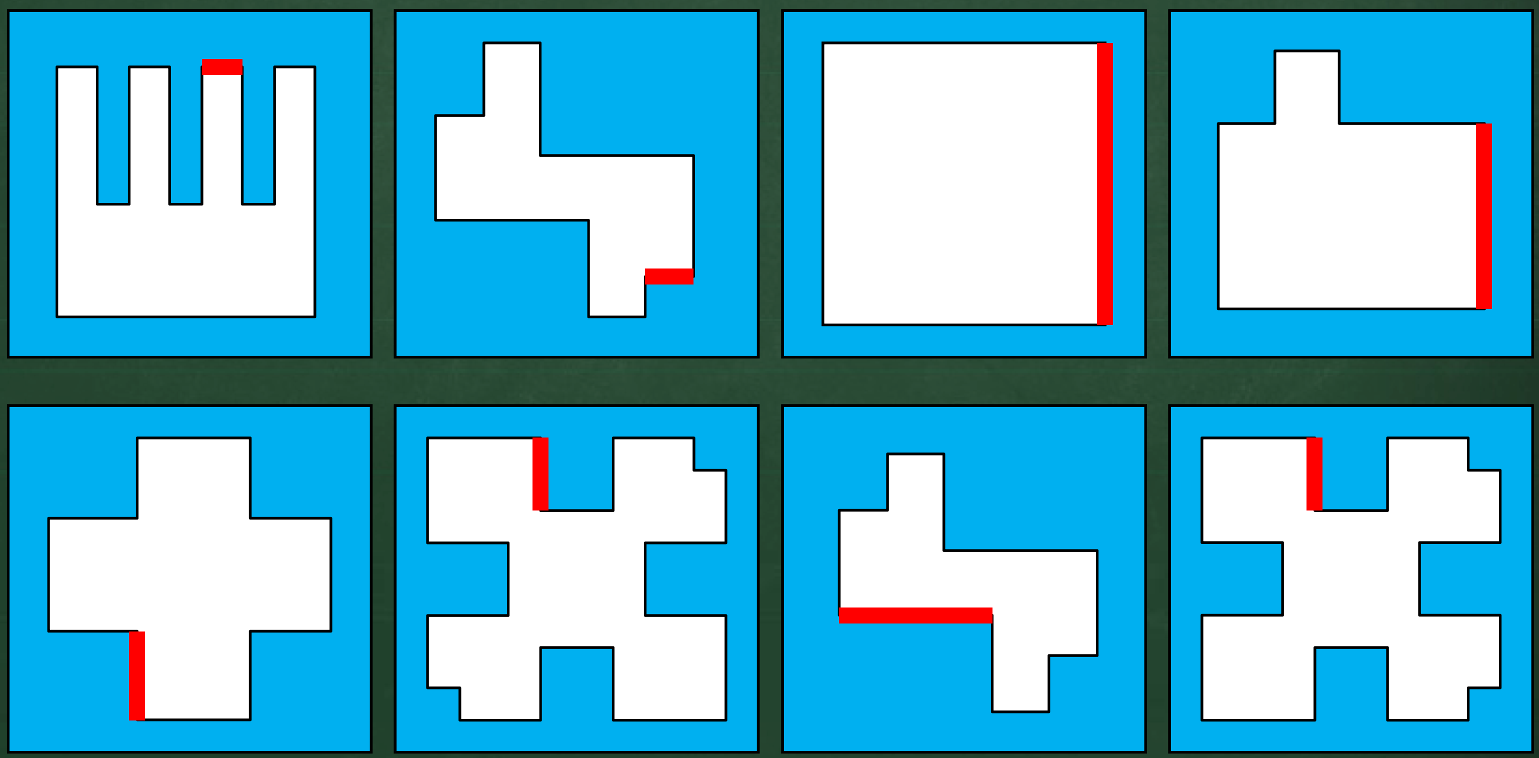
Feature (Landmark) Maps

- Different traces around the exact same obstacle, can produce different maps. Here are two different environments showing CW and CCW traces:



Feature (Landmark) Maps

- When navigating to a specific edge, symmetrical environments will cause ambiguities. Which of these are ambiguous in finding the goal edge ?

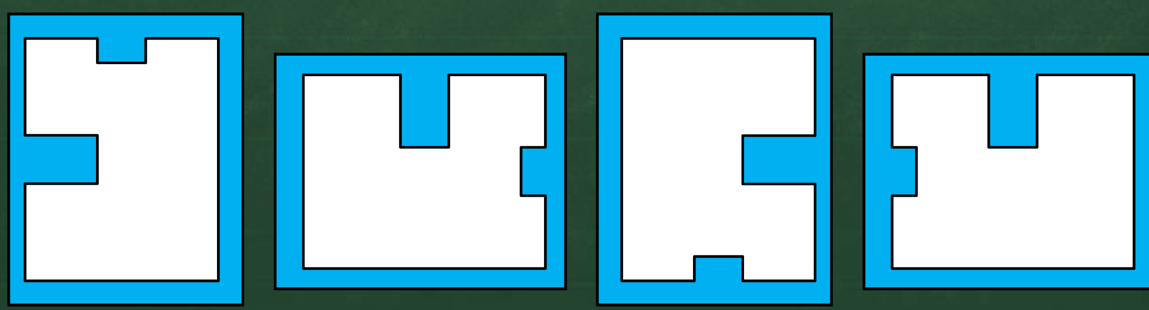


Feature (Landmark) Maps

- Once mapped, a robot cannot determine global orientation of edges.

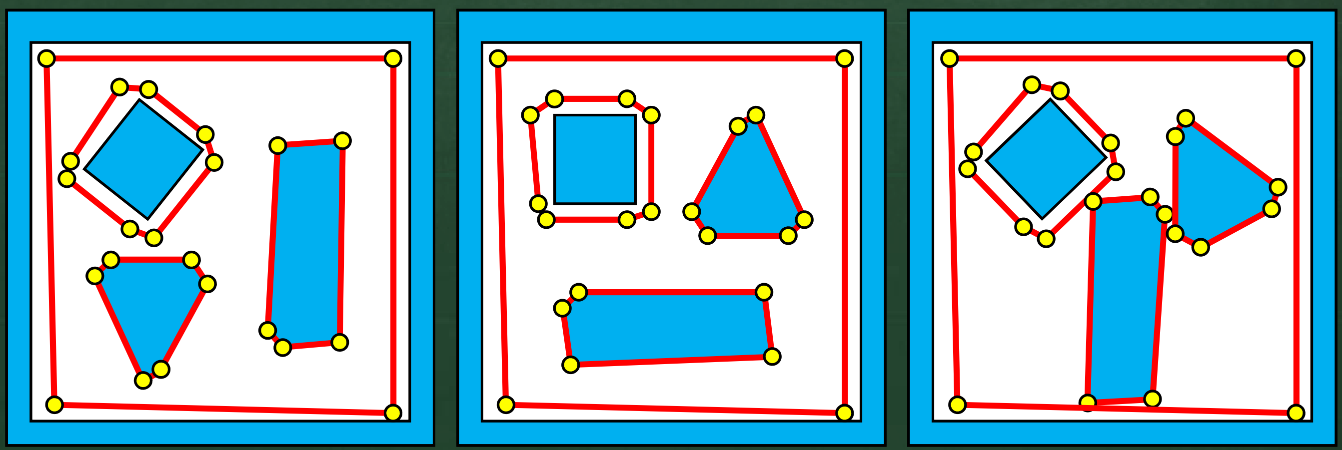
  Unless an external reference or global coordinate system is available.

  - For example, three of the following environments all appear the same to the robot in terms of consecutive edge lengths. Which one appears different ?



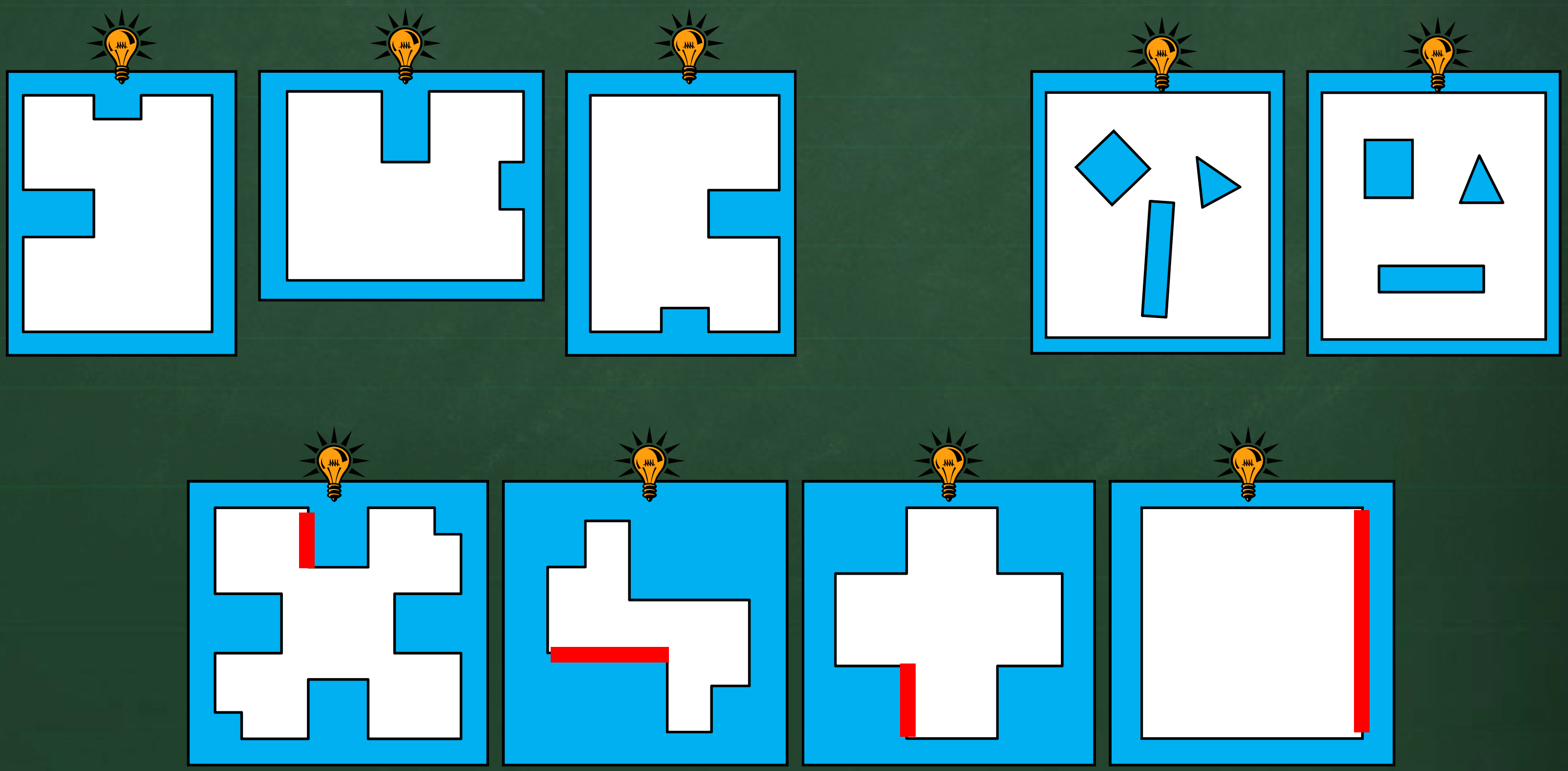
Feature (Landmark) Maps

- In addition, by simple tracing (i.e., without coordinate info), neither global orientation, relative orientation nor relative position can be determined.
  - For example, can you see why the following maps are indistinguishable ?



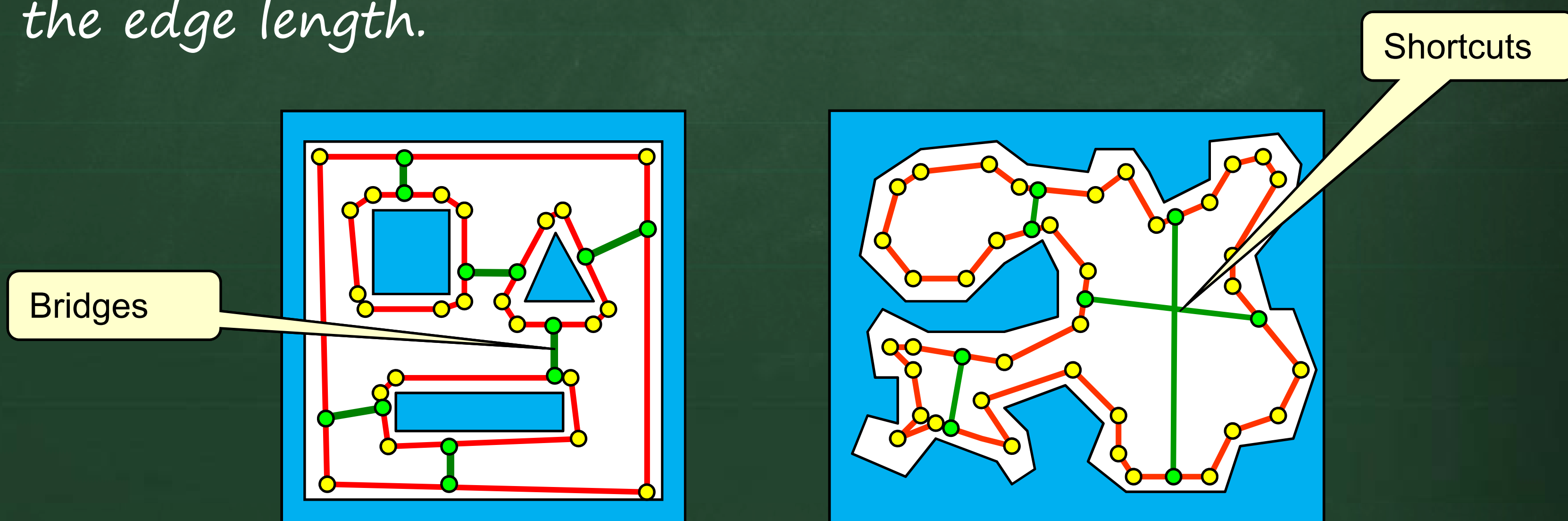
Feature (Landmark) Maps

- Of course, by providing additional global information (e.g., compass or external light source), many of these problems will disappear:



Feature (Landmark) Maps

- Bridges/shortcuts can be determined between various obstacles in order to provide topological ordering as well as distance estimations.
  - One way of implementing this is to have the robot "shoot out" from an edge perpendicularly and form a link in the map from that edge to the one it encountered, remembering the edge length.



Map Choosing

- For now, we will consider all environments to be static (unchanging) and to be available to the robot from some external source (perhaps computed manually).

- How can we move the robot efficiently in such an environment ?

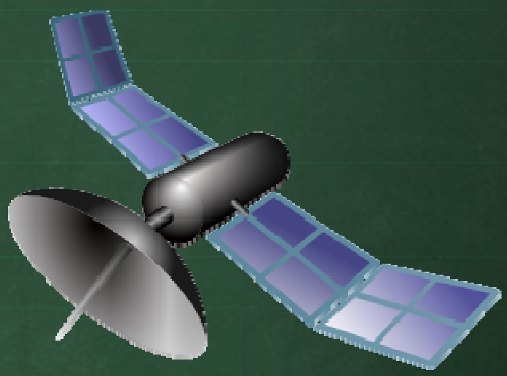
- Many algorithms and problems are solved in the area of computational geometry while assuming known static environments and point-sized robots.

- Consider first vector-based maps …

Navigation in Known Environments

Known Environments

- Consider now the situation in which the obstacles are known:
  - robot may be given a "perfect" map from external source (e.g., satellite photos or predefined fixed environment)
  - robot can find map on its own



- Assume that robot's position & goal are both known
  - i.e., use dead reckoning or GPS to get position
- Algorithms depend on different types of map representation.

Three Strategies

- We will consider 3 strategies for navigating to a goal location when obstacle locations are known.
  - Robot has global knowledge of all obstacles and can pre-compute some information which the robot then uses to navigate to the goal.
  - A fixed path is NOT computed.

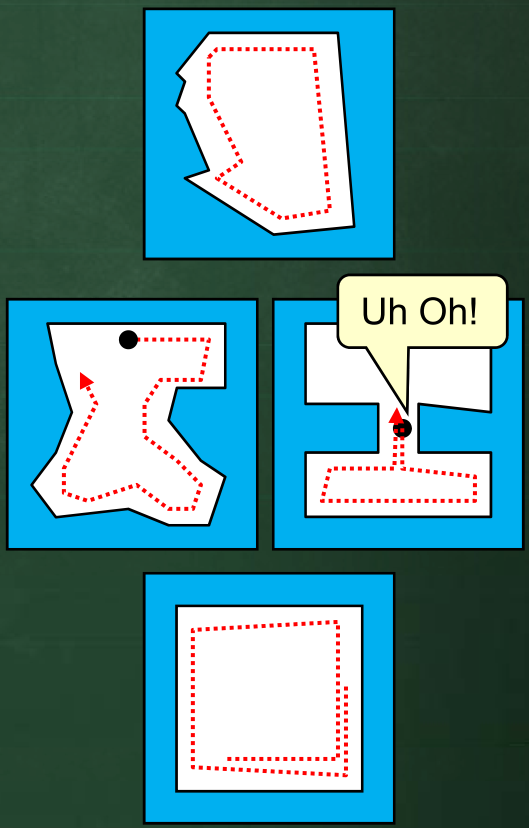
- The strategies are:
  - Feature-Based Navigation
  - Potential Field Navigation
  - Vector Field Histograms (already discussed)

Path Planning in Known Environments

Feature-Based Navigation

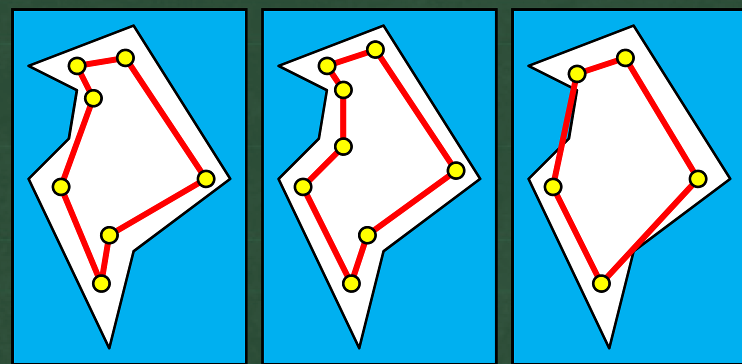
Obtaining Feature Maps

- Use edge following to get trace of an obstacle.
- Problem: How do we know once an obstacle has been completely traced ?
  - If edges are unique in size, can look for previously encountered edge. (impractical)
  - Can use a marker such as a disk. (can lead to hardware issues)
  - Consider maximum perimeter and stop tracing after that (may result in over mapping of an obstacle).

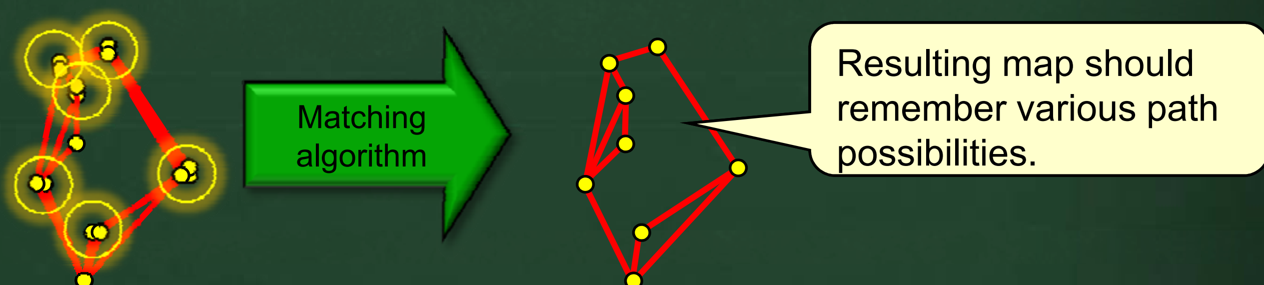


Obtaining Feature Maps

- What do we do about differences during trace ?

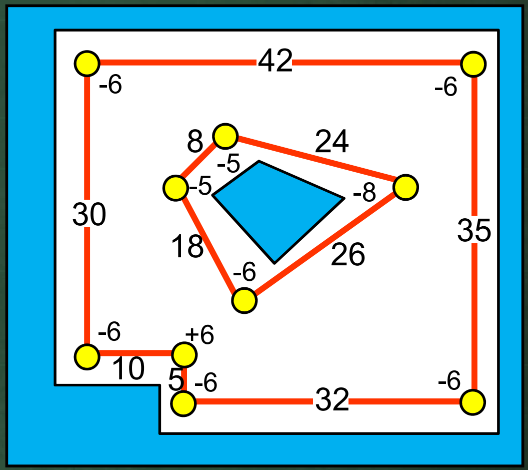


- Must have a way of matching edges and corners
  - Need a measure as to which sequences of edge lengths and corners are to be considered "close enough" or the same.

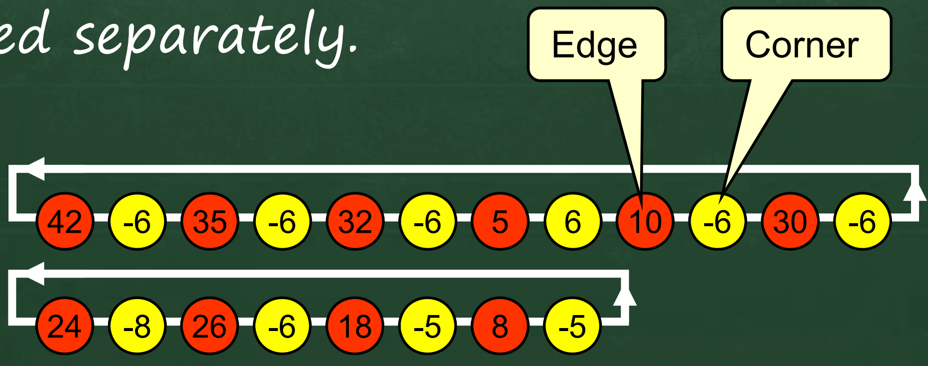


Feature Map Storage

- How do we store feature maps ?
  - Can store integers representing an obstacle's edges and corners as encountered during a trace.

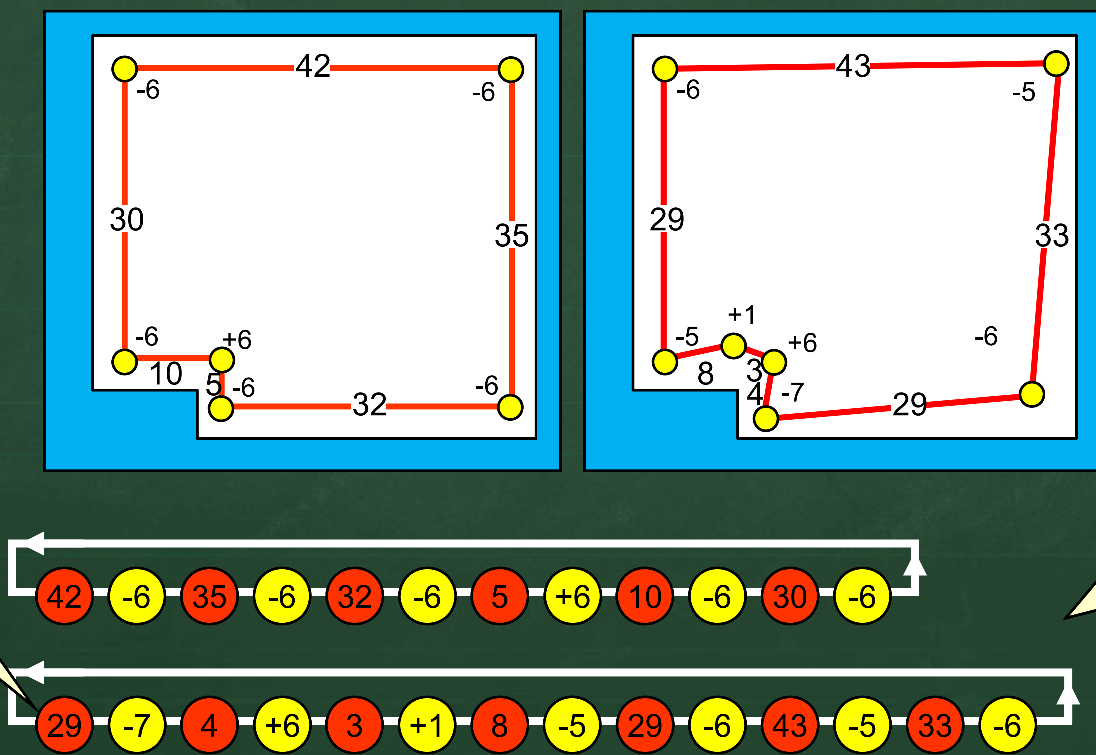


  - Can store as a graph (starting as a circular linked list) or as a network of neurons.
  - Each object stored separately.



Matching Features

- How do we match edges ? Consider two traces of the same environment:

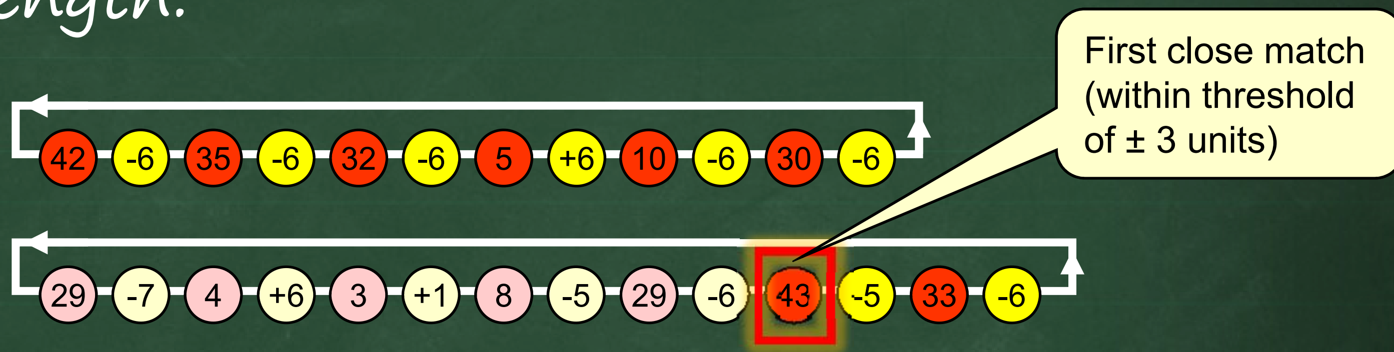


Usually, a different trace begins with a different edge.

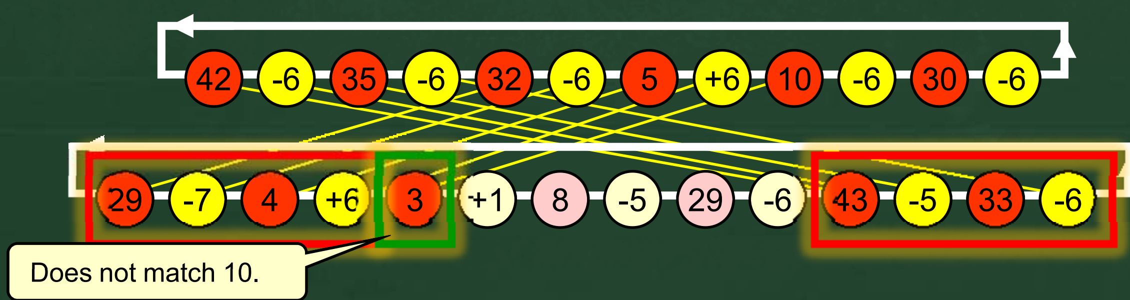
Although not shown here, the traces may be in reverse order (e.g., one CW, the other CCW).

Matching Features

- Begin by searching for the first edge of the new trace within the current map. Find one that is within a threshold length:

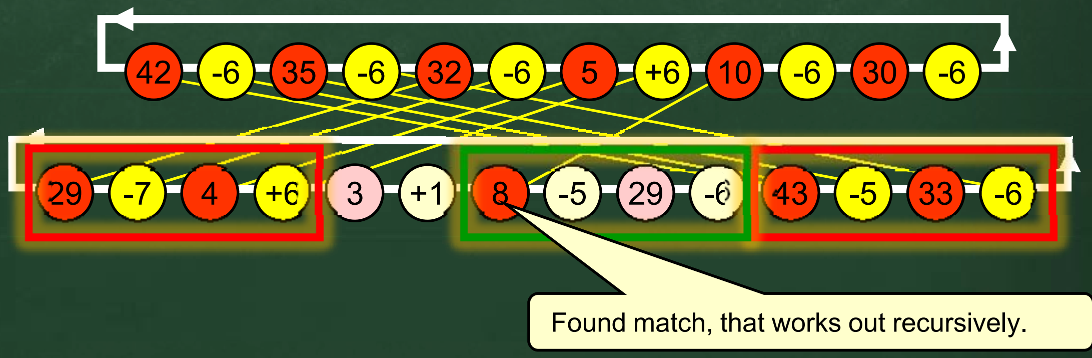


- Check consecutive angles and edges for matches until entire match found or a discrepancy arises:



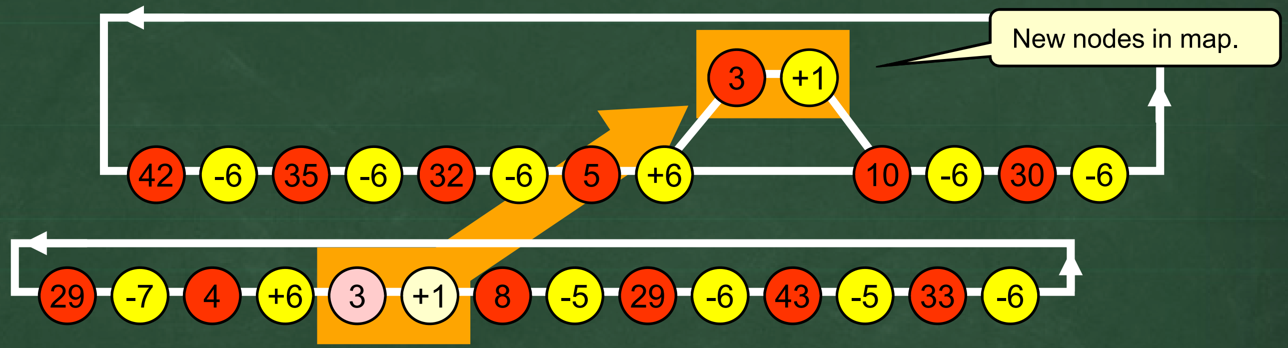
Matching Features

- When this occurs, one of two situations arise:
  - Either the new trace is providing new edges/corners or
  - The new trace did not recognize the turns that were in the original map.
- In the 1st case, we must find a match for the 10 by ignoring the 3, but remembering the skipped edges & corners up until a recursively successful match for 10 is found.

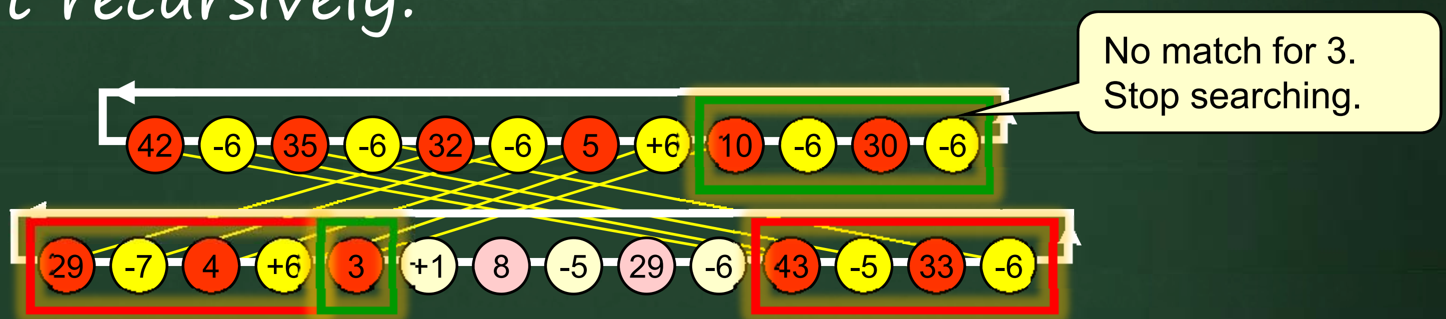


Matching Features

- Now insert the new edges into the map:

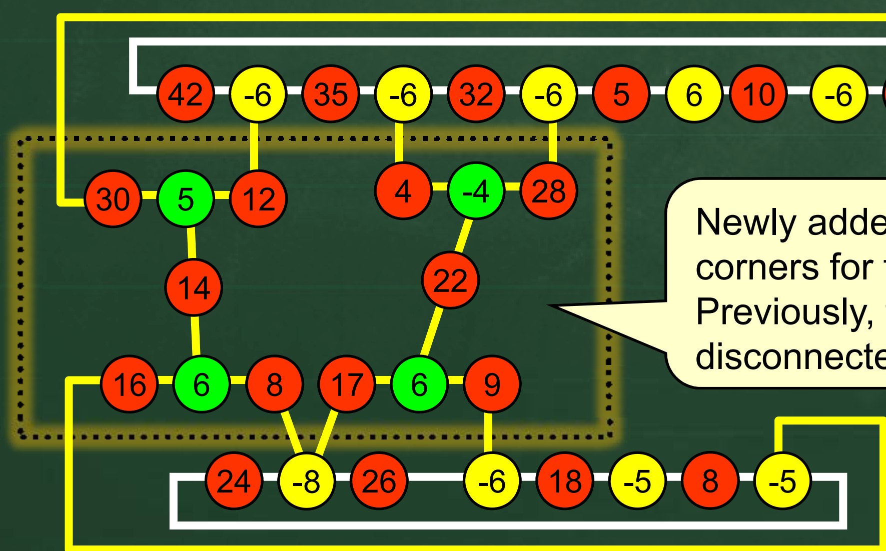


- In the 2nd case, we must try to find a match for the 3 and if one is found, determine if the rest works out recursively:

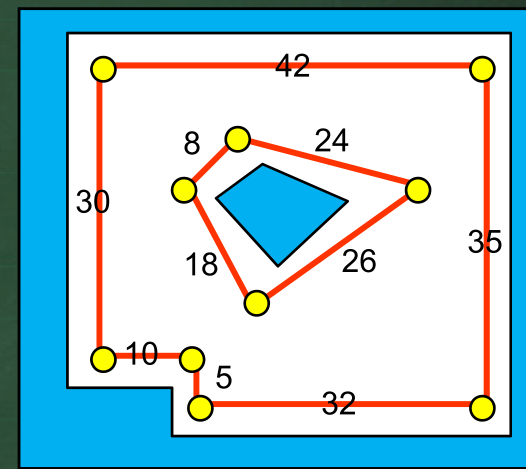


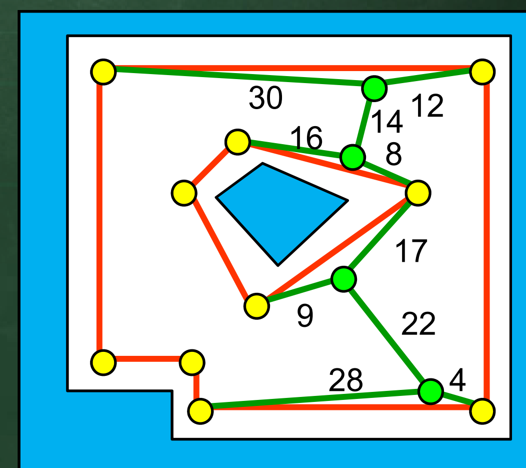
Adding Bridges

- What about bridge/shortcut edges ?
- They can also be stored in the graph as edges joining obstacles:



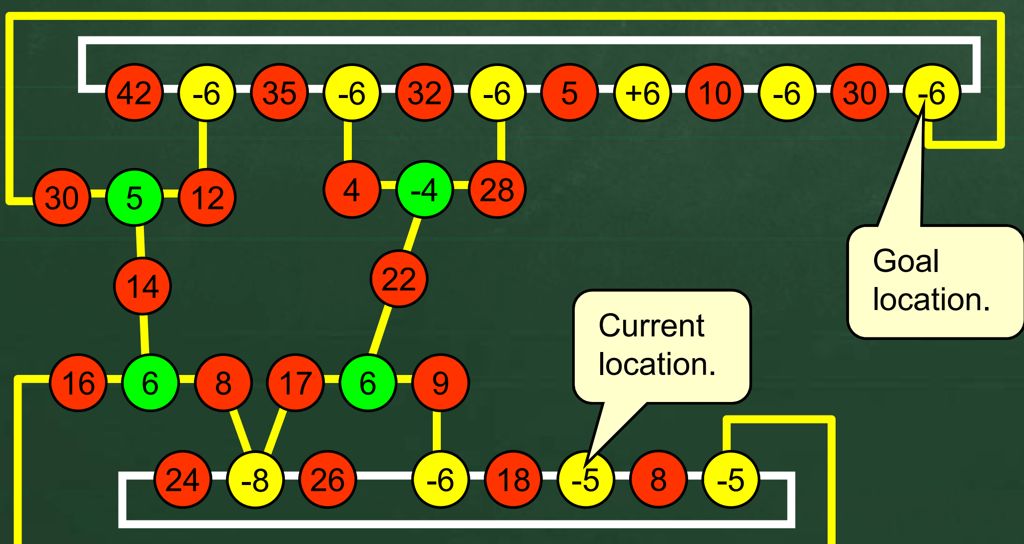
Newly added edges and corners for two bridges. Previously, the graph was disconnected.

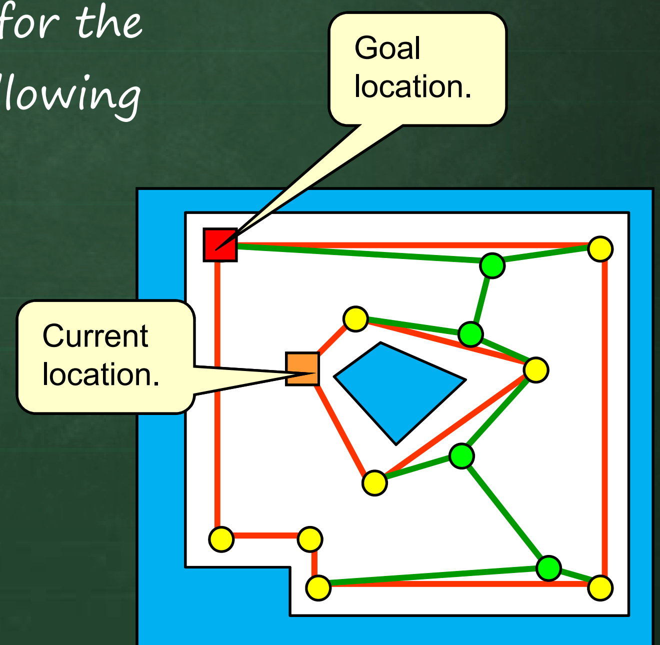




Navigating Feature Maps

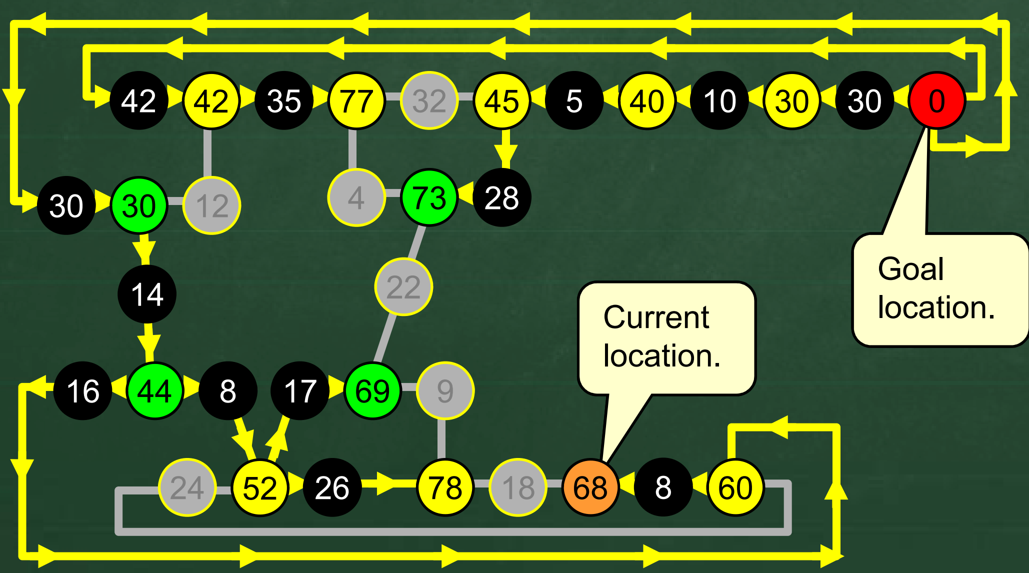
- Once a feature map has been created, the robot can navigate the map by searching the graph.
  - One way of navigating from the current location to the goal is to search the graph, examining edge lengths, for the shortest path, then head in that direction by following either CW or CCW around the obstacle.

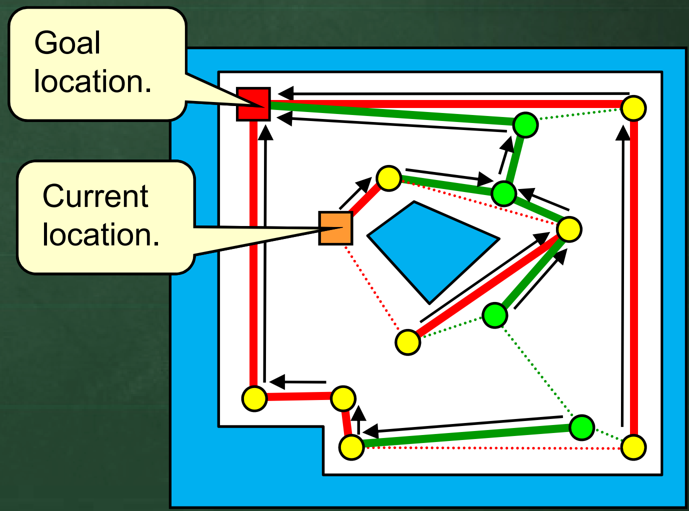




Navigating Feature Maps

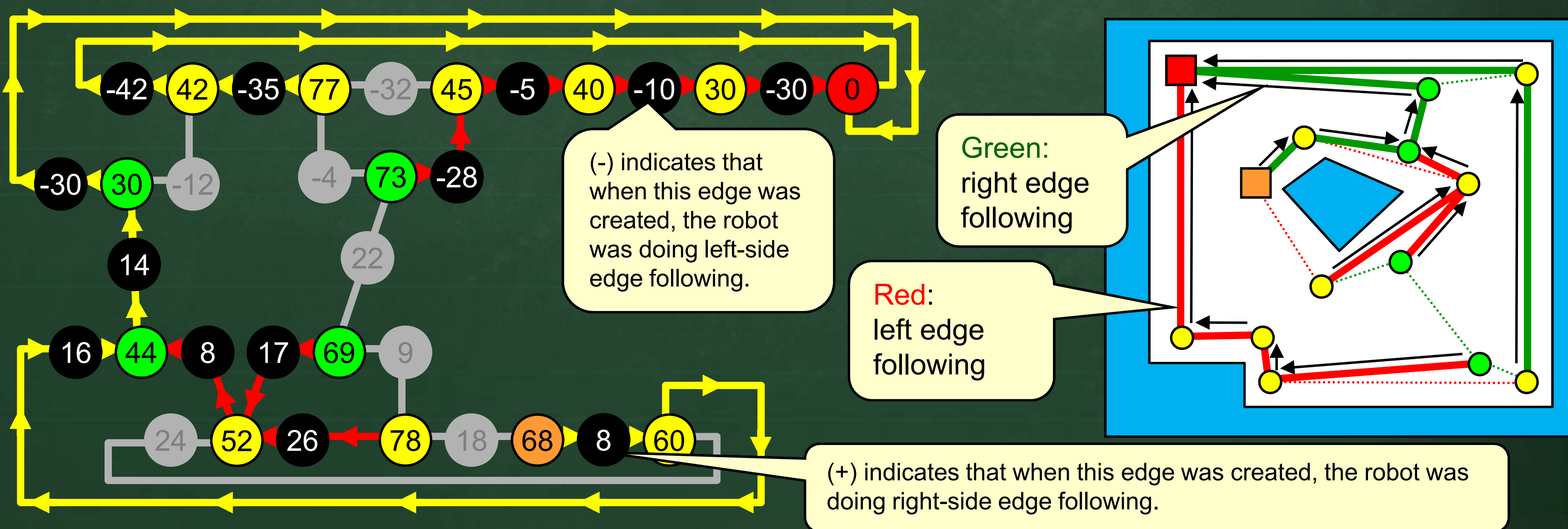
- Start with zero at the goal location and travel outwards, maintaining the cost from each vertex to the goal by adding the edge lengths.





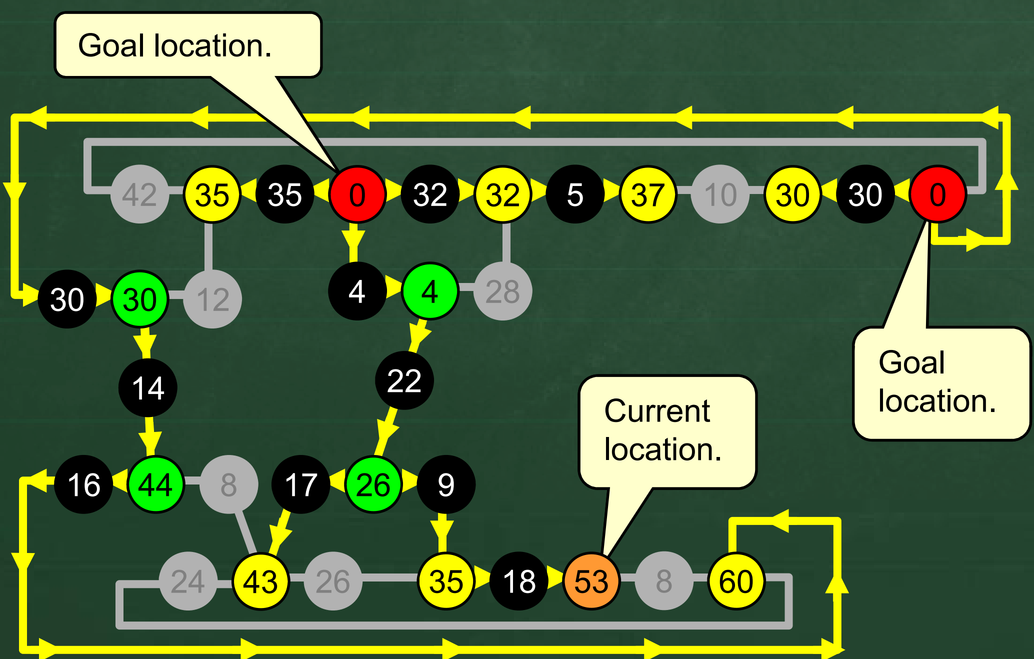
Navigating Feature Maps

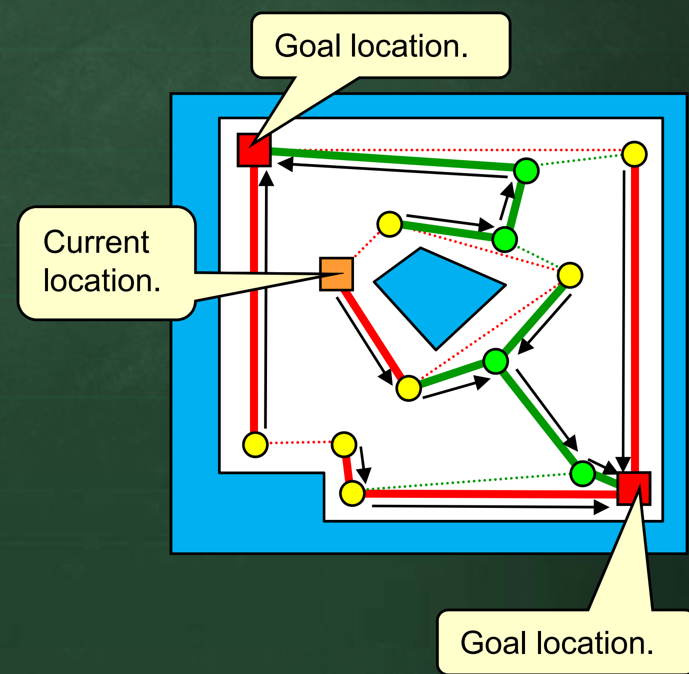
- When standing at a vertex, how do we know which way to go ? (i.e., follow edge on left or right)
  - store for each edge sign indicating whether the edge was on the left (-) or right (+) of the robot when created.
  - Make sure to use absolute value when computing costs



Navigating Feature Maps

- Works with multiple goal scenarios (e.g., multiple recharge stations).
  - Just need to set cost of each goal to zero and propagate outwards again as before.





Path Planning in Known Environments

Potential Field Navigation

Potential Fields

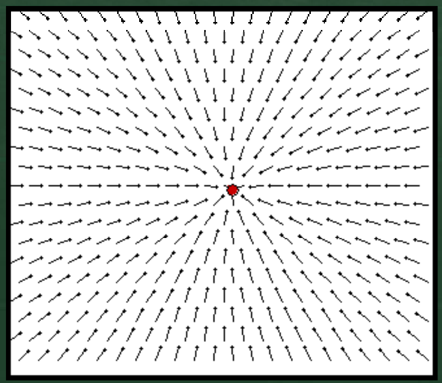
- *Potential field* navigation is based on the idea that environmental objects and locations present *forces* (like magnetic fields) that tend to attract or repel the robot as it moves around in the environment.

- To navigate:
  - the robot computes a *vector* which is a function of its desired goal location as well as the obstacles in the environment.

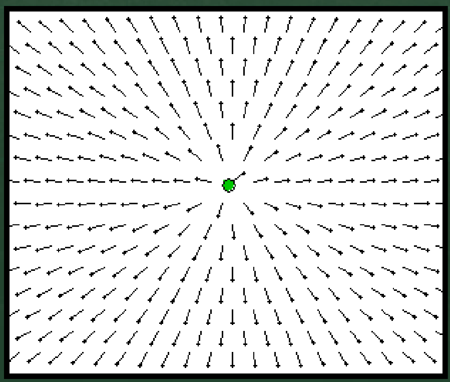
  - The robot heads in the direction of that vector until the goal location is reached, each time computing a new vector.

Potential Fields

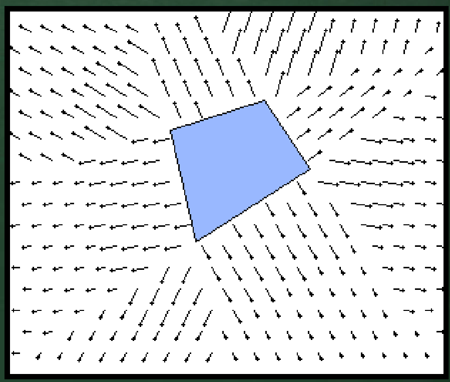
- While there are many "ways" in which potential fields can affect a robot's trajectory, we will examine three basic ones:
  - Attracting towards a goal point (i.e., a sense of where to go)
  - Repelling from a source point (i.e., don't stand still)
  - Repelling from an obstacle (i.e., steering around obstacles)



Attract to goal



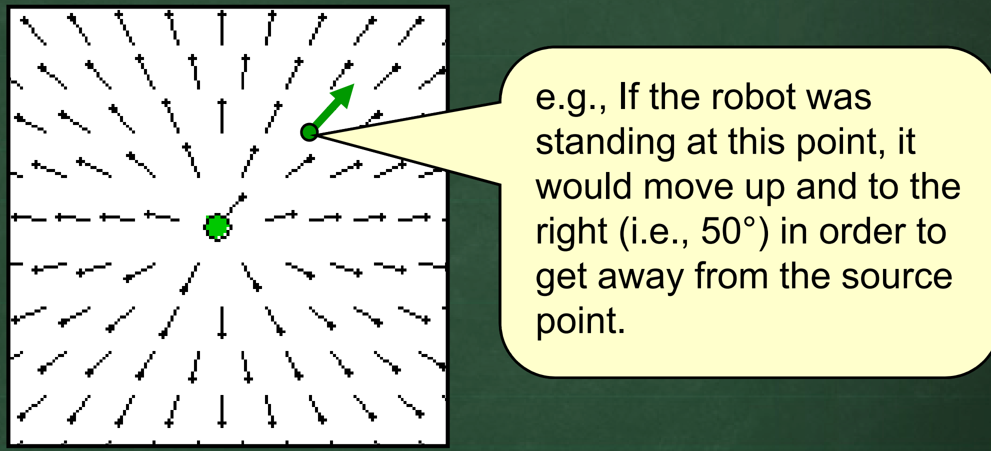
Repel from source



Repel from obstacle

Potential Fields

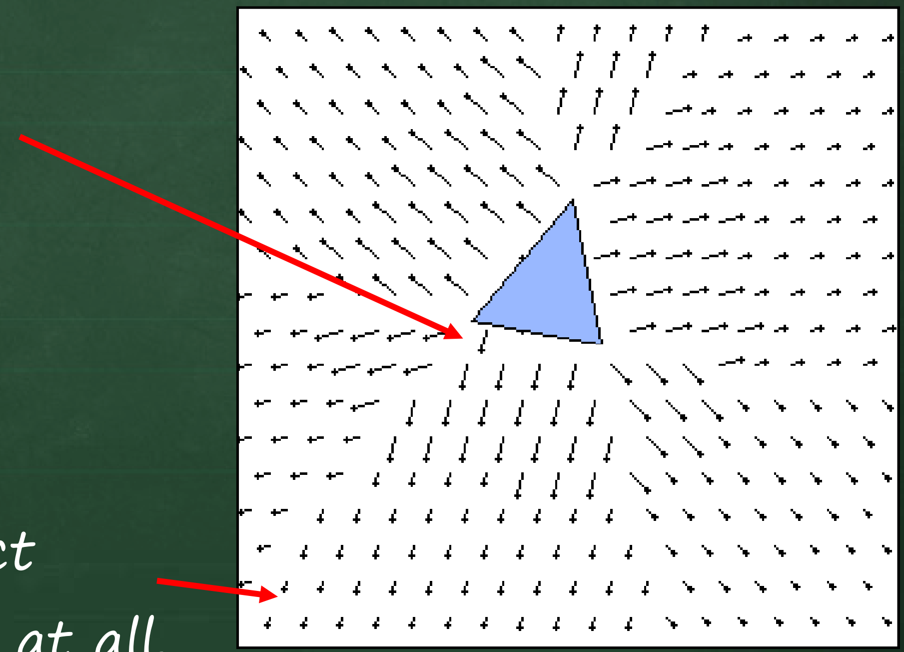
- Each individual vector indicates the direction that the robot should go if it were standing at this particular location.



  - Note that many vectors are displayed to show how the potential fields affect all areas of the environment.
  - In reality though, only one vector is computed based on the robot's location

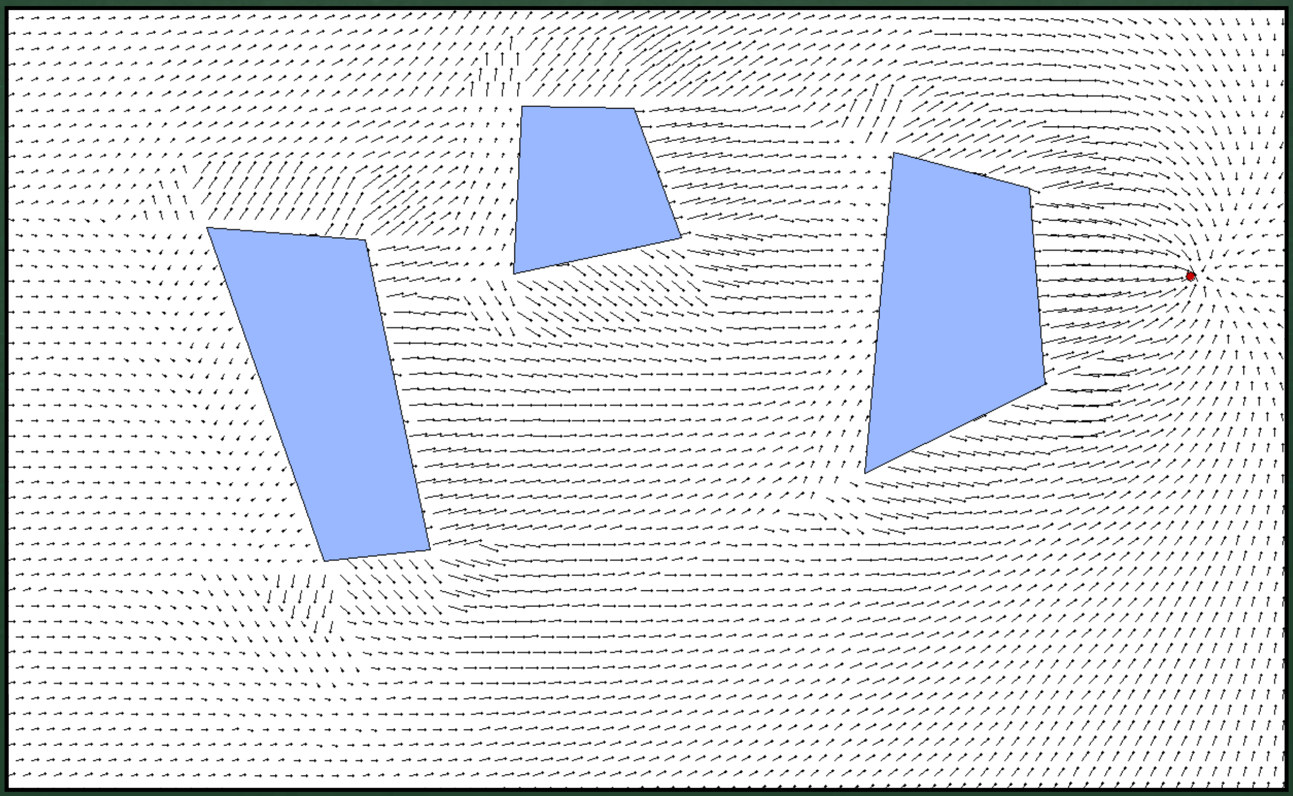
Magnitude

- The magnitude of a potential field vector indicates the "*importance*" of heading in that direction.
  - For example, if the robot is close to an obstacle, it should have a strong desire to move away from it.
  - Likewise if it is far from an obstacle, then this obstacle's potential field does not affect the robot's navigation much at all.



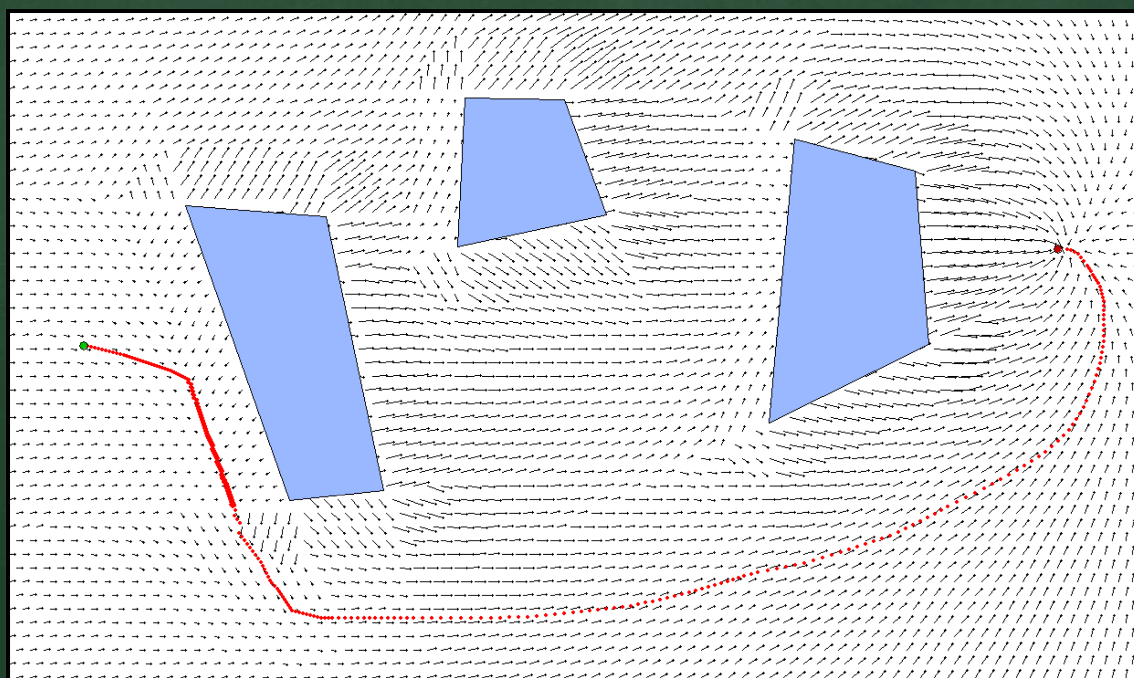
Combining Fields

- The potential field vectors are all combined to achieve an overall field that allows the robot to navigate around obstacles towards a goal.



Combining Fields

- The robot simply calculates and then heads in the direction of a new ***direction vector*** at each location, based on the potential fields that affect its location at that time.

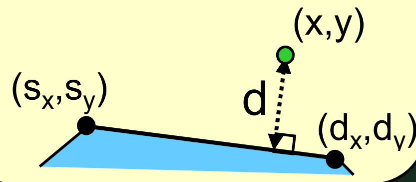


Computing Potential Fields

- How do we calculate the potential field vector for a particular (x, y) location ?
  - Start with vector for goal location (g_x, g_y):
    - magnitude = fixed α_{goal} or
      α_{goal} * 1 / distance from (x,y) to (g_x,g_y)
    - direction = angle from (x,y) to (g_x,g_y)
  - Add to it, a vector for each edge $(s_x,s_y) \rightarrow (d_x,d_y)$ of each obstacle:
    - magnitude = α_{obst} / (d / ($\varepsilon \cdot \alpha_{obst}$) + 1)
    - direction = perpendicular (i.e., 90°) from edge

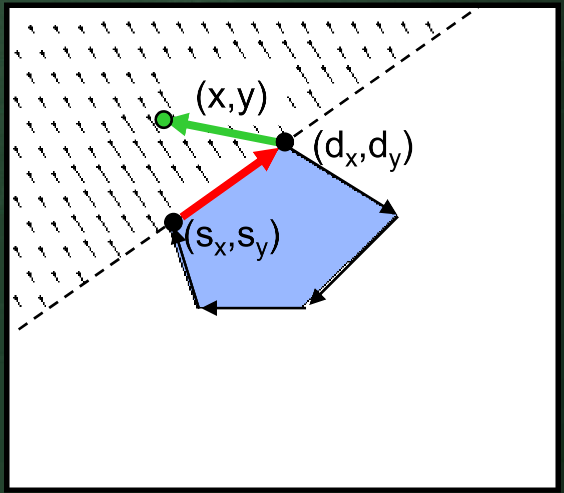
Factor affecting growth of magnitude vectors as distance from edge increases.

d = distance from (x,y) to line $(s_x,s_y) \rightarrow (d_x,d_y)$



Computing Potential Fields

- Each edge separates the plane into two.
- Should ensure that (x,y) only affected if it lies on the half of the plane that does not contain the obstacle
  - Assume obstacle created clockwise, then perform a ***turn test***, only adding potential field vectors for edges to the left of each individual edge.
  - IF $(s_x,s_y) \rightarrow (d_x,d_y) \rightarrow (x,y)$ is a left turn THEN the obstacle should produce a potential field vector for this edge.



Computing Potential Fields

- Compute the *turn type* as the sign of the following determinant:

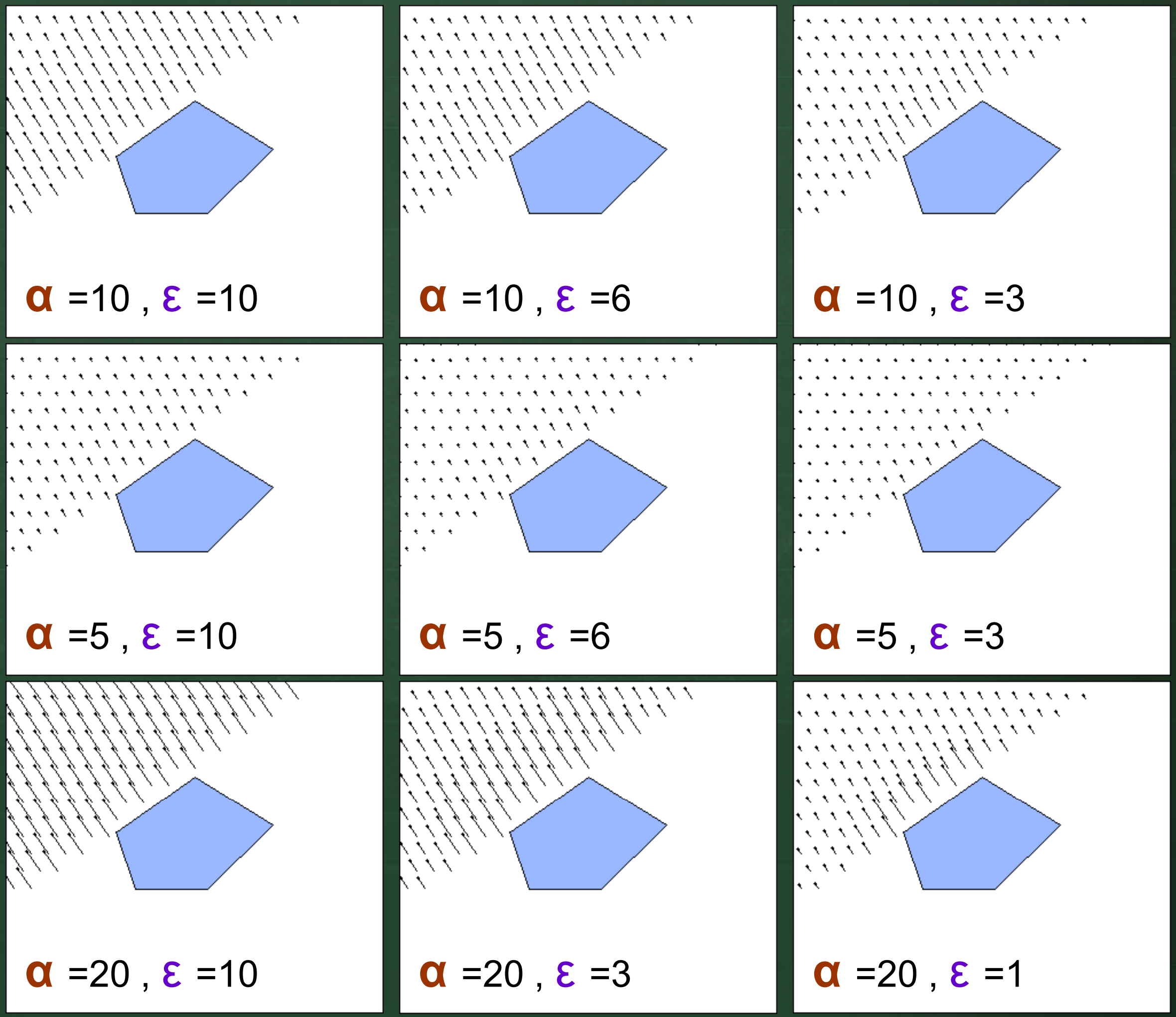
$$\begin{vmatrix} s_x & s_y & 1 \\ d_x & d_y & 1 \\ x & y & 1 \end{vmatrix}$$

  - sign < 0 implies a left turn
  - sign > 0 implies a right turn
  - sign = 0 implies collinear

- Calculate determinant as follows:
  - $s_x d_y + s_y x + d_x y - d_y x - s_y d_x - s_x y$

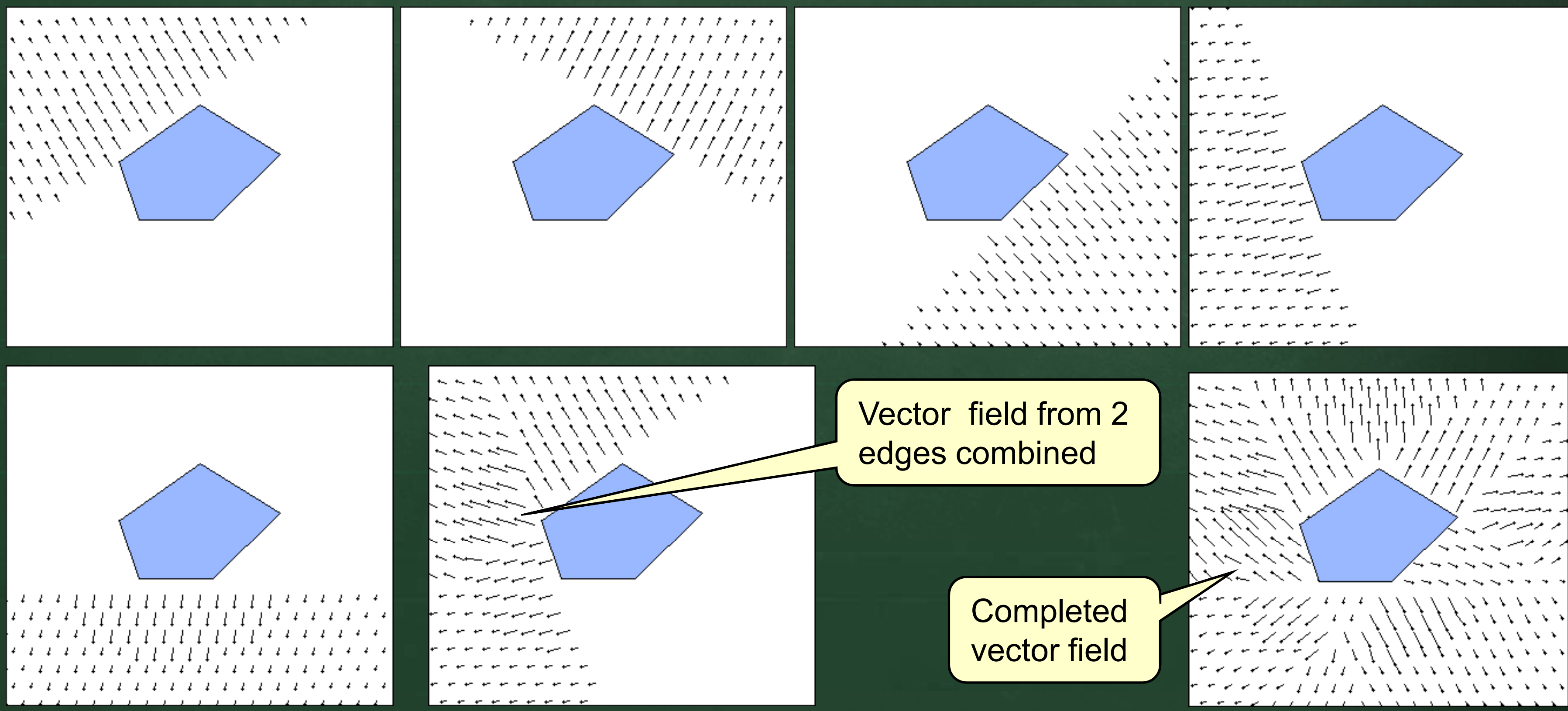
Computing Potential Fields

- Can vary values of α and ε to achieve different field strengths:
  - α indicates maximum magnitude
  - ε indicates magnitude dropoff rate



Computing Potential Fields

- Each edge produces a vector. Edge vectors must be combined.



Computing Potential Fields

- How do we add two vectors ?
  - Can break into x/y components and add those:

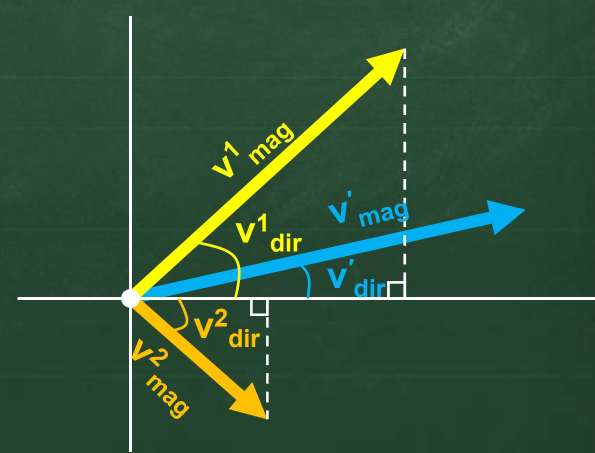
$$c_x = v^1_{mag} \cdot \cos(v^1_{dir}) + v^2_{mag} \cdot \cos(v^2_{dir})$$

$$c_y = v^1_{mag} \cdot \sin(v^1_{dir}) + v^2_{mag} \cdot \sin(v^2_{dir})$$

$$v'_{mag} = \sqrt{c_x^2 + c_y^2}$$

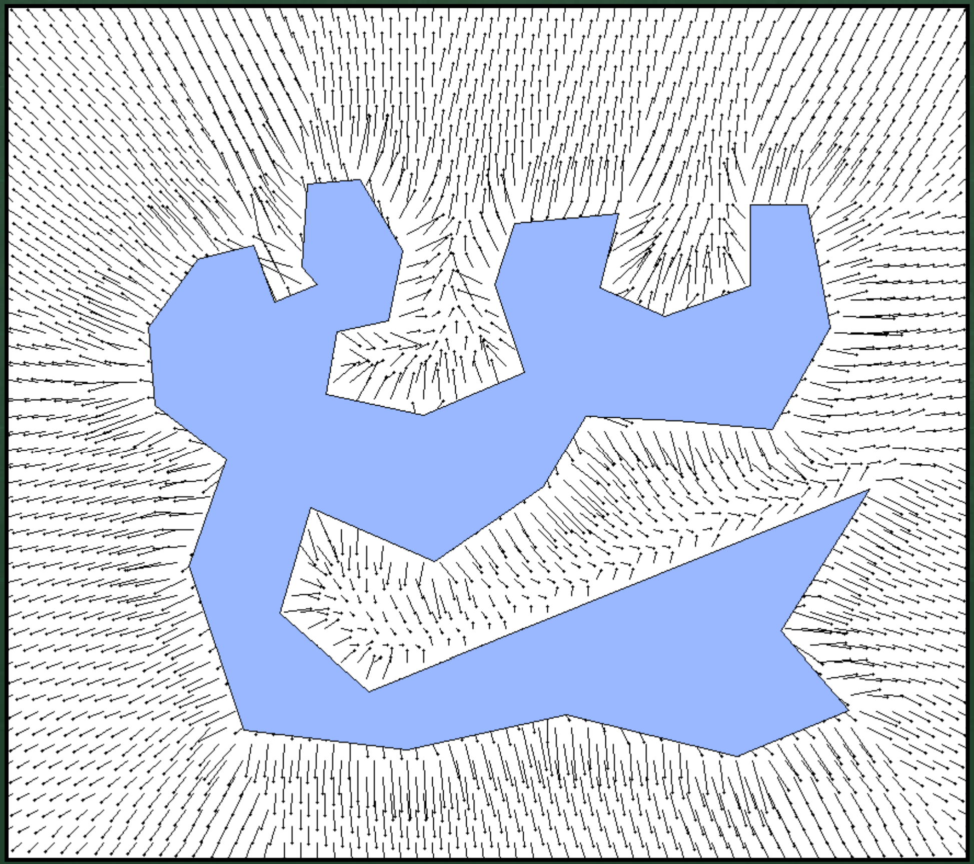
$$v'_{dir} = \text{atan}(c_y / c_x)$$

Need to also add 180° when $c_x < 0$



Computing Potential Fields

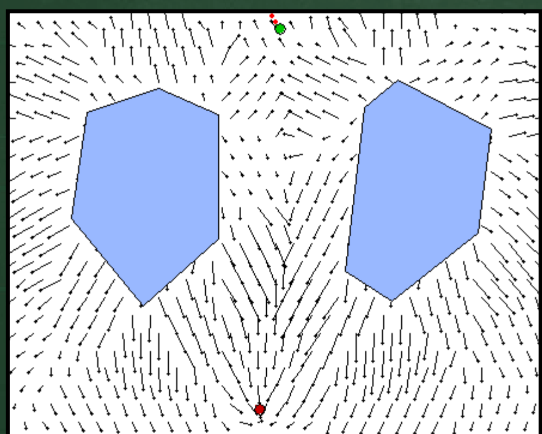
- This left turn test also applies to non-convex obstacles:

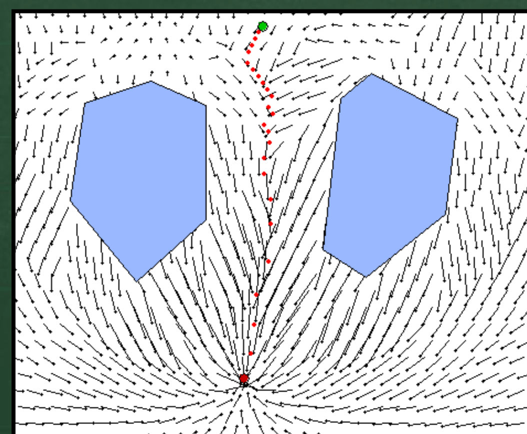


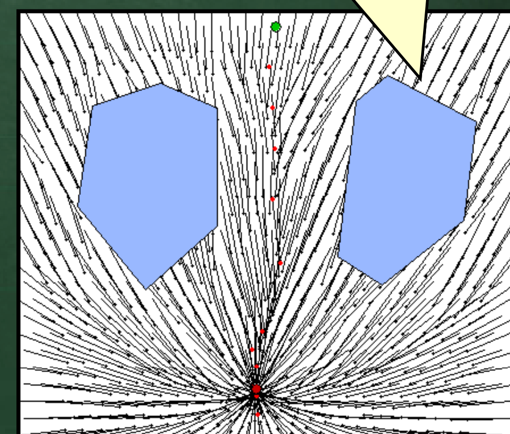
Navigation Problems

- It is not always this straight forward:
  - Sometimes the fields pushing away from obstacles can prevent a solution
  - Depends on strengths of potential fields

If too strong, it may allow or cause collisions.

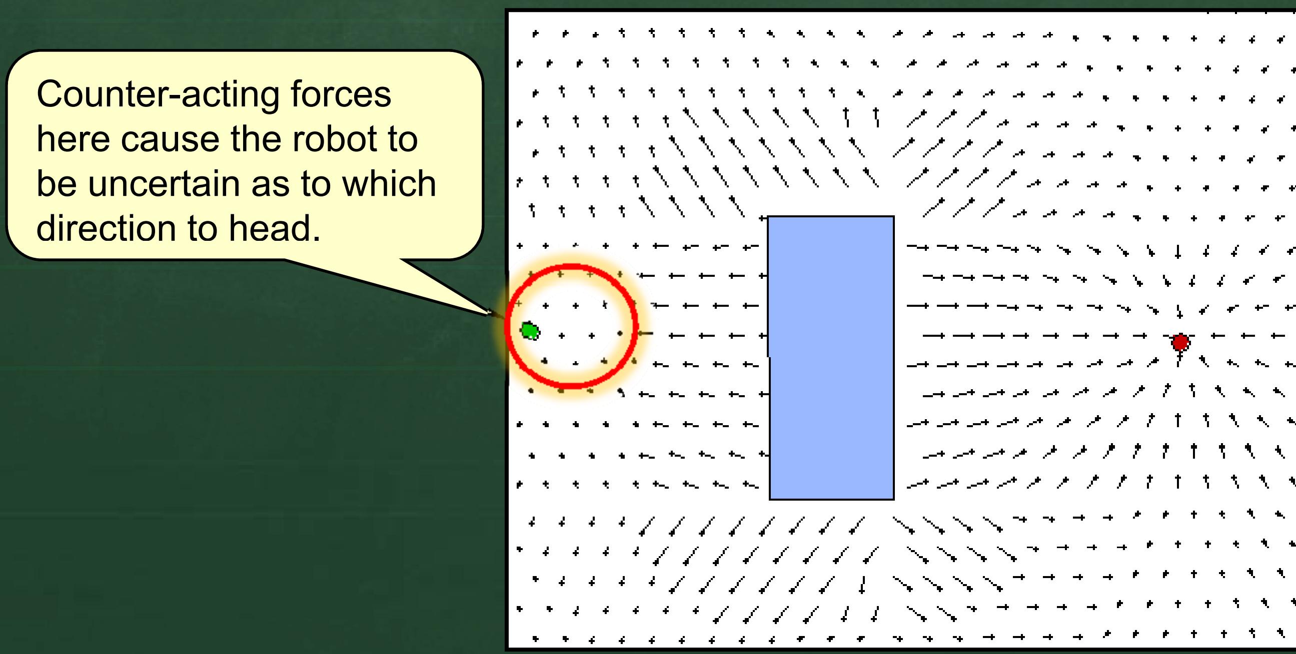

weak goal attraction


medium goal attraction


strong goal attraction

Local Minima

- In some cases, there may be a local minimum problem where the robot gets stuck due to counter-acting forces:

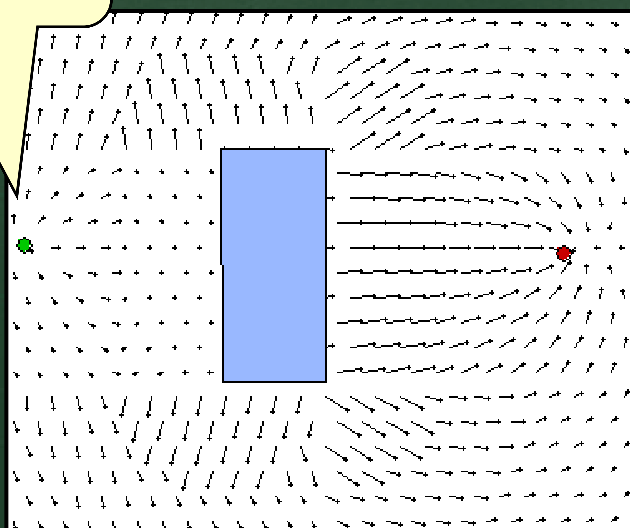


Local Minima

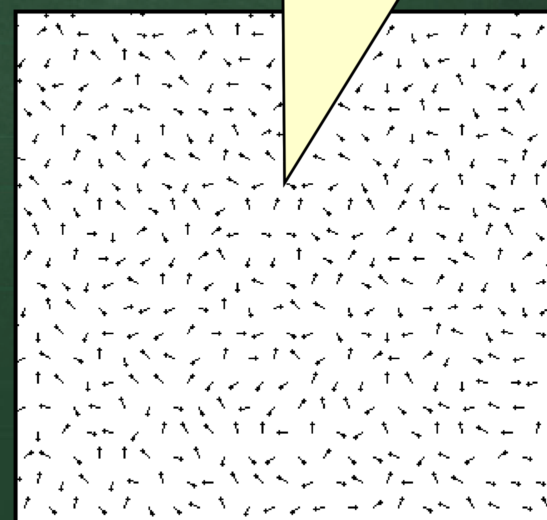
- Can reduce such local minima problems by:
  - adding field from source (i.e., to "push" away from it)
  - introduce ***noise*** into the environment

Source produces vectors to "push" robot outwards.

Usually fixed magnitude, and random offset direction (e.g., ±45°).



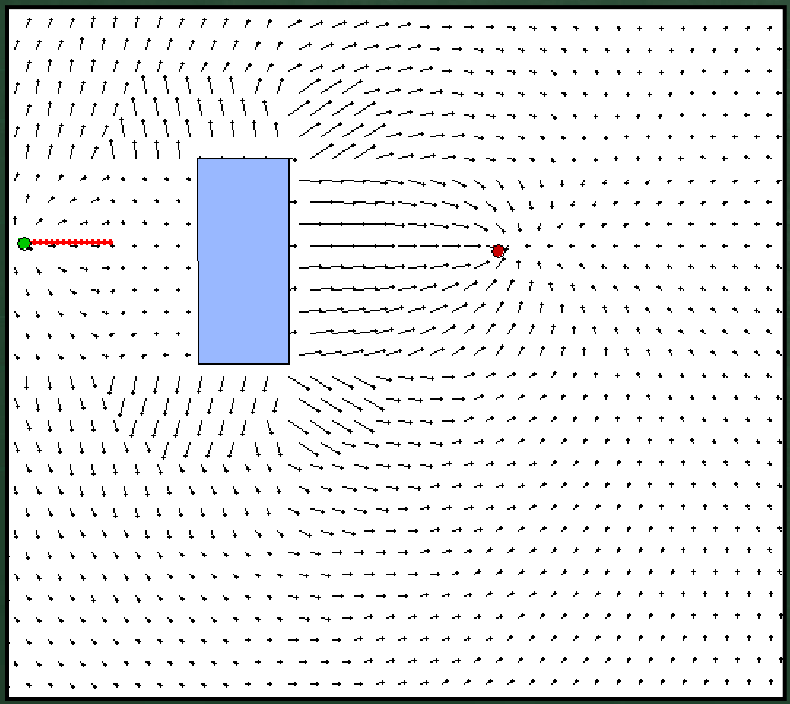
source field *added*



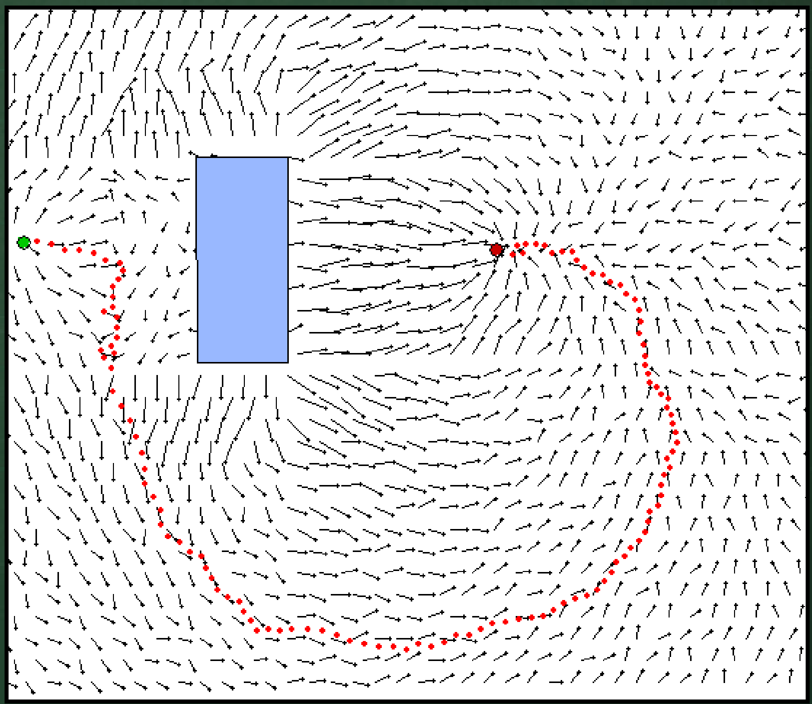
noise (i.e., random vectors)

Local Minima

- The addition of the outwards source field will likely still lead to a local minimum, but added noise often overcomes minimum problem (but no guarantee):



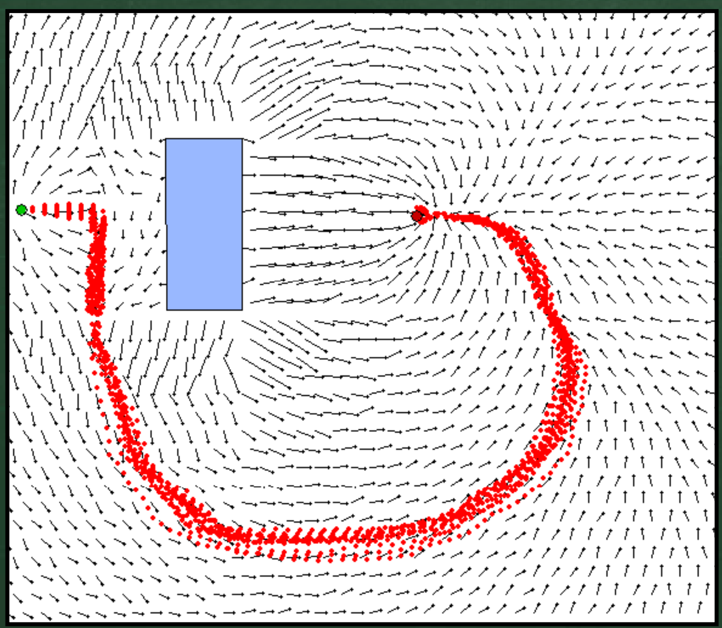
without noise, no path



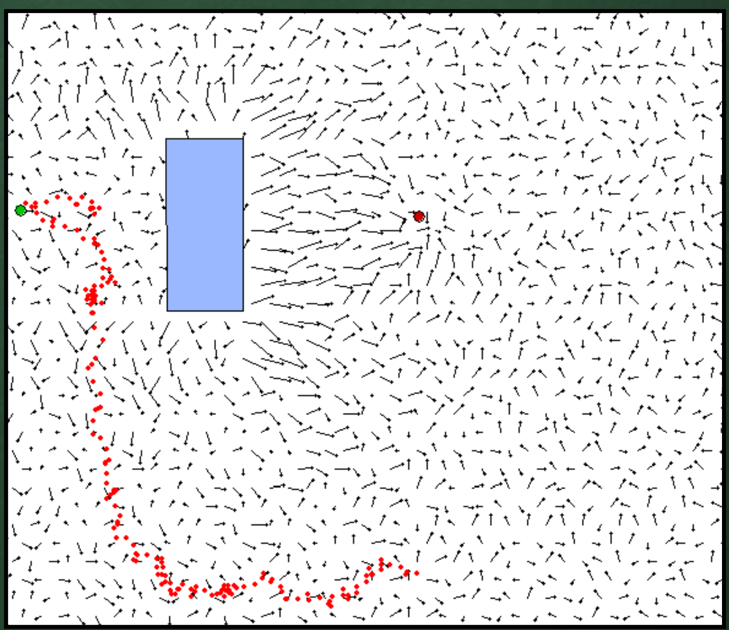
with noise, path found

Local Minima

- During multiple attempts, the path will vary, depending on the random values of the noise vectors.
- Too much noise will not work.



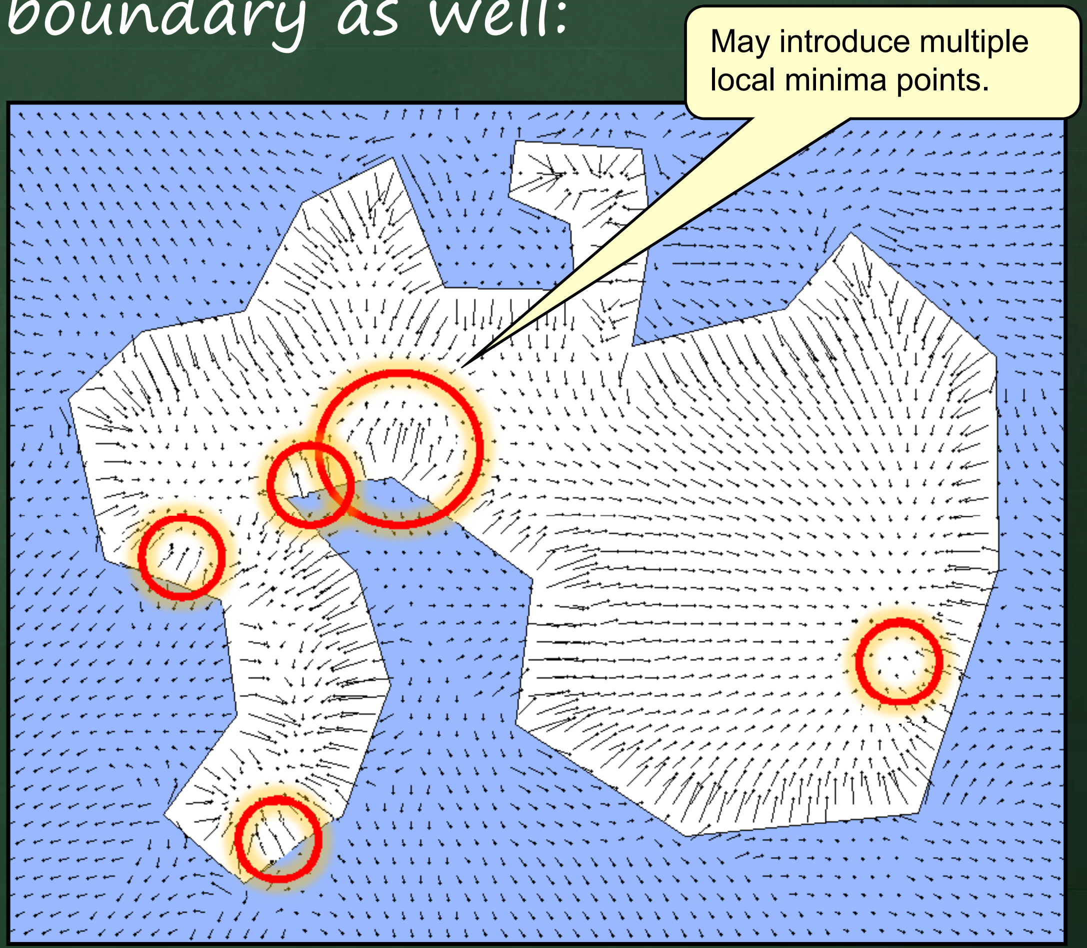
multiple iterations



too much noise, no path found

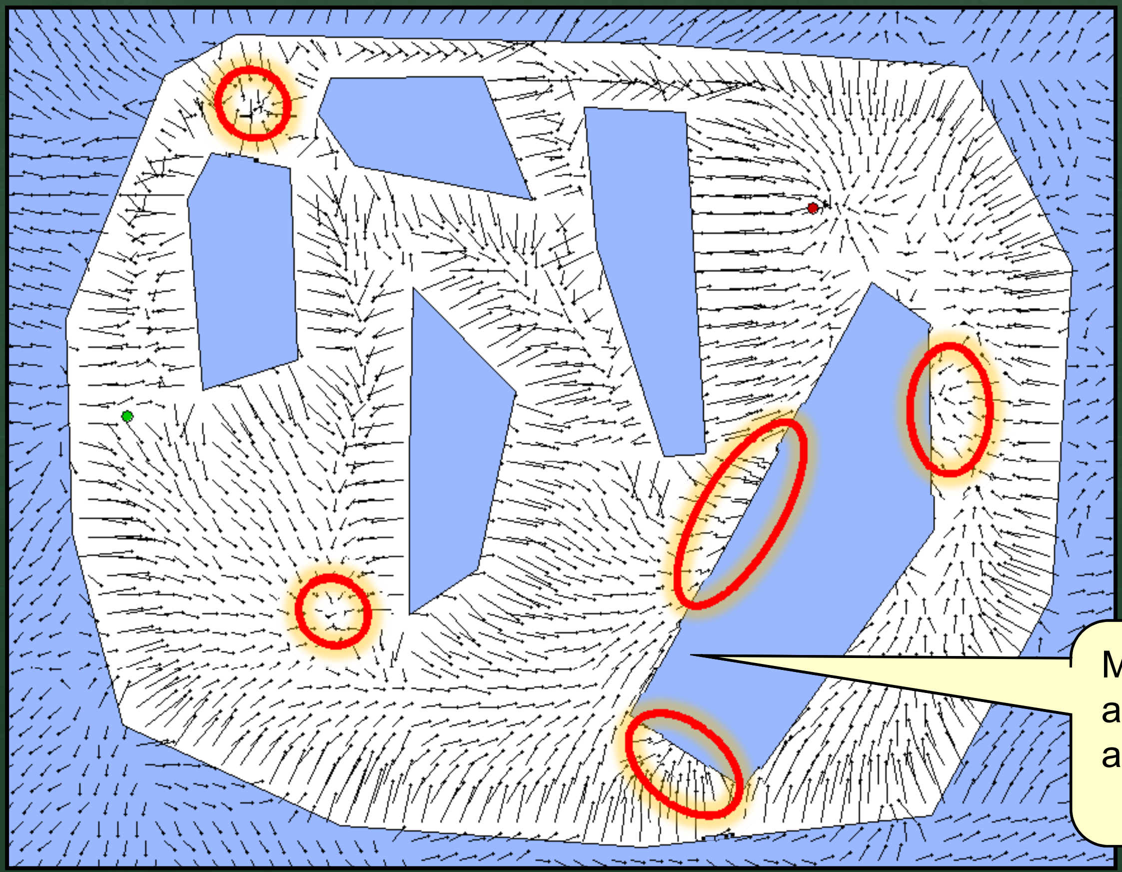
Counteracting Fields

- Certainly, we also want to include vectors from the outer environmental boundary as well:
  - Can keep left turn test, provided that outer boundary is formed in the counter-clockwise direction, otherwise make it a right turn test.



Counteracting Fields

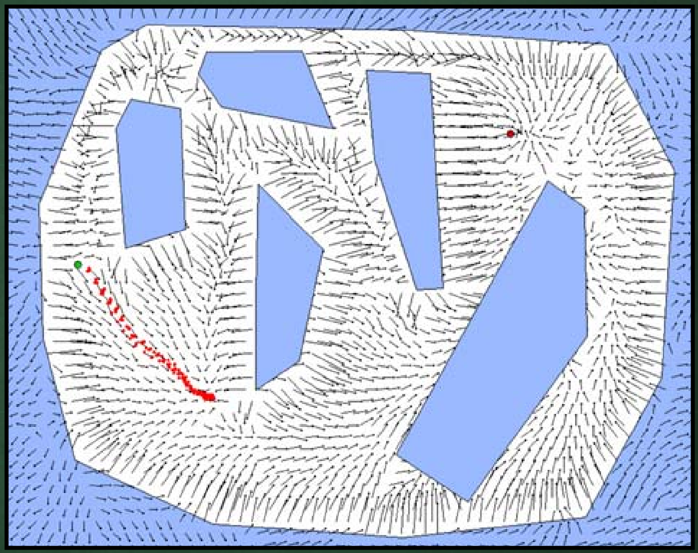
- We combine outer boundary and inner obstacle fields together:



May introduce additional counter-acting problems.

Counteracting Fields

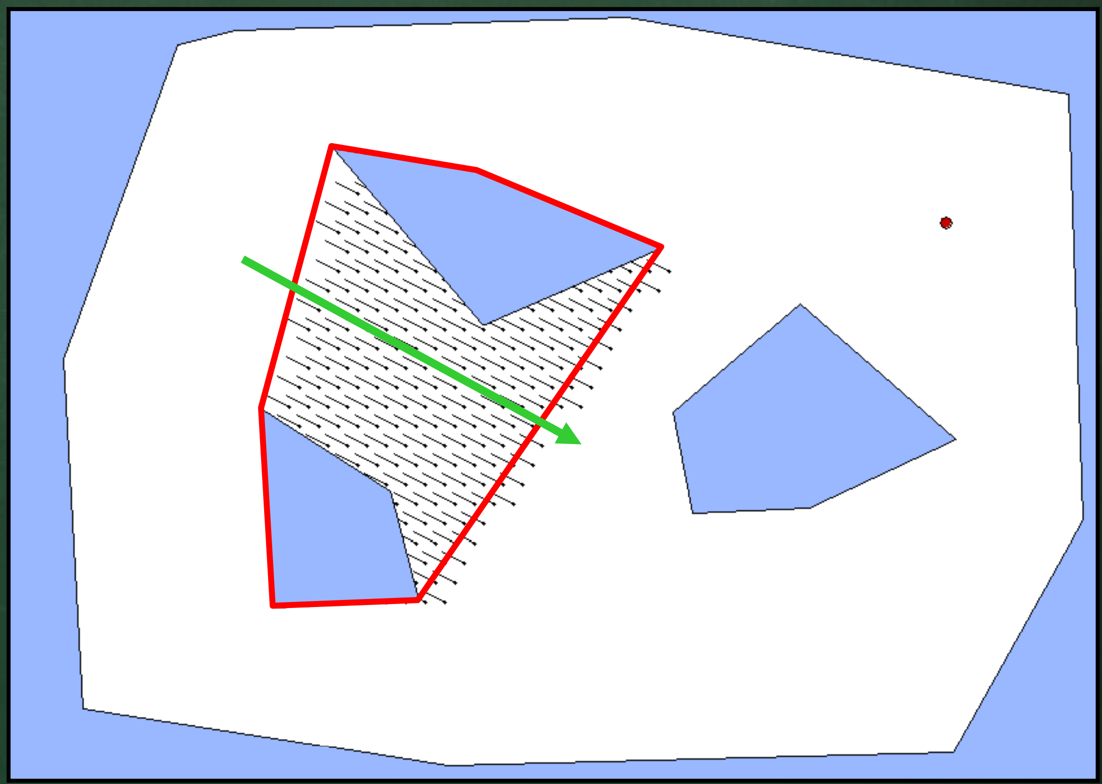
- Still may be no solution, since proper path is NOT a direct path to the goal, but may interleave between obstacles.
- We need a way to "pull" the robot between obstacles in the correct direction from the source and towards the goal.



- Problem: We don't know which way to pull
  - implies that we have some global shortest path knowledge.

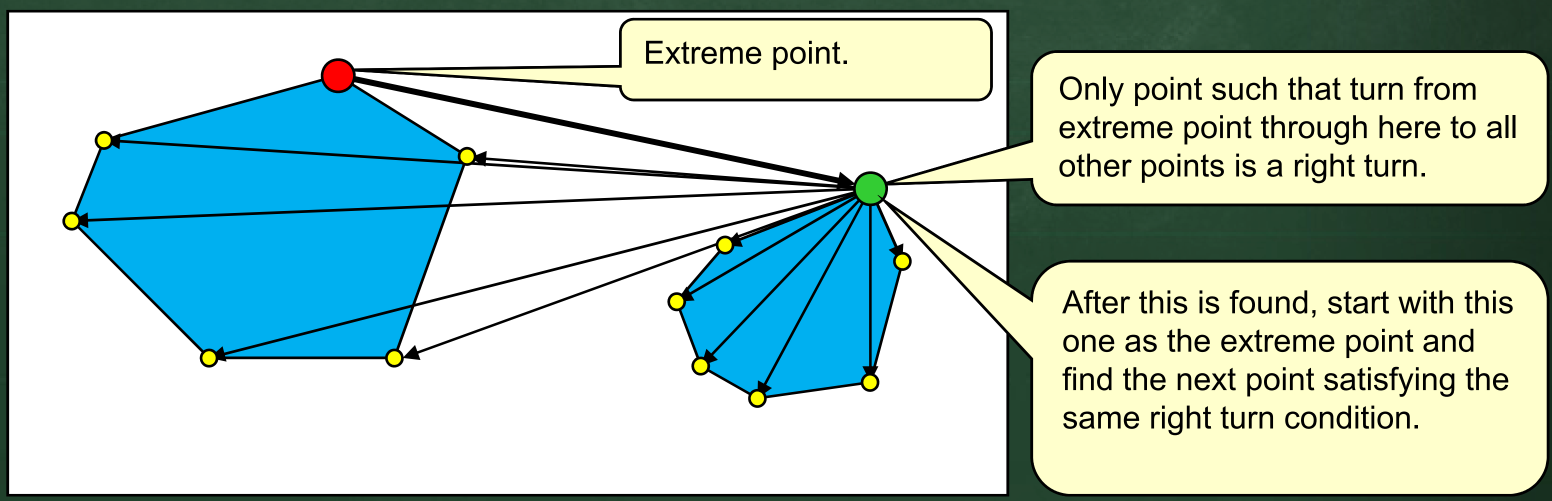
Corridor Fields

- One way of pulling the robot between obstacles is to take pairs of obstacles and for all points in between (i.e., within the convex hull) compute a set of potential fields passing through.
- This presents a flow towards the goal.



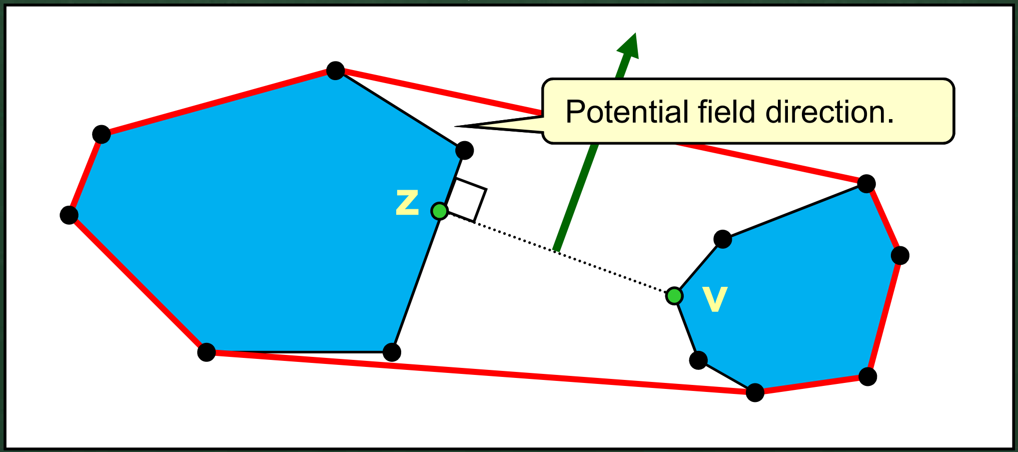
Corridor Fields

- Can compute convex hull of two polygons in many ways. Simplest is to start with an extreme point (e.g., max y value) and finding the next clockwise hull point by choosing any other point (from both polygons) such that the line from the max point and next point to any other point is a right turn:



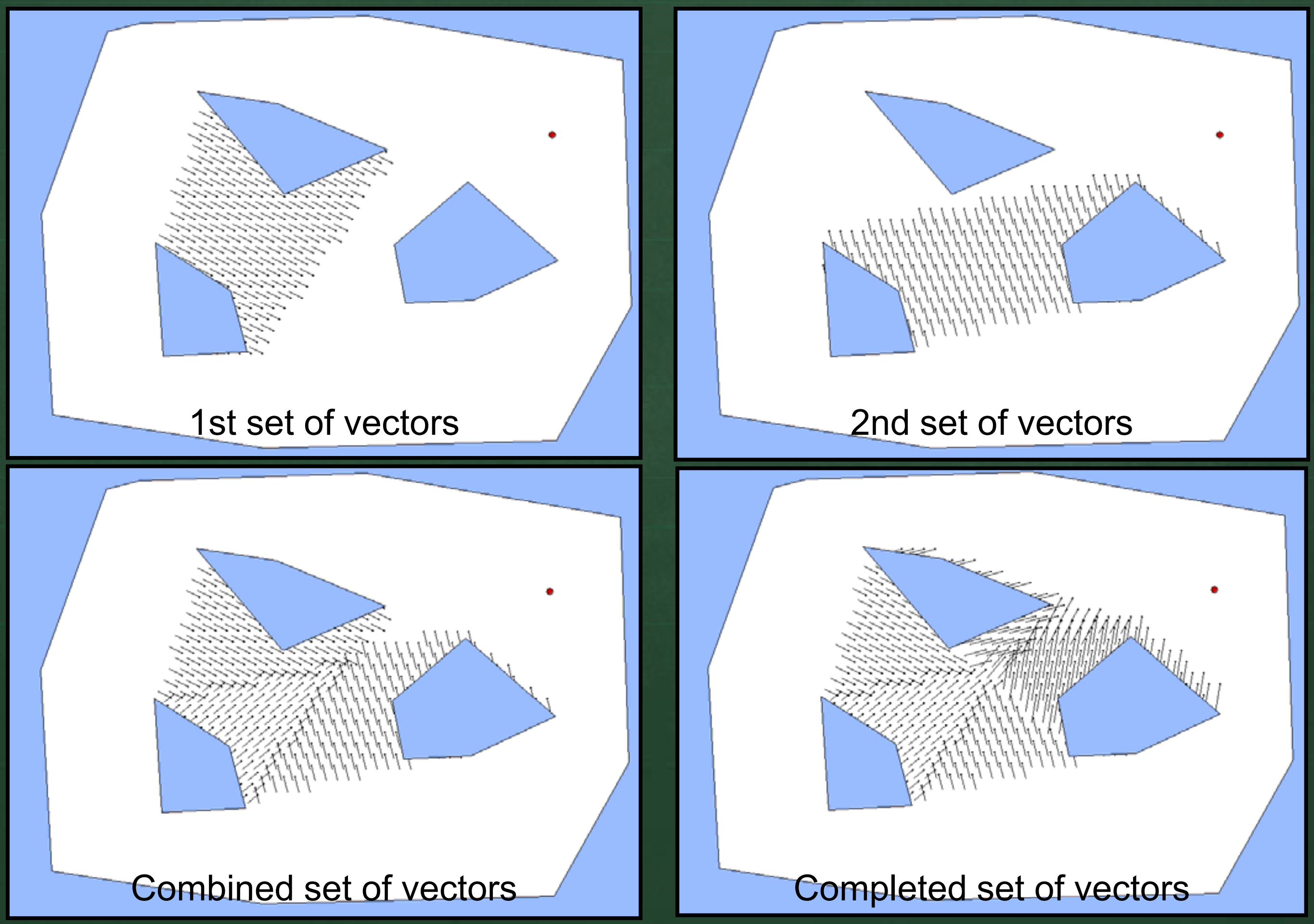
Corridor Fields

- Compute the field direction as follows:
  - Let p_1 and p_2 be the two polygons forming the convex hull.
  - Determine the vertex **v** of p_1 that is closest to p_2, then determine the point **z** on an edge of p_2 that is closest to **v**.
  - Compute the angle of the perpendicular to segment **vz**.
    - There are two choices for the perpendicular direction, depending on which way the robot needs to pass through in order to reach the goal.



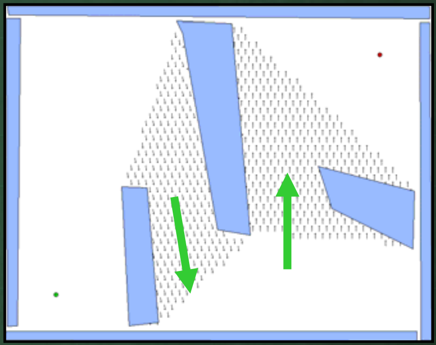
Corridor Fields

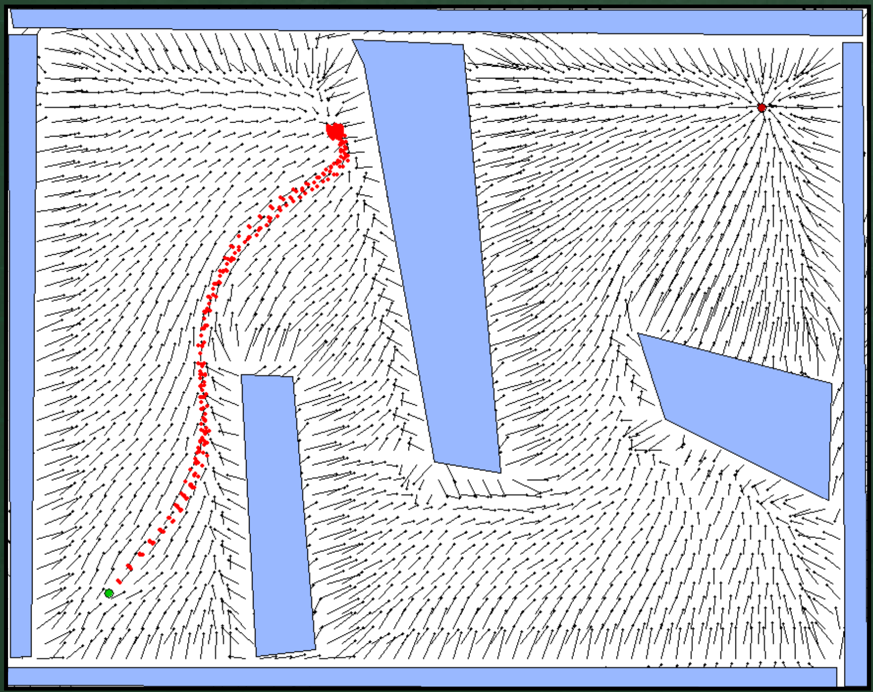
- Here are some such fields produced by this means:



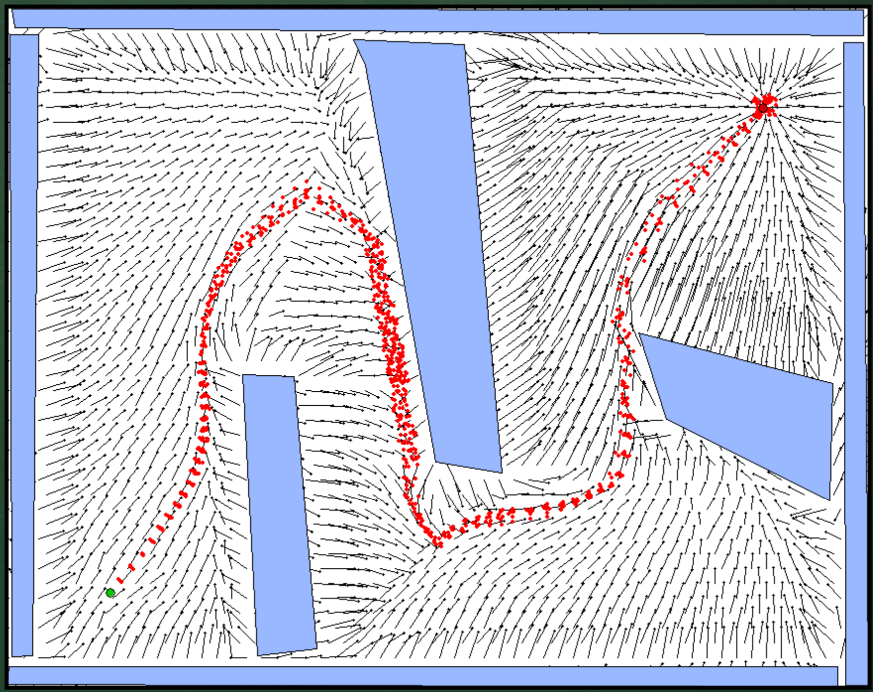
Corridor Fields

- Does this improve the success ?
- Consider using just two of these "corridor" fields:





without corridor fields



with corridor fields

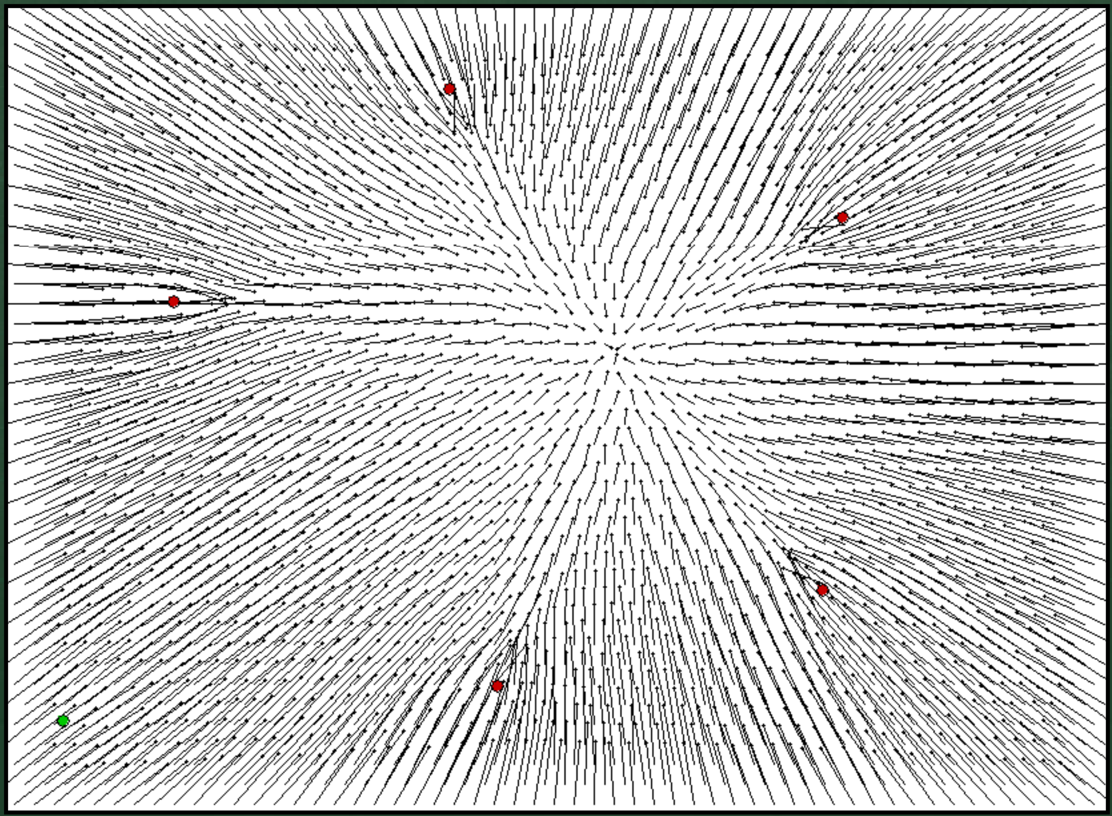
Problems

- Problems:
  - We don't know which direction to pass through the corridor. This depends on some global knowledge.
  - The magnitudes of the vectors used greatly affect the solution. It is difficult to choose appropriate magnitudes for each of the kinds of vector fields:
    - obstacles
    - goal attraction and source repelling
    - noise
    - corridors

  - So, lots of experimentation is needed to find appropriate weights, which highly depends on the obstacle shapes.

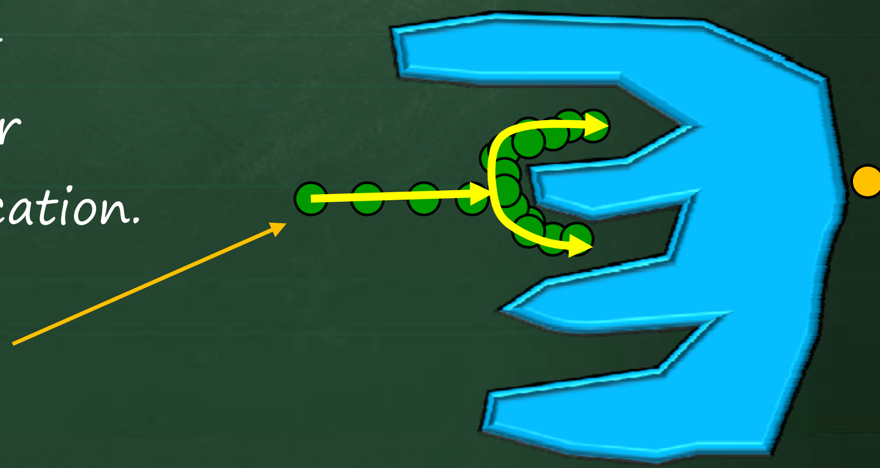
Problems

- Potential Fields cannot handle multiple goals:



Comparing with VFH

- How does this compare with Vector Field Histograms ?
  - VHF can be used in unknown environments, whereas the potential field method requires all obstacle positions.
  - easier for VFH to go down narrow passages.
  - VFH cannot get trapped in a local minimum due to counteracting obstacle and goal forces
    - it always heads towards the best opening, regardless of whether or not it heads towards the goal location.
  - VFH can still get stuck in cycles (like a local minima)



Summary

- You should now understand
  - How to navigate a robot from its current location towards a particular goal.
  - Various navigation techniques which vary according to the knowledge about obstacles and the ability to sense obstacles.
  - How to use a feature map to direct a robot towards a particular location in the map.
  - How to move a robot successfully, but need to consider how to do so more efficiently when more knowledge is available.