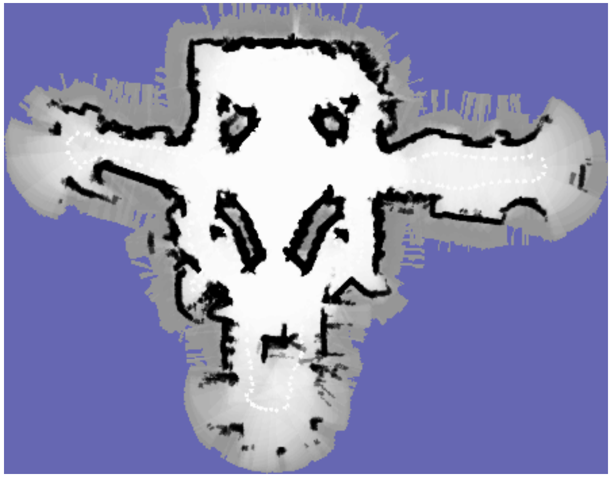
Metric Path Planning

September 23, 2014

Minerva tour guide robot (CMU):
Gave tours in Smithsonian's National Museum of History



Example of Minerva's occupancy map used for navigation

Objectives

- *Path planning*: identifying a trajectory that will cause the robot to reach the goal location when executed

- Understand techniques for metric path planning:
  - Configuration space
  - Meadow maps
  - Generalized Voronoi graphs
  - Grids
  - Quadtrees
  - Graph-based planners: A*
  - Wavefront-based planners

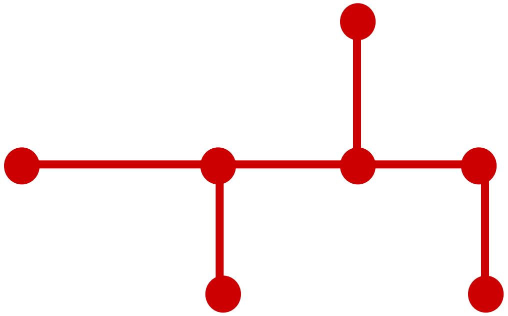
Introduction to Navigation

- Navigation is fundamental ability in autonomous mobile robotics

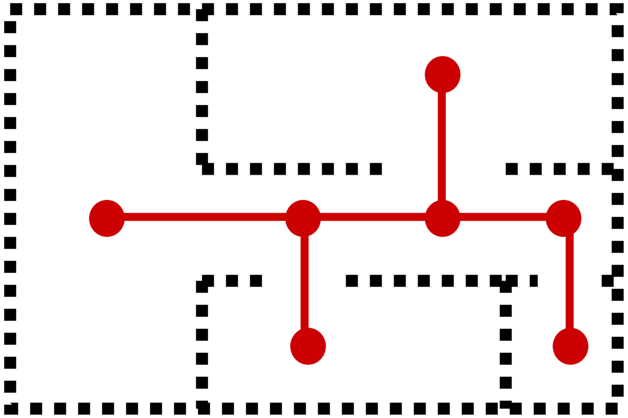
- Primary functions of navigation:
  - Where am I going?
    - Usually defined by human operator or mission planner
  - What’s the best way to get there?
    - Path planning: qualitative and quantitative
  - Where have I been?
    - Map making
  - Where am I?
    - Localization: relative or absolute

Two types of spatial representations commonly used in path planning

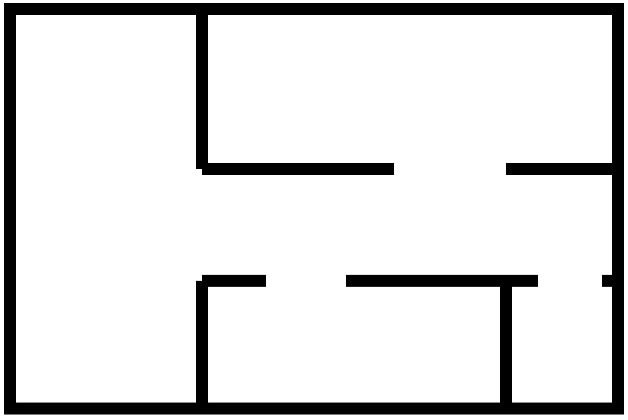
- Examples of two forms of Spatial representations:
  - Qualitative (route) [sometimes called "topological"]:



derived from:



  - Quantitative (metric or layout):



Two types of spatial representations commonly used in path planning (con't.)

- Two forms of Spatial memory:
  - Qualitative (route):
    - Express space in terms of connections between landmarks
    - Dependent upon perspective of the robot
    - Orientation clues are egocentric
    - Usually cannot be used to generate quantitative (metric/layout) representations
  - Quantitative (metric or layout):
    - Express space in terms of physical distances of travel
    - Bird's eye view of the world
    - Not dependent upon the perspective of the robot
    - Independent of orientation and position of robot
    - Can be used to generate qualitative (route) representations

Metric Path Planning

- Objective: determine a path to a specified goal

- Metric methods:
  - Tend to favor techniques that produce an optimal path
  - Usually decompose path into subgoals called waypoints

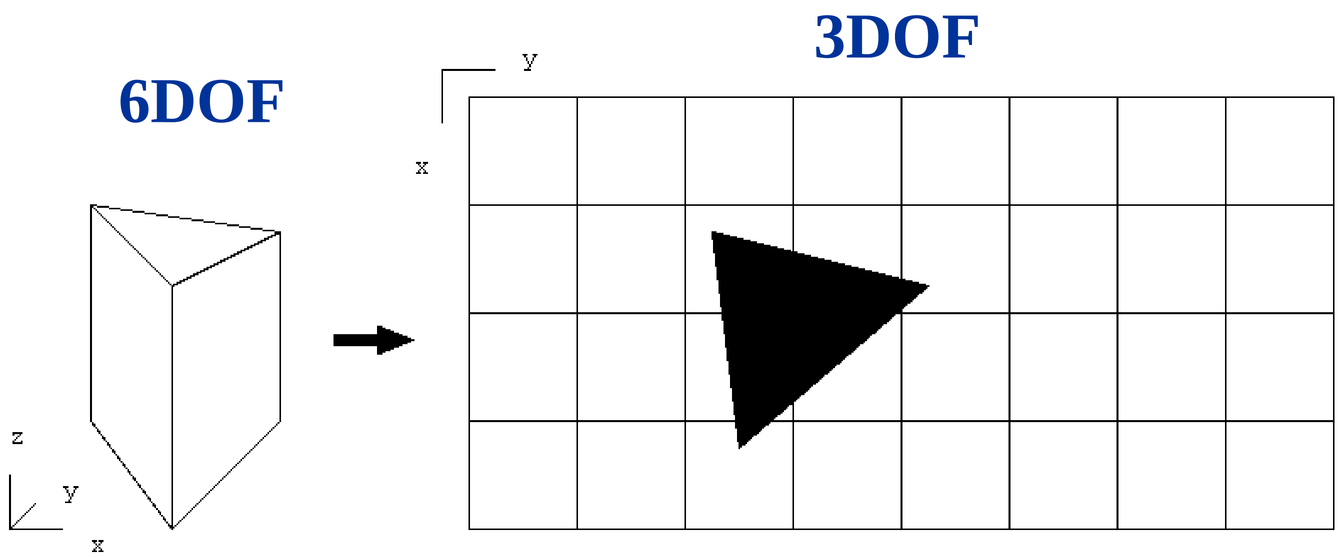
- Two components to metric methods for path planning:
  - Representation (i.e., data structure)
  - Algorithm

Configuration Space

- Configuration Space (abbreviated: “Cspace”):
  - Data structure that allows robot to specify position and orientation of objects and robot in the environment
  - “Good Cspace”: Reduces # of dimensions that a planner has to deal with
  - Typically, for indoor mobile robots:
    - Assume 2 DOF for representation
    - Assume robot is round, so that orientation doesn’t matter
    - Assumes robot is holonomic (i.e., it can turn in place)
      - (Although there is much research dealing with path planning in non-holonomic robots)
  - Typically represents “occupied” and “free” space
    - “Occupied” → object is in that space
    - “Free” → space where robot is free to move without hitting any modeled object

Metric Maps use Cspace

- World Space: physical space robots and obstacles exist in
  - In order to use, generally need to know (x,y,z) plus Euler angles: 6DOF
    - Ex. Travel by car, what to do next depends on where you are and what direction you're currently heading
- Configuration Space (Cspace)
  - Transform space into a representation suitable for robots, simplifying assumptions



Major Cspace Representations

- Idea: reduce physical space to a Cspace representation which is more amenable for storage in computers and for rapid execution of algorithms

- Major types
  - Meadow Maps
  - Generalized Voronoi Graphs (GVG)
  - Regular grids, quadtrees

Object Growing

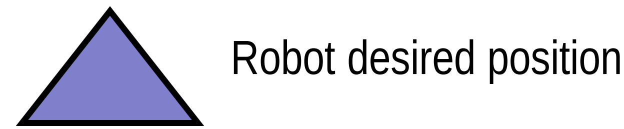
- Since we assume robot is round, we can “grow” objects by the width of the robot and then consider the robot to be a point

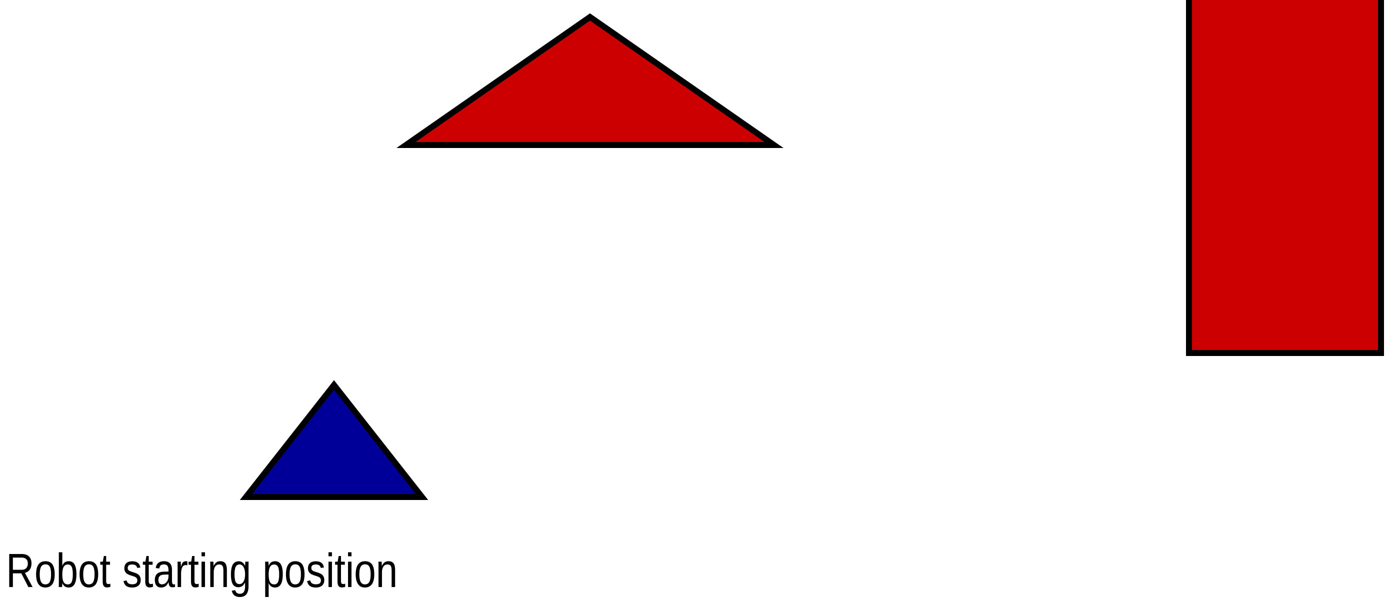
- Greatly simplifies path planning

- New representation of objects typically called “configuration space object”

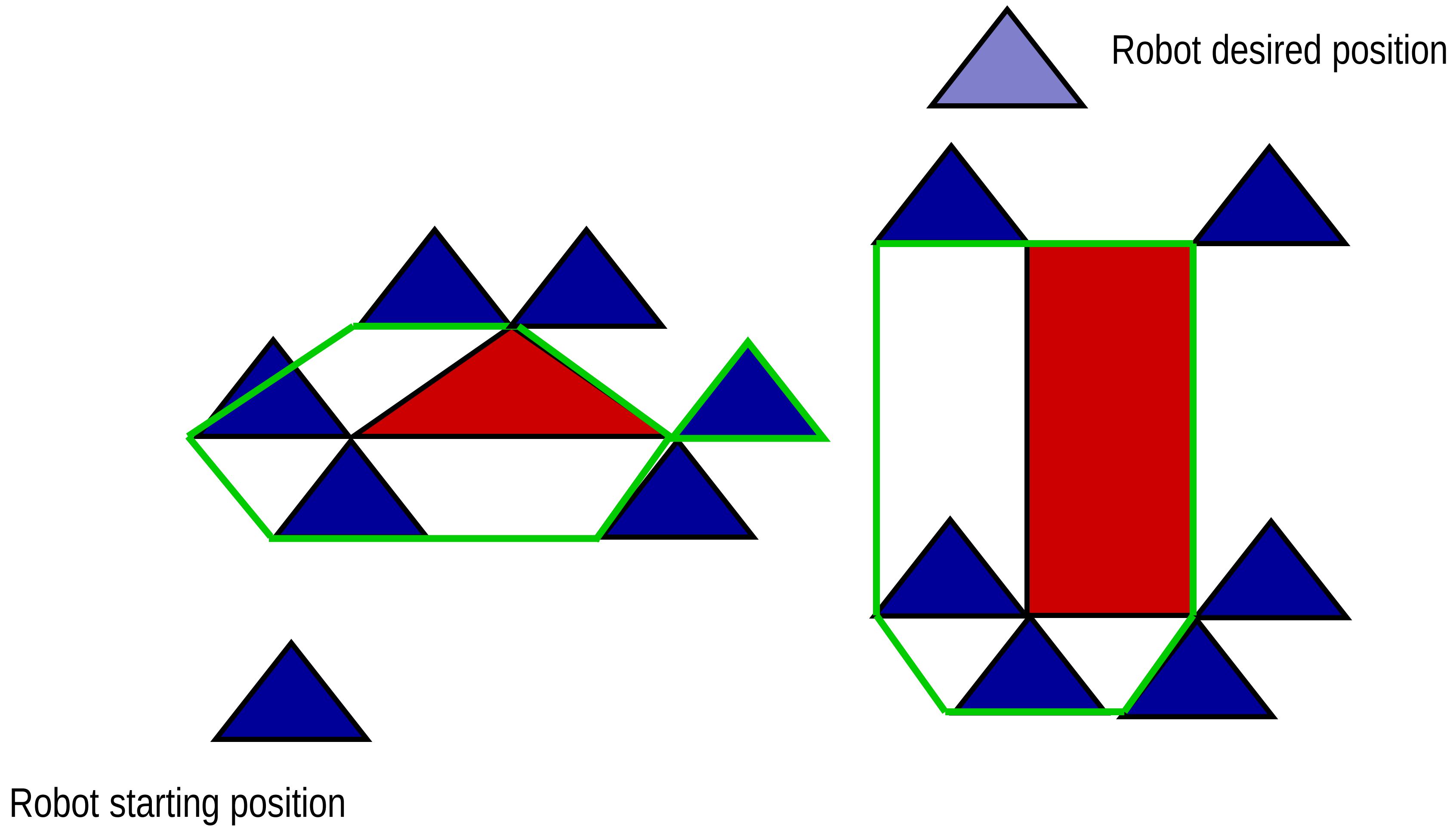
Method for Object Growing

- In this example: Triangular robot
- Configuration growing: based on robot’s bottom left corner
- Method: conceptually move robot around obstacles without collision, marking path of robot’s bottom left corner





Method for Object Growing

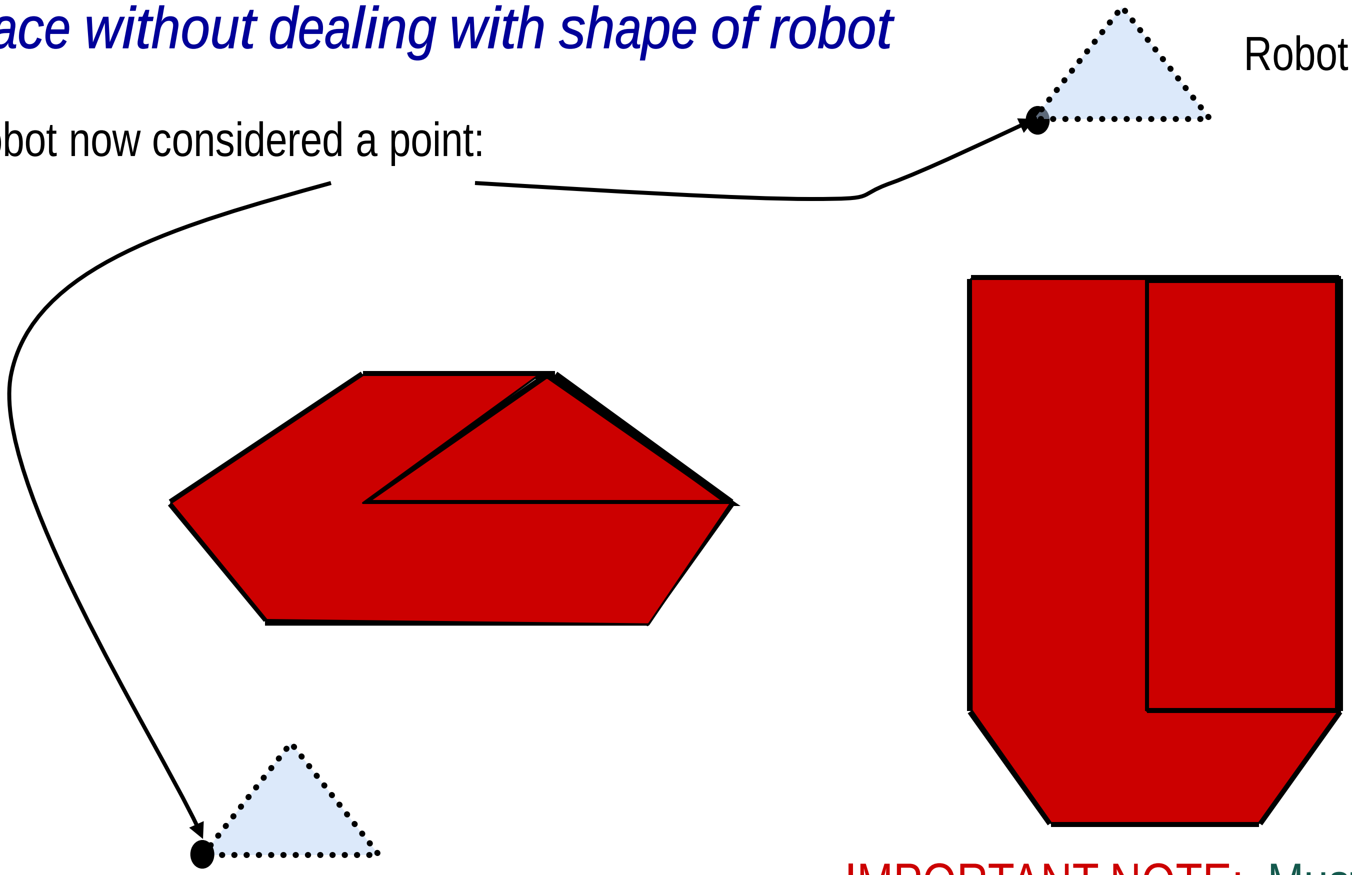


Result of Object Growing: New Configuration Space

Can now plan path of point through this space without dealing with shape of robot

Robot desired position

Robot now considered a point:



Robot starting position

IMPORTANT NOTE: Must make multiple configurations spaces corresponding to various degrees of rotations for moving objects. Then, generalize search to move from space to space

Examples of Cspace Representations

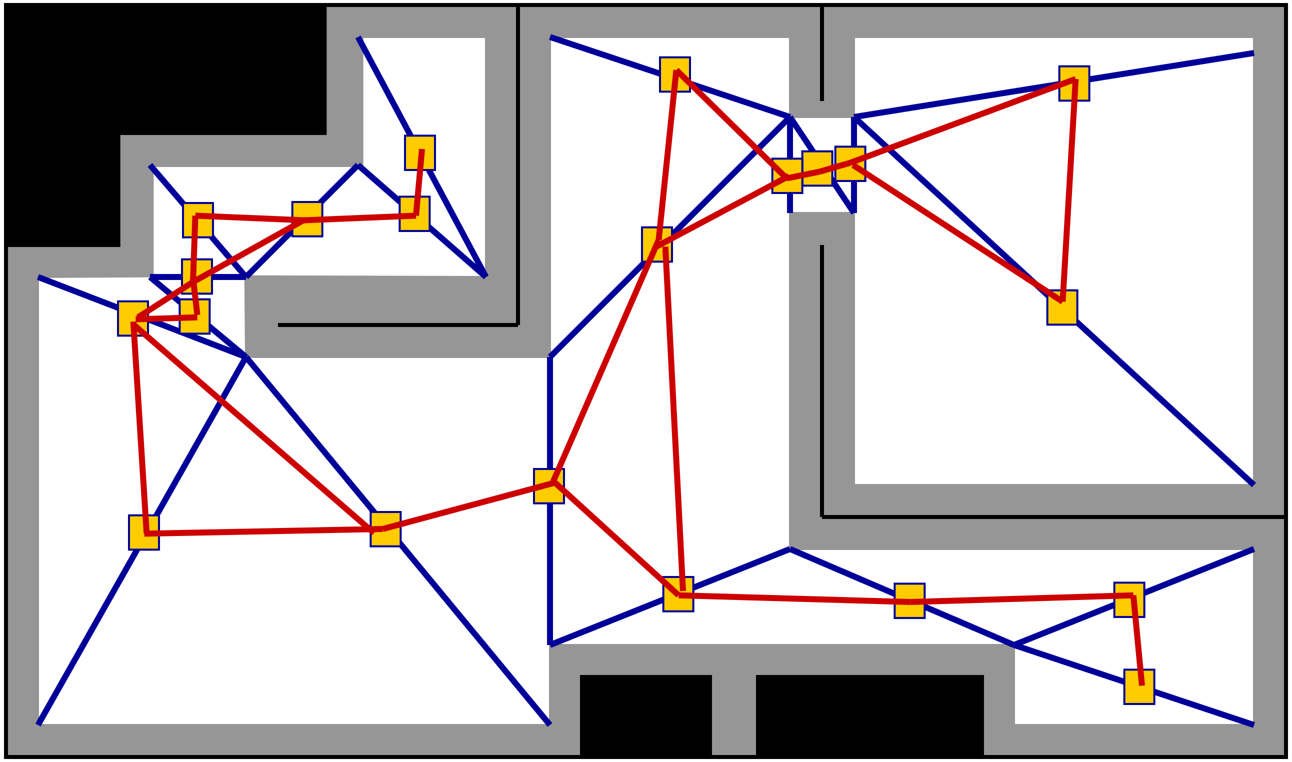
- Voronoi diagrams
- Regular grids
- Quadtrees/octtrees
- Vertex graphs
- Hybrid free space/vertex graphs (meadow map)

Meadow Maps (Hybrid Vertex-graph Free-space)

- Transform space into convex polygons
  - Polygons represent safe regions for robot to traverse
- Important property of convex polygons:
  - If robot starts on perimeter and goes in a straight line to any other point on the perimeter, it will not go outside the polygon
- Path planning:
  - Involves selecting the best series of polygons to transit through

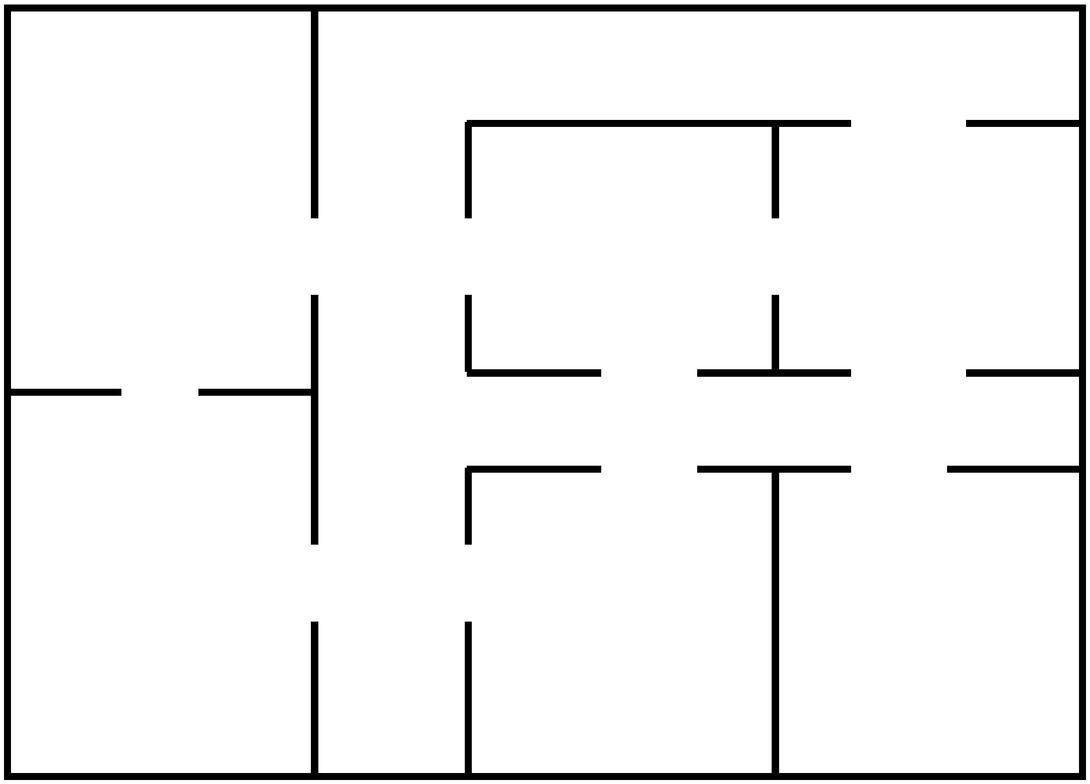
Example Meadow Map

1. Grow objects
2. Construct convex polygons
3. Mark midpoints; these become graph nodes for path planner
4. Path planner plans path based upon new graph



Class Exercise

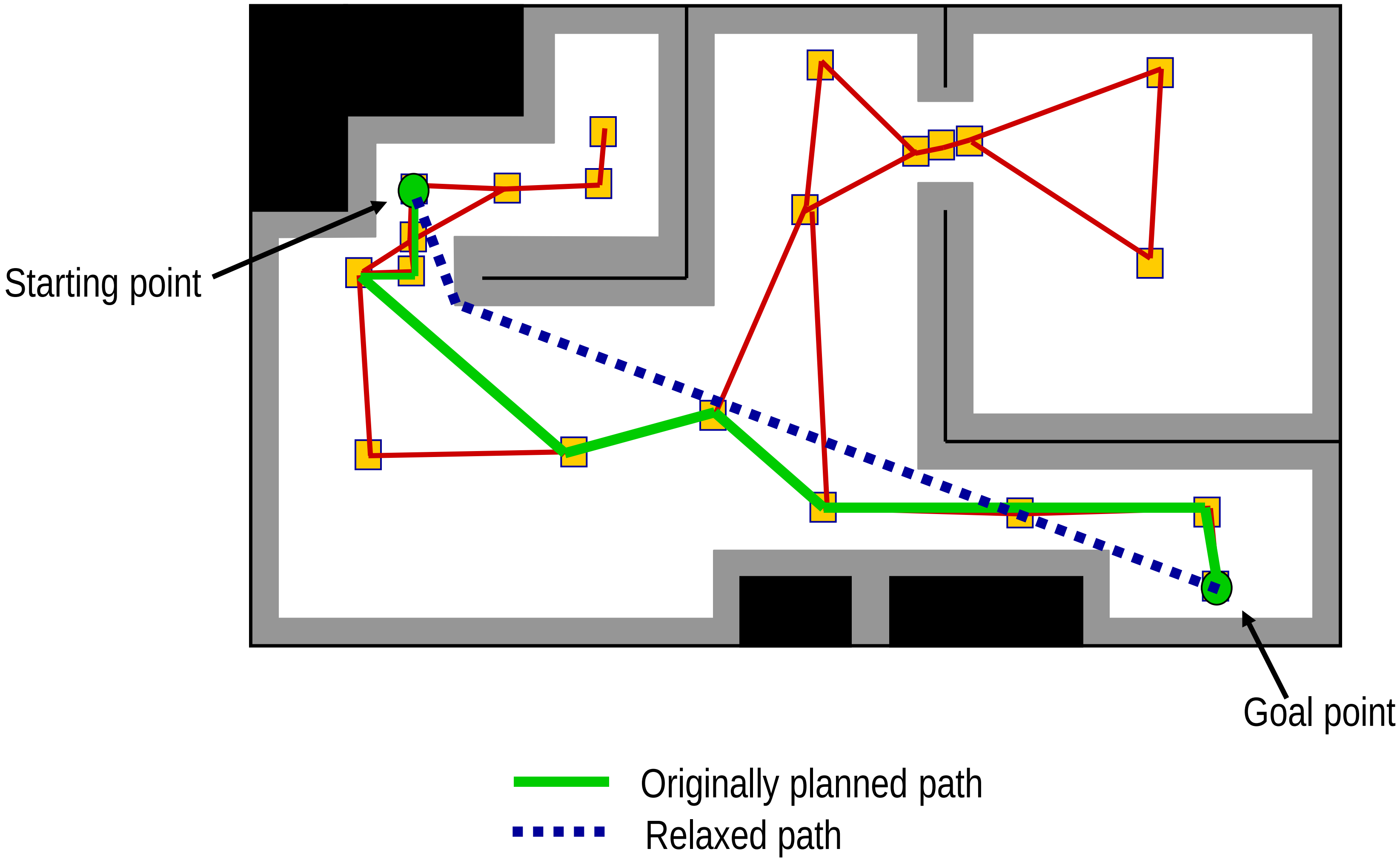
- Create a meadow map and relational graph using mid-point of line segments



Path Relaxation

- Disadvantage of Meadow Map:
  - Resulting path is jagged
- Solution: path relaxation
  - Technique for smoothing jagged paths resulting from any discretization of space
- Approach:
  - Imagine path is a string
  - Imagine pulling on both ends of the string to tighten it
  - This removes most of “kinks” in path

Example of Path Relaxation



Limited Usefulness of Meadow Maps

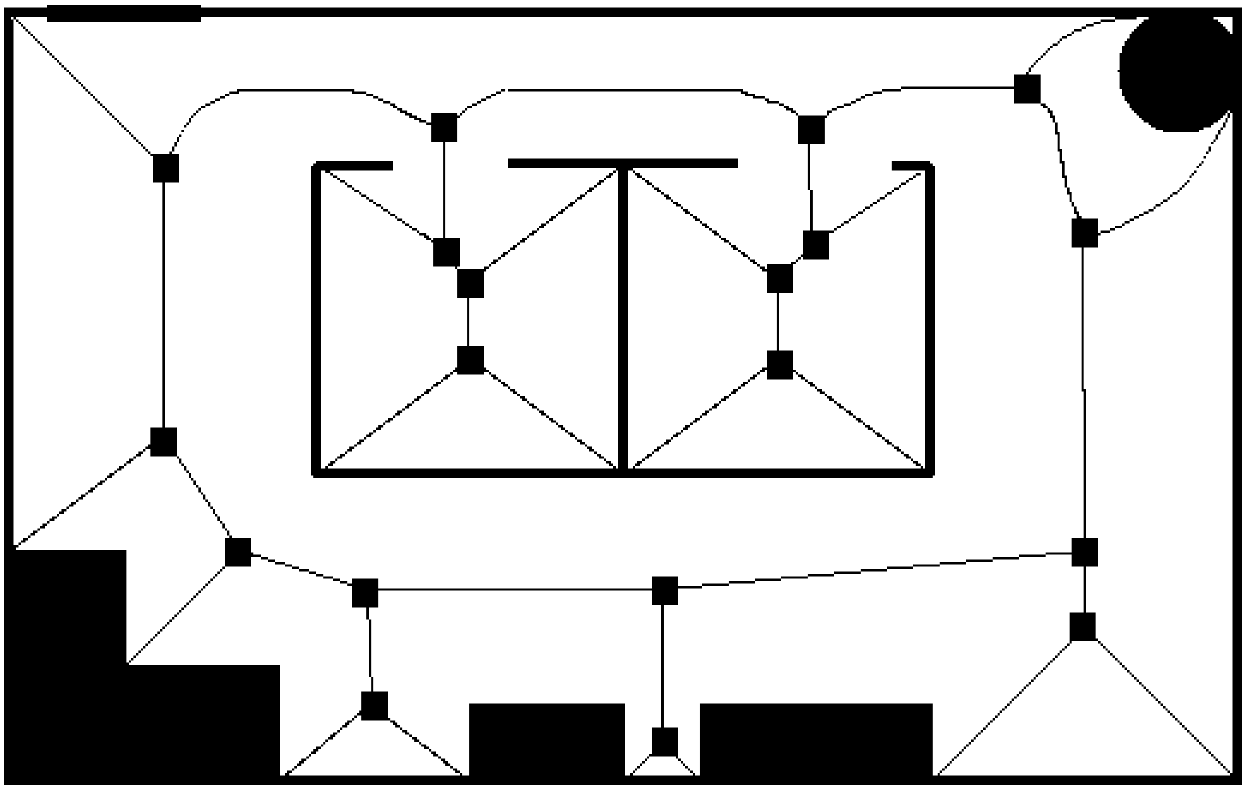
- Three problems with meadow maps:
  - Technique to generate polygons is computationally complex
  - Uses artifacts of the map to determine polygon boundaries, rather than things that can be sensed
  - Unclear how to update or repair diagrams as robot discovers differences between *a priori* map and the real world

Generalized Voronoi Diagrams (GVGs)

- GVGs:
  - Popular mechanism for representing Cspace and generating a graph
  - Can be constructed as robot enters new environment
- Basic GVG approach:
  - Generate a Voronoi edge, which is equidistant from all points
  - Point where Voronoi edge meets is called a Voronoi vertex
  - Note: vertices often have physical correspondence to aspects of environment that can be sensed
  - If robot follows Voronoi edge, it won't collide with any modeled obstacles → don't need to grow obstacle boundaries

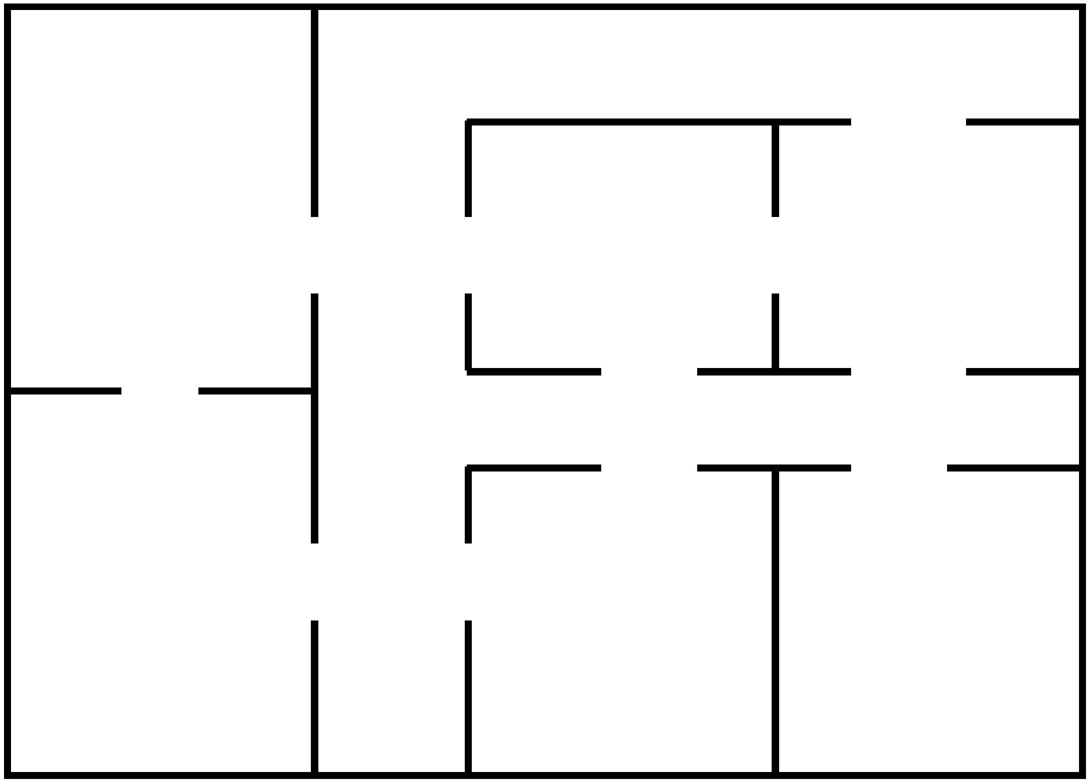
Example Generalized Voronoi Graph (GVG)

- Imagine a fire starting at the boundaries, creating a line where they intersect. Intersection of lines are nodes.
- Result is a relational graph.



Class Exercise

- Create a GVG of this space

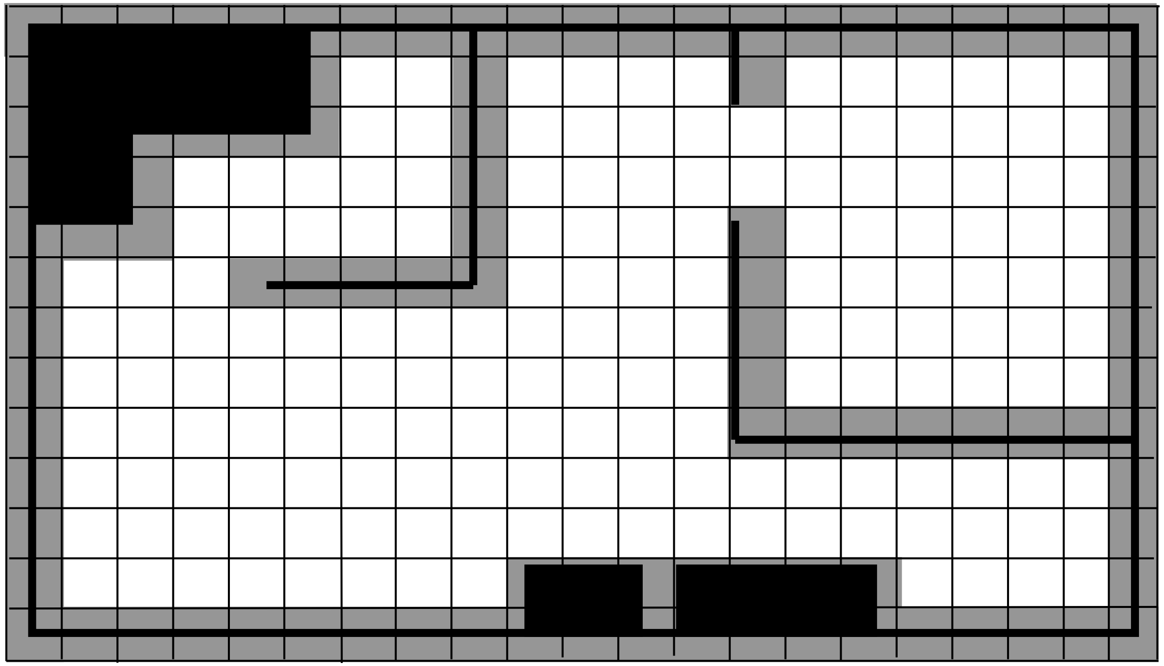


Problems with GVG

- Sensitive to sensor noise
- Path execution: requires robot to be able to sense boundaries

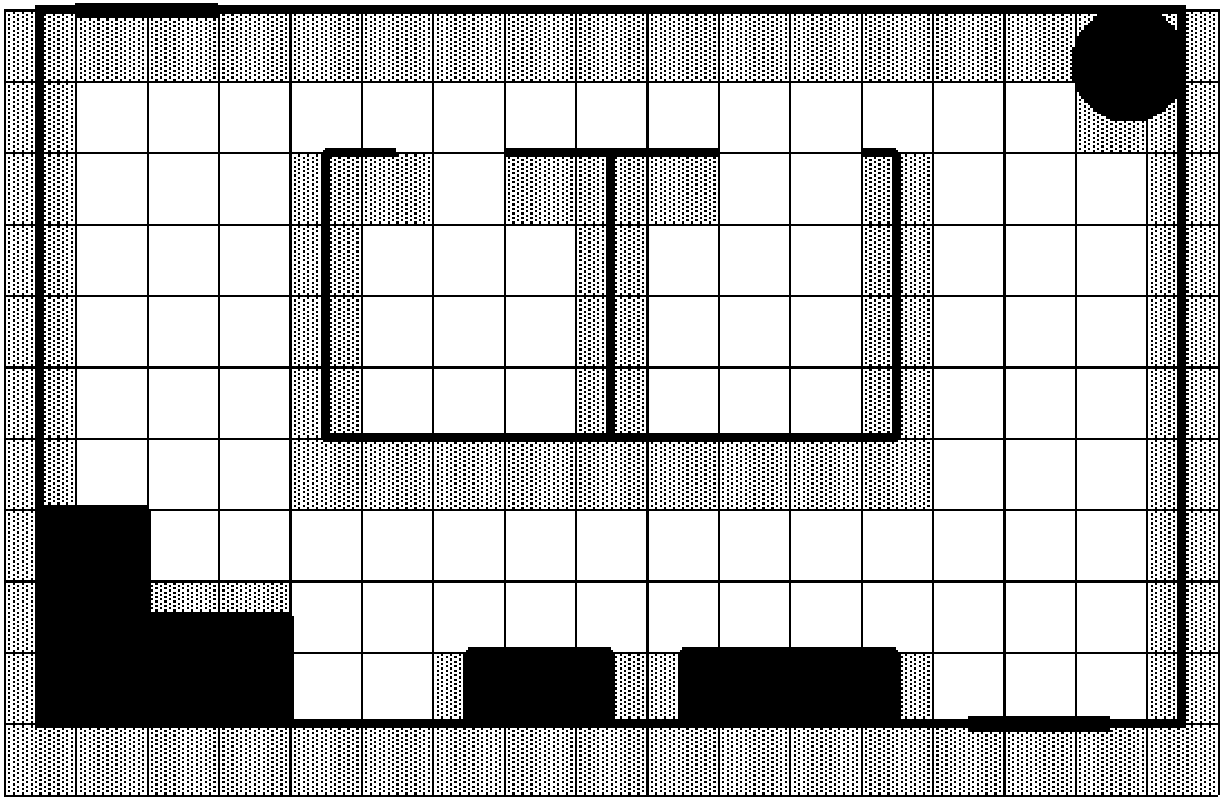
Regular Grids / Occupancy Grids

- Superimposes a 2D Cartesian grid on the world space (bigger than pixels, but same idea)
- If there is any object in the area contained by a grid element, that element is marked as occupied
- Center of each element in grid becomes a node, leading to highly connected graph
- Grid nodes are connected to neighbors (either 4-connected or 8-connected)



Class Exercise

- Use 4-connected neighbors to create a relational graph



Disadvantages of Regular Grids

- Digitization bias:
  - World doesn’t always line up on grids (If object falls into even small portion of grid element, the whole element is marked as occupied)
  - Leads to wasted space
    - Solution: use fine-grained grids (4-6 inches)
    - But, this leads to high storage cost and high # nodes for path planner to consider

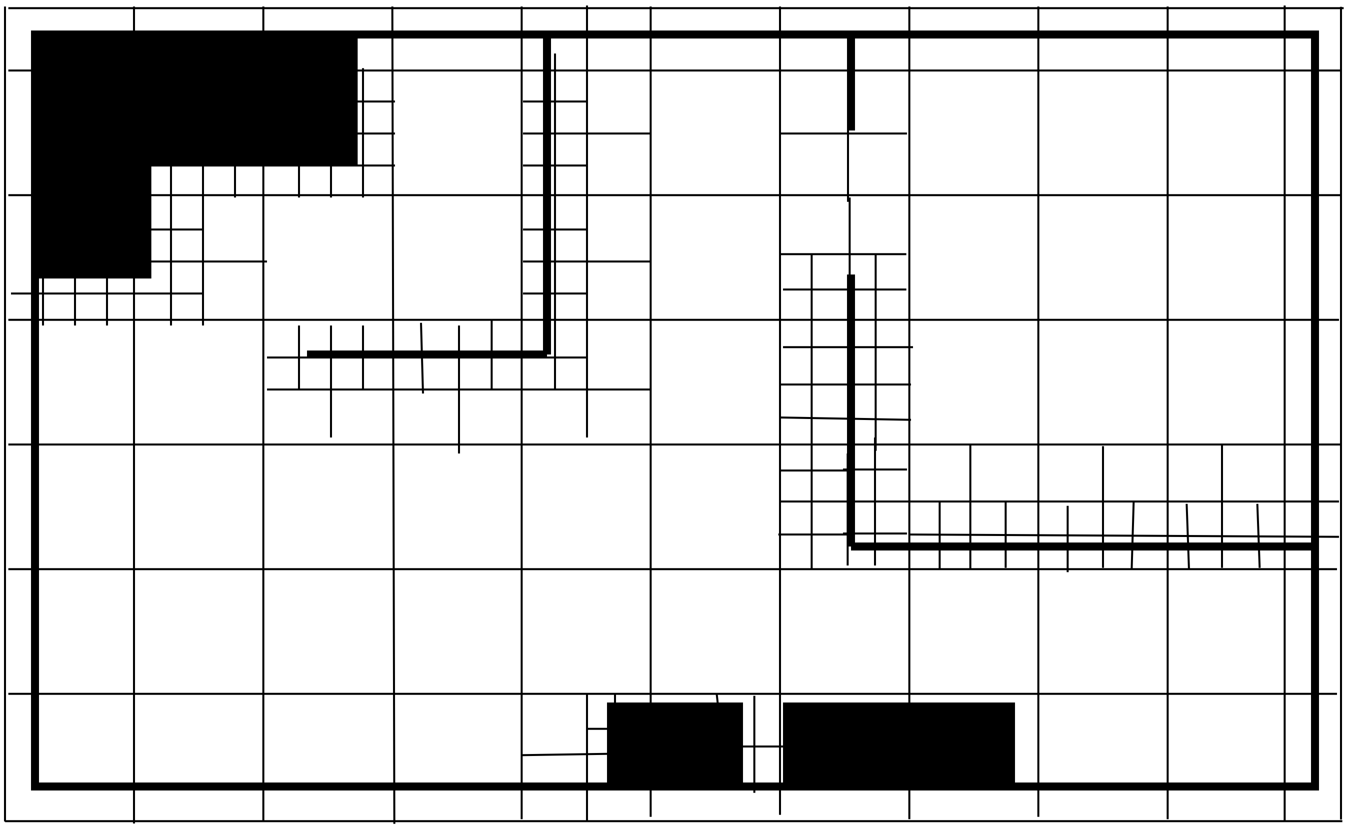
- Partial solution to wasted space: Quadtrees

Quadtrees

- Representation starts with large area (e.g., 8x8 inches)
- If object falls into part of grid, but not all of grid, space is subdivided into for smaller grids
- If object doesn't fit into sub-element, continue recursive subdivision
- 3D version of Quadtree – called an Octree.

Example Quadtree Representation

(Not all cells are subdivided as in an actual quadtree representation (too much work for a drawing by hand!, but this gives basic idea)



Summary of Representations

- Metric path planning requires
  - Representation of world space, usually try to simplify to cspace
  - Algorithms which can operate over representation to produce best/optimal path

- Representation
  - Usually try to end up with relational graph
  - Regular grids are currently most popular in practice, GVGs are interesting
  - Tricks of the trade
    - Grow obstacles to size of robot to be able to treat *holonomic* robots as point
    - Relaxation (string tightening)

- Metric methods often ignore issue of
  - how to execute a planned path
  - Impact of sensor noise or uncertainty, localization

Algorithms

- For Path planning
  - A* for relational graphs
  - Wavefront for operating directly on regular grids

Graph Based Planners

- Finding path between initial node and goal node can be done using graph search algorithms
- Graph search algorithms: found in networks, routing problems, etc.
- However, many graph search algorithms require visiting each node in graph to determine shortest path
  - Computationally tractable for sparsely connected graph (e.g., Voronoi diagram)
  - Computationally expensive for highly connected graph (e.g., regular grid)
- Therefore, interest is in "branch and bound" search
  - Prunes off paths that aren't optimal
- Classic approach: A* ("A Star") search algorithm
  - Frequently used for holonomic robots

Motivation for A*

- Single Source Shortest Path algorithms are exhaustive, visiting all edges
  - *Can't we throw away paths when we see that they aren't going to the goal, rather than follow all branches?*
- This means having a mechanism to "prune" branches as we go, rather than after full exploration
- Algorithms which prune earlier (but correctly) are preferred over algorithms which do it later.
- Issue -> the mechanism for pruning

A* Search Algorithm

- Similar to breadth-first: at each point in the time the planner can only "see" its node and 1 set of nodes "in front"
- Idea is to rate the choices, choose the best one first, throw away any choices whenever you can:

$f^*(n) = g^*(n) + h^*(n)$ // '*' means these are estimates

where:

- $f^*(n)$ is the "goodness" of the path from Start to n
- $g^*(n)$ is the "cost" of going from the Start to node n
- $h^*(n)$ is the cost of going from n to the Goal
  - h is for "heuristic function", because must have a way of guessing the cost of n to Goal since can't see the path between n and the Goal

A* Heuristic Function

$f^*(n) = g^*(n) + h^*(n)$

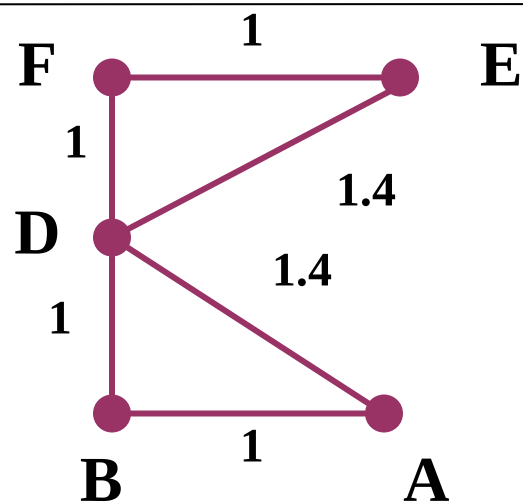
- $g^*(n)$ is easy: just sum up the path costs to n

- $h^*(n)$ is tricky
  - But path planning requires an *a priori* map
  - Metric path planning requires a METRIC *a priori* map
  - Therefore, know the distance between Initial and Goal nodes, just not the optimal way to get there
  - $h^*(n)$= distance between n and Goal

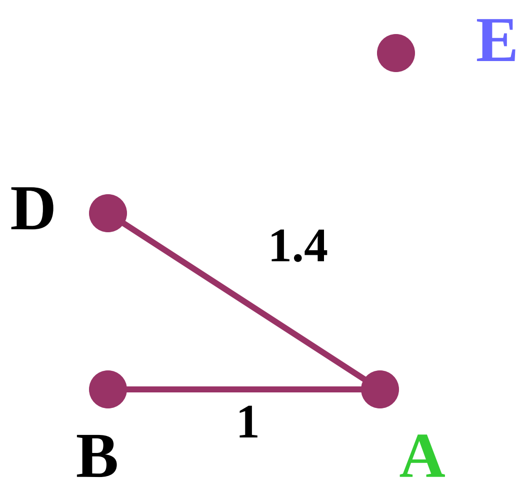
Estimating *h*(*n*)

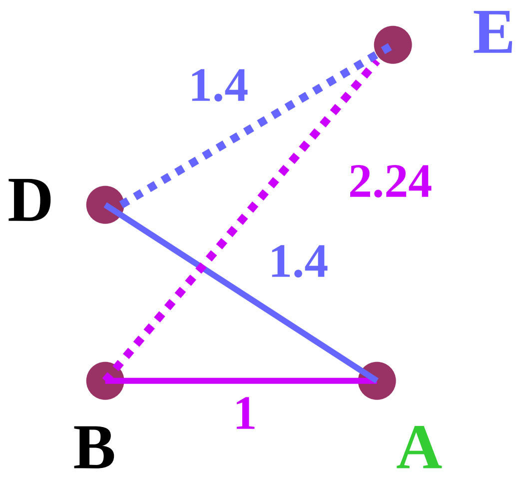
- Must ensure that *h**(*n*) is never greater than *h*(*n*)
- Admissibility condition:
  - Must always underestimate remaining cost to reach goal
- Easy way to estimate:
  - Use Euclidian (straight line) distance
  - Straight line will always be shortest path
  - Actual path may be longer, but admissibility condition still holds

Example: A to E



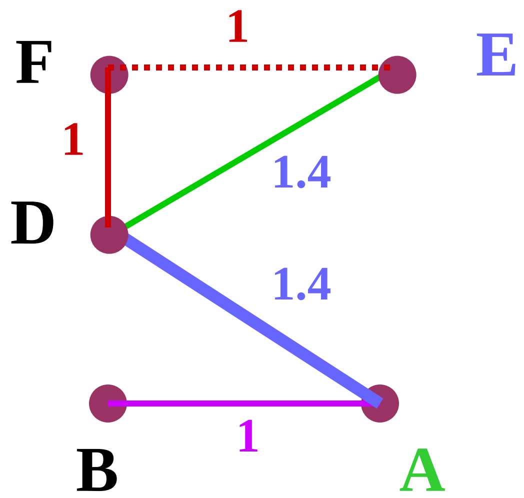
- But since you're starting at A and can only look 1 node ahead, this is what you see:





- Two choices for *n*: B, D
- Do both
  - f*(B) =1 + 2.24 = 3.24
  - f*(D) =1.4 + 1.4 = 2.8
- Can’t prune, so much keep going (recurse)
  - Pick the most plausible path first => A-D-?-E

- A-D-?-E
  - “stand on D”
  - Can see 2 new nodes: F, E
  - f*(F) =(1.4+1)   + 1 = 3.4
  - f*(E) =(1.4+1.4)+ 0 = 2.8



- Three paths
  - A-B-?-E $\geq$ 3.24
  - A-D-E = 2.8
  - A-D-F-?-D $\geq$ 3.4

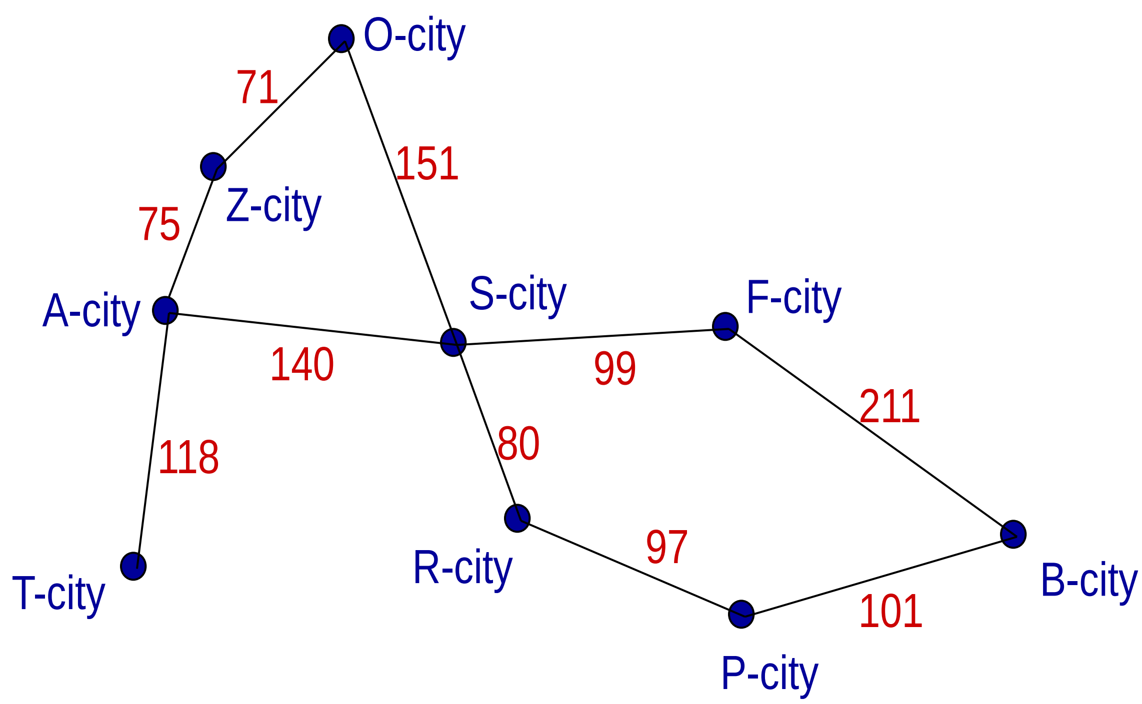
- A-D-E is the winner!
  - Don’t have to look farther because expanded the shortest first, others couldn’t possibly do better without having negative distances, violations of laws of geometry…

Class Exercise

Compute optimal path from A-city to B-city

Straight-line distance to B-city from:

- A-city: 366
- B-city: 0
- F-city: 176
- O-city: 380
- P-city: 98
- R-city: 193
- S-city: 253
- T-city: 329
- Z-city: 374



Pros and Cons of A* Search/Path Planner

- Advantage:
  - Can be used with any Cspace representation that can be transformed into a graph
- Limitation:
  - Hard to use for path planning when there are factors to consider other than distance (e.g., rocky terrain, sand, etc.)