CS380
Artificial Intelligence for Games

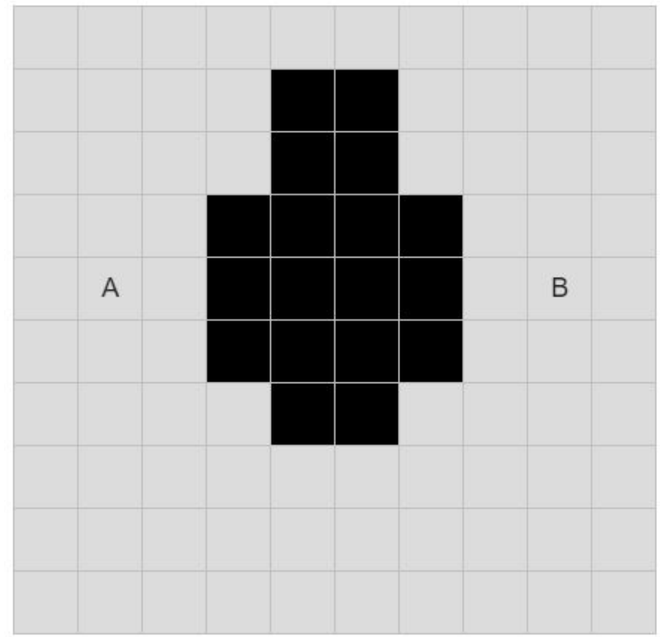
Navigation Graphs And Meshes.
Flood-Fill Search

Outline

- Navigation graphs
  - Tile-based graph
  - Point of Visibility graph
- Navigation meshes
- Uninformed search
  - Flood-fill algorithm

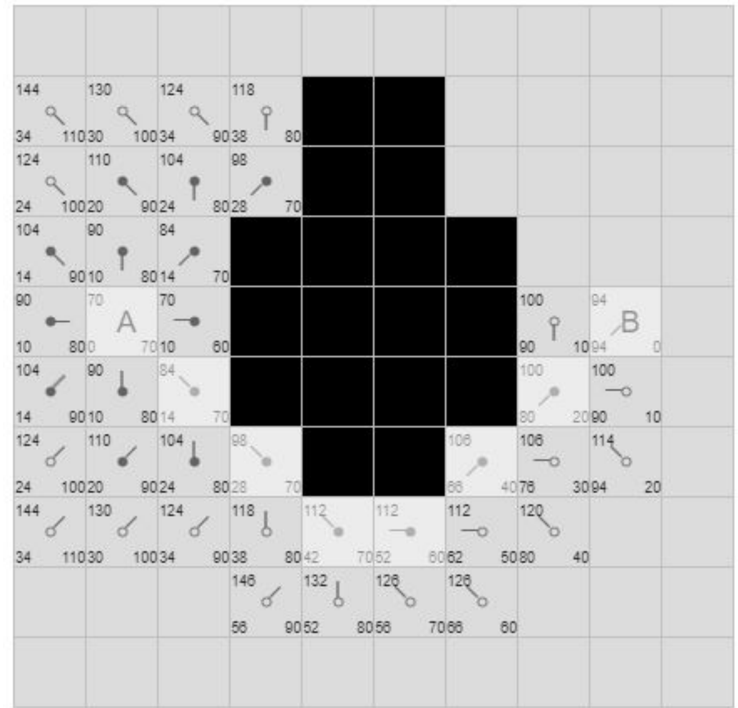
Navigation graphs. Finding a Path

- Often seems obvious and natural in real life
  - e.g., Get from point A to B go around lake
- For computer controlled player or non-player character (NPC), may be difficult
  - e.g., Going from A to B goes through enemy base!
- Want to pick “best” path
- Need to do it in real-time



Navigation graphs. Path

- **Path** – a list of cells, points or nodes that NPC must traverse to get to from start to goal
  - Some paths are better than others
  - So need a **measure of quality**
- A* is commonly used heuristic search
  - *Complete* algorithm in that if there is a path, will find
  - Using “distance” as heuristic measure, then guaranteed optimal

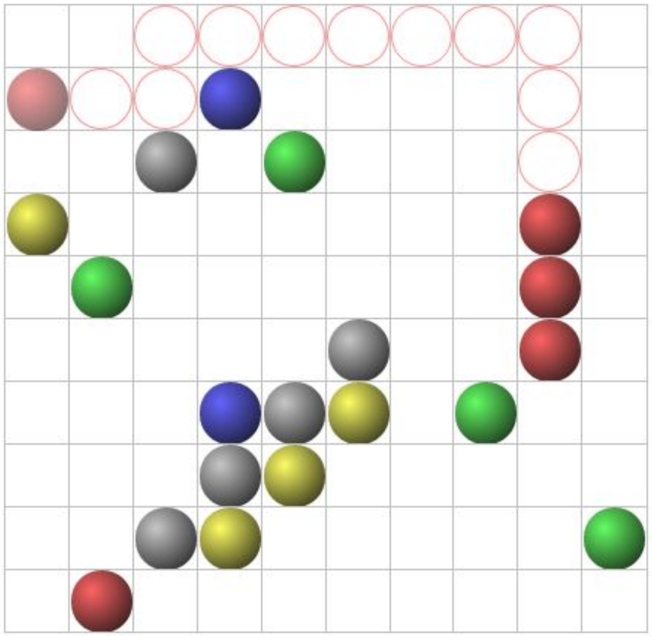


Navigation graphs. Practical Path Planning

- Sometimes, **basic A*** is not enough
- Also often need:
  - Navigation graphs/meshes
  - Path smoothing (optional)
  - Compute-time optimizations (optional)
  - Hierarchical pathfinding (optional)
  - Special case methods (optional)

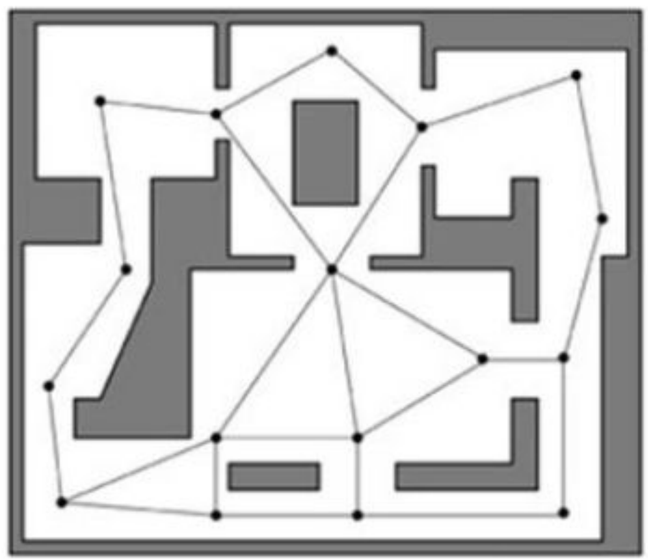
Navigation Graphs. Tile-Based

- Common, especially if environment already designed in squares or hexagons
- Node center of cell; edges to adjacent cells
- Each cell already labeled with material (mud, river, etc.)
- Downside:
  - Can burden CPU and memory
    - e.g., Modest 100x100 cell map has 10,000 nodes and 78,000 edges!
  - Especially if multiple AI's calling at same time



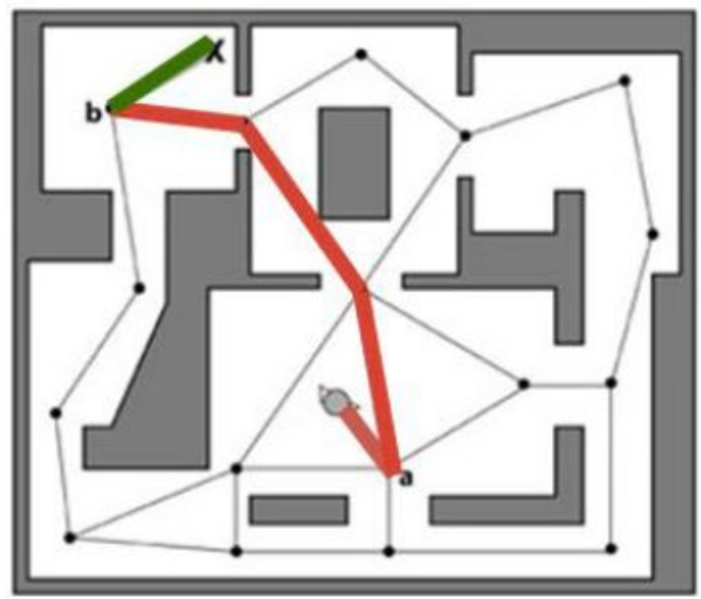
Navigation Graphs. Point of Visibility (POV)

- Place graph nodes (usually by hand) at **important points** in environment
- Such that <u>each node has line of sight to at least one other node</u>



Navigation Graphs. POV Navigation

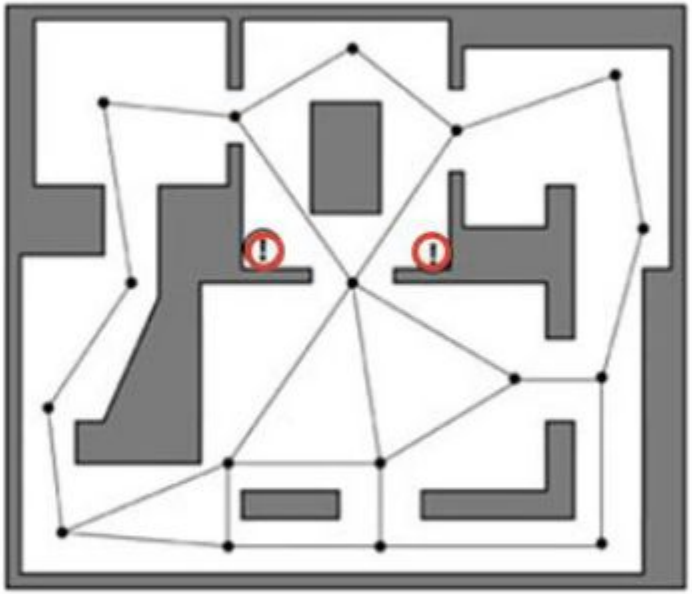
- Find closest **visible** node (a) to current location
- Find closest **visible** node (b) to target location
- Search for least cost path from (a) to (b), e.g. A*
- Move to (a)
- Follow path to (b)
- Move to target location



Note, move to (a) is not
the best option

Navigation Graphs. Blind Spots in POV

- No POV point is visible from red spots!
- Easy to fix manually in small graphs
  - Move to **visible** edge at **shortest** distance!
- A problem in larger graphs



Navigation Graphs. POV

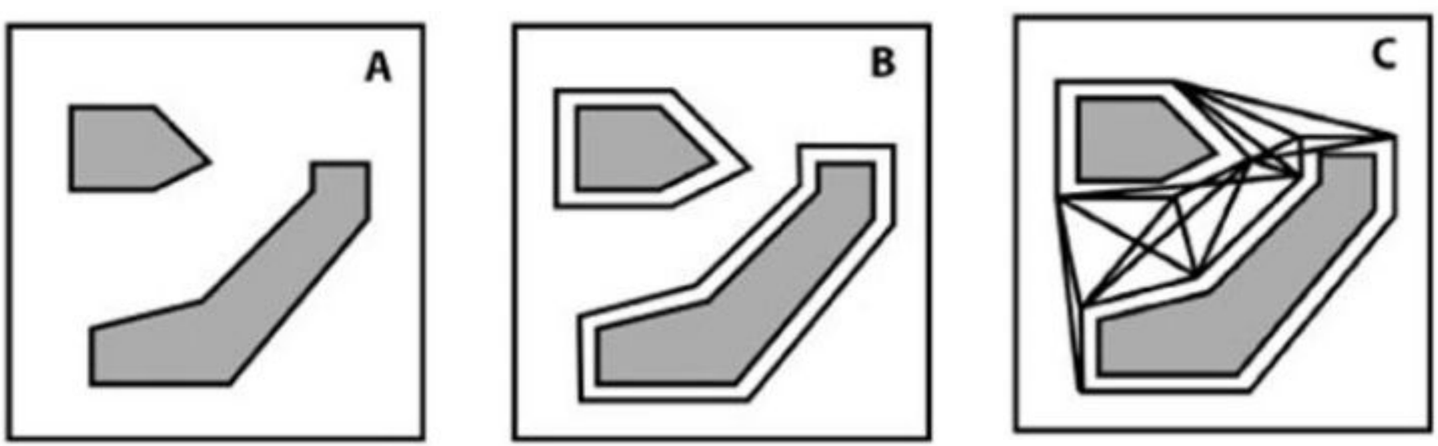
- **Advantage**
  - Obvious how to build and expand manually
- **Disadvantages**
  - Can take a lot of developer time, especially if design is rapidly evolving
  - Problematic for random or user generated maps
  - Can have “blind spots”
  - Can have “jerky” (backtracking) paths
- **Solutions**
  - Automatically generate POV graphs
  - Make finer grained graphs
  - Path smoothing

Navigation Graphs. Automatic POV by Expanded Geometry

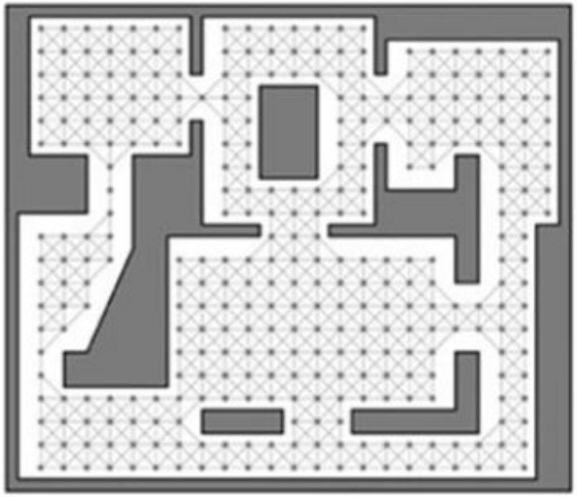
A. Expand geometry
   - By amount proportional to bounding radius of NPC
   - Note: works best if bounding radius similar for all NPCs

B. Connect all vertices

C. Prune non-line of sight points to avoid objects hitting edges when pathing



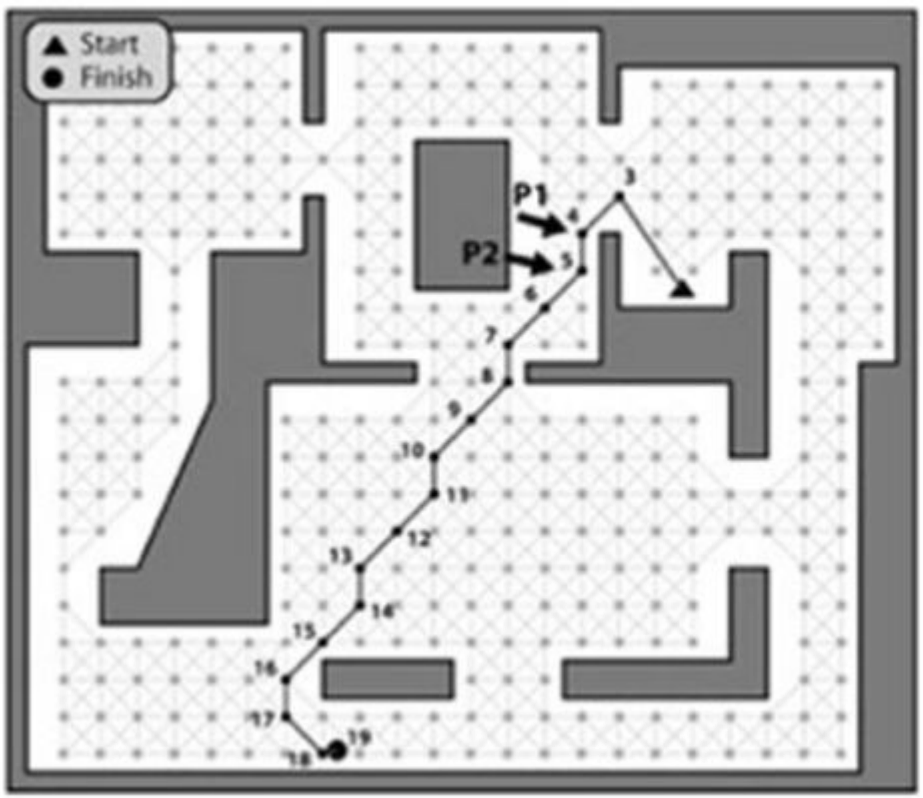
Navigation Graphs. Finely Grained Graphs



- Upside?
  - Improves blind spots and path smoothness
  - Can often generate automatically using Flood-fill algorithm
- Downside?
  - Back to similar performance issues as tiled graphs

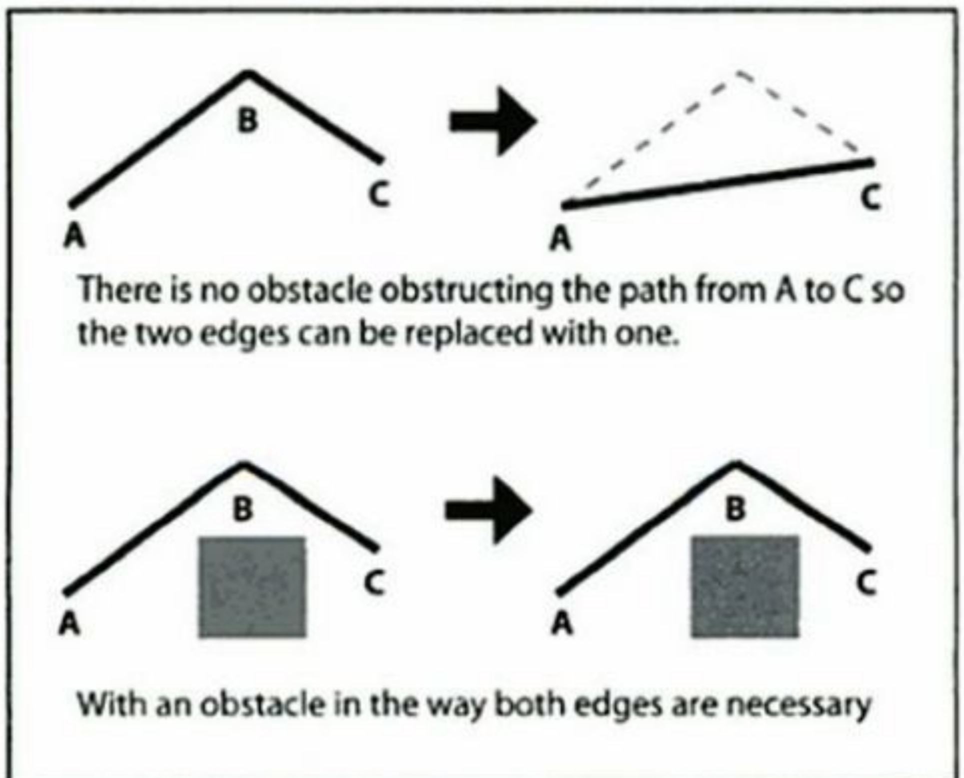
Navigation Graphs. Kinky Path Smoothing

- Problem: Path does not look “natural”
- Solution: “smoothing”



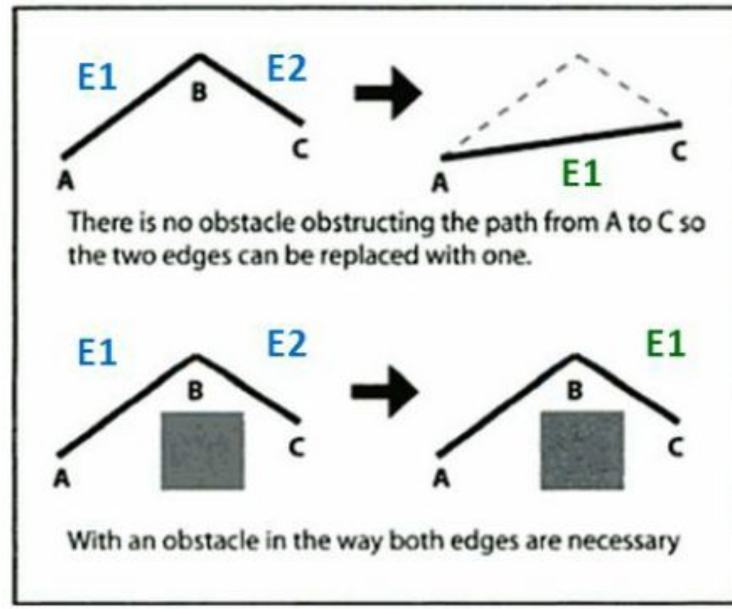
Navigation Graphs. Smoothing

- Check for "passability" between adjacent edges
- Also known as "ray-cast" since if can cast a ray between A and C then waypoint B is not needed

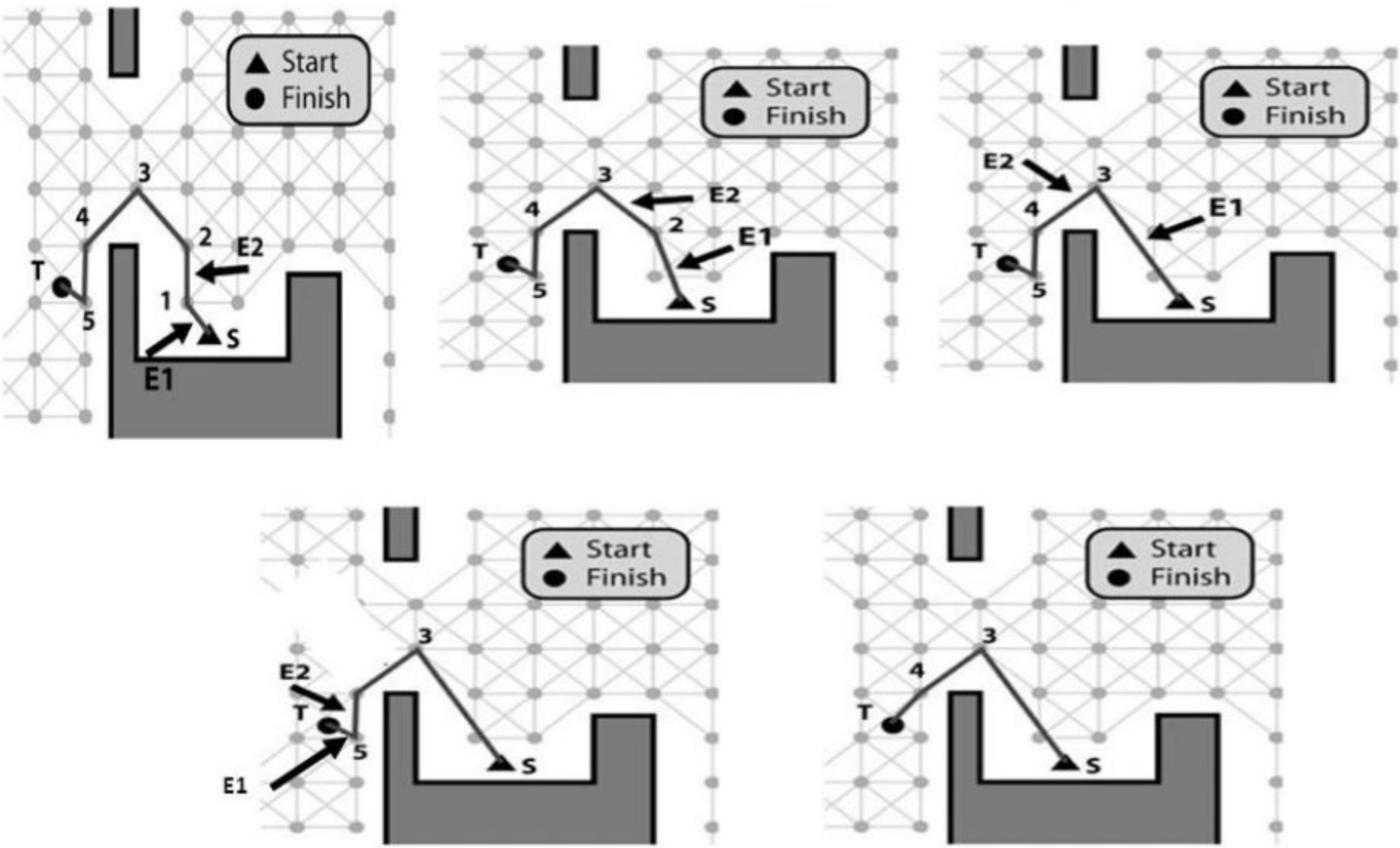


Navigation Graphs. Simple Smoothing Algorithm

1. Grab source edge E1
2. Grab destination E2
3. If NPC can move between,
   a. Assign destination E1 to destination E2
   b. Remove E2
   c. Advance E2
4. If NPC cannot move,
   a. Assign E2 to E1
   b. Advance E2
5. Repeat until destination E1 or destination E2 is endpoint



Navigation Graphs. Smoothing Example

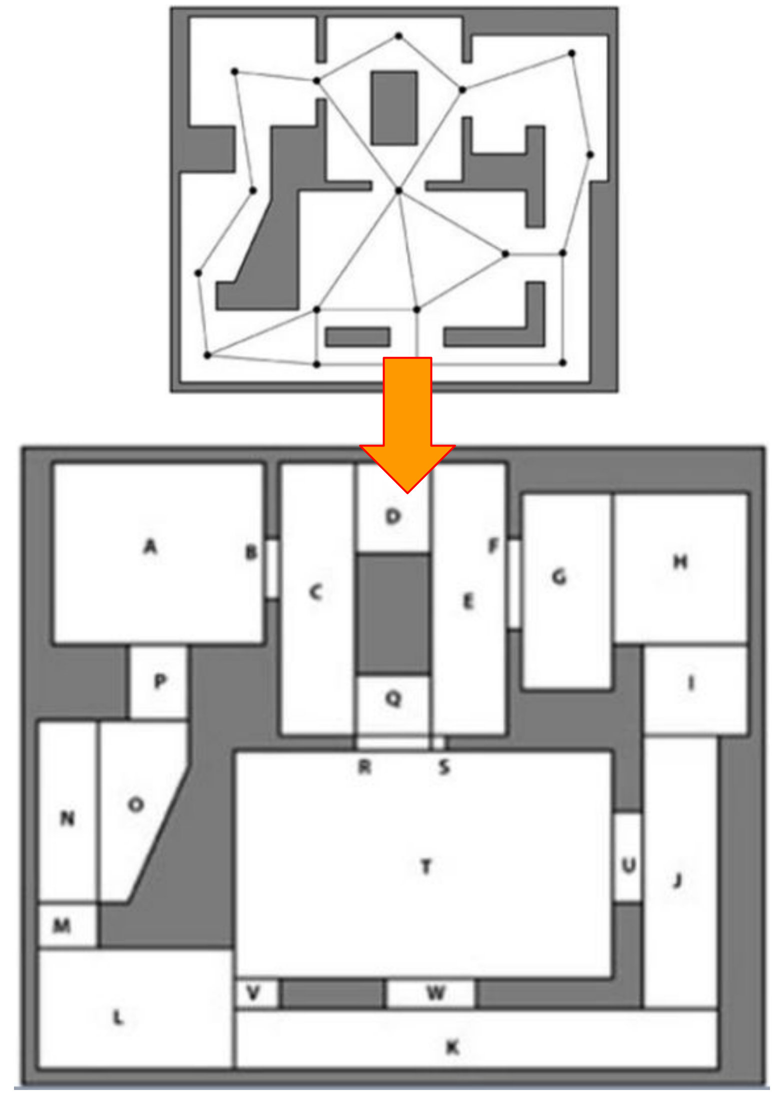


Outline

- Navigation graphs
  - Tile-based graph
  - Point of Visibility graph
- Navigation meshes
- Uninformed search
  - Flood-fill algorithm

Navigation Mesh

- Partition open space into a network of convex polygons
    - Convex shape guaranteed path from any point to any point inside
- Instead of network of points, have network of polygons
- Can be automatically generated from arbitrary polygons
- Becoming very popular (e.g., UE4)



Navigation Mesh

- Has more information (i.e., can walk anywhere in polygon)
- POV needs lots of points. Navmesh needs fewer polygons to cover same area
- No need smoothing, but smoothing works too
- Navmesh is also a graph where nodes are polygons

Navigation Mesh. Generating

- Can be generated by hand
  - e.g., lay out polygons (e.g., squares) to cover terrain for map
  - Takes a few hours for typical FPS map
- Can be generated automatically
  - Various algorithm choices

Outline

- Navigation graphs
  - Tile-based graph
  - Point of Visibility graph
- Navigation meshes
- Uninformed search
  - Flood-fill algorithm

Search. Definition

- **Search -** the process of looking for a sequence of actions/nodes that reaches the **goal/target**
- **Input:**
  - Navigation graph or mesh, state graph,
  - Starting point S and target T
- **Result:**
  - If found, the sequence from S to T (or from T to S)
  - Else “not found” flag
- Sometimes, result can be a flag of existence of such sequence
  - Use Flood-fill algorithm

Uninformed search. Definition

- AKA Blind search, bcs **blindly try all possible ways** to reach the goal (see Brute-Force Search next)
- Uninformed search is a search that has no information about its domain
  - The only things that such search can do are
    - iterate through all states (takes time)
    - distinguish a non-goal state from a goal state
- Form the basis for some of the intelligent searches
- The order in which a blind search expand the nodes:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

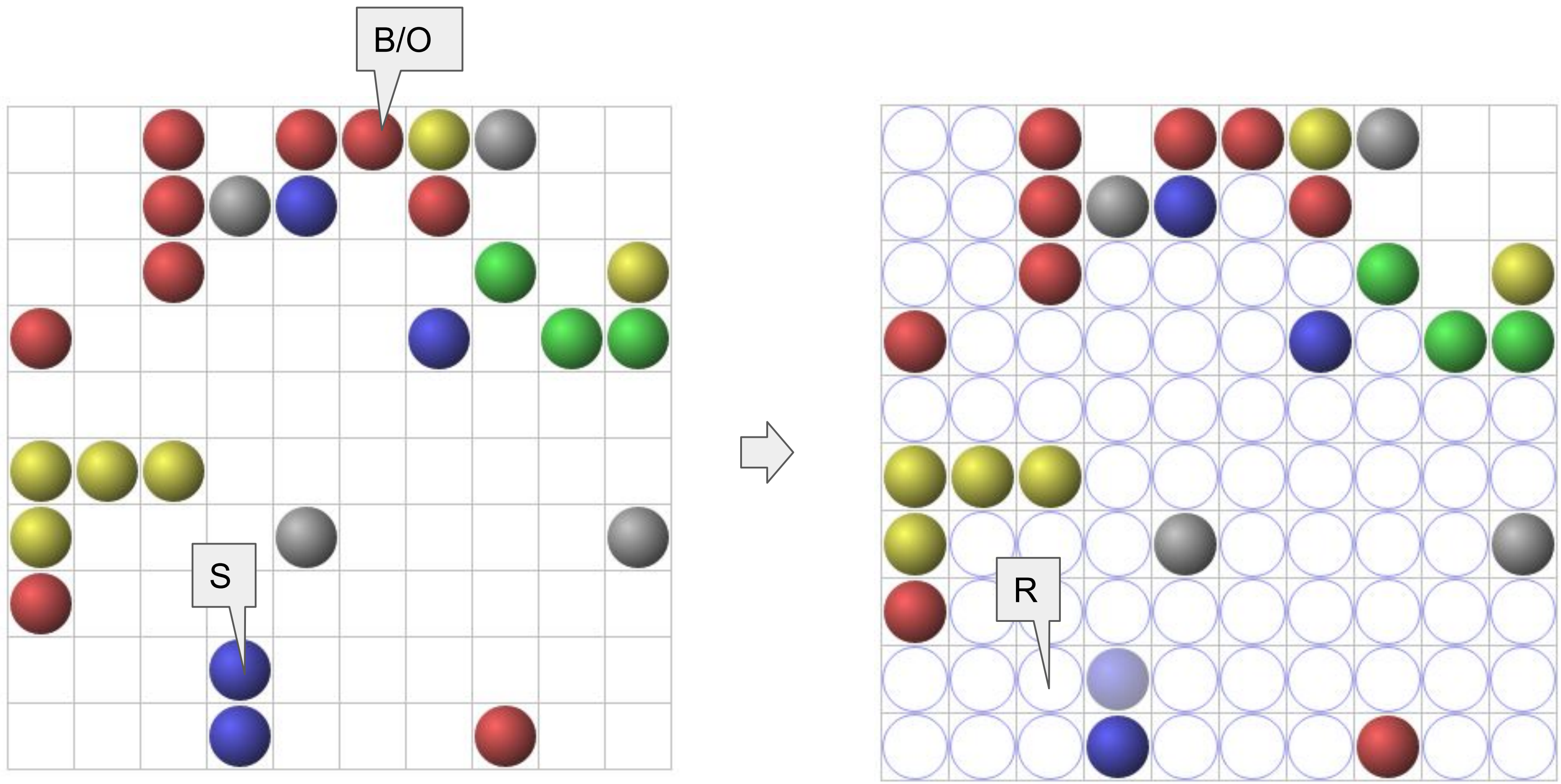
Brute-force search

- AKA **exhaustive** search or **generate and test** search
- Is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. (Wikipedia)

Uninformed search. Flood-Fill Algorithm

- Given a tile-based graph
- Input:
  - start node S,
  - replacement color R,
  - border/obstacle color B/O
  - Optional: target color T
    - If given, the search stopped once R replaces T
- The algorithm recursively looks for all nodes in the graph that are connected to S, that are not B/O or T and changes them to R

Uninformed search. Flood-Fill Example



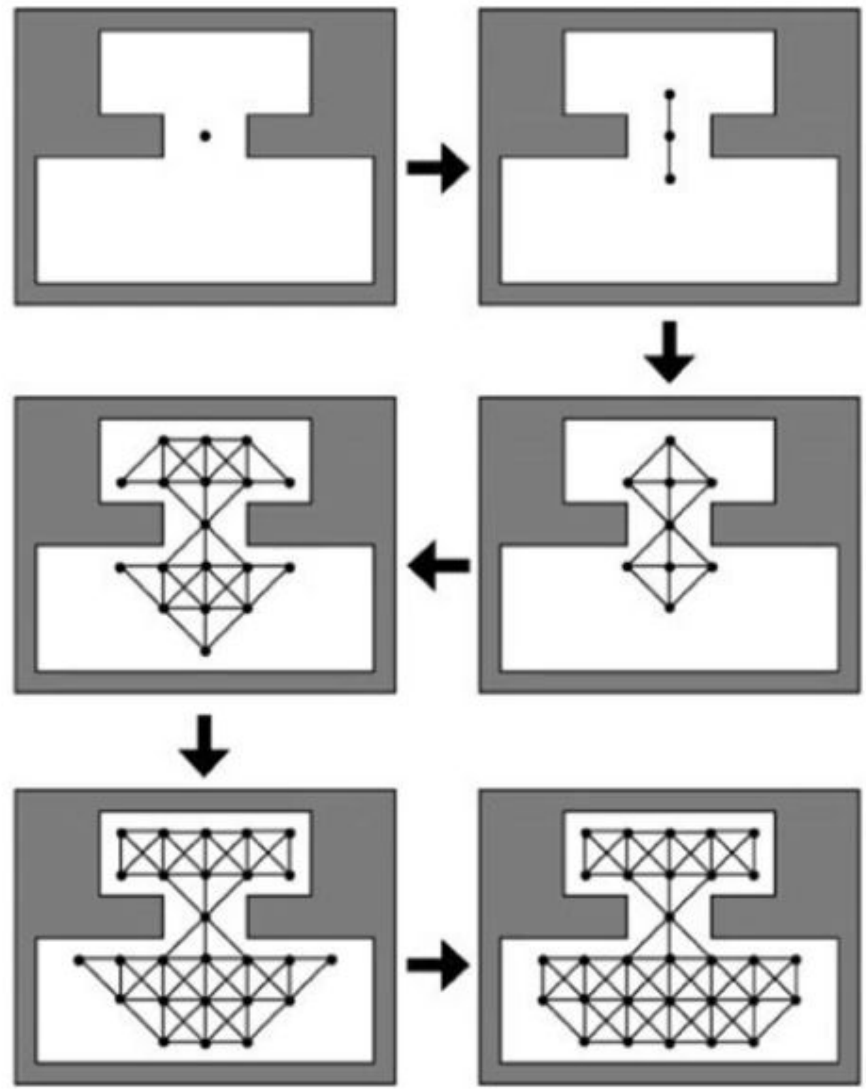
Uninformed search. Flood-Fill Code

```
Flood_Fill_Recursive(S, R)
{
   var adjacents = getAdjacents(S);
   for (var adjacent of adjacents)
   {
     adjacent.setColor(R);
     Flood_Fill_Recursive(adjacent, R);
   }
}
```

Construct Navigation Graph using Flood-Fill Algorithm

- Place "seed" in graph
- Expand outward in all "walkable" directions
- Making sure nodes and edges passable by bounding radius
- Continue until all area covered

- Same algorithm used by "paint" programs to flood fill color



Next Week

- More uninformed searches
- Dijkstra's algorithm

Readings

- Artificial Intelligence: A Modern Approach
  - Chapter 3: Solving Problems by Searching
    - Section 3.41 to Section 3.45