CS 134

Pathfinding

Game Engine Architecture
Chapter 14

Navigation Graph

- To do navigation, you need a graph representing the navigable positions
- Put this intelligence into the world!
  - For a small game, hand-generate this

Navigation Graph

```
// Describes all connections between nodes
ArrayList<GraphNode> graph;

// A specific node in the graph
class GraphNode {
    public NodeLink[] links;
    // Other data the AI cares about
}

// A connection from one node to another
class NodeLink
{
    // Index into graph
    public int destNode;
    // How far apart the nodes are
    public float cost;
    // How you move between nodes, e.g.
    // “WalkLeft”, “WalkRight”, “JumpLeft”, etc
    NodeLinkType type;
}
```

Navigation Graph

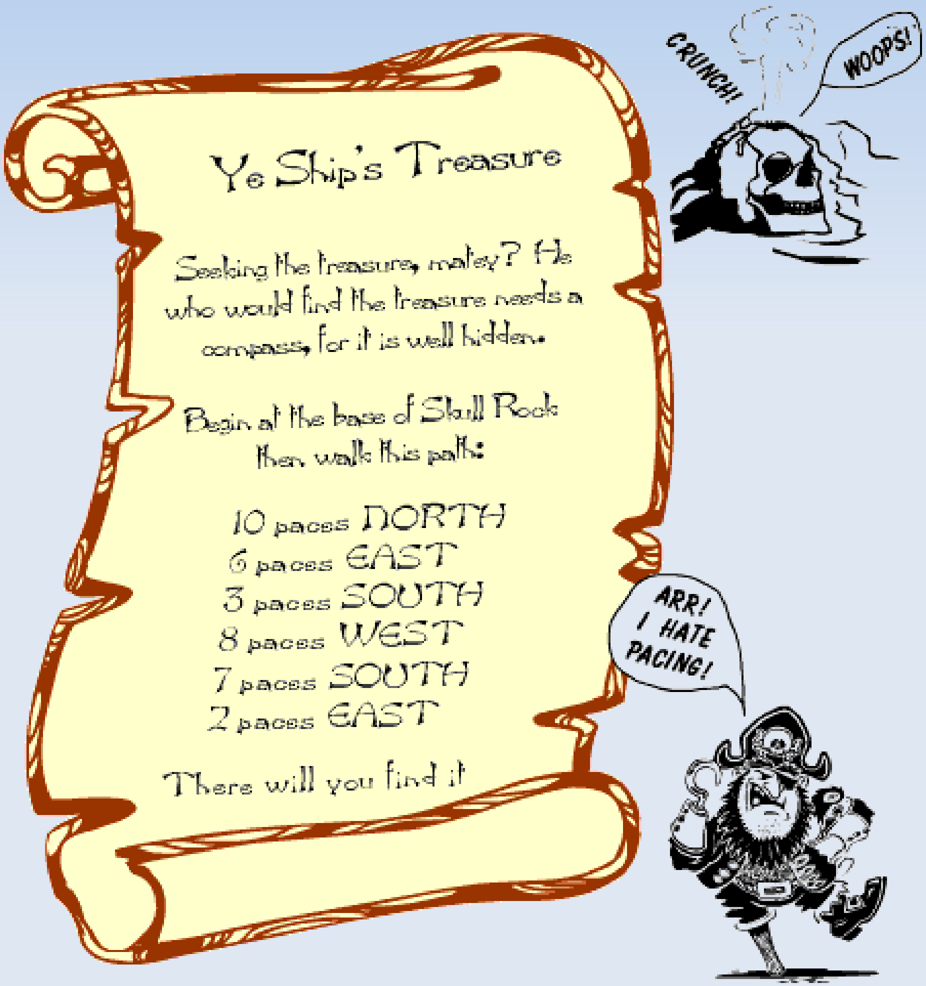
- Flat top down levels don't need a separate graph
- Tile grid already has navigation and collision
- Assume links go in eight directions, so long as not blocked

Navigation Graph

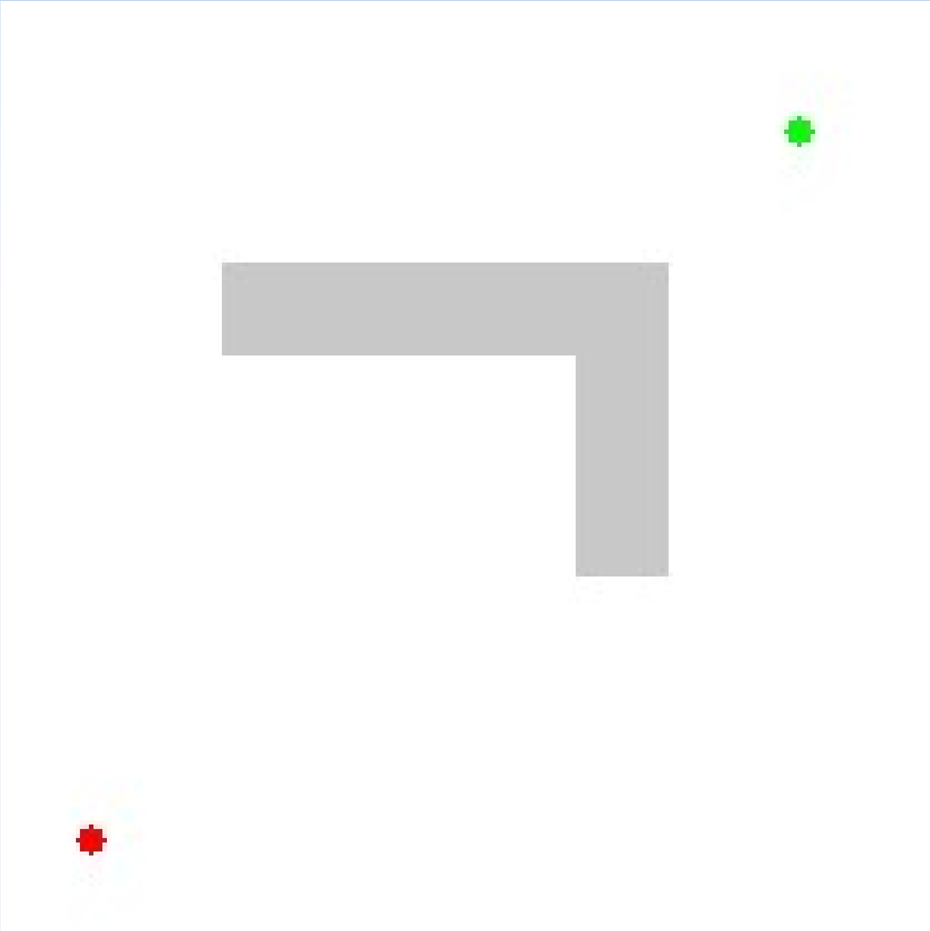
- Any level with jumping will need to know how to move
- GraphLinkType
  - WalkLeft, WalkRight
  - JumpLeft, JumpRight, JumpUp
  - HighJumpLeft, HighJumpRight, HighJumpUp
  - etc.

Navigation Graph

- The key concept:
- An AI must be able to figure out exactly what moves to do to follow a collection of links.
- Every type of motion possible should be in the links.



Navigation – Dijkstra's Algorithm

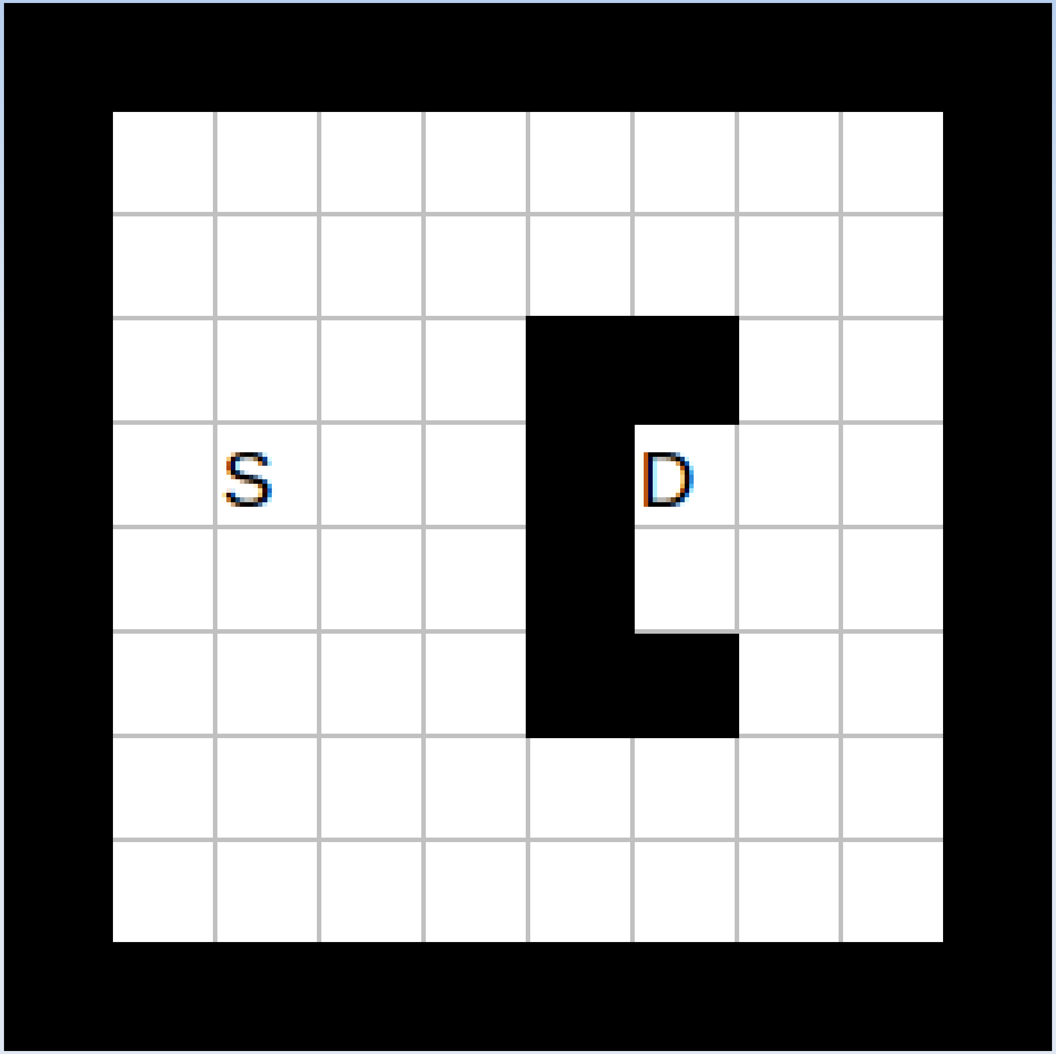


- Visit all reachable nodes from closest to furthest
- For each visited node, remember where you came from

Navigation – Dijkstra's Algorithm

- Create a priority queue of all nodes
- Mark S as distance 0, all other nodes as infinity
- While the cheapest node has non-infinite distance:
  - If node is D, found, follow path back!
  - Remove node from priority queue
  - For each neighbor, update distance and prev node

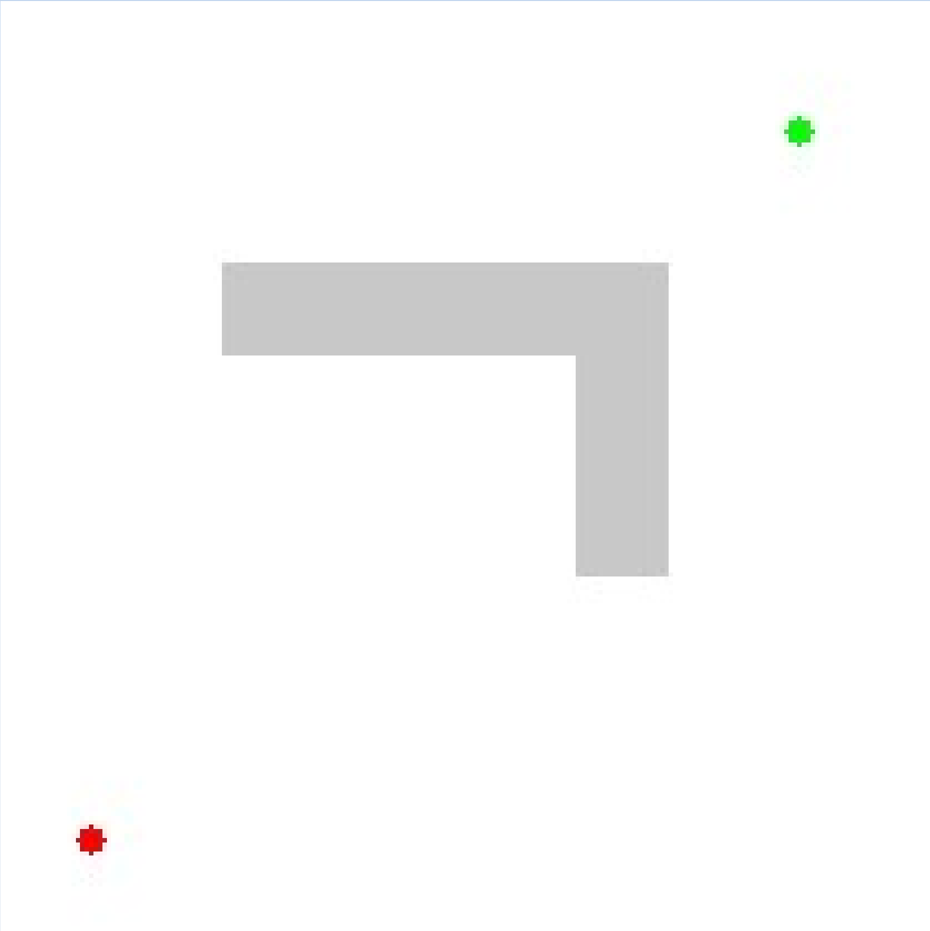
Navigation – Dijkstra's Algorithm



Navigation – Dijkstra's Algorithm

- Performance Concerns:
  - You end up visiting a lot of irrelevant nodes
  - Worst case you have to visit every single node
- DO NOT CALL EVERY FRAME

Navigation – A* Algorithm



- Visit nodes in order of “total estimated cost” = time to get to node + estimated time to get to destination
- For each node, remember parent

Navigation – A* Algorithm

NOTE:

You will commonly find discussion of an “open” and “closed” list in implementations of the A* algorithm. This is an optimization and not strictly necessary.

Navigation – A* Algorithm

- F = total expected cost (G + H)
- G = cost to get to current node
- H = estimated cost to get to node

- Visit nodes in F order
  - (Dijsktra's Algorithm visits nodes purely in G order)

Navigation – A* Algorithm

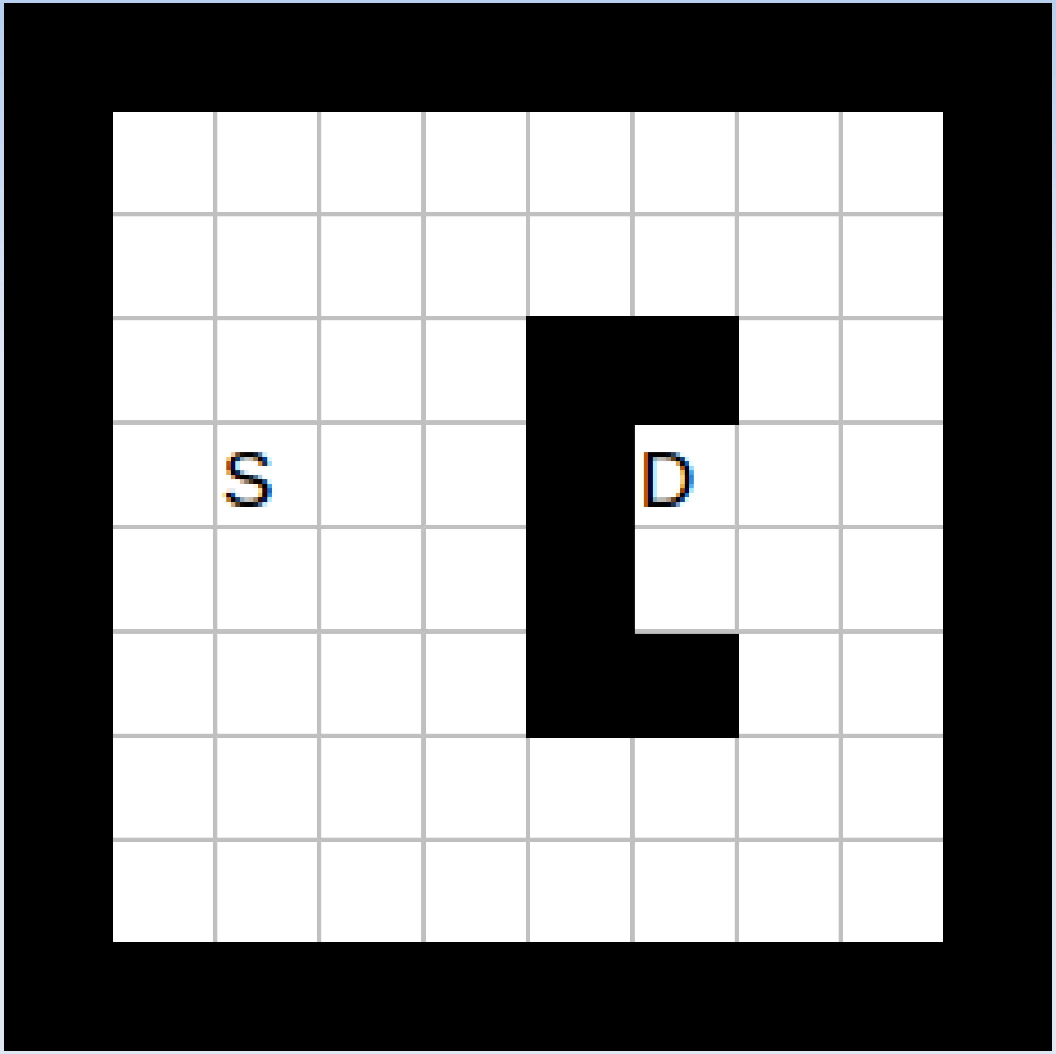
- Create a priority queue of all nodes
- **Calculate H for all nodes**
- **Mark S with G=0, all other nodes as infinity**
- While the cheapest node has non-infinite distance:
  - If node is D, found, follow path back!
  - Remove node from priority queue
  - For each neighbor, update **G** and prev node

Navigation – A* Algorithm

- About those Open and Closed sets...
  - They just makes finding the cheapest node faster

- The Open Set is all nodes that have non-infinite F, so a G has been calculated
- The Closed Set is all nodes that have been removed from the priority queue.

Navigation – A* Algorithm



Navigation

- Use A* when you know where to go, but you don't know how to get there
- Use Dijkstra's when you don't know where to go or how to get there
- And there will be a bonus algorithm next class!

Navigation – Game Loop

- Runtime performance of pathfinding is SPIKEY
  - Very slow, but not usually needed
  - May end up with multiple pathfinds needed in a single frame.
- Do as much as you can each frame, but don't go over some ms budget
  - Check time after every pathfind, and if you've gone over budget, stop pathfinding until the next frame