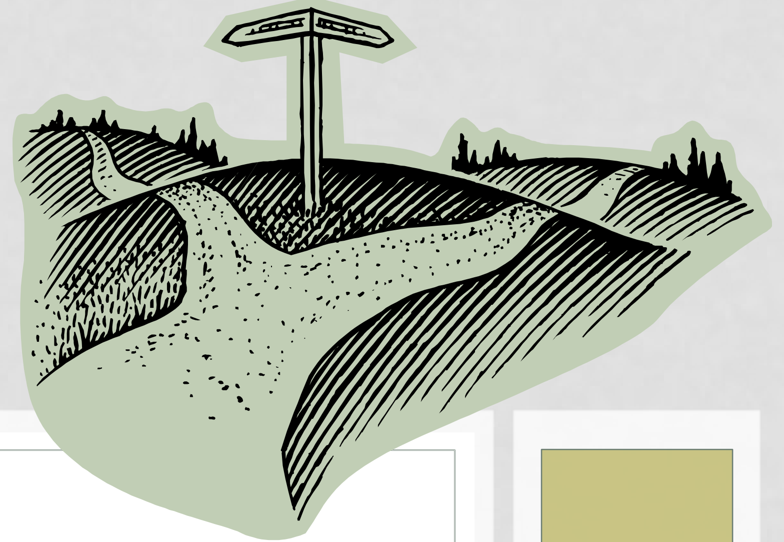
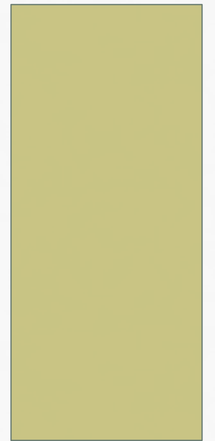


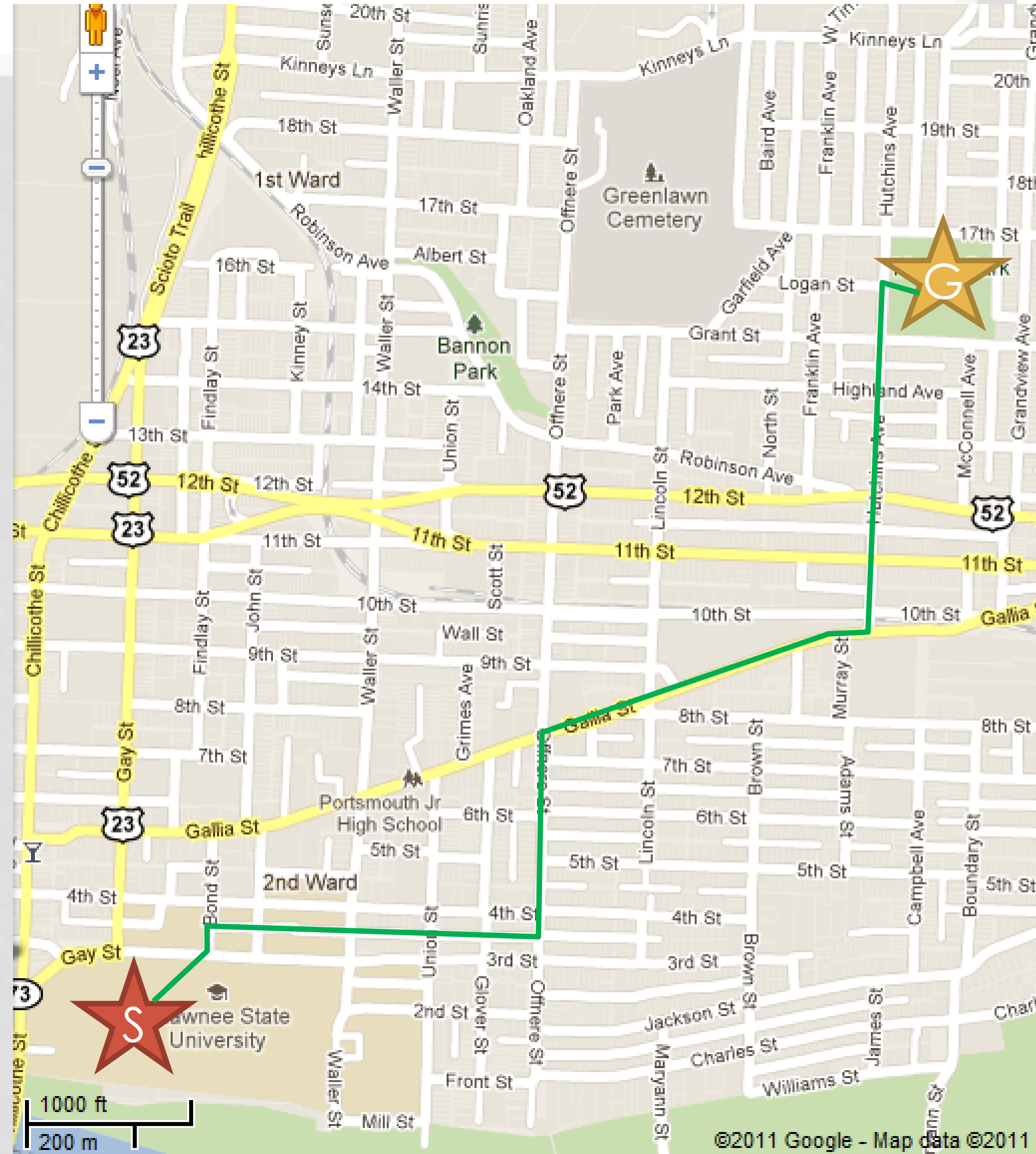
PATHFINDING

REFERENCE: "ARTIFICIAL INTELLIGENCE FOR GAMES", IAN MILLINGTON.



GOALS

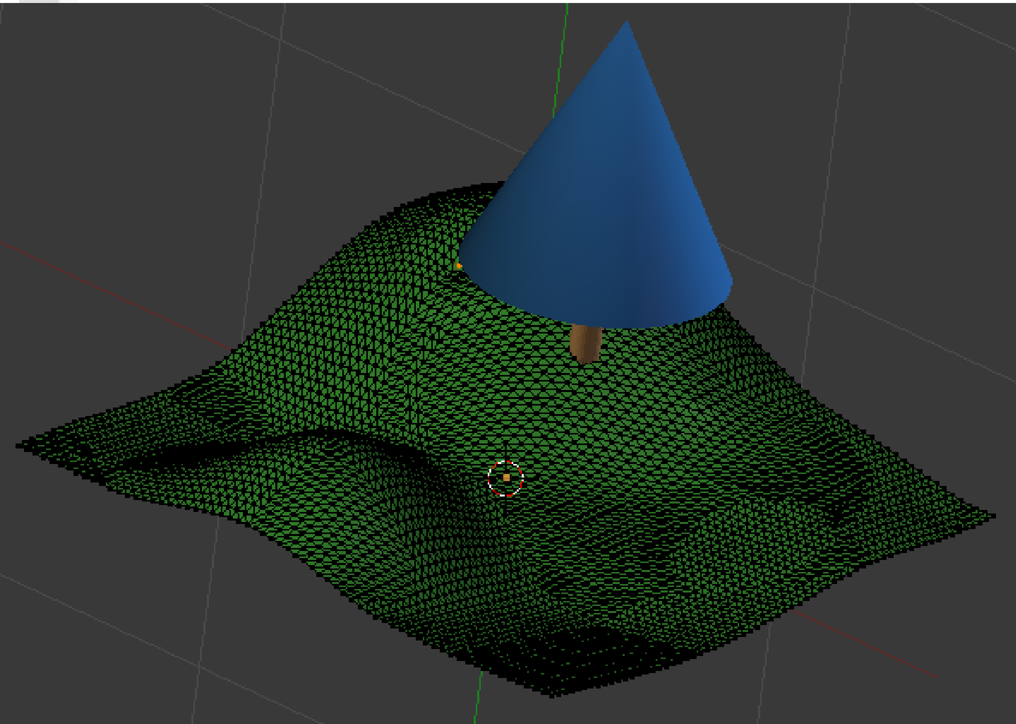
- Find a “pretty-short” path from point A to point B.
  - The *best* path is often prohibitively expensive to find.



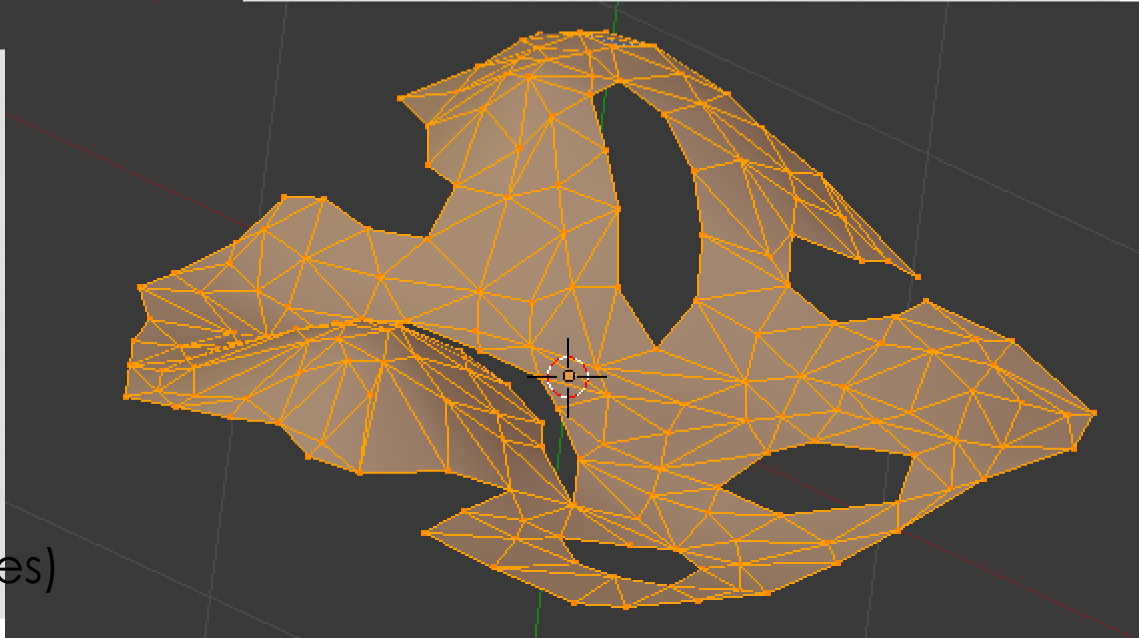
NAV-MESHES

- A common way to represent:
  - Walkable areas of a map.
  - Cover-spots
  - Jumping-points (to cross a chasm)
  - Ladders
  - Save points, ammo drops, etc.
- Key Ideas:
  - Not usually visible
  - Usually much lower poly-count than the actual ground
- We'll use these in this lab to represent the **search graph**.

NAV-MESHES, CONT.



Display Mesh (16541 faces)



Nav Mesh (301 faces)

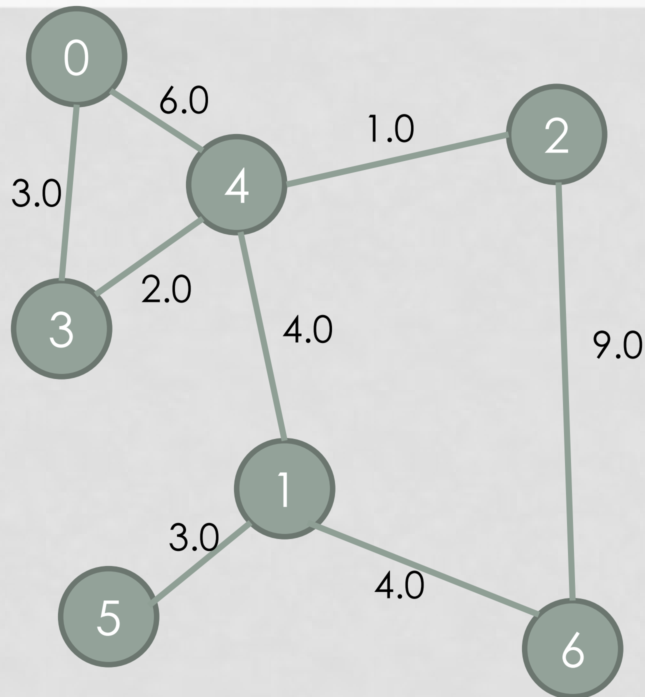
NAV-MESHES, CONT.

- We can use these for pathfinding…
  - Nodes are faces (their center?)
  - Edges are connections between neighboring faces
    - Cost is just the euclidean distance between centers.
- Probably best done off-line
  - Finding neighbors can be costly.

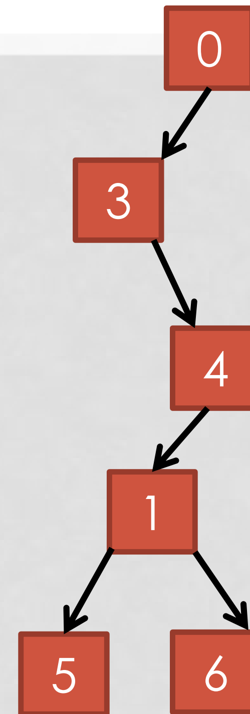
GRAPH-BASED ALGORITHM

- Graph
  - Vertices (Nodes)
  - Edges
    - Single edge
    - Double edge
    - Cost
      - (For map traversal), distance is the norm
      - Cost multiplier (>1) for rough terrain.
- Representation
  - Adjacency Matrix
  - Adjacency List
- [Connections to NavMesh]

GRAPH NODES VS. SEARCH NODES



Graph



Search Tree

Start == 0, Goal == 6

There are usually many search trees for the same graph.

GRAPH NODES VS. SEARCH NODES, CONT.

- Recommended approach:
  - GraphNode class:
    - [Any map-related data]
    - Neighbor_**pointers** = […]
  - SearchNode class (I actually just used a tuple here):
    - GraphNode reference
    - [optional] Heuristic value (later in A*)
    - Cost-so-far value
  - SearchTree <u>map</u>
    - Key = GraphNode reference
    - Value = parent GraphNode reference (or null for root)
      - I actually stored this plus the depth and heuristic.
- Create SearchNodes and add to map as you explore the map.
  - Positive: Easy to tell if you've visited a node – it's in the map.
  - Positive: Easy to restart the search – just throw out the map
    - DON'T make any references to the search in the GraphNodes!

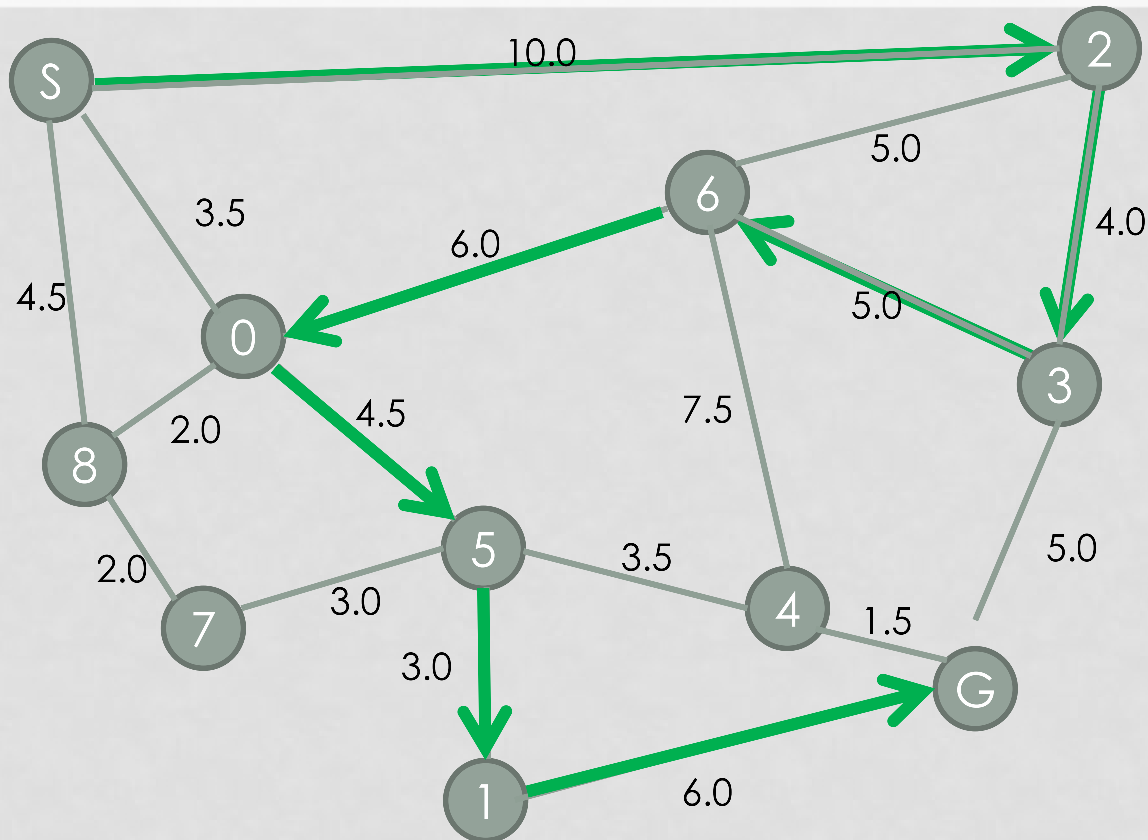
ACTUAL SEARCH

- Start with initial node, add it to the map
- Which node do we visit next?
  - Depth-First Search
  - Breadth-First Search
  - Djikstra Search
  - A* Search
  - …

DEPTH-FIRST P.C.

```
# Could be faster w/ a stack (non-recursive)
def dfs(curSNode, parentMap, mainGraph, g2sMap):
      for n in curSNode.GNode.neighbors:
             # [optional] early termination if n is goal.
             if n not in g2sMap:
                    newCost = curSNode.costSoFar +
                            mainGraph.edgeCost(curSNode.GNode, n)
                    newSNode = new SNode(n, newCost)
                    parentMap[newSNode] = curSNode
                    g2sMap[n] = newSNode
                    dfs(newSNode, parentMap, mainGraph, g2sMap)
initSNode = new SNode(start, 0)
searchTree = {initSNode : null}
graph2searchMap = {start : initSNode}
mapGraph = …
dfs(initSNode, searchTree, mapGraph, graph2searchMap)
```

DEPTH-FIRST SEARCH

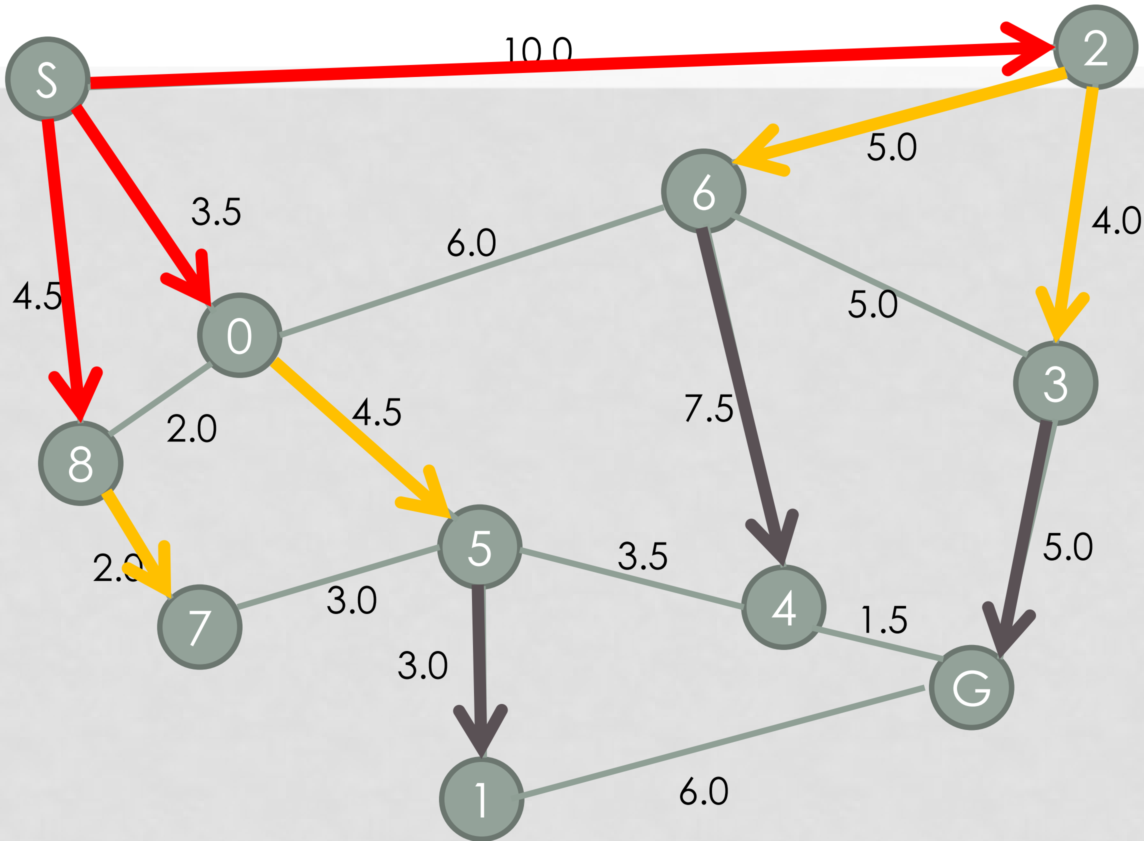


The path we get will, in general, not be the best

BREADTH-FIRST SEARCH P.C.

```
initSNode = new SNode(start, 0)
searchTree = {initSNode : null}
graph2searchMap = {start : initSNode}
mapGraph = …
frontier = [initSNode]
while len(frontier) != 0:
        new_frontier = []
        for cur in frontier:
                # [optional] early termination if
                # cur is the goal
                for n in cur.GNode.neighbors:
                        if n not in graph2searchMap:
                                newCost = cur.costSoFar +
                                        mapGraph.edgeCost(s.GNode, n)
                                newSNode = new SNode(n, newCost)
                                searchTree[newSNode] = cur
                                graph2searchMap[n] = newSNode
                                new_frontier.append(newSNode)
        frontier = new_frontier
```

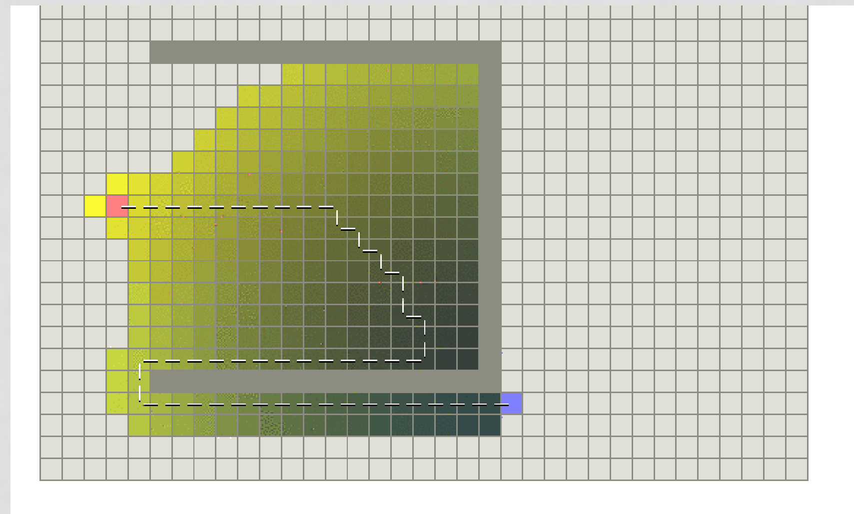
BREADTH-FIRST SEARCH



Again, we don't necessarily get the "best" path.

GREEDY-FIRST

- Just like BFS, but pick the node that *appears* to take us closer to goal
  - For this, we need some way to calculate a **heuristic**: a <u>guess</u> at distance to goal
  - This is usually an under-estimate – see why?
- Problem:



http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

DJIKSTRA'S ALGORITHM P.C.

- Like BFS also, but solves the greedy-first problem.
- But…does more work than necessary.

```
# … everything else as with BFS
for n in cur.GNode.neighbors:
      newCost = cur.costSoFar +
                                mapGraph.edgeCost(s.GNode, n)

      if n not in graph2searchMap:
             newSNode = new SNode(n, newCost)
             searchTree[newSNode] = cur
             graph2searchMap[n] = newSNode
      elif newCost < graph2searchMap[n].costSoFar:
             searchTree[graph2searchMap[n]] = cur
             graph2searchMap[n].costSoFar = newCost
```

- Better than BFS because we *re-route parent pointers if we* ***find a better path***.
- We aren't guaranteed the absolute optimal solution, but it's a pretty-good soln.

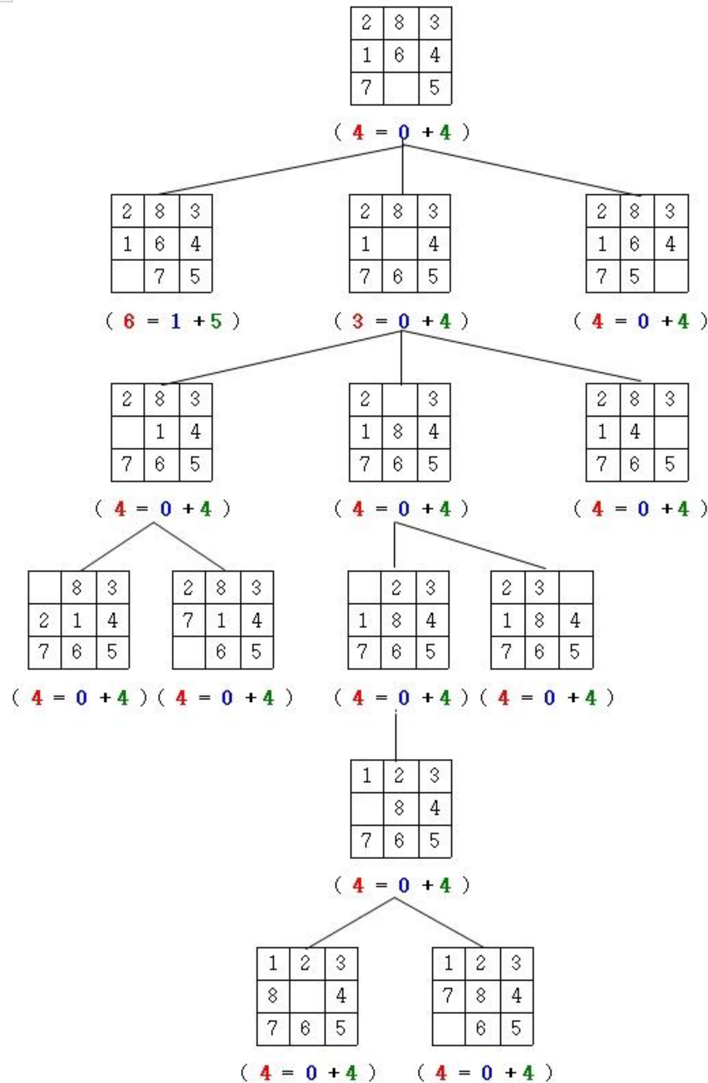
A* ALGORITHM

- Rather than looking at all neighbors in turn, pick the one we *think* will lie on the best path.
  - A combination of actual_cost + heuristic cost
  - Note how greedy-first only considered heuristic cost.
  - f(x) = g(x) + h(x)   # Total = actual + heuristic
  - It is vital that the heuristic be **consistent**: i.e. less than or equal to the actual cost
    - Otherwise, nodes we've visited must be re-examined.
  - We also don't want the hueristic to be too low: otherwise we'll look at too many nodes.
- Re-route visited nodes, like Djikstra

A* P.C.

```
initSNode = new SNode(start, 0)
searchTree = {initSNode : null}
open = new MinHeap() # Of SearchNode’s.  Order by totalCost (actual + heuristic)
open.push(initSNode)
closed = new set()
graph2searchMap = {start : initSNode}
mapGraph = …
while !open.empty():
        cur = open.pop()
        closed.add(cur)
        # [not optional here] Early termination if cur is the goal
        for n in cur.GNode.neighbors:
                newCost = cur.costSoFar + mapGraph.edgeCost(s.GNode, n)
                if n not in graph2searchMap:
                        newSNode = new SNode(n, newCost)
                        searchTree[newSNode] = cur
                        graph2searchMap[n] = newSNode
                        open.push(newSNode)
                elif n not in closed and newCost < graph2searchMap[n].costSoFar:
                        # Better path through us - like Djikstra’s
                        graph2searchNode[n].costSoFar = newCost
                        searchTree[graph2searchNode[n]] = cur
                        # Important: reheapify open!
                # Not necessary if your heuristic is consistent.  If not, we need to re-open
                # closed nodes
                # elif newCost < graph2searchMap[n].costSoFar:
                #         graph2searchNode[n].costSoFar = newCost
                #         searchTree[graph2searchNode[n]] = cur
                #         closed.remove(graph2searchNode[n])
                #         open.add(graph2searchNode[n])
return failure
```

ANOTHER APPLICATION OF A*



THE "NEW" A* ?

- Jump-Point Search seems to be gaining popularity in game programming.
- Good reference:
  - https://harablog.wordpress.com/2011/09/07/jump-point-search/
- Makes many improvements, but only works on square-grids.
  - Can we make this work with nav-meshes?