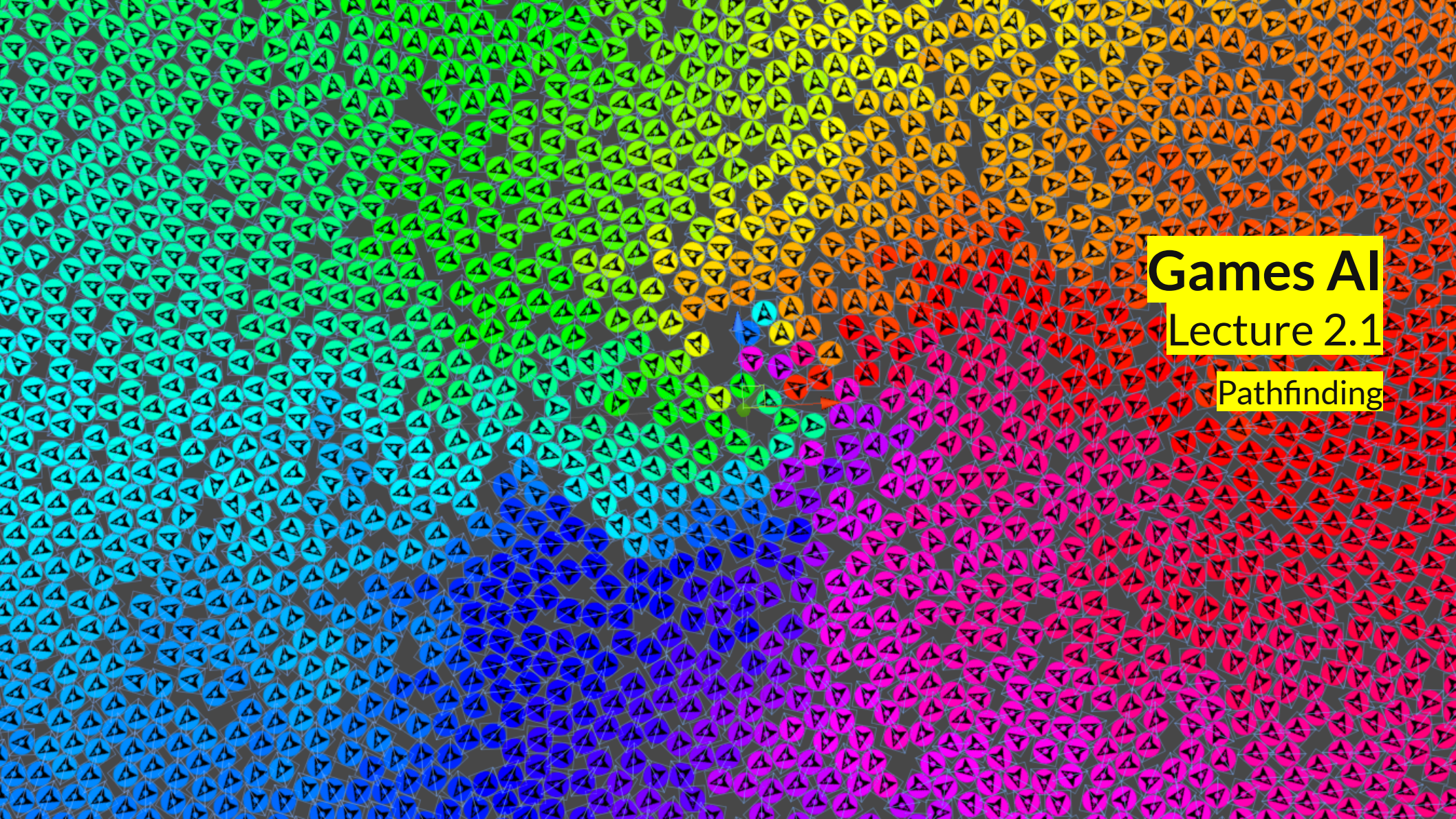
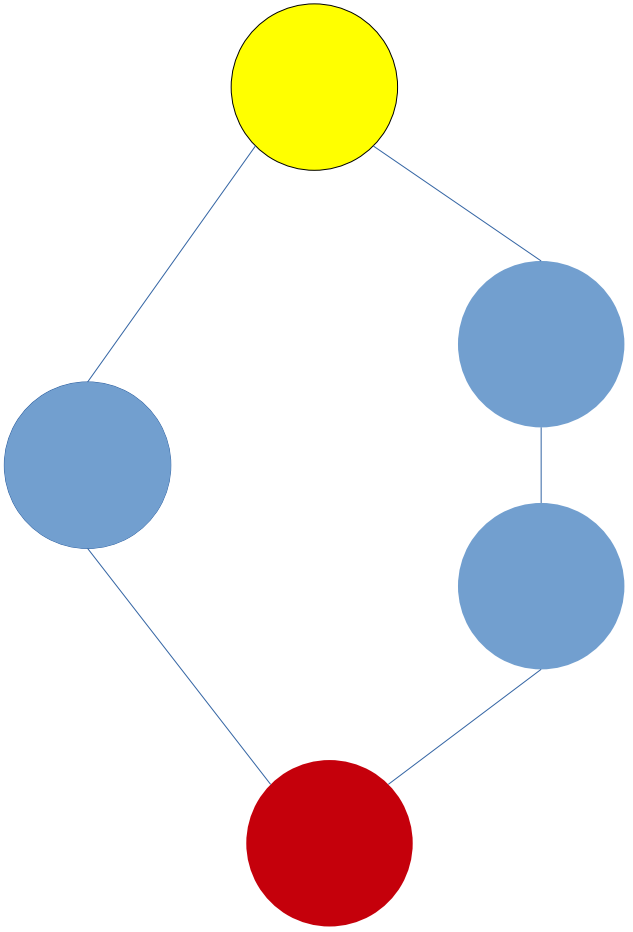
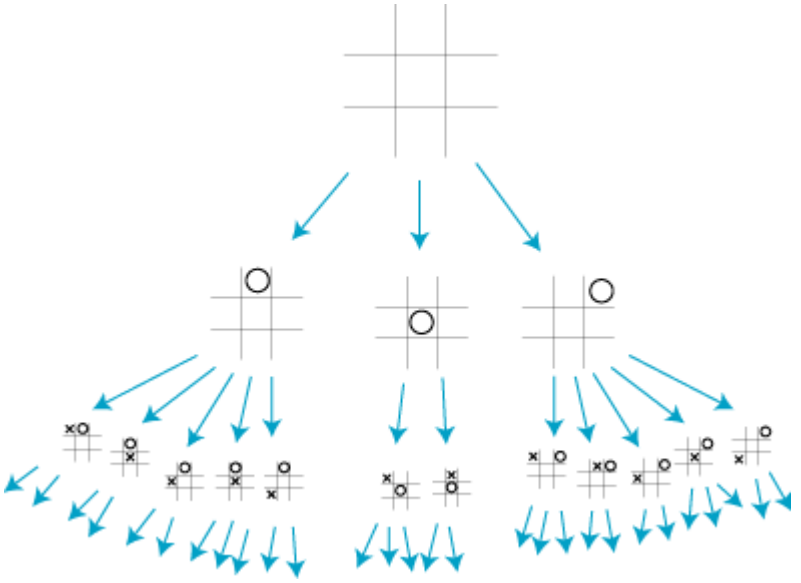

Games AI
Lecture 2.1
Pathfinding

- Pathfinding
  - Breadth-first search
  - A* search
  - Heuristics
  - Navigation graphs
  - A* variants

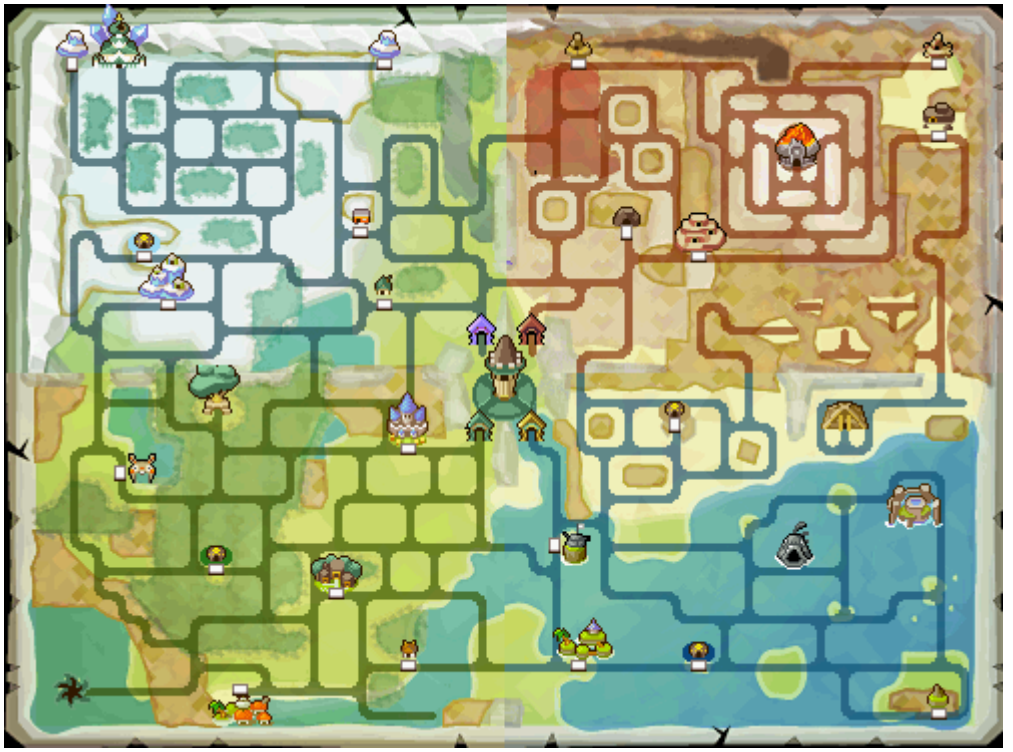
- Graph Search
  - What is a graph?



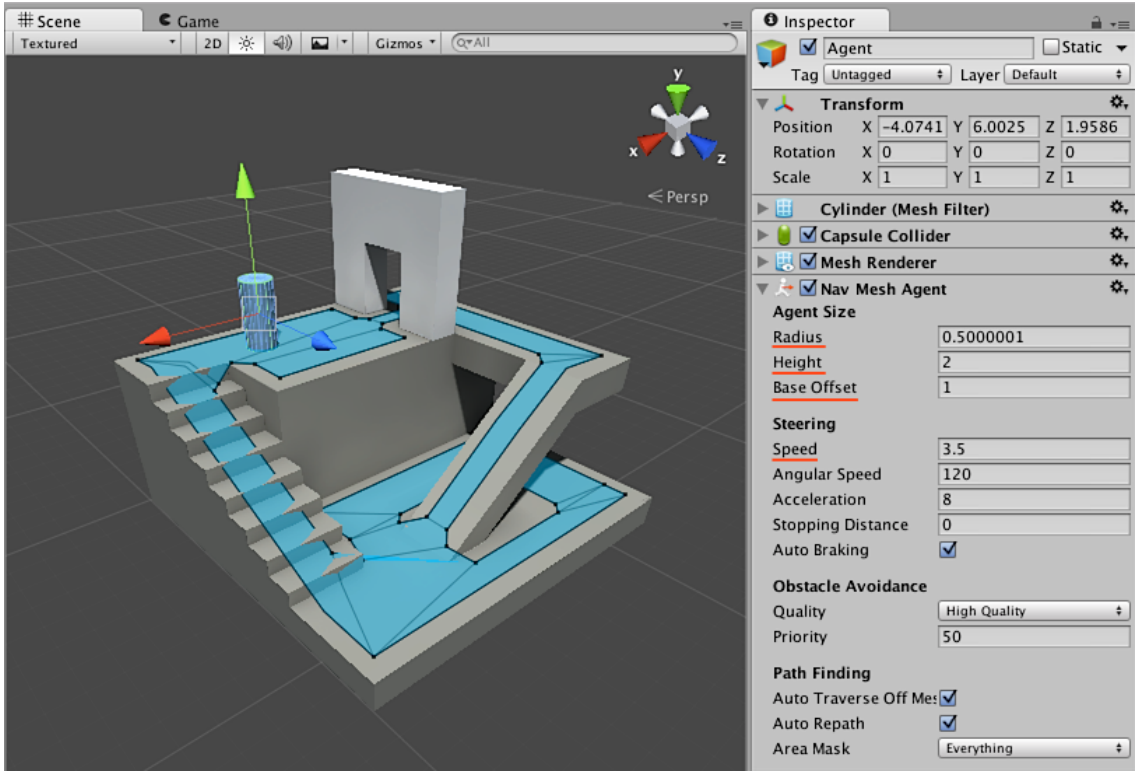
- Graph Search
  - What is a graph?



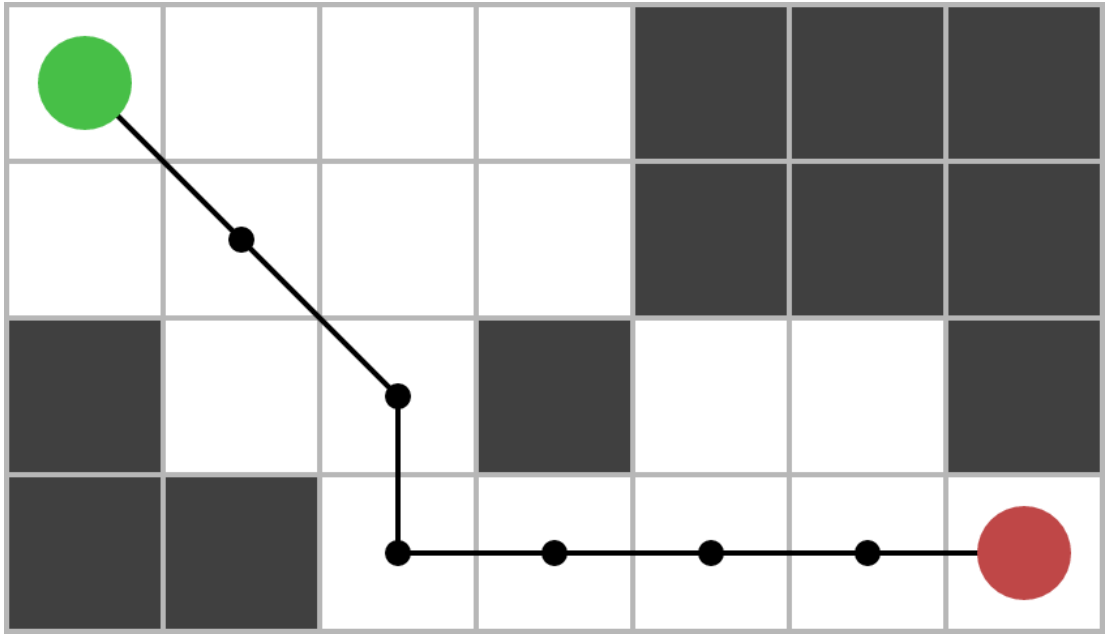
- Graph Search
  - What is a graph?

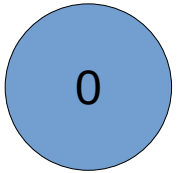


- Graph Search
  - What is a graph?



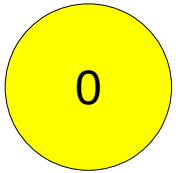
- Graph Search
  - What is a graph?





Queue
0

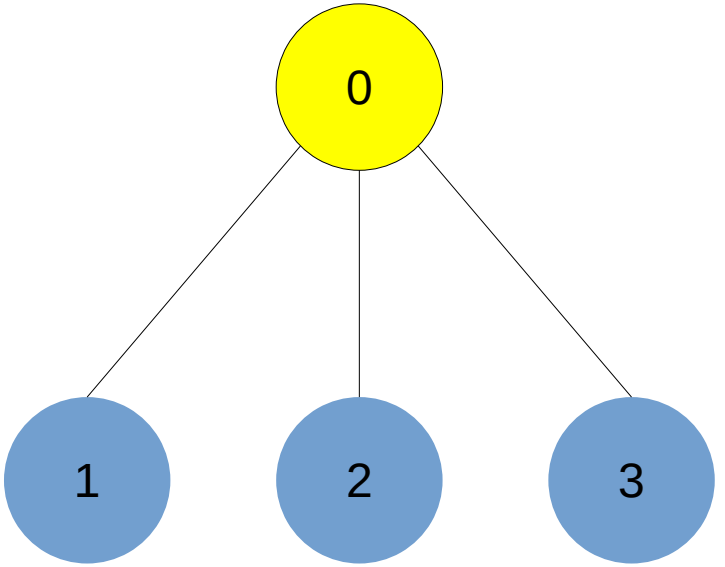
- Breadth-first search
- Set up:
  1. Create queue
  2. Add start to queue
  3. Mark as discovered



Queue

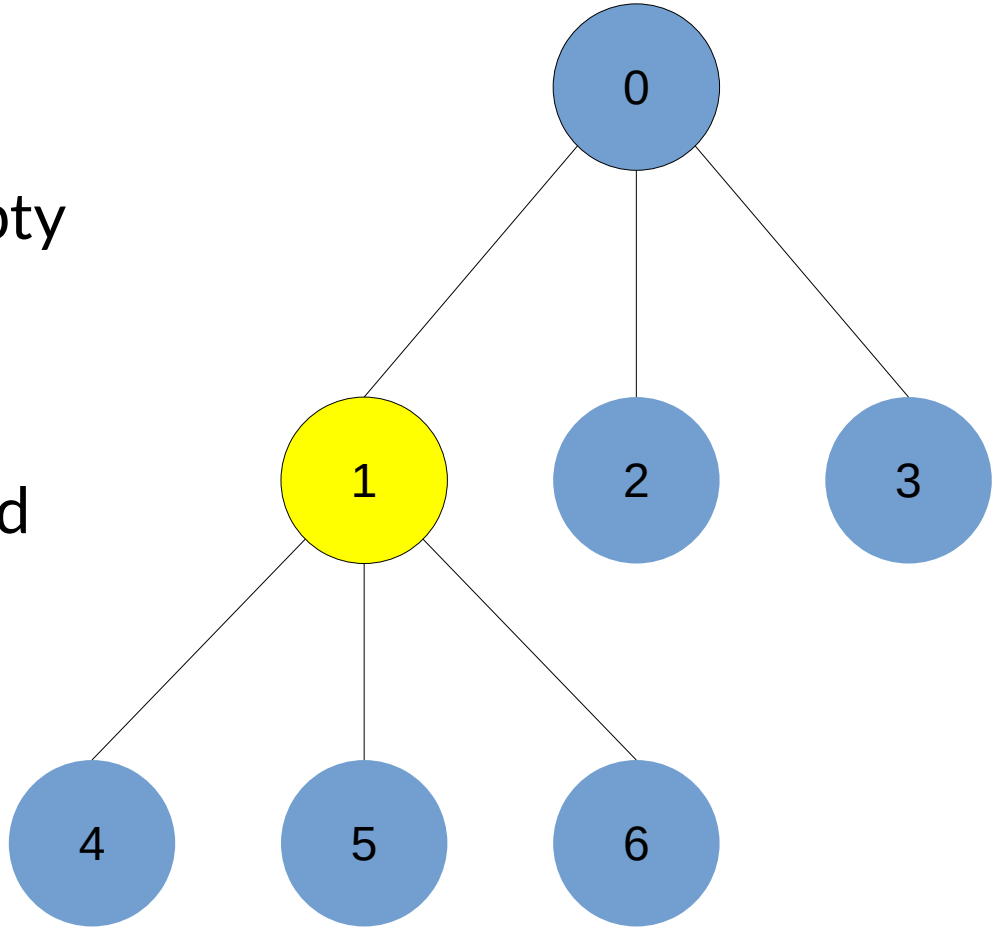
- While queue is not empty
  1. Select top of queue
  2. Check if goal

- While queue is not empty
  1. Select top of queue
  2. Check if goal
  3. Add all undiscovered children to queue
  4. Mark each child as discovered



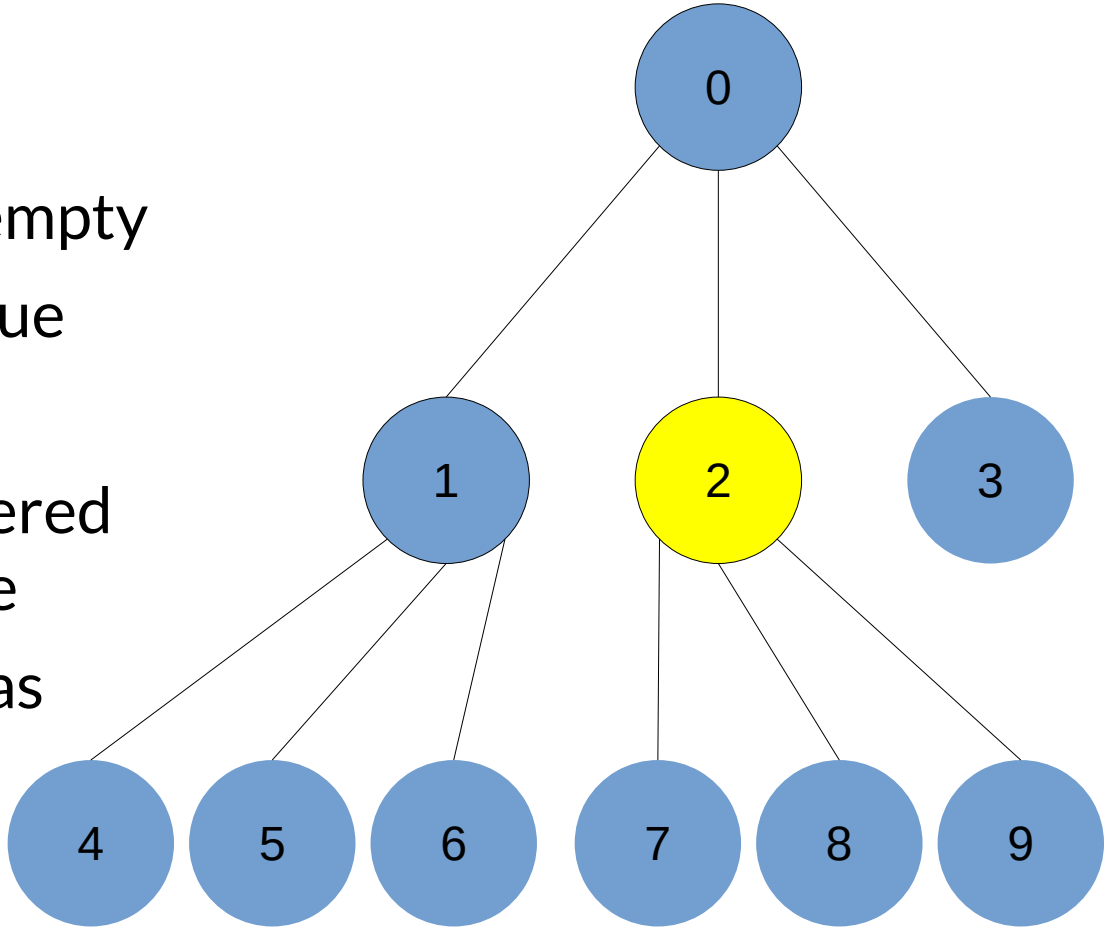
Queue
1
2
3

- While queue is not empty
  1. Select top of queue
  2. Check if goal
  3. Add all undiscovered children to queue
  4. Mark each child as discovered



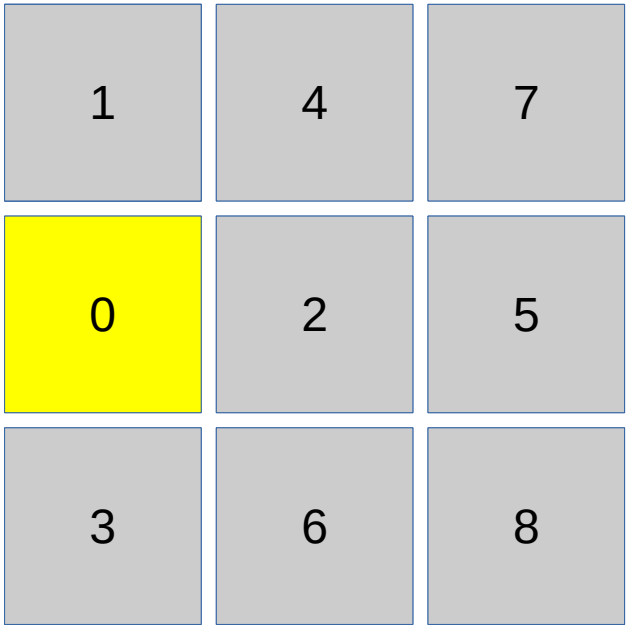
Queue
2
3
4
5
6

- While queue is not empty
  1. Select top of queue
  2. Check if goal
  3. Add all undiscovered children to queue
  4. Mark each child as discovered

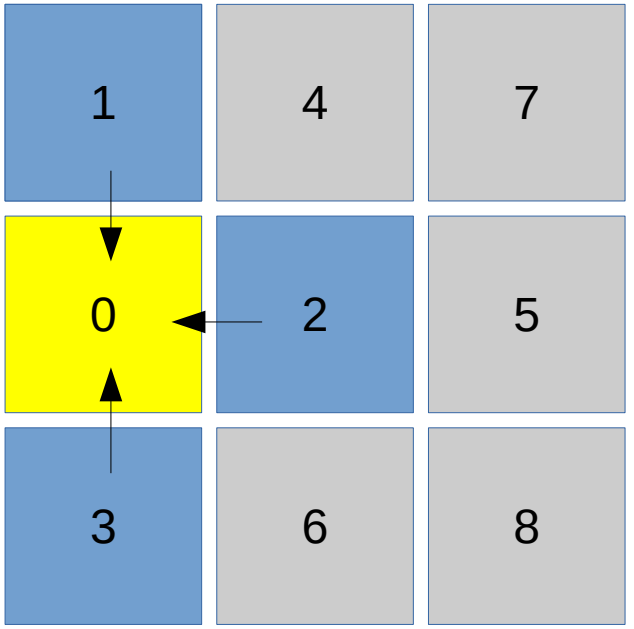


Queue
3
4
5
6
7
8
9

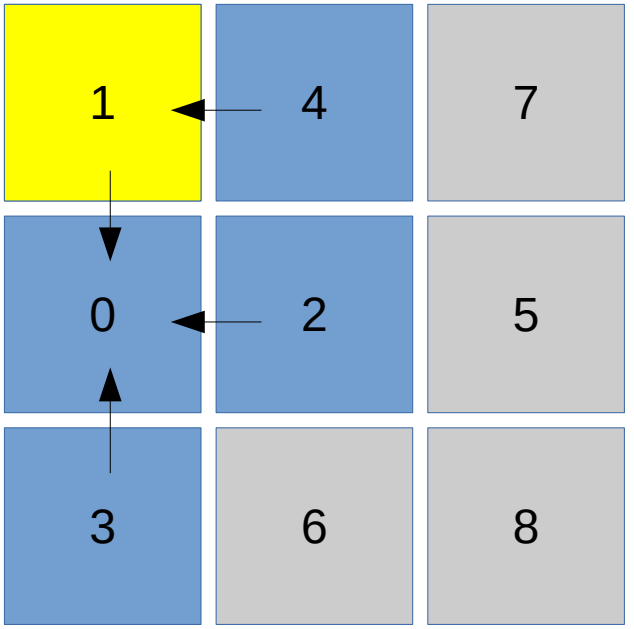
BFS



Queue:

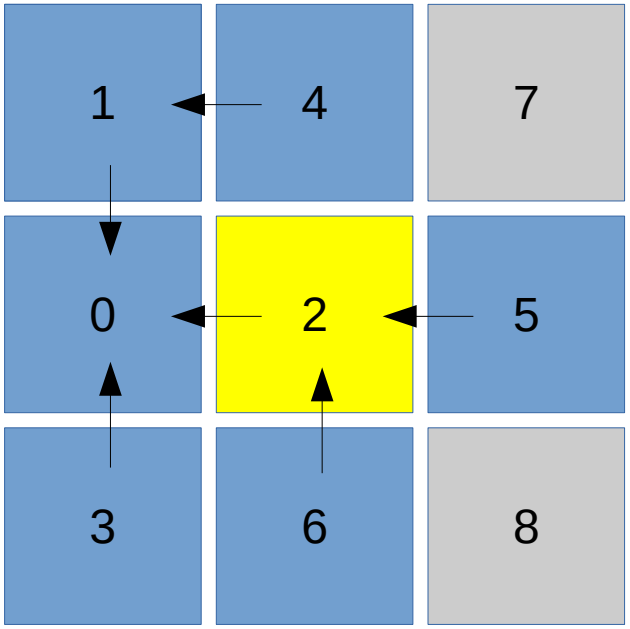


Queue: 1,2,3

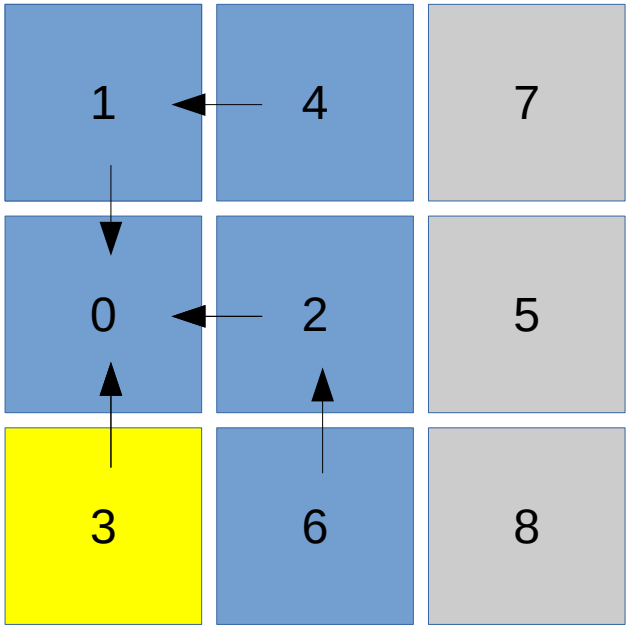


Queue: 2,3,4

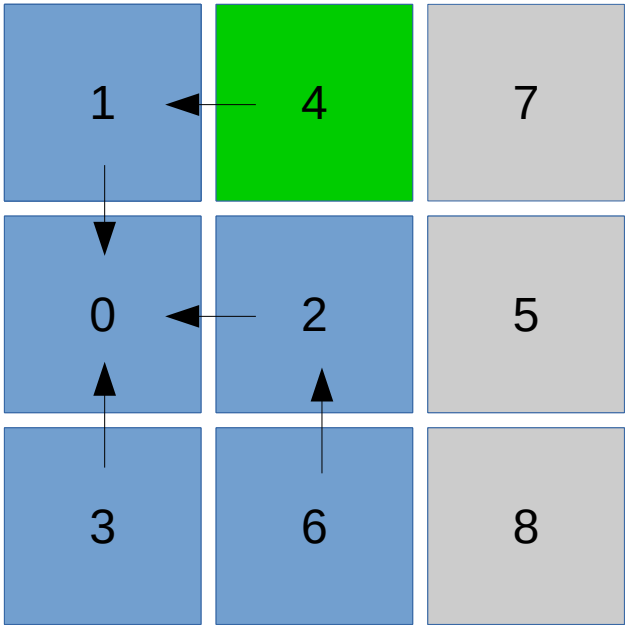
BFS



Queue: 3,4,5,6



Queue: 4,5,6

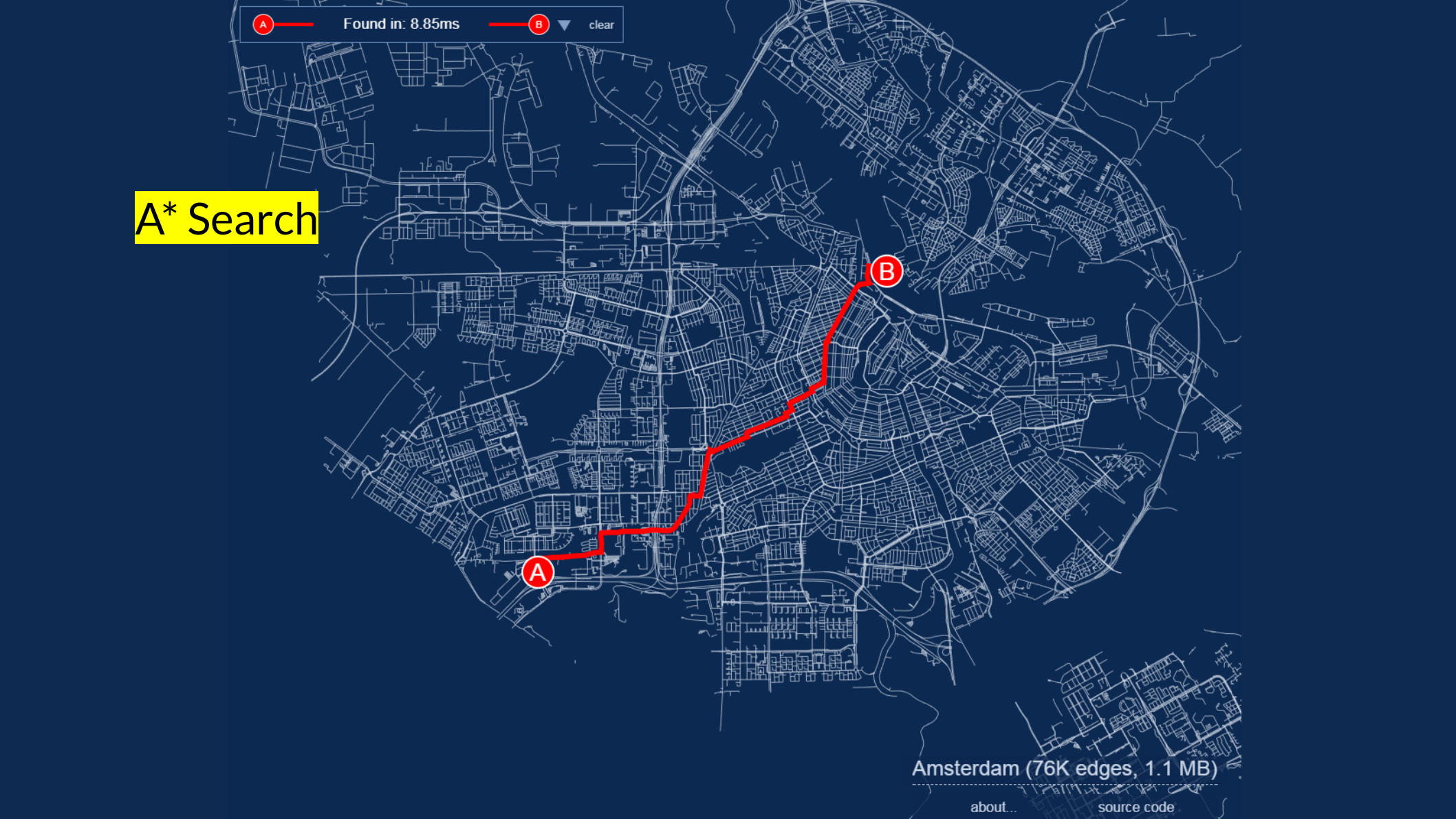


Queue: 5,6,7

```
1. procedure BFS(G: Graph, v: Vertex of G) is
2.    create a queue Q
3.    enqueue v onto Q
4.    mark v
5.    while Q is not empty do
6.       w ← Q.dequeue()
7.       if w is what we are looking for then
8.          return w
9.       for all edges e in G.adjacentEdges(w) do
10.         x ← G.adjacentVertex(w, e)
11.         if x is not marked then
12.            mark x
13.            enqueue x onto Q
14.   return nulls.
```

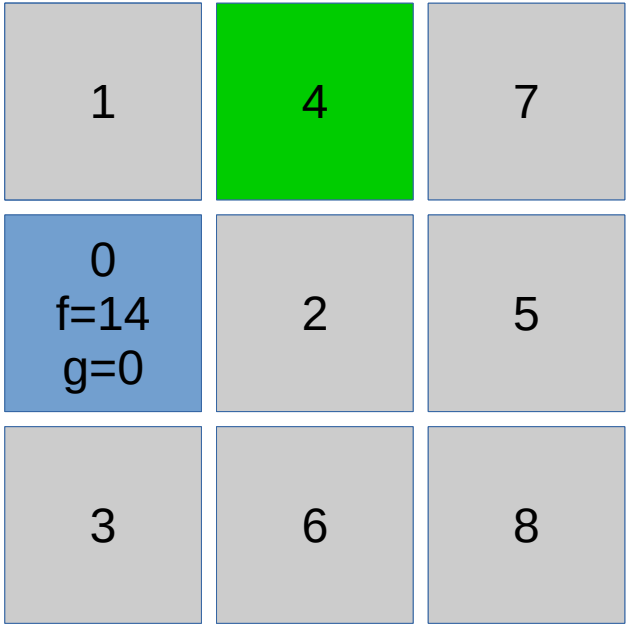
```
breadthFirstSearch(initialState: T): A[]
{
    this.openList.push(new TreeNode<T, A>(initialState, null, null));
    while (this.openList.length > 0)
    {
      let current = this.openList.shift();
      if (this.goalTest(current.state))
          return this.getActions(current);
      let edges = this.stateActionMapping(current.state)
      for (let i=0;i<edges.length;i++)
      {
        let newState = this.transitionModel(current.state, edges[i]);
        let duplicate = false;
        for (let i=0,len=this.closedList.length;i<len;i++)
          if (this.stateEquals(this.closedList[i], newState))
          {
            duplicate = true;
            break;
          }

        if (!duplicate)
        {
          let newNode = new TreeNode<T, A>(newState, edges[i], current);
          this.openList.push(newNode);
          this.closedList.push(newNode.state);
        }
      }
    }
    return null;
}
```

- Pros:
  - Finds shortest path (by number of nodes)
- Cons
  - Unguided

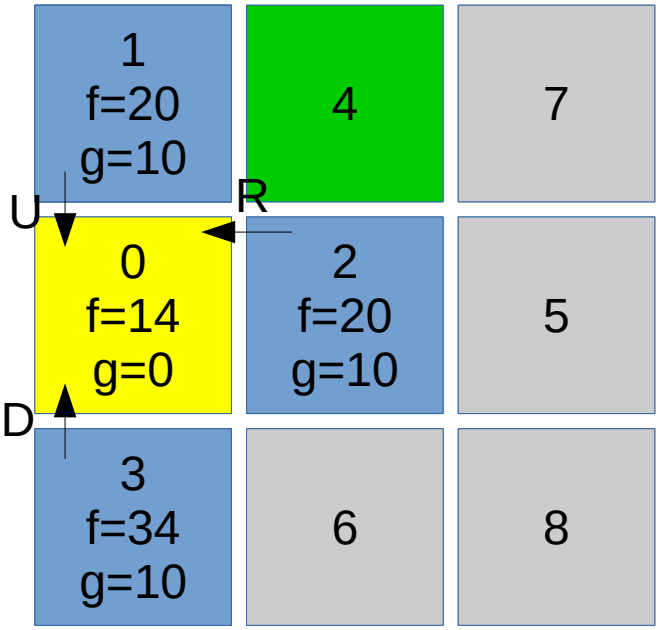

A
Found in: 8.85ms
B
clear
A* Search
B
A
Amsterdam (76K edges, 1.1 MB)
about...
source code

- A* Search
  - Guided / Best-first search
  - Prioritise nodes to explore based on cost + heuristic
    - **g(n)**: cost to reach node
    - **h(n)**: heuristic for node
    - **f(n)** = g(n) + h(n): priority of node

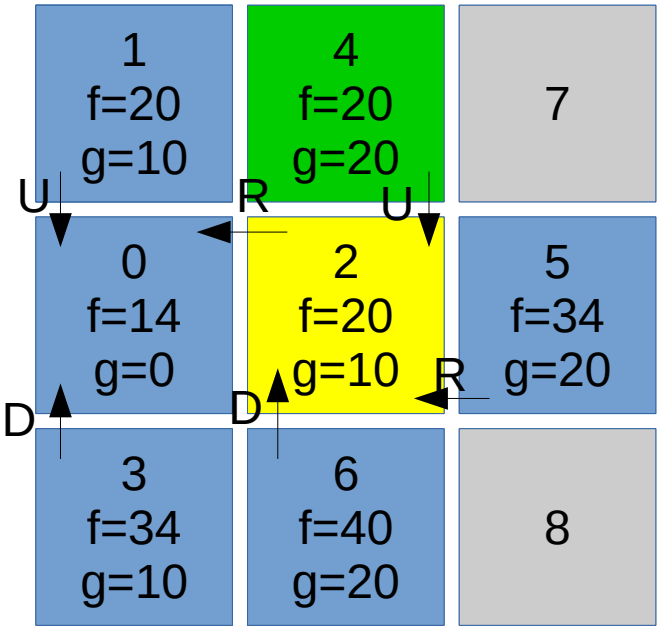
- Set up:
  1. Create empty list (Open set)
  2. Create maps from node → cameFrom, gScore, and fScore (or store f(n), g(n), and parent for each node)
  3. Add start node to Open Set



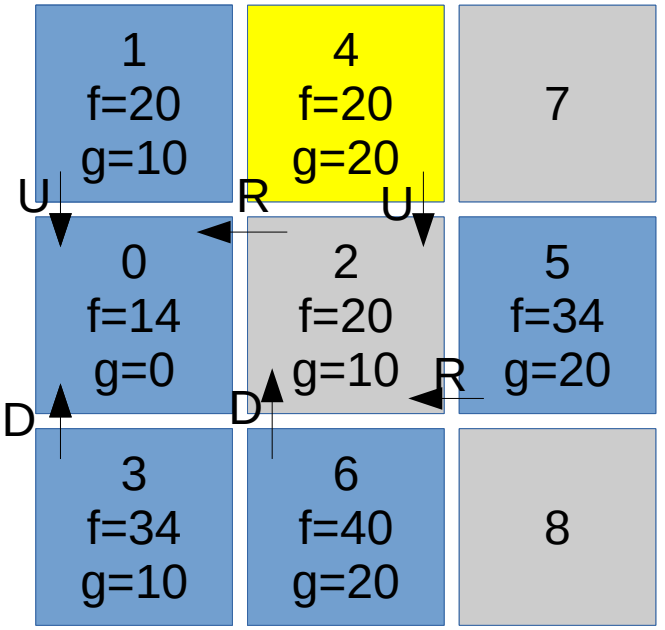
- While open set is not empty:
  1. Remove node in Open Set with lowest f(n)
  2. Check if goal
  3. For each neighbour n:
     1. If new g(n) smaller:
        1. Update g(n), f(n)
        2. Set parent to current node



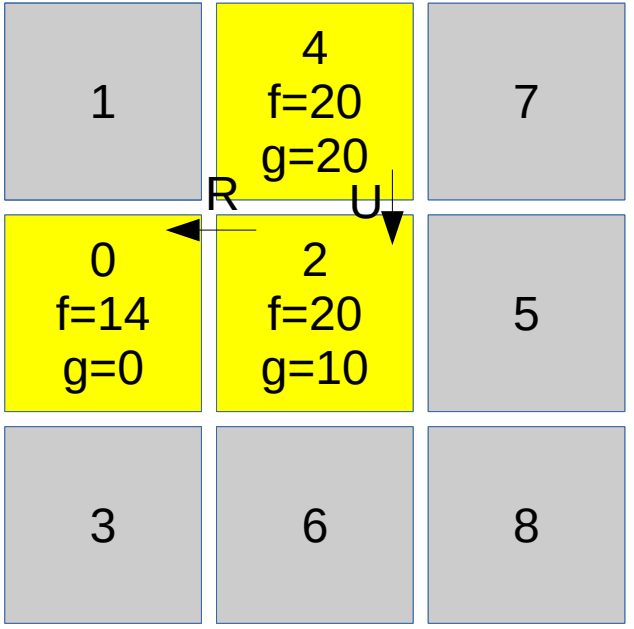
- While open set is not empty:
  1. Remove node in Open Set with lowest f(n)
  2. Check if goal
  3. For each neighbour n:
     1. If new g(n) smaller:
        1. Update g(n), f(n)
        2. Set parent to current node



- While open set is not empty:
  1. Remove node in Open Set with lowest f(n)
  2. Check if goal
  3. For each neighbour n:
     1. If new g(n) smaller:
        1. Update g(n), f(n)
        2. Set parent to current node



- Reconstruct path
  1. Create empty list
  2. Current node = goal node
  3. Add action to reach current node to list
  4. While current node has parent
     1. Set current node = parent
     2. Prepend action to react current node to list



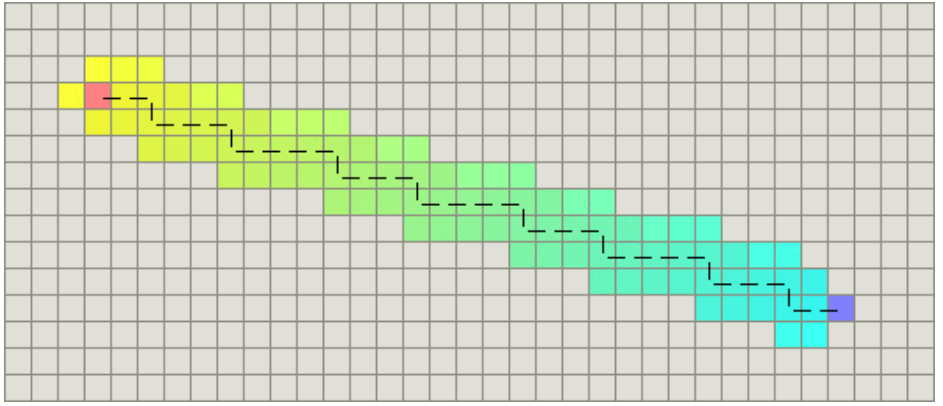

Heuristics

- Heuristics
  - **Quick to compute** approximation of value/distance to goal of a given node
  - For A* must be equal to or an under-estimate of true cost
  - Weighted heuristics: f(n) = w * h(n) + (w-1) * g(n)
    - If w = 0, A* is equivalent to **Dijkstra's algorithm**
    - If w = 1, A* is equivalent to **greedy best-first search**

- Manhattan distance

```
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

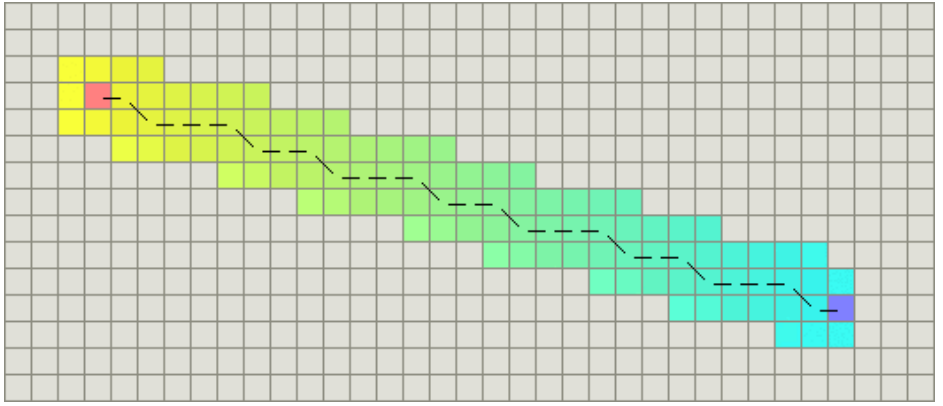
  - Best for 4-connected grids



- Diagonal distance

```
function heuristic(node)
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) +
                (D2 - 2 * D) *
                min(dx, dy)
```

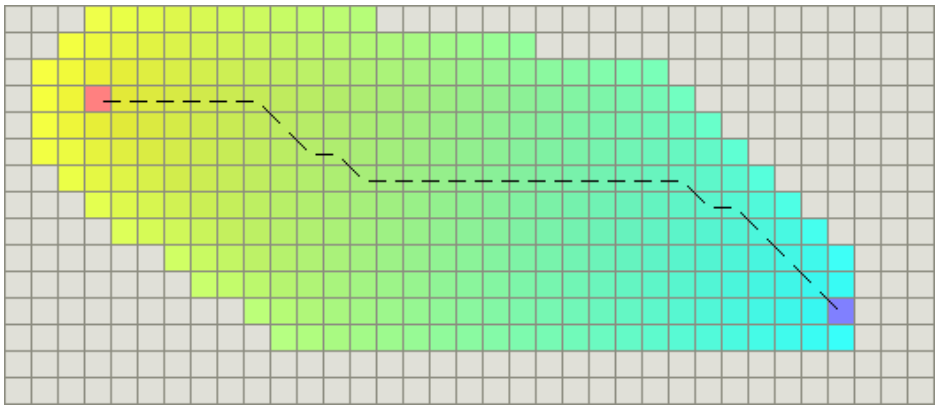
  - Best for 8-connected grids



- Straight line distance

```
function heuristic(node)
  dx = abs(node.x - goal.x)
  dy = abs(node.y - goal.y)
  return D * sqrt(dx * dx +
dy * dy)
```

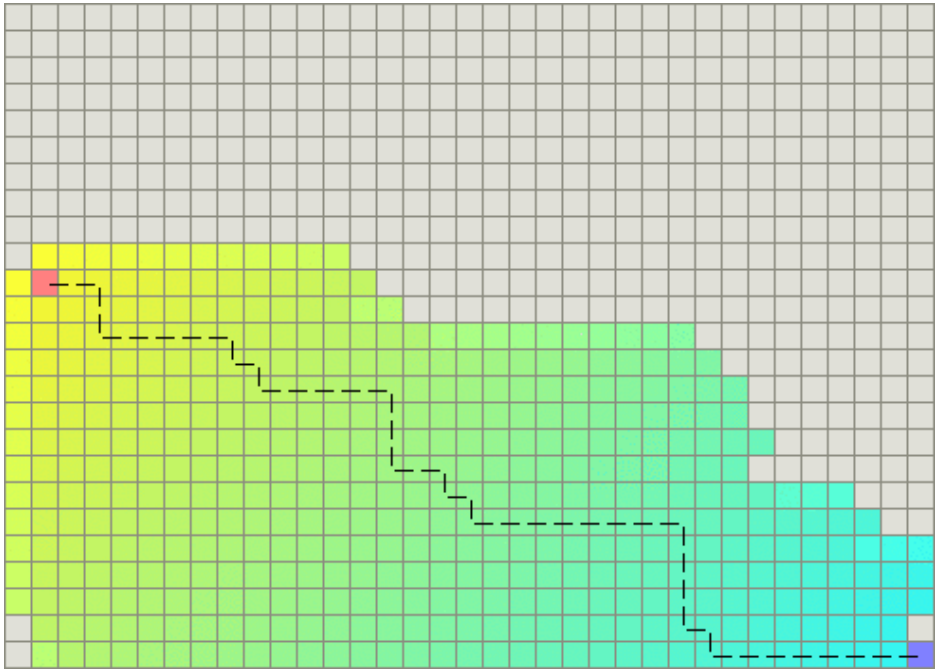
  - If you can move at any angle, but leads to more exploration as cost function *g* and heuristic *h* will not match



- If you have multiple goals

```
function heuristic(node)
    h1 = heuristicToA(node)
    h2 = heuristicToB(node)
    ...
    return min(h1, h2, ...)
```

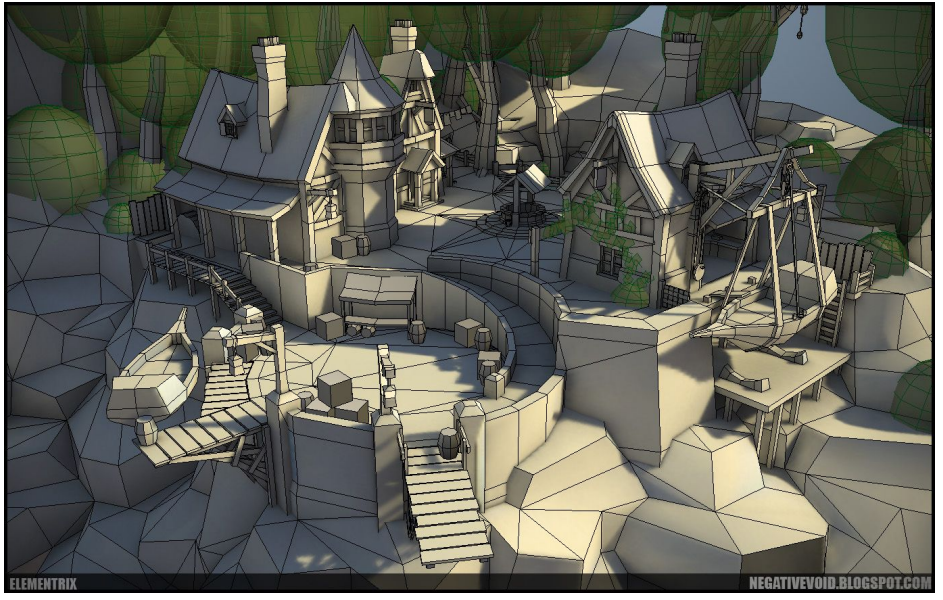
- Breaking ties
  - Often many paths with same length, leading to lots of unnecessary exploration
  - Tweaking A* heuristics can improve performance, but impacts on behaviour



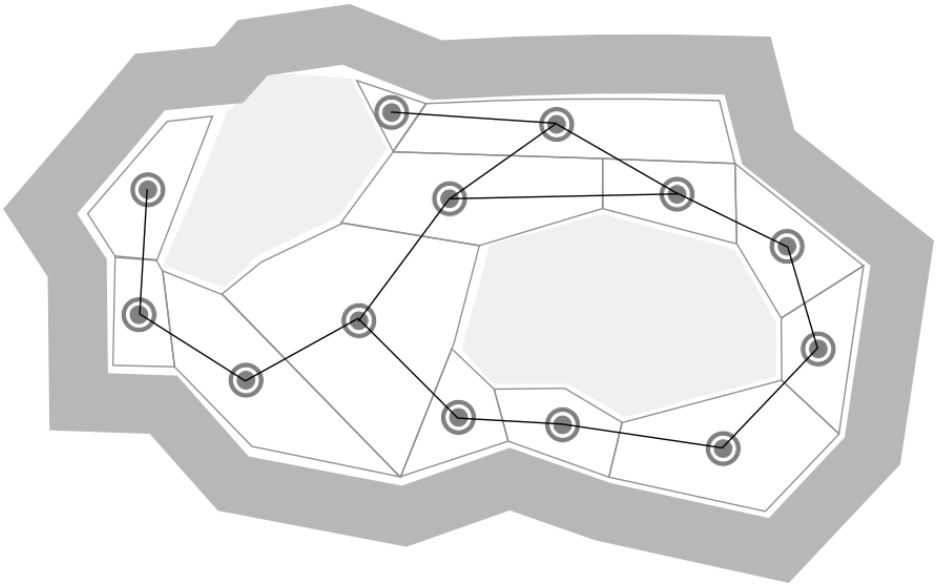
1. Implement an agent that uses A* to plan a path to a goal (e.g. the player)
    1. First **localise** the start and goal positions in terms of grid coordinates
    2. Then **search** to determine a path (sequence of grid cells/directions)
    3. Then **position** the grid cells back in world space and move between them.
2. Explore different options for tie-breaking and see how they affect behaviour and performance:
    - https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html


Map Abstractions
AiGameDev.com

- Game worlds are usually unsuitable for navigation
  - “Polygon soup”
  - Doesn’t encode navigable areas
  - Irrelevant detail



- Navigation graph
  - Graph size affects
    - Search performance
    - Path quality
  - Content creation problem

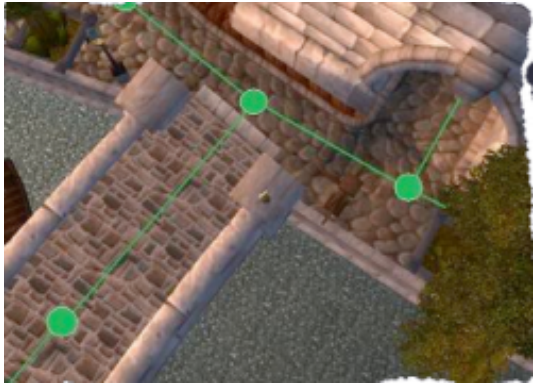


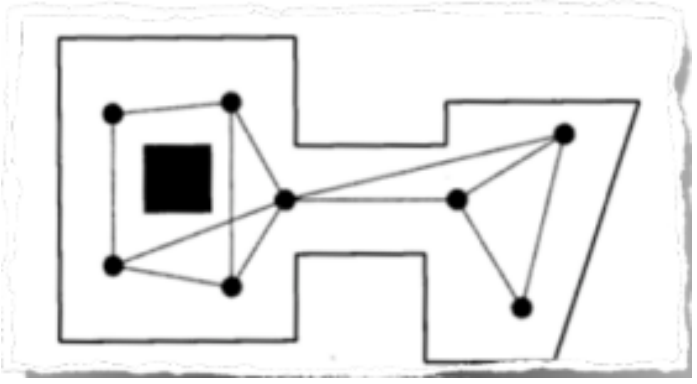
- Pathfinding tasks
  - **Localisation**: convert position to node
  - **Search**: find node path in nav graph
  - **Positioning**: convert node(s) to position(s)
  - **Smoothing**: improve quality of path

- Requirements
  - **Validity**: If two nodes are linked they are navigable in the world (the graph doesn't mislead)
    - Invalid structures result in paths that agent's can't follow
  - **Completeness**: All navigable spaces in the world are represented in the graph (the graph doesn't omit)
    - Incomplete structures prevent agents taking valid paths

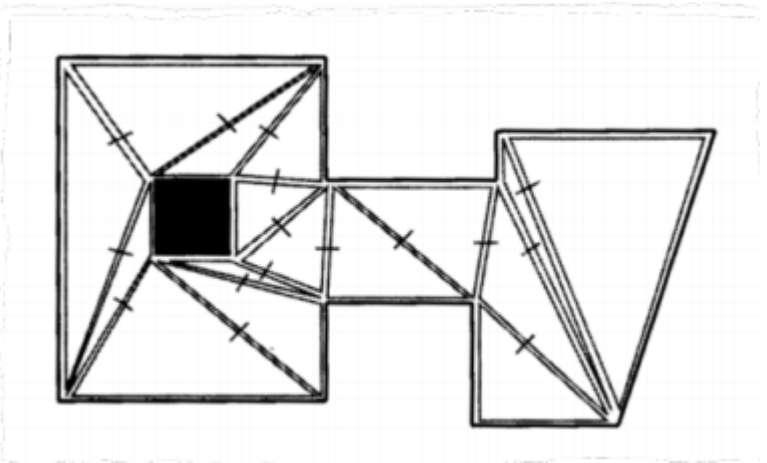
- **Grid Navigation**
  - Divide world into regular regions
  - Easy to generate
  - Large numbers of (unnecessary?) nodes

- **Waypoint Navigation**
  - Place waypoints and connections in the world
  - Difficult to generate
  - Localisation searches for nearest waypoint, but we need to validate movement to the waypoint (raycast)

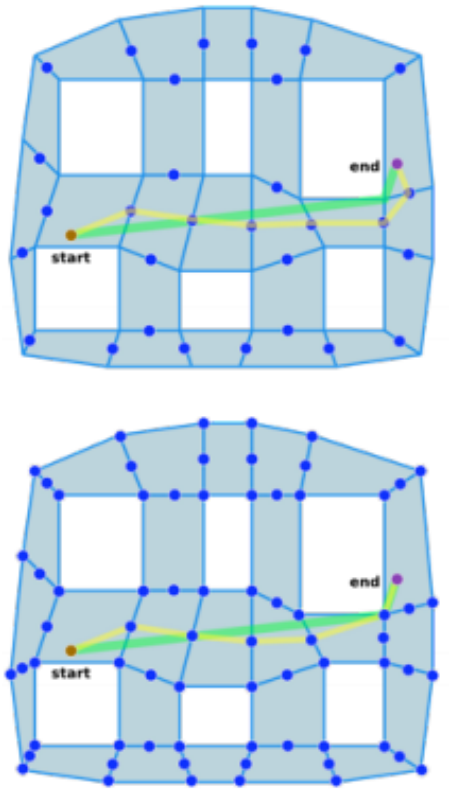




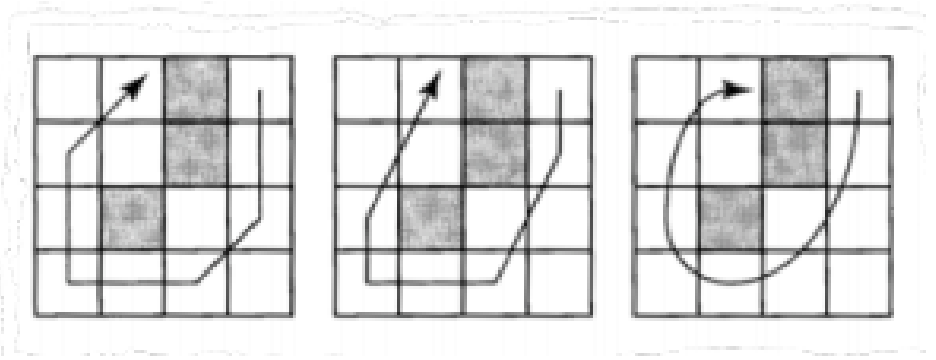
- **Navigation Meshes**
  - Set of polygons (often triangles) that describe navigable surface
  - Better paths from smaller graphs
  - Can generate from level geometry

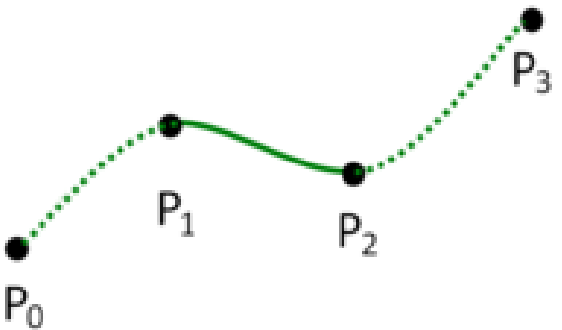


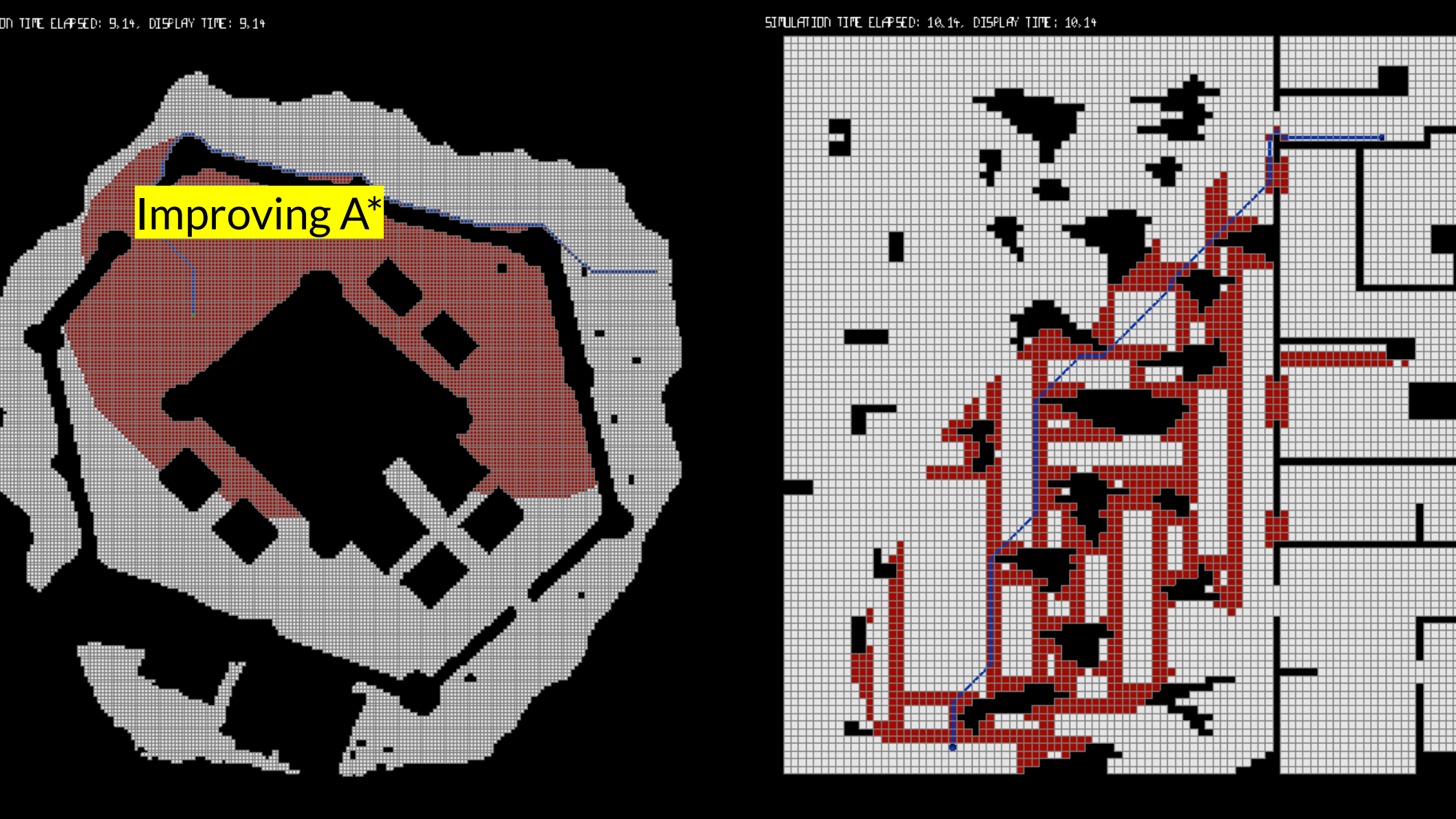
- Navigation Meshes: **Pathfinding**
  - Pathfinding returns a series of polygons
  - Polygons can be crossed using edge midpoints.
    - Adding vertex points improves cornering
  - The path remains valid as long as the same polygons are used



- **Path smoothing**
  - **String pulling** checks three consecutive points at a time: if the line from the first to the third is navigable, then the second is removed
  - A smooth path that pass through the remaining points can be computed, e.g. using Catmull-Rom splines





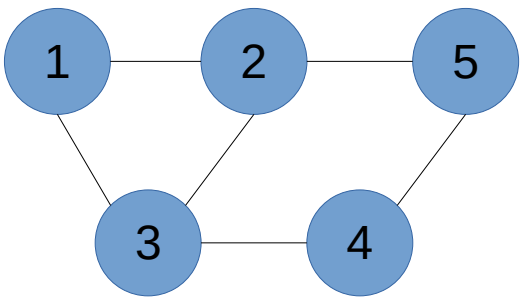

ON TIME ELAPSED: 9.14, DISPLAY TIME: 9.14
Improving A*
SIMULATION TIME ELAPSED: 10.14, DISPLAY TIME: 10.14

- **Memory Heuristics**: Store precomputed distances between key locations
  - Fast
  - Optimal
  - High memory
- **Abstraction**: reduces size of search space
  - Fast
  - Low-memory
  - Suboptimal paths
- **Search-space pruning**: Identify nodes that don’t need to be explored
  - Slow
    - Low memory

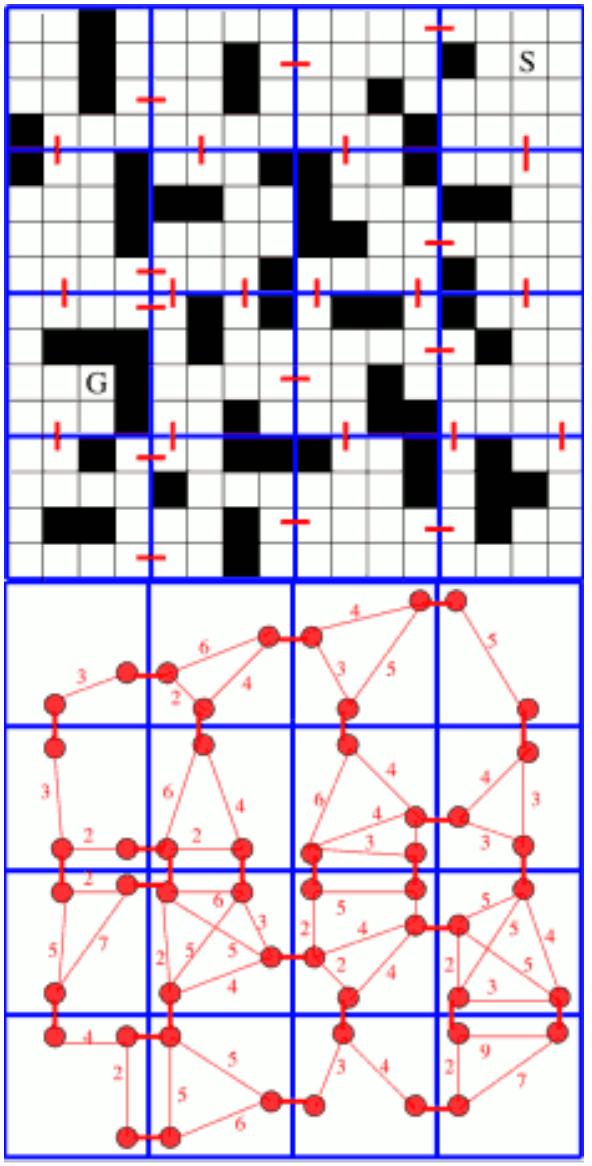
- **Precomputing paths**
  - "Next node" lookup table
    - For given current, goal, return next node/edge in path
    - Time: O(n)
    - Space: O(n^2)
  - Alteratively store as edge number as there are usually fewer edges per node than nodes

Goal node

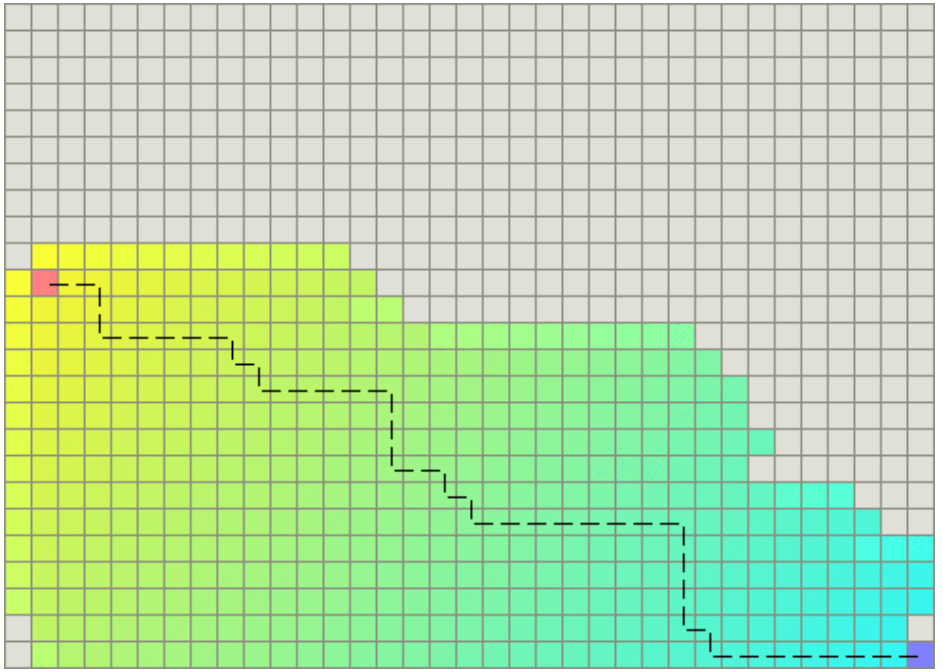
Current node	1	2	3	4	5
1	-	2	3	3	2
2	1	-	3	3	5
3	1	2	-	4	2
4	3	3	3	-	5
5	2	2	2	4	-



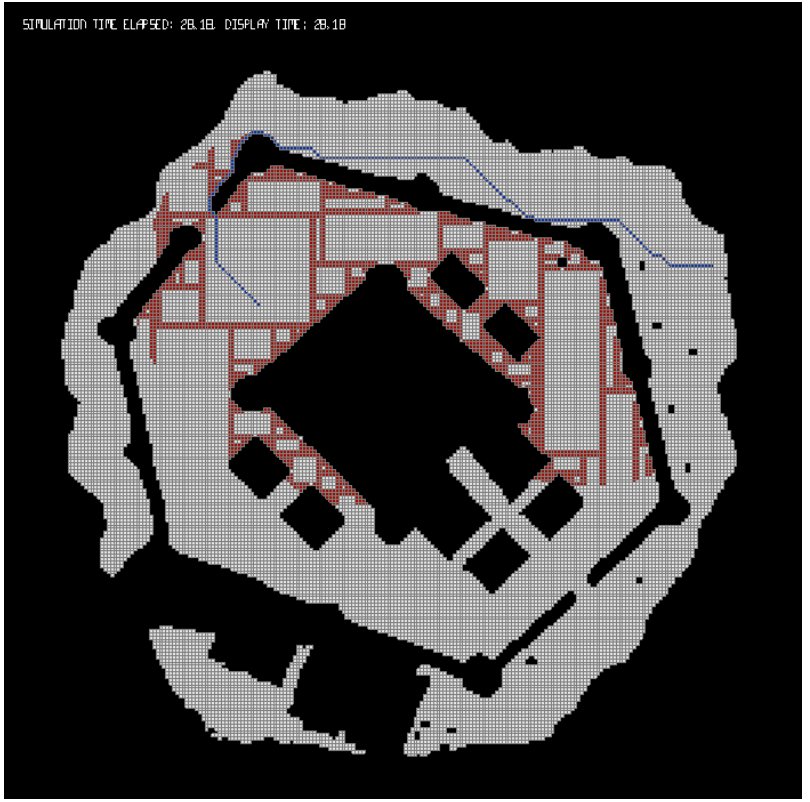
- **HPA*: Heirarchical variant of A***
  - Pre-computed abstract graph
  - Group low-level nodes into clusters
  - Each cluster is represented by a subset that borders other clusters
  - A* from start and goal to members of this cluster
  - A* over abstract graph
  - Refine into a path on original graph using A* or cached paths



- **Symmetry breaking**
  - In grids, you often get paths that only differ in the order of moves
  - Performance gain from avoiding unnecessary work



- Rectangular Symmetry Reduction (RSR)
  - Decompose map into empty rectangles
  - Only expand nodes on perimeter of rectangle



Baulder’s Gate A* + RSR

- Jump Point Search (JPS)
  - Avoid expanding neighbours that could be reached more optimally by parent
  - Make jumps across open spaces
  - More expensive per node, but fewer nodes

Baulder’s Gate A* + JPS

- **Dynamic pathfinding**, e.g. D* Lite
  - Plan in unexplored space or where path costs are changing
- **Angle-angle pathfinding**, e.g. BlockA*
  - Genertes shortest straight-line paths using line-of-sight checks
  - Path smoothing as part of the algorithm
- **Incremental pathfinding**, e.g. LPA*
  - Re-uses information generated for previous (similar) searches
- **Low memory use**, e.g. IDA*

1. Experiment with improvements to A* in your pathfinding agent
   - Breaking ties
     - https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
   - Path smoothing
   - Precomputing paths
   - Symmetry Reduction
     - https://harablog.wordpress.com/2011/08/26/fast-pathfinding-via-symmetry-breaking/
   - Implement an A* variant algorithm

- More resources
  - Amit's A* Pages
    - http://theory.stanford.edu/~amitp/GameProgramming/
  - The Moving AI Lab @ U. Denver movingai.com