Lecture 21

Pathfinding
Take Away for Today

- What are the primary goals for pathfinding?
- Identify advantages/disadvantages of A*
  - In what situations does A* fail (or look bad)?
  - What can we do to fix these problems?
- Why combine steering and A*?
  - Is this combination always appropriate?
- What do commercial games use?
Pathfinding

- You are given
  - Starting location $A$
  - Goal location $B$
- Want **valid** path $A$ to $B$
  - Avoid “impassible” terrain
  - Eschew hidden knowledge
- Want **natural** path $A$ to $B$
  - Reasonably short path
  - Avoid unnecessary turns
  - Avoid threats in the way
Abstraction: Grid & Graph

- Break world into grid
  - Roughly size of NPCs
  - Terrain is all-or-nothing
  - Majority terrain of square
  - Terrain covering “center”
- Gives us a weighted graph
  - Nodes are grid centers
  - Each node has 8 neighbors
  - Weight = distance/terrain
- Search for shortest path
  - Real distance not required
  - 14:10 ratio for diagonals
  - Allows us to use integers
## Breadth-First Search (Lab 2)

### Intuition

- **Search maintains**
  - Current node, initially `start`
  - List of nodes to visit

- **Basic Steps**
  - Have we reached the **goal**?
  - Add neighbors to **end** of list
  - Work from **first** node in list
  - Process “first-in first-out”

### Algorithm

```plaintext
n = start; L = { };
while (n not goal) {
    add n to visited;
    N(n) = unvisited neighbors
    foreach (m ∈ N(n)) {
        add m to end of L;
    }
    n = removeFirst(L);
}
return path to goal;
```
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First

A

B

X

X
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
# Pathfinding: Breadth-First

The Breadth-First Search algorithm is a graph traversal algorithm that explores all the vertices of a graph in breadth-first order. It starts at the root (selecting some arbitrary node as the root) and explores all the neighbor nodes at the current depth prior to moving on to nodes at the next depth level. 

## Algorithm

1. **Initialization:**
   - Create an empty queue `Q`.
   - Add the root node to the queue `Q`.

2. **Traversal:**
   - While `Q` is not empty, do the following:
     1. Remove the first node from `Q`.
     2. Process the node (e.g., print or store for later processing).
     3. For each neighbor of the current node that has not been processed, add it to the end of `Q`.

3. **Termination:**
   - The algorithm terminates when the `Q` is empty.

## Example

Consider a grid with obstacles represented by `X`. The algorithm starts at point A and expands in a breadth-first manner, covering all reachable points before moving to the next depth level.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>24</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>X</td>
<td>14</td>
<td>10</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>X</td>
<td>20</td>
<td>A</td>
<td>14</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>24</td>
<td>10</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>14</td>
<td>14</td>
<td></td>
</tr>
</tbody>
</table>

Pathfinding: Breadth-First

20
Pathfinding: Breadth-First

The diagram illustrates a pathfinding problem using a breadth-first search algorithm. The grid represents a maze with obstacles marked by red Xs and two points of interest: A and B. The algorithm explores all adjacent cells before moving to the next level of cells, ensuring the shortest path is found from A to B.
Pathfinding: Breadth-First
Breadth-First is Slow!

- Searches too many grids
  - Grids far away from goal
  - Works “radially outward”
- What is the problem?
  - Using graph algorithms
  - No spatial knowledge
- **Idea**: Spatial+Graph
  - Measure distance normally
  - Pick neighbor close to goal
# Heuristic Search

## Intuition
- Modified version of BFS
  - Have a list of candidates
  - Always pick *best* candidate
- Need $f$, **heuristic** function
  - Used to pick next step
  - Avoids stupid choices
- Regularly **update** $f$
  - Recompute on all neighbors
  - Reassign value if smaller

## Algorithm

```
n = start; L = {};
while (n not goal) {
  add n to visited;
  N(n) = unvisited neighbors
  foreach (m ∈ N(n)) {
    add m to L;
    update f(m);
  }
  pick n ∈ L with f least;
}
return path to goal;
```
# Heuristic Search

## Intuition

- Modified version of BFS
  - Have a list of candidates
  - Always pick the best candidate
- Need $f$, a heuristic function
  - Used to pick next step
  - Avoids stupid choices
- Regularly update $f$
  - Recompute on all neighbors
  - Reassign value if smaller

## Algorithm

```plaintext
n = start; L = {}; 
while (n not goal) {
  add n to visited;
  N(n) = unvisited neighbors of n;
  foreach (m ∈ N(n)) {
    add m to L;
    update f(m);
  }
  pick n ∈ L with f least;
} 
return path to goal;
```

## Examples:

- **Dijkstra’s Algorithm**
  - $f = \text{dist. from source}$
- **Greedy Algorithm**
  - $f = \text{estimated dist. to goal}$
A* Algorithm

- **Idea**: Dijkstra + Greedy
  - $g$: distance on **current path**
    - An “exact calculation”
    - Distance along graph
  - $h$: estimated dist. to **goal**
    - **Spatial** distance
    - Ignores all obstacles
  - Final heuristic $f = g + h$

- Many variations for $h$
  - Regular distance
  - “Manhattan Metric”

Manhattan distance = $30 + 20 = 50$
Pathfinding: A* Algorithm

The A* Algorithm is a popular algorithm for pathfinding in grid-based environments. It combines the cost of reaching a node (g) with an estimated cost to reach the goal (h) to determine the optimal path. The formula for A* is:

$$f = g + h$$

In the diagram, each cell represents a node in the grid, and the values next to the nodes represent the cost of moving to that node (g), the heuristic cost to the goal (h), and the total cost (f). The algorithm iteratively selects the node with the lowest f value to expand, based on the path cost (g) and the estimated cost to the goal (h). This process continues until the goal node is reached.

In the example shown:
- The start node (A) has a g cost of 14, an h cost of 60, and an f cost of 74.
- The goal node (B) has a g cost of 10, an h cost of 40, and an f cost of 50.

The algorithm expands nodes based on the f value, moving towards the goal with the lowest cost.
# Pathfinding: A* Algorithm

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f: 74</td>
<td>g: 14</td>
<td>h: 60</td>
</tr>
<tr>
<td>f: 60</td>
<td>g: 10</td>
<td>h: 50</td>
</tr>
<tr>
<td>f: 54</td>
<td>g: 10</td>
<td>h: 40</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f: 74</td>
<td>g: 14</td>
<td>h: 60</td>
</tr>
<tr>
<td>f: 60</td>
<td>g: 10</td>
<td>h: 50</td>
</tr>
<tr>
<td>f: 54</td>
<td>g: 14</td>
<td>h: 40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pathfinding
Pathfinding: A* Algorithm
Pathfinding: A* Algorithm

The figure illustrates the A* algorithm with two nodes, A and B, and a path highlighted in green. The algorithm calculates the cost of reaching each node using the functions f, g, and h.

- **f** represents the total cost from the start node to the current node.
- **g** represents the cost from the start node to the current node.
- **h** represents the estimated cost from the current node to the goal node.

The algorithm uses these functions to determine the order in which nodes are explored, with a preference for nodes that are closer to the goal and have lower costs.
Pathfinding: A* Algorithm

The A* algorithm is a popular pathfinding algorithm used in games and other applications. It uses a heuristic function $h$ to estimate the cost from the current node to the goal, and a cost function $g$ to estimate the cost from the start node to the current node.

The algorithm maintains a priority queue of nodes to be explored, sorted by the cost function $f = g + h$. The cost function $f$ represents the total cost of getting to the current node from the start node.

The algorithm proceeds as follows:

1. Start at the start node and add it to the priority queue.
2. While the priority queue is not empty, do:
   a. Remove the node with the lowest $f$ value from the priority queue.
   b. If the node is the goal, return the path.
   c. Otherwise, for each neighbor of the current node:
      i. Calculate $g'$, the cost to move from the current node to the neighbor through the current node.
      ii. If $g' + h$ is less than the current cost of the neighbor in the priority queue, update the neighbor's cost and add it to the priority queue.

The $g$ and $h$ values are shown in the diagram for each node.

The path found by the A* algorithm is the path with the lowest $f$ value from the start node to the goal node.

Path: A -> B
Pathfinding: A* Algorithm

A* algorithm is a search algorithm that finds the path with the lowest cumulative cost from a given start node to a given goal node in a weighted graph. The cost is calculated using the formula:

\[ f(n) = g(n) + h(n) \]

where:
- \( f(n) \) is the total cost of the path through node \( n \) to the goal.
- \( g(n) \) is the cost of the path from the start node to node \( n \).
- \( h(n) \) is the heuristic estimate of the cost from node \( n \) to the goal.

The algorithm works as follows:
1. Initialize the open set with the start node.
2. Initialize the closed set with the start node.
3. If the open set is empty, there is no solution.
4. Select the node with the lowest \( f(n) \) from the open set.
5. If the selected node is the goal node, then stop and return the path.
6. Otherwise, mark the node as visited, and add it to the closed set.
7. For each neighbor of the selected node, calculate the new \( f(n) \) and update its priority in the open set if necessary.
8. Repeat steps 4-7 until the goal node is selected or the open set is empty.
Pathfinding: A* Algorithm
Pathfinding: A* Algorithm

The A* algorithm is a popular pathfinding algorithm used in computer science, particularly in game development. It combines the advantages of Dijkstra's algorithm and greedy best-first search to find the shortest path between two points in a graph.

The cost function for A* is defined as:

\[ f(n) = g(n) + h(n) \]

where:
- \( g(n) \) is the actual cost of the path from the initial node to node \( n \).
- \( h(n) \) is a heuristic estimate of the cost from node \( n \) to the goal node.

The algorithm starts at the initial node and expands nodes in the order of their evaluation function \( f(n) \). When a goal node is found, the algorithm backtracks to find the shortest path.

In the diagram, each node represents a cell in a grid, and the numbers represent the cost values for the heuristic function \( h \), the actual cost \( g \), and the total cost \( f \). The algorithm progresses by evaluating nodes based on their \( f \) values to find the optimal path to the goal node B.
Pathfinding: A* Algorithm
Pathfinding: A* Algorithm

- \( f: 94 \)  
- \( g: 24 \)  
- \( h: 70 \)

- \( f: 74 \)  
- \( g: 24 \)  
- \( h: 30 \)

- \( f: 80 \)  
- \( g: 20 \)  
- \( h: 50 \)

- \( f: 60 \)  
- \( g: 10 \)  
- \( h: 60 \)

- \( f: 74 \)  
- \( g: 24 \)  
- \( h: 50 \)

- \( f: 74 \)  
- \( g: 14 \)  
- \( h: 40 \)

- \( f: 80 \)  
- \( g: 20 \)  
- \( h: 60 \)

- \( f: 94 \)  
- \( g: 24 \)  
- \( h: 70 \)

- \( f: 94 \)  
- \( g: 24 \)  
- \( h: 70 \)

- \( f: 80 \)  
- \( g: 20 \)  
- \( h: 60 \)

- \( f: 74 \)  
- \( g: 24 \)  
- \( h: 50 \)

- \( f: 74 \)  
- \( g: 14 \)  
- \( h: 40 \)

- \( f: 80 \)  
- \( g: 20 \)  
- \( h: 60 \)

- \( f: 94 \)  
- \( g: 24 \)  
- \( h: 70 \)

- \( f: 94 \)  
- \( g: 24 \)  
- \( h: 70 \)
Pathfinding: A* Algorithm

In case of tie, use most recently added
Pathfinding: A* Algorithm

In case of tie, use most recently added
Pathfinding: A* Algorithm

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>g: 24</td>
<td>h: 70</td>
<td>g: 14</td>
<td>h: 60</td>
<td>g: 10</td>
<td>h: 50</td>
<td>g: 14</td>
<td>h: 40</td>
<td>g: 54</td>
<td>h: 20</td>
<td>g: 58</td>
<td>h: 10</td>
<td>g: 38</td>
<td>h: 70</td>
<td>g: 38</td>
<td>h: 60</td>
<td>g: 38</td>
<td>h: 50</td>
<td>g: 48</td>
<td>h: 40</td>
<td>g: 58</td>
</tr>
</tbody>
</table>

In case of tie, use most recently added
Pathfinding: A* Algorithm

In case of tie, use most recently added
Pathfinding: A* Algorithm

A

B

f: 94
g: 24
h: 70

f: 74
g: 24
h: 60

f: 80
g: 20
h: 60

f: 80
g: 20
h: 60

f: 60
g: 10
h: 50

f: 60
g: 10
h: 50

f: 74
g: 24
h: 50

f: 74
g: 24
h: 50

f: 74
g: 24
h: 50

f: 74
g: 24
h: 50

f: 74
g: 24
h: 50

f: 74
g: 24
h: 50

f: 68
g: 54
h: 20

f: 68
g: 54
h: 20

f: 68
g: 54
h: 20

f: 68
g: 54
h: 20

f: 108
g: 38
h: 70

f: 88
g: 38
h: 50

f: 88
g: 38
h: 50

f: 88
g: 38
h: 50

f: 88
g: 38
h: 50

f: 88
g: 38
h: 50
### IndexedGraph

- Array of **IndexedNode** objs
  - Can implement as an array
  - Hard part is IndexedNode
- Each **IndexNode** must store
  - Index into the graph array
  - Array of Connection objs
- Each **Connection** must have
  - The start and end node
  - The cost to traverse edge

### IndexedAStarPathFinder

- Construct with a graph
  - Must use with **IndexedGraph**
  - Graph reference immutable
- To search for path, give
  - The start and end nodes
  - **Heuristic** implementation
  - **GraphPath** for the answer
- Can give search a **timeout**
  - Abort if it takes too long
LibGDX Support

IndexedGraph
- Array of IndexedNode objs
- Can implement as an array
- Hard part is IndexedNode
- Each IndexedNode must store
  - Index into the graph array
  - Array of Connection objs
- Each Connection must have
  - The start and end node
  - The cost to traverse edge

IndexedAStarPathFinder
- Construct with a graph
- Must use with IndexedGraph
  - with reference immutable
- Search for path, give
  - start and end nodes
  - Heuristic implementation
  - GraphPath for the answer
- Can give search a timeout
  - Abort if it takes too long

Everything in blue is an interface
**IndexedGraph**

- Array of **IndexedNode** objs
- Can implement as an array
- Hard part is...
- Each **IndexedNode**...
  - Index into the graph
  - Array of Connection objs
- Each **Connection** must have...
  - The start and end node
  - The cost to traverse edge

**IndexedAStarPathFinder**

- Construct with a graph
- Must use with an **IndexedGraph**
- Graph reference immutable
- To search for path, give...
  - Start and end nodes
  - Heuristic implementation
  - GraphPath for the answer
- Can give search a **timeout**
- Abort if it takes too long

Only these have implementations
Issues with A*: Stair Stepping
Stair Stepping

- What is the problem?
  - Move one square at a time
  - All turns are at 45°

- **Idea**: Path smoothing
  - Path is a series of waypoints
  - Straight line between points
  - Remove unnecessary points

- Can combine with A*
  - Get *degenerative* solution
  - Remove to get waypoints

- Choose first q after p where
  - Line pq is valid
  - Point q has successor s
  - Line ps is not valid
Path Smoothing
Path Smoothing
Path Smoothing

A

B

Pathfinding
Path Smoothing

A

B
Path Smoothing
Limited LibGDX support via SmoothableGraphPath interface
Turning

- **Realistic turns**
  - Smooth paths into line segments
  - Round corners for realistic movement

- **Restricted turns**
  - Limit turns to angles drawn by artist
  - 16 angles standard for 2D top-down

- See online reading for today
  - Pinter, “Toward More Realistic Pathfinding”
  - Requires free registration to Gamasutra
Multiple NPC Sizes

- Grid to largest NPC?
  - Bad for small units
  - Unnecessary blocking

- Grid to smallest NPC!
  - Multiple squares for larger
  - Center fits on grid square

- Pathfinding larger NPCs
  - A* for center-to-center
  - Size to check blocking
  - May alter the path
Multiple NPC Sizes

- Grid to largest NPC?
  - Bad for small units
  - Unnecessary blocking
- Grid to smallest NPC!
  - Multiple squares for larger
  - Center fits on grid square
- Pathfinding larger NPCs
  - A* for center-to-center
  - Size to check blocking
  - May alter the path
Multiple NPC Sizes

- Grid to largest NPC?
  - Bad for small units
  - Unnecessary blocking

- Grid to smallest NPC!
  - Multiple squares for larger
  - Center fits on grid square

- Pathfinding larger NPCs
  - A* for center-to-center
  - Size to check blocking
  - May alter the path
Multiple NPC Sizes

- Grid to largest NPC?
  - Bad for small units
  - Unnecessary blocking
- Grid to smallest NPC!
  - Multiple squares for larger
  - Center fits on grid square
- Pathfinding larger NPCs
  - A* for center-to-center
  - Size to check blocking
  - May alter the path
Multiple NPC Sizes

- Grid to largest NPC?
  - Bad for small units
  - Unnecessary blocking

- Grid to smallest NPC!
  - Multiple squares for larger
  - Center fits on grid square

- Pathfinding larger NPCs
  - A* for center-to-center
  - Size to check blocking
  - May alter the path
Fitting NPCs on a Grid

- Assume NPC is square
  - Represents “reach”
  - Simplifies turning
- Requires “odd” sizes
  - Center must be a grid
  - Radius in full grid squares
  - What about even sizes?
- “Tabletop” solution
  - Round down when moving
  - Round up when in place
Waypoints

Express paths as a sequence of segments
Steering

- Alternative to pathfinding
  - Uses forces to move NPCs
  - Great for small paths
- Examples
  - Artificial potential fields
  - Vortex fields
  - Custom steering behaviors
- See Craig Reynold’s page
  - See “Physics & Motion”
  - com.badlogic.gdx.ai.steer
Steering and Pathfinding

- Use waypoint as “goal”
  - Attract NPC to waypoint
  - When close, next waypoint
- Great for multiple NPCs
  - Pathfind for largest NPC
  - Steering to move along path
  - Repulsion keeps NPCs apart
- **Drawbacks:**
  - Military formations are hard
  - Get stuck at bottlenecks
Dynamic Obstructions

• Others can get in way
  • Enemies guarding locale
  • Friends waiting in queue

• Correct response?
  • Compute a new path?
  • Wait to be unblocked?

• What would you do?
  • See what is blocking
  • Making an educated guess
  • Character AI solution
Pathfinding in Practice

- **Navigation Meshes**
  - Indicates walkable areas
  - 2D geometric representation
  - Connected convex shapes
  - A* graph: center-to-center

- **Making Nav Meshes**
  - Often done by level editor
  - Can be modified by hand
  - Annotate special movement
  - **Example**: jump points
Easy Pathfinding on Meshes

Center of each Region

Corners of the Mesh

Pathfinding
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding

Cost depends on how entered
Alternative: Hierarchical Pathfinding

Design hierarchy to minimize cost artifacts

Cost depends on how entered
LibGDX Support

HierarchicalGraph

- Graph with multiple levels
  - Has a current active level
  - Graph API matches level
  - Can switch this level on fly
- Also can convert levels
  - node + level => node
  - Rules to group nodes
  - Rules to split nodes

HierarchicalPathFinder

- Specify a pathfinder to use
  - Could be A* or otherwise
  - Will use it on each level
- The implementation
  - Finds path at highest level
  - Expands nodes to next level
  - Refines path to expansion
  - Repeats until level 0
HierarchicalGraph

- Graph with multiple levels
- Has a current active level
- Graph API matches level
- Can switch level on fly
- Also can convert levels
  - node + level => node
- Rules to group nodes
- Rules to split nodes

HierarchicalPathFinder

- Specify a pathfinder to use
- Could be A* or otherwise
- Will use it on each level
- The Interface
  - Finds path at highest level
  - Expands nodes to next level
  - Refines path to expansion
  - Repeats until level 0

LibGDX Support
Summary

• **A* algorithm** is primary pathfinding tool
  • Make world into a grid/navigation mesh
  • Search for a path on associated graph
  • Adjust heuristics for terrain, threats

• But there are a lot of “special tricks”
  • Tricks to make movement realistic
  • Tricks to handle coordinated movement
  • Talk to Instructor (or TAs) if need more tricks