Lecture 22

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**

```
n = start; L = {};
while (n not goal) {
    add n to visited;
    N(n) = unvisited neighbors
    foreach (m \in 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

The diagram illustrates a breadth-first pathfinding algorithm. The grid represents the environment, with obstacles marked by 'X'. The algorithm explores all adjacent cells in order of their distance from the starting point 'A'. The path is highlighted in green, showing the sequence of moves from 'A' to 'B' in the optimal way.
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First

The diagram illustrates the pathfinding algorithm, specifically Breadth-First Search (BFS), in a grid environment. The algorithm starts from node A and explores all adjacent nodes before moving to the next level of nodes. The path is highlighted in pink, showing the order in which nodes are visited. The goal is to reach node B, which is marked with a red star. The algorithm continues until all reachable nodes are explored.
Pathfinding: Breadth-First
Pathfinding: Breadth-First

```
28  24  20  24
24  14  10  14
20  10  10  14
24  14  10  14
24  14  10  14
28  24  20  24
34  30  34  34
```

A

B

X

X

X

X

X

X

X

X

X

X

X

X

X

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

28 24 20 24
24 14 10 14
20 10 24 24
24 14 10 14
28 24 20 24
38 34 30 34
34 38 44 48
Pathfinding: Breadth-First
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

\[ n = \text{start}; \ L = \{ \}; \]
\[ \text{while } (n \text{ not goal}) \{ \]
\[ \quad \text{add } n \text{ to visited}; \]
\[ \quad N(n) = \text{unvisited neighbors} \]
\[ \quad \text{foreach } (m \in N(n)) \{ \]
\[ \quad \quad \text{add } m \text{ to } L; \]
\[ \quad \quad \text{update } f(m); \]
\[ \quad \} \]
\[ \quad \text{pick } n \in L \text{ with } f \text{ least}; \]
\[ \} \]
\[ \text{return } \text{path to goal}; \]
Heuristic Search

Intuition

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

Examples:

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

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;
```
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”

![Diagram showing A* Algorithm with points A and B, and Manhattan distance calculation of 30 + 20 = 50]
Pathfinding: A* Algorithm
Pathfinding: A* Algorithm

A

B
Pathfinding: A* Algorithm

The A* Algorithm is a popular pathfinding algorithm used in game design and other applications. It combines the best features of Dijkstra's algorithm and the heuristic search to find the optimal path from a start point to a goal point.

At Cornell University, the game design initiative focuses on teaching students how to apply pathfinding algorithms like A* to create realistic and efficient paths for characters or objects in video games.

The algorithm works by calculating the total path cost (f) for each node, which is the sum of the cost to reach the node (g) and the estimated cost to reach the goal from the node (h). The goal is to find the path that minimizes f.

In the diagram, each cell represents a node with its g, h, and f values. The algorithm explores the network of nodes, using the g and h values to calculate f and choose the best path to the goal (B).
Pathfinding: A* Algorithm

<table>
<thead>
<tr>
<th>Node</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>74</td>
<td>14</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>54</td>
<td>14</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>10</td>
<td>30</td>
</tr>
</tbody>
</table>

The A* algorithm calculates the total cost of reaching a node as follows: f = g + h. The algorithm explores the graph by expanding the node with the lowest f value until the goal is reached.
Pathfinding: A* Algorithm

The Pathfinding algorithm is a search algorithm that is widely used in game design to find the shortest path between two points. It uses a heuristic function to estimate the cost to reach the goal, and it expands the path with the lowest cumulative cost. The algorithm is particularly effective in finding paths in complex environments.

The diagram above illustrates the A* algorithm in action. The algorithm starts at point A and expands outward, calculating the total cost of each possible path. The total cost is calculated as the sum of the movement cost (g) and the heuristic cost (h). The algorithm continues expanding paths until it reaches point B, the goal. The path with the lowest total cost is then selected as the optimal path.
Pathfinding: A* Algorithm

The A* Algorithm is a pathfinding algorithm that finds the path with the lowest cumulative cost from a start node to a goal node. It combines the cost of moving from the start node to the current node (g) and the estimated cost of moving from the current node to the goal node (h) to determine the total cost (f) of each node.

In the example shown, the algorithm is used to find the path from node A to node B. The values for f, g, and h are calculated for each node, with f = g + h. The algorithm expands the node with the lowest f value until it reaches the goal node.
Pathfinding: A* Algorithm

- A
- B

The algorithm calculates the cost of reaching each point from A to B using the formula f = g + h, where f is the total cost, g is the cost from A to the point, and h is the heuristic cost from the point to B. The path with the lowest f value is chosen.
Pathfinding: A* Algorithm

The A* Algorithm

A

B

Pathfinding
Pathfinding: A* Algorithm

A B

A: f: 94, g: 24, h: 70
B: f: 74, g: 14, h: 40

Pathfinding
Pathfinding: A* Algorithm

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f: 94</td>
<td>g: 24</td>
<td>h: 70</td>
<td></td>
</tr>
<tr>
<td>f: 74</td>
<td>g: 24</td>
<td>h: 60</td>
<td></td>
</tr>
<tr>
<td>f: 74</td>
<td>g: 14</td>
<td>h: 60</td>
<td></td>
</tr>
<tr>
<td>f: 60</td>
<td>g: 10</td>
<td>h: 50</td>
<td></td>
</tr>
<tr>
<td>f: 60</td>
<td>g: 10</td>
<td>h: 50</td>
<td></td>
</tr>
<tr>
<td>f: 54</td>
<td>g: 14</td>
<td>h: 40</td>
<td></td>
</tr>
<tr>
<td>f: 80</td>
<td>g: 20</td>
<td>h: 60</td>
<td></td>
</tr>
<tr>
<td>f: 94</td>
<td>g: 24</td>
<td>h: 70</td>
<td></td>
</tr>
<tr>
<td>f: 94</td>
<td>g: 24</td>
<td>h: 70</td>
<td></td>
</tr>
<tr>
<td>f: 80</td>
<td>g: 20</td>
<td>h: 60</td>
<td></td>
</tr>
<tr>
<td>f: 74</td>
<td>g: 24</td>
<td>h: 50</td>
<td></td>
</tr>
</tbody>
</table>

A

B
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

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

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

The A* Algorithm is a popular pathfinding algorithm that efficiently finds the shortest path between two points. It combines the cost of moving from the start node to the current node (g) with an admissible heuristic estimate of the cost to reach the goal (h) to determine the f-score for each node. The f-score guides the search towards the goal, ensuring that the path found is the optimal one.

The diagram above illustrates the A* Algorithm with a grid and nodes labeled A and B. The f, g, and h values for each node are shown, representing the total cost (f), the cost to reach the node from the start (g), and the estimated cost to reach the goal (h) respectively. The algorithm explores the nodes with the lowest f-scores first, pruning branches that do not lead to potential improvements in the path cost.
# 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`
- 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
### IndexedGraph
- Array of IndexedNode objs
  - Can implement as an array
  - Hard part is IndexedNode
- Each IndexedNode must have
  - 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
- 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

---

**Everything in blue is an interface**

---

LibGDX Support
LibGDX Support

**IndexedGraph**
- Array of **IndexedNode** objs
  - Can implement as an array
  - Hard part is...
- Each **IndexedNode**
  - Index into the...
- 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

---

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

A

B

Pathfinding
Path Smoothing

Pathfinding
Path Smoothing
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](http://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
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding
Alternative: Hierarchical Pathfinding

Cost depends on how entered
Alternative: Hierarchical Pathfinding

Cost depends on how entered

Design hierarchy to minimize cost artifacts
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
- There are interfaces
  - Finds path at highest level
  - Expands nodes to next level
  - Refines path to expansion
  - Repeats until level 0
Influence Maps: Pathfinding and AI

Slide courtesy of Dave Mark
Influence Maps: Pathfinding and AI

Slide courtesy of Dave Mark
Influence Maps: Pathfinding and AI

Slide courtesy of Dave Mark
Implementing Influence Maps

- Use the pathfinding grid
  - Track movement in square
  - Track if friend or foe

- Keep count as a queue
  - Count is sum of queue
  - Allows us to “time out”
  - Otherwise, marked forever

- Use queue as a predictor
  - Look at rate of change
  - Also valuable for AI

Sensing is at grid, not NPC
Advantages of Influence Maps

Slide courtesy of Dave Mark
Advantages of Influence Maps

Slide courtesy of Dave Mark
Advantages of Influence Maps

Slide courtesy of Dave Mark
Advantages of Influence Maps

Influence data reflects changes

Slide courtesy of Dave Mark
Advantages of Influence Maps

Slide courtesy of Dave Mark
Advantages of Influence Maps

Add *a priori* assumptions

Slide courtesy of Dave Mark
Advantages of Influence Maps

Add *a priori* assumptions

Slide courtesy of Dave Mark
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