Lecture 21

Pathfinding
Take Away for this Lecture

- 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
```java
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
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
Pathfinding: Breadth-First
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

\[
n = \text{start}; \quad L = \{ \};
\]

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

\[
\text{return path to goal;}
\]
### Intuition
- Modified version of BFS
  - Have a list of candidates
  - Always pick best candidate
- Need $f$, heuristic
  - 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;
```

### 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 computer science. It is a best-first search algorithm that uses a heuristic function to guide its search, allowing it to find the shortest path from a start node to a goal node.

In the diagram, each node represents a cell on the grid, and the values f, g, and h represent the cost from the starting node to the current node, the cost from the current node to the goal node, and the heuristic cost from the current node to the goal node, respectively.

The A* algorithm uses these values to determine the next node to explore, always choosing the node with the lowest f value. The path from the start node to the goal node is then determined by backtracking from the goal node to the start node, following the path with the lowest f values.
Pathfinding: A* Algorithm

The A* Algorithm is a* a* algorithm that combines the best features of Dijkstra's algorithm and the greedy best-first search algorithm. It is a popular algorithm for pathfinding and will be used as an example.

A* algorithm uses a heuristic function h to estimate the cost to reach the goal from a given node. The estimate is added to the actual cost g to determine the f value.

In the diagram, we see the cost of moving from one node to another (g), the estimated cost to reach the goal (h), and the total cost (f).

The algorithm explores the nodes in order of increasing f value, selecting the node with the lowest f value at each step. It expands the node with the lowest f value and repeats the process until the goal node is found or all possible nodes have been explored.

In this example, the algorithm starts at node A and explores the nodes with the lowest f values until it reaches node B, which is the goal node.
Pathfinding: A* Algorithm

f: 74
\(g = 24\)
\(h = 60\)

f: 60
\(g = 24\)
\(f = 74\)

f: 54
\(g = 20\)
\(f = 60\)

X

f: 74
\(g = 24\)
\(f = 74\)

X

f: 60
\(g = 24\)
\(f = 74\)

\(g = 20\)
\(f = 60\)

X

\(g = 20\)
\(f = 60\)

\(g = 14\)
\(f = 74\)

\(g = 14\)
\(f = 60\)

X

X

X

X

X

B
Pathfinding: A* Algorithm

A

B

Pathfinding
Pathfinding: A* Algorithm

The A* algorithm is a popular pathfinding algorithm that combines the best features of Dijkstra’s algorithm and greedy best-first search. It uses a heuristic function $h$ to guide its search, along with the actual path cost $g$. The cost function $f$ for each node is defined as:

$$f = g + h$$

The algorithm starts at the initial node, expands nodes in a prioritized manner, and stops when the goal is found or the search space is exhausted. The image above illustrates the A* algorithm's operation with nodes A and B as the start and goal, respectively, and a heuristic function that weights specific paths differently.
Pathfinding: A* Algorithm

A* Algorithm:

\[ f = g + h \]

- **f** is the total cost from the start node A to any given node.
- **g** is the cost from the start node to the current node.
- **h** is the estimated cost from the current node to the goal node B.

The algorithm uses a priority queue to select the next node to expand based on the lowest **f** value.

The path to the goal node B is determined by backtracking from the goal node to the start node A, following the edges that led to the goal node.
Pathfinding: A* Algorithm

A* Algorithm

A

B

f: 88  
g:28  
h:60

f: 74  
g:24  
h:40

f: 74  
g:14  
h:60

f: 60  
g:10  
h:50

f: 60  
g:10  
h:30

f: 74  
g:14  
h:40

f: 54  
g:14  
h:40

f: 54  
g:10  
h:40

f: 74  
g:10  
h:60

f: 88  
g:28  
h:60

f: 74  
g:24  
h:50
Pathfinding: A* Algorithm

- **A**: Start point
- **B**: End point

- **f**: Total cost (g + h)
- **g**: Cost from start
- **h**: Heuristic cost to target
Pathfinding: A* Algorithm

<table>
<thead>
<tr>
<th>A</th>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Node A" /></td>
<td><img src="image" alt="Node B" /></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Path" /></td>
<td><img src="image" alt="Path" /></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Path" /></td>
<td><img src="image" alt="Path" /></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Path" /></td>
<td><img src="image" alt="Path" /></td>
<td></td>
</tr>
</tbody>
</table>

### A* Algorithm

- **f**: Sum of cost from start to current node plus estimated cost to goal.
- **g**: Cost from start to current node.
- **h**: Estimated cost from current node to goal.

<table>
<thead>
<tr>
<th>Node</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>24</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>B</td>
<td>28</td>
<td>20</td>
<td>60</td>
</tr>
</tbody>
</table>
Pathfinding: A* Algorithm

A* Algorithm

Pathfinding

37
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

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th></th>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: 94</td>
<td></td>
<td>f: 80</td>
<td></td>
<td>f: 74</td>
<td></td>
</tr>
<tr>
<td>g: 24</td>
<td>h: 70</td>
<td>g: 20</td>
<td>h: 60</td>
<td>g: 24</td>
<td>h: 60</td>
</tr>
<tr>
<td>h: 70</td>
<td>g: 14</td>
<td>h: 60</td>
<td>g: 10</td>
<td>h: 50</td>
<td>g: 14</td>
</tr>
<tr>
<td>h: 60</td>
<td>g: 10</td>
<td>h: 50</td>
<td>g: 10</td>
<td>h: 30</td>
<td></td>
</tr>
<tr>
<td>h: 50</td>
<td>g: 14</td>
<td>h: 40</td>
<td>f: 74</td>
<td>f: 68</td>
<td>f: 80</td>
</tr>
<tr>
<td>g: 24</td>
<td>h: 70</td>
<td>g: 20</td>
<td>h: 60</td>
<td>g: 24</td>
<td>h: 50</td>
</tr>
<tr>
<td>h: 60</td>
<td>g: 38</td>
<td>h: 60</td>
<td>g: 38</td>
<td>h: 50</td>
<td>g: 38</td>
</tr>
<tr>
<td>h: 70</td>
<td>g: 38</td>
<td>h: 60</td>
<td>g: 38</td>
<td>h: 50</td>
<td>g: 48</td>
</tr>
</tbody>
</table>

The table represents the pathfinding algorithm using A* algorithm. The table includes the costs of moving from one node to another, where `f` is the total cost, `g` is the cost to reach the node, and `h` is the heuristic cost. The algorithm chooses the path with the lowest `f` value to reach the goal efficiently.
### 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
## LibGDX Support

### IndexedGraph

- Array of IndexedNode objs
  - Can implement as an array
  - Hard part is IndexedNode
- Each IndexedNode must have
  - 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 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

---

Everything in **blue** is an interface
IndexedGraph

- Array of IndexedNode objs
  - Can implement as an array
  - Hard part is...

- Each IndexedNode
  - Index into...
  - 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

- 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
Path Smoothing
Path Smoothing

A

B
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”
  - Techniques from the sprite days of RTSs
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
Multiple NPC Sizes
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
Why Obstructions Matter

North Loop

South Loop
Steering Interfaces in LibGDX

Steerable

- Access to **physics data**
  - `getLinearVelocity()`
  - `getAngularVelocity()`
  - `getBoundingRadius()`

- Also has **limiter** info
  - `get/setMaxLinearSpeed()`
  - `get/setMaxAngularSpeed()`
  - `get/setMaxLinearAccel()`
  - `get/setMaxAngularAccel()`

SteeringBehavior

- Has a Steerable **owner**
  - Object being steered

- Other potential attributes
  - **Target** (goal location)
  - **Path** (path following)

- Calcs **SteeringAcceleration**
  - Physics **recommendation**

- DOES NOT set physics
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
Optimization: Hierarchical Pathfinding
Optimization: Hierarchical Pathfinding
Optimization: Hierarchical Pathfinding
Optimization: Hierarchical Pathfinding

Cost depends on how entered
Optimization: Hierarchical Pathfinding

Design hierarchy to minimize cost artifacts

Cost depends on how entered
<table>
<thead>
<tr>
<th><strong>HierarchicalGraph</strong></th>
<th><strong>HierarchicalPathFinder</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Graph with multiple levels</td>
<td>- Specify a pathfinder to use</td>
</tr>
<tr>
<td></td>
<td>- Has a current active level</td>
</tr>
<tr>
<td></td>
<td>- Graph API matches level</td>
</tr>
<tr>
<td></td>
<td>- Can switch this level on fly</td>
</tr>
<tr>
<td></td>
<td>- Also can convert levels</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LibGDX Support
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
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