Pathfinding

Artificial Intelligence for gaming
Pathfinding

Execution Management

Group AI
- Strategy

Character AI
- Decision Making
- Movement
- Pathfinding

World Interface

Animation

Physics
Pathfinding Graphs

- Pathfinding does not work directly on Geometry
- Simplification:
  - Abstraction of movement possibilities in a graph
Pathfinding Graphs

- **Nodes**: Important places
  - Sometimes just rooms
  - Sometimes different places in a single room

- **Edges**: Connections through which we can travel

- **Weights**: Costs of traveling through a certain connection
  - General assumption for developing algorithms:
    - Weights are positive numbers

Preview:
- For tactical planning, need to
Pathfinding Graphs

- If travelling costs depend on the direction:
  - Use a directed graph
  - All algorithms support directed graphs
Pathfinding Graphs

- Representation of (directed, weighted) graphs
  - Adjacency List
  - Adjacency Matrix
  - List of edges

```
a: [(c,3), (d,4)]
b: [(a,2), (c,2)]
c: [(e,5)]
d: [(b,3), (c,4)]
e: [(a,2), (d,2)]
```

```
\begin{pmatrix}
0 & 0 & 3 & 4 & 0 \\
2 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 \\
0 & 3 & 4 & 0 & 0 \\
2 & 0 & 0 & 2 & 0 \\
\end{pmatrix}
```
Pathfinding Graphs

- Dijkstra Algorithm
  - Input: Weighted, directed graph, starting and ending vertex
  - Output: A minimum path between starting and ending vertex
    - Definition: Costs of a path is the sum of the weights
Pathfinding Graphs

- **Disjkstra Algorithm**
  - **Informal Description:**
    - Processes nodes one-by-one, starting with the starting node
    - Whenever a node is visited, puts neighboring node not yet visited into a list of “seen” nodes
    - Maintains a list of costs to reach each node.
    - Whenever a node is visited:
      - Update all costs when visiting through the node
Pathfinding Graphs

Example:

- Seen = [b]
- Costs in b = 0

Costs = 0
Pathfinding Graphs

Example:

- Go to all the edges leaving b
- Mark the target nodes as seen
- Give the target nodes costs equal to the weight of the edge
- Mark node b as done
Pathfinding Graphs

Example:
- Pick the node with lowest costs in the seen set
  - We break tie by picking a
- Repeat what we did for b
  - Go to all edges leaving a
  - The edge to d put d into the seen list and gives it costs 2+4
Pathfinding Graphs

Example:

- Processing a
- The edge to c goes to a node already seen
  - In this case, we need to compare the costs through a (costs of a + weight of edge) to the costs previously obtained (in this case 2).
  - We give it the minimum costs $2 = \min(5,2)$
Pathfinding Graphs

Example:
Now we can mark a as done
Pathfinding Graphs

Example:
- The seen list has two elements, c and d
- c has minimum costs, so we use it
Pathfinding Graphs

- **Example:**
  - Processing c
  - There is an edge to e, which places e in the seen list with costs 2+5
Pathfinding Graphs

- Example:
  - We select d in the seen list
Pathfinding Graphs

Example:
- We select d in the seen list
Pathfinding Graphs

Example:
- We have an edge to a
- However, a is already done
- Because of the way we select from the seen list, we know that we cannot beat the costs to get to a by going through e
  - Because we know that the costs to go to a is lower than the costs to go to e
Pathfinding Graphs

Example:
- We process the edge to d
- The alternative costs to d is 9, so we do not change the costs in d
- We now can mark e as done
Pathfinding Graphs

Example:
- There is only $d$ left in the seen list
- All other nodes are done
- Can stop
Pathfinding Graphs

**Example:**
- We know the costs of going to $d$
- But we do not know how to get there.

**Solution:**
- Decorate each node with the predecessor when updating the costs.
Pathfinding Graphs

Algorithm for Dijkstra

1. Seen = [Start]
2. While Seen:
3.  current = minimum([seen])
4.  seen.remove(current); done.add(current)
5.  for edge in current.edgesOut:
6.     newVertex = edge.getFinish()
7.     if newVertex not in seen and not in processed:
8.        seen.append(newVertex)
9.        newVertex.costs = current.costs + edge.weight
10.       newVertex.predecessor = current
11. elif newVertex in seen:
12.     altCosts = current.costs + edge.weight
13.     if altCosts < newVertex.costs:
14.        newVertex.costs = altCosts
15.        newVertex.predecessor = current
Pathfinding Graphs

- **Problems with Dijkstra**
  - Dijskstra looks at every edge
  - Can stop Dijkstra when the goal edge has a costs smaller than any of the nodes in the seen category, i.e. when we process the goal
  - But this does not mean that we do not have to look at lots of nodes
Pathfinding Graphs

- Example:
Pathfinding Graphs

Example:

Start

Goal
Pathfinding Graphs

Example:
Pathfinding Graphs

Example:

Start

Goal
Pathfinding Graphs

Example:
Pathfinding Graphs

- It might take processing lots of "obviously" uninteresting nodes before we can process the goal node.
- Dijkstra works well to find minimum paths to all nodes, but not so well for finding a minimum path to a specific node.
Implementing Dijkstra

- **Graph data structure:**
  - List of vertices
  - For each vertex, a list of weighted edges
    - Each edge is defined by a weight and a destination

- **Lists data structure:**
  - Need to find the minimum cost node in the seen list
    - Use priority queue
      - In Python: module queue has PriorityQueue

- **Defining edges:**
  - To use PriorityQueue, need to define a class Vertex, with sorting methods __le__, __ge__
Implementing Dijkstra

Algorithm

- Initialize lists:
  - Seen contains starting node
  - Done is empty

while not seen.empty()
    current = seen.get()
    if current = goal:
        # create path from information
    for edge in current.edges():
        vertex = edge.getDestination()
        if vertex not in done:
            # update vertex,
Pathfinding Graphs

- A*
  - Uses bounds in order to eliminate uninteresting branches
  - Is very interesting for any search problem that can be described with a graph
  - Is a very generic algorithm used in gaming also for strategic planning by avatars.
Pathfinding Graphs

- **A* Idea**
  - Use a heuristic to *estimate* the costs of a node to the goal
  - Similar to Dijkstra
    - Use value of node to select from seen list
Pathfinding Graphs

- In path-finding, the heuristics can be related to bird-flight distance
Pathfinding Graphs

- A*: Stopping Rule
  - In Dijkstra:
    - Stop when the minimum cost in the Seen-list is larger than the current costs to the goal
  - In A*:
    - Goal node has smallest estimated costs in the Seen-list
    - To be sure, need established smallest costs
Pathfinding Graphs

- Algorithm:
  - Tree algorithm:
    - Nodes characterized by location (original graph node) and costs to get there.
    - Additionally evaluated by costs estimate
Romania with step costs in km
A* search example
A* search example
A* search example
A* search example
A* search example
A* search example
Admissible heuristics

- A heuristic $h(n)$ is **admissible** if for every node $n$, $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$.

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is **optimistic**.

- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

- **Theorem**: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal.
Optimality of A* (proof)

- Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $g(G_2) > g(G)$ since $G_2$ is suboptimal
- $f(G) = g(G)$ since $h(G) = 0$
- $f(G_2) > f(G)$ from above
Optimality of $A^*$ (proof)

Suppose some suboptimal goal $G_2$ has been generated and is in the fringe. Let $n$ be an unexpanded node in the fringe such that $n$ is on a shortest path to an optimal goal $G$.

From above:
- $f(G_2) > f(G)$
- $h(n) \leq h^*(n)$ since $h$ is admissible
- $g(n) + h(n) \leq g(n) + h^*(n)$
- $f(n) \leq f(G)$

Hence $f(G_2) > f(n)$, and $A^*$ will never select $G_2$ for expansion.
Consistent heuristics

- A heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,

  $$ h(n) \leq c(n,a,n') + h(n') $$

- If $h$ is consistent, we have

  $$ f(n') = g(n') + h(n') $$

  $$ = g(n) + c(n,a,n') + h(n') $$

  $$ \geq g(n) + h(n) $$

  $$ = f(n) $$

  - i.e., $f(n)$ is non-decreasing along any path.

- **Theorem:** If $h(n)$ is consistent, A* using `GRAPH-SEARCH` is optimal.
Optimality of A* 

- A* expands nodes in order of increasing $f$ value
  - Gradually adds "$f$-contours" of nodes
  - Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Pathfinding Graphs

- **Heuristic selection**
  - Heuristic always too low
    - A* takes longer to run
  - Heuristic sometimes overestimates
    - A* can produce wrong result
    - Might still be OK for gaming
Pathfinding Graphs

- **Heuristics selection**
  - **Euclidean distance**
    - Admissible heuristic (always underestimates)
      - In an indoor environment, can lead to long run-times
  - **Cluster heuristic**
    - Groups nodes in clusters
    - Can be automatic, but is often provided by level design
    - Look-up table gives smallest distance between members of two different clusters
    - **Heuristic:**
      - If start and end node are in the same cluster: Euclidean distance
      - Otherwise, use minimum distance between points in both clusters
  - **Trade-off for choosing cluster size**
    - Clusters small → Large lookup table
    - Clusters big → Inaccurate
World Representation

- Needed to translate from level design to graph representation
  - Generation:
    - Manual
    - Automatic
  - Validity:
    - If graph tells avatar to move from region A to region B then it should be possible to reach any point in B from any point in A
    - Validity is not always enforced
World Representation

**Poor quantization**
But wall avoidance results in useful path

**Bad quantization**
Path ends up in dead end
World Representation

- **Tiling**
  - Tile-based graphs are generated automatically
  - Validity problems can arise if tiles are only partially blocked

Example of an invalid blockage
World Representation

- Dirichlet Domains
  - aka Voronoi polygons
- Tiles the plane into regions defined by “characteristic points”
- Regions made up of the points nearest to a characteristic point
- In general: do not give valid tiling
- Are popular because they can be automatically generated
World Representation

Points of visibility

- Observation: Optimal path has inflection points at convex vertices in the environment
- Generate points at convex vertices
  - If avatars have girth, move away from vertex
World Representation
World Representation

- **Points of Visibility Graph**
  - Edges between points
    - If one can be seen from the other
    - Cast rays
  - Can be taken to represent the centers of Dirichlet domain
  - Can generate too many points
World Representation

Vertices in a bloated visibility graph
World Representation

- Division Schemes
  - Games can have floor polygons (designed by level artist) as regions
  - Each polygon becomes a graph
  - Difficult for artists to maintain validity
  - Very popular
    - Pathengine middleware
World Representation

- Additional information
  - Graph vertex can represent more than just a position
  - Example: Ship
    - Cannot turn sharply
    - Vertex represents position and orientation
Cost functions
- Usually, cost function (edge weight) represents distance
- Can represent costs of moving
  - Moving through swamp takes more time, …
Path Smoothing

- Paths generated by path-finding can be erratic
- Path smoothing gives more believable paths
Path Smoothing

- **Algorithm**
  - Starting with the third node, cast a ray towards the beginning node, the second node, …
  - If ray goes not through, add the node to the smoothed path node list
  - Generates a reasonable smooth path, but not all possible ones
Hierarchical Pathfinding

Idea:

- Clustering:
  - Group nodes into clusters
  - Clusters for nodes of a higher level graph
  - Continue

- Pathfinding:
  - Find path on highest level
  - For each super-node, find path inside super-node
  - continue to lowest level
Hierarchical Pathfinding
Hierarchical Pathfinding
Hierarchical Pathfinding
Hierarchical Pathfinding
Hierarchical Pathfinding
Hierarchical Pathfinding

- Advantages
  - Pathfinding in small graphs: Quick
- Disadvantages
  - Distance between clusters are hard to measure
Hierarchical Pathfinding

- Distance between clusters:
  - Depends on from where you enter the cluster
  - Using this information destroys advantages of hierarchical pathfinding

- Heuristics
  - Minimum distance
  - Maximin distance
  - Average minimum distance
Hierarchical Pathfinding

- **Minimum distance**
  \[ \text{Distance}(A,B) = \min\{|a-b|, \, a \in A, \, b \in B\} \]

- **Maximin distance**
  - For each incoming link into \( B \) and for each outgoing link from \( B \):
    - Calculate distance within \( B \) from incoming to outgoing link
    - Add the maximum of these distance to the cost of the incoming link
Hierarchical Pathfinding

- Maximin distance
Hierarchical Pathfinding

- Maximin distance: Inlink A → B
Hierarchical Pathfinding

- Maximin distance: Inlink A → B
  - Find anchor points for incoming and outgoing links
Hierarchical Pathfinding

- Maximin distance: Inlink $A \rightarrow B$
  - Find anchor points for incoming and outgoing links
  - Calculate minimum distance between anchor points for $A$
Hierarchical Pathfinding

- Maximin distance:
  - Add maximum of these distances to the costs of the inlink
Hierarchical Pathfinding

- **Average Minimum Distance**
  - Add the average distance between anchor points to the costs of the inlink

- **Minimum distance:**
  - connections within the cluster are free

- **Maximin distance**
  - connections within the cluster are taken to be the maximum possible

- **Average Minimum Distance**
  - Is a compromise
Open Goal Pathfinding

- Goal nodes are not necessarily unique
  - e.g.: Avatar needs to find ammunition
    - Ammunition dumps at various locations
    - Pathfinding needs to give path to the nearest point with ammunition
  - A* heuristic is problematic
    - Assume nearest goal point is blocked
    - A* uses the distance to nearest goal point in its heuristic
    - A* will not investigate nodes in direction of alternative goal post until late
Hierarchical Pathfinding

- Dynamic pathfinding
  - Situation can change
    - Pathfinding needs to run again
  - D* algorithm
    - Similar to A*, but updates costs in the open node when the environment changes
Adaptive A*

- A* can use lots of memory
  - IDA*
    - Starts with a cut-off value
    - Explores path only if they are below the cut-off value
    - Uses A* heuristics to determine nodes that should be considered
  - SMA*
    - Uses a fixed limit on the number of “open” nodes
Continuous Time Pathfinding

- Pathfinding task can change quickly, but predictably
  - Example: Police car in a freeway chase
    - Other cars move at constant speed in same lane, police car changes lanes and speeds
  - Solution:
    - Limit problem heuristically
      - Change lanes as soon as possible
      - Move in current lane to potential lane change as quickly as possible
    - Create a graph where each node corresponds to a possible situation
      - Connections:
        - Lane change nodes: Time required to change lanes at current speed
        - Boundary node: Travel in same lane as fast as possible, but brake before slamming into preceding car
        - Safe opportunity nodes: Car travels in same lane as fast as possible until a point where a lane change can be made
        - Unsafe opportunity nodes: Same, but do not brake in order not to slam in the preceding car.
Movement Planning

- Extensions of nodes taking into account different types of motion
  - Animation defines different types of movement
  - Only certain speeds / rotations can be represented

- Create a *movement graph*
  - Each node represents
    - position
    - velocity
    - possible animations
  - Connections only if there is an animation
Footfall

- Movement graph where a node corresponds to a certain foot setting