CS 387/680: GAME AI

PATHFINDING

4/16/2015
Instructor: Santiago Ontañón
santi@cs.drexel.edu

Class website:
https://www.cs.drexel.edu/~santi/teaching/2015/CS387/intro.html
Reminders

- Check BBVista site for the course regularly
- Also: https://www.cs.drexel.edu/~santi/teaching/2015/CS387/intro.html

- Next week is the deadline for people doing Project 1.
  - Submission via learn.drexel.edu
  - Submission page available
  - Questions?
  - Curious video of how did they “solve” steering behaviors in classic games (from 0:45)
    https://www.youtube.com/watch?v=3A5AChi2IMw
Outline

• Student Presentations

• Pathfinding Basics: Breadth First Search, A*
• Pathfinding under Time Constraints: TBA*, LRTA*
• Pathfinding in Dynamic Domains: D*, D* Lite, AD*
• From Level Geometry to Pathfinding Graphs
• Pathfinding and Movement
Outline

- Student Presentations

- **Pathfinding Basics: Breadth First Search, A***
- Pathfinding under Time Constraints: TBA*, LRTA*
- Pathfinding in Dynamic Domains: D*, D* Lite, AD*
- From Level Geometry to Pathfinding Graphs
- Pathfinding and Movement
A*

- **Heuristic search algorithm:**
  - Finds the shortest path between a given start node and a goal
  - Uses a heuristic to guide the search
  - Heuristic: \( h(n) \) -> estimation of the distance from \( n \) to the goal

- **Complete:** if there is a solution, A* finds one

- **Optimal:** if heuristic **admissible**, A* finds the optimal path
  - A heuristic is admissible if it never overestimates the distance to the goal
  - In Pathfinding: Euclidean or Hamming distances are admissible heuristics
Example: A*

Expands the node with the lowest Estimated cost first

OPEN = [A, B, C]
CLOSED = [Start]
A*

Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED nor OPEN:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
ENDFOR
ENDWHILE
Outline

- Student Presentations
- Pathfinding Basics: Breadth First Search, A*
  - **Pathfinding under Time Constraints:** TBA*, LRTA*
  - Pathfinding in Dynamic Domains: D*, D* Lite, AD*
- From Level Geometry to Pathfinding Graphs
- Pathfinding and Movement
TBA*

- Uses A* as a subroutine (executing only NS steps of A* at a time)
LRTA* (Learning Real-Time A*)

- Real-Time A* variant that converges to optimal path:
  - Unit can start moving from step 1, but might waste moves
  - Eventually, algorithm converges to the optimal path, and character reaches destination

- Idea:
  - Learn a better heuristic
  - After each iteration of the algorithm, the heuristic better approximates the real distances to the goal
  - Eventually, it converges to the real distance to the goal
  - Unit just moves to the cell with lowest heuristic around until reaching the goal
LRTA*

Initializes the heuristic in each cell to $h$

For this example, we will use Manhattan distance
LRTA*  

In each step:
- **Breadth-first search with bounded depth (e.g. 1)**
- Updates the heuristic of the current node to the best $f$ found
- Move to the lowest heuristic cell around

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th></th>
<th>Goal 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
LRTA*

In each step:
- Breadth-first search with bounded depth (e.g. 1)
- Updates the heuristic of the current node to the best $f$ found
- Move to the lowest heuristic cell around

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Start 5</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
Outline

• Student Presentations

• Pathfinding Basics: Breadth First Search, A*
• Pathfinding under Time Constraints: TBA*, LRTA*
• **Pathfinding in Dynamic Domains:** D*, D* Lite, AD*
• From Level Geometry to Pathfinding Graphs
• Pathfinding and Movement
Pathfinding in Dynamic Environments

- A* variants (TBA*, LRTA*, etc.) assume environment doesn’t change
- In a RTS game, units move, new buildings are constructed and destroyed: environment is dynamic
- What happens if after the game has a path using A*/TBA*/LRTA*, the map changes?
Pathfinding in Dynamic Environments

- A* variants (TBA*, LRTA*, etc.) assume environment doesn’t change

- In a RTS game, units move, new buildings are constructed and destroyed: environment is dynamic

- Simple solution (used in many games!):
  - Recompute the path each time something changes (always!)
  - (That’s what you are doing for Project 2)

- Better solution:
  - Specialized algorithms that can take this into account
D* Lite

- Same behavior as D* (which we will NOT cover in class), but simpler

- Works on dynamic environments:
  - Modeled by changing the cost of moving from one cell to another
  - If one cell $n$ is not reachable from another $n'$, the cost from $n$ to $n'$ is infinity

- Searches from goal to start

- Uses a “consistency test” on nodes to determine when more search is needed:
  - At the start
  - When the environment changes
D* Lite

- Maintains an estimate (rhs) of the distance to the goal:
  - A* maintains the g-estimates: actual distance from start
  - rhs-estimate: 1-step look ahead g-estimate

\[
rhs(n) = \min_{n' \in Succ(n)} (c(n, n') + g(n'))
\]

- If \( g(n) = rhs(n) \), \( n \) is considered consistent

- If all nodes are consistent, finding the optimal path is just moving to the best neighbor

- Search is only carried out on inconsistent nodes. When the environment changes, nodes become inconsistent
D* Lite: Node Consistency

Node n consistent if: \( rhs(n) = g(n) \)

\[
\begin{align*}
    rhs(n) &= \min_{n' \in Succ(n)} (c(n, n') + g(n'))
\end{align*}
\]

Which are the “rhs” values for these nodes, and are they consistent?
D* Lite: Node Consistency

Node n consistent if: $rhs(n) = g(n)$

$$rhs(n) = \min_{n' \in \text{Succ}(n)} (c(n, n') + g(n'))$$

Which are the “rhs” values for these nodes, and are they consistent?

Inconsistent!
D* Lite (Simplified Algorithm)

Initialize $g$ and $rhs$ of all nodes to infinity
$g(\text{Goal}) = rhs(\text{Goal}) = 0$
$U = \{\text{Goal}\}$
$Loc = \text{Start}$
WHILE $Loc \neq \text{Goal}$
  ComputeShortestPath()
  $Loc = \text{bestNeighbor}(Loc)$
  FOR all cells $N$ with cost changed
    Update $rhs(N)$
    Add to $U$ any inconsistent neighbor of $N$
  ENDFOR
ENDWHILE
D* Lite (Simplified Algorithm)

Initialize $g$ and $rhs$ of all nodes to infinity
$g$(Goal) = $rhs$(Goal) = 0
$U$ = [Goal]
Loc = Start
WHILE Loc != Goal
  ComputeShortestPath()
  Loc = bestNeighbor(Loc)
  FOR all cells $N$ with cost changed
    Update $rhs(N)$
    Add to $U$ any inconsistent neighbor of $N$
  ENDIF
ENDWHILE
**D* Lite (Full Algorithm)**

```plaintext
procedure CalculateKey(s)
    {01'} return \[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))\];

procedure Initialize()
    {02'} U = \emptyset;
    {03'} \(k_m = 0\);
    {04'} for all \(s \in S\) \(rhs(s) = g(s) = \infty\);
    {05'} \(rhs(s_{goal}) = 0\);
    {06'} U.Insert(\(s_{goal}\), CalculateKey(\(s_{goal}\)));

procedure UpdateVertex(u)
    {07'} if \((u \neq s_{goal}) rhs(u) = \min_{s' \in succ(u)} (e(u, s') + g(s'))\);
    {08'} if \((u \in U)\) U.Remove(u);
    {09'} if \((g(u) \neq rhs(u))\) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
    {10'} while U.TopKey() < CalculateKey(s_{start}) OR rhs(s_{start}) \(\neq g(s_{start})\)
    {11'} \(k_{old} = U/top()\);
    {12'} \(u = U/pop()\);
    {13'} if \((k_{old} < CalculateKey(u))\)
        \(U/insert(u, CalculateKey(u))\);
    {14'} else if \((g(u) > rhs(u))\)
        \(g(u) = rhs(u)\);
    {15'} for all \(s \in Pred(u)\) UpdateVertex(s);
    {16'} else
        \(g(u) = \infty\);
    {17'} for all \(s \in Pred(u) \cup \{u\}\) UpdateVertex(s);

procedure Main()
    {18'} \(s_{last} = s_{start}\);
    {19'} Initialize();
    {20'} ComputeShortestPath();
    {21'} if \((s_{start} \neq s_{goal})\)
        then there is no known path */
    {22'} \(s_{start} = \arg\ min_{s' \in succ(s_{start})} (c(s_{start}, s') + g(s'))\);
    {23'} Move to \(s_{start}\);
    {24'} Scan graph for changed edge costs;
    {25'} if any edge costs changed
    {26'} \(k_m = k_m + h(s_{last}, s_{start})\);
    {27'} \(s_{last} = s_{start}\);
    {28'} for all directed edges \((u, v)\) with changed edge costs
        Update the edge cost \(c(u, v)\);
    {29'} UpdateVertex(u);
    {30'} ComputeShortestPath();
```

This method is equivalent to A*, but using g and rhs (ComputeShortestPath from previous slide)

Part shown in the previous slide
Example: D* Lite

- All g and rhs are initialized to infinity
- Goal g and rhs initialized to 0
- The only inconsistent node is Goal

\[
U = [\text{Goal}]
\]

| g = \infty | g = \infty | Goal \\
| rhs = \infty | rhs = \infty | g = 0 \\
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |

| g = \infty | g = \infty | g = \infty |
| rhs = \infty | rhs = \infty | rhs = \infty |
Example: D* Lite

- After running `ComputeShortestPath`, the nodes relevant for the path from Start to Goal will be consistent.

<table>
<thead>
<tr>
<th>g = 9</th>
<th>g = 8</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs = 9</td>
<td>rhs = 8</td>
<td>g = 0</td>
</tr>
<tr>
<td>rhs = 8</td>
<td>rhs = 7</td>
<td>rhs = 1</td>
</tr>
<tr>
<td>rhs = 7</td>
<td>rhs = 6</td>
<td>rhs = 2</td>
</tr>
<tr>
<td>rhs = 6</td>
<td>rhs = 5</td>
<td>rhs = 3</td>
</tr>
</tbody>
</table>
Example: D* Lite

- The Unit can now move by choosing the minimum neighbor around

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>g = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rhs = 0</td>
</tr>
<tr>
<td>g = 8</td>
<td>rhs = 8</td>
<td></td>
</tr>
<tr>
<td>g = 7</td>
<td>rhs = 7</td>
<td>g = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rhs = 1</td>
</tr>
<tr>
<td>g = 6</td>
<td>rhs = 6</td>
<td>g = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rhs = 2</td>
</tr>
<tr>
<td>g = 5</td>
<td>rhs = 5</td>
<td>g = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rhs = 3</td>
</tr>
</tbody>
</table>

\[ U = [] \]
Example: D* Lite

- The Unit can now move by choosing the minimum neighbor around

<table>
<thead>
<tr>
<th></th>
<th>g = 9</th>
<th>g = 8</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs</td>
<td>rhs = 9</td>
<td>rhs = 8</td>
<td>rhs = 0</td>
</tr>
<tr>
<td></td>
<td>g = 8</td>
<td>g = 7</td>
<td></td>
</tr>
<tr>
<td>rhs</td>
<td>rhs = 8</td>
<td>rhs = 7</td>
<td>rhs = 1</td>
</tr>
<tr>
<td></td>
<td>g = 7</td>
<td>g = 6</td>
<td></td>
</tr>
<tr>
<td>rhs</td>
<td>rhs = 7</td>
<td>rhs = 6</td>
<td>rhs = 2</td>
</tr>
<tr>
<td></td>
<td>g = 6</td>
<td>g = 5</td>
<td></td>
</tr>
<tr>
<td>rhs</td>
<td>rhs = 6</td>
<td>rhs = 5</td>
<td>rhs = 3</td>
</tr>
<tr>
<td></td>
<td>g = 4</td>
<td>g = 4</td>
<td></td>
</tr>
<tr>
<td>rhs</td>
<td>rhs = 4</td>
<td>rhs = 4</td>
<td>rhs = 2</td>
</tr>
<tr>
<td></td>
<td>g = 3</td>
<td>g = 3</td>
<td></td>
</tr>
<tr>
<td>rhs</td>
<td>rhs = 3</td>
<td>rhs = 3</td>
<td>rhs = 2</td>
</tr>
</tbody>
</table>

What if this wall disappears?
Example: D* Lite

- If something changes, nodes become inconsistent, and are pushed back into U

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>g = 9</td>
<td>rhs = 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g = 8</td>
<td>rhs = 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g = 7</td>
<td>rhs = 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g = 6</td>
<td>rhs = 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g = 5</td>
<td>rhs = 5</td>
<td>g = 4</td>
<td>g = 3</td>
</tr>
<tr>
<td>g = 4</td>
<td>rhs = 4</td>
<td>g = 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

U = [A, B, Goal]
Example: D* Lite

- After running `ComputeShortestPath`, again, nodes are made consistent again.

- No need to replan from scratch, only inconsistent nodes are updated, and their changes propagated.
Example: D* Lite

- The Unit can again move by choosing the minimum neighbor around

<table>
<thead>
<tr>
<th>g = 3</th>
<th>A</th>
<th>B</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs = 3</td>
<td>g = 2</td>
<td>g = 1</td>
<td>g = 0</td>
</tr>
<tr>
<td></td>
<td>rhs = 2</td>
<td>rhs = 1</td>
<td>rhs = 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>g = 4</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs = 4</td>
<td>g = 3</td>
<td>g = 1</td>
<td>g = 1</td>
</tr>
<tr>
<td></td>
<td>rhs = 3</td>
<td>rhs = 1</td>
<td>rhs = 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>g = 5</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs = 5</td>
<td>g = 4</td>
<td>g = 2</td>
<td>g = 2</td>
</tr>
<tr>
<td></td>
<td>rhs = 4</td>
<td>rhs = 2</td>
<td>rhs = 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>g = 6</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>rhs = 6</td>
<td>g = 5</td>
<td>g = 4</td>
<td>g = 3</td>
</tr>
<tr>
<td></td>
<td>rhs = 5</td>
<td>rhs = 4</td>
<td>rhs = 3</td>
</tr>
</tbody>
</table>
D* Lite

- D* Lite can handle dynamic environments
- Doesn’t need to replan from scratch
- As presented it is not real-time (ComputeShortestPath might have to update an arbitrary number of nodes):
- It can be made real-time by limiting the number of updates ComputeShortestPath performs per cycle
Other Algorithms

• **D***
  - Previous to D* Lite
  - Same behavior, but more complex

• **AD*** (Anytime Dynamic A*)
  - Tunes quality of paths depending on available time
  - Reuses past paths in new searches to improve their quality
  - Uses a “bloating” coefficient to find paths faster when little time available (a multiplier of the heuristic)
Outline

• Student Presentations

• Pathfinding Basics: Breadth First Search, A*
• Pathfinding under Time Constraints: TBA*, LRTA*
• Pathfinding in Dynamic Domains: D*, D* Lite, AD*
• From Level Geometry to Pathfinding Graphs
• Pathfinding and Movement
Waypoints

- Predefine a collection of waypoints, and do pathfinding in the waypoint graph:
Precomputation

• If environment is largely static paths can be precomputed

• To reduce the amount of precomputation:
  • Precomputation only in the waypoint graph
  • Precomputation of paths between each possible map location is not feasible

• At runtime, the precomputed path can be used as a starting point:
  • If environment changes a bit, the path can be modified locally using obstacle avoidance techniques (faster than pathfinding)
Quantization and Localization

- Pathfinding computations are performed with an abstraction over the game map (a graph, composed of nodes and links)

- **Quantization:**
  - Game map coordinates -> graph node

- **Localization:**
  - Graph node -> Game map coordinates
Good and Bad Quantizations

- The pathfinding graph has to be created carefully, or weird pathfinding problems might occur:

![Diagrams showing good and bad quantizations](image_url)
Quantization approaches

- Tile graphs (simplest)
- Dirichlet Domains (aka Voronoi polygon)
- Points of visibility
- Navigation Meshes
Tile Graphs

- Divide the game map in equal tiles (squares, hexagons, etc.)
- Typical in RTS games
Dirichlet Domains

- Define a set of **characteristic points** in the game map (typically defined by the level designer)

Quantization is just computing the closest characteristic point
Dirichlet Domains

- Define a set of characteristic points in the game map (typically defined by the level designer)
Points of Visibility

• The shortest path between two points is a straight line

• Generate a collection of characteristic points in the map

• Connect two points if the character can move in a straight line between them

• Characteristic points can be generated automatically (for example at the corners of the map)
Points of Visibility

- Joint with Dirichlet domains, this method is widely used. However, the resulting graph can be huge!
Navigation Meshes (Navmesh)

• By far the most widely used
• Use a mesh of triangles (center of each triangle is a node in the navigation graph).
• The mesh can be:
  • Procedurally generated:
    • e.g., directly using the level geometry as the pathfinding graph
    • Floor is made out of triangles: Use the center of each “floor” triangle as a characteristic point.
  • Manually generated (error prone)
Navigation Meshes (Navmesh)

- One characteristic point is only connected to neighboring ones
Navigation Meshes (Navmesh)

- The validity of the graph depends on the level designer. It is her responsibility to author proper triangles for navigation:

![Diagram of navigation mesh]

- Usefulness

Using this approach also requires additional processing to take into account the agent's geometry. Since not all locations in a floor polygon may be occupied by a character (some are too close to the wall), some trimming is required; this may affect the connections generated by finding shared edges. This problem is especially evident at convex areas such as doorways.

Despite this, it is an overwhelmingly popular approach. Games such as *Jak and Daxter* [Naughty Dog, Inc., 2001] and hundreds of others use this approach, as does the PathEngine middleware solution. For the occasional game that needs it, this approach offers the additional benefit of allowing characters to plan routes up walls, across ceilings, or for any other kind of geometry. This might be useful if characters stick to walls, for example. It is much more difficult to achieve the same result with other world representations.
Hierarchical Pathfinding

- If the path that needs to be planned is long, it doesn’t make sense to use a standard A* graph
- Hierarchical graphs
Pathfinding

- Simplest scenario:
  - Single character
  - Non-real time
  - Grid
  - Static world
  - **Solution: A***

- Complex scenario:
  - Multiple characters/Dynamic world: D* / D* Lite
  - Real time: TBA* / LRTA*
  - Continuous map: Quantization
Outline

- Student Presentations
- Pathfinding Basics: Breadth First Search, A*
- Pathfinding under Time Constraints: TBA*, LRTA*
- Pathfinding in Dynamic Domains: D*, D* Lite, AD*
- From Level Geometry to Pathfinding Graphs
- Pathfinding and Movement
Game AI Architecture

- Strategy
- Decision Making
- Movement
- World Interface (perception)
Game AI Architecture

- Strategy
- Decision Making
- Movement

World Interface (perception)

Steering behaviors / Movement executed in this module
Game AI Architecture

- **Strategy**
- **Decision Making**
- **Movement**

Pathplanning sits in the middle. Sometimes in “Movement”, sometimes in “Decision Making”
Example

- Grand Theft Auto
Example

Example

- Grand Theft Auto:
  - Pathfinding graph contains roads and intersections as node graphs
  - Pathfinding plans the roads to take
  - Movement controls the car to actually follow the path (same as in the Steering Behaviors project 1)

- In games where the only decision making is path planning, then pathfinding IS the decision making module
- In games where decision making involves other things (e.g. when to attack, when to retreat, etc.), pathfinding is just a layer in between
Summary

- **A***:
  - Optimal paths

- **TBA*/LRTA***:
  - Real-time path generation (bounded computation at each execution cycle)

- **D*-Lite**:
  - Dynamic pathfinding (changing obstacles in the map)

- **Quantization**:
  - From game coordinates to a graph node.
Project 2: Pathfinding

- Implement A* in a RTS Game
- Game Engine: S3 (Java)
Project 1: Steering Behaviors

- Implement steering behaviors
- Game Engine:
  - Simple car driving (Java)
- Goals:
  - Seek
  - Arrive
  - Wall avoidance
- But please feel free to go beyond these!