Pathfinding on a GPU
Questions

Introduction to pathfinding.

Pathfinding on a GPU

- Based on “GPU Accelerated Pathfinding” by Avi Bleiweiss (NVIDIA)
Questions

- Name two problems with using a grid-based search space.
- What does it mean that the A* heuristic is/should be admissible?
- What was the maximum number of nodes in the graph “they” tried?
Pathfinding

- Basic problem: getting from point A to B along the best route.
- “Best” often means shortest or fastest although this is debatable.
- Areas of use: robotics, unmanned vehicles, game AI, etc.
Pathfinding – search space

- Grid-based
- Roadmaps
- Visibility Graphs
- Hierarchial pathfinding
- Corridor Map Method
Pathfinding – Grid-Based

- The world is divided into a regular grid. For each cell you check if there is an obstacle in it.
Pathfinding – Grid-Based

- **Drawbacks:**
  - very large search-space if the environment is large.
  - unnatural-looking paths.
Pathfinding – Roadmaps

- Character/robot can only move between given points. These points are usually manually defined.
Pathfinding – Roadmaps

- **Drawbacks:**
  - Manually defined!
  - Does not provide optimal routes.
  - May give strange-looking paths.
Pathfinding – Probabilistic Road Maps
Each corner/vertex of an obstacle is a node in the graph. If there is no collision between two nodes, a link is set between them.
Pathfinding – Visibility Graphs

- Drawbacks:
  - wall-hugging
  - not very natural-looking paths
Pathfinding – Hierarchical Pathfinding

- Creates hierarchies.
- Good for room-like structures.
Pathfinding – Hierarchical Pathfinding

- **Drawbacks:**
  - Difficult to make automatically
  - Does not work well for large open areas (as in a lot of RTS games)
Pathfinding - Corridor Map Method

- Two phases: construction, query
- Nice-looking paths
- Drawbacks: Only obstacle or no obstacle, Environment may not change.

Fig. 1. The construction phase (top) and the query phase (bottom) of the Corridor Map Method.
Pathfinding – Search Algorithms

Several algorithms:
- A*
- Dijkstra
- D*
- ...

A* by far the most commonly used.
Pathfinding – A*

- Best-first, graph-search algorithm.
- Uses a heuristic function to estimate cost from a given node to the goal.
  - The euclidean distance is often used.
  - If heuristic function $h$ is admissible (never overestimates the actual minimal cost of reaching the goal), then A* is optimal if we do not use a closed set.
- Keeps track of nodes already visited.
- Only expands the node with the least cost so far + the estimated cost.
Pathfinding – A*

\[ f(a) = 1.5 + 4 = 5.5 \]
\[ f(d) = 2 + 4.5 = 6.5 \]
Pathfinding – A*

h(a) = 4
h(b) = 2
h(c) = 4
h(e) = 2

f(a) = 1.5 + 4 = 5.5
f(b) = 3.5 + 2 = 5.5
f(d) = 2 + 4.5 = 6.5
f(d) = 2 + 4.5 = 6.5

h(d) = 4.5

f(a) = 1.5 + 4 = 5.5
f(d) = 2 + 4.5 = 6.5

h(b) = 2
h(c) = 4
h(e) = 2
Pathfinding – A*

The image shows a graph with the following nodes and edge weights:

- Node a: h(a) = 4
- Node b: h(b) = 2
- Node c: h(c) = 4
- Node d: h(d) = 4.5
- Node e: h(e) = 2

The edge weights are as follows:
- a to b: 1.5
- a to c: 3
- b to c: 2
- b to d: 2
- c to d: 4
- c to e: 4
- d to e: 2
- d to f: 4.5
- e to f: 2

The heuristic values are:

- h(a) = 4
- h(b) = 2
- h(c) = 4
- h(d) = 4.5
- h(e) = 2

The cost function f is calculated as follows:

- f(a) = 1.5 + 4 = 5.5
- f(d) = 2 + 4.5 = 6.5
- f(b) = 3.5 + 2 = 5.5
- f(d) = 2 + 4.5 = 6.5
- f(c) = 6.5 + 4 = 10.5
- f(d) = 2 + 4.5 = 6.5
Pathfinding – A*

h(a) = 4
h(b) = 2
h(c) = 4
h(e) = 2

f(a) = 1.5 + 4 = 5.5
f(b) = 3.5 + 2 = 5.5
f(c) = 6.5 + 4 = 10.5
f(d) = 2 + 4.5 = 6.5
f(e) = 5 + 2 = 7
Pathfinding – A*

\[ h(a) = 4 \]

\[ h(b) = 2 \]

\[ h(c) = 4 \]

\[ h(d) = 4.5 \]

\[ h(e) = 2 \]

\[ f(a) = 1.5 + 4 = 5.5 \]

\[ f(b) = 3.5 + 2 = 5.5 \]

\[ f(c) = 6.5 + 4 = 10.5 \]

\[ f(d) = 2 + 4.5 = 6.5 \]

\[ f(e) = 5 + 2 = 7 \]

\[ f(G) = 8 + 0 = 8 \]
Pathfinding – A*

OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
    current = remove lowest rank item from OPEN
    add current to CLOSED
    for neighbors of current:
        cost = g(current) + movementcost(current, neighbor)
        if neighbor in OPEN and cost less than g(neighbor):
            remove neighbor from OPEN, because new path is better
        if neighbor in CLOSED and cost less than g(neighbor): // if we have inadmissible heuristics.
            remove neighbor from CLOSED
        if neighbor not in OPEN and neighbor not in CLOSED:
            set g(neighbor) to cost
            add neighbor to OPEN
            set priority queue rank to g(neighbor) + h(neighbor)
            set neighbor's parent to current
    reconstruct reverse path from goal to start by following parent pointers
Pathfinding – A*

\[ f = \text{priority queue element } \{ \text{node index, cost} \} \]
\[ F = \text{priority queue containing initial } f(0,0) \]
\[ G = \text{g cost set initialized to zero} \]
\[ P,S = \text{pending and shortest nullified edge sets} \]
\[ n = \text{closest node index} \]
\[ E = \text{node adjacency list} \]

while \( F \) not empty do
    \( n \leftarrow F.\text{Extract()} \)
    \( S[n] \leftarrow P[n] \)
    if \( n \) is goal then return SUCCESS
    foreach edge \( e \) in \( E[n] \) do
        \( h \leftarrow \text{heuristic}(e.\text{to}, \text{goal}) \)
        \( g \leftarrow G[n] + e.\text{cost} \)
        \( f \leftarrow \{ e.\text{to}, g+h \} \)
        if not in \( P \) or \( g < G[e.\text{to}] \) and not in \( S \) then
            \( F.\text{Insert}(f) \)
            \( G[e.\text{to}] \leftarrow g \)
            \( P[e.\text{to}] \leftarrow e \)
    return FAILURE
Pathfinding – A*

Problems with A*:
- Can be very time-consuming if heuristic is bad.
- Consumes a lot of memory.
Other problems:
- Replanning needed often due to changes in environment!
- Usually done for hundreds of characters at a time (RPS games)!
A* on a GPU

"GPU Accelerated Pathfinding" Avi Bleiweiss, NVIDIA Corporation, Eurographics 2008

Goal: “to exploit general data parallelism in performing global navigation planning for many thousands of agents.”

Comparison with an equivalent CPU implementation.

Memory structure tweaked.
### A* on a GPU

#### Node

<table>
<thead>
<tr>
<th>id</th>
<th>position.x</th>
<th>position.y</th>
<th>position.z</th>
</tr>
</thead>
</table>

#### Edge

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>cost</th>
<th>reserved</th>
</tr>
</thead>
</table>

#### Adjacency

<table>
<thead>
<tr>
<th>offset</th>
<th>offset+count</th>
</tr>
</thead>
</table>

```
1 | 2 | 2 | ?
1 | 3 | 1 | ?
2 | 3 | 2 | ?
0 | 2
2 | 3
```
A* on a GPU

Each agent corresponds to one thread.

The A* kernel has five input arrays:
- A list of paths. The start node id and goal node id for each agent.
  - \{\{1,3\},\{2,5\},...\}
- A list of costs from the start position G, initialized to zero.
  - \{\{0\},\{0\},...\}
- A list of combined costs from start to goal.
  - \{\{3\},\{6\},...\}
- A list of pointers for the pending (P) edge collections.
- A list of pointers for the shorest (S) edge collections.

... and two outputs:
- List of accumulated costs.
- List of paths.
A* on a GPU

Test setup

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
<th>Agents</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>G1</td>
<td>32</td>
<td>178</td>
<td>1024</td>
<td>8</td>
</tr>
<tr>
<td>G2</td>
<td>64</td>
<td>302</td>
<td>4096</td>
<td>32</td>
</tr>
<tr>
<td>G3</td>
<td>129</td>
<td>672</td>
<td>16641</td>
<td>131</td>
</tr>
<tr>
<td>G4</td>
<td>245</td>
<td>1362</td>
<td>60025</td>
<td>469</td>
</tr>
<tr>
<td>G5</td>
<td>340</td>
<td>2150</td>
<td>115600</td>
<td>904</td>
</tr>
</tbody>
</table>

115600(!) agents but only 340 nodes?

340 nodes ~ 18x18 grid! Not exactly huge.
A* on a GPU

<table>
<thead>
<tr>
<th>Threads per Block</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers per Block</td>
<td>2560</td>
</tr>
<tr>
<td>Warps per Block</td>
<td>4</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>384</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>3</td>
</tr>
<tr>
<td>Thread Blocks per GPU</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph</th>
<th>Roadmap</th>
<th>Working Set</th>
<th>Total</th>
<th>Launches</th>
</tr>
</thead>
<tbody>
<tr>
<td>G0</td>
<td>0.576</td>
<td>0.021</td>
<td>0.021</td>
<td>1</td>
</tr>
<tr>
<td>G1</td>
<td>3.616</td>
<td>1.319</td>
<td>1.322</td>
<td>1</td>
</tr>
<tr>
<td>G2</td>
<td>6.368</td>
<td>10.518</td>
<td>10.519</td>
<td>1</td>
</tr>
<tr>
<td>G3</td>
<td>13.848</td>
<td>86.001</td>
<td>86.001</td>
<td>1</td>
</tr>
<tr>
<td>G4</td>
<td>27.672</td>
<td>588.726</td>
<td>588.726</td>
<td>2</td>
</tr>
<tr>
<td>G5</td>
<td>42.560</td>
<td>1573.086</td>
<td>1573.086</td>
<td>3</td>
</tr>
</tbody>
</table>

**Figure 7:** NVIDIA’s CUDA Occupancy Calculator tool generated output for the default pathfinding block of 128 threads, running on current generation GPU.

**Figure 9:** Benchmark’s GPU global memory footprint for each the roadmap (KBytes), working set (MBytes) and total (MBytes). Multiple launches are the result of exceeding available GPU global memory.
A* on a GPU

Results: Running Djikstra algorithm (A* with heuristics zero):

![Graph showing speedup](image)

**Figure 12:** Performance of GPU running CUDA A* search algorithm using Euclidian heuristic, compared to CPU plain optimized C++ code and to hand-compiler tuned SIMD intrinsics (SSE) implementation.
A* on a GPU

“GPU performance speedup for Dijkstra (against scalar C++) and A* (compared to the SSE implementation) searches reached up to 27X and 24X, respectively.”

Maybe so, but odd testing parameters.

I would like to see:
- 500 agents and 2000+ nodes (common RTS scenario).
A* on a GPU

~ 18x12
... and that's just what's in view.
A* on a GPU

- Not to mention Starcraft
Thank you!