

Pathfinding on a GPU

Introduction

- 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

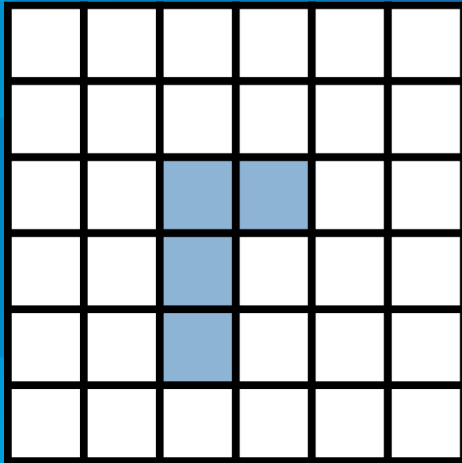
- Basic problem: getting from point A to B along the best route.
- “Best” often means shortest or fastest although this is debatable.
- Areas of use: robotics, unmanned vehicles, game AI, etc.

Pathfinding – search space

- Grid-based
- Roadmaps
- Visibility Graphs
- Hierarchical pathfinding
- Corridor Map Method

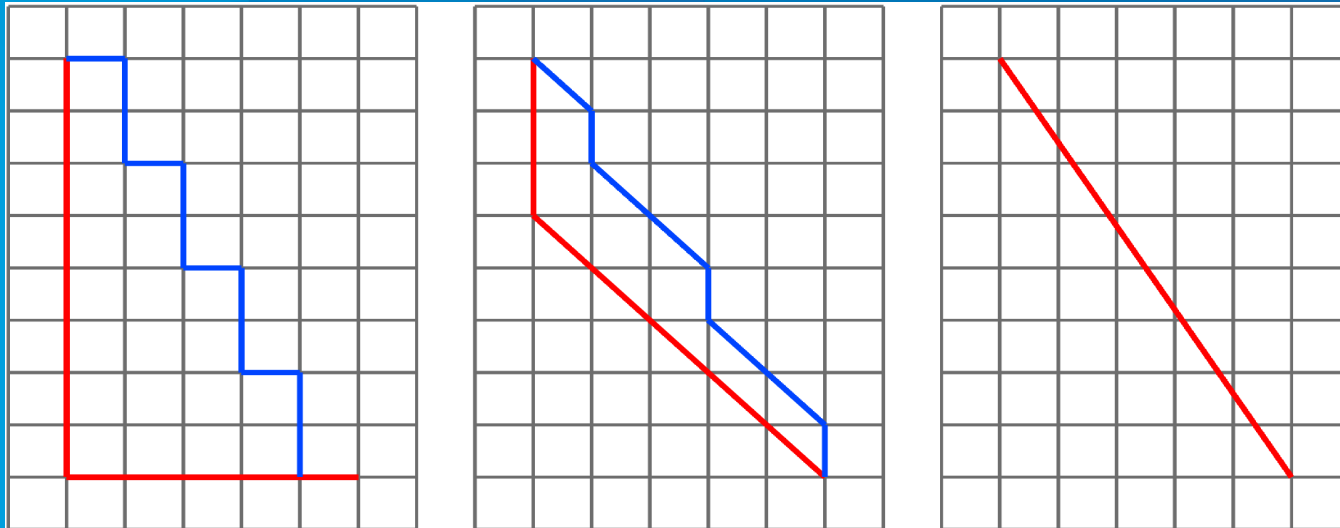
Pathfinding – Grid-Based

- 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.

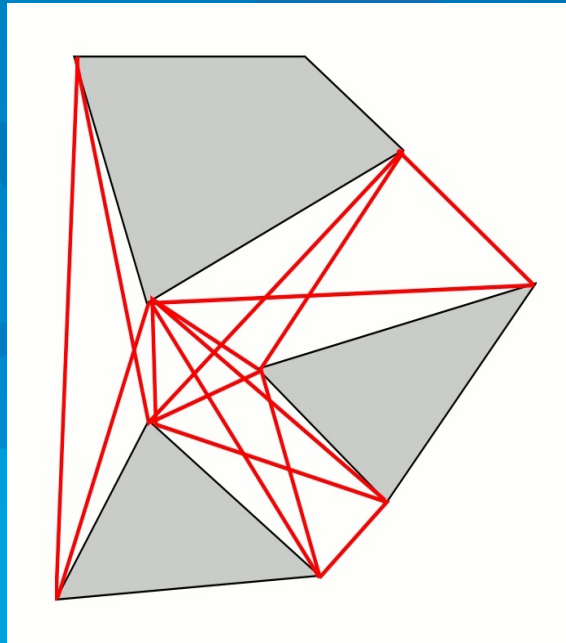


Pathfinding – Roadmaps

- Drawbacks:
 - Manually defined!
 - Does not provide optimal routes.
 - May give strange-looking paths.

Pathfinding – Visibility Graphs

- 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.

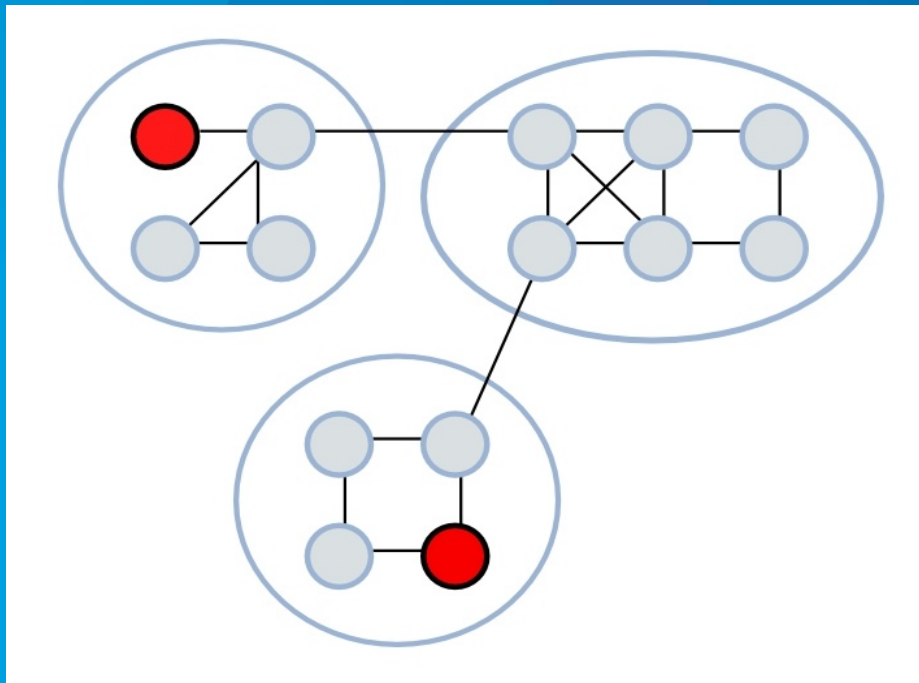


Pathfinding – Visibility Graphs

- Drawbacks:
 - wall-hugging
 - not very natural-looking paths

Pathfinding – Hierarchical Pathfinding

- Creates hierarchies.
- Good for room-like structures.

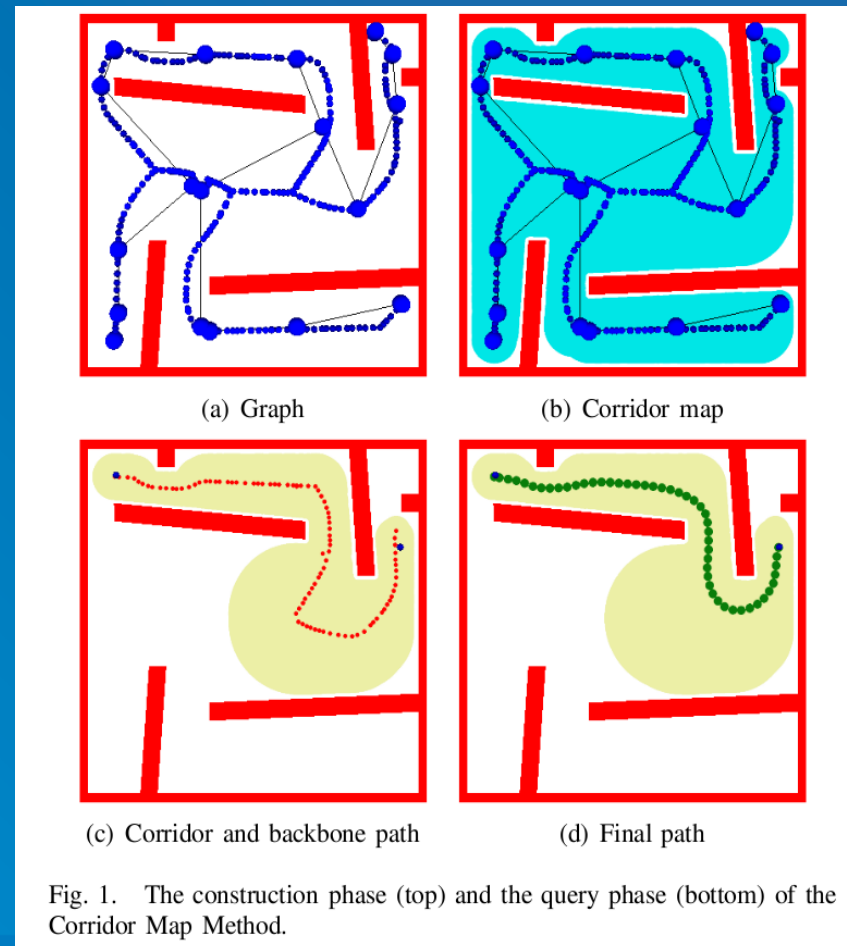


Pathfinding – Hierarchical Pathfinding

- 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.



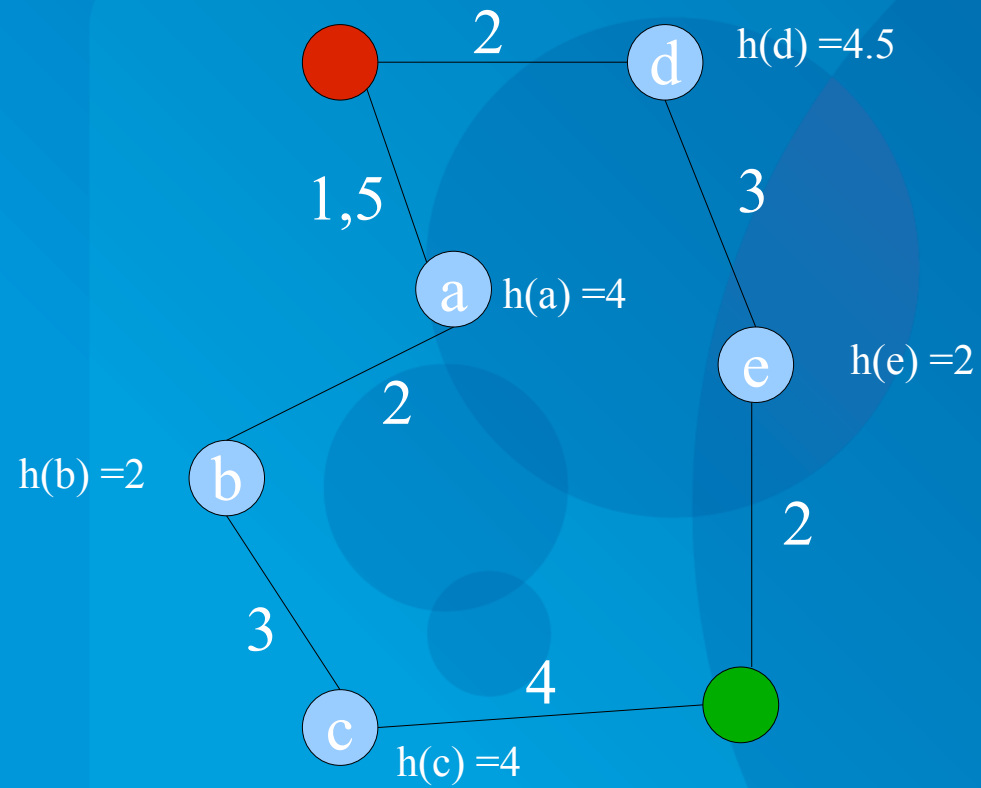
Pathfinding – Search Algorithms

- Several algorithms:
 - A*
 - Dijkstra
 - D*
 - ...
- A* by far the most commonly used.

Pathfinding – A*

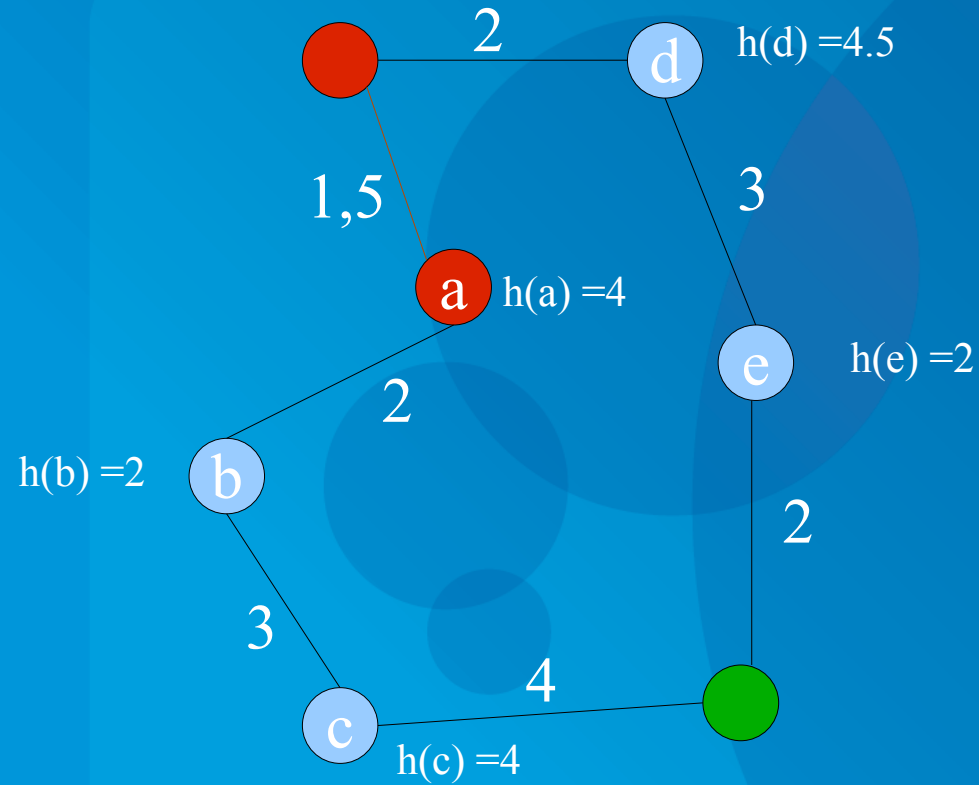
- Best-first, graph-search algorithm.
- Uses a heuristic function to estimate cost from a given node to the goal.
 - The euclidean distance is often used.
 - If heuristic function h is admissible (never overestimates the actual minimal cost of reaching the goal), then A* is optimal if we do not use a 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*



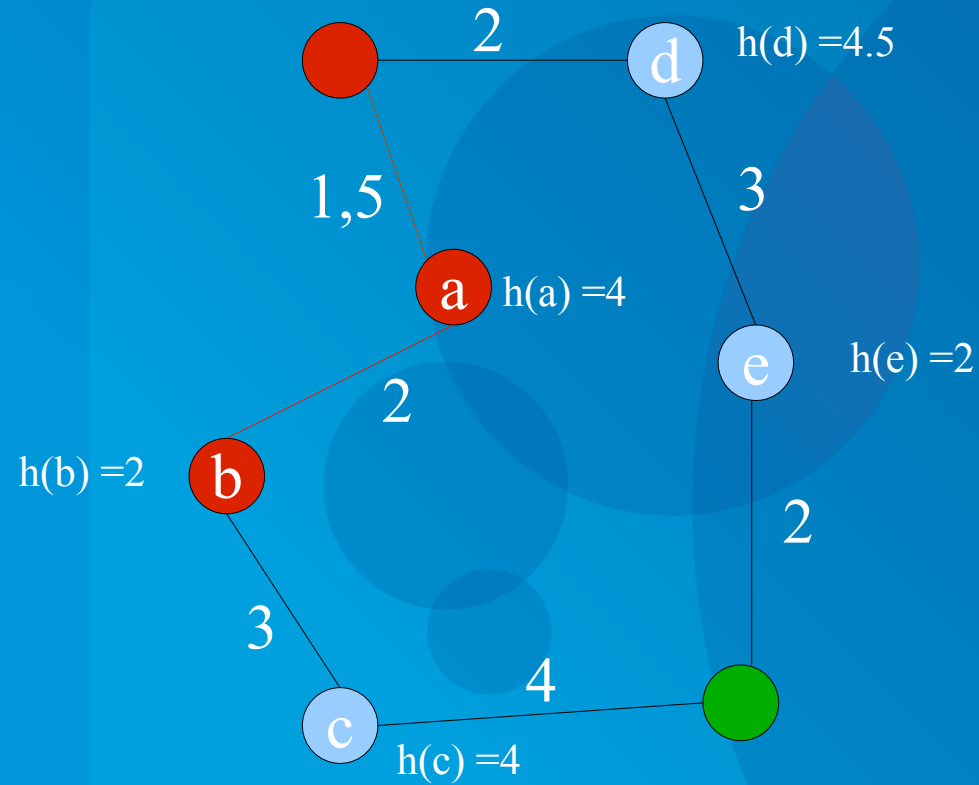
~~$f(a) = 1.5 + 4 = 5.5$~~

~~$f(d) = 2 + 4.5 = 6.5$~~

$f(b) = 3.5 + 2 = 5.5$

$f(d) = 2 + 4.5 = 6.5$

Pathfinding – A*



~~$$f(a) = 1.5 + 4 = 5.5$$~~

~~$$f(d) = 2 + 4.5 = 6.5$$~~

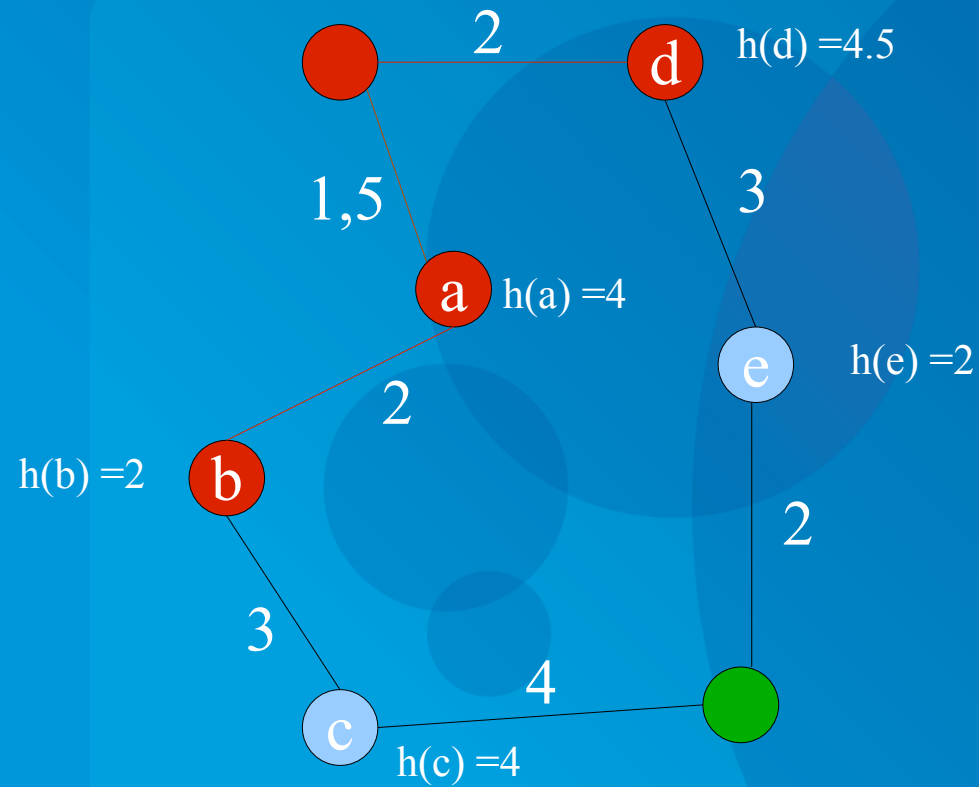
~~$$f(b) = 3.5 + 2 = 5.5$$~~

~~$$f(d) = 2 + 4.5 = 6.5$$~~

$$f(c) = 6.5 + 4 = 10.5$$

$$f(d) = 2 + 4.5 = 6.5$$

Pathfinding – A*



~~$f(a) = 1.5 + 4 = 5.5$~~

~~$f(d) = 2 + 4.5 = 6.5$~~

~~$f(b) = 3.5 + 2 = 5.5$~~

~~$f(d) = 2 + 4.5 = 6.5$~~

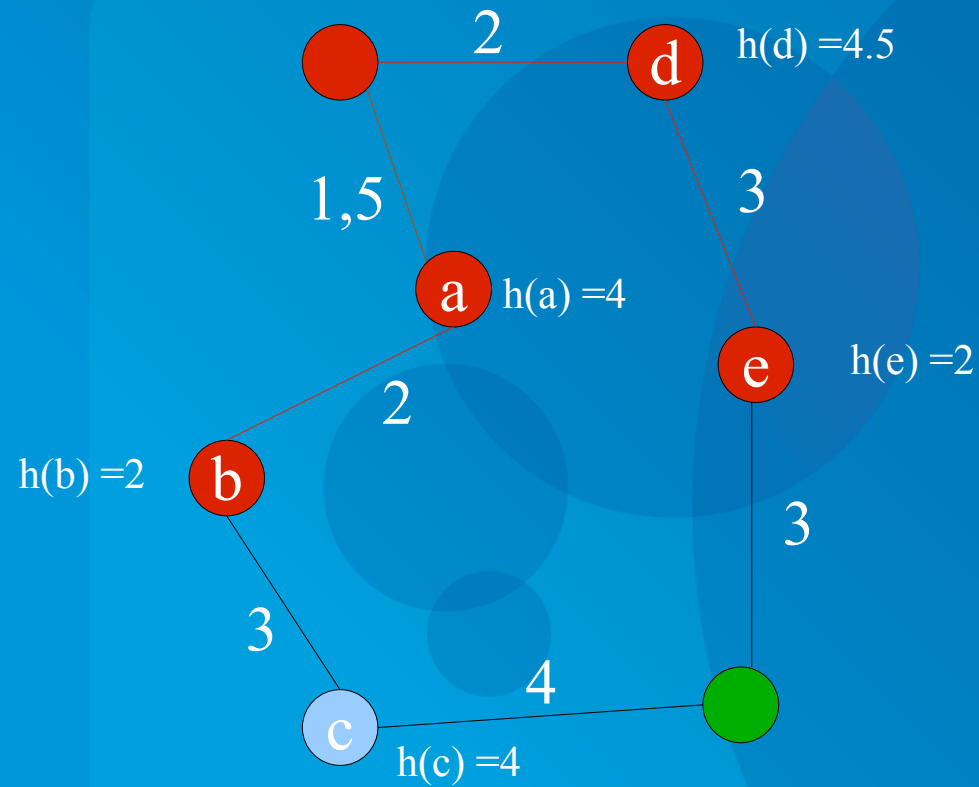
~~$f(c) = 6.5 + 4 = 10.5$~~

~~$f(d) = 2 + 4.5 = 6.5$~~

$f(c) = 6.5 + 4 = 10.5$

$f(e) = 5 + 2 = 7$

Pathfinding – A*



~~$$f(a) = 1.5 + 4 = 5.5$$~~

~~$$f(d) = 2 + 4.5 = 6.5$$~~

~~$$f(b) = 3.5 + 2 = 5.5$$~~

~~$$f(d) = 2 + 4.5 = 6.5$$~~

~~$$f(c) = 6.5 + 4 = 10.5$$~~

~~$$f(d) = 2 + 4.5 = 6.5$$~~

~~$$f(c) = 6.5 + 4 = 10.5$$~~

~~$$f(e) = 5 + 2 = 7$$~~

$$f(c) = 6.5 + 4 = 10.5$$

$$f(G) = 8 + 0 = 8$$

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 = $g(\text{current}) + \text{movementcost}(\text{current}, \text{neighbor})$

 if neighbor in OPEN and cost less than $g(\text{neighbor})$:

 remove neighbor from OPEN, because new path is better

 if neighbor in CLOSED and cost less than $g(\text{neighbor})$: // if we have inadmissible heuristics.

 remove neighbor from CLOSED

 if neighbor not in OPEN and neighbor not in CLOSED:

 set $g(\text{neighbor})$ to cost

 add neighbor to OPEN

 set priority queue rank to $g(\text{neighbor}) + h(\text{neighbor})$

 set neighbor's parent to current

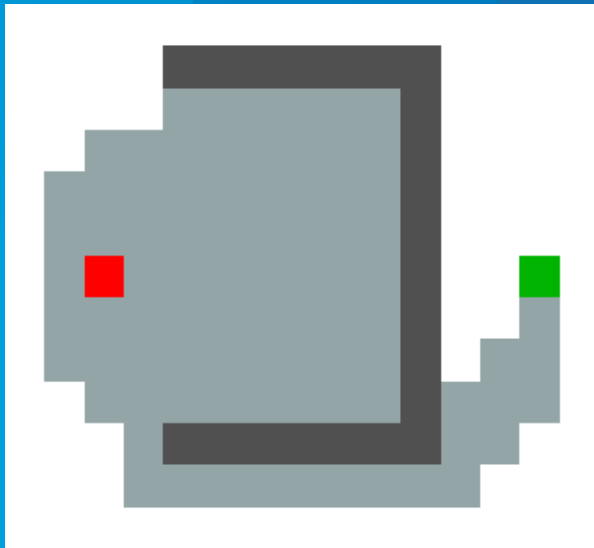
reconstruct reverse path from goal to start by following parent pointers

Pathfinding – A*

```
f = priority queue element {node index, cost}
F = priority queue containing initial f(0,0)
G = g cost set initialized to zero
P,S = pending and shortest nullified edge sets
n = closest node index
E = node adjacency list
while F not empty do
  n ← F.Extract()
  S[n] ← P[n]
  if n is goal then return SUCCESS
  foreach edge e in E[n] do
    h ← heuristic(e.to, goal)
    g ← G[n] + e.cost
    f ← {e.to, g+h}
    if not in P or g < G[e.to] and not in S then
      F.Insert(f)
      G[e.to] ← g
      P[e.to] ← e
return FAILURE
```

Pathfinding – A*

- Problems with A*:
 - Can be very time-consuming if heuristic is bad.
 - Consumes a lot of memory.



Pathfinding – A*

- 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

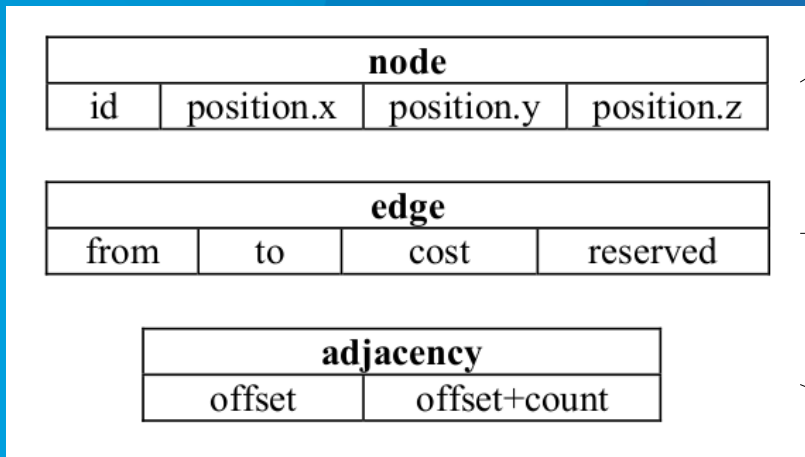
- “*GPU Accelerated Pathfinding*” Avi Bleiweiss, NVIDIA Corporation, Eurographics 2008
- Goal: “to exploit general data parallelism in performing global navigation planning for many thousands of agents.”
- Comparison with an equivalent CPU implementation.
- Memory structure tweaked.

A* on a GPU

1		3.2		5.4		8.0
2		2.1		2.8		7.9
3		3.3		1.7		8.4

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.
 - $\{\{1,3\},\{2,5\},\dots\}$
 - A list of costs from the start position G, initialized to zero.
 - $\{\{0\},\{0\},\dots\}$
 - A list of combined costs from start to goal.
 - $\{\{3\},\{6\},\dots\}$
 - 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

Graph	Nodes	Edges	Agents	Blocks
G0	8	24	64	1
G1	32	178	1024	8
G2	64	302	4096	32
G3	129	672	16641	131
G4	245	1362	60025	469
G5	340	2150	115600	904

- 115600(!) agents but only 340 nodes?
- 340 nodes \sim 18x18 grid! Not exactly huge.

A* on a GPU

Threads per Block	128
Registers per Block	2560
Warps per Block	4
Threads per Multiprocessor	384
Thread Blocks per Multiprocessor	3
Thread Blocks per GPU	48

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

Graph	Roadmap	Working Set	Total	Launches
G0	0.576	0.021	0.021	1
G1	3.616	1.319	1.322	1
G2	6.368	10.518	10.519	1
G3	13.848	86.001	86.001	1
G4	27.672	588.726	588.726	2
G5	42.560	1573.086	1573.086	3

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 Dijkstra algorithm (A* with heuristics zero):

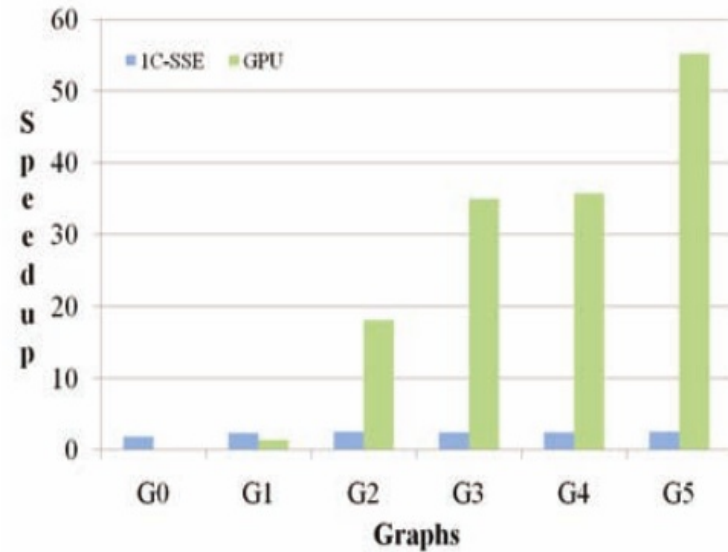
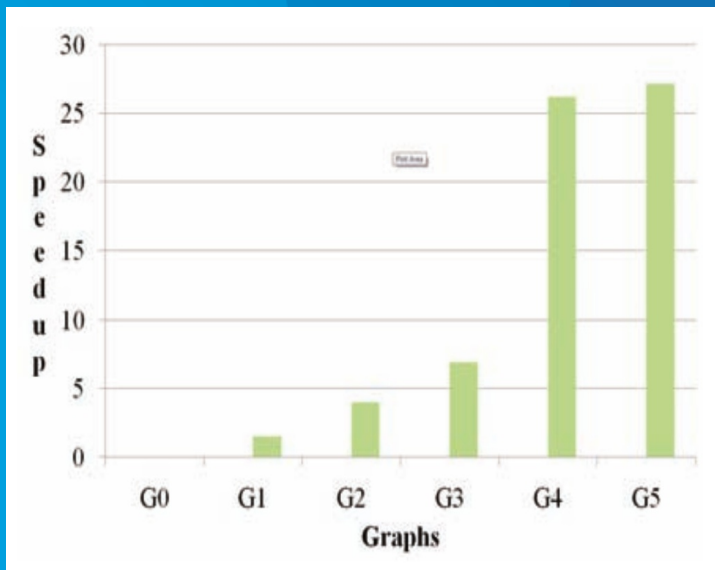


Figure 12: Performance of GPU running CUDA A* search algorithm using Euclidian heuristic, compared to CPU plain optimized C++ code and to hand-compiler tuned SIMD intrinsics (SSE) implementation.

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!