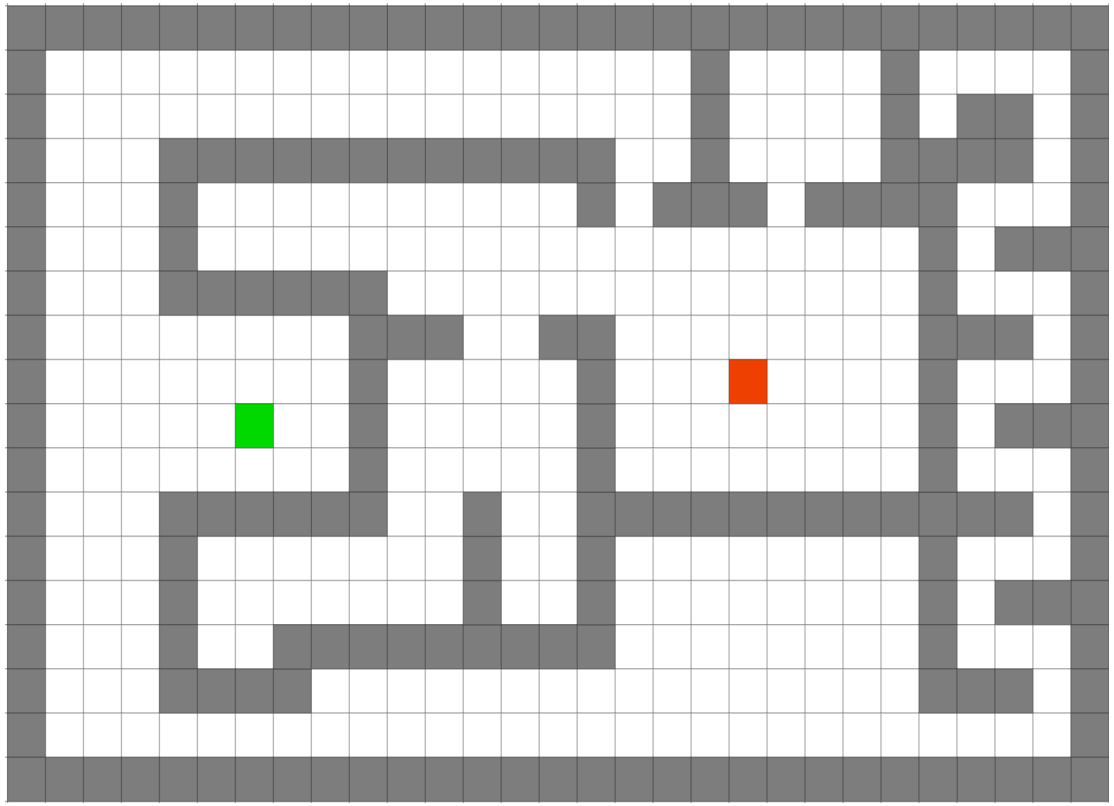
A* Path-Finding with MPI

Thornton Haag-Wolf
University at Buffalo, The State University of New York
CSE 633
Spring 2014

Problem Description

- We want to find the shortest path between two points on a connected graph

- We have a known start point, end point, and know the cost to move into any adjacent node from every point

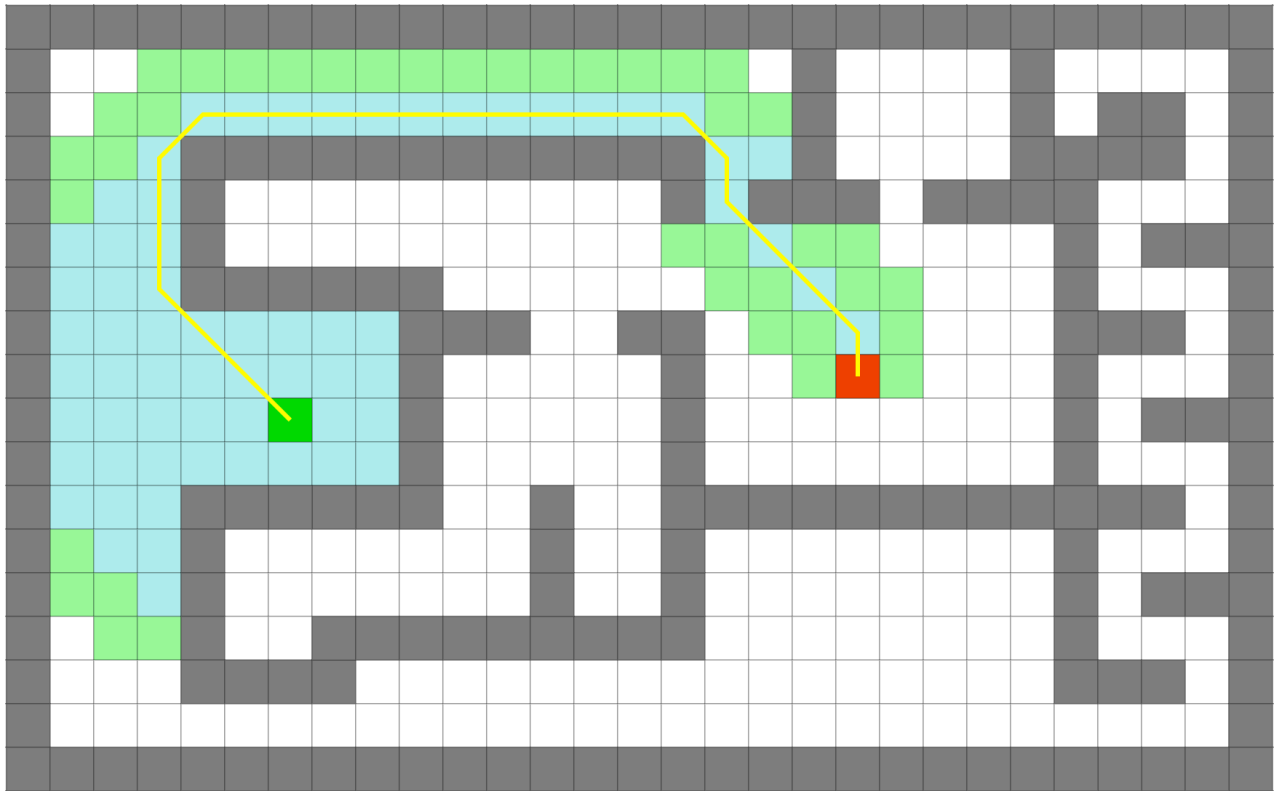


A* Algorithm

- Similar to other search algorithms like Dykstra's or breadth-first
- Uses a heuristic function to make decisions on traversal of graph
- The total cost from start to finish is modeled by F(n) = G(n) + H(n)
  - F(n): the total cost
  - G(n): cost to move into node we are on
  - H(n): Heuristic function estimating remaining cost to goal

A* Algorithm

- When(if) a path is found from start to finish we want this F cost to be minimal
- This is where the heuristic function is used
- By estimating the remaining distance to the goal we can make an educated guess on where to check next



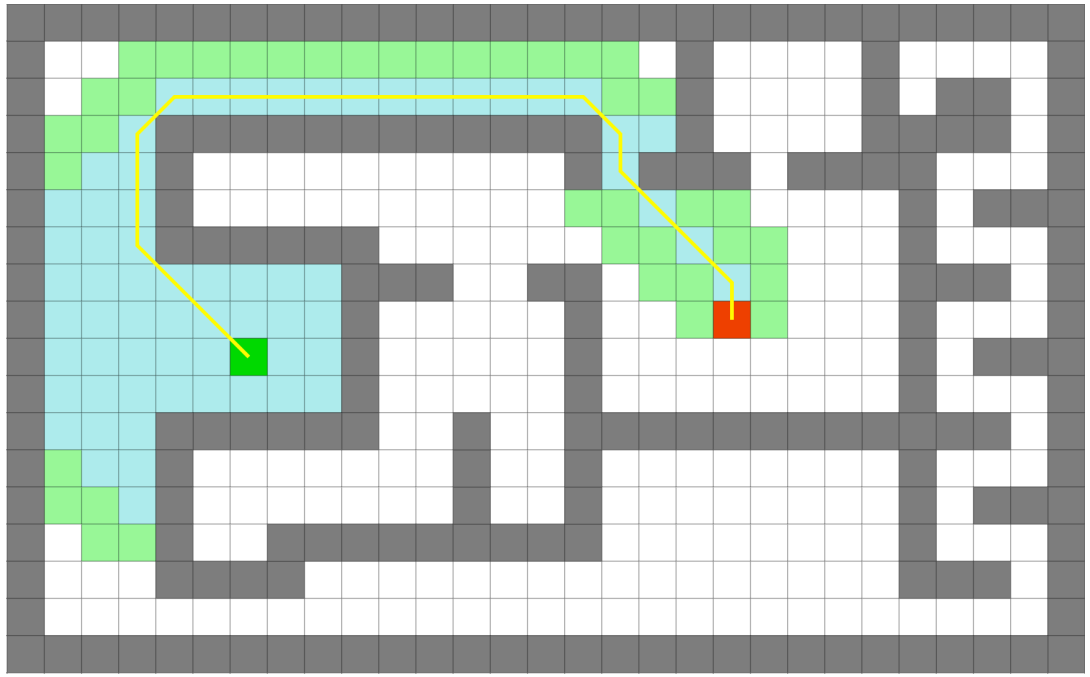
Optimal path found

A* Algorithm - Heuristic Function

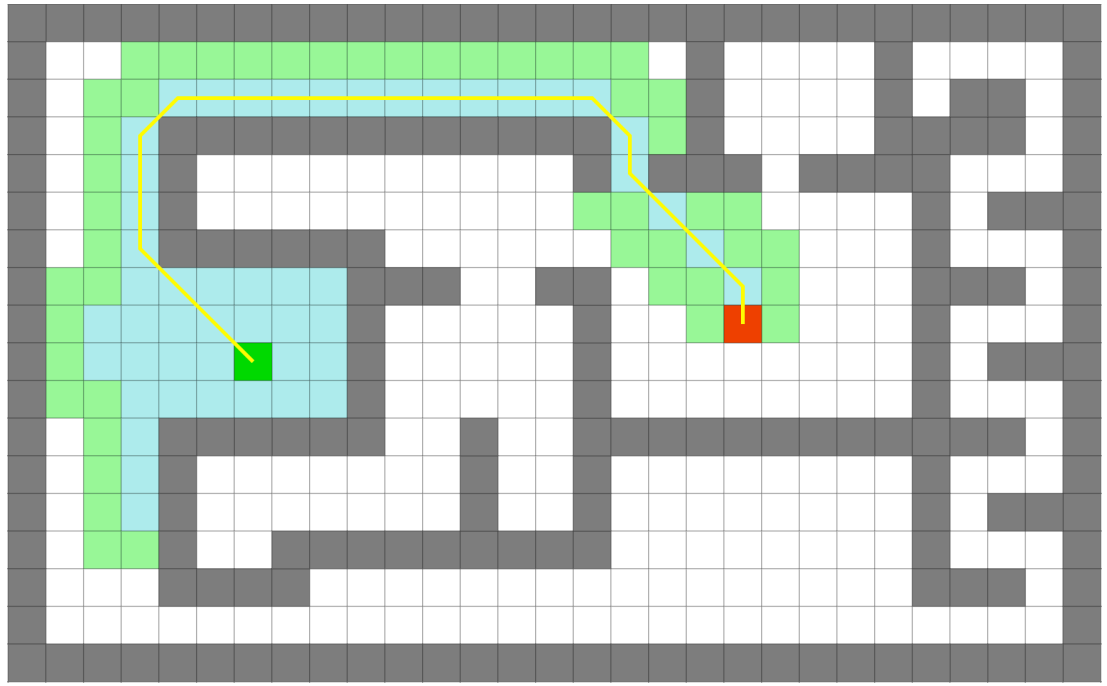
- There are a wide variety of metrics to consider for the heuristic
  - Optimality of solution
  - Completeness of solution set
  - Accuracy
  - Running time
- Must consider what we need and tradeoffs associated
- Common choices
  - Manhattan: x,y vector distance to goal
  - Euclidean : straight line distance to goal
  - Chebyshev: max of either x or y distance to goal

A* Algorithm - Heuristic Function

- Heuristic function must not overestimate remaining distance to be considered admissible

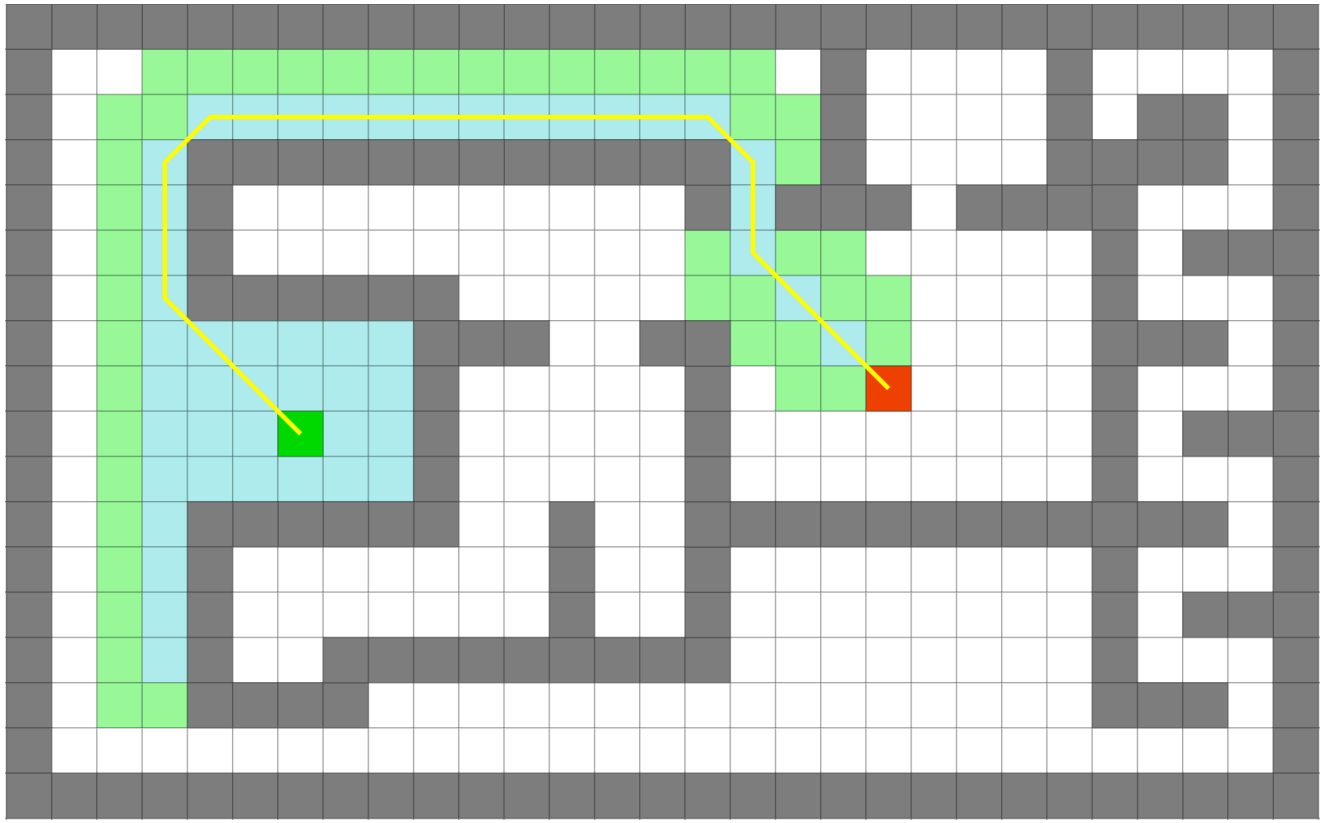


A* using Manhattan



A* using Euclidean

A* Algorithm – Heuristic Function



A* using Chebyshev

- All three heuristics found same optimal solution but did not search the same set of nodes
- Blue nodes represent closed set.
- Green represent open set.
- White nodes were not searched.

A* Algorithm - Process

- To actually perform the search we start with two sets
  - List of open nodes to be searched
  - List of closed nodes that have been searched
- Add starting node to open list initially
  - Expand open set to include each neighbor of starting node
  - Put starting node on closed set
  - Calculate the F costs of those open neighbor nodes
- During the next round we chose the neighbor node with the lowest calculated F cost to expand next and repeat the algorithm, until either a solution is found or we exhaust the set of available nodes

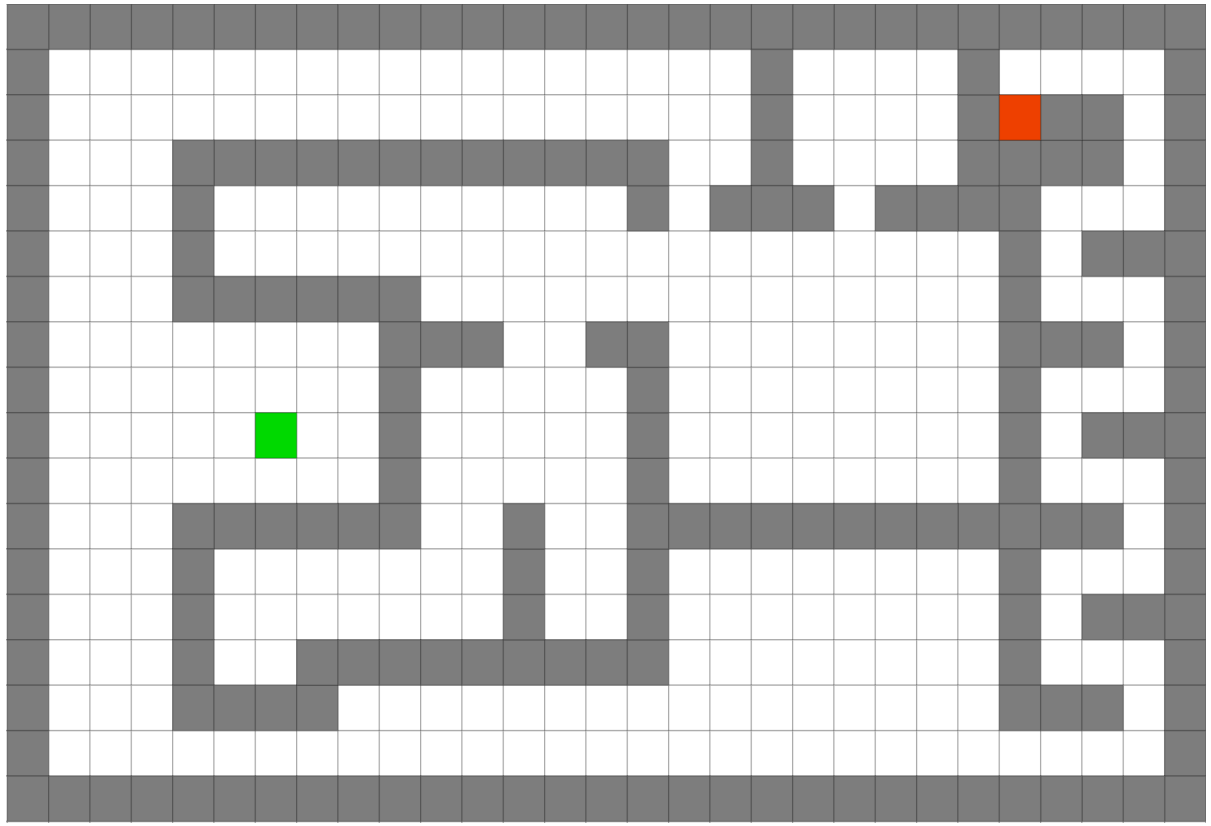
A* Algorithm – Full Pseudo Code

```
initialize the open list
initialize the closed list
put the starting node on the open list (you can leave its f at zero)
while the open list is not empty
    find the node with the least f on the open list, call it "q"
    pop q off the open list
    generate q's 8 successors and set their parents to q
    for each successor
                if successor is the goal, stop the search
        successor.g = q.g + distance between successor and q
        successor.h = distance from goal to successor
        successor.f = successor.g + successor.h

        if a node with the same position as successor is in the OPEN list \
            which has a lower f than successor, skip this successor
        if a node with the same position as successor is in the CLOSED list \
            which has a lower f than successor, skip this successor
        otherwise, add the node to the open list
    end
    push q on the closed list
end
```

A* Algorithm – Common Assumptions

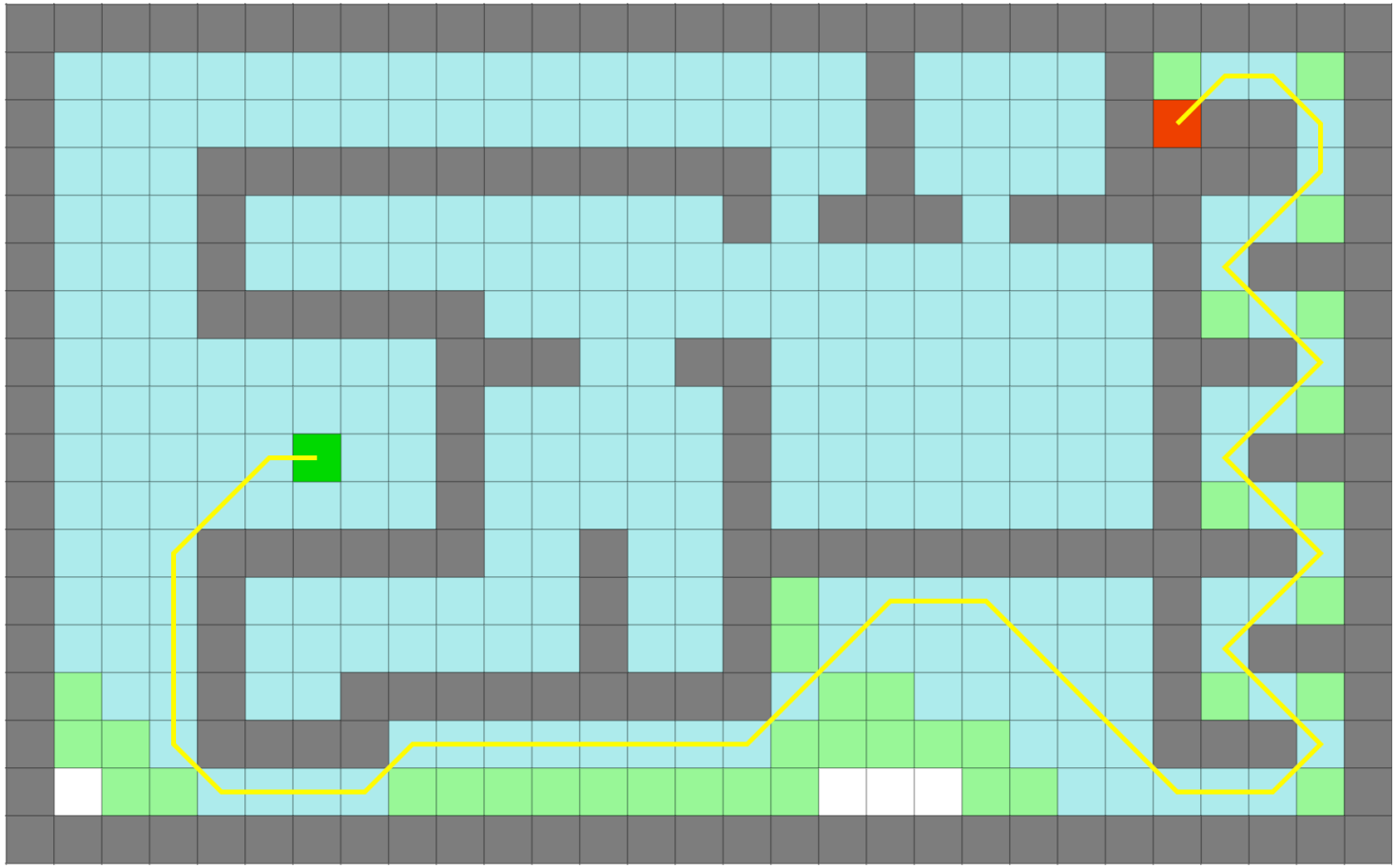
- The graph to be searched is usually a grid with uniform cost to move from node to node
- Optimal data structures like priority queue are available
- Certain nodes act as obstacles that cannot be traversed easily if at all
  - Examples: wall, bodies of water



Our go to visual example with walls and relocated end point

A* Algorithm – Running Time

- Under ideal circumstances the amount of nodes searched relative to the graph size is small
- At worst every node must be searched to find solution as well as when the solution doesn't exist



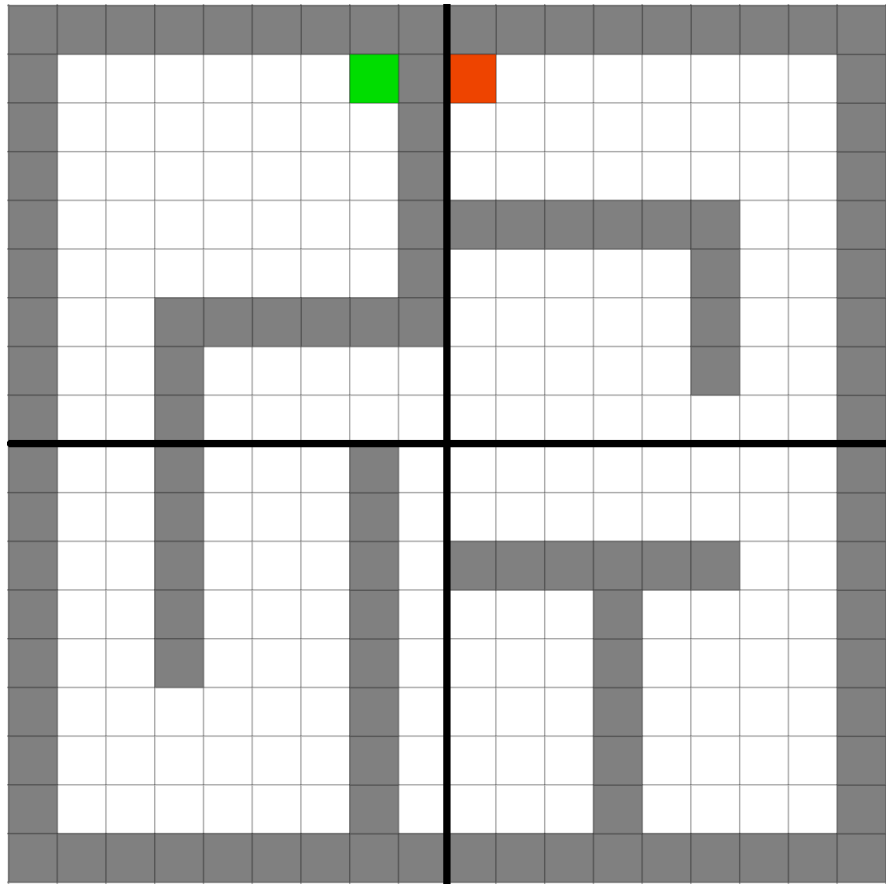
Near worst case performance. Also note solution found is not optimal!!!

A* Algorithm – Parallelizing

- Need to find a method to properly utilize parallel processing environment
- There are lots of variations on the A* algorithm that introduce a parallelizable component
- We are going to look at the hierarchical breakdown method and show that it allows for a parallel prefix style solution

Hierarchical Breakdown A* (HBA)

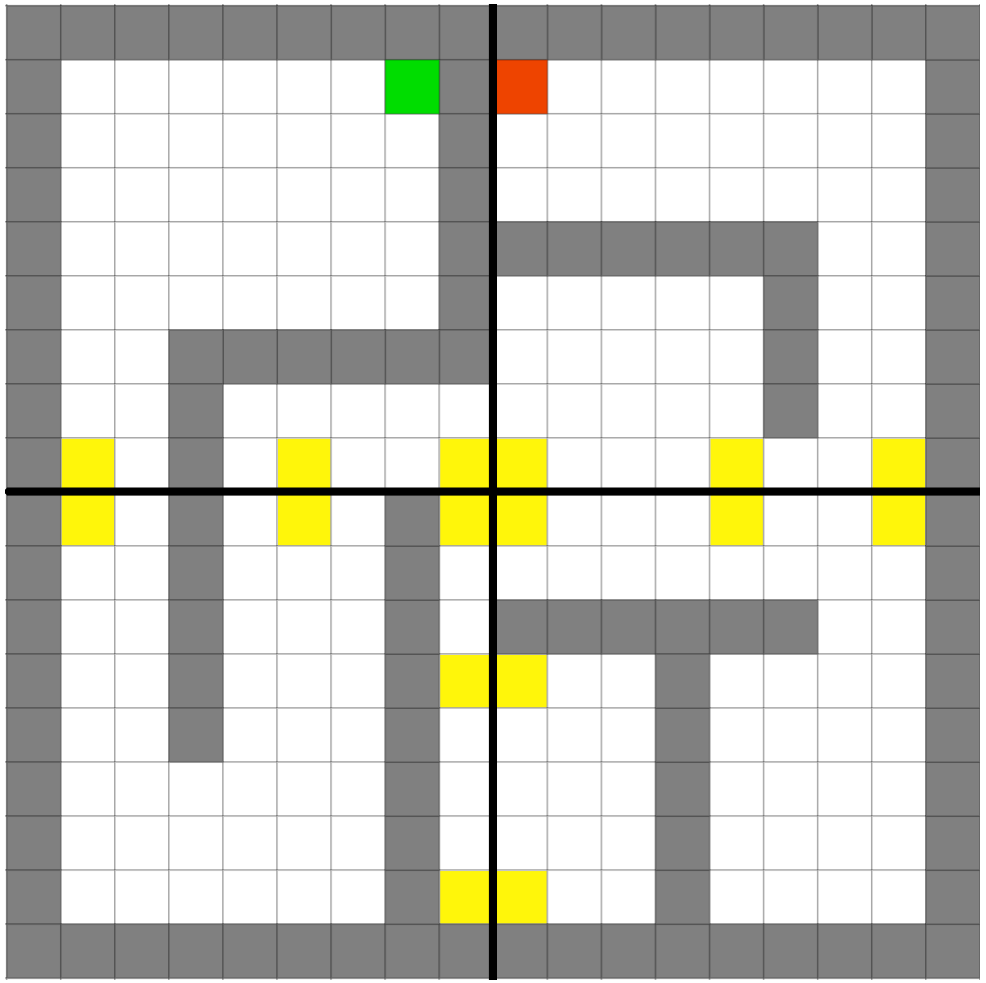
- Adds a level of abstraction over graph
- Splits graph into clusters
- Algorithm finds entry and exit points between clusters



Our base map before HBA divided into four clusters

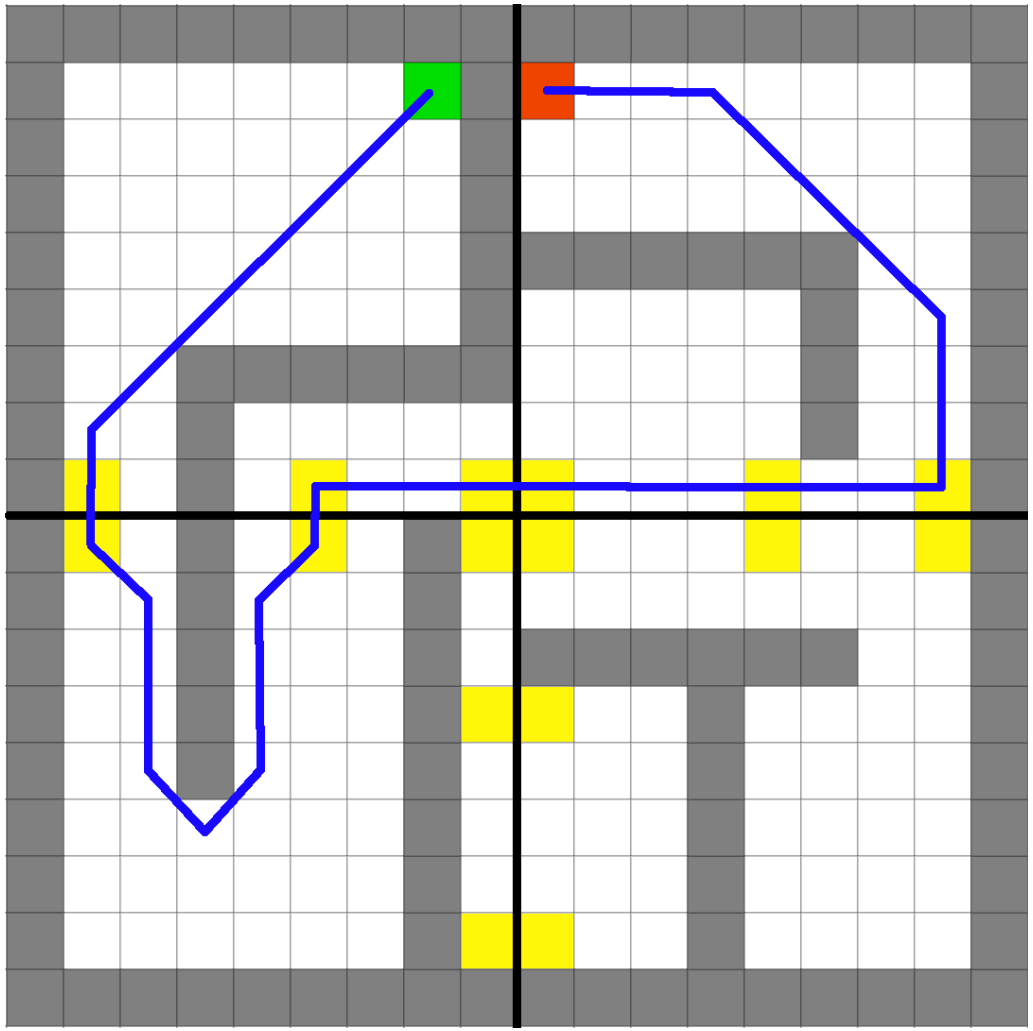
Hierarchical Breakdown A*

- Now use algorithm to find points that connect the clusters
- These points are marked in yellow



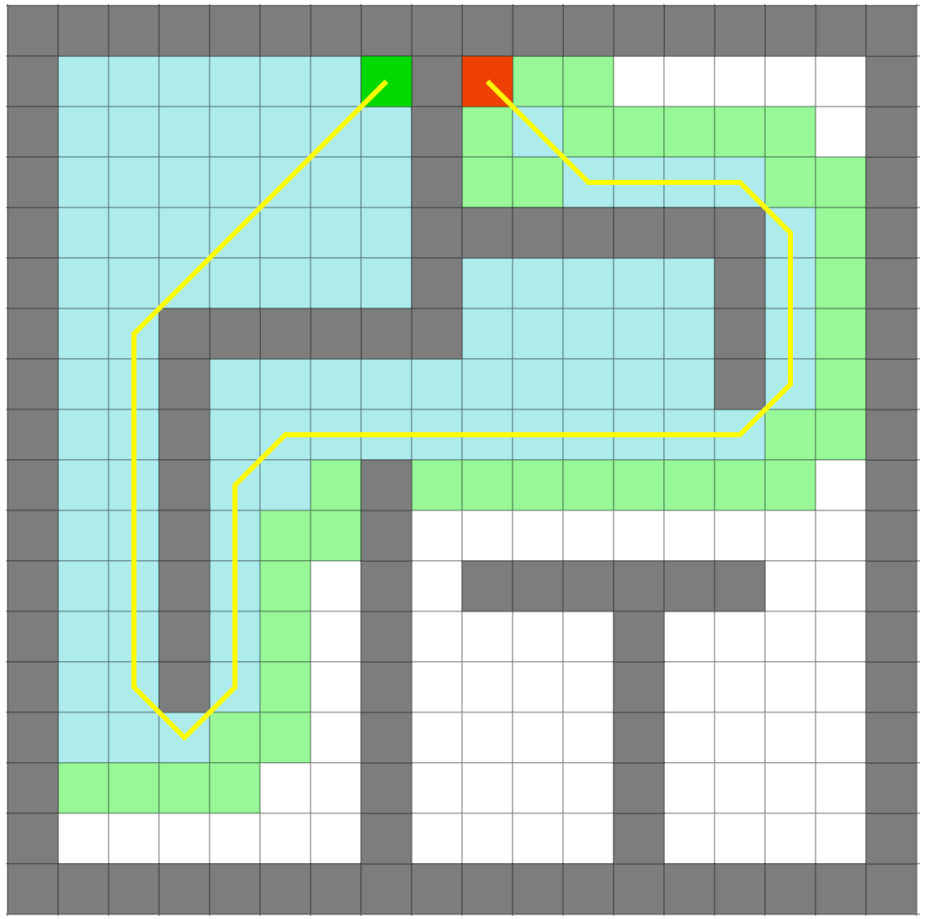
Hierarchical Breakdown A*

- In each cluster use A* to find the distances between each yellow node
- We know the cluster that contains our start and end nodes
- Use local A* to move from start node to HBA paths
- Travel along optimal HBA paths until end node cluster reached
- Use local A* to travel to end



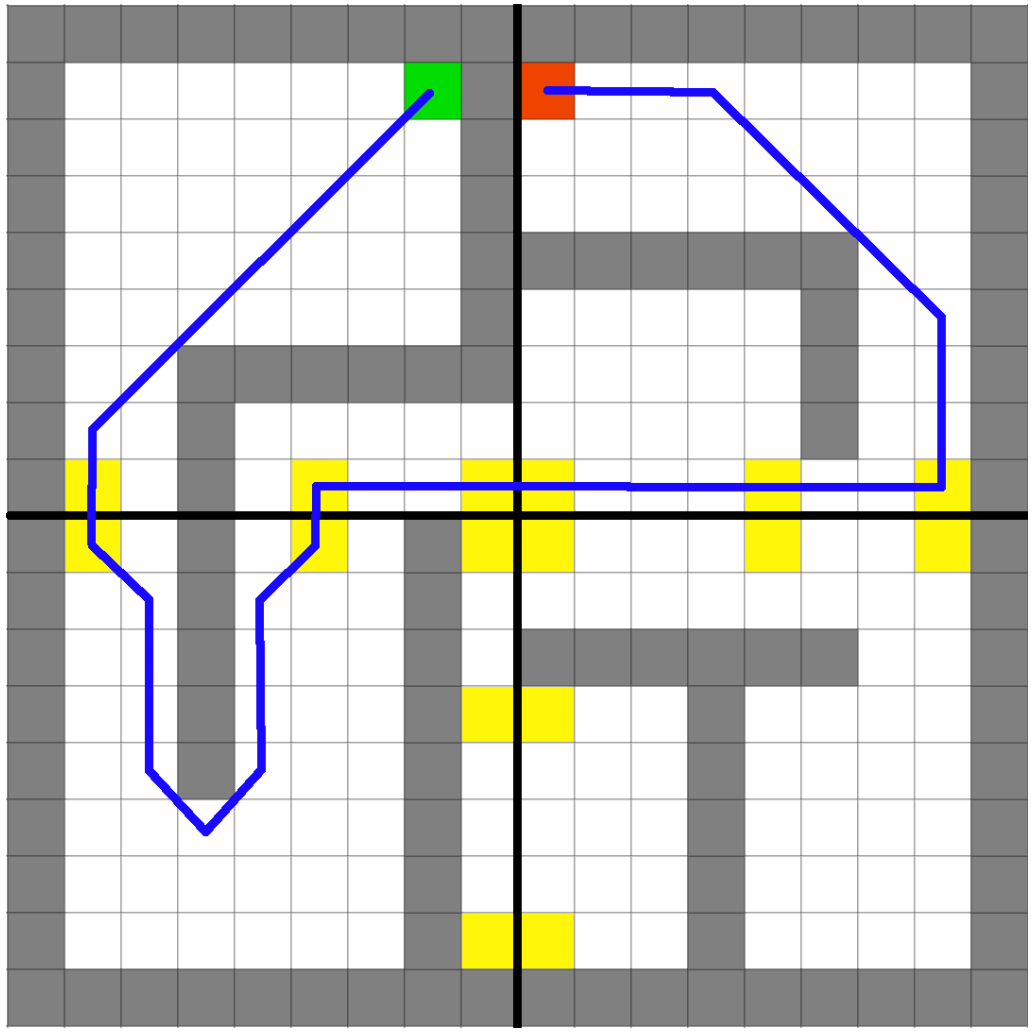
Hierarchical Breakdown A*

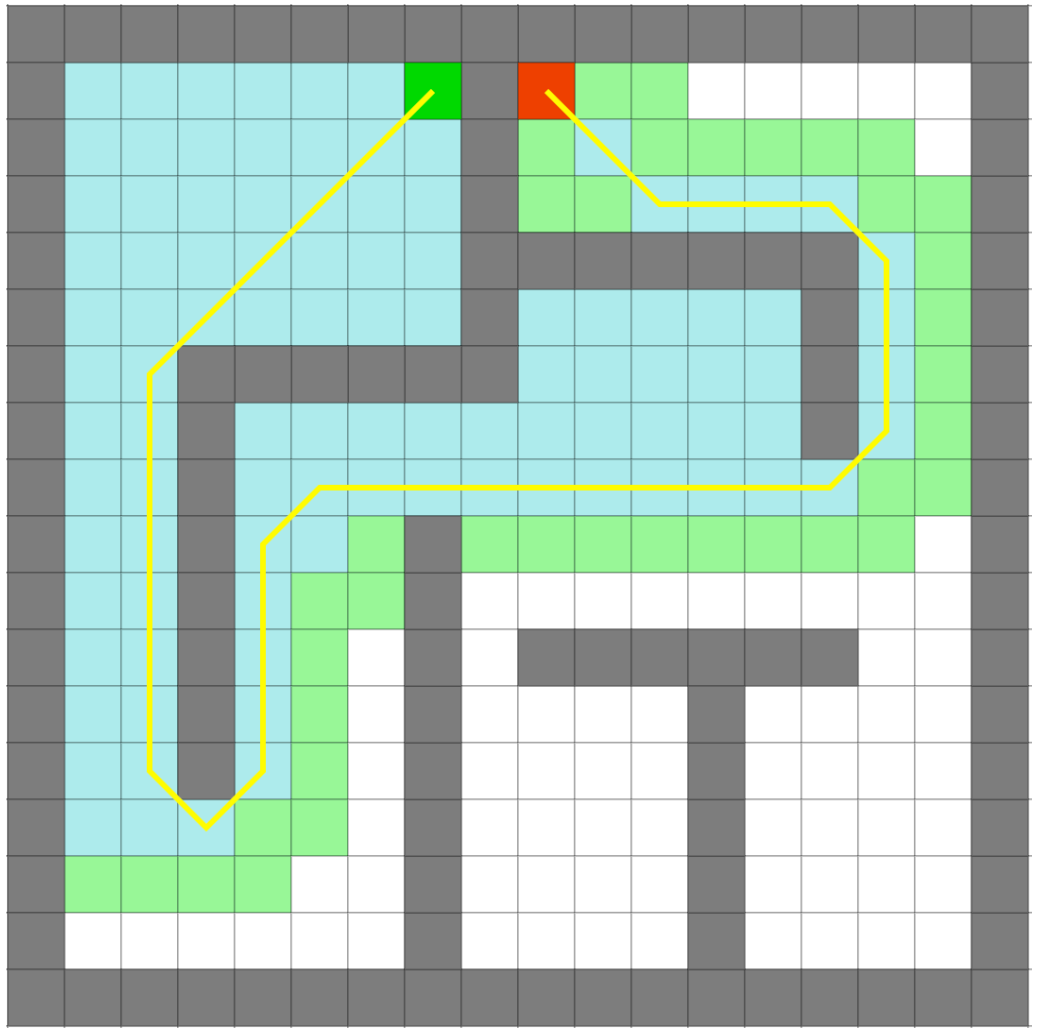
- Path not guaranteed optimal, but within 1%
- Trade off of optimal solution for computation time and smaller memory footprint
- Useful as a preprocessing step that can be reused for multiple path searches on same map



Path taken by base A* using Chebyshev heuristic. Compare to HBA path.

Hierarchical Breakdown A*





Parallelizing HBA

- We break down the graph into clusters during HBA
- These clusters can be given to our set of PEs
- Compute entry points and the paths between them locally on each PE
- We know cluster location of goal node
- Each cluster calculates how to reach desired goal cluster using HBA paths
- PEs communicate these sub solutions to each other until all nodes know a solution (our parallel prefix step)
- Use MPI to accomplish message passing and internode communication

Experimental Setup

- Multiple map types
  - Empty rooms
  - Spiral
  - Maze-like
  - Open area with large obstacles
- Test on various sizes
  - 512x512
  - 1024x1024
  - 4096x4096
  - 8192x8192

Experimental Setup

- Change cluster sizes for each map
  - 16x16
  - 32x32
  - 64x64
- Vary number of PEs used
  - 1,2,4,8,16,etc

Examples of “Professional” Maps Used

Open Area Map

- Map used as a test sample
- Will run tests on this first including changes to
  - Overall map size
  - # of PEs used

Results – Open Map

Map Size

Nodes

	512	1024	4096	8192
2	60.345	279.34	1198.323	12567.409
4	29.876	182.456	637.127	8404.23
8	17.536	90.345	345.924	6290.234
16	12.341	34.092	268.345	4093.104
32	23.567	16.189	210.723	3913.863
64	30.985	15.982	312.9	4323.903
128	31.456	16.934	436.067	4923.5

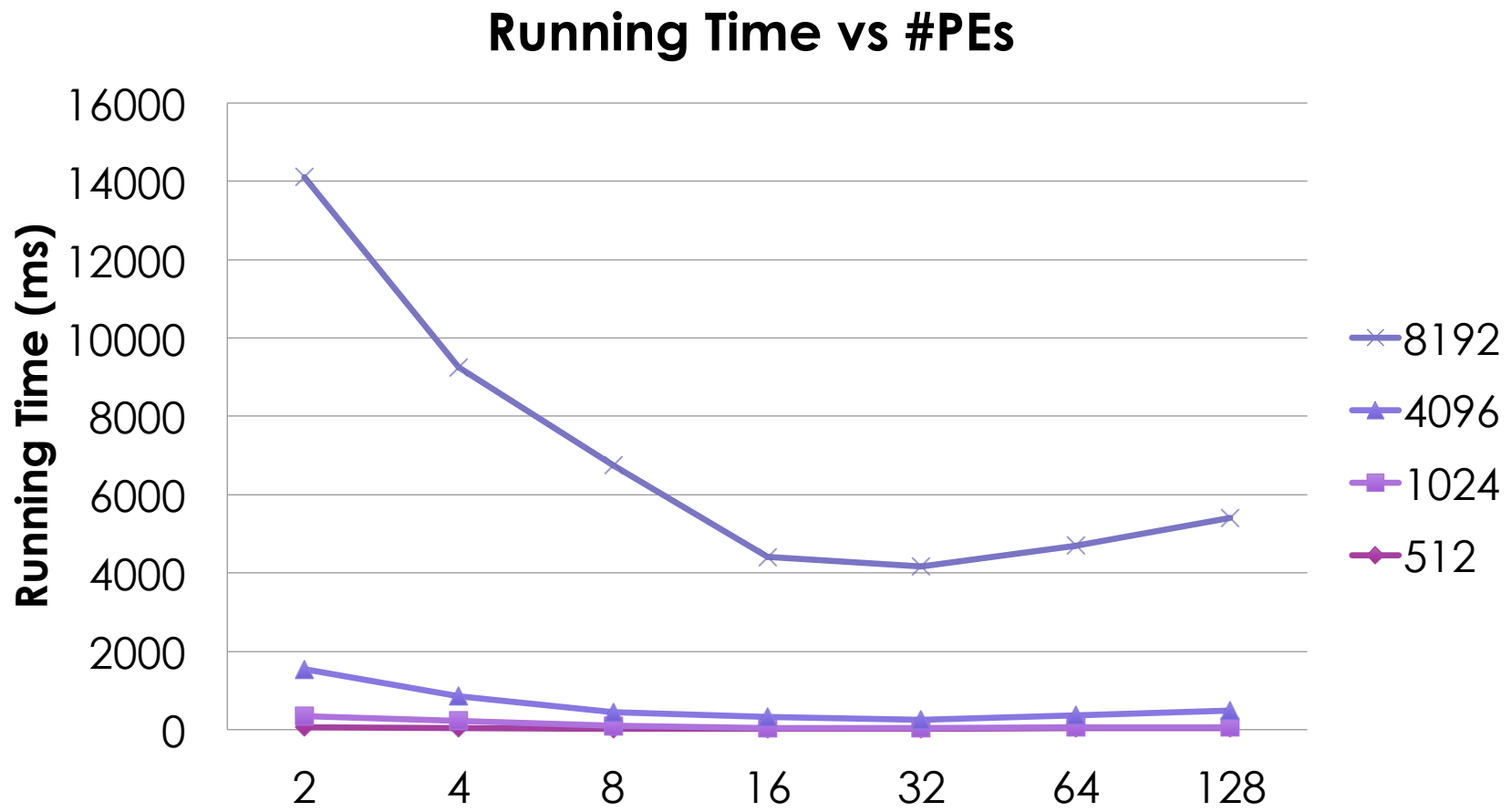
All results using 32x32 HBA cluster sizing

Results are running time in milliseconds

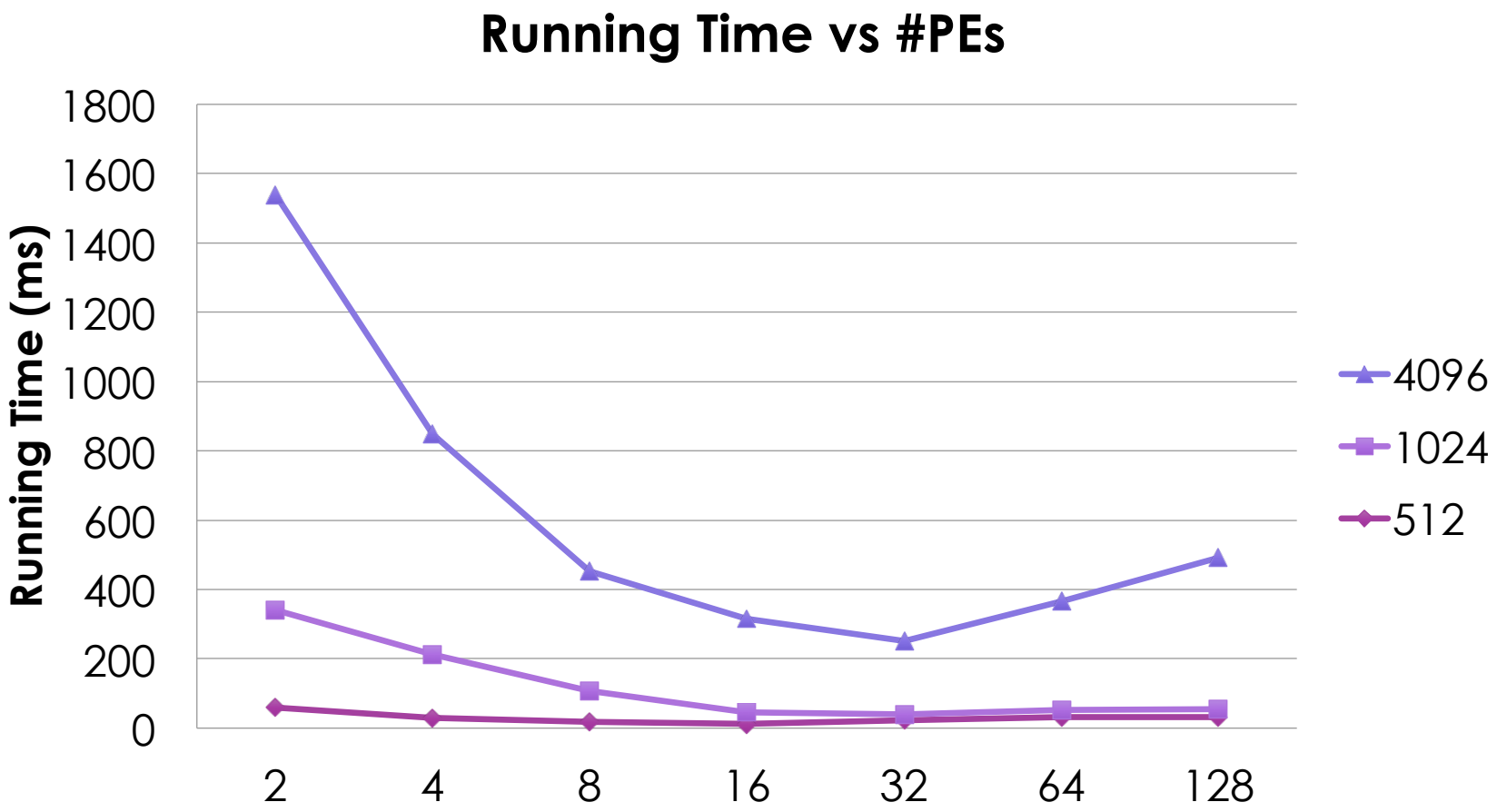
Experiment includes time to set up HBA clusters
and navigate a path spanning the map

Realistically the time spent making HBA clusters can be reused for repeated path calculations resulting in overall dramatically reduced times

Results – Open Map



Results – Open Map

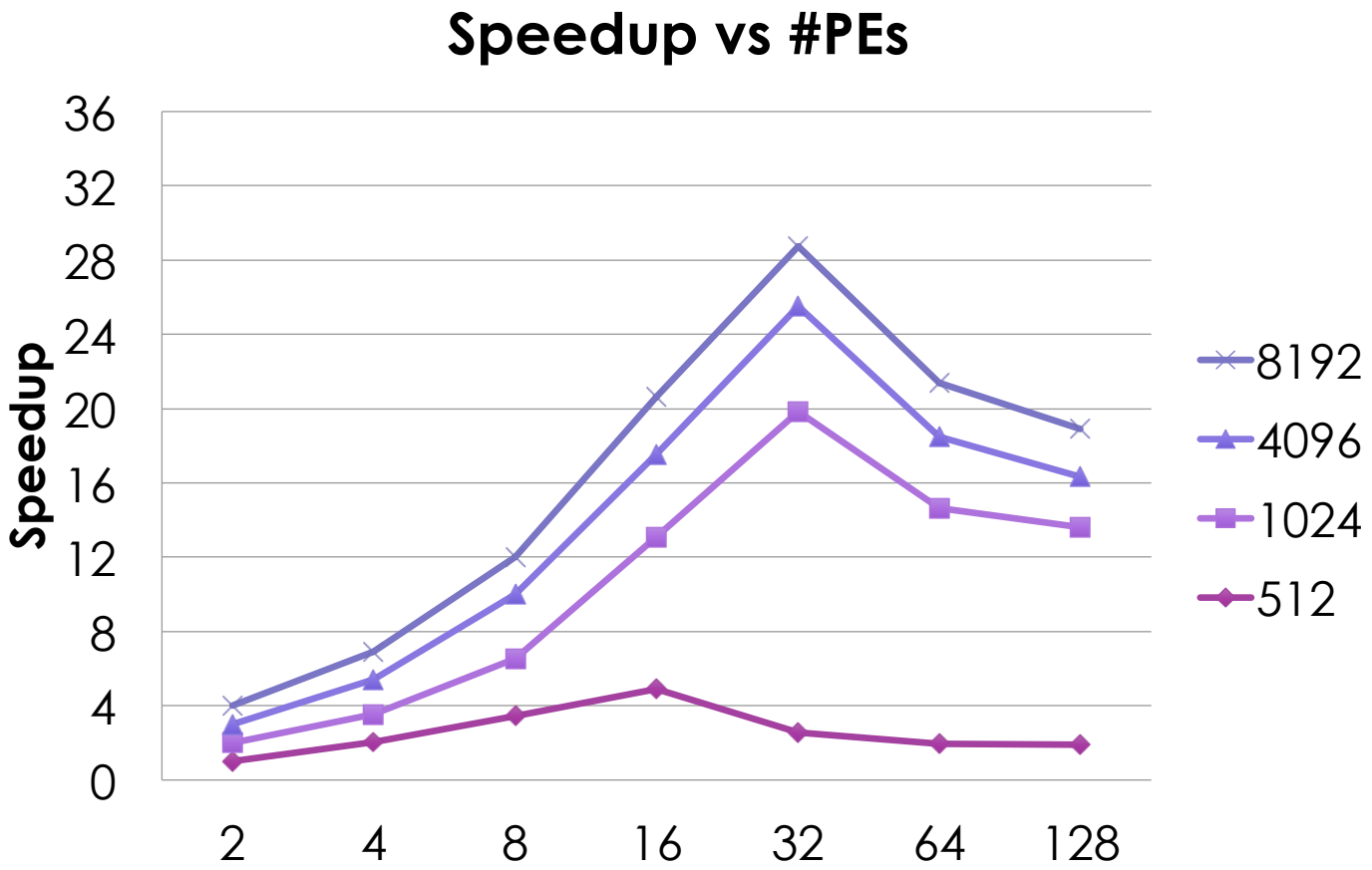


Results – Open Map Speedup

Map Size

#PEs	512	1024	4096	8192
2	1	1	1	1
4	2.019848708	1.530999255	1.880822819	1.495367095
8	3.441206661	3.091925397	3.464122177	1.997923925
16	4.889798234	8.193711135	4.465605843	3.070385947
32	2.560571986	17.25492618	5.686721431	3.210998699
64	1.947555269	12.70766991	3.829731544	2.906496515
128	1.918393947	11.67126264	2.748024959	2.552535595

Results – Open Map Speedup



Conclusions

- HBA allows for an easy implementation of the A* search algorithm
- There are many solutions to parallelizing that can be explored though
- Thorough experiments take time to conduct
  - Did not have enough time/resources to generate data for more than one map type
- Running times increase drastically as map size increase
  - More efficient data structures needed for maintaining node lists
  - Message passing large datasets in costly
- Speedups follow a similar trend though
  - For this setup maximum speedup was achieved with using 32 nodes for computation

References

- A. Botea et al. “Near Optimal Hierarchical Path-Finding”, *Journal of Game Development*, Vol. 1, pp.7-28, 2004.

- H. Cao et al. “OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis”, *Proceedings of the 2009 WRI World Congress on Software Engineering – Volume 01*, pp. 61-66, 2009.

- BLEIWEISS, A. 2008. GPU Accelerated Pathfinding. In Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, 66–73.

References

- Visual examples created using PathFinding.js.
  - http://qiao.github.io/PathFinding.js/visual/