Hierarchical Path-Finding

Addie Audette, Bug Lee, Annorah Lewis, Luke Marks

December 14, 2022

Table of contents

Path finding

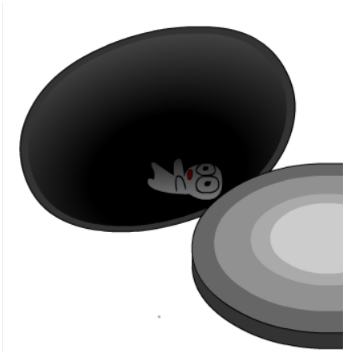
You are given

- Starting location S
- Destination location D

Path finding

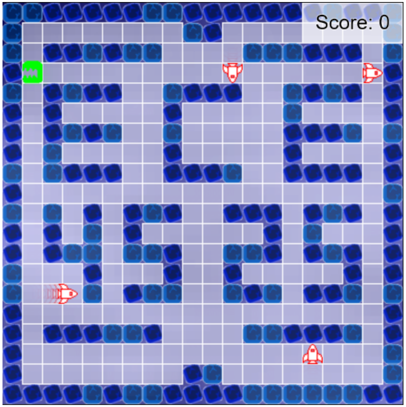
You want to avoid path from S to D that are

- Impossible
- Dangerous
- Unnecessary



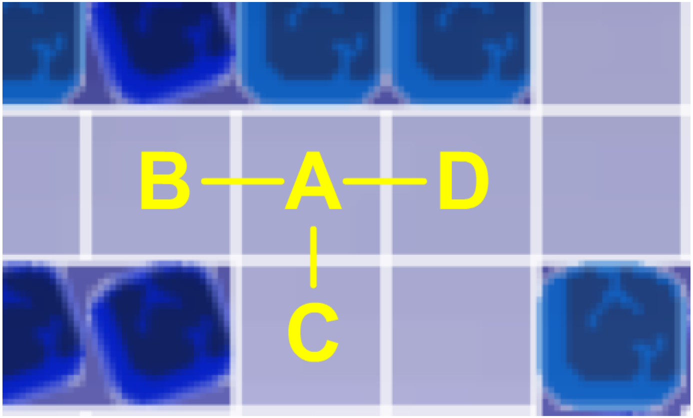
Viewing world with grid graph

Imagine world is made of grid, like Mindcraft...



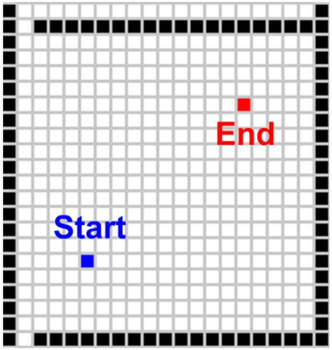
Viewing world with grid graph

- Each cell is a vertex and has an edge to adjacent cells.
- No edges between cells and walls/obstacles
- So, each vertex can have up to degree 4 and edges are undirected with weight 1.



Back to path finding

Now, using the grid graph, how can we find the shortest path from S to D?



Dijkstra

- Dijkstra algorithm finds the shortest path to all the vertices from source s, given that the graph $G = (V, E)$ contains only non-negative weight for all edges[3].
- In other words, it forms the tree that represents the shortest paths to all of the vertices in the graph.

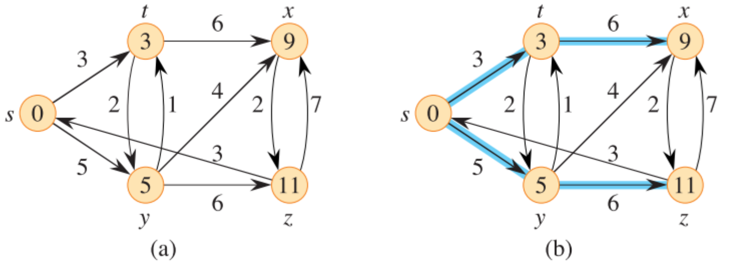


Figure: (a) A weighted, directed graph with source s. (b) The blue edges represent the shortest-path tree rooted at the source s. The figure was taken from *Introduction to Algorithms* by CLRS[3].

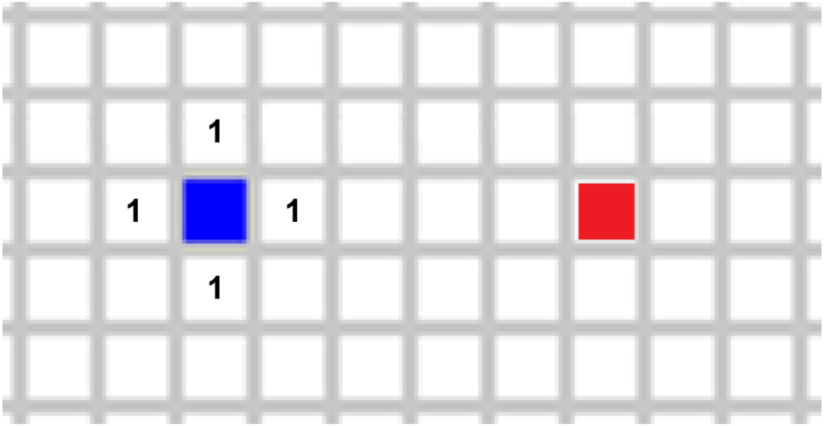
Dijkstra Algorithm

Dijkstra algorithm is a type of greedy algorithm where it makes a locally optimal decision in each step. The following describe the high-level idea:

1. At first, the source vertex only knows the distance to its neighbors and treats other vertices as if they are infinitely far away.
2. Among the univisited vertices, visit the vertex that is closest to the source vertex. That is, add the selected vertex to the shortest path tree and mark visited.
3. Then, updates the distance to the vertices that are now reachable (but still unvisited) by the newly visited vertex.
4. Repeats the step 2-3 until it reached destination or all the vertex get marked visited.

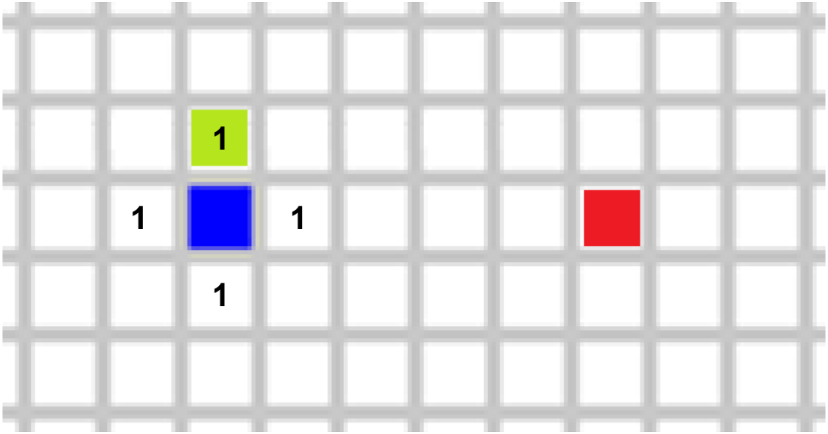
Dijkstra in action

Step 1. Inititally, Dijkstra only know the distance to the neighbors of the source.



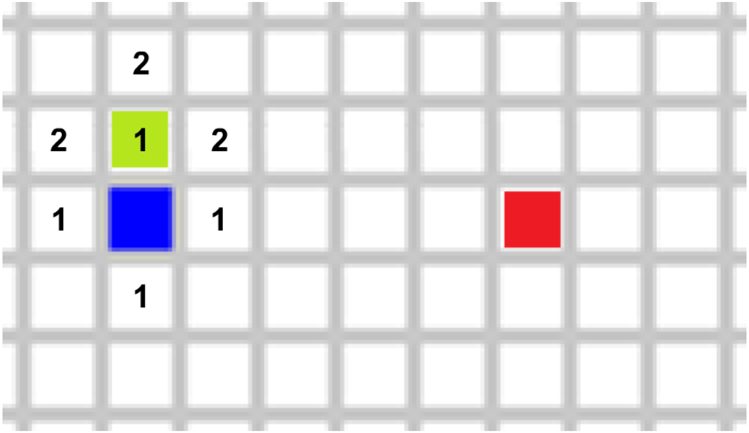
Dijkstra in action

Step 2. Visit the closest vertex from the source. In this case we just picked the upper cell from the 4 closest vertices.



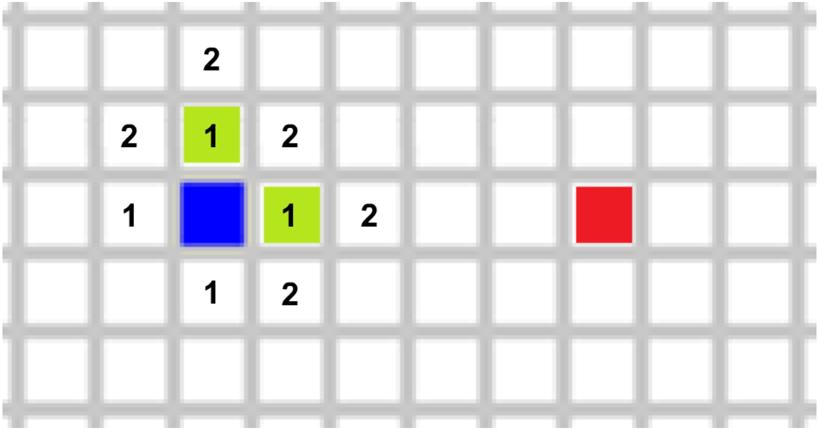
Dijkstra in action

Step 3. Updates the distance to the vertices that are now reachable (but still unvisited) by the newly visited vertex.



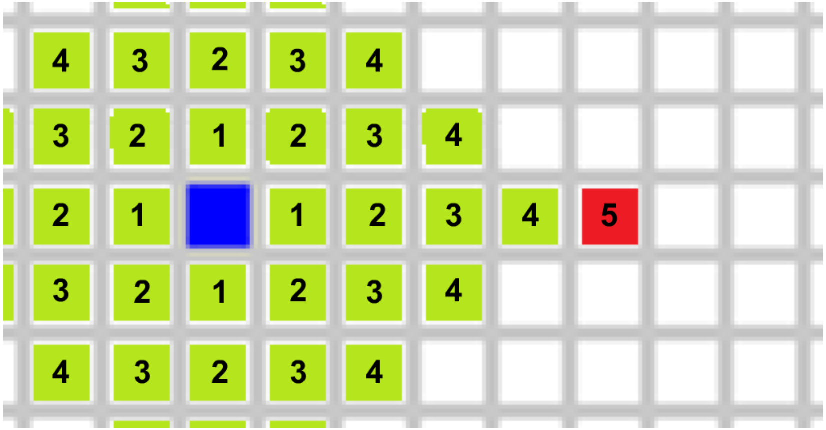
Dijkstra in action

Step 4. Repeats the step 2-3 until it reached destination or all the vertex get marked visited.



Dijkstra in action

So when we are done, we might get something like this:



Shortcoming of Dijkstra

Dijkstra will find the shortest path eventually (proof can be seen at our research paper). However, we can see some shortcomings:

- We were evaluating path that were "obviously" not part of the shortest path.
- Why find shortest path to all vertices when we only need to find one to D?

A*

- "Dijkstra with a twist" [2]
- Dijkstra algorithm blindly selects vertex with minimum distance from S each step. Instead, make a clever guess in each step where the algorithm selects a vertex that is likely part of the shortest path from S to D.[2, 4]

A*: Clever guess?

For A* to work correctly and efficiently, the A* algorithm must guess each step that

- Heuristic:
  Minimize unnecessary computation on finding sub-paths that are obviously not part of the optimal path[4], but also
- Admissiblity:
  Should not ignore the sub-path that can be part of the optimal path[4].

A* Algorithm

The algorithm is almost the same as Dijkstra. Only difference is the step 2:

1. ...
2. Among the univisited vertices, visit the vertex that have the **lowest cost** f to the source vertex.
3. ...
4. ...

Cost?

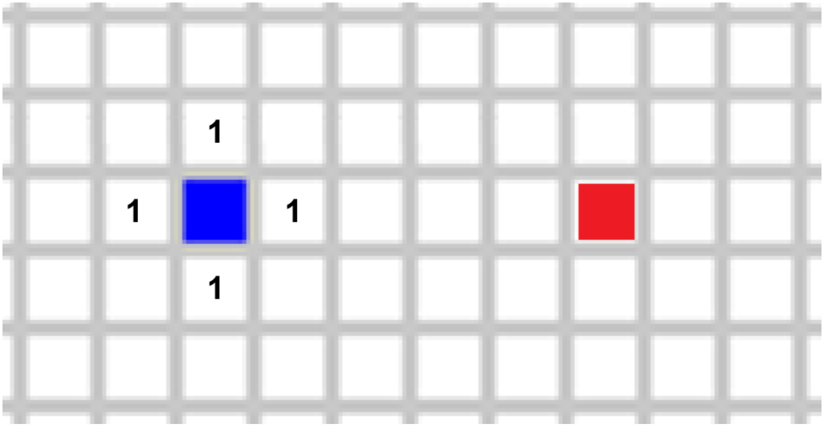
We define the cost of the vertex as a following:

$$f(v) = g(v) + h(v)$$

- $f(v) =$ total cost of the vertex v.
- $g(v) =$ exact distance from source to vertex v.
- $h(v) =$ estimated distance from vertex v to destination. We will use euclidian distance between v and D as our estimation.

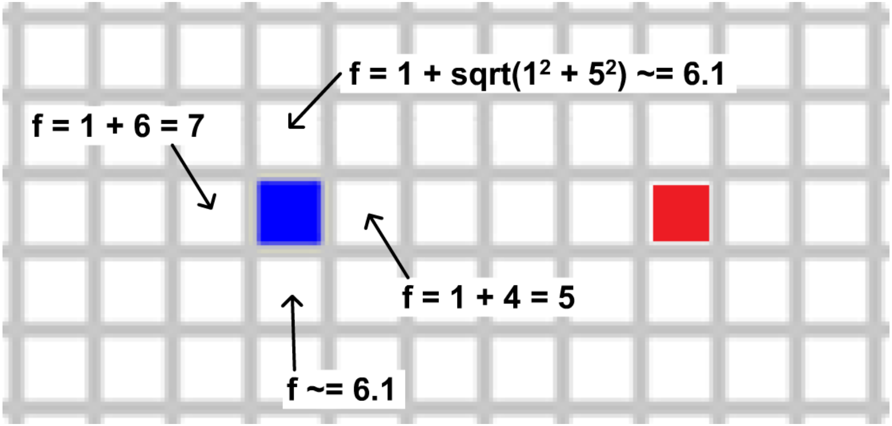
A* in action

Let's revisit the step 2 of Dijkstra.



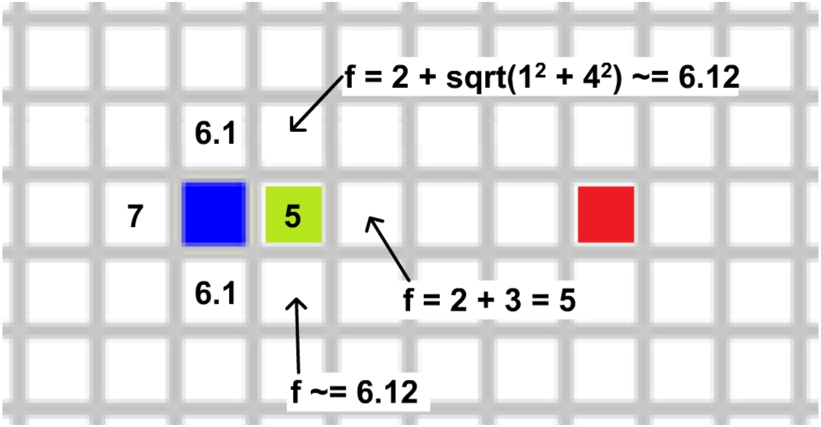
A* in action

Instead of looking for the vertex that are closest to the source, look for the lowest cost *f* as shown below.



A* in action

Clearly, the winner is the right hand side vertex.

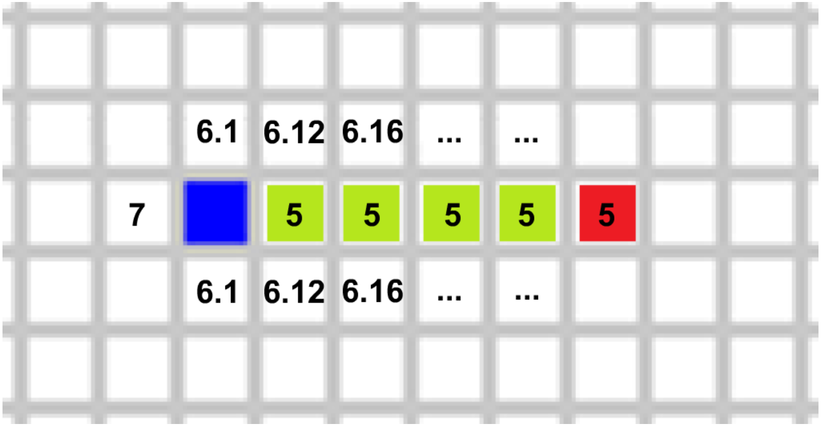


A* in action

This is similar to how human think, where we would unlikely consider traveling to a vertex in the opposite direction of a destination even if it is located close to the source. Rather, we would consider a nearby vertex closer to the destination unless it is deadend.

Dijkstra in action

So when we are done, we might get something like this:



Shortcomings of A*

In practice, a naive A* algorithm is still not sufficient for many modern applications with large network.

1. Many modern applications require computation to happen in real-time for hundreds, if not thousands, users/agents simultaneously[1].
2. The shift to mobile applications has put more limitations on memory and CPU usage[1].

Hierarchical Path Finding

So instead of running a shortest path algorithm on a large network, we run the shortest path algorithm inside the regional, more coarse grain, network.

Using hierarchy to reduce compleixty

The idea of Hierarchical Path Finding is to create a regional graph, then running a shortest path algorithm between the regions instead for all vertices.

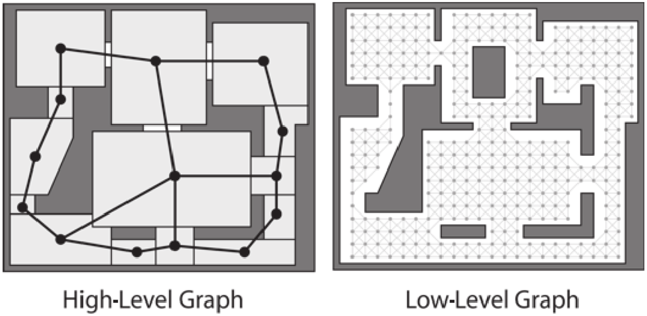


Figure: Forming regions (high-level graph) by clustering neighboring vertices. The figure was taken from *Programming Game AI by Example* by Buckland[2].

How human use Hierarchial Path Finding in real life

1. Plan a route from Virginia Tech to a major highway entrance in Blacksburg.
2. Plan a route from Blacksburg to Charlottesville.
3. Plan a route from the highway exit in Charlottesville to the University of Virginia.

Hierarchical Path Finding algorithm

We apply the similar idea for our grid graph

1. Run A* from source to the source regional entrance,
2. Run A* on regional level, from the source region to destination region, and
3. Run A* from destination region exit to destination.

Challenges

Compare to Dijkstra and naive A*, there are more tunable variables that implementers need to consider

- The number of hierarchy levels,
- Cluster/region size, and
- Placement of regional entrances/exits.
- In practice, optimizations like preprocessing/caching regional routes and path smoothing may be necessary.

Simplification

To avoid overwhelming the algorithm with optimization details, our implementation assume the following simplifications:

1. An input graph is static and known in advance,
2. 2 level of the hierarchy,
3. Preselected regional entrances/exits in reachable locations,
4. No preprocessing/caching, and
5. No path refinement or smoothing.

Regional entrances/exits

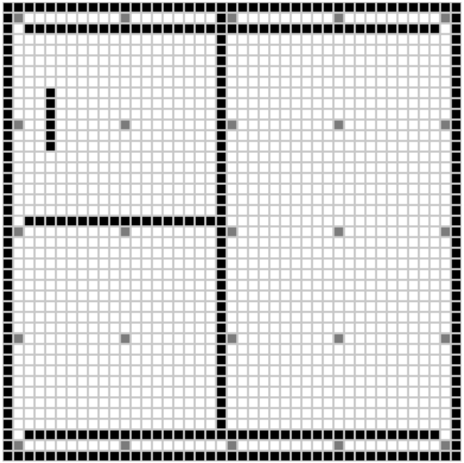


Figure: Regional entrances/exits placement for experiment 1-3.

Experiment setup 1

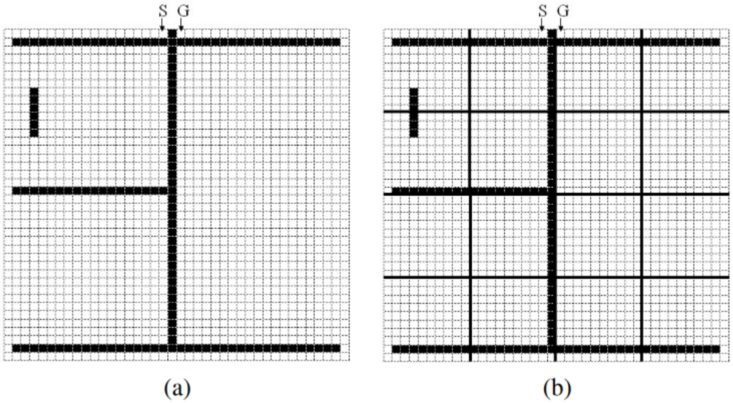
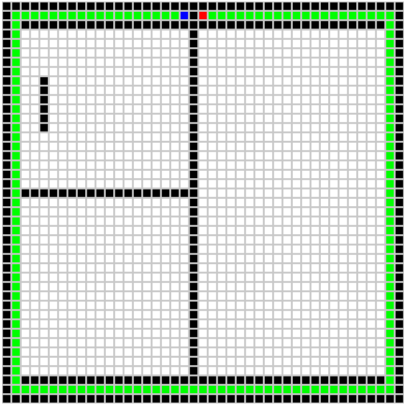


Figure: (a) The 40 X 40 maze used in our example. The obstacles are painted in black. *S* and *G* are the start and the goal nodes. (b) The bold lines show the boundaries of the 10x 10 clusters[1].

Experiment results 1

	Dijkstra	A*	Hierarchical Path Finding
# of vertices visited	1541	1523	993
Path length	159	159	176
Time (ms)	549.95	583.56	18.32



Experiment setup 2

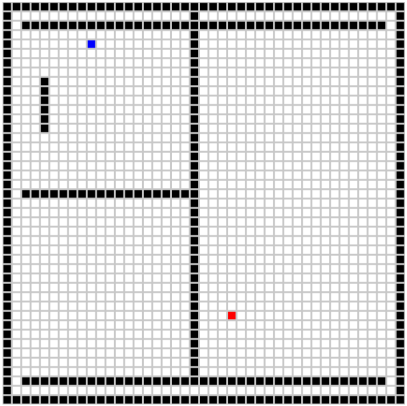


Figure: Gird graph for experiment 2. Blue vertex is the source and red vertex is the destination.

Experiment results 2

	Dijkstra	A*	Hierarchical Path Finding
# of vertices visited	1033	858	610
Path length	111	111	140
Time (ms)	231.84	164.27	10.76

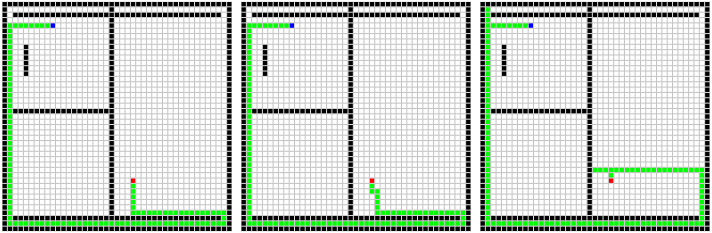


Figure: Results in a left to right order: Dijkstra, A*, Hierarchical Path Finding

Experiment setup 3

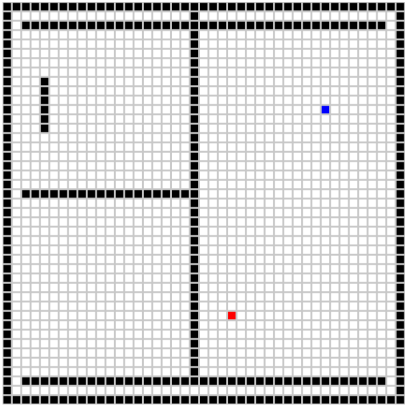


Figure: Gird graph for experiment 3. Blue vertex is the source and red vertex is the destination.

Experiment results 3

	Dijkstra	A*	Hierarchical Path Finding
# of vertices visited	740	329	285
Path length	33	33	44
Time (ms)	149.64	41.82	6.46

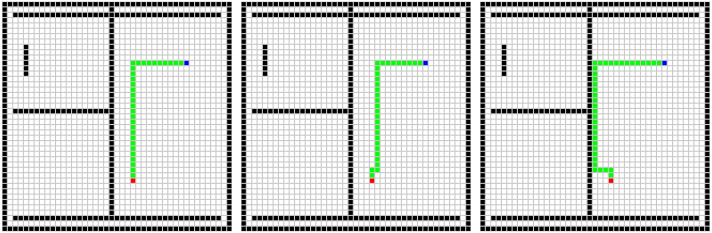


Figure: Results in a left to right order: Dijkstra, A*, Hierarchical Path Finding

Limitations

- Hierarchical Path Finding does not guarantee the shortest path between the source and destination.
- The *Near Optimal Hierarchical Path-Finding* apply a path-smoothing procedure to make a path found by Hierarchical Path Finding within 1% optimal compared to shortest path[1].

Source code

The code was written in Python and can be found at:
`https://github.com/bug-vt/Hierarchical_path_finding`

References

A. Botea, M. Müller, and J. Schaeffer.
Near optimal hierarchical path-finding.
J. Game Dev., 1:1–30, 2004.

M. Buckland.
Programming Game AI by Example, pages 241–247, 359–360, 373.
Wordware Publishing, 2005.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein.
Introduction to Algorithms, pages 620–624.
The MIT Press, 4th ed edition, 2022.

P. Hart, N. Nilsson, and B. Raphael.
A formal basis for the heuristic determination of minimum cost paths.
Transctions on systems science and cybernetics, pages 100–107, 1968.