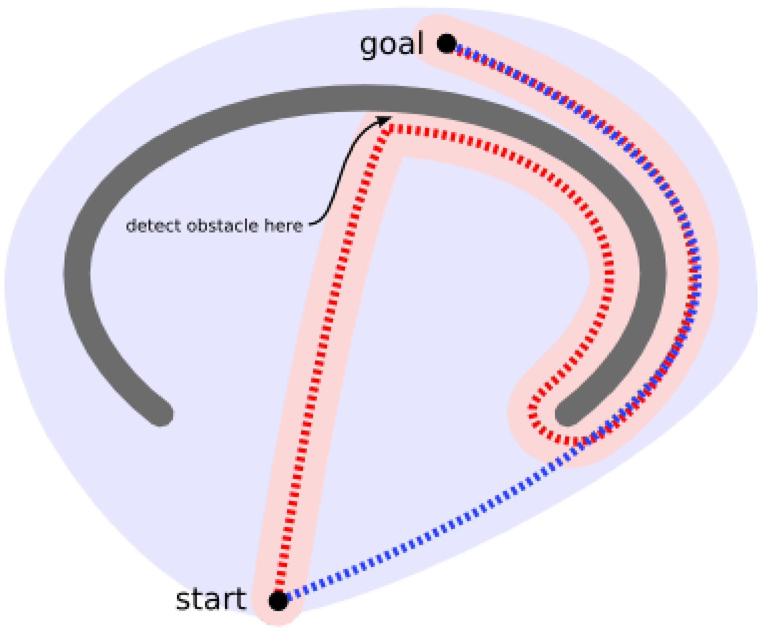
Introduction to A*

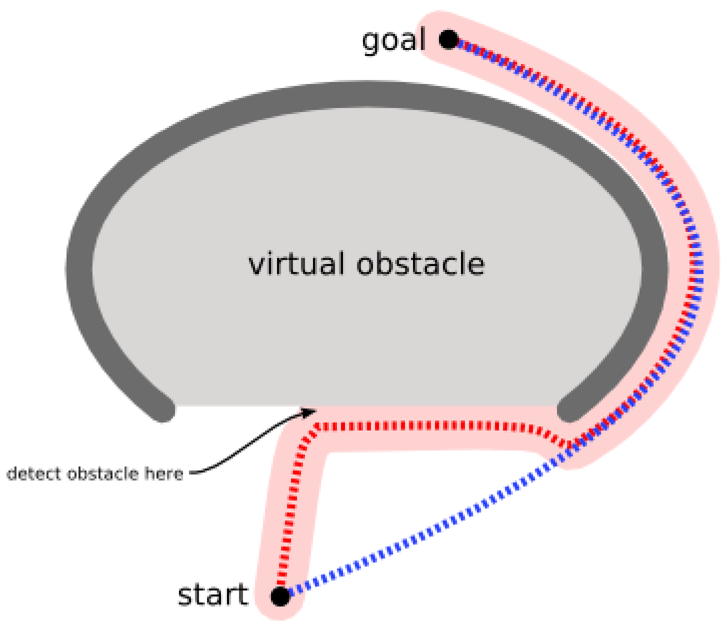
From Amit's Thoughts on Pathfinding

Movement for a single object seems easy. Pathfinding is complex. Why bother with pathfinding? Consider the following situation:



The unit is initially at the bottom of the map and wants to get to the top. There is nothing in the area it scans (shown in pink) to indicate that the unit should not move up, so it continues on its way. Near the top, it detects an obstacle and changes direction. It then finds its way around the "U"-shaped obstacle, following the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue), but found a shorter path (blue), never sending the unit into the concave shaped obstacle.

You can however extend a movement algorithm to work around traps like the one shown above. Either avoid creating concave obstacles, or mark their convex hulls as dangerous (to be entered only if the goal is inside):

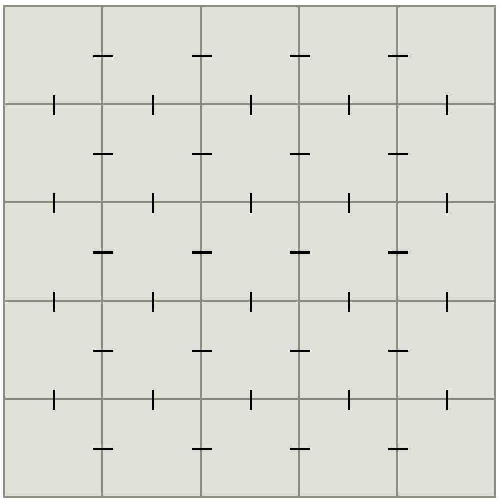


Pathfinders let you plan ahead rather than waiting until the last moment to discover there's a problem. There's a tradeoff between planning with pathfinders and reacting with movement algorithms. Planning generally is slower but gives better results; movement is generally faster but can get stuck. If the game world is changing often, planning ahead is less valuable. I recommend using both: pathfinding for big picture, slow changing obstacles, and long paths; and movement for local area, fast changing, and short paths.

Algorithms

I have written a newer version of this one page[1], but not the rest of the pages. It has interactive diagrams and sample code.

The pathfinding algorithms from computer science textbooks work on *graphs* in the mathematical sense—a set of vertices with edges connecting them. A tiled game map can be considered a graph with each tile being a vertex and edges drawn between tiles that are adjacent to each other:

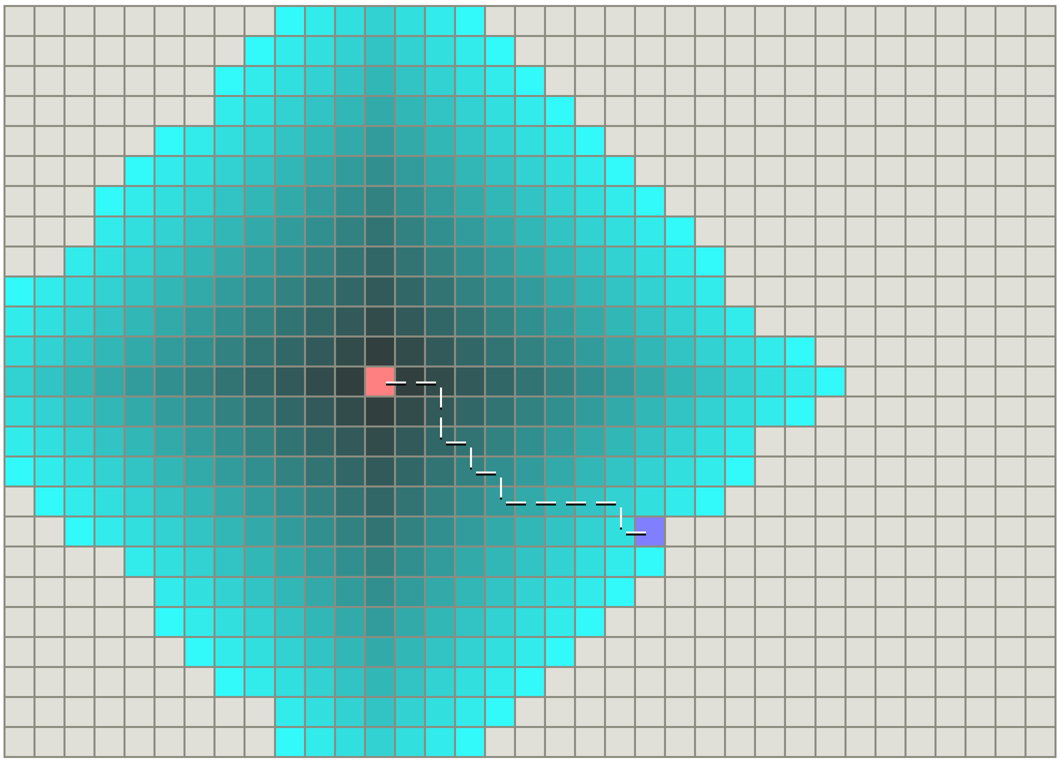


For now, I will assume that we're using two-dimensional grids[2]. If you haven't worked with graphs before, see this primer[3]. Later on, I'll discuss how to build other kinds of graphs out of your game world.

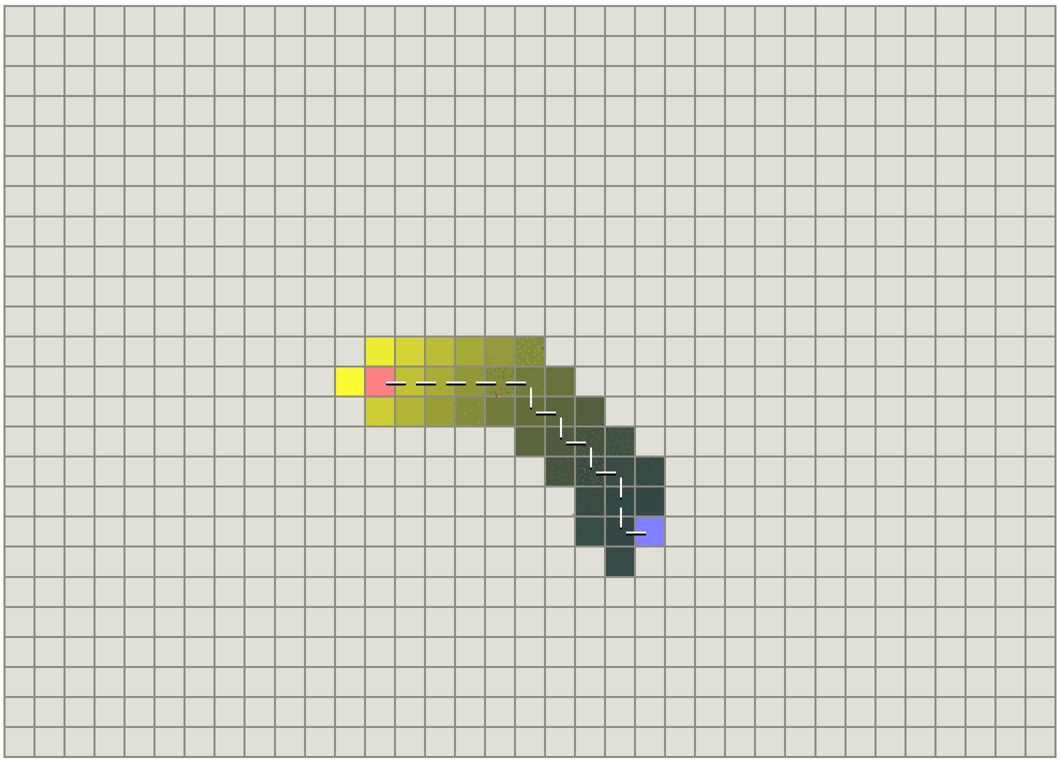
Most pathfinding algorithms from AI or Algorithms research are designed for arbitrary graphs rather than grid-based games. We'd like to find something that can take advantage of the nature of a game map. There are some things we consider common sense, but that algorithms don't understand. We know something about distances: in general, as two things get farther apart, it will take longer to move from one to the other, assuming there are no wormholes. We know something about directions: if your destination is to the east, the best path is more likely to be found by walking to the east than by walking to the west. On grids, we know something about symmetry: most of the time, moving north then east is the same as moving east then north. This additional information can help us make pathfinding algorithms run faster.

Dijkstra's Algorithm and Best-First-Search

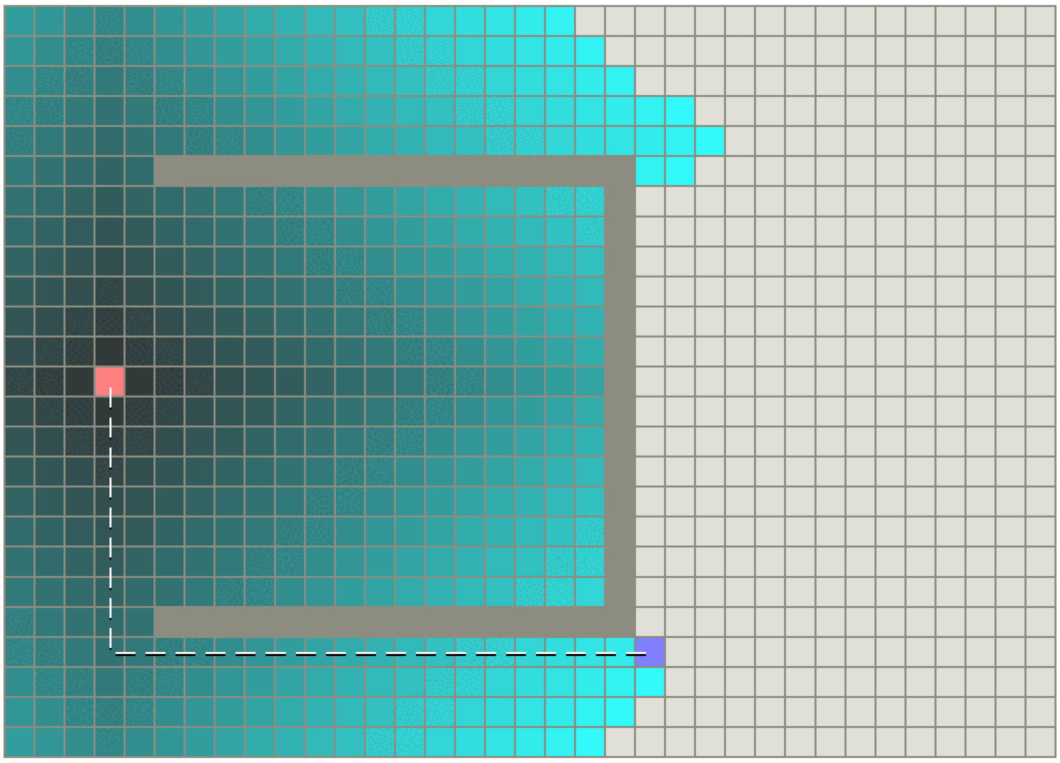
Dijkstra's Algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's Algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost. (I write "a shortest path" because there are often multiple equivalently-short paths.) In the following diagram, the pink square is the starting point, the blue square is the goal, and the teal areas show what areas Dijkstra's Algorithm scanned. The lightest teal areas are those farthest from the starting point, and thus form the "frontier" of exploration:



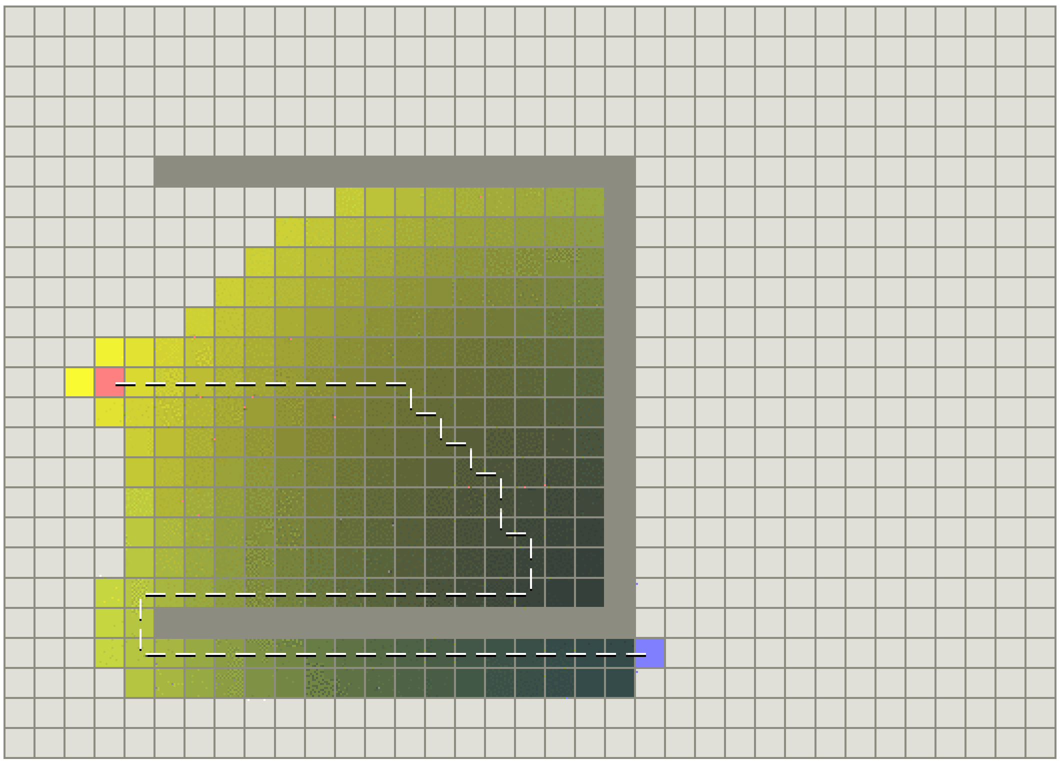
The Greedy Best-First-Search algorithm works in a similar way, except that it has some estimate (called a *heuristic*) of how far from the goal any vertex is. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. Greedy Best-First-Search is *not* guaranteed to find a shortest path. However, it runs much quicker than Dijkstra's Algorithm because it uses the heuristic function to guide its way towards the goal very quickly. For example, if the goal is to the south of the starting position, Greedy Best-First-Search will tend to focus on paths that lead southwards. In the following diagram, yellow represents those nodes with a high heuristic value (high cost to get to the goal) and black represents nodes with a low heuristic value (low cost to get to the goal). It shows that Greedy Best-First-Search can find paths very quickly compared to Dijkstra's Algorithm:



However, both of these examples illustrate the simplest case—when the map has no obstacles, and the shortest path really is a straight line. Let’s consider the concave obstacle as described in the previous section. Dijkstra’s Algorithm works harder but is guaranteed to find a shortest path:



Greedy Best-First-Search on the other hand does less work but its path is clearly not as good:



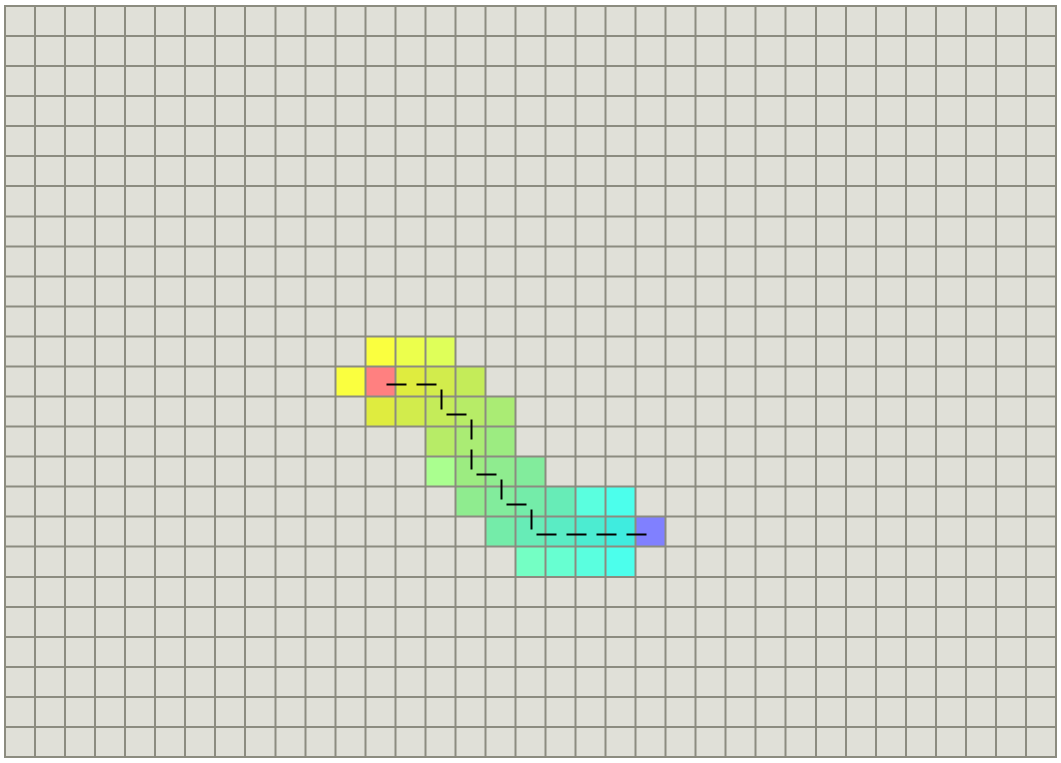
The trouble is that Greedy Best-First-Search is “greedy” and tries to move towards the goal even if it’s not the right path. Since it only considers the cost to get to the goal and ignores the cost of the path so far, it keeps going even if the path it’s on has become really long.

Wouldn’t it be nice to combine the best of both? A* was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra’s Algorithm. It’s a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer. However, A* is built on top of the heuristic, and although the heuristic itself does not give you a guarantee, A* *can* guarantee a shortest path.

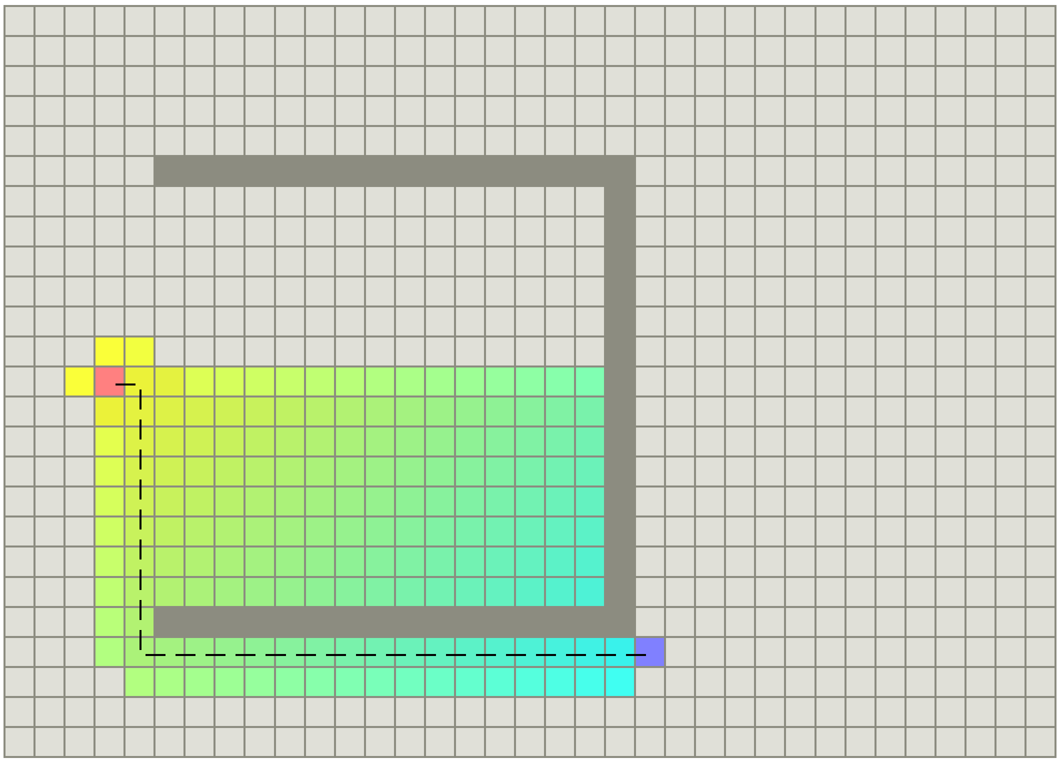
The A* Algorithm

I will be focusing on the **A* Algorithm**[4]. A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's Algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search:



In the example with a concave obstacle, A* finds a path as good as what Dijkstra's Algorithm found:



The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, `g(n)` represents the *exact cost* of the path from the starting point to any vertex `n`, and `h(n)` represents the heuristic *estimated cost* from vertex `n` to the goal. In the above diagrams, the yellow (`h`) represents vertices far from the goal and teal (`g`) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex `n` that has the lowest `f(n) = g(n) + h(n)`.

The rest of this article will explore heuristic design, implementation, map representation, and a variety of other topics related to the use of pathfinding in games. Some sections are well-developed and others are rather incomplete.

Email me redblobgames@gmail.com, or tweet @redblobgames, or comment:

Endnotes

[1]: https://www.redblobgames.com/pathfinding/a-star/introduction.html
[2]: http://www-cs-students.stanford.edu/~amitp/game-programming/grids/
[3]: https://www.redblobgames.com/pathfinding/grids/graphs.html
[4]: http://en.wikipedia.org/wiki/A-star_search_algorithm
