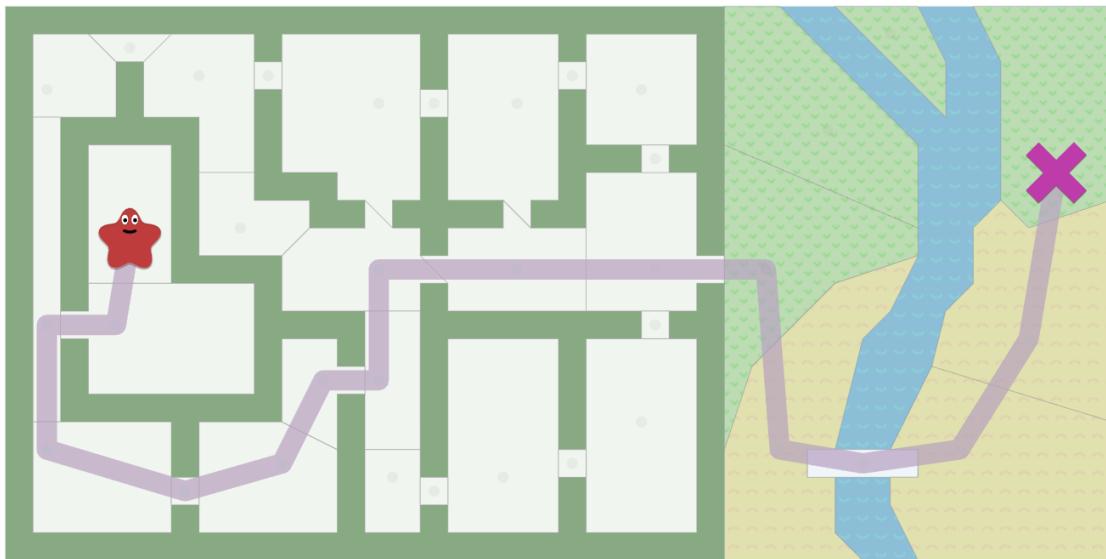


Introduction to the A* Algorithm

from [Red Blob Games](#)

Created 26 May 2014, updated Aug 2014, Feb 2016, Jun 2016, Jun 2020, Jul 2023

Graph search algorithms let us find the shortest path on a map represented as a graph. Move the blob  (start point) and cross  (end point) to see the shortest path found by the **A* Algorithm**:



A* is one of a family of related graph search algorithms:

- ➊ [**Breadth First Search**](#)^[1] explores equally in all directions.
- ➋ [**Dijkstra's Algorithm**](#)^[2] takes into account movement costs.
- ➌ [**A***](#)^[3] explores more towards a single destination.

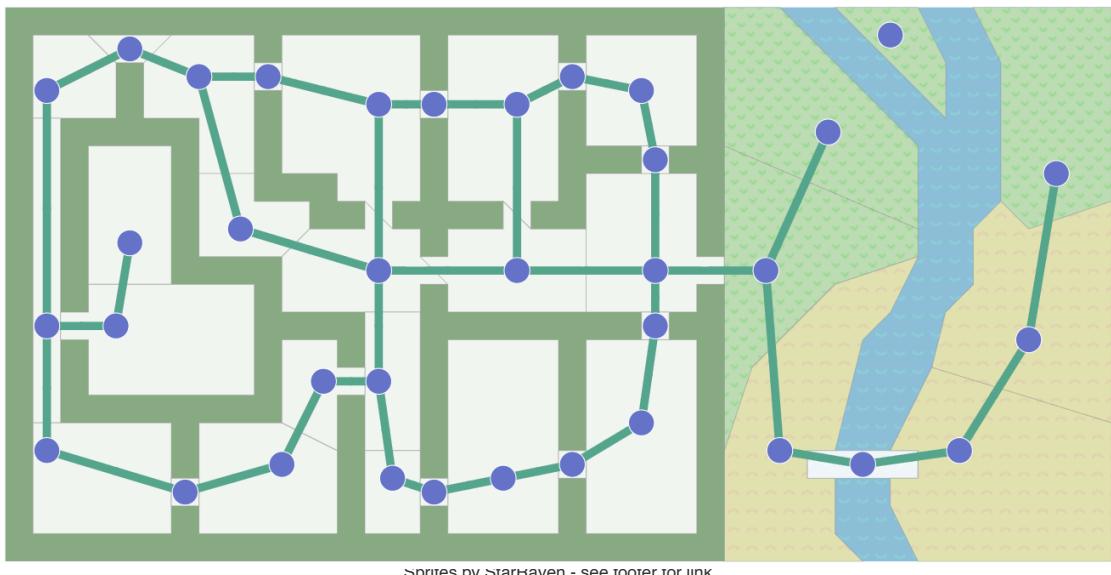
In addition to finding a shortest path, these algorithms can be used for distance maps, flow field pathfinding, connected components, map analysis, garbage collection algorithms, flow networks, and procedural map generation. There are many optimizations and specializations of these algorithms.

Representing the map

#

The first thing to do when studying an algorithm is to understand the **data**. What is the input? What is the output?

Input: Graph search algorithms, including A*, take a “graph” as input. A graph is a set of *locations* ● (“nodes”) and the *connections* — (“edges”) between them. Here’s the graph I gave to A*:

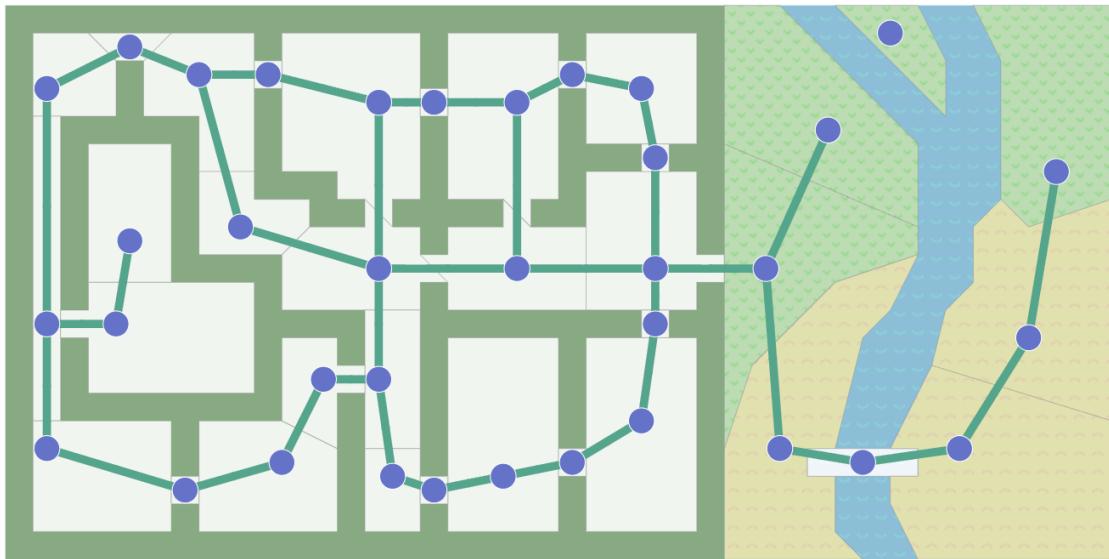


A* doesn’t see anything else. It only sees the graph. It doesn’t know whether something is indoors or outdoors, or if it’s a room or a doorway, or how big an area is. *It only sees the graph!* It doesn’t know the difference between this map and **this other one**.

Output: The path found by A* is **made of graph nodes and edges**.

The edges are abstract mathematical concepts. A* will tell you to move from one location to another *but it won’t tell you how*. Remember that it doesn’t know anything about rooms or doors; all it sees is the graph. You’ll have to decide whether a graph edge returned by A* means moving from tile to tile or walking in a straight line or opening a door or swimming or running along a curved path.

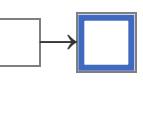
Tradeoffs: There are many different ways of turning a game map into a pathfinding graph. Instead of making doorways into nodes, what if we made **doorways into edges**? What if we used **a pathfinding grid**?

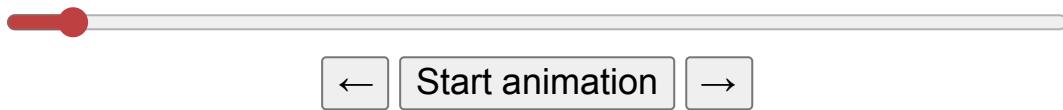
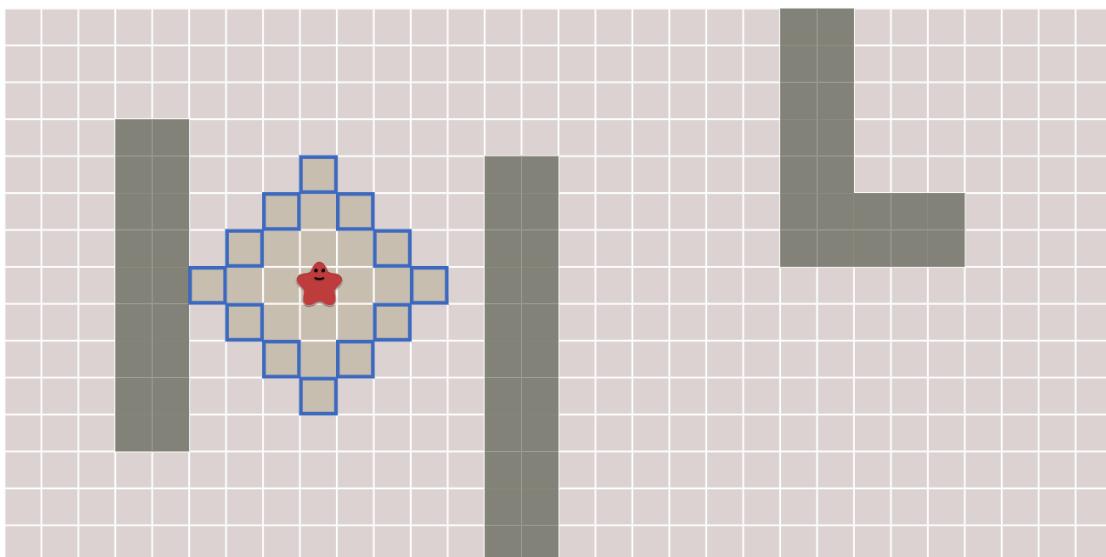


The pathfinding graph doesn't have to be the same as the original problem being solved. A grid map can use a non-grid pathfinding graph, or vice versa. A* runs fastest with the fewest graph nodes; grids are often easier to work with but result in lots of nodes. This page covers the A* algorithm but not graph design; see [my other page](#)^[4] for more about graphs. For the explanations on the rest of the page, *I'm going to use grids because it's easier to visualize the concepts*, but all of these algorithms and sample code work on non-grids too.

Breadth First Search

#

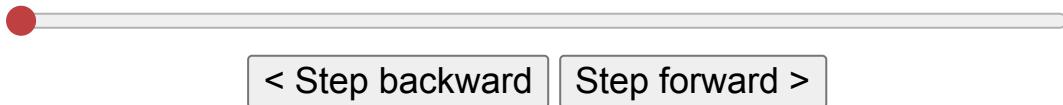
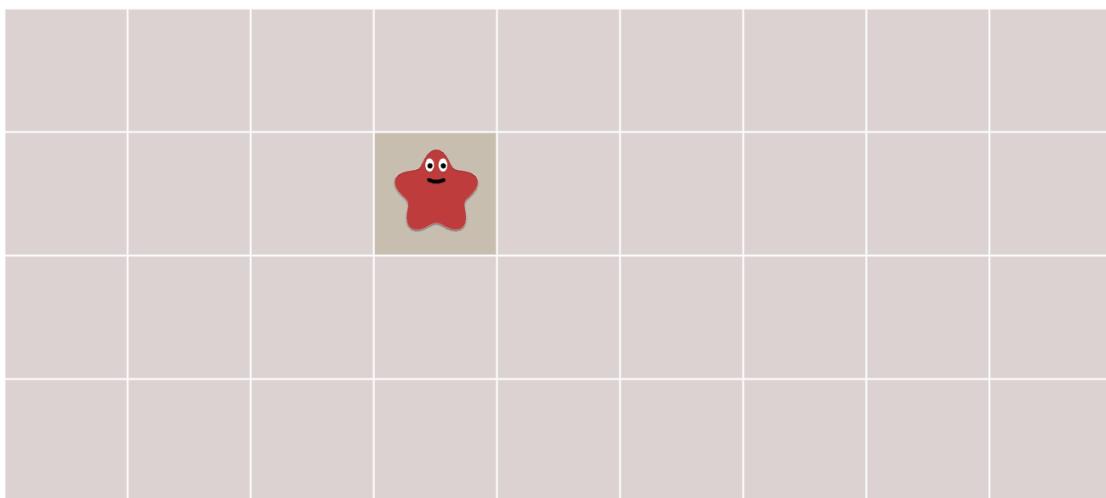
The key idea for all of these algorithms is that we keep track of an expanding ring called the **frontier**. On a grid, this process is sometimes called “flood fill”, but the same technique also works for non-grids. **Start the animation** to see how the frontier expands 



How do we implement this? Repeat these steps until the **frontier** is empty:

1. Pick and remove a **location** from the **frontier**.
2. *Expand* it by looking at its **neighbors** . Skip walls. Any unreached neighbors we add to *both* the **frontier** and the **reached** set .

Let's see this up close. The tiles are numbered in the order we visit them. **Step through** to see the expansion process:



It's only ten lines of (Python) code:

```

frontier = Queue()
frontier.put(start ★)
reached = set()
reached.add(start)

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in reached:
            frontier.put(next)
            reached.add(next)

```

This loop is the essence of the graph search algorithms on this page, including A*. But how do we find the shortest path? The loop doesn't actually construct the paths; it only tells us how to visit everything on the map. That's because Breadth First Search can be used for a lot more than just finding paths; in [this article](#) I show how it's used for tower defense, but it can also be used for distance maps, procedural map generation, and lots of other things. Here though we want to use it for finding paths, so let's modify the loop to keep track of *where we came from* for every location that's been reached, and rename the **reached** set to a **came_from** table (the keys of the table are the reached set):

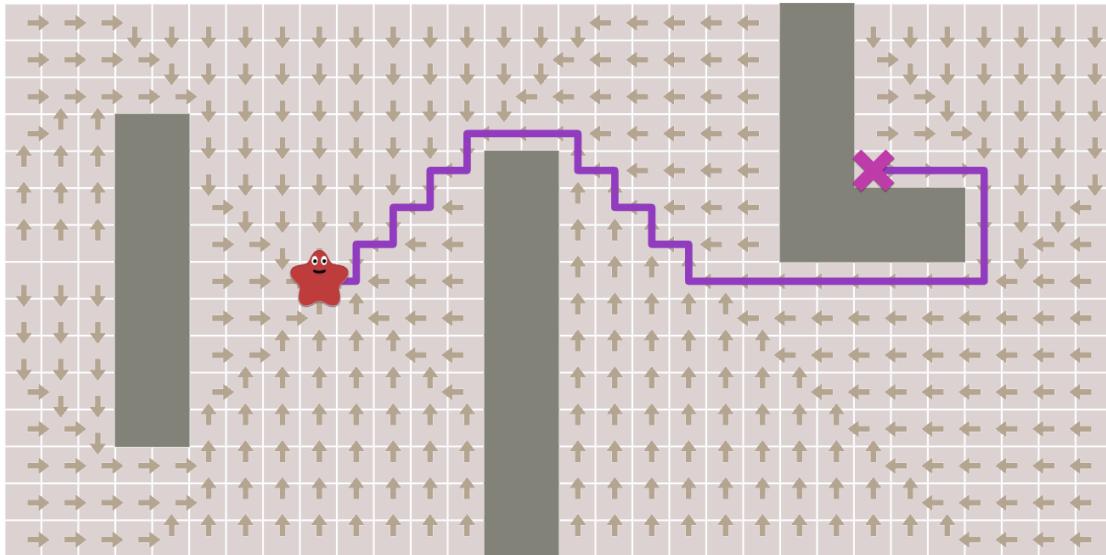
```

frontier = Queue()
frontier.put(start ★)
came_from = dict() # path A->B is stored as came_from[A]
came_from[start] = None

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in came_from:
            frontier.put(next)
            came_from[next] = current

```

Now **came_from** for each location points to the place where we came from. They form a “flow field” and act like “breadcrumbs”. They’re enough to reconstruct the entire path. Move the cross ✕ to see how following the arrows gives you a reverse path back to the start position.



To reconstruct paths: *follow the arrows backwards from the goal to the start*. A path is a *sequence of edges*, but often it’s easier to store the nodes:

```
current = goal ✕
path = [ ]
while current != start: ⭐
        path.append(current)
        current = came_from[current]
path.append(start) # optional
path.reverse() # optional
```

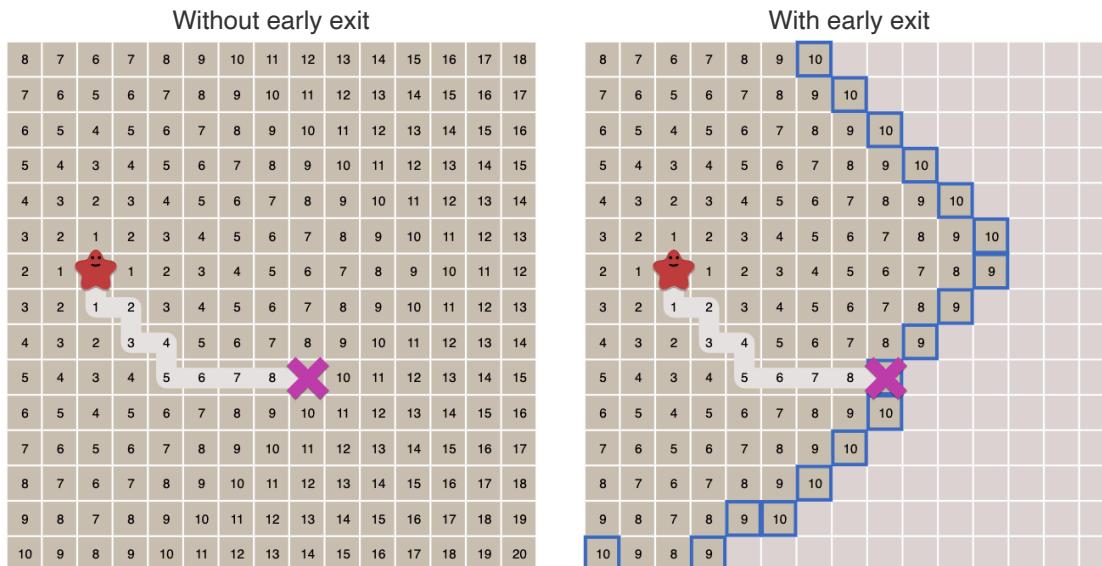
That’s the simplest pathfinding algorithm. It works not only on grids as shown here but on any sort of graph structure. In a dungeon, graph locations could be rooms and graph edges the doorways between them. In a platformer, graph locations could be locations and graph edges the possible actions such as move left, move right, jump up,

jump down. In general, think of the graph as states and actions that change state. I have more written about [map representation](#)^[5]. In the rest of the article I'll continue using examples with grids, and explore why you might use variants of breadth first search.

Early exit

#

We've found paths from *one* location to *all* other locations. Often we don't need all the paths; we only need a path from one location to *one* other location. We can stop expanding the frontier as soon as we've found our goal. Drag the ✕ around see how the frontier stops expanding as soon as it reaches the goal.



We add a test when removing a node from the queue:

```

frontier = Queue()
frontier.put(start ★)
came_from = dict()
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal: ✕
        break

```

```

for next in graph.neighbors(current):
    if next not in came_from:
        frontier.put(next)
        came_from[next] = current

```

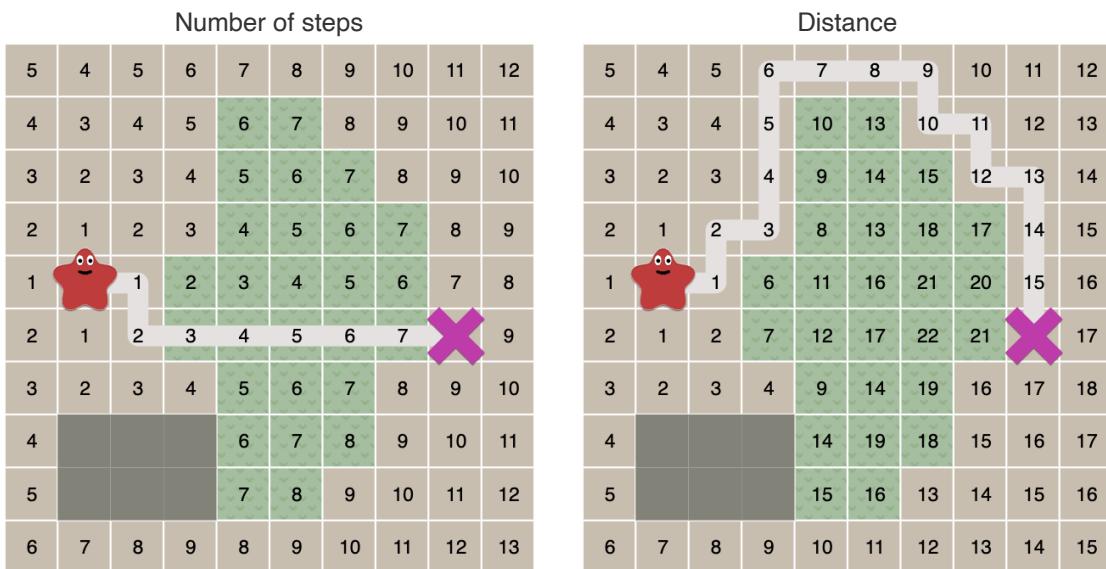
There are lots of cool things you can do with early exit conditions.

Wouldn't it be faster to stop when *adding* the node to the queue? Yes, but only in this specific case. It doesn't work correctly when combining with other features, such as movement costs. I prefer to check when removing the node, so that the same technique works in general. For an example, look at the shortest path from A→E in this graph^[6].

Movement costs

#

So far we've made steps have the same "cost". In some pathfinding scenarios there are different costs for different types of movement. For example in Civilization, moving through plains or desert might cost 1 move-point but moving through forest or hills might cost 5 move-points. In the map at the top of the page, walking through water cost 10 times as much as walking through grass. Another example is diagonal movement on a grid that costs more than axial movement. We'd like the pathfinder to take these costs into account. Let's compare the *number of steps* from the start with the *distance* from the start:



For this we want **Dijkstra's Algorithm** (or Uniform Cost Search). How does it differ from Breadth First Search? We need to track movement costs, so let's add a new variable, **cost_so_far**. This stores the total movement cost from the start location, also called a “distance field”. We want to take the movement costs into account when deciding how to evaluate locations; let's turn our queue into a priority queue. Less obviously, we may end up visiting a location multiple times, with different costs, so we need to alter the logic a little bit. Instead of adding a location to the frontier if the location has never been reached, we'll add it if the new path to the location is better than the best previous path.

```

frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if new_cost < cost_so_far[next]:
            frontier.put(next, new_cost)
            came_from[next] = current
            cost_so_far[next] = new_cost

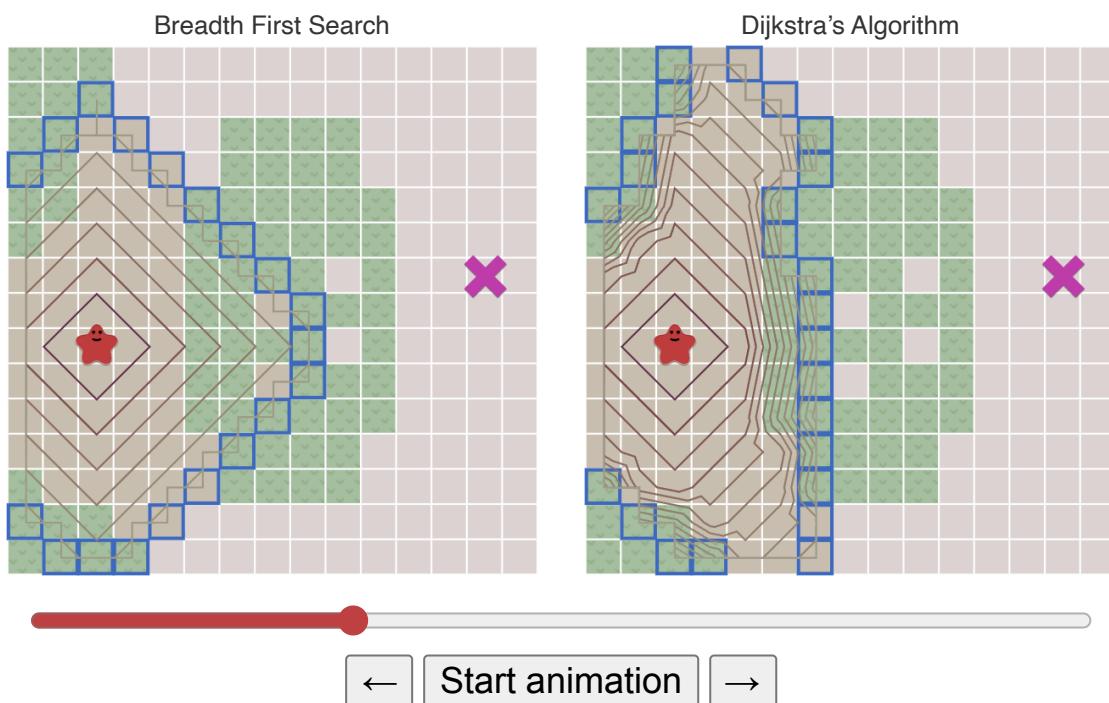
```

```

if next not in cost_so_far or new_cost < cost_:
    cost_so_far[next] = new_cost
    priority = new_cost
    frontier.put(next, priority)
    came_from[next] = current

```

Using a priority queue instead of a regular queue *changes the way the frontier expands*. Contour lines are one way to see this. **Start the animation** to see how the frontier expands more slowly through the forests, finding the shortest path around the central forest instead of through it:



Movement costs other than 1 allow us to explore more interesting graphs, not only grids. In the map at the top of the page, movement costs were based on the distance from room to room. Movement costs can also be used to avoid or prefer areas based on proximity to enemies or allies.

Implementation notes: We want this priority queue to return the *lowest* value first. On the implementation page I show PriorityQueue [in Python](#) using `heapq` to return the lowest value first and also [in C++](#) using `std::priority_queue` configured to return the lowest value first. Also, the version of Dijkstra's Algorithm

and A* I present on this page differs from what's in algorithms textbooks. It's much closer to what's called Uniform Cost Search. I describe the differences [on the implementation page](#).

Heuristic search

#

With Breadth First Search and Dijkstra's Algorithm, the frontier expands in all directions. This is a reasonable choice if you're trying to find a path to all locations or to many locations. However, a common case is to find a path to only one location. Let's make the frontier expand towards the goal more than it expands in other directions. First, we'll define a *heuristic* function that tells us how close we are to the goal:

```
def heuristic(a, b):
    # Manhattan distance on a square grid
    return abs(a.x - b.x) + abs(a.y - b.y)
```

In Dijkstra's Algorithm we used the actual distance from the *start* for the priority queue ordering. Here instead, in **Greedy Best First Search**, we'll use the estimated distance to the *goal* for the priority queue ordering. The location closest to the goal will be explored first. The code uses the priority queue from Dijkstra's Algorithm but without **cost_so_far**:

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in came_from:
```

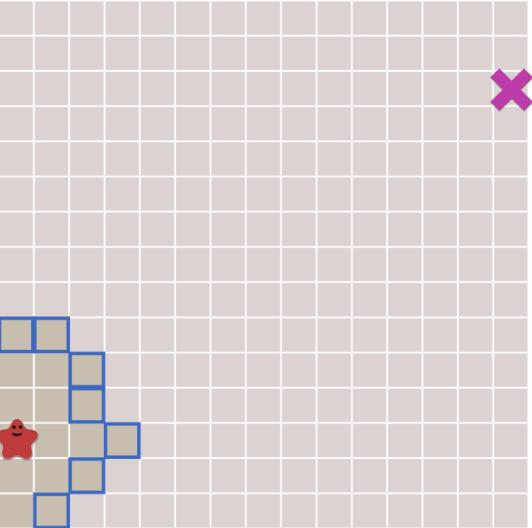
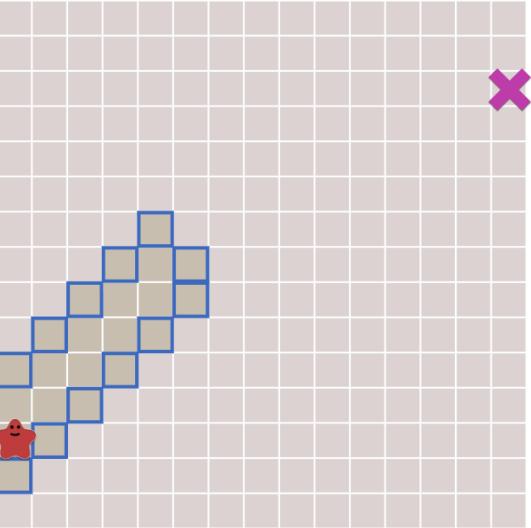
```

priority = heuristic(goal, next)
frontier.put(next, priority)
came_from[next] = current

```

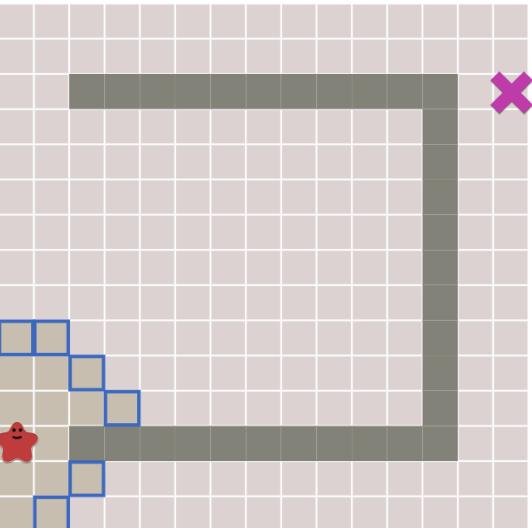
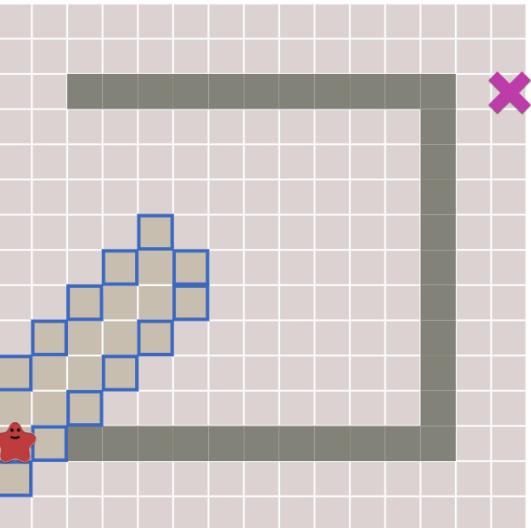
Let's see how well it works:

Dijkstra's Algorithm
Greedy Best-First Search

Wow!! Amazing, right? But what happens in a more complex map?

Dijkstra's Algorithm
Greedy Best-First Search

Those paths aren't the shortest. So this algorithm runs *faster* when there aren't a lot of obstacles, but the paths aren't as *good*. Can we fix this? Yes!

The A* algorithm

#

Dijkstra's Algorithm works well to find the shortest path, but it wastes time exploring in directions that aren't promising. Greedy Best First Search explores in promising directions but it may not find the shortest path. The A* algorithm uses *both* the actual distance from the start and the estimated distance to the goal.

The code is very similar to Dijkstra's Algorithm:

```

frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next]:
            cost_so_far[next] = new_cost
            priority = new_cost + heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

Compare the algorithms: Dijkstra's Algorithm calculates the distance from the start point. Greedy Best-First Search estimates the distance to the goal point. A* is using the sum of those two distances.



Try opening a hole in the wall in various places. You'll find that when Greedy Best-First Search finds the right answer, A* finds it too, exploring the same area. When Greedy Best-First Search finds the wrong answer (longer path), A* finds the right answer, like Dijkstra's Algorithm does, but still explores less than Dijkstra's Algorithm does.

A* is the best of both worlds. As long as the heuristic does not overestimate distances, A* finds an optimal path, like Dijkstra's Algorithm does. A* uses the heuristic to reorder the nodes so that it's *more likely* that the goal node will be encountered sooner.

And ... that's it! That's the A* algorithm.

More



Are you ready to implement this? Consider using an existing library. If you're implementing it yourself, I have [companion guide](#) that shows step by step how to implement graphs, queues, and pathfinding algorithms in Python, C++, and C#.

Which algorithm should you use for finding paths on a map?

- If you want to find paths from or to *all* locations, use Breadth First Search or Dijkstra's Algorithm. Use Breadth First Search if movement costs are all the same; use Dijkstra's Algorithm if movement costs vary.
- If you want to find paths to *one* location, or the closest of several goals, use Greedy Best First Search or A*. Prefer A* in most cases. When you're tempted to use Greedy Best First Search, consider using A* with an "inadmissible" heuristic^[7].

What about optimal paths? Breadth First Search and Dijkstra's Algorithm are guaranteed to find the shortest path given the input graph. Greedy Best First Search is not. A* is guaranteed to find the shortest path if the heuristic is never larger than the true distance. As the heuristic becomes smaller, A* turns into Dijkstra's Algorithm. As the heuristic becomes larger, A* turns into Greedy Best First Search.

What about performance? The best thing to do is to eliminate unnecessary locations in your graph. If using a grid, see this. Reducing the size of the graph helps all the graph search algorithms. After that, use the simplest algorithm you can; simpler queues run faster. Greedy Best First Search typically runs faster than Dijkstra's Algorithm but doesn't produce optimal paths. A* is a good choice for most pathfinding needs.

What about non-maps? I show maps here because I think it's easier to understand how the algorithms work by using a map. However, these graph search algorithms can be used on any sort of graph, not only maps, and I've tried to present the algorithm code in a way that's independent of 2D grids. Movement costs on the maps become arbitrary weights on graph edges. The heuristic encodes information about the graph, so you have to design a heuristic for each type of graph. For planar maps, distances are a good choice, so that's what I've used here.

I have written [more notes about pathfinding](#)^[8]. Keep in mind that graph search is only one part of what you will need. A* doesn't itself handle things like cooperative movement, moving obstacles, map changes, evaluation of dangerous areas, formations, turn radius, object sizes, animation, path smoothing, or lots of other topics.

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

Links

- [1]: https://en.wikipedia.org/wiki/Breadth-first_search
- [2]: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- [3]: https://en.wikipedia.org/wiki/A*_search_algorithm
- [4]: <http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html>
- [5]: <http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html>
- [6]: <https://www.redblobgames.com/pathfinding/posts/reprioritize-example-graph.png>
- [7]: https://en.wikipedia.org/wiki/Admissible_heuristic
- [8]: <http://theory.stanford.edu/~amitp/GameProgramming/>

Copyright © 2025 [Red Blob Games](#)

 [RSS Feed](#)

► Citations:

Created 26 May 2014 with the help of [conrec.js](#); map sprites from StarRaven's [Sketchy Cartography Brushes](#); Last modified: 29 Aug 2025