Implementation of A*

from Red Blob Games

6 Jul 2014

This article is a companion guide to my introduction to A*, where I explain how the algorithms work. On this page I show how to implement Breadth-First Search, Dijkstra's Algorithm, Greedy Best-First Search, and A*. I try to keep the code here simple.

Graph search is a family of related algorithms. There are *lots* of variants of the algorithms, and lots of variants in implementation. Treat the code on this page as a starting point, not as a final version of the algorithm that works for all situations.

1 Python Implementation

I explain most of the code below. There are a few extra bits that you can find in implementation.py. These use **Python 3** so if you use Python 2, you will need to change the `super()` call and the `print` function to the Python 2 equivalents.

1.1 Breadth First Search

Table of Contents

Let's implement Breadth First Search in Python. The main article shows the Python code for the search algorithm, but we also need to define the graph it works on. These are the abstractions I'll use:

Graph
: a data structure that can tell me the `neighbors` for each graph location (see this tutorial). A *weighted* graph can also tell me the `cost` of moving along an edge.

Locations
: a simple value (int, string, tuple, etc.) that *labels* locations in the graph. These are not necessarily locations on the map. They may include additional information such as direction, fuel, lane, or inventory, depending on the problem being solved.

Search
: an algorithm that takes a graph, a starting graph location, and optionally a goal graph location, and calculates some useful information (visited, parent pointer, distance) for some or all graph locations.

Queue
: a data structure used by the search algorithm to decide the order in which to process the graph locations.

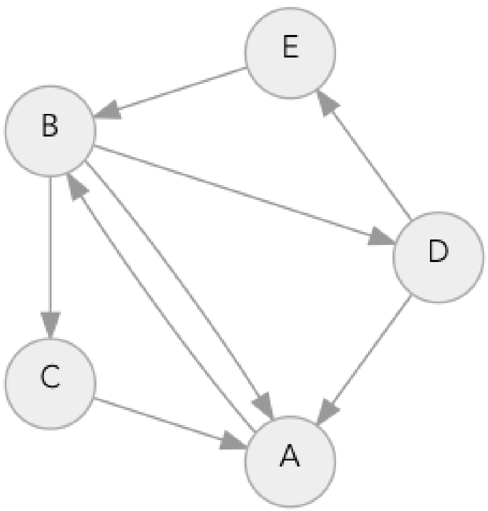
In the main article, I focused on **search**. On this page, I'll fill in the rest of the details to make complete working programs. Let's start with a **graph**:

```
class SimpleGraph:
    def __init__(self):
        self.edges = {}

    def neighbors(self, id):
        return self.edges[id]
```

Yep, that's all we need! You may be asking, where's the Node object? The answer is: I rarely use a node object. I find it simpler to use integers, strings, or tuples as locations, and then use arrays or hash tables that use locations as an index.

Note that the edges are *directed*: we can have an edge from A to B without also having an edge from B to A. In game maps most edges are bidirectional but sometimes there are one-way doors or jumps off cliffs that are expressed as directed edges. Let's make an example graph, where the **locations** are letters A-E.



For each location I need a list of which locations it leads to:

```
example_graph = SimpleGraph()
example_graph.edges = {
    'A': ['B'],
    'B': ['A', 'C', 'D'],
    'C': ['A'],
    'D': ['E', 'A'],
```

```
    'E': ['B']
}
```

Before we can use it with a search algorithm, we need to make a **queue**:

```
import collections

class Queue:
    def __init__(self):
        self.elements = collections.deque()

    def empty(self):
        return len(self.elements) == 0

    def put(self, x):
        self.elements.append(x)

    def get(self):
        return self.elements.popleft()
```

This queue class is just a wrapper around the built-in `collections.deque` class. Feel free to use `deque` directly in your own code.

Let's try the example graph with this queue and the breadth-first search algorithm code from the main article:

```
from implementation import *

def breadth_first_search_1(graph, start):
    # print out what we find
    frontier = Queue()
    frontier.put(start)
    visited = {}
    visited[start] = True

    while not frontier.empty():
```

```
        current = frontier.get()
        print("Visiting %r" % current)
        for next in graph.neighbors(current):
            if next not in visited:
                frontier.put(next)
                visited[next] = True

breadth_first_search_1(example_graph, 'A')
```

```
Visiting 'A'
Visiting 'B'
Visiting 'C'
Visiting 'D'
Visiting 'E'
```

Grids can be expressed as graphs too. I'll now define a new **graph** called `SquareGrid`, with **locations** tuples (int, int). In this map, the locations ("states") in the graph are the same as locations on the game map, but in many problems graph locations are not the same as map locations. Instead of storing the edges explicitly, I'll calculate them in the `neighbors` function. In many problems it's better to store them explicitly.

```
class SquareGrid:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.walls = []

    def in_bounds(self, id):
        (x, y) = id
        return 0 <= x < self.width and 0 <= y < self.height

    def passable(self, id):
        return id not in self.walls

    def neighbors(self, id):
        (x, y) = id
        results = [(x+1, y), (x, y-1), (x-1, y), (x, y+1)]
        if (x + y) % 2 == 0: results.reverse() # aesthetics
        results = filter(self.in_bounds, results)
        results = filter(self.passable, results)
        return results
```

Let’s try it out with the first grid in the main article:

```
from implementation import *
g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS # long list, [(21, 0), (21, 2), ...]
draw_grid(g)
```

```
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . ####. . . . . . . . . . . . . . . . ####. . . . . . .
. . . ####. . . . . . . . ####. . . . . . ####. . . . . . .
. . . ####. . . . . . . . ####. . . . . . ##########. . . .
. . . ####. . . . . . . . ####. . . . . . ##########. . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
```

In order to reconstruct paths we need to store the location of where we came from, so I’ve renamed `visited` (True/False) to `came_from` (location):

```
from implementation import *

def breadth_first_search_2(graph, start):
    # return "came_from"
    frontier = Queue()
    frontier.put(start)
    came_from = {}
    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

    return came_from

g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS

parents = breadth_first_search_2(g, (8, 7))
draw_grid(g, width=2, point_to=parents, start=(8, 7))
```

```
> > > > v v v v v v v v v v v v < < < < < ####v v v v v v v 
> > > > > v v v v v v v v v v < < < < < < ####v v v v v v v 
> > > > > v v v v v v v v v < < < < < < < ####> v v v v v v 
> > ^ ####v v v v v v v v < < < < < < < < ####> > v v v v v 
> ^ ^ ####> v v v v v v < ####^ < < < < < ####> > > v v v v 
^ ^ ^ ####> > v v v v < < ####^ ^ < < < < ##########v v v < 
^ ^ ^ ####> > > v v < < < ####^ ^ ^ < < < ##########v v < < 
^ ^ ^ ####> > > A < < < < ####^ ^ ^ ^ < < < < < < < < < < < 
v v v ####> > ^ ^ ^ < < < ####^ ^ ^ ^ ^ < < < < < < < < < < 
v v v ####> ^ ^ ^ ^ ^ < < ####^ ^ ^ ^ ^ ^ < < < < < < < < < 
v v v ####^ ^ ^ ^ ^ ^ ^ < ####^ ^ ^ ^ ^ ^ ^ < < < < < < < < 
> v v ####^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ < < < < < < < 
> > > > > ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < < < 
> > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < < 
> > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < 
```

Some implementations use *internal storage*, creating a Node object to hold `came_from` and other values for each graph node. I've instead chosen to use *external storage*, creating a single hash table to store the `came_from` for all graph nodes. If you know your map locations have integer indices, another option is to use an array to store `came_from`.

1.2 Early Exit

Following the code from the main article, all we need to do is add an *if* statement in the main loop. This test is optional for Breadth First Search or Dijkstra's Algorithm and effectively required for Greedy Best-First Search and A*:

```python
from implementation import *

def breadth_first_search_3(graph, start, goal):
    frontier = Queue()
    frontier.put(start)
    came_from = {}
    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

    return came_from

g = SquareGrid(30, 15)
g.walls = DIAGRAM1_WALLS

parents = breadth_first_search_3(g, (8, 7), (17, 2))
draw_grid(g, width=2, point_to=parents, start=(8, 7), goal=(17, 2))
```

```
 . > > > v v v v v v v v v v v v < . . . . ####. . . . . . .
 > > > > > v v v v v v v v v v < < < . . . ####. . . . . . .
 > > > > > v v v v v v v v v < < < Z . . . ####. . . . . . .
 > > ^ ####v v v v v v v v < < < < < < . . ####. . . . . . .
 . ^ ^ ####> v v v v v v < ####^ < < . . . ####. . . . . . .
 . . ^ ####> > v v v v < < ####^ ^ . . . . ##########. . . .
 . . . ####> > > v v < < < ####^ . . . . . ##########. . . .
 . . . ####> > > A < < < < ####. . . . . . . . . . . . . . .
 . . . ####> > ^ ^ ^ < < < ####. . . . . . . . . . . . . . .
 . . v ####> ^ ^ ^ ^ ^ < < ####. . . . . . . . . . . . . . .
 . v v ####^ ^ ^ ^ ^ ^ ^ < ####. . . . . . . . . . . . . . .
 > v v ####^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 > > > > > ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 > > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 . > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
```

You can see that the algorithm stops when it finds the goal `Z`.

1.3 Dijkstra's Algorithm

This is what adds complexity to graph search, because we're going to start processing locations in a better order than "first in, first out". What do we need to change?

1. The *graph* needs to know cost of movement.
2. The *queue* needs to return nodes in a different order.
3. The *search* needs to keep track of these costs from the graph and give them to the queue.

1.3.1 Graph with weights

A regular graph tells me the `neighbors` of each node. A *weighted* graph also tells me the cost of moving along each edge. I'm going to add a `cost(from_node, to_node)` function that tells us the cost of moving from location `from_node` to its neighbor `to_node`. In this forest map I chose to make movement depend only on `to_node`, but there are other types of movement that use both nodes[1]. An alternate implementation would be to merge this into the `neighbors` function.

```
class GridWithWeights(SquareGrid):
    def __init__(self, width, height):
        super().__init__(width, height)
        self.weights = {}

    def cost(self, from_node, to_node):
        return self.weights.get(to_node, 1)
```

1.3.2 Queue with priorities

A priority queue associates with each item a number called a “priority”. When returning an item, it picks the one with the lowest number.

insert
: Add item to queue

remove
: Remove item with the lowest number

reprioritize
: (optional) Change an existing item’s priority to a lower number

Here’s a reasonably fast priority queue that uses *binary heaps*, but does not support reprioritize. To get the right ordering, we’ll use tuples (priority, item). When an element is inserted that is already in the queue, we’ll have a duplicate; I’ll explain why that’s ok in the Optimization section.

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]
```

1.3.3 Search

Here's a tricky bit about the implementation: once we add movement costs it's possible to visit a location again, with a better `cost_so_far`. That means the line `if next not in came_from` won't work. Instead, have to check if the cost has gone down since the last time we visited. (In the original version of the article I wasn't checking this, but my code worked anyway; I wrote some notes about that bug.)

This forest map is from the main page.

```
def dijkstra_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    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
                frontier.put(next, priority)
                came_from[next] = current

    return came_from, cost_so_far
```

Finally, after searching I need to build the path:

```
def reconstruct_path(came_from, start, goal):
    current = goal
```

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

Although paths are best thought of as a sequence of edges, it's convenient to store them as a sequence of nodes. To build the path, start at the end and follow the `came_from` map, which points to the previous node. When we reach start, we're done. It is the **backwards** path, so call `reverse()` at the end of `reconstruct_path` if you need it to be stored forwards. Sometimes it's actually more convenient to store it backwards. Sometimes it's useful to also store the start node in the list.

Let's try it out:

```
from implementation import *
came_from, cost_so_far = dijkstra_search(diagram4, (1, 4), (7, 8))
draw_grid(diagram4, width=3, point_to=came_from, start=(1, 4), goal=(7, 8))
print()
draw_grid(diagram4, width=3, number=cost_so_far, start=(1, 4), goal=(7, 8))
print()
draw_grid(diagram4, width=3, path=reconstruct_path(came_from, start=(1, 4), goal=(7, 8)))
```

```
 v  v  <  <  <  <  <  <  <  < 
 v  v  <  <  <  ^  ^  <  <  < 
 v  v  <  <  <  <  ^  ^  <  < 
 v  v  <  <  <  <  <  ^  ^  . 
 >  A  <  <  <  <  .  .  .  . 
 ^  ^  <  <  <  <  .  .  .  . 
 ^  ^  <  <  <  <  <  .  .  . 
 ^ ######### ^  <  v  .  .  . 
 ^ ######### v  v  v  Z  .  . 
 ^  <  <  <  <  <  <  <  <  . 

 5  4  5  6  7  8  9  10 11 12
 4  3  4  5  10 13 10 11 12 13
```

```
3  2  3  4  9  14 15 12 13 14
2  1  2  3  8  13 18 17 14 .
1  A  1  6  11 16 .  .  .  .
2  1  2  7  12 17 .  .  .  .
3  2  3  4  9  14 19 .  .  .
4  #########14 19 18 .  .  .
5  #########15 16 13 Z  .  .
6  7  8  9  10 11 12 13 14 .

.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
@  @  .  .  .  .  .  .  .  .
@  .  .  .  .  .  .  .  .  .
@  .  .  .  .  .  .  .  .  .
@  #########.  .  .  .  .  .
@  #########.  .  @  @  .  .
@  @  @  @  @  @  @  .  .  .
```

The line `if next not in cost_so_far or new_cost < cost_so_far[next]` could be simplified to `if new_cost < cost_so_far.get(next, Infinity)` but I didn't want to explain Python's `get()` in the main article so I left it as is. Another approach would be to use `collections.defaultdict` defaulting to infinity.

1.4 A* Search

Both Greedy Best-First Search and A* use a heuristic function. The only difference is that A* uses both the heuristic and the ordering from Dijkstra's Algorithm. I'm going to show A* here.

```
def heuristic(a, b):
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

def a_star_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start, 0)
```

```
    came_from = {}
    cost_so_far = {}
    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

    return came_from, cost_so_far
```

Let's try it out:

```
from implementation import *
start, goal = (1, 4), (7, 8)
came_from, cost_so_far = a_star_search(diagram4, start, goal)
draw_grid(diagram4, width=3, point_to=came_from, start=start, goal=goal)
print()
draw_grid(diagram4, width=3, number=cost_so_far, start=start, goal=goal)
print()
```

```
 .  .  .  .  .  .  .  .  .  . 
 .  v  v  v  .  .  .  .  .  . 
 v  v  v  v  <  .  .  .  .  . 
 v  v  v  <  <  .  .  .  .  . 
 >  A  <  <  <  .  .  .  .  . 
 >  ^  <  <  <  .  .  .  .  . 
 >  ^  <  <  <  <  .  .  .  . 
 ^  #########^  .  v  .  .  . 
 ^  #########v  v  v  Z  .  . 
 ^  <  <  <  <  <  <  <  .  . 
```

```
 .  .  .  .  .  .  .  .  .  .
 .  3  4  5  .  .  .  .  .  .
 3  2  3  4  9  .  .  .  .  .
 2  1  2  3  8  .  .  .  .  .
 1  A  1  6  11 .  .  .  .  .
 2  1  2  7  12 .  .  .  .  .
 3  2  3  4  9  14 .  .  .  .
 4  #########14 .  18 .  .  .
 5  #########15 16 13 Z  .  .
 6  7  8  9  10 11 12 13 .  .
```

1.4.1 Straighter paths

If you implement this code in your own project you might find that some of the paths aren't as "straight" as you'd like. **This is normal**. When using *grids*, especially grids where every step has the same movement cost, you end up with **ties**: many paths have exactly the same cost. A* ends up picking one of the many short paths, and very often **it doesn't look good to you**. The quick hack is to break the ties[2], but it's not entirely satisfactory. The better approach is to change the map representation, which makes A* a lot faster, and also produces straighter, better looking paths. However, that only works for mostly-static maps where every step has the same movement cost. For the demos on my page, I'm using a quick hack, but it only works with my slow priority queue. If you switch to a faster priority queue you'll need a different quick hack.

---

2 C++ Implementation

Note: some of the sample code needs to include redblobgames/pathfinding/a-star/implementation.cpp to run. I am using **C++11** for this code so some of it will need to be changed if you use an older version of the C++ standard.

The code here is meant for the tutorial and is not production-quality; there's a section at the end with tips on making it better.

2.1 Breadth First Search

Let's implement Breadth First Search in C++. These are the components we need:

Graph
: a data structure that can tell me the `neighbors` for each graph location (see this tutorial). A *weighted* graph can also tell me the `cost` of moving along an edge.

Locations
: a simple value (int, string, tuple, etc.) that *labels* locations in the graph. These are not necessarily locations on the map. They may include additional information such as direction, fuel, lane, or inventory, depending on the problem being solved.

Search
: an algorithm that takes a graph, a starting graph location, and optionally a goal graph location, and calculates some useful information (visited, parent pointer, distance) for some or all graph locations.

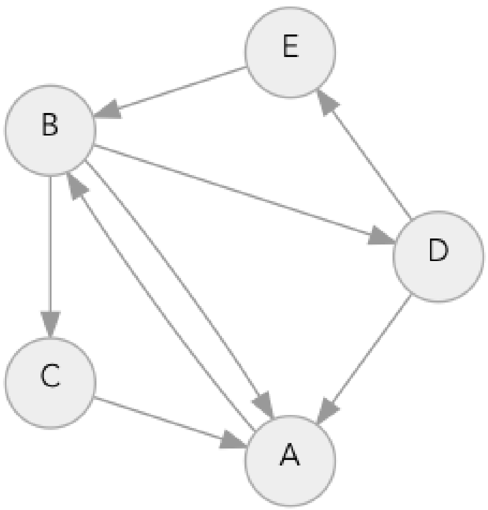
Queue
: a data structure used by the search algorithm to decide the order in which to process the graph locations.

In the main article, I focused on **search**. On this page, I'll fill in the rest of the details to make complete working programs. Let's start with a **graph** where the locations are `char`:

```
struct SimpleGraph {
  std::unordered_map<char, std::vector<char> > edges;

  std::vector<char> neighbors(char id) {
    return edges[id];
  }
};
```

Here's an example:




```
SimpleGraph example_graph {{
    {'A', {'B'}},
    {'B', {'A', 'C', 'D'}},
    {'C', {'A'}},
    {'D', {'E', 'A'}},
    {'E', {'B'}}
  }};
```

The C++ standard library already includes a queue class. We now have a graph (`SimpleGraph`), locations (`char`), and a queue (`std::queue`). Now we can try Breadth First Search:

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

void breadth_first_search(SimpleGraph graph, char start) {
```

```
  std::queue<char> frontier;
  frontier.push(start);

  std::unordered_set<char> visited;
  visited.insert(start);

  while (!frontier.empty()) {
    char current = frontier.front();
    frontier.pop();

    std::cout << "Visiting " << current << '\n';
    for (char next : graph.neighbors(current)) {
      if (visited.find(next) == visited.end()) {
        frontier.push(next);
        visited.insert(next);
      }
    }
  }
}


int main() {
  breadth_first_search(example_graph, 'A');
}
```

```
Visiting A
Visiting B
Visiting C
Visiting D
Visiting E
```

Grids can be expressed as graphs too. I'll now define a new **graph** called `SquareGrid`, with **locations** structs with two ints. In this map, the locations ("states") in the graph are the same as locations on the game map, but in many problems graph locations are not the same as map locations. Instead of storing the edges explicitly, I'll calculate them in the `neighbors` function. In many problems it's better to store them explicitly.

```
struct GridLocation {
  int x, y;
};
```

```
namespace std {
/* implement hash function so we can put GridLocation into an unordered_set */
template <> struct hash<GridLocation> {
  typedef GridLocation argument_type;
  typedef std::size_t result_type;
  std::size_t operator()(const GridLocation& id) const noexcept {
    return std::hash<int>()(id.x ^ (id.y << 4));
  }
};
}


struct SquareGrid {
  static std::array<GridLocation, 4> DIRS;

  int width, height;
  std::unordered_set<GridLocation> walls;

  SquareGrid(int width_, int height_)
     : width(width_), height(height_) {}

  bool in_bounds(GridLocation id) const {
    return 0 <= id.x && id.x < width
        && 0 <= id.y && id.y < height;
  }

  bool passable(GridLocation id) const {
    return walls.find(id) == walls.end();
  }

  std::vector<GridLocation> neighbors(GridLocation id) const {
    std::vector<GridLocation> results;

    for (GridLocation dir : DIRS) {
      GridLocation next{id.x + dir.x, id.y + dir.y};
      if (in_bounds(next) && passable(next)) {
        results.push_back(next);
      }
    }

    if ((id.x + id.y) % 2 == 0) {
      // aesthetic improvement on square grids
      std::reverse(results.begin(), results.end());
    }

    return results;
  }
```

```
};

std::array<GridLocation, 4> SquareGrid::DIRS =
  {GridLocation{1, 0}, GridLocation{0, -1}, GridLocation{-1, 0}, GridLocation{0, 1}};
```

In the helper file `implementation.cpp` I defined a function to make grids:

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

int main() {
  SquareGrid grid = make_diagram1();
  draw_grid(grid, 2);
}
```

```
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . . . . . . . . . . . . . . . . . . . ####. . . . . . .
. . . ####. . . . . . . . . . . . . . . . ####. . . . . . .
. . . ####. . . . . . . . ####. . . . . . ####. . . . . . .
. . . ####. . . . . . . . ####. . . . . . ##########. . . .
. . . ####. . . . . . . . ####. . . . . . ##########. . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . ####. . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
. . . . . . . . . . . . . ####. . . . . . . . . . . . . . .
```

Let’s try Breadth First Search again, keeping track of `came_from`:

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

template<typename Location, typename Graph>
std::unordered_map<Location, Location>
breadth_first_search(Graph graph, Location start) {
  std::queue<Location> frontier;
```

```
  frontier.push(start);

  std::unordered_map<Location, Location> came_from;
  came_from[start] = start;

  while (!frontier.empty()) {
    Location current = frontier.front();
    frontier.pop();

    for (Location next : graph.neighbors(current)) {
      if (came_from.find(next) == came_from.end()) {
        frontier.push(next);
        came_from[next] = current;
      }
    }
  }
  return came_from;
}

int main() {
  SquareGrid grid = make_diagram1();
  auto parents = breadth_first_search(grid, GridLocation{7, 8});
  draw_grid(grid, 2, nullptr, &parents);
}
```

```
> > > > v v v v v v v v v v v v < < < < < ####v v v v v v v
> > > > > v v v v v v v v v v < < < < < < ####v v v v v v v
> > > > > v v v v v v v v v < < < < < < < ####> v v v v v v
> > ^ ####v v v v v v v v < < < < < < < < ####> > v v v v v
> ^ ^ ####v v v v v v v < ####^ < < < < < ####> > > v v v v
^ ^ ^ ####v v v v v v < < ####^ ^ < < < < ##########v v v <
^ ^ ^ ####> v v v v < < < ####^ ^ ^ < < < ##########v v < <
v v v ####> > v v < < < < ####^ ^ ^ ^ < < < < < < < < < < <
v v v ####> > * < < < < < ####^ ^ ^ ^ ^ < < < < < < < < < <
v v v ####> ^ ^ ^ < < < < ####^ ^ ^ ^ ^ ^ < < < < < < < < <
v v v ####^ ^ ^ ^ ^ < < < ####^ ^ ^ ^ ^ ^ ^ < < < < < < < <
> v v ####^ ^ ^ ^ ^ ^ < < ####^ ^ ^ ^ ^ ^ ^ ^ < < < < < < <
> > > > > ^ ^ ^ ^ ^ ^ ^ < ####^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < < <
> > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < < <
> > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ < < < <
```

Some implementations use *internal storage*, creating a Node object to hold `came_from` and other values for each graph node. I've instead chosen to use *external storage*, creating a single `std::unordered_map` to store the `came_from` for all graph nodes. If you know your map locations have integer indices, another option is to use a 1D or 2D array/vector to store `came_from` and other values.

2.2 Early Exit

Breadth First Search and Dijkstra's Algorithm will explore the entire map by default. If we're looking for a path to a single, point we can add `if (current == goal)` to exit the loop as soon as we find the path.

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

template<typename Location, typename Graph>
std::unordered_map<Location, Location>
breadth_first_search(Graph graph, Location start, Location goal) {
  std::queue<Location> frontier;
  frontier.push(start);

  std::unordered_map<Location, Location> came_from;
  came_from[start] = start;

  while (!frontier.empty()) {
    Location current = frontier.front();
    frontier.pop();

    if (current == goal) {
      break;
    }

    for (Location next : graph.neighbors(current)) {
      if (came_from.find(next) == came_from.end()) {
        frontier.push(next);
        came_from[next] = current;
      }
    }
  }
  return came_from;
```

```
}

int main() {
  GridLocation start{8, 7};
  GridLocation goal{17, 2};
  SquareGrid grid = make_diagram1();
  auto came_from = breadth_first_search(grid, start, goal);
  draw_grid(grid, 2, nullptr, &came_from);
}
```

```
 . > > > v v v v v v v v v v v v < . . . . ####. . . . . . .
 > > > > > v v v v v v v v v v < < < . . . ####. . . . . . .
 > > > > > v v v v v v v v v < < < < . . . ####. . . . . . .
 > > ^ ####v v v v v v v v < < < < < < . . ####. . . . . . .
 . ^ ^ ####> v v v v v v < ####^ < < . . . ####. . . . . . .
 . . ^ ####> > v v v v < < ####^ ^ . . . . ##########. . . .
 . . . ####> > > v v < < < ####^ . . . . . ##########. . . .
 . . . ####> > > * < < < < ####. . . . . . . . . . . . . . .
 . . . ####> > ^ ^ ^ < < < ####. . . . . . . . . . . . . . .
 . . v ####> ^ ^ ^ ^ ^ < < ####. . . . . . . . . . . . . . .
 . v v ####^ ^ ^ ^ ^ ^ ^ < ####. . . . . . . . . . . . . . .
 > v v ####^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 > > > > > ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 > > > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
 . > > ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ####. . . . . . . . . . . . . . .
```

In the output we can see that the algorithm did not explore the entire map, but stopped early.

2.3 Dijkstra’s Algorithm

This is what adds complexity to graph search, because we’re going to start processing locations in a better order than “first in, first out”. What do we need to change?

1. The *graph* needs to know cost of movement.
2. The *queue* needs to return nodes in a different order.
3. The *search* needs to keep track of these costs from the graph and give them to the queue.

2.3.1 Graph with weights

A regular graph tells me the `neighbors` of each node. A *weighted* graph also tells me the cost of moving along each edge. I'm going to add a `cost(from_node, to_node)` function that tells us the cost of moving from location `from_node` to its neighbor `to_node`. In this forest map I chose to make movement depend only on `to_node`, but there are other types of movement that use both nodes[3]. An alternate implementation would be to merge this into the `neighbors` function. Here's a grid with a list of forest tiles, which will have movement cost 5:

```
struct GridWithWeights: SquareGrid {
  std::unordered_set<GridLocation> forests;
  GridWithWeights(int w, int h): SquareGrid(w, h) {}
  double cost(GridLocation from_node, GridLocation to_node) const {
    return forests.find(to_node) != forests.end()? 5 : 1;
  }
};
```

2.3.2 Queue with priorities

We need a priority queue. C++ offers a `priority_queue` class that uses a binary heap but not the reprioritize operation. I'll use a pair (priority, item) for the queue elements to get the right ordering. By default, the C++ priority queue returns the maximum element first, using the `std::less` comparator; we want the minimum element instead, so I'll use the `std::greater` comparator.

```
template<typename T, typename priority_t>
struct PriorityQueue {
  typedef std::pair<priority_t, T> PQElement;
  std::priority_queue<PQElement, std::vector<PQElement>,
                 std::greater<PQElement>> elements;
```

```
  inline bool empty() const {
     return elements.empty();
  }

  inline void put(T item, priority_t priority) {
    elements.emplace(priority, item);
  }

  T get() {
    T best_item = elements.top().second;
    elements.pop();
    return best_item;
  }
};
```

In this sample code I'm wrapping the C++ `std::priority_queue` class but I think it'd be reasonable to use that class directly without the wrapper.

2.3.3 Search

See the forest map from the main page.

```
template<typename Location, typename Graph>
void dijkstra_search
  (Graph graph,
   Location start,
   Location goal,
   std::unordered_map<Location, Location>& came_from,
   std::unordered_map<Location, double>& cost_so_far)
{
  PriorityQueue<Location, double> frontier;
  frontier.put(start, 0);

  came_from[start] = start;
  cost_so_far[start] = 0;

  while (!frontier.empty()) {
    Location current = frontier.get();
```

```
    if (current == goal) {
      break;
    }

    for (Location next : graph.neighbors(current)) {
      double new_cost = cost_so_far[current] + graph.cost(current, next);
      if (cost_so_far.find(next) == cost_so_far.end()
          || new_cost < cost_so_far[next]) {
        cost_so_far[next] = new_cost;
        came_from[next] = current;
        frontier.put(next, new_cost);
      }
    }
  }
}
```

The types of the `cost` variables should all match the types used in the graph. If you use `int` then you can use `int` for the cost variable and the priorities in the priority queue; if you use `double` then you should use `double` for these. In this code I used `double` but I could've used `int` and it would've worked the same. However, if your graph edge costs are doubles or if your heuristic uses doubles, then you'll need to use doubles here.

Finally, after searching I need to build the path:

```
template<typename Location>
std::vector<Location> reconstruct_path(
   Location start, Location goal,
   std::unordered_map<Location, Location> came_from
) {
  std::vector<Location> path;
  Location current = goal;
  while (current != start) {
    path.push_back(current);
    current = came_from[current];
  }
  path.push_back(start); // optional
  std::reverse(path.begin(), path.end());
```

```
  return path;
}
```

Although paths are best thought of as a sequence of edges, it's convenient to store them as a sequence of nodes. To build the path, start at the end and follow the `came_from` map, which points to the previous node. When we reach start, we're done. It is the **backwards** path, so call `reverse()` at the end of `reconstruct_path` if you need it to be stored forwards. Sometimes it's actually more convenient to store it backwards. Sometimes it's useful to also store the start node in the list.

Let's try it out:

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

int main() {
  GridWithWeights grid = make_diagram4();
  GridLocation start{1, 4};
  GridLocation goal{8, 5};
  std::unordered_map<GridLocation, GridLocation> came_from;
  std::unordered_map<GridLocation, double> cost_so_far;
  dijkstra_search(grid, start, goal, came_from, cost_so_far);
  draw_grid(grid, 2, nullptr, &came_from);
  std::cout << '\n';
  draw_grid(grid, 3, &cost_so_far, nullptr);
  std::cout << '\n';
  std::vector<GridLocation> path = reconstruct_path(start, goal, came_from);
  draw_grid(grid, 3, nullptr, nullptr, &path);
}
```

```
 v  v  <  <  <  <  <  <  <  <
 v  v  <  <  <  ^  ^  <  <  <
 v  v  <  <  <  <  ^  ^  <  <
 v  v  <  <  <  <  <  ^  ^  <
 >  *  <  <  <  <  <  >  ^  <
 ^  ^  <  <  <  <  .  v  ^  .
 ^  ^  <  <  <  <  <  v  <  .
 ^  ######^  <  v  v  <  .
 ^  ######v  v  v  <  <  <
```

```
^ < < < < < < < < <

5  4  5  6  7  8  9  10 11 12
4  3  4  5  10 13 10 11 12 13
3  2  3  4  9  14 15 12 13 14
2  1  2  3  8  13 18 17 14 15
1  0  1  6  11 16 21 20 15 16
2  1  2  7  12 17 .  21 16 .
3  2  3  4  9  14 19 16 17 .
4  #########14 19 18 15 16 .
5  #########15 16 13 14 15 16
6  7  8  9  10 11 12 13 14 15

.  @  @  @  @  @  @  .  .  .
.  @  .  .  .  .  @  @  .  .
.  @  .  .  .  .  .  @  @  .
.  @  .  .  .  .  .  .  @  .
.  @  .  .  .  .  .  .  @  .
.  .  .  .  .  .  .  .  @  .
.  .  .  .  .  .  .  .  .  .
.  #########.  .  .  .  .  .
.  #########.  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .
```

The results are not exactly the same as the Python version because I'm using the built-in priority queues in C++ and Python may order equal-valued nodes differently.

2.4 A* Search

A* is almost exactly like Dijkstra's Algorithm, except we add in a heuristic. Note that the code for the algorithm *isn't specific to grids*. Knowledge about grids is in the graph class (`SquareGrids` in this case), the locations (`Location` struct), and in the `heuristic` function. Replace those three and you can use the A* algorithm code with any other graph structure.

```
inline double heuristic(GridLocation a, GridLocation b) {
  return std::abs(a.x - b.x) + std::abs(a.y - b.y);
```

```
}

template<typename Location, typename Graph>
void a_star_search
  (Graph graph,
   Location start,
   Location goal,
   std::unordered_map<Location, Location>& came_from,
   std::unordered_map<Location, double>& cost_so_far)
{
  PriorityQueue<Location, double> frontier;
  frontier.put(start, 0);

  came_from[start] = start;
  cost_so_far[start] = 0;

  while (!frontier.empty()) {
    Location current = frontier.get();

    if (current == goal) {
      break;
    }

    for (Location next : graph.neighbors(current)) {
      double new_cost = cost_so_far[current] + graph.cost(current, next);
      if (cost_so_far.find(next) == cost_so_far.end()
          || new_cost < cost_so_far[next]) {
        cost_so_far[next] = new_cost;
        double priority = new_cost + heuristic(next, goal);
        frontier.put(next, priority);
        came_from[next] = current;
      }
    }
  }
}
```

The type of the `priority` values including the type used in the priority queue should be big enough to include both the graph costs (`cost_t`) and the heuristic value. For example, if the graph costs are ints and the heuristic returns a double, then you need the priority queue to accept doubles. In this sample code I use `double` for all three (cost, heuristic, and priority), but I could've used `int` because my costs and heuristics are integer valued.

Minor note: It would be more correct to write `frontier.put(start, heuristic(start, goal))` than `frontier.put(start, 0)` but it makes no difference here because the start node's priority doesn't matter. It is the only node in the priority queue and it is selected and removed before anything else is put in there.

```
#include "redblobgames/pathfinding/a-star/implementation.cpp"

int main() {
  GridWithWeights grid = make_diagram4();
  GridLocation start{1, 4};
  GridLocation goal{8, 5};
  std::unordered_map<GridLocation, GridLocation> came_from;
  std::unordered_map<GridLocation, double> cost_so_far;
  a_star_search(grid, start, goal, came_from, cost_so_far);
  draw_grid(grid, 2, nullptr, &came_from);
  std::cout << '\n';
  draw_grid(grid, 3, &cost_so_far, nullptr);
  std::cout << '\n';
  std::vector<GridLocation> path = reconstruct_path(start, goal, came_from);
  draw_grid(grid, 3, nullptr, nullptr, &path);
}
```

```
v v v v < < < < < <
v v v v < ^ ^ < < <
v v v v < < ^ ^ < <
v v v < < < . ^ ^ <
> * < < < < . > ^ <
> ^ < < < < . . ^ .
^ ^ ^ < < < . . . .
^ ######^ . . . . .
^ ######. . . . . .
^ . . . . . . . . .

5  4  5  6  7  8  9  10 11 12
4  3  4  5  10 13 10 11 12 13
3  2  3  4  9  14 15 12 13 14
2  1  2  3  8  13 .  17 14 15
1  0  1  6  11 16 .  20 15 16
2  1  2  7  12 17 .  .  16 .
3  2  3  4  9  14 .  .  .  .
4  #########14 .  .  .  .  .
5  #########.  .  .  .  .  .
6  .  .  .  .  .  .  .  .  .
```

```
 .  .  .  @  @  @  @  .  .  . 
 .  .  .  @  .  .  @  @  .  . 
 .  .  .  @  .  .  .  @  @  . 
 .  .  @  @  .  .  .  .  @  . 
 .  @  @  .  .  .  .  .  @  . 
 .  .  .  .  .  .  .  .  @  . 
 .  .  .  .  .  .  .  .  .  . 
 . #########.  .  .  .  .  . 
 . #########.  .  .  .  .  . 
 .  .  .  .  .  .  .  .  .  . 
```

2.4.1 Straighter paths

If you implement this code in your own project you might find that some of the paths aren't as "straight" as you'd like. **This is normal**. When using *grids*, especially grids where every step has the same movement cost, you end up with **ties**: many paths have exactly the same cost. A* ends up picking one of the many short paths, and very often **it doesn't look good to you**. The quick hack is to break the ties[4], but it's not entirely satisfactory. The better approach is to change the map representation, which makes A* a lot faster, and also produces straighter, better looking paths. However, that only works for mostly-static maps where every step has the same movement cost. For the demos on my page, I'm using a quick hack, but it only works with my slow priority queue. If you switch to a faster priority queue you'll need a different quick hack.

2.5 Production code

The C++ code I've shown above is simplified to make it easier to follow the algorithm and data structures. In practice there are many things you'd want to do differently:

- inlining small functions
- the `Location` parameter should be part of the `Graph`

- the cost could be int or double, and should be part of the `Graph`
- use `array` instead of `unordered_set` if the ids are dense integers, and reset these values on exit instead of initializing on entry
- pass larger data structures by reference instead of by value
- return larger data structures in out parameters instead of returning them, or use move constructors (for example, the vector returned from the `neighbors` function)
- the heuristic can vary and should be a template parameter to the A* function so that it can be inlined

Here's how the A* code might look different with some (but not all) of these changes:

```
template<typename Graph>
void a_star_search
  (Graph graph,
   typename Graph::Location start,
   typename Graph::Location goal,
   std::function<typename Graph::cost_t(typename Graph::Location a, typename Graph::Location b)> heuristic,
   std::unordered_map<typename Graph::Location, typename Graph::Location>& came_from,
   std::unordered_map<typename Graph::Location, typename Graph::cost_t>& cost_so_far)
{
  typedef typename Graph::Location Location;
  typedef typename Graph::cost_t cost_t;
  PriorityQueue<Location, cost_t> frontier;
  std::vector<Location> neighbors;
  frontier.put(start, cost_t(0));

  came_from[start] = start;
  cost_so_far[start] = cost_t(0);

  while (!frontier.empty()) {
    typename Location current = frontier.get();

    if (current == goal) {
      break;
    }

    graph.get_neighbors(current, neighbors);
    for (Location next : neighbors) {
```

```
        cost_t new_cost = cost_so_far[current] + graph.cost(current, next);
        if (cost_so_far.find(next) == cost_so_far.end()
            || new_cost < cost_so_far[next]) {
          cost_so_far[next] = new_cost;
          cost_t priority = new_cost + heuristic(next, goal);
          frontier.put(next, priority);
          came_from[next] = current;
        }
      }
    }
  }
```

I wanted the code on this page to be about the algorithms and data structures and not about the C++ optimizations so I tried to show simple code instead of fast or abstract code.

3 C# Implementation

These were my first C# programs so they might not be idiomatic or stylistically proper. These examples aren't as complete as the Python and C++ sections, but I hope they're helpful.

Here's a simple graph, and Breadth First Search:

```
using System;
using System.Collections.Generic;


public class Graph<Location>
{
    // NameValueCollection would be a reasonable alternative here, if
    // you're always using string location types
    public Dictionary<Location, Location[]> edges
        = new Dictionary<Location, Location[]>();

    public Location[] Neighbors(Location id)
```

```
    {
        return edges[id];
    }
};


class BreadthFirstSearch
{
    static void Search(Graph<string> graph, string start)
    {
        var frontier = new Queue<string>();
        frontier.Enqueue(start);

        var visited = new HashSet<string>();
        visited.Add(start);

        while (frontier.Count > 0)
        {
            var current = frontier.Dequeue();

            Console.WriteLine("Visiting {0}", current);
            foreach (var next in graph.Neighbors(current))
            {
                if (!visited.Contains(next)) {
                    frontier.Enqueue(next);
                    visited.Add(next);
                }
            }
        }
    }

    static void Main()
    {
        Graph<string> g = new Graph<string>();
        g.edges = new Dictionary<string, string[]>
            {
            { "A", new [] { "B" } },
            { "B", new [] { "A", "C", "D" } },
            { "C", new [] { "A" } },
            { "D", new [] { "E", "A" } },
            { "E", new [] { "B" } }
        };

        Search(g, "A");
    }
}
```

Here's a graph representing a grid with weighted edges (the forest and walls example from the main page):

```
using System;
using System.Collections.Generic;


// A* needs only a WeightedGraph and a location type L, and does *not*
// have to be a grid. However, in the example code I am using a grid.
public interface WeightedGraph<L>
{
    double Cost(Location a, Location b);
    IEnumerable<Location> Neighbors(Location id);
}


public struct Location
{
    // Implementation notes: I am using the default Equals but it can
    // be slow. You'll probably want to override both Equals and
    // GetHashCode in a real project.

    public readonly int x, y;
    public Location(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}


public class SquareGrid : WeightedGraph<Location>
{
    // Implementation notes: I made the fields public for convenience,
    // but in a real project you'll probably want to follow standard
    // style and make them private.

    public static readonly Location[] DIRS = new []
        {
            new Location(1, 0),
            new Location(0, -1),
            new Location(-1, 0),
            new Location(0, 1)
        };
```

```
    public int width, height;
    public HashSet<Location> walls = new HashSet<Location>();
    public HashSet<Location> forests = new HashSet<Location>();

    public SquareGrid(int width, int height)
    {
        this.width = width;
        this.height = height;
    }

    public bool InBounds(Location id)
    {
        return 0 <= id.x && id.x < width
            && 0 <= id.y && id.y < height;
    }

    public bool Passable(Location id)
    {
        return !walls.Contains(id);
    }

    public double Cost(Location a, Location b)
    {
        return forests.Contains(b) ? 5 : 1;
    }

    public IEnumerable<Location> Neighbors(Location id)
    {
        foreach (var dir in DIRS) {
            Location next = new Location(id.x + dir.x, id.y + dir.y);
            if (InBounds(next) && Passable(next)) {
                yield return next;
            }
        }
    }
}


public class PriorityQueue<T>
{
    // I'm using an unsorted array for this example, but ideally this
    // would be a binary heap. There's an open issue for adding a binary
    // heap to the standard C# library: https://github.com/dotnet/corefx/issues/574
    //
    // Until then, find a binary heap class:
    // * https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp
    // * http://visualstudiomagazine.com/articles/2012/11/01/priority-queues-with-c.aspx
```

```
    // * http://xfleury.github.io/graphsearch.html
    // * http://stackoverflow.com/questions/102398/priority-queue-in-net

    private List<Tuple<T, double>> elements = new List<Tuple<T, double>>();

    public int Count
    {
        get { return elements.Count; }
    }

    public void Enqueue(T item, double priority)
    {
        elements.Add(Tuple.Create(item, priority));
    }

    public T Dequeue()
    {
        int bestIndex = 0;

        for (int i = 0; i < elements.Count; i++) {
            if (elements[i].Item2 < elements[bestIndex].Item2) {
                bestIndex = i;
            }
        }

        T bestItem = elements[bestIndex].Item1;
        elements.RemoveAt(bestIndex);
        return bestItem;
    }
}


/* NOTE about types: in the main article, in the Python code I just
 * use numbers for costs, heuristics, and priorities. In the C++ code
 * I use a typedef for this, because you might want int or double or
 * another type. In this C# code I use double for costs, heuristics,
 * and priorities. You can use an int if you know your values are
 * always integers, and you can use a smaller size number if you know
 * the values are always small. */

public class AStarSearch
{
    public Dictionary<Location, Location> cameFrom
        = new Dictionary<Location, Location>();
    public Dictionary<Location, double> costSoFar
        = new Dictionary<Location, double>();
```

```
    // Note: a generic version of A* would abstract over Location and
    // also Heuristic
    static public double Heuristic(Location a, Location b)
    {
        return Math.Abs(a.x - b.x) + Math.Abs(a.y - b.y);
    }

    public AStarSearch(WeightedGraph<Location> graph, Location start, Location goal)
    {
        var frontier = new PriorityQueue<Location>();
        frontier.Enqueue(start, 0);

        cameFrom[start] = start;
        costSoFar[start] = 0;

        while (frontier.Count > 0)
        {
            var current = frontier.Dequeue();

            if (current.Equals(goal))
            {
                break;
            }

            foreach (var next in graph.Neighbors(current))
            {
                double newCost = costSoFar[current]
                    + graph.Cost(current, next);
                if (!costSoFar.ContainsKey(next)
                    || newCost < costSoFar[next])
                {
                    costSoFar[next] = newCost;
                    double priority = newCost + Heuristic(next, goal);
                    frontier.Enqueue(next, priority);
                    cameFrom[next] = current;
                }
            }
        }
    }
}

public class Test
{
    static void DrawGrid(SquareGrid grid, AStarSearch astar) {
        // Print out the cameFrom array
        for (var y = 0; y < 10; y++)
        {
```

```
            for (var x = 0; x < 10; x++)
            {
                Location id = new Location(x, y);
                Location ptr = id;
                if (!astar.cameFrom.TryGetValue(id, out ptr))
                {
                    ptr = id;
                }
                if (grid.walls.Contains(id)) { Console.Write("##"); }
                else if (ptr.x == x+1) { Console.Write("\u2192 "); }
                else if (ptr.x == x-1) { Console.Write("\u2190 "); }
                else if (ptr.y == y+1) { Console.Write("\u2193 "); }
                else if (ptr.y == y-1) { Console.Write("\u2191 "); }
                else { Console.Write("* "); }
            }
            Console.WriteLine();
        }
    }

    static void Main()
    {
        // Make "diagram 4" from main article
        var grid = new SquareGrid(10, 10);
        for (var x = 1; x < 4; x++)
        {
            for (var y = 7; y < 9; y++)
            {
                grid.walls.Add(new Location(x, y));
            }
        }
        grid.forests = new HashSet<Location>
            {
                new Location(3, 4), new Location(3, 5),
                new Location(4, 1), new Location(4, 2),
                new Location(4, 3), new Location(4, 4),
                new Location(4, 5), new Location(4, 6),
                new Location(4, 7), new Location(4, 8),
                new Location(5, 1), new Location(5, 2),
                new Location(5, 3), new Location(5, 4),
                new Location(5, 5), new Location(5, 6),
                new Location(5, 7), new Location(5, 8),
                new Location(6, 2), new Location(6, 3),
                new Location(6, 4), new Location(6, 5),
                new Location(6, 6), new Location(6, 7),
                new Location(7, 3), new Location(7, 4),
                new Location(7, 5)
            };
```

```
        // Run A*
        var astar = new AStarSearch(grid, new Location(1, 4),
                                    new Location(8, 5));

        DrawGrid(grid, astar);
    }
}
```

4 Algorithm changes

The version of Dijkstra's Algorithm and A* on my pages is slightly different from what you'll see in an algorithms or AI textbook.

The pure version of Dijkstra's Algorithm starts the priority queue with all nodes, and does not have early exit. It uses a "decrease-key" operation in the queue. It's fine in theory. But in practice…

1. By starting the priority with only the start node, we can keep it small, which makes it faster and use less memory.
2. With early exit, we almost never need to insert all the nodes into the queue, and we can return the path as soon as it's found.
3. By not putting all nodes into the queue at the start, most of the time we can use a cheap insert operation instead of the more expensive decrease-key operation.
4. By not putting all nodes into the queue at the start, we can handle situations where we do not even know all the nodes, or where the number of nodes is infinite.

This variant is sometimes called "Uniform Cost Search". See Wikipedia[5] to see the pseudocode, or read Felner's paper[6] [PDF] to see justifications for these changes.

There are three further differences between my version and what you might find elsewhere. These apply to both Dijkstra's Algorithm and A*:

5. I eliminate the check for a node being in the frontier with a higher cost. By not checking, I end up with duplicate elements in the frontier. *The algorithm still works.* It will revisit some locations more than necessary (but rarely, in my experience, as long as the heuristic is admissible). The code is simpler and it allows me to use a simpler and faster priority queue that does not support the decrease-key operation. The paper "Priority Queues and Dijkstra's Algorithm" [7] suggests that this approach is faster in practice.
6. Instead of storing both a "closed set" and an "open set", I have a `visited` flag that tells me whether it's in *either* of those sets. This simplifies the code further.
7. I don't need to store a separate open or closed set explicitly because the set is *implicit* in the keys of the `came_from` and `cost_so_far` tables. Since we always want one of those two tables, there's no need to store the open/closed sets separately.
8. I use hash tables instead of arrays of node objects. This eliminates the rather expensive *initialize* step that many other implementations have. For large game maps, the initialization of those arrays is often slower than the rest of A*.

If you have more suggestions for simplifications that preserve performance, please let me know!

5 Optimizations

For the code I present here, I've been focusing on simplicity and generality rather than performance. **First make it work, then make it fast.** Many of the optimizations I use in real projects are specific to the project, so instead of presenting optimal code, here are some ideas to pursue for your own project:

5.1 Graph

The biggest optimization you can make is to explore fewer nodes. My #1 recommendation is that if you're using a grid map, consider using a non-grid pathfinding graph. It's not always feasible but it's worth looking at.

If your graph has a simple structure (e.g. a grid), calculate the neighbors in a function. If it's a more complex structure (either a non-grid, or a grid with lots of walls, like a maze), store the neighbors in a data structure.

You can also save a bit of copying by reusing the neighbors array. Instead of *returning* a new one each time, allocate it once in the search code and pass it into the graph's neighbors method.

5.2 Queue

Breadth First Search uses a simple queue instead of the priority queue needed by the other algorithms. Queues are simpler and faster than priority queues. In exchange, the other algorithms usually explore fewer nodes. In most game maps, exploring fewer nodes is worth the slowdown from the other algorithms. There are some maps though where you don't save much, and it might be better to use Breadth First Search.

For queues, use a deque instead of an array. A deque allows fast insertion and removal on either end, whereas an array is fast only at one end. In Python, see collections.deque[8]; in C++, see the deque[9] container. However, breadth first search doesn't even need a queue; it can use two vectors, swapping them when one is empty.

For priority queues, use a binary heap instead of an array or sorted array. A binary heap allows fast insertion and removal, whereas an array is fast at one or the other but not both. In Python, see heapq[10]; in C++, see the priority_queue[11] container.

In Python, the Queue and PriorityQueue classes I presented above are so simple that you might consider inlining the methods into the search algorithm. I don't know if this buys you much; I need to measure it. The C++ versions are going to be inlined.

In Dijkstra's Algorithm, note that the priority queue's priority is stored twice, once in the priority queue and once in `cost_so_far`, so you could write a priority queue that gets priorities from elsewhere. I'm not sure if it's worth it.

The paper "Priority Queues and Dijkstra's Algorithm"[12] by Chen, Chowdhury, Ramachandran, Lan Roche, Tong suggests optimizing the structure of Dijkstra's Algorithm by not reprioritizing, and it also suggests looking at pairing heaps[13] and other data structures.

If you're considering using something other than a binary heap, first measure the size of your frontier and how often you reprioritize. Profile the code and see if the priority queue is the bottleneck.

My gut feeling is that *bucketing* is promising. Just as bucket sort and radix sort can be useful alternatives to quicksort when the keys are integers, we have an even better situation with Dijkstra's Algorithm and A*. The priorities in Dijkstra's Algorithm are *incredibly narrow*. If the lowest element in the queue has priority `f`, then the highest element has priority `f+e` where `e` is the maximum edge weight. In the forest example, I have edge weights 1 and 5. That means all the priorities in the queue are going to be between `f` and `f+5`. Since they're all integers, *there are only six different priorities*. We could use six buckets and not sort anything at all! A* produces a wider range of priorities but it's still worth looking at. And there are fancier bucketing approaches that handle a wider range of situations.

Note that if all the edge weights are 1, the priorities will all be between `f` and `f+1`. This yields a variant of Breadth First Search that uses two arrays instead of a queue, which I used on my hex grid page[14]. If the weights are 1 or 2, you'll have three arrays; if the weights are 1, 2, or 3, you'll have four arrays; and so on.

I have more note about priority queue data structures here[15].

5.3 Search

The heuristic adds complexity and cpu time. The goal though is to explore fewer nodes. In some maps (such as mazes), the heuristic may not add much information, and it may be better to use a simpler algorithm without a heuristic guide.

Some people use an *inadmissible* (overestimating) heuristic to speed up A* search. This seems reasonable. I haven't looked closely into its implications though. I believe (but don't know for sure) that some already-visited elements may need to be visited again even after they've been taken out of the frontier.

Some implementations *always* insert a new node into the open set, even if it's already there. You can avoid the potentially expensive step of checking whether the node is already in the open set. This will make your open set bigger/slower and you'll also end up evaluating more nodes than necessary. If the open-set test is expensive, it might still be worth it. However, in the code I've presented, I made the test cheap and I don't use this approach.

Some implementations *don't test* whether a new node is better than an existing node in the open set. This avoids a potentially expensive check. However, it also *can lead to a bug*. For some types of maps, you will not find the shortest path when you skip this test. In the code I've presented, I check this (`new_cost < cost_so_far`). The test is cheap because I made it cheap to look up `cost_so_far`.

5.4 Integer locations

If your graph uses integers as locations, consider using a simple array instead of a hash table for `cost_so_far`, `visited`, `came_from`, etc. Since `visited` is an array of booleans, you can use a bit vector. Initialize the `visited` bit vector for all ids, but leave `cost_so_far` and `came_from` uninitialized. Then only initialize on the first visit.

```
vector<uint16_t> visited(1 + maximum_node_id/16);

…
    size_t index = node_id/16;
    uint16_t bitmask = 1u << (node_id & 0xf);
    if (!(visited[index] & bitmask)
        || new_cost < cost_so_far[next]) {
        visited[index] |= bitmask;
        …
    }
```

If you run only one search at a time, you can statically allocate and then reuse these arrays from one invocation to the next. Then keep an array of all indices that have been assigned to the bit vector, and then reset those on exit. For example:

```
static vector<uint16_t> visited(1 + maximum_node_id/16);
static vector<size_t> indices_to_clear;
…
    size_t index = node_id/16;
    uint16_t bitmask = 1u << (node_id & 0xf);
    if (!(visited[index] & bitmask)
        || new_cost < cost_so_far[next]) {
        if (!visited[index]) {
            indices_to_clear.push_back(index);
        }
        visited[index] |= bitmask;
        …
    }
…

for (size_t index : indices_to_clear) {
    visited[index] = 0;
```

```
}
indices_to_clear.clear();
```

(Caveat: I haven't used or tested this code)

6 Troubleshooting

6.1 Wrong path

If you're not getting a shortest path, try testing:

- Does your priority queue work correctly? Try stopping the search and dequeuing all the elements. They should all be in order.
- Does your heuristic ever overestimate the true distance? The `priority` of a new node should never be lower than the priority of its parent, unless you are overestimating the distance (you can do this but you won't get shortest paths anymore).
- In a statically typed language, the cost, heuristic, and priority values need to have compatible types. The sample code on this page works with either integers or floating point types, but not all graphs and heuristics are limited to integer values. Since priorities are the sum of costs and heuristics, the priorities will need to be floating point if *either* costs or heuristics are floating point.

6.2 Ugly path

The most common question I get when people run A* on a grid is *why don't my paths look straight?* If you've told A* that all grid movements are equal, then there are lots of shortest paths of the same length, and it's going to pick one arbitrarily. The path is *short* but it doesn't *look* good.

- One solution is to *straighten* the paths afterwards, using a "string pulling" algorithm.
- One solution is to *guide* the paths in the right direction, by adjusting the heuristic. There are some cheap tricks that don't work in all situations; read more here[16].
- One solution is to *not use a grid*. Tell A* just the places where you might turn, instead of every grid square; read more here.
- One solution is a hack, but it works some of the time. When iterating through neighbors, instead of always using the same ordering (such as north, east, south, west), change the ordering on "odd" grid nodes (those where (x+y) % 2 == 1). **I use this trick on these tutorial pages.**

7 More reading

- Aleksander Nowak has written a **Go version** of this code at https://github.com/vyrwu/a-star-redblob[17]
- Algorithms textbooks often use mathematical notation with single-letter variable names. In these pages I've tried to use more descriptive variable names. Correspondences:
  - `cost` is sometimes written as w or d or l or length
  - `cost_so_far` is usually written as g or d or distance
  - `heuristic` is usually written as h
  - In A*, the `priority` is usually written as f, where $f = g + h$
  - `came_from` is sometimes written as π or parent or previous or prev

- `frontier` is usually called OPEN
- `visited` is the union of OPEN and CLOSED
- locations such as `current` and `next` are called *states* and written with letters u, v

- Wikipedia links:
  - Queue[18]
  - Graph[19]
  - Breadth-First Search[20]
  - (Greedy) Best-First Search[21]
  - Dijkstra's Algorithm[22]
  - A* Algorithm[23]

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

Endnotes

[1]: http://theory.stanford.edu/~amitp/GameProgramming/MovementCosts.html
[2]: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties
[3]: http://theory.stanford.edu/~amitp/GameProgramming/MovementCosts.html
[4]: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties
[5]: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Practical_optimizations_and_infinite_graphs
[6]: https://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4017/4357
[7]: http://www.cs.sunysb.edu/~rezaul/papers/TR-07-54.pdf

[8]: https://docs.python.org/3/library/collections.html
[9]: http://en.cppreference.com/w/cpp/container/deque
[10]: https://docs.python.org/2/library/heapq.html
[11]: http://en.cppreference.com/w/cpp/container/priority_queue
[12]: http://www.cs.sunysb.edu/~rezaul/papers/TR-07-54.pdf
[13]: http://en.wikipedia.org/wiki/Pairing_heap
[14]: https://www.redblobgames.com/grids/hexagons/#range-obstacles
[15]: http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#set-representation
[16]: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties
[17]: https://github.com/vyrwu/a-star-redblob
[18]: http://en.wikipedia.org/wiki/Queue_(abstract_data_type)
[19]: http://en.wikipedia.org/wiki/Graph_(data_structure)
[20]: http://en.wikipedia.org/wiki/Breadth-first_search
[21]: http://en.wikipedia.org/wiki/Best-first_search
[22]: http://en.wikipedia.org/wiki/Dijkstra's_algorithm
[23]: http://en.wikipedia.org/wiki/A*_search_algorithm

RSS Feed

Created 6 Jul 2014 with Emacs Org-mode, from implementation.org. Last modified: 10 May 2019