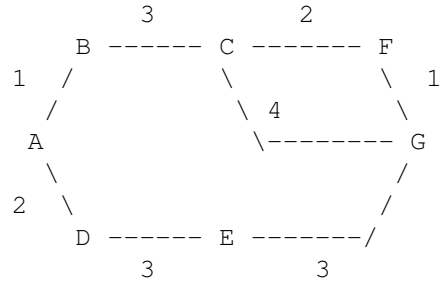
1 Dijkstra's Algorithm

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.




```
Edge weights:
g(A, B) = 1
g(B, C) = 3
g(C, F) = 4
g(C, G) = 4
g(F, G) = 1
g(A, D) = 2
g(D, E) = 3
g(E, G) = 3
```

```
Heuristics:
h(A, G) = 8
h(B, G) = 6
h(C, G) = 5
h(F, G) = 1
h(D, G) = 6
h(E, G) = 3
```

a) Run Dijkstra's Algorithm to find the shortest paths from A to every other vertex. You may find it helpful to keep track of the priority queue and make a table of current distances.

A -> B: 1
A -> C: 4
A -> D: 2
A -> E: 5
A -> F: 6
A -> G: 7

2 Extra for Experts

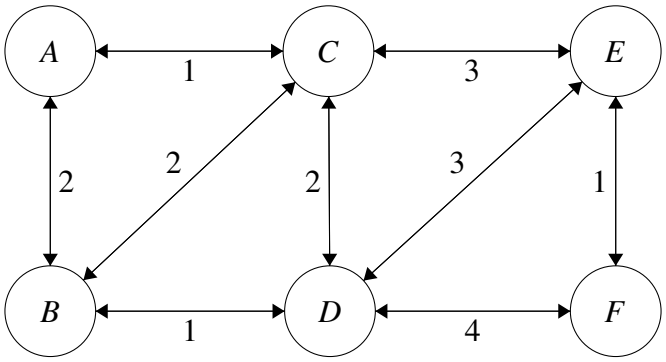
a) Given the weights and heuristic values for the graph below, what path would A* search return, starting from A and with G as a goal?

A* would return A-D-E-G.

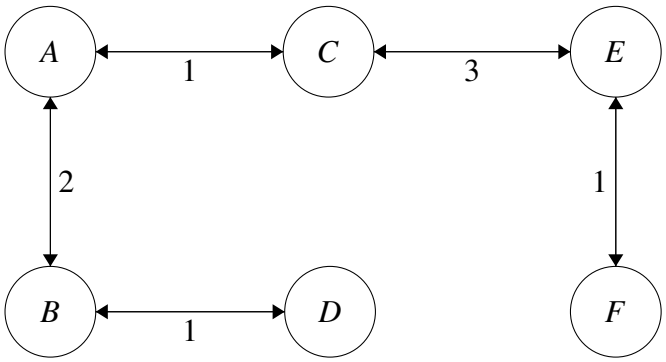
b) Is the heuristic admissible? Why or why not?

A heuristic is admissible if all of its estimations `h(x)` are optimistic. No it's not, because the actual shortest path from `A->G` is of cost 7 if we take the northern route, but the heuristic estimates it will cost 8.

3 Minimum Spanning Trees



a) Perform Prim's algorithm to find the minimum spanning tree of the following graph. Pick A as the initial node. Whenever there are more than one node with the same cost, process them in alphabetical order.



b) Use Kruskal's algorithm to find a minimum spanning tree.

In this case, Prim and Kruskal's output the same MST. This is not always the case.

c) There are quite a few MSTs here. How many can you find?

There are three choices to use an edge of weight 2 that can be used interchangeably and there are two choices of using an edge of weight 3 that can be used interchangeably. So there are 3 * 2 = 6 possible MST's. This math does not always lead to this. The key thing to note is that we could replace one of the weight 2 edges with another weight 2 edge and the entire graph would be spanning. Same for the weight 2 edges

4 Sorting I

Show the steps taken by each sort on the following unordered list:

`106, 351, 214, 873, 615, 172, 333, 564`

(a) Insertion sort. Show the sorted and unsorted portions at every step. 106 | 351 214 873 615 172 333 564
106 351 | 214 873 615 172 333 564
106 214 351 | 873 615 172 333 564
106 214 351 873 | 615 172 333 564
106 214 351 615 873 | 172 333 564
106 172 214 351 615 873 | 333 564
106 172 214 333 351 615 873 | 564
106 172 214 333 351 564 615 873 |

(b) Selection sort. Show the sorted and unsorted portions at every step.

106 | 351 214 873 615 172 333 564
106 172 | 214 873 615 351 333 564
106 172 214 | 873 615 351 333 564
106 172 214 333 | 615 351 873 564
106 172 214 333 351 | 615 873 564
106 172 214 333 351 564 | 873 615
106 172 214 333 351 564 615 | 873
106 172 214 333 351 564 615 873 |

(c) Merge sort. Show the sorted and unsorted portions at every step.

106 351 214 873 615 172 333 564
106 351 214 873 | 615 172 333 564
106 351 | 214 873 | 615 172 | 333 564
106 | 351 | 214 | 873 | 615 | 172 | 333 | 564
106 351 | 214 873 | 172 615 | 333 564
106 214 351 873 | 172 333 564 615
106 172 214 333 351 564 615 873

(c) Use heapsort to sort the following array (hint: draw out the heap). Draw out the array at each step:

`106, 615, 214, 873, 351`

873 615 214 106 351 (turns the array into a valid heap)
615 351 214 106 873 ('delete' 873, then sink 351)
351 106 214 615 873 ('delete' 615, then sink 106)

214 106 351 615 873 ('delete' 351, then sink 214)
106 214 351 615 873 ('delete' 214)
106 214 351 615 873 ('delete' 106)

5 Sorting II

Match the sorting algorithms to the sequences, each of which represents several intermediate steps in the sorting of an array of integers.
Algorithms: Heapsort, insertion sort, selection sort, mergesort

(a)
```
12, 32, 14, 34, 17, 38, 23, 11
32, 17, 23, 11, 12, 14, 34, 38
```

There was a typo in this discussion. The second to last number should be 34 (misprinted as 24). Max heap sort. If we look at the second line, we see that last two numbers are the correct max numbers. If we look at the first six numbers of the second line, this is the array representation of a max heap. For full verification, you should heapify the oriignal list and removeMax twice (for this example) to see if the resulting heap is matches the second line.

(b)
```
23, 45, 12, 4, 65, 34, 20, 43
12, 23, 45, 4, 65, 34, 20, 43
```

Insertion sort. We can see that 23 and 45 stay in place, but 12 gets inserted in the very beginning, and 23 and 45 get scooted over. This is insertion sort in progress, which will insert the 12 into its correct spot relative to the currently sorted region.

(c)
```
12, 7, 8, 4, 10, 2, 5, 34, 14
2, 4, 5, 7, 8, 12, 10, 34, 14
```

Min-selection sort. This one is a bit hard to see. But if we slowly swap out the positions to put the min-element in the correct place, after we place the fifth smallest element, 8, into its correct index, the array will look like the second line.