Assignment 4: The Dijkstra Dance

July 2025

1 Introduction

As I have said many times throughout the semester, there is no better feeling than translating theoretical knowledge about the structure and properties of an algorithm into real, actual code that, amazingly, works the way you think it should! What's even better than that feeling? Not. That. Much!

Well, get ready for a pretty sensational assignment! In this final assignment of the semester you are going to implement Dijkstra's Algorithm for finding the shortest path in a directed, acyclic graph (with edges whose weights are non-negative). Dijkstra's Algorithm is pretty sweet because it gives you the chance to show off twice. Just what do I mean by that? Well, as you know, Dijkstra's Algorithm uses a min-binary-heap. In other words, for you to complete this assignment successfully, you get to build an implementation of two of the conceptual entities that we studied this semester.

Note: It is possible to successfully implement Dijkstra's Algorithm without using a min-binary heap. If, however, you swap the min-priority heap that is central to Dijkstra's Algorithm, with another data structure that provides the same functionality, the runtime will change (and we don't want that!). So, for this implementation of Dijkstra's Algorithm, you must use a min-binary heap!

2 Requirements

To successfully complete this assignment, you must demonstrate a correct implementation (in a language of your choosing) of Dijkstra's Algorithm. You *must* use a min-heap to implement the priority queue (`Black` in the pseudocode in Figure 2) (20 points).

Your demonstration must show the state of the search at each step (i.e., every iteration of the `while` loop in 2) for a path through the three example graphs given below. See Figure 1 for an example (60 points).

Your implementation must meet (or exceed) standards for professionally written code (i.e., it must use descriptive variables and contain comments that are descriptive and grammatically correct) (20 points).

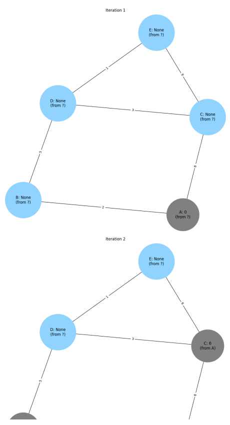

Figure 1: An example of a satisfactory presentation of the internal state of an implementation of Dijkstra's Algorithm as it searches for a path between two nodes.

3 The Algorithm

See Figure 2 for a quick reference for the pseudocode for Dijkstra's Algorithm.

4 Givens

You have been given (see Canvas) some Python skeleton code in a Notebook that has been tested and verified to work on Google's Colab service.[1] In the skeleton, the `PathableGraphVisualizer` class is complete and usable. The `PathableGraphVisualizer` class is good for, well, visualizing the state of an instance of a `PathableGraph`.[2]

`PathableGraph` is designed to make it easy for you to meet the requirements of this assignment. The class' `__init__` method would be a great place to capture references to any variables that your implementation needs from the caller. The `add_node` and `add_edge` would be great methods to implement so that the user can describe the structure of the graph through which they want to calculate paths. The `start_processing` method would be a great place to to do all the required initialization of variables that control the search. The `iteration` method would be a great place to implement the heart of Dijkstra's Algorithm – the `while` loop (see Figure 2).

The shells of the `Heap` and `HeapObject` are there to guide you to a complete, succinct implementation of the min heap that you will need to use for your

[1]If you would like to use another service to run the notebook, I don't believe that will cause too great a problem.

[2]The `PathableGraphVisualizer` interacts with the `PathableGraph` by accessing its `nodes` data attribute (which should be a dictionary that maps the names of nodes (as `strs`) to instances of `Nodes`.

```
dijkstra($G$, $\omega$, $s$):
  for-all $v \in G$
    $v.d = \infty$
    $v.previous = \varnothing$
  Black = $\emptyset$
  Gray = new MinHeap()
  $s.d = 0$
  for-all $v \in G$
    Gray.add($v$)
  while not Gray.empty()
    $u$ = Gray.extract()
    Black.add( $u$ )
    for-each $v \in u.adjacent$
        if $u.d + \omega(u,v) < v.d$
            $v.d = u.d + \omega(u,v)$
            $v.previous = u$
            Black.decrease_priority($v$,$u.d + \omega(u,v)$)
```

Figure 2: Pseudocode for Dijkstra's Algorithm (from [Cor+22]). ω is a function that, given two vertices will return the weight of the edge between them.

implementation of Dijkstra's Algorithm. The stub methods in those classes are there as suggestions – you are under no obligation to implement your min heap according to that interface. Ultimately, the interface to the heap should make sense to you because you are going to be the one using it!

If you do follow this pattern, then capturing a visualization after every invocation of the `iteration` method would satisfy the requirement that you show the state of the search for a path by your implementation at each step (i.e., every iteration of the `while` loop in 2). In fact, the graphic in Figure 1 was made using the code from the skeleton.

5 Go Get 'Em

As usual, if you have *any* questions, please let me know! Good luck!

6 Demonstration Graphs

References

[Cor+22] Thomas H Cormen et al. *Introduction to algorithms*. The Mit Press, 2022.

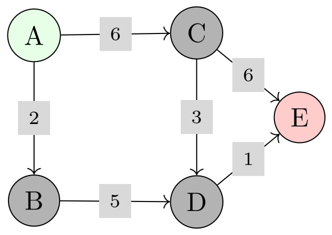

Figure 3: Example 1.

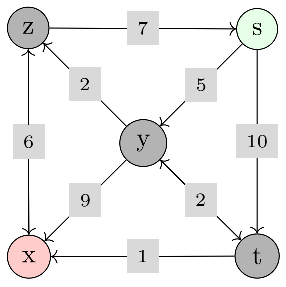

Figure 4: Example 2 (taken from [Cor+22, p. 261]).

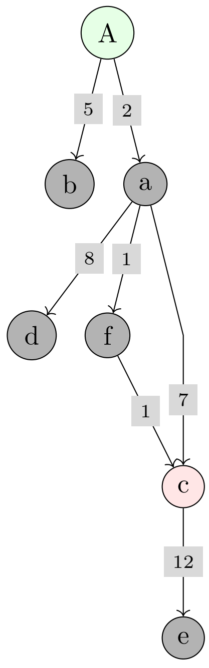

Figure 5: Example 3.