Juniper Pasternak

Professor Gharaibeh

Computer Science 215

14 May 2024

Implementation and Analysis of Dijkstra's Algorithm

Implementation

The simple Dijkstra's algorithm implementation is nearly the same as the pseudo-code provided in Section 9.2 of *Algorithms Illuminated*. This implementation starts by creating a visited list with just the source node and setting the source node length to 0 and all the others to infinity. It then finds the edge from a visited tail to an unvisited head that minimizes the following criterion: the sum of the tail's length and the edge weight. It does this by checking each visited node's outbound edges with unvisited heads. With this edge, the program adds the head to the visited list and sets its length to its respective criterion. The only difference between the provided pseudo-code and this implementation is that the implementation uses a list (dynamic array) while the pseudo-code may suggest the use of a hash-set for near-constant lookup time.

Unlike the former, the heap implementation is significantly altered from Section 10.4's pseudo-code. The start of this implementation is similar: Initialize the lengths as before, create an empty visited list, and initialize a heap. There are some subtle differences though: The pseudo-code dictates that heap keys be separate from the length, but this implementation just uses the length and runs comparisons based on it. Also, unlike the pseudo-code's heap full of every node, this implementation just starts with the source node. It also runs the loop similarly by extracting the min node (call this the current node) from the heap and adding it to visited until the heap is

empty after the loop's body is run. This body is where major differences are found: For every unvisited head of the current node's outbound edges, this code updates the length if the criterion is better and pushes the node to the heap. These modifications were made because a heap which allows arbitrary item deletion would require a custom implementation. This also has an impact on the algorithm's time complexity.

Time Complexity

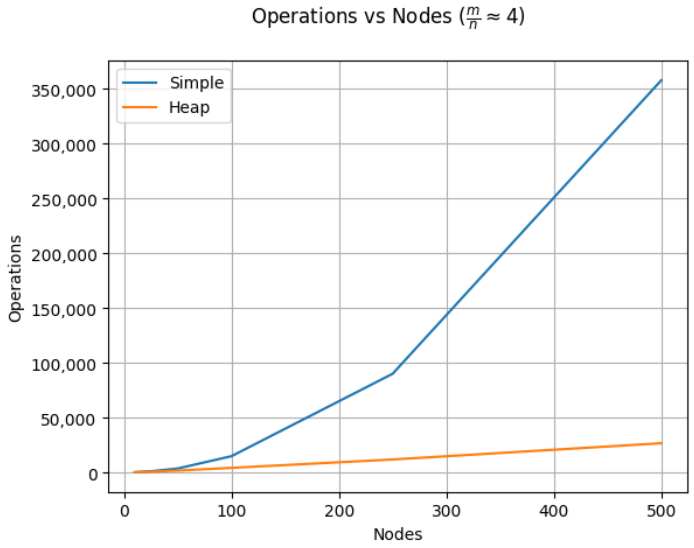
The simple implementation has a time complexity of $O(mn)$, where n is the number of nodes and m is the number of edges. There are two main sections of the code. The first is simply the initialization; this runs a constant number of basic operations n times. Second, the while loop runs n times and finding the best edge requires m operations asymptotically because it checks up to m edges. This second portion asymptotically overpowers the first and results in performing n times m operations.

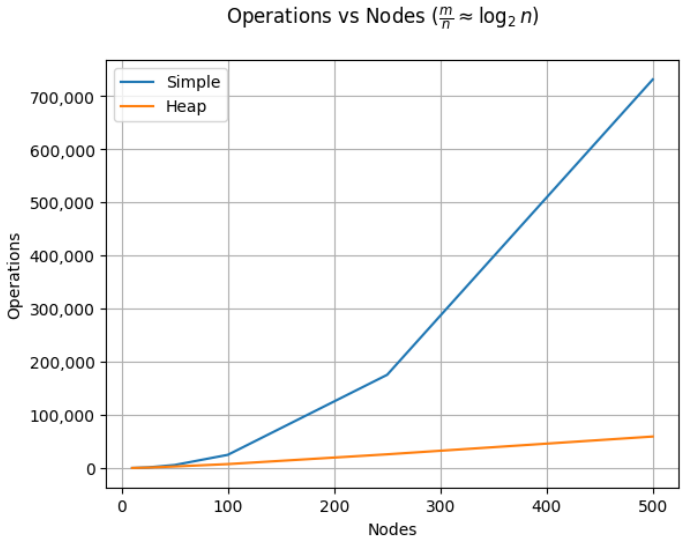
The heap implementation improves upon the former with a time complexity of $O(m \log_2 m)$. Again, initialization requires n operations. The concerning parts of the while loop's body are the heap operations: The min node in the heap will be extracted at most m times; this is because of the next part. Every edge will have a node pushed onto the heap leading to m nodes to both push and extract. How long does each heap operation take? Asymptotically speaking, the logarithm of the heap size, or $\mathrm{O}(\log n_{heap})$. The heap will have $O(m)$ nodes in it in the worst case. Therefore, there are m times $\log m$ operations. This is worse than implementations that restrict the heap size to n through more difficult means.

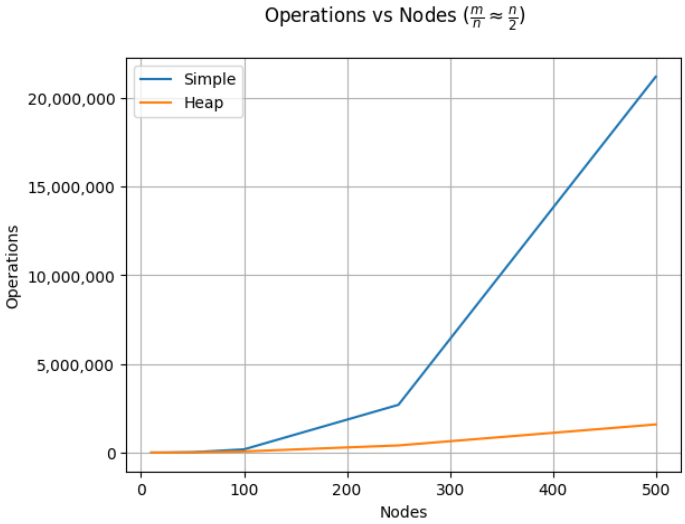
Empirical Analysis

For an empirical analysis of the performance of both algorithms, it is important to note the methodologies used. First, operations are a somewhat arbitrary category of anything that runs in constant time. When measuring operations with a heap, this analysis uses the approximation of $\log_2 n_{heap} + 1$ operations per heap push or pop. Second, this analysis created three different graph datasets. Each dataset contains six graphs with sizes from 10 to 500 nodes. The number of edges varies based on the dataset since each dataset has its own density: a constant (4) number of outbound edges per node resulting in $m = \Theta(n)$, logarithmic edge count per node resulting in $m = \Theta(\log_2 n)$, and finally linear $(0.5n)$ edge count per node resulting in $m = \Theta(\mathrm{n}^2)$. Note that the edge count was not exact and was normally distributed with $\sigma = 0.2\frac{m}{n}$ for real world variance. Finally, the edges were given evenly distributed weights from 1 to 50.

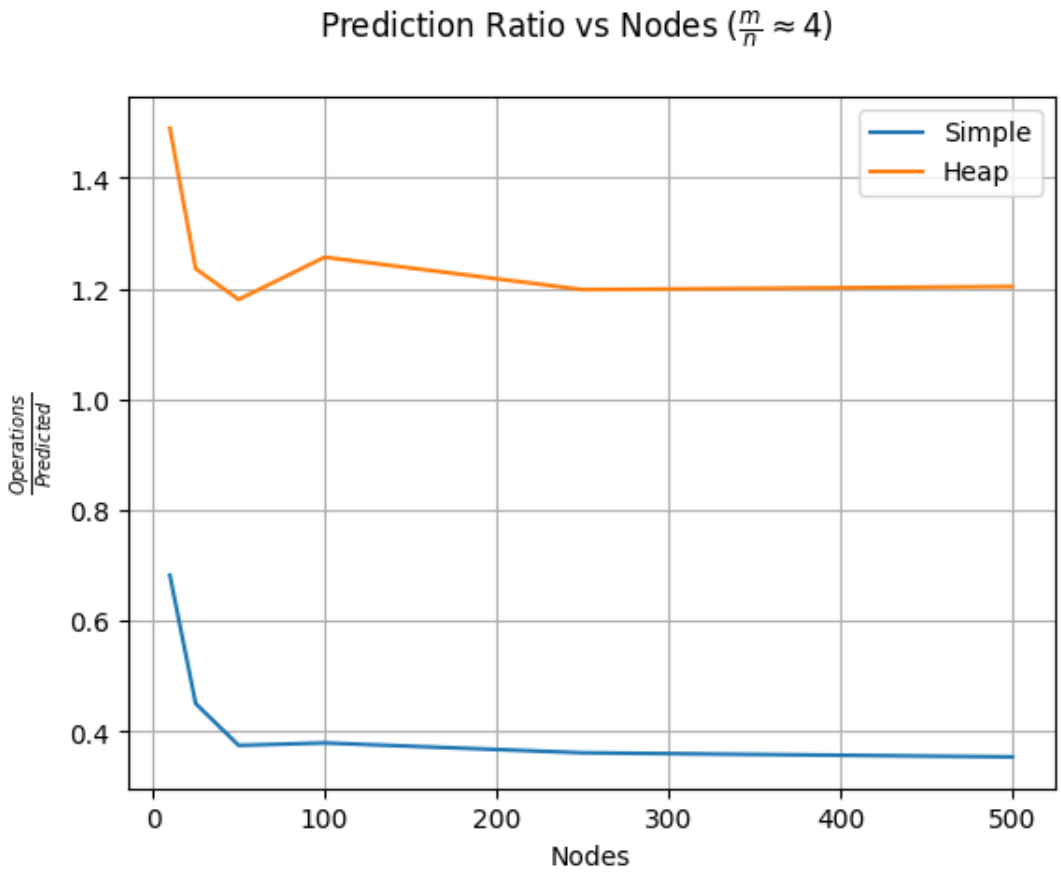
The following plots show the number of operations each implementation of Dijkstra's algorithm performed as a function of the input size n of the graph. Visually, the plots seem to be quadratic and quasi-linear for the simple implementation and heap implementation respectively. But a more rigorous analysis should be used.

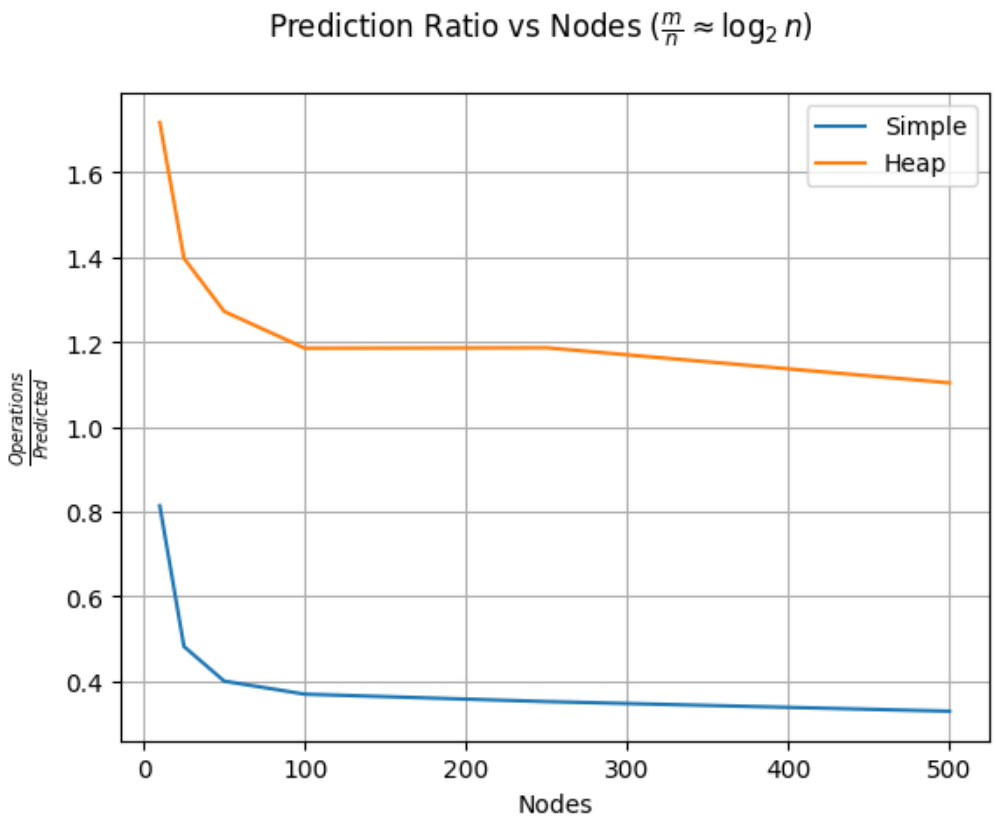

Operations vs Nodes ($\frac{m}{n} \approx 4$)
Simple
Heap
Operations
Nodes
0
50,000
100,000
150,000
200,000
250,000
300,000
350,000
0
100
200
300
400
500

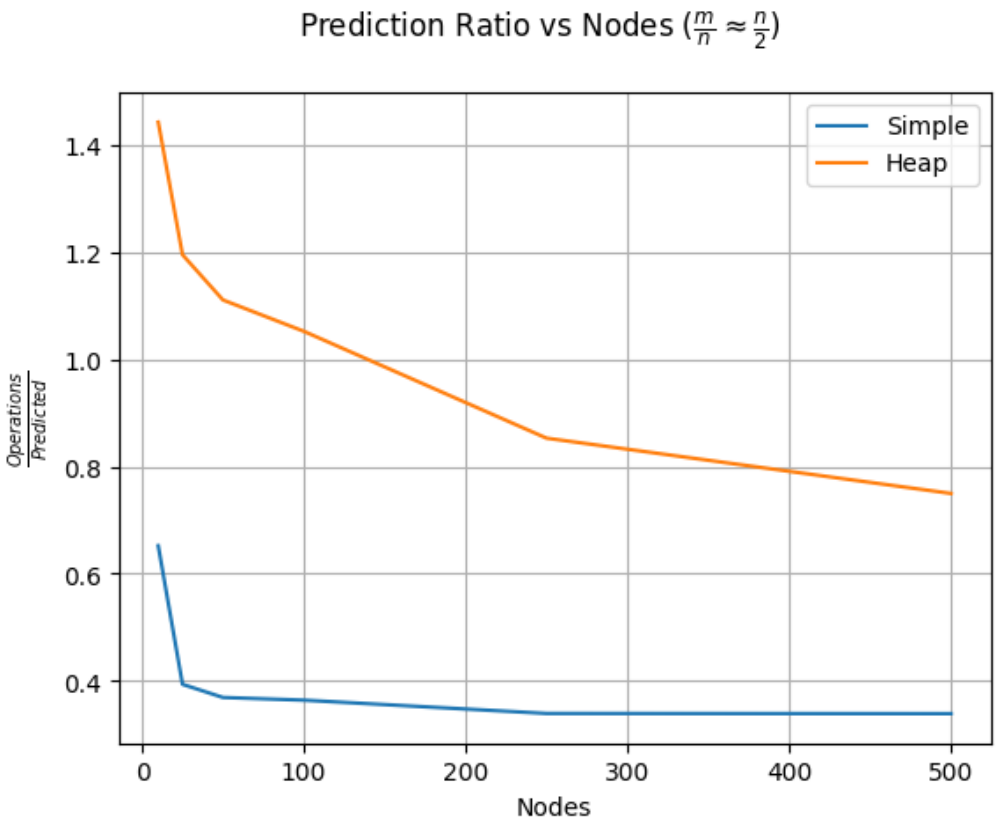

Operations vs Nodes ($\frac{m}{n} \approx \log_2 n$)
Simple
Heap
Operations
Nodes
0
100,000
200,000
300,000
400,000
500,000
600,000
700,000
0
100
200
300
400
500


Operations vs Nodes ($\frac{m}{n} \approx \frac{n}{2}$)
Simple
Heap
Operations
Nodes
0
5,000,000
10,000,000
15,000,000
20,000,000
0
100
200
300
400
500

Plotting the ratio of the actual operations to the predicted operations (prediction ratio) is a more rigorous way to analyze the time complexities in practice. This ratio simply divides the measured operations by a prediction calculated with the discussed time complexities in the former section. This analysis also considers the edges m through this calculation that were previously unseen other than the density groups. As shown below, the ratio demonstrates that the time complexity for the simple implementation is very accurate since it trends towards a constant. In contrast, as the density of the graph increases, the heap implementation seems to become significantly *better* than predicted.




Prediction Ratio vs Nodes ($\frac{m}{n} \approx \log_2 n$)
Simple
Heap
1.6
1.4
1.2
1.0
0.8
0.6
0.4
Operations / Predicted
0
100
200
300
400
500
Nodes


Prediction Ratio vs Nodes ($\frac{m}{n} \approx \frac{n}{2}$)
Simple
Heap
1.4
1.2
1.0
0.8
0.6
0.4
Operations / Predicted
0
100
200
300
400
500
Nodes