Heap Data Structure Implementations in Python

Code: https://github.com/Ginotuch/cs380-heap-structures

XXXXXX XXXXXX (Ginotuch)
xxxxxxx, XXXXXXXXXXX

Supervisor: Dr. Michael J. Dinneen

2020-02-16

Abstract

Heaps are an old data structure that have been widely used and implemented. This paper goes into detail describing modifications that can be made on the original heap to allow for more desirable features. Cases are displayed, demonstrating the issues of using a normal heap, along with solutions that come from these new modifications. A Python library, *HeapQueue*, was developed to implement these new features, showing their usability. Performance of this new library was then looked at, comparing it to that of a basic heap and the already existing Python library *heapq*.

CONTENTS

1 INTRODUCTION

A heap is one of the most basic and commonly taught data structures in computer science. Based on trees from graph theory, heaps take advantage of many properties in trees, giving them unique functionality and low operation cost that can be used in a wide range of applications. Generally, heaps are an array based implementation of a binary tree, originally defined by J. W. J. Williams in 1964 [2] for use in the sorting algorithm heapsort. Given an array of n elements, heapsort is able to sort them in-place with a running time of $O(n \log n)$, as shown in Section 2.2.3.

Heaps can also be used to store an array of n elements in a way that the element of minimum value can be found in $O(1)$ time. They also have the advantage of being able to add new elements in $O(\log n)$ time, avoiding the normal $O(n)$ required operations when inserting an element into a sorted array, further described in Section 2.1. Because of these running times, heaps are a perfect candidate to be used as the basis for priority queues.

The majority of heap implementations are basic and do not provide advanced features such as positional tracking, which will be later discussed covering its advantages and disadvantages. First the basic graph theory needed to understand heaps is covered, followed by an overview of heaps themselves. The purpose of a heap in Dijkstra's path finding algorithm will also be talked about, and how it can be improved upon with positional tracking.

Next, to further look at positional tracking within heaps, a Python library titled *HeapQueue* was developed as an example of implementation, along with the newly possible functions. Finally, the performance of different versions of these heaps, such as the Python standard library of heap called *heapq*, are compared

1.1 GRAPH DEFINITIONS

The majority of the following definitions are taken from *Discrete Structures in Mathematics and Computer Science* [6] and *Introduction to Algorithms and Data Structures* [1].

DEFINITION 1 (Graphs, Paths, Cycle, length) A *simple undirected* ***graph*** $G = (V, E)$ consists of V, a nonempty set of nodes, and E, a set of unordered pairs of distinct elements of V called edges.

A ***path*** of length n from x_0 to x_n, where n is a non-negative integer in a simple undirected graph is denoted by a sequence of distinct nodes $x_0, x_1, \ldots, x_n$.

The path is a ***Cycle*** if it begins and ends at the same node, that is, if $x_0 = x_n$.

The ***length*** of a path or cycle is equal to the number of edges contained within that path or cycle.

DEFINITION 2 (Trees, Roots, Parents, Children, Siblings, Leaf) A ***tree*** is a connected undirected graph with no cycles.

Some node within a tree can be designated the ***root*** of the tree, this represents the top (or base depending on viewpoint) of the tree, generally shown at the top in diagrams.

If v is a node in some tree other than the root, the ***parent*** of v is the unique node u such that there is an edge from u to v, and a path from the root to v that passes through u.

When u is the parent of v, v is called a ***child*** of u.

A node of a tree is called a ***leaf*** if it has no children.

DEFINITION 3 (Level, Height) The ***level*** of a node v in a rooted tree is the length of the unique path from the root to this node. The level of the root is defined to be zero.

The ***height*** of a rooted tree is the maximum of the levels of nodes. This is equal to the maximum length of all paths from the root to any leaf.

DEFINITION 4 (m-ary Tree, Full, Complete) A rooted tree is called an ***m-ary*** tree if every node has no more than m children. An m-ary tree with $m = 2$ is called a ***binary tree***.

In a ***full*** m-ary tree, all vertices have either 0, or m children.

In a ***complete*** binary tree of height h, all levels are full except possibly the bottom level in which children are filled from left to right.

1.2 HEAP DEFINITIONS

DEFINITION 5 (Binary Heap, Max-Heap, Min-Heap) A ***binary heap*** is a complete binary tree that holds exactly one of two properties:

- The value of every child in the tree is less than or equal to the value of their parent. Having this property means the binary heap is considered a ***max-heap***.
- The value of every child is greater than or equal to that of its parent which is a ***min-heap***.

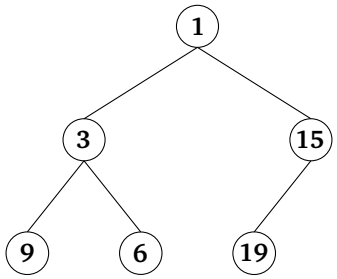


Figure 1: Example min-heap

EXAMPLE 1.1 The graph in Fig. 1 is a binary heap with the min-heap property. Note that the value of a node has no relationship with the values of other nodes on the same level, or their relative order, only the min-heap property described in Definition 5 must hold.

2 HEAP OVERVIEW

2.1 MOTIVATION

To understand why heaps are of importance, we look at how its application can improve performance.

The most basic concept is wanting to find the minimum value from the set of n elements. A straight forward approach would be to store these elements in an array A and create a function $min(A)$ which can be called to return the smallest element. The issues arise when actually implementing $min(A)$. One

approach would be to scan each element in A from $A[0]$ to $A[n-1]$, keeping track of the smallest value seen so far. Then, once the end of the array has been reached, it returns the minimum. This requires one comparison for each of the n elements so the complexity of $min(A)$ is $O(n)$.

This could be improved by keeping A in sorted ascending order. This way, $A[0]$ is always the minimum, and nothing needs to be scanned. Thus, $min(A)$ has a running time of $O(1)$. The downside to this is if a new element needs to be added to the array, it would take $O(\log n)$ time to do a binary search to find the correct position for this new element, and then $O(n)$ to move all the elements to the right when inserting the new element.

Both solutions still have operations with complexity $O(n)$ but by using a min-heap, we can lower the insertion to a complexity of $O(\log n)$ and $min(A)$ to $O(1)$. Conversely, finding the maximum element can similarly be solved with a max-heap with the same running time improvements.

Having this ability to find the lowest valued element efficiently means that a heap is perfect to use as a priority queue. From now on, this use case will be the primary focus when talking about heaps. For clarity, when discussing the attributes of elements stored in the heap, we will use two terms. The ***priority*** of an element is the value which it is sorted and compared to other elements by. The ***key*** is the piece of information that is stored with the element, such as a node's index number for a graph traversal algorithm, or a PID in an operating system. Generally, when implemented, given some element e, its priority p, and key k, we say e is represented as the ordered pair (p,k).

To avoid confusion when discussing a min-heap, given some elements x and y with the priorities p and q respectively, we say that x has a higher priority to y if and only if $p \leq q$. Similarly, in a max-heap, a higher priority refers to an element with a higher priority value.

2.2 Basic Binary Heap Implementation

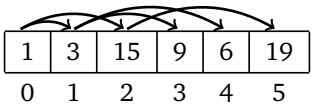
2.2.1 Initialization

There are multiple data structures that can be used to store a heap, such as using pointers or an array. In this case, a one dimensional array will be used to store the elements while making sure to keep an easily modifiable structure to perform basic operations on the heap, such as $O(1)$ minimum (or maximum) finding.

Given a heap containing n nodes, we will want a method to store them and their connecting edges. First, we create an array A of length n, with indices from 0 to $n-1$, such that $A[0]$ is the first element and $A[n-1]$ is the n^{th} element. The root is given the index $i = 0$, and is placed at $A[0]$. Given any node's index i with $0 \leq i < n$, we define the following rules:

- $A[i]$ returns the element at index i.
- $2i+1$ is the index of its left child.
- $2i+2$ is the index of its right child.

Example 2.1 Looking at Fig. 1, which has $n = 6$ nodes, we create an array of length 6 displayed in the below table with the corresponding edges drawn on top and indices below.



2.2.2 Operations

Heaps have a few common basic operations we will cover that all maintain the min-heap or max-heap properties:

- ***Push***: inserts a new elements into the heap.
- ***Peek***: returns the root[1] without removing it from the heap.
- ***Pop***: returns the root and removes it from the heap.

The most basic of all these being *peek*, which just returns the element at index 0 of a given heap A.

The other two operations, *Push* and *Pop*, are more complicated and rely on some internal operations generally not visible to an end user. The two operations are *Heapify Up* and *Heapify Down*. These will move a given element in the heap's array by ***percolating*** up or down the tree swapping with elements until it reaches a position which restores the desired heap property. Both *Heapify Up* and *Heapify Down* rely on the *Swap* operation. The following algorithm definitions are for a min-heap; in the case of a max-heap, the element comparisons are just swapped.[2]

Note on pseudo code: *Arrays and dictionaries are assumed to be passed by reference and modified in-place and therefore will not be returned from the function. Functions will only return newly instantiated variables such as in Pop and HeapSort. An empty* ***return*** *statement exits the function returning nothing.*

Algorithm 1 Swap

Require: $0 \leq j < n$, $0 \leq k < n$

```
1: function SWAP(array A[0,...,n−1], integer j, integer k)
2:     t ← A[j]
3:     A[j] ← A[k]
4:     A[k] ← t
5:     return
```

Algorithm 1 shows the pseudo code for the *Swap* operation that just swaps two elements in the heap's array by their given indices.

Algorithm 2 Heapify Up

Require: $0 \leq i < n$

```
1: function HEAPIFYUP(array A[0,...,n−1], integer i)
2:     while i > 0 do
3:         parent_index ← ⌊(i−1)/2⌋
4:         if A[i] < A[parent_index] then
5:             swap(A, parent_index, i)
6:         else
7:             return
8:         i ← parent_index
```

Algorithm 2, showing *Heapify Up*'s code, continuously percolates an element up the heap by swapping it with its parent until it is larger than or equal to its parent.

[1] This being the minimum or maximum element if it's a min-heap or max-heap respectively.

[2] i.e. in Algorithm 2 $<$ becomes $>$ (line 4). In Algorithm 3 $\geq$ becomes $\leq$ (line 6), and $>$ becomes $<$ (line 8)

Algorithm 3 Heapify Down

Require: $0 \leq i < n$

```
1:  function HEAPIFYDOWN(array A[0,...,n−1], integer i, integer n)
2:      while 2i + 1 < n do
3:          left_child_i ← 2i + 1
4:          right_child_i ← 2i + 2
5:          swap_child_i ← left_child_i
6:          if right_child_i < n and A[left_child_i] ≥ A[right_child_i] then
7:              swap_child_i ← right_child_i
8:          if A[i] > A[swap_child_i] then
9:              swap(A, swap_child_i, i)
10:         else
11:             return
12:         i ← swap_child_i
```

Similar to Algorithm 2, Algorithm 3 will percolate an element down the heap, swapping with its smallest child until both its children are larger than or equal to it, or it becomes a leaf. Note that Algorithm 3 only swaps an element with its smallest child. This is done for the case where one of the element's children, say v, is larger than the other, w, it ensures that v will not get swapped and become the parent of w, which would go against the desired heap property.

LEMMA 2.1 *The height of a complete binary tree with n nodes is at most $\lfloor \log n \rfloor$.*

Proof (from [1, p. 57]). Depending on the number of nodes at the bottom level, a compete binary tree of height h contains between 2^h and $2^{h+1} - 1$ nodes, so that $2^h \leq n < 2^{h+1}$, or $h \leq \log n < h + 1$. □

LEMMA 2.2 *Both Heapify Down and Heapify Up are at worst $O(\log n)$*

Proof. Both *Heapify Down* and *Heapify Up* clearly take $O(\log n)$ time as the most an element can move up or down is the height of the tree, which is shown in Lemma 2.1 to be equal to $\lfloor \log n \rfloor$. □

Algorithm 4 Push

```
1: function PUSH(array A[0,...,n−1], element x)
2:     A.append(x)          ▷ Increases the size of A from n to n + 1, then places x at index n.
3:     heapifyup(A, n)
4:     return
```

Push, shown in Algorithm 4, is one of the most basic functions, entirely built on the *Heapify Up* operation by placing the new element at the end of the array, then moving it up the array and finally restoring the heap property.

LEMMA 2.3 *Push is $O(\log n)$ in the worst case.*

Proof. *Push* places a given new element at the final position in the array in constant time, then calls *Heapify Up* on that position. By Lemma 2.2 we know *Heapify Up* is $O(\log n)$, and since *Push* only adds constant operations ontop of *Heapify Up* it is clear that *Push* $O(\log n)$. □

Algorithm 5 Pop

```
1: function POP(array A[0,...,n−1])
2:     popped_element ← A[0]
3:     A[0] ← A[n−1]
4:     A ← A[0,...,n−2]          ▷ Reduce the size of the array by 1, removing the last element
5:     heapifydown(A, 0)
6:     return popped_element
```

Algorithm 5 removes the root of the tree, which is the element at index 0 in the array representation of the heap. It then calls *Heapify Down* to restore the heap property by moving the element down to a position where it is greater than or equal to both its children, or to become a leaf.

LEMMA 2.4 *Pop is $O(\log n)$ in the worst case.*

Proof. *Pop* removes the root by replacing it with the final element in the array, which takes constant time, then calls *Heapify Down* on the new root. By Lemma 2.2 we know *Heapify Down* is $O(\log n)$ so it is clear that *Pop* is also $O(\log n)$. □

2.2.3 BUILDING A HEAP

Algorithm 6 Heapify

```
1: function HEAPIFY(array A[0,...,n−1], integer n)
2:     for i ← ⌊n/2⌋ − 1; i ≥ 0; i - - do
3:         heapifydown(A, i)
4:     return
```

To build an array from a given heap, we use the algorithm described in Algorithm 6. This works by looking at all nodes that have children[3], which is all nodes with indices from 0 to $\lfloor \frac{n}{2} \rfloor - 1$, then going in reverse from the bottom of the tree and working upwards using *Heapify Down* on each node.

Normally, you would assume that *Heapify* would take $O(n \log n)$ time given that *Heapify Down* itself runs in $O(\log n)$, and is called for $\lfloor \frac{n}{2} \rfloor$ nodes which is linearly proportional to n. Surprisingly, the running time can be more tightly shown to be $\Theta(n)$.

To understand this, we look at the heap slightly differently with each node being the root of a subtree. First we look at the root of the subtree containing the entire heap, which has height h, and then each of its children being roots of their own subtrees with height $h-1$ (or possibly $h-2$ for the right child subtree).

Running *Heapify Down* on the root of a tree with height h results in at most h swaps to get to the bottom of the tree. This then needs to be done for each child subtree, each with at most height $h-1$. From this we get can get the recurrence relation for *Heapify* on a heap with height h: $T(h) = 2T(h-1) + ch$ where c is the constant extra operations per swap of an element down the heap[4]. After looking at some example numbers for h, the formula $T(h) = c(2^{h+1} - h - 2)$ can be derived but must be proved correct for all values $h \geq 0$. [5]

[3] There is no point at looking at leaves as they cannot be moved down any further.

[4] Depending on implementation $c = 3$. 2 for the comparison for each child before a swap, and 1 for the swap itself.

[5] The height of a tree cannot be less than 0.

PROPOSITION 2.1 *The recursion relation $T(h)$ defined*

$$T(h) = \begin{cases} 2T(h-1)+ch, & \text{if } h \geq 2. \\ 1c, & \text{if } h = 1. \\ 0, & \text{if } h = 0 \end{cases}$$

can be solved by the formula $T(h) = c(2^{h+1} - h - 2)$.

Proof. Proved by induction on the height of a heap $h \in \mathbb{N}$, with $0 \leq h$.

Base Cases: $T(0) = 0$ is equal to $c(2^{0+1} - 0 - 2) = 0$
$T(1) = 1c$ is equal to $c(2^{1+1} - 1 - 2) = 1c$
$T(2) = 2T(1) + 2c = 2c + 2c = 4c$ is equal to $c(2^{2+1} - 2 - 2) = 4c$
Inductive Case: Assume for all values up to h that $T(h) = 2T(h-1)+ch$ is equivalent to $c(2^{h+1}-h-2)$, now to prove for $h+1$.
From the relation we know $T(h+1) = 2T(h) + c(h+1)$ and want to show it is equivalent to $c(2^{(h+1)+1} - (h+1) - 2)$. First substituting for $T(h)$:

$$\begin{aligned} T(h+1) &= 2T(h) + c(h+1) \\ &= 2c(2^{h+1} - h - 2) + c(h+1) \\ &= c2^{h+2} - 2ch - 4c + c(h+1) \\ &= c(2^{h+2} - 2h - 4 + h + 1) \\ &= c(2^{h+2} - h - 3) \end{aligned}$$

From the above working we have shown that the formula holds.

Thus, by the principle of mathematical induction, Proposition 2.1 has been proven. □

PROPOSITION 2.2 *With $f(n) = 2^{\lfloor \log n \rfloor + 1}$, $f(n) \in O(n)$*

Proof. Let $f(n) = 2^{\lfloor \log n \rfloor + 1}$, and let $g(n) = cn$. For $f \in g$ to be proven true, by definition of Big O notation, it must be shown that $2^{\lfloor \log n \rfloor + 1} \leq cn$ for some $c > 0$.

$$\begin{aligned} 2^{\lfloor \log n \rfloor + 1} &\leq cn \\ 2^{\lfloor \log n \rfloor + 1} &\leq c \cdot 2^{\log n} \\ 2^{\lfloor \log n \rfloor} &\leq c \cdot 2^{\log n - 1} \end{aligned}$$

Setting $c = 2$,

$$\begin{aligned} 2^{\lfloor \log n \rfloor} &\leq 2^{\log n} \\ 2^{\lfloor \log n \rfloor + 1} &\leq 2n \end{aligned}$$

Therefore $f(n) \in O(n)$.

□

THEOREM 2.1 *Building a heap using the Heapify operation given in Algorithm 6 takes $\Theta(n)$ time.*

Proof. By Proposition 2.1 we know the recurrence relation for *Heapfiy* is solved by the formula $c(2^{h+1} - h - 2)$. We also know from Lemma 2.1 that a heap with n nodes has height $h = \lfloor \log n \rfloor$. Thus, by Proposition 2.2 and substituting h in the formula, we get $c(2^{\lfloor lg n \rfloor + 1} - \lfloor \log n \rfloor - 2) \leq cn - c\lfloor \log n \rfloor - 2c$, which gives us an asymptotic upper bound of $O(n)$. We also know that *Heapify Down* is also called for at least $\lfloor \frac{n}{2} \rfloor$ nodes, which is linearly proportional to n, therefore Algorithm 6 has a lowerbound of $\Omega(n)$. Since *heapify* is both $O(n)$ and $\Omega(n)$, it follows that it is also $\Theta(n)$. □

3 HEAP APPLICATION EXAMPLES

The basic heap that has been described is incredibly useful and is widely implemented in most programming languages. Be that for sorting, as a priority queue, or as a backbone to some higher level algorithm.

3.1 SORTING

Algorithm 7 HeapSort

```
1: function HEAPSORT(array A[0, ..., n − 1], integer n)
2:     heapfiy(A)
3:     array B[0, ..., n − 1]
4:     for i ← 0; i < n; i++ do
5:         B[i] ←pop(A)
6:     return B
```

The original application of a heap to sort arrays is called *HeapSort*, whose pseudo code is described in Algorithm 7. Sorting is easily achievable using the basic operations already described. Given some array A, call *Heapify* on A, then repeatedly *Pop* the root element into a new array, B, until A is empty. Doing so results in the sorted array B. Being that *Heapify* takes $O(n)$ time, and then calling *Pop* for each element is $O(n \log n)$, it is easy to see that *heapsort* is $O(n + n \log n) \iff O(n \log n)$.

3.2 DIJKSTRA

Dijkstra's path finding algorithm is a very commonly taught graph exploration algorithm for finding a path between two nodes that is of the lowest cost on a graph containing no negative edge weights. It can easily be made to run on a directed or undirected graph, can take into account whether edge weights exist, and can also be modified to handle multigraphs[6].

There are two versions of Dijkstra. The first version does not rely on a heap and constantly iterates through all nodes to find the low cost node. This version had a rather poor running time of $\Theta(n^2)$ [1, p. 158] where n is the number of nodes in the graph. The other version uses a min-heap based priority queue to easily lookup the best currently known node to next explore and has a much improved running time of $O((n + m) \log n)$ [1, p. 159] where m is the number of edges.

[6] A multigraph is a graph that can have more than one edge between a pair of nodes.

3.2.1 Priority Queue Based Dijkstra

Algorithm 8 Dijkstra

```
1:  procedure DIJKSTRA(graph G = (V,E), node s ∈ V, node v ∈ V, function cost: V × V → ℝ⁺ ∪ {∞})
2:      array pq                          ▷ The array which is used as the heap based priority queue
3:      array color[0,...,n−1]
4:      array dist[0,...,n−1]
5:      for u ∈ V do
6:          color[u] ← WHITE
7:          dist[u] ← ∞
8:      dist[s] ← 0
9:      color[s] ← BLACK
10:     push(pq, (0,s))                   ▷ (0,s) ensures it is sorted first by cost, 0, then by index, s
11:     while pq is not empty do
12:         t1, u ← pop(pq)               ▷ t1 is the cost, u is the node's index.
13:         if color[u] = BLACK then
14:             Skip to next loop
15:         for x ∈ {x : x is adjacent to u} do
16:             t2 ← t1 + cost(u,x)       ▷ t2 is the cost from s to x through u
17:             if color[x] = WHITE then
18:                 color[x] ← GREY
19:                 push(pq, (t2,x))
20:                 dist[x] ← t2
21:             else if color[x] = GREY and dist[x] > t2 then
22:                 push(pq, (t2,x))
23:                 dist[x] ← t2
24:         color[u] ← BLACK
25:         dist[u] ← t1
26:     return dist[s]
```

Dijkstra's algorithm as described in Algorithm 8, also referred to as PFS Dijkstra, uses a priority first search approach, exploring nodes that have the highest priority first[7]. The priority of each node is the cost of the best known path to that node. The cost being the number of edges in the path for an unweighted graph, or the sum of all edge weights in the path for a weighted graph.

The algorithm works by first looking at each node adjacent to the source node s, then adding each of those nodes to the priority queue with the priority being the cost of that edge. Then it repeats the process of grabbing the lowest cost node that's in the priority queue, say u, and then exploring u's adjacent nodes. If the cost of traveling from s to some adjacent node through u is lower than the currently known cost, then that adjacent node is given the new value and added back to the heap.

The key improvement over the non priority based version is that it can retrieve the current best node in $O(\log n)$ time, instead of looping through all nodes to find the best cost.

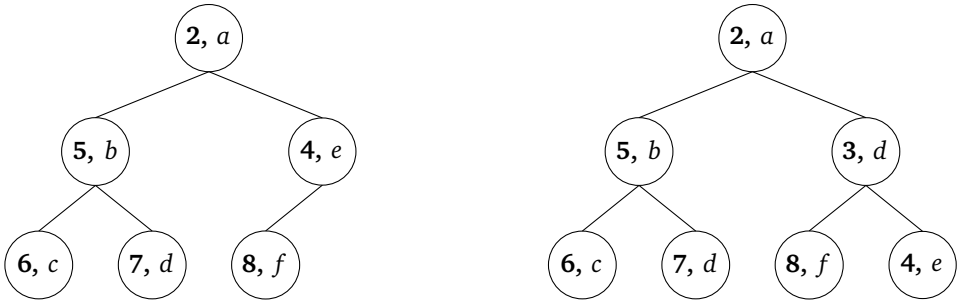
[7] Since it uses a min-heap, the highest priority is the one with the lowest value.

3.3 Issues

Algorithm 8 is written in such a way that it can work on top of the previously described heap in Section 2.2. Although the algorithm is correct, there are issues with using this implementation of a heap which can be improved on to vastly increase performance, as will be described in Section 4.

There are two main issues. When finding a new and better cost to a node, Algorithm 8 will add a duplicate entry to the heap. While this does not cause issues, as it is always of a lower cost to the already existing entries[8], it means the heap is larger than it needs to be. The duplicate entries are handled on line 13 where it skips over nodes that have since been fully explored and are left in the heap because of their old (worse) distance values. The other issue arises from the first issue. When the array stores duplicate entries, it means the heap is larger than it needs to be. Thus, when pushing a new node on to the heap, it takes a longer amount of time because it needs to traverse up the heap past all the duplicate nodes.

These issues aren't limited to Dijkstra itself, it exists in any application that requires changing the priority of a node that is already in the heap.



(a) Before element update

(b) After adding d with updated priority

Figure 2: Duplicate entries in the heap from attempting to modify a node's priority.

Example 3.1 Shown in Fig. 2a is a tree representation of an example priority queue that could exist during a snapshot of Algorithm 8 running. Each node is shown with its priority, then a letter representing the node's index. To demonstrate the duplicate entry issue we take element d. If a new path within the graph was found to d which is of lower cost, say 3, we then push d with its new cost onto the heap. Fig. 2b shows how although the heap property has returned, the old version of d is still present in the heap and only will be removed later once the heap begins to empty[9].

Another problem to note is that this method of updating a priority only works if the new priority is higher than it was previously (meaning its priority is of lower value than it was previously in a min-heap). If you are wanting arbitrary updates where the priority could be increased or decreased, this does not work without keeping track of the most recent priority in a separate variable (or array/dictionary to track all elements easily) to verify the popped element is of correct priority.

This issue not only slows down the heap in general, it also changes the time complexity of its operations. Given that *Push*'s running time is normally $O(\log n)$, with the existence of duplicate elements from updates for every push of an element, it must traverse up the heap past all the previous duplicate entries to reach (at worst) the root.

[8] Dijkstra will only update a value of a node in the heap if it finds one of lower cost, thus it is always higher in the heap and will be popped before any of the duplicates with incorrect priorities.

[9] This is because of the if statement on line 13 in Algorithm 8

LEMMA 3.1 *When allowing duplicates, Push's running time in the worst case is* $\Omega(\log(n+r))$ *with* r *duplicates.*

Proof. By Lemma 2.3 we know that without allowing updates or duplicates, *Push* at its worst case is $\Omega(\log n)$, where the newly pushed element has to traverse all $h+1$ levels. When duplicates are allowed the worst case increases. For a given heap with n distinct elements, if r updates are made and no pops, then the heap will contain r duplicates for a total of $n+r$ elements. If now a new element e with higher priority than all other elements is pushed onto the heap, it will be placed at the end of the array and will need to percolate up the height of the heap to become the root. Since there are now $n+r$ total elements, the height is then $\log(n+r)$, thus $\Omega(\log(n+r))$ swaps must be made for e. □

As previously mentioned, the way we handle a *Pop* must be modified to handle these duplicate entries by repeating the *Pop* operation until a non-duplicate element is returned. This causes *Pop* to have a worse worst case.

LEMMA 3.2 *Pop with duplicates has a worst case of* $\Omega(r\log(n+r))$.

Proof. Given a heap containing n elements, say no *Pop*s occur until r updates are performed, all with new priorities lower than that of all previous priorities. This causes all r duplicate elements with old priorities to rise above all non-duplicates. Thus, when wanting to pop, it must be repeated r times for each of the duplicates until a non-duplicate element becomes root. Since the heap would contain $n+r$ elements, each *Pop* would cause at most $\log(n+r)$ swaps. Because this is done r times, in the worst case, it has a running time of $\Omega(r\log(n+r))$. □

If this implementation of a heap was used in a real application where updating of priorities is needed, suddenly having a *Pop* take much longer than all previous *Pop*s could cause issues. This would be a big issue in something like a real-time system, in which predictable performance is needed for reliable thread scheduling.

4 IMPROVEMENTS THROUGH POSITIONAL TRACKING

Some of the things that a the basic heap lacks is a way to track the elements within the heap as they move around. Doing this can allow extra features such as lookup and modification of any element in the heap. This would be incredibly useful for a priority queue, modifying priorities on the fly without having to rebuild the heap or undertake the cost of storing duplicate elements as described in Section 3.3.

The easiest way to do this, and the one that will be looked at from here on, is using a dictionary. Referred to as D, it stores the indices of elements within the heap's array. There are a few ways to go about implementing operations using this, such as having a distinct operation for updating a priority of a node in the heap, but in this case we will modify *Push* to include automatic updating for keys that already exist. *Pop* also needs to be modified, along with the function underneath all the others, *Swap*. *Heapify Up* and *Heapify Down* do not need to change as we can put the position tracking into *Swap*.

The indices of elements will be stored and accessed in D by an elements key. Because of this, each element's key must be distinct from all other elements' keys. If the updating of a nodes was its own function, it would be possible to *Push* a duplicate item onto the heap, which would cause undefined behavior of the *update* operation as the newly pushed element would replace the original within D, meaning there is an non-tracked element within the heap.

We now define two properties of elements that will be used throughout the new implementations. For some element x, $x.key$ references the element's key. $x.priority$ will refer to the element's priority. These can be used to return the current setting for each property, or used to change the property. For example "$x.priority \leftarrow 5$" will change x's priority to 5, and "**return** $x.key$" will return x's current key from a function.

Given some element x, with the index i, "$D[x.key] \leftarrow i$" will store x's index from the heap into the dictionary D. This can then be retrieved by "$D[x.key]$" and used to access the element with "$A[D[x.key]]$".

4.1 Improved Heap Operations

Although *Heapify Up* and *Heapify Down* will now require D to be passed to them, their pseudo code will not be restated as they use D only to pass onto *swap2*.

Algorithm 9 Swap2

Require: $0 \leq j < n$, $0 \leq k < n$

```
1: function SWAP2(array A[0,...,n−1], dictionary D, integer j, integer k)
2:     D[A[j].key] ← k
3:     D[A[k].key] ← j
4:     t ← A[j]
5:     A[j] ← A[k]
6:     A[k] ← t
7:     return
```

Algorithm 10 Pop2

```
1: function POP2(array A[0,...,n−1], dictionary D, integer n)
2:     popped_element ← A[0]
3:     D[A[n−1].key] ← 0
4:     A[0] ← A[n−1]
5:     A ← A[0,...,n−2]          ▷ Reduce the size of the array by 1, removing the last element
6:     D.pop(popped_element.key)          ▷ Removes the root's entry from the dictionary D
7:     heapifydown(A, 0)
8:     return popped_element
```

Algorithm 11 Push2

```
1:  function PUSH2(array A[0,...,n−1], dictionary D, element x)
2:      if x.key in D then
3:          if x.priority > A[D[x.key]] then
4:              A[D[x.key]] ← x
5:              heapifydown(A, D[x.key])
6:          else if x.priority < A[D[x.key]] then
7:              A[D[x.key]] ← x
8:              heapifyup(A, D[x.key])
9:      else
10:         A.append(x)
11:         heapifyup(A, n)
12:     return
```

Algorithm 11 is changed from Algorithm 4 to handle pre-existing elements within the heap, and to then modify their priorities without pushing duplicate elements to the heap. If the key of a pushed element is already in the heap, it will compare the new and old priorities to determine if *Heapify Down* or *Heapify Up* needs to be called to restore the heap property. If the new and old priorities are equal, then nothing is done, as no update is required.

LEMMA 4.1 *Push2 (Algorithm 11) is still* $O(\log n)$ *after the changes from the original Push (Algorithm 4)*

Proof. The change of tracking positions within the heap was mainly confined to *swap2*, shown in Algorithm 9, where only a constant of 2 extra operations were added per call, which clearly does not change *swap2*'s complexity[10]. *Push2* has 3 cases:

- **Case 1:** The new element's key is not already in the heap.
  This is the same as a normal push with Algorithm 4, thus has same running time of $O(\log n)$.
- **Case 2:** The new element's priority is lower (larger value) than the one that's already in the heap of the same key.
  The priority of the old element is updated to the new one, then *Heapify Down* is called on its index. Clearly, this is just the running time of *Heapify Down*, which we know from Lemma 2.2, is $O(\log n)$.
- **Case 3 :** The new element's priority is higher (smaller value) than the one that's already in the heap of the same key.
  Similarly to Case 2, the priority is updated, then *Heapify Up* is called on the old element's index. Again, by Lemma 2.2, we know this is $O(\log n)$.

Given that all cases are shown to be $O(\log n)$, it has been proven that *Push2* is also of running time $O(\log n)$. □

Heapify stays almost the same, except it must first iterate through all elements to first add their positions to D. As the original *heapify*'s running time is $\Theta(n)$, this does not change its complexity.

[10] Assuming perfect hashing.

4.2 NEW OPERATIONS

With every element's index within the heap's array now being tracked, multiple new operations are now possible and can be added while still keeping the running time low. Two of these will be covered: *Get Priority*, which returns an element's priority from its key, and *Remove* which will allow removal of any element in the heap by its key.

Algorithm 12 Get Priority

```
1: function GETPRIORITY(array A[0,...,n−1], dictionary D, key k)
2:     priority ← A[D[k]].priority
3:     return priority
```

Algorithm 12 while very simple, is incredibly useful, which will later be shown in a modified Dijkstra. Normally, if you're wanting to know the current priority of an element within a heap, you'd have to keep that information separately. For example, in Algorithm 8 it uses the *dist* array to keep track of most recent distances each time they're pushed onto the heap. But now this is an easily added feature using an already existing part of the data structure, which means no extra code or outside storage is needed.

Algorithm 13 Remove

```
1: function REMOVE(array A[0,...,n−1], dictionary D, integer n, key k)
2:     removed ← A[D[k]]
3:     D[A[n−1].key] ← D[k]
4:     A[D[k]] ← A[n−1]
5:     A ← A[0,...,n−2]
6:     if removed > A[D[k]] then
7:         heapifyup(A, D, D[k])
8:     else
9:         heapifydown(A, D, D[k])
10:    D.pop(k)
11:    return removed
```

Algorithm 13 describes the *Remove* operation. All steps except the calls to *Heapify Up* or *Heapify Down* are constant time, and since only one of those two can be called, *Remove* has a running time of those algorithms which by Lemma 2.2 is $O(\log n)$. Although not used in Dijkstra, this is can still be useful in applications where keeping track of a certain element is no longer needed and is better to be removed than to stay and use up space in the heap.

Algorithm 13 is very similar to that of Algorithm 10 (*Pop*). When implemented, *Pop* can piggyback off of *Remove* by just calling *Remove* for index 0.

4.3 IMPROVED DIJKSTRA

Algorithm 14 Dijkstra2

```
1:  procedure DIJKSTRA2(graph G = (V, E), node s ∈ V, node v ∈ V, function cost: V × V → ℝ⁺ ∪ {∞})
2:      array pq
3:      dictionary D
4:      array color[0, ..., n − 1]
5:      array dist[0, ..., n − 1]
6:      for u ∈ V do
7:          color[u] ← WHITE
8:          dist[u] ← ∞
9:      dist[s] ← 0
10:     color[s] ← BLACK
11:     push2(pq, D, (0, s))                ▷ (0, s) ensures it is sorted first by cost, 0, then by index, s
12:     while pq is not empty do
13:         t1, u ← pop2(pq, D)             ▷ t1 is the cost, u is the node's index.
14:         for x ∈ {x : x is adjacent to u} do
15:             t2 ← t1 + cost(u, x)        ▷ t2 is the cost from s to x through u
16:             if color[x] = WHITE then
17:                 color[x] ← GREY
18:                 push2(pq, D, (t2, x))
19:             else if color[x] = GREY and getpriority(pq, D, x) > t2 then
20:                 push2(pq, D, (t2, x))
21:         color[u] ← BLACK
22:         dist[u] ← t1
23:     return dist[s]
```

Algorithm 14, although very similar to Algorithm 8, uses the improved heap methods to avoid having to manually keep track of most recent distances pushed onto pq. It also means that no checks need to be made within the main while loop to see if the popped node is a duplicate that has been fully explored. This version ensures that there are no redundant elements within the heap, and avoid all the issues described in Section 3.3.

5 REAL IMPLEMENTATIONS

With the original goal of this paper being to implement a heap with the described improvements in Python, it makes sense to first look at what is already available.

5.1 PYTHON STANDARD LIBRARY - HEAPQ

Initially developed and first implemented in August 2002 [3, 4, 5], to then be released in 2003 under Python version 2.3 [5], the *heapq* library is Python's first heap based priority queue implementation to be made available as a standard library.

The structure of the implementation is very focused on functions, and very similar to most pseudo code descriptions of a heap. Running on top of Python's *List* object, *heapq* requires the underlying list to be passed as one of the arguments to each function call, similar to how it has been described so far in this paper.

heapq implements the basic operations of a heap such as *Pop* referred to as *heappop*, *Push* as *heappush*, and *heapify*. Each of these modify the *List* passed to it in-place, meaning the passed *List* itself is modified without having to make a copy to modify then replace the old with a returned updated version. This would be unnecessary work that would slow down all of *heapq*'s functions. *heapq* functions only return values when needed, such as in *heappop*, in which it returns the element at the root of the heap. This in-place modification is the same as previously described in Section 2.2.2.

The operations *Heapify Up* and *Heapify Down* have equivalent functions, but their names are not the same as described in this paper. That must be kept in mind when looking at the source code of *heapq* to avoid confusion. The *Heapify Down* function described in this paper is named as *_siftup*, and *Heapify Up* is named *_siftdown*. Although the actual inner workings of the functions differ to the pseudo code described, their purpose is the same.

heapq can be used as a priority queue by storing elements as Python *Tuples*, with the first element in the *Tuple* being the priority, and the second being the key of the element.

heapq, while a fully functional heap, has the same shortcomings talked about in Section 3.3, with no element position tracking, or ways to efficiently update priorities of elements already within the heap. This means that Algorithm 8's version of Dijkstra must be used over the preferred Algorithm 14 version.

Even though *heapq* has these issues, for general use, it would be preferred over a pure Python implementation of a heap with position tracking. This is because *heapq* has a C implementation which is automatically imported over the Python version, and in most cases is much faster than any pure Python version could be, even when considering the duplicate elements brought in for updating priorities.

5.2 Custom Implementation - HeapQueue

Code: https://github.com/Ginotuch/cs380-heap-structures

Developed for the purposes of this paper, *HeapQueue* is the Python implementation of the improved heap described in Section 4 and is a more *Pythonic* approach to code than the standard library *heapq*. Implemented as a class, *HeapQueue* takes advantage of object oriented programming to increase ease of use, readability, testing, and debugging of the code itself. While in real world applications *HeapQueue* may be much slower, it can still be used to demonstrate the potential performance and functionality gains over a basic heap.

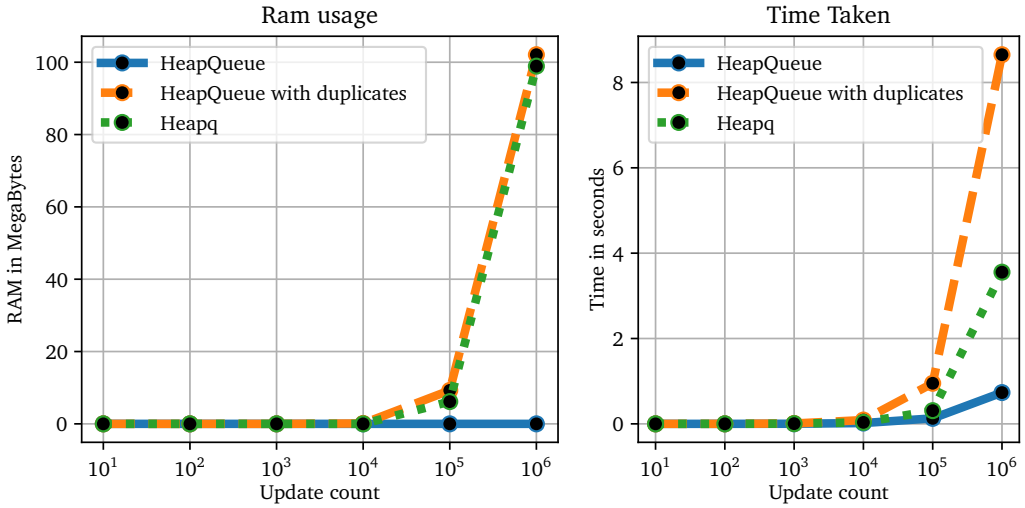
With *HeapQueue* being implemented as a class, the majority of the functions in Section 4 are included as methods which can be called using an instance of the *HeapQueue* class. This means no external *List* has to be created and stored or manually passed every time a heap operation needs to be performed. The downside being that with the more modular and object based approach, there is more overhead from the class itself. There is also more abstraction of some code, such as the *Swap* operation. Compared to *heapq*, this causes more function/method calls and can increase overhead inducing some performance loss.

HeapQueue also allows for position tracking to be disabled, which causes it to have the same functionality as that of a basic heap. Doing so means that comparison of performance can then be more accurate since the majority of the code is shared. Even still, we will next compare performances of *HeapQueue*, *HeapQueue* but with position tracking disabled (also referred to as *HeapQueue* with duplicates), and the standard Python library *heapq*.

5.3 Performance Comparison

5.3.1 Priority updating

Using the most basic example of repeatedly updating the priority of a single element in the heap, we can see a vast difference in performance and resource usage. Since without position tracking it is not possible to update the priority of any arbitrary element easily, we will use the previously described method of pushing a duplicate element onto the heap with a higher priority.
The test case made added a single element to the heap, then had its priority updated multiple times. During this, the extra RAM used and time taken was recorded and is shown below:



As expected, both *heapq* and *HeapQueue with duplicates (tracking disabled)* perform much worse given that the new *HeapQueue* only changes the priority of a single element, compared to the rest which introduce new elements on each update.

5.3.2 Strange Heapq Heapify Down Behavior

Strangely, the *_siftup* fuction in *heapq* is flawed, which causes it to do many more iterations than is needed. To show an extreme example, we create an array of n elements, all with the same key and priority, then run *Heapify* on this array using both *HeapQueue* and *heapq*. After the heap is constructed, we then pop the root element. What should happen is the root element is returned and the last element in the array replaces the root, to then be percolated down (*_siftup* in *heapq*). Since each element is of exactly the same priority and key, no swaps should happen as the heap property is already met.

However, the way *heapq* implemented *Heapify Down* (named as *_siftup*) will replace the root node with whatever node is at the end of the array, say x. It will then traverse down the heap from the new root, x, shifting each child up, replacing its parent (including the just placed x) until a leaf is reached[11]. At this point, x is now placed at this leaf to then have *_siftdown* (*Heapify Up*) called on this new position, although *_siftdown* will do nothing as all elements are equivalent. After all this, it will finally return the original root's key.

Shown in Fig. 4 are the number of *Heapify Down* loops that are performed when popping the root node with both *heapq* and *HeapQueue*. For each input size, *heapq* does $\log(n)$ loops, which is to be expected

[11] It travels down some path from the root to some leaf position.

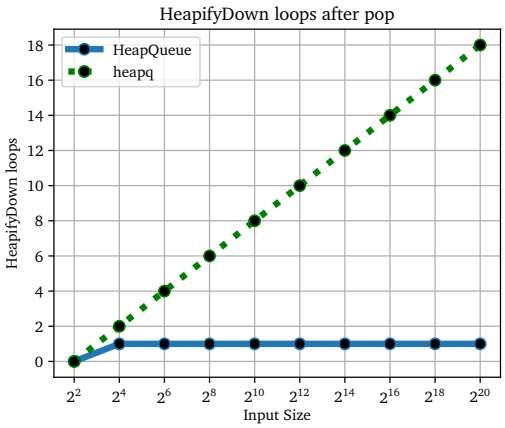


Figure 4: Showing *heapq*'s unnecessary loops

as it travels the height of the heap, from the root to a leaf. Whereas *HeapQueue* only does 1 loop in which it immediately exits the function.

This is easily solvable by replacing the *_siftup* with a Python implementation of Algorithm 3. Doing so provides the same functionality, and causes no conflicts with other functions within the *heapq* library. Developed alongside *HeapQueue* is *myheapq*, a version of *heapq* with a fixed *_siftup* function. Below are performance comparisons when running *Heapify* on different input sizes.

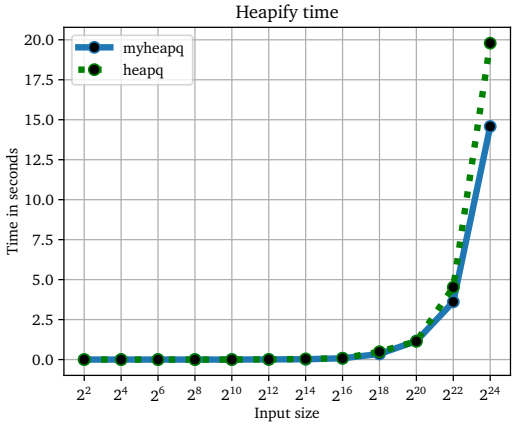


Figure 5: *Heapify* performance difference in *heapq* and *myheapq*

By looking at Fig. 5, we can see that although the results are very close, *myheapq* performs better over *heapq*. Both result in an equally correct heap, demonstrating that fixing the *_siftup* function could improve performance with no downsides. This same improvement would translate over to the C implementation of *heapq*, and most likely produce the same speedups as the Python versions.

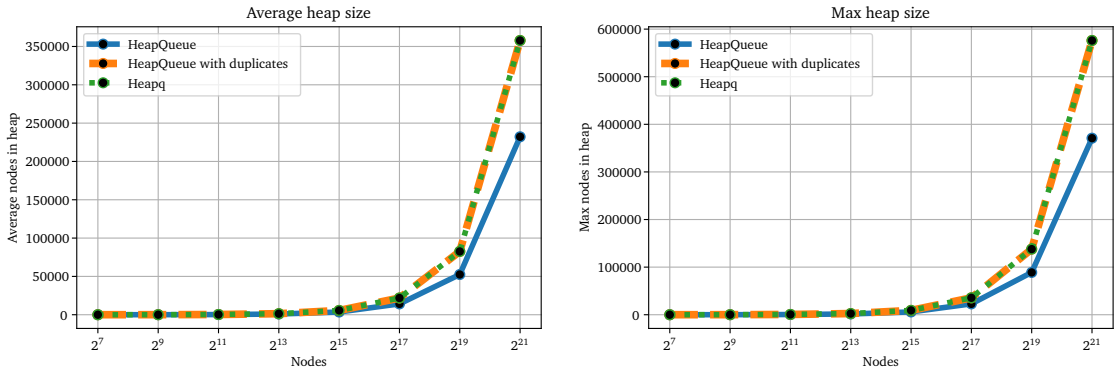
5.3.3 DIJKSTRA

For the purposes of both testing the correctness and efficiency of *HeapQueue*, Dijkstra has been implemented to run on top of three different heap implementation types: *HeapQueue*, *HeapQueue with no*

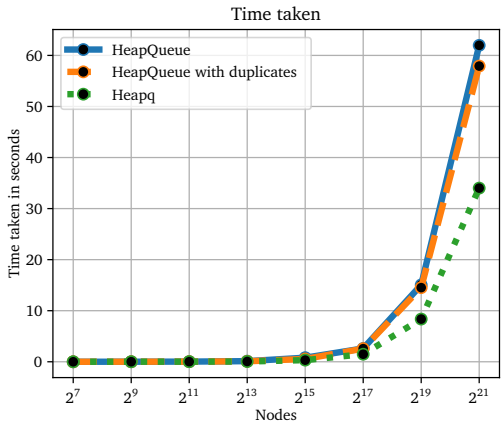
tracking, and *heapq*.

Running these versions on a randomly generated connected graph while tracking the average heap size, maximum heap size, and time taken produces the following results:

Note: *HeapQueue with duplicates and Heapq have identical results, thus their plotted lines lie on top of each other.*



Given these randomly generated graphs, the average and maximum size the heaps reach is smaller when using the new *HeapQueue* and tracking the elements. A clear difference is shown that when including duplicates, as expected, the heap takes up more resources.



Surprisingly, although *HeapQueue* achieves a lower heap size, it takes longer on average than all other heap types. Again, as expected, *heapq* performs better than *HeapQueue with duplicates*.

To understand why, we take a closer look at the call counts and time spent inside each method. The following results are from running the two versions on a graph with 10^6 nodes.

HeapQueue: total time of 39.11s

Name	Call Count	Time (ms)	Own Time (ms)
_heapify_down	992270	24898	13804
_swap	17647346	12760	12760
push	1451683	5030	2174
pop	992270	27442	407

HeapQueue no tracking (with duplicates): total time of 39.64s

Name	Call Count	Time (ms)	Own Time (ms)
_heapify_down	1436555	27915	19737
_swap	25273603	9125	9125
push	1451683	3741	1657
pop	1436555	30230	517

Unsurprisingly, *HeapQueue* has far less calls to *_heapify_down* compared to *HeapQueue with no tracking*. This is because of the massively reduced *pop* calls in *HeapQueue*, given that there are no duplicates that need to be repeatedly popped.

By looking at the call counts for *_swap*, we can see that although *HeapQueue* makes less calls, the method takes a total of 3 seconds longer compared to *HeapQueue with no tracking*. This is because of the overhead brought on from Python's *Dictionary* object that *HeapQueue*'s tracking relies on. Similarly, *push* also relies on the *Dictionary* object, adding and updating elements within it which are costly because of collisions from the hashing. Python will also dynamically increase the size of the *Dictionary* object each time too many elements are added.This takes a long time as large amounts of memory have to be moved whenever this happens.

5.3.4 Dijkstra - Specifically Crafted Test Case

Because of the fact that not tracking elements requires duplicates to be added, we can exploit this to create a test case where a large amount of updates are required to node distances.

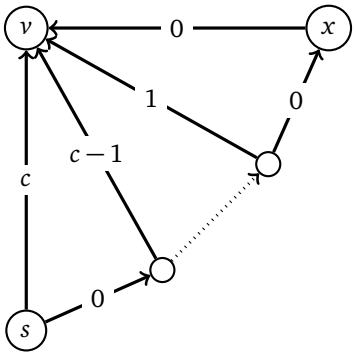
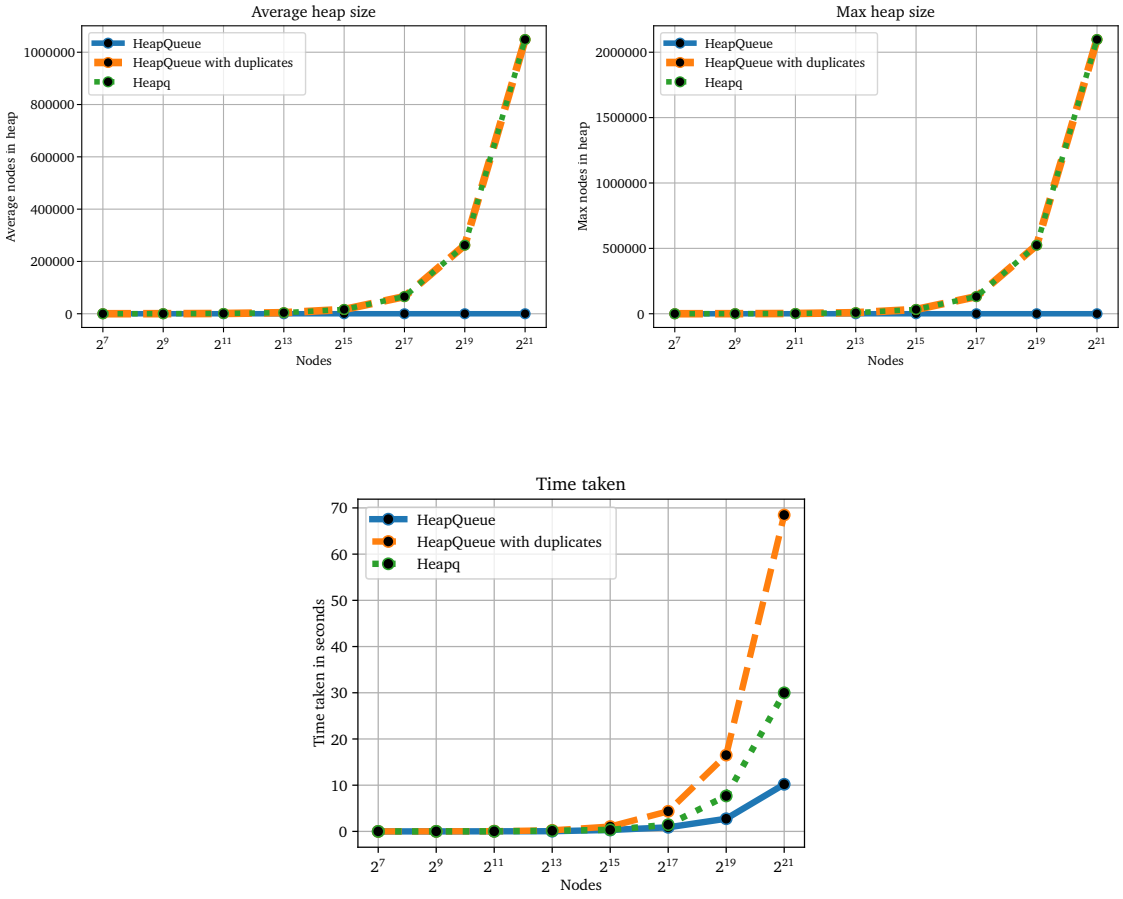


Figure 7: Graph that requires many updates of v when Dijkstra is run

First, we construct the graph shown in Fig. 7. The start node s is at the bottom, with the destination node v up the top. Then, on the path from s to x are nodes all connected by edges of weight 0. Each of these nodes are also connected to v by edges of progressively lower weights. When Dijkstra is run on this graph, it will start at s, adding all adjacent nodes onto its priority queue. It then follows the path of nodes up to x because of the low cost. At each node on this path to x, node v's best known distance is updated again because of the progressively lower cost of each edge.

This graph was constructed using a Python script, and run on the Dijkstra versions for *HeapQueue*, *HeapQueue with duplicates*, and *heapq*.



For this specific case *HeapQueue* far surpasses the other heaps with lower resources used, and faster running time overall. This makes sense as the test is fairly similar to that of the one in Section 5.3.1, with a single element needing to be updated many times.

Even with this case being very specific, it demonstrates how the tracking of elements can affect performance and resource usage in real applications relying on a heap.

6 FUTURE WORK

Implementing *HeapQueue* in C as a library for Python would greatly improve speed, and even make it comparable to that of *heapq*, allowing it to be a potential choice in a real development environment.

Many of the performance testing was done on a computer with 16GB of memory, redoing the tests on a computer with more RAM (64GB+) would allow much larger test cases to be tried. This would give greater insight into how the improvements perform under much more stressful conditions.

As the improvement entirely relies on a dictionary to keep track of element indices the next thing to research would be to look at using a perfect hashing function and limiting the input domain of keys, this could result in actual $O(1)$ complexity for some of the functions versus $O(1 + \textit{(dictionary access time)})$. The current implementation of *HeapQueue* relies on the Python *Dictionary* object, causing massive slowdowns from collisions and memory moves each time the dictionary needs to grow in size. The issue

of memory moves is also present in the Python *List* in which the heap's elements are stored. This would be solved by using a statically sized *List* set when *HeapQueue* is first initialized, but this brings the issue of what do to when the heap grows larger than the initially set size.

7 CONCLUSION

Having first covered the commonly described heap and the running time of its operations, we then moved on to discuss an improved version of a heap. The improvements were shown to allow some advanced functionality not possible with the basic heap, such as updating the priority of arbitrary elements within a heap. Scenarios were described that without positional tracking would have potentially large impacts on performance and efficiency. With the tracking of elements, it was shown that some cases, which causes massive issues in the original heap, can be solved and easily implemented.

Although there were many theoretical performance improvements, when implemented in an actual programming language, we saw that it performed worse on many cases when compared to already existing libraries. This worse performance was in part caused by the overhead of the more *Pythonic* implementation. Issues were also found to arise from the fact that Python's *List* and *Dictionary* objects were used. Hashing collisions and quickly expanding element counts caused operations on these objects to slow the rest of the code down.

The new implementation *HeapQueue* was shown in some specific cases to out perform the other versions of a heap, but these cases are rare and may not be commonly found in practice. Strange issues in *heapq* were displayed and a potential fix was shown, opening the possibility of the standard library to be updated.

With the completed development of *HeapQueue*, it is now an easy resource to look at and learn about adding positional tracking to heaps, and how they can be used in different applications. It is also creates a good starting point for further development and optimizations.

This paper can now be used as a resource for understanding positional tracking and its functions, and can also be used as an easy reference when discussing the improvements.

REFERENCES

[1] Michael J. Dinneen, Georgy Gimel'farb and Mark C. Wilson. *Introduction to Algorithms and Data Structures*. 4th. 2016.

[2] G. E. Forsythe. 'Algorithms'. In: *Communications of the ACM* 7.6 (1964-06), pp. 347–349. ISSN: 15577317. DOI: 10.1145/512274.512284.

[3] *Re: [Python-checkins]python/dist/src/Lib heapq.py,NONE,1.1*. 2002. URL: https://mail.python.org/pipermail/python-dev/2002-August/027283.html (visited on 2020-01-23).

[4] Guido van Rossum. *History for Lib/heapq.py - python/cpython*. URL: https://github.com/python/cpython/commits/2.7/Lib/heapq.py (visited on 2020-01-23).

[5] Guido van Rossum. *PEP 283 – Python 2.3 Release Schedule*. 2002. URL: https://www.python.org/dev/peps/pep-0283/ (visited on 2020-01-23).

[6] School of Computer Science and Department of Mathematics. *Discrete Structures in Mathematics and Computer Science*. Auckland: Univeristy of Auckland, 2019.