Assignment 3

Brandon Kamplain and Mia Weber

October 30, 2022

1 Question 1:

Please see attached files. (main.cpp, Horspool.cpp, KMP.cpp, Karp-Rabin.cpp)

2 Question 2:

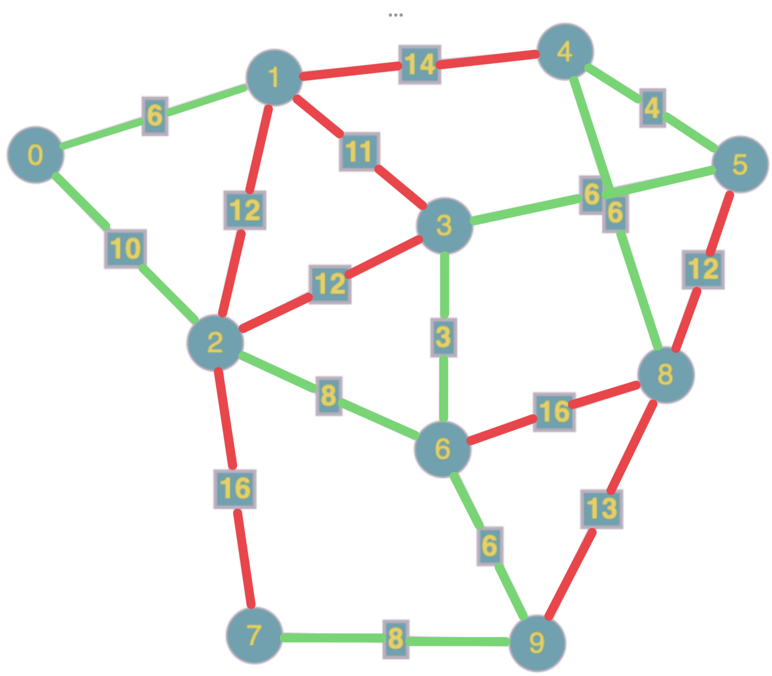
Adjacency Matrix:

	0	1	2	3	4	5	6	7	8	9
0	∞	6	10	∞	∞	∞	∞	∞	∞	∞
1	6	∞	12	11	14	∞	∞	∞	∞	∞
2	10	12	∞	12	∞	∞	8	16	∞	∞
3	∞	11	12	∞	∞	6	3	∞	∞	∞
4	∞	14	∞	∞	∞	4	∞	∞	6	∞
5	∞	∞	∞	6	4	∞	∞	∞	12	∞
6	∞	∞	8	3	∞	∞	∞	∞	16	6
7	∞	∞	16	∞	∞	∞	∞	∞	∞	8
8	∞	∞	∞	∞	6	12	16	∞	∞	13
9	∞	∞	∞	∞	∞	∞	6	8	13	∞

Using Prim.cpp we get the following edge list:

Edge	Weight	Total Cost
$0 \to 1$	6	6
$0 \to 2$	10	16
$6 \to 3$	3	19
$5 \to 4$	4	23
$3 \to 5$	6	29
$2 \to 6$	8	37
$9 \to 7$	8	45
$4 \to 8$	6	51
$6 \to 9$	6	57

Using the following edge list generated from Prim.cpp we get the following minimal spanning tree:



3 Question 3

Please go to the next page to see the Pseudocode for Question 3. When a node is deleted from a heap, the other nodes must be rearranged in order to continue to meet the requirements of a heap (the data value at the root of a heap is larger than all of its children). Therefore, an algorithm that finds and deletes the element of the smallest value in a heap also needs to rearrange the surrounding nodes in order for the remaining data structure to continue to be classified as a heap. The siftdown algorithm accomplishes just this. The siftdown algorithm moves a new data item in a child branch down to the correct location in the heap in order to re-establish the heap. The algorithm must first identify the smallest element in the heap, remove that element from the heap, and then use the siftdown algorithm to re-establish the conditions of the heap. The time efficiency of the following algorithm is O(n).

I declare that all material in this assessment task is my work except where there is clear acknowledgment or reference to the work of others. I further declare that I have complied and agreed to the CMU Academic Integrity Policy at the University website. http://www.coloradomesa.edu/student-services/documents

Algorithm 1 deleteNode

if ($heap\ is\ empty$) **then**
  return false
else
  $size \leftarrow sizeof(arr)/sizeof(arr[0])$
  $N \leftarrow size(heap)$
  $minElement \leftarrow heap[\frac{N}{2}]$
  for $i \leftarrow \frac{n}{2+1} until\ n$ **do**
    $minElement \leftarrow min(minElement, heap[i])$
  end for
  for $i \leftarrow 1\ until\ n$ **do**
    if ($heap[i] = minElement$) **then**
      $M \leftarrow i$
      $heap[i] \leftarrow heap[M]$
    end if
  end for
  for $i \leftarrow m\ unitl\ \frac{2}{n}$ **do**
    if ($heap[2 \times i] > heap[(2 \times i) + 1]\ and\ heap[2 \times i] > heap[i]$) **then**
      $swap(heap[i], heap[2 \times i])$
      $i \leftarrow (2 \times i) + 1$
    else if ($heap[2 \times i] < heap[(2 \times i) + 1]\ and\ heap[(2 \times i) + 1] > heap[i]$)
then
      $swap(heap[i], heap[(2 \times i) + 1])$
      $i \leftarrow (2 \times i) + 1$
    else
      break
    end if
  end for
  $n \leftarrow n - 1$
end if

Algorithm 2 correctHeap

$index \leftarrow 1$
while ($index < size$) **do**
  $siftdown(heap, index)$
  $index + +$
end while

Algorithm 3 siftdown

$leftchildindex \leftarrow root \times 2 + 1$
$rightchildindex \leftarrow root \times 2 + 2$
if $(leftchildindex <= last)$ **then**
    $leftkey \leftarrow heap[leftchildindex].key$
    if $(rightchildindex <= last)$ **then**
        $rightkey \leftarrow heap[rightchildindex].key$
    else
        $rightkey \leftarrow leftkey - 1$
    end if
    if $(leftkey > rightkey)$ **then**
        $largerchildkey \leftarrow leftkey$
        $largerchildindex \leftarrow leftchildindex$
    else
        $largerchildkey \leftarrow rightkey$
        $largerchildindex \leftarrow rightchildindex$
    end if
    if $(heap[root].key < largerchildkey)$ **then**
        $swap(heap, root, largerchildindex)$
        $siftdown(heap, largerchildindex, last)$
    end if
end if

Author's Name: Mia Weber UID: 700510845 Date: 10/30/2022
Author's Name: Brandon Kamplain UID: 700510289 Date: 10/30/2022