Advanced Algorithms – Assignment 2

Lucas Geurtjens (s5132841)

# Q1. Tracking K-Smallest Numbers

**Introduction:**

A data structure was created to store a very large number of random numbers and a separate list of k-smallest numbers in O (N + K) space complexity. This k-smallest list was maintained using a Fibonacci heap.

**How it works:**

The algorithm uses an array for storing the N random numbers, and a Fibonacci Heap for storing the k-smallest numbers. When inserting the random numbers, it will also insert up to K of them into the Fibonacci Heap. Once the Fibonacci Heap is full (it’s number of nodes are equal to K) it stops inserting nodes, unless the node is smaller than the max node of the Fibonacci Heap (it is a max heap, not min heap). When this happens, the max node is removed, and the new node is added.

*Deletion -* Since we are using a Fibonacci Heap, removing the max value is an easy O (1) operation, since we maintain a pointer that points to the max value in the heap. When this is removed, the children of this node are promoted to the root list. There is also the option of removing a non-root element, by using a find () method. This will find the element by value using a BFS. Since it is a heap, we can trim down the search space by not checking the children of nodes that are smaller than the node we are looking for.

**Input/Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Numbers** | **K-Smallest Numbers** | **Seed** |
| N=10, K=3 | 38 19 37 55 97 65 85 50 12 53 | 37 19 12 | 0 |
| N=20, K=5 | 45 16 198 195 84 50 190 31 105 116 157 189 52 6 153 29 13 167 58 161 | 31 6 29 16 13 | 2 |
| N=100, K=15 | 48 196 294 91 31 577 702 503 217 168 409 233 23 152 578 399 863 25 489 718 454 798 164 182 498 731 271 899 936 897 449 232 865 208 561 263 779 893 193 948 45 908 754 459 594 592 0 447 65 240 616 125 892 724 335 279 513 873 389 341 486 600 714 321 96 974 215 882 829 400 376 220 536 549 824 166 757 784 438 404 541 566 847 853 20 149 291 792 978 764 763 71 820 181 924 339 885 648 963 700 | 164 0 91 45 152 65 125 48 96 23 31 25 149 20 71 | 3 |
| N=25000, K=100 | .. Too large to store… | 124 89 126 46 77 32 101 96 47 3 81 38 76 68 54 48 7 1 122 69 118 16 117 51 78 26 112 90 103 98 83 49 20 0 123 10 70 61 116 111 109 66 95 27 74 41 93 50  85 52 113 57 71 43 97 72 94 2 110 14 40 17 80 34 42 6 45 121 115 65 23 92 82 31 11 73  62 29 4 63 25 55 28 114 36 106 9 104 19 37 33 102 84 44 13 24 15 12 5 56 | 3 |

**Amortised Analysis:**

The amortised Fibonacci Heap cost was calculated using the potential function:

*Φ = nodesInRootList*

NOTE: I got help with my Fibonacci Heap amortised analysis by following Damon Wischik’s video on the topic: (<https://www.youtube.com/watch?v=RCCUrmklzjg>).

Insert

Just adds a node to the root list which has an actual cost (C) of O (1).

* C = O (1)
* Φafter = 1
* Φbefore = 0
* ΔΦ = Φafter -Φbefore = 1 + 0 = 1
* C’ = C + ΔΦ = 1 + 1 = 2 = O (2) = O (1)

Thus, *Insert* has an amortised cost of **O (1)**

Delete Max Node

Removes the max node and promotes its children to the root list. Since this is updating the children of the node, it will have a cost (C) of O (noOfChildNodes).

* + C = O (noOfChildNodes)
  + Φafter = *nodesInRootList* – 1 + noOfChildNodes *(NOTE: We’ve got a -1 because we’re removing the max node from the root also.)*
  + Φbefore = *nodesInRootList*
  + ΔΦ = Φafter -Φbefore = -1 + noOfChildNodes
  + C’ = C + ΔΦ = O(noOfChildNodes) + noOfChildNodes – 1 = O (log N) *(NOTE: Because the degree of a node in a Fibonacci Heap will always be logN, we know this will be the number of nodes to move to the root list, thus we get O(logN).)*

Thus, *Delete Max Node* has an amortised cost of **O (log N)**

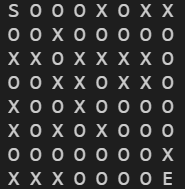
# Q2. Maze Generation

**Introduction:**

A maze generation algorithm was created using the technique of randomly knocking down walls until a path from the start and end is found.

**How it works:**

Here’s an example of an 8x8 maze:



LEGEND:

**S**: Start of maze

**E**: End of maze

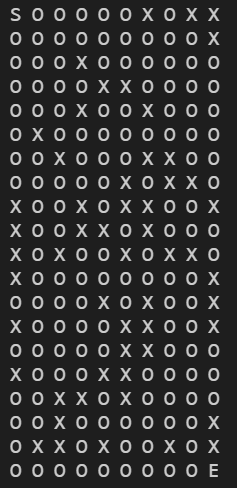
**X**: Wall (cannot pass)

**O**: Path (can pass)

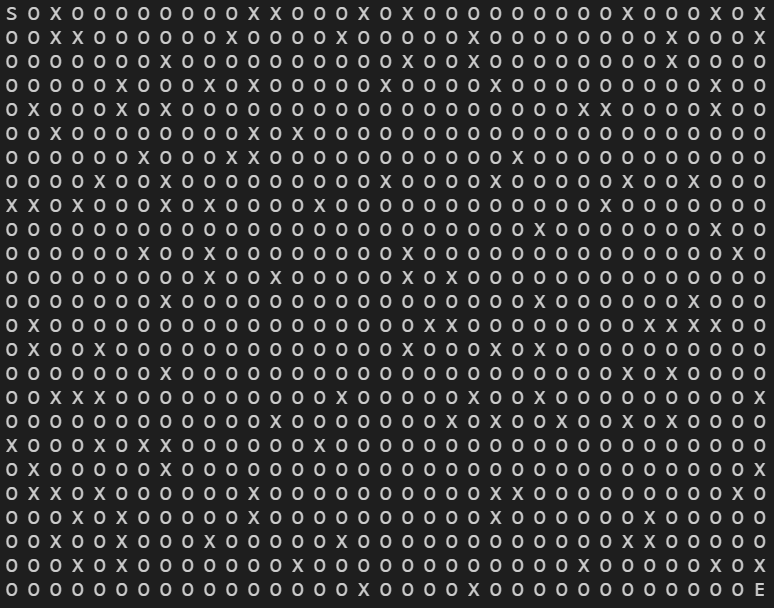
* The maze grid is initialised (filled with walls)
* While there is no complete path from the Start to the End of the maze:
  + Check if the maze is complete (if so, we can end)
    - Run a Breadth First Search from the Start -> End
    - Run a Breadth First Search from the End -> Start
      * During the BFS, calculate the furthest we can get to the goal state (done using Manhattan distance). When the BFS finishes (assuming it didn’t find the goal state), this value will help determine the minimum number of walls we must break down.
    - Calculate the number of walls to break down (the Manhattan distance from the furthest point from the start, to the furthest point from the end).
    - Breakdown the number of walls specified by randomly selecting points. If the point is a wall, break it down, and add this to the total amount of walls broken.

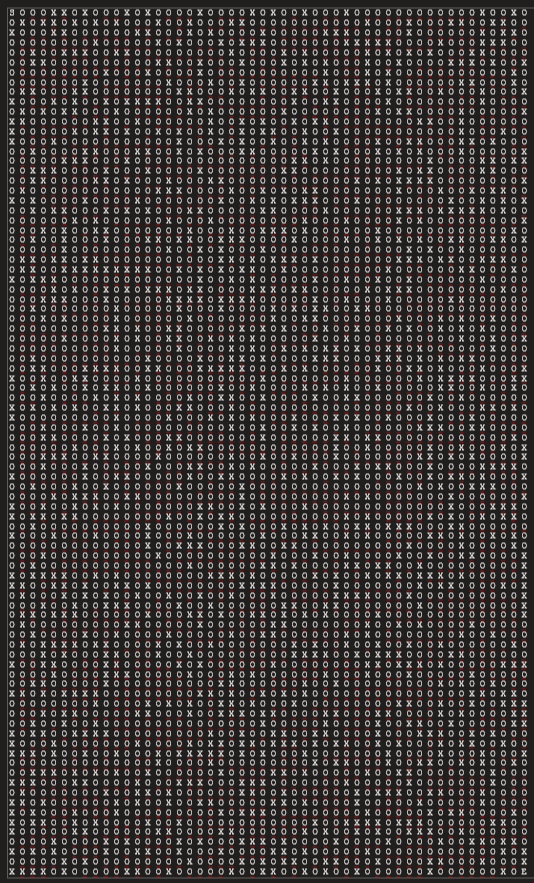
**Input/Output:**

10x20 Maze

****

35x25 Maze

****

50 x 88 Maze: 

# Q3. Red Black Trees vs Van Emde Boas Trees

**Introduction:**

A Red Black Tree and Van Emde Boas Tree data structure was attempted. The goal was to compare the two data structures and see which was most efficient for performing operations such as add, find, delete, and sequential access.

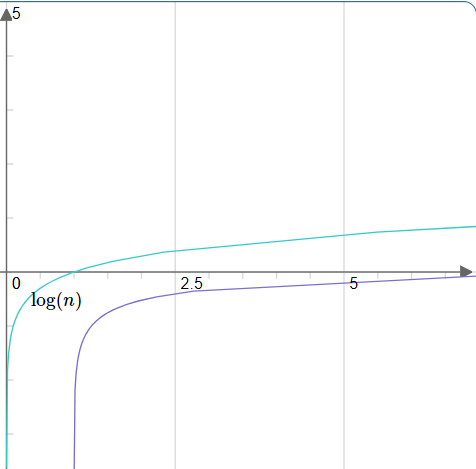
**Implementation:**

Both data structures were attempted, however, only the Red Black Tree’s insert had any sort of success (had some bugs that I was unable to solve). Thus, I was not able to compare the performance of the algorithms through my code. Instead, I will look at the known time complexities for the algorithms and make observations from that.

**Performance Analysis:**

We know that the Van Emde Boas Tree structure has an average time complexity of O (Log Log U) where U is the size of the universe. We also know that Red-Black Trees have a time complexity of O (Log N) where N is the number of elements in the tree.

*Red-Black Tree Complexity vs Ban Emde Boas Tree Complexity*



log(log(n))

(NOTE: Source to graphing tool used: https://www.solumaths.com/en/math-graph-app/graphing-calculator-online)

Graphing these two complexities we see that the difference between the trends is moderately small. Therefore, we could expect that given a large amount of data, the two algorithms would perform at a similar speed with each other, the Van Emde Boas tree being slightly faster.

# Q4. The Kevin Bacon Game

**Introduction:**

The Kevin Bacon Game tries to find the minimum number of links between two actors. The program has three main methods:

1. Find Minimum Links: Finds the minimum number of links between two actors.
2. Find Bacon Number: Finds the minimum number of links between an actor and Kevin Bacon.
3. Find Highest Bacon: Finds the actor with the greatest number of links to Kevin Bacon.

**How it works:**

*Finding links between actors (all methods use this as their key function).*

* Data is input and compiled into more useful data structures (a list containing actors, and a list containing movie productions).
  + Specified actors are input.
  + A Bacon Score is kept to track how many levels down the BFS goes.
  + A Breadth First Search is performed, adding the starting actor to a queue.
    - The queue is popped and we check the popped actor’s movies.
      * If the movie hasn’t been visited, we check the actors in the specific movie.
        + If the actor is the end actor, we return the bacon score.
        + Otherwise, if the actor hasn’t been visited, we add it to the queue.

**Input/Output:**

NOTE:I only used data from bacon1.txt and some of bacon2.txt. Adding too much data caused a bug I wasn’t able to solve. Thus, since we didn’t have access to ALL movies/actors, some number of the links may be slightly inaccurate.

|  |  |
| --- | --- |
| **Method** | **Output** |
| findBaconNumber ("Steven Brill (I)") | 3 links |
| findBaconNumber ("Carrie Fisher") | 2 links |
| findBaconNumber ("Jorn Benzon ") | 4 links |
| findBaconNumber ("Billy Crystal") | 2 links |
| findMinLinks ("Denise Dabrowski", "Roy C. Johnson") | 1 link |
| findMinLinks ("Roman Bohnen", "Alan Rickman") | 4 links |
| findMinLinks ("Albert Brooks (I)", "Steve Buscemi") | 6 links |
| findMinLinks ("Gino Corrado", "Tim Condren") | 3 links |
| findHighestBacon () | 6 links (Toshiyuki Amagasa) |

# Q5. Minimum Vertex Covers for Complement Graphs

**Introduction:**

An algorithm was developed to find the minimum vertex covers of a complement graph. The minimum vertex cover problem aims to select a set of nodes that has edges to every node in the graph. The graphs that we’re looking at are complement graphs. These are graphs whereby all edges disconnect from the nodes they are adjacent to, and connect to all nodes they were previously not adjacent to.

**Literature Review:**

The following algorithms have been developed for finding the minimum vertex cover of a graph:

* One solution was developed using Dijkstra’s algorithm, getting a vertex cover with a time complexity of O(n3) where n is the number of nodes in the graph [1].
* Another current solution is to use an approximate approach instead of an exact approach. In this algorithm, we continue picking the nodes covering the greatest number of edges until all nodes have been covered.

**Algorithm Description:**

The algorithm tries to perform a greedy search that selects vertices that result in the largest number of uncovered nodes being covered. It does this by maintaining a priority queue whereby nodes with the largest number of connected uncovered nodes are at the top. The connected nodes to this node are examined and also added to the priority queue. When all nodes are covered, the algorithm stops searching and returns the minimum vertex cover amount.

**Algorithm Overview:**

*File Input / Data Structure*

* Data is inputted from a specified clq file.
* The data is converted into a list of edges (data structure), that notes the to and from values of an edge.
* From the list of edges, the data is converted into a list of nodes. Each node has the following properties:
  + ID: The Node’s ID
  + Connections: A list of nodes that are connected to the node by an edge.
  + Unseen Connections: The number of nodes connected to the node that have not yet been seen.
  + Visited: If a node has been visited (i.e., selected as a cover vertex).
  + Seen: If a node has been seen (i.e., we know is connected to a cover vertex).

*Finding the minimum covering vertices*

* A priority queue is kept, holding a list of node pointers (nodes with highest number of unseen connections are at the head of the priority queue).
* The node in the graph with the greatest number of connections is selected to be the starting node in the priority queue.
* While the priority queue isn’t empty and not all nodes are covered…
  + We pop the head of the priority queue.
  + If this node hasn’t been visited, we set it as visited (now part of our minimum vertex cover set, and look at the nodes (aka child nodes) it’s connected to…
    - If the child node is uncovered, set it to be covered, and increase the number of covered nodes
    - Add the child to the priority queue, inserting it to the head if it has more uncovered nodes connected to it than the current head of the queue.

**Experimental Results and Comparisons:**

NOTE:Unfortunately, the algorithm was really slow and wouldn’t finish executing in a reasonable amount of time. As a result, I was unable to get any results to analyse.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Graph | Minimum Vertex Cover | | Average CPU Time | |
| Known results | Obtained results | Known results | Obtained results |
| Brock800\_1 | 777 | N/A | 84.54 | 1hr+ |
| Brock800\_2 | 776 | N/A | 74.90 | 1hr+ |
| Brock800\_3 | 775 | N/A | 45.30 | 1hr+ |
| Brock800\_4 | 774 | N/A | 26.75 | 1hr+ |
| C2000.9 | 1922 | N/A | 36.28 | 1hr+ |
| C4000.5 | 3982 | N/A | 142.02 | 1hr+ |
| MANN\_a45 | 691 | N/A | 88.76 | 1hr+ |
| p\_hat1500-1 | 1488 | N/A | 1.39 | 1hr+ |

**Conclusion:**

Overall, the quality of the algorithm was not great, given by the fact that it was not able to run in a reasonable amount of time. Given more time, I would remake the algorithm and improve its underlying data structure to reduce the number of nested loops used.

**References:**

[1] An Approximation Algorithm for the Minimum Vertex Cover Problem, Jingrong Chen, 2016, https://www.sciencedirect.com/science/article/pii/S1877705816002617