HW 4 - CS 120

Evan Jolley

October 2022

1

1.1

All code.

1.2

See Appendix for screenshots of generated graphs.

n	k_n^*
1024	13
2048	14
4096	15
8192	16
16384	17
32768	18

For each value of n, the chosen value k_n^* was the first generated graph where $MergeSortSelect$ appeared to out perform $QuickSelect$ in terms of runtime. Thus, we choose to flip to MSS at these k_n^* values in order to maximize efficiency.

In other words, for $k < k_n^*$, QS is more efficient, and for $k \geq k_n^*$, MSS is more efficient.

1.3

We are attempting to find a function that closely follows:

$$f(1024) = 13$$

$$f(2048) = 14$$

...

$$f(32768) = 18$$

Taking log_2 of each value of n is a good way to get the larger numbers in the same neighborhood as our discovered k values. We can add the same constant to each $log_2(n)$ value to get our k_n^* values.

$$f(n) = log_2(n) + 3$$

$$log_2(1024) + 3 = 13$$

$$log_2(2048) + 3 = 14$$

$$log_2(4096) + 3) = 15$$

$$log_2(8192) + 3 = 16$$

$$log_2(16384) + 3 = 17$$

$$log_2(32768) + 3 = 18$$

Thus, my functional form for k_n^* is:

$$k^*(n) = log_2(n) + 3$$

2

The problem of *DuplicateSearch* is rather easy using dictionaries/hash tables.

First, we initialize a dictionary of size $m = n$ where n is the length of the input. Doing this will run in time $O(m) = O(n)$.

Next we will iterate through our input as follows:

1. *Search* our dictionary using the key of each key value pair. This lookup will run in time $O(1)$ as defined in Lecture 9.
2. If the key is not in the dictionary, *Insert* the key value pair into our data structure. This insertion will run in time $O(1)$ as defined in Lecture 9.
3. If the key is in the dictionary, return that key as it is our duplicate.
4. If no key is returned by the end of the input, there is no duplicate.

In the worst case, this loop will call $Search(K)$ n times and $Insert(K, V)$ n times. Each of these calls runs in constant time, and the loop's total runtime is $O(2n) = O(n)$.

The total runtime for the initialization of our data structure as well as our loop is $O(n + n) = O(n)$ which is what we expected.

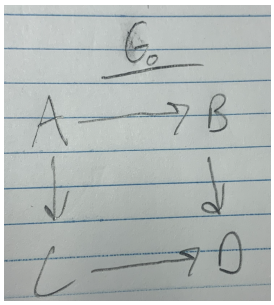
We know this algorithm is correct because:

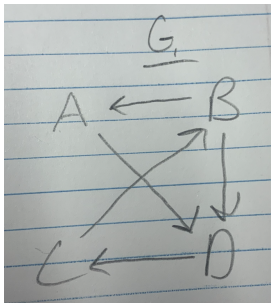
- *Insert* and *Search* are known to be correct.
- If there is a duplicate key, we will catch the collision as two pairs will hash to the same place.
- If there is not a duplicate key, all n values will hash to the n distinct locations in the structure.

3

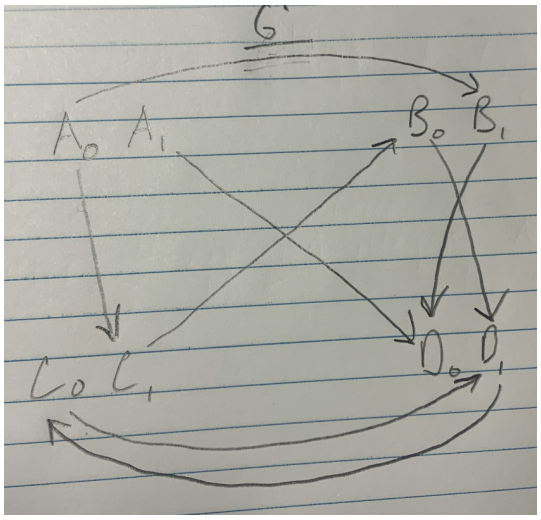
3.1

Consider the two graphs G_0 and G_1 where A is our starting point and D is our desired destination.





Beginning at A in G_0, we can step to either B or C. Given that in our second step we will be using the edges in G_1, we can imagine that we are stepping from A in G_0 to B or C in G_1. In fact, we can expand this thought and say that we step across graphs each turn, as edges constantly rotate back and forth between those present in G_0 and G_1. Let's visualize this in a new graph G':



Note that locations in G_0 (denoted with $_0$) can only step to locations in G_1 (denoted with $_1$) and vice versa.

Another important piece to note is that each vertex X_0 is the same vertex as X_1 in terms of location, as the vertices never change. The different grouping only refers to the routes available for the next turn. Thus, walks that end in D_0 and D_1 are both eligible to be the shortest walk. All walks begin at A_0 as G_0 always starts.

A Shortest Rotating Walk problem using graphs G_0 and G_1 could be solved by finding the Single-Source Walk of graph G'. Thus, let's expand this idea and construct this reduction more generally.

Reduction

1. **Preprocess:**
   - Construct a new graph of $k * n$ vertices where k is the number of graphs in the Rotating Walk and n is the number of vertices in each.
   - The initialization of our edges will follow our example above, following the same path within each step but pointing towards vertices in the next step's graph. Every edge of every graph will be represented, meaning the total amount of edges is the sum of the amount of edges in each graph: $m_0 + m_1 + ... + m_{k-1}$.
   - The resulting graph will have kn vertices and $m_0 + m_1 + ... + m_{k-1}$ edges. Each of these things can be created in constant time, so the total preprocess runtime is $O(kn + m_0 + m_1 + ... + m_{k-1})$.
2. **Oracle:**

- Single-Source Shortest Walk is called on our new graph once, generating the length of the shortest walk to each of the vertices in our graph.
- We know that this runs in time $O(\#\ of\ vertices\ +\ \#\ of\ edges)$ from Lecture 10, and in this example that equals $O(kn+m_0+m_1+...+m_{k-1})$.

3. Postprocess:

- Single-Source shortest walk returns the shortest length to get to each point. There are k possible endpoints in this graph. We must iterate over all kn points, checking if it is an endpoint and then checking to see if it is the shortest length of all endpoints. This will run in time $O(kn)$ as we will have kn vertices to check and constant time operations for each.

Summing the runtime of each of these steps calculates the total runtime of the reduction:

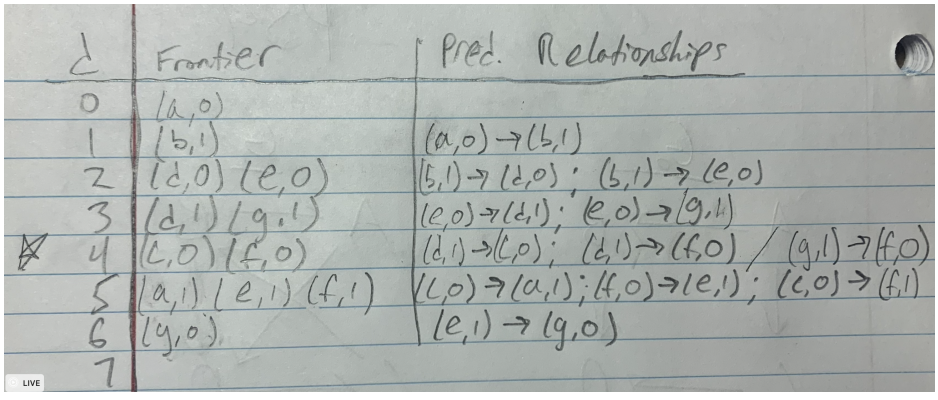
$$O(kn + m_0 + m_1 + ... + m_{k-1}) + O(kn + m_0 + m_1 + ... + m_{k-1}) + O(kn) =$$

$$O(kn + m_0 + m_1 + ... + m_{k-1})$$

We know that this algorithm is correct because:

- Our newly created graph accounts for all possible paths from our starting vertex to one of k possible ending vertices.
- Single-Source Shortest Walk will return the correct length of the shortest walk to each of these end points.
- Finding the minimum of these lengths will always be the overall shortest walk.

3.2



Note that there are two points that are impossible to access. It is impossible to get to $(c, 1)$ without first getting to $(b, 0)$. It is impossible to get to $(b, 0)$

at all as each arrow points away from b in G_1. Thus, both vertices will never be accessed. This table has shown that the shortest rotating walk from a to c is of length 4.

3.3

We can think of this as a rotating walk problem between three graphs, one for each player. In the graphs, there is one vertices for each space on the nxn grid, meaning there will be n^2 total vertices.

This is technically incorrect, as there are actually less than n^2 accessible spaces given the bombs randomly scattered throughout the map. However, when calculating worst case runtime, we can use n^2 as it is $\geq$ the real figure.

So to begin this problem, we'll need to initalize a graph with n^2 vertices in $O(n^2)$ time. As for the edges, we can discuss them on a case by case basis:

Rook

At any given square on the board, there are $2(n-1)$ different squares that the rook can move to (up, down, left, and right to the ends of the board). Thus, the amount of total edges in the rook's graph is $n^2 * (2n-2) = 2n^3 - 2n^2$.

Bishop

At any given square on the board, there are **at most** $2(n-1)$ different squares that the bishop can move to (all diagonals to the ends of the board). Of course, if the bishop is in a corner, there are only $n-1$ squares it can move to, but we are accounting for the worst case. Thus, the amount of total edges in the bishop's graph is $\leq n^2 * (2n-2) = 2n^3 - 2n^2$.

Knight

At any given square on the board, there are **at most** 8 different squares that the knight can move to. The principle of accounting for the worst case that we discussed with the bishop applies here as well. Thus, the amount of total edges in the knight's graph is $\leq 8n^2$.

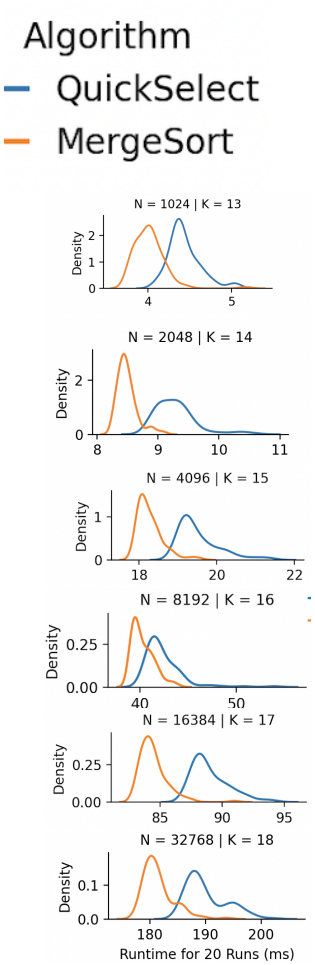
Conclusion

As we proved in problem $3a$, the runtime of a Rotating Walk is $O(\#\ of\ vertices + total\ \#\ of\ edges)$. In this example:

$$O(n^2 + 2(2n^3 - 2n^2) + 8n^2) = O(n^3)$$

The preprocess runtime of turning the game grid into a graphical representation is dominated by the Rotating Walk runtime.

4 Appendix


Algorithm
QuickSelect
MergeSort
N = 1024 | K = 13
N = 2048 | K = 14
N = 4096 | K = 15
N = 8192 | K = 16
N = 16384 | K = 17
N = 32768 | K = 18
Density
Runtime for 20 Runs (ms)