CS120: Intro. to Algorithms and their Limitations Hesterberg & Anshu

Problem Set 4

Harvard SEAS - Fall 2023 *Due: Wed Oct. 18, 2023 (11:59pm)*

Your name: Samuel Osa-Agbontaen
Collaborators: Office Hours
No. of late days used on previous psets: 1
No. of late days used after including this pset: 3

1. (Randomized Algorithms in Practice)

   (a) Implement Randomized QuickSelect, filling in the template we have given you in the Github repository. Implemented and submitted on Gradescope.

   (b) In the repository, we have given you datasets x_n of key-value pairs of varying sizes to experiment with: data set x_n is of size n. For each dataset x_n and any given number k, you will compare two ways of answering the k selection queries $\textsc{Select}(x_n, [n/k])$, $\textsc{Select}(x_n, [2n/k]), \ldots, \textsc{Select}(x_n, [(k-1)n/k])$ on x_n, where $[\cdot]$ denotes rounding to the nearest integer:

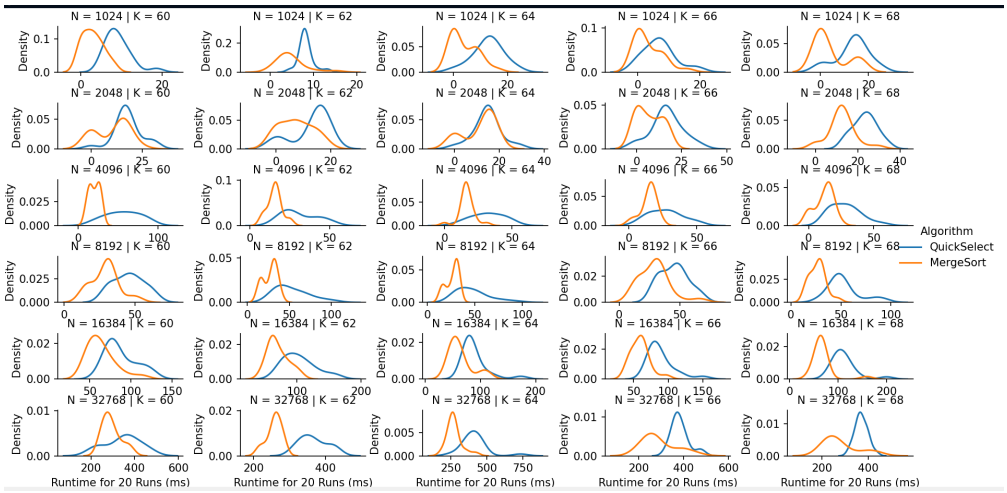
      i. Running Randomized QuickSelect k times

      ii. Running MergeSort (provided in the repository) once and using the sorted array to answer the k queries

      Specifically, you will compare the *distribution* of runtimes of the two approaches for a given pair (n, k) by running each approach many times and creating density plots of the runtimes. The runtimes will vary because Randomized QuickSelect is randomized, and because of variance in the execution environment (e.g. what other processes are running on your computer during each execution).

      We have provided you with the code for plotting. Before plotting, you will need to implement MergeSortSelect, which extends MergeSort to answer k queries. Your goal is to use these experiments and the resulting density plots to propose a value for k, denoted k_n^*, at which you should switch over from Randomized QuickSelect to MergeSort for each given value of n. Do this by experimenting with the parameters for k (code is included to generate the appropriate queries once the ks are provided) and generate a plot for each experiment. Explain the rationale behind your choices, and submit a few density plots for each value of n to support your reasoning. (There is not one right answer, and it may depend on your particular implementation of QuickSelect.)

      MergeSortSelect implemented and submitted on gradescope. My choices for $k*_n$ were a result of trial and error. My goal was to find some $k*_n$ for which the orange peak was further to the left of the blue peak, as this means that for that $k*_n$, MergeSortSelect was executed quicker than QuickSort. The plots can be seen below :

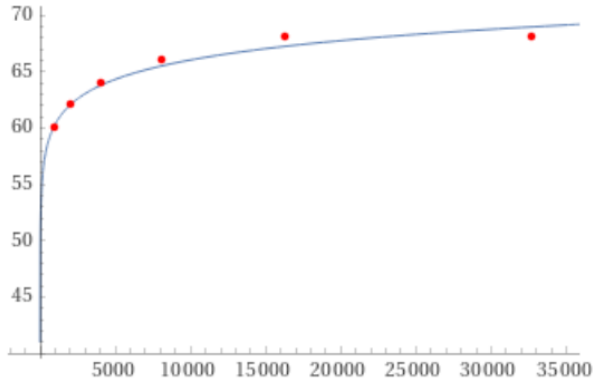


(c) Extrapolate to come up with a simple functional form for k_n^*, e.g. something like $k^*(n) = 3\sqrt{n} + 6$ or $k^*(n) = 10 \log n$. (Again there is not one right answer.) Putting the results of these trials into a least-squares fit regression line in Wolfram Alpha, the following graph and equation were the result :

Least-squares best fit

$$2.47319 \log(3.91422 \times 10^7 \, x)$$

Plot of the least-squares fit



(d) (*optional) One way to improve Randomized QuickSelect is to choose a pivot more carefully than by picking a uniformly random element from the array. A possible approach is to use the ***median-of-3*** method: choose the pivot as the median of a set of 3 elements randomly selected from the array. Add Median-of-3 QuickSelect to the experimental comparisons you performed above and interpret the results.

2. (Dictionaries and Hash Tables) Recall the DuplicateSearch problem from Lecture 3. Show that DuplicateSearch can be solved by a Las Vegas algorithm with expected runtime $O(n)$ using a dictionary data structure. (You can quote the runtimes of the implementation of a dictionary data structure from Lecture 9 without proof.) (No formal probabilistic analysis of

the runtime is necessary.)

On a high level, our algorithm will iterate across each element of our array, $(K_0, V_0), \ldots, (K_{n-1}, V_{n-1})$. For each element (K_i, V_i), we will check if K_i is already in our hash table (We can use a Las Vegas algorithm for our hash function for our dictionary to ensure that our hash function always outputs correct values). If K is not already included in our hash table, this means we have not yet visited an element with the same value, so we will add this element to our hash table. Otherwise, if K is already included in our hash table, this means that it's already been visited. Formally, this means that our hash table already contains K_i, meaning that we have just encountered the second occurrence of K_i in our input array, meaning for some $i \neq j$, $(K_i) = (K_j) = K$, so we will return K. If we iterate through the entire array and never encounter an element that was already visited (added to our hash table), this means there is no duplicate, therefore we return.

We know this algorithm is correct because we only return an element if it was already included in our hash table, which contains all the elements that already appeared to the left of the current element in the array, so encountering an array element that already is in the hash table means that we are encountering an element that has already showed up earlier in our array.

The runtime of iterating across our input array is $O(n)$, where n is the number of elements in our array. At each step, we insert into our hash table (which starts empty), and each insertion can be done in $O(1)$ time. At each step, we're also searching to see if K_i is in the hash table - this search is done in $O(1)$ time as well. So the fact that we are performing these constant-time searches and insertions across n elements makes our entire algorithm $O(n)$.

3. (Rotating Walks) Suppose we are given k digraphs on the same vertex set, $G_0 = (V, E_0), G_1 = (V, E_1), \ldots, G_{k-1} = (V, E_{k-1})$. For vertices $s, t \in V$, a *rotating walk* with respect to $G_0, \ldots, G_{k-1}$ from s to t is a sequence of vertices $v_0, v_1, \ldots, v_\ell$ such that $v_0 = s$, $v_\ell = t$, and $(v_i, v_{i+1}) \in E_{i \bmod k}$ for $i = 0, \ldots, \ell - 1$. That is, we are looking for walks that rotate between the digraphs $G_0, G_1, \ldots, G_{k-1}$ in the edges used.

   (a) Show that the problem of finding a Shortest Rotating Walk from s to t with respect to $G_0, \ldots, G_{k-1}$ can be reduced to Single-Source Shortest Walks via a reduction that makes one oracle call on a digraph G' with kn vertices and $m_0 + m_1 + \cdots + m_{k-1}$ edges, where $n = |V|$ and $m_i = |E_i|$. We encourage you to index the vertices of G' by pairs (v, j) where $v \in V$ and $j \in [k]$. Analyze the running time of your reduction and deduce that the Shortest Rotating Walk can be found in time $O(kn + m_0 + \cdots + m_{k-1})$. To test your reduction and algorithm, try running through the example in Part 3b.

   We simulate the rotating walks across the different graphs with a new graph G'. Where the vertices of G' each represent the state, represented as a pair (v, j), where v is the current vertex, and j is the j-th graph (formally graph G_j). This means that at (v, j) we are in vertex v in graph indexed j. We represent each of the edges in each graph G_j (which lead to a vertex in another graph) with an edge in G' between (v, j) and $(v', j+1 \mod k)$.

   Note that since each vertex is an encoding of each vertex at each graph, and there are n vertices on each of the k graphs, G' will have $k \cdot n$ vertices of (v, j) pairs representing each state in the rotating walk setup. Similarly, m' represents each of the possible

paths between each state in the original setup. We define $m' = m_0 + \ldots + m_{k-1}$, where $m_j = |E_j|$ is the number of edges in graph j. Since we are mapping all the possible states to vertices and all the possible routes between states to edges, our run-time is $O(n \cdot k + m')$.

After pre-processing, we can make an oracle call to Single-Source Shortest Walks on our graph G'. This oracle call finds the shortest path between $(s, 0)$ and a goal state (t, z), where z is the graph index for which the distance between $(s, 0)$ and some goal state that is on vertex t, is minimized. We don't care which $(t, \_)$ goal node we end up on, so long as it is on vertex t and it takes the least amount of distance to get there. This will properly represent how in the rotating walk setup, we want to take the shortest walk from vertex s to vertex t. In terms of the original setup, we don't care what graph we end up on, as long as we took the shortest path from s to t.

After our oracle call, we still need to do post-processing. We want our output to be a sequence of vertices $v_0, \ldots, v_k$, but it's currently in (v, j) pairs to represent different states. One solution is to map each pair to its corresponding vertex. So $(s, 0), \ldots, (t, z)$ is mapped to $s, \ldots, t$. The runtime of this step is $O(n \cdot k)$ since we're going through each of the $n \cdot k$ state pairs and just removing its graph index (effectively just mapping the pair to its corresponding vertex, which is a constant operation).

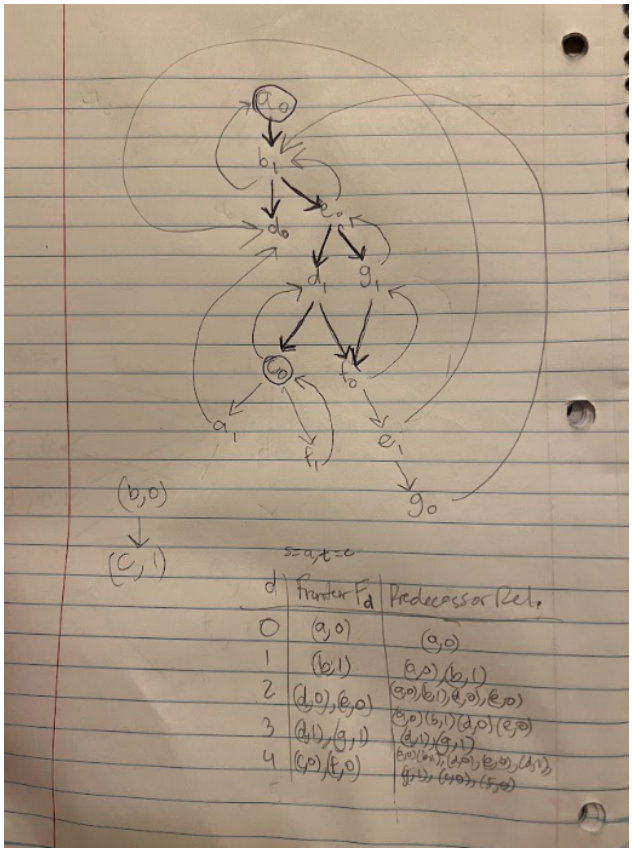
The runtime of the reduction is the runtime of the preprocessing + runtime of the post-processing, which is $O(nk + m' + nk) = O(2nk + m') = O(nk + m')$.

The runtime of our algorithm to solve the Shortest Rotating Walk is the runtime of our pre and post-processing + runtime of an algorithmic implementation of our oracle. We can solve our oracle using BFS, which we saw in lecture can find the shortest walk between two vertices, and it visits every node and vertex in the graph (in our case it visits all of G'). The combined runtime will be $O(\text{processing} + \text{BFS}) = O((2nk + m') + nk + m') = O(3nk + 2m') = O(nk + m')$.

This algorithm is correct because we know the shortest path will show the shortest path in G', and since G' simulates the rotating walk setup, this also proves that the shortest rotating walk can be found in the original setup. To prove this, we would only need to show that there is a 1:1 mapping from paths in the original setup to the paths in G'.

(b) Run your algorithm from Part 3a (by hand; no programming necessary) on the following pair of graphs G_0 and G_1 to find the Shortest Rotating Walk from $s = a$ to $t = c$; this will involve solving Single-Source Shortest Walks on a digraph G' with $2 \cdot 7 = 14$ vertices. Fill out the table provided below with the BFS frontier in G' at each iteration, labelling the vertices of G' as $(a, 0), (b, 0), \ldots, (g, 0), (a, 1), (b, 1), \ldots, (g, 1)$, and for each vertex v in the table, drawing an arrow in the graph from v's BFS predecessor to v.

Below is a drawing of the new graph G'. The BFS traversal is boldened in black ink.



Here is the table shown in the picture :

d	Frontier F_d	Predecessor Relationship
0	$(a, 0)$	$(a, 0)$
1	$(b, 1)$	$(a, 0), (b, 1)$
2	$(d, 0), (e, 0)$	$(a, 0), (b, 1), (d, 0), (e, 0)$
3	$(d, 1), (g, 1)$	$(a, 0), (b, 1), (d, 0), (e, 0), (d, 1), (g, 1)$
4	$(c, 0), (f, 0)$	$(a, 0), (b, 1), (d, 0), (e, 0), (d, 1), (g, 1), (c, 0), (f, 0)$

(c) A group of three friends decides to play a new cooperative game (similar to the real-life board game Magic Maze). They rotate turns moving a shared single piece on an $n \times n$ grid. The piece starts in the lower-left corner, and their goal is to get the piece to the upper-right corner in as few turns as possible. Many of the spaces on the grid have visible bombs, so they cannot move their piece to those spaces. Each player is restricted in how they can move the piece. Player 0 can move it like a chess rook (any number of spaces vertically or horizontally, provided it does not cross any bomb spaces). Player 1 can move it like a chess bishop (any number of spaces diagonally in any direction, provided it does not cross any bomb spaces). Player 2 can move it like a chess knight (move to any non-bomb space that is two steps away in a horizontal direction and one step away in a vertical direction or vice-versa). Using Part 3b, show that given the $n \times n$ game board (i.e., the locations of all the bomb spaces), they can find the quickest solution in time $O(n^3)$. (Hint: give a reduction, mapping the given grid to an appropriate instance

$(G_0, G_1, \ldots, G_{k-1}, s, t)$ of Shortest Rotating Walks.)

We perform the same reduction as we did in part a, with a few caveats. Since there are three players that take turns, we say there are three graphs (G_0, G_1, and G_2), so $k = 3$. In this case, our new graph G' will contain vertices that represent each possible state for each player (v, j), where v can be any position on the $n \times n$ board, and j represents the current player's turn, signifying graph G_j. The edges, M' of graph G', represent all the possible movements that can be made in the game. $M' = M_0, M_1, M_2$, where M_j represents all the possible moves that can be made by player j. This reduction will give us the correct answer because the task to reach the upper right corner in as few turns as possible is directly analogous to finding the shortest walk on a graph.

To show that they can find the quickest solution in $O(n^3)$ time, we analyze how many movements each player can make at any given point. Since player 0 can move like a rook, it can make up to $O(2(n-1)n^2)$ moves, which is $O(2n^3 - 2n^2)$. The same is for player 1, which can move like a bishop. Finally, player 2 can move like a knight, moving up to $8n^2$. This means that each player can find the quickest solution in $O(n^3)$.