1A Lent Algorithms

Supervision 4: Data Structures, Graphs

Hayk Saribekyan

February 27, 2020

When you are asked to describe an algorithm, also analyse its time and space complexity. Mention how you would implement the solution.

In this supervision we finish data structures and move to graph algorithms.

This example sheet is longer than I expected but all the problems are quite important to understand. Also many are just bookwork (e.g. warm up). I do not expect you to solve and write down precise solutions to everything, so think about the things you struggle with. You can go back to them during revision or when you have time later.

I would rather see a failed attempt or ideas handed in, than a beautifully written correct solution of a problem that you found easy. I do not mind if you skip a problem and write "I know how to do it." So focus on the concepts that you have most trouble with rather than writing down solutions to problems that are easy for you.

1 Refresher (Data Structures)

1. Show how to compute the number of valid sequences of parentheses of length n? For example, if $n = 4$ there are only two such sequences: `()()` and `(())`.

   Solution We will use DP to solve this problem. Let d_n be the number of valid parentheses sequences S of length n. Clearly, $S_1 =$ `'('`. Suppose the match of S_1 is at position $k \geq 2$. The number of such sequences is $d_{k-2} \cdot d_{n-k}$, because each possible sequence $S_{[2..k-1]}$ can be 'matched' with every possible sequence $S_{k+1..n}$. Thus,

   $$d_n = \sum_{k=2}^{n} d_{k-2} \cdot d_{n-k}$$

   These are Catalan numbers (`https://en.wikipedia.org/wiki/Catalan_number`), which have plenty of interesting properties.

2. Given a one directional linked list. Using only *constant* memory print the stored values in reverse order.

   Solution We first reverse the list and then print the values.

   `https://www.geeksforgeeks.org/reverse-a-linked-list/`

2 Data Structures Continued

Some problems are from the lecturer's notes[1] (Robert Harle). The ones marked with * are out of the curriculum and are for you to grasp the concepts better, go beyond the material and see how they can be used in real life.

1. (Robert Harle) Show how to delete hash table entries when resolving collisions using:

   i) chaining;

   ii) open addressing.

   Solution See lecture notes.

2. Given m sorted lists, each of length n. Explain how to compute their merge efficiently.

   Solution One solution is to repeatedly replace an arbitrary pair of lists with their merge. When two lists of length n_1 and n_2 are merged, $\Theta(n_1 + n_2)$ operations are required. It is important, however, to notice that the order in which these pairs are picked is important in the runtime.

   Suppose, we merge the lists in order i.e. repeatedly merge list $i > 1$ *into* merge 1. Merging list i will take $\Omega(in)$, thus the total runtime would be $\Omega(nm^2)$.

   Now, consider a different order of pairwise merges: at each step merge the two shortest lists. Notice that when merge-sorting an array of size nm bottom-up, at some point we have m arrays of size n, which get merged in exactly this order. Thus, the total runtime in this order is $O(nm \log m)$.

   There is a different approach that uses a priority queue, and essentially does the generalization of a standard merge operation. Place the first elements of all m lists into a priority queue Q. Pop the smallest element from Q (in $O(\log m)$ time), place in the final array. If the popped element was from the list i, then add the next element of list i into Q. The total number of iterations is nm, thus the runtime is $O(nm \log m)$.

3. (Robert Harle) How would you use a hash table to create a basic spell checker (i.e. decide whether or not a word iscorrectly spelt)? For an incorrectly-spelt word, how would you efficiently provide a set of correctly-spelt alternatives?

   Solution The rolling hash function is one example: `https://en.wikipedia.org/wiki/Rolling_hash`

4. * In the previous problem we use a hash table to determine correctly spelled words. Tries[2] are a beautiful data structure that could be used in such problems. Explain when you would prefer tries over hash tables and vice-versa.

[1] `http://www.cl.cam.ac.uk/teaching/1718/Algorithms/2018-examples-rkh.pdf`
[2] `https://en.wikipedia.org/wiki/Trie`

Solution As always the choice between the two depends on the problem. Trie is a predictable data structure, does not need to a (potentially bad) hash function, there are only a few possible ways of implementing it. The fact that there is little choice to make in trie implementation is also their weak point: to use hash tables, one has to decide what sort of table to use, what hash function to use. Thus, there is more room to tailor for a specific need.

5. * Given two strings a and b, determine if a is a contiguous substring of b. This problem is actually quite relevant in bioinformatics (a and b are gene sequences). There are beautiful and efficient algorithms to do this[3], but one can do it using hashes as well.

   Let the length of a be n and the length of b be m. If $hash(a) = hash(b[i..i+n))$ then we can be quite confident that a appears in b. This means, we have to compute $h_i = hash(b[i..i+n))$ for all $i = 0..m-n$. Computing h_i by itself clearly takes $\Theta(n)$ time simple computation of $h_1, h_2, \ldots, h_{m-n}$ could take $\Omega(nm)$ time.

   Can you come up with a function $hash$ for which, computing a single instance of $hash(a)$ still takes $\Theta(n)$ time, but computing the set of all $h_1, \ldots, h_{m-n}$ can be done in $O(m+n)$ time?

   Hint: You have to use the fact that the substrings of b corresponding to h_i and h_{i+1} are almost the same (differ by a character at each end).

   Solution Rolling hashes can be used here too. They have a nice property that $h(Sx)$ can be computed from $h(S)$ in $O(1)$ time; also $h(S)$ can be computed from $h(xS)$ in $O(1)$ time.

   Thus, we first compute $H_a = h(a)$ and $h(S) = h(b_{[1..n]}$ which takes $O(n)$ time. Then we slide S through b recomputing the hash value of $H_i = h(b_{[i..i+n)})$ for each i in $O(1)$ time. At each step we can compare H_i and H_a to determine if there is a match. This algorithm of course is vulnerable to collisions.

   The Knuth-Morris-Pratt algorithm is a much more beautiful solution to this problem (which is used by tools such as grep).

3 Graphs

Some problems are from the lecturer's notes[4] (Damon Wischik). The ones marked with * are out of the curriculum and are for you to grasp the concepts better and see how they can be used in real life.

1. (Damon Wischik) Draw an example of each of the following, or explain why no example exists:

   (i) A directed acyclic graph with 8 vertices and 10 edges;

   (ii) An undirected tree with 8 vertices and 10;

   (iii) An undirected graph without cycles that is not a tree.

[3] Knuth-Morris-Pratt, look up in Wikipedia.

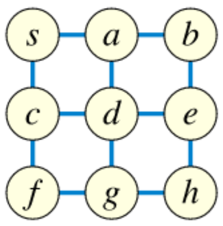
[4] `http://www.cl.cam.ac.uk/teaching/1718/Algorithms/ex5.pdf`

2. **This is perhaps the most important question in the whole example sheet.**

   Suppose we run a DFS from source s on a connected graph G (every vertex is reachable from every other one). Each vertex $v \neq s$ is visited from its "parent" p_v. The subgraph of G composed from edges (v, p_v) forms a tree, which we will call a DFS-tree. s is the root of the tree.

   The lecture notes implement `dfs`: an iterative version of DFS using a stack by just taking the BFS implementation and replacing the queue with a stack. There is also an implementation `dfs_recurse`.

   - Draw the two DFS-trees of the graph below that result from running both versions of DFS.

     Start from a tree containing just a single node s (root). Every time a vertex u becomes visited, add u to the the tree connected it to the vertex v it is visited from.

   - Compare these two trees. In particular, consider the edges that are not included *outside* in the DFS-trees.

     The tree $T_{recurse}$ that results from `dfs_recurse` should have the following property: for every edge (u, v) that is not in $T_{recurse}$, either u is an ancestor of v or vice-versa. This is a crucial property of a DFS-tree!

     Does this property hold for the tree T_{stack} of the function `dfs`?

   - Fix the `dfs` function to make sure that T_{stack} satisfies the property of a DFS-tree.

     This is not an implementation detail. The difference between these two implementations is not just the order in which vertices are visited, it's deeper. This is a major and embarrassing mistake in the `dfs` code that *was* unfortunately in Wikipedia and is in some renowned textbooks. The lecture notes still call the stack traversal algorithm a `dfs`.

   Note 2: Of course, it's a convention what we call a DFS-tree. The `dfs` from the lecture note does indeed traverse the graph, but this is not the only reason why we might want to use the DFS algorithm. The DFS-tree is used in many problems[5] hence it is "correct" and is the "convention." I do not know an example of a problem where the tree from the `dfs` is useful.

   Solution `https://11011110.github.io/blog/2013/12/17/stack-based-graph-traversal.html`

[5] e.g. `https://en.wikipedia.org/wiki/Lowest_common_ancestor`

3. Explain why Dijkstra's algorithm is just a generalisation of BFS. The question refers to the *implementation* of Dijkstra and BFS, not only the result. It's clear that the result of Dijkstra is the same as that of BFS when the graph is unweighted, simply because both algorithms are correct algorithms.

   Solution It is obvious that when a graph is undirected the *result* of Dijkstra's algorithm is exactly the same as that of BFS. The question, however, aims to explore the similarities of the *algorithms.* There are many shortest path algorithms that for an undirected graph would naturally produce the same output as BFS. But Dijkstra is special.

   Both algorithms keep a record vis_v showing whether vertex v is visited. At each step they both choose the closest unvisited vertex v and marks it as visited, and relaxes the distances to its neighbours.

   The difference between Dijkstra and BFS is in how they find the closets vertex that is not yet visited. By the nature of undirected graphs the sooner a vertex is visited, the sooner its neighbours will be visited. Thus, we use a queue that allows us to retrieve the closest node in $O(1)$ time. This invariant does not hold in general, so we have to use a priority queue.

4. Consider a graph G, where each edge has a weight of either 1 or 2. Modify the BFS algorithm to find the shortest paths from a source vertex s to all other vertices.

   Note: Plain BFS does not work in this case and Dijkstra's algorithm is too general. Your algorithm has to trade the generality of Dijkstra with speed.

   Solution Now that we have understood what in essence Dijkstra does, we can try to use the properties of G to make it faster (just like with BFS, where we used the unweightedness of our graph to use a queue instead of a priority queue).

   Notice that if we run Dijkstra on G with a priority queue, it will only have at most three distinct keys at any time, though it may have up to $|E|$ elements. If G was unweighted, the priority queue would have at most 2 distinct keys. This means we do not need $O(\log |E|)$ operations to retrieve the minimum element from the queue, just $O(1)$ (see previous supervision: priority queues with repeating keys). In practice, there would not be any need to actually have a priority queue with key counters.

5. * (Damon Wischik) Consider a graph without edge weights, and write $d(u, v)$ for the length of the shortest path from u to v. The diameter of the graph is defined to be $\max_{u,v \in V} d(u, v)$. Give an efficient algorithm to compute the diameter of an undirected tree, and analyse its running time. [Hint. Use breadth-first search.]

   Solution A neat solution using two BFS executions:

   `https://cs.stackexchange.com/questions/22855/algorithm-to-find-diameter-of-a-tree-using-bfs-df`

   There is an uglier but an easier (both to come up with and to prove) algorithm that uses dynamic programming programming. Suppose the tree is rooted, let d_p be the diameter of the subtree of vertex p and r_p be the longest path inside that subtree that starts at p. Try to find a recurrence relation for d_p and s_p using the values of their children.

6. State the invariant that holds in Dijkstra's algorithm and prove the correctness of the algorithm.

   Solution See lecture notes (or find online).

7. Suppose a graph G has edges with negative weight. Let the minimum of those weights be $w_0 < 0$. Consider the modified graph G', that consists of the same edges but the weights are different: for an edge e, $w'(e) = w(e) - w_0 \geq 0$, where w' and w are the weights of edges of G' and G respectively. Since G' has edges with non-negative weights, can we use Dijkstra's algorithm in G' and find the shortest paths in G?

   Solution No we cannot, because the change does not affect every $s \to t$ path the same amount.

8. We can implement Dijkstra's algorithm in $O(|E| \log |V|)$ time using priority queues (e.g. heaps). Using Fibonacci heaps we can improve it to $O(|V| \log |V|$, however that is quite complicated.

   Modify (actually simplify) Dijkstra's algorithm so that it runs in $O(|V|^2)$ time. Notice that if $|E|$ is close to $|V|^2$ this algorithm is faster.

   Solution Instead of using a priority queue, just do a loop over all the vertices and find the closest non-visited vertex. Then relax its neighbours. The resulting runtime is $O(|V|^2)$.

4 Implementation

Implementations are critical to sharpen your understanding of graph algorithms. I would suggest that you work on these during the break if you do not have time now.

1. Solve this problem using tries (see problem 2.4 above)

   `http://www.spoj.com/problems/PHONELST/`.

2. The following problem is quite challenging but shows a nice combination of two things we have learnt so far: hashing and binary search. It is related to problem 2.5 above.

   `http://www.spoj.com/problems/LPS/`

   Hint: Suppose you have a function `pal_exists(k)` that returns `true` iff the input string contains a palindrome of length k. Suppose it runs in $O(n)$ time where n is the length of the string. Then, you can do a binary search on the length of the palindrome, because if `pal_exists(k) == true` then `pal_exists(k - 2) == true` as well. Thus, your algorithm will run in $O(n \log n)$ time. Notice that you need to consider palindromes of even and odd lengths separately.

   The question now is, how to implement such a function. You can uses rolling hashes for that[6].

[6] `https://en.wikipedia.org/wiki/Rolling_hash`

3. Sharpen your graph algorithms skills

   `http://www.spoj.com/problems/EZDIJKST/`