Section 6: Exam II Review

Amy Zhao

March 01, 2024

1 General Exam Tips

The exam format should follow closely to the last exam, which means you should expect a mix of algorithm design, short answer, and true/false questions. Algorithm design questions typically fall in three main categories: (1) reductions, (2) algorithm utilization, and (3) algorithm modification. Category (1) is self-explanatory; this was the focus of Homework 4 and will likely appear on the exam with high probability. Category (2) refers to problems where you might need to run a graph algorithm twice (on different versions of the graph) or multiple different algorithms to combine information. Section 2.4 presents an example where the solution runs Dijkstra's twice. The bonus problem on Homework 3 involving food substitutions used Kosaraju's algorithm to find strongly connected components and another graph search algorithm to determine reachability. Category (3) problems have the lowest likelihood of showing up the exam, as you haven't really had the opportunity to practice. Examples of this kind of problem include counting shortest paths or identifying fuel capacity from Discussion Section 4.

As a recommendation, try to read through the entire exam before attempting any problem to get a gauge of how much time you might spend on each question. I understand that many people felt time constrained on the last exam; Prof. McKay will try his best to alleviate these constraints this time around. However, it would be useful to think briefly how you might divide your time, relative to the number of points allocated for each question.

2 Graphs

2.1 Removing But Not Disconnecting

Consider a connected, unweighted, undirected graph $G = (V, E)$. Our goal is to find a vertex $v \in V$ whose removal does not disconnect the graph. Note that removing v also removes any edges incident to it. We assume there is at least one such v; in the case there are multiple candidates, we can return any of them.

If we are unconcerned with runtime, we could simply iterate through the list of vertices and test each; for every vertex, we remove it from the graph and run DFS to determine if the modified graph remains connected. We replace the vertex before moving on to the next candidate.

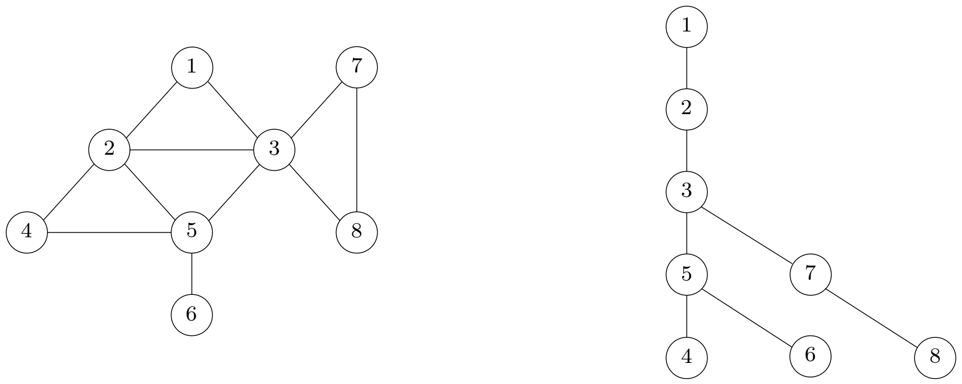

Figure 2.1: Example of a graph (left) and its DFS tree (right). The starting vertex 1 is the root of the tree, and every vertex u which is responsible for the discovery of v means we will have u as the parent of v in the resultant tree.

If we want to design an algorithm which is more efficient, we can run DFS once, starting from any vertex, and return any leaf vertex from the DFS tree generated from the algorithm. This is because removing a leaf vertex from a tree leaves behind a tree as well. An example of a DFS tree is given in Fig. 2.1. This version of the algorithm will run in $O(|V| + |E|)$.

2.2 Detecting Cycle Membership

Consider a simple setting where we have a connected undirected graph $G = (V, E)$ and a candidate edge $e \in E$. We want to design an algorithm to check whether e is part of a cycle in the graph.

If you took CPSC 202 with Prof. McKay, you might recall a proof that we completed for the following claim:

"*There is a cycle in a graph $G = (V, E)$ if and only if there exists vertices $u \in V$ and $v \in V$ such that there are two simple paths from u to v.*"

The main idea for this algorithm relies on this fact. If we remove the edge $e = (u, v)$ from the original graph G to create a new graph G', v should still be reachable from u when running any graph search algorithm (BFS or DFS) on G' due to the existence of two simple paths.

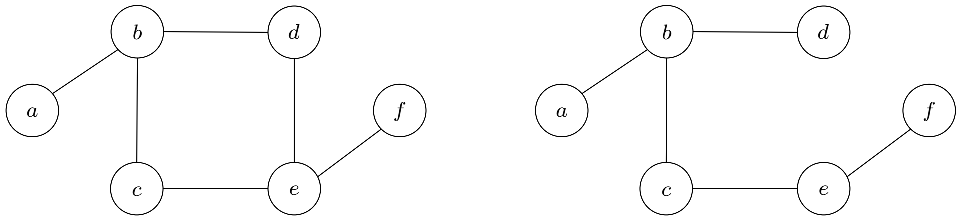

Figure 2.2: If $\{d, e\}$ is the candidate edge, we observe that if this edge is removed, e is still reachable from d if it is removed from the graph, through the path $d \to b \to c \to e$.

2.3 Odd Length Paths

Now, consider an unweighted directed graph $G = (V, E)$. Given a starting vertex s, we want to design an algorithm which returns all the vertices which are reachable from s in an odd length path. Note that such a path could contain a cycle and need not be the shortest path. Can you give an algorithm which runs in $O(|V| + |E|)$?

This is a clear example of a graph reduction problem, although it will use BFS or DFS on the resultant graph, rather than Dijkstra's, as we aren't considered with the shortest paths. For this reduction, we split every vertex $v \in V$ into two vertices $v^{(0)}$ and $v^{(1)}$. For every edge $e = (u, v) \in E$, we replace it with two edges $(u^{(0)}, v^{(1)})$ and $(u^{(1)}, v^{(0)})$ in the modified graph.

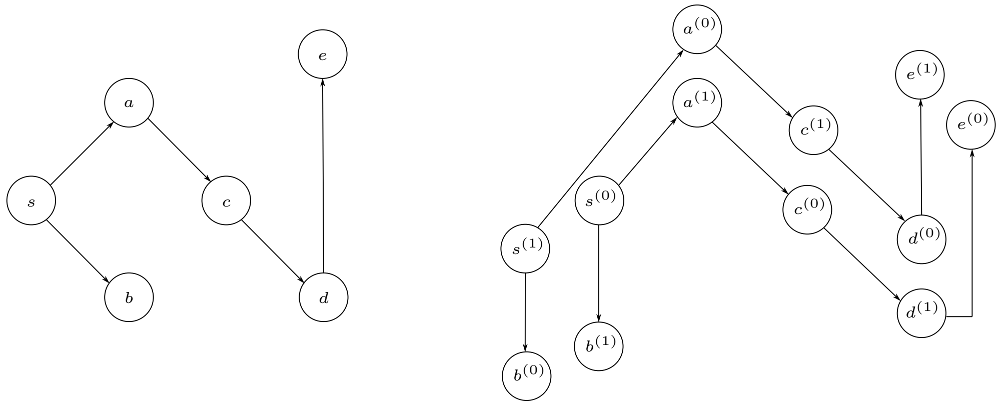

Figure 2.3: Example of an input graph (left) and the resulting modified graph (right) according to the algorithm description. The vertices a, b, and d are reachable with an odd length path.

We then run BFS or DFS on the resultant graph, starting from vertex $s^{(0)}$. To recover the solution, we iterate over the explored (reachability) array and return the vertices which are reachable and superscripted with (1) (i.e. $v^{(1)}$). The proof of correctness relies on the claim that there is an odd path from s to v in the original graph if and only if $v^{(1)}$ is reachable from $s^{(0)}$.

Suppose first there is an odd length path $p = (s, u_1), (u_1, u_2), \ldots, (u_k, v)$ from s to v in G. The corresponding path, which is a valid path following from the graph construction, in the modified graph is $p' = (s^{(0)}, u_1^{(1)}), (u_1^{(1)}, u_2^{(0)}), \ldots, (u_k^{(0)}, v^{(1)})$. In the converse direction, suppose there is a path p' from $s^{(0)}$ to $v^{(1)}$ given by $p' = (s^{(0)}, u_1^{(1)}), (u_1^{(1)}, u_2^{(0)}), \ldots, (u_k^{(0)}, v^{(1)})$. The value in the superscript alternates between 0 and 1, with the first superscript as 0, so this implies that p' must have an odd number of edges. Recovering the path over the original graph produces an odd length path from s to v.

Constructing the new graph and running BFS or DFS on the modified graph both take $O(|V| + |E|)$ time, so the overall runtime is $O(|V| + |E|)$.

2.4 Shortest Path Containing Specified Nodes

Suppose we impose an additional constraint to our shortest path problem. Given a directed weighted graph $G = (V, E)$ with n vertices and m edges with edge weight ℓ_{uv}, a source vertex s, a destination vertex t, and a subset of vertices $D \subseteq V$ of size k, design an algorithm which finds the shortest path from s to t containing at least one vertex from D.

The solution for this problem runs Dijkstra's twice: (1) once from s in the original graph, and (2) once from t in the reversed graph, where every directed edge is replaced by the one pointing in the reverse direction of the same weight. In doing so, we compute the shortest path distance from s to v and v to t for every vertex $v \in V$. To obtain the shortest overall path, we iterate over the vertices $d \in D$ and choose the vertex which yields the smallest $\mathrm{d}(s, v) + d(t, v)$ value, where $d(\cdot, \cdot)$ represents the shortest path computed via Dijkstra's.

Alternatively, you could solve this problem via a reduction. I'll leave this exercise to you.

2.5 Four Cycle

This problem relies on matrix multiplication rather than a graph algorithm. Taking an undirected graph $G = (V, E)$, we want to determine whether G contains a four cycle.

Suppose G is given as an adjacency matrix A. We will compute $B = A^2$ and $C = A^4$; the entries B_{ii} and C_{ii} correspond to walks of length 2 and 4 which begin and end at the same vertex. Notice that these walks of length 4 can either be two walks of length 2 from i back to i of the form $i \to j \to i \to k \to i$ or walks of the form $i \to j \to k \to j \to i$ or a cycle of length four. Thus, the number of cycles of length 4 can be calculated via:

$$\text{num. of four cycles} = C_{ii} - B_{ii}^2 - \sum_{j \neq i} B_{ij}.$$

If the value calculated from this formula is non-zero, then the graph contains a four cycle. The runtime of this algorithm is dominated by the time it takes to multiply two matrices, which we typically express as $O(T_m(n))$, where n is the number of vertices.

3 Data Structures

3.1 Heap Loops

Given an input which consists of k sorted lists $L_1, \ldots, L_k$, each containing a list of n/k integers in increasing order, we want to create an algorithm which outputs a single list that sorts all n integers in increasing order.

The basic idea of this algorithm is to maintain a min heap that contains k elements, specifically the smallest integer from each of the k sorted lists. To elaborate, we first build a min heap of k elements: $(\ell_1, 1)$, $(\ell_2, 2)$, $\ldots$, (ℓ_k, k). Each ℓ_j is the smallest element in the sorted list A_j and is used as the heap key.

We repeatedly perform the following steps: extract the minimum from the heap (based on key) and add it to the output list. Denote this element as (x, i) To maintain the number of elements in the heap, we add the next element from A_i to the heap and re-structure the heap accordingly. If A_i is empty, we skip this step and the number of elements in the heap decreases by one.

The correctness of this algorithm uses the fact that each list A_i is sorted. In the first iteration, the minimum extracted from the heap is the overall minimum because we've taken each minimum from every list and created a heap of these local minimums. At every subsequent iteration, we are functionally comparing the minimum of each list's unseen elements.

The initial construction of the heap requires the insertion of k keys; if done through repeated insertions, this process takes $O(k \log k)$ time. Each time we extract an element from the heap, we incur $O(\log k)$ time. If we perform an insert, this also takes $O(\log k)$ time. In total, there are n elements, so this process is repeated n times. Thus, the total runtime of this algorithm is $O(n \log k)$.

3.2 Analyzing Hash Structures

Suppose we want to hash elements of a set of keys U into m slots. Show that if $|U| > (n-1)m$, there is a subset of U of size n consisting of keys that all hash to the same slot, so the worst case search time for a hash table with chaining is $\Theta(n)$. This problem relies on the pigeonhole principle which states that if x items are put into y containers with $x > y$, then at least one container must contain more than one item.

If we map $(n-1)m+1$ keys into a table of size m, there must be at least one slot with n keys or more, following the pigeonhole principle; if each slot held at most $n-1$ keys, there would only be at most $(n-1)m$ keys. So, the $(n-1)m+1$-th key inserted into the hash table must go into some slot which already has $n-1$ keys. When searching the hash table, we'll search the entire chain of items in this slot, which translates to $\Theta(n)$ time.