CS 124 Homework 3: Spring 2024

Collaborators:

No. of late days used on previous psets:
No. of late days used after including this pset:

Homework is due Wednesday Feb 28 at 11:59pm ET. Please remember to select pages when you submit on gradescope. Each problem with incorrectly selected pages will lose 5 points.

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential-time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or no credit. Again, try to make your answers clear and concise.

Problems

1. **Detecting whether an edge lies in an MST (10 points)**

   (a) **(5 points)** Let G be a weighted graph in which all edge weights are distinct. Prove that an edge e of a graph G belongs in some MST of G if and only if the following property holds: for every cycle in G that contains e, the edge with the highest weight is not e.

   Direction 1: If e belongs in some MST of G, then for every cycle of G containing e, the edge wigh the highest weight is not e.

   Let M be any MST of G containing e. Let c be any edge in G but not M such that adding c to M produces a cycle containing e. By definition of cycle, the vertices connected by e now have a path to each other not using e. So, remove e from M. Because e is no longer in the cycle, the cycle is broken. This leaves a new spanning tree in the graph. This new spanning tree is identical to the previous spanning tree except it contains c instead of e. Because edge weights are distinct, this new spanning

tree with c is either lighter or heavier than the tree using e. If it were lighter, then that would contradict the assumption that M is an MST using e because it would have used c instead. So, it must be heavier. This implies that c is heavier than e and thus e is not the heaviest edge. This proves the assertion that for every cycle of G containing e, the edge with the highest weight is not e.

Direction 2: If for every cycle in G containing e the edge with the highest weight is not e, then e belongs in some MST of G.

Let C be any cycle in G containing e. Let h be the heaviest edge in that cycle. By the claim's assumption, assert that e and h are distinct. Consider any partitioning of the vertices in C such that the only two edges in C crossing the cut are e and h. By the cut property, the least of these edges will be in an MST of G. Because e is lighter than h, the construction of such an MST will choose e over h to connect the two cuts, implying that e will be in some MST of G.

(b) **(5 points)** Given an edge e in G, give an algorithm that outputs YES if there exists an MST of G that contains e, and NO otherwise. Your algorithm must have runtime asymptotically faster than the algorithms given in class for finding MSTs. You must describe your algorithm, prove its correctness and state its run time. You need not prove the runtime. (You may use the result from Part (a) even if you did not prove it.)

Consider this algorithm. In the algorithm, let v_1 and v_2 denote the two vertices connected by e:

i. Remove e from G.
ii. Initialize a Visited set and a PathWeights set for use in the DFS.
iii. Run a modified DFS on the graph starting at v_1. In the DFS, track the maximum-weight edge traversed on the current path. If the DFS ever reaches v_2, add the maximum weight of that path to the PathWeights set.
iv. Let G' be the G without v_2. If DFS did not visit every vertex in G', return NO.
v. If the PathWeights set is empty, return YES.
vi. If e is heavier than any value in PathWeights, return NO. Else, return YES.

Correctness:
If the algorithm returns something, it is correct about the MST membership of e: First, if the DFS didn't visit every vertex in G', then the graph is not connected even with e added back. The vertex v_2 is excluded because it might be unreachable without e. If the graph is not connected, then a spanning tree cannot exist, and e can subsequently not be part of such a tree. Here, it's correct to return NO. Otherwise, the DFS will have collected the maximum weight edges on every path from v_1 to v_2. If no such path exists, then e is the only edge the connect v_1 and v_2. For the tree to span, it must include e. So, it's correct to return YES. Finally, if e is lighter than the heaviest edge on every path from v_1 to v_2, then e is also not the heaviest edge in every cycle containing e. By part a, it is correct to return YES. Otherwise, NO is returned, which is also correct due to the iff nature of part a.

If e is in an MST of G, then the algorithm returns YES: If e is in an MST of G, then such an MST exists. This implies all vertices in the graph are connected, and the DFS will visit the entire graph except possibly v_2. So, the algorithm will not return NO from

that case. If e is the only edge to v_2 from the rest of the graph, then the algorithm will correctly return YES because e is required to make the tree span the entire graph. Otherwise, according to part a, because e is in an MST of G, the maximum-weight edges collected in the PathWeights set must all be greater than e. The final check will observe this and return YES.

Runtime:

The visited set is already part of DFS. Tracking the maximum weight edge along a path and maintaining PathWeights takes $O(1)$ time. So, DFS still takes time $O(E+V)$ where E is the number of edges and V is the number of vertices. This also bounds the size of PathWeights. Thus, the runtime is that of the DFS, $O(V+E)$.

2. **Maximal independent set in an evolving graph (30 points):**

Given an undirected graph $G = (V,E)$, we say that a subset $S \subset V$ is an *independent set* if no two vertices in S are connected by an edge. We say that S is a *maximal independent set (MIS)* if S is an independent set and furthermore there is no strict superset T (i.e., a set T with $S \subsetneq T$) which is also an independent set. In this problem we will explore the running time of algorithms computing and maintaining maximal independent sets in graphs with n vertices, m edges and with maximum degree Δ (i.e., every vertex $u \in V$ has at most Δ edges touching it.)

(a) **(5 points)** Give an algorithm that finds a maximal independent set of G, given G in the adjancency list representation. (You must describe your algorithm fully and give a brief explanation of why it is correct. You should state your runtime but you don't need to prove it.)

Consider this algorithm: Given an adjacency list graph called `adj`, initialize an empty set called `ind_set`. While `adj` is not empty, pop a vertex `v` and its edges from `adj` and add `v` to `ind_set`. Then, for every neighbor in edges, delete that neighbor from `adj`. At the end, return `ind_set`. Here's pseudocode for such an algorithm:

```
fn max_ind_set(adj: adjacency list graph) -> Set(vertices):
  let ind_set = {};
  while adj is not empty:
      let (v, edges) = pop entry from adj;
      ind_set.add(v);
      for neighbor in edges:
          adj.delete(neighbor);
  return ind_set;
```

Correctness:

Lemma, `ind_set` is an independent set and the vertices in `adj` share no edges with vertices in `ind_set`.

- Base case: `ind_set` is an independent set initially because it's empty. There are also trivially no edges between `ind_set` and `adj`.
- Inductive hypothesis: after $0 \leq n$ iterations of the `while` loop, (1) `ind_set` is an independent set and (2) vertices `adj` share no edges with vertices in `ind_set`.
- Inductive step: at the $n+1$ iteration of the while loop, `v` is popped from `adj`. By the inductive hypothesis, `v` shares no edges with vertices in `ind_set`. So,

adding it to `ind_set` maintains `ind_set` as an independent set. Then, every vertex sharing an undirected edge with `v` is deleted from `adj`. After this, `adj` contains no vertices with edges to `v`. By the inductive hypothesis, it also contains no edges with vertices previously in `ind_set`. This preserves the inductive hypothesis and thus proves the claim inductively.

Proof of correctness: From the lemma above, assert that once `adj` is empty, `ind_set` is an independent set and there are no vertices remaining in the graph that don't share an edge with a vertex in `ind_set`. This implies that the value returned from `max_ind_set` is a maximal independent set. Similarly, because a maximal independent set will always exist in a non-empty graph, if a maximal independent set exists in the input graph, one is found and returned by `max_ind_set`.

Runtime:
The algorithm iterates every vertex and every edge in the graph, so it runs in time $O(V+E)$.

Now suppose the graph G is changing over time, and we want to maintain a maximal independent set of this graph without having to recompute it from scratch every time G is updated. Concretely, suppose that in each time step, some edge is either added to or deleted from G. Our basic data structure simply maintains a set $S \subseteq V$ in the form of an array indexed by V such that $S[u] = 1$ if $u \in S$ and 0 otherwise.

(b) **(0 points, not to be turned in)** Prove that the adjacency lists can be maintained with $O(\Delta)$ cost per insertion and deletion. (You may assume you have a solution to this problem in future parts even if you did not solve it.)

(c) **(7 points)** Describe algorithms INSERT(e) and DELETE(e) to maintain S under edge insertions and deletions respectively. Give upper bounds on the runtime of both operations. (Your algorithm and its claimed run times must be correct, but you need not prove these. Note that for full credit your runtimes should depend only on Δ and not on n.)

(d) **(3 points)** For every integer $\Delta > 0$ describe an example (i.e., a graph, an MIS, and an edge to be inserted) such that the number of queries to S for inserting an edge asymptotically match your upper bound from Part 2c.

To speed up the runtimes from Part 2c, suppose we decide to additionally maintain an array A indexed by V, such that for every $u \in V$, $A[u]$ counts the number of neighbors of u that are in S. (So $A[u] = |\{v \in V | v \in S, (u,v) \in E\}|$.)

(e) **(5 points)** Give algorithms A-INSERT and A-DELETE that maintains both S and A under edge insertions and deletions. (While any correct polynomial-time algorithm will get you full points, needlessly inefficient algorithms will lose points in the next part!)

(f) **(10 points)** Assume that initially the graph G is empty (no edges), and S consists of all vertices in V. Give an amortized analysis proof that after T updates to G, the total runtime of the operations is at most $O(\Delta \cdot T)$. (Hint: Consider a charging scheme that charges the runtime of adding a vertex to S to the vertex itself. You should be careful to pay this charge when the same vertex is deleted from S!)

3. **Sorta sorting with heaps (24 points)** Explain how to solve the following two problems using heaps. (No credit if you're not using heaps!)

   (a) **(12 points)** Give an $O(n \log k)$ algorithm to merge k sorted lists with n total elements into one sorted list.

   (b) **(12 points)** Say that a list of numbers is k-close to sorted if each number in the list is less than k positions from its actual place in the sorted order. (Hence, a list that is 1-close to sorted is actually sorted.) Give an $O(n \log k)$ algorithm for sorting a list of n numbers that is k-close to sorted.

4. **d-heaps (0 points, optional)**[1] Consider the following generalization of binary heaps, called d-heaps: instead of each vertex having up to two children, each vertex has up to d children, for some integer $d \geq 2$. What's the running time of each of the following operations, in terms of d and the size n of the heap?

   (a) delete-max()

   (b) insert(x, value)

   (c) promote(x, newvalue)

   The last operation, promote(x, newvalue), updates the value of x to $newvalue$, which is guaranteed to be greater than x's old value. (Alternately, if it's less, the operation has no effect.)

5. **Suboptimality of greedy algorithm for set cover (10 points)** Give a family of set cover problems where the set to be covered has n elements, the minimum set cover is size $k = 3$, and the greedy algorithm returns a cover of size $\Omega(\log n)$. That is, you should give a description of a set cover problem that works for a set of values of n that grows to infinity – you might begin, for example, by saying, "Consider the set $X = \{1, 2, 3, \ldots, 2^b\}$ for any $b \geq 10$, and consider subsets of X of the form...", and finish by saying "We have shown that for the example above, the set cover returned by the greedy algorithm is of size $b = \Omega(\log n)$." (Your actual wording may differ substantially, of course, but this is the sort of thing we're looking for.) Explain briefly how to generalize your construction for other (constant) values of k. (You need not give a complete proof of your generalization, but explain the types of changes needed from the case of $k = 3$.)

6. **Tracking components in an evolving graph (15 points)** You are secretly Gossip Girl, an anonymous gossip blogger who keeps track of friendships at Constance Billard High School. You publish an up-to-date map of the friendships at Constance on your website.[2] You maintain this map by a stream of distinct tips from anonymous followers of the form "A is now friends with B."

   (a) **(5 points)** You call some groups of people a "squad": each person is in the same squad as all their friends, and every member of a squad has some chain of friendships to every other member. For example, if Dan is friends with Serena, Serena is friends with Blair, and Alice is friends with Donald, then Dan, Serena, and Blair are a squad (You make up the name "The Gossip Girl Fan Club") and Alice and Donald are another squad ("The Constance Constants"). Give an algorithm that takes in a stream of (a) tips and (b) requests for a specified person's squad name. You should answer requests that come in between tips consistently—if

[1] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.

[2] There are no ethical concerns here, because you're a character in a highly-rated teen drama.

you make up the name "The Billard billiards players" for Dan's squad, and you're asked for Serena's squad's name before any new tips come in, you should report that it's "The Billard billiards players".

(b) **(10 points)** A "circular squad" is defined to be a squad such that there is some pair of friends within the group that have both a friendship and a chain of friendships of length more than 1. In the example above, if Dan and Blair also became friends, then the group would be a circular squad. If Dan and Donald also became friends, they would all be in one circular squad. Modify your algorithm from the previous part so that you report names that contain the word "circle" for all circular squads (and not for any other squads).

7. **Greedy scheduling (35 points)** Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, \ldots, j_n$ that we have to process. To process a job, we need to assign it to one of the two machines; each machine can only process one job at a time. Each job j_i has an associated positive integer running time r_i. The load on the machine is the sum of the running times of the jobs assigned to it. The goal is to minimize the completion time, which is the maximum of the load of the two machines.

   Suppose we adopt a greedy algorithm: for every i, job j_i is assigned to the machine with the minimum load after the first $i-1$ jobs. (Ties can be broken arbitrarily.)

   (a) **(5 points)** For all $n > 3$, give an instance of this problem for which the completion time of the assignment of the greedy algorithm is a factor of $3/2$ away from the best possible assignment of jobs.

   (b) **(15 points)** Prove that the greedy algorithm always yields a completion time within a factor of $3/2$ of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.)

   (c) **(10 points)** Suppose now instead of 2 machines we have m machines and the completion time is defined as the maximum load over all the m machines. Prove the best upper bound you can on the ratio of the performance of the greedy solution to the optimal solution, as a function of m?

   (d) **(5 points)** Give a family of examples (that is, one for each m – if they are very similar, it will be easier to write down!) where the factor separating the optimal and the greedy solutions is as large as you can make it.