Sample Midterm: CSCI 6212 Algorithms

Policies: Test will be closed book, closed notes. You may bring in one sheet of paper. Tests will be taken individually, with not conversation or discussions with other classmates.

1. Short Answer

   (a) Suppose that in the Gale-Shapley algorithm, a mans proposal has just been accepted. True or false: He is guaranteed to remain engaged (to this person or someone else) for the remainder of the algorithms execution.

   FALSE. A the man's engagement may be broken off if his fiance is proposed to by someone higher on her list.

   (b) Suppose that in the Gale-Shapley algorithm, a woman has just accepted a proposal. True or false: She is guaranteed to remain engaged (to this person or someone else) for the remainder of the algorithms execution. *TRUE. once engaged a woman may accept an offer from a "better" man, but will never become unengaged.*

   (c) As a function of n, what is the asymptotic running time of the following function? (Express your running time using Θ notation.)

```
void scareMe(int n) {
i = n;
    while (i > 0) {
    for (j = 1 to i) print("boo!\n");
        i = floor(i/2);
    }
}
```

   $\Theta(n)$. *The loop runs in time:* $n + \frac{n}{2} + \frac{n}{4} + \ldots$, *which converges to 2n.*

2. Consider using a simple linked list to store items, and assume you will never get duplicate elements. You use a simple linked list as a dictionary, insert any new elements at the beginning of the list. Suppose in your problem domain, there may be any number of insert operations, but at most one lookup operation.

   (a) What is the worst-case running-time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

   $\Theta(n)$. *Insertion in this domain may be done in constant time, because you can just append new elements to the beginning or end of the list. Search takes* $\Theta(n)$ *time; but if only one search is run, that search takes at most n time, and total cost of all n operations is* $\Theta(n)$.

   (b) What is the worst-case amortized running-time of the operations performed on this data structure under the assumptions above? Briefly justify your answer.

   The amortized run time per operation is O(1), because n operations take $\Theta(n)$ *time.*

3. Consider the recurrence $T(n) = 4T(\frac{n}{2}) + n$, Using Θ-notation, give a tight asymptotic bound on $T(n)$. (E.g., $T(n)$ is $\Theta(n)$ or $\Theta(nlogn)$, etc.)

$$
\begin{aligned}
T(n) &= 4T(\frac{n}{2}) + n \\
&= 4(4T(\frac{n}{4}) + \frac{n}{2}) + n \\
&= 16T(\frac{n}{4}) + 4\frac{n}{2} + n \\
&= 16T(\frac{n}{4}) + 2n + n \\
&= 16(4T(\frac{n}{8}) + \frac{n}{4} + 2n + n \\
&= 64T(\frac{n}{8}) + 16\frac{n}{4} + 2n + n \\
&= 64T(\frac{n}{8}) + 4n + 2n + n \\
&= 64T(\frac{n}{8}) + 4n + 2n + n \\
&= 4^3T(\frac{n}{8}) + 2^2n + 2^1n + 2^0n \\
&= 4^3T(\frac{n}{8}) + n(2^2 + 2^1 + 2^0) \\
&= 4^3T(\frac{n}{8}) + n\sum i = 0^{3-1}2^i \\
&= 4^3T(\frac{n}{8}) + n(2^3 - 1)
\end{aligned} \tag{1}
$$

After k substitutions, we have

$$
= 4^kT(\frac{n}{2^k}) + n(2^k - 1) \tag{2}
$$

and when $k = logn$, this becomes:

$$
\begin{aligned}
&= 4^{logn}T(\frac{n}{n}) + n(2^{logn} - 1) \\
&= n^2T(1) + n(n-1),, \text{ but T(1) is constant, so} \\
&= \Theta(n^2)
\end{aligned}
$$

4. Let G be a connected undirected graph $G = (V, E)$ in which each edges weight is either 1 or 2. Present an $O(n + m)$ time algorithm to compute a minimum spanning tree for G, where $n = |V|$ and $m = |E|$. Explain your algorithms correctness and derive its running time. (Hint: This can be done by a variant of DFS or BFS.)

   Two possible solutions:

   Solution 1: Prim's algorithm grows the spanning tree by maintaining a tree and keeping a list of edges that may connect a vertex in the tree to a new vertex. Part of the run time of Prim's algorithm is keeping that list of edges sorted, in order to connect a new vertex to the tree with the lightest weight edge possible. If all edges have weight 1 or 2, we can keep a sorted list in linear time (for example, we can keep a list of all the edges we know of with weight 1, and another list with all edges of weight 2. When we want to find the lightest edge, we always *choose one of the edges of weight 1 if there are any in that list, otherwise we choose any edge from weight 2.*

Because all list operations are then constant, it takes O(E) to build the lists. Prim's algorithm builds 1 tree (connected to the starting point) and we can just label which points we've found, so the "find" operation checking to see if an edge connects two trees is also O(1). Therefore, the algorithm runs in O(V + E) time.

Solution 2:

Step 1. Run DFS (or BFS) just using edges of weight 1 until you are stuck. If you have found all nodes in the graph, then you are done (because you have an MST of all light edges!). If not, restart the DFS with a node that you didn't find at first, and keep doing this until you have have used all the weight 1 edges, (either connecting thing as trees, or ignoring them because they connect two nodes that are already connected by weight 1 edges).

You now have what's called a "forest" (get it? a bunch of trees...) make from weight 1 edges, and none of the trees in that forest are connected by anything but weight 2 edges.

Step 2. Now make a new graph, where there is a node in the new graph for every tree from the weight 1 graph, and there is an edge those nodes if there is a weight 2 edge between any node in those trees. Edges in this new graph connect the nodes that correspond to trees, but each edge comes from some original edge in the original graph, so keep a pointer to that original edge so we can look it up later.

Run DFS (or BFS) on this new graph. The edges chosen in the new graph define connections between the trees, and we can look up which original edges make those connections.

Runtime: Let's define m_1 as the number of edges of weight 1, and m_2 as the number of edges of weight 2. The step 1 runtime is $O(m_1 + n)$ because there are m_1 edges of weight 1 and n vertices. The step 2 runtime is $O(m_2 + n)$ because there are m_2 edges of weight 1 and at most n vertices. So the overall runtime is $O((m_1 + m_2) + n) = O(m + n) = O(V + E)$

Correctness:

Argument 1 (a normal style argument): If the MST is light than our solution, then the MST must use more weight 1 edges than we use. However, we consider all weight 1 edges, and include all those that do not create a cycle. Therefor we are using the maximum possible number of weight 1 edges, so our MST has the minimum possible weight.

Suppose there was a lighter MST of the whole graph than was created by either algorithm. Consider every edge that our algorithm added, and the cut defined

Argument 2 (a clever argument): Kruskals algorithm starts by sorting the edges. Both algorithms 1, 2 consider exactly the same edges as Kruskals (in an order that Kruskal's algorithm might use if the sorting algorithm broke ties the same way) — and both algorithms include exactly the same set of edges that Kruskal's algorithm would. Therefore, because it considers the same edges in the same order, and because Kruskal's algorithm is correct, then our algorithm(s) are also correct.

5. Give an example of a directed graph with negative-weight edges for which Dijkstras algorithm produces incorrect answers. Show why Dijkstras algorithm fails.

   Dijkstra's algorithm fails on the above graph because it first expands node t and will therefore not ever explore the alternative path (of two hops) from s to t. If there are no negative weight

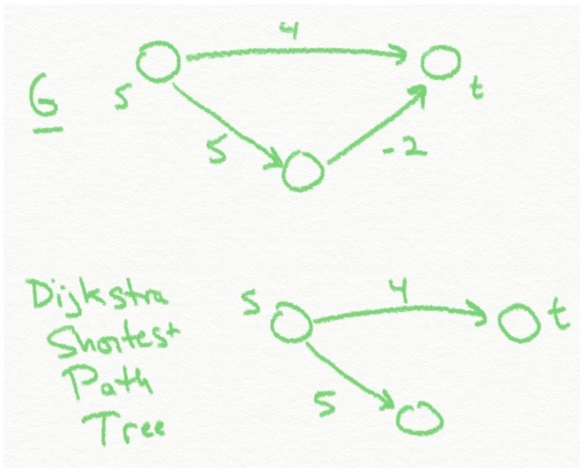


Figure 1: A graph with a negative edge weight and the shortest path tree returned by Dijkstra.

edges, this is safe, but when there are negative weight edges, like in this case, the alg. may miss short paths.

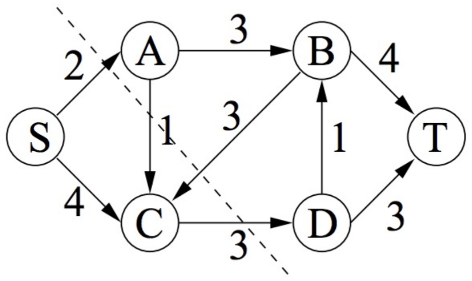
6. Edge-disjoint paths Given a directed graph $G = (V, E)$ with vertices $s, t \in V$, give an algorithm that finds the maximum number of edge-disjoint paths from s to t.

   Assign a weight of 1 to edge edge and run the max-flow algorithm in this graph. The amount of flow is equivalent to the number of edge disjoint paths. A great discussion/proof for this problem is available on the first two and a half slides here: `https://www.cs.princeton.edu/courses/archive/spring05/cos423/lectures/07maxflow-applications.pdf`

7. Alice wants to send Bob a file over a wired network. Their computers are connected to each other by multiple routers and multiple different routes. Given the bandwidth of each connection (edge weights), Alice needs to calculate the route that can send the file the fastest. How do you modify Dijkstras algorithm to find the path with the highest (minimum) bandwidth?

   This is the "Bandwidth" problem on the following page: `http://www.csl.mtu.edu/cs2321/www/newLectures/30_More_Dijkstra.htm`

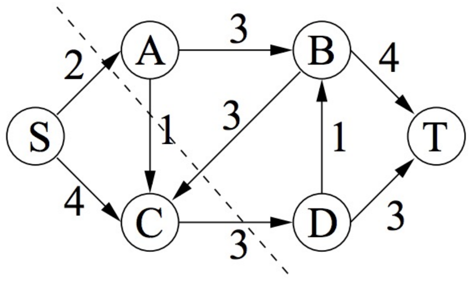
8. Problem 1. (12 points) Network Flows In the flow network illustrated below, each directed edge is labeled with its capacity.



   (a) The dotted line represents a cut. Is this a minimum cut separating S from T?

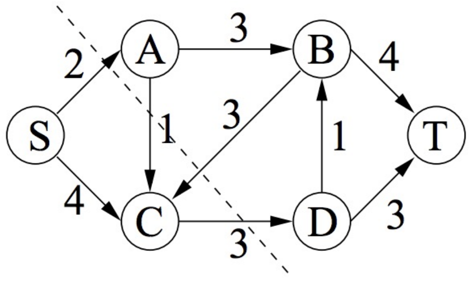
   yes

(b) If you run the Ford Fulkerson Algorithm, what is a possible result in terms of flow after 1 iteration? Show the path chosen and the flow along that path?



A written description of the path is: Send 2 units of flow along S to A to B to T

(c) Show a possible solution at the conclusion of the Ford Fulkerson algorithm with maximum flow



Send 2 units of flow along S to A to B to T
Send 3 units of flow along S to C to D to T