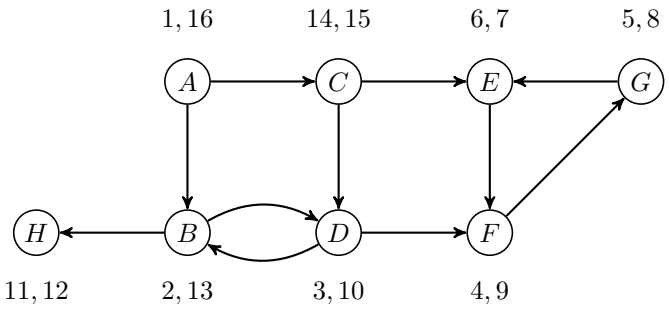
Midterm 1 Solutions

Problem 1 (24 points)

Do a depth-first search of the directed graph below. Process nodes (and edges out of a node) in alphabetic order. Show the **pre** and **post** order numbers on the figure.



List the strongly connected components of this graph in one of the possible orders in which they would be returned by the algorithm in class (depth-first search the reverse graph, etc.). No need to run the algorithm.

SCCs in reverse linearized order: $\{H\}, \{E, F, G\}, \{B, D\}, \{C\}, \{A\}$

What is the DAG of strongly connected components?

Nodes are $\{\{H\}, \{E, F, G\}, \{B, D\}, \{C\}, \{A\}\}$.
Edges are as follows:
from $\{A\}$ to $\{C\}$ and $\{B, D\}$
from $\{C\}$ to $\{B, D\}$ and $\{E, F, G\}$
from $\{B, D\}$ to $\{E, F, G\}$ and $\{H\}$

What is the minimum number of edges you have to add to make the graph strongly connected? Justify your answer very briefly.

We have 2 sinks $\{E, F, G\}$ and $\{H\}$, so we need at least 2 edges. Connecting them both to A with an edge works.

Problem 2 (10 points)

You want to multiply a polynomial of degree 1 with a polynomial of degree 2. On how many points will you do FFT (smallest appropriate power of two)? What is the corresponding ω?

The final polynomial will have degree 3, so it has to be evaluated at 4 points. Hence the corresponding ω is the 4th root of unity, i.

Suppose that the two polynomials are x and x^2. What are the two outputs of the FFT?

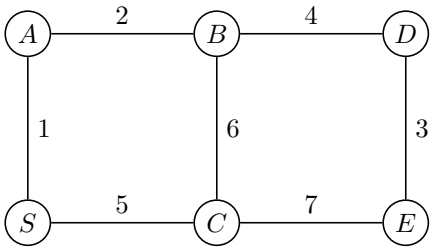
We evaluate both polynomials at points $1, i, -1, -i$. So, for x we get $1, i, -1, -i$ and for x^2 we get $1, -1, 1, -1$

Describe briefly how you will find the product from those outputs (don't do it).

Take the pairwise product of the two FFT outputs, and then compute the inverse FFT on the result.

Problem 3 (15 points)

You are about to perform the third step of an algorithm on the graph below.



Which node(s) would be processed next if the algorithm is:

(a) Prim's algorithm for the MST starting from S.
Starting with S, add nodes at minimum distance to any node already added: add A, B, and finally D.
So D is the third.

(b) Kruskal's algorithm (greedy) for the MST.
Add edges in decreasing order of weight: SA, AB, DE, so nodes D and E.

(c) Dijkstra's algorithm for shortest paths starting from S.
Add vertices in order of minimum distance to S: A, B, C, so C is the third.

(d) In (b), how would the disjoint set data structure look after the third step?
A, B hanging from S, E hanging from D, C on its own.

Problem 4 (16 points)

TRUE or FALSE? No justification is needed. No points will be subtracted for wrong answers, so it is to your best interest to guess all you want.

1. If a directed graph has a source and a sink, it is a dag.
   F: It could have a cycle in the middle (like the graph in question 1).
2. $T(n) = 4T(n/4) + n \log n$ has solution $\Theta(n \log n)$.
   F: $T(n) = \sum_{i=0}^{k} 4^i \frac{n}{4^i} \log \frac{n}{4^i} = n \sum_{i=k}^{0} \log 4^i = 2n\Theta(k^2) = \Theta(n(\log n)^2)$
3. The fastest algorithm to find the median uses sorting.
   F: We can do it in linear time, without the use of sorting. See textbook, chapter 2.
4. Without path compression, the union-find data structure has $\log n$ worst case.
   T: Use union-find by rank.
5. With path compression, the union-find data structure has $\log^* n$ worst case.
   F: It is $\log^* n$ **amortized** cost, not worst case cost (which is still $\log n$).
6. The FFT of $(0, 0, 0, 0)$ is $(0, 0, 0, 0)$.
   T: The FFT is a linear operator (matrix), so it always maps 0 to 0.
7. The minimum spanning tree always contains the shortest edge (assume it is unique).
   T: In fact both algorithms we saw in class will start by adding this shortest edge to the tree.
8. The shortest path from S to A always contains the shortest edge (assume it is unique).
   F: The shortest edge might not even be on a path from S to A.
9. The minimum spanning tree never contains the longest edge.
   F: Consider the case where the graph is already a tree. In that case we must include all edges in the MST (including the longest one).
10. There are 12 numbers smaller than 21 who are relatively prime to 21.
    T: Just notice that 1 is co-prime to any other number (it is not a prime number, but it is still co-prime with other numbers, as $\gcd(1, n) = 1$ for all n).
11. In RSA with $p = 3$ and $q = 7$, $e = 11$ is a legitimate choice.
    T: Since 11 is co-prime with 2 and 6.
12. In question 11, d would be 2.
    F: We should pick d=11 as well, as then we d have $ed = 121 = 1 \pmod{(3-1)(7-1)}$
13. For every file, there is a hash function that avoids collision.
    T: This says that for all files there exists a hash function that avoids collisions. Indeed if we re given the file in advance we can hash every entry in the file to a different bucket, as long as we have more buckets than entries.
14. There is a hash function such that any file of size n would have an expected number of collisions one.
    T: This is the definition of a universal hash function family. The example we saw in class is one such family of functions.
15. There is no hash function such that any file of size n would have an expected number of collisions less than one.
    T: Pigeonhole principle.
16. There is a hash function that avoids collisions for any file.
    F: This says that there exists some hash function such that for all files there is no collision. This is false as, if you give me the hash function in advance, I can always find a large enough appropriate file that maps things into the same bucket.

Problem 5 (15 points)

Suppose that in Dijkstra's algorithm you also want, if there are two or more shortest paths from S to a node, to find the one that has the fewest edges. Describe briefly a simple modification of Dijkstra's algorithm for doing this, and explain why it works.

When running Dijkstra's algorithm we usually have an array dist(u) to store the length of the current best shortest path to u for all $u \in V$ and an array prev(u) to store the parent of node u in the shortest path tree. To solve this problem, we also need to keep track of the length –i.e. the number of edges– of each shortest path from s to u. The way to do that is via some more bookkeeping: let len[u] be an array where we store the number of edges of the current shortest path from s to u. We only need to make sure that we consistently update this throughout our algorithm, ensuring that we always keep track of the shortest path with the least number of edges. Special care is needed in the case of ties, i.e. when we have two shortest paths with a possibly different number of edges each.

Here is the pseudocode:

```
for all u \in V
    dist(u) = \infty
    len(u) = \infty
    prev(u) = nil
dist(s) = 0
len(s) = 0

H = makequeue(V)
while H is not empty:
   u = deletemin(H)
   for all edges (u,v) \in E
       dst = dist(u) + l(u,v)
       if (dist(v) > dst OR (dist(v) == dst AND len(v) > len(u)+1))
            dist(v) = dist(u) + l(u,v)
            len(v) = len(u)+1
            prev(v) = u
```