CS180 midterm

[redacted]

TOTAL POINTS

71 / 100

QUESTION 1

1 Problem 1 **15 / 15**

✓ **- 0 pts Correct**

QUESTION 2

2 Problem 2 **11 / 15**

✓ **- 4 pts 1. the edges weights assigned in the input transformation step will not work**

QUESTION 3

3 Problem 3 **15 / 15**

✓ **- 0 pts Simply Correct**

QUESTION 4

4 Problem 4 **11 / 15**

✓ **- 2 pts not used heap/ another effective data structure**

✓ **- 2 pts Incorrect analysis of runtime complexity**

QUESTION 5

5 Problem 5 **7 / 15**

- **+ 12 pts** Correct optimal algorithm
- **+ 3 pts** Correct and optimal run time
- **+ 9 pts** Algorithm correctly uses sorting

✓ **+ 7 pts Partial credit**

- **+ 0 pts** Incorrect/Incomplete

QUESTION 6

Problem 6 25 pts

6.1 6.a **12 / 15**

✓ **- 3 pts Does not prove correctness**

6.2 6.b **0 / 10**

✓ **- 5 pts Says always optimal**

✓ **- 5 pts Does not provide counter example**

CS 180: Introduction to Algorithms and Complexity

Midterm Exam

May 6, 2019

Name	[redacted]
UID	[redacted]

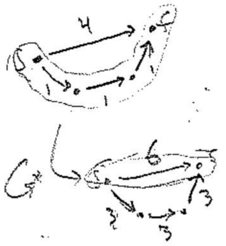
- Print your name, UID in the boxes above, and print your name at the top of every page.
- Exams will be scanned and graded in Gradescope. Use Dark pen or pencil. Handwriting should be clear and legible.
- The exam is a closed book exam, and no electronics of any kind.
- The exam is for 1 hour and 50 minutes during normal lecture hours from 12 noon to 1:50pm.
- Your answers are supposed to be in a simple and understandable manner. Sloppy answers and no justifications of your answers will get fewer points.

Name: [redacted] **UID:** [redacted] **CS180 Midterm Exam**

17

1. As you know from class, given a graph G with positive integer weights, a start node s, and a finish node f, you can find the shortest path S from s to f by running Dijkstra's algorithm. Let's assume that it so happens that this shortest path S is unique in G.

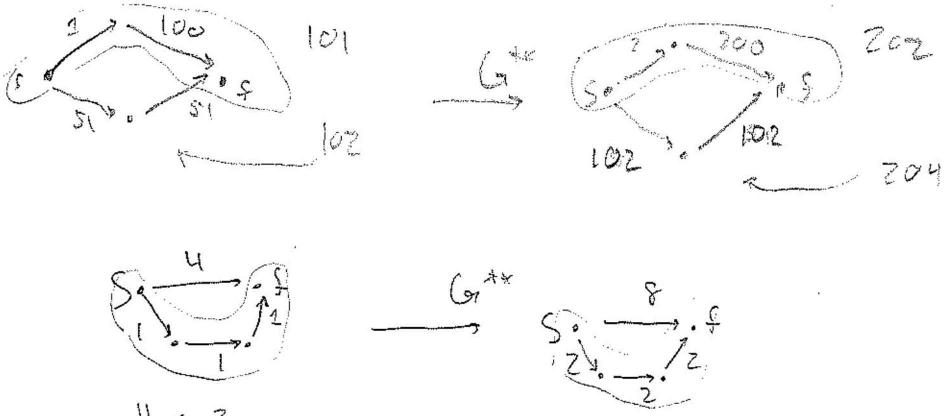
   - Does this shortest path S from s to f change if you increase every edge by 2 in the modified graph G' (i.e. you add weight of 2 to each edge in G to get G'.)? Explain your answer.[**7 pts**]
   - Does this shortest path S from s to f change if you multiply every edge by 2? in G''? (i.e., you multiply each edge by 2 in G to get G''.) Explain your answer. [**8 pts**]



- Yes, it can. Now a path with less edges, lets say n, will only increase by $2n$, where a path with more edges, lets say m, will increase by $2m$. We have $2n < 2m$ since $n < m$. ∴ this difference of $2m - 2n$ may make the difference in the algorithm (as shown to the left where $n = 1$, $m = 3$ and the shortest path switches.)

?

- No. the comparison would still hold true. When the comparison had a difference of 1, it would now have a difference of 2, and the difference would only increase as the edges/path weight increases.



4 vs 3

diff = 1

8 vs 6

diff = 2

diff doubled but still same result.

Name: [redacted] **UID:** [redacted] **CS180 Midterm Exam**

34

2. Let G be an undirected graph with non-negative integer weighted edges. A heavy Hamiltonian cycle is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-Complete. **[15 pts]**

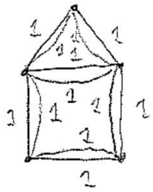
- This Problem (HHC) $\in NP$ b/c to verify you can traverse it to make sure it is a cycle and touches all nodes. As you do this, you can sum up the cycle's weight. Then, sum up the total weight of all edges in G. Then verify it is valid all in $O(n+m) = O(n^2)$. However, you cannot find a HCC in polynomial time $\therefore$ HCC $\in NP$.

- We know Hamiltonian Cycle (HC) $\in$ NP-Complete, so all we need to show is $HC \leq_p HHC$:

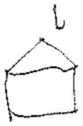
Assuming we have a polynomial solution to HHC, we could use it to solve HC with the following algorithm.

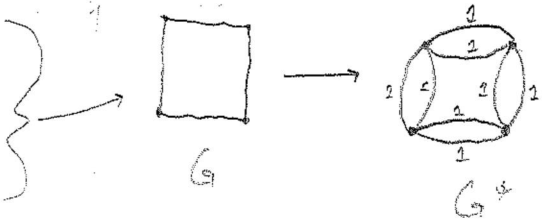
```
HC (G):                              |V|=n , |E|=m
O(m)      ~~for~~ each (u,v) ∈ G.E
              add a copy of (u,v) into G.E
O(m)      for each e ∈ G.E
              set weight to 1

O(N^2)    return HCC(G)
```



total = 12
half = 6
↓
only HHC is one of each edge pair that form HC of original G
↓

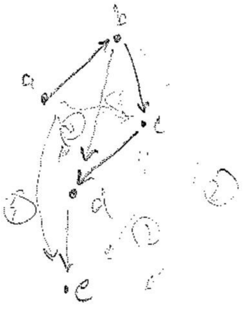




Pf: if our new version of G, G*, has a heavy HC that means each node was still only visited once and the weight was exactly half the total weight. It is impossible for G* to have a HHC w/ weight greater than half b/c for each pair of equal edges, only 1 can be included in the HHC. So, if G* has a HHC ⟹ G has HC and ~~[illegible]~~
if G has a HC ⟹ G* has HHC so,
$HHC(G^*) \Longleftrightarrow HC(G)$. This shows my algorithm using the reduction is correct. Also shown above is that my algorithm runs in polynomial time, $\therefore$
$HC \leq_p HHC$, <u>THERFORE</u> HHC $\in$ NP-Complete

34
17
51

3. Given a directed acyclic graph $G = (V, E)$, explain how to find the maximum number of directed edges that can be added to G so that the modified graph still remains acyclic. Give an algorithm to find out this number, show its running time and prove correctness of your algorithm. **[15 pts]**



We were taught in class that a DAG can be represented as a topological ordering. So First we will have this ordering, then see how many edges can be added from there, to keep orderi

$|V|=n$, $|E|=m$

Make Topo

```
Max_Num_Add_Edges (G)
    Topo = [ ]; Temp.V = G.V; Temp.E = G.E;
    for v ∈ Temp w/ no incoming edges {
        remove v from Temp.V
        remove v's edges from Temp.E
        Topo.append v
    }
    count = 0; inGE[ ][ ]; ~~O(N²)~~ ] O(N²)
    for i = 0, 1, ... Topo.length - 2 { //don't include last element ] O(N²)
        for j = i+1, i+2, ..., Topo.length - 1 {
            if inGE[i][j] == false {: ] O(1)
                count++
            }
        }
    }

    return count;
```

Complexity

You can also keep track of edge counts of each $v \in$ Temp.V by hash value so finding $v \in$ Temp w/ no incoming edges is O(which would make this total process a O(N) operation

for each N^2 edges you need to check if its in G.E, naively this could take O(m) each time but if you initially set up a 2D array for all N^2 edges & set their values to true if $\in$ G.E, it would be $O(N^2)$ outside the for loops which would not add to overall complexity and make this step O(1) inside the for loop.

Conclusion: The overall complexity is $O(N^2)$ due to some datastructure tricks explained above.

$\binom{5}{2} = \binom{\#\text{nodes}}{2}$

↓

All possible including those in E

2D Array of whether an edge is in G.E can be set up in $O(N^2)$

PF Correctness

Given a DAG w/ n vertices, it creates a topological ordering of these n items. The maximum number of edges possible is then calculated by $\binom{n}{2}$ b/c for each pair of vertices (v_i, v_j) there can exist an edge (v_i, v_j) as long as v_i precedes v_j in ordering. My algorithm goes through each of these $\binom{n}{2}$ / N^2 pairs and increments the counter by 1 if the edge of the pair is <u>not</u> in G.E ∴ it could be added & still preserve the DAG's nature. If G contains m edges, my algorithm will return $\underline{\binom{n}{2} - m}$.

Name: [redacted] **UID:** [redacted] **CS180 Midterm Exam**

51
+17
66

1h 8

4. You are given n cables of different lengths, find how to connect these cables into one cable. You can connect only two cables at a time, and the cost to connect two cables into one cable is equal to sum of their lengths. Show a poly-time algorithm to connect all cables with minimum total connection cost. Prove correctness (of finding minimum cost solution) of your algorithm and analyze the running time of your algorithm. **[15 pts]**

1 1 2 3
2 2 3
4 3
7

cost 1+1 2
1+1+2 4
1+1+2+3 7
13

You essentially add length of first 2 cables connected (n-1) times since their length will contribute to the cost every subsequent connection. The next one added will contribute (n-2) times, and so forth until the last one only contributes once. From this explanation it seems straight forward that one should add the smallest cables together first to minimize the subsequent contributions later on and wait to only count the largest cable value once at the end

Algorithm:

```
Sort list of n cables by increasing length        O(NlogN)
While list has >1 item {                          O(N)
    connect 2 smallest cables
    reorder new cable  ------------------------>  O(N) b/c given 1 item and
}                                                 a sorted list, it is O(N) to
return                                            traverse list and insert item
                                                  where it belongs
```

1 1 1 1
2 1 1
1 1 2
2 2
4

cost=8

Overall Complexity = $O(N^2)$

This is minimal cost B/c:

Pf: (Base Case (n=1)) this case is trivial w/ $L_{C_{k-1}}$

Assuming minimal cost up to C_{k-1}, my sol S and optimal sol S^* are the same, but then S^* chooses cable C_j w/ length L_{C_j} and S chooses cable C_i w/ length L_{C_i}. The cost of connecting these cables will be $L_{C_{k-1}}+L_{C_j}$ and $L_{C_{k-1}}+L_{C_i}$, and we know that $L_{C_{k-1}}+L_{C_j} \geq L_{C_{k-1}}+L_{C_i}$ b/c $L_{C_i} \leq L_{C_j}$ since that is how my algorithm cable choice is defined. Given that there are m cables left, that means there are m connections left. From above, we know $m(L_{C_{k-1}}+L_{C_j}) \geq m(L_{C_{k-1}}+L_{C_i})$ and let L_{m_j} be total length left in S^* and L_{m_i} be total length left in S. We know that $L_{m_i} \geq L_{m_j}$. We can see that for the fewer remaining m cables they will not be multiplied as much as the current $L_{C_{k_j}}$ and $L_{C_{k_i}}$ cables so the greater the L_{m_i} is to $L_{C_{k_i}}$ the better. If exchanging this inversion of C_j w/ C_i we can only improve or remaining equal to S^*'s total cost

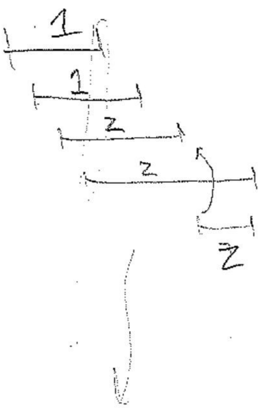
∴ my solution is optimal.

Name: [redacted] **UID** [redacted] **CS180 Midterm Exam**

1h 25

5. Given arrival and departure times of n trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits. A platform can simultaneously service not more than two trains at a time. Give analysis of your algorithm run time. **[15 pts]**

Similar to Classroom scheduling except that each room AKA train station can hold 2 trains at a time.



~~2~~ ~~2~~ ~~3~~ 2

depth = $\left\lceil \frac{\text{max \# overlap}}{2} \right\rceil$

depth = $\left\lceil \frac{4}{2} \right\rceil = 2$

Algorithm

```
TS = {}          //set of used TSs
TSC[]            //count of trains in each TS in TS,
Ts = sorted list of trains by increasing arrival time   O(NlogN)
count=0;
for each t ∈ Ts {
    found=false
    for each ts ∈ TSs {
        if TSC[ts] < 2      //if there a station open, use it
            TSC[ts]++;
            found = true;
            break
        }
    }
    if (found)
        continue
    add new train station nts to TS       //if no station available, add new station w/ 1 space left
    TSC[nts] = 1;
    count ++;
}
return count;
```

Also add check for when an older station becomes available, and if it does, decrement TSC & that will handle it all.

Run Time: to sort, my algorithm takes $O(N\log N)$, then it iterates over all N trains which is $O(N)$. Inside this loop it iterates over all train stations which in a worst case could be $\left\lceil \frac{n-1}{2} \right\rceil$ which is also proportional to n yielding $O(N)$. For all other data accesses, they are done in $O(1)$ b/c they are stored in arrays. This means that the total run time of my algorithm is $O(N^2)$.

7.5
15/2

1h 42

6. Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution. (Recall that quarters are 25 cents; dimes are 10 cents; nickels are 5 cents, and pennies are 1 cent). Prove that your algorithm is correct. **[15 pts]**

```
Get-Change(n):
  while n != 0 {
    if n ≥ 25:
      give quarter()
      n -= 25
    else if n ≥ 10:
      give dime()
      n -= 10
    else if n ≥ 5:
      give nickel()
      n -= 5
    else
      give_penny()
      n -= 1
  }
```

// Greedy Nature: give biggest possible coin without giving too much back

PS: Stays ahead method

given an n value = 25, 10, 5, 1, my solution will return just 1 coin which is base case optimal.

Now, assume optimal sol S* and my sol S are similar up to choosing the k-1th coin, but S* choose C_j and S chooses C_i s.t $C_j \leq C_i$ in value b/c by my definition C_i is maximal option value-wise. This means that if we exchanged C_j w/ C_i in S* to create S**, the amount left in S** would be $S^{**}_{left} \leq S^{*}_{left}$. Since $S^{**}_{left} \leq S^{*}_{left}$ that means the remaing coins of S** is as good or better than the coins remaining in S*. By this ~~dogu~~ my solution at ~~every~~ choice stays ahead or is just as good as the optimal solution so ∴ my solution is Optimal.

(b) Is your greedy algorithm always optimal for any set of coin denominations (i.e if you get to pick which coin values are in circulation)? If yes, provide a proof. If no, give a counter-example for showing that your greedy algorithm is not optimal for a set of coin denominations. Your proof or counter-example should include a penny so that there is a solution for every value of n. **[10 pts]**

<u>Yes.</u> The only change to the algorithm would be that the values 25, 10, 5 could not be hard-coded in on would except use some sorted array to properly find the decreasing order of each denomination. My proof from the last page never specified any specific coin values during its logic, so that proof can also be used here. The change would be that the values C_j, C_i would involve numerical values potentially not 25, 10, 5. This change, however, does <u>not</u> change the optimality of my solution b/c I will know that $\underbrace{C_i \geq C_j}$ b/c $C_i \geq C_k \; \forall k$ and $C_j \in \{C_k \mid \forall k\}$. And this comparison is what provides the mathematical proof and logic behind my solutions optimality b/c it is what allows my solution "stay ahead" of the optimal solution which proves it optimality.