"CS/ECE 374 A": Algorithms & Models of Computation, Spring 2025
Midterm 2 — April 14, 2025

Name:	Heng An Cheng
NetID:	hacheng2

- Please *clearly PRINT* your name and your NetID in the boxes above.
- This is a closed-book but you are allowed a 1 page (2 sides) hand written cheat sheet that you have to submit along with your exam. If you brought anything except your writing implements, put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear. The exam has 6 problems, each worth 10 points.
- **You have 150 minutes (2.5 hours) for the exam.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.**
- **Write everything inside the box around each page.** Anything written outside the box may be cut off by the scanner.
- **Proofs are required only if we specifically ask for them.** You may state and use (without proof or justification) any results proved in class or in the problem sets unless we explicitly ask you for one.
- You can do hard things!
- **Do not cheat.** You know the student code and all that jazz. Grades do matter, but not as much as you may think, and your values are more important.

1 Sums and Recurrences

$n^{\frac{1}{4}} T(n^{\frac{1}{4}}) = n^{\frac{1}{16}} T(n^{\frac{1}{16}}) + n^{\frac{1}{4}}$

$n^{\frac{1}{8}} \cdot n^{\frac{1}{4}}$

(a) Consider the recurrence

$$T(n) = n^{1/4} T\left(n^{1/4}\right) + n \quad n \geq 16, \qquad T(n) = 1 \quad 1 \leq n < 16$$

Give asymptotically tight bounds for the following, assuming the root of the tree is level one. No work is required; write your final answer in the box.

– (1 pt) Number of children at the second level:

$O(n^{\frac{1}{4}})$

– (1 pt) Work at the second level:

$O(n^{\frac{1}{2}})$

– (1 pt) Depth of the recurrence:

$O(\log(\log n))$

– (1 pt) Value of the recurrence:

$O(n)$

$T(n)$ $\quad n$

$n^{\frac{1}{4}} T(n^{\frac{1}{4}})$ $\quad n^{\frac{1}{4}} \times n^{\frac{1}{4}}$

$n^{\frac{1}{2k}} > 16$

$(n^2)^{-k}$ $\quad -k \log n^2 > 16$

$n + n^{\frac{1}{2}} + n^{\frac{1}{8}} +$

(b) Consider the recurrences below and give asymptotically tight bounds for each $T(n)$. No work is required; write your final answer in the box.

– (2 pts) $T(n) = 4T(n/2) + n$ for $n \geq 2$ and $T(1) = 1$

$O(n^2)$

– (2 pts) $T(n) = 4T(n/2) + n^2$ for $n \geq 2$ and $T(1) = 1$

$O(n^2 \log n)$

– (2 pts) $T(n) = 4T(n/2) + n^3$ for $n \geq 2$ and $T(1) = 1$

$O(n^3)$

n^2 $\quad n^2$ $\quad 2^{2k} \cdot \frac{}{2^{2k}}$

$4 \cdot \left(\frac{n}{2}\right)^2$ $\quad 4^k \cdot \left(\frac{n}{2^k}\right)^2$ $\quad n^2$

$16 \cdot \left(\frac{n}{4}\right)^2$

$4^3 \left(\frac{n}{8}\right)^2$

$4^k \cdot \left(\frac{n}{2^k}\right)^3 = 2^{2k-3k} \cdot n^3$

$= 2^{-k} \cdot n^3$

2 Recursion/Divide and Conquer/Sorting/Selection

A is an array of n_1 numbers that is sorted in *ascending* order. B is an array of n_2 numbers sorted in *descending* order. You wished to create a sorted array by merging them but by mistake you concatenated A with B to create an array C with $n = n_1 + n_2$ numbers (assume for simplicity that all the numbers are distinct). You do not have the original arrays anymore nor do you know n_1 and n_2. For example if $A = [1, 2, 6, 7]$ and $B = [8, 4, 3]$ then $C = [1, 2, 6, 7, 8, 4, 3]$. Describe an algorithm, as fast as possible, that given C, its size n, and a number x checks whether x is in C or not.

Iterate array C, use linear search to check whether X is in C or not.

Time = $O(n)$ #

12 23 34 45

$\boxed{1\ 2}\boxed{3\ 4\ 5}$

3 Splitting Strings

Let Σ be a finite alphabet and $L \subseteq \Sigma^*$ be a language. You have access to a routine IsStringinL(x) that on input $x \in \Sigma^*$ returns whether $x \in L$ or not. Given $w \in \Sigma^*$ recall that $w \in L^*$ if and only if $w = w_1 w_2 \ldots w_h$ for some $h \geq 1$ such that each $w_i \in L$; in this case we call $w_1 w_2 \ldots w_h$ a valid L-split. In many applications we are interested in a L-valid split with some additional properties. Given w and a split of w into $w_1, w_2, \ldots, w_h$ we define the cost($w_1, w_2, \ldots, w_h$) to be $\sum_{i=1}^{h} |w_i|^2$.

Describe an algorithm that given $w \in \Sigma^*$ outputs the minimum cost of any L-valid split. Your algorithm should output ∞ if there is no L-valid split of w. You can assume that isStringinL(x) takes $O(1)$ time in your analysis.

Define dp[i] be the minimum cost of any L-vaid spilt from W[1...i]. And we want to find dp[n] #

$$dp[i] = \begin{cases} dp[0] = 0 \\ \min\{_{0 \le k < i} dp[k] + (W[k+1 \cdots i])^2\} & \text{if } W[(k+1) \cdots i] \in L \\ \infty, & \text{otherwise} \end{cases}$$

Code(W).

```
for i=1 to n
    dp[i]=∞
    for k=0 to i-1
        if W[(k+1)···i] ∈ L
            dp[i] = min( dp[k]+(W[k+1···i])^2 )
return dp[n] #
```

⇒ Time Complexity $O(n^2)$ #

4 Collecting Rewards

This is a variant of a problem from Homework 9.

Let $G = (V, E)$ be a directed graph in which each edge $e \in E$ has a non-negative reward $p(e)$ that can be collected by traversing it. Describe an algorithm that given G, a starting vertex $s \in V$ and an integer $k \geq 0$ computes the maximum reward that a walk starting at s can collect if it is required to only collect reward from at most k edges. Note the the reward on an edge e is counted only the first time it is traversed in the walk since it is gone after it is picked up.

(a) (2 pts) How would you solve the problem if G is strongly connected?

if G is strongly connected that means you traverse all edse you want, collect the top k max reward will be the answer. We just sort the reward p(e) among all edges and sum up the top k max reward p(e), it will be the answer.

Time complexity : sorting edges $\Rightarrow O(E \log E)$

(b) (8 pts) How would you solve the problem if G is a DAG?

Define dp(i) be the maximum reward that a walk Starting at s can collect from exactly i edges. There with be two options walking through edge e, collect reward p(e) or not.

$$dp(i) = \begin{cases} 0, & i=0 \\ \max(dp(i), dp(i-1)+p(e)), & i \geq 1 \end{cases}$$

O(K) 1. Set dp(i)=0 from i=0 to K.

O(V+E) 2. Using BFS(s) and Keep updating dp(i) defined above from i=1 to K.

O(K) 3. After BFS all reachable vertex, find max(dp(i), i=1 to K).

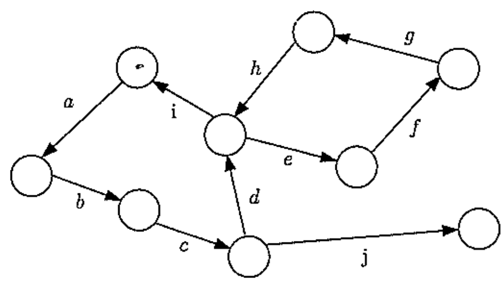
(Exercise for *after* the exam: combine these two parts to get an algorithm that works on an arbitrary directed graph.)

Time complexity $O(V+E+K) \approx O(V+E)$ #

5 Graphs

Let $G = (V, E)$ be a directed graph and let $e_1 = (a, b)$ and $e_2 = (x, y)$ be two distinct edges. We wish to know if there exists a *path* that uses both e_1 and e_2, with e_1 appearing earlier in the path than e_2. Note that a path does not allow repetitions of vertices or edges while a walk allows both.

(a) Consider the following graph:



– (1 pt) Specify two edges e_1 and e_2 such that there is a path P in the graph with e_1 before e_2 in P.

$e_1 =$ a

$e_2 =$ b

– (1 pt) Specify two edges e_1 and e_2 such that there is a *walk* W in the graph with e_1 before e_2 in W but there is *no* such *path* in the graph.

$e_1 =$ i

$e_2 =$ f

(Continued on next page)

(b) (4 pts) Describe an efficient algorithm that given $G = (V, E)$ and two distinct edges $e_1, e_2 \in E$ checks if there is a *walk* in G with e_1 before e_2.

Assume $e_1 = (a, b)$, $e_2 = (x, y)$

Start from b and use BFS to check if it can reach x, if reach(x) = True, then there is a walk in G with e_1 before e_2.

Proof: if x is reachable from b, we can start at a and walk e_1 to b, and b can reach x, then walk e_2.

⇒ Time: BFS ⇒ $O(V+E)$ #

(c) (4 pts) Describe an efficient algorithm that given $G = (V, E)$ and two distinct edges $e_1, e_2 \in E$ checks if there is a *path* in G with e_1 before e_2.

Use almose the same algo from (b), but when starting BFS from b, we will mark both a, b as visited. During BFS, if we reach a node that is visited before we reach x, then directly return False.

Otherwise, if b can reach x and ($x \neq a$ or b) and ($y \neq a$ or b) then there is a path in G with e_1 before e_2.

Time: BFS ⇒ $O(V+E)$ #

6 Shortest Paths with a Twist

Let $G = (V, E)$ be a directed graph where each edge e has a non-negative length $\ell(e) > 0$. Each vertex $v \in V$ also has a non-negative value $a(v)$ specified in an array A. For simplicity, we will assume that the $a(v)$ values are distinct. You want to take a walk on this graph of total length at most L, starting and ending at a specified vertex s.

(a) (3 pts) Given G, A, s, L and a specific vertex t, describe an efficient algorithm that checks if there is a walk of length at most L that starts and finishes at s and visits t.

We want to find a path from s to t and t to s. with total length at most L.

∵ G has non-negative length

∴ Use Dijkstra's algorithm twice to find shortest path from S to t and t to S.

if mindist(s,t) + mindist(t,s) $\leq$ L $\Rightarrow$ there is a walk.

Time : Dijkstra's $\Rightarrow O((V+E)\log V)$ #

(b) (3 pts) Given G, A, s, L describe an efficient algorithm to compute the highest-value of a vertex that any walk of length at most L that starts and finishes at s can visit.

First, we can decrease the number of possible vertices by building the meta-graph, $O(V+E)$.

(∵ We need to start and ends at S $\Rightarrow$ a cycle)

$O(V^3)$ Build ASSP with Floyd-warshall with $G' \in$ SCC of s.

$O(V \log V)$ { Sort the value of $a(v) \mid v \in$ SCC of s. in descending order

Find the first v that mindist(s,v) + mindist(v,t) $\leq$ L

$\Rightarrow$ output a(v)

(Continued on next page)

Time Complexity = $O(V^3)$

(c) (4 pts) Let α be the maximum value that you computed in the previous part. Describe an efficient algorithm to compute the second most valuable vertex your walk can reach subject to visiting the one with value α. Your walk must still start and end at s and have total length at most L.

Because in (b) we already use Folyd-Warshall to compute ASSP in SCC of S.

Code:

For u in descending order of a(u), u in SCC of S.

O(1) ⇐ if min[(dist(s, u) + dist(u, v) + dist(v, s)),

dist(s, v) + dist(v, u) + dist(u, s)] ≤ L {

return a(u).

}

⇒ Time Complexity: O(V) (just iterate all vertex in SCC of S.)

This page is for extra work.

This page is for extra work.