CS 4800 - Final - Review material

Old stuff

Big-O notation Though you won't be quizzed directly on Big-O notation, you should be able to apply it to analyzing algorithms, e.g. ones that you produce a problem solutions.

Recurrence relations As with Big-O.

Basic Arithmetic Most of you still remember the algorithms that you learned in elementary school. You should now have an idea of how they work on a computer and how and when they can be sped up.

Modular arithmetic Never forget this.

Binary search Never forget this.

Merge sort, Partition in quicksort, Insertion sort You really should be able to tell me about these ten years from now even if you don't remember all the details.

New stuff

You should know these algorithms and how to analyze their running times.

- **Chapter 3 - Decomposition of Graphs**
  - Adjacency lit and adjacency matrix representation
  - Depth-first search in undirected graphs
  - Depth-first search in directed graphs
    Topological sort
  - Strongly connected components
- **Chapter 4 - Paths in Graphs**
  - Distances
  - Breadth-first search
  - Lengths on edges (weighted graphs)
  - Dijkstras algorithm
    Priority queue implementations: array and binary heap
  - Heap sort
  - NO – Shortest paths in the presence of negative edges
  - Shortest paths in dags .

- **Chapter 5 - Greedy algorithms**
  - Minimum spanning trees
  - Huffman encoding
  - Horn formulas
  - Set cover
- **Chapter 6- Dynamic programming**
  - Shortest paths in dags, revisited
  - Longest increasing subsequences
  - Edit distance
  - Knapsack
- **Chapter7 NO - Linear programming** (NO– reductions)
  - An introduction to linear programming
  - The simplex algorithm

Graphs
A *undirected graph* $G = (V, E)$ where each *edge* $e \in E$ is 2-way and represented by a set $\{u, v\}$ with $u, v \in V$. We also write (u, v) when we are talking about following the edge from u to v.
A *graph* $G = (V, E)$ where each *edge* $e \in E$ is 1-way and represented by an ordered pair (u, v) with $u, v \in V$. The edge (u, v) goes from u to v.

A graph can be represented by an $n \times n$ *adjacency matrix* where $n = |V|$. The $(i,)$th entry is

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

or an *adjacency list* that consists of $|V|$ linked lists, one per vertex. The linked list for vertex$u$ holds the names of vertices to which u has an outgoing edge, i.e. vertices v for which $(u, v) \in E$.

Depth-first search
You should understand procedures `explore`(G, v) on page 84 and `dfs`(G) on page 85 using the `previsit` and `postvisit` on page 87.

You should be able to perform depth-first search in a directed or undirected graph, label the vertices with `pre` and `post` numbers and label the edges as *tree* or *back* in an undirected graph, *tree*, *forward*, *back*, or *cross* in a directed graph.

Tree edges are actually part of the DFS forest.

Forward edges lead from a node to a non-child descendant in the DFS tree.

Back edges lead to an ancestor in the DFS tree.

Cross edges lead to neither descendant nor ancestor; they therefore lead to a node that has already been completely explored (that is, already post-visited).

Strongly connected components - the algorithm

1. Run DFS on G^R.
2. Run DFS on G from vertex in 1 with highest POST number. Remove those vertices from G (or their component from the dag) and repeat starting with the remaining vertex highest POST number from step 1.

Breadth-first search

```
BFS(G, s)
for all u ∈ V
    dist(u) = ∞
dist(u) = 0
Q = [s] (queue containing just s)
while Q is not empty
    u = eject(Q)
    for all edges (u, v) ∈ E
        if dist(v) = ∞
            inject(Q, v)
            dist(v) = dist(u) + 1
```

Shortest Path

```
DIJKSTRA(G, s)
Initialize dist(s) = 0, other dist(·) values to ∞, R = { } (the "known region")
while R ≠ V:
    Pick the node v ∉ V with smallest dist(·)
    Add v to R
    for all edges (v, z) ∈ E:
        if dist(z) > dist(v) + l(v, z):
            dist(z) = dist(v) + l(v, z)
```

Insert : Add a new element to the set.

Decrease-key : Accommodate the decrease in key value of a particular element. (Notifies the queue that a certain key value has been decreased.)

Delete-min : Return the element with the smallest key, and remove it from the set.

Make-queue : Build a priority queue out of the given elements with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

Priority queue implementations Binary heap wins over array implementation as soon as $|E| < |V|^2/\log|V|$.

- Good for both sparse and dense graphs.
- sparse, $|E| = O(|V|)$, running time is $O(|V|\log|V|)$.
- dense, $|E| = O(|V|^2)$, running time is $O(|V|^2)$.
- intermetiate, $|E| = O(|V|^{1+\delta})$ running time is $O(||E|)$. -linear!

Fibonacci heap is best but hard to implement. Insert averages (*amortized cost*) $O(1)$.

MAX-HEAPIFY(A, i, n)
$l =$ LEFT(i)
$r =$ RIGHT(i)
if $l \leq n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest, n)$

Building a Heap

BUILD-MAX-HEAP(A, n)
for $i = \lfloor n/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i, n)

The heapsort algorithm

Given an input array, the heapsort algorithm acts as follows:

- Build a max-heap from the array.
- The root is the maximum element so swap it with the element in the last position in the array.
- Decrease the heap size by 1, and call MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat until only one node (the smallest element) remains, and therefore is in the correct place in the array.

Analysis
BUILD-MAX-HEAP: $O(n)$
for loop: $n - 1$ times

swap elements: $O(1)$
MAX-HEAPIFY: $O(n \lg n)$

Total time: $O(n \lg n)$.

Minimum spanning trees
Kruskal's algorithm

Repeatedly add the next lightest edge that doesnt produce a cycle.

KRUSKAL(G, w)
Input: A connected undirected graph $G = (V, E)$ with edge weights we
Output: A minimum spanning three defined by the edges X

for all $u \in V$:
    MAKESET(u)

$X = \{\}$
sort the edges E by weight
for all edges $\{u, v\} \in E$,in increasing order of weight:
    if FIND$(u) \neq$ FIND(v):
        add edge $\{u, v\}$ to X
        UNION(u, v)

Data structure for disjoint sets
Union by rank:
store a set is as a directed tree. Nodes of the tree are elements of the set, in no particular order, and each has parent pointers that eventually lead up to the root of the tree. This root element is a *representative*, or *name*, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

MAKESET(x)
$\pi(x) = x$
$\text{rank}(x) = 0$

FIND(x)
if $x \neq \pi(x)$: $\pi(x) = \text{find}(\pi(x))$
return $\pi(x)$

UNION(x, y)
$r_x = \text{find}(x)$
$r_y = \text{find}(y)$
if $r_x = r_y$ **return**
if $\text{rank}(r_x) > \text{rank}(r_y)$
    $\pi(r_y) = r_x$
else
    $\pi(r_x) = r_y$
    if $\text{rank}(r_x) = \text{rank}(r_y)$: $\text{rank}(r_y) = \text{rank}(r_y) + 1$

The *amortized cost* of a sequence of n union find operations starting from an empty data structure averages $O(1)$ down from $O(\log n)$.

Prims algorithm for MST
The intermediate set of edges X always forms a subtree, and S is chosen to be the set of this trees vertices.

On each iteration, the subtree defined by X grows by one edge, namely, the lightest edge between a vertex in S and a vertex outside S.

Huffman encoding
Huffman encoding is a *variable-length encoding* in which just one bit is used for the most frequently occurring symbol.

HUFFMAN(f)
Input: An array $f[1..n]$ of frequencies
Output: An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for $i = 1$ **to** n
    Insert(H, i)
for $k = n + 1$ **to** $2n - 1$
    $i =$ deletemin(H), $j =$ deletemin(H)
    create a node numbered k with children i, j
    $f[k] = f[i] + f[j]$
    insert(H, k)

This algorithm can be described in terms of priority queue operations and takes $O(nlogn)$ time if a binary heap is used.

Horn formulas Knowledge about variables is represented by two kinds of *clauses*:

1. *implication*: left side is an AND of 0 or more variables.

$$(z \wedge w) \Rightarrow u$$

$$u \vee \overline{z} \vee \overline{w}$$

2. *negative clause*: an OR of 0 or more negative literals.

$$(\overline{x} \vee \overline{u} \vee \overline{y})$$

Input: a Horn formula
Output: a satisfying assignment, if one exists

set all variables to `false`

while there is an implication that is not satisfied
    set the right-hand variable of the implication to `true`

if all pure negative clauses are satisfied:
    return the assignment
else
    return formula is not satisfiable

Set Cover - greedy algorithm

Input: A set of elements B; sets $S_1, \ldots, S_m \subseteq B$
Output: A selection of the S_i whose union is B.
Cost: Number of sets picked.

Repeat until all elements of B are covered:
    Pick the set S_i with the largest number of uncovered elements.

Claim Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ sets.

Dynamic Programming

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved.

Shortest path in dags

initialize all dist$(\cdot)$ values to ∞
dist$(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
dist$(v) = \min_{(u,v)\in E}\{dist(u) + l(u,v)\}$

Edit distance

for $i = 0, 1, 2, \ldots, m$
    $E(i, 0) = i$
for $j = 0, 1, 2, \ldots, n$
    $E(0, j) = j$
for $i = 1, 2, \ldots, m$
    for $j = 1, 2, \ldots, n$
        $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \text{diff}(i, j) + E(i-1, j-1)\}$
return $E(m, n)$

Longest common subsequence

```
LCS-Length(X, Y, m, n)
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to nm
    c[i, 0] = 0
for j = 1 to n
    c[0, j] = 0
for i = 1 to nm
    for j = 1 to n
        if x_i = y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = " ↖ "
        else if c[i − 1, j] ≥ c[i, j − 1]
                c[i, j] = c[i − 1, j]
                b[i, j] = " ↑ "
            else c[i, j] = c[i, j − 1]
                b[i, j] = " ← "
return c and b

Print-LCS(b, X, i, j)
if i = 0 or j = 0
    return
if b[i, j] = " ↖ "
    Print-LCS(b, X, i − 1, j − 1)
    print x
elseif b[i, j] == " ↑ "
    Print-LCS(b, X, i − 1, j)
else Print-LCS(b, X, i, j − 1)
```

Knapsack

```
Fractional-Knapsack(v, w, W)
load = 0
i = 1
while load < W and i ≤ n
    if w_i ≤ W − load
        take all of item i
    else take (W − load)/w_i of item i
    add what was taken to load
    i = i + 1
```

Time: $O(n \lg n)$ to sort, $O(n)$ thereafter.

Knapsack with repetition

Define $K(w)$ = maximum value achievable with a knapsack of capacity w.

$K(0) = 0$
for $w = 1$ **to** W:
    $K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
return $K(W)$

This fills a one-dimensional table of length $W + 1$ from left to right.
Each entry can take $O(n)$ to compute so the running time is $O(nW)$.
This is EXPONENTIAL!

Knapsack without repetition Add a second parameter, $0 \leq j \leq n$:
$K(w, j)$ = maximum value achievable using a knapsack of capacity w and items $1, \ldots, j$.
We seek $(K(W, n)$.

Initialize all $K(0, j) = 0$ and $K(w, 0) = 0$
for $w = t$ **to** n
    for $w = 1$ **to** W:
        if $w_j > w$
            $K(w, j) = K(w, j - 1)$
        else $K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$
return $K(W, n)$