Final Exam Review

CMSC 451, Summer 2009

Final Exam Topics

Greedy Algorithms (Interval Scheduling, Shortest Path, Minimum Spanning Tree, Union-Find, Clustering, and Huffman Coding)

Divide and Conquer (Merge Sort, Inversion Counting, Closest Pair of Points, Multiplication, and Matrix Multiplication)

Dynamic Programming (Weighted Interval Scheduling, Knapsack Problem, and Sequence Alignment)

Network Flow (Max Flow/Min Cut, Capacity Scaling, Bipartite Matching, Extensions, and Airline Scheduling)

Intractability (Poly-time reductions, NP, NP-Complete)

Approximation Algorithms (Load Balancing, k-Center Problem, Pricing Method, Integer and Linear Programming)

Randomized Algorithms (Global Minimum Cut and Closest Pair of Points)

Registrar’s Problem
LPs

- Given a flow network \( G(V,E) \) with the set of lower bounds \( l(v,w) \) and upper bounds \( u(v,w) \) for each edge, and demands \( d(v) \) for each vertex, give an LP (with no objective function) to create a valid circulation.

Google/Microsoft Interview Questions

- Given a number, describe an algorithm to find the next larger number that is prime.
Google/Microsoft Interview Questions

• You are given a list of numbers. When you reach the end of the list you will come back to the beginning of the list (a circular list). Write the most efficient algorithm to find the minimum # in this list. Find any given # in the list. The numbers in the list are always increasing but you don’t know where the circular list begins, ie: 38, 40, 55, 89, 6, 13, 20, 23, 36.

Divide and Conquer

• Provide a divide-and-conquer algorithm for determining the largest and second largest values in a given unordered set of numbers. Provide a recurrence equation expressing the time complexity of the algorithm, and derive it’s solution.
NP-Complete Reductions

- Dominating set: Given a graph $G=(V,E)$ and integer $0<k\leq|V|$, is there a subset $D$ of $V$ such that $|D|\leq k$ and every vertex in $V\setminus D$ is joined to at least one member of $D$ by an edge in $E$?
- Set cover: Given a universe $U$, subsets $S_1, S_2, \ldots, S_m$, and integer $k$, are there $\leq k$ subsets $S_i$ such that their union is $U$?
- Reduce dominating set to set cover

NP-Complete Reductions

- Assume that 3-Color is NP-Complete. Prove that 4-Color is NP-Complete
Google/Microsoft Interview Questions

- Assume you have an array that contains a number of strings (perhaps char * a[100]). Each string is a word from the dictionary. Your task, described in high-level terms, is to devise a way to determine and display all of the anagrams within the array (two words are anagrams if they contain the same characters; for example, tales and slate are anagrams.)

Network Flow

- Let M be an n x n matrix with each entry equal to either 0 or 1. Let mij denote the entry in row i and column j. A diagonal entry is one of the form mii for some i. Swapping rows i and j of the matrix M denotes the following action: we swap the values mik and mjk for k=1,2,...,n. Swapping two columns is defined analogously.
- M is rearrangeable if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of M are 1.
- Is M rearrangeable? Poly-time algorithm: yes/no
Selected Algorithms

Interval Scheduling: Analysis

- Analysis: $O(n \log n)$ time
  - Sorting $O(n \log n)$
  - Check compatibility in $O(1)$ by remembering last finish time

```
Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.
A ← φ
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
} 
return A
```
Dijkstra’s Algorithm: Time Analysis

Input: edge-weighted graph $G = (V, E, \ell)$, source $s$, sink $t$
Let $S$ be the set of explored nodes
For each $u$ in $S$, store a distance $d(u)$
Initialize $S=\{s\}$ and $d(s)=0$

While $S \neq V$
    Select a node $v$ not in $S$ with at least one edge from $S$ such that
    $$\pi(v) = \min_{e=(u,v) \in E} \{ d(u) \} + \ell_e \text{ is minimized}$$
    Add $v$ to $S$ and set $d(v) = \pi(v)$
EndWhile

Kruskal’s Algorithm: Time Analysis

Kruskal(G, c) {
    Sort edge weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
    $T \leftarrow \phi$
    foreach $(u \in V)$
        make a set containing singleton $u$
    for $i = 1$ to $m$
        $(u,v) = e_i$
        if $(u$ and $v$ are in different sets) {
            $T \leftarrow T \cup \{e_i\}$
            merge the sets containing $u$ and $v$
        }
    return $T$
}

Use the union-find data structure.
- Sort: $O(m \log n)$ time (since $m=O(n^2)$, $\log m = O(\log n^2) = O(2 \log n)$)
- For all nodes: $O(n)$ total time for MakeUnionFind
- For each edge: $O(\log n)$ time for Find, $O(1)$ time for Union
- Total: $O(m \log n)$ time
Optimal Prefix Codes: Huffman Encoding

Huffman(S) {
    if |S|=2 {
        return tree with root and 2 leaves
    } else {
        let y and z be lowest-frequency letters in S
        S' = S
        remove y and z from S'
        insert new letter \( \omega \) in S' with \( f_\omega = f_y + f_z \)
        T' = Huffman(S')
        T = add two children y and z to leaf \( \omega \) from T'
        return T
    }
}

Build a tree for:
Alphabet: \{a,e,k,l,u\}
Frequencies: \( f_a = 0.32 \), \( f_e = 0.25 \), \( f_k = 0.20 \), \( f_l = 0.18 \), \( f_u = 0.05 \)

Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L
    Divide the list into two halves A and B
    \((r_A, A) \leftarrow \text{Sort-and-Count}(A)\)
    \((r_B, B) \leftarrow \text{Sort-and-Count}(B)\)
    \((r, L) \leftarrow \text{Merge-and-Count}(A, B)\)
    return \( r = r_A + r_B + r \) and the sorted list L
}
Closest Pair Algorithm

```
Closest-Pair(p₁, ..., pₙ) {
    Compute separation line L such that half the points
    are on one side and half on the other side.
    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)
    Delete all points further than δ from separation
    line L
    Sort remaining points by y-coordinate.
    Scan points in y-order and compare distance between
    each point and next 11 neighbors. If any of these
    distances is less than δ, update δ.
    return δ.
}
```

Fast Matrix Multiplication

- **Key idea.** multiply 2-by-2 blocks with only 7 multiplications.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

- \[C_{11} = P₃ + P₄ - P₅ + P₆\]
- \[C_{12} = P₁ + P₂\]
- \[C_{21} = P₃ + P₄\]
- \[C_{22} = P₃ + P₁ - P₅ - P₇\]

- \[P₁ = A₁₁ \times (B₁₂ - B₂₂)\]
- \[P₂ = (A₁₁ + A₁₂) \times B₂₂\]
- \[P₃ = (A₂₁ + A₂₂) \times B₁₁\]
- \[P₄ = A₂₂ \times (B₂₁ - B₁₁)\]
- \[P₅ = (A₁₁ + A₂₂) \times (B₁₁ + B₂₂)\]
- \[P₆ = (A₁₂ - A₂₂) \times (B₂₁ + B₂₂)\]
- \[P₇ = (A₁₁ - A₂₁) \times (B₁₁ + B₂₁)\]

- 7 multiplications.
- \[18 = 8 + 10\] additions and subtractions.
Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

**Input:** $n$, $s_1, s_2, \ldots, s_n$, $f_1, f_2, \ldots, f_n$, $v_1, v_2, \ldots, v_n$  

**Sort** jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$. 

**Compute** $p(1), p(2), \ldots, p(n)$ 

```plaintext
for j = 1 to n
    M[j] = empty
    M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max($v_j + M-Compute-Opt(p(j))$, $M-Compute-Opt(j-1)$)
    return M[j]
}
```

Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an $n$-by-$W$ array.

**Input:** $n$, $W$, $w_1, w_2, \ldots, w_n$, $v_1, v_2, \ldots, v_n$ 

for $w = 0$ to $W$
    $M[0, w] = 0$

for $i = 1$ to $n$
    for $w = 1$ to $W$
        if ($w_i > w$)
            $M[i, w] = M[i-1, w]$
        else
            $M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}$

return $M[n, W]$
Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
  for i = 0 to m
    M[0, i] = iδ
  for j = 0 to n
    M[j, 0] = jδ
  for i = 1 to m
    for j = 1 to n
      M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                     δ + M[i-1, j],
                     δ + M[i, j-1])
  return M[m, n]
}
```

Augmenting Path Algorithm

```
Augment(f, c, P) {
  b ← bottleneck(P) ←
  foreach e ∈ P {
    if (e ∈ E) f(e) ← f(e) + b
    else f(e^R) ← f(e^R) - b
  }
  return f
}
```

```
Ford-Fulkerson(G, s, t, c) {
  foreach e ∈ E  f(e) ← 0
  G_f ← residual graph
  while (there exists augmenting path P) {
    f ← Augment(f, c, P)
    update G_f
  }
  return f
}
```
Capacity Scaling

Scaling-Max-Flow(G, s, t, c) {
  foreach e ∈ E  f(e) ← 0
  Δ ← smallest power of 2 greater than or equal to C
  GΔ ← residual graph
  while (Δ ≥ 1) {
    GΔ(Δ) ← Δ-residual graph
    while (there exists augmenting path P in GΔ(Δ)) {
      f ← augment(f, c, P)
      update GΔ(Δ)
    }
    Δ ← Δ / 2
  }
  return f
}

Load Balancing: List Scheduling

List-scheduling algorithm.
- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

List-Scheduling(m, n, t₁, t₂, ..., tₙ) {
  for i = 1 to m {
    Lᵢ ← 0  --- load on machine i
    J(i) ← φ  --- jobs assigned to machine i
  }
  for j = 1 to n {
    i = argminᵢ Lᵢ  --- machine i has smallest load
    J(i) ← J(i) U {j}  --- assign job j to machine i
    Lᵢ ← Lᵢ + tⱼ  --- update load of machine i
  }
  return J(1), ..., J(m)
}

Implementation. \(O(n \log m)\) using a priority queue.
Load Balancing: LPT Rule

Longest processing time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

\[
\text{LPT-List-Scheduling}(m, n, t_1, t_2, \ldots, t_n) \{ \\
    \text{Sort jobs so that } t_1 \geq t_2 \geq \ldots \geq t_n \\
    \text{for } i = 1 \text{ to } m \{ \\
        L_i \leftarrow 0 \quad \text{load on machine } i \\
        J(i) \leftarrow \emptyset \quad \text{jobs assigned to machine } i \\
    \} \\
    \text{for } j = 1 \text{ to } n \{ \\
        i = \text{argmin}_k L_k \quad \text{machine } i \text{ has smallest load} \\
        J(i) \leftarrow J(i) \cup \{j\} \quad \text{assign job } j \text{ to machine } i \\
        L_i \leftarrow L_i + t_j \quad \text{update load of machine } i \\
    \} \\
    \text{return } J(1), \ldots, J(m) \\
\}
\]

k-Center Selection: Greedy Algorithm

Greedy algorithm [Gonzalez 1985]. Repeatedly choose the next center to be the site farthest from any existing center.

\[
\text{Greedy-Center-Selection}(k, n, s_1, s_2, \ldots, s_n) \{ \\
    C = \emptyset \\
    \text{foreach } s_i : \text{dist}(s_i, C) = \infty \\
    \text{repeat } k \text{ times} \{ \\
        \text{Select a site } s_i \text{ with maximum dist}(s_i, C) \\
        \text{Add } s_i \text{ to } C \quad \text{site furthest from any center} \\
        \text{foreach } s_j : \text{dist}(s_j, C) = \text{distance to closest site in } C \\
    \} \\
    \text{return } C \\
\}
\]

Observation. Upon termination all centers in \( C \) are pairwise at least \( r(C) \) apart.

\textbf{Pf.} By construction of algorithm.
Pricing Method

**Pricing method.** Set prices and find vertex cover simultaneously.

```plaintext
Weighted-Vertex-Cover-Approx(G, w) {
    foreach e in E
        \( p_e = 0 \) \( \sum_{e \in \{i, j\}} p_e = w_i \)
    while (3 edge i-j such that neither i nor j are tight)
        select such an edge e
        increase \( p_e \) as much as possible until i or j tight
    \}
    S ← set of all tight nodes
    return S
}
```

Weighted Vertex Cover: LP Relaxation

**Weighted vertex cover.** Linear programming formulation.

\[
\begin{align*}
(LP) \quad \min \quad & \sum_{i \in V} w_i x_i \\
\text{s.t.} \quad & x_i + x_j \geq 1 \quad (i, j) \in E \\
\quad & x_i \geq 0 \quad i \in V
\end{align*}
\]

**Observation.** Optimal value of (LP) is \( \leq \) optimal value of (IP).

**Pf.** LP has fewer constraints.

**Note.** LP is not equivalent to vertex cover.

**Q.** How can solving LP help us find a small vertex cover?

**A.** Solve LP and round fractional values.
**Example Knapsack**

**Knapsack PTAS.** Round up all values: $\hat{v}_i = \left\lfloor \frac{v_i}{b} \right\rfloor b$, $\hat{v}_j = \left\lfloor \frac{v_j}{b} \right\rfloor b$

**Knapsack-Approx($\epsilon$) {**

$\epsilon = \frac{1}{10}$

$b = (\epsilon / (2n)) \cdot 27343199$

$= (1/100) \cdot 27343199$

$S \leftarrow$ solve Knapsack with values $\hat{v}_i$

return $S$

**}**

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>934,221</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5,956,342</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>17,810,013</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>21,217,800</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>27,343,199</td>
<td>7</td>
</tr>
</tbody>
</table>

original instance

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>78</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>7</td>
</tr>
</tbody>
</table>

rounded instance

**Contraction Algorithm**

**Contraction algorithm.** [Karger 1995]

- Pick an edge $e = (u, v)$ uniformly at random.
- Contract edge $e$.
  - replace $u$ and $v$ by single new super-node $w$
  - preserve edges, updating endpoints of $u$ and $v$ to $w$
  - keep parallel edges, but delete self-loops
- Repeat until graph has just two nodes $v_1$ and $v_2$.
- Return the cut (all nodes that were contracted to form $v_1$).
Recall: Closest Pair Algorithm

\textbf{Closest-Pair}(p_1, \ldots, p_n) \{
\begin{description}
\item[Compute] separation line \( L \) such that half the points are on one side and half on the other side.
\item[\( \delta_1 \)] = Closest-Pair(left half)
\item[\( \delta_2 \)] = Closest-Pair(right half)
\item[\( \delta \)] = min(\( \delta_1 \), \( \delta_2 \))
\item[Delete] all points further than \( \delta \) from separation line \( L \)
\item[Sort] remaining points by \( y \)-coordinate.
\item[Scan] points in \( y \)-order and compare distance between each point and next 11 neighbors. If any of these distances is less than \( \delta \), update \( \delta \).
\end{description}
return \( \delta \).
\}