Homework 1

CSE 202 — Fall 2012

Allyson Cauble-Chantrenne **John Chau** **Charlie Chen**
Nathan Feldman **James Mouradian**

November 8, 2012

Problem 1

Problem: You are devising a flight scheduler for a travel agency. The scheduler will get a list of available flights, and the customer's origin and destination. For each flight, it is given the cities and times of departure and arrival. The scheduler should output a list of flights that will take the customer from her origin to her destination that arrives as early as possible, subject to giving her at least one hour for each connection.

Solution:

Part A - Give a formal specification for this problem:

An instance of the problem is a tuple (G, F, s, t), where G is a graph (V, E), F is a set of flights (v_d, t_d, v_a, t_a), where v_d is the airport of departure, v_a is the airport of arrival, t_d is the time of departure, and t_a is the time of arrival, $s \in V$ is the starting airport, and $t \in V$ is the destination airport.

The solution space is the set of sequences of flights $f_1, f_2, \cdots, f_i = (v_{di}, t_{di}, v_{ai}, t_{ai}), \cdots f_k$, such that $s = v_{d1}$, $t = v_{ak}$, and $\forall i \in 1, \cdots, k-1, v_{ai} = v_{d(i+1)}$, with the additional constraint that $t_{d(i+1)} \geq t_{ai} + 1$ hour.

The objective is to find the sequence of flights with the earliest t_{ak}.

Part B - Give as efficient as possible an algorithm to solve the problem:

The algorithm is a modified version of Dijkstra's:

```
$R \leftarrow \{\}$, $Q \leftarrow \{s\}$, $U \leftarrow V - \{s\}$, $D[u] \leftarrow \infty\ \forall\ u \in$ airports,
    except $D[s] = 0$, $prev[u]$ <- NULL $\forall\ u \in$ airports.
Let $uflights(u, v)$ be an empty list of unavailable flights from $u$ to $v$, to be
    ordered by the latest departure time $t_d$
Let $aflights(u, v)$ be an empty heap of available flights from $u$ to $v$, to be
    ordered by earliest arrival time $t_a$
For every $f = (s, t_d, v_a, t_a)$:
    insert $f$ into $aflights(s, v_a)$
For every $f = (v_d, t_d, v_a, t_a) \in F$ where $v_d \neq s$:
    insert $f$ into $uflights(v_d,\ v_a)$
Sort every list in $uflights(u,\ v)$ in reverse chronological order of arrival time
While $Q\ \neq\ \{\}$
    $u\ \leftarrow\ argmin_{w\ \in\ Q}D[w]$, and remove $u$ from $Q$
    $\forall\ v\ \in\ N(u)$
        if $v\ \in\ U$, continue (skip to next iteration of for loop)
        if $v\ \notin\ Q$, insert $v$ into $Q$
        $(u,\ t_d,\ v,\ t_a)\ \leftarrow$ flight of earliest arrival time in $aflights(u,\ v)$
        if $t_a$ is earlier than $D[v]$,
            $D[v]\ \leftarrow\ t_a$
            $prev[v]\ \leftarrow\ u$
            for every outgoing edge $(v,\ w)$
                move all flights in $uflights(v,\ w)$ where $t_d\ \geq\ D[v]\ +\ 1$ hour
                    to $aflights(v,\ w)$
Let $a\ =\ t$, $result$ be an empty list of airports
While $a\ \neq$ s
 if $a$ is NULL then
 Output "No path"
 break out of while loop
    add $a$ to $result$
    $a\ =\ prev[a]$
Reverse $a$
Output $result$
```

Time Analysis:

Let n be $|F|$. Inserting the flights into the lists of their corresponding directed edges, $uflights(u,v)$ takes $O(n)$ time. (We'll examine the initial inserting into $aflights(s,v_a)$ in a bit.)

While sorting a list $uflights(u,v)$ for every edge seems rather daunting, we're only sorting a total of n flights, and sorting several lists whose lengths add up to n would take no longer than sorting one list of length n with the same prior ordering. Therefore, This step takes $O(n*\log n)$ time.

The outer loop runs a total of $|V|$ times, since in every iteration, one node is removed from Q, and every node can only be put into Q once.

If we implement Q as a mininum heap on $D[u]$, then removing u takes $O(\log|V|)$ time.

We can consider the first for loop separately from the outer while loop, since the for loop will be run exactly once for each edge in the graph, a factor of $|E|$.

Insertion into Q can take $O(\log|V|)$ time.

Finding the earliest flight in the heap $aflights(u,v)$ takes constant time.

The moving from $uflights(v,w)$ to $aflights(v,w)$ is done a total of n times, including when we add flights into $aflights(s,v_a)$ in the first step of the algorithm. Removing from $uflights$ takes $O(1)$ time, and inserting into $aflights$ takes $O(\log a)$. The movement in total therefore takes $O(n*\log a)$, and we can consider it separate from the containing loops. (Technically, however, the innermost for loop runs once for each neighbor of the destination airport of each edge, meaning $O(|E|*|V|)$.)

Following the path from t back to s and then reversing it takes $O(|V|)$ time.

The combined time is then: $O(n)+O(n*\log n)+O(|V|)*O(\log|V|)+O(|E|)*(\ O(\log|V|)[+O(|V|)]\ )+O(|V|)$, the technicality being the part wrapped in square brackets.

Keeping in mind that $|V|$ and $|E|$ are both bounded by n, this simplifies to $O(n * (\log n[+n]))$ Finding a way to remove the small technicality would yield $O(n * \log n)$, but until then, the algorithm is technically $O(n^2)$.

Proof of Correctness:

The algorithm is correct for exactly the same reason that Dijkstra's Algorithm is correct, except that the cost to reach an airport u, $D[u]$, is the earliest possible arrival time at u:

We want to prove that there exists no path from s to t such that the earliest arrival time at t is earlier than $D[u]$. Assume for contradiction, then, that there does exist some path, $s = u_1, u_2, u_3, ..., u_{k-1}, u_k = t$, with flights $(u_i, t_{di}, u_{i+1}, t_{a(i+1)})$ from $i = 1$ to $k-1$, such that t_{ak} is earlier than $D[t]$ (the arrival time of any one flight being at least an hour before the next).

Then there exists some index $i \in \{1, \cdots, k-1\}$ such that $t_{ai} = D[u_i]$ but $t_{a(i+1)}$ is earlier than $D[u_{i+1}]$. But the algorithm takes the flight arriving at u_{i+1} with the earliest possible arrival time that departs at least an hour after $D[u_i]$, so $D[u_{i+1}] \leq t_{a(i+1)} < D[u_{i+1}]$, which is a contradiction.

Thus, our assumption must be false, and the sequence of flights chosen must be optimal.

Problem 2

Problem:

You are given a directed graph with edge weights 1, 0, or -1. A *valid* path is one where, for all k, the sum of the weights of the first k edges is non-negative. The problem is, given nodes s and t and such a graph G, to determine whether there is a valid path from s to t.

Solution:

The Algorithm:

```
Initialize maxVals[i] to -1 for i = 1 to |V|
Enqueue <s, 0> in Queue
While Queue is not empty:
    dequeue < u, partialSum > from Queue
    If u is t then
        Path found! (and break)
    If maxVals[u] < partialSum then
        maxVals[u] <- partialSum
        For each v in Neighbors(u)
            If maxVals[u] + weight((u, v)) > maxVals[v]
                and maxVals[u] + weight((u, v)) <= |V| then
                Enqueue < v, maxVals[u] + weight((u, v)) > in Queue
```

To prove that this algorithm is correct, we must show that it terminates, and that it finds a valid path iff one exists.

Termination:

The algorithm continues as long as *Queue* is not empty. At each iteration, at least one element is removed from *Queue*, and a maximum of $|V| - 1$ elements are inserted. Also, each element can be inserted into $Queue$ at most $|V|$ times, so there are a maximum of $|V|^2$ iterations. Thus, the algorithm terminates.

If a valid path exists, the algorithm finds one:

Assume that a valid path exists in the graph G, and let that path be defined as $u_1 u_2 \cdots u_k$ where $s = u_1$ and $t = u_k$. Also assume that the algorithm does not find this valid path. That means that the node u_k was never inserted into $Queue$. Otherwise, it would have been dequeued since the algorithm does not terminate until $Queue$ is empty, and when u_k is dequeued, a path has been found. This implies that $\forall u \in V$ s.t. $u_k \in Neighbors(u)$, u was not enqueued into $Queue$, or $maxVals[u] + weight((u, u_k)) < 0$. This means that at some u_i where $i \leq k$, all $u \in V$ s.t. $u_i \in Neighbors(u)$, $maxVals[u] + weight((u, u_i)) < 0$. But, this means that there is no valid path, which is a contradiction. Thus, if a valid path exists, the algorithm will find one.

If the algorithm returns "Path found", then a valid path exists:

Assume that our algorithm found a path $u_1 u_2 \cdots u_k$ in the graph G, where $s = u_1$ and $t = u_k$. Assume that this is not a valid path. Then, for some u_i in the path where $i < k$, the partial sum at u_i is 0, and the weight of edge (u_i, u_{i+1}) is -1. This means that $maxVals[u_i]$ is 0 and that $maxVals[u_{i+1}]$ is -1. But this means that u_{i+1} could not have been inserted into $Queue$, because $maxVals[u_i] + weight(u_i, u_{i+1}) = -1$ which is not greater than -1. Because u_{i+1} was never in $Queue$, u_{i+1} could not be in the path found by the algorithm. This is a contradiction. Thus, if the algorithm finds a path, then it is a valid path.

Time Analysis:

Initialization of $maxVals$ is $O(|V|)$. As discussed in the Termination proof, there are a maximum of $|V|^2$ iterations of the while loop. Each iteration includes scanning at most $|V| - 1$ neighbors, and doing constant operations on these neighbors. Enqueuing and Dequeuing are also $O(1)$. So, the total time in the while loop is $O(|V|^3)$, making the total running time of the algorithm $O(|V| + |V|^3) = O(|V|^3) = O(n^3)$, where n is the total number of nodes.

Problem 3

Problem:

UCSD needs long-term contracts for power to keep the lights on. It has a list of n bids B_I , $1 \leq I \leq n$ from power companies, each with a *rate* (cost per megawatt) r_I and a *capacity*C_I (the maximum number of megawatts the company can guarantee). The regents, to encourage low bids, has guaranteed that they will pay for all accepted bids at the highest rate of any accepted bid. The accepted bids capacities must sum to at least M, the universitys demand for power, to ensure enough power. In addition to the money paid per megawatt, the regents expect each contract to cost a fixed amount F for lawyers, setting up connections to the grid, etc. So the total cost will be $Fk + rM$, where k is the number of accepted bids, and r is the maximum rate of an accepted bid. Give an efficient algorithm, polynomial-time in n, that, given F, M and the list of n bids B_I, computes a subset of bids A that minimizes the cost to the university subject to ensuring enough power.

Solution:

The Algorithm:

```
Define a structure <rate, capacity> for a bid
Let R be a list of all bids sorted in descending order by rate
Let C be a list of all bids sorted in descending order by capacity
Let Q be the cost of the final contract list, initialized to $\infty$
Let S be the set of final contracts we take, initialized to the empty set
For each unique rate r in R do:
    Let E be an empty set of bids to include
    Let m be 0
    For each element bid b in C do:
        If the rate of b is less than or equal to r then:
            Add b to E
            Add the capacity of b to m
            If the m is greater than or equal to M then:
                (Potential) Solution found
                Break (out of this for loop)
    If (potential) solution found:
        Let K be F*|E|+r*M
        If K is less than Q then:
            Let Q be assigned to K
            Let S be assigned to the set E
    Else:
        Break (out of main for loop)
```

Proof of Correctness and Complexity

In order to prove our algorithm is correct, we must show that it **1** that it terminates in finite time, and **2** if a solution exists, that it returns the optimum of all existent solutions.

1. Complexity and Termination

Our algorithm has a finite running time, and therefore terminates. We begin by constructing two sorted lists, which takes $O(n\log(n))$ time. Then, we search for our optimum solution using these lists. In our search, we consider at most n distinct rates for our contract. For each distinct rate, we search linearly through a sorted list of n contracts to assemble a possible solution for this rate. As such, we have at most $O(n^2)$ operations in our search.

The final complexity of our algorithm is $O(n^2 + n\log(n)) \in O(n^2)$.

2. Optimality

We know that any solution to this problem will have a final maximum fixed rate r. For a given fixed r, the value rM is also fixed, so our optimal solution is one which minimizes the value Fk. Since F is fixed, the optimal solution given r is one which minimizes k.

In order to minimize k, we greedily take bids with rate at most r by maximum capacity, which is performed by our innermost loop. This is the optimum solution. Suppose it is not. Then, we can compare it to some other hypothetical algorithm which yields the optimum solution. Suppose that hypothetical algorithm did not take the bid of maximum capacity possible at each step. Then, by the "greedy stays ahead" argument, it is clear that at each iteration of taking a new bid and adding it to our set that the greedy algorithm always has a net capacity at least that of the other algorithm. This means the greedy algorithm meets the capacity M at least as early as the optimal solution, so it collects at most as few bids as our optimal solution. Therefore, our greedy algorithm for meeting capacity M for a fixed r yields the optimum solution for a given value of r.

There are many different potential values of r which are the correct maximum rate to choose. Since we do not know a priori which rate r to use as a maximum, we consider all possible distinct values of r in our algorithm, and naively compute the optimum contract cost for the fixed r. Since each net computation is optimum for each given r, and we consider all possible values of r, our algorithm finds the global optimum net contract cost.

Problem 4

Problem:

Let $A[1, ...n]$ be an array of positive integers. A summing triple in A is a set of 3 distinct indices $1 \leq i, j, k \leq n$ so that $A[i] + A[j] = A[k]$. Give and analyze an algorithm that, given A, determines whether there is any summing triple in A. Your algorithm must be faster than $O(n^3)$. (3 points correct algorithm and proof of correctness, 7 points efficiency and time analysis).

```
bool FindTriple (int[] A)
{
     int i, j, k;

     sort A from smallest to largest

     for each k from n-1 down to 2
     {
          i = 0;
          j = k - 1;
          while i is less than j
          {
               if A[k] equals (A[i] + A[j])
               {
                    return true; // found a sum
               }
               else if A[k] is greater than (A[i] + A[j]) //sum too small
               {
                    i = i + 1 // move left index up to increase sum
               }
               else // sum too large
               {
                    j = j - 1 // move right index down to decrease sum
               }
          }
     }
     return false; // no sum found
}
```

Proof of Correctness: To prove that this algorithm is correct we need to prove that if a valid summing triple exists in the array, the algorithm returns true. To show this we need to prove that **(1)** our algorithm explores all possible values of $A[k]$, and **(2)** for each $A[k]$ value, if a summing pair $A[i], A[j]$ exists, that our algorithm will find it. It should be obvious that our algorithm will never return true if a summing triple does not exist, so we only prove our other two claims in the remainder of this section.

1) *Exhausting $A[k]$ values:* Our algorithm explores all values of $A[k]$ by incrementally iterating through the array from the largest value $A[n-1]$ to the smallest plausible $A[k]$ value of $A[2]$. Since $A[0]$ and $A[1]$ are the smallest values in our array, and our array entries are positive, no two distinct array entries can sum to $A[0]$ or $A[1]$, and all plausible $A[k]$ values are therefore explored.

2) *Finding the correct triple for a given $A[k]$ value, if a triple exists:* Since our array is sorted with positive values, we do not need to check indeces greater than k in our array as prospective $A[i], A[j]$ values, because all such values will be greater than $A[k]$; no such entry could be used with any other to sum to $A[k]$. Our search space is therefore $A[0]$ to $A[k-1]$, which are our starting search points for $A[i]$ and $A[j]$, respectively. If the current $A[i] + A[j]$ sum equals $A[k]$, we've found a triple and return true, and our algorithm terminates correctly.

We are left with the task of proving that our algorithm will never reach a state in which it cannot find the correct pair $A[i], A[j]$ which sums to a $A[k]$. Let $A[i], A[j], A[k]$ be a summing triple in our array, where $A[i]$ and $A[j]$ are the outermost summing pair for $A[k]$. Let i_p and j_p denote the values our algorithm is investigating as candidate i and j values. Assume that not both of i_p and j_p equal i and j, respectively. Then, the state of our algorithm can be categorized in one of four ways:

1. $i_p \leq i$ and $j_p \geq j$
2. $i_p > i$ and $j_p \geq j$
3. $i_p \leq i$ and $j_p < j$
4. $i_p > i$ and $j_p < j$

1. Neither i_p nor j_p have been incremented or decremented beyond their target values. Our algorithm may continue to increment i_p and decrement j_p, so this is not contradictory to our algorithm's correctness.
2. Suppose our algorithm has incremented i_p beyond i, but not decremented j_p beyond j. This means that for some $j'_p \geq j$, our algorithm incremented $i_p = i$ to $i + 1$. However, since our array is sorted, we know that $A[i_p] + A[j'_p] \geq A[i] + A[j]$. Since our algorithm did not terminate at this step, we know that $A[i_p] + A[j'_p] > A[i] + A[j]$. However, in this scenario, our algorithm would decrement j'_p, rather than increment i_p, so **2** will never occur.
3. Suppose our algorithm has decremented j_p beyond j, but not incremented i_p beyond i. This means for some $i'_p \leq i$, our algorithm decremented $j_p = j$ to $j - 1$. Since our array is sorted, we know that $A[i'_p] + A[j_p] \leq A[i] + A[j]$, and since our algorithm did not terminate, we know in fact that $A[i'_p] + A[j_p] < A[i] + A[j]$. However, in this scenario, our algorithm would increment $A[i'_p]$ rather than decrement j, so **3** will not occur.
4. Our algorithm has incremented and decremented both i and j beyond their target values. Since only one of these can occur at a time, our algorithm must have at some point passed through either **2** or **3**, which is impossible, so **4** will never occur.

Therefore, if a summing pair $A[i], A[j]$ exists for a given value of $A[k]$, our algorithm will find it. Moreover, since we exhaust all possible $A[k]$ values, if a summing triple exists, our algorithm is guaranteed to find it.

Efficiency: $O(n^2)$
Time Analysis: Sorting the array A from smallest to largest using the best sorting technique takes at most $0(nlogn)$ time. Once sorted, we iterate through each array position from greatest to smallest, checking if there exists a sum equal to the current cell. In determining whether there exists a sum for cell $A[k]$ in the remaining $A[0]...A[k-1]$ cells, we check at most $k-1$ cells which takes $O(k)$ time. Since there are n elements in the array and we perform $O(k)$ checks for each cell, this takes $O(nk)$ time. We know that k is never greater than n so we can also say it takes $O(n^2)$ time. Adding the running time for sorting the array and checking the sum for each array element, we obtain $[O(nlogn) + O(n^2)] \in O(n^2)$.

Problem 5

Problem:
Implement the algorithm from **Problem 4**, and plot results for random array values whose maximum m belongs to the set $\{2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$. Try squaring the value of m, and seeing whether this impacts the running time of the algorithm.

Solution: Using values of m as small as suggested in the problem did not yield sufficient results. For small arrays, the algorithm executed very quickly. For large arrays, due to a more generalized form of the birthday paradox, the existence of a summing triple was overwhelmingly probable, and the algorithm would therefore terminate very quickly. Many of our time tests evaluated to zero. As such, we increased our base values of m for this experiment to $\{2^{18}, 2^{20}, 2^{22}, 2^{24}, 2^{26}, 2^{28}, 2^{30}\}$, and included an additional plot with a linear scale rather than a log scale for time values. We plotted our results with array sizes of $n = 15,000$ and $n = 20,000$. We ran ten trials for each combination of n and m, and super-imposed the trials on each plot. Trials with $n = 15,000$ have a distinct glyph from trials with $n = 20,000$.

.

When squaring the values of our m, the algorithm definitely takes longer to run. This is because with such an increased range in values of m, the probability of finding a summing triple of numbers in our array is greatly reduced. As such, the algorithm is more likely to scan more elements of the array. When the value of m gets too large, the probability of finding a summing triple is so low that the array is scanned in entirety almost every time, and the average running time of our algorithm levels off.

Hw1GraphLogLog.jpg

Figure 1: Maximum $A[i]$ value against time taken for our algorithm to execute.

Hw1GraphLogRegular.jpg

Figure 2: The same plot, with the time (y) axis in a linear scale rather than a log scale.