CMSC 27200 Assigment 2

Rachel Hwang

February 25, 2014

- **Making Change**

  a. Given the American currency system, the greedy algorithm always returns an optimal solution because given any target change value v, it is always possible to find an optimal solution which includes the largest demonination smaller than v.

  By way of contradiction, let's assume the greedy algorithm doesn not produce an optimal solution with American coins. For some target value, let G be the coin list returned by the greedy algorithm and O be an optimal solution to the problem, where both lists are arranged in decreasing order of denomination value. If G is not optimal, then it must differ from O at some index i. The greedy algorithm guarentees that its denomination at i, i_g is larger than i_o. Let's say that the remaining cent value of the original value is v at index i (must be the same for both lists since i is the first index at which they differ). One of the following cases must be true at this point:

  (a) i_g is a nickel.
  This means i_o is a penny. Since coins occur in decreasing order, O will return only pennies from this point onward. Since there must be at least five cents left in value, O must contain at least another 5 pennies, 5 coins where G uses 1. This creates a contradiction, as the optimal solution uses more coins than the greedy.

  (b) i_g is a dime.
  This means that O can use only nickels and pennies at this point. Since there must be another ten cents at least in value, I must use at least 2 coins where G uses one. Again, this creates a contradiction.

  (c) i_g is a quarter.
  O is limited to dimes, nickels and pennies to create at least 25 cents of value. If O uses three or more dimes, three of the dimes can be exchanged for a quarter and a nickel, which is what the greedy algorithm would do, using two coins instead of three. If O uses only two dimes or less, then it must use at least 3 coins, two dimes and a nickel to create 25 cents. In either of these subcases, switching to a configuration that involves a quarter would improve O, which again gives us a contradiction.

  Therefore by contradiction, the greedy algorithm returns the optimal solution which uses a minimal number of coins.

  b. Consider the following counterexample: You have three demoninations, a 12 cent coin (12C), an 11 cent coin (11C) and a 1 cent coin (1C) and you want to make 100 cents in change. The greedy algorithm will pick as many (12C) as it can, which is 8, making 96 cents, then fill in the rest with (1C). In total, it uses 8 (12C) + 4 (1C) = 12 coins to make 100 cents. However, this is not the optimal solution. By starting with (11C), we can use only 10 coins, 9 (11C) + 1 (1C).

- **Exercise 4.2**

  a. True. Assume by way of contradiction that T is no longer a minimum spanning tree of G after squaring costs. Then there must be some edge $e = (v, w)$ in T which does not belong in the minimum spanning tree. If this is the case, we know by (4.41) in the textbook that there must be some path p from v to w consisting of edges of lesser cost that that of e. However, we also know that since T was a spanning tree originally, at least one edge f in p must have cost more than e, otherwise e could not have been a part of T. This means that in the original G, $C_f > C_e$, but after squaring, $C_f < C_e$. Since if $a < b$, $a^2 < b^2$, this is a contradiction, therefore T must still be a minimum spanning tree.

  b. False. Consider the following counterexample: G consists of $V = \{a, b, c\}$ such that cost(a, b) = 4, cost(b, c) = 2, and cost(a, c) = 5. Before squaring, the shortest path(a, c) is to simple traverse edge(a,

c) with cost 5. However, once all costs are squared, cost(a, c) = 25, whereas the path using edge(a, b) then edge(b, c) costs 16+4 = 20. Thus, the original path is no longer the least costly.

- **Exercise 4.4**
  Given S and S', the algorithm is as follows:

```
subsequence(S, S') {
    if (!S') {
        return True;
    }
    if (!S) {
        return False;
    }
    if (first(S) == first(S')) {
         return subsequence(rest(S), rest(S'));
    } else {
         return subsequence(rest(S), S');
    }
}
```

  Claim: This algorithm correctly determines whether S' is a subsequence of S.
  Proof: By induction on the length of S. In the base case, when $|S| = 1$, if $|S'| > 1$, the algorithm fails as it should. Otherwise, the algorithm compares S's single element to S''s single element and returns true if they are the same, false if not. (Note if $|S| = 0$, the algorithm returns False.) Therefore, the claim is true when $|S| = 1$. Now the inductive step: let $|S| = k$. The algorithm compares the first element of S to the first element of S'. If they are same, the algorithm is called again with $rest(S)$ and $rest(S')$. If not, call again with $rest(S)$ and S'. In either case, we have reduced the problem to case with $|S| = n - 1$, which we have assumed to be true by inductive hypothesis. Thus, the algorithm works for all lengths of S.

- **Exercise 4.7**
  Since the supercomputer can only perform one task at a time, the overall time will always be at least $\sum p_i$. Because the the PC tasks can be performed in parallel, jobs with large k times should be begun sooner, because having a job processed by a PC does not preclude other jobs k portions. In essense, when trying to find an optimal solution, we can ignore the p time of a job since p time will always contribute the same ammount of time to the total time, whereas a job's k time does not necessarily. For instance, if we have job 1 with p = 4 and k = 1, and job 2 with p = 2 and k = 3, if job 2 is run first, its k runtime is basically subsumed by the p time of job 1, giving a total time of 7. If job 1 is run first, total runtime is 9, since a longer k time starts after all p times are finished.

  The algorithm hence simply sorts jobs by finishing time, from longest k to shortest. The jobs are then scheduled, passed to the supercomputer, in that order.

  Similar to the proof for the optimality of the minimizing lateness algorithm in the textbook, proof of this algorithm's optimality will use the exchange argument.

  First let us characterize schedules as follows: a schedule S has an inversion if a job i with finishing time k_i is scheduled before another job j with a longer finishing time $k_j > k_i$. Notice that, by definition, the schedule A produced by our algorithm has no inversions. If there are jobs with identical finsihing times then there can be many different schedules with no inversions. However, we can show that all these schedules have the same total time. Note that we assume that no schedule has any idle time; that is, jobs are passed to the supercomputer as soon it is available, and a job is passed off to be finished as soon as it is done being processed.

  Claim 1: All schedules with no inversions have the same total time.
  Proof: If two different schedules have no inversions, then they might not produce exactly the same order of jobs, but they can only differ in the order in which jobs with identical finishing times are scheduled. Since the sum of p times is still an absolute minimum, no inversions means that there is no way to reorder to jobs to reduce the trailing finishing time once all jobs have been processed by the super computer. This means that the two schedules must have the same trailing finishing time, thus the same total time.

  Claim 2: There is an optimal schedule that has no inversions.

Proof: First, we know that if a schedule S has an inversion, then there is a pair of jobs i and j such that j is scheduled immediately after i and has k_j ¿ k_i. Let S be an optimal solution with n inversions. We can find a pair of adjancent jobs i then j where $k_j > k_i$ and switch them. Since swapping those two jobs does not affect any other jobs in the schedule, S now has $n-1$ inversions. Based on our definition of inversion, if S was an optimal solution before the swap, it must still be optimal after the swap. Since k_j ¿ k_i, beginning j first cannot increase the total ammount of time. Let's say k is the ammout of trailing time (after both jobs have been processed) in the original configuration where i comes before j. Since k_j ¿ k_i, trailing time must be determined by k_j, so $k = k_j$. After swapping one of two cases must be true: either k_j no longer contributes to trailing time (having been subsumed by p_i, in which case now $k = k_i$; otherwise k_j still contributes to trailing time, in which case k still must be shorter than before since k_j's possible time contributions have now been reduced by p_i. Since the total time cannot increase by removing this adjancent inversions, the schedule must still be optimal after this swap.

We have shown that it is possible to take any optimal schedule containing n inversions and remove one to get an optimal solution containing n-1 inversions. Given that we known that some optimal schedule must exist, there must be an optimal schedule with no inversions.

Claim 3: The algorithm defined above produces an optimal schedule.

Proof: By the construction of our algorithm, because it schedules jobs by decreasing k value, any produced solutions contain no inversions. By Claim 2, there is some schedule with no inversions that is optimal. By Claim three, all schedules with no inversions have the same total time, so all schedules with no inversions are optimal, therefore schedules produced our algorithm must be optimal.

- **Exercise 4.10**

  a. The algorithm first runs a depth first search over an adjacency list of the graph beginning from v, searching for w. As the search runs, record costs of the edges that it traverses. Whenever the search "backtracks", that is, whenever the algorithm has to return to a node where it picked one neighbor to explore, delete all recorded costs after the cost of that node. This ensures that once the search finds w, it will have recorded the weights of all the path from v to w. The algorithm then compares the cost of e to each recorded cost from the path. If e is less than any of those costs, then T is no longer is minimum spanning tree. Otherwise, T is still a MST.

  This works because of (4.41) from the textbook. By (4.41), if each edge in a path(v,w) costs less than e, e cannot be a part of the MST. If e is not a part of the MST, every other edge in T must still be a part of the MST since no other aspect of the graph has changed, thus T must still be a MST.

  This algorithm runs in $O(|V|)$ time. Running the depth first search requires checking at most $|V|$ vertices. Once the path has been recorded, e's cost must be compared to each of at most $|V|-1$ other vertices. Assuming each of these operations on a single vertex takes constant time, the algorithm therefore takes a maximum of $O(2|V|-1) = O(|V|)time$.

  b. Consider $T \cup \{e\}$, the cycle created by adding e. Similar to in the previous problem, we then want to record every cost of the edges in the cycle by using a depth first search beginning at v and ending at w, then adding the cost of e. We then want to find the maximum cost in this cycle and remove the associated edge. All the remaing edges together compose T', the new MST for G.

  Proof: By way of contradiction, assume T' is not the MST for G, then there exists some edge e'' for which $T' \cup \{e''\}$ has a cycle with an edge more expensive than e''. But if this is the case, then T cannot originally have been a MST, which is a contradiction. Thus, T' must be a minimum spanning tree.

  Again using the depth first search requires only $O(|V|)$ at maximum. Removing the maximum edge requires checking all the edges in the cycle, which is another $|E|$ checks. Since $|E| > |V|$, the final complexity is $O(|E|)$.