Homework 7

Frederick Robinson and Mykell Miller

9 March 2010

1 Problem 11.1

1.1 Algorithm

This algorithm is similar to the one explained in the book with a few minor changes.

For ease of notation we will refer to $\max w_i$ as w^*.

First, we discard any weights w_i that are bigger than the capacity of our knapsack C. Second, we see if $|W| = \sum_{i=1}^{n} w_i \leq C$. If this is the case, then everything fits and we are done.

After finishing this preprocessing, we round our initial knapsack problem. However, instead of rounding the values we round the weights. In particular round each weight w_i *down* to the nearest $b = (\epsilon w^*)/(2n)$ and call the rounded value w_i'. Then divide C by b and call it C'. For all i, divide w_i' by b. Having done this, our value for the maximum rounded weight say w'^* becomes $w'^* = w^*/b = (2n)/\epsilon$.

Now take the algorithm from page 272 and run it with our new weights and capacities.

Return each w_i corresponding to each w_i' in the solution returned by the algorithm on page 272.

1.2 Runtime

Preprocessing runs in $O(n)$. The algorithm on page 272 runs in $O(nC')$ for integral knapsack. Of course $C' \leq w'^*n$ (otherwise everything fits and we have a trivial solution which was checked for in preprocessing). Hence, we have a solution which runs in $O(n^2 w'^*)$. So our runtimes on the rounded problem is just $O(nC') \leq O(n^2 w'^*) = O(n^3/\epsilon)$ which is polynomial for constant ϵ as desired.

1.3 Correctness

Lemma 1.1. *The total weight in our solution is at most $C(1+\epsilon)$.*

Proof. Let S be the solution found by our approximation algorithm, and note that $\max w_i \leq C$ by our first preprocessing step. Say that there are n elements in $|S|$. We first establish that since $bw'^* = w^*$ we have

$$\frac{w^*}{2} < \sum_{i \in S} bw'$$

$$\Rightarrow \frac{w^* n}{2n} < \sum_{i \in S} bw' \Rightarrow \epsilon \frac{w^* n}{2n} < \epsilon \sum_{i \in S} bw'$$

but by definition

$$\epsilon \frac{w^* n}{2n} = nb < \epsilon \sum_{i \in S} bw'$$

Now notice that due to our rounding we have

$$\sum_{i \in S} w_i < \sum_{i \in S} b(w_i' + 1) = nb + \sum_{i \in S} bw_i' < \epsilon \sum_{i \in S} bw' + \sum_{i \in S} bw_i' = (1+\epsilon) \sum_{i \in S} bw_i'$$

However we have by correctness of the integral algorithm that

$$\sum_{i \in S} w_i' < C' \Rightarrow \sum_{i \in S} bw_i' < bC' < C$$

so combining this with the above we get

$$\sum_{i \in S} w_i < (1+\epsilon) \sum_{i \in S} bw_i' < (1+\epsilon) C$$

as desired. □

Lemma 1.2. *Our solution has value at least the value of the optimal solution to the knapsack problem.*

Proof. Suppose towards a contradiction that this is not the case. But, since we round down and

$$\sum x_i \leq v \Rightarrow \sum \lfloor x_i \rfloor \leq \lfloor v \rfloor$$

the optimal solution to the knapsack problem is a feasible solution to our rounded problem. Then, by correctness of the integral knapsack algorithm this is a contradiction. □

Theorem 1.3. *Our algorithm is correct.*

Proof. Lemmas 1.1 and 1.2 together prove that our algorithm is correct. □

2 Problem 1

2.1 Algorithm

Begin by adding an arbitrary node to the cycle.

If there are any adjacent edges with the property that removing it will not cause the remaining graph to be disconnected, add one of them to the cycle and at the same time remove it from the graph. If removing this edge caused a node to be adjacent to no edges remove this node. Now, repeat this procedure.

If the algorithm ever reaches a node with no adjacent edges whose removal will not cause the remaining graph to be disconnected check that this is the vertex where we started and that there are no remaining vertices or edges in the graph. If this is not the case then terminate returning "no cycle."

2.2 Correctness

If the algorithm returns a cycle then the cycle will be valid. In particular note that our algorithm only returns a cycle if it has traversed each edge (it checks this explicitly). Moreover, it cannot traverse an edge more than once since, after it traverses an edge for the first time it removes it from the graph.

Furthermore it is clear that our algorithm must terminate, given sufficient time. Indeed each step of our algorithm either terminates it or removes an edge from the graph. If there are no edges remaining in our graph the algorithm clearly has no edges to choose from and therefore terminates. Since the graph is assumed to have finitely many edges the algorithm must eventually terminate.

If there exists a cycle in the graph which satisfies the specified criteria then it (or some other such cycle) will be returned by the algorithm.

Proof. Suppose towards a contradiction that this is not the case. Then, since we proved that our algorithm terminates it must have reached a vertex with the property that there are no adjacent edges such that removing them will not cause the graph to become disconnected. Furthermore, it must be that either this is not the vertex on which the algorithm started, or alternately that there still remain edges in the graph which have not been traversed. We will deal with these two cases separately

Case 1: Not the vertex where we started:

Since every addition to the path causes the remaining graph to be connected it is never the case that we cannot reach the start vertex from the current one. That is, as we begin our computation of the cycle there is a path from the current node to the end on unused edges because we are assuming that a solution exists. Moreover, adding an edge will never cause the remaining graph to become disconnected. Thus by induction there is always a path to the node where we started.

Case 2: There are still edges:

If there are edges in the graph which have not been chosen but which are not accessible from the the current node this is a contradiction. In particular the current graph is disconnected. Therefore our algorithm could not have gotten to this state since it checks to make sure it is not disconnecting the graph. □

Now that we have established the fact that our algorithm will always return a valid cycle if said exists we need only show that if there is no valid cycle then our algorithm will indicate this. However this is easy to see.

In particular a cycle is invalid if it uses the same edge more than once. However, our algorithm removes an edge from consideration immediately after it has been used so it can construct no such paths. The only other way in which a cycle may be invalid is if it does not include all the edges or if it starts and ends in different places. We however explicitly check for these at the end of the algorithm. Thus, we could never return a path that has these shortcomings. Hence, if there is not a valid cycle on the graph we certainly do not return one.

2.3 Runtime

First recall that checking to see whether or not a given graph is disconnected can be done in $O(m+n)$ time using BFS. At every node we check the adjacent edges to see whether or not their removal will cause the graph to become disconnected. Since each node has at most n adjacent edges we must perform this check at most n times for each edge we add to the cycle. A finished cycle has m edges (and if we terminate without finding a cycle we go through less than m eges). Therefore, our algorithm runs in $O(mn(n+m)) = O(mn^2 + nm^2)$ time.

3 Problem 2

3.1 Algorithm

1. Sort edges in order from greatest value to least value.
2. Make an array of nodes, and label them all unused.
3. Make a number *total* and set it to 0.
4. For each edge e until none left:
   (a) If both adjacent nodes are unused:
      i. Mark them used.
      ii. Add the value of e to *total*.
5. Return *total*

3.2 Runtime

Sorting takes $O(m \log m)$. Making an array of nodes takes $O(n)$. The for loop runs m times, and runs in constant time. Therefore, the running time is $O(m \log m + n)$.

3.3 Correctness

Definition Edge (u,v) *blocks* all other edges adjacent to either u or v. If an edge (u',v') is *blocked*, the matching (u',v') cannot be in the bipartite matching.

Lemma 3.1. *If OPT has n matchings, then Greedy has at least n/2 matchings.*

Proof. Each matching involves at most two vertices. Therefore, if Greedy chooses a matching that is not in OPT, it blocks at most two matchings in OPT. Thus, for each "mistake" Greedy makes, two correct matchings are replaced by one incorrect matching. There are only $n/2$ pairs of correct matchings, so at worst Greedy can make $n/2$ "mistakes" thus resulting in $n/2$ matchings total. □

Theorem 3.2. *If OPT has a total value of v, then Greedy has a total value of at least v/2.*

Proof. By the lemma, we know that Greedy has at least $n/2$ matchings. Since Greedy chooses the edges with greatest value first, any matching e chosen by Greedy has a value greater than or equal to the value of any matching in OPT that e blocks. At worst, if e has a value of w, it blocks two matchings in OPT each with a value w. Thus, for each disjoint pair of matchings i in OPT with a combined value w_i, Greedy has one or more matchings with a combined value at least $w_i/2$. The total value of OPT is $\sum_{i=1}^{n/2} w_i = v$, so the total value of Greedy is at least $\sum_{i=1}^{n/2} w_i/2 = v/2$.

□

4 Problem 3

4.1 Part A

4.1.1 Algorithm

1. Take a Number Partitioning problem with positive integers $W = w_1, \ldots, w_m$, and set $|W| = sum_{i=1}^{m} w_i$.
2. Run the solving-every-problem-in-the-book problem with two students and the book problems W.
3. If the distribution of labor is the same for both students, then return "yes."
4. Otherwise, return "no."

4.1.2 Proof

Lemma 4.1. *The solving-every-problem-in-the-book problem is in NP.*

Proof. Given a set of m book problems, a set of students, an assignment of book problems to students, and a value k, one can check that every problem is assigned to exactly one student in $O(m)$. One can find the student with the minimum workload and verify that it is at least k in $O(n + m)$. Thus, in $O(n+m)$ one can verify that the assignment is valid and the minimum workload is at least k. $\square$

Lemma 4.2. *Number Partitioning reduces into the solving-every-problem-in-the-book problem in polynomial time.*

Proof. One take the input to Number Partitioning and directly feed it into the solving-every-problem-in-the-book algorithm, so it takes $O(1)$ to run an instruction saying to call the other algorithm, which you only call once. It takes $O(m)$ to sum up the problems the students are assigned, and $O(1)$ to compare the students' workloads to each other. Therefore, the conversion occurs in $O(m)$ $\square$

Lemma 4.3. *If the distribution of labor is the same for both students, then the Number Partitioning problem works.*

Proof. Clearly the problem times correspond to the positive integers. The set of problems assigned to one student is a subset of the positive integers. Since the solving-every-problem-in-the-book problem does not allow for any problems to be unassigned, if the total workload is $|W|$ then if the students have the same workload they each have a workload of $|W|/2$, which is the solution to the problem. $\square$

Lemma 4.4. *If the Number Partitioning problem works, then the distribution of labor is the same for both students.*

Proof. If the Number Partitioning problem works, then there are two equal sized disjoint subsets of the problem. If we wanted to give one student more work, we would have to take work away from the other student, thus reducing that student's workload below $|W|/2$ and thus reducing the minimum workload. Therefore, we do not want to give one student more work and we will return a distribution of labor that is the same for both students. $\square$

Theorem 4.5. *The solving-every-problem-in-the-book problem is NP-complete.*

Proof. By Lemma 4.1, it is in NP. By Lemmas 4.2, 4.3, and 4.4, it is NP-hard. Therefore, it is NP-complete. $\square$

4.2 Part B

4.2.1 Algorithm

Sort the problems descending by their time. Then assign the longest unassigned problem to the student with the least work until you are out of problems.

4.2.2 Runtime

Sorting is $O(m \log m)$. Then finding the student with the least work takes $O(n)$ time for each of m problems (this can probably be improved, but is polynomial as written). Thus we have a total time of $O(nm + m \log m) = O(nm)$

4.2.3 Proof of Approximation

In the best case scenario, you can divide the work evenly between all students because the assignment must sum to the total work, which results in $1/n$ per student. Otherwise since the total amount of work remains unchanged, increasing the amount of work done by any student results in a corresponding decrease in another student's work, and the value of the asignment goes down. Thus we have

$$T^* \leq \frac{1}{n} \sum t_j$$

where T^* represents the value of the optimal assignment. Moreover we can assume that

$$T^* \geq \min t_j$$

since given $n > m$ (more students than problems[1]) each student must be given a problem and the least value for a problem is just $\min t_j$

If there are at least as many problems as people, the minimum work load is at least the size of the smallest problem.

Having established these bounds we shall prove the approximation's validity

Proof. Consider the last problem assigned. We have $T_i - \min t_j \leq T_j$ for T_i the amount of work done by the last student who is assigned a problem. That is, before the last student is assigned a problem he has less work than anyone.

Since in particular our solution has each student with (where again T_i is the amount of work done by the student who is assigned the last problem)

$$T_j \geq T_i - \min t_j$$

then our solution has value at least $T_i - \min t_j$ and in particular

$$T^* \geq T_i - \min t_j$$

so summing this with the second inequality above we just have

$$2T^* \geq T_i$$

as desired. □

[1]If there are not more students than problems the optimal solution is trivial: give each of m students one problem. Thus we do not consider this case.