CSC373— Algorithm Design, Analysis, and Complexity — Spring 2016

Solutions for Tutorial Exercise 7: Approximation Algorithms

---

1. **Q6 on the CSC373 Final Exam, April 2012.** Given N items with weights $W = [w_1, w_2, \ldots, w_N]$, we wish to place each of these items into K bins. Multiple items can be placed in the same bin, however each bin has a maximum weight capacity of $C > 0$. So the sum of weights of the items in any single bin cannot exceed C. You can assume that all the quantities above are positive integers, and that $w_n \leq C$ for each n. Define $T = \sum_{n=1}^{N} w_n$ to be the total weight of all the items.

   We wish to minimize the number bins we use, K. Consider the greedy FirstFit algorithm which takes each item from the list $[w_1, w_2, \ldots, w_N]$ in turn (i.e., for n increasing from 1 to N), and places it into the first bin that can fit it, that is, without exceeding that bin's weight capacity.

   For example, consider $C = 10$, and $W = [2, 2, 7, 3, 8, 5, 2, 6, 2]$. Then FirstFit places the first two items in the first bin, the third item is placed in the second bin. The fourth item, with weight 3, is then added to the first bin, since that does not cause its capacity to be exceeded. This process continues, ending up with the bins 1 to 5 each containing a total weight of $[B_1, B_2, \ldots, B_5] = [2+2+3+2, 7+2, 8, 5, 6]$, respectively. Therefore, this FirstFit algorithm uses $K = 5$ bins for this case. (Note an optimal solution requires only $K^* = 4$ bins, e.g., with the first three bins each having a weight of 10.)

   (a) Define K^* to be the minimum number of bins required. Explain why $K^* \geq \text{ceil}(T/C)$. (Here $\text{ceil}(x)$ denotes the minimum integer k such that $k \geq x$.)

   (b) Prove that the FirstFit algorithm leaves at most one bin either half full or less. That is, with B_n denoting the total weight inside the n^{th} bin, after the FirstFit has run, then there is at most one j such that $B_j \in (0, C/2]$. For all other bins k, $1 \leq k \leq K$ and $k \neq j$, we must have $B_k > C/2$. (See the example above.)

   (c) Prove that the number of bins used by the FirstFit algorithm is never more than $\text{ceil}(2T/C)$.

   (d) Prove that FirstFit is a 2-approximation.

   Solutions Q1:

   1a At best we could fill each of $K_0 = \text{floor}(T/C)$ bins to capacity C, and then we would have a weight of $r = T - K_0 C$ left over, with $0 \leq r < C$. Another way to write this is

   $$T = C\ \text{floor}(T/C) + r, \text{ with } r \in [0, 1). \tag{1}$$

   If $r > 0$, we need one more bin, so $K = K_0 + 1$. In this case, it follows from (1) that $\text{ceil}(T/C) = K_0 + 1$, and the result follows.

   Othewise, if $r = 0$, we require only $K = K_0$ bins. In this case, it follows from 1) that $\text{ceil}(T/C) = K_0$, and the result follows for this case too.

   Since these two cases are exhaustive, the result follows.

   1b By contradiction. Suppose the two bins j and k each have $B_j, B_k \in (0, C/2]$, with $j \neq k$. WLOG we take $j < k$ (otherwise swap the labels j and k). Since $B_k > 0$ there must be a first item that was placed in bin k, suppose this item is the n^{th} item. It follows that $w_n \leq B_k \leq C/2$. But this contradicts the algorithm, since at the n^{th} stage the amount in bin j must have been at most $B_j \leq C/2$, and w_n would have fit within this bin.

   1c By part (b) we have the number of bins satisfies $(K-1)C/2 < T$. This implies $(K-1) > 2T/C$. The result follows since $2T/C + 1 \geq \text{ceil}(2T/C)$.

1d The algorithm requires $O(N\text{ceil}(2T/C))$ time, so is poly-time.

The results above show

$$0 < \text{ceil}(T/C) \leq K^* \leq K \leq \text{ceil}(2T/C). \tag{2}$$

Given

$$\text{ceil}(2T/C) \leq 2\text{ceil}(T/C), \tag{3}$$

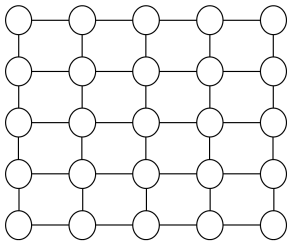
which we show below, it follows that

$$K \leq \text{ceil}(2T/C) \leq 2\text{ceil}(T/C) \leq 2K^*.$$

That is, $K \leq 2K^*$, so the algorithm is a 2-approximation.

Finally, to show (3), let $T/C = K_0 + r$, where $K_0 = \text{floor}(T/C)$ and $r \in [0, 1)$. Then $\text{ceil}(2T/C) = 2K_0 + \text{ceil}(2r)$, and $2\text{ceil}(T/C) = 2(K_0 + \text{ceil}(r))$. We see the result (3) would follow from $\text{ceil}(2r) \leq 2\text{ceil}(r)$. We can prove this with the following two cases: 1) for $r = 0$ both the left and right hand sides are 0, and this inequality is satisfied; and 2) for $r \in (0, 1)$, we have $\text{ceil}(2r) \leq 2\text{ceil}(r) = 2$ and the result follows for this case too.

2. **Q10, Chp 11, Kleinberg and Tardos.** Suppose you are given a weighted graph $G = (V, E, w)$ where G has the form of an $n \times n$ grid graph (see figure below). Assume the weights $w(v)$ are non-negative integers.



Prof. Jot proposes the following greedy algorithm for obtaining an approximate solution to the maximally weighted independent set problem for this type of graph:

[S] = wIndSet(V, E, w)
Initialize $F \leftarrow (V, E)$ and $S \leftarrow \{\ \}$.
While the graph F is not empty:
  Find a vertex u in F with the largest weight $w(u)$.
  $S \leftarrow S \cup \{u\}$
  Update F by deleting the vertex u and all its neighbouring vertices v (i.e., all vertices v with an edge (u, v) still in F), and delete all the edges ending at any of these deleted vertices.
End while
return S

(a) Show that the set S returned by `wIndSet` is an independent set for the graph G.

(b) Show that $w(S) = \sum_{v \in S} w(v)$ is at least $(1/4)w(S^*)$, where S^* is an independent set of G with the maximum possible weight $w(S^*)$.

Solution for Q2:

2a Add the following loop invariant to the end of the loop in the above algorithm:

> **LI:** S is an independent set of G, and the updated graph F does not contain any vertex $u \in S$ nor any neighbour vertex (with respect to G) of u.

This loop invariant can be proved by induction. The key is that whenever a vertex u is added to S, all remaining neighbours of u are deleted from F. The correctness of the result follows from the loop invariant and the fact that the algorithm must terminate in a finite number of steps. We skip the details.

2b Remember that to be a ρ-approximation we need to show the algorithm is poly-time. If, say, a heap is used to store the weights $w(v)$ (paired with v), the the algorithm runs in $O(|E| + |V| \log |V|)$ time.

To show the algorithm achieves an approximation ratio of 4, let S^* be a minimum weight independent set, and suppose S is the set produced by this algorithm. We build a "association" mapping, say $A(u)$, which maps each element $u \in S$ to a subset of its neighbours in S^* as follows.

At each stage of the algorithm, suppose $u \in S$ is selected during the next execution of the loop body, and F is the graph at the beginning of this execution of the loop body (i.e., u is a vertex of maximum weight in F). We define $N_F(u)$ to be the set of neighbours of u (not including u itself) in this graph F. At this stage of the algorithm, we have two cases to consider, either u is also in S^* or not. In the first case, we define $A(u) = \{u\}$. In the latter case, we set $A(u) = N_F(u) \cap S^*$.

We claim that in both cases

$$(N_F(u) \cup \{u\}) \cap S^* = A(u). \tag{4}$$

For the first case, namely $u \in S^*$, equation (4) follows since, by the independent set property, no neighbour of u can also be in S^*. The second case follows directly from the definition of $A(u)$.

Since the algorithm chose u from F, we must have $w(u) \geq w(v)$ for all remaining vertices $v \in F$, and hence for all $v \in N_F(u)$. Moreover, since the original graph G is of maximum degree 4, we know F must be at most degree 4. Therefore, for either case $u \in S^*$ or $u \notin S^*$, it follows that $|A(u)| \leq 4$ and

$$w(u) \geq (1/4) \sum_{v \in A(u)} w(v), \text{ for all } u \in S. \tag{5}$$

Note this bound is not necessarily true if we had used the neighbourhood of u in the original graph G to define $A(u)$ (i.e., $A(u) = S^* \cap N_G(u)$), since a previous iteration of the algorithm could have eliminated a vertex $z \in G$ which is a neighbour of u with $w(z) > w(u)$.

Define $U_A = \bigcup_{u \in S} A(u)$. By construction, we have $U_A \subseteq S^*$. Moreover, we the claim that

$$U_A = S^*. \tag{6}$$

We use contradiction to prove (6). Suppose $v \in S^*$ and $v \notin U_A$. Then, from (4), it must be the case that $v \notin N_F(u) \cup \{u\}$ for any $u \in S$. Therefore v cannot have been deleted from F when the algorithm terminates. But the algorithm only terminates for an empty graph F, which is a contradiction.

Finally, the association mapping $A(u)$ has the property that, for any $v \in S^*$, there can be at most one $u \in S$ with $v \in A(u)$. That is,

$$A(u_1) \cap A(u_2) = \emptyset, \text{ for any } u_1, u_2 \in S \text{ with } u_1 \neq u_2. \tag{7}$$

This disjointness property above follows from the definition of the algorithm, which guarantees that the sets $N_F(u)$ are mutually exclusive sets. That is, once u has be chosen, u and all its neighbours $N_F(u)$ are deleted from F. Clearly this deletion can only happen once, and the equation (7) follows from the definition of $A(u)$ (i.e., either $A(u) = \{u\}$ when $u \in S^*$, or $A(u) = N_F(u) \cap S^*$, otherwise).

Combining these results we find

$$\begin{aligned} w(S) &= \sum_{u \in S} w(u), \text{ by definition} \\ &\geq \sum_{u \in S} \left[(1/4) \sum_{v \in A(u)} w(v) \right], \text{ by (5)}, \\ &= (1/4) \sum_{v \in U_A} w(v), \text{ by (6) and (7)}, \\ &= (1/4) \sum_{v \in S^*} w(v), \text{ by (6)}. \end{aligned}$$

Therefore $w(S) \geq (1/4)w(S^*)$ and the algorithm is a 4-approximation for this maximization problem (or a (1/4)-approximation, depending on the choice of notation).

3. **Modified Q3 Chp 11 Kleiberg and Tardos.** Suppose you are given a list of N integers $L = [a_1, a_2, \ldots, a_N]$, and a positive integer C. The problem is to find a subset $S \subseteq \{1, 2, \ldots, N\}$ such that

$$T(S) = \sum_{i \in S} a_i \leq C, \tag{8}$$

and $T(S)$ is as large as possible.

(a) Consider the decision version of this problem:

Input: A list of positive integers $L = [a_1, a_2, \ldots, a_N]$, and two positive integers $c \leq C$.
Input Size: $|s| = N + b_{max}$, where b_{max} is the maximum number of bits needed to represent c, C or any of the integers a_i (i.e., you can assume the number of bits needed to represent an integer k is $\text{bits}(k) = \text{ceil}(\log_2(k)) + 2$).
Problem rPSS (range positive subset sum): Does there exist a subset $S \subseteq \{1, 2, \ldots, N\}$ such that $c \leq T(S) \leq C$ (where $T(S) = \sum_{i \in S} a_i$)?

Sketch a proof that **rPSS** is NP-Complete by making use of a reduction with the NP-complete problem **subsetSum**, described below.

Input: A list of N integers $L = [a_1, a_2, \ldots, a_N]$.
Input Size: $|s| = N + b_{max}$, where b_{max} is the maximum number of bits needed to represent any of the integers a_i.
Problem subsetSum: Does there exist a non-empty subset $S \subseteq \{1, 2, \ldots, N\}$ such that $T(S) = 0$?

(b) Prof. Jot proposes the following greedy algorithm for obtaining an approximate solution to the rPSS problem:

```
[S] = maxBoundedSetSum([a_1, ..., a_N], B)
Initialize S ← { }, T = 0
For i = 1, 2, ..., N:
    If T + a_i ≤ B:
        S ← S ∪ {i}
        T ← T + a_i
End for
return S
```

Show that Prof. Jot's algorithm is not a ρ-appoximation algorithm for any fixed value ρ.

Note that, since this is a maximization problem, there are two conventions for representing the approximation ratio. Either you can show, for any $\rho > 1$, there exists an example such that $T(S) < (1/\rho)T(S^*)$, or you can switch notation (effectively using $\rho' = 1/\rho$), in which case you need to show that, for any $\rho' \in (0, 1]$, there exists an example such that $T(S) < \rho' T(S^*)$.

(c) Describe a 2-approximation algorithm for this problem (i.e., $\rho = 2 = 1/\rho'$) that runs in $O(N\log(N))$ time.

Solution for Q3:

3a The **rPSS** is a decision problem and, whenever the decision is positive, has a suitable set S as a certificate. The certifier can check the conditions on S in poly-time. We omit the details. Therefore **rPSS** is in NP.

Next we sketch one way to show **subsetSum** $\leq_p$ **rPSS**. Together with the fact that **rPSS** is in NP, and **subsetSum** is NP-complete, it will then follow that **rPSS** is NP-complete.

Suppose the **subsetSum** problem is given $L = [a_1, a_2, \ldots, a_N]$ as an input list. We will sketch a reduction which uses N calls to **rPSS**.

One issue is that **rPSS** only accepts lists of positive integers, so one idea is to add a large positive integer, X say, to each of the items in the list L. That is, define

$$L' = [X + a_1, X + a_2, \ldots, X + a_N]. \tag{9}$$

However, given a set of indicies $S \subseteq \{1, 2, \ldots, N\}$, the sum $T'(S)$ over L' is

$$T'(S) = \sum_{i\in S}(X + a_i) = |S|X + \sum_{i\in S} a_i, \tag{10}$$

which has a contribution of $X|S|$ that depends on the number of elements in S.

Thinking a little further ahead, it would be useful to be able to deduce

$$T'(S) = kX \text{ if and only if } \sum_{i\in S} a_i = 0 \text{ and } |S| = k. \tag{11}$$

But note that we can arrange for (11) so long as X is large enough. In particular, any integer X such that $X > 2\sum_{i=1}^{N} |a_i|$ is sufficient to ensure (11). Moreover, it follows that $X > |a_i|$ for each i and hence L' in (9) is a list of positive integers.

With this set-up, which can be done in poly-time with respect to $|s|$, we then call **rPSS** a total of N-times, with input L' as in (9) and $c = C = kX$, for $k = 1, 2, \ldots, N$. It follows from (11) that the output of **subsetSum(L)** is true iff the output of **rPSS(L', kX, kX)** is true for at least one k with $1 \leq k \leq N$. Note that this is still a polynomial number of calls to **rPSS** with respect to the size of the input (since $|s| \geq N$). It therefore follows that **subsetSum** $\leq_p$ **rPSS**. (It is also possible to provide a somewhat different input that allows for a poly-time reduction that involves just one call to **rPSS**. I leave that to the reader.)

3b Let $\rho > 1$ be a given positive integer. Consider the input set $L = [1, \rho + 1]$, with the upper bound $C = \rho + 1$. Then the Prof. Jot's algorithm returns $S = \{1\}$, for which $T(S) = 1$, while the optimum solution is $S^* = \{2\}$, for which $T(S^*) = \rho + 1$. Therefore

$$1 = T(S) < \frac{1}{\rho}T(S^*) = \frac{\rho + 1}{\rho}, \tag{12}$$

so the algorithm is not a ρ-approximation for this value of ρ. Finally, since ρ was an arbitrary positive integer, this is true for any $\rho > 1$.

3c Consider the slightly modified algorithm:

```
[S] = maxBoundedSetSum([a_1, ..., a_N], B)
Initialize S ← { }, T = 0
For i = 1, 2, ..., N:
    If a_i ≤ B:
        If T + a_i ≤ B:
            S ← S ∪ {i}
            T ← T + a_i
        Else:
            If T < B/2:
                S ← {i}
            break
End for
return S
```

The algorithm runs in $O(N)$ time.

For the analysis, it is useful to first consider any item for which $a_i > B$. Such items cannot appear in any solution, and are simply discarded by the algorithm above. In the remainder of this proof we can therefore assume, without loss of generality, that $a_i \leq B$ for all i.

Given that $a_i \leq B$ for all i, there are now two general cases: 1) $\sum_{i=1}^{N} a_i = A \leq B$; and 2) $\sum_{i=1}^{N} a_i = A > B$. In the first case the alorithm above produces the optimum solution. In the second case, the algorithm above produces a set S such that $T(S) \geq B/2$. But note that, for any optimal solution S^*, it follows that $T(S^*) \leq B$. Therefore $T(S) \geq \frac{1}{2}T(S^*)$, and so the algorithm is a 2-approximation.