CS3230
Tutorial 11

1. Show that checking whether an undirected graph is 2-colorable is in P.

   Ans: Without loss of generality assume that the graph is connected (otherwise consider each component separately).

   Pick vertex v_1, and color it 1.

   Do the following until all vertices are coloured (or the algorithm outputs that the graph is not 2-colourable):

   (a) Pick an uncoloured vertex v, at least one of whose neighbour is coloured.
   (b) If all the coloured neighbours of v are coloured 1, then colour v 2.
   (c) Else If all the coloured neighbours of v are coloured 2, then colour v 1.
   (d) Else, output that graph is not 2 colourable.

   Note that the above algorithm is clearly polynomial time algorithm, as each iteration of the loop above is polynomial time executable and there are at most n iterations of the loop, where the graph has n vertices.

   Note that all the colours given by the above algorithm are forced based on v_1 being coloured 1 (see steps (b) and (c)). Thus if the algorithm outputs that the graph is not 2-colourable, then it is indeed not 2-colourable.

   On the other hand, if the graph is 2-colorable, then in each round the above algorithm will colour at least one new vertex correctly, and thus eventually colour all the vertices giving a 2-colouring of the graph.

2. Consider the following modification of the Knapsack approximation algorithm we did in class, where $r > 1$ is a constant given to you:

   Knapsack
   Inputs:
   (1) A set A of n objects (with corresponding weights and values, which are assumed to be positive integers).
   (2) A knapsack weight L, which is a positive integer.
   (3) An optimality factor r (a positive integer greater than one).
   Output: A subset A' of A, such that $weight(A') \leq L$, and $value(A')$ is at least $(1 - \frac{1}{r})*$ optimal.

1. Let $S_1, S_2, S_3 \ldots$, be all subsets of A which have at most r members.
2. For each S_j, such that $weight(S_j) \leq L$, let B_j be B' given by algorithm Select, when the inputs to Select are $B = A - S_j$ and $K = L - weight(S_j)$.
   (Think of B_j as choosing the remaining elements, when we have already decided to chose S_j).
   Output $A' = S_j \cup B_j$, which maximizes $value(S_j \cup B_j)$.

End

Note that the above algorithm runs in time polynomial in the size of the input, assuming r is a constant (the algorithm's time complexity is exponential in r).

Show that the above algorithm gives an output whose value is at least $(1 - \frac{1}{r})*$ optimal.

Hint: Use the lemma done in class to find how much the difference in value given by the optimal algorithm and the above algorithm could be, when you fix the most valuable r elements which belong to the optimal algorithm.

Ans: Note that the algorithm runs in polynomial time (though it runs in time exponential in r, r is a CONSTANT). Consider any optimal subset A'' of A.

Let S_j be a subset of r elements of A'' with highest value (if A'' has size at most r, then $S_j = A''$).

Thus we have (using the Lemma done in class),

$value(A') \geq value(S_j \cup B_j) \geq value(S_j) + [value(A'' - S_j) - \max(\{value(a) \mid a \in A'' - S_j\})]$.

Since, for all $a \in A'' - S_j$, for all $a' \in S_j$, $value(a) \leq value(a')$:

$\max(\{value(a) \mid a \in A'' - S_j\}) \leq \frac{value(S_j)}{r} \leq \frac{value(A'')}{r}$. Thus,

$value(A') \geq value(A'') - \frac{value(A'')}{r} \geq (1 - \frac{1}{r}) * value(A'')$. QED

3. Consider the following variation of the knapsack problem.

   Input:

   (a) n objects, $O_1, O_2, \ldots, O_n$, where the weight and value of object O_i is w_i and v_i respectively. Here w_i and v_i are both posiitive integers. Furthermore, it is given that, for all i, $v_i \leq c * n$. where c is a prefixed constant and n is the number of objects.
   (b) A knapsack capacity K.

   Output: A subset $S \subseteq \{1, 2, \ldots, n\}$ such that, $\sum_{i \in S} v_i$ is maximized subject to the constraint that $\sum_{i \in S} w_i \leq K$.

   Show that the above problem can be solved in time polynomial in n (note that c is a constant).

   Hint: Use a dynamic programming algorithm.

   Ans:

SpecialCaseKnapsack

Input:

(a) n objects $O_1, O_2, \ldots, O_n$. Weight and value of object O_i is w_i and v_i respectively, where w_i, v_i are positive integers.

(b) Knapsack capacity K.

Assumption about input: $v_i \leq c * n$, where c is a predetermined constant.

(* The following method gives an optimal answer for the knapsack problem, and runs in time polynomial in n, c and length of the binary representation of the weights. *)

(* We will construct arrays $A(i, v)$ and $B(i, v)$, where $0 \leq i \leq n$ and $0 \leq v \leq c * n^2$. *)

(* Intuitively, the value of $A(i, v)$ will be the minimum weight attainable when we choose some subset of $\{O_1, \ldots, O_i\}$ with total value v. This weight may be ∞ if no subset of $\{O_1, \ldots, O_i\}$ has value v. If $A(i, v) \neq \infty$, then $B(i, v)$ gives the corresponding subset of $\{O_1, \ldots, O_i\}$ which has weight $A(i, v)$ and value v. *)

(* We will use dynamic programming method to define $A(i, v)$ and $B(i, v)$. *)

1. Initially set $A(i, 0) = 0$ and $B(i, 0) = \emptyset$, for $0 \leq i \leq n$, and $A(0, v) = \infty$, $B(0, v) = \emptyset$, for $1 \leq v \leq c * n^2$.
2. For $i = 1$ to n do
   For $v = 1$ to $c * n^2$ do
   If $v \geq v_i$ and $A(i-1, v) > A(i-1, v - v_i) + w_i$,
   then let $A(i, v) = A(i-1, v - v_i) + w_i$, and $B(i, v) = B(i-1, v - v_i) \cup \{O_i\}$.
   Else let $A(i, v) = A(i-1, v)$ and $B(i, v) = B(i-1, v)$.
   EndFor
   EndFor
   Let v be the maximum value such that $A(n, v) \leq K$. Output, $B(n, v)$ as the optimal selection with value v.

End

Note that the algorithm runs in time polynomial in n, c and length of binary representation of the weights. Also, using the usual dynamic programming methods one can show that the above algorithm indeed solves the knapsack problem optimally.

4. In this question we give an approximation algorithm for the general knapsack problem based on the previous question. Let $r > 1$ be a constant integer. Intuitively, the algorithm below finds a solution for the knapsack problem, whose value is at least $(1 - \frac{1}{r})$ times optimal value.

   Approximate-Knapsack

   Input:

   (a) n objects, $O_1, O_2, \ldots, O_n$, where the weight and value of object O_i are w_i and v_i respectively. Here w_i and v_i are both posiitive integers. (b) A knapsack capacity K.

(* Assume without loss of generality that $w_i \leq K$ for each i. If not, we can clearly drop such objects from consideration. *)

1. Let $V = \max(\{v_i \mid 1 \leq i \leq n\})$.
   (* V denotes the maximum of the values of all the objects. *)
2. For $1 \leq i \leq n$, let $s_i = \lfloor \frac{v_i * n * r}{V} \rfloor$.
   (* Note that $0 \leq s_i \leq n * r$. *)
3. Using the algorithm done in Q3 of this tutorial, solve the following modified Knapsack problem optimally:
   The objects are $O_1, \ldots, O_n$, knapsack capacity is K, where the weight and value of O_i are w_i and s_i respectively (thus everything remains same except value is changed from v_i to s_i).
   (* Note that this can be done optimally and in polynomial time, since r is a constant. *)
4. Output the optimal set of objects as obtained for the modified problem in step 3.

End

Show that the above algorithm is a good approximation algorithm by showing the following parts.

(a) Show that $v_i - (s_i * \frac{V}{r*n}) \leq \frac{V}{r*n}$.

(b) Show that the value of the output given by the above algorithm differs from the value of optimal set by at most $\frac{V}{r}$.

(c) Show that the value of the output given by the above algorithm is at least $\frac{r-1}{r}$ times optimal.

Ans: (a) Since $(x - \lfloor \frac{x}{y} \rfloor * y) \leq y$, part (a) follows by definition of s_i.

(b) Suppose optimal set is OPT. Let V_{opt} be the value of the optimal set, and S_{opt} be the value of optimal set when one replaces v_i by $s_i * \frac{V}{r*n}$. Let V_{alg} and S_{alg} be the corresponding values for the set output by the algorithm.

Then $V_{alg} \geq S_{alg} \geq S_{opt}$,

$S_{opt} \geq V_{opt} - (\frac{V}{r*n})\text{card}(OPT)$ (using part (a)),

Part (b) now follows as $\text{card}(OPT)$ is at most n.

(c) As V_{opt} is at least V, we have that $(V_{opt} - V_{alg})/V_{opt} \leq (V/r)/V = 1/r$. Thus, $V_{alg}/V_{opt} \geq 1 - 1/r = (r-1)/r$.

5. The multiprocessor scheduling problem is given as follows:

   INPUT:

   (a) There are m processors.

   (b) There are n tasks, $a_0, a_1, \ldots, a_{n-1}$, with task length (time to do the task) $\ell(a_0), \ell(a_1) \ldots$. The tasks cannot be split into different processors.

   (c) a deadline D.

(d) A schedule is a partition of the tasks into m sets $S_0, S_1, \ldots, S_{m-1}$ such that $S_1 \cup S_2 \ldots \cup S_{m-1} = \{a_0, a_1, \ldots, a_{n-1}\}$. Time taken by this schedule is

$$\max(\{\sum_{a \in S_i} \ell(a) : 1 \leq i \leq m\})$$

That is, time taken by the schedule is the maximum load on any of the processors.

QUESTION: Is there a way to assign the tasks to the m processors such that the time taken by the schedule is at most D?

The multiprocessor scheduling problem is NP-complete. Thus the corresponding problem of finding an optimal schedule S, which minimizes the time taken by the schedule, is NP-hard. Consider the following approximation algorithm for Multiprocessor Scheduling. Intuitively the idea is to schedule the tasks in order of decreasing length (time taken to execute the task), where the tasks are assigned to the processors in rotating order.

Multi_Schedule (A, ℓ, m)
(* A is a set of n tasks. ℓ is an array, where $\ell(a)$ denotes the time taken by task a. m is the number of processors. *)
1. Sort all the tasks based on non-increasing order of $\ell(\cdot)$.
Say the sorted order is $a_0, a_1, \ldots, a_{n-1}$, where $\ell(a_i) \geq \ell(a_{i+1})$, for $i < n-1$.
2. For $0 \leq i < m$, let $S_i = \{a_j \mid j < n \text{ and } (j \bmod m) = i\}$
3. $S = (S_0, S_1, \ldots, S_{m-1})$ is the schedule for the m processors, where S_i is the set of tasks assigned to processor i.
End

Show that the above algorithm is a good approximation algorithm by showing that the completion time for the schedule generated by the above algorithm is no worse than twice the completion time of the optimal schedule.

Hint: Consider the load due to the longest task and the rest of the tasks.

Ans: The schedule generated by the algorithm has processor 0 as most loaded. Furthermore, the load on processor 0 is at most

$$\ell(a_0) + \frac{1}{m} \sum_{0<i<n} \ell(a_i)$$

This is so because removing a_0 from processor 0, would make it least loaded, as the $(r+1)$-th process assigned to processor 0 is of length no-greater than the r-th process assigned to any other processor.

On the other hand, the optimal schedule gives time of at least

$$\max(\ell(a_0), \frac{1}{m} \sum_{0<i<n} \ell(a_i))$$

This is so because process a_0 is assigned to some processor. Furthermore, the load due to rest of the processes a_i, $0 < i < n$, can at best be equally distributed. Thus, we have that the algorithm's solution is at most two times the optimal (here note that for any two positive numbers x and y, $\frac{x+y}{\max(x,y)} \leq 2$).