CSC373S Lecture 8

- Questions concerning the problem set.

1. It was asked if Dijkstra would work if all the edges were negative. It was pointed out that the problem is not well defined in that the cost to any node v can be made arbitrarily small by continuing to go around a negative cycle. But what if we ask about the least cost simple path in this case? This is equivalent to asking for the maximum cost path (by negating all edge costs so that they are now positive). This is an NP hard optimization problem and hence we do not expect any algorithm to be able to compute such paths efficiently. We will soon consider the problem of computing least cost simple paths in graphs that do not have any negative cycles.
2. In question 5, the algorithm initially given does *not* have a constant approximation ratio. I have corrected that algorithm to one that is still not optimal but does give a constant approximation ratio. You can submit an answer to bonus question 5b any time during the course.
3. The DP counting question. I meant the question to mean counting the number of optimal solutions. But it is OK if you are counting the number of (feasible) solutions.
4. Approximation bounds. As I indicated, optimization problems are often "NP-hard" which (according to well held beliefs) means that there does not exist a polynomial time algorithm that always (i.e. for every input instance) computes an optimal solution. When faced with NP hard optimization problems, there are various possibilities; namely,
   (a) We can look for an efficient algorithm that performs well or optimally on a restricted class of input instances. We already pointed out that the interval selection and (respectively) interval colouring are the graph MIS (resp. colouring) problems when restricted to interval graphsi. We showed that there are optimal (and very efficient) greedy algorithms for both interval selection and interval colouring.
   (b) We can look for an efficient algorithm that performs well or optimally on "most inputs" (e.g. say is optimal with high probability with respect to some given distribution on input instances).
   (c) We can look for an algorithm whose complexity is not polynomial but still is sufficiently fast for small inputs (e.g. having complexity say c^n for inputs of "size" n where $c > 1$ is a "small" constant).
   (d) We can look for an efficient algorithm that always produces a solution which provides a "good" approximation to that of an optimal solution.

   It is this last alternative (worst case approximation algorithms) that we will often consider in this course. For a profit maximization problem, we say that an algorithm $\mathcal{A}$ is a c-approximation algorithm if for every input instance I, Profit$[OPT(I)] \leq c \cdot$ Profit$[\mathcal{A}(I)]$. For a cost minimization problem, a c-approximation algorithm $\mathcal{A}$ is a c-approximation algorithm if for every input instance I, Cost$[\mathcal{A}(I)] \leq c \cdot$ Cost$[OPT(I)]$ The constant c is called the approximation ratio and we have expressed this ratio so that $c \geq 1$ for both maximization

and minimization problems. For maximization problems, many people express approximation ratios with $d \leq 1$ by saying Profit$[\mathcal{A}(I)] \geq d \cdot$Profit$[OPT(I)]$ for every input instance I. Clearly these are equivalent statements with $d = \frac{1}{c}$.

Aside: There is also a concept of an asymptotic approximation ratio which we will usually not consider in this course. In the case of a cost minimization problem, we say that algorithm $\mathcal{A}$ has an asymptotic approximation ratio of c if $limsup_{Cost[OPT(I) \to \infty} \frac{Cost[OPT(I)]}{Cost[\mathcal{A}(I)]} \leq c$.

We saw two different styles of proving an upper bound on the approximation ratio of an algorithm. Namely, we used a charging argument to show that the SPT interval selection algorithm is a 2-approximation algorithm; that is, the algorithm has an approximation ratio ≤ 2. (The same type of charging argument will work for question 2 in the assignment.) In fact, this ratio is a tight bound in the sense that there is an input instance I such that Profit$[OPT(I)] = 2 \cdot$ Profit $[SPT(I)]$. This is what I would call an "inapproximation" result (showing a limitation on the given algorithm). Often when we find one "bad" input instance I, we can find an infinite set of such instances.

We also gave an inpapproximation result for the m machine makespan cost problem showing that the online greedy algorithm has approximation ratio $2 - \frac{1}{m}$ and this ratio is also tight. The positive result (i.e. that the ratio is $\leq 2 - \frac{1}{m}$ was derived by observing the cost of any algorithm in terms of the average per machine processing time of the jobs and the largest processing time of any job, and then relating the cost of the greedy algorithm to those necessary costs.

More generally, for some problems we may be able to show that every algorithm within a class of algorithms (like greedy algorithm suitably formalized) have such an inapproximation limitation. For greedy algorithms, I claim this is the case for the weighted interval selection problem and for the knapsack problem. And for some problems assuming our standard complexity assumptions, we can argue that every polynomial time algorithm has such an inapproximation limitation. This is the situation for problems like graph colouring (for arbitrary graphs).

- We return to the {0,1} knapsack problem (where we usually just say knapsack problem). Recall that an input instance consists of n items $\{(w_i, v_i)\}$ and a knapsack weight bound W.

We already saw one incorrect DP algorithm, incorrect in that it can produce solutions that are not allowed. I should also note that the most natural (to me) greedy algorithm fails to provide any constant approximation; that is, sort so that $v_1/w_1 \geq v_2/w_2 \ldots \geq v_n/w_n$ and then accept greedily (i.e. take an item as long as it fits). Can you think of a set of bad examples?

Here is one way to solve the {0,1} knapsack problem. We define the semantic array $S'(i,w)$ = largest profit obtainable by a feasible solution $T \subseteq \{1,\ldots i\} : \sum_{i\in T} w_i \leq w$. Now we can define an appropriate computational array $S(i,w)$ which can be proved correct by induction on i. Namely, we define :
$S(i,w) = 0$ if $i = 0$ for all w and $S(i,w) = \max\{A,B\}$ for $i > 0$ where $A = S(i-1,w)$ and $B = v_i + S(i-1, w-w_i)$ if $w_i \leq w$ and 0 otherwise.

As already suggested , one proves $S'(i,w) = S(i,w)$ by induction on i.

- Lets consider the time complexity (assuming memorization is used to efficiently implement the recursion or an iterative implementation is used. There are nW entries of S that have to be computed and assuming all previous entries have been computed, computing any $S(i,w)$ take $O(1)$ time for a total time complexity of $O(nW)$. This is called a pseudo polynomial time algorithm since W might be large (i.e if the w_i and W are n bits then the encoding of the input uses $O(n^2)$ bits and the complexity is $O(n2^n)$. But if all w_i are small (or if W is small), say $O(logn)$ bits, then the alg is running in polynomial time.

- There is an alternative DP for knapsack which is polynomial time when all values are small. This in turn can be used to provide arbitrarily good approximations for every knapsack instance. Now instead of determining the maximum value for a given weight w, we want to determine the minimum weight to obtain a given value v.

  That is, we define a semantic array $M'(i,v)$ = the minimum weight w such that there is a solution $T \subseteq \{1,\ldots,i\}$ and $\sum_{j\in T} v_j \geq v$. The desired optimum value is the maximum v such that $M'(n,v) \leq W$. We now have to indicate how to compute each $M'(i,v)$ by providing an appropriate recursive array M.

  To initialize, we define $M(0,v) = \infty$ for all $v > 0$ which indicates that there is no solution if we do now have any items and define $M(0,v) = 0$ for all $v < 0$. For $i > 0$, we have
  $M((i,v) = \min\{A,B\}$ where $A = M(i-1,v)$ and $B = w_i + M(i-1, v-v_i)$.

  One benefit of this method is that the complexity is now $O(nV)$ where $V = \sum_i \{v_i\}$ which of course is useful when V is small (compared to W). But more important is how this algorithm can be used as the basis for an efficient approximation algorithm for all input instances. The basic idea is relatively simple: the high order bits/digits of the values $\{v_i\}$ can determine an approximate solution (disregarding the low order bits). The fewer high order bits we use, the faster the algorithm but the worse the approximation. The goal is to scale the values in terms of a parameter ϵ so that a $(1+\epsilon)$ approximation is obtained with time complexity polynomial in n and $\frac{1}{\epsilon}$. The details are given in the KT text (section 11.8).