CSC373S Lecture 9

- Last time we concluded with a DP algorithm for the knapsack problem having complexity $O(nV)$ where $V = \sum_{i=1}^{n} v_i$.

  We remarked that by "scaling" the $\{v_i\}$ values, this algorithm can be used to derive a $(1+\epsilon)$ approximation with time complexity polynomial in n (the size of the input representation) and $\frac{1}{\epsilon}$. SEe KT text, section 11.8

  An algorithm achieving a $(1+\epsilon)$ approximation with time complexity polynomial in n (the size of the input representation) and $\frac{1}{\epsilon}$ is said to be an FPTAS algorithm (standing for *fully polynomial time approximation scheme*). An algorithm which achieves a $(1+\epsilon)$ approximation with time complexity polynomial in n but possibly exponential or worse in $\frac{1}{\epsilon}$ is called a PTAS algorithm.

  iIn term of computing optimal solutions, all "NP complete optimizaton problems" (i.e. optimization problems corresponding to NP complete decision problems) can be viewed (up to polynomial time) as a single class of problems. But in the world of approximation algorithms, this single class splits into many classes of approximation guarantees. Up to our believed complexity assumptions there are:

  1. problems with an FPTAS
  2. problems with a PTAS but (given complexity assumption) no FPTAS
  3. problems with with some $c > 1$ approximation but no PTAS
  4. problems with an $\Omega(logn)$ approximation but no constant approximation
  5. problems with no $n^{1-\epsilon}$ approximation for any $\epsilon > 0$.

  Here n stands for some input size parameter (e.g. number of nodes in a graph)

- We now consider a dynamic programming algorithm which has a somewhat different style in that the proof of correctness (showing the equivalence between the semantic and computationally defined arrays) is not an induction on the number of input items in the solution but is based on some other parameter of the solution.

  Namely, lets consider the single source least cost paths problem which is efficiently solved for graphs in which all edge costs are non-negative. The least cost paths problem is still well defined as long as there are no negative cycles; that is, the least cost path is a simple path.

  We consider a directed graph $G = (V, E)$ with edge costs $c : E \to \Re$ assuming that there are no negative cycles although there can be negative edges. (We will later see how to detect negative cycles.) So since there are no negative cycles we can assume that the problem is computing the least cost *simple* path between each pair of vertices. As before, we consider the nature of an optimal solution and how it is composed of optimal solutions to "subproblems". Consider an optimal simple path π from source s to some node v. This path could be just an edge but if the path π

has length greater than 1, then there is some node u which immediately proceeds v in π. If π is an optimal path to v, then the path leading to u must also be an optimal path. We are led to define the following semantic array: $C'[i, v]$ = the minimum cost of a simple path with path length at most i from source s to node v. (If there is no such path then this cost is ∞.) The desired answer is then the single dimensional array derived by setting $i = n$. We then have the following recursively defined computational array:

$C[0, v] = 0$ if $v = s$ and ∞ otherwise. $C[i, v] = \min\{A, B\}$ where $A = C[i-1, v]$ and $B = \min_{u:(u,v) \in E} C(i-1, u) + c(u, v)$.

IMPORTANT NOTE: How can we know that we are only obtaining the best *simple* path cost? In the proof that the arrays C' and C are the same, when considering case B, how do we know that $C[i, v]$ corresponds to a simple path? The path to u could be simple but what if v is on the path to u? If this were the case, then there would be a cycle from v to v which could be removed (since there are no negative cycles) and this would shorten the path length and can only decrease the path cost. So that would show that the path corresponding to case A is at least as good.

The time complexity of this algorithm is $O(n^3)$ as there are $O(n^2)$ array entries and each entry requires $O(n)$ steps assuming all previous entries have already been computed.

Note: This presentation is slightly different than in the KT text.

- Suppose we wanted to compute the all pairs least cost problem (i.e. the least cost path from u to v for all vertices u, v in V). Here again we are assuming that we can have negative edges but not negative cycles. We could simply run the above algorithm $n = |V|$ times choosing each $v \in V$ as the source node. This would then have time complexity $O(n^4)$. There is a variant of this method utilizing an additional divide and conquer idea to achieve complexity $O(n^3 \log n)$. Namely, we need to define a semantic array $C'[k, u, v]$ = minimum cost of a simple path with path length at most 2^k from source s to node v.

  We present here an alternative DP for the all pairs problem which has yet another style. Let us assume (wlg = without loss of generality) that $V = \{1, 2, \ldots, n\}$. We now define the following semantic array: $D'[i, j, k]$ = the cost of an optimal simple path π from i to j such that all internal path nodes in π are in $\{1, 2, \ldots, k\}$. The desired answer is the array with $k = n$.

  For the computational array, the base case is $D[i, j, 0] = c(i, j)$ if $(i, j) \in E$, $D[i, i, 0] = 0$ and $D[i, j, 0] = \infty$ if $i \neq j$ and $(i, j) \notin E$. For $k \geq 1$, $D[i, j, k] = \min\{A, B\}$ where $A = D[i, j, k-1]$ and $B = D[i, k, k-1] + D[k, j, k-1]$.

That is, the choice A refers to the optimal path not using node k and choice B using node k. The time complexity of this algorithm is $O(n^3)$.

- This last DP for the all pairs least cost problem is different than the previous DPs in the sense that the previous DPs only made one recursive call (for each choice A,B, ..). Here in choice B, we make two recursive calls. There are a number of other prominent DPs that make more than one recursive call for each "choice". In section 6.5, the text presents a DP for a simplified RNA structure problem. The KT text refers to this type of DP as "DP over Intervals". This example and other similar DP examples (e.g. optimal binary search trees, CFL recognition and parsing, polygon triangularization, iterated matrix multiplication) are essentially finding optimal "parse trees" for a given problem. In terms of the KT text, the "choice" to be made (in determining the relevant subproblems) is how to divide the interval into (two) subintervals.

  Lets consider *the iterated matrix multiplication problem* (also called the chain multiplication problem) which is often used as a canonical example of this kind of DP. Consider a sequence of matrices (over some ring or field F, say the reals) $M_1, M_2, \ldots, M_n$. Assume that we have some known algorithm for computing the product of two matrices. That is, given matrices $A_{p\times q}$ and $B_{q\times r}$, we compute the product $A * B$ in some number of scalar (i.e. over the underlying ring or field F) operations. Using classical matrix multiplication, such a matrix product takes pqr scalar multiplications and $p(q-1)r$ scalar additions. To simplify the discussion, lets say that the cost of such a pairwise matrix product is pqr.

  Aside: When we discuss divide and conquer algorithms, we will show that for $p = q = r = n$, one can reduce the scalar costs to n^α for $\alpha < 3$.

  We wish to have an algorithm for the iterated matrix product $(M_1 * M_2 \ldots * M_n)$ using the known algorithm for multiplying two matrices. We know that the matrix multiplication operator $*$ is associative; that is, $(M_1(M_2 * M_3)) = ((M_1 * M_2) * M_3)$. So the order in which we apply the known algorithm for two matrices (i.e. the way we parenthesize the iterated product) does not change the resulting matrix product. However, the way we carry out the pairwise products does change the number of scalar operations. Consider matrices M_1, M_2, M_3 with dimensions (respectively) $(5 \times 10), (10 \times 100, (100 \times 50)$, then
  $(M_1 * M_2) * M_3$ costs $5000 + 25000 = 30000$ scaler ops and $M_1 * (M_2 * M_3)$ costs $50000 + 2500 = 52500$ scalar ops.

  For this problem, the input items are the matrix dimensions (and not their entries). Lets say M_i has dimension $d_{i-1} \times d_i$ for $1 \leq i \leq n$. We view the order of pairwise matrix mults as a parse tree where the root of the tree is the last pairwise multiplication. That is, the last pairwise multiplication is
  $(M_1 * M_2 \ldots M_i) * (M_{i+1} * \ldots M_n)$.

Once we determine the best i, we then have to solve the subproblems for the subproblems $(M_1 * M_2 \ldots M_i)$ and $(M_{i+1} * \ldots M_n)$. This suggests the following semantic array: $C'[i,j]$ = the optimal cost to compute $M_i M_{i+1} \ldots * M_j$ for $1 \leq i \leq j \leq n$. The corresponding computational array is:
$C[i,i] = 0$ and for $i < j$ $C[i,j] = min_{k:i \leq k \leq j-1}\{C[i,k] + C[k+1,j] + d_{i-1}d_k d_j\}$