CSC2420 Fall 2012: Algorithm Design, Analysis and Theory
An introductory (i.e. foundational) level graduate course.

Allan Borodin

October 4, 2012

Lecture 4

1. We begin where we left off with the DP (+ scaling) based algorithm for deriving a FPTAS for the knapsack problem.
2. DP based approach for a DP based algorithm for deriving a PTAS (poly in n and m, and exponential in $\frac{1}{\epsilon}$) for makespan on identical machines.
3. Some attempts to formally model DP algorithms
4. Introducing local search

Dynamic programming and scaling

- We have previously seen that with some use of brute force and greediness, we can achieve PTAS algorithms for the identical machines makespan (polynomial in the number n of jobs but exponential in the number m of machines) and knapsack problems.
- We now consider how dynamic programming (DP) can be used to acheive
  1. An FPTAS for the knapsack problem
  2. A PTAS for the makespan problem which is polynomial in m and n,
- To achieve these improved bounds we will combine dynamic programming with the idea of scaling inputs.

An FPTAS for the knapsack problem

Let the input items be $I_1, \ldots, I_n$ (in any order) with $I_k = (v_k, s_k)$. The idea for the knapsack FPTAS begins with a "pseudo polynomial" time DP for the problem, namely an algorithm that is polynomial in the numeric values v_j (rather than the encoded length $|v_j|$) of the input values.

Define $S[j, v]$ = the minimum size s needed to achieve a profit of at least v using only inputs $I_1, \ldots I_j$; this is defined to ∞ if there is no way to achieve this profit using only these inputs.

This is the essense of DP algorithms; namely, defining an approriate generalization of the problem (which we usually give in the form of an array) such that

1. the desired result can be easily obtained from ths array
2. each entry of the array can be easily computed given "pevious entries"

How to compute the array $S[j, v]$ and why is this sufficient

- The value of an optimal solution is $max\{v|S[n, v]$ is finite$\}$.
- We have the following equivalent recursive definition that shows how to compute the entries of $S[j, v]$ for $0 \leq j \leq n$ and $v \leq \sum_{j=1}^{n} v_j$.
  1. Basis: $S[0, v] = \infty$ for all $v > 0$ and $S[0, 0] = 0$
  2. Induction: $S[j, v] = \min\{A, B\}$ where $A = S[j-1, v]$ and $B = S[j-1, \max\{v - v_j, 0\}] + s_j$.
- It should be clear that while we are computing these values that we can at the same time be computing a solution corresponding to each entry in the array.
- For efficiency one usually computes these entries iteratively but one could use a recursive program with *memoization*.
- The running time is $O(n, V)$ where $V = \sum_{j=1}^{n} v_j$.
- Finally, to obtain the FPTAS the idea (due to Ibarra and Kim [1975]) is simply that the high order bits/digits of the item values give a good approximation to the true value of any solution and scaling these values up to the high order bits does not change feasibility.

The better PTAS for makespan

- We can think of m as being a parameter of the input instance and now we want an algorithm whose run time is poly in m, n for any fixed $\epsilon = 1/s$.
- The algorithm's run time is exponential in $\frac{1}{\epsilon^2}$.
- We will need a combination of paradigms and techniques to achieve this PTAS; namely, DP and scaling (but less obvious than for the knapsack scaling) and binary search.

The high level idea of the makespan PTAS

- Let T be a candidate for an achievable makespan value. Depending on T and the ϵ required, we will scale down "large" (i.e. if $p_i \geq T/s = T \cdot \epsilon$) to the largest multiple of T/s^2 so that there are only $d = s^2$ values for scaled values of the large jobs.
- When there are only a fixed number d of job sizes, we can use DP to test (and find) in time $O(n^{2d})$ if there is a soluton that achieves makespan T.
- If there is such a solution then small jobs can be greedily scheduled without increasing the makespan too much.
- We use binary search to find the best T.

The optimal DP for a fixed number of job values

- Let $z_1, \ldots, z_d$ be the d different job sizes and let $n = \sum n_i$ be the total number of jobs with n_i being the number of jobs of size z_i.
- $M[x_1, \ldots, x_d] =$ the minimum number of machines needed to schedule x_i jobs having size z_i within makespan T.
- The n jobs can be scheduled within makespan T iff $M[n_1, , n_d]$ is at most m.

Computing $M[x_1, \ldots, x_d]$

- Clearly $M[0, \ldots, 0] = 0$ for the base case.
- Let $V = \{(v_1, , v_d) | \sum_i v_i z_i \leq T\}$ be the set of configurations that can complete on one machine within makespan T; that is, scheduling all of the v_i jobs with size z_i on one machine does not exceed the target makespan T.
- $M[x_1, \ldots, x_d] = 1 + \min_{(v_1, \ldots, v_d) \in V : v_i \leq x_i} M[x_1 - v_1, \ldots, x_d - v_d]$
- There are at most n^d array elements and each entry uses approximately n^d time to compute (given previous entries) so that the total time is $O(n^{2d})$.
- Must any (say DP) algorithm be exponential in d?

Large jobs and scaling (not worrying about any integrality issues)

- A job is large if $p_i \geq T/s = T \cdot \epsilon$
- Scale down large jobs to have size $\tilde{p}_i =$ largest multiple of $T/(s^2)$
- $p_i - \tilde{p}_i \leq T/(s^2)$
- There are at most $d = s^2$ job sizes $\tilde{p}$
- There can be at most s large jobs on any machine not exceeding target makespan T.

Taking care of the small jobs and accounting for the scaling down

- We now wish to add in the small jobs with sizes less than T/s. We continue to try to add small jobs as long as some machine does not exceed the target makespan T. If this is not possible, then makespan T is not possible.
- If we can add in all the small jobs then to account for the scaling we note that each of the at most s large jobs were scaled down by at at most $T/(s^2)$ so this only increases the makespan to $(1 + 1/s)T$.

Models for DP algorithms

- We previously presented priority algorithms as a model for greedy and greedy-like algorithms and then considered extending that framework to the stack priority model which can be viewed as a model for a basic class of *primal dual with reverse delete* algorithms (alternatively, a basic class of *local ratio* algorithms). (We will return to primnal dual algorithms later.)
- Can we formulate a model for dynamic programming (DP) algorithms that
  - Captures many/most known DP algorithms
  - Is amenable to analysis in terms of what can and cannot be done (efficiently) by such algorithms?
- Is this worth doing?

Why formalize? : The attacks against formalization

- " .... trying to define what may be indefinable. .... I shall not today attempt further to define the kinds of material...But *I know it when I see it ...* " U.S. Supreme Court Justice Potter Stewart in discussing obscenity, 1964.
- Samuel Johnson (1709-1784): All theory is against freedom of the will; all experience for it.
- Anonymous theoretican: What you can do by DP in polynomial time is what you can do in polynomial time (and hence not currently tractable).
- Anonymous students: Who cares what kind of an algorithm it is?

Bellman [1957] arguing against defining DP (in the spirit of Samuel Johnson)

We have purposely left the description a little vague, since it is the spirit of the approach to these processes that is significant, rather than a letter of some rigid formulation.

It is extremely important to realize that one can neither axiomatize mathematical formulation [1] nor legislate away ingenuity.

In some problems, the state variables and the transformations are forced upon us; in others, there is a choice in these matters and the analytic solution stands or falls upon this choice; in still others, the state variables and sometimes the transformations must be artificially constructed. Experience alone, combined with often laborious trial and error, will yield suitable formulations of involved processes.

[1] I take this to mean that one cannot axiomatize or completely formalize an intuitive concept.

The pBT model: An attempt to model some simple DP (and backtracking) algorithms

- In an extension of the priority framework, Alekhnovich et al [2011] consider the pBT model (for *prioritized branching tree* or *prioritized backtrack* where upon considering an input item, the algorithm can branch on different possible decisions. The algorithm can also terminate branches whenever it wishes.
- For search problems, the goal is to have a branch that produces a feasible solution if one exists, and for optimization problems the solution having the best approximation ratio is chosen. (Aside: it would have been better to just have non deterministic branching instead of branching on decisions.)
- The complexity of such an algorithm is size (or maximum "width") or the time in say a depth first search of the pBT tree.
- The pBT model can capture DPs where the implicit induction is on the number of items as in the interval scheduling and knapsack DPs.

Some pBT results

- In the pBT model, we can optimally solve one machine interval scheduling with fixed order width n (the number of intervals) using the standard DP, and $\Omega(n)$ width is required for any adaptive order pBT that optimally solves the problem. Furthermore for any fixed m, the width required for optimally solving the m machine problem is $\Omega(n^m)$ which can be achieved again using DP.
- In the pBT model, we have the following result for the knapsack problem: We can obtain a $(1+\epsilon)$-approximation with width $O(\frac{1}{\epsilon^2})$ (based on the Lawler adaption of the Ibarra and Kim FPTAS) and any adaptive order pBT algorithm that achieves a $(1+\epsilon)$-approximation requires width $\Omega(\frac{1}{\epsilon^{3.17}})$ and width $\binom{n/2}{n/4} = \Omega(2^{n/2}/\sqrt{n})$ for optimality. The lower bounds hold even for the Subset-Sum problem.
- Chvátal [1980] established an exponential time bound for the knapsack problem with respect to a model that captures a style of branch and bound algorithms. Similar attempts to formalize some branch and bound methods were obtained by Chvátal [1977] for the MIS problem and by McDiarmid [1979] for the graph colouring.

The pBP model: a more ambitious DP model

- The pBP (for *prioritized branching program*) model extends the pBT model by combining merging with branching so that the underlying structure of a pBP algorithm is a rooted DAG and not a rooted tree.
- The semantics are a little involved but the idea is meant to better capture memoization which is central to DP algorithms (in the sense of distinguishing them from divide and conquer algorithms).
- In the pBP model, there is an optimal $O(n^3)$ width algorithm for solving the shortest path problem when there are negative weights but not negative cycles. If the input graph has negative cycles the algorithm will output an arbitrary set of edges. Here the input items are .... In contrast, any pBT algorithm would require exponential width to solve the promise version of the shortest path problem on some instance (which could be a graph with negative cycles).
- For the bipartite matching problem where the input items are edges, any pBP algorithm requires exponential width. A challenge is to prove such a result when the input items are vertices.
- There is an optimal max flow algorithm for bipartite matching.

Combinatorial DP Programs

Finally we mention another model by Bompadre [2010] that also captures a limited class of DP algorithms. This model is incomparable with the pBT and pBP models. There are a number of positive and negative results derived by Bompadre. We will hopefully return to consider this model as well as another new model now in the editorial process.

Local search: the other conceptually simplest approach

We now begin a discussion of the other (than greedy) conceptually simplest search/optimization algorithm, namely localsearch.

The vanilla local search paradigm

Initialize S
While there is a "better" solution S' in "$Nbhd(S)$"
$S := S'$
EndWhile

If and when the algorithm terminates, the algorithm has computed a *local optimum*. To make this a precise algorithmic model, we have to say:

1. How are we allowed to choose an initial solution?
2. What consititutes a local neighbourhood $Nbhd(S)$?
3. What do we mean by "better"?

Answering these questions (especially as to defining local neighbourhood) will often be quite problem specific.

Towards a precise definition for local search

- We clearly want the initial solution to be efficiently computed and to that end (so as to be precise) we can (for example) say that the initial solution is a random solution, or a greedy solution or adversarially chosen. Of course, in practice we can use any efficiently computed solution and this is done in practice.
- We want the local neighbourhood $Nbhd(S)$ to be such that we can efficiently search for a "better" solution (if one exists).
  1. In many problems, a solution S is a subset of the input items or equivalently a $\{0,1\}$ vector, and in this case we often define the $Nbhd(S) = \{S' | d_H(S, S') \leq k\}$ for some small k where $d_H(S, S')$ is the Hamming distance.
  2. More generally whenever a solution is a vector over a small domain D, we can use Hamming distance to define a local neighbourhood. Hamming distance k implies that $Nbhd(S)$ can be searched in at most time $|D|^k$.
  3. We can view Ford Fulkerson flow algorithms as local search algorithms where the neighbourhood of a solution S (i.e. a flow) are flows obtained by adding an augmenting path flow. This is an exponential size neighbourhood but one that can be searched efficiently.

What does "better" solution mean? Oblivious and non-oblivious local search

- For a search problem, we would generally have a non-feasible initial solution and "better" can then mean"closer" to being feasible.
- For an optimization problem it usually means being an improved solution which respect to the given objective. For reasons I cannot understand, this has been termed *oblivious* local search.
- For some applications, it turns out that rather than searching to improve the given objective function, we search for a solution in the local neighbourhood that improves a related potentental function and this has been termed non-oblivious local search.
- And in searching for an improved solution, we may want an arbitrary improved solution, a random improved solution, or the best improved solution in the local neighbourhood.
- For efficiency we may insist that there is a "sufficient" improvement.

The jump local search algorithm for makespan on identical machines

- Start with any initial solution
- It doesnt matter how jobs are arranged on a machine so the algorithm can move any job (on a "critical machine" defining the current makespan value) if that move will "improve things".That is, we will say that a (successful) jump move is one that moves any job to another machine so that either the makespan is decreased or the number of machines determining the current makespan is decreased.
- Note that technically speaking this is a non-oblivious local search as we may not be decreasing the current makepsan in moving to a better solution.
- Finn and Horowitz[1979] prove:that the "localitygap" for this local search algorithm is is$2 - \frac{2}{m+1}$. That is, this is the worst case ratio for some local optimum compared to the global optimum.
- To bound the number of iterations, in moving J_k, it should be moved to the machine having the current minimum load.