CSC2420: Algorithm Design, Analysis and Theory Fall 2017

Allan Borodin and Nisarg Shah

September 27, 2017

Lecture 3

Announcements:

- First few questions for assignment 1 have been posted. We will set a due date after posting remaining questions this week (hopefully).

Todays ambitious agenda:

- Finish (at least for now) discussion of greedy and greedy like algorithms. The priority stack model as in Bar-Noy et al [2001], Akcoglu et al [2000], Ye and Borodin [2011]. Greedy interval coloring and coloring of chordal graphs. One machine and m-machine weighted interval scheduling.
- Dynamic programming and some applications
  Note: We will start with some standard examples that would be seen in an undergrad course)
  1. Revisiting one machine and m-machine weighted interval scheduling
  2. The knapsack problem and an FPTAS
  3. DP for the makespan problem for identical maxchines
  4. The priority branching tree (pBT) model as a model for simple DP and parallel greedy algorithms.

Priority Stack Algorithms

- For packing problems, instead of immediate permanent acceptances, in the first phase of a priority stack algorithm, items (that have not been immediately rejected) can be placed on a stack. After all items have been considered (in the first phase), a second phase consists of popping the stack so as to insure feasibility. That is, while popping the stack, the item becomes permanently accepted if it can be feasibly added to the current set of permanently accepted items; otherwise it is rejected. Within this priority stack model (which models a class of primal dual with reverse delete algorithms and a class of local ratio algorithms), the weighted interval selection problem can be computed optimally.
- For covering problems (such as min weight set cover and min weight Steiner tree), the popping stage is to insure the minimality of the solution; that is, while popping item I from the stack, if the current set of permanently accepted items plus the items still on the stack already consitute a solution then I is deleted and otherwise it becomes a permanently accepted item.

Chordal graphs and perfect elimination orderings

An interval graph is an example of a chordal graph. There are a number of equivalent definitions for chordal graphs, the standard one being that there are no induced cycles of length greater than 3.

We shall use the characterization that a graph $G = (V, E)$ is chordal iff there is an ordering of the vertices $v_1, \ldots, v_n$ such that for all i, $Nbdh(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ is a clique. Such an ordering is called a perfect elimination ordering (PEO).

It is easy to see that the interval graph induced by interval intersection has a PEO (and hence is chordal) by ordering the intervals such that $f_1 \leq f_2 \ldots \leq f_n$. Using this ordering we know that there is a greedy (i.e. priority) algorithm that optimally selects a maximum size set of non intersecting intervals. The same algorithm (and proof by charging argument) using a PEO for any chordal graph optimally solves the unweighted MIS problem. Bar-Noy et al [2001] provide an optimal solution for weighted interval scheduling which immediately generalizes to chordal graphs.

The optimal priority stack algorithm for the weighted max independent set problem (WMIS) in chordal graphs

```
$Stack$ := $\varnothing$          % $Stack$ is the set of items on stack
Sort nodes as in a PEO
**For** $i = 1..n$
    $C_i$ := nodes on stack that are adjacent to $v_i$
    **If** $w(v_i) > w(C_i)$ **then** push $v_i$ onto stack, **else** reject
**End For**
$S$ := $\varnothing$              % $S$ will be the set of accepted nodes
**While** $Stack \neq \varnothing$
    Pop next node $v$ from $Stack$
    **If** $v$ is not adjacent to any node in $S$, **then** $S$ :=$S \cup \{v\}$
**End While**
```

Figure : Priority stack algorithm for chordal WMIS

Sketch of the WMIS chordal graph result

Let ALG (resp. OPT) denote the nodes in the solution of the algorithm (resp. of an optimal solution). Let S be the contents of the stack at the end of the push phase and let S_i be the contents of the stack as we are about to consider v_i.

Define $\tilde{w}(v_i) = w(v_i) - \sum_{v_j \in S_i \cap Nbhd(v_i)} \tilde{w}(v_j)$. Then we push v_i on the stack iff $\tilde{w}(v_i) > 0$.

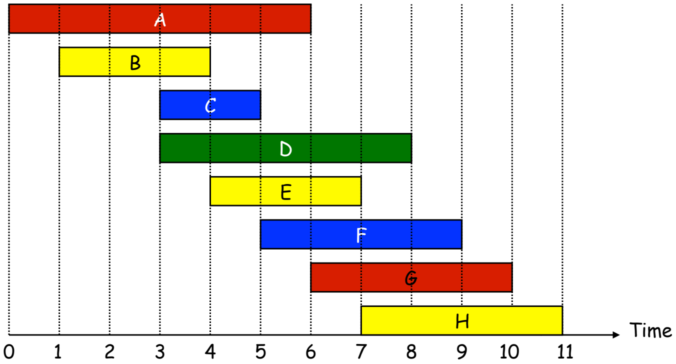
Fact: $\sum_{v_t \in ALG} w(v_t) = \sum_{v_t \in S} \tilde{w}(v_t)$

Then using the fact that the nodes were considered in the PEO ordering, we can show $\sum_{v_t \in OPT} w(v_t) = \sum_{v_t \in S} \tilde{w}_t$

Interval colouring

Interval Colouring Problem

- Given a set of intervals, colour all intervals so that intervals having the same colour do not intersect
- **Goal:** minimize the number of colours used.



We use 4 colors in this example. **Question:** Is this optimal?

Interval colouring

Interval Colouring Problem

- Given a set of intervals, colour all intervals so that intervals having the same colour do not intersect
- **Goal:** minimize the number of colours used.

- We could simply apply the m-machine ISP for increasing m until we found the smallest m that is sufficient. But this would not be as efficient as the greedy algorithm to follow.

Greedy interval colouring algorithm

- Consider the EST (earliest starting time) for interval colouring.
  - Sort the intervals by non decreasing starting times
  - Assign each interval the smallest numbered colour that is feasible given the intervals already coloured.
- Recall that EST is a terrible algorithm for ISP.
- **Note:** this algorithm is equivalent to LFT (latest finishing time first).

Theorem

EST is optimal for interval colouring

Proof idea: When does the algorithm use a new colour? In any graph, the colouring number is at least as large as the maximum clique size (and equal for interval and more generally perfect graphs).

Greedy Interval Colouring

```
Sort intervals so that $s_1 \leq s_2 \leq \ldots \leq s_n$
FOR $i = 1$ to $n$
  Let $k := \min\{\ell : \ell \neq \chi(j)$ for all $j < i$ such that the
    $j^{th}$ interval intersects the $i^{th}$ interval$\}$
$\sigma(i) := k$
    % The $i^{th}$ interval is greedily coloured by the smallest non conflicting
colour.
ENDFOR
```

How does this generalize to a greedy algorithm for vertex coloring chordal graphs?

The m machine weighted interval scheduling problem and its graph theoretic interpretation

As a graph problem, m machine weighted interval scheduling becomes the maximum vertex m-colourable problem for interval graphs. Do results for m machine (weighted or unweighted) interval scheduling carry over to the maximum vertex m-colourable problem for chordal graphs?

Borodin, Cashman and Magen [2011] show that fixed order priority stack algorithms cannot optimally solve the m machine weighted interval scheduling problem. The one-machine stack algorithm extends to yield a $2 - \frac{1}{m}$ approximation which Bar-Noy et al first obtained by applying the one machine algorithm, "one machine at a time".

However, for any fixed m, dynamic programming can optimally solve the m machine weighted interval scheduling problem in polynomial time $O(n^m)$.

There is a min cost, max flow algorithm that can solve m (wlg. $m \leq n$) machine weighted interval scheduling in time $O(n^2 \log n)$.

m machine interval scheduling continued

We previously mentioned Regev's $\frac{\log m}{\log \log m}$ fixed order priority inapproximation for the restricted machines makespan problem. It is not currently known how to obtain an analogous inapproximation for adaptive priority algorithms.

Similarly, we can derive a (weak) fixed order priority stack inapproximation for the m machine weighted interval scheduling problem but do not know how to obtain an inapproximation for the adaptive model.

Curiously, thus far the best inapproximation we have is for 2 machines (namley, $\frac{6}{\sqrt{30}}$) ≈ 1.095 which can be extended to $\frac{m}{m-1}$ for m machines (i.e. the bound gets weaker) in contrast to the $2 - \frac{1}{m}$ algorithm where the approximation gets weaker.

And now to answer a question we previously raised, Yannakakis and Gavril [1987] show how to solve the chordal graph maximum m vertex colourable problem in polynomal time for any fixed m but show that it is NP-hard when m is a parameter of the problem. This shows that we cannot expect to reduce min vertex colouring to the max m colourable problem.

A k-PEO and inductive k-independent graphs

- An alternative way to describe a PEO is to say that $Nbhd(v_i) \cap v_{i+1}, \ldots, v_n\}$ has independence number 1.
- We can generalize this to a k-PEO by saying that $Nbhd(v_i) \cap v_{i+1}, \ldots, v_n\}$ has independence number at most k.
- We will say that a graph is an inductive k-independent graph if it has a k-PEO.
- Inductive k-independent graphs generalize both chordal graphs and $k+1$-claw free graphs.
- The intersection graph induced by the JISP problem is an inductive 2-independent graph.
- Using a k-PEO, a fixed-order priority algorithm (resp. a priority stack algorithm) is a k-approximation algorithm for MIS (resp. for WMIS) wrt inductive k-independent graphs.
- There are analogous (and weaker) results for the max m vertex colourable problem with respect to inductive k-independent graphs.

Dynamic programming and scaling

We now move on to one of the main objects of study in an undergraduate algorithms course.

- We have previously seen that with some use of brute force and greediness, we can achieve PTAS algorithms for the identical machines makespan which is polynomial in the number n of jobs but exponential in the number m of machines and $\frac{1}{\epsilon}$ where $1+\epsilon$ is the approximation guarantee.
- For the knapsack problem we had a PTAS that was polynomial in n and exponential in $\frac{1}{\epsilon}$. .
- We briefly mentioned that dynamic programming (DP) and scaling can be used to achieve an FPTAS for the knapsack problem. We will show how this idea works for the knapsack problem and also to improve the results for the makespan problem on identical machines.
- The application and importance of dynamic programming now goes beyond search and optimzation problems (its original purpose).

What is Dynamic Programming (DP)

- We will consider a few more or less "natural" DP algorithms and at least one not so obvious DP algorithm.
- In greedy like algorithms (and also local search, our next major paradigm) it is often easy to come up with reasonably natural algorithms (although we have seen some not so obvious examples) whereas sometimes the analysis can be relatively involved.
- In contrast, once we come up with an appropriate DP algorithm, it is often the case that the analysis is relatively easy.
- Here informally is the essense of DP algorithms: define an approriate generalization of the problem (which we usually give in the form of a multi-dimensional array) such that
  1. the desired result can be easily obtained from the array $S[\ ,\ , \ldots\ ]$
  2. each entry of the array can be easily computed given "previous entries"
  3. This latter computational statement embodies the *optimal substructure property* of a problem. That is, an optimal solution contains within it optimal solutions to subproblems.

What more precisely is dynamic programming?

- So far, there are only a few attempts to formalize precise mdoels for (types) of dynamic programming algorithms.
- There are some who say this is not a useful question.
- I would disagree with the following comment: Whatever can be done in polynomial time, can be done by a polynomial time DP algorithm. What is the reasoning behind such a comment?
  Open problem: Can there be an optimal polynomal time DP (in any "reasonable" meaning of what is DP) for the maximum size or weight bipartite matching problem? Note: There are polynomial time optimal algorithms for these problem.
- And there may be more fundamdental or philosophical reasons for arguing against such attempts to formalize concepts.

What more precisely is dynamic programming?

- So far, there are only a few attempts to formalize precise mdoels for (types) of dynamic programming algorithms.
- There are some who say this is not a useful question.
- I would disagree with the following comment: Whatever can be done in polynomial time, can be done by a polynomial time DP algorithm. What is the reasoning behind such a comment?
  Open problem: Can there be an optimal polynomal time DP (in any "reasonable" meaning of what is DP) for the maximum size or weight bipartite matching problem? Note: There are polynomial time optimal algorithms for these problem.
- And there may be more fundamdental or philosophical reasons for arguing against such attempts to formalize concepts.
- Samuel Johnson (1709-1784): All theory is against freedom of the will; all experience for it.

Bellman 1957 (in the spirit of Samuel Johnson)

Bellman (who introduced dynamic programming) argued against attempts to formalize DP.

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 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.

Some simple DP algorithms

- Let's begin with an example used in many texts, namely a DP for the weighted interval scheduling problem WISP.
- We have already claimed that no priority algorithm can yield a constant approximation ratio but that we can obtain a 4-approximation using a revocable accaptance priority algorithm and an optimal algorithm using a priority stack algorithm.
- The optimal DP algorithm for WISP is based on the following "semantic array":
  - Sort the intervals $I_j = [s_j, f_j)$ so that $f_1 \leq f_2 \ldots \leq f_n$ (i.e. the PEO).
  - Define $\pi(i) = \max j : f_j \leq s_i$ (Note; if we do not want intervals to touch then use $f_j < s_i$.)
  - The definition of $\pi()$ is specific to this problem and I do not know a generalization for chordal graphs and hence the DP approach does not naturally extend.
  - For $1 \leq i \leq n$, Define $V[i]$ = optimal value obtainable from intervals $\{I_1, \ldots I_i\}$.

The DP for WISP continued

- We defined the array $V[\,]$ just in terms of the optimal value but the same array element can also contain a solution associated with this optimal value.
- So how would we efficiently compute the entries of $V[\,]$.
- The computation or recursive array (let's temporarily call it $\tilde{V}[\,]$) associated with $V[\,]$ is defined as follows:
  1. $\tilde{V}[1] = v_1$
  2. For $i > 1$, $\tilde{V}[i] = \max\{A, B\}$ where
     - ⋆ $A = V[i-1]$
     - ⋆ $B = v_i + \tilde{V}[\pi(i)]$

     That is, either we use the i^{th} interval or we don't.
- So why am I being so pedantic about this distinction between $V[\,]$ and $\tilde{V}[\,]$?

The DP for WISP continued

- We defined the array $V[\,]$ just in terms of the optimal value but the same array element can also contain a solution associated with this optimal value.
- So how would we efficiently compute the entries of $V[\,]$.
- The computation or recursive array (let's temporarily call it $\tilde{V}[\,]$) associated with $V[\,]$ is defined as follows:
  1. $\tilde{V}[1] = v_1$
  2. For $i > 1$, $\tilde{V}[i] = \max\{A, B\}$ where
     - ★ $A = V[i-1]$
     - ★ $B = v_i + \tilde{V}[\pi(i)]$

     That is, either we use the i^{th} interval or we don't.
- So why am I being so pedantic about this distinction between $V[\,]$ and $\tilde{V}[\,]$?
- I am doing this here just to point out that a proof of correctness would require showing that these two arrays are indeed equal! I will hereafter not make this distinction with the understanding that one does have to show that the computational or recursive array does indeed compute the entries correctly.

Some comments on DP and the WISP DP

- We can sort the intervals and compute $\pi()$ in time $O(n \log n)$ and then sequentially compute the entries of V in time $O(1)$ per iteration.
- We can also recursivley compute V, BUT must use memoization to avoid recomputing entries.
- To some extent, the need to use memoization distinguishes dynamic programming from divide and conquer.
- We can extend this DP to optimally solve the weighted interval scheduling problem when there are m machines; that is, we want to schedule intervals so that there is no intersection on any machine.
- This extension would directly lead to time (and space) complexity $O(n^{m+1})$; $O(n^m)$ with some more care.
- As we will soon discuss, we can model this simple type of DP by a priority branching tree (pBT) algorithm as formulated by Alekhnovich et al. Within this model, we can prove that for any fixed m, the width (and hence the space and thus time) of the algorithm for optimally scheduling intervals on m machines is $\Omega(n^m)$. The curse of dimensionality is necessary within this model.

A pseudo polynomial time "natural DP" for knapsack

Consider an instance of the (NP-hard) knapsack problem; that is we are given item $\{(v_k, s_k) | 1 \leq k \leq n\}$ and a knapsack capacity C. Following along the lines of the WISP DP, the following is a reasonably natural approach to obtain a "pseudo polynomal space and time" DP:

- For $1 \leq i \leq n$ and $0 \leq c \leq C$, define $V[i, c]$ to be the value of an optimum solution using items $\mathcal{I}_i \subseteq \{I_1, \ldots, I_i\}$ and satisfying the size constraint that $\sum_{I_j \in \mathcal{I}_i} s_j \leq c$.
- A corresponding recursive DP is as follows:
  1. $V[0, c] = 0$ for all c
  2. For $i > 0$, $V[i] = \max\{A, B\}$ where
     - ★ $A = V[i-1, c]$
     - ★ $B = v_i + V[c - s_i]$ if $s_i \leq c$ and $V[i-1, c]$ otherwise.

  Note: easy to make mistakes so again have to verify that this recursive definition is correct.
- The space and time complexity is $O(nC)$ which is pseudo polynomial in the sense that C can be exponential in the encoding of the input.

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 n and the numeric value $V = \sum_i v_i$ (or $\max_i v_i$) (rather than the encoded length of the maximum possible profit.

Define $S[j, v] =$ the minimum capacity 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 give in the form of an array) such that

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

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

1. The value of an optimal solution is $max\{v | S[n, v] \leq C\}$.
2. 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 $v \leq \sum_{j=1}^{n} v_j$.
    - Basis: $S[0, v] = \infty$ for all v
    - Induction: $S[j, v] = \min\{A, B\}$ where $A = S[j-1, v]$ and $B = S[j-1, \max\{v - v_j, 0\}] + s_j$.
3. 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.
4. For efficiency one usually computes these entries iteratively but one could use a recursive program with *memoization*.
5. The running time is $O(n, V)$ where $V = \sum_{j=1}^{n} v_j$.
6. 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 down (or 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 combinaton 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 a good T.

The optimal DP for makespan on identical machines when there is 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.
- The array we will use to obtain the desired optimal makespan is as follows:
  $M[x_1, \ldots, x_d] =$ the minimum number of machines needed to schedule x_i jobs having size z_i within makespan T. (Here we can assume $T \geq \max p_i \geq \max z_i$ so that this minimum is finite.)
- The n jobs can be scheduled within makespan T iff $M[n_1, , n_d]$ is at most m.

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 $M[x_1, \dots, x_d]$

- Clearly $M[0, \dots, 0] = 0$ 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 v_i jobs with size z_i on one machine does not exceed the target makespan T.
- $M[x_1, \dots, x_d] = 1 + \min_{(v_1,\dots,v_d) \in V : v_i \leq x_i} M[x_1 - v_1, \dots, x_d - v_d]$
- 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 $(1 + 1/s)T$.

The pBT model

Despite the warnings of Samuel Johnson and Richard Bellman, there are models of dynamic programming. None of these models have garnered that much attention but I still believe that formalizing dynamic programming is worthwhile.

Specific Challenge: Find a *convincing* DP model with respect to which we can prove (if true) that no efficient DP algorithm can optimally solve unweighted or edge weighted bipartite matching.

Alekhnovich et al [2011] proposed a model for a simple class of DP algorithms. In hindsight, this model could perhaps better serve as a class of algorithms modeling parallel greedy algorithms, as well as a model for simple branch and bound algorithms.

The priority branching tree (pBT) model is informally a levelled tree where each path in the tree is a priority algorithm. At any node on a path, the algorithm can branch according to different decisions concerning the input item, can continue the path having made a single decision, or can decide to terminate this path.

pBT model continued

We can distinguish three cases:

1. Fixed order pBT (with online oBT as a special case). Here the algorithm chooses an ordering of items $I_1, \ldots, I_n$ and then at every tree node at level k of the tree, item I_k is being considered.
2. Adaptive order pBT. Now as in the fixed order model, at every tree node at level k of the tree, the algorithm is considering some $I_{\pi(k)}$ where this item is determined adaptively as in the adaptive priority model.
3. Strongly adaptive pBT. Now each path in the tree is an adaptive priority algorithm and the input item being considered at a node depends on the path taken.

Some pBT results

We will state all results in terms of the required width of pBT algorithms to obtain optimal solutions (or obtain a solution for a search problem).

- For fixed m, any adaptive order pBT for weighted interval scheduling on m machines requires width $\Omega(n^m)$ which is a tight bound.
- For the knapsack problem, any adaptive order pBT algoroithm requires $\Omega(\frac{2^{n/2}}{\sqrt{n}})$ width (even for the proportional profit version of the problem where $v_i = s_i$ for all i. The proportional profit knaosack problem is refered to as the subset sum problem).
- And fully adaptive pBT for 3SAT requires width $2^{\Omega n}$. for problem of returing a satisfying instance if one exists.

There are more results and other models of DP. These results sometimes give some useful inapproximation results but mostly weak results.