CSC2420 Spring 2015: Lecture 3

Allan Borodin

January 22, 2015

Announcements and todays agenda

- Assignment 1 due next Thursday. I may add one or two additional questions today or tomorrow.
- Todays agenda
  1. Complete missing discussion from last week (i.e. slides 10-16 on extensions of the priority model and slides 22-27 start of dynamic programming).
  2. We will discuss *Greedy*$_\alpha$, a *revocable priority algorithm* for the weighted interval selection problem WISP (and WJISP, weighted job interval selection). For WISP, this algorithm has an approximation ratio of $\frac{1}{\alpha(1-\alpha)}$ which is optimized at $\alpha = 1/2$.
  3. We then introduce the *priority stack model*. This model provides an optimal algorithm for WISP and a 2-approximation for WJISP.
  4. The ISP (resp. JISP) problems are instances of the MIS problem for chordal graphs (resp inductive 2-indpendent graphs) which will be defined.
  5. We return briefly to greedy algorithms and consider "the natural" and modified greedy algorithms for the weighted vertex cover problem..
  6. We then move on to dynamic programming.

Extensions of the priority model: priority with revocable acceptances

- For packing problems, we can have priority algorithms with revocable acceptances. That is, in each iteration the algorithm can now reject previously accepted items in order to accept the current item. However, at all times, the set of currently accepted items must be a feasible set and all rejections are permanent.
- Within this model, there is a 4-approximation algorithm for the weighted interval selection problem WISP (Bar-Noy et al [2001], and Erlebach and Spieksma [2003]), and a ≈ 1.17 inapproximation bound (Horn [2004]). More generally, the algorithm applies to the weighted job interval selection problem WJISP resulting in an 8-approximation.
- The model has also been studied with respect to the proportional profit knapsack problem/subset sum problem (Ye and B [2008]) improving the constant approximation. And for the general knapack problem, the model allows a 2-approximation.

Other similar models allowing limited changes of previous decisions

- There are other online and greedy-like settings (beyond packing problems) in which algorithms allowing limited changes of previous decisions have been considered.
- Later we will consider online bipartite matching (still a packing problem) where the concern has been how many edge reassignments are needed to obtain an optimal or near optimal matching.
- Makespan has also been studied in terms of reassignments of items.
- We will soon mention Graham's convex hull algorithm which can be viewed as another example of a revocable acceptance type of algorithm.

The *Greedy*$_\alpha$ algorithm for WISP and WJISP

The Erlebach and Spieksma (independently, Bar Noy et al ADMISSION) algorithm:

```
$S$ := $\emptyset$                % $S$ is the set of currently accepted intervals
Sort input intervals so that $f_1 \leq f_2 \ldots \leq f_n$
**for** $i = 1..n$
    $C_i$ := min weight subset of $S : (S/C_i) \cup \{I_i\}$ feasible
        For WISP and WJISP there is a unique conflicting set;
        not so for knapsack problem.
    **if** $v(C_i) \leq \alpha \cdot v(I_i)$ **then**
        $S$ := $(S/C_i) \cup \{I_i\}$
    **end if**
**END FOR**
```

Figure : Priority algorithm with revocable acceptances for WISP and WJISP

The approximation bounds for *Greedy*$_\alpha$

- We note that the JISP problem is known to be NP-hard (indeed, NP hard to approximate to within $(1+\delta)$ factor for some very small $\delta > 0$). Best known approximation for WJISP is a 2-approximation (for all practical algorithms this is also the best approximation for the unweighted case).
- The *Greedy*$_\alpha$ algorithm (which is not greedy by my definition) has a tight approximation ratio of
  1. $\frac{1}{\alpha(1-\alpha)}$ for WISP and
  2. $\frac{2}{\alpha(1-\alpha)}$ for WJISP.
- Sketch of the charging argument for WISP
  1. Let A be the final set of intervals selected by *Greedy*$_\alpha$, T the set of intervals that occur in the current solution at some time during the execution and OPT is an optimal solution.
  2. Show $v(A) \geq (1-\alpha)v(T)$.
  3. Show how to charge weight of intervals in OPT to intervals in T so that any interval in $I_j \in T$ will receive at most a charge of at most v_j/α. That is, $v(T) \geq \alpha v(OPT)$.

Graham's [1972] convex hull algorithm

Graham's scan algorithm for determining the convex hull of a set of n points in the plane is an efficient ($O(n \log n)$ time) algorithm in the framework of the revcocable priority model. (This is not a search or optimization problem but does fit the framework of making a decision for each input point. For simplicity assume no three points are colinear.)

```
Choose a point that must be in the convex hull
    (e.g. the (leftmost) point $p_1$ having smallest $y$ coordinate)
Sort remaining points $p_2, \ldots, p_n$ by increasing angle with respect to $p_1$
Push $p_1, p_2, p_3$ on a Stack
**for** $i = 4..n$
Push $p_i$ onto stack
**While** the three top points $u, v, p_i$ on the Stack make a "right turn"
    Remove $v$ from stack
**End While**
Return points on Stack as the points defining the convex hull.
```

Figure : The Graham convex hull scan algorithm

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, when 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 interval graphs are chordal) by ordering the intervals such that $f_1 \leq f_2 \ldots \leq f_n$. Trees are also chordal graphs and a PEO is obtained by stripping off leaves one by one.

MIS and colouring chordal graphs

- Using this ordering (by earliest finishing time), 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 in a fixed order greedy algorithm optimally solves the unweighted MIS problem for any chordal graph. This can be shown by an inductive argument showing that the partial solution after the ith iteration is *promising* in that it can be extended to an optimal solution; and it can also be shown by a simple *charging argument*.
- We also know that the greedy algorithm that orders intervals such that $s_1 \leq s_2 \ldots \leq s_n$ and then colours nodes using the smallest feasible colour is an optimal algorithm for colouring interval graphs. Ordering by earliest starting times is (by symmetry) equivalent to ordering by latest finishing times first. The generalization of this is that any chordal graph can be optimally coloured by a greedy algorithms that orders vertices by the reverse of a PEO. This can be shown by arguing that when the algorithm first uses colour k, it is witnessing a clique of size k.

The optimal priority stack algorithm for the (WMIS) problem in chordal graphs ; Akcoglu et al [2002]

```
$Stack$ := $\varnothing$          % $Stack$ is the set of items on stack
Sort nodes using a PEO
Set $w'(v_i)$ := $w(v_i)$ for all $v_i$
          % $w'(v)$ will be the residual weight of a node
**For** $i = 1..n$
    $C_i$ := $\{v_j | j < i, v_i \in Nbhd(v_j)$ and $v_j$ on $Stack\}$
    $w'(v_i)$ := $w'(v_i) - w'(C_i)$
    **If** $w'(v_i) > 0$ **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**
```

A common generalization of $k+1$-claw free graphs and chordal graphs

One vague theme I try to think about is the interplay between classes of problems and classes of algorithms. In some way this leads to a common extension of chordal and $k+1$-claw free graph implicitly defined in Akcoglu et al [2002] and pursued in Ye and B. [2009].

A graph is inductively k-independent is there is a "k-PEO" ordering of the vertices $v_1, \ldots, v_n$ such that $Nbhd(v_i) \cap \{v_{i+1}, \ldots, v_n\}$ has at most k independent vertices.

For example,

- The JISP problem induces an inductively 2-independent graph.
- Every planar graph is inductively 3-independent.

Extending results from chordal to inductive k independent graphs

- The WMIS stack algorithm and analysis for chordal graphs extends to provide a k approximation for WMIS on inductive k independent graphs by using a k-PEO.
- The reverse order k-PEO greedy algorithm is a k-approximation graph colouring algorithm for inductive k independent graphs.
- Note that for interval graphs, the reverse order PEO is equivalent to ordering the intervals so that $s_1 \leq s_2 \ldots \leq s_n$ where s_j is the start time of interval I_j.
- This concept generalizes "inductive degree k" and all graphs having treewidth k are inductive degree k. (Treewidth k will be defined later.)

Another example where the "natural greedy" is not best

- Before moving (for now) beyond greedy-like algorithms, we consider another example (as we saw for set packing) where the "natural greedy algorithm" does not yield a good approximation.
- The vertex cover problem: Given node weighted graph $G = (V, E)$, with node weights $w(v), v \in V$.
  Goal: Find a subset $V' \subset V$ that covers the edges (i.e. $\forall e = (u, v) \in E$, either u or v is in V') so as to mininize $\sum_{v \in V'} w(v)$.
- Even for unweighted graphs, the problem is known to be NP-hard to obtain a 1.3606 approximation and under another (not so universally believed) conjecture (UGC) one cannot obtain a $2 - \epsilon$ approximation.
- For the unweighted problem, there are simple 2-approximation greedy algorithms such as just taking V' to be any maximal matching.
- The set cover problem is as follows: Given a weighted collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ over a universe U with set weights $w(S_i)$.
  Goal: Find a subcollection $\mathcal{S}'$ that covers the universe so as to minimize $\sum_{S_i \in \mathcal{S}'} w(S_i)$.

The natural greedy algorithm for weighted vertex cover

If we consider vertex cover as a special case of set cover (how?), then the natural greedy (which is essentially optimal for set cover) becomes the following:

```
$d'(v)$ := $d(v)$ for all $v \in V$
        % $d'(v)$ will be the residual degree of a node
While there are uncovered edges
    Let $v$ be the node minimizing $w(v)/d'(v)$
    Add $v$ to the vertex cover;
    remove all edges in $Nbhd(v)$;
    recalculate the residual degree of all nodes in $Nbhd)v)$
End While
```

Figure : Natural greedy algorithm for weighted vertex cover with approximation ratio $H_n \approx \ln n$ where $n = |V|$.

Clarkson's [1983] modified greedy for weighted vertex cover

```
$d'(v)$ := $d(v)$ for all $v \in V$
        % $d'(v)$ will be the residual degree of a node
$w'(v)$ := $w(v)$ for all $v \in V$
        % $w'(v)$ will be the residual weight of a node
**While** there are uncovered edges
    Let $v$ be the node minimizing $w'(v)/d'(v)$
    $w$ :=$w'(v)/d'(v)$
    $w'(u)$ :=$w'(u) - w$ for all $u \in Nbhd(v)$
    Add $v$ to the vertex cover;
    remove all edges in $Nbhd(v)$;
    recalculate the residual degree of all nodes in $Nbhd(v)$
**End While**
```

Figure : Clarkson's greedy algorithm for weighted vertex cover with approximation ratio 2

Dynamic Programming (DP)

- The application and importance of dynamic programming goes well beyond search and optimzation problems.
- We will consider a few more or less "natural" DP algorithms and some not so obvious DP algorithms.
- 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"

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?
  Can there be an optimal polynomal time DP for say maximum size or weight bipartite matching?
- 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?
  Can there be an optimal polynomal time DP for say maximum size or weight bipartite matching?
- 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 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 $f_j < s_i$.)
    - 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.
- 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 $\Omega(n^m)$. That is, the curse of dimensionality is necessary within this model.