CSC2420 Fall 2012: Algorithm Design, Analysis and Theory

Allan Borodin

September 27, 2012

Lecture 3

1. We will first do the analysis of the *Greedy*$_\alpha$ revocable priority algorithm for the WISP problem showing that the algorithm has an approximation ratio of $\frac{1}{\alpha}(1-\alpha)$ which is optimized at $\alpha = 1/2$.
2. We mention a similar algorithm: Graham's convex hull scan algorithm
3. We then introduce the priority stack model
4. We return briefly to greedy algorithms and consider "the natural greedy algorithms for the weighted versions of set packing and vertex cover".
5. We then move on to two dynamic programming algorithms, for the makespan and knapsack problems.

The $Greedy_\alpha$ algorithm for WJISP

The algorithm as stated by Erlebach and Spieksma (and called ADMISSION by Bar Noy et al) is as follows:

$S := \varnothing$ % 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
    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 WJISP

The $Greedy_\alpha$ algorithm (which is not greedy by my definition) has a tight approximation ratio of $\frac{1}{\alpha(1-\alpha)}$ for WISP and $\frac{2}{\alpha(1-\alpha)}$ for WJISP.

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 a 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

The "natural greedy algorithm": WMIS for $k+1$-claw free graphs

We return briefly to greedy algorithms considering the vague idea of "the natural greedy algorithm".

$k+1$-claw free graphs

A graph $G = (V, E)$ is $k+1$-claw free if for all $v \in V$, the induced subgraph of $Nbhd$(v) has at most k independent vertices (i.e. does not have a $k+1$ claw as an induced subgraph).

- There are many types of graphs that are $k+1$ claw free for small k; in particular, the intersection graph of axis parallel translates of a convex object in the two dimensional plane is a 6-claw free graph. For rectangles, the intersection graph is 5-claw free.
- Let $\{S_1, \ldots, S_n\}$ be subsets of a universe U such that $|S_i| \leq k$. The intersection graph $G = (V, E)$ defined by $(S_i, S_j) \in E$ iff $S_i \cap S_j \neq \varnothing$ is a $k+1$ claw free graph.

$k+1$-claw free graphs and the natural greedy algorithm continued

Of special note are 3-claw free graphs which are simply called claw free. For example, line graphs are 3-claw free. A matching in a graph G corresponds to an independent set in the line graph $L(G)$.

The natural greedy algorithm for WMIS

The natural greedy algorithm sorts the vertices such that $w(v_1) \geq w(v_2) \ldots \geq w(v_n)$ and then accepts vertices greedily; i.e. if the vertex is independent of previously accepted vertices then accept.

The natural greedy algorithm provides a k-approximation for WMIS on $k+1$ claw free graphs. (A more complex algorithm, generalizing weighted matching, optimally solves the WMIS for 3-claw free graphs.)

The "natural greedy" might not be the best greedy: Set packing

We consider two examples (weighted set packing and vertex cover) where the (seemingly) most natural) greedy algorithm is not the best greedy (approximation) algorithm.

The *weighted set packing problem* (AKA *the single minded combinatorial auction problem*) is defined as follows: There is a universe U of size m, a collection of subsets $\{S_1, \ldots, S_n\}$ of U and a weight w_i for each set S_i. The objective is to select a non-intersecting subcollection of these subsets so as to maximize the weight of the chosen sets.

- It is NP hard to obtain approximation ratio $m^{\frac{1}{2}-\epsilon}$ for any $\epsilon > 0$.
- The induced intersection graph is $m+1$ *claw free* and hence the natural greedy algorithm yields a a $\min\{m, n\}$ approximation. Let's assume $n >> m$ and try to improve upon the m approximation.
- The "next most natural greedy algorithm" (ordering sets by non-increasing $w(S)/|S|$) is still an m-approximation.
- We can obtain approximation $2\sqrt{m}$ by ordering sets by non-increasing $w(S)/\sqrt{|S|}$.

Second example where natural greedy is not best: weighted vertex cover

If we consider vertex cover as a special case of set cover 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

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

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.

It can be shown that the WMIS stack algorithm and analysis for chordal graphs extends to provide a k approximation for inductive k independent graphs by using a k-PEO. The reverse order k-PEO greedy algorithm is a k-approximation for inductive k independent graphs.

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 a PTAS for the makespan problem which is polynomial in m and n, and how to achieve an FPTAS for the knapsack problem.
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 give in the form of an array) such that

1. the desired result can be easily obtained from ths array $S[\ ,\ ]$
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
  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 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 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 a good 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 szie 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 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 $(1 + 1/s)T$.