CSC2420: Algorithm Design, Analysis and Theory Spring (or Winter for pessimists) 2017

Allan Borodin

January 16, 2017

Lecture 2

Announcements:

- I plan to have the first couple of questions for assignment 1.

Todays agenda:

- Continue "whirlwind" discussion of basic combinatorial algorithms.
- Greedy and myopic algorithms.
- The priority model (which we briefly introduced in Lecture 1)
- Dynamic programming

A perspective on algorithm design and analysis

- I (consistent with many texts) am organizing material mainly by the algorithmic paradigm(s) used to solve a problem (to the extent we can identify specific paradigms.
- When an algorithmic approach works for a given problem, we can ask if that approach still is useful for a generalization of that problem?
- When an approach does not work for a problem, is there a way to extend that approach?

Vertex cover: where the "natural greedy" is not best

- We will consider two examples (weighted vertex cover and 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 (WVC)

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. Approximation ratio $H_n \approx \ln n$ where $n = |V|$.

Clarkson's [1983] modified greedy for WVC

$d'(v) := d(v)$ 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 $u \in Nbhd(v)$

% For analysis only, set $we(u, v) = w$

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. Approximation ratio 2. Invariant: $w(v) = w'(v) + sum_{e \in E} we(e)$

Greedy decisions and priority algorithms

- For example, in the knapsack problem, a greedy decision always takes an input if it fits within the knapsack constraint and in the makespan problem, a greedy decision always schedules a job on some machine so as to minimize the increase in the makespan. (This is somewhat more general than saying it must place the item on the least loaded machine.)
- If we do not insist on greediness, then priority algorithms would best have been called myopic algorithms.
- We have both fixed order priority algorithms (e.g. unweighted interval scheduling and LPT makespan) and adaptive order priority algorithms (e.g. the set cover greedy algorithm and Prim's MST algorithm).
- The key concept is to indicate how the algorithm chooses the order in which input items are considered. We cannot allow the algorithm to choose say "an optimal ordering".
- We might be tempted to say that the ordering has to be determined in polynomial time but that gets us into the "tarpit" of trying to prove what can and can't be done in (say) polynomial time.

The priority model definition

- We take an information theoretic viewpoint in defining the orderings we allow.
- Lets first consider deterministic fixed order priority algorithms. Since I am using this framework mainly to argue negative results (e.g. a priority algorithm for the given problem cannot achieve a stated approximation ratio), we will view the semantics of the model as a game between the algorithm and an adversary.
- Initially there is some (possibly infinite) set $\mathcal{J}$ of potential inputs. The algorithm chooses a total ordering π on $\mathcal{J}$. Then the adversary selects a subset $\mathcal{I} \subset \mathcal{J}$ of actual inputs so that $\mathcal{I}$ becomes the input to the priority algorithm. The input items $I_1, \ldots, I_n$ are ordered according to π.
- In iteration k for $1 \leq k \leq n$, the algorithm considers input item I_k and based on this input and all previous inputs and decisions (i.e. based on the current state of the computation) the algorithm makes an irrevocable decision D_k about this input item.

The fixed (order) priority algorithm template

$\mathcal{J}$ is the set of all possible input items
Decide on a total ordering π of $\mathcal{J}$
Let $\mathcal{I} \subset \mathcal{J}$ be the input instance
$S := \varnothing$ % S is the set of items already seen
$i := 0$ % $i = |S|$
while $\mathcal{I} \setminus S \neq \varnothing$ **do**
    $i := i + 1$
    $\mathcal{I} := \mathcal{I} \setminus S$
    $I_i := \min_\pi \{I \in \mathcal{I}\}$
    make an irrevocable decision D_i concerning I_i
    $S := S \cup \{I_i\}$
end

Figure : The template for a fixed priority algorithm

Some comments on the priority model

- A special (but usual) case is that π is determined by a function $f : \mathcal{J} \to \Re$ and and then ordering the set of actual input items by increasing (or decreasing) values $f()$. (We can break ties by say using the index of the item to provide a total ordering of the input set.) N.B. We make no assumption on the complexity or even the computability of the ordering π or function f.
- NOTE: Online algorithms are fixed order priority algorithms where the ordering is given *adversarially*; that is, the items are ordered by the index of the item.
- As stated we do not give the algorithm any additional information other than what it can learn as it gradually sees the input sequence.
- However, we can allow priority algorithms to be given some (hopefully easily computed) global information such as the number of input items, or say in the case of the makespan problem the minimum and/or maximium processing time (load) of any input item. (Some inapproximation results can be easily modified to allow such global information.)

The adaptive priority model template

$\mathcal{J}$ is the set of all possible input items
$\mathcal{I}$ is the input instance
$S := \varnothing$ % S is the set of items already considered
$i := 0$ % $i = |S|$
while $\mathcal{I} \setminus S \neq \varnothing$ **do**
    $i := i + 1$
    decide on a total ordering π_i of $\mathcal{J}$
    $\mathcal{I} := \mathcal{I} \setminus S$
    $I_i := \min_{\leq_{\pi_i}} \{I \in \mathcal{I}\}$
    make an irrevocable decision D_i concerning I_i
    $S := S \cup \{I_i\}$
    $\mathcal{J} := \mathcal{J} \setminus \{I : I \leq_{\pi_i} I_i\}$
    % some items cannot be in input set
end

Figure : The template for an adaptive priority algorithm

Inapproximations with respect to the priority model

Once we have a precise model, we can then argue that certain approximation bounds are not possible within this model. Such inapproximation results have been established with respect to priority algorithms for a number of problems but for some problems much better approximations can be established using extensions of the model.

1. For the weighted interval selection (a *packing problem*) with arbitrary weighted values (resp. for proportional weights $v_j = |f_j - s_j|$), no priority algorithm can achieve a constant approximation (respectively, better than a 3-approximation).
2. For the knapsack problem, no priority algorithm can achieve a constant approximation. We note that the maximum of two greedy algorithms (sort by value, sort by value/size) is a 2-approximation.
3. For the set cover problem, the natural greedy algorithm is essentially the best priority algorithm.
4. As previously mentioned, for deterministic fixed order priority algorithms, there is an $\Omega(\log m / \log \log m)$ inapproximation bound for the makespan problem in the restricted machines model.

Greedy algorithms for the set packing problem

The set packing problem

We are given n subsets $S_1, \ldots, S_n$ from a universe U of size m. In the weighted case, each subset S_i has a weight w_i. The goal is to choose a disjoint subcollection $\mathcal{S}$ of the subsets so as to maximize $\sum_{S_i \in \mathcal{S}} w_i$. In the s-set packing problem we have $|S_i| \leq s$ for all i.

- This is a well studied problem and by reduction from the max clique problem, there is an $m^{\frac{1}{2}-\epsilon}$ hardness of approximation assuming $NP \neq ZPP$. For s-set packing with constant (wrt m) $s \geq 3$, there is an $\Omega(s/\log s)$ hardness of approximation assuming $P \neq NP$.
- Set packing is the underlying allocation problem in what are called combinatorial auctions as studied in mechanism design.
- We will consider two "natural" greedy algorithms for the s-set packing problem and a somewhat less obvious greedy algorithm for the set packing problem. These greedy algorithms are all fixed order priority algorithms.

The first natural greedy algorithm for set packing

Greedy-by-weight (*Greedy*$_{wt}$

Sort the sets so that $w_1 \geq w_2 \ldots \geq w_n$.
$\mathcal{S} := \emptyset$
For $i : 1 \ldots n$
    If S_I does not intersect any set in $\mathcal{S}$ then
        $\mathcal{S} := \mathcal{S} \cup S_i$.
End For

- In the unweighted case (i.e. $\forall i, w_i = 1$), this is an online algorithm.
- In the weighted (and hence also unweighted) case, greedy-by-weight provides an s-approximation for the s-set packing problem.
- The approximation bound can be shown by a charging argument where the weight of every set in an optimal solution is charged to the first set in the greedy solution with which it intersects.

The second natural greedy algorithm for set packing

Greedy-by-weight-per-size

Sort the sets so that $w_1/|S_1| \geq w_2/|S_2| \ldots \geq w_n/|S_n|$.
$\mathcal{S} := \varnothing$
For $i : 1 \ldots n$
    If S_I does not intersect any set in $\mathcal{S}$ then
        $\mathcal{S} := \mathcal{S} \cup S_i$.
End For

- In the weighted case, greedy-by-weight provides an s-approximation for the s-set packing problem.
- For both greedy algorithms, the approximation ratio is tight; that is, there are examples where this is essentially the approximation. In particular, greedy-by-weight-per-size is only an m-approximation where $m = |U|$.
- We usually assume $n >> m$ and note that by just selecting the set of largest weight, we obtain an n-approximation.

Improving the approximation for greedy set packing

- In the unweighted case, greedy-by-weight-per-size can be restated as sorting so that $|S_1| \leq |S_2| \ldots \leq |S_n|$ and it can be shown to provide an $\sqrt{m}$-approximation for set packing.
- On the other hand, greedy-by-weight-per-size does not improve the approximation for weighted set packing.

Greedy-by-weight-per-squareroot-size

Sort the sets so that $w_1/\sqrt{|S_1|} \geq w_2/\sqrt{|S_2|} \ldots \geq w_n/\sqrt{|S_n|}$.
$\mathcal{S} := \varnothing$
For $i : 1 \ldots n$
    If S_I does not intersect any set in $\mathcal{S}$ then
        $\mathcal{S} := \mathcal{S} \cup S_i$.
End For

Theorem: Greedy-by-weight-per-squareroot-size provides a $2\sqrt{m}$-approximation for the set packing problem. And as noted earlier, this is essentially the best possible approximation assuming $NP \neq ZPP$.

Another way to obtain an $O(\sqrt{m})$ approximation

There is another way to obtain the same aysmptototic improvement for the weighted set packing problem. Namely, we can use the idea of partial enumeration greedy; that is somehow combining some kind of brute force (or naive) approach with a greedy algorithm.

Partial Enumeration with Greedy-by-weight ($PGreedy_k$)

Let Max_k be the best solution possible when restricting solutions to those containing at most k sets. Let G be the solution obtained by $Greedy_{wt}$ applied to sets of cardinality at most $\sqrt{m/k}$. Set $PGreedy_k$ to be the best of Max_k and G.

- Theorem: $PGreedy_k$ achieves a $2\sqrt{m/k}$-approximation for the weighted set packing problem (on a universe of size m)
- In particular, for $k = 1$, we obtain a $2\sqrt{m}$ approximation and this can be improved by an arbitrary constant factor $\sqrt{k}$ at the cost of the brute force search for the best solution of cardinality k; that is, at the cost of say n^k.

$(k+1)$-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).

$(k+1)$-claw free graphs abstract a number of interesting applications.

- In particular, we are interested in the (weighted) maximum independent set problem (W)MIS for $(k+1)$-claw free graphs. Note that it is hard to approximate the MIS for an arbiitrary n node graph to within a factor $n^{1-\epsilon}$ for any $\epsilon > 0$.
- We can (greedily) k-approximate WMIS for $(k+1)$-claw free graphs.
- The (weighted) k-set packing problem is an instance of (W)MIS on $k+1$-claw free graphs. What algorithms generalize?
- There are many types of graphs that are $k+1$ claw free for small k; in particular, the intersection graph of 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.

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 knapsack problem, the model allows a 2-approximation.

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.t. $(S/C_i) \cup \{I_i\}$ 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.

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 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. The following priority stack algorithm provides an optimal solution for the WMIS problem on 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 input intervals so that $f_1 \leq f_2 \ldots \leq f_n$
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

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.**

More extensions of the priority model

- So far we have been implicitly assuming deterministic priority algorithms. We can allow the ordering and/or the decisions to be randomized.
- A special case of fixed priority with randomized orderings is when the input set is ordered randomly without any dependence on the set of inputs. In the online setting this is called the random order model.
- The revocable acceptances model is an example of priority algorithms that allow reassignments (of previous decisions) to some extent or at some cost.
- The partial enumeration greedy is an example of taking the best of some small set of adaptive priority algorithms.
- Priority stack algorithms are an example of 2-pass (or multi-pass) priority algorithms where in each pass we apply a priority algorithm. Of course, it has to be well specified as to what information can be made available to the next pass.

The random order model (ROM)

Motivating the random order model

The random order model provides a nice compromise between the often unrealistic negative results for worst case (even randomized) online algorithms and the often unrealistic positive setting of inputs being generated by simple distributions.

- In many online scenarios, we do not have realistic assumptions as to the distributional nature of inputs (so we default to worst case analysis). But in many applications we can believe that inputs do arrive randomly or more precisely uniformly at random.
- The ROM can be (at least) traced back to what is called the (classical) secretary (aka marriage or dowry) problem, popularized in a Martin Gardner Scientific American article.
- As Fiat et al (SODA 2015) note, perhaps Johannes Kepler (1571-1630) used some secretary algorithm when interviewing 11 potential brides over two years.

The secretary problem

The classical problem (which has now been extended and studied in many different variations is as follows:

- The classic problem (as in the Gardiner article) assumes an adversarially chosen set of distinct values for (say N) items that arrive in random order (e.g. candidates for a position, offers for a car, etc.). N is assumed to be known.
- Once an item (e.g. secretary) is chosen, that decision is irrevocable. Hence, this boils down to finding an *optimal stopping rule*, a subject that can be considered part of stochastic optimization.
- The goal is to select one item so as to maximize the probability that the item chosen is the one of maximum value.
- For any set of N values, maximizing the probability of choosing the best item immediately yields a bound for the expected (over the random orderings) value of the chosen item. For an "ordinal algorithm", these two measures are essentially the same. Why?

The secretary problem continued

- It is not difficult to show that any deterministic or randomized (adversarial order) online algorithm has competitive ratio [1] at most $O(\frac{1}{N})$. Hence the need to consider the ROM model to obtain more interesting (and hopefully more meaningful) results.
- We note (and this holds more generally) that "positive results" for the ROM model subsume the stochastic optimization scenario where inputs are generated by an unknown (and hence known) i.i.d. process. Why?
- There are many variations and extensions of the secretary problem some of which we will consider later (or at least mention).
- In general, any online problem can be studied with respect to the ROM model.

[1] Recall that for maximization problems, competitive and approximation ratios can sometimes presented as fractions $\alpha = \frac{ALG}{OPT} \leq 1$ and sometimes as ratios $c = \frac{OPT}{ALG} \geq 1$. I will try to follow the convention mainly used in each application.

The optimal stopping rule for the classical secretary problem

The amusing history of the secretary problem and the following result is taken up by Ferguson in a 1989 article.

Theorem: For N and r, there is an exact formula for the probability of selecting the maximum value item after observing the first r items, and then selecting the first item (if any) that exceeds the value of the items seen thus far. In the limit as $N \to \infty$, the optimal stopping rule is to observe (i.e. not take) the first $r = N/e$ items. The probability of obtaining the best item is then $1/e$ and hence the expected value of the item chosen is at least $\frac{1}{e} v_{max}$.

Variations and extensions of the secretary problem

- Instead of maximizing the probability of choosing the best item, we can maximize the expected rank of the chosen item.
- Perhaps the most immediate extension is to be choosing k elements.
- This has been generalized to the matroid secretary problem by Babaioff. For arbitrary matroids, the approximation ratio remains an open problem.
- Another natural extension is to generalize the selection of one item to the online (and ROM) edge weighted bipartite matching problem, where say $N = |L|$ items arrive online to be matched with items in R. In online matching the goal is usually to maximize the size (for the unweighted case) or weight of a maximum matching.
- I will next to discuss online matching and then later (hopefully) the extension to the adwords problem where the online nodes L represent advertisers/bidders with budgets and preferences/values for the R nodes representing keywords/queries.

The unweighted bipartite matching problem

Before leaving (for now) online, ROM and priority algorithms, I want to briefly discuss one more (surprising) ROM algorithm (equivalently for this algorithm, a randomized online algorithm) that has generated a good deal of recent research.

- Let $G = (U, V, E)$ be an unweighted bipartite graph with edges $E \subset U \times V$. Lets say that the vertices in U are the online vertices that arrive one at a time $u_1, \ldots u_n$, revealing the offline nodes in V to which they are adjacent.
- The online algorithm must irrecvocably decide whether and how to match u_i to an unmatched $v \in V$ (if there is such a node).
- It is easy to see that any greedy algorithm (i.e. one that matches each u_i if possible) produces a maximal matching and hence is a $\frac{1}{2}$-approximation (following the convention here for using fractional approximation ratios). This is also a tight bound for any deterministic online algorithm as can be seen by a simple 2×2 bipartite graph.

The Karp, Vazirani, Varizani (KVV) algorithm

- The KVV Ranking algorithm chooses a random permutation of the nodes in V and then when a node $u \in U$ appears, it matches u to the highest ranked unmatched $v \in V$ such that (u, v) is an edge (if such a v exists).
- Aside: making a random choice for each u is still only a $\frac{1}{2}$ approx.
- The analysis of this algorithm can be used to show that there is a deterministic greedy algorithm in the ROM model.
- That is, let $\{v_1, \ldots, v_n\}$ be any fixed ordering of the vertices and let the nodes in U enter randomly, then match each u to the first unmatched $v \in V$ according to the fixed order.
- To argue this, consider fixed orderings of U and V; the claim is that the matching will be the same whether U or V is entering online.

The KVV result and recent progress

KVV Theorem

Ranking provides a $(1 - 1/e) \approx .63$ approximation.

- Original analysis is not rigorous.
- There is an alternative proof (and extension) by Goel and Mehta [2008], and other proofs (e.g. in Birnbaum and Mathieu [2008], Devanur, Jain, Kleinberg [2013]).
- KVV show that the $(1 - 1/e)$ bound is essentially tight for any randomized online (i.e. adversarial input) algorithm. In the ROM model, Goel and Mehta state inapproximation bounds of $\frac{3}{4}$ (for deterministic) and $\frac{5}{6}$ (for randomized) algorithms.
- In the ROM model, Karande, Mehta, Tripathi [2011] show that Ranking achieves approximation at least .653 (beating $1 - 1/e$) and no better than .727.

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.
- NOTE: Defining "useful" precise models of DP algorithms is challenging.

What is 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 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"

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.