CSC2421 - Approximation Algorithms*
Lecture 4: Stack Model and Local Ratio Algorithms

Notes taken by Mea Wang

January 27, 2004

1 Review

In our previous lecture, we studied both fixed and adaptive ordering priority algorithms. The adversary provides a large finite number of items as input. The algorithm orders these items into a sequence. Subsequently, the adversary picks a subset of the ordered set as the actual input to the algorithm. The algorithm chooses the first item from the input and makes its decision, correspondingly the adversary then removes some or all items. The algorithm runs recursively and makes irrevocable decisions to construct a feasible solution. In the adaptive case, the algorithm can choose which item will be selected next in each iteration. We concluded that any priority algorithm for unweighted set cover problem cannot perform better than the approximation ratio of $\frac{logn+1}{2}$, where n is the number of elements in the universe.

2 Overview

The local ratio technique was first proposed by Bar-Yehuda and Even [BYE85], later extended by Bafna et al [BBF99], and recently extended again by Bar-Yehuda [BY00]. The local ratio algorithm tends to provide reasonably good approximation factors for many NP-hard problems, while being computationally not much more complex than a basic priority algorithm.

In this lecture, we will introduce a generalization of the priority algorithm model, which models the basic local ratio technique. This model extends the priority algorithm to a two-phase algorithm, in which the first pass (the push phase) irrevocably places items onto the stack, and the second pass (the pop phase) constructs the solution by popping items off the stack. Though complicated things can be done in the pop phase, the algorithm that we will study is a simple, greedy, stack popping algorithm. In contrast to the priority algorithm, the stack model does not force the stack to be a feasible solution. Instead, it prunes the stack items in the second phase to construct the final solution.

We refer to such model as the stack model, which is a meta algorithm with a parameter. We can vary the parameter to model different resource allocation problems. After introducing the general stack models for covering and packing problems, we will study example algorithms for each model. In particular, we will present a meta algorithm (with a parameter) for a general case of resource allocation problems. We can vary the parameter to model different resource allocation problems. We will show that this meta algorithm provides approximation ratio $1/5$ for the general case of resource allocation problems and an optimal algorithm for the $WISP_1$ problem.

* Lecture Notes for a course given by Allan Borodin, Dept. of Computer Sciecne, University of Toronto.

3 Steiner Tree/Forest Problem

In Lecture 1, we briefly considered the Steiner tree problem. We are given an undirected graph $G = (V, E)$ with nonnegative edge costs assigned by the weight function $w : E \to \mathbb{Q}^+$. The vertices are partitioned into two sets, *terminal* (T) and *Steiner*. A solution $S = (V', E')$, where $E' \subseteq E$ is feasible if it connects all terminal vertices and any subset of the Steiner vertices. The *Steiner tree problem* is to find a minimum cost tree S, with the objective function $COST(E') = \sum_{e \in E'} w(e)$. A *minimal Steiner tree* is a steiner tree F if $F \setminus \{e\}$ is not a Steiner tree for any edge $e \in F$. A Steiner tree is minimal if and only if every leaf in the tree is a terminal vertex.

The *Steiner forest problem* is defined as follows [Vaz98]. Given an undirected graph $G = (V, E)$, a weight function on edges $w : E \to \mathbb{Q}^+$, and a collection of disjoint subsets of V, $S_1, \cdots, S_k$, find a minimum cost subgraph in which each pair of vertices belonging to the same set S_i is connected. Similarly, a *minimal Steiner forest* is a steiner forest F if $F \setminus \{e\}$ is not a Steiner forest for all edge $e \in F$.

3.1 Stack Model for Covering Problem

The stack model fits the general scheme described in Sec. 2. During the push phase, the algorithm first initializes the $Stack$ to empty. Subsequently, it generates a solution to the problem by running an adaptive priority algorithm. Each item that is accepted is pushed onto the $Stack$. The push phase takes place when either all items have been considered or the $Stack$ contains a feasible solution. The items are popped from the $Stack$ in a last-in-first-out order. An item is rejected if the partial solution and the remaining items on the stack constitute a feasible solution with the absence of this item. The algorithm just described is as follows. S is the feasible solution returned by the algorithm.

Algorithm 3.1 (Adaptive Priority Stack Algorithm).

Push Phase:
  while ($\mathcal{I}$ is not empty & $Stack$ is infeasible)
    Ordering: Determine (without looking at I) a total ordering of all possible input items
    I_{next} := item in $\mathcal{I}$ that comes first in the ordering
    Decision: Decide whether to push I_{next} on to $Stack$, or throw it away
    Remove I_{next} from $\mathcal{I}$.
Pop Phase:
  $S := 0$
  while ($Stack$ is not empty)
    Pop the item i at the top of $Stack$.
    if ($Stack \cup S$ is feasible)
      reject this item.
    else
      $S := S \cup \{i\}$
  return S

Algorithm 3.1 is adaptive since the ordering of input items is computed in each iteration of the push phase based on the items that have been processed. The stack model is at least as powerful as the basic priority model because any priority algorithm can be simulated by running the priority algorithm in the push phase. The pop phase will simply accept the entire stack, since a priority

algorithm is not allowed to create an infeasible solution.

3.2 Approximation Algorithm for Steiner Tree/Forest

For an edge $e \in E$, we denote the number of terminals incident to e, or the *terminal degree* of e, by $\tau(e) = |e \cap T|$. The following is an example of local ratio algorithm [BYR01] that finds a minimal Steiner tree, a 2-approximation with respect to G, T, and w. The algorithm repeatedly contracts the edges in the graph until only one terminal vertex is left. Contracting an edge (u, v) consists of "fusing" its two endpoints u and v into a new vertex z. The edge connecting u and v is deleted and every other edge incident on u or v becomes incident on z. If either u or v are terminals then z is a terminal too.

Algorithm 3.2 (LR-ST (G,T,w)).

1. **if** G contains only one terminal then return ϕ
2. **else**
3. Let $\epsilon = min_e\{w(e)/\tau(e) \mid \tau(e) \neq 0\}$
4. Define the weight function $\delta(e) = \epsilon \cdot \tau(e)$
5. Let e be an edge such that $w(e) = \delta(e)$
6. Let (G', T') be the instance obtained by contracting e
7. $F' \leftarrow$ **LR-ST**$(G', T', w - \delta)$
8. **if** F' is not feasible
9. **return** $F = F' \cup \{e\}$
10. **else**
11. **return** $F = F'$

This recursive algorithm can be implemented in an iterative fashion using the stack model described in Sec. 3.1. The edge selection remains the same, but instead of recursing, the selected edge e is pushed onto a stack. When the remaining graph contains no terminal vertices, a solution is constructed by popping items from the stack. An edge e is thrown away if $F \setminus \{e\}$ is still a steiner tree.

The 2-approximation algorithm for Steiner forest shares the same structure with a new definition for the terminal degree of an edge e, $\tau(e) = |(e \cap S_1) \cup \cdots \cup (e \cap S_k)|$.

Up to now, we have an approximation algorithm which makes irrevocable decisions on what to push onto the stack. Instead of forming a solution immediately, we build a stack and pop it to construct the final solution. It is a generalization of the priority algorithm model for covering problems.

4 Job Bandwidth Allocation Problems

The resource allocation and scheduling algorithm presented in [BnFS$^+$00] is a generalization of WISP, WJISP, and Knapsack problems. We will refer to this class of problems as job bandwidth allocation problem (JBAP). The input of this problem consists of a set of *jobs*, $u_1, \cdots, u_n$. An instance $I \in u_j$ is defined by $I_j = (s_j, f_j, v_j, b_j, u_j)$. The half-open interval $[s_j, f_j)$ defines the *start-time* and *finish-time* of execution. Each instance utilizes certain amount, b_j, of the given, limited, bandwidth resource. The amount of resource available is fixed over time; we normalize it to unit size for convenience, that is $0 < b_j = \frac{bw_j}{bw_{max}} \leq 1$, where bw_j is the actual bandwidth

required by interval I_j and bw_{max} is the total available bandwidth. The profit gained by scheduling I_j is denoted by $v_j \geq 0$. [1] Different instances of the same job may have different parameter settings, that is $u_j = \{I_{j_1}, \cdots, I_{j_k}\}$.

A *schedule* S is a collection of instances. It is *feasible* if: (1) it contains at most one instance of every job, and (2) at any given time t, the total bandwidth of all scheduled intervals is at most 1, that is, for all t, $\sum_{I_j \in S, s_j \leq t < f_j} \leq 1, 0 \leq t \leq max(f_i)$. We call $\sum_{I_j \in S} v_j$ the *profit* of the set or the schedule S. A feasible schedule is *I-maximal* if either it contains instance I ($I \in S$), or it does not contain I but adding I to it will render it infeasible ($S \cup \{I\}$ is infeasible). The goal of JBAP is to construct a feasible set of instances that maximizes the profit of the schedule S. We sometimes consider the special case of JBAP in which each job consists of a single instance. In this case, the profit of a job is the same as the profit of its instance. We refer to this special case as the bandwidth allocation problem (BAP).

To model different problems, we simply put restrictions on the nature of the input. For example, to model the Knapsack problem, we require that all intervals have the same start time and end time ($s_j = 0, f_j = 1, \forall j$), and that there is only one interval per job. The bandwidth can be thought of as the weight that the item adds to the knapsack. To model the $JISP$, WISP$_1$ and $WJISP$ problems, we require that each interval consumes all given bandwidth, that is $b_j = 1, \forall j$. Note that whenever we have a "job" problem, we do not need more than 1 machine. We can use time to simulate more than one machine as in Fig. 1.

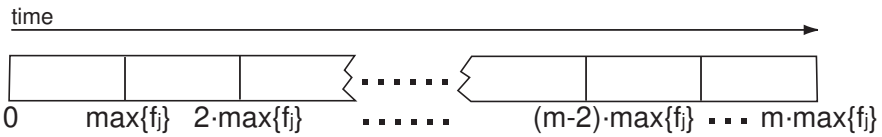

Figure 1: Simulating JISP on 1 machine.

To encode WISP$_m$ problem, we require each interval consume only a fixed portion of the given bandwidth, more precisely $b_j = 1/m, \forall j$. At most m intervals can be scheduled at the same time.

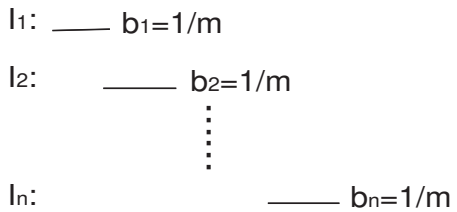

Figure 2: Simulating WISP$_m$ on 1 machine.

4.1 The Stack Model for Packing Problems

The model fits the general scheme described in Sec. 2. During the push phase, the algorithm sorts the input sequence $\mathcal{I}$ and initializes the $Stack$ to empty. Subsequently, it generates a (possibly infeasible) solution to the problem by running a priority algorithm. Each item that is accepted is pushed onto a stack. The pop phase takes place when all items have been considered. The items are popped from the stack in a last-in-first-out order, and are accepted greedily to construct a solution. Any item that would render the partial solution infeasible is thrown away, while every other item

[1] The variables v_j, b_j, and u_j are $p(I)$, $w(I)$, and $\mathcal{A}_j$ respectively in [BnFS$^+$00]

is accepted. The algorithm just described is as follows. S is the feasible solution returned by the algorithm.

Algorithm 4.1 (Priority Stack Algorithm).

Push Phase:
Ordering: Determine (without looking at $\mathcal{I}$) a total ordering of all possible input items
while ($\mathcal{I}$ is not empty)
I_{next} := item in $\mathcal{I}$ that comes first in the ordering
Decision: Decide whether to push I_{next} on to $Stack$, or throw it away
Remove I_{next} from $\mathcal{I}$.
Pop Phase:
$S := 0$
while ($Stack$ is not empty)
Pop the item at the top of $Stack$, and add it to S, if feasible.
return S

Algorithm 4.1 is a fixed satck algorithm since the ordering of input items remains unchanged throughout the push phase. Hwever, we can obviosuly formulate an adaptive ordering stack algorithm for packing problmes where the ordering of input items is computed in each iteration of the push phase based on the items that have been already processed.

4.2 Approximation Algorithm for JBAP

In this section, we present a meta algorithm with a parameter for all general JBAPs. Bar-noy [BnFS$^+$00] extends the local ratio technique to solve JBAPs; that is, given a profit vector $v \in \mathbb{R}^n$, find a solution vector x that approximately maximizes the inner product $(v \cdot x)$, subject to a given set of feasibility constraints on x. The local ratio algorithm for the bandwidth problem (and its variations) is as follows. All intervals are sorted in non-descending order according to their finish time, that is $f_1 \leq f_2 \leq \cdots \leq f_n$. Note that the profits of certain items may drop below zero as the "local profit" changes. These are the items to be deleted in the next recursive call, as it is obviously not desirable for an algorithm to schedule intervals with negative profits.

Algorithm 4.2 (LR-BANDWIDTH).

1. Delete all instances with non-positive profit.
2. If no instances remain, return the empty schedule.
3. Otherwise, select the interval $\tilde{I}$ with earliest end time, and decompose v by $v = v_1 + v_2$ as described below. Set $v_2(\tilde{I}) = 0$.
4. Solve the problem recursively using v_2 as the profit function. Let S' be the schedule returned by the recursive call.
5. If $S' \cup \{\tilde{I}\}$ is a feasible schedule, then return $S = S' \cup \{\tilde{I}\}$. Otherwise, return $S = S'$.

All that remains is to determine $\tilde{I}$ and to define the profit function v_1, and $v_2 = v - v_1$. We

choose $\tilde{I}$ and decompose v in such a way that: (1) $v_2(\tilde{I}) = 0$, and (2) for a certain r, which depends on the problem being solved, every $\tilde{I}$-maximal schedule will be an r-approximation with respect to v_1.

At each stage, we select an instance with minimum finish-time among all remaining instances as $\tilde{I}$, breaking ties arbitrarily. We introduce a parameter $\alpha > 0$, whose value varies depending on the particular case of the problem being solved. For a given instance $\tilde{I}$, let $\mathcal{A}(\tilde{I})$ be the job to which instance $\tilde{I}$ belongs, $\mathcal{A}(\tilde{I}) = \{I_j | u(I_j) = u(\tilde{I}), \forall j\}$, and let $\bigcap(\tilde{I})$ be the set of instances intersecting $\tilde{I}$ but belonging to jobs other than $\mathcal{A}(\tilde{I})$, $\bigcap(\tilde{I}) = \{I_j | I_j \cap \tilde{I} \neq \phi, I_j \notin \mathcal{A}(\tilde{I})\}$. For any interval I_j,

$$v_1(I_j) = v(\tilde{I}) \cdot \begin{cases} 1 & I_j \in \mathcal{A}(\tilde{I}) \\ \alpha \cdot b(I_j) & I_j \in \bigcap(\tilde{I}) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Algorithm 4.2 can easily be modeled by the stack model described in Sec. 4.1. Essentially, the modification at each stage is the same, but rather than recursing, the selected interval $\tilde{I}$ is pushed onto a stack, and the profits of all remaining intervals I_j are decremented by $v_1(I_j)$ as defined in Eqn. 1. After all items have either been pushed or reached non-positive profit, a solution is constructed by popping items off the stack. An item is accepted if it does not render the partial solution infeasible. It is not difficult to see that this iterative algorithm will produce the same solution as the recursive one given above.

5 Analysis of the Priority Stack Algorithm for JBAP

Before going to the detail analysis of Algorithm 4.2, we first take a look at some nice properties provided by the local ratio technique.

5.1 The Local Ratio Technique

Let $v \in \mathbb{R}^n$ be a *profit* vector, and let F be a set of feasibility *constraints* on the solutions. A vector $x \in \mathbb{R}^n$ is a *feasible* solution to a given problem instance (F, v) if it satisfies all of the constraints in F. Its *value* is the inner product $(v \cdot x)$. A feasible solution is *optimal* for a maximization (or minimization) problem if its value is maximal (or minimal) among all feasible solutions. A feasible solution is an *r-approximation* solution, if $p \cdot x \geq$ (or $\leq$) $r \cdot p \cdot x^*$, where x^* is the optimal solution. An algorithm is said to have a performance guarantee of r if it always computes r-approximate solutions. In this lecture, we use fractional approximation ratio for maximization problems, which means $0 < r \leq 1$.

Theorem 5.1. *(Local Ratio). Let F be a set of constraints and let v, v_1, and v_2 be profit vectors such that $v = v_1 + v_2$. Then, if x is an r-approximation solution with respect to (F, v_1) and with respect to (F, v_2), then it is an r-approximate solution with respect to (F, v).*

Proof. Let x^*, x_1^*, x_2^* be optimal solutions for maximization problem instances (F, v), (F, v_1), and (F, v_2) respectively. Then $v \cdot x = v_1 \cdot x + v_2 \cdot x \geq r \cdot v_1 \cdot x_1^* + r \cdot v_2 \cdot x_2^* \geq r \cdot (v_1 \cdot x^* + v_2 \cdot x^*) = r \cdot v \cdot x^*$. For minimization problems, simply replace $\geq$ with $\leq$. □

The Local Ratio Theorem applies to all constraint satisfaction problems in the above formulation. Using this theorem, and an appropriate division of profits, it is often quite easy to prove approximation factors of algorithms for particular problems.

5.2 Analysis

Now, we are ready to analyze the quality of the solution returned by Algorithm 4.2.

Lemma 5.2. *Let r be a constant. Suppose that the method for choosing $\tilde{I}$ and decomposing the profit function is such that: (1) $v_2(\tilde{I}) = 0$, and (2) for all profit functions v, every $\tilde{I}$-maximal schedule is an r-approximation with respect to v_1. Then, the schedule S returned by the algorithm is an r-approximation with respect to v.*

Proof. In the first step, instances of non-positive profit are deleted, which does not change the optimum value. Thus, it is sufficient to show that S is an r-approximation with respect to the remaining instances. We prove this by induction on the number of recursive calls.

basis: At the basis of the recursion, step 2, the schedule returned is optimal, hence an r-approximation, since no remaining instances.

induction step: The induction hypothesis is that the solution returned by the k^{th}, $k \geq 1$, recursive call S' is an r-approximation with respect to v_2. Let S be the solution returned by the $k+1$ recursion. Thus, $S' \subseteq S \subseteq S' \cup \{\tilde{I}\}$ and $v_2(\tilde{I}) = 0$. It follows that S is an r-approximation with respect to v_2. Since S is $\tilde{I}$-maximal, it is also an r-approximation with respect to v_1. By Theorem 5.1, S is an r-approximation with respect to v.

Therefore, the schedule S returned by the algorithm is an r-approximation with respect to v. □

To analyze the performance guarantee r, Lemma 5.3 argued that we only need to examine the value of r for every $\tilde{I}$-maximal schedule. To do so, we need to derive an upper bound v_{opt} on the optimum v_1-profit and a lower bound v_{max} on the v_1-profit of every $\tilde{I}$-maximal schedule. Both bounds are normalized by $v(\tilde{I})$, that is $v(\tilde{I}) \cdot v_{max}$ and $v(\tilde{I}) \cdot v_{opt}$ are the actual bounds. The ratio $r = \frac{v_{max}}{v_{opt}}$ is the lower bound on the performance guaranteed by Algorithm 4.2.

Lemma 5.3. *The approximation factor of the algorithm is at least $\frac{v_{max}}{v_{opt}}$ for all provable bounds v_{max} and v_{opt}, and v_{max} and v_{opt} can always be chosen such that this ratio is at least*

$$\frac{min\{1, \alpha \cdot max\{min\{b_j\}, 1 - max\{b_j\}\}\}}{1+\alpha}$$

Proof. In the optimal schedule, only instances in $\mathcal{A}(\tilde{I}) \cup \bigcap(\tilde{I})$ contribute their v_1-profits. Instances in $\mathcal{A}(\tilde{I})$ can contribute at most $v(\tilde{I})$ profit since a feasible schedule contains at most one instance of each job. Since $\tilde{I}$ was chosen as an instance with the earliest finish-time, all instances belonging to $\bigcap(\tilde{I})$ intersect with $\tilde{I}$ at some points close to $f(\tilde{I})$. Hence, the total bandwidth of the scheduled instances from this set can be at most 1, and they contribute at most $\alpha \cdot v(\tilde{I})$ to v_1-profit. Thus, the upper bound of the optimum v_1-profit is $v_{opt} = 1 + \alpha$.

An $\tilde{I}$-maximal schedule contains either an instance of $\mathcal{A}(\tilde{I})$ or a set $\mathcal{X} \neq 0$ consists of instances from $\bigcap(\tilde{I})$ that prevent $\tilde{I}$ from being added to the schedule. In the former case, the v_1-profit of the schedule is at least $v(\tilde{I})$. Let $min\{b_j\}$ and $max\{b_j\}$ be the lower and upper bounds, respectively, on the required bandwidth of all intervals. In the latter case, the total bandwidth of instances in $\mathcal{X}$ is at least $1 - b(\tilde{I}) \geq 1 - max\{b_j\}$, otherwise $\tilde{I}$ can be scheduled without violating the feasibility. The total bandwidth is also not less than $min\{b_j\}$ since $\chi \neq 0$. Since $s(I) < f(\tilde{I}) < f(I)$, $\mathcal{X} \subseteq \bigcap(\tilde{I})$ and the v_1-profit is at least $\alpha \cdot v(\tilde{I}) \cdot max\{min\{b_j\}, 1 - max\{b_j\}\}$. Thus, a "lower bound " [2] of an $\tilde{I}$-maximal schedule v_1-profit is $v_{max} = min\{1, \alpha \cdot v(\tilde{I}) \cdot max\{min\{b_j\}, 1 - max\{b_j\}\}\}$.

Therefore, the approximation factor of Algorithm 4.2 is $r = \frac{min\{1, \alpha \cdot max\{min\{b_j\}, 1 - max\{b_j\}\}\}}{1+\alpha} \leq 1$. □

[2] We will refer to negative results concerning the approximation ratio as lower bounds even though the bound is actually an upper bound on the ratio which we have defined here to be at most 1.

5.3 Performance Guarantee

Let's consider two extreme cases of the JBAP. The first case is where all instances are *narrow*, that is $b_j \leq 1/2, \forall j$. Thus, $max\{b_j\} \leq 1/2$ and $min\{b_j\} = 0$. We set $\alpha = 2$ so that the v_1-profit is maximized in Eqn. 1. Lemma 5.3 provides a performance guarantee of $1/3$. The second case is where all instances are *wide*, that is $b_j \geq 1/2, \forall j$. The problem is reduced to interval scheduling, where $\alpha = 1$, $max\{b_j\} = 1$ and $min\{b_j\} = 1/2$, since no pair of intersecting instances can be scheduled together. Lemma 5.3 provides a performance guarantee of $1/2$.

The general case of JBAP is a mix of narrow instances and wide instances. The algorithm solves it separately for the narrow instances and for the wide instances, and returns the solution of greater profit. The profit of this case is bounded from below when either the optimum for the narrow instances is at least $\frac{3}{5}OPT$, or the optimum for the wide instances is at least $\frac{2}{5}OPT$. The performance guarantee provided by the algorithm is at least

$$max\{\frac{1}{3} \cdot \frac{3}{5}OPT, \frac{1}{2} \cdot \frac{2}{5}OPT\} = \frac{1}{5}OPT$$

Recall that each job has only one instance in BAP. When all instances are narrow, $max\{b_j\} \leq 1/2$ and $min\{b_j\} = 0$. Since the v_1-profit of any $\tilde{I}$-maximal solution is at least $v(\tilde{I}) \cdot min\{1, \alpha(1 - b(\tilde{I}))$, we set $\alpha = \frac{1}{1-b(\tilde{I})}$ to maximize the v_1-profit. Correspondingly, the optimal v_1-profit is at most $v(\tilde{I}) \cdot max\{\alpha, 1 + \alpha(1 - b(\tilde{I}))$. Hence, $r = \frac{1}{2}$. In the case with all wide instances, we have $b_j = 1, \forall j$. Thus, $max\{b_j\} = min\{b_j\} = 1$, so $v_{max} = 1$. Since $\mathcal{A}(\tilde{I}) = \{\tilde{I}\}$, only one instance from $\mathcal{A}(\tilde{I}) \cap \bigcap(\tilde{I})$ may be scheduled in any feasible solution, hence $v_{opt} = 1$. We obtain a performance guarantee of 1. The profit of this case is bounded from below when either the optimum for the narrow instances is at least $\frac{2}{3}OPT$, or the optimum for the wide instances is at least $\frac{1}{3}OPT$. The performance guarantee of BAP provided by the algorithm is at least

$$max\{\frac{1}{2} \cdot \frac{2}{3}OPT, 1 \cdot \frac{1}{3}OPT\} = \frac{1}{3}OPT$$

In WISP$_1$, each job has only one instance and $b_j = 1, \forall j$. The problem is equivalent to the "wide" extreme case of BAP. We thus obtain a performance guarantee of 1. Hence, WIPS$_1$ can be solved optimally by Algorithm 4.2. However, this is not the case for WISP$_m$, $m \geq 2$, though we can get the optimal solution using dynamic programming and by a min cost/max flow based algorithm.

In the next lecture, we will prove that the local ratio algorithm will not achieve a bound better than $1/2$ even when all intervals are very narrow. In fact, the approximation ratio achieved by any fixed ordering stack algorithm can never achieve 1; however, as we will see, our "lower bound" quickly approaches 1 as $max\{b_j\}$ decreases.

References

[BBF99] Bafna, Berman, and Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIJDM: SIAM Journal on Discrete Mathematics*, 12, 1999.

[BnFS$^+$00] Amotz Bar-noy, Ari Freund, Baruch Schieber, Joseph (seffi) Naor, and Reuven Bar-yehuda. A unified approach to approximating resource allocation and scheduling. February 24 2000.

[BY00] R. Bar-Yehuda. One for the price of two: A unified approach for approximating covering problems. *Algorithmica*, 27:131–144, 2000.

[BYE85] R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Descrete Mathematics*, 25:27–46, 1985.

[BYR01] Reuven Bar-Yehuda and Dror Rawitz. On the equivalence between the primal-dual schema and the local-ratio technique. *Lecture Notes in Computer Science*, 2129:24–??, 2001.

[Vaz98] Vijay V. Vazirani. *Approximation Algorithms*. 1998.