CSC2421 - Algebraic and Combinatorial Techniques in Complexity Theory
Lecture 3

Instructor: Allan Borodin
Scribe: Shizhong Li
University of Toronto

Jan 20, 2005

Summary: In this lecture, we will show upper and lower bounds for adaptive priority algorithms on weighted interval scheduling problems with proportional profits. We consider priority algorithm analysis as a natural game between our algorithm and the adversary. The adversary presents a set of items. Our algorithm assigns a priority to each of the items in the set. The adversary then chooses a subset of the items on which our algorithm is executed. The next part of this lecture will discuss set cover/vertex cover problems.

1 Weighted Interval Scheduling Problems with proportional profits

Definition 1.1. $WISP_m$: weighted interval scheduling problem for m machines.

Definition 1.2. memoryless algorithm: an algorithm is memoryless if it can only use the information in the currently accepted set,on the other hand, an algorithm has memory if it can use the information both in the currently accepted set and the items previously rejected. Thus an algorithm with memory can possibly learn something from the complete history.

Theorem 1.3. *[1]* $\forall \epsilon > 0$, $\nexists$ *a* $(3 - \epsilon)$ *- approximation (adaptive) priority algorithm for* $WISP_1$ *with proportional profit.*

Proof. The adversary set consists of "long jobs" and "short jobs". The long jobs are shown in Figure[1]. There are $2q-1$ long jobs, with two jobs of size from $1, 2, ..., q-1$ and one job of size q. Every long job overlaps two other long jobs by ϵ, except the two end jobs. Additionally, the adversary will give each long job 3 non intersecting short jobs of size $\frac{p_i - 2\epsilon}{3}$, all included in the interval of the job(See figure[2]).

The algorithm assigns priorities to the jobs. Suppose the algorithm assigns J_1 the highest priority. Depending on the size of job J_1, there are four cases to consider:

- Case 1: J_1 is a short job.
- Case 2: J_1 has profit 1.
- Case 3: J_1 has profit $t, 1 < t < q$.
- Case 4: J_1 has profit q.

In anll cases, after the algorithm chooses J_1, the adversary will remove all intervals not used by the algorithm or the optimal assignment given below.

If Case 1 occurs(See figure[3]), the algorithm selects a small job of profit $\frac{p_i-2\epsilon}{3}$, OPT will select the job of profit p_i, then the ratio is $\frac{p_i}{(p_i-2\epsilon)/3}$.
If Case 2 occurs(See figure[4]), the algorithm selects a job with profit 1, OPT will select a long job of profit$=2$ and three small jobs with total profit $1-\epsilon$. The ratio is $\frac{2+(1-\epsilon)}{1}$.
If Case 3 occurs(See figure[5]), the algorithm selects a long job with profit t, OPT will select two long jobs of profit $t-1$ and $t+1$ and three small jobs with profits $\frac{t-2\epsilon}{3}$ each. The ratio is $\frac{(t-1)+(t+1)+(t-2\epsilon)}{t}$.
If Case 4 occurs(See figure[6]), the algorithm selects a long job with profit q, OPT will select two jobs of profit $q-1$ and small jobs with profits $\frac{q-2\epsilon}{3}$ each. The ratio is $\frac{2(q-1)+(q-2\epsilon)}{q}$.
In all these cases, the ratio are arbitrarily close to 3 by making ϵ sufficiently small and q sufficiently large.

□

Corollary 1.4. *As a corollary, we can show* $\forall\epsilon > 0$, $\nexists$ *a* $(3-\epsilon)$ *- approximation fixed, greedy priority algorithm for* m *machines.*

Next we show an algorithm called CHAIN-2 which works on 2 machines and yields a better upper bound of approximation ratio of 2. (See figure[7]).

Theorem 1.5. *[1] The algorithm CHAIN-2 is a* $2-$*approximation for* $m = 2$ *for interval scheduling with proportional profit.*

Definition 1.6. CHAIN-2 algorithm: Let $m = 2, S = \{J_1, J_2, ..., J_n\}$ be an input set, and let $f_j = s_j + p_j$ be the finish time of job j, where s_j is the release time and p_j is the processing time of job j. The algorithm will pick up a job with the earliest release time, if ties it picks the longest precessing time.We say a job J_k *properly extends* a job J_j if $s_k \leq f_j < f_k$. If there is a job properly extending job $J_{n(i)}$, we choose one having the latest completion time and denote it $J_{n(i+1)}$. If there is no job properly extending $J_{n(i)}$, then we start a new phase by choosing a job with the earliest release time larger than $f_{n(i)}$. If we can not start a new phase, then the algorithm terminates.

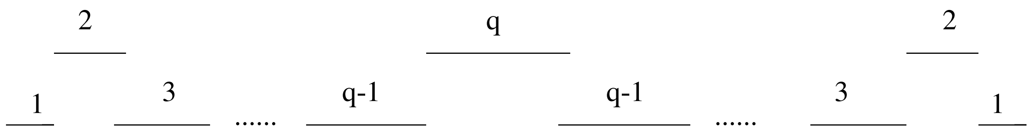

Figure 1: The ”long jobs” from the worst case sequence for any priority algorithm for m=1

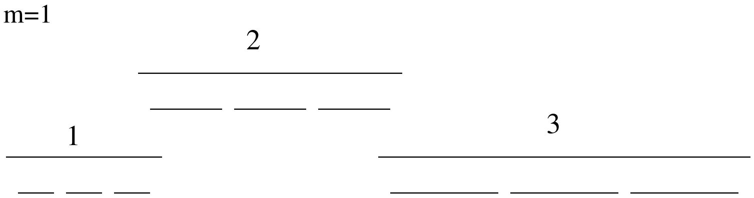

Figure 2: each long job has 3 non intersecting short jobs

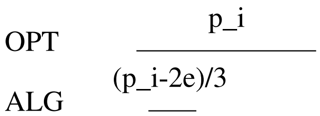

Figure 3: Case 1: algorithm selects a small job

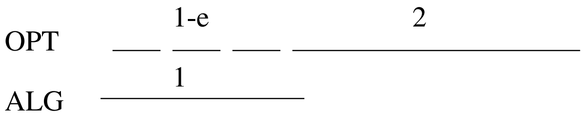

Figure 4: Case 2: algorithm selects a job with profit=1

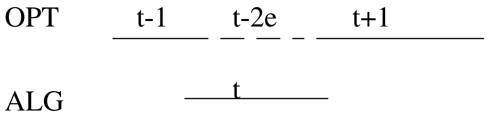

Figure 5: Case 3: algorithm selects a job with profit=t, $1 < t < q$

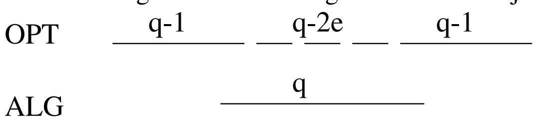

Figure 6: Case 4: algorithm selects a job with profit=q

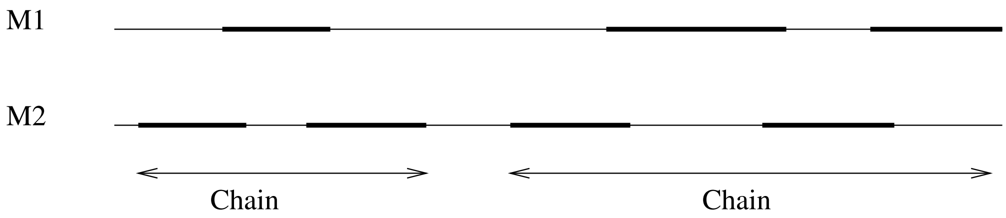

Figure 7: The input instance for algorithm CHAIN-2

Before we prove the theorem, we first prove the following lemma. In the lemma, we denote $d(S,t) = |\{J_i|[t, t+\delta t) \subseteq [s_i, f_i)\}|$, which is the number of possible jobs covering the time slot t. We also denote $C_2(S,t)$ the number of jobs the algorithm CHAIN-2 has scheduled covering time $[t, t+\delta t)$, we also call it slot t. Since $m = 2$, $C_2(S,t) \in \{0, 1, 2\}$.

Definition 1.7. a time slot t **is covered**: suppose time period t is $[t, t+\delta t)$, there $\exists$ a job J_i whose release time is s_i and finish time is f_i such that $s_i \leq t < t+\delta t \leq f_i$.

Lemma 1.8. *[1] For* $m = 2$*,* $\forall t$*: if* $d(S,t) \geq 1$*, then* $C_2(S,t) \geq 1$.

Proof. Suppose some job J_k covers time slot t. If algorithm CHAIN-2 schedules this job J_k, then clearly the time slot t is covered by a job by CHAIN-2.
If J_k is not scheduled by the CHAIN-2, we prove by cases that CHAIN-2 covers this slot t.

1. For some $i \geq 1, s_{n(i)} \leq s_k \leq f_k \leq f_{n(i)}$. In this case, $J_{n(i)}$ covers slot t.
2. For some $i \geq 1$, J_k properly extends $J_{n(i)}$. If $t \leq f_{n(i)}$, slot t is clearly covered by CHAIN-2. Otherwise, by the way CHAIN-2 chooses $J_{n(i+1)}$ in a phase, $f_k \leq f_{n(i+1)}$ and again slot t is covered.
3. For some $i \geq 1$, $f_{n(i)} < s_k < s_{n(i+1)}$. Since $f_{n(i)} < s_{n(i+1)}$, a new phase has started and $s_k < s_{n(i+1)}$ contradicts the way a job is chosen to start a new phase.
4. CHAIN-2 has scheduled a total of q jobs and $f_n(q) < s_k$. But then J_k would have been scheduled so that there would have been more than q jobs scheduled.

□

Lemma 1.8 shows us if OPT is busy at time slot t, algorithm CHAIN-2 will assign a job to make at least one machine busy at time slot t. Now lets prove Theorem 1.5 by using Lemma 1.8 .

Proof. We let $OPT(S,t)$ denote the number of jobs scheduled by OPT covering time slot t. It is clearly $d(S,t) \geq OPT(S,t)$. From Lemma 1.8, we get $2C_2(S,t) \geq OPT(S,t)$ for $\forall t$. Since $OPT(S) = \sum_t OPT(S,t)$, we have $2C_2(S) \geq OPT(S)$. Thus we proved CHAIN-2 is 2-approximation algorithm. □

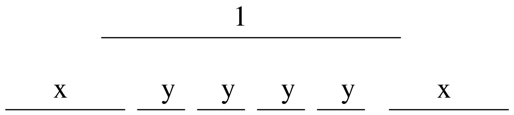

Figure 8: the lower bound of interval scheduling with proportional profits is 1.56

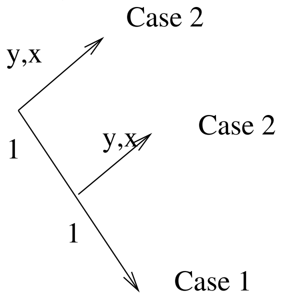

Figure 9: Two cases for Theorem 1.9

We will prove a lower bound for $m = 2$ for interval scheduling with proportional profits.

Theorem 1.9. *For $m = 2$, no ADAPTIVE PRIORITY GREEDY algorithm is a ρ-approximation algorithm for interval scheduling with proportional profit for $\rho < 1 + \frac{\sqrt{17}-3}{2} \approx 1.56$.*

Proof. For the case shown in figure[8], we have four y jobs and two x jobs where $y = \frac{1-2\epsilon}{4}, y < x$, for any small $\epsilon > 0$.The job of size 1 overlaps job x by size ϵ. There are two cases to consider(See figure[9]):

- Case 1: The algorithm picks two jobs of size 1 on two machines. The OPT will pick two groups of $2x$ and $4y$ jobs. Thus the ratio is $\frac{2((1-2\epsilon)+2x)}{2} = 1 + 2x - 2\epsilon$.
- Case 2: The algorithm picks one job of size 1 and at most one x(by $y < x$). The OPT will pick two jobs of size 1. Thus the ratio is $\frac{2}{1+x}$.

We equate these two equations and let $\epsilon \to 0$, we get $\frac{2}{1+x} = 1 + 2x - 2\epsilon \implies x = \frac{\sqrt{17}-3}{4} \approx 1.56$. $\square$

Next we consider one-pass algorithms with revocable acceptances and show a lower bound for this model.

Definition 1.10. One-pass algorithm with revocable acceptances: when a one-pass algorithm with revocable acceptance processes an input item, it can decide to reject a set of previously accepted items so as to accept the current item to make it a feasible solution.

The difference between priority algorithms and one-pass algorithms with revocable acceptances is that the latter can reject items already accepted in order to get a better solution. The following is a one-pass algorithm based on the order of the non-decreasing finish times:

Algorithm 1.11 (One-pass based on the order of NON-DECREASING FINISH TIME).

sort intervals in order of non-decreasing finish time
let S be the current set of accepted intervals. Initially $S = \emptyset$
for interval i, the next interval to be processed **do**
  let C_i be a minimum-weight subset of S such that $(S \backslash C_i) \cup \{i\}$ is feasible
  if $w(C_i) \leq \alpha w(i)$ **then**
    $S = (S \backslash C_i) \cup \{i\}$
  end if
end for

Note: $\alpha \in (0, 1)$ is a parameter that can be optimized for different variations for the problem. For $WISP_1$, setting $\alpha = 1/2$ results in a 4-approximation algorithm.

Theorem 1.12. *[2] For $WISP_1$ with equal lengths and arbitrary weights (or for arbitrary lenghts and proportional profit), any one-pass algorithm with revocable acceptance cannot have an approximation ratio better than 1.17.*

Proof. We consider an instance of 5 intervals $\{I_1, I_2, I_3, I_4, I_5\}$ (See figure[10]). Suppose

- $w(I_1) = w(I_5) = \alpha$
- $w(I_2) = w(I_4) = \beta$
- $w(I_3) = \gamma$
- $\gamma < \beta$
- $\beta < \alpha + \gamma$
- $2\alpha + \gamma < 2\beta$

We consider a game between the adversary and the algorithm. We show the game sequences in different cases. At the beginning, the adversary gives a set of $\{I_1, I_2, I_3, I_4, I_5\}$. Then the algorithm orders the items by different ways(in different cases below). We denote adversary as *ADV*, algorithm as *ALG*:

1. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 and accepts it, thus has to reject I_3 $\Longrightarrow$ *ADV* removes I_4 $\Longrightarrow$ *ALG* picks I_5, thus algorithm picks $\{I_2, I_5\}$.

2. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 but rejects it $\Longrightarrow$ *ADV* removes I_1 and I_5 $\Longrightarrow$ *ALG* sees I_4 and accepts it, thus has to rejects I_3, thus algorithm picks $\{I_4\}$.

3. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 and accepts it, thus has to removes I_1 and I_3 $\Longrightarrow$ *ADV* removes I_4 and I_5, thus algorithm picks $\{I_2\}$.

4. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 but rejects it $\Longrightarrow$ *ADV* removes I_5 $\Longrightarrow$ *ALG* sees I_4 and accept it,thus has to remove I_3, therefore algorithm picks $\{I_1, I_4\}$.

5. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_4 and accepts it, thus has to reject I_3 $\Longrightarrow$ *ADV* removes I_2, thus algorithm picks $\{I_1, I_4\}$.

6. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_4 but rejects it $\Longrightarrow$ *ADV* removes I_5, thus algorithm picks $\{I_1, I_3\}$.

7. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_5 and accepts it, thus algorithm picks $\{I_1, I_3, I_5\}$.

8. *ALG* sees I_3 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_5 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 and accepts it, thus has to remove I_1, I_3 $\Longrightarrow$ *ADV* removes I_4, thus algorithm picks $\{I_2, I_5\}$.

9. *ALG* sees I_2 and accepts it $\Longrightarrow$ *ADV* removes I_4, I_5 $\Longrightarrow$ *ALG* picks the item from $\{I_1, I_3\}$ with maximum weight, thus has to reject I_2, therefore algorithm picks $\{I_1 or I_3\}$.

10. *ALG* sees I_2 and accepts it $\Longrightarrow$ *ADV* removes I_4, I_5 $\Longrightarrow$ *ALG* sees I_1 but rejects it, thus algorithm picks $\{I_2\}$.

11. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 and accepts it, thus has to reject I_1 $\Longrightarrow$ *ADV* removes I_4, I_5, thus algorithm picks $\{I_2\}$.

12. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 but rejects it $\Longrightarrow$ *ADV* removes I_3, I_4, I_5, thus algorithm picks $\{I_1\}$.

13. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_4 and accepts it $\Longrightarrow$ *ADV* removes I_2 $\Longrightarrow$ *ALG* picks the item from $\{I_3, I_5\}$ with maximum weight, thus has to reject I_4, therefore algorithm picks $\{I_1, I_3 or I_5\}$.

14. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_4 and accepts it $\Longrightarrow$ *ADV* removes I_2 $\Longrightarrow$ *ALG* removes I_3 or I_5, thus algorithm picks $\{I_1, I_4\}$.

15. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_5 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 and accepts it, thus has to reject I_1 $\Longrightarrow$ *ADV* removes I_4, thus algorithm picks $\{I_2, I_5\}$.

16. *ALG* sees I_1 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_5 and accepts it $\Longrightarrow$ *ADV* does nothing and lets the *ALG* move $\Longrightarrow$ *ALG* sees I_2 but rejects it $\Longrightarrow$ *ADV* removes I_3 $\Longrightarrow$ *ALG* sees I_4 and accepts it, thus has to removes I_5, therefore algorithm picks $\{I_1, I_4\}$.

The next table shows the actual input(after the adversary removed the items), the optimal and algorithm's profit and its ratios. The case number corresponds to the previous cases.

CASE	INPUT	ALG	OPT	ratio
1	$\{I_1, I_2, I_3, I_5\}$	$w(I_2) + w(I_5)$	$w(I_1) + w(I_3) + w(I_5)$	$\frac{2\alpha+\gamma}{\alpha+\beta}$
2	$\{I_2, I_3, I_4\}$	$w(I_4)$	$w(I_2) + w(I_4)$	2
3	$\{I_1, I_2, I_3\}$	$w(I_2)$	$w(I_1) + w(I_3)$	$\frac{\alpha+\gamma}{\beta}$
4	$\{I_1, I_2, I_3, I_4\}$	$w(I_1) + w(I_4)$	$w(I_2) + w(I_4)$	$\frac{2\beta}{\alpha+\beta}$
5	$\{I_1, I_3, I_4, I_5\}$	$w(I_1) + w(I_4)$	$w(I_1) + w(I_3) + w(I_5)$	$\frac{2\alpha+\gamma}{\alpha+\beta}$
6	$\{I_1, I_2, I_3, I_4\}$	$w(I_1) + w(I_3)$	$w(I_2) + w(I_4)$	$\frac{2\beta}{\alpha+\gamma}$
7	$\{I_1, I_2, I_3, I_4, I_5\}$	$w(I_1) + w(I_3) + w(I_5)$	$w(I_2) + w(I_4)$	$\frac{2\beta}{2\alpha+\gamma}$
8	$\{I_1, I_2, I_3, I_5\}$	$w(I_2) + w(I_5)$	$w(I_1) + w(I_3) + w(I_5)$	$\frac{2\alpha+\gamma}{\alpha+\beta}$
9	$\{I_2, I_j \mid j = 1or3\}$	$max(w(I_1), w(I_3))$	$w(I_2)$	$min(\frac{\beta}{\alpha}, \frac{\beta}{\gamma})$
10	$\{I_1, I_2, I_3\}$	$w(I_2)$	$w(I_1) + w(I_3)$	$\frac{\alpha+\gamma}{\beta}$
11	$\{I_1, I_2, I_3\}$	$w(I_2)$	$w(I_1) + w(I_3)$	$\frac{\alpha+\gamma}{\beta}$
12	$\{I_1, I_2\}$	$w(I_1)$	$w(I_2)$	$\frac{\beta}{\alpha}$
13	$\{I_1, I_4, I_j \mid j = 3or5\}$	$w(I_1) + max(w(I_3), w(I_5))$	$w(I_1) + w(I_4)$	$min(\frac{\alpha+\beta}{\alpha+\gamma}, \frac{\alpha+\beta}{2\gamma})$
14	$\{I_1, I_3, I_4, I_5\}$	$w(I_1) + w(I_4)$	$w(I_1) + w(I_3) + w(I_5)$	$\frac{2\alpha+\gamma}{\alpha+\beta}$
15	$\{I_1, I_2, I_3, I_5\}$	$w(I_2) + w(I_5)$	$w(I_1) + w(I_3) + w(I_5)$	$\frac{2\alpha+\gamma}{\alpha+\beta}$
16	$\{I_1, I_2, I_4, I_5\}$	$w(I_1) + w(I_4)$	$w(I_2) + w(I_4)$	$\frac{2\beta}{\alpha+\beta}$

The input is the actual input (after some items removed by the adversary). Take an example of the case in the first row of input $\{I_1, I_2, I_3, I_5\}$. Both of the algorithm and the OPT can only select items from $\{I_1, I_2, I_3, I_5\}$. Thus the algorithm can select at best $\{I_2, I_5\}$, but OPT selects $\{I_1, I_3, I_5\}$. The ratio is $\frac{2\alpha+\gamma}{\alpha+\beta}$.

From the table, we can see the lower bound is decided by three ratios: $\frac{2\beta}{2\alpha+\gamma}, min(\frac{\alpha+\beta}{\alpha+\gamma}, \frac{\alpha+\beta}{2\alpha}), \frac{2\alpha+\gamma}{\alpha+\beta}$. The intersection of the three ratios produces the best lower bound. We define a line segment as:

1. $\alpha = kx$

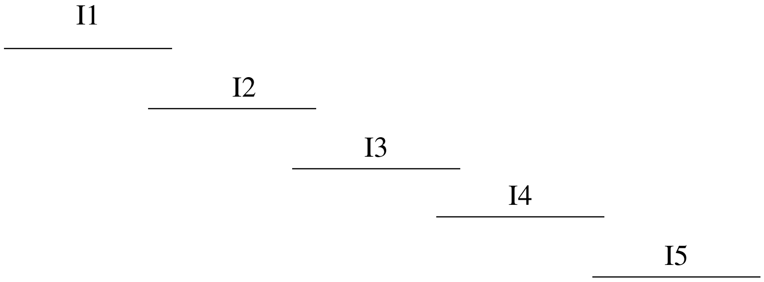

Figure 10: interval selection instance shows a lower bound of approximation ratio 1.17

2. $\beta = k(2k+1)x$

3. $\gamma = x$

where $k \approx 0.58504$ is the only positive solution to $4k^3+4k^2-2k-1=0$, and $x > 0$ in the line segment. When $x = 1$, we get the solution $(\alpha, \beta, \gamma) = (0.58504, 1.26959, 1)$ and all three ratios approache 1.17. $\square$

2 Set Cover/Vertex Cover problems

We define the Weighted Set Cover Problem as follows: a domain $U = \{u_1, u_2, ..., u_m\}$ with m elements, a set $\mathcal{S} = \{S_1, S_2, ..., S_n\}$, each S_i has a weight $w(S_i) = W_i$. We want to find a subset of $\mathcal{C} \subseteq \mathcal{S}$ where $\cup_{S_j \in \mathcal{C}} S_j = U$ to minimize $\sum_j w(C_j)$.
We consider the following algorithm:

Algorithm 2.1 (Weighted Set Cover).
Input: $\mathcal{C} = \emptyset, U = \{u_1, u_2, ..., u_m\}, \mathcal{S} = \{S_1, S_2, ..., S_n\}, R = U$ where R is a set of uncovered elements.

while $U \neq \emptyset$
  pick S_i in $\mathcal{S}$ such that $S_i \cap U \neq \emptyset$ and $\frac{w(S_i)}{|R \cap S_i|}$ is minimal
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{S_i\}$
  $R \leftarrow R - \{S_i\}$
end while
return $\mathcal{C}$

We have a conclusion that this algorithm is a H_d-approximation algorithm where $H_d = \sum_{i=1}^{d} \frac{1}{i}$, $d = max|S_i|$, $H_d \approx ln(d)$.
Before we prove this theorem, we first prove the following lemma.

Lemma 2.2. *For any $S_k, w_k \geq \frac{\sum_{u \in S_k} c_u}{H_{|S_k|}} = \frac{\sum_{u \in S_k} c_u}{H_{|d|}}$, where $c_u = \frac{w(S_i)}{|R \cap S_i|}$, S_i is the first set in algorithm to cover u.*

Proof. Let $u_1, u_2, ..., u_{|S_k|}$ be the ordered sequence in which elements of S_k are covered. Lets consider u_j and suppose u_j is covered for the first time by S_i.
$|S_k \cap R| \geq |S_k| - j + 1$. [By $|S_k \cap R|$ also contains the elements from other set S_t besides S_k]
$c_{u_j} = \frac{w(S_j)}{|S_k \cap R|} \leq \frac{w(S_j)}{|S_k|-j+1} \leq \frac{w(S_j)}{d-j+1}$ (By $d = max(|S_t|) \geq |S_k|$)
$\sum_{j=1}^{|S_k|} c_{u_j} \leq \sum_{j=1}^{|S_k|} \frac{w(S_k)}{|S_k|-j+1} = w(S_k) \sum_{j=1}^{|S_k|} \frac{1}{j} = w(S_k)H(|S_k|) \leq w(S_k)H_d \Longrightarrow$
$w(S_k) \geq \frac{\sum_{u \in S_k} c_u}{H_d}$. □

We will use Lemma 2.2 to prove our main theorem.

Theorem 2.3. *The "natural greedy algorithm" for weighted set-cover problem is an H_d-approximation algorithm where $H_d = \sum_{i=1}^{d} \frac{1}{i}$, $d = max|S_i|$, $H_d \approx ln(d)$.*

Proof. The cost of our algorithm $w(Greedy) = \sum_{u \in U} c_u$
Suppose $\mathcal{C}^* = \{S_1, S_2, ..., S_n\}$ is an optimal solution.

$$\begin{array}{rcll} w(OPT) & = & \sum_{k:S_k \in \mathcal{C}^*} \frac{\sum_{u \in S_k} c_u}{H_d} & \\ & = & \frac{1}{H_d} \sum_{k:S_k \in \mathcal{C}^*} \sum_{u \in S_k} c_u & \\ & \geq & \frac{1}{H_d} \sum_{u \in U} c_u & \text{(by } C^* \text{ is a cover)} \\ & = & \frac{1}{H_d} w(Greedy). & \end{array}$$

□

We will give an instance of set cover problem with a lower bound of H_k by Algorithm 2.1.

Theorem 2.4. *The natural set cover greedy algorithm when applied to (the special case of) the vertex-cover problem (See figure[11]) has a lower bound of H_k.*

Proof. This is a bipartite graph: on the left side we have a set U_0 with $k!$ nodes of degree k, on the right side we have k groups of nodes $V_k, V_{k-1}, ..., V_1$. In group V_i, we have $\frac{k!}{i}$ nodes with degree i.There are $k!$ edges going out of each group on the right side, thus total $k * k!$ edges out of right side. There are total $k!$ edges out of left side. Each node of U_0 connects to only one node in each group $V_i, i \in \{1, ..., k\}$.
Our algorithm picks up nodes to our vertex cover set from the nodes with the largest degree.Both U_0 and V_k has the largest degree of k. If unfortunately we pick up the nodes from V_k, then we must pick V_{k-1} which has degree $k-1$,...,in the end we pick the nodes from group V_0 which has degree 1.
Thus our vertex cover has nodes: $\frac{k!}{k} + \frac{k!}{k-1} + ... + \frac{k!}{1} = H_k k!$.
But obviously U_0 is a vertex cover with only $k!$ nodes.
Hence our ratio is $\frac{H_k k!}{k!} = H_k$. □

Finally we show a logarithmic lower bound for priority algorithms applied to the set cover problem.

Theorem 2.5. *[3] There is no priority algorithm for unweighted set cover problem with approximation ratio better than $\frac{log(n)+1}{2} < H_n$.*

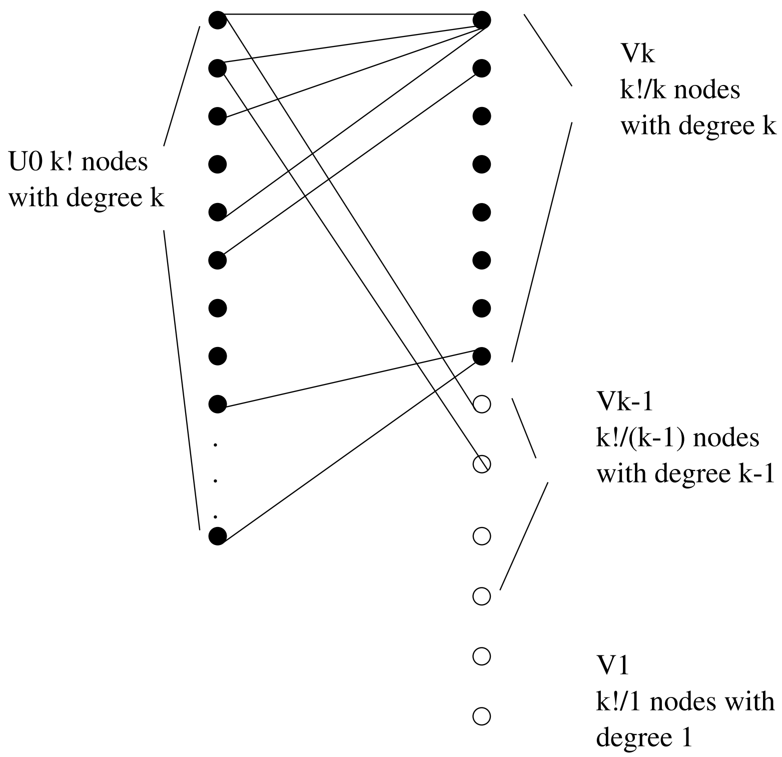

Figure 11: vertex cover with approximate ratio H_n

We define
a set $\mathcal{S} = \{S_1, S_2, ..., S_{2(n-1)}\}$
$\mathcal{U} = \{A | A \subseteq \{1, 2, ..., log(n)\}\}$
$|\mathcal{U}| = n$
$1_A \in \{0, 1\}^{log(n)}$, in which i^{th} bit is 1 if $i \in A$, and 0 otherwise
$v \in \{0, 1\}^{log(n)}$
$S_v^0 = \{A | < 1_A, v >= 0\}$ and $S_v^1 = \{A | < 1_A, v >= 1\}$
Thus $S_v^0 \cap S_v^1 = \emptyset$ and $S_v^0 \cup S_v^1 = \mathcal{U}$
To prove Theorem 2.5, we first prove a lemma:

Lemma 2.6. *For linearly independent vectors $\{v_1, v_2, ..., v_k\}$, the set $\mathcal{S} = \{S_{v_1}^{b_1}, ..., S_{v_k}^{b_k}\}$, after k iterations, there are still $\frac{|\mathcal{U}|}{2^k}$ uncovered elements in $\mathcal{S}$.*

Proof. Given the covered elements $S_{v_1}^{b_1} \cup ... \cup S_{v_k}^{b_k}$, the uncovered elements are $S_{v_1}^{1-b_1} \cap ... \cap S_{v_k}^{1-b_k}$.
The uncovered elements satisfies $< A, v_i >= 1 - b_i$.
Since the vectors $\{v_1, ..., v_k\}$ are linearly independent, we have a system of k linearly independent equations in $log(n)$ dimensions. The dimension of the solution space is $log(n) - k$, thus the number of solutions A to this system is $|F|^{log(n)-k} = 2^{log(n)-k} = \frac{n}{2^k}$. That's exactly the number of uncovered elements. ☐

We will give a sketch proof of Theorem 2.5.

Proof. Sketch: For the unweighted set cover problem described above, we will show no prioirty algorithm (and furthermore, no stack algorithm as described in the next lecture) can achieve a set cover of size smaller than $log(n) + 1$ and the optimal set cover will always be of size 2.
We claim that the algorithm can never afford to reject any of the first $log(n)$ sets that it sees. If at stage $i < log(n)$, the algorithm rejects the set $S_{v_i}^{b_i}$. Then the adversary deletes all unseen inputs except for $S_{v_i}^{1-b_i}$. Now the algorithm can only choose $S_{v_i}^{1-b_i}$. Since $\{v_1, v_2, ..., v_i\}$ is linearly independent and $i < log(n)$, by Lemma 2.6, there are at least $\frac{n}{2^i} \geq 1$ uncovered items. Thus it can not achieve a set cover. However the optimal solution is $\{S_{v_i}^{b_i}, S_{v_i}^{1-b_i}\}$ and the size of it is 2.
After $log(n)$ steps. the algorithm has accepted $log(n)$ sets whose defining vectors $\{v_1, ..., v_n\}$ are linearly independent. By lemma 2.6, there is exactly $\frac{n}{2^{log(n)}} = 1$ uncovered element. Let $v^* = \sum_{i=1}^{log(n)} v_i$, the adversary deletes all sets except $S_{v^*}^0$ and $S_{v^*}^1$ which is a set cover. Since only one element is left uncovered, and $S_{v^*}^0$ and $S_{v^*}^1$ are disjoint, then one of these two sets is already covered by the sets on the stack, thus one of them will be rejected during the pop phase, the other one will be accepted by the algorithm. Now the algorithm has $log(n) + 1$ sets on the stack which is actually a set cover. During the pop phase, the algorithm will accept all the $log(n) + 1$ sets, hhile the optimal solution is of size 2. Hence the ratio is $\frac{log(n)+1}{2}$. ☐

References

[1] Allan Borodin, Morten N.Nielsen, and Charles Rackoff. (Incremental)Priority algorithms. In `Algorithmica 37(4)`, pages 295-326,2003

[2] Stephanie Lorraine Horn. One-pass algorithms with revocable acceptances for job interval selection. Master's thesis, University of Toronto, Jan 2004

[3] David Cashman. Approximate Truthful mechanisms for the knapsack problem, and negative results using a stack model for local ratio algorithms. Master's thesis, University of Toronto, Jan 2005