The combination of bounds method

Russell Impagliazzo

January 26, 2004

1 The general method

Sometimes very simple approximation algorithms are tricky to analyze. The combination of bounds method is one approach. In this method, we compare the solution our algorithm finds to the optimal solution indirectly, through one or more intermediaries called bounds. A bound can be a simple function of the instance, or perhaps arises from a more complex structure that is only revealed after the algorithms run.

This method is also used for exact algorithms, but comes into its own for approximation algorithms. A good example of a simple bound argument for approximation algorithms is a greedy interval scheduling algorithm, that needs to schedule a set of events (each with an interval of time) in a different room. The algorithm schedules each event, from first to last, in the first available room. The bound is the number of events that occur at the same time; any solution must use this number of rooms. The algorithm achieves this bound, since room k is only used if the previous $k-1$ rooms are used, and a new event has just started.

A more complex function, only revealed after the algorithm finishes, arises in the Ford-Fulkerson max flow algorithms. The set of bounds used are the cuts in the graph; no flow can be larger than any cut. Then we show that, when the algorithm terminates, from the flow, we can compute a cut of the same cost as the flow's value. Thus, the algorithm's solution is at least as good as an upper bound on that solution. But the upper bound (the min cut) only becomes apparent after the algorithm has found the good solution (the max flow).

In using this method for approximation algorithms, we often combine more than one type of bound. Say we have a minimization problem. We might define $LB_1, ...LB_k$ as functions of the instance, and prove that for each i, $Cost(S) \geq LB_i$ for any solution S. Then we find $\alpha_1, ..\alpha_k$ and prove that, for our algorithm's solution $A(I)$ $Cost(A(I)) \leq \alpha_1 LB_1 + ...\alpha_k LB_k$. The conclusion is that $Cost(A(I)) \leq (\alpha_1 + ...\alpha_k) Cost(S)$ for the optimal solution S, and so our algorithm has approximation guarantee $\alpha_1 + ...\alpha_k$. It's the fact that we are allowing the sum of the α_i to be greater than one that gives us more flexibility in using this type of argument for approximation algorithms, because once we start combining different bounds, the sum is usually bigger than one.

We frequently use the above type reasoning inside a proof-by-cases, possibly using different bounds and α_i's in each case.

We also have arguments similar to the flow case, above, where the algorithm itself steers us to the right bound to use. There, we define a family F of bounds, and prove for every S and $L \in F$, $Cost(S) \geq L$. We use the algorithm to find $L_1, ..L_k \in F$, and α_i, so that $Cost(A(I)) \leq \alpha_1 L_1 + ... + \alpha_k L_k \leq (\alpha_1 + ...\alpha_k)Cost(S)$ for any solution S.

2 Parallel job scheduling

Let the parallel job scheduling problem be as follows:

Instance: A positive integer number of processors available, p; A list of n jobs $Job_1, ..Job_n$, each with a positive integer *duration* d_i, and a integer *processor requirement* p_i with $1 \leq p_i \leq p$.

Solution format Assign each job a non-negative integer start time s_i (implicitly defining the finish time $f_i = s_i + d_i$)

Constraint We can't require more than p processors at any one time, i.e., for all $t \geq 0$, $\sum_{i | s_i \leq t < f_i} p_i \leq p$.

Objective Mimimize the maximum completion time, $max_i f_i$.

This problem is NP hard to solve exactly. For example, when the durations are all 1, this is exactly the bin packing problem, so we cannot do any better for this problem than for bin-packing.

2.1 A greedy algorithm

A simple greedy approximation algorithm for this problem is:
Largest Processor Requirement Packing.

1. $t \leftarrow 0$. $used \leftarrow 0$, $OnNow \leftarrow \emptyset$ $Undone \leftarrow Jobs[1..n]$.
2. Repeat until $Undone = \emptyset$
3. Repeat until no jobs $Job_i \in Undone$ have $p_i \leq p - used$
4. Let Job_i be the job in $Undone$ with largest $p_i \leq p - used$.
5. Schedule Job_i at $s_i = t$ to $f_i = t + d_i$. Delete it from $Undone$, add it to $OnNow$. Add p_i to $used$.
6. Let Job_j be the first job to finish in $OnNow$.

7. Delete all jobs Job_k that finish at time f_j from $OnNow$ and subtract p_k from $used$

8. $t \leftarrow f_j$.

2.2 Two lower bounds on the optimal time

In order to use the combine bounds method, we need to define some bounds, simple functions of the input, and show that no solution can be better than these bounds. Then we'll show that the algorithm above has a solution that is upper bounded by a combination of these bounds.

Let F be the finish time of any legal schedule. Let $W = \sum p_i d_i$ be the *total work* that needs to be performed, and let $LB_1 = W/p$. The intuition is that, even if we use all p processors for all the time from 0 to F, then the most work we accomplish is W/p.

Lemma 1: $F \geq LB_1$.

Proof of Lemma 1: First, let's say a job Job_i is *active* at time t if $s_i \leq t < f_i$. Since each job has $f_i = s_i + d_i$, there are exactly d_i times at which a job is active. Look at a running sum of the processor utilization $\sum_{t=0}^{t=F-1} \sum_{i|Job_i \text{ is active at time } t} p_i$. On the one hand, since the total number of processors in active jobs at any time t is at most p, this sum is at most pF. On the other hand, if we reverse the order of the summands we get $pF \geq \sum_{t=0}^{t=F-1} \sum_{i|Job_i \text{ is active at time } t} p_i = \sum_i \sum_{t|Job_i \text{ is active at time } t} p_i = \sum_i p_i d_i = W$, so $F \geq W/p = LB_1$.

LB_1 will be a tight bound if the schedule can keep most of the processors occupied. But what if there's one job that requires few processors but has a long duration? That job could add a lot to the total finish time, but relatively little to W. So let's define $LB_2 = max_i d_i$ to be the longest duration of any job.

Lemma 2: $F \geq LB_2$.

Proof of Lemma 2: Let i be the job of longest duration. Then $F \geq f_i = s_i + d_i \geq d_i = LB_2$.

2.3 Bounding our algorithm's cost

Now we need to relate our algorithm to the bounds in the previous subsection. What we'll show is that our algorithm's schedule can be broken up into two phases, the first utilizing at least half the processors, and the second when no jobs are added to the schedule. We'll compare the first phase to LB_1, since we're getting a lot of work done; and the second phase to LB_2, since the time of this phase is dominated by the longest remaining duration.

Consider the schedule produced by our algorithm, and let F_{LPR} be the finish time of this schedule. Let t_1 be the first time $\leq 1/2p$ processors are being utilized in this schedule.

Lemma 3: All jobs Job_i with $p_i > 1/2p$ are completed by time t_1.

Proof of lemma 3: Let k be the number of such jobs, and assume we have the job list sorted by p_i, so these jobs are $Job_1, ..Job_k$, and $p_1 \geq p_2 ... \geq p_k$. Job_1 will be started at time 0, and none of the later jobs can start until Hob_1 finishes. As soon as Job_1 finishes, we'll have enough free processors to schedule Job_2, and it will have by definition the largest processor requirement of any Undone job. Similarly, Job_3 will be scheduled as soon as Job_2 finishes, and so on until Job_k finishes. Since all k jobs require $> 1/2p$ processors, this means that $> 1/2p$ processors are utilized at all times until Job_k finishes. So the first time t_1 when $\leq 1/2p$ processors are utilized must be after Job_k and hence all of $Job_1, ..Job_k$ have finished.

Lemma 4: All jobs have started by time t_1.

Proof: By contradiction. Assume there were some jobs that had not started by time t_1, and let Job_i be such a job of maximum processor requirement. Then since we have finished all jobs that require more than $p/2$ processors, $p_i \leq p/2$. Since the number of jobs being used is $\leq p/2$, the number of processors available is $\geq p/2 \geq p_i$. So our algorithm would schedule Job_i to start at time t_1, a contradiction to the assumption that it had not started by time t_1.

Lemma 5: $t_1 \leq 2LB_1$.

Proof: Similar to Lemma 1, so I'll skip some calculations. The total work done up to time t_1 is at least $p/2t_1$, since p_2 processors are utilized at all times less than t_1. Thus, $p/2t_1 \leq W$, or $t_1 \leq 2W/p = 2LB_1$.

Lemma 6: $F_{LPR} \leq t_1 + LB_2$.

Proof of Lemma 6: Let Job_i be the last job to finish in the LPR schedule. Then $F_{LPR} = f_i = s_i + d_i$. Since all jobs have started by time t_1, and $d_i \leq max_j d_j = LB_2$, $s_i + d_i \leq t_1 + LB_2$.

2.4 Conclusion: LPR a ratio 3 approximation algorithm

By Lemmas 5 and 6 above, $F_{LPR} \leq 2LB_1 + LB_2$. By Lemmas 1 and 2, for any legal schedule, its finish time $F \geq LB_1, LB_2$. Thus, $F_{LPR} \leq 3F$, so LPR acheives approximation ratio 3.

This is a classic "combine bounds" argument, with $\alpha_1 = 2, \alpha_2 = 1$.

3 The minimum makespan problem

We'll illustrate using this method inside a case analysis with the *minimum makespan problem*, from the last class (sacrificing precision). Recall:

Instance: A number of identical machines m, and a list of positive real job sizes $a_1, ..a_n$.

Solution Format: Each job is assigned to one of the m machines

Objective: Minimize the maximum load of a machine,
$Cost(sched) = max_{1 \leq i \leq m} \sum_{jobs\ j\ assigned\ to\ machine\ i} a_j$.

Last class, we gave a PTAS for this problem, but saw that for even small ratios, it was very time-consuming. Here's a quick-and-dirty greedy algorithm we can use instead:

1. Sort the jobs from largest to smallest.
2. In order, assign each job to the machine whose load is currently smallest.

How good is this algorithm? Say $m = 2$, and we give it jobs of sizes $6, 6, 4, 4, 4$. The optimal solution is to use one machines for the two jobs of size 6, and one for the 3 of size 4. But the greedy algorithm will instead split the large jobs between the machines, and have to give one machine two of the three 4's. Thus, the total cost is 14 rather than the optimal 12. So the ratio for this algorithm is at least 7/6.

To get an approximation ratio guarantee, let's look at two lower bounds: L_1 is the average load: $L_1 = \frac{\sum a_i}{m}$, since at least one machine must have a load above average. L_2 is the length of the longest job, which, after sorting, is a_1.

A simple ratio guarantee using these two bounds is as follows. Call a_i a *tipping job* if it is the last job placed on a machine that ends up with a heaviest load by the algorithm.

Let a_i be a tipping job. Let M be the makespan our algorithm achieves, and hence the same as the load of the machine a_i is given to. Before a_i was assigned to this machine, its load was $M - a_i$ (since no other job is later assigned to it.) Since it was the least burdened machine at this time, all other machines had (and still have) load at least $M - a_i$. So at the end, all machines have load $M - a_i$. If we sum the load of all machines, we sum each job once, so we then get $\sum_j a_j \geq m(M - a_i)$. Thus, $M \leq \sum_j a_j/m + a_i = LB_1 + a_i \leq LB_1 + LB_2$. Thus, M is at most twice the optimal makespan, $(\alpha_1 = 1 = \alpha_2)$.

However, this isn't a tight guarantee, and we can do better with a more refined argument, that splits it into cases. Let all be as above. We'll divide into three cases: $i \geq 2m$, $i \leq m$ and $m + 1 \leq i < 2m$.

Case 1: $i \geq 2m$. Then $\sum_j a_j \geq \sum_{j \leq i} a_j \geq i a_i$, so $a_i \leq 1/i \sum_j a_j \leq 1/2m \sum_j a_j = 1/2 LB_1$ Since we have $M \leq LB_1 + a_i \leq 3/2 LB_1$, in this case, M is at most 1.5 times the optimal $(\alpha_1 = 3/2, \alpha_2 = 0)$

Case 2: $i \leq m$. In this case, each job gets assigned to a currently unused machine, so no machine gets more than one job. Therefore, the makespan for our algorithm is $M = a_1 = LB_2$ so our algorithm is optimal in this case.

Case 3: $m + 1 \leq i < 2m$. In this case, the first m jobs go on distinct machines. After that, there is always a machine with a single job assigned to it. Since if $a_j > a_{j'}$ we'll assign a job to a machine with $a_{j'}$ before we do to one with load a_j, if machine $j' > j$ has a single job, then so does machine j. Let $k = 2m - i$. At the end, there must be at least k machines with single jobs, and by the above these must include machines $1..k$.

Case 3a: The optimal schedule has a machine assigned two jobs , one of which is in $a_1, ..a_k$ and the other is before a_i. In this case, the optimal schedule has makespan at least $M_{opt} \geq a_j + a_l$, where $1 \leq j \leq k$ and $1 \leq l \leq i$. In particular, $M_{opt} \geq a_k + a_i$, since the jobs are sorted. On the other hand, there was a machine assigned only a_k when a_i was added to a machine with smaller load $L \leq a_k$, to get our algorithm's makespan $M = L + a_i \leq a_k + a_i \leq M_{opt}$. Thus, our algorithm is optimal in this subcase.

Case 3b: The optimal schedule schedules no jobs before a_k with jobs before a_i. In this case, the optimal schedule must use k distinct machines for $a_1, ..a_k$, and not use these machines for any jobs $a_{k+1}, ..a_i$. (This is very similar to the case when $a_{k+1}, ..a_i$ are scheduled on $m' = m - k$ machines. Since in this case, we are scheduling $i - k = i - 2m + i = 2(i - m)$ jobs on $m - 2m + i = i - m$ machines, this is similar to Case 1. I don't know a direct reduction, so I'm going to repeat a number of calculations relative to these jobs, but that's the intuition.) Let $S = \sum_{j=k+1}^{j=i} a_j$. Since all the jobs $a_{k+1}, ..a_i$ are assigned to $m-k$ machines in the optimal schedule, at least one must have load $\frac{S}{m-k}$, which we'll call LB_3. Note that $S \geq (i - k)a_i = (2i - 2m)a_i$, so $a_i \leq \frac{S}{2(i-m)}$. Consider the machine we add a_i to in our algorithm's schedule. At this point, all of the jobs $a_{k+1}, ..a_i$ are in the last $m - k$ machines, none of the jobs $a_1, ..a_k$ are on these machines, and each of them has load at least $M - a_i$. So summing up all of their loads, we get $S \geq (M - a_i)(m - k)$, so $M \leq S/(m - k) + a_i \leq S/(m - k) + S/2(m - k) = (1.5)LB_3$, so in this case M is at most 1.5 times the optimal makespan.

Hence, in all cases, the algorithm's ratio is at most 1.5.

4 The set cover problem

We'll use set cover to illustrate the combine bounds method with a family of bounds. Here, the algorithm itself defines both a solution and a bound in the family that almost match.

4.1 The set cover problem

Instance: A set $U = \{u_1, ..u_n\}$ and a family of subsets $S_1, ..S_m \subseteq U$.

Solution Format: A set $C \subseteq \{1, ..m\}$

Constraints: $\cup_{i \in C} S_i = U$, i.e., every element is covered by some set in C.

Objective: Minimize $|C|$.

We can think of this problem as the universe being a set of type of jobs we might need to perform, and each set is the set of jobs a machine we can buy might be able to perform. We need to buy as few machines as possible to be able to do any type of job.

4.2 A family of bounds

Let k be the size of the largest set S_i. Clearly, we'll need at least n/k sets in our cover, since each can cover at most k of the n jobs. We can generalize this to weighted sizes as follows: Let $w(u) > 0$ be assigned to each $u \in U$. Let $W(S_i) = \sum_{u \in S_i} w(u)$, and let $W = \sum_{u \in U} w(u)$. Let $\kappa = max_i W(S_i)$. Define the lower bound for the array of weights $\vec{w}$ by $L_{\vec{w}} = W/\kappa$.

Lemma 1: For any set of positive weights $\vec{w}$, and any set cover C, $|C| \geq L_{\vec{w}}$.

Proof: Consider $\sum_{i \in C} W(S_i)$. On the one hand, it is at most $\kappa|C|$, since each $W(S_i) \leq \kappa$ by definition of κ. On the other hand, $\sum_{i \in C} W(S_i) = \sum_{i \in C} \sum_{u \in S_i} w(u) \geq \sum_u w(u) = W$, since all u are in some S_i, and the weights are postive. Thus, $|C|\kappa \geq W$, and the lemma follows.

Thus, each set of weights on the universe defines a lower bound.

4.3 A simple greedy algorithm

1. $Undone \leftarrow U$.
2. $C \leftarrow \emptyset$.
3. Repeat until $Undone = \emptyset$:
4. Pick the set S_i which maximizes $|S_i \cap Undone|$
5. Add i to C
6. (For analysis only): for each $x \in S_i \cap Undone$, define $w(x) = 1/|S_i \cap Undone|$.
7. $Undone \leftarrow Undone - S_i$
8. Return C.
9. (For analysis only:) Return $LB_{\vec{w}}$.

Line 6 is not actually used in the algorithm itself. With this extra line, this algorithm will not only produce a reasonably good set cover, but will produce a *reason* why we can't do much better, i.e., it produces its own bound. (Similiarly to asking a flow algorithm to also output the corresponding cut.)

4.4 An $H_k \leq \ln(n+1)$ approximation guarantee

We'll now prove an approximation guarantee for the algorithm by showing that the lower bound found is not too far off from the set cover found. The ratio will be at most $H_k = \sum_{i=1}^{i=k} 1/i$. This is at most $H_n \leq \ln(n+1)$, but is better if the sets are guaranteed to be small.

Lemma 2: For C the cover and $\vec{w}$ the weights produced by the algorithm, $W = |C|$.

Each iteration, we assign weight $1/|S_i \cap Undone|$ to $|S_i \cap Undone|$ elements of U. Thus, W increases by one each iteration. Also, we add one element to C each iteration, so $|C|$ stays equal to W throughout.

Lemma 3: $max_i|S_i \cap Undone|$ never increases during the algorithm.

Elements are only removed from $Undone$, never added. So for each i, $|S_i \cap Undone|$ never increases. Thus the maximum never increases.

Lemma 4: For any i, j, at most j elements u of S_i have $w(u) \geq 1/j$.

Assume not. Consider the first iteration when $maxS_l|S_l \cap Undone| \leq j$. Until this point, each element with $w(u)$ defined has $w(u) \leq 1/(j+1)$. Thus, at this time, S_i has $j+1$ elements with $w(u)$ undefined, i.e., $|S_i \cap Undone| \geq j+1$. This contradicts $maxS_l|S_l \cap Undone| \leq j$.

Lemma 5: $\kappa \leq H_k$.

Fix any set S_i. Let $k' \leq k = |S_i|$. Sort its elements as $u_1, ..u_{k'}$ in non-increasing order of $w(u)$. From Lemma 4, for each j, S_i has at most j elements u with $w(u) \geq 1/j$, so $w(u_{j+1}) \leq 1/(j+1)$. Thus, $W(S_i) = \sum_{j=1}^{j=k'} w(u_j) \leq \sum_{j=1}^{j=k'} 1/j = H_{k'} \leq H_k$, for each i; thus, the same is true for the maximum.

Theorem: The algorithm above acheives approximation guarantee H_k.

Let $C, \vec{w}$ be the cover and weights the algorithm produces. Let C_{opt} be the optimum cover. Then by Lemma 1, $|C_{opt}| \geq LB_{\vec{w}} = W/\kappa \geq |C|/H_k$ (by Lemmas 2 and 5). So $|C| \leq H_k|C_{opt}|$, as claimed.