Scheduling to Minimize Power Consumption using Submodular Functions

Erik D. Demaine
MIT
edemaine@mit.edu

Morteza Zadimoghaddam
MIT
morteza@mit.edu

ABSTRACT

We develop logarithmic approximation algorithms for extremely general formulations of multiprocessor multi-interval offline task scheduling to minimize power usage. Here each processor has an arbitrary specified power consumption to be turned on for each possible time interval, and each job has a specified list of time interval/processor pairs during which it could be scheduled. (A processor need not be in use for an entire interval it is turned on.) If there is a feasible schedule, our algorithm finds a feasible schedule with total power usage within an $O(\log n)$ factor of optimal, where n is the number of jobs. (Even in a simple setting with one processor, the problem is Set-Cover hard.) If not all jobs can be scheduled and each job has a specified value, then our algorithm finds a schedule of value at least $(1-\varepsilon)Z$ and power usage within an $O(\log(1/\varepsilon))$ factor of the optimal schedule of value at least Z, for any specified Z and $\varepsilon > 0$. At the foundation of our work is a general framework for logarithmic approximation to maximizing any submodular function subject to budget constraints.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; G.2.2 [**Discrete Mathematics**]: Graph Theory

General Terms

Algorithms, Theory

Keywords

sleep state, pre-emptive scheduling, multiprocessor scheduling, approximation algorithms

1. INTRODUCTION

Power management systems aim to reduce energy consumption while keeping the performance high. The motivations include battery conservation (as battery capacities continue to grow much slower than computational power) and reducing operating cost and environmental impact (both direct from energy consumption and indirect from cooling).

Processor energy usage. A common approach in practice is to allow processors to enter a *sleep state*, which consumes less energy, when they are idle. All previous work assumes a simple model in which we pay zero energy during the sleep state (which makes approximation only harder), a unit energy rate during the awake state (by scaling), and a fixed restart cost α to exit the sleep state. Thus the total energy consumed is the sum over all awake intervals of α plus the length of the interval.

There are many settings where this simple model may not reflect reality, which we address in this paper:

1. When the processors are not identical: different processors do not necessarily consume energy at the same rate, so we cannot scale to have all processors use a unit rate.

2. When the energy consumption varies over the time: keeping a processor active for two intervals of the same length may not consume the same energy. One example is if we optimize energy *cost* instead of actual energy, which varies substantially in energy markets over the course of a day. Another use for this generalization is if a processor is not available for some time slots, which we can represent by setting the cost of the processor to be infinity for these time slots.

3. When the energy consumption is an arbitrary function of its length: the growth in energy use might not be an affine function of the duration a processor is awake. For example, if a processor stays awake for a short time, it might not need to cool with a fan, saving energy, but the longer it stays awake, the faster the fan may need to run and the more energy consumed.

We allow the energy consumption of an awake interval to be an arbitrary function of the interval and the processor. We also allow the processor to be idle (but still consume energy) during such an interval. As a result, our algorithms automatically choose to combine multiple awake intervals (and the intervening sleep intervals) together into one awake interval if this change causes a net decrease in energy consumption.

Multi-interval task scheduling. Most previous work assumes that each task has an arrival time, deadline, and

SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

processing time. The goal is then to find a schedule that executes all tasks by their deadlines and consumes the minimum energy (according to the notion above). This setup implicitly assumes identical processors.

We consider a generalization of this problem, called *multi-interval scheduling*, in which each task has a list of one or more time intervals during which it can execute, and the goal is to schedule each job into one of its time intervals. The list of time intervals can be different for each processor, for example, if the job needs specific resources held by different processors at different times.

Prize-collecting version. All previous work assumes that all jobs can be scheduled using the current processors and available resources. This assumption is not necessarily satisfied in many practical situations, when jobs outweigh resources. In these cases, we must pick a subset of jobs to schedule.

We consider a general weighted *prize-collecting* version in which each job has a specified *value*. The bicriterion problem is then to find a schedule of value at least Z and minimum energy consumption subject to achieving this value.

Our results. We obtain in Section 3 an $O(\log n)$-approximation algorithm for scheduling n jobs to minimize power consumption. For the prize-collecting version, we obtain in Section 4 an $O(\log(1/\varepsilon))$-approximation for scheduling jobs of total value at least $(1-\varepsilon)Z$, comparing to an adversary required to schedule jobs of total value at least Z (assuming such a schedule exists), for any specified Z and $\varepsilon > 0$. Both of our algorithms allow specifying an arbitrary processor energy usage for each possible interval on each processor, specifying an arbitrary set of candidate intervals on each processor for each job, and specifying an arbitrary value for each job.

These results are all best possible assuming P $\neq$ NP: we prove in Appendix A that even simple one-processor versions of these problems are Set-Cover hard.

Our approximation algorithms are based on a technique of independent interest. In Section 2, we introduce a general optimization problem, called *submodular maximization with budget constraints*. Many interesting optimization problems are special cases of this general problem, for example, Set Cover and Max Cover [7, 11] and the submodular maximization problems studied in [9, 10]. We obtain bicriteria $((1-\varepsilon), O(\log 1/\varepsilon))$-approximation factor for this general problem.

In Section 3, we show how our schedule-all-jobs problem can be formulated by a bipartite graph and its matchings. We define a matching function in bipartite graphs, and show that this function is submodular. Then the general technique of Section 2 solves the problem.

In Section 4, we show how the prize-collecting version of our scheduling problem can be formulated with a bipartite graph with weights on its nodes. Again we define a matching function in these weighted bipartite graphs, and with a more complicated proof, show that this function is also submodular. Again the general technique of Section 2 applies.

The general algorithm in Section 2 has many different and independent applications because submodular functions arise in a variety of applications. They can be seen as utility and cost functions of bidding auctions in game theory application [5]. These functions can be seen as covering functions which have many applications in different optimization problems: Set Cover functions, Edge Cut functions in graphs, etc.

Previous work. The one-interval one-processor case of our problem with simple energy consumption function (α plus the interval length) remained an important and challenging open problem for several years: it was not even clear whether it was NP-hard.

The first main results for this problem considered the power-saving setting, which is easier with respect to approximation algorithms. Augustine, Irani, and Swamy [1] gave an online algorithm, which schedules jobs as they arrive without knowledge of future jobs, that achieves a competitive ratio of $3 + 2\sqrt{2}$. (The best lower bound for this problem is 2 [2, 6].)

For the offline version, Irani, Shukla, and Gupta [6] obtained a 3-approximation algorithm. Finally, Baptiste [2] solved the open problem: he developed a polynomial-time optimal algorithm based on an sophisticated dynamic programming approach. Demaine et al. [4] later generalized this result to also handle multiple processors.

The multi-interval case was considered only by Demaine et al. [4], after Baptiste mentioned the generalization during his talk at SODA 2006. They show that this problem is Set-Cover hard, so it does not have an $o(\log n)$-approximation. They also obtain a $1 + \frac{2}{3}\alpha$-approximation for the multi-interval multi-processor case, where α is the fixed restart cost. Note that α can be as large as n, so there is no general algorithm with approximation factor better than $\Theta(n)$ in the worst case (when α is around n).

However, both the Baptiste result [2] and Demaine et al. results [4] assume that processors enter the sleep state whenever they go idle, immediately incurring an α cost. For this reason, the problem can also be called *minimum-gap scheduling*. But this assumption seems unreasonable in practice: we can easily leave the processor awake during sufficiently short intervals in order to save energy. As mentioned above, the problem formulations considered in this paper fix this issue.

2. SUBMODULAR MAXIMIZATION WITH BUDGET CONSTRAINTS

Submodular functions arise in a variety of applications. They can represent different forms of functions in optimization problems. As a game theoretic example, both profit and budget functions in bid optimization problems are Set-Cover type functions (including the weighted version) which are special cases of submodular functions. As another application of these functions in online algorithms, we can mention the secretary problem in different models, the bipartite graph setting in [8], and the submodular functions setting in [3].

The authors of [10] studied the problem of submodular maximization under matroid and knapsack constraints (which can be seen as some kind of budget constraints), and they give the first constant factor approximation when the number of constraints is constant. We try to find solutions with more utility by relaxing the budget constraints. We give the first $(1-\varepsilon)$-approximation for utility maximization with relaxing the budget constraint by $\log(1/\varepsilon)$. In our model, we allow the cost of a subset of items be less than their sum. This way we can cover more general cases (non-linear or submodular cost functions). All previous works on submodular functions assume that the cost function is linear. Therefore they can not cover many interesting optimization problems including the scheduling problems we are studying in this paper. Later we combine this result with other techniques to give optimal scheduling strategies for energy minimization problem with parallel machines.

Now we formulate the problem of submodular maximization with budget constraints.

DEFINITION 1. *Let $U = \{a_1, a_2, \ldots, a_n\}$ be a set of n items. We are given a set $S = \{S_1, S_2, \ldots, S_m\} \subseteq 2^U$ specifying m allowable subsets of U that we can add to our solution. We are also given costs $C_1, C_2, \ldots, C_m$ for the subsets, where S_i costs C_i. Finally, we are given a* utility function *$F : 2^U \to \mathbb{R}$ defined on subsets of U. We require that F is* submodular *meaning that, for any two subsets A, B of U, we have*

$$F(A) + F(B) \geq F(A \cap B) + F(A \cup B).$$

We also require that F is monotone *(being a utility function) meaning that, for any subsets $A \subseteq B \subseteq U$, we have $F(A) \leq F(B)$.*

The problem is to choose a collection of the input subsets with reasonable cost and utility. The cost of a collection of subsets is the sum of their costs. The utility of these subsets is equal to the utility of their union. In particular, if we pick k subsets $S_1, S_2, \ldots, S_k$, their cost is $\Sigma_{i=1}^{k} C_i$ and their utility is equal to $F(\cup_{i=1}^{k} S_i)$. We are given a utility threshold x, and the problem is to find a collection with utility at least x having minimum possible cost.

Note that all previous work assumes that the set S of allowable subsets consists only of single-item subsets, namely $\{a_1\}, \{a_2\}, \ldots, \{a_n\}$. Equivalently, they assume that the cost of picking a subset of items is equal to the sum of the costs of the picked items (a linear cost function). By contrast, we allow that there be other subsets that we can pick with different costs, but that all such subsets are explicitly given in the input. The cost of a subset might be different from the sum of the costs of the items in that subset; in practice, we expect the cost to be less than the sum of the item costs.

We need the following result in the proof of the main algorithm of this section. Similar lemmas like this are proved in the literature of submodular functions. But we need to prove this more general lemma.

LEMMA 2.1. *Let T be the union of k subsets $S_1, S_2, \ldots, S_k$, and S' be another arbitrary subset. For a monotone submodular function F defined on these subsets, we have that*

$$\sum_{j=1}^{k} [F(S' \cup S_j) - F(S')] \geq F(T) - F(S').$$

PROOF. Let T' be the union of T and S'. We prove that $\sum_{j=1}^{k} [F(S' \cup S_j) - F(S')] \geq F(T') - F(S')$ which also implies the claim. Define subset S'_i be $(\cup_{j=1}^{i} S_j) \cup S'$ for any $0 \leq i \leq k$. We prove that

$$F(S' \cup S_i) - F(S') \geq F(S'_i) - F(S'_{i-1}).$$

Because F is submodular, we know that $F(A) + F(B) \geq F(A \cup B) + F(A \cap B)$ for any pair of subsets A and B. Let A be the set $S' \cup S_i$, and B be the set S'_{i-1}. Their union is S'_i, and their intersection is a superset of S'. So we have that

$$\begin{aligned} F(S' \cup S_i) + F(S'_{i-1}) &\geq F(S'_i) + F([S' \cup S_i] \cap [S'_{i-1}]) \\ &\geq F(S'_i) + F(S'). \end{aligned}$$

This completes the proof of the inequality, $F(S' \cup S_i) - F(S') \geq F(S'_i) - F(S'_{i-1})$.

If we sum this inequality over all values of $1 \leq i \leq k$, we can conclude the claim:

$$\begin{aligned} \sum_{i=1}^{k} F(S' \cup S_i) - F(S') &\geq \sum_{i=1}^{k} F(S'_i) - F(S'_{i-1}) \\ &= F(T') - F(S') \\ &\geq F(T) - F(S'). \end{aligned}$$

□

Now we show how to find a collection with utility $(1-\varepsilon)x$ and cost $O(\log(1/\varepsilon))$ times the optimum cost. Later we show how to find a subset with utility x in our particular application, scheduling with minimum energy consumption. It is also interesting that the following algorithm generalizes the well-known greedy algorithm for Set Cover in the sense that the Set-Cover type functions are special cases of monotone submodular functions. In order to use the following algorithm to solve the Set Cover problem with a logarithmic approximation factor (which is the best possible result for Set Cover), one just needs to set ε to some value less than 1 over the number of items in the Set-Cover instance.

LEMMA 2.2. *If there exists a collection of subsets (optimal solution) with cost at most B and utility at least x, there is a polynomial time algorithm that can find a collection of subsets of cost at most $O(B \log(1/\varepsilon))$, and utility at least $(1-\varepsilon)x$ for any $0 < \varepsilon < 1$.*

PROOF. The algorithm is as follows. Start with set $S = \emptyset$. Iteratively, find the set S_i with maximum ratio of $\min\{x, F(S \cup S_i)\} - F(S)/C_i$ for $1 \leq i \leq m$ where $\min\{a, b\}$ is the minimum of a and b. In fact we are choosing the subset that maximizes the ratio of the increase in the utility function over the increase in the cost function, and we just care about the increments in our utility up to value x. If a subset increases our utility to some value more than x, we just take into account the difference between previous value of our utility and x, not the new value of our utility. We do this iteratively till our utility is at least $(1-\varepsilon)x$.

We prove that the cost of our solution is $O(B \log(1/\varepsilon))$. Assume that we pick some subsets like $S'_1, S'_2, \ldots, S'_{k'}$ respectively. We define the subsets of our solution into $\log(1/\varepsilon)$ phases. Phase $1 \leq i \leq \log(1/\varepsilon)$, ends when the utility of our solution reaches $(1 - 1/2^i)x$, and starts when the previous phase ends. In each phase, we pick a sequence of the k' subsets $S'_1, S'_2, \ldots, S'_{k'}$. We prove that the cost of each phase is $O(B)$, and therefore the total cost is $O(B \log(1/\varepsilon))$ because there are $\log(1/\varepsilon)$ phases.

Let S'_{a_i} be the last subset we pick in phase i. So $F(\cup_{j=1}^{a_i} S'_j)$ is our utility at the end of phase i, and is at least $(1 - 1/2^i)x$, and $F(\cup_{j=1}^{a_i - 1} S'_j)$ is less than $(1 - 1/2^i)x$. So we pick subsets $S'_{a_{i-1}+1}, S'_{a_{i-1}+2}, \ldots, S'_{a_i}$ in phase i. We prove that the ratio of utility per cost of all subsets inserted in phase i is at least $\frac{x/2^i}{B}$. Assume that we are in phase i, and we want to pick another set (phase i is not finished yet). Let S' be our current set (the union of all subsets we picked up to now). $F(S')$ is less than $(1-1/2^i)x$. We also know that there exists a solution (optimal solution) with cost B and utility x. Without loss of generality, we assume that this solution consists of k subsets $S_1, S_2, \ldots, S_k$. Let T be the union of these k subsets. Using lemma 2.1, we have that

$$\sum_{j=1}^{k}[F(S' \cup S_j) - F(S')] \geq F(T) - F(S') > x/2^i.$$

If $F(S' \cup S_j)$ is at most x for any $1 \leq j \leq k$, we can say that

$$\sum_{j=1}^{k}[\min\{x, F(S' \cup S_j)\} - F(S')] =$$

$$\sum_{j=1}^{k}[F(S' \cup S_j) - F(S')] \geq F(T) - F(S') > x/2^i.$$

Otherwise there is some j for which $F(S' \cup S_j)$ is more than x. So $\min\{x, F(S' \cup S_j)\} - F(S')$ is at least $x/2^i$ because $F(S')$ is less than $(1-1/2^i)x$. So in both cases we can claim the above inequality. We also know that

$$\sum_{j=1}^{k} C_j \leq B,$$

where C_j is the cost of set S_j. In every iteration, we find the subset with the maximum ratio of utility per cost (the increase in utility per the cost of the subset). Note that we also consider these k subsets $S_1, S_2, \ldots, S_k$ as candidates. So the ratio of the subset we find in each iteration is not less than the ratio of each of these k subsets. The ratio of subset S_j is $[\min\{x, F(S' \cup S_j)\} - F(S')]/C_j$. The maximum ratio of these k subsets is at least the sum of the nominators of the k ratios of these sets over the sum of their denominators which is

$$\frac{\sum_{j=1}^{k}[\min\{x, F(S' \cup S_j)\} - F(S')]}{\sum_{j=1}^{k} C_j} > \frac{x}{2^i B}.$$

So in phase i, the utility per cost ratio of each subset we add is at least $\frac{x}{2^i B}$. Now we can bound the cost of this phase. We pick subsets $S'_{a_{i-1}+1}$, $S'_{a_{i-1}+2}$, $\ldots$, S'_{a_i} in phase i. Let u_0 be our utility at the beginning of phase i. In other words, u_0 is $F(\cup_{j=1}^{a_{i-1}} S'_j)$. Assume we pick l subsets in this phase, i.e., l is $a_i - a_{i-1}$. Let u_j be our utility after inserting jth subset in this phase where $1 \leq j \leq l$. Note that we stop the algorithm when our utility reaches $(1-\varepsilon)x$. So our utility after adding the first $l-1$ subsets is less than x. Our utility at the end of this phase, u_l might be more than x. For any $1 \leq j \leq l-1$, the utility per cost ratio is $u_j - u_{j-1}$ divided by the cost of the jth subset. For the last subset, the ratio is $\min\{x, u_l\} - u_{l-1}$ divided by the cost of the last subset of this phase. According to the definition of the phases, our utility at the beginning of this phase, u_0 is at least $(1-1/2^{i-1})x$. So we have that

$$\min\{x, u_l\} - u_{l-1} + \sum_{j=1}^{l-1} u_j - u_{j-1} =$$

$$\min\{x, u_l\} - u_0 \leq x - (1-1/2^{i-1})x = x/2^{i-1}.$$

On the other hand, we know that the utility per cost ratio of all these subsets is at least $\frac{x}{2^i B}$. Therefore the total cost of this phase is at most

$$\frac{[\min\{x, u_l\} - u_{l-1} + \sum_{j=1}^{l-1} u_j - u_{j-1}]}{x/2^i B} \leq \frac{x/2^{i-1}}{x/2^i B},$$

which is at most $2B$. So the total cost in all phases is not more than $\log(1/\varepsilon) \cdot 2B$. $\square$

3. SCHEDULING TO MINIMIZE POWER IN PARALLEL MACHINES

We proved how to find almost optimal solutions with reasonable cost when the utility functions are submodular. Here we show how the scheduling problem can be formulated as an optimization problem with submodular utility functions.

First we explain the power minimization scheduling problem in more detail.

Definition 2. *There are p processors $P_1, P_2, \ldots, P_p$ and n jobs $j_1, j_2, \ldots, j_n$. Each processor has an energy cost $c(I)$ for every possible awake interval I. Each job j_i has a unit processing time (which is equivalent to allowing preemption), and set T_i of valid time slot/processor pairs. (Unlike previous work, T_i does not necessarily form a single interval, and it can have different valid time slots for different processors.) A* feasible schedule *consists of a set of awake time intervals for each processor, and an assignment of each job to an integer time and one of the processors, such that jobs are scheduled only during awake time slots (and during valid choices according to T_i) and no two jobs are scheduled at the same time on the same processor. The* cost *of such a schedule is the sum of the energy costs of the awake intervals of all processors.*

In the simple case which has been studied in [2, 4], it is assumed that the cost of an interval is a fixed amount of energy (restart cost α) plus the size of the interval. We assume a very general case in which the cost of keeping a machine active during an interval is a function of that machine, and the interval. For instance, it might take more energy to keep some machines active comparing to other machines, or some time intervals might have more cost. So there is a cost associated with every pair of a time interval and a machine. These costs might be explicitly given in the input, or can be accessed through a query oracle, i.e., when the number of possible intervals are not polynomial.

If we pick a collection of active intervals for each machine at first, we can then find and schedule the maximum number of possible jobs that can be all together scheduled in the active time slots without collision using the maximum bipartite matching algorithms. So the problem is to find a set of active intervals with low cost such that all jobs can be done during them.

Let U be the set of all time slots in different machines. In fact for every unit of time, we put p copies in U, because at each unit of time, we can schedule p jobs in different machines, so each of these p units is associated with one of the machines. We can define a function F over all subsets of U as follows. For every subset of time slot/processor pairs like $S \subset U$, $F(S)$ is the maximum number of jobs that can be scheduled in time slot/processor pairs of S. Our scheduling problem can be formulated as follows. We want to find a collection of time intervals $I_1, I_2, \ldots, I_k$ with minimum cost and $F(\cup_{i=1}^{k} I_i) = n$ (this means that all n jobs can be scheduled in these time intervals). Note that each I_i is a pair of a machine and a time interval, i.e., I_1 might be $(P_2, [3,6])$ which represents the time interval $[3,6]$ in machine P_2. The cost of each I_i can be accessed from the input or a query oracle. The cost of this collection of intervals is the sum of the costs of the intervals. We just need to prove that function F is monotone and submodular. The monotonicity comes from its definition. The submodularity proof is involved, and needs some graph theoretic Lemmas. Now we can present our main result for this broad class of scheduling problems.

THEOREM 3.1. *If there is a schedule with cost B which schedules all jobs, there is a polynomial time algorithm which schedules all jobs with cost $O(B \log n)$.*

PROOF. We are looking for a collection of intervals with utility at least n, and cost $O(B \log n)$. Lemma 3.2 below states that F (defined above) is submodular. Using the algorithm of Lemma 2.2, we can find a collection of time intervals with utility at least $(1-\varepsilon)n$ and cost at most $O(B \log(1/\varepsilon))$ because there exists a collection of time intervals (schedule) with utility n (schedules all n jobs) and cost B. Let ε be $1/(n+1)$. The cost of the result of our algorithm is $O(B \log(n+1))$, and its utility is at least $(1 - 1/(n+1))n > n - 1$. Because the utility function F always take integer values, the utility of our result is also n. So we can find a collection of time intervals that all jobs can be scheduled in them. We just need to run the maximum bipartite matching algorithm to find the appropriate schedule. This means that our algorithm also schedules all jobs, and has cost $O(B \log(n+1))$. $\square$

There is another definition of submodular functions that is equivalent to the one we presented in the previous section. We will use this new definition in the following lemma.

DEFINITION 3. *A function F is submodular if for every pair of subsets $A \subset B$, and an element z, we have:*

$$F(A \cup \{z\}) - F(A) \geq F(B \cup \{z\}) - F(B)$$

Now we just need to show that F is submodular. We can look at this function as the maximum matching function of subgraphs of a bipartite graph. Construct graph G as follows. Consider time slots of U as the vertices of one part of G named X. Put n vertices representing the jobs in the other side of G named Y. Note that the time slots of U are actually pairs of a time unit and a processor. Put an edge between one vertex of X and a vertex of Y if the associated job can be scheduled in that time slot (which is a pair of a time unit and a processor), i.e., if the job can be done in that processor and in that time unit. Now every subset of $S \subset X$ is a subset of time slots, and $F(S)$ is the maximum number of jobs that can be executed in S. So $F(S)$ is in fact the maximum cardinality matching that saturates only vertices of S in part X (it can saturate any subset of vertices in Y). A vertex is saturated by a matching if one of its incident edges participates in the matching. Now we can present this submodularity Lemma in this graph model.

LEMMA 3.2. *Given a bipartite graph G with parts X and Y. For every subset $S \subset X$, define $F(S)$ to be the maximum cardinality matching that saturates only vertices of S in part X. The function F is submodular.*

PROOF. We just need to prove that, for two subsets $A \subset B \subset X$ and a vertex v in X, the following inequality holds:

$$F(A \cup \{v\}) - F(A) \geq F(B \cup \{v\}) - F(B).$$

Let M_1 and M_2 be two maximum matchings that saturate only vertices of A and B respectively. Note that there might be more than just one maximum matching in each case (for sets A and B). We first prove that there are two such maximum matchings that M_1 is a subset of M_2, i.e., all edges in matching M_1 also are in matching M_2. This can be proved using the fact that $A \subset B$ as follows.

Consider two maximum matchings M_1 and M_2 with the maximum number of edges in common. The edges of $M_1 \Delta M_2$ form a bipartite graph H where $A_1 \Delta A_2$ is $A_1 \cup A_2 - A_1 \cap A_2$ for every pair of sets A_1 and A_2. Because it is a disjoint union of two matchings, every vertex in H has degree 0, 1 or 2. So H is a union of some paths and cycles. We first prove that there is no cycle in H. We prove this by contradiction. Let C be a cycle in H. The edges of C are alternatively in M_1 and M_2. All vertices of this cycle are either in part Y of the graph or in $A \subset X$. Now consider matching $M'_1 = M_1 \Delta C$ instead of M_1. It also saturates only some vertices of A in part X, and has the same size of M_1. Therefore M'_1 is also a maximum matching with the desired property, and has more edges in common with M_2. This contradiction implies that there is no cycle in H.

Now we study the paths in H. At first we prove that there is no path in H with even number of edges. Again we prove this by contradiction. The edges of a path in H alternate between matchings M_1 and M_2. Let P be a path in H with even number of edges. This path has equal number of edges from M_1 and M_2. Now if we take $M'_2 = M_2 \Delta P$ instead of M_2, we have a new matching with the same number of edges, and it has more edges in common with M_1. This contradiction shows that there is no even path in H.

Finally we prove that all other paths in H are just some single edges from M_2, and therefore there is no edge from M_1 in H. This completes the proof of the claim that M_1 is a subset of M_2. Again assume that there is a path P' with odd and more than one number of edges. Let $e_1, e_2, \ldots, e_{2l+1}$ are the edges of P'. The edges with even index are in M_1, the rest of the edges are in M_2 otherwise $M''_2 = M_2 \Delta P'$ would be a matching for set B which has more edges than M_2 (this is a contradiction). Because P' is an odd path, we can assume that it starts from part Y, and ends in part X without loss of generality. Now if we delete edges $e_2, e_4, \ldots, e_{2l}$ from M_1, and insert edges $e_1, e_3, \ldots, e_{2l-1}$ instead, we reach a new matching M'_1. This matching uses a new vertex from Y, but the set of saturated vertices of X in matching M'_1 is the same as the ones in M_1. These two matchings also have the same size. But M'_1 has more edges in common with M_2. This is also contradiction, and implies that there is no such a path in H. So M_1 is a subset of M_2.

We are ready to prove the main claim of this theorem. Note that we have to prove this inequality:

$$F(A \cup \{v\}) - F(A) \geq F(B \cup \{v\}) - F(B).$$

We should prove that if adding v to B increases its maximum matching, it also increases the maximum matching of A. Let M_3 be the maximum matching of $B \cup \{v\}$. Let H' be the subgraph of G that contains the edges of $M_2 \Delta M_3$.

Because M_3 has more edges than M_2, there exists a path Q in H' that has more edges from M_3 than M_2 (cycles have the same number of edges from both matchings). The vertex v should be in path Q, otherwise we could have used the path Q to find a matching in B greater than M_2, i.e., matchings $M_2 \Delta Q$ could be a greater matching for set B in that case which is a contradiction.

The degree of v in H is 1, because it does not participate in matching M_2, does participate in M_3. So v can be seen as the starting vertex of path Q. Let $e_1, e_2, \ldots, e_{2l'+1}$ be the edges of Q. The edges $e_2, e_4, \ldots, e_{2l'}$ are in M_2, and some of them might be in M_1. Let $0 \leq i \leq l'$ be the maximum integer number for which all edges $e_2, e_4, \ldots, e_{2i}$ are in M_1. If e_2 is not in M_1, we set i to be 0. If we remove edges $e_2, e_4, \ldots, e_{2i}$ from M_1, and insert edges $e_1, e_3, \ldots, e_{2i+1}$ instead, we reach a matching for set $A \cup \{v\}$ with more edges than M_1. So adding v to A increases the size of its maximum matching.

Now the only thing we should check is that edges $e_1, e_3, \ldots, e_{2i+1}$ does not intersect with other edges of M_1. Let $v = v_0, v_1, v_2, \ldots, v_{2l'+1}$ be the vertices of Q. Because we remove edges $e_2, e_4, \ldots, e_{2i}$ from M_1, we do not have to be worried about inserting the first i edges $e_1, e_3, \ldots, e_{2i-1}$. The last edge we add is $e_{2i+1} = (v_{2i}, v_{2i+1})$. If v_{2i+1} is not saturated in M_1, there will be no intersection. So we just need to prove that v_{2i+1} is not saturated in M_1.

If i is equal to l', the vertex $v_{2i+1} = v_{2l'+1}$ is not saturated in M_2. Because M_1 is a subset of M_2, the vertex v_{2i+1} is also not saturated in M_1.

If i is less than l', the vertex v_{2i+1} is saturated in M_2 by edge e_{2i+2}. Assume v_{2i+1} is saturated in M_1 by an edge e'. The edge e' should be also in M_2 because all edges of M_1 are in M_2. The edge e' intersects with e_{2i+2}, so e' has to be equal to e_{2i+2}. The definition of value i implies that e_{2i+2} should not be in M_1 (we pick the maximum i with the above property). This contradiction shows that the vertex v_{2i+1} is not saturated in M_1, and therefore we get a greater matching in $A \cup \{v\}$ using the changes in M_1. □

4. PRIZE-COLLECTING SCHEDULING PROBLEM

We introduce the prize-collecting version of the scheduling problems. All previous work assumes that we can schedule all jobs using the existing processors. There are many cases that we can not execute all jobs, and we have to find a subset of jobs to schedule using low energy. There might be priorities among the jobs, i.e., there might be more important jobs to do. We formalize this problem as follows.

As before, there are P processors and n jobs. Each job j_i has a set T_i of time slot/processor pairs during which it can execute. Each job j_i also has a value z_i. We want to schedule a subset of jobs S with value at least a given threshold Z, and with minimum possible cost. The *value* of set S is the sum of its members' values, and it should be at least Z. Following we prove that there is a polynomial-time algorithm which finds a schedule with value at least $(1-\varepsilon)Z$ and cost at most $O(\log(1/\varepsilon))$ times the optimum solution. Note that the optimum solution has value at least Z.

Later in this section, we show how to find a solution with utility at least Z, and logarithmic approximation on the energy consumption (cost).

THEOREM 4.1. *If there is an schedule for the prize-collecting scheduling problem with value at least Z and cost B, there is an algorithm which finds a schedule with value at least $(1-\varepsilon)Z$ and cost at most $O(B\log(1/\varepsilon))$.*

PROOF. Like the simple version of the scheduling problem, we construct a bipartite graph, and relate it to our algorithm in Lemma 2.2. The difference is that the bipartite graph here has some weights (job values) on the vertices of one of its parts. And it makes it more complicated to prove that the corresponding utility function is submodular. At first we explain the construction of the bipartite graph, and show how to reduce our problem to it. Then we use Lemma 4.2 to prove that the utility function is submodular.

We make graph G with parts X and Y. The vertices of part X represent the time slot/processor pairs. So for each pair of a time unit in a processor, we have a vertex in X. On the other part, Y, we have the n jobs. The edges connect jobs to their sets of time slot/processor pairs, i.e., job j_i has edges only to time slot/processors pairs in T_i, so a job might have edges to different time units in different processors. The only difference is that each edge has a weight in this graph. Each edge connects a job to a time slot/processor pair, the weight of an edge is the value of its job. Every schedule is actually a matching in this bipartite graph, and the value of a matching is the sum of the values of the jobs that are scheduled in it. This is why we set the weight of an edge to the value of its job.

The problem again is to find a collection of time intervals for each processor, and schedule a subset of jobs in those intervals such that the value of this subset is close to Z, and the cost of the schedule is low. If we have a subset of intervals, we can find the best subset of jobs to schedule in it. This can be done using the maximum weighted bipartite matching. The only thing we have to prove is that the utility function associated with this weighted bipartite graph is submodular. This is also proved in Lemma 4.2. □

LEMMA 4.2. *Given a bipartite graph G with parts X and Y. Every vertex in Y has a value. For every subset $S \subset X$, define $F(S)$ be the maximum weighted matching that saturates only vertices of S in part X. The weight of a matching is the sum of the values of the vertices saturated by this matching in Y. The function F is submodular.*

PROOF. Let A and B be two subsets of X such that $A \subseteq B$. Let v be a vertex in X. We have to prove that:

$$F(A \cup \{v\}) - F(A) \geq F(B \cup \{v\}) - F(B)$$

Let M_1 and M_2 be two maximum weighted matchings that saturate only vertices of A and B in X respectively. Among all options we have, we choose two matchings M_1 and M_2 that have the maximum number of edges in common. We prove that every saturated vertex in M_1 is also saturated in M_2 (note that we can not prove that every edge in M_1 is also in M_2). We prove this by contradiction.

The saturated vertices in M_1 are either in set A or in set Y. At first, let v' be a vertex in A that is saturated in M_1, and not saturated in M_2. Let u' be its match in part Y (v' is a time slot/processor pair, and u' is a job). The vertex u' is saturated in M_2 otherwise we could add edge (v', u') to matching M_2, and get a matching with greater value instead of M_2. So u' is matched with a vertex of B like v'' in matching M_2. If we delete the edge (v'', u')

from matching M_2, and use edge (v', u') instead, the value of our matching remains unchanged, but we get a maximum matching instead of M_2 that has more edges in common with M_1 which is contradiction. So any vertex in X that is saturated in M_1 is also saturated in M_2.

The other case is when there is vertex in Y like u' that is saturated in M_1, and not saturated in M_2. The vertex u' is matched with vertex $w \in A$ in matching M_1. Again if w is not saturated in M_2, we can insert edge (w, u') to M_2, and get a matching with greater value. So w should be saturated in M_2. Let u'' be the vertex matched with w in M_2. For now assume that u'' is not saturated in M_1. Note that u' and u'' are some jobs with some values, and w is a time slot/processor pair. If the values of jobs u' and u'' are different, we can switch the edges in one of the matchings M_1 or M_2, and get a better matching. For example, if the value of u' is greater than u'', we can use edge (w, u') instead of (w, u'') in matching M_2, and increase the value of M_2. If the value of u'' is greater than u', we can use edge (w, u'') instead of (w, u') in matching M_1, and increase the value of M_1. So the value of u' and u'' are the same, we again can use (w, u'') instead of (w, u') in matching M_1, and get a matching with the same value but more edges in common with M_2. This is a contradiction. So u'' should be saturated in M_1 as well, but if we continue this process we find a path P starting with vertex u'. The edges of this path alternate between M_1 and M_2. Path P starts with an edge in M_1, so it can not end with another edge in M_1 otherwise we can take $M_2 \Delta P$ instead of M_2 to increase the size of our matching for set B which is a contradiction. So path P starts with vertex u' and an edge in M_1, and ends with an edge in M_2. We have the same situation as above, and we can reach the contradiction similarly (just take the last vertex of the path as u''). So we can say that all saturated vertices in M_1 are also saturated in M_2.

Despite the unweighted graphs, $F(A \cup \{v\}) - F(A)$ and $F(B \cup \{v\}) - F(B)$ might take values other than zero or one.

If M_2 is also a maximum matching for set $B \cup \{v\}$, we do not need to prove anything. Because $F(B \cup \{v\})$ would be equal to $F(B)$ in that case, and we know that $F(A \cup \{v\})$ is always at least $F(A)$. So assume that M_2' is a maximum matching for set $B \cup \{v\}$ that has the maximum number of edges in common with M_2, and its value is more than the value of M_2. Consider the graph H that consists of edges $M_2' \Delta M_2$. We know that H is union of some paths and cycles. We can prove that H is only a path that starts with vertex v. In fact, if there exists a connected component like C in H that does not include vertex v, we can take matching $M_2' \Delta C$ which is a matching for set $B \cup \{v\}$ with more edges in common with M_2. Note that the value of matching $M_2' \Delta C$ can not be less than the value of M_2' otherwise we can use the matching $M_2 \Delta C$ for set B instead of matching M_2, and get a greater value which is a contradiction (M_2 is a maximum value matching for set B).

So graph H has only one connected component that includes vertex v. Because vertex v does not participate in matching M_2, its degree in graph H should be at most 1. We also know that v is saturated in M_2', so its degree is one in H. Therefore, graph H is only a path P. This path starts with vertex v, and one of the edges in M_2'. The edges of P are alternatively in M_2' and M_2. If P ends with an edge in M_2, the set of jobs that these two matchings, M_2 and M_2', schedule are the same. So their values would be also the same, and $F(B \cup \{v\})$ would be equal to $F(B)$ which is a contradiction. So path P has odd number of edges. Let $e_1, e_2, \ldots, e_{2l+1}$ be the edges of P, and $v = v_0, v_1, v_2, \ldots, v_{2l+1}$ be its vertices. Note that $v_0, v_2, \ldots, v_{2l}$ are some time slot/processor pairs, and the other vertices are some jobs with some values. Edges $e_2, e_4, \ldots, e_{2l}$ are in M_2, and the rest are in M_2'.

The only job that is scheduled in M_2', and not scheduled in M_2 is the job associated with vertex v_{2l+1}. Let x_i be the value of the vertex v_{2i+1} for any $0 \leq i \leq l$. So $F(B \cup \{v\}) - F(B)$ is equal to x_l. We prove that x_l is not greater than any x_i for $0 \leq i < l$ by contradiction. Assume x_i is less than x_l for some $i < l$. We could change the matching M_2 in the following way, and get a matching with greater value for set B. We could delete edges $e_{2i+2}, e_{2i+4}, \ldots, e_{2l}$, and insert edges $e_{2i+3}, e_{2i+5}, \ldots, 2_{2l+1}$ instead. This way we schedule job v_{2l+1} instead of job v_{2i+1}, and increase our value by $x_l - x_i$. Because M_2 is a maximum matching for set B, this is a contradiction so x_l should be the minimum of all x_is.

If all edges $e_2, e_4, \ldots, e_{2l}$ are also in matching M_1, we can use path P to find a matching for set $A \cup \{v\}$ with value x_l more than the value of M_1. We can take matching $M_1 \Delta P$ for set $A \cup \{v\}$. Because vertex v_{2l+1} is not saturated in M_2, it is also not saturated in M_1. So $M_1 \Delta P$ is a matching for set $A \cup \{v\}$. We conclude that $F(A \cup \{v\}) - F(A)$ is at least x_l which is equal to $F(B \cup \{v\}) - F(B)$. This completes the proof for this case.

In the other case, there are some edges among $e_2, e_4, \ldots, e_{2l}$ that are not in M_1. Let e_{2j} be the first edge among these edges that is not in M_1. So all edges $e_2, e_4, \ldots, e_{2j-2}$ are in both M_1 and M_2. Note that e_{2j} matches job v_{2j-1} with the time slot/processor pair v_{2j} in matching M_2. If job v_{2j-1} is not used (saturated) in matching M_1, we can find a matching as follows for set $A \cup \{v\}$. We can delete edges $e_2, e_4, \ldots, e_{2j-2}$ from M_1, and insert edges e_1, e_3, $\ldots$, e_{2j-1} instead. This way we schedule job x_{2j-1} in addition to all other jobs that are scheduled in M_1. So the value of $F(A \cup \{v\})$ is at least x_{j-1} (the value of job x_{2j-1}) more than $F(A)$. We conclude that $F(A \cup \{v\}) - F(A) = x_{j-1}$ is at least $F(B \cup \{v\}) - F(B) = x_l$.

Finally we consider the case that v_{2j-1} is also saturated in M_1 using some edge e other than e_{2j}. Edges e and e_{2j} are in M_1 and M_2 respectively, and vertex v_{2j-1} is their common endpoint. So these two edges should come in the same connected component in the graph $M_1 \Delta M_2$. We proved that all connected components of $M_1 \Delta M_2$ are paths with odd number of edges that start and end with edges in M_2. Let Q be the path that contains edges e and e_{2j}. This path contains edges e_1', e_2', $\ldots$, $e_i' = e_{2j}$, $e_{i+1}' = e$, e_{i+2}', $\ldots$, $e_{2l'+1}'$. The last edge of this path, $e_{2l'+1}'$ matches a job v' with a time slot/processor pair. Let x' be the value of v'. Vertex v' is not scheduled in matching M_1. At first we prove that x' is at least x_l (the value of job v_{2l+1}). Then we show how to find a matching for set $A \cup \{v\}$ with value at least x' more than the value of M_1.

If x' is less than x_l, we can find a matching with greater value for set B instead of M_2. Delete edges $e_i' = e_{2j}$, e_{i+2}', e_{i+4}', $\ldots$, $e_{2l'+1}'$, and also edges e_{2j+2}, e_{2j+4}, $\ldots$, e_{2l} from M_2, and insert edges $e_{i+1}' = e$, e_{i+3}', $\ldots$, $e_{2l'}'$, and edges e_{2j+1}, e_{2j+3}, $\ldots$, e_{2l+1} to M_2 instead of the deleted edges. In the new matching, job v' with value x' is not saturated any more, but the vertex v_{2l+1} with value x_l is saturated. So the value of the new matching is $x_l - x' > 0$ more than the value of M_2 which is a contradiction. So x' is at least x_l.

Now we prove that there is a matching for set $A \cup \{v\}$ with value x' more than the value of M_1. We can find this matching as follows. Delete edges $e'_{i+1} = e$, e'_{i+3}, ..., $e'_{2l'}$, and edges e_2, e_4, ..., e_{2j-2}, and insert edges e'_{i+2}, e'_{i+4}, ..., $e'_{2l'+1}$, and edges e_1, e_3, ..., e_{2j-1}. This way we schedule job v' with value x' in addition to all other jobs that are scheduled in M_1. So we find a matching for set $A \cup \{v\}$ with value x' more than the value of M_1.

So $F(A \cup \{v\}) - F(A)$ is at least x'. We also know that $F(B \cup \{v\}) - F(B)$ is equal to x_l. Because x' is at least x_l, the proof is complete. □

Now we are ready to represent our algorithm which finds an optimal solution (with respect to values).

THEOREM 4.3. *If there is an schedule for the prize-collecting scheduling problem with value at least Z and cost B, there is an algorithm which finds a schedule with value at least Z and cost at most $O([\log n + \log \Delta]B)$ where δ is the ratio of the maximum value over the minimum value of all n jobs.*

PROOF. Let $v_{\max}$ and $v_{\min}$ be the maximum and minimum value among all n jobs respectively. We know that Z can not be more than $n \cdot v_{max}$. Define ε to be $\frac{v_{\min}}{n \cdot v_{max}} = \frac{1}{n\Delta}$. Using Theorem 4.1, we can find a solution with value at least $(1-\varepsilon)Z$ and cost at most $O(B\log(n\Delta)) = O([\log n + \log \Delta]B)$. Let S' be this solution. If the value of S' is at least Z, we exit and return this set as our solution. Otherwise we do the following. Note that we just need εZ more value to reach the threshold Z, and εZ is at most $v_{\min}$. So we just need to insert another interval which increases our value by at least $v_{\min}$. In the proof of Lemma 4.2, we proved that the value of $F(B \cup \{v\}) - F(B)$ is either zero or equal to the value of some jobs (in the proof it was x_l the value of vertex v_{2l+1}). So if we add an interval the value of set is either unchanged or increased by at least $v_{\min}$. So among all intervals with cost at most B, we choose one of them that increase our value by at least $v_{\min}$. At first note that this insertion reaches our value to Z, and our cost would be still $O([\log n + \log \Delta]B)$.

We now prove that there exists such an interval. Note that the optimum solution consists of some intervals $S_1, S_2, \ldots, S_k$. The union of these intervals, T has value $F(T)$ which is at least Z. So $F(T)$ is greater than the value of our solution $F(S')$. Using Lemma 2.1, $F(S' \cup S_i) - F(S')$ should be positive for some $1 \le i \le k$. We also know that the cost of this set is not more than B because the cost of the optimum solution is not more than B. So there exists a time interval (a set like S_i) that solves our problem with additional cost at most B. We also can find it by a simple search among all time intervals. □

Note that in the simple case studied in the literature, the values are all identical, and Δ is equal to 1.

5. REFERENCES

[1] J. Augustine, S. Irani, and C. Swamy. *Optimal power-down strategies.* In Proceedings of the 45th Symposium on Foundations of Computer Science, pages 530–539, Rome, Italy, October 2004.

[2] P. Baptiste. *Scheduling unit tasks to minimize the number of idle periods: a polynomial time algorithm for offline dynamic power management.* In Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm, pages 364–367, Miami, Florida, 2006.

[3] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Morteza Zadimoghaddam. *The submodular secretary problem and its extensions* Manuscript.

[4] Erik D. Demaine, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, Amin S. Sayedi-Roshkhar and Morteza Zadimoghaddam. *Scheduling to Minimize Gaps and Power Consumption.* In Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Pages 46-54, San Diego, California, June 2007.

[5] E. Even-dar, Y. Mansour, V. S. Mirrokni, M. Muthukrishnan, U. Nadav. *Bid Optimization for Broad-Match Ad Auctions.* In Proceedings of the 18th International World Wide Web Conference, Pages 231-240, Madrid, Spain, 2009.

[6] S. Irani, S. Shukla, and R. Gupta. *Algorithms for power savings.* In Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 37–46, Baltimore, Maryland, 2003.

[7] Johnson, D. S. *Approximation Algorithms for Combinatorial Problems.* In Proceedings of the fifth annual ACM Symposium on Theory of Computing, Pages 38-49, Austin, Texas, 1973.

[8] Nitish Korula and Martin Pal. *Algorithms for Secretary Problems on Graphs and Hypergraphs.* In Proceedings of the 36th International Colloquium on Automata, Languages and Programming, Pages 508-520, Rhodes, Greece, July, 2009.

[9] Ariel Kulik, Hadas Shachnai, and Tami Tamir. *Maximizing submodular set functions subject to multiple linear constraints.* In Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms, Pages 545-554, New York, 2009.

[10] Jon Lee, Vahab S. Mirrokni, Viswanath Nagarajan, Maxim Sviridenko. *Non-monotone submodular maximization under matroid and knapsack constraints.* In Proceedings of the 41th annual ACM Symposium on Theory of Computing, Pages 323-332, Bethesda, Maryland, 2009.

[11] Raz, R., and Safra, S. *A sub-constant error-probability low-degree test, and sub-constant error-probability PCP characterization of NP.* In Proceedings of the 29th annual ACM Symposium on Theory of Computing, Pages 475-484, El Paso, Texas, 1997.

APPENDIX

A. HARDNESS RESULTS

Here we show some matching hardness results to show that our algorithms are optimal unless $P = NP$. Surprisingly the problem we studied does not have better than $\log n$ approximation even in very simple cases, namely, one interval scheduling with nonuniform parallel machines, or multi-interval scheduling with only one processor.

It is proved in [4] that the multi-interval scheduling problem with only one processor and simple cost function is Set-Cover hard, and therefore the best possible approximation factor for this problem is $\log n$. We note that in the simple cost function the cost of an interval is equal to its length plus a fixed amount of energy (the restart cost). All previous work studies the problem with this cost function. In fact, Theorem 7 of [4] shows that the problem does not have a $o(\log N)$-approximation even when the number of time intervals of each job is at most 2 (each job has a set of time intervals in which it can execute).

THEOREM A.1. *It is NP-hard to approximate* 2*-interval gap scheduling within a* $o(\log N)$ *factor, where* N *is the size of input.*

Now we show that the one-interval scheduling problem, for which there exists a polynomial-time algorithm in [4], does not have any $o(\log N)$-approximation when only a subset of processors are capable of executing a job. Assume that each job has one time interval in which it can execute, and for each job, we have a subset of processors that can execute this job in its time interval, i.e., the other processors do not have necessary resources to execute the job. We also consider the generalized cost function in which the cost of an interval is not necessarily equal to its length plus a fixed amount. We call this problem *one-interval scheduling with nonuniform processors.*

THEOREM A.2. *It is NP-hard to approximate one-interval scheduling with nonuniform processors problem within a* $o(\log N)$ *factor, where* N *is the size of input.*

PROOF. Like previous hardness results for these scheduling problems, we give an approximation-preserving reduction from Set Cover, which is not $o(\log n)$-approximable unless P = NP [11]. Let $E = \{e_1, e_2, \ldots, e_n\}$ be the set of all elements in the Set-Cover instance. There are also m subsets of E, $S_1, S_2, \ldots, S_m$ in the instance. We construct our scheduling problem instance as follows. For each set S_j, we put a processor P_j in our instance. For each element e_i, we put a job j_i. Only jobs in set S_j can be done in processor P_j. The time interval of all jobs is $[1, n]$. The cost of keeping each processor alive during a time interval is 1. Note that the cost a time interval is not a function of its length in this case, i.e., the cost of an interval is almost equal to a fixed cost which might be the restart cost. So the optimum solution to our scheduling problem is a minimum size subset of processors in which we can schedule all jobs because we can assume that when a processor is alive in some time units, we can keep that processor alive in the whole interval $[1, n]$ (it does not increase our cost). In fact we want to find the minimum number of subsets among the input subset such that their union is E. This is exactly the Set Cover problem. □

B. POLYNOMIAL-TIME ALGORITHM FOR PRIZE-COLLECTING ONE-INTERVAL GAP MINIMIZATION PROBLEM

The simple cost function version of our problem is studied in [2, 4] as the gap-minimization problem. Each job has a time interval, and we want to schedule all jobs on P machines with the minimum number of gaps. (A gap is a maximal period of time in which a processor is idle, which can be associated with a restart for one of the machines.) There are many cases in which we can not schedule all jobs according to our limitation in resources: number of machines, deadlines, etc. So we define the prize-collecting version of this simple problem. Assume that each job has some value for us, and we get its value if we schedule it. We want to get the maximum possible value according to some cost limits. Formally, we want to schedule a subset of jobs with maximum total value and at most g gaps. The variable g is given in the input. Now we show how to adapt the sophisticated dynamic program in [4] to solve this problem.

THEOREM B.1. *There is a* $(n^7 p^5 g)$*-time algorithm for prize-collecting* p*-processor gap scheduling of* n *jobs with budget* g*, the number of gaps should not exceed* g*.*

PROOF. In the proof of Theorem 1 of [4], C_{t_1,t_2,k,q,l_1,l_2} is defined to be the number of gaps in the optimal solution for a subproblem defined there. If we define $C'_{t_1,t_2,k,q,l_1,l_2,g'}$ to be the maximum value we can get in the same subproblem using at most $g' \leq g$ gaps, we can update this new dynamic program array in the same way. The rest of the proof is similar; we just get an extra g in the running time. □