Approximation Algorithms: Greedy Algorithm and Local Search

Zixuan Fan*

April 25, 2024

1 Introduction

The theory of computer science has grown significant throughout the past century. Among many topics, **NP**-hardness has drawn the attention of many researchers. While there is no proof whether $\mathbf{P} = \mathbf{NP}$, people tend to be interested in compensation: solving NP-hard problems with a sub-optimal solution in polynomial time. This is the motivation of approximation algorithms. This paper introduces two fundamental techniques in designing approximation algorithms: greedy algorithm and local search. We start with some mathematical backgrounds, followed by a brief introduction of greedy and local search. Then, we present a few examples that exploit these techniques.

2 Mathematical Backgrounds

2.1 Approximation Ratio

Approximation algorithms attempt to solve optimization problems with a sub-optimal solution in polynomial time. Hence, in this context, we refer to an **NP**-hard problem as an **NP**-hard optimization problem instead of a decision problem. With the goal problems clarified, we introduce the concept of approximation ratio.

Definition 1 (Approximation Ratio)**.** *Given an optimization problem, its optimal solution Opt^*, and a sub-optimal solution Opt given by algorithm A, suppose the size of solutions are measured by $|\cdot|$. The approximation ratio α of algorithm A is defined as*

$$\alpha = \frac{|Opt|}{|Opt^*|}$$

The approximation ratio α shows how close the solution Opt is to the optimal solution Opt^*. For an maximization problem, $\alpha < 1$, while $\alpha > 1$ for a minimization problem. The better the approximation, the closer α is to 1.

2.2 Greedy Algorithms and Local Search

Both greedy algorithms and local search contructs the solution step by step. In each step, the strategy makes certain desicions such that the result is locally optimal. The difference between two techniques lies in the

*Department of Computer Science, Technical University of Munich. ge43yeb@mytum.de

strategy of making decisions. The greedy algorithms try to make the best decision at each step, the solution is not guaranted to be *feasible* at the beginning. On the other hand, local search starts with an arbitrary feasible solution, and maintains the feasibility while improving the solution. Hence greedy algorithms are called *primal infeasible* algorithms, while local search algorithms are referred to as *primal feasible* algorithms.

3 Scheduling on Single Machine

Scheduling or Job Sequencing, is one of the earliest discovered NP-complete problems. It was amongst the first 21 NP-complete problems published by Karp in 1972. In this section, we examine a simplified version of the problem: only one machine is available for scheduling.

Definition 2 (Scheduling on Single Machine)**.** *Given n jobs to be processed with processing time p_j, release time $r_j \geq 0$, and deadline d_j with $j = 1, \cdots, n$. Suppose j-th job is completed at time C_j, we define the lateness as $L_j = C_j - d_j$. The goal is minimizing the maximal lateness*

$$L_{max} = \max_{j=1,\cdots,n} L_j$$

Unfortunately, the problem is **NP**-hard, and it remains **NP**-hard even if we apply a few constraints to it. **cite the source** One exceptional case is when all deadlines are non-negative: the lateness is always positive. We are able to give a 2-approximation algorithm for this case. The algorithm simply exploits the earliest due date rule(**EDD**). Just as the name suggests, whenever the machine finishes a job, it picks the job with the earliest due date from all **available** jobs. Here, a job is *available* at time t if $r_j \leq t$.

To analyze the algorithm, we have to introduce a few notations. Let S denote a set of jobs, $r(S) = \min_{j \in S} r_j$, $p(S) = \sum_{j \in S} p_j$, and $d(S) = \max_{j \in S} d_j$. While L_{max} denotes the maximal lateness computed by the algorithm, L^*_{max} denotes the optimal solution. As preparation, an interesting lowerbound on the optimal solution is observed.

Lemma 3. *For any set of jobs S, $L^*_{max} \geq r(S) + p(S) - d(S)$*

Proof. Suppose job j is the last finished job in the optimal schedule. The scheduling cannot start until $r(S)$. In an optimal schedule, there is no gap between the processing of jobs, all jobs are processed consecutively in $p(S)$. Hence j cannot be finished until $r(S) + p(S)$. In addition, we have $d_j \leq d(S)$ by definition. It follows

$$L^*_{max} \geq L_j = C_j - d_j \geq r(S) + p(S) - d_j \geq r(S) + p(S) - d(S)$$

□

We then take a closer look at the **EDD** rule. Suppose the release time r_j, processing time p_j, and the deadline d_j are encoded as a ternary tuple (r_j, p_j, d_j). The only part the requires heavy computation for **EDD** rule is choosing the job with the earliest due date. The easiest implementation is preserve a linked list to store the information of each job, which takes $\mathcal{O}(n^2)$ time in total. A more efficient implementation takes advantage of a FIFO queue, which takes only $\mathcal{O}(n)$ time. In either cases, the algorithm runs in polynomial time. This directly leads to the following theorem.

Theorem 4. *The EDD-rule algorithm runs in polynomial time.*

What we care more in the context of approximation algorithms is the approximation ratio. We then show that the algorithm yields the promised approximation ratio.

Theorem 5. *The EDD-rule algorithm yields a 2-approximation ratio.*

Proof. Suppose j is the job with the maximal lateness in the schedule generated by the **EDD** rule. Let t be the earliest time such that the machine keeps processing jobs continuously in the interval $[t, C_j]$. Let S be the set of jobs that are processed in this interval. In our assumption, $r(S) = t$ is guaranteed. Suppose for contradiction that $r(S) > t$, the processing would not be able to start at time t. Similarly, if $r(S) < t$, the processing would start earlier than t. Both cases contradict the assumption. Furthermore, $p(S) = C_j - t$, for the processing is continuous and it stops at time C_j. Applying lemma lemma 3, we obtain

$$L^*_{max} \geq r(S) + p(S) - d(S) = t + C_j - t - d(S) = C_j - d(S) \geq C_j$$

Additionally, it holds $d_j < d(S)$ by definition. Applying the same lemma, we have

$$L^*_{max} \geq r(S) + p(S) - d(S) \geq -d(S) \geq -d_j$$

Observe that $L_{max} = C_j - d_j$. Hence in this schedule, it holds

$$L_{max} = C_j - d_j \leq 2L^*_{max}$$

which justifies the 2-approximation ratio. □

4 Scheduling on Identical Parallel Machines

Just as we can run k-bands of Turing machines in parallel, jobs can be processed simultaneously on k machines, if available. The problem setting is then a bit different: there is no release date of jobs, and the optimization goal is to minimize the time all jobs are finished.

Definition 6 (Job Scheduling on Identical Parallel Machines)**.** *Given n jobs to be processed on k identical parallel machines, each job j has a processing time p_j with $j = 1, \cdots, n$. Suppose job j is finished at time C_j, the minimization goal, the makespan, is defined as*

$$C_{max} = \max_{j=1,\cdots,n} C_j$$

For this problem, we present a local search algorithm and a greedy algorithm. The ideas of both algorithms are both simple, while the analysis requires some effort.

4.1 Local Search Algorithm

The local search algorithm starts with an arbitrary schedule. In each step, we try to find a better schedule for the last job to be completed. More specifically, we traverse all machines, and try to move the job to another machine such that it finishes earlier. Whenever no such move is possible, the algorithm terminates. To simply the analysis, we assume that there is no idle time on any machine in the initial schedule. Again, we start with a lowerbound on the size of the optimal solution.

Lemma 7. *For a scheduling problem with n jobs and k identical machines, its optimal makespan is less equal than the average processing time of all jobs.*

$$C^*_{max} \geq \frac{1}{k} \sum_{j=1,\cdots,n} p_j$$

Proof. We perform a case distinction

- All machines terminate at the same time. In this case, each machine runs exactly the average processing time, i.e. the equality holds in the lemma.
- At least two machines do not finish at the same time. In this case, the machine that finishes later runs more than the average processing time. Furthermore, the optimal solution has to run no shorter than this machine. Hence the strict inequality holds in the lemma.

□

Using the same idea, we observe that the starting time of the last job satisfies an upperbound.

Observation 8. *Let j be the last job completed in the local search algorithm. We denote its starting time as $S_j = C_j - p_j$. It holds*

$$S_j \leq \frac{1}{k} \sum_{j=1,\cdots,n} p_j$$

Proof. If the last job is completed just as the average processing time, its starting time has to be strictly less than the average processing time. Otherwise, we suppose for contradiction that $S_j > \frac{1}{k} \sum_{j=1,\cdots,n} p_j$. By the strategy of the local search algorithm, all other machines are still running at time S_j, otherwise the job can be moved to another machine. Hence all machines terminate later than S_j, implying that all machines terminate later than the average processing time. Since not all machines can terminate later than the average processing time, this is a contradiction. □

Combining the two observations, we obtain $C^*_{max} > S_j$. Observe that $p_j \leq C_j$ by definition. Hence we obtain the approximation ratio for this local search algorithm.

Theorem 9. *The local search algorithm yields a 2-approximation ratio.*

Proof. Let j be the last job completed. We have

$$C_{max} = C_j = S_j + p_j \leq C^*_{maxc} + p_j \leq 2C^*_{max}$$

□

4.2 Greedy Algorithm

The greedy algorithm is similar to the **EDD** rule in the previous section. Whenever a machine is available, we assign it a job from the job list. Since there is no release date, the choice is arbitrary. This is referred to as **list scheduling** algorithm.

References

[1] R.J. Ahuja, T.L. Magnanti and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.