Approximation Algorithms: Greedy Algorithm and Local Search

Zixuan Fan[*]

May 2, 2024

1 Introduction

The theory of computer science has grown significantly 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, which 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 construct the solution step by step. In each step, the strategy makes certain decisions to optimize the result locally. The difference between the two techniques lies in the strategy

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

of making decisions. The greedy algorithms try to make the best decision at each step, and the solution is not guaranteed 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 A 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 to minimize 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 lower bound 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, and 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 that requires heavy computation for **EDD** rule is choosing the job with the earliest due date. The easiest implementation is to 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 case, 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 about 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 for the machine to keep 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, it holds for this schedule

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

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 different: there is no release date for 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 so that it finishes earlier. Whenever no such move is possible, the algorithm terminates. To simplify the analysis, we assume that there is no idle time on any machine in the initial schedule. Again, we start with a lower bound 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 than or equal to 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.
Case 1: 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.
Case 2: 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, strict inequality is held in the lemma. □

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

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 as soon 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

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

□

While it is easy to find an approximation ratio, the polynomial time complexity of the local search algorithm is not trivial. Observe that the algorithm terminates only when no job can be moved to improve the schedule.
omit the analysis(and refinement) here? cuz not enough space

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. The execution of the list scheduling algorithm can be viewed as a special case of the **local search** algorithm, where the initial schedule cannot be improved.

Theorem 10. *The **list scheduling** algorithm yields a* 2*-approximation ratio.*

Proof. Let j be the last job completed in a schedule output by the **list scheduling** algorithm. Suppose the **local search** algorithm can improve the schedule. Then, we can find a machine such that $S'_j < S_j$ where S'_j is the new starting time of j after moving it to this new machine. However, the greedy strategy always assigns the job to the first available machine, so no such machine exists. Thus, the schedule cannot be improved by **local search** algorithm. Hence, its approximation ratio can also bounded by $\alpha = 2$ as in theorem theorem 9. □

If we execute the **list scheduling** algorithms on some examples fill in graphs if needed , we can observe that the longer the processing time of the last job, the worse the approximation ratio is. If we process the longer jobs first and fill in the shorter jobs later, the approximation ratio may be improved. This is the idea of the longest processing time rule(**LPT**). As expected, it yields a better approximation ratio.

Theorem 11. *The **LPT**-rule algorithm yields a* $4/3$*-approximation ratio.*

Proof. Recall in lemma lemma 7, we have proven that the optimal makespan is bounded by the average processing time. Let $E = \sum_{j=1,\cdots,n} p_j$ as the average processing time. Suppose for contradiction that $C_{max} > \frac{4}{3}C^*_{max}$ for the last completed job j. When j started, its starting time $C_j - p_j$ must be less than or equal to E. If not, there must have been another machine finishing at time E or earlier, to which we could have moved job j for a better schedule. Hence we have

$$p_j \geq C_j - E = C_{max} - E > \frac{4}{3}C^*_{max} - C^*_{max} = \frac{1}{3}C^*_{max}$$

In **LPT**-rule, we can assume the last job completed always has the shortest processing time. If this is not the case, we simply discard the shorter jobs while the new job list still has the same makespan. Thus, for each job $i = 1, \cdots, n$, it holds $p_i \geq p_j > \frac{1}{3}C^*_{max}$. There are, thus, at most two jobs on each machine. Otherwise, the optimal makespan would exceed C^*_{max}. We are able to show that **LPT**-rule yields the optimal solution by case distinction.

Case 1: There is no job processed before j on the same machine. Since j is the shortest job, we conclude that each machine processes at most one job, and the optimal solution is obtained.

Case 2: There is a job processed before j on the same machine. Since there are at most two jobs on each machine, we may assume that there are $k + k'$ jobs in total, where $k' \leq k$. Sort the jobs reversely with regard to their processing time. The optimal schedule is assigning $(k + i)$-th job directly after the i-th job, where $i < k'$. If $(k + k')$-th job is assigned after a job finishes later than the k'-th job, it will definitely finish later than the optimal schedule. If it is assigned after a job finishing earlier than the k'-th job, a job that takes longer will be assigned after the k'-th job, which is also suboptimal.

To summarize, the optimal schedule is obtained by the **LPT**-rule algorithm, which contradicts the assumption $C_{max} > \dfrac{4}{3}C^*_{max}$. □

5 k-Center Problem

Many graph problems are known to be **NP**-hard. Furthermore, greedy algorithms are known to be helpful in computing many graph structures in polynomial time, such as minimum spanning trees, shortest paths, etc. Unaccidentally, this strategy is also helpful in terms of approximation algorithms. In the following sections, we introduce the approximation for k-Center, Travelling Salesman Problem, and Edge Coloring.
k-Center is a classical problem in computational geometry. It is also related to well-studied k-means clustering in machine learning. The problem is defined as follows.

Definition 12 (k-Center)**.** *Given a set of n points P in a metric space with distance function $|\cdot|$. Find k centers $S \subseteq P$ so that the maximal distance from any point to its nearest center is minimized.*

$$r = \max_{p \in P} d(p, S)$$

where $d(p, S) = \min_{s \in S} |s - p|$

The problem can be reduced to a problem in an undirected weighted graph, where the weight is the distance between two vertices. The greedy algorithm for this problem starts with an arbitrary center. In each iteration, we pick the point that is farthest from the current centers, i.e., choosing $\arg\max d(p, S)$ from $p \in P \backslash S$. By adding such a point to the center set in $(k-1)$ iterations, we obtain an approximation for k-Center.

Theorem 13. *The presented greedy problem yields a 2-approximation ratio for k-Center.*

Proof. Let r^* denote the optimal distance radius. The distance between any point p to its nearest center is bounded by $|p - s| \leq r^*$. Let q be another point from the same cluster. By triangular inequality, we have

$$|p - q| \leq |p - s| + |s - q| \leq 2r^*$$

Thus, the distance between points in the same cluster is bounded by $2r^*$. If the presented greedy algorithm chooses a center from each cluster, the approximation ratio is guaranteed. For an arbitrary point, if the center from its cluster is its nearest neighbor, the presented bound of $2r^*$ is sufficient. Otherwise, there would be a nearer center, which is also dominated by the bound of $2r^*$.
A more interesting case is when not all clusters have a point selected as a center. There would be then at least two points, p and q, from the same cluster. By the presented upper bound, we know $|p - q| \leq 2r^*$. In our algorithm, we always choose the point with maximal $d(p, S)$. Hence when both p and q were added into S, it holds

$$d(p', S) \leq d(p, S) \leq |p - q| \leq 2r^*$$

for an arbitrary p' not in S, which justifies the 2-approximation ratio for points whose cluster has no center selected. □

In fact, $\alpha = 2$ is the best possible approximation for k-Center , if **P** $\neq$ **NP**. To prove this, we reduce k-Center from Dominating Set, an **NP**-hard decision problem.

Definition 14 (Dominating Set)**.** *Given an undirected graph $G = (V, E)$, we need to find a set $S \subseteq V$ of size k s.t. each vertex in V is either in S or adjacent to a vertex in S.*

Lemma 15. *If there is a polynomial-time approximation algorithm for k-Center with $\alpha < 2$, then there is a polynomial-time algorithm for deciding Dominating Set.*

Proof. We start with a reduction from Dominating Set to k-Center. For an instance of Dominating Set, let the distance between adjacent vertices be 1, and the distance between non-adjacent vertices be 2. It is easy to see that there exists a Dominating Set of size k if and only if there exists a k-Center with radius 1. If k-Center is approximated with $\alpha < 2$, the radius given by this problem would be

$$r = \lfloor \alpha r^* \rfloor = \lfloor \alpha \rfloor = 1$$

Hence, the approximation yields the optimal solution for k-Center, which results in a polynomial-time algorithm to decide Dominating Set. □

Since Dominating Set is **NP**-hard, we may conclude that $\alpha = 2$ is the best possible approximation given that **P** $=$ **NP** is not solved yet.

Theorem 16. *k-Center cannot be approximated with $\alpha < 2$ unless **P** $=$ **NP**.*

6 Traveling Salesman Problem

The Traveling Salesman Problem(**TSP**) is one of the most studied combinatorial optimization problems. In this problem, we want to minimize the total cost of the salesperson's tour, which visits each city and returns to the initial city. By convention, we assume the cost between two cities is non-negative and symmetric.

Definition 17 (TSP)**.** *Give an undirected completed graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}^+$, find a path $\pi \in V^*$ such that all vertices in V exist exactly once in π, and its total cost is minimized.*

For a general case of **TSP**, a polynomial-time approximation exists if and only if **P** $=$ **NP**. This is proven by a reduction from Hamiltonian Cycle to **TSP**.

Definition 18 (Hamiltonian Cycle)**.** *Given an undirected graph $G = (V, E)$, find a cycle that visits each vertex exactly once.*

Theorem 19. *For any $\alpha > 1$, Hamiltonian Cycle is solvable in polynomial time if there is a polynomial-time α-approximation algorithm for **TSP**.*

Proof. Given an instance of Hamiltonian Cycle $G = (V, E)$, we need to fill in the missing edges such that G becomes a completed graph for **TSP**. Pick $\alpha > 1$ arbitrarily. We assign a cost of 1 to the existing edges and a cost of $(\alpha - 1)n + 2$ to the missing edges where $n = |V|$. If there is a Hamiltonian Cycle in G, the total cost of the tour is n. Otherwise, the total cost of the cycle is at least $\alpha n + 1$. If there exists a polynomial-time α-approximation algorithm for **TSP**, the cost output by this algorithm is at most $\alpha n < \alpha n + 1$. Thus, there exists a Hamiltonian Cycle in G if such approximation exists. □

Since a polynomial-time approximation is not possible for general **TSP**. We may wonder if any constraints can be added to the problem to enable a polynomial-time approximation. If we consider the cost function as the distance between two real cities, an interesting property, the triangular inequality, can be exploited. In fact, the triangular inequality holds for any metric space. Hence, we may consider the **metric TSP** problem.

Definition 20 (metric TSP)**.** *Let $G = (V, E)$ and $c : E \to \mathbb{R}^+$ be an instance of **TSP**. It is an instance of metric **TSP** if and only if the triangular inequality holds for all edges in E.*

$$\forall i, j, k \in V.c(i, j) + c(j, k) \geq c(i, k)$$

While a round trip in metric **TSP** visits all cities and returns to the initial city, a minimum spanning tree(**MST**) connects all cities with minimum total cost without forming a cycle. We may wonder if **MST** can be used for analysis of approximation or simply as a sub-routine in the algorithm. In fact, if we break any edge in the round trip, a spanning tree can be obtained. This spanning tree must have a total cost less than or equal to the cost of the **MST**. This leads to the following lemma.

Lemma 21. *For any metric **TSP** instance, the cost of the optimal tour is at least the cost of the **MST** on the same instance.*

From now on, we denote the cost of the **MST** as c_{MST} and the cost of the optimal tour as c^*. lemma 21 then yields $c_{MST} \leq c^*$. which is useful in the proofs of the next three algorithms.

6.1 A Greedy Algorithm

The most straightforward idea of a greedy algorithm for **TSP** is adding a vertice with the lowest cost until all vertices are visited. This is referred to as the *nearest addition algorithm*. To be more specific, the algorithm starts with a round tour between an arbitrary vertice and its nearest neighbor. In each iteration, let S be the set of visited cities. We pick $v \in S$ and $u \notin S$ such that $c(u, v)$ is minimized. Let w be the vertex following v during the current tour. We discard the edge (v, w) and add (v, u) and (u, w) to the tour. The algorithm terminates when all vertices are visited and hence runs in linear time. One may observe that the greedy strategy for *nearest addition algorithm* is similar to that of *Prim's algorithm* for **MST**. Combining this observation and lemma 21, we obtain the following theorem. cite prim

Theorem 22. *The nearest addition algorithm yields a 2-approximation ratio for metric **TSP**.*

Proof. Consider the edge $e = (u, v)$ found in each iteration. We claim that e is precisely the same edge found by *Prim's algorithm*. In each step, *Prim's algorithm* also chooses the minimum cost edge $e' = (v', u')$ such that $v' \in S$ and $u' \notin S$, while S is the **MST**built step by step. While the sets of edges that we preserve between iterations are different in the two algorithms, the set of vertices S is exactly the same. Hence, the edge e is the same as e'. Let T be the set of edges picked in the iterations along with the initial edge. Apparently, T is an **MST**. The next step is bounding the cost of a round tour with the cost of T.
Consider the cost change in each iteration. While (u, v) and (v, w) are added to the tour, (u, w) is removed. The cost change is $c(u, v) + c(v, w) - c(u, w)$. Using the triangular inequality, we have $c(u, v) + c(u, w) \geq c(v, w)$. Combining both, we obtain

$$c(u, v) + c(v, w) - c(u, w) \leq 2c(u, v)$$

Recall that the initial cost of the tour in *nearest addition algorithm* is $2c(e_0)$ where e_0 is the initial edge because we started with a round tour between two vertices. Since each (u, v) chosen in the iteration is distinct, the total cost of the tour is bounded by twice of the cost of T. Applying lemma 21, we have

$$c \leq 2c_{MST} \leq 2c^*$$

which justifies the 2-approximation ratio. □

6.2 A Little Help from Euler

While Hamiltonian Cycle is **NP**-hard, deciding if there is a path that traverses each edge exactly once is solvable in polynomial time. This is known as the Eulerian Path. Thanks to Euler's efforts in bridges of Königsberg, we know that a graph has an Eulerian Path if all vertices have an even degree.
Here comes an even simpler greedy algorithm: compute the **MST**, create a copy of each edge, and add those copies back to the **MST**. The resulting graph has an Eulerian Path. Find a Eulerian Path

$$\pi = (v_1, v_2) \cdots (v_i, v_{i+1}) \cdots (v_{n-1}, v_n)$$

We now have a tour visiting each vertex, but we still have to remove all the repeated vertices. This is also simple: remove all repeated existence of vertices in π and connect the gap between two vertices with the edge from the original graph. In fact, this is just the same as the *nearest addition algorithm* with a manner of analysis. Thus, this *double-tree algorithm* has exactly the same approximation ratio.

Theorem 23. *The double-tree algorithm yields a* 2*-approximation ratio for metric* ***TSP****.*

Proof. The Eulerian Path over the **MST** has the cost of $2c_{MST}$, which is already a nice upper bound. It suffices to show that the modification on the Eulerian Path does not increase the cost.
Let v_i and v_j be two vertices in the Eulerian Path. Furthermore, they have their first existence at i-th and j-th position, respectively. Suppose all vertices between them have existed earlier than v_i. We have to remove the edges $(v_i, v_{i+1}), \cdots, (v_{j-1}, v_j)$, and insert a new edge (v_i, v_j) into the path. The cost change would be

$$c_i = c(v_i, v_j) - \sum_{k=i}^{j-1} c(v_k, v_{k+1})$$

By applying triangular inequality repeatedly, we are able to show $c_i \leq 0$ visualization if needed . This simply means that the modification in each step does not increase the cost. Hence, the 2-approximation ratio is justified. □

6.3 A Little More Help from Christofides

Copying the edges of the **MST** is a simple idea, but it is not the best idea. Is there another way to construct an Eulerian Path with less cost? The answer is yes. Christofides proposed a $\frac{3}{2}$-approximation taking advantage of Perfect Matching.

Definition 24 (Perfect Matching)**.** *Given a graph $G = (V, E)$. A matching is a set of edges such that no two edges share a common vertex. A matching is perfect if and only if each vertex in V is incident to exactly one edge in the matching.*

How do we take advantage of Perfect Matching? We would like to return to the existence of the Eulerian Path. For any tree, there exists no Eulerian Path, for leaves can only have an odd degree. However, if we connect them with edges from perfect matching, the degree of each vertex becomes even. This applies not only to leaves but also to internal nodes with odd degrees. Let O be the set of vertices with odd degree in the **MST**. We still need to prove that a perfect match exists.

Lemma 25. *For any tree T, the number of vertices with odd degrees is even.*

Proof. In any graph, we have $|E| = 2|V|$ and $\sum_{v \in V} \deg(v) = 2|E|$. Observe that the sum of even numbers is always even. Hence, the total degree of vertices with an even degree is even. We denote this value with $2k$. It follows that the total degree of vertices with odd degrees is also even.

$$\sum_{v \in O} \deg(v) = \sum_{v \in V} \deg(v) - \sum_{v \in V \setminus O} \deg(v) = 2|E| - 2k = 2(|E| - k)$$

Since the sum of odd numbers is even if and only if the number of odd numbers is even, the lemma is proven. □

lemma 25 shows that $|O|$ is even. In addition, we have a complete graph in **TSP**. Thus, a perfect matching exists for O. We add the edges of the perfect matching to the **MST**, obtaining a graph with a Eulerian Path. The rest of the work is the same as the *double-tree algorithm*. It can be inferred that the modification on the Eulerian Path does not increase the cost as in the proof of theorem 23. The approximation ratio for *Christofides' algorithm* is dependent on the size of the Perfect Matching.

Theorem 26. *Christofides' algorithm yields a $\frac{3}{2}$-approximation ratio for metric **TSP**.*

Proof. We construct a tour on vertices from O. This can be constructed from the optimal tour. Let v and u be two vertices from O, between which there are only vertices not from O. Using the same idea as in theorem 23, we discard those intermediate vertices and insert a new edge (v, u). The cost never increases. Let c_O denote the cost of this tour, it holds

$$c_O \leq c^*$$

Since the number of vertices is even, the number of edges in this tour is also even. We can then construct two Perfect Matchings by coloring: pick an arbitrary edge with color 1 and color its neighbor with color 2. Repeat the procedure until all edges are colored. We always pick the smaller one for *Christofides' algorithm*. Let c_1 and c_2 be the cost of the two Perfect Matchings. It follows

$$\min(c_1, c_2) \leq \frac{2 \cdot \min(c_1, c_2)}{2} \leq \frac{c_1 + c_2}{2} \leq \frac{c_O}{2} \leq \frac{c^*}{2}$$

Hence, the size of the Eulerian Path in *Christofides' algorithm* is bounded by

$$c_{\text{Christofides}} = c_{MST} + \min(c_1, c_2) \leq \frac{3}{2} c^*$$

which justifies the $\frac{3}{2}$-approximation ratio. □

7 Conclusion

In this paper, we studied the techniques of local search and greedy algorithms in the context of approximation algorithms. We explored two topics: scheduling and graph problems. For all studied problems, an excellent constant approximation ratio is obtained. There are also other techniques in approximation, such as linear programming, randomization, and the primal-dual method. We also noticed that there is a limit to the approximation ratio for some problems. This leaves people with space and imagination to explore more significant questions of **NP**-completeness and other complexity classes such as **APX**.

References

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