COMPSCI 530: Design and Analysis of Algorithms 12/3/2013

Lecture # 16

Lecturer: Debmalya Panigrahi *Scribe: Utku Sirin*

1 Overview

In the previous lecture we covered polynomial time reductions and approximation algorithms for vertex cover and set cover problems. By reductions we showed that SAT, 3SAT, Independent Set, Vertex Cover, Integer Programming, and Clique problems are NP-Hard. In this lecture we will continue to cover approximation algorithms for maximum coverage and metric TSP problems. We will also cover Strong NP-hardness, PTAS/FPTAS topics; and Knapsack and Bin Packing problems.

2 Maximum Coverage

In this section, we will cover unweighted maximum coverage problem. Note that there are different versions of maximum coverage problem such as weighted and/or budgeted maximum coverage problems. We will cover only unweighted case. Firstly, we will give the definition of the problem; then we will give a greedy approximation algorithm; lastly, we will give analysis of the greedy algorithm concluding that the greedy algorithm has $(1-\frac{1}{e})$ approximation factor.

2.1 Problem Definition

Unweighted maximum coverage problem is defined as following. Given a set of sets $U = \{S_1, S_2, ..., S_m\}$, and a number k; select at most k sets $S' = S_{i_1}, S_{i_2}, ..., S_{i_k}$ from U such that the number of covered elements, i.e., the size of the union of S', $|\bigcup_{j=1}^{k} S_{i_j}|$, is maximized. Observe that the sets in U may have some elements in common.

2.2 Greedy Approximation

It is know that maximum coverage problem is NP-hard. However, we can give a greedy approximation algorithm whose approximation factor is $(1-\frac{1}{e})$. The algorithm is as following.

Algorithm 1: Greedy Maximum Coverage

```
1 Input: Set of sets $U$, $k$
2 for $i = 1...k$ do
3     pick the set that covers the maximum number of uncovered elements
4 end
```

Observe that we select exactly k-many sets, and at each step we pick the one that maximizes the number of uncovered elements.

2.3 Analysis of Greedy Approximation

Now, we will show the analysis of the greedy approximation algorithm for maximum coverage problem. Throughout the analysis, OPT denotes the optimal solution of the maximum coverage problem; a_i denotes the number of newly covered elements at the i^{th} iteration; b_i is the total number of covered elements up to the i^{th} iteration (including the i^{th} iteration), i.e., $b_i = \sum_{j=1}^{i} a_j$; and c_i is the number of uncovered elements after the i^{th} iteration, i.e., $c_i = OPT - b_i$. Moreover, $a_0 = b_0 = 0$, and $c_0 = OPT$ [1].

Lemma 1. *Number of newly covered elements at the $(i+1)^{th}$ iteration is always greater than or equal to $\frac{1}{k}$ of the number of uncovered elements after the i^{th} iteration, i.e., $a_{i+1} \geq \frac{c_i}{k}$.*

Proof. Optimal solution covers OPT elements at k iterations. That means, at each iteration there should be some sets in U whose size is greater than or equal to the $\frac{1}{k}$ of the remaining uncovered elements, i.e., $\frac{c_i}{k}$. Otherwise, it was impossible to cover OPT many elements at k steps by the optimal solution. Since the approximation algorithm is greedy, i.e., choosing always the set covering maximum number of uncovered elements, the chosen set at each iteration should be at least the $\frac{1}{k}$ of the remaining uncovered elements. That is, $a_{i+1} \geq \frac{c_i}{k}$. □

Lemma 2. $c_{i+1} \leq (1-\frac{1}{k})^{i+1} \cdot OPT$

Proof. We will prove by induction. Let us firstly show that the claim is true for $i = 0$.

$$\begin{aligned}
c_1 &\leq (1-\frac{1}{k}) \cdot OPT && (1)\\
c_1 &\leq OPT - OPT \cdot \frac{1}{k} && (2)\\
OPT - b_1 &\leq OPT - OPT \cdot \frac{1}{k} \quad \text{; since } c_1 = OPT - b_1 && (3)\\
-b_1 &\leq -OPT \cdot \frac{1}{k} \quad \text{; } OPT\text{'s cancels each other} && (4)\\
b_1 &\geq OPT \cdot \frac{1}{k} && (5)\\
a_1 &\geq OPT \cdot \frac{1}{k} \quad \text{; since } b_1 = a_1 && (6)\\
a_1 &\geq c_0 \cdot \frac{1}{k} \quad \text{; since } c_0 = OPT \text{ by our initial assumptions} && (7)
\end{aligned}$$

By lemma 1, we actually know that $a_1 \geq c_0 \cdot \frac{1}{k}$. Hence, the proof for $i = 0$ follows. Now, for inductive hypothesis assume $c_i \leq (1-\frac{1}{k})^i \cdot OPT$ is true, and show that $c_{i+1} \leq (1-\frac{1}{k})^{i+1} \cdot OPT$ is true.

$$
\begin{aligned}
c_{i+1} &= c_i - a_{i+1} \quad ; \text{by definition of } c_i = OPT - \sum_{j=1}^{i} a_j && (8)\\
c_{i+1} &\leq c_i - \frac{c_i}{k} \quad ; \text{by lemma 1} && (9)\\
c_{i+1} &\leq c_i(1-\frac{1}{k}) && (10)\\
c_{i+1} &\leq (1-\frac{1}{k})^i \cdot OPT \cdot (1-\frac{1}{k}) \quad ; \text{by inductive hypothesis} && (11)\\
c_{i+1} &\leq (1-\frac{1}{k})^{i+1} \cdot OPT && (12)
\end{aligned}
$$

□

Having proved the lemma 1 and 2, now we are ready to prove the approximation factor for greedy algorithm for maximum coverage problem.

Theorem 3. *Greedy approximation algorithm has $(1-\frac{1}{e})$ approximation factor.*

Proof. By lemma 2, we know that $c_k \leq (1-\frac{1}{k})^k \cdot OPT$. Since $(1-\frac{1}{k})^k \approx \frac{1}{e}$, we can say that $c_k \leq \frac{OPT}{e}$. Then,

$$
\begin{aligned}
b_k &= OPT - c_k \quad ; \text{by definition of } b_k && (13)\\
b_k &= OPT - \frac{OPT}{e} && (14)\\
b_k &= OPT(1-\frac{1}{e}) && (15)
\end{aligned}
$$

Since $b_k = \sum_{j=1}^{k} a_j$, i.e., the total number covered elements after the k^{th} iteration which is the output of the greedy algorithm, we can conclude that greedy approximation has $(1-\frac{1}{e})$ approximation factor. □

3 Traveling Salesman Problem (TSP)

In this section, we will cover the Traveling Salesman Problem (TSP). TSP is also an NP-hard problem. We will firstly give the problem definition. Then, we will cover an MST-based 2-approximation algorithm and its analysis for a specific type of TSP, namely metric TSP. Next, we will cover an $\frac{3}{2}$-approximation algorithm, namely Christofides algorithm, and its analysis for metric TSP.

3.1 Problem Definition

Given an undirected graph $G = (V,E)$ and a weight function over edges $w : E \to \mathbb{R}$, TSP tries to find the minimum weight cycle in the graph where each vertex is visited exactly once. In general, weight function can be any function. However, we will cover a specific version of TSP problem, namely metric TSP. In metric TSP, weight function satisfies the three metric criteria. Note that we represent an edge by a pair of vertices (v_i, v_j).

Definition 1. *Metric TSP is a TSP where weight function satisfies the following three metric criteria.*

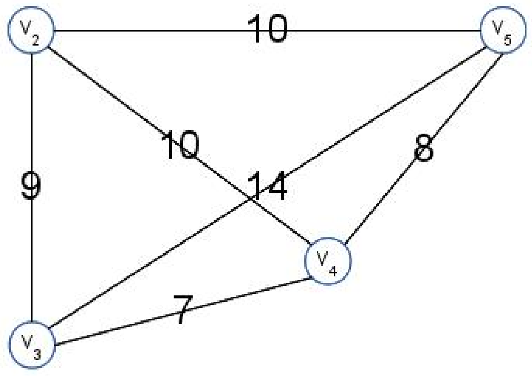

Figure 1: Example graph for metric TSP. Observe that weights satisfy the metric criteria.

1. $\forall i, j, w(v_i, v_j) = w(v_j, v_i)$
2. $\forall i, j, w(v_i, v_j) \geq 0$
3. $\forall i, j, k, w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$

One important observation here is that the graph has to be a *complete* graph in a metric TSP. Because, otherwise there will be at least one edge that does not satisfy the third criterion of being metric, which is the triangle equality. If we think an absent edge as an edge having infinite weight, it will always be greater than the sum of weights of any other two edges, which is violation of $\forall i, j, k, w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$ rule. Figure 1 shows an example graph for metric TSP.

3.2 MST-based Approximation Algorithm

In this section, we will show the first approximation algorithm for metric TSP problem. The algorithm is based on Minimum Spanning Trees (MSTs). Hence, let us remember MSTs.

Definition 2. *Given an undirected and connected graph, a spanning tree is a tree in the graph where each vertex is connected to each other. Since there can be different ways of connecting vertices to each other by trees, minimum spanning tree is the spanning tree whose weight is minimum.*

Note that weights are defined over the edges just as in metric TSPs. Figure 2 shows the minimum spanning of the graph shown in Figure 1 (the edges in bold composes the MST). Now, let us present the MST-based approximation algorithm for metric TSPs. Algorithm 2 shows the MST-based approximation algorithm for metric TSPs [2].

In the 2nd line of the Algorithm 2, we find the MST of the given undirected graph. In the 3rd line of the algorithm we apply depth-first search to the found MST. Moreover, while traversing the tree, we save each vertex in the order it is traversed. For the MST shown in Figure 2, for example, we will have sequence of $v_4, v_5, v_4, v_3, v_2, v_3, v_4$, if we start traversing from v_4. Observe that this sequence of vertices composes a cycle as shown by Figure 3.

However, in order to be a feasible solution for a TSP, a cycle has to visit each vertex exactly once. Hence, we eliminate vertices that has been visited more than once by shortcutting. Shortcutting means to remove a vertex that has already included in the cycle, and put an edge between the previous and next vertices of

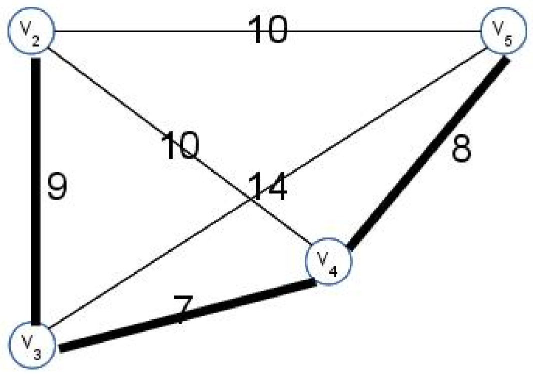

Figure 2: Minimum spanning tree for the graph in Figure 1

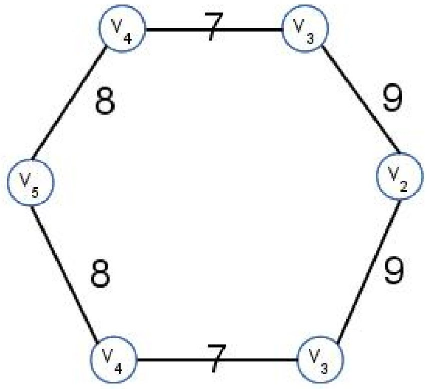

Figure 3: Cycle from the MST shown in Figure 2

Algorithm 2: MST-based approximation for metric TSP

```
1 Input: Undirected graph $G = (V,E)$, and weight function $w(\cdot)$
2 find the MST of the graph
3 apply depth-first search to the found MST until getting back to the root; save each vertex in the order
  it is traversed
4 the ordered edges of the traversed tree composes a cycle; eliminate vertices visited more than once by
  applying shortcutting
5 Output: the cycle of the ordered edges without duplicate vertices
```

the removed vertex in the cycle. Since the graph is complete there are edges between every pair of vertices. More importantly, since the weight function satisfies the triangle inequality (the third criterion in Definition 1), shortcutting will not increase the cost of the cycle. Figure 4 shows the shortcutting process for the cycle in Figure 3. The red vertices in the left-hand side of the figure are the shortcut vertices; and the red edges are the new edges introduced to eliminate those red vertices. Observe that the new cycle contains each element exactly once, and its weight is not larger (indeed less) than the weight of the cycle in the left-hand side. Hence, in the 4^{th} line of the algorithm, we apply shortcutting to the repeated vertices. Resulting cycle is a feasible solution for the TSP problem defined over the graph and weight function shown in Figure 1. Now, let us analyze the approximation factor for the given algorithm.

3.3 Analysis of MST-based Approximation

In this section, we will analyze the approximation factor for the MST-based approximation algorithm for metric TSP problem. Throughout the analysis, *OPT* denotes the optimal solution to the metric TSP; MST denotes the minimum spanning tree of the graph of the metric TSP.

Lemma 4. *Removal of an edge from the OPT will produce a spanning tree of the graph.*

Proof. OPT is a cycle visiting each vertex exactly once. Hence, when we remove an edge from the OPT, the remaining subgraph is a path visiting every vertex exactly once, which is a spanning tree. □

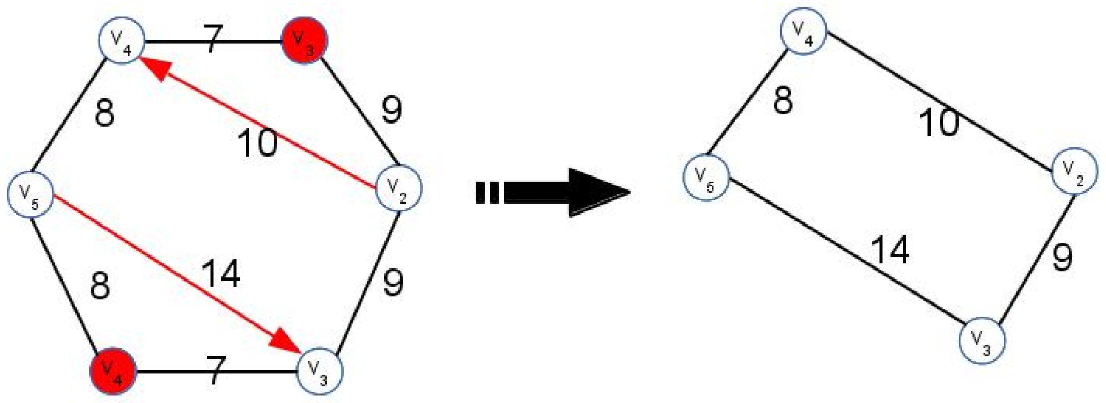

Figure 4: Shortcutting for the cycle in left-hand side. The shortcut vertices are v_4 and v_3.

Lemma 5. *MST of a graph is a lower bound for the OPT.*

Proof. By Lemma 4, we know that removal of an edge leaves the OPT as a spanning tree. Let us call this spanning tree as T. Since weight function on metric TSP is non-negative, we have:

$$T \leq OPT \tag{16}$$

Since MST is a lower bound for all possible spanning trees, we also have:

$$MST \leq T \tag{17}$$

Once we combine Equation 16 and 17, we can obtain:

$$MST \leq OPT \tag{18}$$

Hence, MST is indeed a lower bound on OPT. □

Lemma 6. *Algorithm 2 is a 2-approximation for metric TSP.*

Proof. Remember that, in Algorithm 2, we obtain the MST of the given graph, and traverse the edges of the MST in a depth-first manner to come up with a cycle. Observe that each edge is traversed exactly two times in the cycle obtained at the third line of Algorithm 2 for which an example is shown in Figure 4. Let us call this cycle as C_{inter}. Then, for the weight of the cycle, $|C_{inter}|$ we have:

$$|C_{inter}| \leq 2MST \tag{19}$$

By Lemma 5, specifically by Equation 18, we also have

$$2MST \leq 2OPT \tag{20}$$

By combining Equation 19 and 20, we get:

$$|C_{inter}| \leq 2OPT \tag{21}$$

Note that we apply one more step to the cycle to eliminate the repeated vertices by shortcutting at the 4th line of Algorithm 2. Let us call this latest cycle as C_{ALGO}. Thanks to triangle inequality satisfied by the weight function of metric TSP, shortcutting cannot increase the cost of the cycle (as exemplified in Figure 4). Hence, we have

$$\begin{aligned} |C_{ALGO}| &\leq |C_{inter}| && (22)\\ &\leq 2OPT && (23) \end{aligned}$$

Therefore, the MST-based approximation algorithm is indeed a 2-approximation for metric TSP problem. □

3.4 Christofides Algorithm

In this section, we will give the 2nd approximation algorithm, namely Christofides algorithm, for metric TSP problem by improving the approximation factor to $\frac{3}{2}$. Note that Christofides algorithm also makes use of MSTs, however, it also makes use of minimum weight matching in order to improve the approximation factor. Algorithm 3 presents the Christofides algorithm.

Algorithm 3: Christofides algorithm for metric TSP

```
1 Input: Undirected graph $G = (V, E)$, and weight function $w : E \to \mathbb{R}$
2 find an MST of $G$
3 take the subgraph $G' = (V', E')$ of $G$, where $V'$ is the vertices whose degree is an odd number in the
  found MST; $E'$ is the all the edges between vertices of $V'$
4 find a minimum weight perfect matching $M$ in $G'$
5 add the edges of $M$ to the found MST; that forms a graph, name it $H$
6 find a Eulerian tour on $H$, name it $T$
7 apply shortcutting to $T$ to obtain a feasible solution to the metric TSP, name it $F$
8 Output: $F$
```

Before getting to explain the algorithm, let us give the definitions of matching, minimum weight perfect matching and Eulerian tour.

Definition 3. *A matching of a given graph $G = (V, E)$ is a set of edges without common vertices.*

Definition 4. *A minimum weight perfect matching of a given graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$ is a matching including all vertices of the graph and having minimum weight.*

Definition 5. *A Eulerian tour (cycle) is a tour which visits every edge exactly once.*

One important fact about Eulerian tour is as following.

Fact 7. *A connected graph has a Eulerian tour if all of its vertices has even degree.*

Now let us explain Algorithm 3. We find an MST in line 2. Then, we take the subgraph G' of G where its vertices are the odd degree vertices of MST. Note that G' is a complete graph since we do not remove any edges from G, but only take the subgraph with odd degree vertices in the MST and all the edges between those vertices. The reason we find the subgraph with odd degree vertices is that we want to convert the MST to a graph in which all vertices has even degree so that we will be able to find a Eulerian tour leading to an approximate solution for metric TSP. In order to do this, in line 4, we find a minimum weight perfect matching M in the subgraph G'. In line 5, we add the edges of the found matching to the MST and convert the MST in to graph which we name as H. Note that, by definition 4, we know that M includes all the vertices of G'. Moreover, none of the edges in the matching share a vertex. Therefore, by adding the edges in matching M to the MST, we are able to make the odd degree vertices in the MST as even degree vertices. By definition 5 we know that we can obtain a Eulerian tour if all vertices are even degree. Hence, in line 6, we find the Eulerian tour T in H. Observe that T may include the same vertex more than once, whereas a feasible solution for TSP may not. Hence, in the last line of Algorithm 3 we apply shortcutting to T just as we have done in Algorithm 2, shown in Figure 4. The resulting tour is an approximation for the metric TSP problem. Let us analyze the approximation factor for Christofides algorithm.

3.5 Analysis of Christofides Algorithm

In this section, we will analyze the approximation factor of Christofides algorithm. Throughout the analysis, OPT denotes the optimal solution to the metric TSP; MST denotes the minimum spanning tree of the graph of the metric TSP.

Lemma 8. *Given a graph $G = (V,E)$, the sum of degrees in G is even.*

Proof. Since every edge contributes exactly 2 to the sum of the degrees, the sum of degrees is $2 \times |E|$, which is always even. □

Lemma 9. *Given any graph $G = (V,E)$, the number of odd degree vertices are even.*

Proof. Let s be the sum of degrees; o be the number of odd degree vertices; and e be the number of even degree vertices. Then,

$$s = \sum_{i=1}^{o} degree(v_i) + \sum_{j=1}^{e} degree(v_j) \tag{24}$$

By lemma 8 we know that sum of degrees is even. The sum of even degrees, $\sum_{j=1}^{e} degree(v_j)$, is also even. Hence, the sum of odd degrees, $\sum_{i=1}^{o} degree(v_i)$, should also be even. Since we sum odd degrees and obtain an even number, the number of odd degrees should also be even. □

Corollary 10. *The number of vertices in subgraph G' obtained in line 3 of algorithm 3 is even.*

Proof. The vertices of G' is the odd degree vertices of the MST found in line 2. Since MST is also a graph, the proof is immediate by lemma 9. □

Lemma 11. *The weight of optimal TSP solution S^* for subgraph G' is at most OPT, $w(S^*) \leq OPT$.*

Proof. Since triangle inequality holds on metric TSP problem, the cost of the optimal TSP solution, S^*, of subgraph, G', cannot be more than the optimal solution, OPT, of the original graph, G. Because we are not adding but only removing some vertices from original graph. Note that G' is a complete graph. Hence, there is always a feasible TSP solution in G'. □

Lemma 12. *The weight of the perfect matching found in line 4 of algorithm 3 is less than half of OPT, i.e., $w(M) \leq \frac{OPT}{2}$.*

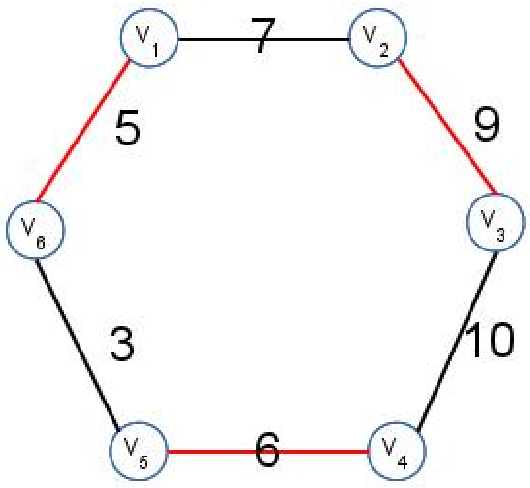

Figure 5: Alternating edges composes a perfect matching for the cycle.

Proof. By corollary 10, we know that the number of vertices of G' is even. Observe that the optimal TSP solution S^* for G' has to contain all the vertices since it is a TSP solution. Since the number of vertices are even, we can take each alternating sequences of edges in S^* as a matching. Figure 5 shows the alternating edges for an example TSP solution. The black edges and red edges compose two separate perfect matchings. Let B and R be the two matchings obtained from S^*. Then,

$$w(B) + w(R) = w(S^*) \tag{25}$$

Without loss of generality, we can assume $w(B) \leq w(R)$. Then,

$$w(B) \leq \frac{w(S^*)}{2} \tag{26}$$

Since know take the minimum weight perfect matching M, we also have

$$\begin{aligned} w(M) &\leq w(B) && (27)\\ &\leq \frac{w(S^*)}{2} && (28)\end{aligned}$$

By combining lemma 11 with equation 28, we can conclude that

$$\begin{aligned} w(S^*) &\leq OPT && (29)\\ \frac{w(S^*)}{2} &\leq \frac{OPT}{2} && (30)\\ w(M) &\leq \frac{w(S^*)}{2} \leq \frac{OPT}{2} && (31)\\ \Rightarrow w(M) &\leq \frac{OPT}{2} && (32)\end{aligned}$$

□

Lemma 13. *The total weight of the graph H composed of the MST and matching M is at most $\frac{3 \cdot OPT}{2}$, i.e., $w(M) + w(MST) \leq \frac{3 \cdot OPT}{2}$.*

Proof. By lemma 5, we know that weight of MST is a lower bound on OPT. Since we add edges of the matching M to the MST, by lemma 5 and 12 we have

$$\begin{aligned} w(M) &\leq \frac{OPT}{2} && (33)\\ w(MST) &\leq OPT && (34)\\ \Rightarrow w(M)+w(MST) &\leq \frac{3\cdot OPT}{2} && (35) \end{aligned}$$

$\square$

Lemma 14. *Approximation factor Christofides algorithm is $\frac{3}{2}$.*

Proof. Weight of the Eulerian tour T of the graph H cannot be larger than the total weight of the graph. Moreover, applying shortcut to T cannot increase the weight of T due to triangle equality of metric TSP. Hence, the output tour F of Christofides algorithm has:

$$\begin{aligned} w(F) &\leq w(T) && (36)\\ &\leq w(M)+w(MST) && (37)\\ &\leq \frac{3\cdot OPT}{2} && (38) \end{aligned}$$

Hence the approximation factor of Christofides algorithm is indeed $\frac{3}{2}$. The proof follows. $\square$

4 Strongly NP-hardness

In this section, we will cover strongly NP-hardness including polynomial-time approximation scheme (PTAS) and fully polynomial-time approximation scheme (FPTAS) terminologies.

Definition 6. *A problem is said to be strongly NP-hard if there is no pseudo-polynomial time algorithm for the problem.*

The fact that a problem is strongly NP-hard means that there can not be any efficient solution for solving the problem regardless of the numeric values of the input. Weakly NP-hard problems, on the other hand, may have polynomial time solutions if the numeric values of the input are relatively small. We will present knapsack problem as an example of a weakly NP-hard problem; and, bin packing problem as an example of strongly NP-hard problem. Before explaining knapsack and bin packing problems, let us introduce two approximation schemes, PTAS and FPTAS.

Definition 7. *Given an error parameter $\varepsilon > 0$, an algorithm is called as an approximation scheme if its output satisfies*

$$ALGO \leq (1+\varepsilon)OPT \qquad (39)$$

for minimization problems and

$$ALGO \leq (1-\varepsilon)OPT \qquad (40)$$

for maximization problems [3].

Definition 8. *An approximation scheme is a polynomial-time approximation scheme (PTAS) if, for a fixed error parameter $\varepsilon > 0$, its running time is polynomial in the size of the input.*

Definition 9. *An approximation scheme is a fully polynomial-time approximation scheme (FPTAS) if, for an error parameter $\varepsilon > 0$, its running time is polynomial both in the size of the input and $\frac{1}{\varepsilon}$.*

Although running time of PTAS algorithms are polynomial in the size of the input, say the size of the input is n, their running time can increase exponentially depending on the error parameter ε. For example, if the running time is $O(n^{(1/\varepsilon)^{(1/\varepsilon)}})$, as the error term reduces the running of the algorithm increases exponentially. FPTAS algorithms, on the other hand, alleviates this problem by requiring that the running of the algorithm is polynomial both in n and $1/\varepsilon$.

Corollary 15. *If there is an FPTAS algorithm for an NP-hard problem, there is also a pseudo-polynomial time algorithm for the problem.*

Proof. Assume we have an NP-hard problem, let us call it as Γ. Assume Γ has an FPTAS solution with error term $\varepsilon > 0$. Then by definition of FPTAS algorithm, the running time of the algorithm is $O(n \times \frac{1}{\varepsilon})$. If we set the error term as $\varepsilon = 1/I$ where I is polynomial in unary representation of the numeric values of the input, then the running time of the algorithm would be

$$O(n \times \frac{1}{\varepsilon}) \quad = \quad O(n \times I). \tag{41}$$

which is pseudo-polynomial [4]. □

Corollary 16. *There is no FPTAS algorithm for a strongly NP-hard problem.*

Proof. Assume there is an FPTAS for a strongly NP-hard problem. Then, by corollary 15, we know that there is also a pseudo-polynomial time algorithm for that problem. However, this contradicts with the definition of strongly NP-hard problem shown in Definition 6. The proof follows. □

4.1 Fractional Knapsack Problem

In this subsection, we will cover fractional knapsack problem. Note that there are two types of knapsack problem, fractional knapsack and 0-1 knapsack. 0-1 knapsack will be covered in Section 4.2. Firstly, we will give the definition of the fractional knapsack problem; then we will give a greedy algorithm; lastly, we will give analysis of the greedy algorithm concluding that the greedy algorithm produces the optimal result for fractional knapsack problem.

4.1.1 Problem Definition

Given a set of n items each of which has a particular weight w_i and value v_i, and a knapsack with maximum total weight capacity W; fractional knapsack problem selects a set of proportions from the items so that the total value is maximized and the total weight capacity W is not exceeded.

Example 1. Table 1 shows an example set of items with associated weights and values. Assume the capacity of the knapsack is $W = 150$. Then, fractional knapsack problem tries to find how much proportion of each item we should put into the knapsack in order to maximize the total value we get without exceeding the weight capacity of the knapsack, which is 150 for the particular example. Now, we will show how to optimally solve this problem with a greedy algorithm.

objects:	o_1	o_2	o_3	o_4	o_5
weights:	20	40	60	80	100
values:	40	50	60	110	130

Table 1: Example input for knapsack problem

4.1.2 Greedy Algorithm for Fractional Knapsack

Although there could be different greedy algorithms for fractional knapsack such as choosing always the lightest object until the knapsack is full, or choosing always the most valuable object until the knapsack is full; none of these two approach is able to produce the optimal solution for the fractional knapsack problem. The greedy algorithm we present, on the other hand, always chooses the object who has maximum value per weight until the knapsack is full. That is, we always choose the object who has maximum $\frac{v_i}{w_i}$ ratio value, $1 \leq i \leq n$ where n is the number of objects. Algorithm 4 shows the greedy algorithm for fractional knapsack problem. In Section 4.1.3, we will show that this algorithm indeed produces the optimal solution for fractional knapsack.

Algorithm 4: Greedy Algorithm for Fractional Knapsack

```
1  Input: Set of $n$ items with weights and values, a knapsack with capacity $W$.
2  $k = 0$
3  while $k \leq W$ do
4      l = $\text{argmax}_i \frac{v_i}{w_i}$
5      add the $l^{th}$ item to the knapsack and remove it from the list of candidate items
6      $k = k + w_l$
7  end
8  if $k > W$ then
9      take $\frac{w_{last}-(k-W)}{w_{last}}$ portion of the last item
10 end
11 Output: The set of items put into the knapsack
```

Table 2 shows the greedy solution for Example 1 where the knapsack capacity is 150. Observe that the order of the items with respect to the $\frac{v_i}{w_i}$ ratio value, $1 \leq i \leq n$, is as below:

$$\frac{v_3}{w_3} = \frac{60}{60} = 1.0 \geq \frac{v_2}{w_2} = \frac{40}{50} = 0.8 \geq \frac{v_5}{w_5} = \frac{100}{130} = 0.77 \geq \frac{v_4}{w_4} = \frac{80}{110} = 0.73 \geq \frac{v_1}{w_1} = \frac{20}{40} = 0.5$$

Since the maximum capacity of the knapsack we have is 150, we take object 3, 2, and half of the object 5 as shown in the last row of Table 2.

4.1.3 Analysis of Greedy Algorithm

In this section, we will analyze the greedy algorithm we presented for fractional knapsack.

Lemma 17. *The greedy algorithm shown in Algorithm 4.1.3 always produces optimal solution for fractional knapsack [5].*

objects:	o_1	o_2	o_3	o_4	o_5
weights:	20	40	60	80	100
values:	40	50	60	110	130
proportion:	0	1	1	0	$\frac{1}{2}$

Table 2: Solution for the example input for knapsack problem

Proof. Assume we are given n items with each associated a weight and value, and a knapsack with weight capacity W. Let x_i, for $1 \leq i \leq n$, be the indicator variable for the solution of the greedy algorithm. x_i simply shows the proportion of the object i that is put to the knapsack. The last row of Table 2, for example, corresponds to the values of the indicator variables for each item. Hence, the latest values of the indicator variables are the result of the greedy algorithm, indeed.

Let us reorder the objects with respect to their $\frac{v_i}{w_i}$ ratio values so that $X = (x_1, x_2, ..., x_n)$ where

$$\frac{v_1}{w_1} \geq \frac{v_1}{w_1} \geq ... \geq \frac{v_n}{w_n} \tag{42}$$

If all $x_i = 1$, $1 \leq i \leq n$, then the output is optimal. That means the capacity of the knapsack is large enough to put all the items into the knapsack. Otherwise, assume x_j is the first value that is less than 1. Remember that greedy algorithm selects objects with respect to the order shown in Equation 42. Hence, x_i's will be 1 up to some j value, implying that object $i < j$'s are selected, and be 0 after j, implying that object $i > j$'s are not selected. Object j may be selected proportionally. Then, let us assume there exist a feasible solution to the given knapsack problem $Y = (y_1, y_2, ..., y_n)$ where

$$\sum_{i=1}^{n} y_i v_i > \sum_{i=1}^{n} x_i v_i \tag{43}$$

Now, we need to find a contradiction with inequality 43. By feasibility of the solution Y, we know that

$$\sum_{i=1}^{n} y_i w_i \leq W \tag{44}$$

$$\sum_{i=1}^{n} y_i w_i \leq \sum_{i=1}^{n} x_i w_i \quad \text{; since } \sum_{i=1}^{n} x_i w_i = W \tag{45}$$

$$0 \leq \sum_{i=1}^{n} (x_i - y_i) w_i \tag{46}$$

We also know that

$$(x_i - y_i)\frac{v_i}{w_i} \geq (x_i - y_i)\frac{v_j}{w_j} \tag{47}$$

Because, for $1 \leq i < j$, we have $x_i = 1$. Hence $x_i - y_i \geq 0$. We also have $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ based on the order shown in Equation 42 for $1 \leq i < j$. On the other hand, for $j < i \leq n$, we know that $x_i - y_i \leq 0$. Moreover, $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$ also holds again based on the order shown in Equation 42. Then, if we use inequalities 46 and 47 in the differences between the total values of solution X and Y, we get:

$$\begin{aligned}
\sum_{i=1}^{n} x_i v_i - \sum_{i=1}^{n} y_i v_i &= \sum_{i=1}^{n} (x_i - y_i) v_i && (48)\\
&= \sum_{i=1}^{n} (x_i - y_i) \frac{v_i}{w_i} w_i && (49)\\
&\geq \sum_{i=1}^{n} (x_i - y_i) \frac{v_j}{w_j} w_i \quad \text{; by inequality 47} && (50)\\
&\geq \frac{v_j}{w_j} \sum_{i=1}^{n} (x_i - y_i) w_i && (51)\\
&\geq 0 \quad \text{; by inequality 46} && (52)
\end{aligned}$$

Inequality 52 contradicts with inequality 43. Hence, there cannot be any feasible solution to the fractional knapsack problem whose total value is more than total value of the solution of the greedy algorithm. The proof follows. □

4.2 0-1 Knapsack Problem

In Section 4.1, we covered fractional knapsack problem where items could be divided into some portions so that only a part of the items could be included to the knapsack. In this section, we will cover 0-1 knapsack problem, which is the one that is usually referred as knapsack problem. Firstly, we will give the definition of the problem. Next, we will present a dynamic programming algorithm for 0-1 knapsack problem followed by the analysis of the dynamic programming algorithm. Lastly, we will present an FPTAS for 0-1 knapsack problem based on the presented dynamic programming algorithm.

4.2.1 Problem Definition

Given a set of items n items each of which has a particular weight w_i and value v_i, and a knapsack with maximum total weight capacity is W; 0-1 knapsack problem selects a set of items so that the total value is maximized and the total weight capacity W is not exceeded. Note that, unlike the fractional knapsack, in 0-1 knapsack problem items cannot be divided into portions. That is, each item is either should be included as a whole or not.

4.2.2 Dynamic Programming Algorithm

It is known that 0-1 knapsack problem is NP-hard. In this section, we will present a pseudo-polynomial time dynamic programming (DP) algorithm for 0-1 knapsack problem. Let us name the DP table we will use as A. Then, each cell of the DP table A, i.e., A[i,p], will keep the least capacity for obtaining total value p with the objects $\{o_1, o_2, ..., o_i\}$ [6]. Hence, the rows of the table A will be $0, 1, ..., n$; whereas the columns of the table A will be $0, 1, 2, ..., P$ where $P = \sum_{i=1}^{n} v_i$. For each $p \in \{0, 1, ..., P\}$, we will try to find the least capacity for obtaining total value p by using only the items in $\{o_1, o_2, ..., o_i\}$. We will also assign ∞ to $A[i,p]$ if there is no such a subset of $\{o_1, o_2, ..., o_i\}$ whose total value is p. The base cases will be as following:

$$A[i,p] = \begin{cases} \infty, & \text{if } i == 0 \text{ and } p > 0 \\ 0, & \text{if } p == 0 \end{cases} \tag{53}$$

The reason we set $A[0,p]=\infty$ for $0<p<P$ is that, by using an empty set of objects, we cannot obtain any total value that is greater than 0. The reason we set $A[i,0]=\infty$ for $0\leq i\leq n$ is that, we cannot obtain value 0 with more than 0 objects. Then, at each step of the algorithm, for each $1\leq p\leq P$, we will use the following recurrence equation.

$$A[i+1,p]=\begin{cases}\min\{A[i,p],A[i,p-v_{i+1}]+w_{i+1}\} & \text{,if } v_{i+1}\leq p\\ A[i,p] & \text{,otherwise}\end{cases} \tag{54}$$

Let us explain the recurrence equation. At each step, what we think is whether to add the $(i+1)^{th}$ item to the knapsack to obtain a total value p. Hence, we apply the recurrence equation row by row for a fixed p value. That is, we assign values to $A[i+1,p]$ for $0\leq i\leq n-1$. Then, we update $p\leftarrow p+1$, and assign values to $A[i+1,p]$ for $0\leq i\leq n-1$ again. The reason we take $A[i,p]$ immediately if $v_{i+1}>p$ is that, if value of the $(i+1)^{th}$ item is greater than the p value, it is not possible to add the $(i+1)^{th}$ item to the knapsack for obtaining total value of p. The following equation

$$\min\{A[i,p],A[i,p-v_{i+1}]+w_{i+1}\}$$

actually determines whether to add the $(i+1)^{th}$ item to the knapsack or not. The $A[i,p-v_{i+1}]$ value corresponds to the least capacity for obtaining total value of $p-v_{i+1}$ with $\{o_1,o_2,...,o_i\}$ items. When we add the $(i+1)^{th}$ item, it will make the total value as $p-v_{i+1}+v_{i+1}=p$, and the required capacity as $A[i,p-v_{i+1}]+w_{i+1}$. If this capacity is greater than $A[i,p]$ which is the capacity of obtaining total value p with objects $\{o_1,o_2,...,o_i\}$, it means there is no need to add $(i+1)^{th}$ item to the knapsack. If $A[i,p-v_{i+1}]+w_{i+1}$ is less than $A[i,p]$, on the other hand, it means we should add the $(i+1)^{th}$ item to the knapsack. Note that ties are broken randomly.

Once we assigned all values in the DP table A, we need to find the row p who has some capacity value that is less than or equal to the total capacity of the knapsack W. p is the maximum total value we could obtain by putting the objects into the knapsack.

4.2.3 Analysis of Dynamic Programming Algorithm

The running time of the DP algorithm shown in Section 4.2.2 is $O(nP)$ where $P=\sum_{i=1}^{n}v_i$. Then,

$$\begin{aligned} O(nP) &= O(n\times\sum_{i=1}^{n}v_i) && (55)\\ &\leq O(n\times n\times v_{max}) && (56)\\ &= O(n^2\times v_{max}) && (57)\end{aligned}$$

where $v_{max}=\max_i v_i$. Hence, the shown DP algorithm is pseudo-polynomial time algorithm, which actually shows that 0-1 knapsack problem is not a strongly NP-hard problem.

4.2.4 FPTAS for 0-1 Knapsack Problem

In this section, we will show an FPTAS algorithm for 0-1 knapsack problem based on the DP algorithm shown in Section 4.2.2. Remember that the running time of an FPTAS should be polynomial in n and the error term $\frac{1}{\varepsilon}$. Moreover, it also has to $(1-\varepsilon)$ approximation factor since it is a maximization problem. What we do is as following. Given a 0-1 knapsack problem, we scale and round value of each item based on the following equation

$$v'_i = \lfloor \frac{v_i}{K} \rfloor, \quad 1 \leq i \leq n \tag{58}$$

, where

$$K = \frac{\varepsilon v_{max}}{n} \tag{59}$$

Then, we run the presented DP algorithm with these new v'_i values, and output the result. Algorithm 5 shows the algorithm.

Algorithm 5: FPTAS Algorithm for 0-1 Knapsack

```
1 Input: Set of $n$ items with weights and values, a knapsack with capacity $W$.
2 $K = \frac{\varepsilon v_{max}}{n}$
3 for $i = 1...n$ do
4     $v'_i = \lfloor \frac{v_i}{K} \rfloor$
5 end
6 run the presented DP algorithm on the scaled and rounded $v'_i$ values.
7 Output: The set of items returned by the DP algorithm.
```

Lemma 18. *The running time of Algorithm 5 is polynomial in the input size n and $\frac{1}{\varepsilon}$.*

Proof. The running time the DP algorithm on scaled and rounded values is $O(n^2 \times v'_{max})$. Since we scaled each value by $K = \frac{\varepsilon v_{max}}{n}$, we have

$$\begin{aligned}
O(n^2 \times v'_{max}) &= O(n^2 \times \lfloor \frac{v_{max}}{K} \rfloor) && (60)\\
&= O(n^2 \times \lfloor \frac{n v_{max}}{\varepsilon v_{max}} \rfloor) && (61)\\
&= O(n^2 \times \lfloor \frac{n}{\varepsilon} \rfloor) \quad ; v_{max}\text{'s cancel out} && (62)\\
&= O(n^2 \times \frac{n}{\varepsilon}) && (63)\\
&= O(\frac{n^3}{\varepsilon}) && (64)
\end{aligned}$$

The proof follows. □

Lemma 19. *Algorithm 5 has $(1-\varepsilon)$ approximation factor.*

Proof. Let I be the set of items returned by the DP algorithm run on the scaled and rounded item values; O be the set of items returned by the optimal solution. Then,

$$
\begin{aligned}
ALGO &= \sum_{o_i \in I}^{n} v_i && (65)\\
&\geq K\sum_{o_i \in I}^{n} \lfloor \frac{v_i}{K} \rfloor \quad \text{; since we floor} && (66)\\
&\geq K\sum_{o_i \in O}^{n} \lfloor \frac{v_i}{K} \rfloor \quad \text{; since } I \text{ is optimal for } \lfloor \frac{v_i}{K} \rfloor \text{ values} && (67)\\
&\geq (\sum_{o_i \in O}^{n} v_i) - |O|K \quad \text{; since we lost at most } K \text{ for each } o_i \in O && (68)\\
&\geq OPT - |O|K \quad \text{; since } OPT = \sum_{o_i \in O}^{n} v_i && (69)\\
&\geq OPT - |O|\frac{v_{max}\varepsilon}{n} \quad \text{; since } K = \frac{\varepsilon v_{max}}{n} && (70)\\
&\geq OPT - |O|\frac{v_{max}\varepsilon}{n}\frac{OPT}{OPT} && (71)\\
&\geq OPT - \frac{|O|}{n}\frac{v_{max}}{OPT}\varepsilon OPT && (72)\\
&\geq OPT - \varepsilon OPT \quad \text{; since } \frac{|O|}{n} \leq 1 \text{ and } \frac{v_{max}}{OPT} \leq 1 && (73)\\
&\geq (1-\varepsilon)OPT && (74)
\end{aligned}
$$

The proof follows. □

Lemma 20. *Algorithm 5 is an FPTAS for 0-1 knapsack problem.*

Proof. By lemma 18 and 19 the proof is immediate. □

Corollary 21. *0-1 knapsack is not a strongly NP-hard problem.*

Proof. The proof is immediate by lemma 20. □

4.3 Bin Packing Problem

In this subsection, we will cover bin packing problem, which is an example for strongly NP-hard problems. Firstly, we will give the definition of the bin packing problem; then we will give a greedy algorithm; lastly, we will give analysis of the greedy algorithm concluding that the greedy algorithm achieves approximation factor of 2.

4.3.1 Problem Definition

Given n items with sizes a_i, where $0 < a_i \leq 1$ for $1 \leq i \leq n$, bin packing problem aims to find the minimum number of bins of size 1 into which all n items can be packed.

4.3.2 Greedy First Fit Algorithm

In this section, we will cover the greedy first fit algorithm for bin packing problem. The idea is to process items in an arbitrary order; and put each item into the first bin that fits the item. Algorithm 6 shows greedy first fit algorithm.

Algorithm 6: Greedy First Fit Algorithm

```
1 Input: Set of $n$ items with sizes $a_i$ where $0 < a_i \leq 1$ for $1 \leq i \leq n$
2 for $i = 1...n$ do
3     put $i^{th}$ item into the first bin that the item fits
4     if the item does not fit any of the available bins; open a new bin, and put the item into that bin
5 end
6 Output: The number of opened bins
```

4.3.3 Analysis Greedy First Fit Algorithm

In this section, we will analyze the approximation factor of the greedy first fit algorithm shown in Section 4.3.2.

Lemma 22. *The number of bins found by the optimal algorithm is greater than the sum of the sizes of the values. That is,*

$$OPT \geq \sum_{i=1}^{n} a_i$$

Proof. This is because at best case, all items can be packed into unit sized packs without any overflow [7]. □

Lemma 23. *If there are ALGO bins used by first fit algorithm, then there are at least $ALGO - 1$ bins that are more than half-full.*

Proof. If there were more than one bin that is less than half-full, greedy first fit algorithm would immediately merge items in different bins into one bin. □

Lemma 24. *Greedy first fit algorithm has approximation factor of* 2.

Proof. By lemma 22 we know that

$$OPT \geq \sum_{i=1}^{n} a_i \tag{75}$$

Since there are at least $ALGO - 1$ bins that are more than half full by lemma 23, we also have

$$\sum_{i=1}^{n} a_i > \frac{ALGO - 1}{2} \tag{76}$$

Once we combine inequailities in 75 and 76, we get

$$\begin{aligned} OPT &\geq \sum_{i=1}^{n} a_i && (77)\\ OPT &> \frac{ALGO-1}{2} && (78)\\ 2OPT &> ALGO-1 && (79)\\ 2OPT &> ALGO && (80)\end{aligned}$$

The proof follows. □

5 Summary

In this lecture, we covered maximum coverage problem, and showed a $(1-\frac{1}{e})$-approximation greedy algorithm for maximum coverage problem. Then, we studied traveling salesman problem (TSP). We showed minimum spanning tree (MST)-based 2-approximation algorithm for TSP. We also showed Christofides algorithm for TSP whose approximation factor is $\frac{3}{2}$. Then, we studied strongly NP-hardness, PTAS, and FPTAS terminologies. We showed that there cannot be an FPTAS for a strongly NP-hard problem. Next, we studied fractional and 0-1 knapsack problems. For fractional knapsack problem, we showed an optimal greedy algorithm. For 0-1 knapsack problem, we showed an optimal dynamic programming algorithm whose running time is pseudo-polynomial. Based on the pseudo-polynomial time dynamic programming algorithm, we developed an FPTAS for 0-1 knapsack problem. Lastly, we studied bin packing problem, and finished by presenting a greedy 2-approximation algorithm for bin packing problem.

REFERENCES

[1] Lecture notes, CS 598CSC: Approximation Algorithms, Lecture 3 scribe notes, Spring 2011, Department of Computer Science, University of Illinois, Urbana-Champaign, http://goo.gl/GxqdUf.
[2] Lecture notes, COMP260: Advanced Algorithms, Spring 2011, Computer Science Department, Tufts University, http://goo.gl/DfYueN.
[3] Lecture notes, 550.770: Approximation Algorithms, Lecture 1 scribe notes, Fall 1998, Computer Science Department, Tufts University, http://goo.gl/1tsRSy.
[4] Lecture notes, 15-854: Approximation Algorithms, Lecture 10 scribe notes, Fall 2005, Computer Science Department, Carnegie Mellon University http://goo.gl/mEEaPz.
[5] Lecture notes, Advanced Algorithms, Computer Science Department, Universitat Politecnica de Catalunya, Barcelona, http://goo.gl/A2UoTo.
[6] Lecture notes, Approximation Algorithms and Hardness of Approximation, Lecture 3 scribe notes, Spring 2013, School of Computer and Communication Sciences, EPFL, http://goo.gl/OvciTO.
[7] Lecture notes, CS260: Approximation Algorithms, Chapter 9 slides, Fall 2006, Computer Science and Engineering Department, University of California, Riverside, http://goo.gl/vQ4mjd.