1 Introduction

In the Week 9 of this semester, Professor Goldberg and Sankaranarayanan talked about the concepts about NP-Completeness. Since our lecture will be based on these concepts, a brief review would expedite our understanding in approximation algorithm.

1.1 NP-Completeness Problems

An *optimization problem* is a computational problem looking for the best legal solutions. For example, what is the minimum spanning tree (MST) of a given graph G? A *decision problem* is a computational problem with a single boolean "yes" or "no" answer. For example, is 5 divisible by 3? Three complexity classes are defined based on *decision problems*:

- **P** is the set of problems that are polynomial-time solvable. Generally speaking, problems in **P** are "easy".
- **NP** is the set of problems that has the following properties: if the answer is yes, then this can be verified in polynomial time.
- **NP-Complete** Informally, a problem is called NP-complete when every other problem in NP can be transformed into this one in polynomial time[1]. Therefore, if one could find polynomial time solution to even one problem in NP-complete, then all the problems in NP-complete would have polynomial solutions. Generally speaking, NP-complete problems are "very hard". Several problems that we are going to cover in this lecture, like vertex cover problem, traveling-salesman problem are NP-complete.

Whether $\mathbf{P} = \mathbf{NP}$ is one of the most important problems in the theoretical computer science. Up to now, most scientists believe that they are actually different. NP-complete problems are restricted to the decision problems, but an optimization problem can usually be translated into a decision problem by introducing an upper/lower bound. For example, a decision version of the MST is : for a given graph G and number k, does there exist a spanning tree whose total cost is no more than k?

1.2 Approximation Algorithm

All NP-complete problems have no polynomial-time algorithm to find their optimal solutions up to now, and many theoretical computer scientists believe that there could be no polynomial-time algorithm at all. However, finding some *near optimal* solutions in polynomial-time algorithms is still possible. If these near optimal solutions are not very far away from the actual optimal solutions, these algorithms are still useful and worth investigating. Such algorithms are *approximation algorithms.*

The performance of approximation algorithm is described by *approximation ratio* $\rho(n)$. For a given input X, if the approximate solution $APP(X)$ is from the approximation algorithm, and the actual optimal solution $OPT(X)$ is from exact algorithm, the approximation

[1]There are more formal definitions of NP-complete and NP, please see Page 1060 and 1064 of [1]

ratio $\rho(n)$ satisfies

$$\rho(n) \geq \max\left(\frac{APP(X)}{OPT(X)}, \frac{OPT(X)}{APP(X)}\right) \tag{1}$$

for any input X with size n. An approximation algorithm with ratio $\rho(n)$ is called $\rho(n)$-approximation algorithm. The maximization of in the above equation is because we may deal with either a minimization or maximization problem when doing "optimization". For the minimization problem, $APP(X) \geq OPT(X)$, and therefore the second term in the max function is trivial. For the maximization problem, $APP(X) \leq OPT(X)$, and therefore the first term in the max function is trivial. A solution with $\rho = 1$ is exactly the optimal solution.

For some problems, like the vertex-cover problem, the approximation ratio $\rho(n)$ of the well-known algorithm is a simple positive constant. However, for some other problems, like the set-cover problem, $\rho(n)$ slowly increases with the input size n, like in $\log n$.

Approximation scheme is the approximation algorithm whose input includes not only X but also a positive ϵ, and whose output has $(1+\epsilon)$-approximation ratio. If for a positive ϵ, the running time of the approximation scheme is dependent on ϵ and polynomial in input size n, this scheme is called *PTAS (polynomial-time approximation scheme)*. If the running time is polynomial in n and $1/\epsilon$, this sheme is called *FPTAS (fully polynomial-time approximation scheme)*. For example, running time in $O(n^{1/\epsilon^2})$ is PTAS but not FPTAS, but running time in $O\left(n^3(\log n)/\epsilon\right)$ is both PTAS and FPTAS. Later in this lecture, we will show the approximation algorithm for subset-sum problem is an FPTAS.

Now we have finished introduction to the most important concepts in this lecture and will focus on individual approximation algorithm for several NP-complete problems. First, let's look at the vertex-cover problem.

2 Vertex Cover (VC) Problem

Vertex Cover: Given an undirected graph $G(V, E)$, a vertex cover V' is a subset of vertices V, such that for each edge (u, v) belongs to E, at least one of its endpoints, either vertices u or v, is in V'. An example of VC is shown in Fig. (1).

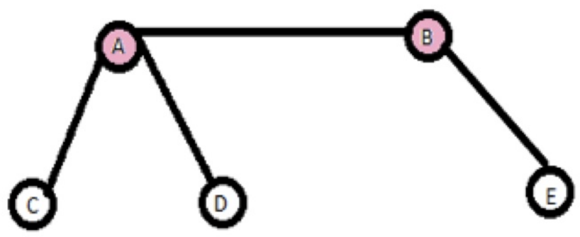

Figure 1: One example of vertex cover of this graph is $V' = \{A, B\}$, since all the edges of the graph has at least one vertices in V'.

The **optimization version** of the VC problem is to find the optimal vertex cover for the given undirected graph G.

Optimal vertex cover means smallest number of vertices in vertex cover for the given undirected graph. Finding the optimal Vertex cover is NP-Hard.

Consider the below shown graph:

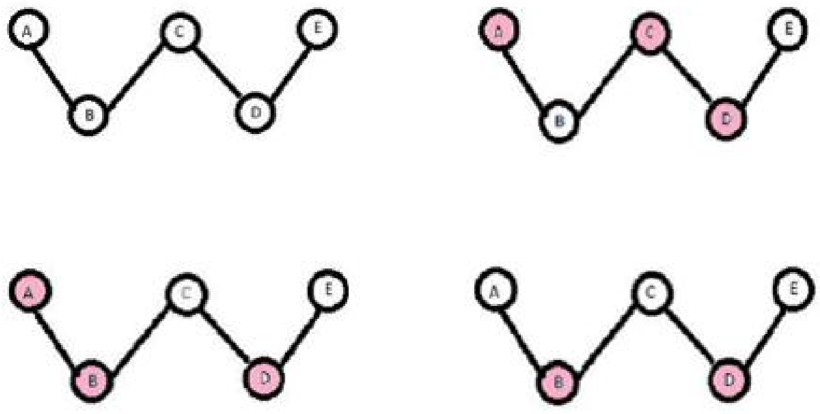

Figure 2: A graph (top left), and its three possible vertex covers (vertices are highlighted). The bottom right one is the optimal, since its only has only two vertices. The other two are NOT optimal.

The vertex cover problem is an NP-complete problem. The proof of this can be found from Page 1089 of [1].

Brute force method:

Given an undirected graph $G(V, E)$, to find the vertex cover of G, brute force method would involve checking all possible subsets of vertices V.
If there are n vertices, algorithm tries all 2^n subsets, which grows exponentially. To check whether a subset is a vertex cover it takes polynomial time.
Since the running time grows exponentially with respect to n, we can only use this method for very small values of n. For larger values of n, it takes extreme computation time and will be intractable.

2 - Approximation algorithm:

As we know that, finding an optimal vertex cover in polynomial time is hard, but we can find a near optimal vertex cover in polynomial time using approximation algorithm. Here is the algorithm:

Algorithm 2.1: APPROX−VERTEX−COVER(G)

```
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
    do { choose an arbitrary edge $(u, v)$ from $E'$
         $C = C \cup \{u, v\}$
         Remove from $E'$ all edges whose endpoints are either $u$ or $v$
return ($C$)
```

At each iteration of the while-loop, algorithm picks an edge (u, v), and adds its both endpoints u and v to the vertex cover set C, and deletes all the other edges whose endpoints are either u or v (because the vertices u and v also covers these edges).

Running time:
Assuming the graph is represented as adjacency list. The while-loop runs at most E times, and if we use adjacency list to represent E', the overall running time is $O(V + E)$, which is polynomial.

Algorithm 2.1 is an algorithm that runs in polynomial time and has approximation ration 2.

Let's prove that algorithm returns vertex cover that is at most twice the size of an optimal cover (the cover with smallest number of vertices).

Let,
OPT be the optimal vertex cover,
and C be the vertex cover calculated by Algorithm 2.1.

We want to prove that algorithm returns vertex cover C such that

$$|C| <= 2\,|OPT|\,, \tag{2}$$

where the $|C|$ means the number of elements in set C.

And at each iteration of the while-loop, endpoints of an arbitrary edge (u, v) will be added to vertex cover set C, and all the other edges incident either at u or at v will be removed. Hence no two edges selected at line 4 are incident on the same vertex.

Let,
$ALLE$ be set of all edges picked at line 4.

Since no two edges in $ALLE$ share a vertex, thus the optimal cover OPT must include at least one endpoint of the edges in $ALLE$ (to cover all the edges in $ALLE$). Hence the lower bound of OPT is

$$|OPT| \geq |ALLE|. \tag{3}$$

And, as we can see inside the while-loop, when an edge is picked at Line 4, both of its endpoints will be added to the vertex cover set C, thus the size of the set C is twice the size of edges in $ALLE$. Therefore,

$$|C| = 2\,|ALLE|. \tag{4}$$

By combining above two relations, we have

$$|C| = 2\,|ALLE| \leq 2\,|OPT|. \tag{5}$$

Hence we have proved that algorithm returns vertex cover that is at most twice the size of an optimal cover.

Example of Algorithm 2.1: Consider the below graph $G = (V, E)$ and let's apply Algorithm 2.1 on this graph (from Fig. (3) to Fig. (7)).

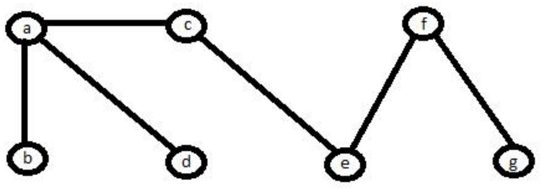

Figure 3: The origin graph. Now $C = \emptyset$, and $E' = \{(a, b), (a, d), (a, c), (c, e), (e, f), (f, g)\}$.

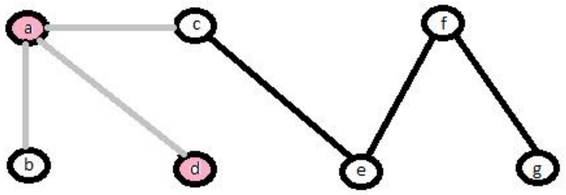

Figure 4: First iteration of the while-loop: assume edge (a, d) is picked (at line 4) and both the vertices a and d are added to vertex cover set C. Next, all the other edges which are incident at either a or d are removed (at line 6 of Algorithm 2.1). So edges (a, b) and (a, c) are also removed from E' since they are covered by vertex a. So far, vertex cover $C = \{a, d\}$, edges covered $= \{(a, b), (a, d), (a, c)\}$ (lightened edges in the above graph), and $E' = \{(c, e), (e, f), (f, g)\}$.

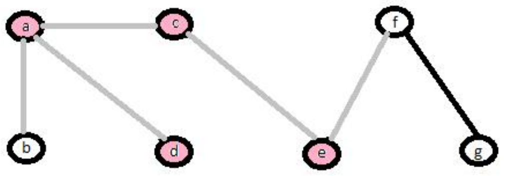

Figure 5: Second iteration of the while-loop: assume edge (c, e) is picked (at line 4) and both the vertices c and e are added to vertex cover set C. Next, edge (e, f) will be removed at Line 6 as it covered by vertex e. After second iteration, vertex cover $C = \{a, d, c, e\}$, edges covered $= \{(a, b), (a, d), (a, c), (c, e), (e, f)\}$ (lightened edges in the above graph), and $E' = \{(f, g)\}$.

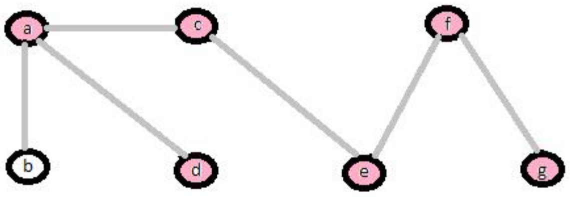

Figure 6: Third iteration of the while-loop: the only remaining edge (f, g) is picked (at line 4) and both the vertices f and g are added to vertex cover set C. After the third iteration, vertex cover $C = \{a, d, c, e, f, g\}$, edges covered $= \{(a, b), (a, d), (a, c), (c, e), (e, f), (f, g)\}$ (lightened edges in the above graph), and $E' = \emptyset$. $|C| = 6$. The algorithm terminates.

3 Traveling Salesman Problem (TSP)

TSP is the following problem: for a given a complete undirected graph $G(V, E)$, each edge (u, v) is assigned a non-negative weight $w(u, v)$, and we are looking for such a circle that contains each vertex in V exactly once and minimizes the total weight of all the edges included.

The most direct and straightforward algorithm would be try all the permutations of the vertices and see which one has the smallest weight. This is an exact algorithm. However, the running time of this brute force search will be $O(n!)$. Even if we have 50 vertices, this may have 10^{64} possible permutations and it may take 10^{46} years (much longer than age of the

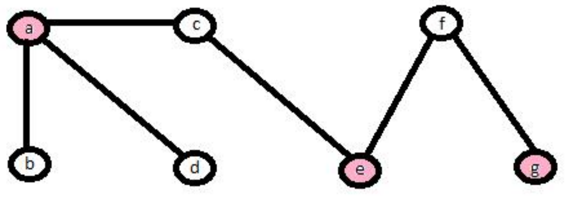

Figure 7: Optimal vertex cover for Fig. 3: its optimal size of vertex cover is 3, while size of the vertex cover C produced by Algorithm 2.1 is 6. Here $|C| \,/\, |OPT| = 2$.

universe) if the computer can calculate the weights of 4×10^{10} possible routines per second. This is not an acceptable algorithm at all. If one use *dynamic programming*, Held-Karp algorithm could exactly solve TSP in $O(n^2 2^n)$ [2]. For 50 vertices, this can be solved in about 2 years by such a computer, which may be somehow acceptable, but is still too long!

3.1 Naive Approximation Algorithm

A common special case in the TSP is that all the edges satisfy the *triangle inequality*:

$$w(u, v) \leq w(u, x) + w(x, v) \text{ for any vertices } u,\ v, \text{ and } x \in V. \tag{6}$$

This is usually true for most weights in real life, like the distance in Euclidean spaces. Exact TSP is still an NP-complete problem even with the triangle inequality imposed. However, assuming the triangle inequality, people developed the $\rho = 2$-approximation algorithm which runs in polynomial time. Here is the algorithm:

Algorithm 3.1: APPROX−TSP($G(V, E)$)

```
select an arbitrary vertex $r \in V$.
MST = PRIM ($G$, $r$)
PREORDER(MST, $r$), record the order of vertices
construct a circle $H$ of vertices in the order they are visited above, going back to $r$.
return ($H$)
```

Prim(G, r) is to construct a Minimum Spanning Tree (MST) for graph G with root vertex r by Prim algorithm. The details of Prim algorithm can be found [3], Aaron's lecture notes 8, or Page 634 of [1]. The PREORDER (MST,r) is the preorder tree walk for MST, starting at r, which visits the root vertex first and then its any children. A diagram of the above algorithm is shown below, from [1]. Both Prim algorithm ($O(V^2)$ for trivial implementation or $O(E+V \log V)$ for Fibonacci heap and adjacency list) and preorderly walk of MST ($O(V)$) are in polynomial-time, so the time complexity of Algorithm 3.1 is in polynomial.

Algorithm 3.1 has approximation ratio of 2. Here is the proof. Here OPT denotes the optimal solution to TSP, H is the total weight of circle from the above algorithm, MST denotes the total weight of the MST, and a TOUR is the list of vertices that are encountered in preorder tree walk, no matter how many times we have visited them, and $TOUR$ denotes the weight of TOUR. For example, in Fig. (8.c), a TOUR is $\{a,\ b,\ c,\ b,\ h,\ b,\ a,\ d,\ e,\ f,\ e,\ g,\ e,\ d,\ a\}$. H is taking shortcut on TOUR by skipping visited vertices.

First, removing an edge in OPT gives rise to a spanning tree and MST has the minimum weight among all spanning trees. Therefore,

$$MST \leq OPT. \tag{7}$$

The TOUR has exactly covered each edge in MST twice, which indicates that

$$TOUR = 2 \times MST. \tag{8}$$

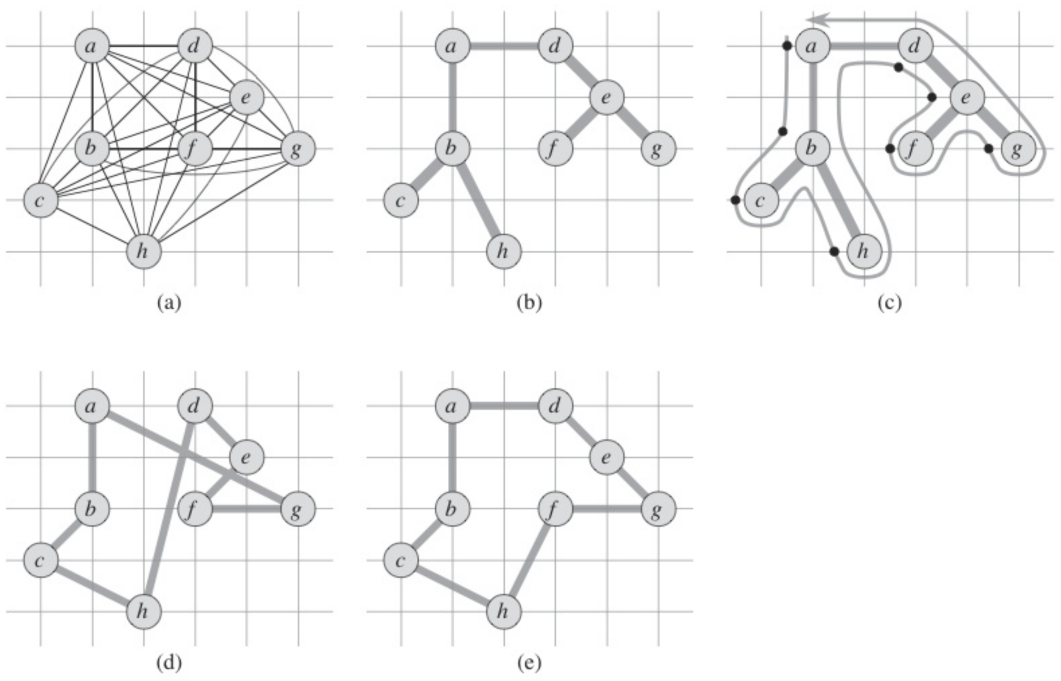

Figure 8: (a) The origin complete graph. (b) MST from the Prim algorithm, with root a. (c) A preorder walk of MST. (d) The solution by approximation algorithm, with total weight of 19.074. (e) The actual optimal solution, with total weight of 14.715. This whole figure is from Page 1113 of [1].

On the other hand, because of the triangle inequality, taking short cuts would not exceed the tour, thus

$$H \leq TOUR. \tag{9}$$

Combining the three relationships above, we obtain

$$H \leq TOUR = 2 \times MST \leq 2 \times OPT \tag{10}$$

We have proved that Algorithm 3.1 has approximation ratio 2.

3.2 Christofides Algorithm

Algorithm 3.1 is a good algorithm that runs in polynomial time. Christofides algorithm is another polynomial-time algorithm which requires the triangle inequality for TSP, and has even better approximation ratio (1.5) [4]. Here is the algorithm:

Algorithm 3.2: CHRISTOFIDES$-$TSP$(G(V, E))$

```
construct MST for $G$
construct a vertex set $O$, which contains all the *odd degree* vertices in MST
find a *perfect matching* $M$ with minimum weight for $O$
combine $M$ and MST into a new graph $G'$
form *Eulerian circuit* $E$ in $G'$
Convert $E$ into $H$ by shortcutting nodes that have already been visited.
**return** $(H)$
```

Firstly we need to explain some concepts here. Degree of a vertex is the number of edges that connect to this vertex. For example, degree of vertex b in Fig. (8.b) is 3, and degree of vertex d is 2. Perfect matching is to find the set M, consisting of edges whose both endpoints are in O and all vertices in O shows up exactly in one edge of M. The number of elements in O must be an even integer, since every edge has two endpoints. Eulerian circuit is a trail that starts at one vertex, ends on the same vertex, and visits every edge exactly once. For the graph G', in which every vertex has even degree, therefore the Eulerian circuit is achievable[2]. An example of Algorithm 3.2 is in Fig. (9).

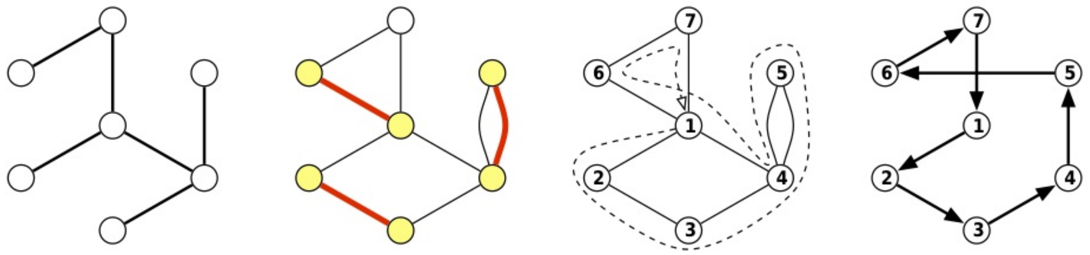

Figure 9: The four panels are: MST, odd degree vertices and their minimum perfect matching, Eulerian circuit, and approximation solution to TSP, respectively. credit from [5]

Each step in the Algorithm 3.2 is in polynomial time: MST can be constructed in $O(E \log V)$ by quick sort in Kruskals algorithm; perfect matching with minimum weight can be implemented by Edmond's algorithm, which runs in $O(V^2E) = O(V^4)$ for straightforward implementation and the best known algorithm in $O(V(E + \log V))$ [6]; and forming an Eulerian circuit can be computed in $O(E)$, by Fleury's algorithm. We can see that Christofides algorithm is dominated by finding the perfect matching with minimum weight. The details of Edmond's algorithm is beyond the scope of this topic. A very good review about it can be found at [7].

Now we are going to prove that the approximation ratio of Christofides algorithm is 1.5. OPT, MST, and H denote the same thing as when we prove the ratio of Algorithm 3.1. $OPTO$ denotes the optimal solution to TSP for the graph with vertex set O. By triangle inequality,

$$OPTO \leq OPT, \tag{11}$$

[2]see http://en.wikipedia.org/wiki/Euler_tour

which simply means that visiting fewer vertices has less (at most the same) total weight. On the other hand, $OPTO$ can seen as a combination of two perfect matching, and the weight of the smaller one is therefore no more than $OPTO/2$. Thus, the total weight of the minimum perfect matching M follows

$$M \leq OPTO/2 \leq OPT/2. \tag{12}$$

Finally, the same as above, total weight of H, which is generated by taking shortcut in Eulerian circuit $E = (MST + M)$, cannot exceed E,

$$H \leq MST + M \leq OPT + OPT/2 = 1.5 \times OPT. \tag{13}$$

Eq. (7) was used here. Now I have proved that Christofides algorithm is a 1.5-approximation of the algorithm.

If the weight of each edge can only take two possible values 1 or 2, then TSP can be solved with a ratio $\rho = 7/6$ in polynomial time, by Papadimitriou and Yannakakis [8].

Last, it has been shown in Page 1115 of ([1]) that if the triangle inequality does not apply to the graph $G(V, E)$, there is no approximation algorithm for TSP which runs in polynomial time unless $\mathbf{P} = \mathbf{NP}$.

4 Subset-sum Problem

The optimization version of the subset-sum problem is the following: given a set $S = S[1], S[2], \cdots, S[n]$, where all $S[i]$ are positive integers, and positive integer t, we want to find the subset that maximizes its sum but no more than t. This problem has direct application in reality. For example, there is a cargo ship with capacity of t, and we need it to carry the n different freight containers, with weight $S[1], S[2], \cdots, S[n]$, what is the maximum weight we could carry?

The known exact solution to this problem is the following exponential algorithm.

Algorithm 4.1: SUBSET−SUM(S, t)

```
$L_0 = \{0\}$
for $i = 1$ to $n$
    do { $L_i = L_{i-1} \cup (L_{i-1} + S[i])$
       { remove elements in $L_i$ that are larger than t
return $(\max\{L_n\})$
```

$L_{i-1} + S[i]$ means that all the elements in L_{i-1} add $S[i]$ respectively. $\cup$ means the union of two sets. The length of list L_n would be up to 2^n and hence the total number of operations would be $O(2^n)$. Therefore, this is an exponential algorithm at the worst case. The worst case corresponds to that there is no duplicate number when L_{i-1} and $(L_{i-1}+S[i])$ are merged at all times. If when they are merged, the length of the new list L_i is less than 2^i (due to duplicate number in L_{i-1} and $(L_{i-1}+S[i])$), then this algorithm would run faster. This would the main point to introduce the approximation algorithm. After L_{i-1} and $(L_{i-1} + S[i])$ are merged into a sorted list L_i, if two or more elements in L_i are "close" enough, only one of them is kept. The shorter L_i, the faster the algorithm is. TRIM algorithm implements the extraction of "extra" elements in L_i, from Page 1130 of [1].

Algorithm 4.2: TRIM(L, δ)

```
$L' = \{L[1]\}$
$now = L[1]$
for $i = 2$ to $n$
    do if $L[i] > now \times (1 + \delta)$
        then append $L[i]$ to $L'$
             $now = L[i]$
return $(L')$
```

Now, based on the two Algorithms 4.1 and 4.2, the approximation algorithm is

Algorithm 4.3: Approx−Subset−Sum(S, t, ϵ)

```
$A_0 = \{0\}$
sort $S$
for $i = 1$ to $n$
    do $A_i = A_{i-1} \cup (A_{i-1} + S[i])$
       $A_i = \mathrm{TRIM}(A_i, \epsilon/(2n))$
       remove elements in $A_i$ that are larger than t
return $(\max\{A_n\})$
```

Let's look at an example, here my $S = \{100, 105, 108, 200\}$ and the maximum capacity $t = 315$, and $\epsilon = 0.8$ such that $\delta = 0.1$ in Algorithm 4.2.

When we do the for-loop for the first time:

$$\begin{aligned} A_1 &= \{0, 100\} \text{ after the combination} \\ A_1 &= \{0, 100\} \text{ after trim} \\ A_1 &= \{0, 100\} \text{ after removing elements greater than } t = 315 \end{aligned}$$

When we do the for-loop for the second time:

$$\begin{aligned} A_1 &= \{0, 100, 105, 205\} \text{ after union} \\ A_1 &= \{0, 100, 205\} \text{ after trim} \\ A_1 &= \{0, 100, 205\} \text{ after removing elements greater than } t = 315 \end{aligned}$$

When we do the for-loop for the third time:

$$\begin{aligned} A_1 &= \{0, 100, 108, 205, 208, 313\} \text{ after union} \\ A_1 &= \{0, 100, 205, 313\} \text{ after trim} \\ A_1 &= \{0, 100, 205, 313\} \text{ after removing elements greater than } t = 315 \end{aligned}$$

When we do the for-loop for the fourth time:

$$\begin{aligned} A_1 &= \{0, 100, 200, 205, 300, 313, 405, 513\} \text{ after unio} \\ A_1 &= \{0, 100, 200, 300, 405, 513\} \text{ after trim} \\ A_1 &= \{0, 100, 200, 300\} \text{ after removing elements greater than } t = 315 \end{aligned}$$

Here our solution from the approximation algorithm is 300, while the exact solution is $100 + 105 + 108 = 313$. It is about 4% off, within $\epsilon = 0.8$. Let's do some analysis on this algorithm.

Firstly, we will show that Algorithm 4.3 would give $(1 + \epsilon)$-approximation algorithm if $0 < \epsilon \leq 1$. By induction (see Page 11 of [5]), we can show that for any number in $l \in L_i$ (in Algorithm 4.1), there is an element $a \in A_i$ (in Algorithm 4.3) such that

$$a \leq l \leq a\left(1 + \frac{\epsilon}{2n}\right)^i. \tag{14}$$

For $(\max\{L_n\})$ in Algorithm 4.1, there must be an element $a' \in A_n$, such that $\max\{L_n\} \leq a'\left(1 + \epsilon/2n\right)^n$. Moreover,

$$\max\{L_n\} \leq a'\left(1 + \epsilon/2n\right)^n \leq a' e^{\epsilon/2} \leq a'(1 + \epsilon) \leq \max\{A_n\}(1 + \epsilon). \tag{15}$$

The second $\leq$ in the above equation is because the function $f(n) = (1 + 1/n)^n$ is monotonically increasing, and $\lim_{n\to\infty} f(n) = e$. The third $\leq$ is due to Taylor expansion, $e^x \leq 1 + x + x^2/2 \leq 1 + 2x$ for $0 < x \leq 1$. The last $\leq$ is due to $a' \in A_n$ so that $a' \leq \max\{A_n\}$. Eq. (15) is exactly the definition of $(1 + \epsilon)$-approximation for a maximization problem.

Secondly, we show that time complexity of Algorithm 4.3 is $O(n^3 \log n/\epsilon)$, which is an FPTAS.

In the Algorithm 4.3, at the ith iteration of the for-loop, combine the sorted list L_{i-1} and $(L_{i-1} + S[i])$ would take $O(\text{length}(L_{i-1}))$. TRIM also takes $O(\text{length}(L_{i-1}))$. Therefore, each step at for-loop would take $O(\text{length}(L_{i-1}))$ and the total time complexity is $O\left(\sum_i \text{length}(L_i)\right)$.

All the elements in $(L_{i-1} + S[i])$ are at least $S[i]$ and at most $i \times S[i]$, since S is a sorted list. During trimming, if two elements survive, then they must differ by a factor of at least $(1 + \epsilon/2n)$. Therefore, the number of surviving elements in $(L_{i-1} + S[i])$ is $[\log_{1+\epsilon/2n}(i \times S[i])] - [\log_{1+\epsilon/2n}(S[i])] = [\log_{1+\epsilon/2n} i]$. Now

$$\begin{aligned} \text{length}(L_i) &\leq \text{length}(L_{i-1}) + [\log_{1+\epsilon/2n} i] \leq \text{length}(L_{i-1}) + \left[\frac{\log n}{\log(1 + \epsilon/2n)}\right] \\ &\leq \text{length}(L_{i-1}) + \left[\frac{4n \log n}{\epsilon}\right] \leq \text{length}(L_{i-2}) + \left[\frac{8n \log n}{\epsilon}\right] \leq \left[\frac{4n^2 \log n}{\epsilon}\right] = O(n^2(\log n)/\epsilon) \end{aligned} \tag{16}$$

Hence, the total time complexity is $O\left(\sum_i \text{length}(L_i)\right) = O(n^3(\log n)/\epsilon)$. For the first $\leq$ in line 2, we used $e^x \leq 1 + 2x$ again.

The proofs about approximation ratio and time complexity follow the ideas from [5].

5 Difficulty in Approximation Algorithm

A natural question is what the best approximation ratio ρ we can get in polynomial time is. For some optimization problems, finding an approximation algorithm with ρ smaller than a certain value ρ_c is also almost impossible. With the help of PCP theorem, one of the cutting-edge topic in approximation algorithm nowadays is the hardness of approximation, i.e. to find the best ρ_c with which an approximation algorithm can achieve in polynomial time and to show that algorithm with $\rho < \rho_c$ is not achievable. For example, for the MAX-3SAT problem, Johan Hastad [9] have proved that there is no polynomial-time algorithm with approximation ratio smaller than 8/7 unless **P=NP**. Furthermore, in the 2000s, computer scientists have proposed *unique games conjecture* (UGC). Under UGC, the best approximation ratios for more problems have been estimated or reestimated. For example, the vertex cover problem don't have any approximation algorithm with $\rho < 2$ under UGC [10].

References

[1] Thomas H. Cormann et al. *Introduction to Algorithm.* MIT Press, 2009.

[2] Michael Held and Richard M. Karp. A dynamic programming approach to sequencing problems. *J. Soc. Indust. and Appl. Math.*, 10(1):196–210, 1962.

[3] R. C. Prim. Blossom v: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1:43–67, 2009.

[4] N. Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem.* CMU, 1976.

[5] Jeff Erickson. Lecture 30: Approximation algorithm. Lecture Notes, 2010. Available from: `http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/notes/30-approx.pdf`.

[6] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–444, 1990.

[7] V. Kolmogorov. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[8] C. H. Papadimitriou and M. Yannakakis. The traveling salesman problem with distances one and two. *Mathematics of Operations Research*, pages 1–11, 1993.

[9] Hastad J. Some optimal inapproximability results. *Journal of the ACM*, 48:798859, 2001.

[10] S. Khot and O. Regev. ertex cover might be hard to approximate to within 2-ϵ. *Journal of Computer and System Sciences*, 74:335–349, 2008.