1 Introduction

1.1 Motivation

Many problems of practical computation problems are NP-hard, which means that no polynomial-time algorithm exists that solves the problem optimally unless P=NP. There are the three approaches to get around the NP-complete problem:

First, if the actual inputs are small, the exact algorithm with exponential time may be perfectly satisfactory.

Secondly, heuristics methods may produce good solutions but they do not come with a guarantee on the quality of their solution. (e.g. Simulated Annealing, Genetic Algorithm).

Thirdly, we might come up with approaches to find the near-optimal solutions in polynomial time. It turns out that we can often design these algorithms in such a way that the quality of the output is guaranteed to be within a constant factor of an optimal solution. This is the approach that we will investigate throughout the lecture.

1.2 Performance ratios for approximation algorithms

Suppose we are working on an optimization problem P. P aims to find a Min/ Max subject to certain constraints.

Let $C*$ denote the global optimal solution to P. And C denote solution our algorithm produced for P. We say that our algorithm has an approximation ratio of $\rho(n)$ if, for any input of size n, C is within a factor of $\rho(n)$ of the cost C*:

$$\rho(n) \geqslant max(\frac{C*}{C}, \frac{C}{C*})$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algorithm. For a maximization problem, $0 \leqslant C \leqslant C*$, and the ratio $\frac{C*}{C}$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution. Similarly, for a minimization problem, $0 \leqslant C* \leqslant C$, and the ratio $\frac{C}{C*}$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. The approximation ratio of an approximation algorithm is never less than 1. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

Different problems exhibit different approximation factors. There is a set of problems have the property that given any positive constant ϵ, we can give a $(1 + \epsilon)$-approximation algorithm for it. Although the running time of the approximation algorithm is efficient for fixed ϵ, it may get much worse as ϵ becomes small, for example if the running time is $n^{\frac{1}{\epsilon}}$. In this case we are trading

running time for quality of solutions. Problems with this property are said to have a polynomial time approximation scheme (PTAS).

$(1 + \epsilon)$-approximation scheme is one of the best scenarios for approximation algorithms. They have constant approximation ratio $\rho(n)$ for all input size n. In other cases, we have approximation algorithm of $\rho(n) = logn$ ratio or even worse, say $\rho = n$. We can then categorize optimization problems in terms of how well they can be approximated by efficient algorithms.

Approximation ratio $\rho(n)$	Approximability
$1+\epsilon$ (PTAS)	Very approximable
1.1	
2	
constant c	Somewhat approximable
$logn$	
$log^2 n$	
$\sqrt{n}$	
n^c, constant c	
n	Not very approximable

Now we know the problem with PTAS algorithms, that the exponent of the polynomial running time function could increase dramatically as ϵ shrinks. Is there any better algorithms that the running time would not grow exponentially as ϵ decrease? Fully polynomial-time approximation scheme (FPTAS) satisfy this requirement. FPTAS algorithms are polynomial in both the problem size n and $\frac{1}{\epsilon}$. This lecture will give a FPTAS algorithm for subset sum problem.

2 Vertex Cover

The vertex cover problem is defined as follows:

Definition Given a graph $G = (V, E)$ find a subset $C \subseteq V$ such that for all $(u, v) \in E$, at least one of u or v is included in C and the cardinality of set C is minimized.

This problem is to find a vertex cover of minimum size in a given undirected graph. It is an NP-complete problem. Currently there is no solution to find the optimal vertex cover in G in polynomial time, approximation algorithm can efficiently find a vertex cover that is near optimal.

The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

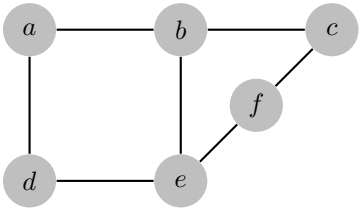
Figure 1. illustrates how APPROX-VERTEX-COVER operates on an example input graph G. First

Algorithm APPROX-VERTEX-COVER(G)

1: $C \leftarrow \emptyset$
2: $E' \leftarrow G.E$
3: **while** $E' \neq \emptyset$ **do**
4: let (u, v) be an arbitrary edge of E'
5: $C = C \cup \{u, v\}$
6: remove from E' every edge incident on either u or v
7: **end while**
8: **return** C

initialize C with an empty set, E' with a copy of the edge set $G.E$ of G. E' is represented by adjacency list. The while loop repeatedly picks an arbitrary edge (u, v) from E', adds its endpoints u and v to C, and deletes all edges in E' that are covered by either u or v. Finally, it returns the vertex cover C.

Figure.1 (a) is the input graph G, $|V| = 6$, $|E| = 7$. In Figure.1 (b), the edge (a, b), marked in red, is the first edge chosen by APPROX-VERTEX-COVER. Vertices a and b, shown in red, are added to the set C. Edges incident to a or b: (a, b), (a, d), (b, c), (b, e), shown in dash line, are removed from E'. In Figure.1 (c), the edge (e, f), marked in red, is chosen, Vertices e and f, shown in red, are added to the set C. Edges incident to e or f: (e, f), (d, e), (f, c), shown in dash line, are removed from E'. E' is now empty, APPROX-VERTEX-COVER halts. The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the four vertices a, b, e and f. The OPTIMAL solution contains only 3 vertices b, d and f.



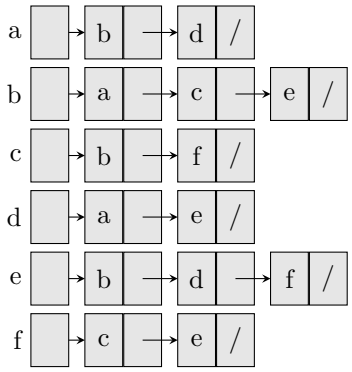


Figure.1(a). G (left) and E' (right). $C = \emptyset$.

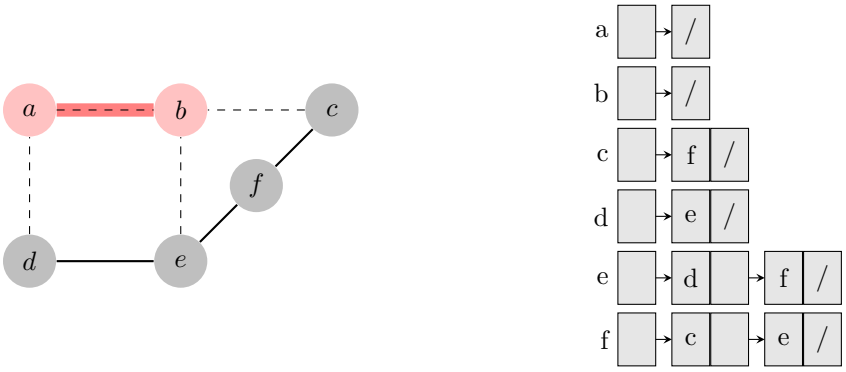


Figure.1(b). Pick edge (a, b), add a and b into C, remove (a, b), (a, d), (b, c) and (b, e) from E'. $C = \{a, b\}$

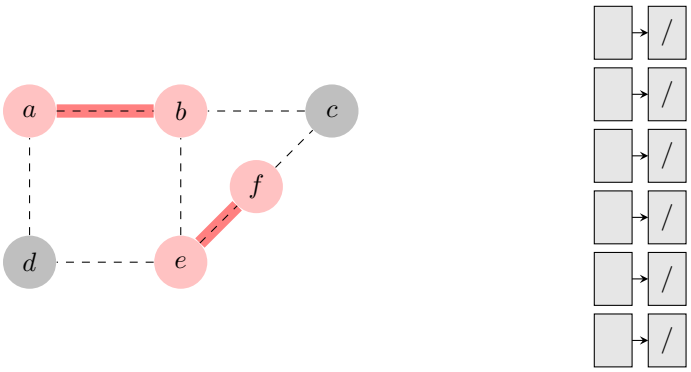


Figure.1(c). Pick edge (e, f), add e and f into C, remove (d,e), (f, c) and (e, f) from E'. $C = \{a, b, e, f\}$, $E' = \emptyset$, halt.

When using adjacency matrix to represent E', the worst case running time of this algorithm is $O(V^2)$. We can achieve a better running time of $O(V + E)$ by using adjacency list to represent E'.

We use a vector Q of size $|V|$ to mark if the vertices are covered. We then put the adjacency list in order in a linear structure, for example, an array. Figure 2 shows how we put adjacency list of G into array S.

APPROX-VERTEX-COVER algorithm scans all edge in the array S in order. When edge (a, b) is picked in line 4. We mark vertices a and b as *covered* is vector Q and put both vertices into vertex cover C. The algorithm keeps visit the next edge in the array and check if it has covered endpoint in Q. If the current edge has endpoint marked as *covered* in Q, we skip this edge and move to the next edge. If not, this edge would be picked and both of its endpoints would be put into C and marked as *covered* in Q. When we reach the end of this array, all the edges in E' are visited, the algorithm halts.

In above process, each element in S is visited once. The cost for scan S is $O(V + E)$, which is the space for adjacency list. And each time we visit an edge, we check both endpoints in vector Q. Each lookup costs $O(1)$. There are $O(V + E)$ times of lookup. Then the total cost of query in Q is $O(V + E)$. Therefore the total cost of APPROX-VERTEX-COVER is $O(V + E)$.

a.head	b	d	/	b.head	a	c	e	/	c.head	b	f	/	d.head	a	e	/	e.head	b	d	f	/	f.head	c	e	/

Figure. 2 Array S of adjacency list for E'.

Theorem 1. *APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.*

Proof. We have shown previously that the running time of APPROX-VERTEX-COVER is $O(V+E)$.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.

Let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in A, any vertex coverin particular, an optimal cover $C*$ must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same vertex from $C*$, and we have the lower bound on the size of an optimal vertex cover:

$$|C*| \geqslant |A|$$

Each execution of line 4 picks an edge for which neither of its endpoints is already in C, and add both endpoints in C. The exact upper bound of size of C is:

$$|C| = 2|A|$$

.
Combine both equations above, we obtain:

$$|C| = 2|A| \leqslant 2|C*|$$

Thereby proving the theorem.

□

3 Subset Sum Problem

3.1 Introduction

The **subset sum problem** is an important problem in complexity theory and cryptography. The problem is this: given a set of integers, is there a non-empty subset whose sum is zero? . Or another

definition of the problem is: a pair (S, t), where S is a set $\{x_1, x_2, \cdots, x_n\}$ of positive integers and t is a positive integer. This decision problem asks whether there exists a subset of S that adds up exactly to the target value t.

This problem is NP-complete. For the proof you can see the Section 34.5.5 in the textbook.

To apply the approximation algorithm , we need to transfer this problem to be a optimization problem. The optimization problem associated with this decision problem arises in practical applications. In the optimization problem, we wish to find a subset of $\{x_1, x_2, \cdots x_n\}$ whose sum is as large as possible but not larger than t. For example, we may have a truck that can carry no more than t pounds, and n different boxes to ship, the ith of which weighs x_i pounds. We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit.

In this section, we present an exponential-time algorithm that computes the optimal value for this optimization problem, and then we show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme. (Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $(1/\epsilon)$ as well as in the size of the input.)

3.2 An exponential-time exact algorithm

Suppose that we computed, for each subset S' of S, the sum of the elements in S', and then we selected, among the subsets whose sum does not exceed t , the one whose sum was closest to t . Clearly this algorithm would return the optimal solution, but it could take exponential time. To implement this algorithm, we could use an iterative procedure that, in iteration i , computes the sums of all subsets of $\{x_1, x_2, \cdots, x_i\}$,using as a starting point the sums of all subsets of $\{x_1, x_2, \cdots, x_{i-1}\}$. In doing so, we would realize that once a particular subset S' had a sum exceeding t , there would be no reason to maintain it, since no superset of S' could be the optimal solution. We now give an implementation of this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $\{x_1, x_2, \cdots, x_n\}$ and a target value t ; well see its pseudocode in a moment. This procedure iteratively computes L_i,the list of sums of all subsets of $\{x_1, x_2, \cdots, x_n\}$ that do not exceed t , and then it returns the maximum value in L_n.
If L is a list of positive integers and x is another positive integer, then we let $L + x$ denote the list of integers derived from L by increasing each element of L by x. For example,if $L =< 1, 2, 3, 5, 9 >$, then $L + 2 =< 3, 4, 5, 7, 11 >$. We also use the notation for the sets, so that

$$S + x = \{s + x : s \in S\}$$

We also use an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed. Like the MERGE procedure we used in merge sort, MERGE-LISTS runs in time $O(|L| + |L'|)$. We omit the pseudocode for MERGELISTS.

Algorithm EXACT-SUBSET-SUM $((S, t))$

1: $n \leftarrow |S|$
2: $L_0 \leftarrow < 0 >$
3: **for** $i = 1$ to n **do**
4: $\quad L_i =$ MERGE-LISTS $(L_{i-1}, L_{i-1} + x_i)$
5: $\quad$ remove from L_i every element that is greater than t
6: **end for**
7: **return** the largest element in L_n

To see how EXACT-SUBSET-SUM works, let P_i denote the set of all values obtained by selecting a (possibly empty) subset of $\{x_1, x_2, \cdots, x_n\}$. and summing its members. For example, if $S = \{1, 4, 5\}$, then
$P_1 = 0, 1$
$P_2 = \{0, 1, 4, 5\}$
$P_3 = \{0, 1, 4, 5, 6, 9, 10\}$
Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i)$$

we can prove by induction on i that the list L_i is a sorted list containing every element of P_i. whose value is not more than t. Since the length of L_i can be as much as 2^i.EXACT-SUBSET-SUM is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which t is polynomial in $|S|$ or all the numbers in S are bounded by a polynomial in $|S|$.

3.3 A fully polynomial-time approximation scheme

We can derive a fully polynomial-time approximation scheme for the subset-sum problem by trimming each list L_i after it is created. The idea behind trimming is that if two values in L are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter δ such that $0 < \delta < 1$.When we trim a list L by δ, we remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for every element y that was removed from L, there is an element z still in L' that approximates y, that is,

$$\frac{y}{1+\delta} \leq z \leq y.$$

We can think of such a z as "representing" y in the new list L'. Each removed element y is represented by a remaining element z satisfying inequity above.

The following procedure trims list $L =< y_1, y_2, \cdots, y_m >$ in them $\Theta(m)$, given L and δ and assuming that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.
Here is the psudocode for TRIM:

The procedure scans the elements of L in monotonically increasing order. A number is appended

Algorithm TRIM(L, δ)

```
1: m ← length(L)
2: L' ←< y_1 >
3: last ← y_1
4: for i = 2 to m do
5:     if y_i > last × (1 + δ) then
6:         append y_i noto the end of L'
7:         last ← y_i
8:     end if
9: end for
10: return L'
```

onto the returned list L' only if it is the first element of L or if it cannot be represented by the most recent number placed into L'.

Given the procedure TRIM, we can construct our approximation scheme as follows. This procedure takes as input a set $S = \{x_1, x_2, \cdots, x_n\}$ of n integers (in arbitrary order), a target integer t ,and an approximation parameter ϵ, where $0 < \epsilon < 1$.

It returns a value z whose value is within a $1 + \epsilon$ factor of the optimal solution.
Here is the psudocode for APPROX-SUBSET-SUM:

Algorithm APPROX-SUBSET-SUM(S, t)

```
1: n ← |S|
2: L_0 ←< 0 >
3: for i = 1 to n do
4:     L_i ← MERGE − LISTS(L_{i−1}, L_{i−1} + x_i)
5:     L_i ← TRIM(L_i, ε/2n)
6:     remove from L_i every element that is greater than t
7: end for
8: Z* ← max(L_n)
9: return Z*
```

As an example, suppose we have the instance.
$S =< 104, 102, 201, 101 >$ with $t = 308$ and $\epsilon = 0.40$. The trimming parameter δ is $\epsilon/8 = 0.05$. APPROXSUBSET- SUM computes the following values on the indicated lines:
line 2 : $L_0 =< 0 >$,

line 4 : $L_1 =< 0, 104 >$,
line 5 : $L_1 =< 0, 104 >$,
line 6 : $L_1 =< 0, 104 >$,

line 4 : $L_2 =< 0, 102, 104, 206 >$,
line 5 : $L_2 =< 0, 102, 206 >$,

line 6 : $L_2 =< 0, 102, 206 >$,

line 4 : $L_3 =< 0, 102, 201, 206, 303, 407 >$,
line 5 : $L_3 =< 0, 102, 201, 303, 407 >$,
line 6 : $L_3 =< 0, 102, 201, 303 >$,

line 4 : $L_3 =< 0, 101, 102, 201, 206, 302, 303, 404 >$,
line 5 : $L_3 =< 0, 101, 201, 302, 404 >$,
line 6 : $L_3 =< 0, 101, 201, 302 >$,

The algorithm returns $z^\star = 302$ as its answer, which is within $\epsilon = 40\%$ of the optimal answer 307.

Theorem 2. *APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.*

Proof. The operations of trimming L_i in line 5 and removing from L_i every element that is greater that t maintain the property that every element of L_i is also a member of P_i. Therefore , the calue $z^\star$ returned in line 8 is indeed the sum of some subset of S. let $y^\star \in P_n$ denote an optimal solution to the subset-sum problem. Then, from line 6, we know that $z^\star \leq y^\star$. By inequality $max\left(\frac{C}{C^\star}, \frac{C^\star}{C}\right) \leq \rho(n)$ we need to show that $y^\star/z^\star \leq 1 + \epsilon$ We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.
or every element y inP_i that is at most t , there exists an element $z \in L_i$ such that

$$\frac{y}{(1+\epsilon/2n)^i} \leq z \leq y.$$

This inequality must hold for$y^\star \in P_n$, and therefore there exists an element $z \in L_n$ such that

$$\frac{y^\star}{(1+\epsilon/2n)^i} \leq z \leq y^\star.$$

and thus,

$$\frac{y^\star}{z} \leq (1 + \frac{\epsilon}{2n})^n$$

Since there exists an element $z \in L_n$ fulfilling inequality above, the inequality must hold for $z^\star$, which is the largest value in L_n; that is ,

$$\frac{y^\star}{z^\star} \leq (1 + \frac{\epsilon}{2n})^n$$

Now, we show that $\frac{y^\star}{z} \leq (1+\epsilon)$. We do so by showing that $(1+\epsilon/2)^n \leq 1+\epsilon$ .We have $\lim_{n\to\infty}(1+\epsilon/2)^n = e^{\epsilon/2}$, and

$$\frac{d}{dn}(1 + \frac{\epsilon}{2n})^n > 0.$$

Therefore,the function $(a + \epsilon/2)^n$ increases with n as at approaches its limit of $e^{\epsilon/2}$, and we have

$$e^{\epsilon/2} \leq 1 + \epsilon/2 + (\epsilon/2)^2 \leq 1 + \epsilon$$

as $\epsilon \leq 1$. Thus $(1 + \epsilon/2)^n < 1 + \epsilon$, which means

$$\frac{y^\star}{z^\star} \leq (1 + \epsilon).$$

Thus we finish the analysis of the ratio.
To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, we derive a bound on the length of L_i,After trimming, successive elements z and z' of L_i must have the relationship $z'/z > 1 + \epsilon/2$. That is, they must differ by a factor of at least $1 + \epsilon/2$. Each list, therefore, contains the value 0, possibly the value 1, and up to $\lfloor log_{1+\epsilon/2n} t \rfloor$ additional values. The number of elements in each list L_i is at most

$$\begin{aligned} log_{1+\epsilon/2} t &= \frac{lnt}{ln(1 + \epsilon/2n)} + 2 \\ &\leq \frac{2n(1 + \epsilon/2n)lnt}{\epsilon} + 2 \\ &\leq \frac{3nlnt}{\epsilon} + 2 \end{aligned}$$

This bound is polynomial in the size of the inputwhich is the number of bits lgt needed to represent t plus the number of bits needed to represent the set$S$, which is in turn polynomial in $n$and in $1/\epsilon$. Since the running time of APPROX-SUBSETSUM is polynomial in the lengths of the L_i , we conclude that APPROX-SUBSETSUM is a fully polynomial-time approximation scheme. □

4 Approximation Algorithm Design Techniques

Approximation algorithms have wide practical application. There are several fundamental techniques used in the design and analysis of approximation algorithms. This section will have a brief review of these techniques.

One important technique is by using linear programming. Combinatorial problems such as subset sum can be present in the form of integer programming. We can get a linear programming from this integer programming problem by relaxation. Then we can use linear programming rounding or the primal-dual property to design approximation algorithm.

The primal-dual property is that each linear programming problem, referred to as a primal problem, can be converted into a dual problem, which provides an upper bound to the optimal value of the primal problem. Therefore we can solve the prime problem by finding a feasible solution for the dual problem[2].

$$\begin{aligned} &prime\ problem \quad Max \qquad C^T x \\ &\qquad\qquad\qquad\quad Subject\ to. \ Ax \leqslant b, x \geqslant 0 \end{aligned}$$

$$\begin{array}{lll} \textit{dual problem} & Min & y^T b \\ & Subject\ to. & y^T A \geqslant c^T, y \geqslant 0 \end{array}$$

The primal-dual property is widely used in design approximation algorithm. We can give a 2-approximation algorithm for weighted vertex cover with this technique.

Greedy algorithms and local search are also popular techniques for designing approximation algorithm. Greedy algorithm of set cover problem gives a *logn* approximation ratio (CLRS C35.3). Local search algorithm for job scheduling problem can give a constant factor approximation ratio.

5 Reference

1. Thomas H. Cormen, Charles E. Introduction To Algorithms, Third Edition. MIT Press 2009.
2. Vijay Vazirani. Approximation Algorithms. Springer-Verlag, 2004.
3. University of Wisconsin Madison. CS 880: Topics in TCS: Approximation Algorithms. Spring 2007.