

**Revision: approximation algorithm**

- Since many optimization problems are *NP*-hard, we cannot expect that we can find the optimal solution in polynomial time.
- We thus accept algorithm that returns a solution that is not optimal.
- On the other hand, we do not want to accept solutions that are arbitrarily bad, i.e., need much higher cost or gain much less value than the optimal solution.
- An approximation algorithm finds a solution that has a ratio bound.
- We say that an approximation algorithm has the ratio bound  $\rho$  if the solution found by the algorithm is never more than  $\rho$  times as costly as the optimal solution, or achieves less than  $1/\rho$  of the value of the optimal solution.

ALG/TITCS L08-1

**Ratio bounds that depends on size of input**

- In the last lecture, we show two problems (Vertex-cover, TSP-with-triangle-inequality) that can be approximated with a ratio bound of 2.
- We also see that some problems (TSP) are so hard to approximate, that it cannot be approximated to any factor  $\rho$ .
- There is an intermediate, that approximation algorithm does exist, but their ratio bound depends on the size of the problem.
- We study one such problem in this lecture: the Set-cover problem. We will learn an approximation algorithm that has a ratio bound of about  $\ln(n)$ .

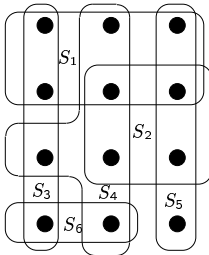
ALG/TITCS L08-2

**The Set-cover problem**

The Set-cover problem is defined as follows:

Given a set  $X$  and a set of subsets  $\mathcal{F}$  of  $X$ , find a set  $\mathcal{C} \subseteq \mathcal{F}$  of minimum size that is a cover for  $X$ , i.e., for each element  $e \in X$ ,  $e \in S$  for some  $S \in \mathcal{C}$ .

Example: Optimal solution is



ALG/TITCS L08-3

**Set-cover is NP-hard**

It is relatively trivial to see that Set-cover is *NP*-hard: it is an extension of the vertex cover problem, so VERTEX-COVER reduces to it:

To solve an instance  $\{G = (V, E), k\}$  for VERTEX-COVER, we turn it into the following instance of Set-cover:  $X = E$ ,  $\mathcal{F} = V$ , and  $e \in v$  if and only if  $e$  is adjacent to  $v$ , where  $v \in V$ . Then we check the size of the solution. If it is at most  $k$ , we accept. Otherwise, we reject.

In general, **generalizations of NP-hard problems are also NP-hard**. Be careful: we are talking about generalizations, i.e., those problems that has a special case being a *NP*-hard problem.

We are not talking about restriction or relaxations of the requirement for which an instance is in a language. I.e., we are not talking about another language that is a subset or a superset of a *NP*-complete language. Such a language need not be *NP*-hard. (Examples?)

ALG/TITCS L08-4

**A simple algorithm: GREEDY-SET-COVER**

Here is a very simple greedy strategy for solving the Set-cover problem:

```

C ← ∅
While there is still an uncovered element
do Let  $X_i \in \mathcal{F} - C$  be the set that contains the most uncovered element.
   C ← C ∪ { $X_i$ }
return C
    
```

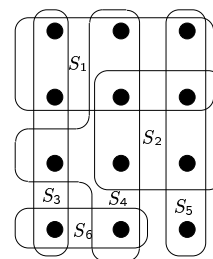
Unlike the greedy strategies that we learned at the beginning of the course, this algorithm is not optimal. (Which property fails to hold, greedy-choice or optimal-substructure, or both?)

It is to be expected, since Set-cover is *NP*-hard, and the greedy strategy is a polynomial time algorithm.

ALG/TITCS L08-5

**Example run of GREEDY-SET-COVER**

For the previous set...



Note that the solution is not very bad: it has "cost" of 4, instead of 3 (optimal). The question is, is there a guarantee that the cost will not be too bad for any graph?

ALG/TITCS L08-6

**The guarantee of GREEDY-SET-COVER**

The algorithm has the following interesting ratio bound:

- Let  $m$  be the size of the largest set in  $\mathcal{F}$ . Then the algorithm has a ratio bound of  $1 + 1/2 + 1/3 + \dots + 1/m \equiv H(m)$ , the  $m$ -th harmonic number.
- Since for all  $m$ ,  $H(m) < \ln m + 1$ , the algorithm has a ratio bound of  $\ln m + 1$ .

Exercise:

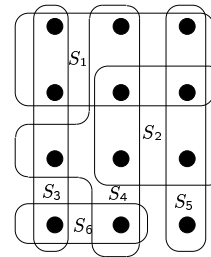
- Show the bound is tight, i.e., there exists sets that the number of subsets chosen by the greedy algorithm may be  $H(m)$  times the number of subsets chosen by the optimal solution.
- What will happen if we use this algorithm for vertex cover? (p968, Ex 37.1-2)

**The cost of an element**

For analysing the algorithm GREEDY-SET-COVER, we share the unit cost in selecting a set among all the elements that are newly covered.

That is, cost  $c(x)$  of an element  $x \in X$  is defined to be  $1/N(x)$ , where  $N(x)$  is the number of new elements covered by the set covering  $x$  for the first time during GREEDY-SET-COVER.

Example:



**The total cost of GREEDY-SET-COVER**

The total cost of GREEDY-SET-COVER is the number of sets selected,  $|C|$ .

For each set, we assign costs  $c(x)$  to some elements  $x$ . It is clear that the total cost assigned is  $N(x)(1/N(x)) = 1$ . Therefore, for each set we assigned a total cost of 1 to the elements.

If we sum the costs of all the elements, we get exactly the total cost assigned by all the sets. This is of course equal to  $|C|$ .

Since each element is assigned a cost once, the cost of the greedy solution is  $|C| = \sum_{x \in X} c(x)$ , i.e., the total of the cost of each element.

**The benefit of using element cost**

- The unit costs of the sets is simple: they are always one.
- But they do not help when we compare our solution to the optimal solution: the optimal solution might not choose our sets.
- On the other hand, the optimal algorithm must still cover all the elements. We can thus achieve a possible comparison if we get down to the element level: we push our costs down to each of the elements.
- Element costs thus give us a way for comparing the costs of the optimal solution and the greedy algorithm: by seeing how much cost we incurred for each set of the optimal algorithm.

**The comparison**

More precisely, the element costs allow us to write the following:

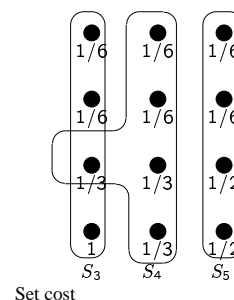
$$|C| = \sum_{x \in X} c(x) \leq \sum_{S \in C^*} \sum_{x \in S} c(x)$$

This is because  $C^*$  does include all the elements, so for all  $x$ ,  $x \in S$  for some  $S \in C^*$ .

Now it is clear that the only thing that we need to do is to bound  $\sum_{x \in S} c(x)$ , where  $S \in C^*$ . Exactly, we bound it by  $H(|S|)$ .

Since  $|S|$  is at most  $m$ , we know that  $|C| < \sum_{S \in C^*} H(m) = H(m)|C^*|$ , concluding the proof.

**Example**



### Bounding the set costs

Consider a particular set  $S$  chosen by the optimal solution  $C^*$ . Let the elements in  $S$  be  $x_1, x_2, \dots, x_k$ , listed in the reverse order when the elements are covered by GREEDY-SET-COVER.

We will show that  $c(x_k)$  is no larger than  $1/k$ , leading to  $\sum_{x \in S} c(x) \leq H(|S|)$ .

**Proof:** Suppose this is not true, so  $c(x_k) = 1/t$ , where  $t < k$ .

That is, when  $x_k$  is chosen, GREEDY-SET-COVER chooses a set  $S'$  of size  $t < k$ . But at that time,  $x_1, x_2, \dots, x_k$  are all uncovered.

This means that GREEDY-SET-COVER should really choose  $S$  (and so  $c(x_k) \leq 1/k$ ) instead, since  $S$  contains more uncovered elements than  $S'$ . We thus have a contradiction, proving the bound.

ALG/TITCS L08-13

### Problems that are easy to approximate

- There are  $NP$ -complete problems that are very easy to approximate.
- For these problems, there are not just approximation algorithms with constant ratio bound, but the constants can be made arbitrarily close to 1 by increasing the amount of time spent by the algorithm.
- Technically, we should not call them approximation algorithms, but approximation schemes. That is, they have an external parameter specifying the target accuracy. When we need better accuracy, we spend more time on it.
- The target accuracy is usually given by a parameter  $\epsilon > 0$ . Then the ratio bound of the algorithm should be things like  $1 + \epsilon$  or  $1/(1 - \epsilon)$ . The algorithm should still be polynomial time whatever value of  $\epsilon$  it receives.
- If the running time is polynomial to  $1/\epsilon$  as well, we say the scheme is fully polynomial time.

ALG/TITCS L08-14

### Subset-sum

Subset-sum is the following optimization problem:

Given a set  $S$  of positive integers and a target  $t$ , find the maximum number  $L \leq t$  that is a sum of a subset of  $S$ .

Even the decision problem asking whether there exists a subset of  $S$  summing to  $t$  (SUBSET-SUM) is  $NP$ -complete. Subset-sum is a generalization of SUBSET-SUM, so Subset-sum is  $NP$ -hard.

**Proof:** textbook page 951. It reduces VERTEX-COVER to it in polynomial time.

ALG/TITCS L08-15

### $NP$ -complete problems and Strongly $NP$ -complete problems

- Subset-sum is a very special  $NP$ -complete problem: it is  $NP$ -hard only if we consider a number  $x$  to be of size  $\log x$  (number of bits needed to represent  $x$ ).
- If we consider the number to be of size  $x$ , then it is polynomial-time solvable.
- That is, a "pseudo-polynomial-time algorithm" exists for Subset-sum. The algorithm runs in time proportional to a polynomial of  $t + \sum_{x \in S} x$ . (Actually, it runs in  $O(|S|t)$  time.)
- If we consider that we only need  $\log t + \sum_{x \in S} \log x$  bits to specify the input, such an algorithm is clearly too slow. We want an algorithm with running-time polynomial to  $\log t + \sum_{x \in S} \log x$ .
- Sometimes such  $NP$ -complete problems are said to be "weakly  $NP$ -complete", and other  $NP$ -complete problems are said to be "strongly  $NP$ -complete".

ALG/TITCS L08-16

### Example

Suppose  $S = \{104, 102, 201, 101\}$ , and  $t = 308$ .

- Using 104 alone we can make 0 and 104 (either choosing 104 or not choosing 104).
- Using 104 and 102 we can make
- Using 104, 102 and 201 we can make
- Using 104, 102, 201 and 101 we can make

The optimal value is  $307 = 101 + 102 + 104$ .

Of course, if we follow this, we get an exponential time algorithm. Since the problem is  $NP$ -hard, any solution is expected to be super-polynomial time.

ALG/TITCS L08-17

### The exponential-time algorithm

We can write the exponential-time algorithm in the following way:

```

L ← ∅
for each number x ∈ S
do L' ← L
  Add x to each element of L'
  L ← L ∪ L'
  // (1)
Return the largest number x in L that is at most t.

```

We can actually reduce the amount of time needed, by dropping all the numbers that are larger than  $t$  at (1) above. If we get an intermediate number that is already larger than  $t$ , there is no reason that we need to continue working on it.

Exercise: implement this algorithm using  $O(|S|t)$  time.

ALG/TITCS L08-18

### Approximating Subset-sum

Since the problem is  $NP$ -hard, we cannot really hope that we can find a polynomial-time algorithm for it. But we can approximate it in polynomial-time.

The key idea of the approximation is called “trimming”, which usually works for weakly  $NP$ -complete problems.

Basically, we ignore all the “low-level details”, and focus only on the “larger parts of the numbers”.

More precisely, after each time we produce a new list of numbers, we remove all numbers that are “very close together”, except the smallest of them.

The smallest number thus represents all the numbers that are “very close” to it.

ALG/TITCS L08-19

### The polynomial-time approximation scheme

```
 $L \leftarrow \{0\}$ 
for each number  $x \in S$ 
do  $L' \leftarrow L$ 
   Add  $x$  to each element of  $L'$ 
    $L \leftarrow L \cup L'$ 
   Remove all elements larger than  $t$  from  $L$ 
    $L' \leftarrow \emptyset$ 
   for each number  $x$  from the smallest to the largest
   do if the largest number in  $L'$  is less than  $(1 - \epsilon/n)x$ 
      then  $L' \leftarrow L' \cup \{x\}$ 
    $L \leftarrow L'$ 
Return the largest number  $x$  in  $L$  that is at most  $t$ .
```

ALG/TITCS L08-20

### Example

Suppose  $S = \{104, 102, 201, 101\}$ ,  $t = 308$  and  $\epsilon = 0.2$ .

- Using 104 alone we can make 0 and 104.
- Adding in 102 we can make
  - Cleaning it up, we get
- Adding in 201 we can make
  - Cleaning it up, we get
- Adding in 101 we can make
  - Cleaning it up, we get

The found value:

ALG/TITCS L08-21

### Analysis

- Intuitively, each time we round a number, we introduce an error of at most a factor of  $(1 - \epsilon/n)$ . Thus the accumulated error is at most a factor of  $(1 - \epsilon/n)^n > (1 - \epsilon/n)^{n-1} - \epsilon/n > \dots > 1 - \epsilon$ . (Exercise: prove it rigorously! i.e., show that if the solution we found is  $L$  and the optimal solution is  $L^*$ , then  $L/L^* > 1 - \epsilon$ .)
- Thus the algorithm achieves a ratio bound of  $1/(1 - \epsilon)$ .
- On the other hand, the running time of the algorithm depends on the length of  $L$  at the beginning of each iteration.
- $L$  contains only integers of size at most  $t$ . Each two integers must be at least a factor of  $(1 - \epsilon/n)$  from each other.
- Thus there are only at most  $\log_{(1-\epsilon/n)} t$  integers in  $L$ . That is, at most  $n \ln t/\epsilon$  integers. So the running time is bounded by  $n^2 \ln t/\epsilon$ .

ALG/TITCS L08-22