CSE 6311 : Advanced Computational Models and Algorithms Jan 28, 2010

Lecture Notes For Subset Sum

Professor: Dr.Gautam Das *Lecture by: Saravanan*

Introduction

Subset sum is one of the very few arithmetic/numeric problems that we will discuss in this class. It has lot of interesting properties and is closely related to other NP-complete problems like *Knapsack* . Even though Knapsack was one of the 21 problems proved to be NP-Complete by *Richard Karp* in his seminal paper, the formal definition he used was closer to subset sum rather than Knapsack.

Informally, given a set of numbers S and a target number t, the aim is to find a subset $S^{'}$ of S such that the elements in it add up to t. Even though the problem appears deceptively simple, solving it is exceeding hard if we are not given any additional information. We will later show that it is an NP-Complete problem and probably an efficient algorithm may not exist at all.

Problem Definition

The decision version of the problem is : Given a set S and a target t does there exist a subset $S^{'} \subseteq S$ such that $t = \sum_{s \in S'} s$.

Exponential time algorithm approaches

One thing to note is that this problem becomes polynomial if the size of $S^{'}$ is given. For eg,a typical interview question might look like : given an array find two elements that add up to t. This problem is perfectly polynomial and we can come up with a straight forward $O(n^2)$ algorithm using nested for loops to solve it. (what is the running time of best approach ?).

A slightly more complex problem asks for ,say, 3 elements that add up to t. Again, we can come up with a naive approach of complexity $O(n^3)$. (what is the best running time?). The catch in the general case of subset sum is that we do not know $|S^{'}|$. At the worst case $|S^{'}|$ is $O(n)$ and hence the running time of brute force approch is approximately $n^{O(n)}$.

A slightly more efficient algorithm checks out all possible 2^n subsets. One typical way to

do this is to express all numbers from 0 to $2^n - 1$ in binary notation and form a subset of elements whose indexes are equal to the bit positions that correspond to 1. For eg, if n is 4 and the current number, in decimal, is say 10 which in binary is 1010. Then we check the subset that consists of 1^{st} and 3^{rd} elements of S. One advantage of this approach is that it uses constant space. At each iteration, you examine a single number. But this approach will lead to a slower solution if $|S^{'}|$ is small. Consider the case where $t = S[\frac{n}{2}]$. We will have to examine around $O(2^{\frac{n}{2}})$ different subsets to reach this solution.

A slightly different approach finds all possible sums of subsets and checks if t has occurred in the subset.

```
EXPONENTIAL-SUBSET-SUM(S,t):
n =  |S|
L_0 = 0
for i in 1 to n
     L_i = merge-lists(L_i − 1, L_i − 1 + S[i])
     if L_i has t, return true.
     remove all elements greater than t from L_i
if L_n has t, return true else return false
```

This algorithm uses the notation $S + x$ to mean $s + x : s \in S$. Refer CLRS 35.5 for a discussion of a similar algorithm for a variant of subset sum problem.

NP-Completeness of Subset Sum Decimal

In this section we will prove that a specific variant of Subset sum is NP-Complete. Subset sum decimal is defined very similar to standard Subset sum but each number in S and also t is encoded in decimal digits.

We can show that Subset sum decimal is in class NP by providing the subset $S^{'}$ as the certificate. Clearly, we can check if elements in S' adds up to t in polynomial time.

The next step is to select another NP-Complete problem which can be reduced to Subset sum decimal. So far we have not discussed any arithmetic NP complete problems. The only non graph theoretic probem that we have discussed in 3SAT and we will use it for the proof. Of course there are multitude of other reductions including Vertex cover, 3 dimensional matching, partition etc.

We are now given a 3SAT formula ϕ with n variables - x_1, x_2,. . . ,x_n and m clauses - C_1, C_2,. . . , C_m. Each clause C_i contains exactly 3 literals. Our aim is to construct an instance of subset sum problem $< S, t >$ such that ϕ is satisfiable if and only if a solution to our instance of Subset sum decimal exists. The outline of the proof is as follows :

1. Construct a set S of *unique* large decimal numbers that somehow encode the constraints of ϕ. Additionally this operation must take polynomial time.
2. Construct an appropriate target t such that this instance of Subset sum decimal is solvable if and only if a solution to 3SAT instance exists. Handle complications like carries in addition.
3. Devise a way to find the satisfying assignment from subset solution and vice versa.

To simplify the proof, we make the following assumptions :

1. All the literals x_1 to x_n is used in some clause of ϕ .
2. No clause can contain both a literal and its complement.

As a consequence of these assumptions, we do not have any variables that are superflous. Also we do not have any clauses that get satisfied trivially.

We will not duplicate the proof in the lecture notes as a detailed sketch of the reduction is given in CLRS section 34.5.5. Instead we will focus on certain observations.

Observation 1 : Construction of S and t takes polynomial time

This is easy to see. For each variable x_i we create 2 variables. Similarly we create two variables for each clause C_j. The total number of variables in S is $2(m+n)$. Each number in set S and t contains exactly $n+m$ digits. Hence the total construction takes time polynomial in $n+m$.

Observation 2 : There are no carries when elements in subset are added to form t.

We can see that the only allowed integers in number construction are 0,1 and 2. The columns corresponding to variables (the leading n digits) can add up to at the most 2. The columns corresponding to clauses (trailing m digits) cannot have a sum of more than 6. This is because of two facts : (a) 3SAT has atmost 3 literals in each clause (b) A clause cannot contain a literal and its complement. So, each variable can add atmost 1 to that clause column and there atmost 3 variables in a clause. Additionally, we have 1 and 2 from the slack variables. Concisely, we get atmost 3 from v_i or $v_i^{'}$ and 3 from s_i and $s_i^{'}$.

Hence we can conclude that carries does not occur at each column(digit) as the base we use is 10.

Observation 3 : All variables in S corresponding to x_is are unique.

Each variable x_i creates two variables v_i and $v_{'}$. The proof is in two parts : (a) First we show that if $i \neq j$, v_i and v_j does not match in the leading n digits.

Similar argument holds for $v_i^{'}$ and $v_j^{'}$. (b) Next, we can show that v_i does not equal to $v_i^{'}$. This is because our assumption that a literal and its complement does not occur in the same clause. This means that the trailing m digits will not be equal.

In conclusion, no pair of variables in S corresponding to x_i are equal.

Observation 4 : All variables in S corresponding to C_is are unique.

Each clause C_i creates two variables s_i and $s_i^{'}$. If $i \neq j$, $s_i(s_i^{'})$ and $s_j(s_j^{'})$ does not match in the trailing m digits. Additionally, by construction, $s_i \neq s_i^{'}$ as the digit position corresponding to C_i has 1 for s_i and 2 for $s_i^{'}$.

Observation 5 : All variables in S is unique. i.e. S forms a set.

This can observed from Observation 3 and 4. By construction v_i and s_i do not match. Similar argument hold for $v_i^{'}$ and $s_i^{'}$.

Observation 6 : New variables corresponding x_i and C_j are both needed for proof.

A detailed sketch is given in CLRS. The variables v_i and $v_i^{'}$ created from x_i makes sure that each variable has a unique boolean assignment of 0 or 1. Else the sum for that column in target will be 2. This is due to the assumption that all variables x_i **HAS** to be used in some clause C_j and hence has a unique assignment. Of course, it is possible that ϕ has multiple satisfying assignment but the target digit forces only one of them to be selected when you select the elements of subset $S^{'}$.

The digits corresponding to clauses makes sure that each clause has atleast one variable that evaluates to true. This is because each digit of slack variable corresponding to C_i (ie $s_i, s_i^{'}$) contribute atmost 3 towards t and hence the remaining (atleast) 1 has to come from v_j or $v_j^{'}$s.

So variables v_i ensure that each x_i has a unique assignment. Variables s_i ensure that each clause C_j of ϕ is satisfied.

Observation 7 : Subset sum is NP complete if the numbers are expressed in base $b \geq 7$.

From observation 2 , we know that the maximum possible digit due to summation of elements in S is 6. This means we can reuse the proof of Subset sum decimal to prove that Subset sum is NP-Complete for any base b that is greater that 6.

Observation 8 : Given $S^{'}$ we can find a satisfying assignment for ϕ.

We know that any satisfying subset $S^{'}$ must include either v_i or $v_i^{'}$ for $\forall i$ $1 \leq i \leq n$. If $S^{'}$ includes v_i then set x_i to 1. Else set it to 0.

Observation 9 : Given a satisfying assignment for ϕ , we can find $S^{'}$

This is a bit tricky and is done in two steps. More details can be found in CLRS proof.

- If the satisfying assigment had x_i , then select v_i. Else select $v_i^{'}$.
- For each clause C_j find how many variables in it evaluated to true due to the boolean assignment. Atleast one variable has to be true and atmost 3 variables are true.
  - If C_j has only one variable that evaluates to true, then select s_j and $s_j^{'}$.
  - If C_j has two variables that evaluate to true, then select $s_j^{'}$.
  - If C_j has three variables that evaluate to true, then select s_j.

Observation 10 : If ϕ is not satisfied, then $S^{'}$ cannot be found.

If ϕ is not satisfied, then there exist atleast one clause C_j that is not satisfied. This means that for $n + j^{th}$ digit, the slack variables $s_j, s_j^{'}$ contribute only 3 but the corresponding digit in t has 4. Hence no $S^{'}$ exists.

NP-Completeness of Subset Sum Binary

The formal definition of Subset sum binary is similar to Subset sum decimal . The only difference is that all numbers are encoded in bits.

We can notice that the above proof for Subset sum decimal holds only for numbers expressed in base of atleast 7 (from observation 7). For bases from 1-6, the previous proof does not apply - partly due to the fact that there will be carries during addition. We need an alternate proof approach. Since we have proved Subset sum decimal as NP-Complete , we can use the result to prove Subset sum binary as NP-Complete.

The certificate is the subset $S^{'}$ given in binary. We can see that it can be done in polynominal time and hence Subset sum binary is in NP.

The next step is to reduce Subset sum decimal to Subset sum binary. First we observe that any number encoded in decimal can be encoded to binary in polynomial time and vice versa. When given S and t in decimal as input, we encode them in binary and pass it to our Subset sum binary routine. The decision version of Subset sum binary returns true or false which can be fed directly as result of Subset sum decimal. In the optimization version , we just convert the S' returned by the Subset sum binary subroutine to decimal.

Observation 11 : A decimal number can be converted to binary in polynomial time.

Assume some number n is encoded in both binary and decimal. This means $n = 10^k = 2^{k1}$ where k is the number of digits in the decimal representation and $k1$ is the number of bits needed to encode it.

Taking log to the base 2 on both sides,
$k\ log_2 10 = k1 \implies 3.3k = k1$

So to express a decimal number with k digits, we need between $3k$ - $4k$ bits.

Observation 12 : Subset sum is NP complete for any base $b \geq 2$.

The logarithms of the same number in two different bases differ by atmost a constant. ie, $log_{b1}^{b2} = \frac{log_{b1}^{n}}{log_{b2}^{n}}$.

log_{b1}^{b2} is a constant irrespective of n. So if n needs k digits in base $b1$, then it needs atmost $\frac{k}{log_{b1} b2}$ to be represented in base $b2$. (Verify observation 11 using this equation !).

NP-Completeness of Subset Sum Unary

From observation 12, the only base left is 1 and this section handles the special case where all numbers are expressed in base 1. Subset sum unary is similar to Subset sum decimal where all numbers are expressed in unary notation. Numbers in base 1 are called as being represented in unary. Any number k is represented as 1^k which is a string of k 1's. Let us check if Subset sum unary is NP-Complete .

The certificate is the subset where all elements are expressed in unary. If we are given numbers in *unary*, then verification takes time that is polynomial in the length of individual unary numbers. Hence Subset sum unary is in unary.

To prove Subset sum unary is in NP-Complete , we have to reduce either Subset sum decimal/binary to unary. Superficially, it looks straightforward and hence it seems as though Subset sum unary is in NP-Complete. But the catch is that expressing a number n in base b to unary needs time *exponential* when computed wrt the size of n's representation in base b. For eg, representing a binary number n that needs k bits needs around 2^k's unary digits. We can see that 2^k is exponential when viewed from k.

In summary, converting a number from any base to unary takes exponential time. So we cannot use our reduction technique as there the reduction is *not* polynomial.

Dynamic Programming solution for Subset Sum Unary

What we showed above was that Subset sum unary is in NP but not NP-Complete. Here we show that there exists a dynamic programming formulation for this problem. We represent

the problem as a matrix A of size $n * t$. A is a boolean matrix where the interpretation of cell $A[i, j] = True$ is that there exists a subset of $x_1, x_2, \ldots, x_i$ that sum up to j. ie $\exists S^{'} \subseteq \{x_1, x_2, \ldots, x_i\}$ such that $j = \sum_{s \in S'} s$.

The algorithm goes as follows :

```
SUBSET-SUM-UNARY(S,t):
Form matrix A
Set A[1,0] = True
Set A[1,j] = False unless j==S[1] in which case set A[1,j] to True
for i=2 to t
    for j=2 to n
        if A[i-1,j] == True
           A[i,j] = True
        else if A[i-1,j-x_i] == True
           A[i,j] = True
        else
           A[i,j] = False
```

Consider the set $S = \{2, 3, 4, 5\}$ and let $t = 8$. The worked out DP is given below :

	0	1	2	3	4	5	6	7	8
2	T	F	T	F	F	F	F	F	F
3	T	F	T	T	F	T	F	F	F
4	T	F	T	T	T	T	T	T	F
5	T	F	T	T	T	T	T	T	T

Since $A[5, 8] = True$, we conclude that there exists a subset of S that sum up to t(8).

Strong and Weak NP-Complete Problems

Subset sum is interesing in the sense that its binary/decimal can be proved as NP-Complete but its unary version seems to allow a polynomial looking dynamic programming solution.

Looking at the dynamic programming solution carefully, the time (and space) complexity of the approach is $O(n * t)$ where $n = |S|$ and t is the target. By itself, the DP solution looks feasible and 'somehow' polynomial. But one of the reasons that Subset sum is NP-Complete is due to the fact that it allows "large" numbers. If t is large, then the table A is huge and the DP approach takes a lot of time to complete.

Given S and t , there are two ways to define an polynomial algorithm. One uses the length of S ie n to measure algorithm complexity. From this angle, $O(n * t)$ is not polynomial. This is because t can be huge irrespective of n. For eg, we have have a small set with 4

elements but the individual elements (and t) are of the order , say, $O(10^{10})$. But from the perspective of magnitiude of t, this dynamic programming approach is clearly polynomial.

In other words, we have two ways to anchor our polynomial - $Length[S]$ and $Magnitude[t]$. An algorithm is called pseudo polynomial, if its time complexity is bounded above by a polynomial function of two variables - $Length[S]$ and $Magnitude[t]$.

Problems that admit pseduo polynomial algorithms are called *weak* NP-Complete problems and those that do not admit are called *Strong* NP-Complete problems. For example, Subset sum is a weak NP-Complete problem but Clique is a strong NP-Complete problem.

There are lot of interesting discussion about the strong/weak NP-Complete problems in both Garey and Johnson and in Kleinberg/Tardos. See references for more details.

Observation 13 : Only number theoretic problems admit pseudo polynomial algorithms.

Observation 14 : Strong NP-Complete problems do not admit a pseudo polynomial time algorithm unless P=NP.

References

1. CLRS 34.5.5 - Proof of NP-Completeness of Subset sum.
2. CLRS 35.5 - An exponential algorithm to solve a variant of subset sum problem.
3. Garey and Johnson 4.2 - Discussion of pseudo polynomial time algorithms along with strong and weak NP-Complete problems.
4. Klienberg and Tardos 6.4 - Discusses a variant of the DP algorithm given in the lecture notes and the concept of pseudo polynomial time algorithms. Section 8.8 has an alternate NP-Completeness proof of Subset sum using vertex cover which you can skim through if interested.