CMSC 451: Lecture 19
Approximation: Subset Sum

Subset Sum: The Subset Sum problem (SS) is the following. Given a finite set S of positive integers $S = \{x_1, \ldots, x_n\}$ and a *target value*, t, we want to determine the subset $S' \subseteq S$ that sums to a value that is as close as possible to t, without exceeding it. Consider the following example.

$$S = \{3, 6, 9, 12, 15, 23, 32\} \qquad \text{and} \qquad t = 34.$$

The subset $S' = \{6, 12, 15\}$ sums to $t = 33$, which is optimal, since there is no subset adding up to 34. We can turn this into a decision problem by asking whether there exists a subset that sums exactly to t.

The decision version of the subset sum problem is known to be NP-complete. SS is NP-complete. We will not prove this, but there is a reduction from Vertex Cover. We will present the reduction at the end of the lecture. In this lecture, we will present an approximation algorithm.

Polynomial Approximation Scheme: Next, we will present an approximation algorithm for Subset Sum. In a previous lecture, we introduced the notion of a *polynomial approximation scheme*. This is a function that, for any $\varepsilon > 0$, computes a factor-$(1+\varepsilon)$ approximation to our problem, such that the running time is a polynomial of the input size n.

For example, the running time of the algorithm might be $O(2^{(1/\varepsilon)}n^2)$. It is easy to see that in such cases the user pays a big penalty in running time as a function of ε. (For example, to produce a 1% error, the ε component of the running time would be on the order of 2^{100}, which is way too large for practical purposes.) A *fully polynomial approximation scheme* is one in which the running time is polynomial in both n and $1/\varepsilon$. For example, a running time of $O((n/\varepsilon)^2)$ would satisfy this condition. In such cases, reasonably accurate approximations are computationally feasible.

While PTAS's exist for some NP-complete problems, there are a significant number of NP-complete problems, called *APX-complete*, for which is it suspected that none have PTAS solutions. Indeed, if any of these problems had a PTAS solution, then they all would. Examples include optimization versions of many of the NP-complete decision problems we have studied, including

- MAX-3SAT-3: a variant of 3SAT where the objective is to maximize the number of satisfied clauses
- MAX-IS: maximum-sized independent set
- MIN-VC: minimum-sized vertex cover
- MIN-DS: minimum-sized dominating set
- MIN-TSP: minimum-weight TSP tour in metric spaces

For more information, see this Wikipedia article.

Subset Sum Optimization: In the optimization version of the problem, we are given a set of positive integers $S = \{x_1, \ldots, x_n\}$ and a target value t, and we want to determine the subset whose sum is as large as possible but not larger than t. (In knapsack terms, this corresponds to filling the knapsack as full as possible, without overflowing.)

Let z^* denote the optimum sum achievable. (Clearly, $z^* \leq t$.) Any answer we compute cannot exceed z^*, but we don't want it to be too much smaller. Given an approximation parameter ε, where $0 < \varepsilon$, we seek a subset $S' \subseteq S$ whose sum z achieves an approximation ratio of at most $1 + \varepsilon$, that is,

$$\frac{z^*}{1+\varepsilon} \;\leq\; z \;\leq\; z^*.$$

It will be easier to prove a similar bound. Let's assume that $\varepsilon < 1/2$. We will show that our approximation z satisfies $z^*(1-\varepsilon) \leq z$. This is not quite what we want, but if we run our algorithm with the ε parameter set to $\varepsilon/2$, then our approximation error will be

$$1 - \frac{\varepsilon}{2} \;=\; \left(1 - \frac{\varepsilon}{2}\right)\frac{1+\varepsilon}{1+\varepsilon} \;=\; \frac{1 + \varepsilon - \varepsilon/2 - \varepsilon^2/2}{1+\varepsilon} \;\geq\; \frac{1+\varepsilon-\varepsilon}{1+\varepsilon} \;=\; \frac{1}{1+\varepsilon}.$$

(In the middle inequality, we used the fact that $\varepsilon < 1$ so $\varepsilon^2/2 < \varepsilon/2$.) This change in the value of ε will only affect the constant factors in our running time.

In order to discuss running times, we need to agree on the input size. Each number in the input is of magnitude at most t (since otherwise, we may just ignore it), therefore each needs at most $\lceil \lg t \rceil$ bits to represent. We will take the input size to be $n \lceil \lg t \rceil = O(n \log t)$.

Exact Algorithm: Before presenting the approximation algorithm, let's start with a simple, but very inefficient, exact solutions. The approach will be reminiscent of dynamic programming. We will consider subproblems involving the first i values, for $i = 1, 2, \ldots, n$. Let L_i denote a list of all possible sums of the 2^i subsets of $\{x_1, \ldots, x_i\}$. This includes the empty set whose sum is 0. For example, for the set $\{1, 4, 6\}$, we have

$$L_3 \;=\; \langle 0, 1, 4, 5, 6, 7, 10, 11 \rangle,$$

(which includes the original values plus the sums $1 + 4$, $1 + 6$, $4 + 6$, and $1 + 4 + 6$). L_i can have as many as 2^i elements, but may have fewer, since there may be duplicates. There are two things we will want to do for efficiency:

(1) Remove any duplicates from L_i.

(2) Keep only those sums that are less than or equal to t.

Let us assume we have access to a function `merge`(L_1, L_2), which merges two sorted lists and returns the result. Let $L + x$ denote the list resulting by adding the number x to every element of list L. To incorporate x_i, we perform `Merge`$(L, L + x_i)$, which merges L with a shifted copy of L by x_i. We also have a function `trim`(L, t), which eliminates any duplicates that arose in the merging process and also removes any elements larger than t (see Fig. 1). (Of course, both merging and trimming could be done in a single function, but in the next section we will see why it is useful to separate them.) Both functions run in linear time.

The algorithm operates by initializing L to a sequence containing just the element 0, and then repeatedly includes x_i by merging L with $L + x_i$ and trimming the result. The algorithm

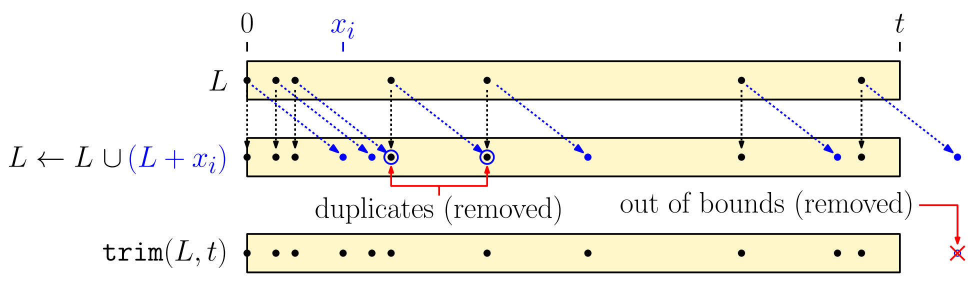


Fig. 1: Merging lists and trimming.

Exact Subset Sum

```
exact-subset-sum(x[1..n], t) {                 // exact solution to subset sum
    L = <0>                                    // basis case - no elements in sum
    for (i = 1 to n) {
        L = merge(L, L + x[i])                 // add in x[i] and...
        L = trim(L, t)                         // ...remove duplicates and items > t
    }
    return the largest element in L            // return largest sum that is at most t
}
```

is presented in the code block below. The final algorithm is presented in the following code block.

A simulation of the algorithm on the input $S = \{1, 3, 4, 6\}$ and $t = 11$ is illustrated in Fig. 2.) The algorithm runs in $O(2^n)$ (exponential!) time in the worst case, because this is the number of sums that are generated if there are no duplicates, and no items are trimmed.

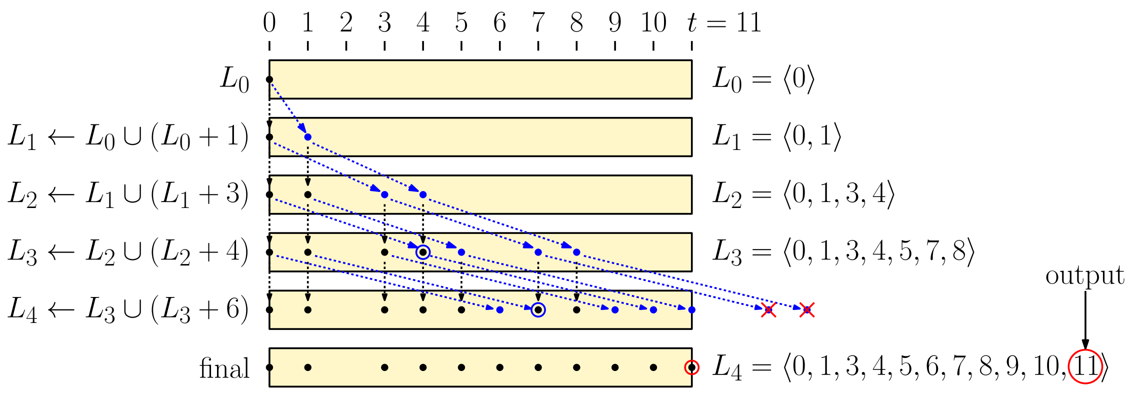

0 1 2 3 4 5 6 7 8 9 10 $t = 11$
L_0
$L_1 \leftarrow L_0 \cup (L_0 + 1)$
$L_2 \leftarrow L_1 \cup (L_1 + 3)$
$L_3 \leftarrow L_2 \cup (L_2 + 4)$
$L_4 \leftarrow L_3 \cup (L_3 + 6)$
final
$L_0 = \langle 0 \rangle$
$L_1 = \langle 0, 1 \rangle$
$L_2 = \langle 0, 1, 3, 4 \rangle$
$L_3 = \langle 0, 1, 3, 4, 5, 7, 8 \rangle$
output
$L_4 = \langle 0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11 \rangle$

Fig. 2: The execution of `exact-subset-sum` on the input $S = \{1, 3, 4, 6\}$ and $t = 11$.

Approximation Algorithm: The exact algorithm runs in exponential time because it is explicitly generating the sums of all 2^n subsets of the n input values. In order to obtain an efficient approximation algorithm, we will not save all the possible sums. Instead, we will be a employ an operation that "compresses" each of the lists L_i by identifying elements that have very similar values and replacing them by a single value. For example, suppose that some list contains both the values 9851 and 9852. Do we really need to keep both of these values, considering that they are roughly 0.01% different from each other?

We need to be careful, however. Suppose, for example, that we have an entire sorted sequence where each value is nearly equal to its predecessor, e.g., $\langle 9851, 9852, 9853, 9854, \ldots\rangle$. Say we kept 9851, how many subsequent values should we eliminate? Given that we want to achieve a relative error of at most $1 - \varepsilon$, the answer is that as soon as we see a value that exceeds $9851/(1 - \varepsilon)$, we must save it. For example, if $\varepsilon = 0.01$, then $9851/(1 - \varepsilon) \approx 9950$, and as soon as we see a value larger than this, we need to keep it.

However, this alone will lead to errors that are too large. The issue is that the small errors that we commit at the early phases of the algorithm will accumulate as more items are included. Since there are n phases in total, it seems like we should make the allowed error smaller as a function of n. Indeed, we will see that the proper allowable error is not ε but $\delta = \varepsilon/n$. (Our final analysis will bear this out.)

The approximation algorithm works as follows. Let L be a sorted list of integers, and let δ denote the desired error bound. We build the compressed list L' as follows. Start by adding the first element of L to L'. The variable `prev` stores the previous item that was added. We traverse L (in ascending order) and when we find the first next element y that is sufficiently different from `prev`, we add it to L'. Because the relative allowed relative error is δ, the condition that an element y is "sufficiently different" is

$$\frac{y - \texttt{prev}}{y} > \delta \qquad \text{or equivalently} \qquad y > \frac{\texttt{prev}}{1 - \delta}.$$

The final compression algorithm is presented in the following code block.

Compressing a list with precision δ

```
compress(L, delta, t) {    // compress sorted list L with error delta and limit t
   L' = empty                          // start with an empty list
   prev = 0                            // previous item added
   for each (y in L) {
       if (y > t) break                // ignore values larger than t
       if (y > prev/(1-delta)) {       // y is different enough?
           append y to L'              // add y
           prev = y                    // update previous
       }
   }
   return L'                           // return the compressed list
}
```

For example, given $\delta = 0.1$ and $t = 60$, and given the input list

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 51, 53, 54, 56, 80\rangle,$$

the algorithm first takes 10. It skips 11, since $11 \le 10/0.9 \approx 11.1$. It takes 12, 15, and 20, but then skips 21 and 22 since both are $\le 20/0.9 \approx 22.2$. It takes 51, but then skips 53, 54 and 56 since they are all $\le 51/0.9 \approx 56.7$. Finally, we ignore 80 because it is larger than t. The final compressed list L' is

$$L' = \langle 10, 12, 15, 20, 51\rangle.$$

There is an alternative way to understand the compression process via bucketing. Suppose that we subdivide the interval from $[1, t]$ into a set of *buckets* of exponentially increasing size. That is, let $\sigma = 1/(1-\delta)$, which is greater than 1. Next, break the real line at the points $\{1, \sigma, \sigma^2, \sigma^3, \ldots\}$, and consider the *bucket* of points lying between any two consecutive break points, $[\sigma^{i-1}, \sigma^i]$. Define an alternative compression procedure `compress`$^*(L, \delta, t)$, which keeps the smallest element in each bucket, provided that element is at most t (see Fig. 3).

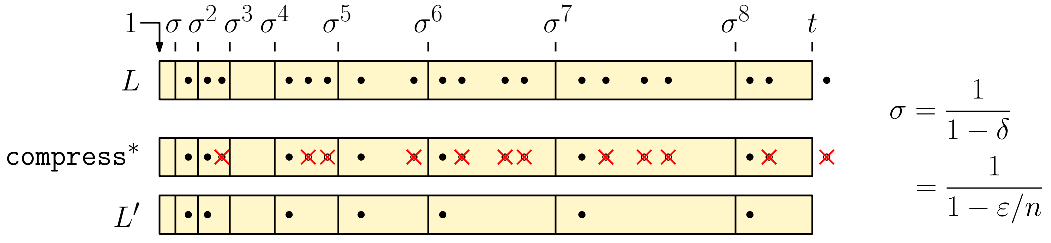


Fig. 3: An alternative view of compression by bucketing.

While these methods do not produce identical results, they are effectively doing the same sort of pruning. For any two such points that lie within the same bucket, $\sigma^{i-1} \leq y \leq y' \leq \sigma^i$, their relative error is

$$\frac{y'-y}{y'} = 1 - \frac{y}{y'} \leq 1 - \frac{\sigma^{i-1}}{\sigma^i} = 1 - \frac{1}{\sigma} = \delta,$$

which means that we can safely keep y and eliminate y'. Thus, we could have equivalently defined the compression process by keeping the smallest value in each bucket and eliminating all the others. (We keep the smallest because any larger value might exceed t, which would be of not use to us.)

Compression is valuable because, rather than storing all 2^n possible sums, we keep a much smaller set. Our next lemma shows that, after compression, the number of items remaining is polynomial in the input size $(n \log t)$ and $1/\varepsilon$.

Lemma: After invoking `compress`(L, δ, t), where $\delta = \varepsilon/n$, the number of nonzero items in the resulting list is $O((n \log t)/\varepsilon)$.

Proof: We know that each pair of consecutive elements in the compressed list differ by a ratio of at least σ, where σ is defined to be $1/(1-\delta)$. Let k denote the number of nonzero elements in the compressed list. It follows that the ratio between the maximum and minimum nonzero elements is at least σ^{k-1}. Since the smallest element is at least 1, and the largest is not greater than t, we have

$$\sigma^{k-1} \leq t \implies k - 1 \leq \frac{\ln t}{\ln \sigma} = \frac{\ln t}{\ln(1/(1-\delta))}.$$

Using the standard facts that $\ln(1/x) = -\ln x$ and $-\ln(1-x) \geq x$, we obtain

$$k \leq 1 + \frac{\ln t}{\ln(1/(1-\delta))} = 1 + \frac{\ln t}{-\ln(1-\delta)} \leq 1 - \frac{\ln t}{\delta}.$$

By our definition of δ, we have

$$k \;\le\; 1 + \frac{n \ln t}{\varepsilon},$$

which is $O((n \log t)/\varepsilon)$, as desired.

We can now present our approximation algorithm. It is essentially the same as the exact algorithm, except we replace the call to `trim` with our more aggressive `compress` function. It is presented in the following code block.

Approximate Subset Sum

```
approx-subset-sum(x[1..n], t, eps) {       // eps-approximate subset sum
    delta = eps/n                          // approx factor per stage
    L = <0>                                // basis case - no elements in sum
    for (i = 1 to n) {
        L = merge(L, L + x[i])             // add in x[i] and...
        L = compress(L, delta, t)          // ...compress similar values and items > t
    }
    return the largest element in L        // return largest sum that is at most t
}
```

An example of the execution of a single phase of the algorithm is shown in Fig. 4 (where we have take the perspective of compression based on bucketing.)

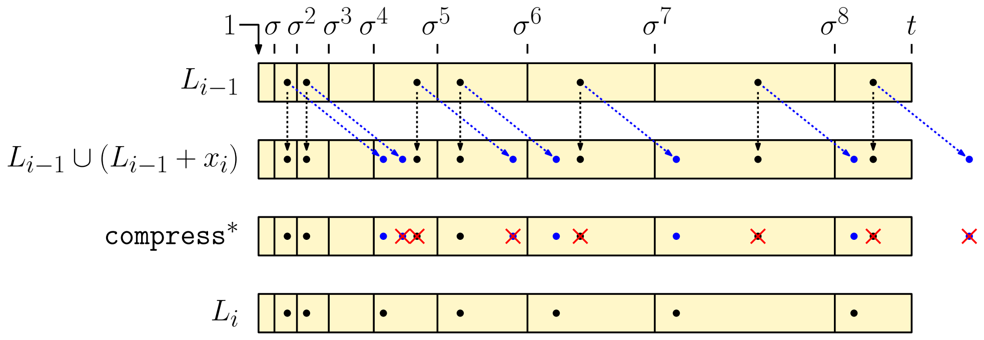


Fig. 4: A single phase of the algorithm (from the bucketing perspective).

The running time of the procedure is $O(n|L|)$. By our previous lemma $|L| = O((n \log t)/\varepsilon)$, so the running time is $O((n^2 \log t)/\varepsilon)$. Note that the input size is $O(n \log t)$, so this is (weakly) polynomial as a function of input size.

Approximation Analysis: The final question is why the algorithm achieves an relative error of at most ε over the optimum solution. Let Y^* denote the optimum (largest) subset sum and let Y_ε denote the value returned by our ε-approximation algorithm. We know that $Y_\varepsilon \le Y^*$. We want to show that Y_ε is not too much smaller, that is,

$$Y_\varepsilon \;\ge\; Y^*(1 - \varepsilon).$$

Recall that our intuition was that we would allow a relative error of ε/n at each stage of the algorithm. Since the algorithm has n stages, the total relative error should be (obviously?)

$n(\varepsilon/n) = \varepsilon$. The catch is that these are relative, not absolute errors. These errors to not accumulate additively, but rather by multiplication. So we need to be more careful.

Let L_i^* denote the i-th list in the exponential-time (exact) algorithm, and let L_i denote the i-th list in the approximate algorithm. We want to show that these two lists are similar. In particular, we will show that for each y in the exact list L_i^*, there is an element z that is close to y. By "close", we mean that z's value is smaller by a factor of just[1] $(1 - \varepsilon/n)^i$. We refer to z as y's *representative*.

Lemma: For $0 \le i \le n$, for each $y \in L_i^*$ there exists $z \in L_i$ such that

$$(1-\delta)^i y \;\le\; z \;\le\; y,$$

where $\delta = \varepsilon/n$.

Proof: We will only prove the lower bound ($(1-\delta)^i y \le z$), and we'll leave the upper bound as an exercise. The proof by induction on i. Initially $L_0 = L_0^* = \langle 0\rangle$. Since both lists are the same, the lemma holds trivially. Let's assume that the induction hypothesis holds for $i-1$, and we will prove it for i. Every element in L_i^* originates in one of two ways, either it is equal to an element $y \in L_{i-1}^*$ or it is $y + x_i$ for some $y \in L_{i-1}^*$. It suffices to show that a representative exists for both y and $y + x_i$.

By our induction hypothesis, there is a representative z to y in L_{i-1}, which means that

$$(1-\delta)^{i-1} y \;\le\; z \tag{1}$$

(see Fig. 5). By adding x_i to each of the above terms and observing that $(1-\delta)^{i-1} < 1$, we obtain a similar bound.

$$(1-\delta)^{i-1}(y + x_i) \;\le\; z + x_i. \tag{2}$$

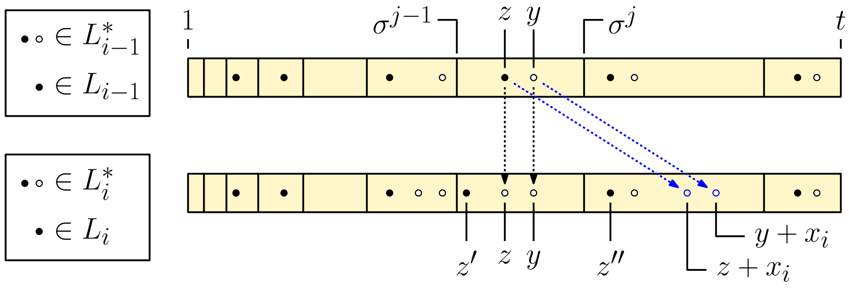


Fig. 5: Subset sum approximation analysis.

One or both of z and $z + x_i$ might not appear in L_i because they were removed by compression. Let z' and z'' be elements of L_i that caused them to be removed. (In

[1]By "just" we mean that this factor is quite close to 1. Recall that we assumed that $\varepsilon < 1$, and $1/n$ is certainly very small for large input sizes. Thus, $1 - \varepsilon/n$ is very close to 1. Raising this to the power i will make is smaller, but we'll show that it won't be much smaller.

the bucket perspective, these are the smallest elements of the buckets containing z and $z+x_i$, respectively.) Thus, z' and z'' are elements of L_i, and by the rules of compression

$$(1-\delta)z \ \le z' \qquad \text{and} \qquad (1-\delta)(z+x_i) \ \le z''.$$

Combining with Eq. (1), we have

$$(1-\delta)^i y \ = \ (1-\delta)^{i-1}(1-\delta)y \ \le \ z'.$$

Doing the same for z'' yields $(1-\delta)^i(y+x_i) \le z''$. Since this applies to all $y \in L^*_{i-1}$ and z and z'' are both in L_i this completes the proof.

Applying the above lemma with $i = n$, with y playing the role of Y^*, and with Y playing the role of z, it follows that there exists $Y \in L_n$ (the final approximation list) such that

$$\left(1-\frac{\varepsilon}{n}\right)^n Y^* \ \le \ Y \ \le \ Y^*.$$

To complete the proof, we will apply a standard result from real analysis, which we will not prove.

Lemma: For $n > 0$ and real a,

$$(1+a) \ \le \ \left(1+\frac{a}{n}\right)^n \ \le \ e^a.$$

By applying this with $a = -\varepsilon$, we have

$$(1-\varepsilon)Y^* \ \le \ Y \ \le \ Y^*.$$

which implies that our final answer Y is a valid answer and satisfies the ε-approximation bound! As observed earlier, the algorithm runs in time $O((n/\varepsilon)\log t)$, which is a polynomial function of both the input size and ε. Therefore, we have obtained the desired PTAS for subset-sum.

Summary: In this lecture, we introduced the subset-sum problem (SS), a simplification of the 0-1 knapsack problem. We showed that the problem is NP-complete, and we also presented a $(1-\varepsilon)$-factor polynomial-time approximation algorithm, that is, a PTAS.

SS is NP-complete: (Optional) The proof that Subset Sum (SS) is NP-complete involves the usual two elements.

(i) SS $\in$ NP.
(ii) Some known NP-complete problem is reducible to SS. In particular, we will show that Vertex Cover (VC) is reducible to SS, that is, VC $\le_P$ SS.

To show that SS is in NP, we need to give a verification procedure. Given S and t, the certificate is just the indices of the numbers that form the subset S'. We can add two b-bit numbers together in $O(b)$ time. So, in $O(n \cdot b)$ time, which is polynomial in the input size, we can compute the sum of elements in S', and verify that this sum equals t.

For the remainder of the proof we show how to reduce vertex cover to subset sum. We want a polynomial time computable function f that maps an instance of the vertex cover (a graph G and integer k) to an instance of the subset sum problem (a set of integers S and target integer t) such that G has a vertex cover of size k if and only if S has a subset summing to t (see Fig. 6). Thus, if subset sum were solvable in polynomial time, so would vertex cover.

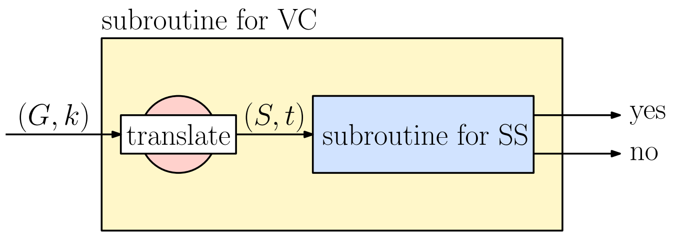
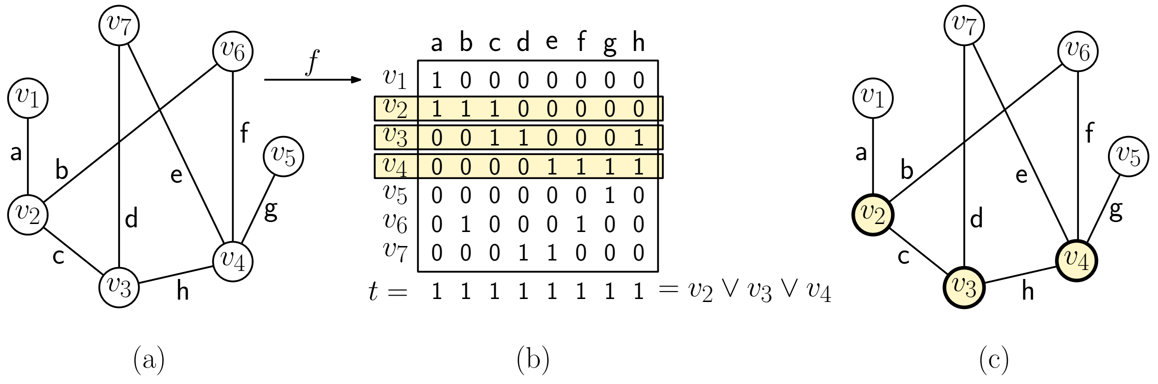


Fig. 6: Reducing VC to SS.

How can we encode the notion of selecting a subset of vertices that cover all the edges to that of selecting a subset of numbers that sums to t. In the vertex cover problem we are selecting vertices, and in the subset sum problem we are selecting numbers, so it seems logical that the reduction should map vertices into numbers. The constraint that these vertices should cover all the edges must be mapped to the constraint that the sum of the numbers should equal the target value.

An Initial Approach: Here is an idea, which does not work, but gives a sense of how to proceed. We are given a pair (G, k), and want to know whether G has a vertex cover of size k. Let m denote the number of edges in the graph. First number the edges of the graph from 1 through m. Then represent each vertex v_i as an m-element bit vector, where the j-th bit from the left is set to 1 if and only if the edge e_j is incident to vertex v_i (see Fig. 7(a) and (b)).



$$\begin{array}{c|cccccccc|}
 & \mathsf{a} & \mathsf{b} & \mathsf{c} & \mathsf{d} & \mathsf{e} & \mathsf{f} & \mathsf{g} & \mathsf{h} \\
v_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
v_2 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
v_3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
v_4 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
v_5 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
v_6 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
v_7 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\end{array}$$

$$t = 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 = v_2 \vee v_3 \vee v_4$$

Fig. 7: First attempt at VC to SS reduction.

Suppose that G has a vertex cover V'. If we take the logical-or of the corresponding bit vectors, since every edge is covered, it follows that the result will be the bit vector consisting of m consecutive 1's, where $m = |E|$. If any edge is not covered, that bit position will have a 0.

This is not quite what we want for two reasons. First, the subset sum problem involves addition, not logical-or. Second, we have no way of controlling how many vertices are used in the cover.

Let's address the first problem. There are two ways in which addition differs significantly from logical-or.

- The first issue is that addition can involve "carries". For example, the $1101 \vee 0011 = 1111$, but in binary $1101_2 + 0011_2 = 13 + 3 = 16 = 1000_2$.

  To fix this, we need to spread the bits out further, so the carries from each column cannot spill over into the next. We'll see that we can do this by representing the numbers in base 4. (Each edge has only two endpoints, so we have at most two 1's in each column.) Of course, the final numbers will be encoded in whatever format our algorithm expects (e.g., binary or decimal).

- The second issue is that an edge may generally be covered either once or twice in the vertex cover. So, a valid vertex cover will be manifest as a sum whose digits are either 1 or 2, e.g. $1211\ldots21112_4$. This is problematic, since we need unique target value t.

  To fix this, we will create a set of m additional *slack values*. For $1 \le i \le m$, the ith slack value, denoted y_i, is a string of all 0's, except for a single 1-digit in the ith position. Since we are working in base 4, this means that $y_i = 4^i$. We will set our final target value to a string of 2's, that is, $t = 2222\ldots222_4$.

  Why does this work? For any edge $e_a = (v_i, v_j)$, if both vertices are in the vertex cover, we know that they will both contribute a 1 in column k, for a total of 2, and we do not need to include the slack value (see Fig. 3(a)). On the other hand, if only one of them is in the cover, then the kth digit will only be 1, and so we will include the slack value y_k to bring the total in column k up to 2, as desired (see Fig. 8(b)).

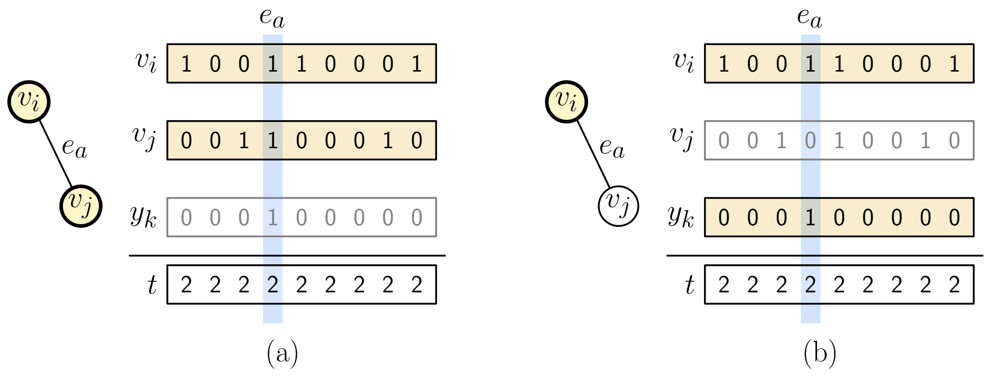


Fig. 8: Using slack values to normalize column sums to 2.

The final issue is how to control the number of vertices in the vertex cover to be k. We will handle this by adding an additional column. For each value arising from a vertex, this column is set to 1. And for each slack variable the value is set to 0. In the target, we will require that the entries in this column sum to the value k (encoded in base-4), the desired size of the vertex cover. Thus, to form the desired sum, we must select exactly k of the vertex values.

Note that since we only have a base-4 representation, there might be carries out of this last column (if $k \geq 4$). But since this is the last column, it will not affect any of the other aspects of the construction.

The Final Reduction: Here is the final reduction, given the graph $G = (V, E)$ and integer k for the vertex cover problem.

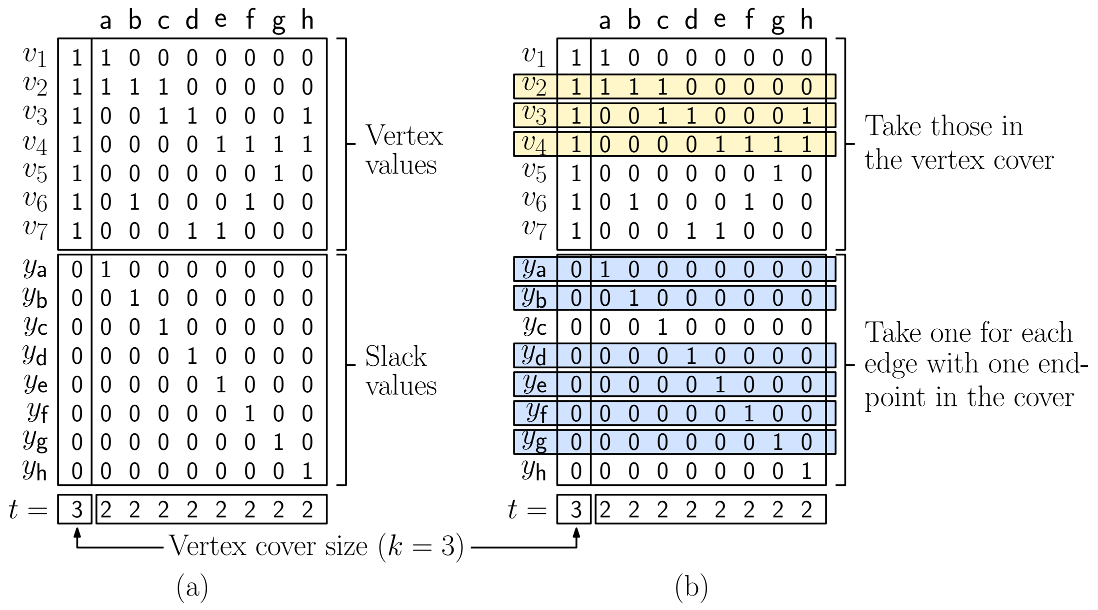
(1) Create a set of n vertex values, $x_1, x_2, \ldots, x_n$ using base-4 notation. Viewed as a base-4 number, x_i, is equal a 1 followed by a sequence of m base-4 digits. The j-th digit of this sequence is 1 if edge e_j is incident to vertex v_i and 0 otherwise.

(2) Create m slack values $y_1, y_2, \ldots, y_m$, where $y_k = 4^k$, that is, a single 1-digit in position k and 0's otherwise.

(3) Let t be the base-4 number whose first digit is k (this may actually span multiple base-4 digits), and whose remaining m digits are all 2.

(4) Convert the x_i's, the y_j's, and t into whatever base numeric base is used for the subset sum problem (e.g. base 10). Output the set $S = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ and t.

Observe that this can be done in polynomial time, in $O(m^2)$, in fact, where $m = |E|$. The construction is illustrated in Fig. 9(a) for the graph shown in Fig. 7(a).



(a)

		a	b	c	d	e	f	g	h
v_1	1	1	0	0	0	0	0	0	0
v_2	1	1	1	1	0	0	0	0	0
v_3	1	0	0	1	1	0	0	0	1
v_4	1	0	0	0	0	1	1	1	1
v_5	1	0	0	0	0	0	0	1	0
v_6	1	0	1	0	0	0	1	0	0
v_7	1	0	0	0	1	1	0	0	0
y_a	0	1	0	0	0	0	0	0	0
y_b	0	0	1	0	0	0	0	0	0
y_c	0	0	0	1	0	0	0	0	0
y_d	0	0	0	0	1	0	0	0	0
y_e	0	0	0	0	0	1	0	0	0
y_f	0	0	0	0	0	0	1	0	0
y_g	0	0	0	0	0	0	0	1	0
y_h	0	0	0	0	0	0	0	0	1
$t =$	3	2	2	2	2	2	2	2	2

(b)

		a	b	c	d	e	f	g	h
v_1	1	1	0	0	0	0	0	0	0
v_2	1	1	1	1	0	0	0	0	0
v_3	1	0	0	1	1	0	0	0	1
v_4	1	0	0	0	0	1	1	1	1
v_5	1	0	0	0	0	0	0	1	0
v_6	1	0	1	0	0	0	1	0	0
v_7	1	0	0	0	1	1	0	0	0
y_a	0	1	0	0	0	0	0	0	0
y_b	0	0	1	0	0	0	0	0	0
y_c	0	0	0	1	0	0	0	0	0
y_d	0	0	0	0	1	0	0	0	0
y_e	0	0	0	0	0	1	0	0	0
y_f	0	0	0	0	0	0	1	0	0
y_g	0	0	0	0	0	0	0	1	0
y_h	0	0	0	0	0	0	0	0	1
$t =$	3	2	2	2	2	2	2	2	2

Vertex values
Slack values
Vertex cover size ($k = 3$)
Take those in the vertex cover
Take one for each edge with one end-point in the cover

Fig. 9: (a) VC to SS reduction and (b) correctness for the vertex cover $V' = \{v_2, v_3, v_4\}$.

Correctness: Correctness is established in the following lemma.

Lemma: G has a vertex cover of size k if and only if S has a subset that sums to t.

Proof: ($\Rightarrow$) Suppose that G has a vertex cover $V' = \{v_1, \ldots, v_k\}$. We construct a solution to subset sum as follows. First, we take the vertex value x_i for each $v_i \in V'$(see Fig. 9(b)).

For each edge e_a that is covered only once in V', we take the corresponding slack variable y_a . It follows from the comments made earlier that the lower-order m digits of the resulting sum will be of the form $222\ldots2$, and because there are k elements in V', the leftmost digit of the sum will be k. Thus, the resulting subset sums to t.

($\Leftarrow$) Conversely, if S has a subset S' that sums to t then we assert that it must select exactly k vertex values x_i, since the first digit must sum to k. We claim that these vertices V' form a vertex cover. In particular, no edge can be left uncovered by V', since (because there are no carries) the corresponding column would be 0 in the sum of vertex values. Thus, no matter what slack values we add, the resulting digit position could not be equal to 2, and so this cannot be a solution to the subset sum problem.

It is worth noting again that in this reduction, we needed to have large numbers. For example, the target value t is at least as large as $4^m \geq 4^n$ (where n is the number of vertices in G). In our dynamic programming solution $W = t$, so the DP algorithm would run in $\Omega(n \cdot 4^n)$ time, which is not polynomial time.