LECTURE NOTES: NP-COMPLETE PROBLEMS

BILL CASSON AND NORA CONNOR

1. Introduction

The study of complexity classes arises from the simple question of "what can be computed?" In this class we have worked mostly with algorithms that run in either polynomial or exponential time.

1.1. **Bridges of Konigsberg.** One of the earliest questions in graph theory arose from Euler in Konigsberg, Prussia[1] when Leonhard Euler was walking through a park. The park was on both sides of the river Pregel with two islands in the river with seven bridges crossing between the banks and the two islands as shown in Figure 1.

Euler was wondering if it was possible to walk through the park and cross each bridge exactly once. This is now known as a Eulerian Path.

Date: April 22, 2013.

[1]Now Kaliningrad, Russia

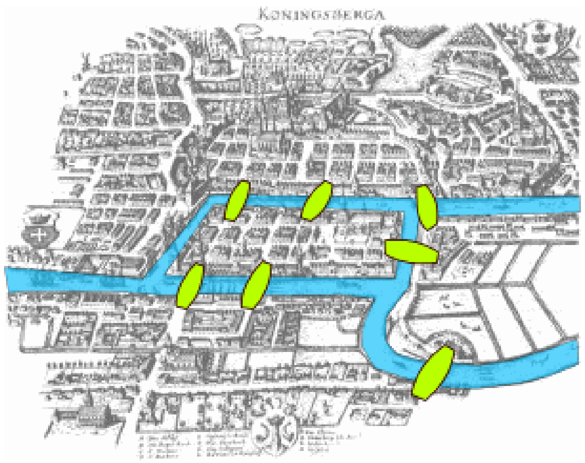


Figure 1. The bridges of Konigsberg with two bridges from the left island to each of the north bank and the south bank, one bridge between the two islands and one bridge from the right island to each of the north and south banks.

1.1.1. *Eulerian Path.* More formally, we give the following definition:

Given a graph $G = (V, E)$, does there exist a path $P = e_1, e_2, \ldots, e_n$ such that each edge $e \in E$ is crossed exactly once?

For the example of the bridges in Figure 1 there does not exist such a path. In general, it is easy to tell if a graph has such a path. Given a graph G = (V, E), G has a Eulerian Path if and only if no more than two vertices have odd degree.

1.1.2. *Hamiltonian Path.* Another question we can ask about the bridges of Konigsberg is: is it possible to visit each vertex exactly once. More formally defined as:

Given a graph $G = (V, E)$, does there exist a path $P = e_1, e_2, \ldots, e_i$ such that each vertex $v \in V$ is visited exactly once?

The solution for this question is not as easy as it was for the Eulerian path. The best known solution is to search through all the paths until either a Hamiltonian Path is found or all paths have been explored.

2. Complexity Classes

Now that we have explored an intuititive definition of what is easy and hard, let's more formally define the classes of P and NP.

2.1. **Complexity class P.** We define the complexity class P as follows:

P is the class of problems for which an algorithm exists that solves instances of size n in time $O(n^c)$ for some constant c.[1]

This is what is usually referred to as a polynomial time algorithm. Problems in this class are considered easy to solve. a few examples of these algorithms would be:

- Most algorithms from class
  - MST
  - BFS
  - Quicksort
- Eulerian path

2.2. **Complexity class NP.** On the other hand, the complexity class NP is based on the time it takes to verify a solution is correct. More formally:

NP is the class of problems A of the following form: x is a yes-instance of A iff there exists a w such that (x, w) is a yes-instance of B, where B is a decision problem in P regarding pairs (x, w) and where $|w| = poly(|x|)$.[1]

In other words, NP is the class of problems for which $w = O(c^n)$, where w is the number of possible solutions, and n is the size of the problem and where a solution w can be checked in polynomial time. W is known as the "witness" or "certificate." At worst, all solutions w must be checked, giving exponential running time. The complexity class P is fully contained in the class NP since it takes polynomial time to solve the problem, it also takes polynomial time to verify that the solution is correct. A few examples of class NP problems are:

- Knapsack
- Hamiltonian paths
- Vertex Cover
- Prime factorization

All of these problems are NP-complete except prime factorization, which is known to be in NP, but has yet to be proven to be NP-complete. We will talk more about NP-completeness in Section 3.

2.3. **P vs. NP.**

One of the great undecided questions in theoretical computer science is whether the class P is a subset of NP or if the classes are equivalent. It is generally believed that P and NP are different but there exists no proof as of now proving that p is equivalent to NP, nor is there a proof that P is a subset of NP.

An interesting side effect of this debate which makes it so important[2] is if any NP-complete problem can be solved in polynomial time, ALL NP-complete problems can be solved in polynomial time by reducing the problem of interest to the problem that had been solved in polynomial time. What we mean by reduce will be covered in 3.2. Therefore, P would equal NP.

3. NP-complete

Problems that can be verified to be in the class NP but not P are considered to be NP-complete. The set of NP-complete problems require exponential time to find a solution and polynomial time to check that a solution is correct. For all NP-complete problems, there exists an algorithm to convert an instance of that problem to an instance of any other NP-complete problem in polynomial time, called a reduction. We will show how these reductions work in 3.2.

The traveling salesman problem (TSP) nicely illustrates what can be NP-complete. The TSP optimization version of the problem where we ask what is the cheapest or shortest path the salesman can take is not NP-complete. However, a trivial modification of the TSP problem to ask can the salesman travel the route in less than a certain distance or spend less than a certain amount then becomes a decision problem and is NP-complete.

[2]This question is part of the Millenium Prize questions[7] along with 6 other hard questions. Thus far, only one of the 7 have been solved.

3.1. Showing NP-completeness.

Now we have shown what NP-complete is, let's show how we prove a problem is NP-complete. First, show that the problem p is in NP. This can be done by showing that given the witness w, the correctness of w can be verified in polynomial time. The other part is to show that another problem p', which is known to be NP-complete, is reducible to p. How this is done is shown next.

3.2. Reductions.

We define A reducible to B, written as $A \leq_p B$ as follows:

A problem A is polynomial-time reducible to a problem B if there exists a polynomial-time function f that converts an instance α of A into an instance β of B such that for all α,A(α) = B(β).

In other words, f converts α into β in polynomial time such that B run with β returns true if and only if A run with α would return true.

4. Example Proof of NP-completeness

To make this process clearer, we will now walk through an example of proving subset sum is NP-complete. First let's define what subset sum is.

4.1. Subset Sum.

We formally define the problem subset sum as follows:

Given a set of numbers S and a target T, does there exist a set $S\prime \subseteq S$ such that the elements of S sum to T?

In other words, is there a subset of the numbers in S, that when summed, sum to some target value.

4.2. Subset Sum Algorithm.

The algorithm for subset sum is recursive. It takes as inputs, the set of numbers, the index of the current element being considered and the remaining sum. These start out as S, n, and T respectively. For each number in S, we will consider whether it should be in the sum or if it should be left out. This is done by subtracting it from sum and making the recursive call with the reduced sum and the next element to be considered for the first case, and calling the recursive function with sum unchanged and the next element for the second case. The base cases are two-fold. First, if the sum is zero, than a subset of the numbers does add to T and the function should return true. The other is if we have visited all the elements and sum is still not zero. In this case, we did not find a subset that summed to T, therefore we should return false. One optimization that can be made is if the element being considered is greater than sum, then the recursive function should only be called without this element. The pseudo code below illustrates this algorithm.

```
// Assuming arrays are indexed from 0.
isSubsetSum(S, n, sum){
  if(sum == 0){
    return True;
  }
  if(n == 0){
    return False;
  }
  n--;
  if(sum < set[n]){
    return isSubsetSum(set, n, sum);
  }
  return isSubsetSum(set, n, sumset[n]) or isSubSetSum(set, n, sum);
  }
}
```

This version of the algorithm only returns whether or not a valid $S\prime$ exists. It is trivial however to modify it to return the indicies of the elements of $S\prime$ in S.

4.3. **Subset Sum Example.** Let's test this on a small example. Let the inputs be $S = 1, 3, 6, 8, 15, 29$ and $T = 21$. For this example, $n = |S| = 6$. The first call to isSubsetSum will be isSubsetSum(S, n, T).

call	variable values	what happens
isSubsetSum(S, 6, 21)	$n = 6$, $sum = 21$	For the initial call, $sum \neq 0$ and $n \neq 0$, so the base cases are not met and n is decremented to 5. Next it checks if $S[n] > sum$. Since $S[n] = S[5] = 29$ and $sum = 21$, $S[n] > sum$, so it calls isSubsetSum(S, n, sum), which means it is not including 29 in $S\prime$.
isSubsetSum(S, 5, 21)	$n = 5$, $sum = 21$	Both sum and n are not zero, so the base cases are skipped and n is decremented to 4. This time $S[n] = S[4] = 15$and $sum = 21$. Since $S[n] \not> sum$, the algorithm will call isSubset-Sum(S, n, sum-S[n]) —— isSubsetSum(S, n, sum)
isSubsetSum(S, 4, 6)	$n = 5$, $sum = 6$	As before, the same thing happens with the base cases and n will be decremented to 3. $S[n] = S[3] = 8$ and $sum = 6$ so, as in the first call,$S[n] > sum$ so isSubsetSum(S, n, sum) is called.
isSubsetSum(S, 3, 6)	$n = 3$, $sum = 6$	Once again, the base cases are not met and n is decremented to 2. This time $S[n] = S[3] = 6$ and $sum = 6$, thus $S[n] \not> sum$ and the algorithm calls isSubsetSum(S, n, sum - S[n]) —— isSubsetSum(S, n, sum)
isSubsetSum(S, 2, 0)	$n = 2$, $sum = 0$	Finally, the first base case of $sum = 0$ is met. This means the algorithm knows that a subset does exist and will return True.
isSubsetSum(S, 3, 6)	$n = 3$, $sum = 6$	When the algorithm returns up to this call, it will have received true for the first recursive call, and thus does not need to make the second call since it already knows the result of the or operation is True. Thus, it now returns True as well.
isSubsetSum(S, 4, 6)	$n = 5$, $sum = 6$	This receives the True back from the call to isSubsetSum and returns True.
isSubsetSum(S, 5, 21)	$n = 5$, $sum = 21$	As with the first return, the or is known to be True so it returns True.
isSubsetSum(S, 6, 21)	$n = 6$, $sum = 21$	This receives the True back from the call to isSubsetSum and returns True. Since this was the initial call, the answer is now returned to the original caller as True, a subset does exist that sums to T.

We can verify that the algorithm returned a correct answer by providing an $S\prime$ that sums to T. If we had run the version of the algorithm that returned indices, it would have returned True and $[2, 4]$ as the indices. This means that $S\prime = 6, 15$. This gives us $sum(S\prime) = sum(6, 15) = 6 + 15 = 21$ which is the target T that we were looking for, thus, the algorithm returned the correct answer.

4.4. **The steps.** Now that we have defined the problem subset sum, we need to prove that it is NP-complete. The proof involves the usual two elements: (i) SS is an element of NP (ii) Some known NP-complete problem is reducible to SS For part (ii) we will use the known NP-complete problem of Vertex Cover and will show that $vertexcover \leq_p SS$.

5. Vertex Cover

5.1. **Description of Vertex Cover.** Say you are hosting a party at your house this coming weekend[3]. You have a wide social network, but being the über-friendly person that you are, you have friends from many different groups. In fact, many of your friends don't get along with each other at all. In trying to figure out your invitation list, you can represent each of your friends as nodes in a graph. Unlike regular social network graphs where edges represent friendships, you make a graph of rivalries and conflicts among your wide circle of friends. In your graph, each edge represents a rivalry between a pair of individuals. In order to invite as many friends as possible to your party, you want to make a list of your least agreeable friends – those who have the most rivalries – so you can be sure to dis-invite them. If these disagreeable, conflict-prone individuals aren't invited, you can invite most of your other friends without worrying about a fistfight breaking out over the punch bowl.

This is an example of the VERTEX COVER problem. However, as we have stated it, this is an optimization problem. As the host of your party, you want to invite as many friends as possible, without allowing any conflicts. In other words, you want to minimize your "Don't Invite" list. Because we cant put a value on the threshold $as - many - as - possible$, we need to change the phrasing of this problem into a question with a yes-or-no answer.

(As an interesting aside, the set of friends who will be at the party – total friends minus the "don't invite" list – forms what is called an independent set. Finding an INDEPENDENT SET is another NP-Complete problem. These problems, therefore, are easily reducible to each other: INDEPENDENT SET = V – VERTEX COVER.)

So, as a first step, we change this problem into a decision problem. Is there a subset S of size k of my friend circle V, where members of S are my particularly feisty friends, who I can exclude from my party so that there will be no conflicts at my party? In other words, is there a subset of size k of friends (nodes) who are involved in every single conflict in my circle of friends – i.e., such that the nodes in S touch every single edge E in the graph? Now

[3]Example taken from [1]

we have rephrased the VERTEX COVER problem such that it is a decision problem rather than an optimization problem. Let's look at the formal definition of VERTEX COVER.

5.2. **Formal definition of Vertex Cover.** A vertex cover of an undirected graph G = (V, E) is a subset of the vertices V' $\subseteq$ V such that for any edge (u,v) $\in$ E, then u $\in$ V' or v $\in$ V' or both[4].

In other words, a vertex in the vertex cover set is considered to "cover" every edge to which it is incident[5]. A vertex cover is a subset of all of the vertices of a graph such that every edge in the graph is "covered" or incident to at least one vertex in the vertex cover subset. The size of the vertex cover, then, is simply the number of vertices in the vertex cover subset.

5.3. **Vertex Cover as a decision problem.** The problem VERTEX COVER, stated as a decision problem, is to determine whether a given graph G has a vertex cover of size k. We write this as:

VERTEX-COVER = $\{\langle G, k\rangle :$ graph G has a vertex cover of size k$\}$.

So, the input to VERTEX COVER is a graph G and an integer k. The algorithm returns either a certificate/witness, or "no such vertex cover exists".

5.4. **Output and polynomial time confirmation for Vertex Cover.** The output from VERTEX COVER is a candidate list of vertices that comprise the vertex cover subset[6]. This candidate vertex cover V' can be checked in polynomial time.

The verification algorithm for V' first confirms that the number of vertices in V' is less than k. Then, for every edge (u,v) in G, the algorithm checks whether u $\in$ V' or v $\in$ V'. Therefore, this verification algorithm at worst has to scan through the whole list V' for every edge E. This verification algorithm, then, is O(VE). We can verify the correctness of V' in polynomial time. Therefore, Vertex Cover is in NP.

In fact, Vertex Cover is NP-Complete. Without covering this in detail, other NP-complete problems can be reduced to Vertex Cover. For a complete proof, see page 1090 of CLRS, which goes through the reduction from CLIQUE to VERTEX COVER. This proof shows that VERTEX COVER $\in$ NP-COMPLETE.

5.5. **Example problem for Vertex Cover.** Let's look at a toy example of Vertex Cover.
This simple graph has 5 nodes (labeled A through E) and 6 edges. So, the input to VERTEX COVER would be, then, the graph G and a threshold value k. Let's say that

[4]CLRS pg.1089

[5]In this case, the word "incident" describes a vertex that is on one end of the edge in question. A more intuitive word might be "adjacent," but "adjacency" has a formal definition in graph theory: it describes two vertices that are connected by an edge.

[6]CLRS pg. 1090

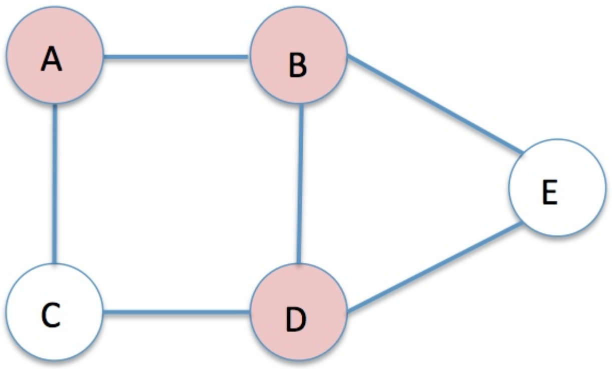


Figure 2. Example of Vertex Cover

k =3, and in this example, G = ((A,B,C,D,E), ((A,B),(A,C),(B,D),(B,E),(C,D), (D,E))). The algorithm for Vertex Cover brute-force searches through the exponential space of possible solutions until it finds a correct solution, V'. The algorithm then outputs 1 (yes, I found a solution), and the certificate w. To verify the correctness of this putative solution, we check that every edge in G is incident to at least one vertex in V'.

V' = (A, B, D). These nodes are colored pink in the figure.

We ask: is a vertex in edge (A,B) in V'? Yes, they both are.

Is a vertex in edge (A,C) in V'? Yes, A is.

Is a vertex in edge (B,D) in V'? Yes, they both are.

Is a vertex in edge (B,E) in V'? Yes, B is.

Is a vertex in edge (C,D) in V'? Yes, D is.

Is a vertex in edge (D,E) in V'? Yes, D is.

Therefore, we confirm that V' is a correct solution to VERTEX COVER. Also, we did the verification is polynomial time: for each edge, we check whether either of its endpoints were in V'. Stepping through every edge once and looking at every vertex in V', this verification procedure takes O(VE) time, so it is polynomial.

6. Poly-time reduction from Vertex Cover to Subset Sum

6.1. **Figuring out the reduction.** Now that we have presented VERTEX COVER as an NP-COMPLETE problem, we can use Vertex Cover to show that Subset Sum is also NP-Complete. To do so, we have to convert the input of Vertex Cover (G, k) into the correct input for Subset Sum (S, t) in polynomial time. In the flowchart, the boxes labeled "?" must be shown to be a polynomial time function, converting between inputs and outputs of Subset Sum and Vertex Cover[7].

[7]Reduction adapted from [6] and [3]

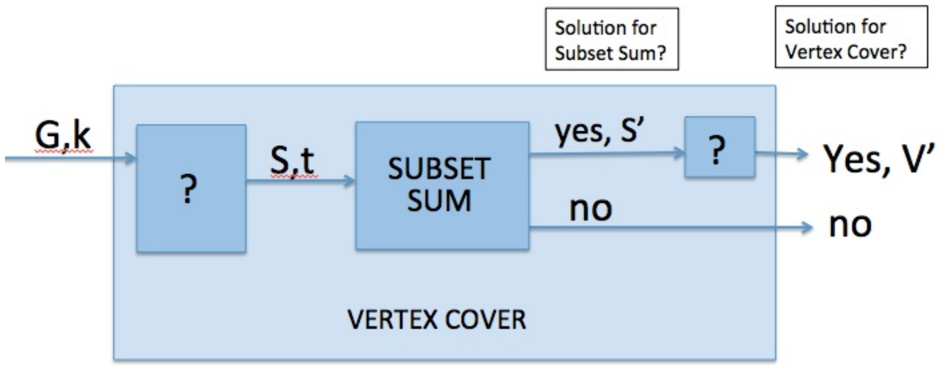


Figure 3. Flowchart of reduction from Vertex Cover to Subset Sum

Using this flowchart, our first step is to figure out how to map between (G,k) and (S,t). To help clarify this process, we will show the reduction on a very simple graph (see figure). We can easily see that there are many possible vertex covers for this simple graph. We use it simply for the purposes of illustration.

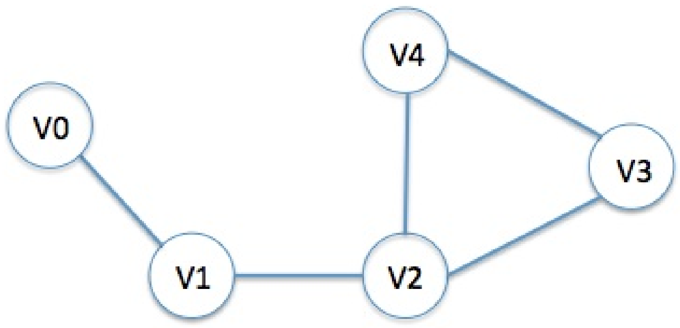


Figure 4. An example graph G, the input to Vertex Cover

As our first step, we make an educated guess that we will somehow convert the threshold from Vertex Cover (k) to the target for Subset Sum (t). Therefore, we need to figure out a mapping from graph G to set S, which consists of integers.

6.2. **Step 1: label edges and convert G into a matrix.** The input graph doesn't lend us many clues, so our first step is to try a different representation of the graph. We will represent it in a matrix. We label the edges in the graph e0 through e4. Then, we

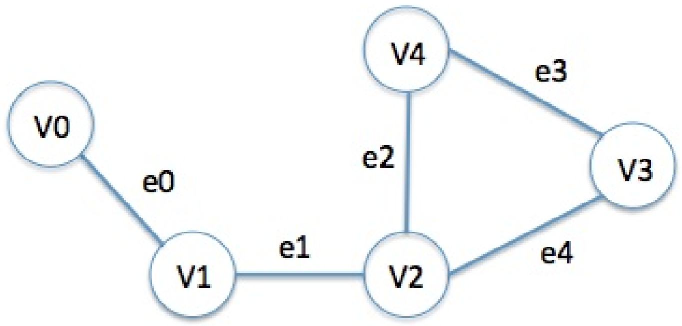
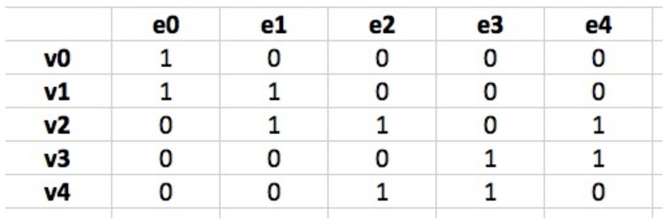


FIGURE 5. Reducing Vertex Cover to Subset Sum: labeling edges



	e0	e1	e2	e3	e4
v0	1	0	0	0	0
v1	1	1	0	0	0
v2	0	1	1	0	1
v3	0	0	0	1	1
v4	0	0	1	1	0

FIGURE 6. Step 1 of reduction: convert graph G into a matrix showing connections between vertices and edges

create and fill in a matrix that is |V| by |E|. In our matrix, we insert a "1" if e_i and v_j are incident; otherwise, we insert a "0". See the matrix in the figure for graph G.

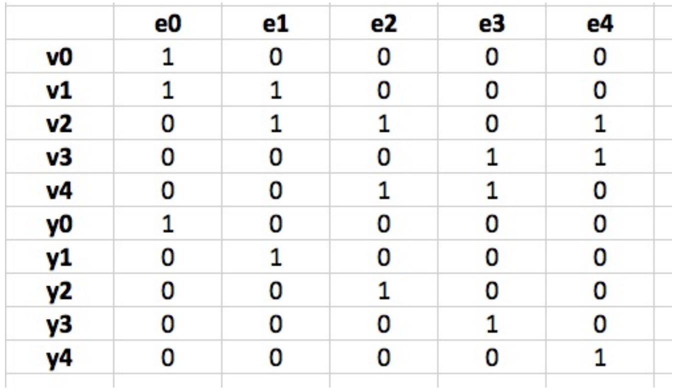
Right away, we notice two things. FIrst of all, reading across the rows, each row looks like an integer written in binary. Second, each column sums to at most 2, since each column represents a specific edge, and each edge can only connect two vertices. In a naive first stab, we try adding the first three rows of the matrix. This addition is: 10000 + 11000 + 01101 = 110101.

Right away a problem becomes apparent. Our binary addition has led us to carry a 1 from column e1 to column e0. Now we have a sum, 110101, with ambiguous zeros. The first zero represents a carry, while the second 0 in the sum is a "true zero" – i.e., the edge e3 is not covered by any of the vertices v0, v1, or v2.

To solve this problem, we do our addition in base 4. There is no particular reason that we need to do addition in base 2; we generally think about doing addition in base 10, then ask our computers to do a summation in base 2. We can easily translate between integers in different base representations. Now, when we sum the rows v0, v1, and v2, we get sum = 22101.

We are making progress. Now, there is only one kind of 0 in the sum – a true zero. Now, our sum (which we hope to map to a target to input to Subset Sum) contains 1's and 2's, which represent coverage by a vertex for that edge, and true 0's, which represent an edge that is not covered by any vertices.

However, the point of this reduction is to map an instance of Vertex Cover to an instance of Subset Sum. By the above definition, we could search for any number of vertex covers, with sums such as 12111, 22121, 11111, 21212, etc. These are all valid vertex covers. However, we only need to show that we can find one correct vertex cover. And, in fact, by the nature of Subset Sum, there needs to be one particular target value, t, that we feed to the algorithm. So, we need to find a target for Subset Sum that is consistent.



	e0	e1	e2	e3	e4
v0	1	0	0	0	0
v1	1	1	0	0	0
v2	0	1	1	0	1
v3	0	0	0	1	1
v4	0	0	1	1	0
y0	1	0	0	0	0
y1	0	1	0	0	0
y2	0	0	1	0	0
y3	0	0	0	1	0
y4	0	0	0	0	1

FIGURE 7. Step 2 of reduction: add slack values

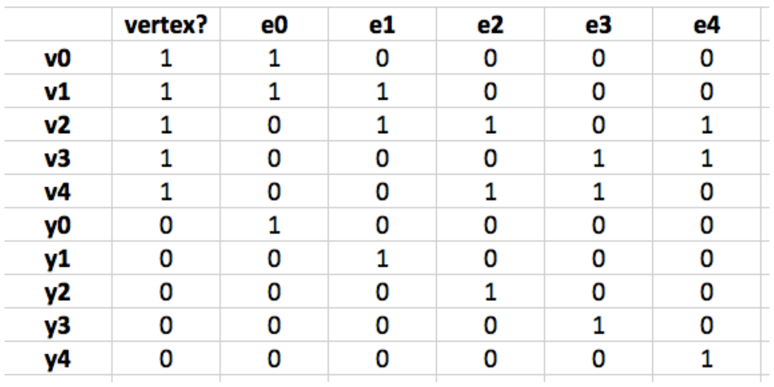
6.3. **Step 2: Add slack values to the matrix.** Slack values are a common gadget or widget used in reductions. They are what they sound like – slack values, that don't change the output for Vertex Cover, but allow us to provide a consistent target value to Subset Sum. Slack values consist of a row of all zero's except for a single 1. We name them y_i, and we add one slack value for every edge. The value for y_i is all zeros with a 1 in the i^{th} position. See the modified matrix, including slack values.

Previously, we had the sum of v0, v1, and v2 as 22101. Now, to get a consistent sum, we aim to get all 2's in columns where that edge is covered. We can therefore add v0 + v1 + v2 + y2 + y4. The sum is 22202. We are getting closer to our goal.

We still have not found a vertex cover that covers edge e3, as the 4th position in our sum is a true zero. If we tried to cover this value using a slack value, we might try adding y3 to our sum. However, 22202 + 00010 = 22212, and so we have shown that adding slack values doesn't impact the correctness of our reduction.

Now, we finally want to try to find a vertex cover that, in integer form, adds to all 2's. (Putative target = 22222). Since e3 isn't covered yet, we add another row to our summation such that that vertex does cover e3. How about v3? Now, we consider: v0+v1+v2+v3+y2+y3 = 22222. We are very close to the true solution now. We have found a true vertex cover for our graph G: v0, v1, v2 and v3. This is a correct vertex cover (which we can easily check in polynomial time, of course).

However, there is one last catch: we need to find a vertex cover that satisfies the final constraint k = 3. Our current S' output includes vertices 0,1,2, and 3, so has length 4. We have one final transformation to make to our matrix (and, thus, to our mapping from vertices to integers).



	vertex?	e0	e1	e2	e3	e4
v0	1	1	0	0	0	0
v1	1	1	1	0	0	0
v2	1	0	1	1	0	1
v3	1	0	0	0	1	1
v4	1	0	0	1	1	0
y0	0	1	0	0	0	0
y1	0	0	1	0	0	0
y2	0	0	0	1	0	0
y3	0	0	0	0	1	0
y4	0	0	0	0	0	1

FIGURE 8. Step 3 of reduction: add a column on the leftmost side of the matrix

6.4. **Step 3 of reduction: add a column on the leftmost side of the matrix.** To include the constraint from vertex cover that it must be size = k or smaller, we add a column at the left side of the matrix. The values in this column are all 1's, for the v_i rows, and all 0's for the y_i rows. Then, we set the target for subset sum to the integer k22...222, with as many 2's as there are edges. For the example, our threshold is k = 3, so the target for Subset Sum is 322222. This is the k integer, followed by five 2's. Now, we finally have a mapping from our Vertex Cover input (G, k) to the input for Subset Sum (S, t). In our example, reading each of the rows, the final input set S for Subset Sum is the following:

Set S = $\{110000, 111000, 101101, 100011, 100110, 010000, 001000, 000100, 000010, 000001\}$

The target t = 322222. We input (S, t) into the Subset Sum algorithm.

6.5. **Check correctness of Subset Sum.** Subset Sum returns either a 0 (no, there does not exist a subset S' that adds up to target t), or a candidate subset S'. In our example, one possible output of the Subset Sum algorithm would be the following:
S' = $\{v1, v2, v3, y0, y2, y3\}$

First, we check the correctness of S'.
111000 + 101101 + 100011 + 010000 + 000100 + 000010 = 322222. This answer is correct, and we have verified its correctness in polytime through simple addition.

6.6. **Convert S' to V'.** To convert S' to V', we simply take the labeled rows that are derived from vertices and have retained their v- prefixes. For the example, V' = v1, v2, and v3. The other elements of S' are slack variables; they are not related to the vertex cover of the original graph G.

6.7. **Check correctness of Vertex Cover.** To check V' for correctness, we have to check two things: first, that the cover is less than or equal to k, and then that every edge is covered. We quickly check the length of V', which is equal to 3 and therefore equal to the threshold k = 3.

Next, we check each of the edges to be sure that one of its two endpoints is in the vertex cover.
Is e0 covered? Yes, by v1.
Is e1 covered? Yes, by both v1 and v2.
Is e2 covered? Yes, by v2.
Is e3 covered? Yes by v3.
Is e4 covered? Yes, by both v2 and v3.
By checking each edge, we have confirmed in polynomial time that this proposed vertex cover V' is correct.

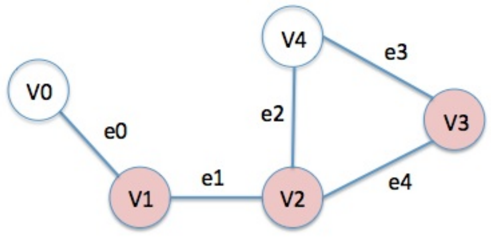


FIGURE 9. Vertex Cover V' output from Subset Sum

6.8. **Show that the reduction runs in polynomial time.** We started this reduction with the goal of showing that Subset Sum is in the class NP-Complete. We have showed that we have a reduction from Vertex Cover to Subset Sum that is correct. But, for the proof to be complete, we have to show that the reduction runs in polynomial time.

Here is the final reduction and its running time for each step: First, we converted graph G into a V by E matrix. Therefore, this step took O(VE). Second, we added slack variables to the matrix, one for each edge. Therefore, this step took O(E^2). Third, we added an extra column at the front of the matrix, for both v-rows and y-rows. This step took O(V+E). Fourth, we converted the output S' of Subset Sum to V'. This step took O(V) time.

At worst, for a very dense graph, E = V^2. Let's substitute this value in for E in the running time analysis:
Step one: O(V^3)
Step two: O(V^4)
Step three: O($V + V^2$)
Step four: O(V)
Sum = $O(V^4 + V^3 + V^2 + 2V) = O(V^4)$
By definition, $O(V^4)$ is in the form $O(n^c)$. Here, n = V and c = 4. Therefore, our reduction runs in polynomial time, and we can conclude that Subset Sum is an NP-Complete problem. It satisfies the two requirements: (1) It is a problem in NP, with witnesses that can be check in polynomial time, and (2) another NP-Complete problem reduces to it (Vertex Cover $\rightarrow$ Subset Sum).

References

[1] Chris Moore and Stephan Mertens, *The Nature of Computation.* Oxford University Press Oxford, UK, 1st Edition, 2012.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT, Cambridge, MA, 3rd Edition, 2009.

[3] David M. Mount *Lecture Notes for CMSC451: Design and Analysis of Computer Algorithms.* University of Maryland, College Park, MD
http://wwwisg.cs.uni-magdeburg.de/ag/lehre/WS0506/ThICV/notes/NPC_DavidMount.pdf Retrieved April 22, 2013. 2004.

[4] GeeksforGeeks.org *Dynamic Programming Set 25 (Subset Sum Problem).*
http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/ Retrieved April 22, 2013. December 24, 2012.

[5] Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani *Algorithms.* McGraw-Hill, New York, USA, 1st Edition, 2008.

[6] R. Chandrasekaran *Lecture Notes for 6363.* University of Texas-Dallas, Dallas, TX
http://www.utdallas.edu/ chandra/documents/6363/lec14.pdf Retrieved April 22, 2013.

[7] James Carlson, ed. *The Millennium Prize Problems.*
http://www.claymath.org/millennium/ Clay Mathematics Institute, Cambridge, MA, 2006.