1 Introduction

Turning away for the moment from graph-theoretic problems, I want to look at two problems involving numbers that we can show are **NP**-complete. These are:

SUBSET-SUM
Instance A sequence $v_1, v_2, \ldots v_n$ of positive integers and a target value t
Problem Is there a subset $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} v_i = t$?

FAIR-DIVISION
Instance A sequence $v_1, v_2, \ldots v_n$ of positive integers.
Problem Is there a subset $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} v_i = \sum_{j \notin I} v_j$?

2 Reduction of SUBSET-SUM to FAIR-DIVISION

Clearly, SUBSET-SUM is more general than FAIR-DIVISION. That is, if we could solve SUBSET-SUM efficiently, then it would be easy to solve FAIR-DIVISION efficiently simply by computing the sum, S, of all the v's, answering "no" if S is odd, and otherwise delegating to SUBSET-SUM with $t = S/2$.

The reduction from SUBSET-SUM to FAIR-DIVISION is a little more interesting. Suppose we have an instance of SUBSET-SUM with target t and values v_1, v_2, ..., v_n. Obviously, unless $0 < t < S$ we don't have any serious difficulties, so suppose that to be the case. Create an instance of FAIR-DIVISION with values:

$$v_1,\ v_2,\ \ldots\ ,\ v_n,\ 3S - t,\ 2S + t.$$

Suppose that this has a fair division. The total sum is $6S$ so the two new parts would have to be on opposite sides of any fair division (together they add to $5S$ and there's only S left over). But then if there's a fair division, the original values that go along with $3S - t$ must add up to t, i.e., the problem was a positive instance for SUBSET-SUM. On the other hand, if the original problem was a positive instance for SUBSET-SUM, then the new one is a positive instance for FAIR-DIVISION using the division that produces t from among the v's, and then adds the part of value $3S - t$.

Is S perhaps too big to be computed in polynomial time (or space) from the v's (i.e., why do we know this is a polynomial time reduction?) The original problem requires nN space where N is the maximum number of bits required to represent a v. The space required to represent S is certainly less than nN (more like $\log_2 n + N$) and the time required for n additions of numbers of size at most N is also bounded (more or less) by $n(\log_2 n + N)$ so that's all ok (in fact sub-quadratic in the original input size).

3 But isn't SUBSET-SUM easy?

There's a standard dynamic programming approach to solving SUBSET-SUM:

```
sums ← {0}
for i from 1 to n do
  for s in sums do
    if s + v_i < t then
      add s + v_i to sums (if necessary)
    else if s + v_i = t then
      return t
    end if
  end for
end for
return f
```

All the values in $sums$ are less than t, so we could just represent $sums$ as an array of booleans of size t. Therefore the complexity of this algorithm is $O(tn)$. Consequently, if the target is polynomial in the number of items, this is a polynomial-time algorithm.

However, imagine that our values are something like n, n-bit integers. Then our target, t could also be an n bit integer (or thereabouts) – but that means it could be as large as 2^n. So the complexity bound would be $O(n2^n)$ for that situation, and that's not polynomial in n^2 which would be the total size of the input.

A corollary to this observation is that if we're going to look for difficult instance of SUBSET-SUM, or, what amounts to the same thing a reduction from 3-SAT to SUBSET-SUM, we're going to need to use values that have lots of bits.

4 SUBSET-SUM is NP-complete

It remains to produce a polynomial-time reduction from 3-SAT to SUBSET-SUM. For this reduction it's convenient to imagine that we write our numbers in base 4 (really, any base larger than 3 will do – and if you do insist on binary representation, then just replace every "digit" in what follows by the appropriate pair of digits).

Suppose that our instance of 3-SAT involves k variables x_1 through x_k and c clauses C_1 through C_c. The "values" that make up our corresponding instance of SUBSET-SUM will all have $k + c$ digits. Call the leading k digits the *variable* digits, and the trailing c digits the *clause* digits.

The total number of values we use for our instance of SUBSET-SUM will be $2k + 2c$. So there will be two values associated with each variable, and two values associated with each clause (the latter ones are mostly used for book keeping).

The two values associate to each variable, x_i, will correspond to the variable and its negation. They will each have a digit 1 in the variable digit corresponding to x_i (i.e., the i^{th} digit) and 0s in the other variable digits. In the clause digits, if a literal occurs in a clause there will be a 1 digit there, while if it does not, there will be a 0 digit.

Consider our usual example (indexed in the obvious way)

$$x \vee y \vee z$$
$$\neg x \vee \neg y \vee z$$
$$\neg x \vee y \vee \neg z.$$

So far we would have the following

	var			$clause$		
x	1	0	0	1	0	0
$\neg x$	1	0	0	0	1	1
y	0	1	0	1	0	1
$\neg y$	0	1	0	0	1	0
z	0	0	1	1	1	0
$\neg z$	0	0	1	0	0	1

What should our target be? To ensure that each variable gets a truth value we just set the value of the target in the "variable" digits to be 1. We'd like to see non-zero values in the clause digits (since this would represent a satisfying assignment – at least one of the corresponding literals is true). However, we need a single target value, not a range of possibilities. That's where the extra book keeping digits come in. In each clause column we add two spare 1's and ask for the column sum to be 3. That way, we can always get a 3 if the clause is satisfied (if exactly one variable is true we use both the spare entries, if two are true we just use one and if three are true we don't bother – further we can't generate both a sum of 3 and a carry, so we can't interfere with other clauses), while if the clause is not satisfied, the most we can get is 2. So the complete set-up for this set of variables and clauses is:

	var			$clause$		
x	1	0	0	1	0	0
$\neg x$	1	0	0	0	1	1
y	0	1	0	1	0	1
$\neg y$	0	1	0	0	1	0
z	0	0	1	1	1	0
$\neg z$	0	0	1	0	0	1
c_{11}	0	0	0	1	0	0
c_{12}	0	0	0	1	0	0
c_{21}	0	0	0	0	1	0
c_{22}	0	0	0	0	1	0
c_{31}	0	0	0	0	0	1
c_{32}	0	0	0	0	0	1
t	1	1	1	3	3	3

And that's exactly what we do in the general case as well. If the instance of SUBSET-SUM we produce is positive, then the solution must use exactly one literal per variable (because of the variable columns) and the corresponding truth assignment must satisfy all the clauses (because of the clause columns). Conversely, if we have a satisfying assignment of truth-variables, it translates directly into an appropriate subset of values to satisfy the associated SUBSET-SUM instance.

5 Decision versus search

Most of the **NP**-complete problems we've been talking about as decision problems have a corresponding search or construction problem. That is, instead of asking "Does a graph have an independent set of size k" as we do in INDEPENDENT-SET we might well ask "Construct an independent set of size k if one exists" or "Construct an independent set of the largest possible size".

Similarly, in SUBSET-SUM we might actually want to find a subset of the values with the given target sum rather than just know that one exists.

Fortunately, some obvious search techniques will usually show that if we can solve the decision version then, with polynomial additional overhead we can solve the construction version. For instance:

MAXIMUM INDEPENDENT SET
Instance: A graph G
Problem: Determine an independent subset I of G of maximum possible size.

> **Aside**: If all we want is an independent set that can't be further extended, i.e., a *maximal* independent set, then a trivial greedy algorithm works: choose a vertex, delete all its neighbours, choose a vertex, ...

Now imagine we have a black box (i.e., method, subroutine) for the original

INDEPENDENT-SET
Instance: A graph G and a positive integer k
Problem: Does G have an independent set of size k?

Let's first use calls to this routine to determine the size of the maximum independent set. Let n be the number of vertices of G. That's certainly an upper bound! So call INDEPENDENT-SET with $k = n/2$. According to the answer, call it with $k = 3n/4$ (if yes) or $k = n/4$ (if no). That is, do binary search. After $\log_2 n$ iterations or thereabouts you'll have a value of k for which the answer is yes, but such that the answer for $k + 1$ is no which is therefore the size of a maximum independent set in G.

Now we know the size of a maximum independent set. How can we construct one?

Choose a vertex v of G. Delete it and its neighbours. Ask INDEPENDENT-SET if the remaining graph has an independent set of size $k-1$. If yes, we've found our first vertex (and we'll find $k-1$ more in what's left). If no, then we can't use v in a maximum independent set, so just delete v and proceed.

We'll make at most n calls to INDEPENDENT-SET in this phase, all on graphs having the same number as, or fewer edges than G, so if we could resolve INDEPENDENT-SET efficiently, we'll have build our maximum independent set efficiently as well.

Similar tricks work in almost all instances. For instance, consider DIRECTED HAMILTON CYCLE and the problem of actually generating a cycle. Choose an edge $v \to w$ and delete it. If you still have a Hamilton cycle carry on. If not, you must use this edge in every cycle. So, make a new graph where you fuse its endpoints into a single new vertex vw and delete all outgoing edges from v (since you know your plan is to "leave" v via w) and all incoming edges to w (since you plan to arrive at w from v).

The undirected case for HAMILTON CYCLE is also interesting. Start with your graph G. Make sure it has a Hamilton cycle (if it doesn't you're done). Choose any edge $\{v, w\}$ and delete it. Check for Hamiltonicity again. If you're still ok, just carry on with the new graph. If not, then every Hamilton cycle of the original graph must use that edge (in either direction). Delete it, add a new vertex vw and edges $\{v, vw\}$ and $\{w, vw\}$. This graph now has a Hamilton cycle and because vw has only the two neighbours v and w that cycle must go $\ldots v$, vw, w $\ldots$or in reverse. Now proceed to another edge and continue. Eventually, after having added at most a number of new vertices equal to the length of the original cycle you'll have produced a genuine Hamilton cycle of the modified graph from which you can read off one in the original. Since the overhead is polynomial, we're happy.

What about SUBSET-SUM? That's even easier. Given a positive instance $(v_1, v_2, \ldots, v_n, t)$ just ask about the instance $(v_2, v_3, \ldots v_n, t - v_1)$. If the answer is still "yes" then you can hit the target using v_1 and you should proceed to solve the modified problem. Otherwise, you can't use v_1 so the instance $(v_2, v_3, \ldots v_n, t)$ must be positive and you can carry one again.

6 Problems

1. How do you search for a satisfying assignment of a 3-SAT instance if you have a $(\leqslant 3)$-SAT black box?
2. How do you search for a satisfying assignment of a 2-SAT instance if you have a 3-SAT black box? The difference is the requirement that 3-SAT clauses contain exactly three literals rather than 2.
3. How can you find a 3-colouring of a graph given a black box for the 3-colouring decision problem?
4. Consider a clause of three distinct literals. If we choose a random truth assignment for each variable independently (and with a $1/2$ chance of either t or f), then what is the probability, p, that we will satisfy that particular clause?
5. (*) Given the previous problem, if we have k clauses there must be a truth assignment that satisfies at least kp of them (because this is the average number satisfied by any given assignment). Show that there is a polynomial-time deterministic algorithm that determines such an assignment.