COSC 341
Theory of Computing
Lecture 22
$\mathbf{NP}$-complete problems involving numbers: (FAIR-DIVISION and SUBSET-SUM)

Stephen Cranefield
stephen.cranefield@otago.ac.nz

Keywords: 3-SAT, FAIR-DIVISION and SUBSET-SUM

Problem definitions

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 $\sum_{i \in I} v_i = \sum_{j \notin I} v_j$?

Is there a reduction from one to the other?

- If we could solve SUBSET-SUM efficiently, then it would be easy to solve FAIR-DIVISION efficiently. We can construct a Turing machine that computes the sum, S, of all the v's, answers "no" if S is odd, and otherwise runs SUBSET-SUM with input $v_1, v_2, \ldots v_n$ and $t = S/2$.
- However, we can also reduce SUBSET-SUM to FAIR-DIVISION!

Reduction of SUBSET-SUM to FAIR-DIVISION

A SUBSET-SUM instance $v_1, v_2, \ldots, v_n$ with target t is trivially positive if $t = 0$ or $t = S$ where $S = \sum_i v_i$. That can be reduced to any positive FAIR-DIVISION instance, e.g. $1, 1$. If $0 < t < S$, consider this FAIR-DIVISION instance:

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

- Suppose this has a fair division. The total sum is $6S$ so the two new parts must be in different partitions (they add to $5S$ and there's only S left over). But then, one partition contains vs and $3S - t$ and sums to $3S$, so $S = t$, which is positive for SUBSET-SUM.
- On the other hand, if the original SUBSET-SUM instance was positive 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 $3S - t$.

See Notes 19 for the argument for why this is a polynomial-time reduction.

Complexity of SUBSET-SUM

Is SUBSET-SUM easy? Consider this dynamic programming algorithm from Wikipedia (lightly edited):

> Suppose we have the following sequence of elements in an instance: $x_1, \ldots, x_n$.
>
> We define a *state* as a pair (i, s) of integers. This represents the fact that:
>
> *there is a nonempty subset of $x_1, \ldots, x_i$ that sums to s*
>
> Each state (i, s) has two *next* states:
> - $(i + 1, s)$, implying that x_{i+1} is not included in the subset;
> - $(i + 1, s + x_{i+1})$, implying that x_{i+1} is included in the subset.
>
> Starting from the initial state $(0, 0)$, it is possible to use any graph search algorithm (e.g. breadth-first search) to search for the state (n, t) where t is the target value for the sum. If the state is found, then by backtracking we can find a subset with a sum of exactly t.

The run-time of this algorithm is at most linear in the number of states. The number of states is at most n times the number of different possible sums. . . . if all input values are positive and bounded by some constant c, then the sum of the x_i is at most nc, so the time required is $O(n^2c)$.

Why does this not count as a polynomial-time algorithm? If the inputs are in binary (at least) and are n-bit integers, then the target t could be as large as 2^n, giving time complexity $O(n2^n)$, which is not polynomial in the input size n^2.

SUBSET-SUM is NP-complete (reduction from 3-SAT)

Suppose the 3-SAT instance has k variables x_1 to x_k and c clauses C_1 to C_c. We reduce this to a SUBSET-SUM instance with numbers written in base 4 (this is a convenience: any base larger than 3 will work).

Each number will have k *variable* digits followed by c *clause* digits. Each variable x_i is mapped to two numbers: one recording which clauses contain x_i and another recording which clauses contain $\neg x_i$. For example:

$$\begin{aligned} x \vee y \vee z \\ \neg x \vee \neg y \vee z \\ \neg x \vee y \vee \neg z. \end{aligned}$$

	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

The SUBSET-SUM target value has a 1 for each variable digit to ensure each variable has exactly one truth value:

t	1	1	1	?	?	?

What about the target's *clause* digits?

SUBSET-SUM is NP-complete (reduction from 3-SAT, continued)

Each clause has three literals and must be satisfied by between 1 and 3 of them. We can set the clause digits for the target to be 3 and add to our SUBSET-SUM instance two extra numbers for each clause, each with the corresponding clause variable set to 1 (and other digits 0):

	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

- A satisfying assignment must have $1 \leq n \leq 3$ literals true for each clause C_i.
- It will map to a SUBSET-SUM subset that uses $n - 1$ of the numbers corresponding to C_i.
- Note that it is impossible for any sum to have a 3 for some digit and also carry a 1 to the previous digit, so columns are independent of each other.
- Convince yourself that this reduction preserves positive and negative instances and is bounded by a polynomial function of the input size.

Do search version of decision problems have the same complexity?

Recall the INDEPENDENT-SET decision problem:

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

Consider this "search" version of the problem:

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

Note: This is not the same as finding a *maximal* independent set: one that can't be further extended. A trivial greedy algorithm can find one of those: choose a vertex, delete all its neighbours, choose a vertex, ...

Is the time-complexity of this the same as INDEPENDENT-SET?

Using INDEPENDENT-SET to solve MAXIMUM-INDEPENDENT-SET

1. Perform a binary search between 1 and $n =$ the number of vertices of G, to find the maximum k for which INDEPENDENT-SET succeeds.
2. Construct a maximum independent set of that size:

   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 almost always work for other decision problems. See Notes 19 for details.