NP-Completeness

Subhash Suri

May 22, 2018

1 Computational Intractability

- The classical reference for this topic is the book *Computers and Intractability: A guide to the theory of NP-Completeness* by Michael Garey and David Johnson, 1979.
- Nearly all problems discussed so far can be solved by algorithms with worst-case time complexity of the form $O(n^k)$, for some fixed value of k, which is independent of the input size n.
- Typically k is small, say, 2 or 3. Examples include *shortest paths (SP), minimum spanning trees (MST), sorting, LCS, matrix chain multiplication etc.*
- These algorithms are considered "efficient" because their performance degrades gently (*polynomially*) as the problem size grows. Problems solvable in times of the form $O(n^k)$ are, therefore (and naturally), *polynomially-solvable*, and called *tractable.*
- We use P to denote this class of problems.
- To better appreciate the *special-ness* of these efficient algorithms, consider the alternative. In all these problems, we are searching for a solution (path, tree, subsequence, sorted order) from among an *exponential size* list of possibilities.
- A graph will typically contain exponentially many paths, and trees; a sequence will have exponentially many subsequences; number of matrix multiplication orders is also exponential. All of these problem could in principle be solved by checking through all the possible solutions, and finding the best. But such an algorithm will in practice be useless.
- The insights and techniques that led to efficient shortest path, MST, LCS algorithms are among the most notable successes of our quest for computational breakthroughs that defeat the specter of exponentiality.

- *Does this mean that no problem is too big if we put our collective minds to it?* If we think hard enough, an efficient (polynomial) algorithm will ultimately emerge? Unfortunately, such an optimism is most likely unwarranted.
- We now discuss *problems where we have failed to find efficient algorithms*, and it appears that *no such method may be possible!*
- Problems that require *exponential or worse time* are called "intractable," thus associating the label "tractable" with polynomial-time solvability.
- First, *why do we define tractability as polynomial time solvability?* We can defend this choice on multiple grounds:
  1. **Universality:** polynomial solvability holds across a wide spectrum of computational models: Turing machines, random access machines, most modern programming languages.
  2. Polynomial functions are **closed under composition:** if an algorithm calls another polynomial subroutine, it remains in the class P.
  3. Polynomial time is a good **proxy for practically** feasible: as the problem size grows, the running time worsens, but smoothly (polynomially) not abruptly the way exponential explosion occurs.
  4. **Problem Structure:** Achieving polynomial time inevitably entails discovering some special structure in the problem that allows one to find the optimum without enumerating all possibilities.

2 A Special Class of Problems: NP-Complete

- NP-Complete is a special class of intractable problems. This class contains a very diverse set of problems with the following intriguing properties:
  1. We only know how to solve these problems in exponential time e.g. $O(2^{O(n^k)})$.
  2. If we can solve **any** NP-Complete problem in polynomial time, then we will be able to solve **all** NP-Complete problems in poly time.
- The name NP stands for *Non-deterministic Polynomial Time*, and not what one may easily infer "not polynomial."
- Draw a picture of P, NP, NPC class hierarchy.
- If we recognize problem is NP-Complete, then we 3 choices:

1. be prepared for any algorithm to take exp time in worst-case;
2. settle for an approximate solution, instead of optimal.
3. alter the problem, or constrain it further, so it becomes tractable.

- "Is $P = NP$?" is the most famous computer science open problem. There is even a 1 million dollar (Millenimum) prize for it. The general belief is that $P \neq NP$ but there are good arguments on both sides.
- The intractability of some problems is also basis for theoretically secure crypto-systems.
- **Informal definition of NP-Completeness.** We will introduce the formal definition later but let us begin with an informal (intuitive) definition of a larger class called NP: *these are problems whose solutions have a "polynomially checkable" solution.* That is, if someone handed you the solution, you can verify it in poly time. For instance,

  1. does a graph G have a simple (loopless) path of length K?
  2. Is a number composite?
  3. Does a graph have a vertex cover of size C?

- Clearly, all problems in P are also in NP, so $P \subseteq NP$.
- NP-Complete problems are in a formal sense the *hardest* problems in NP—if any one of them can be solved in poly time, then they can all be solved in poly time; this would result in the set equality $P = NP$.
- Similarly, if any one of the NP-Complete problem can be shown to require exponential time, then they all will be exp.
- NP is not the hardest class of all computational problems—there are harder ones. For instance,

  1. **PSPACE:** class of problems that can be solved using only a polynomial amount of space. (Thus, no regard for the time; exp algorithms often take only poly space.)
  2. **EXPTIME:** Problems that can be solved in exp time. By contrast, a running time of the form $n^{\log n}$ is not exponential, but neither is it polynomial. It may be surprising that there are problems outside EXPTIME too!
  3. **UNDECIDABLE:** Problems for which no algorithm exists, no matter how much time is allowed. The Halting problem is equivalent to writing a "compiler" to check whether a program has the potential to go into infinite loop.

2.1 Decision Problems and Reducibility

- While in CS, we often deal with optimization problems, it will be more convenient to discuss decision problems, that have YES/NO answers. For instance, consider the TSP problem in a graph with non-neg edge weights.

  Optimization: What is the minimum cost cycle that visits each node of G exactly once?

  Decision: Given an integer K, is there a cycle of length at most K visiting each node exactly once?

- Decision problems seems easier than optimization: *given optimization solution, decision is easy to solve.* Since we are interested in hardness, showing that decision problem is hard will also imply hardness for the optimization. In general, for most problems, decision complexity is in the same class as optim.

- The main technical tool for proving a problem NP-Complete is *reduction.*

  Reduction from problem A to problem B is a **poly-time algorithm** R that transforms inputs of A to equivalent inputs to B. More precisely, given an input $x \in A$, the algorithm R produces an input $R(x) \in B$ such that "x is a YES instance for A" *if and only if* "$R(x)$ is a YES input for B."

- Mathematically, we say that R is a *reduction from A to B* iff, for all x, the equality $A(x) = B(R(x))$ holds.

`Picture:`

- When such a reduction exits, we write $A \leq B$, suggesting that (give or take a polynomial), A is no harder than B.

- [**Important Implication:**]

$$\begin{aligned} \text{If } B \text{ is known to be easy} &\quad\Rightarrow\quad A \text{ is easy too.} \\ \text{If } A \text{ is known to be hard} &\quad\Rightarrow\quad B \text{ must be hard too.} \end{aligned}$$

- The second implication is the one we use in NP-Completeness reductions.

2.2 Sample NP-Complete Problems

To show how similar they can be to P problems, we show them in pairs.

- **MST:** given a weighted graph and integer K, is there a **tree** of weight $\leq K$ connecting all the nodes.
- **TSP:** given a weighted graph and integer K, is there a (simple) **cycle** of weight $\leq K$ connecting all the nodes.

- **Euler Tour:** given a directed graph, is there a closed path visiting every **edge** exactly once?
- **Hamilton Tour:** given a directed graph, is there a closed path visiting every **node** exactly once?
- **Circuit Value:** given a boolean circuit, and 0/1 values for inputs, is the output TRUE, *i.e.*1?
- **Circuit SAT:** given a Boolean circuit, is there a 0/1 setting of inputs for which the output is 1?
- Boolean satisfiability is a central problems in theory of NP-Complete. General circuits are difficult to reason about, so we consider them in a standard form: CNF (conjunctive normal form).
  - Let $x_1, x_2, \ldots, x_n$ be input booleans, and let x_{out} be the output boolean variable. Then a boolean expression for x_{out} in terms of x_i's is in CNF if it is the AND of a set of clauses, each of which is an OR of some subset of literals; input vars and their negations. Example:

$$x_{out} \;=\; (x_1 \vee x_2 \vee \;!x_3) \wedge (x_3 \vee \;!x_1 \vee \;!x_2) \wedge (x_1)$$

  - Such as formula has an obvious translation to circuits, with one gate per logical operation.
  - An expression is 2-CNF if each clause has 2 literals; and 3-CNF if each has 3 literals. Examples.

$$(x_1 \vee\ !x_1) \wedge (x_3 \vee x_2) \wedge (!x_1 \vee x_3) \text{ is 2-CNF}$$

$$(x_1 \vee\ !x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee\ !x_1) \wedge (!x_1 \vee x_3 \vee\ !x_2) \text{ is 3-CNF}$$

  - Boolean satisfiability has many applications:
    1. Model-checking,
    2. Design Debugging
    3. Circuit delay computation
    4. functional dependence
    5. AI planning
    6. Inference in bioinformatics
    7. Knowledge-compilation,
    8. Software testing
    9. Package management in software distributions
    10. Checking of pedigree consistency
    11. test-pattern generation in digital systems
    12. Crosstalk noise prediction in integrated circuits
    13. termination analysis in term-rewrite system

- **2-CNF:** given a boolean formula in 2-CNF, is there a satisfying assignment?
- **3-CNF:** given a boolean formula in 3-CNF, is there a satisfying assignment?
- **Matching:** given a boys-girls compatibility graph, is there a complete matching?
- **3D Matching:** given a boys-girls-houses tri-partite graph, is there a complete matching; i.e. set of disjoint boy-girl-house triangles?

3 How to prove NP-Completeness

- Class NP: problems whose YES instances have a polynomial-size and polynomial-time checkable solution. (Clearly every problem in P is also in NP.)

- Not all problems have polynomially-checkable solutions. Take, for instance, the Halting Problem, or generalized Chess.
- A problem A is NP-complete if $A \in NP$ and **every** problem $X \in NP$ is reducible to A.
- **Lemma:** If A is NP-Complete, then A is in P iff $P = NP$.
- **Circuit SAT and NP-Completeness.** To get the theory going, we need to establish at least one NP-Complete problem.
- Circuit-SAT is such a problem. It is clearly in NP: given an instance C, and a *satisfying* setting of booleans $x_1, x_2, \ldots, x_n$, we just simulate the circuit, and check if $x_{out} = 1$.
- The hard part is show that all other problems in NP reduces to CSAT. This is the famous Cook's Theorem (1971), and a difficult result. We will give an informal proof sketch later, but for now let us assume CSAT is hard and use that to prove intractability of other problems. (Karp 1970s).
- Overall plan:
  1. Circuit SAT $\leq$ Boolean SAT
  2. Boolean $SAT \ \leq 3SAT$
  3. $3SAT \leq$ Max Independent Set (MIS)
  4. $MIS \leq CLIQUE$
  5. $MIS \leq$ Vertex Cover
  6. $3SAT \leq$ Hamiltonian Cycle
  7. $3SAT \leq$ Subset SUM
- We assume for now that Circuit-SAT is NP-complete.
- Reduction of Circuit-SAT to Boolean SAT is easy: a digital circuit can be expressed as a formula whose clauses express the function of each logic gate.

3.1 Reducing Boolean SAT to $3SAT$

- We take an instance of SAT, and convert into a poly-size instance of $3SAT$, as follows.
  1. Leave alone any clause with exactly 3 literals. For the rest:

2. If a clause has exactly one literal, as (x), then replace it:

$$(x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee\ !y_2) \wedge (x \vee\ !y_1 \vee y_2) \wedge (x \vee\ !y_1 \vee\ !y_2)$$

   where y_1, y_2 are 2 new variables introduced just for this clause.

3. If clause has length 2, such as $x_1 \vee x_2$, we do

$$(x_1 \vee x_1 \vee y) \wedge (x_1 \vee x_2 \vee !y)$$

4. If clause has 4 or more literals such as $(x_1 \vee x_2 \vee \cdots \vee x_k)$:

$$(x_1 \vee x_2 \vee y_1) \wedge (!y_1 \vee x_3 \vee y_2) \wedge (!y_2 \vee x_4 \vee y_3) \cdots$$

3.2 Reducing Boolean $3SAT$ to MIS

- Given an undirected, unweighted graph $G = (V, E)$, an *Independent Set* is a subset $I \subseteq V$ in which no two vertices are adjacent.

  (A classical problem, modeling conflicts among tasks, with many applications.)

- **Decision version:** Given a graph G, and an integer k, does G have an IS of size at least k?

- Membership in NP is easy; just check that no two vertices of I have an edge between them.

- To show NP-Completeness, we reduce $3SAT$ to MIS.

  1. Starting with a formula of n vars $x_1, \cdots, x_n$, and m clauses, construct a graph with $3m$ vertices. This graph has an IS of size m if and only if the $3SAT$ formula is satisfiable.
  2. Construction: The graph G has a triangle for every clause.
  3. The vertices in the triangle correspond to the 3 literals in that clause. Vertices in different clauses are joined by an edge **iff** those vertices correspond to literals that are *negations of each other.*
  4. EXAMPLE: (from KT)

$$(!x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee\ !x_2 \vee x_3) \wedge (!x_1 \vee x_2 \vee x_4)$$

5. **Proof of Correctness.** Assuming that the formula is satisfiable, we show that IS of size m must exist. Conversely, if IS of size m exists, we show formula can be satisfied.
6. Suppose formula is satisfiable. Thus, in each clause, at least one literal is satisfied. Construct the IS as follows: pick exactly one vertex corresponding to a satisfied literal in each clause (break ties arbitrarily).

   No two such vertices can have an edge between them—within each triangle (clause), we choose only vertex, and across triangles each edge joins literals that are negations of each other, so our construction will not pick both. So, the resulting set of vertices is an IS and it has exactly m vertices.
7. Conversely, suppose we have an IS with m vertices. We can have only one vertex in each triangle, since two are always adjacent. We make the variable assignment so that these chosen literals are true. This will consistently satisfy all the clauses.
8. **End of proof.**

3.3 Reducing MIS to $CLIQUE$

- Given a graph G (undirected, unweighted), a CLIQUE is a set of vertices K such that every pair is adjacent.
- **Decision Version:** Given G and an integer k, does G contain a CLIQUE of size at least k?
- Membership in NP is easy. For the NP-Completeness, we actually reduce the MIS problem to CLIQUE.
- Take an instance (G, k) of MIS. Construct the complement graph G' which has the same vertex set as G, but there is an edge (u, v) in G' precisely when (u, v) is not an edge in G.

- The reduction clearly takes polynomial time.
- Now an IS in G' must be a clique in G and every clique in G is an IS in G'.
- Thus, G' has a k-clique if and only if G has a k-vertex IS.

3.4 Reducing MIS to Vertex Cover

- Given a graph G, a vertex cover C is a subset of vertices such that every edge (u, v) in G has either u or v (or both) in C. (Set of vertices that cover all the edges.)
- In the MIN VC problem, we want to find the cover of smallest size. In the Decision Version, we want to decide if G has a vertex cover of size at most k.
- Membership in NP is straightforward; just check if all edges have at least one endpoint in the cover.
- For the NP-Completeness, we show reduction from MIS.
- **Lemma.** Suppose I is an Independent Set in $G = (V, E)$. Then, the remaining vertices $(V - I)$ form a vertex cover in G. Conversely, if C is a VC in G, then $(V - C)$ is an IS of G.
- **Proof.** Let I be an IS of G, and let $C = V - I$. For the sake of contradiction suppose C fails to be a vertex cover. Then there is at least one edge (u, v) for which neither u nor v is in C. This means that both u and v are in $I = V - C$, but that's a contradiction because u and v are not independent.

  Conversely, suppose $I = (V - C)$ is not an independent set. Then there is some edge (u, v) for which both u and v are in I. But then $C = V - I$ is not a vertex cover; it doesn't cover (u, v).
- Thus G has a MIS of size k if and only if G has a VC of size $n - k$.

4 Cook's Theorem: The Seed Problem

- Back to Cook's proof. Instead of Boolean SAT, we prove hardness of Circuit-SAT (CSAT). Conversion between Boolean SAT and Circuit SAT is easy.
- The intuitive idea is the following: Any problem $X \in NP$ has the property that a YES answer can be checked in poly time. This can be transformed into a circuit, whose input variables are the solution bits and the output bit = 1 iff the input forms a solution. The circuit has polynomial size.

- The Circuit-SAT is able to decide *if there is* **any** *setting of the input variables that makes the output* $= 1$, which means that Circuit-SAT can solve the problem X using this circuit, under a poly reduction. Now a more technical sketch.

- To show that every NP problem reduces to $CSAT$, we reason as follows. Take an arbitrary problem A in NP. There is a poly-time algorithm (verifier) V_A and a polynomial p_A such that $A(x) =$ YES if and only if there is an (output solution) y, with $length(y) \leq p_A(length(x))$, such that $V_A(x, y)$ outputs YES. (That is, the YES answer y can be written in polynomial length and verified in polynomial time by V_A.)

- Our reduction now works as follows. Suppose we have an input x of length n for A. We construct a circuit C that mimics V_A. We feed this circuit the input x (say, the graph for HAM), and then give it free variables for y (the answer bits).

- Since y has length at most $p(n)$, the construction takes polynomial time.

- We now just check if this circuit is satisfiable, which happens only if there is some setting of y for which circuit output is YES.

- Take the example of Hamilton cycle. Given a graph G, with n vertices and m edges, we build a circuit that outputs 1 iff G is Hamiltonian.

- How? Well, there is a computer program that checks in polynomial time if a given sequence of edges is answer to the HAM problem. So, there is a polynomial size circuit that can do the same. Then, we *hard wire* G into the circuit.

- There are a bunch of technical details and fine points, but this is the general idea.

- As one small detail, for instance, we have the following claim.

    If a verifier program A runs in polynomial time $p(n)$ on input of size n, then for every fixed n, there is a circuit C_n of size about $O((p(n))^2(\log p(n))^k)$ such that for every input $x = (x_1, \cdots, x_n)$, $A(x) = C_n(x_1, ..., x_n)$. That is, the circuit C_n solves problem A on all inputs of length n.

- **Proof Sketch.** Assume the program P is written in some low-level machine language (or compile it). Because P takes at most $p(n)$ steps, it can access at most $p(n)$ memory cells.

    - So, at any step, the *global state* of the program is given by the contents of these $p(n)$ cells plus $O(1)$ program counters. In addition, no register/cell needs to contain numbers bigger than $\log p(n) = O(\log n)$ bits. Let

$$q(n) \; = \; (p(n) + O(1)) \times O(\log n)$$

  be the size of the global state.
  - We maintain a $p(n) \times q(n)$ tableau that describes the computation.
  - The row i of the table is the state at time i. Each row of the table can be computed starting from the previous row by means of an $O(q(n))$ size circuit. (In fact, the microprocessor that executes the program P is such a circuit.)
  - End of Proof.

- Note that we don't need to know anything about the program P; the proof only uses the fact that it runs in polynomial time $p(n)$.

4.1 Proving More NP-Completeness Results

- **Lemma.** Suppose C is a NP-Complete problem, and A is some problem in NP. If we can show that C reduces to A, then A must be NP-Complete.

- By definition, all problems in NP reduce to C; since C reduces to A; all problems also reduce to A.

4.2 NP-Completeness of HAMILTON Cycle

- Given a directed graph $G = (V, E)$, is there a simple cycle (tour) T visiting each vertex of V exactly once?

- The problem is in NP clearly: given a tour T, verifying that it is HAM is easy: check that it visits each vertex exactly once, and that each edge is in E.

- Why reduce from $3SAT$? In part because we don't know what else to do. No other problem we know so far (Vertex Cover, MIS etc) seems related to a cycle.

- Consider an (arbitrary) instance of $3 - SAT$, with variables $x_1, \cdots, x_n$, and clauses $C_1, C_2, \ldots, C_k$. We show how to solve the 3SAT, given the power to decide if a graph has a HAM cycle or not.

- We begin by describing a graph that has 2^n different cycles that correspond in a natural way to the 2^n possible truth assignments to variables x_i After that we add nodes to model constraints imposed by clauses.

- Construct n paths $P_1, P_2, \cdots, P_n$, where P_i consists of nodes $v_{i,1}, \cdots, v_{i,b}$, for $b = 3k+3$, where recall that k is the number of clauses.

- There are edges between $v_{i,j}$ and $v_{i,j+1}$, in both directions. Thus, P_i can be traversed from *left to right* or from *right to left.* We then hook up these n paths as shown below:

```
                 s


  O1 --- O --- O ............ Ob


  O1 --- O --- O ............ Ob


  O1 --- O --- O ............ Ob


                 t
```

- Join each $O1$ to $O1$ and Ob in the next row; similarly, join each Ob to $O1$ and Ob in the next row.
- Finally, join edges from s to $O1$ and Ob in first row, from $O1$ and Ob in last row to t, and one edge from t to s.
- Consider what the HAM cycle in this graph looks like. After entering s, the path can take P_1 left-right or right-left; and regardless of how it traverse P_1, it can then do the same for P_2, etc etc, until it finishes P_n, and then enters t, and then finally take the edge from t to s.
- Thus, it has precisely 2^n ways to form a HAM cycle in this graph.
- Each path P_i, thus, models an *independent choice* for setting the variable x_i: left-to-right means $x_i = 1$; right-to-left means $x_i = 0$.
- Now we add extra nodes to model clauses, so that the final graph has a HAM cycle iff the formula is satisfiable. Consider a clause

$$(x_1 \vee \ !x_2 \vee x_3)$$

- In the language of HAM cycle this clause is saying that the cycle should traverse P_1 *left to right* OR it should traverse P_2 *right-to-left* OR it should traverse P_3 *left to right.* (At least one of these should be true.) So, we add a new node c_1 (for clause 1) that forces just this.

- For each clause c_j, we will reserve two adjacent node positions ($3j$ and $3j+1$) in each path where c_j can be spliced.

- If c_j contains un-negated variable x_i, then we add edges from $v_{i,3j}$ to c_j, and back edge from c_j to $v_{i,3j+1}$.

- If x_i is negated in c_j, then we add an edge from $v_{i,3j+1}$ to c_j, and a back edge from c_j to $v_{i,3j}$.

- Thus, the node c_j can be easily spliced into the HAM cycle if the cycle traverses path P_i corresponding to x_i in correct direction (left to right for 1 and right to left for 0).

- This completes the construction. Now, let us discuss the correctness.

  - First, suppose 3SAT is satisfiable. Then we form a HAM cycle following our plan: if x_i is set to 1, traverse path P_1 left to right; otherwise right to left. For each clause c_j, since it is satisfied by assignment, there will be at least one path P_i in it that is going in the *correct direction relative* to c_j, and so we can splice c_j into it via edges incident to $v_{i,3j}$ and $v_{i,3j+1}$.
  - Conversely, suppose the graph has a HAM cycle. If the cycle enters a node c_j on edge from $v_{i,3j}$, it must depart on an edge $v_{i,3j+1}$. If not, then $v_{i,3j+1}$ will have only one un-visited neighbor left, namely, $v_{i,3j+2}$, and so the tour will not be able to visit this node and maintain HAM property.
  - Symmetrically, if the tour enters $v_{i,3j+1}$, it must depart immediately to $v_{i,3j}$.

5 Some NP-Complete Numerical Problems

5.1 Subset Sum

- Given a sequence of integers $a_1, \cdots, a_n$, and a parameter k, decide if there is a subset of integers that sum to exactly k.

- This is a true decision problem, not an optimization problem forced into a decision version. Membership in NP is easy. For the completeness part, we reduce *Vertex Cover* to *Subset Sum.*

- Outline:
  1. Start with G and parameter k.
  2. Create a sequence of integers and parameter k'.
  3. Prove that G has VC of size k if and only if there is subset of integers that sum to k'.
- **Reduction.** For simplicity, assume the vertices of G are labeled $1, 2, \cdots, n$.
- We define an integer a_i for vertex i, an integer b_{ij} for each edge (i, j), and finally an integer k'.
- Our reduction will have the property that if there is a subset of a_i's and b_{ij}'s that sum to k', then the subset of chosen a_i's form a vertex cover, and the set of chosen b_{ij}'s correspond to edges in G with exactly one endpoint in the cover. Furthermore, the size of the cover will be exactly k.
- Represent the integers in a matrix form. Each integer is a row, and the row should be seen as base-4 representation of the integer, with $|E| + 1$ digits.
- The first column of the matrix (the most significant digit) is a special one: it contains 1 for the a_i's and 0 for the b_{ij}'s.
- Then, there is a column (digit) for each edge. The column (i, j) has a 1 in a_i, a_j, and b_{ij}, and 0's everywhere else.
- The parameter k' is defined as

$$k' = k(4^{|E|}) + \sum_{i=0}^{|E|-1} 2(4^i)$$

- **Analysis.**
  - **From Cover to Subset Sum.** If there is a vertex cover of size k, then choose all integers a_i for which i is in C, and all b_{ij} such that exactly one of i, j is in C. Then, when we sum all these integers, doing operations in base 4, we have a 2 in all digits except for the most significant one. In the MSD, we have 1s appearing $|C| = k$ times. Thus, the sum of integers is k'.

- **From Subset Sum to Cover.** Suppose we find a subset C of V and $E' \subset E$ such that

$$\sum_{i \in C} a_i + \sum_{(i,j) \in E'} b_{ij} = k'$$

First, note that we never have a carry in the $|E|$ less significant digits; operations are base 4, and there are at most 3 ones in each column. Since the b_{ij} can contribute at most 1 in every column, and k' has a 2 in all the $|E|$ less significant digits, it means that for every edge (i, j), C must contain either i or j. So, C is a cover. Every a_i is at least $4^|E|$, and k' gives a quotient of k when divided by $4^|E|$. So, k's cannot have more than k elements.

5.2 Partition and Bin Packing

- **PARTITION.** Given a sequence of integers $a_1, \cdots, a_n$, determine if there is a partition into two subsets with equal sums.
- Formally, is there a subset I of $\{1, 2, \cdots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.
- PARTITION is clearly a special case of sumset sum; the k here is just half the total sum of elements, and is also known to be NP-Complete.
- Bin Packing is one of the most widely studied problems in CS and OR, perhaps the second most after TSP.
- Given items of size $a_1, a_2, \cdots, a_n$, and given an unlimited supply of bins, each of size B, we want to pack items into the fewest possible bins.
- The decision version is to decide if the items can be packed in k or fewer binds.
- The problem is in NP (easy), and for NP-Completeness we reduce PARTITION to it.
- Given an instance of PARTITION $\{a_1, a_2, \cdots, a_n\}$, create items of size $a_1, a_2, \cdots, a_n$. Let $S = \sum a_i$, be their total size. Make bin size B to be $S/2$, and let $k = 2$.
- The items can be packed in 2 bins if and only if the partition has a solution.
- Subset Sum is a special case of the Knapsack problem. An instance of the Subset Sum has integers $a_1, \cdots, a_n$, a parameter k, and the goal is to decide if there is a subset of integers that sum to exactly k.
- We can frame this as a knapsack problem by introducing n items, where item i has both size and value equal to a_i. The knapsack size is k.

- Now, the Subset Sum problem has answer YES if and only if the Knapsack problem has a solution with value k. (Because the size and value are equal, the knapsack can achieve value k only if it can be packed completely.)