16
NP-Completeness

Chapter Gist: *There are many important practical problems for which no polynomial time (P) algorithms are known. These problems can, so far, only be solved in nondeterministic polynomial time (NP), and currently this amounts to being intractable (exponential or worse). We define NP to be the class of polynomial time verifiable problems, and NP-Complete to be the* hardest of all NP problems *(§16.1). We present the role of NDTMs in formulating the theory of NP-Completeness in precise terms (§16.2). We take up the study of the Boolean satisfiability problem (SAT) given it has the distinction of being the first NP-Complete (NPC) problem identified (§16.3). We explain why SAT matters in practice, and also introduce a SAT-solver that can run within your own web browser (§16.8). We begin with the simpler 2-SATpolynomial time algorithm (§16.3.1). We describe a canonical problem called 3-SAT, and describe its role in showing new problems to be NP-Complete (§16.4). The idea of mapping reductions is central to this study, and we show that 3-SAT itself can be shown to be NP-Complete (§16.5). We show that the problem of finding k-cliques in a graph is NP-Hard by presenting a mapping reduction from 3-SAT to it (§16.6). We finish with some caveats and also a discussion of CoNP and allied complexity classes (§16.7).*

16.1 What Does NP-Complete Mean?

In the 1960s, computer scientists started noticing that many problems defy polynomial time algorithms; all they could come up with were exponential (or worse) algorithms. Examples of these problems included everyday scheduling problems such as the *Traveling Salesperson Problem* (**TSP**), one version of which is the following. Suppose you are asked to start from Salt Lake City, UT, travel by road and visit all the 48 US

state capitals of the contiguous USA exactly once and return to Salt Lake City in **18 days or less**.[1] What is the *optimal* route you would take?[2] Is there a better algorithm than computing the cost of all 48! such tours and picking the one that finishes in 18 days or less? All algorithms so far have been **intractable** (*exponential or worse*; an example is factorial).

However, scientists also noticed that given a **claimed solution** in the form of a sequential listing of state capitals (a "tour"), they can indeed **easily check**[3] (in polynomial time) whether the cost is below 18 days. This evidence presented for checking (called a **certificate**) is compact (*i.e.,* polynomial in length), and one can simply go as per this sequence, add up the costs and check if it is below 18 or not.[4] So while the cost of solving might be intractable, the cost of checking claimed solutions is polynomial. Scientists started calling this problem class of easy-to-check and difficult-to-solve problems "NP."

At this juncture, one might protest saying that the cost of checking claimed solutions being polynomial time is not a worthwhile "consolation prize." The real goal, they might say, is to *solve* problems in polynomial time and not merely *check a given solution* in polynomial time. They might[5] ask, "who will come up with a solution to check in the first place?!"

Unfortunately, in the world of algorithm design, one sometimes has to be humble and be content with what's available and feasible. In §16.1.1, we will show that knowing that a problem is checkable in polynomial time is a valuable piece of knowledge.[6] Easy checkability has, in fact, a crucial role to play in being able to settle the P versus NP question one day, showing that:

- *Either* it is impossible to have a polynomial time algorithm for a problem in NP that is currently intractable;
- *Or* all such problems have polynomial time algorithms (hence one can stop bothering about NP)![7]

There are actually many problems for which even the cost of checking the solution appears to be intractable. Let us take the following variant of the traveling salesperson problem: *'show that* **there isn't a tour** *where the cost is ≤ 18 days.'* Here, to answer "yes," it appears that one must list *every* tour and show that each tour takes more than 18 days. Even the certificate involved in this check case is exponentially long (*i.e.*, the sequence of all tours). This is not easily checkable (takes exponential time).[8] Thus, having easy checkability is really getting us somewhere.

16.1.1 Grouping Problems: Solving One Implies Solving All

What researchers in complexity have done is to group[9] problems into a class of problems called NP, and then identify the *hardest* problems in this class, which is the NP-Complete class. All problems in NP-Complete are equally hard; the problems outside of NP-Complete (but inside NP) are

[1] We assume that there are fixed **time costs** to go between any two capital cities. For instance, the time it takes to go between Salt Lake City and Boise is assumed to be fixed and known. It has been estimated that it will take about 10 days to do all 48 states without traffic and without any stoppage. Our capital-to-capital costs can be assumed to be such that the 18-day figure is met by some tours and not met by others.

[2] Studies have shown that bees solve the traveling salesperson problem while covering a collection of flower patches optimally.

[3] We use the word *check* in the sense of checking whether the claim is true for a specific instance. For example, we can easily *check* that $(3,4,5)$ form a Pythagorean triple by checking the identity $3^2+4^2 = 5^2$ to be true. We will use the term *verify* to connote something deeper, such as Fermat's last theorem: there are no Pythagorean triples of the form $x^n + y^n = z^n$ for n above 2.

[4] In fact, if the certificate itself is exponentially long, even reading in the certificate will take an exponential amount of time. In this case, the cost of checking cannot be polynomial.

[5] ...somewhat irately...

[6] ...and perhaps the best known computation that can actually be carried out in P-time for many a problem. One has to keep in mind that for many problems, even the "easy checkability" in P-time has not been proven. This would make one feel grateful that at least easy checkability in P-time is an option.

[7] Many scientists think that this is a highly unlikely outcome.

[8] This is not a proof, but it already shows that sometimes all certificates we manage to come up with end up being long, causing the checking cost to go up.

[9] Or "corral"

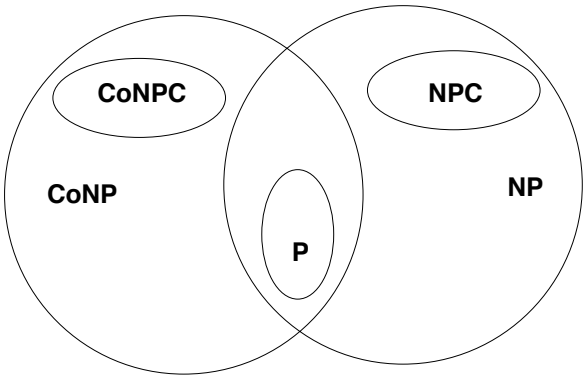


Figure 16.1: The language families P, NP, and NPC. All these set inclusions are likely to be proper.

less hard. These aspects are illustrated in the Venn diagram of language family inclusions in Figure 16.1 (for now, please ignore the families whose names start with "Co"; they are discussed in §16.7). The key property that one ensures (before calling a problem NP-Complete) is that *if even one problem in NP-Complete has a polynomial time algorithm*, then *all of NP will have a polynomial time algorithm*.[10] This behavior will be ensured in the process of showing a problem to be NP-Complete (§16.5.3). All we will be left with (*in this context*) will be polynomial time algorithms.[11] In summary, with respect to the "consolation prize" discussion earlier:

- We group polynomial time checkable (synonymous with verifiable) problems into NP. Computer science research has been inducting problems into the NP class beginning in the 1960s (detailed in §16.2).[12]
- Showing that an NPC problem can be solved in polynomial time will collapse the entire class NP, and essentially turn it into the class P.

This chapter studies *algorithms* (not *procedures*) using TMs (*i.e.*, these TMs will halt on all inputs). Also, given the introductory nature of this chapter, we only study *time complexity*. *Space complexity* is also studied using TMs.

16.1.2 Some Historical Notes

Let us understand the ideas thus far in the context of prime numbers.

Given a natural number of d digits, what is the algorithmic complexity of checking whether it is a prime? Is this 21-digit number prime: 147,573,952,589,676,412,927 ? (It equals $2^{67}-1$, the 67th Mersenne number (M_{67}); the nth Mersenne number, M_n, is 2^n-1.)

In 1903, Frank Cole gave a lecture in which he performed the multiplication of 193,707,721 and 761,838,257,287 by hand on a chalkboard, obtaining M_{67}.[13] Thus Cole was able to prove (over the course of one lecture) that this number wasn't a prime. Yet, to obtain the factors, he

[10] ..and the "Co" families will also disappear.

[11] Clearly there will continue to be other exponential algorithms; it is only the very important NP class that will become equal to P.

[12] If one **proves** that one of these NPC problems *does not have a P-time algorithm*—the scenario that most scientists believe is likely—they would have solved one of the most important of open problems in CS. They would also win the $1 million prize money that the Clay Mathematics Institute has set apart for this challenge.

There is also another term lurking around in this area called **NP-hard**. It means *at least as hard as NP*—meaning it could be even harder than NP. In fact, some NP-hard problems are so hard that they are actually undecidable! We cover this topic in §16.7.

[13] He silently put down the chalk and walked away to a thunderous applause. https://en.wikipedia.org/wiki/Frank_Nelson_Cole

spent "*three years of Sundays.*" This anecdotal evidence itself shows that proving that a number is composite (non-prime) can take very long, but checking that it is composite does not. This problem happens to be in NP.

The largest known prime number has 23,249,425 digits;[14] how hard would it be to *prove* that it is prime? Until the year 1977, there wasn't a definite algorithmic classification that applied to the entire set of primes. In 1977, Pratt [39] proved that primality checking is an NP-problem, meaning it still defied a polynomial solution, but for primes, the certificate—a proof that a number is prime—is indeed succinct, and the proof can be checked in P-time.

In 2002, Agrawal, Kayal and Saxena obtained a P-time algorithm [3] (now called the "AKS algorithm") for primality checking[15] with complexity roughly $O(d^{12})$ where d is the number of digits in the number.[16] Please note:

- Complexity classifications evolve with the state of human knowledge.
- While we can check whether a number is prime or composite in P-time (thanks to the AKS algorithm), it does not mean that *finding the factors* can be done in P-time.

Factorization is the key hard algorithm upon which cryptographic systems are built. We can easily *check* that a given number is factored correctly (as evidenced by a certificate), but the hardness of *generating* factors is still unknown—and appears to be hard. One wishes for such algorithms to remain hard—or else today's crytographic systems could be rendered useless.[17] See Cook's discussions [18] on the importance of P versus NP.

16.2 NPC Notions Defined Based on NDTMs

Let us consider the computation trees supplied with Figures 16.2 and 16.3 (repeated from Chapter 13) for the DTM and NDTM designed to recognize a '101' in the input string. The DTM accepts 10101 by recognizing the first 101 and rejects both 01 by going to state StuckNo0Aft1, and rejects ε ("") by going to state StuckNo1beg.

The NDTM has six executions when fed 10101: it accepts both the 101's and rejects upon badly chosen nondeterministic selections.

16.2.1 P-time

We define P-time with respect to the runtime of DTMs. The Turing machine model is **robust** in that it faithfully models *not just the computing power* but also the *computational efficiency* of realistic computers within a polynomial factor. Anything realizable on a TM with polynomial ($\mathcal{O}(n^k)$) complexity[18] can be realized on a realistic computer with complexity $\mathcal{O}(n^{k'})$ for perhaps $k' < k$.

[14] At the time I wrote this sentence; see https://en.wikipedia.org/wiki/Prime_number.

[15] One may prefer the term "primality testing" also, in this context. In our sense, it means the same as checking, as the answer is produced for the given instance.

[16] https://en.wikipedia.org/wiki/AKS_primality_test mentions the details including the notation $\tilde{O}(log^{12}(n))$ where n is the number itself; thus $log(n)$ obtains the number of digits. It has been improved to $\tilde{O}(log^{6}(n))$. The notation $\tilde{O}$ stands for "soft $\mathcal{O}$" and is explained in the references; it ignores logarithmic terms.

[17] Quantum computers have the ability to factor numbers in polynomial time. See a video tutorial on how this is done by Vazhirani https://youtu.be/YhjKWAMFBUU.

[18] If the complexity measure $\mathcal{O}(g(n))$ is ascribed to a function $f(n)$, it means that there exists some $k \in Nat$ such that $f(n) < C \cdot g(n)$ for all $n \geq k$, and $C \in Nat$ being a constant. This is the same "big Oh" one studies in a basic course on algorithms.

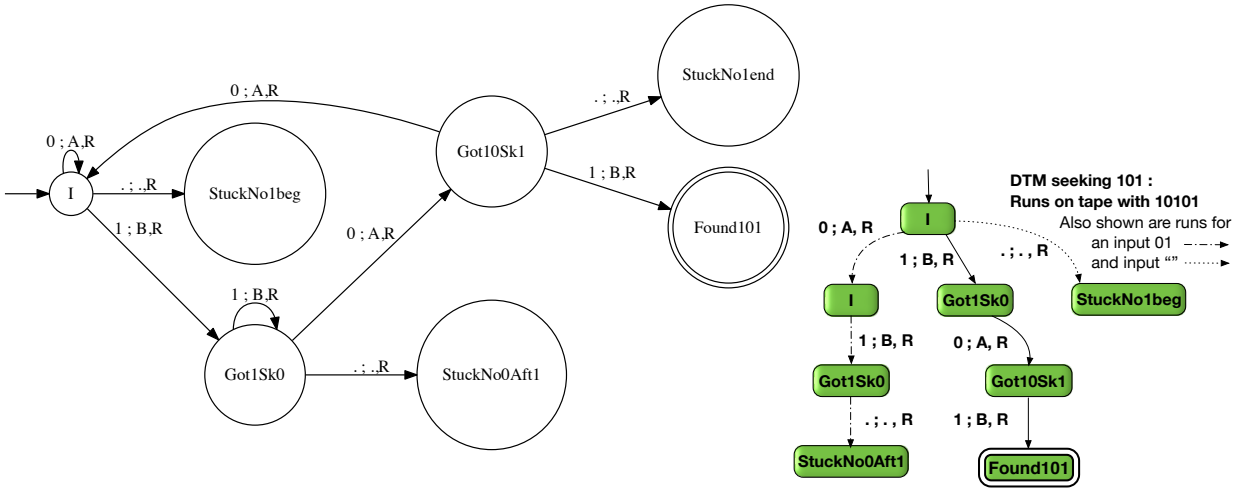
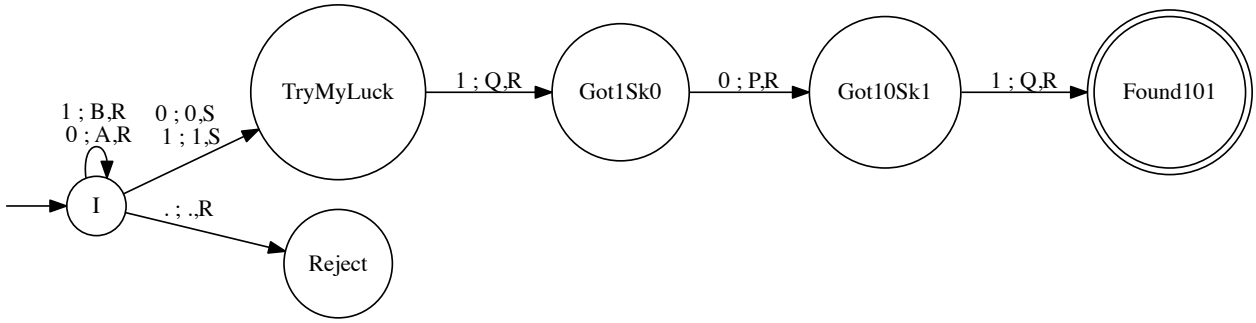


Figure 16.2: Transition diagram and computation tree for a DTM that looks for 101 within given w.



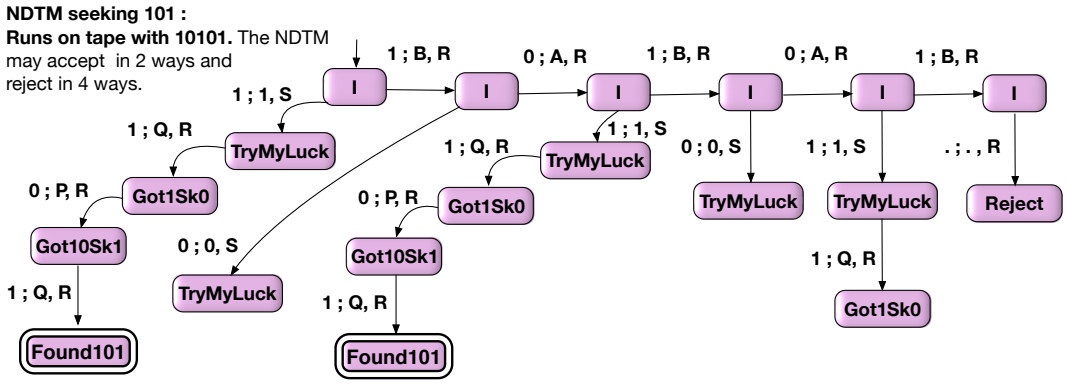


Figure 16.3: Transition diagram and computation tree for an NDTM that looks for 101 within given w.

Definition 16.2.1:
(a) For input w of length n, the **execution time of a DTM** is the number of steps taken along its (**only**) computational path.
(b) If a DTM D_L has $\mathcal{O}(n^k)$ execution time, it is said to be a **polynomial time decider.**
(c) **P** is the family of languages where for each language $L \in$ P, there is a polynomial time decider D_L.

16.2.2 NP-time

NP-time is the upper bound of the computational cost across all the nondeterministic computational paths.

Definition 16.2.2:
(a) For input w of length n, the **execution time of an NDTM** is the **maximum** number of steps taken along **any computational path**.
(b) If an NDTM N_L has $\mathcal{O}(n^k)$ execution time, it is said to be a **nondeterministic polynomial time decider.**
(c) **NP** is the family of languages where for each language $L \in$ NP, there is a nondeterministic polynomial time decider N_L.

Across all the executions produced by the NDTM of Figure 16.3 on input 10101, the maximum number of steps[19] taken is 6.

[19] For Jove TMs, it is the maximum amount of fuel consumed along **any path.** Note that it is very easy to be somewhat mischievous and design an NDTM that branches infinitely, but with each computational path taking a finite number of steps (just put a . ; . , R transition from Reject going back to Reject). Thanks to this mischief, we can create a situation in which there isn't such a maximum. But now, recall the "no wimp clause" we stated in §14.2.4. *We must avoid creating a semi-decider when a decider is possible.*

Just to drive this point home, imagine that someone added a linear transition sequence of 10^6 steps after the Reject state, with each transition labeled by . ; . , R. This would bloat the execution length to 10^6! All these "definition destroying constructions" are immaterial! The kinds of reasoning we will be engaged in will be of the form "**IF** problem P with complexity x can be solved, **THEN** via mapping reductions, we will show that a related problem Q can be solved with complexity y." In other words, we will be pursuing *relative complexity measurements* achieved through mapping reductions.

16.2.3 NP Verifier

To ease the construction of proofs in this area, one likes to have *two alternative definitions of NP*. Definition 16.2.2 presented the so-called *decider view of NP.* The other view is the *verifier view of NP*. This construction also equates the notion of "a certificate" to internal NDTM decisions.

Definition 16.2.3: A nondeterministic polynomial time verifier is an NDTM that takes an input w along with a certificate c (together packaged as $\langle w, c \rangle$) and makes a decision in NP time. The purpose of such a verifier is to check w's acceptance status by exploiting the given certificate c.

We will provide an example of such a verifier in §16.2.4, and a proof sketch that given an NP decider we can define an NP verifier (and vice-versa) in §16.2.5.

16.2.4 Examples of P-time and NP-time Deciders

In Figure 13.10, we presented a P-time decider for strings of the form $w\#w$ where $w \in \{0,1\}^*$. This is also an NP-time decider because DTMs are a special case of NDTMs. In Figure 13.11, we presented a decider for strings of the form ww where $w \in \{0,1\}^*$. The fact that all the computational paths of this NDTM take only a polynomial amount of steps allows us to state that this is indeed an NP-time decider.

To envisage an NP decider that is not in P, consider the Traveling Salesperson problem. Here is the high-level code for an NDTM. (Note that we don't know whether there is a P-time decider for the TSP. Also, from now on, we will not bother to present actual DTMs or NDTMs but only present their high-level pseudo-code):

NP decider for TSP:

- Accept the US map and the costs between the states on the input tape. Write SLC at the end of the given input.
- Starting from the start state (when we are in SLC), choose a state capital that hasn't been considered thus far (say state S_1). Write S_1 at the end of the input tape.
- Pick another state S_2 that hasn't been considered. Again, write S_2 at the end of the input tape.
- Do this for all 47 other states.
- Call a verifier Turing machine that checks that the state sequence SLC $S_1\, S_2\, \ldots S_{47}$ SLC written at the end of the tape meets the given cost criterion. **Accept** if so; **Reject** otherwise.
- This ND algorithm has cost NP-time because along *any one computational path*, what happens is this:
  - We choose a linear list of states
  - We make a subroutine call to another TM—called a **verifier TM**—that simply adds up the cost of the edges present in SLC $S_1\, S_2\, \ldots S_{47}$ SLC
  - This computational path will only involve a polynomial number of steps. □

16.2.5 Decider versus Verifier Views

The aforesaid algorithm to write an NP decider for TSP used the following trick: it converted the given problem into two subproblems:

- Write a "certificate" at the end of the user-given input.
- The TM is made to decide based on the certificate by calling an "inner" TM. We call this latter TM a *verifier*.

In general, an NP decider can be converted to an NP verifier, and vice-versa. We now explain the process abstractly. Let L be a language and

N_L stand for the NDTM decider for L and V_L for a P-time verifier for L.

Given N_L, Obtain V_L: The verifier V_L that "we must output" is built as follows. Given an NDTM N_L that is an NP-time decider, ask for an additional input c that is also of polynomial length with respect to the input w that N_L already expects. Use c to pick the nondeterministic selections that N_L makes at each juncture when N_L is confronted with nondeterminism. Now, given that no path of N_L is longer than a polynomial quantity, we can completely resolve the nondeterminism via c and obtain V_L, a verifier that now takes $\langle w, c \rangle$. When N_L decides, V_L also reaches the same decision.[20]

Given V_L, Obtain N_L: The decider N_L that "we must output" is built as follows. Prepend V_L's code with a phase that can be called the *internal certificate auto-generation phase* in which the prepended code simply writes out a c on the TM tape the end of w and then feeds $\langle w, c \rangle$ to V_L. The decision of V_L is emitted as the decision of N_L. [21]

[20] Think of c as "rudder steering control" on a boat in a vast lake. c is then a sequence of steps "take the second turn; then the first turn; then the fourth turn; …." V_L simply allows us to input the "guess" via c. **Depending on the purpose N_L is supposed to achieve (i.e. NP-decide membership in L), a suitable set of turn instructions can be generated.** Externally, it will appear as if the "newly minted" V_L is asking for help ("Phone a friend" in "Who Wants to Be a Millionaire"), but internally it is simply doing "backseat driving" of N_L.

[21] Again, depending on the language that V_L verifies membership into, aided by certificate c, one can always construct the certificate auto-generation phase to force the same decision "out of V_L." Externally it will appear as if the "newly minted" N_L is being "smart" and deciding in NP-time. But internally, it is actually coughing up a certificate and feeding that to V_L which needs this crutch.

16.3 Introducing SAT Problems

The term Boolean satisfiability or SAT refers to the satisfiability of a general Boolean expression. It suffices for us to study two special cases of SAT:

- 2-SAT: the satisfiability of conjunctive normal form (CNF) Boolean formulae with exactly two literals (a Boolean variable or its negation) per clause.
- 3-SAT: the satisfiability of conjunctive normal form (CNF) Boolean formulae with exactly three literals per clause.

The reason we study 2-SAT is that it has a beautiful P-time algorithm due to Aspvall, Plass and Tarjan [5]. The reasons to study 3-SAT are several, some of which are: (1) merely going from "2 to 3" shoots up the complexity from P-time to NP-Complete. (2) it is the canonical NP-Complete problem, by understanding which deeply, one tends to understand the theory of NP-Completeness rather well. We now define the basic notions surrounding 2-SAT and 3-SAT.

Definition 16.3: A variable or its negation is called a *literal*. A Boolean formula in *conjunctive normal form* (CNF) is a conjunction of clauses. It is a 2-CNF formula if it is a conjunction of 2-Clauses and a 3-CNF formula if it is a conjunction of 3-Clauses. A 2-Clause is a disjunction of two literals, and a 3-Clause is a disjunction of three literals. A 2-Clause is equivalent to a conjunction of two implications.

Notations and Examples: Let ! stand for NOT, . for AND, and + for OR. Let → stand for *implication* (we use this notation to make this operator look like a graph edge, as we will be building implication graphs).

Let a through e be variables (they are also literals). Also !a through !e are literals. We employ **True** or 1 interchangeably, but prefer the former for *strongly connected components* (SCCs, see Figure 16.4 for a definition) and the latter for literals. Similarly, we employ **False** or 0 interchangeably, but prefer the former for SCCs and the latter for literals. Here are sample 2-CNF and 3-CNF formulae:

2-CNF: (a + b).(!a + c).(!b + e).(f+!f)
3-CNF: (a + b + c).(!a + b + b).(b + d + e).(a + f+!f)

The 2-Clause (a + b) is equivalent to the conjunction of two implications:

- !a → b
- !b → a

This is because

- (a + b) is equivalent to !a → b.
- (a + b) is equivalent to (b + a) which can be translated to !b → a.

We cannot translate a 3-Clause in this manner.

Definition 16.8
$L_{2sat} = \{\langle\phi\rangle : \phi \text{ is a 2-CNF formula that is satisfiable.}\}$
$L_{3sat} = \{\langle\phi\rangle : \phi \text{ is a 3-CNF formula that is satisfiable.}\}$

Note: Some readers may wish to look at §16.8 and actually play with a SAT solver to cultivate some familiarity with Boolean satisfiability.

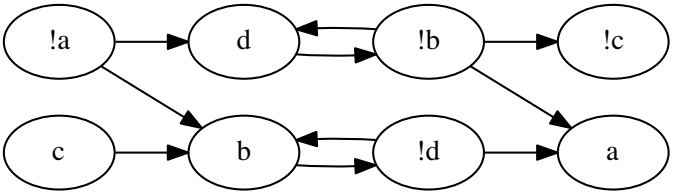
16.3.1 A Warmup: 2-SAT

The fact that a 2-Clause is equivalent to a conjunction of two implications can be exploited in obtaining a P-time satisfiability checking algorithm. Figure 16.4 shows an example 2-CNF formula.

Implication Graph: We can build the implication graph for the formula `F` above, also shown in Figure 16.4. An implication graph is one that treats each 2-CNF clause as *two* implications. More specifically, this graph is obtained by modeling each 2-CNF clause of the form (p + q) as the conjunction of two implications, namely !p → q and !q → p. We treat each implication as a graph edge. As soon as we add all possible implication edges, we end up creating a graph with the following structural properties:

- The implication graph ends up having groups of nodes and edges that form *maximal strongly connected components* (maximal SCCs). In a directed graph, a strongly connected component (SCC) is a collection of nodes that are reachable from each other. A maximal SCC is one that pulls in the maximum number of nodes into each SCC such that

```
F = (a+b).(b+!c).(!b+!d).(b+d).(d+a)
```



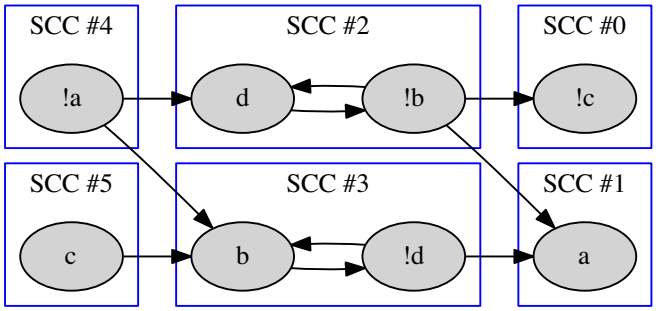


Figure 16.4: 2-CNF formula, and illustration of Aspvall et al's 2-SAT algorithm. Here, + stands for Boolean OR, . stands for Boolean AND, and ! stands for Boolean negation. The first directed graph above is what we initially obtain after we convert each disjunction such as (a + b) into a pair of implication edges. The second directed graph clusters groups of nodes and edges into *maximal strongly connected components* (maximal SCCs). In a directed graph, a strongly connected component (SCC) is a collection of nodes that are reachable from each other. A maximal SCC is one that pulls in the maximum number of nodes into each SCC such that they can all reach each other. In our example, maximal SCCs (#2 and #3) can have at most two nodes. All other maximal SCCs only have one node each.

they can all reach each other. **In Figure 16.4, we enclose all the maximal SCCs within rectangular boxes.**

- In our example, there are two maximal SCCs with two nodes each, namely SCC #2 and SCC #3. All other maximal SCCs only have one node each. Note that we cannot merge any of the maximal SCCs #0 through #5 and *still have each remain a maximal SCC*. For example, we cannot merge SCC #2 and SCC #0 because whereas !b can reach !c, !c does not reach !b.
- If any SCC includes a variable and its complement, the given formula is not satisfiable. For example, suppose an SCC contains a literal p and its negation !p. Then, such an SCC represents the conjunction of !p → p and p → !p, which is not satisfiable. This is because !p → p simplifies to p and p → !p simplifies to !p, and their conjunction cannot be satisfied. **In our example, this situation does not arise, and hence our formula is satisfiable.**
- Given that a formula is satisfiable, we can order the SCCs into a partial order, as also shown in Figure 16.4. **This is a topological sort of the SCCs.**

- **If we do all these steps correctly, we will notice that for each SCC, there is also a dual SCC.** For example, SCC #2 and SCC #3 are duals, with the negation signs in the literals flipped. Likewise, SCC #4 and SCC #1 are duals, as are SCC#5 and SCC#0.

Obtaining the Satisfying Assignment: For a collection of maximal SCCs that are situated in a partial order, we can proceed as follows in order to find a satisfying assignment. Hereafter we use "SCC" to refer to maximal SCCs.

- Consider the collection of SCCs "bottom-up" (or more precisely, as per the *reverse topological sort order*).
- If an SCC is unmarked, mark it **True**. Immediately mark its dual **False**.
- When we assign truth values to an SCC, all the literals in the SCC obtain the same assignment as the SCC. In our example,
  - we mark SCC #1 **True**, and hence mark SCC #4 **False**. This assigns a = 1 and !a = 0.
  - we mark SCC #0 **True**, and hence mark SCC #5 **False**. This assigns !c = 1 and c = 0.
- In some cases, we have a choice of making an SCC **True** or **False**. For example, SCC #3 containing b, !d and SCC #2 containing d, !b are incomparable in a topological sort; so they can be assigned arbitrarily.
- Pick **True** for SCC #3 containing b, !d. Pick **False** for SCC #2 containing d, !b.
- Now we've assigned all SCCs. The final assignment obtained is: !c = 1, c = 0, a = 1, !a = 0, b = 1, !d = 1, d = 0, !b = 0.

The following facts are true of this algorithm:

- An SCC assigned **False** only has **False** as predecessors.
- An SCC assigned **True** only has **True** as successors.

These facts are important in so far as they guarantee that the algorithm will never introduce a contradiction.

16.3.2 2-SAT: Examples and Algorithm

We now illustrate our construction on two additional examples, the first being satisfiable and the second unsatisfiable. These two examples are quite related in that they consider Boolean combinations of two Boolean variables, namely a and b.

In Figure 16.5, the graphs were obtained from the Boolean formula:

$$(a+b).(a+!b).(!a+b)$$

Given that the SCC #0 is encountered first in the reverse topological sort, we can assign a = b = 1, which assigns for the dual graph !a =!b = 0.

In Figure 16.6, the graphs came from the Boolean formula

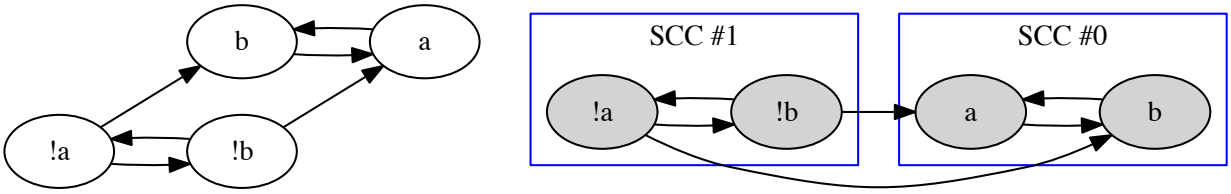


Figure 16.5: The implication graph and the SCCs for the CNF formula

$$(a+b).(a+!b).(!a+b).(!a+!b).(!a+!b)$$

Now we have only one SCC within which all the nodes fall, and thus we cannot obtain a satisfying instance for this Boolean formula.

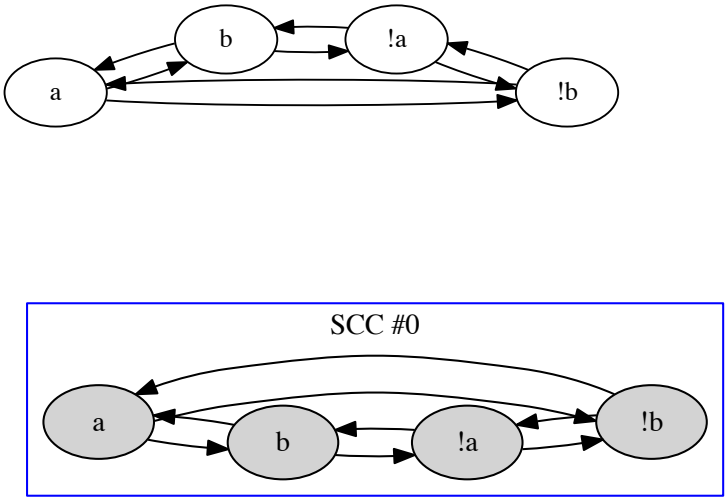


Figure 16.6: This graph and the SCCs are for the CNF formula $(a+b).(a+!b).(!a+b).(!a+!b)$

2-SAT Algorithm Recap: To recap, the algorithm to determine 2-SAT consists of the following steps:

- Obtain the implication graph.
- Divide the implication graph into maximal SCCs ("SCCs").
- If any SCC contains a literal and its negation, exit with "UNSAT".

- Else, consider the SCCs according to the reverse topological sort order.
- For an SCC without any truth assignment, assign that SCC **True**.
- Immediately assign its dual SCC **False**.
- Move up the topological sort.
- For an SCC and its dual that are not ordered, pick the assignment arbitrarily for one of the members of the dual; the other member naturally receives an inverted assignment.
- Move up the order and complete assigning all SCCs.
- We revert to assigning the next unassigned SCC to be **True**, in case there are still unmarked SCCs.
- In the end, output the assignment for all literals, by replicating the truth assignment for the SCC as really being the corresponding truth assignments for the literals in the SCC.

16.4 3-SAT and Its NP-Completeness

We could "pull off" an algorithm for 2-SAT (§16.3.2) only because of the implication graph that underlies 2-SAT. Unfortunately, there is no polynomial time algorithm for 3-SAT that we know of. The best-known result is that it is NP-Complete. We now provide two **equivalent** definitions for NP-completeness.

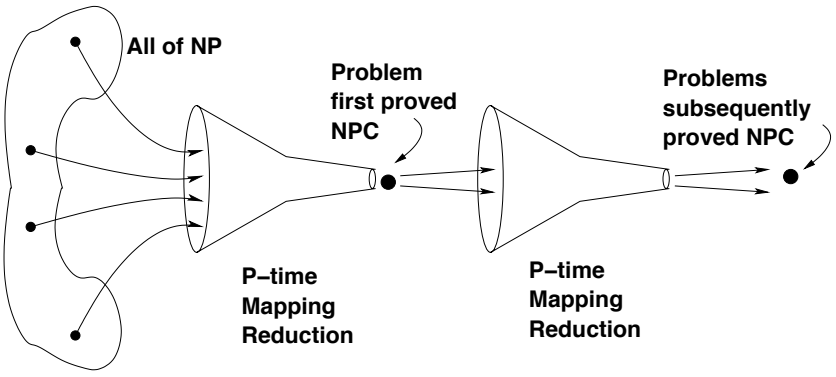


Figure 16.7: Diagram illustrating how NPC proofs are accomplished. The problem first proved NPC is 3-SAT. Definition 16.4(a) is illustrated by the "left funnel" while Definition 16.4(b) is illustrated by the "right funnel." (The funnels serve as a gentle reminder that mapping reductions need not be onto.)

Definition 16.4:(a) L is NPC if
(i) L is in NP, and
(ii) **for every language** $X \in$ NP, we have $X \leq_P L$.
(Accomplishing (ii) alone means that L is **NP-Hard**.)
Definition 16.4:(b) L is NPC if
(i) L is in NP, and
(ii) for **some other language** $L' \in$ NPC, we have $L' \leq_P L$.
(Accomplishing (ii) alone means that L is **NP-Hard**.)
The definitions are equivalent because if a language L' is NPC,

then we have, for every $X \in \mathsf{NP}$, $X \leq_P L'$; then, given that $L' \leq_P L$, we have $X \leq_P L$. This is because the composition of mapping reductions is also a mapping reduction.

The P-time mapping reductions illustrated by the funnels in Figure 16.7 are the same mapping reductions defined in Definition 15.5, except the function f in that definition is a **deterministic polynomial-time** (P) reduction. In §16.6, we will concretely demonstrate how the "second funnel" works by taking a 3-SAT formula and turning it into an undirected graph.

Say one of the "problems subsequently proved NPC," modeled by language L_{new}, ends up having a P-time algorithm D_{new}.[22] We can then have a P-time algorithm for 3-SAT as follows:

- Input a 3-SAT instance ϕ modeled by language L_{3sat}.
- Apply the second mapping reduction $\leq_P$ (of the second funnel) and create an instance l_{new} of the problem modeled by L_{new}.
- Apply D_{new} on l_{new}. Because of $\leq_P$ being a mapping reduction, D_{new} accepts l_{new} **if and only if** ϕ is satisfiable.
- This gives us a P-time decider for L_{3sat}.
- Via the "first funnel" we now have a P-time decision procedure for the whole of NP. This will essentially eliminate NP and establish P=NP.

[22] Perhaps L_{new} itself is the language of graphs that have k-cliques.

16.5 3-SAT Is NP-Complete

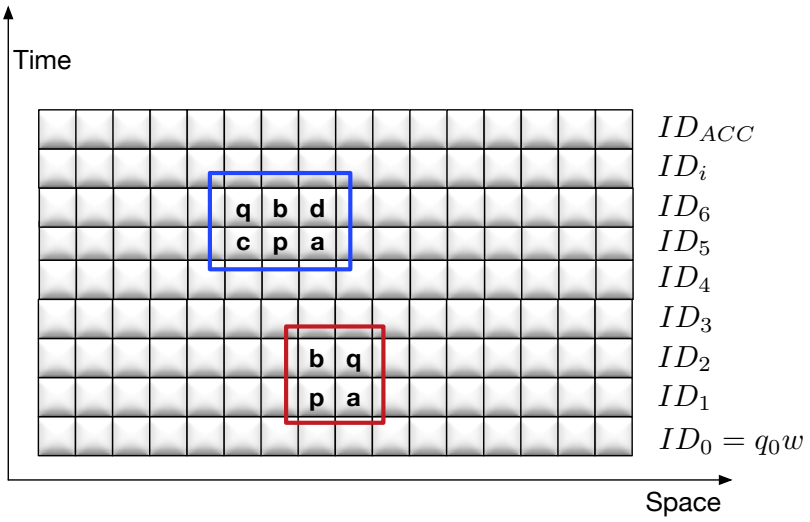


Figure 16.8: Proof of the Cook-Levin Theorem

Theorem 16.5: 3-SAT is NP-Complete. (This is known as the Cook-Levin theorem, discovered independently by Cook [17] and Levin [31].)

Proof: According to both Definition 16.4(a) and (b), we have to show that 3-SAT is in NP. We will now show it using both the decider and the verifier views.

16.5.1 3-SAT is in NP

We are given a 3-SAT instance—a 3-CNF formula ϕ. The NDTM presented below can easily extract the variables in w in polynomial time.

NP-decider: Build an NDTM that chooses a random assignment (0 or 1) for each variable in ϕ. Check that this assignment satisfies ϕ. This NDTM runs in NP time.

NP-verifier: The NDTM being built also asks for an actual variable assignment c ("certificate") to follow ϕ on the tape. The NDTM then checks whether $\phi(c)$ is true, accepting exactly when so.

16.5.2 Every Language in NP Reduces to 3-SAT

To show 3-SAT is NP-Hard, we have to go by Definition 16.4(a), as there is no previous NP-Complete problem to reduce from. Every language in NP has an associated NDTM that accepts or rejects in NP-time.[23] The crux of our proof is that given one of these NDTMs, one can devise a general way to check for the acceptance of an input w by the NDTM using 3-SAT.

Let us focus our attention on the first reduction (funnel) of Figure 16.7. Given an arbitrary N_L and an arbitrary $w \in \Sigma^*$, consider the computation of N_L on w starting from the instantaneous description $ID_0 = q_0 w$. In Figure 16.8, we portray this as the bottommost layer along the space/-time diagram against ID_0 where we use the notation for instantaneous descriptions (IDs) introduced in §13.8.[24]

When this NDTM computes, it checks the cell under its tape head, changes this cell and moves right (an example is in the ID_1 to ID_2 march). It can also move left if it is not walking off the left end of the tape (an example is in the ID_5 to ID_6 march).[25] It can be seen that all changes caused by the TM's transition relation Δ affect only a window of size 3 at most. During the ID_1 to ID_2 march, an ID where the "head state" is **p** and the tape symbol under the head is **a** changes to the head state becoming **q**, the head moving right and with the **a** changed to a **b**. During the ID_5 to ID_6 march, an ID where the "head state" is **p** and the tape symbol under the head is **a** changes to the head state becoming **q**, the head moving left and with the **a** changed to a **b**. (The tape symbols **c** and **d** capture enough "stuff" around the TM head.) In Figure 16.8, we show

[23] Recall that this notion can be summarized as **reaching a decision** in NP-time.

[24] We assume that we are working with a singly infinite tape with the tape extending in "space" to the right. One can systematically translate TMs that carry out computations on doubly infinite tapes to those that use singly infinite tapes with only a polynomial increase in time.

[25] We can prevent a TM from going to the left of the leftmost cell of a singly infinite tape. Again such a transformation takes only a polynomial amount of extra cost. We omit showing such steps for simplicity to highlight our main reduction arguments.

a march of ID_1 through ID_6, show ID_i for generality, and the accepting ID is shown as ID_{ACC}.

What you really have to imagine is not just one of the NDTM choices, but **all** NDTM choices. Thus, imagine ID_1 to encompass all possible changes that ID_0 could have been subject to, for all the nondeterministic options of N_L. In general, we assume that when going from layer ID_i to ID_{i+1}, we imagine that the layer we draw for ID_{i+1} represents all possible (nondeterministic) ways in which it could have been obtained from ID_i.

Now, we know that we will "pile on" only a polynomial number of layers in this manner before a decision is reached. The reason is of course that N_L is a TM with nondeterministic runtime being NP.

Here is how SAT enters the picture:

- We can capture the evolution from ID_i to ID_{i+1} through a 3-CNF Boolean formula of polynomial length ϕ_i. The construction of this formula is described in many references [42] in this field, and we don't repeat that. Fortunately, this single formula can capture *all the nondeterministic evolutions* from layer i to layer $i+1$ in one shot.
- We can also introduce formula ϕ_0 to capture the constraints on ID_0 and formula ϕ_{ACC} to capture the constraints on the final ID containing the accepting ID.
- Thus, the entire "pile of IDs" depicted in Figure 16.8 can be captured by a formula:

  $$\Phi = \phi_0 \wedge \phi_1 \wedge \ldots \phi_i \wedge \ldots \phi_{ACC}$$

  This formula encodes all the nondeterministic evolutions from start to finish of the NDTM. *All the NDTM paths* are rolled into this single formula.
- Thus, given **any** NDTM N_L, we can synthesize a Boolean 3-CNF formula Φ describing **all the nondeterministic accepting computational histories** of N_L in *one fell swoop*. The formula Φ is polynomially sized and can be obtained at polynomial cost. Thus the *existence* of the $\leq_P$ mapping reduction modeled by the "first funnel" of Figure 16.7 has been demonstrated. □

16.5.3 How P=NP is Obtained if 3-SAT ∈ P?

To acid-test our construction, we now offer an algorithm to decide *any* N_L in NP in *deterministic* polynomial time, **if** we are given a SAT-solver that runs in polynomial time.[26]

[26] This is a magical SAT-solver that does not exist; but should it exist, it will let us collapse NP down to P.

- **Input:** An NDTM N_L and input string w
- **Output:** A P-time decision if $w \in L$
- **Method:** Since we don't know **how long** N_L will run on input w before a decision is reached (except the run is polynomially long for some polynomial), we devise an incremental checking method:
  - Start with $\Phi_0 = \phi_0 \wedge \phi_{ACC}$, and call the P-time SAT solver to see if

it reaches a decision, outputting this decision if so.
- If not, increase i step by step, generating

$$\Phi_i = \phi_0 \wedge \phi_1 \wedge \ldots \phi_i \wedge \phi_{ACC}$$

and checking for a decision (accept/reject).
- The decision is guaranteed in polynomial time, and this algorithm will only make a polynomial number of SAT-solver calls (before a decision is reached), with each call involving a polynomially sized formula, and each such call returning in polynomial time.[27] □

[27] That is, there isn't *a priori* knowledge on how much of the Φ_i formula to generate, before a decision is guaranteed.

16.6 Show that Clique Is NPC: Reduction from 3-SAT

The language of interest is

$$Clique = \{\langle G,k\rangle \ : \ G \text{ is an undirected graph having a } k\text{-clique}\}.$$

16.6.1 Clique is in NP

We will employ the verifier view. The NDTM being built, in addition to receiving its input which is $\langle G,k\rangle$, also asks for a certificate in the form of a list of k nodes. The NDTM then checks whether these k nodes are pairwise connected, accepting exactly when so. This checking procedure runs in polynomial time.

16.6.2 Some Language in NPC Reduces to Clique

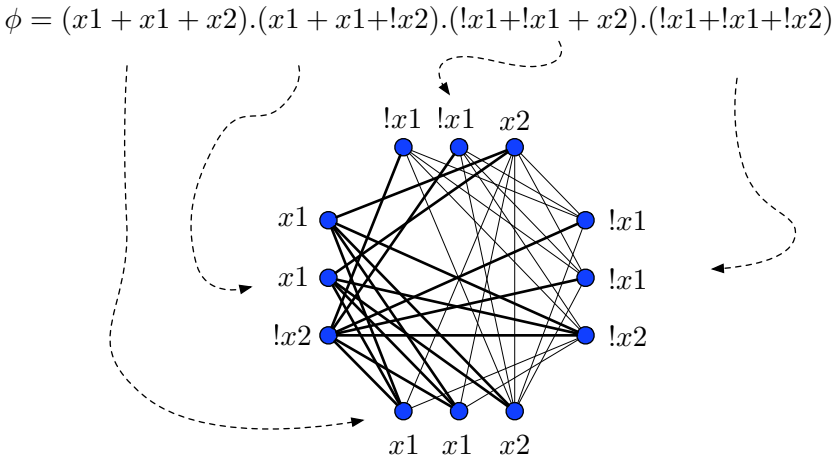


Figure 16.9: The Proof that $Clique$ is NPH using an example formula $\varphi = (x1 + x1 + x2).(x1 + x1 + !x2).(!x1 + !x1 + x2).(!x1 + !x1 + !x2)$. We never connect the nodes within each clause "island" (there are four such islands, each with three nodes). Across each clause island, we draw edges in all possible ways *provided* we never connect a literal and its complement. For visual clarity, we show through dark edges all the edges emanating from the clause island for (x1+x1+!x2) going to all other clause islands. We also show the remaining edges, but using fainter lines.

To show Clique is NP-Hard, we can go by Definition 16.4(b), as we can attempt to reduce 3-SAT to Clique. All we need to do is produce a Clique $graph(\phi)$ given a 3-CNF formula ϕ with k clauses such that $graph(\phi)$

has a k-clique exactly when ϕ is satisfiable. The construction is illustrated in Figure 16.9. The basic idea is this:

- For each clause, draw an "island" of three nodes with the literals in the clause labeling the nodes.
- Never introduce any edges within an island.
- Between two islands of three nodes each, connect pairwise all pairs of literals that are compatible. Two literals are compatible if they are not of the form x and !x. Compatible literals can be satisfied simultaneously.

Observe that the example formula ϕ considered is not satisfiable as it basically is

$$\varphi = (x1 + x2).(x1 + !x2).(!x1 + x2).(!x1 + !x2)$$

where, for any chosen assignment of x1 or x2, one of the clauses will be false. Correspondingly, we cannot find a 4-clique in this graph. The existence of a 4-clique would mean that there are "compatible" literals between all the clauses. That is, we could find a setting to make all these literals true. But that would be a satisfying assignment for ϕ. □

16.7 Complexity Classes, Closing Caveats

Proving a language $L \in \mathsf{NP}$ is often the easy part of an NP-Completeness proof; yet, forgetting this part and merely showing $L \in \mathsf{NPH}$ can not only render your proof incomplete, it can also make it incorrect, as we discuss in §16.7.1. In §16.7.2, we discuss a theorem that has been used in the past by researchers to argue that a language may, after all, be in P.

16.7.1 NP-Hard Problems can be Undecidable (Pitfall in Proofs)

We will now show that the language of Diophantine equations is NP-Hard. *Diophantine*, so-called, is the language of equations that have integer roots, an example being $6x^3z^2 + 3xy^2 - x^3 - 10 = 0$. In general, it is the sum of products of powers of integer variables weighted by integer constants.

This language was shown to be **undecidable** by Yuri Matijasević in a very celebrated theorem [34].[28] This is not a contradiction because NP-Hard only means *at least as hard as* NP (it could be harder, including being undecidable). But claiming that $Diophantine$ is NPC will be tantamount to the claim that something undecidable is decidable![29] Thus, in general, one must not leave an NPC proof unfinished by forgetting to show that the language in question is in NP: it may, after all, not be in NP!

We will now show[30] that $Diophantine \in \mathsf{NPH}$.

[28] One may be tempted to think that this problem is in NP: why not provide the integer roots (the values of x, y, z in the above equation) as a certificate and check that the equation is satisfied? The flaw in this argument lies in being unable to constrain the *size* of the certificate to be polynomially bounded. In particular, the values of the variables can grow without bound.

[29] Recall that all NPC problems are decidable.

[30] This proof comes from Stephen Cook's lecture notes. Please read Cook's bio at `https://amturing.acm.org/award_winners/cook_n991950.cfm`. He won the 1982 ACM Turing Award "*For his advancement of our understanding of the complexity of computation in a significant and profound way. His seminal paper, "The Complexity of Theorem Proving Procedures," presented at the 1971 ACM SIGACT Symposium on the Theory of Computing,* **laid the foundations for the theory of NP-Completeness.**

Theorem 16.7.1: The language $Diophantine$ is NPH:
$Diophantine = \{p : p \text{ is a polynomial with an integral root}\}$.

Proof: We will show that 3-SAT $\leq_P Diophantine$. This means that given a 3-SAT instance

$$\Phi = \phi_0 \wedge \phi_1 \wedge \ldots \phi_i \wedge \phi_{(N-1)}$$

we must produce a Diophantine equation $f(\Phi) = 0$ such that this equation is satisfied if and only if Φ is satisfiable. We now explain the design of f:

- Take any arbitrary clause $\phi_i = (l_{i0} + l_{i1} + l_{i2})$ of Φ. Function f turns ϕ_i into the arithmetic expression

  $$E_i = (g(l_{i0}) \times g(l_{i1}) \times g(l_{i2}))^2$$

  where the mapping g works on literals as follows (this is called the *literal gadget*):
  - If the argument to g is a variable x, then $g(x) = (1 - X)$ where $X \in Nat$ is an integer variable that we introduce in order to model x.[31]
  - If the argument to g is a negated variable $!x$, then $g(x) = X$ where $X \in Nat$.
- Now, function f turns the whole formula Φ into expression $E = \Sigma_{i=0}^{(N-1)} E_i$.
- Example: Function f maps
  $\Phi = (x + y + y) \cdot (x + !y + !y) \cdot (!x + y + y) \cdot (!x + !y + !y)$ into this expression E consisting of sums of squares of three-way products of expressions:
  $((1-X) \times (1-Y) \times (1-Y))^2 + ((1-X) \times Y \times Y)^2 + (X \times (1-Y) \times (1-Y))^2 + (X \times Y \times Y)^2$

[31] We use the uppercase convention for the integer variables that correspond to Booleans.

To argue that this is a **mapping reduction**, we must show that Φ is satisfied iff $f(\Phi) = 0$ has integral roots.

Proof that $f(\Phi) = 0$ is satisfiable iff Φ is satisfiable:

- (1) Φ is satisfiable: Then, there **exists a variable assignment** such that every clause ϕ of Φ has a literal that is true. Let $\phi_i = (l_{i0} + l_{i1} + l_{i2})$ be an arbitrary clause. Without loss of generality, let $l_{i1} = 1$.
  - If l_{i1} is an ordinary variable x, then $g(x) = (1 - X)$, and we can turn the situation "$x = 1$" into the corresponding integer assignment $X = 1$, making $g(x) = 0$.
  - If l_{i1} is $!x$, then $g(!x) = X$, and we can turn the situation "$x = 0$" into the corresponding integer assignment $X = 0$, making $g(!x) = 0$.

  Thus, corresponding to this assignment $l_{i1} = 1$, expression $E_i = 0$. This is the case for every E_i, and so $E = 0$, or that the equation $f(\Phi) = 0$ is satisfied.
- (2) $f(\Phi) = 0$ is satisfiable: This means that each three-way product term $E_i = f(\phi_i)$ must individually be 0 (since we are squaring the three-way products, we avoid the possibility of an E_i and E_j that are non-zero and cancelling each other). This means that each three-way product has one term being 0. There are two cases here:

- This term is of the form X, where $X = 0$: It is clear that X originated from a literal $!x$ via the g mapping. We can choose the value $x = 0$, thus satisfying the clause that $!x$ came from.[32]
- This term is of the form $(1 - X)$, where $X = 1$: It is clear that $(1 - X)$ originated from a literal x via the g mapping. We can choose the value $x = 1$, thus satisfying the clause that x came from.

This method allows us to create a satisfying instance for Φ from the integer roots of $f(\Phi)$.

[32] We don't care how we assign other variables that might be present in the same clause.

16.7.2 The CoNP *and* CoNPC *Complexity Classes*

Definition 16.7.2: A language $L \in$ CoNP if $\overline{L}$ is in NP.

Similarly, L is said to be CoNPC exactly when $\overline{L}$ is in NPC. Figure 16.1 depicts these additional language classes and their likely containments. To illustrate these ideas, consider the following languages which are both subsets of Nat. The language

$$Primes = \{ n : (n > 1) \wedge (\forall p, q : (p \times q = n) \Rightarrow (p = 1 \vee q = 1)). \}$$

The language $Composites = \overline{Primes}$, where the complementation is with respect to positive naturals. $Composites$ is in NP because there exists a P-time verifier for this language, given a certificate which is a pair of natural numbers suggested to be factors. As pointed out in §16.1.2, Pratt proves that $Primes$ is also in NP; he shows this result by demonstrating that there are polynomially long proofs for primes (given a prime p, a polynomially long sequence of proof steps can serve to demonstrate that p is such). Furthermore, he showed that such a proof for $Primes$ can be checked in polynomial time. Now, $Composites$ is in CoNP because $Primes$ is in NP, and $Primes$ is in CoNP because $Composites$ is in NP. The question now is: could either of these languages be NPC?

Theorem 16.7.2 shows that *even if there exists one such language*, then NP and CoNP would become equal—a result thought to be highly unlikely. As pointed out in §16.1.2, the AKS algorithm is proof that $Primes$ is in P (and hence $Composites$ is also in P). Theorem 16.7.2 has helped anticipate the direction in which some of the open problems in this area could be resolved.

Theorem 16.7.2: $\exists L : (L \in$ NPC and $L \in$ CoNP) if and only if NP = CoNP.

Proof:

- ($\Rightarrow$) To show that if $L \in$ NPC and $L \in$ CoNP then NP = CoNP.
  - Assume L is NPC; therefore,
    - L is in NP (Definition 16.4(a), Part 1)

    - Also, L is in CoNP, and thus $\overline{L} \in$ NP (Definition 16.7.2).
    - Thus $\overline{L} \leq_P L$ (Definition 16.4(a), Part 2, the "every language" is $\overline{L}$).
  - To show NP = CoNP, we will take an arbitrary L' in NP and show that it is in CoNP. Then we will take an arbitrary L' in CoNP and show that it is in NP.
    - Consider an arbitrary L' in NP. Then $L' \leq_P L$.
    - From $L' \leq_P L$, it follows that $\overline{L'} \leq_P \overline{L}$. Also $\overline{L'} \leq_P \overline{L} \leq_P L$.[33]
    - Now, since there is an NP decider for L, there is an NP decider for $\overline{L'}$ also, using the above mapping reduction chain. In other words, $\overline{L'} \in$ NP, or L' **is in** CoNP. $(*)$
    - Now, consider an arbitrary L' in CoNP.
    - This means that $\overline{L'}$ is in NP. Since L is in NPC, we have $\overline{L'} \leq_P L$. From this we have $L' \leq_P \overline{L}$.
    - Using the fact that $\overline{L} \leq_P L$, we have $L' \leq_P \overline{L} \leq_P L$, or that there is an NP decider for L', or in other words $L' \in$ NP. $(**)$
    - From $(*)$ and $(**)$, NP = CoNP.
- ($\Leftarrow$) To show that if NP = CoNP, then there exists an L that is in NPC and in CoNP. This is straightforward: consider any NPC language L; it would be in CoNP because L is in NP and NP = CoNP. □

[33] These are basic properties of mapping reductions. If $A \leq_P B$ then there is a polynomial-time function f such that $x \in A \Leftrightarrow f(x) \in B$ (definition of mapping reductions). This also means that $x \in \overline{A} \Leftrightarrow f(x) \in \overline{B}$ or that $\overline{A} \leq_P \overline{B}$. Mapping reductions also compose: if $x \in A \Leftrightarrow f(x) \in B$, and $y \in B \Leftrightarrow g(y) \in C$, then $x \in A \Leftrightarrow g(f(x)) \in C$.

16.8 SAT in Practice

Thousands of verification, counting and optimization problems are currently being modeled in terms of Boolean satisfiability checking.[34] These include 3-SAT formulae generated during the formal verification of programs that are used in a number of safety-critical areas such as embedded systems and computer security. Despite SAT being NP-complete, heuristics invented over the last two decades have achieved several orders of magnitude increase in the efficacy of SAT-solving [9]. SAT solvers nowadays routinely deal with thousands of variables and clauses. Even "hard instances" of SAT that involve more than 50 variables and 200 clauses are routinely solved by SAT solvers.[35]

[34] https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

A SAT solver: Thanks to the work of Mate Soos, you can invoke a SAT solver called CryptoMiniSat in your web browser, with a SAT instance already loaded in the DIMACS format [29]. This format is explained in Figure 16.10 (along with a screenshot; the <-- are added notes for clarity). By clicking the "Play" button on the top right of this browser (near the legend "Ready"), you can invoke the CryptoMiniSat tool on the SAT instance contained in your browser.

[35] These facts show us that the pursuit of NP-Completeness with respect to 3-SAT does not shut the door toward practical uses of SAT. Solvers based on Integer Linear Programming (ILP) are employed in literally tens of thousands of engineering tasks even though ILP is NP-Complete.

Note that SAT solvers, in general, take general CNF formulas that have more than three literals per clause. These general CNF instances can be translated into *equisatisfiable* 3-SAT formulae.

Definition 16.8: A Boolean expression E_1 is equisatisfiable with E_2 if E_1 is satisfiable exactly when E_2 is satisfiable. If $E_1 \equiv E_2$ (*i.e.*, $E_1 \rightarrow E_2$ and $E_2 \rightarrow E_1$), then of course these expressions are equisatisfiable. However, even if $E_1 \not\equiv E_2$, it is possible for these expressions to be equisatisfiable. This fact is illustrated in the callout below entitled **Equisatisfiability without Equivalence**.

Equisatisfiability without Equivalence: Given $(a+b+c+d)$, we can rewrite it into $(a+b+!p).(p+c+d)$ which is **equisatisfiable** to $(a+b+c+d)$. That is, $(a+b+c+d)$ is satisfiable if and only if $(a+b+!p).(p+c+d)$ is satisfiable. In this translation, we introduce a fresh variable p to "bridge" clauses.

These formulae are not logically equivalent. Suppose someone claims otherwise, and claims that this is a tautology: $(a+b+c+d) \equiv ((a+b+!p).(p+c+d))$. It is clear that we can falsify the implication $(a+b+c+d) \rightarrow ((a+b+!p).(p+c+d))$ by picking $a=b=1$, $c=d=p=0$. Thus the formulae are not logically equivalent. They are equisatisfiable: if a=b=0 and c=d=1, then we can choose p=0; if a=b=1 and c=d=0, we can choose p=1. One can work out other combinations suitably.

This idea of obtaining equisatisfiable 3-SAT formulae works for any number of variables. Consider $(a+b+c+d+e+f)$. We can rewrite it into the equisatisfiable formula $(a+b+!p).(p+c+!q).(q+d+!r).(r+e+f)$. This way, a k-CNF clause can be turned into $(k-2)$ 3-CNF clauses.

Exercise 16.7.2, NP-Completeness

1. A SAT instance is given below in the DIMACS format.
   (a) What is the CNF formula captured by this instance?
   (b) By inspection, answer whether the instance is satisfiable, and why.
   (c) If it is not satisfiable, then what is the minimal number of rows that must be deleted before the instance becomes satisfiable? If these rows are not unique, list the first two possible such omissions (of sets of rows), starting from the top of the given listing.
   (d) Check your answer using CryptoMiniSat.

```
c A SAT instance in DIMACS format
c Your task is to determine whether this instance is satisfiable
c
p cnf 5 32
1 2 3 4 5 0
1 2 3 4 -5 0
```

```
1 2 3 -4 5 0
1 2 3 -4 -5 0
1 2 -3 4 5 0
1 2 -3 4 -5 0
1 2 -3 -4 5 0
1 2 -3 -4 -5 0
1 -2 3 4 5 0
1 -2 3 4 -5 0
1 -2 3 -4 5 0
1 -2 3 -4 -5 0
1 -2 -3 4 5 0
1 -2 -3 4 -5 0
1 -2 -3 -4 5 0
1 -2 -3 -4 -5 0
-1 2 3 4 5 0
-1 2 3 4 -5 0
-1 2 3 -4 5 0
-1 2 3 -4 -5 0
-1 2 -3 4 5 0
-1 2 -3 4 -5 0
-1 2 -3 -4 5 0
-1 2 -3 -4 -5 0
-1 -2 3 4 5 0
-1 -2 3 4 -5 0
-1 -2 3 -4 5 0
-1 -2 3 -4 -5 0
-1 -2 -3 4 5 0
-1 -2 -3 4 -5 0
-1 -2 -3 -4 5 0
-1 -2 -3 -4 -5 0
```

2. Consider the set of undirected graphs $\langle G \rangle$ with a set of nodes N and a set of edges $E \subseteq N \times N$ such that we can two-color the graph (meaning no two nodes connected by an edge have the same color).[36] Argue that we can determine a two-coloring for such graphs using breadth-first search.
3. Using Aspvall's algorithm explained in §16.3.2, check whether the following 2-CNF formula is satisfiable. Detail the entire construction. Do not simplify the given formula (do your work on the given formula). $(!a + b) \cdot (!b + c) \cdot (!c + d) \cdot (!d + a)$
4. Check the satisfiability of the 2-CNF formula given in Exercise 3 using CryptoMiniSat.
5. Repeat Exercise 3 with $(!d + a)$ replaced by the conjunction of two clauses: $(!d + !a) \cdot (a + a)$
6. Check the satisfiability of the 2-CNF formula given in Exercise 5 using CryptoMiniSat.
7. Suppose we write a program that traverses a "tape" of n cells, numbered 1 through n. The program performs n traversals of

[36] For instance, a triangle graph G with $N = \{a, b, c\}$ and $E = \{(a, b), (b, c), (c, a)\}$ cannot be two-colored.

the tape, with the ith traversal sequentially examining elements i through n. What is the runtime of such a program in the Big-O notation?

8. A Hamiltonian cycle in a graph with respect to a given node n is a tour that begins at n and visits all other nodes exactly once, returning to n. In a 5-clique, how many distinct Hamiltonian cycles exist? How about in an n-clique?
9. Define the language $HalfClique$ to be the set of input encodings $\langle G \rangle$ such that G is an undirected graph having a clique with at least $n/2$ nodes, where n is the number of nodes in G. Show that $HalfClique$ is NPC. *Hint: Mapping reduction from* $Clique$.
10. The game of Sudoku has been shown to be in NPC. In practice, one can encode and solve Sudoku using SAT solvers. This is also a good way to understand the power of modern SAT solvers. Study the Sudoku solver (MIT license) written by Nicholas Pilkington at `https://gist.github.com/nickponline/9c91fe65fef5b58ae1b0`. Test it on the instance provided as well as a few that you create. Note: This solver will need Python2 (or you may adapt it for Python3).
11. In [15], Cantin et al. prove that the problem of verifying memory coherence is in NPC. Read and summarize this proof in about a page, focusing on the construction of the mapping reduction.
12. Show that a 3-CNF formula

$$\Phi = \phi_0 \wedge \phi_1 \wedge \ldots \phi_i \wedge \phi_{(N-1)}$$

is unsatisfiable *if and only if* **for any variable assignment**, there is one clause ϕ_j with all the literals true and another clause ϕ_k with all the literals false.
Illustration: Consider

$$\Phi = (x+y+y)\cdot(x+!y+!y)\cdot(!x+y+y)\cdot(!x+!y+!y)$$

For any assignment (say $x = 0, y = 1$), we have one clause whose literals are all true, and one clause whose literals are all false (these are respectively $(!x+y+y)$ and $(x+!y+!y)$). If we leave out any one clause, that is not the case, as Φ becomes satisfiable. Now finish the proof.

```
c CryptoMiniSat demo, NOT the full solver
c This is uum8.smt2-stp212.cnf from SAT Competition 2016
c It should solve in about 30 seconds
c You can edit ...
p cnf 1006 3359       <-- problem CNF with 1006 vars, 3359 clauses
2 -3 -159 -214 -374 0 <-- Clause (v2 + !v3 + !v159 + !v214 + !v374)
-2 374 0              <-- Clause (!v2 + v374)
...3354 CLAUSES OMITTED...
-681 -942 -950 0      <-- Each clause ends with 0
1006 0                <-- All clauses are implicitly conjoined
2 0                   <-- This is the 3359th clause (last one)
UNSATISFIABLE <<-- Make last line "-2 0" to get SAT instantly!
```

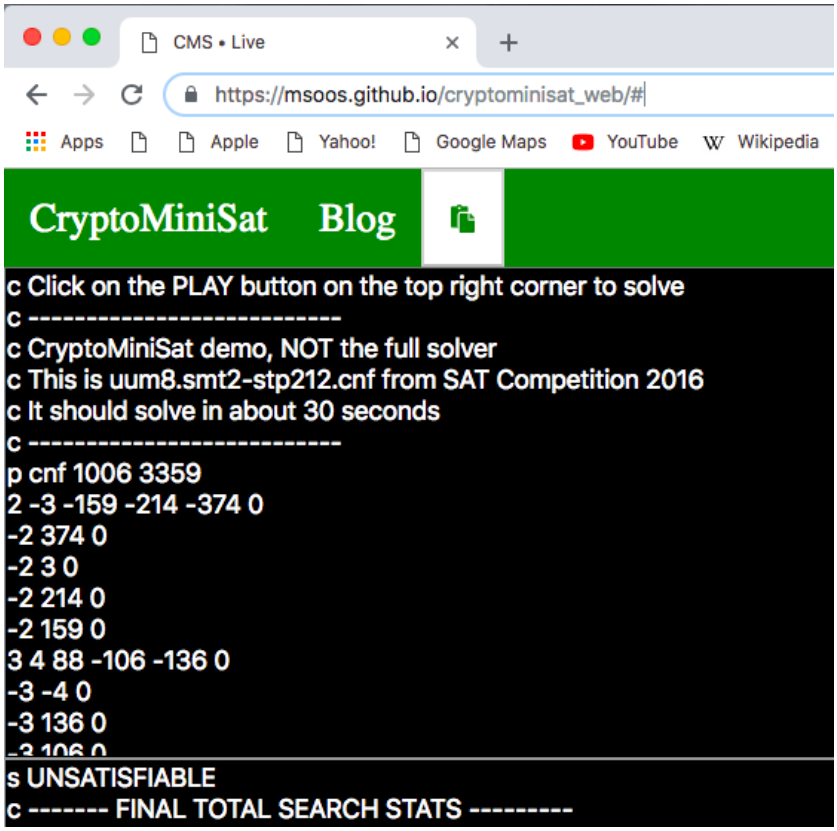


Figure 16.10: CryptoMiniSat, `https://msoos.github.io/cryptominisat_web/`, with DIMACS instance explained at the top, and a screenshot at the bottom.

17
Binary Decision Diagrams as Minimal DFA

Chapter Gist: *We motivate the importance of efficient representation and manipulation of Boolean functions (§17.1). The Binary Decision Diagram (BDD) data structure can be viewed as an optimized representation of minimal DFA for Boolean function on-sets (§17.2). For this idea to pay off in practice, the Boolean variables must be ordered as per their semantic correlation (§17.3). We provide an intuitive overview of what BDD algorithms end up doing: as if the full exponential tree is built and common subexpressions shared (§17.4). We close off with a discussion of BDD sizes including connections with NP-Completeness (§17.5).*

17.1 Boolean Functions in Computing Theory and Practice

The representation and manipulation of Boolean functions is central to computing theory and practice.

x_1	x_2	0	$x_1 \wedge x_2$	$x_1 \wedge \neg x_2$	x_1	$\neg x_1 \wedge x_2$	x_2	$x_1 \oplus x_2$	$x_1 \vee x_2$	$\neg(x_1 \vee x_2)$	$x_1 \equiv x_2$	$\neg x_2$	$x_2 \rightarrow x_1$	$\neg x_1$	$x_1 \rightarrow x_2$	$\neg(x_1 \wedge x_2)$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Figure 17.1: All 2-input truth-tables

For pedagogical purposes, Boolean functions are commonly represented using truth-tables. Figure 17.1 presents a gallery of *all* possible 2-input

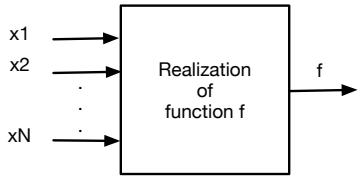


Figure 17.2: An N-input Boolean function

truth-tables. We fix the listing order of inputs x_1 and x_2 to be the standard binary counting order, namely 00, 01, 10, and 11. Each 2-input Boolean function is then characterized by how we fill the output column—we call this the *personality* of the Boolean function. For example, the personality listed under $x_1 \wedge x_2$ is 0001 while for $x_1 \oplus x_2$, it is 0110. Given that there are 2^{2^2} (16) positions to fill (all combinations from 0000 to 1111), there are as many 2-input Boolean functions.

Unfortunately, representing N-input Boolean functions (Figure 17.2) using truth tables is impractical for larger N. For $N = 64$, we would have a truth-table of 2^{64} rows. Even to print that truth-table, we would need about 46 billion tons of paper (assuming 80 truth-table rows are printed per sheet of paper, with each sheet weighing five grams). The hardware industry needs to routinely process Boolean functions with many more inputs than 64. This is to support *formal verification* to detect bugs in the design of critical components such as microprocessors. The first efficient data structure that came to the rescue of the industry is the Binary Decision Diagram ("BDD" for short).

BDDs were introduced in [11] by Randal Bryant. Knuth, one of the giants of computer science, calls BDDs *one of the only really fundamental data structures that came out in the last twenty-five years*. They were instrumental in many of the advances in hardware verification up until (roughly) the year 2000, since when Boolean satisfiability methods have taken the front seat (with BDDs still continuing to play an important role)[1].

As we will see shortly, BDDs are polynomially sized for many of the Boolean functions that arise in practice. There are 2^{2^N} distinct Boolean functions (listing all personalities of length 2^N). This is an astronomical number: there are 256 3-input functions, 65,536 4-input functions, over 4 billion 5-input functions, and over 18 quintillion (billion billion) 6-input functions.[2] Out of these humongous numbers of Boolean functions, those that arise in practice are a miniscule fraction, and out of these, many of them tend to have *polynomially sized BDDs*. More discussions on this topic appear in §17.5.

[1] Bryant's paper is one of the most cited of papers in Computer Science.

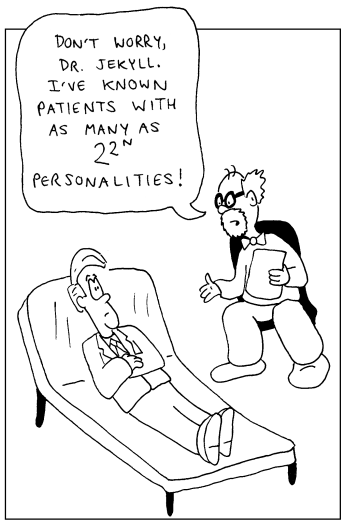


Figure 17.3: Each N-input gate is captured by its personality. Cartoon by Geoff Draper.

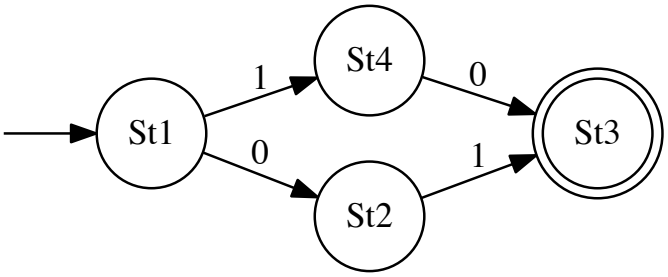
[2] For comparison, a human lives about 3 billion seconds.

17.2 Boolean Functions as Minimal DFA of Their On-Sets

We are studying BDDs in our book because BDDs are nothing but a small variant of minimal DFAs. In particular, BDDs are graph structures that summarize a Boolean function's on-sets (sets of inputs for which the function is true). The on-set of the And function is {11} while that for the Or function is {01, 10, 11}. The on-set of a Boolean function can be treated as a formal language. This language is {01, 10, 11} for an Or-gate and {11} for an And-gate. We will now build minimal DFA for these sets.

Minimal DFA and BDD for Xor*:* The minimal DFA and BDD for the Xor function are now obtained.

```
L_XOR = "(01+10)" # The regexp for the on-set of the XOR function
dotObj_dfa(min_dfa(nfa2dfa(re2nfa(L_XOR))), STATENAME_MAXSIZE=4)
```



We can see that the minimal DFA[3] for Xor accepts 01 and 10. The BDD for Xor can be obtained using an online tool called PBDD[4] that can be invoked as follows (it will open the BDD tool in a new browser tab):

```
import webbrowser

# This is the URL for our PBDD tool that can be opened on a new tab
url = 'http://formal.cs.utah.edu:8080/pbl/BDD.php'
webbrowser.open(url)
```

Type in the following commands and click "build BDD" to obtain the BDD of Figure 17.4:

```
Var_Order : x1 x0
Main_Exp : x1 XOR x0
```

Basically, `Main_Exp` provides the Boolean function under study, and `Var_Order` specifies that the on-set of this function must be built for the language of two-bit words where `x1` comes before `x0`, *i.e.*, the "x1,x0" words are in the on-set language. It can be seen that the BDD is very similar to the minimal DFA if one focuses on all paths that lead to the "1" node. The BDD also shows that the decoding goes on as per `Var_Order`: `x1` is decoded first, and then `x0`, as shown by the edge labels. It is apparent that the "0" leaf node corresponds to the black-hole state.[5] As one

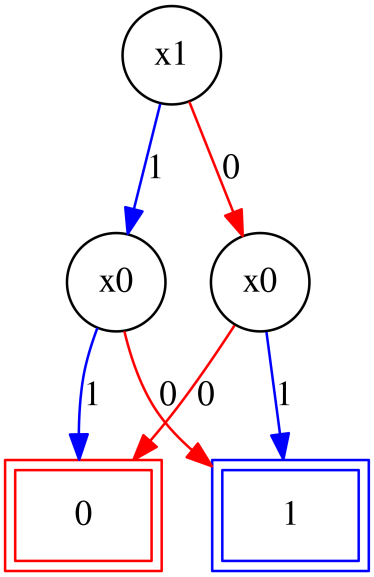


Figure 17.4: BDD for Xor

[3] In most of the DFA presented, we do not show the moves to black-hole states.

[4] Built by Dr. Tyler Sorensen when he was a BS/MS student working with this author.

[5] A state from which we can't get out, as discussed in §4.3.

example, the path 01 ends up in the 1 node (the function output is 1) while the path 00 ends up in the 0 node (the function output is 0).

Minimal DFA and BDD for Or*:* The minimal DFA and BDD for the Or function are now obtained.

```
L_OR  = "(01+10+11)" # Regexp for the on-set of the OR function
dotObj_dfa(min_dfa(nfa2dfa(re2nfa(L_OR))), STATENAME_MAXSIZE=4)
```

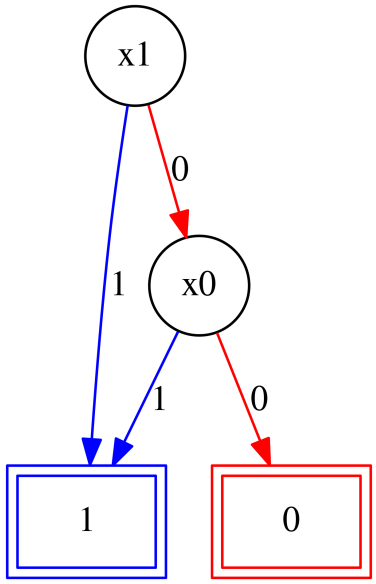
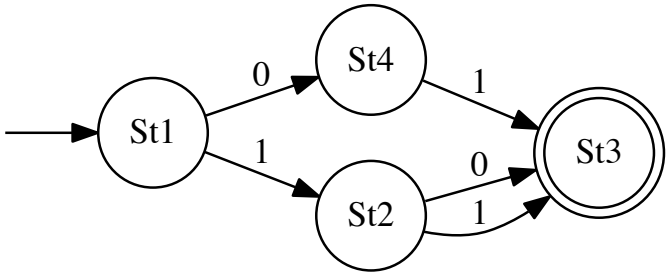


Figure 17.5: BDD for Or



By entering these commands and clicking "build BDD," one obtains the Or BDD of Figure 17.5.

```
Var_Order : x1 x0
Main_Exp : x1 | x0
```

Again it can be seen that the BDD is very similar to the minimal DFA if one focuses on all paths that lead to the "1" node. *But there is one crucial difference*: When `x1` is 1, the BDD directly jumps to the "1" node. The minimal DFA on the other hand goes to state `St2`, but in that state, **regardless of whether a 0 or a 1 comes**, the DFA jumps to the final state `St3`. It is clear that *we can simply drop all such parallel transitions* when interpreting the DFA moves as satisfying the function. In other words, the second bit is a "don't-care" and leads to state `St3` no matter what. That is, `x1=1` ensures that the function output is a 1, ignoring `x0`.

Thus far, the advantage of minimal DFA (or BDD) over truth-tables hasn't been quite apparent. We now proceed to demonstrate that for larger functions, with the right decoding order of the variables, BDDs (and minimal DFA) can indeed be far more compact. On the other hand, truth-tables are guaranteed to be exponential for any N-input Boolean function.

17.3 The Importance of Variable Ordering

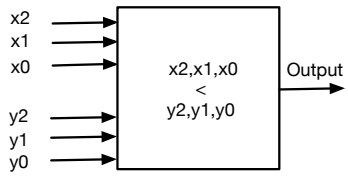


Figure 17.6: The < comparator.

Let us do some more experiments, this time taking a more practical (and non-trivial) function: that of a magnitude comparator for < (Figure 17.6). Suppose there is a 6-input Boolean function modeling a magnitude comparator that compares the binary value coming in through input ports `x2,x1,x0` against the binary value coming through ports `y2,y1,y0`. The function is "<" where "A < B" means the usual "less than" (<) comparison. More specifically, we write "x2,x1,x0 < y2,y1,y0" and we interpret the word x2,x1,x0 using the standard positional binary notation (likewise also for y2,y1,y0). Here are some examples:

- 000 < 001: 0 is < 1 (0 is encoded in binary as 000 and 1 as 001)
- 010 < 110: 2 is < 6 (2 is encoded as 010 while 6 is encoded as 110)
- 110 < 111: 6 < 7

Let us now define a language of strings of length 6 representing the values of x2,x1,x0,y2,y1,y0 written adjacently, such that for those x,y values, the function outputs a 1. Call this language L. For instance, L contains 010101 because 010 is < 101 (i.e. 2 < 5). The reader may verify that the full L language written out as a regular expression (called R below) has 28 strings (out of the $2^6 = 64$ possible length-6 strings):

```
R =  "(000001+000011+000111+001011+001111+010011+010111+011111+\
100101+100111+101111+110111+000010+000101+000110+001010+001101+\
001110+010101+010110+011101+011110+100110+101110+000100+001100+\
010100+011100)"
```

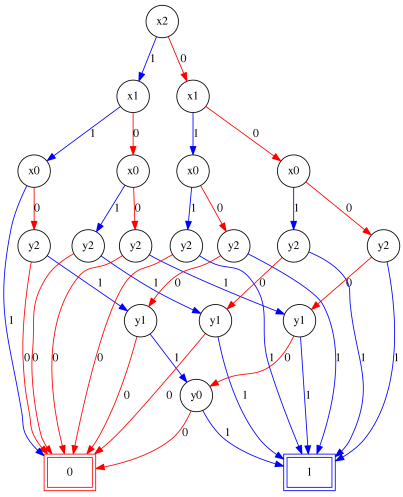
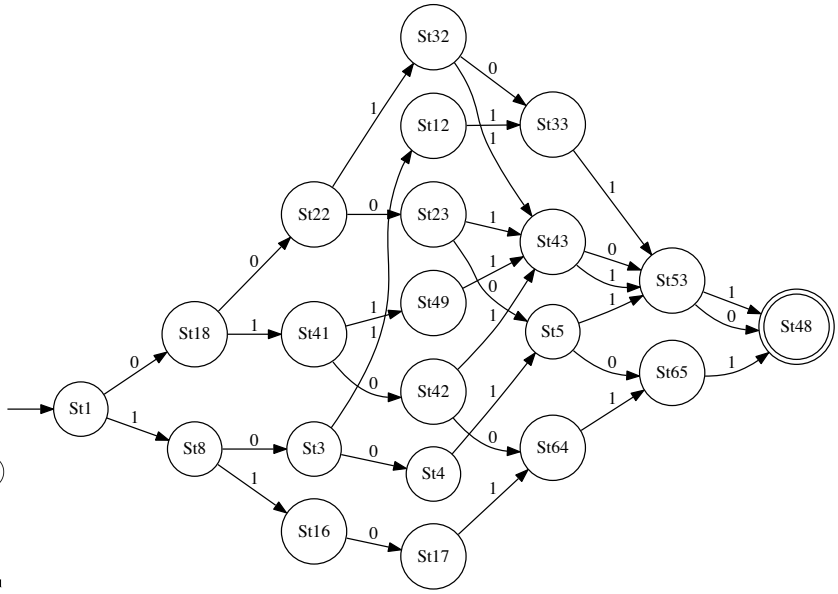


Figure 17.7: BDD for the magnitude comparator: bad input-variable order



The minimal DFA for R using the method shown earlier is above, and the BDD for it is in Figure 17.7 (for `Var_Order` being `x2 x1 x0 y2 y1 y0`). This DFA is in fact exponential in the x2,x1,x0 bits (those are the first three bits to arrive at this machine, and the machine grows exponentially with respect to these inputs). It must represent every x2,x1,x0 combination because the DFA does "not yet know" which y-bits are going to arrive. It then collapses as soon as the y bits come in.

The similarity of the DFA and BDD paths is apparent once again, with the BDD directly jumping to the "0" or "1" node when a decision is made (bypassing the don't-care decodings that minimal DFA end up passing through). However the ability to directly jump (bypassing the don't-cares) still cannot save the BDD from being exponentially big.

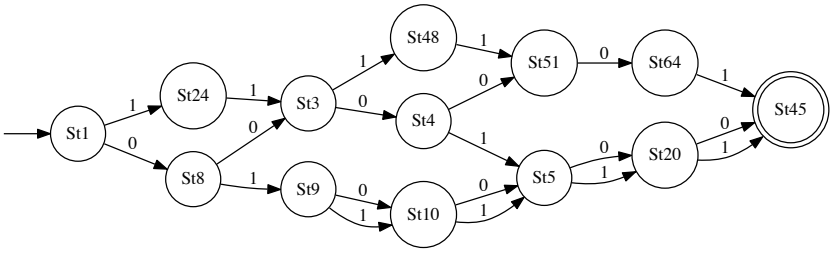
17.3.1 Finding a better input variable order

The main purpose of BDDs is to try and improve over truth-tables, and for that to happen the correct variable order must be presented so that "the Boolean function can decide as quickly as possible." This suggests that we pick the variable order to be x2,y2,x1,y1,x0,y0. This order makes sense because (for example) as soon as we know that x_2 is 1 and y_2 is 0, a decision can be made: < must be false. Likewise, if x_2 is 0 and y_2 is 1, again the decision is made: < must be true. Only otherwise (when $x_2 = y_2$) is it necessary to descend into the remaining bits. **This manner of keeping semantically correlated variables proximally in the BDD** ("quick decoding ability") indeed shows up as a reduced minimal DFA size (and also a reduced minimal BDD size) as we shall now demonstrate.

Let us call the regular expression obtained by interleaving the input bits "Rmix":

```
Rmix="(000001+000111+001101+011111+110001+110111+111101+000101+\
000110+010111+011011+011101+011110+110101+110110+000100+010011+\
010101+010110+011001+011010+011100+110100+010001+010010+010100+\
011000+010000)"
```

The minimal DFA for Rmix is below, and the BDD for it is in Figure 17.8 (for `Var_Order` being `x2 y2 x1 y1 x0 y0`). Again, it is easy to see that the BDD does not "trudge through" the redundant decodings. For instance, in the DFA, state St8 is reached when $x_2 = 0$, and then when $y_2 = 1$ is seen, a pathway of redundant decodings leading to the accept state is entered. Correspondingly, in the BDD, after seeing $x_2 = 0$, we reach a node which decodes y_2, and if $y_2 = 1$, the BDD jumps to the "1" leaf node.



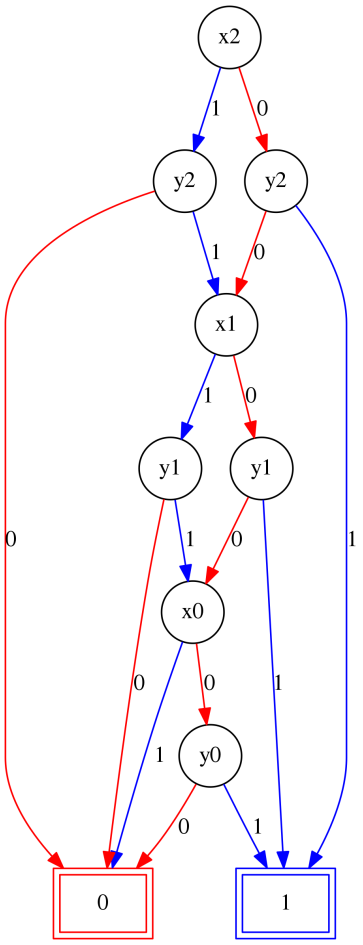


Figure 17.8: BDD for the magnitude comparator: good input-variable order

17.3.2 *Functions with linearly sized BDDs*

As an example of a BDD that is in fact linearly sized, see Figure 17.9 showing the BDD for a 5-input Xor function. Given that the binary Xor function is commutative and associative, we can obtain the 5-input Xor by connecting the input variables *in any order* and *without the use of parentheses*, as shown in the following PBDD commands (`Var_Order` does not matter, so an arbitrary one can be specified):

```
Var_Order : a b c d e
Main_Exp : a XOR b XOR c XOR d XOR e
```

Notice that Xor's parity-checking behavior is quite apparent from its BDD. *All paths from the root* described by an *even* number of 1's ends up in the 0 leaf node, while paths described by an *odd* number of 1's ends up in the 1 leaf node. (Think of a train starting from the BDD's root

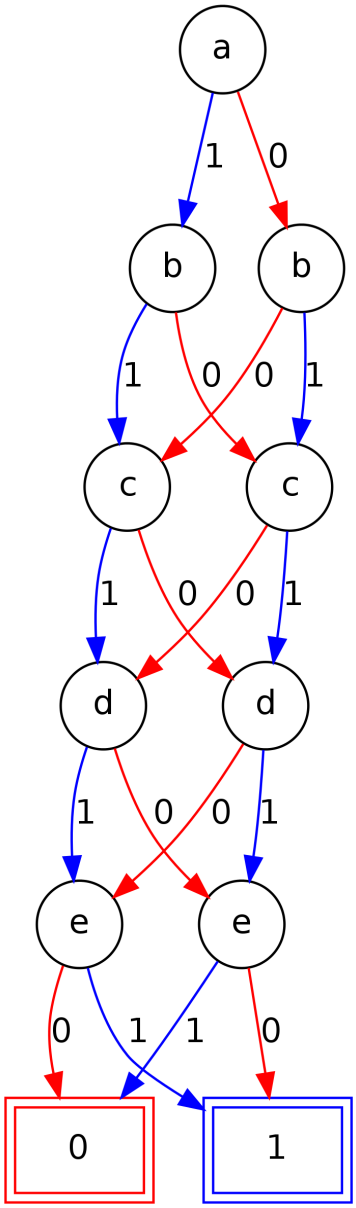


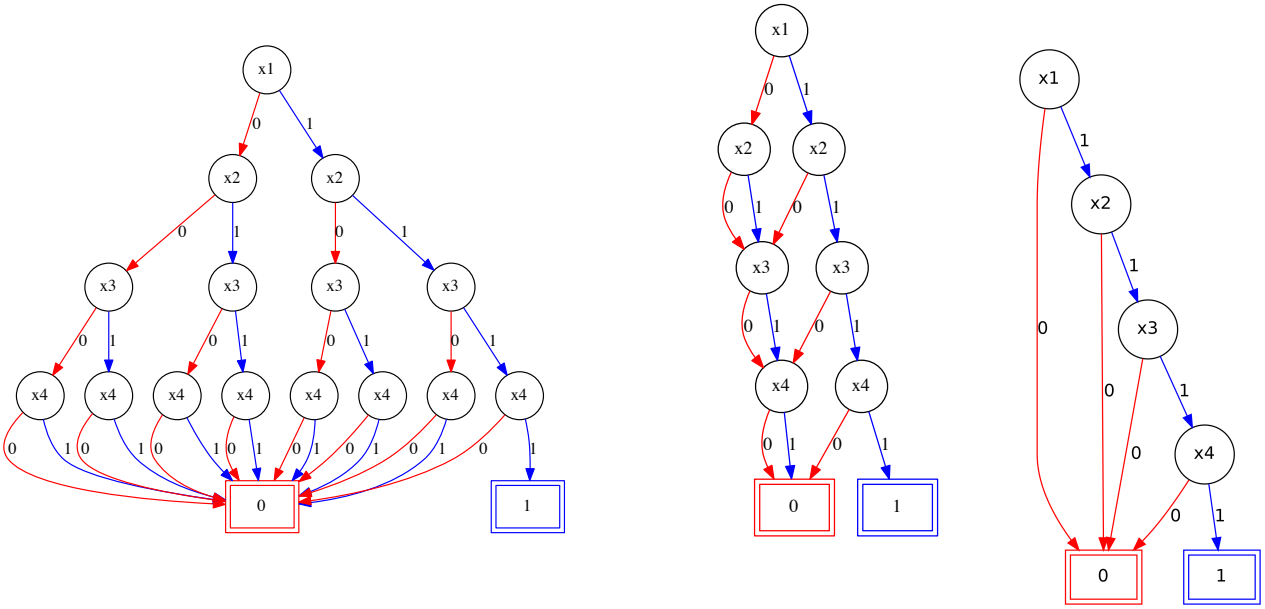
Figure 17.9: BDD for a 5-input Xor gate

node, wanting to head to one of two "leaf stations 0 and 1"; the train gets shunted between two tracks based on the bits that arrive.)

How do BDDs avoid/hide the exponentiality of truth-tables? From Figure 17.8, we can see that there are cases where a BDD can skip over several levels of decoding. After seeing $y_2 = 1$, the BDD must consider x_1, while after seeing $y_2 = 0$, it can directly jump over to the 0 node. This is one way in which the exponentiality is avoided. However, the BDD for a 5-input Xor (Figure 17.9) does not skip levels. Here, we exploit the fact that in a directed graph with node sharings, even though there are a polynomial number of nodes, *there are an exponential number of paths* with each path spelling out a truth-table row. Even though the number of paths is exponential, due to the node sharings, we "forget" which path is taken to reach particular states. This helps us to represent BDDs such as an N-input Xor with linear size.

17.4 From Minimal DFA to BDD: Intuitive Presentation

Figure 17.10: BDDs as Optimized "Decision Trees"



The uniqueness of minimal DFA for a given Boolean function (the Myhill-Nerode theorem, §6.4) does in fact apply to BDDs as well. Figure 17.10 is an attempt to portray this connection conveniently. Notice that the minimal DFA discussed so far were determined by regular expressions (two examples being "R" on Page 271 and "Rmix" on Page 273). These regular expressions are, in turn, completely determined by the sequence of inputs that make up the on-sets.

```
Var_Order : x1 x2 x3 x4
Main_Exp : x1 & x2 & x3 & x4
```

Notice the leftmost graph (almost a full "decision tree" sans the leaf-level nodes) in Figure 17.10. This is the minimal DFA for the four-input And's on-set (if one treats the 1 node as the accepting node and the 0 node as the black-hole state).

The BDD construction algorithm **does not build this graph**, as doing so would make the graph exponential. It instead builds the graph incrementally, bottom up, sharing common subexpressions, and eliminating redundant decodings (see Bryant's paper for details). For the sake of clarity, we explain these steps now as if we are building the whole graph:

- Notice that all the x_4 nodes *except for* the rightmost one have their

left and right child point to the red double-rectangle 0 node.[6] So we first collapse all these x_4 nodes into a single x_4 node whose left and right children go to this 0 node. We leave the last x_4 node alone as a separate entity.

- We repeat the sharing of nodes, now pushing together the three leftmost x_3 nodes into a single node. Again we have to leave the last x_3 node as a separate entity.
- In the same vein, we cannot combine both x_2 nodes, so we leave them as separate entities. This obtains the diagram in the middle of Figure 17.10.
- We now observe that this diagram has *redundant decodings*. More specifically,
  - The leftmost x_4 node has both children being 0. Thus, whether x_4 is 0 or 1, the outcome will be 0. This is because node x_4 lies under the case where x_1 is 0.
  - Thus,
    - We can simply *eliminate* the node x_4, making the left and right children of the leftmost x_3 node point directly to 0.
    - We can also make the left (0) child of the *right-hand side* x_3 *node* point directly to 0.
  - Now we create a situation where the leftmost x_3 node's left and right children point to 0. This causes the leftmost x_2 node to point directly to 0 (via its 0 and 1 children) and also the left child of the right-hand side x_2 node can also point to 0.
  - Proceeding in this manner, we obtain the rightmost diagram of Figure 17.10.

The above steps do not destroy the canonicity of BDDs. Thus, much like minimal DFA, BDDs are also unique (for a given function and variable order).

[6] The last x_4 node has one child being the 1 node.

Myhill-Nerode Theorem in BDD Construction: For a given Boolean function, the generated BDD graphs are isomorphic.[7] This permits fast equality checking between two different Boolean functions. This is based on the Myhill-Nerode theorem already discussed in §6.4.

Bryant proposed a hash-table based representation for BDDs in such a way that isomorphic BDDs map into the same hash-table slot. This way, one can perform Boolean function comparisons in constant time. Also Bryant introduced the *Apply* operation that takes two BDDs and combines them using a Boolean operator. This operator is polynomial with respect to the sizes of the constituent BDDs. Thus if we can build polynomially sized BDDs, Boolean reasoning using BDDs can be done in polynomial time.

[7] Strictly speaking, "for a given variable order" [12].

17.5 On BDD Sizes

For many commonly occurring Boolean functions, the BDDs involved are polynomially sized, and for these problems, Boolean reasoning becomes polynomial-time. Heuristics help choose variable orders that often ensure polynomially sized BDDs. In Figure 17.8, we chose the heuristic of clustering "closely related variables" in the variable order. Variants of this heuristic are employed in practice.

BDDs exhibit another curious fact: their size tends to blow up during BDD manipulations. Measures such as *dynamic reordering* of the variables are often able to minimize many of these bloated BDDs. Such *sifting* algorithms have been well studied in the literature.

From Chapter 16 we know that Boolean satisfiability is NP-Complete. Thus, there shouldn't be a way to get away with satisfiability checking with a lower cost even by using BDDs. This is indeed clinched by the result that discovering a good variable ordering for BDDs is NP-Complete [10].

Exercise 17.5, BDDs

1. Using PBDD produce a BDD for a 4-input Nor function over variables x_1 through x_4. How does it compare with the BDD for the 4-input And function?
2. Using PBDD, produce a BDD for a five-input Xnor circuit (similar to the 5-input Xor on the right-hand side of Figure 17.9). What are the salient differences between these BDDs?
3. Draw a decision tree for a four-input Or gate with inputs x_1, x_2, x_3 and x_4. Then apply the steps suggested in Figure 17.10. Do you obtain a linear-sized BDD at the end of the process? Check your answer by typing in the expression for a four-input Or and using PBDD.
4. Someone encodes the following PBDD file to model the situation:

```
# Implement A < B
# i.e. a2,a1,a0 < b2,b1,b0 where a2/b2 are the MSBs

Var_Order : a2, b2, a1, b1, a0, b0

Main_Exp  : ~a2 & b2 | ~a1 & b1 | ~a0 & b0
```

   (a) Is the above encoding correct? Argue by generating and studying the BDD.

   (b) If there is a flaw in the encoding of `Main_Exp`, fix the flaw, regenerate the BDD, and argue that it now stands corrected.