BOOLEAN SATISFIABILITY

We now continue our dabbling into theoretical computer science with a study of one of its most important problems – Boolean Satisfiability. More precisely, we shall spend some time looking at CNF (Conjunctive Normal Form) Boolean Satisfiability. As is the case with many of the great problems in theoretical computer science, the problem may be understood fairly quickly and yet provide enough for a lifetime of study.

We should first note that the terminology used in this branch of theoretical computer science differs from that used in other branches of the discipline that cover similar problems. We first relate the new terminology to that which we have already studied in this course.

The term "**CNF (Conjunctive Normal Form)**" is used to describe Boolean expressions that we have previously called "Normal Product of Sums" and includes those expressions that we label as "Canonical Product of Sums". CNF == POS.

The term **"DNF (Disjunctive Normal Form)"** is used to describe Boolean expressions that we have previously called "Normal Sum of Products" and included those expressions that we label as "Canonical Sum of Products". DNF == SOP.

One other difference that will be noted in this chapter is that the variables tend to be identified by subscripts, so that we might use "X_1, X_2, and X_3", rather than "X, Y, and Z". This is due to the fact that typical problem instances may involve 50 to 100 Boolean variables and it is somewhat inconvenient to assign that many different letter names.

Boolean Satisfiability Defined

Let B be a Boolean expression of N Boolean variables, X_1 to X_N, expressed in Conjunctive Normal Form (that is – it is a Normal Product of Sums Expression). The expression is said to be **satisfiable** if and only if there is an assignment of truth values to the variables that causes the expression to evaluate to true. The SAT problem (Satisfiability problem) is to determine whether a given Boolean expression is satisfiable.

We now present two examples of Boolean expressions over two Boolean variables, which we shall call X and Y, rather than X_1 and X_2.

$$B_1 = \left(\overline{X} + \overline{Y}\right)\bullet\left(\overline{X} + Y\right)\bullet\left(X + \overline{Y}\right)$$
$$B_2 = \left(\overline{X} + \overline{Y}\right)\bullet\left(\overline{X} + Y\right)\bullet\left(X + \overline{Y}\right)\bullet(X + Y)$$

The expression B_1 is easily seen to be satisfiable, being rendered True by X = 0 and Y = 0.

If X = 0 and Y = 0 then $B_1 = \left(\overline{0} + \overline{0}\right)\bullet\left(\overline{0} + 0\right)\bullet\left(0 + \overline{0}\right)$

$= (1 + 1)\bullet(1 + 0)\bullet(0 + 1)$

$= 1\bullet1\bullet1 = 1$

The expression B_2 is shown to be not satisfiable, but this requires a bit more work.

If X = 0 and Y = 0 then $B_2 = (\overline{0} + \overline{0})\bullet(\overline{0} + 0)\bullet(0 + \overline{0})\bullet(0 + 0)$
$= (1 + 1)\bullet(1 + 0)\bullet(0 + 1)\bullet(0 + 0)$
$= 1\bullet1\bullet1\bullet0 = 0$

If X = 0 and Y = 1 then $B_2 = (\overline{0} + \overline{1})\bullet(\overline{0} + 1)\bullet(0 + \overline{1})\bullet(0 + 1)$
$= (1 + 0)\bullet(1 + 1)\bullet(0 + 0)\bullet(0 + 1)$
$= 1\bullet1\bullet0\bullet1 = 0$

If X = 1 and Y = 0 then $B_2 = (\overline{1} + \overline{0})\bullet(\overline{1} + 0)\bullet(1 + \overline{0})\bullet(1 + 0)$
$= (0 + 1)\bullet(0 + 0)\bullet(1 + 1)\bullet(1 + 0)$
$= 1\bullet0\bullet1\bullet1 = 0$

If X = 1 and Y = 1 then $B_2 = (\overline{1} + \overline{1})\bullet(\overline{1} + 1)\bullet(1 + \overline{1})\bullet(1 + 1)$
$= (0 + 0)\bullet(0 + 1)\bullet(1 + 0)\bullet(1 + 1)$
$= 0\bullet1\bullet1\bullet1 = 0$

We now turn our attention to problem instances that are more in the typical form. As expected, one is satisfiable and one is not.

$B_3 = (\overline{W} + X)\bullet W\bullet(\overline{X} + \overline{Y})\bullet(Y + Z)$
$B_4 = (\overline{W} + \overline{Y})\bullet(\overline{W} + Y)\bullet(W + \overline{Y})\bullet(W + Y)\bullet\overline{X}\bullet\overline{Z}$

For B_3, try W = 1, X = 1, Y = 0, Z = 1 to get $B_3 = (\overline{1} + 1)\bullet1\bullet(\overline{1} + \overline{0})\bullet(0 + 1) = 1$

For $B_4 = (\overline{W} + \overline{Y})\bullet(\overline{W} + Y)\bullet(W + \overline{Y})\bullet(W + Y)\bullet\overline{X}\bullet\overline{Z}$, we try all possibilities.

W = 0	X = 0	Y = 0	Z = 0	$B_4 = (1 + 1)\bullet(1 + 0)\bullet(0 + 1)\bullet(0 + 0)\bullet1\bullet1 = 0$
W = 0	X = 0	Y = 0	Z = 1	$B_4 = (1 + 1)\bullet(1 + 0)\bullet(0 + 1)\bullet(0 + 0)\bullet1\bullet0 = 0$
W = 0	X = 0	Y = 1	Z = 0	$B_4 = (1 + 0)\bullet(1 + 1)\bullet(0 + 0)\bullet(0 + 1)\bullet1\bullet1 = 0$
W = 0	X = 0	Y = 1	Z = 1	$B_4 = (1 + 0)\bullet(1 + 1)\bullet(0 + 0)\bullet(0 + 1)\bullet1\bullet0 = 0$
W = 0	X = 1	Y = 0	Z = 0	$B_4 = (1 + 1)\bullet(1 + 0)\bullet(0 + 1)\bullet(0 + 0)\bullet0\bullet1 = 0$
W = 0	X = 1	Y = 0	Z = 1	$B_4 = (1 + 1)\bullet(1 + 0)\bullet(0 + 1)\bullet(0 + 0)\bullet0\bullet0 = 0$
W = 0	X = 1	Y = 1	Z = 0	$B_4 = (1 + 0)\bullet(1 + 1)\bullet(0 + 0)\bullet(0 + 1)\bullet0\bullet1 = 0$
W = 0	X = 1	Y = 1	Z = 1	$B_4 = (1 + 0)\bullet(1 + 1)\bullet(0 + 0)\bullet(0 + 1)\bullet0\bullet0 = 0$
W = 1	X = 0	Y = 0	Z = 0	$B_4 = (0 + 1)\bullet(0 + 0)\bullet(1 + 1)\bullet(1 + 0)\bullet1\bullet1 = 0$
W = 1	X = 0	Y = 0	Z = 1	$B_4 = (0 + 1)\bullet(0 + 0)\bullet(1 + 1)\bullet(1 + 0)\bullet1\bullet0 = 0$
W = 1	X = 0	Y = 1	Z = 0	$B_4 = (0 + 0)\bullet(0 + 1)\bullet(1 + 0)\bullet(1 + 1)\bullet1\bullet1 = 0$
W = 1	X = 0	Y = 1	Z = 1	$B_4 = (0 + 0)\bullet(0 + 1)\bullet(1 + 0)\bullet(1 + 1)\bullet1\bullet0 = 0$
W = 1	X = 1	Y = 0	Z = 0	$B_4 = (0 + 1)\bullet(0 + 0)\bullet(1 + 1)\bullet(1 + 0)\bullet0\bullet1 = 0$
W = 1	X = 1	Y = 0	Z = 1	$B_4 = (0 + 1)\bullet(0 + 0)\bullet(1 + 1)\bullet(1 + 0)\bullet0\bullet0 = 0$
W = 1	X = 1	Y = 1	Z = 0	$B_4 = (0 + 0)\bullet(0 + 1)\bullet(1 + 0)\bullet(1 + 1)\bullet0\bullet1 = 0$
W = 1	X = 1	Y = 1	Z = 1	$B_4 = (0 + 0)\bullet(0 + 1)\bullet(1 + 0)\bullet(1 + 1)\bullet0\bullet0 = 0$

At this point, we should mention a few characteristics of the Boolean Satisfiability problem that make it, along with many other problems, to have considerable significance in the world of theoretical computer science. The first is that it is a decision problem.

Definition: A **decision problem** is a problem for which the output for any possible input instance is either "Yes" or "No", equivalently "True" or "False", etc. Put this way, the SAT problem is phrased as "Is this Boolean expression satisfiable – answer either Yes or No".

The student will note that the amount of work required to support a "Yes" answer is much less than that required to justify a "No" answer on the SAT problem. Consider the Boolean expression $B_3 = (\overline{W} + X)\bullet(W + \overline{X})\bullet(\overline{X} + \overline{Y})\bullet(Y + Z)$. In order to support the claim that B_3 is satisfiable, we only need to examine the proposed solution. To be more specific, we need to show only one of the possibly many solutions that cause the expression to evaluate to "True". Here we posit that W = 1, X = 1, Y = 0, Z = 1 will cause B_3 to be true, and quickly show that the solution works.

In general, the only way to prove that a Boolean expression of N Boolean variables is not satisfiable is to try all 2^N possible inputs and demonstrate that none of them work. For Boolean expression B_4, we had to try $2^4 = 16$ possible solutions. Although many experts in theoretical computer science believe that this is the only way to show a general instance of the SAT problem not to be satisfiable, nobody can prove that other and faster methods do not exist. Thus we say that this is an "open problem".

Let's look again at the Boolean expression B_4. In so doing, the student will note that this instance of the problem is really quite simple and amenable to some obvious simplifications.

Given $B_4 = (\overline{W} + \overline{Y})\bullet(\overline{W} + Y)\bullet(W + \overline{Y})\bullet(W + Y)\bullet\overline{X}\bullet\overline{Z}$, we note immediately that the expression cannot be true unless X = 0 and Z = 0. This leaves us with the problem of evaluating a much smaller expression $B = (\overline{W} + \overline{Y})\bullet(\overline{W} + Y)\bullet(W + \overline{Y})\bullet(W + Y)$, which is easily seen not to be satisfiable by any assignments to variables W and Y.

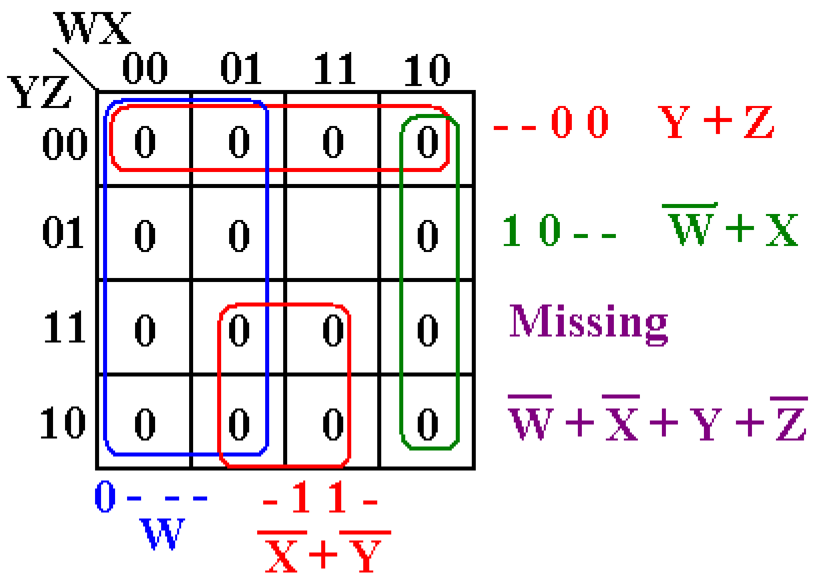
At this point, the student may well ask one of two questions (if not more).

1) What is the relationship of Boolean Satisfiability to the discussions of the previous chapters?

2) How did the author pick these last two formulae and was the solution to B_3 (apparently picked just by luck) really picked at random?

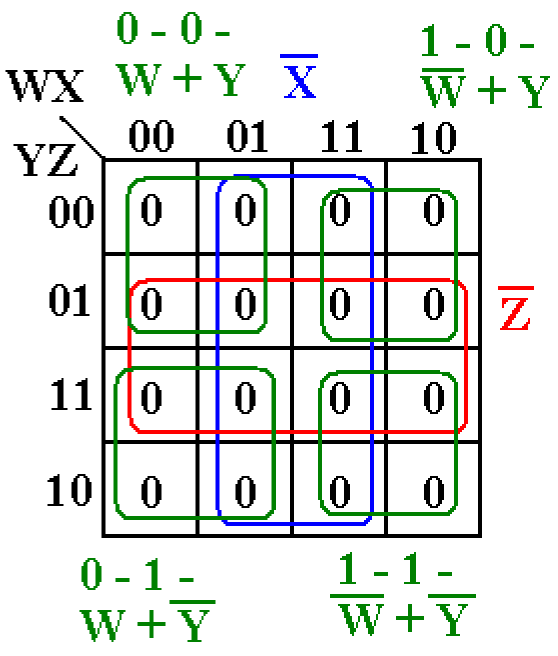
The clever reader will already have decided that there is a very strong connection between Boolean Satisfiability and methods for minimization of Boolean expressions, and also that the author of these notes has used that connection in selecting the solution to B_3. We now show the connection. Each of these expressions was generated by application of an unusual mapping on a Karnaugh map.

We begin with the K-Map that generated Boolean expression B_3. Note that it is a POS K-Map with the entries labeled with a "0". Note also that the square 1101 does not have a 0 in it; so that B_3 can be said to be "missing" the term ($\overline{W} + \overline{X} + Y + \overline{Z}$). This term is made False by the assignment W = 1, X = 1, Y = 0, and Z = 1. For that assignment, B_3 = 1.


WX
YZ
00 01 11 10
00
01
11
10
0 0 0 0
0 0 0
0 0 0 0
0 0 0 0
- - 0 0 Y + Z
1 0 - - $\overline{W}$ + X
Missing
$\overline{W}$ + $\overline{X}$ + Y + $\overline{Z}$
0 - - - W
- 1 1 - $\overline{X}$ + $\overline{Y}$

$$B_3 = (\overline{W} + X)\bullet W \bullet (\overline{X} + \overline{Y})\bullet(Y + Z)$$

Boolean expression B_4 was generated by applying an offbeat mapping to the following K-Map, completely filled with 0's. The more conventional mapping is B4 = 0. The Boolean expression evaluates identically to False, so it cannot be satisfied.



$$\mathbf{B_4 = (\overline{W} + \overline{Y})\bullet (\overline{W} + Y)\bullet(W + \overline{Y})\bullet(W + Y)\bullet \overline{X} \bullet \overline{Z}}$$

We now state the connection between solving the problem of Boolean Satisfiability and the use of methods, such as K-Maps and Q-M, to minimize Boolean expressions. As expected, the lemma applies to CNF expressions (those in **C**onjunctive **N**ormal **F**orm), which we also call Normal Product of Sums.

Lemma: Let B be a CNF Boolean expression over N Boolean variables. B is satisfiable if and only if it cannot be minimized by any method to B = 0 identically.

Proof: Suppose that B is such a Boolean expression. If B can be reduced by some valid minimization procedure to B = 0, then B is identically False, and no assignment of Boolean values to the Boolean variables will make it satisfiable (equal to True). If B cannot be reduced by any valid minimization procedure to B = 0, it is due to the existence of one or more sum terms, also called “clauses”, that are missing from B. (For Boolean expressions of 2, 3, or 4 variables, think of the K-Map as missing one or more 0’s.) Select one of these missing sum terms at random and make the assignment of Boolean values to the Boolean variables that will make that one term evaluate to False. The same assignment will make the given Boolean expression, B, evaluate to True, and thus B is satisfiable.

An “Easy Variant” of the Boolean Satisfiability Problem

We now examine a variant of the SAT problem that superficially appears to be easy in the theoretical sense of computational complexity in that it can be quickly solved. There is a hidden problem, which becomes quite obvious when examined even superficially.

Corollary: Let B be a Canonical SOP expression on N Boolean variables. Then B is satisfiable if and only if B has less than 2^N sum terms (clauses).

Proof: It B has less than 2^N sum terms, then there is a missing sum term that can be used to construct a solution causing B to evaluate to True. If B has exactly 2^N sum terms, then it can be minimized to B = 0 (False) and thus cannot be satisfied.

Remark: This problem is considered easy to solve in that all that the algorithm needs to do is to count the terms in the Canonical POS expression. Let M denote the number of sum terms in the Canonical POS expression; it takes M steps to count these and we say that the counting algorithm takes linear time and claim that the problem is easily solved.

Big Problem: This problem is exponential space. Consider a Boolean expression over 40 Boolean variables. Each sum term will require 40 bits (or 5 bytes) to encode it, one bit for the literal corresponding to each of the 40 variables. There potentially can be 2^{40} terms, so that the memory requirement to store such a formula is $5 \bullet 2^{40}$ bytes, or five terabytes, otherwise stated as 5120 gigabytes. Current disk drives could not store the complete Canonical POS description of a Boolean expression that is not satisfiable.

We shall conclude this chapter with a discussion intended to suggest the reasons that Boolean Satisfiability is such a significant problem in theoretical computer science. To do this, we must take a detour and present a few new topics.

Time Efficiency of Algorithms
If we want to gain any understanding of the significance of the Satisfiability problem, we must first pay some attention to the idea of time-efficiency of algorithms. Intuitively we know that some algorithms take more "work" than others, for example the manual algorithm for taking the square root of an integer takes more work than that for long division.

Obviously, the amount of work done by an algorithm depends on the size of the input. For example, it would take longer to add a list of one hundred numbers than it would to add a list of three numbers; although the basic algorithm is the same. For some problems, the measure of the size of the input is obvious, for other problems it is a bit trickier.

1) For addition of a list of integers, we say that the size is the number of integers to add.
2) For sorting a list of integers, we again say that the size is the number of integers in the list.
3) For a problem involving trips to a number of cities, we say that the size of the input is the number of cities to be visited.
4) For problems on integers, such as factoring and testing for the number being prime, the size is the number of digits in the integer; 1,234,567 is of size 7.
5) For most variants of SAT, we say that the size is the number of Boolean variables used in defining the Boolean expression.

While the precise definition of size of the input is fairly tricky (and used in the above silly discussion of the "easy variant" of SAT), it is usually sufficient to rely on common sense in determining the input size. In most of theoretical computer science, we denote the input size by the variable "N"; say N = the number of integers to be sorted.

We think of two functions to describe the efficiency of algorithms. These might be called

T(N)	a measure of the time for the algorithm as a function of the size of the input
M(N)	a measure of the amount of memory required as a function of the input size.

As it develops, we discover that time efficiency seems to be more important that the amount of memory used by the algorithm. As a result, we usually pay little attention to the space efficiency of an algorithm and focus on the time complexity.

The Measure of Algorithm Time
If we ask how long an algorithm takes to execute, we immediately might think of clock time. There are a number of good reasons not to use time in seconds as a measure of T(N) – the time taken by the algorithm. These all relate to the fact that the time in seconds to run an algorithm depends on too many factors external to the specification of the algorithm. Some of the factors affecting the total running time of an algorithm include:
1) the number of computer instructions executed to complete the algorithm, and
2) the speed of the computer CPU (Central Processing Unit), and
3) the effectiveness of the computer's instruction set, and
4) the amount of memory in the computer, and
5) the fraction of time that the Operating System has scheduled for the program.

In the study of algorithms, it is better to focus on the algorithm itself and not on factors, such as CPU speed and scheduling by the Operating System, that are external to the algorithm. The way to do this is to count the number of steps that the algorithm must execute for a given instance of the problem and specify how that varies as a function of the problem size.

In order to simplify the analysis, we often choose not to count every instruction or step in the execution of the algorithm, but only the **basic operations** (I often call these “basic steps”). The idea is that a count of these basic operations is representative of the total number of instructions executed and can form the basis for an accurate time estimate.

Consider the following code fragment, written in FORTRAN (just to be different). Note that FORTRAN arrays are indexed 1 to N, with N being the size of the array.

```
C      EXECUTE THE LOOP ENDING AT STATEMENT 500
       DO 500 J = 1, N
       C(J) = A(J) * B(J)
   500 CONTINUE
```

We should note that the loop above is executed N times, once for every value of J between 1 and N inclusive. We can list the steps for each loop.

1. Assign a value to J
2. Multiply A(J) by B(J)
3. Assign a value to C(J)
4. Test the value of J and loop if $J < N$. If $J = N$, the loop terminates.

We can express the time efficiency of this code fragment as $T(N) = 4\bullet N + K$, where K is some constant number of steps indicating the work to set up the loop. We generally focus our attention on the type of the function T(N); here it is a **linear function**. From this, we deduce that increasing the size of the input by a factor of 10 will increase the number of steps by almost exactly a factor of 10.

There are three major classes of time complexity functions:

logarithmic	T(N) is a logarithmic function, say $T(N) = 7\bullet\log(N)$
polynomial	T(N) is a polynomial, usually either a linear function, quadratic function, or cubic function.
exponential	T(N) is an exponential function. In our discussion of SAT, we mentioned that it took 2^N tries to show a formula not satisfiable.

To see the significance of this classification, we speak loosely and suppose an algorithm that has time complexity T(N) and that takes 60 seconds to solve a problem of size 10. What about the time to solve a problem of size 100? We consider a number of cases.

T(N) is logarithmic, say $T(N) = 60\bullet\log(N)$. Then for N = 100, we have
$T(N) = 60\bullet\log(100) = 60\bullet 2 = 120$, and the time doubles.

T(N) is linear, say $T(N) = 6\bullet N$. Then for N = 100, we have
$T(N) = 6\bullet 100 = 600$, and the time increases by a factor of 10.

T(N) is quadratic, say $T(N) = 0.6\bullet N^2$. Then for N = 100, we have
$T(N) = 0.6\bullet(100)^2 = 6000$, and the time increases by a factor of 100.

T(N) is cubic, say $T(N) = 0.06\bullet N^3$. Then for N = 100, we have $T(N) = 0.06\bullet(100)^3 = 0.06\bullet 10^6 = 6\bullet 10^4 = 60000$, and the time increases by a factor of 1000.

T(N) is exponential, say $T(N) = 0.0586\bullet 2^N$. Then for N = 100, we have
$T(N) = 0.0586\bullet 2^{100}$, and have to struggle to evaluate 2^{100}.

We approximate the number 2^{100} by noting that log(2) = 0.30103, so that
$2^{100} = (10^{0.30103})^{100} = 10^{30.103} = 10^{0.103}\bullet 10^{30} \approx 1.268\bullet 10^{30}$, as log(1.268) = 0.1031.
The time increases by a factor of approximately $120\bullet 10^{28} / 60 = 2\bullet 10^{28}$.

For those of morbid curiosity, we note that a year is approximately equal to $3.15\bullet 10^7$ seconds so that our algorithm now takes only $4\bullet 10^{21}$ years to execute. The age of the universe is usually considered to be about 10^9 years, though some consider it much smaller.

The Problem Class NP

In theoretical computer science, the term NP stands for "**N**ondeterministic **P**olynomial" and refers to the operation of a nondeterministic Turing machine. Later in this course, we shall have something (though not very much) to say about Turing machines (named after Alan Turing, the very famous British mathematician who developed this important concept in the understanding of computation) but at this point, we just accept "NP" as a label.

To see what we are trying to get at with the label "NP", consider the Boolean expression $B_3 = (\overline{W} + X)\bullet W\bullet(\overline{X} + \overline{Y})\bullet(Y + Z)$. This is an **instance** of the SAT problem, a particular example of an expression for which we can ask whether or not it is satisfiable. We have seen that the answer for this instance is "Yes" and the solution is W = 1, X = 1, Y = 0, and Z = 1. What is important is that SAT is a problem in which it is quite easy to verify a "Yes" answer; technically that a correct solution can be verified by a procedure that takes a number of steps, T(N), that is a polynomial function of the size of the input. The precise term for this concept is "**polynomial time verifiability**". It is this concept of verifying an answer in polynomial time that the problem class NP is attempting to isolate.

Definition: Informally, we define the problem class NP as follows.
1) NP includes only decision problems, that is – problems that can be answered either "Yes" or "No".
2) For any instance of a problem in this class, it is possible to verify a correct "Yes" answer fairly efficiently, strictly speaking in polynomial time.
3) No statement is made about the amount of time to verify a correct "No" answer.

Reduction
Suppose we are given two problems, P and Q. We assume that we know how to solve problem P and want to solve problem Q. There are a number of ways that we can do this, including the process called "**reduction**". Informally, we say that problem Q reduces to problem P, if we can take an instance of problem Q, change it to an instance of problem P, solve problem P, and use that solution to build a solution to problem Q.

This may seem a bit mysterious, so we illustrate with an example of problem reduction. Since all of the interesting instances of problem reduction require some mathematical sophistication to understand, we illustrate the problem with a rather silly example.

Let P be the problem of multiplying two integers; given X and Y, determine $Z = X \bullet Y$.
Let Q be the problem of squaring an integer; given X, determine $W = X^2$.

Problem Q is easily seen to be reducible to problem P; as a matter of fact, that is how we solve problem Q. We do not have a separate procedure for squaring a number, we just multiply it by itself. Thus, reduce squaring to multiplication by setting $Z = X \bullet X$, thus producing $Z = X^2$, and then set $W = Z$.

Problem P can be reduced to problem Q, although this seems a bit strange. Recall that
$(X + Y)^2 = X^2 + 2XY + Y^2$, and that
$(X - Y)^2 = X^2 - 2XY + Y^2$.

From this we can see that $(X + Y)^2 - (X - Y)^2 = 4XY$, so that
$X \bullet Y = [(X + Y)^2 - (X - Y)^2] / 4$ and the problem of producing a product is reduced to the problem of squaring two numbers. We have reduced problem P to problem Q and reduced problem Q to problem P. As in much of theoretical computer science, we are interested in proving that we can make the two reductions without actually having a desire to use either.

Reduction of Decision Problems
Let's now focus on problem reduction as applies to decision problems and restate the steps involved in reducing one decision problem (call it Q) to another decision problem (call it P). Recall that reduction applies to instances of a problem, and we say that a problem Q is reducible to problem P if and only if every instance of Q can be reduced to an instance of P.

1) Take the instance of Q and apply the specific reduction algorithm to change that into an instance of problem P.
2) Solve problem P for the given instance.
3) The answer to problem P is "Yes" if and only if the answer to problem Q is "Yes". This last result is guaranteed by the design of the reduction algorithm.

Definition: Let P_1 and P_2 be two decision problems. We say that problem P_1 is **polynomially reducible** to problem P2 if there exists a polynomial-time algorithm that converts each instance of P_1 to an instance of P_2 with the property that the answer to problem P_1 is "Yes" if and only if the answer to problem P_2 is "Yes".

What makes Boolean Satisfiability such a significant problem is the fact that a number of very significant problems can be said to be equivalent to it. Put another way, we have a large class of problems (it is called NP-Complete), such that if Q is in that class then both

1) Q reduces in polynomial time to SAT, and
2) SAT reduces in polynomial time to Q.

Here are a few of these important problems. The reference Garey & Johnson [R11] lists over 320 such problems.

1 **Hamiltonian Cycle** Given a graph G = (V, E), is it possible to find a simple cycle that includes every vertex exactly once, except that the start vertex is also the end vertex.

2. **Traveling Salesman** Given a weighted graph G = (V, E, W), that is a complete graph (with each vertex adjacent to every other vertex), is it possible to specify a Hamiltonian cycle of total weight less than a specific value.

3. **Partition** Given a set X such that each element $x \in X$ has an associated size s(x). Is it possible to partition the set X into two subsets with exactly the same total size.

4. **0/1 Knapsack** Given a set X of items, each having size s(x) and value v(x), is it possible to create a subset $Y \subseteq X$ such that the total size of Y is less than a fixed value S while the total value of the items in Y is at least another fixed value V.

5. **3-Coloring** Given a graph G = (V, E), is it possible to assign one of three "colors" to each of the vertices so that no two adjacent vertices have the same color.

6. **Clique** Given an (N, M)-graph G and an integer L, $2 < L < N$, determine whether G contains a clique of size greater than or equal to L; that is a subgraph isomorphic to K_L, the complete graph on L vertices.

7. **Dominating Set** Let G be an (N, M)-graph. A **dominating set** D is a set of vertices in G such that every vertex in V(G) is either in D or adjacent to a vertex in D. Given an integer L, determine whether or not G has a dominating set containing not more than L vertices.

8. **Vertex Cover** Let G = (V, E) be an undirected graph. A **vertex cover** is a set C of vertices such that every edge in E(G) is incident to at least one vertex in C. Given an integer L, determine whether G has a vertex cover containing not more than L vertices.

9. **3-SAT** Given a Boolean expression in Conjunctive Normal Form (CNF) such that each clause contains literals for exactly three variables, determine whether it is satisfiable.

10. **Minimization to 0** Given a Boolean expression in Product of Sums Form. Is it possible to minimize this expression to identically false, B = 0?