SAT and SMT Solving

Many verification and testing problems involve the following problem:

"can a given statement be true, and if so, how?"

This is the *satisfiability* problem, and if it is not possible, the statement is *unsatisfiable*. A model checker finding a counterexample, or a symbolic executor statically analysing code, reduces its task to a satisfiability question, and a positive answer becomes a counterexample in the model checker or a test input in the symbolic executor. The problem is NP-complete, so no algorithm is known to answer it quickly for all formulas [1]. Yet, current solvers answer this problem for formulas with millions of variables in industry [1]. In this exercise sheet, we use such a solver, and then study how it works, in four sections:

1. *Using an SMT solver* (Section 1). Install the SMT solver Z3, and write formulas in SMT-LIB, the input language of SMT solvers.
2. *Applications* (Section 2). Choose the versions of software packages as a package manager does, and compare two access policies of a cloud service.
3. *Normal forms* (Section 3). Convert a formula into the input form of SAT solvers: conjunctive normal form (keeping the size linear with Tseitin's encoding).
4. *Search* (Section 4). Find a satisfying assignment with the DPLL algorithm.

1 Using an SMT solver

A SAT solver decides propositional formulas, formulas whose atoms are Boolean variables such as $p \wedge q \vee (r \vee (p \to s))$. In contrast, an SMT solver (*satisfiability modulo theories*) also decides formulas whose atoms are predicates $P(x, y)$ where the arguments to predicates can be variables (x and y) that range over integers, strings, real numbers, or other domains, or terms such as $f(x, y)$, built with function symbols. The formula

$$x + y = 5 \wedge ((r = \texttt{"hello"} \wedge s = \texttt{" world"}) \to r +\!\!+ s = \texttt{"hello world"})$$

is an example, where $+\!\!+$ is the concatenation of strings. In this section, we install the SMT solver Z3 and write formulas in SMT-LIB, the standard input language of SMT solvers [4]. Section 2 uses Z3 on two problems of software engineering, and Sections 3 and 4 explain the basics of how these solvers find solutions.

Exercise 1 (Install Z3).

Part 1 — *Install Z3.* Install Z3 by following the instructions for your platform.

macOS. With Homebrew:

```
$ brew install z3
```

Linux (Debian, Ubuntu).

```
$ sudo apt update
$ sudo apt install z3
```

Windows. Use one of the two options below.

- *Zip file.* Open https://github.com/Z3Prover/z3/releases, and under *Assets* of the latest release, download the zip file for your processor. To find your processor, open *Settings* → *System* → *About*, and read *System type*.
  - `z3-<version>-x64-win.zip`: x64-based processor (Intel or AMD, most PCs),
  - `z3-<version>-arm64-win.zip`: ARM-based processor (such as Snapdragon),
  - `z3-<version>-x86-win.zip`: 32-bit Windows.

  Ignore the `.whl`, `.jar` and `.nupkg` files. Those are for using Z3 from Python, Java and .NET, not for the command `z3`. Unpack the zip file. Its `bin` folder contains `z3.exe`. Add this folder to the `PATH`: search for *environment variables* in the Start menu, open *Edit environment variables for your account*, select `Path`, click *Edit*, then *New*, paste the full path of the `bin` folder, and confirm with *OK*. Then open a new terminal.
- *Windows Subsystem for Linux (WSL).* WSL runs Linux inside Windows. Open PowerShell as administrator, run `wsl -install`, and restart the computer (see https://learn.microsoft.com/windows/wsl/install). Then open *Ubuntu* from the Start menu, and follow the Linux instructions above in it.

Part 2 — *Check that Z3 runs.* Run the command below, which prints the version of Z3.

```
$ z3 --version
Z3 version 4.15.4 - 64 bit
```

Exercise 2 (A first SMT-LIB file). SMT solvers read files in the standard language SMT-LIB [4, 3]. Save the file below as `first.smt2`.

```
(declare-fun a () Int)
(declare-fun f (Int Bool) Int)
(assert (> a 10))
(assert (> (f a true) 100))
(check-sat)
(get-model)
```

Part 1 — *Match the commands.* Match each SMT-LIB expression with its meaning by writing its number in the brackets. Expressions 1–6 are the lines of `first.smt2`, and expressions 7–14 are used in Section 2. An example is provided.

	SMT-LIB		Meaning
1	`(declare-fun a () Int)`	( )	adds the formula $f(a, true) > 100$
2	`(declare-fun f (Int Bool) Int)`	( )	the formula $b \to a > 10$
3	`(assert (> a 10))`	( )	the formula $\neg b$
4	`(assert (> (f a true) 100))`	( )	prints a model of the assertions
5	`(check-sat)`	(3)	adds the formula $a > 10$
6	`(get-model)`	( )	declares a, a function without arguments
7	`(declare-const b Bool)`	( )	prints the values of a and b in the model
8	`(=> b (> a 10))`	( )	the formula $b \vee a > 10$
9	`(or b (> a 10))`	( )	checks whether the conjunction of the assertions is satisfiable
10	`(and b (> a 10))`	( )	declares b, a Boolean variable
11	`(not b)`	( )	is true if the string s starts with `"ab"`
12	`(get-value (a b))`	( )	the formula $b \wedge a > 10$
13	`(define-fun g () Bool (> a 10))`	( )	declares f, of an integer and a Boolean
14	`(str.prefixof "ab" s)`	( )	names the formula $a > 10$ as g

Part 2 — *Read the question.* Together, the lines of `first.smt2` ask Z3 one question. Below, $\mathbb{Z}$ is the set of integers and $\mathbb{B} = \{true, false\}$. Mark with a checkmark (✓) the formula that `first.smt2` asks Z3 to decide:

( ) $\forall a \in \mathbb{Z}.\ \forall f : \mathbb{B} \times \mathbb{Z} \to \mathbb{Z}.\ \big(a > 10 \to f(a, true) > 100\big)$

( ) $\exists a \in \mathbb{Z}.\ \exists f : \mathbb{Z} \times \mathbb{B} \to \mathbb{Z}.\ \big(a > 10 \wedge f(a, true) > 100\big)$

( ) $\exists a \in \mathbb{Z}.\ \exists f : \mathbb{Z} \times \mathbb{B} \to \mathbb{Z}.\ \big(a > 10 \vee f(a, true) > 100\big)$

Mark with a checkmark (✓) what running `z3 first.smt2` does:

( ) It only checks that the file has no syntax errors, and prints nothing.

( ) It evaluates each assertion separately, and prints `true` or `false` for each one.

( ) It searches for a value of a and a function f that make both assertions true, prints `sat` if they exist, and then prints such a value and function.

Part 3 — *Read the report.* Run `z3 first.smt2`. Each item below is a comment on a line of Z3's output. Write the letter of each comment after the `;;` of the line it explains.

a) The conjunction of the two assertions is satisfiable.

b) The model starts here: one definition per declared symbol.

c) In the model, the variable `a` has the value 11.

d) In the model, `f` is a function whose two arguments Z3 has named `x!0` and `x!1`.

e) For every pair of arguments, the value of `f` is 101.

```
sat                                          ;; (a)
(                                            ;; ____
  (define-fun a () Int
    11)                                      ;; ____
  (define-fun f ((x!0 Int) (x!1 Bool)) Int   ;; ____
    101)                                     ;; ____
)
```

Part 4 — *Write a formula.* Write a file `uf.smt2` for the formula $F(10) = 1 \wedge F(1) = 2$, where F is a function from integers to integers that Z3 may choose, and run Z3. The output below comes from the file in the solution. Yours may differ if you write the assertions differently, so answer from the output below. In it, `(ite` c t e`)` is "if c then t else e". Complete the values of F in this model.

```
sat
(
  (define-fun F ((x!0 Int)) Int
    (ite (= x!0 1) 2
      1))
)
```

$F(10) =$ ____ $F(1) =$ ____ $F(7) =$ ____

2 Applications

Many design questions in software engineering ask whether some object with given properties exists: a set of package versions that fit together, or a request that one access policy grants and another denies. This section formalises two such questions and lets Z3 answer them.

2.1 Choosing package versions

A package manager such as `micromamba` installs a requested package together with its dependencies ensuring that each package version works well with the rest. Choosing these versions "can be expressed as a Boolean satisfiability problem (SAT)", and mamba and micromamba solve it with the SAT solver libsolv [5]. The table below lists the packages of a small, invented channel.

Package version	Depends on
python 3.11, python 3.12	—
numpy 1.26	python 3.11 or python 3.12
numpy 2.1	python 3.12
scipy 1.11	numpy 1.26
scipy 1.14	numpy 2.1
oldplot 0.9	python 3.11

Exercise 3 (Choosing package versions with Z3).

Part 1 — *Formalise the constraints.* Let us assume that at most one version of each package can be installed and let us encode one Boolean variable per package version, making it true if the version is installed. We write $numpy_{1.26}$ for the variable of numpy 1.26, and similarly for the other versions. Mark with a checkmark (✓) the formula that states that numpy 1.26 depends on python 3.11 or python 3.12:

( ) $(python_{3.11} \vee python_{3.12}) \rightarrow numpy_{1.26}$

( ) $numpy_{1.26} \rightarrow (python_{3.11} \vee python_{3.12})$

( ) $numpy_{1.26} \wedge (python_{3.11} \vee python_{3.12})$

Mark with a checkmark (✓) the formula stating that at most one python is installed:

( ) $(python_{3.11} \vee python_{3.12}) \wedge \neg(python_{3.11} \wedge python_{3.12})$

( ) $\neg python_{3.11} \wedge \neg python_{3.12}$

( ) $\neg(python_{3.11} \wedge python_{3.12})$

Part 2 — *Encode the channel.* A user requests scipy, in any version. Complete the file `env.smt2` below with one assertion per line, and run `z3 env.smt2`. An example is provided in each group of assertions.

```
(declare-const python-3.11 Bool)
(declare-const python-3.12 Bool)
(declare-const numpy-1.26 Bool)
(declare-const numpy-2.1 Bool)
(declare-const scipy-1.11 Bool)
(declare-const scipy-1.14 Bool)
(declare-const oldplot-0.9 Bool)
; At most one version of each package.
(assert (not (and python-3.11 python-3.12)))
_____________________________________
_____________________________________
; Dependencies.
(assert (=> numpy-1.26 (or python-3.11 python-3.12)))
_____________________________________
_____________________________________
_____________________________________
_____________________________________
; Request: scipy, in any version.
_____________________________________
(check-sat)
(get-value (python-3.11 python-3.12 numpy-1.26 numpy-2.1
            scipy-1.11 scipy-1.14 oldplot-0.9))
```

Z3 answers:

```
sat
((python-3.11 true)
 (python-3.12 false)
 (numpy-1.26 true)
 (numpy-2.1 false)
 (scipy-1.11 true)
 (scipy-1.14 false)
 (oldplot-0.9 false))
```

Z3 installs scipy 1.11, although the newest scipy can also be installed. Which versions are installed with the newest scipy? python=_____, numpy=_____ and scipy=1.14.

Part 3 — *Prefer the newest versions.* The Z3 command `(assert-soft` ψ`)` adds a *soft constraint*. Among the models of the assertions in a `.smt2` file, Z3 returns one that satisfies as many soft constraints as possible [6]. Write the formulas that Z3 should satisfy if possible:

_____ _____ _____

Add them as soft constraints before `(check-sat)` in `env.smt2`, and run Z3 again.

```
_____________________________
_____________________________
_____________________________
```

Z3 answers:

```
sat
((python-3.11 false)
 (python-3.12 true)
 (numpy-1.26 false)
 (numpy-2.1 true)
 (scipy-1.11 false)
 (scipy-1.14 true)
 (oldplot-0.9 false))
```

Part 4 — *Explain a conflict.* The user now also requests oldplot, and scipy 1.14 explicitly. Write the formula of this request:

In the file `conflict.smt2` below, every dependency and request is asserted with a name: for instance, `(assert (! scipy-1.14 :named request-scipy))` asserts $scipy_{1.14}$ and names this assertion `request-scipy`. Here, `!` attaches the name and does not negate: negation is `not` [4]. When Z3 answers `unsat`, the command `(get-unsat-core)` prints the names of some named assertions whose conjunction with the unnamed assertions is already unsatisfiable [4]. Run Z3 on this file.

```
(set-option :produce-unsat-cores true)
(declare-const python-3.11 Bool)
(declare-const python-3.12 Bool)
(declare-const numpy-1.26 Bool)
(declare-const numpy-2.1 Bool)
(declare-const scipy-1.11 Bool)
(declare-const scipy-1.14 Bool)
(declare-const oldplot-0.9 Bool)
; At most one version of each package.
(assert (not (and python-3.11 python-3.12)))
(assert (not (and numpy-1.26 numpy-2.1)))
(assert (not (and scipy-1.11 scipy-1.14)))
; Dependencies and requests, named.
(assert (! (=> numpy-1.26 (or python-3.11 python-3.12))
           :named numpy-1.26-deps))
(assert (! (=> numpy-2.1 python-3.12) :named numpy-2.1-deps))
(assert (! (=> scipy-1.11 numpy-1.26) :named scipy-1.11-deps))
(assert (! (=> scipy-1.14 numpy-2.1) :named scipy-1.14-deps))
(assert (! (=> oldplot-0.9 python-3.11) :named oldplot-0.9-deps))
(assert (! scipy-1.14 :named request-scipy))
(assert (! oldplot-0.9 :named request-oldplot))
(check-sat)
(get-unsat-core)
```

Z3 answers:

```
unsat
(numpy-2.1-deps scipy-1.14-deps oldplot-0.9-deps request-scipy request-oldplot)
```

Mark with a checkmark (✓) the explanation that the core gives:

( ) Packages scipy 1.14 and oldplot 0.9 both depend on numpy, in different versions.

( ) Package oldplot 0.9 depends on python 3.11, scipy 1.14 depends on numpy 2.1, which depends on python 3.12, and at most one version of python can be installed.

( ) Package oldplot 0.9 depends on python 3.11, but numpy 1.26 excludes python 3.11.

2.2 Comparing access policies

Cloud services such as Amazon Web Services (AWS) protect stored files with *access policies*, which grant or refuse the requests made to the files. AWS checks such policies with Zelkova, a tool that encodes them as formulas over strings and asks an SMT solver whether one policy grants a request that another one denies [7]. The two policies below come from the paper on Zelkova [7]. Policy X lets students read the exam of the course cs240, and teaching assistants read the exam and its answers. Policy Y is a shorter rewrite of X.

Statement	Effect	Principal	Action	Resource
X_0	allow	`students`	`getObject`	`cs240/Exam.pdf`
X_1	allow	`tas`	`getObject`	`cs240/Exam.pdf`, `cs240/Answer.pdf`
Y_0	allow	`*`	`getObject`	`cs240/*`
Y_1	deny	`students`	`getObject`	`cs240/Answer.pdf`

Exercise 4 (Comparing access policies with Z3).

Part 1 — *Read the policies.* Observe the policies X and Y in the table above, then match each term with its meaning by writing its number in the brackets. An example is provided.

Term		Meaning
1	principal p ( )	one row of the table, such as X_0
2	action a ( )	refuses the matching requests, even if an allow statement matches them
3	resource r (1)	who makes the request, such as `students`
4	statement ( )	stands for any string, as in `cs240/*`
5	policy ( )	the file that the request names, such as `cs240/Exam.pdf`
6	effect allow ( )	grants the matching requests, unless a deny statement matches them
7	effect deny ( )	what the request does with the file, such as `getObject`
8	wildcard `*` ( )	a list of statements, such as X_0 and X_1

Part 2 — *Evaluate the policies.* Indicate with a checkmark (✓) or a cross (×) whether each policy grants each request. An example is provided.

Principal	Action	Resource	X grants	Y grants
`students`	`getObject`	`cs240/Exam.pdf`	(✓)	(✓)
`students`	`getObject`	`cs240/Answer.pdf`	( )	( )
`tas`	`getObject`	`cs240/Answer.pdf`	( )	( )
`tas`	`putObject`	`cs240/Exam.pdf`	( )	( )

Part 3 — *Encode the policies.* Each statement and each policy is a formula over p, a and r, which holds if it grants the request (p, a, r). Complete the formulas of Y_0, Y_1 and Y,

where *prefixof*(s, t) stands for `(str.prefixof s t)`. The formulas of X are provided.

$X_0 := p = \texttt{"students"} \wedge a = \texttt{"getObject"} \wedge r = \texttt{"cs240/Exam.pdf"}$
$X_1 := p = \texttt{"tas"} \wedge a = \texttt{"getObject"} \wedge (r = \texttt{"cs240/Exam.pdf"} \vee r = \texttt{"cs240/Answer.pdf"})$
$X := X_0 \vee X_1$
$Y_0 :=$ _____
$Y_1 :=$ _____
$Y :=$ _____

Complete the file `policy.smt2` below with these formulas.

```
(declare-const p String)
(declare-const a String)
(declare-const r String)
(define-fun X0 () Bool
  (and (= p "students") (= a "getObject") (= r "cs240/Exam.pdf")))
(define-fun X1 () Bool
  (and (= p "tas") (= a "getObject")
       (or (= r "cs240/Exam.pdf") (= r "cs240/Answer.pdf"))))
(define-fun X () Bool (or X0 X1))
(define-fun Y0 () Bool
  ____________________________________)
(define-fun Y1 () Bool
  ____________________________________)
(define-fun Y () Bool ________________)
```

Part 4 — *Compare the policies.* Write the formula that a request granted by X and denied by Y satisfies:

_____

Add its assertion and `(check-sat)` to `policy.smt2`, and run Z3.

```
_____________________________
(check-sat)
```

Z3 answers _____. Mark with a checkmark (✓) what this answer shows:

( ) Policy Y denies some request that X grants.

( ) Policy Y grants every request that X grants.

( ) Policies X and Y grant exactly the same requests.

Write the formula that a request granted by Y and denied by X satisfies:

_____

Replace the assertion by the one of this formula, and print the request.

```
_____________________________
(check-sat)
(get-value (p a r))
```

Z3 answers:

```
sat
((p "!0!")
 (a "getObject")
 (r "cs240/"))
```

Mark with a checkmark (✓) what this answer shows:

( ) No principal is called `"!0!"`: the request is not real, and Y is correct.

( ) Policy Y lets every principal read the files of cs240: they are public.

( ) Policy Y denies this request, because its deny statement Y_1 matches it.

Part 5 — *Restrict the principals.* Write the formula stating that p is `students` or `tas`:

_____

Add its assertion before `(check-sat)`, and run Z3 again.

```
_____________________________________
```

Z3 answers:

```
sat
((p "students")
 (a "getObject")
 (r "cs240/"))
```

Complete: Y grants the principal $p =$ _____ the resource $r =$ _____, which X denies.

3 From formulas to clauses

SAT solvers take formulas in conjunctive normal form (CNF). This section introduces CNF together with disjunctive normal form (DNF), compares the cost of converting a formula into each of them, and ends with a conversion into CNF that keeps the size of the formula linear. We use propositional formulas built from variables and the connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.

Exercise 5 (Completing the definitions).

Part 1 — *The connectives.* The table gives the truth values of the connectives for every assignment of truth values to the formulas φ and ψ, where 1 stands for true and 0 for false [1].

φ	ψ	$\neg\varphi$	$\varphi \wedge \psi$	$\varphi \vee \psi$	$\varphi \rightarrow \psi$	$\varphi \leftrightarrow \psi$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

Complete the sentences below with *true* or *false*.

- The only case where the negation $\neg\varphi$ is true is when φ is false.
- The only case where the conjunction $\varphi \wedge \psi$ is _____ is when both φ and ψ are _____.
- The only case where the disjunction $\varphi \vee \psi$ is _____ is when both φ and ψ are _____.

- The only case where the implication $\varphi \to \psi$ is _____ is when its premise φ is _____ and its conclusion ψ is _____.
- The biconditional $\varphi \leftrightarrow \psi$ is _____ if and only if φ and ψ are both true or both _____.

Part 2 — *Satisfiability and normal forms.* Complete the definitions below [1] with the words in the box, one word per blank.

conjunction	every	disjunctive	negation	some
disjunction	conjunctive			

- An *assignment* maps (propositional) variables to the truth values 1 and 0.
- A *model* of a formula is an assignment that makes the formula true.
- A formula is *satisfiable* if it can be made true, that is, if _____ assignment makes it true.
- A formula is *valid* if it is always true, that is, if _____ assignment makes it true.
- A *literal* is a variable x or its _____ $\neg x$.
- A *clause* is a _____ of literals, such as $p \vee \neg q \vee r$.
- A *term* is a _____ of literals, such as $p \wedge \neg q \wedge r$.
- A formula is in _____ normal form (CNF) if it is a conjunction of clauses.
- A formula is in _____ normal form (DNF) if it is a disjunction of terms.

Exercise 6 (Formulas in DNF and in CNF).

Part 1 — *Decide satisfiability.* For each formula below, give its normal form (DNF, CNF or both), say whether it is satisfiable, and give a model if it is.

Formula	Normal form	Satisfiable?	Model
$p \wedge q \wedge \neg r$	both	yes	$p = 1,\ q = 1,\ r = 0$
$(p \wedge \neg q) \vee (q \wedge \neg q)$	_____	_____	_____
$(p \wedge \neg p \wedge q) \vee (\neg q \wedge r \wedge q)$	_____	_____	_____
$(p \vee q) \wedge (\neg p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee \neg q)$	_____	_____	_____
$(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p)$	_____	_____	_____

Mark with a checkmark (✓) what the table confirms, once for DNF and once for CNF:

( ) A formula in DNF is satisfiable if and only if every term contains no pair x, $\neg x$.

( ) A formula in DNF is satisfiable if and only if some term contains no pair x, $\neg x$.

( ) A formula in CNF is satisfiable if and only if no clause contains a pair x, $\neg x$.

( ) A formula in CNF can be unsatisfiable although no clause contains a pair x, $\neg x$.

Part 2 — *Convert.* Every formula can be converted into an equivalent formula in CNF and into one in DNF with the following equivalences [1].

(E1) Implication: $\varphi \to \psi \equiv \neg\varphi \vee \psi$.

(E2) Double negation: $\neg\neg\varphi \equiv \varphi$.

(E3) De Morgan's laws: $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ and $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$.

(E4) Distributing $\wedge$ over $\vee$: $\varphi \wedge (\psi_1 \vee \psi_2) \equiv (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)$.

(E5) Distributing $\vee$ over $\wedge$: $\varphi \vee (\psi_1 \wedge \psi_2) \equiv (\varphi \vee \psi_1) \wedge (\varphi \vee \psi_2)$.

(E6) Commutativity: $\varphi \wedge \psi \equiv \psi \wedge \varphi$ and $\varphi \vee \psi \equiv \psi \vee \varphi$.

(E7) Associativity: $(\varphi \wedge \psi) \wedge \chi \equiv \varphi \wedge (\psi \wedge \chi)$ and $(\varphi \vee \psi) \vee \chi \equiv \varphi \vee (\psi \vee \chi)$.[1]

Use them to write each formula below in CNF and in DNF.

Formula	CNF	DNF
$\neg(p \wedge \neg q)$	$\neg p \vee q$	$\neg p \vee q$
$p \to (q \wedge r)$		
$\neg(p \vee q) \vee r$		
$\neg(p \to q) \wedge (r \vee s)$		

Exercise 7 (The cost of converting to DNF). The conversions of Exercise 6 were done on small formulas. Here, we measure the size of the result of distributing on a family of formulas in CNF, one formula for each $n \geq 1$:

$$C_n = (x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \cdots \wedge (x_n \vee y_n).$$

Part 1 — *Convert a small case.* Convert $C_2 = (x_1 \vee y_1) \wedge (x_2 \vee y_2)$ into DNF with the equivalences of Exercise 6, one law per line. Write the law you use on the right of each line.

$$\begin{aligned} C_2 &\equiv \big((x_1 \vee y_1) \wedge x_2\big) \vee \big((x_1 \vee y_1) \wedge y_2\big) && \text{(E4)} \\ &\equiv _____ && ____ \\ &\equiv _____ && ____ \\ &\equiv _____ && ____ \end{aligned}$$

[1] When an operation is associative, we can avoid writing parentheses. Thus, parentheses inside a conjunction of conjunctions (or a disjunction of disjunctions) can be dropped.

Part 2 — *Count the terms.* Distributing in the same fashion, we can turn every C_n into a DNF. The formula $C_1 = x_1 \vee y_1$ is already a DNF, and for C_3 the result is

$$(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge x_2 \wedge y_3) \vee (x_1 \wedge y_2 \wedge x_3) \vee (x_1 \wedge y_2 \wedge y_3)$$
$$\vee (y_1 \wedge x_2 \wedge x_3) \vee (y_1 \wedge x_2 \wedge y_3) \vee (y_1 \wedge y_2 \wedge x_3) \vee (y_1 \wedge y_2 \wedge y_3).$$

Complete the table for C_1, your DNF of C_2 and the DNF of C_3 above. Then continue the pattern for $n = 10$ and for any n.

Hint. Each term of the DNF of C_2 occurs twice in the DNF of C_3.

n	1	2	3	5	10	n
Terms of the DNF of C_n	2	_____	_____	32	_____	_____

Part 3 — *The other direction.* The formula $D_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$ is in DNF. Converting it into CNF with (E5), (E6) and (E7) gives how many clauses?

Hint. Compare with Part 2: exchange the roles of $\wedge$ and $\vee$.

Clauses of the CNF of $D_n =$ _____

Mark the conclusion with a checkmark (✓):

( ) Converting to DNF and checking each term takes time linear in the size of the formula.

( ) Converting into an equivalent DNF, or into an equivalent CNF by distributing, can produce a formula exponentially larger than the original one.

( ) Converting into CNF by distributing is always linear, so only DNF is expensive.

3.1 Tseitin's encoding

A SAT solver does not need a formula *equivalent* to the original one. It needs an *equisatisfiable* one: two formulas are equisatisfiable if both are satisfiable or both are unsatisfiable [1]. Tseitin's encoding builds such a formula in CNF, without distributing as above. It reads a formula as a tree [1]:

- A *leaf* is an occurrence of a variable, and an *inner node* is an occurrence of a connective. The children of an inner node are its *arguments*.
- A *gate* is an inner node with its arguments. The *root gate* is the gate at the root.
- Each gate gets a *new variable*, a variable that does not occur in the formula.
- The *constraint* of a gate with new variable a is $a \leftrightarrow \gamma$, where γ is the gate's connective applied to its arguments, with every argument that is a gate replaced by its new variable.
- The *root clause* is the clause (a), where a is the new variable of the root gate.

The encoding is the conjunction of the CNFs of all constraints and of the root clause [1].

Exercise 8 (Tseitin's encoding).

Part 1 — *The parts of the encoding.* Figure 1 shows the tree of $(p \wedge q) \vee \neg r$ with its new variables in brackets, its constraints and its root clause. Label it with the terms defined above.

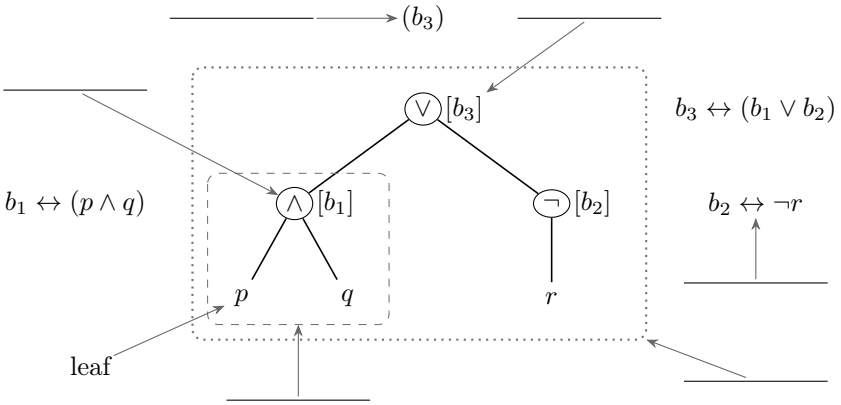

Figure 1: The parts of Tseitin's encoding of $(p \wedge q) \vee \neg r$.

Part 2 — *One gate at a time.* The encoding needs the CNF of every kind of constraint. Match each constraint on the left with its CNF on the right by writing its letter in the brackets. An example is provided.

Hint. The constraint $a \leftrightarrow \psi$ is $(a \to \psi) \wedge (\psi \to a)$, and $\varphi \to \psi \equiv \neg\varphi \vee \psi$.

a)	$a \leftrightarrow (x \wedge y)$	( )	$(\neg a \vee x \vee y) \wedge (a \vee \neg x) \wedge (a \vee \neg y)$
b)	$a \leftrightarrow \neg x$	(a)	$(\neg a \vee x) \wedge (\neg a \vee y) \wedge (a \vee \neg x \vee \neg y)$
c)	$a \leftrightarrow (x \vee y)$	( )	$(\neg a \vee \neg x \vee y) \wedge (a \vee x) \wedge (a \vee \neg y)$
d)	$a \leftrightarrow (x \to y)$	( )	$(\neg a \vee \neg x) \wedge (a \vee x)$

Part 3 — *A whole formula.* Figure 2 shows the gates of

$$\phi = ((p \vee q) \wedge r) \to \neg s$$

with their new variables in brackets. Complete the table with the gate of each new variable, with every argument that is a gate replaced by its new variable, and with the CNF of the gate's constraint from Part 2.

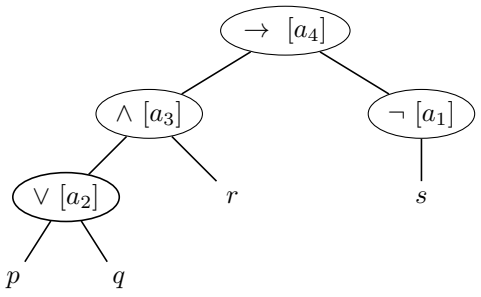

Figure 2: The gates of ϕ and their new variables.

New variable	Gate	CNF of the constraint
a_1	_____	_____
a_2	_____	_____
a_3	$a_2 \wedge r$	$(\neg a_3 \vee a_2) \wedge (\neg a_3 \vee r) \wedge (a_3 \vee \neg a_2 \vee \neg r)$
a_4	_____	_____
Root clause		_____

Exercise 9 (Equisatisfiable and small). The formula $\phi = ((p \vee q) \wedge r) \rightarrow \neg s$ and its gates are those of Exercise 8, Part 3.

Part 1 — *A model of ϕ.* Give an assignment v_1 that satisfies ϕ.

$$v_1(p) = ___, \quad v_1(q) = ___, \quad v_1(r) = ___, \quad v_1(s) = ___$$

Part 2 — *Extend the model.* Let v_2 be the same as your v_1 of Part 1 but it assigns the variables a_1 to a_4 the corresponding values of the gates. Complete those values in the table below, and check that v_2 satisfies Tseitin's encoding of ϕ.

New variable a	Gate	$v_2(a)$
a_1	$\neg s$	____
a_2	$p \vee q$	____
a_3	$a_2 \wedge r$	____
a_4	$a_3 \rightarrow a_1$	____

Part 3 — *Equisatisfiable, not equivalent.* For each question below, mark the answer with its justification with a checkmark (✓).

(i) Is every model of Tseitin's encoding of ϕ a model of ϕ?

( ) Yes, because the encoding only has the gates of ϕ, so it is equivalent to ϕ.

( ) Yes, because the constraints force each new variable to have the value of its gate, so ϕ has the value of a_4, which the root clause (a_4) makes 1.

( ) No, because a model of the encoding can give a new variable a value that differs from the value of its gate.

(ii) Is every model of ϕ a model of Tseitin's encoding of ϕ?

( ) Yes, because Part 2 extends every model of ϕ to a model of the encoding.

( ) No, because some models of ϕ cannot be extended to a model of the encoding, whatever the values of $a_1, \ldots, a_4$.

( ) No, because changing $v_2(a_4)$ of Part 2 to 0 gives an assignment that is still a model of ϕ, but falsifies the root clause (a_4) of the encoding.

Mark the conclusion with a checkmark (✓):

( ) The formula ϕ and its encoding are equivalent.

( ) The formula ϕ and its encoding are equisatisfiable, but not equivalent.

( ) The formula ϕ and its encoding are not equisatisfiable.

Part 4 — *Count the clauses.* The table below gives the constraints and the root clause of Tseitin's encoding of the formula D_n of Exercise 7 for $n = 1, 2, 3$.

n	Constraints	Root clause
1	$g_1 \leftrightarrow (x_1 \wedge y_1)$	(g_1)
2	$g_1 \leftrightarrow (x_1 \wedge y_1)$, $g_2 \leftrightarrow (x_2 \wedge y_2)$, $h_2 \leftrightarrow (g_1 \vee g_2)$	(h_2)
3	$g_1 \leftrightarrow (x_1 \wedge y_1)$, $g_2 \leftrightarrow (x_2 \wedge y_2)$, $g_3 \leftrightarrow (x_3 \wedge y_3)$, $h_2 \leftrightarrow (g_1 \vee g_2)$, $h_3 \leftrightarrow (h_2 \vee g_3)$	(h_3)

Count the gates and the clauses of the encoding for $n = 1, 2, 3$, continue the pattern for $n = 10$ and for any n, and compare with the corresponding CNF by distributing.

Hint. In Part 2 of Exercise 8, the CNF of every binary gate has 3 clauses, and the root clause adds 1.

n	Gates of D_n	Clauses of Tseitin's encoding	Clauses by distributing
1	1	$3 \cdot 1 + 1 = 4$	2
2	_____	_____	4
3	_____	_____	8
10	_____	_____	1024
n	_____	_____	2^n

4 Searching for a satisfying assignment

With a formula in CNF, a solver searches for a model. Trying all 2^n assignments of n variables is expensive for large n. The DPLL algorithm (Davis, Putnam, Logemann and Loveland, 1962) instead builds a partial assignment step by step [2]. Modern solvers extend DPLL so that explored failures are not repeated (CDCL) [1].

4.1 DPLL: propagate, decide, backtrack

Exercise 10 (States of clauses). Given the definitions below [1] and using a partial assignment v, which gives values only to y_1, y_2 and y_4, with $v(y_1) = 1$, $v(y_2) = 0$ and $v(y_4) = 1$, complete the table below. A clause $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_m$ is

- *satisfied* if one or more of its literals ℓ_i are satisfied,
- *conflicting* if all of its literals ℓ_i are false,
- *unit* if it is not satisfied and all of its literals ℓ_i but one are false,
- *unresolved* otherwise.

Clause	$y_1 \vee y_3 \vee \neg y_4$	$\neg y_1 \vee y_2$	$\neg y_1 \vee \neg y_4 \vee y_3$	$\neg y_1 \vee y_3 \vee y_5$	$y_2 \vee \neg y_4 \vee \neg y_5$
Kind	satisfied	_____	_____	_____	_____

Exercise 11 (Propagating unit clauses). Follow the steps of the Boolean constraint propagation (BCP) algorithm [1], described below, on the formula

$$c_1 = (\neg q \vee r) \qquad c_2 = (p) \qquad c_3 = (\neg p \vee q) \qquad c_4 = (p \vee \neg r),$$

starting from the empty assignment v, which gives no value to any variable.

(B1) **Conflict or no-unit?:** If a clause is conflicting, stop and report the lowest-numbered conflicting one. If no clause is unit, stop. Otherwise, continue with (B2).

(B2) **Unit:** Take the lowest-numbered unit clause, and give the variable of its unassigned literal the value that satisfies this literal (the *unit clause rule*).

(B3) **Simplify:** Delete satisfied clauses, and false literals from the other clauses. A unit clause then has one literal, and a conflicting clause has zero. Go to (B1).

Step	Outcome	Assignment v	Formula
start		no value	$c_1 = (\neg q \vee r)$, $c_2 = (p)$, $c_3 = (\neg p \vee q)$, $c_4 = (p \vee \neg r)$
(B1)	continue		
(B2)	$c_2 = (p)$	$v(p) = 1$	
(B3)			$c_1 = (\neg q \vee r)$, $c_3 = (q)$
(B1)	_____		
(B2)	_____	_____	
(B3)			_____
(B1)	_____		
(B2)	_____	_____	
(B3)			_____
(B1)	_____		

The input formula is satisfiable, and the assignment found by BCP that satisfies it is $v(p) =$ ___, $v(q) =$ ___ and $v(r) =$ ___.

Exercise 12 (Running DPLL). Giving a fixed order to the variables of a formula, the DPLL algorithm executes the steps below [2, 1].

(D1) **BCP:** Run BCP (Exercise 11).

(D2) **Conflict? ⇒ flip, or unsat:** If a clause is conflicting and every decision has been flipped, then stop: the formula is unsatisfiable. If a clause is conflicting otherwise, *backtrack*: undo the most recent unflipped decision (see (D4)) and all later assignments, and *flip* the value of its variable. Go to (D1).

(D3) **All sat?:** If every clause is satisfied, stop: the formula is satisfiable, and the variables that are still unassigned may take any value in its model.

(D4) **Decide:** Otherwise, *decide*: set the first unassigned variable of the fixed order to 1. This value may be flipped later at (D2). Go to (D1).

Run DPLL on the clauses below, in the variable order $p_1, \ldots, p_4$.

$$c_1 = (\neg p_1 \vee p_2) \quad c_2 = (p_3 \vee \neg p_2) \quad c_3 = (\neg p_3 \vee p_4) \quad c_4 = (\neg p_4 \vee \neg p_2) \quad c_5 = (p_1 \vee p_3)$$

The table below shows the first steps of the run. Complete the reason of each step: a decision, a unit clause c_i, a conflicting clause c_i, or a flip. Then, continue the run in the empty rows until DPLL stops. The first column gives the steps of DPLL or of BCP that each row carries out.

Step	Reason	$v(p_1)$	$v(p_2)$	$v(p_3)$	$v(p_4)$	Formula
start		—	—	—	—	$c_1, \ldots, c_5$
(D4)	decision	1	—	—	—	$c_1 = (p_2)$, $c_2 = (p_3 \vee \neg p_2)$, $c_3 = (\neg p_3 \vee p_4)$, $c_4 = (\neg p_4 \vee \neg p_2)$
(B1)–(B3)		1	1	—	—	$c_2 = (p_3)$, $c_3 = (\neg p_3 \vee p_4)$, $c_4 = (\neg p_4)$
(B1)–(B3)		1	1	1	—	$c_3 = (p_4)$, $c_4 = (\neg p_4)$
(B1)–(B3)		1	1	1	1	$c_4 = ()$
(B1)		1	1	1	1	$c_4 = ()$
(D2)		0	—	—	—	$c_2 = (p_3 \vee \neg p_2)$, $c_3 = (\neg p_3 \vee p_4)$, $c_4 = (\neg p_4 \vee \neg p_2)$, $c_5 = (p_3)$
(B1)–(B3)		—	—	—	—	
(B1)–(B3)		—	—	—	—	
(B1)–(B3)		—	—	—	—	

Note: A formula φ is valid if and only if $\neg\varphi$ is unsatisfiable (why?). To check whether φ is valid, run DPLL on a CNF of $\neg\varphi$, such as Tseitin's encoding of $\neg\varphi$ (Exercise 8). If DPLL stops at (D2), then $\neg\varphi$ is unsatisfiable, and φ is valid. If DPLL stops at (D3), then the values that its model gives to the variables of φ make φ false: they are a counterexample to the validity of φ.

References

[1] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*, 2nd edition. Springer, 2016.

[2] Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (editors). *Handbook of Satisfiability*. IOS Press, 2009.

[3] Dennis Yurichev. *SAT/SMT by Example*. https://smt.st.

[4] Clark Barrett, Pascal Fontaine and Cesare Tinelli. *The SMT-LIB Standard*. https://smt-lib.org.

[5] The mamba developers. *Solving Package Environments*, mamba documentation. https://mamba.readthedocs.io/en/latest/usage/solver.html.

[6] Nikolaj Bjørner, Anh-Dung Phan and Lars Fleckenstein. νZ – An Optimizing SMT Solver. In *TACAS 2015*, LNCS 9035, Springer, 2015.

[7] John Backes et al. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *FMCAD 2018*, IEEE, 2018.