Monash University
Faculty of Information Technology
2nd Semester 2025

# FIT2014
# Assignment 1
# Linux tools, logic, regular expressions, induction
# DUE: 11:55pm, Friday 22 August 2025 (Week 4)

Start work on this assignment early. Bring questions to Consultation and/or the Ed Forum.

## Instructions

- **Generative AI tools must not be used for any part of this Assignment.**
  You must not use generative artificial intelligence (AI) to generate any materials or content in relation to this Assignment (or any other assessment in this unit). For example, **no** GPT, DeepSeek, Copilot, Claude, Cohere, Gemini/Bard, etc.
- The work you submit for this Assignment must be your own individual work.
- To start work, download the workbench `asgn1.zip` from Moodle. Create a new Ed Workspace and upload this file, letting Ed automatically extract it. Edit the `student-id` file to contain your name and student ID. Refer to Lab 0 for a reminder on how to do these tasks.
- The workbench provides names for all solution files. These will be empty, needing replacement. Do **not** add or remove files from the workbench.
- Solutions to written questions must be submitted as PDF documents. You can create a PDF file by scanning your **legible** (use a pen, write carefully, etc.) hand-written solutions, or by directly typing up your solutions on a computer. If you type your solutions, be sure to create a PDF file. There will be a penalty if you submit any other file format (such as a Word document). Refer to Lab 0 for a reminder on how to upload your PDF to the Ed workspace and replace the placeholder that was supplied with the workbench.
- Every PDF file submitted must also contain your name and student ID at the start.
- When you have finished your work, download the Ed workspace as a zip file by clicking on "Download All" in the file manager panel. **You must submit this zip file to Moodle by the deadline given above.**
- To aid the marking process, you must adhere to all naming conventions that appear in the assignment materials, including files, directories, code, and mathematics. Not doing so will cause your submission to incur a one-day late-penalty (in addition to any other late-penalties you might have). Be sure to check your work carefully.

Your submission must include:

- the file `student-id`, edited to contain your name and student ID
- a one-line text file, `prob1.txt`, with your solution to Problem 1;
- another one-line text file, `prob2.txt`, with your `awk` pattern for vertex lines in Problem 2;
- an `awk` script, `prob2.awk`, for Problem 2;
- a PDF file `prob3.pdf` with your solution to Problem 3;
- an `awk` script, `prob4.awk`, for Problem 4;
- a file `prob5.pdf` with your solution to Problem 5.

Initially, the `asgn1` directory contains empty files (or dummy files) with the required filenames. These must each be replaced by the files you write, as described above. Before submission, **check** that each of these empty files is, indeed, replaced by your own file, and that the `student-id` file is edited as required.

## Introduction to the Assignment

In Lab 0, you met the stream editor `sed`, which detects and replaces certain types of *patterns* in text, processing one line at a time. These patterns are actually specified by *regular expressions*.

In this assignment, you will use `awk` which does some similar things and a lot more. It is a simple programming language that is widely used in Unix/Linux systems and also uses regular expressions. In Problems 1–4, you will construct an `awk` program to construct, for any directed graph, a logical expression that describes the conditions under which the directed graph has a kernel.

Finally, Problem 5 is about applying induction to a problem about structure and satisfiability of some Boolean expressions in Conjunctive Normal Form (CNF).

## Introduction to `awk`

An `awk` program takes an input file and processes it line-by-line. In an `awk` program, each line has the form

*/pattern/* `{` *action* `}`

where the *pattern* is a regular expression (or certain other special patterns) and the *action* is an instruction that specifies what to do with any line in the input file that contains a match for the *pattern*. The *action* (and the {...} around it) can be omitted, in which case any line that matches the *pattern* is printed.

Once you have written your program, it does not need to be compiled. It can be executed directly, by using the `awk` command in Linux:

`$ awk -f` *programName* *inputFileName*

Your program is then executed on an input file in the following way.

```
//   Initially, we're at the start of the input file, and haven't read any of it yet.
If the program has a line with the special pattern BEGIN, then
    do the action specified for this pattern.
Main loop, going through the input file:
{
    inputLine := next line of input file
    Go to the start of the program.
    Inner loop, going through the program:
    {
        programLine := next line of program (but ignore any BEGIN and END lines)
        if inputLine contains a string that matches the pattern in programLine, then
            if there is an action specified in the programLine, then
            {
                do this action
            }
            else
                just print inputLine        //   it goes to standard output
    }
}
If the program has a line with the special pattern END, then
    do the action specified for this pattern.
```

Any output is sent to standard output.

You should read about the basics of `awk`, including

- the way it represents regular expressions,
- the variables `$1`, `$2`, *etc.*, and `NF`,
- the function `printf(···)`,
- `if` statements
- `for` loops. For these, you will only need simple loops like

```
for (i = 1; i <= n; i++)
{
        ⟨body of loop⟩
}
```

  This particular loop executes the body of the loop once for each $i = 1, 2, \ldots,$ `n` (unless the body of the loop changes the variable `i` somehow, in which case the behaviour may be different, but you should not need to do that). You can nest loops, so the body of the loop can be another loop. It's also ok to write loops more compactly,

```
for (i = 1; i <= n; i++)  {      ⟨body of loop⟩      }
```

  although you should ensure it's still clearly readable.

Any of the following sources should be a reasonable place to start:

- A. V. Aho, B. W. Kernighan and P. J. Weinberger, *The AWK Programming Language*, Addison-Wesley, New York, 1988.
  (The first few sections of Chapter 1 should have most of what you need, but be aware also of the regular expression specification on p28.)
- the GNU Awk User's Guide.
- the Wikipedia article looks ok
- the `awk` manpage

The following is a useful reference, but might be less suitable as an introduction, as its detailed treatment of patterns and actions (the heart of `awk`) comes very late, long after extensive discussion of other programming aspects:

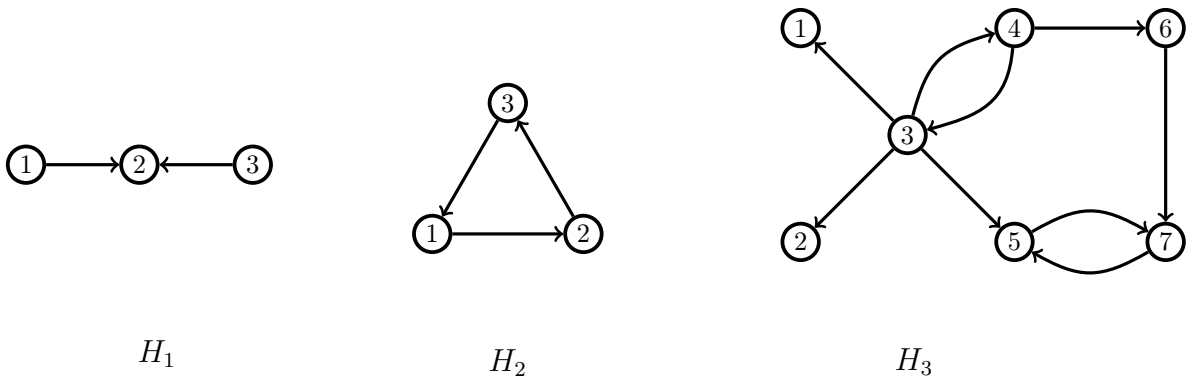
- `https://www.grymoire.com/Unix/Awk.html`

## Introduction to Problems 1–4

Many systems and structures can be modelled as graphs (abstract networks) or as directed graphs, also called digraphs (in which every edge is an *ordered* pair of vertices, and therefore has a direction, and is shown as an arrow in diagrams).

Suppose you have a collection of devices. Each device can monitor some other devices. Note that the monitoring relationship has a direction: if $A$ monitors $B$, then it does not always follow that $B$ monitors $A$. In some cases, it might, but in other cases, it might not. There can also be pairs of devices such that neither monitors the other.

In order to help detect any faults in the network, we want to equip some of the devices with alarms. An alarm goes off if its own device is faulty or if that device is monitoring another device that is faulty. We also require that, when a device monitors another device, they cannot both have alarms (which could be because the alarms interfere with each other in some way, or it could be just a constraint introduced to reduce cost). So we want to identify a selection of devices and equip only those selected devices with alarms, and to do this in such a way that (i) every device either has an alarm or is monitored by a device with an alarm, and (ii) no device with an alarm monitors another device with an alarm.




$H_1$ $H_2$ $H_3$

Figure 1: Three digraphs, $H_1, H_2, H_3$. The first, $H_1$, has a kernel, but the second one, $H_2$, does not.

We can model this abstractly as follows. Throughout, we use $G$ to denote a directed graph (also called a **digraph**), $n$ denotes its number of vertices and $m$ denotes its number of directed edges, also called **arcs**. $V(G)$ denotes the set of vertices of $G$, and $A(G)$ denotes the set of arcs of $G$. In terms of our application, the vertices represent devices and the arcs represent the monitoring relation among devices.

If there is an arc $(v, w)$ from vertex $v$ to vertex $w$, then $v$ is an **in-neighbour** of $w$, because the arc from $v$ comes into $w$, and $w$ is an **out-neighbour** of $v$, because the arc to $w$ comes out of $v$.

A **kernel** in a directed graph $G$ is a set $X \subseteq V(G)$ of vertices such that

(i) for every vertex $v \in V(G)$, either $v \in X$ or there exists another vertex $u \in X$ such that $(u, v) \in A(G)$;

(ii) for every edge $(v, w) \in A(G)$, either $v \notin X$ or $w \notin X$.

For our network of devices, the kernel represents the set of devices that are equipped with alarms according to our conditions.

For example, consider the digraphs in Figure 1.

In the digraph $H_1$, the vertex set $X = \{1, 3\}$ is a kernel. The only vertex not in $X$ is 2, but there is a vertex in X from which an arc goes into 2. In particular, there is an arc $(1, 2)$ into 2 from vertex 1, and vertex 1 is in $X$. (As it happens, vertex 3 could play this role just as well as vertex 1. But we only need *at least one* arc from a vertex in $X$ into vertex 2.)

However:

- $Y = \{2\}$ is not a kernel, because no vertex in $Y$ has an arc going from it into vertex 1 which is not in $Y$. (The same can be said about vertex 3.) In the device-network context: there is no way for an output gadget to report on vertex 1 (or on vertex 3). So condition (i) in the definition of a kernel is not satisfied. It does not matter that condition (ii) is satisfied.

- $Z = \{1, 2\}$ is not a kernel, because both vertices in the arc $(1, 2)$ are in $Z$, which is not allowed in a kernel. So condition (ii) is not satisfied. It does not matter that condition (i) is satisfied.

Digraph $H_2$ has no kernel. For example:

- the set $X = \{1\}$ is not a kernel, because the vertex 3 is not in $X$ and there is no arc going into vertex 3 from a vertex in $X$. In this case, the only arc going into 3 is the arc $(2, 3)$ from vertex 2, but vertex 2 is not in $X$ either. So condition (i) is not satisfied.

- the set $Y = \{2, 3\}$ is not a kernel, because in this case we have both endpoints of the arc $(2, 3)$ belonging to $Y$, which is not allowed in kernels. So condition (ii) is not satisfied.

Does digraph $H_3$ have any kernels? If so, how many? If not, why not?

In Problems 1–4, you will complete a partly-written program in `awk` that constructs, for any digraph $G$, a Boolean expression $\varphi_G$ in Conjunctive Normal Form, with variables $v_i$ (described below), such that $\varphi_G$ captures, in logical form, the statement that $G$ has a kernel. This assertion might be True or False, depending on $G$, but the requirement is that

$G$ has a kernel $\iff$ you can assign truth values to the variables $v_i$ of $\varphi_G$ to make $\varphi_G$ True.

For each vertex $i \in V(G)$ we introduce a Boolean variable $v_i$. It is our *intention* that each such variable represents the statement that "vertex $i$ is included in the kernel" (which might be True or False). When we write code, each variable $v_i$ is represented by a name in the form of a text string v$i$ formed by concatenating `v` and $i$. So, for example, $v_2$ is represented by the name `v2`.

But if all we have is all these variables, then there is nothing to control whether they are each True or False. We will need to build some logic, in the form of a CNF expression, to make this interpretation work. So, we need to encode the definition of a kernel into a CNF expression using these variables. The expression we construct must depend on the digraph. Furthermore, it must depend *only* on the digraph.

We begin with a specific example (Problem 1) and then move on to general graphs (Problems 2–4).

## Problem 1. [2 marks]

For the graph $H_3$ shown on the right in Figure 1, construct a CNF Boolean expression $\varphi_H$ using the variables $v_i$ such that $\varphi_H$ is True if and only if the assignment of truth values to the variables represents a kernel of $H$.

Now type this expression $\varphi_H$ into a one-line text file, using our text names for the variables (*i.e.*, `v1`, `v2`, `v3`, etc.), with the usual logical operations replaced by text versions as follows:

| logical operation | text representation |
|---|---|
| $\neg$ | `~` |
| $\wedge$ | `&` |
| $\vee$ | `\|` |

Put your answer in a one-line text file called `prob1.txt`.

We are now going to look at the general process of taking any digraph $G$ on $n$ vertices, and constructing a corresponding Boolean expression $\varphi_G$ with $n$ variables, in such a way that:

- the variables of $\varphi_G$ correspond to vertices in $G$, and
- $\varphi_G$ evaluates to True if and only if the assignment of truth values to those variables corresponds to a kernel in $G$.

The next part of the assignment requires you to write an awk script to automate this construction. For the purposes of this task, every digraph is encoded in the following format (which is somewhat nonstandard):

- One line contains $n$, the number of vertices, followed by $m$, the number of edges, with at least one blank separating them, and possibly some blanks before and after them.
- Every other line is a **vertex line** and has the following format:

  $$j_1 \;\; j_2 \;\; \cdots \;\; j_d \;\; \texttt{:} \;\; i$$

  where $i$ and $j_1, j_2, \ldots, j_d$ (with $d \geq 0$) are all positive integers, with the $j_1, j_2, \ldots, j_d$ all being distinct. This means that the *in*-neighbours of vertex $i$ are $j_1, j_2, \ldots, j_d$, and that the arcs

going *into* vertex $i$ are $(j_1, i), (j_2, i), \ldots, (j_d, i)$. So this line serves as a complete list of those in-neighbours and of their associated arcs into vertex $i$.

- In a vertex line, we allow any positive number of spaces between numbers and on either side of the colon. So there is always space between the numbers and around the colon. There may also be some number of spaces before the first number $j_1$ and after the last number $i$.
- For each $i$ in the range $1 \leq i \leq n$, there is exactly one vertex line for vertex $i$.
- Lines that are not in one of the two formats above can be present but they must be ignored by your programs.
- The lines can be in any order in the file. There is no requirement that the line stating $n$ and $m$ be at the start, and there is no requirement that the vertex lines are listed in order of vertex number.
- For example, the digraph $H_3$ in Figure 1 could be represented by the eight-line file on the left below, or (less neatly) by the one on the right (which has the line with $n$ and $m$ later on in the file, which is permitted, and it also has a spurious line of text, which is also permitted but must be ignored).

```
7 9
3 : 1
3 : 2
4 :  3
3 :   4
3    7  :     5
4  : 6
5 6 :  7
```

```
4 :  3
 3 : 1
 5 6 :  7
FIT2014 : A1
3    7  :     5
7 9
 3 : 2
4  : 6
3 :   4
```

- Each digraph is represented in a file of its own. Each input file contains exactly one digraph represented in this way.
- Positive integers, for $n$ and the vertex numbers, can have any number of digits. They must have no decimal point and no leading 0.

## Problem 2. [7 marks]

Complete the partial `awk` script in the provided file `prob2.awk`, also shown below, so that when it takes, as input, a graph $G$ represented in the specified format in a text file, it should produce, as output, a one-line file containing the text representation of a Boolean expression $\varphi_G$ in CNF which uses the variables we have specified and which is `True` if and only if the variables describe a kernel of $G$.

The text representation is the same as that described on pp. 5–6 and used for Problem 1. You must provide two files for this problem:

- `prob2.txt`, a one-line file containing your `awk` pattern for matching a vertex line;
- `prob2.awk`, your completed `awk` script for the above task. The pattern it uses for matching vertex lines must be the same as the one you give in `prob2.txt`.

```
#  Faculty of IT, Monash University
#  FIT2014 Theory of Computation
#  2nd Semester, 2025
#  Assignment 1
#  Your name:
#  Your Student ID:

 BEGIN  {

     firstMatch = 1;

 }

 /Write your vertex-line pattern here, and also in prob2.txt/     {
     #  This action statement spans several lines.
     #  You will need to add code in certain places:
     #  to control some for-loops and to complete some  printf  statements
     #  that print parts of the CNF expression.
     #  Every string  ####  must be replaced by something, not necessarily
     #  of the same length.

     #  Think: why might we want to flag whether we are at the first clause?
     if (firstMatch == 1) {firstMatch = 0} else {printf(####);}


        #  Think: what should we print if a vertex has no incoming edges?
        if (NF == 2)
        {
          printf(####);
        }
        else
        {
                #  this loop ensures the first condition of kernels is met
                printf(####);
                for  (####)
                {
                        printf(####);
                }
                printf(####);

                # this loop ensures the second condition of kernels is met
                for  (####)
                {
                        printf(####);
                }
        }
 }


 END  {
   printf("\n");
 }
```

**Problem 3. [3 marks]**

(a) Derive an upper bound, in terms of $n$, for the number of calls to `printf` that are required to output $\varphi_G$.

(b) Express the upper bound for (a) in big-O notation.

Put your answers in a PDF file called `prob3.pdf`.

We are now going to modify the `awk` script so that the output it produces can be taken as input by a program for testing satisfiability.

SageMath is software for doing mathematics. It is powerful, widely used, free, open-source, and based on Python. It is already available in your Ed Workspace.[1] You don't need to learn SageMath for this assignment (although it's good to be aware of what it is); you only need to follow the instructions below on how to use a specific function in SageMath for a specific task. If you're interested, you may obtain further information, including tutorials, documentation and installation instructions, at `https://www.sagemath.org`.

In this part of the assignment, we just use one line of SageMath via the Linux command line in order to determine whether or not a Boolean expression in CNF is satisfiable (i.e., has an assignment of truth values to its variables that makes the expression True). Suppose we have the Boolean expression

$$(a \vee b) \wedge (\neg a \vee \neg b) \wedge (\neg a \vee b),$$

which we note is satisfiable because it can be made True by putting $a =$ False and $b =$ True. We first translate the expression into text in the way described earlier (p4), replacing $\neg,\wedge,\vee$ by `~`,`&`,`|` respectively. This gives the text string

```
(a | b) & (~a | ~b) & (~a | b).
```

We ask SageMath if this is satisfiable by entering the following text at the Linux command line:

```
$ sage -c 'print(propcalc.formula("(a | b) & ( a |  b) & ( a | b)").is_satisfiable())'
True
```

You can see that SageMath outputs `True` on the next line to indicate that the expression is satisfiable.

In `sage -c`, the "`-c`" instructs `sage` to execute the subsequent (apostrophe-delimited) Sage command and output the result (to standard output) without entering the SageMath environment.

This is all you need to do to use the SageMath satisfiability test on your expression. If you want to actually enter SageMath and use it interactively, you can do so:

```
$ sage
sage: print(propcalc.formula("(a | b) & (~a | ~b) & (~a | b)").is_satisfiable())
True
```

Again, the output `True` indicates satisfiability.

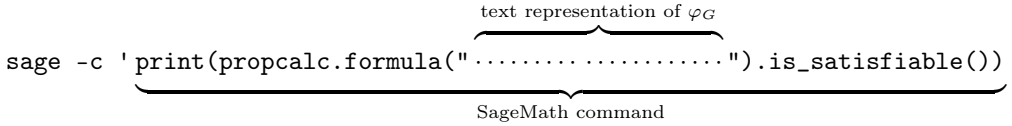
The SageMath satisfiability-testing function we are using here, `is_satisfiable()`, is the most primitive satisfiability-tester that you can use in SageMath. It determines if an expression is satisfiable by constructing the entire truth table for the expression and determining if any row (i.e., any truth assignment to the variables) gives the value True for the expression. For an expression with $n$ variables, this involves listing all $2^n$ truth assignments. This requires an amount of work

[1] You can start interacting with it by just entering the command `sage` at the Linux command line. It will then give you a new prompt, `sage:`, and you can enter SageMath commands and see how it responds. But you would need to learn more in order to know how to interact usefully with it.

that is exponential in the number of variables. So it will not be very efficient, even for modest-sized expressions. (The fact that truth tables are exponentially large, in the number of variables, is closely related to the fact that converting a Boolean expression to DNF can lead to an expression that is exponentially large in the number of variables.) SageMath also provides access to more sophisticated satisfiability-testers. They too can take exponential time in the worst case, though in practice they are much faster than `is_satisfiable()` (but they take some more effort to use for the first time).

## Problem 4. [2 marks]

Copy your `awk` script from Problem 2 and then modify it so that, when you run it on a digraph $G$ (with same input file as before) it creates the following one-line command:




So, instead of just outputting $\varphi_G$, we now output $\varphi_G$ with extra stuff before and after it. The new stuff before it is the text string "`sage -c 'print(propcalc.formula("`", and the new stuff after it is the text string "`").is_satisfiable())'`". These new strings don't depend on $\varphi_G$; they just provide what is needed to make a valid Linux command that invokes `sage` to test whether or not $\varphi_G$ is satisfiable.

Put your answer in a file called `prob4.awk`.

You should test your `prob4.awk` on several different graphs and, for each, use `sage`, as described above, to determine if it is satisfiable. You should ensure that satisfiability of $\varphi_G$ does indeed correspond to $G$ having a kernel.

Figure 2 illustrates the relationships between the files and actions involved in Problem 4.

*See page 11 for Problem 5.*

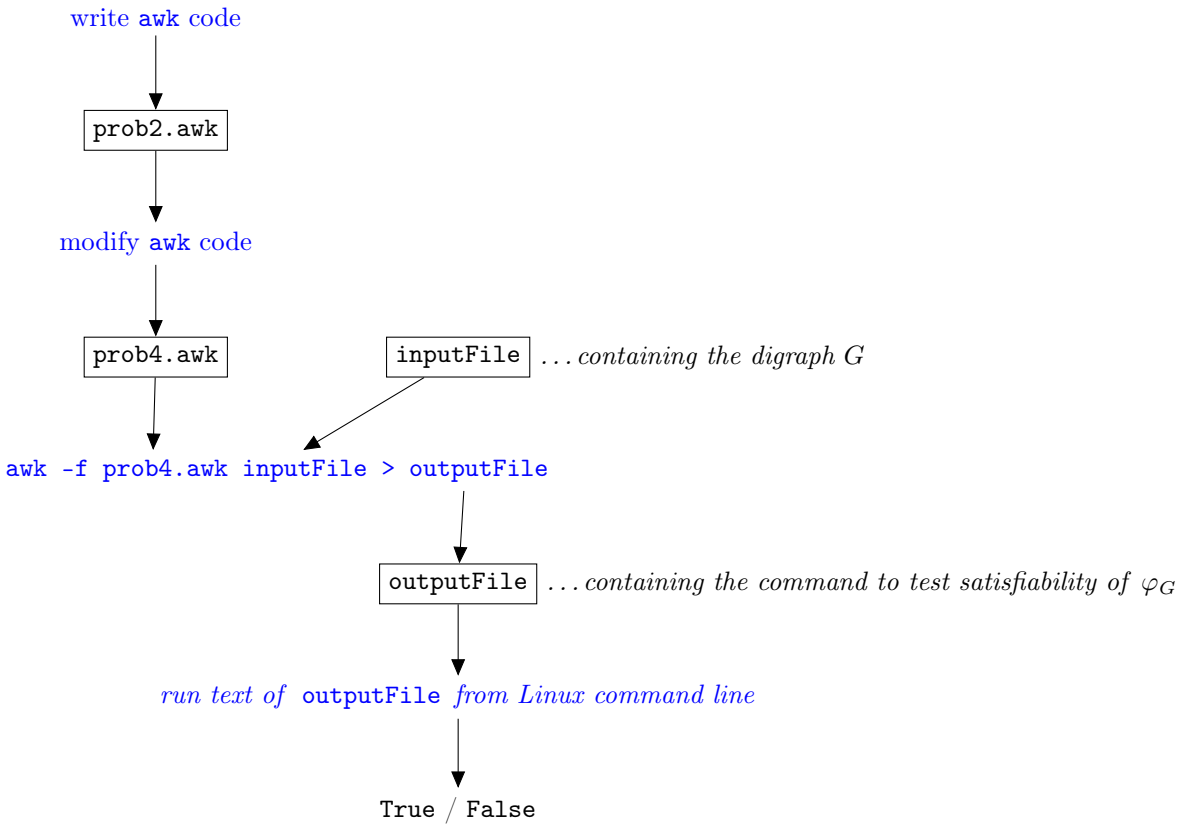



Figure 2: The plan for Problem 4.

## Introduction to Problem 5.

A Boolean expression $\varphi$ in CNF is called **frumious** if every clause has a variable that appears in only one form of literal in $\varphi$. If $x$ is such a variable, then this means that $x$ only ever appears positively (i.e., without negation, so just appears as $x$), or it only ever appears negatively (i.e., negated, as $\neg x$).

For example, the following Boolean expressions are frumious:

| | |
|---|---|
| $(\neg x \vee \neg y) \wedge (\neg x \vee y)$ | First clause contains variable $x$, via literal $\neg x$, and this variable only appears in the expression in this negative literal form. Second clause also contains $x$ in the same literal form, $\neg x$. So each clause contains a variable which only appears in one form. |
| $(x \vee z) \wedge (\neg y \vee \neg z) \wedge (\neg w \vee x \vee \neg y)$ | First clause contains $x$, which only appears positively. Second clause contains $y$, which only appears negatively. For the third clause, we could use any of its variables $w, x, y$, as each of them only appears in one form. |

The following expressions are *not* frumious:

| | |
|---|---|
| $(\neg x) \wedge (x \vee y)$ | First clause only has one variable, namely $x$, which appears in both positive form (in second clause) and negative form (in first clause). So the first clause fails to satisfy the required condition. It's not enough, in this case, that the second clause has a variable, namely $y$, that only appears positively in the expression. |
| $(w \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (\neg w \vee x \vee \neg y \vee z)$ | Variables $w, y, z$ each appear in both positive and negative forms, so so the first clause has no variable that appears in only one form. Variable $x$ only appears positively, but in this case, that's not enough. |
| $a \vee (b \wedge (c \vee d))$ | not in CNF |

If $\varphi$ is a Boolean expression, then we denote its number of clauses by $m$ and its number of variables by $n$.

**Problem 5. [6 marks]**

Prove, by induction on $m$, that a frumious Boolean expression in CNF with $m$ clauses is satisfiable.

Put your answer in a PDF file called prob5.pdf