Monash University
Faculty of Information Technology
2$^{\text{nd}}$ Semester 2023

FIT2014
Assignment 1
Linux tools, logic, regular expressions, induction
DUE: 11:55pm, Friday 18 August 2023 (Week 4)

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

Instructions

Please read these instructions carefully **before** you attempt the assessment:

- To begin working on the assignment, 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 locations and 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.
- Before you attempt any problem—or seek help on how to do it—be sure to read and understand the question, as well as any accompanying code.
- 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:

- a one-line text file, `prob1.txt`, with your solution to Problem 1;
- 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.

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 graph, a logical expression that describes the conditions under which the graph is 3-colourable.

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

Introduction to `awk`

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 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 `NR`,
- the function `printf(···)`,
- `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.)
- `https://www.grymoire.com/Unix/Awk.html`
- the Wikipedia article looks ok
- the `awk` manpage
- the GNU Awk User's Guide.

Introduction to Problems 1–4

Many systems and structures can be modelled as graphs (abstract networks). Many problems on these structures require allocation of some scarce resources to the objects in a network in such a way that objects that are close together do not share the same resource. For example, in timetabling, two classes with some students in common should not be scheduled in the same timeslot; in communications networks, two neighbouring cellphone towers should not be assigned the same transmission frequency (to minimise interference). This wide family of problems, drawn from many different domains, can be modelled abstractly by **graph colouring**. To keep things simple for the assignment, we will only use a few colours, but in real applications the numbers of colours used can be very large.

Throughout, we use G to denote a graph, n denotes its number of vertices and m denotes its number of edges.

A **3-colouring** of a graph G is a function $f : V(G) \to \{\text{Red}, \text{White}, \text{Black}\}$ such that adjacent vertices are given different colours by f. Here, $V(G)$ denotes the set of vertices of G. A graph is **3-colourable** if it has a 3-colouring.

For example, the graph in Figure 1(a) is 3-colourable, and an actual 3-colouring is shown for it.

Note that a 3-colouring does not have to use all three available colours. The graph in Figure 1(b) is shown with a 2-colouring, using just the colours Black and White, so it is 2-colourable. It is also 3-colourable, since every 2-colouring is also a 3-colouring.

The graph in Figure 1(c) is not 3-colourable. Study it carefully and check that this assertion is correct.

In Problems 1–4, you will write a program in `awk` that constructs, for any graph G, a Boolean expression φ_G in Conjunctive Normal Form that captures, in logical form, the statement that G is 3-colourable. This assertion might be True or False, depending on G, but the requirement is that

$$G \text{ is 3-colourable} \quad \Longleftrightarrow \quad \text{you can assign truth values to the variables of } \varphi_G \text{ to make it } \mathsf{True}.$$

For each vertex $i \in V(G)$ and each colour $c \in \{\text{Red}, \text{White}, \text{Black}\}$, we introduce a Boolean variable $v_{i,c}$. It is our *intention* that this represents the statement that "vertex i gets colour c" (which might be True or False). So, for example, we want the variable $v_{2,\text{Red}}$ to represent the

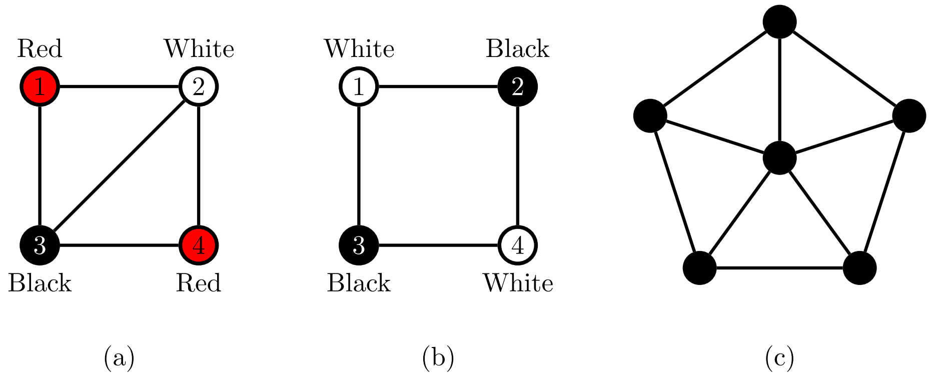

Figure 1: Three graphs. (a) A 3-colourable graph, with a 3-colouring. (b) Another 3-colourable graph. This one is also 2-colourable. It is shown with a 2-colouring, which is also a 3-colouring. (c) A non-3-colourable graph.

statement "vertex 2 gets colour Red". But we will need to build some logic, in the form of a CNF expression, to make this interpretation work.

When we write code, each variable $v_{i,c}$ is represented by a name in the form of a text string `v`ic formed by concatenating `v`, i and c. So, for example, $v_{2,\mathrm{Red}}$ is represented by the name `v2Red`.

If all we have is all these variables, then there is nothing to control whether they are each True or False. Their values might not bear any relation to their intended meaning. For example, we might have $v_{4,\mathrm{Red}} = \mathsf{True}$ and $v_{4,\mathrm{Black}} = \mathsf{True}$, meaning that vertex 4 gets two colours instead of one. Or we might have $v_{1,\mathrm{Red}} = v_{1,\mathrm{White}} = v_{1,\mathrm{Black}} = \mathsf{False}$, meaning that vertex 1 gets no colour at all. Or we might violate the constraint that adjacent vertices get different colours: if vertices 2 and 3 are adjacent, and $v_{2,\mathrm{Red}} = v_{3,\mathrm{Red}} = \mathsf{True}$, then the interpretation is that both these vertices get colour Red even though they are adjacent, which breaks the rules.

So, we need to encode the definition of 3-colourability into a CNF expression using these variables. The expression we construct must depend on the graph. Furthermore, it must depend *only* on the graph. Think of the graph as uncoloured: you cannot assume that any vertex gets any specific colour.

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 shown on the right (Fig. 2), construct a Boolean expression φ_H in CNF using the variables $v_{i,c}$ which is True if and only if the assignment of truth values to the variables represents a 3-colouring of H.

Now type this expression φ_H into a one-line text file, using our text names for the variables (*i.e.*, `v1Red`, `v1White`, `v1Black`, `v2Red`, *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`.

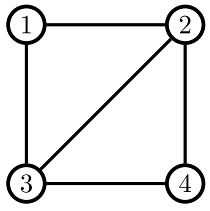

Figure 2:
The graph H.

For the purposes of this assignment, every graph is encoded as an *edge list* in the following format:

- The first line specifies the number of vertices, as a single positive integer. Let n be the number of vertices. Then the vertices of the graph are assumed to be numbered $1, 2, \ldots, n$.
- Each subsequent line contains a single edge, represented as

  $$i \texttt{ -- } j$$

  where i and j are positive integers representing the two vertices linked by the edge (with $1 \le i \le n$ and $1 \le j \le n$).
- We allow any number of spaces before, between, and after the numbers on each line, subject to the requirement that there be at least one space on either side of the double-hyphen `--`.
- For example, the graph in Figure 1(a) could be represented by the six-line file on the left below, or (less neatly) by the one on the right.

```
4
1 -- 2
1 -- 3
2 -- 3
2 -- 4
3 -- 4
```

```
      4
   1 -- 2
    1  --    3
  2   -- 3
2 -- 4
   3 -- 4
```

- Each graph is represented in a file of its own. Each input file contains exactly one graph 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]

Write an `awk` script that takes, as input, a graph G represented in the specified format in a text file, and produces, as output, a one-line file containing the text representation of a Boolean expression φ_G in CNF which uses the variables we have specified and which is `True` if and only if the variables describe a 3-colouring of G.

The text representation is the same as that described on p4 and used for Problem 1.

Put your answer in a file called `prob2.awk`.

Problem 3. [2 marks]

Give, with brief justification, an expression for the number of clauses of φ_G in terms of n (the number of vertices of G) and m (the number of edges of G).

For full marks, the number of clauses produced by your `awk` program (`prob2.awk`) must be correct as well as equalling the expression you give here.

Put your answer 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.

Problem 4. [2 marks]

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

$$\texttt{sage -c '}\underbrace{\texttt{print(propcalc.formula("}\overbrace{\cdots\cdots\cdots\cdots\cdots}^{\text{text representation of }\varphi_G}\texttt{").is\_satisfiable())}}_{\text{SageMath command}}\texttt{'}$$

So, instead of just outputting φ_G, we now output φ_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 φ_G; they just provide what is needed to make a valid Linux command that invokes `sage` to test whether or not φ_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 φ_G does indeed correspond to 3-colourability of G.

[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.

Figure 3 illustrates the relationships between the files and actions involved in Problem 4.

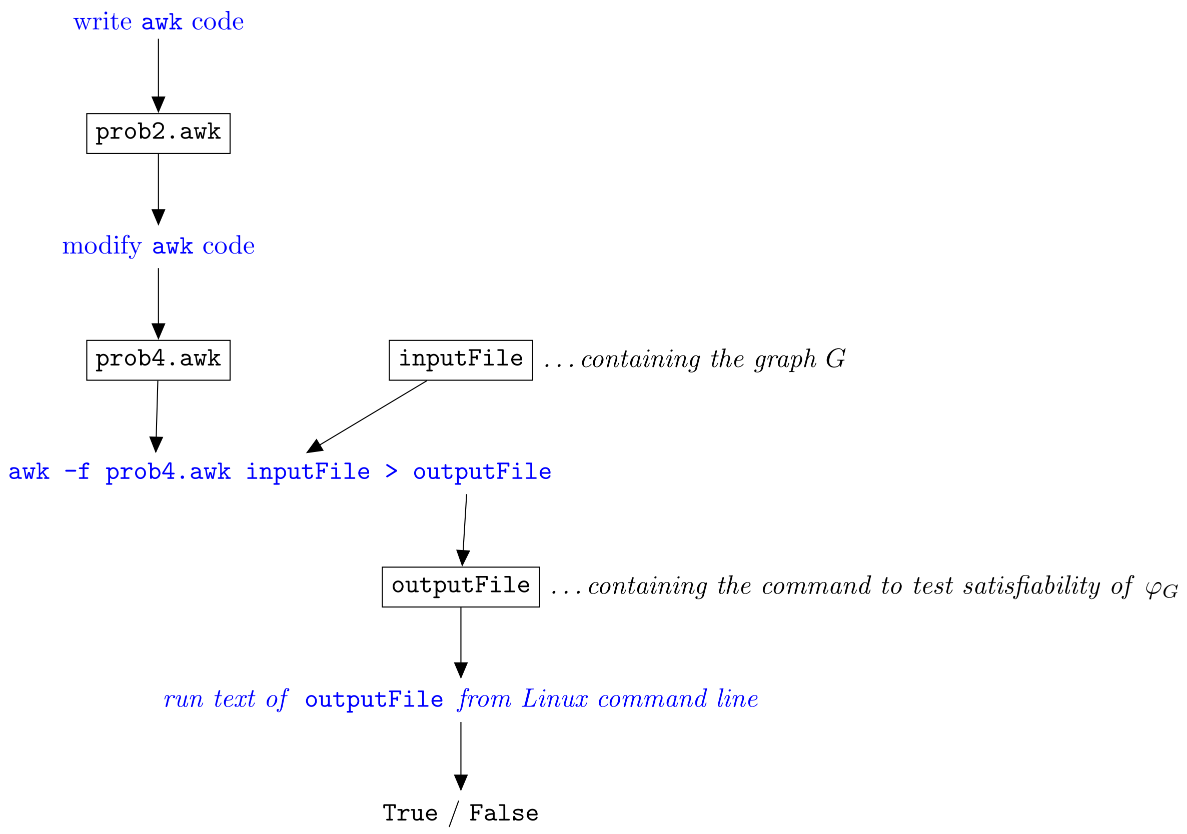
write awk code
prob2.awk
modify awk code
prob4.awk
inputFile
. . . containing the graph G
awk -f prob4.awk inputFile > outputFile
outputFile
. . . containing the command to test satisfiability of φ_G
run text of outputFile from Linux command line
True / False

Figure 3: The plan for Problem 4.

Introduction to Problem 5.

A Boolean expression in CNF is **slithy** if, for all $k \geq 1$, every set of k clauses includes a variable that appears only once among those clauses. So this condition must hold for *every possible set* of clauses in the expression.

In counting appearances of variables among a set of clauses, we don't mind whether the variable appears in its normal or negated form; we count them all.

Examples:

- The expression
$$(a \vee b) \wedge (b \vee \neg c) \wedge (c \vee \neg d)$$
is slithy. To see this, consider each k and each set of k clauses. First, consider $k = 1$. Each clause in the expression includes a variable that appears once in that clause (in fact, both variables in each clause appear exactly once in that clause). They might appear in other clauses too, but that doesn't matter when we consider what happens in just one specific clause. Now consider $k = 2$. If we take the two clauses $a \vee b$ and $b \vee \neg c$, we see that a appears only once among those two clauses, and the same holds for c. You can check that the other pairs of clauses have the required property too. Finally, consider $k = 3$. Now we have to consider all three clauses. The variable a appears only once (as does the variable d), so the condition is satisfied.

- The expression
$$(a \vee b) \wedge (b \vee \neg c) \wedge (\neg a \vee \neg c) \wedge (a \vee \neg d)$$
is not slithy. There are *some* sets of clauses that have the required property: for example, the first two clauses (being the same as before) still do. But consider the first, second and third clauses. Between them, they contain two appearances of a, two appearances of b, two appearances of c, and no appearances of any other variable. So this set of clauses does not have a variable that only appears once in those clauses.

Problem 5. [7 marks]

Prove, by induction on n, that a slithy Boolean expression in CNF with at most n variables has at most n clauses and is satisfiable.