Monash University
Faculty of Information Technology
2nd Semester 2024

FIT2014
Assignment 1
Linux tools, logic, regular expressions, induction
DUE: 11:55pm, Friday 16 August 2024 (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 resource to the objects in a network in such a way that there is suitable "coverage" of that resource throughout the network. For example, in a city road network, it may be necessary to place cameras at various road intersections, for traffic management or security, in such a way that every road throughout the network can be viewed by at least one camera. However, cost factors mean that there are constraints on the total number of cameras that can be placed.

This can be modelled abstractly by the **vertex cover** problem, which is a well-known graph theory problem that we will consider here.

Throughout, we use G to denote a graph, n denotes its number of vertices and m denotes its number of edges. $V(G)$ denotes the set of vertices of G, and $E(G)$ denotes the set of edges of G.

A **vertex cover** of a graph G is a set of vertices X in $V(G)$ such that every edge in $E(G)$ has at least one endpoint in X.

In this assignment, we are interested in a particular variant of the vertex cover problem in which **triangles** cannot be included in the vertex cover. A triangle in G is a set of three vertices, all of which are pairwise-adjacent (i.e., every pair in the set is joined by an edge). A vertex cover of G that contains no triangles is called a **triangle-free vertex cover**.

For example, in the graph H in Figure 1, the vertex sets $\{1, 3, 6, 7\}$, $\{2, 3, 4, 7\}$, and $\{1, 2, 5, 7\}$ all form triangle-free vertex covers. But $\{1, 2, 3, 7\}$ and $\{1, 3, 5, 7\}$ are not triangle-free vertex covers, the first because it contains the subset $\{1, 2, 3\}$, which forms a triangle in H, and the second because it does not meet every edge in H. (Specifically, it misses the edge from 2 to 6.)

Although every graph has a vertex cover, not all graphs have a triangle-free vertex cover. An example of a graph with *no* triangle-free vertex cover is shown in Figure 2.

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 has a triangle-free vertex cover. This assertion might be True or False, depending on G, but the requirement is that

G has a triangle-free vertex cover $\iff$ you can assign truth values to the variables of φ_G to make it True.

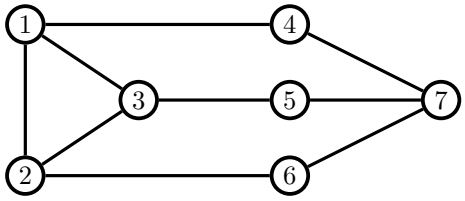


Figure 1: A graph, H, which has a number of different triangle-free vertex covers.

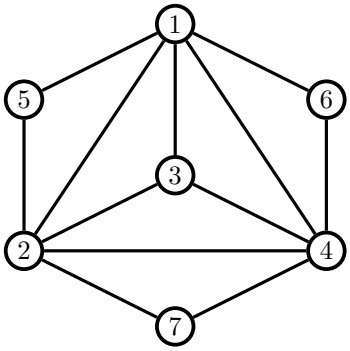


Figure 2: A graph with no triangle-free vertex cover.

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 triangle-free vertex cover" (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 triangle-free vertex cover into a CNF expression using these variables. The expression we construct must depend on the graph. Furthermore, it must depend *only* on the graph.

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 in Figure 1, construct a Boolean expression φ_H using the variables v_i which is True if and only if the assignment of truth values to the variables represents a triangle-free vertex cover of H.

Now type this expression φ_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 graph G on n vertices, and constructing a corresponding Boolean expression φ_G with n variables, in such a way that:

- the variables of φ_G correspond to vertices in G, and
- φ_G evaluates to True if and only if the assignment of truth values to those variables corresponds to a triangle-free vertex cover 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 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 $1 \leq i \leq n$ and $1 \leq j \leq n$).
- 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 could be represented by the ten-line file on the left below, or (less neatly) by the one on the right.

```
7
1 -- 2
1 -- 3
1 -- 4
2 -- 3
2 -- 6
3 -- 5
4 -- 7
5 -- 7
6 -- 7
```

```
      7
   1 -- 2
    1  --    3
  1   -- 4
2 -- 3
   2 -- 6
      3   -- 5
  4 -- 7
           5 --  7
       6      -- 7
```

- 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. [5 marks]

Complete the partial `awk` script in Figure 3 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 φ_G in CNF which uses the variables we have specified and which is True if and only if the variables describe a triangle-free vertex cover 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`.

```
BEGIN  {

}

#  The next line of code gets the number of vertices from the first line
#   and stores it in the variable n
NR == 1  { n = $1; }

#  Write code below to generate clauses ensuring that
#   each edge must be included in the vertex cover

Your code goes here

#  The line below should work in conjunction with what you wrote
#  above, to add each edge into an associative array for checking later

a[$1, $3] = 1;
a[$3, $1] = 1;

}


END  {
# Looping through triples of vertices in the array we created
for  (i = 1; i <= n-2; i++)
{
     for (j = i+1; j <= n-1; j++)
     {
          if (a[i, j] == 1) # (i,j) edge found, now check for triangles
          {
               for (k = j+1; k <= n; k++)
               {
                    if (a[i, k] == 1 && a[j, k] == 1)
                    {
                         # Triangle found! Write code here to ensure
                         # the triangle is excluded from the vertex cover


                           Your code goes here


                    }
               }
          }
     }
}
}
```

Figure 3: A partial awk script to use for Problem 2

Problem 3. [4 marks]

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

(b) Give, with brief justification, an expression for an upper bound on the number of triangles t in G in terms of m. (You don't need to find the tightest bound possible, but it needs to be tight enough to support your argument in part (c).)

(c) Based on your answers to (a) and (b), give with brief justification an upper bound on the number of clauses of φ_G, expressed *only* in terms of m.

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:

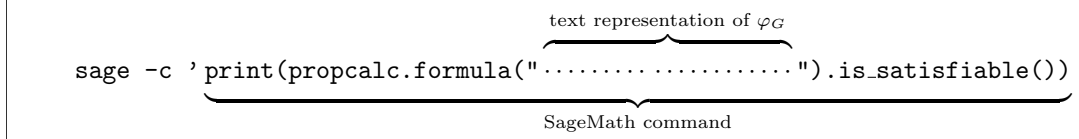
```
$ sage
sage: print(propcalc.formula("(a | b) & (~a | ~b) & (~a | b)").is_satisfiable())
True
```

Again, the output `True` indicates satisfiability.

[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.

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:



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 of φ_G does indeed correspond to G having a triangle-free vertex cover.

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

Introduction to Problem 5.

A graph G is said to be **2-regular** if every vertex in G has *exactly* two neighbours (i.e., has degree 2). 2-regular graphs are not necessarily connected, however, each connected component in a 2-regular graph must be a circuit. The smallest possible 2-regular graph has three vertices and three edges.

The graph X shown in Figure 5 is an example of a 2-regular graph with two connected components.

Problem 5. [7 marks]

(a) Using the same approach as you did in Problem 1, construct a Boolean expression φ_X using variables v_i, where the expression is True if and only if the assignment of truth values to the variables represents a triangle-free vertex cover of X. The relationship between the number of clauses in your expression and the number of edges and triangles in X should match the expression that you gave in Problem 3 (a).

(b) Recall the process described at the top of page 5 of taking any graph G and constructing a Boolean expression φ_G from that graph. In this question, we look at a restricted version of this problem where only 2-regular graphs are considered.

For any 2-regular graph G, let φ_G be a Boolean expression with n variables that is constructed from G, as described at the top of page 5.

Prove by induction that φ_G has at most $\frac{4n}{3}$ clauses.

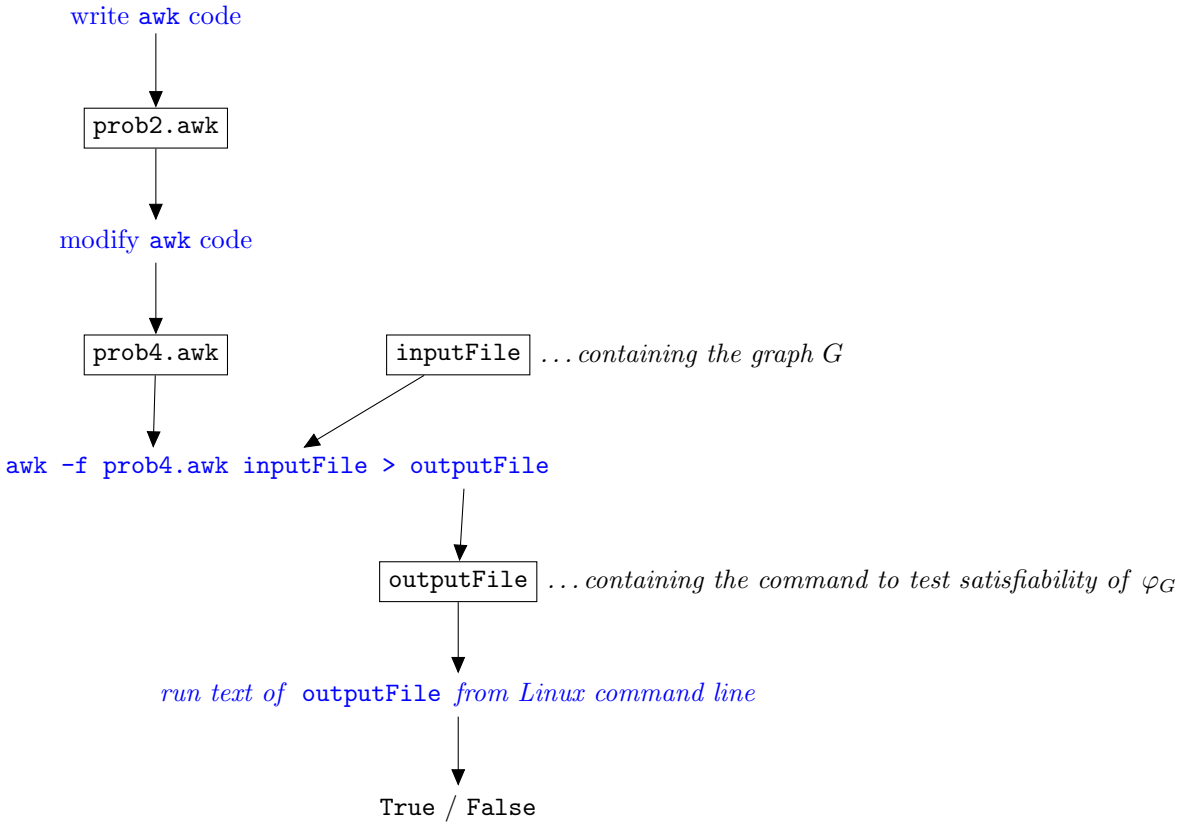


Figure 4: The plan for Problem 4.

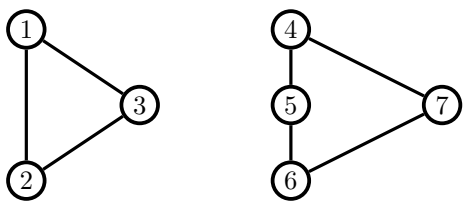


Figure 5: A 2-regular graph, X