

**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 `asn1.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 `asn1` 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.

这是莫纳什大学 FIT2014 《Linux 工具、逻辑、正则表达式、归纳法》作业 1 的首页说明。按原文意思，关键点如下（我只解释要求，不参与解题）：

## 作业基本信息

- **课程/作业：** FIT2014 作业 1 (Linux 工具、逻辑、正则表达式、归纳法)
- **截止时间：** 2025 年 8 月 22 日 (学期第 4 周) 周五晚 **11:55pm**
- 建议**尽早开始做**；有问题带去咨询时段或在 Ed 论坛提问。

## 硬性规则（必须遵守）

- **禁止使用生成式 AI 工具** 参与本作业的任何部分（举例：GPT、DeepSeek、Copilot、Claude、Cohere、Gemini/Bard 等）。
- **必须独立完成**，不得与他人合做或共享答案。
- **从 Moodle 下载**作业工作台 (workbench) 的 **.zip** 压缩包。
  - 在 **Ed** 上创建一个新的 Workspace，并**上传该 zip**；Ed 会自动解压。
  - 打开并**编辑 student-id 文件**，填入你的姓名和学号。可参考 Lab 0 里关于这些操作的说明。
- 作业工作台里已经给好了**占位文件/空文件**（对应各题要交的文件名）。**不要新增或删除**这些文件。
- **书面题的答案**必须以 **PDF** 形式提交：
  - 可以把清晰可读的**手写**解答扫描成 PDF；
  - 也可以**电脑打字**后导出为 PDF；
  - 若提交为 Word 等**非 PDF 格式会被扣分**；Lab 0 里有如何导出/上传及在 Ed 里替换占位文件的提醒。
- **每个 PDF 的第一页**都必须写上**姓名和学号**。
- 完成后，在 Ed 的文件面板点击 **“Download All”**，把整个 Workspace **打包成 zip**，并在**截止前**把这个 zip **提交到 Moodle**。
- **必须遵守命名/格式约定**（适用于文件、目录、代码、数学书写等）。不遵守会被扣分。提交前务必**自查**。

## 你最终提交的内容应当包含

（这些文件名是固定要求，最初在工作台里都是“空壳”，提交前需要用你的答案替换）

- `student-id`：写明你的姓名与学号（文本文件）。
- `prob1.txt`：**一行文本**，为第 1 题的指定答案/表达式/命令（按题面要求）。
- `prob2.txt`：**一行文本**，为第 2 题按题面要求填写的模式/表达式。
- `prob2.awk`：用于第 2 题的 **awk 脚本**。
- `prob3.pdf`：第 3 题的 **PDF 解答**。
- `prob4.awk`：第 4 题的 **awk 脚本**。
- `prob5.pdf`：第 5 题的 **PDF 解答**。

备注：工作台起始目录里已经放好了这些**空文件 (dummy files)**，文件名与上面一致。提交前你要把它们**逐一替换**为你自己的解答文件，并且确认 `student-id` 已正确填写。

如果你需要，我可以把这些要求做成一份**自查清单**或**提交前检查表**，方便对照逐项核对。

## 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 inputFile
```

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

下面这页是在**解释本次作业要做什么**，以及**awk 程序是怎样工作的**。我按原文逐条用中文说明（不超出原文信息）：

## 作业简介 (Introduction to the Assignment)

- 在 Lab 0 里你接触过 **sed**（流编辑器），它能按**模式**匹配逐行处理文本，这些模式通常用**正则表达式**来写。
- 本次作业将改用 **awk**——它和 sed 有相似之处，但功能更多；awk 是 Unix/Linux 上常用的小型编程语言，也大量使用**正则表达式**。
- **第 1–4 题**：你需要**编写 awk 程序**。目标是：针对任意一个**有向图**，构造一个**逻辑表达式**，用来描述该有向图在什么条件下**存在一个 kernel（核）**。
- **第 5 题**：关于**归纳法**（induction）的应用，研究某些**合取范式（CNF）布尔表达式**的结构与可满足性。

## awk 简介 (Introduction to awk)

- awk 程序**逐行读取输入文件**并处理。awk 程序中的每一行通常写成下面形式：

```
bash
```

复制 编辑

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

- **pattern**：一个**正则表达式**（或其他特殊模式）。
  - **action**：当输入文件中的某一行**匹配该 pattern** 时，要对这行执行的操作。
  - 可以**省略 action**（连同花括号）。**若省略 action，匹配到的那行会被直接打印。**
  - 也存在**特殊模式** **BEGIN** 与 **END**，用于在读入任何数据之前/之后执行指定操作。
- 写好程序后**无需编译**，可直接在 Linux 下运行，例如：

```
ruby
```

复制 编辑

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

- awk 的执行流程（页面里的伪代码框）大意如下：
  1. **开始时**：如果程序里有 `BEGIN` 这一行，就**先执行**它对应的 action。
  2. **主循环**：对输入文件**每一行**依次处理：
    - 取到当前的 `inputLine`；
    - **遍历程序中的各条规则**（忽略 `BEGIN / END` 行）：
      - 如果 `inputLine` **匹配**当前规则里的 `pattern`：
        - 若该规则**写了** `action`，就**执行这个 action**；
        - 若**没写** `action`，就**直接打印**这行（输出到标准输出）。
      - 不匹配就继续检查下一条规则。
  3. **结束时**：如果程序里有 `END` 这一行，就**执行**它对应的 action。
- **所有输出**都会写到**标准输出**（终端）。
- 文末提示：你应该去阅读 awk 的基础知识（接下来页会继续列出要点）。

下面这页是在**解释本次作业要做什么**，以及**awk 程序是怎样工作的**。我按原文逐条用中文说明（不超出原文信息）：

## 作业简介 (Introduction to the Assignment)

- 在 Lab 0 里你接触过 **sed**（流编辑器），它能按**模式**匹配逐行处理文本，这些模式通常用**正则表达式**来写。
- 本次作业将改用 **awk**——它和 sed 有相似之处，但功能更多；awk 是 Unix/Linux 上常用的小型编程语言，也大量使用**正则表达式**。
- **第 1–4 题**：你需要**编写 awk 程序**。目标是：针对任意一个**有向图**，构造一个**逻辑表达式**，用来描述该有向图在什么条件下**存在一个 kernel（核）**。
- **第 5 题**：关于**归纳法**（induction）的应用，研究某些**合取范式（CNF）布尔表达式**的结构与可满足性。

## awk 简介 (Introduction to awk)

- awk 程序**逐行读取输入文件**并处理。awk 程序中的每一行通常写成下面形式：

```
bash
```

[复制](#) [编辑](#)

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

- **pattern**：一个**正则表达式**（或其他特殊模式）。
  - **action**：当输入文件中的某一行**匹配该 pattern** 时，要对这行执行的操作。
  - 可以**省略 action**（连同花括号）。**若省略 action，匹配到的那行会被直接打印。**
  - 也存在**特殊模式** **BEGIN** 与 **END**，用于在读入任何数据之前/之后执行指定操作。
- 写好程序后**无需编译**，可直接在 Linux 下运行，例如：

```
ruby
```

[复制](#) [编辑](#)

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

- 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  $i = 1, 2, \dots, n$  (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.

下面这页主要分两部分：上半部分是“学 awk 需要掌握的最少知识 + 参考资料”，下半部分是“题目 1–4 的背景故事（图论/监控与报警）”。我按原文要点逐句解释：

## 一、学 awk 需要掌握的内容

- **正则表达式的写法**：awk 用正则来匹配文本模式，这是使用 awk 的核心。
- **字段变量** `$1`, `$2`, ... 以及 `NF`：`$1` 表示每行的第 1 个字段、`$2` 表示第 2 个字段，以此类推；`NF` 是“当前行字段数”。
- **`printf(...)` 函数**：像 C 语言那样按格式输出（控制对齐、小数位数等）。
- **`if` 语句**：按条件执行不同的动作。
- **`for` 循环**：只需要会最常见的这种形式即可

awk

复制 编辑

```
for (i = 1; i <= n; i++) {  
    # 循环体  
}
```

这段循环会让 `i` 依次取 `1, 2, ..., n`，每次执行一次循环体（除非你在循环体里改动了 `i`，一般不需要那么做）。

循环可以**嵌套**（一个循环里再写一个循环）。也可以把花括号写成一行的紧凑形式，但要**保证可读性**。

## 入门/参考资料（原文建议）

- 《**The AWK Programming Language**》（Aho、Kernighan、Weinberger, 1988, Addison-Wesley）。  
建议读第 1 章前几节就能满足大部分需求；另外注意第 28 页的“正则表达式规范”。
- **GNU Awk User's Guide (gawk 用户手册)**
- **维基百科的 awk 条目**（原文说看起来也不错）
- `awk` 的 **man 手册页**
- **Grymoire 的 awk 教程**：很有用，但**不太适合做入门**，因为它把“模式与动作”（awk 的心脏）讲得很靠后，前面花了很多篇幅在其他编程细节上。



## 二、题目 1–4 的引言（监控网络与报警）

- 许多系统/结构可以用图（graph）来建模；如果边带方向，就叫有向图（digraph），图上常用箭头表示方向。
- 设想你有一组**设备**。每台设备可以**监控**若干其他设备。  
监控关系是**有方向**的：**A 监控 B**并不意味着**B 也监控 A**；有的成对设备可能**互不监控**。
- 为了发现网络中的故障，我们想给**部分设备**装**报警器**。

### 报警触发条件：

1. 设备自身出故障；或
  2. 该设备**监控**的某个设备出故障。
- 另外有一个**约束**：如果一台设备监控另一台设备，那么**这两台不能同时装报警器**（可能因为报警器相互干扰，或出于成本限制）。
  - 目标：从所有设备里选择一批装报警器，使得满足：
    1. **覆盖性**：每台设备要么**自己有报警器**，要么**被一台有报警器的设备监控**；
    2. **互不相监**：任何**装了报警器**的设备之间，**不存在监控关系**（谁也不监控谁，也不被对方监控）。

通俗地说：要挑一些“打点”的设备放报警器，既要把全网“照顾到”（每台设备都被自己或他人“管着”），又不能让两台装了报警器的设备彼此存在监控箭头。

（如果你熟悉图论，这相当于在有向图里找一个满足条件的“独立支配集合”的变体；原文没有使用术语，只是把规则叙述出来。）

需要我把这页里的英文逐句翻译成中文，或者配一个小例子画图说明怎么选设备装报警器吗？

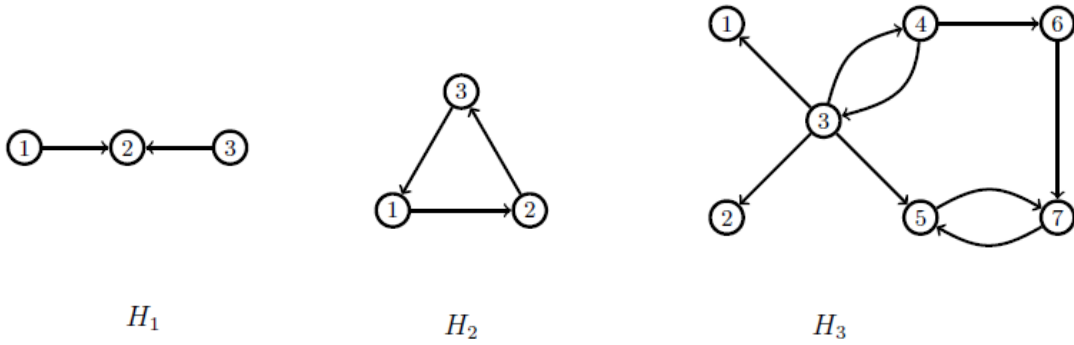


Figure 1: Three digraphs,  $H_1, H_2, H_3$ . The first,  $H_1$ , has a kernel, but the second one,  $H_2$ , does not.

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?

下面这一页是在定义“有向图的 kernel (核)”，并用 3 个小图  $H_1, H_2, H_3$  做示例说明。

## 图与记号

- 用  $G$  表示有向图 (digraph)， $n$  是顶点数， $m$  是有向边 (arc) 数。
- $V(G)$ : 顶点集合； $A(G)$ : 有向边集合。
- 若存在弧  $(v, w)$  (从  $v$  指向  $w$ )，则
  - $v$  是  $w$  的 in-neighbour (入邻居) (有一条进入  $w$  的弧来自  $v$ )；
  - $w$  是  $v$  的 out-neighbour (出邻居) (从  $v$  指向  $w$  的弧从  $v$  “出去”)。
- 在“设备监测”的场景里：顶点表示设备，弧表示设备间的“监测关系”。

## kernel (核) 的定义

集合  $X \subseteq V(G)$  是  $G$  的一个 kernel，当且仅当同时满足：

1. **覆盖/支配性 (domination)**：对每个顶点  $v \in V(G)$ ，要么  $v \in X$ ，要么存在某个  $u \in X$  使得  $(u, v) \in A(G)$ 。  
——意思是每个点要么被选进  $X$ ，要么至少被  $X$  中某点指向 (被“覆盖”)。
2. **独立性 (independence)**：对每条弧  $(v, w) \in A(G)$ ，不能同时有  $v \in X$  且  $w \in X$ 。  
——也就是  $X$  内部不允许有一条从  $X$  中某点指向  $X$  中某点的弧。

直观理解：kernel 是一个相互不指向的“代表点”集合，并且能覆盖所有顶点 (每个点要么就是代表，要么被某个代表指到)。

## 例子 (对应图 1)

- $H_1$ ：集合  $X = \{1, 3\}$  是一个 kernel。
  - 图中不在  $X$  的点只有 2，但存在弧  $(1, 2)$  (或由 3 指向 2 的弧)；因此点 2 被  $X$  覆盖，条件 (1) 满足；
  - 且  $X$  内部没有一条从  $X$  指向  $X$  的弧，条件 (2) 也满足。
  - 反例：
    - $Y = \{2\}$  不是 kernel，因为没有从 2 指向 1 (或 3) 的弧；1、3 得不到覆盖，违背 (1)。
    - $Z = \{1, 2\}$  不是 kernel，因为弧  $(1, 2)$  的两个端点都在  $Z$  中，违背 (2)。
- $H_2$ ：没有 kernel。说明性举例：
  - $\{1\}$  不是 kernel——顶点 3 不在集合里，且从集合中的点没有入弧指向 3 (进入 3 的唯一弧是  $(2, 3)$ ，而 2 不在集合里)，违背 (1)。
  - $\{2, 3\}$  不是 kernel——因为存在弧  $(2, 3)$ ，两个端点都在集合里，违背 (2)。
- $H_3$ ：文本在此提问你去判断：
 

“ $H_3$  有 kernel 吗？如果有，有几个？如果没有，原因是什么？”

——这是一道练习题，要求你按上面的两条条件去检查  $H_3$  的所有可能集合。

如果你愿意，我可以把“如何判定某集合是否为 kernel”的操作步骤清单列出来，帮你自己去验证  $H_3$ 。

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  $vi$  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
$\neg$	<code>~</code>
$\wedge$	<code>&amp;</code>
$\vee$	<code> </code>

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:

$$j_1 \ j_2 \ \cdots \ j_d : i$$

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



这页在说两件事：①题目的总体目标与变量约定；②**Problem 1** 要你先对一个小图手工写出 CNF 公式，并说明以后要写 awk 程序把这个过程自动化。按原文逐点解释如下。

## 1) 题目总体目标 (Problems 1–4)

- 你将补全一个 awk 脚本。它对任意有向图 *digraph* 记作  $\mathbb{V}(G)$  构造一个合取范式 (CNF) 的布尔公式  $\varphi_G$ 。
- 这个公式要精确刻画命题“ $G$  有一个 kernel” (上一页把 kernel 解释成“选一些顶点装报警器：彼此之间不能有监控边；每个顶点要么自己被选中、要么被某个被选中的入邻点监控”)。
- 形式化要求是：  
 $G$  有 kernel  $\Leftrightarrow$  能给  $\varphi_G$  的变量们赋真值使  $\varphi_G$  为真。

### 变量约定

- 对  $G$  的每个顶点  $i$ ，引入一个布尔变量  $v_i$ 。它表达的语义是：“顶点  $i$  被选入 kernel” (真/假)。
- 在代码/文件里，变量名写成文本 **v1, v2, v3, ...** (把字母  $v$  与编号拼起来，如  $v_2$  写 **v2**)。
- 仅有这些变量还不够，你必须用 CNF 的逻辑约束把“kernel 的定义”编码进去；而且这个公式只依赖于图结构 (不依赖别的东西)。

## 2) Problem 1 (2分)

- 给定图  $H_3$  (见 Figure 1 右侧的小图)，请你手工构造一个 CNF 公式  $\varphi_H$ ，使用变量  $v_i$ ，使得：
  - 当且仅当这些变量的一组真值恰好对应  $H$  的一个 kernel 时， $\varphi_H$  为真。
- 把你得到的  $\varphi_H$  写成一行文本，变量名用 **v1, v2, v3, ...**，且把逻辑符号按下表替换成文本写法：
  - $\neg$  (非)  $\rightarrow$  `~`
  - $\wedge$  (与)  $\rightarrow$  `&`
  - $\vee$  (或)  $\rightarrow$  `|`
- 把答案存成单行文件 `prob1.txt`。

小提示 (便于你写  $\varphi_H$ )：

- “独立性：每条有向边  $a \rightarrow b$  都要禁止“同时选  $a$  和  $b$ ”：子句  $(\sim v_a \mid \sim v_b)$ 。”
- “覆盖性：对每个顶点  $i$ ，要求“要么选  $i$ ，要么选它的某个入邻居”：子句  $(v_i \mid v_{j1} \mid \dots \mid v_{jd})$  (若  $i$  没有入邻居，则子句就是  $(v_i)$ ，强制选它)。”
- “把所有这些子句用 `&` 串起来，就是 CNF。”

## 3) 之后要做的通用过程 (Problems 2–4)

- 我们把上面的思路一般化：对任意有向图  $G$  (有  $n$  个顶点)，构造含  $n$  个变量的  $\varphi_G$ ，满足：
  - 变量  $\leftrightarrow$  顶点；
  - $\varphi_G$  为真  $\Leftrightarrow$  这些变量的取值对应于  $G$  的一个 kernel。

接下来作业要求你写一个 awk 脚本把这个构造自动化。



## 4) 输入图的编码格式（供 awk 脚本读取）

这是一种“非标准但本题专用”的文本格式。

- **第一行**：两个正整数 `n m`（至少一个空格分隔，行首行尾可以有空白）。
  - `n` 是顶点数，`m` 是边数。
- **其余每一行**是一个**顶点行**，格式：

less

复制 编辑

```
j1 j2 ... jd : i
```

其中  $i$  与  $j_1, j_2, \dots, j_d$ （允许  $d \geq 0$ ）都是**正整数**，并且  $j_1, \dots, j_d$ **两两不同**。这表示：

- $i$  的**入邻居**是  $j_1, j_2, \dots, j_d$ ；
- 因而图中存在有向边  $j_k \rightarrow i$ （对所有  $k = 1..d$ ）。

### 例子

如果有一行

yaml

复制 编辑

```
2 5 7 : 6
```

表示存在三条边  $2 \rightarrow 6$ ,  $5 \rightarrow 6$ ,  $7 \rightarrow 6$ ；顶点 6 的入邻居是  $\{2, 5, 7\}$ 。

**你需要做的**（本页的落地事项）：

1. 先对给定小图  $H_3$  手工写出  $\varphi_H$ ，按 `~ & |` 记号写成**一行**放进 `prob1.txt`。
2. 后续题目里，用同样规则写 awk 程序：读入上面的图格式，自动输出对应的 CNF 公式  $\varphi_G$ 。

如果你愿意，我可以根据 Figure 1 的  $H_3$  帮你把 `prob1.txt` 的公式也直接写出来（发给我那张图或说明边集即可）。

going *into* vertex  $i$  are  $(j_1, i), (j_2, i), \dots, (j_d, i)$ . 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`.



这页讲两件事：（1）输入文件如何表示有向图；（2）Problem 2 的具体交付物。我按原文逐条解释：

## 一、输入文件格式（digraph 的文本表示）

- 共有两类有效行：
  - 规模行**： `n m`（两个正整数），分别是顶点数和弧数。
  - 顶点行**： `j1 j2 ... jd : i`
    - 表示所有指向顶点  $i$  的弧是  $(j_1, i), (j_2, i), \dots, (j_d, i)$ 。
    - 这一行就是顶点  $i$  的完整入邻居列表。
- 空格规则**：数字之间与冒号两侧允许任意个正数的空格（至少 1 个）。行首在第一个  $j_1$  前、行尾在  $i$  后也都可以有空格。
- 完备性**：对每个  $i$  ( $1 \leq i \leq n$ )，恰好有一行对应它的顶点行。
- 忽略无关行**：不是上面两种格式的行可以出现在文件里，但你的程序必须忽略它们。
- 行的顺序任意**： `n m` 不必在开头，各顶点行也不必按编号排序。
- 一个文件只表示一个图**：每个输入文件只包含一个这样表示出来的 digraph。
- 数字规范**： `n` 和所有顶点编号都是正整数，位数不限，不能带小数点，也不能有前导 0。

示例（图中  $H_3$  的两种等价表示）

左边 8 行文件：

yaml

复制 编辑

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

含义是：

- $n = 7, m = 9$ ;
- `3 : 1` 表示  $(3, 1)$ ; `3 : 2` 表示  $(3, 2)$ ; `4 : 3` 表示  $(4, 3)$ ; `3 : 4` 表示  $(3, 4)$ ;
- `3 7 : 5` 表示  $(3, 5), (7, 5)$ ; `4 : 6` 表示  $(4, 6)$ ; `5 6 : 7` 表示  $(5, 7), (6, 7)$ 。

右边文件把 `7 9` 放在后面，还多了一行 `***FIT2014 : A1***` 这样的文本行——这行**不符合两种格式**，所以**被允许出现但必须忽略**。

## 二、Problem 2 (7 分)

- 你会拿到一个**部分完成的** awk 脚本 `prob2.awk` (题目下方会给出)。
- 任务：完善它，使之**读入**按上述格式表示的图  $G$ ，**输出**一行文本，内容是一个**布尔表达式**  $\varphi_G$  的**文本表示**，该式为 **CNF (合取范式)**，并使用题目指定的变量；而且**当且仅当**这些变量描述了  $G$  的一个 **kernel (核)** 时， $\varphi_G$  为 True。
- 文本表示法与前面第 5–6 页、Problem 1 使用的**完全相同**。
- 需要提交**两个文件**：
  - `prob2.txt`：只有一行，写你用于**匹配“顶点行”**的 awk **模式**；
  - `prob2.awk`：你完善后的 awk 程序。它在匹配顶点行时用的**模式必须与** `prob2.txt` **中的一致**。

注：以上只是对格式和要求的解释与翻译，不包含任何解题思路或代码。

$^(\backslash s^*[0-9]+\backslash s^*)^*\backslash s^*:\backslash s^*[0-9]+\backslash s^*$$

在正则表达式中，`\s` 并不表示**任意多个空格**，而是表示**任意空白字符**。它包含了以下几种类型的字符：

1. **空格 (space)**：普通的空格字符。
2. **制表符 (tab)**：表示水平制表符 (`\t`)。
3. **换行符 (newline)**：表示换行 (`\n`)。
4. **回车符 (carriage return)**：表示回车 (`\r`)。
5. **换页符 (form feed)**：表示换页 (`\f`)。
6. **垂直制表符 (vertical tab)**：表示垂直制表符 (`\v`)。

因此，`\s` 代表的是**所有空白字符**，不仅仅是空格，而是所有可以在文本中用作间隔的字符。

`\s` 和 `\s*` 的区别：

- `\s`：表示**一个空白字符**，它可以是空格、制表符、换行符等。
- `\s*`：表示**零个或多个空白字符**。这意味着，它可以匹配没有空白字符的地方，也可以匹配多个空白字符，比如多个空格、制表符等。

你提出的正则表达式 `\s*(\s*[0-9]+\s*)*\s*:\s*[0-9]+\s*$` 也基本是有效的，但会出现一些潜在的问题，尤其是在处理空格时。让我们详细分析一下它与我们原来的正则表达式的区别：

### 正则表达式分析：

`\s*(\s*[0-9]+\s*)*\s*:\s*[0-9]+\s*$`

1. `\s*`：
  - 允许行开始处有任意数量的空格，包括零个空格。
  - 这部分没问题，表示允许行前的空白字符。
2. `(\s*[0-9]+\s*)*`：
  - 这部分的含义是：**零次或多次匹配由空格包围的数字**。
  - **潜在问题**：我们这里的 `\s*` 会在每个数字的前后都允许空格，这意味着：
    - 空格可能会出现在每个数字的前后，比如：
      - `1 2 3` 和 `1 2 3`（这两种格式都能匹配）
    - 但是，如果这部分没有明确要求“数字之间应该有一个空格”，那么有些空格可能会被错误地匹配成“空格之后紧接着另一个数字”，尤其是当没有数字时（比如空行或者冒号前后有空格时）。
3. `\s*`：
  - 匹配冒号前的空格。这部分没有问题。
4. `:`：
  - 匹配冒号。没有问题。
5. `\s*[0-9]+\s*$`：
  - 匹配冒号后的目标数字，允许目标数字两侧有空格。
  - 没问题，正常匹配目标顶点编号。

### 关键问题：`(\s*[0-9]+\s*)*` 和空格的重复使用

- `(\s*[0-9]+\s*)*`：表示可以有多个数字，而每个数字的前后都允许空格。
  - 这样可能会导致空格匹配太过宽松。例如，在没有数字的情况下（如果数据中某一行只有冒号和目标顶点），这个表达式仍然会匹配行的空白部分，而导致不符合预期。
  - 如果在输入数据中存在空行或者额外的空格，正则表达式可能会错误地匹配空行或错误的顶点行。

### 第一步：理解图 $H_3$

从你提供的第二张图来看，图  $H_3$  包含了7个顶点，分别标记为1到7，并且有一些有向边。题目要求我们构造一个CNF（合取范式）布尔表达式  $\varphi_H$ ，当且仅当给变量赋予真值时，表示该图的一个核（Kernel）。

在图论中，核的定义是：

1. 核中的任意两个顶点之间没有边（即它们之间不直接相连）。
2. 图中的每个顶点，要么在核中，要么有一个邻居在核中。

### 第二步：确定变量

每个顶点  $v_i$ （其中  $i \in \{1, 2, 3, 4, 5, 6, 7\}$ ）都会对应一个布尔变量。也就是说，变量是：

- $v_1, v_2, v_3, v_4, v_5, v_6, v_7$

### 第三步：构造CNF表达式

为了表示核的条件，我们构造CNF表达式：

1. 条件1：任意两个顶点在核中不能相邻

如果有从顶点  $i$  到顶点  $j$  的有向边，那么  $v_i$  和  $v_j$  不能同时为真。用布尔公式表示为：

- $\neg v_i \vee \neg v_j$ （即“不是  $v_i$ ”或“不是  $v_j$ ”）

2. 条件2：每个顶点要么在核中，要么有邻居在核中

对于每个顶点  $v_i$ ，如果它不在核中（即  $v_i$  为假），那么它必须有一个邻居  $v_j$  在核中（即  $v_j$  为真）。对于每个有向边  $i \rightarrow j$ ，我们写出公式：

- $\neg v_i \vee v_j$

### 第四步：应用到图 $H_3$

现在我们将这些条件应用到图  $H_3$  上。

- 图  $H_3$  中的边：根据图的边，我们有：

- $1 \rightarrow 3, 2 \rightarrow 3, 3 \rightarrow 1, 3 \rightarrow 4, 4 \rightarrow 6, 5 \rightarrow 3, 5 \rightarrow 7, 6 \rightarrow 7, 7 \rightarrow 6$

对于每一条边，我们可以写出类似的CNF子句，例如对于边  $1 \rightarrow 3$ ，我们写出子句：

- $\neg v_1 \vee \neg v_3$

- 每个顶点的邻居条件：对于每个顶点，确保如果它不在核中，就有一个邻居在核中。例如，顶点3的条件可以表示为：

- $\neg v_3 \vee v_1, \neg v_3 \vee v_4$ ，以此类推。



### 第五步：构造CNF表达式

将这些子句结合起来，就得到了最终的CNF表达式  $\varphi_H$ 。请注意，题目要求使用文本表示布尔操作：

- $\neg$  使用 `~`
- $\wedge$ （与）使用 `&`
- $\vee$ （或）使用 `|`

最终你需要将这个表达式写入一个名为 `prob1.txt` 的文件中。

如果需要，我可以帮助你生成这个完整的CNF表达式，并为你提供答案。

```

# 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");
}

```

这页给的是 **Problem 2 的“半成品” awk 脚本骨架**。你的任务是在不改变总体结构的前提下，把注释里说的空缺补上（把所有 `####` 换成合适的内容、补全循环的头尾、补全 `printf` 要输出的文本等），从而把图 *G* 转换为满足 kernel 定义的 CNF 公式的文本表示。逐行解释如下（只做说明，不提供任何解题代码）：

## 文件头与 BEGIN/END

- 顶部几行以 `#` 开头的是**注释**，包含课程名、学期、作业编号、你的姓名与学号等；按要求填写。
- `BEGIN { firstMatch = 1; }`
  - `BEGIN` 是 awk 的**特殊模式**，在读入任何数据之前执行。
  - 这里把变量 `firstMatch` **置为 1**，用作一个“是否正在输出第一个子句”的标志位（后面需要据此决定是否在子句前输出连接符）。
- `END { printf("\n"); }`
  - `END` 是在**所有输入处理完**之后执行；这里做的是**换行收尾**。

## 顶点行的模式与动作块（你要填的主体）

- `/ Write your vertex-line pattern here, and also in prob2.txt / { ... }`
  - 这行的意思是：把“**匹配顶点行**”的 awk 模式写在这里（与 `prob2.txt` 里写的**完全一致**）。一旦匹配到顶点行，就执行大括号里的动作。
  - 大括号里的注释提示了你要做什么：
    - 这段 **action** 跨越多行；
    - 你需要在指定位置**补齐 for 循环并完善若干** `printf`，把 **CNF 公式** 的各个部分按文本形式拼出来；
    - 每一个 `####` **都要被替换**成具体内容（不一定等长）。
- 关键逻辑（一）**控制子句之间的连接符**

awk

复制 编辑

```
if (firstMatch == 1) { firstMatch = 0 } else { printf(####); }
```

- 这段用上面的 `firstMatch`：
  - **第一次**输出子句时不打印连接符，只把标志改为 0；
  - 之后每遇到一个新子句，就先**打印 CNF 的“合取”连接符**（例如 “&” 或 “^” 等——具体文本你要在 `####` 处填），再输出子句主体。
- 注释“**Think: why...**”就是在提醒你：这样做是为了**避免在第一个子句前多打一个连接符**。

- 关键逻辑（二）按是否有入边分支

```
awk
```

复制 编辑

```
if (NF == 2) {
    printf(####);
} else {
    ...
}
```

- `NF` 是 `awk` 的**字段个数**。顶点行形如 `j1 j2 ... jd : i`，冒号两侧必须有空格。
  - 若**没有入邻居**（没有 `j`），顶点行就变成 `: i`，这时**只有两个字段**（冒号和 `i`），因此 `NF == 2`。
  - 这种情况下，注释问你：“**顶点没有入边时应该打印什么？**”——直观上，这个子句应当强制让该顶点自身满足 `kernel` 的**覆盖条件**（如何写入 `CNF` 的文本，由你在 `####` 里补齐）。
- **否则（有一个或多个入邻居）：**
  - **第一段循环**（注释：*ensures the first condition of kernels is met*）

```
awk
```

复制 编辑

```
printf(####);
for (####) {
    printf(####);
}
printf(####);
```

- 这段用当前顶点 `i` 以及它的所有**入邻居** `j1..jd` 拼出一个子句，对应 `kernel` 的**条件 (i)**（每个顶点要么在集合里，要么被某个在集合里的入邻居指到）。
- 你需要在三个 `printf(####)` 里补上**子句的开头/中间拼接/结尾**（比如括号、析取符号等），并在 `for (####)` 里补上**循环范围**（遍历所有入邻居）。
- **第二段循环**（注释：*ensures the second condition of kernels is met*）

```
awk
```

复制 编辑

```
for (####) {
    printf(####);
}
```

- 这段为**每一条入弧**  $(j, i)$  输出一个**额外子句**，对应 `kernel` 的**条件 (ii)**（集合内部**不允许存在**从集合中某点指向集合中某点的弧）。
- 同样，你需要填好**循环范围**与**每个子句的文本**（一般是一类“不可同时选中”的约束）。



## 你需要决定/填充的典型位置

- **模式**：能够唯一匹配“顶点行”（带空格、带冒号），并与 `prob2.txt` 中的模式相同。
- **连接符**：子句之间的合取连接符文本（例如 `&` 或其他按题面指定的表示）。
- **子句结构**：括号、变量名（题前几页已规定变量与 CNF 的**文本表示法**）、析取符号等。
- **循环头**：如何从当前行中**遍历所有入邻居**字段（注意 `awk` 的字段编号以及 `:` 所在列）。
- **无入边的特例**：`NF==2` 时应输出的**强制性子句**。

小结：这页不是要你写出完整答案，而是告诉你**在哪些位置**要填什么类型的内容，以及**每段代码对应 kernel 两条定义中的哪一条**。照着这些“锚点”把 CNF 的每个子句正确打印出来即可。

![image-20250814184557928](readme.assets/image-20250814184557928.png)



### 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<sup>1</sup>. 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  $a = \text{False}$  and  $b = \text{True}$ . We first translate the expression into text in the way described earlier (p4), replacing  $\neg, \wedge, \vee$  by `~, &, |` respectively. This gives the text string

$$(a \mid b) \ \& \ (\sim a \mid \sim b) \ \& \ (\sim a \mid b).$$

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

---

<sup>1</sup>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.

已思考 1m 20s >

下面这页主要做两件事：给出 **Problem 3 (3分)** 的要求，并教你如何用 **SageMath** 来检测你输出的 CNF 公式是否“可满足” (satisfiable)。我按版面从上到下逐段解释——基本都是把原文意思换成中文并补几句注释，方便你照做。

## Problem 3 要求 (框内)

(a) 设图有  $n$  个顶点。请你用  $n$  表达一个**上界**：你的 awk 程序为了把  $\varphi_G$  (那个描述“图  $G$  有 kernel”的 CNF 公式) 完整打印出来，**最多需要调用几次** `printf`。

(b) 把 (a) 的上界改写成 **大 O 记号** (big-O)。

把答案写成一个 **PDF**，文件名 `prob3.pdf`。

注：这里数的是“调用 `printf` 的次数”，不是字符数量。通常思路会用到“每条边/每个顶点对应若干子句，因此最坏情况下边数  $m \leq n(n-1)$ ”这类图论上界；最后把得到的上界用  $O(\cdot)$  表达即可。

## 下面进入：让 awk 输出能被“可满足性测试程序”读入

你要稍微修改 awk 脚本，让它的输出**正好**是一个一行的 CNF 布尔表达式文本，这样就能丢给一个程序去判断是否可满足。

### 用 SageMath 做可满足性测试

- **SageMath** 是一个基于 Python 的开源数学软件，课堂/平台里已经装好了。你**不需要系统学习** Sage，只要按说明用到一个函数就行。想了解更多可去它的官网 (原文给了链接)。
- 我们只用 Sage 的一行命令来判断：**给定的 CNF 是否可满足** (有没有一组对变量的真值赋值能让整个表达式为 True)。

### 例子 (原文示范)

有表达式：

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

它是**可满足的**，比如取  $a = \text{False}$ ,  $b = \text{True}$  就能使其为真。

把它换成题面要求的**纯文本写法** (逻辑符号替换规则： `¬→~` , `∧→&` , `∨→|` )：

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

复制 编辑

然后在 Linux 命令行里直接问 Sage：

```
bash
```

复制 编辑

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

- `sage -c '...'`： `-c` 表示**直接执行**引号里的那段 Sage 代码后退出。
- `propcalc.formula("...")`：把字符串解析成命题逻辑公式对象。
- `.is_satisfiable()`：判断这个公式是否可满足。
- `print(...)`：把结果打印到标准输出。

运行后会输出：

```
graphql
```

复制 编辑

```
True
```

表示**可满足**。

你也可以进入 Sage 的交互环境再敲那行：

```
bash
```

复制 编辑

```
sage
```

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

```
True
```

## 这个测试器做了什么

这里用到的 `is_satisfiable()` 是 Sage 里最原始的可满足性测试器：它实际上是**枚举真值表**。

如果一个表达式含  $n$  个变量，它会检查所有  $2^n$  种赋值，看看有没有一行能让表达式为 True。

启示：随变量数增长很快，复杂度是指数级的（这是解释“为什么不要指望暴力法很快”的背景）。

## 把这页内容用于你的作业的要点

### 1. Problem 3:

- (a) 给出“打印  $\varphi_G$  需要多少次 `printf`”的**最坏情况上界**（只用  $n$  表示即可，必要时用到  $m \leq n(n-1)$  这类基本上界）；
- (b) 把上界写成  $O(\cdot)$  形式；
- 交 `prob3.pdf`。

### 2. Sage 测试:

- 让 `awk` 输出**单行**公式文本，符号用 `~ & |`；
- 把这行放进 `propcalc.formula("...")` 里，调用 `.is_satisfiable()`，即可验证你的  $\varphi_G$  是否与“存在 kernel”一致。

如果你想，我可以把你 `awk` 程序生成的一行 CNF 先过一遍 Sage，演示命令该怎么拼接；或者一起梳理 (a) 该怎么数 `printf` 的最坏次数（不直接给最终数值，但把拆分思路讲清楚）。

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:

$\text{text representation of } \varphi_G$   
`sage -c 'print(propcalc.formula(".....").is_satisfiable())'`  
 $\text{SageMath command}$

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.

这是在讲：把你在**Problem 2**里写的 `awk` 脚本再改一下，让它**不再只输出公式  $\varphi_G$** ，而是**输出一整条可以直接丢给 SageMath 执行的命令**，去判断这个公式是否“可满足”（satisfiable）。

### 原文要求在做什么？

- 目标输出是一行 Linux 命令，形式精确如下（中间的省略号换成你生成的公式文本）：

```
rust                                                                    复制 编辑

sage -c 'print(propcalc.formula(".....").is_satisfiable())'
```

- 也就是说：
  - 前缀固定是： `sage -c 'print(propcalc.formula("`
  - 中间夹入  **$\varphi_G$  的文本表示**（就是你 Problem 2 生成的那个命题公式字符串）
  - 后缀固定是： `").is_satisfiable())'`
- 这行命令的含义：调用 SageMath（`sage -c`）执行一条 Python 语句：
  - `propcalc.formula("...")` 把你的公式字符串解析成命题逻辑公式；
  - `.is_satisfiable()` 判断这个公式是否可满足；
  - `print(...)` 打印结果（True/False）。

### 你需要改动 `awk` 的点

在 Problem 2 中，你的 `awk` 最终是**打印  $\varphi_G$  本身**。现在改成**在外面包上前后两段固定字符串**。

示例（核心思路）：

```
awk                                                                    复制 编辑

# 假设变量 f 就是你在 Problem 2 算出来的  $\varphi_G$  的字符串
# 以前: print f
# 现在:
printf "sage -c 'print(propcalc.formula(\"%s\").is_satisfiable())'\n", f
```

要点：

- `awk` 的字符串用**双引号**包，所以为了在输出里保留 **传给** `propcalc.formula` **的双引号**，需要写成 `\ "%s\"`。
- 外层的 **单引号**（把整段 Python 代码包起来）在 `awk` 里直接写即可，不用转义。

## 交付物与测试

- 把改好的脚本存成 `prob4.awk`。
- 用几张不同的有向图输入测试：
  1. 运行 `awk -f prob4.awk input.txt`（或按作业要求的方式喂入图）；
  2. 它会**打印出一条** `sage -c 'print(...)'` **命令**；
  3. 把这条命令在终端执行（或直接 `$(awk -f prob4.awk input.txt)` 进行命令替换执行），Sage 会输出 `True` 或 `False`。
- 原文还强调：要核对“ $\varphi_G$  可满足  $\Leftrightarrow$  图  $G$  有 kernel”是否成立（你应当用若干图验证这一对应关系）。

## 小结（按原文的精炼复述）

- 不改变你生成  $\varphi_G$  的逻辑，只是**在输出时**加上固定前后缀，形成 Sage 可执行的一行命令；
- 这行命令会调用 `propcalc` 解析你的公式，并用 `.is_satisfiable()` 判断可满足性；
- 文件名：`prob4.awk`；
- 用多组图测试，确认“可满足性”与“ $G$  是否有 kernel”——对应。





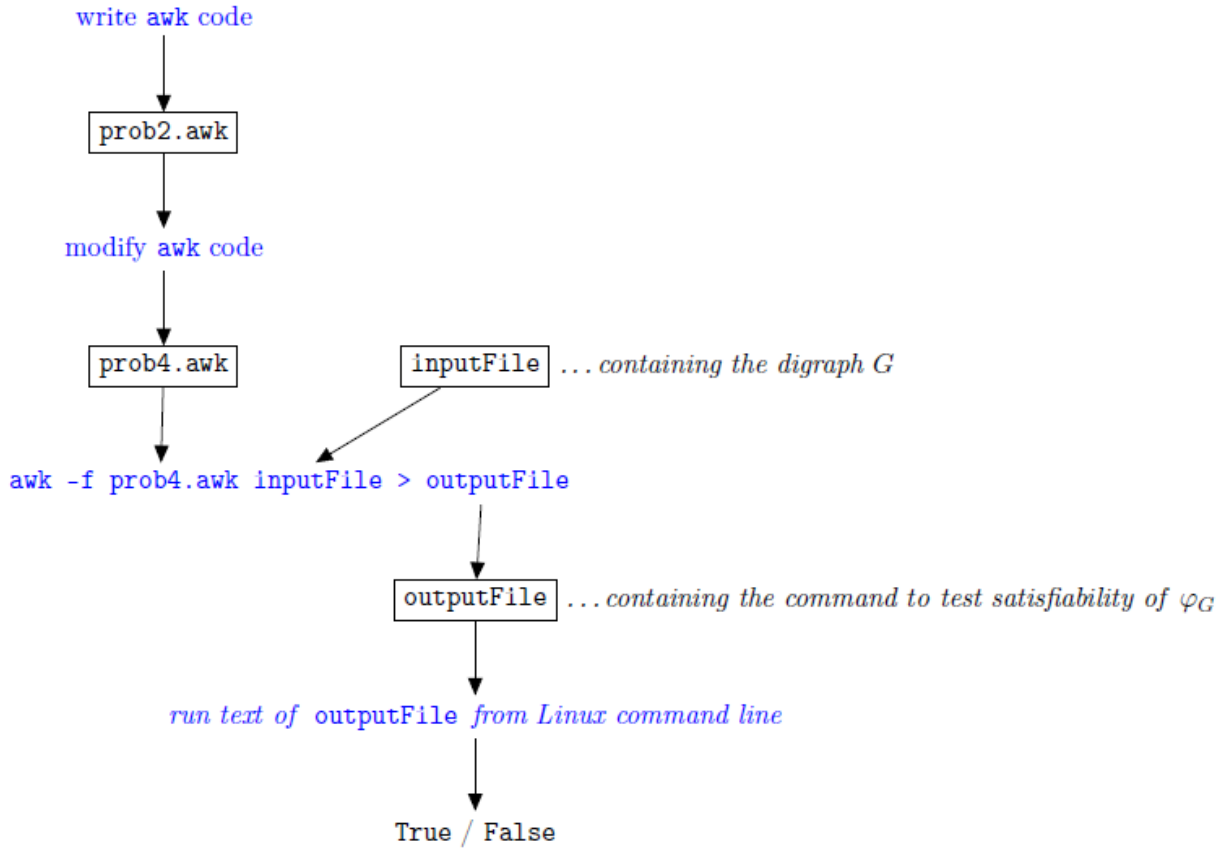


Figure 2: The plan for Problem 4.

## 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:



已思考 1m 1s >

这页分两部分：上面是Problem 4 的流程图；下面是Problem 5 的概念引入。

## Problem 4 的计划 (Figure 2)

- 先完成 prob2.awk (上一题里把图  $G$  转成 CNF  $\varphi_G$  的文本)。
- 在此基础上修改得到 prob4.awk。
- 准备一个输入文件 inputFile (里面按指定格式写着有向图  $G$ )。
- 运行：

```
awk -f prob4.awk inputFile > outputFile
```

这会生成 outputFile，其内容是“一条命令行”，用于测试  $\varphi_G$  是否可满足 (文字说明里就写着 *containing the command to test satisfiability of  $\varphi_G$* )。

- 然后把 outputFile 里的文本拿到 Linux 命令行执行 (run text of outputFile from Linux command line)，得到 True/False 的结果。

要点：Problem 4 不是直接打印公式本身，而是让 awk 输出一条“可执行的检测命令”，再到命令行运行它来得到真假。

## Problem 5 引言 (Introduction to Problem 5)

- 定义：一个处于 CNF (合取范式) 的布尔表达式  $\varphi$ ，如果每个子句都包含某个变量，并且这个变量在整个  $\varphi$  里只以一种形式出现 (要么总是正文字  $x$ ，要么总是否定  $\neg x$ )，那么称  $\varphi$  是 **frumious**。  
换句话说：对每个子句，都能在其中找到一个“只出现单一极性”的变量；不同子句可以选不同变量。
- 给出的 **frumious** 示例：
  1.  $(\neg x \vee \neg y) \wedge (\neg x \vee y)$ 
    - 第一个子句含有变量  $x$  (以  $\neg x$  形式)；
    - 第二个子句也含有  $x$  (同样只以  $\neg x$  形式)。
    - 因而每个子句都有只以一种形式出现的变量  $x$ 。
  2.  $(x \vee z) \wedge (\neg y \vee \neg z) \wedge (\neg w \vee x \vee \neg y)$ 
    - 第一个子句可选  $x$  (它在全式里只正向出现)。
    - 第二个子句可选  $y$  (它在全式里只负向出现)。
    - 第三个子句里  $w$ 、 $x$ 、 $y$  任取其一都满足“只一种形式出现”的要求。
    - 因而该式也是 **frumious**。
- 文末提示：“下面这些表达式**不是** frumious” (对应的反例在下一页继续给出)。

上述内容完全来自原文：这一页只是在说明 Problem 4 的产出与执行方式，以及 Problem 5 中“frumious”这一性质的精确定义与例子。

$(\neg x) \wedge (x \vee y)$       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

这页在说两件事：

1. 用几个例子说明“frumious 的 CNF”到底要求什么；
2. 给出 **Problem 5**：用对子句数  $m$  的数学归纳法证明“frumious 的 CNF 一定可满足”。

先把关键术语理顺：

## 什么是 CNF、正负出现、frumious

- **CNF (合取范式)**：顶层是“与” ( $\wedge$ )，每个子式 (叫子句) 是若干“或” ( $\vee$ ) 起来的文字 (变量或它的否定)。  
例如  $(x \vee y \vee \neg z) \wedge (\neg x \vee z)$  是 CNF；而  $a \vee (b \wedge (c \vee d))$  **不是** CNF (见本页第三个例子)。
- **正/负出现**：变量  $x$  以 **正** 形式出现写作  $x$ ，以 **负** 形式出现写作  $\neg x$ 。
- **frumious (按本页给出的例子含义)**：一个 CNF 若 **对每一个子句**，都能在该子句中找到**某个变量**，使得这个变量在**整条公式里**只以**一种形式**出现 (要么始终正，要么始终负)，则称该 CNF 是 *frumious*。  
换句话说：**每个子句里都必须含有一个“纯文字 (pure literal)”**。  
注意：不是“全式中存在一个纯文字”就够了，而是**每个子句**都得有。

## 逐条解读本页的三个例子

1.  $(\neg x) \wedge (x \vee y)$ 
  - 第一子句只有变量  $x$ ，但  $x$  在全式中既以负形 (第一子句的  $\neg x$ ) 又以正形 (第二子句的  $x$ ) 出现，**不纯**，所以这个子句**不满足**“每句含纯文字”的要求。
  - 即便第二子句里  $y$  只以正形出现 (是纯的)，也**无济于事**，因为**第一子句**没有纯文字。  
 $\Rightarrow$  整个式子**不是** frumious。
2.  $(w \vee y \vee \neg z) \wedge (x \vee y \vee z) \wedge (\neg w \vee x \vee \neg y \vee z)$ 
  - 变量  $w, y, z$  在全式里都既有正也有负的出现 (如  $w$  与  $\neg w$ ， $y$  与  $\neg y$ ， $z$  与  $\neg z$ )，因此**第一子句** (只含  $w, y, \neg z$ ) 里**没有**只以单一形式出现的变量。
  - 虽然  $x$  在全式里**只以正形**出现 (是纯的)，但它**不在第一子句里**，所以仍然不满足“每个子句都要有纯文字”。  
 $\Rightarrow$  这个式子也**不是** frumious。
3.  $a \vee (b \wedge (c \vee d))$ 
  - 顶层是“或”连接了一个变量和一个“与”式，这**不是** CNF 的结构。  
 $\Rightarrow$  谈不上 frumious 与否。

## 记号说明

若  $\varphi$  是一个布尔表达式：

- 用  $m$  表示它的子句数；
- 用  $n$  表示它的变量数。

## Problem 5 (6 分) 到底要做什么

**命题：**任何 *frumious* 的 CNF (有  $m$  个子句) 都是可满足的。

**要求：**对  $m$  做数学归纳证明，并把证明写成 PDF，文件名 `prob5.pdf`。

### 证明思路提示 (按“归纳于 $m$ ”来写)

- **基例  $m = 1$ ：**只有一个子句。*frumious* 要求这个子句里有一个变量在全式中只以一种形式出现 (纯文字)。把这个纯文字赋为 **真** 就能满足该子句，因此整个式子可满足。
- **归纳步：**设任意含  $k < m$  个子句的 *frumious* CNF 都可满足。

对含  $m$  个子句的 *frumious* CNF：

1. 取任意一个子句，其中按定义**存在**一个纯文字  $\ell$ ；
2. 令  $\ell = \text{True}$ 。含  $\ell$  的子句都被满足并可从公式里**删除**；由于  $\ell$  是**全式纯**的， $\neg \ell$  **不会出现在任何子句里**，因此不会使别的子句变假；
3. 剩下的公式仍是 CNF，子句数 **减少** ( $< m$ )，且对保留下来的每个子句，它原来拥有的纯变量仍然纯，因此**仍然是 frumious**；
4. 由归纳假设，剩余公式可满足；与  $\ell = \text{True}$  合并得到原公式的一个满足赋值。  
 $\Rightarrow$  命题得证。

如果你需要，我可以帮你把这份思路整理成你课程风格的完整证明稿 (英语/中文均可)，或者先检查你写好的 `prob5.pdf`。

