

LAB1: LLM + SAT Solver



1 环境配置

0. 请在本课程提供的虚拟机镜像（或WSL）中完成此实验以及后续实验，请先阅读 [env.pdf](#)。
1. 如果你使用的是VMware虚拟机，请在虚拟机内从canvas上下载lab1.zip放至桌面并解压，然后使用VScode进入lab1文件夹。如果你使用的是WSL，请直接从canvas上下载lab1.zip，然后把他解压并复制到Ubuntu22.04系统的home目录下，使用VScode连接到WSL进行开发。
2. 在 lab1 文件夹中打开命令行，并在命令行中输入以下命令来安装所有依赖，此脚本会帮你安装实验1所需要的所有依赖以及工具。

```
sudo bash ./install_minisat.sh
```

3. 在命令行中使用以下命令来验证你的依赖是正常的（SAT Solver正常工作，大模型正常访问）

```
make example
```

你将看到

```
Testing minisat...
SAT
Model found:
A := 0
B := 0
C := 0
Testing llm...
Model output: Hello! How can I help you today?
Environment check complete.
```

2 MiniSAT

通过本实验我们希望学生能实际使用SAT求解器解决一些简单的问题，我们使用的是MiniSAT，它是一个开源的 SAT 求解器，你可以在网络上找到它的源代码。（但这并不重要，我们只希望你能通过SAT求解器来解决问题，若你对其内部实现非常感兴趣可以尝试去阅读其源代码）

我们将为你简单地介绍MiniSAT 的使用方法，在此之前，请你仔细阅读 `minisat.cpp` 的内容，这个实例程序求解了命题 $P = (\neg A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (A \vee \neg B \vee C)$ 通过注释简单理解每一行代码，这将有助于你理解MiniSAT 的使用方法。

首先，MiniSAT 使用合取范式（CNF）作为其输入，要使用MiniSAT 求解命题逻辑表达式，首先我们需要初始化一个Solver对象，即 `minisat.cpp` 中的

```
// Create a solver
Solver solver;
```

然后我们需要定义一些命题变量，这里我们使用MiniSAT 提供的 `newVar` 函数, 比如在 `minisat.cpp` 中, 我们定义了文字A, B, C:

```
// Create variables
auto A = solver.newVar();
auto B = solver.newVar();
auto C = solver.newVar();
```

接下来我们需要向solver中添加子句，这里我们使用MiniSAT 提供的 `mkLit` 和 `addClause` 函数, `mkLit` 可以将命题变量转化成子句需要的格式，`addClause` 可以向solver中添加子句。比如在 `minisat.cpp` 中, 你可以看到我们添加了三次子句：

```
// Add the clauses
// (~A v ~B v C)
solver.addClause(~mkLit(A), ~mkLit(B), mkLit(C));
// (~A v ~B v ~C)
solver.addClause(~mkLit(A), ~mkLit(B), ~mkLit(C));
// (A v ~B v C)
solver.addClause(mkLit(A), ~mkLit(B), mkLit(C));
```

接下来我们正式调用 `solve` 函数进行求解，函数的返回值是一个bool类型，标志这此命题是否存在一组解释使其为真：

```
// Solve the problem
auto sat = solver.solve();
```

如果返回值为真，我们可以通过 `modelValue` 函数获取一种解释，若返回值为假，我们直接输出“UNSAT”：

```
// Check solution and retrieve model if found
if (sat)
{
    std::clog << "SAT\n"
               << "Model found:\n";
    std::clog << "A := " << (solver.modelValue(A) == l_True) << '\n';
    std::clog << "B := " << (solver.modelValue(B) == l_True) << '\n';
}
```

```

        std::clog << "C := " << (solver.modelValue(C) == l_True) << '\n';
        return 0;
    }
    else
    {
        std::clog << "UNSAT\n";
        return 1;
    }
}

```

(提示, 在你要完成的实验中, 你不能有任何使用 `printf`, `cout` 或者 `clog` 的输出, 你需要做的是正确实现需要的函数, 这样我们才能够对你的程序进行测试。违规的同学将按零分处理。)

最后, 你可以在lab1目录下打开命令行, `make minisat` 来运行这个例子并看到输出结果如下:

```

SAT
Model found:
A := 0
B := 0
C := 0

```

3 LLM

SAT Solver可以准确而可靠地解决命题逻辑的可满足性问题。然而, 命题逻辑公式本身是抽象而无意义的, 为了解决现实生活中的问题, 我们需要人工定义命题变量, 并将问题表述成命题逻辑公式。这使得SAT Solver的应用依旧依赖人对实际问题进行建模。

大语言模型 (LLM) 的出现赋予了计算机强大的自然语言处理能力。在本实验中, 我们希望同学使用大模型将自然语言问题转换为计算机可以处理的命题逻辑公式, 再交给SAT Solver解决。以此解放问题建模这一繁重的人力工作。

我们将为你简单地介绍C++中调用LLM的方法, 在此之前, 请你仔细阅读 `test_llm`, `llm.cpp` 的内容。`llm.cpp` 是我们为你准备的对openai提供的大模型接口的封装。你可以创建 `LLM` 对象并直接调用 `generate` 方法实现与大模型的问答。例如 `test_llm.cpp` 中实现了与大模型的简单问候

```

LLM llm("deepseek-ai/DeepSeek-V3.1");
const std::string prompt = "请用几十字简要介绍离散数学。";
const std::string output = llm.generate(prompt);
std::cout << "Model output: " << output << std::endl;

```

我们的实验使用一个第三方的大模型服务平台, 详细注册流程见 `env.pdf`。记得修改 `llm.cpp` 第42行中的API KEY, 并在创建LLM实例时选择自己喜欢的模型, 例如 `deepseek-ai/DeepSeek-V3.1`

4 实验内容

4.1 问题描述

我们在 `questions` 目录中准备了三个问题。本实验的目的是使用大模型+SAT Solver解决他们。基本流程如下:

1. 将问题加上合适的提示词发送给大模型, 要求其生成合取范式 (CNF)
2. 解析大模型的回复, 提取并检查其中的合取范式
3. 交给MiniSAT求解该式的可满足性
4. 直接使用大模型求解可满足性

程序的基本流程在 `main.cpp` 和 `lab1.cpp` 中。`main.cpp` 中的主函数遍历和读取 `questions` 目录中的所有问题，并交给 `lab1` 函数处理。`lab1` 函数实现在 `lab1.cpp` 中，依次进行 `llm_get_cnf`，`parse_cnf`，`solve_cnf`，`llm_solve_cnf` 四个函数，对应上述基本流程的四步。你的任务是完成这四个函数。

4.2 你的任务

任务一：补全 `llm_get_cnf` 函数

由于MiniSAT只支持合取范式（CNF），`llm_get_cnf` 函数的目标就是借助大模型把原问题转换为CNF。你需要完成 `llm_get_cnf` 函数中对提示词的构建。

要求：提示词应该包含待解决的问题（`problem`），和你对任务的描述。提示词对后续大模型的输出是否可靠且易处理影响巨大，因此提示词的设计至关重要，在本实验中，你必须要求大模型把关键的结果用“@@”包裹起来，以方便后续提取。

成功设置提示词后，大模型的回答可能如下面所示：

```
--- LLM Generated Answer ---
我们需要将三个规则转化为命题逻辑公式，并写出对应的**合取范式（CNF）**，然后回答它们能否同时满足。

## 第一步：定义变量
- \(\neg W\)：小王来
- \(\neg L\)：小李来
....(篇幅原因省略部分大模型输出)....
**最终 CNF 公式**：

@@(\neg W \vee L) \wedge (W \vee Z) \wedge (\neg Z \vee L)@@
```

任务二：实现 `parse_cnf` 函数

`parse_cnf` 函数的目标是从大模型输出的字符串（可能非常杂乱）中提取并检查CNF，将他转换成 `CnfFormula`。`CnfFormula` 表示一个合取范式，是我们自己定义的一个类，在 `common.h` 中可以看到它的结构。它的本质是一个子句列表，表示这些子句进行合取。每个子句是一个文字列表，表示这些文字进行析取。每个文字是一个命题变量，可能取反。

在 `llm_get_cnf` 函数中，你的提示词包含了对输出结果的约束，你要求大模型把关键的结果用“@@”包裹起来，那么函数开头这段代码可以先帮你找到包含CNF的字符串。

```
formula_str = formula_str.substr(formula_str.find("@@") + 2,
                                formula_str.rfind("@@") -
                                (formula_str.find("@@") + 2));
```

随后，由于大模型一般会选择markdown格式输出公式（当然也有可能是普通文本），你的程序应该针对这种格式进行CNF的识别和构建。例如，以下是大模型可能的一个markdown输出。

```
$$(\neg W \vee L) \wedge (W \vee Z) \wedge (L \vee \neg Z)$$
```

，其中的 `\xxx` 就表示 $\vee$ 析取词、 $\wedge$ 合取词等，实际对应的公式是：

$$(\neg W \vee L) \wedge (W \vee Z) \wedge (L \vee \neg Z)$$

注意事项:

1. 大模型给出的式子一般都是正确的合式公式，因此，在此函数中你只需要考虑它是不是一个合法的合取范式，而不需要考虑它是不是一个合式公式。
2. 大模型可能会使用 `\neg` 来代替 `\lnot`，用 `\vee` 来代替 `\lor`，用 `\wedge` 来代替 `\land`；也有可能大模型会直接输出“`∨`”“`^`”“`¬`”等字符。这些都是你必须要考虑的。
3. 在解析过程中一旦判定输入的不是合法的CNF，你必须使用以下语句抛出一个异常，该异常会被 `tab1` 函数中的 `try`，`catch` 语句接收，打印错误信息并跳过此个问题。

```
throw std::runtime_error("<此处随意设置你想要的错误信息>");
```

任务三：补全 `solve_cnf` 函数

`solve_cnf` 函数使用MiniSAT求解 `CnfFormula`。你需要仿照 `minisat.cpp`，把 `CnfFormula` 内的子句和命题变量添加到MiniSAT的 `solver` 中。

注意事项:

1. `CnfFormula` 中同名的文字只需要创建一个MiniSAT中的命题变量，即调用 `solver.newVar()` 一次，否则可能会导致求解结果错误。我们为你创建好了 `var_map` 用来储存文字名字与MiniSAT中 `Var` 对象的对应关系，你可以按下面的方式来管理它

```
var_map[文字名字] = solver.newVar() //创建新的变量保存到map中
var var = var_map[文字名字] //获取旧的名字
if (var_map.find(文字名字) == var_map.end()); //判断一个名字是否已经在map中
```

2. 在 `minisat.cpp` 中，我们使用 `solver.addClause(~mkLit(A), ~mkLit(B), mkLit(C));` 来增加一个带有三个文字的子式。而现在我们需要增加的子式可能有不确定的文字个数，因此我们需要先创建一个MiniSAT自带的列表容器来添加任意长的子式

```
Minisat::vec<Lit> minisat_clause;
...(多次使用minisat_clause.push(lit);向列表中增加Lit文字)...
solver.addClause(minisat_clause);
```

任务四：补全 `llm_solve_cnf` 函数

`llm_solve_cnf` 函数使用大模型代替SAT Solver求解CNF。传入的是 `CnfFormula` 的字符串形式。请设计提示词让大模型完成这个任务。你必须在提示词中明确要求大模型的输出只包含“SAT”或“UNSAT”来表示可满足或不可满足，这样这个函数剩余部分才能正确返回布尔类型表示可满足性。

注意：你可以随意修改`//Your code here`和`//Your code end here`.包裹的代码，其余代码不允许任何修改，`//Your code here`和`//Your code end here`这两行注释不允许删除。你可以增加一些输出来方便调试，但是最终上传版本中请不能包含任何额外的输出语句。

提示：由于大模型输出不稳定，你可以在调试时手动修改 `lab1` 函数，跳过大模型回答的步骤，用一份已经准备好的大模型输出（比如前几次的输出）来继续实验。但在最终提交时，请不要这么做。

4.3 辅助函数

`common.cpp` 中准备了几个工具函数，注释中有对他们的介绍，不强制使用。

4.4 最终效果

使用 `make lab1` 可以一键编译和运行。下面是一个可能的输出：

```
Solving Question: questions/q1.txt
--- Question ---
假设你是一家特别受欢迎的餐厅的老板，为了控制用餐人数，你设定了以下几个规则：
规则 1：小王必须来。
规则 2：如果小王来了，那么小李必须来。
规则 3：如果小李来了，那么小张必须来。
规则 4：如果小张来了，那么小王就不能来。
问题： 餐厅的用餐规则能否同时被满足？
--- LLM Generated Answer ---
我们先定义三个命题变量：
....(篇幅原因省略此处LLM的回答)...
因此，答案是：
\\@(W) \\land (\\neg W \\vee L) \\land (\\neg L \\vee Z) \\land (\\neg Z \\vee \\neg W)@@
--- Parsed CNF ---
(W) \\wedge (\\neg W \\vee L) \\wedge (\\neg L \\vee Z) \\wedge (\\neg Z \\vee \\neg W)
--- MINISAT Result ---
UNSAT
--- LLM SAT Result ---
UNSAT
```

由于大模型的输出结果不稳定，但你只需要处理文档中提到过的情况，在评分时其它情况不作要求。

4.5 提交

运行 `make handin` 可以得到 `lab1.cpp` 的压缩包 `lab1.zip`，把它上传到 canvas 即可。注意，压缩包中只能有 `lab1.cpp` 一个文件。