第一章作业

1. $\frac{\partial(f\cdot g)}{\partial x} = \left[f\cdot\frac{\partial g}{\partial x}\right]\oplus\left[g\cdot\frac{\partial f}{\partial x}\right]\oplus\left[\frac{\partial f}{\partial x}\cdot\frac{\partial g}{\partial x}\right]$

   证明：首先 $\frac{\partial f}{\partial x} = f_{x=x}\oplus f_{x=\bar{x}} = f\oplus f_{\bar{x}}$

   可以推出 $\frac{\partial f}{\partial x}\oplus f = f\oplus f_{\bar{x}}\oplus f = 0\oplus f_{\bar{x}} = f_{\bar{x}}$

   $LHS = (f\cdot g)\oplus(f\cdot g)_{\bar{x}} = (f\cdot g)\oplus(f_{\bar{x}}\cdot g_{\bar{x}}) = (f\cdot g)\oplus\left[(\frac{\partial f}{\partial x}\oplus f)\cdot(\frac{\partial g}{\partial x}\oplus g)\right] = (f\cdot g)\oplus\left[(\frac{\partial f}{\partial x}\cdot\frac{\partial g}{\partial x})\oplus(\frac{\partial f}{\partial x}\cdot g)\oplus(f\cdot\frac{\partial g}{\partial x})\oplus(f\cdot g)\right]$

   不难看到存在 $(f\cdot g)\oplus(f\cdot g)$ 的项，可以直接化简 $LHS = (\frac{\partial f}{\partial x}\cdot\frac{\partial g}{\partial x})\oplus(\frac{\partial f}{\partial x}\cdot g)\oplus(f\cdot\frac{\partial g}{\partial x})$

   对比左右，不难看出这就是 RHS ，故证毕。

2. (a) 首先 NAND2 输出接一个 NOT 就是 AND ；根据 NAND2 等价非或，两个输入端各接 NOT 就是 OR；既然有了 AND，OR，NOT 那么所有布尔表达式最终都可以表示。

   (b) $f = ad + bcd + ac = \overline{a\dagger d} + \overline{\overline{b\dagger c}\dagger d} + \overline{a\dagger c} = \overline{\overline{\overline{a\dagger d}\dagger\overline{\overline{b\dagger c}\dagger d}}\dagger\overline{a\dagger c}} = \overline{(a\dagger d)\dagger(\overline{b\dagger c}\dagger d)}\dagger(a\dagger c)$

   (c) 如图1 所示，使用了一个NOT，一个AND，一个OR，一个AOI22，一共4个 tree nodes。

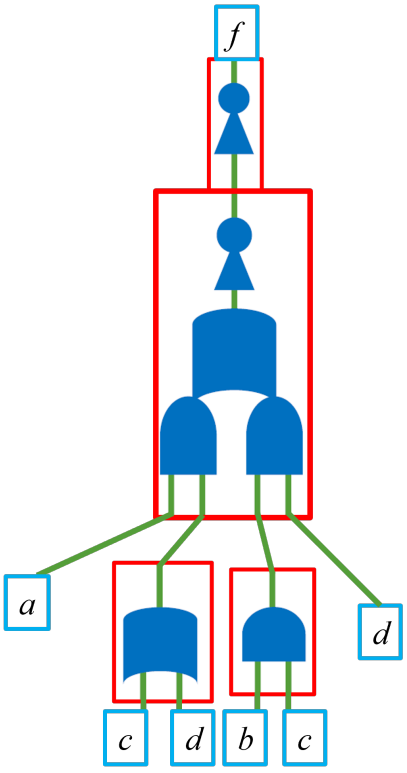

图 1: 2.(c) 电路实现图

3. **DPLL SAT Solver Report**

朴素DPLL算法概述

戴维斯-普特南-洛吉曼-洛夫兰（DPLL）算法是一种用于解决布尔可满足性问题（SAT）的完整的回溯搜索算法。它通过系统地为变量赋值来检验是否存在一组赋值能够满足给定的合

取范式（CNF）公式。相比于简单的暴力搜索，DPLL算法通过以下几个关键步骤来有效地裁剪搜索空间：

- **单元传播（Unit Propagation）：** 这是DPLL算法的核心优化手段。如果一个子句中只包含一个未赋值的文字（称为单元子句），那么为了满足这个子句，该文字必须被赋值为真。这个赋值可能会连锁地产生新的单元子句，从而进行一系列的确定性赋值，极大地减少了搜索的分支。
- **纯文字消除（Pure Literal Elimination）：** 如果一个变量在整个公式中只以一种极性出现（即，只出现 x 或者只出现 $\neg x$），那么这个文字被称为纯文字。我们可以安全地将该文字赋值为真，因为这不会导致任何子句变为假。这样，所有包含该纯文字的子句都可以被满足并移除，从而简化问题。
- **分裂规则（Splitting Rule）：** 如果无法再进行单元传播或纯文字消除，算法会选择一个未赋值的变量，并尝试将其赋值为真。然后递归地调用DPLL算法求解简化的公式。如果递归调用返回“可满足”，则原问题也可满足。否则，算法会回溯，并将该变量赋值为假，再次进行递归求解。

算法的终止条件有两个：如果所有子句都被满足（即公式为空），则问题是可满足的（SAT）。如果在赋值过程中出现了一个空子句（即子句中所有文字都被赋值为假），则说明当前的赋值路径是错误的，需要进行回溯。如果所有可能的赋值路径都导致了冲突，那么该公式是不可满足的（UNSAT）。

代码实现说明

本次作业中，我使用C++实现了一个基础的DPLL SAT求解器。代码主要由几个关键部分组成：DIMACS格式解析、公式化简以及核心的DPLL递归函数。

- **DIMACS格式解析 (parse_dimacs 函数):** 该函数负责读取以DIMACS CNF格式表示的SAT问题实例文件。它会解析文件头部的 `p cnf` 行来获取变量数量和子句数量，并逐行读取子句信息，将其存储在一个二维向量 `vector<vector<int>>` 的数据结构中，其中每个内部向量代表一个子句，整数代表文字（正数表示原变量，负数表示其否定）。
- **公式化简 (simplify 函数):** 这个辅助函数是实现DPLL算法中赋值操作的核心。给定一个公式和一个文字（例如 x 或 $\neg x$），它会根据该赋值对公式进行化简。具体操作包括：

  (a) 移除所有包含该文字的子句（因为这些子句已经被满足）。

  (b) 从其余子句中删除该文字的否定形式（例如，如果赋值为 x，则从子句中删除 $\neg x$）。

  这个过程返回一个新的、经过化简的公式。
- **DPLL核心逻辑 (dpll 函数):** 这是实现DPLL算法主体的递归函数。它的执行流程严格遵循DPLL算法的步骤：

(a) **循环处理确定性赋值：** 函数首先进入一个循环，持续进行单元传播和纯文字消除，直到无法再找到单元子句或纯文字。
   - 它会遍历所有子句，寻找大小为1的单元子句。如果找到，就将该单元文字加入赋值，并调用 simplify 函数化简公式，然后重新开始循环。
   - 如果没有找到单元子句，它会检查是否存在纯文字。通过一个 unordered_set 统计所有出现的文字，然后遍历这些文字，检查其否定形式是否存在。如果不存在，则该文字为纯文字，同样地，将其加入赋值并化简公式。

(b) **检查终止条件：** 在循环结束后，函数会检查两个终止条件：
   - 如果公式变为空（所有子句都被满足），则返回 true 和当前的赋值解。
   - 如果公式中出现了空子句，说明当前路径导致了冲突，返回 false。

(c) **分裂和递归：** 如果以上条件都不满足，算法会选择一个尚未赋值的变量进行“分裂”。它首先尝试将该变量赋值为真（例如 v），并递归调用 dpll 函数。
   - 如果递归调用成功（返回 true），则将结果向上传递。
   - 如果失败，则进行回溯，尝试将该变量赋值为假（例如 $\neg v$），并再次递归调用 dpll 函数，返回其结果。

- **主函数 (main):** main 函数是程序的入口。它负责处理命令行参数以获取输入的DIMACS文件名，调用 `parse_dimacs` 解析文件，初始化一个空的赋值向量，然后调用 dpll 函数启动求解过程。最后，根据返回结果输出 "SAT" 或 "UNSAT"，并在找到解时打印出具体的赋值情况。同时，它还记录了求解过程所花费的时间。

总的来说，这个实现是一个朴素但完整的DPLL求解器，它正确地实现了算法的核心逻辑，并能够处理标准的DIMACS格式输入，为解决SAT问题提供了一个基础的框架。