基于 KL 算法的电路两路划分实现、分析与优化

项目报告

中山大学集成电路学院 微电子科学与工程　苏依然 No.22362067 金凤琳 No.22362034

2025 年 6 月 22 日

一、项目背景与目标

（一）项目背景

随着摩尔定律的演进，超大规模集成电路的复杂度和集成度呈指数级增长，芯片上承载的晶体管数量已达百亿级别。面对如此庞大的电路系统，如果直接进行全局的布局布线，其计算复杂度将是无法承受的。因此，在整个 EDA 物理设计自动化流程中，电路划分（Circuit Partitioning）作为布局布线的前置步骤，扮演着至关重要的角色。它的核心任务是将一个巨大的电路网络，依据一定的优化目标，切割成若干个规模更小、关联更少的子模块。一个高质量的划分方案，能够显著降低后续步骤的复杂度，优化关键路径延迟，减少跨区域连线长度，从而降低功耗，并最终对芯片的整体 PPA 指标产生深远的影响。

在众多的划分算法中，由 B.W.Kernighan 和 S.Lin 于 1970 年提出的 Kernighan-Lin（KL）算法，无疑是划分领域的一座里程碑。它是一种启发式的迭代改进算法，专门用于解决图的两路划分问题（Bipartitioning）。尽管后世涌现了 FM、多级划分等更先进的算法，但 KL 算法因其直观的核心思想、巧妙的增益计算和迭代策略，至今仍是学习和理解电路划分问题以及启发式优化思想不可或得的重要范例。

（二）项目目标

本项目的核心目标不仅仅是简单地实现一个划分算法，而是希望通过一个完整的工程实践，经历一个“学习-实现-分析-优化”的全过程。具体目标分解如下：

1. 算法实现：基于对 KL 算法原理的理解，对其进行算法层面的实现。
2. 合理评估：设定一套科学的性能指标评价体系，确保全面且真实地反映

所实现算法的性能。

3. 对照实验：为了科学地评估我们所实现的 KL 算法性能，我们计划实现一个简单的贪心策略算法作为 Baseline。

4. 进阶优化：通过将经典 KL 算法与 baseline（Simple_Greedy）进行对比，对 KL 算法的劣势指标（速度）进行有针对性的算法优化。

5. **完成课程项目要求：通过脚本文件（实验启动器），实现“生成网表文件（三种规模）——解析网表文件并进行网图构建——利用不同算法（对照-KL-优化）单次布图快速全流程可视化验证——基于设定的试验指标对每种算法分别使用每种规模数据运行多次实验、收集并处理数据、结果存储可视化呈现”全流程自动控制；并规范接口保证工程项目的可扩展性。**

6. 结果分析：对实验结果进行深入对比分析，从多维度探讨三种算法（baseline－classical_KL－improved_KL_with_BFS_initial）各自的优劣性。

7. 项目启发：争取通过完成项目深化对课程基础知识的理解以及对未来硕士生涯研究方向的开拓与启发。

二、算法原理及实现方案

（一）思路变迁：从理所当然到推倒重来，从一知半解到提出优化

我们项目的整个算法设计过程，可以说是一部生动的“试错史”；它完整记录了我们从一个看似合理但存在根本性缺陷的“预设方案”出发，历经困惑、挫败，最终通过追本溯源，回归原始文献，才得以构建出正确、高效算法的完整心路历程。请允许笔者将我们这段曲折的心路历程写进项目报告中。

1. 最初的设想：预设方案的诞生

项目启动之初，我们依据课程项目要求，在对课程内容忘掉大半的前提下，基于对 KL 算法原理的粗浅理解，想当然设计了一套我们当时认为逻辑自洽的“预设方案”（kl_original）。这个方法的“特点”在于：**D 值只在每一轮迭代开始时计算一次，并在该轮迭代中保持不变。**正是这一“核心特点”，为我们后续必然面临的曲折探索埋下伏笔。

2. 第一次碰壁：迭代的“震荡”与失效

幸好我们不至于忘记 KL 算法本质是启发式算法，需要迭代计算以尽可能接

近全局最优；所以我们在对 kl_original 核心模块进行测试时人为设置了迭代计算，欲观察其收敛性。最初我们设置的跳出迭代的条件是某一轮的最大总增益 Gmi 小于等于 0。而当我们用一个小规模的网表进行测试时，一个奇怪的现象出现了。我们观察到，算法在执行了几轮，成功将割边数降低到一个值之后，就陷入了一种“原地踏步”的怪圈。它会在后续的很多轮迭代中，反复执行一些交换，但最终的割边数却不再有任何改善，始终维持在那个不好不坏的水平上。更糟糕的是，由于它每一轮计算出的最大累积增益 G 始终大于 0，算法的收敛判据（G ≤0）永远无法满足，导致程序如果不是因为我们设置了最大迭代轮数的上限，就会陷入无限循环。这种现象，我们愿称之为“迭代震荡”。

面对这个问题，我们的第一反应是算法的收敛条件太“天真”了。于是，我们立刻着手打了第一个“补丁”：在 kl_original 的基础上，增加了一个更实际的收敛判据。我们引入了一个新参数（patience = 3），实现如果连续三轮迭代中，历史最优的割边数都没有任何改善，就强制认为算法已经收敛并提前退出循环。

3. 第二次重创：割边数不降反增的“灾难”

带着这个“补丁版”的算法，我们再次进行了测试。这一次，程序确实没有陷入无限循环，它在几轮无效的迭代后自动退出了。但当我们查看日志时，一个更令人沮丧，甚至可以说是“颠覆三观”的结果出现了。在某一次迭代中，算法计算出的最大累积增益 G 明明是一个正数，预示着本次交换序列会带来优化，但当交换实际执行后，我们通过重新计算发现，最终的割边数非但没有减小，反而显著增大了！（如图 1，从 9 条割边“优化”到了 15 条）而程序在观察到连续 3 轮割边数不再下降（11-14-15）后，便让程序带着这个糟糕的结果退出了。

```
总运行时间: 0.002423 秒

--- 历史记录 ---
Pass 0: Cut Size = 10, Details: Initial state
Pass 1: Cut Size = 9, Details: Applied 2 swaps.
Pass 2: Cut Size = 11, Details: Applied 2 swaps.
Pass 3: Cut Size = 14, Details: Applied 3 swaps.
Pass 4: Cut Size = 15, Details: Applied 4 swaps.
(pytorch) PS C:\Users\22439\Desktop\EDA_Circuit_Par
KL>
```

图 1. 一次失败的实验结果

虽然实验结果不理想，但我们本能的回避考虑算法本身的问题，妄图增加最大迭代轮次以及调高 patience 使我们不争气的算法越过反向优化的“珠穆朗玛峰”，进入持续正向优化的“马里亚纳海沟”。然而，事与愿违；当我们把 patience（参数）不断调高，本人的 patience（耐心）却随之不断减小；当我们的精神状态最终转换为 patient（病人）时，我们得出了本项目的第一条结论：“**patience 越大，patience 越小**”。

一个旨在减少割边数的优化算法，竟然会做出让结果恶化的决策，这使我们不得不接受一个残酷的现实：**我们对算法的原理理解从根上就出了问题**。经过翻查课件我们发现我们设计算法的**根本问题出现在 D 值更新机制上**。我们的算法中 D 值是不随节点固定而更新的，而课件上的算法 D 值是动态更新的。虽然课件在伪代码部分没有体现在每个迭代轮次（Pass）内，每次锁定（Fixed）一对节点后需要更新 D 值（图 2）；但后续的图示中却清晰地展现了这一过程（图 3）。这个操作的必要性是如此显而易见但我们却没有将其考虑到算法实现方案中，我们至此终于切身体会到：**听课不认真是会付出代价的**。

Divide the network into 2 parts (A, B) of equal size arbitrarily

Repeat until no more vertices are left:

① **Select** $a_i \in$ A, $b_i \in$ B, such that the reduction in cost Δg_i is as large as possible and neither a_i, b_i has been chosen before

② **Swap** a_i and b_i

③ **Let** $G_m = \sum_{i=1}^{m} \Delta g_i$ be the cost of the partition after swapping a_i, b_i

Return (A', B') corresponding to the smallest G_m observed

Find the move sequence 1...m ($1 \le m \le i$), such that $G_m = \sum_{i=1}^{m} \Delta g_i$ is maximized

图 2. 课件 KL 伪代码

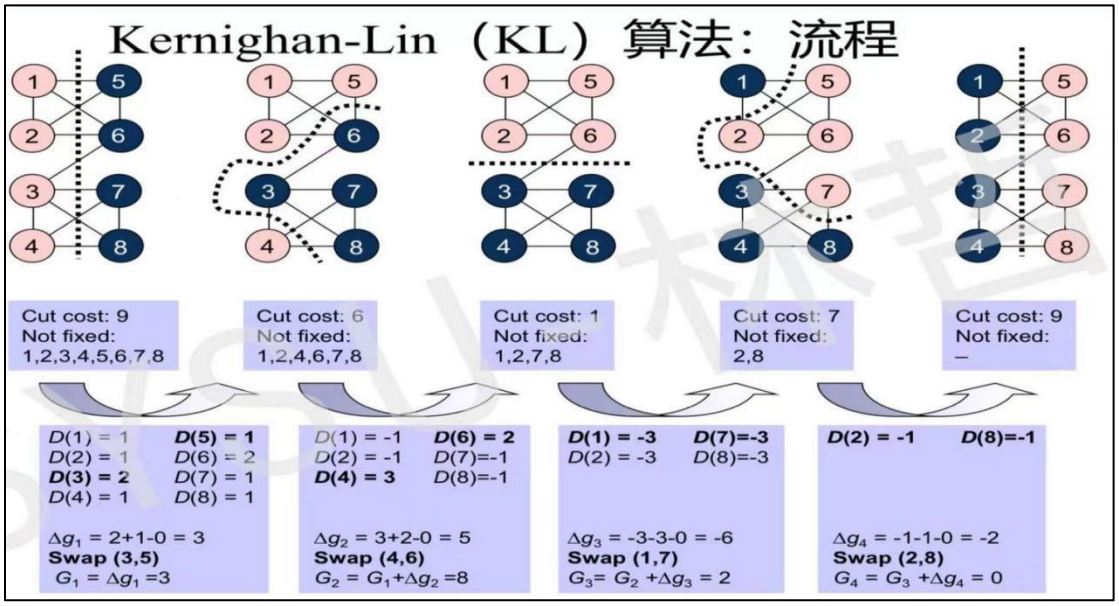


图 3. 课件 KL 算法流程

4. 吃一堑，长一智：完善对 KL 算法的理解

为什么必须动态更新 D 值呢？我们经过学习尝试给出自己的理解如下。KL 算法电路划分很像棋手下棋，锁定节点对计算Δg 序列就像棋手在考虑每一步的落子；每锁定一对节点都是棋手考虑下一步该怎么落子，虽然实际上并没有动，但整个棋盘的局势（D 值分布）在棋手的眼中都会发生翻天覆地的变化；这时进一步选择落子（计算下一个Δg）就必须基于新的棋局（新的 D 值）考虑。依赖一个过时的、静态的 D 值评估体系来指导一整轮复杂的交换，就像让一个棋手闭着眼睛下棋，其决策的不可靠性可想而知。而那个“割边数不降反增”的灾难性结果，正是这种不可靠性最极端的体现：算法的“乐观预期”（G > 0）与残酷的“现实”（割边数增加）完全背道而驰。

我们终于意识到，这个基于“D 值在轮次内固定不变”的预设方案，是彻彻

底底错误的。

5. 追本溯源：回归原始文献

在推倒了最初的方案后，我们陷入了新的困境。我们重新翻看课程课件，课件的流程演示中确实展示了 D 值在交换后发生了变化，但问题是，课件并没有详细说明 D 值具体是如何更新的，是每次都按照 D(v)=|Ec(v)|-|Enc(v)|暴力地重算一遍吗？ 显然不是。因为每次更新 D 值之前进行的操作是“固定节点”，而非“进行交换”；由于网表本质没有变化，所以不能用初始的“detective”手段进行计算；只能在数学层面进行严格估计。那么，事实真的是这样的吗？

为了寻求最权威的答案，我们决定不再依赖任何二手资料，而是去查找并研读了 B. W. Kernighan 和 S. Lin 在 1970 年发表的那篇奠基性的原始论文——《An Efficient Heuristic Procedure for Partitioning Graphs》。这次阅读给我们带来了豁然开朗的体验。原始文献清晰地揭示了 KL 算法的精髓。它不仅强调了在一轮 Pass 中每选择一对节点(a, b)进行交换后必须更新 D 值，更给出了一个极其高效的更新公式。对于分区 A 中一个尚未被锁定的节点 x，当(a, b)交换后，它的新 D 值 D'(x) 并不需要重新遍历所有邻居来计算，而是可以通过旧的 D 值 D(x) 进行简单的修正：D'(x) = D(x) + 2*c(x,a) - 2*c(x,b)。其中 c(x,a) 是 x 和 a 之间的连接权重。我们对这个公式的直观理解是，交换后，a 从 x 的“内部邻居”变成了“外部邻居”，这会使 x 的外部成本增加 c(x,a)，内部成本减少 c(x,a)，D 值的净变化为 +2*c(x,a)；同理，b 从 x 的“外部邻居”变成了“内部邻居”，D 值的净变化为 -2*c(x,b)。B 分区中节点的更新公式亦同理。

这个发现是整个项目的转折点。我们终于理解了 KL 算法的真正面貌：它不是一个静态评估、批量决策的系统，而是一个动态的、每一步决策都基于最新信息反馈的迭代过程。我们立刻决定，修改之前的代码，以这篇原始文献的描述为唯一蓝本，从零开始构建一个正确的、经典的 KL 算法实现，我们将其命名为 kl_classic.py。

6. 基于对算法的深刻理解，确定对照及优化方案

确定了主算法 kl_classic 的具体实现思路后，需要设置 baseline 算法。由于 KL 本身也是一种贪心算法，我们打算探究他与最原始的贪心算法有什么不同。基于此，我们设计了“简单贪心布图算法”。它的流程与 KL 类似，但不进行 Gm

的计算，只追求实现每一步的最大增益；当在某一步中的Δg_max≤0;则立即终止计算。我们认为这种算法虽然会容易深陷“局部最优”导致优化效果欠佳，但由于其每轮计算大概率不会每次锁定网表中的所有节点，其计算速度会比经典快很多（尤其是大规模数据测试）。因此，我们确定针对经典 kl 在运算时间上的劣势进行优化。

拆解 KL 计算流程我们发现，对其各个计算步骤的速度优化空间十分有限。但巧合的是，当时廖思宇老师的《数据结构》刚好在讲《图》，里面提到了基于广度的优先探索算法（BFS）。这给了我们整个项目周期中最天才的启发（kl_BFS_initial）：如果先设置一个基准节点，然后采用广度优先算法进行初始划分，就可以将与基准节点逻辑上相邻（直接相连以及经过最少的节点间接相连）的节点分到同一边，这将大幅减少初始划分图的割边数。**虽然好的初始化分策略不能减少单轮计算的时间（因为每轮计算无论如何都要锁定所有节点），但可以使得迭代结果迅速收敛以减少迭代轮次，从而减小迭代总时间。并且我们惊喜地发现，由于“聚拢密集连接点群”的初始策略，优化算法可能比经典算法更容易趋近于全局最优解，简直是“一箭双雕”。**

（二）最终确定的算法体系

在厘清了核心算法的正确逻辑后，我们围绕着正确的经典 KL 算法，构建了一套包含基线对照和优化探索的完整算法体系。该体系由三个核心算法构成，它们的实现都严格遵循了相同的函数接口，以便于进行自动化的实验和对比分析。核心算法流程图如附图 1 所示。

1. 基线算法：单步最优贪心策略（base_partitioning）

该算法的逻辑直观，代表了一种最朴素的优化思路，其核心思想是“目光短浅，但行动果断”，即在每一步都立即执行当前能找到的、收益最大的单次交换。

实现步骤如下：

（1） 初始划分：与 KL 算法相同，从一个随机的、均衡的两路划分 A 和 B 开始。

（2） 启动迭代循环：进入一个主循环，该循环的终止条件是“找不到任何可以带来正增益的交换”或达到预设的最大迭代次数上限。

（3） 计算 D 值：在每一次主循环的开始，都基于当前的分区 A 和 B，为所

有节点 v 计算其 D 值。

$$D(v) = E(v) - I(v)$$

（4） 寻找最佳单步交换：遍历所有可能的、由一个 A 区节点 a 和一个 B 区节点 b 组成的交换对(a,b)。计算交换这对节点能带来的即时增益 gain(a,b)。其计算公式为：

$$gain(a,b) = D(a) + D(b) - 2c_{ab}$$

在整个遍历过程中，记录下能使增益最大的那个交换对 (a*,b*) 及其对应的最大增益g_{max}。

（5） 决策与执行：在遍历结束后，检查找到的最大增益g_{max}。

① 如果$g_{max} > 0$ ，则说明找到了一个有利可图的交换。立即、永久地执行这次交换，即将 a* 移至 B 区，将 b* 移至 A 区。然后，主循环继续，开始下一次寻找。

② 如果$g_{max} \leq 0$，说明当前状态下任何单步交换都无法减少割边数，算法陷入局部最优。此时，跳出主循环，算法立即终止。

（6） 输出结果：返回最终稳定下来的分区和对应的割边数。

2. 经典 KL 算法 (kl_classic)

这是我们根据原始文献精确复现的核心算法。它与贪心算法的根本区别在于引入了“轮次（Pass）”和“延迟决策”机制，并通过在一轮中高效地更新 D 值来保证决策的准确性。

实现步骤如下：

（1） 初始划分：从一个随机的、均衡的两路划分 A 和 B 开始。

（2） 启动轮次循环(Pass)：进入一个主循环，每一轮循环被称为一个 Pass。该循环的终止条件是“最近一轮 Pass 未能产生任何正的累积增益”或达到预设的最大轮次上限。

（3） 轮次内初始化：在每一轮 Pass 开始时，所有节点均被标记为“未锁定”状态。然后，基于当前分区 A 和 B，为所有节点计算一次初始 D 值：

$$D(v) = E(v) - I(v)$$

（4） 轮次内迭代：进入一个内部循环，该循环将执行 N/2 次（N 为总节点

数)。在每一次内部迭代（第 i 步）中：

① 寻找最佳交换对：在所有未锁定的节点中寻找一对节点 (ai ,bi) （其中 ai ∈A,bi ∈B)，使得交换增益 gi 最大。

$$g_i = D(a_i) + D(b_i) - 2c_{a_i b_i}$$

② 记录与锁定：将本次找到的最佳交换对 (ai ,bi) 及其增益 gi 记录下来。然后，将节点 ai 和 bi 的状态更新为“锁定”，它们将不再参与本轮后续的寻找过程。

③ 高效更新 D 值：根据原始文献的更新公式，为所有剩余的**未锁定**节点更新其 D 值。对于 A 区中任一未锁定节点 x 和 B 区中任一未锁定节点 y，其 D 值更新如下：

$$D'(x) = D(x) + 2c(x, a_i) - 2c(x, b_i)$$

$$D'(y) = D(y) + 2c(y, b_i) - 2c(y, a_i)$$

（5） 寻找最佳交换序列：在内部循环（即一个 Pass）结束后，我们得到了一个增益序列 g1 ,g2 ,…,gN/2 。计算这个序列的累积增益

$$G_k = \sum_{i=1}^{k} g_i$$

并找到一个正整数 k*，使得 Gk* 的值是所有累积增益中最大的。

（6） 决策与执行：检查找到的最大累积增益 Gk* 。

① 如果 Gk* >0，则说明执行前 k* 次交换可以带来整体收益。此时，**实际地、永久地**执行记录下来的前 k* 次交换，以更新分区 A 和 B。然后，主循环继续，开始一个全新的 Pass。

② 如果 Gk* ≤0，说明在本轮中，即使是表现最好的交换序列也无法带来任何优化。算法收敛，跳出主循环，整个算法终止。

（7） 输出结果：返回在所有 Pass 中出现过的、割边数最小的那个划分方案（就是最后一轮迭代结果）。

3. **优化算法：基于 BFS 的初始划分（kl_improvements）**

该算法的核心思想是通过 BFS 初始划分来代替纯随机划分，以期为后续的经典 KL 算法提供一个更好的起点，从而达到加速收敛和提升结果质量的双重目的。

实现步骤如下：

（1） **生成 BFS 初始划分**：此步骤取代了之前算法的随机初始划分。

① **选择起点**：从图的所有节点中，随机选择一个节点作为广度优先搜索（BFS）的起始节点 s。

② **执行 BFS**：从节点 s 开始，执行标准的 BFS 遍历，直到访问完图中所有可达的节点。记录下节点被访问的先后顺序。

③ **构建分区**：根据 BFS 的遍历顺序，将前 N/2 个被访问到的节点放入分区 A，将其余所有节点放入分区 B。这样就创建了一个基于拓扑邻近性的初始划分。

（2） 执行经典 KL 算法：将上一步生成的初始划分(A, B)作为输入，完整地执行前述“经典 KL 算法”的所有步骤（从第 2 步到第 7 步）。

（3） 输出结果：返回经过经典 KL 算法优化后的最终划分方案（最后一轮迭代结果）。

4. **性能分析与评估：**

针对不同的测试用例，系统地评估 KL 算法相对于基线对照算法的划分质量（割边数、平衡性）、收敛速度和运行时间，并进行必要的比较分析。

三、实验设计与测试方法

（一） 配置环境

所有算法实现和实验脚本均基于 Python 3.9 语言环境，并使用了以下核心第三方库：networkx（版本 2.8.4）用于图的构建与操作，numpy（版本 1.22.4）和 pandas（版本 1.4.2）用于数据统计与处理，matplotlib（版本 3.5.1）用于结果的可视化。

算法架构图通过集成在 VSCODE 中的 PlantUML 插件绘制。

（二） 测试数据

我们使用 scripts/generate_netlists.py 脚本来生成测试用的网表数据。为了考察算法在不同问题规模下的表现，我们按要求生成了三种规模的网表：

1. 小规模：10 个节点，20 条边。
2. 中等规模：20 个节点，40 条边。
3. 大规模：50 个节点，100 条边。

为了保证实验的公平性和可重复性，我们通过设置固定的随机种子（seed=42）来生成一套固定的网表文件。在实验对照数据生成及统计过程中，所有算法都使用这同一套网表文件进行测试，以确保对比分析的结论可靠性。如果想更新网表只需运行脚本即可重新生成。

（三） 实验流程

为了对我们最终确定的三种算法（基线贪心算法、经典 KL 算法、BFS 优化 KL 算法）进行全面、公平的性能评估，我们设计了一套由三个核心脚本驱动的、自动化的实验项目测试平台。整个实验全流程分为数据准备、不同算法单次划分全流程快速可视化验证和批量性能测试三个阶段，分别由 generate_netlists.py、create_combined_view.py 和 run_experiments.py 三个脚本协同完成。实验项目测试平台架构如附图 2 所示。

1. **阶段一：测试数据的生成**

实验的第一步是准备统一的、标准化的测试数据集。这个任务由 scripts/generate_netlists.py 脚本负责。

（1） 脚本功能：该脚本的核心功能是生成三种不同规模（小、中、大）的电路网表，并将其以文本文件的格式保存在 data/generated_netlists/ 目录下。

（2） 执行流程：我们首先运行此脚本。它会创建三种规模的网表文件。

（3） 保证公平性：为了保证所有后续实验都在相同的数据集上进行，从而确保对比分析的可靠性，generate_netlists.py 脚本在我们的实验流程中仅需执行一次。后续的算法测试将反复读取这些已经生成好的文件，而不会重新生成，避免了因测试数据不同而引入的变量。

2. **阶段二：不同算法单次划分全流程快速可视化验证**

在进行大规模批量测试之前，我们需要一个能够快速验证单个算法效果并直观展示其过程的工具。这个任务由 scripts/create_combined_view.py 脚本承担。

（1） 脚本功能：该脚本是一个灵活的单一实验启动器。它可以通过命令行参数自由选择要运行的算法（greedy, kl_random, 或 kl_bfs）。

（2） 执行流程：例如，我们可以通过在终端输入

```
python scripts/create_combined_view.py --algorithm kl_bfs
```

命令来启动它。脚本会：

① 自动解析所有三种规模的网表文件。

② 对每个规模的网表，仅运行一次指定的 kl_bfs 算法。

③ 调用 graph_visualizer.py 模块，生成一个包含“原始图——初始划分——最终划分”的 3x3 布局对比图。

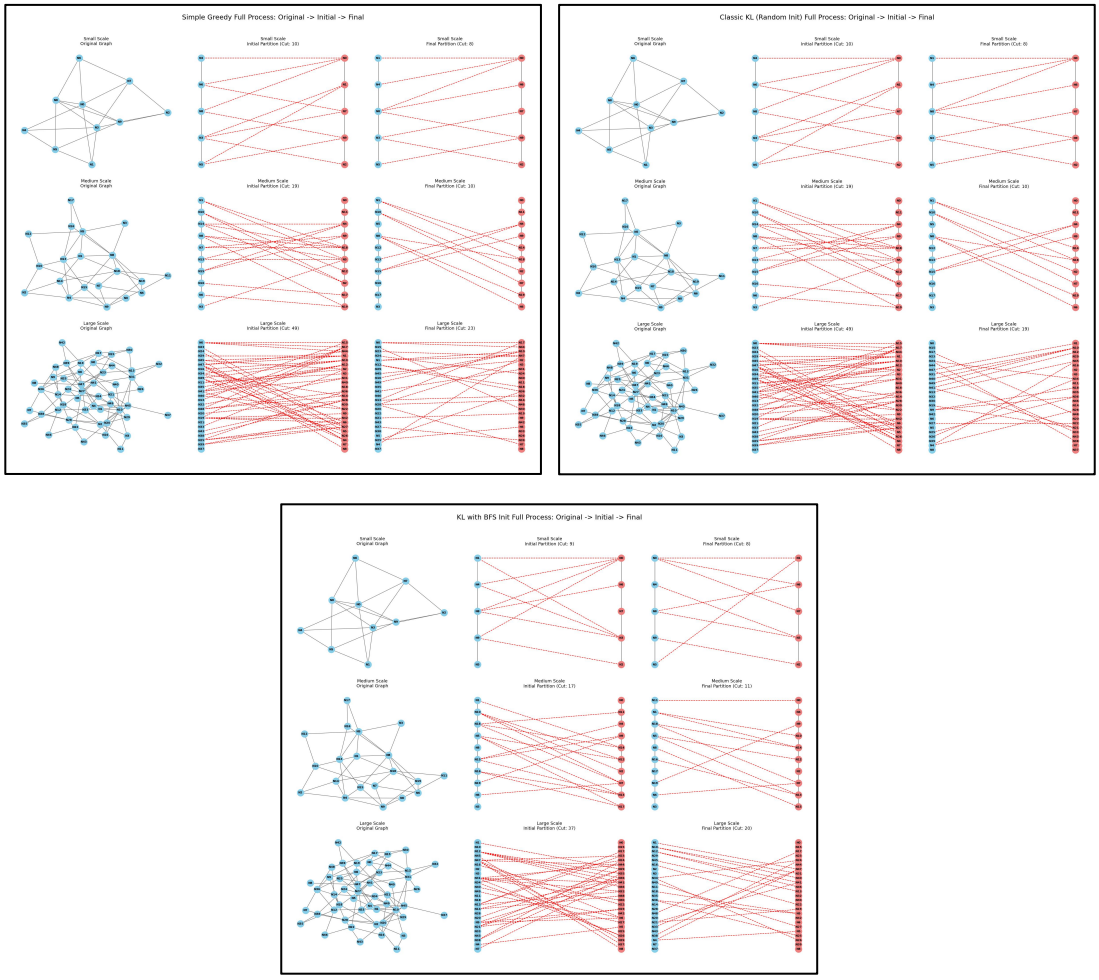


图 4. Complete layout process of the three algorithms

④ 将这张内容详尽的对比图保存在 results/images/ 目录下，并实时弹出显示（注意：Ⅰ.由于个人电脑配置与软件兼容性不同，实时弹出的图片可能存在一定的显示问题；请进行代码检查时以存储在 results/images/ 目录下的 png 格式图片为准！Ⅱ.create_combined_view.py 生成的 3x3 布局对比图中的 initial 与 final 流程网表并不能将每一条边完整展现，原因是为了实现有序且清晰的分区观感，分区节点均保持纵向对齐，这导致了分区

内部边在视觉上的重合；但割边可以不重合地清晰显示，这也是我们主要关注的地方。如果质疑网表边数不符合要求，可以查看最左列的 initial 图，该列清晰展现了三种不同规模网表的每个节点与每条边）。

（3） 实验价值：

这个脚本对于算法的调试和定性分析至关重要。它能让我们在开发过程中，快速地、直观地看到一个算法从初始状态到最终状态的全貌，从而验证算法逻辑的正确性，并对算法效果有一个初步的感性认识。

3. **阶段三：批量实验与量化分析**

在确认所有算法都能正常工作后，我们便进入最终的、最核心的性能测试阶段。这个阶段的所有工作都由 scripts/run_experiments.py 脚本自动化完成。

（1） 脚本功能：

这是一个批量化的、可重复的科学实验框架。它的目标是对所有算法进行多次测试，以消除随机性带来的偶然误差，并收集足够的数据来进行严谨的统计分析。

（2） 执行流程：

在终端运行 python scripts/run_experiments.py 命令。脚本会启动一个三层嵌套的循环：

① 算法层：遍历我们定义好的三种算法。

② 规模层：对于每种算法，遍历小、中、大三种规模的网表。

③ 重复层：对于每个“算法-规模”组合，独立运行 20 次，每一次都使用不同的随机种子来保证初始状态的多样性。

（3） 数据处理与输出：

在 20 次运行结束后，脚本会立即对收集到的数据（如最终割边数、运行时间等）进行统计处理，计算出预先定义的四项核心性能指标（最大/平均割边减少率、平均运行时间、稳定性）。然后，它会生成两种最终产出物：

① CSV 数据报告：为每种算法在 results/generate_data/ 目录下生成一个对应的 CSV 文件（如 kl_classic_performance.csv），以表格形式精确记录下所有性能指标。

② PNG 对比图：在 results/images/ 目录下生成一张名为

experiments_results.png 的“四合一”折线图，将所有算法的四项性能指标绘制在一起，用于直观地进行横向性能对比。

通过这三个脚本的协同工作，我们构建了一套从数据生成、快速验证到批量测试和分析的完整、严谨的实验流水线，确保了我们最终结论的科学性和可靠性。

（四） 性能评估指标

为了全方位地评估算法性能，我们选取了以下四个维度的指标，并给出了其精确的计算公式。在我们的实验中，对于每个“算法-规模”组合，都进行了 N=20 次独立运行。具体的性能评估指标如下：

1. **最大割边减少率 (Maximum Cut-Side Reduction Rate)**

这个指标反映了算法所能达到的**最佳优化潜力**。它是在 N 次运行中，单次运行所能实现的最大优化幅度。

首先，对于第 i 次运行（i=1,⋯,N），计算其独立的割边减少率 Ratei ：

$$Rate_i = \frac{C_{initial,i} - C_{final,i}}{C_{initial,i}}$$

其中，$C_{initial,i}$ 是第 i 次运行的初始割边数，$C_{final,i}$ 是第 i 次运行的最终割边数。

最大割边减少率 MaxRate 即为这 N 个比率中的最大值：

$$MaxRate = max(Rate_1, Rate_2, \ldots, Rate_N)$$

2. **平均割边减少率 (Average Cut-Side Reduction Rate)**

这个指标反映了算法稳定、平均的优化能力，是衡量算法有效性的核心指标。它是 N 次运行的割边减少率的算术平均值。

$$\bar{Rate} = \frac{1}{N}\sum_{i=1}^{N} Rate_i = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{C_{initial,i} - C_{final,i}}{C_{initial,i}}\right)$$

3. **平均算法运行时间 (Average Algorithm Execution Time)**

这个指标直接反映了算法的计算效率。它是 N 次运行耗时的算术平均值。

$$\bar{Time} = \frac{1}{N}\sum_{i=1}^{N} t_i$$

其中，ti 是第 i 次运行的耗时。

4. **结果稳定性（标准差表示）(Result Stability)**

这个指标反映了算法性能的鲁棒性，即其结果受随机因素影响的程度。我们使用 N 次运行得到的最终割边数的标准差来度量。标准差越小，结果越稳定。

$$\sigma_{cut} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(C_{final,i} - \overline{C_{final}})^2}$$

其中，$\overline{C_{final}}$ 是 N 次运行最终割边数的平均值，即：

$$\overline{C_{final}} = \frac{1}{N}\sum_{i=1}^{N} C_{final,i}$$

四、结果分析

在完成了系统的实验后，我们得到了清晰的数据报告和可视化图表。这些结果不仅验证了我们的理论预期，更深刻地揭示了三种算法在性能上的内在差异与权衡。

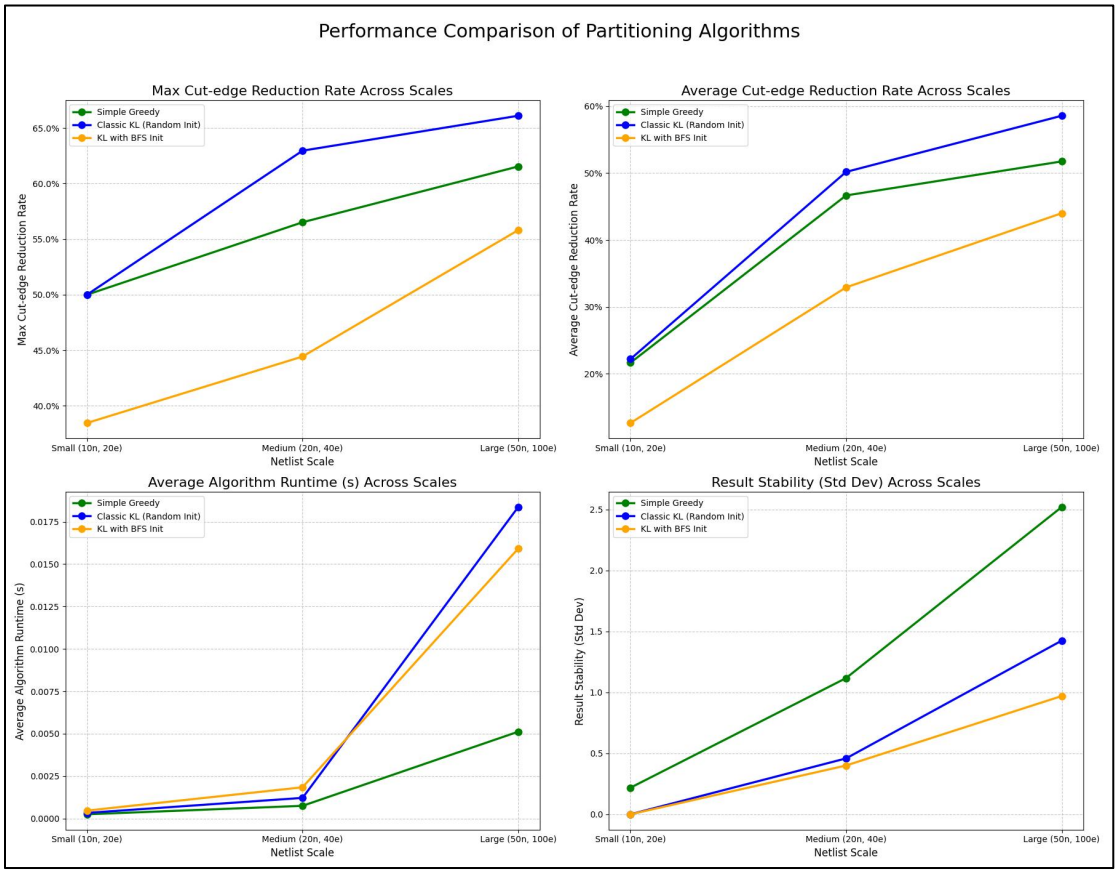


图 5. experiments_results_view

（一）划分质量分析

从实验结果图的左上角“最大割边减少率”和右上角“平均割边减少率”两个子图中，我们可以得出关于划分质量的明确结论。无论是看峰值性能还是平均性能，经典 KL 算法的表现都远远优于作为基线的简单贪心算法。这说明 KL 算法的“轮次（Pass）”迭代机制是极其有效的，成功避免了简单贪心算法极易陷入的“局部最优”陷阱。在简单贪心算法早早停止优化的地方，KL 算法依然能找到路径，继续降低割边数。

但是，我们也注意到，无论是“最大割边减少率”还是“平均割边减少率”，BFS 优化版的 KL 均小于其他两种算法，**这不是因为其划分质量差，而是由统计指标计算公式决定的**——因为 BFS 优化了初始划分；使得初始割边大幅减小，而我们对割边减小率的计算公式为：

$$Rate_i = \frac{C_{initial,i} - C_{final,i}}{C_{initial,i}}$$

由于 BFS 优化版的 KL 的$C_{initial,i}$较小，故而割边减少率小；所以**实验结果图恰恰证明了 BFS 初始划分策略优化对于各种规模数据普遍有效**。由于 BFS 优化版的 KL 与经典 KL 只有初始划分策略的不同，使得其最终划分质量应与经典 KL 相当（或略优于，见前文算法设计思路分析），而这不是我们重点关注的部分。我们重点关注的是：BFS 优化版 KL 能否减少 KL 总运算时间，而这在图中的左下部分子图展示。

（二）算法效率分析

在左下角的“平均算法运行时间”子图中，我们看到了不同算法之间在效率上的巨大差异。正如理论预期的那样，简单贪心算法（绿色折线）凭借其极其简单的单步迭代逻辑，运行速度最快，几乎不耗费什么时间。而两种 KL 算法，由于其内部复杂的 Pass 机制，计算成本要高得多，耗时也显著更长。

然而，这张图中最令人振奋的发现，来自两种 KL 算法的对比。我们看到，虽然橙色折线（BFS 改进版的运算耗时）在小、中规模图上与蓝色折线（经典 KL 耗时）接近重合；但在大规模的图上，其明显低于蓝色折线。这完美地印证了我们之前的假设：虽然 BFS 本身会带来一点点预处理的开销，但它提供的高质量起点，使得 KL 算法收敛所需的总迭代轮数显著减少。最终，节省下来的迭代时间，不仅弥补了 BFS 的开销，甚至还让算法的总体运行时间变得更短（**尤其适用于大**

规模数据）。这是一个非常理想的优化结果，我们不仅提升了结果的质量，还提升了算法的效率。

（三）结果稳定性分析

右下角的“结果稳定性”子图则揭示了算法性能的另一个重要维度。最出色的无疑是我们优化的 BFS 版本 KL 算法（橙色线），它的标准差在所有规模上都是最低的。这说明这个算法非常“靠谱”，不管我们重复实验多少次，它总能稳定地给出高质量的划分结果。这背后的原因也很好理解，因为 BFS 初始划分这个步骤本身就不是纯随机的，它利用了图的结构信息，所以每次给出的“起点”质量都很高，后续 KL 算法的发挥自然也就非常稳定了。

而经典的 KL 算法（蓝色线）则表现得中规中矩。它的稳定性比贪心算法要好，因为它强大的 Pass 机制让它有能力摆脱一部分特别差的局部最优解，不至于像贪心那样完全听天由命。但它毕竟还是从一个纯随机的点出发，所以结果还是会有一些波动，尤其在大规模数据测试，其稳定性无法和经过 BFS“校准”过的优化版相提并论。

最有戏剧性的是简单贪心算法（绿色线）的表现。在中等和大规模的图上，它的标准差急剧飙升，稳定性与其余两种算法相去甚远。贪心算法的致命弱点就是它会立刻陷入遇到的第一个“局部最优”的坑里，再也爬不出来。对于复杂的图来说，这种坑遍地都是，而且有好有坏。20 次的随机初始划分，就像是把算法随机扔在了地图的 20 个不同位置，有的离好坑近，有的离坏坑近。由于贪心算法没有 KL 算法那种“爬山”的能力，所以它的最终结果完全取决于“出生点”的好坏，结果自然非常分散，稳定性也就无从谈起了。

五、对三种算法性能和局限的讨论

通过这次系统的实验和分析，我们对这三种算法的性能画像、适用场景和内在局限有了更深刻地认识。

首先，简单贪心算法，作为基线，它的表现可以说是在意料之中。它的最大优点是“快”，逻辑简单，实现容易。但它的缺点也同样致命，那就是“短视”。它对解空间的探索极其有限，几乎不可避免地会陷入质量很差的局部最优解中。在实际工程中，除非是对性能要求极低且对速度要求极高的场景，否则基本不会采用如此简陋的策略。它在我们项目中的最大价值，就是作为一把“标尺”，清

晰地度量出更复杂算法的价值所在。

其次，经典 KL 算法（随机初始划分），展现了一个优秀启发式算法的典型特征。它通过牺牲一定的计算时间，换来了远超贪心算法的划分质量。它精妙的 Pass 机制，允许算法“以退为进”，是其能够找到高质量解的关键。然而，它的性能也并非完美无瑕。我们从实验数据中清晰地看到，它的结果受到随机初始化的影响较大，存在一定的不稳定性。同时，对于一个糟糕的起点，它可能需要较多的迭代轮次才能收敛，时间开销不容忽视。在实际应用中，为了获得可靠的结果，通常需要像我们的实验一样，采用多次运行取最优值（Multi-Start）的策略来弥补其不稳定性。

最后，我们实现的改进版 KL 算法（BFS 初始划分），可以说是本次项目中最大的亮点。实验数据表明，这是一个非常成功的优化。它从问题的“根源”之一——初始状态的随机性入手，通过一个计算成本很低的 BFS 预处理，实现了“一石三鸟”的效果。第一，它保证了最终解的平均质量和最优质量；第二，它通过减少收敛所需的迭代轮数，缩短了算法的平均运行时间；第三，它增强了算法的稳定性，使得单次运行就能得到一个非常接近最优水平的可靠结果。当然，它也并非没有局限。BFS 策略本身也是一种启发式算法，它并不能保证每次都生成绝对最优的起点。对于某些特殊的图结构，BFS 生成的划分甚至可能不如随机划分。但从我们的实验结果来看，在随机生成的、具有一定通用性的图上，这种策略是明确有效的。这也启发我们，对于更复杂的、真实的电路图，或许可以设计出更具针对性的、利用更多电路先验知识的初始划分策略。

六、项目中的挑战与收获

回顾整个项目历程，我们遇到的挑战与获得的成长同样深刻。

最大的挑战无疑来自项目初期对 KL 算法原理的“想当然”。我们最初构建的 kl_original 算法的失败，给了我们一次印象深刻的教训：在没有完全、精确地理解一个算法的每一个细节之前，任何看似合理的简化都可能埋下致命的隐患。从发现“迭代震荡”，到见证“割边数不降反增”的灾难性结果，再到最终通过阅读原始文献推倒重来，这个过程虽然充满挫败感，但它强迫我们养成了“追本溯源、尊重原创”的严谨科研习惯。我们深刻地认识到，对于那些经历了数十年考验的经典算法，其每一个设计细节背后都可能蕴含着作者深刻的思考和权衡，

绝不能轻易地想当然。

另一个挑战在于实验框架的设计与实现。要对一个具有随机性的启发式算法进行公平的性能评估，单次运行的结果是毫无意义的。如何设计一个能够进行多次运行、自动收集数据、科学计算统计指标并最终将结果清晰可视化的自动化流程，是对我们工程实践能力的一次综合考验。从设计 run_experiments.py 的逻辑，到处理不同算法的接口差异，再到使用 pandas 和 matplotlib 进行数据处理和绘图，这个过程让我们对“如何做实验”这件事本身有了更深入的理解。

通过这个项目，我们的收获是多方面的。首先，我们不再仅仅是算法的“使用者”，而是通过亲手实现、调试和优化，成为算法的“理解者”。我们对 KL、贪心、BFS 等图算法的内在机制有了更具体、更感性地认识。其次，我们的工程能力得到了极大地锻炼，学会了如何在一个小型的项目中进行模块化设计、接口定义、版本迭代和测试验证。最后，也是最重要的，我们体会到了科学研究的完整过程：从提出问题（开题报告），到初步尝试（预设方案），再到遭遇失败、分析问题、寻找解决方案（阅读文献），最终到设计实验、验证猜想、得出结论。这个完整的闭环体验，远比单纯地实现一个既定功能的课程作业要宝贵得多。

七、小结

本项目围绕经典的 KL 电路划分算法，成功地经历了一个从初步实现、遭遇问题、深入研究、精确复现到最终优化的完整研发流程。我们首先实现了我们最初理解的一个简化版 KL 算法，并通过测试发现了其因“D 值不更新”而导致的逻辑缺陷。在回归并研读原始文献后，我们精确地复现了包含轮次内 D 值高效更新的经典 KL 算法。

为了系统地评估算法性能，我们设计并实现了一个简单的贪心算法作为性能基线，以及一个采用广度优先搜索（BFS）进行初始划分的优化版本。通过搭建自动化实验框架，在三种不同规模的网表上对这三种算法进行了多次重复实验，并从划分质量、运行效率和结果稳定性三个维度对它们进行了全面的量化对比。

实验结果清晰地表明：经典 KL 算法相较于简单贪心策略，在划分质量上有巨大优势，验证了其复杂设计策略的有效性；而我们提出的基于 BFS 初始划分的优化版本，相较于经典 KL 算法，不仅保证了划分质量，还缩短了平均运行时间，并极大地增强了结果的稳定性，证明了该优化策略的成功。整个项目不仅加深了

我们对启发式优化算法的理解，也极大地锻炼了我们的工程实践能力和科研探索精神。

八、参考文献

[1] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," The Bell System Technical Journal, vol. 49, no. 2, pp. 291-307, Feb. 1970.

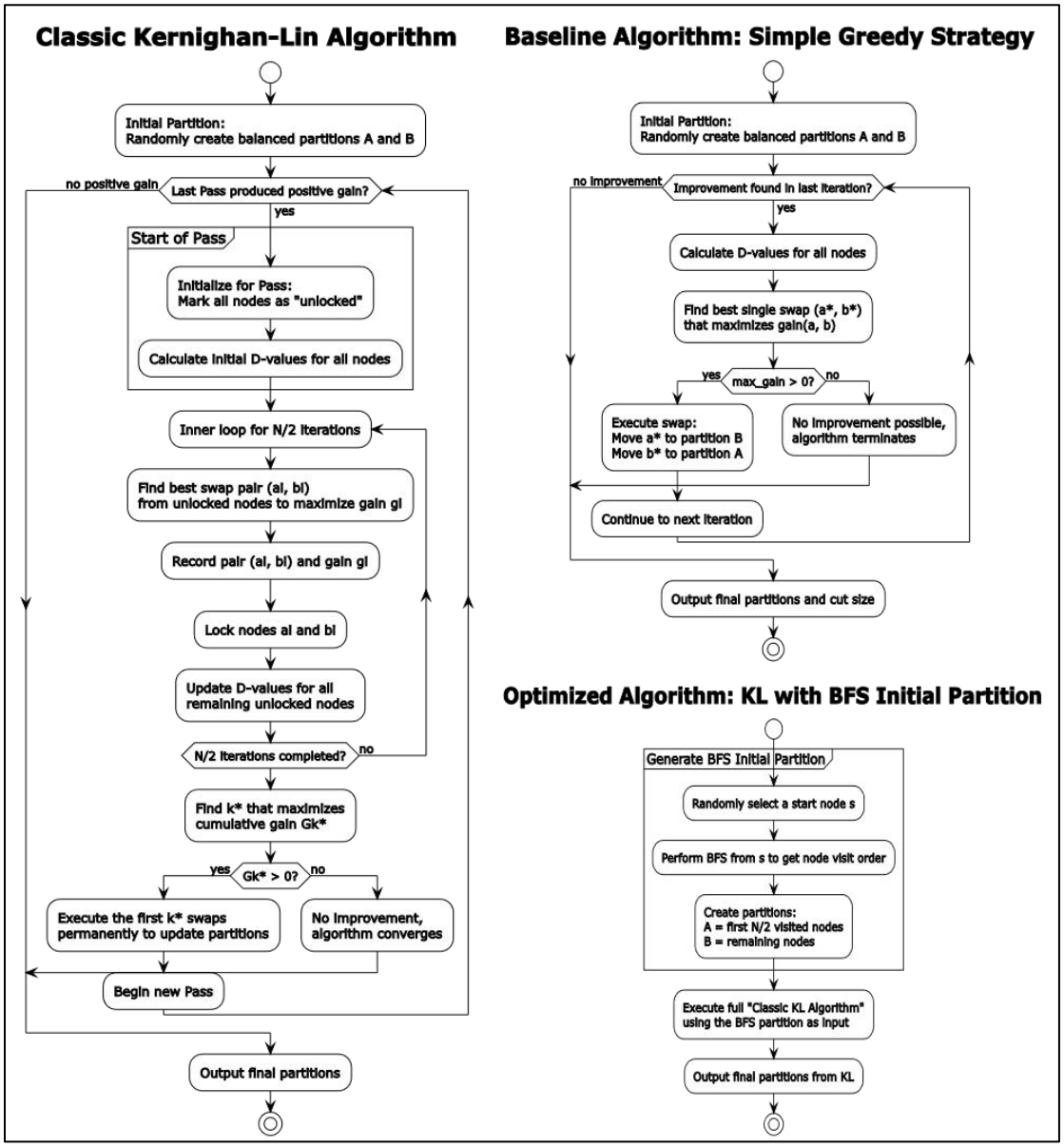
[2] EDA 课程项目开题报告_原始提交版.pdf

[3] 课程项目-EDA Spring 2025.pdf

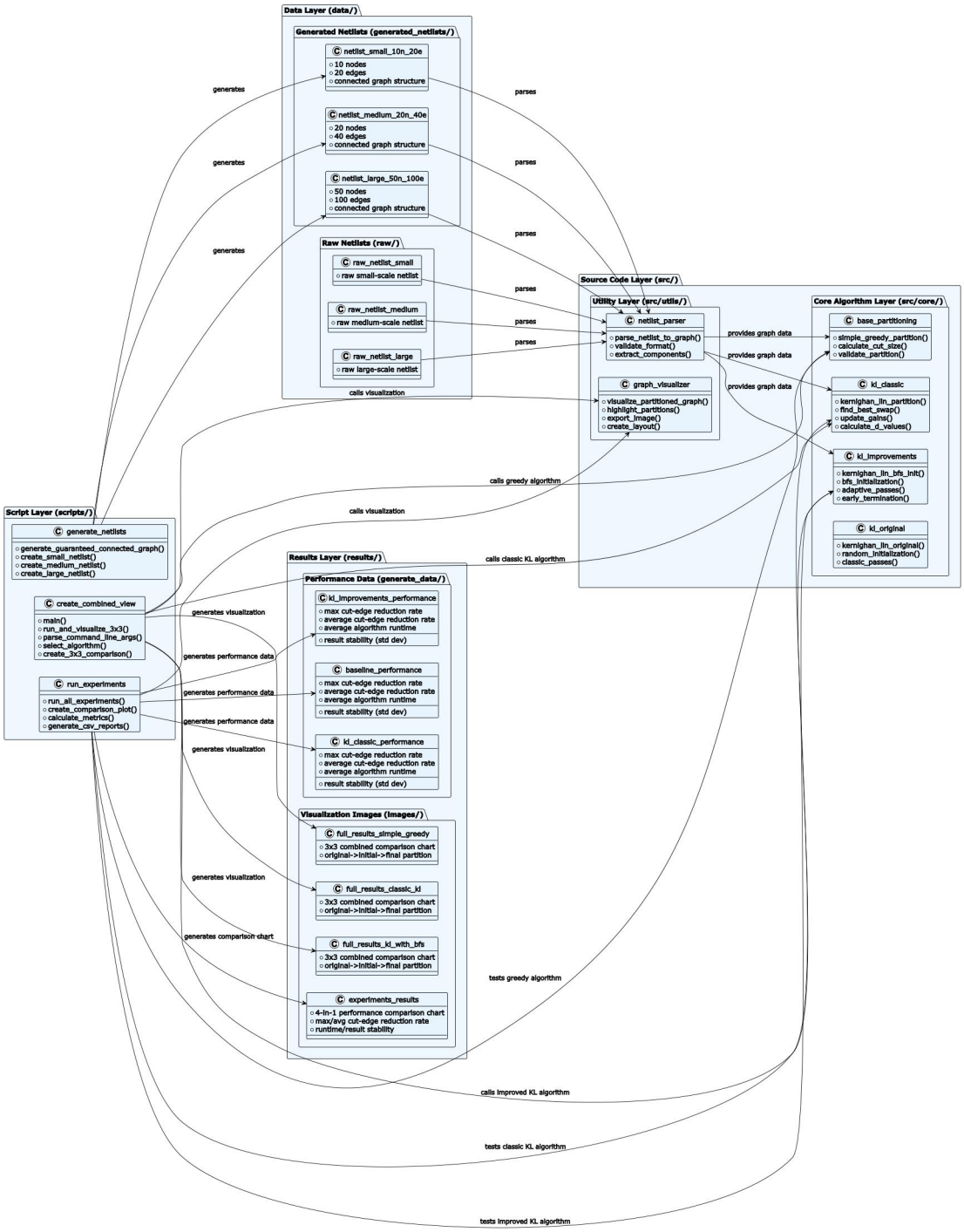
[4] L4-电路划分与布图规划.pdf

九、附录

1. 附图：



附图 1. 三种核心算法流程图



附图 2. 实验项目测试平台架构

2. 附表：

附表 1. 算法核心性能指标数据统计（运行一次 run_experiments.py）

Algorithm core performance indicator data statistics				
Algorithm Name	**Performance Indicators**	**Small (10n, 20e)**	**Medium (20n, 40e)**	**Large (50n, 100e)**
baseline	**Max Cut-edge Reduction Rate(%)**	50.00%	56.52%	61.54%
	Average Cut-edge Reduction Rate(%)	21.63%	46.66%	51.75%
	Average Algorithm Runtime (s)	0.000247	0.000746	0.005116
	Result Stability (Std Dev)	0.2179	1.1169	2.5194
kl_classic	**Max Cut-edge Reduction Rate(%)**	50.00%	62.96%	66.10%
	Average Cut-edge Reduction Rate(%)	22.18%	50.18%	58.58%
	Average Algorithm Runtime (s)	0.000328	0.001214	0.018364
	Result Stability (Std Dev)	0	0.4583	1.4239
kl_improvements	**Max Cut-edge Reduction Rate(%)**	38.46%	44.44%	55.81%
	Average Cut-edge Reduction Rate(%)	12.62%	32.91%	44.00%
	Average Algorithm Runtime (s)	0.000464	0.00184	0.015913
	Result Stability (Std Dev)	0	0.4	0.9695