Homework 2 Report

114062584 王子銜

October 21, 2025

1. Introduction

本作業目標爲實作並修改 **Fiduccia–Mattheyses (FM)** 演算法，用以解決二分與四分之最小割問題 (2-way / 4-way min-cut partitioning)。透過平衡條件的限制，盡可能最小化跨群組連接的網路數量 (cut size)。

2. Results

```
+-----------------------------------------------+
|                                               |
|    This script is used for PDA HW2 grading.   |
|                                               |
+-----------------------------------------------+
host name: ic51
compiler version: g++ (GCC) 9.3.0

grading on 114062584:
 checking item          | status
------------------------|--------
 correct tar.gz         | yes
 correct file structure | yes
 have README            | yes
 have Makefile          | yes
 correct make clean     | yes
 correct make           | yes

  testcase |      #ways |   cut size |    runtime | status
-----------|------------|------------|------------|--------
   public1 |          2 |        181 |      40.87 | success
   public1 |          4 |        543 |      59.57 | success
   public2 |          2 |       1571 |      65.25 | success
   public2 |          4 |       3656 |      96.71 | success
+-----------------------------------------------+
|                                               |
|   Successfully write grades to HW2_grade.csv  |
|                                               |
+-----------------------------------------------+
```

Figure 1: 上圖爲運行公開測資結果

3. Implementation Details

3.1 演算法差異

- **Multi-start** 先以 `generateInitPartitions` 大量產生起始分割（貪婪 + 隨機），用 OpenMP 平行評估篩選成 `topM`，之後在時間預算內反覆對前段種子執行 FM 強化，動態更新最佳解。

- **Time budget & watchdog.** 設定總時限（`TOTAL_BUDGET`），並以 `watchdog_run` 在尾端強制輸出當前最佳解，避免逾時無輸出。

- **Randomized `initBuckets`.** 每次 FM pass 開始時，於 `initBuckets()` 先對所有 cell 之索引做隨機洗牌（`shuffle`），再依該順序計算 gain 並插入 bucket。此舉能打破固定掃描偏好，避免局部模式重複（降低相同初始分割造成的走訪同質性），讓高增益候選在不同 pass 有機會優先被考慮。
  同時，演算法不會在任意一次 `fmPass` 無改善時立刻終止，而是容許連續多次（例如三次）增益爲零後才結束整個 FM 流程。這樣設計可讓演算法在重新初始化 bucket、重新隨機化 cell 掃描順序後，仍有機會跳出前一輪的局部最佳，獲得些微但穩定的 cut size 改善。

- **Top-M 評估機制（10 次 `fmPass`).** 對每個起始分割 init：複製一份 `Parser`（不污染全域狀態），以該 init 執行至多 10 次 `fmPass` 作爲快速評估；記錄其最終 `cutSize` 與快照，收集成候選集後依 `cut` 升冪排序，僅保留前 M 筆爲 `topM`。限制 10 pass 可在種子階段兼顧覆蓋度與速度，將完整強化留給後續總時限內的精煉階段。

Algorithm 1 GreedyRandomInit(C, size[·], α)

Require: Cell set C, cell sizes size[·], random perturbation factor α
Ensure: Initial 2-way partition vector init[·]

Initialize an empty list L
for each cell $v \in C$ **do**
    $w_v \leftarrow \text{size}[v] \times(1 + \text{Uniform}(0, \alpha))$ ▷ apply random perturbation
    Append (w_v, v) into L
end for
Sort L in descending order of w_v
sum[0] $\leftarrow 0$, sum[1] $\leftarrow 0$
for each (w_v, v) in L **do**
    if sum[0] $\leq$ sum[1] **then**
        init[v] $\leftarrow 0$
        sum[0] $\leftarrow$ sum[0] + size[v]
    else
        init[v] $\leftarrow 1$
        sum[1] $\leftarrow$ sum[1] + size[v]
    end if
end for
return init[·]

Algorithm 2 SeedScreeningAndTopM(P, k, M, T_{seed})

Require: Parser P, partitions $k \in \{2, 4\}$, Top-M M, seed time T_{seed}
Ensure: Top-M list of (cut, P') pairs ranked by cut size

```
Initialize empty list L
parallel for each random seed in parallel do
    init ← GreedyRandomInit(P.cells, P.size, 0.2)
    P' ← copy of P
    Run FM on P' with init and at most 10 passes
    cut ← P'.cutSize
    Append (cut, P') to L
    if elapsed time ≥ T_seed then
        break
    end if
end parallel for
Sort L ascending by cut
return the first M elements of L as Top-M
```

3.2 Bucket List 資料結構

本實作確實使用 bucket list，但未採用 STL 的 `std::list`，而是以陣列搭配僞指標（index-based linked list）自行實作。每個節點僅以整數索引表示前後鏈結（`prev`, `next`），所有節點集中於連續陣列 `bucketNodes[]` 中。

結構範例：

```
struct BucketNode {
    int prev = -1; // 指向前一個節點（以索引表示）
    int next = -1; // 指向下一個節點（以索引表示）
};
vector<BucketNode> bucketNodes;   // 連續配置的節點池
vector<int> buckets;              // 每個 gain 對應一個 head 索引
```

每個 bucket 僅記錄 head index，若爲 `-1` 表示該 bucket 爲空。插入、移除節點時僅需修改陣列索引欄位，而不需操作實際指標。

效能考量： 理論上 STL 的 `list` 插入與刪除操作皆爲 $O(1)$，但實務上由於：

- STL 節點分散配置（非連續記憶體），cache miss 頻繁；
- 每次插刪需經由 allocator（`new`/`delete`）動態配置；
- iterator 封裝產生額外的間接存取成本；

因此在 FM 演算法中大量呼叫 `addToBucket()` 與 `removeFromBucket()` 時，STL 版本的常數時間開銷顯著。

相對地，以陣列儲存節點的「僞指標 linked list」：

1. 採連續記憶體配置（cache locality 佳）；
2. 僅以整數索引操作，完全避免動態配置；
3. 插入、刪除僅修改固定數量索引欄位，常數因子極小；

實際經由 `perf` 量測後，確實能減少大量 allocator 開銷與 cache miss，整體 bucket 操作效能優於 STL `list` 實作。

3.3 最大部分和與回復機制

在單一 FM pass 中，我們對每一次可行移動（將某 cell 從 fromP 移到 toP）計算其增益 g_i，並維護累積增益序列

$$S_t \;=\; \sum_{i=1}^{t} g_i \,.$$

pass 結束後，取使 S_t 最大的索引 $t^\star = \arg\max_t S_t$，表示「保留前 $t^\star$ 個移動」能得到本 pass 的最佳淨改善。接著將第 $t^\star+1$ 個以後的移動全部回復（undo），同步回滾 partition 尺寸與 net 計數，回到最佳前綴所對應的狀態。

在程式中，我們以 `moves`（紀錄 ⟨cellIdx, fromP, toP⟩）與 `sums`（保存 S_t）兩個容器實作；尋得 `best` 後，對 $i = |moves|-1, |moves|-2, \ldots,$ `best` 逐一反向套用（將先前的 from→to 還原爲 to→from），並更新 `partitionSizes` 與 `netCount`。最後解除所有鎖定（`locked=false`），使結束狀態可作爲下一個 pass 的起點。

此外，若單次 **pass** 無改善（$\max S_t \le 0$），本實作不會立即停止整體 FM；而是允許連續多次（例如三次）無改善才終止。原因是每個 pass 會在 `initBuckets()` 重建 bucket 並隨機洗牌 cell 掃描順序，常能從不同路徑取得微幅下降。

Algorithm 3 FM_Pass_with_MaxPrefix_Restore()

```
InitBuckets
moves ← [ ], sums ← [ ], sum ← 0
for i = 1 to n do
    (cell, g) ← FindBestMove
    if cell = −1 then
        break
    end if
    from ← cells[cell].partition;  to ← 1−from
    append (cell, from, to) to moves
    sum ← sum + g;  append sum to sums
    MakeMove(cell); RemoveFromBucket(cell); UpdateGains(cell)
end for
if moves is empty then
    return 0
end if
(maxGain, best) ← (0, 0)
for t = 1 to |sums| do
    if sums[t] > maxGain then
        maxGain ← sums[t];  best ← t
    end if
end for
for i = |moves| − 1 downto best do
    (c, f, t) ← moves[i]
    cells[c].partition← f; update size/netCount
end for
unlock all cells; return maxGain
```

3.4 四分割 (4-way) 擴充

本實作採用 階層式二分法（**hierarchical bipartition**）來達成四分割，而非直接進行 4-way 的 gain 計算。具體而言，演算法先執行一次二分（2-way）FM 分割，將電路切爲兩半，接著分別在每一半上再各自進行一次 2-way FM，即可得到最終的四個區塊。

避免不平衡問題　若直接沿用原本二分時的允許比例範圍（例如 $[0.45, 0.55]$），第二層遞迴後會導致最終四分區塊嚴重不平衡：

$$0.45^2 = 0.2025 < 0.225, \quad 0.55^2 = 0.3025 > 0.275$$

因此，在進行四分割時，本實作於兩層 2-way 分割皆採用較緊的比例限制區間：

$$[0.48, 0.52],$$

如此可保證最終四個區塊的面積比例皆維持於約 $[0.225, 0.275]$ 之間，有效避免因遞迴誤差累積導致的全域不平衡。

實作方式　階層式四分割流程如下：

1. 進行第一層 2-way FM 分割，使用 $(l, r) = (0.48, 0.52)$；
2. 根據結果建立兩個子電路（`subA`, `subB`），僅保留對應的 cell 與 net；
3. 對每個子電路再次執行 2-way FM 分割，採相同比例限制；
4. 將兩層結果合併（`mergeBack`）得到最終四個群組。

此方法兼具 2-way FM 的穩定性與良好收斂特性，同時透過嚴格比例約束確保四分割結果的全域平衡。

4. Discussion

在這次作業中，我發現 FM 是一個非常強大的貪婪式演算法，效果遠超過我原先的預期。起初我嘗試使用基因演算法（GA）來產生較佳的初始分割，希望藉此提升 FM 的最終結果品質。然而在多次實驗後發現，FM 本身的收斂效果已相當優異，而 GA 難以在合理時間內找到明顯更好的初始化。其原因在於，若要評估個體的優劣，GA 的 fitness 函式仍需完整執行一次 FM，導致其計算下界與直接執行 FM 幾乎相同。換言之，GA 的搜尋成本遠高於其潛在收益。因此，最有效的策略反而是「暴力地」提升 FM 嘗試次數，而非改進初始化方法。

因此在作業後半階段，我的重心轉向效能優化，幾乎像在做平行程式設計作業一樣，著重於提升執行效率與時間利用率，使程式能在相同的總時限內對更多初始分割執行 FM 演算法，以獲得更佳的結果。