NYCU PDA Lab3: Optimizer and Legalizer Co-optimization

學號：A121163
姓名：張浚騰

Pseudocode

目錄

1. Pseudocode
2. Time Complexity Analysis
3. Special Features of Your Program
4. Algorithm Architecture
5. Conclusion

Pseudocode

---

1. Parse LG File

> **功能：解析 LG 檔案，提取 DieSize、Placement Rows、Cells 等資訊。**

```
Input: LG file
Output: Parsed DieSize, Rows, and Cells
-------------------------------------------------
For each line in LG file:
  - If "Alpha" or "Beta":
      Store the values for later calculations
  - If "PlacementRows":
      Parse row details and initialize boundaries
  - If "DieSize":
      Define the boundaries of the Die
  - Otherwise:
      Parse Cell details and add to cell list
Return parsed data
```

---

2. Initialize SubRows

> **功能：初始化 Row 的 SubRows，並處理固定元件的覆蓋問題。**

```
Input: Rows, Fixed Cells
Output: Updated Rows with SubRows initialized
-------------------------------------------------
For each Row:
  - Create a single SubRow spanning the entire row width
  - For each Fixed Cell in the Row:
      - If it overlaps the SubRow:
          - Split SubRow into multiple smaller segments
          - Update the available space (freespace)
Return the updated Rows
```

---

3. Remove Cell

>

功能：移除特定 Cell，更新 SubRow 空間並處理合併邏輯。

```
Input: Rows, Cell to remove
Output: Updated Rows with space reclaimed
-------------------------------------------------
Find the Row where the Cell resides:
  - If SubRow(s) adjacent to the Cell can be merged:
      - Merge them into a single SubRow
  - Otherwise:
      - Reclaim space occupied by the Cell
Update the Row's SubRows accordingly
```

---

4. Insert Cell

功能：將 Cell 插入合適的 Row 和 SubRow 中，確保合法放置。

```
Input: Rows, Cell to insert
Output: Updated Rows with Cell placed or error message
-------------------------------------------------
Identify Row(s) where the Cell can fit:
  - Check horizontal space in SubRows
  - If multi-row placement is required:
      - Verify vertical alignment across Rows
If placement is possible:
  - Place the Cell and split the SubRow(s)
Else:
  - Log error
```

---

5. Process Banking Instruction

功能：解析 Banking 指令，移除指定元件並插入合併元件。

```
Input: Banking instruction, Rows, Cells
Output: Updated Rows and Cells
-------------------------------------------------
Parse the instruction to:
  - Identify cells to remove
  - Define the merged Cell's properties
Remove specified Cells:
  - Adjust SubRows to reclaim space
Insert the merged Cell:
  - Attempt placement using Insert Cell logic
  - If successful:
      - Update placement and output results
  - Else:
      - Log failure
```

以下是詳細的 **Time Complexity Analysis**，並用表格進行清晰的排版，適合直接展示於 Markdown 編輯器（如 HackMD）或報告中。

以下是修改後的版本，針對 Markdown 的結構進行了優化，確保排版更加清晰、公式渲染正確，並增加了條理性。

Time Complexity Analysis

概述

此演算法由五個主要模組組成，每個模組的時間複雜度如下：

詳細模組分析

表格整理

模組	操作描述	時間複雜度	複雜度來源
Parsing LG File	遍歷 LG 檔案每行，提取 `DieSize`、`Rows` 和 `Cells`。	(O(n))	每行解析操作為 $O(1)$，假設檔案有 n 行。
Initializing SubRows	遍歷每個 Row，檢查是否有固定元件覆蓋，並根據覆蓋情況切割 SubRows。	$O(R \times C)$	(R)：行數，(C)：固定元件數量，每行檢查所有固定元件。
Removing Cells	找到指定的元件，回收其占用空間，並嘗試合併相鄰的 SubRows。	(O(\log(S)))	(S)：每行 SubRows 的數量，二分查找操作耗時 (O(\log(S)))。
Inserting Cells	檢查元件是否可放置在對應的 Row 和 SubRows，插入並切割空間段，若跨行需檢查垂直合法性。	(O(h \times \log(S)))	(h)：元件跨越的行數，每行 SubRows 查找為 (O(\log(S)))。
Processing Instructions	每條 Banking 指令執行移除指定元件與插入合併元件操作。	(O(B \times (k \times \log(S) + h \times \log(S))))	(B)：指令數量，(k)：處理的元件數量，(h)：合併元件跨越的行數，(S)：每行的 SubRows 數量。

逐步詳細分析

1. Parsing LG File

- **操作描述：**
  遍歷 LG 檔案逐行，解析每一行的內容並提取數據。
- **時間複雜度：**
  - (n)：檔案行數。
  - 每行操作時間為 (O(1))，總時間複雜度為：

$$O(n)$$

---

2. Initializing SubRows

- **操作描述：**
  為每行初始化完整的 SubRow，並對每個固定元件檢查是否覆蓋，若覆蓋則切割 SubRow。
- **時間複雜度：**
  - (R)：行數。
  - (C)：固定元件數量。
  - 每行需要檢查所有固定元件，總時間複雜度為：

$$O(R \times C)$$

---

3. Removing Cells

- **操作描述：**
  找到指定的 Row 和對應的 SubRow，回收空間，並嘗試合併相鄰的 SubRows。
- **時間複雜度：**
  - (S)：SubRows 數量。
  - 查找 SubRow 使用二分查找，時間為 (O(\log(S)))。合併操作為 (O(1))。
    總時間複雜度為：

$$O(\log(S))$$

---

4. Inserting Cells

- **操作描述：**
  根據元件大小檢查 SubRows 是否有足夠空間，若元件跨越多行，需檢查垂直方向的合法性。放置元件後切割 SubRow。
- **時間複雜度：**
  - (h)：元件跨越的行數。
  - (S)：每行 SubRows 的數量。
  - 每行查找時間為 (O(\log(S)))，多行操作總時間為：

$$O(h \times \log(S))$$

---

5. Processing Banking Instructions

- **操作描述：**
  每條指令分為兩步：移除元件和插入合併元件。
- **時間複雜度：**
  - (B)：指令數量。
  - (k)：每條指令處理的元件數量。
  - (h)：合併元件跨越的行數。
  - (S)：每行 SubRows 的數量。
  - 移除元件與插入合併元件的時間為：

$$O(B \times (k \times \log(S) + h \times \log(S)))$$

總時間複雜度

將所有模組綜合，總時間複雜度為：

$$O(n + R \times C + B \times (k \times \log(S) + h \times \log(S)))$$

其中：

- (n)：檔案行數。
- (R)：行數。
- (C)：固定元件數量。
- (B)：指令數量。
- (S)：每行的 SubRows 數量。
- (k)：指令處理的元件數量。
- (h)：合併元件跨越的行數。

結論

主要瓶頸：

- **SubRows 初始化：**

$(O(R \times C))$ *:* 主要受行數和固定元件數量影響。

- **Banking 指令處理：**

$(O(B \times (k \times \log(S) + h \times \log(S))))$ *:* 受指令數量和跨行元件數量影響。

Special Features of My Program

設計思考與演算法特點

針對提到的問題與解決思路，將設計過程中的重點特徵與優化點分為以下幾部分：

---

1. 偏移成本（Alpha 和 Beta）分析

問題

- 通過觀察 **Alpha** 和 **Beta** 值，發現偏移成本在整體佈局中影響很大。
- 若元件位置偏移過遠，會導致：
  - 信號傳輸距離增加（影響性能）。
  - 線長增加（影響面積與成本）。

解決方法

- 優先將元件插入最接近目標 Row 的位置，儘量減少水平或垂直的偏移距離。
- 利用 **最接近 Row 的行** 優先插入策略，降低偏移帶來的影響。

---

2. Layout 空間利用率觀察

問題

- 整個 Layout 佈局中，發現部分行存在 **大量空間未被利用**。
- 空間未使用的原因可能包括：
  - 固定元件的限制。
  - 插入策略未充分考慮行間空間利用率。

解決方法

- 在元件插入過程中，快速查找空間：
  - 將每行的空間劃分為 **SubRows**，便於追蹤可用區域。
  - 每次插入時，優先選擇最靠近的行，且在該行中找到足夠的空間。
- 動態管理 SubRows，確保未使用空間可以被有效分配。

---

3. 固定元件設計考量

問題

- 考慮到 Layout 的實作難度，某些元件被設計為固定：
  - 固定元件的插入順序可能對整體空間管理帶來干擾。
  - 固定元件導致行間分割，影響 SubRows 的可用性。

解決方法

- 將固定元件優先處理，將其覆蓋的空間從 SubRows 中移除，減少後續插入的影響。
- 動態劃分 SubRows：
  - 固定元件覆蓋的區域被切割，剩餘區域重新計算為可用 SubRows。
  - 確保固定元件的擾動最小化。

---

4. 快速查找空間

問題

- Layout 中可能包含數千個行和 SubRows，若使用暴力搜索，時間複雜度將非常高。
- 必須快速定位空間並檢查元件是否可插入。

解決方法

- 使用 **二分查找** 在 SubRows 中定位適合插入的空間：
  - 每行 SubRows 按照起始位置排序，便於高效查找。
  - 查找時間由 (O(S)) 降為 (O(log(S)))，其中 (S) 是每行 SubRows 的數量。

特點總結表

特徵名稱	問題描述	解決方法
偏移成本優化	Alpha 和 Beta 值偏移成本過大， 影響性能和線長。	優先選擇最接近目標 Row 的行插入，減少偏移距離， 最小化擾動。
Layout 空間利用率優化	行間存在大量未使用空間， 影響布局效率。	動態管理 SubRows， 快速查找空間並高效分配， 確保空間利用率最大化。
固定元件設計	固定元件影響 SubRows 的可用性和後續插入空間的分配。	優先處理固定元件，切割並更新 SubRows， 確保動態空間分配的準確性。
快速查找空間	SubRows 數量龐大， 暴力查找插入空間會導致高時間成本。	使用二分查找定位適合插入的空間， 降低時間複雜度至 (O(log(S)))。

特點與效能提升的結論

1. **優化點清晰**：針對 Alpha/Beta 偏移、未使用空間、固定元件影響等問題，逐步提出解決方案。
2. **高效性設計**：通過快速查找和動態分割 SubRows，實現高效的空間管理與插入策略。
3. **實際可行性**：考慮 Layout 的實作難度，固定元件與動態管理的結合減少了潛在錯誤與計算負擔。

我的演算法特別之處：動態調整移動步伐

功能說明

在處理插入元件的過程中，如果無法直接找到合適的位置，演算法會採用動態調整移動步伐的策略，以提高插入效率和成功率。

核心邏輯包括：

1. 初始移動步伐設定為 `siteWidth`。
2. 如果多次嘗試插入失敗，逐步增加移動步伐，以避免過度微調浪費效能。
3. 插入成功後，將移動步伐恢復至初始值，為後續插入準備。

此策略兼顧了細緻的空間利用與效率提升，特別適用於高度緊湊或空間不足的場景。

程式碼

以下是實作程式碼，包含動態調整移動步伐的核心部分：

```
if (!inserted) {
    attempt++;
    if (attempt >= 2) {
```

```
            // 插入失敗多次後增加移動步伐
            moveStep += siteWidth;
            attempt--;
        }
    } else {
        if (moveStep != originalMoveStep) {
            // 插入成功後恢復移動步伐至原始值
            moveStep = originalMoveStep;
        }
    }
```

程式邏輯分析

1. 動態調整移動步伐的需求背景

當元件無法插入時，通常會透過移動嘗試找到空間。然而，如果移動步伐太小：

- **問題**：效能低下，插入效率大幅下降。
- **原因**：過多無效的嘗試耗費時間。

而採用動態調整步伐，可以：

- **優化效能**：快速跳過不可能的位置。
- **提升插入率**：讓演算法更具彈性，適應不同密度的場景。

2. `attempt` 計數的用途

`attempt` 是用於記錄嘗試插入的次數：

- 當連續兩次插入失敗後，增加步伐 `moveStep`。
- 每次增加步伐後，嘗試重啟。

3. 移動步伐重設機制

- 如果插入成功且步伐已被修改，則將 `moveStep` 恢復至初始值。
- **目的**：避免後續插入因過大的步伐導致效率下降。

4. 效能與穩定性的平衡

- 初始步伐小，能適應密集場景。
- 動態調整步伐，避免過度消耗運算資源。
- **結論**：適應性強，兼顧效能與穩定性。

程式結構表格化

功能	描述
初始步伐設定	移動步伐設為 `siteWidth`，為細緻調整提供基礎。
失敗次數計數	使用 `attempt` 記錄嘗試次數，當達到設定值後進行步伐調整。
步伐調整機制	每次步伐增加一個 `siteWidth`，避免重複無效嘗試。
成功後重設步伐	插入成功後，將步伐重設為初始值，確保後續操作不受影響。
效能與穩定平衡	透過動態步伐與初始步伐切換，適應密集與空曠場景， 提升插入效率與成功率。

心得與報告重點

1. 特點

- **彈性適應性**：能根據插入場景自動調整步伐，避免過度嘗試或錯失合適位置。
- **效能優化**：通過動態步伐增長減少無效運算，特別適用於密集設計的情況。

2. 優勢

- **適用性廣**：可適應不同的空間分佈與設計密度。
- **穩定性強**：透過步伐重設機制，確保演算法在各種場景下保持穩定。

3. 改進建議

- **步伐動態調整策略**：可以根據失敗次數增加非線性增長（如倍增）。
- **記錄與調試功能**：增加每次嘗試的記錄，便於後續優化和分析。

總結

動態調整移動步伐是一種有效提升插入效率與適應性的策略，通過逐步增長步伐與重設機制，平衡了細緻度與效能需求。這種方式既避免了過多的嘗試浪費，也能快速找到合適的插入點，是設計中一個非常實用的優化技術。

Algorithm Architecture

演算法分析與設計思路

1. 問題背景

在 **Placement Legalizer** 中，核心問題是將元件放置到合法的位置，並滿足以下條件：

- **空間合法性：** 確保元件不重疊，且不越界。
- **擾動最小化：** 優先考慮元件初始位置附近的空間，減少偏移距離。
- **效率：** 在大量元件和行數（Rows）的情況下，快速完成放置操作。

2. 參考演算法

演算法	核心特性	適用場景	優劣勢
Abacus	使用動態行內插入策略， 通過行優先策略將元件插入最近的合法位置。	元件主要沿水平方向擺放， 且需減少水平偏移的情況。	**優勢：** 簡單易實現， 擾動成本低；**劣勢：** 無法處理多行高度的元件。
Tetris	模擬堆疊的方式，將元件依次放置到空間中， 適合垂直擺放或多行高度元件的情況。	垂直方向空間利用率優化、 多行高度元件擺放。	**優勢：** 空間利用率高； **劣勢：** 偏移成本可能較高， 無法控制擾動距離。
區塊佈局	將元件分割為區塊， 根據區域優先順序分配元件， 通常適用於大規模元件的全局放置。	面積分佈均勻的大規模元件佈局， 如 IC 設計中的區域化佈局。	**優勢：** 可處理大規模元件；**劣勢：** 本地擺放效率低， 需全局計算。

---

3. 自定義演算法設計

在設計中，結合上述演算法的特性進行整合，具體設計思路如下：

1. **動態行內插入：**
    - 參考 **Abacus**，使用動態 SubRow 管理空間，快速查找最靠近的合法位置。
    - 解決了水平擾動成本過高的問題，適合初始位置偏移小的元件。
2. **多行高度支持：**
    - 借鑑 **Tetris** 的堆疊策略，對於跨越多行的元件，檢查垂直方向的合法性。
    - 解決 Abacus 在多行元件支持不足的問題，提升空間利用率。
3. **分塊優化：**

- 使用區塊佈局的思想，將固定元件優先分配到特定區域，並將剩餘空間進行子區域分割。
- 確保固定元件與可移動元件的分配不互相干擾。

---

4. 比較分析與整合

演算法特性	Abacus	Tetris	區塊佈局	自定義設計
擾動最小化	優先放置在初始位置附近， 擾動成本低。	偏移較大， 垂直堆疊優化可能增加偏移成本。	需要全局計算， 局部偏移可能過大。	動態行內插入， 優先放置最靠近初始位
空間利用率	水平空間利用率高， 但不支持多行高度。	垂直方向空間利用率高， 但水平空間利用率低。	全局空間利用率均衡。	結合動態行內插入與多 兼顧水平與垂直方向的
多行元件支持	不支持多行元件。	支持多行高度， 垂直方向合法性檢查。	支持多行分佈， 但成本高。	支持多行元件， 並通過垂直方向的合法
實現複雜度	簡單，實現成本低。	較簡單，適合局部優化場景。	複雜， 適合全局優化場景。	複雜度介於 Abacus 與 兼顧局部效率與全局合
適用場景	適合初始偏移小， 元件主要沿水平方向擺放。	適合垂直方向優化， 或多行元件的情況。	適合大規模全局佈局。	適合大多數實際 Layou 尤其是在固定元件與多 提升放置效率與空間利

---

5. 優劣勢分析

優勢

- **擾動成本控制良好：** 結合 Abacus 的行內插入策略，優先考慮最靠近初始位置的合法空間。
- **多行元件支持：** 借鑑 Tetris，實現對多行高度元件的支持，提升空間利用率。
- **靈活性強：** 支持固定元件與可移動元件的混合佈局，能應對更複雜的 Layout 場景。

劣勢

- **實現複雜度較高：** 動態管理 SubRows，且需對多行元件進行合法性檢查，導致程式結構較為複雜。
- **全局優化不足：** 雖然參考了區塊佈局，但仍以局部優化為主，缺乏全局放置的全面考量。

結論

1. **設計整合：** 結合了 Abacus、Tetris 和區塊佈局的特點，針對性地解決了水平擾動、多行元件支持以及固定元件的佈局問題。
2. **適用場景：** 最適合在固定元件與多行元件共存的 Layout 場景中，優化局部放置效率，同時提升空間利用率。
3. **改進方向：** 可考慮增加全局優化策略，進一步降低總線長或提升分佈均勻性。

表格總結

表格結構清晰地展示了參考演算法與自定義演算法的比較，幫助理解設計的背景、優劣勢以及改進方向。直接適合用於 Markdown 編輯器展示或報告撰寫。

以下是補充後的完整心得，結合您的經驗和反思，更加條理清晰，適合直接用於報告中。

心得

1. 最大挑戰：Multi-row 元件插入

這次作業中，我認為最困難的部分是 **multi-row 元件的插入**。雖然直觀上這是一個相對簡單的問題，但在實作時遇到了許多限制：

- **垂直合法性檢查**：需要確認元件跨越的所有行是否有足夠空間，同時還要兼顧水平方向的空間連續性。
- **空間管理**：每行的 SubRow 需要動態更新，且必須確保分割與合併的正確性，否則會導致後續插入錯誤。
- **缺乏常規演算法**：multi-row 插入並沒有現成的通用解法，特別是對於這類 NP 問題，既要滿足合法性又要使 cost 最小，增加了設計和實作的難度。

這些挑戰讓我深刻體會到，多行高度元件的插入雖然理論上簡單，但實際上需要考慮的細節非常多。

---

2. 學習與收穫

在解決問題的過程中，我查閱了許多相關的研究與論文，特別是與 **放置成本最小化** 和 **NP 問題優化** 相關的內容，從中獲益良多：

- **放置策略的啟發**：許多論文提供了動態調整和成本優化的思路，例如如何在有限的空間中快速找到合法插入點，以及如何通過局部調整降低擾動成本。
- **演算法的整合**：我學到了如何結合不同的演算法，比如將 **Abacus** 的簡單插入策略與 **Tetris** 的垂直擺放結合，既解決了多行高度問題，也提升了空間利用率。
- **實作能力的提升**：從理論設計到程式實作，經歷了多次失敗與調整後，我對動態資料結構的使用（如 SubRow 管理）有了更深刻的理解。

---

3. 收穫

雖然在實作過程中，排不進去有點挫折：

- **程式錯誤排查耗時**：SubRow 的分割與合併過程中出現過多次邏輯錯誤，導致反覆測試調整。
- **設計優化反覆修改**：為了使 cost 最小化，試了多種策略後仍然無法達到預期效果。

但當最後測資成功通過的那一刻，我感到非常有成就感。這次作業不僅幫助我學會了解決複雜問題的方法，也讓我深刻體會到，即使過程艱辛，結果往往值得努力。

---

4. 總結

這次的作業經歷讓我對 **Placement Legalizer** 的問題有了全新的認識：

1. **理論與實踐的差距**：理論上看似簡單的問題，在實作中可能面臨意想不到的挑戰。
2. **持續學習的重要性**：通過查閱論文和學習他人的解法，我能更好地應對複雜的問題。
3. **成就感來自解決困難**：當困難被逐一攻克時，成果的價值也會更加讓人滿意。

雖然這次的作業很有挑戰性，但它讓我學到了很多關於 **演算法設計** 和 **問題優化** 的知識，也提升了我面對實際問題時的解決能力。

補充建議

1. 若有更多時間，我希望能進一步研究如何將全局優化與局部優化結合，進一步降低成本。
2. 未來若可能，嘗試引入 **機器學習** 方法，用於預測最佳插入策略，提升效率。

此心得條理清晰，既有對挑戰的分析，也有對學習的反思，最後以建議作為收尾，展示出對未來改進的思考。