DSnP Final Report in Fraig Project

R08921053
r08921053@ntu.edu.tw

1. 設計的資料結構與演算法

a. sweep

首先，我會把所有的 gate 都存入一個 vector 容器, gateList 內. 在做完 dfs 以後，開始清洗 gateList 的所有 gate. 此時，如果有 gate 沒被經過，還是乾淨的，那麼就確定是未使用的 gate. 如果它不是 input gate，就直接把它給移除掉. 如果它是 input gate，由於所以跟它相連的 gate 都被移除光了，否則他就不會是未使用的 gate 了，所以就把它設定成 undefined gate.

基本上，就是順著 dfs 做完後的髒資料結構，一邊清洗一邊處理. 順便把沒經過沒用到的 gate 結構給移除就是了.

至於時間複雜度分析的話，就是走遍一次 gateList，或是拜訪過所有 gate 的時間複雜度，而每個 gate 花費 O(1)time，所以總共是 O(n), n 為 gate 數目.

---

b. optimize

基本上，在這裡就是用 dfs 的方式來一一拜訪所有可到達的 gate. 值得一提的是，我採用一個有點巧思的三段式方法融入 dfs()，借用到 Lazy 演算法概念.

- 更新:

在拜訪完小孩後，馬上檢查小孩的 type 是否被改成'r', 'l'，或'0'. 如果證實被更改了，那麼就馬上把自己儲存的小孩資訊更新成指定的對象. 也就是更新.

- 標示:

在 dfs()拜訪到自己時，如果要移除自己這個 gate，就把 type 改成'r', 'l'或'0'. 也就是合併為右小孩，左小孩，或是 Const 0 gate 的代號.

- 移除:

最後，當我做完整體的 dfs 後，跑遍裝有所有的 gate 的容器—gateList. 一方面，如果 type 沒被標示，就清洗資料結構. 如果 type 有被標示，就刪除該 gate. 此外，我們只需要檢查 dfs list 裡面是否有"拜訪次數=0"的 gate，如果有，那麼他們就是 opitmize 的犧牲者，要被加入 un-used 容器中.

從時間複雜度的角度來看，前兩步基本上就是在 dfs 的過程中順便做，每個 gate 頂多 O(1)時間. 而最後移除的動作也是跑遍所有 gate 就好，所以時間介於 n 到 2n 之間，其中 n 為 gate 數目. 所以是 O(n) 複雜度.

---

c. Strash

在這裡，很顯然的我們要使用到作業七的 MyHashSet 工作．流程是:

Aig gate->gateId->TaskNode->HashSet->afford group in each bucket

換言之，我們利用 HashSet 來分類所有的 Aig gate，並且假設沒有錯誤碰撞產生，認定同一個箱子內的 gate 就是具有同樣 fanin 的 gate，接下來再把他們合併.

而接下來的工作才是難關—如何有效率的合併 gate，並且更新他們父親的資訊．比方說 z 連接 a,b; y 連接 a, c．假設 b, c 同屬一群，被合併後要馬上通知 y,z 讓他們也跟著合併.

在此，我借用並修改了 optimize 的巧思演算法，也就是採用三段式方法:

- 標示:

首先，在開始 dfs()前，先把 Hashset 內，檢查超過 1 個元素的 bucket，把裡面的元素的 type 都標示成's'，以及在"strash"的元素空間內存入該 bucket 第一個 gate 的 Id.

- 更新:

在拜訪完小孩後，假查小孩的 type 是否有被更改，如果有就按照小孩的"strash"元素來把小孩資訊做更新.

- 再標示:

由於小孩更新後，父母的 bucket 有可能隨著變動，所以我們馬上把自己從 HashSet 中移除，再馬上插入 HashSet．如果原先 type 就是 s，要看看新的 bucket 領頭羊是否有換對象了，要小心原先就是領頭羊的 corner case．如果原先 type 不是 s，卻而後被插入超過 1 個元素的 bucket，那麼就把 type 換成 s，更新 strash.

- 移除:

最後，再做完 dfs()之後，拜訪過所有 gate 的容器—gateList，再把被標示 type 的 gate 移除即可．如果沒被標示的 gate 就清洗資料結構.

在時間複雜度部分，開頭放入 HashSet 的階段需要 O(n)，一一通知 gate 他們發生碰撞的時間最多也是 O(n)，而標示/更新/再標示都是在 dfs()內發生的，每次 O(1)，所以整體也是 O(n)．最後，跑遍所有 gate 做清洗和移除的動作也是 O(n)．雖然常數應該比較大，但仍然是 O(n)的複雜度.

---

d. Simulation

在這個部分，基本上我們還是利用 HashSet 的方式來分類 gate，把同一個 bucket 內的都當成是同一群 FEC group．基本上每輸入一組 input，也就是每個 input gate 的值是 0 或 1 時, HashSet 的大小，也就是 bucket 數目會增加成兩倍．而我們同樣使用 dfs()的結構，在獲取小孩的值後，與 invPhase 一起做加法，就可得

到自己的值，也就是 Hash 值．這樣就可以走完 dfs()．流程如下:

獲取小孩的值是 0/1->搭配 invPhase 做計算->得到 Hash 值.

最後，在做完 dfs()後．每個 bucket 都分成兩個新的 bucket．如果 Hash 值是 0 的就放入偶數的 bucket；如果 Hash 值是 1 的舊放入奇數的 bucket．然而如果 bucket 的數目是 1 就可以把資料給捨棄了．那麼，既然我們有所有 gate 的 Hash 值，而且 HashSet 當中的分類和大小也確定了，基本上就完成了 Simulation.

關於時間複雜度部分，每一次給的 sim 值，都需要跑完一次 dfs()後才能夠消化，並且得到每個 gate 的 Hash 值，這部分是 O(n)複雜度．而且得到 Hash 值以後，還要移到新的兩倍大 Hashset，這個部分則是 2n=O(n)複雜度．所以每一組 sim 值都需要 O(n)的時間複雜度，常數大約比 strash 小一點或持平，然而需要做很多次．所以基本上是 O(nm)，其中 n 是 gate 數目, m 為 sim 組的數目.

---

e. FRAIG

我們規定要做完 simulation 才能做這步驟．所以目前已經有一組一組的 FEC group，也就是 equivalent gate 的候選人了．在這裡，我們一次針對一組 FEC group 做運算．每次把 gate 編號成 SAT 的 variable ID，讓他們符合 SAT 工具的格式．接下來使用 SAT 工具，經過一些具體步驟與已經提供的程式碼，我們可以得到他們是/否等價的答案，在此不考慮耗時過長的特殊例子．如果兩個 gate 確定相等，那麼就把它們合併.

值得一提的是，由於孩子合併的結果會影響到父母驗證的困難度，所以我們會採用 dfs()，先檢查完孩子的驗證與合併後，再檢查父母的驗證與合併，演算法類似於前方的三段式方法—檢查孩子的 type，自己要合併則更改 type，最後在走遍所有的 gate—gateList．一邊清洗資料結構，一邊移除被合併掉的 gate.

在時間複雜度上面，顯然每一次的驗證都是 NP-Complete 的問題，而且兩兩一組在 FEC group 裡面的 gate 可能都需要驗證，所以自然是 NP 的時間複雜度.

f. 表格整理

在下列的表格中，以 n 表示所有 gate 的數目(n~I+O+A).

	大致演算法	時間複雜度
Sweep	After DFS, go through all gate and check used/not	O(n),
Optimize	In DFS, update children's value then check ourself. After DFS, go through all gates and remove data.	O(n), 大約是 Sweep 的兩倍
Strash	Before DFS, transfer gate into HashSet and clarify. In DFS, update children's value then update our value and check ourself. After DFS, go through all gates and remove data.	O(n), 大約是 Sweep 的五倍
Simulation	Given a group of Sim val. In DFS, update children's value and count ourself. After DFS, move data into new HashSet.	[每一組 Simulation] O(n), 大約是 Sweep 的三倍.
FRAIG	Given FEC result. In DFS, verify children's situation and update, then verify on ourself. After DFS, go through all gates and remove data.	[每兩個 FEC 內的 gate] NP-Complete Problem

2. the results and analysis

基本上比照我的程式與範例的程式，針對 sweep, optimize, strash 等三個功能.

i. 以 sim13.aag，有 88410 個 gate 為例子

	Mine	Reference
Sweep	0.17s/15.37MB	>0.01s/13.99MB
Optimize	0.04s/15.37MB	0.01s/14.85MB
Strash	crash	0.01s/17.69MB

- 比較:

首先，在 Sweep 部分，我花費的時間蠻高的，可能是因為要當場做一次 DFS，才能再一一檢查是否有用到的緣故. 或許範例檔的實作方式，是跳過 DFS 的過程，直接從檢查開始做起. 但我為了警慎起見，還是保留這樣的做法.

再來, Optimize 部分，基本上兩個檔案都幾乎沒有花到時間. 而且我的檔案反而幾乎沒使用到多餘的記憶體，但老師的檔案卻多了一些. 或許是我在最初就把所有用到的資料結構宣告在 CirGate 上，然而老師卻到這一步才宣告資料結構吧.

最後，在 strash 部分，由於我的 HashSet 大小設定成 getHashSize(23*M)，所以我猜測 88410*23 可能是個過於龐大的數字，因而導致了 crash. 而老師的程式還是幾乎沒花到時間，但記憶體則是花了不少，想必是花費在宣告 HashSet 的部分.

- 反思:

值得反思的一點是，違背我們一開始所分析的, optimize 的運算速度意外地比 Sweep 還要快上很多，幾乎只有 25%的時間. 仔細思考後是表達符號上的缺陷:

1. Gate 數目會隨著演算法而減少
2. Gate 數目又分成所有的 Gate 數目 v.s. DFS 走遍的 Gate 數目

更具體的來說，一方面 Sweep 已經減少了一些的 Gate，一方面 Optimize 又可以再合併，減少需要拜訪的 gate 數目，另一方面 undefined 的 input gate 也會造成誤差. 因此，在這個議題上，理論分析與實作結果有所差異.

ii. 以 sim14.aag，有 928 個 gate 為例子

	Mine	Reference
Sweep	0s/0.4023MB	0s/0.4023MB
Optimize	0s/0.4023MB	0s/0.4023MB
Strash	0.07s/0.7773MB	0s/0.4023MB

- 比較:

首先，我和範例檔在一開始都沒花到時間，而且記憶體使用上是相同的. 我猜測應該就是基本的構造資料結構的花費等等，沒有太多的變化. 然而，值得注意的是，我在 strash 步驟花費的不少的時間和記憶體，而且也刪除/合併了數十個的 gate，不過範例檔卻一個都沒有刪除/合併. 這顯現出我的 strash 另外一個問題—

Hash 值生成的過程不夠隨機，導致碰撞機率太高，失誤碰撞的情況連連發生，才會讓不需要合併的電路圖也合併了數十個 gate.

- 反思:

從實驗的結果也確實可以看出, HashSet 的分群方式雖然非常快速，甚至可以到達 O(n)的時間複雜度，但是如果操控不好，設計的不夠精確，不小心發生問題，卻可能得到錯誤的答案，甚至是浪費了很多的時間和資源在上面. 所以，每一種演算法都有它的好處，優勢，以及所需要付出的代價和犧牲的一面.