EE6094

CAD for VLSI Design

Programming Assignment I Report

Student Name:李庭緯

Student ID:106303010

零、Introduction

[作業描述](#作業描述)

[How to execute&成果](#How_to_execute成果)

1. [檔案結構](#檔案結構)
2. [Lib功能&資料結構](#LIB功能)
3. [DFG\_node](#DFG_node)
4. [DFG](#DFG)
5. [Schler\_prototype](#Schler_prototype)
6. [UC\_Schler](#UC_Schler)
7. [List\_Schler](#List_Schler)
8. [演算法Revisted](#演算法Revisted)
9. [遇到的困難](#遇到的困難)
10. [Bonus](#Bonus)
11. [實驗](#實驗)
12. Decode對時間的影響
13. ML\_RCS/MR\_LCS的不可逆性

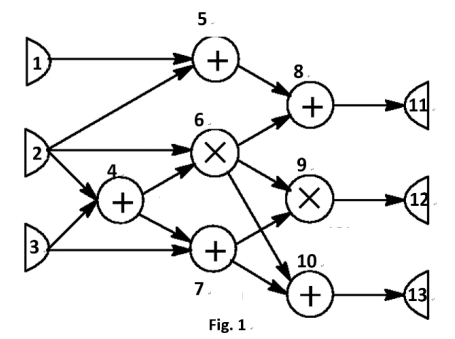
作業描述:

本次作業是要將DFG(data flow graph)/硬體的delay/硬體的resource

當作input輸入，並根據resource constraint的狀況下將latency盡可能的減少。

輸出的格式:

1. 得出的latency結果
2. 每一個時間被schedule的node清單



以上圖DFG為例，

輸入的硬體

1. 型態為[+,\*]
2. 數量為[1,1]
3. delay為[1,3]

經過我使用的List scheduling演算法得出的結果為

7

4

6 7

5 6

6

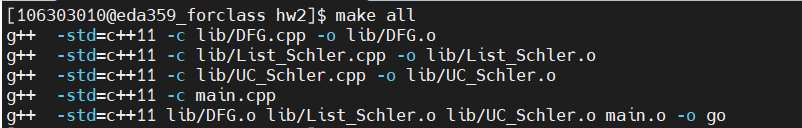
8 9

9 10

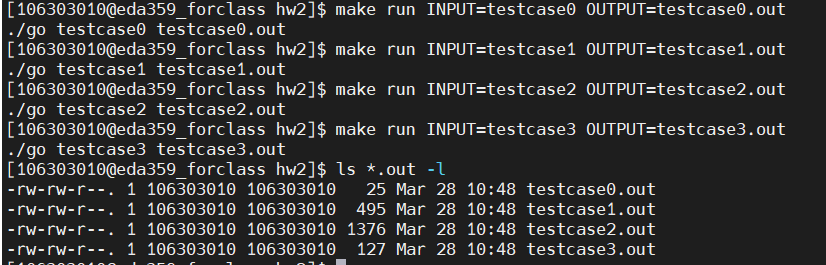
9

How to execute&成果

1. Make all



1. 產生.out



1. Testcase0 (PA2範例) 1/2/3



一、檔案結構

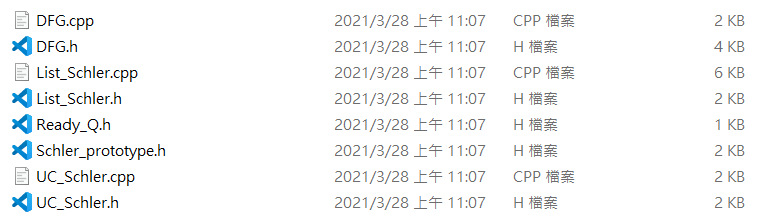
1. Lib:

H\_FILE : DFG.h/Schler\_prototype.h/UC\_Schler.h/List\_Schler.h/Read\_Q.h

CPP\_FILE : DFG.cpp/UC\_Schler.cpp/List\_Schler.cpp

1. TestUnit: List\_Schler\_ML\_RCS\_MR\_LCS/ UC\_Schler\_ASAP\_ALAP
2. Makefile&main.cpp

Lib:



TestUnit:



本Lib共實作四種演算法base on DFG

包含UC\_Schler當中的

1. ASAP
2. ALAP

以及包含List\_Schler當中的

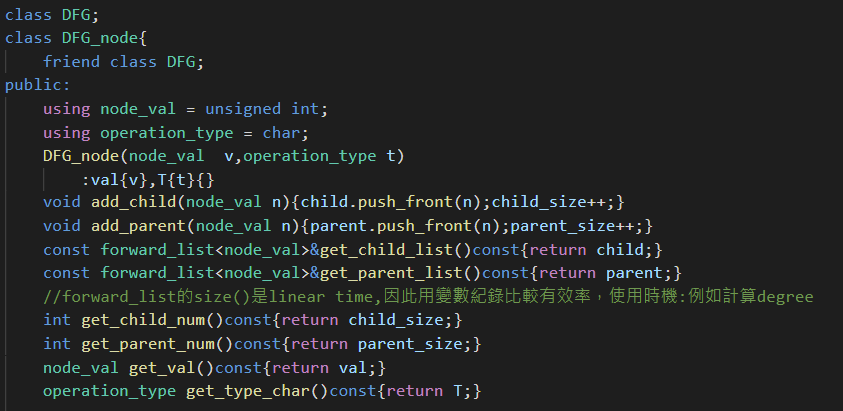
1. ML\_RCS
2. MR\_LCS

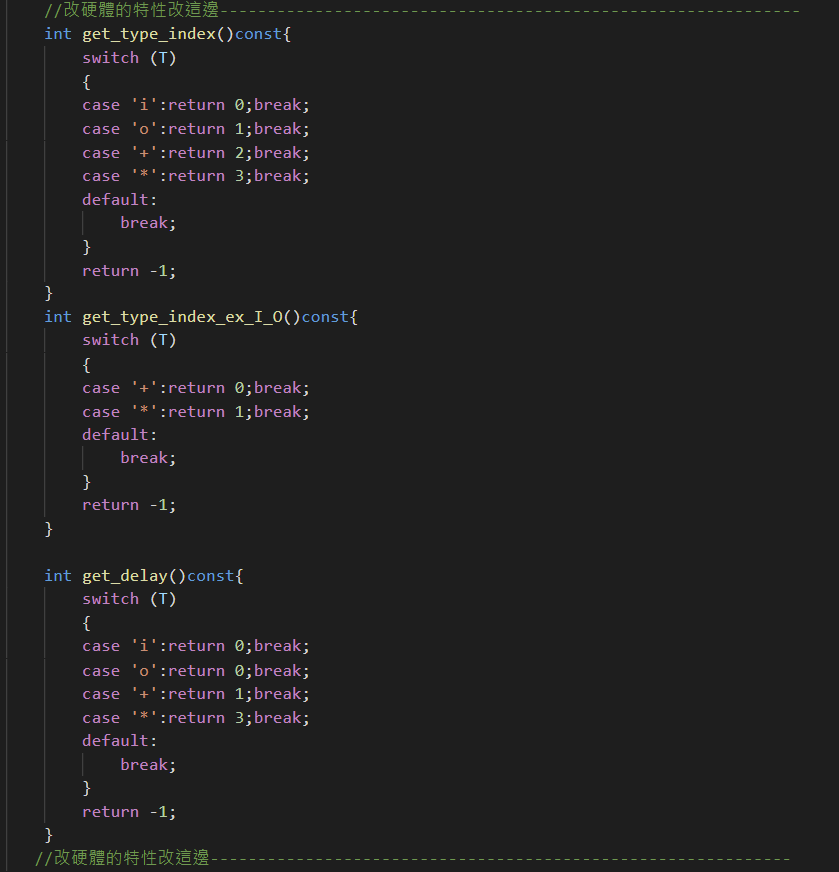
本文將詳細介紹

1. DFG\_node的實作
2. DFG 如何處理不照順序輸入的DFG\_val
3. Schler\_prototype
4. UC\_Schler 及ASAP/ALAP
5. List\_Schler 及ML\_RCS/ ML\_RCS

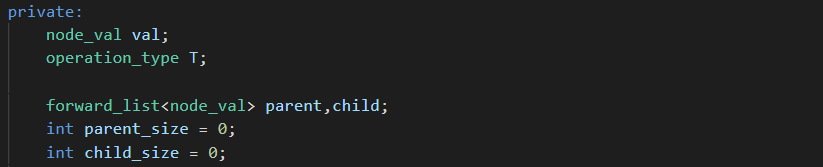
二、各Lib功能介紹&資料結構

1. DFG\_node





(更改硬體特性改上面)



主要data member:

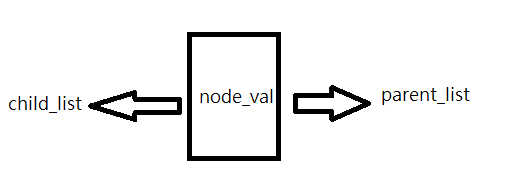
1. Val (型別為node\_val) :

用以紀錄node的數值

(強調數值,因為之後是透過index跟val之間轉換達到快速存取)

1. T(型別為Operation\_type)
2. Parent,child (forward\_list):使用單向list實作
3. Parent\_size/child\_size :因為**forward\_list取得size是線性時間**，因此我決定額外花費一點記憶體代價去紀錄長度。

DFG\_node示意圖



2. DFG



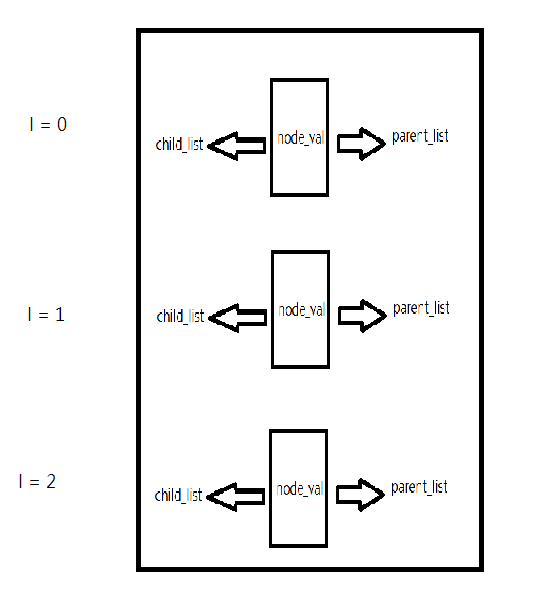
主要data\_member:

1. V : 用來存放整張圖的node val
2. Decode : **用來記錄不同的node val對應到V當中的index,可以達到不照順序輸入node val**

核心功能(Method):

1. 讀取INPUT時會用到的
2. add\_node
3. add\_edge
4. debug用的show\_DFG
5. ASAP/ML\_RCS/MR\_LCS實作時需要快速的get\_input\_index
6. ALAP 實作時需要快速的get\_Output\_index
7. Schler\_protype設定delay時需要的get\_node\_vector
8. get\_index : 因為存在DFG\_node當中parent/child存的是node\_val而不是node\_index,因此無法直接指向V,需要先透過Decode將node\_val轉成node\_index.

DFG當中的V示意圖



val轉換index解釋:

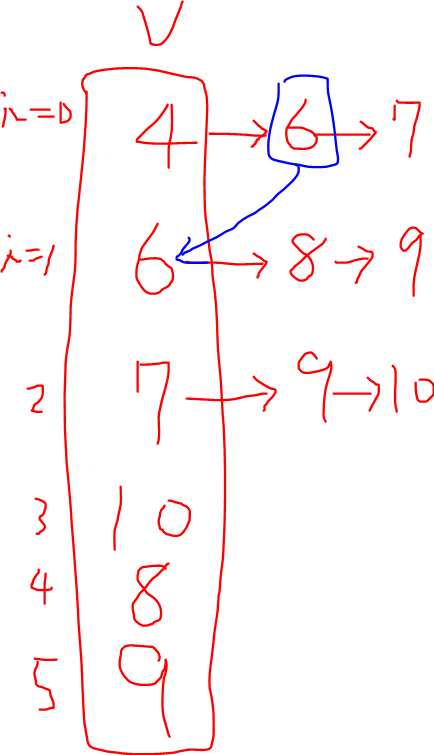
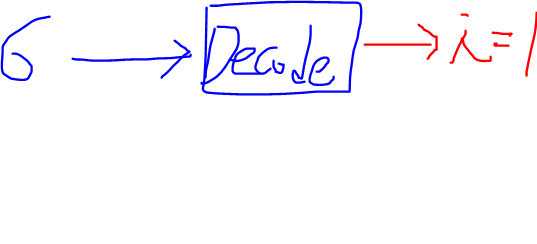
因為在做traversal的時候會去檢查child\_list或parent\_list

但list存的是node\_val而不是V當中的index,為了去成功尋訪V當中的對應node，我們需要val轉index的方法。

我使用的是Decode,型別為map(紅黑樹)。

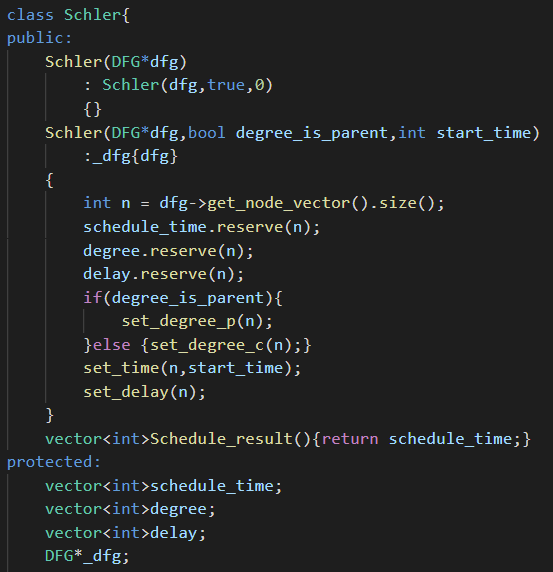
假設V當中的NODE為下面這樣(我省略parent)

4去檢查自己的child 6時必須找到真正6在V當中的位置(藍色框框)

**不過後來我發現測資好像都照順序排(1開始到n結束)**，我就把這個功能關掉了，後面會做實驗比較decode跟沒decode的時間。

3. Schler\_prototype



觀察到那幾種Scheduler都會需要

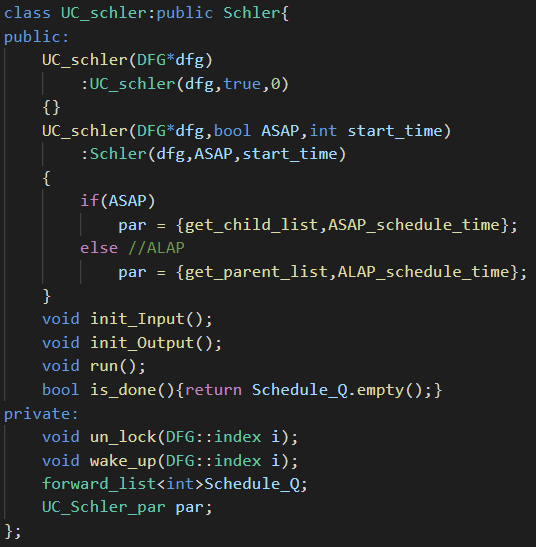
1. schedule\_time
2. degree
3. delay
4. dfg

等四種資料來做排程，因此封裝在一起，之後UC\_Schler(ASAP/ALAP)以及List\_Schler(ML\_RCS/MR\_LCS)可以透過繼承來減少代碼的重複性，也更加好維護。

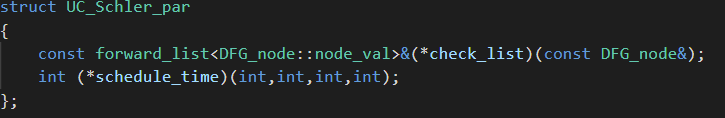
主要method:

1. 建構子:用來設定schedule\_time /degree/delay。
2. 回傳排程結果

4. UC\_Schler (ASAP/ALAP)



以及設定不同的參數(針對ASAP/ALAP)



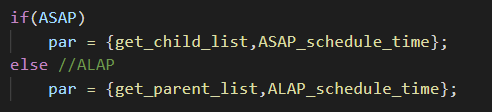
主要data\_memer:

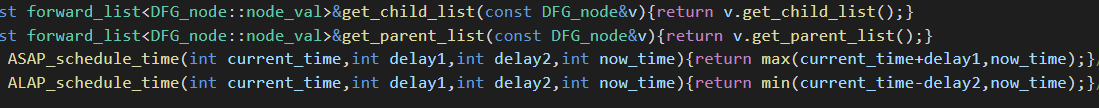
1. Schedule\_Q: 用來做排程的QUEUE
2. Par:設定不同的參數(針對ASAP/ALAP)

例如在跑ASAP需要檢查的degree是往parent檢查

在跑ALAP時需要檢查的degree則是往child檢查

(這邊就是用到DFG\_node裡面的get\_child\_list/get\_parent\_list)





以及不同的時間計算方式:

ASAP的每個節點排程時間為:MAX(父節點+父delay)

ALAP的每個節點排程時間為:MIN(子節點時間-自己delay)

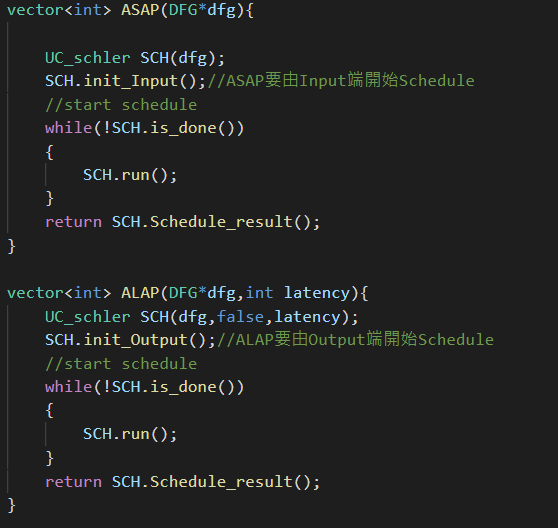
主要method:

1. Init\_Input():ASAP的排程由INPUT端開始

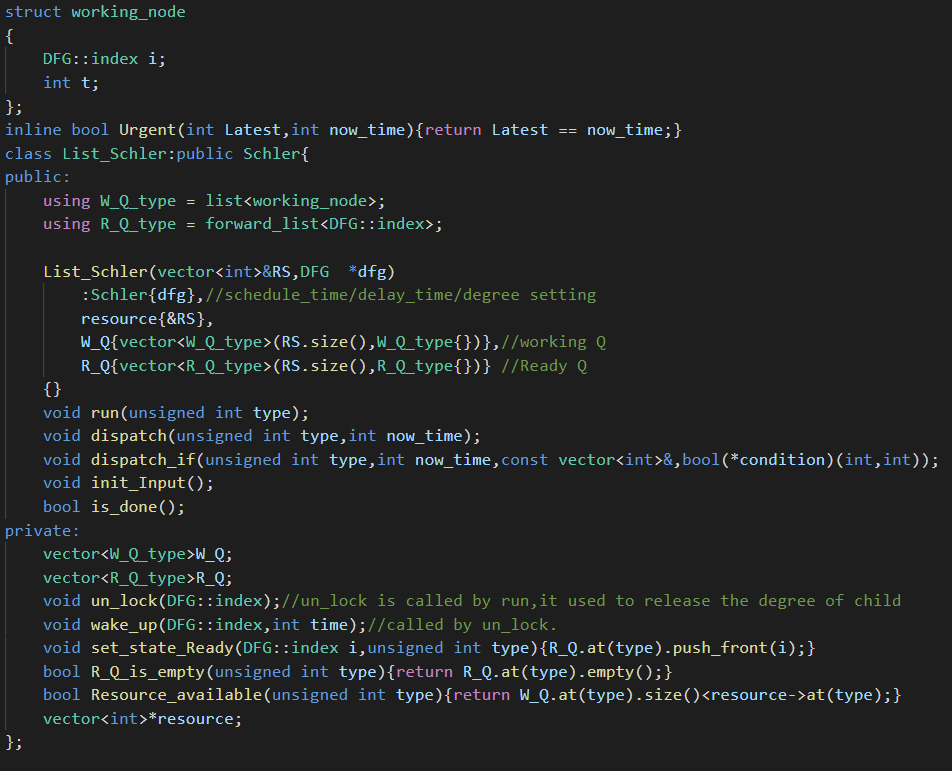
(剛剛DFG當中的get\_input\_index就可以快速使用)

1. Init\_Output():ALAP的排程由OUTPUT端開始
2. (剛剛DFG當中的get\_output\_index就可以快速使用)
3. Run : 開始排程
4. Is\_done : 排程是否完成
5. Un\_lock : 會被run呼叫，用來對child/parent做degree釋放(un\_lock)
6. Wake\_up : 會被Un\_lock呼叫，如果degree扣除後剛好為0,將叫醒他並放入Q當中。

**ASAP/ALAP實作**



5. List\_Schler (ML\_RCS/MR\_LCS)



主要data\_memer:

1. W\_Q:working Queue : 還在進行操作的operation會被放在此
2. R\_Q:Ready Queue:degree為0,等待被放入W\_Q的operation
3. Resource:用來記錄/調整硬體資源

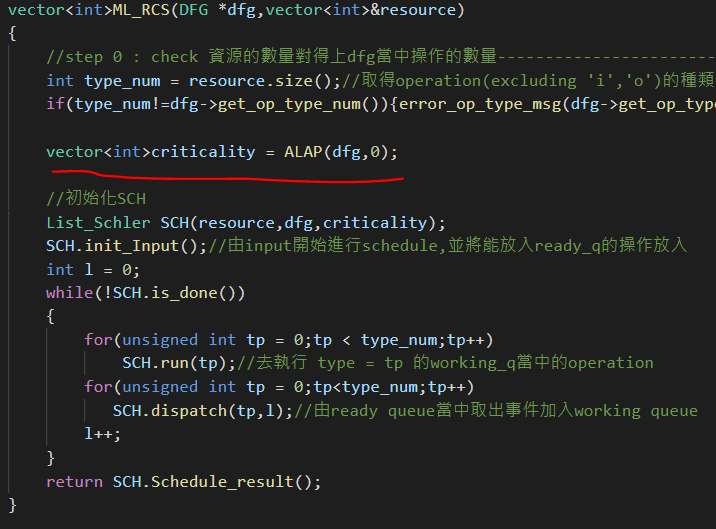
主要method:

1. Run : 負責執行W\_Q內的operation,並若該operation完成(delay時間結束)

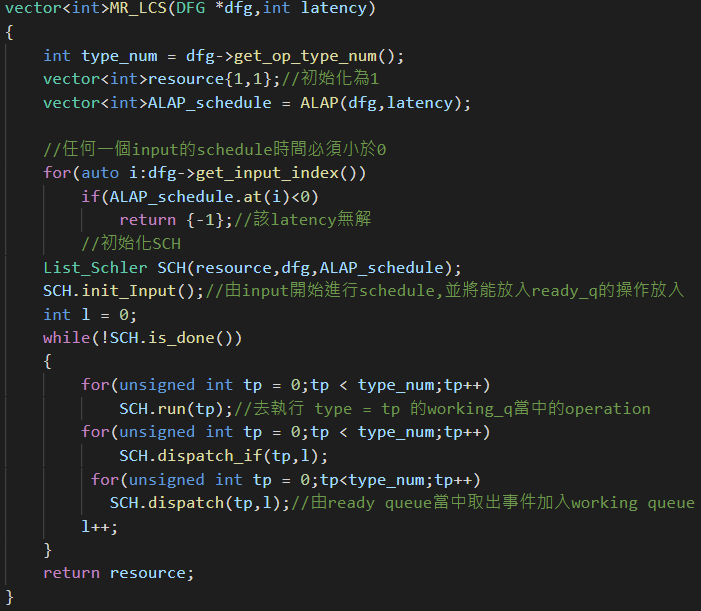
會呼叫un\_lock,釋放對child的資源。

1. Dispatch:由R\_Q當中挑選operation放入W\_Q的動作
2. Dispath\_if:若是urgernt(slack = 0),則調整resource並將op由R\_Q放入W\_Q
3. Init\_Input():由input開始進行Schedule
4. Is\_done:檢查所有output是否被schedule,若是則回傳true

ML\_RCS實作(用ALAP的結果當criticality(ALAP出來值越大代表離OUTPUT越遠)



MR\_LCS實作(多了一個dispatch\_if)



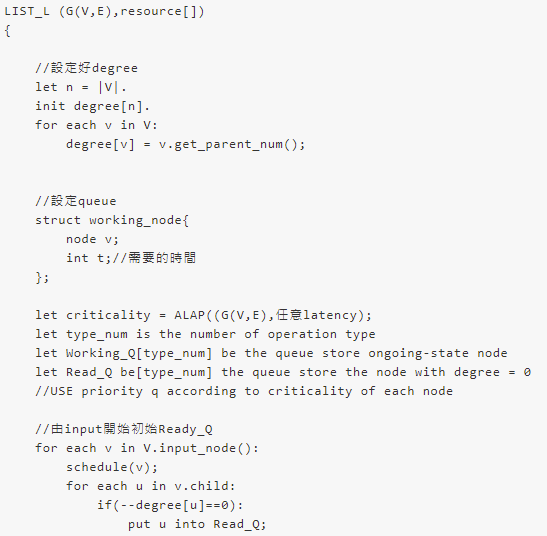
三、演算法Revisted

使用到的演算法是List Scheduling Algorithm: **ML-RCS**。

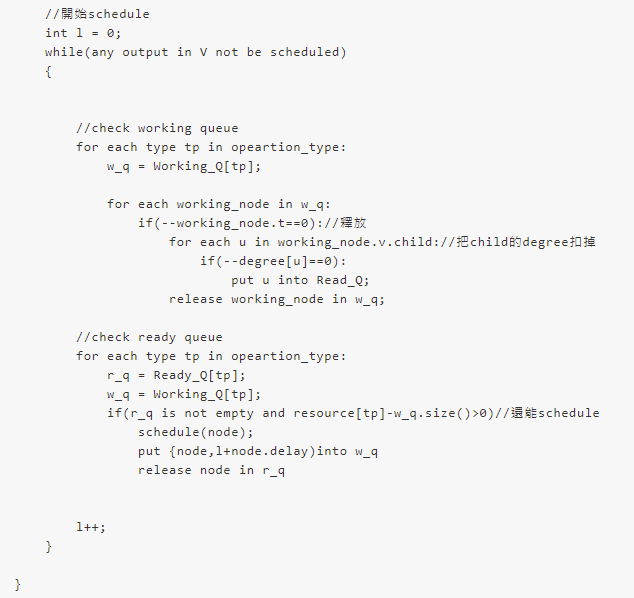
因為講義的pseudo code省略許多，我自己版本的pseudo code如下

1. 前置作業:

跑ALAP(criticality)/產生Ready\_Q/Working\_Q以及degree設定



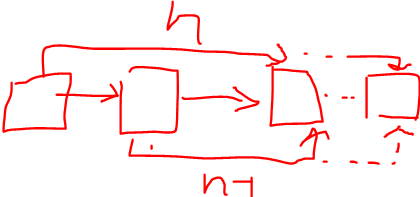
1. Schedule流程(比講義的pseudo code多了working\_q的run過程)



時間複雜度分析:

定義n = |V|為DFG\_node個數。

定義m = |E|為DFG邊數,DFG為DAG,分析



最大邊數為N+N-1+N-2+…+1 = O(n^2)

1. 前置作業:

ALAP/產生Ready\_Q/Working\_Q以及degree設定 : **O(n+m)**

1. Schedule
2. Run():虛擬碼中working\_q檢查的部分
3. 對於每個op的工作: working\_node.t –=1 :

時間複雜度為所有的node的delay總和 :**O(K\*n),**

(K理論上可視為常數)

1. check所有的child:

O(m)\*查到child在V的真實位置

我一開始看作業要求:有敘述說NODE\_NUM不會照順序輸入，所以我設計了decode去轉真實index，但是我後來看測資發現都是按照順序(1開始)輸入，因此我用Deocde(紅黑樹)去查找(O(lgn)會比較吃虧，我後面會將Decode功能開啟以及關閉來做時間上的測試。

**O(m) \* 查找時間。**

1. 放入所有DFG\_node進Ready\_Q :

Ready\_Q實作方式為STL:priority\_queue,因此插入是O(lgn)

時間複雜度為**O(nlgn)**

1. 釋放working\_node

因為實作上不應該在遍歷容器的時候進行刪除。

所以我利用一個vector回收那些iterator,遍歷完後才刪除

因為STL的forward\_list沒有提供可以操作內部Node->next的介面，因此只有remove可以使用(查找到相同值後刪除，需要時間複雜度為O(list長度)，因此我使用stl:list(雙向list)並使用erase(O(1))。

**所以時間複雜度為O(n).**

這邊會選擇用STL來完成實作是因為我不想再額外去寫linked\_list的操作，操作本身並不難寫，但是要自己做記憶體管理，我認為單純用head的架構去做linked\_list會有管理記憶體的問題(dangling pointer)，勢必需要容器去負責管理這些NODE，但我覺得太麻煩了,因為還要處理解構子的部分。不如使用STL這樣簡便的介面。

不過後續使用STL的時候才發現,

如果對於效率錙銖必較，或是彈性有特別的需求

那就得自己寫了。

總結:

1. 前置作業(ALAP/設定degreee) 的複雜度為**O(m+n)**

2. Run的時間複雜度為

1.使用Decode

O(K\*n)+ O(m)\*O(lgn)+O(nlgn)+O(n) **= O((m+n)lgn) + O(Kn)**

**2.不使用Decode**

O(K\*n)+ O(m)\*O(1)+O(nlgn)+O(n) **= O(m+nlgn) + O(Kn)**

3.Dispatch():虛擬碼中Ready\_q檢查的部分

O(n) \* release的時間(priority\_q的pop:logn) = **O(nlgn)**

分析整體複雜度

保留decode功能拿掉:整體複雜度為(詳細實驗)

**O(m+n)+O((m+n)lgn) + O(Kn)+ O(nlgn)**

將decode功能:整體複雜度為

**O(m+n)+O(m+nlgn) + O(Kn)+ O(nlgn)**

四、遇到的困難

困難一: 架構設計

第一次練習開發比較大型的Lib,在設計程式架構的過程中一直思考如何能讓Lib更好維護，重用性高。

我在這一部分花了蠻久的時間，直到後來想到應該從這些排程器的行為來思考，排程器不過就run/dispatch兩個模式，run的行為就是將Working\_Q內的操作執行，dispath就是將ReadyQ內的操作派發給Working\_Q，而MR\_LCS則多了一個urgernt狀態(slack=0不得不派發)，在想通這些行為後就讓程式碼更加簡潔且分離實作，讓Lib變得更好維護。

不過自身程度仍不足，感覺有些寫法還可以更加乾淨

，可能需要再去補足C++的泛型以及lambda等語法。

另外就是編譯器跟工作站的不同，我在建構子的initialize list當中

初始化參考型態的Resource。這樣的寫法在工作站無法使用，因此只好改成指標型態，這讓我發現，如果不使用標準c++，在不同的編譯器下就可能出錯。

最後是版本控制/Lib切割仍不太熟，可能需要多看看別人是怎麼將檔案結構設計好的。

困難二:

MR\_LCS跟ML\_RCS是不可逆的。

我本來以為將ML\_RCS得出的latency丟給MR\_LCS會跑出相同的resource，但我測出來不同，我一開始以為是我寫的程式有bug,de了很久，後來印出MR\_LCS各個階段的排成結果跟資源增加的時間點，我才知道兩者是不可逆的。

原因如下:

MR\_LCS透過調整資源大小

以testcase1為例子,**若一開始將MR\_LCS的資源初始值**

設定{3,3}將不會增加resource。

但如果設為{3,1}/{3,2}等等，就會增加resource甚至超過{3,3}

因此我研究的方向轉變為:

資源能否過了3,3就不增加:

結論是無法:

如果設定{3,2},因為資源只有2,**因此在前幾階段有一些操作無法做，導致某些op迫不得已得在slack=0時才做**。

也就是說擁有{3,3}資源以及相同t時的**整個電路的狀態是不同的**。

因此透過累加上**去調整resource不可能保證找到最佳解的。**

五、Bonus:

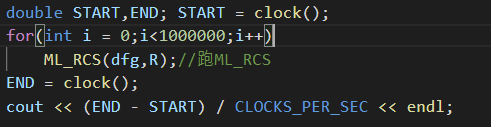
1. 可不按照node值順序輸入node

2.多做了ASAP/MR\_LCS兩種類型的排程。

3. 實驗數據

六、實驗:

1.實驗Decode對於整體效率的影響



我讓ML\_RCS執行一百萬次，並用clock來觀察時間

(主要是想測演算法時間，因此這部分不包含input/output)

要關掉Decode功能只要更改下列



為



即可。(不過如果input也考慮的話就要改add\_edge那個method)

前面[複雜度](#分析整體複雜度)提到過的

保留decode功能拿掉:整體複雜度為

**O(m+n)+O((m+n)lgn) + O(Kn)+ O(nlgn)**

將decode功能:整體複雜度為

**O(m+n)+O(m+nlgn) + O(Kn)+ O(nlgn)**

**為了用excel簡單的畫圖(x,y)數據**

**由測資(by case)可以發現EDGE數量大約為1~2n,這邊抓個2倍**

保留decode功能拿掉:整體複雜度為

**O(2n+n)+O((2n+n)lgn) + O(Kn)+ O(nlgn) = O(3n)+O(kn)+O(4lgn)**

將decode功能:整體複雜度為

**O(2n+n)+O(2n+nlgn) + O(Kn)+ O(nlgn) = O(5n)+O(kn)+O(2lgn)**

**又K為OP的delay值，這邊平均值去算=(1+3)/2 = 2**

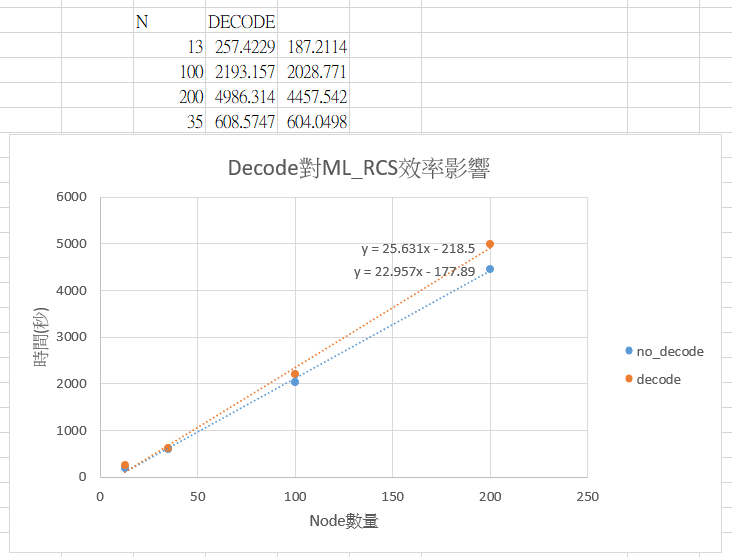
保留decode功能拿掉:整體複雜度為

**O(3n)+O(2n)+O(4lgn)= O(5n)+O(4nlgn)**

將decode功能:整體複雜度為

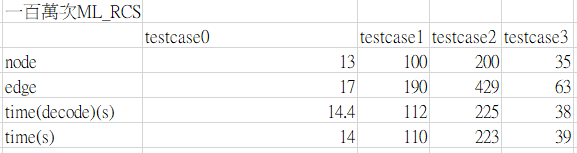
**O(5n)+O(2n)+O(2lgn)= O(7N)+O(2nlgn)**

**這樣就簡化成n跟time之間的關係圖了。**



(理論時間複雜度圖)

由理論得出的時間複雜度圖顯示，不管有沒有decode，因為n太小了，在這四個測資中的時間複雜度差異定不大。

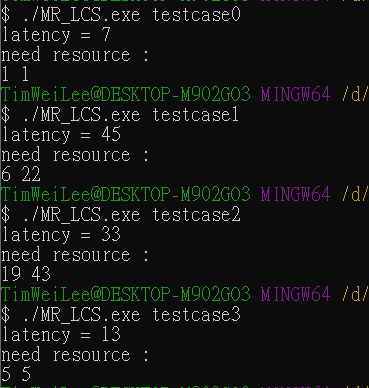
實驗結果: 

果然差異不大。

**實驗二:ML\_RCS/MR\_LCS的可逆性**

我在testUnit內有放每個schedule的測試單元。

MR\_LCS.cpp內是利用ML\_RCS的結果(latency)輸入MR\_LCS得到RESOURCE



得出來的結果



可以由testacse3得知，得出來的資源不一定會比較差，但也不一定會比較好，MR\_LCS跟ML\_RCS兩者是獨立的。

用一個簡單的道理解釋:

一開始設的資源數太少會導致有些時間點會有很多的slack=0的工作

因此會一瞬間大量增加資源，

所以**不可能透過慢慢增加資源**保證找到最佳解。

簡單的例子如下

將作業類比成工作OP數量/每天努力的最大程度為資源

1. 有10份作業,Deadline是三天後
2. 努力程度:
3. 如果一開始設為1

那就是1，1，8，最大資源就是8。

但如果是一開始努力一點

1. 一開始設為2

2，2，6，最大資源就是6

1. 再努力一點設為3

3，3，4，所有最大資源就變4了

要證明MR\_LCS無法透過慢慢增加資源得到跟ML\_RCS的值的證明如下:**只要證明這些不同的資源狀態起始值最後得出的結果會不同**

這樣就可以證明MLRCS MRLCS不可逆了

**因為MLRCS就是屬於其中的那個case。**