Programming Assignment Report

工海三

B05602022

盧庭偉

1. 演算法流程 (Algorithm Flow) <解釋程式的運作>

依照題目的 hint，將 grid model 成一個 2D array of node，其中 node 是我們自訂的 class，包含一些計算 overflow 及跑 Dijkstra 演算法所需要的資訊，然後每個 net 跑一次 Dijkstra 演算法，跑完後將路上經過的邊 demand + 1。

Pseudo code 如下:

```
create graph
for all nets:
    graph.initialize();
    path = graph.dijkstra();
    graph.update_weight(path);
    output(path);
```

2. 問題與討論 (Discussion) <討論實作中遇到的問題及疑問>

(1) 自訂 class: node

代表題目中表格的一格。包含四項資料:

short processor // 紀錄 Dijkstra 演算法中的 processor

double distance // 紀錄. Dijkstra 演算法中的距離

bool relaxed // 紀錄 Dijkstra 演算法中使否被看過

double edges[4] // 記錄四個邊界的 wire 數量

(2) std::map (RB tree)

為了在 Dijkstra 演算法中的 Extract-Min 有更好的 performance，因此每當我們 relax 一個 vertex 時，我們就將它放進一個 map(RB tree)，Extract-Min 則

是從這個 map 中 pop 出 root。這樣 Extract-Min 的 time complexity 即可由 linear search 的$O(n^2)$下降為$O(2 * \lg n) = O(\lg n)$

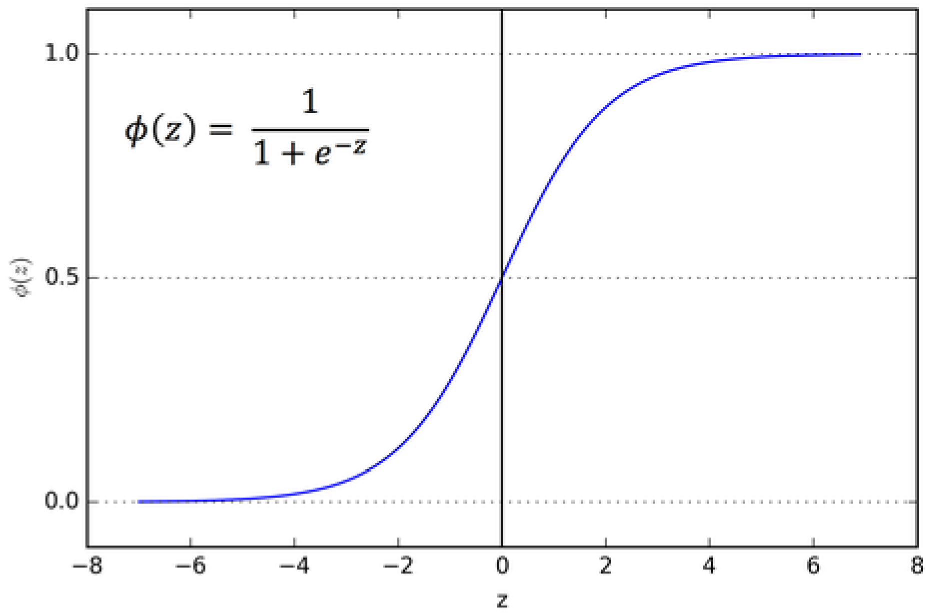
(3) std::stack

在跑完 Dijkstra 演算法後，沿著終點往回推到起點，沿路把經過的 vertices 放到一個 stack 裡，再回傳給主程式。

3. 問題與討論 (Discussion) <討論實作中遇到的問題及疑問>

如同題目所說，我們對 edge weight 的定義會影響到 path 的選擇，也就會影響到最終 overflow 的數量。

由於 overflow 的計算是「超過 capacity 後多一條 ＋1」，也就是說低於 capacity 前有幾條都不影響，超過 capacity 後超過多少都一樣多一條 ＋1。因此我們要限制的條件主要會放在接近 capacity 的附近。所以我們很自然想到了 sigmoid function，定義如下(定義 z = demand/capacity):



sigmoid function 符合了我們想要的條件，在低於 capacity 的地方沒有明顯限制，因為都不算 overflow；高於 capacity 的限制也不會隨著 demand 有明顯的增加因為都一樣是多一條 overflow + 1。

比較不同 edge weight 定義的 overflow 差別:

	4x4	5x5	10x10	20x20	60x60
F1	0	0	11	2267	128119
F2	0	0	13	2128	141422
F3	0	0	11	5806	180412
sigmoid	0	0	8	14	52533

F1 = log(demand / capacity + 1)

F2 = demand / capacity

F3 = $2^{demand/capacity} - 1$