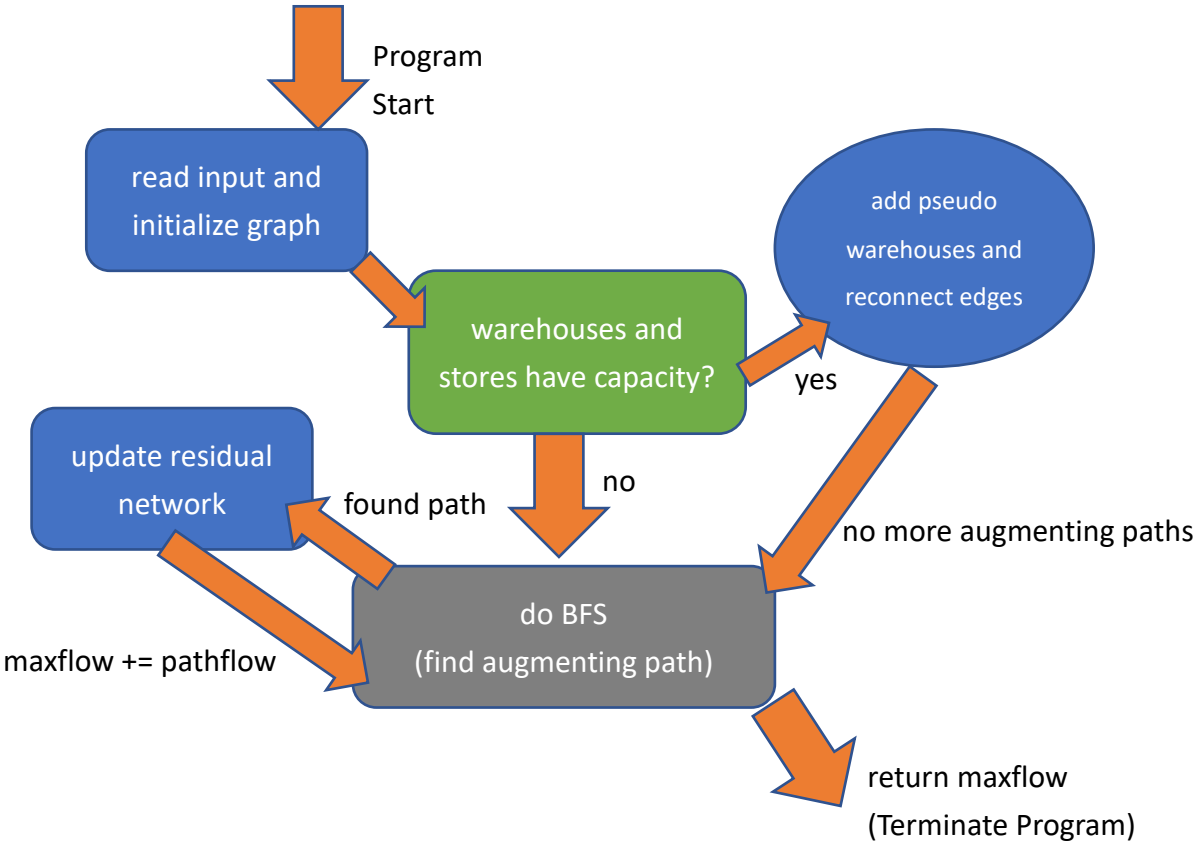
A. Flow Chart



B. Design Concept

1. Overall Design (Ford-Fulkerson Implementation)

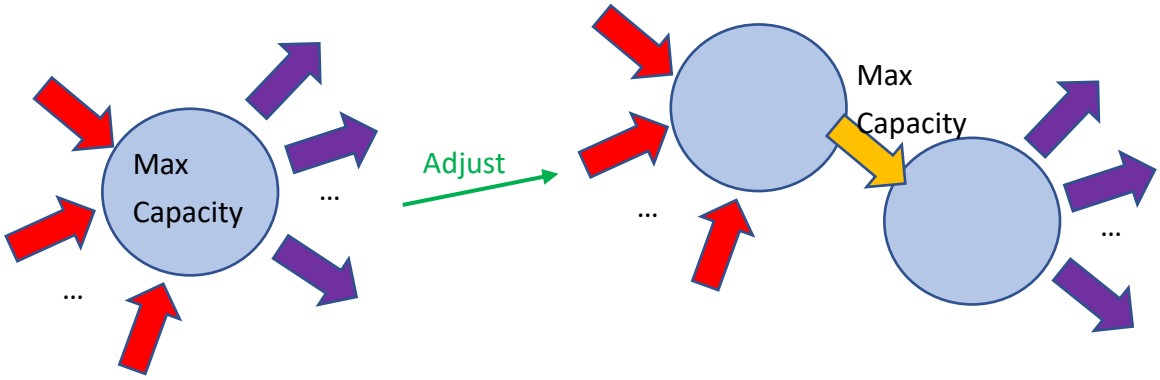
有一點要提到的是，在解典型的 maximum flow 的問題時，都是一個 source 和一個 sink，但是這次的作業有好幾個 store，所以為此我又多加了一個 pseudo sink，然後把所有的 store 和這個 sink 相連接，然後把這些連接的 edge weight capacity 設為無限，這樣就等效於一個 source 和一個 sink，同時也不影響到 max flow 的計算。

如上所示，從原本的 graph 開始，不斷地用 BFS 尋找 augmenting path，再把找到的 residual flow 更新到 residual network 上，並且把 residual flow 加到 maximum flow，再對這張 residual network 做重覆的事，一直到沒有 augmenting path，回傳 maximum flow 就可以得到答案了。

2. Bonus Design (Modifications to solve for capacity problem)

原本的題目只有對 edge 的 capacity 做限制，vertices 本身沒有，而若 vertices 有容量限制的時候，我們其實可以把它拆成兩個 node 來看，而這兩個 node 之間的 edge capacity 就是這個 vertex 的 capacity，原本流入的 path 就從其中一個節點進入、另外一個出去，如下圖所示。這樣的修正好處就是

不用重新設計 BFS 和 Ford-Fulkerson 的函式，而對原本的 graph 做更動就好了，其餘都是在做一樣的事。



C. Discussion or Problems in Implement Time

i. Time Complexity

Ford-Fulkerson 迴圈會在有 augmenting path 的情況下不斷地工作，而最壞狀況就是當每跑一次迴圈只增加了大小為 1 的 flow，且 path 把每條邊都走一次，若是這樣的話，就一共要跑 max_flow 次，而每一次都要跑 E 個邊，所以 Ford-Fulkerson 的 time complexity 為 O(max_flow*E)。

但是對於找 augmenting path 這件事若能利用比較好的演算法，就能改善 time complexity 的 worse case，所以在這次的作業中，我使用了 Edmonds-Karp 方法，也就是利用 BFS 去找 augmenting path，這樣找到的 augmenting path 會在每跑一次迴圈逐漸遞增，每做一次就會拿掉一個 critical path，而這樣每邊有 V/2 次是 critical paths，另外一共有 O(E) 對 vertices，而如上述對於每對 vertices 之間的邊能為 critical path 的次數為 O(V)，所以一共要做 O(VE)的累加，又 BFS 為 O(E)，因此，Edmonds-Karp 的複雜度為 $O(VE^2)$。

ii. Code Implementation

除了演算法對 execution time 有影響之外，實際在用程式語言實現演算法的方式也會有舉足輕重的影響。在設計的一開始，我存 flow graph 是利用二維的 adjacency matrix，但這樣其實很容易浪費 memory，因為若 graph 的邊數不多的話矩陣很容易變成大部分為 0 的 sparse matrix，一旦跑很大的測資就會出現像下圖的錯誤：

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
```

因此，後來我改良了自己的寫法，將原本大小為 (所有節點)^2 的二維 vector 改成二維的 map，只把有存在的邊存到記憶體，就可以節省許多空間，在跑回圈的時候也可以省掉很多時間因為就不用去檢查不存在的邊了。