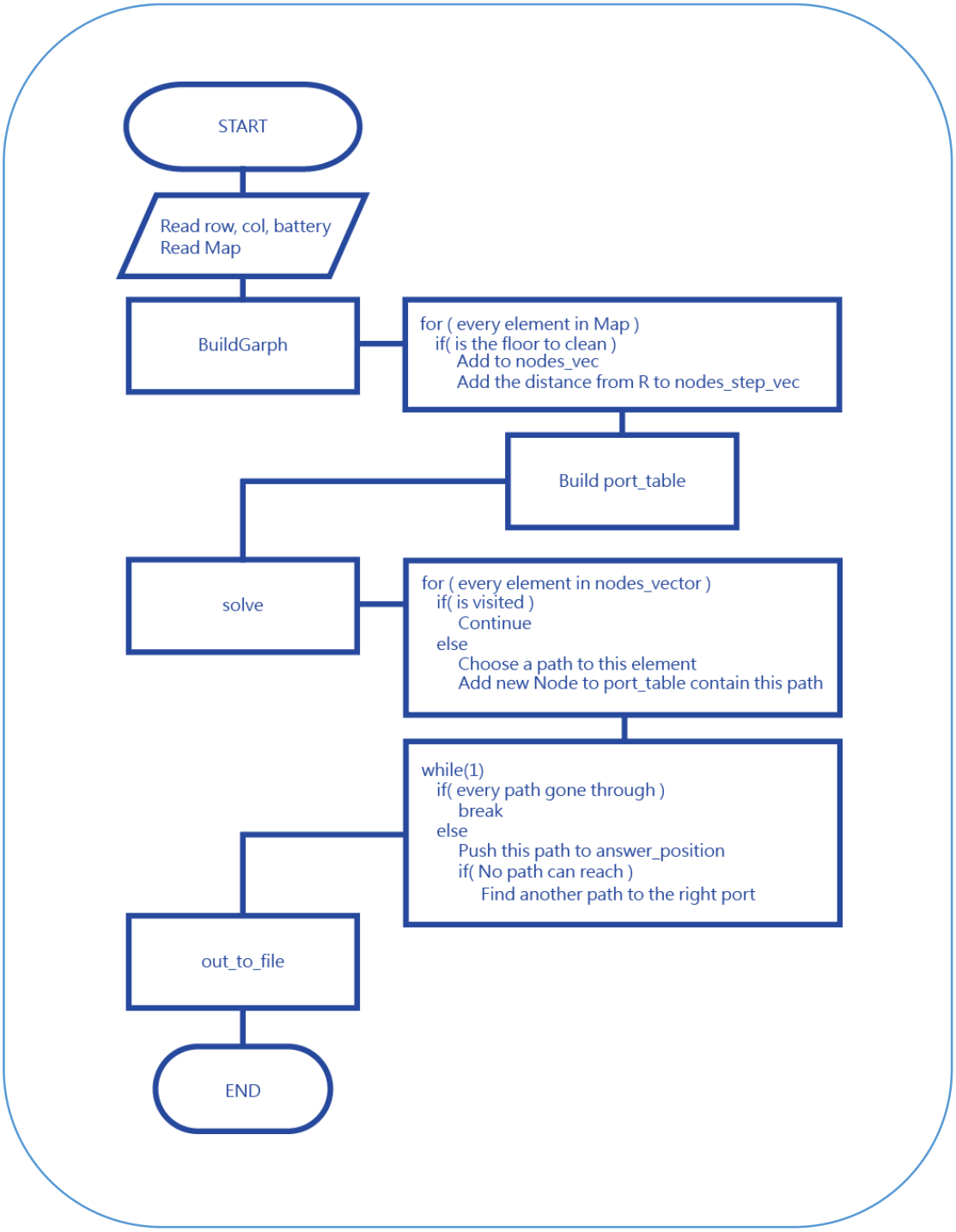
PROJECT2 – FLOOR CLEANING ROBOT

106000103 趙貞豪

Project Description

-Program Flow Chart:



-Detailed Description

設計的架構如上圖所示，接下來我將一一細說各項目所要達成的目標以及實作方式，並闡述原因與想法。

首先，我要介紹這次 project 的流程。由於在這次的運算過程當中會運用到很多繁雜的計算，所以為了方便管理，我建立了 class FCR，並以呼叫 member functions 的方式逐步拆解問題。對照上圖來看，我們首先呼叫的是 BuildGraph 這個 function，在這個步驟當中有幾個目標需要完成：(1)nodes_vec 的建立、(2)nodes_step_vec 的建立、(3)StepTable 的建立、(4)Port 的建立以及(5)PortTable 的建立，基本上這個步驟有點像是初始化，把剛剛從 Map 讀進來的資料做一些整理，並儲存在 FCR 當中的 private 變數當中，以利之後作更進一步的處理。以下方表格說明個變數的意義及功能：

變數名稱	功能
nodes_vec	儲存 pair<bool, Position>，代表各點是否清掃過。
nodes_step_vec	儲存各點到 R 的最短距離。
StepTable	儲存各種距離相對於 nodes_vec 的 index 區間。
Port	儲存各個 Port 的 Position。
PortTable	儲存 Port-to-Port 分類 Path 的 Link List。

Note.(a)Port 所代表的是 R 的出口，已可以說是以 R 為起點走第一步能到的地方。(b)Port-to-Port 分類 Path 所指的是各種 Path 依照進出 Port 的不同而給不同的分類值去代表他們的屬性，舉例而言，從 0 號 Port 出去，1 號 Port 回來，我們會給它 type=4，下面會有主題專門討論。

在建立完基本的資訊後，我們進入 Flow Chart 當中的下一個步驟，也就是去呼叫 FCR 當中的 solve function。在這個步驟當中我們要完成幾件事情：(1)對於每個點，找出它所屬的 paths、(2)將這些由點構成的 paths 配對成一去一回的 Node，並依性質不同加入所屬的 PortTable 當中、(3)從 PortTable 當中取出路徑開始走，確保每一條 paths 都有走到，並將答案 push 到 answer_position 以利之後輸出、(4)若遇到找不到相應 port 的情況，則呼叫 revise_path function，去修正成能走且正確的 port。

經 solve 的步驟之後，我們能得到一連串的 Position 存在 answer_position 當中，接下來的任務就只要呼叫 out_to_file，將答案輸出即可。

Test case Design

-Detailed Description of the Test case

在 test case 的方面有兩方面的考量，(1)夠大、(2)夠難走。

第一點很好理解，也就是直接開 1000*1000，讓 Map 本身很佔空間。第二點，是針對我個人測試的情況，相信每個人因為演算的方式不同，難走的情況也就不同。以我的方式而言，因為我是用任意路徑去找，所以 port-matching 就會變成是一大工程，也會是錯誤容易出現的地方。所以我的設計是 4 個 port(依我的設計，1 個跟 2 個 port 的 case 不會出現需要呼叫 revise_path 的情況)。另外就是呼叫 revise_path 後需要修正的步伐數，所以我的設計是將任兩個 port 離得很遠，讓修正需要增加的步數增加。

但後來由於我沒辦法在 10 秒內將這筆測資跑完(1000*1000 大約需要 17~21 秒左右)，所以我後來改成 800*800，讓結果能符合題意。

Discussions

-Concept of paths and ports

有關於上面所提到 PortTable 的部分，以下將更細部的講解。首先，要知道的是 PortTable 當中所裝的是 Node，而各個 Node 包含了 type, u, v 等 elements，分別代表的就是各個 Path 的進出的 port 編號，以及 in-path 跟 out-path，而 port 編號是定義出來的，是依據 u 跟 v 最後一個 Position 所在位置所編號，也就是說，當這條 path(v)是從 1 號 port 出發到某個 destination position，再從同個 destination position 回歸到 2 號 port，我們會給它 type value = 9，以下表作為參照：

in\out	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

而另一個討論是有關於 revise_path。就像上面有提到，這是一個先選好

path，再事後挑選任意 path 的方式，所以 port 一旦變多，無法銜接的狀況就會比較嚴重。舉例而言，底下是一個 PortTable type 的分布：

in\out	0	1	2	3
0	0	0	1	0
1	0	0	1	0
2	0	0	23	0
3	0	0	0	4

起初，我們選 type 2 號的 path，也就是從 2 號 port 出發到 0 號 port。再來，我們的起點變成 0 號 port，這時經過搜索就會發現，PortTable 當中沒有可以走的 Path 了，如下圖：

in\out	0	1	2	3
0	0	0	0	0
1	0	0	1	0
2	0	0	23	0
3	0	0	0	4

這時候就會呼叫 revise_path，讓原本所在的 0 號 port 轉換成 2 號 port(依據是否能到達最多地方作為判斷)，並將這個過程的所需經過的 Positions 加入 answer_position 當中，讓下一次 iteration 能從 2 號 port 開始，以此類推。

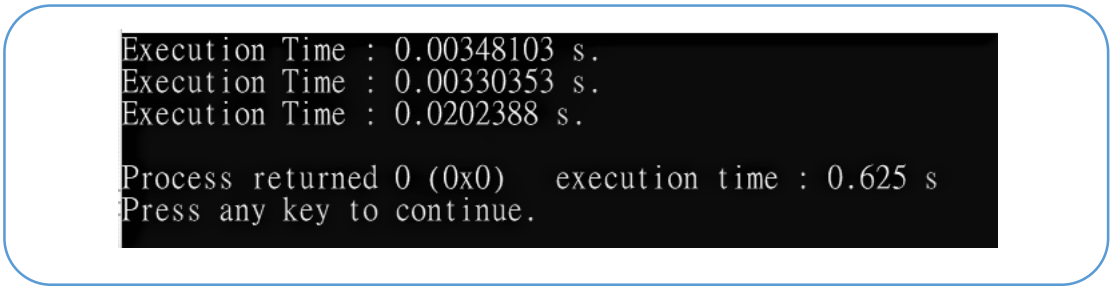
- Battery Limitation

可以發現我上面的討論當中都沒有提到 battery，事實上，還真的沒用到 battery，原因是有關題目的限制，沒有任何無法到達的地方，所以依據我最短步數的搜尋法，因此任意 path 是不會超出限制的。但其實這裡是一個非常好的發揮點，因為我的這些路徑只是短，並不佳，我這樣的方式只有在 battery 剛剛好的情況下能發揮比較貼近最佳的 step，其他情況勢必會多走冤枉路，所以要如何在既定 battery 當中尋找最佳 path 也是另一值得研究的領域，在這邊我還

沒想出相對應的方式去處理多於資源的問題。

- Time Conservation

在這個部分我的想法是將原本是 vector 的部分改為使用動態陣列去操作，由於我需要的操作非常簡單，方式有點類似 queue，所以我只要一開始要一塊記憶體，並且多用一個變數去記陣列的 size 就可以控制好存取了。這樣的方式雖然 time complexity 跟 vector 是一樣的，但在實際執行的時間上差距是非常多的。個人做了一項測試，比較靜態陣列、動態陣列、vector 在最後插入一個元素所需的時間，每個都插入 1000000 次，結果如下：



結果是動態陣列最為快速，比 vector 快出了 6 倍之多，因此後來的修正版，將 nodes_vec、nodes_step_vec 都改成了動態陣列的形式，以節省過多的時間。