二維運輸網路中具有轉彎加權之最佳路徑規劃

詹景裕
國立台灣海洋大學
資訊工程系
b0199@mail.ntou.edu.tw

謝効輯
國立台灣海洋大學
電機工程系
lsword@edirect168.com

呂紹偉
國立台灣海洋大學
電機工程系
b0119@mail.ntou.edu.tw

摘要

路徑長度和轉彎次數一直是運輸和航海交通工具的主要成本因素，路徑越長則所花時間越久；而每次轉彎必須減速也增加時間成本的支出。自Lee於1961年提出路徑連接演算法後，一直被廣泛應用在地理資訊系統(Geographical Information System, GIS)或印刷電路板佈線與積體電路元件配置的研究課題上，用以尋求最短路徑。

然而，一條最佳的路徑不應只考慮距離長短，轉彎次數也是一項不可忽略的因素，但是在搜尋時同時考慮兩個以上參數的路徑搜尋演算法，眾所週知是一個NP-complete的課題，除非使用線性組合的方式將兩個參數結合為一個參數[4]。

本文應用類似Lee演算法的觀念，提出最少轉彎路徑演算法，用以在網格圖上尋找最少轉彎路徑，另一方面，我們結合了Kirby所提出的構想及Ahuja等人所改進的Dijkstra演算法發展而成的最小成本演算法，可在網格圖及網狀架構上，尋找最小成本路徑，兩者的時間複雜度皆為$O(N)$，而N為網格數或節點總數。

關鍵詞：最短路徑、網格圖、Lee演算法、Dijkstra演算法

1. 緒論

在網格圖（Raster）及二維網狀架構（Mesh）上依照所要求的條件，來規劃一條由起點到終點的連接路徑，是相當受到重視的研究課題。其可應用的範圍相當廣泛，包括電路板的佈線或是積體電路的元件佈置，另外，在地理資訊系統的應用上，也可作為交通工具及機器人的行進路線規劃。目前關於交通工具路徑規劃的研究，皆假設載具行進速度固定，故大多以尋求最短路徑為主，以期能在最短的時間內到達目的地，然而，在行進過程中，因為慣性的緣故，每逢轉彎的地方，則必須減速，此舉將延緩到達的時間，是故，轉彎次數的多寡，對於路徑規劃而言，也是不可忽略的因素。

最少轉彎的路徑意味著因轉彎減速所花費的額外時間最少；規劃一條最少轉彎的路徑，也是研究人員所追求的目標。本文提出兩個演算法，即最少轉彎路徑演算法及最小成本路徑演算法，其中最少轉彎路徑演算法將找出二維網狀架構上由起點到終點的所有連接路徑中，轉彎數最少的路徑。另一方面，一味地專注於減少轉彎次數通常會造成路徑長度的增加。而本文所提出的另一個演算法，即最小成本路徑演算法，則針對此問題提出折衷之道，尋求包含適當轉彎數與長度兩樣因素的最小成本路徑。

本文的研究背景啟始於Lee對網格式資料結構中，尋找最短路徑的研究，其於1961年所提出的最短路徑連接演算法(簡稱Lee演算法)，至今仍受到廣泛應用[6]。除了尋找最短路徑外，盡力避免轉彎也是研究人員所追求的目標。第一篇提到最少轉彎問題的論文是由Caldwell所提出，他以建構虛擬網路的方式來表示原始網格圖上網格和網格間的連結關係[2]。大約在與Caldwell同一時間，Martin也提出了另一個方法，以一個常數的變化來偵測轉彎的發生[7]。1969年，Kirby回顧整理了相關的論文並提出他的演算法[5]。而在1974年，Frank對於最少轉彎問題提出了他的看法並指出，Caldwell，Kirby和Potts，他們所提出的演算法時間複雜度太高，然而Frank僅僅在概念上描述了他的想法而缺乏實做的細節[8]。

本文擷取類似Lee演算法的觀念，並使用了*wave*及*new_wave*兩個串列來儲存運算中所產生的暫存值，加以改良成適用於在二維網狀架構上尋找最少轉彎路徑的演算法。另外，我們以累加的方式定義路徑的時間成本，並結合了Kirby所提出的構想以及Ahuja等人所改進的Dijkstra演算法以重新建構的方式來尋求包含適當路徑長度和轉彎次數的最小成本路徑演算法，可通用於網格圖及網狀架構。兩者的時間複雜度皆為$O(N)$。

2. Lee最短路徑演算法

為了解決邏輯繪圖、佈線、最佳化的路徑規劃之有效的路徑連接， Lee提出一演算法，使計算機能夠解決下列三個問題：

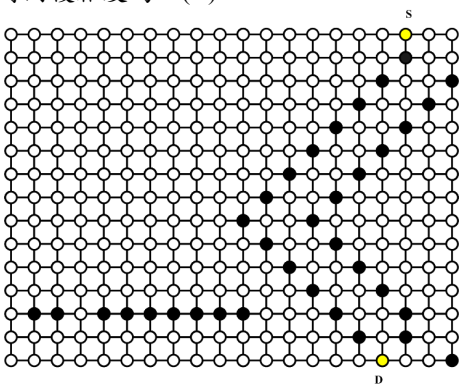
1) 找出一條跨越最少已存在路徑之路徑。
2) 找出一條避開障礙物的路徑。
3) 找出一條最佳化的路徑，例如跨越最少已存在路徑或這些路徑中最短的路徑。

Lee演算法工作於C空間(C-space)中[6]，在2D平面中的應用上，通常以四角網格來表示C空間，雖然Lee演算法並不限定在四角網格，因為三角網格或是六角網格，在電腦的圖形的資料結構上不如四角網格容易表示，所以都以四角網格來代表C空間。

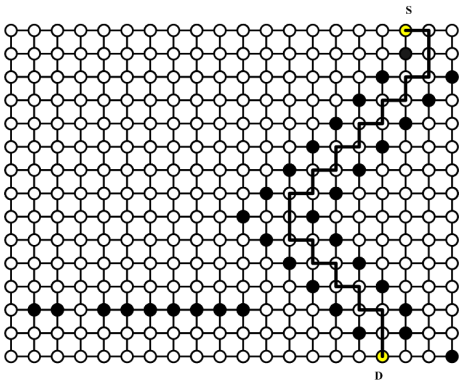
Lee演算法將起始的網格定為search wave(儲存在L串列中的網格)的第0圈，並在其周圍產生第1圈的search wave，不斷擴張此search wave，直到遇到終點或是search wave無法繼續擴張為止。在search wave的外緣之所有

網格稱作frontier cells(儲存於*L*1串列中的網格)。一個網格不能成為frontier cells除非它是search wave的鄰居；一個網格不能被刪除，除非它的所有可能成為frontier cells的鄰居都成為frontier cells，這兩點是Lee演算法的特性。

Lee 演算法在四角網格的空間資料結構中，只對直接相連的四個鄰居作運算，也就是只能朝四個方向搜尋路徑，而由於frontier cells在其所有鄰居被處理過後，就會標示起來，因此不再進入迴圈中，故演算法有很好的效率，其時間複雜度為 $O(N)$。



(a) 選定起點與目的地



(b) 經由Lee演算法的計算後的最短路徑

圖1 最佳路徑範例圖

圖1 為規劃最佳路徑的範例，經由Lee演算法的計算後，規劃出圖1(b)的最短路徑。

3. 最少轉彎路徑演算法

本文應用類似Lee演算法的模式，並加入一些邏輯判斷式來決定轉彎數增加與否，逐一計算出由起點到每個網格所需的轉彎數，進而找出轉彎次數最少的路徑。

3.1 資料結構

本節將介紹最少轉彎路徑演算法所使用的資料結構及方法。假設網格圖的大小為$n \times n$，共有N個網格；每個相鄰網格的距離都設為1。位於網格圖上座標為(i, j)的網格(cell)記為$C_{i,j}$，每個網格各有北、東、南、西四個鄰居(neighbor)與其直接相連，分別為$C_{i,j+1}$，$C_{i+1,j}$，$C_{i,j-1}$，和 $C_{i-1,j}$。

我們使用的資料結構大致與Lee的演算法類似，只是將網格內的欄位$Dist_{i,j}$改為$Turn_{i,j}$用以儲存從起點到此網格的轉彎數，其起始值為∞。

由於演算法進行的過程中，網格$C_{i,j}$將試算其鄰居網格的轉彎數，而我們稱網格$C_{i,j}$為接受其試算之網格的前任網格(predecessor)。而任一網格的前任網格可能不只一個，因為可能有一個以上的網格可以對同一個網格試算出同樣是最少轉彎數的不同路徑。

為了紀錄網格$C_{i,j}$之前任網格的方向，以備處理過程中回溯使用，我們定義了四個布林變數分別為North, East, South, 以及West，用以表示前任網格與網格$C_{i,j}$的相對位置，其起始值為假(false)。此外，我們使用*wave*和*new_wave*兩個串列來暫存運算時待處理的網格座標，圖2為每個網格的資料結構示意圖。

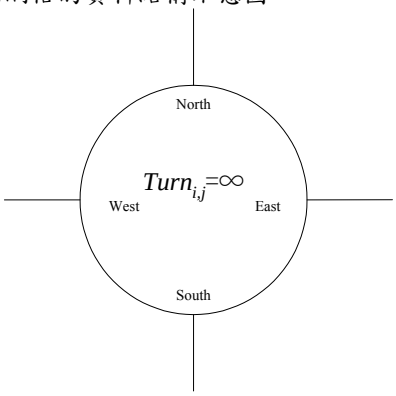


圖2 最少轉彎路徑演算法的資料結構示意圖

3.2 最少轉彎路徑演算法

最少轉彎路徑演算法包含三個步驟：第一步驟為初始化，我們將起始網格內的轉彎數$Turn_{i,j}$值給定為0，表示不需轉彎即可到達；第二步驟藉由擴張的方式計算每一個網格的$Turn_{i,j}$值；第三步驟再藉由回溯的方式輸出最少轉彎路徑。

最少轉彎路徑演算法的第一步驟將起始的網格定為search wave(儲存在*wave*串列中的網格)的第0圈；第二步驟再逐一檢視其相鄰的網格，若不需轉彎就能到達此相鄰網格則將其加入第0圈；若需轉彎才能到達此相鄰網格則將其加入第1圈的search wave(儲存於*new_wave*串列中的網格)，同時記錄前任網格的方向，不斷擴張此search wave，直到search wave無法繼續擴張為止。相對於第i圈的網格而言，所有需經一次轉彎才能到達的網格都登錄在第i+1圈。而不需轉彎即可到達的網格仍陸續登錄於第i圈；當所有可能經由第i圈網格產生的第i+1圈的網格都登錄後，我們才將第i圈的網格刪除。最後，進行第三步驟，我們藉由第二步驟所記錄的前任網格方向，一步一步回溯而得到最少轉彎路徑。

最少轉彎路徑演算法使用了兩個副程式：Counting_Turns及Update。Counting_ Turns將判斷網格$C_{i,j}$是否需要轉彎才能到達輸入的相鄰網格，並累計其轉彎數。其判斷轉彎與否的方法如圖3所示的四連接網格圖，我們以網格$C_{i,j}$為例，若其predecessor的方向為南北向，則其

南方與北方的neighbors，即$C_{i,j+1}$和$C_{i,j-1}$。因位於與$C_{i,j}$的predecessor方向相同的路徑上故不需轉彎即可到達；然而，東方與西方的neighbors，即$C_{i+1,j}$ 和$C_{i-1,j}$。則因其方位與原先路徑不同而需再經一次轉彎才能到達，故其轉彎記數比$C_{i,j}$的轉彎記數多1。

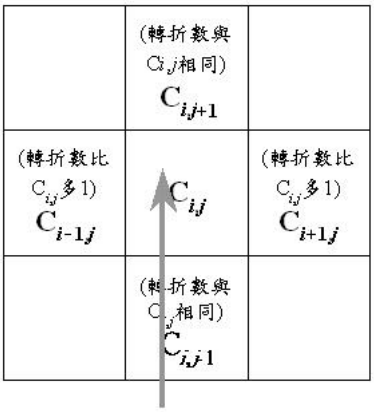


圖3 計算最少轉彎數的方法

Update依轉彎與否將相鄰網格登錄到第*i*+1圈或第*i*圈的search wave，並藉由相鄰網格內的布林變數(North, East, South, 以及West)，留下最少轉彎路徑的相關資訊。副程式Counting_Turns及副程式Update描述如下：

Function: Counting_Turns($C_{i,j}$, neighbor)

Step1: 計算由$C_{i,j}$進入此相鄰網格，所需之轉彎數。

Function: Update ($C_{i,j}$, neighbor)

Step1: 依Couting_Turns所計算出來的轉彎數來更新此相鄰網格的轉彎數。

Step2: 清除此相鄰網格內原先有關predecessor的資訊；記錄$C_{i,j}$成為新的predecessor。

Step3: 若轉彎數未增加，則將此一相鄰網格加到串列*wave*。

Step 4: 若轉彎數遞增，則將此一相鄰網格加到串列*new_wave*。

最少轉彎路徑演算法的第一步驟將起始的網格定為search wave的第0圈，第二步驟經由副程式Counting_Turns逐一試算其相鄰網格的轉彎數，若副程式Counting_Turns所計算出來的轉彎數較小則藉由副程式Update予以更新；反之，則不予更新。若計算出來的轉彎數與原先存於相鄰網格的轉彎數相等，則留下相關的路徑資訊而不需更新。不斷擴張此search wave，直到search wave無法繼續擴張為止；第三步驟則是根據第二步驟所留下的路徑資訊回溯輸出最少轉彎路徑，若有多重選擇則以不改變原路徑方向為原則。最少轉彎路徑演算法描述如下：

Algorithm：最少轉彎路徑演算法

Step1: *初始化*

將起始網格的$Turn_{i,j}$設為0並將起始網格座標存在串列*new_wave*。

Step2: 擴張

Step2.1: *wave* ← *new_wave*。

new_wave ← ϕ。

Step2.2: 從串列*wave*內取出一個網格C_i。

Step2.3:呼叫Counting_Turns($C_{i,j}$, neighbor)去逐一拜訪$C_{i,j}$的鄰居，但不包括$C_{i,j}$的predecessor。

Step2.4:若Counting_Turns($C_{i,j}$, neighbor)算得一個較小的轉彎數，則呼叫Update ($C_{i,j}$, neighbor)。

Step2.5:若Counting_Turns($C_{i,j}$, neighbor)算得一個相等的轉彎數，則$C_{i,j}$ 亦為此一相鄰網格的predecessor，我們將此訊息記錄在此相鄰網格內的flag。

Step2.6: 若串列*wave*內仍有網格未處理則回到step 2.2。

Step2.7: 若串列*new_wave*內仍有網格未處理則回到step 2.1。

Step3: 回溯

從目的網格開始，藉由原先記錄的predecessor找尋下一個合適的網格直到起始網格為止。若某一網格同時有數個predecessor可供選擇，則以不改變原路徑方向為原則來選出下一個網格。

圖4沿用圖1(a)的最佳路徑規劃範例，經由上述演算法的計算後，規劃出最少轉彎路徑。

3.3 最少轉彎路徑演算法的正確性及時間複雜度

3.3.1 最少轉彎路徑演算法的正確性

最少轉彎路徑演算法包含三個步驟：第一步驟為初始化，我們將起始網格內的轉彎數$Turn_{i,j}$值給定為0，表示不需轉彎即可到達；第二步驟藉由擴張的方式計算每一個網格的$Turn_{i,j}$值，這裡我們將使用數學歸納法來證明，當我們結束第*i*圈的試算，刪除第*i*圈內所含之網格時，所刪除的網格皆已得到自起始網格至該網格之最少轉彎數。

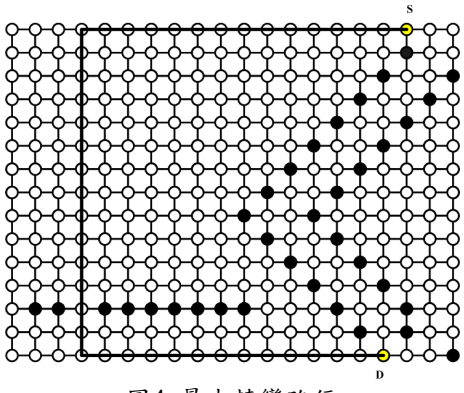


圖4 最少轉彎路徑

當$i = 0$時，所有位於第0圈內網格皆為由起始網格不經轉彎即可到達之網格，其轉彎數皆為0，即每個網格皆得到最少轉彎數。

當$i = 1$時，所有位於第1圈內網格皆為由第0圈之網格經一次轉彎即可到達之網格，及其直行延伸，故皆得到最少轉彎數。

假設$i = k$成立，即當第*k*圈結束時，所有

位於其內之網格皆有最少轉彎數，在此我們必須證明當$i = k+1$亦成立，即當我們結束第$k+1$圈的試算，刪除第$k+1$圈內所含之網格時，所有位於第$k+1$圈內的網格皆得到最少轉彎數。

對位於第$k+1$圈的網格而言，其組成的來源有兩種：由第k圈的網格經一次轉彎，以及由第$k+1$圈的網格不經轉彎而到達的相鄰網格；由第k圈產生的網格我們稱為集合P_k，而由第$k+1$圈產生的網格我們稱為集合P_{k+1}。由於我們逐一進行第k圈的網格對其四周相鄰網格的試算，將需轉彎一次才能到達的網格登錄在第$k+1$圈，而不需轉彎即可到達之網格將陸續登陸在第k圈，直到所有存在於第k圈的網格都試算完後，才將第k圈的網格刪除，是故當我們結束第k圈的試算後，第$k+1$圈內位於P_k中的網格皆得到最少轉彎數，即$k+1$，否則就應該登錄於第k圈。既然P_k內的網格皆為最少轉彎數，那麼因為P_{k+1}的網格具有和P_k的網格相等的轉彎數，所以亦為最少轉彎數，故得證。

由此可知，執行完第二步驟後即可確保所有網格所得到的轉彎數為由起始網格至該網格的最少轉彎數。

第三步驟我們將以回溯的方法，從目的地開始，一步一步地從在第1步驟所紀錄的前任網格中選取一適當的網格做為最少轉彎路徑的下一個網格，直至遇到起點為止。這裡要注意的是，若回溯時發現有一個以上的前任網格，我們以不改變原方向為原則來加以選取。如此，我們便可避免多餘的轉彎而得到最少轉彎路徑。

3.3.2 最少轉彎路徑演算法的時間複雜度

對於每個網格而言，最少轉彎路徑演算法的第二步驟只對其四個相鄰的鄰居做運算，而這四個鄰居所能的到的結果只有兩種可能：需增加一次轉彎或者不需增加轉彎數；我們將需增加一次轉彎的鄰居加入第$i+1$圈而不需增加轉彎數的鄰居加入第i圈。

對於第i圈的網格而言，所有需經1次轉彎才能到達的網格都登錄在第$i+1$圈。而不經轉彎即可到達的網格仍陸續登錄於第i圈；當所有可能經由第i圈網格產生的第$i+1$圈的網格都登錄後，我們才將第i圈的網格刪除。

令網格圖上任一網格$C_{i,j}$是經由第i圈內的另一網格$C_{i1,j1}$運算後才加入第i圈，則$C_{i,j}$將不再進入第$i+1$圈，因為第i圈的轉彎數小於第$i+1$圈的轉彎數。若任一網格$C_{i,j}$已由第i圈內的另一網格$C_{i1,j1}$運算後加入第$i+1$圈，然而稍後才經第i圈的另外一個網格$C_{i2,j2}$運算更新為較小的轉彎數而加入第i圈，由於處理完第i圈後才會處理第$i+1$圈內的網格，因此對於任何屬於第$i+1$圈內的另一網格$C_{i3,j3}$而言，其所能提供給$C_{i,j}$的轉彎數必不小於第i圈所提供的轉彎數，於是$C_{i,j}$的轉彎數將不被更新而不再進入迴圈。

最少轉彎路徑演算法的第一步驟為初始化，故只需常數時間。對於第二步驟而言，一個網格至多進入迴圈兩次，每次只對四個鄰居做運算，處理過後便不再進入迴圈，故最少轉彎路徑演算法第二步驟的時間複雜度為$O(N)$。而最少轉彎路徑演算法的第三步驟藉由回溯的方式找出最少轉彎路徑，其最糟的情形將要經過N次計算才能夠找出最少轉彎路徑，故最少轉彎路徑演算法的第三步驟的時間複雜度為$O(N)$。

由以上可知最少轉彎路徑演算法的時間複雜度為$O(N)$。

4. 最小成本路徑演算法

在路徑搜尋過程中，若只專注於縮短路徑長度則可能得到包含過多轉彎次數的路徑；同樣地，若一味地專注於減少轉彎次數，通常會造成路徑長度的增加，如何同時考慮距離長度與轉彎次數兩項時間成本，找出包含適當轉彎數與長度的最小成本路徑，一直以來是研究人員所追求的目標。

然而，同時考慮兩項參數的路徑搜尋演算法，眾所週知是一個NP-complete的課題[4]，於是本文應用Kirby所提出的架構，來修改每個節點的資料結構，將每個節點更換為以若干節點(vertices)組成的虛擬網路，使之能夠表達轉彎與直行等等各種行進狀態[5]，接下來應用Dijkstra演算法[3]來尋求最小成本路徑。

4.1資料結構

這節我們將介紹Kirby所提出的構想。自Caldwell於1961年提出最少轉彎問題後[2]，許多研究人員即對此一問題提出後續的研究。研究的重心大多集中在如何在網狀架構上表達轉彎的概念，並依此計算轉彎數，用以避免轉彎[5,7,8]。其中，Kirby提出了如圖5的構想，我們將每個節點改為一個由8個虛擬節點所組成的虛擬網路來代替。各節點間由鏈結(edges)互相連接，這些鏈結代表了網狀架構上的各種行進狀態，包括行進方式及行進方向等等，每項動作所須付出的時間成本權重，我們則以$W_{T_h}^{D_m}\left(C_{i,j},t\right)$來表示，使用者可以用常數或時間函式定義，而$D_m$代表行進方向，$D_m$= $\{(D_0,D_1,D_2,D_3)|(D_N,D_E,D_S,D_W)\}$，分別代表向北、向東、向南、向西等行進方向，$T_h$代表為行進方式，$T_h = \{(T_0,T_1,T_2,)|(T_L,T_R,T_S,)\}$，分別代表左轉、右轉、以及直行。為了簡化問題起見，本文暫不使用時間函式來定義$W_{T_h}^{D_m}\left(C_{i,j},t\right)$。Kirby同時也提出將虛擬節點與相鄰節點共享的做法，如圖6所示，當圖5的虛擬節點與相鄰節點共享時，其節點到節點間所耗費的時間成本將轉嫁到它的3個輸出路徑上[5]，為了有別於原先定義的$W_{T_h}^{D_m}\left(C_{i,j},t\right)$，我們令新的權重為

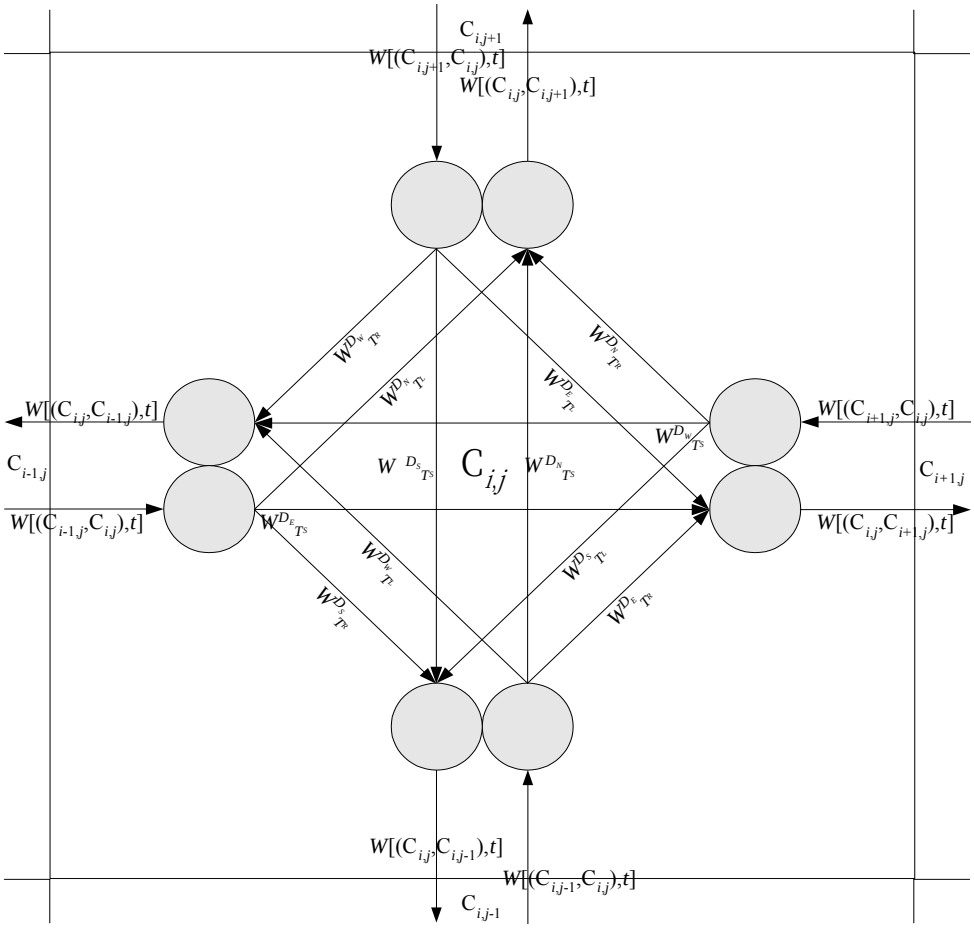


圖5 最小成本路徑演算法的節點資料結構示意圖

$SW_{T_h}^{D_m}$ ，其函式如下:

$SW_{T_h}^{D_m} = \sum W_{T_h}^{D_m}(C_{i,j},t) + W[(C_{i,j},C_{i',j'}),t]$ 函式(1)

其中包含有:

$i' = i + (+1)^E + (-1)^W$ 、 $j' = j + (+1)^N + (-1)^S$ ，E、W、N、S分別代表適用的是節點$C_{i,j}$的東、西、北、南四個方向上的鄰居節點，$W[(C_{i,j},C_{i',j'}),t]$代表從節點$C_{i,j}$到節點$C_{i',j'}$的所需的花費，使用者可以用常數或時間函式定義，但在此為了簡化問題起見，本文暫不使用時間函式來加以定義。

本節所使用的路徑時間成本計算方式，是以累加的方式，將該路徑所經過的鏈結花費累計起來，即是該路徑的時間成本。設由起始節點到節點$C_{i,j}$所經過的路徑包含之節點依序記錄為 $C_{i(0),j(0)\ i(1),j(1)\ i(2)j(2).....i(d),j(d)}$ ，其中，$i(0),j(0)$ $i(1),j(1)$ $i(2), j(2)$ ….$i(d),j(d)$為由起始節點到節點$C_{i,j}$所經過的節點座標，令該路徑的時間成本為 $Cost_s^d$ ，則：

$$Cost_s^d = \sum_{l=0}^{d} SW_{Th}^{Dm}(C_{i(l),j(l)},t) \quad \text{函式2}$$

雖然Kirby提出如此完備的虛擬網路來模擬節點間行進的行為，然而卻使用樹狀搜尋法來尋求最少轉彎路徑，造成非常高的時間複雜度[5]。

4.2 Dijkstra最短路徑演算法

Dijkstra於1959年所提出的最短路徑演算法，適用於有向圖上搜尋最小成本路徑，只要有向圖上的所有鏈結花費為非負整數皆可使用[3]。其主要想法是維持一個集合S，使之包含所有已找到由起點s至該節點最短路徑的節點，所有不屬於集合S的剩餘節點所成的集合稱之為Q，即$Q = \{u \mid u \in V-S\}$，接著由Q中找出包含最小花費的節點，用以加入集合S；不斷重複此一步驟直至到達目的節點為止。令有向圖G = (V,E)且假設每個鏈結$(u,v) \in E$ 的花費$w(u,v) \geq 0$，Dijkstra演算法描述如下[3]：

Algorithm：Dijkstra(G,w,s)
Initialize-Single-Source(G,s)
$S \leftarrow \phi$; $Q \leftarrow V[G]$
While $Q \neq \phi$
  do u ← Extract-Min(Q)
    $S \leftarrow S$ U $\{u\}$
    for each vertex $v \in Adj[u]$
      do Relax(u,v,w)

因為Dijkstra使用線性排序，來尋找集合Q中擁有最小值的節點來加入集合S，故其時間複雜度為$O(V^2+E)$，其中V為節點總數，E為鏈結總數[3]。自此之後，許多相關的研究陸續被提出，這些研究大多針對排序方面加以改進，以期

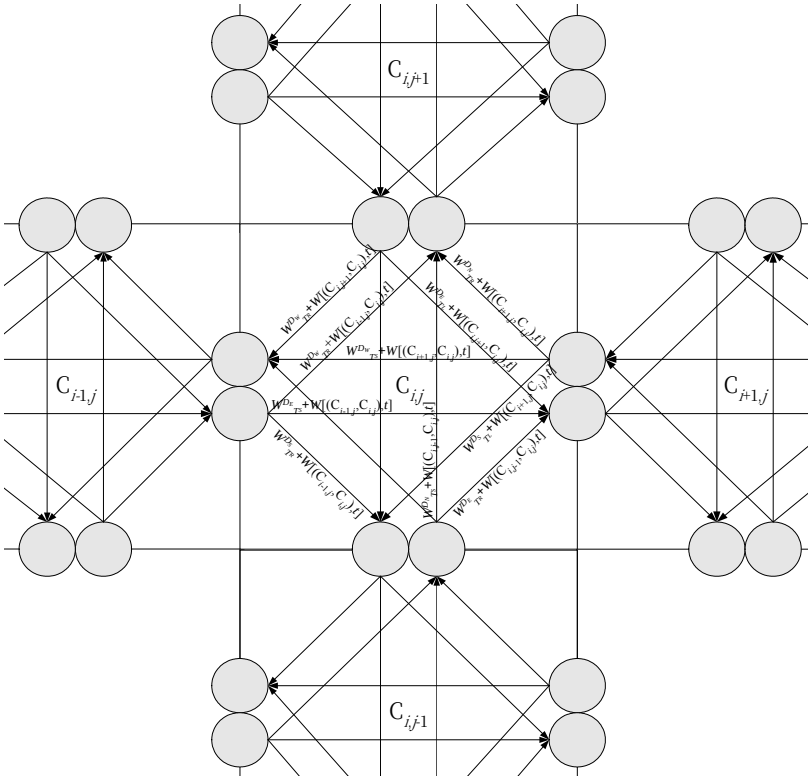


圖6 共享後的節點資料結構示意圖

能在最短時間內搜尋擁有最小值的節點，其中，以Ahuja等人於1988年所提出的改進方式為最快[1]。其改進方式為使用Radix heap 配合Fibonacci heap來加以排序，使得改進後的Dijkstra最短路徑搜尋演算法的時間複雜度可達到 $O(E+V\sqrt{\log W_{max}})$ ，而 W_{max} 為鏈結花費(edge weight)中的最大值。

4.3 最小成本路徑演算法

我們所提出的最小成本路徑演算法包含四個步驟，第一步驟為初始化，我們採用Kirby的方法，重新建構網狀架構，並將位於起始節點的虛擬節點存到集合S；第二步驟再呼叫Ahuja所改進的Dijkstra演算法，不斷擴充集合S直至所有節點均找到由起點到該節點的最小成本路徑為止；第三步驟則是由目的節點所含的節點中，選出時間成本最小的節點，其時間成本即為由起始節點到目的節點的最小時間成本 $SW_{T_h}^{D_m}$ ，所代表的路徑即為由起始節點到目的節點的最小時間成本路徑，第四步驟則是藉由第二及第三步驟所留下的路徑資訊，回溯而得到最小成本路徑。最小成本演算法描述如下：

Algorithm：最小成本路徑演算法

Step1: 初始化

Step1.1: 採用Kirby的方法，重新建構網狀架構

Step1.2: 將位於起始節點的虛擬節點存到集合S。

Step2: 計算每個虛擬節點的時間成本。

Step2.1: 呼叫Ahuja所改進的Dijkstra演算法，到所有節點均加到集合S為止。

Step3: 選出目的節點內具有最小時間成本的虛擬節點

Step3.1: 由目的節點內的虛擬節點中，選出時間成本最小的虛擬節點，其時間成本即為由起點到目的節點的最小時間成本。

Step4: 回溯

從目的節點開始，藉由第二及第三步驟所留下的路徑資訊，回溯到起始節點為止即得最小成本路徑。

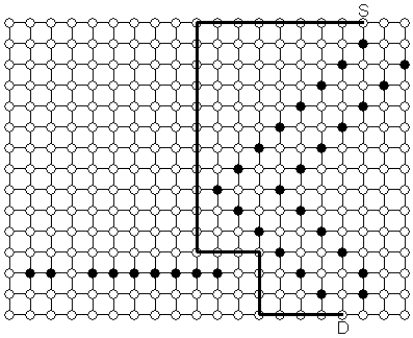


圖7 轉彎一次所需耗費的成本為前進一格的6倍之最小成本路徑

我們假定各行進方向的左右轉彎所需耗費的時間成本相同，都是前進一格的6倍，而直行不須耗費時間成本，只需耗費前進一格時的時間成本，以圖1(a)的最佳路徑規劃範例，經由最小成本路徑演算法的計算後，規劃出圖7的最小成本路徑：

4.3.1 最小成本路徑演算法的正確性

我們應用Kirby所提出的虛擬網路架構取代二維網狀架構上的所有節點，使得二維網狀架構成為有向連接圖(direct-connected graph)，於是便可使用Dijkstra演算法來尋求最小成本路徑，從而輸出正確的最小成本路徑。

4.3.2 最小成本路徑演算法的時間複雜度

我們假設網狀架構上有$n \times n$共N個節點，每個節點為由4個虛擬節點及12條鏈結所組成的虛擬網路，而最小成本路徑演算法的第一步驟為初始化，只需常數時間。第二步驟藉由Ahuja所改進的Dijkstra演算法，來尋求最短路徑，其時間的複雜度為$O(E+V\sqrt{\log W_{max}})$，因為$N$個節點的網狀架構其鏈結總數為$12N$，節點總數為$4N$，而$W_{max}$為常數，是故最小成本路徑演算法第二步驟的時間複雜度為$O(N)$；第三步驟每個節點各自由其組成的4個節點選出擁有最小時間成本的節點，使其時間成本成為由起點到此節點的最小時間成本，故最小成本路徑演算法的第三步驟僅需常數時間；最小成本路徑演算法的第四步驟，我們使用回溯的方法來輸出最小成本路徑，最糟的情形將不多於$2N$次計算，即可找出最小成本路徑，是故第四步驟的時間複雜度亦為$O(N)$。

由以上所述，我們可知最小成本路徑演算法的時間複雜度為$O(N)$。

4.4最小成本路徑演算法之參數調整

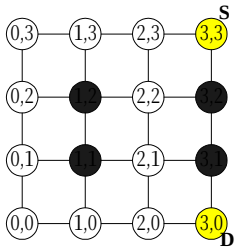
我們將路徑所經過的鏈結成本累計起來，定義為路徑之時間成本，進而尋求最少成本路徑。使用者可藉由定義各種鏈結的時間成本，使得最小成本路徑具有不同意義;當使用者定義所有的轉彎皆不占權重，即權重為零時，函式2所計算的時間成本，將只考慮距離長短這項影響時間成本的因素，所規劃的最小成本路徑將成為最短路徑；同理，當使用者定義距離不占權重時，所規劃出的最小成本路徑即為最少轉彎路徑。

然而，同一個網狀架構上可能有一條以上的最少轉彎路徑，因其具有相同的並且是最少的轉彎數，如圖8(a)所示，網狀架構上由起始節點$C_{3,3}$到目的節點$C_{3,0}$共有兩條最少轉彎路徑，分別如圖8(b)及圖8(c)所示，其中圖8(b)所示的路徑長度較圖8(c)所示的路徑長度為短，而兩條路徑卻具有相同的轉彎數。若我們要尋求最少轉彎路徑中，長度最短的路徑，很顯然地，我們必須調整轉彎一次與前進一格所需之時間成本的權重，使得距離長短不會影響轉彎的決策並且使得圖8(c)的路徑所需之時間成本大於圖8(b)的路徑。我們知道在具有N個節點的網狀架構上最長的路徑為蛇狀路徑(snack)，其長度為N，當使用者定義前進一格的權重皆為1，左右轉彎的權重皆為N，直行一次的權重皆為0時，即使是最長的路徑亦無法影響最小成本演算法對於轉彎與否的決策。圖8(c)的路徑所需之時間成本大於圖8(b)的路徑，因為最小成本演算法能尋求最小成本路徑，故藉由此一方法，我們可以找出最少轉彎路徑中，長度最短的路徑。

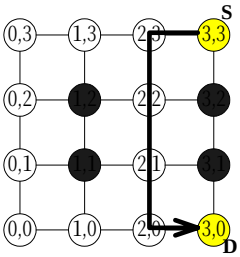
同理，如圖9(a)所示的網狀架構，由起始節點$C_{1,3}$到目的節點$C_{2,0}$共有三條具有相同距離的最短路徑，分別如圖9(b)、(c) 以及(d)所示，其中圖9(b)所示路徑具有最少的轉彎數，當使用者定義前進一格的權重皆為N，左右轉彎的權重皆為1，直行一次的權重皆為0時，即使是轉彎次數最多的路徑亦無法影響最小成本演算法對於縮短距離的決策，因而我們可得到最短路徑中，轉彎數最少的路徑。

4.5 最小成本路徑演算法於運輸網路之實例

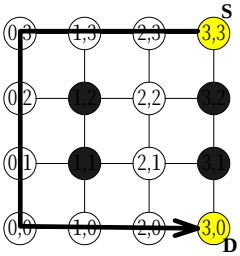
最小成本路徑演算法可以在網狀架構上尋找最小成本路徑，是故我們可以將之應用在運輸網路上，做最佳路徑規劃。如圖10(a)為一個真正的市區街道圖，其中包含了若干單行道，我們令左轉所需付出的時間成本大於右轉所需付出的時間成本，並且通過交通瓶頸的路段，需付出較高的時間成本。使用者選定起點與目的地之後，規劃出圖10(b)的最小成本路徑。



(a)選定起點與目的地

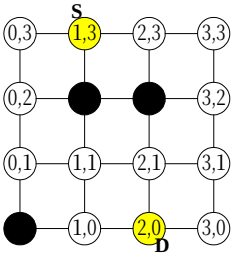


(b)最少轉彎路徑之一

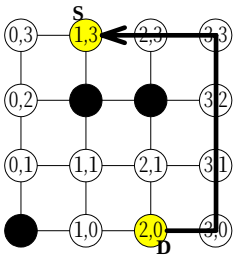


(c)最少轉彎路徑之二

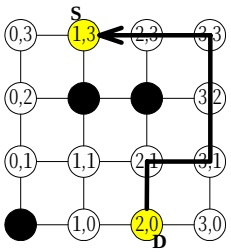
圖8 搜尋最少轉彎路徑的範例



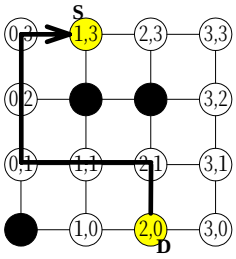
(a)選定起點與目的地



(b)最短路徑之一

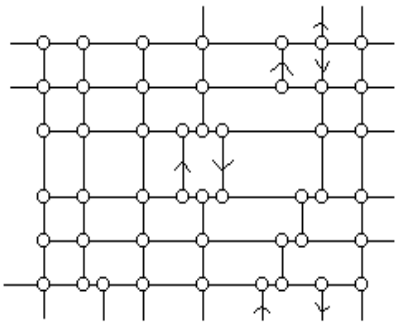


(c) 最短路徑之二

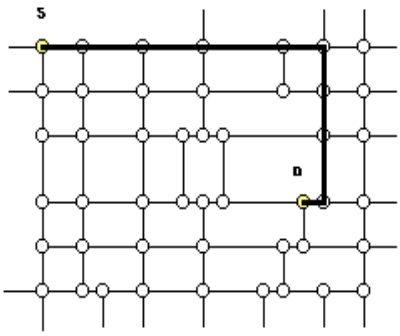


(d) 最短路徑之三

圖9 搜尋最短路徑的範例



(a)一個市區街道圖



(b)最小成本路徑

圖10 最小路徑演算法的應用實例

5. 結論與未來發展

本論文所提出的最少轉彎路徑演算法，是針對在二維網狀架構上尋求最少轉彎路徑而設計，其原理為應用類似Lee演算法的模式，並加入一些邏輯判斷式以得知轉彎數是否增加，逐一計算出由起點到每個節點所需的轉彎數，進而找出轉彎次數最少的路徑，其時間複雜度為$O(N)$，與Lee演算法相同，其中N為節點總數。

由於在路徑規劃演算法中，只專注於減少轉彎，通常會造成路徑長度的增加，是故本文採用累加的方式，定義出路徑的時間成本計算方法，進而發展出最小成本路徑演算法，用以尋求包含適當距離及轉彎數的最小成本路徑。使用者可依照不同的需求，定義不同的參數以求得最短路徑，最少轉彎路徑，最短路徑中轉彎數最少的路徑，最少轉彎路徑中距離最短的路徑，以及包含適當距離及轉彎數折衷的最小成本路徑。我們所發展的演算法有很好的效能，其時間複雜度為$O(N)$。

6. 參考文獻:

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E.Tarjan, “Faster algorithms for the shortest path problem,” *Journal of ACM*, Vol. 37, pp. 213-223, 1990.

[2] T. Caldwell, “On finding minimum routes in a network with turn penalties,” *Communs Ass. Comput. Mach.* 4, pp. 107-108, 1961.

[3] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische Mathematik*, Vol. 1, pp. 508-517, 1993.

[4] M. R. Garey and D. S. Johnson, Computer and Intractability, “A Guide to the Theory of NP-Completeness,” *W. H. Freeman*, 1979.

[5] R. F. Kirby and R. B. Potts, “The Minimum Route Problem for Networks with Turn Penalties and Prohibitions,” *Transportation Research*, Vol.3, pp.397-408, 1969.

[6] C. Y. Lee, “An Algorithm for Path Connections and Its Applications,” *IRE Trans. on Electron. Computer*, Vol. EC-10, pp. 346- 365, Sept. 1961.

[7] B. V. Martin, Memmott F. W., III and Bone A. J., “Principles and techniques of predicting future demand for urban area transportation,” *M.I.T. Report* No.3, 1961.

[8] Frank Rubin, “The Lee path connection algorithm,” *IEEE Transactions on Computers*, Vol. 9, pp. 907-914, Sept. 1974.