

逢 甲 大 學  
交通工程與管理學系碩士班  
碩 士 論 文



路口延滯下路徑演算法之研究

Development of Shortest Path Algorithms  
Under Intersection Delay Considerations

指導教授：胡大瀛

研 究 生：林蔚明

中 華 民 國 九 十 三 年 七 月

## 誌 謝

一本論文的完成，除了靠著自身的努力外，最大的功勞當然是於屬那群在旁邊給予支持與鼓勵的人們。其中，最重要的當然是我的父母及家人，因為有你們的體諒與支持，讓我這個回家像撿到的孩子能夠無慮的完成學業。

此外，在逢甲的六年歲月中，另一個對我影響最深的莫過於是我的指導教授 胡大瀛教授。在這說長不長說短不短的歲月中，從老師身上學到不少為人處事的道理，在老師不厭其煩的指導與叮嚀之下，學生終於能夠充滿自信的踏出校園進入社會，在此獻上學生最深的感謝之意。

而在論文撰寫期間承蒙系上 李克聰教授與 邱裕鈞教授提供不少寶貴的意見，讓學生能夠一步一步的完成這本論文。論文口試期間，非常感謝淡江大學 董啟崇教授與環球技術學院 魏慶地教授在百忙之中抽空指導，使學生的論文更加完整。另外，也感謝 張 湧教授在學生兩年的研究所期間，所給予的叮嚀與照顧。也感謝系上其他老師與三位助教在生活上或學業上的照顧與幫助。

在這段既忙碌又辛苦的歲月中，Pluto 的 yoyo 學姊、惠如學姊、文能學長、tt、怡如、小圭和 manto 以及其他許多的學長姊、學弟妹們，很感謝你們能夠適時的滿足我所需要的幫助；逢甲資工所的乃偉、朝陽資管所的偉育，感謝你們讓我認識物件導向這門從沒接觸過的玩意兒；大學好友小江，感謝你在我叫救命時趕來救我，使我的程式能順利完成。

對於同班同學國樑、老鼠、小潔、小 P、振嘉、佳穎、延祥、TACO、宗泓、伯鴻，我不會忘記這兩年我們一同經歷的槍戰、球賽與車賽；靖峰、阿旁、Jacky、阿亮、芳誼、婉郁，你們的歌聲我也會細細回味；阿鵝、秋美、凱斌，在班上的各個活動中你們都是不可缺少的靈魂人物。在這兩年中，不論任何困難我們都一同撐過，不論任何歡樂我們也一起度過，我會好好珍惜這段緣分的。

另外，在這段充滿壓力與忙碌的歲月中，多虧亭榕的體貼，即使沒有很多陪伴她的時間，她也能無怨無悔地在我身邊給我鼓勵與支持。

需要感謝的人實在是太多了，在此雖然無法一一的列出來，但我仍衷心的感謝每一位我的朋友，因為有你們的存在才会有現在的我，真的謝謝你們。在此，僅將我的論文獻給你們，與你們一起分享。

謹誌

2004 年 仲夏

逢甲大學 e-Thesys(92 學年度)

## 摘 要

隨著資訊與通訊技術的不斷進步，智慧型運輸系統 (Intelligent Transportation Systems, ITS) 的發展也日漸進步，其中先進旅行者資訊系統 (Advanced Traveler Information Systems, ATIS)，可提供用路人資訊服務。這些資訊不僅可以方便用路人選擇出發時間、行進路線等，也可以透過即時資訊的更新，變更途中行進路線。在 ITS 提供路徑導引資訊時，一個最重要的問題就是如何產生有效與可靠的路徑資訊。

本研究主要目的為在考慮路口的成本下，發展最短路徑、K 條最短路徑與依時性最短路徑 (Time-dependent Shortest Path) 演算法，並探討其合理性。三種演算法配合路口的延滯計算，可更進一步反應交通相關的動態特性，並可應用於未來 ITS 的環境。

當車輛在交通路網中行進時，其所遭遇到的成本簡單分為兩種，一為路段間的旅行時間成本，一為通過路口時所發生的延滯成本。一般而言，路段間的旅行時間成本在短時間內除非遇有突發事故或是處於交通尖峰時段中，不然通常的旅行速率變化並不會太大，因此，影響整體旅行時間最大的應屬於通過路口時所產生的延滯成本。所以若在計算最短路徑時未將路口延滯問題列入考慮，其所得到的結果無法反應真正的交通狀況，所以在路徑計算中若能加入路口延滯成本的問題，應當可以提供用路人確實可靠的路徑資訊。

因此，本研究在分析路口相關延滯成本的計算模式後，乃採用以號誌為主的控制延滯成本 (Control Delay)，以及考量交通流量的平均紓解率延滯成本 (Average Discharge Rate) 來做為主要的路口延滯成本。並將這兩種路口延滯成本加入三種路徑演算法的計算成本中。

系統建置過程中，本研究採用物件導向式 (Object-Oriented) 的分析方式來進行系統分析，以了解系統需求及繪製相關分析圖。接著根據系統分析的結果，進行系統程式撰寫。由於物件導向系統分析能提供程式開發者充足的需求資訊及訊息傳遞流程，因此，在程式的開發過程中能夠有效的縮短開發時程。此外，物件導向所提供的設計樣式及其他工具，皆能夠提昇程式建置及系統運作效率。

進行完系統分析及系統建置後，本研究乃使用台中市路網進行數值實驗，並依據實驗結果進行演算法修正。實驗結果顯示在考慮路口延滯後，不論採用何種路口延滯成本，其所獲得的路徑結果皆比不考慮路口延滯成本的路徑結果來的接近實際情形，顯示若在路徑計算中加入路口延滯成本的考量，將能夠提供用路人更為可靠的路徑資訊。

關鍵字：最短路徑演算法、K 條最短路徑演算法、依時性最短路徑演算法、路口延滯成本、物件導向

## Abstract

Advanced Traveler Information Systems (ATIS), one of the subsystems of Intelligent Transportation Systems (ITS), are designed to deliver information directly to travelers. Based on such information, travelers can make better decisions about route and departure time. One of the key issues of providing route guidance information is how to effectively generate reliable route information.

When vehicles are traveling in the traffic network, the travel cost could be divided into two parts, link travel time and intersection delay. Most of the shortest path algorithms assume that there is no costs or prohibitions associated with intersection delays, but the intersection delays might be the dominant factor when calculating SP in a congested network. Thus, this research aims at developing one-to-all shortest path, K shortest path, and time-dependent shortest path algorithms with different cost considerations, such as intersection delays. These algorithms could reflect the dynamic characteristic of traffic and application to the environment of ITS in the future.

Two methods of calculating intersection delays are (1) average delay under signal control and (2) average discharge delay. The delay under signal control could reflect the impact of signal and variations of flow, and the average discharge delay could reflect average discharge rate.

In the research, the object-oriented approach is adopted to analyze and develop the system. According to the object-oriented analysis (OOA), this analysis could analyze the system demand and illustrate the system through different diagrams. The object-oriented approach provides two major functions: maintainability and reusability. Because of these two functions, it can reduce the programming time.

Numerical experiments are conducted based on the Taichung city. In order to illustrate the system, several databases are constructed, such as network data, signal data, volume on links, number of vehicles crossing intersection, and velocity. Thus, the system can compute the travel time on links and time of delays which crossing intersection. Thus, the system can generate several different path databases that are in accordance with different intersection delays and algorithms. The numerical experiments indicate that the SP calculation with delay consideration could indeed capture the dynamic of flow variations in a realistic traffic network.

**Keywords:** Shortest Path, K Shortest Path, Time-dependent Shortest Path, intersection delay, and Object-Oriented.

# 目 錄

目錄.....	
圖目錄.....	
表目錄.....	
<b>第一章 緒論.....</b>	<b>1</b>
1.1 研究背景與動機.....	1
1.2 研究目的.....	2
1.3 研究方法與流程.....	3
<b>第二章 文獻回顧.....</b>	<b>5</b>
2.1 智慧型運輸系統.....	5
2.1.1 智慧型運輸系統發展沿革.....	5
2.1.2 先進交通管理系統.....	6
2.1.3 先進旅行者資訊系統.....	7
2.2 最短路徑問題.....	12
2.2.1 最短路徑.....	12
2.2.2 K 條最短路徑.....	17
2.2.3 依時性最短路徑問題.....	21
2.2.4 考慮路口延滯之最短路徑問題.....	22
2.2.5 路網資料結構.....	24
2.2.6 路口延滯成本.....	25
2.3 物件導向分析與程式.....	27
2.3.1 物件導向分析與設計.....	27
2.3.2 物件導向程式設計.....	29
2.4 綜合分析.....	30
<b>第三章 最短路徑演算法.....</b>	<b>32</b>
3.1 路段與路口成本.....	32
3.1.1 路段旅行時間成本.....	33
3.1.2 平均路口延滯成本模式.....	33
3.1.3 車流量對路口延滯成本的影響.....	35
3.2 最短路徑演算法.....	36
3.2.1 問題定義.....	37
3.2.2 求解演算法.....	37
3.2.3 演算法的合理性.....	41
3.3 K 條最短路徑演算法.....	42

3.3.1 問題定義.....	42
3.3.2 求解演算法.....	42
3.3.3 演算法的合理性.....	45
3.4 依時性最短路徑演算法.....	45
3.4.1 問題定義.....	46
3.4.2 求解演算法.....	46
3.4.3 演算法的合理性.....	49
3.5 小型範例介紹.....	49
3.6 綜合討論.....	55
<b>第四章 物件導向系統分析.....</b>	<b>57</b>
4.1 物件導向系統分析.....	57
4.1.1 系統功能需求.....	57
4.1.2 系統活動圖.....	59
4.1.3 系統循序圖.....	62
4.1.4 系統類別圖.....	65
4.2 物件導向與設計樣式.....	66
4.2.1 策略性設計樣式基本架構.....	67
4.2.2 策略性設計樣式應用於最短路徑演算法.....	68
4.3 程式輸入/輸出示意圖.....	69
4.4 程式撰寫.....	70
4.4.1 路網資料結構.....	70
4.4.2 標準樣板程式庫.....	72
4.5 綜合討論.....	73
<b>第五章 數值實驗.....</b>	<b>74</b>
5.1 實驗設計.....	74
5.1.1 測試環境.....	74
5.1.2 測試資料.....	74
5.2 系統操作.....	77
5.3 結果討論.....	81
5.3.1 無路口延滯下三種路徑演算法的結果比較.....	82
5.3.2 使用控制延滯下三種路徑演算法的結果比較.....	90
5.3.3 使用平均紓解率路口延滯下三種路徑演算法的結果比較.....	94
5.3.4 綜合比較.....	102
<b>第六章 結論與建議.....</b>	<b>110</b>

6.1 結論 .....	110
6.2 建議 .....	111
<b>參考文獻.....</b>	<b>113</b>
<b>附錄.....</b>	<b>115</b>
附錄一 路徑結果.....	115
附錄二 Label correcting Algorithm 程式碼 .....	121
附錄三 Label setting Algorithm 程式碼 .....	121
附錄四 KSP 程式碼.....	121
附錄五 Time-Dependent Shortest Path 程式碼.....	121



## 圖目錄

圖 1.1 研究流程圖 .....	4
圖 2.1 自主式路徑導引產品組合 .....	8
圖 2.2 動態式路徑導引 .....	9
圖 2.3 資訊服務提供者 (ISP) 之路徑導引產品組合 .....	10
圖 2.4 整合式運輸管理及路徑導引產品組合 .....	11
圖 2.5 Dijkstra's Algorithm 演算流程圖 .....	14
圖 2.6 Label correcting Algorithm 演算流程圖 .....	16
圖 2.7 Yen's Algorithm 演算流程圖 .....	20
圖 2.8 延滯之定義 .....	26
圖 2.9 4+1 觀點的 UML 軟體架構 .....	29
圖 3.1 延滯時間估算示意圖 .....	36
圖 3.2 路口延滯下最短路徑演算法程式流程圖 .....	40
圖 3.3 路口延滯下 K 條最短路徑演算法程式流程圖 .....	44
圖 3.4 路口延滯下依時性最短路徑演算法程式流程圖 .....	48
圖 3.5 田字型範例路網 .....	49
圖 4.1 本研究之 UML 系統分析流程圖 .....	57
圖 4.2 系統功能需求 .....	59
圖 4.3 路口延滯下最短路徑演算法活動圖 .....	60
圖 4.4 路口延滯下 K 條最短路徑演算法活動圖 .....	61
圖 4.5 路口延滯下依時性最短路徑演算法活動圖 .....	62
圖 4.6 路口延滯下最短路徑演算法循序圖 .....	63
圖 4.7 路口延滯下 K 條最短路徑演算法循序圖 .....	64
圖 4.8 路口延滯下依時性最短路徑演算法循序圖 .....	65
圖 4.9 本系統所使用的類別圖 .....	66
圖 4.10 策略性設計樣式基本架構圖 .....	67
圖 4.11 最短路徑之策略性設計樣式架構圖 .....	68
圖 4.12 程式輸入/輸出示意圖 .....	70
圖 4.13 田字型路網 .....	71
圖 4.14 田字型路網的相鄰串列 .....	72
圖 5.1 台中市路網圖 .....	75
圖 5.2 本研究比較所選用的七個起點與迄點 .....	77
圖 5.3 系統操作流程圖 .....	78
圖 5.4 系統匯入檔案畫面 .....	78
圖 5.5 依時性最短路徑運算結果輸出畫面圖 .....	79
圖 5.6 Label correcting Algorithm 運算結果輸出畫面圖 .....	79
圖 5.7 K 條最短路徑演算法運算結果輸出畫面圖 .....	80
圖 5.8 無路口延滯下最短路徑結果示意圖 .....	81



圖 5.9 起始時間 100 分鐘的旅行時間趨勢圖 .....	95
圖 5.10 起始時間 10 分鐘的旅行時間趨勢圖 .....	96
圖 5.11 起始時間 50 分鐘的旅行時間趨勢圖 .....	97
圖 5.12 K 條最短路徑運算時間比較圖 .....	109



## 表目錄

表 3.1 本研究考慮範圍 .....	37
表 3.2 田字型範例路網之路網資料 .....	50
表 3.3 田字型範例路網之號誌資料 .....	50
表 5.1 數值實驗所產生的路徑數目 .....	82
表 5.2 三個不同時段下最短路徑結果之比較 .....	84
表 5.3 自由車流下 K 條最短路徑成本差異之比較 .....	85
表 5.4 自由車流下最短路徑與依時性最短路徑旅行時間之比較 .....	86
表 5.5 無路口延滯下 LCSP 與 KSP 比較結果 .....	87
表 5.6 無路口延滯下 TDSP 與 KSP 結果比較 .....	88
表 5.7 無路口延滯下 TDSP 與 LCSP 路徑結果比較 .....	89
表 5.8 無路口延滯下 TLCSP 與 TDSP 成本差異百分比 .....	90
表 5.9 控制延滯下 LCSP 與 KSP 比較結果 .....	91
表 5.10 控制延滯下 TDSP 與 KSP 結果比較 .....	92
表 5.11 控制延滯下 TDSP 與 LCSP 路徑結果比較 .....	92
表 5.12 控制延滯下 TLCSP 與 TDSP 成本差異百分比 .....	93
表 5.13 平均紓解率路口延滯下 LCSP 與 KSP 比較結果 .....	98
表 5.14 平均紓解率路口延滯下 TDSP 與 KSP 結果比較 .....	99
表 5.15 平均紓解率路口延滯下 TDSP 與 KSP 結果比較 .....	100
表 5.16 平均紓解率路口延滯下 TDSP 與 LCSP 路徑結果比較 .....	101
表 5.17 平均紓解率路口延滯下 TLCSP 與 TDSP 成本差異百分比 .....	101
表 5.18 無路口延滯成本與控制延滯成本 LCSP 路徑結果比較 .....	102
表 5.19 無路口延滯成本與平均紓解率延滯成本 LCSP 路徑結果比較 ..	103
表 5.20 無路口延滯成本與控制延滯成本 TDSP 路徑結果比較 .....	103
表 5.21 無路口延滯成本與平均紓解率延滯成本 TDSP 路徑結果比較 ..	104
表 5.22 三種路口延滯成本 LCSP 之旅行時間 .....	105
表 5.23 三種路口延滯成本 TDSP 之旅行時間 .....	105
表 5.24 控制延滯下路口延滯成本與路段旅行時間的比較 .....	106
表 5.25 平均紓解率延滯下路口延滯成本與路段旅行時間的比較 .....	106
表 5.26 三種路口延滯下 LCSP 與 LSSP 的平均運算時間表 .....	107
表 5.27 三種路口延滯下 LCSP 與 TDSP 的平均運算時間表 .....	108

# 第一章 緒論

## 1.1 研究背景與動機

由於人口與經濟成長快速，各主要城市的交通問題愈來愈嚴重，尤其是與日俱增的車輛，對各大都市的交通環境造成嚴重的衝擊，整體運輸系統因缺乏有效的管理與運作，因而無法滿足現有的需求。因此，各國均開始著手進行智慧型運輸系統 (Intelligent Transportation Systems, ITS) 的研究與發展。ITS 的主要發展目標是希望結合先進電子、控制、及通訊等新興技術，來解決各種交通問題，並加強運輸系統的效率與安全。

其中，先進交通管理系統 (Advanced Traffic Management Systems, ATMS) (交通部，2001) 係利用偵測、通訊及控制等技術，由交通控制中心藉由即時資料的蒐集、分析來制定及評估交通控制策略。在交通繁忙期間，交通控制中心可以在分析即時交通狀況後產生路徑資訊及交控策略，並將路徑導引資訊透過可變資訊標誌 (Changeable Message Sign, CMS) 即時告知用路人選擇替代路徑。對於進行整體性交通管理的 ATMS 而言，更是智慧型運輸系統的核心與基礎。

而 ATMS 所產生的資訊，可以藉由先進旅行者資訊系統 (Advanced Traveler Information Systems, ATIS)(交通部，2001) 透過先進資訊、通訊以及相關的技術，即時的將資訊傳達出去。經由此系統，用路人可以在車內、家裡、辦公室、車站等地點方便地取得所需資訊，這些資訊不僅可以方便用路人選擇出發時間、行進路線等，也可以透過即時資訊的更新，避開擁擠路段以及變更途中行進路線，以順利到達目的地。而在 ATIS 相關的產品如自主式路徑導引、動態式路徑導引、資訊服務提供者式之路徑導引、整合式運輸管理及路徑導引等，這些產品中，對路徑導引的需求十分重要，若無法提供有效率及正確的導引資訊，其所產生的效益也將相對降低許多，由此可知，路徑導引不論是在一般用路人，或是交通控制中心，皆有其重要的地位。

由於智慧型運輸系統對於資訊的即時性及正確性相當重視，因此，為了能夠達成對用路人進行最佳的路徑導引，可以透過路網指派模式來進行求解。但由於傳統靜態的分析方式無法完全滿足即時的需求，因此必須發展新的分析方式，也就是動態交通指派 (Dynamic Traffic Assignment)。

在進行動態交通指派時，一個最重要的問題就是如何產生路徑資訊，使用路人能在最短的時間內，選擇最好的一條路線來前進。因此，在路徑導引中，路徑問題實為最基本的問題。

然而在實際的交通運輸路網進行動態交通指派時，所面臨的並非一個單純以最短路徑來計算就能解決的問題。因為最短路徑在實際的應用上並不一定為最佳的，所以才會有 K 條最短路徑演算法的產生。K 條最短路徑主要是在產生成本略多於最短路徑的次佳路徑，並將所求出之次佳路徑依序排列，供用路人選擇。

但事實上，雖然 K 條最短路徑演算法已經比最短路徑演算法所求的路徑還要佳，但實際所面對的卻是一個更為複雜的最短路徑問題，稱之為「依時性最短路徑問題 (Time-Dependent Shortest Path Problem, TDSPP)」<sup>1</sup>。所謂的依時性最短路徑問題，乃係指路段間的旅行時間成本會隨著出發時間的不同而有所不同，且路徑成本除了跟出發時間有關，其也會隨著用路人出發後到達節線的時間而不同。正因為其路徑成本更加複雜，因此就必須再尋找一個可有效處理不同時間點的演算法，以進行求解。

此外，當車輛在交通路網中行進時，其所遭遇到的成本可簡單分為兩種，一為路段間的旅行時間成本，一為通過路口時所發生的延滯成本。一般而言，路段間的旅行時間成本在短時間內除非遇有突發事故或是處於交通尖峰時段中，不然通常的旅行速率變化並不會太大；因此，影響整體旅行時間最大的應屬於通過路口時所產生的延滯成本。所以若在計算最短路徑時未將路口延滯問題列入考慮，其所得到的結果無法反應真正的交通狀況，所以在路徑計算中若能加入路口延滯成本的問題，應當可以提供用路人確實可靠的路徑資訊。對於路段成本方面，在最短路徑以及 K 條最短路徑的計算開始時，便固定所使用的路段旅行時間成本以及路口延滯成本，使其不會隨時間變動而改變，而依時性最短路徑的求解過程中，路段旅行時間成本以及路口延滯成本將會隨著時間變動而使用不同的路段旅行時間成本。

## 1.2 研究目的

為了能夠提供較為實用的路徑資訊，以因應智慧型運輸系統的發展與需要，本研究針對路徑演算法加入路口延滯成本，進行演算法的開發與探討。此外，也將對進行實驗測試時所要使用的路段旅行時間成本以及路口

延滯成本來加以討論。

除了對路徑演算法和路口延滯問題進行探討外，由於近幾年來物件導向 (Object Oriented) 的概念越來越受人重用，因為其對系統進行分析的動作可以明確的定義每一個流程步驟，也能讓使用者與系統管理者之間的溝通更容易。加上物件導向的應用程式架構是由一組常用來形成應用程式核心或基礎的類別所組成，所以，物件導向程式所提供的可重覆使用性可以大幅縮減程式開發時間。

因此，根據上述說明，本研究的目的敘述如下：

1. 分別探討最短路徑演算法、K 條最短路徑演算法以及依時性最短路徑演算法，並在其中加入路口延滯成本。
2. 分析路口相關延滯成本，分別建立計算模式，使其能在各演算法求解過程中方便取得與使用。
3. 利用物件導向的觀念進行系統分析以及程式撰寫，以進行數值實驗。
4. 使用台中市路網資料進行數值實驗，並進行結果分析。

### 1.3 研究方法與流程

本研究係探討在考慮路口號誌影響的相關延滯下，進行依時性最短路徑計算，並以物件導向之系統分析、程式架構來進行之研究。研究流程圖如圖 1.1 所示，各步驟及進行流程如下所示：

#### 1. 文獻回顧

為構建所須之路徑演算法及路口延滯成本模式，針對使用之演算法及模式進行文獻回顧。此外，因本研究乃係以物件導向觀念為系統架構，因此再回顧物件導向相關文獻。

#### 2. 研究方法

針對路徑計算演算法（最短路徑、K 條最短路徑、依時性最短路徑）以及路口延滯成本模式進行兩方面的理論構建系統架構。

#### 3. 物件導向系統分析

運用物件導向觀念，對整體系統進行系統分析，分析各元件之關係，並繪製相關流程圖。

#### 4. 程式撰寫

在系統分析構建完畢以及各元件之間的連接關係確定後，著手進行程式撰寫。

#### 5. 數值實驗與結果分析

研究主要以台中市路網為例進行數值實驗，過程中根據最短路徑、K條最短路徑以及依時性最短路徑之計算結果適時的進行演算法修正。

#### 6. 結論與建議

根據分析所得之結果，提出具體的結論與後續研究之建議。

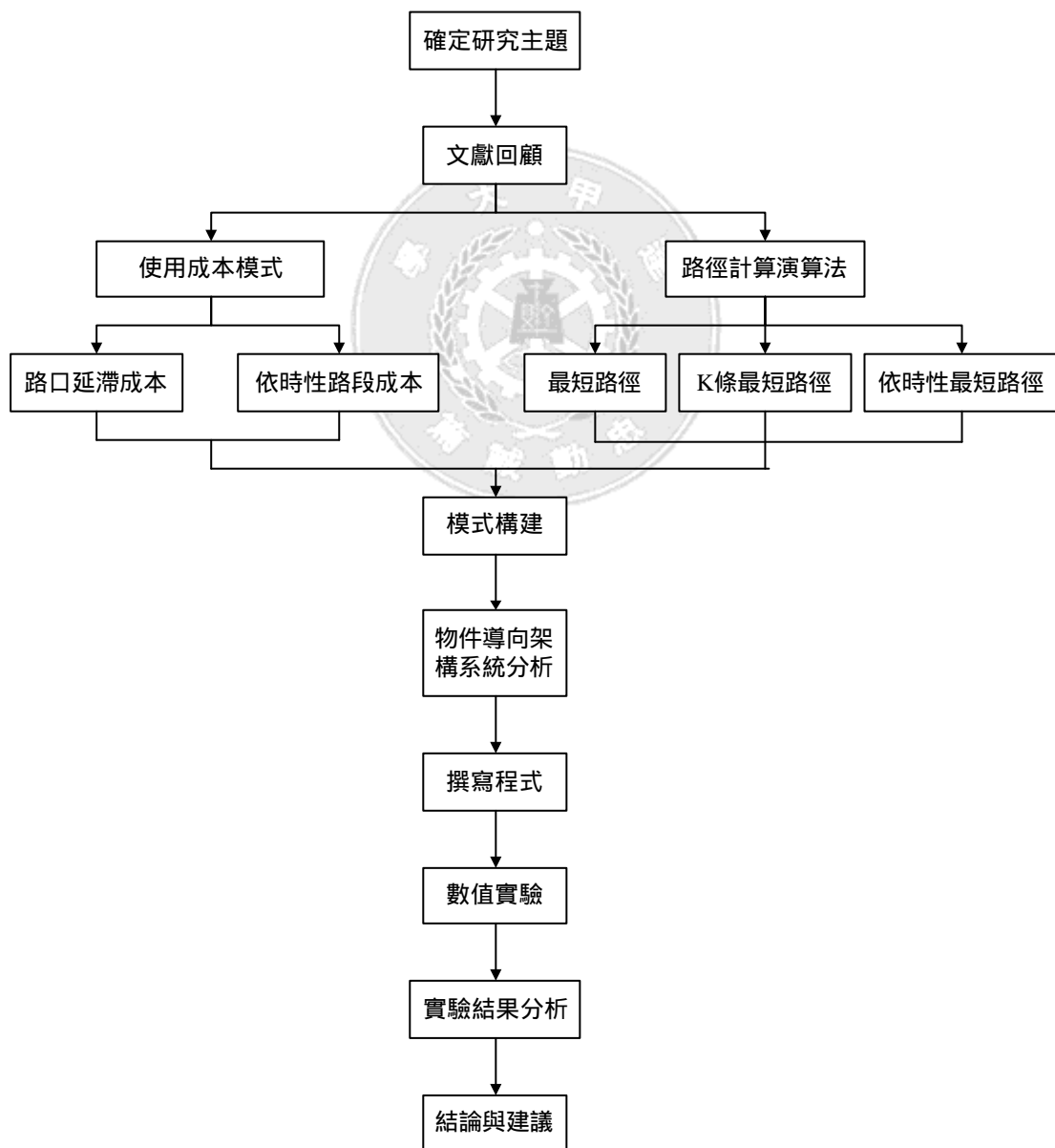


圖 1.1 研究流程圖

## 第二章 文獻回顧

本章將於第 2.1 節回顧智慧型運輸系統、先進交通管理系統以及先進旅行者資訊系統；第 2.2 節回顧最短路徑演算法，其中包括最短路徑、K 條最短路徑、依時性最短路徑問題以及所考慮的路口延滯成本問題，藉以探討近年來各界所發展之演算法；第 2.3 節物件導向系統分析與程式，回顧物件導向功用，以說明何以採用物件導向為系統主要架構核心；第 2.4 節綜合分析，說明文獻回顧與本研究之關係。

### 2.1 智慧型運輸系統

智慧型運輸系統 (Intelligent Transportation Systems, ITS)，早期又稱為智慧型車路系統 (Intelligent Vehicle Highway Systems, IVHS)，其主要是要將先進的資訊、通信、車輛、導航，以及控制等技術整合在各種運輸系統、運具中，並提供即時且正確的交通資訊，讓有限的運輸資源發揮其最大效用，提昇整體運輸效率，並改善當前的交通問題。

#### 2.1.1 智慧型運輸系統發展沿革

近幾年來世界各國皆為發展 ITS 而在各個運輸系統上努力，我國為了因應世界潮流與提昇國家競爭力，也於民國 87 年度先後完成了「智慧型運輸系統 (ITS) 發展演進與相關技術之探討」，以及「台灣地區發展智慧型運輸系統 (ITS) 綱要計劃」(交通部，2001) 等兩項基礎研究報告，為的就是要有系統、有計劃的發展適合我國國情，且符合各地需求的先進運輸科技。在綱要計劃中根據對各公私部門所進行的 ITS 單元需求調查，而整理出國內 ITS 之 7 大發展領域，其中包含先進交通管理系統 (Advanced Traffic Management System, ATMS)、先進旅行者資訊系統 (Advanced Traveler Information System, ATIS)、先進大眾運輸系統 (Advanced Public Transportation System, APTS)、商車營運系統 (Commercial Vehicle Operation, CVO)、緊急救援系統 (Emergency Management System, EMS)、先進車輛控制及安全系統 (Advanced Vehicle Control and Safety System,

AVCSS)、自動公路系統 (Automated Highway System, AHS) 等。其發展目標就在於將國內的交通運輸提升到更安全、更環保、更有效率、且更具經濟生產力。

在行政院於民國 91 年所提出的「挑戰 2008 國家發展重點計畫」(行政院, 2002) 中也特別強調「e 化交通」這項議題, 而所謂的 e 化交通就是藉由 ITS 等建設計劃的完成, 來達成整合創新科技產業的發展、提供民眾優質的運輸服務、促進國際交流接軌等願景, 由此可見國家對於 ITS 的重視。在計劃中也明示其具體策略為「ITS 技術平台及系統開發」、「智慧型都市交通控制系統」、「安全 e 計畫」、「聰明公車與交通 IC 智慧卡計畫」、「交通 e 服務計畫」等五大子項計畫, 藉由各產、官、學、研等單位共同來努力, 使台灣在 2008 年能夠達到整合創新科技、優質運輸服務、國際交流接軌的目標。

因此, 為了能夠有效地推動 ITS 整合性系統、保障系統的相容性與擴充性、輔助國內相關 ITS 標準化工作並避免重複的投資浪費, 必須研擬一套完整的 ITS 系統架構 (System Architecture, SA)。透過 ITSSA 的發展, 可以讓全國在推動發展 ITS 相關系統時, 兼顧到系統間的相容性和資料間的可交換性。因此, 在 ITS 後續的研究中也針對 ITS 中各項的子系統, 制定了許多的產品組合。(交通部運輸研究所, 2001)

### 2.1.2 先進交通管理系統

先進交通管理系統 (Advanced Traffic Management Systems, ATMS) 係一項「路的智慧化」的發展項目, 也是 ITS 重要的核心跟基礎系統。ATMS 可以透過偵測、通訊及控制等技術, 由交通控制中心藉由即時資料的蒐集、分析來制定及評估交通控制策略。因此, 藉由 ATMS 的發展, 可以在衍生出旅行者資訊服務、事故管理、交通監測等系統。在都會區中, 他甚至可以與停車管理系統和大眾運輸系統結合, 為用路人提供更大的服務。

ATMS 的相關技術有匝道儀控、事件自動偵測、動態交通預測、可變資訊標誌 (CMS)、行進間測重 (Weigh in Motion, WIM)、自動車輛定位 (Automatic Vehicle Location, AVL)、最佳路線導引等。

根據「台灣地區發展智慧型運輸系統系統架構之研究」(交通部運輸研究所, 2001) 所制定的 ATMS 產品組合主要可分為 15 大項, 包括路網交通監視、探測車交通監視、平面道路控制、高速公路控制、高乘載 (HOV) 車



道管制、交通資訊發佈、區域性交通控制、事件管理、交通預測及需求管理、電子收費、空氣污染監測、虛擬交控中心與智慧型探測資料、停車設施管理、調撥車道管理、道路天候監測等。

在上述 15 項產品中，雖然未直接說明路徑導引的用處，但路徑導引資訊卻隱含在各項目的交通資訊或是駕駛者資訊中。這些資訊除了可以提供道路路況、事故狀況等，當然也可以是替代路徑的提供。除此之外，路徑導引資訊也能協助交通控制中心處理擁擠路段或事故路段時對交通進行紓解計畫，以及對交通流量進行管理控制。

### 2.1.3 先進旅行者資訊系統

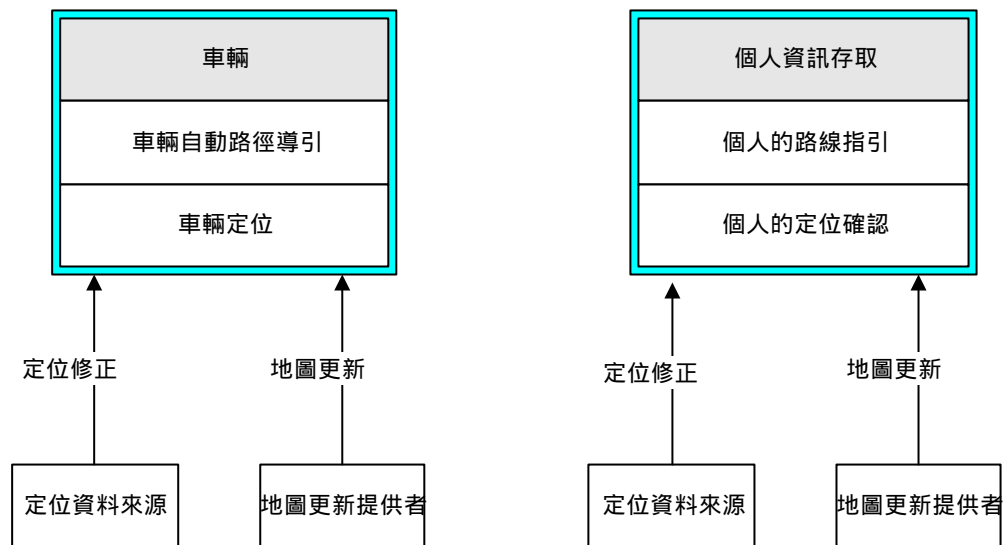
先進旅行者資訊系統 (Advanced Traveler Information Systems, ATIS) 係一項「系統的智慧化」的發展項目，使用路人不論身在何處，均可以方便地取得所需資訊。

ATIS 之相關技術有可變資訊標誌 (CMS)、公路路況廣播 (Highway Advisory Radio, HAR)、車內顯示系統、最佳路線導引、無線電通訊 (Wireless Communications)、電視路況報導、旅行服務資訊等。

根據「台灣地區發展智慧型運輸系統系統架構之研究」(交通部運輸研究所，2001) 所制定的 ATIS 產品組合主要可分為 9 大項，包括廣播式旅行者資訊、互動式旅行者資訊、自主式路徑導引、動態式路徑導引、資訊服務提供者式之路徑導引、整合式運輸管理及路徑導引、資訊查詢服務及預約、動態式共乘以及車內顯示等。其中將針對與本研究相關的課題進行回顧整理。

#### 1. 自主式路徑導引

自主式路徑導引係組合車內感應器、位置測定以及計算設備、地圖資料和通訊介面的一項產品組合，透過這些設備的整合，可以取得詳盡的路徑導引資訊。請參考圖 2.1。

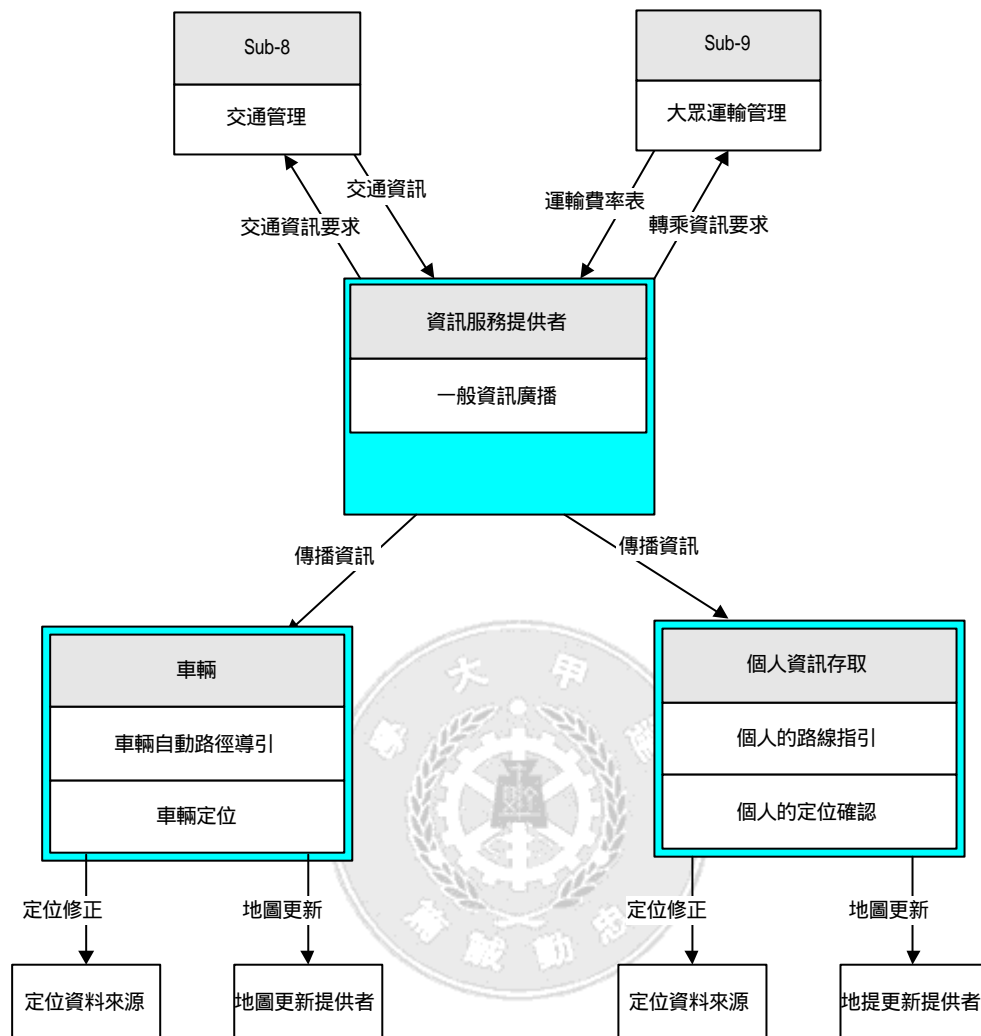


資料來源：交通部運輸研究所，2001

圖 2.1 自主式路徑導引產品組合

## 2. 動態式路徑導引

動態式路徑導引可提供符合現況的路徑規劃以及導引資訊。其係結合裝有數位式接受器之自動路徑導引設備，用以接收即時訊息，以提供動態路徑導引之用。請參考圖 2.2。

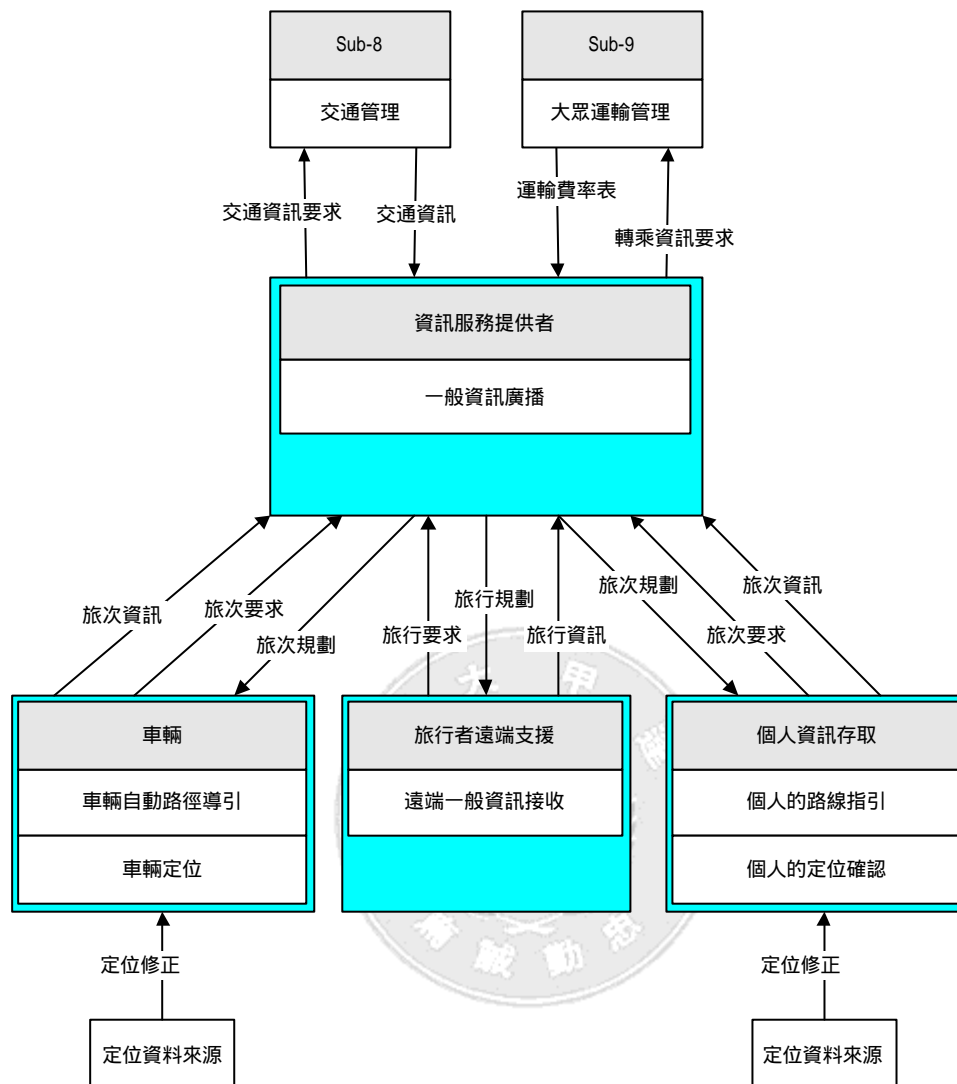


資料來源：交通部運輸研究所，2001

圖 2.2 動態式路徑導引

### 3. 資訊服務提供者式之路徑導引

資訊服務提供者式之路徑導引提供的資訊與上述相同，惟不同的是，此產品係將計算設備轉移到資訊服務提供者身上，藉以簡化使用者配備。請參考圖 2.3。

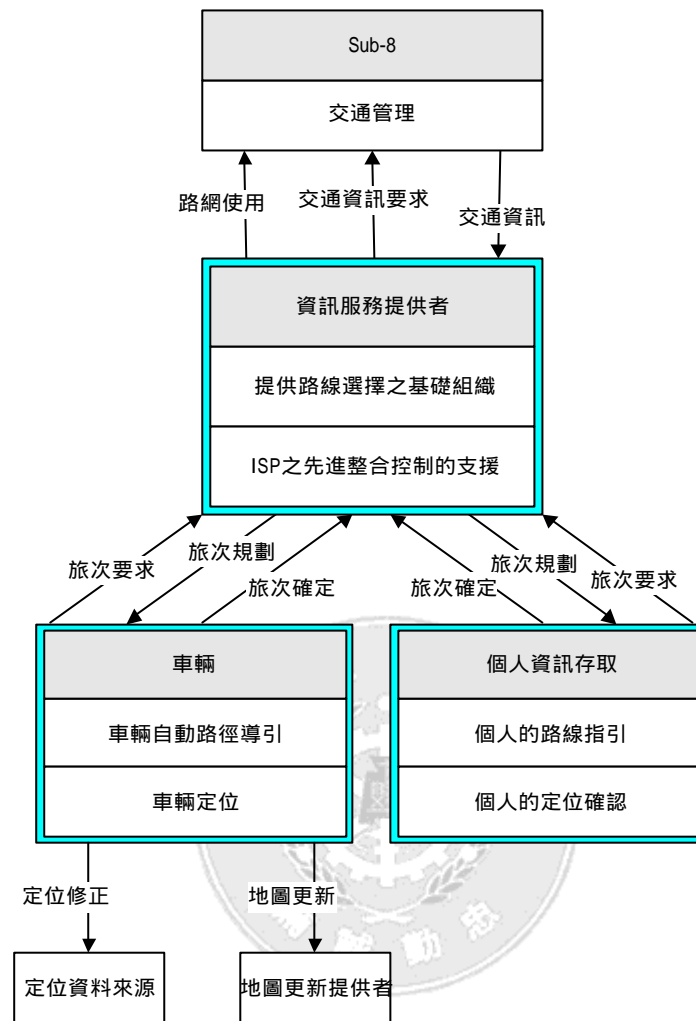


資料來源：交通部運輸研究所，2001

圖 2.3 資訊服務提供者 (ISP) 之路徑導引產品組合

#### 4. 整合式運輸管理及路徑導引

整合式運輸管理及路徑導引可以讓交通管理中心，利用即時資訊將交通策略最佳化。針對用路者提供先前路線規劃及相對應的路徑導引資訊。請參考圖 2.4。



資料來源：交通部運輸研究所，2001

圖 2.4 整合式運輸管理及路徑導引產品組合

在這些路徑導引系統中，都必須先透過定位資料，然後取得即時路況，再進行路徑計算以進行導引。因此，進行路徑導引時，一方面為了避免提供過時的資訊，一方面又要提供最佳的路徑，所以有效率且正確的最短路徑演算法確實是必須的。因此，在路徑導引中，最短路徑問題實為最基本也是最重要的問題。而又為了能滿足符合各時段的最短路徑，發展依時性最短路徑演算法更是刻不容緩。

## 2.2 最短路徑問題

最短路徑問題起源於求解兩點之間的最短路徑，但隨著各演算法的發展以及問題的考慮，以及在實際交通的應用上，最短路徑會因為許多外部問題而無法達到最佳化，因而必須尋找成本略多於最短路徑的次佳路徑，所以更進一步發展出  $K$  條最短路徑。

根據歷年來研究中可以發現，最短路徑問題的討論在未談及時間變化的前提下，已經由一條擴展至  $K$  條，此為空間上的變化。但因為其僅僅是空間上的數量變化，因此在求解的運算過程中可以藉由重複計算達到所需目的。也因為利用重複運算的特性，可以發現  $K$  條最短路徑演算法與最短路徑演算法的相似程度很高，而最短路徑演算法的演算效率，也直接影響到  $K$  條最短路徑的演算效率。

因為最短路徑問題為路徑導引中最重要之一環，所以不管在進行  $K$  條最短路徑或是依時性最短路徑的計算，皆必須從最短路徑為起點開始。因此，本小節將回顧最短路徑相關的演算法以及  $K$  條最短路徑相關的演算法。

### 2.2.1 最短路徑

最短路徑問題 (Shortest Path Problem)，乃是路網研究中極為重要的一部份，在一個路網當中，給定一起點和迄點，並考慮所有可能影響之因素，來求得最低之成本或是最快之時間到達目的。最短路徑在實際的應用上也極為廣泛，如路徑選擇與排序、文字、電腦網路傳輸和語音的辨識系統等等。而最常見於交通運輸方面，如目前最熱門之物流運輸和旅客的運輸，另外在緊急救難時是最為需要運用到最短路徑，如地震、戰爭、核電廠災變等重大災難時，提供人員疏散或緊急救難之決策方案。

根據路網上起迄點及問題特性可分為以下三類：

- (1) 一對一 (one-to-one)：由特定起點至特定迄點之最短路徑。
- (2) 一對多 (one-to-all)：由一個點至其他各點之最短路徑。
- (3) 多對多 (all-to-all)：路網上任意兩點之最短路徑。

對於前兩類最短路徑問題，主要的求解方式有 Label setting Algorithm 以及 Label Correcting Algorithm。前者必須在路網節線成本為非負的情形

之下才能夠使用；後者除了可以在非負的節線成本使用外，也可以用於節線成本為負的路網，但不能在有負循環(negative cycle)的路網中使用。至於第三類多對多的最短路徑問題，則是可以透過一對一最短路徑的求解方式，將其擴大為每一個節點都是終點來計算。而由於最短路徑演算法發展以久，所產生的演算法也不少，以下將針對較常見的演算法進行整理介紹。(Ahuja et al., 1993)

### 1. 標記設定法 (Label setting Algorithm)

Label setting Algorithm 的演算法中較具代表性的就是 Dijkstra's Algorithm，這個方法的基本概念是給節點暫時標號，這個標號表示由起點  $S$  到此節點路徑之長度上限，這些標號將隨著重複之演算步驟而增大，而且每個循環將會比較所有暫時性標號的距離標籤，選擇最小的距離標籤並將之轉換成永久性標號。而永久性標號代表起點  $S$  到該節點之最短路徑。以下則是 Dijkstra's Algorithm 的演算步驟：

令  $L(X_i)$  為  $X_i$  的成本標號值

起始：

Sept 1：令起點  $S$  為永久標號， $L(S) = 0$ ，且所有不等於  $S$  之  $X_i$  設為暫時標號， $L(X_i) = \infty$ ，令  $P = S$ ， $(P)$  為點  $P$  以一個節點相連之所有點集合。

改變標號：

Step 2：對所有  $(P)$  且為暫時標號之  $X_i$ ，更新標號值：

$$L(X_i) = \min [L(X_i), L(P) + C(P, X_i)]$$

Step 3：於所有暫時性標號中找出  $L(X_i') = \min [L(X_i)]$  及  $X_i'$

Step 4：定  $X_i'$  的標號為永久性且令  $P = X_i'$

Step 5：(1) 若只找  $S$  到  $T$  之最短路徑

i 如果  $P = T$ ，即為所求。

ii 如果  $P \neq T$ ，則重複 Step 2。

(2) 若係找  $S$  到所有點之最短路徑，則到 Step 6。

Step 6：當所有點皆為永久標號，則該標號即為該點之最短路徑，若標號仍為暫時標號，則重複 Step 2。

而根據上述演算步驟，其程式流程圖如圖 2.5 所示。

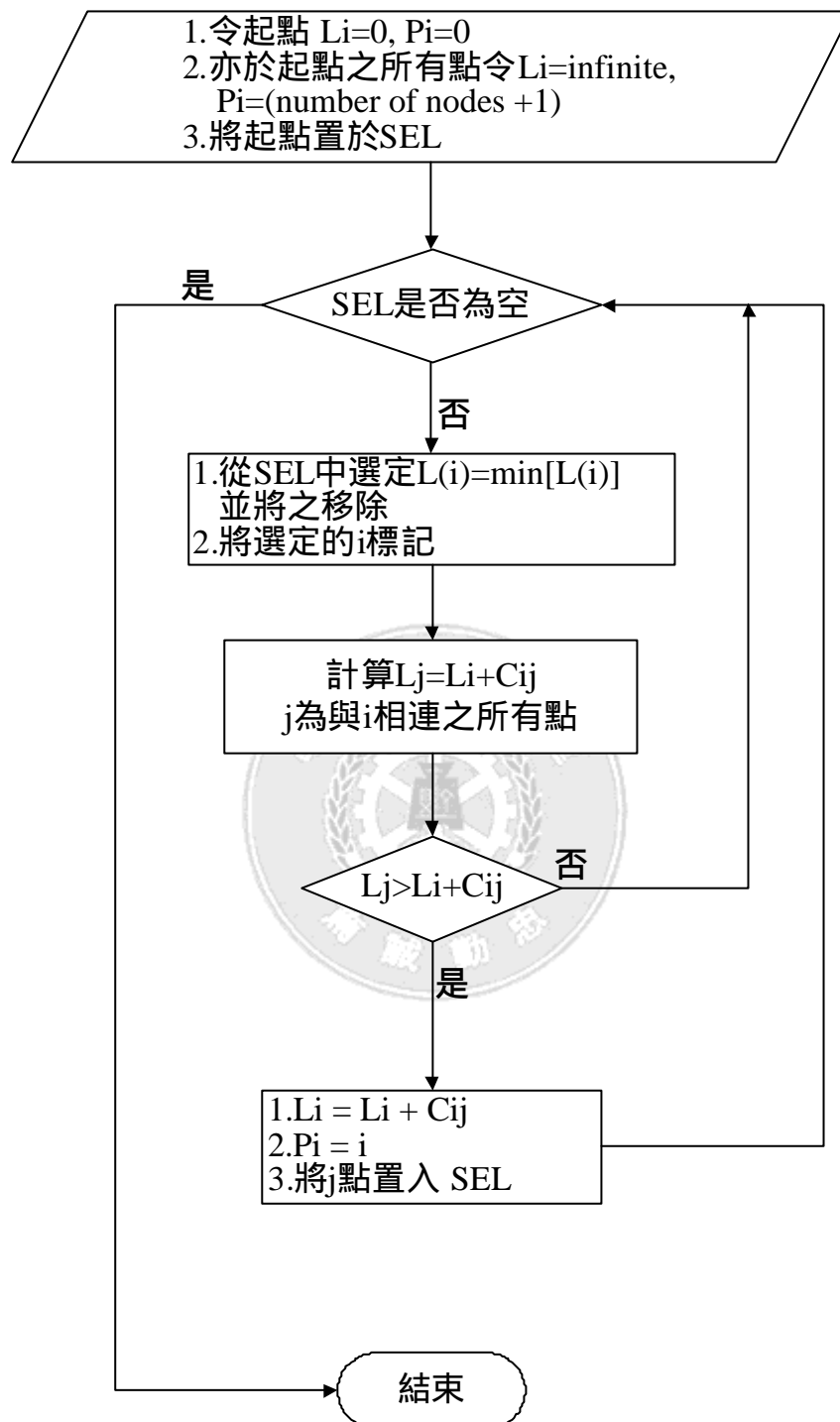


圖 2.5 Dijkstra's Algorithm 演算流程圖

## 2. 標記修正法 (Label correcting Altorithm)

Label correcting Algorithm 是除了 Label setting Algorithm 以外另一個常見的最短路徑演算法，其計算方式跟 Label setting Algorithm 有點類似，



不同的是他係將所有節點視為暫時標籤，直到最後才同時將所有節點的距離標籤變成永久標籤。以下時 Label correcting Algorithm 的演算步驟。

起始：

Step 1：令起點  $S$  的  $L(X_i) = 0$ ， $P(X_i) = 0$ ，且所有不等於  $S$  之  $L(X_i)$  設為  $L(X_i) = \infty$ ，並將  $S$  置入  $SEL$  中。

改變標號：

Step2：從  $SEL$  中選擇最前面的一點，將該點設為  $P$ ，並從  $SEL$  中移除。

Step3：找出與該點  $P$  相連之所有點  $j$ ，將所有點  $j$  設為暫時標號。

Step4：計算  $L(X_i) = \min [L(X_j), L(P) + C(P, X_j)]$ ，將  $j$  置入  $SEL$  中。

Step5：檢查  $SEL$ ，若為空集合則結束，若不是則回到步驟 2。

而根據上述演算步驟，其程式流程圖如圖 2.6 所示。



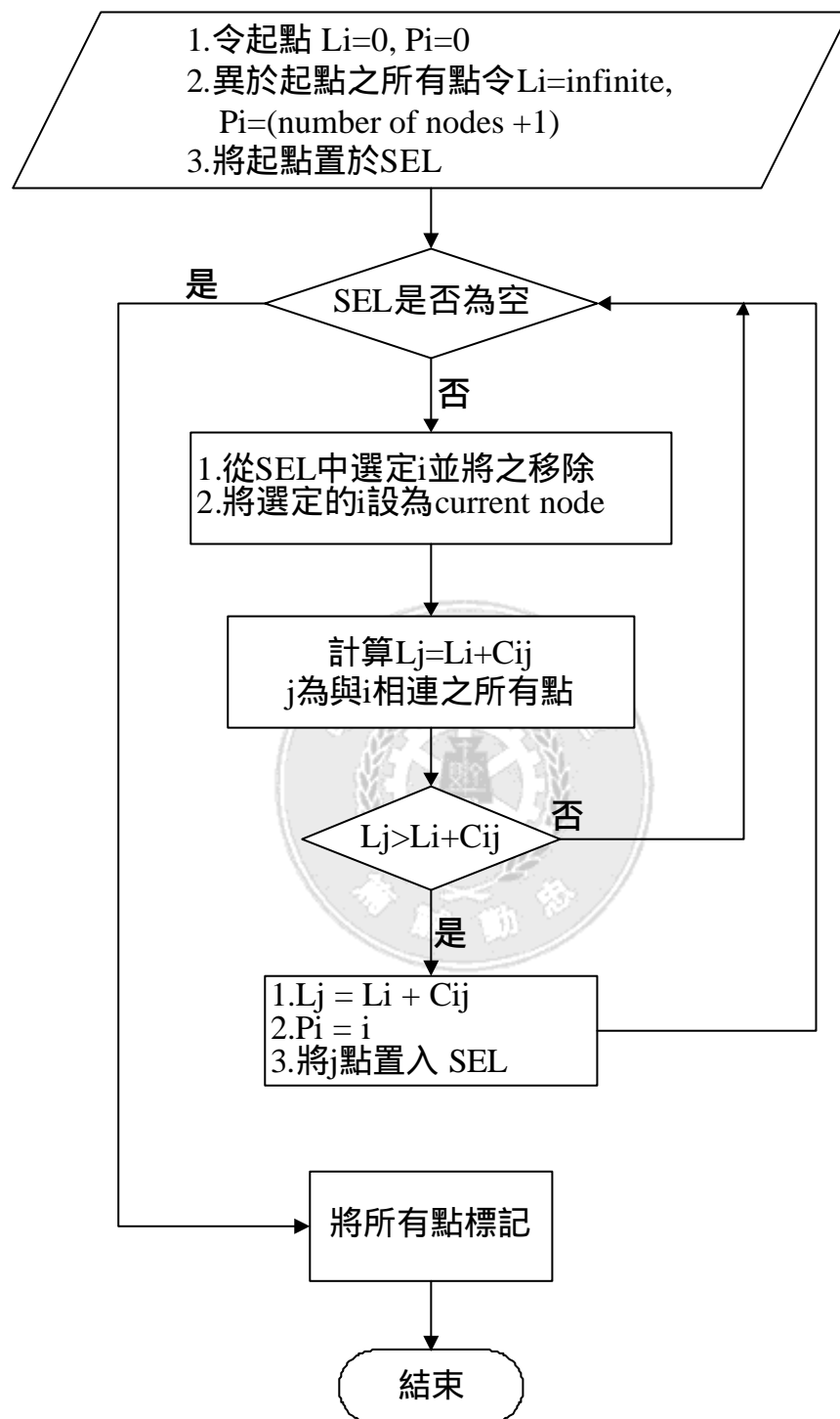


圖 2.6 Label correcting Algorithm 演算流程圖

### 3. Floyd-Warshall Algorithm

Floyd-Warshall Algorithm 是利用矩陣運算計算出 all-to-all 最短路徑演

算法。其演算步驟如下：

Step1：定義  $d_0$  與  $P_0$  的初始值。若  $arc(i,j) \in A$  集合中，則  $d_0$  矩陣中的元素  $d_{ij}$  即等於  $(i,j)$  所給予的值。否則  $d_0 = \infty$  而  $P_0$  矩陣中的元素皆等於  $i_0$ 。令  $k = 1$

Step2：根據下列式子計算  $D^k$  的元素  $d_k(i,j)$

$$d_k(i,j) = \min[d_{k-1}(i,j), d_{k-1}(i,k) + d_{k-1}(k,j)] \quad (2-1)$$

Step4：以下列的式子計算  $P^k$

$$P_k(i,j) = \begin{cases} P_{k-1}(k,j) & , \quad d_k(i,j) \neq d_{k-1}(i,j) \\ P_{k-1}(i,j) & , \quad otherwise \end{cases} \quad (2-2)$$

Step5：當  $k = n$ ，停止，否則令  $k = k+1$  然後回到 Step2。

上述三種常見的演算法，在演算過程中，Label setting Algorithm 和 Label correcting Algorithm 能夠隨著路徑成本的改變，隨時進行更新，相較之下 Floyd-Warshall Algorithm 因為是透過連續的矩陣運算，在更改路徑成本的功能上就顯得較為不足；此外，在計算的時間複雜度上 Label setting Algorithm 為  $O(n^2)$ ，也同樣比 Floyd-Warshall Algorithm 的  $O(n^3)$  為佳，因此 Label setting Algorithm 和 Label correcting Algorithm 就更常被應用在 K 條最短路徑和依時性最短路徑的計算上。

### 2.2.2 K 條最短路徑

最短路徑問題為一般路網問題求解的基礎，但是在實際交通的應用上最短路徑卻會因許多外部問題而無法達到最佳化，因此 K 條最短路徑問題便因應而生，K 條最短路徑主要是在產生成本略多於最短路徑的次佳路徑，此次佳路徑可以是第 2、第 3 條等等 K 條最短路徑以提供用路人更多的選擇，並將所求出之次佳路徑依序排列。Yen 於 1971 年提出找尋第 K 條最短路徑之方法，因此 K 條最短路徑理論也逐漸廣泛的被應用如：人員的緊急救援與疏散等。

而卓訓榮、陳信雄 (1993) 在 CONVEX C240 超級電腦下利用 FORTRAN 程式語言來撰寫 Yen's Algorithm，用以比較純量最佳化、向量最佳化、平行最佳化和起迄點的平行演算，並比較其執行速度。其結果顯示利用平行處理的方式確實可以縮短求解時間。

胡大瀛等人 (2002) 在不同的資料結構下，對於 Label setting Algorithm 和 Yen's Algorithm 進行執行效率比較，所採用的資料結構有陣列 (Array) 形式以及二元累堆 (Binary Heap) 等。根據理論推導使用陣列結構所得到的時間複雜度為  $O(V^2)$ ，使用二元累堆所得到的時間複雜度為  $O((V+E)\lg V)$ 。但在數值實驗中由於其使用的路網較小，因此實驗效果顯著性有限。

上述兩篇文獻在進行  $K$  條最短路徑求解時都是使用常見的 Yen's Algorithm。Yen's Algorithm 的基本概念是：第 2 條最短路徑與第 1 條最短路徑有部分相同，因此第 2 條最短路徑必須經由第 1 條最短路徑來尋找，而第 3 條最短路徑則必須經由第 1 條最短路徑與第  $k-1$  條最短路徑來尋找，依此程序而找出第  $K$  條最短路徑。

其中令  $P(k) = S, X(k, 2), \dots, X(k, qk), T$ ，代表  $S$  到  $T$  的最短路徑； $X(k, 2), X(k, 3), \dots, X(k, qk)$  為此路徑的第 2 點、第 3 點、 $\dots$ ，第  $q$  點； $P(k, i)$  表示一條路徑，此路徑從  $S$  到第  $i$  點與  $P(k-1)$  相同，而從第  $i$  點分離出來到先前所產生的路徑 ( $P(j) (j = 1, 2, \dots, k-1)$ ) 的任何第  $(i+1)$  不同的點； $P(k, i)$  經過一條子路徑後抵達  $T$ ，此子路徑不包含  $S, X(k-1, 2), X(k-1, 3), \dots, X(k-1, i)$ ，稱為  $P(k, i)$  的枝 (spur)，以  $S(k, i)$  表示，路徑節點為  $X(k-1, i), X(k-1, i+1), \dots, T$ 。而  $P(k, i)$  的根 (root)，以  $R(k, i)$  表示，其路徑節點為  $S, X(k-1, 2), X(k-1, 3), \dots, X(k-1, i)$ 。

利用上述演算法求得的  $P(k, i)$  等路徑則放入集合  $L1$ ，再由  $L1$  中挑選出最短路徑，此為第  $k$  條最短路徑，連同之前產生的第  $1, \dots, k-1$  條路徑均放入集合  $L0$  中，如此重覆計算，直到  $k=K$  時才結束。以下則是 Yen's Algorithm 的演算步驟以及程式演算流程圖 (圖 2.7)。

Step 1：找到  $P(1)$ ，令  $k = 2$ ，將  $P(1)$  致入  $L0$ 。假設找到第  $k-1$  條最短路徑，根據  $P(k-1)$  找到所有分離路徑。

Step 2：對所有  $i = 1, 2, 3, \dots, q(k-1)$  執行 Step 3~6 以找出  $P(k-1)$  的所有所有分離路徑。

Step 3：檢查由  $P(k-1)$  的第  $i$  點所成的子路徑是否由任何  $P(j) (j = 1, 2, \dots, k-1)$  的前  $i$  點所組成的子路徑結合成一條，如此則令  $C[X(k-1, i), X(j, i+1)] = \infty$ ，否則不改變其成本。

Step 4：找出不經過  $S, X(k-1, 2), \dots, X(k-1, i)$  各點之最短路徑。

Step 5：將  $R(k, i) (S, X(k-1, 2), \dots, X(k-1, i))$  和  $S(k, i)$  結合成  $P(k, i)$  並放入  $L1$  中

Step 6：將在 Step 3 中之成本陣列還原，回到 Step 3。

選出最短分離路徑：

Step 7：從 L1 中找出最短的路徑，將其放入 L0 中，其為第 k 條最短路徑，若 k 滿足所求的 K 則停止，否則回到 Step 2。

在 Yen's Algorithm 中，須特別注意的是以此演算法進行求解 K 條最短路徑時，在每一段的路徑求解時，仍必須搭配使用最短路徑的演算法才能進行運算。因此，最短路徑演算法的效率將仍會是一重要的因素。

此外，於 1976 年 Shier 提出一個序列式 K 條最短路徑演算法：DoubleSweep Algorithm，其係一個以 Label correcting 為主的 K 條最短路徑演算法，因此，在此演算法中允許路網中存在負成本的路段，但不允許負成本迴圈。DoubleSweep Algorithm 主要的演算法流程下所述。

定義：一路網  $G(V, E, c)$ ， $V$  代表所有點之集合、 $E$  代表所有節線之集合、 $c$  為節線成本的集合；任意兩相鄰節點  $i, j$  之路段旅行時間成本為，路網中所有點對點的成本矩陣  $C = c_{ij}$ ，將  $C$  分成兩個上、下三角矩陣  $L = l_{ij}$ 、 $U = u_{ij}$ 。

Step 1：設定  $v = 0$ ， $d_j^v, \forall j$ 。

Step 2： $v = v + 1$ ， $d_j^v = \lfloor d_j^v \otimes u_{.j} \rfloor \oplus d_j^{v-1}$ ， $j = 1, 2, \dots, n$ 。

Step 3： $d_j^v = d_j^{v-1}, \forall j$ ，到 Step 6。

Step 4： $v = v + 1$ ， $d_j^v = \lfloor d_j^v \otimes u_{.j} \rfloor \oplus d_j^{v-1}$ ， $j = n, n-1, n-2, \dots, 1$ 。

Step 5： $d_j^v = d_j^{v-1}, \forall j$ ，到 Step 6，否則回到 Step 2。

Step 6：停止

洪百賢 (2003) 提出一在 CORBA-Based 分散式架構下的 K 條最短路徑演算法，其演算法基礎為 DoubleSweep Algorithm。由於 K 條最短路徑演算法是動態路網分析中求解的基礎，且求解過程十分複雜，因此為了避免耗費大量的計算時間，其在 CORBA-Based 分散式架構下進行計算，並透過定義的四個績效指標：回應時間、執行時間、錯誤率以及記憶體使用量，藉由數值實驗分析各項指標的變化。

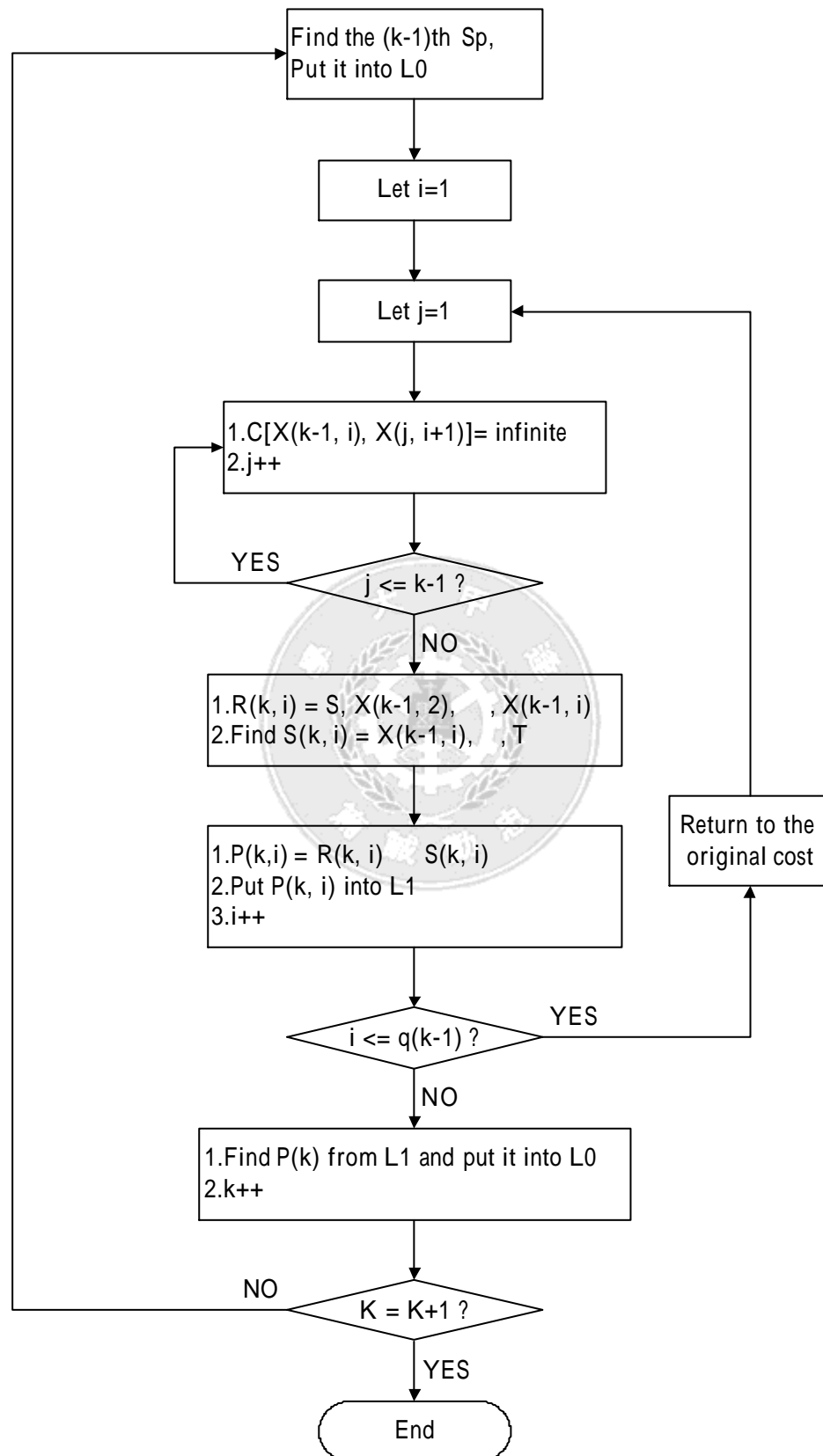


圖 2.7 Yen's Algorithm 演算流程圖

### 2.2.3 依時性最短路徑問題

為了符合現實的交通情形，在最短路徑之後發展了  $K$  條最短路徑，相較於最短路徑， $K$  條最短路徑提供了同一個時間點下的多條路徑選擇，屬於空間上的討論層面。但在依時性最短路徑中，每一個路段的旅行時間成本將隨著時間的變化隨之變化，跟傳統的最短路徑問題比較起來已經從空間層面，跳脫到時間層面了。因此，在面對時間層面的考量下，所面臨的複雜程度也將比一般的最短路徑計算還要困難。

依時性最短路徑問題 (Time-dependent Shortest Path Problem)，與上述最短路徑問題最大的不同，主要在於每一個路段上的旅行時間成本不同。一般的最短路徑問題之路段間旅行成本為固定常數，但在依時性最短路徑問題中，路段間的旅行時間成本會隨著時間的不同而有所差異。

Hall(1986) 為了求解隨機且與時間相依的旅行時間之最小期望旅行時間路徑問題，提出了利用分支定限法 (branch-and-bound) 以及  $K$  條最短路徑演算法來進行求解。在研究中將旅行時間分為「定性」與「隨機變數」兩種，前者定性的旅行時間為蒐集歷史資料進行統計所獲得的定性資料，後者則是利用到達節線時間與滿足統計分配的隨機變數來作為旅行時間的根據。根據定性的旅行時間資料可以求得最小可能時間最短路徑，而根據隨機變數的旅行時間則是求得其最小期望時間最短路徑。

Ziliaskopoulos 與 Mahmassani(1996) 除了考慮時間所造成的路段旅行時間外，並加入考慮轉向所產生的延滯成本，其利用 forward star 的路網結構，並加入轉向時的延滯成本，使其成為具有轉向成本的路網，然後利用修正的 Label correcting 來進行求解，而其路徑時間成本以及轉向成本是根據歷史資料所得。本研究最大的特點為考慮了轉向資料，使其更符合現實社會的交通變化。

Ziliaskopoulos et al.(1997) 介紹一種平行的依時性最短路徑演算法 其所使用的演算法與 Ziliaskopoulos 與 Mahmassani(1996) 相同，不同的僅是在於求解最短路徑時，採用多台的處理器同時進行計算，僅在讀取旅行時間成本時使用共同記憶體 (shared memory)，在此處理過程中，比較有可能出現的問題是在處理 SE list(scan eligible) 上。而在這兩篇文獻中為了簡化其依時性的路段旅行時間成本，其將一天中分成數個時段，定義時間集合  $S$ ，根據交通狀況而產生變化的時間 ( )，分為  $M$  個時間區間，即  $S = \{t_0, t_{0+}, t_{0+2}, \dots, t_{0+M}\}$ ，其中  $t_0$  為任一節點之最早可能出發時間，對一路網上任一節點  $i$  之標記(  $i$ )而言，根據不同時間區段有不同之標記， $i = [i(t_0), i(t_{0+}), \dots, i(t_{0+M})]$ ，其中  $i(t)$  為節點  $i$  在  $t$  時間出發，到達

終點的最短路徑總旅行時間。其依時性最短旅行時間如下所示：

$$I_i[t] = \begin{cases} \min_{j^1} \{I_i[t+d_{ij}[t]]+d_{ij}[t]\}, & \text{for } i=1,2,\dots,N-1; t \in S \\ 0, & \text{for } i=N; t \in S \end{cases} \quad (2-3)$$

陳慧琪 (1999) 將旅行時間視為一個滿足威伯分配 (Weibull distribution,  $W(1, \quad)$ ) 之連續型隨機變數，令  $I_i[t]$  為隨著出發時間而改變的時間函數，並將每一天分為數個時段，自行假設每條節線在各個不同時段所相對應之期望值函數，透過模擬方法，得到滿足該分配的旅行時間模擬值，再以 Label setting Algorithm 為基礎進行修正，進行求解。

江文聲 (2001) 利用工程系統可靠度評估方法，以可靠度指標進行路徑選擇，其採用 K 條最短路徑演算法及 DoubleSweep Algorithm 進行計算，透過 DoubleSweep Algorithm 在隨機路網及 FIFO 動態隨機時間相依路網，同時求解最短旅行時間路徑以及最可靠路徑，以滿足用路者需求，此外還能解決計算 K 條最短路徑時易產生的時間過長問題。另外，在路口轉向限制方面，其係在假設路網  $G(V, A, J, T)$  中以利用簡單的處理，建立一個子集合  $T(n,a)$ ，而此子集合即是以禁止左右轉之資料所構成。

經由以上的回顧可以發現根據時間資料的使用上，可分為兩種情形，一種是以統計分配方式獲得每個時間點下的旅行時間，屬於連續型 (continuous)；另一種則是透過資料蒐集或其他方式，將時間以時段來區分，依據每個時間的所屬區段來獲得所需的旅行時間，屬於離散型 (discrete)。

#### 2.2.4 考慮路口延滯之最短路徑問題

Chen 和 Yang(2000) 主要是透過時窗限制式 (time windows constraints) 的方式來計算路口號誌所產生的延滯，利用該限制式將每個路口的通行方向切割出來，當最短路徑演算法計算到某一路口時，便可以將其最早離開時間計算出來並指出該離開時間所通行的方向。該最短路徑演算法為修正的 Label Setting Algorithm，配合此號誌限制式所得到的時間複雜度為  $O(m^3)$ 。以下介紹其所使用的路網及演算法。



定義：一含有交通號誌的路網  $N = (V_1, V_2, A, WL, t, s, d)$ ， $V_1$  為無時窗限制下的節點集合， $V_2$  為時窗限制下的節點集合， $A$  為路段集合且這些路段並無多重路段 (multiple arcs) 及自我迴圈 (self-loop)； $t(u, v)$  為路段  $(u, v) \in A$  的旅行時間成本，時窗列表 (windows-list)  $WL(u) = (ws_u, w_{u,1}, w_{u,2}, \dots, w_{u,r})$ ， $ws_u$  為開始的第一個時窗， $w_{u,i}$  為節點  $u$  的第  $i$  個時窗。

函式定義：

$arrived(v, u)$ ：經由路段  $(v, u)$  抵達節點  $u$  的最早抵達時間；

$leaving(v, u, w)$ ：從路段  $(v, u)$  到路段  $(u, w)$  通過節點  $u$  的最早離開  $u$  的時間；

$P^*(v, w)$ ：經由路段  $(u, w)$  到節點  $w$  的最短路徑；

$pre(u, w) = (v, u)$ ：經由  $u$  到  $w$  的路徑的前一個路段為  $(v, u)$ ；

$earliest(v, u, w, t)$ ：計算在  $t$  時間下，由  $(v, u)$  通過節點  $u$  到達路段  $(u, w)$  的最早離開  $u$  時間；

函式間的關係：

$$leaving(v, u, w) = earliest(v, u, w, arrived(v, u))$$

$$arrived(u, w) = \min_{\text{for all } v} \{leaving(v, u, w) + t(u, w)\}$$

演算法為：

1. Set  $arrived(0, s) = 0$   
 Set all  $arrived(v, u) = \infty$  for all arcs  $(v, u)$  in  $A$   
 Insert all values of  $arrived(v, u)$  into the set  $HP$
2. Find and remove the minimum element  $arrived(v, u)$  from  $HP$
3. If  $u = d$  then go to Step 5
4. For each arc  $(u, w)$  emanating from node  $u$ , do  
 Begin  
    $leaving(v, u, w) = earliest(v, u, w, arrived(v, u))$   
    $temp(u, w) = leaving(v, u, w) + t(u, w)$   
   If  $temp(u, w) < arrived(u, w)$  then  
      $arrived(u, w) = temp(u, w)$ ,  $pred(u, w) = (v, u)$ , and  
     Update the value  $arrived(u, w)$  in  $HP$   
 End  
 Go to Step 2

5. From  $pred(v, u)$  we find the shortest path  
Output  $arrived(v, u)$  as the minimum time

在這篇文獻中利用「最早離開時間」來作為選擇路徑往下游前進的方向，並使其抵達的旅行時間成本最小。這樣的考慮方式在一般的路徑計算上是很少使用到的，因此其所求解出來的結果在實際交通應用上仍待討論。

Chen 和 Yang(2003) 主要研究在於具有交換時間 (time-switch) 限制的網路上找尋最短路徑，研究中在路口加入 on 或 off 的狀態來模擬做 go 或 wait，利用這個方式建構出一個 on-off time-switch 的路網。此外，研究中也考慮了加權平均的停止次數 (weighted number of stops)。根據研究的結果，找尋 traffic-light network 雙目標 (bi-criteria) 的最短路徑的時間複雜度為  $O(\#Wn^3)$ ，其中  $n$  是網路節點的數目， $\#W$  是已知的加權平均停止次數。

### 2.2.5 路網資料結構

在最短路徑運算的過程中，除了演算法本身外，另一個會影響系統運算效率的便是所使用的路網資料結構。一般常見的路網資料結構有下列幾種：(江文聲，2001)

#### 1. 權數矩陣 (weight matrix)

此為路網結構中最為簡單的一種資料結構，其處理路段旅行時間的方式是利用陣列的形式來儲存，陣列中的每一個元素代表每個路段的旅行時間。但在交通路網中，並非每個節點都與其他節點相連接，因此容易形成所謂的稀疏矩陣，如此一來，不但浪費系統記憶體空間，在運算效率上也非常不佳。

#### 2. 相鄰串列 (Node Adjacency-list)

利用相鄰串列可以將路網中與某一節點相連的所有節點，利用串列 (linked) 的方式串接在一起。因此，路網中若有  $n$  節點，將會形成  $n$  個串列存在。而在相鄰串列結構中的每個節點連結屬性可以分成三個主要部分：節點、路段資料以及指標。在路段資料部分，不僅僅儲存路段旅行時間，也可以儲存其他與路段相關的屬性資料，對於描述一完整路網來說相

當便利。

### 3. 星結構 (star structure)

星結構的資料形式由三個陣列所組成，三個陣列分別儲存指標、起點與迄點。而依據儲存的方式星結構可分為兩種：向前星法 (forward star) 以及向後星法 (backward star)。兩種結構的差別在於起點與迄點的位置不同。由於在星結構中主要是利用指標來建立起點與迄點的關係，因此在程式設計上只需要指標便能代表路網的結構。而星結構的主要缺點在於修改成本時較為不方便。

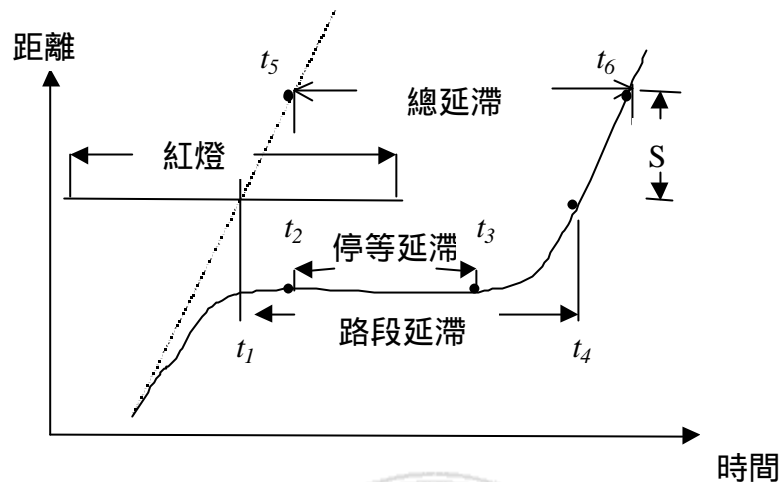
## 2.2.6 路口延滯成本

過去的最短路徑的計算中多未將路口延滯成本列入計算，然而在服務水準績效指標中「延滯」卻是一項不可忽略的重要因子，尤其是發生在路口號誌控制下所產生的延滯最為明顯。

在服務水準績效指標中多採用平均延滯時間 (average delay) 來描述。平均延滯時間係指車輛因為減速、停等以及加速時所增加的平均旅行時間。在號誌化路口中則又可以分為平均停等延滯時間 (average stopped delay)、平均路段延滯 (average approach delay) 以及平均總延滯時間 (average total delay)，此三種延滯的定義可用圖 2.8 說明之。此外，由於平均停等延滯最容易透過所蒐集的現場資料中計算求得，所以交通界多以平均停等延滯最為評估號誌化路口的服務水準。(交通部運輸研究所，2001)

圖 2.8 是假設一車子不受到其他車輛或者是行人以及號誌的干擾時，車尾在時間  $t_1$  時通過交叉路口停止線附近之一參考線。如遇到紅燈時，則該車必須減速和停等，然後在綠燈的時候通過該參考線。若是因紅燈或其他車輛的阻擋而使其在時間  $t_2$  時加入該停等車隊，而在時間  $t_3$  時因車輛疏解而開始加速，此時其停等延滯為  $t_3 - t_2$ 。若該車車尾通過參考線之瞬間為時間  $t_4$ ，則其路段延滯為  $t_4 - t_1$ 。該車通過參考線後繼續加速，直到在時間  $t_6$  時達到穩定的自由旅行速率。若在時間  $t_6$  時該車抵達參考線下游 S

公尺，而在無其他行人或車輛的干擾下，該車到達同一點之瞬間為時間  $t_6$ ，則總延滯為  $t_6 - t_5$ 。



資料來源：交通部運輸研究所，2001

圖 2.8 延滯之定義

而根據 1998 年的美國公路容量手冊 (Highway Capacity Manual) 中首次定義，舉凡因號誌控制而產生的延滯均改稱為控制延滯 (control delay)，理論上，控制延滯應該不能包含因為幾何設計以及轉彎所造成的延滯，但目前美國公路容量手冊中將總延滯以控制延滯作為代表。

根據我國交通部運輸研究所所制定的公路容量手冊 (交通部運輸研究所，2001) 所示，目前尚未有適合我國車流特性的分析模式來估計控制延滯，因此，其參考美國 1985 年的公路容量手冊將號誌控制所造成的平均延滯以下式來計算：

$$d = 0.494p \frac{C(1 - \frac{g}{C})^2}{1 - \frac{g}{C}Z} + 224.9X^2 [X - 1 + \sqrt{(X - 1)^2 + \frac{16X}{c}}] \quad (2-4)$$

式中的各項變數如下所述：

$d$  = 平均延滯 (秒/輛)；

$p$  = 延滯調整因素；

$C$  = 週期長度 (秒)；

- $g$  = 有效綠燈長度 (秒) ;  
 $X$  = 流量/容量比 (流量 = 需求流率  $Q$ ) ;  
 $Z = X$  或 1.0 , 取較小值 ;  
 $c$  = 幹道直行車道總容量 (輛/小時)。

在 Dion et al.(2004) 的研究中指出延滯的估算對於評估號誌化路口的服務水準有其重要性，而各界致力發展的延滯相關模式也呈現多樣化，因此在其研究中將目前較為常見的幾個延滯估計模式進行整理比較。研究中利用 INTEGRATION 微觀交通模擬軟體進行模擬比較，比較的式子中主要有估算等候線長度模式 (Deterministic queuing model)、衝擊波延滯模式 (Shock wave delay model)、穩定狀態隨機延滯模式 (Steady-state stochastic delay models)、未飽和及過飽和限制下的依時隨機延滯模式 (Time-dependent stochastic delay models) 以及微觀模擬延滯模式 (Microscopic simulation delay models) 等。研究結果顯示，在較低的交通需求下，各個模式所產生的結果差異不大，但在當交通需求增加時，各模式之間的差異也越來越明顯。

## 2.3 物件導向分析與程式

物件導向觀念的出現，為程式發展世界帶來一片新的思維，其是有別以往架構式的思考觀念，而採用物件式的思維。而本研究也採用物件導向的觀念來進行系統分析與程式撰寫。

### 2.3.1 物件導向分析與設計

分析 (analysis) 是把焦點放在調查問題與需求上，而不是提供解決方案。設計 (design) 把焦點放在滿足需求的概念性解決方案 (conceptual solution) 而不是在實作上。進行物件導向分析 (object-oriented analysis) 時，會把焦點放在找出並描述問題領域的物件 (或概念) 上。而在進行物件導向設計時，則是把焦點放在定義軟體物件，並且還要知道他們之間是如何合作以滿足彼此的需求。(趙光正，2002)

在真實世界的所有物體都是以物件的形式展現出來，並且透過物件的方式進行處理運作。而物件導向分析方法主要有五個步驟：(趙光正，2002)

1. 主體層：給予被分析的模型一個整體性的概觀。
2. 物件層：反映系統在現實世界中處理事務的訊息能力。
3. 結構層：將問題範圍的複雜性表現出來。
4. 屬性層：藉由屬性來表現物件特有的資料。
5. 方法層：在蒐集訊息後，利用方法實行一些處理動作。

於物件導向設計方法中，以往注意的焦點已經從問題空間轉移到理解空間。根據所設計的系統來描述其邏輯與實體過程，以及系統靜態和動態的設計方法。透過物件導向可以讓系統製作週期縮短、提昇系統維護性、具較佳的資訊結構和適應性。

在使用物件導向的過程中通常會有下列幾個動作：(鄭家瑜，2000)

#### 1. 定義使用案例

使用案例 (use case) 是用來描述領域相關流程之用，它把敘事情節寫出來，其在需求分析中相當普遍也是常用的工具。

#### 2. 定義領域模型

物件導向分析就是把物件類別化以產生領域描述，於其中找出須注意的概念、屬性與關聯，在將他們表現在領域模型之中，產生表達領域概念或物件的圖。

#### 3. 定義互動圖 (Interaction Diagram)

物件導向設計中需要定義物件及物件之間的合作關係，而最常用的表示方法即是利用互動圖，在互動途中會展現各物件之間的訊息傳遞，亦即呼叫方法。

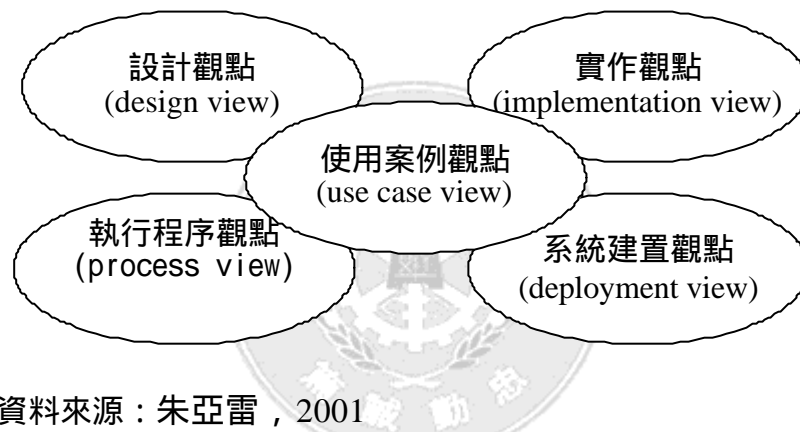
#### 4. 定義設計類別圖 (Class Diagram)

在動態觀點中利用互動圖定義了物件之間的合作關係，亦會用設計類別圖來展示類別定義的靜態觀點，在設計類別圖中，將展示類別的屬性以及方法之間的差異。

物件導向分析與設計方法在 1994 年就已經存在了，但當初並未訂定統一的分析設計方法，直到 1997 年才制定出一套統一塑模語言 (Unified Modeling Language, UML)，自此，物件導向分析設計才有標準化的方法。

UML 是現行塑模語言的標準，所謂塑模語言是一種用來表現模型的工具，是一群語意以及語法的集合，主要的功能就是讓開發人員之間，透過圖形化的方式來進行溝通。

一個系統在 UML 中被分為五個觀點來看，包括：使用案例觀點 (use case view)、設計觀點 (design view)、執执行程序觀點 (process view)、實作觀點 (implementation view) 以及系統建置觀點 (deployment view)，如圖 2.9。其中每一個觀點都代表用一個角度來檢視系統，其中用使用案例跨越其他四個觀點，也就是說其他四個觀點都必須符合使用案例的需求，因此其又稱為 4+1 觀點的軟體架構。(朱亞雷，2001)



資料來源：朱亞雷，2001

圖 2.9 4+1 觀點的 UML 軟體架構

在 UML 中也定義了各種圖形的表示法，其中包括：使用案例圖 (use case diagrams)、循序圖 (sequence diagrams)、合作圖 (collaboration diagrams)、狀態圖 (state diagrams)、活動圖 (activity diagrams)、類別圖 (class diagrams)、物件圖 (object diagrams)、元件圖 (component diagrams)、部署圖 (deployment diagrams)。這些圖形是由相關聯的事物與其間的關係所繪製而成，透過這些圖形也就可以表現出各個觀點的內容。因此，UML 實為物件導向分析設計的實用工具。

### 2.3.2 物件導向程式設計

以往的程式語言是以結構化程式為主，而如今已經擴充到以物件導向

式的程式設計為主要觀念。在物件導向程式設計中，並不是捨棄過去的結構化程式，而是保留其原本精神，並加入新的型態與方法，讓程式在設計上及維護上更顯的容易。而物件導向程式中的基礎：類別 (class)，更是了解物件導向程式設計特型時最重要的型態。

就程式設計而言，有兩種主要的方式，一為結構化程式設計的重點，即是只有透過程式才能對資料作改變 (around the code)，在這裡資料跟程式是分開的；另一個則是物件導向程式設計的特色，也就是程式碼的存取是被資料所控制著 (data controlling access to code)，只有利用所定義的資料與程式 (函式) 才可以對資料作存取，在這裡資料與程式是並存的。(鄭家瑜，2000)

物件導向程式設計中有三個特性，封裝 (encapsulation)、多型 (polymorphism) 與繼承 (inheritance)，以下將就此三特性進行簡單說明。(鄭家瑜，2000)

### 1. 封裝 (encapsulation)

封裝是一種將資料 (data) 與程式碼 (code) 整合在一起的一種機制，以確保安全與避免來自外界的誤用。在物件導向程式設計中，資料與程式就是以封裝的形式聯結在一起而建立起物件。

### 2. 多型 (polymorphism)

多型是指一種介面有很多種方法，簡單來說就是以單一介面來控制存取一般類別資料的行為。利用多型可以減少程式的複雜度，並整合相類似或相同的功能附以統一的介面 (函數名稱)，來完成不同的工作。

### 3. 繼承 (inheritance)

繼承是一種承襲的過程，亦即一個物件除了可具有另一個物件的特性外，自己也能具有自己的特性，而自己也可以讓另一物件所繼承。而此繼承的特性更是物件導向程式設計的重要觀念。

類別與物件的關係是類別為產生物件的模板，透過類別可以將具有相同特點的物件集合在一起，再經由封裝、多型與繼承的處理特性為基礎而產生物件導向程式。因此，物件導向可以說是以類別為基礎的程式設計。



## 2.4 綜合分析

本章就智慧型運輸系統、最短路徑問題、物件導向分析與程式等三項重點進行相關文獻回顧。智慧型運輸系統為目前交通界中致力發展的目標，而路徑的計算在其中的先進交通管理系統以及先進旅行者資訊系統中扮演著相當重要的角色。管理者透過即時資料的蒐集研擬交通策略，透過路徑的計算提供給不同需求的用路者最佳的路線選擇，而用路者也就由這些資訊來降低其旅行成本、提昇運輸效率。而本研究所發展出來的依時性最短路徑演算法，不僅可以符合時間變化的需求，也可以針對路口控制延滯等相關變化進行最佳路徑的尋找，此一精神更能符合即時且動態的智慧型運輸系統發展。

最短路徑演算法是路網研究中極為重要的一部份，除了基礎求解最短路徑的計算方式外，對於考慮滿足實際交通應用中不可預期的外部問題，K 條最短路徑則提供用路人更多樣的選擇，而考慮不同時間下的旅行成本的依時性最短路徑問題在近幾年內更是蓬勃發展，其可以在動態路網分析中得到更符合實際情況的最佳解。而在計算控制延滯方面，由於有許多中可供參考的延滯模式，因此在研究中也將找出一個適合用於本研究所使用的交通模擬軟體，以方便產生計算路口延滯下最短路徑所需的延滯成本。

物件導向分析為新興的分析方式，透過物件導向的方式可以對系統更進一步的定義與說明，讓使用者與系統管理者之間的介面更明顯也更清楚，在程式開發中也可以藉由分析所得的各種類別資料更快的完成程式撰寫，透過物件的方式也能讓程式架構更輕易的管理維護。

## 第三章 路口延滯下最短路徑演算法

在本章中針對本研究所要探討的成本計算方式以及每一種最短路徑演算法分別進行模式與演算法建構。在成本計算方面說明本研究的計算與考量方式；而接著的路徑演算法，則分別說明其問題定義、演算法發展以及演算法的合理性。第 3.1 節說明本研究所採用的路段旅行成本及路口延滯成本模式；第 3.2 節建構路口延滯下最短路徑演算法；第 3.3 節建構路口延滯下 K 條最短路徑演算法；第 3.4 節則是建構路口延滯下依時性最短路徑演算法；第 3.4 節小型範例介紹；第 3.5 節綜合討論。

### 3.1 路段與路口成本

在最短路徑的計算中，主要的成本考慮可分為路段距離與路段旅行時間。因此，在最短路徑問題中，首先要知道的便是成本的描述。目前市面上所使用的 GIS 軟體中，其用來計算最短路徑的路段成本，多係採用路段的距離長度作為主要成本考量，因為路段距離資料與旅行時間資料相比之下較容易獲得，所以以其作為路段成本較為簡便。但是在實際交通行為中旅行距離最短，並不代表旅行時間也能跟著縮短，而且使用路段長度為路段成本，並不能有效的反應出路段上的交通變化情形，因此，為了能夠將路段上的交通變化反應在成本上，必須採用時間來作為路段成本，因為時間會受到路段上車流、號誌等交通情況所影響，因此，本研究主要是以考量旅行時間為成本的最短路徑問題。

在以時間來作為最短路徑成本時，所面對的成本主要是路段旅行時間成本、路段及路口延滯時間成本。而在最短路徑中所討論的路段旅行時間成本，主要的計算方式是用路段距離除以速率來求得，但對於延滯時間成本而言，卻有著許多不同的影響因子，因此在計算上及使用上也較為複雜及多樣化，而其中較常見也是影響最大的莫過於通過路口時所產生的路口延滯時間成本。所以本研究也會針對路口所產生的相關延滯成本進行討論與計算。

### 3.1.1 路段旅行時間成本

為了能夠滿足各種路徑計算演算法所需，本研究必須獲得各個時段下的路段旅行時間成本。做法主要是利用交通模擬軟體 DYNASMART，依據依時性 OD 模擬計算不同時間的路段平均速率，並搭配路段長度資料，來計算通過該路段的旅行時間，以供本研究使用。

由於利用模擬的方式可以獲得許多不同時段的旅行速率資料，因此，當要開始進行傳統的最短路徑及 K 條最短路徑計算時，就必須將起始時間決定，利用該時間來決定整個路徑計算所會使用到的路段旅行速率，進而求得其旅行時間成本。而在進行依時性最短路徑計算時，時間變數將會隨著路口節點的更新來決定要取哪一個時段的路段旅行速率資料，來計算相對應的路段旅行時間成本。

過去在進行依時性最短路徑計算時，取得路段旅行時間的方式常見的有兩種，一為透過調查的方式來蒐集旅行時間資料 (Hall, 1986、Ziliaskopoulos, 1996)，此種做法因為每個時間點的路段旅行時間資料取得不易，因此會將所蒐集到的資料依時段進行調整，以方便其模式計算；而另一種方式則是利用統計分析方法，將旅行時間視為滿足一統計分配的方式來計算求得 (Hall, 1986、陳慧琪, 1999)。而本研究主要乃透過系統模擬的方式，模擬出每個時段下不同交通、車流情形的路段旅行時間，以求得更精準的路段旅行時間資料，使所求的依時性最短路徑更能符合實際情形。

### 3.1.2 平均路口延滯成本模式

在路口延滯中，影響最大的就是轉向延滯以及號誌所引起的停等延滯。而根據 1998 年的美國公路容量手冊中首次定義的控制延滯，其中已經包括了所有路口的相關延滯。換言之，控制延滯可以將號誌化路口所產生的停等延滯、轉向延滯等相關延滯成本一起計算。因此，本研究乃決定假設所有通過路口的延滯皆由號誌控制影響而產生。

就交通部運輸研究所(2001)以及 Dion et al.(2004)的研究中比較，本研究乃決定採用交通部運輸研究所所制定的公路容量手冊中的控制延滯模式來當作本研究路口號誌影響的延滯成本。但由於目前尚未有適合我國車

流特性的分析模式來估計控制延滯，因此，交通部運輸研究所乃參考美國公路容量手冊制定下式來進行計算：(交通部運輸研究所，2001)

$$d = Fd_1 + d_2 \quad (3-1)$$

式中的各項變數如下所述：

$d$  = 平均停等延滯時間 (秒/輛)；

$F$  = 車流續進及號誌控制種類調整因素；

$d_1$  = 車輛以均勻車距 (constant headway) 抵達交叉路口，而且每週期到達之車輛皆可在該週期疏解之情況下的平均停等延滯時間 (秒/輛)；

$d_2$  = 因車輛抵達交叉路口之車距有隨機變化及每週期到達之車輛可能因壅塞而不能在該週期疏解而造成的額外平均停等延滯 (秒/輛)。

其中，

$$d_1 = \frac{0.38C \left(1 - \frac{G_e}{C}\right)^2}{1 - \frac{G_e}{C} \text{Min}(1.0, X)} \quad (3-2)$$

$$d_2 = 173X^2 \left( X - 1 + \sqrt{(X - 1)^2 + \frac{mX}{c}} \right) \quad (3-3)$$

此兩式中，

$G_e$  = 有效綠燈之秒數

$C$  = 週期長度(秒)；

$c$  = 車道或車道群之容量(輛/小時)；

$X$  = 流量/容量比；

$\text{Min}(1.0, X) = 1.0$  及  $X$  之較小值；

$m$  = 車輛到達型態之調整因素。

為便於本研究使用，在上述延滯計算公式中假設路網中的車輛到達型態為隨機到達，且存在於獨立（沒連鎖）之交叉路口，經查詢公路容量手冊，得到  $m = 16$ ；定時號誌控制之調整因素  $F = 1.0$ 。另外，從式子中可以得知延滯時間與流量與容量比 ( $X$ ) 成正比，因此，為了避免給予的  $X$  值過高因而導致路段車流過多使得延滯時間過長，或是  $X$  值過小導致車流的影響不夠明顯，因此假設  $X = 0.5$ 。

經過上述參數假設及整理得到下式：

$$d = \frac{0.38C \left(1 - \frac{Ge}{C}\right)^2}{1 - \frac{Ge}{2C}} + 43 \left( \sqrt{\frac{1}{4} + \frac{8}{c}} - \frac{1}{2} \right) \quad (3-4)$$

另外，在公式中的車道或車道群容量 ( $c$ )，可以利用傳統計算容量的公式  $c = S \times \frac{Ge}{C}$  求得，其中  $S$  為飽和流率。因此，雖然在式 3-2 以及式 3-3 中的  $X$ ，本研究中將其假設為 0.5，而導致無法藉由  $X$  來反映不同大小道路的差異，但透過  $c$  的數值，同樣可以反映出大小道路容量上的差異。

本研究在最短路徑演算法、K 條最短路徑演算法以及依時性最短路徑演算法中，乃加入每個路口的控制延滯來進行更新標號值的計算。

### 3.1.3 車流量對路口延滯成本的影響

在傳統車流分析理論中，對於路口鄰近路段停等車流的紓解，常會忽略可能受到下游路口車輛等候的影響，而視為可以在有效綠燈時間內以飽和流率來進行紓解。但在實際交通情況中，即使是綠燈時相，也有可能因為突發狀況或交通擁擠導致路口車輛擁擠，使其紓解率降低。因此，車流量在路口延滯中不能不加以考慮。為了反映出路口容量的限制，過去在 DYNASMART 中，曾採用平均紓解率 (Average Discharge Rate) 來作為延滯時間估算的主要變數。其描述如下所示(參考圖 3.1)。

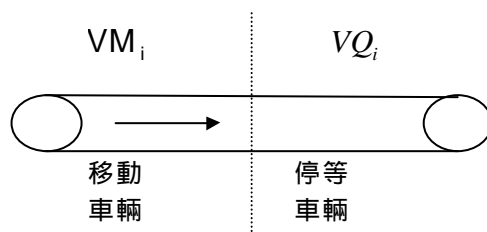


圖 3.1 延滯時間估算示意圖

$$\text{延滯時間 } d = \frac{VQ_i}{AS_i} ; \quad (3-5)$$

$$AS_i (\text{平均紓解率}) = \frac{\sum_{m=1}^{t+T} \sum_{j \in OUTBOUND} AVM_{ij}(m)}{T} ; \quad (3-6)$$

$VQ_i$ ：路段  $i$  的等候車輛數；

$T$ ：計算紓解率的時段長度；

$AVM_{ij}(m)$  = 在  $m$  的時段中，實際由  $i$  移至  $j$  的車輛總數。

因此，在考慮車流量對路口延滯時間的影響時，便可以利用此一公式來估算其路口延滯時間。

## 3.2 最短路徑演算法

最短路徑演算法為所有路徑演算法的基礎，因此，不管是在進行  $K$  條最短路徑或是依時性最短路徑，皆必須從此處開始進行。表 3.1 說明了路徑計算演算法考慮因素的變化情形以及本研究的考慮範圍，從原先單純的空間變化上，如從最短路徑擴展至  $K$  條最短路徑以及加入了其他成本的考量。而後來增加了時間因素，因而演進到依時性最短路徑。也因為討論層級的不斷提昇，使得其在運算上更趨複雜。

表 3.1 本研究考慮範圍

路徑演算法	空間上的因素		時間上的因素
	路徑數目	路口延滯	
最短路徑演算法	1	√	×
K 條最短路徑演算法	k	√	×
依時性最短路徑演算法	1	√	√

因此，本研究也先從固定旅行時間成本的最短路徑演算法開始加入路口延滯的考量，再循序發展所需的依時性最短路徑演算法，以達到所訂定的目標。

### 3.2.1 問題定義

在本研究中最短路徑問題的基本定義如下：一路網  $G(V, E)$ ， $V$  代表所有點之集合、 $E$  代表所有節線之集合；任意兩相鄰節點  $i, j$  之路段旅行時間成本為  $C(i, j)$ 、路口延滯成本為  $d(i, j)$ ，給定一起點  $S$ ，並考慮所有可能之因素，找到由該起點至路網中所有節點的路徑，使每條路徑的總旅行時間成本皆為最小。

因此，在進行路口延滯下最短路徑時，系統必須提供其所需的路網資料、一組同一時段下的路段旅行時間成本以及路口延滯成本，以利計算獲得一對多 (one-to-all) 的路口延滯下最短路徑。

### 3.2.2 求解演算法

最短路徑演算法中，較常見的最短路徑演算法為 Label setting Algorithm(LSA) 以及 Label correcting Algorithm(LCA)。兩者的演算法都是先將出發節點的成本設為 0，路網中所有其他節點的成本設為無限大，再由出發節點開始搜尋下游節點並將其置入暫存集合中，接著從暫存集合中

選取節點進行節點成本更新動作，直到全部路網都處理完為止。

而 LSA 與 LCA 的不同之處在於從暫存集中選取節點的方式，LSA 在選取節前會先將暫存集合裡面的節點成本值進行比較，然後選出成本最小的節點來進行後續計算；而 LCA 則是利用先進先出 (First In First Out, FIFO) 的概念，從暫存集中最前面的節點開始選取節點進行計算，此為兩種演算法最大的不同之處。

另外，在時間複雜度上，雖然 LSA 的時間複雜度  $O(n^2)$  與 LCA 的時間複雜度  $O(n^3)$  比較之下前者較佳，但由於前者在選取節點時，必須從暫存集中選擇節點的成本標號值最小值，而此選擇過程對於整個演算的時間耗費影響甚大，若其中節點數增多，其運算效率將會降低許多。相較之下由於 LCA 在選取節點時採 FIFO 的方式，所以不需在選取節點時增加系統額外負擔。不過雖然 LCA 不需要在選取節點時先行比較每個節點標號值的大小，但因為其演算法的特性，使其路網中每個節點都會重覆進出暫存集合多次，同樣會增加系統的負擔。因此，若只從演算法的討論上來選擇較不客觀，所以本研究在進行路口延滯下最短路徑的運算時將先採用 LCA 來作為發展的基礎，而在後續的路口延滯下 K 條最短路徑計算時，將選用 LSA 作為搭配的最短路徑演算法。並在後續討論上針對兩者進行比較與分析。

而在本研究中，乃係針對二種路口延滯成本進行計算比較分析，分別為以號誌延滯為主的路口延滯成本以及車流量影響下的路口延滯成本，其中號誌延滯將以前述公路容量手冊所制定的控制延滯為計算模式，而車流量影響的延滯成本則是以平均紓解率作為計算模式。由於研究乃係考量路口延滯成本，因此必須在演算法中的節點更新標號值時，加入此一延滯成本，而修改後的 LCA 程式流程圖如圖 3.2，詳細求解步驟如下所述：

參數定義：

路網  $G(V, E)$ ：V 代表所有點之集合、E 代表所有節線之集合；

S：起點；

$C(i, j)$ ：任意兩相鄰節點 i、j 之路段旅行時間成本；

$L(i)$ ：累積之旅行時間成本標號值；

$PreNode(i)$ ：點 i 之上游節點；

$CurrentNode$ ：處理中之節點；

$Temp(i)$ ：暫存之節點 i 標號值；



$SEL$ ：待選取節點之集合；

$d(CurrentNode, i)$ ：節點  $CurrentNode$  的路口延滯成本；

$dL(CurrentNode, i)$ ： $CurrentNode$  的累積旅行時間成本標號值加上  $CurrentNode$  到  $i$  路口延滯成本；

Step 0：設定所有點之初始值：起點  $S$  之標號值  $L(S) = 0$ ，起點  $S$  之上游節點  $PreNode(S) = S$ ，路網上所有點  $i (i \neq S)$  之標號值  $L(i) = \infty$ ，點  $i$  之上游節點  $PreNode(i)$  為最大節點編號+1。

Step 1：將  $S$  置入  $SEL$  中。

Step 2：檢查  $SEL$  是否為空集合，

是，到 Step 8；

否，到 Step 3。

Step 3：依據 FIFO 的規則從  $SEL$  中選一點  $i$ ，設  $CurrentNode = i$ ，從  $SEL$  中移除  $i$ 。

Step 4：計算  $CurrentNode$  到  $i$  的路口延滯成本  $d(CurrentNode, i)$ ，並更新  $CurrentNode$  到  $i$  之起始標號值  $dL(CurrentNode, i) = L(CurrentNode) + d(CurrentNode, i)$ 。

Step 5：取得所有與  $CurrentNode$  相連之下游點  $i$  之路段成本  $C(CurrentNode, i)$ ，並計算  $Temp(i) = dL(CurrentNode, i) + C(CurrentNode, i)$ 。

Step 6：檢查  $L(i)$  是否大於  $Temp(i)$ ，

是，到 Step 7；

否，回到 Step 2。

Step 7：更新點  $i$  之標號值  $L(i) = Temp(i)$ ，點  $i$  之上游節點為  $CurrentNode$ ， $PreNode(i) = CurrentNode$ ，將點  $i$  置入  $SEL$  中，到 Step 2。

Step 8：將所有點標記，結束。

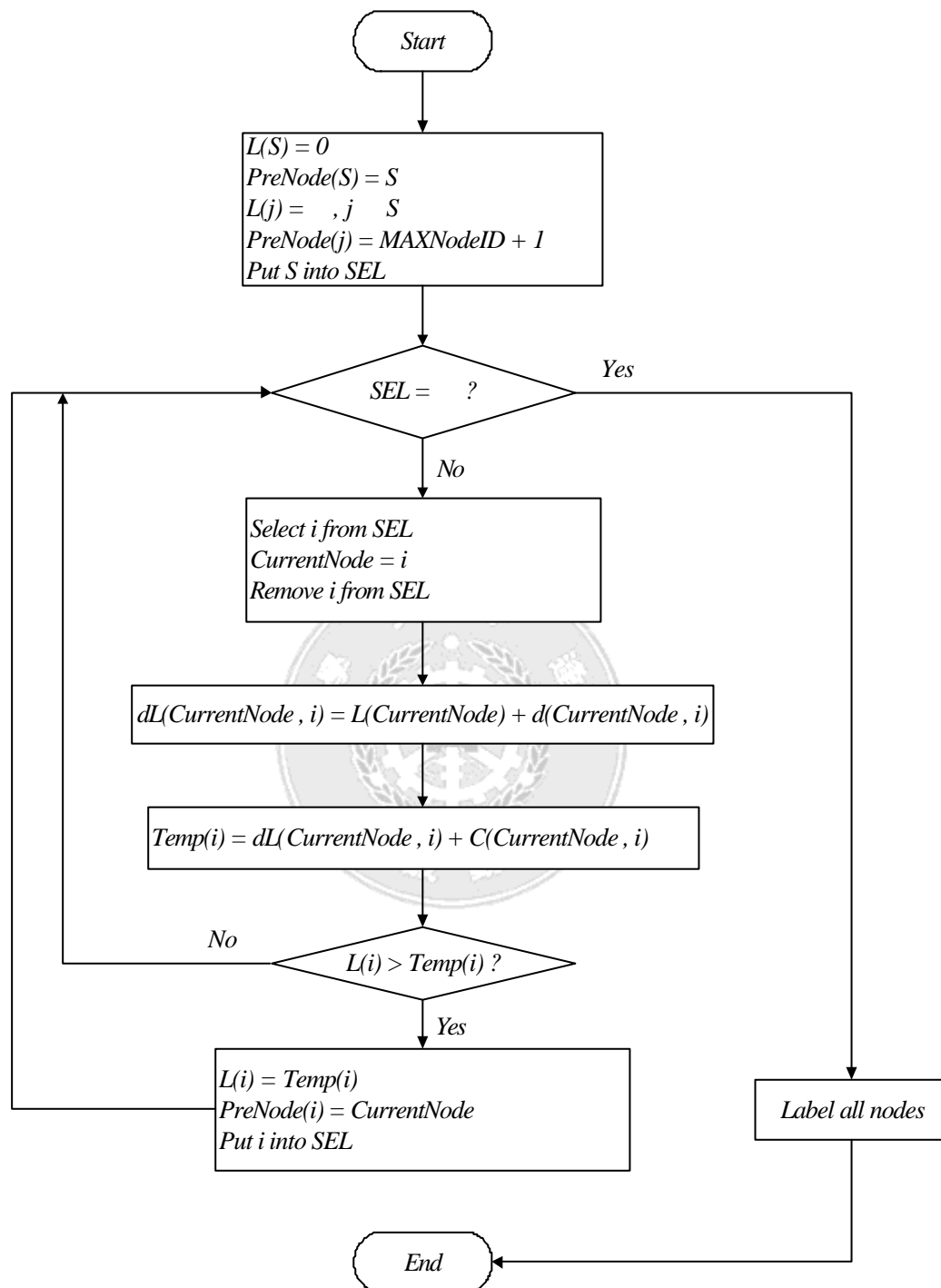


圖 3.2 路口延滯下最短路徑演算法程式流程圖

上述演算法中，與原本 LCA 不同的在於 Step 3，以往演算法從 *SEL* 取出點 *i* 後便計算其下游點之路徑成本以及總旅行時間，在本研究因為考慮了路口延滯的問題，因此在更新下游點標號值前，必須先考慮通過該點的延滯成本，因而加入此路口的延滯成本。

此外，因為這個演算法所求之最短路徑並未考慮變動的路段成本，所以當演算法開始進行計算時，便停止任何資料進入，也就是說整個演算法的路段成本皆以開始計算的時間點，作為取得路段成本的時間參考值。

### 3.2.3 演算法的合理性

本研究乃根據 LCA 來進行修正的路口延滯下最短路徑演算法。因此，在基本的演算流程中，能夠無誤的進行計算並求解得到一條從起點至終點的最短路徑。然而，本研究在每一次從 *SEL* 中選定計算中的節點 *CurrentNode* 後，會加入一個由該 *CurrentNode* 到其下游節點的路口延滯成本，所以每次選定 *CurrentNode* 後，皆必須進行  $dL(CurrentNode, i) = L(CurrentNode) + d(CurrentNode, i)$  的計算， $d(CurrentNode, i)$  為 *CurrentNode* 到下游節點 *i* 的路口延滯。此後，原本在進行下游節點累積旅行時間成本的起始標號值並非原本從起點開始累積的旅行時間成本  $L(CurrentNode)$ ，取而代之的則是以加入路口延滯後的  $dL(CurrentNode, i)$ 。因此，計算下游節點的累積旅行時間標號值公式將變成  $Temp(i) = dL(CurrentNode, i) + C(CurrentNode, i)$ ，若將  $dL(CurrentNode, i)$  的部份展開，原來的式子變成如下： $Temp(i) = \underline{L(CurrentNode) + d(CurrentNode, i)} + C(CurrentNode, i)$ ，若是將路口延滯成本與路段成本加在一起，則演算法成本計算部分將與原始的 LCA 相同。而且接下來的運算中，同樣會透過  $L(i)$  與  $Temp(i)$  之間的大小關係來進行更新節點標號值，經過每一次重複的運算，便可以將每一個節點的累積旅行時間成本標號值更新至最小。

### 3.3 K 條最短路徑演算法

最短路徑問題為一般路網問題求解的基礎，但是在實際交通的應用上最短路徑卻會有可能其他的突發狀況如交通事故或路線中斷等，導致原本所得的最短路徑無法使用。因此，必須仰賴 K 條最短路徑，來尋求所有可能的次佳路徑，以提供其他可供使用的替代路徑。

而過去數十年來，在 KSP 的研究上除了討論求解的演算法和應用外，還有一項重要的課題，那就是所產生的路徑是否含有迴圈 (loop)。所謂的迴圈是指一條路徑上允許有重覆的節點產生，尤其在進行 K 條最短路徑的計算時，為了能夠尋求到成本略低的替代路徑，有可能會透過產生迴圈的方式來獲得次佳路徑。因此，為了產生無迴圈 (loopless) 的路徑結果，就必須對所設定的路網以及演算法進行較為嚴格的限制。

此外，由於 K 條最短路徑的計算對系統效率上有很大的負荷，因此，除了先前的研究外，利用多部電腦進行平行化處理計算，以降低系統負擔提昇計算效率也是目前研究的趨勢。

#### 3.3.1 問題定義

K 條最短路徑為一般最短路徑的延伸，一般最短路徑主要在於求得一路網  $G(V, E)$  中，兩點之間的最短路徑，但在 K 條最短路徑中，不僅要求得最短的路徑，也要將其他次佳的路徑一併獲得。KSP 問題的基本定義為「求解路網中一個起點跟迄點間的所有路徑，並依成本大小進行排序，從 1 排到 k,  $k > 1$ , k 值依需求可改變大小 (Shier, 1979) 。

因此，本研究除了在計算最短路徑時考慮了路口延滯的問題，也將在 K 條最短路徑的計算中，加入考慮路口延滯問題，期能獲得並提供更完整的路徑資訊。

#### 3.3.2 求解演算法

比較本研究所回顧的 Yen's Algorithm 以及 DoubleSweep Algorithm

後，由於 DoubleSweep Algorithm 的求解方式在於使用矩陣運算，在運算過程中無法對路口進行延滯成本的考慮，而 Yen's Algorithm 必須搭配一個最短路徑演算法來計算求得最短路徑，路口延滯成本便可以在進行最短路徑計算時加入考慮。因此，本研究乃選擇 Yen's Algorithm 作為計算路口延滯下 K 條最短路徑的演算法。

Yen's Algorithm 主要是透過第  $k-1$  條最短路徑來尋找第  $k$  條最短路徑，做法是利用第  $k-1$  條最短路徑上面的第  $i$  點，找第 1 條至第  $k-1$  條最短路徑上的第  $i$  點相連的節線，將所有相連節線的成本設為無限大，再找出第  $i$  點到終點的最短路徑，最後再將所找到的路徑與起點到第  $i$  點的最短路徑結合，求得一完整的最短路徑，如此將所有第  $k-1$  條最短路徑上面的所有節點重複計算，求得一組路徑，並從這組路徑中找一最短的路徑，則求得第  $k$  條最短路徑。由此可以知道，根據這個演算法在求每條最短路徑時，必須搭配使用最短路徑的演算法才能進行運算。

由 Yen's Algorithm 可知，其主要做法是依據求得的最短路徑，對其路徑上每一個路段進行成本變動，經過不斷反覆的計算來獲得 K 條最短路徑。因此，對於路口節點成本的變化並未加以描述，所以若要在求解 K 條最短路徑的演算法中加入路口延滯成本，就必須在其利用最短路徑演算法進行路徑計算時來進行加入的動作。

而在選擇所搭配的最短路徑演算法時，因為必須考慮加入路口延滯方便性以及每一次循環只須求出所須起點到迄點的最短路徑，因此，本研究從 Label setting Algorithm(LSA) 以及 Label correcting Algorithm(LCA) 兩者中來選擇一較佳的演算法來搭配使用。經比較兩者演算法以及視 Yen's Algorithm 所需情況後發現，LCA 計算最短路徑的過程中，其演算法必須將整個路網計算結束才能停止其演算，而 LSA 卻可以經過簡單的設定，使其便成可以求解一個一對一 (one-to-one) 的最短路徑演算法。因此，為了能夠迅速的獲得 Yen's Algorithm 所需要的最短路徑，本研究乃選用 LSA 作為搭配 Yen's Algorithm 的最短路徑演算法。

在考慮路口延滯的部分，因為 LSA 除了選擇節點的方式與 LCA 不同外，其餘的運算比較過程則差異不大，因此可以延續前節對 LCA 所修正的計算方式來進行。其演算法程式流程圖如圖 3.3 所示。

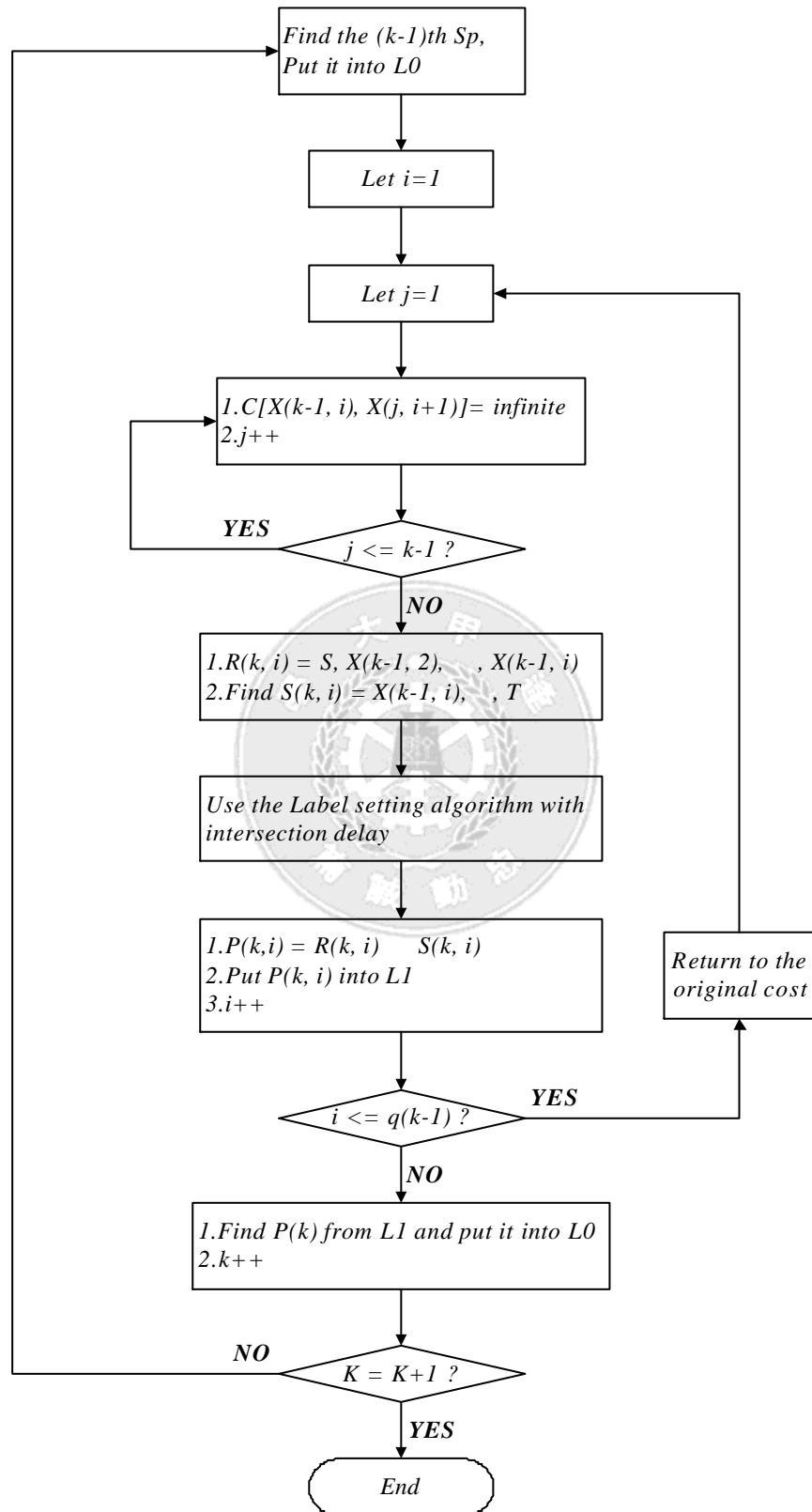


圖 3.3 路口延滯下 K 條最短路徑演算法程式流程圖

### 3.3.3 演算法的合理性

由於 Yen's Algorithm 主要是依據每一次所求得的最短路徑來對路段成本進行改變，對於在路口節點中加入其他延滯成本上，尚無特定的方式。因此，本研究在計算路口延滯下的 K 條最短路徑時，乃利用每一次循環，在系統呼叫最短路徑求取子路徑時才加入路口延滯成本。而為了能夠滿足條件所需，也選定使用 LSA 來作為搭配 Yen's Algorithm 的最短路徑演算法。

從前面的比較可以知道 LSA 的演算方法與 LCA 的演算方法只有從暫存集中選取節點時的做法不同，所以若是使用 LSA 來作為最短路徑的演算方式，便可以仿照 LCA 加入路口延滯成本的方法，也較能確保其演算的正確性。

而在路段旅行時間成本中，本研究並不考慮其會依照這時間的變化而不同，因此當系統開始進行 K 條最短路徑計算時，變固定其每個路段的旅行時間成本。

## 3.4 依時性最短路徑演算法

隨著交通情況的日趨複雜，以及即時資訊的影響，如果在計算最短路徑仍只考慮到空間上的變化，而忽略了時間上的因素，將會所得的路徑資訊在實際應用上不夠可靠。因此，近年來的最短路徑問題開始加入了時間的考量。

相較於傳統最短路徑或是 K 條最短路徑空間上的變化，因為時間的複雜特性，也將連帶使依時性最短路徑的演算增加了運算上的難度。而與傳統的最短路徑一樣，依時性最短路徑仍然可以進一步討論空間上的因素，發展 K 條依時性最短路徑問題 (Time-dependent K Shortest Path Problem)，但由於其同時牽涉到時間與空間的因素，所以計算的複雜難度將提昇許多。

在 ITS 的發展過程中，在即時資訊的傳遞以及對交通進行即時控制的前提下，正確且有效率的資訊處理技巧變得十分重要。因此，發展可以符合時間變化的需求，也可以針對路口控制延滯等相關變化進行最佳路徑的尋找，使此一精神更能符合即時且動態的智慧型運輸系統發展。

### 3.4.1 問題定義

在本研究中的依時性最短路徑問題其基本定義：一路網  $G(V, E)$ ， $V$  代表所有點之集合、 $E$  代表所有節線之集合；任意兩相鄰節點  $i, j$  之路段旅行時間成本為  $C(t, i, j)$ ，路口延滯成本為  $d(t, i, j)$ ，給定一起點  $S$ ，並考慮所有可能之因素，找到由該起點至路網中所有節點的路徑，使每條路徑的總旅行時間成本皆為最小。

由定義可知，本研究不僅在路段上考慮了依時間變動的路段旅行時間，也在節點的計算中加入了延滯成本，而這個加入的延滯成本也是一樣會隨著計算時間的不同而跟著變動。利用系統提供的路網資訊、不同時段下的路段旅行時間成本、路口延滯成本來計算獲得一對多 (one-to-all) 的路口延滯下依時性最短路徑。

### 3.4.2 求解演算法

根據過去相關文獻，計算依時性最短路徑時所使用的演算法不外乎都是利用 LSA 與 LCA 來進行演算法的修正。而本研究不僅使用到動態的路段成本，尚加入了路口號誌所產生的延滯成本，以使其較為符合現實世界的交通狀況。

與前節的求解最短路徑相同，本研究將同樣使用 LCA 的最短路徑演算法作為基礎，再針對運算期間使用的依時性路徑成本以及依時性路口延滯成本的變化，對演算法在更新標號值時進行修改。而在研究中，也針對二種路口延滯成本進行計算比較分析，其修正後 LCA 程式流程圖如圖 3.4，詳細求解步驟如下所述：

參數定義：

路網  $G(V, E)$ ： $V$  代表所有點之集合、 $E$  代表所有節線之集合；

$S$ ：起點；

$C(t_k, i, j)$ ：時間點  $t_k$  下任意兩相鄰節點  $i, j$  之路段旅行時間成本；

$L(t_m, CurrentNode)$ ：累積之旅行時間成本標號值；

$PreNode(i)$ ：點  $i$  之上游節點；



$CurrentNode$  : 處理中之節點 ;

$Temp(t_k, i)$  : 暫存之節點  $i$  標號值 ;

$SEL$  : 待選取節點之集合 ;

$d(t_k, CurrentNode, i)$  : 節點  $CurrentNode$  的路口延滯成本 ;

$dL(t_k, CurrentNode, i)$  :  $CurrentNode$  的累積旅行時間成本標號值加上  $CurrentNode$  到  $i$  路口延滯成本 ;

Step 0 : 設定所有點之初始值 : 起點  $S$  之標號值  $L(t_m, S) = 0$  ,  $t_m = 0$  , 起點  $S$  之上游節點  $PreNode(S) = S$  , 路網上所有點  $i$  ( $i \neq S$ ) 之標號值  $L(t_m, i) = \infty$  , 點  $i$  之上游節點  $PreNode(i)$  為最大節點編號 +1。

Step 1 : 將  $S$  置入  $SEL$  中。

Step 2 : 檢查  $SEL$  是否為空集合 ,

是 , 到 Step 8 ;

否 , 到 Step 3。

Step 3 : 依據 FIFO 的規則從  $SEL$  中選一點  $i$  , 設  $CurrentNode = i$  , 從  $SEL$  中移除  $i$ 。

Step 4 : 計算  $CurrentNode$  之路口延滯成本  $d(t_k, CurrentNode, i)$  , 此時  $t_k = L(t_m, CurrentNode)$  ,  $CurrentNode$  到  $i$  之起始標號值  $dL(t_k, CurrentNode, i) = L(t_m, CurrentNode) + d(t_k, CurrentNode, i)$ 。

Step 5 : 取得所有與  $CurrentNode$  相連之下游點  $i$  之路段成本  $C(t_k, CurrentNode, i)$  , 並計算  $Temp(t_k, i) = dL(t_k, CurrentNode, i) + C(t_k, CurrentNode, i)$ 。

Step 6 : 檢查  $L(t_m, i)$  是否大於  $Temp(t_k, i)$  ,

是 , 到 Step 7 ;

否 , 回到 Step 2。

Step 7 : 更新點  $i$  之標號值  $L(t_m, i) = Temp(t_k, i)$  , 且使  $t_m = t_k$  , 點  $i$  之上游節點為  $CurrentNode$  ,  $PreNode(i) = CurrentNode$  , 將點  $i$  置入  $SEL$  中 , 到 Step 2。

Step 8 : 將所有點標記 , 結束。

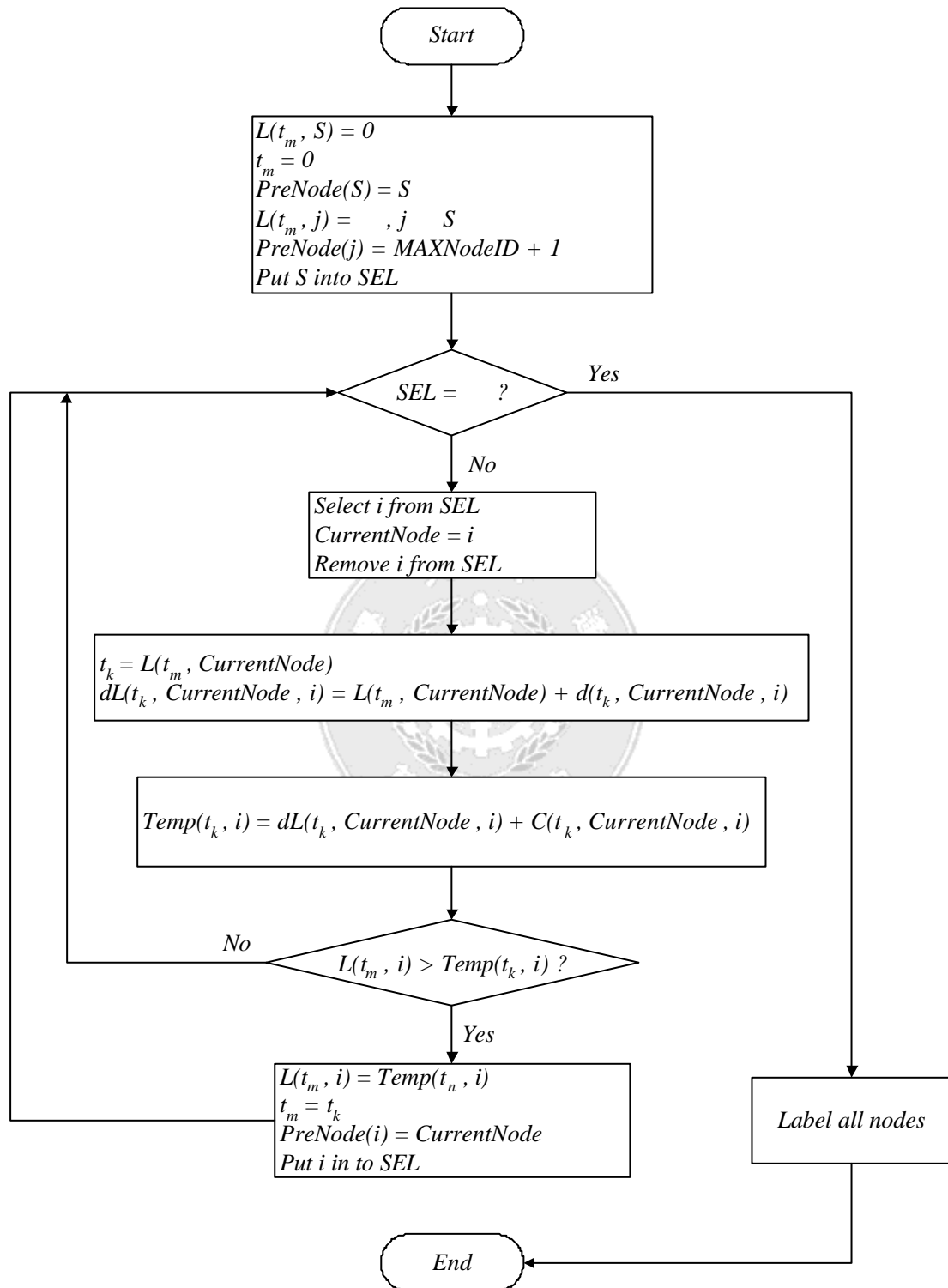


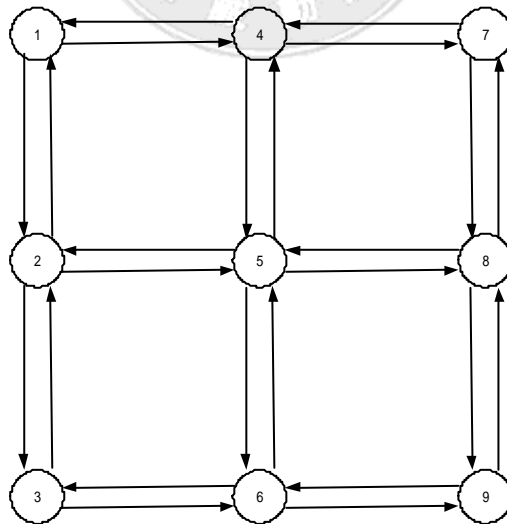
圖 3.4 路口延滯下依時性最短路徑演算法程式流程圖

### 3.4.3 演算法的合理性

上述演算法與第 3.2 節所用之演算法主要不同的部分在於  $C(t_k, i, j)$  以及  $d(t_k, CurrentNode, i)$  兩個部分，因為依時性最短路徑的所有成本會隨著時間的推進而變化，因此每次要計算其所需成本時皆須考慮相對應的時間因子，因此在此二成本計算模式中加入時間  $t_k$  用以表現出其時間相依的特性，而時間因子的決定性主要掌握於每個節點的標號值，利用標號值轉換得到時間，再透過時間取得當時的成本，如此才能符合依時性最短路徑的精神。

### 3.5 小型範例介紹

本節將舉一簡化之小型路網為例（如圖 3.5）說明上述 LCA 演算法之求解流程。而在路口延滯部分，將僅使用控制延滯做為路口延滯成本之計算模式。所使用之路網資料及號誌資料如表 3.2、表 3.3 所示。



資料來源：交通部運輸研究所，2003

圖 3.5 田字型範例路網

表 3.2 田字型範例路網之路網資料

路 段	旅行時間 (s)	路 段	旅行時間 (s)
1 2	30	5 6	35
1 4	30	5 8	35
2 1	40	6 3	45
2 3	35	6 5	45
2 5	40	6 9	50
3 2	45	7 4	50
3 6	50	7 8	30
4 1	55	8 5	30
4 5	65	8 7	35
4 7	60	8 9	35
5 2	40	9 6	40
5 4	35	9 8	40
車道數：2			
飽和流率 S：0.5 veh/sec			

表 3.3 田字型範例路網之號誌資料

節點	時相一	時相二
	綠燈時間 (sec)	綠燈時間 (sec)
1		
2	65	45
3		
4	45	65
5	45	65
6	65	45
7		
8	65	45
9		
：表無號誌		
週期：120sec		
黃燈時間：5sec		
時相一：圖形左右方向		
時相二：圖形上下方向		

計算由節點 1 到節點 9 的最短路徑，詳細流程說明如下：

#### Iteration 0

Step 1：初始設定： $L(1) = 0$ ， $PreNode(1) = 1$ ， $L(i) = \infty$ ， $PreNode(i) = 10$  ( $i = 2 \sim 9$ )，將節點 1 置入  $SEL$  中，此時  $SEL = \{1\}$ 。

Step 2：從  $SEL$  中選定節點 1 做為  $CurrentNode$ ，並將之移除。

Step 3：計算節點 1 到其所有下游節點的路口延滯成本， $d(1, 2) = 11.34$ ， $d(1, 4) = 11.34$ 。 $dL(1, 2) = 11.34$ ， $dL(1, 4) = 11.34$ 。

Step 4：計算節點 1 到其所有下游節點的路段旅行時間成本， $C(1, 2) = 30$ ， $C(1, 4) = 30$ 。 $Temp(2) = 41.34$ ， $Temp(4) = 41.34$ 。

Step 5：此時  $L(2) = \infty$ ， $L(4) = \infty$  皆比其  $Temp$  值大，所以更新節點 2、4 的標號值，使  $L(2) = 41.34$ ， $L(4) = 41.34$ ，並將節點 2、4 置入  $SEL$  中，此時  $SEL = \{2, 4\}$ 。 $PreNode(2) = 1$ ， $PreNode(4) = 1$ 。

#### Iteration 1

Step 6：檢查  $SEL$ ，並依 FIFO 規則選取節點 2 做為  $CurrentNode$ ，並將之移除。

Step 7：計算節點 2 到其所有下游節點的路口延滯成本， $d(2, 1) = 0$ ， $d(2, 3) = 0$ ， $d(2, 5) = 11.34$ 。 $dL(2, 1) = 41.34$ ， $dL(2, 3) = 41.34$ ， $dL(2, 5) = 52.68$ 。

Step 8：計算節點 2 到其所有下游節點的路段旅行時間成本， $C(2, 1) = 40$ ， $C(2, 3) = 35$ ， $C(2, 5) = 40$ 。 $Temp(1) = 81.34$ ， $Temp(3) = 76.34$ ， $Temp(5) = 92.68$ 。

Step 9：此時  $L(1) < Temp(1)$ ， $L(3) > Temp(3)$ ， $L(5) > Temp(3)$ ，所以更新節點 3、5 的標號值，使  $L(3) = 76.34$ ， $L(5) = 92.68$ ，並將節點 3、5 置入  $SEL$  中，此時  $SEL = \{4, 3, 5\}$ 。 $PreNode(3) = 2$ ， $PreNode(5) = 2$ 。

#### Iteration 2

Step 10：檢查  $SEL$ ，並依 FIFO 規則選取節點 4 做為  $CurrentNode$ ，並將之移除。

Step 11：計算節點 4 到其所有下游節點的路口延滯成本， $d(4, 1) = 0$ ， $d(4,$

$5) = 19.83, d(4, 7) = 0$ 。  $dL(4, 1) = 41.34, dL(4, 5) = 61.17, dL(4, 7) = 41.34$ 。

Step 12：計算節點 2 到其所有下游節點的路段旅行時間成本， $C(4, 1) = 55, C(4, 5) = 65, C(4, 7) = 60$ 。  $Temp(1) = 96.34, Temp(5) = 126.17, Temp(7) = 101.34$ 。

Step 13：此時  $L(1) < Temp(1), L(5) < Temp(5), L(7) > Temp(7)$ ，所以更新節點 7 的標號值，使  $L(7) = 112.68$ ，並將節點 7 置入  $SEL$  中，此時  $SEL = \{3, 5, 7\}$ 。  $PreNode(7) = 4$ 。

### Iteration 3

Step 14：檢查  $SEL$ ，並依 FIFO 規則選取節點 3 做為  $CurrentNode$ ，並將之移除。

Step 15：計算節點 3 到其所有下游節點的路口延滯成本， $d(3, 2) = 11.34, d(3, 6) = 19.83$ 。  $dL(3, 2) = 87.68, dL(3, 6) = 96.17$ 。

Step 16：計算節點 3 到其所有下游節點的路段旅行時間成本， $C(3, 2) = 45, C(3, 6) = 50$ 。  $Temp(2) = 132.68, Temp(6) = 146.17$ 。

Step 17：此時  $L(2) < Temp(2), L(6) > Temp(6)$ ，所以更新節點 6 的標號值，使  $L(6) = 146.17$ ，並將節點 6 置入  $SEL$  中，此時  $SEL = \{5, 7, 6\}$ 。  $PreNode(6) = 3$ 。

### Iteration 4

Step 18：檢查  $SEL$ ，並依 FIFO 規則選取節點 5 做為  $CurrentNode$ ，並將之移除。

Step 19：計算節點 5 到其所有下游節點的路口延滯成本， $d(5, 2) = 19.83, d(5, 4) = 19.83, d(5, 6) = 11.34, d(5, 8) = 19.83$ 。  $dL(5, 2) = 112.51, dL(5, 4) = 112.51, dL(5, 6) = 104.02, dL(5, 8) = 112.51$ 。

Step 20：計算節點 5 到其所有下游節點的路段旅行時間成本， $C(5, 2) = 40, C(5, 4) = 35, C(5, 6) = 35, C(5, 8) = 35$ 。  $Temp(2) = 152.51, Temp(4) = 147.51, Temp(6) = 139.02, Temp(8) = 147.51$ 。

Step 21：此時  $L(2) < Temp(2), L(4) < Temp(4), L(6) > Temp(6), L(8) > Temp(8)$ ，所以更新節點 6、8 的標號值，使  $L(6) = 139.02, L(8) =$

147.51，並將節點 6、8 置入  $SEL$  中 (6 在  $SEL$  中已存在所以不再加入)，此時  $SEL = \{7, 6, 8\}$ 。  $PreNode(6) = 5$ ， $PreNode(8) = 5$ 。

#### Iteration 5

Step 22：檢查  $SEL$ ，並依 FIFO 規則選取節點 7 做為  $CurrentNode$ ，並將之移除。

Step 23：計算節點 7 到其所有下游節點的路口延滯成本， $d(7, 4) = 11.34$ ， $d(7, 8) = 11.34$ 。  $dL(7, 4) = 112.68$ ， $dL(7, 8) = 112.68$ 。

Step 24：計算節點 7 到其所有下游節點的路段旅行時間成本， $C(7, 4) = 50$ ， $C(7, 8) = 30$ 。  $Temp(4) = 162.68$ ， $Temp(8) = 142.68$ 。

Step 25：此時  $L(4) < Temp(4)$ ， $L(8) > Temp(8)$ ，所以更新節點 8 的標號值，使  $L(8) = 142.68$ ，並將節點 8 置入  $SEL$  中 (8 在  $SEL$  中已存在所以不再加入)，此時  $SEL = \{8\}$ 。  $PreNode(8) = 7$ 。

#### Iteration 6

Step 26：檢查  $SEL$ ，並依 FIFO 規則選取節點 6 做為  $CurrentNode$ ，並將之移除。

Step 27：計算節點 6 到其所有下游節點的路口延滯成本， $d(6, 3) = 0$ ， $d(6, 5) = 19.83$ ， $d(6, 9) = 0$ 。  $dL(6, 3) = 139.02$ ， $dL(6, 5) = 158.85$ ， $dL(6, 9) = 139.02$ 。

Step 28：計算節點 6 到其所有下游節點的路段旅行時間成本， $C(6, 3) = 45$ ， $C(6, 5) = 45$ ， $C(6, 9) = 50$ 。  $Temp(3) = 184.02$ ， $Temp(5) = 203.85$ ， $Temp(9) = 189.02$ 。

Step 29：此時  $L(3) < Temp(3)$ ， $L(5) < Temp(5)$ ， $L(9) > Temp(9)$ ，所以更新節點 9 的標號值，使  $L(9) = 189.02$ ，並將節點 9 置入  $SEL$  中，此時  $SEL = \{8, 9\}$ 。  $PreNode(9) = 6$ 。

#### Iteration 7

Step 30：檢查  $SEL$ ，並依 FIFO 規則選取節點 8 做為  $CurrentNode$ ，並將之移除。

Step 31：計算節點 8 到其所有下游節點的路口延滯成本， $d(8, 5) = 11.34$ ，

$d(8, 7) = 0, d(8, 9) = 0$ 。  $dL(8, 5) = 154.02, dL(8, 7) = 142.68, dL(8, 9) = 142.68$ 。

Step 32：計算節點 8 到其所有下游節點的路段旅行時間成本， $C(8, 5) = 30, C(8, 7) = 35, C(8, 9) = 35$ 。  $Temp(5) = 184.02, Temp(7) = 177.68, Temp(9) = 177.68$ 。

Step 33：此時  $L(5) < Temp(5), L(7) < Temp(7), L(9) > Temp(9)$ ，所以更新節點 9 的標號值，使  $L(9) = 177.68$ ，並將節點 9 置入  $SEL$  中 (9 在  $SEL$  中已存在所以不再加入)，此時  $SEL = \{9\}$ ， $PreNode(9) = 8$ 。

### Iteration 8

Step 34：檢查  $SEL$ ，並依 FIFO 規則選取節點 9 做為  $CurrentNode$ ，並將之移除。

Step 35：計算節點 9 到其所有下游節點的路口延滯成本， $d(9, 6) = 19.83, d(9, 8) = 11.34$ 。  $dL(9, 6) = 197.51, dL(9, 8) = 179.02$ 。

Step 36：計算節點 9 到其所有下游節點的路段旅行時間成本， $C(9, 6) = 40, C(9, 8) = 40$ 。  $Temp(6) = 237.51, Temp(8) = 219.02$ 。

Step 37：此時  $L(6) < Temp(6), L(8) < Temp(8)$ ，所以不進行任何更新動作，此時  $SEL = \{ \}$ 。

### Iteration 9

Step 38：檢查  $SEL$  為空集合，所以結束。

求解結果：由節點 1 到節點 9 的最短路徑為 1 4 7 8 9，旅行時間成本為 177.68 sec。

另外，根據窮舉法列出所有由節點 1 到節點 9 可能的最短路徑並計算其旅行時間成本得到以下結果：

1. 1 2 3 6 9，旅行時間成本為 196.17 sec。
2. 1 2 5 6 9，旅行時間成本為 189.02 sec。
3. 1 2 5 8 9，旅行時間成本為 182.51 sec。
4. 1 4 5 6 9，旅行時間成本為 222.51 sec。
5. 1 4 5 8 9，旅行時間成本為 216 sec。



6. 1 4 7 8 9, 旅行時間成本為 177.68 sec。

上列六條路徑中，所得結果與使用本研究所開發的演算法所求得的结果相同，都是 1 4 7 8 9 且旅行時間成本為 177.68 sec。因此，可知本研究所開發的演算法確實可以求得考慮路口延滯成本下的最短路徑。

### 3.6 綜合討論

#### 1. 路段旅行時間成本

以往進行依時性最短路徑計算時，其所採用的路段旅行時間成本主要有將其視為滿足統計分配的型態，並隨著出發時間或到達時間的不同來計算，以及有透過資料調查的方式，來進行旅行時間估算。而在本研究中，乃係採用系統模擬的方式，透過交通模擬軟體，依據當時的路段流量下所產生的路段速率來求得。而在本研究中所使用的旅行時間主要有兩種，一為瞬時 (instantaneous) 的旅行時間，此旅行時間係用在計算最短路徑以及 K 條最短路徑上，系統在開始計算這兩種路徑時，取得該時間點的所有路段速率，以獲得每一個路段的旅行時間，並依演算法獲得最後的路徑結果；另一種為經驗 (experienced) 或歷史性 (historical) 的旅行時間，此旅行時間主要係用在求解依時性最短路徑時，根據歷史的資料來獲得每一個不同的時間下的路段速率，以進行路段旅行時間的計算，在依演算法獲得最後的路徑結果。

#### 2. 路口延滯成本

路口延滯成本方面，由於路口延滯成本在取得上較不容易，因此一般多採用調查或是以假設值來進行，而本研究乃利用公路容量手冊中的控制延滯計算模式，再以交通模擬軟體利用流量、號誌等相關資訊，來計算所需延滯成本。利用交通模擬軟體，能夠反應出每個時間下的各種交通情況，因此所獲得的旅行時間及路口延滯成本應能夠較其他方式來的接近真實的交通情形。

#### 3. 路徑演算法

在第二章的介紹中，Chen 和 Yang (2000, 2003) 發展了一套求解號誌化路網的最短路徑演算法，其乃係利用下游節點來選擇可通行的時相，以

計算獲得「最早離開時間」，並經由重複計算而得到一最短路徑結果。但在本研究中，並非以考慮車輛何時離開路口的方式，而是只要通過路口，就必須加上一平均路口延滯成本，並對該路口的所有下游節點進行計算並更新每個節點的成本標號值。因此，本研究與 Chen 的研究最大的差異乃在路口延滯的處理方式以及下游節點的選擇方式。



## 第四章 物件導向與最短路徑演算法

### 4.1 物件導向系統分析

系統分析，可以用來瞭解使用者的需求，並針對其需求來分析要解決的問題。因此本節本研究將依循 UML 分析方式進行系統分析，其分析流程圖如圖 4.1 所示。

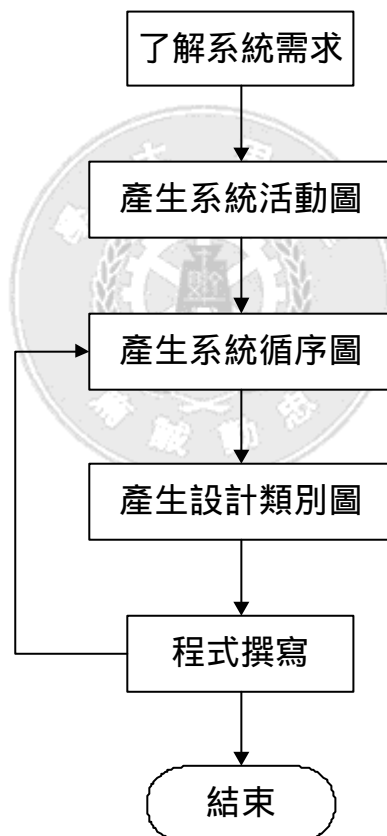


圖 4.1 本研究之 UML 系統分析流程圖

#### 4.1.1 系統功能需求

如圖 4.2 所示，根據系統計算需求，本研究將其分為四個主要部分：

匯入資料、成本計算、路徑計算以及匯出資料等。以下將就各部分進行說明。

#### 1. 匯入資料

在匯入資料的部分，為提供路徑計算所需資料，需將所要使用的台中市路網資料、各路口的號誌資料以及路口延滯相關資料匯入，以方便計算使用。

#### 2. 成本計算

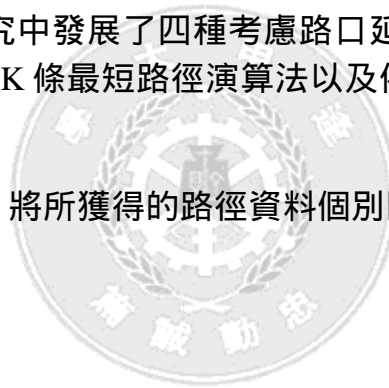
成本計算方面，由於研究主要考量路口延滯成本，因此必須利用所獲得的資料來建立兩種成本的計算方式；另外在路段旅行時間成本上，也必須處理相關路段資料後再行計算。

#### 3. 路徑計算

路徑計算方面，研究中發展了四種考慮路口延滯的路徑演算法，包括兩種最短路徑演算法、K 條最短路徑演算法以及依時性最短路徑演算法。

#### 4. 匯出資料

在結束路徑計算後，將所獲得的路徑資料個別匯出，以方便後續討論。



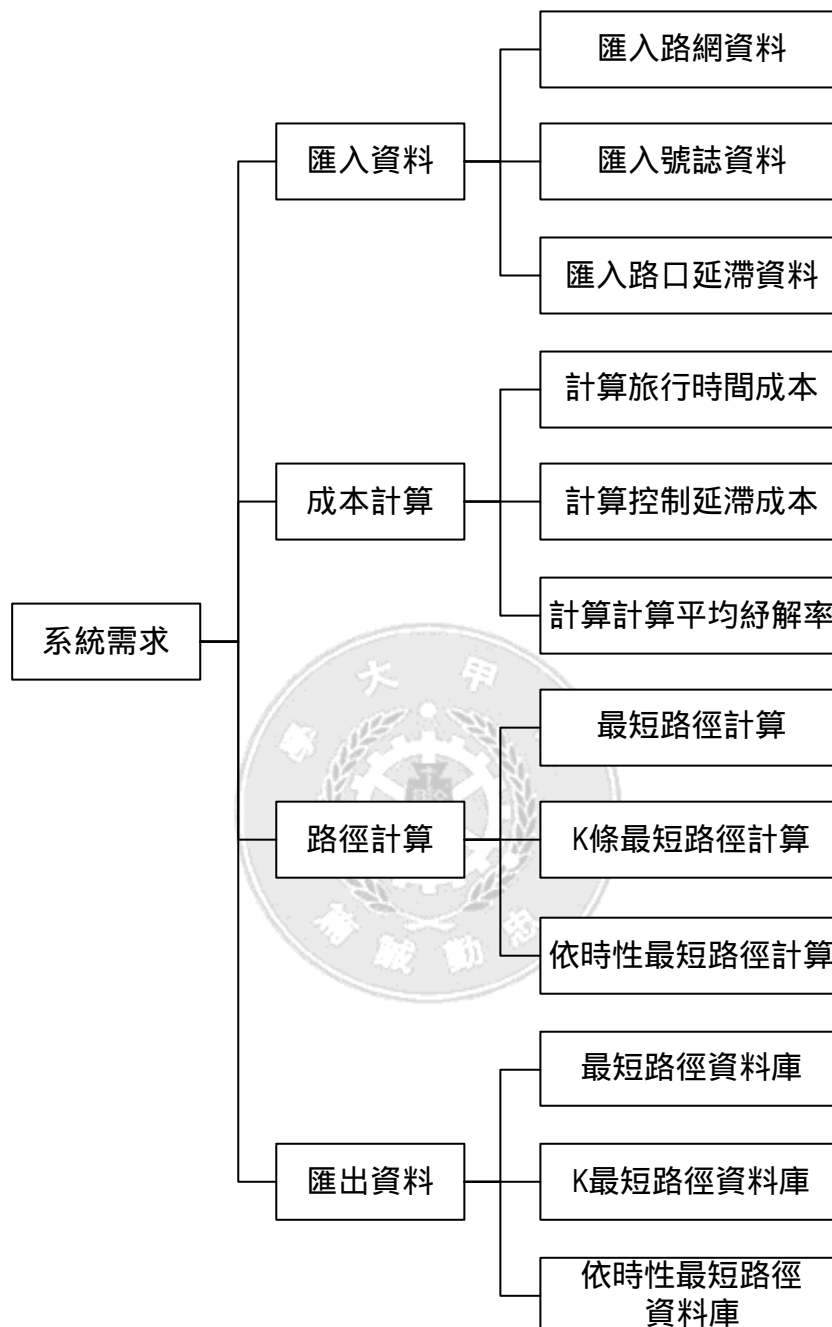


圖 4.2 系統功能需求

#### 4.1.2 系統活動圖

本研究路徑計算部分是架構在一套同樣是透過物件導向分析而實作出來的交通模擬系統 DynaTAIWAN(交通部運輸研究所, 2003) 上面, 因

此對於整個系統運作流程便不多加敘述與分析，僅分析研究中所採用的演算法相關流程與分析圖。

首先透過活動圖 (activity diagrams) 來描述系統在進行路徑計算時的運算流程。活動圖，其定義為當某個物件執行某個操作 (operation) 時，需要經歷各項工作。因而，若能將這些工作一步步地顯示出來，會有助於軟體的開發。所以，UML 提供了活動圖，用來描述物件執行某項操作的經歷過程。

圖 4.3 為路口延滯下最短路徑演算法的活動圖。圖中首先對路網中所有節點進行初值化動作，並給定一個初始時間，初始時間一旦決定後，便決定接下來所使用的各路段旅行時間成本及路口延滯成本。在取得路口延滯成本時，利用所要計算的目標可以選擇是否計算路口延滯、採用平均紓解率延滯或者是採用控制延滯作為路口延滯成本的計算。接著在依序完成所有演算動作，並進行路徑資料的輸出。

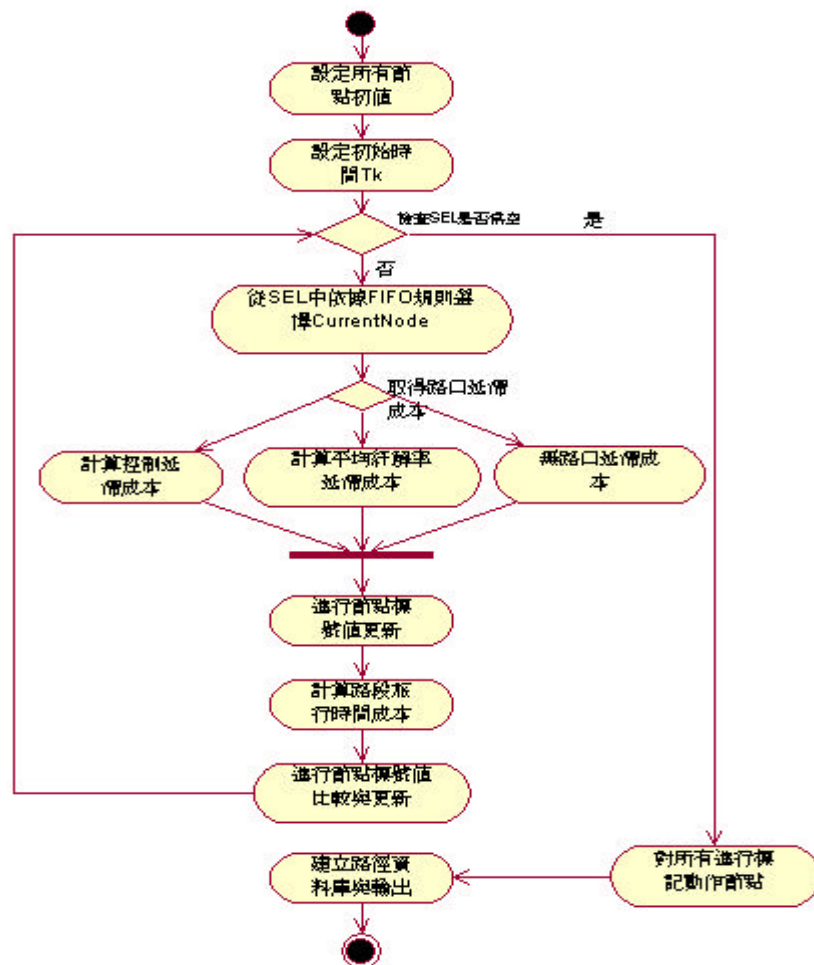


圖 4.3 路口延滯下最短路徑演算法活動圖

圖 4.4 為路口延滯下  $K$  條最短路徑演算法的活動圖，其主要仍是依循  $K$  條最短路徑演算法的步驟建構出來，同樣給定一個初始時間，並在初始時間決定後，決定後續所使用的各路段旅行時間成本及路口延滯成本。本修改後的演算法主要不同的差別在於取得每一次循環時所搭配的 LSA 的最短路徑演算法，都必須在路口加入其相對應的路口延滯成本，加入的方式跟前述的 LCA 相同。

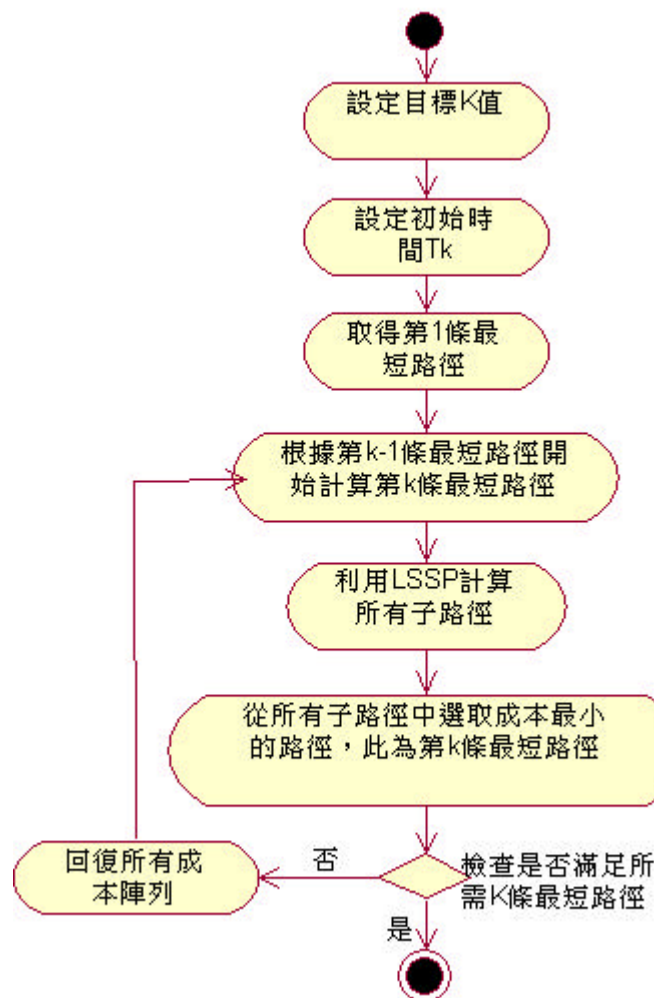


圖 4.4 路口延滯下  $K$  條最短路徑演算法活動圖

圖 4.5 為路口延滯下依時性最短路徑演算法的活動圖，其主體跟最短路徑演算法的主體相同，不同在於其每一次計算取值前，都會對時間進行更新，依照每一個選擇到的節點時間，來取得當時的路段旅行時間成本及路口延滯成本。

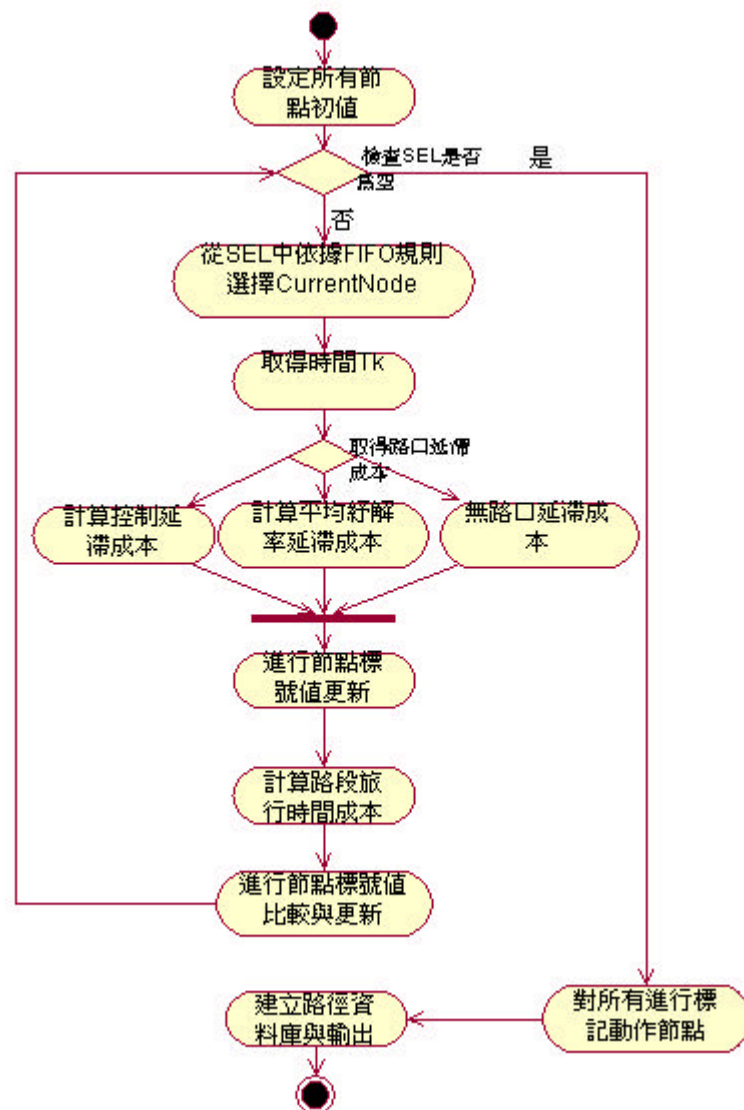


圖 4.5 路口延滯下依時性最短路徑演算法活動圖

### 4.1.3 系統循序圖

循序圖主要是被用來對使用情節的邏輯概念塑模，處理情節 (usage scenarios) 就是描述使用系統的潛在方法。循序圖以化虛擬為實境的方式為系統中的邏輯概念流程塑造模型，以便於記載和驗證邏輯，其能顯示出物件、類別間的互動關係，主要強調每一個訊息的時間順序。因此，循序圖在分析和設計上有相當大的幫助。

圖 4.6 為路口延滯下最短路徑演算法的循序圖。圖中說明如下：



1. 呼叫最短路徑方法：系統透過 GenSP() 對最短路徑演算法進行呼叫。
2. 呼叫取得路段方法：此處採用物件多型的方法，分別對不同的需要取得不同的路段資訊。
3. 呼叫取得資料：透過 GetCBNDData() 取得當時路段上的速率資料以及路口延滯資料
4. 呼叫取得路口延滯成本：此處有三種處理方式，分別為無路口延滯成本、控制延滯成本、平均紓解率延滯成本，依據系統所需傳回相對應的延滯成本。
5. 呼叫取得路段成本：透過這個方法可以獲得路段長度等資訊，並計算其旅行時間成本。

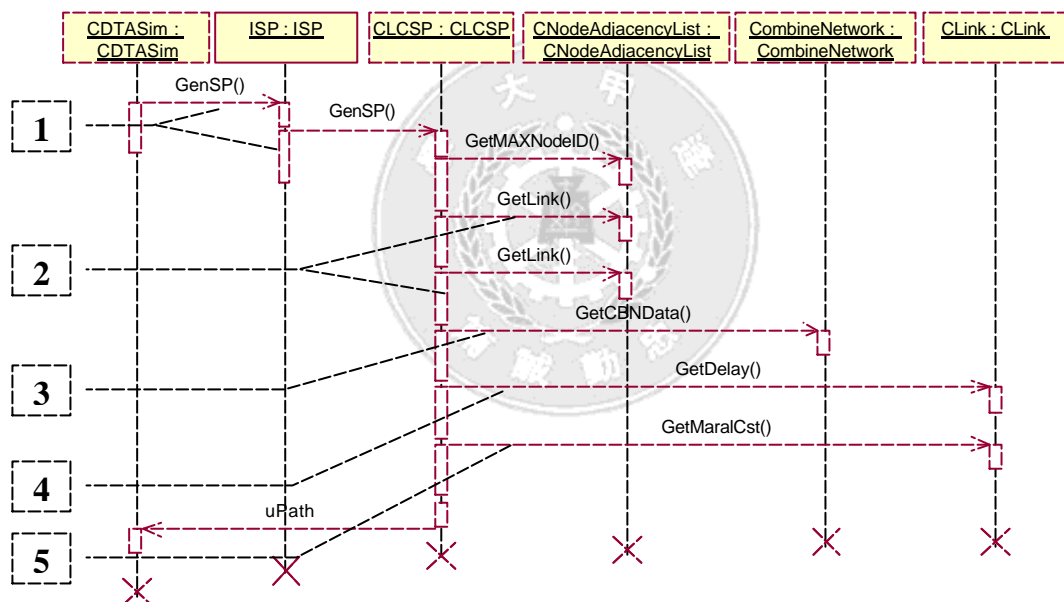


圖 4.6 路口延滯下最短路徑演算法循序圖

圖 4.7 為路口延滯下 K 條最短路徑演算法的循序圖。圖中說明如下：

1. 呼叫 K 條最短路徑方法：系統透過 GenSP() 對最短路徑演算法進行呼叫。
2. 呼叫最短路徑方法：CKSP 透過 GenSP() 對 LSA 最短路徑演算法進行呼叫並在 LSA 演算過程中加入路口延滯成本。
3. 呼叫 ListN 類別方法：在這些呼叫方法的訊息中，根據不同的需要來呼

叫 ListN 中的各種方法，以幫助 KSP 演算法的循環計算。

4. 呼叫自身方法函數：在 CKSP 類別中，也相相關的方法可供呼叫，在這邊 CKSP 利用自己的方法函數取得計算最短路徑前的根路徑。

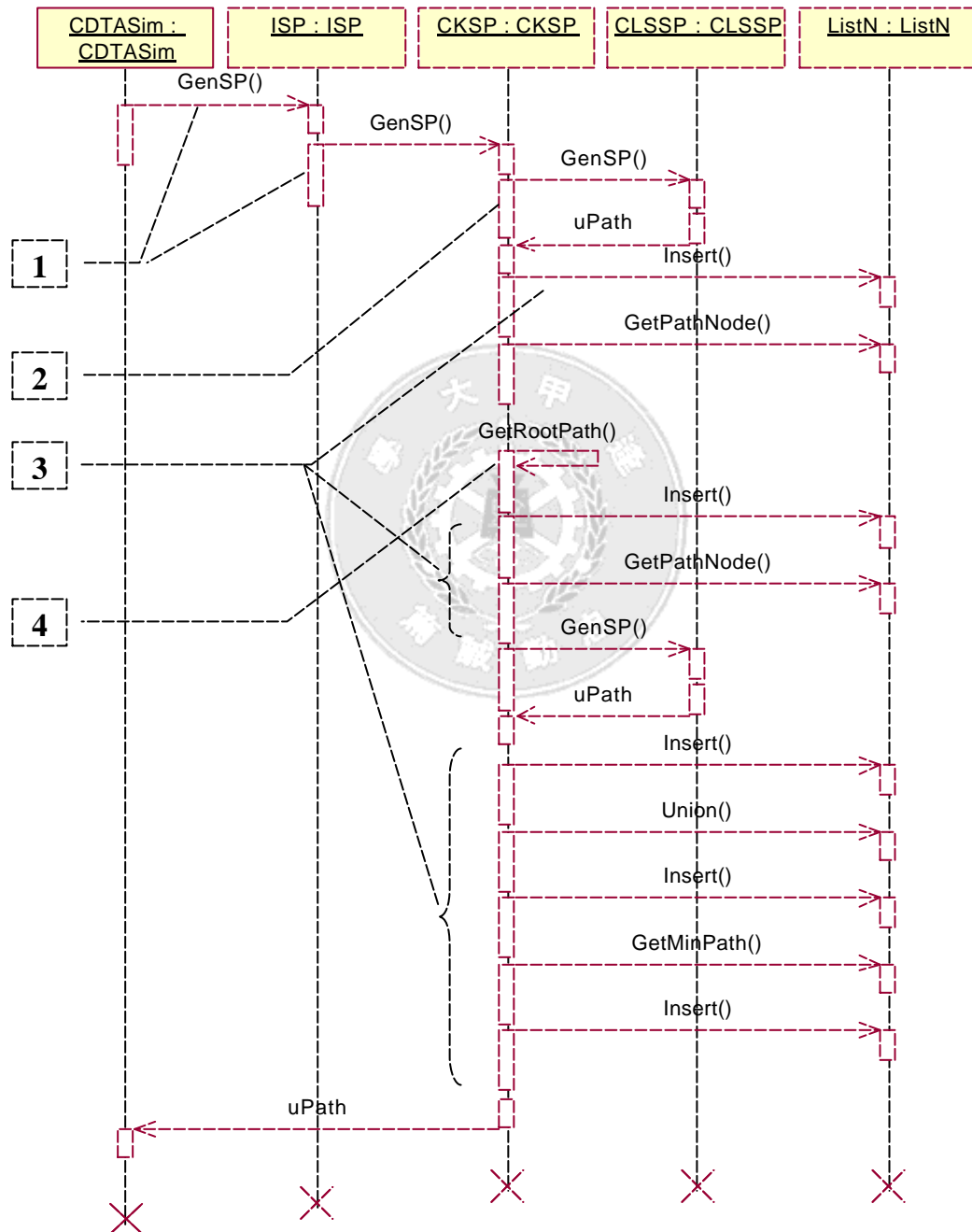


圖 4.7 路口延滯下 K 條最短路徑演算法循序圖

圖 4.8 為路口延滯下依時性最短路徑演算法的循序圖。其中各部分的呼叫方法皆與最短路徑演算法相同，不同的地方主要在於每一次進行成本呼叫時，都必須先取得當時選取節點的時間標號值，再利用該時間標號值取得符合當時的路段旅行時間成本及路口延滯成本。

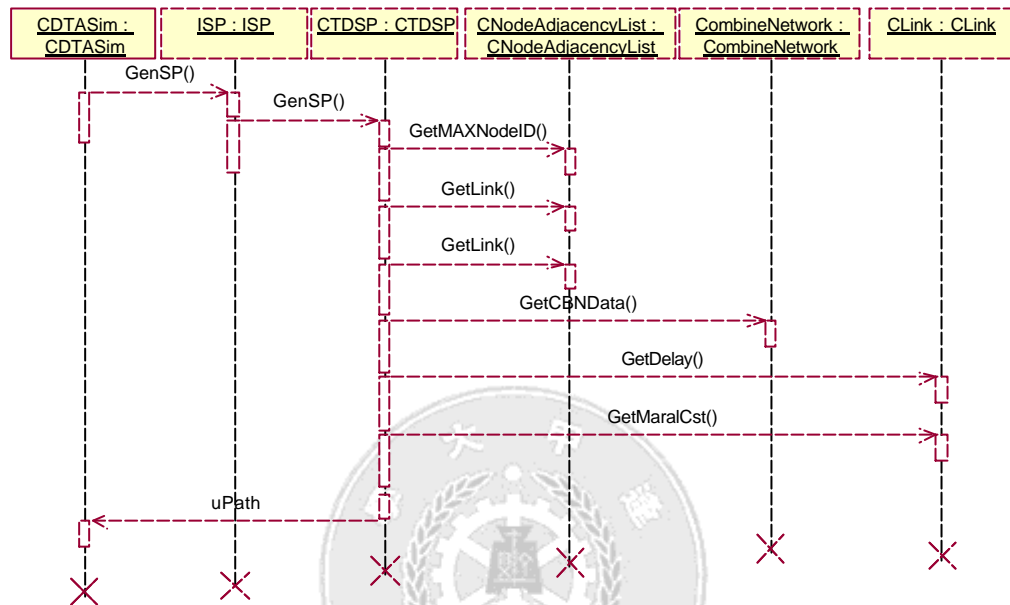


圖 4.8 路口延滯下依時性最短路徑演算法循序圖

#### 4.1.4 系統類別圖

類別圖是用來描述系統中所需的相關類別，以及類別間的靜態關係（如繼承關係）。經由定義循序圖與互動圖後，可以初步判斷出本系統中所用的類別。依據上列使用的關係，本研究中所使用的到類別包括：CDTASim、CLink、CNodeAdjacencyList、CombineNetwork、DySignal、ISP、CLCSP、CTDSP、CLSSP、CKSP 及 ListN。圖 4.9 為這些類別間的相互關係。

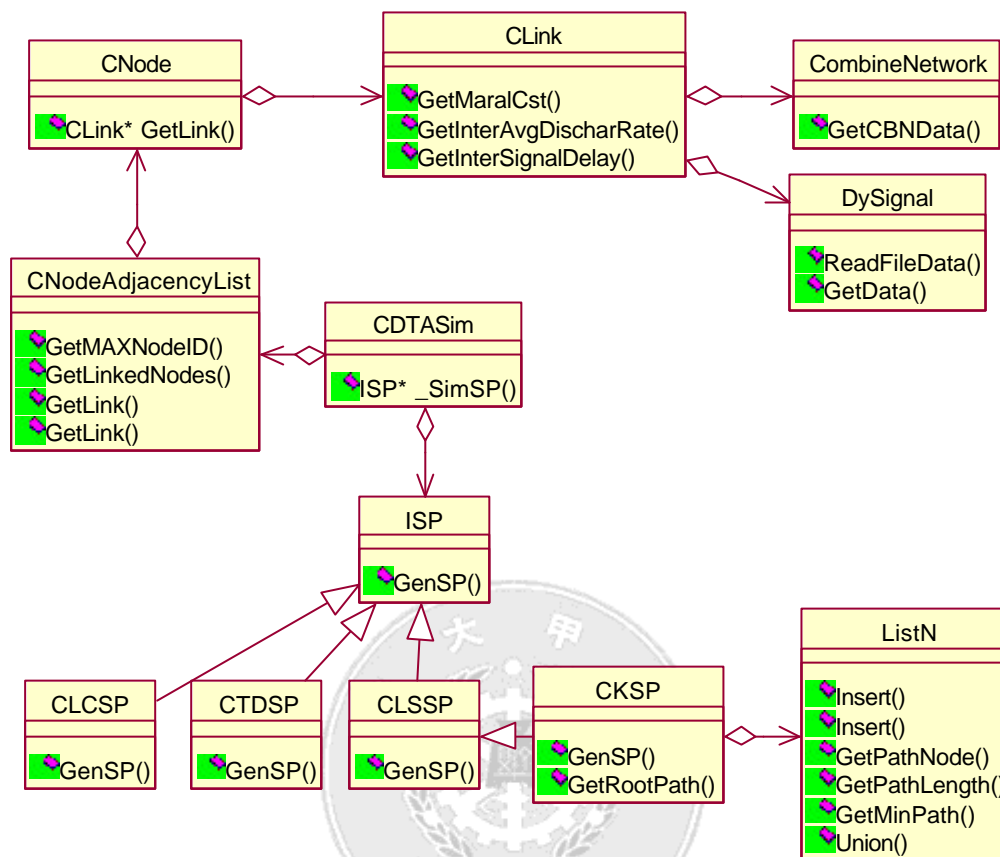


圖 4.9 本系統所使用的類別圖

## 4.2 物件導向與設計樣式

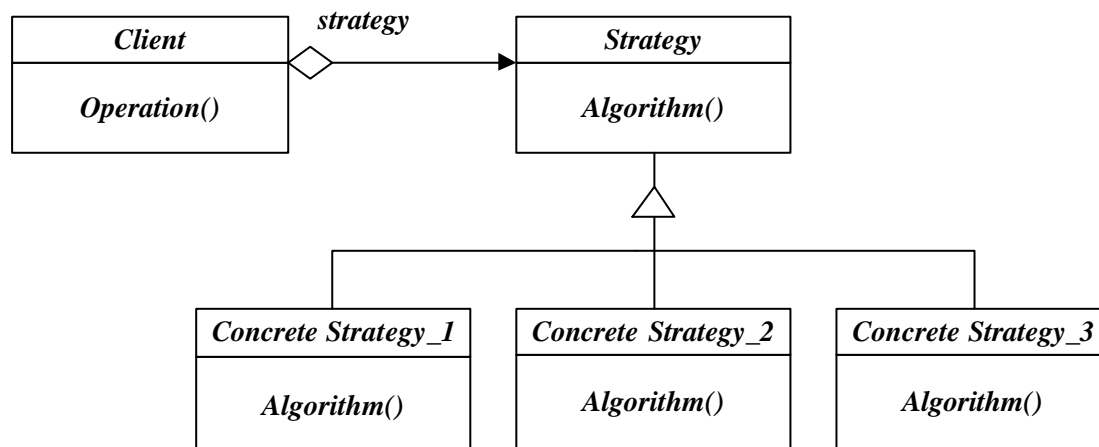
在系統實作中，本研究相繼產生考慮路口延滯下的兩種最短路徑演算法、K 條最短路徑演算法以及依時性最短路徑演算法，在其每一組程式架構中皆有相同的資料結構及使用變數，此外，為使系統能夠方便選擇所需要的演算法來求得路徑資料，本系統使用物件導向技術中的設計樣式的概念來開發相關程式。

設計樣式 (Design Pattern) 是一種將設計的知識，以特定的形式來適當地表達以及紀錄，藉由這些良好可行的紀錄與設計讓其他人再使用到類似的設計時，能夠運用在其設計問題上。設計樣式的基本精神主要是將實作的設計經驗透過抽象化整理，以一種適當的格式來描述，藉以達到設計的再使用和分享 (葉道明等人，2000)。而透過過去有經驗的程式設計師的

整理、包裝成特定的架構，可以來解決系統設計上遭遇的特定問題。因此，透過設計樣式，能夠有效的提昇系統擴展性、維護性以及程式的重用性。(薛念林等人，2003)

#### 4.2.1 策略性設計樣式基本架構

策略性設計樣式中主要有兩個成員：策略使用者或稱用戶端 (client) 及策略 (strategy) 或演算法 (Algorithm)。當用戶端不想在程式中明定所使用的策略或演算法時，可以將其抽離出來，定義一個整族演算法，將每個演算法封裝 (Encapsulate) 起來，使其不僅可以互換使用，更可讓用戶端在使用或改變其中一個演算法時，不會對其他演算法造成改變或影響。透過這個方式，策略性設計樣式可以讓系統在運用各種演算法時更具彈性，因為一旦有新的演算法加入時，用戶端可以在不更動原有演算法的架構，而將新的演算法類別繼承原有的抽象類別，只要重新定義兩者共同的介面函式，便可執行新的演算法。因此，策略性設計樣式可以充分應用物件導向觀念，來讓基本架構達到容易擴充以及重複使用的優點。(薛念林等人，2003、邱曄翰，2000)



資料來源：邱曄翰，2000

圖 4.10 策略性設計樣式基本架構圖

圖 4.10 為策略性設計樣式的基本架構圖，其中的抽象類別 Strategy 宣

告一個共同的函式介面，而 Concrete Strategy\_1、Concrete Strategy\_2、Concrete Strategy\_3 則分別為選用的三種演算法。這三種演算法繼承來自抽象類別的函式 Algorithm()，並新改寫自己的演算法。而 Client 端則透過指標 strategy 來存取 Strategy 物件下的任何一種演算法。其中，Client 端物件與 Strategy 物件間的關係是聚合關係 (Aggregation)，因此，一個 Strategy 物件可以同時被多個 Client 端物件所使用。

#### 4.2.2 策略性設計樣式應用於最短路徑演算法

本研究所提出之路徑演算法包含了兩種最短路徑、K 條最短路徑以及依時性最短路徑演算法，為了將這四個演算法加入系統中，避免造成彼此間執行時的相互影響。因此，必須讓每個演算法成為一個獨立的個體，讓使用者能依據所需，隨時選擇所要用的演算法，而不影響原有的物件。另一方面，使用者僅需知道如何使用演算法，而不必對其演算法實作內容進行了解。因此，根據以上分析，可以採用策略性設計樣式來設計最短路徑演算法的執行架構，如圖 4.11 所示。

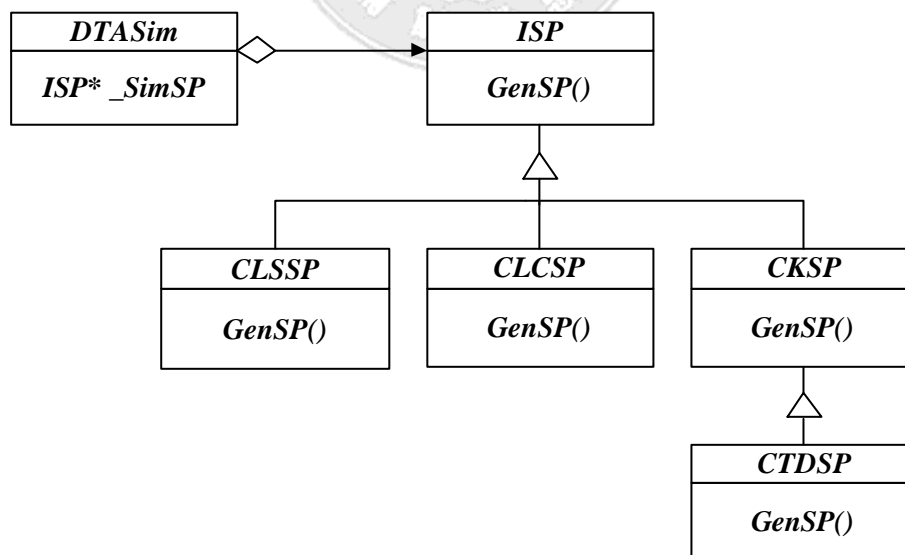


圖 4.11 最短路徑之策略性設計樣式架構圖

在圖 4.11 中，DTASim 為系統的主體，其提供一個指標 \_SimSP 來執行選取任一最短路徑演算法。而 ISP 為一個抽象類別，然後將所要發展的

最短路徑演算法繼承為抽象類別 ISP 的具體類別，並重新定義在具體類別中繼承自抽象類別的 GenSP() 等函式，藉由指標\_SimSP 執行該演算法。因此，只須在各個演算法中，將演算法類別繼承自抽象類別下並重新定義 GenSP()，即可新增完成每一個演算法。

當使用者要執行系統中的路徑演算法時，只須將該指標指向其中一個演算法即可進行路徑計算，如此不僅不用對系統的基本架構造成額外負擔，同時也達到容易擴充及重覆使用等特性，使得系統選擇路徑演算法時更具彈性。

此外，由於計算 K 條最短路徑時，必須搭配一個最短路徑演算法，因此透過策略性樣式的特性，可以讓 CKSP 能夠直接呼叫 CLSSP 來進行最短路徑計算，使其能在不改變原本最短路徑演算法程式的前提下，達到重覆使用的目的。

### 4.3 程式輸入/輸出示意圖

本系統以 Microsoft Visual C++6.0 版編譯器作為軟體工具進程式開發。圖 4.12 為程式輸入/輸出示意圖，圖中說明本研究進行路徑計算時的輸入資料與輸出結果。包含各資料及程式之間的關係。路網及號誌資料分別由 DYNASMART 及 DynaTAIWAN 匯入；此外由於 DynaTAIWAN 目前正在開發中，尚無法提供研究所需的路段速率以及路口延滯資料，因此必須透過 DYNASMART 模擬產生所需的速率及延滯相關資料，再行匯入 DynaTAIWAN 中使用；接著利用 DynaTAIWAN 所讀入資料，提供路口延滯計算程式碼以及路徑演算法程式碼進行計算，並透過 VC++ 編譯器來編譯及執行整個測試系統。由於系統並無一實際操作介面，因此，在起迄點設定時仍需直接於程式碼中加入。

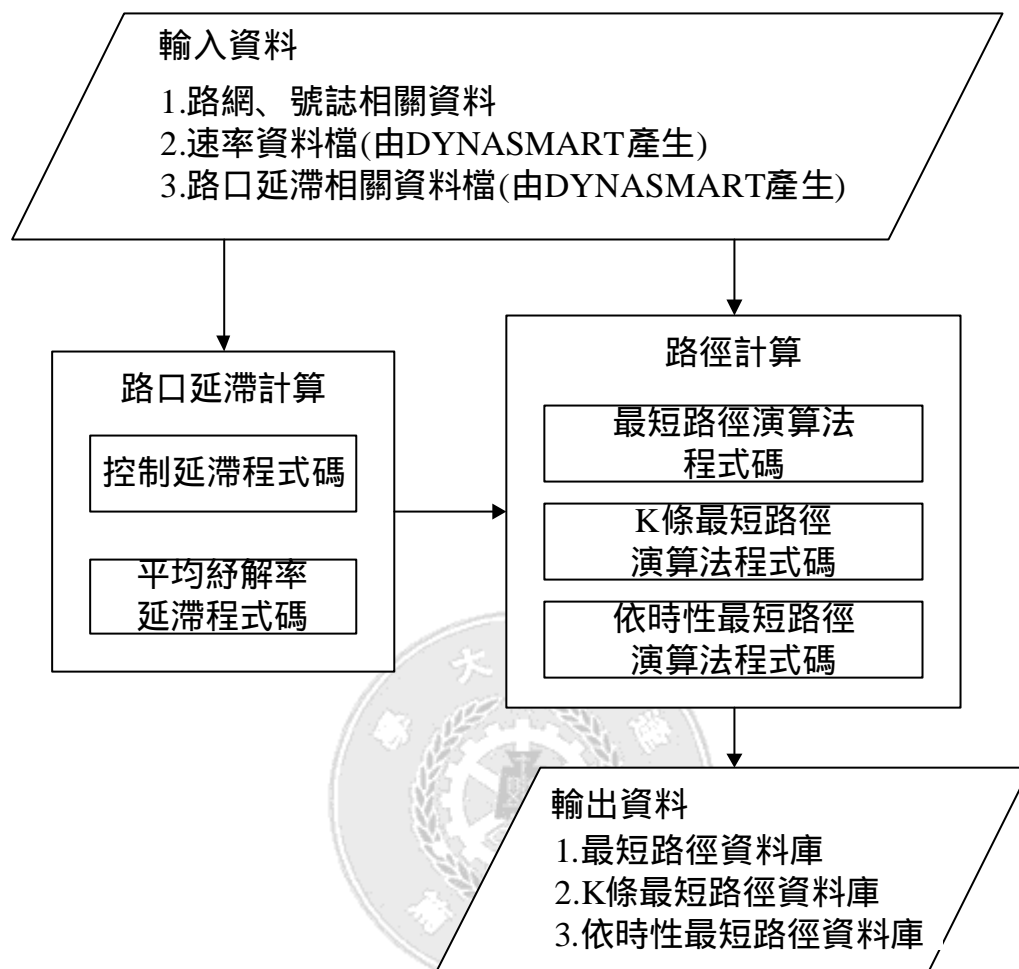


圖 4.12 程式輸入/輸出示意圖

## 4.4 程式撰寫

在進行完前述的系統分析後，接著就是要將所設計出來的系統進行實作。以下將描述在開發系統時，所使用的到路網資料結構，以及 VC++ 所提供的工具。

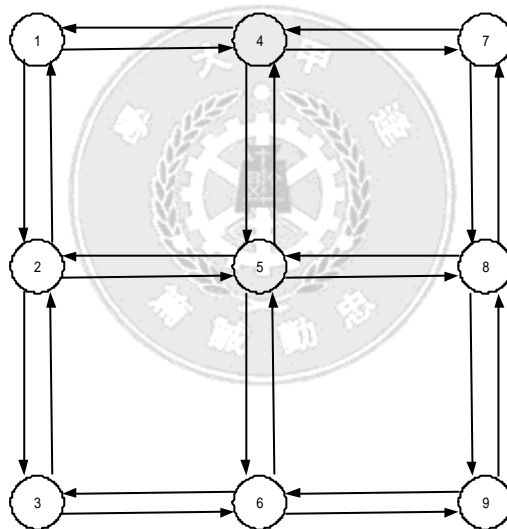
### 4.4.1 路網資料結構

由於本研究乃係架構在 DynaTAIWAN 的系統上面，因此，路網資料



結構也繼承其所使用的相鄰串列 (Node Adjacency-list)，在系統中以相鄰串列資料結構來呈現所使用的路網的連接情形。而使用相鄰串列的主要原因是因為路網中的所有節點並非全部相連在一起，若採用一般的陣列儲存方式，不僅降低存取效率也浪費了許多記憶體空間，因此在處理有存在許多不相連接的節點的路網時，採用相鄰串列對提昇系統效率以及節省記憶體空間上將有所幫助。

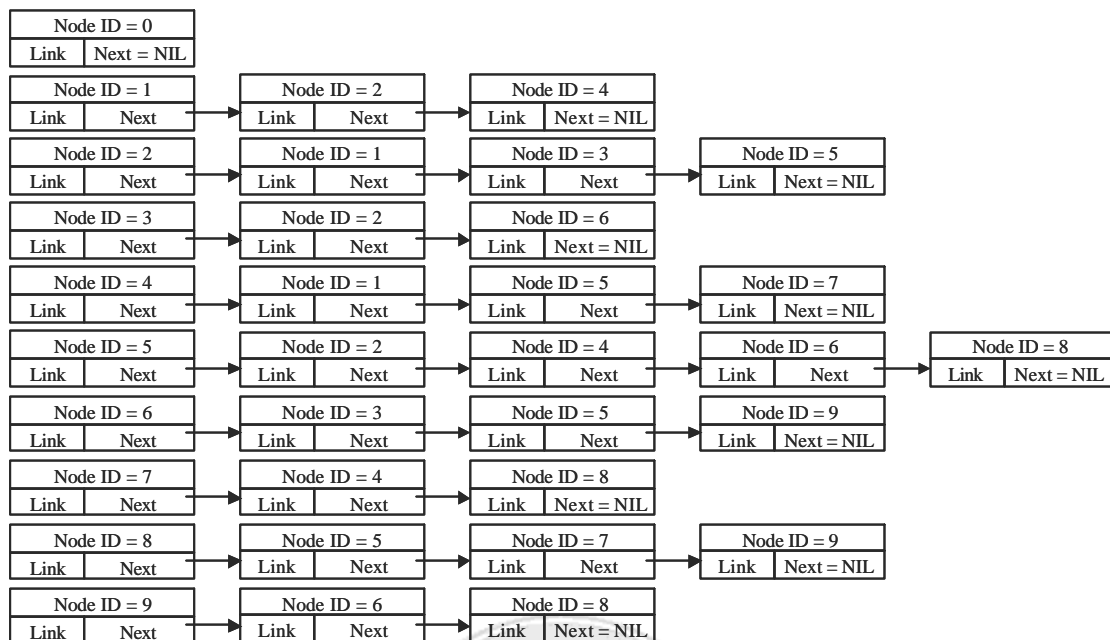
在此相鄰串列結構中的每個節點連結屬性分成三個主要部分，分為是 Node ID：其代表著每個節點的編號；Link：其描述著路網中節點間彼此的路段屬性如車道數、路長、路段流率等相關資料；最後則是 Next：代表指向下一個節點的指標。以田字型路網(圖 4.13) 為例，圖 4.14 則為此相鄰串列資料結構的呈現。



資料來源：交通部運輸研究所，2003

圖 4.13 田字型路網

圖 4.14 中最左邊為資料結構的根 (Root)，Node ID 從 1 到 9 為路網中的所有節點，每個路網節點往右邊連接的節點，是與該節點相連接的節點，以節點 3(Node ID = 3) 為例，與其相連的節點有節點 2(Node ID = 2) 和節點 6(Node ID = 6)。在系統中，所有靜態路網資訊皆以此形式儲存於此。



資料來源：交通部運輸研究所，2003

圖 4.14 田字型路網的相鄰串列

#### 4.4.2 標準樣板程式庫

標準樣板程式庫 (Standard Template Library, STL), 其所主要是將程式設計中, 常用的基本資料結構 (如陣列、鏈結串列等) 與演算法 (如搜尋、排序等), 建立起可供程式設計者方便使用的程式庫。透過 STL 的建立, 程式設計師可以直接套用與其相關的部分, 有效減輕程式設計師在開發程式上的負擔。另外在 STL 中所建立的演算法, 都經過不斷的修正與開發, 進行了最佳化的動作, 使其搭配使用的資料結構都能獲得不錯的效率。

本研究在開發路徑演算法程式的過程中, 為了方便處理暫存節點集合, 因而經常使用 STL 中的 vector。vector 和陣列差不多, 兩者在記憶體中都佔有一塊連續的空間。但在宣告 vector 時, 並不需要像陣列一樣必須先獲得陣列大小, 其可以自動配置記憶體空間給存入的資料, 所以 vector 可說是一種可以變動大小的智慧型陣列。利用 vector 可以使程式在暫存節點中方便的加入與取得節點, 利用其搭配的演算法, 如比對演算法 (equal()) 可以迅速的比對每一條路徑, 或者尋找演算法 (find()) 可以尋找所需要的節點, 相較於以往使用陣列的方式來說方便許多。

## 4.5 綜合討論

從上述的物件導向分析的流程中，可以知道透過系統功能需求分析可以了解本研究在建置系統時，其必要的資料輸入、運算以及結果輸出的項目及需求。透過系統活動圖來描述系統進行各個路徑演算法計算時的運算流程，並根據此運算流程來繪製系統循序圖，把程式物件與類別間的互動關係加以描述，以方便架構系統程式開發時每一個訊息傳送的時間順序，並藉由系統類別圖來呈現系統中每一個類別之間的關係。接著再利用物件導向程式設計中的設計樣式，來建置整個路徑運算的核心介面，以方便不同演算法的程式互相利用，使程式碼能夠有效的達到重複使用及維護管理。最後並使用其所提供的 STL 工具，提昇程式運作的效率。



## 第五章 數值實驗

### 5.1 實驗設計

本研究利用 VC++ 6.0 進行系統程式開發，並使用台中市路網資料來進行數值實驗。因此，以下將先介紹系統開發的環境與測試軟硬體，接著再對所使用的測試資料進行說明。

#### 5.1.1 測試環境

為了得到每個時間下各個路口及路段的資料，本研究以美國 PC 版本的 DYNASMART 為核心程式，來產生所需要的資料；而路徑計算程式主要是架構在採用物件導向分析與設計的 DynaTAIWAN 交通模擬軟體上，此為國內開發中的交通模擬軟體，透過同樣是採用物件導向以及使用 VC++ 6.0 進行開發的系統，可以將本研究所開發的程式方便的納入系統中來加以使用。本研究所採用的軟硬體條件為下：

##### 軟體規格

1. PC 版本之 DYNASMART；
2. Microsoft Visual C++ 6.0；
3. Microsoft Windows 2000 作業系統。

##### 硬體規格

1. CPU：Intel P4 2.0G；
2. RAM：DDR333 512Mb。

#### 5.1.2 測試資料

本研究所使用的基本資料來自台中市路網，而所使用的模擬相關資訊則是透過模擬的方式來產生，以下為所使用的各項資料詳細說明。

### 1. 路網資料

在路網資料方面，研究中所使用的台中市路網共計 87 個分區、574 個節點與 1892 條節線。目前台中市的道路系統如圖 5.1 所示。路網資料提供路徑演算法所需的節點、路段，其中根據所建構的 Node-Adjacency List 資料結構形式，儲存其他路段相關屬性如車道數、路長、路段流率等資料，以提供給路徑計算時的各種成本計算來使用。

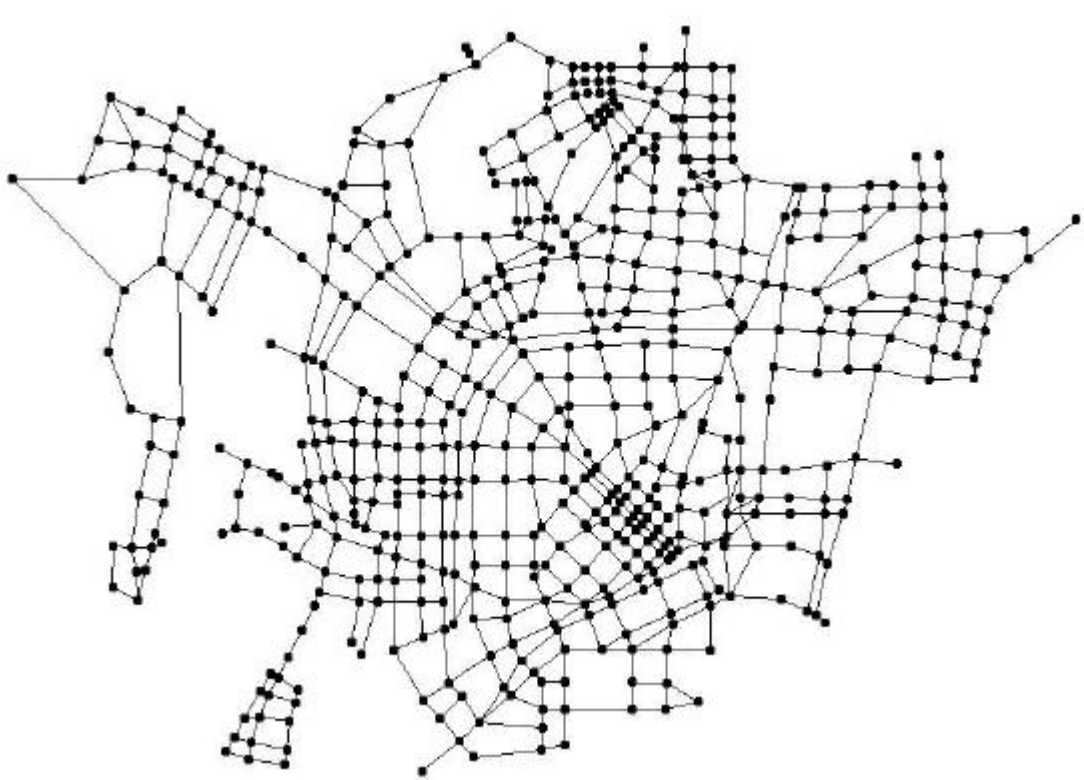


圖 5.1 台中市路網圖

### 2. 路段速率資料 (fort.33)

透過 DYNASMART 依據依時性 OD 資料，對台中市路網進行模擬，獲得 1998 個時間區間，每個時間區間為 6 秒（根據 DYNASMART 多次的運算統計得到，使用 6 秒的區間長度，可以得到效率及精度最佳的結果），共計約 3.33 小時的速率資料，此速率資料加上路網資料中路段的長度屬性，提供給系統進行路段旅行時間的計算。由此可知，本研究所使用的時間是屬於離散型 (discrete) 的而非連續型的時間資料。

### 3. 穿過路口車輛數 (fort.39)

同樣利用 DYNASMART 模擬產生每個時段穿過路口的車輛數，其產

生的時段數與速率資料相同。此資料乃是用來計算路口的平均紓解率，根

據平均紓解率公式 ( $AS_i$ ): 
$$\frac{\sum_{m=t}^{t+T} \sum_{j \in OUTBOUND} AVM_{ij}(m)}{T}$$
，以此計算該路口的平均紓解率，提供給系統計算路口延滯成本。

#### 4. 路口停等車輛數 (fort.32)

此部份的資料也是提供給系統來進行路口延滯成本的計算，其搭配平均紓解率利用公式： $\frac{VQ_i}{AS_i}$ ，來求得需要的路口延滯成本。

#### 5. 號誌資料

由於原本 DYNASMART 所使用的號誌資料中，每個號誌化路口的週期、綠燈時間長度、黃燈時間長度皆相同，但在本研究中為考量號誌所產生的延滯成本，若是每個路口號誌資料皆相同的話，將可能無法正確反應其影響程度。因此，根據陳文能 (2003) 的研究中發現，中港路、大雅路（中清路）、文心路這 3 條台中市的主要道路，其在尖峰時間中的流量與旅行時間都很大，因此，本研究針對這 3 條道路，利用假設的方式，將其號誌進行調整。

在中港路方面，將其週期延長為 180 秒、東西向的綠燈時間長度為 95 秒、南北向的綠燈時間長度為 75 秒、黃燈時間長度皆為 5 秒，共更改 12 個路口；大雅路（中清路）方面，週期延長為 160 秒、東西向的綠燈時間長度為 85 秒、南北向的綠燈時間長度為 65 秒、黃燈時間長度皆為 5 秒，共更改 15 個路口；文心路則僅調整中港路及大雅路間路段及其附近路段，週期長度為 150 秒、南北向的綠燈時間長度為 80 秒、東西向的綠燈時間長度為 60 秒、黃燈時間長度皆為 5 秒，共更改 10 個路口。除此三個的路段外，其餘路口皆為預設資料值，號誌週期為 120 秒、綠燈時間長度為 55 秒、黃燈時間長度為 5 秒。

#### 6. 所比較的起點與迄點

由於本研究所使用的台中市路網共有 574 個節點，若要將每一組 OD 都進行比較，將會耗費相當大的時間；此外，為了能夠在控制延滯中有效反應出改變週期所產生的影響，所以本研究乃從路網中挑選幾個節點，針對這幾個節點進行配對，而這幾個點分別位於該三條調整過號誌資料的路段上，共六個點，另外再行加上一個火車站週邊節點，最後以這七個點（圖 5.2，圖中『』所示）來，共產生 42 條路徑，來進行模擬比較與分析。

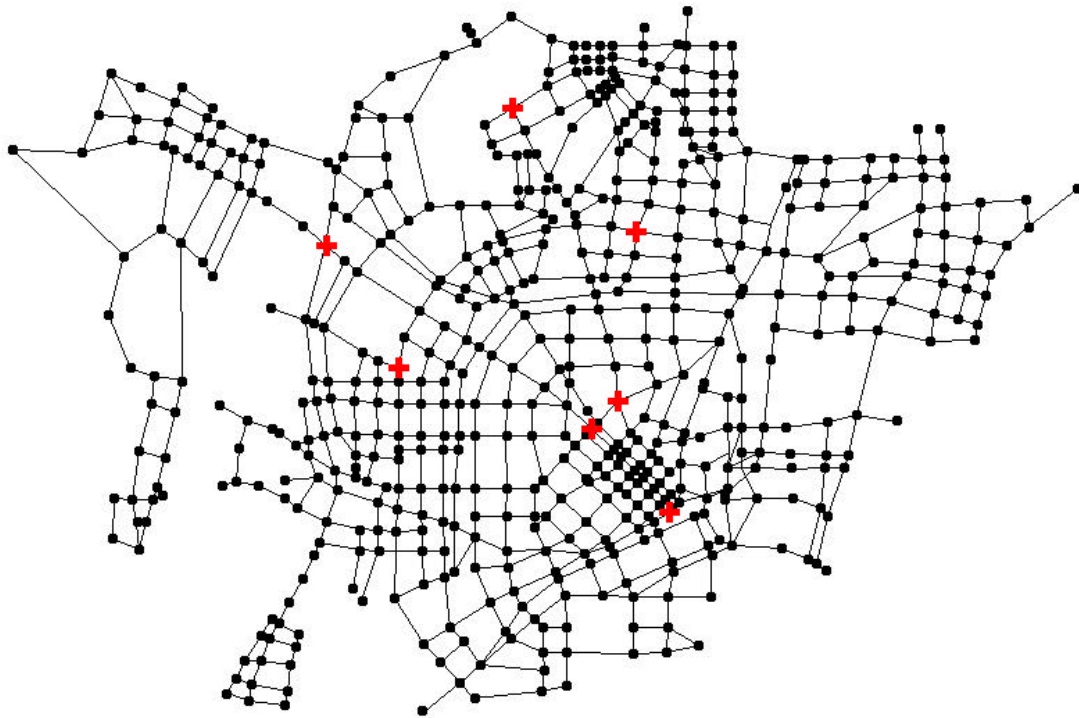


圖 5.2 本研究比較所選用的七個起點與迄點

#### 7. 所使用的資料索引值與時間的轉換方式

由於本研究所使用的資料僅以 DYNASMART 模擬 3.33 小時，共計 1998 個時間區間，每個區間為 6 秒的交通資料。因此，在計算依時性最短路徑的過程中，若其旅行時間大於 3.33 小時，在資料索引值的取得上將會選擇第 1998 個時段，否則將會依所選節點的旅行時間成本值進行轉換成時間區間，做法為將旅行時間成本值  $\div 6$  後取商數，餘數部分由於每個時間區間為 6 秒，因此可視為其存在於該區間中。時間轉換範例如下：假設節點  $i$  旅行時間為 623.4 秒，則  $623.4 \div 6 = 103$  餘 5，因此資料的索引值為 103。

## 5.2 系統操作

在開始執行路徑計算前，必須先利用 DYNASMART 來模擬產生所需要的路段速率資料以及其他路口延滯相關資料，當準備完所需資料後便可開始進行系統計算。系統操作流程圖如圖 5.3 所示。

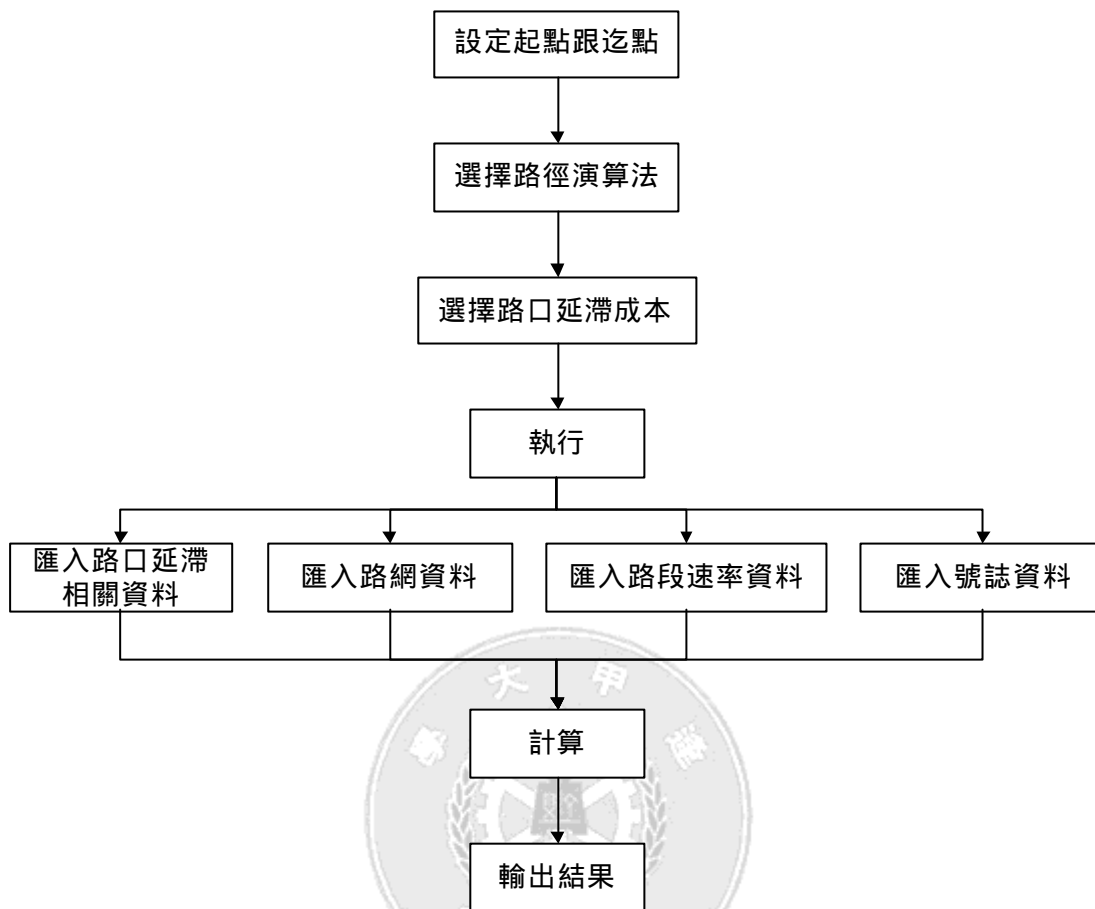


圖 5.3 系統操作流程圖

系統匯入檔案之顯示畫面如圖 5.4 所示。

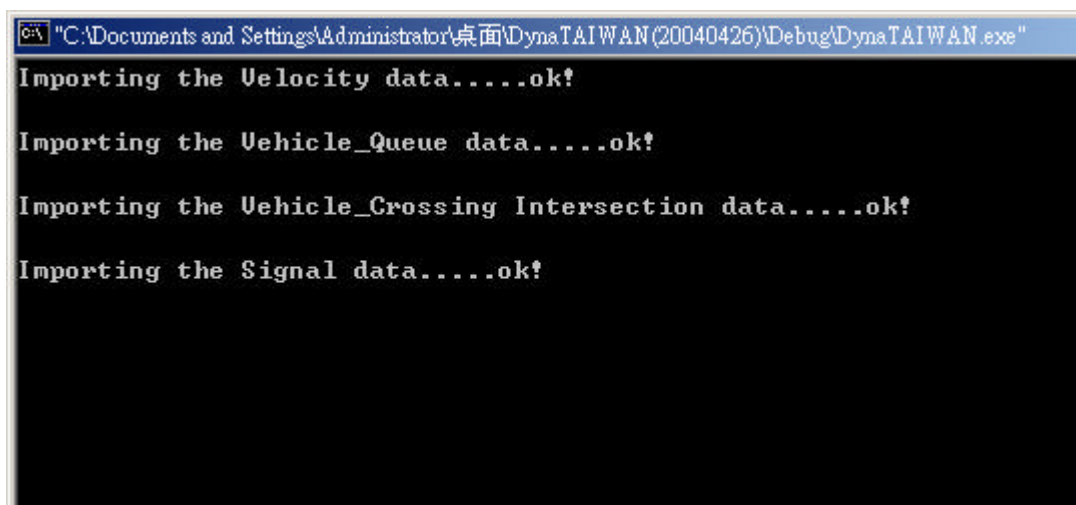
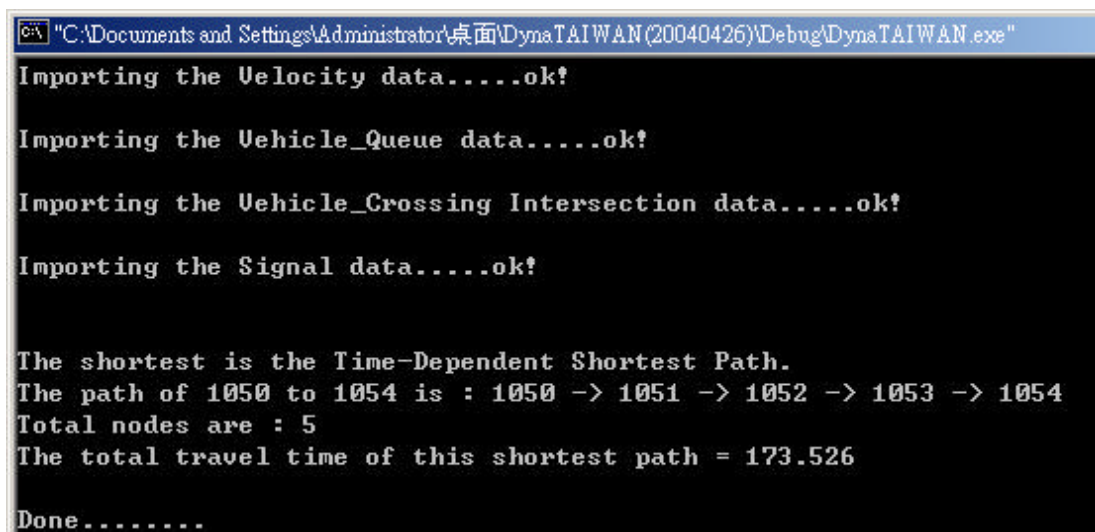


圖 5.4 系統匯入檔案畫面



系統輸出依時性最短路徑運算結果輸出畫面如圖 5.5 所示。

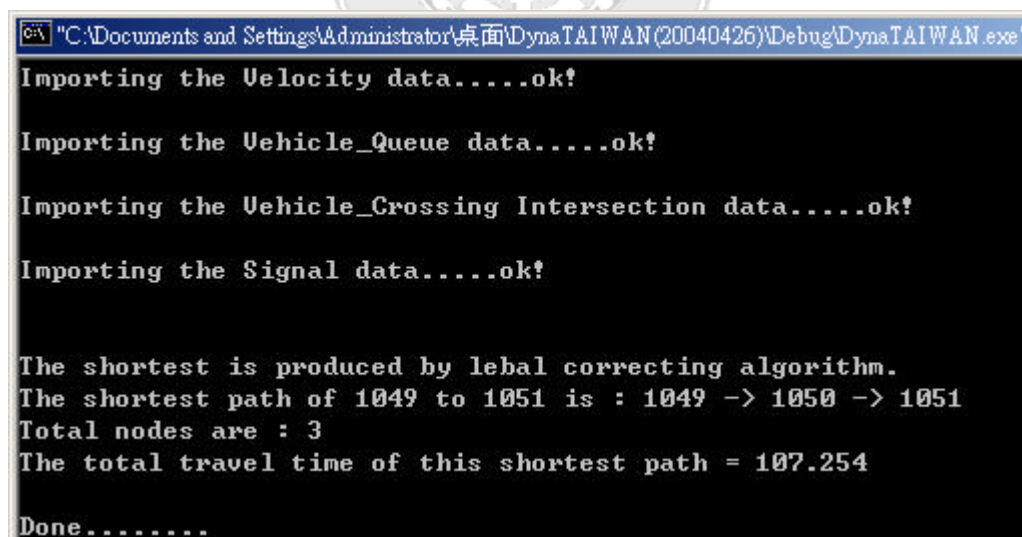


```
"C:\Documents and Settings\Administrator\桌面\DynaTAIWAN(20040426)\Debug\DynaTAIWAN.exe"
Importing the Velocity data.....ok!
Importing the Vehicle_Queue data.....ok!
Importing the Vehicle_Crossing Intersection data.....ok!
Importing the Signal data.....ok!

The shortest is the Time-Dependent Shortest Path.
The path of 1050 to 1054 is : 1050 -> 1051 -> 1052 -> 1053 -> 1054
Total nodes are : 5
The total travel time of this shortest path = 173.526
Done.....
```

圖 5.5 依時性最短路徑運算結果輸出畫面圖

系統輸出 Label correcting Algorithm 運算結果輸出畫面如圖 5.6 所示。

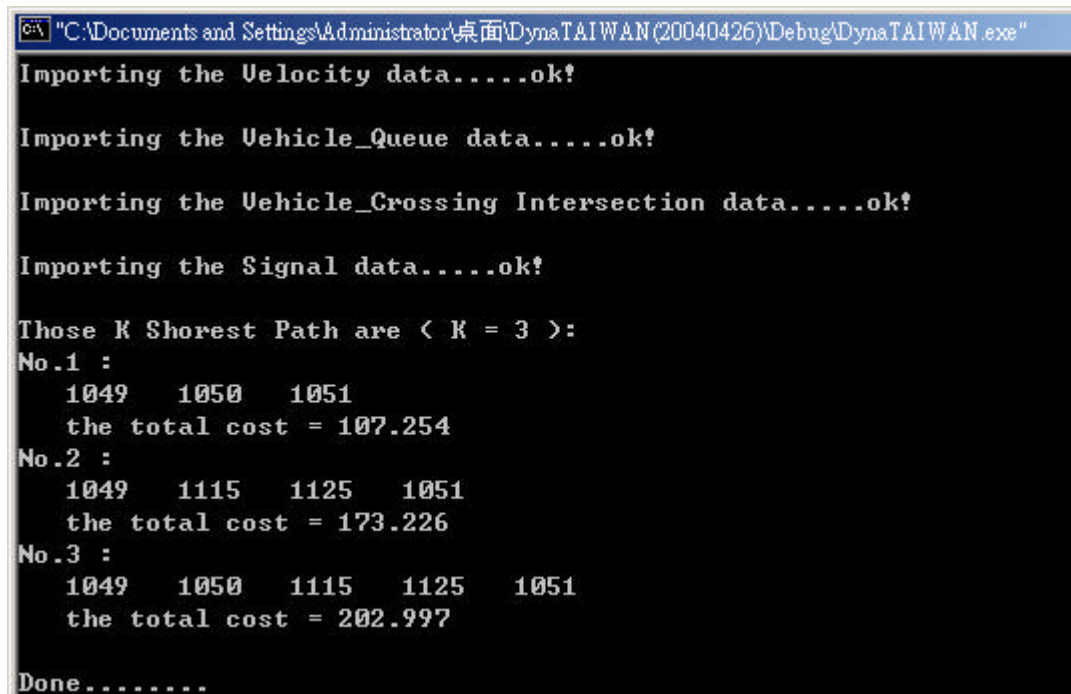


```
"C:\Documents and Settings\Administrator\桌面\DynaTAIWAN(20040426)\Debug\DynaTAIWAN.exe"
Importing the Velocity data.....ok!
Importing the Vehicle_Queue data.....ok!
Importing the Vehicle_Crossing Intersection data.....ok!
Importing the Signal data.....ok!

The shortest is produced by lebal correcting algorithm.
The shortest path of 1049 to 1051 is : 1049 -> 1050 -> 1051
Total nodes are : 3
The total travel time of this shortest path = 107.254
Done.....
```

圖 5.6 Label correcting Algorithm 運算結果輸出畫面圖

系統輸出 K 條最短路徑演算法運算結果輸出畫面如圖 5.7 所示。



```
"C:\Documents and Settings\Administrator\桌面\DynaTAIWAN(20040426)\Debug\DynaTAIWAN.exe"
Importing the Velocity data.....ok!
Importing the Vehicle_Queue data.....ok!
Importing the Vehicle_Crossing Intersection data.....ok!
Importing the Signal data.....ok!
Those K Shorest Path are < K = 3 >:
No.1 :
1049 1050 1051
the total cost = 107.254
No.2 :
1049 1115 1125 1051
the total cost = 173.226
No.3 :
1049 1050 1115 1125 1051
the total cost = 202.997
Done.....
```

圖 5.7 K 條最短路徑演算法運算結果輸出畫面圖

由於本研究並未開發圖形展示介面，但為了能展示路徑結果，因此使用軟體 MapInfo 根據計算所得的路徑結果繪製路徑示意圖，並以編號 1048 的節點（圖中『 』所示）為起點至其他六個節點之無路口延滯下最短路徑為範例進行圖形繪製，結果如圖 5.8 所示。



圖 5.8 無路口延滯下最短路徑結果示意圖

### 5.3 結果討論

本研究使用了三種路徑演算法以及兩種路口延滯成本計算模式，因此，接下來將針對所發展的演算法及模式進行數值實驗，過程如下所述：

#### 1. 無路口延滯下三種路徑演算法的結果比較

首先在自由車流下最短路徑演算法、K 條最短路徑演算法以及依時性最短路徑演算法的結果個別分析，再針對三種演算法所求解出來的路徑結果，互相進行結果比較與分析。

#### 2. 使用控制延滯下三種路徑演算法的結果比較

針對三種演算法所求解出來的路徑結果，互相進行結果比較與分析。

#### 3. 使用平均紓解率路口延滯下三種路徑演算法的結果比較

首先比較使用不同時段長度所求得的紓解率，以獲得一個較合理的時段長度，方便後續計算使用。再針對三種演算法所求解出來的路徑結果，互相進行結果比較與分析。

#### 4. 綜合比較

在綜合比較中，將分別比較不同路口延滯成本考量的最短路徑、不同路口延滯成本考量的依時性最短路徑、路段旅行時間與路口延滯時間二者權重之比較、比較 Label setting Algorithm 及 Label correcting Algorithm 的運算時間、比較最短路徑以及依時性最短路徑的運算時間、K 值改變對 K 條最短路徑運算時間的影響等。

此外，本研究依據每個不同的演算法以及路徑比較的方便與需要所產生的路徑數目如表 5.1 所示，而所產生的部分路徑結果請參閱附錄一。

表 5.1 數值實驗所產生的路徑數目

	無路口延滯 成本	控制延滯 成本	平均紓解率 延滯成本	備 註
LCA	42	42	42	
LSA	42	42	42	
KSP	4	4	4	1022 1419、 1036 1221、 1059 1022、 1221 1168
TDSP	42	42	42	

#### 5.3.1 無路口延滯下三種路徑演算法的結果比較

本節中所討論的路徑結果皆不包含路口延滯成本，單純的以路段旅行時間作為路徑計算的成本。並針對無路口延滯下的最短路徑、K 條最短路徑、依時性最短路徑的結果進行兩兩比較。而為了方便比對不同路徑演算法所產生的結果，在進行路徑比對時，本研究將先計算出每條路徑所通過的總節點數得到一個值 SumNode，再將路徑所通過的每個節點編號加總得到一個值 SumID，當 SumNode 及 SumID 兩者皆相同時，則兩條路徑視為同一條路徑，反之則為不相同的路徑。

##### 1. 自由車流下路徑結果分析

所謂的自由車流 (free flow) 係指車流量幾乎為零時，車輛彼此間不相互干擾之車輛流動。因此在本研究中所指的自由車流發生在  $t = 0$  時，此時 DYNASMART 剛開始進行模擬，因此車流產生最少，所以可視為自由車流。

而在自由車流的前提下，本研究將先比較最短路徑結果，透過結果說明自由車流下與其他車流情形的旅行時間關係，然後再針對  $K$  條最短路徑方面，將比較第  $k$  條最短路徑與第 1 條最短路徑成本差異的增加情形，最後再進行依時性最短路徑的成本分析。

而由於進行  $K$  條最短路徑所需花費的時間較多，因此在進行  $K$  條最短路徑的比較時，將從原先設定的 7 組起迄點所產生的 42 條的無路口延滯成本下最短路徑中，挑選 4 組所通過的節點數量較多的路徑來作為代表。而根據最短路徑的結果，分別選擇 1022 1419、1036 1221、1059 1022、1221 1168 等共 4 組起迄點，來進行  $K$  條最短路徑的計算與相關分析。而在其他相關路徑結果的分析與說明上，本研究同樣以這 4 組起迄點的結果來進行說明。

#### (1) 自由車流下最短路徑結果分析

本研究選定  $t = 0$  分鐘 (自由車流)、 $t = 50$  分鐘 (非自由車流) 以及  $t = 100$  分鐘 (非自由車流) 三個時段、1022 1419、1036 1221、1059 1022、1221 1168 等共 4 組起迄點進行最短路徑計算，結果如表 5.2 所示。

表 5.2 三個不同時段下最短路徑結果之比較

1022 1419		t=0	t=50	t=100
	SumNode	13	13	13
	SumID	15050	15050	15050
	旅行時間(秒)	511.451	542.442	521.789
1036 1221		t=0	t=50	t=100
	SumNode	25	25	25
	SumID	27474	27474	27474
	旅行時間(秒)	946.324	997.426	995.157
1059 1022		t=0	t=50	t=100
	SumNode	14	14	15
	SumID	14685	14823	17111
	旅行時間(秒)	618.507	711.548	675.39
1221 1168		t=0	t=50	t=100
	SumNode	19	19	19
	SumID	20913	21387	21387
	旅行時間(秒)	608.639	660.317	628.776

根據表 5.2 的結果，在 3 個不同的時間下，4 組起迄點的最短路徑在  $t=0$  的時候所獲得的最短路徑旅行時間皆為最小，因此，根據本研究可說明在自由車流下所獲得的最短路徑旅行時間，確實比在其他受車流影響的情況下所求的得旅行時間來的小。

## (2) K 條最短路徑成本分析

在 K 條最短路徑成本的分析中，本研究以自由車流下進行 K 條最短路徑的計算，並將所獲得的第 k 條最短路徑結果與第 1 條最短路徑相比較，以獲得其成本變動的趨勢。計算結果如表 5.3 所示。

表 5.3 自由車流下 K 條最短路徑成本差異之比較

1022 1419					1036 1221				
K	SumNode	SumID	旅行時間 (秒)	差異 (%)	K	SumNode	SumID	旅行時間 (秒)	差異 (%)
1	13	15050	511.451		1	25	27474	946.324	
2	13	15350	521.051	1.88%	2	25	27380	958.324	1.27%
3	13	15659	525.851	2.82%	3	25	27402	958.396	1.28%
4	14	16238	538.8	5.35%	4	25	27391	958.541	1.29%
5	14	16100	540.251	5.63%	5	27	31075	976.035	3.14%
6	13	14705	541.762	5.93%	6	25	27173	976.324	3.17%
7	14	16114	543.6	6.29%	7	27	30527	987.124	4.31%
8	13	15067	547.451	7.04%	8	27	30981	988.035	4.41%
9	14	16538	548.4	7.22%	9	27	31003	988.107	4.42%
10	14	16400	549.851	7.51%	10	27	30992	988.252	4.43%
1059 1022					1221 1168				
K	SumNode	SumID	旅行時間 (秒)	差異 (%)	K	SumNode	SumID	旅行時間 (秒)	差異 (%)
1	14	14685	618.507		1	19	20913	608.639	
2	14	14823	623.09	0.74%	2	22	24799	691.439	13.60%
3	15	17024	630.473	1.93%	3	22	25002	694.006	14.03%
4	15	15816	633.128	2.36%	4	21	23345	700.8	15.14%
5	14	14902	641.09	3.65%	5	21	23287	703.439	15.58%
6	15	15863	653.09	5.59%	6	21	23657	707.329	16.21%
7	15	15749	655.928	6.05%	7	21	24412	713.459	17.22%
8	14	14811	657.89	6.37%	8	21	23282	720.239	18.34%
9	15	17971	658.073	6.40%	9	24	27300	739.584	21.51%
10	14	14817	659.09	6.56%	10	21	24367	758.914	24.69%

根據表 5.3 的結果顯示，K 條最短路徑的計算結果中，雖然無法明確的指出當 k 值為多少時其會增加多少成本，但從結果中可以得知每增加一個 k 值的路徑旅行時間成本都很穩定的上升，其上升幅度平均約 1%~3% 左右。而檢視其每條路徑的節點後，可知在本研究中 K=1~10 之間並未有

迴圈 (loop) 的情形出現。但若是根據本研究所撰寫的演算法及程式，由於在進行路段成本變動時，本研究並未給予太多限制，所以若 K 值在繼續增加的話，則有可能會有迴圈的結果出現。

### (3) 依時性最短路徑成本分析

由於依時性最短路徑所使用的路段旅行時間隨著時間的變動而跟著變動，因此，其所計算得到的旅行時間若與自由車流下所求出來的旅行時間進行比較時，其旅行時間值應大於或等於自由車流下最短路徑的旅行時間成本，因為自由車流下所求得的最短路徑旅行時間應為系統中所有路徑結果中的最小值。

因此，本研究設定起始時間  $t=0$  分鐘，分別計算最短路徑以及依時性最短路徑，並以最短路徑為基準，比較兩者的旅行時間成本差異，結果如表 5.4 所示。

表 5.4 自由車流下最短路徑與依時性最短路徑旅行時間之比較

差值 (秒)		起點						
		1022	1036	1048	1059	1168	1221	1419
迄	1022		-35.208	0	-11.997	-5.077	-2.266	-7.008
	1036	-4.545		-8.549	-5.08	-1.361	-16.192	-2.786
	1048	0	-44.064		-9.773	-7.118	-2.888	-4.081
	1059	-4.795	-10.943	-3.597		-3.953	-10.951	-0.583
	1168	-2.152	-1.191	-5.268	-7.359		-12.113	-7.038
點	1221	-2.07	-47.625	-0.726	-13.076	-6.776		-5.41
	1419	-4.037	-12.589	-1.341	-0.482	-4.661	-8.467	

註：自由車流下最短路徑旅行時間 - 依時性最短路徑旅行時間

根據表 5.4 的結果說明確實依時性最短路徑的旅行時間皆大於或等於自由車流下最短路徑的旅行時間。

### 2. 最短路徑與 K 條最短路徑

而在最短路徑與 K 條最短路徑的比較中，將比較本研究的最短路徑計算所產生的結果是否為 K 條最短路徑中的第 1 條最短路徑。由於本研究所



使用的最短路徑演算法為 Label correcting Algorithm，而搭配 Yen's Algorithm 來計算 K 條最短路徑的最短路徑演算法為 Label setting Algorithm，所以透過這邊的比較可以檢視兩種最短路徑演算法所計算的結果是否相同。

此外，在後續的討論中，本研究將以 LCSP 表示 Label correcting Algorithm 所計算的最短路徑；KSP 表示所計算的 K 條最短路徑；LSSP 表示由 Label setting Algorithm 所計算的最短路徑。

而根據本研究所計算求得的 LCSP 與 KSP 結果如表 5.5 所示。

表 5.5 無路口延滯下 LCSP 與 KSP 比較結果

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	15050	13	521.789	27474	25	995.157
LCSP	15050	13	521.789	27474	25	995.157
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	17111	15	675.39	21387	19	628.776
LCSP	17111	15	675.39	21387	19	628.776

註：路徑計算起始時間：100 分鐘

根據表 5.5 得知，在 K=1 時的路徑結果，不論是 SumNode、SumID 或是旅行時間成本，兩種路徑演算法產生的結果皆完全一致。因此，表示本研究所開發的最短路徑計算程式與 K 條最短路徑計算程式正確無誤。另外，也代表 LCSP 的結果與 LSSP 的結果相同。

### 3. K 條最短路徑與依時性最短路徑

由於 KSP 所使用的路段成本為固定值，而依時性最短路徑（以 TDSP 表示）所使用的路段成本會隨著時間改變的改變，兩者使用的時間成本性質大不相同。因此，本研究主要比較這兩種演算法所計算的結果，檢視是

否可以在 KSP 的結果中找到相同的路徑，以說明是否可以透過求解 KSP 的方式，來獲得與 TDSP 相同的結果。比較結果如表 5.6 所示

表 5.6 無路口延滯下 TDSP 與 KSP 結果比較

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	15050	13	521.789	27474	25	995.157
TDSP	15050	13	519.854	27474	25	962.08
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1~50)	不存在	不存在	不存在	不存在	不存在	不存在
TDSP	17111	15	682.461	21387	19	623.338

註：路徑計算起始時間：100 分鐘

根據表 5.6 的結果顯示，1022 1419 以及 1036 1221 這兩組起迄點皆可以在 KSP 的結果中找到 TDSP 所產生的路徑，且碰巧都在 K=1 時找到相吻合的路徑。而在 1059 1022 以及 1221 1168 這兩組起迄點中，將 KSP 的 K 值增加到 K=50 仍然無法找到相同的路徑。檢視 1059 1022 的 TDSP 以及 KSP(K=1~5) 路徑結果，發現在 K=1~5 之間的路徑，都僅有 K=1、3、4 的路徑在其從起點出發後的一小段路徑和 TDSP 的結果一樣，但其也僅有 2 個節點相同。而在 1221 1168 的 TDSP 以及 KSP(K=1~5) 路徑結果中，在 K=1 時的路徑僅有中間約 4 個節點與 TDSP 不相同，其餘部分則相同，而在其他 K 值部分，多數路徑的後半段路徑皆與 TDSP 後半段相同。

從上面的分析中可知雖然無法在 KSP 找到與 TDSP 完全相同的路徑，但仍然可以發現某些路段還是會有重覆現象發生，至於無法找到完全與該 2 組路徑相同的結果，研判其可能原因是因為該依時性最短路徑所通過的節點中，有些路段旅行時間成本在 KSP 的計算時段中相當大，而影響其在計算路徑時的結果。因此，由此結果說明在 KSP 的路徑結果中，確實存在與 TDSP 所得路徑結果相同的機會。

#### 4. 依時性最短路徑與最短路徑

在依時性最短路徑與最短路徑的比較上，本研究分成兩個部分進行比較，首先先比較兩種路徑演算法所求解出來的路徑，接著再對所計算的路徑成本進行討論與分析。而用來比較的路徑則是以之前所設定的 7 組起迄點所產生的 42 條路徑為對象。

##### (1) 兩種演算法所得路徑的變化情形

進行兩種演算法比較時，本研究以 LCSP 的路徑結果作為基準，其比較結果如表 5.7 所示。

表 5.7 無路口延滯下 TDSP 與 LCSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	7	16.67%	增加	2	28.6%
不變	35	83.33%	減少	1	14.3%
			不變	4	57.1%

註：路徑計算起始時間：100 分鐘

根據表 5.7 的結果顯示，在 42 條路徑中，只有 7 條路徑有產生變化，所佔比例約為 16.67%，從比例上來看有些偏低。透過資料的研判，得知原因應起源於每個時段下的路段旅行速率差異不大，所以才會降低其路段旅行時間成本的影響。

##### (2) 兩種演算法所得路徑的旅行時間成本變化情形

本研究為了能夠建立一個基準來對路徑成本進行比較，另外製作了一個程式，其可以將最短路徑演算法所產生出來的路徑，依照依時性路段成本及依時性路口成本的特性進行模擬，重新計算該路徑的總旅行時間，以方便與依時性最短路徑所產生的結果進行比較。而此將最短路徑演算法所產生出來的路徑以依時性路段成本與依時性路口成本計算所得到的結果以 TLCSP 表示。

在 TDSP 與 TLCSP 的比較中，本研究以 TLCSP 為基準，進行 TLCSP 與 TDSP 的成本差異比較，結果如表 5.8 所示。

表 5.8 無路口延滯下 TLCSP 與 TDSP 成本差異百分比

差值 %		起點						
		1022	1036	1048	1059	1168	1221	1419
迄點	1022		0	0	1.88%	0	0.47%	0
	1036	0		0	0.01%	0	-0.07%	0
	1048	0	0		0	0	0	0
	1059	0	0	0		0	0	0
	1168	0	0	0	0		-0.11%	0
	1221	0	0	0	0	0.73%		0.43%
	1419	0	0	0	0	0	0	

註： $\frac{TLCSP - TDSP}{TLCSP} \times 100\%$

比較結果中出現 0 的部分為前面指出的路徑相同的部分。此外，在 1221 1036 以及 1221 1168 產生了負值，也就是說使用原本最短路徑演算法所產生的路徑較依時性最短路徑產生的結果來的好，探究其原因，因為其產生的值都很小，所以有可能是在計算成本時，系統中對於數字精度上的誤差導致的，所以才發生這種結果。

### 5.3.2 使用控制延滯下三種路徑演算法的結果比較

本節在路徑計算中加入了路口延滯成本的考量，而此路口延滯成本是以號誌所產生的控制延滯成本為主。其三種演算法之間的比較方式如同第 5.3.1 節。

#### 1. 最短路徑與 K 條最短路徑

根據本研究所開發的控制延滯下最短路徑以及控制延滯下 K 條最短路徑所得的結果如表 5.9 所示。

表 5.9 控制延滯下 LCSP 與 KSP 比較結果

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	16238	14	1011.37	27380	25	1999.41
LCSP	16238	14	1011.37	27380	25	1999.41
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	17106	15	1189.11	18872	16	1207.08
LCSP	17106	15	1189.11	18872	16	1207.08

註：路徑計算起始時間：100 分鐘

根據表 5.9 得知，在 K=1 時的路徑結果，不論是 SumNode、SumID 或是旅行時間成本，兩種路徑演算法產生的結果皆完全一致。此外，檢視所有 KSP 的結果，並未發現任何有比所得到的 K=1 的旅行時間來的小，顯示本研究所求得的路口延滯下最短路徑結果確實正確無誤。

## 2. K 條最短路徑與依時性最短路徑

根據所設定的 4 組起迄點，分別計算控制延滯下 TDSP 以及控制延滯下 KSP，其比較結果如表 5.10 所示。

表 5.10 控制延滯下 TDSP 與 KSP 結果比較

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	16238	14	1011.37	27380	25	1999.41
TDSP	16238	14	1013.64	27380	25	1961.83
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	17106	15	1189.11	18872	16	1207.08
TDSP	17106	15	1160.39	18872	16	1204.04

註：路徑計算起始時間：100 分鐘

由表 5.10 的結果顯示，在考慮控制延滯的情況下，KSP 的結果中可以找到 TDSP 所產生的路徑，且同樣都是在 K=1 時找到相吻合的路徑。由此說明即使是考慮了路口延滯成本的因素，KSP 的結果中仍然可以找到與 TDSP 相同的路徑結果。

### 3. 依時性最短路徑與最短路徑

#### (1) 兩種演算法所得路徑的變化情形

以控制延滯下 LCSP 的路徑結果作為基準，比較結果如表 5.11 所示。

表 5.11 控制延滯下 TDSP 與 LCSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	7	16.67%	增加	1	14.3%
不變	35	83.33%	減少	2	28.6%
			不變	4	57.1%

註：路徑計算起始時間：100 分鐘

根據表 5.11 的結果顯示，在 42 條路徑中，僅有 7 條路徑有產生變化，所佔比例約為 16.67%，從比例上來看仍然偏低。除了有可能跟前節所述是由於路段旅行時間資料的差異不大外，另一個可能的導致如此結果的原因，應該是本研究所使用的路口號誌資料的關係，由於本研究所用資料除了中港路、大雅路、文心路三條主要道路的號誌週期有所調整，其餘路口號誌的資料皆相同，因此在計算路口延滯時，其所求得的延滯成本皆相同。此外，控制延滯成本並不會因為時間的變動而有所更改，所以雖然 TDSP 是採依時性的成本來計算，但對路口號誌而言並沒有影響，才導致如此結果。

## (2) 兩種演算法所得路徑的旅行時間成本變化情形

在 TDSP 與 TLCSP 的比較中，本研究以 TLCSP 為基準，進行 TLCSP 與 TDSP 的成本差異比較，結果如表 5.12 所示。

表 5.12 控制延滯下 TLCSP 與 TDSP 成本差異百分比

差值 (%)		起				點		
		1022	1036	1048	1059	1168	1221	1419
迄	1022		0	0	0	0	0.19%	0
	1036	0		0	0	0	0.06%	0.15%
	1048	0	0		0	0	0	0.94%
	1059	0.26%	0	0		0	0	0.24%
點	1168	0	0	0	0		0	0
	1221	0	0	0	0	0		0.22%
	1419	0	0	0	0	0	0	

註： $\frac{TLCSP - TDSP}{TLCSP} \times 100\%$

比較結果中出現 0 的部分為前面指出的路徑相同的部分。而在其他有變化的路徑中，成本的差異同樣很小。

### 5.3.3 使用平均紓解率路口延滯下三種路徑演算法的結果比較

本節在路徑計算中同樣加入了路口延滯成本的考量，但而此路口延滯成本是以考量到車流量對於路口延滯所產生的影響，因此以平均紓解率來作為本研究的路口延滯成本。其三種演算法之間的比較方式如同第 5.3.1 節，但在進行比較各演算法結果之間的差異前，必須先針對所使用的計算紓解率的時段長度計算先決定一合適的時段長度，所以本研究將先比較不同計算紓解率的時段長度所產生的結果。

#### 1. 不同計算紓解率的時段長度的比較

為了選擇一較適合的時段長度來計算平均紓解率，以避免因為時段長度太長或太短以至於所求得之平均紓解率差異情形過大，本研究分別以 1~10 分鐘的時段長度，計算其平均紓解率路口延滯下的路徑成本，並比較其不同時段長度下所求得的總旅行時間成本，選擇一個較合理的時段長度來作為後續的測試和比較。

本研究先從無路口延滯下每一組起迄點的最短路徑中，選取節點數較多的路徑共 7 條路徑，並以這 7 條路徑作為比較基準。

接著將該 7 條路徑，分別以起始時間 100 分鐘及 10 個時段長度的平均紓解率來計算其旅行時間成本，其各路徑計算結果趨勢圖如圖 5.9 所示。圖中 X 軸為時段長度，單位分鐘；Y 軸為旅行時間，單位秒。



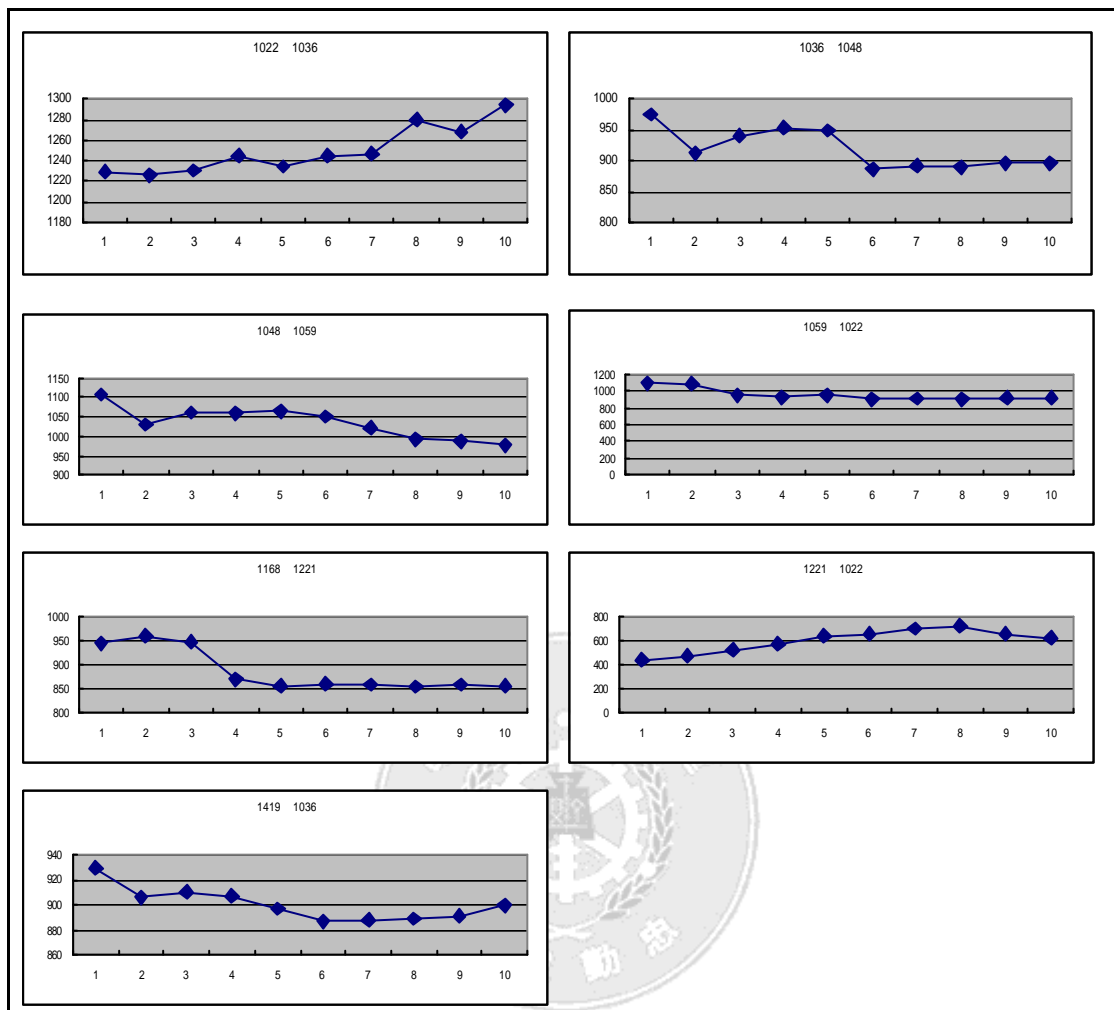


圖 5.9 起始時間 100 分鐘的旅行時間趨勢圖

根據圖 5.9 的結果顯示，不論使用哪一個時段長度，各路徑的結果差異性相當不一樣。因此，為判斷是否是資料本身影響所致，故在分別以起始時間 10 分鐘及 50 分鐘進行計算，結果如圖 5.10、圖 5.11 所示。圖中 X 軸為時段長度，單位分鐘；Y 軸為旅行時間，單位秒。

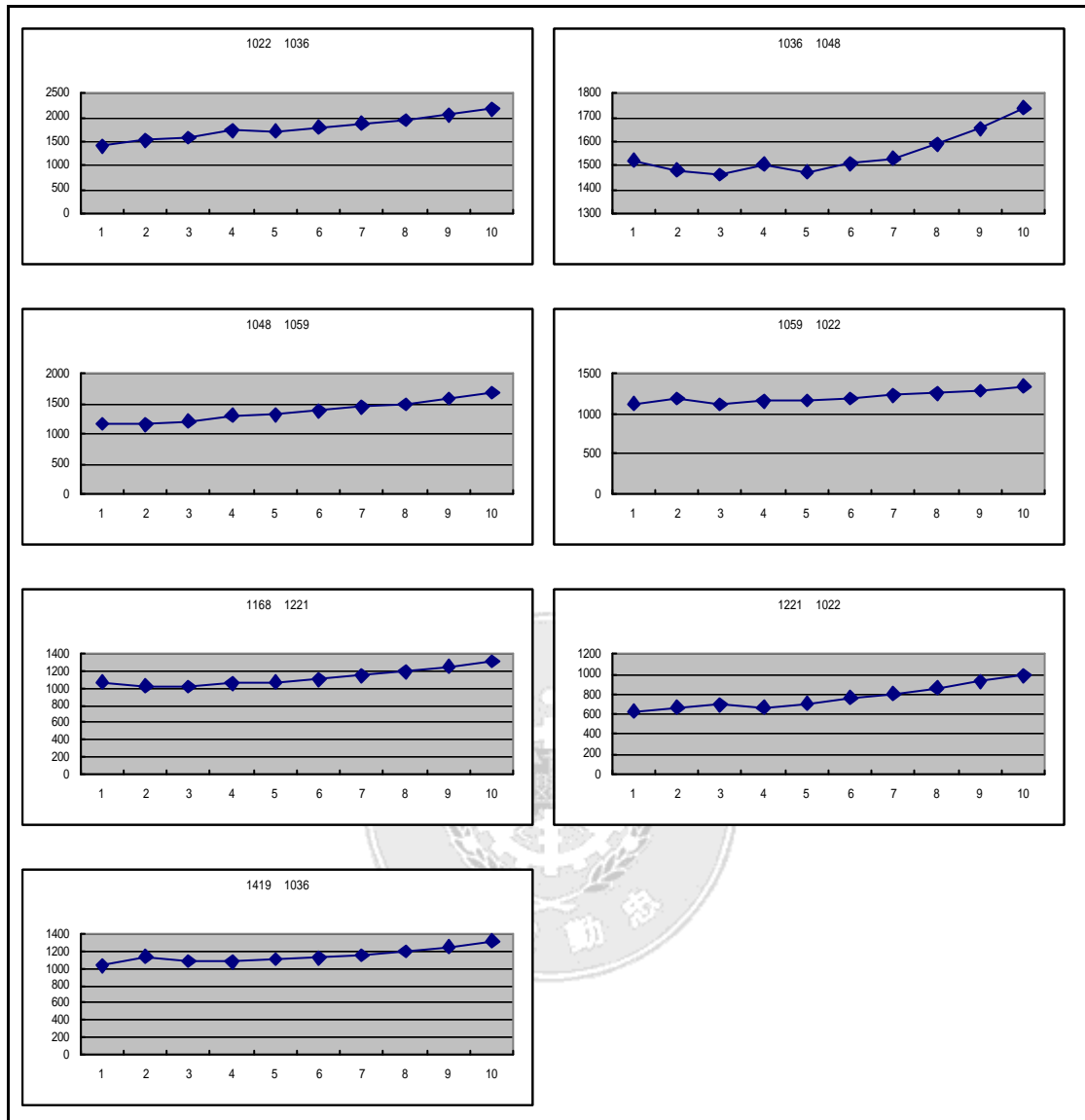


圖 5.10 起始時間 10 分鐘的旅行時間趨勢圖

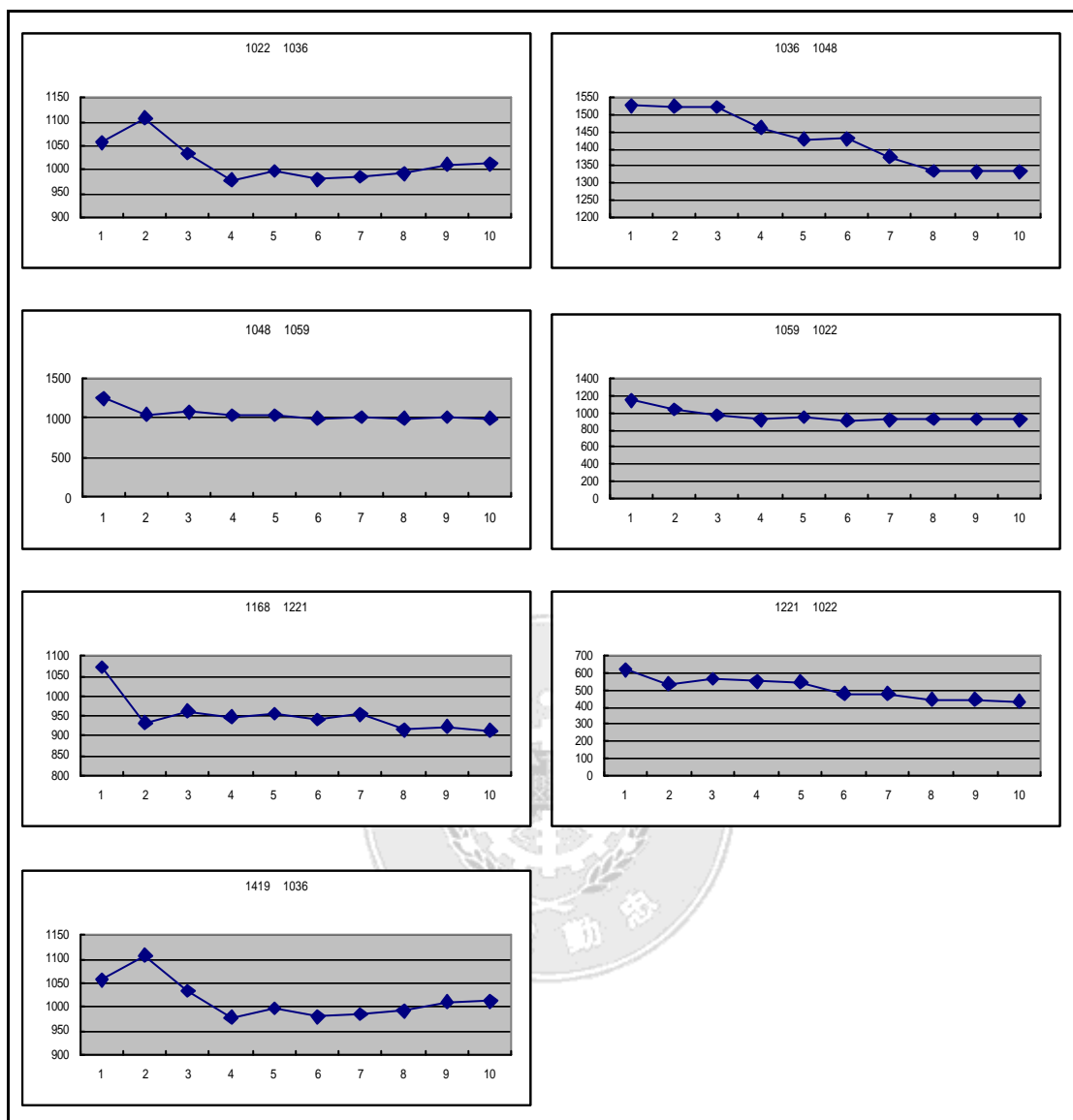


圖 5.11 起始時間 50 分鐘的旅行時間趨勢圖

根據圖 5.10 以及圖 5.11 所示，其結果同樣有明顯的差異，因此研判應該是所使用的資料所致。經過檢查所使用的資料，發現在資料前面時段中，因為系統剛開始模擬，所以其不論是通過路口的車輛數或是路段停等中的車輛數，皆有很多值為 0，而隨著時間的增加，由於系統已經產生越來越多的車輛進入路網，也漸漸達到交通的尖峰時段，結果使的資料數值逐漸增加，也讓所獲得的紓解率變化不一。

因此，為了能夠在進行最短路徑計算時，能夠明顯的反應出交通尖峰對路口延滯所造成的影響，又不讓過度集中或是過度分散的流量對結果產生太大的影響，所以本研究在進行 LCSP 以及 KSP 時，將選擇起始時間為

100 分鐘，並以 30 分鐘作為計算紓解率的時段長度，該 30 分鐘資料除了能夠反應車輛逐漸產生的情形，也能反應尖峰時的交通流量情形，使其能得到一個較為穩定的紓解率，以提供本研究計算路口延滯成本。而在進行計算依時性最短路徑時，由於其選擇資料的時間點會一直隨著時間改變，因此本研究將依據 DYNASMART 在進行平均紓解率計算時所使用的 3 分鐘為時段長度。

## 2. 最短路徑與 K 條最短路徑

根據本研究所開發的平均紓解率路口延滯下最短路徑以及平均紓解率路口延滯下 K 條最短路徑，並以所設定的起始時間 100 分鐘，30 分鐘為計算紓解率時間長度，所得的結果如表 5.13 所示。

表 5.13 平均紓解率路口延滯下 LCSP 與 KSP 比較結果

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	18772	16	654.411	30042	27	1084.35
LCSP	18772	16	654.411	30042	27	1084.35
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1)	18069	15	830.668	23778	21	788.853
LCSP	18069	15	830.668	23778	21	788.853

註：路徑計算起始時間：100 分鐘

根據表 5.13 得知，跟前面兩節的結果相同，在 KSP 的結果中，其第 1 條最短路徑與 SP 的結果相同，且同樣在往後的 K 值中也都未發現比第 1 條路徑的旅行時間還要短的路徑，顯示本研究以平均紓解率作為路口延滯成本所開發出來的路徑演算法同樣正確無誤。

### 3. K 條最短路徑與依時性最短路徑

根據所設定的 4 組起迄點，分別計算平均紓解率路口延滯下依時性最短路徑以及平均紓解率路口延滯下 K 條最短路徑，其比較結果如表 5.14 所示。

表 5.14 平均紓解率路口延滯下 TDSP 與 KSP 結果比較

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1~50)	不存在	不存在	不存在	不存在	不存在	不存在
TDSP	14	16238	593.969	25	27474	962.08
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1~50)	不存在	不存在	不存在	不存在	不存在	不存在
TDSP	16	18099	756.764	19	20913	638.023

註：路徑計算起始時間：100 分鐘

由表 5.14 的結果顯示，在考慮平均紓解率路口延滯的情況下，不論是哪一處起迄點，都未能找到兩者相符合的路徑。檢視 1022 1419 這組路徑，發現在 K=2、3 時，其路徑的前面及後面有部分相同；檢視 1036 1221 這組路徑，發現在 K=1、5 時，其前半條路徑皆與 TDSP 前半部分相同，但後半部的路徑則都不相同；檢視 1059 1022 這組路徑，發現在 K=2、3 時，路徑前面有部分路段相同；檢視 1221 1168 這組路徑，發現在 K=1 時，其前半條路徑與 TDSP 前半部分相同，但後半部分都不相同。

至於造成這個結果的一個可能原因在於所使用的紓解率計算時段長度不同。因此，本研究再以起始時間 100 分鐘，計算紓解率的時段長度 30 分鐘計算 TDSP，以進行比較。結果如表 5.15 所示。

表 5.15 平均紓解率路口延滯下 TDSP 與 KSP 結果比較

	1022 1419			1036 1221		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1~50)	不存在	不存在	不存在	不存在	不存在	不存在
TDSP	14	16238	565.556	25	27474	962.08
	1059 1022			1221 1168		
	SumNode	SumID	旅行時間 (秒)	SumNode	SumID	旅行時間 (秒)
KSP (K=1~50)	不存在	不存在	不存在	不存在	不存在	不存在
TDSP	14	14823	723.264	18	20929	655.237

註：路徑計算起始時間：100 分鐘

根據表 5.15 的結果顯示，雖然計算 TDSP 的平均紓解率改為 30 分鐘的時段長度，但仍然在 K=1~50 之間無法找到與之相符的路徑。檢視 1022 1419 這組路徑，發現在 K=3 時，約前面 2/3 的路徑與 TDSP 相同，後面部分也僅有 2 個節點不同；檢視 1036 1221 這組路徑，發現在 K=1 時，同樣在其路徑前半部分與 TDSP 的前半部分相同；檢視 1059 1022 這組路徑，發現在 K=1、2、3 時，僅有前面剛開始以及最後面的幾個節點相同；檢視 1221 1168 這組路徑，發現在 K=1 時，僅有前面剛開始以及最後面的幾個節點相同。

因此，經由比較結果顯示有可能在考慮平均紓解率下的 TDSP 計算結果，可能無法透過計算平均紓解率下的 KSP 來獲得相同的路徑。但因為本研究所使用的 K 值僅到 50，所以也有可能會出現在更大的 K 值之後。

#### 4. 依時性最短路徑與最短路徑

##### (1) 兩種演算法所得路徑的變化情形

以平均紓解率路口延滯下 LCSP 的路徑結果作為基準，比較結果如表 5.16 所示。

表 5.16 平均紓解率路口延滯下 TDSP 與 LCSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	33	78.57%	增加	4	12.1%
不變	9	21.43%	減少	25	75.8%
			不變	4	12.1%

註：路徑計算起始時間：100 分鐘

根據表 5.16 的結果顯示，在使用平均紓解率作為路口延滯成本的計算下，其路徑改變的結果相當明顯，路徑有改變的路徑下通過的總節點數有 75.8% 的路徑選擇減少通過路口，避免得到更長的旅行時間。顯示使用平均紓解率的路口延滯，對路徑的結果有很明顯的影響。

## (2) 兩種演算法所得路徑的旅行時間成本變化情形

在 TDSP 與 TLCSP 的比較中，本研究以 TLCSP 為基準，進行 TLCSP 與 TDSP 的成本差異比較，結果如表 5.17 所示。

表 5.17 平均紓解率路口延滯下 TLCSP 與 TDSP 成本差異百分比

差值 %	起點						
	1022	1036	1048	1059	1168	1221	1419
迄點	1022	0	0	11.14%	0	0.88%	17.91%
	1036	31.91%	14.74%	14.46%	12.01%	3.28%	0
	1048	0	20.95%	25.79%	0.95%	24.15%	24.23%
	1059	6.78%	10.25%	-0.12%	17.10%	14.20%	
	1168	16.42%	16.62%	15.27%	12.29%	16.81%	
	1221	0	15.03%	16.19%	4.41%	16.31%	
	1419	13.59%	11.61%	1.23%	3.27%	25.66%	

$$\text{註：} \frac{TLCSP - TDSP}{TLCSP} \times 100\%$$

比較結果中出現 0 的部分為前面指出的路徑相同的部分。而在其他有

變化的路徑中，成本的差異明顯增加很多，顯示在考慮以平均紓解率作為路口延滯的情形下，選擇使用 TDSP 的結果將會比使用 LCSP 的結果來的要好，其表現在成本的節省上有很顯著的效益。此外，也顯示出以交通流量作為延滯的考量，確實能夠反映出其對於路徑計算上的影響。

### 5.3.4 綜合比較

在本節中，將分別針對有無路口延滯下 LCSP 的路徑結果、有無路口延滯下 TDSP 的路徑結果進行比較，還會針對 K 值改變對於 KSP 的運算時間的影響以及 LSSP、LCSP 和 TDSP 之間的運算時間的影響比較。

#### 1. 比較有無路口延滯成本考量的 LCSP 路徑結果

本研究針對有無路口延滯對於所計算求得的路徑結果進行比較與分析。

##### (1) 無路口延滯成本與控制延滯成本的 LCSP 路徑結果比較

根據兩種路口延滯考量方式所獲得的 LCSP 結果進行路徑上的比對，以無路口延滯的結果為基準，比對結果如表 5.18 所示。

表 5.18 無路口延滯成本與控制延滯成本 LCSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	13	30.95%	增加	2	15.4%
不變	29	69.05%	減少	7	53.8%
			不變	4	30.8%

註：路徑計算起始時間：100 分鐘

根據表 5.18 的結果顯示，在加入號誌對路徑的影響後，路徑有改變的路徑中有 7 條路徑 (53.8%) 會減少通過路口，以避免增加旅行時間，而在整體路徑結果上僅有 30.95% 的路徑會產生改變，從所佔比例的角度上來看，在 LCSP 的計算中，在本研究中號誌對路徑的影響並不是很明顯。



## (2) 無路口延滯成本與平均紓解率延滯成本的 LCSP 路徑結果比較

根據兩種路口延滯考量方式所獲得的 LCSP 結果進行路徑上的比對，以無路口延滯的結果為基準，比對結果如表 5.19 所示。

表 5.19 無路口延滯成本與平均紓解率延滯成本 LCSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	34	80.95%	增加	28	82.4%
不變	8	19.05%	減少	1	2.9%
			不變	5	14.7%

註：路徑計算起始時間：100 分鐘

根據表 5.19 的結果顯示，在考慮交通流量的情形後，路徑有改變的 34 條路徑中有 28 條路徑 (82.4%) 選擇增加通過路口的情形來避免增加旅行時間，而在整體路徑中有 80.95% 的路徑發生變化。從所佔比例上來看，交通流量對於路徑計算的結果有很顯著的影響。

## 2. 比較有無路口延滯成本考量的 TDSP 路徑結果

## (1) 無路口延滯成本與控制延滯成本的 TDSP 路徑結果比較

根據兩種路口延滯考量方式所獲得的 TDSP 結果進行路徑上的比對，以無路口延滯的結果為基準，比對結果如表 5.20 所示。

表 5.20 無路口延滯成本與控制延滯成本 TDSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	14	33.33%	增加	2	14.3%
不變	28	66.67%	減少	8	57.1%
			不變	4	28.6%

註：路徑計算起始時間：100 分鐘

根據表 5.20 的結果顯示，在加入號誌對路徑的影響後，其結果與 LCSP

大致相同，路徑有變化的比例都明顯偏低，顯示在本研究中號誌對 TDSP 路徑計算的影響並不是很明顯。

## (2) 無路口延滯成本與平均紓解率延滯成本的 TDSP 路徑結果比較

根據兩種路口延滯考量方式所獲得的 TDSP 結果進行路徑上的比對，以無路口延滯的結果為基準，比對結果如表 5.21 所示。

表 5.21 無路口延滯成本與平均紓解率延滯成本 TDSP 路徑結果比較

路徑變化			路徑改變下節點數的變化		
改變	15	35.71%	增加	9	60.0%
不變	27	64.29%	減少	1	6.67%
			不變	5	33.3%

註：路徑計算起始時間：100 分鐘

根據表 5.21 的結果顯示，在加入平均紓解率的路口延滯成本後，TDSP 所計算求的路徑改變比例同樣並不是很高，顯示在 TDSP 的計算中，考慮交通流量所計算而得的平均紓解率對於其結果影響並不是很明顯。其可能原因是，由於 TDSP 所使用的路段旅行時間隨著時間變動而變動，因此在交通流量改變時，路段的旅行時間也會受到影響，所以交通流量的影響已經有轉嫁到路段旅行時間上，所以以平均紓解率來作為路口延滯成本的影響才會不是很顯著。

另外，從模擬結果中，選擇節點編號 1059(中港路交流道附近) 為起點，節點編號 1221(火車站附近) 為迄點。比較其所計算的路徑總旅行時間成本，結果如表 5.22、5.23。

表 5.22 三種路口延滯成本下 LCSP 之旅行時間

	總旅行時間(分)
無路口延滯成本	13.7
平均紓解率路口延滯成本	19.1
控制延滯成本	27.7

表 5.23 三種路口延滯成本下 TDSP 之旅行時間

	總旅行時間(分)
無路口延滯成本	13.8
平均紓解率路口延滯成本	15.4
控制延滯成本	27.7

從兩張表中可以得知使用控制延滯來作為路口延滯的 LCSP 與 TDSP 路徑結果，其由中港交流道附近到火車站附近所需花費的時間皆約為 28 分鐘，這個旅行時間與實際經驗值來比較相當合理。由此可見，以號誌作為路口延滯的依時性最短路徑較為使用其他種延滯來的符合實際所需。

### 3. 路段旅行時間與路口延滯時間二者權重之比較

根據本研究所有路徑旅行時間的比較中可以發現，當在演算法中加入路口延滯成本的考量時，對於其總旅行時間成本所產生變化有著明顯的變化，因此，本研究將比較路段旅行時間與路口延滯時間二者成本的權重大小。

#### (1) 控制延滯下路口延滯成本與路段旅行時間的比較

比較中，從控制延滯成本下的 LCSP 的路徑結果中選取數條通過節點數較多的路徑，比較其平均每個路段旅行時間成本與平均每個路口旅行時間成本，結果如表 5.24 所示。

表 5.24 控制延滯下路口延滯成本與路段旅行時間的比較

起點	迄點	節點數	平均每一個路段的旅行時間成本	平均每一個路口的延滯成本	比值
1022	1036	15	49.25	46.99	1.05
1036	1048	17	49.73	45.94	1.08
1048	1036	17	48.02	46.88	1.02
1059	1221	20	43.21	44.13	0.98
1221	1036	25	39.52	42.47	0.93

註：比值 = 平均路段旅行時間 / 平均路口延滯成本

根據表 5.24 的結果顯示，使用控制延滯所得到的路口延滯成本，其值與路段旅行時間成本相比接近 1 : 1。

#### (2) 平均紓解率延滯下路口延滯成本與路段旅行時間的比較

比較中，平均紓解率延滯下路口延滯成本下的 LCSP 的路徑結果中選取通過節點數約在 15 個節點以上的路徑，比較其平均每個路段旅行時間成本與平均每個路口旅行時間成本，結果如表 5.25 所示。

表 5.25 平均紓解率延滯下路口延滯成本與路段旅行時間的比較

起點	迄點	節點數	平均每一個路段的旅行時間成本	平均每一個路口的延滯成本	比值
1036	1022	15	50.63	3.12	16.21
1036	1419	15	49.60	17.26	2.87
1419	1036	15	49.26	14.67	3.36
1036	1059	14	49.68	2.78	17.88
1168	1022	9	45.80	10.89	4.20

註：比值 = 平均路段旅行時間 / 平均路口延滯成本

從表 5.25 的結果顯示，使用平均紓解率延滯所得到的路口延滯成本，其值與路段旅行時間成本相比，平均每一個路段的旅行時間成本約為平均每一個路口的延滯成本的 2~17 倍不等，其值差異相當大，研判可能是因

為其路口延滯成本值計算主要是以考慮通過路口的車流量為主，而每一個路徑所通過的路段其車流量變化也不相同，因此流量的大小將直接影響所求得的路口延滯成本值。

#### 4. LCSP 與 LSSP 運算時間的比較

為建立一比較基準，本研究隨機選擇一固定起點，計算由該起點出發到路網上所有節點的最短路徑 (one-to-all)，並分別以三種不同的路口延滯成本計算方式，來進行多次運算，計算求得其平均運算時間，結果如表 5.26 所示。

表 5.26 三種路口延滯下 LCSP 與 LSSP 的平均運算時間表

	LCSP 運算時間 ( $10^{-3}$ 秒)	LSSP 運算時間 ( $10^{-3}$ 秒)
無路口延滯成本	35.8	34.3
平均紓解率路口延滯成本	67.2	30.9
控制延滯成本	29.7	31.1

註：此運算時間不包含路徑輸出以及路口延滯計算時間。

根據表 5.26 發現，在本研究所使用的路網上，若不考慮路口延滯的話，兩種演算法的所花費的時間幾乎一樣。但在加入路口延滯成本的影響後，LCSP 的時間便起了變化，在平均紓解率路口延滯部分，其演算法迴圈的運算次數從無路口延滯的 804 次增加到 1221 次，導致其運算時間增加，而控制延滯部分，運算次數反而從 804 次減少到 764 次，因此也使得其運算時間減少。在 LSSP 的部分，不論是否考慮路口延滯，其迴圈的運算次數皆為 573 次，而從運算時間上看來，三者差異並不太大。

而在兩種演算法的比較上來說，在無路口延滯以及控制延滯部分差異並不大，但在平均紓解率路口延滯部分則有明顯的差異，從數據上看來使用 LSSP 的效率會比 LCSP 來的好，因此，若將來要使用平均紓解率做為路口延滯成本算，選擇使用 LSSP 應可以獲得較佳的效率。

但由於本研究所用路網節點數僅為 574 個，因此，不論 LSSP 從暫存節點中選取最小成本的節點，或是 LCSP 重覆計算的次數，多少有所限制。所以應該在進行更大型的路網來進行比較，才能獲得更好的結果。

### 5. LCSP 與 TDSP 運算時間的比較

為建立一比較基準，本研究隨機選擇一固定起點，計算由該起點出發到路網上所有節點的最短路徑 (one-to-all)，並分別以三種不同的路口延滯成本計算方式，來進行多次運算，計算求得其平均運算時間，結果如表 5.27 所示。

表 5.27 三種路口延滯下 LCSP 與 TDSP 的平均運算時間表

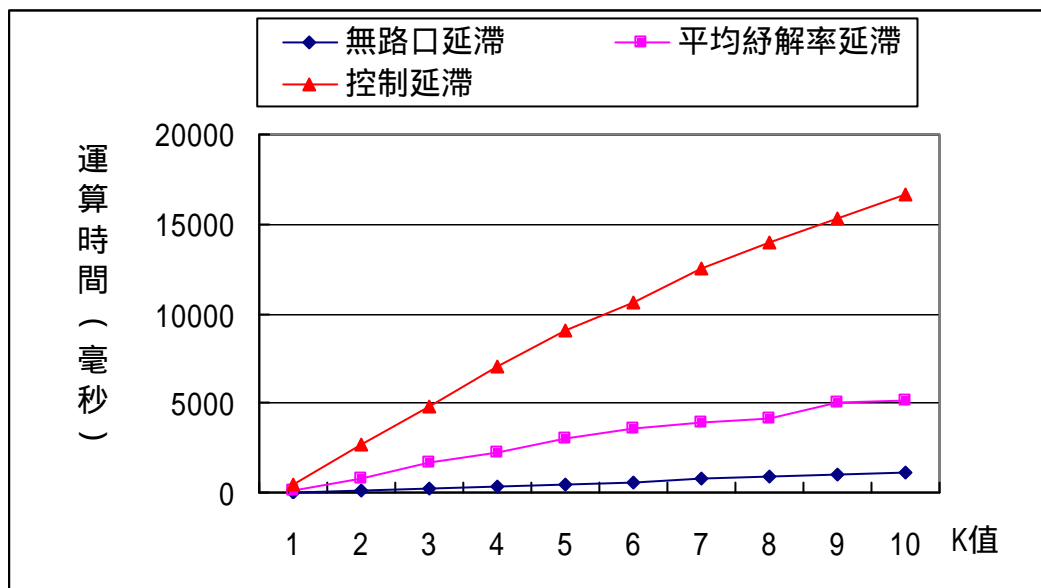
	LCSP 運算時間 ( $10^{-3}$ 秒)	TDSP 運算時間 ( $10^{-3}$ 秒)
無路口延滯成本	35.8	39.1
平均紓解率路口延滯成本	67.2	37.8
控制延滯成本	29.7	34.4

註：此運算時間不包含路徑輸出以及路口延滯計算時間。

根據表 5.27 的結果顯示，不論是否加入路口延滯，在 TDSP 的運算時間上差異不大，檢視其迴圈運算次數無路口延滯有 806 次，平均紓解率路口延滯有 776 次，而控制延滯則僅有 704 次，從數據資料中可以清楚發現運算時間與其運算的迴圈次數有很明顯的關係。而在 TDSP 與 LCSP 的運算時間比較上，在無路口延滯以及控制延滯部分，可以發現 TDSP 的運算時間比 LCSP 稍微多一些，比較其運算次數兩者差異不大，因此其時間應該是在進行 TDSP 的運算時，在取每個時間的路段或路口資料時需要進行的一些判斷以及時間的轉換，因此花費的些許時間，導致其運算時間較 LCSP 來的多。而在平均紓解率的部分，雖然 TDSP 需花費時間在取每個時間點的資料，但從迴圈的運算次數來看，LCSP 的 1221 次遠比 TDSP 的 776 次多很多，因此 LCSP 的運算時間也比 TDSP 的運算時間來的長。

### 6. K 值改變對運算時間的影響

在路口延滯下 K 條最短路徑的測試比較上，主要針對比較有無路口延滯下 K 條最短路徑在系統中的運算時間。測試中，本研究給定一組起迄點計算一到一 (one-to-one) 的 K 條最短路徑，並設定 K 值從 1~10，起始時間為 100 分鐘，平均紓解率的時段長度為 3 分鐘，結果如圖 5.12 所示。



註：此運算時間不包含路徑輸出，但包含路口延滯計算時間。

圖 5.12 K 條最短路徑運算時間比較圖

從圖中可以看出系統計算 K 條最短路徑時所花費的時間，隨著 K 值的增加，很平均且穩定的上升。經過數據的比對發現 K 值每增加 1，在無路口延滯成本時，其運算時間平均增加約 116.3 毫秒，平均紓解率延滯平均增加約 560.8 毫秒，而控制延滯則平均增加約 1805.2 毫秒。另外，檢視三種路口延滯成本的路徑運算時間後發現，K 值的改變對系統所花費的時間增加幅度不大，除了從 K = 1 到 K = 2 的運算時間差異約 6 倍左右外，其餘 K 值每增加 1，平均約增加 1.3 倍的運算時間，經分析後得知之所以 K 值從 1 增加到 2 的變化會如此大，乃因為 K = 1 時為單純的最短路徑計算，在其計算過程中，並不需要對路段進行任何改變，因而與 K = 2 時會有非常明顯的差距，所以此結果可以接受。而從 Yen's Algorithm 的時間複雜度  $O(Kn^3)$  來看，在路網結構不變下，若單純考慮改變 K 值，系統運算時間增加幅度主要也是受到 K 值的影響。

## 第六章 結論與建議

### 6.1 結論

#### 1. 本研究所開發的演算法與 Chen 和 Yang 所開發的演算法不同之處

在 Chen 和 Yang 所開發的演算法中，其乃係利用下游節點來選擇可通行的時相，以計算獲得「最早離開時間」，並經由重複計算而得到一最短路徑結果。而本研究所開發的演算法並非以可通行的時相及方向做為路口延滯的計算基礎，在本研究中車輛一旦要通過某一路口，就必須給予一個屬於該路口的路口延滯成本，並將該延滯成本加到所有相連的下游節點成本標號值中，不會因為下游節點的不同而有不同的延滯成本。在通過路口的延滯成本方面，Chen 和 Yang 僅考慮號誌時相的時間與停等次數權重，而本研究乃考慮車輛因為號誌控制的關係所產生的各種延滯成本，以及交通流量對於車輛在路口紓解時所產生的延滯成本。因此，不論在演算法或是路口延滯成本的考慮及計算方式，本研究與 Chen 和 Yang 的研究皆不相同。

#### 2. 考量路口延滯對最短路徑計算結果的影響

在分別比較兩種主要的路口延滯計算結果後，發現使用路口號誌所計算得到的控制延滯，計算出來的總旅行時間與實際的旅行時間相當接近，可見其能提供更可靠的路徑資訊。而利用平均紓解率法來獲得的路徑旅行時間成本，雖然距離實際旅行時間仍有一段差距，但其所求得的路徑結果與無路口延滯下的路徑結果差異相當明顯，顯示其能反映出路口流量變化對路徑計算時所產生的影響。因此，在進行路徑計算時考慮路口延滯問題確實可以提供更為可靠的路徑資訊。

#### 3. 依時性最短路徑實際上的應用

根據數值實驗的結果，在無路口延滯以及控制延滯下，因為本研究所使用的資料關係，使其旅行時間在使用 TDSP 和使用 LCSP 所計算得到的多數結果差異不大，但在使用考慮交通流量的平均紓解率做為路口延滯成本後，兩種路徑演算法所得到的結果有了非常明顯的差距。因此，未來在提供路徑導引資訊時，若能考慮時間因素，應當可以提供用路人較佳的



## 路徑資訊

### 4. 利用物件導向系統分析的優點

本研究乃係利用物件導向的分析方式來進行系統分析，並依照其分析結果來進程式撰寫。透過物件導向的觀念，使原本的 DynaTAIWAN 系統能夠方便的加入本研究所開發的路徑演算法。而在彼此不同的路徑演算法之間，也因為使用到物件設計觀念的策略性設計樣式，而使系統不僅能夠方便的切換不同的演算法，也不需為了改變不同演算法而對系統程式大肆修改。可見物件導向式的分析與建置，提供系統開發者與維護管理者很具彈性的實質幫助。

## 6.2 建議

### 1. 路段旅行時間與的路口延滯成本計算方式

從數值實驗結果可以知道，路段旅行時間與路口延滯成本的計算方式，明顯的對結果產生影響。而本研究在路段旅行時間的計算中，並未對在路段上可能發生的延滯問題進行討論，這些都有可能直接或間接影響到路短旅行時間。另外，在路口延滯成本的考慮上，也是透過假設性的資料來計算獲得。因此，未來如果能夠考慮其他可能影響路段旅行速率的因子，以及使用更符合實際情形的號誌情況如觸動號誌、號誌連鎖等等因素，相信所獲得的路徑資訊將更為可靠。

### 2. 路口轉向的考慮

在交通路網中，路口轉向的影響也十分重要，但在本研究中並未加入的其對路徑計算所產生的延滯成本，因此，建議後續研究能夠加入路口轉向的影響，以獲得更符合實際情形的路徑結果。

### 3. 即時資訊的使用

本研究未使用即時資訊，但在智慧型運輸系統的發展中，即時資訊扮演著極為重要的角色，過即時資料也能更加反映出路段或路口的延滯情形，對於路徑計算上也能提供更精確的成本資訊。

#### 4. 分散式演算法的考量

路徑的計算常伴隨著龐大的路網資料，尤其在進行 K 條最短路徑計算時，龐大的資料處理以及不斷重覆的運算，常會為系統帶來極大的負擔。建議後續研究可以考慮分散式的演算法，藉以提高系統的計算效率。

#### 5. 更多樣化的路網資料與測試

由於研究上的限制，並未能對不一樣的路網進行測試，而資料的取得也不能完全反應現實生活的情形。因此，建議後續的研究能夠取得其他地區的路網資料以及道路屬性資料，來進行系統測試，此外，也能在測試中加入路徑資訊的回饋機制，以獲得更為精確的分析。



## 參考文獻

1. 交通部 (2001), 「台灣地區智慧型運輸系統綱要計劃」。
2. 行政院 (2002), 「挑戰 2008：國家重點發展計劃 (2002~2007)」。
3. 交通部運輸研究所 (2001), 「二〇〇一年台灣地區公路容量手冊」。
4. 交通部運輸研究所 (2001), 「台灣地區發展智慧型運輸系統 (ITS) 系統架構之研究 ( )」。
5. 交通部運輸研究所 (2003), 「區域級智慧型運輸系統示範計劃 核心交通與預測系統 (第一年期)」。
6. 交通部運輸研究所 (1998), 「智慧型運輸系統 (ITS) 發展演進與相關技術之探討」。
7. 交通部運輸研究所 (2001), 「台灣地區發展智慧型運輸系統 (ITS) 系統架構之研究」。
8. 朱亞雷 (2001), 「策略性人力資源系統物件導向式分析、設計方法」, 朝陽科技大學資訊管理研究所碩士論文。
9. 江文聲 (2001), 「動態隨機時間相依路網可靠路徑選擇」, 國立臺灣大學土木工程學研究所碩士論文。
10. 卓訓榮、林文斌 (1991), 「YEN 第 K 條最短路徑演算法及其應用」, 交通運輸, 第 13 期, 頁 201-215。
11. 卓訓榮、陳信雄 (1993), 「電腦平行處理技術在 YEN 第 K 條最短路徑演算法之應用」, 土木水利, 第 19 卷, 第 4 期, 頁 21-33。
12. 邱炫儒 (2002), 以文件式 Model-View-Controller 設計樣式為基礎的應用系統開發方法, 中原大學資訊管理研究所碩士論文。
13. 洪百賢 (2003), 「KSP 演算法於 CORBA-Based 分散式架構績效評估之研究」, 逢甲大學交通工程與管理學系碩士班碩士論文。
14. 胡大瀛 (1996), 「DYNASMART 在動態交通指派模式中之應用」, 中華民國運輸學會第 11 屆研討會, 頁 873-883。
15. 胡大瀛 (1998), 「模擬式動態交通指派模式之研究」, 中華民國運輸學會第十三屆學術論文研討會, 頁 791-799。
16. 胡大瀛 (2001), 「模擬式動態交通指派模式之研究」, 運輸計劃季刊, 第三十卷, 第一期, 頁 1-32。

17. 胡大瀛、江鴻昇、林蔚明、陳顯文 (2002), 「K 條最短路徑演算法之效率比較」, *現代交通*, 第二十九期, 頁 16-27。
18. 陳文能 (2003), 「智慧型運輸系統下 動態運輸規劃模型之建立」, 逢甲大學交通工程與管理學系碩士班論文。
19. 陳慧琪 (2000), 「時間相依最短路徑問題演算方法之研究」, 國立交通大學運輸工程與管理系碩士論文。
20. 葉道明, 郭文源, 吳漢璋 (2000), 「以設計樣式為基礎之物件導向設計再造工程」, 第 11 屆物件導向技術及應用研討會
21. 趙光正譯 (2002), 「UML 與樣式徹底研究 物件導向分析與設計以及統一流程入門」, 培生教育出版集團。
22. 鄭家瑜 (2000), 「C++ 物件導向程式設計入門與應用」, 博碩文化股份有限公司。
23. 薛念林, 賴聯福, 張世璟 (2003), 「應用設計樣式以實作 MVC 架構」, 第 14 屆物件導向技術及應用研討會。
24. Ahuja, R. K., Magnanti, T. L. and Orlin, J. B. (1993), *Network Flows : Theory , Algorithms , and Applications*, Prentice-Hall Inc.
25. Chen, Y. L., and Yang, H. H. (2000), "Shortest paths in traffic-light networks," *Transportation Research Part B* 34, pp.241-253.
26. Chen, Y. L., and Yang, H. H. (2003), "Minimization of travel time and weighted number of stops in a traffic-light network," *European Journal of Operational Research* 144, pp.565-580.
27. Dijkstra, E. W. (1959), "A Note on Two Problems on Connection with Graphs," *Numer.Math.*1, pp.395-412.
28. Dion, F., Rakha, H. and Kang, Y. S. (2004), "Comparison of delay estimates at under-saturated and over-saturated pre-timed signalized intersections," *Transportation Research Part B* 38, pp.99-122.
29. Fu, L. and Rilett, L. R. (1998), "Expected Shortest Paths in Dynamic and Stochastic Traffic Networks," *Transportation Research PartB*, Vol.32, No.7, pp.499-516.
30. Hall, R. W. (1986), "The Fastest Path through a Network with Random Time-Dependent Travel Times," *Transportation Science*, Vol.20, No.3, pp.182-188.

31. Miller-Hooks, E. D., and Mahmassani, H. S. (1998), "Least Possible Time Paths in Stochastic, Time-Varying Networks," *Computer Operations Research*, Vol.25, No.12, pp.1107-1125.
32. Shier, D. R. (1979), "On Algorithm for Finding the K Shortest Paths in a Network," *Network*, Vol. 9, pp. 195-214.
33. Yen, J. Y. (1971), "Finding The K Shortest Loopless Paths in a Network," *Management Science*, Vol.17, No.11, pp.712-716, July.
34. Ziliaskopoulos, A. K. and Mahmassani, H. S. (1996), "A Note On Least Time Path Computation Considering Delays and Prohibitions for Intersection Movements," *Transportation Research PartB*, Vol.30, No.5, pp.359-367.
35. Ziliaskopoulos, A. K. and Mahmassani, H. S. (1996), "Time-Dependent, Shortest-Path Algorithm for Real-Time Intelligent Vehicle Highway System Applications," *Transportation Research Record* 1408, pp94-100.
36. Ziliaskopoulos, A. K., Kotzinos, D. and Mahmassani, H. S. (1997), "Design and Implementation of Parallel Time-Dependent Least Time Path Algorithm for Intelligent Transportation System Applications," *Transportation Research PartC*, Vol.5, No.2, pp.95-107.

```

/*
** Copyright (c) Feng Chia University. 2003-2004. All Rights Reserved.
*/
#include "LCSP.h"
#include "../CombineNetwork.h"
// Construction/Destruction
extern CombineNetwork *GetData;
CLCSP::CLCSP()
{
}
CLCSP::~CLCSP()
{
}
NodeList CLCSP::GenSP(CNodeAdjList* cpNodeAdjList, USI uOrigin, USI uDes)
{
    _uSPNumOfNode = 1 + cpNodeAdjList->GetMAXNodeID();
    Node = new SPNode [_uSPNumOfNode];
    vector<USI> uPath;
    vector<USI> SEL;
    bool flagDischarR = false;
    double tmpcost1;
    double tmpcost2;

    double StartTime = 30; // 計算路徑起始時間
    _Tn = StartTime;

    TotalStruct TotalData;
    // 演算法初始化開始
    USI i;
    for ( i=0; i<= _uSPNumOfNode-1 ; i++)
    {
        Node[i].Length = 99999;
        Node[i].PreNode = 99999;
        Node[i].Status = 'n';
    }
    Node[uOrigin].Length = 0;
    Node[uOrigin].PreNode = uOrigin;
    SEL.push_back(uOrigin);

    // 開始計算 Label Correcting Algorithm
    while ( SEL.empty() == false )
    {
        _uCurrentNode = SEL.front(); // 依據 FIFO 從 SEL 中取得 _uCurrentNode
        SEL.erase(SEL.begin()); // 移除 SEL 中被選取的最前面的點
        NodeList uNodes = cpNodeAdjList->GetLinkedNodes(_uCurrentNode); // 將找出所有下游連接點
        // 計算所有下游連接點
        while ( uNodes.empty() == false )
        {
            _uConnectNode = uNodes.front();
            uNodes.erase( uNodes.begin() );
            CLink* cpLink = cpNodeAdjList->GetLink( _uCurrentNode , _uConnectNode);
            _Tm = Node[ _uCurrentNode ].Length; // _Tm 起始值
            // 路口延滯
            // 控制延滯
            // IntersectionDelay = cpLink->GetInterSignalDelay( _uConnectNode , _uCurrentNode);
            // 平均紓解率延滯
            // IntersectionDelay = cpLink->GetInterAvgDischarRate( _Tn, _uCurrentNode ,
            _uConnectNode);

```



```

    _Tm = Node[ _uCurrentNode ].Length + _IntersectionDelay;
    //////////////////////////////////////
    TotalData = GetData->GetCBNDData(_Tn, _uCurrentNode, _uConnectNode);
    _Speed = TotalData.Vel;
    cpLink = cpNodeAdjList->GetLink( _uCurrentNode , _uConnectNode);
    if ( Node[ _uConnectNode ].Length > _Tm + cpLink->GetMaralCst(_Speed) )
    {
        Node[ _uConnectNode ].Length = _Tm + cpLink->GetMaralCst(_Speed);
        Node[ _uConnectNode ].PreNode = _uCurrentNode;
        USI* selcheck;
        selcheck = find(SEL.begin() , SEL.end() , _uConnectNode);
        if(selcheck == SEL.end())
        {
            SEL.push_back(_uConnectNode);
        }
    }
    else if ( Node[ _uConnectNode ].Length == _Tm + cpLink->GetMaralCst(_Speed) &&
flagDischarR == true)
    {
        tmpcost1 = cpLink->GetInterAvgDischarRate( _Tn, Node[ _uConnectNode ].PreNode ,
_uConnectNode);
        tmpcost2 = cpLink->GetInterAvgDischarRate( _Tn, _uCurrentNode, _uConnectNode);
        if ( tmpcost1 > tmpcost2)
        {
            Node[ _uConnectNode ].PreNode = _uCurrentNode;
            USI* selcheck;
            selcheck = find(SEL.begin() , SEL.end() , _uConnectNode);
            if(selcheck == SEL.end())
            {
                SEL.push_back(_uConnectNode);
            }
        }
    }
}
// 標記所有節點
for ( i=0; i<= _uSPNumOfNode-1 ; i++)
{
    Node[i].Status = 'I';
}
// 建立路徑資料庫
PathList Path;
USI desnode;
_PathList.push_back(Path); // 第 0 個位置置入空的路徑(方便取路徑)
for( USI z=0 ; z<DesNode.size() ; z++)
{
    desnode = DesNode[z];
    Path._Path.push_back(desnode);
    for( i=desnode ; i !=uOrgin ; )
    {
        Path._Path.insert( Path._Path.begin(), Node[i].PreNode);
        i = Node[i].PreNode;
    }
    _PathList.push_back(Path);
    Path._Path.clear();
}
uPath = _PathList[1]._Path;
////////////////////////////////////
// 輸出為檔案
USI counter=1;
fstream SPFileOut;
fstream SPcostFileOut;

```

```

SPFileOut.open("../DTA//SP//PathDataBase//LCSP-path-.txt", ios::out|ios::app);
SPcostFileOut.open("../DTA//SP//PathDataBase//LCSP-cost-.txt", ios::out|ios::app);

SPFileOut << "Those shortest pathes from 'Node " << uOrigin << "': " << endl;
for( i=1 ; i<_PathList.size() ; i++)
{
    SPFileOut << "to 'Node" << _PathList[i]._Path[_PathList[i]._Path.size()-1] << "': " << endl << " ";
    for( z=0 ; z<_PathList[i]._Path.size() ; z++)
    {
        SPFileOut << _PathList[i]._Path.at(z) << " ";
        if(counter%10==0)
        {
            SPFileOut << endl << " ";
        }
        counter++;
    }
    SPFileOut << endl;
    SPFileOut << "    The total travel time : ";
    SPcostFileOut << Node[_PathList[i]._Path.at(z-1)].Length << "    seconds " << endl;
    SPFileOut << endl;
    counter = 1;
}
SPFileOut << endl;
//////////
//螢幕輸出路徑
cout << endl << "The shortest is produced by lebal correcting algorithm.";
cout << endl << "The shortest path of " << uOrigin << " to " << uDes << " is : ";
for( i=0; i<uPath.size()-1 ; i++)
{
    cout << uPath.at(i) << " -> ";
}
cout << uDes << endl;
cout << "Total nodes are : " << uPath.size() << endl;
cout << "The total travel time of this shortest path = " << Node[uDes].Length << endl;

delete [] Node;
return uPath;
}

```



```

/*
** Copyright (c) Feng Chia University. 2003-2004. All Rights Reserved.
*/
#include "LSSP.h"
#include "../CombineNetwork.h"
// Construction/Destruction
extern CombineNetwork *GetData;
CLSSP::CLSSP()
{
    flagKSP = false; // 若單純計算最短路徑則須宣告 LinkStatus 以及 顯示結果
}
CLSSP::~CLSSP()
{
}

NodeList CLSSP::GenSP(CNodeAdjList* cpNodeAdjList, USI uOrigin, USI uDes)
{
    _uSPNumOfNode = 1 + cpNodeAdjList->GetMAXNodeID();
    if( !flagKSP )
    {
        LinkStatus=new char *[_uSPNumOfNode];
        for(USI j=0 ; j<_uSPNumOfNode ; j++)
        {
            LinkStatus[j]=new char [_uSPNumOfNode];
        }
        for (USI m=1 ; m<_uSPNumOfNode ; m++)
        {
            for (USI n=1 ; n<_uSPNumOfNode ; n++)
            {
                LinkStatus[m][n]='o';
            }
        }
    }

    SEL = new double [_uSPNumOfNode];
    Node = new SPNode [_uSPNumOfNode];
    vector<USI> uPath;
    bool flagDischarR = false;
    double tmpcost1;
    double tmpcost2;
    double StartTime = 30; // 計算路徑起始時間
    _Tn = StartTime;
    TotalStruct TotalData;

    // 演算法初始化 開始
    USI i;
    for ( i=0 ; i <= _uSPNumOfNode-1 ; i++)
    {
        Node[i].Length = 99999;
        Node[i].PreNode = 99999;
        Node[i].Status = 'n';
        SEL[i] = 99999;
    }
    Node[uOrigin].Length = 0;
    Node[uOrigin].PreNode = uOrigin;
    SEL[uOrigin] = Node[uOrigin].Length;
    // Label Setting Algorithm 計算開始
    _uCurrentNode = uOrigin;
    SEL[_uCurrentNode] = 99999;
    Node[_uCurrentNode].Status = 'l';

```

```

while ( _uCurrentNode != uDes)
{
    NodeList uNodes = cpNodeAdjList->GetLinkedNodes(_uCurrentNode);
    bool flagNearExist=false; // 若 CurrentNode 所有連接點皆被 label 則傳回 NULL
    while ( uNodes.empty() == false )
    {
        _uConnectNode = uNodes.front();
        uNodes.erase( uNodes.begin() );
        flagNearExist=false;
        if (LinkStatus[_uCurrentNode][_uConnectNode] != 'c' )
        {
            flagNearExist=true;
            if ( Node[ _uConnectNode ].Status != 'I' )
            {
                CLink* cpLink = cpNodeAdjList->GetLink( _uCurrentNode , _uConnectNode);
                _Tm = Node[ _uCurrentNode ].Length;
                //////////////////////////////////// 路口延滯 ////////////////////////////////////
                // 控制延滯
                // _IntersectionDelay = cpLink->GetInterSignalDelay( _uConnectNode ,
                _uCurrentNode);

                //平均紓解率延滯
                // _IntersectionDelay = cpLink->GetInterAvgDischarRate( _Tn, _uCurrentNode ,
                _uConnectNode);

                _Tm = Node[ _uCurrentNode ].Length + _IntersectionDelay;
                ////////////////////////////////////
                TotalData = GetData->GetCBNDData(_Tn, _uCurrentNode, _uConnectNode);
                _Speed = TotalData.Vel;
                if ( Node[ _uConnectNode ].Length > _Tm + cpLink->GetMaralCst(_Speed))
                {
                    Node[ _uConnectNode ].Length = _Tm + cpLink->GetMaralCst(_Speed);
                    Node[ _uConnectNode ].PreNode = _uCurrentNode;
                    SEL[_uConnectNode] = Node[_uConnectNode].Length;
                }
                else if ( Node[ _uConnectNode ].Length == _Tm +
                cpLink->GetMaralCst(_Speed) && flagDischarR == true)
                {
                    tmpcost1 = cpLink->GetInterAvgDischarRate( _Tn,
                    Node[ _uConnectNode ].PreNode , _uConnectNode);
                    tmpcost2 = cpLink->GetInterAvgDischarRate( _Tn, _uCurrentNode,
                    _uConnectNode);

                    if ( tmpcost1 > tmpcost2)
                    {
                        Node[ _uConnectNode ].PreNode = _uCurrentNode;
                        SEL[_uConnectNode] = Node[_uConnectNode].Length;
                    }
                }
            }
        }
    }
    if( flagNearExist==false )
    {
        return uPath;
    }
    // Get CurrentNode
    double min;
    min = 99999;
    for(USI j=1 ; j <= _uSPNumOfNode-1 ; j++)
    {
        if( SEL[j] < min )
        {
            min = SEL[j];
            _uCurrentNode = j;
        }
    }
}

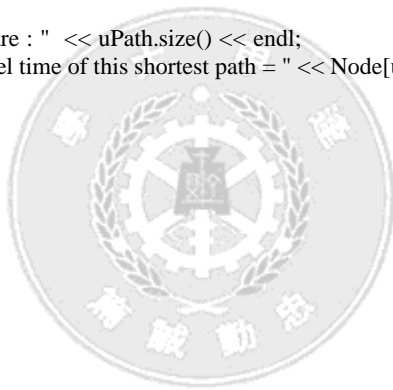
```

```

    }
}
SEL[_uCurrentNode] = 9999;
Node[_uCurrentNode].Status = 'I';
}
//////////
// 建立路徑
uPath.push_back(uDes);
for( i=uDes ; i !=uOrigin ;)
{
    uPath.insert( uPath.begin(), Node[i].PreNode);
    i = Node[i].PreNode;
}
//螢幕顯示路徑
if( !flagKSP )
{
    cout << endl << "The shortest is produced by lebal setting algorithm.";
    cout << endl << "The shortest path of " << uOrigin << " to " << uDes << " is : ";
    for( i=0; i<uPath.size()-1 ; i++)
    {
        cout << uPath.at(i) << " -> ";
    }
    cout << uDes << endl;
    cout << "Total nodes are : " << uPath.size() << endl;
    cout << "The total travel time of this shortest path = " << Node[uDes].Length << endl;
}

delete [] SEL;
if( !flagKSP )
{
    delete [] Node;
}
return uPath;
}

```



```

/*
** Copyright (c) Feng Chia University. 2003-2004. All Rights Reserved.
*/
#include "KSP.h"
#include "../CombineNetwork.h"
// Construction/Destruction
// *****
/* Method:Insert(NodeList uPath,SPNode* Node,bool flagCheckExist) */
/* Description: 置入路徑結構 */
// *****
void ListN::Insert(NodeList uPath,SPNode* Node,bool flagCheckExist)
{
    PathStruct tmpPS;
    bool fEleNotExist; // flag of element Exists or not

    if(flagCheckExist)
    {
        for(USI i=0 ; i<_KPath.size() ; i++)
        {
            if (equal(_KPath[i]._uKPath.begin(), _KPath[i]._uKPath.end(), uPath.begin())==false)
            {
                fEleNotExist=true;
            }
            else
            {
                fEleNotExist=false;
                break;
            }
        }
    }
    if(fEleNotExist || !flagCheckExist)
    {
        tmpPS._uKPath =uPath;
        tmpPS.Length = new double [uPath.size()];

        for(USI i=0 ; i < uPath.size() ; i++)
        {
            tmpPS.Length[i] = Node[uPath.at(i)].Length;
        }

        _KPath.push_back(tmpPS);
    }
}
// *****
/* Method:Insert(PathStruct PS) */
/* Description: 置入路徑 nodes。 */
// *****
void ListN::Insert(PathStruct PS) // pass in structure "PathStruct"
{
    bool fEleExist=false; // flag of element Exists
    if(_KPath.size()==0)
        _KPath.push_back(PS);
    else
    {
        for(USI i=0 ; i<_KPath.size() ; i++)
        {
            if (equal(_KPath[i]._uKPath.begin(), _KPath[i]._uKPath.end(),
PS._uKPath.begin())==true)
            {
                fEleExist=true;
                break;
            }
        }
    }
}

```

```

    }
    }
    if(!fEleExist)
        _KPath.push_back(PS);
}

}

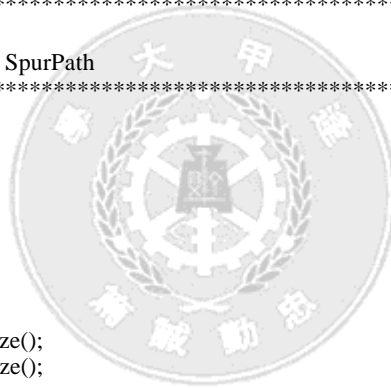
/*****
/* Method:CopyPS(USI index)
/* Description: 複製路徑結構
*****/
PathStruct ListN::CopyPS(USI index)
{
    PathStruct tmpPS;
    tmpPS._uKPath = _KPath.at(index)._uKPath;
    tmpPS.Length=new double[_KPath[index]._uKPath.size()];

    for(USI i=0 ; i<_KPath[index]._uKPath.size() ; i++)
    {
        tmpPS.Length[i] = _KPath[index].Length[i];
    }
    return tmpPS;
}

/*****
/* Method:Union()
/* Description: 聯集 RootPath 與 SpurPath
*****/
PathStruct ListN::Union()
{
    double* Length;
    USI i;
    double base;
    USI size;
    USI rsize; // RootPath size
    USI ssize; // SpurPath size
    rsize = _KPath[0]._uKPath.size();
    ssize = _KPath[1]._uKPath.size();
    size = rsize+ssize-1;
    Length = new double [size];
    for(i=0 ; i<rsize ; i++)
    {
        Length[i] = _KPath[0].Length[i];
    }
    base = _KPath[0].Length[_KPath[0]._uKPath.size()-1];
    for(i=1 ; i<ssize ; i++)
    {
        _KPath[0]._uKPath.push_back(_KPath[1]._uKPath[i]);
        Length[rsize-1+i] = base + _KPath[1].Length[i];
    }
    _KPath[0].Length = new double[size];
    for(i=0;i<size;i++)
    {
        _KPath[0].Length[i]=Length[i];
    }
    return _KPath[0];
}

/*****
/* Method:GetMinPath()
/* Description:取得最短路徑
*****/
PathStruct ListN::GetMinPath()
{
    USI index;

```



```

double Min = 99999;
PathStruct tmpPS;
for(USI i=0 ; i<_KPath.size() ; i++)
{
    if ( _KPath[i].Length[_KPath[i]._uKPath.size()-1] < Min )
    {
        Min = _KPath[i].Length[_KPath[i]._uKPath.size()-1];
        index = i;
    }
}
tmpPS = CopyPS(index);
_KPath.erase(&_KPath.at(index));
return tmpPS;
}
/*****
/* Method:GetPathNode(USI NumofPath,USI NumofNode)                */
/* Description:取得節點編號                                         */
/*****
USI ListN::GetPathNode(USI NumofPath,USI NumofNode)
{
    if( NumofNode < _KPath[NumofPath]._uKPath.size())
    {
        return _KPath[NumofPath]._uKPath.at(NumofNode);
    }
    else
    {
        return -1;
    }
}
/*****
/* Method:GetPathLength(USI NumofPath,USI NumofNode)              */
/* Description:取得路徑旅行時間成本                               */
/*****
double ListN::GetPathLength(USI NumofPath,USI NumofNode)
{
    return _KPath[NumofPath].Length[NumofNode];
}
/*****
/* Method:GetRootPath(USI NumofK,USI IofY)                         */
/* Description:建立根路徑                                          */
/*****
PathStruct CKSP::GetRootPath(USI NumofK,USI IofY)
{
    PathStruct PS;
    USI i;
    PS.Length=new double[IofY+1];
    for(i=0 ; i<=IofY ; i++)
    {
        PS._uKPath.push_back(L0->GetPathNode(NumofK,i));
        PS.Length[i] = L0->GetPathLength(NumofK,i);
    }
    return PS;
}
CKSP::CKSP()
{
}
CKSP::~~CKSP()
{
}
NodeList CKSP::GenSP(CNodeAdjList* cpNodeAdjList, USI uOrgin, USI uDes)
{
    USI IofY,JofY,KofY; // 程式裡面與演算法的 i, j, k 相對應

```

```

USI m,n;
_uNomOfK = 10;          // 欲求的 K 條路徑
flagKSP = true; // 表示由 KSP.cpp 宣告 LinkStatus
_uSPNumOfNode = 1 + cpNodeAdjList->GetMAXNodeID();
LinkStatus = new char *[_uSPNumOfNode];
for(USI j=0 ; j<_uSPNumOfNode ; j++)
{
    LinkStatus[j]=new char [_uSPNumOfNode];
}
//路段狀態初始化
for ( m=1 ; m<_uSPNumOfNode ; m++)
{
    for ( n=1 ; n<_uSPNumOfNode ; n++)
    {
        LinkStatus[m][n]='o';
    }
}
L0=new ListN();
L1=new ListN();
L2=new ListN();
//取得第一條最短路徑 並置入 L0 中 開始
NodeList uKPath = CLSSP::GenSP(cpNodeAdjList, uOrigin, uDes);
L0->Insert(uKPath,Node,true);
//取得第一條最短路徑 並置入 L0 中 結束
uKPath.clear();
// 開始計算 k 條路徑
for ( _uKth=2 ; _uKth <= _uNomOfK ; _uKth++)
{
    KofY=_uKth-1; // 目前求第 _uKth 條路徑, KofY 為前一條路徑在程式中在 KPath 中的位置

    for ( _uNumOfKthPath = 1 ; _uNumOfKthPath < L0->_KPath[KofY-1]._uKPath.size() ;
        _uNumOfKthPath++)
    {
        IofY=_uNumOfKthPath-1; // 第 i 個 node 在 _uKPath 的位置編號
        //對路段成本進行改變開始
        for ( _uPreKthPath = 1 ; _uPreKthPath < _uKth ; _uPreKthPath++)
        {
            JofY=_uPreKthPath-1; // 第 j 條路徑在 L0 的位置編號
            m = L0->GetPathNode(KofY-1,IofY);
            if( IofY != L0->_KPath[KofY-1]._uKPath.size() )
            {
                n = L0->GetPathNode(JofY,IofY+1);
                if(n >= 0)
                {
                    LinkStatus[m][n]='c';
                }
            }
        }

        L2->Insert(GetRootPath(KofY-1,IofY));
        USI KOrigin; // 計算 KSP 中 變動的起點
        KOrigin = L0->GetPathNode(KofY-1,IofY);
        // 讓路徑不往回走 Start////////////////////
        USI noreturn;
        noreturn = L2->_KPath[0]._uKPath.size() -1;
        m = L2->_KPath[0]._uKPath[noreturn];
        for (USI ii=noreturn-1 ; ii<-1 ; ii--)
        {
            n = L2->_KPath[0]._uKPath[ii];
            LinkStatus[m][n] = 'c';
            m = L2->_KPath[0]._uKPath[ii];
        }
    }
}

```

```

    }
    uKPath = CLSSP::GenSP(cpNodeAdjList, KOrigin, uDes); // 計算最短路徑
    // 恢復被改變的路段成本狀態
    for ( _uPreKthPath = 1 ; _uPreKthPath < _uKth ; _uPreKthPath++ )
    {
        JofY=_uPreKthPath-1;
        m = L0->GetPathNode(KofY-1,JofY);
        if( JofY != L0->_KPath[KofY-1]._uKPath.size() )
        {
            n = L0->GetPathNode(JofY,JofY+1);
            if(n >= 0)
            {
                LinkStatus[m][n]='o';
            }
        }
    }
    noreturn = L2->_KPath[0]._uKPath.size() -1;
    m = L2->_KPath[0]._uKPath[noreturn];
    for (USI i=noreturn-1 ; i<-1 ; i--)
    {
        n = L2->_KPath[0]._uKPath[i];
        LinkStatus[m][n] = 'o';
        m = L2->_KPath[0]._uKPath[i];
    }
    ////////////
    if(uKPath.size()!=0) // 檢查是否有產生 uKPath
    {
        L2->Insert(uKPath,Node,false);
        L1->Insert(L2->Union());
        L2=new ListN();
    }
    else
    {
        L2=new ListN();
    }
}
L0->Insert(L1->GetMinPath());
}
//////////
// 螢幕顯示 KSP
cout << "Those K Shorest Path are ( K = " << _uNomOfK << " ):" << endl;
for(USI z=0 ; z<L0->_KPath.size() ; z++)
{
    double cost;
    cout << "No." << z+1 << " : " << "node unumber = " << L0->_KPath[z]._uKPath.size();
    for(USI zz=0 ; zz<L0->_KPath[z]._uKPath.size() ; zz++)
    {
        if( zz%10==0 )
        {
            cout << endl << " ";
        }
        cout << L0->_KPath[z]._uKPath[zz] << " ";
        cost = L0->_KPath[z].Length[zz];
    }
    cout << endl << " the total cost = " << cost << endl;
}

// 輸出 K 條最短路徑 為檔案
fstream KSPFileOut;
KSPFileOut.open("../DTA//SP//PathDataBase//KSP-控制延滯.txt", ios::out|ios::app);

KSPFileOut << "Those K Shorest Path from 'Node "<< uOrigin << " to " << "Node " << uDes << "' ( K = "

```



```

<< _uNomOfK << "):" << endl;
for( z=0 ; z<L0->_KPath.size() ; z++)
{
    double cost;
    KSPFileOut << "No." << z+1 << " : ";
    for(USI zz=0 ; zz<L0->_KPath[z]._uKPath.size() ; zz++)
    {
        if( zz%10==0 )
        {
            KSPFileOut << endl << " ";
        }
        KSPFileOut << L0->_KPath[z]._uKPath[zz] << " ";
        cost = L0->_KPath[z].Length[zz];
    }
    KSPFileOut << endl << " the total cost = " << cost << endl;
}
KSPFileOut << endl;

vector<USI> upath;
upath = L0->_KPath[0]._uKPath;

delete [] Node;
return upath;
}

```



```

/*
** Copyright (c) Feng Chia University. 2003-2004. All Rights Reserved.
*/
#include "TDSP.h"
#include "../CombineNetwork.h"
// Construction/Destruction
extern CombineNetwork *GetData;
CTDSP::CTDSP()
{
}
CTDSP::~~CTDSP()
{
}

NodeList CTDSP::GenSP(CNodeAdjList* cpNodeAdjList, USI uOrgin, USI uDes)
{
    _uSPNumOfNode = 1 + cpNodeAdjList->GetMAXNodeID();
    Node = new SPNode [_uSPNumOfNode];
    vector<USI> uPath;
    vector<USI> SEL;
    double StartTime = 6000; // 起始時間 100 分鐘 * 60 單位 "秒"
    TotalStruct TotalData;
    // 演算法初始化 開始
    USI i;
    for ( i=0; i<= _uSPNumOfNode-1 ; i++)
    {
        Node[i].Length = 99999;
        Node[i].PreNode = 99999;
        Node[i].Status = 'n';
    }
    Node[uOrgin].Length = StartTime;
    Node[uOrgin].PreNode = uOrgin;
    SEL.push_back(uOrgin);
    // TDSP 開始
    while ( SEL.empty() == false )
    {
        _uCurrentNode = SEL.front();
        SEL.erase(SEL.begin());
        NodeList uNodes = cpNodeAdjList->GetLinkedNodes(_uCurrentNode); // 將找出與 CurrentNode
        的所有連接點
        while ( uNodes.empty() == false )
        {
            _uConnectNode = uNodes.front();
            uNodes.erase( uNodes.begin() );
            CLink* cpLink = cpNodeAdjList->GetLink( _uCurrentNode , _uConnectNode);
            _Tm = Node[ _uCurrentNode ].Length; // _Tm 起始值，避免拿掉路口延滯產生的影響
            _Tk = Node[ _uCurrentNode ].Length / 6; //時間"秒"轉成時間"區間"
            if(_Tk > 1998)
            {
                _Tk = 1998;
            }
            _IntersectionDelay = 0;
            // 路口延滯
            // 控制延滯
            // IntersectionDelay = cpLink->GetInterSignalDelay( _uConnectNode , _uCurrentNode);
            // 平均紓解率延滯
            _IntersectionDelay = cpLink->GetInterAvgDischarRate( _Tk, _uCurrentNode ,
            _uConnectNode);
            _Tm = Node[ _uCurrentNode ].Length + _IntersectionDelay;
        }
    }
}

```

```

TotalData = GetData->GetCBNDData( _Tk, _uCurrentNode, _uConnectNode);
_Speed = TotalData.Vel;
if ( Node[ _uConnectNode ].Length > _Tm + cpLink->GetMaralCst(_Speed) )
{
    Node[ _uConnectNode ].Length = _Tm + cpLink->GetMaralCst(_Speed);
    Node[ _uConnectNode ].PreNode = _uCurrentNode;
    USI* selcheck;
    selcheck = find(SEL.begin() , SEL.end() , _uConnectNode);
    if(selcheck == SEL.end())
    {
        SEL.push_back(_uConnectNode);
    }
}
}
for ( i=0; i<= _uSPNumOfNode-1 ; i++)
{
    Node[i].Status = 'I';
}
////////// 建立從 uOrigin 開始的路徑資料庫 //////////
PathList Path;
USI desnode;
_PathList.push_back(Path); // 第 0 個位置置入空的路徑(方便取路徑)
for( USI z=0 ; z<DesNode.size() ; z++)
{
    desnode = DesNode[z];
    Path._Path.push_back(desnode);
    for( i=desnode ; i !=uOrigin ; )
    {
        Path._Path.insert( Path._Path.begin(), Node[i].PreNode);
        i = Node[i].PreNode;
    }
    _PathList.push_back(Path);
    Path._Path.clear();
}
////////// 輸出為檔案 //////////
USI counter=1;
fstream SPFileOut;
fstream SPcostFileOut;
SPFileOut.open("../DTA//SP//PathDataBase//TDSP-path-.txt", ios::out|ios::app);
SPcostFileOut.open("../DTA//SP//PathDataBase//TDSP-cost-.txt", ios::out|ios::app);
SPFileOut << "Those shortest pathes from 'Node " << uOrigin << "': " << endl;
for( i=1 ; i<_PathList.size() ; i++)
{
    SPFileOut << "to 'Node" << _PathList[i]._Path[_PathList[i]._Path.size()-1] << "': " << endl << " ";
    for( z=0 ; z<_PathList[i]._Path.size() ; z++)
    {
        SPFileOut << _PathList[i]._Path.at(z) << " ";
        if(counter%10==0)
        {
            SPFileOut << endl << " ";
        }
        counter++;
    }
    SPFileOut << endl;
    SPFileOut << " The total travel time : ";
    SPcostFileOut << Node[_PathList[i]._Path.at(z-1)].Length-StartTime << " seconds " << endl;
    SPFileOut << endl;
    counter = 1;
}
SPFileOut << endl;

```

```
////////////////////////////////////
//////////////////////////////////// Show Path //////////////////////////////////////
cout << endl << "The shortest is the Time-Dependent Shortest Path.";
cout << endl << "The path of " << uOrigin << " to " << uDes << " is : ";
for( i=0; i<uPath.size()-1 ; i++)
{
    cout << uPath.at(i) << " -> ";
}
cout << uDes << endl;
cout << "Total nodes are : " << uPath.size() << endl;
cout << "The total travel time of this shortest path = " << Node[uDes].Length-StartTime << endl;
////////////////////////////////////
delete [] Node;
return uPath;
}
```

