國立嘉義大學/嘉義高中科學班

(專題研究、個別科學研究)成果報告

遺傳演算法在旅行推銷員問題上的應用

學年度：108 學年度

指導教授：生化科技學系翁秉霖老師

學生姓名：吳柏橙

評分
指導教授 簽名：

中華民國一〇九年五月二十五日

目錄

壹、摘要

旅行推銷員問題又稱 TSP（traveling salesman problem）問題通常會給出城市數與城市之間的距離（花費），需要求出一種可以從起點城市開始經過所有城市「一次」後回到起點的最短花費走法。可以應用在企劃、物流、晶片製造、天文學……等方面。

本次研究試圖利用暴力窮舉法、動態規劃、遺傳演算法由低效率到高效率的探討解決旅行推銷員問題的方式。並探討各方法的差異。

貳、前言

如何定義一個方法是否有效率，常用的方法是計算出方法的時間複雜度和空間複雜度，並且比較 Big-O。例子：要讓圓周上的 n 個點任意兩點都有連線需要連$(n^2-n)/2$ 條，Big-O 會記作 $O(n^2)$

這種計算一個方法是否有效率，是因為在 n 極大的情況下，常數與其他項的影響都不大，所以能夠粗略且快速的表示一個算法的優劣。（但部份時候還是會考慮常數，但還是以 Big-O 優先）

此外，在本次研究中，城市將被視為節點，而距離、花費被視為兩點之間邊的權值，這些定義將把問題抽象化，以方便在研究中討論。

參、研究目的

一、介紹 TSP 與應用

二、利用多種方式求解 TSP 並比較各方法的差異

三、統整多種方法，找到最適合解決 TSP 的方法

肆、實驗器材實驗過程及方法

一、介紹旅行推銷員問題及其應用

(一)定義問題

旅行推銷員問題又稱 TSP（traveling salesman problem）問題通常會給出城市數與城市之間的距離（花費），需要求出一種可以從起點城市開始經過所有城市「一次」後回到起點的最短花費走法。可以應用在企劃、物流、晶片製造、天文學.....等方面。

（二）介紹應用

純粹形式的 TSP 的應用就是處理物流的問題。若能求解出 TSP，那送信、送貨……等現實中出現的問題，都能有效率的被解決，以減少成本。TSP 也能應用在晶片製造，只要改變距離矩陣就能將晶片製造的問題轉化成 TSP。解決 TSP 將能有效的降低晶片製造的成本。除此之外， TSP 也能應用在天文學上。當天文學家要觀察多顆星體時，需要轉動望遠鏡，此時可以將從一顆星星轉到另一顆的時間視為 TSP 中的邊權重，而每個星星都是一節點，如此一來，求解 TSP 也就能得出轉動望遠鏡的最低成本。

二、測試資料驗證作法正確性

（一）測試資料的形式

第一行生成一數 n 代表節點數，接下來 n 行每行有 n 個數字，第 i 行第 j 列數字代表城市 i 到城市 j 的花費（單向）

（二）生成測試資料的方法

1. 人工生成

人工生成數個隨機數據，並人工窮舉出最佳解，詳細資料在附錄

2. 電腦生成

利用以當下時間為種子的梅森旋轉，製造不大於 10000 的隨機非負整數，製造 n 排 n 列個隨機數（n 為城市數），其中第 a 列、第 b 行的數代表城市 a 到 b 的花費。

（三）TSPLIB

TSPLIB 有各式各樣的 TSP 測試例題，例題的最佳解不僅有經過其他啟發式演算法的驗證，也有利用數學方法確認，其準確性相當高。有許多論文在討論 TSP 時，都會採用 TSPLIB 作驗證。

（四）如何驗證

從程式碼最好寫、最不容易出現 bug 的暴力窮舉法開始寫，並且利用人工生成的測試資料驗證暴力窮舉法的正確性；確認暴力窮舉法無誤後，用暴力窮舉法驗證動態規劃法的正確性；最後用動態規劃法驗用遺傳演算法的正確性，並且用

TSPLIB 測試遺傳演算法的精確度。

三、暴力窮舉法（Brute Force）

（一）簡介暴力窮舉法

窮舉所有可能的路徑，並計算出走所有路徑的花費，找到最小的花費，並將其紀錄。

由於暴力窮舉法需要找出所有路徑，因此此作法的時間複雜度極大，但僅須要紀錄最佳的一條路徑，因此空間複雜度不大。

時間複雜度：O(|V|!)

空間複雜度：O(|E|)

有鑑於暴力窮舉法的時間複雜度極大，因此，暴力窮舉法最多只能用來解決節點數不大於 15 的情況，若超過 15，所花費的計算時間將會成長到非常大，以至於無法在數分鐘內完成計算。

（二）優化暴力窮舉法

在窮舉所有路徑中，會發現有些路徑顯然不是最佳解，例如：你發現你尚未走過所有路徑，而所需要的花費就已經超過當前找到的最佳解，那顯然那條路不會是最佳路徑。

但此優化仍然無法改變暴力窮舉法的時間複雜度，僅能作一點常數上的優化。

四、動態規劃法（Dynamic Programming）

（一）簡介動態規劃法

此作法的核心思想為儲存已經解決的子問題，以減少重複計算，並利用那些子問題，求出最終解答。

由於需要儲存相當多的子問題，因此需要花費極為龐大的空間，不過也因此，時間複雜度下降了

時間複雜度：$O(n^2*2^n)$

空間複雜度：$O(n*2^n)$

（二）定義與作法簡介

把所有城市的狀態以一個二進位的數表示（每個點的狀態區

分為 0 跟 1，第 i 個點在第 i 位）

定義：S 為走過的城市集合，而 u,v 是三個相異的城市，且滿足 v 屬於 S 和 u 不屬於 S。dp(S,v)是以 v 為最後經過的城市，且經過集合 S 所有城市所需的最小花費。dis(u,v)表示從 u 前往 v 的路徑花費。對於任意 u 皆滿足

dp(S,u)=min(dis(u,v)+dp(S 聯集 u,u))

利用 bottom-up 或 top-down 的作法就能得出 dp(S’,i)的結果，再窮舉 i，就能找到 dp(S’,i)+dis(i,起點)的最小值了，因為 TSP 的路徑是一個環形，因此起點的選擇不影響運算結果

（S’ 表所有城市的集合，i 和起點 皆為任意城市）

五、遺傳演算法（Genetic Algorithm）

（一）遺傳演算法簡介

遺傳演算法又稱基因演算法，其核心思想是利用達爾文提出的「物競天擇、適者生存」，將可能的答案一步步的逼近問題最佳解。

遺傳演算法將欲求解的問題變數或參數以一種類似染色體的

資料結構(Chromosome-Like Data Structure)來編碼，並應用一些遺傳運算元(Operators)如交換(Crossover)、突變(Mutation)對大量的染色體作運算，運算後產生的子代除了保存親代中具優勢的特質外，也有可能因為基因的交換與突變而比親代的表現更佳。然後經由自然選擇的過程，保留適應程度高的個體，並淘汰適應程度低的。經過數代演化後，找出適應程度最高的個體，作為遺傳演算法的計算結果。

（二）實作簡介

本次研究想要解決的問題是旅行推銷員問題（TSP）。首先必須選出適合的變數作為染色體。在旅行推銷員問題中路徑的選擇就是一個非常適合的變數。例如：路徑 1->2->3 就可以編碼成數列 1,2,3，其中數列 1,2，也就是路徑 1->2 就可以作為生物中的基因，而數列 1,2,3 這條完整的路徑字串就相當於生物學中的染色體。包含路徑與所需花費的，我稱之為一個獨立的個體

1. 初始狀態：

利用完全隨機的方式，生成多個個體，作為初始狀態。

2. 交配：

利用輪盤法選擇交配的對象，並利用 Partially Mapped Crossover Operator 交換染色體跟基因。並且將交配的機率設為 80%

3. 突變：

交換染色體中兩點的位置，以產生基因形式，幫助跳脫區域最佳解，往全局最佳解前進。本研究中，我將突變的機率設為 1%

4. 汰弱留強：

首先，定義適應度為所有邊的花費總和減去此路徑的花費。再者，本研究利用精英策略，只留下前面數個適應性好的個體。最後，經過數代演化，我們將得到一個適應度最高，也就是花費最小的個體。

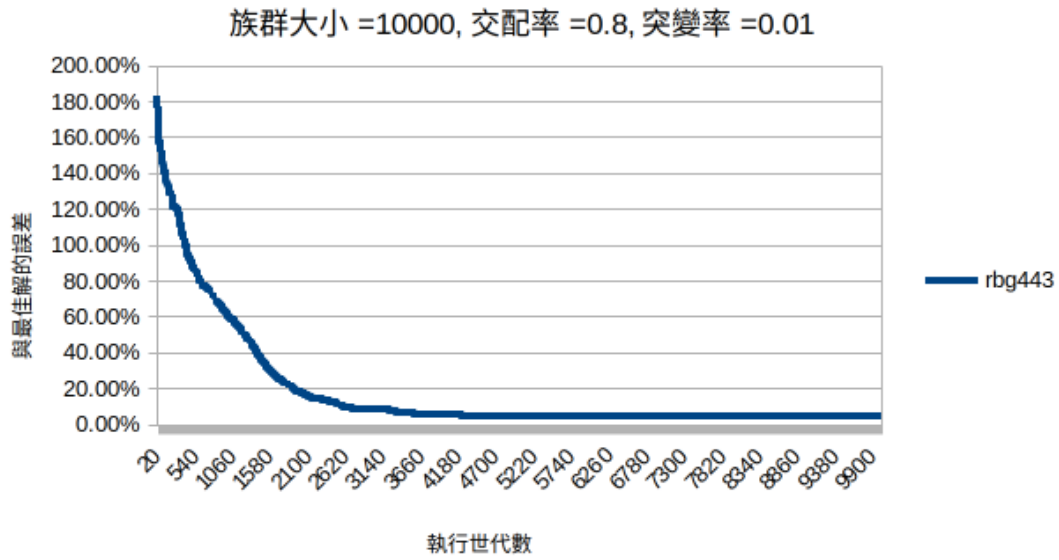
伍、研究結果

本研究先利用自己生成的測試資料驗證我實作的遺傳演算法沒有大問題，再利用 TSPLIB 進一步的驗證在城市數多的時候，能否有良好的精確度。

測試資料名稱	測試資料的城市數	初始的誤差	執行完成後的誤差
br17	17	87.18%	0.00%
bays29	29	117.53%	0.00%
ft53	53	205.78%	2.68%
ft70	70	70.26%	2.79%
ftv33	34	115.21%	4.12%
ftv35	36	150.92%	0.95%
ftv38	39	160.85%	3.40%
ftv44	45	202.73%	0.00%
ftv55	56	276.79%	5.16%
ftv64	65	286.79%	6.74%
ftv70	71	306.10%	8.21%
ftv170	171	753.25%	31.62%
kro124p	100	333.22%	4.05%
p43	43	204.18%	1.90%
rbg323	323	329.19%	16.52%
rbg358	358	442.82%	22.96%
rbg403	403	188.97%	6.41%

rbg443	443	183.26%	4.30%
ry48p	48	169.87%	4.63%
swiss42	42	187.82%	2.91%

表一、TSPLIB 的測試結果



圖一、TSPLIB 的 rbg443 測試結果（演化示意圖）

結果顯示，在城市數量不大於 100 時，都能獲得誤差在 10%以下的結果，即使城市數量大於 100 仍能有約 30%誤差的表現，代表此遺傳演算法確實能解決旅行銷售員問題（TSP）

陸、討論

一、使用完全隨機的方式生產起始個體？

使用完全隨機的方式生產起始個體非常快速，也可以避免起始個體有特定偏差，導致遺傳演算法收斂到局部最佳解（local optimum）而非全局最佳解，但收斂速度會較慢。

二、交配選擇中的輪盤法是什麼？

將族群中所有染色體的適應度加總後表示成輪盤的總面積，每個染色體分配在輪盤上的面積依照適應度高低做調整，接著隨機產生一個亂數值，就像射飛鏢一樣射向輪盤，選中的染色體就進行到下一階段的交配，此方法可使適應度較佳的染色體有較大的機率被選取，但因為用隨機的方式挑選，適應值較差的染色體也有機會被選取。

三、城市數量大於 100，為何無法維持 10%的誤差？

當城市數量增加時，初始族群僅 10000 大小，不一定會包含到會讓花費最小化的路徑，僅靠著突變也難以讓將誤差壓到 10%以下。

四、如何選擇參數？

1. 族群大小：

由 Analysis of the impact of parameters values on the Genetic Algorithm for TSP 這篇論文得知，族群大小越大，將有助於降低誤差，使得收斂結果接近最佳解，但族群大小越大同時代表需要耗費更多的記憶體與執行時間，因此我選擇沿用該論文中的數據，將族群大小設定為 10000。

2. 突變率：

由 Analysis of the impact of parameters values on the Genetic Algorithm for TSP 這篇論文得知，突變率的大小對收斂的結果沒有顯著影響，因此採用該論文中使用的 0.01。而且突變率較小，產生突變運算少，運作速度會更快，但太小將無法脫離局部最佳解。(也符合 nasa 文件中對遺傳演算法的突變率的建議，小於 0.05)

3. 交配率：

Analysis of the impact of parameters values on the Genetic Algorithm for TSP 裡沒有提到交配率的選擇，因此我沿用 Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator 裡使用

的 0.8。論文裡不只 Cycle Crossover Operator，裡面還有 Partially Mapped Crossover Operator，也就是本次研究使用的 Crossover Operator.（也符合 nasa 文件中對遺傳演算法的交配率的建議，在 0.5 到 1 之間）

柒、結論

雖然遺傳演算法能解決多數情況的 TSP，但在處理城市數小於 22 的 TSP 時，動態規劃法（DP）會是一個更快速，且保證是最佳解的選擇。結合動態規劃與遺傳演算法兩者，將能有效率且準確的解出 TSP，並用於實際生活，改進人類生活。

捌、參考文獻

一、 Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator（Abid Hussain,Yousaf Shad Muhammad,M.Nauman Sajid,Ijaz Hussain,Alaa Mohamd Shoukry, and Showkat Gan）

二、 Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems（ Sanjeev Arora ）

三、NASA 和遺傳演算法有關的文件

https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19910014737.pdf

四、TSPLIB

1. http://vrp.atd-lab.inf.puc-rio.br/attachments/article/6/TSPLIB%2095.pdf

2. http://www.math.uwaterloo.ca/tsp/data/index.html

五、 Analysis of the impact of parameters values on the Genetic Algorithm for TSP (Avni Rexhepi, Adnan Maxhuni, Agni Dika)

玖、附錄

一、暴力窮舉法

(一)程式碼與詳細實作步驟

```
//預設定 start
#include<bits/stdc++.h>
```

```
#define ll long long
#define MAXN 50

using namespace std;
//預設定 end


//宣告變數
ll n,ans=(ll)1<<61;     //儲存 節點數 當前最佳解
bool visit[MAXN];       //紀錄節點是否走過
vector<int> route,temporaryRoute(MAXN);         //儲存最佳解路徑 儲存當前遞迴路徑（初始化大小為 MAXN，意即節點數的上限）
vector<int> E[MAXN];    //存邊 index 代表起始位置

void dfs(int now,ll COST,int node,int start)  //遞迴路徑的函數，有分個變數分別代表 現在位置,當前花費,剩下幾個節點沒走過,起點
{
        if(COST>ans)    //若當前結果已大於之前求得的最小值則該路徑不可能是最佳路徑，所以跳出此路徑
                return;
        if(node==0){    //若已走過每個城市一次，執行以下

                COST+=E[now][start-1];

                if(COST<ans){   //找到路徑，比較當前路徑的花費與當前找到的最小值
                        ans=COST;
                        route.clear(); //清空紀錄器
                        for(int i=0;i<n;++i)    //紀錄下最佳路徑
                                route.push_back(temporaryRoute[i]);
                }
                return; //跳出該層遞迴
        }

        int next;       //存放下一個節點
        for(int i=0;i<n;++i){   //窮舉下一個點
                next=i+1;
                if(!visit[next]){       //確定是沒走過的點
```

```
                        temporaryRoute[node-1]=next;   //紀錄遞迴路徑
                        visit[next]=true;       //標記 next 已經走到了
                        dfs(next,COST+E[now][i],node-1,start);//進入下
一層遞迴
                        visit[next]=false;      //因為離開遞迴了，所以
可以變回原本的狀態
                }
        }
}

void init()
{
        ans=(ll)1<<61;
        route.clear();
        temporaryRoute.clear();
        for(int i=1;i<=n;++i)
                E[i].clear();
}

void input()   //輸入資料
{
        int temporaryCOST;
        for(int from=1;from<=n;++from){        //輸入邊的資料
                for(int j=1;j<=n;++j){
                        cin >> temporaryCOST;
                        E[from].push_back(temporaryCOST);
                }
        }
}

void solve()   //calculate the answer
{
        int start=1;
        temporaryRoute[n-1]=start;     //紀錄起點
        visit[start]=true;     //標記起點已走過
        dfs(start,0,n-1,start);        //計算以 start 為起點的最短迴路
        if(ans!=((ll)1<<61)){          //判斷路徑是否存在
                cout << "cost: " << ans << endl;       //輸出答案
```

```
                cout << "route: ";
                for(int i=n-1;i>=0;--i)        //輸出最佳解路徑（若有
多條僅輸出其中一條）
                        cout << route[i] << "->";
                cout << route[n-1] << endl;
        }
        else
                cout << "cost: 0\nroute doesn't exist" << endl;

}

int main()
{
        FILE *fPtr=freopen("input_RAW.txt","r",stdin);        //開啟要
讀取的檔案

        while(cin >> n){//輸入 n
                init();         //初始化
                input();        //input data
                solve();        //output data
        }

        fclose(fPtr);  //關閉讀取的檔案

        return 0;
}
```

二、動態規劃法

(一)程式碼與詳細實作步驟

```
#include <bits/stdc++.h>
#define N 22
#define ll long long

using namespace std;
```

```
int n;
vector<int> G[N];
int dp[N][(ll)1<<N],NEXTPOS[N];        //dp[若為 i 表示目前在 i+1 點][走過的路徑，二進位下第 i 位為 1 表示第 i+1 點有走過，0 表示未走過]

void init()     //初始化，清除上次的運算結果
{
        for(int i=0;i<n;++i){
                G[i].clear();
                NEXTPOS[i]=i;
        }
}

void input()    //輸入資料
{
        int temporaryCOST;
        for(int i=0;i<n;++i){
                for(int j=0;j<n;++j){
                        cin >> temporaryCOST;
                        G[i].push_back(temporaryCOST); //將花費存成邊的權重
                }
        }
}

void solve()
{
        for(ll path=1;path<((ll)1<<(n));++path){       //窮舉經歷過的路徑
                for(int next=0;next<n;++next){                  //窮舉下一個要走的點
                        if( ((ll)1<<next)&path )    continue; //確定 next 不是走過的點
                        for(int FROM=0;FROM<n;++FROM) {         //窮舉經歷過的點裡，是由那一個連向 next
                                if ( ((ll) 1 << FROM) & path && (dp[next][path + ((ll) 1 << next)]>( dp[FROM][path] + G[FROM][next])) ) {     //確定 FROM 真的走過（在 path 裡），且若由 FROM
```

```
連到 next 形成的路徑圖會是比之前的好，就更新 最小權重和 與 連線方式 (NESTPOS[])
                                        NEXTPOS[FROM]=next;     //紀錄 FROM 連向 next 會是更佳的連線方式
                                        dp[next][path | ((ll) 1 << next)]= dp[FROM][path] + G[FROM][next];        //紀錄下此連線方式的權重和
                                }
                        }
                }
        }
}

void output()
{
        int ans=1000000000,bestStart,start=0;

        memset(dp,0x3f,sizeof(dp));
        dp[start][(ll)1<<start]=0;
        solve();

        for(int FROM=0;FROM<n;++FROM) //窮舉由那一個點連回起始點會有最佳結果
                        ans=min(ans,dp[FROM][((ll)1<<(n))-1]+G[FROM][start]);

        cout << "cost: " << ans << endl;       //輸出值
}

int main()
{
        FILE *fPtr=freopen("input_RAW.txt","r",stdin);        //開啟輸入資料的檔案

        while(cin >> n){
                init();
                input();
                output();
```

```
        }

        fclose(fPtr);  //關閉輸入資料的檔案

        return 0;
}
```

三、遺傳演算法

(一)程式碼與詳細實作步驟

```
#include<bits/stdc++.h>
#define ll long long
#define N 100000

using namespace std;
//呼叫執行程式所需的相關檔案
//設定隨機數生成器 start
random_device rd;
auto tt = chrono::high_resolution_clock::now();
std::mt19937_64 gen = std::mt19937_64(tt.time_since_epoch().count());
std::uniform_real_distribution<> dis(0 , 1);
auto randGenerator = bind(dis , gen);
//設定隨機數生成器 end

int n;  //城市數
const int population=10000;    //族群大小
const long double
crossOverRate=0.8,mutationRate=0.01,generationMax=10000;      //交配率
為 0.8,突變率為 0.01,最大世代數為 10000

vector< vector<int> > TSP_order;        //儲存所有個體的基因
vector<long double> G[N];               //儲存路徑圖
vector<long double> TSPdistance;        //儲存每個個體的花費
long double TSPdistanceMAX=0;  //儲存所有路徑花費的總和

auto randShuffle(int x)         //製造一個小於 x 的非負亂數
```

```
{
    return (int)(randGenerator()*1e8)%x;
}

bool cmp(int a,int b)  //排序的比較函數
{
    if(TSPdistance[a]<TSPdistance[b]) //若
TSPdistance[a]<TSPdistance[b]則讓 a 排在 b 前面
        return true;
    else
        return false;
}

void init()    //初始化所有變數
{
    TSP_order.clear(); //確保一開始沒有任何個體
    TSPdistance.clear();       //清空 TSPdistance

    int tmp;
    for(int i=0;i<n;++i){      //輸入表示路徑的矩陣，並儲存
        G[i].clear();  //在使用前先清空
        for(int j=0;j<n;++j){
            cin >> tmp;
            G[i].push_back(tmp);       //放入儲存資料
            TSPdistanceMAX+=tmp;       //紀錄所有路徑的花費
        }
    }
}

void randomlyGenerated()       //隨機生成個體
{
    vector<int> ordinaryOrder; //暫存個體的基因
    for(int i=0;i<n;++i)
        ordinaryOrder.push_back(i);    //初始化為 1,2,3...n 基因的個體

    for(int i=0;i<population;++i){
        TSP_order.push_back(ordinaryOrder);   //放入族群中
```

```
random_shuffle(TSP_order[i].begin(),TSP_order[i].end(),randShuffle);
        //打亂剛剛放入族群中的個體的基因（變成隨機的基因）

        long double tmpDistance=G[TSP_order[i][n-
1]][TSP_order[i][0]];  //暫存剛剛個體的花費
        for(int j=1;j<n;++j)
            tmpDistance+=G[TSP_order[i][j-1]][TSP_order[i][j]];
        //計算剛剛個體的花費
        TSPdistance.push_back(tmpDistance);           //將計算結果存
起來
    }
}

void crossOver(int father,int mother) //交換（採用Partially Mapped
Crossover Operator）
{
        /*
        example:
                P1 = (3 4 8 2 7 1 6 5)
                P2 = (4 2 5 1 6 8 3 7)
        */

    int crossPoint1=randShuffle(n),crossPoint2=randShuffle(n);
        //隨機選取基因上兩點
    if(crossPoint2<crossPoint1)               //確保選取兩點
crossPoint1>crossPoint2
        swap(crossPoint1,crossPoint2);

        /*
           以 | 代表基因的切割點
        example:
                P1 = (3 4 8 2 7 1 6 5)
                P2 = (4 2 5 | 1 6 8 | 3 7)
        */

    vector<int> tmp;   //暫存下一代的基因
    set<int> needToRemove;
```

```
    for(int i=crossPoint1;i<=crossPoint2;++i) //將 mother 的第
crossPoint1 個到第 crossPoint2 個基因選出來
        needToRemove.insert(TSP_order[mother][i]);
        /*
           建構後代基因型
           C = (× × × | 1 6 8 | × ×)
           P1 = (3 4 x 2 7 x x 5)
        */
    int COUNT=0;
    for(int i=0;i<n;++i){
        if(COUNT==crossPoint1){
            for(int j=crossPoint1;j<=crossPoint2;++j)
                tmp.push_back(TSP_order[mother][j]);
            ++COUNT;
        }

if(needToRemove.find(TSP_order[father][i])==needToRemove.end()){
            tmp.push_back(TSP_order[father][i]);
            ++COUNT;
        }
    }

        /*
           依序填入 P1 的基因到 C 裡面
           C = (3 4 2 | 1 6 8 | 7 5)
           將 C 放入族群中
        */

    if(COUNT==crossPoint1){
        for(int j=crossPoint1;j<=crossPoint2;++j)
            tmp.push_back(TSP_order[mother][j]);
    }
    TSP_order.push_back(tmp);
}

void mutation(int num) //突變
{
        /*
```

```
            P = (1 3 5 4 6 7 8)
        */
    vector<int> tmp;   //暫存突變後的基因

    int a=randShuffle(n),b=randShuffle(n);    //隨機兩個城市（核苷
酸）
    while(b==a){
        b=randShuffle(n);
    }

        /*
            P = (1 3 5 4 6 7 8)
            a=3
            b=7
        */


    for(int i=0;i<n;++i){      //尋找兩城市（核苷酸）的位置並交換
        if(TSP_order[num][i]==a)
            tmp.push_back(b);
        else if(TSP_order[num][i]==b)
            tmp.push_back(a);
        else
            tmp.push_back(TSP_order[num][i]);
    }
        /*
            突變後的個體
            Q =(1 7 5 4 6 3 8)
        */
    TSP_order.push_back(tmp);
}

int main()
{
    //freopen("input_RAW.txt","r",stdin);
        cout << fixed << setprecision(6);     //設定輸出精度到小數點
下六位
    while(cin >> n){   //輸入城市數
```

```
        init();         //執行初始化
        randomlyGenerated();   //生成初始個體

        long double distanceSum[population],MIN0=1e18;

                for(int i=0;i<population;++i)          //紀錄隨機生成
的初始個體的最小值
                        MIN0=min(MIN0,TSPdistance[i]);
                cout << MIN0 << endl;

        ll t=0,num;
        for(int generation=0;generation<generationMax;++generation){
        //重複數代
            memset(distanceSum,0,sizeof(distanceSum));        //清空機
率輪盤
            for(int i=1;i<population;++i)
                distanceSum[i]=TSPdistanceMAX-TSPdistance[i-
1]+distanceSum[i-1];   //計算每個個體的適應值，適應值=所有路徑花費-個
體花費
            for(int i=0;i<population;++i)              //設定機率輪盤
                distanceSum[i]/=distanceSum[population-1];

            for(int i=0;i<population;++i){
                if(randGenerator()<=crossOverRate){           //對於每
一個個體若可以交換（交換率決定）
                    long double Probability=randGenerator(); //透過機
率輪盤決定交換對象
                    int
pos=distance(distanceSum,lower_bound(distanceSum,distanceSum+populati
on,Probability));
                    crossOver(i,pos);
                }
            }

            for(int i=0;i<population;++i)     //對每一個個體（不含剛
交換後的下一代）決定是否突變（由突變率決定）
                if(randGenerator()<=mutationRate)
```

```
                mutation(i);

            for(int i=population;i<TSP_order.size();++i){
        //計算新產生的個體的花費（突變與交換）
                ll tmpDistance=G[TSP_order[i][n-1]][TSP_order[i][0]];
                for(int j=1;j<TSP_order[i].size();++j)
                    tmpDistance+=G[TSP_order[i][j-
1]][TSP_order[i][j]];
                TSPdistance.push_back(tmpDistance);
            }

            vector<int> index;          //自然淘汰：將花費前 population
小的個體保留，其他淘汰------start
            for(int i=0;i<TSP_order.size();++i)
                index.push_back(i);

            sort(index.begin(),index.end(),cmp);
            index.erase(index.begin(),index.begin()+population);
            sort(index.begin(),index.end());

            for(int i=0;i<index.size();++i){
                TSP_order.erase(TSP_order.begin()+index[i]-i);
                TSPdistance.erase(TSPdistance.begin()+index[i]-i);
            }           //自然淘汰------end

            long double min0=1e18;              //紀錄所有個體的花費最
小值
            for(int i=0;i<population;++i){
                if(min0>TSPdistance[i]){
                    min0=TSPdistance[i];
                    num=i;
                }
            }
                    MIN0=min(MIN0,min0);
                    cout << MIN0 << endl;  //輸出每代的最小值
        }
        cout << "cost: " << MIN0 << endl;     //輸出運算最終的最小花
費
```

```
            /*              //除錯用，輸出路徑
        for(int i=0;i<n;++i)
            cout << TSP_order[num][i]+1 << "->";
        cout << TSP_order[num][0]+1 << endl;
            */
    }
    return 0;
}
```

四、電腦生成測資

(一)程式碼與詳細實作步驟

```
//預設定 start
#include<bits/stdc++.h>
#include <random>
#include <time.h>
#include <ratio>
#include <chrono>
#include <functional>

using namespace std;
//預設定 end


//宣告變數
int n;

void input()    //輸入資料
{
        cin >> n;       //input the number of cities
        cout << n << endl;
}

void solve()
{
        //設定隨機數生成器 start
```

```
        random_device rd;
    auto tt = chrono::high_resolution_clock::now();
    std::mt19937_64 gen =
std::mt19937_64(tt.time_since_epoch().count());
    std::uniform_int_distribution<> dis(0 , 10000);
    auto randfunction = bind(dis , gen);
        //設定隨機數生成器 end

        for(int i=1;i<=n;++i){          //輸出 n*n 個隨機數
                for(int j=1;j<=n;++j){
                        cout << randfunction() << " ";
                }
                cout << endl;
        }

}

int main()
{

        cout << "input the number of city:";
        FILE *fPtr=freopen("input_RAW.txt","a",stdout);       //開啟要
寫入的檔案

        input();        //input data
        solve();        //output data

        fclose(fPtr);

    return 0;
}
```

五、驗證正確性的程式碼

(一)驗證動態規劃

```
//預設定 start
```

```
#include<bits/stdc++.h>
#include <random>
#include <time.h>
#include <ratio>
#include <chrono>
#include <functional>

using namespace std;
//預設定 end


//宣告變數
int n,t,num,tmp;

void solve()
{
        //設定隨機數生成器 start
        random_device rd;
    auto tt = chrono::high_resolution_clock::now();
    std::mt19937_64 gen =
std::mt19937_64(tt.time_since_epoch().count());
    std::uniform_int_distribution<> dis(0 , 10000);
    auto randfunction = bind(dis , gen);
        //設定隨機數生成器 end

        num=(abs(randfunction())%(n-1))+2;     //隨機一個城市數
        cout << num << endl;            //輸出程式數
        for(int i=1;i<=num;++i){                //輸出 num*num 個隨機數
                for(int j=1;j<=num;++j){
                        tmp=randfunction();
                        if(tmp<=0)      //若權重不大於 0 就當沒有那條邊
                                tmp=0;
                        cout << tmp << " ";
                }
                cout << endl;
        }

}
```

```
int main()
{
	cout << "input the number of test data:";
	cin >> t;	//輸入生成測資數
	cout << "\ninput the maximun of city:";
	cin >> n;

	//生成測資 start
	FILE *fPtr=freopen("input_RAW.txt","w",stdout);	//開啟要
寫入的檔案
	for(int i=0;i<t;++i)	//重複 t 次
		solve();	//output data
	fclose(fPtr);	//關閉測資檔案
	//生成測資 end
	FILE *BruteForce=popen("./BruteForce","r");

	char trash=fgetc(BruteForce);
	int BruteForce_Ans[100000];

	for(int i=0;i<t;++i){
		fscanf(BruteForce,"ost: %d", &BruteForce_Ans[i]);
		do{
			trash=fgetc(BruteForce);
		}while(trash!='c' && trash!=EOF);
	}

	pclose(BruteForce);

	FILE *DP=popen("./DP","r");

	trash=fgetc(DP);
	int DP_Ans[100000];

	for(int i=0;i<t;++i){
		fscanf(DP,"ost: %d", &DP_Ans[i]);
		do{
			trash=fgetc(DP);
```

```
                }while(trash!='c' && trash!=EOF);
        }

        fclose(DP);

        freopen("result.txt","w",stdout);
        bool flag=true;
        int wrongLine;
        for(int i=0;i<t;++i){
                if(DP_Ans[i]!=BruteForce_Ans[i]){
                        flag=false;
                        wrongLine=i;
                        break;
                }
        }
        if(flag)
                cout << "Correct" << endl;
        else
                cout << "Wrong" << endl << "wrong line: " <<
wrongLine <<  endl;

    return 0;
}
```

(二)驗證遺傳演算法

```
//預設定 start
#include<bits/stdc++.h>
#include <random>
#include <time.h>
#include <ratio>
#include <chrono>
#include <functional>

using namespace std;
//預設定 end
```

```
//宣告變數
int n,t,num,tmp;

void solve()
{
        //設定隨機數生成器 start
        random_device rd;
    auto tt = chrono::high_resolution_clock::now();
    std::mt19937_64 gen =
std::mt19937_64(tt.time_since_epoch().count());
    std::uniform_int_distribution<> dis(-100 , 1000); // 約 1/11 的機
率沒有邊
    auto randfunction = bind(dis , gen);
        //設定隨機數生成器 end

        num=(abs(randfunction())%(n-1))+2;     //隨機一個城市數
        cout << num << endl;            //輸出程式數
        for(int i=1;i<=num;++i){                //輸出 num*num 個隨機數
                for(int j=1;j<=num;++j){
                        tmp=randfunction();
                        if(tmp<=0)      //若權重不大於 0 就當沒有那條邊
                                tmp=0;
                        cout << tmp << " ";
                }
                cout << endl;
        }

}

int main()
{
        cout << "input the number of test data:";
        cin >> t;       //輸入生成測資數
        cout << "\ninput the maximun of city:";
        cin >> n;

        //生成測資 start
```

```
        FILE *fPtr=freopen("input_RAW.txt","w",stdout);       //開啟要
寫入的檔案
        for(int i=0;i<t;++i)   //重複 t 次
                solve();       //output data
        fclose(fPtr);  //關閉測資檔案
        //生成測資 end

        FILE *GA=popen("./GA","r");

        char trash=fgetc(GA);
        int GA_Ans[100000];
        double GA_Time[100000];

        for(int i=0;i<t;++i){
                fscanf(GA,"ost: %d\n", &GA_Ans[i]);
                fscanf(GA,"TIME: %lf",&GA_Time[i]);
                do{
                        trash=fgetc(GA);
                }while(trash!='c' && trash!=EOF);
        }

        pclose(GA);

        FILE *DP=popen("./DP","r");

        trash=fgetc(DP);
        int DP_Ans[100000];

        for(int i=0;i<t;++i){
                fscanf(DP,"ost: %d", &DP_Ans[i]);
                do{
                        trash=fgetc(DP);
                }while(trash!='c' && trash!=EOF);
        }

        fclose(DP);

        freopen("result.txt","w",stdout);
```

```
        double
deviation[100000],MAX0=0,MIN0=100000,sum=0,TIME_MAX0=0,TIME_MIN0=1000
00,TIME_sum=0,SD=0;

        for(int i=0;i<t;++i){
                if(DP_Ans[i]==0){
                        if(GA_Ans[i]==0)
                                continue;
                        else{
                                cout << "wrong\n";
                                break;
                        }
                }

                deviation[i]=(GA_Ans[i]-DP_Ans[i])/(double)DP_Ans[i];
                sum+=deviation[i];
                SD+=pow(deviation[i],2)/t;
                TIME_sum+=GA_Time[i];
                TIME_MAX0=max(TIME_MAX0,GA_Time[i]);
                TIME_MIN0=min(TIME_MIN0,GA_Time[i]);
                MAX0=max(MAX0,deviation[i]);
                MIN0=min(MIN0,deviation[i]);
        }

        SD-=pow(sum/t,2);
        SD=sqrt(SD);
        cout << fixed << setprecision(5);
        cout << "Average deviation: " << sum/t << "%" << endl;
        cout << "Standard deviation: " << SD << endl;
        cout << "Average Time: " << TIME_sum/t << " seconds" << endl;
        cout << "MAX deviation:" << MAX0 << "%" << endl << "min
deviation: " << MIN0 << "%" << endl;
        cout << "MAX TIME: " << TIME_MAX0 << " seconds" << endl <<
"min TIME: " << TIME_MIN0 << " seconds" << endl << endl;

        for(int i=0;i<t;++i)
```

```
        cout << "Case: " << i+1 << endl <<"GA_Ans: " << GA_Ans[i] <<
"   TIME: " << GA_Time[i]  << "\nDP_Ans: " << DP_Ans[i] <<
"\ndeviation: " << deviation[i] << "%" << endl << endl;

    return 0;
}
```