Department of Mathmatics

離散數學期末報告

旅行推銷員問題的解法

數學系大四李香儀

2023

Contents

簡介

旅行推銷員問題

旅行推銷員問題（也稱為旅行商問題或TSP）提出以下問題：“給定一個城市列表和每對城市之間的距離，訪問每個城市一次並返回到初始城市的最短路線是什麼？” 它是組合優化中的**NP-Hard**問題，在**理論計算機科學**和**作業研究**中具有重要意義。旅行購買者問題和車輛路徑問題都是TSP 的推廣。

作為計算複雜性理論中一個典型的判定性問題，TSP的一個版本是給定一個圖和長度L，要求回答圖中是否存在比L 短的迴路（英語：circuit或tour）。該問題被劃分為NP-Complete問題。已知TSP演算法最壞情況下的時間複雜度隨著城市數量的增多而成超多項式（可能是指數）級別增長。

問題在1930年首次被形式化，為最佳化中被研究得最深入的問題之一，許多最佳化方法都奉其為基準。儘管問題在計算上很困難，但已經有了大量的啓發式和精確方法，因此可以完全求解城市數量上萬的實例，並且甚至能在誤差0.01範圍內估計上百萬個城市的問題。

即使是最純粹的TSP 也有多種應用，例如規劃、物流和微芯片製造。稍微修改一下，它在許多領域都作為一個子問題出現，例如DNA 測序。在這些應用中，概念城市代表，例如，客戶、焊接點或DNA 片段，概念距離代表旅行時間或成本，或DNA 片段之間的相似性度量。TSP 也出現在天文學中，因為觀察許多來源的天文學家希望盡量減少在來源之間移動望遠鏡所花費的時間；在此類問題中，TSP 可以嵌入到最優控制問題。在許多應用程序中，**可能會施加額外的約束**，例如有限的資源或時間窗口。

References: Wiki n.d.(b)

圖論中的旅行推銷員問題

在圖論中，可以用**無向加權圖**來對TSP建模，則城市是圖的頂點，道路是圖的邊，道路的距離就是該邊的長度。它是起點和終點都在一個特定頂點，訪問每個頂點恰好一次的最小化問題。通常，該模型是一個完全圖（即每對頂點由一條邊連接）。如果兩個城市之間不存在路徑，則增加一條非常長的邊就可以完成圖，而不影響計算最佳迴路。

References: Wiki n.d.(b)

離散數學中的解決方法

TSP涉及非常多領域，因此在各個領域都發展出了對應了演算法，在離散數學中也是。這裡列舉其中幾種演算法。

蠻力法，brute force method

找出所有可能的路徑，計算每條路徑的成本後，找出最小者。因為在有n個城市時，一共有(n-1)!種可能路徑，所以需要耗費非常多的時間，即使點(城市)並不多。

References: 劉涵初 1987

分支界定法，Brance-and-Bound Method

分支界定法屬於啓發式算法的一種，也是離散數學中用於離散最佳化、組合最佳化的演算法之一。它搜尋最佳解的觀念為，將樹形不斷分支，但隨時以問題的條件限制分支的持續，亦即若知最佳解不會出現在經由此點的路徑，則不必繼續分支，因此所需搜尋的路徑相對蠻力法大為減少。在使用分支界定法時，可以用成本矩陣(在不考慮成本，只考慮距離時，可以以距離取代成本)解決問題。具體的解決方法如下：

最開始先建立一個成本/距離矩陣，假設這是一個考慮4個城市的TSP，從城市1出發最後回到城市1，城市間成本設定如下：

$$\begin{bmatrix} +\infty & 3 & 9 & 7 \\ 3 & +\infty & 6 & 5 \\ 5 & 6 & +\infty & 6 \\ 9 & 7 & 4 & +\infty \end{bmatrix}$$

其中，第ij個元素代表城市i與城市j之間的成本，而每一個城市到自己的成本都定義為無限大(因為不需要用到)。

1. 首先，我們找此問題的一個下限，在成本矩陣中，若每一行(每一列)減去相同的值，顯然，最低成本的路徑不會改變。選擇每一列/行最小的元素，再將每列同時減去該列對應最小元素。以第一列為例，最小的是3，所以整列直接減去3。現在，矩陣變為：

$$\begin{bmatrix} +\infty & 0 & 6 & 3 \\ 0 & +\infty & 3 & 1 \\ 0 & 1 & +\infty & 0 \\ 5 & 3 & 0 & +\infty \end{bmatrix}$$

要注意的是，矩陣中元素不可為負。下限為減去的數字相加的和，這裡是16，代表這個問題中所有解的下限為16。

2. 在知道所有解的下限之後，從第一列開始，考慮現在該列最小的元素所代表的路徑，分別求出選或不選兩種可能性的下限，並且由下限更小的分支再次進行同樣的步驟。

   例如，在這個例子中，第一列中，最小的是0，對應城市1到城市2的成本(這裡0已經不再是成本)，所以這裡考慮選擇這段路徑與不選擇兩種情況。選擇路徑後，所在的行/列已經有路徑，可以直接刪除；不選擇時則設對應元素為無限大。再次計算後，矩陣如下：

$$\begin{bmatrix} +\infty & 2 & 0 \\ 0 & +\infty & 0 \\ 5 & 0 & +\infty \end{bmatrix} \quad \begin{bmatrix} +\infty & +\infty & 3 & 0 \\ 0 & +\infty & 3 & 1 \\ 0 & 0 & +\infty & 0 \\ 5 & 2 & 0 & +\infty \end{bmatrix}$$

   左邊為選擇城市1到城市2的路徑，右邊為不選。計算後可以得到選擇後的下限為17，不選的下限為20。接下來以城市2為起點，再次進行同樣的計算，直至全部走完回到城市1。

3. 這個例子最終結果如下：

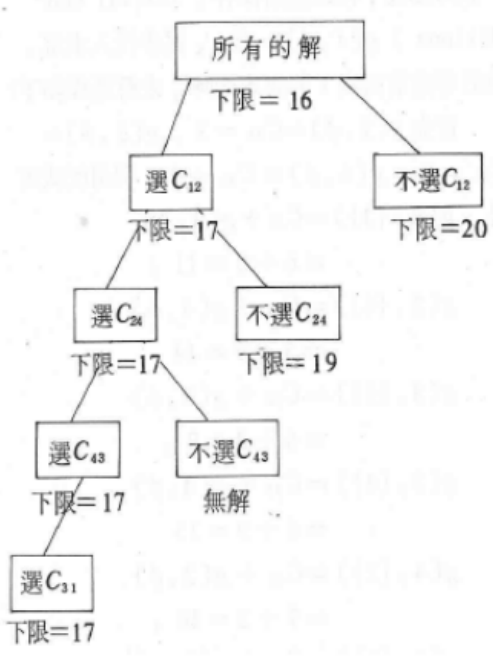

Figure 1: 分支界定法結果

References: 劉涵初 1987

動態規劃，Dynamic Programming

動態規劃常用於解決最佳化問題，主要的方法是將問題分解為若干個子問題，先求解子問題，由子問題的結果得到原問題的解。在TSP中，動態規劃法的求解方法是先建立遞迴關係，再以初始條件代入逐步求解。

定義C_{ij}為城市i,j之間的成本。初始條件為$g(i,\varnothing) = C_{i1}$，即由城市i回到城市1的成本。在最佳路徑中，若由1出發，下一個城市為k，則最佳路徑成本必為加上由k到1的最佳路徑成本。

令所有城市集合為V，上述內容可以表示為數學式如下：

$$g(1, V-1) = \min_{2<=k<=n} C_{1k} + g(1, V-1, k) \tag{1}$$

一般化之後，對於$S \subset V$：

$$g(i, S) = \min_{j \in S} C_{1k} + g(j, S-j) \tag{2}$$

迭代之後，可得出最小路徑成本。具體的程式可以參考winrey (n.d.)。

References: 劉涵初 1987

貪婪演算法，Greedy Algorithm

是一種在每一步選擇中都採取在當前狀態下最佳的選擇，從而希望導致結果是最佳的演算法。貪婪演算法的能夠解決的問題通常包含兩個特性：

1. 貪婪選擇性質,Greedy choice property

   做出目前看來最好的任何選擇，然後解決以後出現的子問題。貪心算法做出的選擇可能取決於目前所做的選擇，但不取決於未來的選擇或子問題的所有解決方案。它迭代地做出一個接一個的貪心選擇，將每個給定的問題簡化為一個更小的問題。

2. 最佳子結構,Optimal substructure

   如果問題的最佳解決方案包含子問題的最佳解決方案，則此問題為最佳子結構。

   References: Wiki n.d.(a)

最近鄰居法，Nearest Neighbor

在TSP中，如果每次都選擇最近的城市，那這就是一種貪婪演算法，這種方法被稱作最近鄰居法（Nearest Neighbor），但是與前面所提到的演算法不同的是，可能無法確定找到的是最短路徑，因

此只能說是一種近似法。jinchenghao (n.d.)提供了程式碼。

References: 劉涵初 1987

Dijkstra Algorithm以及與動態規劃的比較

課本中用於解決最短路徑問題的Dijkstra Algorithm 有時被視為貪婪演算法的一種，有時也被視為動態規劃，但這兩種演算法實質上是不同的。

貪婪演算法對每個子問題的解決方案都做出選擇，不會重新考慮之前的選擇；動態規劃則會儲存以前的運算結果，並根據以前的結果對當前進行選擇，有回退功能(或者說迭代)，詳盡並且保證找到解決方案。

雖然Dijkstra Algorithm 的確在每一步都做出最佳選擇，但是還是會不斷計算、更新距離，所以這種演算法被普遍認為兼具兩者的特性。

References: Wiki n.d.(a)

最小生成樹，Minimum Spanning Tree

1. Prim's algorithm

   普林演算法的策略與先前提到尋找最短路徑的Dijkstra演算法非常相似(事實上這個演算法也被Dijkstra再次發現與發表，所以也稱為Prim–Dijkstra algorithm)，它的作法是先將任意節點加入到樹之中，再選擇與樹距離最近的節點，加入到樹中，反覆此步驟直到所有節點均在樹中，即為最小生成樹。

2. Kruskal's algorithm

   Kruskal演算法的方式是從所有邊中，反覆選擇最短的邊，它的步驟是：將所有的邊依照權重由小到大排序，從最小開始，選擇不會形成環的邊，直到連接所有節點。

   References: ramenkun 2021

這裡需要注意的是，在TSP中，使用這兩中方法之後並不能得到完整的解答，需要再最最小生成樹進行處理使其形成一個circuit，才會是TSP的最終結果。

基因遺傳演算法，Genetic Algorithm

核心概念是先隨機生成一個群體，然後從其中選出基因最為優良的個體。接著讓這些個體去繁衍，產生他們的子代，不斷重複這樣的動作以確保最優良的基因能一直傳承下去。產生一堆隨機的可行路徑，從中挑出較短的路徑進行繁衍，其中一定比率進行突變，使新的路徑誕生（子代路徑+ 突變路徑）。重覆突變、新的路徑誕生的次數取決於我們的進化次數。

TSP的啓發式算法

模擬退火算法，Simulated Annealing, SA

模擬退火演算法（Simulated Annealing, SA）是S.Kirkpatrick, C. D. Gelatt和M.P.Vecchi等人於1983年所提出的通用概率演算法，用來在一個範圍空間內搜尋問題的最佳解。是基於Monte-Carlo迭代求解策略的一種隨機尋優算法。

首先初始化基本參數，決定起始温度t_0、終止温度t_{min}、温度衰退係數、初始解，並透過城市點N 生成相對應的距離矩陣（Distance Matrix），於每次降温過程中，重複k 次運算以決定該温度内的最適解。

在不改變起始和結尾城市的情況下，每次以交換2 個城市順序來產生一新路徑，並計算新路徑的總距離，再和原距離相減以得到兩者之間的差異量diff。

因TSP本身為最小化問題，如果diff<0則直接替換新解，反之若是>0則以$exp^{-diff/t}$的概率接受其成為最適解；在此t 是當前温度，原理為通過在温度高時較有可能接受差解和温度低時幾乎不接受差解從而避免陷入局部最優的情況。每個温度内需完成k 次運算，温度從t_0降到t_{min}，方可達到終止條件。

雖然這個方法無法確定找到最佳解(距離最小值)，可是當使用inverse-log schedule時，可以證明最終結果機率收斂至最佳解，當迭代次數增加時，結果為最佳解的機率趨近1。

References: Cheng 2021b

```
initialize (temperature T, random starting point)
while cool_iteration <= max_iterations
    cool_iteration = cool_iteration + 1
    temp_iteration = 0
    while temp_iteration <= nrep
        temp_iteration = temp_iteration + 1
        select a new point from the neighborhood
        compute current_cost (of this new point)
        δ = current_cost - previous_cost
        if δ<0, accept neighbor
        else, accept with probability exp(-δ/T)
    end while
    T = α * T      (0<α<1)
end while
```

Figure 2: 假代碼

人工蜂群算法，Artificial Bee Colony, ABC

人工蜂群算法（Artificial Bee Colony, ABC）是由Karaboga於2005年提出的全面優化算法，主要來源於蜂群採集蜜的行為，是屬於群體智能算法的其中一種。

ABC是一種全局優化演算，來自蜂群採蜜的行為，蜜蜂根據各自承擔的角度不同進行不同的活動，消息會在蜂群間共享和交流，進入並找到問題的最佳解決方案。優點為穩定性高、控制參數比較少、比較容易理解，不足之處則為存在著著過早收斂的疑惑。求解的流程採用方法為，優先生成距離矩陣（distance matrix），此矩陣是求解最佳路徑的中，所需要點中彼間的距離，並以此石陣為基礎，在每個點不重複的情況下，求出一條總離最短的路。

References: Cheng 2021a

以美國前二十大城市為例的TSP實作

在開始之前，先嘗試將美國前二十大城市以城市中心的經緯度作為其座標，再將其轉換為距離矩陣(或在前述內容中稱為成本矩陣)。這裡採用的前二十大城市與其經緯度如下：

1. 紐約市（紐約州）：緯度40.7128 度，經度-74.0060 度
2. 洛杉磯市（加利福尼亞州）：緯度34.0522 度，經度-118.2437 度
3. 芝加哥市（伊利諾伊州）：緯度41.8781 度，經度-87.6298 度
4. 休士頓市（德克薩斯州）：緯度29.7604 度，經度-95.3698 度
5. 費城市（賓夕法尼亞州）：緯度39.9526 度，經度-75.1652 度
6. 鳳凰城（亞利桑那州）：緯度33.4484 度，經度-112.0740 度
7. 聖安東尼奧市（德克薩斯州）：緯度29.4241 度，經度-98.4936 度
8. 舊金山市（加利福尼亞州）：緯度37.7749 度，經度-122.4194 度
9. 辛辛那提市（俄亥俄州）：緯度39.1031 度，經度-84.5120 度
10. 達拉斯市（德克薩斯州）：緯度32.7767 度，經度-96.7970 度
11. 印第安納波利斯市（印第安納州）：緯度39.7684 度，經度-86.1581 度
12. 亞特蘭大市（喬治亞州）：緯度33.7490 度，經度-84.3880 度
13. 波士頓市（麻薩諸塞州）：緯度42.3601 度，經度-71.0589 度
14. 華盛頓特區（哥倫比亞特區）：緯度38.9072 度，經度-77.0379 度
15. 邁阿密市（佛羅里達州）：緯度25.7617 度，經度-80.1918 度
16. 菲尼克斯市（亞利桑那州）：緯度33.4484 度，經度-112.0740 度
17. 聖地牙哥市（加利福尼亞州）：緯度32.7157 度，經度-117.1611 度
18. 明尼阿波利斯市（明尼蘇達州）：緯度44.9778 度，經度-93.2650 度

19. 聖荷西市（加利福尼亞州）：緯度37.3382 度，經度-121.8863 度

20. 奧斯汀市（德克薩斯州）：緯度30.2672 度，經度-97.7431 度

如果要較精確地計算城市之間的距離，則需要考慮地球曲率，對於相距較遠的城市非常重要。詳細方法可以參考Karr (2022)。但是對於美國國內的城市而言，即使忽略了也沒關係，所以這一部份可做可不做。在進行實作時，主要的目的如下：

1. 解決這20個城市的TSP

2. 確認啓發式算法在參數調整後的受到的影響

3. 對比各種演算法的速度與實用性

分支界定法

在這個例子中，我嘗試用分支定界法解決20個城市的TSP，但是需要時間過長，運行時間接近2小時依舊跑不出結果，因此這裡只做前十大城市的TSP，並且以此對比另一個較常用的精確算法–動態規劃法。用這個方法得到的結果為：

$0 \to 4 \to 8 \to 9 \to 3 \to 6 \to 5 \to 1 \to 7 \to 2 \to 0$

這條路徑距離為9536.636468983726，計算時間為0.34375秒。

動態規劃

動態規劃在實作上相比最小生成樹更簡單一些，可以直接得路徑以及最佳解；在計算時間上比分支界定法短了非常多，是我實作的這三個精確算法中最好的一個方法。用這個方法得到20個城市的結果為：

$0 \to 12 \to 8 \to 10 \to 2 \to 17 \to 5 \to 16 \to 1 \to 18 \to 7 \to 15 \to 6 \to 19 \to 9 \to 3 \to 14 \to 11 \to 13 \to 4 \to 0$

這條路徑距離為12687.075872657446，計算時間為108.390625秒。

在進行10個城市的實作時，所得到的路徑與分支界定法相同，但是計算時間相較分支界定法卻略長了一點，是2.671875秒。關於這一點，可以由這兩個演算法的特性進行一些推測：

分支界定法在城市數量較少時，可以以較少的步數確定下界，但是在城市數量增加之後，卻無法確定會多出多少步(可能選到的都是無法使下界較小的)；相反，動態規劃法在城市數量相同的問題上，計算量也相同，不會因為選的路徑不合適而改變。因此，在計算20個城市的TSP時，兩者的計算時間出現了顯著差異。

最小生成樹法

直接將前述城市的經緯度視為x,y軸座標，並且順序與上述相同，生成最小生成樹可以在1秒內完成(但是畫圖需要的時間較久，不過這裡)，結果如下：

References: Martrix-revolution 2021

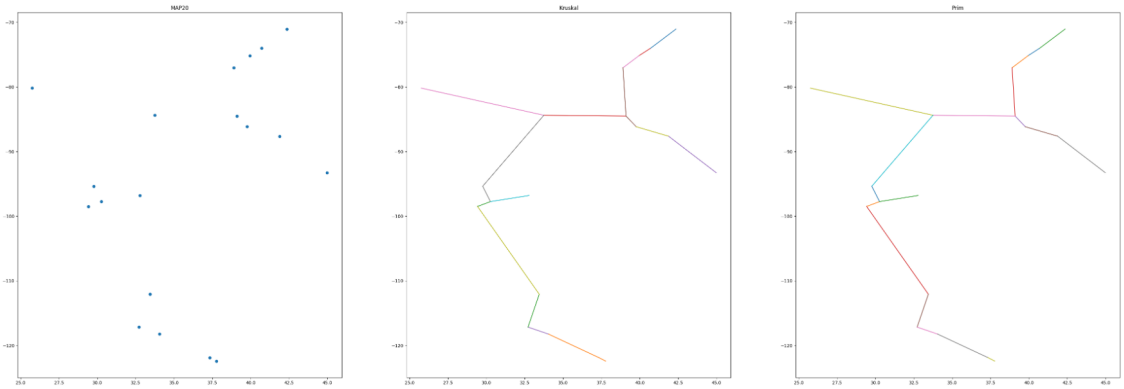

Figure 3: 最小生成樹

因為最小生成樹的演算法是不斷抓最近的點進來，並且不形成閉環，所以無法直接使用其結果，需要再進行"剪枝"，然後連接成一個封閉圖：

1. 刪除$degree > 2$的邊
2. 連接isolation point
3. 如果有兩個以上degree為1，則連通子圖並且刪去用過的點
4. 連接degree為1的兩個點

```
----------------------Kruskal----------------------
[[ 13.          26.          16.24471   ]
 [ 13.          22.          20.5614159 ]
 [ 28.          30.          34.76300223]
 [ 18.          39.          35.01812876]
 [ 27.          29.          42.79594625]
 [ 21.          36.          64.37825374]
 [  6.           7.          68.55971249]
 [ 17.          26.          78.76133032]
 [ 14.          23.          79.49996246]
 [ 10.          12.          79.60181097]
 [  9.          23.          81.07315429]
 [  4.          10.          82.73037045]
 [ 11.          15.          84.42556824]
 [ 26.          37.          85.58855472]
 [  1.          14.          86.14778665]
 [ 16.          35.          92.87254475]
 [  8.          38.         102.66433122]
 [  9.          20.         106.16989458]
 [  2.           5.         106.99338934]
 [ 11.          34.         108.03937114]
 [  3.          30.         114.26407255]
 [  0.          25.         116.68161195]
 [  6.          18.         117.6402133 ]
 [  3.          20.         121.43800064]
 [ 29.          33.         121.55288772]
 [ 22.          33.         130.35430266]
 [ 28.          31.         137.52673628]
 [ 34.          36.         140.20581125]
 [  0.          31.         151.98710099]
 [ 27.          31.         157.61863889]
 [ 12.          24.         158.81397826]
 [ 19.          38.         166.25560145]
 [ 17.          32.         167.51862801]
 [  4.          21.         171.2310691 ]
 [  0.           7.         191.72408473]
 [ 10.          32.         197.55249077]
 [ 24.          38.         199.76043131]
 [  5.          19.         201.73232946]
 [ 19.          35.         276.16185184]]
```

Figure 4: Kruskal的edge

```
----------------------Prim----------------------
[[  0.          25.         116.68161195]
 [  0.          31.         151.98710099]
 [ 31.          28.         137.52673628]
 [ 28.          30.          34.76300223]
 [ 30.           3.         114.26407255]
 [  3.          20.         121.43800064]
 [ 20.           9.         106.16989458]
 [  9.          23.          81.07315429]
 [ 23.          14.          79.49996246]
 [ 14.           1.          86.14778665]
 [ 31.          27.         157.61863889]
 [ 27.          29.          42.79594625]
 [ 29.          33.         121.55288772]
 [ 33.          22.         130.35430266]
 [ 22.          13.          20.5614159 ]
 [ 13.          26.          16.24471   ]
 [ 26.          17.          78.76133032]
 [ 26.          37.          85.58855472]
 [ 17.          32.         167.51862801]
 [  0.           7.         191.72408473]
 [  7.           6.          68.55971249]
 [  6.          18.         117.6402133 ]
 [ 18.          39.          35.01812876]
 [ 32.          10.         197.55249077]
 [ 10.          12.          79.60181097]
 [ 10.           4.          82.73037045]
 [ 12.          24.         158.81397826]
 [  4.          21.         171.2310691 ]
 [ 21.          36.          64.37825374]
 [ 36.          34.         140.20581125]
 [ 34.          11.         108.03937114]
 [ 11.          15.          84.42556824]
 [ 24.          38.         199.76043131]
 [ 38.           8.         102.66433122]
 [ 38.          19.         166.25560145]
 [ 19.           5.         201.73232946]
 [  5.           2.         106.99338934]
 [ 19.          35.         276.16185184]
 [ 35.          16.          92.87254475]]
```

Figure 5: Prim的edge

模擬退火法

因為這是啓發式算法，所以在使用這個方法時，基本上只能得到近似解，無法確定是最佳解，所以需要進行窮舉以確保是最佳解。但是對於RAM比較小的電腦(我的電腦)，窮舉法需要的記憶體太大，會無法執行，所以只能捨棄這一步，而這個動作將會導致每一次迭代結果不一，甚至在t_0更高、迭代次數更多時可能反而模擬出更遠的路徑。導致這種原因的因子有很多：

1. 初始路徑為隨機，最初的總長度長短不一，可能差別非常大。例如figure6,7中，兩次模擬所有參數相同，但是初始距離相差甚遠：一個從42298收斂至33483，一個從39200收斂至29818。
   因為這裡的參數設定是初始溫度t_0=100，迭代次數k=10，冷卻係數coolnum=0.7，所以並未收斂到一個令人滿意的結果也可想而知。

```
Initial Route:  [13, 10, 18, 1, 17, 6, 0, 19, 15, 11, 7, 12, 14, 4, 2, 5, 8, 16, 3, 9, 13]
Initial distance:  42298.04515262552
==================================================
By using method of exhustive, can get the min distance below:
Final Route:  [13, 0, 17, 18, 1, 15, 6, 19, 11, 10, 7, 12, 14, 4, 2, 5, 8, 16, 3, 9, 13]
Final Distance:  33483.40676437872
```

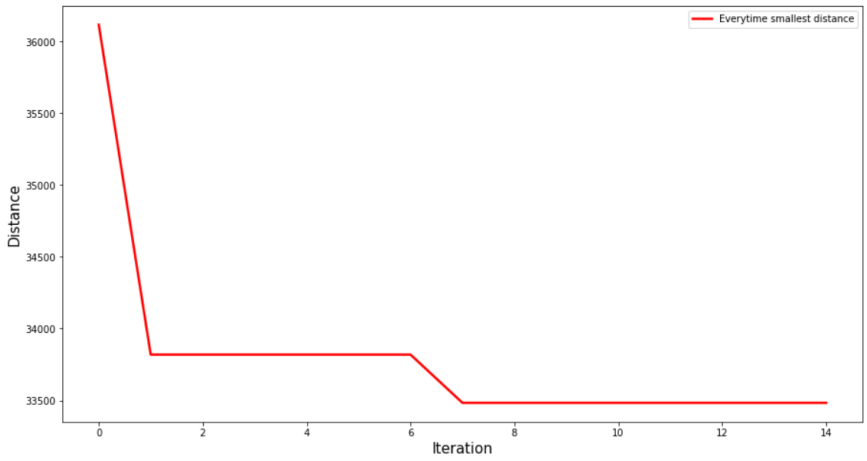

Figure 6: 初始距離更大

即使如此，在實作時依舊採用了隨機生成初始路徑，是為了防止這個演算法在最初選擇不合適的初始路徑，導致運作過程中出現問題。

2. 每次都隨機決定交換兩個城市的順序，在不同初始路徑的基礎上增加了更多變異。

3. 在新生成路徑距離更長時，依舊以$exp^{-diff/t}$的概率接受新路徑，在迭代越少次時越容易接受。(這裡的t是指當時的溫度)

初始温度t_0

為了確認模擬退火算法中個個參數值對於最終結果的影響，我將初始温度t_0調升至500，其餘不變，與t_0=100的做對比。在時間上，即使將初始温度調升至1000，也幾乎沒有影響，都不到1秒；在距離上，t_0=100時在模擬十次內最短的距離是17075，t_0=500時在模擬十次內最短的距離是15508，相比精確解12687，相對誤差從34%降到了22%，但是誤差依舊有些大，初始温度再繼續提升之後也無法看到比較明顯的相對誤差下降了。

迭代次數k

現在固定初始温度為500，將k調整至50與之前的k=10結果做對比。相比調整初始温度，調整迭代次數出現的最變化是最終路徑長度的變異性降低，當然路徑長度也變小了，而且執行時間依舊接近0秒。模擬十次內最短距離為13112，相對誤差降至3.3%，已經是令人較為滿意的結果了。

迭代次數k

固定初始温度為500，迭代次數為50，將冷卻係數coolnum調整至0.98與之前的coolnum=0.7結果做對比，冷卻係數越高，退火越難、需要進行更多次冷卻才能低於最低温度。這一步對於計算時間的影響較大，但是也都在0.2秒的範圍內。即使將另外兩個參數調整回原本的值，計算時間依舊比前兩步調

```
Initial Route:  [1, 10, 18, 2, 14, 3, 6, 8, 9, 13, 4, 0, 7, 19, 15, 11, 12, 16, 17, 5, 1]
Initial distance:  39200.58986671622
==================================================
By using method of exhustive, can get the min distance below:
Final Route:  [1, 18, 6, 3, 9, 2, 10, 8, 14, 13, 4, 0, 7, 19, 15, 11, 12, 16, 17, 5, 1]
Final Distance:  29818.388109649164
```

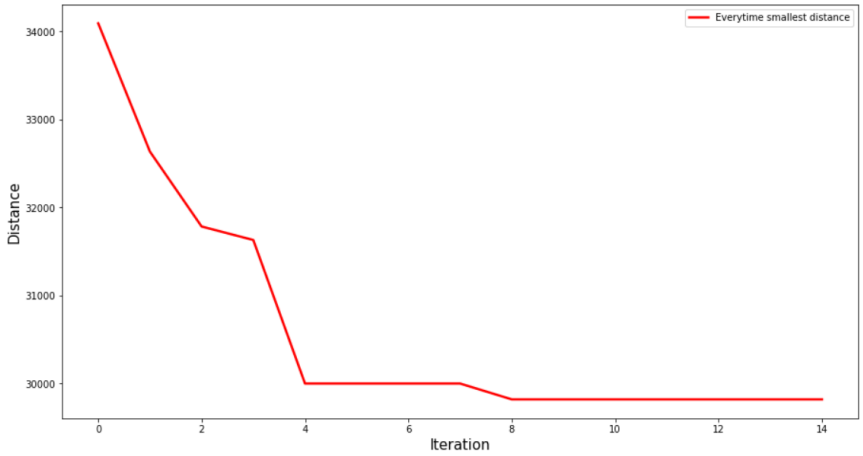

Figure 7: 初始距離更小

整後要更長一些，不過時間依舊很短，大約只有0.05秒左右。

在十次模擬中，有一次恰與動態規劃法的解完全相同。但是在模擬時還出現了一條更短的路徑，長度為12123：

$15 \to 5 \to 16 \to 1 \to 18 \to 7 \to 17 \to 2 \to 10 \to 8 \to 12 \to 0 \to 4 \to 13 \to 11 \to 14 \to 3 \to 9 \to 19 \to 6 \to 15$

References

Cheng, Hunter (2021a). *Python — (Artificial Bee Colony, ABC)(Traveling Salesman Problem, TSP)*. https://reurl.cc/r583L4.

Cheng, Hunter (2021b). *Python — (Simulated Annealing, SA)(Traveling Salesman Problem, TSP)*. https://reurl.cc/AA86Ae.

jinchenghao (n.d.). *TSP*. https://github.com/jinchenghao/TSP/blob/master/Greedy.py.

Karr, Douglas (2022). *Calculate or query great circle distance between latitude and longitude points using Haversine formula (PHP, JavaScript, Java, Python, MySQL, MSSQL example)*. https://zh-tw.martech.zone/calculate-great-circle-distance/.

Martrix-revolution (2021). *MSTTSP*. https://www.cnblogs.com/Martrix-revolution/p/15412430.html.

ramenkun (2021). *MST*. https://ithelp.ithome.com.tw/articles/10278295?sc=iThomeR.

Wiki (n.d.[a]). *Greedy algorithm*. https://en.wikipedia.org/wiki/Greedy_algorithm.

Wiki (n.d.[b]). *Travelling salesman problem*. https://en.wikipedia.org/wiki/Travelling_salesman_problem.

winrey (n.d.). *tsp.py*. https://gist.github.com/winrey/c7de9e4743a2fc79cf8a30cfcd9c604d.

(1987). *Travelling salesman problem*. https://web.math.sinica.edu.tw/mathmedia/HTMLarticle18.jsp?mID=11302.