程式設計 (112-1) 作業六解法分享

B12705055 張佑丞

November 2023

Table of Contents

題目架構與設計 TA-algorithm

題目架構

根據題目敘述，我們需要做的是找到一組解決方案 x（將 x_{ij} 輛車型為 i 的車分配至場站 j），使得在考慮競食效應下能夠擁有最大化利潤，也就是將以下式子的值最大化

目標式

$$\sum_{i \in I} \sum_{j \in J} 24 R_i \left(U_{ij} - \sum_{k \in K} Q^{(k)} \sum_{j' \in N_j^{(k)}} \sum_{i' \in I} x_{i'j'} + Q^{(0)} \right) x_{ij}$$

TA-algorithm 告訴了我們什麼

TA-algorithm 的初衷

每一輪中，演算法會去找在哪個場站多投放哪一個車型的車一輛能最大化考慮了競食效應的目標式值。... 一直重複直到多投放任何一輛車都不會提高目標式值為止。

Pseudo Code of TA-algorithm

```
while(true)
    Set when x[l][v] add 1 can optimaize the value
    Initialize [l,v] = [-1,-1]
    for i from 0 to n-1
        for j from 0 to m-1
            Set x[i][j] add 1
            if Current Value is the greatest
                Set [l, v] = [i, j]
            Set x[i][j] deduct 1
    if l is -1
        break the loop
```

計算當下目標值

回到目標式，我們將 Summation 從外到內計算

目標式

$$\sum_{i \in I} \sum_{j \in J} 24 R_i \left(U_{ij} - \sum_{k \in K} Q^{(k)} \sum_{j' \in N_j^{(k)}} \sum_{i' \in I} x_{i'j'} + Q^{(0)} \right) x_{ij}$$

Pseudo Code of getVal()

我們定義 $N_j^{(k)}$ 為與站場 j 距離為 k 的所有站場所形成的集合

getVal (to compute the current value of the given solution)

```
Set N[j][k] = {l}, where Distance between j and l is k
Initialize the return value Ret to 0
for i from 0 to n-1
    for j from 0 to m-1
        Ret add ...
        for k from 0 to 3
            for l in N[j][k]
                for t from 0 to n-1
                    Ret deduct ...
            Ret add ...
Return Ret
```

這個程式碼的效率

因此 getVal() 這個函數的時間複雜度為 $O(N^2M^2)$，搭配前幾頁所提及的 TA-algorithm 算法需額外 $O(NM)$ 時間，總複雜度即為 $O(N^3M^3)$。

計算量分析

題目限制 $1 \leq n \leq 20, 1 \leq m \leq 500$，$O(N^3M^3)$ 的計算量大約為 10^{12}

優化計算目標值函數

優化計算目標值函數

TA-algorithm 的關鍵

在哪個場站投了哪「一」輛車 ... 能讓當前目標值最大化

我們可以很好地運用這個特性，將思考從「每投一次就重新計算一次目標值」變成「在這裡投了一輛車對整體目標值的改變」

新增一輛車所造成的值變化

依據目標式，每新增一輛車在 x_{lv}，在式中會影響的地方即為 x_{lv} 加一，以及其他「與 x_{ij} 有關的 $x_{i'j'}$」加一

目標式

$$\sum_{i\in I}\sum_{j\in J} 24R_i(U_{ij} - \sum_{k\in K} Q^{(k)} \sum_{j'\in N_j^{(k)}} \sum_{i'\in I} x_{i'j'} + Q^{(0)})x_{ij}$$

新增一輛車所造成的值變化

$$24R_l(U_{lv} - \sum_{k\in K} Q^{(k)} \sum_{j'\in N_v^{(k)}} \sum_{i'\in I} x_{i'j'} + Q^{(0)}) + (-24\sum_{k\in K} Q^{(k)} \sum_{j'\in N_v^{(k)}} \sum_{i'\in I} R_i x_{i'j'})$$

這個式子看似複雜，其實討論 i, j 是否相等於 l, v 即可得出，建議大家自己想想看！

這樣對優化有什麼幫助

我們接著定義兩個參數 SumOfVal 與 SumOfStation，其中

定義 SumOfVal

$$\mathrm{SumOfVal}^{(j)} = \sum_{i \in I} R_i \times x[i][j]$$

定義 SumOfStation

$$\mathrm{SumOfStation}^{(j)} = \sum_{i \in I} x_{ij}$$

新的目標式

我們可以發現，只要好好維護好這兩個參數的值，上式便可以化簡至非常簡單，如下所示

新的目標式

$$24R_l(U_{lv} - \sum_{k \in K} Q^{(k)} \sum_{j' \in N_v^{(k)}} \mathsf{SumOfStation}^{(j')} + Q^{(0)})$$

$$-24 \sum_{k \in K} Q^{(k)} \sum_{j' \in N_v^{(k)}} \mathsf{SumOfVal}^{(j')}$$

而要維護這兩個參數也十分簡單，由於每次搜尋後僅增加一輛車，故只要在每次搜尋最後對 SumOfVal[j] 和 SumOfStation[j] 分別加上 $R[i]$ 與 1 即可

將其寫成 Pseudo Code

```
FastVal (to compute the current value in an optimized way)

Set return value Ret to current value before operation
for k from 0 to 3
    for l in S[j][k]
        Ret Deduct SumOfStation[l]...

for k from 0 to 3
    for l in S[j][k]
        Ret Deduct SumOfVal[l]...
Return Ret
```

分析這段程式碼的效率

由此代碼可見，這樣做的時間複雜度能夠大大降低，僅須 $O(M)$ 即可完成一次目標值的查詢，搭配前面所提到的 TA-algorithm 的算法，我們可以得出總複雜度即為 $O(NM^2)$，計算量大約落在 10^7，可謂是足夠有效率的算法了！

將 TA-algorithm 再優化

將 TA-algorithm 再優化

TA-algorithm 的初衷

每一輪中，逐一檢查哪個組合加一後能使目標值最大化。有的話就加一 ... 一直重複直到多投放任何一輛車都不會提高目標式值為止。

但一定要每次都走最大值的路線才能使最終目標值最大化嗎?

改進 TA-algorithm

換個想法

是否在不走當下最大值的情況下也能達到甚至超越原本的最大值?

我們不難想像若是暴力解一定無法在指定時限內完成

合理假設

只有當前目標值與當前最大值在指定誤差範圍內，我們才都納入考慮

換句話說，若是當前目標值與最大值相差甚遠，未來能「追上」的機率也十分低，故在此我們無須考慮

Remark

兩數的誤差可以考慮相減的差，或著相除等...

Pseudo Code of the Optimized Approach

與上次不同，我採用遞迴 (Recursive) 與深度優先搜尋 (Depth First Search) 的技巧進行設計

```
solve with given current value before operation

Find max possible value and set it as MX
if MX > Global MX
    Set Global MX = MX
    Save the current solution
for i from 0 to n-1
    for j from 0 to m-1
        update current value
        if difference between MX and current value is less than
            DiffAcceptable
           Set x[i][j] add 1
           solve() with current value
           Set x[i][j] deduct 1
```

缺點與其解決方法

缺點:

1. 遞迴時限不易估計，不容易判斷何時能夠使用遞迴
2. DiffAcceptable 參數不容易估計

解決方法:

1. 只在前二十項小測資執行，只要當前運行時間已超出指定範圍，便跳回原本的算法執行
2. 需要多次嘗試來找到能讓分數最大化的值

對於範例測資是否有更好的解

Sample Input:

```
2 3 0.1 0.05 0.03 0.01
50 70
10 5
10 8 7
0.9 0.8 0.7
0.7 0.85 0.75
0 350 800
350 0 450
800 450 0
```

Output of TA-algorithm

```
3,0,0
1,2,2
```

Output of the optimized approach

```
4,0,0
0,2,3
```