ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA

BÁO CÁO BÀI TẬP LỚN
CẤU TRÚC RỜI RẠC

ĐỀ TÀI

BÀI TOÁN NGƯỜI GIAO HÀNG
(TRAVELING SALESMAN PROBLEM)

GV hướng dẫn: Mai Xuân Toàn

Sinh viên thực hiện: Nguyễn Việt Hoàng

Mã số sinh viên: 2311066

Mã lớp: CO1007(L01)

HỒ CHÍ MINH, 2024

Mục lục

1 Giới thiệu bài toán Người Giao Hàng (Traveling Salesman Problem)

Bài toán người giao hàng (tiếng Anh: travelling salesman problem - TSP) là một bài toán NP-Hard thuộc thể loại tối ưu rời rạc hay tổ hợp được nghiên cứu trong vận trù học hoặc lý thuyết khoa học máy tính. Bài toán được phát biểu như sau. Cho trước một danh sách các thành phố và khoảng cách giữa chúng, tìm chu trình ngắn nhất thăm mỗi thành phố đúng một lần.

Bài toán này đã được nghiên cứu nhiều và đưa ra rất nhiều kỹ thuật cũng như phương pháp để giải quyết. Các kỹ thuật thường được nghĩ tới đầu tiên như kỹ thuật **vét cạn (Complete Search)**, kỹ thuật **tham ăn (Greedy)**, kỹ thuật **nhánh cận (Branch and Bound)**,... Nhưng lý do em chọn thuật toán **Quy hoạch động (Dynamic Programing)** - Trong bài toán này em cần giải quyết với rất nhiều đỉnh vì thế bài toán về thời gian cũng như độ phức tạp của thuật toán là vấn đề rất nan giải đối với em. Trong quá trình tìm hiểu thì thuật toán này rất thích hợp trong trường hợp này vì có thể nói khi xét đến tất cả các trường hợp một cách tổng quát thì thuật toán này có thể giải quyết nhanh hơn và độ phức tạp của thuật toán cũng ít hơn so với các thuật toán khác như nhánh cận, tham ăn,.. Tuy nhiên, thuật toán còn có nhiều hạn chế vì thuật toán sẽ lưu tất cả các trường hợp có thể xảy ra nên chúng ta cần một mảng rất là lớn để có thể thực hiện được điều này(tuy nhiên, em đã nghiên cứu để một phần nào đó khắc phục được điều này, em sẽ nói rõ hơn về phần này ở phía sau).Nhưng nhìn chung thì kỹ thuật này cũng đáng để triển khai và thực hiện trên TSP.

2 Thuật toán Quy Hoạch Động - Dynamic Programing

Khi áp dụng **Dynamic Programming** (DP) vào bài toán **TSP (Traveling Salesman Problem)**, mục tiêu là tối ưu hóa quá trình tìm kiếm hành trình ngắn nhất qua một tập hợp các điểm (thành phố) sao cho mỗi điểm chỉ được thăm một lần và kết thúc tại điểm xuất phát.

Mô hình hóa bài toán

Gọi số lượng thành phố là n và các thành phố được đánh số từ 0 đến $n-1$. Ta cần tìm một chuỗi thành phố sao cho tổng khoảng cách từ điểm xuất phát quay lại ban đầu là tối thiểu. Vấn đề có thể được biểu diễn bằng một đồ thị có các đỉnh là các thành phố và các cạnh có trọng số là khoảng cách giữa các thành phố.

Biểu diễn trạng thái bằng DP

Dùng $dp[S][i]$ để lưu trữ chi phí ngắn nhất khi đã thăm các thành phố trong tập con S và hiện tại đang ở thành phố i.

- S là một tập con của các thành phố (một bitmask), có thể biểu diễn là một số nguyên với mỗi bit tương ứng với một thành phố. Nếu thành phố đó đã được đi thăm thì bit tương ứng với thành phố đó là 1. Ngược lại, nếu thành phố đó chưa được đi thăm thì giá trị bit tương ứng của thành phố đó sẽ là 0.
- i là thành phố hiện tại trong tập hợp S.

Chuyển trạng thái

Từ trạng thái $dp[S][i]$, có thể chuyển sang các trạng thái tiếp theo bằng cách đi đến thành phố chưa thăm trong tập con S. Giả sử ta đang ở thành phố i và muốn chuyển đến thành phố j (chưa thăm). Công thức cập nhật:

$$dp[S \cup \{j\}][j] = \min(dp[S][i] + \text{dist}(i, j))$$

Trong đó $\text{dist}(i, j)$ là khoảng cách giữa thành phố i và thành phố j.

Tối ưu bài toán

Nếu như ta xét tất cả các trạng thái của dp thì số phần tử của mảng này khi xét đối với n thành phố (n đỉnh) sẽ là $2^n \cdot n$ phần tử. Trong quá trình thực hiện bài toán, em phát hiện ra rằng mình chỉ cần quan tâm đến các trường hợp trong đó xuất hiện đỉnh xuất phát đã đi rồi. Vì thế, em chỉ xét các trạng thái mà bit

tương ứng với đỉnh xuất phát có giá trị là 1. Điều này giúp làm giảm 1 nữa số lượng phần tử của mảng dp đi 1 nửa.

Kết thúc

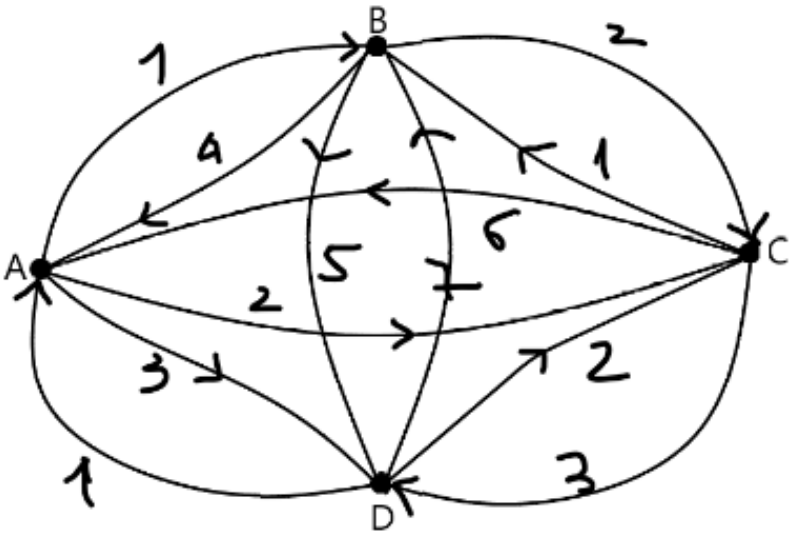
Cuối cùng, ta tìm giá trị nhỏ nhất trong các trạng thái $dp[S][i]$ khi S là tập hợp tất cả các thành phố và i là bất kỳ thành phố nào, cộng với khoảng cách quay lại thành phố xuất phát.

Độ phức tạp

Bởi vì ta cần tính toán cho mỗi tập con S và mỗi thành phố i, độ phức tạp thời gian của thuật toán này là $O(n^2 \cdot 2^n)$, với n là số thành phố. Đây là một thuật toán có độ phức tạp cao, nhưng có thể áp dụng cho các bài toán có số lượng thành phố vừa phải (khoảng 20-30 thành phố). Dynamic Programming giúp giảm thiểu sự tính toán trùng lặp trong quá trình tìm kiếm hành trình ngắn nhất, từ đó giải quyết bài toán TSP hiệu quả hơn so với cách tiếp cận brute force.

Mô phỏng bài toán

Sau đây sẽ là phần mô phỏng thuật toán tổng quát của Kỹ thuật Quy Hoạch Động (Dynamic Programing) khi đã được cải tiến: Cho đồ thị có hướng, đầy đủ, với số đỉnh là 4:

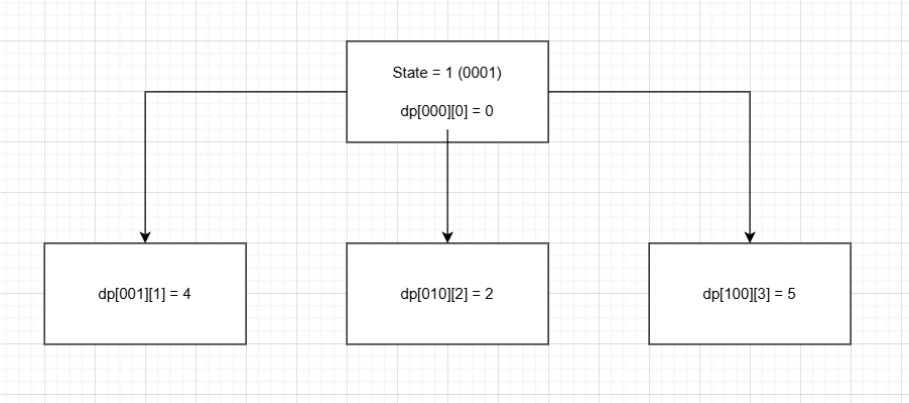


Đồ thị có hướng đầy đủ 4 đỉnh.

Chúng ta sẽ có ma trận kề của đồ thị trên như sau: $\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 0 & 2 & 5 \\ 6 & 1 & 0 & 3 \\ 1 & 7 & 2 & 0 \end{bmatrix}$

Giả sử bài toán có đỉnh xuất phát là đỉnh B.

Ta sẽ có mảng dp[8][4], cài đặt cho các phần tử của mảng là số vô cùng và dp[0][0] = 0. Thuật toán của em thay vì trạng thái em sẽ xét DCBA như bình thường thì lúc này em sẽ đổi lại thành DCAB để sao cho đỉnh xuất phát sẽ là bit có trọng điểm thấp nhất (bit ứng với đỉnh A) và bit tương ứng với đỉnh A sẽ là bit nằm ở vị trí của bit ứng với đỉnh xuất phát trong trường hợp này là B. Bởi lẽ em muốn thay đổi điều này để có thể giảm bớt trạng thái và dễ quản lí các bit hơn. Các trạng thái của biến State của em sẽ là các số lẻ đi từ 1 cho đến nhỏ hơn $2^4 = 16$. **Lưu ý** ở mảng dp[S][i]: S thể hiện ở mô phỏng dưới đang ở dạng hex. bài toán sẽ có sơ đồ như sau:



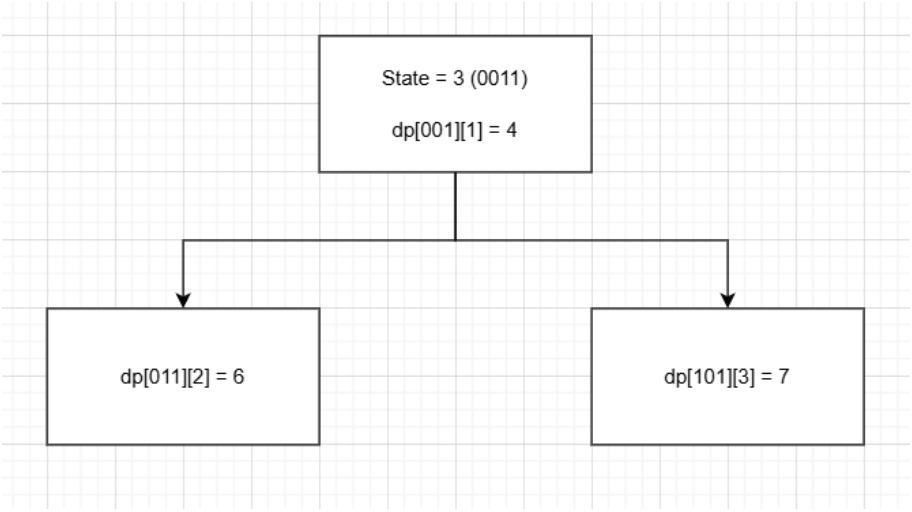
Trạng thái thứ 1

Ở trạng thái 1 này ta sẽ sẽ tính trọng số đi từ đỉnh đầu đến các đỉnh còn lại của đồ thị và cập nhật lại các giá trị này vào dp. Ta sẽ có:

$dp[001][1] = dp[000][0] + G[1][0] = 0 + 4 = 4$

$dp[010][2] = dp[000][0] + G[1][2] = 0 + 2 = 2$

$dp[100][3] = dp[000][0] + G[1][3] = 0 + 5 = 5$

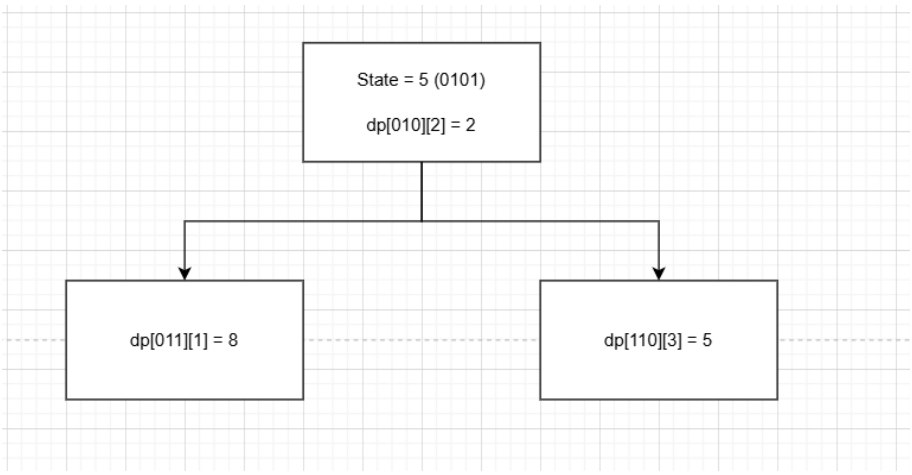


Trạng thái thứ 3

Ở trạng thái thứ 3 này, chúng ta sẽ xét đỉnh tại vị trí thứ 1 đến các vị trí khác(thứ 2 và thứ 3) bằng cách lấy tổng của dp[001][1] + G[0][i] (trong đó i lần lượt là 2 và 3) và lưu lại trong mảng dp. Ta sẽ có:

$dp[011][2] = dp[001][1] + G[0][2] = 4 + 2 = 6$

$dp[110][3] = dp[001][1] + G[0][3] = 4 + 3 = 7$

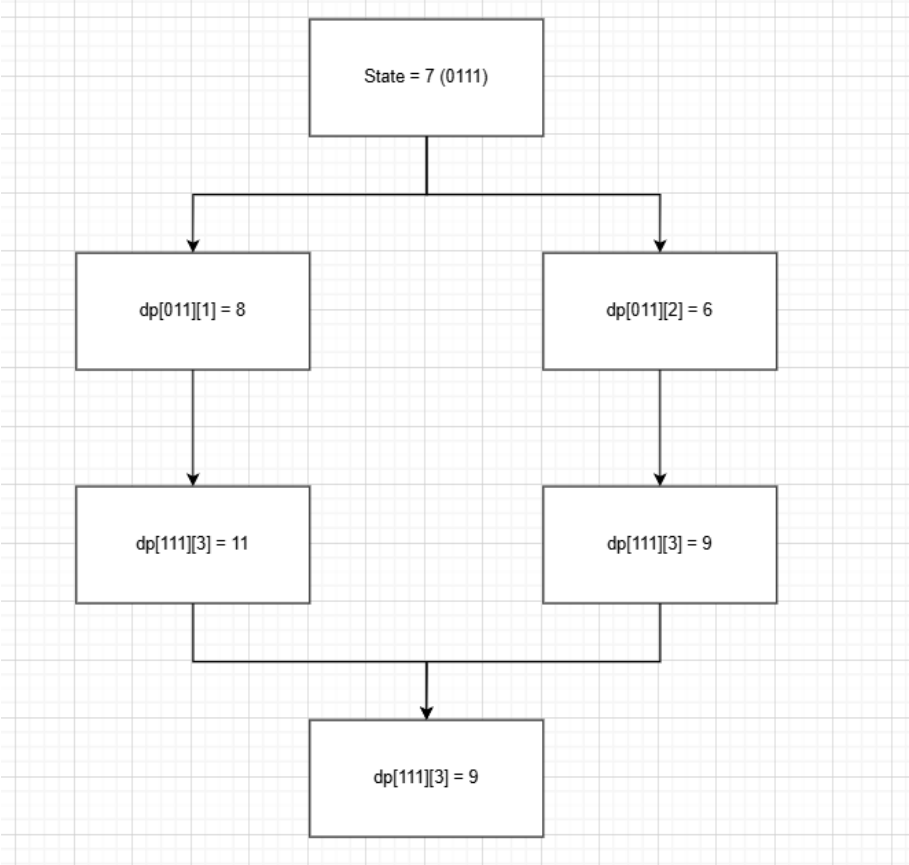


Trạng thái thứ 5

Đến trạng thái thứ 5, chúng ta sẽ đi từ đỉnh thứ 2 đến 2 đỉnh chưa đi là đỉnh 1 và đỉnh 3 sau đó cập nhật lại vào mảng dp. Ta sẽ có:

$dp[011][1] = dp[010][2] + G[2][0] = 2 + 6 = 8$

$dp[110][3] = dp[010][2] + G[2][3] = 2 + 3 = 5$



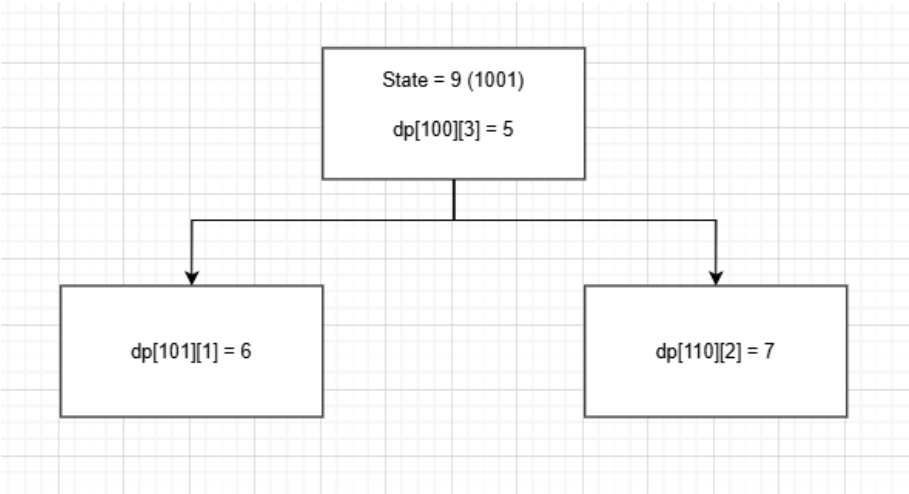
Trạng thái thứ 7

Ở bước thứ 7, Chúng ta có thể xuất phát từ 2 đỉnh là đỉnh thứ 1 và đỉnh thứ 2 để đi đến đỉnh thứ 3. Tuy nhiên chúng ta sẽ gặp phải trường hợp là đi hết các đỉnh xong rồi đến đỉnh cuối cùng là đỉnh 3 được tính 2 lần nhưng trong 2 lần tính đó chúng ta sẽ lấy lần tính có trọng số nhỏ nhất. Ta sẽ có:

$dp[111][3] = dp[011][1] + G[0][3] = 8 + 3 = 11$

$dp[111][3] = dp[011][2] + G[2][3] = 6 + 3 = 9$

Vì $9 < 11$ nên sẽ lấy $dp[111][3] = 9$

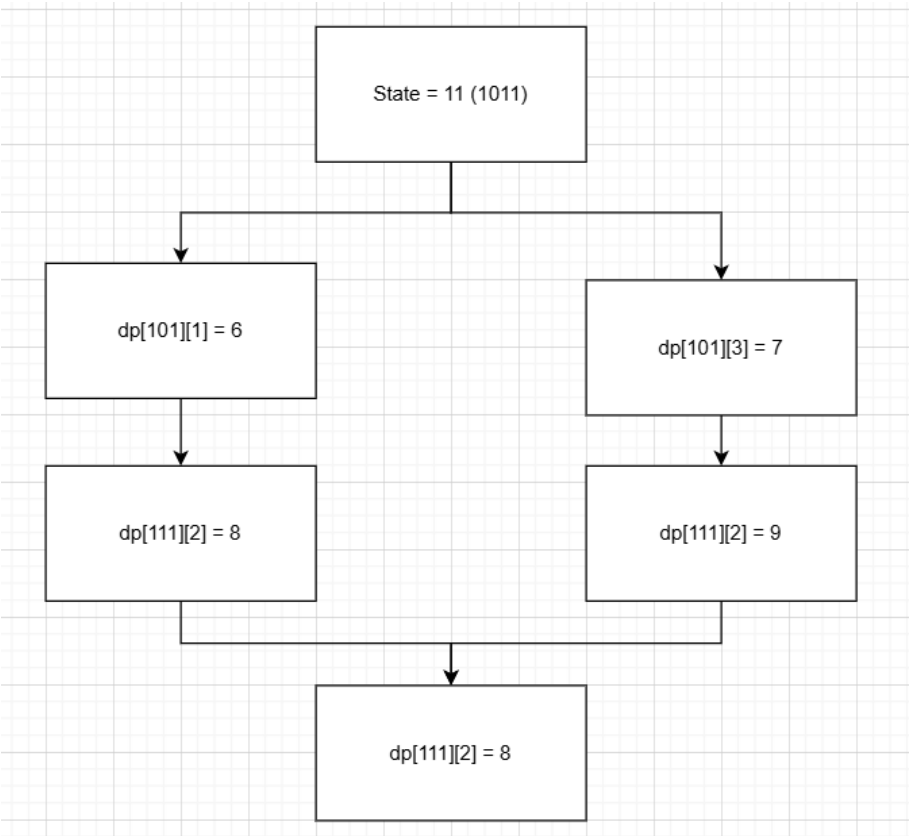


Trạng thái thứ 9

Ở trạng thái thứ 9 này, chúng ta sẽ bắt đầu đi từ đỉnh thứ 3 đến các đỉnh còn lại là đỉnh 1 và đỉnh 2, ta sẽ có:

$dp[101][1] = dp[100][3] + G[3][0] = 5 + 1 = 6$

$dp[101][2] = dp[100][3] + G[3][2] = 5 + 2 = 7$



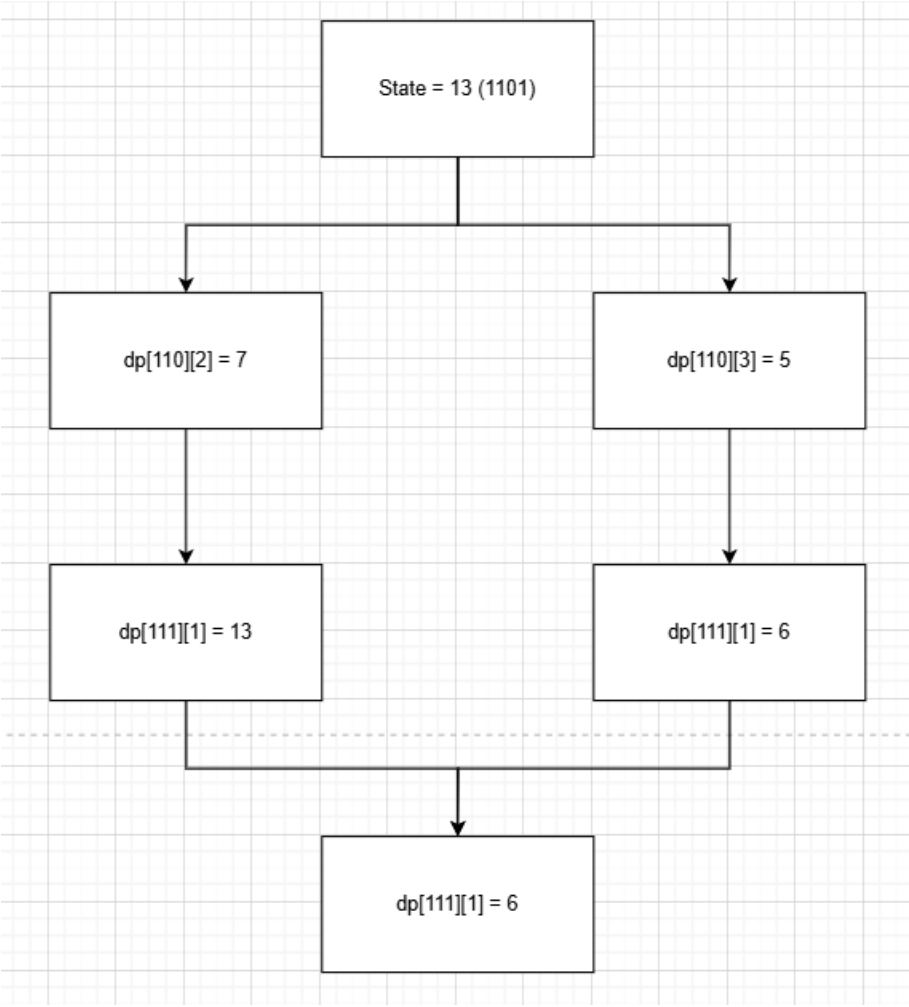
Trạng thái thứ 11

Tiếp tục đến trạng thái thứ 11, chúng ta sẽ có thể bắt đầu đi từ đỉnh thứ 1 và đỉnh thứ 3 để đi đến đỉnh thứ 2, sau đó ta cũng sẽ thực hiện giống như đã làm ở bước 7, ta sẽ có:

$dp[111][2] = dp[101][1] + G[0][2] = 6 + 2 = 8$

$dp[111][2] = dp[101][3] + G[3][2] = 7 + 2 = 9$

Vì $8 < 9$ nên ta sẽ có $dp[111][2] = 8$



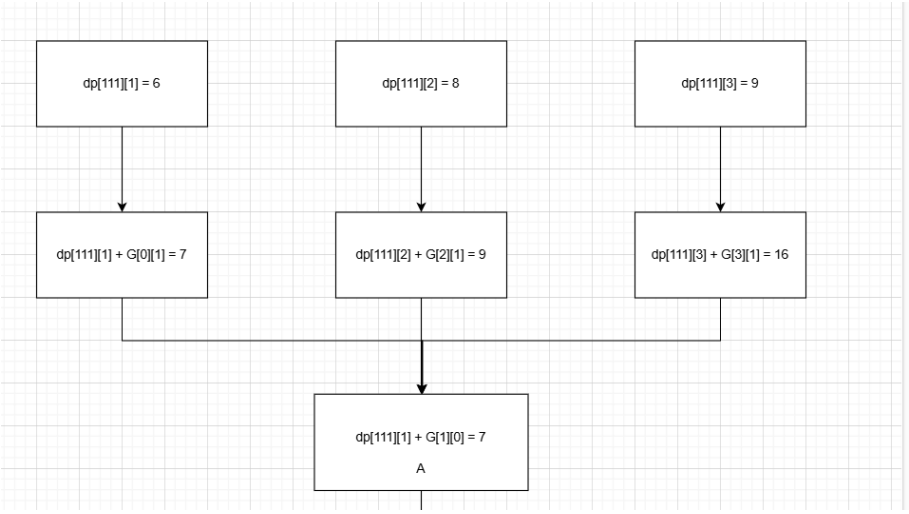
Trạng thái thứ 13

Ở trạng thái thứ 13, chúng ta có thể đi từ đỉnh thứ 3 và đỉnh thứ 2 để đi đến đỉnh thứ 1 và sẽ làm tương tự như bước 7, ta sẽ có:

$dp[111][1] = dp[110][2] + G[2][0] = 7 + 6 = 13$

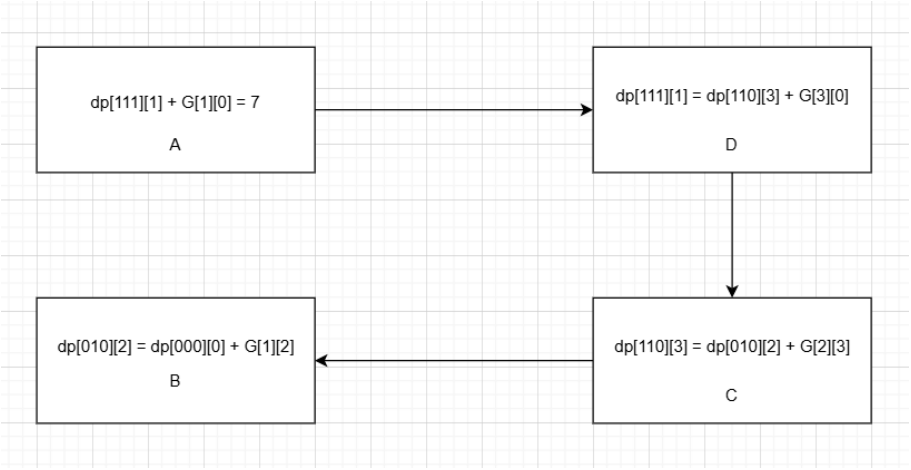
$dp[111][1] = dp[110][3] + G[3][0] = 5 + 1 = 6$

vì $6 < 13$ nên $dp[111][1] = 6$



Tìm kiếm đường đi ngắn nhất

Sau khi đã thực hiện tính toán hết giá trị của mảng dp thì sau đó ta sẽ xét tại vị đã đi hết tất cả các đỉnh đã đi hết trong biểu đồ và sau đó chúng ta sẽ phải cộng từ vị trí cuối cùng đó đến điểm xuất phát để có thể tìm được đường đi ngắn nhất, thì sau khi cộng lại thì có thể thấy đường đi từ đỉnh thứ 1(đỉnh A) đến đỉnh bắt đầu (đỉnh B) là ngắn nhất nên ta sẽ lấy đỉnh A là đỉnh kết thúc.



Tìm kiếm các đỉnh phía trước

Sau khi đã tìm kiếm được đỉnh A là đỉnh kết thúc thì ta cũng dựa vào kết quả tính giá trị hiện tại của đỉnh A để lần mò đến các đỉnh trước đó nữa. Từ đã ta sẽ có: A <- D <- C <- B.

kết luận: Chúng ta sẽ có 1 chu trình là B -> C -> D -> A -> B với chi phí nhỏ nhất là 7

lưu vào mảng dp như sau: dp[1«i][i] = $G[Start\_idx][i]$ (trong đó i sẽ đi từ đỉnh

có index là 1 đến hết). lưu ý tại vị trí của $Start\_idx$ chúng ta sẽ sử dụng vị trí này để biểu diễn đỉnh A vì chúng ta đã thay thế vị trí của đỉnh bắt đầu vào vị trí của đỉnh A nên khi gặp vị này thì nó sẽ bằng trọng số đi từ đỉnh bắt đầu đến đỉnh A.

3 Các hàm thực thi

1. Hàm Left_Shift

```
5   int Left_Shift(int Num_Ver) {
6       return 1 << Num_Ver;
7   }
```

Hàm dịch trái (bit)

Hàm trên có tác dụng dịch trái Num_Ver lần

2. Hàm Run_DP

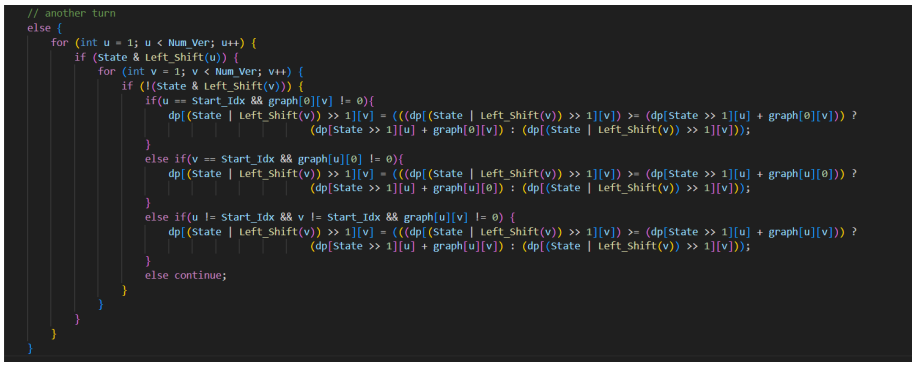
```
void Run_DP(int graph[30][30], int Num_Ver, int Start_Idx) {

    for (int State = 1; State < Left_Shift( Num_Ver); State += 2) {
            for (int i = 0; i < Num_Ver; i++) {
                dp[State >> 1][i] = _INFI;
            }
    }

    dp[0][0] = 0;

    for (int State =  1; State < Left_Shift( Num_Ver); State += 2) {
        // turn 1
        if(State == 1){
            for(int u = 1; u < Num_Ver; u++){
                if(graph[Start_Idx][u] != 0){
                    dp[(State | Left_Shift(u)) >> 1][u] = graph[Start_Idx][u];
                }
            }
            if(graph[Start_Idx][0] != 0){
                dp[(State | Left_Shift(Start_Idx)) >> 1][Start_Idx] = graph[Start_Idx][0];
            }
        }
```

Tính toán mảng dp với State = 1

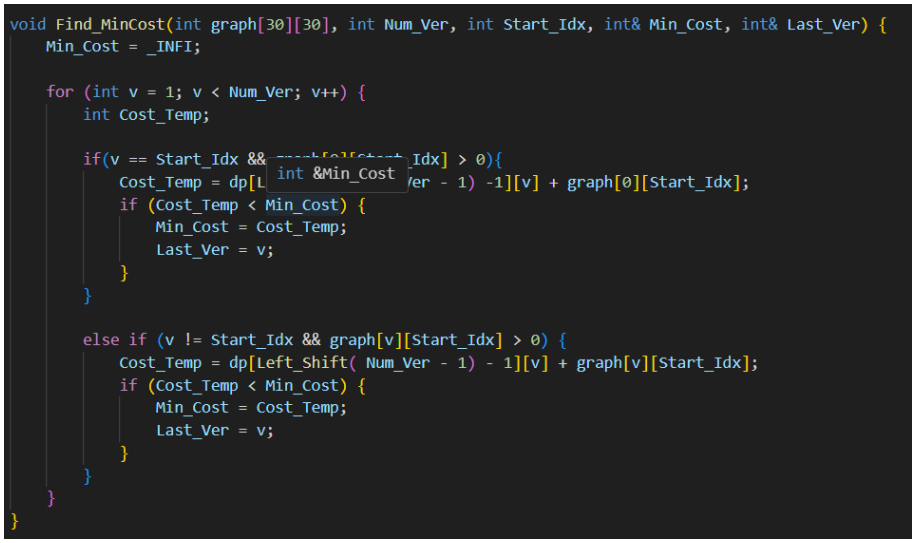


```
// another turn
else {
    for (int u = 1; u < Num_Ver; u++) {
        if (State & Left_Shift(u)) {
            for (int v = 1; v < Num_Ver; v++) {
                if (!(State & Left_Shift(v))) {
                    if(u == Start_Idx && graph[0][v] != 0){
                        dp[(State | Left_Shift(v)) >> 1][v] = (((dp[(State | Left_Shift(v)) >> 1][v]) >= (dp[State >> 1][u] + graph[0][v])) ?
                                                (dp[State >> 1][u] + graph[0][v]) : (dp[(State | Left_Shift(v)) >> 1][v]));
                    }
                    else if(v == Start_Idx && graph[u][0] != 0){
                        dp[(State | Left_Shift(v)) >> 1][v] = (((dp[(State | Left_Shift(v)) >> 1][v]) >= (dp[State >> 1][u] + graph[u][0])) ?
                                                (dp[State >> 1][u] + graph[u][0]) : (dp[(State | Left_Shift(v)) >> 1][v]));
                    }
                    else if(u != Start_Idx && v != Start_Idx && graph[u][v] != 0) {
                        dp[(State | Left_Shift(v)) >> 1][v] = (((dp[(State | Left_Shift(v)) >> 1][v]) >= (dp[State >> 1][u] + graph[u][v])) ?
                                                (dp[State >> 1][u] + graph[u][v]) : (dp[(State | Left_Shift(v)) >> 1][v]));
                    }
                    else continue;
                }
            }
        }
    }
}
```

Tính toán mảng dp với các State còn lại

Hàm Run_DP có tác dụng tính tất cả các giá trị vào mảng dp để từ đó có thể truy ngược là và tìm được đường đi ngắn nhất.

4. Hàm Find_MinCost



```
void Find_MinCost(int graph[30][30], int Num_Ver, int Start_Idx, int& Min_Cost, int& Last_Ver) {
    Min_Cost = _INFI;

    for (int v = 1; v < Num_Ver; v++) {
        int Cost_Temp;

        if(v == Start_Idx && graph[0][Start_Idx] > 0){
            Cost_Temp = dp[L  int &Min_Cost  /er - 1) -1][v] + graph[0][Start_Idx];
            if (Cost_Temp < Min_Cost) {
                Min_Cost = Cost_Temp;
                Last_Ver = v;
            }
        }

        else if (v != Start_Idx && graph[v][Start_Idx] > 0) {
            Cost_Temp = dp[Left_Shift( Num_Ver - 1) - 1][v] + graph[v][Start_Idx];
            if (Cost_Temp < Min_Cost) {
                Min_Cost = Cost_Temp;
                Last_Ver = v;
            }
        }
    }
}
```

Hàm Find_MinCost

Hàm này được sử dụng để tính toán trọng số nhỏ nhất của các đường đi từ đó nhận giá trị và đỉnh cuối cùng của đường đi có trọng số nhỏ nhất.

5. Hàm CPath

```
string CPath(int graph[30][30], int Num_Ver, int Start_Idx, int Last_Ver) {
    string path;
    int State = Left_Shift(Num_Ver) - 1;
    int pos = Last_Ver;

    path += (char)(Start_Idx + 'A'); // Start at the starting vertex
    path += ' ';
    if(Last_Ver == Start_Idx){
        path += 'A';
    }
    else {
        path += (char)(Last_Ver + 'A');
    }


    while (State != 1 + (1 << pos)) {
        int prev_pos = -1;
        for (int u = 1; u < Num_Ver; u++) {
            if((u == Start_Idx) && (State & Left_Shift(u)) && (u != pos) && (graph[0][pos] != 0)
                && (dp[State >> 1][pos] == dp[(State ^ Left_Shift(pos)) >> 1][u] + graph[0][pos])){
                prev_pos = u;
                break;
            }
            else if(pos == Start_Idx && (State & Left_Shift(u)) && u != pos && graph[u][0] != 0
                    && dp[State >> 1][pos] == dp[(State ^ Left_Shift(pos)) >> 1][u] + graph[u][0]){
                prev_pos = u;
                break;
            }
            else if (u != Start_Idx && pos != Start_Idx && (State & Left_Shift(u)) && u != pos && graph[u][pos] != 0
                    && dp[State >> 1][pos] == dp[(State ^ Left_Shift(pos)) >> 1][u] + graph[u][pos]) {
                prev_pos = u;
                break;
            }
        }
```

```
        if(prev_pos == -1){
            return "No valid path";
        }

        path += ' ';
        if(prev_pos == Start_Idx){
            path += 'A';
        }
        else{
            path += (char)(prev_pos + 'A');
        }
        State ^= Left_Shift(pos);
        pos = prev_pos;
    }

    path += ' ';
    path += (char)(Start_Idx + 'A');


    string CPath;
    for (int v = path.size() - 1; v >= 0; v--) {
        CPath += path[v];
    }

    return CPath;
}
```

Hàm CPath này có tác dụng truy ngược từ đỉnh cuối đến đỉnh đầu để có thể tìm được đường đi ngắn nhất theo các đỉnh và trả về chuỗi ký tự cần tìm.

6. Hàm Traveling

```
string Traveling(int graph[30][30], int Num_Ver, char Start_Ver) {

    int Start_Idx = Start_Ver - 'A';
    Run_DP(graph, Num_Ver, Start_Idx);

    int Min_Cost = _INFI;
    int Last_Ver;
    Find_MinCost(graph, Num_Ver, Start_Idx, Min_Cost, Last_Ver);

    if (Min_Cost == _INFI) {
        return "No valid tour exists.";
    }

    return CPath(graph, Num_Ver, Start_Idx, Last_Ver);
}
```

Hàm này sẽ là hàm thực thi bài toán bằng cách gọi các hàm ở trên và trả về 1 chuỗi string thể hiện đường đi ngắn nhất.

Cách tối ưu bài toán

Như đoạn code đã thể hiện, em đã tối ưu bài toán bằng cách làm giảm số lượng phần tử mà mảng dp cần lưu trữ bằng cách loại bỏ được đỉnh bắt đầu trong mảng này. Nhưng để loại bỏ đỉnh này trong mảng thì em cần đổi vị trí thứ tự của đỉnh này so với đỉnh A với mục đích là em sẽ xét tất cả các trạng thái (State) là số lẻ (trạng thái mà luôn ghé thăm đỉnh thứ 0 - đỉnh bắt đầu). Từ việc làm trên mà ban đầu chỉ có thể làm được với tối đa 20 đỉnh mà hiện giờ đoạn code của em có thể thực hiện với 26 đỉnh, bên cạnh đó thời gian thực thi của code em có thể nói là chạy rất nhanh trong các phương pháp tìm đường đi ngắn nhất và chính xác.

4 Lời kết

Em xin chân thành cảm ơn Thầy đã đọc báo cáo của em về đề tài TSP. Những nội dung em tìm hiểu ở trên đều là những gì em đã tự tìm hiểu và mày mò nên có thể có những sai sót hoặc có những điều chưa thực sự hay và tối ưu. Trên tinh thần học hỏi và cầu tiến, em sẽ luôn ghi nhận những đóng góp của Thầy để có thể sửa đổi và phát triển ạ.

5 Tài liệu tham khảo

Tìm hiểu về bài toán người bán hàng (Traveling Salesman Problem)
Đường dẫn: Click here

Tìm hiểu về giải thuật Quy Hoạch Động (Dynamic Programing)
Đường dẫn: Click here

Tìm hiểu về giải thuật Quy Hoạch Động (Dynamic Programing) trong bài toán người giao hàng(Traveling Salesman Problem)
Đường dẫn: Click here