Unidirectional TSP

Arief Widhiyasa, 13505126

Mahasiswa Teknik Informatika ITB
Jl. Wira Angun-Angun No.26, Bandung
e-mail: arief@clawford.net

ABSTRAK

Makalah ini akan membahas suatu metode penyelesaian masalah unidirectional TSP. Mungkin sudah banyak yang mengenal tentang TSP. TSP (Travelling Salesperson Problem) merupakan masalah NP (Non Polynomial) yang sampai sekarang masih tidak dapat dipecahkan. TSP secara abstrak dapat dikatakan pencarian rute tersingkat untuk mengunjungi seluruh tempat yang ada. Sedangkan, Unidirectional TSP merupakan masalah turunan dari TSP. Penjelasan lebih lanjut mengenai Unidirectional TSP ini akan dijelaskan kemudian.

Masalah ini merupakan masalah yang diajukan pada lomba pemrograman ACM. Sekarang masalah ini masih dapat diakses melalui arsip soal di dalam situs ACM dengan nomor kode masalah 00116. Masalah ini juga terdapat pada buku pemrograman berjudul Programming Challenge karangan 'Steven S. Skiena' dan 'Miguel A. Revilla'. Berbeda dengan TSP standar, masalah ini dapat dipecahkan dalam waktu *polynomial*. Metode utama yang akan digunakan dalam memecahkan masalah ini adalah dengan memanfaatkan *Dynamic Programming* (Pemrograman Dinamis).

Kata kunci: Unidirectional TSP, Dynamic Programming, Recursive (Rekursif).

1. PENDAHULUAN

TSP merupakan masalah klasik dalam dunia pemrograman yang sampai saat ini belum dapat dipecahkan. Unidirectional TSP, sangat berbeda dengan TSP yang biasa. Soal ini memiliki solusi polinomial, sehingga dapat dipecahkan. Berikut adalah definisi permasalahannya (dalam Bahasa Inggris), yang disadur langsung dari web resmi ACM online problem archive (http://acm.uva.es/p/v1/116.html):

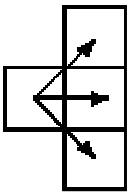
Background

Problems that require minimum paths through some domain appear in many different areas of computer science. For example, one of the constraints in VLSI routing problems is minimizing wire length. The Traveling Salesperson Problem (TSP) -- finding whether all the cities in a salesperson's route can be visited exactly once with a specified limit on travel time -- is one of the canonical examples of an NP-complete problem; solutions appear to require an inordinate amount of time to generate, but are simple to check.

This problem deals with finding a minimal path through a grid of points while traveling only from left to right.

The Problem

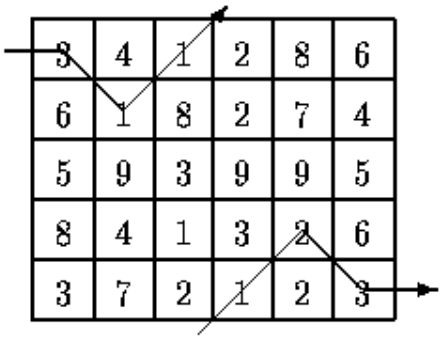
Given an $m \times n$ matrix of integers, you are to write a program that computes a path of minimal weight. A path starts anywhere in column 1 (the first column) and consists of a sequence of steps terminating in column n (the last column). A step consists of traveling from column i to column i+1 in an adjacent (horizontal or diagonal) row. The first and last rows (rows 1 and m) of a matrix are considered adjacent, i.e., the matrix ``wraps'' so that it represents a horizontal cylinder. Legal steps are illustrated below.



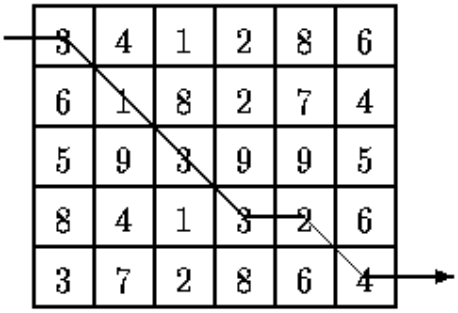
Gambar 1. Legal Steps

The *weight* of a path is the sum of the integers in each of the n cells of the matrix that are visited.

For example, two slightly different 5×6 matrices are shown below (the only difference is the numbers in the bottom row).



Gambar 2. Sample Case 1



Gambar 3. Sampe Case 2

The minimal path is illustrated for each matrix. Note that the path for the matrix on the right takes advantage of the adjacency property of the first and last rows.

The Input

The input consists of a sequence of matrix specifications. Each matrix specification consists of the row and column dimensions in that order on a line followed by $m * n$ integers where *m* is the row dimension and *n* is the column dimension. The integers appear in the input in row major order, i.e., the first *n* integers constitute the first row of the matrix, the second *n* integers constitute the second row and so on. The integers on a line will be separated from other integers by one or more spaces. Note: integers are not restricted to being positive. There will be one or more matrix specifications in an input file. Input is terminated by end-of-file.

For each specification the number of rows will be between 1 and 10 inclusive; the number of columns will be between 1 and 100 inclusive. No path's weight will exceed integer values re-presentable using 30 bits.

The Output

Two lines should be output for each matrix specification in the input file, the first line represents a minimal-weight path, and the second line is the cost of a minimal path. The path consists of a sequence of *n* integers (separated by one or more spaces) representing the rows that constitute the minimal path. If there is more than one path of minimal weight the path that is *lexicographically* smallest should be output.

Sample Input

```
5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 8 6 4
5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 1 2 3
2 2
9 10 9 10
```

Sample Output

```
1 2 3 4 4 5
16
1 2 1 5 4 5
11
1 1
19
```

2. METODE

Dapat dilihat dalam deskripsi permasalahan di atas, permasalahan ini tergolong cukup mudah. Pada kasus terburuk, akan terdapat 10 baris dan 100 kolom sehingga secara total akan terdapat 3^(100-1) * 10 jalur perjalanan. Yaitu sekitar 1.718 x 10^48 kemungkinan jalur, sehingga bila menggunakan metode brute force akan menghabiskan terlalu banyak waktu.

Jikalau kita menggunakan metode traversal graph standar, yaitu DFS (Depth First Search) atau BFS (Breadth First Search). Waktu yang dihabiskan tidak jauh

dengan metoda brute force karena semua jalur harus dicoba. Metoda backtrack yang menggunakan pembatas berupa jarak minimum yang telah dicapai sampai saat ini juga tidak terlalu efisien, karena ada kasus terburuk dimana dia akan bekerja seburuk brute force.

Dalam kasus ini yang dibutuhkan adalah runtime yang sangat singkat sehingga dapat memecahkan banyak kasus dalam batas waktu yang ditentukan.

Jika diperhatikan lebih mendalam, kita dapat membuat suatu fungsi rekursif top-down dalam memecahkan masalah ini.

2.1 Top Down Recursive Solution

Melihat kasus tersebut, kita dapat membuat suatu fungsi jalanMin(baris, kolom) yang akan menghasilkan jumlah weight minimum untuk mencapai baris dan kolom tersebut. Sedangkan untuk mencari weight minimum tersebut, kita akan memanggil fungsi yang sama pada (baris-1, kolom-1), (baris, kolom-1), dan (baris+1, kolom-1), kemudian memilih yang paling kecil diantara 3 tempat itu, dan menjumlahkannya dengan weight (baris, kolom) sehingga didapat total weight dalam mencapai suatu kotak.

Jikalau kita melihat kasus secara keseluruhan, maka untuk memecahkan kasus ini, kita hanya perlu memanggil fungsi rekursif jalanMin() pada setiap kotak di kolom akhir. Pembatas rekursinya adalah bila dia mencapai kolom pertama. Kemudian kita tinggal memilih dari hasil rekursi semua baris yang terminimum adalah jawabannya.

Namun ternyata, fungsi rekursif tersebut memiliki runtime yang tidak berbeda jauh dengan metoda brute force, traversal ataupun backtrack. Untuk mempersingkat waktunya kita harus meneliti fungsi rekursif ini lebih dalam. Ternyata pada suatu pemanggilan fungsi rekursif akan terdapat banyak sekali pemanggilan fungsi rekursif yang berulang-ulang.

Sehingga dengan menggunakan teknik memoisasi akan didapatkan solusi yang cocok untuk permasalahan ini. Teknik memoisasi merupakan penyimpanan data hasil perhitungan fungsi rekursi dimana data tersebut akan digunakan lagi. Pseudocode fungsi rekursifnya kurang lebih seperti ini :

```
//array m menyimpan data input
int M[10][100]
//array r menyimpan nilai rekursi
int r[10][100]
//int m dan n menyimpan ukuran

int jlMin(int row, int col) {
      if (r[row][col]!= -1)
            return r[row][col];
      else {
            r[row][col] =
            min(jlMin(row-1,col-1),
            jlMin(row,col-1),
            jlMin(row+1,col-1) +
            M[row][col];
            return r[row][col];
      }
}

int main() {
      //inisialisasi r = -1
      for (i=0; i<m; i++)
            for (j=1; j<n; j++)
                  r[i][j] = -1;
      //inisialisasi col ke 0
      for (i=0; i<m; i++)
            r[i][0]=M[i][0];
      //rekursi col terakhir
      for (i=0; i<m,; i++)
            jlnMin(i,n-1);
}
```

Fungsi rekursif diatas akan menghasilkan sebuah matriks yang mengandung nilai weight minimum untuk sampai ke kotak tersebut. Selanjutnya untuk mencari hasilnya hanya perlu memilih diantara matriks kolom terakhir, dan untuk mencatat pathnya hanya tinggal menggunakan sebuah matriks tambahan yang akan mencatat arah jalur pengambilan nilai minimum setiap kali matriks memoisasi diisi.

Penarikan solusi seperti ini merupakan penarikan solusi secara Top Down, dimana solusi dicari dari hasil akhir yang akan mencari ke bawah. Selain cara Top Down ini terdapat cara Bottom Up yang lebih effisien karena tidak menggunakan stack memory. Cara bottom up ini lebih dikenal dengan sebutan Dynamic Programming.

2.2 Bottom Up Dynamic Programming

Pembangunan ruang solusi menggunakan metoda bottom up ini menyerupai pembangunan solusi top down. Hanya saja merupakan kebalikannya. Dari awal pembangunan kita memulai dari kotak pada kolom pertama. Kemudian untuk weight kotak pada kolom selanjutnya merupakan minimum dari 3 arah weight kotak di belakangnya ditambah dengan weight dirinya sendiri. Sehingga bila diteruskan sampai akhir akan menemukan solusinya dengan memilih minimum pada kotak-kotak di kolom terakhir.

Sebenarnya dalam mencari suatu solusi Dynamic Programming, pencarian suatu fungsi rekursi yang melakukan perulangan mutlak wajib dilakukan untuk memudahkan pembangunan solusi dan memastikan solusi benar. Sehingga dengan berbekal fungsi rekursif bada subbab 2.1, maka pembangunan solusi dapat dengan mudah dilakukan, dan Unidirectional TSP tersebut dapat terpecahkan.

Berikut adalah kode lengkap untuk masalah tersebut dalam bahasa C/C++ :

```
#include <stdio.h>
int M[10][100], int min[10][100],int path[10][100];
int m,n, finalnode;

int minvalue(int col){
  int i, finalmin;
  finalmin = min[0][col];
  finalnode = 0;
  for(i = 0; i < m; i++){
    if(min[i][col] < finalmin){
      finalnode = i;
      finalmin = min[i][col];
    }
  }
  return finalmin;
}

int FindMin(){
  int i, j, north,east,south, np, ep, sp;
  for(i = 0; i < m; i++){
    path[i][0] = 0;
    min[i][0] = M[i][0];
  };
  if(n > 0){
    for(j = 1; j < n; j++){
      for(i = 0; i < m; i++){
        np = (i+m-1)%m;
        ep = i;
        sp = (i+1)%m;
        north = min[np][j-1];
        east = min[ep][j-1];
        south = min[sp][j-1];

        if(north <= east && north <= south){
          min[i][j] = north + M[i][j];
          path[i][j] = np;
          if( (north == east) && ( ep < path[i][j])) path[i][j] = ep;
          if( (north == south) && ( sp < path[i][j])) path[i][j] = sp;
        } else if(east <= north && east <= south){
          min[i][j] = east + M[i][j];
          path[i][j] = ep;
          if( (east == north) && ( np < path[i][j])) path[i][j] = np;
          if( (east == south) && ( sp < path[i][j])) path[i][j] = sp;
        } else if(south <= north && south <= east){
          min[i][j] = south + M[i][j];
          path[i][j] = sp;
          if( (south == north) && ( np < path[i][j])) path[i][j] = np;
          if( (south == east) && ( ep < path[i][j])) path[i][j] = ep;
        };
      };
    };
  };
  return minvalue(n-1);
}
```

```
void printpath(int col, int
row){
  if(col == 0){
    printf("%d", row+1);
  } else {
    printpath(col-1,
path[row][col]);
    printf(" %d",row+1);
  }
}


int main(){
  int i, j, fmin;

  while(scanf("%d %d", &m, &n)==2){

    for(i = 0; i < m; i++)
      for(j = 0; j < n; j++)
        scanf("%d", &M[i][j]);

    fmin = FindMin();

    printpath(n-1, finalnode);
    printf("\n");

    printf("%d\n", fmin);
  }
  return 0;
}
```

IV. KESIMPULAN

1. Unidirectional TSP merupakan masalah yang dapat dipecahkan dalam waktu polinomial sehingga tidak tergolong NP seperti TSP standar.
2. Dalam pembangunan ruang solusi terdapat dua metoda yaitu Top-Down dan Bottom-Up. Kedua solusi ini sangat erat hubungannya, dan keduanya membutuhkan suatu fungsi rekursif sebelum pembangunan ruang solusinya.
3. Pada metoda Top-Down, solusi dibangun menggunakan teknik memoisasi dimana hasil perhitungan fungsi rekursi disimpan sehingga tidak memerlukan penghitungan ulang.
4. Pada metoda Bottom-Up, solusi dibangun menggunakan Dynamic Programming dari lemma-lemma dasar menuju suatu solusi global.

REFERENSI

[1] Steven S.Skiena & Miguel A. Revilla, "Programming Challenges", Springer: USA, 2003.