Bitmasking and Dynamic Programming | Set-2 (TSP)

In this post, we will be using our knowledge of dynamic programming and Bitmasking technique to solve one of the famous NP-hard problem “Travelling Salesman Problem”.

Before solving the problem, we assume that the reader has the knowledge of

- DP and formation of DP transition relation
- Bitmasking in DP
- Travelling Salesman problem

To understand this concept lets consider the below problem :

Problem Description:

```
Given a 2D grid of characters representing
a town where '*' represents the
houses, '#' represents the blockage,
'.' represents the vacant street
area. Currently you are (0, 0) position.

Our task is to determine the minimum distance
to be moved to visit all the houses and return
to our initial position at (0, 0). You can
only move to adjacent cells that share exactly
1 edge with the current cell.
```

The above problem is the well-known Travelling Salesman Problem.

The first part is to calculate the minimum distance between the two cells. We can do it by simply using a BFS as all the distances are unit distance. To optimize our solution we will be pre-calculating the distances taking the initial location and the location of the houses as the source point for our BFS.

Each BFS traversal takes O(size of grid) time. Therefore, it is **O(X * size_of_grid)** for overall pre-calculation, where X = number of houses + 1 (initial position)

Now lets's think of a DP state

So we will be needing to track the visited houses and the last visited house to uniquely identify a state in this problem.

Therefore, we will be taking **dp[index][mask]** as our DP state.

Here,
index : tells us the location of current house
mask : tells us the houses that are visited (if ith bit is set in mask then this means that the ith dirty tile is cleaned)

Whereas dp[index][mask] will tell us the minimum distance to visit X(number of set bits in mask) houses corresponding to their order of their occurrence in the mask where the last visited house is house at location index.

State transition relation

So our initial state will be **dp[0][0]** this tells that we are currently at initial tile that is our initial location and mask is 0 that states that no house is visited till now.

And our final destination state will be **dp[any index][LIMIT_MASK]**, here LIMIT_MASK = **(1<<N) – 1**
and N = number of houses.

Therefore our DP state transition can be stated as

```
dp(curr_idx)(curr_mask) = min{
    for idx : off_bits_in_curr_mask
       dp(idx)(cur_mask.set_bit(idx)) + dist[curr_idx][idx]
}
```

The above relation can be visualized as the minimum distance to visit all the houses by standing at curr_idx house and by already visiting cur_mask houses is equal to min of distance between the curr_idx house and idx house + minimum distance to visit all the houses by standing at idx house and by already
visiting (**cur_mask | (1 <<idx)**) houses.

So, here we iterate over all possible idx values such that cur_mask has i^{th} bit as 0 that tells us that i^{th} house is not visited.

Whenever we have our mask = LIMIT_MASK, this means that we have visited all the houses in the town. So, we will add the distance from the last visited town (i.e the town at cur_idx positon) to the initial position (0, 0).

The C++ program for the above implementation is given below:

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

```
#include <bits/stdc++.h>
using namespace std;
```

```
#define INF 99999999
#define MAXR 12
#define MAXC 12
#define MAXMASK 2048
#define MAXHOUSE 12

// stores distance taking souce
// as every dirty tile
int dist[MAXR][MAXC][MAXHOUSE];

// memoization for dp states
int dp[MAXHOUSE][MAXMASK];

// stores coordinates for
// dirty tiles
vector < pair < int, int > > dirty;

// Directions
int X[] = {-1, 0, 0, 1};
int Y[] = {0, 1, -1, 0};

char arr[21][21];

// len : number of dirty tiles + 1
// limit : 2 ^ len -1
// r, c : number of rows and columns
int len, limit, r, c;


// Returns true if current position
// is safe to visit
// else returns false
// Time Complexity : O(1)
bool safe(int x, int y)
{
    if (x >= r or y>= c or x<0 or y<0)
       return false;
    if (arr[x][y] == '#')
       return false;
    return true;
}


// runs BFS traversal at tile idx
// calulates distance to every cell
// in the grid
// Time Complexity : O(r*c)
void getDist(int idx){

    // visited array to track visited cells
    bool vis[21][21];
    memset(vis, false, sizeof(vis));

    // getting current positon
    int cx = dirty[idx].first;
    int cy = dirty[idx].second;

    // initializing queue for bfs
    queue < pair < int, int > > pq;
    pq.push({cx, cy});

    // initializing the dist to max
    // because some cells cannot be visited
    // by taking source cell as idx
    for (int i = 0;i<= r;i++)
        for (int j = 0;j<= c;j++)
            dist[i][j][idx] = INF;

    // base conditions
    vis[cx][cy] = true;
    dist[cx][cy][idx] = 0;

    while (! pq.empty())
```

```
    {
        auto x = pq.front();
        pq.pop();
        for (int i = 0;i<4;i++)
        {
           cx = x.first + X[i];
           cy = x.second + Y[i];
           if (safe(cx, cy))
           {
               if (vis[cx][cy])
                   continue;
               vis[cx][cy] = true;
               dist[cx][cy][idx] = dist[x.first][x.second][idx] + 1;
               pq.push({cx, cy});
            }
         }
    }
}

// Dynamic Programming state transition recursion
// with memoization. Time Complexity: O(n*n*2 ^ n)
int solve(int idx, int mask)
{
    // goal state
    if (mask == limit)
       return dist[0][0][idx];

    // if already visited state
    if (dp[idx][mask] != -1)
       return dp[idx][mask];

    int ret = INT_MAX;

    // state transiton relation
    for (int i = 0;i<len;i++)
    {
        if ((mask & (1<<i)) == 0)
        {
            int newMask = mask | (1<<i);
            ret = min( ret, solve(i, newMask)
                + dist[dirty[i].first][dirty[i].second][idx]);
        }
    }

    // adding memoization and returning
    return dp[idx][mask] = ret;
}

void init()
{
    // initializing containers
    memset(dp, -1, sizeof(dp));
    dirty.clear();

    // populating dirty tile positions
    for (int i = 0;i<r;i++)
        for (int j = 0;j<c;j++)
        {
            if (arr[i][j] == '*')
               dirty.push_back({i, j});
        }

    // inserting ronot's location at the
    // begining of the dirty tile
    dirty.insert(dirty.begin(), {0, 0});

    len = dirty.size();

    // calculating LIMIT_MASK
    limit = (1<<len) - 1;

    // precalculating distances from all
    // dirty tiles to each cell in the grid
```

```
    for (int i = 0;i<len;i++)
       getDist(i);
}

int main(int argc, char const *argv[])
{
    // Test case #1:
    //     .....*.
    //     ...#...
    //     .*.#.*.
    //     .......

char A[4][7] = {    {'.', '.', '.', '.', '.', '*', '.'},
                    {'.', '.', '.', '#', '.', '.', '.'},
                    {'.', '*', '.', '#', '.', '*', '.'},
                    {'.', '.', '.', '.', '.', '.', '.'}
                };

    r = 4; c = 7;

    cout << "The given grid : " << endl;

    for (int i = 0;i<r;i++)
    {
        for (int j = 0;j<c;j++)
        {
            cout << A[i][j] << " ";
            arr[i][j] = A[i][j];
        }
        cout << endl;
    }

    // - initializitiation
    // - precalculations
    init();

    int ans = solve(0, 1);

    cout << "Minimum distance for the given grid : ";
    cout << ans << endl;


    // Test Case #2
    //     ...#...
    //     ...#.*.
    //     ...#...
    //     .*.#.*.
    //     ...#...

    char Arr[5][7] = {  {'.', '.', '.', '#', '.', '.', '.'},
                        {'.', '.', '.', '#', '.', '*', '.'},
                        {'.', '.', '.', '#', '.', '.', '.'},
                        {'.', '*', '.', '#', '.', '*', '.'},
                        {'.', '.', '.', '#', '.', '.', '.'}
                };

    r = 5; c = 7;

    cout << "The given grid : " << endl;

    for (int i = 0;i<r;i++)
    {
        for (int j = 0;j<c;j++)
        {
            cout << Arr[i][j] << " ";
            arr[i][j] = Arr[i][j];
        }
        cout << endl;
    }

    // - initializitiation
    // - precalculations
    init();
```

```
    ans = solve(0, 1);
    cout << "Minimum distance for the given grid : ";
    if (ans >= INF)
        cout << "not possible" << endl;
    else
        cout << ans << endl;

    return 0;
}
```

Run on IDE

Output:

```
The given grid :
. . . . . * .
. . . # . . .
. * . # . * .
. . . . . . .
Minimum distance for the given grid : 16
The given grid :
. . . # . . .
. . . # . * .
. . . # . . .
. * . # . * .
. . . # . . .
Minimum distance for the given grid : not possible
```

Note:

We have used the initial state to be **dp[0][1]** because we have pushed the start location at the first position in the container of houses. Hence, our Bit Mask will be 1 as the 0^{th} bit is set i.e we have visited the starting location for our trip.

Time Complexity:

Consider the number of houses to be **n**. So, there are **n * (2^n)** states and at every state, we are looping over n houses to transit over to next state and because of memoization we are doing this looping transition only once for each state. Therefore, our Time Complexity is **O(n^2 * 2^n)**.

Recommended:

- http://www.spoj.com/problems/CLEANRBT/
- https://www.youtube.com/watch?v=-JjA4BLQyqE

This article is contributed by **Nitish Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Practice Tags : Dynamic Programming Bit Magic

Article Tags : Bit Magic Dynamic Programming

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content. Login to Improve this Article

Recommended Posts:

Bitmasking and Dynamic Programming | Set 1 (Count ways to assign unique cap to every person)
How to solve a Dynamic Programming Problem ?
Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)
Find the Longest Increasing Subsequence in Circular manner
Digit DP | Introduction
Count pairs with Bitwise XOR as ODD number
Sum of XOR of sum of all pairs in an array
Count pairs with Bitwise AND as ODD number
Count pairs with Bitwise OR as Even number
Check if bits in range L to R of two numbers are complement of each other or not

(Login to Rate)

4.6 Average Difficulty : **4.6/5.0**
Based on **10** vote(s)

Basic Easy Medium Hard Expert

Add to TODO List
Mark as DONE

Feedback Add Notes Improve Article

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments Share this post!