USACO JAN13 Problem 'island2' Analysis

by Neal Wu

This problem is somewhat complex, and the algorithm to solve it uses the following three major steps:

1. Flood fill to find the islands. (Both depth-first search, DFS, and breadth-first search, BFS, will work fine here.)

2. Flood fill to find the distances between all pairs of islands. (BFS should be considerably faster than DFS here.)

3. After finding the distances between all pairs of islands, find the minimum distance needed to traverse all islands. (This is a well-known problem that is also known as the Traveling Salesman Problem.) The simplest solution to this would be to try all possible orderings of the islands, but this is far too slow for N = 15. To speed up the algorithm, we can use dynamic programming, with our state consisting of our current location and the subset of islands that we have visited, and the value as the current total distance. This algorithm can be implemented either recursively or iteratively for a complexity of $O(N^2 \times 2^N)$.

The following is a solution using this idea:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

FILE *fout = fopen ("island2.out", "w");
FILE *fin = fopen ("island2.in", "r");

const int INF = 1000000000;
const int dr [] = {-1, 0, 0, 1};
const int dc [] = {0, -1, 1, 0};

const int MAXN = 16;
const int MAXG = 55;
const int MAXS = 70000;

struct loc
{
    int row, col, dis;

    loc (int r, int c, int d)
    {
        row = r, col = c, dis = d;
    }
};

int N, R, C;

char grid [MAXG][MAXG];
int group [MAXG][MAXG];

int tdist [MAXG][MAXG];
int dist [MAXN][MAXN];
```

```
queue <loc> q;

int best [MAXN][MAXS];
int masks [MAXS];
int msize;

int ans;

inline bool comp (int a, int b)
{
    return __builtin_popcount (a) < __builtin_popcount (b);
}

void floodfill (int r, int c)
{
    group [r][c] = N;

    for (int k = 0; k < 4; k++)
    {
        int nr = r + dr [k];
        int nc = c + dc [k];

        if (grid [nr][nc] == 'X' && group [nr][nc] == -1)

            floodfill (nr, nc);
    }
}

void solveislands ()
{
    memset (group, -1, sizeof (group));

    N = 0;
    for (int i = 1; i <= R; i++)
        for (int j = 1; j <= C; j++)
            if (grid [i][j] == 'X' && group [i][j] == -1)
            {
                floodfill (i, j);
                N++;
            }
}

void solvedist ()
{
    memset (dist, 63, sizeof (dist));

    for (int i = 0; i < N; i++)
    {
        int ir = -1, ic = -1;
        bool found = false;

        for (int r = 1; r <= R && !found; r++)
            for (int c = 1; c <= C && !found; c++)
                if (group [r][c] == i)
                {
                    ir = r, ic = c;
                    found = true;
                }
```

```
        memset (tdist, 63, sizeof (tdist));

        q.push (loc (ir, ic, 0));
        tdist [ir][ic] = 0;


        while (!q.empty ())
        {
            loc top = q.front ();
            q.pop ();


            if (group [top.row][top.col] != -1)
            {
                if (top.dis < dist [i][group [top.row][top.col]])
                    dist [i][group [top.row][top.col]] = top.dis;
            }

            for (int k = 0; k < 4; k++)
            {
                int nr = top.row + dr [k];
                int nc = top.col + dc [k];

                if (grid [nr][nc] == 'X')
                {
                    if (top.dis < tdist [nr][nc])
                    {
                        tdist [nr][nc] = top.dis;
                        q.push (loc (nr, nc, top.dis));
                    }
                }
                else if (grid [nr][nc] == 'S')
                {
                    if (top.dis + 1 < tdist [nr][nc])
                    {
                        tdist [nr][nc] = top.dis + 1;
                        q.push (loc (nr, nc, top.dis + 1));
                    }
                }
            }
        }
    }
}

void solvetsp ()
{
    memset (best, 63, sizeof (best));

    for (int i = 0; i < N; i++)
        best [i][1 << i] = 0;


    msize = 0;

    for (int m = 1; m < (1 << N); m++)
        masks [msize++] = m;

    sort (masks, masks + msize, comp);
```

```
    for (int ind = 0; ind < msize; ind++)
    {
        int m = masks [ind];

        for (int i = 0; i < N; i++)
            if (best [i][m] < INF)
            {
                for (int j = 0; j < N; j++)
                    if (best [i][m] + dist [i][j] < best [j][m | (1 << j)])
                        best [j][m | (1 << j)] = best [i][m] + dist [i][j];
            }
    }


    ans = INF;

    for (int i = 0; i < N; i++)
        if (best [i][(1 << N) - 1] < ans)
            ans = best [i][(1 << N) - 1];
}

int main ()
{
    memset (grid, '.', sizeof (grid));
    fscanf (fin, "%d %d", &R, &C);

    for (int i = 1; i <= R; i++)
        fscanf (fin, "%s", &grid [i][1]);

    solveislands ();
    solvedist ();
    solvetsp ();

    fprintf (fout, "%d\n", (ans < INF) ? ans : -1);
}
```

Fatih Gelgi's additions:

In the third step, one can also use A* kind of method for searching instead of dynamic programming since N is small (N=15).

```
// TSP using A*
#include <iostream>
#include <fstream>
#include <cstdlib>

#define MAXN 20

using namespace std;

const int dir[4][2]={{1,0},{-1,0},{0,1},{0,-1}};

int r, c, islands[50][50], n;
int graph[MAXN][MAXN], mdist[MAXN], mtotal, mindist=10000000;
string mat[50];

// dfs
void extract_island(int k,int y,int x)
{
```

```
	islands[y][x]=k;
	for (int i=0; i<4; i++)
	{
		int ny=y+dir[i][0],nx=x+dir[i][1];
		if (ny>=0 && ny<r && nx>=0 && nx<c && mat[ny][nx]=='X'
			&& !islands[ny][nx])
			extract_island(k,ny,nx);
	}
}

// extract neighbors of node v
void extract_graph(int v)
{
	int mark[50][50]={};

	for (int k=1,flag=1; flag; k++)
	{
		flag=0;
		for (int i=0; i<r; i++)
			for (int j=0; j<c; j++)
				if (islands[i][j]==v || (mat[i][j]=='S' && mark[i][j]
					&& mark[i][j]==k-1))
					for (int l=0; l<4; l++)
					{
						int y=i+dir[l][0],x=j+dir[l][1];
						if (y>=0 && y<r && x>=0 && x<c)
							if (mat[y][x]=='S' && !mark[y][x])
								mark[y][x]=k,flag=1;
							else
							{
								int u=islands[y][x];
								if (u && u!=v && !graph[v][u])
									graph[v][u]=graph[u][v]=k-1;
							}
					}
	}
}

int visit[MAXN];
// TSP with A* {x:node,c:remaining non-visited,d:current distance,r:h*(x)}
void tsp(int x,int c,int d,int r)
{
	if (c==1)
	{
		mindist=d;
		return;
	}

	visit[x]=1;
	for (int i=1; i<=n; i++)
		if (graph[x][i] && !visit[i] && d+graph[x][i]+(r-mdist[i])<mindist)
			tsp(i,c-1,d+graph[x][i],r-mdist[i]);
	visit[x]=0;
}

int main()
{
	ifstream fin("island2.in");
	fin >> r >> c;
	for (int i=0; i<r; i++)
```

```
            fin >> mat[i];
      fin.close();

      // extract islands
      for (int i=0; i<r; i++)
            for (int j=0; j<c; j++)
                  if (mat[i][j]=='X' && !islands[i][j])
                        extract_island(++n,i,j);

      // extract graph
      for (int i=1; i<=n; i++)
            extract_graph(i);

      // floyd-warshall
      for (int k=1; k<=n; k++)
            for (int i=1; i<=n; i++)
                  if (graph[i][k])
                        for (int j=1; j<=n; j++)
                              if (graph[k][j])
                                    if (!graph[i][j] ||
graph[i][j]>graph[i][k]+graph[k][j])
                                          graph[i][j]=graph[i][k]+graph[k][j];

      // get minimum distances of each node for h* function
      for (int i=1; i<=n; i++)
            for (int j=1; j<=n; j++)
                  if (!mdist[i] || mdist[i]>graph[i][j])
                        mdist[i]=graph[i][j];
      for (int i=1; i<=n; i++)
            mtotal+=mdist[i];

      // search tsp starting from each node
      for (int i=1; i<=n; i++)
            tsp(i,n,0,mtotal-mdist[i]);

      ofstream fout("island2.out");
      fout << mindist << "\n";
      fout.close();
}
```