By gladius - Topcoder Member
Discuss this article in the forums

Basic methods for searching graphs

Introduction

So far we have learned how to represent our graph in memory, but now we need to start doing something with this information. There are two methods for searching graphs that are extremely prevalent, and will form the foundations for more advanced algorithms later on. These two methods are the Depth First Search and the Breadth First Search.

We will begin with the depth first search method, which will utilize a stack. This stack can either by represented explicitly (by a stack data-type in our language) or implicitly when using recursive functions.

Stack

A stack is one of the simplest data structures available. There are four main operations on a stack:

1. Push – Adds an element to the top of the stack
2. Pop – Removes the top element from the stack
3. Top – Returns the top element on the stack
4. Empty – Tests if the stack is empty or not

In C++, this is done with the STL class stack:

```
#include <stack>
std::stack<int> myStack;
```

In Java, we use the Stack class:

```
import java.util.*;
Stack stack = new Stack();
```

```
Stack stack = new Stack();
```

In C#, we use Stack class:

```
using System.Collections;
Stack stack = new Stack();
```

Depth First Search

Now to solve an actual problem using our search! The depth first search is well geared towards problems where we want to find any solution to the problem (not necessarily the shortest path), or to visit all of the nodes in the graph. A recent topcoder problem was a classic application of the depth first search, the flood-fill. The flood-fill operation will be familiar to anyone who has used a graphic painting application. The concept is to fill a bounded region with a single color, without leaking outside the boundaries.

This concept maps extremely well to a Depth First search. The basic concept is to visit a node, then push all of the nodes to be visited onto the stack. To find the next node to visit we simply pop a node of the stack, and then push all the nodes connected to that one onto the stack as well and we continue doing this until all nodes are visited. It is a key property of the Depth First search that we not visit the same node more than once, otherwise it is quite possible that we will recurse infinitely. We do this by marking the node as we visit it.

So the basic structure will look something like this:

```
dfs(node start) {
stack<node> s;
s.push(start);
while (s.empty() == false) {
top = s.top();
s.pop();

if (top is not marked as visited) {
check for termination condition (have we reached the node we want to?)

mark top as visited;
add all of top's neighbors to the stack.
}
}
}
```

Alternatively we can define the function recursively as follows:

```
dfs(node current) {
mark current as visited;
visit all of current's unvisited neighbors by calling dfs(neighbor)
}
```

The problem we will be discussing is grafixMask, a Division 1 500 point problem from SRM 211. This problem essentially asks us to find the number of discrete regions in a grid that has been filled in with some values already. Dealing with grids as graphs is a very powerful technique, and in this case makes the problem quite easy.

We will define a graph where each node has 4 connections, one each to the node above, left, right and below. However, we can represent these connections implicitly within the grid, we need not build out any new data structures. The structure we will use to represent the grid in grafixMask is a two dimensional array of booleans, where regions that we have already determined to be filled in will be set to true, and regions that are unfilled are set to false.

To set up this array given the data from the problem is very simple, and looks something like this:

```
bool fill[600][400];
initialize fills to false;

foreach rectangle in Rectangles
set from (rectangle.left, rectangle.top) to (rectangle.right, retangle.bottom) to true
```

Now we have an initialized connectivity grid. When we want to move from grid position (x, y) we can either move up, down, left or right. When we want to move up for example, we simply check the grid position in (x, y-1) to see if it is true or false. If the grid position is false, we can move there, if it is true, we cannot.

Now we need to determine the area of each region that is left. We don't want to count regions twice, or pixels twice either, so what we will do is set fill[x][y] to true when we visit the node at (x, y). This will allow us to perform a Depth-First search to visit all of the nodes in a connected region and never visit any node twice, which is exactly what the problem wants us to do! So our loop after setting everything up will be:

```
int[] result;

for x = 0 to 599
for y = 0 to 399
```

```
if (fill[x][y] == false)
result.addToBack(doFill(x,y));
```

All this code does is check if we have not already filled in the position at (x, y) and then calls doFill() to fill in that region. At this point we have a choice, we can define doFill recursively (which is usually the quickest and easiest way to do a depth first search), or we can define it explicitly using the built in stack classes. I will cover the recursive method first, but we will soon see for this problem there are some serious issues with the recursive method.

We will now define doFill to return the size of the connected area and the start position of the area:

```
int doFill(int x, int y) {
// Check to ensure that we are within the bounds of the grid, if not, return 0
if (x < 0 || x >= 600) return 0;
// Similar check for y
if (y < 0 || y >= 400) return 0;
// Check that we haven't already visited this position, as we don't want to count it twice
if (fill[x][y]) return 0;

// Record that we have visited this node
fill[x][y] = true;

// Now we know that we have at least one empty square, then we will recursively
attempt to
// visit every node adjacent to this node, and add those results together to return.
return 1 + doFill(x - 1, y) + doFill(x + 1, y) + doFill(x, y + 1) + doFill(x, y - 1);
}
```

This solution should work fine, however there is a limitation due to the architecture of computer programs. Unfortunately, the memory for the implicit stack, which is what we are using for the recursion above is more limited than the general heap memory. In this instance, we will probably overflow the maximum size of our stack due to the way the recursion works, so we will next discuss the explicit method of solving this problem.

Sidenote:

Stack memory is used whenever you call a function; the variables to the function are pushed onto the stack by the compiler for you. When using a recursive function, the variables keep getting pushed on until the function returns. Also any variables the compiler needs to save between function calls must be pushed onto the stack as well.

This makes it somewhat difficult to predict if you will run into stack difficulties. I

recommend using the explicit Depth First search for every situation you are at least somewhat concerned about recursion depth.

In this problem we may recurse a maximum of 600 * 400 times (consider the empty grid initially, and what the depth first search will do, it will first visit 0,0 then 1,0, then 2,0, then 3,0 … until 599, 0. Then it will go to 599, 1 then 598, 1, then 597, 1, etc. until it reaches 599, 399. This will push 600 * 400 * 2 integers onto the stack in the best case, but depending on what your compiler does it may in fact be more information. Since an integer takes up 4 bytes we will be pushing 1,920,000 bytes of memory onto the stack, which is a good sign we may run into trouble.

We can use the same function definition, and the structure of the function will be quite similar, just we won't use any recursion any more:

```
class node { int x, y; }

int doFill(int x, int y) {
int result = 0;

// Declare our stack of nodes, and push our starting node onto the stack
stack<node> s;
s.push(node(x, y));

while (s.empty() == false) {
node top = s.top();
s.pop();

// Check to ensure that we are within the bounds of the grid, if not, continue
if (top.x < 0 || top.x >= 600) continue;
// Similar check for y
if (top.y < 0 || top.y >= 400) continue;
// Check that we haven't already visited this position, as we don't want to count it twice
if (fill[top.x][top.y]) continue;

fill[top.x][top.y] = true; // Record that we have visited this node

// We have found this node to be empty, and part
// of this connected area, so add 1 to the result
result++;
```

```
// Now we know that we have at least one empty square, then we will attempt to
// visit every node adjacent to this node.
s.push(node(top.x + 1, top.y));
s.push(node(top.x - 1, top.y));
s.push(node(top.x, top.y + 1));
s.push(node(top.x, top.y - 1));
}
return result;
}
```

As you can see, this function has a bit more overhead to manage the stack structure explicitly, but the advantage is that we can use the entire memory space available to our program and in this case, it is necessary to use that much information. However, the structure is quite similar and if you compare the two implementations they are almost exactly equivalent.

Congratulations, we have solved our first question using a depth first search! Now we will move onto the depth-first searches close cousin the Breadth First search.

If you want to practice some DFS based problems, some good ones to look at are:

TCCC 03 Quarterfinals – Marketing – Div 1 500

TCCC 03 Semifinals Room 4 – Circuits - Div 1 275

Queue

A queue is a simple extension of the stack data type. Whereas the stack is a FILO (first-in last-out) data structure the queue is a FIFO (first-in first-out) data structure. What this means is the first thing that you add to a queue will be the first thing that you get when you perform a pop().

There are four main operations on a queue:

1. Push – Adds an element to the back of the queue
2. Pop – Removes the front element from the queue
3. Front – Returns the front element on the queue
4. Empty – Tests if the queue is empty or not

In C++, this is done with the STL class queue:

```
#include <queue>
queue<int> myQueue;
```

In Java, we unfortunately don't have a Queue class, so we will approximate it with the LinkedList class. The operations on a linked list map well to a queue (and in fact, sometimes queues are

implemented as linked lists), so this will not be too difficult.

The operations map to the LinkedList class as follows:

1. Push – boolean LinkedList.add(Object o)
2. Pop – Object LinkedList.removeFirst()
3. Front – Object LinkedList.getFirst()
4. Empty – int LinkedList.size()

```
import java.util.*;
LinkedList myQueue = new LinkedList();
```

In C#, we use Queue class:

The operations map to the Queue class as follows:

1. Push – void Queue.Enqueue(Object o)
2. Pop – Object Queue.Dequeue()
3. Front – Object Queue.Peek()
4. Empty – int Queue.Count

```
using System.Collections;
Queue myQueue = new Queue();
```

Breadth First Search

The Breadth First search is an extremely useful searching technique. It differs from the depth-first search in that it uses a queue to perform the search, so the order in which the nodes are visited is quite different. It has the extremely useful property that if all of the edges in a graph are unweighted (or the same weight) then the first time a node is visited is the shortest path to that node from the source node. You can verify this by thinking about what using a queue means to the search order. When we visit a node and add all the neighbors into the queue, then pop the next thing off of the queue, we will get the neighbors of the first node as the first elements in the queue. This comes about naturally from the FIFO property of the queue and ends up being an extremely useful property. One thing that we have to be careful about in a Breadth First search is that we do not want to visit the same node twice, or we will lose the property that when a node is visited it is the quickest path to that node from the source.

The basic structure of a breadth first search will look this:

```
void bfs(node start) {
queue<node> s;
```

```
s.push(start);
mark start as visited
while (s.empty() == false) {
top = s.front();
s.pop();

check for termination condition (have we reached the node we want to?)

add all of top's unvisited neighbors to the queue
mark all of top's unvisited neighbors as visited
}
}
```

Notice the similarities between this and a depth-first search, we only differ in the data structure used and we mark a vertex visited as we push it into the queue, not as we pop it.

The problem we will be discussing in relation to the Breadth First search is a bit harder than the previous example, as we are dealing with a slightly more complicated search space. The problem is the 1000 from Division 1 in SRM 156, Pathfinding. Once again we will be dealing in a grid-based problem, so we can represent the graph structure implicitly within the grid.

A quick summary of the problem is that we want to exchange the positions of two players on a grid. There are impassable spaces represented by 'X' and spaces that we can walk in represented by ' '. Since we have two players our node structure becomes a bit more complicated, we have to represent the positions of person A and person B. Also, we won't be able to simply use our array to represent visited positions any more, we will have an auxiliary data structure to do that. Also, we are allowed to make diagonal movements in this problem, so we now have 9 choices, we can move in one of 8 directions or simply stay in the same position. Another little trick that we have to watch for is that the players can not just swap positions in a single turn, so we have to do a little bit of validity checking on the resulting state.

First, we set up the node structure and visited array:

```
class node {
int player1X, player1Y, player2X, player2Y;
int steps; // The current number of steps we have taken to reach this step
}

bool visited[20][20][20][20];
```

Here a node is represented as the (x,y) positions of player 1 and player 2. It also has the current steps that we have taken to reach the current state, we need this because the problem asks us what the minimum number of steps to switch the two players will be. We are guaranteed by the

what the minimum number of steps to switch the two players will be. We are guaranteed by the

properties of the Breadth First search that the first time we visit the end node, it will be as quickly as possible (as all of our edge costs are 1).

The visited array is simply a direct representation of our node in array form, with the first dimension being player1X, second player1Y, etc. Note that we don't need to keep track of steps in the visited array.

Now that we have our basic structure set up, we can solve the problem (note that this code is not compilable):

```
void pushToQueue(queue<node> q, node v) {
if (visited[v.player1X][v.player1Y][v.player2X][v.player2Y]) return;
q.push(v);
visited[v.player1X][v.player1Y][v.player2X][v.player2Y] = true;
}

int minTurns(String[] board) {
int width = board[0].length;
int height = board.length;

node start;
// Find the initial position of A and B, and save them in start.

queue<node> q;
pushToQueue(q, start)
while (q.empty() == false) {
node top = q.front();
q.pop();

// Check if player 1 or player 2 is out of bounds, or on an X square, if so continue
// Check if player 1 or player 2 is on top of each other, if so continue

// Check if the current positions of A and B are the opposite of what they were in start.
// If they are we have exchanged positions and are finished!
if (top.player1X == start.player2X && top.player1Y == start.player2Y &&
top.player2X == start.player1X && top.player2Y == start.player1Y)
return top.steps;

// Now we need to generate all of the transitions between nodes, we can do this quite
easily using some
```

```
// nested for loops, one for each direction that it is possible for one player to move.
Since we need

// to generate the following deltas: (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1)
// we can use a for loop from -1 to 1 to do exactly that.
for (int player1XDelta = -1; player1XDelta <= 1; player1XDelta++) {
for (int player1YDelta = -1; player1YDelta <= 1; player1YDelta++) {
for (int player2XDelta = -1; player2XDelta <= 1; player2XDelta++) {
for (int player2YDelta = -1; player2YDelta <= 1; player2YDelta++) {
// Careful though! We have to make sure that player 1 and 2 did not swap positions on
this turn
if (top.player1X == top.player2X + player2XDelta && top.player1Y == top.player2Y +
player2YDelta &&
top.player2X == top.player1X + player1XDelta && top.player2Y == top.player1Y +
player1YDelta)
continue;

// Add the new node into the queue
pushToQueue(q, node(top.player1X + player1XDelta, top.player1Y + player1YDelta,
top.player2X + player2XDelta, top.player2Y + player2YDelta,
top.steps + 1));
}
}
}
}
}
// It is not possible to exchange positions, so
// we return -1. This is because we have explored
// all the states possible from the starting state,
// and haven't returned an answer yet.
return -1;
}
```

This ended up being quite a bit more complicated than the basic Breadth First search implementation, but you can still see all of the basic elements in the code. Now, if you want to practice more problems where Breadth First search is applicable, try these:

Inviational 02 Semifinal Room 2 – Div 1 500 – Escape

…continue to Section 3