10/19/2012

Lab – 8 INFO I 500 / CSCI 609

Depth First search

This is a popular technique for systematically traversing vertices of a graph. The method starts traversing the graph from a given vertex V_0 which is the first vertex need to be visited. The next vertex to be visited is an unvisited vertex adjacent to the V_0. If V_0 has number of unvisited adjacent vertices then any one of them may be selected for visiting next. Once a vertex is visited it is marked as "visited" to make it easier to find non-visited vertices. Thus from any vertex V_1 a new adjacent unvisited vertex V_2 is explored. Next another new vertex adjacent to V_2 is explored without traversing along the other edges incident on V_1.When a Vertex V_3 does not have unvisited adjacent vertex V_4 the depth first search is initiated from V_4. The process continues until all adjacent vertices if the starting vertex has been visited.

This algorithm executes a depth-first search on a graph G beginning with a starting vertex A.

Step 1. Initialize all vertices to the ready state (STATUS = I)

Step 2. Push the starting vertex A onto STACK and change the status of A to the waiting state (STATUS = 2).

Step 3. Repeat Steps 4 and 5 until STACK is empty.

Step 4. Pop the top vertex N of STACK. Process N, and set STATUS (N) = 3; the processed state

Step 5. Examine each neighbor J of N.

(a) If STATUS (J)= 1 (ready slate), push Jonto STACK and reset STATUS (J) = 2 (waiting state).

(b) If STATUS (J)= 2 (waiting state), delete the previous J from the STACK and push the current J onto STACK

(c) If STATUS (J)= 3 (processed state), ignore the vertex J.
[End of Step 3 loop.]

Step 6. Exit.

An example of Depth first Search is given in
http://www.cs.umd.edu/class/sum2005/cmsc451/dfsimplementation.pdf

Exercise: Imagine that you are a travel agent and a rather quarrelsome customer wants you to book a flight from New York to Los Angeles with XYZ Airlines. You try to tell the customer that XYZ does not have a direct flight from New York to Los Angeles, but the customer insists that XYZ is the only airline that he will fly. XYZ's scheduled flights are as follows:

New York to Chicago	1000 miles
Chicago to Denver	1000 miles
New York to Toronto	800 miles
New York to Denver	1,900 miles
Toronto to Calgary	1,500 miles
Toronto to Los Angeles	1,800 miles
Toronto to Chicago	1000 miles
Denver to Urbana	1,000 miles
Denver to Houston	1,500 miles
Houston to Los Angeles	1000 miles
Denver to Los Angeles	1,000 miles

You quickly see that there is a way to fly from New York to Los Angeles by using XYZ if you book connecting flights, and you book the fellow his flights.The diagram shown in Figure -1.
Write a C program to solve the problem for yourself using depth first search. The depth search won't give you an optimal solution**(BFS will give)** but it will help you find a path or route for the travel.

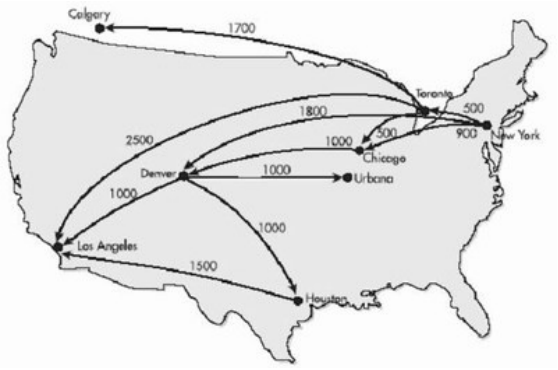

Figure -1

To make things easier to understand, we can redraw this graph in a treelike fashion, as shown in Figure 2. Refer to this version for the rest of this chapter. The goal, Los Angeles, is circled. Notice also that various cities appear more than once to simplify the construction of the graph. Thus, the treelike representation does not depict a binary tree. It is simply a visual convenience.Now we are ready to develop the various search programs that will find paths from New York to Los Angeles.

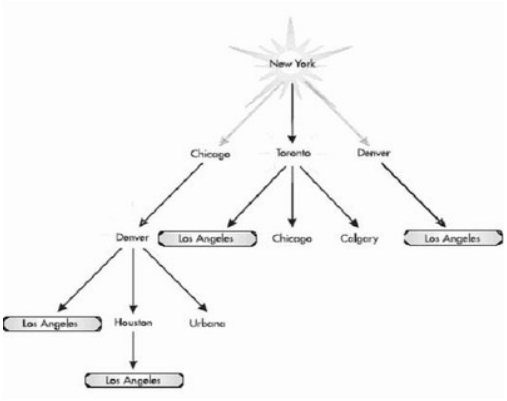

Figure 2

I have given the part of the C code in **dfs.c** complete the sections of the code like stack routines **pop and push is_flight() and find()** . The **route()** function prints the **distance between the two places**.In the code the structure of the flight database is given along with **insert_flight() and setup()** which initializes all the flight information, fl_pos is a global variable which holds index of the last item in the database.The cities have **skip** field which is set to 1 are not valid connections. If a connection is found **skip** marked as active this control backtracking from dead ends. **Also declare the stack data structure by your own.**

Note: Backtracking is accomplished through the use of recursive routines and a backtrack stack **bt_stack**. Almost all backtracking situations are stack like in operation— that is, they are first-in, last-out(FILO). As a path is explored, nodes are pushed onto the stack as they are encountered. At each dead end, **the last node is popped off the stack and a new path, from that point, is tried**. This process continues until either the goal is reached or all paths have been exhausted. The functions **push( )** and **pop( )**, which manage the backtrack stack.

Other than this you may design the code in your own way and implement.
Below shows the picture of code execution and the ouput.