CS2303 In Class Exercises

Jan 28, 2020

January 12, 2020

Abstract

Today's goal is that you can search through the nodes of a graph, using a linked list as a (FIFO) queue, or as a (LIFO) stack, and that you recognize an implementation is extracting a tree from the graph as the search is carried out. You will see how to carry out both breadth-first search vs. depth first search.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Searching a Graph

What is meant by search a graph is visiting nodes in a systematic fashion, until a particular item is found, or the conclusion that the item is not present in the graph is reached.

There are systematic ways of searching trees; we shall look at two, depth-first search and breadth-first search.

We shall extract a tree from the graph as we search the graph, such that we perform only a slight modification of the tree search algorithm to apply it to a more general graph.

If the items stored in a graph have some organization, we might be able to conclude searching before visiting every node. When this notion is applied to a tree, and we are able to exclude a subtree from the search, this exclusion process is called pruning. Pruning is very important for optimizations carried out by searching trees.

Our search algorithm makes use of a linked list. There are two specializations of linked list we shall examine, first-in-first-out (like a line at the cashier in the food court) or last-in-first-out (like a stack of washable cafeteria trays).

2 Linked List as FIFO

In general a linked list allows items to be added to the list at any point. One merely traverses the list from the head to the desired location, sets appropriate values into next and previous fields, and the item is in place. Likewise, an element at any location in the list can be removed.

In a FIFO, only the two ends of the list support changes. The head-end supports removals, and the tail-end supports additions.

Thus we can imagine a FIFO, at first empty, having an item (labelled 1) added, then having an item labelled 2 added. The first removal will remove the item labelled 1.

Example:
Alex queues up at the cashier, followed by Andy, while Sandy queues up next, the cashier serves Alex first.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Describe an activity with queueing, such as waiting at a traffic light, in terms of who arrives first and who gets to leave first.

waiting in line for popcorn first person in line gets the goods first

2.1 Considerations of Return Value for Dequeue of FIFO

We we remove a value from a queue, we usually return the item that was on the queue. This item will have a datatype to which Payload has been made equivalent, so we can say, it can be of datatype Payload. In a FIFO queue there is more. The head-end of the queue had an address before the data item was removed, but the data item being removed was attached to the head. So, that infrastructure element should be released into the storage pool, and the address of the head should be moved on to the next element. Moreover, that next element's address should be returned. The two fields, the data item being removed and the new address for the head of the FIFO can be combined into a typedef struct.

Example:

```
backFromDQFIFO* dequeueFIFO(LLNode* lp)
{
    backFromDQFIFO* fp = (backFromDQFIFO*) malloc(sizeof(backFromDQFIFO));
    if(lp->next == (struct LLNode*)0)
    {
      //list of length 1 or 0
        fp->newQHead= lp;
```

```
    }
    else
    {
        fp->newQHead= (LLNode*) lp->next;
    }
    fp->mp= lp->payP;//all return values are set
    if(lp->next != (struct LLNode*)0)
    {
       //length list is >1
       free(lp);
    }
    return fp;
}
```

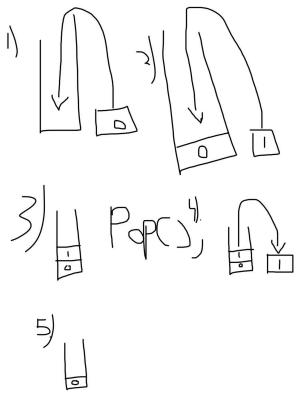
3 Linked List as LIFO

By contrast, there is a notion of last-in first-out. This is used a great deal in computer science. Unlike in the LIFO, we do not envision that adding to the queue can occur simultaneously with removal. Instead we serialize adding and removing from the queue. Thus, when some newly washed cafeteria trays are being placed into the stack, people who want to remove trays wait. Imagine three washed trays are added to the stack, first tray 1 on the top, then tray 2 on top of it, then tray 3 on top of that. Then Alex removes the top tray (3), next Andy removes the newly revealed top tray, which is tray 2, and lastly Sandy obtains a tray, which is tray 1. Note that the order is reversed.

Example:
The stack is used in computation. Suppose the function main calls the function tests(). Information needed for the function to execute, and to return (such as the address of the location to return to) are put on the stack. The tests() calls a particular test, say test1(). Information is put on the stack so that test1 can return to test. If test1() invokes a production function, information is put on the stack. Then the production function takes that most recent information off the stack so as to return to test1(). The test1 takes the newly revealed information off the stack and returns to tests(). Then tests() takes the newly revealed information off the stack so as to return to main().

To use a linked list in a LIFO fashion, we add new items by traversing to the end of the list, and we also choose the item to remove by traversing to the end of the list.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)



1. Draw a set of diagrams that show a LIFO (stack) storing more and more data, then, being emptied out, one item at a time.

4 Extracting a Tree From a Graph

When every node found in the graph is present in the extracted tree, we call that a spanning tree. Here, because we are searching the whole graph, not performing any pruning, we will use the term tree to mean spanning tree.

Suppose we have a graph, and we wish to extract a spanning tree. All the nodes will still be present, but some of the edges might be removed. We know that some of the edges might be removed because a general simple graph can have $n(n-1)$ edges for n nodes. We also know a tree has $n-1$ edges for n nodes. (Incidentally, a graph may have fewer nodes than a tree, but such a graph is not a connected graph, and we are limiting our discussion to connected graphs.) A general graph may have components that are not connected to each other. In this case, the overall search is composed of one search per component.

So for our purpose we might have to ignore some edges. In particular, we shall ignore edges that lead from a node we are exploring to a node we have already explored.

To accomplish this we include a field in the node data structure indicating whether the node has been explored already or not. We initialize this searched boolean to false, and set it to true when the node has been explored. When the algorithm finds a node, this boolean is checked. If the boolean is true, the node is not further used.

5 Search Algorithm

The search algorithm begins at a designated node. That node is marked as having been explored. Any edge impinging on this node is followed to obtain the node on the other end of the edge, which is called a child node. One-by-one, as each child node is found, it is checked for whether it has already been explored. Nodes that have not already been explored are placed in the linked list. When all child nodes of the designated node have been found, checked, and possibly added to the linked list, it is time to obtain a new node. The new node is obtained from the linked list. The process repeats, i.e., this node may have children and if so, they are found, checked and possibly added to the linked list. When all child nodes of this node have been processed, a new node is sought from the linked list. When the linked list is empty, the search is over.

Example:

```
LLNode* searchQ = makeEmptyLinkedList();
//we'll start searching with room 0
bool done = false;
int searchedRooms = 0;
float foundTreasure = 0.0;
Room* roomBeingSearchedP = theRoomPs[0];
//we set its searched field to true, and take its treasure
roomBeingSearchedP->searched = true;
```

```
puts("starting search"); fflush(stdout);
while(!done)
{   //here we want to find all of the rooms adjacent to the roomBeingSearched,
    //so we look in the adj matrix
    for(int col = 0; col<nrooms; col++)
    {
        //we check whether this room is neighboring
        if(getEdge(adjMP,roomBeingSearchedP->roomNumber, col)==1)
        {
            //if so, we check whether room has been searched
            if(!(theRoomPs[col]->searched))
            {//if it hasn't been searched
             //we set it to searched
               theRoomPs[col]->searched=true;
               {//we enqueue it for search
                  savePayload(searchQ, theRoomPs[col]);
               }//check about search limits
            }//room can still be searched
         }//room is a neighbor
     }//scan for all possible rooms to search from this room
     //time to get the next room
     if(isEmpty(searchQ))
     {
         done=true;
     }
     if(!done)
     {
           //Invoking dequeue
           roomBeingSearchedP = dequeueLIFO(searchQ);
      }
}//while search is not done
//search is done
```

6 Breadth-first Search

Imagine a tree, the top has a single node that is the designated node for the search. This node is called the "root". Every edge impinging on the designated node reaches a child node. Imagine each of those child nodes arranged on a shared horizontal line below the root. Then, for each of these child nodes, one-at-a-time, treat it as a root. A lower horizontal line is shared by the children of these child nodes. These node, "grandchildren" of the root node, are two edges away from the root.

Now we can describe the order in which we search the tree. We start with the root. Then we search every node on the highest horizontal line. Then we

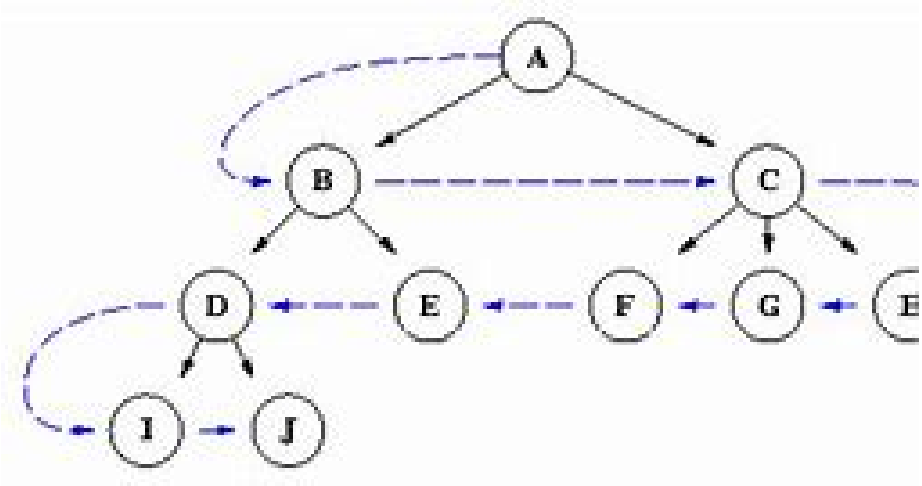


Figure 1: Tree diagram showing breadth first search order of visiting nodes

search every node on the next lower horizontal line. Because the tree is arranged to show breadth, and we search all the nodes across that breadth before moving to a lower horizontal line, this is called breadth-first search.

6.1 Using Breadth-first Search

Recall we described two restrictions on linked lists, one was FIFO, the other LIFO. Recall that we put nodes into a linked list, and took nodes from a linked list, in the search algorithm. When the FIFO queue is used as the linked list in the search algorithm, a breadth-first search is obtained.

7 Depth-first Search

By contrast there is depth-first search. Depth first search does not follow the horizontal line of the tree arrangement. Instead, the search order moves from the root towards an ever more distant child.

7.1 Using Depth-first Search

When the linked list used in our search algorithm is a LIFO, a depth first search is obtained. These are fine for finite graphs. For infinite graphs, breadth first searches can be preferred.

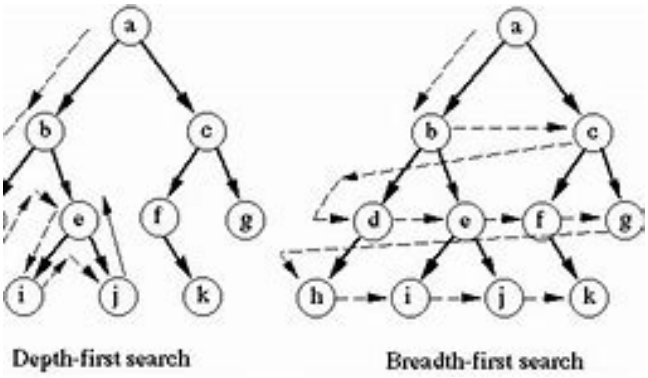


Figure 2: Depth first search moves farther and farther from the root, before visiting other nodes that are closer to the root. Breadth first search starts one edge away from the root, and after all nodes of that distance are visited moves to nodes of the next distance.