

## Data Structures and Algorithms II

### Lecture 10: Graph Search Algorithms

- Breadth first and depth first search
- Search algorithms
- Returning path information

1

### Breadth First vs Depth First Search

Two main search methods:

**Depth First:** Continue down current path until no more options. Then backup and try alternatives. (Children of current node explored before siblings).

**Breadth First:** Explore paths of length  $M$  before paths of length  $M+1$ .

Easiest illustrated by considering how they apply to searching *trees*:

3

### Graph Search

How do we find a path from a given 'start' node to a 'target' node, e.g.,:

Need methods to systematically go through all possible paths until target found.

2

### Searching Graphs

Illustrate for non-trees..

4

## Algorithms for Tree Search

If we restrict to trees, search algorithms are very simple:

For Depth First: Use a *stack* to hold nodes that are encountered in tree (but whose neighbours aren't examined):

```
push start node onto stack
do
  - Let current be top node in stack.
  - Pop this node off the stack.
  - For each neighbour n of current
    - push n onto the stack
while the stack isn't empty and haven't
found target.
```

Example:

1	stack:	current:
/ \	(1)	1
2 5	(2 5)	2
/ \	(3 4 5)	3
3 4	...	

5

## Breadth First Search

Breadth first search can be done the same method, but using a queue rather than stack:

```
put start node on end of queue
do
  - Let current be node on front of queue
  - Remove this node from the queue
  - For each neighbour n of the current
    - Add n to the end of the queue
while the queue isn't empty and haven't
found target.
```

Example:

1	queue:	current:
/ \	(1)	1
2 5	(2 5)	2
/ \	(5 3 4)	5
3 4	...	

7

## Depth First Search

Depth first search may also be done using recursion. This implicitly uses the run-time stack of the programming language (so is same alg underneath).

6

## Example Code in C++

Loop to do breadth first search, exiting when "target" found, using queue ADT from DS&A 1:

```
IntQueue nodeq;

nodeq.Add(startnode);
do
{
  currentnode = nodeq.GetValue();
  nodeq.Remove();
  // for each neighbour n of current node
  nodeq.Add(n);
}
while(! nodeq.IsEmpty() &&
      currentnode != target)
```

How would you do "for each neighbour n of current node" given graph ADT operations we have specified?

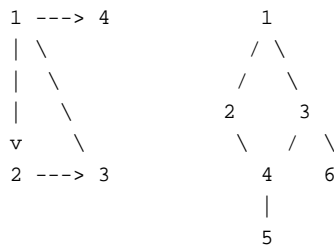
8

## Graphs with Loops

For cyclic graphs above algorithm would loop forever.

For graphs where node has more than one "parent" above algorithm inefficient.

Examples:



9

## Returning the Path

- So far, at best our routine will only tell you IF there is a path from start to target.
- Usually we want to know the path - ie, sequence of adjacent nodes that allows us to reach target from path.
- This can be done using neat trick: Store "parent" of each node added to the stack or queue:

```
// for each neighbour n of current node
if ( !visited[n] )
{
    nodeq.Add(n);
    parent[n] = currentnode;
}
```

11

## Graph Search with Loops

To avoid loops or repeated work, need to keep track of which nodes already visited.

Use array visited[GraphSize] where visited[n] = true if node n has been "visited". Add checks in algorithm to avoid revisiting nodes

```
nodeq.Add(startnode);
do
{
    currentnode = nodeq.GetValue();
    nodeq.Remove();
    if(! visited[currentnode])
    {
        visited[currentnode] = true;
        // for each neighbour n of current node
        if ( !visited[n] )
            nodeq.Add(n);
    }
}
while(! nodeq.IsEmpty() &&
currentnode != target)
```

10

## Returning the Path

Example:

12

### Returning the Path

- Once 'parent' array constructed, can output path by starting with node = target, then:

```
repeat
  node = parent[node];
output node
```

- This outputs path in order target to start.
- Can write a simple *recursive* function to output them in the right order.

13

### Summary

- Many applications require you to search for path from start to target node in graph.
- Standard systematic methods are using breadth first and depth first.
- Algorithms are identical but one uses stack, other queue.
- Simple methods to avoid loops and return the path.

15

### Exercise and Labs

Next exercise and labs involve:

- Implementing graph ADT using arrays.
- Using inheritance to create labelled graph ADT.
- Revising implementation to use pointers.
- Implementing algorithms for depth first search, returning and printing out the path (needs stack ADT).
- Using all this to solve a simple problem.

You should now be able to do most of this..

14