Graph Search

Computing 2 COMP1927 16x1
PROBLEMS ON GRAPHS

What kinds of problems do we want to solve on/via graphs?
- Is there a simple path from A to B
- Is the graph fully-connected?
- Can we remove an edge and keep it fully connected?
- Which vertices are reachable from v? (transitive closure)
- What is the cheapest cost path from v to w?
- Is there a cycle that passes through all V? (tour)
- Is there a tree that links all vertices (spanning tree)
- What is the minimum spanning tree?
- Can a graph be drawn in a place with no crossing edges?
- Are two graphs “equivalent”? (isomorphism)
We learn about properties of a graph by systematically examining each of its vertices and edges, for example:

- to compute the degree of all vertices, we visit each vertex and count its edges
- for path related properties, we have to move from vertex to vertex, along the graph's edges

We implement a general graph search algorithms we can use to solve a wide range of graph problems.
**Simple Path Search**

- **Problem:** is there a path from vertex $v$ to vertex $w$?

- **Approach to solving problem:**
  - examine vertices adjacent to $v$
    - if any of them is $w$, then we are done
    - otherwise check if there is a path from any of the adjacent vertices
  - repeat looking further and further from $v$

- **Two different approaches to order of searching:**
  breadth-first search (BFS), depth-first search (DFS)
BFS VS DFS PATH FINDING

- Is there a path from a to h?

Breadth-first Search

Depth-first Search
DFS vs BFS Approaches

- DFS and BFS are closely related.
- Implementation differs only their use of a stack or a queue
  - BFS implemented via a queue of to-be-visited vertices
  - DFS implemented via a stack of to-be-visited vertices (or recursion)

- Both approaches ignore some edges and avoid cycles by remembering previously visited vertices.
**Exercise: DFS and BFS Traversal**

- Show the DFS order we visit to determine `isPath(a,k)`
- Show the BFS order we visit to determine `isPath(a,k)`
- Assume neighbours are chosen in alphabetical order
**Depth First Search**

- Basic approach to depth-first search:
  - visit and mark current vertex
  - for each neighbour, traverse it recursively

- Notes:
  - need a mechanism for "marking" vertices
  - in fact, we number them as we visit them
    (so that we could later trace path through graph)

- Make use of three global variables:
  - count ... counter to remember how many vertices traversed so far
  - pre[] ... array saying order in which each vertex was visited (pre stands for preorder)
  - st[] ... array storing the predecessor of each vertex (st stands for spanning tree)
The edges traversed in a graph walk form a tree
It corresponds to the call tree of the recursive dfs function
Represents the original graph minus any cycles or alternate paths
We can use a tree to encode the whole search process
Each time we visit a vertex we record the previous vertex we came from - if the graph is connected this forms a spanning tree
• We store this in the st array
// Assume we start with dummy Edge {0,0}  
// assume we start with count = 0  
// pre[v] = -1 for all v  
// st[v] = -1 for all v (stores the predecessor)  
// assume adjacency matrix representation  
void dfsR (Graph g, Edge e) {  
    Vertex i, w = e.w;  
    pre[w] = count++;  
    st[w] = e.v;  
    for (i=0; i < g->V; i++) {  
        if ((g->edges[w][i] == 1) && (pre[i] == -1))  
            dfsR (g, mkEdge(g,w,i));  
    }  
}
Depth First Search (DFS)

- the edges traversed in the graph walk form a tree
- the tree corresponds to the call tree of the depth first search
  - and to the contents of the st array - spanning tree
- pre contains the pre-ordering of the vertices
**PROPERTIES OF DFS FORESTS**

- If a graph is not connected it will produce a spanning forest
  - If it is connected it will form a spanning tree
- We call an edge connecting a vertex with an ancestor in the DFS tree that is not its parent a **back edge**

![Graph Diagram]

- **Red dotted line:** back edge
EXERCISE: DFS TRAVERSAL

- Which vertices will be visited during dfs(g):

- How can we ensure that all vertices are visited?
The graph may not be connected

We need to make sure that we visit every connected component:

```c
void dfSearch (Graph g) {
    int v;
    count = 0;
    pre = malloc (sizeof (int) * g->nV));
    st = malloc (sizeof (int) * g->nV));
    for (v = 0; v < g->nV; v++) {
        pre[v] = -1;
        st[v] = -1;
    }
    for (v = 0; v < g->nV; v++) {
        if (pre[v] == -1) {
            dfsR (g, mkEdge(g,v,v));
        }
    }
}
```

The work complexity of the graph search algorithm is $O(V^2)$ for adjacency matrix representation, and $O(V+E)$ for adjacency list representation.
**Exercise: DFS Traversal**

- Trace the execution of dfs(g,0) on:

![Graph with nodes 0, 1, 2, 3, 4, 5]

- What if we were using DFS to search for a path from 0..5? We would get 0-1-2-3-4-5. If we want the shortest (least edges/vertices) path we need to use BFS instead. See later slides for this.
**Exercise: DFS Traversal**

- Show the final state of the pre and st arrays after dfs(g,0):

```
0 ---- 2 ---- 3
  |      |
  |      |
  v      v
  1 ---- 5
    |
    |   |
    |    v
    |      |
    |      |
    |    6
    |
  7 ---- 8
    |
    |
    |
    9
```
Non-Recursive Depth First Search

- We can use a stack instead of recursion:

```c
void dfs (Graph g, Edge e) {
    int i;
    Stack s = newStack();
    StackPush (s,e);
    while (!StackIsEmpty(s)) {
        e = StackPop(s);
        if (pre[e.w] == -1) {
            pre[e.w] = count++;
            st[e.w] = e.v;
            for (i = 0; i < g->nV; i++) {
                if ((g->edges[e.w][i] == 1) &&
                    (pre[i] == -1)) {
                    StackPush (s,mkEdge(g,e.w,i));
                }
            }
        }
    }
}
```
**DFS Algorithms: Cycle Detection**

- **Cycle detection**: does a given graph have any cycles?
  - if and only if the DFS graph has back edges, it contains cycles
  - we can easily detect this in the DFS search:

![Diagram of a graph with cycles detected by DFS](image)
DFS ALGORITHMS: CYCLE DETECTION

- We are only checking for the existence of cycle, we are not returning it

```c
//Return 1 if there is a cycle
int hasCycle (Graph g, Edge e) {
    int i, w = e.w;
    pre[w] = count++;
    st[w] = e.v;
    for (i=0; i < g->V; i++){
        if ((g->edges[w][i] == 1) && (pre[i] == -1)) {
            if(hasCycle (g, mkEdge(g,w,i)))
                return 1;
        } else if( (g->edges[w][i] == 1) && i != e.v) {
            //if it is not the predecessor
            return 1;
        }
    }
    return 0;
}
```
DFS Algorithms: Connectivity

- Each vertex belongs to a connected component
- The function `connectedComponents` sets up the array `cc` to indicate which component contains each vertex

```
Component 0
0 - 1

Component 1
2 - 4
3

Component 2
5 - 6 - 7 - 8

cc = [0, 0, 0, 1, 1, 2, 2, 2, 2]
```
DFS ALGORITHMS

- Connectivity:
  - maintain an extra array \( cc \) for connected components

```c
void connectedComponents (Graph g) {
    int v;
    count = 0;
    ccCount = 0;
    pre = malloc (g->nV * sizeof (int));
    cc = malloc (g->nV * sizeof (int));
    st = malloc (g->nV * sizeof (int));

    for (v = 0; v < g->nV; v++) {
        pre[v] = -1;
        st[v] = -1;
        cc[v] = -1;
    }

    for (v = 0; v < g->V; v++) {
        if (pre[v] == -1) {
            connectedR (g, mkEdge(g, v, v));
            ccCount++;
        }
    }
}

void connectedR (Graph g, Edge e) {
    int i, w = e.w;
    pre[w] = count++;
    st[w] = e.v;
    cc[w] = ccCount;

    for (i=0; i < g->V; i++){
        if ((g->edges[currV][i] == 1) &&
            (pre[i] == -1)) {
            dfsR (g, mkEdge(g, w, i));
        }
    }
}
```
What if we want the shortest path between two vertices?

- DFS doesn’t help us with this problem

To find the shortest path between \( v \) and any vertex \( w \):

- we visit all the vertices adjacent to \( v \) (distance 1)
- then all the vertices adjacent to those we visited in the first step (distance 2)
**Breadth-First Search**

- We observed previously that we can simply replace the stack with a queue in the non-recursive implementation to get breadth-first search:

```c
void bfs (Graph g, Edge e) {
    int i;
    Queue q = newQueue();
    QueueJoin(q, e);
    while (!QueueIsEmpty(q)) {
        e = QueueLeave(q);
        if (pre[e.w] == -1) {
            pre[e.w] = count++;
            st[e.w] = e.v;
            for (i = 0; i < g->nV; i++) {
                if ((g->edges[e.w][i] != 0) &&
                    (pre[i] == -1)) {
                    QueueJoin(q, mkEdge(g, e.w, i));
                }
            }
        }
    }
}
```
**IMPROVED BREADTH-FIRST SEARCH**

- We can mark them as visited as we put them on the queue since the queue will retain their order. Queue will have at most V entries

```java
void bfs(Graph g, Edge e) {
    int i;
    Queue q = newQueue();
    QueueJoin(q, e);
    pre[e.w] = count++;
    st[e.w] = e.v;
    while (!QueueIsEmpty(q)) {
        e = QueueLeave(q);
        for (i = 0; i < g->V; i++) {
            if ((g->edges[e.w][i] != 0) && (pre[i] == -1)) {
                QueueJoin(q, mkEdge(g, e.w, i));
                pre[i] = count++;
                st[i] = e.w;
            }
        }
    }
}
```
**Exercise: BFS Traversal**

- Show the final state of the pre and st arrays after bfs(g,0):

Write code to print out the shortest path from 0 to a given vertex v using the st array.
BREADTH-FIRST SEARCH

- For one BFS: $O(V^2)$ for adjacency matrix and $O(V+E)$ for adjacency list
- We can do BFS for every node as root node, and store the resulting spanning trees in a $V \times V$ matrix to store all the shortest paths between any two vertices
- To store and calculate these spanning trees, we need
  - memory proportional to $V \times V$
  - time proportional to $V \times E$
- Then, we can
  - return path length in constant time
  - path in time proportional to the path length