CSC 344 – Algorithms and
Complexity

Lecture #12 – Graphs (Extended)

What is a Graph?

- A graph consists of a set of *nodes* (or *vertices*) and a set of *arcs* (or *edges*).
- Each arc in a graphs is specified by a pair of nodes.
- If the pair of nodes that make up the arcs are *ordered pairs* then the graph is a *directed graph* or *digraph*.
Undirected Graph – An Example

Set of nodes = \{A, B, C, D, E, F, G, H\}
Set of arcs = \{(A, B), (A, D), (A, C), (C, D),
(C, F), (E, G), (A, A)\}

Directed Graph – An Example

Set of nodes = \{A, B, C, D, E, F, G, H\}
Set of arcs = \{<A, B>, <A, D>, <A, C>, <C, D>,
<F, C>, <E, G>, <A, A>\}
A graph need not be a tree but a tree must be a graph.
Other Definitions

- A node $n$ is incident to an arc $x$ if $n$ is one of the two nodes in the ordered pair of nodes constituting $x$. We also say that $x$ is incident to $n$.
- The **degree of a node** is the number of arcs incident to it.
- **indegree of $n$** – the number of arcs with $n$ as the head.
- **outdegree of $n$** – the number of arcs with $n$ as the tail.

Weighted Graphs

- A number may be associated with each arc of a graph. Such a graph is called a **weighted graph** or **network**. The number associated with an arc is called the **weight**.
Operations Used With Graphs

• *join* \((a, b)\) – adds an arc from node \(a\) to \(b\).
• *joinwt* \((a, b, x)\) – adds an arc from \(a\) to \(b\) with weight \(x\).
• *remove* \((a, b)\) – removes an arc from \(a\) to \(b\) if it exists.
• *removewt* \((a, b, x)\) – removes an arc from \(a\) to \(b\) and sets \(x\) to the weight of the now-defunct arc.

Paths and Cycles

• A path of length \(k\) from node \(a\) to node \(b\) is defined as a sequence of \(k + 1\) nodes \(n_1, n_2, \ldots, n_{k+1}\) such that \(n_1 = a\) and \(n_{k+1} = b\) and \(adjacent(n_i, n_{k+1})\) is true for all \(i\) between 1 and \(k\).
• A path from one node to itself is called a *cycle*.
• A graph with a cycle is cyclic; a graph without cycles is acyclic.
• Directed Acyclic Graphs are called *dags*. 
Transitive Closure

- Let’s assume that the adjacency matrix \( adj \) completely describes the graph (the nodes contain no data and the graph is unweighted).
  - if \( (adj[i][k] \&\& adj[k][j] == true) \)
  - // we have a 2-arc path from i to j
- What if the path requires 3 or more arcs?

Sample Graph

![Sample Graph](image-url)
### adj

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### adj₂

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
$adj_3$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$adj_4$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
\[
path = adj_1 \mid adj_2 \mid adj_3 \mid adj_4 \mid adj_5
\]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Graph.h

```cpp
#ifndef __GRAPH__
#define __GRAPH__
#endif

using namespace std;

const int MaxNodes = 50;
typedef int NodeStuffType;

struct node {
    NodeStuffType data;
};

struct arc {
    bool adj;
};
```
class Graph
{
public:
    Graph(void);
    void join(int node1, int node2);
    void remove(int node1, int node2);
    bool adjacent(int node1, int node2);
    void transClose(int path[][MaxNodes]);
private:
    void prod(int a[][MaxNodes],
              int c[][MaxNodes]);
    struct node nodes[MaxNodes];
    struct arc arcs[MaxNodes][MaxNodes];
};

Graph.cpp
#include "Graph.h"

Graph::Graph(void)
{
    int i, j;
    for (i = 0; i < MaxNodes; i++)
        for (j = 0; j < MaxNodes; j++)
            arcs[i][j].adj = false;
}

void Graph::join(int node1, int node2) {
    arcs[node1][node2].adj = true;
}
void Graph::remove(int node1, int node2) {
    arcs[node1][node2].adj = false;
}

bool Graph::adjacent(int node1, int node2) {
    return((arcs[node1][node2].adj == true)?
            true : false);
}

void Graph::transClose(int path[][MaxNodes]) {
    int i, j, k;
    int newprod[MaxNodes][MaxNodes],
        adjprod[MaxNodes][MaxNodes];

    for (i = 0;  i < MaxNodes; i++)
        for (j = 0;  j < MaxNodes;  j++)
            adjprod[i][j] = path[i][j] = arcs[i][j].adj;

    for (i = 1;  i < MaxNodes; i++)  
        // i represents the number of times adj
        // has been multiplied by itself to
        // obtain adjprod. At this point path
        // represents all paths of length i or
        // less
        for (j = 0;  j < MaxNodes;  j++)
            newprod[i][j] = adjprod[i][j];
prod(adjprod, newprod);
for (j = 0;  j < MaxNodes;  j++)
    for (k = 0;  k < MaxNodes;  k++)
        path[j][k] = path[j][k] || newprod[j][k];

for (j = 0;  j < MaxNodes;  j++)
    for (k = 0;  k < MaxNodes;  k++)
        adjprod[j][k] = newprod[j][k];

void Graph::prod(int a[][MaxNodes],
                 int c[][MaxNodes])
{    int i, j, k, val;

    for (i = 0;  i < MaxNodes; i++)
        //pass through rows
        for (j = 0;  j < MaxNodes;  j++)
        {        // pass through columns
            val = false;
            for (k = 0;  k < MaxNodes;  k++)
                val
                    = val ||
                    (a[i][k] && arcs[i][j].adj);
                c[i][j] = val;
        }  // for j..
}
Shortcoming in `transClosure()`

- The matrix multiplication that is performed is $O(n^3)$. It is performed $n-1$ times. That makes the efficiency of the algorithm $O(n^4)$, which is generally unacceptable.

Warshall's Algorithm

- We need a more efficient algorithm.
- Matrix $path_k$ is defined such that $path_k[i][j]$ is true if and only if there is path from node $i$ to $j$ that does not pass through any node numbered higher than $k$.
- Can we determine $path_{k+1}$ from $path_k$?
Warshall's Algorithm

- $path_{k+1}$ will be true if and only if:
  1. $path_k[i][j] == true$
  2. $path_k[i][k+1] == true$
     \&\& path_k[k+1][j] == true$

quickTransClose()

```cpp
void Graph::quickTransClose(int path[][MaxNodes]) {
    int i, j, k;
    for (i = 0; i < MaxNodes; i++)
        for (j = 0; j < MaxNodes; j++)
            path[i][j] = arcs[i][j].adj;

    for (k = 0; k < MaxNodes; k++)
        for (i = 0; i < MaxNodes; i++)
            if (path[i][k] == 1)
                for (j = 0; j < MaxNodes; j++)
                    path[i][j] = path[i][j] || path[k][j];
}
```
We want to find the shortest path from A to Z

A's distance is 0; everyone else is initialize to $\infty$
Dijkstra's Algorithm

We update A's neighbor's since it is closer to A

We update A's neighbor's

We update the distance to B

We update the distance to D

We update the distance to E

We update C's neighbor's since it is closer to A
We update C's neighbor's since it is closer to A.

We update the distance to D.

We update the distance to Z.

We update C's neighbor's since it is closer to A.
We update C's neighbor's distance since it is closer to A.

We update the distance to Z.