CS490: Problem Solving in Computer Science

Lecture 7: Introductory Graph Theory II

Dustin Tseng
Mike Li

Wednesday January 16, 2006

- Review of DFS/BFS
- Euler Cycles

- Review of DFS/BFS
- Euler Cycles

Introduction

Recall that

- an Euler cycle is a closed walk that visits each edge exactly once.
- we mentioned that there is a linear algorithm for finding an Euler cycle if it exists

Definition:

- the degree of a vertex is the number of edge coming into the vertex

Corrections

Let's first correct some of the statements I(Mike) made last time:
Definition:

- a path is a walk that does not pass any edge more than once.
- a simple path is a walk that does not pass any vertex more than once

Eulerian Path:

- when a graph has 2 vertices with odd degree, we have to start at one and end at the other
- this is a Eulerian path, not Euler cycle

The correct statement should be:

- a connected graph, G, has an Euler path if there are exactly two vertices with odd degree.

Euler Cycle

Claim:

- a connected graph, G, has an Euler cycle if the degree of each vertex is even.
- this should be somewhat intuitive, because if there is a vertex with odd degree, then we cannot return to this vertex.
- next let us explore how we can find an Euler cycle in a given graph

Cycle Detector

First let us start with a function that finds any cycle:

```
void greedyCycle( int u ) {
    while( true ) {
        int v;
        for( v = 0; v < n; v++ )
            if( graph[u][v] ) break;
        if( v < n ) {
            graph[u][v] = graph[v][u] = false;
            // add the edge (u,v) to the cycle
            u = v;
        }
        else break;
    }
}
```

Cycle Detector

The idea is quite simple here

- find an edge and add it to the cycle
- moreover erase this edge in the graph
- it should be clear now that if u has a odd degree, then we can always find a edge that leaves u and never be able to come back
- on the other hand if all vertices have even degree, by taking out (u,v), we now have exactly 2 odd vertices.
- and we will always have 2 odd vertices until we get back to u
- since there are finite number of edges, we know either we run out of edges (problem solved) or we get back to u prematurely

Euler Cycle

So how can we use our greedyCycle() to find us an Euler Cycle?

- notice that, if we reached back to u early, there will be a subgraph whose vertices' degree will all be even.
- this is because we subtracted a cycle away from the original graph
- in this subgraph, we can repeat our greedyCycle(), such that it gives us a new cycle, from v to v, and so on...
- we can actually insert these new cycles and form a complete Euler cycle

Euler Cycle

To do all that we can construct a recursive function:

```
list< int > cyc;

void euler( list< int >::iterator i, int u ) {
    for( int v = 0; v < n; v++ ) if( graph[u][v] ) {
        graph[u][v] = graph[v][u] = false;
        euler( cyc.insert( i, u ), v );
    }
}

int main() {
    // read graph into graph[][] and set n to the number of vertices
    euler( cyc.begin(), 0 );
    // cyc contains an euler cycle starting at 0
}
```

Euler Cycle

Couple things to notice here

- we are using a list to keep track of where to insert the next vertex
- two things can happen here:
- the function takes us back to the initial vertex u
- the function calls itself at v, and constructs a cycle to be inserted at v
- the complexity of this implementation is $O(mn)$
- this can be reduced to $O(m)$ if we were to use adjacency list plus some iterator manipulations

What Else?

- References:
  - Frank and Igor's CS490 notes.
  - Cormen, Thomas H., et al. Introduction to Algorithms.
- Good Luck!
- Seminar Schedule Change
- Tentative Midterm times
- Guest Speakers
- Problem Sets are really up!