References for Today’s Lecture

• Required reading
  – CLRS Chapter 22

• References
Operations on Graphs

- Last lecture, we saw an algorithm that could be used to analyze the connectivity of a graph without actually storing the graph.
- In discarding the list of edges, we lose a great deal of information about the structure of the graph.
- In many cases, we need to make use of that information to perform other operations.
- What are the basic operations we might want to perform on a graph if we could store it somehow?
class Graph{
    private:
        // Implementation dependent code
    public:
        Graph(int);
        ~Graph();
        int V() const;
        int E() const;
        int insert(Edge e);
        int delete(Edge e);
        bool edge(int v, int w) const;
        Vertex* getFirst(int i);
};
Vertex and Edge Classes

class Edge{
    public:
        Edge(int i, int j): v(i), w(j) {}
        ~Edge();
        int v, w;
};

class Vertex{
    private:
        // Implementation dependent code
    public:
        int getIndex() const;
        Vertex const * getNext() const;
    }

A Client Function for Printing a Graph

• Here’s an example of a standard way in which the graph interface class is used.

• Here, we print out a graph by enumerating all the edges incident to each vertex.

```cpp
void printGraph(const Graph& G)
{
    for (int s = 0; s < G.V(); s++){
        cout << s << ":\n";
        for (Vertex* t = G.getFirst(s); t != 0; t = t->getNext()){
            cout << Vertex->getIndex() << ":";
        }
    }
}
```
Graph Data Structures

- To support these basic graph operations, we need a data structure to store the graph.

- As with many previous data structures, there are basically two different ways to compactly represent a graph.
  - **Adjacency matrix**: An implementation based on arrays.
  - **Adjacency lists**: An implementation based on linked lists.

- We have to analyze the tradeoffs between these two representations, as we have before.
Adjacency Matrix Implementation

- Consider an undirected graph \( G = (V, E) \).
- The adjacency matrix \( A \) of \( G \) is an \( n \times n \) symmetric \( 0 - 1 \) matrix constructed as follows:

\[
A_{ij} = A_{ji} = \begin{cases} 
1 & \text{if } \{i, j\} \in E, \\
0 & \text{otherwise.}
\end{cases}
\]

- How do we implement the Graph class using an adjacency matrix?
Adjacency Lists Implementation

- The **adjacency list** for node \( i \) is a linked list of all other nodes adjacent to \( i \) in the graph.
- Note that adjacency lists do not have to be in any particular order.
- An adjacency list representation of a graph consists of an adjacency list for each node in the graph.
- How do we implement the **Graph** class using an adjacency lists?
Comparing the Implementations

- How does the *adjacency list* implementation compare to the *adjacency matrix* implementation?
  - Efficiency of basic operations
  - Memory requirements
Finding a Simple Path

• We now revisit the question of whether there is a path connecting a given pair of vertices in a graph.

• Using the operations in the *Graph* class, we can answer this question directly using a recursive algorithm.

• We must pass in a vector of bools to track which nodes have been visited.

```cpp
bool SPath(const graph& G, int v, int w, bool* visited) {
    if (v == w) return true;
    visited[v] = true;
    for (Vertex* t = G.getFirst(v); t != 0; t = t->getNext())
        if (!visited[t->getIndex()])
            if (SPath(G, t->getIndex(), w, visited)) return true;
    return false;
}
```
Finding a Hamiltonian Path

• Now let’s consider finding a path connecting a given pair of vertices that also visits every other vertex in between (called a Hamiltonian path).

• We can easily modify our previous algorithm to do this by passing an additional parameter $d$ to track the path length.

• What is the change in running time?

```cpp
bool HPath(const graph& G, int v, int w, bool* visited, int d) {
    if (v == w) return (d == 0);
    visited[v] = true;
    for (int* t = G.getFirst(v); t != 0; t = t->getNext())
        if (!visited[t->getIndex()])
            if (HPath(G, t->getIndex(), w, visited, d-1))
                return true;
    visited[v] = false;
    return false;
}
```
Hard Problems

- We have just seen an example of two very similar problems, one of which is hard and one of which is easy.

- In fact, there is no known algorithm for finding a Hamiltonian path that takes less than an exponential number of steps.

- This is our first example of a problem which is easy to state, but for which no known efficient algorithm exists.

- Many such problems arise in graph theory and it’s difficult to tell which ones are hard and which are easy.

- Consider the problem of finding an Euler path, which is a path between a pair of vertices that includes every edge exactly once.

- Does this sound like a hard problem?
Depth-first Search

• With an algorithm called *depth-first search*, we can compute the number of connected components of a graph, and label each node with a component number.

• This allows to answer the question of whether two nodes are in the same connected component in constant time.

• Here is the basic depth-first search function.

```cpp
void DFS(const graph& G, int v, const int compNum, int* comps) {
    comps[v] = compNum;
    for (Vertex* t = G.getFirst(v); t != 0; t = t->getNext())
        if (!comp[t->getIndex()])
            DFS(G, t->getIndex, compNum, comp)
}
```

• What is the running time of this algorithm?
Component Labeling

• To label all vertices with a component number, we simply call DFS iteratively on each vertex that has not been labeled.

```cpp
int Label(const graph& G) {
    int numComps(0);
    int *comps(new int[G.V()]);
    for (int s = 0; s < G.V(); s++) comps[v] = 0;
    for (int s = 0; s < G.V(); s++) {
        if (!comps[v]) {
            numComps++;
            DFS(G, v, compNum, comp);
        }
    }
    return numComps;
}
```
DFS vs. Union-Find

• What is the overall running time of the labeling algorithm?
• How does this compare to union-find?
• What are the advantages of union-find?