1 Graph Concepts

2 Graph Storage

3 Directed Graphs

4 Finding a Path Between Two Vertices
   4.1. Depth-First Search
   4.2. Breadth-First Search
   4.3. Dijkstra’s Shortest Path Algorithm

5 Constructing a Minimum Spanning Tree
   5.1. Prim’s Algorithm
   5.2. Kruskal’s Algorithm

6 Interplay between Algorithms and Data Structures
1. Graph Concepts
2. Graph Storage

![Graph Diagram]

- Nodes: 0, 1, 2, 3, 4
- Edges with weights:
  - From 0 to 1: 5
  - From 0 to 2: 6
  - From 1 to 3: 13
  - From 2 to 3: 3
  - From 3 to 1: 1
  - From 3 to 4: 2
  - From 4 to 3: 10

- Direction of edges indicated by arrows.
3. Directed Graphs

- Focus on object-oriented adjacency-list-based directed graphs storing int weights
  - Could apply same approach to undirected graphs
  - Could use object-oriented programming and dynamic polymorphism
  - Could use generic programming and static polymorphism
  - Could use functional programming to analyze graph
  - Could use concurrent programming to analyze graph in parallel

```cpp
class GraphInt
{
  public:
    int add_vertex();
    void add_edge( int src_id, int dest_id, int w );
    Vector<int> get_neighbors( int id );
    int get_weight( int src_id, int dest_id );

  private:
    Vector< Vector< Pair<int,int> > > m_graph;
};
```
3. Directed Graphs

```cpp
int GraphInt::add_vertex()
{
    m_graph.push_back( Vector<Pair<int,int>>() );
    return m_graph.size() - 1;
}

void GraphInt::add_edge( int src_id, int dest_id, int w )
{
    m_graph.at(src_id).push_back(
        Pair<int,int>( dest_id, w ) );
}

Vector<int> GraphInt::get_neighbors( int id )
{
    Vector<int> neighbors;
    for ( auto e : m_graph.at(id) )
        neighbors.push_back( e.first );
    return neighbors;
}

int GraphInt::get_weight( int src_id, int dest_id )
{
    for ( auto e : m_graph.at(src_id) )
        if ( e.first == dest_id )
            return e.second;
    assert(false);
}
```
Draw the conceptual graph and the adjacency list storage resulting from this code sequence:

```
GraphInt g;

int v0 = g.add_vertex();
int v1 = g.add_vertex();
int v2 = g.add_vertex();
int v3 = g.add_vertex();
int v4 = g.add_vertex();
int v5 = g.add_vertex();
int v6 = g.add_vertex();

g.add_edge( v0, v1, 1 );
g.add_edge( v0, v2, 1 );
g.add_edge( v0, v3, 1 );
g.add_edge( v1, v6, 1 );
g.add_edge( v2, v4, 1 );
g.add_edge( v3, v5, 1 );
g.add_edge( v4, v6, 1 );
g.add_edge( v5, v4, 1 );
```
### Time and space complexity analysis for different storage

- Let a graph $G$ be a pair $(V, E)$
  - $V$ is a set of vertices, $|V|$ is the number of vertices
  - $E$ is a set of edges, $|E|$ is the number of edges
  - we often informally just use $V$ and $E$ to represent $|V|$ and $|E|$  

<table>
<thead>
<tr>
<th>Adjacency Matrix</th>
<th>Adjacency List: Outer Vector with ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vector</td>
</tr>
</tbody>
</table>

#### Space Usage

<table>
<thead>
<tr>
<th>Operation</th>
<th>Adjacency Matrix</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_vertex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add_edge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_neighbors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>get_weight</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Adjacency Matrix Diagram

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>3</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td>13</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

#### Adjacency List Diagram (with inner Vector)

```
<table>
<thead>
<tr>
<th>src</th>
<th>dest</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
```

---

**Topic 19: Graphs**
Can we use an alternative inner data structure to improve the performance of getting the weight for a given edge?

- `InnerDataStruct<K,V>` is a map implemented with BST or hash table

```cpp
int GraphInt::add_vertex() {
    m_graph.push_back( InnerDataStruct<int,int>() );
    return m_graph.size() - 1;
}

void GraphInt::add_edge( int src_id, int dest_id, int w ) {
    m_graph.at(src_id).add( dest_id, w );
}

Vector<int> GraphInt::get_neighbors( int id ) {
    Vector<int> neighbors;
    for ( auto n : m_graph.at(id) )
        neighbors.push_back( n.first );
    return neighbors;
}

int GraphInt::get_weight( int src_id, int dest_id ) {
    return m_graph.at(src_id).lookup(dest_id);
}
```

**Adjacency List (with inner BinarySearchTree)**

**Adjacency List (with inner HashTree)**
4. Finding a Path Between Two Vertices

- Given
  - graph $G = (V, E)$
  - source vertex $V_s$
  - destination vertex $V_d$

- Find a path from $V_s$ to $V_d$
4. Finding a Path Between Two Vertices

- We will explore three different algorithms:
  - **Depth-First Search**: finds a path if it exists
  - **Breadth-First Search**: finds a path if it exists
  - **Dijkstra’s Algorithm**: finds *shortest* path if it exists

```cpp
class GraphInt {
public:
    ... Vector<int> dfs ( int src_id, int dest_id );
    Vector<int> bfs ( int src_id, int dest_id );
    Vector<int> dijkstra( int src_id, int dest_id );
};
```
4.1. Depth-First Search

```python
def GraphInt::dfs( src_id, dest_id ):
    set visited to be a set # vertices already visited
    set worklist to be a stack # pending paths to search

    worklist.push( [src_id] )

    while worklist is not empty:
        path = worklist.pop()
        set v to be final vertex in path

        if v == dest_id:
            return path

        if v not in visited:
            visited.add( v )
            for n in get_neighbors( v ):
                worklist.push( path + n )
```

![Diagram of a graph with Depth-First Search visits and worklist updates]

**Visited:**

**Worklist:**

---

Topic 19: Graphs
Vector<int> GraphInt::dfs( int src_id, int dest_id )
{
    Set<int> visited; // vertices already visited
    Stack<Vector<int>> worklist; // pending paths to search

    // Initialize worklist with path containing just source vertex
    Vector<int> p; p.push_back(src_id); worklist.push( p );

    // Keep working until worklist is empty
    while ( worklist.size() != 0 ) {
        // Pop path from _top_ of stack
        auto path = worklist.pop();

        // Check if final vertex in current path is destination
        int v = path.at( path.size()-1 );
        if ( v == dest_id ) return path;

        // Check if final vertex has already been visited
        if ( !visited.contains( v ) ) {
            // Mark final vertex as visited
            visited.add( v );

            // Iterate through neighbors
            auto neighbors = get_neighbors( v );
            for ( int n : neighbors ) {
                // Create temporary new path with neighbor at end
                auto temp = path;
                temp.push_back(n);

                // Push this new path onto _top_ of stack
                worklist.push( temp );
            }
        }
    }
}
4.2. Breadth-First Search

```python
def GraphInt::bfs( src_id, dest_id):
    set visited to be a set # vertices already visited
    set worklist to be a queue # pending paths to search

    worklist.enq( [src_id] )
    while worklist is not empty:
        path = worklist.deq()
        set v to be final vertex in path

        if v == dest_id:
            return path

        if v not in visited:
            visited.add( v )
            for n in get_neighbors( v ):
                worklist.enq( path + n )
```

**Diagram:**

- **Visited:** Visiting vertices in order of their encounter.
- **Worklist:** Queue of paths to explore.

**Graph:**
- **Visited:** Vertices visited in the order of exploration.
- **Worklist:** Paths explored in the order of their entry into the queue.
4. Finding a Path Between Two Vertices

4.2. Breadth-First Search

```cpp
Vector<int> GraphInt::bfs( int src_id, int dest_id )
{
    Set<int> visited; // vertices already visited
    Queue<Vector<int>> worklist; // pending paths to search

    // Initialize worklist w/ path containing just source vertex
    Vector<int> p; p.push_back(src_id); worklist.enq( p );

    // Keep working until worklist is empty
    while ( worklist.size() != 0 ) {
        // Dequeue path from _head_ of queue
        auto path = worklist.deq();

        // Check if final vertex in current path is destination
        int v = path.at( path.size()-1 );
        if ( v == dest_id ) return path;

        // Check if final vertex has already been visited
        if ( !visited.contains( v ) ) {
            // Mark vertex as visited
            visited.add( v );

            // Iterate through neighbors
            auto neighbors = get_neighbors( v );
            for ( int n : neighbors ) {
                // Create temporary new path with neighbor at end
                auto temp = path;
                temp.push_back(n);

                // Enqueue this path on _tail_ of queue
                worklist.enq( temp );
            }
        }
    }
}
```
4. Finding a Path Between Two Vertices  4.3. Dijkstra’s Shortest Path Algorithm

4.3. Dijkstra’s Shortest Path Algorithm
5. Constructing a Minimum Spanning Tree

5.1. Prim’s Algorithm

5.2. Kruskal’s Algorithm
### Algorithms

- **mul**: iter, single step
- **sqrt**: iter, recur
- **search**: linear, binary
- **sort**: insertion, selection, merge, quick, hybrid, bucket
- **set intersection, set union**
- **find path**: DFS, BFS, Dijkstra

### Data Structures

- **chain of nodes**
- **array of elements**
- **list, vector**
- **stack, queue, set, map**
- **tree, table, graph**

---

- Simple algorithms do not use a non-trivial data structure
- Simple data structures do not provide non-trivial operations
- Many algorithms operate on a simple data structure
- Many data structures provide operations which are implemented using an algorithm that operates on a simple data structure

- Sometimes our programs are more **algorithm centric**, sometimes they are more **data-structure centric**, but they **almost always use both algorithms and data structures**

### Algorithm + Data Structure = Program