ECE 2400 Computer Systems Programming Fall 2021

Topic 19: Graphs

School of Electrical and Computer Engineering
Cornell University

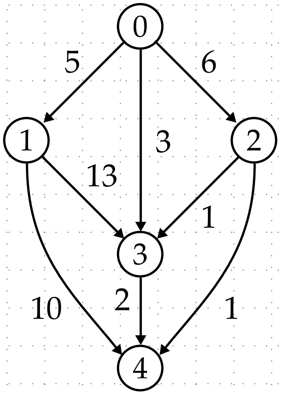
revision: 2021-08-29-22-40

Handout for Sections 4.3 and 5 will be released later in the semester!

zyBooks The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

1. Graph Concepts

2. Graph Storage



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

```
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;
};
```

```
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:

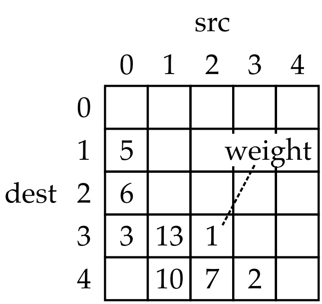
```
GraphInt g;

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

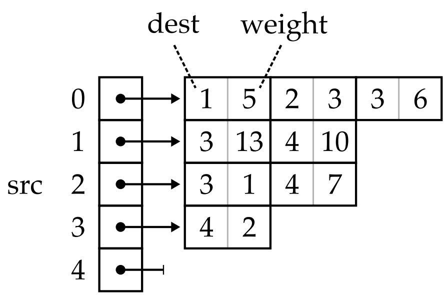
- Let a graph G be a pair (V, E)
  - V is a set of vertices, $|V|$ is the number of vertices
  - E is a set of edges, $|E|$ is the number of edges
  - we often informally just use V and E to represent $|V|$ and $|E|$

	Adjacency Matrix	**Adjacency List:** Inner data structure is ...		
		`Vector`	`BST`	`HashTable`
Space Usage				
`add_vertex`				
`add_edge`				
`get_neighbors`				
`get_weight`				

Adjacency Matrix



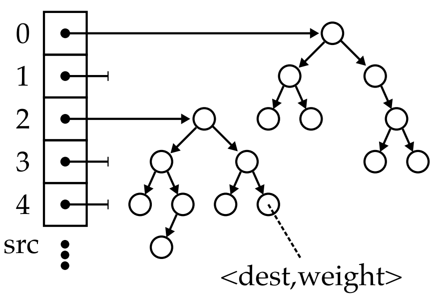
Adjacency List
(with inner `Vector`)



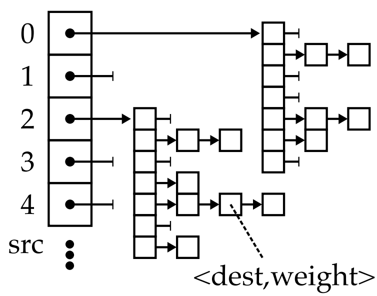
- Can we use an alternative inner data structure to improve the performance of getting the weight for a given edge?
  - `InnerDataStruct<K,V>` is a map implemented with BST or hash table

```
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`)

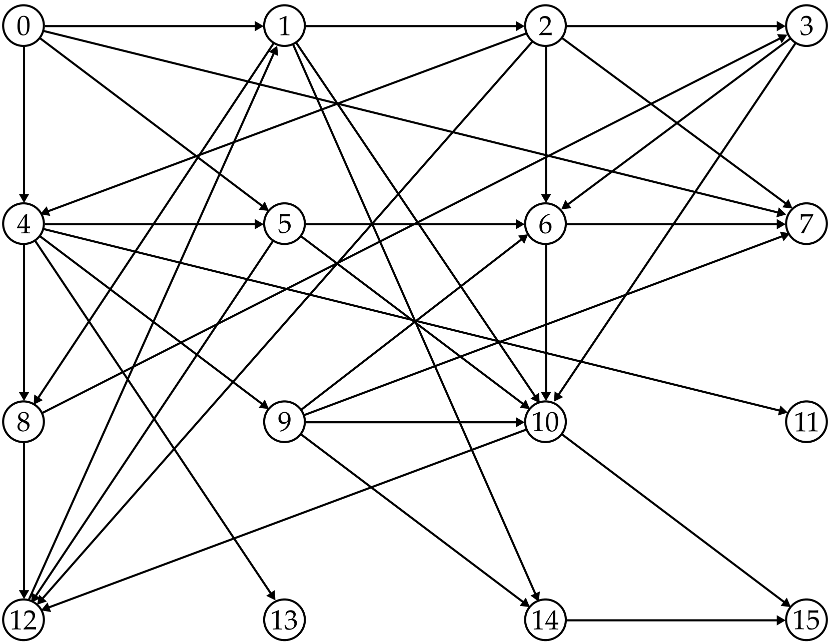


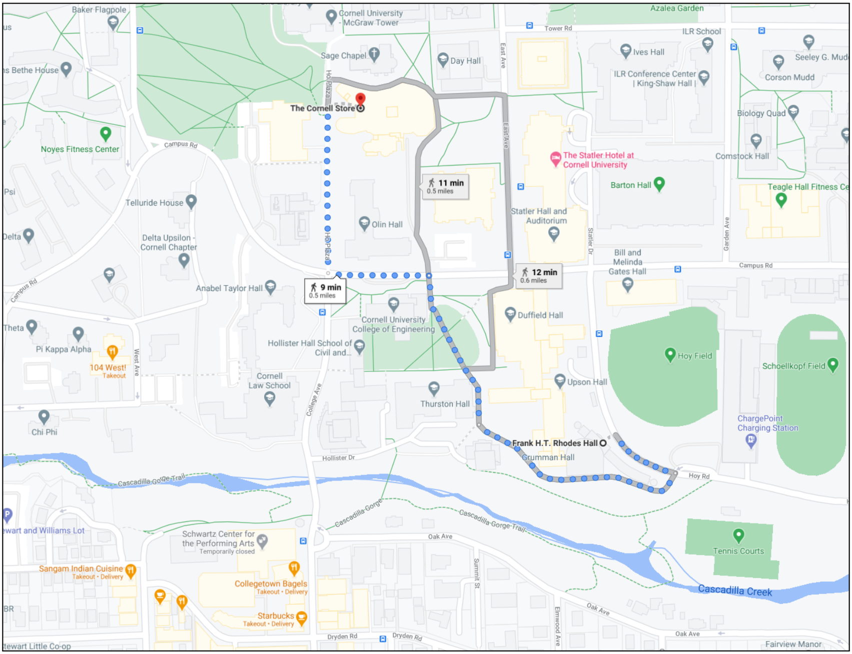
Adjacency List
(with inner `HashTree`)



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



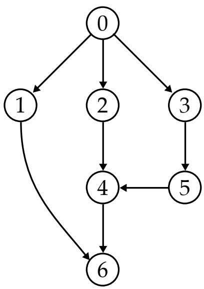


- We will explore three different algorithms:
  - Depth-First Search: finds a path if it exists
  - Breadth-First Search: finds a path if it exists
  - Dijkstra's Algorithm: finds *shortest* path if it exists

```
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 );
};
```

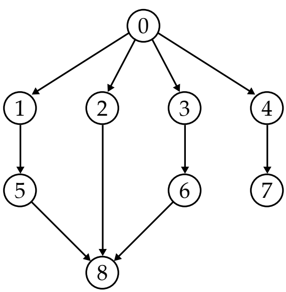
4.1. Depth-First Search

```
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 )
```



visited:

worklist:



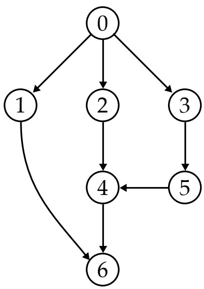
visited:

worklist:

```
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 w/ 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 );
      }
    }
  }
}
```

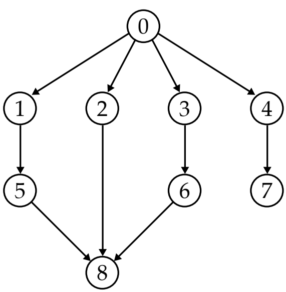
4.2. Breadth-First Search

```
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 )
```



visited:

worklist:



visited:

worklist:

```
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 );
      }
    }
  }
}
```

4.3. Dijkstra's Shortest Path Algorithm

5. Constructing a Minimum Spanning Tree

5.1. Prim's Algorithm

5.2. Kruskal's Algorithm

Algorithms	**Data Structures**
mul: iter, single step sqrt: iter, recur	chain of nodes array of elements
search: linear, binary sort: insertion, selection, merge, quick, hybrid, bucket set intersection, set union find path: DFS, BFS, Dijkstra	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