SELF-BALANCING SEARCH TREES

Chapter 11
Tree Balance and Rotation

Section 11.1
Algorithm for Rotation

root

BTNode
left
right
data = 10

BTNode
left
right
data = 20

BTNode
left
right
data = 40

NULL

BTNode
left
right
data = 5

NULL

BTNode
left
right
data = 15

NULL

BTNode
left
right
data = 7

NULL
Algorithm for Rotation (cont.)

1. Remember value of root->left
   (temp = root->left)
Algorithm for Rotation (cont.)

1. Remember value of root->left (temp = root->left)
2. Set root->left to value of temp->right
Algorithm for Rotation (cont.)

1. Remember value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
Algorithm for Rotation (cont.)

1. Remember value of root->left (temp = root->left)
2. Set root->left to value of temp->right
3. Set temp->right to root
4. Set root to temp
#ifndef BST_WITH_ROTATE_H_
#define BST_WITH_ROTATE_H_
#include "Binary_Search_Tree.h"

/** This class extends the Binary_Search_Tree by adding the rotate operations. Rotate will change the balance of a search tree while preserving the search tree property. Used as a common base class for self-adjusting trees. */

template<typename Item_Type>
class BST_With_Rotate : public Binary_Search_Tree<Item_Type> {

protected:

/** rotate_right
 * pre:  local_root is the root of a binary search tree
 * post: local_root->left is the root of a binary search tree
 *       local_root->left->left is raised one level
 *       local_root->left->right does not change levels
 *       local_root->right is lowered one level
 *       local_root is set to the new root
 * @param local_root The root of the binary tree to be rotated
 */

void rotate_right(BTNode<Item_Type> *& local_root) {
    BTNode<Item_Type> * temp = local_root->left;
    local_root->left = temp->right;
    temp->right = local_root;
    local_root = temp;
}

// Insert rotate_left here
// Exercise

};

#endif
Section 11.2

AVL Trees
Implementing an AVL Tree

- Binary_Tree
  - get_left_subtree()
  - get_right_subtree()
  - get_data()
  - to_string()

- Binary_Search_Tree
  - bool insert(const Item_Type&)
  - bool erase(const Item_Type&)
  - const Item_Type& find(const Item_Type&)

- BST_With_Rotate
  - rotate_left()
  - rotate_right()
Implementing an AVL Tree (cont.)

```cpp
#ifndef AVL_TREE_H
#define AVL_TREE_H

#include "BST_With_Rotate.h"
#include "AVLNode.h"

/** Definition of the AVL Binary Search Tree class.
   * @param Item_Type The type of item to be stored in the tree
   * Note: Item_Type must define the less-than operator as a
   * total ordering.
   */

template<typename Item_Type>
class AVL_Tree : public BST_With_Rotate<Item_Type> {

public:
  // Constructor
  /** Construct an empty AVL_Tree */
  AVL_Tree() : BST_With_Rotate<Item_Type>() {} {

  // Public Member Functions
  /** Insert an item into the tree.
      * post: The item is in the tree.
      * @param item The item to be inserted
      * @return true only if the item was not
      * already in the tree
      */
  virtual bool insert(const Item_Type& item) {
    return insert(this->root, item); }
```

Implementing an AVL Tree (cont.)

```cpp
/** Remove an item from the tree.
   post: The item is no longer in the tree.
   @param item The item to be removed
   @return true only if the item was in the tree
 */
virtual bool erase(const Item_Type& item) {
    return erase(this->root, item); }

private:
    // Private member functions declarations
    ...

    // Data Fields
    /** A flag to indicate that the height of the tree has increased */
    bool increase;

}; // End of AVL_Tree class definition

// Implementation of member functions
... #endif
```
The AVLNode Class

```c++
#ifndef AVLNODE_H_
#define AVLNODE_H_
#include <sstream>

/** A node for an AVL Tree. */
template<typename Item_Type>
struct AVLNode : public BTNNode<Item_Type> {
    enum balance_type {LEFT_HEAVY = -1, BALANCED = 0, RIGHT_HEAVY = +1};
    // Additional data field
    balance_type balance;

    // Constructor
    AVLNode(const Item_Type& the_data, BTNNode<Item_Type>* left_val = NULL,
             BTNNode<Item_Type>* right_val = NULL) :
        BTNNode<Item_Type>(the_data, left_val, right_val), balance(BALANCED) {}

    // Destructor (to avoid warning message)
    virtual ~AVLNode() {}

    // to_string
    virtual std::string to_string() const {
        std::ostringstream os;
        os << balance << "": " << this->data;
        return os.str();
    }
}; // End AVLNode

#endif
```
Inserting into an AVL Tree

- The easiest way to keep a tree balanced is never to let it remain critically unbalanced.
- If any node becomes critical, rebalance immediately.
- Identify critical nodes by checking the balance at the root node as you return along the insertion path.
Inserting into an AVL Tree (cont.)

Algorithm for Insertion into an AVL Tree

1. if the root is NULL
2. Create a new tree with the item at the root and return true
   else if the item is equal to root->data
3. The item is already in the tree; return false
   else if the item is less than root->data
4. Recursively insert the item in the left subtree.
5. if the height of the left subtree has increased (increase is true)
   6. Decrement balance
   7. if balance is zero, reset increase to false
   8. if balance is less than −1
   9. Reset increase to false.
10. Perform a rebalance_left
   else if the item is greater than root->data
11. The processing is symmetric to Steps 4 through 10. Note that balance is incremented if increase is true.
Recursive insert Function

- The recursive insert function is called by the insert starter function (see the AVL_Tree Class Definition)

```cpp
/** Insert an item into the tree.
 * post: The item is in the tree.
 * @param local_root A reference to the current root
 * @param item The item to be inserted
 * @return true only if the item was not already in the tree
 */
virtual bool insert(BTNode<Item_Type>*& local_root,
                     const Item_Type& item) {
    if (local_root == NULL) {
        local_root = new AVLNode<Item_Type>(item);
        increase = true;
        return true;
    }
}
```
Recursive insert Function (cont.)

if (item < local_root->data) {
    bool return_value = insert(local_root->left, item);
}
if (increase) {
    AVLNode<Item_Type>* AVL_local_root =
    dynamic_cast<AVLNode<Item_Type>*>(local_root);
    switch (AVL_local_root->balance) {
        case AVLNode<Item_Type>::BALANCED :
            // local root is now left heavy
            AVL_local_root->balance =
            AVLNode<Item_Type>::LEFT_HEAVY;
            break;
        case AVLNode<Item_Type>::RIGHT_HEAVY :
            // local root is now right heavy
            AVL_local_root->balance = AVLNode<Item_Type>::BALANCED;
            // Overall height of local root remains the same
            increase = false;
            break;
    }
case AVLNode<Item_Type>::LEFT_HEAVY :
    // local root is now critically unbalanced
    rebalance_left(local_root);
    increase = false;
    break;
    } // End switch
    } // End (if increase)
    return return_value
} // End (if item <local_root->data)
else {
    increase = false
    return false;
}
Recursive insert Function (cont.)

Balance before insert is 0  
Balance is decreased due to insert; Overall height increased

Balance before insert is +1  
Balance is decreased due to insert; Overall height remains the same
Initial Algorithm for `rebalance_left`

1. if the left subtree has positive balance (Left-Right case)
2. Rotate left around left subtree root.
3. Rotate right.
The rebalance algorithm on the previous slide is incomplete as the balance of the nodes has not been adjusted.

For a Left-Left tree the balances of the new root node and of its right child are 0 after a right rotation.

Left-Right is more complicated:
- the balance of the root is 0.
Effect of Rotations on Balance (cont.)

- if the critically unbalanced situation was due to an insertion into
  - subtree $b_L$ (Left-Right-Left case), the balance of the root's left child is 0 and the balance of the root's right child is +1
Effect of Rotations on Balance (cont.)

- If the critically unbalanced situation was due to an insertion into subtree $b_R$ (Left-Right-Right case), the balance of the root's left child is -1 and the balance of the root's right child is 0.
Revised Algorithm for rebalance_left

1. if the left subtree has a positive balance (Left-Right case)
2.   if the left-right subtree has a negative balance (Left-Right-Left case)
3.      Set the left subtree (new left subtree) balance to 0
4.      Set the left-left subtree (new root) balance to 0
5.      Set the local root (new right subtree) balance to +1
6.   else if the left-right subtree has a positive balance (Left-Right-Right case)
7.      Set the left subtree (new left subtree) balance to −1
8.      Set the left-left subtree (new root) balance to 0
9.      Set the local root (new right subtree) balance to 0
10.  else (Left-Right Balanced case)
11.     Set the left subtree (new left subtree) balance to 0
12.     Set the left-left subtree (new root) balance to 0
13.     Set the local root (new right subtree) balance to 0
14.   Rotate the left subtree left
15. else (Left-Left case)
16.     Set the left subtree balance to 0
17.     Set the local root balance to 0
18. Rotate the local root right
Function rebalance_left

template<typename Item_Type>
void AVL_Tree<Item_Type>::rebalance_left(BTNode<Item_Type>* &local_root)
{
    // Cast local_root to an AVLNode pointer
    AVLNode<Item_Type>* AVL_local_root =
        dynamic_cast<AVLNode<Item_Type>*>(local_root);
    // Obtain reference to the left child
    AVLNode<Item_Type>* left_child =
        dynamic_cast<AVLNode<Item_Type>*>(local_root->left);
    // See whether left-right-heavy
    if (left_child->balance == AVLNode<Item_Type>::RIGHT_HEAVY) {
        // Obtain a reference to the left-right child
        AVLNode<Item_Type>* left_right_child =
            dynamic_cast<AVLNode<Item_Type>*>(left_child->right);
        // Adjust the balances to be the new values after rotations are
        // performed
        if (left_right_child->balance == AVLNode<Item_Type>::LEFT_HEAVY) {
            left_child->balance = AVLNode<Item_Type>::BALANCED;
            left_right_child->balance = AVLNode<Item_Type>::BALANCED;
            AVL_local_root->balance = AVLNode<Item_Type>::RIGHT_HEAVY;
        } else if (left_right_child->balance == AVLNode<Item_Type>::BALANCED) {
            left_child->balance = AVLNode<Item_Type>::BALANCED;
            left_right_child->balance = AVLNode<Item_Type>::BALANCED;
            AVL_local_root->balance = AVLNode<Item_Type>::BALANCED;
        } else {
            left_child->balance = AVLNode<Item_Type>::LEFT_HEAVY;
            left_right_child->balance = AVLNode<Item_Type>::BALANCED;
            AVL_local_root->balance = AVLNode<Item_Type>::BALANCED;
        }
    }
    // Perform left rotation
    rotate_left(local_root->left);
} else { // Left-Left case
    /* In this case the left child (the new root) and the
    local root (new right child) will both be balanced
    after the rotation.
    */
    left_child->balance = AVLNode<Item_Type>::BALANCED;
    AVL_local_root->balance = AVLNode<Item_Type>::BALANCED;
}
// Finally rotate right
rotate_right(local_root);
Removal from an AVL Tree

- Removal
  - from a left subtree, increases the balance of the local root
  - from a right subtree, decreases the balance of the local root
- The binary search tree removal function can be adapted for removal from an AVL tree
- A data field `decrease` tells the previous level in the recursion that there was a decrease in the height of the subtree from which the return occurred
- The local root balance is incremented or decremented based on this field
- If the balance is outside the threshold, a rebalance function is called to restore balance
Functions `rebalance_left`, and `rebalance_right` need to be modified so that they set the balance value correctly if the left (or right) subtree is balanced.

- When a subtree changes from either left-heavy or right-heavy to balanced, then the height has decreased, and decrease should remain `true`.
- When the subtree changes from balanced to either left-heavy or right-heavy, then decrease should be reset to `false`.

Each recursive return can result in a further need to rebalance.
Performance of the AVL Tree

- Since each subtree is kept close to balanced, the AVL has expected $O(\log n)$
- Each subtree is allowed to be out of balance ±1 so the tree may contain some holes
- In the worst case (which is rare) an AVL tree can be 1.44 times the height of a full binary tree that contains the same number of items
- Ignoring constants, this still yields $O(\log n)$ performance
- Empirical tests show that on average $\log_2 n + 0.25$ comparisons are required to insert the $n$th item into an AVL tree – close to insertion into a corresponding complete binary search tree
Chapter 12

GRAPHS
Overview of the Hierarchy

iter_impl \rightarrow \text{iterator} \rightarrow \text{Graph}

\text{List}_\text{Graph}

\text{Matrix}_\text{Graph}

\text{iter_impl} \rightarrow \text{Edge} \rightarrow \text{iter_impl}
# Class Graph

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool directed</td>
<td>true if this is a directed graph.</td>
</tr>
<tr>
<td>int num_v</td>
<td>The number of vertices.</td>
</tr>
</tbody>
</table>

## Constructor

- **Graph(int n, bool d)**
  - Constructs an empty graph with the specified number of vertices \( n \) and with the specified directed flag \( d \). If directed is **true**, this is a directed graph.

## Destructor

- **virtual ~Graph()**
  - The destructor.

## Function

### Behavior

- **int get_num_v()**
  - Gets the number of vertices.

- **bool is_directed()**
  - Returns **true** if the graph is a directed graph.

- **void load_edges_from_file(istream& in)**
  - Loads edges from a data file.

- **static Graph* create_graph (istream& in, bool is_directed, const string& type)**
  - Factory function to create a graph and load the data from an input of the specified type.
Implementation

```cpp
#include "Graph.h"
#include "Matrix_Graph.h"
#include "List_Graph.h"
#include <string>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <limits>
using std::string;
using std::istream;
using std::stringstream;
using std::numeric_limits;

/** Load the edges of a graph from the data in an input file. 
The file should contain a series of lines, each line 
with two or three data values. The first is the source, 
the second is the destination, and the optional third 
is the weight. 
@param[in] The istream that is connected 
to the file that contains the data
*/
void Graph::load_edges_from_file(istream& in) {
    // Programming exercise
}

/** Factory function to create a graph and load the data from an input 
file. The first line of the input file should contain the number 
of vertices. The remaining lines should contain the edge data as 
described under load_edges_from_file. 
@param[in] The istream that is connected to the file that contains 
the data 
@param[in] The string "Matrix" if an adjacency matrix is to be 
created, and the string "List" if an adjacency list 
is to be created 
@throws std::invalid_argument if type is neither "Matrix" nor "List"
*/
Graph* Graph::create_graph(istream& in, bool is_directed, 
    const std::string& type) {
    int n;
    in >> n;
    in.ignore(numeric_limits<int>::max(), '\n'); // Skip rest of this line
    Graph* return_value = NULL;
    if (type == "Matrix")
        return_value = new Matrix_Graph(n, is_directed);
    else if (type == "List")
        return_value = new List_Graph(n, is_directed);
    else
        throw std::invalid_argument("Unrecognized Graph Type");
    return_value->load_edges_from_file(in);
    return return_value;
}```
The iterator and iter_impl Classes

/** An iterator provides sequential access to the edges adjacent to a given vertex. */

class iterator {

public:
    Edge operator*() { return ptr_to_impl->operator*(); }

    iterator& operator++() {
        ++(*ptr_to_impl);
        return *this;
    }

    iterator operator++(int) {
        iterator temp(*this);
        ++(*ptr_to_impl);
        return temp;
    }

bool operator==(const iterator& other) const {
    return *ptr_to_impl == *other.ptr_to_impl;
}

bool operator!=(const iterator& other) const {
    return !((*this) == other);
}

~iterator() {delete ptr_to_impl;}

iterator(const iterator& other) :
    ptr_to_impl(other.ptr_to_impl->clone()) {}}

/** Constructor. */
    @param p_graph Pointer to the graph being iterated over
    @param p_impl Pointer to iterator implementation
*/
iterator(iter_impl* p_impl) : ptr_to_impl(p_impl) {}
The iterator and iter_impl Classes

```cpp
private:
    /** Pointer to the implementation */
    iter_impl* ptr_to_impl;
}; // End iterator

/** The iter_impl class defines abstract functions
   to implement the iterator operations.
*/
class iter_impl {
    public:
        virtual Edge operator*() = 0;
        virtual iter_impl& operator++() = 0;
        virtual bool operator==(const iter_impl&) const = 0;
        virtual iter_impl* clone() = 0;
        virtual ~iter_impl() {}
};
```
# The `List_Graph` Class

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;list&lt;Edge&gt; &gt; edges</code></td>
<td>A vector of lists to contain the edges that originate with each vertex.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>List_Graph(int n, bool d)</code></td>
<td>Constructs a graph with the specified number of vertices and directionality.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Public Member Functions</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterator begin(int source) const</code></td>
<td>Returns an iterator to the edges that originate from a given vertex.</td>
</tr>
<tr>
<td><code>iterator end(int source) const</code></td>
<td>Returns an iterator that is one past the edges that originate from a given vertex.</td>
</tr>
<tr>
<td><code>Edge get_edge(int source, int dest) const</code></td>
<td>Gets the edge between two vertices.</td>
</tr>
<tr>
<td><code>void insert(const Edge&amp; e)</code></td>
<td>Inserts a new edge into the graph.</td>
</tr>
<tr>
<td><code>bool is_edge(int source, int dest) const</code></td>
<td>Determines whether an edge exists from vertex source to vertex dest.</td>
</tr>
</tbody>
</table>
The List_Graph Class (cont.)

```cpp
#ifndef LIST_GRAPH_H
#define LIST_GRAPH_H

#include "Graph.h"
#include <list>
#include <vector>
using std::list;
using std::vector;

/** A List_Graph is an implementation of the Graph abstract class that uses a vector of lists to contain the edges adjacent to a given vertex. */
class List_Graph : public Graph {

public:

    // Constructors
    /** Constructs a graph with the specified number of vertices and directionality. 
     * @param n The number of vertices 
     * @param d The directionality flag
     */
    List_Graph(int n, bool d);

    // Declaration of abstract functions.
    /** Insert a new edge into the graph. 
     * @param edge The new edge
     */
    void insert(const Edge& edge);

};
#endif
```
The List_Graph Class (cont.)

/** Determines whether an edge exists.
 * @param source The source vertex
 * @param dest The destination vertex
 * @return true if there is an edge from source to dest
 */
bool is_edge(int source, int dest) const;

/** Get the edge between two vertices. If an edge does not exist, an Edge with a weight of numeric_limits<double>::infinity() is returned.
 * @param source The source vertex
 * @param dest The destination vertex
 * @return The edge between these two vertices
 */
Edge get_edge(int source, int dest) const;

/** Return an iterator to the edges adjacent to a given vertex.
 * @param source The source vertex
 * @return An iterator positioned at the beginning of the vertices adjacent to source
 */
iterator begin(int source) const;
/** Return an iterator that is one past the edges adjacent to a given vertex. */
@param source The source vertex
@return An iterator positioned one past the edges adjacent to source

iterator end(int source) const;

private:

// Data field
/** The vector of lists of edges */
std::vector<std::list<Edge> > edges;

public:

// iter_impl class
...

}; // end List_Graph

#endif
The Constructor

/** Constructs a graph with the specified number of vertices and directionality.
 * @param n The number of vertices
 * @param d The directionality flag
 */
List_Graph::List_Graph(int n, bool d) : Graph(n, d) {
    edges.resize(n);
}
/** Determines whether an edge exists. 
   @param source The source vertex  
   @param dest The destination vertex  
   @return true if there is an edge from source to dest */ 
bool List_Graph::is_edge(int source, int dest) const { 
  list<Edge>::const_iterator itr = find(edges[source].begin(), 
                                      edges[source].end(), 
                                      Edge(source, dest)); 
  return itr != edges[source].end(); 
}
/** Get the edge between two vertices. If an edge does not exist, an Edge with a weight of numeric_limits<double>::infinity() is returned. */

Edge List_Graph::get_edge(int source, int dest) const {
  list<Edge>::const_iterator itr = find(edges[source].begin(), edges[source].end(), Edge(source, dest));
  if (itr != edges[source].end())
    return *itr;
  else
    return Edge(source, dest, numeric_limits<double>::infinity());
}
The `insert` Function

```cpp
/**
 * Insert a new edge into the graph.
 * @param edge The new edge
 */
void List_Graph::insert(const Edge& edge) {
    edges[edge.get_source()].push_back(edge);
    if (!is_directed()) {
        edges[edge.get_dest()].push_back(Edge(edge.get_dest(),
                                              edge.get_source(),
                                              edge.get_weight()));
    }
}
```
The `begin` Function

```cpp
/**
 * Return an iterator to the edges adjacent to a given vertex.
 * @param source The source vertex
 * @return An iterator positioned at the beginning of the vertices adjacent to source
 */
Graph::iterator List_Graph::begin(int source) const {
    return Graph::iterator(new iter_impl(edges[source].begin()));
}
```
/** Return an iterator that is one past the edges
 adjacent to a given vertex.
 @param source The source vertex
 @return An iterator positioned one past the
     edges adjacent to source
 */

Graph::iterator List_Graph::end(int source) const {
    return Graph::iterator(new iter_impl(edges[source].end()));
}
The List_Graph::iter_impl Class

- The List_Graph::iter_impl class is a subclass of the Graph::iter_impl class
- Recall that the Graph::iter_impl class is abstract, and that all of its member functions are abstract
- The List_Graph::iter_impl class provides implementations of the minimum iterator functions that are defined for the Graph::iterator
- We designed the Graph::iterator this way to provide a common interface for iterators defined for different Graph implementations
- If we had only the List_Graph, we could use the list<Edge>::iterator directly as the Graph::iterator
The List_Graph::iter_impl Class (cont.)

- One major difference between the `Graph::iterator` and other iterator classes is the behavior of the dereferencing operator (`operator*()`).
- In other iterator classes we have shown, the dereferencing operator returns a reference to the object that the iterator refers to.
- Thus the iterator can be used to change the value of the object referred to. (This is why we define both an iterator and `const_iterator`.)
- The `Graph::iterator`, and thus the `iter_impl` classes, however, return a copy of the referenced `Edge` object.
- Thus changes made to an `Edge` via a `Graph::iterator` will not change the `Edge` within the graph.
The `List_Graph::iter_impl` Class (cont.)

```cpp
// TheList_Graph::iter_impl Class from List_Graph.h

/** Implementation class for an iterator to the edges.
 * 
 * class iter_impl : public Graph::iter_impl {

 private:
    // Constructor
    /** Construct an iter_impl for a given vertex.
    *     @param start An iterator to the list of edges adjacent to the desired vertex
    */
    iter_impl(std::list<Edge>::const_iterator start) : current(start) {} 

 public:
    /** Return the current edge */
    Edge operator*() { return *current; }

    /** Advance to the next edge */
    Graph::iter_impl& operator++() { ++current; return *this; }

    /** Determine whether two iter_impl objects are equal */
    bool operator==(const Graph::iter_impl& other) const {
        const iter_impl* ptr_other =
            dynamic_cast<const iter_impl*>(&other);
        if (ptr_other == NULL) return false;
        return current == ptr_other->current;
    }

    /** Make a deep copy of this iter_impl */
    Graph::iter_impl clone() { return new iter_impl(current); }

 private:
    // Data fields

    /** Iterator to the list of edges */
    std::list<Edge>::const_iterator current;
    friend class List_Graph;

}; // End iter_impl
```
The Matrix_Graph Class

- The Matrix_Graph class extends the Graph class by providing an internal representation using a two-dimensional array for storing edge weights.
- This array is implemented by dynamically allocating an array of dynamically allocated arrays:
  ```c
  double** edges;
  ```
- Upon creation of a Matrix_Graph object, the constructor sets the number of rows (vertices)
The Matrix\_Graph Class

- For a directed graph, each row is then allocated to hold the same number of columns, one for each vertex.
- For an undirected graph, only the lower diagonal of the array is needed.
- Thus the first row has one column, the second two, and so on.
- The is\_edge and get\_edge functions, when operating on an undirected graph, must test to see whether the destination is greater than the source, if it is, they then must access the row indicated by the destination and the column indicated by the source.
The Matrix_Graph Class

- The iter_impl class presents a challenge
- An iter_impl object must keep track of the current source (row) and current destination (column)
- The dereferencing operator (operator*) must then create and return an Edge object (This is why we designed the Graph::iterator to return an Edge value rather than an Edge reference)
- The other complication for the iter_impl class is the increment operator
- When this operator is called, the iterator must be advanced to the next defined edge, skipping those columns whose weights are infinity
- The implementation of the Matrix_Graph is left as a project
Comparing Implementations

- Time efficiency depends on the algorithm and the density of the graph.

- The density of a graph is the ratio of $|E|$ to $|V|^2$.
  - A dense graph is one in which $|E|$ is close to, but less than $|V|^2$.
  - A sparse graph is one in which $|E|$ is much less than $|V|^2$.

- We can assume that $|E|$ is
  - $O(|V|^2)$ for a dense graph
  - $O(|V|)$ for a sparse graph
Comparing Implementations (cont.)

Many graph algorithms are of the form:
1. for each vertex \( u \) in the graph
2. for each vertex \( v \) adjacent to \( u \)
3. Do something with edge \((u, v)\)

- For an adjacency list
  - Step 1 is \( O(|V|) \)
  - Step 2 is \( O(|E_u|) \)
    - \( E_u \) is the number of edges that originate at vertex \( u \)

- The combination of Steps 1 and 2 represents examining each edge in the graph, giving \( O(|E|) \)
Comparing Implementations (cont.)

Many graph algorithms are of the form:
1. for each vertex \( u \) in the graph
2. for each vertex \( v \) adjacent to \( u \)
3. Do something with edge \((u, v)\)

- For an adjacency matrix
  - Step 1 is \( O(|V|) \)
  - Step 2 is \( O(|V|) \)

- The combination of Steps 1 and 2 represents examining each edge in the graph, giving \( O(|V^2|) \)

- The adjacency list gives better performance in a sparse graph, whereas for a dense graph the performance is the same for both representations
Comparing Implementations (cont.)

Some graph algorithms are of the form:

1. for each vertex $u$ in some subset of the vertices
2. for each vertex $v$ in some subset of the vertices
3. if $(u, v)$ is an edge
4. Do something with edge $(u, v)$

- For an adjacency matrix representation,
  - Step 3 tests a matrix value and is $O(1)$
  - The overall algorithm is $O(|V^2|)$
Comparing Implementations (cont.)

Some graph algorithms are of the form:

1. For each vertex \( u \) in some subset of the vertices
2. For each vertex \( v \) in some subset of the vertices
3. if \((u, v)\) is an edge
4. Do something with edge \((u, v)\)

- For an adjacency list representation,
  - Step 3 searches a list and is \( O(|E_u|) \)
  - So the combination of Steps 2 and 3 is \( O(|E|) \)
  - The overall algorithm is \( O(|V| |E|) \)
Comparing Implementations (cont.)

Some graph algorithms are of the form:

1. \texttt{for} each vertex $u$ in some subset of the vertices
2. \texttt{for} each vertex $v$ in some subset of the vertices
3. if $(u, v)$ is an edge
4. Do something with edge $(u, v)$

- For a dense graph, the adjacency matrix gives better performance
- For a sparse graph, the performance is the same for both representations
Comparing Implementations (cont.)

- Thus, for time efficiency,
  - if the graph is dense, the adjacency matrix representation is better
  - if the graph is sparse, the adjacency list representation is better
- A sparse graph will lead to a sparse matrix, or one where most entries are infinity
- These values are not included in a list representation so they have no effect on the processing time
- They are included in a matrix representation, however, and will have an undesirable impact on processing time
Storage Efficiency

- In an adjacency matrix,
  - storage is allocated for all vertex combinations (or at least half of them)
  - the storage required is proportional to $|V|^2$
  - for a sparse graph, there is a lot of wasted space

- In an adjacency list,
  - each edge is represented by an Edge object containing data about the source, destination, and weight
  - there are also pointers to the next and previous edges in the list
  - this is five times the storage needed for a matrix representation (which stores only the weight)
  - if we use a single-linked list we could reduce this to four times the storage since the pointer to the previous edge would be eliminated
Comparing Implementations (cont.)

- The break-even point in terms of storage efficiency occurs when approximately 20% of the adjacency matrix is filled with meaningful data.
- That is, the adjacency list uses less (more) storage when less than (more than) 20 percent of the adjacency matrix would be filled.
Traversals of Graphs

Section 12.4
Algorithm for Breadth-First Search

1. Take an arbitrary start vertex, mark it identified (color it light blue), and place it in a queue.
2. while the queue is not empty
3.     Take a vertex, $u$, out of the queue and visit $u$.
4.     for all vertices, $v$, adjacent to this vertex, $u$
5.         if $v$ has not been identified or visited
6.             Mark it identified (color it light blue).
7.             Insert vertex $v$ into the queue.
8. We are now finished visiting $u$ (color it dark blue).
We can build a tree that represents the order in which vertices will be visited in a breadth-first traversal.

The tree has all of the vertices and some of the edges of the original graph.

A path starting at the root to any vertex in the tree is the shortest path in the original graph to that vertex (considering all edges to have the same weight).
We can save the information we need to represent the tree by storing the parent of each vertex when we identify it.

We refine Step 7 of the algorithm to accomplish this:

7.1 Insert vertex v into the queue
7.2 Set the parent of v to u
Performance Analysis of Breadth-First Search

- The loop at Step 2 is performed for each vertex.
- The inner loop at Step 4 is performed for $|E_v|$, the number of edges that originate at that vertex).
- The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges.
- The algorithm is $O(|E|)$. 
Implementing Breadth-First Search

```cpp
#include <vector>
#include <queue>
#include "Graph.h"
using namespace std;

/** Perform a breadth-first search of a graph.
   * The vector p will contain the predecessor of each
   * vertex in the breadth-first search tree.
   * @param graph The graph to be searched
   * @param start The start vertex
   * @return The vector of parents */
vector<int> breadth_first_search(const Graph& graph, int start) {
    int num_v = graph.get_num_v();
    queue<int> the_queue;
    vector<int> parent(num_v, -1);
    vector<bool> identified(num_v, false);
    identified[start] = true;
    the_queue.push(start);

    /* While the queue is not empty */
    while (!the_queue.empty()) {
        /* Take a vertex, current, out of the queue 
           (Begin visiting current).*/
        int current = the_queue.front();
        the_queue.pop();

        /* For all vertices, neighbor, adjacent to current */
        Graph::iterator itr = graph.begin(current);
        while (itr != graph.end(current)) {
            Edge edge = *itr;
            int neighbor = edge.get_dest();

            /* If neighbor has not been identified */
            if (!identified[neighbor]) {
                /* Mark it identified */
                identified[neighbor] = true;

                /* Place it into the queue */
                the_queue.push(neighbor);

                /* Insert the edge (current, neighbor) 
                   into the tree */
                parent[neighbor] = current;
            }
            ++itr;
        }

        // Finished visiting current.
    }

    return parent;
}
```
The method returns vector `parent` which can be used to construct the breadth-first search tree.

If we run the `breadth_first_search` function on the graph we just traversed, `parent` will be filled with the values shown on the right.
Implementing Breadth-First Search (cont.)

- If we compare vector parent to the top right figure, we see that parent[i] is the parent of vertex i.
- For example, the parent of vertex 4 is vertex 1.
- The entry parent[0] is –1 because node 0 is the start vertex.
Implementing Breadth-First Search (cont.)

- Although `vector parent` could be used to construct the breadth-first search tree, generally we are not interested in the complete tree but rather in the path from the root to a given vertex.

- Using `vector parent` to trace the path from that vertex back to the root gives the reverse of the desired path.

- The desired path is realized by pushing the vertices onto a stack, and then popping the stack until it is empty.
Depth-First Search

- In a depth-first search,
  - start at a vertex,
  - visit it,
  - choose one adjacent vertex to visit;
  - then, choose a vertex adjacent to that vertex to visit,
  - and so on until you can go no further;
  - then back up and see whether a new vertex can be found
Algorithm for Depth-First Search

1. Mark the current vertex, $u$, visited (color it light blue), and enter it in the discovery order list.
2. For each vertex, $v$, adjacent to the current vertex, $u$
3.     if $v$ has not been visited
4.         Set parent of $v$ to $u$.
5.     Recursively apply this algorithm starting at $v$.
6. Mark $u$ finished (color it dark blue) and enter $u$ into the finish order list.
Performance Analysis of Depth-First Search

- The loop at Step 2 is executed $|E_v|$ times.
- The recursive call results in this loop being applied to each vertex.
- The total number of steps is the sum of the edges that originate at each vertex, which is the total number of edges, $|E|.$
- The algorithm is $O(|E|).$
- An implicit Step 0 marks all of the vertices as unvisited – $O(|V|)$
- The total running time of the algorithm is $O(|V| + |E|).$
Implementing Depth-First Search

- The function `depth_first_search` performs a depth-first search on a graph and records the
  - start time
  - finish time
  - start order
  - finish order

- For an unconnected graph or for a directed graph, a depth-first search may not visit each vertex in the graph

- Thus, once the recursive method returns, all vertices need to be examined to see if they have been visited— if not the process repeats on the next unvisited vertex

- Thus, a depth-first search may generate more than one tree

- A collection of unconnected trees is called a forest
Implementing Depth-First Search (cont.)

```cpp
#include <vector>
#include "Graph.h"
using namespace std;

/** Perform a depth first search of a graph (recursive function).
   @param graph The graph to be searched
   @param current The current vertex being visited
   @param parent The parents in the depth-first search tree
   @param discovery_order The discovery order for each vertex
   @param finish_order The finish order for each vertex
   @param visited The vector that records whether a vertex has been visited
   @param discovery_index The index into the discovery_order vector
   @param finish_index The index into the finish_order vector
*/
void depth_first_search(const Graph& graph, int current,
                        vector<int>& parent,
                        vector<int>& discovery_order,
                        vector<int>& finish_order,
                        vector<bool>& visited,
```
Implementing Depth-First Search (cont.)

```c++
int& discovery_index,
int& finish_index) {

visited[current] = true;
discovery_order[discovery_index++] = current;
/* For each vertex adjacent to the current vertex. */
for (Graph::iterator itr = graph.begin(current);
   itr != graph.end(current); ++itr) {
   int neighbor = (*itr).get_dest();
   // if neighbor has not been visited
   if (!visited[neighbor]) {
      /* Insert (current, neighbor) into the depth-first search tree */
      parent[neighbor] = current;
      // Recursively apply the algorithm starting at neighbor.
      depth_first_search(graph, neighbor,
                         parent, discovery_order,
                         finish_order, visited,
                         discovery_index, finish_index);
   }
}
// Mark current finished
finish_order[finish_index++] = current;
}

/** Perform a depth-first search of a graph (starter function).
 * @param graph The graph to be searched
 * @param start The start vertex
 * @param parent The parents in the depth-first search tree
 * @param discovery_order The discovery order for each vertex
 * @param finish_order The finish order for each vertex
 */
```
Implementing Depth-First Search (cont.)

```cpp
void depth_first_search(const Graph& graph, int start,
        vector<int>& parent,
        vector<int>& discovery_order,
        vector<int>& finish_order) {

    int num_v = graph.get_num_v();
    parent.clear();
    parent.resize(num_v, -1);
    discovery_order.clear();
    discovery_order.resize(num_v, -1);
    finish_order.clear();
    finish_order.resize(num_v, -1);
    vector<bool> visited(num_v, false);
    int discovery_index = 0;
    int finish_index = 0;
    for (int i = 0; i < num_v; i++) {
        if (!visited[i]) {
            depth_first_search(graph, i, parent,
                discovery_order, finish_order, visited,
                discovery_index, finish_index);
        }
    }
}
```
/** Main function to demonstrate the algorithm. 
pre: argc[1] is the name of the input file.
pre: argc[2] is the type of graph.
@param argc The count of command line arguments
@param argv The command line arguments
*/
int main(int argc, char* argv[]) {
    if (argc < 3) {
        cerr << "Usage Depth_First_Search <input> <graph type>\n";
        return 1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "Unable to open " << argv[1] << " for input\n";
        return 1;
    }
    Graph* g = Graph::create_graph(in, false, "List");
    vector<int> parent;
    vector<int> discovery_order;
    vector<int> finish_order;
    depth_first_search(*g, 0, parent, discovery_order, finish_order);
    cout << setw(4) << "i";
    cout << setw(8) << "discovery_order";
    cout << setw(8) << "finish_order";
    cout << setw(8) << "parent";
    cout << endl;
    for (int i = 0; i < g->get_num_v(); i++) {
        cout << setw(4) << i;
        cout << setw(8) << discovery_order[i];
        cout << setw(8) << finish_order[i];
        cout << setw(8) << parent[i];
        cout << endl;
    }
}
Application of Graph Traversals

Section 12.5
Problem

- Design a program that finds the shortest path through a maze
- A recursive solution is not guaranteed to find an optimal solution
- (On the next slide, you will see that this is a consequence of the program advancing the solution path to the south before attempting to advance it to the east)
- We want to find the shortest path (defined as the one with the fewest decision points in it)
Problem (cont.)
Analysis

- We can represent the maze on the previous slide as a graph, with a node at each decision point and each dead end.
With the maze represented as a graph, we need to find the shortest path from the start point (vertex 0) to the end point (vertex 12).

The breadth-first search method returns the shortest path from each vertex to its parents (the vector of parent vertices).

We use this vector to find the shortest path to the end point which will contain

- the smallest number of vertices
- but not necessarily the smallest number of cells
Design

- The program needs the following data structures:
  - an external representation of the maze, consisting of the number of vertices and the edges
  - an object of a class that implements the Graph interface
  - a vector to hold the predecessors returned from the breadth_first_search function
  - A stack to reverse the path
Algorithm for Shortest Path

1. Read in the number of vertices and create the graph object.
2. Read in the edges and insert the edges into the graph.
3. Call the `breadth_first_search` function with this graph and the starting vertex as its argument. The function returns the vector `parent`.
4. Start at \( v \), the end vertex.
5. \textbf{while} \( v \) is not \(-1\)
6. \hspace{1em} Push \( v \) onto the stack.
7. \hspace{1em} Set \( v \) to `parent`[\( v \)].
8. \textbf{while} the stack is not empty
9. \hspace{1em} Pop a vertex off the stack and output it.
Implementation

#include <iostream>
#include <fstream>
#include <vector>
#include <stack>
#include "Graph.h"

using namespace std;

vector<int> breadth_first_search(const Graph&, int);

/** Program to solve a maze represented as a graph.
   This program performs a breadth-first search of the graph to
   find the "shortest" path from the start vertex to the end. It is
   assumed that the start vertex is 0, and the end vertex is num_v - 1.
   @param argc Count of command line arguments
   @param argv The command line arguments
   @pre argv[1] Contains the name of the input file
   @pre argv[2] Contains the type of graph
*/
int main(int argc, char* argv[]) {
    if (argc < 3) {
        cerr << "Usage Maze <input> <graph type>\n";
        return 1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "Unable to open " << argv[1] << " for input\n";
        return 1;
    }
    Graph* the_maze = Graph::create_graph(in, false, "List");
    // Perform breadth-first search
    vector<int> parent = breadth_first_search(*the_maze, 0);
    // Construct the path
    stack<int> the_path;
    int v = the_maze->get_num_v() - 1;
    while (parent[v] != -1) {
        the_path.push(v);
        v = parent[v];
    }
    // Output the path
    cout << "The Shortest path is:\n";
    while (!the_path.empty()) {
        cout << the_path.top() << endl;
        the_path.pop();
    }
    return 0;
}
Testing

- Test the program with a variety of mazes.
- Use mazes for which the original recursive program finds the shortest path and those for which it does not.
This is an example of a directed acyclic graph (DAG)

DAGs simulate problems in which one activity cannot be started before another one has been completed

It is a directed graph which contains no cycles (i.e. no loops)

Once you pass through a vertex, there is no path back to the vertex
Another Directed Acyclic Graph (DAG)
Topological Sort of a Graph (cont.)

- A topological sort of the vertices of a DAG is an ordering of the vertices such that if \((u, v)\) is an edge, then \(u\) appears before \(v\).
- This must be true for all edges.
- There may be many valid paths through a DAG and many valid topographical sorts of a DAG.
0, 1, 2, 3, 4, 5, 6, 7, 8 is a valid topological sort
but 0, 1, 5, 3, 4, 2, 6, 7, 8 is not
Another valid topological sort is
0, 3, 1, 4, 6, 2, 5, 7, 8
Analysis

- If there is an edge from $u$ to $v$ in a DAG,
  - then if we perform a depth-first search of the graph
  - the finish time of $u$ must be after the finish time of $v$
- When we return to $u$, either $v$ has not been visited or it has finished
- It is not possible for $v$ to be visited but not finished (a loop or cycle would exist)
We start the depth first search at 0
Then visit 4
Followed by 6
Followed by 8
Then return to 4
Analysis (cont.)

Visit 7
Then we are able to return to 0.
Then we visit 1
We see that 4 has finished and continue on....
If we perform a depth-first search of a graph and then order the vertices by the inverse of their finish order, we will have one topological sort of a directed acyclic graph.
The topological sort produced by listing the vertices in the inverse of their finish order after a depth-first search of the graph to the right is

0, 3, 1, 4, 6, 2, 5, 7, 8
Algorithm for Topological Sort

1. Read the graph from a data file
2. Perform a depth-first search of the graph
3. List the vertices in reverse of their finish order
Implementation

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include "Graph.h"
using namespace std;

void depth_first_search(const Graph&, int,
    vector<int>&, vector<int>&,
    vector<int>&);

/** This program outputs the topological sort of a directed graph
that contains no cycles.
pre: argv[1] will contain the file name that contains the graph.
pre: argv[2] will contain the type of graph representation.
@param argc The count of command line arguments
@param argv The command line arguments
*/
int main(int argc, char* argv[]) {
    if (argc < 3) {
        cerr << "Usage Topological_Sort <input> <graph type>\n";
        return 1;
    }
    ifstream in(argv[1]);
    if (!in) {
        cerr << "Unable to open " << argv[1] << " for input\n";
        return 1;
    }
    Graph* the_graph = Graph::create_graph(in, true, argv[2]);
    // Perform the depth-first search
    vector<int> parent;
    vector<int> discovery_order;
    vector<int> finish_order;
    depth_first_search(*the_graph, 0, parent, discovery_order,
        finish_order);
    cout << "The Topological Sort is\n";
    for (int i = the_graph->get_num_v() - 1; i >= 0; i--)
        cout << finish_order[i] << endl;
    return 0;
```
Testing

- Test the program on several different graphs
- Use sparse graphs and dense graphs
- Avoid graphs with loops or cycles
Algorithms Using Weighted Graphs

Section 12.6
Dijkstra's Algorithm (cont.)

Dijkstra's Algorithm

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. For all $v$ in $V-S$
3.  
   Set $p[v]$ to $s$.
4.  
   If there is an edge $(s, v)$
5.   
      Set $d[v]$ to $w(s, v)$.
6.  
   Else
7.   
      Set $d[v]$ to $\infty$.
8. While $V-S$ is not empty
9.  
   For all $u$ in $V-S$, find the smallest $d[u]$.
10.  
    Remove $u$ from $V-S$ and add $u$ to $S$.
11.  
    For all $v$ adjacent to $u$ in $V-S$
12.  
    If $d[u] + w(u, v)$ is less than $d[v]$.
13.  
    Set $d[v]$ to $d[u] + w(u, v)$.
14.  
    Set $p[v]$ to $u$. 

Analysis of Dijkstra's Algorithm

Dijkstra's Algorithm

1. Initialize $S$ with the start vertex, $s$, and $V - S$ with the remaining vertices.
2. for all $v$ in $V - S$
4.   if there is an edge $(s, v)$
5.     Set $d[v]$ to $w(s, v)$.
6.   else
7.     Set $d[v]$ to $\infty$.
8. while $V - S$ is not empty
9.   for all $u$ in $V - S$, find the smallest $d[u]$.
10. Remove $u$ from $V - S$ and add $u$ to $S$.
11. for all $v$ adjacent to $u$ in $V - S$
12.   if $d[u] + w(u, v)$ is less than $d[v]$.

Step 1 requires $|V|$ steps
Analysis of Dijkstra's Algorithm (cont.)

Dijkstra's Algorithm

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
4.     if there is an edge $(s, v)$
5.         Set $d[v]$ to $w(s, v)$.
6.     else
7.         Set $d[v]$ to $\infty$.
8. while $V-S$ is not empty
9.     for all $u$ in $V-S$, find the smallest $d[u]$.
10. Remove $u$ from $V-S$ and add $u$ to $S$.
11. for all $v$ adjacent to $u$ in $V-S$
12.     if $d[u] + w(u, v)$ is less than $d[v]$.

The loop at Step 2 is executed $|V-1|$ times.
The loop at Step 7 also is executed $|V-1|$ times.
Analysis of Dijkstra's Algorithm (cont.)

Dijkstra's Algorithm

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
4.   if there is an edge $(s,v)$
5.     Set $d[v]$ to $w(s,v)$.
6.   else
7.     Set $d[v]$ to $\infty$.
8. while $V-S$ is not empty
9.     for all $u$ in $V-S$, find the smallest $d[u]$.
10.    Remove $u$ from $V-S$ and add $u$ to $S$.
11.    for all $v$ adjacent to $u$ in $V-S$
12.      if $d[u] + w(u,v)$ is less than $d[v]$.

Steps 8 and 9 search each value in $V-S$, which decreases each time through loop 7:
$V| - 1 + |V| - 2 + \cdots 1$
This is $O(|V|^2)$. 
Dijkstra's Algorithm

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
   4.     if there is an edge $(s, v)$
   5.         Set $d[v]$ to $w(s, v)$.
   else
   6.         Set $d[v]$ to $\infty$.
7. while $V-S$ is not empty
   8.     for all $u$ in $V-S$, find the smallest $d[u]$.
   9.     Remove $u$ from $V-S$ and add $u$ to $S$.
10. for all $v$ adjacent to $u$ in $V-S$
11.     if $d[u] + w(u, v)$ is less than $d[v]$.

Dijkstra's Algorithm is $O(|V|^2)$
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include "Graph.h"

using namespace std;

/** Dijkstra's Shortest-Path algorithm.
 * @param graph The weighted graph to be searched
 * @param start The start vertex
 * @param pred Output vector to contain the predecessors in the shortest path
 * @param dist Output vector to contain the distance in the shortest path
 */
void dijkstras_algorithm(const Graph& graph,
                         int start,
                         vector<int>& pred,
                         vector<double>& dist) {

    int num_v = graph.get_num_v();
    // Use a set to represent V - S
    set<int> v_minus_s;
    // Initialize V - S.
    for (int i = 0; i < num_v; i++) {
        if (i != start) {
            v_minus_s.insert(i);
        }
    }
    // Initialize pred and dist
    for (set<int>::iterator itr = v_minus_s.begin();
         itr != v_minus_s.end(); ++itr) {
        pred[*itr] = start;
        dist[*itr] = graph.get_edge(start, *itr).get_weight();
    }
// Main loop
while (!v_minus_s.empty()) {
    // Find the value u in V - S with the smallest dist[u].
    double min_dist = numeric_limits<double>::infinity();
    int u = -1;
    for (set<int>::iterator itr = v_minus_s.begin();
         itr != v_minus_s.end(); ++itr) {
        int v = *itr;
        if (dist[v] < min_dist) {
            min_dist = dist[v];
            u = v;
        }
    }
    // Remove u from v_minus_s
    v_minus_s.erase(u);
    // Update the distances
    for (set<int>::iterator itr = v_minus_s.begin();
         itr != v_minus_s.end(); ++itr) {
        int v = *itr;
        if (graph.is_edge(u, v)) {
            double weight = graph.get_edge(u, v).get_weight();
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pred[v] = u;
            }
        }
    }
}
...
For an adjacency list representation, modify the code:

```cpp
    // Update the distances
    for (Graph::iterator itr = graph.begin(u);
         itr != graph.end(u); ++itr) {
        Edge edge = *itr;
        int v = edge.get_dest();
        if (contains(v_minus_s, v)) {
            double weight = edge.get_weight();
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pred[v] = u;
            }
        }
    }
```
Minimum Spanning Trees

- A spanning tree is a subset of the edges of a graph such that there is only one edge between any two vertices, and all of the vertices are connected.
- If we have a spanning tree for a graph, then we can access all the vertices of the graph from the start node.
- The cost of a spanning tree is the sum of the weights of the edges.
- We want to find the minimum spanning tree or the spanning tree with the smallest cost.
If we want to start up our own long-distance phone company and need to connect the cities shown below, finding the minimum spanning tree would allow us to build the cheapest network.

The solution to this problem was formulated by R.C. Prim and is very similar to Dijkstra’s algorithm.
Overview of Prim's Algorithm

- The vertices are divided into two sets:
  - S, the set of vertices in the spanning tree
  - V-S, the remaining vertices
- As in Dijkstra's algorithm, we maintain two vectors,
  - d[v] contains the length of the shortest edge from a vertex in S to the vertex v that is in V-S
  - p[v] contains the source vertex for that edge
- The only difference between the two algorithms is the contents of d[v]; in Prim’s algorithm, d[v] contains only the length of the final edge
Prim’s Algorithm for Finding the Minimum Spanning Tree

1. Initialize $S$ with the start vertex, $s$, and $V−S$ with the remaining vertices.
2. for all $v$ in $V−S$
4.     if there is an edge $(s, v)$
5.         Set $d[v]$ to $w(s, v)$.
6.     else
7.         Set $d[v]$ to $\infty$.
8. while $V−S$ is not empty
9.     for all $u$ in $V−S$, find the smallest $d[u]$.
10. Remove $u$ from $V−S$ and add it to $S$.
11. Insert the edge $(u, p[u])$ into the spanning tree.
12. for all $v$ in $V−S$
13.     if $w(u, v) < d[v]$      

Analysis of Prim's Algorithm

Prim's Algorithm for Finding the Minimum Spanning Tree

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
4. if there is an edge $(s, v)$
5.   Set $d[v]$ to $w(s, v)$.
6. else
7.   Set $d[v]$ to $\infty$.
8. while $V-S$ is not empty
9.   for all $u$ in $V-S$, find the smallest $d[u]$.
10. Remove $u$ from $V-S$ and add it to $S$.
11. Insert the edge $(u, p[u])$ into the spanning tree.
12. for all $v$ in $V-S$
13.   if $w(u, v) < d[v]$  

Step 8 is $O(|V|)$ and is within loop 7, so it is executed $O(|V|)$ times for a total time of $O(|V|^2)$.
Step 11 is $O(|E_u|)$, and is executed for all vertices for a total of $O(|E|)$.
Analysis of Prim's Algorithm (cont.)

**Prim's Algorithm for Finding the Minimum Spanning Tree**

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
3. \quad Set $p[v]$ to $s$.
4. \quad if there is an edge $(s, v)$
5. \quad \quad Set $d[v]$ to $w(s, v)$.
6. \quad else
7. \quad \quad Set $d[v]$ to $\infty$.
8. while $V-S$ is not empty
9. \quad for all $u$ in $V-S$, find the smallest $d[u]$.
10. \quad Remove $u$ from $V-S$ and add it to $S$.
11. \quad Insert the edge $(u, p[u])$ into the spanning tree.
12. \quad for all $v$ in $V-S$
13. \quad \quad if $w(u, v) < d[v]$
14. \quad \quad \quad Set $d[v]$ to $w(u, v)$.
15. \quad \quad Set $p[v]$ to $u$.

The overall cost is $O(|V|^2)$

($|V|^2$ is greater than $|E|$)
Analysis of Prim's Algorithm (cont.)

Prim's Algorithm for Finding the Minimum Spanning Tree

1. Initialize $S$ with the start vertex, $s$, and $V-S$ with the remaining vertices.
2. for all $v$ in $V-S$
   4. if there is an edge $(s, v)$
      5. Set $d[v]$ to $w(s, v)$.
   else
      6. Set $d[v]$ to $\infty$.
7. while $V-S$ is not empty
   8. for all $u$ in $V-S$, find the smallest $d[u]$.
   9. Remove $u$ from $V-S$ and add it to $S$.
10. Insert the edge $(u, p[u])$ into the spanning tree.
11. for all $v$ in $V-S$
   12. if $w(u, v) < d[v]$
      13. Set $d[v]$ to $w(u, v)$.

Using a priority queue can reduce Step 8 to $O(|E| \log |V|)$ where $n$ is the size of the priority queue, the algorithm becomes $O(|E| \log |V|)$. 
Analysis of Prim's Algorithm (cont.)

- Using a priority queue, in the worst case, all of the edges are inserted into the priority queue making the overall cost of the algorithm \( O(|E| \log|V|) \).
- We say that the algorithm is
  - \( O(|E| \log|V|) \)
  - instead of saying that it is \( O(|E| \log|E|) \),
  - even though the maximum size of the priority queue is \( |E| \),
  - because \( |E| \) is bounded by \( |V|^2 \) and \( \log|V|^2 \) is \( 2 \times \log|V| \).
Analysis of Prim's Algorithm (cont.)

- For a dense graph (where $|E|$ is approximately $|V|^2$), this is not an improvement; for a sparse graph, it is an improvement.
- Researchers have developed an improved priority queue implementation that gives $O(|E| + |V|\log|V|)$ or better performance.
Implementation

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <set>
#include <queue>
#include "Graph.h"
#include "set_functions.h"

using namespace std;

/** Comparator function class to compare Edge weights. */
struct Compare_Edges {
  typedef Edge value_type;
  bool operator()(const Edge& left, const Edge& right) {
    return left.get_weight() < right.get_weight();
  }
};

/** Prim's Minimum Spanning Tree algorithm.
   * @param graph The weighted graph to be searched
   * @param start The start vertex
   * @return A vector of edges that forms the MST
   */
vector<Edge> prims_algorithm(const Graph& graph,
                             const int start) {
  vector<Edge> result;
  int num_v = graph.get_num_v();
  // Use a set to represent V - S
  set<int> v_minus_s;
  // Declare the priority queue
  priority_queue<Edge, vector<Edge>, Compare_Edges> pQ;
  // Initialize V - S.
  for (int i = 0; i < num_v; i++) {
    if (i != start) {
      v_minus_s.insert(i);
    }
  }
  int current = start;
  ```
Implementation

```cpp
// Main loop
while (!v_minus_s.empty()) {
    // Update priority queue
    Graph::iterator iter = graph.begin(current);
    while (iter != graph.end(current)) {
        Edge edge = *iter++;
        int dest = edge.get_dest();
        if (contains(v_minus_s, dest)) {
            PQ.push(edge);
        }
    }

    // Find the shortest edge whose source is in S and
    // destination is in V - S.
    int dest;
    Edge edge;
    do {
        edge = PQ.top();
        PQ.pop();
        dest = edge.get_dest();
    } while (!contains(v_minus_s, dest));

    // Take dest out of v_minus_s
    v_minus_s.erase(dest);
    // Add edge to result
    result.push_back(edge);
    // Make this the current vertex
    current = dest;
}
return result;
...
```