A Generic All-Pairs Shortest-Paths Algorithm

Lauren Foutz Scott Hill

April 15, 2003

Abstract

The problem of finding the shortest path from one point to another arises in such diverse areas as Internet routing and navigation. There are numerous graph algorithms that are used to solve this problem, which appears in several variations depending on whether one wants shortest paths for some or all of the pairs of points. For the All-Pairs Shortest-Paths problem on a dense graph, the Floyd-Warshall algorithm is usually best, with a running time of $\Theta(V^3)$, where V is the number of points (graph vertices). This paper presents an implementation of Floyd-Warshall as a generic algorithm in C++ using the Boost Graph Library (BGL), along with results of experiments to test its correctness and compare its performance with alternatives already available in BGL such as Johnson's algorithm (which has a theoretical bound of $\Theta(VE \log V)$, where E is the number of edges, and is thus more suitable for sparse graphs) and repeated applications of the Bellman-Ford single-source shortest-paths algorithm (which has a running time of $\Theta(VE)$ for one source and thus takes $\Theta(V^4)$ time for all pairs on a dense graph). The presentation includes a user's guide and reference manual for the implementation. Additional sections describe in detail the design issues of a generic version of the algorithm, the testing approach and results, and plans for future improvements to the implementation.

Contents

1 User's Guide

1.1 Overview

This paper presents a generic version of the Floyd-Warshall All-Pairs Shortest-Paths algorithm, implemented in C++ using the Boost Graph Library (BGL) [1]. As such, this is intended for those who are familiar with BGL and C++. Specifically, it is intended for programmers with experience using generic programming methods (such as templates and STL, including generic comparison functions, vectors, and maps), and with familiarity with graph computation and terminology (such as vertices, edges, weights, and cycles). Extensive knowledge of BGL is not necessary, but familiarity with the library is recommended.

We used a generic programming approach in designing the software. This makes our algorithm extremely useful in almost any problem involving finding the shortest path between every pair of points in a graph. We also did this to maintain consistency with the design of other algorithms already implemented in BGL (such as the Johnson All-Pairs Shortest-Paths algorithm, and the Bellman-Ford Single-Source Shortest-Paths algorithm) [2]. Finally, the generic approach allowed us to make changes while implementing the algorithm with ease, and allows for extensions to the code to be made more easily.

The All-Pairs Shortest-Paths problem, specifically, involves finding the shortest path between every pair of vertices on a graph. The shortest overall path is dependent upon the weights that each path has. Negative weights can be present, but negative cycles are not permitted, because a vertex cannot be less than 0 distance from itself. Users can pass in a matrix, which can contain the shortest distance between all vertices in the graph; or an edge map, which can contain a list of the edges and their associated weights, with which the matrix can be initialized.

1.2 Tutorial

This tutorial is designed to help you apply the Floyd-Warshall function to your application. It gives two sample programs demonstrating the use of the function, and explains how it can be modified for more generic use.

The following sample program applies the Floyd-Warshall algorithm for all-pairs shortest paths to an example graph from Gosper [3].

"Floyd_Warshall_example.cpp" 3 ≡

```
#include <map>
#include <algorithm>
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include "Floyd_Warshall_all_pairs_shortest.hpp"
```

```
using namespace boost;

⟨For adding numbers to infinity 4⟩

int main() {

  ⟨Define the vertices, edges, and weights 5a⟩

  ⟨Define and initialize the graph 5b⟩

  ⟨Define the distance matrix 5c⟩

  ⟨Define the compare and combine functions 6a⟩

  ⟨Call the Floyd Warshall function 6b⟩

  ⟨Check for negative cycles 6c⟩

  ⟨Display results 7a⟩

  return 0;
}
```

The sample program demonstrates a simple but effective use of the algorithm. It is broken down into parts below, so you can understand how to utilize the function for your own application.

If you are using a function of your own to add weight values together, it is important to implement a functor to address the issue of adding a number to infinity, which isn't always handled correctly. The functor ensures that two values are not added together if one of them is infinity. If one of them is infinity, it simply returns infinity, and avoids the possible complications of adding a non-infinite value to infinity.

⟨For adding numbers to infinity 4⟩ ≡

```
template <typename T>
struct inf_plus {
  T operator()(const T& a, const T& b) const {
    T inf = std::numeric_limits<T>::max();
    if (a == inf || b == inf)
      return inf;
    return a + b;
  }
};
```

Used in parts 3, 7b.

The vertices, edges, and weights are defined here. For simplicity, the vertices are referenced in an enum as A, B, C, D, E. The weights correspond to each edge, so for example, the first weight (3) corresponds to the first edge (A, B).

⟨Define the vertices, edges, and weights 5a⟩ ≡

```
enum { A, B, C, D, E };
typedef std::pair < int, int > edge;
const int num_edges = 9;
edge edge_array[num_edges] = { edge(A, B), edge(A, C), edge(A, D), edge(B, D),
  edge(B, E), edge(C, B), edge(D, E), edge(E, A), edge(E, C) };
int i, j, weight[num_edges] = { 3, 8, -4, 1, 7, 4, 6, 2, -5 };
```

Used in parts 3, 7b.

The graph is initialized here. The graph must be a model of VertexAndEdgeListGraph (or VertexListGraph to call the initialized function—more on that shortly). After the graph is created (initialized with the number of nodes in the graph), the pre-defined edges are added to the graph, and a weight map is created and initialized with the pre-defined weights.

⟨Define and initialize the graph 5b⟩ ≡

```
typedef adjacency_matrix<directedS, property<vertex_distance_t,
  int, property<vertex_name_t, int> > , property<edge_weight_t, int> > Graph;
Graph g(5);

for (std::size_t k = 0; k < num_edges; ++k)
  add_edge(edge_array[k].first, edge_array[k].second, g);

graph_traits<Graph>::edge_iterator e, e_end;
property_map<Graph, edge_weight_t>::type weight_pmap = get(edge_weight, g);
for (i = 0, tie(e, e_end) = edges(g); e != e_end; ++e, ++i)
  weight_pmap[*e] = weight[i];
```

Used in parts 3, 7b.

The distance matrix is defined here. Notice that it is not initialized—only if you were to use a graph that is a model of VertexListGraph would you need to initialize the matrix yourself before passing it into the function. Otherwise, the matrix will be initialized in the function, whether or not it is initialized beforehand.

⟨Define the distance matrix 5c⟩ ≡

```
typedef graph_traits<Graph>::vertex_descriptor vertex_des;
std::map<vertex_des, std::map<vertex_des, int> > matrix;
graph_traits<Graph>::vertex_iterator first, last, first2, last2;
```

Used in part 3.

The compare and combine functions are defined here. These functions are optional—if they are not provided, default functions based upon the value type of the weight map are used instead.

⟨Define the compare and combine functions 6a⟩ ≡

```
std::less<int> compare;
inf_plus<int> combine;
```

Used in parts 3, 7b.

Once everything is defined and initialized, the Floyd_Warshall function itself can be called. Notice that named parameters are used, hence the only two required values are the graph and the distance matrix. All other parameters are optional, and can be passed into the function in any order (by using BGL's "named parameter" feature). Defaults based upon the type of the weight map will be used in place of any parameter not specified by the user. If the weight map is not provided, it will be extracted from the graph.

⟨Call the Floyd Warshall function 6b⟩ ≡

```
bool valid = Floyd_Warshall_all_pairs_shortest_paths(g, matrix, weight_map
  (weight_pmap).distance_compare(compare).distance_combine(combine)
  .distance_inf(std::numeric_limits<int>::max()).distance_zero(0));
```

Used in part 3.

If you a using a graph that is a model of VertexListGraph, the distance matrix must be initialized, and the weight map will be used only to determine the value type for the other parameters. You must also call the initialized Floyd_Warshall function in place of the non-initialized one. An example of this usage is to follow.

The function will return a boolean value based on whether or not it was successful. If it was not successful, it is because a negative cycle was detected, which is not allowed. This should be checked before analyzing the results, since the values in the distance matrix will not reflect a valid solution:

⟨Check for negative cycles 6c⟩ ≡

```
if (!valid) {
  std::cout<< "Error - Negative cycle in matrix" << std::endl;
  return 1;
}
```

Used in parts 3, 7b.

If the function was successful, the distance matrix will contain the results—that is, it will contain the shortest path from each vertex to every other vertex in the graph. For example, `matrix[A][B]` will contain the shortest path from vertex `A` to vertex `B`. The results can be displayed in a variety of ways, but for simplicity, it is done as follows:

⟨Display results 7a⟩ ≡

```
i = j = 1;
std::cout << "The Matrix: " << std::endl;
for (tie(first, last) = vertices(g); first != last; first++, i++) {
  for (j = 1, tie(first2, last2) = vertices(g); first2 != last2; first2++, j++) {
    std::cout << "From vertex " << i << " to " << j << " : ";
    if(matrix[*first][*first2] == std::numeric_limits<int>::max())
      std::cout << "inf" << std::endl;
    else
      std::cout << matrix[*first][*first2] << std::endl;
  }
}
```

Used in parts 3, 7b.

As previously mentioned, this sample program is not designed for users who wish to pass in a matrix that has previously been initialized. A similar function, Floyd_Warshall_initialized_all_pairs_shortest_paths, should be used instead. The following sample program, similar to the previous one, initializes the same graph, but also initializes the distance matrix and calls the aforementioned function.

"Floyd_Warshall_initialized_example.cpp" 7b ≡

```
#include <map>
#include <algorithm>
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include "Floyd_Warshall_all_pairs_shortest.hpp"

using namespace boost;

⟨For adding numbers to infinity 4⟩

int main() {

  ⟨Define the vertices, edges, and weights 5a⟩

  ⟨Define and initialize the graph 5b⟩

  ⟨Define and initialize the distance matrix 8a⟩
```

```
  ⟨Define the compare and combine functions 6a⟩

  ⟨Call the Floyd_Warshall_initialized function 8b⟩

  ⟨Check for negative cycles 6c⟩

  ⟨Display results 7a⟩

  return 0;
}
```

When the matrix is initialized, notice how the entire distance matrix is first set to infinity, which in this case, is represented by the maximum possible integer. Also notice that afterward, all paths from a vertex to itself is set to 0. This initialization is utilized in the Floyd-Warshall function, and we recommend you utilize it as well if you choose to initialize your matrix. The min function is in the for loop to ensure that in the case of parallel edges, the minimum weighted edge is used.

⟨Define and initialize the distance matrix 8a⟩ ≡

```
typedef graph_traits<Graph>::vertex_descriptor vertex_des;
std::map<vertex_des, std::map<vertex_des, int> > matrix;
graph_traits<Graph>::vertex_iterator first, last, first2, last2;
int inf = std::numeric_limits<int>::max();

for(tie(first, last) = vertices(g); first != last; first++)
  for(tie(first2, last2) = vertices(g); first2 != last2; first2++)
    matrix[*first][*first2] = inf;

for(tie(first, last) = vertices(g); first != last; first++)
  matrix[*first][*first] = 0;

for(tie(e, e_end) = edges(g); e != e_end; e++) {
  if (matrix[source(*e, g)][target(*e, g)] != inf)
    matrix[source(*e, g)][target(*e, g)] = std::min(weight_pmap[*e],
      matrix[source(*e, g)][target(*e, g)]);
  else
    matrix[source(*e, g)][target(*e, g)] = weight_pmap[*e];
}
```

Used in part 7b.

The actual function call for obtaining shortest paths is the same as in the first program, with the exception of the name of the function that is called:

⟨Call the Floyd_Warshall_initialized function 8b⟩ ≡

```
bool valid = Floyd_Warshall_initialized_all_pairs_shortest_paths(g, matrix,
  weight_map(weight_pmap).distance_compare(compare).distance_combine(combine)
  .distance_inf(std::numeric_limits<int>::max()).distance_zero(0));
```

Used in part 7b.

There are a couple things to keep in mind when using these functions. First, although the examples given are for directed graphs, the functions will work with undirected graphs as well. Second, if you have used another shortest-paths algorithm already implemented in BGL (such as the Johnson All-Pairs Shortest-Paths algorithm or the Bellman-Ford Shortest-Paths algorithm), then it is clear that the setup and usage of the Floyd-Warshall functions is very similar. This is by design, so users familiar with the other algorithms can easily make the transition to this one. By avoiding the pitfalls previously discussed, such as adding values to infinity and using graphs with negative cycles, users will find this algorithm to be a powerful tool in finding the shortest paths in a graph.

2 Reference Manual

2.1 Floyd_Warshall_all_pairs_shortest_paths

// Named parameter versions

```
template <class VertexListGraph, class DistanceMatrix,
    class P, class T, class R>
bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
    const VertexListGraph& g, DistanceMatrix& d,
    const bgl_named_params<P, T, R>& params)

template <class VertexAndEdgeListGraph, class DistanceMatrix,
    class P, class T, class R>
bool Floyd_Warshall_all_pairs_shortest_paths(
    const VertexAndEdgeListGraph& g, DistanceMatrix& d,
    const bgl_named_params<P, T, R>& params)
```

// Positional parameter versions

```
template <typename VertexListGraph, typename DistanceMatrix,
    typename BinaryPredicate, typename BinaryFunction,
    typename Infinity, typename Zero>
bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
    const VertexListGraph& g, DistanceMatrix& d,
    const BinaryPredicate& compare, const BinaryFunction& combine,
    const Infinity& inf, const Zero& zero)

template <typename VertexAndEdgeListGraph, typename DistanceMatrix,
    typename WeightMap, typename BinaryPredicate,
    typename BinaryFunction, typename Infinity, typename Zero>
bool Floyd_Warshall_all_pairs_shortest_paths(
    const VertexAndEdgeListGraph& g, DistanceMatrix& d,
    const WeightMap& w, const BinaryPredicate& compare,
    const BinaryFunction& combine,
    const Infinity& inf, const Zero& zero)
```

These algorithms find the shortest distance between every pair of vertices in the graph. The algorithms return false if there is a negative weight cycle in the graph, true otherwise. The shortest distance between each pair of vertices is stored in the distance matrix `d`. The difference between the two algorithms is in whether the distance matrix is assumed to be initialized or not, as discussed below under the OUT parameter description.

2.2 Where defined

Floyd_Warshall_all_pairs_shortest.hpp

2.3 Parameters

IN: `Graph& g`

A directed or undirected graph. The graph must be a model of VertexListGraph for calls to Floyd_Warshall_initialized_all_pairs_shortest_paths, and VertexEdgeAndListGraph for calls to Floyd_Warshall_all_pairs_shortest_paths .

OUT: `DistanceMatrix& d`

The length of the shortest path between each pair of vertices u,v are stored in the matrix at location D[u][v]. The DistanceMatrix must be of type {M, I, V} where I is of type vertex_descriptor and V is the type of the weight_map. The set of types must be a model of BasicMatrix, with the exceptions that it isn't required to run in constant time, and it must be mutable. The matrix must be properly initialized when it is passed to the function Floyd_Warshall_initialized_all_pairs_shortest_paths. If the function Floyd_Warshall_all_pairs_shortest_paths is used then the matrix will be initialized for the user.

2.4 Named Parameters

IN: `weight_map(WeightMap w)`

The weight of length of each edge in the graph. The `WeightMap` must be a model of ReadablePropertyMap. The edge descriptor type of the graph needs to be usable as the key type for the weight map. The value_type of the weight map must be the type of the DistanceMatrix, and must always either be part of the graph passed to the function, or passed in as a parameter.
Default: `get(edge_weight, g)`

IN: `distance_compare(CompareFunction cmp)`

The function used to compare distances to determine which target vertex is closer to the source vertex. The CompareFunction must be a model of BinaryPredicate. The argument types must match the value type of the `WeightMap`.
Default: `std::less<WM>`
with `WM = typename property_traits<WeightMap>::value_type`

IN: `distance_combine(CombineFunction cmb)`

The function used to combine distance to compute the distance of a path. The CombineFunction must be a model of BinaryFunction. The argument types must match the value type of the `WeightMap`. The result type must be the same as the distance value type.

Default: `std::plus<WM>`
with `WM = typename property_traits<WeightMap>::value_type`

IN: `distance_inf(WM inf)`

The value used to initialize the distance for each vertex before starting the algorithm, and to represent the distance between vertices for which there is not path. Should be larger than any possible valid path length. The argument type must match the value type of the `WeightMap`.
Default: `std::numeric_limits<WM>::max()`
with `WM = typename property_traits<WeightMap>::value_type`

IN: `distance_zero(WM zero)`

The value used to represent the distance from a vertex to itself, and to determine if a value is negative. The argument type must match the value type of the `WeightMap`.
Default: `0`

2.5 Complexity

The time complexity is $O(V^3)$.

3 Design Issues

3.1 Basic Design Issues

In the implementation of any algorithm, design issues are sure to surface. This implementation is no different. The current design is similar to the other shortest-path algorithms already implemented (such as Johnson and Bellman-Ford), with an emphasis on keeping the implementation generic and robust.

One of the first major decisions was to determine what type of graph to require the user to pass into the function. At first we decided to require that the graph only meet the requirements of a VertexListGraph, since this type meets the minimum requirements for executing the algorithm. However, we decided the subtleties of initializing the matrix for the problem are too complicated to require the user to do it. In order to initialize the matrix for the user, the minimum graph requirements were increased to include VertexListGraph, EdgeListGraph, and IncidenceGraph. In keeping with the goal of making the implementation generic, a second function called `Floyd_Warshall_initialized_all_pairs_shortest_paths` was added that only requires the graph to belong to VertexListGraph, and assumes that the user passes in a matrix that is correctly initialized.

After the graph, we designed the DistanceMatrix. The distance matrix stores the length of the shortest path between each pair of vertices. In a DistanceMatrix D, the lengths are stored in $D[u][v]$, where u, v are a pair of vertices in the graph, and the DistanceMatrix is a model of BasicMatrix, with the exceptions that it isn't required to access values in constant time, and it must be mutable. When the algorithm is complete, it stores its results in this parameter, which is passed by reference for that purpose. This is done to be consistent with the other shortest-paths algorithms. The implementation is split into two major functions, one that accepts the matrix and assumes that it is correctly initialized, and one that initializes the matrix for the user based on values from an edge weight map.

As with most other Boost algorithms, we use a named parameter list to allow the user maximum flexibility in passing in different parameters in a non-specified order. None to all of the named parameters can be specified, and they are done so in the same manner as the other shortest-path algorithms already implemented. Thus, someone looking to utilize this algorithm with little to no familiarity could do so easily if they have experience with another BGL shortest-path algorithms.

Once the parameters were finalized, the implementation and the interface were developed. This was done separately so changes in the interface did not necessarily reflect a change in the implementation. The implementation also checks for parallel edges, and uses the minimum-weighted edge in that case. Finally, it accounts for an edge from a vertex to itself by setting its weight to 0 for all vertices. These checks add to the integrity of the algorithm, and do not add to the overall $O(V^3)$ time bound. The interface allows for an initialized or non-initialized matrix to be passed in, as well as the option for a WeightMap to be passed in. The interface includes a boolean return value, true for success,

false if negative cycles are found, with the DistanceMatrix storing the results of a successful run. This is done, again, for consistency with the already defined shortest-paths algorithms.

Overall, we believe the design is an efficient and effective one in keeping the algorithm within the $O(V^3)$ time bound, and keeping it both as generic as possible and relatively consistent with the other shortest-path algorithms already defined in BGL.

3.2 A Flaw in the Boost implementation of the Bellman-Ford algorithm

Infinity is the value assigned to the shortest path between two nodes for which there is no connecting path, in algorithms that solve the shortest-paths problem. For numeric types whose machine representations have a fixed upper bound and no special value representing infinity, a common convention is use the maximum possible value of the type as a stand-in for infinity. This value is returned by the C++ standard library function std::numeric_limits<T>::max, where T is a numeric type, and this is the common default for the functions that implement shortest-paths algorithms in the BGL.

When any value is added to infinity, the result is infinity. For the BGL algorithms, when any value is added to the stand-in value for infinity, a common occurrence in all the algorithms, the result should be the stand-in value. The functor closed_plus, provided in the Boost header file boost/graph/relax.hpp, is the default addition functor used by the shortest-path algorithms.

"closed_plus" 14 ≡

```
// The following version of the plus functor prevents
// problems due to overflow at positive infinity.

template <class T>
struct closed_plus {
  // std::abs just isn't portable :(
  template <class X>
  inline X my_abs(const X& x) const { return x < 0 ? -x : x; }

  T operator()(const T& a, const T& b) const {
    using namespace std;
    T inf = numeric_limits<T>::max();
    if (b > 0 && my_abs(inf - a) < b)
      return inf;
    return a + b;
  }
};
```

This functor returns the right results when `a` is positive and `b` is positive or negative, but not when `a` is negative. As the comments say, the functor's purpose is not to correctly add infinite values, but to prevent overflow at positive

infinity. The following program shows a case where an error can occur in the bellman_ford_shortest_paths function.

"relax_error.cpp" 15 ≡

```
#include <iostream>
#include <string>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/bellman_ford_shortest_paths.hpp>
#include <boost/graph/relax.hpp>
#include <boost/graph/graph_utility.hpp>

int main() {
  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, std::string,
        boost::property<boost::vertex_index_t, int> > > ,
        boost::property<boost::edge_weight_t, int>,
        boost::disallow_parallel_edge_tag> Graph;
  Graph g;
  boost::property_map<Graph, boost::vertex_name_t>::type index =
        boost::get(boost::vertex_name, g);
  boost::property_map<Graph, boost::edge_weight_t>::type weight =
        boost::get(boost::edge_weight, g);
  boost::graph_traits<Graph>::vertex_descriptor v0, v1, v2;
  boost::graph_traits<Graph>::edge_descriptor e;
  v0 = boost::add_vertex(g);
  boost::put(index, v0, "0");
  v1 = boost::add_vertex(g);
  boost::put(index, v1, "1");
  e = boost::add_edge(v0, v1, g).first;
  boost::put(weight, e, -33);
  v2 = boost::add_vertex(g);
  boost::put(index, v2, "2");
  e = boost::add_edge(v1, v2, g).first;
  boost::put(weight, e, -34);
  boost::print_graph(g, index);
  boost::graph_traits<Graph>::edge_iterator firste, laste;
  for(boost::tie(firste, laste) = boost::edges(g); firste != laste; firste++) {
    std::cout << "The edge from " << index[boost::source(*firste, g)]
        << " " << index[boost::target(*firste, g)] << " has weight "
        << weight[*firste] << std::endl;
  }
  boost::property_map<Graph, boost::vertex_distance_t>::type distance =
        boost::get(boost::vertex_distance, g);
  boost::graph_traits<Graph>::vertex_iterator first, last;
  for(boost::tie(first, last) = boost::vertices(g); first != last; first++) {
    distance[*first] = std::numeric_limits<int>::max();
  }
```

```
  distance[v2] = 0;
  boost::bellman_ford_shortest_paths(g, 3);
  std::cout << "Bellman Ford results" << std::endl;
  for(boost::tie(first, last) = boost::vertices(g); first != last; first++) {
    std::cout << "Distance from " << index[v2] << " to " << index[*first]
        << " is " << distance[*first] << std::endl;;
  }
  std::vector<int> matrix_row(3, std::numeric_limits<int>::max());
  std::vector<std::vector<int> > matrix(3);
  for(boost::tie(first, last) = boost::vertices(g); first != last; first++) {
    matrix[*first] = matrix_row;
  }
}
```

The output of this function is:

"relax_error_output.txt" 16 ≡

```
0 --> 1
1 --> 2
2 -->
The edge from 0 1 has weight -33
The edge from 1 2 has weight -34
Bellman Ford results
Distance from 2 to 0 is 2147483647
Distance from 2 to 1 is 2147483614
Distance from 2 to 2 is 0
```

The value 21474883647 is the stand-in value for infinity. The output implies that there is a very long path from vertex 2 to 1, even though none exists. Here the error is easy to see, but for very large graphs with very long paths it could be more difficult. Now while it is reasonable to demand that users of the shortest-paths algorithms be aware of such subtleties and take steps to protect against them, it would be better if the function designers could prevent such user errors in their code. Designers can add simple checks before any values are summed to check if either of the values are stand-in values for infinity, and take appropriate measures if they are. This fix makes the code more robust at the cost of extra computations.

A more elegant and universal fix would be to implement a generic class that creates an infinity object for whatever type is used, and uses that value appropriately in any mathematical computations.

The infinity value is dealt with in this paper by wrapping a function called inf_combine around the combine functor provided by the user. This function returns either infinity, or the results of the combine functor, depending on which result is appropriate. A functor called inf_plus, which is very similar in function to inf_combine, is passed to the other shortest-paths functions to ensure that they return the right results.

4 Source Code

The following lists and describes the source code for the Floyd_Warshall functions:

"Floyd_Warshall_all_pairs_shortest.hpp" 17a ≡

```
⟨Boost comments 17b⟩

#ifndef BOOST_GRAPH_FLOYD_WARSHALL_HPP
#define BOOST_GRAPH_FLOYD_WARSHALL_HPP

#include <boost/property_map.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/named_function_params.hpp>

namespace boost
{
  ⟨Floyd Warshall Initialized function 19a⟩

  ⟨Floyd Warshall function 19b⟩

  namespace detail {

    ⟨Function that deals with summing infinite values 21b⟩

    ⟨Floyd Warshall dispatch function 21a⟩

    ⟨Floyd Warshall Initialized named parameter dispatch function 22b⟩

    ⟨Floyd Warshall named parameter dispatch function 23a⟩

  }   // namespace detail

  ⟨Floyd Warshall Initialized named parameter function 23b⟩

  ⟨Floyd Warshall named parameter function 24⟩

} // namespace boost

#endif
```

Should this function be submitted and accepted into the Boost Library, then the following legal comments will appear at the top of the file, as well as a description of the functions this file implements.

⟨Boost comments 17b⟩ ≡

```
//=========================================================================
// Copyright 2002 Rensselaer Polytechnic Institute
// Authors: Lauren Foutz, Scott Hill
//
// This file is part of the Boost Graph Library
//
// You should have received a copy of the License Agreement for the
// Boost Graph Library along with the software; see the file LICENSE.
// If not, contact Office of Research, University of Notre Dame, Notre
// Dame, IN 46556.
//
// Permission to modify the code and to distribute modified code is
// granted, provided the text of this NOTICE is retained, a notice that
// the code was modified is included with the above COPYRIGHT NOTICE and
// with the COPYRIGHT NOTICE in the LICENSE file, and that the LICENSE
// file is distributed with the modified code.
//
// LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.
// By way of example, but not limitation, Licensor MAKES NO
// REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY
// PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE COMPONENTS
// OR DOCUMENTATION WILL NOT INFRINGE ANY PATENTS, COPYRIGHTS, TRADEMARKS
// OR OTHER RIGHTS.
//=========================================================================

/*
  This file implements the functions

  template <class VertexListGraph, class DistanceMatrix,
    class P, class T, class R>
  bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
    const VertexListGraph& g, DistanceMatrix& d,
    const bgl_named_params<P, T, R>& params)

  AND

  template <class VertexAndEdgeListGraph, class DistanceMatrix,
    class P, class T, class R>
  bool Floyd_Warshall_all_pairs_shortest_paths(
    const VertexAndEdgeListGraph& g, DistanceMatrix& d,
    const bgl_named_params<P, T, R>& params)
*/
```

Used in part 17a.

There are two Floyd_Warshall functions: One with an initialized matrix using a VertexListGraph, and one that may or may not be initialized using a VertexAndEdgeListGraph, which will be initialized automatically. The initialized function

assumes the user has properly initialized the matrix, performs a concept check, and calls the main dispatch function.

⟨Floyd Warshall Initialized function 19a⟩ ≡

```
template <typename VertexListGraph, typename DistanceMatrix,
  typename BinaryPredicate, typename BinaryFunction,
  typename Infinity, typename Zero>
bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
  const VertexListGraph& g, DistanceMatrix& d,
  const BinaryPredicate& compare,
  const BinaryFunction& combine, const Infinity& inf,
  const Zero& zero)
{
  function_requires<VertexListGraphConcept<VertexListGraph> >();

  return detail::Floyd_Warshall_dispatch(g, d, compare, combine,
  inf, zero);
}
```

Used in part 17a.

The other Floyd_Warshall function must first properly initialize the matrix using the WeightMap before calling the main dispatch function.

⟨Floyd Warshall function 19b⟩ ≡

```
template <typename VertexAndEdgeListGraph, typename DistanceMatrix,
  typename WeightMap, typename BinaryPredicate,
  typename BinaryFunction, typename Infinity, typename Zero>
bool Floyd_Warshall_all_pairs_shortest_paths(
  const VertexAndEdgeListGraph& g,
  DistanceMatrix& d, const WeightMap& w,
  const BinaryPredicate& compare, const BinaryFunction& combine,
  const Infinity& inf, const Zero& zero)
{
  function_requires<VertexListGraphConcept<VertexAndEdgeListGraph> >();
  function_requires<EdgeListGraphConcept<VertexAndEdgeListGraph> >();
  function_requires<IncidenceGraphConcept<VertexAndEdgeListGraph> >();

  typename graph_traits<VertexAndEdgeListGraph>::vertex_iterator
    firstv, lastv, firstv2, lastv2;
  typename graph_traits<VertexAndEdgeListGraph>::edge_iterator first, last;

  ⟨Initialize the matrix to infinity 20a⟩
  ⟨Initialize edges to & from the same vertex to 0 20b⟩
  ⟨Initialize the matrix using the WeightMap 20c⟩

  return detail::Floyd_Warshall_dispatch(g, d, compare, combine,
```

```
    inf, zero);
}
```

Used in part 17a.

First, we initialize the entire matrix to the `Infinity` value passed into the function, so we then only have to initialize vertices that are connected (since non-connected vertices are given a value of infinity):

⟨Initialize the matrix to infinity 20a⟩ ≡

```
for(tie(firstv, lastv) = vertices(g); firstv != lastv; firstv++)
  for(tie(firstv2, lastv2) = vertices(g); firstv2 != lastv2; firstv2++)
    d[*firstv][*firstv2] = inf;
```

Used in part 19b.

Next, we initialize all edges to and from the same vertex in the matrix to 0, since the distance between a vertex and itself is 0:

⟨Initialize edges to & from the same vertex to 0 20b⟩ ≡

```
for(tie(firstv, lastv) = vertices(g); firstv != lastv; firstv++)
  d[*firstv][*firstv] = 0;
```

Used in part 19b.

Finally, we initialize the rest of the matrix using the WeightMap, assigning the appropriate weights to the proper edges. In cases where there are parallel edges (notice the second check for parallel edges for undirected graphs), the edge with the lowest weight is used, since the minimum edge is the only one that matters in this problem.

⟨Initialize the matrix using the WeightMap 20c⟩ ≡

```
for(tie(first, last) = edges(g); first != last; first++)
{
  if (d[source(*first, g)][target(*first, g)] != inf)
    d[source(*first, g)][target(*first, g)] =
      std::min(get(w, *first),
        d[source(*first, g)][target(*first, g)]);
  else
    d[source(*first, g)][target(*first, g)] = get(w, *first);
}

bool is_undirected = is_same<typename
  graph_traits<VertexAndEdgeListGraph>::directed_category,
  undirected_tag>::value;
if (is_undirected)
{
```

```
  for(tie(first, last) = edges(g); first != last; first++)
  {
    if (d[target(*first, g)][source(*first, g)] != inf)
      d[target(*first, g)][source(*first, g)] =
        std::min(get(w, *first),
        d[target(*first, g)][source(*first, g)]);
    else
      d[target(*first, g)][source(*first, g)] = get(w, *first);
  }
}
```

Used in part 19b.

The function Floyd_Warshall_dispatch is the function that implements the Floyd-Warshall algorithm. The shortest paths are calculated inside three nested for loops. Then the matrix is tested for any negative cycles. If any are found then the function returns false, otherwise it returns true. The shortest paths are found and entered into the matrix inside the three nested loops.

⟨Floyd Warshall dispatch function 21a⟩ ≡

```
template<typename VertexListGraph, typename DistanceMatrix,
  typename BinaryPredicate, typename BinaryFunction,
  typename Infinity, typename Zero>
bool Floyd_Warshall_dispatch(const VertexListGraph& g,
  DistanceMatrix& d, const BinaryPredicate &compare,
  const BinaryFunction &combine, const Infinity& inf,
  const Zero& zero)
{
  typename graph_traits<VertexListGraph>::vertex_iterator
    i, lasti, j, lastj, k, lastk;

  ⟨Calculate the shortest paths 22a⟩

  for (tie(i, lasti) = vertices(g); i != lasti; i++)
    if (compare(d[*i][*i], zero))
      return false;
  return true;
}
```

Used in part 17a.

The shortest paths are calculated by taking the minimum of the current minimum path and a new path k edges long. The new path is calculated using the special function inf_combine. This function compares the two values to be summed, and if either is infinity it returns infinity. If both values are below infinity then the function calls the functor combine to sum the values.

⟨Function that deals with summing infinite values 21b⟩ ≡

```
template<typename MatrixValue, typename BinaryFunction,
  typename BinaryPredicate>
inline MatrixValue inf_combine(MatrixValue a, MatrixValue b,
  MatrixValue inf, BinaryFunction combine,
  BinaryPredicate compare)
{
  if(compare(a, inf) && compare(b, inf))
    return combine(a, b);
  else
    return inf;
}
```

Used in part 17a.

⟨Calculate the shortest paths 22a⟩ ≡

```
for (tie(k, lastk) = vertices(g); k != lastk; k++)
  for (tie(i, lasti) = vertices(g); i != lasti; i++)
    for (tie(j, lastj) = vertices(g); j != lastj; j++)
    {
      d[*i][*j] = std::min(d[*i][*j],
        inf_combine(d[*i][*k], d[*k][*j], inf, combine,
        compare), compare);
    }
```

Used in part 21a.

Named parameters are utilized in the Floyd_Warshall functions to be compatible with the other Boost functions, as well as to provide maximum flexibility to the user. As such, two functions were created to dispatch the named parameters to the main functions. The functions extract any named parameters passed in, use defaults based on the value type for the rest, and call the main non-named parameter functions. The first such function uses the value type of the WeightMap and calls the main Floyd_Warshall Initialized function:

⟨Floyd Warshall Initialized named parameter dispatch function 22b⟩ ≡

```
template <class VertexListGraph, class DistanceMatrix,
  class WeightMap, class P, class T, class R>
bool Floyd_Warshall_init_dispatch(const VertexListGraph& g,
  DistanceMatrix& d, WeightMap w,
  const bgl_named_params<P, T, R>& params)
{
  typedef typename property_traits<WeightMap>::value_type WM;

  return Floyd_Warshall_initialized_all_pairs_shortest_paths(g, d,
    choose_param(get_param(params, distance_compare_t()),
      std::less<WM>()),
```

```
    choose_param(get_param(params, distance_combine_t()),
      std::plus<WM>()),
    choose_param(get_param(params, distance_inf_t()),
      std::numeric_limits<WM>::max()),
    choose_param(get_param(params, distance_zero_t()),
      WM()));
}
```

Used in part 17a.

The second function, almost identical to the first, uses the value of the WeightMap and calls the other main Floyd_Warshall function:

⟨Floyd Warshall named parameter dispatch function 23a⟩ ≡

```
template <class VertexAndEdgeListGraph, class DistanceMatrix,
  class WeightMap, class P, class T, class R>
bool Floyd_Warshall_noninit_dispatch(const VertexAndEdgeListGraph& g,
  DistanceMatrix& d, WeightMap w,
  const bgl_named_params<P, T, R>& params)
{
  typedef typename property_traits<WeightMap>::value_type WM;

  return Floyd_Warshall_all_pairs_shortest_paths(g, d, w,
    choose_param(get_param(params, distance_compare_t()),
      std::less<WM>()),
    choose_param(get_param(params, distance_combine_t()),
      std::plus<WM>()),
    choose_param(get_param(params, distance_inf_t()),
      std::numeric_limits<WM>::max()),
    choose_param(get_param(params, distance_zero_t()),
      WM()));
}
```

Used in part 17a.

The following are the named parameter functions the user can call. All functions call the appropriate named parameter dispatch function. If no named parameters are specified, a default named parameter list is created and used.

The following functions call the Initialized named parameter dispatch function:

⟨Floyd Warshall Initialized named parameter function 23b⟩ ≡

```
template <class VertexListGraph, class DistanceMatrix, class P,
  class T, class R>
bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
  const VertexListGraph& g, DistanceMatrix& d,
  const bgl_named_params<P, T, R>& params)
```

```
{
  return detail::Floyd_Warshall_init_dispatch(g, d,
    choose_const_pmap(get_param(params, edge_weight), g, edge_weight),
    params);
}

template <class VertexListGraph, class DistanceMatrix>
bool Floyd_Warshall_initialized_all_pairs_shortest_paths(
  const VertexListGraph& g, DistanceMatrix& d)
{
  bgl_named_params<int,int> params(0);
  return detail::Floyd_Warshall_init_dispatch(g, d,
    get(edge_weight, g), params);
}
```

Used in part 17a.

The following functions call the other named parameter dispatch function:

⟨Floyd Warshall named parameter function 24⟩ ≡

```
template <class VertexAndEdgeListGraph, class DistanceMatrix,
  class P, class T, class R>
bool Floyd_Warshall_all_pairs_shortest_paths(
  const VertexAndEdgeListGraph& g, DistanceMatrix& d,
  const bgl_named_params<P, T, R>& params)
{
  return detail::Floyd_Warshall_noninit_dispatch(g, d,
    choose_const_pmap(get_param(params, edge_weight), g, edge_weight),
    params);
}

template <class VertexAndEdgeListGraph, class DistanceMatrix>
bool Floyd_Warshall_all_pairs_shortest_paths(
  const VertexAndEdgeListGraph& g, DistanceMatrix& d)
{
  bgl_named_params<int,int> params(0);
  return detail::Floyd_Warshall_noninit_dispatch(g, d,
    get(edge_weight, g), params);
}
```

Used in part 17a.

5 Test Plan & Results

5.1 Overview

The test plan is to run the BGL algorithm bellman_ford_shortest_paths for every vertex in the graph and test the results against the results of the Floyd-Warshall

function. This algorithm solves the same problem as the dijkstra_shortest_paths algorithm, but has the added bonus of checking for negative cycles and being defined for negative weights. The consequences of this solution are that it runs in $O(V^2E)$ time for graphs without negative weights, and runs in $O(V^4)$ time for graphs with negative weights, which could be very long since graphs tested by Floyd-Warshall are most likely to be very dense. It also requires that the graph tested belong to the EdgeListGraph concept, which is acceptable since this implementation of the Floyd-Warshall algorithm requires that the tested graph be a VertexAndEdgeListGraph.

The tests consists of two parts. In the first part different types of graphs, such as adjacency lists, directed, undirected, etc, are submitted to the test function. This ensures that the algorithm works properly for all graph types in the specified graph concepts. Also, graphs with negative weights and negative cycles are tested to ensure the function handles them properly. The second part tests the time it takes for the program to run, to ensure it is properly scaled to the expected $O(V^3)$ run time, and that it competes well against other functions that solve the same problem. The two tests ensure both the correctness and generality of the code, as well as ensuring a valid implementation of the algorithm with respect to the maximum time constraints.

5.2 Correctness Tests

Since the Boost Graph Library provides a number of different ways to represent graphs it is necessary to test that the function returns the correct results for all graph implementations that meet the minimum standards of the function. Two generic acceptance functions were written to test the correctness of the function. The functions acceptance_test.h and acceptance_test2.h test results from the Floyd_Warshall_all_pairs_shortest_paths function against results from the bellman_ford_all_pairs_shortest_paths function. The function acceptance_empty_test.h tests that the function returns correct results when given an empty graph.

5.3 The acceptance_test.h and acceptance_test2.h functions

There are three major types of adjacency lists provided by the Boost Graph Library. Those stored as vectors, those stored as lists, and those stored as sets. There are also adjacency lists stored as hash sets and single linked lists, but those implementations are compiler dependant, so testing for them will be left for future development.

The functions acceptance_test.h and acceptance_test2.h test graphs stored as adjacency lists or adjacency matrices that are directed and undirected, have parallel edges, have the number of edges and vertices passed by the function caller, and have self loops. These functions are identical except in how they create random graphs. If the Floyd Warshall function and the Bellman function both return true and produce matrices with equal values, or both return false, then the acceptance test is passed.

The functions acceptance_test.h and acceptance_test2.h are defined as follows:

"acceptance_test.h" 26a ≡

```
#include <map>
#include <algorithm>
#include <iostream>
#include <boost/random/linear_congruential.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include <boost/graph/bellman_ford_shortest_paths.hpp>
#include "Floyd_Warshall_all_pairs_shortest.hpp"

⟨Functor for adding numbers to infinity 27⟩

template<typename Graph>
bool acceptance_test (Graph g, int vec, int e){
  boost::minstd_rand ran(vec);

  {
        ⟨Create an index map for the vertices of the graph 28a⟩
        ⟨Add random edges to the graph 28b⟩
        ⟨Create and initialize the edge weight map 28c⟩
        ⟨Create and initialize the matrices and distance map 29⟩
        ⟨Call the Floyd-Warshall function 30a⟩
        ⟨Call the Bellman-Ford algorithm 30b⟩
        ⟨Compare the results 31⟩
  }
  return true;
}
```

"acceptance_test2.h" 26b ≡

```
#include <map>
#include <algorithm>
#include <iostream>
#include <boost/random/linear_congruential.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include <boost/graph/bellman_ford_shortest_paths.hpp>
#include "Floyd_Warshall_all_pairs_shortest.hpp"

⟨Functor for adding numbers to infinity 27⟩

template<typename Graph>
bool acceptance_test2 (Graph g, int vec, int e){
  boost::minstd_rand ran(vec);
```

```
  {
        ⟨Create an index map for the vertices of the graph 28a⟩
        boost::generate_random_graph(g, vec, e, ran, true);
        ⟨Create and initialize the edge weight map 28c⟩
        ⟨Create and initialize the matrices and distance map 29⟩
        ⟨Call the Floyd-Warshall function 30a⟩
        ⟨Call the Bellman-Ford algorithm 30b⟩
        ⟨Compare the results 31⟩
  }
  return true;
}
```

When any value is added to infinity, the result is infinity. This subtlety is not addressed by the Johnson or Bellman-Ford default implementations in the BGL as the program relax_error.cpp shows. This functor handles addition with infinity for the purpose of this function. An implementation of a more generic inf_plus functor or of an infinity object would be a good addition to the Boost libraries.

⟨Functor for adding numbers to infinity 27⟩ ≡

```
template <typename T>
struct inf_plus{
  T operator()(const T& a, const T& b) const {
    T inf = std::numeric_limits<T>::max();
    if (a == inf || b == inf){
      return inf;
    }
    return a + b;
  }
};
```

Used in parts 26a, 26b, 40b, 48.

An index map of vertices is created so the function can identify which vertices do not match when the functions do not return the same results. The property tag vertex_name is used instead of vertex_index because compiler errors result when a vertex_index map is redefined when the base graph is stored as a vector.

⟨Create an index map for the vertices of the graph 28a⟩ ≡

```
typename boost::property_map<Graph, boost::vertex_name_t>::type index =
      boost::get(boost::vertex_name, g);
typename boost::graph_traits<Graph>::vertex_iterator firstv, lastv,
      firstv2, lastv2;
int x = 0;
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  boost::put(index, *firstv, x);
  x++;
}
```

Used in parts 26a, 26b.

In this section edges are added to the graph using a random number generator to specify the source and target vertices.

⟨Add random edges to the graph 28b⟩ ≡

```
for(int i = 0; i < e; i++){
  boost::add_edge(index[ran() % vec], index[ran() % vec], g);
}
```

Used in part 26a.

The edge weight map is created and initialized in this section using a random number generator to assign both positive and negative values to the edges. The values remain in the range $(-100, 100)$.

⟨Create and initialize the edge weight map 28c⟩ ≡

```
typename boost::graph_traits<Graph>::edge_iterator first, last;
typename boost::property_map<Graph, boost::edge_weight_t>::type
      local_edge_map = boost::get(boost::edge_weight, g);
for(boost::tie(first, last) = boost::edges(g); first != last; first++){
  if (ran() % vec != 0){
    boost::put(local_edge_map, *first, ran() % 100);
  } else {
    boost::put(local_edge_map, *first, 0 - (ran() % 100));
  }
}
```

Used in parts 26a, 26b.

This section creates the matrices for the Floyd-Warshall function and the distance map for the Bellman-Ford function. The distance between a vertex and itself is always zero, that is why negative loops are undefined for both algorithms. Two vertices that are not connected are given a value of infinity, and if

there are parallel edges the one with the minimum weight is selected to be put into the matrix, since the minimum edge is the only one that matters.

⟨Create and initialize the matrices and distance map 29⟩ ≡

```
typedef typename boost::graph_traits<Graph>::vertex_descriptor vertex_des;
std::map<vertex_des,int> matrixRow;
std::map<vertex_des, std::map<vertex_des ,int> > matrix;
typedef typename boost::property_map<Graph, boost::vertex_distance_t>::type
      distance_type;
distance_type distance_row = boost::get(boost::vertex_distance, g);
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  boost::put(distance_row, *firstv, std::numeric_limits<int>::max());
  matrixRow[*firstv] = std::numeric_limits<int>::max();
}
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  matrix[*firstv] = matrixRow;
}
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  matrix[*firstv][*firstv] = 0;
}
std::map<vertex_des, std::map<vertex_des, int> > matrix3(matrix);
std::map<vertex_des, std::map<vertex_des, int> > matrix4(matrix);
for(boost::tie(first, last) = boost::edges(g); first != last; first++){
  if (matrix[boost::source(*first, g)][boost::target(*first, g)]
      != std::numeric_limits<int>::max()){
    matrix[boost::source(*first, g)][boost::target(*first, g)] =
      std::min(boost::get(local_edge_map, *first),
      matrix[boost::source(*first, g)][boost::target(*first, g)]);
   } else {
    matrix[boost::source(*first, g)][boost::target(*first, g)] =
      boost::get(local_edge_map, *first);
   }
}
bool is_undirected =
      boost::is_same<typename boost::graph_traits<Graph>::directed_category,
      boost::undirected_tag>::value;
if (is_undirected){
  for(boost::tie(first, last) = boost::edges(g); first != last; first++){
    if (matrix[boost::target(*first, g)][boost::source(*first, g)]
              != std::numeric_limits<int>::max()){
      matrix[boost::target(*first, g)][boost::source(*first, g)] =
              std::min(boost::get(local_edge_map, *first),
              matrix[boost::target(*first, g)][boost::source(*first, g)]);
    } else {
      matrix[boost::target(*first, g)][boost::source(*first, g)] =
              boost::get(local_edge_map, *first);
```

```
    }
  }
}
```

Used in parts 26a, 26b.

The Floyd_Warshall_all_pairs_shortest_paths function is called several times to test that it is producing consistent results if the same information is passed in different ways to it. The results are stored in the matrix# variable.

⟨Call the Floyd-Warshall function 30a⟩ ≡

```
bool bellman, floyd1, floyd2, floyd3;
std::less<int> compare;
inf_plus<int> combine;
floyd1 = boost::Floyd_Warshall_initialized_all_pairs_shortest_paths(g,
      matrix, weight_map(boost::get(boost::edge_weight, g)).
      distance_compare(compare). distance_combine(combine).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));

floyd2 = boost::Floyd_Warshall_all_pairs_shortest_paths(g, matrix3,
      weight_map(local_edge_map).  distance_compare(compare).
      distance_combine(combine).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));

floyd3 = boost::Floyd_Warshall_all_pairs_shortest_paths(g, matrix4);
```

Used in parts 26a, 26b.

The bellman_ford_shortest_paths function is called for each vertex in the graph, and the results are stored in the variable matrix2 to be compared to the results from the Floyd-Warshall implementation. If a negative cycle is found then the testing is stopped since any further results will be invalid.

⟨Call the Bellman-Ford algorithm 30b⟩ ≡

```
boost::dummy_property_map dummy_map;
std::map<vertex_des, std::map<vertex_des, int> > matrix2;
for(boost::tie(firstv, lastv) = vertices(g); firstv != lastv; firstv++){
  boost::put(distance_row, *firstv, 0);
  bellman = boost::bellman_ford_shortest_paths(g, vec,
      weight_map(boost::get(boost::edge_weight, g)).
      distance_map(boost::get(boost::vertex_distance, g)).
      predecessor_map(dummy_map).distance_compare(compare).
      distance_combine(combine));
  distance_row = boost::get(boost::vertex_distance, g);
  for(boost::tie(firstv2, lastv2) = vertices(g); firstv2 != lastv2;
              firstv2++){
      matrix2[*firstv][*firstv2] = boost::get(distance_row, *firstv2);
```

```
        boost::put(distance_row, *firstv2, std::numeric_limits<int>::max());
    }
    if(bellman == false){
      break;
    }
  }
```

Used in parts 26a, 26b.

If all tests return false, or if all return true and all matrices match in value then the acceptance test returns true. If some tests find negative cycles that others did not, or the matrices do not agree in value, then the acceptance test returns false.

⟨Compare the results 31⟩ ≡

```
if (bellman != floyd1 || bellman != floyd2 || bellman != floyd3){
  std::cout <<
      "A negative cycle was detected in one algorithm but not the others. "
      << std::endl;
  return false;
}
else if (bellman == false && floyd1 == false && floyd2 == false &&
      floyd3 == false){
  return true;
}
else {
  typename boost::graph_traits<Graph>::vertex_iterator first1, first2,
      last1, last2;
  for (boost::tie(first1, last1) = boost::vertices(g); first1 != last1;
      first1++){
    for (boost::tie(first2, last2) = boost::vertices(g); first2 != last2;
            first2++){
      if (matrix2[*first1][*first2] != matrix[*first1][*first2]){
        std::cout << "Algorithms do not match at matrix point "
              << index[*first1] << " " << index[*first2]
              << " Bellman results: " << matrix2[*first1][*first2]
              << " floyd 1 results " << matrix[*first1][*first2]
              << std::endl;
        return false;
      }
      if (matrix2[*first1][*first2] != matrix3[*first1][*first2]){
        std::cout << "Algorithms do not match at matrix point "
              << index[*first1] << " " << index[*first2]
              << " Bellman results: " << matrix2[*first1][*first2]
              << " floyd 2 results " << matrix3[*first1][*first2]
              << std::endl;
        return false;
      }
      if (matrix2[*first1][*first2] != matrix4[*first1][*first2]){
```

```
            std::cout << "Algorithms do not match at matrix point "
                  << index[*first1] << " " << index[*first2]
                  << " Bellman results: " << matrix2[*first1][*first2]
                  << " floyd 3 results " << matrix4[*first1][*first2]
                  << std::endl;
            return false;
          }
        }
      }
    }
```

Used in parts 26a, 26b.

5.4 Programs that call acceptance_test.h and acceptance_test2.h

Tests a graph stored as a directed adjacency list that is stored as a vector of vectors.

"bellman_vec_vec_test.cpp" 32 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vertices num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::vecS, boost::vecS,
        boost::directedS, boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > VecVecDirected;
  VecVecDirected g(vec);
  if (acceptance_test(g, vec, e)){
    std::cout << "Pass test for vector vector directed adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for vector vector directed adjacency list." << std::endl;
```

```
  return 1;
}
```

Tests a graph stored as an undirected adjacency list stored as a vector of vectors.

"bellman_vec_vec_un_test.cpp" 34 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > VecVecUnDirected;
  VecVecUnDirected g(vec);
  if (acceptance_test(g, vec, e)){
    std::cout << "Pass test for vector vector undirected adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for vector vector undirected adjacency list."
        << std::endl;
  return 1;
}
```

Tests a graph stored as a directed adjacency list stored as two lists.

"bellman_list_list_test.cpp" 35 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test2.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertex.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::listS, boost::listS, boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > ListListDirected;
  ListListDirected g;
  if (acceptance_test2(g, vec, e)){
    std::cout << "Pass test for List List directed adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for List List directed adjacency list." << std::endl;
  return 1;
}
```

Tests a graph stored as an undirected adjacency list stored as two lists.

"bellman_list_list_un_test.cpp" 36 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test2.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::listS, boost::listS,
        boost::undirectedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > ListListUnDirected;
  ListListUnDirected g;
  if (acceptance_test2(g, vec, e)){
    std::cout << "Pass test for List List undirected adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for List List undirected adjacency list." << std::endl;
  return 1;
}
```

Tests a graph stored as a directed adjacency list stored as two sets.

"bellman_set_set_test.cpp" 37 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test2.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::setS, boost::setS,
        boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > SetSetDirected;
  SetSetDirected g;
  if (acceptance_test2(g, vec, e)){
    std::cout << "Pass test for set set directed adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for set set directed adjacency list." << std::endl;
  return 1;
}
```

Tests a graph stored as an undirected adjacency list stored as two sets.

"bellman_set_set_un_test.cpp" 38 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "acceptance_test2.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_list<boost::setS, boost::setS,
        boost::undirectedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > SetSetUnDirected;
  SetSetUnDirected g;
  if (acceptance_test2(g, vec, e)){
    std::cout << "Pass test for set set undirected adjacency list."
        << std::endl;
    return 0;
  }
  std::cout << " for set set undirected adjacency list." << std::endl;
  return 1;
}
```

Tests a graph stored as a directed adjacency matrix.

"bellman_matrix_directed.cpp" 39 ≡

```
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include "acceptance_test.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 1){
    std::cout << "Graph must contain at least 1 vertex.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_matrix<boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > MatrixDirected;
  MatrixDirected g(vec);
  if (acceptance_test(g, vec, e)){
    std::cout << "Pass test for directed adjacency matrix." << std::endl;
    return 0;
  }
  std::cout << " for directed adjacency matrix." << std::endl;
  return 1;
}
```

Tests a graph stored an undirected adjacency matrix.

"bellman_matrix_undirected.cpp" 40a ≡

```
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include "acceptance_test.h"

int main (int argc, char* argv[]){
  long vec, e;
  if (argc == 3){
    vec = atoi(argv[1]);
    e = atoi(argv[2]);
  } else {
    std::cout << "Usage: " << argv[0] << " num_vectors num_edges "
        << std::endl;
    return 1;
  }
  if (vec < 1){
    std::cout << "Graph must contain at least 1 vertex.  Aborting test."
        << std::endl;
    return 0;
  }
  typedef boost::adjacency_matrix<boost::undirectedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > MatrixUnDirected;
  MatrixUnDirected g(vec);
  if (acceptance_test(g, vec, e)){
    std::cout << "Pass test for undirected adjacency matrix." << std::endl;
    return 0;
  }
  std::cout << " for undirected adjacency matrix." << std::endl;
  return 1;
}
```

5.5 Tests for empty graphs

When an empty graph is submitted to the function then it should return the matrix untouched and should return true, since negative cycles cannot exist in a graph without edge weights or edges.

The function acceptance_empty_test.h tests the Floyd-Warshall function when an empty graph is submitted.

"acceptance_empty_test.h" 40b ≡

```
#include <vector>
#include <algorithm>
#include <iostream>
```

```
#include <boost/random/linear_congruential.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include "Floyd_Warshall_all_pairs_shortest.hpp"

⟨Functor for adding numbers to infinity 27⟩

template<typename Graph>
bool acceptance_empty_test (Graph g, int vec, int e){
  boost::minstd_rand ran(1);

  {
    ⟨Set up maps 41⟩
    ⟨Set up matrices 42⟩
    ⟨Call Floyd-Warshall function 43a⟩
    ⟨Test results 43b⟩
  }
  return true;
}
```

This section of the code sets up the maps. Most of this should not be executed, it is added for consistency.

⟨Set up maps 41⟩ ≡

```
typename boost::property_map<Graph, boost::vertex_name_t>::type index =
      boost::get(boost::vertex_name, g);
typename boost::graph_traits<Graph>::vertex_iterator firstv, lastv,
      firstv2, lastv2;
int x = 0;
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  boost::put(index, *firstv, x);
  x++;
}
typename boost::graph_traits<Graph>::edge_iterator first, last;
typename boost::property_map<Graph, boost::edge_weight_t>::type
      local_edge_map = boost::get(boost::edge_weight, g);
for(boost::tie(first, last) = boost::edges(g); first != last; first++){
  if (ran() % vec != 0){
    boost::put(local_edge_map, *first, ran() % 100);
  } else {
    boost::put(local_edge_map, *first, 0 - (ran() % 100));
  }
}
```

Used in part 40b.

This section of the code sets up the matrices. Most of this should not be executed, it is added for consistency.

⟨Set up matrices 42⟩ ≡

```
std::vector<int> matrixRow(vec, std::numeric_limits<int>::max());
std::vector<std::vector<int> > matrix;
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  matrixRow[*firstv] = std::numeric_limits<int>::max();
}
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  matrix[*firstv] = matrixRow;
}
for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
  matrix[*firstv][*firstv] = 0;
}
std::vector<std::vector<int> > matrix3(matrix);
std::vector<std::vector<int> > matrix4(matrix);
for(boost::tie(first, last) = boost::edges(g); first != last; first++){
  if (matrix[boost::source(*first, g)][boost::target(*first, g)]
      != std::numeric_limits<int>::max()){
    matrix[boost::source(*first, g)][boost::target(*first, g)] =
      std::min(boost::get(local_edge_map, *first),
      matrix[boost::source(*first, g)][boost::target(*first, g)]);
  } else {
    matrix[boost::source(*first, g)][boost::target(*first, g)] =
      boost::get(local_edge_map, *first);
  }
}
bool is_undirected =
      boost::is_same<typename boost::graph_traits<Graph>::directed_category,
      boost::undirected_tag>::value;
if (is_undirected){
  for(boost::tie(first, last) = boost::edges(g); first != last; first++){
    if (matrix[boost::target(*first, g)][boost::source(*first, g)]
              != std::numeric_limits<int>::max()){
      matrix[boost::target(*first, g)][boost::source(*first, g)] =
              std::min(boost::get(local_edge_map, *first),
              matrix[boost::target(*first, g)][boost::source(*first, g)]);
    } else {
      matrix[boost::target(*first, g)][boost::source(*first, g)] =
              boost::get(local_edge_map, *first);
    }
  }
}
```

Used in part 40b.

This section calls the Floyd-Warshall function in three different ways to make sure that they all treat the empty graph in a consistent way.

⟨Call Floyd-Warshall function 43a⟩ ≡

```
bool floyd1 = true;
bool floyd2 = true;
bool floyd3 = true;
std::less<int> compare;
inf_plus<int> combine;
floyd1 = boost::Floyd_Warshall_initialized_all_pairs_shortest_paths(g,
      matrix, weight_map(boost::get(boost::edge_weight, g)).
      distance_compare(compare). distance_combine(combine).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));

floyd2 = boost::Floyd_Warshall_all_pairs_shortest_paths(g, matrix3,
      weight_map(boost::get(boost::edge_weight, g)).
      distance_compare(compare). distance_combine(combine).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));

floyd3 = boost::Floyd_Warshall_all_pairs_shortest_paths(g, matrix4);
```

Used in part 40b.

Finally the results from the three function calls are checked for correctness. If all the functions return true and have left the matrices empty then the test returns true, otherwise it returns false.

⟨Test results 43b⟩ ≡

```
if (floyd1 == false){
  std::cout << "Fail acceptance test for floyd1";
  return false;
}else if (floyd2 == false){
  std::cout << "Fail acceptance test for floyd2";
  return false;
}else if (floyd3 == false){
  std::cout << "Fail acceptance test for floyd1";
  return false;
}else {
  if(matrix.empty() && matrix3.empty() && matrix4.empty()){
    std::cout << "Pass acceptance test ";
  } else {
    std::cout << "Matrices were not empty " << std::endl;
    return false;
  }
}
```

Used in part 40b.

The program empty_graph.cpp calls the function acceptance_empty_test.h

"empty_graph.cpp" 44 ≡

```
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include "acceptance_empty_test.h"


int main (){
  int vec = 0;
  int e = 0;
  typedef boost::adjacency_matrix<boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_name_t, int> > ,
        boost::property<boost::edge_weight_t, int> > EmptyGraph;
  EmptyGraph g(0);
  if (acceptance_empty_test(g, vec, e)){
    std::cout << "Pass acceptance test for an empty graph" << std::endl;
    return 0;
  }
  std::cout << " Fail acceptance test for an empty graph." << std::endl;
  return 1;
}
```

5.6 Correctness Tests Results

These are the results from submitting various kinds of graphs with varying number of vertices and edges to the function acceptance_test.h. These test where performed using a gcc version 3.2 compiler with the Boost Library version 1.29.0 on a Sun Microsystems machine running Solaris Operating System 8.

Graph Type	Vertices	Edges	Results
Undirected Matrix	0	0	Pass
Directed Adjacency Matrix	2	0	Pass
Directed Adjacency Matrix	2	4	Pass
Directed Adjacency Matrix	5	10	Pass
Directed Adjacency Matrix	20	20	Pass
Directed Adjacency Matrix	100	1000	Pass
Undirected Adjacency Matrix	2	0	Pass
Undirected Adjacency Matrix	2	4	Pass
Undirected Adjacency Matrix	5	10	Pass
Undirected Adjacency Matrix	20	20	Pass
Undirected Adjacency Matrix	100	1000	Pass
Directed Adjacency List stored as a List	2	0	Pass
Directed Adjacency List stored as a List	2	4	Pass
Directed Adjacency List stored as a List	5	10	Pass
Directed Adjacency List stored as a List	20	20	Pass
Directed Adjacency List stored as a List	100	1000	Pass
Undirected Adjacency List stored as a List	2	0	Pass
Undirected Adjacency List stored as a List	2	4	Pass
Undirected Adjacency List stored as a List	5	10	Pass
Undirected Adjacency List stored as a List	20	20	Pass
Undirected Adjacency List stored as a List	100	1000	Pass
Directed Adjacency List stored as a Set	2	0	Pass
Directed Adjacency List stored as a Set	2	4	Pass
Directed Adjacency List stored as a Set	5	10	Pass
Directed Adjacency List stored as a Set	20	20	Pass
Directed Adjacency List stored as a Set	100	1000	Pass
Undirected Adjacency List stored as a Set	2	0	Pass
Undirected Adjacency List stored as a Set	2	4	Pass
Undirected Adjacency List stored as a Set	5	10	Pass
Undirected Adjacency List stored as a Set	20	20	Pass
Undirected Adjacency List stored as a Set	100	1000	Pass
Directed Adjacency List stored as a Vector	2	0	Pass
Directed Adjacency List stored as a Vector	2	4	Pass
Directed Adjacency List stored as a Vector	5	10	Pass
Directed Adjacency List stored as a Vector	20	20	Pass
Directed Adjacency List stored as a Vector	100	1000	Pass
Undirected Adjacency List stored as a Vector	2	0	Pass
Undirected Adjacency List stored as a Vector	2	4	Pass
Undirected Adjacency List stored as a Vector	5	10	Pass
Undirected Adjacency List stored as a Vector	20	20	Pass
Undirected Adjacency List stored as a Vector	100	1000	Pass

The Floyd-Warshall implementation works correctly for all submitted graphs.

5.7 Time Tests

The run time of the Floyd_Warshall_all_pairs_shortest_paths function is tested against its chief rival function johnson_all_pairs_shortest_paths, which solves the same problem as the Floyd-Warshall function. In theory the Floyd-Warshall algorithm should run faster than the Johnson algorithm on dense graphs, but the results from the tests did not follow that pattern on every case.

The function test_time.h runs the functions Floyd_Warshall_all_pairs_shortest_paths and johnson_all_pairs_shortest_paths ten times each and prints out the average run time. This function produces consistent average run times between executions with the same size graphs.

The header files timer.h and recorder0.h, from [4], are used to time the functions and store the results respectively. The names of the header files of these files have been changed to comply with current compiler standards.

"timer.h" 46 ≡

```
/*

Defines class timer for measuring computing times. Implemented
using the standard clock function from time.h.


*/

/*
 * Copyright (c) 1997 Rensselaer Polytechnic Institute
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Rensselaer Polytechnic Institute makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 */

#ifndef TIMER_H
#define TIMER_H

#include <vector>
#include <ctime>

using namespace std;

class timer {
protected:
  double start, finish;
```

```
  public:
    vector<double> times;
    void record() {
      times.push_back(time());
    }
    void reset_vectors() {
      times.erase(times.begin(), times.end());
    }
    void restart() { start = clock(); }
    void stop() { finish = clock(); }
    double time() const { return ((double)(finish - start))/CLOCKS_PER_SEC; }
  };

  #endif
```

"recorder0.h" 47 ≡

```
  /*

  Defines class recorder<Timer> for recording computing times as
  measured by objects of class Timer.  See also recorder.h, which
  defines another recorder class capable of also recording operation
  counts.

  */

  /*
   * Copyright (c) 1997 Rensselaer Polytechnic Institute
   *
   * Permission to use, copy, modify, distribute and sell this software
   * and its documentation for any purpose is hereby granted without fee,
   * provided that the above copyright notice appear in all copies and
   * that both that copyright notice and this permission notice appear
   * in supporting documentation.  Rensselaer Polytechnic Institute makes no
   * representations about the suitability of this software for any
   * purpose.  It is provided "as is" without express or implied warranty.
   *
   */

  #ifndef RECORDER_H
  #define RECORDER_H

  #include <vector>
  #include <iomanip>

  using namespace std;

  template <class Container>
  typename Container::value_type median(Container& c)
  {
```

```
  typename Container::iterator midpoint = c.begin() + (c.end() - c.begin())/2;
  nth_element(c.begin(), midpoint, c.end());
  return *midpoint;
}

template <class Timer>
class recorder {
  vector<double> times;

public:
    void record(const Timer& t) {
      times.push_back(t.time());
  }
  void report(ostream& o, int repeat_factor)
  {
    o << setiosflags(ios::fixed) << setprecision(3) << setw(12)
      << median(times)/repeat_factor;
      o << "     ";
  }
  void reset() {
    times.erase(times.begin(), times.end());
  }
};

#endif
```

"time_test.h" 48 ≡

```
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <boost/random/linear_congruential.hpp>
#include <boost/graph/graph_utility.hpp>
#include <boost/graph/properties.hpp>
#include <boost/graph/johnson_all_pairs_shortest.hpp>
#include "timer.h"
#include "recorder0.h"
#include "Floyd_Warshall_all_pairs_shortest.hpp"
```

⟨Functor for adding numbers to infinity 27⟩

```
template<typename Graph>
void time_test (Graph gr, int vec, int sparse){
  boost::minstd_rand ran(vec);

  {
    Graph g(vec);
```

⟨Create a complete graph or a sparse graph 49⟩

⟨Create the matrices for the Floyd-Warshall and Johnson functions 50a⟩
⟨Call the Floyd-Warshall function 20 times 50b⟩
⟨Call the Johnson function 20 times 50c⟩
}
}

The graph is created as either a complete graph or a sparse graph depending on the value of the function parameter sparse. The number of edges generated in the sparse case is equal to half the number of vertices. If the graph is dense then the Floyd-Warshall function should be faster; if it is sparse, the Johnson function should be faster.

⟨Create a complete graph or a sparse graph 49⟩ ≡

```
typename boost::graph_traits<Graph>::vertex_iterator firstv, lastv,
      firstv2, lastv2;
typename boost::graph_traits<Graph>::edge_descriptor ed;
bool added;
typename boost::property_map<Graph, boost::edge_weight_t>::type
      local_edge_map = boost::get(boost::edge_weight, g);
if (sparse == 0){
  for(boost::tie(firstv, lastv) = boost::vertices(g); firstv != lastv;
      firstv++){
    for(boost::tie(firstv2, lastv2) = boost::vertices(g); firstv2 != lastv2;
              firstv2++){
      boost::tie(ed, added) = boost::add_edge(*firstv, *firstv2, g);
      if (added){
        boost::put(local_edge_map, ed, ran() % 100);
      }
    }
  }
} else{
  for (int i = 0; i < vec/2;){
    boost::tie(ed, added) = boost::add_edge(ran() % vec, ran() % vec, g);
    if (added){
      boost::put(local_edge_map, ed, ran() % 100);
      i++;
    }
  }
}
```

Used in part 48.

One un-initialized matrix is created to submit to the functions.

⟨Create the matrices for the Floyd-Warshall and Johnson functions 50a⟩ ≡

```
typename boost::graph_traits<Graph>::edge_iterator first, last;
typedef typename boost::graph_traits<Graph>::vertex_descriptor vertex_des;
std::vector<int> matrixRow(vec, std::numeric_limits<int>::max());
std::vector<std::vector<int> > matrix(vec, matrixRow);
```

Used in part 48.

The Floyd-Warshall function is called 20 times so that an average run time can be produced consistently for graphs with small numbers of vertices. The function is called only once if the graph has over 100 vertices, since small variations in run time will not impact the analysis of run times that long. The version of the Floyd-Warshall function is used that initializes the matrix in the function so it will match more closely to the Johnson function, which always initializes the matrix for the caller.

⟨Call the Floyd-Warshall function 20 times 50b⟩ ≡

```
int repetition = 1;
if (vec < 100){
  repetition = 20;
}
std::less<int> compare;
std::plus<int> combine;
timer timer1;
recorder<timer> stats;
for (int i = 0; i < repetition; i++){
  timer1.restart();
  Floyd_Warshall_all_pairs_shortest_paths(g, matrix,
      weight_map(local_edge_map). distance_compare(compare).
      distance_combine(combine).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));
  timer1.stop();
  stats.record(timer1);
}
std::cout << std::setw(6) << vec;
stats.report(std::cout, 1);
std::cout << "  ";
```

Used in part 48.

The Johnson function is called 20 times for small graphs so that the average run time can be produced consistently.

⟨Call the Johnson function 20 times 50c⟩ ≡

```
inf_plus<int> combine2;
timer timer2;
```

```
recorder<timer> stats2;
for (int i = 0; i < repetition; i++){
  timer2.restart();
  boost::johnson_all_pairs_shortest_paths(g, matrix,
      weight_map(boost::get(boost::edge_weight, g)).
      vertex_index_map(boost::get(boost::vertex_index, g)).
      distance_compare(compare). distance_combine(combine2).
      distance_inf(std::numeric_limits<int>::max()). distance_zero(0));
  timer2.stop();
  stats2.record(timer2);
}
stats2.report(std::cout, 1);
std::cout << std::endl;
```

Used in part 48.

5.8 Programs that use time_test.h

Two programs use time_test.h. Both programs accept as command line arguments a number of vertices, the number of times the vertices should be multiplied by 2 and tested again, and whether the graph should be sparse or dense. The first program, list_time_test.cpp, tests the run time for a graph stored as an adjacency list.

"list_time_test.cpp" 51 ≡

```
#include <iostream>
#include <boost/graph/adjacency_list.hpp>
#include "time_test.h"

int main (int argc, char* argv[]){
  long vec, repetition, sparse;
  if (argc == 4){
    vec = atoi(argv[1]);
    repetition = atoi(argv[2]);
    sparse = atoi(argv[3]);
  } else {
    std::cout << "Usage: " << argv[0] <<
        " <num_vertices> <repetition> <sparse> where repetition is "
        << "the number of times <num_vertices> is multiplied by 2 and "
        << "sparse = 0 for a complete graph, and 1 for a graph with "
        <<" |V|/2 edges " << std::endl;
    return 1;
  }
  if (vec < 2){
    std::cout << "Graph must contain at least 2 vertices.  Aborting test."
        << std::endl;
    return 0;
  }
```

```
  typedef boost::adjacency_list<boost::vecS, boost::vecS,
        boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_index_t, int> > ,
        boost::property<boost::edge_weight_t, int> > ListDirected;
  ListDirected g(1);
  std::cout << "Average run times of the "
        << "floyd_warshall_all_pairs_shortest_paths and "
        << "johnson_all_pairs_shortest_paths function: " << std::endl;
  std::cout <<std::setw(8) << "vertices" <<  std::setw(11) << "floyd"
        << std::setw(19) << "johnson" << std::endl;
  for (int i = 0; i < repetition; i++){
    if (vec > 10000){
      std::cout << "The number of vectors has gone over 10000, "
        << "aborting test because it will take too long." << std::endl;
      return 0;
    }
    time_test(g, vec, sparse);
    vec = vec * 2;
  }
  return 0;
}
```

The second program is matrix_time_test.cpp, which tests the run time for a graph stored as an adjacency matrix.

"matrix_time_test.cpp" 52 ≡

```
#include <iostream>
#include <boost/graph/adjacency_matrix.hpp>
#include "time_test.h"

int main (int argc, char* argv[]){
  long vec, repetition, sparse;
  if (argc == 4){
    vec = atoi(argv[1]);
    repetition = atoi(argv[2]);
    sparse = atoi(argv[3]);
  } else {
    std::cout << "Usage: " << argv[0] << " <num_vertices> <repetition> "
        << "<sparse> where repetition is the number of times <num_vertices> "
        << "is multiplied by 2, and sparse = 0 means a complete graph, "
        << "and sparse = 1 means a sparse graph " << std::endl;
    return 1;
  }
  if (vec < 1){
    std::cout << "Graph must contain at least 1 vertex.  Aborting test."
        << std::endl;
    return 0;
```

```
  }
  typedef boost::adjacency_matrix<boost::directedS,
        boost::property<boost::vertex_distance_t, int,
        boost::property<boost::vertex_index_t, int> > ,
        boost::property<boost::edge_weight_t, int> > MatrixDirected;
  MatrixDirected g(1);
  std::cout << "Average run times of the "
        << "floyd_warshall_all_pairs_shortest_paths and "
        << "johnson_all_pairs_shortest_paths function: " << std::endl;
  std::cout <<std::setw(8) << "vertices" <<  std::setw(11) << "floyd"
        << std::setw(19) << "johnson" << std::endl;
  for (int i = 0; i < repetition; i++){
    if (vec > 10000){
      std::cout << "The number of vectors has gone over 10000, "
        << "aborting test because it will take too long." << std::endl;
      return 0;
    }
    time_test(g, vec, sparse);
    vec = vec * 2;
  }
  return 0;
}
```

5.9 Time test results

The tests were performed using a gcc version 3.2 compiler with the Boost Library version 1.29.0 on a Sun Microsystems machine running Solaris Operating System 8 with the optimization option turned on. The results are as follows:

Graph Type	Vertices	Sparse or Dense	Floyd Results	Johnson Results
Adjacency List	2	Dense	0.00	0.00
Adjacency List	4	Dense	0.00	0.00
Adjacency List	8	Dense	0.00	0.01
Adjacency List	16	Dense	0.00	0.00
Adjacency List	32	Dense	0.01	0.02
Adjacency List	64	Dense	0.04	0.13
Adjacency List	128	Dense	0.28	0.92
Adjacency List	256	Dense	2.20	6.84
Adjacency List	512	Dense	17.43	57.96
Adjacency List	1024	Dense	147.90	464.69
Adjacency Matrix	2	Dense	0.00	0.00
Adjacency Matrix	4	Dense	0.00	0.00
Adjacency Matrix	8	Dense	0.00	0.00
Adjacency Matrix	16	Dense	0.00	0.00
Adjacency Matrix	32	Dense	0.01	0.02
Adjacency Matrix	64	Dense	0.04	0.14
Adjacency Matrix	128	Dense	0.28	1.02
Adjacency Matrix	256	Dense	2.20	7.18
Adjacency Matrix	512	Dense	17.79	58.97
Adjacency Matrix	1024	Dense	148.74	455.57
Adjacency List	2	Sparse	0.00	0.00
Adjacency List	4	Sparse	0.00	0.00
Adjacency List	8	Sparse	0.00	0.00
Adjacency List	16	Sparse	0.00	0.00
Adjacency List	32	Sparse	0.00	0.00
Adjacency List	64	Sparse	0.03	0.00
Adjacency List	128	Sparse	0.27	0.01
Adjacency List	256	Sparse	2.14	0.02
Adjacency List	512	Sparse	17.16	0.08
Adjacency List	1024	Sparse	143.91	0.31
Adjacency Matrix	2	Sparse	0.00	0.00
Adjacency Matrix	4	Sparse	0.00	0.00
Adjacency Matrix	8	Sparse	0.00	0.00
Adjacency Matrix	16	Sparse	0.00	0.00
Adjacency Matrix	32	Sparse	0.00	0.00
Adjacency Matrix	64	Sparse	0.03	0.00
Adjacency Matrix	128	Sparse	0.30	0.00
Adjacency Matrix	256	Sparse	2.13	0.01
Adjacency Matrix	512	Sparse	17.20	0.05
Adjacency Matrix	1024	Sparse	144.23	0.19

The number of vertices is multiplied by 2 in each test, so the run times of the Floyd-Warshall function should be increasing by a factor of $(2n)^3/n^3 = 8$ each time, and the test results bear this out approximately for all tests past 64

vertices. In the sparse graph case, the number of edges generated is equal to half the number of vertices in the graph.

The number of edges should not have a major effect on the run time of the Floyd-Warshall function, and the Floyd-Warshall function should run faster than the Johnson function on dense graphs. The tests show that both of these requirements are being met.

6 Further Development

The current version of our implementation of the Floyd-Warshall algorithm works correctly and efficiently, but still requires some additional testing before it can be seriously considered for the Boost Library.

Further testing will have to be done to ensure that this implementation functions correctly for all possible inputs and environments that exist in the Boost Library. Currently this implementation has been proven to work correctly for the inputs and environments that it is most likely to experience. Future test should include submitting a graph to the function Floyd_Warshall_initialized_all_pairs_shortest_paths that meets only the minimum requirements of the concept VertexListGraph, to test the function generality. Future generality tests should also include testing that the function works correctly on edge weight types that are not number types.

Future testing should address the issue of consistency checks. Ideally, all test cases would use a method of assertion to ensure that all parameters and values are within an acceptable range. A future implementation of this algorithm should also include these tests, since more error checking is always a good thing, unless it significantly slows performance.

A more elegant solution for summing values with infinity would be an excellent addition to this function and the Boost Library as a whole, but such a task will most likely involve more work than designing this implementation. If such a task it undertaken, it will be a separate project.

7 Summary

In summary, this implementation of the Floyd-Warshall algorithm for the Boost Library is robust, flexible, and efficient. All that needs to be done before it can be submitted for acceptance into the Boost Library is additional testing of its generality.

8 References

References

[1] The Boost Graph Library, 2002, url = http://www.boost.org 1.1

[2] Jeremy Siek, Lie-Quan Lee, Andrew Lumsdaine, *The Boost Graph Library User Guide and Reference Manual*, Addison-Wesley, 2002, ISBN 0-201-72914-8 1.1

[3] Dr. Jeffrey J. Gosper, Floyd-Warshall All-Pairs Shortest Pairs Algorithm, Brunel University, 1998, url = http://www.brunel.ac.uk/~castjjg/java/shortest_path/shortest_path.html 1.2

[4] D. R. Musser, G. J. Derge, A. Saini. *STL Tutorial and Reference Guide, 2nd edition*, Addison-Wesley, 2001, ISBN 0-201-37923-6 5.7