An Algorithm Object
for Depth First Search

Nick Markham       Greg Medoff

December 30, 2003

Contents

1 User’s Guide
  1.1 Background ........................................... 1
  1.2 A Better Depth First Search ...................... 1

2 Reference Manual ........................................ 3
  2.1 DepthFirstSearch .................................... 4
  2.2 DepthFirstSearch<G, C>::iterator .............. 9
  2.3 DepthFirstSearch::vertex_iterator ............ 11
  2.4 DepthFirstSearch::edge_iterator ............. 11

3 Design Issues and Source Code ....................... 12
  3.1 Designing the basic interface ................. 12
     3.1.1 Algorithm objects vs. visitor-based algorithms .... 12
     3.1.2 What DepthFirstSearch needs to do ........ 13
     3.1.3 The interface ................................ 14
  3.2 Implementation of the algorithm ............. 16
     3.2.1 A not so in-depth look at the BGL depth_first_search ... 16
     3.2.2 Which control points to borrow? .......... 17
     3.2.3 Recursive function to a not so recursive function ... 18
     3.2.4 What kind of data members? .......... 19
     3.2.5 Implementation ............................... 20
     3.2.6 Advance ...................................... 24
     3.2.7 Putting it all together .................. 28
  3.3 Extending the interface(Iterators) .......... 29
     3.3.1 What iterator category? ................. 30
     3.3.2 What does a compliant iterator need? .... 30
<table>
<thead>
<tr>
<th>3.3.3 Many iterators for different things?</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.4 The iterator innards</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Test Plan</th>
<th>38</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Correctness Testing</td>
<td>38</td>
</tr>
<tr>
<td>4.1.1 Framework</td>
<td>38</td>
</tr>
<tr>
<td>4.1.2 Test Cases</td>
<td>43</td>
</tr>
<tr>
<td>4.2 Performance Testing</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1 Topological Sort</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2 Topological Sort — Sparse Graphs</td>
<td>50</td>
</tr>
<tr>
<td>4.2.3 Cycle Detection</td>
<td>55</td>
</tr>
</tbody>
</table>

| 5 Conclusion | 58 |

| A Source Code for Timer Class | 60 |
1 User’s Guide

1.1 Background

Traditionally in C++, generic algorithms take functors as parameters to modify their behavior. In the Boost Graph Library [3], the concept of a functor is expanded somewhat to that of a visitor, but the usage remains the same: the programmer must construct the functor or visitor ahead of time, and then pass it to the algorithm which will make use of it.

Dietmar Kühl proposes a different approach: the idea of an algorithm object [1]. Rather than having an algorithm that calls the programmer’s code, Kühl suggests that the algorithm be encapsulated in an object that would iterate over the states of the algorithm, much as traditional iterators iterate over containers.

Algorithm objects could provide many advantages over the functor/visitor approach, such as the ability to perform additional computations at intermediate steps and the option to terminate the algorithm at any time. In addition, an algorithm object that acts as an iterator interacts better with iterator-based algorithms such as those in the STL.

1.2 A Better Depth First Search

The BGL provides a depth first search algorithm written in the traditional way, that takes a visitor object to allow the algorithm to be modified. However, as we described above, algorithm objects are more generic than visitor algorithms. Therefore, we have reimplemented the depth first search algorithm as an algorithm object.

Our depth first search object can in principle do everything the BGL depth first search can, as well as some extra features such as early termination. For example, to print a depth first ordering with our implementation requires only one call to the copy algorithm.

\[
\text{(Print depth first ordering of vertices 1)} \equiv
\]

\[
\text{DepthFirstSearch}\langle G, C \rangle \text{ dfs}(g, c);
\]

\[
\text{std::copy(dfs.vertex_it(), dfs.vertex_end(),}
\]

\[
\text{std::ostream_iterator<vertex_t>(std::cout, " "));}
\]

\[
\text{std::cout} \text{ << std::endl;}
\]

Used in part 2b.
Printing the edges in depth first order is accomplished similarly, with an `edge_iterator`:

(Print depth first ordering of edges 2a) ≡

```cpp
DepthFirstSearch<G, C> dfs(g, c, DFSCP_EXAMINE_EDGE);
std::copy(dfs.edge_it(DFSCP_EXAMINE_EDGE), dfs.edge_end(),
          std::ostream_iterator<edge_t>(std::cout, " ");
std::cout << std::endl;
```

Used in part 2b.

tutorial.cpp illustrates the use of these fragments:

"tutorial.cpp" 2b ≡

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <iostream>
#include <map>
#include "depth_first_search.hpp"

int main()
{
    typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> G;
    typedef graph_traits<G>::vertex_descriptor vertex_t;
    typedef graph_traits<G>::edge_descriptor edge_t;
    typedef std::map<vertex_t, boost::default_color_type> ColorMap;
    typedef boost::associative_property_map<ColorMap> C;

    ColorMap cm;
    C c(cm);

    G g(4);
    add_edge(0, 1, g);
    add_edge(0, 2, g);
    add_edge(1, 3, g);
    add_edge(2, 3, g);

    {  // (Print depth first ordering of vertices 1)
    }

    {  // (Print depth first ordering of edges 2a)
    }
```
The reader should have knowledge of graph concepts; familiarity with the Boost Graph Library will be helpful but is not required.

Performance testing indicates that our algorithm object is usually somewhat slower than the BGL implementation, but that both share the same $O(|V| + |E|)$ bound. Since our approach is primarily designed to simplify programming, and since the BGL has undergone much more review and optimization than our code has, we consider this result a success.

## 2 Reference Manual

The following reference manual is based on the Boost Graph Library reference manual.

### Notation

The notation used in this section is collected here.

- $D$ is the DepthFirstSearch type
- $dfs$ is an object of type $D$
- $C$ is the color map type
- $G$ is a type that is a model of Graph
- $g$ is an object of type $G$
- $u$ is an object of type `graph_traits$G$$::$vertex_descriptor`
- $e$ is an object of type `graph_traits$G$$::$edge_descriptor`
- $cp$ is a type that specifies a control point
- $cpl$ is a type that specifies a list of control points
- $i$ is an object of type $D$$::$iterator;
- $vi$ is an object of type $D$$::$vertex_iterator;
- $ei$ is an object of type $D$$::$edge_iterator;
2.1 DepthFirstSearch

template<typename VertexListGraph, typename ColorMap>
class DepthFirstSearch;

The DepthFirstSearch object performs a depth-first traversal of the vertices in a directed or undirected vertex-list graph. When possible, a depth-first traversal chooses a vertex adjacent to the current vertex to visit next. If all adjacent vertices have already been discovered, or there are no adjacent vertices, then the algorithm backtracks to the last vertex that had undiscovered neighbors. Once all reachable vertices have been visited, the algorithm selects from any remaining undiscovered vertices and continues the traversal. The algorithm finishes when all vertices have been visited. Depth-first search is useful for categorizing edges in a graph, and for imposing an ordering on the vertices.

Color markings are used to keep track of which vertices have been discovered. White marks vertices that have yet to be discovered, gray marks a vertex that is discovered but still has vertices adjacent to it that are undiscovered. A black vertex is a discovered vertex that is not adjacent to any white vertices.

The DepthFirstSearch object yields at the specified control points, defined by the object. This provides a mechanism for adapting the generic DFS algorithm to the many situations in which it can be used. The DepthFirstSearch algorithm object is advanced using the advance member functions, until it has finished. Users can halt the algorithm at anytime, since they have full control over when to advance to the next state. There are seven control points for the algorithm: StartVertex, DiscoverVertex, ExamineEdge, TreeEdge, BackEdge, ForwardOrCrossEdge, and FinishVertex. Three of the states are vertex states and have a valid vertex descriptor associated with them, and the other four states are edge states and have a valid edge descriptor associated with them.

Where Defined

depth_first_search.hpp

Associated Types

DFSControlPoints
The different states of a depth first search algorithm. The following are all the different control points:

**DFSCP_START_VERTEX**
The start vertex state is set when the algorithm starts its traversal from a specific vertex. Not all vertices will have a start vertex state. The first vertex for each disjoint graph will have a start vertex state associated with it.

**DFSCP_DISCOVER_VERTEX**
The discover vertex state is set once and only once for each vertex in the graph. It is called when the vertex is first discovered.

**DFSCP_EXAMINE_EDGE**
The examine edge state is set whenever an edge is about to be traversed.

**DFSCP_TREE_EDGE**
The tree edge state is set after the examine edge state, whenever an edge is traversed toward an undiscovered vertex.

**DFSCP_BACK_EDGE**
The back edge state is set after the examine edge state, whenever an edge is traversed toward a currently traversed vertex. This state only gets entered for graphs that are cyclic.

**DFSCP_FORWARD_OR_CROSS_EDGE**
The forward or cross edge state is set after the examine edge state, whenever an edge is traversed toward a vertex that has already been finished.

**DFSCP_FINISH_VERTEX**
The finish vertex state is set for all vertices in the graph. It is set after all adjacent vertices have been finished.

D::iterator
Input iterator type (see next section for full reference).

D::vertex_iterator
Input iterator type (see section ... ).

d::edge_iterator
Input iterator type (see section ... ).
Valid Expressions

DepthFirstSearch(G, C, cp)
Return type: D
Semantics:
This is a constructor of DepthFirstSearch type. G is the graph object, and C is the color map used for the depth first search algorithm. cp is an optional parameter, determining which control point to start the algorithm at (defaults to START_VERTEX).

DepthFirstSearch(G, C, u, cp)
Return type: D
Semantics:
This is a constructor of DepthFirstSearch type. G is the graph object, and C is the color map used for the depth first search algorithm. u is the start vertex of the depth first search algorithm. cp is an optional parameter, determining which control point to start the algorithm at (defaults to START_VERTEX).

dfs.finished()
Return type: bool
Semantics:
The function returns true if the algorithm has finished, otherwise it returns false.

dfs.currentControlPoint()
Return type: cp
Semantics:
This function returns the algorithms current control point. This value is undefined if dfs.finished() is true.

dfs.currentVertex()
Return type: u
Semantics:
This function returns the algorithm’s current vertex. This value is undefined if \( \text{dfs.currentControlPoint}() \) is not one of the following: \text{DFSCP\_START\_VERTEX}, \text{DFSCP\_DISCOVER\_VERTEX}, or \text{DFSCP\_FINISH\_VERTEX}.

\textbf{dfs.currentEdge()}

Return type: \( e \)

Semantics:

This function returns the algorithm’s current edge. This value is undefined if \( \text{dfs.currentControlPoint}() \) is not one of the following: \text{DFSCP\_EXAMINE\_EDGE}, \text{DFSCP\_BACK\_EDGE}, \text{DFSCP\_TREE\_EDGE}, \text{DFSCP\_FORWARD\_OR\_CROSS\_EDGE}.

\textbf{dfs.advance(cpl)}

Return type: \text{void}

Semantics:

This function advances the algorithm to the next control point. This function has an undefined behavior if \( \text{dfs.finished}() \) is true. Calling this function invalidates the last \( \text{dfs.currentControlPoint}() \), \( \text{dfs.currentVertex}() \), \( \text{dfs.currentEdge}() \), \( \text{dfs.finished}() \), and \( \text{dfs.it}() \). \( \text{dfs.advance}() \) takes an optional parameter, which specifies which control points to advance to; it returns when the first one reached.

\textbf{dfs.reset(cp)}

Return type: \text{void}

Semantics:

This function resets the algorithm to an initial state.

\textbf{dfs.it(cpl)}

Return type: \( i \)

Semantics:

This function returns an input iterator, of the depth first search algorithm. The iterator is directly bound to the algorithm object. Thus if the associated object changes, then so will the iterator. The \( \text{it}() \) function takes an optional \( \text{cpl} \), which specifies which control points the iterator should stop at.

\textbf{dfs.end()}

7
Return type: i
Semantics:
This function returns an input iterator that signifies the end of the depth first search algorithm. It is used for (in)equality checks against dfs.it(), to determine if the algorithm has finished.

**dfs.vertex_it(cpl)**

Return type: vi
Semantics:
This function returns an input iterator, of the depth first search algorithm. The iterator is directly bound to the algorithm object. Thus if the associated object changes, then so will the iterator. The it() function takes an optional cpl, which specifies which control points the iterator should stop at.

**dfs.vertex_end()**

Return type: vi
Semantics:
This function returns an input iterator that signifies the end of the depth first search algorithm. It is used for (in)equality checks against dfs.it(), to determine if the algorithm has finished.

**dfs.edge_it(cpl)**

Return type: ei
Semantics:
This function returns an input iterator, of the depth first search algorithm. The iterator is directly bound to the algorithm object. Thus if the associated object changes, then so will the iterator. The it() function takes an optional cpl, which specifies which control points the iterator should stop at.

**dfs.edge_end()**

Return type: ei
Semantics:
This function returns an input iterator that signifies the end of the depth first search algorithm. It is used for (in)equality checks against dfs.it(), to determine if the algorithm has finished.
2.2 DepthFirstSearch\textless G, C\textgreater ::iterator

DepthFirstSearch\textless G, C\textgreater ::iterator is an input iterator type which allows the user to adapt a depth first search algorithm to work with iterator algorithms, such as those supplied with STL. DepthFirstSearch iterators are simple adapters to the algorithm object. The default iterator has the algorithm object as its value type. So you can access the current control point, as well as the current vertex and edge values. In the following sections you will see refinements of the default iterator, which have vertex_descriptors and edge_descriptors as their value types.

Refinement of
input_iterator

Where defined
depth_first_search.hpp

Associated Types

\texttt{i::iterator\_category}

The iterator category, used for dispatching overloaded functions in STL algorithms. A D::iterator is typedef’d as an input\_iterator\_tag.

\texttt{i::difference\_type}

NOT USED. Difference value when two iterators are subtracted. Since this is an input iterator, subtracting has no definitive meaning.

\texttt{i::value\_type}

Value of the iterator when it is dereferenced. For D::iterator, this is simply the associated algorithm object(of type D).

\texttt{i::pointer}

Pointer type, to i::value\_type.

\texttt{i::reference}

Reference type, to i::value\_type.
Valid Expressions

+++i
Return type: D::iterator
Semantics:
Advances the DepthFirstSearch algorithm object associated with this iterator to the next valid control point. Since this is an input_iterator, all other iterators associated with this DepthFirstSearch algorithm object also advance.

i++
Return type: D::iterator
Semantics:
Advances the DepthFirstSearch algorithm object associated with this iterator to the next valid control point. Since this is an input_iterator, all other iterators associated with this DepthFirstSearch algorithm object also advance.

*i
Return type: D
Semantics:
Returns the algorithm object type associated with this iterator.

i->func()
Return type: D
Semantics:
Calls the member function func, of the associated DepthFirstSearch object.

i == i
Return type: bool
Semantics:
Compares two iterators for equality. All iterators associated with the same DepthFirstSearch object are the same. i == dfs.end(), only when dfs.finished() is true.

i != i

10
Return type: bool
Semantics:

Compares two iterators for inequality. All iterators associated with the same DepthFirstSearch object are the same. i != dfs.end(), whenever dfs.finished() is false.

2.3 DepthFirstSearch::vertex_iterator

The vertex iterator is implemented in terms of the default iterator, the only difference being, that the vertex iterator has a value type of vertex_descriptor_t, rather than DepthFirstSearch. This iterator can be used to iterate over a set of vertex control points.

Refinement of

DepthFirstSearch<G, C>::iterator

Valid Expressions
*vi

Return type: u
Semantics:

Returns the vertex_descriptor associated with the current control point of the algorithm (returns dfs.currentVertex(), on the associated algorithm object).

2.4 DepthFirstSearch::edge_iterator

The edge iterator is implemented in terms of the default iterator, the only difference being, that the edge iterator has a value type of edge_descriptor_t, rather than DepthFirstSearch. This iterator can be used to iterate over a set of edge control points.

Refinement of

DepthFirstSearch<G, C>::iterator
Valid Expressions

*ei

Return type: e
Semantics:

Returns the edge descriptor associated with the current control point of the algorithm (returns dfs.currentEdge(), on the associated algorithm object).

3 Design Issues and Source Code

This chapter will discuss the design issues and source code associated with implementing algorithms as algorithm objects, specifically the depth first search algorithm. We will discuss and attempt to resolve as many of the issues as possible, such as what should the value type of a traversal iterator be? Or, assuming the iterator stops at some number of control points, what if you only need it to stop at a very limited subset of those control points? How do you determine what control point the iterator is currently referring too? Is it possible to extend the iterator from an input iterator to a forward iterator, and if so, what are the trade-offs? Also, is it possible to execute the algorithm in reverse? All of these issues are discussed, as well as a complete overview of all the source code, in this chapter.

(DM: I don’t see anywhere that you discuss execution in reverse.)

3.1 Designing the basic interface

In this section we discuss the design of the basic interface and semantics of an algorithm object. The first issue is, how is an algorithm object different from a visitor algorithm? We then go on to discuss some basic terminology associated with algorithm objects, such as control points. We conclude the section by looking at what a depth first search algorithm does, and come up with its interface.

3.1.1 Algorithm objects vs. visitor-based algorithms

The algorithm object presented in this paper is intended as an alternative to the conventional depth first search (DFS) algorithm as implemented in the Boost Graph Library (BGL). Many of the BGL algorithms, DFS included, are parameterized by a visitor. Essentially a visitor is defined by a class that has a set of member functions that are invoked at different stages of
the algorithm. The user calls the algorithm, passing in a visitor object, and the algorithm moves from state to state, calling the object’s member functions. An algorithm object, on the other hand, advances from one state to the next, and at each state it yields control back to the user. This gives the user more control than in the visitor approach. The user can stop the algorithm at any point, by ceasing to advance the object’s state. Another important advantage is that the algorithm can break up computations over sections of time if this needs to be in a real time system; you can, say, give the depth first search one millisecond of time during each frame.

There are some drawbacks to the algorithm object approach, however. One drawback relative to the visitor approach is that the visitor’s members are called by the algorithm; however, if the user only defines one of the states (such as start vertex), the compiler can easily optimize out the calls of empty functions, resulting in essentially the same efficiency as with ordinary code unmodified by visitor calls. We also have issues with cache coherency. For an algorithm object, the state is always returned to the user, whose code gets executed, but then to advance to the next state, we have to enter back into the algorithm code. Thus, there is a good possibility that all this code (both the user’s and the algorithm’s) will not be maintained in the cache, causing much slowdown over the simple visitor approach. One final drawback is that the depth first search algorithm is recursive. When implementing this as a non-recursive algorithm we will be using an explicit stack (such as STL’s stack). Because of this we can expect performance penalties, due to the data being allocated in the free store rather than the stack, as well as some minor penalties such as function calls (i.e., pop, push, etc., if they are not inlined) and also checking if the stack needs to grow.

(DM: By “free store” do you mean heap storage? Why would the explicit stack be allocated in the heap? It doesn’t appear that you do that in your implementation. If it’s not in the heap, it’s less clear why the explicit stack operations should be less efficient than the language-implemented stack operations.)

3.1.2 What DepthFirstSearch needs to do

The depth first search algorithm traverses the graph by recursively visiting the adjacent vertices of each vertex. Whenever a vertex that has already been discovered is encountered, the algorithm backtracks to the most recent unexplored edge.

It seems logical then to have a current vertex of the algorithm. We can also make the algorithm more powerful with a current edge. The algorithm
will have many different control points, so the user will also need access to
the current control point. Each control point will be intuitively associated
with either an edge or a vertex.

3.1.3 The interface

It is finally time to start putting all the design together and build the first
pass of the depth first search algorithm object’s interface.

```
(DepthFirstSearch Interface 14a) ≡

template <class VertexListGraph, class ColorMap>
class DepthFirstSearch
{
    public:
        (Some useful typedefs 14b)
        (Constructors 15a)
        (Getting the details of the current state 15b)
        (Controlling the algorithm’s state 15c)

    public:
        (Extended interface 31a)

    private:
        (Implementation details 20)
};
```

Used in part 28.

This basic interface is templated on two types. First, it’s templated on the
graph type, so that it works with any BGL VertexListGraph. The second
template parameter is the color map. Both will be described in greater
detail in the next section.

The interface is split up into five sections. The implementation details
are discussed in the next section, and the Extended Interface is discussed in
the section following the next. The first thing to look at is some typedefs:

```
(Some useful typedefs 14b) ≡
```
typedef typename
graph_traits<VertexListGraph>::vertex_descriptor vertex_descriptor_t;
typedef typename
graph_traits<VertexListGraph>::edge_descriptor edge_descriptor_t;

Used in part 14a.

Here we have some typical typedefs of the graphs vertex_descriptor and edge_descriptor.

(Constructors 15a) ≡

DepthFirstSearch(const VertexListGraph&, ColorMap, DFSControlPoints);
DepthFirstSearch(const VertexListGraph&, ColorMap,
vertex_descriptor_t, DFSControlPoints);

Used in part 14a.

The interface supports two constructors, one that takes a vertex_descriptor and one that does not. They both take a VertexListGraph and a ColorMap. The DFSControlPoints parameter tells the algorithm at which control point to start, and the optional vertex_descriptor gives a starting vertex for the search.

(Getting the details of the current state 15b) ≡

DFSControlPoints currentControlPoint() const;
vertex_descriptor_t& currentVertex();
const vertex_descriptor_t& currentVertex() const;
edge_descriptor_t& currentEdge();
const edge_descriptor_t& currentEdge() const;
bool finished() const;

Used in part 14a.

As discussed earlier, the algorithm will need to supply the user with the current vertex, current edge, and current control point. It will also need the functionality to tell the user the algorithm is done. That is the only information needed by the user of the algorithm. We now come to the final part of the public interface:

(Controlling the algorithm’s state 15c) ≡
void advance(int);
void reset(DFSControlPoints);
void reset(vertex_descriptor_t, DFSControlPoints);

Used in part 14a.

Advance advances the algorithm to the next state. It takes an optional parameter which is a set of control points that are valid. It will therefore not stop at a state not in the set. The reset member is similar to the constructor. It resets the algorithm to the beginning, with an optional start vertex.

And that is the basic interface. The next section goes into the implementation of the algorithm.

3.2 Implementation of the algorithm

This section goes deep into the source code, and begins the implementation of the algorithm object. We first start off by looking at how BGL implemented their depth-first-search algorithm. We then decide on the control points needed by our algorithm, so that it’s just as powerful, if not more so, than BGL’s depth-first-search algorithm. After that we begin to discuss how it will be implemented, and finally finish off by implementing the algorithm.

3.2.1 A not so in-depth look at the BGL depth_first_search

BGL’s depth_first_search algorithm takes in three parameters: The graph, a color map, and a DFSVisitor. The DFSVisitor must contain the following members: initialize_vertex, start_vertex, discover_vertex, examine_edge, tree_edge, back_edge, forward_or_cross_edge, and finish_vertex. These members are a good indication of which control points, the depth first search algorithm object should define. If the algorithm object is to be as powerful as BGL’s, then it needs to define all of them. (With the exception of initialize_vertex, which will be explained later).

Next we look at a pseudocode version of BGL depth_first_search:

\[
\text{(BGL DFS pseudocode 16) } \equiv \\
\text{DFS}(G) \\
\quad \text{for each vertex } u \\
\quad \quad \text{color}[u] = \text{WHITE} \\
\quad \quad \text{initialize_vertex}(u)
\]

16
for each vertex u
    if color[u] = WHITE
        start_vertex(u)
        call DFS-VISIT(G, u)

DFS-VISIT(G, u)
    color[u] = Gray
    discover_vertex(u)
    for each adjacent_vertex v
        if (color[v] = WHITE)
            examine_edge(u, v)
            call DFS-VISIT(G, v)
        else if (color[v] = GRAY)
            tree_edge(u, v)
        else if (color[v] = BLACK)
            cross_or_forward_edge(u, v)
    color[u] = BLACK

Not used.

Here we see that we will need a ColorMap, to label vertices, as undiscovered (WHITE), currently being processed (GRAY), and finished (BLACK). We also see that we will need to iterate over all the vertices in the graph, and in order to do this we will need at least a VertexListGraph. The algorithm also needs to be changed to be nonrecursive.

3.2.2 Which control points to borrow?

It’s now time to talk about which control points the algorithm will need. Here we will go through all the control points defined by BGL visitor, and discuss if we need them in the algorithm object.

initialize_vertex is the first visitor function called by the algorithm. It simply iterates through all the vertices, and initializes them. It does not seem that we need this kind of functionality with an algorithm object, because the algorithm object, would just give control back to the user. So it seems better to just let users initialize all its vertices before running the algorithm, if they wish to do so.

start_vertex is the next visitor function called by the algorithm. This is called whenever a new search is started from a disjoint graph. It therefore has special functionality, unlike initialize_vertex, and should be one of the control points.

discover_vertex is called next, whenever a vertex is first discovered. This also has special functionality, because of the order in which it is called.
It is called for all vertices, but in the order in which they are discovered. Therefore this should also be one of the control points of the algorithm.

**finish_vertex** is called when the algorithm has finished processing this vertex, and all its children. This, like **discover_vertex** is called for all vertices in the graph, but its order is what gives this control point its uniqueness, and should also be one of the control points of the algorithm.

**examine_edge** is the first edge visitor function the algorithm calls. It is called for each edge, but the order in which it is called, makes this a good candidate for a control point.

The next three edges are very unique, and will thus be kept as control points. Each edge falls into one and only one of the following categories:

- **tree_edge** is called on an edge that leads to an undiscovered vertex.
- **back_edge** is called on an edge that leads to a currently-being-processed vertex.
- **forward_or_cross_edge** is called on an edge that leads to a vertex that has already been finished.

Those are all the control points that uniquely define the BGL’s depth first search. The depth first search algorithm object will use all of them, except for the **initialize_vertex** control point, as it is mostly useless, for an algorithm object. We now define an enum that will hold all such control points:

```cpp
enum DFSControlPoints {
    DFSCP_START_VERTEX = 1,
    DFSCP_DISCOVER_VERTEX = 2,
    DFSCP_EXAMINE_EDGE = 4,
    DFSCP_TREE_EDGE = 8,
    DFSCP_BACK_EDGE = 16,
    DFSCP_FORWARD_OR_CROSS_EDGE = 32,
    DFSCP_FINISH_VERTEX = 64,
};
```

The control points are numbered such that they can be easily bitwise or’d together to form a set of control points.

### 3.2.3 Recursive function to a not so recursive function

Depth first search is a recursive algorithm, however an algorithm object must yield at specified control points in the algorithm. These two concepts
do not work well together. Therefore the algorithm must be implemented as a nonrecursive function first of all. It must then be implemented so that it can stop itself and start back up later, where it left off.

The first thing to do is convert it to a nonrecursive algorithm. A recursive function is one in which it calls itself, until it reaches some sort of base case, where it starts unwinding its calls. The simplest way (possibly the only real way) to implement a recursive function is with a stack. Whenever you call the recursive function, you push all the variables needed to be saved onto the stack, and start the function over. When you reach the end of the function, you simply pop the variables back off the stack, and go back to where you last called the recursive function from. Luckily in the depth first search algorithm, there is only one place that calls the function recursively, so there is no need to keep track of where it got called. And looking at the pseudocode, the only data that needs to be pushed on the stack is the range of edges, and the current vertex.

3.2.4 What kind of data members?

It’s now time to decide on what private members are needed in order to implement the algorithm. We need to store the graph object, so we have something to work with. We also need to store the color map, which is used specifically by the depth first search algorithm. We also need a stack, as mentioned above, that will hold a range of edges (as edge iterators), and a current vertex. The algorithm, also has to iterate over all of the vertices in the graph, once one disjoint graph set is finished, therefore we need a vertex iterator to hold that. It should also hold the current control point, current edge, and current vertex, which are needed by the public interface of the algorithm.

\[
\text{Private member variables 19} \equiv
\]

\[
\begin{align*}
&\text{const VertexListGraph& graph_;} \\
&\text{ColorMap color_;} \\
&\text{struct stack_object_t \{} \\
&\quad \text{edge_iterator_t first;} \\
&\quad \text{edge_iterator_t second;} \\
&\quad \text{vertex_descriptor_t vert;} \\
&\quad \text{stack_object_t(const vertex_descriptor_t& v,} \\
&\quad \quad \text{const std::pair<edge_iterator_t,} \\
&\quad \quad \quad \text{edge_iterator_t>& e)} \\
\end{align*}
\]
: first(e.first), second(e.second), vert(v)

{};

stack_object_t(const vertex_descriptor_t& v)
  : vert(v)
{};
std::stack<stack_object_t> working_;

vertex_iterator_t current_base_vertex_;

DFSControlPoints state_;
vertex_descriptor_t current_vertex_;
edge_descriptor_t current_edge_;
Constructor Implementation 21a ≡

```
template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>::DepthFirstSearch(  
    const VertexListGraph& graph,  
    ColorMap color,  
    DFSControlPoints start_state = DFSCP_START_VERTEX)
    : graph_(graph),  
      color_(color),  
      state_(DFSCP_START_VERTEX)
{
    if (vertices(graph_).first != vertices(graph_).second) {
        current_vertex_ = *vertices(graph_).first;
        init(start_state);
    }
}
```

```
template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>::DepthFirstSearch(  
    const VertexListGraph& graph,  
    ColorMap color,  
    vertex_descriptor_t start_vertex,  
    DFSControlPoints start_state = DFSCP_START_VERTEX)
    : graph_(graph),  
      color_(color),  
      state_(DFSCP_START_VERTEX),  
      current_vertex_(start_vertex)
{
    init(start_state);
}
```

Used in part 29.

As one can see, the constructors are set up very simply. They assign the current graph and color map. They also set the start state to `start_vertex`, and then they call the private `init` function. The only difference between the two constructors is that one of them sets the current vertex to the user supplied start vertex, while the other sets it to the first vertex in the graph (if one exists).

The init function 21b ≡

```
template <class VertexListGraph, class ColorMap>
void DepthFirstSearch<VertexListGraph, ColorMap>::init(
```
DFSControlPoints start_state)
{
    vertex_iterator_t ui, ui_end;
    for (tie(ui, ui_end) = vertices(graph_); ui != ui_end; ++ui)
        put(color_, *ui, color_t::white());
    current_base_vertex_ = vertices(graph_).first;
    working_.push(stack_object_t(current_vertex_,
        out_edges(current_vertex_, graph_)));
    if (state_ != start_state) {
        advance(start_state);
    }
}

Used in part 29.

The init function first initializes the color map, so all vertices are colored white. It then sets the vertex iterator for the entire graph, to the first vertex. It then proceeds to set up the stack. Finally it advances the state to the start_state specified by the user. The next members to implement are the reset functions, which are very similar to the constructors:

(The reset functions 22) ≡

    template <typename T>
    inline void pop_all(T& s)
    {
        while (!s.empty()) {
            s.pop();
        }
    }

    template <class VertexListGraph, class ColorMap>
    void DepthFirstSearch<VertexListGraph, ColorMap>::reset(
        DFSControlPoints state = DFSCP_START_VERTEX)
    {
        if (vertices(graph_).first != vertices(graph_).second) {
            reset(*vertices(graph_).second, state);
        } else {
            pop_all(working_);
        }
    }
template <class VertexListGraph, class ColorMap>
void DepthFirstSearch<VertexListGraph, ColorMap>::reset(
    vertex_descriptor_t start_vertex,
    DFSControlPoints state = DFSCP_START_VERTEX)
{
    pop_all(working_);
    state_ = state;
    if (vertices(graph_).first != vertices(graph_).second) {
        current_vertex_ = *vertices(graph_).first;
        init(state);
    }
}

Used in part 29.

The reset functions simply remove all elements from the working stack. It then sets the current state to the new state, sets the current vertex, and then calls the init function. The last part of the implementation, before getting to the advance function, is the public interface:

(Public interface implementation 23) ≡

 template <class VertexListGraph, class ColorMap>
 bool DepthFirstSearch<VertexListGraph, ColorMap>::finished() const
 {
     return working_.empty();
 }

 template <class VertexListGraph, class ColorMap>
 DFSControlPoints
 DepthFirstSearch<VertexListGraph, ColorMap>::currentControlPoint() const
 {
     return state_; 
 }

 template <class VertexListGraph, class ColorMap>
 typename DepthFirstSearch<VertexListGraph, ColorMap>::vertex_descriptor_t&
 DepthFirstSearch<VertexListGraph, ColorMap>::currentVertex()
 {
     return current_vertex_; 
 }

23
The implementation of the public interface is rather straightforward. They are just accessor functions to our private members. We now have the interface and implementation set up, except for the advance functions, which will be done in the next section.

3.2.6 Advance

The advance function loops over the advance_internal function until either the algorithm is finished, or the current control point is one that is in the set of control points passed in as the first parameter. The first parameter is defaulted to 0xff, which contains all of the control points.

\[
\text{Advance Function 24) } \equiv \text{template <class VertexListGraph, class ColorMap>}
\]

\[
\text{void DepthFirstSearch< VertexListGraph, ColorMap>::advance(}
\]

\[
\text{ int cpoints = 0xff) }
\]

\[
\{ 
\text{ do { }
\text{ advance_internal(); }
\}
\]
while (!finished() && !(cpoints & state_));
}

The `advance_internal` function is as follows:

(Internal Advance Function 25a) ≡

```
template <class VertexListGraph, class ColorMap>
void DepthFirstSearch<VertexListGraph, ColorMap>::
    advance_internal()
{
    switch (state_)
    {
    case DFSCP_TREE_EDGE:
        (Tree Edge State 27a)
        break;
    case DFSCP_START_VERTEX:
        (Start Vertex State 25b)
        break;
    case DFSCP_FINISH_VERTEX:
        (Finish Vertex State 27b)
        break;
    case DFSCP_DISCOVER_VERTEX:
        break;
    case DFSCP_BACK_EDGE:
        (Discover Vertex, Back Edge, and ForwardCross Edge State 26a)
        break;
    case DFSCP_EXAMINE_EDGE:
        (Examine Edge State 26b)
        break;
    }
}
```

(Start Vertex State 25b) ≡

```
put(color_, current_vertex_, color_t::gray());
state_ = DFSCP_DISCOVER_VERTEX;
```

Used in part 29.

The `advance_internal` function is a simple state machine. First we start by looking at the `start_vertex` state:

(Start Vertex State 25b) ≡

```
put(color_, current_vertex_, color_t::gray());
state_ = DFSCP_DISCOVER_VERTEX;
```
Before entering the start state, the algorithm is guaranteed to have at least one item on the stack. This is done whenever you enter the start vertex state. This state is very simple, if you look at the pseudocode. If you advance from the `start_vertex` state, it simply calls DFS-VISIT, which sets the current vertex color to gray, and enters the discover vertex state. So the next state the algorithm would enter is:

\[
\text{(Discover Vertex, Back Edge, and ForwardCross Edge State 26a) } \equiv
\]

\[
\begin{align*}
\text{if (working_.top().first == working_.top().second) } & \{
  \text{put(color_, current_vertex_, color_t::black());}
  \text{state_ = DFSCP_FINISH_VERTEX;}
\} & \text{ else } & \{
  \text{current_edge_ = *working_.top().first++;}
  \text{state_ = DFSCP_EXAMINE_EDGE;}
\}
\end{align*}
\]

Used in part 25a.

First, assume that the algorithm is advancing from the start vertex state. Then there must be at least one item on the stack. The algorithm first checks if there are any edges left; since we just put this pair of edges on the stack, if there are no edges, we skip over the for loop in the pseudocode, and then arrive at the finish vertex state. If on the other hand, there are edges, then we enter the examine edge state with the current edge, and advance the edge iterator pair to the next edge.

Now looking ahead in the algorithm, we see that if it was in the `back_edge` state, it will then start the loop check over again, just as if we had just entered the loop. The same goes for the `forward_or_cross_edge` state. Therefore all three control points should have the same effect.

The next control point to look at is the `examine_edge` state:

\[
\text{(Examine Edge State 26b) } \equiv
\]

\[
\begin{align*}
\{ & \text{const color_value_t v_color =}
  \text{get(color_, target(current_edge_, graph_));}
  \text{if (v_color == color_t::white()) } \{
    \text{state_ = DFSCP_TREE_EDGE;}
  } & \text{ else if (v_color == color_t::gray()) } \{
    \text{state_ = DFSCP_BACK_EDGE;}
  } & \text{ else } & \{
\}
\end{align*}
\]
The `examine_edge` control point is extremely straightforward. It checks the color of the target vertex of the current edge and dispatches to the specific edge control point. We know how the `back_edge` and `forward_or_cross_edge` states are handled, so the next state to look at is the `tree_edge` state:

```
⟨Tree Edge State 27a⟩ ≡

  working_.push(stack_object_t(current_vertex_));
  current_vertex_ = target(current_edge_, graph_);
  tie(working_.top().first, working_.top().second) =
      out_edges(current_vertex_, graph_);
  // FALLTHROUGH
```

Used in part 25a.

The tree edge is where the recursion comes into play. In the pseudocode, the DFS-VISIT function is called recursively. So, naturally that means the stack needs to be updated. The first thing to do is push the current vertex onto the stack, and then update the current vertex, to point to the current edges target. We then put the new edge pair on the stack. Looking at the pseudocode, the first thing DFS-VISIT does is color the current vertex gray, and set the state to `discover_vertex`. This is the same thing our `start_vertex` state does. So the simple thing to do is let it fall through to the `start_vertex` state.

We now have almost finished the implementation of the algorithm. The last state to do is `finish_vertex`:

```
⟨Finish Vertex State 27b⟩ ≡

  current_vertex_ = working_.top().vert;
  working_.pop();

  if (working_.empty()) {
    for (; current_base_vertex_ != vertices(graph_).second;
        ++current_base_vertex_)
  }
```

27
current_vertex_ = *current_base_vertex_;  
if (get(color_, current_vertex_) == color_t::white()) {
    state_ = DFSCP_START_VERTEX;
    working_.push(stack_object_t(current_vertex_,
                                out_edges(current_vertex_, graph_)));
    break;
}
}  
break;
}  
//FALLTHROUGH

Used in part 25a.

When finish_vertex is reached in the pseudocode, it has reached the end of the recursive function. It must therefore unwind the stack one level. We do this by updating the current vertex and popping off the stack. There are two cases after popping the item off the stack. One is that we have finished all recursive calls and should be back in the DFS function. Or, we are still in DFS-VISIT. If the stack is empty, then we are in the first case. What does it mean if we are back in DFS function? It means we have to iterate through the rest of the graph and check for white vertices, to perform the algorithm once again. And that is exactly what it does. If the stack is empty, it continues looping through all the vertices in the graph. If it comes across a white vertex, it sets up the stack again, and enters the start_vertex state. On the other hand, if the stack is not empty, then it needs to continue where it left off. It’s the same case as the back_edge and forward_or_cross_edge states; therefore, we just fall through into that state.

The algorithm is now complete. It will iterate through all the defined control points, yielding to user defined code at each one.

### 3.2.7 Putting it all together

We now have a fully working algorithm, with a nice public interface to it. Here we finally put all the pieces together, to create the basic depth first search algorithm object:

```
"depth_first_search.hpp" 28 ≡

#ifndef _DFS_H_
#define _DFS_H_

#define _DFS_H_
```
#include <boost/graph/properties.hpp>
#include <stack>
#include <iterator>

using boost::graph_traits;

(Control Points 18)

(DepthFirstSearch Interface 14a)

(DepthFirstSearch Implementation 29)

#Endif

(DepthFirstSearch Implementation 29) ≡

(Constructor Implementation 21a)
(The init function 21b)
(The reset functions 22)
(Public interface implementation 23)
(Advance Function 24)
(Internal Advance Function 25a)
(Extended interface implementation 38)

Used in part 28.

3.3 Extending the interface (Iterators)

We now have a complete algorithm object. We give it a graph, and a color map, and tell it to advance away. We now plan to extend upon this interface, and adapt some iterators to work with this algorithm object. This will allow the depth first search algorithm object to be used in already-created algorithms, such as STL algorithms. The first thing to decide is what iterator category to refine from. We then discuss what additional requirements might be placed on our chosen iterator category, and justify implementing multiple types of iterators, as needed for different tasks.
3.3.1 What iterator category?

The basic iterator that we would need would be the input iterator. In order

to have an input iterator, the following expressions must be valid: \( x, *x, *x = t, \) and \( x->m. \) It must also support these expressions: \( ++x, x++. \) So in

order for it to be an input iterator, the iterator needs to have some value
type, when dereferenced. One idea would be to give the algorithm object
reference, when the iterator is dereferenced. It also needs to support pre-
and post-increment operations. We can map those to the advance function
of the algorithm object. The default constructor and assignment operator
are trivial. So therefore, we can define an input iterator, which simply stores
a pointer to the algorithm object.

Could it also be a forward iterator? In order for it to be a forward
iterator, it would need to be usable in multi-pass algorithms. So, if the
iterator was copied, it would save the current state of the iterator. This
would not work correctly with the approach taken above, of just storing a
pointer to the object, because if one iterator changes the state of the object,
then all iterators change. One way to remedy this would be to store the
state of the algorithm in the iterator, which would be the stack and the
current vertex, state, and edge. This would work well, except for the fact
that iterators are frequently copied, such as when passed by value, which
could be very expensive. Therefore we to limit it to being only an input
iterator.

3.3.2 What does a compliant iterator need?

In dealing with getting an iterator to correctly compile with certain versions
of STL, a few typedefs must be defined in the iterator classes. \texttt{iterator\_category}

must be defined, as \texttt{input\_iterator\_tag, difference\_type} is defined as

\texttt{unsigned int, value\_type} is defined, as the value type when the object is
dereferenced. \texttt{pointer} and \texttt{reference} are typedef’d as a pointer to value
type and a reference to value type respectively.

3.3.3 Many iterators for different things?

Having the value type of the iterator being the algorithm object itself is very
limiting. Consider the examples in Section 1.2: sometimes we desire a ver-
tex descriptor, sometimes an edge descriptor and sometimes the algorithm
object itself. With that in mind, instead of just have one iterator type, we
create three: one that has a value type of DepthFirstSearch, one that has a
value type of DepthFirstSearch::vertex\_descriptor\_t, and one that has a value
type of `DepthFirstSearch::edge_descriptor_t`. That way, for example, if all you were interested in was vertices, you would just want a vertex iterator. In the next section we build the extended interface, based on iterators.

### 3.3.4 The iterator innards

The first thing to do is extend the basic public interface, to now support iterator assessors:

```c++
(Extended interface 31a) ≡
(Extended interface 31b) ≡
```

Used in part 14a.

```c++
(The basic iterator 31b) ≡
class iterator
{
public:
    typedef std::input_iterator_tag iterator_category;
    typedef int difference_type;
    typedef DepthFirstSearch value_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;

    iterator(DepthFirstSearch* = 0, int = 0xFF);

    iterator& operator++();
    const iterator operator++(int);

    const DepthFirstSearch& operator*() const;
    DepthFirstSearch& operator*();
    const DepthFirstSearch* operator->() const;
    DepthFirstSearch* operator->();

    bool operator==(const iterator&) const;
    bool operator!=(const iterator&) const;

protected:
    DepthFirstSearch* object_;
    int control_points_;
iterator it(int control_points = 0xFF);
iterator end() const;

Used in part 31a.

This is the basic iterator type. It has a value type of DepthFirstSearch. This type gives the user full access to the edge, vertex and state information. It supports the usual iterator operators, post/pre increment, dereferencing, and equality comparison operators. It is default constructible because it has two default parameters; therefore it is a valid input iterator.

(Basic iterator implementation 32a) ≡

```
template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::iterator(
    DepthFirstSearch* object,
    int control_points)
    : object_(object), control_points_(control_points)
{
}
```

```
template <class VertexListGraph, class ColorMap>
typename DepthFirstSearch<VertexListGraph, ColorMap>::iterator
DepthFirstSearch<VertexListGraph, ColorMap>::it(int cps)
{
    return iterator(this, cps);
}
```

```
template <class VertexListGraph, class ColorMap>
typename DepthFirstSearch<VertexListGraph, ColorMap>::iterator
DepthFirstSearch<VertexListGraph, ColorMap>::end() const
{
    return iterator();
}
```

Used in part 38.

The iterator constructor simply sets the DepthFirstSearch object and stores the control points over which to iterate. The it and end functions simply return an iterator.

(Basic iterator implementation 2 32b) ≡
template <class VertexListGraph, class ColorMap>
typeid DepthFirstSearch<VertexListGraph, ColorMap>::iterator &
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator++() {
    if (object_)
        object_->advance(control_points_);

    return *this;
}

template <class VertexListGraph, class ColorMap>
const typeid DepthFirstSearch<VertexListGraph, ColorMap>::iterator
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator++(int) {
    iterator tmp(*this);
    ++*this;
    return tmp;
}

template <class VertexListGraph, class ColorMap>
bool DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator==(const iterator & rhs) const {
    const bool lf = !object_ || object_->finished();
    const bool rf = !rhs.object_ || rhs.object_->finished();
    return (lf == rf);
}

template <class VertexListGraph, class ColorMap>
bool DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator!=(const iterator & rhs) const {
    return !(this == rhs);
}

Used in part 38.

The increment operators are defined here. They simply call advance, passing in the valid control points specified by the user (or 0xff by default). The operators == and != are defined here as well. An iterator is defined as the end iterator if its object pointer is null or its object pointer says it has finished. Therefore if both the left hand side and the right hand side are finished, or both not finished, then they are considered equal.
The last set of operators and member functions needed to be implemented for the basic iterator are its * and -> operators. Since the value type of iterator is DepthFirstSearch, it simply returns a reference and a pointer to the object.

Now that the basic iterator is defined, we look at a more specialized iterator:

```
(Basic iterator implementation 34a) ≡

template <class VertexListGraph, class ColorMap>
const DepthFirstSearch<VertexListGraph, ColorMap>&
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator*() const
{
    return *object_;  
}

template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>&
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator*()  
{
    return *object_;  
}

template <class VertexListGraph, class ColorMap>
const DepthFirstSearch<VertexListGraph, ColorMap>*
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator->() const
{
    return object_;  
}

template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>*
DepthFirstSearch<VertexListGraph, ColorMap>::iterator::operator->()  
{
    return object_;  
}
```

Used in part 38.

The last set of operators and member functions needed to be implemented for the basic iterator are its * and -> operators. Since the value type of iterator is DepthFirstSearch, it simply returns a reference and a pointer to the object. Now that the basic iterator is defined, we look at a more specialized iterator:

```
(The vertex iterator 34b) ≡

class vertex_iterator : public iterator
{
    public:

```
typedef vertex_descriptor_t value_type;
typedef const value_type* pointer;
typedef const value_type& reference;

vertex_iterator(DepthFirstSearch* = 0, int = DFSCP_START_VERTEX |
                DFSCP_DISCOVER_VERTEX |
                DFSCP_FINISH_VERTEX);

const vertex_descriptor_t& operator*() const;
vertex_descriptor_t& operator*();

vertex_iterator vertex_it(int cps = DFSCP_START_VERTEX|
                           DFSCP_DISCOVER_VERTEX|
                           DFSCP_FINISH_VERTEX);

vertex_iterator vertex_end() const;

Used in part 31a.

vertex_iterator derives publicly from iterator, because most of the functionality is the same. The only difference is that vertex_iterator has a value type of vertex_descriptor. This iterator does not have access to the full amount of information provided by iterator, such as the current edge or the current state. vertex_iterator’s constructor and the member function vertex_it both have default parameters, taking a set of control points. The default control points for the vertex iterator are the vertex control points, because the value of the vertex descriptor at an edge control point is undefined.

(Vertex iterator impl 35) ≡

template <class VertexListGraph, class ColorMap>  
typename DepthFirstSearch<VertexListGraph, ColorMap>::vertex_iterator
DepthFirstSearch<VertexListGraph, ColorMap>::vertex_it(int cps)  
{
  return vertex_iterator(this, cps);
}

template <class VertexListGraph, class ColorMap>  
typename DepthFirstSearch<VertexListGraph, ColorMap>::vertex_iterator
DepthFirstSearch<VertexListGraph, ColorMap>::vertex_end() const  
{
  return vertex_iterator();
}
template <class VertexListGraph, class ColorMap>
DepthFirstSearch<VertexListGraph, ColorMap>::vertex_iterator::vertex_iterator(
    DepthFirstSearch* object, int control_points)
{
}
}

template <class VertexListGraph, class ColorMap>
typename DepthFirstSearch<VertexListGraph, ColorMap>::vertex_descriptor_t&
DepthFirstSearch<VertexListGraph, ColorMap>::vertex_iterator::operator*()
{
    return object_->currentVertex();
}

template <class VertexListGraph, class ColorMap>
const typename DepthFirstSearch<VertexListGraph, ColorMap>::vertex_descriptor_t&
DepthFirstSearch<VertexListGraph, ColorMap>::vertex_iterator::operator*() const
{
    return object_->currentVertex();
}

Used in part 38.

The `vertex_it` and `vertex_end` member functions are the same as `it` and `end` except they return vertex iterators rather than normal iterators. Since the value type of a `vertex_iterator` is `vertex_descriptor`, the `*` operator must be overwritten, which is done by simply returning the current vertex of the object. The rest of the functionality of the vertex iterator is the same as the basic iterator. We now look at the last iterator, the edge iterator:

(The edge iterator 36) ≡

class edge_iterator : public iterator
{
public:
    typedef edge_descriptor_t value_type;
    typedef const value_type* pointer;
    typedef const value_type& reference;

    edge_iterator(DepthFirstSearch* = 0, int = DFSCP_EXAMINE_EDGE |
        DFSCP_TREE_EDGE | DFSCP_BACK_EDGE | DFSCP_FORWARD_OR_CROSS_EDGE);
const edge_descriptor_t& operator*() const;
    edge_descriptor_t& operator*();
};

edge_iterator edge_it(int cps = DFSCP_EXAMINE_EDGE|
                      DFSCP_TREE_EDGE|
                      DFSCP_BACK_EDGE|
                      DFSCP_FORWARD_OR_CROSS_EDGE);

edge_iterator edge_end() const;

Used in part 31a.

Again, the edge iterator is set up identically to the vertex iterator, except for a different value type, and different set of default control points. The value type of the edge_iterator is an edge_descriptor_t, and the default control points are the four edge control points. Here is a look at the implementation of the edge iterators:

(Edge iterator impl 37) ≡

```cpp
template <class VertexListGraph, class ColorMap>
    typename DepthFirstSearch<VertexListGraph, ColorMap>::edge_iterator
    DepthFirstSearch<VertexListGraph, ColorMap>::edge_it(int cps)
    {
        return edge_iterator(this, cps);
    }

template <class VertexListGraph, class ColorMap>
    typename DepthFirstSearch<VertexListGraph, ColorMap>::edge_iterator
    DepthFirstSearch<VertexListGraph, ColorMap>::edge_end() const
    {
        return edge_iterator();
    }

template <class VertexListGraph, class ColorMap>
    typename DepthFirstSearch<VertexListGraph, ColorMap>::edge_descriptor_t&
    DepthFirstSearch<VertexListGraph, ColorMap>::edge_iterator::operator*()
    {
    }
```
Used in part 38.

Again, the implementation is very simple. The only difference between this and the vertex iterator is that the * operator returns an edge descriptor, rather than a vertex descriptor.

Finally, we put all of this together:

\[
\langle \text{Extended interface implementation 38} \rangle \equiv \\
\langle \text{Basic iterator implementation 32a} \rangle \\
\langle \text{Basic iterator implementation 2 32b} \rangle \\
\langle \text{Basic iterator implementation 3 32a} \rangle \\
\langle \text{Vertex iterator impl 35} \rangle \\
\langle \text{Edge iterator impl 37} \rangle
\]

Used in part 29.

4 Test Plan

4.1 Correctness Testing

4.1.1 Framework

In order to perform correctness testing on our algorithm object, we must determine exactly what form the output takes, and what it should be. It might seem that the algorithm object on its own produces no output, just as the BGL depth_first_search produces no output when run with no visitor. Yet in fact, the output takes the form of the sequence of states of the algorithm. That is, for the BGL depth_first_search, the output is the sequence of calls to the member functions of the visitor object. For
our algorithm object, it is the sequence of control points reached, and the
current edge or vertex (as appropriate) at each control point.

Clearly we need to a special data structure to store the type of output,
in order to compare the results of running our algorithm object with the
results of the BGL depth_first_search (which we assume to be correct).
We define a template class called DFSState, which will hold the type of a
control point and the corresponding edge descriptor or vertex descriptor.

(Define DFSState class 39a) ≡

```
template <typename Graph>
class DFSState
{
    public:
        typedef typename graph_traits<Graph>::edge_descriptor edge_t;
        typedef typename graph_traits<Graph>::vertex_descriptor vertex_t;

        DFSControlPoints controlPoint;
        vertex_t vertex;
        edge_t edge;

    };  // Define DFSState
```

Used in part 43a.

We define three constructors. Each takes a DFSControlPoints as this mem-
ber is mandatory. Since a DFSState must have a vertex or an edge (or both),
the constructors allow for creating a DFSState from a control point and a
vertex, from a control point and an edge, or from all three.

(Define constructors 39b) ≡

```
DFSState(DFSControlPoints cp, vertex_t v) :
    controlPoint(cp), vertex(v) { }

DFSState(DFSControlPoints cp, edge_t e) :
    controlPoint(cp), edge(e) { }

DFSState(DFSControlPoints cp, vertex_t v, edge_t e) :
```
controlPoint(cp), vertex(v), edge(e) { }

Used in part 39a.

Because the members are all primitive data types, the default assignment operator is acceptable. However, we define our own comparison operators, since the vertex is not relevant for edge-related control points, and vice versa.

(Define == operator 40a) ≡

        bool operator==(const DFSState<Graph>& other) const
        {
            if (controlPoint != other.controlPoint)
                return false;

            if (controlPoint == DFSCP_START_VERTEX ||
                controlPoint == DFSCP_DISCOVER_VERTEX ||
                controlPoint == DFSCP_FINISH_VERTEX)
                return vertex == other.vertex;
            else
                return edge == other.edge;
        }

Used in part 39a.

We now define a function to iterate through the states of a DepthFirstSearch object, saving each control point to an output iterator (which is probably a back_inserter for a container, but could also be, for example, an ostream_iterator).

(Define saveDFS function 40b) ≡

        template <typename VertexListGraph, class ColorMap, class OutIt>
        void saveDFS(DepthFirstSearch<VertexListGraph, ColorMap>& dfs, OutIt out)
        {
            for (; !dfs.finished(); dfs.advance())
                *out++ = DFSState<VertexListGraph>(dfs.currentControlPoint(),
                                                  dfs.currentVertex(),
                                                  dfs.currentEdge());
        }

Used in part 43a.
In order to compare our implementation with that of the BGL, we also define a visitor that can be used with the Boost depth first search, which saves the states of the algorithm. The visitor is constructed with an output iterator, and each control point the algorithm reaches is written using that iterator.

\[(\text{Define saveStateVisitor 41a}) ≡\]

```cpp
template<typename Graph, typename OutIt>
struct saveStateVisitor : public boost::dfs_visitor<>
{
    typedef typename graph_traits<Graph>::vertex_descriptor vertex_t;
    typedef typename graph_traits<Graph>::edge_descriptor edge_t;

    OutIt out_;

    saveStateVisitor(OutIt out) : out_(out) {}
};
```

Used in part 43a.

Each member function outputs a DFSState based on the current control point and the current vertex or edge using the output iterator.

\[(\text{Define member functions 41b}) \equiv\]

```cpp
void start_vertex(vertex_t v, const Graph& g)
{
    *out_++ = DFSState<Graph>(DFSCP_START_VERTEX, v);
}

void discover_vertex(vertex_t v, const Graph& g)
{
    *out_++ = DFSState<Graph>(DFSCP_DISCOVER_VERTEX, v);
}

void examine_edge(edge_t e, const Graph& g)
{
    *out_++ = DFSState<Graph>(DFSCP_EXAMINE_EDGE, e);
}

void tree_edge(edge_t e, const Graph& g)
{

41
```
Used in part 41a.

Finally, we define an acceptance test function, which will take as parameters a graph, a start vertex and a color map (the inputs to the depth first search algorithm) as well as a sequence of states, in the form of an iterator range. The acceptance test function stores the correct sequence of states, based on the BGL depth first search, and then compares that sequence to the iterator range.

\[
\text{(Define acceptance test function 42) } \equiv
\]

```cpp
template <typename VertexListGraph, typename ColorMap, typename InIt>
bool accept_dfs(const VertexListGraph& g, ColorMap cm, graph_traits<VertexListGraph>::vertex_descriptor start, InIt first, InIt last)
{
    typedef std::vector< DFSState<VertexListGraph> > state_vector;
    state_vector states;
    typedef std::back_insert_iterator<state_vector> state_inserter;
    saveStateVisitor<VertexListGraph, state_inserter> vis(back_inserter(states));
    depth_first_search(g, vis, cm, start);
    return equal(states.begin(), states.end(), first);
}
```

42
The DFSState class, the saveDFS function, the saveStateVisitor struct and the acceptance test function are combined in the test.hpp header file.

"test.hpp" 43a =

```cpp
#include <boost/graph/depth_first_search.hpp>
#include <vector>

// Define DFSState class 39a

// Define saveDFS function 40b

// Define saveStateVisitor 41a

// Define acceptance test function 42
```

4.1.2 Test Cases

The most obvious test is to generate a graph randomly and verify that it passes the acceptance test. test-random.cpp contains code that reads a number of vertices and an edge probability on the command line, generates a corresponding random graph, and tests the depth first search algorithm:

"test-random.cpp" 43b =

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <map>
#include "depth_first_search.hpp"
#include "test.hpp"

int main(int argc, char** argv)
{
  typedef boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_t;
```
typedef std::map<vertex_t, boost::default_color_type> ColorMap;
typedef boost::associative_property_map<ColorMap> Color;

srand((unsigned) time(NULL));

⟨Define color map 44a⟩
⟨Parse command line 44b⟩
⟨Create random graph 45a⟩
⟨Run DFS on graph 45b⟩
⟨Run acceptance check 45c⟩

return 0;
}

For a color map we use an STL map, wrapped in a Boost associative_property_map:

⟨Define color map 44a⟩ ≡

    ColorMap colorMap;
    Color color(colorMap);

Used in parts 43b, 45d, 47, 52b, 55a, 56b.

The command line must consist of an int, the number of vertices; and a double, the edge probability:

⟨Parse command line 44b⟩ ≡

    if (argc != 3)
    {
        std::cout << "Usage: "
                    << argv[0]
                    << " <vertices> <edge probability>\n";
        return 0;
    }

    const int V = atoi(argv[1]);
    const double P = atof(argv[2]);
The random graph contains $V$ vertices and an edge between each pair with probability $P$:

\[
\langle \text{Create random graph 45a} \rangle \equiv
\]
\[
\text{Graph } g(V);
\text{for (int } i = 0; i < V; ++i)
\text{for (int } j = 0; j < V; ++j)
\text{if (} i \neq j \&\& \text{rand()} / \text{RAND\_MAX} \leq P) \\
\text{add\_edge}(i, j, g);
\]

We use the saveDFS function to save the output of the depth first search algorithm:

\[
\langle \text{Run DFS on graph 45b} \rangle \equiv
\]
\[
\text{std::vector< DFSState<Graph> > states;}
\text{DepthFirstSearch<Graph, Color> dfs(g, color);}
\text{saveDFS(dfs, back\_inserter(states));}
\]

Finally, we run the acceptance test and output the result.

\[
\langle \text{Run acceptance check 45c} \rangle \equiv
\]
\[
\text{bool result = accept\_dfs(g, color, 0, states.begin(), states.end());}
\text{std::cout \ll "Result: " \ll (result ? "PASS" : "FAIL") \ll std::endl;
\]

Another useful test involves reading a graph from a file on disk (in the Graphviz format) and running a depth first search on that graph. `test-file.cpp` contains code for this test:

"test-file.cpp" 45d \equiv
#include <boost/graph/graphviz.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <cstdlib>
#include <iostream>
#include <map>
#include "depth_first_search.hpp"
#include "test.hpp"

int main(int argc, char** argv)
{
    typedef boost::GraphvizDigraph Graph;
    typedef graph_traits<Graph>::vertex_descriptor vertex_t;
    typedef std::map<vertex_t, boost::default_color_type> ColorMap;
    typedef boost::associative_property_map<ColorMap> Color;

    (Define color map 44a)

    (Read graphviz file 46)

    (Run DFS on graph 45b)

    (Run acceptance check 45c)

    return 0;
}

The graph is read from a file specified on the command line. If no file is
specified, usage information is displayed:

(Read graphviz file 46) ≡

if (argc == 1)
{
    std::cout << "Usage: " << argv[0] << " <file>\n";
    return 0;
}

Graph g;
boost::read_graphviz(argv[1], g);

Used in part 45d.
Using the **test-file** program, we verified that our algorithm object performs correctly for the following special cases:

- An empty graph (no vertices)
- A graph with one vertex
- A graph with ten vertices but no edges

### 4.2 Performance Testing

We tested the performance of our algorithm object on a Red Hat Linux 8.0 machine with eight 700 MHz Xeon processors and 32GB of RAM. We compiled with g++ 3.2 using -O3 optimization.

#### 4.2.1 Topological Sort

One common use of a depth first search traversal is to obtain a topological ordering of the vertices; that is, an ordering where for every edge $(i, j)$ the vertex $i$ precedes the vertex $j$. We use the timer class from [2] to compute the run time of topological sort algorithm based on our depth first search algorithm object. (See Appendix A for the source code for the timer class.) Since speed is critical, we use a **hash map** rather than a **map**.

"test-top.cpp" 47 ≡

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <ext/hash_map>
#include "depth_first_search.hpp"
#include "timer.hpp"

int main()
{
    typedef boost::adjacency_list<
        boost::vecS,
        boost::vecS,
        boost::directedS>
        Graph;
    typedef graph_traits<
        Graph>::vertex_descriptor
        vertex_t;
    typedef __gnu_cxx::hash_map<vertex_t,
        boost::default_color_type>
        ColorMap;
    typedef boost::associative_property_map<ColorMap>
        Color;
    srand((unsigned) time(NULL));
}````
Define color map 44a

```
timer tim;
```

Perform baseline test 48b

```
for (unsigned int size = 16; size <= 1024; size *= 2)
{
    (Create acyclic graph with size vertices 48a)

    (Perform topological sort test 48c)
}
```

return 0;
}

We ensure that the graph will have no cycles by adding an edge from $i$ to $j$ for all $i < j$ with probability one-half.

Create acyclic graph with size vertices 48a ≡

```
Graph g(size);
    for (unsigned int i = 0; i < size; ++i)
        for (unsigned int j = i + 1; j < size; ++j)
            if ((double) rand() / RAND_MAX < 0.5)
                add_edge(i, j, g);
```

Used in parts 47, 49.

Perform baseline test 48b ≡

```
tim.startBaseline(10);
    do {
        } while (tim.check());
tim.report(false);
```

Used in parts 47, 49, 52b, 53, 55a, 56b.

The actual test uses a vertex_iterator that stops at each DFSCP_DISCOVER_-VERTEX control point.

Perform topological sort test 48c ≡
```cpp
tim.start(10, size);
do {
    std::vector<vertex_t> v;
    v.reserve(num_vertices(g));
    DepthFirstSearch<Graph, Color> dfs(g, color);
    std::copy(dfs.vertex_it(DFSCP_DISCOVER_VERTEX),
             dfs.vertex_end(),
             back_inserter(v));
} while (tim.check());
tim.report(false);
```

Used in parts 47, 52b.

For comparison purposes, we time the BGL depth first search as well:

"test-top-bgl.cpp" 49 ≡

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/topological_sort.hpp>
#include "timer.hpp"

int main()
{
  using boost::graph_traits;
  typedef boost::adjacency_list<boost::vecS,
                                boost::vecS,
                                boost::directedS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_t;
  srand((unsigned) time(NULL));

timer tim;

  (Perform baseline test 48b)
  for (unsigned int size = 16; size <= 1024; size *= 2)
  {
    (Create acyclic graph with size vertices 48a)
    (Perform topological sort test using BGL 50)
  }
```

49
Table 1: Results of Topological Sort Test

<table>
<thead>
<tr>
<th>Size</th>
<th>BGL Visitor</th>
<th>Algorithm Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Growth Factor</td>
</tr>
<tr>
<td>16</td>
<td>0.0072</td>
<td>—</td>
</tr>
<tr>
<td>32</td>
<td>0.0196</td>
<td>2.7381</td>
</tr>
<tr>
<td>64</td>
<td>0.0685</td>
<td>3.4989</td>
</tr>
<tr>
<td>128</td>
<td>0.2505</td>
<td>3.6563</td>
</tr>
<tr>
<td>256</td>
<td>0.9571</td>
<td>3.8212</td>
</tr>
<tr>
<td>512</td>
<td>4.2366</td>
<td>4.4263</td>
</tr>
<tr>
<td>1024</td>
<td>28.5707</td>
<td>6.7438</td>
</tr>
</tbody>
</table>

Instead of using our algorithm object, we use the Boost topological_sort function:

```cpp
return 0;
}
```

The results of the topological sort test are summarized in Table 1. Figure 1 shows a log-log plot of the execution times for both methods. Our algorithm object is significantly slower than the BGL topological sort algorithm; however, since the primary rationale for an algorithm object is ease of programming, not execution speed, this is not unreasonable. What is important is that both our implementation and that of the BGL are $O(n^2)$.

4.2.2 Topological Sort — Sparse Graphs

A depth first search typically has the time bound $O(|V| + |E|)$, where $V$ and $E$ are the vertex and edge sets. In the previous test, we constructed
Figure 1: Results of Topological Sort Test
the graphs so that $|E| = \Theta(|V|^2)$, making the total time $O(|V|^2)$. It is also useful to consider sparse graphs — graphs where $|E| = \Theta(|V|)$. In this case the run time should be $O(|V|)$.

(Create sparse acyclic graph with size vertices 52a) ≡

```cpp
Graph g(size);
  for (unsigned int i = 0; i < size - 1; ++i)
    add_edge(i, rand() % (size - i - 1) + i + 1, g);
```

Used in parts 52b, 53.

Except for the creation of the graph, the test code is the same:

"test-top-sparse.cpp" 52b ≡

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <ext/hash_map>
#include "depth_first_search.hpp"
#include "timer.hpp"

int main()
{
  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
  typedef graph_traits<Graph>::vertex_descriptor vertex_t;
  typedef __gnu_cxx::hash_map<vertex_t, boost::default_color_type> ColorMap;
  typedef boost::associative_property_map<ColorMap> Color;
  srand((unsigned) time(NULL));

  (Define color map 44a)

  timer tim;

  (Perform baseline test 48b)

  for (unsigned int size = 16; size <= 1024; size *= 2)
    {
      (Create sparse acyclic graph with size vertices 52a)
      ```cpp
```
The results of the topological sort test for sparse graphs are summarized in Table 2. Figure 2 shows a log-log plot of the execution times for both methods. Again, our algorithm object is significantly slower than the BGL topological sort algorithm, but they share the same bound — in this case, $O(n)$. 

53
Table 2: Results of Topological Sort Test

<table>
<thead>
<tr>
<th>Size</th>
<th>BGL Visitor</th>
<th>Algorithm Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Growth Factor</td>
</tr>
<tr>
<td>16</td>
<td>0.0040</td>
<td>—</td>
</tr>
<tr>
<td>32</td>
<td>0.0076</td>
<td>1.9077</td>
</tr>
<tr>
<td>64</td>
<td>0.0153</td>
<td>2.0248</td>
</tr>
<tr>
<td>128</td>
<td>0.0295</td>
<td>1.9237</td>
</tr>
<tr>
<td>256</td>
<td>0.0586</td>
<td>1.9855</td>
</tr>
<tr>
<td>512</td>
<td>0.1160</td>
<td>1.9803</td>
</tr>
<tr>
<td>1024</td>
<td>0.2342</td>
<td>2.0194</td>
</tr>
</tbody>
</table>

Figure 2: Results of Topological Sort Test on Sparse Graphs
4.2.3 Cycle Detection

Another common use of a depth first search traversal is to check whether a graph has a cycle — a path whose start and end vertices are the same.

"test-cycle.cpp" 55a ≡

```cpp
#include <boost/graph/adjacency_list.hpp>
#include <ext/hash_map>
#include "depth_first_search.hpp"
#include "timer.hpp"

int main()
{
    typedef boost::adjacency_list<boost::vecS,
                                boost::vecS,
                                boost::directedS> Graph;
    typedef graph_traits<Graph>::vertex_descriptor vertex_t;
    typedef __gnu_cxx::hash_map<vertex_t,
                                 boost::default_color_type> ColorMap;
    typedef boost::associative_property_map<ColorMap> Color;
    srand((unsigned) time(NULL));

    timer tim;

    for (unsigned int size = 16; size <= 1024; size *= 2)
    {
        typedef boost::adjacency_list<boost::vecS,
                                       boost::vecS,
                                       boost::directedS> Graph;
        typedef graph_traits<Graph>::vertex_descriptor vertex_t;
        typedef __gnu_cxx::hash_map<vertex_t,
                                     boost::default_color_type> ColorMap;
        typedef boost::associative_property_map<ColorMap> Color;
        srand((unsigned) time(NULL));

        timer tim;

        for (unsigned int size = 16; size <= 1024; size *= 2)
        {
            typedef boost::adjacency_list<boost::vecS,
                                           boost::vecS,
                                           boost::directedS> Graph;
            typedef graph_traits<Graph>::vertex_descriptor vertex_t;
            typedef __gnu_cxx::hash_map<vertex_t,
                                         boost::default_color_type> ColorMap;
            typedef boost::associative_property_map<ColorMap> Color;
            srand((unsigned) time(NULL));

            return 0;
        }
    }
}
```

We allow for both cyclic and acyclic graphs by adding an edge from each i to each j with probability one-half.

"test-cycle.cpp" 55b ≡

(Define color map 44a)

(timer tim)

(Perform baseline test 48b)

for (unsigned int size = 16; size <= 1024; size *= 2)
{
    (Create graph with size vertices 55b)

    (Perform cycle-detection test 56a)
}

return 0;
}
Graph g(size);
for (unsigned int i = 0; i < size; ++i)
    for (unsigned int j = 0; j < size; ++j)
        if ((double) rand() / RAND_MAX < 0.5)
            add_edge(i, j, g);

The actual test uses an edge_iterator that stops at each DFSCP_BACK_EDGE control point.

(Perform cycle-detection test 56a) ≡

tim.start(10, size);
    do {
        DepthFirstSearch<Graph, Color> dfs(g, color);
        if (++dfs.edge_it(DFSCP_BACK_EDGE) != dfs.end())
            //cycle
    } while (tim.check());
tim.report(false);

For comparison purposes, we wish also to test cycle-detection with the BGL depth first search.

"test-cycle-bgl.cpp" 56b ≡

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/depth_first_search.hpp>
#include <ext/hash_map>

#include "timer.hpp"

using boost::graph_traits;

(Define cycle-detection visitor 57)

int main()
{
    typedef boost::adjacency_list<boost::vecS,
                                boost::vecS,
                                boost::directedS> Graph;
typedef graph_traits<Graph>::vertex_descriptor vertex_t;
typedef __gnu_cxx::hash_map<vertex_t,
    boost::default_color_type> ColorMap;
typedef boost::associative_property_map<ColorMap> Color;
srand((unsigned) time(NULL));

(Define color map 44a)
timer tim;

(Perform baseline test 48b)
for (unsigned int size = 16; size <= 1024; size *= 2)
{
    (Create graph with size vertices 55b)
    (Perform cycle-detection test using BGL 58)
}

return 0;
}

The BGL has no explicit cycle-detection algorithm, so we create a visitor for the depth_first_search algorithm that mimics our approach: run a depth first search until a back-edge is found or the search terminates. If a back-edge is found, we throw the string “cycle” as an exception to stop the algorithm.

(Define cycle-detection visitor 57)  

    template <typename Graph>
    struct cycleDetectionVisitor : public boost::dfs_visitor<>
    {
        void back_edge(graph_traits<Graph>::edge_descriptor e, const Graph& g)
        {
            throw "cycle";
        }
    };

Used in part 56b.

The use of an exception means that the call to depth_first_search must be enclosed in a try block.
Table 3: Results of Cycle-detection Test

<table>
<thead>
<tr>
<th>Size</th>
<th>BGL Visitor</th>
<th>Algorithm Object</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (ms)</td>
<td>Growth Factor</td>
</tr>
<tr>
<td>16</td>
<td>0.0527</td>
<td>—</td>
</tr>
<tr>
<td>32</td>
<td>0.0549</td>
<td>1.0417</td>
</tr>
<tr>
<td>64</td>
<td>0.0593</td>
<td>1.0801</td>
</tr>
<tr>
<td>128</td>
<td>0.0683</td>
<td>1.1512</td>
</tr>
<tr>
<td>256</td>
<td>0.0947</td>
<td>1.3867</td>
</tr>
<tr>
<td>512</td>
<td>0.1378</td>
<td>1.3867</td>
</tr>
<tr>
<td>1024</td>
<td>0.1922</td>
<td>1.3941</td>
</tr>
</tbody>
</table>

(Perform cycle-detection test using BGL 58) \equiv

```c++
    tim.start(10, size);
    do {
        try {
            depth_first_search(g,
                cycleDetectionVisitor<Graph>(),
                color,
                *vertices(g).first);
        } catch (const char*) {
        }
    } while (tim.check());
    tim.report(false);
```

Used in part 56b.

The results of the cycle-detection test are summarized in Table 3. Figure 3 shows a log-log plot of the execution times for both methods. Our algorithm object significantly outperforms the BGL algorithm; one possible reason is the overhead associated with throwing an exception and/or cleaning up the stack, which is not incurred using our algorithm object. The plot suggests that the BGL approach incurs some constant-time penalty, whose impact decreases as the size of the graph increases.

5 Conclusion

We have presented here an implementation of the depth first search algorithm as an algorithm object. Our algorithm object is fully compatible with the Boost Graph Library while offering substantial advantages in ease of
Figure 3: Results of Cycle-detection Test
programming. In addition, performance testing shows that, while our implementation is generally slower than that of the BGL, the \(O(|V| + |E|)\) bound is preserved. For some specialized uses, our algorithm object may even be faster than the BGL implementation.

A  Source Code for Timer Class

We use the timer class from \([2]\), with minor modifications to avoid including the \texttt{std} namespace:

"timer.hpp" 60  

```cpp
// Define a timer class for analyzing algorithm performance.
#include <iostream>
#include <iomanip>
#include <vector>
#include <map>
#include <algorithm>
using std::cout; using std::endl;
using std::setprecision; using std::setw;
using std::map; using std::vector;

class timer {
  public:
    timer(); // Default constructor
    // Start a series of r trials for problem size N:
    void start(unsigned int r, unsigned long N);
    // Start a series of r trials to determine baseline time:
    void start_baseline(unsigned int r);
    // Returns true if the trials have been completed, else false
    bool check();
    // Report the results of the trials on cout
    // with additional output if verbose is true:
    void report(bool verbose);
    // Returns the results for external use
    const map<unsigned int, double>& results() const;
  private:
    unsigned int reps; // Number of trials
    // For storing loop iterations of a trial
    vector<long> iterations;
    // For saving initial and final times of a trial
```

60
time_t initial, final;
// For counting loop iterations of a trial
unsigned long count;
// For saving the problem size (N) for current trials
unsigned int problem_size;
// For storing (problem size, time) pairs
map<unsigned int, double> result_map;
// true if this is a baseline computation, false otherwise
bool baseline;
// For recording the baseline time
double baseline_time;
};
timer::timer() { baseline = false; }
void timer::start(unsigned int r, unsigned long N)
{
    reps = r;
    problem_size = N;
    count = 0;
    iterations.clear();
    iterations.reserve(reps);
    initial = time(0);
}
void timer::start_baseline(unsigned int r)
{
    baseline = true;
    start(r, 0);
}
bool timer::check()
{
    ++count;
    final = time(0);
    if (initial < final) {
        iterations.push_back(count);
        initial = final;
        count = 0;
    }
    return (iterations.size() < reps);
}
void timer::report(bool verbose)
{
    if (verbose) {
        for (unsigned int k = 0; k < iterations.size(); ++k) {
            cout << iterations[k] << " ";
            if ((k+1) % 10 == 0)
cout << endl;
}
cout << endl;
}

sort(iterations.begin(), iterations.end());
if (verbose) {
  cout << "Sorted counts:" << endl;
  for (unsigned int k = 0; k < iterations.size(); ++k) {
    cout << iterations[k] << " ";
    if ((k+1) % 10 == 0)
      cout << endl;
  }
  cout << endl;
}
cout << endl;

int selected_count = iterations[reps/2];
if (verbose)
  cout << "Selected count: " << selected_count << endl;
if (baseline) {
  baseline_time = 1000.0/selected_count;
  cout << "Baseline time: " << baseline_time << endl;
  baseline = false;
} else {
  double calculated_time, growth_factor;
  result_map[problem_size] = calculated_time = 1000.0/selected_count - baseline_time;
  cout << setiosflags(std::ios::fixed) << setprecision(4)
       << setw(35) << problem_size << setw(12)
       << calculated_time << " ms ";
  if (result_map.find(problem_size/2) != result_map.end()) {
    growth_factor = calculated_time / result_map[problem_size/2];
    cout << setiosflags(std::ios::fixed) << setprecision(4)
         << setw(8) << growth_factor;
  }
  cout << endl;
}

const map<unsigned int, double>& timer::results() const {
  return result_map;
}
References

