Revisiting the Design of a

Generic Graph Library

Leandro T. Melo and Renato C. Mesquita*†

Departamento de Engenharia Elétrica, Universidade Federal de Minas Gerais
Antonio Carlos 6627 - 31270-010, Belo Horizonte, MG, Brasil.

SUMMARY

The most significant work that combines the paradigm of generic programming with graph related problems is the Boost Graph Library (BGL). Besides providing efficient and flexible graph representations and algorithms implementations, the BGL also introduces an external adaptation system that allows its integration with proprietary data structures. In this paper, we present a new generic graph library, called Graph Toolkit for Algorithms and Drawings (GTAD), that is constructed with the same fundamental principles of the BGL. However, many design decisions were taken differently. In the GTAD, algorithms and policies are formalized as concepts. Many BGL concepts are refactored for better cohesion. Implementation details regarding flexibility and ease-of-use of algorithms and data structures in the GTAD, along with performance comparisons against the BGL, are also provided.

*Correspondence to: Departamento de Engenharia Elétrica, Universidade Federal de Minas Gerais
Av. Antonio Carlos 6627 - 31270-010, Belo Horizonte, MG, Brasil.
†E-mail: renato@ufmg.br
INTRODUCTION

Generic programming, or programming with *concepts*, is an increasingly used paradigm for the development of software libraries. A very significant work in this area is the Standard Template Library (STL) [14], which provides implementations for several fundamental algorithms and data structures. Following the design of the STL, the well known Boost Graph Library (BGL) [18] was constructed for solving graph related problems.

Genericity in the BGL is achieved by an *external adaptation* system that allows the algorithms to operate over proprietary graph representations. This process consists of the specialization of a traits [1] class and the overload of global functions. Furthermore, flexibility of the algorithms is also accomplished by the parametrization of visitors (a variation of the visitor design pattern [13]) and property maps that hold all information required.

In this paper, we present a new generic graph library through a comparison against the BGL. Our library, called Graph Toolkit for Algorithms and Drawings (GTAD) [21], is based on ideas similar to the ones existent in the BGL. Interesting construction principles were kept, but we have made a number of improvements regarding flexibility and ease-of-use. Therefore, this paper is not focused on presenting the GTAD itself, but on these improvements.

Initially, we argue that implementing algorithms as functions, like in the STL and in the BGL, is not the best approach in the context of graphs. Implementing them as classes
and formalizing them as concepts is more suitable for such rich combinatorial structures \[4\] and fit better in the generic programming paradigm. Additionally, in order to bring down many abstractions of algorithms descriptions to implementations, policies, also formalized as concepts, are employed.

Still regarding algorithms, we mention how simple implementation details can contribute for the ease-of-use of the library. Basically, we have removed template parameters of BGL’s algorithms with no semantic loss. We present an interesting parametrization, which removes the necessity of visitors for storing data generated by the execution of the algorithm. Finally, we make observations regarding some names of functions and concepts in the the BGL and suggest a few changes.

A very important part of a generic library is its concepts. They should be carefully modeled and provide only the necessary and sufficient abstractions for implementations. Therefore, we compare particular GTAD’s concepts against BGL’s concepts and show their advantages. We also point out cases where BGL’s concepts are inappropriate and may lead to undesired behavior.

Even offering a mechanism for the use of proprietary graph representations, it is important for a generic graph library to provide one or more graph implementations. The BGL provides two highly parameterized graph classes: \texttt{adjacency\_list} and \texttt{adjacency\_matrix}. In most situations, the former is preferable to the latter (in real applications graphs are usually sparse \[20\]). Since many users will not implement their own graph representation, we show in details how the adjacency list of the GTAD is implemented.
Throughout this paper we expect familiarity with C++, both STL and BGL, generic programming techniques, and graph nomenclature.

RELATED WORK

The C++ STL [14, 9] is a well known work in the area of generic programming. One of the most important ideas behind this library is to provide specifications with minimal assumptions about data representation, lifting away issues related to concrete implementations, and thus achieving more flexibility. This is done by creating concepts which combine all requirements that a particular implementation must satisfy in order to be functional.

There are two important sets in the STL: the containers and the algorithms. Since genericity is a primary concern, containers are not directly attached to algorithms or vice-versa. They are connected by special entities called iterators that, in conjunction with functors, can be used to access data.

Graphs are not included as part of the STL, which is a library for fundamental algorithms and data structures. For this mean, the BGL [18] was created. In the same way that STL containers and STL algorithms are mapped through iterators and functors, visitors and graph iterators are used to map graph algorithms and graph data structures in the BGL. Furthermore, property maps that can be attached to graph components provide a data parametrization that is equivalent to the one in the STL containers.

The BGL introduces several graph concepts and algorithms implementations. Proprietary data structures and even graph representations from other libraries can be used in BGL algorithms as long as they follow the required concept. This mechanism which consists of the
overload of global functions and specialization of a traits class is called external adaptation. The result is a transparent way to work with a variety of graph types through well defined interfaces.

Like the STL and the BGL there are other generic programmed libraries that fit a particular domain. One example is the Matrix Template Library (MTL) [24], that provides linear algebra functionality and many matrix formats. Another extensively used library for computational geometry that is based on generic programming is the Computational Geometry Algorithms Library (CGAL) [25].

ALGORITHMS

In the GTAD algorithms are implemented as classes. We believe this is more flexible and maintainable than implementing them as functions, like in the BGL and STL. Particularly in the case of the STL, we understand such choice, since algorithms are very fundamental and they basically operate over container of elements. Functions like std::find, std::sort, std::reverse, and std::fill have well defined input and output. However, in the context of graphs, this is not always true. Graphs are very rich combinatorial structures that can be associated with very complicated algorithms. Consider, for example, BGL’s named parameter version of function biconnected_components. The goal of this function is to return the number of biconnected components in the graph. There is one overload that adds to the return an Output Iterator for the articulation points of the graph. In addition, there is an associated function that only computes the articulation points of the graph. What if one would also like to retrieve the bridges of the graph? Probably, more overloads or functions refactoring would
be necessary. On the other hand, if the biconnectivity algorithm was implemented as a class, the simple addition of a new member function would be sufficient.

The purpose of the GTAD is not to serve as a graph library only but as a framework for a large number of algorithms that can interact with each other through well defined interfaces. Besides formalizing concepts for graphs categories, the GTAD goes one step further and formalize concepts for graphs algorithms. Doing that also makes it easier to establish policies and document how they integrate with algorithms.

Concepts form a complete set of requirements that data types must support. They usually specify behavior, constructors, operators and even associated types. Therefore, implementation of algorithms as classes is a natural choice in this context. Furthermore, it allows for algorithm objects to be self-contained in terms of all information they need, which avoids scattered data being passed around through functions. Pre-processing can also be done only once with this approach - when successive executions are needed.

Algorithm Concepts

All algorithm concepts in the GTAD refine the Graph Algorithm concept. This concept declares an associated type graph_type and a valid expression a.execute (g), where a is an object of a type that models Graph Algorithm and g is an object of type graph_type. These two items are essential to every algorithm implementation. Thus, this concept is on the top of the hierarchy.

Direct refinements of Graph Algorithm include, among others, Search Algorithm, Weighted Graph Algorithm, and Flow Algorithm. Naturally, we do not mention details of all algorithm
concepts, even though some of them are described in appendix I. Instead, we use the classical shortest-path problem as an example.

The shortest-path problem consists of finding a minimum weight path, which is the sum of the weights of the path’s edges, between two given vertices. An usual variation is the single-source shortest-path problem, where paths are found from a particular source vertex to all other vertices in the graph. Algorithms for solving the single-source shortest-path problem are incorporate by the GTAD through the Single Source Shortest Path Algorithm concept.

The Single Source Shortest Path Algorithm is a refinement of Weighted Graph Algorithm. The latter declares the associated type weight_map_type and the valid expressions that manipulate the weight map, the infinity weight value, and the zero weight value. The former does not declare any type, but declares valid expressions to manipulate the source vertex, to obtain the shortest-paths and distances.

Well known solutions for this problem are Dijkstra’s[5] and Bellman-Ford’s[3, 12] algorithms. Suppose, then, an application is using an implementation of Dijkstra’s in a scenario where weights are non-negative. However, a change in requirements now allows the presence of negative weights, and, consequently, Dijkstra’s is no longer a suitable solution. A possible option is to refactor the code so that Bellman-Ford’s is used. This change can be problematic or just boring if both implementations do not follow the exactly same interface, something that happens in Dijkstra’s and Bellman-Ford’s implementations of the BGL. But if concepts are used, things are a lot easier, as the code below shows.

```cpp
typedef adjacency_list<multigraph, directed> Graph; Graph g;
//Populate graph...

//Define Dijkstra’s or Bellman-Ford’s.
```
//typedef dijkstra_shortest_path<Graph> ShortestPath;
typedef bellman_ford_shortest_path<Graph> ShortestPath;

typedef ShortestPath::vertex_index_map_type vim_type; typedef
ShortestPath::weight_map_type wm_type;

vim_type vim;
//Populate vertex index map...

wm_type wm;
//Populate weight map...

ShortestPath sp; sp.set_vertex_index_map(vim);
sp.set_weight_map(wm); sp.set_source(v); sp.execute(g); 
std::cout << "Distance from v to w: " << sp.distance(w);

There is still the concern that the type of the graph object under which the algorithm
is being executed must model the concepts expected by both implementations of Dijkstra
and Bellman-Ford. In many situations this will not be a problem, since graph representations
like GTAD’s and BGL’s adjacency_list are very complete in terms of the concepts they
model and shortest-path implementations will probably expect similar or even the same graph
concepts.

A question that may arise is which concept declares the type vertex_index_map_type. It
comes from Vertex Indexed Construct, which is a general purpose concept for any kind of
construct that requires a vertex index map. This concept also declares valid expressions to
manipulate the vertex index map, like the one used in the code above.

The algorithm concepts do not refine Vertex Indexed Construct, but dijkstra
_shortest_path and bellman_ford_shortest_path model it. That is why the code above is
valid. Although we believe that most generic implementations of such algorithms will require
some kind of vertex index map, we have opted for this design because we did not want to impose this restriction - just like the *Vertex Indexed Construct*, there is a similar concept, called *Edge Indexed Construct*, for situations where an edge index map is required.

Although this is a simple example, it should be clear that the same benefits exposed by STL container concepts are present in GTAD algorithm concepts. Basically, any legacy algorithm implementation can be wrapped around a well defined interface and used in a transparent way with the GTAD and with other software applications that adopt GTAD’s concepts. Every procedure can now be built entirely based on graph and algorithm specifications, lifting away all related type details.

**Policies**

It is relatively common for graph algorithms to be described with a high level of abstraction. The *augmenting-path method* by Ford-Fulkerson [12] for finding a maximum flow in a network is an elucidating example. We quote Sedgewick’s [16] summarization of it:

"Start with zero flow everywhere. Increase the flow along any path from source to sink with no full forward edges or empty back edges, continuing until there are no such paths in the network."

Although BGL’s implementation of the above method and many others adhere to the strategy of Edmonds and Karp [2], which uses the shortest paths as the augmenting paths, there is not a specific rule about how such paths are chosen. For instance, the user might choose an implementation that uses the longest paths as the augmenting paths or even an implementation that takes other information into consideration. In the GTAD we try to generalize abstractions
like this through the use of policies [7], a technique with the same rationale of the Strategy design pattern [13]. For this particular situation, class `augmenting_path_maxflow` of the GTAD has a template template parameter called `find_augmenting_path_t`, responsible for finding the paths. This parameter must model the `Find Augmenting Path Policy` concept.

```cpp
template <
  class graph_t,
  class vertex_index_map_t = ...
  class flow_unit_map_t = ...
  class reverse_edge_map_t = ...
  template <class> class find_augmenting_path_t = ...
  class policy_data_t = ...>
  class augmenting_path_maxflow {
    //...
  }
```

In the GTAD, policies concepts are not refinements of any other concept. Nevertheless, they have the same basic structure: two associated types, `graph_type` and `algorithm_type`, and a static "execute" valid expression that returns `void` and accepts as parameters the graph and algorithm related data is declared. The "return" is done through one or more parameters passed to the policy. Therefore, these particular parameters should be passed by reference.

Policies need to manipulate the graph and the algorithm data. Hence, they should specify which graph and algorithm concepts they expect. Since the graph interface required by a particular policy is implementation dependant, they usually do not impose significant restrictions on specific graph concepts. However, the algorithm data types must be defined. This is done by requiring that the algorithm type models the desired algorithm concepts. For instance, `Find Augmenting Path Policy` requires that the maximum flow implementation for which it integrates is a model of `Lower Bounded Flow Algorithm`, `Vertex Indexed Construct`, and `Policy Based Construct`. Just like `Vertex Indexed Construct` and `Edge Indexed Construct`,
Policy Based Construct is a general purpose concept that can be used by any construct that is associated with user defined data. The Find Augmenting Path Policy concept is described in appendix II. An implementation that is a model of it and finds the shortest augmenting-paths might look the code below.

```cpp
template <class algorithm_t> class find_shortest_augmenting_path_via_bfs { public:
  typedef algorithm_t algorithm_type;
  typedef typename algorithm_t::graph_type graph_type;

private:
  find_shortest_augmenting_path_via_bfs();

  typedef typename algorithm_t::graph_type graph_t;
  typedef typename algorithm_t::vertex_index_map_type vertex_index_map_t;
  typedef typename algorithm_t::flow_unit_map_type flow_unit_map_t;
  typedef typename algorithm_t::reverse_edge_map_type reverse_edge_map_t;
  typedef typename algorithm_t::policy_data_type policy_data_t;

  typedef typename graph_traits<graph_t>::vertex_descriptor vertex_descriptor;
  typedef typename graph_traits<graph_t>::edge_descriptor edge_descriptor;
  typedef typename graph_traits<graph_t>::out_edge_iterator out_edge_iterator;

public:
  static void execute(
    algorithm_t& algorithm,
    graph_t& g,
    vertex_descriptor s,
    vertex_descriptor t,
    vertex_index_map_t& vertex_index_map,
    flow_unit_map_t& capacity_map,
    reverse_edge_map_t& reverse_edge_map,
    flow_unit_map_t& lower_bound_map,
    flow_unit_map_t& residual_capacity_map,
```
The reader should note that method `execute` of `find_shortest_augmenting_path_via_bfs` accepts many parameters. Those correspond to all information that can be set by the user through any of the algorithm concepts for which this policy integrates - in most situations only a few of them will be used, but from a generic perspective a policy writer should have them available. In this case they are `Lower Bounded Flow Algorithm`, `Vertex Indexed Construct`, and `Policy Based Construct`. The last parameter, `augmenting_path`, is the one that will contain the augmenting path found by the policy.

The need for policies is very common in graph algorithms. A situation similar to the one we have described is found on implementations of Goldberg-Tarjan’s `preflow-push method` [10] for finding a maximum flow in a network. Once more, a common way to initialize the `height` (or `distance`) labels is through the shortest paths of the residual network. However, different strategies may also be suitable for this task.

Even a more interesting scenario appears when it comes to advanced graph algorithms which are built on top of others. For such cases, there actually might be a complete framework of algorithms that interact with each other and highly depend on well defined interfaces. For instance, consider the planarization by vertex addition algorithm described in [11]. There are three crucial steps in the algorithm: computation of a maximal planar subgraph, computation of a planar embedding, and how the non-planar edges will be inserted into the graph. There are
several different strategies for each one of these steps. How should one deal with such orthogonal aspects in a generic way? A very flexible possibility is a policy based implementation of the algorithm that might look like this:

```cpp
template <
    class graph_t,
    class vertex_index_map_t = ...,
    class edge_index_map_t = ...,
    template <class> class compute_max_planar_subgraph_t = ...,
    template <class> class compute_planar_embedding_t = ...,
    template <class> class insert_non_planar_edges_t = ...,
    class policy_data_t = ...>
class vertex_addition_planarization {
    //...
};
```

In the above code, all three template template parameters can be specified in terms of policy concepts.

**Implementation Issues**

Following the example of the BGL, algorithms of our library are based on the same external adaptation system. The idea of property maps and visitors is also present in the GTAD. Thus, we do not talk about these items. We do talk, however, about other types of template parameters that are available in the algorithms. One of them is responsible to determine which elements should be stored during the algorithm execution. A good example is GTAD’s BFS (breadth-first search) implementation. Usual elements to be stored in a BFS search are: the list of vertices in the order they were found; the tree edges (which are either tree
links or parent links) \(^1\); the cross edges (which are always cross links) \(^1\); the shortest path from the start vertex and other vertices; and the distance values from the start vertex and other vertices. Therefore, one of the template parameters of our BFS implementation is called \texttt{search\_storable\_items\_t}. With this parameter, users are able to configure the \texttt{bfs} class behavior in accordance to their needs.

\begin{verbatim}
template<
  class graph_t,
  class vertex_index_map_t = ...,
  class bfs_visitor_t = ...,
  class search_storable_items_t = ...,
  class visibility_adaptor_t = ...
>
class bfs
  //...
\end{verbatim}

As an argument to the \texttt{search\_storable\_items\_t} template parameter, users must pass a valid definition of class \texttt{search\_storable\_items}, which was developed with a named template parameter technique \([8]\). Definitions like the following ones are all valid.

\begin{verbatim}
search_storable_items<tree_edges, cross_edges>;
search_storable_items<cross_edges, tree_edges>;
search_storable_items<cross_edges, short_path, distance>;
search_storable_items<search_vertices, tree_edges, cross_edges, short_path, distance>;
search_storable_items<distance>;
search_storable_items<>
  //Others...
\end{verbatim}

Customization of the information that should be stored in BGL algorithms is done through visitors. For instance, the predefined visitor \texttt{distance\_recorder} can be used to store the distances in the \texttt{breadth\_first\_search} function. A primary advantage of our approach is that the user does not need to write code that performs relatively usual choices for particular

\(^1\)We follow the nomenclature of Sedgewick \([16]\)
algorithms, it is just a matter of supplying the desired template argument. In order to achieve the same flexibility of our approach, a large number of different predefined visitors may need to be available in the BGL.

The last template parameter of class \texttt{bfs} is a visibility adaptor. Its goal is to allow the user to filter specific edges of the graph during the algorithm execution. (Filtering vertices directly is not permitted in order to avoid inconsistent situations like edges that are incident to a ”non existent” vertex. Nevertheless, it is still possible to filter all incident edges of a vertex and thus make it invisible.) This is similar in functionality to BGL's \texttt{filtered_graph}. However, although we think that BGL's adaptor is useful and necessary, we believe that the existence of template parameter \texttt{visibility_adaptor_t} is also important because if a \textit{Mutable Graph} is expected by a particular algorithm, a \texttt{filtered_graph} cannot be used. Naturally, with our approach the user still must be careful to implement the \texttt{is_edge_visible} function, and handle properly edges that might have been added or removed from the graph.

The \texttt{breadth_first_search} function of the BGL is parameterized by two other items that are not available in GTAD's \texttt{bfs} class. One of them is the \texttt{buffer} used to determine the discover order of the vertices. Depending on the kind of buffer supplied, a different search is performed. For instance, if a LIFO queue is passed as an argument, the result is actually a DFS (depth-first search). Since this behavior might be a little awkward, we prefer to provide a corresponding functionality through a \texttt{generalized_search} class, which then serves as the core implementation for others algorithms, including the BFS. The second item is the \texttt{color_map}. The GTAD does not offer an equivalent parametrization for this particular case because visitors and visibility adaptors can be used in conjunction to provide a similar functionality.
In general, we have removed from BGL’s algorithms template parameters that we considered to be not very important. Specially because the GTAD already introduces others for policies, storable items and visibility adaptors. Basically, the ones that are related to input information, property maps, and visitors are maintained, but with a couple of exceptions. Consider, for example, BGL’s `dijkstra_shortest_paths` function. Among several template parameters, there is one for the `weight_map`, one for `distance_inf`, and one `distance_zero`, which correspond, respectively, to the data structure that maps edges to their weights, a value that is the greatest possible weight (infinity), and a value that is the zero weight. In this case, only the first template parameter is necessary, because the types of the infinity and zero values must be the same as the `value_type` of the weight map type, and thus can be inferred from it. In GTAD implementation of Dijkstra’s algorithm, there are two member functions to set these values: `set_zero_weight` and `set_infinity_weight`. No template parameters are needed for them.

Another situation where the number of template parameters can be reduced is found in BGL’s flow algorithms. Both the `capacity_map` and the `residual_capacity_map` are data structures that map edges to a type that represents the flow unit. It is reasonable to assume that the value type of these maps are the same or, at least, convertible one to another. Under this strategy, if BGL’s flow algorithms accepted lower bounds for the edges of the graph, an additional template parameter, probably named something like `lower_bound_map`, would be required. In the GTAD, only one template parameter named `flow_unit_map` exists. This parameter corresponds to the type of the capacity map, the residual capacity map, and the lower bound map.
GRAPH CONCEPTS

Concepts play a fundamental role in a generic library. Particularly in the case of a generic graph library, it is expected that a concept with the minimum requirements of a graph exists. Just like the BGL, the GTAD defines the Graph concept on top of the hierarchy of all graph concepts. This concept should be the least restrictive as possible. In the GTAD, it only declares a vertex handle and an edge type, respectively named vertex_descriptor and edge_descriptor in order to guarantee easy interoperability with the BGL. However, we believe that BGL’s Graph concept imposes inappropriate constraints for types that model it, as discussed in the following paragraphs.

In a way very similar to the iterator_category of the STL, the BGL’s Graph concept declares three category types: directed_category, edge_parallel_category, and traversal_category. These types are used in static dispatching mechanisms. The problem is that there are no guarantees that they are not associated with dynamic characteristics of graph representations from other libraries. Basically, it is possible that whether the graph is directed or undirected, or whether it accepts parallel edges or not, is an information bound at run-time.

Consider, for example, the GRAPH type of LEDA (Library of Efficient Data Types and Algorithms) [22], which is used as an example of how to convert existent graphs to the BGL [23]. Although the GRAPH type represents, by default, a directed a graph, it can be made undirected through a call to its member function make_undirected. Conversely, a call to make_directed has the opposite effect. Therefore, what would be the appropriate tag for the directed_category in the graph_traits specialization for GRAPH: directed_tag or
**undirected_tag**? Depending on the run-time characteristic of a **GRAPH** object, this tag might become inconsistent.

A similar situation that might arise is when **disallow_parallel_edge_tag** is used for the **edge_parallel_category**. Parallel edges are not prevented in the **GRAPH** type. However, after a call to LEDA’s function **Make_Simple** or in a graph generated by **random_graph** with **no_parallel_edges** set to true, there will be no parallel edges in the graph. A function that uses a dispatching mechanism based on this tag could waste a lot of time looking for parallel edges in such a graph.

The GTAD is highly compatible with the BGL. Associated types and valid expressions of graph concepts follow the ones in the BGL (actually, the associated types are declared under GTAD’s namespace, but this is transparent to the user). There is no need for template specialization when using GTAD’s **adjacency_list** with BGL algorithms or BGL’s **adjacency_list** with GTAD’s algorithms. However, we do categorize our graph concepts differently. There are not too many differences, but they exist. One of them was already mentioned in previous paragraphs.

In the GTAD, we have weakened **Vertex List Graph** and **Edge List Graph** concepts of the BGL by decomposing each one in a two layer hierarchy. As direct refinements of **Graph**, there are the **Countable Vertex Set Graph** and **Countable Edge Set Graph** concepts. The former declares an associated type **vertices_size_type** and a valid expression **num_vertices(g)**, where **g** is an object of a type that models **Countable Vertex Set Graph**. Similarly, the latter declares an associated type **edges_size_type** and a valid expression **num_edges(g)**. Then, two other concepts, **Traversable Vertex Set Graph** and **Traversable Edge Set Graph**, refine
the above mentioned countable concepts and declare associated types vertex_iterator and edge_iterator, and valid expressions vertices(g) and edges(g). This is different from the BGL where Vertex List Graph and Edge List Graph declare all the associated types and valid expressions of the equivalent four concepts of the GTAD. This decomposition was made because obtaining the number of vertices and edges of a graph is simple and easy to implement in most graph representations, but traversing the vertices and edges might not be as obvious. Since we want concepts to be a minimum requirement interface and there are algorithms implementations that might need the number of vertices or edges, but not to traverse all of them (maybe they only need incidence of adjacency information), this change seems to be worth it.

Two common graph representations are the adjacency list and the adjacency matrix. As we have mentioned before, adjacency lists are usually preferable to adjacency matrices. They have a significant drawback though: testing the existence of an edge composed by two specific vertices is more difficult to implement efficiently. In the adjacency matrix, this operation is straightforward to implement in constant time. Probably because of that the BGL defines a concept Adjacency Matrix which declares the valid expression edge(u, v, g) responsible for finding a particular edge composed by vertices u and v in the graph g. This concept also imposes a constant time complexity guarantee for this expression.

Although efficient find-edge operations are easier to implement with an adjacency matrix, it is still possible to accomplish it with an advanced and well designed adjacency list. As presented in section 5, GTAD’s adjacency_list provides an amortized constant time find-edge implementation. Nevertheless, we think that imposing a constant time complexity guarantee
is too strong for a concept like Adjacency Matrix. In fact, even BGL’s adjacency_list does not provide an implementation that meets this requirement. We also believe that since this operation is not meant to be specific to adjacency matrices, a more appropriate name for such concept would be Edge Findable Graph, which is the chosen name in the GTAD. Different from BGL’s Adjacency Matrix concept, Edge Findable Graph only imposes a complexity guarantee of $O(E/V)$ for the find-edge operation, where $E$ is the total number of edges and $V$ is the total number of vertices in the graph.

A last observation about BGL’s graph concepts regards the Graph concept. Once more, in order to achieve the minimal requirement interface goal and separation of concerns, we have removed valid expression null_vertex() from Graph. In the GTAD, there are two concepts, Nullable Vertex Graph and Nullable Edge Graph, that declare, respectively, valid expressions null_vertex() and null_edge(). Some of the GTAD graph concepts discussed in this section are described in appendix III.

**VISITOR CONCEPTS**

This is also an appropriate moment for a brief review about BGL’s visitor concepts. Among all valid expressions of a visitor, there are several of them that take an edge descriptor as a parameter. One example is the tree_edge(e, g) expression, where e is an edge descriptor and g is a graph. The dijkstra_shortest_paths function, as many others of the BGL, is implemented in terms of a BFS visitor. The edge relaxation step of Dijkstra’s algorithm is performed on callback functions. Particularly in this case, there is an interesting use of item directed_category, mentioned in the last section, to determine if the graph is directed.
or undirected. The problem is that the relaxation condition of a particular edge \( e \) must be performed in terms of the vertex \( v \) currently being processed by the algorithm and the vertex \( w \), which is the other end of \( e \). Since in undirected graphs, an edge \((s, t)\) is the same as an edge \((t, s)\), only through an edge descriptor, passed by the visitor’s callback function, it is not possible to determine if calls to \texttt{source} and \texttt{target} will actually return vertices \( v \) and \( w \), respectively. The solution is then to verify the \texttt{directed\_category} of the graph and if it is \texttt{undirected\_tag}, a double check must be made in the edge relaxation condition.

This kind of problem is avoided in the GTAD because the expressions of our visitors concepts not only pass the edge descriptor as an argument, but they also pass the vertex descriptor which was used as the ”origin” to retrieve a particular edge. With this approach there is no need to identify, through calls to \texttt{source} and \texttt{target}, the vertex currently being processed by the algorithm.

In a search tree there are two representations of each graph edge. Hence, the GTAD visitors concepts follow the nomenclature of Sedgewick [16] and declare valid expressions that are associated to edge traversal events with the term \textit{link} instead of edge. For example, BGL’s \texttt{tree\_edge(e, g)} is declared in GTAD’s visitors as \texttt{tree\_link(e, v, g)} - also note the extra parameter for the vertex descriptor.

Another significant change in GTAD’s search visitors concepts is how they are categorized. There is a base \textit{Visitor} concept, which is refined by all other visitors, including a \textit{Search Visitor}. Unlike the BGL, there are not visitors such as \textit{BFS Visitor} or \textit{DFS Visitor}. Instead, there are two refinements of \textit{Search Visitor} that are \textit{Recursive Search Visitor} and \textit{Non Recursive Search Visitor}. The reason for that is because the events of a search algorithm are better identified
by the kind of queue used in the implementation: either the built-in LIFO queue in a recursive implementation or any other kind of queue in a non-recursive implementation. For instance, `generalized_search` class of the GTAD expects a visitor type that is a model of `Non Recursive Search`. If a LIFO queue is passed to this class, a DFS is resulted. If a FIFO queue is passed, a BFS is resulted. If priority-queue is passed, a PFS (priority-first search) is resulted. On the other hand, in a recursive implementation of a DFS, like GTAD’s `dfs` class, a `Recursive Search Visitor` is more appropriate because it provides, for example, an extra event handler for a tree link after the recursive call, what is not possible to do in a non-recursive implementation.

THE GRAPH REPRESENTATION

A primary step towards the development of efficient graph algorithms implementations is to provide efficient implementations for operations that manipulate and access the graph data structure. Although both the GTAD and the BGL are generic graph libraries, it is important that they provide efficient graph implementations. After all, not all users will write their own graph representation.

Adjacency lists usually presented in books \cite{16, 17} suffer from several disadvantages. For instance, operations for finding a particular edge and removing a vertex can be expensive. Even though the adjacency list of the BGL is an improvement over traditional adjacency lists, it still has some of these drawbacks.

In our adjacency list, we have tried to combine several benefits of traditional adjacency lists, adjacency matrices and incidence matrices. Like in the BGL, there are two separate containers
for the vertices and edges of the graph ‡, but all insertion, search and removal is done through two separate auxiliary dynamic symbol tables. All adjacency information and the mapping between the symbol table data structures and their associated containers is preserved by a large number of cross-pointers. For efficient navigability, cross-pointers are also used to keep track of both representations of an edge (one for each of its incident vertices). A simplified illustration of our design is shown in Fig. 1.

This graph representation not only supports an amortized constant time operation for finding a particular edge, but makes it possible to implement efficient operations that directly manipulate the internal structure of the adjacency list. For instance, operations for circulating the adjacency list of a vertex in both directions, rotating the adjacency list of a vertex, inserting

‡ Actually, we store the vertices and edges in what we call a Vertex box and an Edge box, which are then stored in the containers. These boxes hold additional information that is useful for some operations.
an edge into a specific position of the adjacency list of a vertex, switching positions of two edges in the adjacency list of vertex, and others, may be very important for planarity tests [6][19] and graph drawing related algorithms.

Customization

One of the first steps before creating a graph is deciding whether or not it should accept self-loops and parallel edges. Self-loops cannot be prevented automatically in BGL graph classes, this control is left to the user. On the other hand, in BGL’s adjacency list parallel edges can be avoided by the selection of an Associative Container for the incident edges of a vertex. Although this is a valid solution for the problem, we think that the presence or absence of parallel edges does not need to be coupled with the selected container type. Furthermore, a Unique Associative Container like std::set has a complexity guarantee of at most logarithmic for many operations.

The vertex and edge containers of GTAD’s adjacency list are implemented with std::list. The list of incident edges of a vertex is implemented with a hand-made doubly linked list (nodes in this list are actually pointers to the edges in the container). Since all access to the vertices and edges is done through the auxiliary symbol tables, there is not a strong necessity to parameterize the corresponding containers. In fact, std::list perfectly meets our requirements and as shown in section 6 we have achieved good performance results. Nevertheless, we do provide a separate parametrization for self-loops and parallel edges through the criteria_t template parameter, as the partial definition of our adjacency list class shows below.
template <
    class criteria_t = ...,
    class directionality_t = ...,
    class vertex_t = ...,
    template <class, class> class edge_t = ...,
    template <class> class vertex_list_alloc_t = ...,
    template <class> class edge_list_alloc_t = ...>
class adjacency_list
 //...

The criteria_t parameter is used in a tag dispatching mechanism to verify whether or not a particular edge is allowed to be inserted into the graph. There are four options available: no_parallel_edges, no_self_loops, simple_graph (parallel edges and self-loops are not accepted), and multigraph (parallel edges and self-loops are accepted). Naturally, the lookup procedure is done through the auxiliary symbol table of the edge container and has amortized constant time complexity. Consequently, the impact on the insert operation is very small.

There is not much to say about the second template parameter: directionality_t. Its purpose is to dictate if the graph should be undirected or directed. In a fashion similar to the BGL, the argument for this parameter, which can be either undirected or directed, is used in a type selection mechanism to bind the adjacency list class hierarchy at compile time. This way, some global functions can be overloaded in terms of undirected and directed adjacency lists types. The only observation we must make is that the directed tag of the GTAD is equivalent to the bidirectionalS tag of the BGL, since information about the in-edges of a vertex is always stored. We are already working on an alternative solution in order to let the user decide whether or not the in-edges should be stored. However, since from a mathematical
point of view graphs are either undirected or directed, it might not be the same approach as in the BGL.

A direct parametrization of the vertices and edges of the graph is supported in GTAD’s adjacency list by template parameters `vertex_t` and `edge_t`, which must model, respectively, `Vertex` and `Edge` concepts. This is different from the BGL where such flexibility is accomplished with the attachment of properties. Besides being more natural, an advantage of our approach is that information is self-contained inside the vertex and edge data structures.

Consider, for example, a situation where a map is modeled by a graph. Vertices and edges of the graph correspond, respectively, to cities (class `city`) and highways (class `highway`) of the map. As long as classes `city` and `highway` offer appropriate member functions to manipulate their internal data, objects that are stored in the graph can be accessed by means of two utility functions, as the code below illustrates.

```cpp
typedef adjacency_list<multigraph, undirected, city, highway> Graph;

// Types exported for convenience.
typedef Graph::vertex_type V; typedef Graph::edge_type E;

//... (application code.)

// v is a vertex descriptor and city has members:
// - std::string city::get_name() const;
// - void city::set_name(std::string);
std::string name; get_vertex_property(v, &V::get_name, name);
set_vertex_property(v, &V::set_name, "New York");

// e is an edge descriptor and highway has members:
// - int highway::get_speed_limit() const;
// - void highway::set_speed_limit(int);
int limit; get_edge_property(e, &E::get_speed_limit, limit);
```

For brevity, we do not show these concepts, but they are available at [21].
A great benefit of GTAD’s direct parametrization approach is that there is no need to carry property maps or graph pointers, references or objects around the user application to make information accessible. Only a valid vertex or edge descriptor is necessary.

It remains to talk about the last two template parameters of our adjacency list: `vertex_list_alloc_t` and `edge_list_alloc_t`. As we said before, the vertex and edge containers of GTAD’s adjacency list are implemented with `std::list`. But the choice for the memory management system is left to the user. The default argument for both containers is based on a simple segregated allocation strategy similar to the one presented in [9].

Regarding performance, it is strongly recommended that our adjacency list is initialized with the approximate numbers of vertices and edges that the graph will have. This avoids the resizing of the dynamic symbol tables every time the load factor reaches 0.5. Since these are just approximate numbers, we believe this should not be a problem for users.

As a final and general observation, our opinion is that BGL’s parametrization of the vertex and incident edge containers, if not handled properly, might lead to a rigid code. This is because other parts of the application might be dependent on the selected type for these containers. For example, if a non-careful initial modeling of an application opted for `vecS` as the vertex container, and because of a later requirement change this same container needs now to be implemented with `listS`, parts of the code that assumed integers for mapping properties or other associated data could be broken. We think this contributes negatively for the ease-of-use of the library, and, from our experience, it is one of the reasons beginners find it difficult to use the BGL. Removing this choice away from the client seems to be a valid trade-off.
PERFORMANCE

Although the primary objective of this paper is to describe the overall structure and organization of the GTAD, we also would like to mention that the different design decisions we have made do not incur in significant performance overhead. Actually, for a particular scenario, the GTAD is faster and consumes less memory than the BGL. In this section, we provide performance results for fundamental operations of the adjacency list and a couple of search algorithms.

For the evaluation of the fundamental operations of the graph representation, we have defined a set of operations that are very important in environments where general use of graphs are required. Our set includes the following items:

1. Operation *add vertex* (AV).
2. Operation *add edge* (AE).
3. Operation *remove vertex* (RV).
4. Operation *remove edge* (RE).
5. Operation *remove edge by vertices* (REBV).
6. Operation *find edge* (FE).
7. Operation *vertex traversal* (VT).
8. Operation *edge traversal* (ET).

Semantics of most operations are direct from their names. However, we would like to clarify the meaning of three of them.
• Operation 3 expects that a vertex and its adjacent edges are removed from the graph. It does not make sense to keep edges of a non-existent vertex.

• Operation 4 expects that an edge is removed from the graph. This edge must be passed as argument for the corresponding method.

• Operation 5 expects that one or more edges are removed from the graph. Two vertices must be passed as arguments for the corresponding method. Any edge comprised by these two vertices must be removed.

Our tests were conducted using a graph "model" where the vertices correspond to cities and the edges correspond to highways of a map. The highway class we used for the GTAD is not exactly the same as the one we used for BGL. This is because in our library, the edge type is a template template parameter and must model the Edge concept.

We tested most operations with undirected (designated by U) and directed (designated by D) graphs. Obviously, we did not use all possible combinations for the vertex and incident edge containers of the BGL. Only selectors vecS(adjacency_list<vecS, vecS, ...>) and listS(adjacency_list<listS, listS, ...>) were used, since they are good representatives of the time and space complexity trade-off. Still regarding the BGL, for directed graphs we used the bidirectionalS tag, because we want BGL’s adjacency list to keep information about the in-edges and out-edges of a directed graph. We did allow parallel edges on the graph but no self-loops were inserted. Other general observations related to our tests are:

• Operations add vertex, vertex traversal, and edge traversal were tested with undirected graphs. The remaining ones were tested with both directed and undirected graphs.
<table>
<thead>
<tr>
<th>Op.</th>
<th>Elements</th>
<th>GTAD (ms)</th>
<th>BGL (list)</th>
<th>BGL (vec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AV (U)</td>
<td>$1 \times 10^5$</td>
<td>58</td>
<td>285</td>
<td>402</td>
</tr>
<tr>
<td>AE (U)</td>
<td>$1 \times 10^5$</td>
<td>235</td>
<td>526</td>
<td>574</td>
</tr>
<tr>
<td>AE (D)</td>
<td>$1 \times 10^5$</td>
<td>229</td>
<td>523</td>
<td>566</td>
</tr>
<tr>
<td>RV (U)</td>
<td>$1 \times 10^4$</td>
<td>109</td>
<td>308</td>
<td>very large</td>
</tr>
<tr>
<td>RV (D)</td>
<td>$1 \times 10^4$</td>
<td>101</td>
<td>255</td>
<td>very large</td>
</tr>
<tr>
<td>RE (U)</td>
<td>$5 \times 10^4$</td>
<td>72</td>
<td>188</td>
<td>97</td>
</tr>
<tr>
<td>RE (D)</td>
<td>$5 \times 10^4$</td>
<td>66</td>
<td>151</td>
<td>92</td>
</tr>
<tr>
<td>REBV (U)</td>
<td>$5 \times 10^4$</td>
<td>86</td>
<td>269</td>
<td>102</td>
</tr>
<tr>
<td>REBV (D)</td>
<td>$5 \times 10^4$</td>
<td>67</td>
<td>212</td>
<td>95</td>
</tr>
<tr>
<td>FE (U)</td>
<td>$5 \times 10^4$</td>
<td>45</td>
<td>132</td>
<td>46</td>
</tr>
<tr>
<td>FE (D)</td>
<td>$5 \times 10^4$</td>
<td>35</td>
<td>85</td>
<td>43</td>
</tr>
<tr>
<td>VT (U)</td>
<td>$25 \times 10^3$</td>
<td>1.20</td>
<td>2.33</td>
<td><strong>0.15</strong></td>
</tr>
<tr>
<td>ET (U)</td>
<td>$1 \times 10^4$</td>
<td><strong>5.20</strong></td>
<td>14.15</td>
<td>10.3</td>
</tr>
</tbody>
</table>

- The *add edge* operation was tested with graphs previously constructed with 50000 vertices.
- All variations of the *remove* operations, the *find edge* operation, and *traversal* operations were tested with sparse randomly generated graphs with 25000 vertices and 150000 edges.
- For the *traversal* operations of BGL, not only the actual traversing time was considered, but also the time to access the corresponding property (a city or a highway) from the respective element (a vertex or an edge). This is not necessary in the GTAD, since the vertex and edge descriptors are a direct access to their corresponding properties.

All operations we tested presented an asymptotically linear behavior on the number of vertices or edges in the graph. Table I contains the average values obtained in significant runs of all the tested operations.
It can be seen in table I that for the stated scenario GTAD’s performance was the best for most of the operations under evaluation. For the *add vertex* operation, GTAD’s execution times were much better than BGL’s. For both variations of *add edge*, our implementation was more than two times faster than BGL. In the remaining operations, with an exception to the *vertex traversal* operation, GTAD’s performance was also better. An important observation though is that as opposed to the extremely fast *vertex traversal* operation when BGL’s adjacency list is parameterized with vecS, the *remove vertex* operation for the same parametrization is very slow.

Memory consumption was evaluated through the creation of sparse randomly generated graphs with different numbers of vertices (designated by \( \#V \)) and edges (designated by \( \#E \)). There were no significant differences between memory consumed for undirected and directed (again, considering bidirectionalS for BGL) graphs. Results, available in table II, show that our graph representation is also very efficient regarding memory consumption. We achieved better results than the adjacency list of the BGL even when its vertex and edge containers were parameterized by vecS.

Comparing shortest-paths, maximum flow, biconnectivity, and other algorithms on graphs from real scenarios is neither a simple task nor the focus of this paper. Therefore, we have

<table>
<thead>
<tr>
<th>#V</th>
<th>#E</th>
<th>GTAD (list)</th>
<th>BGL (list)</th>
<th>BGL (vec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>5000</td>
<td>1.1</td>
<td>1.4</td>
<td>1.2</td>
</tr>
<tr>
<td>5000</td>
<td>50000</td>
<td>9.2</td>
<td>12.1</td>
<td>10.3</td>
</tr>
<tr>
<td>10000</td>
<td>150000</td>
<td>26.5</td>
<td>35.0</td>
<td>29.9</td>
</tr>
</tbody>
</table>
Table III. Algorithms execution times (ms)

<table>
<thead>
<tr>
<th></th>
<th>GTAD impl.</th>
<th>BGL impl.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>1.15</td>
<td>0.9</td>
</tr>
<tr>
<td>DFS</td>
<td>1.44</td>
<td>2.17</td>
</tr>
<tr>
<td>GTAD Graph</td>
<td>0.84</td>
<td>1.51</td>
</tr>
<tr>
<td>BGL Graph</td>
<td>7.48</td>
<td>1.66</td>
</tr>
</tbody>
</table>

chosen to compare only a BFS and a DFS, which are underlying components for many other algorithms. We used a recursive DFS implementation and a non-recursive BFS implementation both in the GTAD and in the BGL. Both GTAD’s and BGL’s adjacency list were used as the graph representation. BGL’s adjacency list was parameterized with vecS for both the vertex and out-edges container. The graphs were randomly generated with 1000 vertices and 10000 edges. Table III shows the result for the average values of hundreds of executions.

The best BFS execution time is obtained when running BGL’s implementation with GTAD’s adjacency list. But the times obtained with the other variations of the BFS test are very close to each other. Regarding DFS, the best result is obtained when running GTAD’s implementation with GTAD’s adjacency list. However, when using GTAD’s adjacency list with BGL’s implementation of DFS, the performance is significantly deteriorated.

All tests from this section were performed on a Intel Centrino Duo T2400 (1.83 GHz) machine running Windows XP Professional (SP2) with 2 GB of RAM. We compiled the code with Microsoft Visual C++ Express 8.0 with the higher optimization for speed.
CONCLUSIONS

This paper presents a new generic graph library that is constructed with ideas introduced by the BGL. Since the BGL is a powerful library, we have kept many fundamental construction principles. Particularly, all underlying mechanisms for the external adaptation system are the same. However, we have made significant design changes concerning the structure and organization of the library.

Algorithms in the GTAD are implemented as classes and formalized as concepts. We believe this to be more flexible given the rich combinatorial nature of graphs. Policies, also formalized as concepts, are introduced to incorporate the high level of abstractions in which algorithms can be constructed. By using concepts for both of these items, we make reusability, extensibility, and many other aspects of the generic programming paradigm also available when writing algorithms. Implementation issues like a treatment for an excessive number of template parameters and an idea to parameterize the items to be stored during the algorithm execution are presented.

Concepts are a central key in a generic library. Hence, we review BGL’s concepts, discuss what their drawbacks are, and show how they can be improved.

Since many users of a generic graph library depend on the graph representation provided by the library, we also give details of GTAD’s adjacency list. Our graph representation allows a separate control of the edge insertion criteria and the parametrization of its vertices and edges. A great advantage of this approach is that information related to these elements is accessible directly through the handles. We do not provide parametrization of the containers used to store the vertices and edges of the graph. In fact, we also think that this contributes
to the ease-of-use of the library, since we prevent possible refactorings due to changes in the parametrization of the adjacency list, something that is relatively common to happen in the BGL. Therefore, we prefer to provide interfaces that are easy to use and less likely to lead to problems due to misuse.

It should be clear that this paper does not intend to make a feature comparison between the GTAD and the BGL. The latter is a well known high quality software that has been available for many years. We do intend, however, to expose our library design and discuss the major improvements we believe to have made from the generic programming point of view. As results from section 6 show, such improvements were made without compromising the performance of the library.

REFERENCES


21. GTAD (Graph Toolkit for Algorithms and Drawings). http://gtad.sourceforge.net/ [1 April 2008].


**APPENDIX I: Algorithm Concepts**

This appendix briefly describes the *Single Source Shortest Path Algorithm* concept and its related concepts. Detailed documentation is available at [21].

**Graph Algorithm**
Notation

A - A type that is a model of Graph Algorithm.

a - An object of type A.

G - A type that is a model of Graph.

g - An object of type G.

Associated types

graph_type

Valid expressions/Return type

a.execute(g)/void

Weighted Graph Algorithm

Refinement of

Graph Algorithm

Notation

A - A type that is a model of Weighted Graph Algorithm.

a - An object of type A.

WM - A type that is a model of Readable Property Map. The key type must be an edge descriptor. The value type must be a model of Arithmetic.

wm - An object of type WM.

wv - An object of type property_traits<WM>::value_type.

Associated types

weight_map_type

Valid expressions/Return type
a.is_weight_map_set() / bool
a.set_weight_map(wm) / void
a.get_weight_map() / A::weight_map_type
a.is_infinity_weight_set() / bool
a.set_infinity_weight(wv) / void
a.get_infinity_weight() / property_traits<A::weight_map_type>::value_type
a.is_zero_weight_set() / bool
a.set_zero_weight(wv) / void
a.get_zero_weight() / property_traits<A::weight_map_type>::value_type

**Single Source Shortest Path Algorithm**

*Refinement of*

Weighted Graph Algorithm

*Notation*

A - A type that is a model of Single Source Shortest Path Algorithm.

a - An object of type A.

v - An object of type graph_traits<A::graph_type>::vertex_descriptor.

g - g An object of type A::graph_type.

*Valid expressions/Return type*

a.is_source_set() / bool

a.set_source(v) / void

a.get_source() / graph_traits<A::graph_type>::vertex_descriptor

a.shortest_path(v, g) / std::vector<graph_traits<A::graph_type>::edge_descriptor>
a.raw_shortest_path() / std::vector<graph_traits<A::graph_type>::edge_descriptor>

a.distance(v) / property_traits<A::weight_map_type>::value_type

a.distance() / std::vector<property_traits<A::weight_map_type>::value_type>

APPENDIX II: Policy Concepts

This appendix briefly describes the Find Augmenting Path Policy concept. Detailed documentation is available at [21].

**Find Augmenting Path Policy**

**Notation**

P - A type that is a model of Find Augmenting Path Policy.

A - A type that is a model of Lower Bounded Algorithm, Policy Based Construct, and Vertex Indexed Construct.

a - An object of type A.

G - A type that is a model of Graph Algorithm.

g - An object of type G.

s, t - Are objects of type graph_traits<G>::vertex_descriptor.

vim - An object of type A::vertex_index_map_type.

cm, rcm, lbm - Are objects of type A::flow_unit_map_type.

pd - An object of type A::policy_data_type.

vim - An object of type A::vertex_index_map_type.

z - An object of type property_traits<A::flow_unit_map_type>::value_type.

path - An object of type std::vector<gtad::graph_traits<graph_type>::edge_descriptor>.
Associated types

graph_type

algorithm_type

Valid expressions/Return type

P<A>::execute(a, g, s, t, vim, cm, rem, lbm, rcm, pd, z, path)/void

APPENDIX III: Graph Concepts

This appendix briefly describes some of the GTAD graph concepts that differ from some of the BGL graph concepts. Detailed documentation is available at [21].

Graph

Notation

G - A type that is a model of Graph.

Associated types

graph_traits<G>::vertex_descriptor

graph_traits<G>::edge_descriptor

Countable Vertex Set Graph

Refinement of

Graph

Notation

G - A type that is a model of Countable Vertex Set Graph.

g - An object of type G.

Associated types
graph_traits<G>::vertices_size_type

Valid expressions/Return type

num_vertices(g)/vertices_size_type

**Countable Edge Set Graph**

Refinement of

Graph

Notation

G - A type that is a model of Countable Edge Set Graph.

g - An object of type G.

Associated types

graph_traits<G>::edges_size_type

Valid expressions/Return type

num_edges()/edges_size_type

**Traversable Vertex Set Graph**

Refinement of

Countable Vertex Set Graph

Notation

G - A type that is a model of Traversable Vertex Set Graph.

g - An object of type G.

Associated types

graph_traits<G>::vertex_iterator

Valid expressions/Return type
vertices(g)/std::pair<vertex_iterator, vertex_iterator>

**Traversable Edge Set Graph**

**Refinement of**

Countable Edge Set Graph

**Notation**

G - A type that is a model of Traversable Edge Set Graph.

g - An object of type G.

**Associated types**

graph_traits<G>::edge_iterator

**Valid expressions/Return type**

edges(g)/std::pair<edge_iterator, edge_iterator>

**Edge Findable Graph**

**Refinement of**

Graph

**Notation**

G - A type that is a model of Edge Findable Graph.

g - An object of type G.

u - An object of type graph_traits<G>::vertex_descriptor.

v - An object of type graph_traits<G>::vertex_descriptor.

**Valid expressions/Return type**

edge(u, v, g)/std::pair<edge_descriptor, bool>