STAPL Graph Generator Library *

Harshvardhan Saransh Mittal Kelli Bacon
Olga Tkachyshyn Chidambareswaran Raman Nancy M. Amato

Parasol Lab, Dept. of Computer Science, Texas A&M University
{ananvay,olgat,chids,amato}@cs.tamu.edu

August 16, 2006

Abstract

This research aims to design a generic Graph Generator Library(GGL) as a part of the Standard Template Adaptive Parallel Library (STAPL) framework. An important container in STAPL is the graph and its parallel equivalent, the pGraph. This generic library provides functionalities to generate graphs in many ways via random, user specified traits or stencil based patterns. A stencil is a user provided subgraph. Our framework will support joining of preexisting graphs to make a composite of them, and use this functionality to generate output graphs. Besides this, the framework will also support ways to calculate and store various properties of the graphs. A property of a graph is a value that describes an aspect of the graph, i.e. number of edges, diameter, number of connected components. These properties are computed, and the framework has functionality to have a valid current value for the properties as the graph is modified. The framework also supports optimized techniques to keep the property values current and valid. Maintaining the properties for the graphs facilitates graph generation based on a particular value set corresponding to a property (sub)set. Further, it provides a quick way for the user to query the properties of the graph being modeled. Many real world applications like maps, and scheduling model data in the form of graphs, would be able to use GGL . This will definitely make the task of generating test datasets easier, improving the testing confidence level for these applications. In the STAPL framework, we plan on using GGL for task scheduling problems and data dependency graphs generation. Future work includes extension of these techniques to generate pGraphs.

*This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, and by the DOE.

Mittal (saransh@cse.iitb.ac.in) from the Department of Computer Science and Engineering at the Indian Institute of Technology Bombay.
Bacon (kbacon@gonzaga.edu) from the School of Engineering, Department of Computer Science, Gonzaga University.

Work performed at the Parasol Lab during research internship in Summer 2006 and Bacon supported in part by the CRA Distributed Mentor Project.

1 Introduction

The Graph Generator Library (GGL) is a toolset to make it easier and faster to make and update graphs. Graphs are used in many disciplines to represent connections or relationships between pieces of information. Motion Planning uses graphs to plot paths for robots. Map makers use graphs to plot and calculate routes. As useful as graphs can be for a wide range of applications, they are tedious and time-consuming to set up, especially if it is needed to be done many times. Our toolset makes this task comparatively easy.

A graph can be set up in a number of ways. Graphs also have many traits associated with them (i.e. number of edges, completeness). We have implemented calculation methods for some of these graph characteristics as part of our GGL design (Section 2). We have also implemented optimized calculation techniques for determining graph information from previous graph knowledge (Section 3). For programming purposes, a graph library provides the tools necessary to build a graph structure. While some previous groups have developed graph libraries, they do not generally support graph generation. There are also some graph generators already set up and used in the computing world, but we feel that our combination of these useful tools (Section 4) will prove beneficial.

2 The Graph Varieties and Properties

In mathematics, a graph is a visual representation of a data set. Graphs can show relationships and tendencies of mathematical formulas that are usually only symbolic or abstract. In computer science, graphs are data structures based on the mathematical graph used to store information according to relations. They are usually implemented as an abstract data type (ADT) that consists of a list of nodes and a list of edges. Nodes store the data, and they are connected to each other by edges. The list of edges establishes the connections, or relationships, between these nodes.

There are several ways to represent Graph Data Structures:

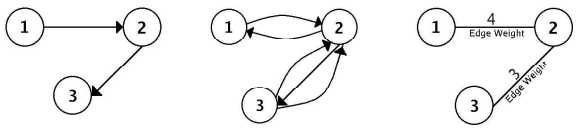
- As an Adjacency List, which stores a list of incident edges in each node
- As an Adjacency Matrix, which represents the graph as a boolean square matrix (M). The value at Mij specifies if an egde exists between nodes i and j.
- As a list of edges, with source and target information

Our graph is represented as a pair*(V,E)*. The vertext set *V* is a finite set of vertices and the edge set *E* describes the relation on *V*. An edge is a set (*a,b*) where $a,b \in V$ [3].

2.1 Graph Varieties

Graphs are set up from a combination of orthogonal varieties. A graph can be *directed* or *undirected*, *weighted* or *unweighted*, *multiedge* or *non-multiedge*. We define these terms as follows:

- **Directed** (DG) - the pair *(a,b)* is ordered; the edge set *E* refers to edges that are incident from, or leaving, a vertex from *V*. Vertices are visually represented by circles and edges by arrows. In a DG , an edge from a vertex to itself (as in *(a,a)*) is possible [3].
- **Directed Predecessor** (DPG) - has the same properties as a DG , with the addition of a *predecessor set* that contains edges that are incident to, or entering, a vertex.
- **Undirected** (UG) - the edge set *E* contains unordered pairs. In an UG , *(a,b)* is the same as *(b,a)* and an edge cannot exist from a vertex to itself [3].
- **Weighted** (WG) - edges are each assigned a value called the weight. The value can represent whatever the user wants it to, i.e. a distance or time.
- **Unweighted** (UWG) - edges are not assigned a weight, thus the information associated with the edge defaults to -1.
- **Multiedge** (MG) - allows multiple edges to connect any given pair of vertices.
- **Non-Multiedge** (NMG) - can only have up to one edge per pair of vertices.



(a) DG, UWG, NMG (b) DG, UWG, MG (c) UG, WG, NMG

Figure 1: Example Graphs. (a) A Directed, Unweighted, Non-Multiedge Graph. (b) A Directed, Unweighted, Multiedge Graph. (c) An Undirected, Weighted, Non-Multiedge.

2.2 Graph Properties

Depending on the type of graph you are working with, there are many characteristics of that graph to consider. We refer to characteristics as the properties of the graph. As part of our generator framework, we have implemented calculation methods for some of these properties.

We define the subset of property calculations we have implemented as follows:

2.2.1 All Graphs

- **Number of Vertices** - The total number of nodes, or objects, in the graph.
- **Number of Edges** - The total number of connections between nodes, or vertices in the graph.
- **Diameter** - The longest of the shortest paths from any given vertex to any other vertex in the graph is the graph's diameter. In an UWG , the diameter is found as if the edge weights were all 1.

2.2.2 Undirected Graphs

- **Number of Connected Components** - The measure of connectedness of the graph.
- **Maximum Degree** - The maximum number of edges incident on a vertex of the graph.
- **Minimum Degree** - The minimum number of edges incident on a vertex of the graph.
- **Average Degree** - The average number of edges incident on a vertex of the graph.

2.2.3 Directed Graphs

- **Acyclic** - A cycle is a sequence of vertices $< v1, v2, ...vk >$ in which $v1 = vk$ [3]. Thus an acyclic graph is one in which there is no cycle.
- **Maximum Out Degree** - The maximum number of edges leaving a vertex.
- **Minimum Out Degree** - The minimum number of edges leaving a vertex.
- **Average Out Degree** - The average number of edges leaving a vertex.

2.2.4 Directed Predecessor Graphs

- **Maximum In Degree** - The maximum number of edges entering a vertex.
- **Minimum In Degree** - The minimum number of edges entering a vertex.
- **Average In Degree** - The average number of edges entering a vertex.

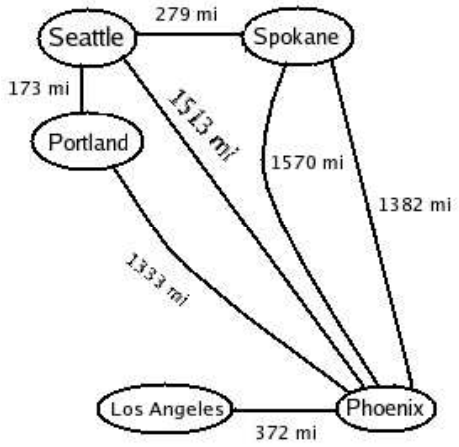


Figure 2: Example Graph: A Map

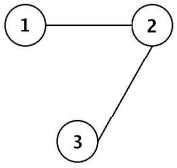
Property	value
number of vertices	5
number of edges	7
diameter	2221 mi
number of CC	1
maximum degree	5
minimum degree	1
average degree	2.8

Table 1: Property Values for Figure 2

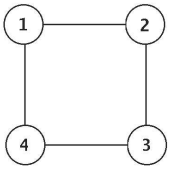
3 Graph Composition Methods and Properties

3.1 Composition Methods

We have identified four fundamental ways in which graphs can be combined. These "compositional operators" can be used as building blocks for combining graphs to make the desired output graph. The compositional operators are:



(a) Graph A



(b) Graph B

Figure 3: (a) and (b) are the original Graphs.

- **Empty Join:** Original graphs are merged into one graph without being connected to each other. (Figure 4)
- **Join At Vertex:** Original graphs are connected at a specified vertex. The corresponding vertices in both graphs are merged together in the new graph. (Figure 5)
- **Join At Edge:** Original graphs are connected at a specified edge. The corresponding edges in both graphs are merged together in the new graph. (Figure 6)
- **Join By Edge:** Original graphs are connected by adding an edge between the corresponding vertices of the two graphs. (Figure 7)

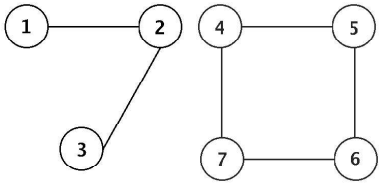


Figure 4: Graphs A & B: Empty Join

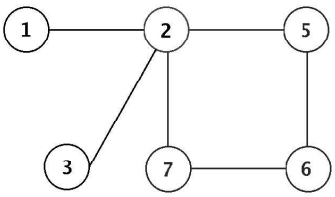


Figure 5: Graphs A & B: Join At Vertex

Each join starts by combining the graphs with an empty join operation, which relabels all the vertices so that each vertex of the new graph has a unique ID. After relabeling, the appropriate join method is called which connects the different components together. For a "join at vertex", two vertices are specifies to be merged together, keeping the ID of the first vertex. In a "join at edge", two edges are

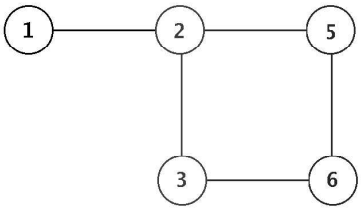


Figure 6: Graphs A & B: Join At Edge

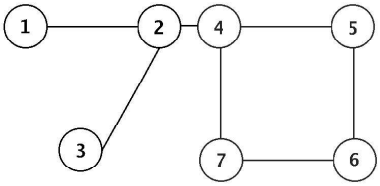


Figure 7: Graphs A & B: Join By Edge

specified as two pairs of vertices. Like "join at vertex", the merged edge keeps the first IDs. A "join by edge" call takes the vertex IDs to be joined and simply adds an edge between them. In the DGcase, the edge is added from the first vertex to the second.

3.2 Compositional Properties

The compositional properties provide a way to recalculate the property values after changing the graph. When the graph is modified, for example adding an edge or a vertex, some property values of the graph might change. Previously, these changes were reflected by recalculating the property. However, we found that we could optimize the calculation if we knew the initial value of the property and the graph modification being performed. Based on this, we were able to reduce most complex algorithms to one line formulas. This drastically reduced the computational cost (See Table 2).

After a graph is joined, the composing method calls the corresponding recalculate method for each property in the graph. The recalculate method then determines if there is sufficient previous data to optimize the calculation. If so, the recalculate method uses the optimized method for updating the property value (if there exists an optimized version for the property). If the data is not sufficient, the recalculate method computes from scratch.

3.3 Algorithms

For each of the joining methods above, we developed algorithms to calculate the property value for the new graph. Our goal was to call upon previously recorded knowledge of the original graphs to reduce cost. Below are the algorithms for the simplist property calculations, *number of*

	Property	number of vertices	number of edges	diameter
Calculate from Scratch	Complexity	$O(n)$	$O(n \times m)$	$O(n^3 \times m)$
	Why?	every vertex must be visited	every edge of each vertex must be visited	depends on the type of graph
Optimized Calculate				
Empty Join	Complexity	$O(1)$	$O(1)$	$O(1)$
	How?	G1.num_vertices + G2.num_vertices	G1.num_edges + G2.num_edges	max(G1.diameter, G2.diameter)
Join At Vertex	Complexity	$O(1)$	$O(1)$	$O(1)$
	How?	G1.num_vertices + G2.num_vertices - 1	G1.num_edges + G2.num_edges	max(G1.diameter, G2.diameter, diameter through joined vertex)
Join At Edge	Complexity	$O(1)$	$O(1)$	$O(n^3 \times m)$
	How?	G1.num_vertices + G2.num_vertices	G1.num_edges + G2.num_edges - 1	must recalculate from scratch
Join By Edge	Complexity	$O(1)$	$O(1)$	$O(1)$
	How?	G1.num_vertices + G2.num_vertices	G1.num_edges + G2.num_edges + 1	max(G1.diameter, G2.diameter, diameter through new edge)

Table 2: Comparing Computational Cost

vertices and *number of edges*, as well as the most complex, the property *diameter*. See Table 2 for a quick comparison of the costs.

Number of Vertices, as defined in Section 2.2, maintains the total number of vertices in the graph. The algorithm to find this value without any prior knowledge of the graph is not very involved, just a run through the list of vertices in the graph structure (Algorithm 3.1). This has a cost of n, where *n* is the number of vertices. However by using prior knowledge of graphs, finding the total number of vertices in a composed graph can have a smaller cost, namely a constant time algorithm. Our implementation is outlined in Algorithm 3.2.

Algorithm 3.1 property_num_vertices

```
1: for each vertex in the graph structure do
2:     increment count
3: end for
4: return count
```

Algorithm 3.2 compositional_property_num_vertices

```
1: set join to type of join
2: num_vertices = Graph1.num_vertices + Graph2.num_vertices
3: CASE join at vertex:
4: num_vertices = Graph1.num_vertices + Graph2.num_vertices
   - 1
5: CASE join at edge:
6: num_vertices = Graph1.num_vertices + Graph2.num_vertices
7: CASE join by edge:
8: num_vertices = Graph1.num_vertices + Graph2.num_vertices
```

Number of Edges, as defined in Section 2.2, maintains the total number of edges in the graph. The algorithm to find this value without any prior knowledge of the graph finds the number of edges per every vertex in the graph and adds them together (Algorithm 3.3). This has a cost of $n \times m$, where *n* is the number of vertices and *m* is the number of edges. By using any prior knowledge of the graphs, finding the total number of edges in a composed graph can have a constant time cost like that for *number of vertices*. Our implementation is outlined in Algorithm 3.4.

Algorithm 3.3 property_num_edges

```
1: for each vertex in the graph structure do
2:     for every vertex in the edge list do
3:         increment count
4:     end for
5: end for
6: return count
```

Algorithm 3.4 compositional_property_num_edges

```
1: CASE empty join:
2: num_edges = Graph1.num_edges + Graph2.num_edges
3: CASE join at vertex:
4: num_edges = Graph1.num_edges + Graph2.num_edges
5: CASE join at edge:
6: num_edges = Graph1.num_edges + Graph2.num_edges - 1
7: CASE join by edge:
8: num_edges = Graph1.num_edges + Graph2.num_edges + 1
```

Diameter, also defined in Section 2.2, contains the longest of the shortest paths in the graph. Without any previous knowledge of the graph this value is found through an applied algorithm, depending on the type of graph. In the worst case senario, the complexity of our algorithm is $O(n^3 \times m)$. These methods are outlined in Algorithm 3.5). In most cases, using our optimized calculate calls for diameter cuts down the cost in computing, seen in Algorithm 3.6.

4 Libraries

4.1 STAPL Graph

Our graph data structure is implemented in the STAPLframework. STAPL, the Standard Template Adaptive Parallel Library, is a parallel C++ library being developed in the Parasol Laboratory at Texas A&M University. It is

Algorithm 3.5 property_diameter

```
1: CASE "WG with positive weights": use dijkstra's algorithm
2: for every starting point (vertex) do
3:    for every ending point (vertex) do
4:       path = dijkstra
5:       for every location in the path do
6:          current_distance = current_distance + pathLocation.weight
7:       end for
8:       if current_distance   diameter_value then
9:          diameter_value = current_distance
10:      end if
11:   end for
12: end for
13: CASE "WG with negative weights": use bellman ford
14: for every starting point (vertex) do
15:   for every ending point (vertex) do
16:      path = bellmanFord
17:      for every spot in the path do
18:         current_distance = current_distance + pathLocation.weight
19:      end for
20:      if current_distance   diameter_value then
21:         diameter_value = current_distance
22:      end if
23:   end for
24: end for
25: CASE "UG":
26: for every starting point (vertex) do
27:   for every ending point (vertex) do
28:      path = dijkstra
29:      for every location in the path do
30:         current_distance = current_distance + 1
31:      end for
32:      if current_distance   diameter_value then
33:         diameter_value = current_distance
34:      end if
35:   end for
36: end for
37: CASE "more than one CC":
38: for every CC do
39:   if this_diameter   diameter_value then
40:      diameter_value = this_diameter
41:   end if
42: end for
```

Algorithm 3.6 compositional_property_diameter

```
CASE empty join:
diameter = maximum of Graph1.diameter and Graph2.diameter
CASE join at vertex:
diameter = maximum of Graph1.diameter, Graph2.diameter and through_joined_vertex.diameter
CASE join at edge:
must recalculate diameter
CASE join by edge:
diameter = maximum of Graph1.diameter, Graph2.diameter and through_new_edge.diameter
```

based on a library of ISO standard C++ components with interfaces similar to the (sequential) Standard Template Library (C++ STL). The standard STL, however, does not provide a graph container. Hence, the STAPL graph is a unique container which extends the facilities provided by the C++ standard. The STAPL graph stores its data in the form of a vector of vertices. Each vertex has a vector which contains the list of vertices that it connects to.

One way to cut down the time required to make and modify a graph is to set it up in parallel. One of STAPL's containers is the parallel graph, or pGraph. The pGraph provides all of the functionality a traditional sequential graph and is already being used on a number of projects in the Parasol Lab. This parallel graph was designed for applications which require highly scalable data structures, which is our goal as well, and is implemented using a vector of nodes. At execution, the parallel graph is physically distributed over multiple processors.

4.2 Other Libraries

There are currently a few external (not standard ISO C++) graph libraries for C++, like the Boost Graph Library, which is similar to the STAPL Graph, but none of them include any graph generation facilities. Currently, the only way to generate a graph is to manually add vertices to the graph and then connect the vertices together with edges. This can get very cumbersome and time-consuming, especially since most graph applications require graphs that can easily contain thousands of vertices and edges. This being said, a typical graph requires hours of manual labor. We aim to cut down this time by automating the actual process of generation.

4.3 Graph Generation

Our framework generates graphs using the properties of the graph and the compositional operators. The graph can be generated using the following steps.

First, the users specify the desired properties of the output graph. For example, a graph with 98 vertices and a diameter of 20. Next, they assign priorities and tolerance limits to these properties. For example, give priority to the diameter, and the number of vertices can have a value within 10% accuracy. The graph generator can then try to optimize the graph strictly for the diameter, and put less emphasis on the number of vertices, as long as it is within the tolerance limit.

Another option is for the users to specify a stencil, which is a "bulding block" element used to construct the graph (see Figure 8). The stencil defines the structure of the fundamental element that can be used to build the output graph, and the generator combines these stencils together to get the desired output.

Stencils can be combined in the following ways to generate the output:

- Regular Combination of Stencils (Mesh): The stencils are combined in a repeated pattern to create a mesh, or grid. The user can specify the number of stencils to be used and/or the pattern, for example: create a 3X4 mesh (Figure 9).
- Irregular Combination: The stencils are combined depending on the properties. The generator takes two stencils and combines them using one of the four compositional operators at random. It then computes the desired property values, and compares them to the final values. If the final values are not reached, it combines the output graph from the previous step with another stencil, and recalculates the properties. It then compares the property values again, and keeps repeating this process until the desired values have been reached. This method heavily relies on the compositional properties of the graph.
- User Determined Combination: In this method, the user specifies the order in which to combine stencils, and also decides which compositional operator to use at each step.

4.3.1 User Specified

As seen in Section 2.2, we have implemented calculate methods for some graph properties. We are aware that a user may have their own properties to add to this list. Our framework allows users to add their own properities in the following way:

- The user must specify the resulting value type (i.e. integer, boolean, etc.).
- The user must specify a method to calculate the property without any previous knowledge about the value.

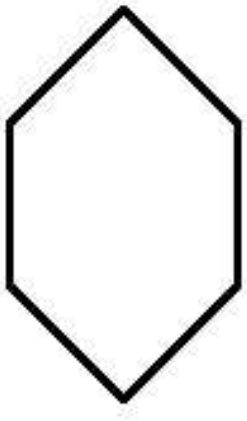


Figure 8: A stencil 'S'

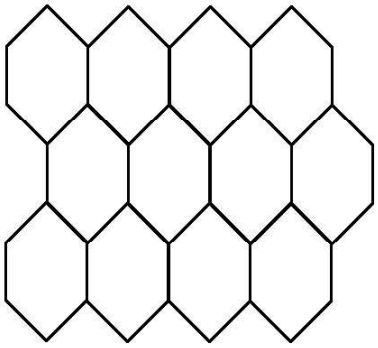


Figure 9: A 3X4 mesh of stencil 'S'

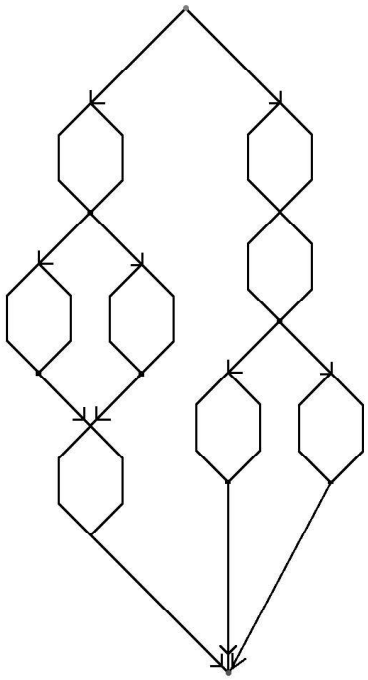


Figure 10: A random network of stencil 'S'

The user could also specify optimized methods to calculate the property for situations in which initial values are known. (See Section 3 for more details on optimized calculations.) This helps in speeding up property calculations when changes to the graph are made, but is not required.

5 Conclusions and Future Work

Our framework provides multiple ways to automate the task of generating graphs. With graphs being used in various applications, the framework will find its use in a variety of places. This framework should increase productivity, due to the ease of generating graphs and the more efficent way to keep track of properties.

In the future, we plan to parallelize the framework to be able to generate graphs on parallel systems. We would also need to parallelize the property calculation and combination methods to maintain the efficiency of the framework. Finally, we plan to use the library for applications such as Scheduling, Motion Planning, Protein/RNA folding, and Campus Navigator.

6 Acknowledgments

We would like to thank Roger Pearce for setting up our code repository and listserv, and Nathan Thomas for giving us insight on elegant C++ programming.

7 References

[1] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, Chicago, Ill, Jun 2005.

[2] S. Thomas, G. Tanase, L. Dale, J. Moreira, L. Rauchwerger, N. Amato. Parallel Protein Folding with STAPL. In *Concurrency and Computation: Practice and Experience*, Dec 2005.

[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms. Second Edition, Nov 2002.

[4] R. Johnsonbaugh, M. Kalin. Graph Generation Software Package. Chicago, Ill. http://condor.depaul.edu/~rjohnson/source/graph_ge.c

[5] L. Lee, A. Lumsdaine, J. Siek. The Boost Graph Library: User Guide and Reference Manual. First Edition, Dec 2001

[6] Standard Template Library Programmer's Guide. http://www.sgi.com/tech/stl/

[7] G. Tanase. Adaptive Parallel Containers in STAPL, Phd Proposal Dept of CS. Texas A&M University.

[8] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, L. Rauchwerger. STAPL: An Adaptive, Generic Parallel C++ Library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pp. 193-208, Cumberland Falls, Kentucky, Aug 2001.