BRITE: Universal Topology Generation from a User's Perspective *

Alberto Medina, Anukool Lakhina, Ibrahim Matta, John Byers
{amedina, anukool, matta, byers}@cs.bu.edu
Computer Science Department
Boston University
BUCS-TR-2001-003

April 12, 2001

Abstract

Effective engineering of the Internet is predicated upon a detailed understanding of issues such as the large-scale structure of its underlying physical topology, the manner in which it evolves over time, and the way in which its constituent components contribute to its overall function. Unfortunately, developing a deep understanding of these issues has proven to be a challenging task, since it in turn involves solving difficult problems such as mapping the actual topology, characterizing it, and developing models that capture its emergent behavior. Consequently, even though there are a number of topology models, it is an open question as to how *representative* the topologies they generate are of the actual Internet. Our goal is to produce a topology generation framework which improves the state of the art and is based on design principles which include representativeness, inclusiveness, and interoperability. *Representativeness* leads to synthetic topologies that accurately reflect many aspects of the actual Internet topology (e.g. hierarchical structure, degree distribution, etc.). *Inclusiveness* combines the strengths of as many generation models as possible in a single generation tool. *Interoperability* provides interfaces to widely-used simulation and visualization applications such as *ns* and *SSF*. We call such a tool a *universal topology generator*.

In this paper we discuss the design, implementation and usage of the BRITE universal topology generation tool that we have built. We also describe the *BRITE Analysis Engine, BRIANA*, which is an independent piece of software designed and built upon BRITE design goals of flexibility and extensibility. The purpose of BRIANA is to act as a repository of analysis routines along with a user–friendly interface that allows its use on different topology formats.

Keywords: topology generation, graph models, network topology, growth models, annotated topologies, simulation environments.

*BRITE is in part funded by NSF grants CAREER ANI-0096045 and ANI-9986397.

1 Introduction

To effectively engineer the Internet, crucial issues such as the large scale structure of its underlying physical topology, its time evolution and the contribution of its individual components to its overall function need to be well understood.

During the design phase of an Internet-based technology, extensive simulations are usually performed to assess its feasibility, in terms of efficiency and performance. In general, Internet studies and simulations assume certain topological properties or use synthetically generated topologies. If such studies are to give accurate guidance as to Internet–wide behavior of the protocols and algorithms being studied, the chosen topologies must exhibit fundamental properties or invariants empirically found in the actual extant structure of the Internet. Otherwise, correct conclusions cannot be drawn.

Unfortunately, achieving a deep understanding of the topology of the Internet has proven to be a very challenging task since it involves solving difficult problems such as mapping the actual topology, characterizing it, and developing generation models that capture its fundamental properties. In addition, the topology of the Internet is a target that is constantly evolving, and it is controlled by a set of autonomous authorities that are not often willing to exchange low-level connectivity information [21].

There are several synthetic topology generators available to the networking research community [25, 8, 5, 16, 13, 1]. Many of them differ significantly with respect to the characteristics of the topologies they generate. A researcher is faced with the question of which topology generator to use for a specific simulation study. The answer may be to use a specific generator, or perhaps to use several topologies generated by different generators. More challenging yet, a completely new generator may be required if the existing generation models do not address specific issues of importance to a particular study. Furthermore, existing topology generators fail to produce complete representations of the Internet since they focus primarily on network connectivity or structural characteristics only, and do not attempt to model other properties of the network such as link bandwidths and delays.

Our objective caters to two groups of researchers. On the one hand, there are researchers investigating Internet protocols and algorithms who need topology generation tools to obtain good synthetic topologies that are the base of their simulations. On the other hand, there are researchers (like us) investigating the challenges associated with generating accurate synthetic topologies. For both groups it would be very useful to have topology generation tools that allow them to easily evaluate the pros and cons of new generation models.

An attractive scenario is to have a topology generation tool that provides a researcher with a wide variety of generation models, as well as the ability to easily extend such a set by combining existing models or adding new ones. In this paper we discuss the design and implementation of BRITE, the Boston university Representative Internet Topology gEnerator, which is a tool designed to realize this scenario.

This paper is organized as follows. In Section 2 we discuss the challenges that must be tackled to generate accurate synthetic topologies, what are the characteristics of an ideal generation tool and the approach taken to achieve these. In Section 3 we describe the general design of BRITE and some implementation details. In Section 4 we describe the graphical user interface (GUI) that can be used as a front-end to

BRITE. In Section 5 we provide a walk-through of generating a topology using BRITE. Section 6 explains how to extend BRITE by adding a new generation model. Section 7 briefly describes the BRITE Analysis Engine. Section 8 presents some results obtained using BRITE as the generation tool. Section 9 presents concluding remarks. In the Appendices we describe several technical issues of interest to BRITE users. Appendix A discusses the issue of including heavy-tailed distributions in several aspects of BRITE. Appendix B gives a summary of the parsing support routines provided by BRITE. Finally, Appendix C describes how to download and install BRITE.

2 Wish List for a Topology Generator

An ideal topology generator should enable the use and development of generation models that produce accurate representations of Internet topologies. Thus, it should include features that appeal to the researcher who is in need of accurate synthetic topologies for studying the correctness and performance of protocols and algorithms, as well as to the researcher who is in search for better and more powerful generation models. The following is a list of desirable characteristics for a topology generator.

1. *Representativeness.* Produces accurate synthetic topologies. Accuracy should be reflected in as many aspects of the actual Internet topology as possible (e.g. hierarchical structure, degree distribution characteristics, etc.).

2. *Inclusiveness*. Combines the strengths of as many generation models as possible in a single generation tool.

3. *Flexibility*. Generates topologies over a wide range of sizes. Restrictions such as minimum and maximum number of nodes should be reasonably avoided.

4. *Efficiency*. Generates large topologies (e.g. number of nodes $> 100,000$) with reasonable CPU and memory consumption.

5. *Extensibility*. Provides mechanisms that allow the user to easily extend its capabilities by adding new generation models.

6. *User–friendliness*. Follows the usage principles of standard user interfaces. The user should learn the mechanics of the generation tool only once. For each generation model incorporated in the tool, she should only need to learn the functionality associated with the new model.

7. *Interoperability.* Provides interfaces to main simulation and visualization applications. It should be possible to generate topologies that can be processed by widely used simulators such as *ns* [18] and *SSF* [22].

8. *Robustness.* Does not sacrifice robustness in the name of efficiency and includes extensive error detection capabilities.

In Section 2.1 we describe the main topology generators and generation models available, in Section 2.2 we discuss some challenges that must be overcome to develop a universal generation tool satisfying our wish list, and in Section 2.3 we argue about a possible approach to tackling those challenges.

2.1 Available Topology Generators

There are several topology generators available to the research community. Some of them mainly aim to generate random topologies [25], others aim to imitate the hierarchical properties of the Internet [5, 8], and still others aim to reproduce degree-related properties of the Internet [16, 13, 1]. Each of these generators implement a different set of generation models. Selecting one for a particular study depends on several factors [26], including the nature of the study to be performed, the size of the required generated topology, the weight certain characteristics of the generated topologies may have (e.g. structural properties such as hierarchical structure, or connectivity properties such as the distribution of outdegrees of the nodes), etc. A brief description of the main topology generators available follows.

Waxman [25] developed one of the first topology generators. This generator produces random graphs based on the Erdös-Renyi [4] random graph model, but it includes network–specific characteristics such as placing the nodes on a plane and using a probability function to interconnect two nodes in the Waxman model which is parameterized by the distance that separates them in the plane.

One of the most popular generators available is GT-ITM [5]. The main characteristic of GT-ITM is that it provides the Transit-Stub (TS) model, which focuses on reproducing the hierarchical structure of the topology of the Internet. In the TS model, a connected random graph is first generated (e.g. using the Waxman method or a variant of it). Each node in that graph represents an entire *Transit domain*. Each transit domain node is expanded to form another connected random graph, representing the backbone topology of that transit domain. Next, for each node in each transit domain, a number of random graphs are generated representing *Stub domains* that are attached to that node. Finally, some extra connectivity is added, in the form of "back-door" links between pairs of nodes, where a pair of nodes consists of a node from a transit domain and another from a stub domain, or one node from each of two different stub domains. GT-ITM also includes about five flavors of flat random graphs.

Another generator that implements models trying to imitate the structure of the Internet is Tiers [8]. The generation model of Tiers is based on a three-level hierarchy aimed at reproducing the differentiation between Wide-Area, Metropolitan-Area and Local-Area networks comprising the Internet.

BRITE 1.0 [16] is the precursor to the universal generation tool we are presenting in this paper. BRITE 1.0 implements a single generation model that has several degrees of freedom with respect to how the nodes are placed in the plane and the properties of the interconnection method to be used. Under certain configuration of the parameters, BRITE 1.0 generation model is equivalent to Waxman. Under other configuration of parameters, BRITE 1.0 implements the Barabási-Albert model proposed in [2] in which a network grows incrementally and the nodes interconnect with preference towards higher degree nodes.

Inet [13] and PLRG [1] are two generators aimed at reproducing the connectivity properties of Internet topologies as reported in [9]. These generators initially assign node degrees from a power-law distribution

and then proceed to interconnect them using different rules. Inet first determines whether the resulting topology will be connected, forms a spanning tree using nodes of degree greater than two, attaches nodes with degree one to the spanning tree and then matches the remaining unfulfilled degrees of all nodes with each other. PLRG works similarly to Inet in that it takes as an argument the number of nodes to be generated and exponent value α. This exponent value is the parameter of a power-law distribution which is used to assign a priori degrees to the nodes of the topology. For any given node n with degree d_v, n is cloned d_v times and then the resulting nodes are randomly interconnected.

Another set of topologies for which special generators are not required are *regular topologies* such as the mesh, star, tree, ring, lattice, etc. These topologies have the advantage that they are very simple, and are generally used for simplicity or to simulate specific scenarios such as LANs or other shared communication media. Finally we note that not all existing topology generation models have been implemented in a generator — the "small-world" model of [24] is one such example.

As we can see, there are a wide variety of generators available. Most of them differ in fundamental ways. For example, Waxman is concerned only with general random networks, GT-ITM and Tiers are concerned with the hierarchical properties of the Internet, BRITE 1.0, Inet and PLRG are concerned with resemblance to Internet topologies in terms of connectivity properties, and regular topologies are concerned only with specific and restricted scenarios. Furthermore, generators such as Inet and PLRG can be characterized as *causality–oblivious* since they do a fairly good job reproducing for example the outdegree distribution of Internet topologies but their corresponding generation models do not provide insights into why such properties arise in the Internet in the first place. GT-ITM and Tiers are concerned with more specific hierarchical properties which are related to how the Internet is organized. However, the fact that they fail to reproduce properties of the Internet [9] suggests that the generation models they implement are lacking some fundamental characteristics. BRITE 1.0 could be characterized as a *causality–aware* generator since its main model is aimed to trace back the origin of the power laws in Internet topologies [16]. Note that this situation does not imply that one category of models is "better" than the other. However, a unified model that considers both hierarchical properties, degree distributions and connectivity properties, and incorporates causal models has not yet been developed.

2.2 Challenges for a Universal Generation Tool

Having so many independent generation models and topology generators is disadvantageous in many respects. A researcher in need of synthetic topologies to investigate the correctness and performance of protocols and algorithms is forced to learn the nuances of many of these models/generators. Consequently she may be forced to use the most popular one, the one supported in the simulation environment used, or the easiest one. As we mentioned before, different generators produce topologies that are aimed to be used in different contexts and with different goals. For example, it does not make too much sense to use BRITE 1.0, Inet or PLRG to generate 20-node topologies for simulations. Similarly, if the AS-level connectivity properties (e.g. degree-related characteristics) are an important consideration, then using GT-ITM may not be appropriate. The result is that the researcher may end up having to use more than one generator or use one that does not offer important properties in the generated topologies (hierarchy in some cases, power-laws in

others). Analogously, for a researcher investigating the challenges of topology generation and looking for better and more powerful generation models, having so many generators available makes comparative analyses of different models significantly more difficult. For example, in order to perform a thorough comparative study one may have to learn to use GT-ITM, BRITE 1.0, Inet, PLRG and others, as well as to understand the different output formats, write different filtering routines for different output files, etc. Furthermore, if a new generation model is envisioned, a researcher has two options. Either a new generator is developed or an existing one is extended. Clearly, developing a new generator is cumbersome and available generators are not designed to be easily extended or even modified.

These difficulties are in addition to the inherently hard problems encountered when developing models that accurately capture fundamental properties of the Internet topology. Such a model is usually developed based on actual topological information that is not completely accurate. Such lack of accuracy is mainly due to the fact that mapping the Internet topology is a very challenging task [11, 23]. At the Autonomous System (AS) level, available information is richer because it can be obtained or inferred from BGP tables [17, 10]. In contrast, accurate router-level topological information is hard to obtain and until now inferring router-level connectivity has been done by using traceroute or traceroute-like probing mechanisms [11, 6]. Identifying the actual fundamental properties of topologies at the router-level is still an open research question [26]. Most Internet topology studies have approached topology modeling relying only on physical connectivity. However, routing in the Internet is determined by a policy–based routing protocol (BGP) and consequently physical connectivity does not always implies reachability. Customer–provider and peering relationships play a deciding role in determining whether or not traffic can flow between connected nodes. As an example, consider a customer AS that is connected to two provider ASs, AS–1 and AS–2. In general, a customer AS does not provide transit between its providers. So, even though there is a path from provider AS–1 to provider AS–2, they may not actually exchange traffic via the customer AS. Hence the connectivity of a topology alone does not completely characterize the structural properties of the corresponding routing topology [10]. Even if we knew the actual relationships between ASs, such relationships are continuously changing. Therefore, in order to generate accurate representative topologies the invariants of such relationships across time and size must be discovered.

In short, research in topology generation is in its infancy. New models will be developed as research will expose new and more powerful mechanisms to accurately characterize the topology of the Internet. topologies. Our challenges can be concisely put into two issues:

1. How can we develop an adapting and evolving generation tool that constitutes an interface between general Internet research and pure topology generation research? Through this interface, representative topologies developed by the topology generation research community, can be made readily available to the Internet research community at large.

2. How to design such a tool so that it also achieves the goal of facilitating pure topology generation research? A researcher that devises a new and clever generation model should be able to test it readily without having to develop a topology generator from scratch.

2.3 How to Approach the Design of a Universal Generation Tool

We address the challenges described above by establishing a differentiation between **model-oriented** (i.e. specific) topology generators, and a **universal** topology generation tool. *Model-oriented* generators are those generators designed and implemented with a specific set of models in mind. All the generators described above fall in this category. In contrast, a *universal generator* should not be tied to a specific model or set of models. Instead, this generator should be *extensible*, allowing the addition of new models in an easy way. This characteristic makes a universal topology generator *flexible* and *adaptable*, generating *representative* topologies to be used in different simulation scenarios. In addition a universal topology generator should be *robust*. It must be as *efficient* as possible without sacrificing robustness. Finally, a universal generation tool must integrate to existing network simulation tools and be user–friendly.

3 Design and Implementation of BRITE

BRITE was designed to be a flexible topology generator, not restricted to any particular way of generating topologies. As such, it supports multiple generation models. In this section we describe how this design goal was approached and how BRITE is implemented. Figure 1 depicts a schematic view of the structure of BRITE as it is being used at Boston University. The different components are labeled (1)–(4).

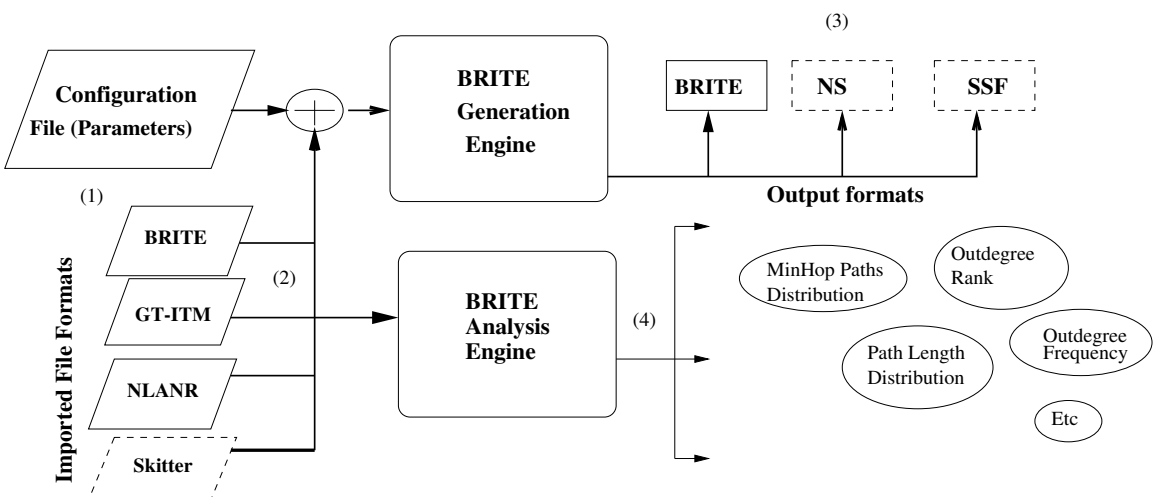

Figure 1: Schematic structure of BRITE

BRITE reads the generation parameters from a configuration file (1) that can be either hand written by the user or automatically generated by BRITE's GUI (described in Section 4). BRITE provides the capability of importing topologies (2) generated by other topology generators (GT-ITM [5], Inet [13], Tiers [8], BRITE 1.0 [16]) or topological data gathered directly from the Internet (NLANR [17], Skitter [6]). Note that we include BRITE in the "imported" file formats, because it is possible to generate topologies using BRITE and then reusing them to generate other topologies by combining them with BRITE models or other imported formats. In the current distribution BRITE produces a topology in its own file format (3), and

output capabilities for producing topologies that can be used directly by the Network Simulator (NS [18]) and the Scalable Simulation Framework (SSF [22]) simulators are currently being developed.

We developed a piece of software, separate from BRITE's generation tool, and we call it the BRITE Analysis Engine or BRIANA (4). BRIANA takes advantage of the flexible design approach of BRITE. The idea of BRIANA is to provide a set of analysis routines that may be applied to any topology that can be imported into BRITE. If we need to analyze a new topology, we just add a parsing procedure to BRITE for that new format, and once that is done, the set of analysis routines can be used on the new topology. Section 7 summarizes BRIANA's features.

3.1 BRITE's Architecture

Figure 2 depicts the main components of a topology as seen by BRITE. In BRITE, a topology is represented by a class Topology. This class contains a Model (1) and a Graph (2) as data members, and among others, a set of exporting methods and function members (3).

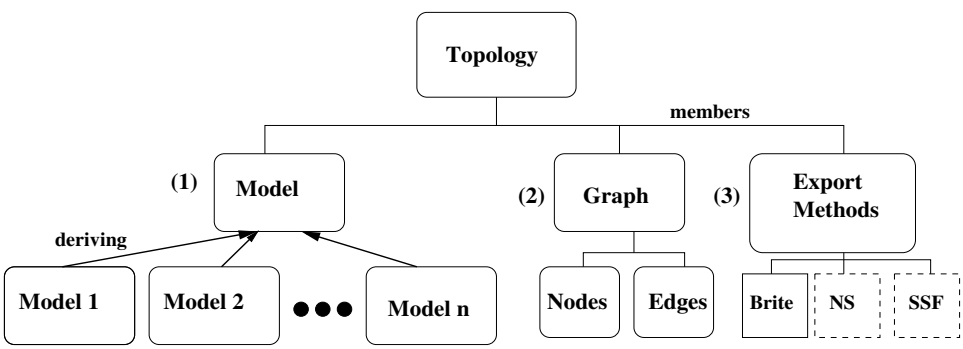

Figure 2: A Topology as seen by BRITE

The Model class is an abstract base class from which multiple specific generation models are derived. Each specific topology generated by BRITE can use a single instance of one of the available generation models if the generated topology is flat, or more than one instance if the topology is a combined hierarchical topology. (Section 3.2). The Graph data member (2) is a Graph class with the minimal functionality required by the generation models. Should more capabilities from the Graph component be required, this class may be extended or replaced with minimum effects on the remaining code. Finally, the general architecture shows a set of export methods which output BRITE topologies into specific formats.

3.2 How BRITE Works

The specific details regarding how a topology is generated depend on the specific generation model being used. We can think of the generation process as divided into a four-step process:

1. Placing the nodes in the plane

2. Interconnecting the nodes
3. Assigning attributes to topological components (delay and bandwidth for links, AS id for nodes, etc.)
4. Outputting the topology to a specific format.

This of course is not a clear-cut division that will fit every generation model but conceptually reflects what happens when a topology is being generated. Also, several models may share specific steps during the generation process, while other models differ significantly on the individual steps. In the next section we will discuss these steps in the context of particular models provided in the current distribution of BRITE.

3.3 Models

BRITE's architecture is centered around the Model class. Figure 3 depicts the current status of the model class and the specific models deriving from it.

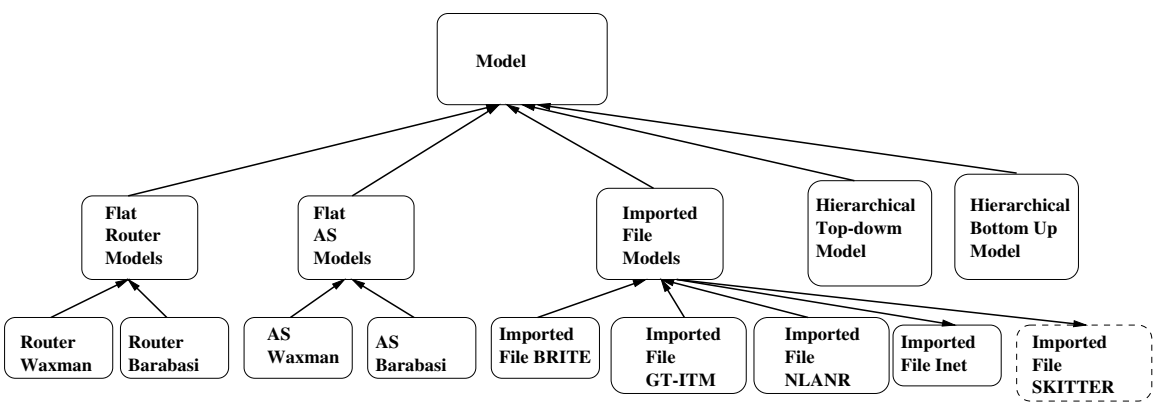

Figure 3: Model class and its deriving classes

As we can see in Figure 3, the current distribution of BRITE contains eight different generation models. Some of them are very similar and share implementation code, and others are completely different and share no functionality. Every model has a *Generate* method which returns a graph containing the generated topology. In the next subsections, we describe each of the available models.

3.4 Flat Router-level Models

BRITE contains a class *RouterModel* derived from the Model class. The idea of having such a class is to separate models that generate router-level topologies, from models specific to other environments (AS, LANs, etc). Keep in mind that the intrinsic details of any of the provided models do not represent a limitation with respect to the flexibility offered by BRITE. If none of the available individual models satisfy the requirements of a specific simulation environment, one could combine existing models or create a completely new model and integrate it into BRITE.

The router-level models currently provided with BRITE are called *RouterWaxman* (Section 3.4.3) and *RouterBarabasiAlbert* (3.4.4). These models share certain functionality. Specifically, both models place the nodes in the same way into the plane, and once the topology has been fully generated, they both assign bandwidth attributes to the links in the same way. They mainly differ in the network growth model and the node interconnection method used.

3.4.1 Placing the Nodes

BRITE separates the placement of the nodes from the process of growing the topology and interconnecting the nodes. By placing a node we mean just selecting a location in the plane for it and creating and initializing the data structures for the node in the graph. This phase does not mean that the nodes already belong to the topology because the specific joining time of a node to the topology will depend on the growth model employed.

The class *RouterModel* provides a method called *PlaceNodes* that places the nodes on the plane in one of two ways: randomly or heavy tailed. The motivation behind providing heavy-tailed distributions is explained in Appendix A. When node placement is random, each node is placed in a randomly selected location of the plane. When the placement is heavy-tailed, BRITE divides the plane into squares (the size of the plane and the size of the squares is controlled by parameters passed to BRITE. See Section 3.8). Each of these squares is assigned a number of nodes drawn from a heavy-tailed distribution. Once that value is assigned, then that many nodes are placed randomly in the square. Again, this placement mechanism can be modified or overridden by particular models.

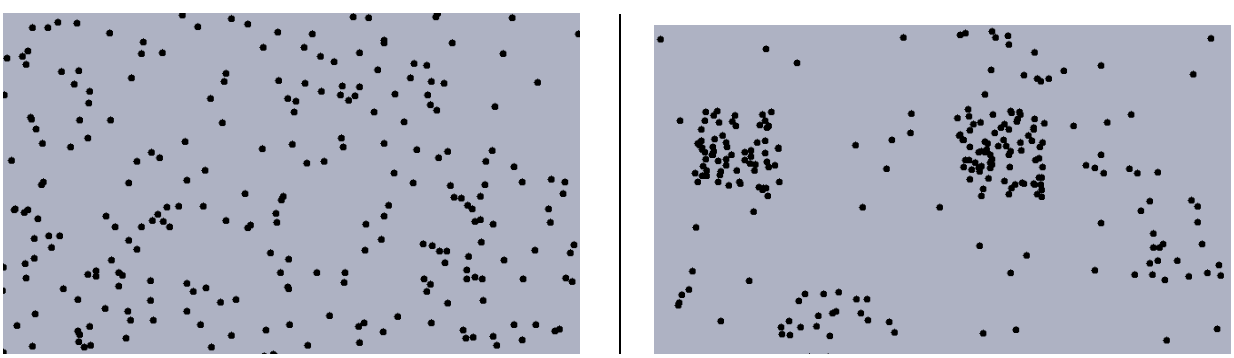

Figure 4: Snapshot of random node placement (left) and heavy-tailed node placement (right)

Figure 4 shows the difference between random and heavy-tailed node placement. The clustering provided by heavy-tailed placement can be used for specific generation models [16].

3.4.2 Assigning Bandwidths

Once the topology has been completely generated, both router-level models invoke the *AssignBandwidth* method of the *RouterModel* class. New router-level models can override this method or choose not to call it

at all.

BRITE assigns bandwidths to links according to one of four possible distributions. The user specifies in the configuration file passed to BRITE, which distribution is going to be used *($BWdist$)*, along with a minimum *($BWmin$)* and maximum *($BWmax$)* values for possible bandwidths that can be assigned. BRITE assigns a bandwdith to each link that is either:

1. **Constant**: the value specified by $BWmin$ (equal for all links in the topology).
2. **Uniform**: a value uniformly distributed between $BWmin$ and $BWmax$.
3. **Exponential**: a value exponentially distributed with mean $BWmin$.
4. **Heavy-tailed**: a value heavy-tailed distributed (Pareto with shape 1.2) with minimum value $BWmin$ and maximum value equal to $BWmax$.

Note that the user's choice of $BWdist$, $BWmin$ and $BWmax$ drives BRITE's bandwidth assignment. BRITE treats bandwidth values as unitless. Users interpret the meaning of bandwidth units according to their needs. See Section 3.8 for more details on parameter choice and values.

3.4.3 Router Waxman

RouterWaxman basically refers to a generation model for a random topology using Waxman's probability model for interconnecting the nodes of the topology, which is given by:

$$P(u, v) = \alpha\ e^{-d/(\beta L)} \tag{1}$$

where $0 < \alpha, \beta \leq 1$, d is the Euclidean distance from node u to node v, and L is the maximum distance between any two nodes.

RouterWaxman only differs from the *ASWaxman* model in that the nodes of a RouterWaxman topology represent routers rather than ASs.

3.4.4 Router BarabasiAlbert (Barabási-Albert model)

BRITE provides a *RouterBarabasiAlbert* model deriving from the Model class. It is called *RouterBarabasiAlbert* because it implements a model proposed by Barabási and Albert [2]. This model suggests two possible causes for the emergence of a power law in the frequency of outdegrees in network topologies: incremental growth and preferential connectivity. *Incremental growth* refers to growing networks that are formed by the continual addition of new nodes, and thus the gradual increase in the size of the network. *Preferential connectivity* refers to the tendency of a new node to connect to existing nodes that are highly connected or popular.

RouterBarabasiAlbert interconnects the nodes according to the incremental growth approach. When a node i joins the network, the probability that it connects to a node j already belonging to the network is given by:

$$P(i,j) = \frac{d_j}{\sum_{k \in V} d_k} \tag{2}$$

where d_j is the degree of the target node, V is the set of nodes that have joined the network and $\sum_{k \in V} d_k$ is the sum of outdegrees of all nodes that previously joined the network.

3.5 Flat AS-level Models

For the current distribution of BRITE, the provided AS-level models are very similar to the models provided for generating router-level topologies. The main difference between these router-level and AS-level models is the fact that AS models place AS nodes in the plane and these have the capability of containing associated topologies. Note that this does not mean that there are no AS-level and router-level models that differ substantially from each other. The idea of separating router-level from AS-level from the beginning is to allow for the flexibility of developing independent models for each scenario. The two AS-level models provided with the initial distribution of BRITE are **ASWaxman** and **ASBarabasiAlbert**.

3.5.1 Placing the Nodes and Assigning Bandwidths

The functionality required by the two provided AS-level models with respect to placing nodes and assigning bandwidths if similar to the router-level case. We did not combine both models for the reasons stated above. Similarly, we did not put the *PlaceNodes* method at the Model level because there are models that may not require placement of nodes in a plane at all.

3.6 Hierarchical Topologies

Generation models such as *Transit-stub* [5] and *Tiers* [8], are centered around reproducing structural properties of the Internet. In particular, Transit-stub has a well-defined hierarchy representing transit and stub autonomous systems in the Internet. Tiers is based on a three-level hierarchy of the Internet as represented by wide-area, metropolitan-area and local-area networks.

Producing synthetic topologies that possess similar structural characteristics to the Internet is important since such properties reflect how the Internet is engineered. On the other hand, achieving hierarchical similarities should not be accomplished at the expense of accuracy with respect to other properties such as degree distributions. There must be a generation model that strikes a good balance between structural properties and degree-related properties. However, it has yet to be developed.

BRITE currently supports generation of two-level hierarchical topologies. The two-level limit might be overcome by recursively generating a $2n$–level topology in n "phases". However, two-level hierarchical

topologies are in concordance to the two level routing hierarchy that has persisted in the Internet since ARPANET evolved into a network of networks interconnecting multiple autonomous systems. We plan to extend BRITE to natively support more than two levels if we find that it would allow for the generation of topologies that actually reflect real-world scenarios.

3.6.1 Top-down Hierarchical Topologies

Top-down is one of the approaches used by BRITE to generate hierarchical topologies. Figure 5 depicts the structure of the top-down approach for generating hierarchical topologies. The main steps are labeled (1)–(3).

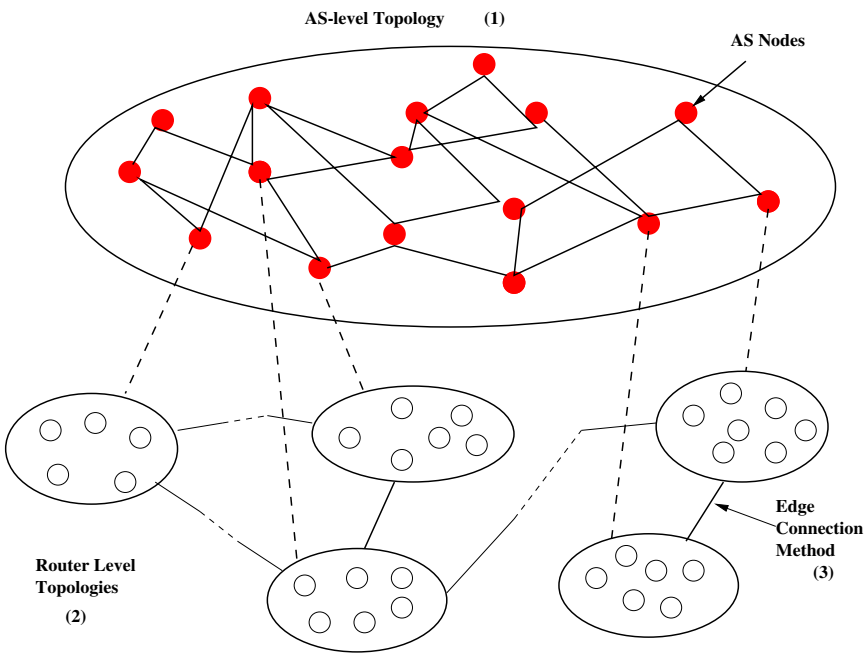

Figure 5: Top-down Approach for Generating Hierarchical Topologies

Top-down means that BRITE generates first an AS-level topology (1) according to one of the available flat AS-level models (e.g. Waxman, Imported File, etc.). Next, for each node in the AS-level topology BRITE will generate a router-level topology (2) using a different generation model from the available flat models that can be used at the router-level. BRITE uses an edge connection mechanism to interconnect router-level topologies as dictated by the connectivity of the AS-level topology. Performing this interconnection of router-level topologies in a representative way is an open research question. BRITE provides four edge connection mechanisms, borrowed from the popular GT-ITM [5] topology generator. The idea is to gradually increase the set of edge connection methods with models that reflect what actually happens in Internet topologies.

The basic edge connection methods provided with BRITE operate as follows. If (i, j) is a link in the AS-level topology, then pick a node u from the router-level topology associated with AS node i, $RT(i)$, and

a node v from the router-level topology associated with the AS node j, $RT(j)$, such that:

- **Random**: u is picked randomly from $RT(i)$ and v randomly from $RT(j)$
- **Smallest degree**: u and v are nodes with the smallest degrees in $RT(i)$ and $RT(j)$, respectively.
- **Smallest degree non-leaf**: u and v are nodes of smallest degree in $RT(i)$ and $RT(j)$ respectively but are not leaves.
- **Smallest k-degree**: u and v are nodes of degree greater that or equal to k in $RT(i)$ and $RT(j)$ respectively.

The final topology is obtained by flattening the hierarchical topology into a router-level topology composed of the individual topologies associated with each node at the AS-level.

Note that we are implicitly assuming that the topology being manipulated is represented by an undirected graph. BRITE internal data structures do not restrict the graph associated with a topology to be undirected. However, some of BRITE functionality, such as importing a topology from a file in BRITE's output format, assumes that the corresponding topology has undirected links. Increasing BRITE's flexibility by explicitly distinguishing between directed and undirected topologies is straightforward and it is part of our ongoing work.

The configuration file used by BRITE to generate a top-down topology contains parameters controlling the bandwidth distribution for inter- and intra-domain links. These parameters override the specific parameters for the AS- and router-level topologies. Bandwidths for the generated AS-level topology will be assigned according to the inter-domain distribution. Furthermore, bandwidths for each generated router-level topology are assigned according to the intra-domain distribution. During the flattening process, the links established between different router-level topologies will have assigned the bandwidth associated with the corresponding AS-AS link. This bandwidth–assignment method represents just one possible mechanism. Different models can be implemented and added to BRITE.

3.6.2 Bottom-up Hierarchical Topologies

Another viable approach to generate hierarchical topologies is the bottom-up approach. The accuracy of this approach with respect to generating representative Internet topologies has yet to be validated. Nonetheless, we believe it is an alternative mechanism to generate hierarchical topologies. The interesting question to be answered with this approach is: how can we infer topological characteristics at the AS-level from known topological information at the router-level. BRITE provides a model that generates hierarchical topologies following this approach.

In this model, BRITE first generates a router-level topology using any of the available models (router Waxman, imported file, etc.). Once this topology has been constructed, BRITE assigns to each AS node (level-2 node) a number of routers according to an assignment type specified by the user. With this number of assigned routers to an AS node, BRITE groups that many nodes from the router topology following a

grouping method specified also by the user as a parameter to BRITE. The next two subsections describe the assignment types and grouping mechanisms provided as a base bottom-up model by BRITE.

Assignment Types: The set of parameters associated with the Bottom-up model include $NumAS$, the number of ASs requested by the user. BRITE then assigns router-nodes to ASs in one of the following ways:

- **Constant:** Assign each AS an equal number of router-level nodes, i.e. $NumNodes/NumAS$, where $NumNodes$ is the number of nodes in the router-level topology.
- **Uniform:** Pick a value uniformly distributed in $[1, NumNodes]$.
- **Exponential:** Pick a value exponentially distributed with mean $\frac{NumNodes}{NumAS}$.
- **Heavy-tailed:** Pick a value from a truncated heavy-tailed distribution between 1 and $NumNodes$.

Grouping Mechanisms. Once the number of nodes for an AS has been assigned, BRITE assigns this number of router nodes to a single AS. BRITE does this step in one of two ways:

- **Random pick:** Pick randomly one node at a time and assign it to AS i until it reaches its size. Repeat for all ASs.
- **Random walk:** Perform a random walk through the graph, where each step in the walk corresponds to choosing a random neighbor from a given vertex. Each visited node is assigned AS i until it reaches its size. Repeat for all ASs.

The random walk mechanism is slightly more complicated than random pick for two reasons. First, we need to keep track of which nodes has been visited (assigned). After finishing with a given AS, BRITE picks a not-yet-visited router node and starts the random walk for the next AS. Second, it can happen that a random walk can not be completed for an intermediate AS (not the first one) because the random-walk algorithm cannot continue walking the graph any further through not-yet-visited router nodes.[1] In this approach the parameter $NumAS$ provided by the user is used as a guideline for BRITE. As many nodes as possible, up to the value assigned to the corresponding AS, will be picked from the set of nodes during a random walk. Certain number of ASs may get less nodes than initially assigned to them. BRITE reports the actual number of ASs assigned, which may exceed the parameter $NumAS$.

We emphasize that the Bottom-Up approach is an experimental model for generating hierarchical topologies. Providing several ways to group nodes into ASs is aimed at facilitating the process of experimentation. However, there is no analysis supporting the use of random pick over random-walk or vice-versa. One could implement an assignment mechanism of routers to ASs that mimics the assignment procedure proposed in [12] to compute an AS overlay on top of a measured router-level topology. This and other assignment/grouping mechanisms is the subject of ongoing research.

[1] Such a random walk is called self-avoiding.

3.7 Imported File Model

As we mentioned before, one of the design goals of BRITE was to combine strengths of available models into a single tool. BRITE incorporates an *ImportedFileModel* class deriving from the base abstract class Model. Figure 6 shows the current structure of the ImportedFileModel.

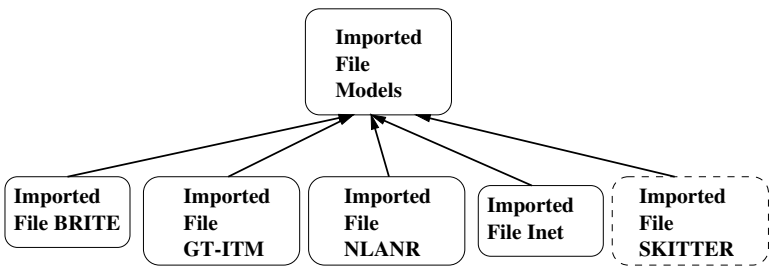

Figure 6: Imported File Model: parsing topologies in other formats

The *Generate* method associated with a model derived from ImportedFileModel will be in charge of parsing a file in the format of the corresponding imported topology, and it will load it into BRITE Graph data structures. Once such a topology has been parsed and loaded, it can be used as a native BRITE topology. We have applied this approach to combine topologies from existing generators with topologies generated with a variety of other models. There are many useful scenarios where a researcher may benefit from having such a capability. For example, we could generate a top-down hierarchical topology, where the AS-level topology has been imported from NLANR topological data, and the router-level corresponds to Waxman topologies or topologies generated by the GT-ITM generator. It is also possible to generate hierarchical topologies with more than two levels. For example, a four-level topology may be obtained by recursively generating a top-down hierarchical topology where the AS-level corresponds to a Transit-Stub topology generated using GT-ITM and the router level topologies correspond to a top-down topology generated using BRITE. Thus, BRITE's architecture allows a researcher to combine topologies incorporating diverse research themes, as well as to create new models specific to certain scenarios.

The available models in this category are:

- Imported BRITE Topology
- Imported GT-ITM Topology (flat and Transit-Stub)
- Imported NLANR Topology
- Imported Inet Topology

We are currently working on importing topological data from the CAIDA project, such as the data gathered by Skitter [6].

3.8 BRITE's Parameters

Tables 1, 2, 3 and 4 describe the meaning of the parameters of the models provided with the current distribution of BRITE. Parameters used by more that one model, are mentioned only once.

Parameter	Meaning	Values
`HS`	Size of one side of the plane	int ≥ 1
`LS`	Size of one side of a high-level square	int ≥ 1
`N`	Number of nodes	int $1 \leq N \leq HS * HS$
`Model`	model id	int ≥ 1
`alpha`	Waxman-specific exponent	$0 < \alpha \leq 1, \alpha \in R$
`beta`	Waxman-specific exponent	$0 < \beta \leq 1, \beta \in R$
`Node Placement`	how nodes are placed in the plane	1: `Random`, 2: `HT`
`m`	Number of links per new node	int ≥ 1
`Growth Type`	how nodes join the topology	1: `Incremental`, 2: `Random`
`BWdist`	bandwidth assignment to links	1: `Const`, 2: `Unif`, 3: `Exp`, 4: `HT`
`MaxBW, MinBW`	min, max link bandwidth values	float > 0

Table 1: Flat Topology (AS Only or Router Only) Parameters.

Parameter	Meaning	Values
`Edge Connection`	method for interconnecting router topologies	1: `Random node` 2: `Smallest degree`, 3: `Smallest degree non-leaf` 4: `k-Degree`
`Intra BWdist`	Intra-domain bandwidth assignment distribution	1: `Constant` 2: `Uniform` 3: `Exponential` 4: `Heavy-tailed`
`Intra BWMax/Min`	min, max bandwidth values for intra-domain links	$float > 0$
`Inter BWdist`	inter-domain bandwidth assignment distribution	1: `Constant` 2: `Uniform` 3: `Exponential` 4: `Heavy-tailed`
`Inter BWMax/Min`	min, max bandwidth values for inter-domain links	$float > 0$

Table 2: Top-down Hierarchical topology Parameters.

3.8.1 Parsing Support

The C++ version of BRITE provides its own parsing routines. When building a parsing routine for the parameters of a new model, or for parsing a new Imported File Model, the user can benefit from the functionality provided by the parsing routines (Parser.h) summarized in Appendix B. The Java version uses the functionality of the *StreamTokenizer* class found in *java.io.StreamTokenizer*.

Parameter	Meaning	Values
`AssignType`	assignment of routers to ASs	1: Constant 2: Uniform 3: Exponential 4: Heavy-tailed
`NumAS`	number of ASs "desired"	$int > 0$
`Grouping Method`	how routers are grouped into ASs	1: `Random pick` 2: `Random Walk`
`Inter BWdist`	inter-domain bandwidth assignment distribution	1: `Constant` 2: `Uniform` 3: `Exponential` 4: `Heavy-tailed`
`Inter BWMax/Min`	min, max link bandwidth values	float > 0

Table 3: Bottom-Up topology Parameters.

Parameter	Meaning	Values
`Format`	File format to be imported	1: `BRITE` 2: `GT-ITM` 3: `NLANR` 4: `SKITTER` 5: `GT-ITM Transit-Stub` 6: `Inet`
`File`	file pathname	valid file name
`HS, LS`	Plane dimensions	same as above
`BWdist`	Bandwidth distribution	1: `Constant` 2: `Uniform` 3: `Exponential` 4: `HT`
`BWmin, BWmax`	min, max link bandwidth values	float > 0

Table 4: Imported File topology Parameters.

4 BRITE Interface Design and Implementation

BRITE has two input interfaces: a Java based graphical interface and a command line interface. All topology generation parameters are specified in a configuration file, which can either be generated automatically by the GUI or manually created.

At present, BRITE has only a single output interface: a BRITE format topology file. Visualization output and export to simulation environments such as *NS* and *SSF* are in development.

In the following sections, we discuss the input interfaces of BRITE: the graphical interface, the command line interface and finally the BRITE configuration file format. Next, we describe the output interface of BRITE: the BRITE topology file format.

4.1 BRITE's GUI

One of our design goals was to make BRITE easy to learn and use. To this end, we built a cross platform GUI front-end to the BRITE generation engine. Although the graphical interface is written in Java, it can be used as a front-end to the C++ version without any performance loss. This is achieved by divorcing the GUI from the topology generation engine. All interaction between the GUI and the generation engine takes place through a configuration file. Once a user specifies desired parameters, the GUI generates a corresponding configuration file, *GUI_GEN.conf*. It then spawns a new C++ or Java process and runs the generation engine with this configuration file as input. Figure 7 illustrates this interaction between the GUI and the generation engine.

4.2 The command line interface

The command line interface directly invokes the BRITE generation engine. It must receive as input a *configuration file*, a location for an *export file*, and a *seed file*. BRITE uses pseudo-random numbers at different points during the generation process. For example, nodes are placed randomly in the plane, nodes are interconnected according to certain probability function, etc. The seed_file contains seeds to initialize the pseudo-random number generator independently each time it is required. BRITE updates the seed file with new seeds in such a way that it can be used to generate independent topologies on subsequent runs. Also, the seed file for the current run is saved into a backup file named last_seed_file to allow reproducing topologies in different runs.

To use the command line interface, go to the BRITE directory and at the command prompt, type:

```
$ brite my-config.conf my_export.brite seed-file              (C++)
$ java Main.BRITE my_config.conf my_export.brite seed-file   (Java)
```

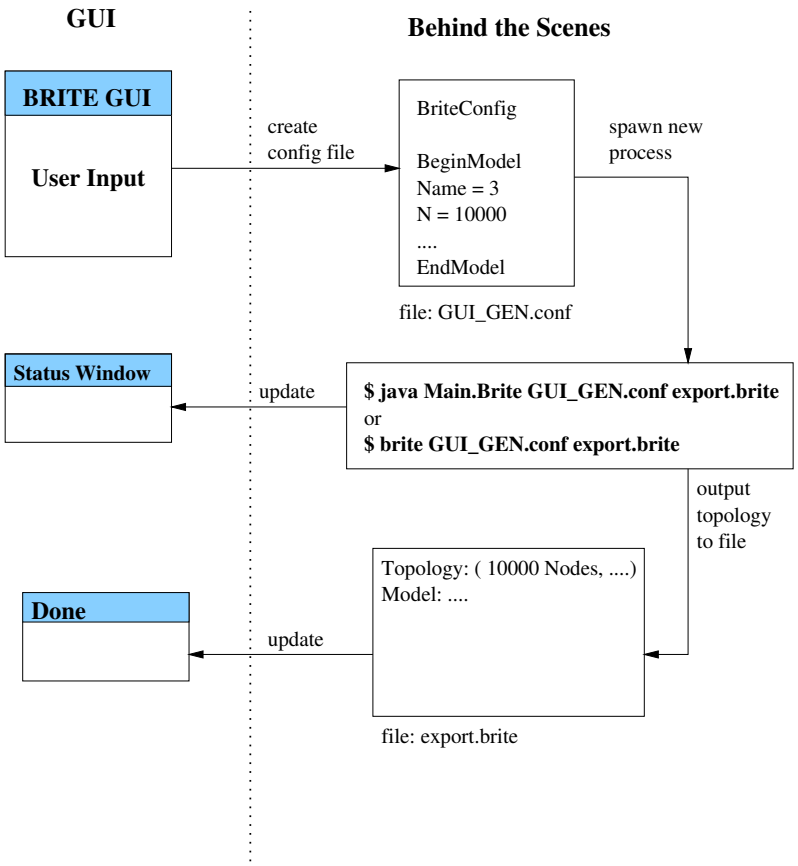

Figure 7: Interaction between GUI and BRITE Generation Engine

4.3 Configuration Files

All BRITE configuration files must begin with the *BriteConfig* keyword. Listing 4.1 shows a configuration file for generating an AS-level topology using the Waxman model. A BRITE configuration file is a simple *Key-Value* text file. A *Key* is an alpha-numeric string and a *Value* must be a numeric value. The *Key* and *Value* must be separated by a '=' character. The *BeginModel* and *EndModel* keywords are delimiters for each single–level model parameters. The '#' character serves as a comment character. The following is a sample of a single–level *ASWaxman* configuration file.

The Key-Value parameters are unique for each model. Suppose, for instance that we were interested in importing an NLANR file as a BRITE AS-level topology. Listing 4.2 shows a configuration file that achieves this.

A number of sample configuration files are included in both versions of BRITE in the *conf_files/* directory.

Listing 4.1 Configuration file for AS Waxman Model

```
BriteConfig

BeginModel
   Name =  3           #Router Waxman = 1, AS Waxman = 3
   N = 10000           #Number of nodes in graph
   HS = 1000           #Size of main plane
   LS = 100            #Size of inner planes
   NodePlacement = 2   #Random = 1, Heavy Tailed = 2
   GrowthType = 2      #Incremental = 1, All = 2
   m = 2               #Number of neighboring nodes for each new node
   alpha = 0.19        #Waxman Parameter
   beta = 0.2          #Waxman Parameter
   BWDist = 2          #Const = 1, Uniform = 2, HT = 3, Exp = 4
   BWMin = 10.0
   BWMax = 1024.0
EndModel

BeginOutput
   Brite = 1           # 0/1: Save/Don't save in BRITE's format
   Otter = 0           # 0/1: Save/Don't save in Otter's format
EndOuput
```

Listing 4.2 Configuration file for importing an NLANR (ASConnList) topology

```
BriteConfig

BeginModel
  Name =  8             #AS File = 8, Router File = 7
  Format = 3            #BRITE = 1, GT-ITM = 2, NLANR = 3, SKITTER = 4, GT-ITM(TS) = 5
  File = 19980101.nlanr #NLANR (ASConnlist) data
  HS = 1000             #Size of main plane
  LS = 100              #Size of inner planes
  BWDist = 1            #Constant = 1, Uniform = 2, HeavyTailed = 3, Exponential = 4
  BWMin = 10.0          #Min bandwidth value
  BWMax = 1024.0        #Max bandwidth value
EndModel

BeginOutput
   Brite = 1           # 0/1: Save/Don't save in BRITE's format
   Otter = 0           # 0/1: Save/Don't save in Otter's format
EndOuput
```

4.4 The BRITE Output Format

A BRITE-formatted output file contains three sections:

1. Model information: information about the topology contained in the file. Includes number of nodes and edges, and information specific to the model used to generate the topology.
2. Nodes: for each node in the graph, a line is written into the output file with the following format :

   NodeId xpos ypos indegree outgdegree ASid type

   The meaning of each field is described in Table 5.

Field	Meaning
`NodeId`	Unique id for each node
`xpos`	x-axis coordinate in the plane
`ypos`	y-axis coordinate in the plane
`indegree`	Indegree of the node
`outdegree`	Outdegree of the node
`ASid`	id of the AS this node belongs to (if hierarchical)
`type`	Type assigned to the node (e.g. router, AS)

Table 5: BRITE Output format: Nodes section

3. Edges: for each edge in the graph, a line with the following format is written in the output file:

   EdgeId from to length delay bandwidth ASfrom ASto type

   The meaning of each field is described in Table 6.

Field	Meaning
`EdgeId`	Unique id for each edge
`from`	node id of source
`to`	node id of destination
`length`	Euclidean length
`delay`	propagation delay
`bandwidth`	bandwidth (assigned by *AssignBW* method)
`ASfrom`	if hierarchical topology, AS id of source node
`ASto`	if hierarchical topology, AS id of destination node
`type`	Type assigned to the edge by classification routine

Table 6: BRITE Output format: Edges section

As an example, the output file shown in Listing 4.3 corresponds to a *Waxman* topology of 5 nodes and 8 edges, which was generated using the model *RouterWaxman* (Model number 1). Since it is a flat router-level

topology, nodes do not have a corresponding AS id and therefore the **ASid** field contains -1. For the same reason, the **ASfrom** and **ASto** fields for the edges contain -1 values. The final field for both nodes and edges, correspond to a type assigned to them by a classification routine. BRITE provides a classification routine based on a classification method proposed in [3]. The classification routine is not called by default and hence both nodes and edges are unclassified (NONE type).

Listing 4.3 Output format: Flat router-level topology, 5 nodes, 8 edges

```
Topology: ( 5 Nodes, 8 Edges )
Model ( 1 ): 5 1000 100 1 1 2 0.15 0.2 1 10 1024

Nodes: (5)
0 216.00 663.00 3 3 -1 RT_NONE
1 347.00 333.00 3 3 -1 RT_NONE
2 384.00 926.00 3 3 -1 RT_NONE
3 27.00 309.00  4 4 -1 RT_NONE
4 212.00 187.00 3 3 -1 RT_NONE

Edges: (8):
0 2 0 312.08 1.04 10.00 -1 -1 E_RT_NONE
1 2 1 594.15 1.98 10.00 -1 -1 E_RT_NONE
2 3 1 320.90 1.07 10.00 -1 -1 E_RT_NONE
3 3 2 712.84 2.38 10.00 -1 -1 E_RT_NONE
4 4 0 476.02 1.59 10.00 -1 -1 E_RT_NONE
5 4 3 221.61 0.74 10.00 -1 -1 E_RT_NONE
6 0 3 401.29 1.34 10.00 -1 -1 E_RT_NONE
7 1 4 198.85 0.66 10.00 -1 -1 E_RT_NONE
```

5 BRITE Topology Generation Process Walk-through

We now provide step-by-step instructions on creating a topology using the BRITE GUI.

The first step is to start the GUI by typing the following command at the prompt:

```
\$ startGUI
```

A window as shown in Figure 8 will pop up.

To create a new Topology, select in **Topology Type** the kind of topology that you'd like to create first. You may select one of the options shown in Figure 9, namely, *Flat Router–Level* only, *Flat AS–Level* only or Hierarchical topologies. Although the interface shows the parameters available for other models as well, only those applicable to the selected model type will be enabled.

Next, specify the topology parameters. A complete description of these parameters is given in Section 3.8. BRITE comes with two models: *Waxman* and BarabásiAlbert for both AS-level and router-level topolo-

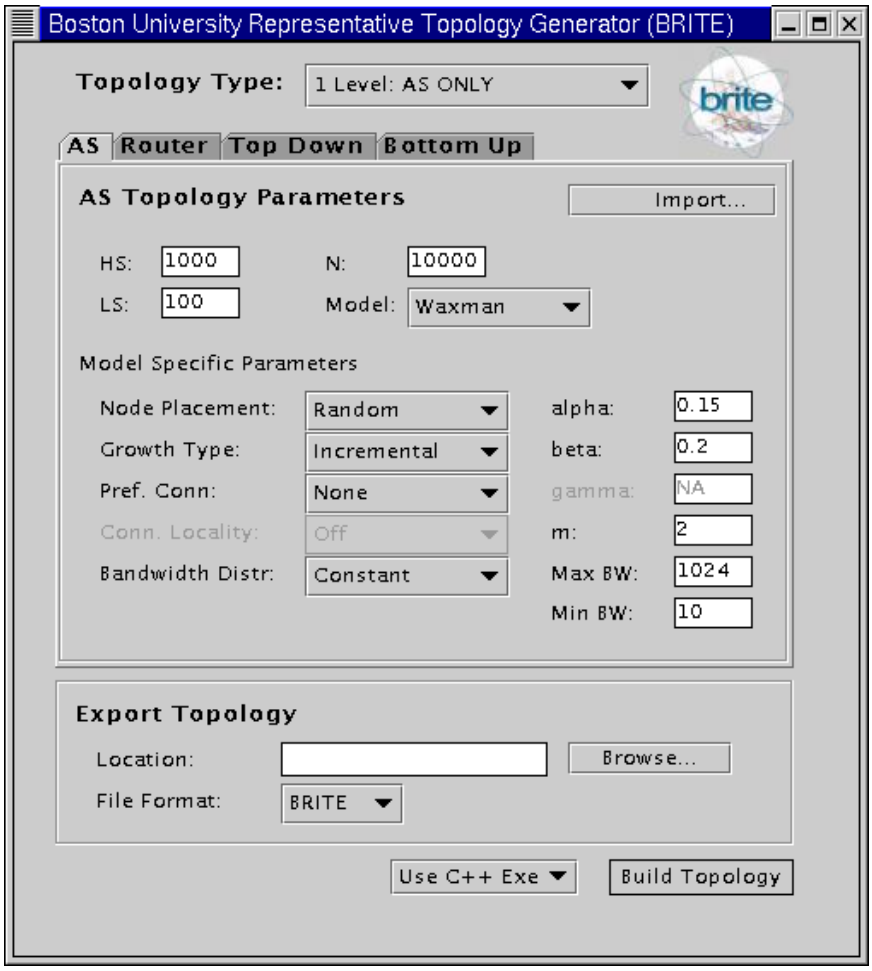

Figure 8: Main window of BRITE's GUI

gies. You may select one of these. You may also import an existing topology file and use it in combination with other topologies in a Hierarchical Model. To do this click on the **Import** button and a window will pop up from which you will be able to select the desired file to import.

Once you have selected a model, you may edit the default model–specific parameters. In Figure 8 the selected topology type is flat AS-level and the parameters in the middle section of the window are parameters passed to the ASWaxman model.

In the **Export file** section of the window, specify the file and format you want to export the generated topology to. In the current distribution of BRITE only BRITE's output format is supported but soon there will be exporting capabilities for *NS* and *SSF* formats. See Section 4.3 for details on BRITE's output format.

Finally, select the version of the BRITE generation engine to use: C++ or Java. Click Build Topology. A status window, as shown in Figure 10, should appear that will detail the topology generation process. You may at a later time modify the configuration file that the GUI generates, *GUI_GEN.conf*, and generate similar topologies.

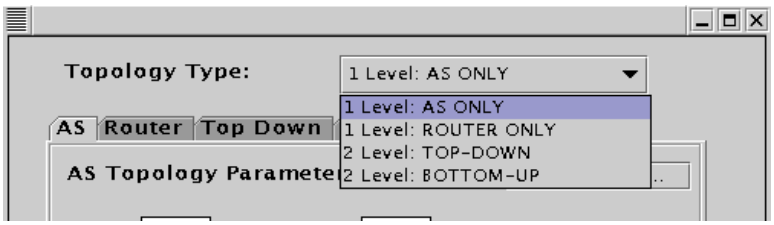

Figure 9: Topology types available

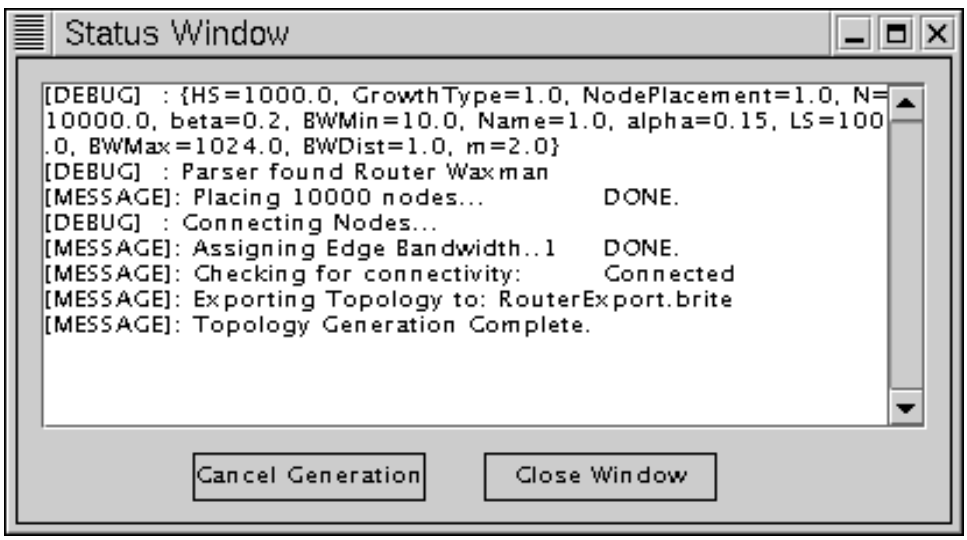

Figure 10: Status window showing generation progress

6 Extending BRITE

One design goal of BRITE was to easily allow researchers to incorporate new models for specific scenarios. In this section we describe with a reasonable level of detail, the necessary steps that must be taken in order to extend BRITE by incorporating new generation models to its framework. Since there are two very similar implementations of BRITE (Java and C++), we present a separate section with extension guidelines for each version.

6.1 Extending BRITE's C++ Version

We have to worry mainly about two things regarding a new model: passing parameters to BRITE required by the new model, and implementing the generation functionality that's specific to the new model.

6.1.1 Passing parameters to the new Model

Passing of parameters is done in BRITE via a configuration file. This file must be parsed and its contents must be put in a parameter data structure associated with the specific model we are implementing. BRITE provides a base class called **ModelPar** which is the base class for parameter classes for the models provided with BRITE.

For example, C++ excerpt 1 shows the declaration of the *RouterWaxPar* class which is the parameter class associated to the Router Waxman model. This class contains data members for all parameters required by the Router Waxman model.

C++ excerpt 1 Class associated to router-level Waxman Model

```
class RouterWaxPar : public ModelPar{

public:

  RouterWaxPar(int n, int hs, int ls, int np,
       int ig, int m_edges, double a, double b,
       int bw, double bw_min, double bw_max);

  int GetN() { return N; }
  int GetNP() { return NP; }
  int GetIG() { return IG; }
  int GetM() { return m; }
  double GetA() { return alpha; }
  double GetB() { return beta; }
  int GetBW() { return BW; }
  double GetBWMin() { return BWmin; }
  double GetBWMax() { return BWmax; }
  void SetBW(int bw) { BW = bw; }
  void SetBWMin(double bw_min) { BWmin = bw_min; }
  void SetBWMax(double bw_max) { BWmax = bw_max; }

private:
  int N;              // Size
  int NP;             // Node placement strategy
  int IG;             // Growth type
  int m;              // Number of edges per newly added node
  double alpha, beta; // Waxman parameters
  int BW;             // Bandwidth distribution
  double BWmin;
  double BWmax;

};
```

There is a single entry point to the parsing of configuration files, which is the method:

```
ModelPar* Parse::ParseConfigFile();
```

The method *ParseConfigFile*, which is invoked from BRITE's main routine, will determine the type of model that must be parsed and then it will invoke the specific method in charge of parsing parameters for the corresponding model. As an example, let's suppose that *ParseConfigFile* is parsing a configuration file for generating a Waxman topology. Listing 6.1 shows such a configuration file.

Listing 6.1 Configuration file for Router Waxman Model

```
BriteConfig

BeginModel
   Name =  1            #Router Waxman = 1, AS Waxman = 3
   N = 100              #Number of nodes in graph
   HS = 1000            #Size of main plane (number of squares)
   LS = 100             #Size of inner planes (number of squares)
   NodePlacement = 1    #Random = 1, Heavy Tailed = 2
   GrowthType = 1       #Incremental = 1, All = 2
   m = 2                #Number of neighboring node per each new node.
   alpha = 0.15         #Waxman Parameter
   beta = 0.2           #Waxman Parameter
   BWDist = 1           #Const = 1, Uniform =2, HT = 3, Exp =4
   BWMin = 10.0
   BWMax = 1024.0
EndModel

BeginOutput
   Brite = 1            # 0/1: Save/Don't save in BRITE's format
   Otter = 0            # 0/1: Save/Don't save in Otter's format
EndOuput
```

Upon parsing the field **Name = 1**, *ParseConfigFile* will determine that it must parse parameters for the *RouterWaxman* model and will invoke the appropriate parsing procedure, *ParseRouterWaxman* in this case. Excerpt 2 shows this logic.

ParseRouterWaxman, shown in code excerpt 3, will parse the parameters associated with the Waxman model.

At this point, *ParseConfigFile* returns to BRITE's main routine the data structure with the parameters just parsed. Next, BRITE instantiates an object of the model class we are creating, and uses it to create the corresponding topology. Code excerpt 4 shows the logic for the last step.

In summary with respect to parameter passing, when we are adding a new model we must create a parameter class, *NewModelPar*, which will contain the parameters for the new model. In addition, we must provide parsing procedure such as *ParseRouterWaxman* for our new model. Finally, we have to modify the following parts of the code:

C++ excerpt 2 Parsing logic for a RouterWaxman configuration file

```
    ...
  /* Name for first model */
  ParseIntField("Name", model);

  switch (model) {
  case RT_WAXMAN:
    rt_wax_par = ParseRouterWaxman();
    return rt_wax_par;
    break;
    ...
```

C++ excerpt 3 Parsing routine for Router Waxman model

```
RouterWaxPar* Parse::ParseRouterWaxman() {

     ...
  RouterWaxPar* rt_wax_par;
  ParseIntField("N", n);
  ParseIntField("HS", hs);
  ParseIntField("LS", ls);
  ParseIntField("NodePlacement", np);
  ParseIntField("GrowthType", ig);
  ParseIntField("m", m);
  ParseDoubleField("alpha", a);
  ParseDoubleField("beta", b);
  ParseIntField("BWDist", bw);
  ParseDoubleField("BWMin", bw_min);
  ParseDoubleField("BWMax", bw_max);
  ParseStringField("EndModel");
     ...

  rt_wax_par = new RouterWaxPar(...);

  return rt_wax_par;
}
```

C++ excerpt 4 Parsing routine for Router Waxman model

```
    ...
  /* Parse configuration file */
  par = p.ParseConfigFile();

  switch (par->GetModelType()) {
  case RT_WAXMAN:
    rt_wax_model = new RouterWaxman((RouterWaxPar*)par);
    topology = new Topology(rt_wax_model);
    break;
    ...
}
```

1. Add a new model id to the enum type *ModelType* in **Model.h**
2. Add the code for the new parsing routine in **Parser.cc**
3. Modify *ParseConfigFile* in *Parser.cc* to check for the new model type and invoke the new parsing routine.
4. Modify BRITE's main routine in *BriteMain.cc* to check for the type of the new model and invoke the constructor for the new model class.

6.1.2 Generation Functionality

Extending the functionality is done basically by adding a new model class to BRITE. As described in Section 3.3, BRITE architecture's is centered around the Model class (see Figure 3). If we wanted to add a new router-level model to BRITE, we could create a new class, *NewRouterModel* deriving from RouterModel. Code excerpt 5 shows the declaration of such a class.

Note that the *NewRouterModel* class contains two public methods that are a fundamental part of its interface: **Generate** and **toString**. Every model in BRITE, including the models for imported files, have a *Generate* method which is in charge of generating the graph for the constructed topology according to the model. Also, the *toString* method is used when exporting a BRITE topology into a file.

In BRITE, a new topology is generated by instantiating an object of the class Topology. The constructor for the class Topology receives a pointer to a Model as an argument. Next, the Topology constructor populates its model member with the received pointer, and populates its Graph member by invoking the *Generate* method of the received model. Once this is done, the topology has been created. Code excerpt 6 from file *Topology.cc*, shows the actions of the Topology constructor.

The *Generate* method of every new model should return a pointer to a Graph, containing the topology created. As an example, let's take a look at the code excerpt 7 which shows the *Generate* method of the *RouterWaxmanModel* class, taken from the file *RouterWaxmanModel.cc*.

C++ excerpt 5 Model class for new router-level model

```
class NewRouterModel : public RouterModel {

 public:

  NewRouter(NewRouterModelPar* par);
  ~NewRouterModel() { }
  Graph* Generate();
  string ToString();
  ...

 protected:

  void InterconnectNodes(Graph *g);
  ...

 private:

  /* private data/function members */
  double ProbFunc(Node* src, Node* dst);
  ...
};
```

C++ excerpt 6 Topology class constructor

```
   Topology::Topology(Model* model)
   {
      m = model;
      g = m->Generate();
   }
```

C++ excerpt 7 Generate method for Router Waxman model

```
Graph* RouterWaxman::Generate() {

  Graph* graph;

  graph = new Graph(size);

  /* Place nodes into plane */
  PlaceNodes(graph);

  /* Build topology graph using Waxman */
  InterconnectNodes(graph);

  /* Assign bandwidths to edges */
  AssignBW(graph);

  return graph;
}
```

As we can see, this **Generate** method performs the required steps to generate the topology. The first step is to place the nodes into the plane. The specific functionality associated with model **RouterWaxman** is in particular encapsulated inside the method *InterconnectNodes*. Finally, the *AssignBW* method, which in this case is not overridden by the *RouterWaxman* class, is invoked to assign bandwidths to the links according to certain bandwidth distribution. Thus, for the new model, we must provide a *Generate* method with the appropriate functionality to produce topologies according to the new model.

6.2 Extending BRITE's Java Version

The Java version of BRITE is very similar to the C++ version in design. However, there are few implementation differences between the two versions. This section details the process of adding your own generation models to the Java version of BRITE.

Like the C++ version, adding a new model in the Java version requires: 1) Extending the parameter passing method to handle parameters that are unique to the new model, and 2) implementing the generation functionality of this model.

6.2.1 Passing Parameters to the new model

All parameter passing is done in BRITE via a configuration file. As such, creating your own model first requires modifying the configuration file parser (located at *Main/ParseConfFile.java*) to handle the parameters of your model. You do not need to write your own parsing routines. The *Parse* method of the configuration file parser stores all parsed attribute-value pairs of a model in a Hash-table, *params*. All you need to do is

interpret the keys and values in this Hash-table according to the semantics of your model, create your model and return it to the caller. For instance, the RouterWaxman interprets the Hash-table parameters as shown in code excerpt 8.

Java excerpt 8 Parsing routine for RouterWaxman model

```
private static Model ParseWaxman(Hashtable params) {

    ...
    int N =  params.get("N");
    int HS = params.get("HS");
    int LS = params.get("LS");
    int np = params.get("NodePlacement");
    int m =  params.get("m");
    int bwDist = params.get("BWDist");
    float bwMax = params.get("BWMax");
    float  bwMin = params.get("BWMin");
    ...

    return new  RouterWaxman(N, HS, LS, ...);

}
```

The *Model* object is then returned to the main caller (usually the main entry point to BRITE, *Main/Main.java*), which in turn creates a Topology instance with this newly created model. See code excerpt 9 extracted from *Main/Main.java*.

Java excerpt 9 Model-parameter parsing and Topology instantiation logic

```
//create a model according to the parameters in the BriteConfig file:
Model m = ParseConfFile.Parse(filename);

//now create the topology with this model:
Topology t = new Topology(m);
```

6.2.2 Overriding the Generate method

Like in the C++ version, the *Generate* method is the entry point to creating a topology Graph for all models. Generation of flat topologies is usually a three step process: 1) Placing the nodes on a plane, 2) adding edges between the nodes, and 3) assigning weights to these edges (representing bandwidth, delay and so forth). Code excerpt 10, shows as an example, the *Generate* method for the *RouterWaxman* model.

The parent class for *RouterWaxman*, *RouterModel* implements the node placement functionality and is used unmodified by RouterWaxman. You may use the functionality provided in the existing models by

Java excerpt 10 Generate method for Router Waxman model

```
public Graph Generate() {

  //create a new graph of size N:
  Graph g = new UndirectedGraph(N);

  //1) Place router nodes on our plane:
  PlaceNodes(g, ModelConstants.ROUTER_NODE);

  //2) Connect Edges following Waxman:
  ConnectNodes(g);

  //3) Assign bandwidth to our edges:
  AssignBW(g.getEdgesArray());

  return g;
}
```

deriving your model from one (or more) of these models and overriding unwanted methods.

6.3 An example of adding a new model to BRITE

One interesting topology generator that was recently made publicly available is Inet [13] developed at the University of Michigan at Ann Arbor.

Let's imagine a scenario in which BRITE is used as a framework to develop a new topology generation model. We illustrate this scenario by incorporating a simplified version of Inet's generation model into BRITE's framework. We provide the guidelines and pseudo-code in the context of the C++ version. Adding a new model to the Java version is similar. The actual Inet generator is much more complex than what we are conveying in this example but its actual interface is concise and clean which facilitates its use as an example.

As was explained in Section 6.1, we need to consider mainly two issues: the passing of parameters and the generation functionality.

Inet basically receives the following parameters (we ignore the rest):

1. n: topology size
2. d: fraction of nodes of degree one
3. p: plane size

Therefore we create a ToyInetPar class deriving from *ModelPar*, as shown in code excerpt 11.

C++ excerpt 11 ToyInet model Parameter structure

```
class ToyInetPar : public ModelPar{

public:

  ToyInetPar(int n, double d, int p);

  int GetN() { return n; }
  int GetD() { return d; }
  int GetP() { return p; }

private:
  int n;             // size
  int d;             // fraction of degree-one nodes
  int p;             // plane size

};
```

BRITE will be instructed to generate a topology according to the new ToyInet model, by specifying the model type along with the required parameters in a configuration file as shown in Listing 6.2.

The next step is to provide a parsing routine specific for the Inet model. Such a parsing routine, built using the basic parsing facilities of BRITE, is shown in code excerpt 12.

Now we have the parameter passing done. The next step is to add the generation functionality according to the Inet model. We can do that by creating a new *InetModel class* derived directly from the base Model class and overriding its Generate method. This is shown in code excerpt 13.

7 BRIANA: The BRITE Analysis Engine

The BRITE Analysis Engine, BRIANA, is a piece of software developed at Boston University which is fully integrated with BRITE but that at the same time can be used independently. The purpose of BRIANA is to serve as a repository of routines for analysis of topological data. BRIANA takes advantage of the functionality implemented in BRITE's generation tool by reusing a fair amount of BRITE's code. If a significant change needs to be done to one of the parsing routines for the ImportedFileModel, a single change will update both BRITE and BRIANA simultaneously. On the other hand, this synchronization is nice but not required. BRIANA can be compiled and run completely independently of BRITE, that is, it is not needed to generate a topology or import it using BRITE in order to analyze it.

The command line execution for BRIANA is, with respect to the set of analysis routines implemented at the time of this writing, is shown in Listing 7.1.

Listing 6.2 Configuration file for ToyInet model

```
#
# Configuration file for ToyInet model
#

BriteConfig

BeginModel
  Name = 8      # new model id (Model.h)
  N = 10000     # topology size
  D = 0.1       # fraction of degree-one nodes
  P = 1000      # plane size
EndModel

BeginOutput
   Brite = 1            # 0/1: Save/Don't save in BRITE's format
   Otter = 0            # 0/1: Save/Don't save in Otter's format
EndOuput
```

C++ excerpt 12 Parsing routine for ToyInet model

```
InetPar* Parse::ParseInet() {

  ...
  InetPar* in_par;
  ParseIntField("N", n);
  ParseDoubleField("D", d);
  ParseIntField("P", p);
  ParseStringField("EndModel");
  ...

  in_par = new InetPar(...);

  return in_par;
}
```

C++ excerpt 13 ToyInet new model class and Generate methods

```
class ToyInetModel : public Model {

public:
   ToyInetModel(ToyInetPar* par);
   Graph* Generate();

private:
   void generate_degrees(/* args */);
   void place_nodes(/* args */);
   void connect_nodes(/* args */);
   int n;
   double d;
   int p;

}

Graph* ToyInetModel::Generate() {

   Graph* graph = new Graph(n);

   generate_degrees();
   place_nodes(graph);
   connect_nodes(graph);

   return graph;

}
```

Listing 7.1 BRIANA command-line interface

```
briana <config_file> [-o drf] [-i drf] [-h dr] [-c] [-p] [-m]

where:

  -o: outdegree (d: distribution, r: rank, f: frequency)
  -i: indegree (d: distribution, r: rank, f: frequency)
  -h: neighborhood size (d: distribution, r: rank)
  -c: clustering coefficient
  -p: path length distribution
  -m: min-hop path distribution
  -l: link-utilization analysis
```

8 A Comparative Study of Generation Models Using BRITE

Now that we know the design goals and the relevant implementation details of BRITE, in this section we provide a symbolic comparative study of some generation models. The idea of doing this comparison is to illustrate the design principles of BRITE in a "real-world" environment.

Recent empirical studies [9] have shown that Internet topologies exhibit power-laws of the form $y = x^{\alpha}$ for, among other properties, (P1) the outdegree of a node versus rank, and (P2) frequency of an outdegree versus outdegree. The seeming invariance of these properties with respect to size and time suggests they are fundamental properties of Internet topologies. After this discovery, the question to be asked, do the currently used topology generators generate topologies that satisfy this property? [16].

Using BRITE, we generated topologies according to the *RouterWaxman* and *RouterBarabasiAlbert* models. In addition, we used the *ImportFileModel* to import GT-ITM flat, GT-ITM Transit-Stub, and NLANR topologies and output them into BRITE's format. In order to do the analysis intended here, it is not needed to import the GT-ITM and NLANR topologies since they can be read directly by BRIANA. For each topology, we plot two types of plots for property (P1) and property (P2).

For the (P1) property, we plot outdegree versus rank in a log-log plot. For the (P2) property, we plot the frequency of outdegrees versus outdegrees as done in [9].

Figures 11 and 12 show the results for two of the data sets used in [9].

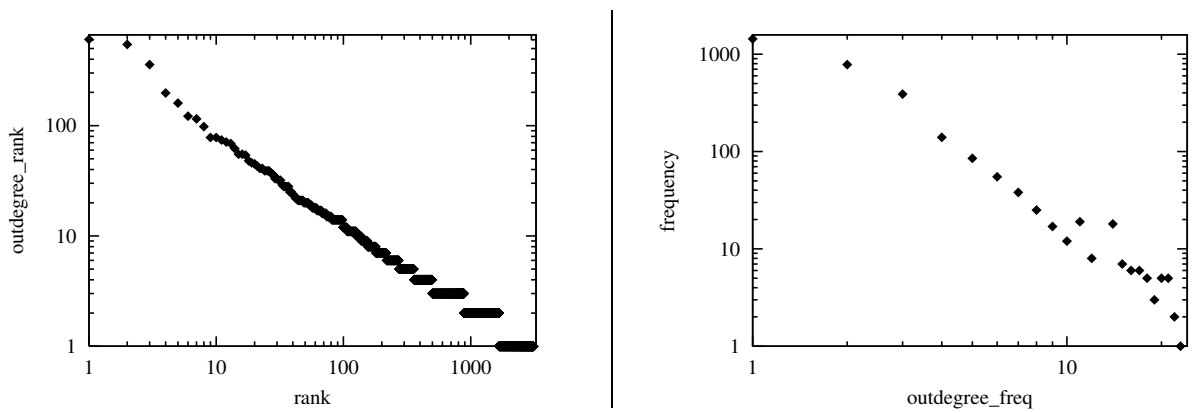

Figure 11: Rank-outdegree (left) and frequency-outdegree (right) for NLANR (11/1997) topology

Figures 13 and 14 show the results for one GT-ITM flat topology and one GT-ITM Transit-Stub topology, respectively.

Figures 15 and 16 show the results for the BRITE topologies using the *RouterWaxman* and *RouterBarabasiAlbert* models, respectively.

The goal of this section is not to make conclusive remarks with respect to the differences between the involved models/generators. However, we can see that when the Internet properties analyzed are the rank and frequency of node outdegrees [9], we can establish clear differences between generators aimed at reproducing degree-related properties (e.g. *BRITE/BarabasiAlbert*) and generators aimed at reproducing

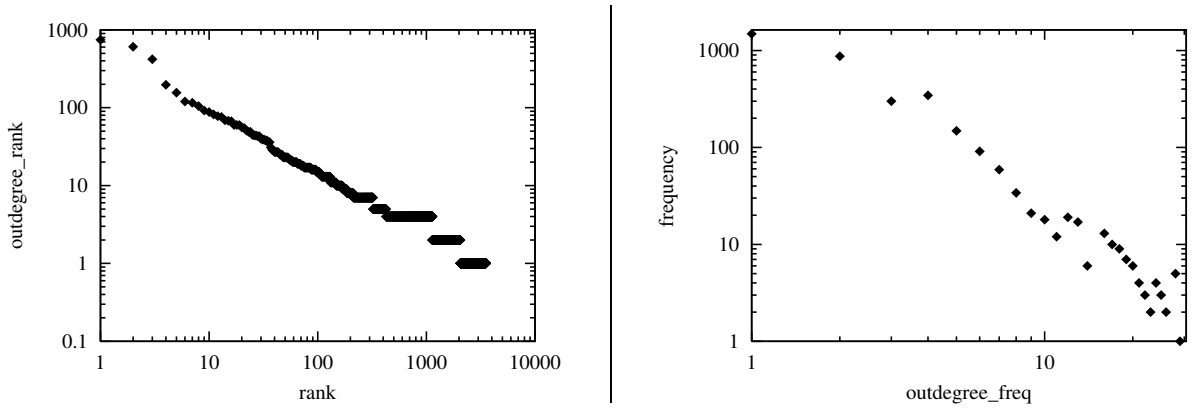

Figure 12: Rank-outdegree (left) and frequency-outdegree (right) for NLANR (04/1998) topology

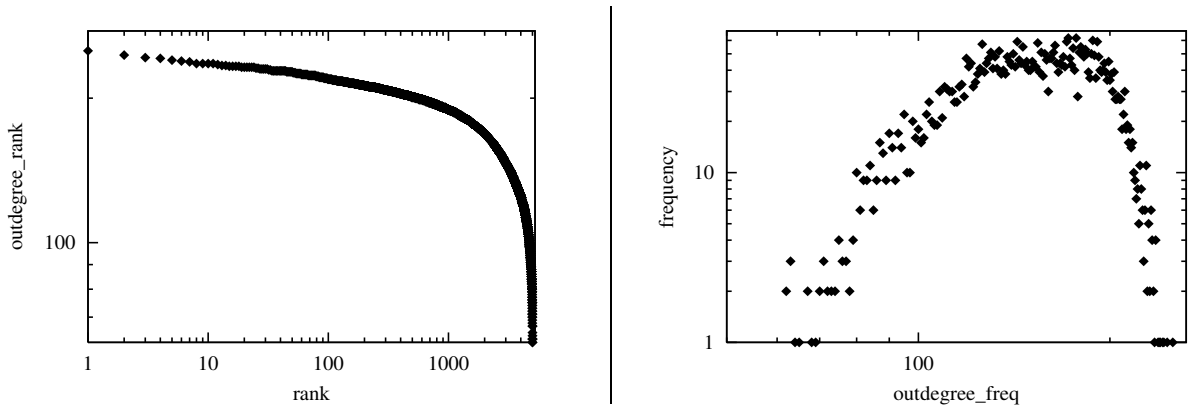

Figure 13: Rank-outdegree (left) and frequency-outdegree (right) for GT-ITM flat topology

hierarchical properties (e.g. GT-ITM). In Figures 11 and 12 we observe the same type of results obtained in [9]. In Figures 13 and 14 we can observe that GT-ITM models lack some characteristic that would allow them to strike a balance between hierarchical properties and degree-related properties. Figure 15 shows that the *Waxman* model implemented in BRITE, aimed at generating random networks, fails in reproducing the outdegree distribution properties of the Internet topologies. Finally, Figure 16 shows that the *BarabasiAlbert* model implemented in BRITE does a fairly good job in reproducing the outdegree properties of Internet topologies [16].

We want to emphasize that this symbolic comparative study was performed in about 20 minutes using BRITE and BRIANA. Even when the goal was not to reach conclusions from the comparisons, this exercise illustrates how the principles of BRITE and BRIANA translate into an increased efficiency and productivity in the generation and analysis of topologies.

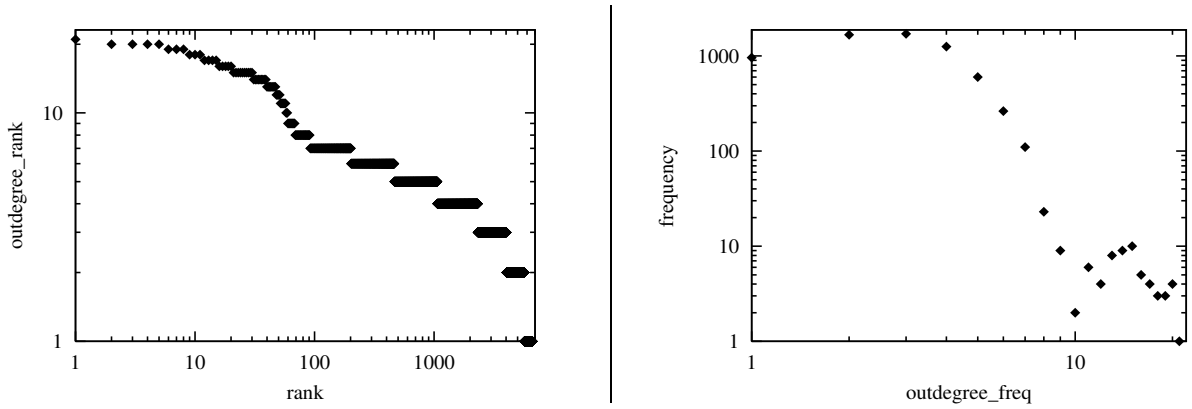

Figure 14: Rank-outdegree (left) and frequency-outdegree (right) for GT-ITM TS topology

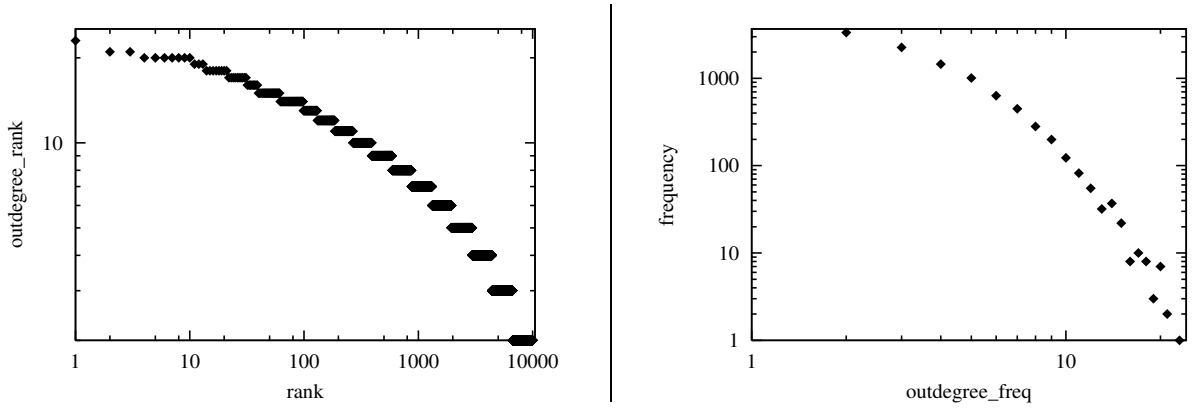

Figure 15: Rank-outdegree (left) and frequency-outdegree (right) for BRITE (Waxman) topology

9 Conclusions

Internet research requires good topology generation models that reproduce fundamental properties of the topology of the Internet. It is also a requirement to be able to use such models in simulations in an easy and effective way. In this paper, we have described BRITE, a universal topology generation tool. Furthermore, BRITE's is aimed at facilitating research in the area of topology generation by providing a generation tool that is inclusive, flexible, extensible and efficient. We also described the BRITE Analysis Engine, BRIANA, which is an independent piece of software designed and built upon BRITE design goals. The goal for BRIANA is to act as a repository of analysis routines along with a user–friendly interface that allows its use on different topology formats.

The main characteristics of BRITE are summarized as follows:

1. *Inclusiveness*. BRITE supports multiple generation models including models for flat AS, flat Router and hierarchical topologies. Models can be enhanced by assigning link attributes such as bandwidth and delay.

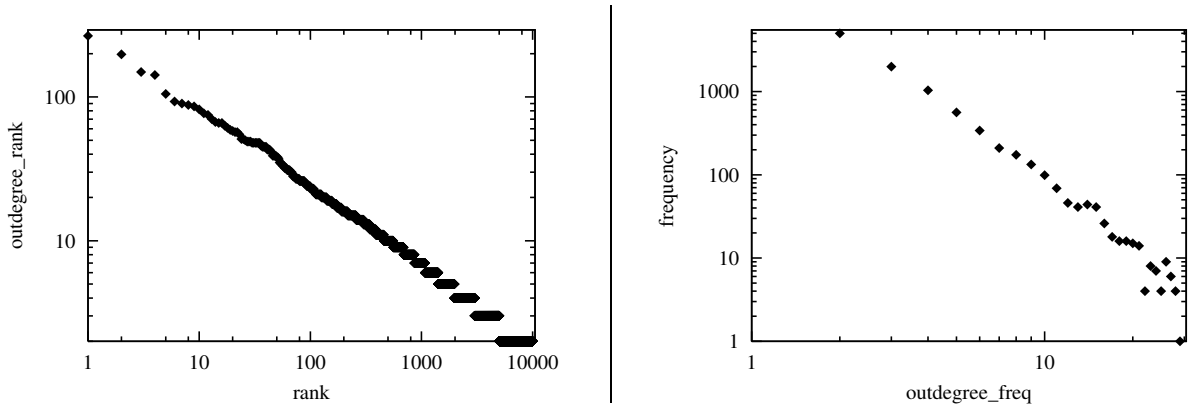

Figure 16: Rank-outdegree (left) and frequency-outdegree (right) for BRITE (BarabasiAlbert) topology

2. *Flexibility*. Generates topologies over a wide range of sizes. Restrictions such as minimum and maximum number of nodes should be reasonably avoided.

3. *Extensibility*. Object-oriented architecture makes BRITE extensible providing the researcher with the ability to add new models of generation, and the ability to import from or export to custom topology files.

4. *Ease of use*. Specifying topology generation parameters is simple via a GUI or a configuration file.

5. *Interoperability*. BRITE allows importing topologies from other topology generators and extend them or combine them with topologies generated according to other models.

6. *Robustness*. BRITE have been implemented to be tolerant to wrong inputs and extreme scenarios providing error detection and reporting capabilities.

7. *Efficiency*. BRITE is capable of generating large topologies (more than 100,000 nodes) using reasonable amounts of resources. However, it must be clear that the amount of resources used by BRITE will ultimately depend on the specific model being used.

8. *Portability*. Implemented in Java and C++.

The work presented in this paper constitutes the first release of BRITE. The success of a software tool could be said to be successful when it is used for purposes undreamed of by its authors. Furthermore, we hope to measure BRITE's and BRIANA's success by the number of suggestions, ideas and (hopefully not too many) bugs that we receive as a result of researchers embracing and benefiting from them.

We will continue improving the design of BRITE to include multiple inheritance, further import/export formats, increased GUI capabilities for BRITE, etc. In the current implementation, the GUI is not extensible in the sense that newly added models to BRITE are not easily incorporated into the GUI. In the front of BRIANA there is also a good deal of work waiting to be done. The design goals of BRIANA promise to increase the productivity of research on generation models for representative synthetic topologies. However, further

improvements in some aspects of its design include increased extensibility, as well as some implementation details such as the generation of plot files in a more refined "standard" format and a versatile GUI front-end.

In summary, topology generation is an exciting research topic along other Internet research (protocols, traffic characterization, etc.). We hope that new research will shape BRITE and BRIANA and that future releases will be carved by many in the networking research community.

A Appendix: Heavy-tailed Distributions

Heavy-tailed distributions (also known as power-law distributions) have been observed in many natural phenomena including both physical and sociological phenomena. One example is the geographic distribution of people around the world. Most places in the world are completely empty or barely populated, while there are a relatively small number of geographical locations which are very densely populated.

A distribution is said to have a heavy-tail if:

$$P[X > x] \sim x^{-\alpha}, \text{ as x } \rightarrow \infty, 0 < \alpha < 2$$

This means that regardless of the distribution for small values of the random variable, if the asymptotic shape of the distribution is hyperbolic, it is heavy-tailed [7]. The simplest heavy-tailed distribution is the Pareto distribution which is hyperbolic over its entire range and has probability mass function:

$$p(x) = \alpha k^{\alpha} x^{-\alpha-1}, \alpha, k > 0, x \geq k.$$

and its cumulative distribution function is given by:

$$F(x) = P[X \leq x] = 1 - (k/x)^{\alpha}$$

where k represents the smallest value the random variable can take.

Heavy-tailed distributions have properties that are qualitatively different to commonly used (memoryless) distributions such as the exponential, normal or Poisson distribution.

In the Internet, heavy-tailed distributions have been observed in the context of **traffic characterization** and in the context of **topological properties**. In the area of traffic characterization, evidence indicates that Ethernet traffic exhibits self–similar properties [14]; also WAN traffic exhibits self-similar properties [20], as is the case for traffic specifically associated with WWW transfers [7]. The main implication of such discoveries is that most previous analytic work done in Internet studies adopted assumptions such as exponentially–distributed packet interarrivals. Conclusions reached under such exponentiality assumptions may be misleading or incorrect in the presence of heavy-tailed distributions.

In the context of topological properties, recent empirical studies [9] have shown that Internet topologies exhibit power laws of the form $y = x^a$ for the following relationships: (P1) outdegree of node (domain or

router) versus rank, (P2) number of nodes versus outdegree, (P3) number of node pairs within a neighborhood versus neighborhood size (in hops), and (P4) eigenvalues of the adjacency matrix versus rank. Prior to this discovery, most Internet studies and analyses had been done using underlying topologies that lack such properties (e.g. random networks). Several possible causes and plausible analytical models that explain the appearance of these properties in Internet topologies have been proposed. However, the area of topology characterization has not been explored so extensively and causes for the appearance of such power laws have not been convincingly given. In [2] the authors indicate that two possible causes are (F1) preferential connectivity and (F2) incremental growth. In [16] the authors examine these two factors in the formation of Internet topologies, plus (F3) distribution of nodes in space, and (F4) locality of edge connections.

There is agreement in that heavy-tailed distributions are ubiquitous in the Internet. To the best of our knowledge, node placement and the distribution of bandwidth and delays have not been conclusively established, although Paxson observed wide variability in path characteristics such as losses, round-trip times and bandwidth [19], and high variability is one of the landmarks of heavy-tailed distributions. BRITE incorporates heavy-tails in the topology generation for some models. In particular, for models such as *Waxman* and *BarábasiAlbert*, the user can select to place the nodes in the plane according to a heavy-tailed distribution. Furthermore, for all the models provided, included the *Imported file model*, the user can select a heavy-tailed distribution of bandwidths to links. The idea is to generate *annotated* graphs with bandwidth information to study the effect that such distributions may have on the performance of certain protocols and algorithms. Finally, for the experimental generation model *Bottom-up hierarchical*, the user can select to assign routers to ASs according to a heavy-tailed distribution.

B Appendix: Parsing Support Routines (C++ Version)

In order to parse files such as the configuration files for a given model, the programmer will make use of the parsing functionality provided by BRITE. First it is neccesary to create an instance object of the *Parse* class. The corresponding constructor takes as an argument the name of the file that will be parsed. Following is a summary of the parsing routines provided in the C++ version of BRITE.

- **int GetNextTokenList(vector<string>& toks)**: Returns the set of string tokens in the vector **toks**, corresponding to the next line in the file being parsed.
- **int GetNextTokenList(string& from, int& pos, vector<string>& toks)**: Same as the previous routine but the tokens are extracted from a string instead of a file.
- **void ParseIntField(char* f, int& v)**: Used to parse configuration files. It parses an integer value which is preceded by a label represented by **f**. The parsed value is returned in **v**.
- **void ParseIntField(int& v)**: Used to parse imported files. It parses the next integer value. The parsed value is returned in **v**.
- **void ParseDoubleField(char* f, double& v)**: Used to parse configuration files. It parses a double value which is preceded by a label represented by **f**. The parsed value is returned in **v**.

- **void ParseDoubleField(double& v)**: Used to parse imported files. It parses the next double value. The parsed value is returned in **v**.
- **void ParseStringField(char* f)**: Parses a string from a file. It does not return any value because it is used to verify that the file being parsed is written in the right format.
- **void ParseStringField(char* f, string& s)**: Used to parse configuration files. Parses the next string field which is preceded by a label represented by **f**. The string field parsed is returned in **s**.
- **int FileSize()**: Used when parsing imported files (e.g. NLANR data). Returns the size of the file being parsed (in number of lines).
- **bool IsDelim(char ch)**: The Parse class contains an array of delimiters. Given a characters **ch** it returns whether or not the character is a delimiter.
- **bool IsComment(char ch)**: Used when parsing configuration files. Returns true if **ch** matches character '#'.

C Appendix: Downloading and Installing BRITE

In this section we describe how to generate topologies using BRITE. The current distribution of BRITE provides two versions of the generation tool: a Java implementation and a C++ implementation. Both implementations are very similar since both were implemented following the same design goals. There are some minor implementation differences though we will not discuss here.

C.1 Download

BRITE can be downloaded from [15]. Currently we have two almost identical versions of BRITE, one implemented in Java and the other implemented in C++. You can use either one. We plan to provide an applet that will allow generating BRITE topologies without downloading and installing it in the near future.

C.2 Installing and Running the Java version

Download the Java version at [15]. Once downloaded, follow these instructions to get BRITE up and running. Please note that you must have Java 2 (JDK1.3) installed in order to do so.

1. Unzip the downloaded file:

```
$ gunzip BRITE_JAVA.tar.gz
$ tar xvf BRITE_JAVA.tar
```

Directory/File	Contents
`Model/`	Model-related source files
`Graph/`	Graph-related source files
`Import/`	Import file formats
`Export/`	Export file formats
`../GUI/`	GUI source files
`Util/`	Miscellaneous utility functions
`Main/`	Entry point
`conf_files/`	Sample configuration files
`Makefile`	Makefile

Table 7: Contents of Java version's directory

2. Directory structure of Java version

   The contents of the directory where the Java version is untarred is shown in Table 7.

3. Change to the directory created when the Java distribution file is untarred. Assuming it is called *BRITE*, we change to that directory and compile:

```
$ cd BRITE
$ make java
```

4. Running BRITE from the GUI:

```
$ startGUI
```

5. Running BRITE from the command line (No Java required):

```
$ cd Java
$ java Main.BRITE <config_file.conf>  <output_file>  <seed_file>
```

C.3 Installing and Running the C++ version

Once downloaded, follow these instructions to get BRITE up and running.

1. Unzip the downloaded file:

```
$ gunzip BRITE_CPP.tar.gz
$ tar xvf BRITE_CPP.tar
```

2. Directory structure of C++ version

   The contents of the directory where the C++ version is untarred is shown in Table 8.

Directory/File	Contents
`Models/`	Source files for all supported models
`Util.h, Util.cc`	Utility functions
`Topology.h, Topology.cc`	Topology class
`Graph.h, Graph.cc`	Graph class
`Node.h, Node.cc`	Node class (graph)
`Edge.h, Edge.cc`	Edge class (graph)
`Parse.h, Parse.cc`	Parsing class
`BriteMain.cc`	Entry point
`conf_files/`	Sample configuration files
`Makefile`	Makefile

Table 8: Contents of C++ version's directory

3. Change to the directory created when the C++ distribution file is untarred. Assuming it is called *BRITE*, we change to that directory and compile:

```
$ cd BRITE
$ make c++
```

4. Running BRITE from the GUI (Java is required since the GUI is implemented in Java):

```
$ startGUI
```

5. Running BRITE from the command line (No Java required):

```
$ cd C++
$ brite <config_file.conf>  <output_file> <seed_file>
```

References

[1] W. Aiello, F. Chung, and L. Lu. A Random Graph Model for Massive Graphs. In *32nd Annual Symposium in Theory of Computing*, 2000.

[2] A.L. Barábasi and R. Albert. Emergence of Scaling in Random Networks. *Science*, pages 509–512, October 1999.

[3] P. Barford, A. Bestavros, J. Byers, and M. Crovella. On the Marginal Utility of Deploying Measurement Infrastructure. Technical Report Computer Science Technical Report 2000-018, Boston University, July 2000.

[4] B. Bollobas. *Random Graphs*. Academic Press, Inc., Orlando, Florida, 1985.

[5] K. Calvert, M. Doar, and E. Zegura. Modeling Internet Topology. *IEEE Transactions on Communications*, pages 160–163, December 1997.

[6] K.C. Claffy and D. McRobb. Measurement and Visualization of Internet Connectivity and Performance. *http://www.caida.org/Tools/Skitter*.

[7] M. Crovella and A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking*, pages 835–846, December 1997.

[8] M. Doar. A Better Model for Generating Test Networks. In *Proceeding of IEEE GLOBECOM*, November 1996.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-Law Relationships of the Internet Topology. In *ACM Computer Communication Review*, Cambridge, MA, September 1999.

[10] L. Gao. On Inferring Autonomous System Relationships in the Internet. *IEEE Global Internet*, November 2000.

[11] R. Govindan and H. Tangmunarunkit. Heuristics for Internet Map Discovery. In *Proceedings of IEEE INFOCOM'00*, March 2000.

[12] S. Shenker H. Tangmunarunkit, R. Govindan and D. Estrin. The Impact of Routing Policy on Internet Paths. In *Proceedings of IEEE INFOCOM 2001*, 2001.

[13] C. Jin, Q. Chen, and S. Jamin. Inet: Internet Topology Generator. Technical Report Research Report CSE-TR-433-00, University of Michigan at Ann Arbor, 2000.

[14] W.E. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson. On the Self-Similar Nature of Ethernet Traffic (Extended Version). *IEEE/ACM Transactions on Networking*, pages 1–15, Feburary 1994.

[15] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: Boston University Representative Internet Topology Generator. *http://cs-pub.bu.edu/brite/index.htm*, March 2001.

[16] A. Medina, I. Matta, and J. Byers. On the Origin of Power-laws in Internet Topologies. *ACM Computer Communication Review*, pages 160–163, April 2000.

[17] National Laboratory for Applied Network Research (NLANR). *http://moat.nlanr.net/rawdata/*.

[18] The Network Simulator (ns). *http://www.isi.edu/nsnam/ns/*.

[19] V. Paxson. End-to-End Internet Packet Dynamics. In *ACM/SIGCOMM'97*, Cannes, France., September 1997.

[20] V. Paxson and S. Floyd. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking*, pages 236–244, June 1995.

[21] V. Paxson and S. Floyd. Why We Don't Know How To Simulate The Internet. In *Proceedings of the 1997 Winter Simulation Conference*, Atlanta, GA, January 1997.

[22] Scalable Simulation Framework (SSF). *http://www.ssfnet.org/*.

[23] W. Theilmann and K. Rothermel. Dynamic distance maps of the Internet. In *Proceedings of IEEE INFOCOM'00*, March 2000.

[24] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, pages 440–442, June 1998.

[25] B. Waxman. Routing of Multipoint Connections. *IEEE J. Select. Areas Commun.*, December 1988.

[26] E. Zegura. Thoughts on Router-level Topology Modeling. *The End-to-end interest mailing list*.