Generating Synthetic RDF Data with Connected Blank Nodes for Benchmarking

Christina Lantzaki, Thanos Yannakis, Yannis Tzitzikas, and Anastasia Analyti

Computer Science Department, University of Crete,
Institute of Computer Science, FORTH-ICS, GREECE
{kristi,yannakis,tzitzik,analyti}@ics.forth.gr

Abstract. Generators for synthetic RDF datasets are very important for testing and benchmarking various semantic data management tasks (e.g. querying, storage, update, compare, integrate). However, the current generators do not support sufficiently (or totally ignore) blank node connectivity issues. Blank nodes are used for various purposes (e.g. for describing complex attributes), and a significant percentage of resources is currently represented with blank nodes. Moreover, several semantic data management tasks, like isomorphism checking (useful for checking equivalence), and blank node matching (useful in comparison, versioning, synchronization, and in semantic similarity functions), not only have to deal with blank nodes, but their complexity and optimality depends on the connectivity of blank nodes. To enable the comparative evaluation of the various techniques for carrying out these tasks, in this paper we present the design and implementation of a generator, called BGen, which allows building datasets containing blank nodes with the desired complexity, controllable through various features (morphology, size, diameter, density and clustering coefficient). Finally, the paper reports experimental results concerning the efficiency of the generator, as well as results from using the generated datasets, that demonstrate the value of the generator.

1 Introduction

Several works (e.g. [10]) have demonstrated the usefulness of blank nodes for the representation of the Semantic Web data. In a nutshell, from a theoretical perspective blank nodes play the role of the existential variables and from a technical perspective, as gathered in [2], they give the capability to (a) describe multi-component structures, like the RDF containers, (b) apply reification (i.e. provenance information), (c) represent complex attributes without having to name explicitly the auxiliary node (e.g. the address of a person consisting of the street, the number, the postal code and the city) and (d) offer protection of the inner information (e.g. protecting the sensitive information of the customers from the browsers). In [10] the authors survey the treatment of blank nodes in RDF data and prove the relatively high percentages of their usage. Indicatively, and according to their results, the data fetched from the 'rdfabout.com' domain

and the 'opencalais.com' domain, both of them parts of the LOD (Linked Open Data) cloud, consist of 41.7% and 44.9% of blank nodes, respectively.

However, their existence requires special treatment in various tasks. For instance, [10] states that the inability to match blank nodes increases the delta size (the number of triples that need to be deleted and added in order to transform one graph to another) and does not assist in detecting the changes between subsequent versions of a Knowledge Base, while [15] proves that building a mapping between the blank nodes of two compared Knowledge Bases that minimizes the delta size is NP-Hard in the general case. In [6] it is proved that (a) deciding simple or RDF/S entailment of RDF graphs is NP-Complete, and (b) deciding equivalence of simple RDF graphs is Isomorphism-Complete. In [8], a tutorial on how to publish Linked Data, the authors state that it becomes much more difficult to merge data from different sources when blank nodes are used, as there is no URI to serve as a common key. However we should note that the above tasks become tractable for the cases of non-directly connected blank nodes. Still, more complex blank node structures (i.e. cyclic) occur in practice. Indicatively, in [10] 1.6% of the structures are cyclic, while when querying the LOD Cloud Cache endpoint[1] we found out that it contains around 19 millions of blank nodes and almost 30 thousands of them participate in cyclic blank node structures.

In the face of strong identification needs, skolemization[2] is suggested, that replaces (some or all of) the anonymous resources with globally unique URIs. However, we will never "escape" from blank nodes. Even if we assign to all blank nodes URIs, we are obliged to treat them as *unnamed elements* when comparing or integrating data. For example, suppose that we want to integrate personal data from two or more sources where URIs are used for addresses (an address groups a street, a number, a city, etc). If we do not treat addresses as blank nodes, then we will treat all addresses as different, and thus we will end up with very poor information integration. We should also note that blank nodes is not an idiosyncratic feature of RDF. They occur everywhere; consider for instance the world of relational databases, and suppose that the same information is stored in two different relational databases, each supporting two different policies for *autokeys*. If we compare these databases then we would like to conclude that they are identical, but without treating the autokeys as blank nodes this is obviously impossible.

As regards the tasks in which blank nodes require special treatment, studies are oriented towards either finding special cases where the problems become tractable, or constructing algorithms that approximate the optimal solutions. Indicatively, [15] elaborates on a special case (i.e. RDF graphs with no directly connected blank nodes) where the problem of finding the optimal blank nodes mapping is solved in polynomial time, and provides two polynomial algorithms that approximate this mapping and can be used in the general case. One of them can map 150 thousands of bnodes in around 10 seconds. Another notable instance is [14], where the authors introduce the concept of bounded treewidth to prove

[1] `http://lod.openlinksw.com/sparql/`
[2] `http://www.w3.org/TR/rdf11-concepts/#section-skolemization`

that entailment checking can be efficient for RDF blank node structures that have bounded treewidth. Other works avoid matching blank nodes and instead make some quite simplistic assumptions (e.g. [9] for studying the dynamics of Linked Data).

In any of the above cases, an integrated benchmark would be really useful in order to create a common way to evaluate and compare these (and forthcoming) works. To fill this gap, in this paper we present the design and implementation of a generator, called BGen, which allows building big datasets containing blank nodes satisfying particular connectivity requirements. BGen is not the first RDF generator; there are several examples of RDF generators (described in Section 2). However, none of them deals sufficiently with the issue of benchmarking methods that become hard in the presence of blank nodes. The main objective of this generator is to create datasets with blank nodes of variable complexity. Key element for controlling the complexity of blank nodes is the notion of *BComponent* which is essentially a maximal sub-graph of blank nodes that is part of the whole graph. Having isolated this component we can control its complexity, through features like the size, the diameter, the density and the clustering coefficient.

The key contribution of this work is that we provide a method to produce variable in size and in complexity blank node components supporting a plethora of configuration parameters; including diameter, density, clustering coefficient, as well as a parameter for controlling the similarity of the named resources connected to the blank nodes. With the introduced method we can produce big graphs in size under a plausible time and without main memory problems. We also provide evidence that the selected features succeed in capturing the complexity that is crucial for the intended tasks, by reporting experimental results of blank node matching over the produced datasets.

The rest of this paper is organized as follows. Section 2 describes related generators and introduces the basic requirements of BGen. Section 3 describes the generator, i.e. its schema, parameters and phases. Section 4 reports experimental results regarding time and space, as the generated datasets are scaled up, and uses the generated datasets to evaluate an approximation task. Finally, Section 5 concludes the paper and identifies issues for further research. More information is available in the Web[3].

2 Related Work

Benchmarking in RDF is focused on the performance evaluation of the Semantic Web repositories. Some notable benchmark tools and works follow. The Lehigh University Benchmark (LUBM) [5, 4] aims at benchmarking systems with respect to use in OWL applications with large repositories. For data generation, they have built the UBA (Univ-Bench Artificial) data generator, that features random and repeatable data generation. The minimum unit of data generation is the university and for each university a set of OWL files describing its departments (e.g. courses, students, professors) are generated.

[3] `http://www.ics.forth.gr/isl/bnodeland`

The BSBM benchmark [1] is built around an e-commerce use case, and its data generator supports the creation of arbitrarily large datasets using the number of products as scale factor.

The Social Intelligence BenchMark (SIB) [11] is an RDF benchmark that introduces the S3G2 (Scalable Structure-correlated Social Graph Generator) for generating social graphs that contain certain structural correlations. Regarding qualitative evaluation, they evaluate the ability to have some plausible correlation in the data, while regarding quantitative evaluation, they evaluate scalability in terms of various parameters like clustering coefficient, average path length and number of the users. Even though S3G2 offers correlation between the graph structure and the generated data, it does not handle blank nodes and their connectivity issues.

A slightly adjusted version of the UBA generator was used to generate synthetic data with blank nodes in [15]. However, that version supports a limited set of control parameters (e.g. it does not support control over cycles, clustering coefficient etc); consequently it is not convenient for benchmarking.

To the best of our knowledge there is no generator in the literature that deals with the generation of blank nodes adequately. Although there is not any particular difficulty in adjusting an already existing generator to produce blank nodes, the difficulty arises when the blank nodes should be connected under particular connectivity patterns. These patterns differentiate from the previous, as the performance criteria of the evaluated functions are different, too. To fill this gap in this paper we focus on connectivity issues between the blank nodes of the instance layer.

3 The BGen Generator

At first we describe the requirements of the generator (§3.1), the RDF/S schema that we have defined (§3.2), and provide a simple instantiation example of that schema which also introduces the notion of *BComponent* (§3.3). Then, we analyze how we control the structural complexity of a *BComponent* (§3.4), and finally we present the main algorithm and its phases (§3.5 - §3.7).

3.1 Requirements

Here we list the main requirements, while in Section 3.5 we describe how BGen meets these requirements.

A. Correlation of data. The generator should produce resources that are not randomly correlated in order to control the structure of the generated data set and gain realism. A sensible method to implement such a correlation is to produce data over a specific real-world-like schema that supports various kinds of relationships (i.e. one-to-one, one-to-many, many-to-many).

B. Scaling of data. The generator should be able to generate big datasets suitable for evaluating how the tasks/applications (or the RDF systems upon which they are built) scale.

C. Generation of anonymous data. The generator should support the creation

of blank nodes as a percentage of the totally generated resources.
D. Connectivity in anonymous data. The generator should allow controlling the way blank nodes are connected using various features like diameter, density, clustering coefficient. The connectivity between the anonymous and the named data should also be controlled (through the schema and a similarity mode).

3.2 The Social Network Schema

Analogously to the UBA generator [5], we have created a schema that describes some basic classes and relations inside a social network. It is illustrated as a UML class diagram in Figure 1. A *social network* is the minimum unit of data generation. The primitive building block of this network is the class *Person* representing the members of this network. Each *person* has its *personal info*, described through its *name* (first name and last name), its *address* (street, zipcode, number and *city*), its gender and its birth date. Additionally, it has one or more *public messages* (where each *public message* is characterized by its content and its date). The instance *personal info* has its own *security mode*, which can get one of the following values: FriendsOnly, Public, Private.

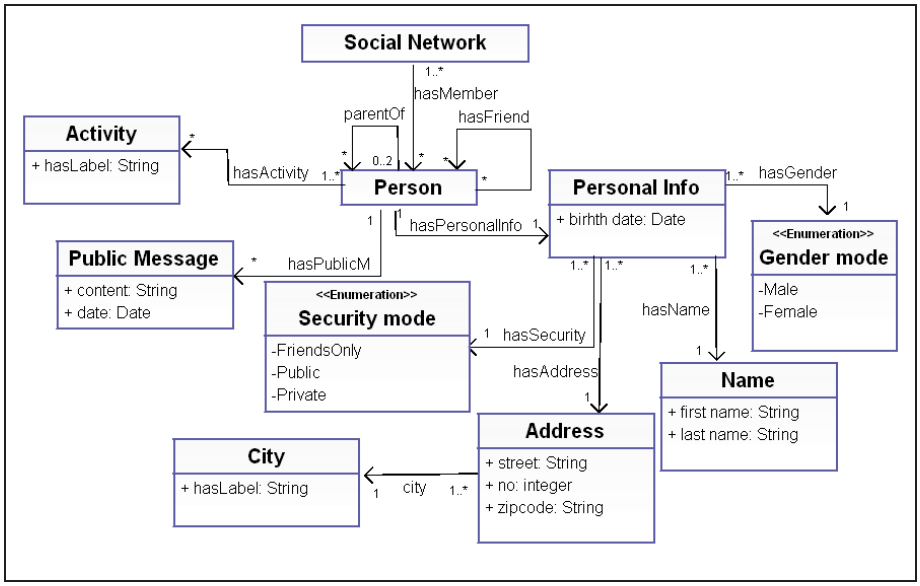

Fig. 1. The Social Network Schema of BGen

Each *person* is also connected with other *person*s through the `hasFriend` property indicating who is friend of whom. Apart from the friendship connections, other relationships (`parentOf, siblingOf`) can be created, too.

The class diagram of Figure 1 is represented in RDF/S and all classes are represented as instances of the `rdfs:Class`, while the ranges of their attributes are represented as subclasses of the `rdfs:Literal`. The enumeration classes

(Gender mode, Security mode) are represented through the `owl:DataOneOf`. To make the schema as realistic as possible, some restrictions were applied based on common sense and domain investigation: they are denoted in Figure 1 by the multiplicities of the depicted associations, e.g. each *person* has one *personal info*, while it can have more than one *public messages* and friends.

3.3 Instantiating the Social Network Schema: Example

Here we provide an example showing how the schema is instantiated. Figure 2 shows a *person* (always represented as a blank node instance) accompanied with its *personal info*, its *messages* etc. inside a *social network*, named `sn:socialNet1` (always represented as a URI instance). We shall call this a *BPerson instantiation*.

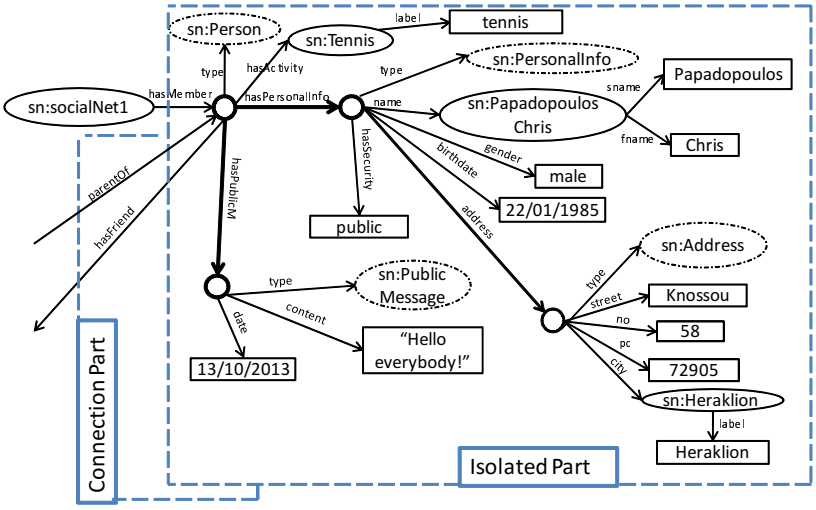

Fig. 2. A simple instantiation paradigm

The decision on how many of the generated instances inside a *BPerson instantiation* will be presented as blank nodes is based on a parameter named $|bPer|$, where $|bPer| \geq 1$ (i.e. a *person* is always instantiated as blank node), while $|uPer|$ controls how many will be presented as URIs. For the *BPerson instantiation* of Figure 2 we can see that $|bPer| = 4$ and $|uPer| = 3$ (3 URI instances). Although all classes of the social network schema can potentially acquire blank nodes as instances, the classes *City* and *Activity* are instantiated only by URIs because these resources need to be identified and indicated locally and even globally in real-world-like datasets.

Notice that each *BPerson instantiation* produces exactly one maximal tree of blank nodes, that is called *bTree* (nodes and edges in bold in Figure 2). More complex structures of blank nodes (e.g. cycles) require more than one *BPerson instantiations* to be connected and will be analyzed later. For the case where $|bPer| = 1$ the *bTree* is actually a single node and its height is 0, while as $|bPer|$ increases it becomes wider and its height can come up to 2 (the schema itself

poses this upper bound, as the longest paths that can be created for a *BPerson instantiation* is `hasPersonalInfo-address` and `hasPersonalInfo-name`).

For comprehension reasons we further separate a *BPerson instantiation* into two parts: (a) the *isolated part* that contains the personal information, the activities, and the public message(s) (upper right part of Figure 2), and (b) the *connection part* that contains all the connections of the *person* with other *persons* (lower left part). These connections are achieved through the properties `hasFriend` and `parentOf` (or `siblingOf`).

Let us now focus on the way a *person* is connected with other *persons*. To make the example as simple as possible, we shall consider the whole *isolated part* instantiation of a *person* as *one* instance that we will illustrate as a *single node* for visualization convenience. In Figure 3 (left) we have three *BPerson instantiations* which are connected through two friendship properties. Essentially three *bTrees* are connected and merged under a common tree, which will be called *BComponent*. However, this component is not always that simple. Figure 3 (right) shows a more complex connection between seven *BPerson instantiations* through six friendship properties, two parent relations and four sibling relations. We can now formalize the notion of *BComponent*.

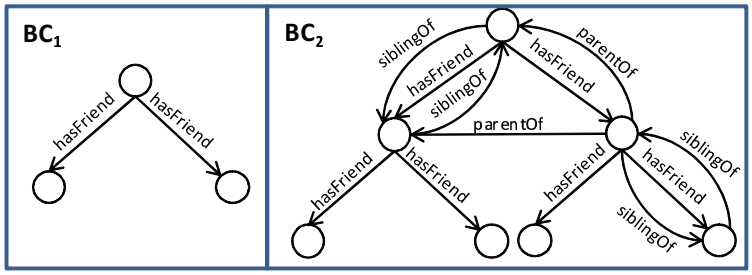

Fig. 3. The construction of two *BComponents*

Definition 1. We call a triple *btriple* if it contains only one blank node, and *bbtriple* if it contains two. Each maximal set of connected *bbtriples* of an RDF graph G forms a sub-graph, called *BComponent* of G. $\diamond$

It follows that Figure 3 illustrates two *BComponents*. These components and their features are crucial for controlling the blank nodes connectivity (analyzed shortly). In the context of the social network schema of BGen, a *BComponent* actually forms a community of connected *persons*.

3.4 Controlling the Complexity of *BComponents*

The problems, whose tasks are under evaluation (e.g. isomorphism checking, optimal bnode matching), become hard or even intractable for cases where the *BComponents* contain blank nodes that are connected with properties of the same label and where their directly connected named parts are similar enough (i.e. blank nodes of the same `rdf:type`). In such cases, each blank node cannot easily be distinguished from the others and the evaluated functions either become

more time consuming, or their output deviates significantly from the optimal one. Therefore, the following parameters are critical for controlling the *BComponents* and generating the desired datasets.

Intra-*BComponent* complexity (*morphology, clustering coefficient, density*) The generator's parameter *morphology* controls how the blank nodes of a *BComponent* are connected through `hasFriend` properties, and can take four values: 1) *'Single'* corresponding to a *BComponent* with only one *person* (i.e. 'anti-social' community), 2) *'Tree'* corresponding to a *BComponent* whose *persons* form a directed *tree* structure (i.e. 'pyramid' community), 3) *'DAG'* corresponding to a *BComponent* whose *persons* form a *directed acyclic graph* (i.e. 'semi-sociable' community), and 4) *'Graph'* corresponding to a *BComponent* whose *persons* form a graph with directed *cycles*, like that in Figure 4 (i.e. 'sociable' community). Note that the `hasFriend` property is not symmetric and both directions should be defined explicitly in order two *persons* to be friends of each other.

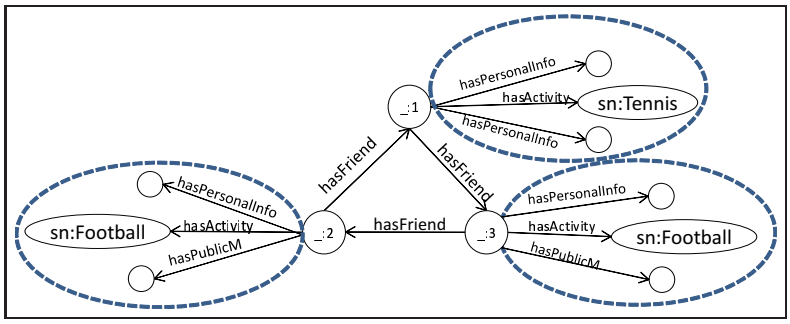

Fig. 4. A BComponent with similarly structured blank nodes

As a refinement parameter (over the *morphology*) we propose the parameter *average clustering coefficient*, $\bar{C}$ (formulated in equation 1) [16], that gives an average of the *local clustering coefficients*, C_k (formulated in equation 2), that quantifies the degree to which the friends of each *person* (incoming and outgoing) in the *BComponent* BC are friends between them (i.e. how strongly connected the community is). In the following equations assume that n is the number of blank node instances of the class *Person* inside the BC.

$$\bar{C} = \frac{1}{n}\sum_{k=1}^{n} C_k, \tag{1}$$

$$C_k = \frac{|\{(s, \texttt{hasFriend}, o) \in BC \mid s, o \in DFG(k)\}|}{|DFG(k)|(|DFG(k)|-1)}, where \tag{2}$$

$$DFG(k) = \{s \mid (s, \texttt{hasFriend}, k) \in BC\} \cup \{o \mid (k, \texttt{hasFriend}, o) \in BC\}$$

Note that only the `hasFriend` property and the instances of the class *Person* are taken into account. For the *Single* and *Tree* morphologies $\bar{C} = 0$, while for the *DAG* and *Graph* morphologies $\bar{C} \in (0, 1]$. For example, for the *Graph* *BComponent* of Figure 4 the average clustering coefficient is $\bar{C} = 1/2$.

For making the dataset more realistic we allow other relations (apart from `hasFriend`) to exist between two *persons* (i.e. the relations `parentOf` and `siblingOf` for our schema). The number of these properties inside a *BComponent* is computed by the parameter *density* [3], D (formulated in equation 3).

$$D = \frac{|\{(s, \texttt{parentOf}, o) \in BC\} \cup \{(s, \texttt{siblingOf}, o) \in BC)\}|}{n(n-1)} \quad (3)$$

Density actually counts the number of `parentOf` and `siblingOf` relations that are going to be added between the *persons* of the *BComponent*. Even though the theoretical upper bound of *density* is 1, it will rarely have high values (i.e. it is very rare all the *persons* of the *BComponent* to be parents or siblings between them).

***BComponent* similarity** It is not hard to see that the named parts of the RDF graphs assist the matching algorithms to identify and match their blank nodes. In our schema the named information (URIs or literals) that is connected directly with the *person* (i.e. *activity*, *personal info*, *public messages*) gives to this blank node a higher discrimination ability. For instance, in Figure 4 *person* _ : 1 differentiates from *person* _ : 2 through its different *activity*. On the other hand, *persons* _ : 2 and _ : 3 have exactly the same named parts (i.e. both of them have Football as their activity). The $similarityMode$ of the generator provides the ability to make the adjacent named structure of the blank nodes as similar as possible by (i) turning more URIs to blank node instances (increasing the $|bPer|$ parameter), (ii) making more URIs and literals same (i.e. more *persons* with same *activity*, street or *city*). In particular, the generator supports three scales of similarity: *easy*, *medium*, and *hard*. As it scales up, the similarity becomes higher and thus the bnode matching (as well as other tasks like *blocking* [13]) becomes more difficult.

3.5 The Generation Algorithm (Parameters and Phases)

The main algorithm of BGen takes the following seven input parameters[4] that express the desired features of the data set to be created:

1. N: the number of *resources* of the graph
2. P: the number of *persons*
3. $|BC|$: the number of *BComponents* (i.e. the number of communities of the social network)
4. $[minDmtr, maxDmtr]$: the range of the *diameter*[5] between the *persons* of the *BComponents*. Diameters will be distributed uniformly to the *BComponents*.
5. *morphologies*: a subset of $\{Single, Tree, DAG, Graph\}$ that controls how the *persons* of the *BComponents* are connected (as described earlier)
6. $mode_1$: its range is $\{realistic, uniform, powerlaw\}$ and controls the distribution of the *BComponents* to the *morphologies*. The *realistic* mode distributes them according to the results of an analyzed corpus of real data in

[4] In case the combination of the values produces non-valid states, the user is urged to adjust them appropriately.

[5] The diameter is the greatest distance between any pair of vertices [7].

[10], the $uniform$ mode distributes them equally, while the *powerlaw* mode approximates the 80 - 20 rule [12].

7. $mode_2$: its range is $\{random, uniform, powerlaw\}$ and controls the distribution of the *persons* to the *BComponents*. The *random* mode distributes the *persons* randomly, the $uniform$ equally, and the *powerlaw* according to the Zipfian distribution [12].

In order to give the user the ability to evaluate the approximation functions to a greater extent, the following three parameters are also configurable (their values are auto-configured in case they are not given as input):

1. $[min\bar{C}, max\bar{C}]$: the range for the *average clustering coefficient* between the *persons* of the *BComponents*.
2. $[minD, maxD]$: the range of the *density* between the *persons* of the *BComponents*.
3. $similarityMode$: its range is $\{easy, medium, hard\}$ and controls the similarity of the named resources connected to the *BComponents*.

Note that for the ranges ($[min\bar{C}, max\bar{C}]$,$[minD, maxD]$) their values are distributed uniformly to the *BComponents*. Recall that both *morphology* and *average clustering coefficient* are computed in terms of the `hasFriend` properties of a *BComponent*, while the *density* is computed in terms of the `parentOf` and `siblingOf` properties. Let us now see the production process. Initially, the generator enters the *Preparation* phase (described in §3.6) that aims at computing all the necessary parameters for the internal steps of the algorithms (i.e. it computes the features of *each BComponent* that will be created). Then, it enters the *Instance Generation* phase (described in §3.7) that produces all the *BComponents*, as well as the named resources that are connected with them. The last phase is the *Connection* phase, that connects all the generated *BComponents* under the finally generated graph. Below we present analytically these three phases.

3.6 Phase I: The Preparation Phase

The *Preparation Phase* takes as input the parameters of the main algorithm and outputs a set of arrays; one array with $|BC|$ elements for each feature of the *BComponents* to be created. Thus, each *BComponent* BC_i of the final graph is described through the values of the i-th elements of the exported arrays.

At first, the algorithm computes the number of blank nodes (B) and the number of URIs (U) taking N, P and $similarityMode$ into account. The rest of the algorithm computes the following features of each *BComponent* BC_i: (a) $|bPer_i|$: the number of blank nodes inside each one of the *BPerson instantiations* of BC_i, (b) $|uPer_i|$: the number of URIs inside each one of the *BPerson instantiations* of BC_i, (c) $|per_i|$: the number of *persons* in BC_i, (d) mor_i: the *morphology* of BC_i, (e) $dmtr_i$: the diameter of BC_i, (f) $|friends_i|$: the number of friends that each *person* of BC_i has, when $\bar{C}_i = 0$, (g) $\bar{C}_i$: the *average clustering coefficient* of the *persons* in BC_i and (h) D_i: the density of *persons* in BC_i.

Specifically, in order to compute the $|uPer_i|$, the U URIs are shared to the P *persons* uniformly. The $|BC|$ *BComponents* are split to the available *morphologies* ($\subseteq$ {*Single*, *Tree*, *DAG*, *Graph*}) based on the parameter $mode_1$.

The $|Single|$ *BComponents* can be easily initialized, since all their features are fixed. For the rest $|BC| - |Single|$ *BComponents*, $|per_i|$ is decided according to $mode_2$.

It follows the computation of the parameters $dmtr_i$, $\bar{C}_i$ and D_i which applies the uniform distribution to the ranges $[minDmtr, maxDmtr]$, $[min\bar{C}, max\bar{C}]$ and $[minD, maxD]$, respectively. Finally, the parameter $|friends_i|$ is computed for each *BComponent* separately by solving the following equation:

$$dmtr_i = \left\lceil \log_{|friends_i|}(|friends_i| - 1) + \log_{|friends_i|} |per_i| - 1 \right\rceil$$

As regards $dmtr_i$, the main structure of each *BComponent* forms a non-perfect k-ary tree of size N, where $k = |friends_i|$ and $N = |per_i|$ (recall the connection of *bTrees* into a common tree). The *diameter* of the *BComponent*, $dmtr_i$, is actually given as the height of this tree[6]. Afterwards, this tree is enriched with more `hasFriend` properties according to the value of $\bar{C}_i$.

3.7 Phase II: Instance Generation and Connection phase

At this phase BGen has a table with all the values that are needed to describe the *BComponents*. For each tuple i of this table it produces the triples of the *BComponent* BC_i. Having explained the features of a *BComponent*, let us show how the parameters determine the generated sub-graph through the examples of Figure 3. Recall that each *isolated part* instantiation (illustrated as a super node in Figure 3) is actually a set of instances of a *BPerson instantiation*. For this example, suppose that the *similarityMode* is set to *easy* for both *BComponents*; each instantiation contains only one blank node, so $|bper_1| = |bper_2| = 1$.

For the first *BComponent*, say BC_1, we get that there are three *BPerson instantiations*, so $|per_1| = 3$ and $dmtr_1 = 1$. From these values, we get that each *person* (apart from the leaf nodes) should be connected with two outgoing *hasFriend* relations, so $|friends_1| = 2$. Finally, $mor_1 = Tree$, $\bar{C}_1 = 0$, as no cycles or DAGs are formed between them through the *hasFriend* relations, and $D_1 = 0$, as no other relations (i.e.`parentOf, siblingOf`) exist.

For the second *BComponent*, say BC_2, there are seven *BPerson instantiations*, so $|per_2| = 7$. As, $dmtr_2 = 2$ each one of the *persons* is connected with two outgoing `hasFriend` relations; so $|friends_2| = 2$. Moreover, there are four `siblingOf` relations and two `parentOf` relations, as $D_2 = 0.07$. Again, since $mor_2 = Tree$, $\bar{C}_2 = 0$ (i.e. there are no cycles or DAGs between them through friendship relations).

Phase III: Connection phase Finally, all the created *BComponents* with their connected information (URIs and literals) are connected into the same graph.

[6] `http://xlinux.nist.gov/dads/HTML/perfectKaryTree.html`

This is implemented by connecting each *person* with a common instance of the *Social Network*.

4 Usage and Experimental Evaluation

The experimental evaluation[7] has two main objectives: (a) to investigate the time efficiency of the generator and how that depends on the input parameters (§4.1), and (b) to provide evidence that the selected features succeed in capturing the complexity that is crucial for the intended tasks (§4.2).

4.1 Evaluating efficiency

In brief, for creating a data set of 5 millions triples the required time ranges from 3.5 to 8 minutes depending on the complexity of blank nodes. These timings do not include the time required for saving in disk the output which depends on the technology of the storage media, rather than the problem at hand. Most of the time is spent in the instance generation phase, so the algorithm does not have any forbidding complexity, memory requirements or overhead.

Increasing the Number of Resources We generated datasets having *BComponents* of moderate complexity, where their resources (N, P, $|BC|$) gradually scale up ceteris paribus (i.e. keeping the complexity of the *BComponents* stable).

Figure 5(left) shows how this scaling impacts the total time, as well as the partial times for (i) the *preparation* phase, (ii) the construction of blank nodes in the *BComponents*, (iii) the generation of the rest of the graph (URIs and literals) (i.e. population time), and (iv) writing triples to the repository. It is evident that the times increase linearly to the number of resources. Just indicatively, for the given main memory space, BGen can generate a data set of up to 15 million resources (53 million triples) in less than 10 minutes.

Increasing the Complexity of Resources Regarding the rest of the parameters, we produce datasets with a stable number of resources ($N = 5$ million) scaling up the complexity of the *BComponents*, by gradually scaling the complexity parameters.

Figure 5(right) shows four different groups of datasets and how the total time is affected when increasing the complexity of the *BComponents*. Inside each group the aforementioned parameters are scaled up in three complexity levels. *Clustering coefficient* gets $[0.2, 0.4]$,$[0.6, 0.8]$ and $[0.6, 0.8]$, *density* gets $[0, 0.1]$, $[0, 0.2]$ and $[0, 0.4]$, while the *similarityMode* gets *easy*, *medium* and *hard*, for each complexity level, low, medium, high, respectively. From one group to the

[7] All experiments were conducted using the Sesame SailRepository over Main Memory Store (`http://openrdf.callimachus.net/sesame/2.7/apidocs/org/openrdf/repository/sail/SailRepository.html`)using a PC with Intel i5-2500 3.3 GHz, 8 GB Ram, running Windows 7 (64-bit).

other, $|BC|$ is scaled down creating *BComponents* with more *persons* (as shown in the X-axis). As the complexity of *BComponents* increases we can see a linear increase in the total generation time; thus BGen remains time efficient.

The only restriction factor for BGen is the data structure that temporarily stores the `hasFriend` properties of a current *BComponent*. On the worst case scenario, where $|BC| = 1$ and $\bar{C}$ =1, the space complexity comes to P^2. Indicatively, for the given memory space, in the extreme case where $N = 10^7$, $P = 10^7$, and $|BC| = 1$ we get an out of memory exception for $\bar{C} = 0.8$.

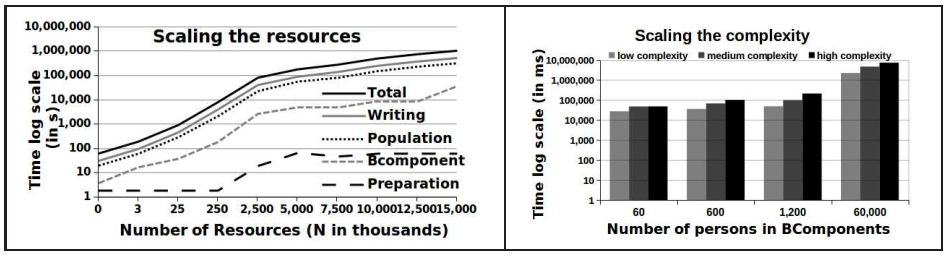

Fig. 5. Generation times scaling up the resources and their complexity

4.2 Using the Generated datasets for Benchmarking

For checking that the selected features succeed in capturing the complexity that is crucial for tasks like blank node matching, we generated one group of eight datasets scaling up the values of the *average clustering coefficient* for each *similarityMode*, *easy*, *medium* and *hard*, respectively. Each generated dataset has N = 25,000, P = 3,000, $|BC| = 10$, $[minDmtr, maxDmtr] = [1, 3]$, contains *graph morphologies*, $[minD, maxD] = 0$, and $mode1 = mode2 = uniform$. The range $[min\bar{C}, max\bar{C}]$ is [0, 0.2] for the dataset KB_1, [0,0.4] for KB_2, [0.2, 0.4] for KB_3, [0.3,0.5] for KB_4, [0.4, 0.6] for KB_5, [0.5, 0.7] for KB_6, [0.6, 0.8] for KB_7, and [0.7, 0.8] for KB_8. Each dataset is compared to itself. Because of space limitations, the datasets were only tested using the `signature`-based blank node matching method introduced in [15][8]. This method returns a mapping between the blank nodes of two graphs, aiming at minimizing the delta size when comparing these graphs. However, note that this method only approximates the optimal solution. The following experiments will allow us to consider which factors and in what way make this method deviate from the optimal. Figure 6 (left) shows the delta size, when each one of these datasets is compared to itself and Figure 6 (right) shows the time needed for this matching. It is evident that both the *average clustering coefficient* and the *similarityMode*, affect the deviation from the optimal delta size, that is zero, as the dataset is compared to itself. They also affect the time efficiency of the `signature` method. Recall that according to [15] the method can map 153,600 bnodes in 11 seconds. The current experiments

[8] The datasets are also tested using other blank node matching methods in `http://www.ics.forth.gr/isl/bgen/results`.

show that for the non-*easy* $similarityMode$ the method loses its capability to detect the optimal solutions even for low values of the *average clustering coefficient*. As these values increase the deviation is increased gradually. Notice that the delta size comes up to 234,676 triples for the last pair of datasets, where each dataset contains 291,000 triples, has $[min\bar{C}, max\bar{C}]$ in $[0.6, 0.8]$ and the $similarityMode$ is *hard*. As regards time, the `signature`-based method has time complexity linear to the number of blank nodes (i.e. $P * |bPer|$); therefore the increase of the clustering coefficient does not impact significantly on time. The small increases are explained because of the increase of the `hasFriend` properties that means bigger signatures. As the $similarityMode$ increases we observe bigger increases in time because of the increase of the number of blank nodes (through the increase of $bPer$).

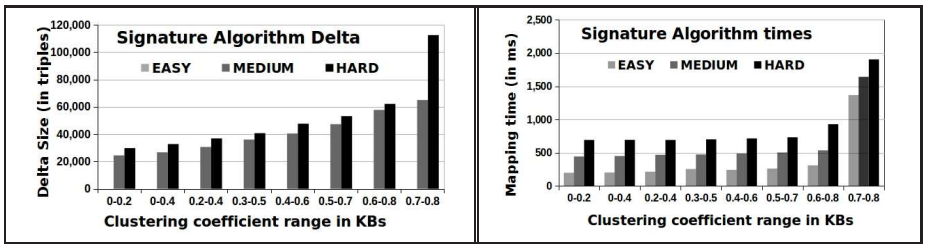

Fig. 6. Using the generated datasets to evaluate the `signature` method

In conclusion, we can say the generated datasets succeed in making clear how approximate solutions deviate from the optimal solution, and thus such datasets are suitable for comparatively evaluating such methods, e.g. for rapidly evaluating the potential and limits of various heuristics. All datasets used in this paper, as well as the current version of the generator are accessible for use[9].

5 Concluding Remarks

BGen is the first Semantic Web synthetic data generator able to create datasets with blank nodes appropriate for comparing and benchmarking various semantic data management tasks, e.g. equivalence and comparison.

Since the complexity of solving optimally such problems depends on the connectivity of blank nodes, the datasets produced by BGen can aid the assessment and the comparative evaluation of the various techniques that have been (or will be) proposed for carrying out such tasks.

The proposed method can produce variable in size and in complexity structures of blank nodes controlled through a plethora of configuration parameters, e.g. morphology (tree, DAG, cycle), diameter, density, clustering coefficient, similarity of the named resources connected to the blank nodes.

The construction algorithm has linear time and space requirements with respect to the number of resources. We also provided evidence that the selected

[9] `http://www.ics.forth.gr/isl/bgen/results`

features succeed in capturing the complexity that is crucial for the intended tasks, by reporting experimental results of bnode matching over the produced datasets. The results make evident how approximate solutions deviate from the optimal ones.

In the future we will use BGen for evaluating (and devising new) bnode matching techniques. We also plan to make this generator publicly available and associate it with LDBC[10]. Moreover one could easily extend the generator, e.g. by adding parameters controlling the lexical similarity of URIs and literals, for using it also for entity matching.

References

1. C. Bizer and A. Schultz. The berlin SPARQL benchmark. *International Journal On Semantic Web and Information Systems*, 2009.
2. L. Chen, H. Zhang, Y. Chen, and W. Guo. Blank Nodes in RDF. *Journal of Software*, 2012.
3. Thomas F. Coleman and Jorge J. More. Estimation of Sparse Jacobian Matrices and Graph Coloring Problems. *SIAM Journal on Numerical Analysis*, 1983.
4. Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. In *International Semantic Web Conference*, 2004.
5. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. In *Selected Papers from the Intern. Semantic Web Conf. ISWC*, 2004.
6. C. Gutierrez, C. Hurtado, and A. Mendelzon. Foundations of Semantic Web Databases. In *Proceedings of the Twenty-third Symposium on Principles of Database Systems (PODS), 2004, Paris, France*, 2004.
7. F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, 1969.
8. T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 2011.
9. T. Kfer, A. Abdelrahman, J. Umbrich, P. O'Byrne, and A. Hogan. Observing Linked Data Dynamics. In *ESWC*, 2013.
10. A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On Blank Nodes. In *Procs of the 10th Intern. Semantic Web Conference (ISWC 2011)*, 2011.
11. P. Minh Duc, P. A. Boncz, and O Erling. S3g2: A Scalable Structure-Correlated Social Graph Generator. In *TPC Technology Conference on Performance Evaluation & Benchmarking*, 2012.
12. M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 2005.
13. G. Papadakis, E. Ioannou, T. Palpanasa, C. Niederee, and W. Nejdl. A blocking framework for entity resolution in highly heterogeneous information spaces. *IEEE Knowledge and Data Engineering*, 2012.
14. R. Pichler, A. Polleres, F. Wei, and S. Woltran. dRDF: Entailment for Domain-Restricted RDF. In *Extended Semantic Web Conference (ESWC)*, 2008.
15. Y. Tzitzikas, C. Lantzaki, and D. Zeginis. Blank Node Matching and RDF/S Comparison Functions. In *Procs of the 11th Intern. Semantic Web Conference (ISWC 2012)*, 2012.
16. D.J. Watts and S.H. Strogatz. Collective dynamics of small-world networks. *Nature*, 1998.

[10] http://www.ldbc.eu/