Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Synthetic graph data generation

Web Engineering Seminar final report

Thom Hurks

Supervisors:
dr. G.H.L. (George) Fletcher
dr. N. (Nikolay) Yakovets

1.1

Eindhoven, January 2018

Abstract

The rise of social network applications over the past years has sparked massive interest in the study of such networks, specifically how to efficiently store and query them using databases. Besides that, many (social) domains are easily and naturally expressible in graphs and networks, such as corporate (financial) networks, criminal networks and even physical network systems such as the internet.

When studying social networks, or more general, graphs, it is obviously necessary to obtain test data. Such test data can be actual social networks or domain graphs, but this data is not always easily accessible. Companies and other institutions, such as governments, may have various reasons why they do not want to publish or share their graph databases. Reasons can vary from financial motivations where a company does not want to give away their core business asset to privacy concerns where data may not even be allowed to be shared. Sometimes the necessary graph data is available, but a researcher may require a much larger dataset to stress test a system, so additional data still needs to be obtained.

The solution to these problems is synthetic graph generation, where graphs with the appropriate size and structural properties are generated. These structural properties may be pre-defined using a graph schema or may be generated as well. This report examines the state of the art regarding synthetic graph generation, specifically focusing on possible improvements or extensions that can be applied to gMark [4], a schema driven generator for graphs and associated query workloads. These extensions, mostly regarding generating attribute values and their correlations, are discussed and a prototype implementing these extensions is created.

Preface

Thanks go to dr. G.H.L. (George) Fletcher and dr. N. (Nikolay) Yakovets for their supervision over this project during the web engineering seminar.

Contents

Chapter 1

Introduction

gMark [4] is a domain- and query language-independent framework targeting highly tunable generation of both graph instances and graph query workloads based on user-defined schemas [3]. As such, gMark can be used to generate synthetic graph instances to be used in experiments where graph datasets are needed but for various reasons not easily acquired. This study will first examine gMark's data model, then possible extensions to gMark's data model and will then examine the state of the art in the area of synthetic graph generation in order to identify possible improvements to gMark's graph generation capabilities.

Chapter 2

gMark data model

gMark generates directed edge-labelled graphs [4]. Edge-labeled graphs are graphs where labels are assigned to edges. These labels indicate the type of relationship that the edge denotes in the application domain.

Definition 1. The definition of an edge-labelled graph [2]

1. V is a finite set of vertices (or nodes).
2. E is a finite set of edges; formally, $E \subseteq V \times Lab \times V$ where Lab is a set of labels.

Edge-labelled graphs are simple as they only consist of vertices, edges (relationships) and labels and as such widely used. For example, they form the basis of the Resource Description Framework (RDF) used for encoding machine-readable content on the world-wide web [2]. In figure 2.1 you can see an example of an edge-labelled graph.

Figure 2.1: An edge-labelled graph encoding basic movie information with dashed labels on edges[2]

Since gMark generates directed edge-labelled graphs, we can amend definition 1 by adding that a tuple $e \in E$ is ordered, and these edges may be called arrows or directed edges. For example, in a social network a directed edge $(A, knows, B)$ implies A knows B, but does not imply that B knows A.

Edge-labelled graphs do have limitations due to their simplicity. For example, it is not possible to label nodes. If you need to indicate what type a node is, you would need to create a new node with the name of the type and then create a directed "is" edge from the entity node to the type node. Various limitations of edge-labelled graphs can be worked around by simply creating more nodes and relationships to encode properties, but how does one encode that a person is born on a certain date? Creating nodes for every possible birthday and then creating "born on" relationships is certainly possible, but also very cumbersome and it greatly increases the size of the graph. It is also not possible to assign properties to the labelled edges, so if one wants to record properties of relationships such as the date the relationship is created, then that also requires more nodes and edges. Figure 2.2 shows a graph similar to the one in figure 2.1 but with extra nodes to encode various properties.

Another downside of edge-labelled graphs is that it is more difficult to indicate a schema. As per our example, a node for the person "Anna" may have an "is" relationship to the node "person" to indicate that Anna is a person. Anna may also have a "born on" relationship to the node "01-01-1988" to indicate she was born on that date. However, it is not clear in this way that being born on a certain date is in fact a property of being a person.

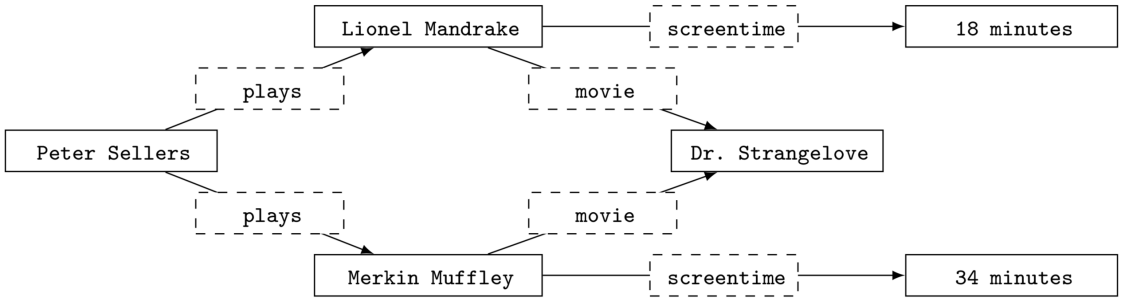

Figure 2.2: A version of figure 2.1 where extra nodes are used to encode properties [2]

Chapter 3

Property graph data model

To address the difficulties regarding edge-label graphs mentioned in the previous chapter, we can instead consider the property graph. Property graphs allow also labeling nodes, as shown in definition 2. For example, in a social network one could have nodes of type Person and University. Whereas with edge-labeled graphs one would create a node Person and have a relationship "is" from nodes to the Person node to indicate those nodes are Persons, with property graphs we simply label those nodes as Person. Such a system makes the resulting graphs smaller and more intuitive to understand. In addition to labeled nodes, property graphs also associate attributes with edges and nodes. For example, a node of type Person can have the property *family name*. Again, using extra nodes and edges this could also be modelled using an edge-labeled graph, but it would be more cumbersome. Also, with the property graph model we have a much more direct notion of a schema; the set of attribute names of nodes of a certain type can be considered as the schema of that node type. In edge-labeled graphs it is less clear which edges contribute to the schema of the node type and which edges express a relationship with a different node type, in addition to the fact that nodes cannot be labeled in edge-labeled graphs.

Definition 2. The definition of a property graph [2]. A property graph G is a tuple $(V, E, \rho, \lambda, \sigma)$, where:

1. V is a finite set of vertices (or nodes).
2. E is a finite set of edges such that V and E have no elements in common.
3. $\rho : E \to (V \times V)$ is a total function. Intuitively, $\rho(e) = (v_1, v_2)$ indicates that e is a directed edge from node v_1 to node v_2 in G.
4. $\lambda : (V \cup E) \to Lab$ is a total function with Lab a set of labels. Intuitively, if $v \in V$ (resp., $e \in E$) and $\rho(v) = l$ (resp., $\rho(e) = l$), then l is the label of node v (resp., edge e) in G.
5. $\sigma : (V \cup E) \times Prop \to Val$ is a partial function with $Prop$ a finite set of properties and Val a set of values. Intuitively, if $v \in V$ (resp., $e \in E$), $p \in Prop$ and $\sigma(v, p) = s$ (resp., $\sigma(e, p) = s$), then s is the value of property p for node v (resp., edge e) in the property graph G.

By extending gMark to the property graph model, users can easily and clearly indicate different node types and give those nodes a set of attributes. For example, gMark could generate nodes of type Person each with attributes first name, family name, gender and birth date. This moves the scope of gMark away from just generating graphs and into the area of generating valid numeric and textual values for these attributes.

Chapter 4

Synthetic graph generation techniques

Before extending gMark, we examine the state-of-the-art in synthetic graph generation to see what other techniques exist and what the strengths and weaknesses of the various methods are.

4.1 LDBC

The LDBC (Linked Data Benchmark Council) uses a benchmark called the Social Network Benchmark [8, 1]. To generate the scaleable synthetic data sets necessary for this benchmark, the LDBC uses a tool called datagen. The LDBC wants to use datasets with sensible PageRank outcomes and graph clustering structure, so their approach is to design a data model and generate data sets within this fixed model. Their data model has fixed node types such as "Person", "University", "Country" and the attributes of nodes of these types are generated using pre-defined functions (age) and dictionaries (name). The advantage of using a fixed data model is that advanced correlations can be generated that would be difficult to generate in the generic case where nothing about the model is assumed. For example, datagen ensures that persons are more likely to be connected in the social network when they are similar. They accomplish this by pre-defining a similarity metric between nodes of type Person and by then grouping persons having similar scores together when generating edges. This ensures that persons living in the same location, studying at the same university or having similar interests are more likely to be connected in the network. A few other correlations that datagen ensures are sensible are person name with respect to location and gender, person email address with respect to the popularity of the email's domain name, IP address with respect to location and duration of employment with respect to age. The result is that datagen can generate graphs containing realistic data up to correlations between attribute values, but the clear downside is that it can only generate graphs with one specific fixed schema. Graphgen cannot generate graphs based on an arbitrary schema.

4.2 Kronecker graphs

Kronecker graphs are based around the Kronecker product. The Kronecker product is an operation on two matrices as shown in definition 3.

Definition 3. $A \otimes B = \begin{bmatrix} a_{11}\mathbf{B} \dots a_{1n}\mathbf{B} \\ \vdots \ddots \vdots \\ a_{m1}\mathbf{B} \dots a_{mn}\mathbf{B} \end{bmatrix}$ for matrices A, B. [6]

The Kronecker product of two graphs is then the Kronecker product of their corresponding adjacency matrices [6]. This can be used to generate synthetic graphs as follows. Firstly, take

or randomly generate an initiator graph. Take the Kronecker product of this graph with itself to produce a new graph. One can then keep iteratively taking the Kronecker product of the resulting graph with itself in order to produce new bigger graphs. The reasoning behind this process is to recursively create self-similar graphs that exhibit the densification power law, for which the Kronecker product is suitable. The recursive self-similar structure can be observed in figure 4.1. Intuitively, real-world communities also recursively create new communities, which is how Kronecker graphs mimic real-world social networks.

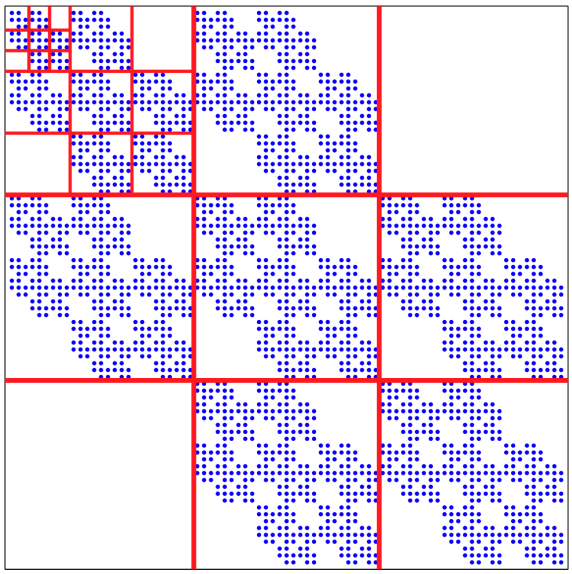

Figure 4.1: The adjacency matrix of the 4^{th} Kronecker power of a matrix. Dots are non-zero matrix entries, white space represents zeroes. [6]

By nature of their discrete creation process, Kronecker graphs exhibit a "staircase effect" in various metrics such as degree distribution and network value. The solution is to add randomness to the creation process to form stochastic Kronecker graphs. This variant works as follows. The adjacency matrix of the initiator graph now becomes a probability matrix. Each value $[0, 1]$ denotes the probability that the edge is present. Apply the Kronecker graph creation process as usual. The end result then defines a probability distribution over all graphs. One can then simply sample an instance from this distribution.[6]

As we are interested in generating synthetic graphs given a certain specification, the question now becomes how to tune the Kronecker creation process. Leskovec et al. [6] describe an optimization process that aims to find an initiator matrix that has the highest likelihood to produce the adjacency matrix of a given real graph. The idea is that if the adjacency matrices of the synthetic graph and the real graph are similar then the global statistical properties over these graphs will also match. An implementation of this process called KronFit is available as open-source software as part of the Stanford Network Analysis Platform (SNAP). [13]

A clear advantage of Kronecker graphs is that one can generate graphs exhibiting desirable structural properties also found in real-world networks "from nothing" and that one can also generate synthetic graphs matching a given real-world graph by carefully choosing the initiator matrix. However, Kronecker graphs are not an all-in-one solution as they do not solve the problems of generating nodes of various types, generating different kinds of relationships and generating node attribute values. Assigning attribute values randomly to nodes would violate sensible correlations such as persons having friends within the same age group as themselves. Either one would need to graft labels, predicates and attribute values onto a Kronecker graph after-the-fact or the graph generation process somehow needs to account for attribute correlations and node type and predicate distributions.

4.3 MAGs

The multiplicative attribute graph (MAG) model is a stochastic network model that captures the interactions between node attributes and the network structure. [11] It only considers nodes with categorical attributes. In the MAG model, categorical node attributes have affinities to compute the probability of a link in the network. Affinities can be positive, such as shared interests increasing the probability of a link, or negative, such as same gender decreasing the probability of a relationship link. Formally, each categorical attribute has a square affinity matrix where the distinct categorical values form both the rows and the columns. Each cell of the matrix then stores the probability $p_{ij} \in (0, 1)$ for a pair of nodes to form a link, given given attribute value i of the first node and the value j of the second node. The total link probability between two nodes is then the product of all the selected cells of the affinity matrices of the attributes of the nodes. The MAG model was proved [11] to capture patterns observed in real-world networks, such as power-law distributions, small diameters, unique giant connected component and local clustering of the edges. Once a set of nodes with categorical attribute values is generated and a set of affinity matrices is constructed, the MAG model can be applied to generate the final graph. Affinity matrices can possible be manually constructed, but much more interesting and less tedious is to estimate the affinity matrices from a given graph. More specifically, given a network one wants to estimate the parameters of the distributions used to generate the node attribute values as well as the affinity matrices. An algorithm called MAGFit [10] was developed with this purpose and is available as open-source software as part of the Stanford Network Analysis Platform (SNAP). [13] Unfortunately, MAGFit assumes that each node attribute takes the value 0 or 1 and follows a Bernoulli distribution, such that each affinity matrix is 2 by 2. This limits the applicability of the MAGFit algorithm in practice where synthetic graph nodes are expected to have realistic and usable values. Also, the MAG model does not tackle the problem of non-categorical node attributes.

4.4 GScaler

GScaler [16] is a method for scaling an existing graph, analogous to DNA shotgun sequencing. GScaler works as follows. First, for each node in the input graph generate two pieces: a node with incoming edges and a node with outgoing edges. The result of this step is two bags (multisets) of pieces, one containing only pieces with incoming edges, and one containing only pieces with outgoing edges. Both bags are then scaled, ensuring the proportion of the pieces stays the same. After the scaling step it is possible the number of edges is incorrect, so the in- or out-degree of a few nodes can be adjusted in order to correct this. The pieces are then connected back to form nodes again, while making sure that the proportion of each node with a given combination of in- and out-degree stays the same. These nodes are then again connected together, ensuring that the proportion of the pairs of nodes A, B where node A has a certain in- and out-degree and node B has a certain in- and out-degree stays the same. Once all nodes are connected back together, the result is a scaled graph. It must be noted that this whole process is never perfect; in each step a solution is chosen that is as optimal as possible, where the optimization is done using the Manhattan distance. GScaler produces graphs more similar to the original graph than a method like stochastic Kronecker graphs, but just like Kronecker graphs it only generates the graph structure and does not account for node attributes. Also, the GScaler algorithm does not account for graphs that have multiple node types and where relations may only be valid between nodes of certain types.

4.5 DataSynth

DataSynth [15] is a conceptual framework for property graph generation with customizable schemas and characteristics. It describes the generation of graph structure, properties, cardinality of edge

types, distributions and correlations. The authors do not provide an implementation of DataSynth, but give an overview of their framework's algorithm.

4.6 Others

A few other models for graph generation are BTER [12] and Darwini [7], which do not allow specifying a schema. Myriad is another property graph generator that generates node attributes but does not generate edge attributes. Also, Myriad cannot generate many-to-many relations and does not allow specifying distributions over attribute values.

Chapter 5

Correlated attribute values

In the literature overview we covered various approaches at generating graphs. The most diversity can be found in the area of generating graphs with a realistic structure. Property graph generators that allow generating graphs given a schema with realistic attribute values for both nodes and edges, in other words true property graph generators, do not seem to exist to the best of our knowledge. This problem is also difficult to solve because one cannot just randomly generate attribute values.

The approach of randomly generating attribute values may seem tempting, but if the resulting synthetic graph is expected to be representative of real-world graphs clearly this will generate invalid or conflicting information. If a person is still alive, then the person's birth date needs to be plausible given the average lifespan of humans and names should not be random sequences of characters and be consistent with gender, country of birth and even year of birth.

In other words, since property values are correlated in real-world graphs, they should be correlated in the synthetic graphs as well. Besides correlations between property values for a given node or edge, the property values also need to be sensible within the whole network. For example, if nodes of type Person have a property *income*, then the distribution of all *income* values over all nodes of type Person should match typical real-world income distributions. Similarly, one can argue that the distribution of Person names should follow the popularity distribution of names in the real world.

5.1 Inter-node correlations

Inter-node correlations, or correlations *between* nodes are the easiest constraint to solve. As an example, for person names the user can be expected to provide a dictionary of name popularity in certain countries, and this dictionary can be used as a discrete distribution to generate person names. A user can also set a probability distribution for person *income* to generate income values. In this way it is possible to ensure the global distributions of attribute values are similar to that of real-world networks.

5.2 Intra-node correlations

Intra-node correlations, or correlations within a node or edge, are much more difficult to solve due to dependencies between attributes. For example, a person's country, gender and age may influence the person's name. Then, these same attributes together with other variables, like IQ or education, may influence a person's income. A person's income and age again influence a person's net worth. From this example, it becomes clear that the dependencies between attributes form a dependency graph. When generating sensible attribute values, one should start at the root of the graph and work their way down. This also means that the graph should ideally have a tree structure, because otherwise there may be circular dependencies between attribute values.

When attribute values cannot be freely assigned but need to make sense within the node or edge's attribute dependency graph, it also becomes more difficult to retain the global attribute value distributions. In the most simple case, one generates all attribute values according to the global distributions and randomly distributes them across nodes and edges. In the case where dependency graphs are respected, this process of distributing attribute values becomes a jigsaw puzzle where a perfect solution is likely impossible, so one would require some form of optimizing process.

This optimizing process becomes even more difficult when one considers that relations are also part of the attribute dependency graph. In a social network example, a person's country and gender may influence the "study at" relation to an educational institution node, and this relation then influences person *income* and other relations such as "works at". This brings us to another constraint.

5.3 Attribute-relation interactions

Besides using the attribute values to generate other attribute values, as is the case with intra-node correlations, one can also take attribute values into account when generating relations between nodes. For example, two person nodes can be more likely to form a "knows" relation when they are of similar age. LDBC graphgen implements this by defining a (fixed) similarity metric between nodes, and then nodes that are similar are more likely to form relations. These interactions between attributes and the network structure are also the most important feature of the MAG model [11], where these can be user-defined using affinities.

5.4 Temporal correlations

To produce a synthetic graph similar to real world networks, one must also consider temporal patterns. In a social network example, when a person has studied at a technical university, this increases the probability of that person then having a "works at" relation to a technical company in the person's field of study. Similarly, the "works at" and "worked at" relations all influence person node attributes such as income and net worth. Both "works at", "worked at" and "studied at" then influence "knows" relations between persons. Similarly to the problem of intra-node correlations, relations over time form another dependency graph, where newer relations are influenced by older relations.

5.5 Generating correlated graphs

As explained, inter-node correlations, intra-node correlations and temporal correlations all influence each other, but to ease graph generation a simplified model can be chosen.

First, one can generate attribute values given the global distributions of those values. Secondly, given the dependency graphs of node and edge attributes one can distribute these attribute values over the nodes and edges, trying to reach a closest fit such that attribute values are not clearly conflicting. Then, edges can be generated by starting at the root of the relation dependency graph and each time instantiating a relation between two nodes that are likely to form that relation based on the attribute values and other relations of both nodes.

Note that it is mentioned that relations influence node attribute values, such as "studied at" and "works at" relations influencing person income and net worth. However, to prevent circular dependencies in the graph generation process, we can instead "work our way back" for relations. As an example, after generating a person node with consistent attribute values, we can estimate the probability, taking into account age, income and net worth, that this person studied at a certain university or worked at a certain company. These generated "studied at" and "worked at" relations together with attribute values such as age and gender can then be used to generate "knows" relations between persons.

The idea behind this process of separated steps and clear hierarchies is to prevent attempting to solve a very complex problem all in one go, which is likely possible but will require sophisticated techniques with high time complexity.

Chapter 6

Extending gMark

Now that we have examined property graphs, the state-of-the-art in graph generation and the interesting problem of attribute correlations, we can look at extending gMark.

It is out of the scope of this study to fully extend gMark and solve the attribute correlation problem, so we start with the basics of letting gMark generate full property graphs.

In order to let gMark generate property graphs, we need to add attribute value specification to the gMark schema. gMark already knows the concept of node and edge labels, even though it technically produces edge-labeled graphs.

6.1 Attribute generation

We can divide attribute value specifications into three categories:

1. Numbers that follow a probability distribution.
2. Categorical strings.
3. Strings that follow a pattern.

Each attribute specification always includes the name of the attribute.

6.1.1 Numbers that follow a probability distribution

These values are numeric values such as age, income and wealth that are associated with a probability distribution, for example power-law distributions.

The user should be able to specify in the schema the probability distribution that the value is drawn from, the parameters of that probability distribution and possibly extra rules. For example, net worth can be negative if a person is in debt, while age can never be negative.

6.1.2 Categorical strings

Categorical strings cover all categorical values. Even categorical values that are numeric can be considered as strings. Examples of categorical values are gender and title. Each possible categorical value for a certain attribute is then associated with a Bernoulli distribution. The Bernoulli probabilities of the set of categorical values should sum to 1, unless empty values are allowed, and not be bigger than 1. One can then simply generate a random value $x \in [0, 1)$ and use x to select the right category.

6.1.3 Strings that follow a pattern

Strings, or numbers that can be considered as strings, that follow a pattern are the most difficult case. For example, one may consider phone numbers. Phone numbers cannot simply be randomly generated as numbers, because they follow a certain pattern with country codes, regional prefixes and local prefixes. Other examples are email addresses and postal (zip) codes. These values can be specified using regular expressions. The attribute value generator must then be able to generate random strings given a regular expression. Luckily, this is a well-known and solved problem [14].

Within the Java ecosystem there is a tool called Xeger (regex spelled backwards) that generates random strings from a regex. This tool inspired other tools of different and same names. For example, within the Python ecosystem there are various such packages available on the Python Package Index (pypi). Amongst others there are Exrex, Xeger, Egret, rstr and StringGenerator. For our purposes we will use "rstr", a Python tool inspired by Xeger [5].

6.2 A new gMark input schema

In the input schema that gMark uses, we need to encode a lot more information. For each type and predicate we need to encode attribute names and attribute value specifications, such as numeric distributions, regular expressions patterns or lists of categories with associated probabilities.

Also, the original XML schema for gMark has not been formally defined and has been created in an ad-hoc manner. XML lends itself really well to hierarchical data; if something is a property or child of a certain element, then you can simply nest it within the XML document. In the gMark XML schema, many attributes are encoded as elements, and elements that should be nested are not nested but siblings linked with a unique identifier such as a "symbol" for predicates. To properly encode attributes in the schema, we decided to create a new XML schema that is more user friendly and easy to read. This schema has been defined using RelaxNG [9], an XML schema specification language. The RelaxNG schema for the extended gMark can be found in appendix A. Besides adding the attribute encodings and properly nesting associated data, this schema also explicitly does not encode predicate constraints. Predicate constraints are specified in the gMark paper [3] but are ignored in the gMark graph generation algorithm, in which only type constraints are respected. As such we can remove them from the XML schema to avoid confusion.

The minimal valid gMark input schema can be seen in appendix B. This input schema contains the size of the output graph, one node type with a name, proportion and no attributes, one predicate with name and no attributes, and does not specify relations.

6.3 Attribute-relation interactions

Attribute-relation affinities, such as defined in the MAG model, have been added to the extended gMark schema, see appendices A and B, but are currently not implemented.

The specification of the affinities in extended gMark is very basic and allows listing the attributes, per predicate, that need to be taken into account when computing the overall affinity between two nodes of the same type. For example, when creating a Person to Person relation an affinity can be specified, but not when creating a Person to University relation. The parameters of an attribute affinity are weight and whether the affinity is inverse. Weight allows the user to specify how important the attribute is to compute the overall affinity between nodes. The inverse parameter denotes if the affinity is stronger with similar values or with dissimilar values. For example, when specifying a Marries relation between nodes of type Person the attributes with affinity can be age and gender. Gender will likely have a higher weight value, while age can have a somewhat lower weight value. For age the affinity will not be inverse, as similar age increases the probability of a Marries relation, whereas the affinity for gender will be inverse, as opposite gender increases the probability of a Marries relation. A possible naive implementation of affinity in gMark is then to use the affinity value to decide whether to emit a generated edge or not. This

means the affinity can then effectively only *decrease* the probability of two nodes having a relation, but not *increase* it. A more thorough implementation of affinity can be examined in future work.

6.4 Implementation

gMark is originally implemented in C++, but to allow for quick prototyping and to easily use existing libraries like XML parsers, RelaxNG validation support and regex-to-text systems it was decided to make a prototype in Python. The implementation is roughly 350 lines of code and uses the RelaxNG schema to validate the user's graph schema, then parses the schema into a simpler internal representation and then uses this representation to firstly generate edges exactly like gMark does and then generates node and predicate attributes. The implementation can be found in appendix C. The code will also be published on GitHub at a later date.

Chapter 7

Conclusions

gMark is a tool to generate synthetic graphs and associated query workloads. gMark currently uses a modified edge-labeled graph structure and we have shown that gMark can be extended to the property graph model. The property graph model allows nodes and edges to be labeled and allows for both node types and edge predicates to have attributes. Other state-of-the-art graph generation tools are examined and notably the LDBC graphgen tool is the target to aim for; LDBC graphgen is an advanced tool that ensures many correlations on inter-node, intra-node and temporal levels are present.

A new gMark graph schema is developed that allows for the specification of type and predicate attributes, amongst being subjectively easier to interpret for new users. The validation, interpretation and (basic) execution of this graph schema is prototyped in the Python3 programming language.

The implementation supports three attribute types: numerical, categorical and strings that follow a pattern. Other traits that can be specified for each attribute value are uniqueness and if the value is required or not. This prototype works and can already be used in practice.

7.1 Future work

Future work is to examine if the three attribute types are sufficient or maybe need to be expanded, and the clear main goal is to implement support for intra-node correlations as well as temporal correlations between relations. gMark has been extended to support inter-node correlations, so correlations on a global level covering the whole graph, but the problem of correlations between attribute values inside a node, and correlations between relations, is much harder and can be further explored. Node-relation affinities have been added to the extended gMark schema, but have not been implemented, so future work can also focus on incorporating affinities into the gMark algorithm. A graph generation tool that can take an arbitrary schema including advanced correlations and then generate synthetic graph instances does not exist to the best of our knowledge, so further work on extending gMark will be valuable.

Bibliography

[1] LDBC Social Network Benchmark. 5

[2] A Foundations of Modern Query Languages for Graph Databases. 2017. 2, 3, 4

[3] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark GitHub project, 2016. 1, 13

[4] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H L Fletcher, Au elien Lemay, and Nicky Advokaat. gMark: Schema-Driven Generation of Graphs and Queries. 2016. ii, 1, 2

[5] Brendan McCollam. rstr = Random Strings in Python, 2011. 13

[6] Jure@cs Stanford Edu, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, Zoubin Ghahramani, and Zoubin@eng Cam Ac Uk. Kronecker Graphs: An Approach to Modeling Networks Jure Leskovec. *Journal of Machine Learning Research*, 11:985–1042, 2010. 5, 6

[7] Sergey Edunov, Dionysios Logothetis, Cheng Wang, Avery Ching, and Maja Kabiljo. Darwini: Generating realistic large-scale social graphs. 10 2016. 8

[8] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The LDBC Social Network Benchmark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 619–630, New York, New York, USA, 2015. ACM Press. 5

[9] Murata Makoto James Clark. RELAX NG Specification, 2001. 13

[10] Myunghwan Kim and Jure Leskovec. Modeling Social Networks with Node Attributes using the Multiplicative Attribute Graph Model. 7

[11] Myunghwan Kim and Jure Leskovec. Multiplicative Attribute Graph Model of Real-World Networks. *Internet Mathematics*, 82(1):113–160, 2012. 7, 10

[12] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A Scalable Generative Graph Model with Community Structure. 2 2013. 8

[13] Jure Leskovec and Rok Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016. 6, 7

[14] Bruce McKenzie and Bruce McKenzie. Generating Strings at Random from a Context Free Grammar. 1997. 13

[15] Arnau Prat-Pérez, Joan Guisado-Gámez, Xavier Fernndez Salas, Petr Koupy, Siegfried Depner, and Davide Basilio Bartolini. Towards a property graph generator for benchmarking. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems - GRADES'17*, pages 1–6, New York, New York, USA, 2017. ACM Press. 7

[16] J W Zhang and Y C Tay. GSCALER: Synthetically Scaling A Given Graph. 7

Appendix A

gMark RelaxNG schema

```
<?xml version="1.0"?>
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
         xmlns="http://relaxng.org/ns/structure/1.0">
    <start>
        <element name="gmark">
            <attribute name="size">
                <data type="integer">
                    <param name="minInclusive">1</param>
                </data>
            </attribute>
            <element name="types">
                <oneOrMore>
                    <element name="type">
                        <ref name="typeContent"/>
                    </element>
                </oneOrMore>
            </element>
            <element name="predicates">
                <oneOrMore>
                    <element name="predicate">
                        <ref name="predicateContent"/>
                    </element>
                </oneOrMore>
            </element>
        </element>
    </start>
    <define name="typeContent">
        <attribute name="name">
            <data type="string">
                <param name="minLength">1</param>
            </data>
        </attribute>
        <element name="count">
            <ref name="countContent"/>
        </element>
        <optional>
            <element name="relations">
                <oneOrMore>
                    <element name="relation">
```

```
                    <ref name="relationContent"/>
                </element>
            </oneOrMore>
        </element>
    </optional>
    <optional>
        <element name="attributes">
            <oneOrMore>
                <element name="attribute">
                    <ref name="attributeContent"/>
                </element>
            </oneOrMore>
        </element>
    </optional>
</define>
 <define name="predicateContent">
     <attribute name="name">
        <data type="string">
            <param name="minLength">1</param>
        </data>
    </attribute>
     <optional>
         <element name="attributes">
             <oneOrMore>
                 <element name="attribute">
                     <ref name="attributeContent"/>
                 </element>
             </oneOrMore>
         </element>
     </optional>
</define>
<define name="attributeContent">
    <attribute name="name">
        <data type="string">
            <param name="minLength">1</param>
        </data>
    </attribute>
    <attribute name="required">
        <data type="boolean"/>
    </attribute>
    <attribute name="unique">
        <data type="boolean"/>
    </attribute>
    <choice>
        <element name="numeric">
            <ref name="numericContent"/>
        </element>
        <element name="categorical">
            <ref name="categoricalContent"/>
        </element>
        <element name="regex">
            <ref name="regexContent"/>
        </element>
    </choice>
```

```
</define>
<define name="relationContent">
    <attribute name="predicate">
        <data type="string">
            <param name="minLength">1</param>
        </data>
    </attribute>
    <attribute name="target">
        <data type="string">
            <param name="minLength">1</param>
        </data>
    </attribute>
    <optional>
        <attribute name="allow_loops">
            <data type="boolean"/>
        </attribute>
    </optional>
    <element name="inDistribution">
        <ref name="distributionContent"/>
    </element>
    <element name="outDistribution">
        <ref name="distributionContent"/>
    </element>
    <optional>
        <element name="affinities">
            <oneOrMore>
                <element name="attributeAffinity">
                    <attribute name="name">
                        <data type="string">
                            <param name="minLength">1</param>
                        </data>
                    </attribute>
                    <attribute name="inverse">
                        <data type="boolean"/>
                    </attribute>
                    <attribute name="weight">
                        <data type="float">
                            <param name="minExclusive">0</param>
                            <param name="maxInclusive">1</param>
                        </data>
                    </attribute>
                </element>
            </oneOrMore>
        </element>
    </optional>
</define>
<define name="distributionContent">
    <choice>
        <element name="uniformDistribution">
            <ref name="uniformContent"/>
        </element>
        <element name="gaussianDistribution">
            <ref name="gaussianContent"/>
        </element>
```

```
            <element name="zipfianDistribution">
                <ref name="zipfianContent"/>
            </element>
            <element name="exponentialDistribution">
                <ref name="exponentialContent"/>
            </element>
        </choice>
    </define>
    <define name="numericContent">
        <optional>
            <attribute name="min">
                <data type="float"/>
            </attribute>
        </optional>
        <optional>
            <attribute name="max">
                <data type="float"/>
            </attribute>
        </optional>
        <ref name="distributionContent"/>
    </define>
     <define name="categoricalContent">
        <oneOrMore>
            <element name="category">
                <ref name="categoryContent"/>
            </element>
        </oneOrMore>
    </define>
    <define name="categoryContent">
        <data type="string">
            <param name="minLength">1</param>
        </data>
        <optional>
            <attribute name="probability">
                <data type="float">
                    <param name="minExclusive">0</param>
                    <param name="maxExclusive">1</param>
                </data>
            </attribute>
        </optional>
    </define>
     <define name="regexContent">
        <data type="string">
            <param name="minLength">1</param>
        </data>
    </define>
    <define name="uniformContent">
        <attribute name="min">
            <data type="float"/>
        </attribute>
        <attribute name="max">
            <data type="float"/>
        </attribute>
    </define>
```

```
    <define name="gaussianContent">
        <attribute name="mean">
            <data type="float"/>
        </attribute>
        <attribute name="stdev">
            <data type="float">
                 <param name="minInclusive">0</param>
            </data>
        </attribute>
    </define>
    <define name="zipfianContent">
        <attribute name="alpha">
            <data type="float">
                <param name="minExclusive">1</param>
            </data>
        </attribute>
    </define>
    <define name="exponentialContent">
        <attribute name="scale">
            <data type="float">
                <param name="minExclusive">0</param>
            </data>
        </attribute>
    </define>
    <define name="countContent">
        <choice>
            <element name="fixed">
                <data type="integer">
                    <param name="minInclusive">1</param>
                </data>
            </element>
            <element name="proportion">
                <data type="float">
                    <param name="minExclusive">0</param>
                    <param name="maxInclusive">1</param>
                </data>
            </element>
        </choice>
    </define>
</grammar>
```

Appendix B

Example gMark input schema

```
<?xml version="1.0"?>
<gmark size="100000">
    <types>
        <type name="person">
            <count>
                <proportion>
                    1
                </proportion>
            </count>
            <relations>
                <relation predicate="knows" target="person" allow_loops="false">
                    <inDistribution>
                        <zipfianDistribution alpha="2.5"/>
                    </inDistribution>
                    <outDistribution>
                        <zipfianDistribution alpha="2.5"/>
                    </outDistribution>
                    <affinities>
                        <attributeAffinity name="gender" inverse="true" weight="1"/>
                    </affinities>
                </relation>
            </relations>
            <attributes>
                <attribute name="email" unique="true" required="true">
                    <regex>[a-z0-9][a-z0-9][a-z0-9_-][a-z0-9][a-z0-9]@(gmail|hotmail|live|outlook
                </attribute>
                <attribute name="gender" unique="false" required="true">
                    <categorical>
                        <category probability="0.5">male</category>
                        <category probability="0.5">female</category>
                    </categorical>
                </attribute>
                <attribute name="age" unique="false" required="true">
                    <numeric min="0" max="122">
                        <gaussianDistribution mean="43" stdev="15"/>
                    </numeric>
                </attribute>
                <attribute name="income" unique="false" required="true">
                    <numeric>
```

```
                    <exponentialDistribution scale="26000"/>
                </numeric>
            </attribute>
        </attributes>
    </type>
</types>
<predicates>
    <predicate name="knows">
    </predicate>
</predicates>
</gmark>
```

Appendix C

Python implementation of extended gMark

```
#! /usr/bin/env python3

import sys
import argparse
import numpy
from random import shuffle
from lxml import etree
import rstr
from json import dumps
from copy import deepcopy

__author__ = 'Thom Hurks'


def parse_args():
    parser = argparse.ArgumentParser(prog='gMark', description='Given a graph configuration, gene
    parser.add_argument('schema', metavar='schema_file', type=open, help='The graph schema XML fi
    parser.add_argument('--showschema', action='store_true', help='Show the parsed graph schema')
    parser.add_argument('--silentrun', action='store_true', help='Don\'t generate a graph, but si
    return parser.parse_args()


def generate_edges(graph_configuration, silent_run):
    schema = graph_configuration['schema']
    constraints = schema['constraints']
    distributions = schema['distributions']
    for distribution in distributions:
        v_src = []
        v_trg = []
        nr_source_nodes = constraints[distribution['source']]
        draws = draw_distribution(distribution['out_distribution'], nr_source_nodes)
        if draws is not None:
            for i in range(nr_source_nodes):
                times = max(int(draws[i]), 0)
                v_src.extend([i for _ in range(times)])
        nr_target_nodes = constraints[distribution['target']]
```

```
        draws = draw_distribution(distribution['in_distribution'], nr_target_nodes)
        if draws is not None:
            for j in range(nr_target_nodes):
                times = max(int(draws[j]), 0)
                v_trg.extend([j for _ in range(times)])
        shuffle(v_src)
        shuffle(v_trg)
        nr_edges = min(len(v_src), len(v_trg))
        allow_loops = distribution['allow_loops']
        if not silent_run:
            for i in range(nr_edges):
                source = v_src[i]
                target = v_trg[i]
                if allow_loops or source != target:
                    print('{},{},{},{}'.format(distribution['source'], source, distribution['pred


def generate_nodes(graph_configuration, silent_run):
    schema = graph_configuration['schema']
    constraints = schema['constraints']
    types = schema['types']
    for (node_type, attributes) in types.items():
        nr_nodes = constraints[node_type]
        for attribute in attributes:
            name = attribute['name']
            kind = attribute['type']
            prefix = '{},{},'.format(node_type, name)
            if kind == 'regex':
                for i in range(nr_nodes):
                    value = rstr.xeger(attribute['regex'])
                    if not silent_run:
                        print('{}{},{}'.format(prefix, i, value))
            elif kind == 'categorical':
                randoms = draw_distribution({'name': 'random'}, nr_nodes)
                categories = []
                cumulative_probabilities = []
                cumulative_probability = 0
                for (category, probability) in attribute['categories'].items():
                    cumulative_probability += probability
                    cumulative_probabilities.append(cumulative_probability)
                    categories.append(category)
                category_range = range(len(categories))
                for i in range(nr_nodes):
                    random = randoms[i]
                    for cat_index in category_range:
                        if random < cumulative_probabilities[cat_index] and not silent_run:
                            print('{}{},{}'.format(prefix, i, categories[cat_index]))
                            break
            elif kind == 'numeric':
                numbers = draw_distribution(attribute['distribution'], nr_nodes)
                attr_min = attribute['min']
                attr_max = attribute['max']
                if attr_min or attr_max:
                    numpy.clip(numbers, attr_min, attr_max, out=numbers)
```

```
                # TODO: make nr of decimals configurable.
                numpy.around(numbers, decimals=0, out=numbers)
                # TODO: only cast to int when nr of decimals = 0 (to remove the decimal .0)
                numbers = numbers.astype(int, copy=False)
                for i in range(nr_nodes):
                    if not silent_run:
                        print('{}{},{}'.format(prefix, i, numbers[i]))


def draw_distribution(distribution, number):
    name = distribution['name']
    if name == 'uniform':
        return numpy.random.uniform(low=distribution['min'], high=distribution['max'], size=numbe
    elif name == 'gaussian':
        return numpy.random.normal(loc=distribution['mean'], scale=distribution['stdev'], size=nu
    elif name == 'zipfian':
        return numpy.random.zipf(distribution['alpha'], size=number)
    elif name == 'exponential':
        return numpy.random.exponential(scale=distribution['scale'], size=number)
    elif name == 'random':
        return numpy.random.random(size=number)
    else:
        sys.exit('Cannot draw from unknown distribution "{}"'.format(name))


def parse_input_schema(filename):
    relaxng = etree.RelaxNG(etree.parse('schema.rng'))
    parser = etree.XMLParser(remove_blank_text=True)
    document = etree.parse(filename, parser)
    try:
        relaxng.assertValid(document)
    except etree.DocumentInvalid as err:
        sys.exit('Input graph schema invalid: {}'.format(err))
    root = document.getroot()
    size = int(root.get('size'))
    type_nodes = root.find('types').findall('type')
    predicate_nodes = root.find('predicates').findall('predicate')
    (type_names, predicate_names) = get_unique_names(type_nodes, predicate_nodes)
    constraints = get_constraints(type_nodes, size)
    distributions = get_distributions(type_nodes, type_names, predicate_names)
    types = get_types(type_nodes)
    graph_schema = {
        'predicates': predicate_names,
        'types': types,
        'constraints': constraints,
        'distributions': distributions
    }
    return {
        'size': size,
        'schema': graph_schema
    }


def get_distributions(type_nodes, type_names, predicate_names):
```

```
    distributions = []
    for typeNode in type_nodes:
        source = typeNode.get('name')
        relations = typeNode.find('relations')
        if relations is None:
            continue
        relations = relations.findall('relation')
        for relation in relations:
            predicate = relation.get('predicate')
            if predicate not in predicate_names:
                sys.exit('Found relation with unspecified predicate "{}"'.format(predicate))
            target = relation.get('target')
            if target not in type_names:
                sys.exit('Found relation with unspecified target type "{}"'.format(target))
            if source != target:
                allow_loops = True
            else:
                allow_loops = relation.get('allow_loops')
                if not allow_loops:
                    allow_loops = False
            affinities = get_affinities(relation.find('affinities'))
            if affinities is not None and source != target:
                sys.exit('Affinities can only be specified on relations between the same node typ
            # TODO: check for duplicate distributions (target+predicate must be unique for this s
            distributions.append({
                'source': source,
                'target': target,
                'predicate': predicate,
                'allow_loops': allow_loops,
                'in_distribution': parse_distribution(relation.find('inDistribution')),
                'out_distribution': parse_distribution(relation.find('outDistribution')),
                'affinities': affinities
            })
    return distributions


def parse_distribution(distribution_node):
    distribution = distribution_node.find('uniformDistribution')
    if distribution is not None:
        low = float(distribution.get('min'))
        high = float(distribution.get('max'))
        if low > high:
            sys.exit('Invalid uniform distribution found')
        return {
            'name': 'uniform',
            'min': low,
            'max': high
        }
    distribution = distribution_node.find('gaussianDistribution')
    if distribution is not None:
        return {
            'name': 'gaussian',
            'mean': float(distribution.get('mean')),
            'stdev': float(distribution.get('stdev'))
```

```
        }
    distribution = distribution_node.find('zipfianDistribution')
    if distribution is not None:
        return {
            'name': 'zipfian',
            'alpha': float(distribution.get('alpha'))
        }
    distribution = distribution_node.find('exponentialDistribution')
    if distribution is not None:
        return {
            'name': 'exponential',
            'scale': float(distribution.get('scale'))
        }
    else:
        sys.exit('Could not parse distribution node "{}"'.format(distribution))


def parse_categories(category_nodes):
    categories = dict()
    total_probability = 0
    uniform_probability = 1 / len(category_nodes)
    for category_node in category_nodes:
        name = category_node.text
        probability = category_node.get('probability')
        if probability:
            probability = float(probability)
        elif total_probability > 0:
            sys.exit('Probability needs to be specified on all categories or none')
        else:
            probability = uniform_probability
        categories[name] = probability
        total_probability += probability
    if abs(total_probability - 1) > (2000 * sys.float_info.epsilon):  # 2000 is just an empirical
        sys.exit('The probabilities of the categories need to sum to 1, not "{}"'.format(total_pr
    return categories


def get_types(type_nodes):
    types = dict()
    for type_node in type_nodes:
        name = type_node.get('name')
        types[name] = get_attributes(type_node.find('attributes'))
    return types


def get_attributes(attribute_nodes):
    attributes = []
    if attribute_nodes is None:
        return attributes
    for attribute_node in attribute_nodes:
        name = attribute_node.get('name')
        required = attribute_node.get('required') == 'true'
        unique = attribute_node.get('unique') == 'true'
        kind = attribute_node.find('numeric')
```

```
        if kind is not None:
            low = kind.get('min')
            if low:
                low = float(low)
            high = kind.get('max')
            if high:
                high = float(high)
            if low and high and low > high:
                sys.exit('Invalid min and max attributes for numeric attribute')
            attributes.append({
                'name': name,
                'type': 'numeric',
                'required': required,
                'unique': unique,
                'min': low,
                'max': high,
                'distribution': parse_distribution(kind)
            })
            continue
        kind = attribute_node.find('categorical')
        if kind is not None:
            attributes.append({
                'name': name,
                'type': 'categorical',
                'required': required,
                'unique': unique,
                'categories': parse_categories(kind.findall('category'))
            })
            continue
        kind = attribute_node.find('regex')
        attributes.append({
            'name': name,
            'type': 'regex',
            'required': required,
            'unique': unique,
            'regex': kind.text
        })
    return attributes


def get_affinities(affinity_node):
    type_affinities = dict()
    if affinity_node is not None:
        affinities = affinity_node.findall('attributeAffinity')
        for attribute_affinity in affinities:
            type_affinities[attribute_affinity.get('name')] = {
                'inverse': attribute_affinity.get('inverse'),
                'weight': attribute_affinity.get('weight')
            }
    return type_affinities


def get_unique_names(types, predicates):
    type_names = set()
```

```
    predicate_names = set()
    for typeNode in types:
        type_name = typeNode.get('name')
        if type_name in type_names:
            sys.exit('Duplicate entry for type "{}"'.format(type_name))
        type_names.add(type_name)
    for predicate_node in predicates:
        predicate_name = predicate_node.get('name')
        if predicate_name in predicate_names:
            sys.exit('Duplicate entry for predicate "{}"'.format(predicate_name))
        predicate_names.add(predicate_name)
    if not type_names.isdisjoint(predicate_names):
        sys.exit('The type and predicate names overlap')
    return type_names, predicate_names


def get_constraints(types, size):
    constraints = dict()
    for typeNode in types:
        name = typeNode.get('name').strip()
        if name in constraints:
            sys.exit('Duplicate constraint found for type "{}"'.format(name))
        count = typeNode.find('count')
        fixed = count.find('fixed')
        if fixed:
            fixed = int(fixed.text.strip())
        else:
            proportion = float(count.find('proportion').text.strip())
            fixed = int(proportion * size)
        constraints[name] = fixed
    return constraints


def main():
    args = parse_args()
    graph_configuration = parse_input_schema(args.schema)
    if args.showschema:
        print_config = deepcopy(graph_configuration)
        print_config['schema']['predicates'] = list(print_config['schema']['predicates'])
        print(dumps(print_config, indent=4))
    generate_edges(graph_configuration, args.silentrun)
    generate_nodes(graph_configuration, args.silentrun)
    args.schema.close()


if __name__ == "__main__":
    main()
```