Final Project CS807

André Evaristo dos Santos
Jhonatan de Souza Oliveira

March, 2016

University of Regina

Department of Computer Science

Final Project CS807

André Evaristo dos Santos
Jhonatan de Souza Oliveira

March, 2016

André Evaristo dos Santos
Jhonatan de Souza Oliveira
Final Project CS807
March, 2016
Instructor: Dr. David Gerhard

University of Regina
Department of Computer Science
3737 Wascana Parkway
S4S 0A2Regina

Abstract

[TODO]

Contents

1 Introduction **1**

2 Background **2**
2.1 Bayesian Networks 2
2.2 Darwinian Networks 3
2.2.1 Definitions 4
2.2.2 Operations 4
2.2.3 Modeling 4
2.2.4 Inference 5

3 Darwin Library **6**
3.1 Structure 6
3.2 Features 7
3.3 Usage 8

4 Darwinian Network Library **9**
4.1 Structure 9
4.2 Features 9
4.3 Usage 11

5 Conclusion **12**

Bibliography **13**

1 Introduction

> *TO ADD*
>
> — **Somebody**

[TODO]

2 Background

We review [TODO].

2.1 Bayesian Networks

Let $U = \{v_1, v_2, \ldots, v_n\}$ be a finite set of variables, each with a finite domain. A singleton set $\{v\}$ may be written as v, $\{v_1, v_2, \ldots, v_n\}$ as $v_1 v_2 \cdots v_n$, and $X \cup Y$ as XY. For disjoint $X, Y \subseteq U$, a *conditional probability table* (CPT) $P(X|Y)$ is a potential over XY that sums to one for each value y of Y. The *children* $Ch(v_i)$ and *parents* $Pa(v_i)$ of v_i are those v_j such that $(v_i, v_j) \in \mathcal{B}$ and $(v_j, v_i) \in \mathcal{B}$, respectively.

BAYESIAN NETWORK

A *Bayesian network* (BN) [7] is a *directed acyclic graph* (DAG) $\mathcal{B}$ on U together with CPTs $P(v_1|Pa(v_1))$, $P(v_2|Pa(v_2))$,..., $P(v_n|Pa(v_n))$, where the *parents* $Pa(v_i)$ of v_i are those v_j such that $(v_j, v_i) \in \mathcal{B}$. We call $\mathcal{B}$ a BN, if no confusion arises. Figure 2.1 shows a BN with CPTs $P(fo)$, $P(bp)$, $P(lo|fo)$, $P(do|bp, fo)$, and $P(hb|do)$ shown.

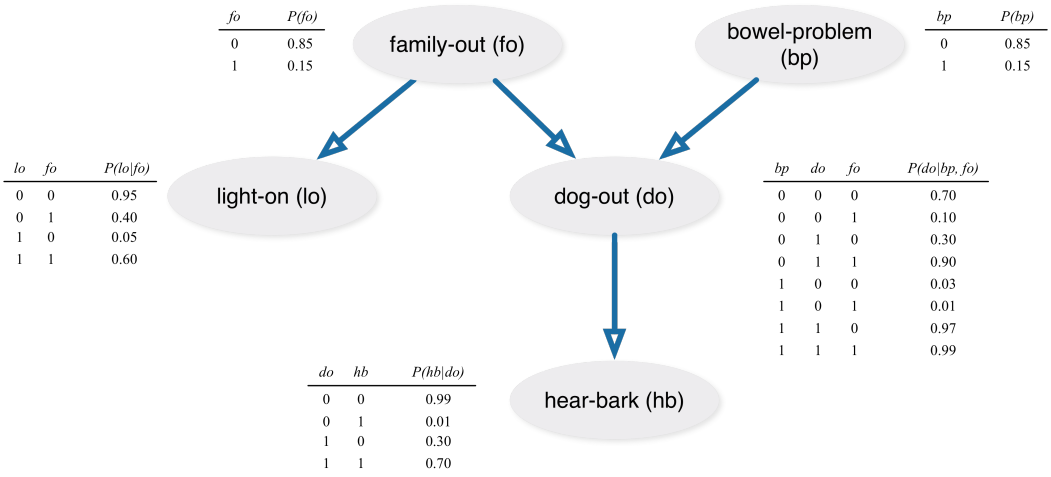

Fig. 2.1: A DAG $\mathcal{B}$ and its CPTs from [2].

The *conditional independence* [7] of X and Z given Y holding in $P(U)$ is denoted $I(X, Y, Z)$.

D-SEPARATION

The *d-separation method* [7] can be used to read independencies from a DAG. It is known that if $I_{\mathcal{B}}(X, Y, Z)$ holds by d-separation in $\mathcal{B}$, then $I(X, Y, Z)$ holds in $P(U)$.

VARIABLE ELIMINATION

One method to perform inference on a DAG is called *Variable elimination* (VE) [9], which computes $P(X|Y = y)$ from a BN $\mathcal{B}$ as follows: (i) all barren variables are removed recursively, where v is *barren* [9], if $Ch(v) = \emptyset$ and $v \notin XY$; (ii) all independent by evidence variables are removed, giving $\mathcal{B}^s$, where v is an *independent by evidence* variable, if $I(v, Y, X)$

holds in $\mathcal{B}$ by m-separation; (iii) build a uniform distribution $1(v)$ for any root of $\mathcal{B}^s$ that is not a root of $\mathcal{B}$; (iv) set Y to $Y = y$ in the CPTs of $\mathcal{B}^s$; (v) determine an elimination ordering σ from the moral graph $\mathcal{B}^s_m$ (as explained later); (vi) following σ, eliminate variable v by multiplying together all potentials involving v, and then summing v out of the product; and, (vii) multiply together all remaining potentials and normalize to obtain $P(X|Y = y)$.

MARKOV NETWORK

To perform probabilistic inference, a BN is commonly transformed into a *Markov network* (MN) [7], also called a *decomposable* MN. A MN consists of a triangulated (chordal) graph together with a potential defined over each maximal clique of triangulated graph. Given a DAG, the *moralization* [6] of the DAG is the undirected graph constructed by adding an undirected edge between each pair of parents of a common child and then dropping directionality. When necessary, edges are added to the moralized graph to obtain a triangulated graph. The maximal cliques are organized as join tree $\mathcal{T}$. Finally, the CPTs of the BN are assigned to nodes of $\mathcal{T}$.

2.2 Darwinian Networks

DARWINIAN NETWORKS

Darwinian networks (DNs) [1] were proposed to simplify working with *Bayesian networks* (BNs) [7]. Rather than modeling the variables in a problem domain, DNs represent the probability tables in the model. The graphical manipulation of the tables then takes on a biological feel, where a CPT $P(X|Y)$ is viewed as the novel representation of a *population* $p(C, D)$ using both *combative* traits C (coloured clear) and *docile* traits D (coloured dark).

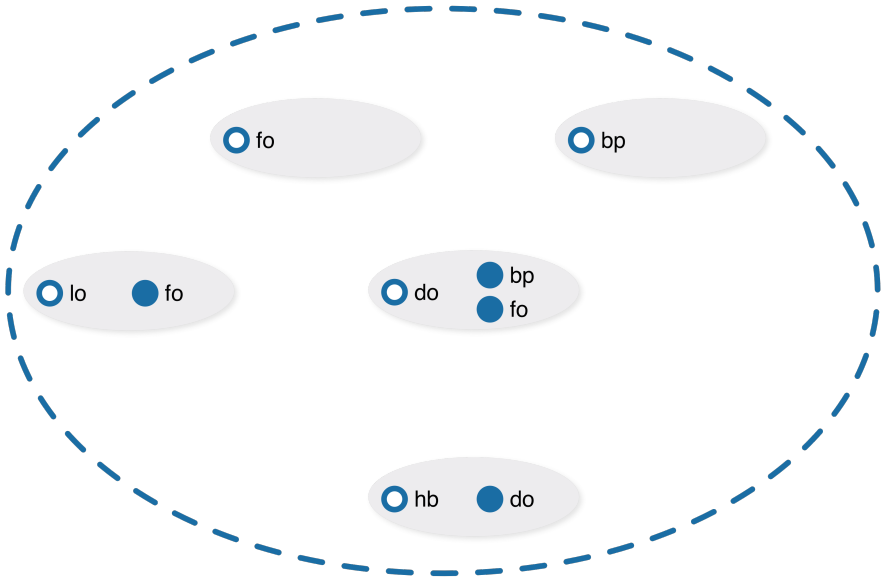

Fig. 2.2: A DN $\mathcal{D}$ representing DAG $\mathcal{B}$ of Figure 2.1.

Adaptation and evolution are used to represent the testing of independencies and inference, respectively. Thus, DNs can represent exact inference VE, as well the test of independencies d-separation. Hence, DNs can unify modeling and reasoning tasks into a single platform. The query $P(X|Y)$ posed to a BN $\mathcal{B}$ is represented by DN $\mathcal{D}' = \{p(X, Y)\}$ and the test

of independence $I_{\mathcal{B}}(X, Y, Z)$ holds in a BN $\mathcal{B}$ if and only if tha adaptation $A(\mathcal{P}_X, \mathcal{P}_Y, \mathcal{P}_Z)$ succeeds in the DN $\mathcal{D}$ for $\mathcal{B}$.

2.2.1 Definitions

TRAIT A *trait* t can be combative or docile. A *combative* trait t_c is depicted by a clear (white) circle. A *docile* trait t_d is illustrated by a dark (black) circle.

POPULATION A *population* $p(C, D)$ contains a non-empty set CD of traits, where C and D are disjoint, C is exclusively combative, and D is exclusively docile. A population is depicted by a closed curve around its traits.

DN A *Darwinian network* (DN) , denoted $\mathcal{D}$, is a finite, multiset of populations. A DN $\mathcal{D}$ is depicted by a dashed closed curve around its populations. All combative traits in a given DN $\mathcal{D}$ are defined as $T_c(\mathcal{D}) = \{t_c \mid t_c \in C, \text{for at least one } p(C, D) \in \mathcal{D}\}$. All docile traits in $\mathcal{D}$, denoted $T_d(\mathcal{D})$, are defined similarly

2.2.2 Operations

DOCILIZATION *Docilization* of a DN $\mathcal{D}$ adds $p(\emptyset, D)$ to $\mathcal{D}$, for every population $p(C, D)$ in $\mathcal{D}$ with $|D| > 1$.

DELETION To *delete* a population $p(C, D)$ from a DN $\mathcal{D}$ is to remove all occurrences of it from $\mathcal{D}$.

MERGE Two populations *merge* together as follows: for each trait t appearing in either population, if t is combative in exactly one of the two populations, then t is combative in the merged population; otherwise, t is docile.

NATURAL SELECTION In adaptation, *natural selection* removes recursively all barren populations from a DN $\mathcal{D}$ with respect to another DN $\mathcal{D}'$.

REPLICATION *Replication* of a population $p(C, D)$ gives $p(C, D)$, as well as any set of populations $p(C', D)$, where $C' \subset C$.

2.2.3 Modeling

In DNs, how populations "adapt" to the deletion of other populations corresponds precisely with testing independencies in BNs.

ADAPTATION Let $\mathcal{P}_X$, $\mathcal{P}_Y$, and $\mathcal{P}_Z$ be pairwise disjoint subsets of populations in a DN $\mathcal{D}$ and let DN $\mathcal{D}' = p(C)$, where $C = T_c(\mathcal{P}_X \mathcal{P}_Y \mathcal{P}_Z)$. The test *adaptation* of $\mathcal{P}_X$ and $\mathcal{P}_Z$ given $\mathcal{P}_Y$, denoted $A(\mathcal{P}_X, \mathcal{P}_Y, \mathcal{P}_Z)$, in $\mathcal{D}$ with four simple steps: (i) let natural selection act on $\mathcal{D}$ with respect to $\mathcal{D}'$, giving $\mathcal{D}^s$; (ii) construct the docilization of $\mathcal{D}^s$, giving $\mathcal{D}^s_m$; (iii) delete $p(C, D)$ from $\mathcal{D}^s_m$, for each $p(C, D)$ in $\mathcal{P}_Y$; and, (iv) after recursively merging populations sharing a common trait, if there exists a population containing both a combative trait in $T_c(\mathcal{P}_X)$ and a combative trait in $T_c(\mathcal{P}_Z)$, then $A(\mathcal{P}_X, \mathcal{P}_Y, \mathcal{P}_Z)$ fails; otherwise, $A(\mathcal{P}_X, \mathcal{P}_Y, \mathcal{P}_Z)$ succeeds.

2.2.4 Inference

EVOLUTION The *evolution* of a DN $\mathcal{D}$ into a DN $\mathcal{D}'$ occurs by natural selection removing recursively all barren, independent, and spent populations, merging existing populations, and replicating to form new populations.

3 Darwin Library

Darwin is a simple library written in Python for BN modeling and inference. The main purpose of the library is teaching BNs and quick prototyping or testing small networks. In order to achieve these goals, the library has a simplistic approach of implementation, using native Python data structure and light usage of object oriented programming. The structure of Darwin is basically divided into two categories: potentials and graph manipulations. The features are also split into two main categories: inference and modeling tools. Later, in this section, we also present a guide for getting started using Darwin.

3.1 Structure

In order to work with BNs, Darwin has two main set of implementations: potential and graph manipulations. These two categories form individually or together the tree structure of the library as presented in Figure 3.1.

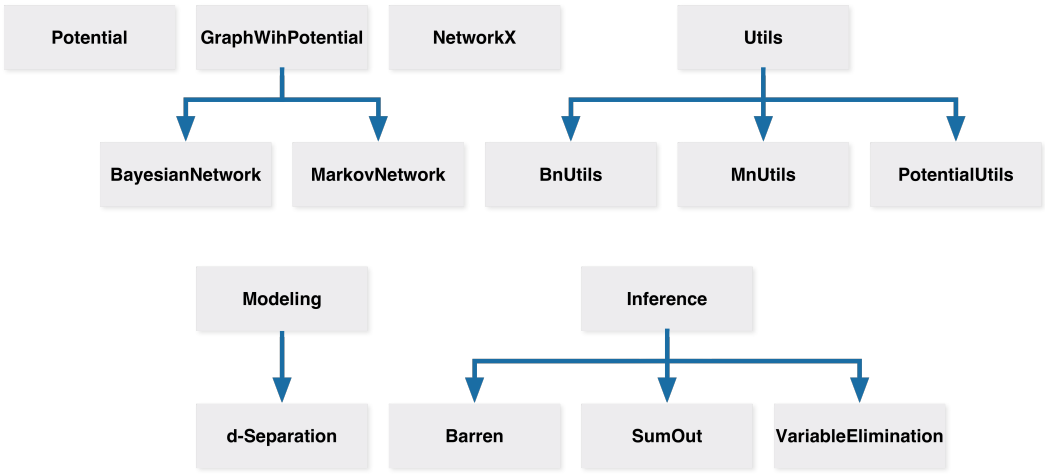

Fig. 3.1: Three structure of Darwin.

At the root, the class *Potential* is a data structure for modeling a probability table. The *NetworkX* [3] is set of data structures for graph manipulation and is also included at the root of the library. Similarly, *GraphWithPotential* is a class which basically maps nodes in a graph with a set of potentials, besides providing manipulations on the graph and the potentials on it. *BayesianNetwork* is a class that inherits from *GraphWithPotential* but maps only one potential per node. This data structure is used for modeling a BN. In the same way, the class *MarkovNetwork* entirely inherits the behaviour and properties of *GraphWithPotential*, therefore it can be used for MN modeling.

The *Utils* branch is formed by a set of standalone functions which implement core procedures for the other classes. Mainly, these functions are defined by numerical operations on lists of probabilities. One advantage of having those implementations as standalone functions is that future optimized code can be included or modified without disturbing the other classes, since the other classes only makes a function call to those utilities functions. Basically, there are three main utilities: *BnUtils*, *MnUtils*, and *PotentialUtils*. The BnUtils has a set of functions for BN manipulations and operations, such as moralization, triangulization, join tree construction, among others. In MnUtils, we have utilities for MN propagation such as finding an optimal path for propagating in a tree. Lastly, PotentialUtils is formed by a set of procedures for numerical operations such as multiplication, division and marginalization of tables. All potential manipulations in PotentialUtils is implemented according to the efficient implementation proposed in [5].

In the *Modeling* brach, there is an implementation of d-Separation as proposed in Algorithm 3.1 of [5]. While in the branch *Inference* there are few data structures useful for exact inference in BNs. The *Barren* function is used for identifying barren potentials in factorizations. *SumOut* is a function which systematically removes a set of variables from a factorization by multiplying potentials with the variables and them marginalizing the variables out. The class *VariableElimination* implements the exact inference algorithm VE as originally proposed by [9]. Finally, the *Test* branch has a set of unit tests which assure the correct functioning of core functions in the whole library.

3.2 Features

Here, we highlight some feature of the library. In general, the features are tools to facilitate the use of Darwin in teaching and prototyping of small system. The library is not intended for fast inference, neither high accuracy. Therefore, all numerical operations are implemented using native Python code, instead of high performance libraries such as *numpy* [8]. The graph manipulations are done by an external library called *NetworkX*, a robust and well known library with high performance and large set of tools.

For modeling, Darwin has the testing of d-Separation implemented using a reachability algorithm. Also, the library has built in tools for converting a BN into a MN, including the join tree construction by moralization, triangulization and the assignment of potentials. The triangulization step, specifically, has implemented 4 different heuristics, the same as implemented in *PgmPy* [1].

For inference, the library provides a basic function for eliminating variables in a factorization, called *SumOut*. But for faster inference, it is recommended to use the VE implementation which absorb evidence, removes barren and independent by evidence potentials, perform inference by summing out non relevant variables and, finally, normalize the final result.

[1] http://pgmpy.org

3.3 Usage

In order to get started with Darwin, we now present a quick overview of the most common classes and functions. The main classes are *Potential*, *GraphWithPotential*, *BayesianNetwork*, and *MarkovNetwork*.

The Potential class is defined by the given arguments: *variables* which is a list with strings, *cardinalities* corresponding to the variables which is a list with integers in the same order than the variables, probabilities *values* in a list with floating numbers, *left hand side* which is a list with the variables in the LHS of the potential, and similarly the *right hand side* is defined. For example, considering a CPT $P(a|b)$ with binary variables and probability values 0.4, 0.5, 0.6, 0.5, we can use Darwin to represent this potential as:

```
Potential(["a", "b"], [2, 2], [0.4, 0.5, 0.6, 0.5], ["b"], ["a"])
```

The GraphWithPotential class contains basically a list with potentials, a graph defined using NetworkX, and a dictionary mapping nodes in the graph to a list of potentials. After declaring a GraphWithPotential, the user can add potentials to a node using the *add_potential* method. For example, the following code creates a graph with two nodes $\{a\}$ and $\{a, b\}$ and assign $P(a)$ to $\{a\}$ and $P(b|a)$ to $\{a, b\}$ in a GraphWithPotential.

```
p1 = Potential(["a"], [2], [0.2, 0.8], ["a"], [])
p2 = Potential(["a", "b"], [2, 2], [0.4, 0.5, 0.6, 0.5], ["b"], ["a"])

G = networkx.Graph()
G.add_nodes(["a", "ab"])

GwP = GraphWithPotential()
GwP.add_potential("a", p1)
GwP.add_potential("ab", p2)
```

The MarkovNetwork inherits from GraphWithPotential, therefore modeling a MN is simply using the GraphWithPotential like in the code above. In the same way, modeling a BN uses the BayesianNetwork class which also inherits from GraphWithPotential. The difference for a BN is that only one potential is assigned at one node and the graph used is a DAG. For instance, the code above can be used to declare a BN with two nodes $a \rightarrow ab$ just by changing the definition of G to the below code, which is a directed graph.

```
G = networkx.DiGraph()
G.add_edge("a", "ab")
```

4 Darwinian Network Library

4.1 Structure

The DN library is a simple extension of Darwin, also intended for teaching and prototyping purposes. Basically, the library inherits the potential manipulation tools from Darwin, ignoring the graphical manipulation ones. From there, DN library builds the analysis of set of potentials, which forms a DN by definition. The structure of the library is mainly formed by two classes: *Population* and *DarwinianNetwork*.

The Population class inherits directly from Potential's class in Darwin. But here, few keywords are override in order to adapt the language for DNs. For instance, the left and right hand side of the arguments in a Potential are called combative and docile in a Population. Moreover, a Population have the methods merge and replicate which internally calls multiply (or divide) and marginalize from Potential. Deciding where it is a division or multiplication is handled internally by the merge method.

DarwinianNetworks is defined with a set of Populations. This data structure basically offers common methods for set manipulations, for instance, inserting and removing Populations. Here, the set of populations are stored in an array, in order to allow the multiset populations as defined in DNs.

4.2 Features

Few features are present in DN library in order to guarantee a smooth use for teaching and prototyping. Now, we highlight there features of the library: keyword usage, initialization of parameters and drawing of data structure.

When instantiating a Population, the user can use keywords for the combative and docile arguments. In this way, the user does not need to remember the ordering of the arguments, but only their names. For example, defining a population $p(a, b)$ can be defined with the below code by using keywords "combative" and "docile":

```
Population(combative=["a"],docile=["b"])
```

Also for simplifying usage, the library has standard initialization of internal parameters. That is, if the user does not pass all the required arguments, the DN library initialize them correctly when possible with standard values. For example, in the above code the cardinalities of

variables “a” and “b” are set to 2 and the 4 probabilities values are randomly generated between 0 and 1.

Finally, the DN library has a set of procedures for drawing Populations and DarwinianNetworks whenever the user calls them. Those tools are built using the *matplotlib* [4] library for Python and is the only requirement for the drawings. If the user wants to see how a population looks like, the user can define the Population and then calls the *draw_population* function as illustrate below:

```
p = Population(combative=["d"],docile=["f", "b"])
draw_population(p)
```

These commands will create a drawing of a population as illustrated in Figure 4.1.

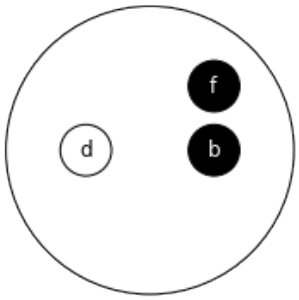

Fig. 4.1: Drawing of population $p(d, fb)$ created by the DN library.

The same idea works for a whole DN. After defining a DarwinianNetwork, the user can call *draw_dn* passing the DN object. Figure 4.2 shows one example of the drawing of a DN as generated by the DN library.

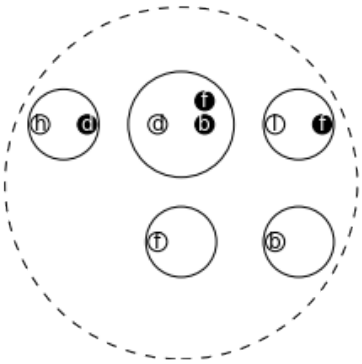

Fig. 4.2: Drawing of the DN $\{p(h, d), p(l, f), p(f), p(b), p(d, fb)\}$ created by the DN library.

4.3 Usage

The use of the two main classes in DN library is introduced now. The Population class is defined exactly like a Potential, since it inherits from that class. But the left and right hand side argument names are replaced by combative and docile, respectively. Two methods are also exposed for quick references: combative and docile, which returns the respective informations for that Population. A helper method called *from_potential* is also available in order to convert a Potential object to a Population one. Moreover, merge receives another Population as argument, while replicate receives a list of variables which will be marginalized from the Population.

The DarwinianNetwork class has a list member where all the Populations are hold. The methods *add_population* and *delete_population* are used for adding and deleting Population, respectively, from the list of Populations This class has no constructor and the list of Populations is used to simulate the multi-set definition of DNs.

5 Conclusion

We proposed Darwinian Network Library as new framework for modeling and inference in DNs. DN library is a implementation of the basic operations for adaptation and evolution in DNs. It is implemented with the Python programming language and the basic data structures are provided by Darwin library. The main advantage of the DN library is to apply novel ideas and techniques of DNs. DN library also is great for teaching, quick prototyping with DNs and testing.

DN library is based upon the novel library Darwin, a Python framework for BN modeling and inference. Darwin library has two main categories called potentials and graph manipulations. The former is a data structure for modeling a probability table. The latter maps nodes in the graph to a list of potentials. Together they form a reliable structure to reason on BNs.

We have established in Chapter 4 features of DN library given the manipulation tools from Darwin. We have shown that DN library is an intuitive framework for working with DNs. For instance, to build a set of potentials, one can simply utilize the class *Population*. Intuitively, with objects *Population* together we can obtain a class *DarwinianNetwork*.

Another salient feature is the set of procedures for drawing Populations and DarwinianNetworks. It allows the user to see how looks like a population that has just been created - or even the entire DN!

This project proposes the implementation of a DN library. In summary, we have stablished four main advantages of using DN library: (i) due its simple approach given the Python implementation, it has an expressive language that facilitate its usage; (ii) it has interactive visual tools to compute and draw DNs, enabling users to better interact with the library; (iii) it is a great tool for learning and its operations are sound given the unit tests; (iv) and all source code is available free online on *GitHub* through the webpage:

`https://github.com/Darwinian-Networks`

With DN library, DNs can be applied as the simple and yet remarkably robust tool they are, allowing users to simplify reasoning with BNs.

Bibliography

[1] Butz, C.J., Oliveira, J.S., dos Santos, A.E.: Darwinian networks. In: Proceedings of the Twenty-Eighth Canadian Artificial Intelligence Conference. pp. 16–29 (2015)

[2] Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)

[3] Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008). pp. 11–15. Pasadena, CA USA (Aug 2008)

[4] Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing In Science & Engineering 9(3), 90–95 (2007)

[5] Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. MIT Press (2009)

[6] Lauritzen, S.L., Spiegelhalter, D.J.: Local computation with probabilities on graphical structures and their application to expert systems. Journal of the Royal Statistical Society 50, 157–244 (1988)

[7] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann (1988)

[8] Van Der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: a structure for efficient numerical computation. Computing in Science & Engineering 13(2), 22–30 (2011)

[9] Zhang, N.L., Poole, D.: A simple approach to Bayesian network computations. In: Proceedings of the Tenth Canadian Artificial Intelligence Conference. pp. 171–178 (1994)