

Banjo



Bayesian
Network Inference
with
Java
Objects, Version 2.2

User Guide

By Jürgen Sladeczek, Alexander J.
Hartemink, and Joshua Robinson

**Banjo is licensed from Duke University.
Copyright © 2005-2008 by Alexander J. Hartemink.
All rights reserved.**

=====
Licensing Overview
=====

You may license this software either under the non-commercial use license shown below or under a specially-negotiated non-exclusive commercial use license. You may choose which type of license is more appropriate for your needs. For strictly non-commercial use of the software, you may prefer to license the software under the non-commercial use license below. The term 'commercial use' is defined broadly: if the software is used for commercial gain or to further any commercial purpose, a commercial use license is required. If you have any question about whether your use would be considered commercial, or if you would like to negotiate a non-exclusive commercial use license, please contact us. Our addresses in April 2008 are:

Alexander J. Hartemink, Ph.D.
Assistant Professor
Department of Computer Science
Duke University
Box 90129
Durham, NC 27708-0129
amink@cs.duke.edu

Henry Berger, Ph.D.
Licensing Director for Pratt School of Engineering
Office of Licensing & Ventures
Duke University
Box 90083
Durham, NC 27708-0083
hrberger@duke.edu

=====
Non-Commercial Use License Agreement
=====

IMPORTANT NOTICE: PLEASE READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE PROCEEDING TO USE THE SOFTWARE.

1. DEFINITIONS:

"Licensor" means Duke University.

"Licensee" means You, if you accept the terms of this Agreement.

"Authors" means Alexander J. Hartemink and Jürgen Sladeczek.

"Software" means the Banjo (Bayesian Network Inference with Java Objects) software package, including executable and source code versions, and any subsequent upgrades, updates, or modifications to Banjo provided by Authors or Licensor.

"Commercial Use" means any attempt, whether intentional or not, to copy, use, modify, or distribute, in whole or in part, Software or related documentation for financial or commercial gain, or to further the aims of any company, including but not limited to use of Software in the research division of a company, use of Software to improve a business or financial model, production of derivative works based on Software that will

be sold or offered for sale, incorporation of Software into a product or collection of products that will be sold or offered for sale, or distribution of Software commercially.

2. LICENSE GRANT. Licensor grants to Licensee a non-exclusive, worldwide, royalty-free, perpetual, non-transferable, single-user license to copy, use, modify, and distribute executable and source code versions of Software and related documentation for any use that is not Commercial Use. Licensee is free to make modified versions of Software, provided that executable and source code versions of such derivative works are made readily available to others on these same terms, without fee or any other charge. Unmodified Software or related documentation should be distributed by providing a link to Licensor's Software website (<http://www.cs.duke.edu/~amink/software/banjo/> is the URL in April 2008). This license does not entitle Licensee to any installation support, technical support, telephone assistance, maintenance, or enhancements, modifications, or updates to Software.
3. RESTRICTIONS. Except as otherwise expressly permitted in this Agreement, Licensee may not (i) sell, rent, lease, or sublicense rights in Software; (ii) remove or alter any trademark, logo, copyright, or other proprietary notices, legends, symbols, or labels in Software or related documentation; or (iii) use the name of Licensor or Authors in any manner related to Software without their prior written permission.
4. FEES. Software is provided without fee, provided that Licensee's use is not Commercial Use. If Licensee is interested in Commercial Use of Software, Licensee shall contact Licensor to negotiate a non-exclusive commercial use license, which may include a license fee. If Licensee fails to negotiate such a license, but nevertheless uses Software for any Commercial Use, Licensee shall be deemed in material breach of this Agreement and agrees to pay monetary damages to Licensor, compensatory and possibly punitive. In the event of a successful action brought to enforce this provision, Licensor shall be further entitled to recover reasonable attorney's fees and costs.
5. TERMINATION. License and the rights granted hereunder shall terminate automatically if Licensee breaches any of the terms or conditions of this Agreement, unless Licensee receives written permission from Licensor to exercise rights under the license in spite of a breach. Upon termination, Licensee agrees to destroy all copies of Software and related documentation, including copies made for backup purposes.
6. PROPRIETARY RIGHTS. Software and related documentation constitute published works and are protected by copyright and other intellectual property laws and by international treaties. All rights, title to, and ownership interest in Software, including all intellectual property rights therein, belong to and shall remain with Licensor. Licensee acknowledges such ownership and intellectual property rights and agrees not to take any action that jeopardizes, limits, undermines, or in any manner interferes with Licensor's ownership and intellectual property rights with respect to Software.
7. MANDATORY NOTICE. Both the notice below and the full terms of this Agreement shall be embedded in any location or medium in which Software or related documentation is stored, copied, or reproduced, and shall be loaded into computer memory for use, display, or reproduction in any copy of Software or related documentation, including derivative works. The notice to accompany the full terms of this Agreement shall state:

"Banjo is licensed from Duke University.
Copyright (c) 2005-2008 by Alexander J. Hartemink.
All rights reserved."

8. **DISCLAIMER OF WARRANTY.** SOFTWARE IS PROVIDED IN ACCORDANCE WITH THE TERMS AND CONDITIONS OF THIS AGREEMENT ON AN "AS IS" BASIS. LICENSOR MAKES NO REPRESENTATIONS OF AND SPECIFICALLY DISCLAIMS WARRANTIES OF ANY KIND, EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING BUT NOT LIMITED TO WARRANTIES THAT SOFTWARE IS MERCHANTABILITY, FIT FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, ACCURATE, OR FREE FROM DEFECTS, WHETHER DISCOVERABLE OR NOT. LICENSEE BEARS THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF SOFTWARE. SHOULD SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, LICENSEE ASSUMES SOLE RESPONSIBILITY AND LIABILITY FOR THE ENTIRE COST OF ANY SERVICE AND REPAIR IN CONNECTION THEREWITH. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT. NO USE OF SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

9. **LIMITATION OF LIABILITY.** IN NO EVENT AND UNDER NO CIRCUMSTANCES WILL LICENSOR BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, INCIDENTAL, PUNITIVE, OR EXEMPLARY DAMAGES OF ANY KIND WHATSOEVER ARISING OUT OF THE USE OF OR INABILITY TO USE ANY PORTION OF SOFTWARE, INCLUDING BUT NOT LIMITED TO DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER DAMAGES OR LOSSES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF, AND WITHOUT REGARD TO WHETHER SUCH CLAIM OR ALLEGATION IS BASED IN CONTRACT, TORT, OR ANY OTHER LEGAL OR EQUITABLE THEORY. TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, LICENSOR'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS AGREEMENT SHALL NOT EXCEED IN THE AGGREGATE THE SUM OF THE FEES LICENSEE PAID FOR THIS LICENSE (IF ANY).

10. **MISCELLANEOUS.** (a) This Agreement constitutes the entire Agreement between the parties concerning the subject matter hereof. (b) This Agreement may be amended only by mutual written agreement, signed by both parties. (c) Except to the extent applicable law, if any, provides otherwise, this Agreement shall be governed by the laws of the State of North Carolina, U.S.A., excluding its conflict of law provisions. (d) This Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods. (e) If any provision in this Agreement should be held invalid or unenforceable by a court having jurisdiction, such provision shall be modified to the minimum extent necessary to render it enforceable without losing its intent, or severed from this Agreement if no such modification is possible, and other provisions of this Agreement shall remain in full force and effect. (f) A waiver by either party of any term or condition of this Agreement or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof. (g) The provisions of this Agreement which require or contemplate performance after the termination of this Agreement shall be enforceable notwithstanding said termination. (h) Licensee may not assign or otherwise transfer by operation of law or otherwise this Agreement or any rights or obligations herein. (i) This Agreement shall be binding upon and shall inure to the benefit of the parties, their successors, and permitted assigns. (j) The relationship between Licensor and Licensee is that of independent contractors and neither Licensee nor its agents shall have any authority to bind Licensor in any way. (k) The headings to the sections of this Agreement are used for convenience only and shall have no substantive meaning.

Contents

What is New in Version 2	1
Introduction.....	5
Getting Started	6
Requirements.....	6
Downloading the Banjo Files	6
Quick Start	7
Using Your Own Data.....	8
Searching Using Multiple Threads	8
Searching Using a Compute Cluster	9
Supported Data Formats.....	10
Sample Output and Screenshots.....	11
Example: Searching for the “Best” Static Bayesian Network	11
Example: Searching for the “Best” Dynamic Bayesian Network	16
Using Banjo	23
The Banjo Application.....	23
The Banjo Components	24
The Searchers	24
The Proposers	25
The Cycle Checkers.....	25
The Evaluators.....	25
The Deciders.....	26
The Equivalence Checkers.....	26
Summary of Component Options.....	27
Setting up a Banjo Search	28
Tuning the Memory Use	28
Performance Tuning.....	29
Input Discretization Options.....	30
Experimenting with Banjo	32
Example: Choice of Discretization.....	32
Example: Combinations of useCache, MaxParentCount, and precomputeLogGamma.....	33
Example: Varying the Cache Level in an Intermediate-size Problem	35
Example: Varying the Cache Level in a Large-size Problem	35
Example: Comparing Searchers.....	36
Example: The Effects of Equivalence Checking on Performance	37
Example: Comparing different Cycle Checking Methods.....	37
Example: Cycle Checking Methods, Revisited.....	38
Advanced Features.....	39
Post-Processing Options.....	39
Using <i>dot</i> to Generate a Graph Representing the Found Network.....	39
Influence Scores.....	42
Consensus Graph.....	43
Finding Non-equivalent Networks	45
Using Banjo in Matlab	47
Hints and Tips	49
Computing a Network Score without Running a Search	49
Displaying Debug Info.....	49
Adding Structure Files to Output.....	49
Using Time Stamps in Output Files	49
Memory Info and Performance Tuning	49

Accessing Additional Options via Internal Code Changes.....	50
Unique Output File Names.....	50
Combining Multiple Observations Files	51
Error Reporting to File	51
More Flexible Structure Files.....	51
Specifying Observations in Row or Column Format.....	51
Specifying Names for the Variables	52
Using a Greedy Searcher with the AllLocalMoves Proposer.....	52
Troubleshooting	53
When Little Things Don't Work As Expected.....	53
Missing or Invalid Parameter in the Settings File	53
Miscellaneous Errors and Warnings	55
Problems During Post-Processing	55
Error During Program Execution: Handled by Banjo	55
Error During Program Execution: "Insufficiently Handled" by Banjo	56
Running Out of Memory.....	57
Parameters Affecting Memory Use	57
The First Thing to Check.....	57
What if Banjo still Runs out of Memory?.....	58
What if Banjo Runs out of Memory well into a Search?.....	59
Crash with "System" Error Message.....	60
Submitting an Error Report	61
Appendix A.....	62
File Formats.....	62
Example of a Minimal Settings File	62
Example of a Comprehensive Settings File	63
Observations File Example	65
Structure File: Static Bayesian Network.....	65
Structure File: Dynamic Bayesian Network.....	66
Results Output in XML Format.....	67
Appendix B	69
Settings File: Parameter Names and Values.....	69
Appendix C	79
References.....	79
Papers.....	79
Books and Manuals	79
Appendix D	80
Project Background and Acknowledgements	80
Index.....	81

What is New in Version 2

For the release of Banjo Version 2, we focused our development efforts on four areas:

- We dramatically reduced the memory requirements for running Banjo, which in turn resulted in significantly improved search performance, by as much as an order of magnitude for problems with thousands of variables.
- We added a number of new features, such as automatic dot graphics file creation, consensus graph computation, and equivalence checking for the computation of the n-best graphs.
- We improved the overall ease-of-use with more convenient input and output handling, as well as widely expanded control over various facets of the application's execution.
- We changed the underlying framework, especially in the handling of settings, their validation, and error reporting, to make the infrastructure part of development easier, more consistent, and more efficient.

In more detail, here is a complete list of improvements.

Performance:

- For large problems with several thousands of variables, the memory required for executing a search is reduced by more than 10-fold, and the newfound ability to use Banjo's extended caching options for such problems reduces search times by a similar factor. Search performance for small problems (with less than 100 variables) is up to twice as fast as Banjo 1.0.
- An improved cycle checking implementation with optional optimization based on a paper by Oded Shmueli further increases the search speed.

New features:

- Consensus graph for n-best networks.
- Equivalence comparison for n-best networks (with statistics). When executing a search, the tracking of the top-scoring networks can be done using 3 different settings: "nonIdentical" works the way Banjo 1.0 did, namely without any equivalence checking; "nonIdenticalThenPruned" prunes the set of non-identical networks down to a set of non-equivalent ones, but only after the search is completed; and "nonEquivalent" collects only non-equivalent networks as it maintains its list of top-scoring networks during the search.
- Output of dot files (top and consensus graphs), and automatic execution of *dot* to generate graphics files (in user-specified formats).
- Stopping criteria can be based on any combination of time, networks, and restarts. The first criterion to be satisfied will cause the search to stop.
- User can specify labels for the variables. These labels will be used for dot output.
- Additional functionality is now under user control via the settings file, with default values for unspecified settings. This includes various post-processing options such as `createDotOutput`, `computeInfluenceScores`, and `computeConsensusGraph`. If the `fullPathToDotExecutable` setting is correctly specified, then Banjo will automatically launch the *dot* application after the search to create the graphic representation of the obtained network (the top scoring and/or the consensus one). The output of the data report, and the

display of specified structures (namely, for the initial, must-be-present, and must-not-be-present parents) can now also be controlled via settings.

- Input of time values can now be done in 3 valid ways: a single number (interpreted as seconds), a single number with a time qualifier (e.g., h for hour), or in the colon-delimited format that we used in the past. Note that the colon-delimited format does not require “leading zeros” any more (e.g., 10:00 is equivalent to 0:0:10:00, or to 10 m, or to 600, all resulting in 10 minutes of search time).
- The feedback of results has been streamlined via the new `fileReportingInterval` and `screenReportingInterval` settings, which are being specified as time values. The feedback display itself is more uniform and informative.
- For keeping results organized, a time stamp can be embedded in the names of output files.
- Users with large data sets may appreciate the settings-based control over the internal caching mechanism, as well as the optional memory usage information. Should Banjo run out of memory, it now exits gracefully with a final report on memory use and a listing of the results obtained before running out of memory.
- And when things don’t go as expected, a new setting tells Banjo to display the stack trace in run time mode, to provide the maximum amount of information in case of an internal problem or bug.

Ease-of-use improvements:

- Optional settings can either be omitted entirely from the settings file, or can have their values omitted.
- A number of optional “settings” that used to be controlled via internal constants can now be configured via the settings file.
- The spelling of setting names in the settings file is now case-independent.
- Observations and structures can be supplied using arbitrary white-space, or using different delimiters (‘:’ are implemented); even more generalization can be applied by setting a custom pattern in the global setup.
- Observation and structure files can include descriptive comments (any text that follows a # in any text line is now being ignored).
- Validation is performed as much as possible in a single pass (i.e., as long as no “show-stopper” error is encountered, errors are collected and then reported all at once, instead of one error at a time). In addition, non-fatal errors are collected separately, and displayed at the end of the feedback section for the search.
- When a network that is provided to Banjo (i.e., an initial or mandatory structure) contains a cycle, the resulting error message will name a node in the cycle, making it easier for the user to correct the input.

Development-related changes:

- The handling of settings and validations has changed substantially. As the number of options for controlling Banjo’s behavior grew, the centralized management of its options would be virtually impossible to maintain due to the increasingly complex logic required. Thus, settings and their validation are now handled within the object that uses a particular setting. This makes intuitive sense, since each object can be assumed to know what conditions it needs to impose on a setting’s data type and value. As a benefit of this approach any setting that is shared between multiple objects can now be validated by any number or all of such objects, each imposing its own set of restrictions.
- Hand in hand with our new validation handling, we added a more granular tracking of the results (i.e., errors and warnings) of the validation. Instead of immediately throwing an error when an unacceptable input value is encountered, we now try to record as many errors as we can within our validation pass, so that we can provide more comprehensive

feedback in a single message. Of course, there are instances of dependencies between settings, where we cannot continue once we encounter a “show stopper” error.

- If you have used Banjo 1.0, and have made changes to the source code, please review the Banjo Developer Guide for a more detailed discussion of the changes.

Of course, Banjo Version 2 contains the corrections for the (four) bugs from the Version 1.0.x maintenance releases.

Finally, we updated the user guide to reflect the changes that went into Banjo Version 2. Based on direct and indirect user feedback, there is now an extensive “Experimenting with Banjo” section to help the users navigate the at times tricky trade-off between performance and memory use based on the multitude of settings that Banjo affords for steering the underlying search process.

New Features added in Version 2.1

In December 2007, Banjo Version 2.1 has added parallel execution capabilities to the application: based on the value of the *threads* setting, Banjo will now execute multiple searches in parallel on a multi-processor machine. By design the multiple threads share a single copy of the observations, and – if used – the fast cache and the pre-computed log-Gamma values. Also by design, the “regular” cache is not shared between the threads based on the heuristic that each thread may visit very different regions of the search space.

When running a search with the n-best networks settings greater than 1, each of the threads will maintain its own set of n-best networks. After completion of the search these sets of networks are then combined by the thread “controller” into a single, combined results set of n-best networks.

While executing the parallel search we save each thread’s results to individual files, specified by a prefix to the results file name. The final combined results are then saved to the single, originally specified results file. Note that tracked data such as intermediate results and search statistics are written (only) to the individual files. On the other hand, only the combined results file contains the n-best networks and any post-processing results.

New Features added in Version 2.2

In April 2008, Banjo Version 2.2 has added capabilities for convenient search execution in a cluster environment. The conduit for this feature is the introduction of a (optional) results file in XML format, together with Banjo’s new functions for reading and combining a set of such results files into a single results file.

When executing a cluster-based search, one simply collects the produced output files (in the newly introduced XML format), then executes Banjo while pointing it to the directory with the results files: As a final output, Banjo will produce a single output file with the combined n-best networks from all the searches.

As part of the implementation of the new features, new classes for handling the XML processing as well as for handling the sets of n-best networks have been added in the utilities package. In addition, the code for handling wildcard file processing has been extracted as a class of its own. (For details please refer to the Banjo Developer Guide)

As an aide to testing and error tracking, we added a new (optional) setting called *seedForStartingSearch*, which can be used to generate repeatable random number sequences.

Finally, the error handling and error message feedback has received an overhaul: the error message is now included in the regular report file; the stack trace is included in the error message by default; and the seed number is included to make it easier to reproduce a scenario.

Introduction

Banjo is a software application and framework for structure learning of static and dynamic Bayesian networks, developed under the direction of Alexander J. Hartemink in the Department of Computer Science at Duke University. Banjo was designed from the ground up to provide the performance for analyzing large, research oriented data sets, while at the same time being accessible enough for students and researchers to explore and experiment with the algorithms. Because it is implemented in Java, the framework is both powerful and easy to maintain and extend.

Banjo focuses on structure inference methods; for inference within a Bayesian network of known structure, a plethora of existing code and applications are available. Banjo currently performs structure inference for static and dynamic Bayesian networks using the Bayesian Dirichlet (BDe) scoring metric for discrete variables; available search strategies include simulated annealing and greedy, paired with evaluation of a single random local move or all local moves at each step. A search algorithm in Banjo consists of a set of individual core components:

- Proposing a new network (or networks), handled by a “proposer” component,
- Checking the proposed network(s) for cycles, handled by a “cycle checker” component,
- Computing the score(s) of the proposed network(s), handled by an “evaluator” component, and
- Deciding whether to accept a proposed network, handled by a “decider” component.

These core components are organized and implemented in such a way that they can be used to study or extend the search algorithms themselves: a set of (easily expandable) statistics is provided for monitoring the actual search process.

The core algorithms assume and have been optimized for discrete variables, but if some of your variables are continuous, the current version of Banjo provides simple discretization functionality using either quantile or interval discretization methods. Any number of highest scoring networks can be retained in the search, and the highest can be processed by Banjo to compute influence scores on the edges, or to generate a file formatted for rendering with *dot*, a graph layout visualization tool from AT&T.

This **Banjo User Guide** explains how to install and run Banjo, the parameters that the program requires, the names and formats of the data files that it uses, and how to put all the different pieces together to flexibly use Banjo with your own data. It is organized into the following main sections:

- The *Getting Started* section provides instructions on how to obtain and set up the Java executable and the source code, and how to execute the code on your computer.
- The *Using Banjo* section describes the high-level organization of the application, with a brief explanation of Banjo’s core components.
- The *Troubleshooting* section describes the types of issues that a user might encounter when running the Banjo code, and how to try to resolve them.
- Finally, appendices provide examples of the file formats that Banjo uses, an annotated list of available parameter settings, and examples of error messages.

Getting Started

The installation instructions for Banjo are the same whether you simply want to use the code, (i.e., run the application with your own data), or whether you also intend to modify the actual source code to create your own applications. The sections below describe the details for downloading and setting up Banjo on your system.

Requirements

The main prerequisite for using Banjo on a particular computer or operating system is the availability of a Java virtual machine (JVM). The code has been written to comply with Java 2, using the 1.4.2 or later release of the Java SDK.

We have successfully run Banjo on PCs running Mac OS X, Red Hat Linux, Sun Solaris, and Windows (98, 2000, and XP), with available memory of at least 256 MB. If your system does not have a JVM installed, you can download one for free from <http://www.java.com> (the Java Runtime Environment) or <http://java.sun.com> (the Java SDK, needed only if you intend to make changes to the Banjo code).

The amount of the memory available to the JVM will determine the size of the data set that Banjo can handle. In Banjo 2 the memory requirements are substantially reduced, so that a data set with 3000 variables, 2000 observations, and a fastCache setting of level 1 (i.e., all 0 and 1 parent configurations are cached in a special, non-hashed, cache) can now be tackled in about 500 MB of memory. If you intend to tackle large data sets, you may want to take a look at the section on memory management via the cache settings.

Any Java development environment, from simple text editor to full-featured IDE, can be used to modify the Banjo code. We use the open source Eclipse IDE (<http://www.eclipse.org>) to develop Banjo. The **Banjo Developer Guide** features a brief tutorial on how to modify Banjo within the Eclipse IDE, version 3.0. We are currently using version 3.1.x of Eclipse, which sports some interface and functional enhancements, but the setup instructions remain essentially the same.

Downloading the Banjo Files

Before downloading, please verify that your system has the required version of Java installed.

Banjo is available for download as a Java Archive file (`banjo.jar`) in a larger zip file (`banjo.zip`). As long as you have the required version of Java installed, you will be able to run Banjo from the command line without any need for compiling. The zip file also contains Banjo source files, documentation, license information, and two examples for performing Bayesian network structure inference. We recommend that you extract `banjo.zip` in a way that preserves the directory structure within the zip file. This way you can test Banjo without any additional configuration. The following is a manifest of the files within `banjo.zip`:

- `README.txt` a welcome, a basic description of the files, and information on how to test your whether your installation was successful

- `LICENSE.txt` an overview of how Banjo may be licensed, along with the full text of the Non-Commercial Use License Agreement; please read this carefully before proceeding to use Banjo
- `banjo.jar` a Java archive file containing all the compiled Banjo code, which you can use to run Banjo
- `data/release2.0` a directory structure containing two subdirectories with settings reflecting the changes in Version 2. One directory contains an example settings file and data for learning static Bayesian networks; the other similarly for learning dynamic Bayesian networks; these will help you understand how Banjo works (see below for information on how to run these two examples)
- `data/release1.0` a directory structure containing two subdirectories as distributed in version 1, one with an example settings file and data for learning static Bayesian networks; the other similarly for learning dynamic Bayesian networks; these will help you understand how Banjo works (see below for information on how to run these two examples)
- `doc/` a directory with documentation describing Banjo; within this folder are PDF versions of the Banjo User Guide, the Banjo Developer Guide, and a Javadoc directory containing a description of the Banjo class APIs in browseable HTML
- `src/` a directory containing Banjo's full Java source tree
- `template.txt` a template that can be filled in to create a settings file for using Banjo with your own data

Quick Start

For the following, we assume that you have a working Java VM installed and `banjo.zip` downloaded and unzipped. Here are the steps for “test-driving” the Banjo software:

- Open a command shell, and change directories to the directory where you expanded `banjo.zip` (and that now contains `banjo.jar`).
- To run the first test, type

```
java -jar banjo.jar settingsFile=data/release2.0/static/static.settings.txt
```

in your command shell, and press Enter. If your environment is set up properly, the Banjo application will start by printing out the parameters, progress, and then final results of the search, which is using simulated annealing to find a high scoring static Bayesian network (this is set to search for 1 minute).

- To run the second test, and to demonstrate an alternate syntax for specifying the location of the settings file, type

```
java -jar banjo.jar settingsFile=dynamic.settings.txt settingsDirectory=data/release2.0/dynamic
```

- To use the Banjo application with your own data, edit the settings template file called `template.txt`. Unlike the settings files in the previous two tests, this template file does not contain enough information to specify a valid search, so to make it work, appropriate values of the settings to describe your own data must be inserted. If you choose to save your settings file with the name `banjo.txt` and place it in the same directory as `banjo.jar`, you can then run Banjo with a shorter command line:

```
java -jar banjo.jar
```

By the way, Banjo lets you override any parameter from the settings file by appending it to the command line in the form `item=value`. A detailed description and additional information about parameter use in the settings file and from the command line is provided below. Feel free to experiment with different parameter combinations to discover their effects on the runtime performance of a particular algorithm.

Using Your Own Data

After your successful test drive of Banjo, you may want to try Banjo with your own data. Here are the steps to accomplish just that.

To prepare your data, you'll need a settings file that describes your data and how you want the inference to proceed, i.e., the number of variables, your choice of search algorithm, etc. You'll also need an observations file in a very basic format: Each row contains a single observation, with one entry for the value of each variable delimited by a tab between the values. Let's say that you saved your settings in `my.settings.txt` and saved your data as `my.data.txt`; the `observationsFile` setting in `my.settings.txt` should list `my.data.txt` as its value (select values for the remaining settings as you desire).

You can then run Banjo on your data with

```
java -jar banjo.jar settingsFile=my.settings.txt
```

You may want to save your data and associated settings file in a custom directory structure that keeps your data sets organized for easy later retrieval. Simply set the parameters for the file and directory information in the settings file to point to your structure.

Searching Using Multiple Threads

In Banjo version 2.1 we added support for executing our code on multi-processor equipped hardware. Simply add the `threads=N` setting with an integer value, and Banjo will automatically spawn N search threads.

```
java -jar banjo.jar settingsFile=my.settings.txt threads=15
```

All threads will use the same parameters as specified in the settings file. After all search threads have completed their execution, Banjo will combine the obtained results and provide them in a single output file.

Note that you want to match the number of threads to be used closely with the number of processors available. In some basic experiments we found that on a 16-core SUN machine we obtained better performance (as defined as the combined number of networks visited by all the threads) when selecting threads=15 than when matching the thread count to the number of CPUs. This is likely attributable to the role of the scheduler/controller thread that is being run by the OS in the background.

On commodity multi-core hardware that is becoming more and more standard for today's desktops, we found a performance increase of 1.8-1.9 fold (for dual core) to 3.2 fold (for quad core) as compared to a single thread. In the extreme case of our 16-processor hardware the performance increase was around 13 fold when we specified threads=15, compared to about 11 fold when we specified threads=16.

Searching Using a Compute Cluster

While each compute cluster may be set up a little different, the basic use is the same: one submits a (large) "batch" of jobs -- in our case Banjo searches -- to a queue for distributed execution, and "harvests" the results after the last job has completed its execution.

Previous versions of Banjo could already be used to execute distributed searches on a compute cluster as long as the target machines were supporting Java. Of course, the challenge that remained was how to efficiently manage the dozens if not hundreds of possible results files to combine them into a single, combined results file. So we added some basic features to Banjo version 2.2 to first export the search results in machine-readable, namely XML-based, format, as well as an easy way to "harvest" a large number of results files into a single combined results set.

For easy reference we provide a sample of the XML format used by Banjo in Appendix A.

The (optional) newly introduced settings for using a distributed search are XMLreportFile and XMLoutputDirectory, for specifying the name and location of the results file with the XML formatted data, and the XMLsettingsToExport setting, for listing those settings that we want to have recorded as part of the XML output.

```
reportFile = static.report.@TS@.txt
XMLreportFile = static.@TS@.xml
XMLoutputDirectory = data/static/output/xml
XMLsettingsToExport = all

XMLinputFiles = *.xml
XMLinputDirectory = data/static/input/xml
```

Currently the output to the XML files is limited to the core search information, namely the network and its score, and the settings related to the search. We do not export any of the search statistics to XML; the original report file is the only file that will contain such information.

It is possible to run multi-threaded searches in conjunction with a compute cluster approach. When we execute a multi-threaded search on any single machine, the results from those threads will be combined first into the overall results for that particular machine before the XML formatted data is written to file.

Once we have obtained a set of xml files from a set of searches, we run Banjo one more time, but now with the settings XMLinputFiles and XMLinputDirectory specified; in XMLinputFiles we list the names of the XML files that we want to combine into a single results set.

When specifying the XMLinputFiles, we can use the Banjo wildcard file-naming conventions, including the “-” prefix for files to be excluded from processing.

Any non-empty specification of the XMLinputFiles settings indicates to Banjo that during this execution, no search is to be performed, but instead the XML input files are to be combined into a single results set of n-best networks.

The final combined output based on the listed XML files will be written out to a text file (named based on the reportFile setting), with a brief header and the list of the combined n-best networks with their scores, as well as the name of the XML file that provided the network.

```
-----  
- Banjo                               Bayesian Network Inference with Java Objects -  
- Release 2.2.0                        15 Apr 2008 -  
- Licensed from Duke University        -  
- Copyright (c) 2005-08 by Alexander J. Hartemink -  
- All rights reserved                  -  
-----  
- Project:                             banjo static example  
- User:                                demo  
- Dataset:                             33-vars-320-observations  
- Notes:                               cluster results  
-----  
  
Combined results from the 8 supplied XML files:  
  
Original file: (path)\static.2008.06.06.12.59.41.xml  
Network #1 of 10  
Score: -8503.5789  
33  
 0 2 25 30  
 1 2 4 19  
 2 1 17  
 ...  
32 1 13  
  
Original file: (path)\static.2008.06.06.12.59.01.xml  
Network #2 of 10  
Score: -8504.7225  
33  
 0 2 5 25  
 1 1 17  
 2 1 17  
 ...  
32 1 13
```

Supported Data Formats

In Banjo 1.0.x we only supported a single format for the observation data: each row had to contain a single observation, listing the values of the variables delimited by spaces.

Banjo 2 also supports the use of observations that are column oriented, i.e., each observation is supplied as a single column, by specifying the optional *variablesAreInRows* setting as “yes”.

In addition, users can now also specify the names of the variables, using the *variableNames* Setting. By default Banjo expects a white-space separated list of strings of exactly as many items as the specified variable count. Alternatively, one can prefix the list by “commas:” and then provide a comma-separated list.

For observations with variables in columns (the original Banjo format) the variable names can also be entered on the very first data line (i.e., the first line in the settings file that is not blank, and doesn't start with a #-symbol) directly within the file, by setting *variableNames* to "inFile". Note: When specified in this way, the variable names need to be in the form of a white-space separated list.

The import of observation data in Banjo 2 has changed towards more flexibility, by using an implicit data mapping of integer data towards the 0..maxValueCount range of all supplied data values. This implies that as long as the observation data is in integer format, Banjo 2 always eliminates any gaps in the range of observations entries, as shown in the following example.

0	none	2.0	8.0	2	2	counts: {1.0=46, 4.0=6}
1	none	2.0	8.0	2	2	counts: {1.0=2, 4.0=50}
2	none	2.0	8.0	2	2	counts: {1.0=2, 4.0=50}

In this example the only supplied observation values for each of the 3 variables were 1 and 4. Without the need for the user to select any discretization, the data import code will internally map 1 to 0, and 4 to 1. Note that this mapping is desirable in view of computations for the used BDe metric, which at one point utilizes the number of assumed values for each of the variables. If we didn't specify any discretization, then Banjo 1.0.x would have assumed a range of values from 0 to 4 for each of the variables, and thus 5 would have been the "maximum value count" in the score computation. Not surprisingly this results in different scores compared to using the correct value count of 2 (because there are only 2 values, namely 1 and 4).

By providing the implicit data mapping we take some burden off the user when supplying the observation data – and eliminate a potential reason of getting unexpected results that can be quite difficult to reconcile with expectations.

Sample Output and Screenshots

In the examples below, the output was taken directly from the Banjo application and can be viewed as simulated screenshots obtained by running Banjo in a console or terminal window on a workstation.

Example: Searching for the "Best" Static Bayesian Network

The underlying problem for this example is a static Bayesian network with 33 variables and 320 observations. Say that we slightly modified the provided settings file to permit a longer run (of 1 hour) and made a few other changes, saving them as `static.settings.long.txt`. Then we execute Banjo by typing:

```
java -jar banjo.jar settingsFile=data/release2.0/static/static.settings.long.txt
```

The application will provide immediate feedback on the settings that were supplied. In Banjo 2 the feedback is organized into multiple sections, separated by dividing lines:

- The general header info for tracking the version of Banjo being used.
- The project information, which includes the four free-form settings 'project', 'user', 'data set', and 'notes'.
- The settings file from where Banjo loads its parameters.

- The values of the general parameters that set up a search, including the names and locations of input directory and observations file, and the optional discretization policy for the supplied data
- The optional structure files for specifying a initial structure, the must-be-present and the must-not-be-present parents, as well as the Markov lags, maximum parent count for each variable, and the equivalent sample size for the Dirichlet parameter prior.
- The core components that define the Search strategy, i.e., the searcher, proposer, evaluator, and decider objects.
- The main runtime performance settings, controlling the cache and the pre-computed array of log-Gamma values.
- The parameters that defines the specific searcher. In our case, Simulated Annealing is governed by the values for the initial temperature, the cooling factor, the reannealing temperature, the max. number of accepted networks before cooling, the max. number of proposed networks before cooling, and the min. number of accepted networks before reannealing.
- The output related information, which includes the output directory, the results file name, the number of high-scoring networks tracked, the stopping criteria (in terms of time, number of networks, or number of restarts), and the limit on iterations of the inner search loop (i.e., the number of search iterations to be executed without checking the stopping criteria). It also includes the intervals that feedback information id written to the screen (i.e., the console) and to the result file.
- The parameters for using the post-processing functions, which include the computation of influence scores, automatic dot output to text file as well as in various graphics formats, and the computation of a consensus graph.

As sample output is shown here:

```

-----
- Banjo                      Bayesian Network Inference with Java Objects -
- Release 2.0                  1 Apr 2007 -
- Licensed from Duke University -
- Copyright (c) 2005-2007 by Alexander J. Hartemink -
- All rights reserved         -
-----
- Project:                    banjo static example
- User:                       demo
- Dataset:                    33-vars-320-observations
- Notes:                      static bayesian network inference
-----
- Settings file:              data/static/static.settings.txt
-----
- Input directory:            data/static/input
- Observations file:          static.data.txt
- Number of observations:      320
- Number of variables:        33
- Discretization policy:      none
- Exceptions to the discretization policy: none
-----
- Initial structure file:
- 'Must be present' edges file: static.mandatory.str
- 'Must not be present' edges file:
- Min. Markov lag:            0
- Max. Markov lag:            0
- Max. parent count:          5
- Equivalent sample size for Dirichlet parameter prior: 1.0
-----
- Searcher:                   SimAnneal
- Proposer:                    ProposerRandomLocalMove
- Evaluator:                   defaulted to EvaluatorBDe
- Cycle checker:               CycleCheckerDFS
- Decider:                     defaulted to DeciderMetropolis
-----

```

```

- Pre-compute logGamma: yes
- Cache: fastLevel2
-----
- Initial temperature: 1000
- Cooling factor: 0.7
- Reannealing temperature: 800
- Max. accepted networks before cooling: 2500
- Max. proposed networks before cooling: 10000
- Min. accepted networks before reannealing: 500
-----
- Output directory: data/static/output
- Report file: static.report.txt
- Number of best networks tracked: 5
- Best networks are: nonidentical
- Max. time: 1.0 h
- Min. networks before checking: 1000
- Screen reporting interval: 3.0 m
- File reporting interval: 10.0 m
-----
- Compute influence scores: no
- Compute consensus graph: no
- Create consensus graph as HTML: no
- Create 'dot' output: no
- Location of 'dot': C:/Program Files/ATT/Graphviz/bin/dot.exe
-----

Memory info before starting the search: Banjo is using 11 mb of memory

```

Banjo can also display a “discretization report” for listing some of the core characteristics of the data supplied in the observations file. In our case, the data was already discrete, with values from 0 to 3, so no discretization was necessary. For this reason the “discretization policy” setting was set to “none”. The report simply describes the original data and what was done to it by the discretization policy, if anything.

- Pre-processing				Discretization report			
Variable	Discr.	Min. Val.	Max. Val.	Orig. points	Used points		
0	none	0.0	1.0	2	2		
1	none	0.0	3.0	4	4		
2	none	0.0	3.0	4	4		
3	none	0.0	3.0	4	4		
...							
29	none	0.0	3.0	4	4		
30	none	0.0	3.0	4	4		
31	none	0.0	3.0	4	4		
32	none	0.0	3.0	4	4		

In Banjo 2, the display of the discretization report can be controlled by the end user via the *createDiscretizationReport* setting, which can take the values “none”, “standard” (the report format displayed above), “withMappedValues” (which includes a summary of the mapping to the discretized values), and “withMappedandOriginalValues” (which contains a complete description of the mapping; note that for data sets with a large number of distinct values, this results in a lot of data being displayed).

Banjo then starts the execution of the search, and provides periodic feedback on its progress: in our case, the max. time allotted for the search was 60 minutes, and we had requested a progress reports every 3 minutes. In addition, we instructed Banjo that the intermediate results be saved to file every 10 minutes.

```
Starting search at 8/31/06 1:04:53 PM
Prep. time used: 953.0 ms
Beginning to search: expect a status report in 3.0 m

Status: Networks      25349900
       Time           3.0 m ( 5.0% of max.  1.0 h)
       Re-anneals     266
       Banjo is using 11 mb of memory

Status: Networks      51209600
       Time           6.0 m ( 10.0% of max. 1.0 h)
       Re-anneals     537
       Banjo is using 11 mb of memory

Status: Networks      77414900
       Time           9.0 m ( 15.0% of max. 1.0 h)
       Re-anneals     811
       Banjo is using 11 mb of memory

...
```

Finally, when the maximum allotted search time or number of search loops is reached, Banjo prints out the search results, which includes the n best networks – in our case the single best network – and a set of statistical information collected by the search components. Since each data set has its own unique characteristics, the statistics are helpful in tuning a search strategy.

```
Status: Networks      493480900
       Time           57.0 m ( 95.0% of max. 1.0 h)
       Re-anneals     5174
       Banjo is using 11 mb of memory

Status: Networks      518834300
       Time           1.0 h (100.0% of max. 1.0 h)
       Re-anneals     5440
       Banjo is using 11 mb of memory

-----
- Final report                                     Best network(s) overall
-----
These are the 5 top-scoring non-identical networks found during the search:

Network #1, score: -8451.65, first found at iteration 164065386
33
 0 2 5 25
 1 1 17
 2 1 17
 3 1 5
...
29 1 13
30 1 0
31 1 29
32 1 13

Network #2,...

-----
- Search Statistics
-----

Statistics collected in searcher 'SearcherSimAnneal':
Search completed at 8/31/06 2:04:53 PM
Number of networks examined: 518834300
Total time used: 1.0 h
High score: -8451.65, first found at iteration 164065386
```

```

Number of re-anneals: 5440

Statistics collected in proposer 'ProposerRandomLocalMove':
  Additions -- proposed: 173576103
  Deletions -- proposed: 172645027
  Reversals -- proposed: 172613169

Statistics collected in cycle checker 'CycleCheckerDFS':
  Additions -- considered: 173576103, acyclic: 157439233
  Deletions -- considered: 172645027, acyclic: 172645027
  Reversals -- considered: 172613169, acyclic: 169869811

Statistics collected in evaluator 'EvaluatorBDe':
  Scores computed: 15424311
  Scores (cache) placed fetched
    with 0 parents: 33 285857828
    with 1 parents: 1056 154165202
    with 2 parents: 16368 209307755
    with 3 parents: 14092605 4972684
    with 4 parents: 1198133 107711
    with 5 parents: 116116 5881

Statistics collected in decider 'DeciderMetropolis':
  Additions -- considered: 157439233, better score: 22710155, other accepted:
41811555
  Deletions -- considered: 172645027, better score: 41864242, other accepted:
22657459
  Reversals -- considered: 169869811, better score: 48425220, other accepted:
21930375
  Average permissivity: 0.223

No Statistics collected in equivalence checker 'EquivalenceCheckerSkip'.

Memory info after completing the search: Banjo is using 11 mb of memory

```

Explanation of the Results

Banjo supplies the obtained high-scoring Bayesian network in the following form (the data for nodes id = 4 to id = 28 is omitted):

```

Network #1, score: -8451.65, first found at iteration 164065386
33
0 2 5 25
1 1 17
2 1 17
3 1 5
...
29 1 13
30 1 0
31 1 29
32 1 13

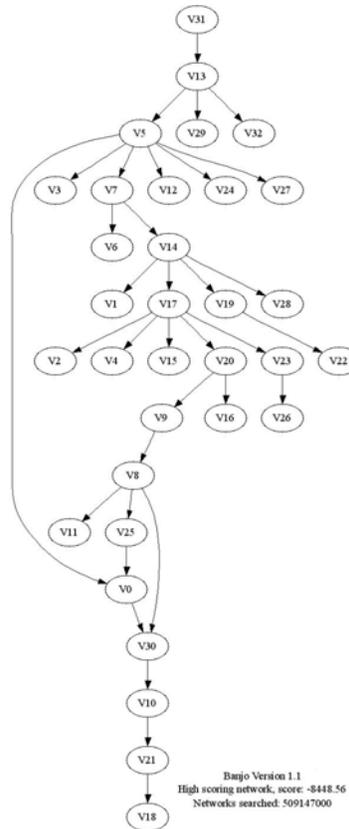
```

The first line indicates the score (-8451.65) and when it was first encountered (iteration 164065386).

Line 2 indicates that the number of variables in the network is 33.

Lines 3 to 35 (one for each of the 33 variables) first list the id of a variable, then the number of parents, and then a listing of the parents. E.g., “0 2 5 25” means that variable id = 0 has 2 parents, namely id = 5 and id = 25.

The graphical representation of the obtained network, generated using the Banjo dot format output looks like this. Note that we used the variableNames setting to display the name of each variable (we assigned variable “i” the name “Vi”, for i from 0 to 32)



Example: Searching for the “Best” Dynamic Bayesian Network

The second example is a search for a dynamic Bayesian network (DBN), described as a problem with 20 variables and 2000 observations. The minimum and maximum Markov lag in this example are both equal to 1, which means that no links between nodes of Markov lag 0 are permitted. You may notice in the resulting statistics that no reversals were considered as possible changes to the Bayesian network. In addition, there was no need for the search algorithm to perform any cycle checking when proposing a change. We run the search with

```
java -jar banjo.jar settingsFile=data/release2.0/dynamic/dynamic.settings.txt
```

The application will provide immediate feedback on the settings that were supplied:

```
-----
- Banjo                               Bayesian Network Inference with Java Objects -
- Release 2.0                          1 Apr 2007 -
- Licensed from Duke University        -
- Copyright (c) 2005-2007 by Alexander J. Hartemink -
- All rights reserved                  -
-----
- Project:                             banjo dynamic example
- User:                                 demo
- Data set:                             20-vars-2000-temporal-observations
- Notes:                                dynamic bayesian network inference
-----
- Settings file:                        data/dynamic/dynamic.settings.txt
-----
```

```

- Input directory: data/dynamic/input
- Observations file: dynamic.data.txt
- Number of observations (in file): 2000
- Number of observations used for learning DBN: 1999
- Number of variables: 20
- Discretization policy: none
- Exceptions to the discretization policy: none
-----
- Initial structure file:
- 'Must be present' edges file:
- 'Must not be present' edges file:
- Min. Markov lag: 1
- Max. Markov lag: 1
- DBN mandatory identity lag(s): 1
- Max. parent count: 5
- Equivalent sample size for Dirichlet parameter prior: 1.0
-----
- Searcher: SearcherGreedy
- Proposer: ProposerAllLocalMoves
- Evaluator: defaulted to EvaluatorBDe
- Cycle checker: CycleCheckerDFS
- Decider: defaulted to DeciderGreedy
-----
- Pre-compute logGamma: yes
- Cache: fastLevel2
-----
- Min. proposed networks after high score: 1000
- Min. proposed networks before restart: 3000
- Max. proposed networks before restart: 5000
- Restart method: use random network
- with max. parent count: 3
-----
- Output directory: data/dynamic/output
- Report file: dynamic.report.txt
- Number of best networks tracked: 5
- Best networks are: nonidentical
- Max. time: 3.0 m
- Min. networks before checking: 1000
- Screen reporting interval: 20.0 s
- File reporting interval: 10.0 m
-----
- Compute influence scores: no
- Compute consensus graph: no
- Create consensus graph as HTML: no
- Create 'dot' output: no
- Location of 'dot': not supplied
-----
- Pre-processing Discretization report
-----

```

Variable	Discr.	Min. Val.	Max. Val.	Orig. points	Used points
0	none	0.0	2.0	3	3
1	none	0.0	2.0	3	3
2	none	0.0	2.0	3	3
3	none	0.0	2.0	3	3
4	none	0.0	2.0	3	3
5	none	0.0	2.0	3	3
6	none	0.0	2.0	3	3
7	none	0.0	2.0	3	3
8	none	0.0	2.0	3	3
9	none	0.0	2.0	3	3
10	none	0.0	2.0	3	3
11	none	0.0	2.0	3	3
12	none	0.0	2.0	3	3
13	none	0.0	2.0	3	3
14	none	0.0	2.0	3	3
15	none	0.0	2.0	3	3
16	none	0.0	2.0	3	3
17	none	0.0	2.0	3	3
18	none	0.0	2.0	3	3
19	none	0.0	2.0	3	3

Banjo then provides periodic feedback on its progress, and, when the search is completed, it supplies the final results. In our case this includes the 5 highest scoring networks, the statistical information about the search, a basic output of the best network for generating a graph in dot, and the list of influence scores.

```

Memory info before starting the search: Banjo is using 12 mb of memory

Starting search at 8/31/06 10:46:08 AM
Prep. time used: 1.3 s
Beginning to search: expect a status report in 20 s

Status: Networks 376201
       Time      20.05 s ( 11.1% of max.  3.0 m)
       Restarts  65
       Banjo is using 13 mb of memory

Status: Networks 758101
       Time      40.06 s ( 22.2% of max.  3.0 m)
       Restarts 132
       Banjo is using 12 mb of memory

...

Status: Networks 2991361
       Time      2.67 m ( 88.9% of max.  3.0 m)
       Restarts 524
       Banjo is using 12 mb of memory

Status: Networks 3373261
       Time      3.0 m (100.0% of max.  3.0 m)
       Restarts 591
       Banjo is using 12 mb of memory

-----
- Final report                                     Best network(s) overall
-----
These are the 5 top-scoring non-identical networks found during the search:

Network #1, score: -15935.29, first found at iteration 4941
20
0  1:  2 0 7
1  1:  1 1
2  1:  3 0 1 2
3  1:  2 2 3
4  1:  2 1 4
5  1:  2 4 5
6  1:  1 6
7  1:  2 3 7
8  1:  2 3 8
9  1:  3 5 6 9
10 1:  3 8 9 10
11 1:  2 10 11
12 1:  1 12
13 1:  1 13
14 1:  1 14
15 1:  1 15
16 1:  1 16
17 1:  1 17
18 1:  1 18
19 1:  1 19

Network #2, score: -15939.38, first found at iteration 4561
20
0  1:  2 0 7
1  1:  1 1
2  1:  3 0 1 2
3  1:  2 2 3
4  1:  2 1 4
5  1:  2 4 5
6  1:  1 6

```

```
7 1: 2 3 7
8 1: 2 3 8
9 1: 2 5 9
10 1: 3 8 9 10
11 1: 2 10 11
12 1: 1 12
13 1: 1 13
14 1: 1 14
15 1: 1 15
16 1: 1 16
17 1: 1 17
18 1: 1 18
19 1: 1 19
```

Network #3, score: -15986.77, first found at iteration 4181

```
20
0 1: 2 0 7
1 1: 1 1
2 1: 3 0 1 2
3 1: 2 2 3
4 1: 2 1 4
5 1: 2 4 5
6 1: 1 6
7 1: 2 3 7
8 1: 2 3 8
9 1: 2 5 9
10 1: 2 9 10
11 1: 2 10 11
12 1: 1 12
13 1: 1 13
14 1: 1 14
15 1: 1 15
16 1: 1 16
17 1: 1 17
18 1: 1 18
19 1: 1 19
```

Network #4, score: -15996.63, first found at iteration 3801

```
20
0 1: 2 0 7
1 1: 1 1
2 1: 3 0 1 2
3 1: 2 2 3
4 1: 2 1 4
5 1: 2 4 5
6 1: 1 6
7 1: 2 3 7
8 1: 2 3 8
9 1: 2 5 9
10 1: 1 10
11 1: 2 10 11
12 1: 1 12
13 1: 1 13
14 1: 1 14
15 1: 1 15
16 1: 1 16
17 1: 1 17
18 1: 1 18
19 1: 1 19
```

Network #5, score: -16061.54, first found at iteration 3421

```
20
0 1: 2 0 7
1 1: 1 1
2 1: 3 0 1 2
3 1: 2 2 3
4 1: 2 1 4
5 1: 2 4 5
6 1: 1 6
7 1: 2 3 7
8 1: 2 3 8
9 1: 1 9
10 1: 1 10
11 1: 2 10 11
12 1: 1 12
13 1: 1 13
```

```
14 1: 1 14
15 1: 1 15
16 1: 1 16
17 1: 1 17
18 1: 1 18
19 1: 1 19
```

- Search Statistics

Statistics collected in searcher 'SearcherGreedy':

```
Search completed at 8/31/06 10:49:08 AM
Number of networks examined: 3075721
Total time used: 3.0 m
High score: -15935.29, first found at iteration 4941
Number of restarts: 539
```

Statistics collected in proposer 'ProposerAllLocalMoves':

```
Additions -- proposed: 3019628
Deletions -- proposed: 56092
Reversals -- proposed: 0 (min. Markov lag = 1)
```

Statistics collected in cycle checker 'CycleCheckerDFS':

```
Additions -- no cyclicity test necessary
Deletions -- no cyclicity test necessary
Reversals -- none proposed
```

Statistics collected in evaluator 'EvaluatorBDeOriginal':

```
Scores computed: 820787
Scores (cache) placed fetched
with 0 parents: 0 0
with 1 parents: 20 55034
with 2 parents: 380 2140957
with 3 parents: 727674 70156
with 4 parents: 92713 8080
with 5 parents: 0 0
```

Statistics collected in decider 'DeciderGreedy':

```
Additions -- considered: 7016, better score: 7016
Deletions -- considered: 1078, better score: 0
Reversals -- considered: 0 (min. Markov lag = 1)
```

No Statistics collected in equivalence checker 'EquivalenceCheckerSkip'.

Memory info after completing the search: Banjo is using 12 mb of memory

- Post-processing DOT graphics format output

digraph abstract {

```
label = "Banjo Version 1.0.0\nHigh scoring network, score: -15935.29\nProject:
banjo dynamic example\nUser: demo\nData set: 20-vars-2000-temporal-
observations\nNetworks searched: 3075721";
labeljust="l";
```

```
7->0;
0->2;
1->2;
2->3;
1->4;
4->5;
3->7;
3->8;
5->9;
6->9;
8->10;
9->10;
10->11;
}
```

- Post-processing Influence scores

```

-----
Influence score for (7,1) -> (0,0) -0.4377
Influence score for (0,1) -> (0,0) 0.7398
Influence score for (1,1) -> (1,0) 0.8321
Influence score for (2,1) -> (2,0) 0.7182
Influence score for (1,1) -> (2,0) -0.2764
Influence score for (0,1) -> (2,0) 0.1788
Influence score for (3,1) -> (3,0) 0.7487
Influence score for (2,1) -> (3,0) 0.4088
Influence score for (4,1) -> (4,0) 0.831
Influence score for (1,1) -> (4,0) 0.2829
Influence score for (5,1) -> (5,0) 0.7699
Influence score for (4,1) -> (5,0) -0.3771
Influence score for (6,1) -> (6,0) 0.8502
Influence score for (7,1) -> (7,0) 0.7327
Influence score for (3,1) -> (7,0) 0.4538
Influence score for (8,1) -> (8,0) 0.759
Influence score for (3,1) -> (8,0) -0.4027
Influence score for (9,1) -> (9,0) 0.693
Influence score for (6,1) -> (9,0) -0.1581
Influence score for (5,1) -> (9,0) 0.3163
Influence score for (10,1) -> (10,0) 0.6724
Influence score for (9,1) -> (10,0) 0.1913
Influence score for (8,1) -> (10,0) 0.3029
Influence score for (11,1) -> (11,0) 0.7665
Influence score for (10,1) -> (11,0) -0.4201
Influence score for (12,1) -> (12,0) 0.8469
Influence score for (13,1) -> (13,0) 0.8396
Influence score for (14,1) -> (14,0) 0.8457
Influence score for (15,1) -> (15,0) 0.8427
Influence score for (16,1) -> (16,0) 0.8625
Influence score for (17,1) -> (17,0) 0.8746
Influence score for (18,1) -> (18,0) 0.8476
Influence score for (19,1) -> (19,0) 0.8608

```

Explanation of the Results

The obtained high-scoring Bayesian network is supplied in the following form (the data for nodes id = 4 to id = 17 is omitted):

```

Network #1, score: -15935.29, first found at iteration 4941
20
0 1: 2 0 7
1 1: 1 1
2 1: 3 0 1 2
3 1: 2 2 3
...
18 1: 1 18
19 1: 1 19

```

In Banjo 2, we omit the (redundant) data for lags that are less than the specified minimum Markov lag. For illustration, we show what the output would have been in Banjo 1.0:

```

[Deprecated Banjo 1.0 format]
Network #1, score: -15935.2860609, first found at iteration 4941
20
0 0: 0 1: 2 0 7
1 0: 0 1: 1 1
2 0: 0 1: 3 0 1 2
3 0: 0 1: 2 2 3
...
18 0: 0 1: 1 18
19 0: 0 1: 1 19

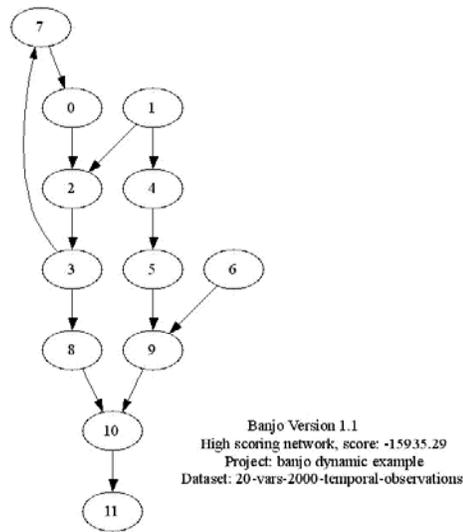
```

The first line indicates the ranking of the network (here: #1), and its associated score (here: -15935.29), first encountered at iteration 4941.

The second line indicates that the number of variables is 20.

Since we have a dynamic network with max. Markov lag 1, we list the parents for each node and for each Markov lag in a separate “block”, starting with the respective Markov lag and a colon (“:”). I.e., lines 3 to 22 (one for each of the 20 variables) first list the id of a variable, and then a block for each Markov lag starting at the minimum Markov lag (here, 1), and up to the maximum Markov lag (here, also 1). As an example, for node id = 0, “1: 2 0 7” indicates that variable id = 0 has 2 parents of Markov lag 1, namely variable id = 0 and variable id = 7). In Banjo 2, the block of data for lag 0 (which is excluded from contributing parents by the specified minimum Markov lag) is suppressed from the output.

The graphical representation of the network, obtained using dot, is this:



Note that since the min. and max. Markov lags are the same, all displayed nodes (variables) are displayed without reference to their corresponding lag.

Using Banjo

To make the most of Banjo, it is useful to take a look at what approach Banjo takes to solving the network inference problem.

The Banjo Application

The Banjo application is built around the `Searcher` class. A `Searcher` examines the space of possible solutions using a suitable search strategy. Banjo currently implements greedy and simulated annealing searches.

The focus of the remainder of this section will be the internal structure of the `Searcher` class because the structure of the internal components of `Searchers` lets us fine-tune the search strategy that we choose.

From a high-level point of view, each `Searcher` can be decomposed into the following tasks:

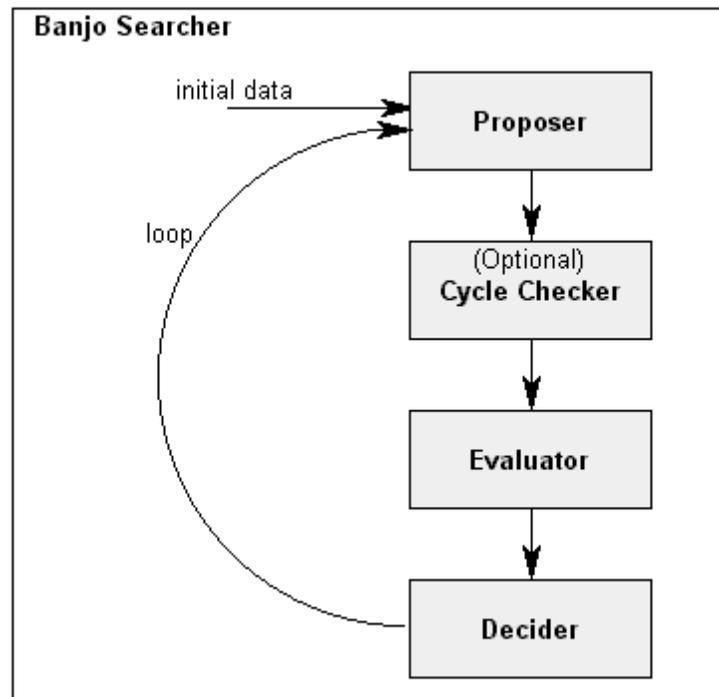
- Select an initial “current” network (can be the empty network, or some other pre-selected network). Then iterate through the following set of steps:
 1. Propose a new network that is to be considered. Often, the proposed network is dependent on the current network (it represents a local change to the current network).
 2. Check the proposed network for cycles (but only if they are even possible).
 3. Compute the score of the proposed network using a predefined metric.
 4. Decide, possibly stochastically, whether to accept the proposed network (as the new current network).

Banjo is also able to propose, check for cycles, and evaluate a *set* of local changes to the current network (for example, an exhaustive list of all local changes). In this case, the decider then decides whether to accept the best change in the set.

Beneath this conceptual component layer is a secondary data structure layer that is crucial for Banjo to achieve high performance. These data structures include the actual representation of a Bayesian network, a “change to a Bayesian network”, a “high-scoring network structure”, etc.

A note to developers: From a development point of view, the component layer and the data structure layer are almost completely separated, so it would be fairly straightforward to experiment with a different set of data structures. There are only a few (2 or 3) well-documented code locations where we chose to sacrifice the separation between the layers to achieve better performance.

The Banjo Components



(The Core Banjo Objects)

The Searchers

As the top-level component of the Banjo architecture for implementing a structure learning strategy, a Searcher's main task is to manage all aspects of searching in a space of possible networks for the "best possible" (i.e., highest scoring based on a scoring metric) network. In general, a search is built around a search loop that executes for an allotted amount of time or until a specified number of networks have been proposed and considered. At the end of the search execution, Banjo reports the network(s) with the best score(s) found.

Within the search loop, Banjo allows various combinations of Proposer, CycleChecker, Evaluator, and Decider components to handle the internal aspects of each iteration step. For example, a greedy search may examine a single, randomly selected change to the network and keep the new network if it scores better than the current network, or discard the change if it scores lower. Note that for efficiency reasons, a change to the network (for the purpose of our application) is typically defined as a local change: the addition, deletion, or reversal of a single edge in the current network.

Let's take a look at Banjo's design and break down the Searcher into some well-defined internal components. Instead of examining the single, randomly selected change to the network, a greedy search could also be implemented by examining all possible moves that are available from the current network in a single step (i.e., by examining all possible additions, deletions, and reversals of any single edge in the current network). These two variations of greedy search share most of their logic except for the "proposing" of the change(s) to the current network! By having a clearly defined "Proposer" subtask, we only have to implement a new Proposer, combine it with the other existing component tasks, and we conveniently end up with a completely new search strategy. To produce a new search strategy, we just have to

specify what components we want the search to use, mixing and matching with what is already available or possibly implementing a new search component.

The Proposers

A Proposer implements the part of the search algorithm that specifies what possible change or changes are to be considered at a single search iteration step. Choices for available Proposers depend on the selected search algorithm. If incompatible choices are selected for the Proposer and Searcher components, Banjo will notify the user and stop execution.

Currently, Banjo implements 2 types of Proposers, namely ProposerRandomLocalMove, and “ProposerAllLocalMoves”, which both turn out to be compatible with either currently available search strategy, *greedy* and *simulated annealing*. The designated default proposer is ProposerRandomLocalMove, which simply selects a move at random from all available local moves. By a move we mean the adding of subtracting of a parent, or the reversing of a parent relationship (i.e., reversing of an edge in the corresponding graph representation of the network). In contrast, ProposerAllLocalMoves composes a list of all available local moves, given the current state of the network, and then selects the move that yields the highest scoring network. Note that for greedy searches, this tends to find local maxima quickly, and, by its nature, only makes sense to be used with restarts based on randomly configured networks.

The Cycle Checkers

The task of the cycle checker is to examine whether each proposed network contains a cycle. Trivially, if it does, then the proposed change is discarded, and the search goes back to the Proposer to request another possible network change.

If the proposed network does not contain a cycle, then the next step in the search is the score computation performed by an Evaluator.

In version 2 we have replaced the depth first search (DFS) strategy with an improved version, and have added a modified DFS strategy based on the paper by O. Shmueli. Both strategies provide significant improvements in performance, so we discontinued the use of the breadth first search strategy altogether.

The structure of the underlying problem will affect the performance of the algorithms. We added user access to the selection of the search strategies, because each has its unique strength. Some experimentation may be necessary to find the best method for a particular problem.

From a functional (code) point of view, the cycle checker choice is independent of the choices of the other core objects. However, the choice of searcher and proposer does significantly influence the effectiveness of the Shmueli optimization: there is a “penalty” for reversals of edges, which is especially pronounced for a Greedy search with AllRandomMoves.

The Evaluators

An Evaluator in Banjo computes the score of a network, based on some scoring metric. There is currently only one Evaluator available in Banjo, which uses the BDe metric to compute a network’s score, as described first by Cooper and Herskovits and later by Heckerman.

You can specify an Evaluator via the value of `evaluatorChoice` in the settings file. The valid choices are “default” and “EvaluatorBDe”, with identical effect: both will cause Banjo to select the BDe metric in the score computation.

The evaluator choice is also independent of the choices of the other core objects.

The Deciders

A Decider in Banjo determines whether the proposed network in the current search iteration will be accepted as the new current network for the next iteration, or if it will be rejected, in which case the search proceeds from the current network.

The choice of the decider is tightly connected to the selected search strategy, which is expressed in fact via the naming of our deciders, `DeciderMetropolis` and `DeciderGreedy`.

The Equivalence Checkers

The `EquivalenceChecker` has been added in Banjo 2 for comparing networks when tracking a set of high-scoring networks during a search. In Banjo 1.0, the obtained high-scoring set was composed of N networks that were only compared to each other with respect to identity, thus allowing multiple equivalent networks to be part of the final result.

Note that the choice of equivalence checker is determined by the *bestNetworksAre* setting.

The equivalence checker choice is also independent of the choices of the other core objects.

Summary of Component Options

When you decide you want to implement a new search strategy, you may want to look at the existing search algorithms in Banjo. Both Greedy and Simulated Annealing search strategies can be applied via the provided Searchers. Each of them can be used with a random local move or all local moves approach by specifying the appropriate Proposer. The greedy approach uses a greedy Decider, which only accepts networks with better scores, whereas the simulated annealing approach accepts networks based on a stochastic Decider (implementing Metropolis-Hastings). The table below shows how one can select different components in the existing search implementations:

Searcher Options	Dependent core objects	Choices for the dependent core objects	Explanation
SimAnneal (for simulated annealing search)	Proposer	RandomLocalMove	Addition, deletion, or reversal of an edge in the current network, selected at random.
		AllLocalMoves	All changes arising from a single addition, deletion, or reversal of an edge in the current network.
	Decider	Metropolis	A Metropolis-Hastings stochastic decision mechanism, where any network with a higher score is accepted, and any with a lower score is accepted with a probability based on a system parameter known as the “temperature”.
Greedy (for “greedy” search)	Proposer	RandomLocalMove	Addition, deletion, or reversal of an edge in the current network, selected at random.
		AllLocalMoves	All changes arising from a single addition, deletion, or reversal of an edge in the current network.
	Decider	Greedy	A network is accepted if and only if its score is better than or equal to that of the current network. In the case of AllLocalMoves, the best local move is considered.
Skip	N/A	N/A	The search is skipped entirely, to immediately apply the post-processing options. (Introduced in Banjo 2)

A new searcher in Banjo version 2 is indicated by the searcher option named “**Skip**”, which simply instructs Banjo to only setup the underlying network structures, but skip the actual search execution, and go straight to the post processing code.

In addition, we want to add that the *AllLocalMoves* proposer option should be used cautiously, so that the search does not visit the same networks over and over again. In particular, the *restartWithRandomNetwork* setting should always be set to “yes” when using *AllLocalMoves*.

Setting up a Banjo Search

One of the goals of the Banjo distribution was to provide all the pieces for a researcher to quickly get the program up and running, by including sample settings files and detailed instructions in both this user guide and the developer guide. However, there are some intrinsic issues with using software that is intended for the serious researcher, the most important being the fact that the user needs to have a thorough understanding of the methodology that he/she is going to use, and that software like Banjo cannot be expected to be successfully applied to a research problem without some further investigating, and in many cases, careful and extensive experimenting with a number of parameters that govern the program execution.

The sections on tuning the memory use, tuning the search performance, and selecting the input discretization describe in detail the tuning parameters and their settings. In the section on experimenting with Banjo we use some basic examples to try to illustrate some effects that may surprise a first time user.

Tuning the Memory Use

The previous section describes the methods that are available for a search. Obviously, there are only a small number of ways to combine the core options for a search. However, to run each search effectively it is quite important to tune a number of parameters that determine the run time performance of the program. This section is intended to assist the user in selecting the appropriate choices for their problem at hand.

The main constraint on Banjo as it is executed on a computer is the memory available to its code. Depending on the size of the underlying problem, i.e., the number of variables and the number of observations, Banjo may not be able to run at all unless we modify some or all of the settings that control the amount of memory that Banjo will request from the system.

The settings that may need to be adjusted are:

1. The *useCache* setting, which adjusts the amount of information that Banjo caches to avoid re-computing already obtained scores. Note that internally there are 2 different storage containers (“caches”), namely the so-called “fastCache”, and a “general cache”. The fastCache stores all scores for variables with up to a fixed number of parents, up to a maximum level of 2 (parents). E.g., when fastCache is set to level 1, then the fast cache will request room for storing scores for all variables, and their parent configurations with 0 and 1 parent. Note that the fastCache is filled on an as-needed basis, but no scores are ever deleted from the fastCache. The heuristic justification for the fastCache use lies in the observation that the search algorithms visit these network configurations over and over, so we don’t want them to be subjected to the same mechanism as the regular cache. All other scores are stored in a general cache that uses whatever memory is “left”.
Note that for performance reasons, a higher fastCache level is always better.
2. The *precomputeLogGamma* setting, which – when set to “yes” – causes Banjo to pre-compute a table for looking up the potential log gamma value for the score computation based on the BDe metric.
3. On a secondary level, Banjo’s memory use gets affected (for large problems) by the *nBestNetworks* setting. We need to keep in mind that storing potentially thousands of networks, each with thousands of variables, will substantially increase the memory requirements.

The default settings for Banjo 2 are tuned for small problems, which we define as up to 100 variables, with up to several thousand observations. There is nothing to adjust in this case, since the required memory is well within the bounds of the standard allocation by the Java virtual machines. The cache use is set at `fastLevel2`, and the `precomputeLogGamma` is set to “on”. We can expect the total memory use by Banjo to be far less than the 64 mb, the standard memory allocation for the SUN JVM.

For a medium size problem, which we define as up to 2000-3000 variables, and 2000-3000 observations, we need to restrict the `fastCache` to Level 1 (raising the `fastCache` to level 2 requires an additional storage of up to several GB of memory). At this point we also want to turn off the `precomputeLogGamma` setting. With these modified settings we can expect the memory use to be within about 300-500 MB, easily feasible for a suitably equipped desktop computer.

For all problems that are larger than mentioned above, it will be important to know how much memory can be made available to the JVM, and a little trial and error may be necessary to find the optimal balance between performance (i.e., the number of networks we can visit in a given amount of time), and having sufficient memory available simply to be able to run a search at all. As long as the `displayMemoryInfo` setting is set to “yes”, the total amount of memory used by Banjo will be displayed. To gain even more insight where the memory is being allocated, a brief excursion into the Banjo code will be necessary: at the developer level one can set the `TRACE_MEMORYUSE` constant to true – the provided info is the most detailed for a SimAneal search and a `localRandomMove` proposer.

Performance Tuning

The second major consideration for setting up Banjo is the overall performance (speed) of the search. For obvious reasons, the more variables and/or observations have to be factored into the computations, the more time-consuming it becomes for the search to move from one network to the next. Thus one should closely examine a problem theoretically before running a search, and eliminate all redundant or unnecessary information, whenever possible: for many large data sets this could mean anything from removing variables to limiting the maximum parent count that each node may have.

Another major influence on performance is the choice for the `bestNetworksAre` setting. This is a Banjo 2 feature that allows a search to be restricted to finding non-equivalent networks only. There are three valid choices for `bestNetworksAre`, namely `nonidenticalThenPruned`, `nonidentical`, and `nonequivalent`. “Nonidentical” corresponds to the way that Banjo 1.0.x used to search, i.e., when `nBestNetworks=N`, then simply find the N highest scoring, different networks. The drawback of this approach is that the result set can consist of 1 or more subsets of networks that are “equivalent”.

It turns out that when comparing networks for equivalence each time we encounter a new network comes at a very high performance cost (as documented in the literature). We can get partially around this performance penalty by letting the search collect the best n non-identical networks, and pruning this result set at the very end. Of course, in general we will likely end up with fewer non-equivalent networks than the found non-identical networks, so we have to compensate by tracking a larger set of non-identical networks from the start. Still, the mentioned problem remains – there is no exact recipe for working around this issue.

Finally, there are the choices for intermediate feedback to the console (via the `screenReportingInterval` setting), and the intermediate results output to file (via the `fileReportingInterval`). Each of these operations will slow down the actual search, so we want to choose reasonable intervals. On the other hand, should a Banjo search be terminated

externally, then the last intermediate report that was written to file would be the final recorded search result.

Input Discretization Options

This version of Banjo requires networks to be over discrete variables, since it only implements the BDe scoring metric. It is often best to transform your data to discrete values using some intelligent discretization strategy, but if you simply want to perform either quantile or interval discretization, Banjo can do this for you.

Discretization is controlled by two settings. The first, called `discretizationPolicy`, specifies a default policy for all variables; the second, called `discretizationExceptions`, specifies a list of potential exceptions to the default policy. The default policy can be either the token “none” or a token like “q2” or “i4”. The latter types of tokens begin with either a “q” for quantile discretization or an “i” for interval discretization and are then followed by an integer specifying the desired number of states. This integer should be at least 1, and no larger than the maximum number of states permitted for any discrete variable in Banjo (which is 5 by default, but can be adjusted to a larger value in `BANJO.java` (beware that this can consume a lot of memory)). It should be noted that if you specify quantile discretization and a number of states that is greater than the number of values in your data for one of your variables, then that variable’s states will not be altered.

If there are any exceptions to the default policy, then you specify a comma-separated list of tokens which have the form: `variable index (starting at 0), colon, and discretization policy` for that variable, e.g. “0:q4,5:none,7:i2”.

To monitor the data mapping done by Banjo, version 2 lets you specify the new setting `createDiscretizationReport`, using the values `no`, `standard`, `withMappedValues` or `withMappedAndOriginalValues`, shown below with a sample output:

- “standard”,

Variable	Discr.	Min. Val.	Max. Val.	Orig. points	Used points
0	q5	0.0	1.0	2	2
1	q5	0.0	3.0	4	4
2	q5	0.0	3.0	4	4
3	q5	0.0	3.0	4	4
4	q5	0.0	3.0	4	4

This is the most compact output, showing only the minimum and maximum values, as well as the number of different original and used values, for each variable.

- “withMappedValues”

Variable	Discr.	Min. Val.	Max. Val.	Orig. points	Used points	
0	q5	0.0	1.0	2	2	map: {0.0=0, 1.0=1}
1	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}
2	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}
3	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}
4	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}

In our example, the mapping consists of a trivial one, since our original data had the integer values 0 to 3. So the mapping has 0 (interpreted as the real valued 0.0) going to 0, 1 (again, interpreted as a real value, 1.0) going to 1, etc.

- “withMappedAndOriginalValues”.

Variable	Discr.	Min. Val.	Max. Val.	Orig. points	Used points		
0	q5	0.0	1.0	2	2	map: {0.0=0, 1.0=1}	orig. values with counts: {0.0=250, 1.0=70}
1	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}	orig. values with counts: {0.0=4, 1.0=2, 2.0=60, 3.0=254}
2	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}	orig. values with counts: {0.0=12, 1.0=54, 2.0=116, 3.0=138}
3	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}	orig. values with counts: {0.0=8, 1.0=36, 2.0=234, 3.0=42}
4	q5	0.0	3.0	4	4	map: {0.0=0, 1.0=1, 2.0=2, 3.0=3}	orig. values with counts: {0.0=4, 1.0=12, 2.0=108, 3.0=196}

The additional information provided tells us what values (for each variable) occurred how many times in the original data set. E.g., variable id=2 had 12 occurrences of 0.0, 54 occurrences of 1.0, etc.

Experimenting with Banjo

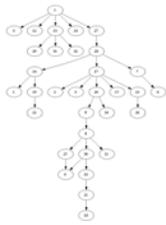
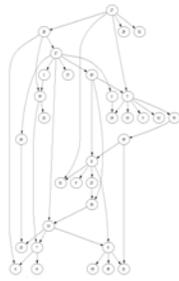
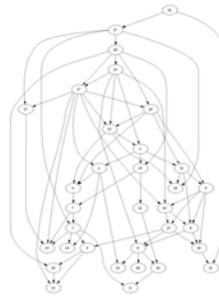
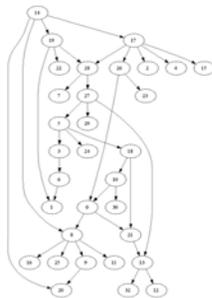
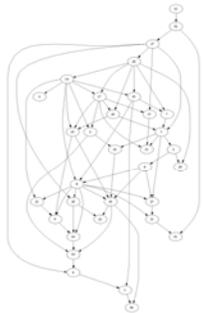
In the sections above we have described in detail the most important tuning parameters. The question of course remains how to actually use them in a “real” problem. Unless otherwise indicated, we use our supplied example problem for the static bayesnet case, with 33 variables and 320 observations, to show how various setting values affect the outcome of a search. Ultimately, our goal is to alert Banjo users that the program is not a magic black box, but a tool that requires careful planning, experimenting, and execution on part of the user.

The results below show that Banjo 2 allows the user to tackle problems with a large number of variables of, say, in the thousands. However, we admit frankly that the Banjo code is not fully optimized for such large problems. While the current code base has eliminated a number of inefficiencies stemming from Java’s implementation of multidimensional arrays, one can expect a sizable performance boost from rewriting the underlying parent set implementation using any of a number of well-known techniques for implementing sparse matrices. Tackling these large problems is not within the scope and immediate interest of our research group, and, with finite resources at our disposal, we leave such changes to the motivated user/developer/student. The required changes are easily identifiable within the Edges subhierarchy of the Banjo code, and the few locations where we have to break the encapsulation of our classes for performance reasons are well documented in the code.

Note: All searches for a given example have been executed on the same machine, running as the only computing-intensive process running at the time. Searches for different examples may have been executed on different machines, so absolute comparisons across examples may not be meaningful.

Example: Choice of Discretization

The data for our static example has 2 values for variable 0, 3 values for variable 1, and 4 values for all the rest of the variables. To illustrate the effect of something as simple as the choice input discretization, we will compare the results from a batch of short searches for 1 minute each, using internal and quantile discretization of 2 and 3 values for each variable.

Discretization	None	Q3	Q2	I3	I2
Networks visited (1 min. test run)	9 million	6.5 million	4.5 million	7.2 million	4.6 million
Graph					
Networks visited that have 4 or 5 parents	23,000	62,000	248,000	51,000	235,000
Highest parent count for a single variable in top network	2 (for 1 var.)	2 (for 12 var.)	4	2 (for 7 var.)	4

We observe a much higher complexity of the resulting networks for the i2 and q2 discretizations when compared to the i3 and q3 results, and for all of them compared to the reference result based on the “known” data. In addition, there is a significant performance impact from visiting networks with higher parent counts for the individual variables.

Example: Combinations of useCache, MaxParentCount, and precomputeLogGamma

In this case we execute short (1 minute) searches to examine the effect of the various cache settings when combined with different maxParentCount and precomputeLogGamma values, on the search performance, in our 33 variable sample data. The search used a Simulated Annealer searcher with a RandomLocalMove proposer.

Test 1: maxParentCount=5, precomputeLogGamma=no

Cache level	None	Basic	Fastlevel0	Fastlevel1	Fastlevel2
Number of networks examined	1,404,000	1,867,000	1,927,000	2,510,000	6,778,000
Number of Re-anneals	12	17	17	23	63
Memory used	1 mb	1 mb	1 mb	1 mb	1 mb
High score	-8,452.44	-8,467.48	-8,477.89	-8,475.26	-8,471.7
(Node) Scores computed	1,812,226	1,092,991	1,127,346	807,418	217,877
Networks visited improvement	Baseline	1.33	1.37	1.79	4.83

We observe that the performance in terms of searched networks increases substantially the higher we set our cache. Since our problem only has a small number of variables we are not impacted by an increase in memory requirements, so it is safe to turn the cache to its maximum level.

Test 2: maxParentCount=5, precomputeLogGamma=yes

Cache level	None	Basic	Fastlevel0	Fastlevel1	Fastlevel2
Number of networks examined	2,449,000	3,237,000	3,181,000	4,128,000	8,419,000
Number of Re-anneals	22	30	29	38	78
Memory used	11 mb	11 mb	11 mb	11 mb	11 mb
High score	-8,472.79	-8,459.66	-8,466.67	-8,465.07	-8,464.89
(Node) Scores computed	3,163,662	1,891,031	1,859,609	1,334,203	267,525
Networks visited improvement	Baseline	1.32	1.30	1.68	3.43

We observe that using the precomputed log-gamma values allow us to compute the node scores faster, so that for each cache setting we can visit a larger number of networks. The relative advantage of the highest cache level shrinks somewhat, but only because the baseline scenario of using no cache at all performs the most computations, so it also benefits the most from the precomputed values.

Test 3: maxParentCount=10, precomputeLogGamma=no

Cache level	None	Basic	Fastlevel0	Fastlevel1	Fastlevel2
Number of networks examined	1,311,000	1,702,000	1,740,000	2,173,000	6,741,000
Number of Re-anneals	11	16	16	20	63
Memory used	21 mb	21 mb	21 mb	21 mb	21 mb
(Node) Scores computed	-8468.1	-8469.37	-8473.54	-8481.13	-8455.11
Scores computed	1,691,666	1,000,986	1,025,207	691,906	215,815
Networks visited improvement	Baseline	1.29	1.32	1.66	5.14

When we increase the maximum number of parents that we allow any variable to have, the performance for each cache level decreases, because computing scores for nodes with more parents are more expensive.

Test 4: When we set precomputeLogGamma to yes with maxParentCount=10 or larger, we cannot fit our problem in (512mb) of memory, even with no caching of node scores.

Finally, we compare what happens when we increase the maximum parent count, combined with potential cache and pre-compute logGamma choices. Again, the tests are short 1 minutes runs that one might perform to get a “feel” for different parameter choices, before running a full blown search (that may many hours).

Max. parent count	5	6	7	8	9	10	11	12
Cache level	Fastlevel2							
Pre-compute log-gamma	yes	yes	yes	no	no	no	no	no
Number of networks examined	8,388,000	8,586,000	8,564,000	6,885,000	6,826,000	6,730,000	6,423,000	6,552,000
Prep. time	1.0 s	3.2 s	12.6 s	0.265 s	0.281 s	0.329 s	0.421 s	0.766 s
Memory used	11 mb	42 mb	163 mb	2 mb	6 mb	21 mb	81 mb	321 mb
(Node) Scores computed	265235	270,105	272,409	225,323	224,198	215,955	206,893	211,800

We clearly observe the effects on the required memory and the prep time, when we increase the parent counts, especially when the “precompute logGamma values” is specified. Also note that even for the short 1 minute searches we already observe a drop in networks visited, due to the more expensive computation of scores with higher parent counts.

In a variation of this search we compare the lowest and highest cache levels when paired with the precompute-logGamma parameter. Again, we perform 1 minute searches using a Simulated Annealer searcher with a RandomLocalMove proposer.

Cache level + Precompute of logGamma	None + precompute logGamma=no	None + precompute logGamma=yes	Fastlevel2 + precompute logGamma=no	Fastlevel2 + precompute logGamma=yes
Number of networks examined	1,429,000	2,474,000	6,800,000	8,500,000
Number of Re-anneals	13	22	64	79
Memory used	1 mb	11 mb	1 mb	11 mb
High score	-8,492.8	-8,472.97	-8,463.58	-8,467.29
(Node) Scores	1,844,520	3,193,712	221,069	269,878

computed				
Networks visited improvement	Baseline	1.73	4.75	5.95

Here we want to point out the increase in performance related to the increase in cache use, as well as the increase in memory use due to the precomputing of the log-Gamma values.

However, you have probably noticed that the scores for the obtained top networks are not always in the order that we would expect. Well, since we are executing a random process, there is no guarantee that the number of networks searched is directly correlated to the quality of the obtained top-scoring network. On the other hand we only ran very short tests, and we will definitely increase our “odds” of finding better and better scores by searching over as many networks as possible.

Example: Varying the Cache Level in an Intermediate-size Problem

In this example we take a look at the effect of the cache for a larger problem of 320 variables and 33 observations, precompute Log-Gamma = yes, max. time = 1 minute.

Cache level	None	Basic	Fastlevel0	Fastlevel1	Fastlevel2
Number of networks examined	332000	336000	333000	331000	348000
Number of Re-anneals	2	2	2	2	3
Memory used	5 mb	5 mb	5 mb	6 mb	258 mb
(Node) Scores computed	436976	221625	219153	144640	137831
Networks visited improvement	Baseline	1.01	1	1	1.05

As expected, the cache setting has a larger influence on the memory use than in the 33 variable case, since the fast cache is proportional to the number of variables. On the other hand, the effect of the precomputeLogGamma setting is not as pronounced, because we have a smaller number of observations. Note that in this case a 1 minute search reveals no difference in performance between the different cache settings – this is simply due to the fact that we are still in the startup phase of the search, where the compute more scores than we retrieve from the cache (compare to the values in the 33 variables case!). The following table lists the memory use as we increase the maximum parent count:

Max. parent count	5	6	7	8	9	10	11	12
Cache level	Fastlevel2	Fastlevel2	Fastlevel2	Fastlevel2	Fastlevel1	Fastlevel2	Fastlevel1	none
Pre-compute log-gamma	yes	yes	yes	yes	yes	no	no	no
Prep. time	1.7 s	1.7 s	3.2 s	7.5 s	26.1 s	1.6 s	0.875 s	0.86 s
Memory used	258 mb	258 mb	275 mb	335 mb	301 mb	276 mb	325 mb	324 mb

Example: Varying the Cache Level in a Large-size Problem

In this example we take a look at the effect of the cache for a larger problem of 3200 variables and 33 observations. The following data is from 1 hour searches.

Cache level	None	Basic	Fastlevel0	Fastlevel1
Number of networks	178,000	178,000	180,000	176,000

examined				
Number of Re-anneals	0	0	0	0
Memory used	295 mb	296 mb	296 mb	374 mb
(Node) Scores computed	240,029	113,066	114,386	87,813
Networks visited improvement	(see comment)			

In this case the outcome is somewhat more drastic, since we cannot select the 2-parent fast cache without running out of memory (on a desktop machine with 1 gb of memory). Even after one hour into the search there is no noticeable difference between the results from the different cache settings. Note that we can expect to visit approximately as many networks as we visit in a 1-3 minute search for our 33 variable problem. Of course, there are many more values to cache, so we don't get as much of an acceleration effect as we do for smaller problems.

Example: Comparing Searchers

In this example we compare search results when combining our available searchers with the available proposers. Note that the quality of the search is not directly related to the number of networks visited, but instead also depends on the intrinsic capabilities of a searcher to visit different regions within the (very large) solution space. In our experience the simulated annealer method seems to produce the best results.

Searcher	Sim. Anneal	Greedy	Sim. Anneal	Greedy
Proposer	RandomLocal	RandomLocal	AllLocal	AllLocal
Networks visited	12,705,000	11,998,000	21,563,355	10,285,791
additions/ deletions/ reversals	4,251,588 4,22,8316 4,225,095	3,996,182 4,002,915 3,998,902	21,563,355 695,013 695,013	9,552,226 366,782 366,782
Top score	-8,457.15	-8,585.73	-8,476.19	-8,476.19
(Node) Scores computed	393,636	461,451	2,130	1,645,477
Scores computed for variables with 4 or 5 parents	32,997	11,417	0	68,051

Anologous results hold still true when we run with the same search parameters, but a search time of 60 minutes:

Searcher	Sim. Anneal	Greedy	Sim. Anneal	Greedy
Proposer	RandomLocal	RandomLocal	AllLocal	AllLocal
Networks visited	765,474,000	726,968,000	1,374,637,606	610,888,955
additions/ deletions/ reversals	256,132,104 254,670,930 254,670,965	242,332,695 242,322,670 24,2312,634	1,291,327,323 41,655,141 41,655,141	566,997,844 21,945,569 21,945,541
Top score	-8450.66	-8567.85	-8476.19	-8468.73
Scores computed	22,654,599	26,737,902	2,130	100,126,278
Scores computed for variables with 4 or 5 parents	1,950,474	574,571	0	4,120,024

Example: The Effects of Equivalence Checking on Performance

We now take a look at the various effects of the `bestNetworksAre` parameter, both in terms of performance and obtained results. Our underlying data is from our 33 variable problem, using a simulated annealer search with the `randomMove` proposer.

NBestNetworks=10, max. time = 5 m

BestNetworksAre	Nonidentical (obtained 10 <i>non-identical</i> networks only)	nonidenticalThenPruned (obtained 2 non- equivalent networks)	nonequivalent (obtained 10 non- equivalent networks)
Networks visited	66,552,000	65,821,000	66,423,000 (13,538 equivalence checks)
Number of re- anneals	626	617	624
Top score	-8453.97	-8459.01	-8457.67
Memory used	11 mb	11 mb	11 mb

We observe that when we search for a small number of `nBest` networks, there is not much difference in performance between the 3 choices. We want to point out, though, that only the `nonidentical` and the `nonequivalent` choices will provide us with exactly `nBest` networks. It is the nature of the pruning approach that will yield anywhere from 1 to `nBest` results when we use `nonidenticalThenPruned`.

NBestNetworks=100, max. time = 5 m

BestNetworksAre	Nonidentical (obtained 100 <i>non-identical</i> networks only)	nonidenticalThenPruned (obtained 11 non- equivalent networks)	nonequivalent (obtained 100 non- equivalent networks)
Networks visited	67,007,000	66,617,000	60,322,000 (678,080 equivalence checks)
Number of re- anneals	631	627	567
Top score	-8455.13	-8454.56	-8461.37
Memory used	12 mb	12 mb	12 mb

NBestNetworks=1000, max. time = 5 m

BestNetworksAre	Nonidentical (obtained 1000 <i>non-identical</i> networks only)	nonidenticalThenPruned (obtained 231 non- equivalent networks)	nonequivalent (obtained 1000 non- equivalent networks)
Networks visited	66,657,000	65,581,000	49,000 (8,033,553 equivalence checks)
Number of re- anneals	626	618	0
Top score	-8461.97	-8455.18	-9465.26
Memory used	16 mb	13 mb	17 mb

We notice that as we increase the `nBest` setting, the performance the “`nonequivalent`” is severely impacted by the number of equivalence checks necessary to maintain the set of `nonequivalent` networks during the search.

Example: Comparing different Cycle Checking Methods

This is a comparison between the different choices for cycle checking. Note that for Banjo 2, the choices are a new depth-first search (dfs), a variation of the dfs using an optimization based on the paper by Oded Shmueli, and the (non-recursive) dfs implementation from version 1. It turns out that all 3 implementations produce relatively similar results. The main surprise is the fact that the Shmueli optimization is not very effective whenever we allow reversals, because whenever we need to undo a reversal, we encounter a penalty for having to bring the

“tracking numbers” back up to date. However, as we show in the second example on experimenting with cycle checking, when we restrict our proposer to additions and deletions only, then the Shmueli-based optimization produces better performance than the other methods.

Cycle-checking method	Dfs	Dfs from version 1	Dfs with Shmueli optimization
Networks visited	8,469,000	8,155,000	8,292,000

Example: Cycle Checking Methods, Revisited

It turns out that by using additions, deletions, and reversals as part of our standard proposers, the optimization of the Shmueli-variation of the depth-first search is somewhat lost due to the necessary adjustments every time we discard a reversal and need to revert back to a previous network. This effect is illustrated by looking at a slightly modified proposer that doesn't use reversals. Note that this test requires a simple code change to the internal constant `CONFIG_OMITREVERSALS`.

Cycle-checking method	Dfs	Dfs from version 1	Dfs with Shmueli optimization
Networks visited	11,255,000	10,465,000	12,141,000

We wanted to present this fact to the user, because it may come in handy when examining large problems where score computations tend to be expensive, and where it may be desirable to visit as many networks as possible just to obtain the full effect of the score caching.

Advanced Features

Banjo contains a few advanced features that may be of interest to some users. These include expanded post-processing options including the automatic generation of a graph using dot, the computation of influence scores for each node with a parent, the computation of a consensus graph, and how to use Banjo to find non-equivalent networks.

Post-Processing Options

All post-processing options that were – prior to Banjo 2 – only controllable via internal constants, are now fully accessible via input settings. This includes the following:

- Creating the graphical representation of the top-scoring network using dot.
- Creating the graphical representation for the consensus graph.
- Creating a text file with the commands for creating dot output, for both the top-scoring and the consensus graph.
- Creating a table representation of the nodes that are part of the consensus graph, in html format.
- Computing the influence scores for the nodes in the top-scoring network.

The *createDotOutput* setting controls whether dot graphics files are produced for the top scoring network and the consensus graph (not necessarily a network). To function properly, the user will have to supply the path to the dot executable, via the *fullPathToDotExecutable* setting (on a standard graphViz install, “C:/Program Files/ATT/Graphviz/bin/dot.exe” on a Windows PC).

The *computeInfluenceScores* setting instructs Banjo to compute the influence scores for the top scoring network. The *computeConsensusGraph* setting is for computing the consensus graph from the N highest-scoring networks. The *createConsensusGraphAsHtml* setting outputs the consensus graph together with the N best networks in form of a simple html table, for convenient review.

At a more granular level, the *dotGraphicsFormat* setting provides access to a number of the graphics formats that dot supports (e.g., jpg, imap, vml, pic, ps, etc). The *dotFileExtension* setting adds an extension to the graphics file; Ditto for the *htmlFileExtension*.

The *fileNameForTopGraph* and the *fileNameForConsensusGraph* specify the file names for the top-scoring and the consensus graph, respectively.

Note that by embedding a token string (@*time_stamp*@), one can specify the time when the search has started within the result file names. For additional customization, one can change the default time stamp, by simply providing a valid Java time format for the *timeStampFormat* setting.

A special searcher (“Skip”) skips the regular search part, and proceeds directly to the post-processing. This enables us to keep the search-specific setting values in our settings file without performing an actual search.

Using dot to Generate a Graph Representing the Found Network

The free GraphViz library from AT&T has powerful capabilities for laying out graphs and creating images of networks such as the ones we try to discover with Banjo. The *dot*

application is responsible for graph layout within GraphViz. For instructions on the use of GraphViz (it's very easy) and for downloading the library, visit <http://www.graphviz.org/>.

GraphViz *dot* lays out graphs and then generates images. It takes as input graph files that have a particular format. For the highest scoring network it finds, Banjo creates a set of instructions in this format so that they can be passed to *dot*. You can use this output to create a picture of the learned network. Just copy the *dot* commands from the Banjo output, and save them to a file.

As an example, suppose Banjo creates the following set of *dot* commands for its highest scoring network:

```
digraph abstract {
label = "Banjo Version 1.0\nHigh scoring network, score: -6764.73\nProject:
Banjo_dev\nUser: hjs\nData set: 33-vars-320-cases ";
labeljust="l";

    2->17;
    4->3;
    5->16;
    7->1;
    8->16;
    9->11;
    9->13;
    9->25;
    9->29;
    10->1;
    11->1;
    12->0;
    12->8;
    13->23;
    15->4;
    16->14;
    17->14;
    17->25;
    18->24;
    19->0;
    21->15;
    21->16;
    22->15;
    22->29;
    24->7;
    25->1;
    26->17;
    31->11;
}
```

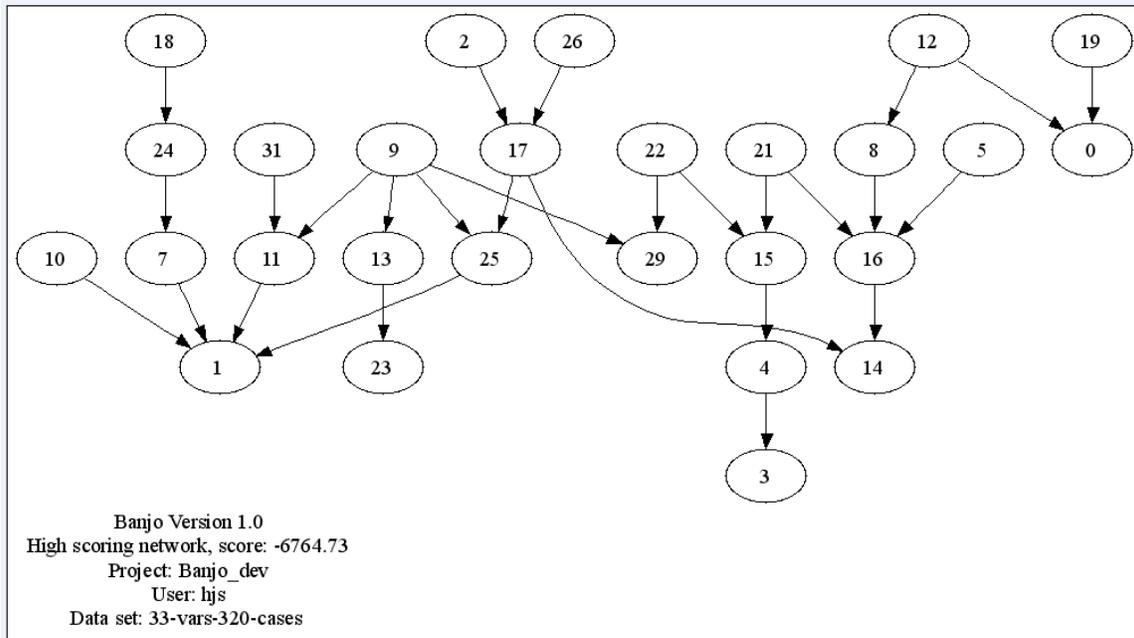
Suppose that we save the *dot* commands in the file *graph.dot*. The program within the GraphViz package that interprets the *dot* commands and generates an image is called *dot.exe* in Windows. If the program is installed to *C:\ATT\Graphviz\bin*, then we can generate an image by opening the command shell, changing to the directory containing *graph.dot*, and typing:

```
C:\ATT\Graphviz\bin\dot.exe -Tgif graph.dot -o graph.gif
```

Note that the flag “-Tgif” specifies the type of output graphics file, and “-o” specifies the name of our output file. Provided that the *dot* executable is in your path, the corresponding command in Linux, Unix, or Mac OS X would be:

```
dot -Tgif graph.dot -o graph.gif
```

The output image file `graph.gif` then looks like this:



For dynamic Bayesian networks with min. Markov lag equal to the max. Markov lag, we use a collapsed representation of the network. For all other dynamic Bayesian networks, we display the nodes in a format that indicates the lag, e.g., node 2 of lag 1 would be denoted by “(lag 1) 2”. Adding additional representations simply requires you to modify the existing code for the `composeDotGraph` method in the `PostProcessor` class.

Banjo 2 expands on the use of `dot`, by automating the creation of the graphics file. In order to take advantage of this capability, we need to specify the location of the `dot` executable in the settings file, as well as the names of the files where we want Banjo to place its output. Here is an example:

```
# Excerpt from the settings file:  
  
createDotOutput =           yes  
computeConsensusGraph =    yes  
  
fullPathToDotExecutable =   C:/ATT/Graphviz/bin/dot.exe  
fileNameForTopGraph =       @_time_stamp@.graph.top  
fileNameForConsensusGraph = @_time_stamp@.graph.consensus  
  
dotGraphicsFormat =         jpg  
dotFileExtension =          txt  
  
timeStampFormat =           yyyy.MM.dd.HH.mm.ss
```

Notes:

1. The specified path to the `dot` executable in `fullPathToDotExecutable` must be specified using forward slashes (important for Windows users). If Banjo cannot find `dot`, no graphic output will be created, and an error message will be recorded.
2. The file names specified in `fileNameForTopGraph` and `fileNameForConsensusGraph` can contain a (relative) path. All directories that are being specified need to exist in the

outputDirectory that Banjo uses for all its output. Both *fileNameForTopGraph* and *fileNameForConsensusGraph* can also contain a time stamp, which is specified by embedding any of the time stamp tokens (e.g., *@timestamp@*) anywhere in the file name (only). A time stamp token in the directory part of the string will cause an error, since Banjo is not capable of creating directories on the fly.

3. By specifying the (optional) *dotGraphicsFormat*, you can control the format of the graphics that *dot* produces.
4. The (optional) *dotFileExtension* is used for the accompanying *dot* text file that contains the instructions for creating the graphics.
5. The (optional) *timeStampFormat* lets you specify the exact time stamp that you may want to use.

Executing Banjo 2 with the above settings would produce 2 graphics files, one each for the top graph and the consensus graph, with *.jpg* extension, and 2 structure files with the *dot* commands, with *.txt* extension. The time stamp token would be replaced with the current time in the specified format (the time “snapshot” is taken when the search starts), e.g., the file name for the top graph could be *_2006.03.31.00.25.28_graph.top.jpg*, located in *(myproject/output)/graph*.

Another new feature of Banjo version 2 is the *variableNames* setting, which lets the user specify a comma-delimited list of labels for each of the variables, to be used in the creation of the graphics file via *dot*. Note that the standard Banjo structure output to file still uses the variable indexes instead of the labels, so that the structures can easily be re-imported into Banjo.

Influence Scores

An influence score is a metric for representing the degree to which a parent variable’s influence on a child is monotonic in nature, and if so, in what direction (positive or negative) and what magnitude. A more detailed description of influence scores, and how they are computed, can be found in a paper by Yu, et al., in *Bioinformatics* (2004). For our purposes, it should simply be noted that a value of zero does not mean that a parent has no influence on a child, but simply that the influence is not definitively positive or negative in nature, given the observed data.

Banjo automatically computes and displays the influence scores for the top-scoring network found in the search. Suppose the top-scoring network is represented by the data below.

33
0 1 32
1 1 5
2 1 12
3 1 26
4 0
5 1 16
6 0
7 1 17
8 1 16
9 0
10 0
11 1 16
12 0
13 1 7
14 1 28
15 0
16 0
17 1 27
18 0
19 0

```
20 2 7 11
21 1 16
22 0
23 0
24 0
25 1 9
26 0
27 0
28 0
29 0
30 0
31 2 16 26
32 0
```

The influence scores for all nodes that have a parent might be computed and printed out as follows.

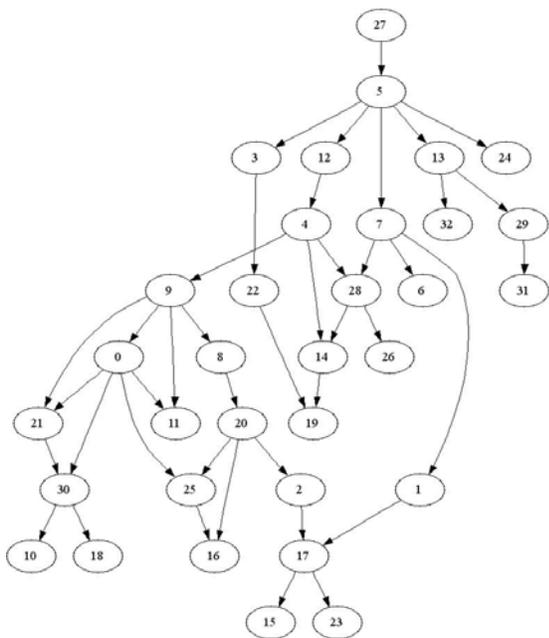
```
Influence score for (32,0) -> (0,0) 0.0
Influence score for (5,0) -> (1,0) 0.0
Influence score for (12,0) -> (2,0) 0.0
Influence score for (26,0) -> (3,0) 0.0
Influence score for (16,0) -> (5,0) 0.0
Influence score for (17,0) -> (7,0) 0.0
Influence score for (16,0) -> (8,0) 0.0
Influence score for (16,0) -> (11,0) 0.0
Influence score for (7,0) -> (13,0) 0.0
Influence score for (28,0) -> (14,0) 0.5934
Influence score for (27,0) -> (17,0) 0.5500
Influence score for (11,0) -> (20,0) 0.2116
Influence score for (7,0) -> (20,0) 0.0
Influence score for (16,0) -> (21,0) 0.0
Influence score for (9,0) -> (25,0) 0.0
Influence score for (26,0) -> (31,0) 0.0
Influence score for (16,0) -> (31,0) 0.0
```

In Banjo 2 you can control the computation of influence scores by specifying the new (optional) setting *computeInfluenceScores*, with values *yes* and *no*.

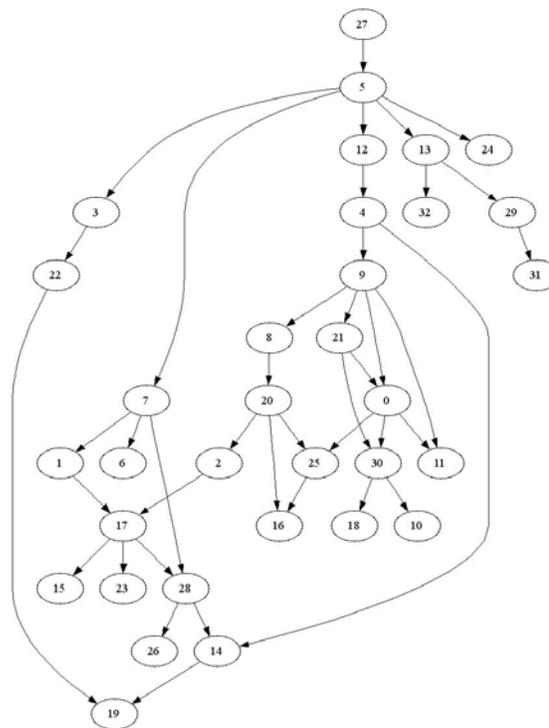
Consensus Graph

Banjo 2 provides a post processing option of computing a “consensus” graph from a set of high-scoring networks, by assigning exponentially weighted probabilities to the individual edges in each of the high-scoring networks, based on the “ranking” of each network in the set.

Note that the consensus graph does not need to be a valid network structure, i.e., a consensus graph can contain cycles.



Banjo Version 1.1
Consensus graph (based on top 10 networks)



Banjo Version 1.1
High scoring network, score: -8798.58

The 2 pictures above show the consensus graph and the top scoring graph, based on a search that found 10 top-scoring networks.

As an aid for comparing the consensus graph to the high-scoring networks that it is based on, we added an option to output all structures in the form of an html table. Column 1 (“Var”) contains the index for each variable, column 2 (“Consensus”) contains the edges for the consensus graph, and columns #1 to #N contain the edges for the N top scoring networks.

Var	Consensus	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
0	9	9 21	9	9 11	9	9 21	9 11	9 21	9	9 11	9
1	7	7	7	7	7	7	7	7	7	7	7
2	20	20	20	20	20	20	20	20	20	20	20
3	5	5	5	5	5	5	5	5	5	5	5
4	12	12	12	12	12	12	12	12	12	12	12
5	27	27	27	27	27	27	27	27	27	27	27
6	7	7	7	7	7	7	7	7	7	7	7
7	5	5	5	5	5	5	5	5	5	5	5
8	9	9	9	9	9	9	9	9	9	9	9
9	4	4	4	4	4	4	4	4	4	4	4
10	30	30	30	30	30	30	30	30	30	30	30
11	0 9	0 9	0 9	9	0 9	0 9	9	0 9	0 9	9	0 9
12	5	5	5	5	5	5	5	5	5	5	5
13	5	5	5	5	5	5	5	5	5	5	5
14	4 28	4 28	4 28	4 28	4 28	4 28	4 28	4 28	4 28	4 28	4 28
15	17	17	17	17	17	17	17	17	17	17	17
16	20 25	20 25	20 25	20 25	20 25	20 25	20 25	20 25	20 25	20 25	20 25
17	1 2	1 2	1 2	1 2	1 2	1 2	1 2	1 2	1 2	1 2	1 2
18	30	30	30	30	30	30	30	30	30	30	30
19	14 22	14 22	14 22	14 22	14 22	14 22	14 22	14 22	14 22	14 22	14 22
20	8	8	8	8	8 14	8 14	8 14	8	8	8	8
21	0 9	9	0 9	0 9	0 9	9	0 9	9	0 9	0 9	0 9
22	3	3	3	3	32	32	32	3	3	3	3
23	17	17	17	17	17	17	17	17	17	17	17
24	5	5	5	5	5	5	5	5	5	5	5
25	0 20	0 20	0 20	0 20	0 20	0 20	0 20	0 20	0 20	0 20	0 20
26	28	28	28	28	28	28	28	28	28	28	28
27											
28	4 7	7 17	7 17	7 17	4 7	4 7	4 7	4 7	4 7	4 7	4 7
29	13	13	13	13	13	13	13	13	13	13	13
30	0 21	0 21	0 21	0 21	0 21	0 21	0 21	0 21	0 21	0 21	0 21
31	29	29	29	29	29	29	29	29	29	29	29
32	13	13	13	13	13	13	13	13	13	13	13

Finding Non-equivalent Networks

The N top-scoring networks obtained by a search in Banjo 1.0 were a set of non-identical networks, some of which were potentially (actually: likely) equivalent structures. Banjo 2 provides the option to also obtain sets of non-equivalent structures. To access this new feature, one can specify the optional “nBestNetworksAre” setting. The default value remains the Banjo 1.0 behavior of “nonIdentical”. The listing of the obtained results will indicate the selected option.

```

-----
- Best 10 non-identical Structures
-----

Network #1, score: -9685.83, first found at iteration 59177
...

```

Banjo 2 provides 2 choices for obtaining a set of non-equivalent networks: “nonEquivalent” and “nonIdenticalThenPruned”.

When “nonEquivalent” is selected, the search will compare potential high-scoring networks for equivalence whenever a new high-scoring network is encountered. The actual test that we employ makes use of several “shortcuts” via already available internal data, but still is extremely time consuming. Due to the large amount of additional computations necessary for

comparing 2 networks for equivalence one can expect that the runtime performance for a “nonEquivalent” search will be an order of magnitude slower than a search that only employs identity checking.

This performance issue led us to implement the special “hybrid” option, called “nonIdenticalThenPruned”, which performs a search using the identity checking during the search, then prunes away any equivalent networks every time a restart or reannealing is performed, as well as after the search is complete.

```
-----  
- Best 8 non-identical then pruned Structures  
-----  
  
Network #1, score: -9893.69, first found at iteration 57937  
...
```

There is one inherent drawback to this hybrid approach, though: While the performance of a “nonIdenticalThenPruned” search is close to the performance of a “nonIdentical” search, the nonIdenticalThenPruned option will likely not yield a “full” set of N top-scoring networks, but instead a subset anywhere from 1 to N non-equivalent networks. The reason for this lies in the “bumping” of non-equivalent networks by a potentially large number of higher scoring (but possibly – in fact, very likely – equivalent) networks found further into the search process.

Using Banjo in Matlab

Since the release of Matlab 6, it has been possible to run Java programs from within Matlab. With only a little bit of work, we can get Banjo to run in Matlab.

Although Matlab integrates heavily with Java, you may notice a decrease in performance when running Banjo from within Matlab. This is most likely due to translating the user's requests into commands in Java, and thus will probably not dramatically affect the running time of a long search.

Change to the Correct Directory

The first step is to change to the directory in which Banjo is stored. Open up Matlab and change the directory to the location of `banjo.jar` or the `edu/` directory if you extracted the jar file already. For example, if the jar file is located at `C:\code\banjo.jar` or if you extracted the jar file to `C:\code`, you would type:

```
cd C:\code
```

Update the classpath

Next, you need to tell Matlab the location of the Banjo class files. If you wish to use the jar file, simply type:

```
javaclasspath('banjo.jar');
```

If you are not using the jar file, you must be in the directory that `edu/` is located in. Note that this must be an absolute pathname.

```
javaclasspath('C:\code\');
```

Run the Banjo Program

Now that the classpath has been set, we need to import the classes that we would like to run. Since we only want to run the Banjo class, we only need to import the application package. Assuming that the settings file is `my.settings.txt`, running Banjo just requires a simple call to the main function:

```
import edu.duke.cs.bayes.application.*;  
Banjo.main('settingsFile=my.settings.txt');
```

You will notice that nothing initially prints out on the screen. Unfortunately, everything that is normally printed to the console as the program runs is buffered internally in Matlab. Thus, the feedback and search results will only be printed after the program has completed running.

Specifying Additional Options

To provide several options to the Banjo program that would normally be specified on the command line, you must use an array of strings. An array of strings in Matlab is specified in brackets `{}`. For example, if we wished to additionally specify that the user is john, the new command would be:

```
Banjo.main( {'settingsFile=my.settings.txt', 'user=john'} );
```

More options can be specified the same way: with comma separated strings between brackets.

Hints and Tips

Computing a Network Score without Running a Search

To have Banjo compute the score of a single, initial network only, simply set the `searcher` setting to “Skip”. Specify the `initialStructure` setting to point to the file that describes the structure. Then execute Banjo. This will ignore any termination criteria in the settings file, compute the network score, and then proceed straight to the post-processing options.

By using the command line option you can conveniently keep all the values in your settings file unchanged; simply add “`searcher=Skip initialStructure=filename`” to the end of the command string for running Banjo.

Displaying Debug Info

When encountering a problem while running Banjo as a jar file, the `displayDebugInfo` setting, when set to “yes”, can reveal additional information (by displaying the stack trace) without having to load Banjo into a development environment.

Adding Structure Files to Output

In case any of the structure files, such as `initialStructure`, `mustBePresentEdges` or `mustNotBePresentEdges`, are specified, they can be displayed (and saved in the results file) as part of the output, by setting the `displayStructures` setting to “yes”.

Using Time Stamps in Output Files

Using a time stamp as part of a file name provides a simple mechanism to save the results for each search in individual files. Simply embed a token (`@time_stamp@`) anywhere in the file name string, and Banjo will replace it at run time with the current time/date, based on the (optional) time stamp format specified by the `timeStampFormat` setting. The default time stamp format is of the form “`yyyy.MM.dd.HH.mm.ss`”, where `yyyy` = year in 4-digits, `MM` = month (1 to 12), `dd` = day, `HH` = hour in 24h notation, `mm` = minutes, `ss` = seconds, of the current time and date. The format follows the Java time stamp format.

The time stamp can be applied to the report file, as well as the file names for the top graph and the consensus graph.

Memory Info and Performance Tuning

The `displayMemoryInfo` can come in handy when running Banjo on large data sets that may reach the computer’s limit of available memory. When set to “yes”, the memory used by Banjo is being displayed at every regular feedback interval. This makes it a little easier to tune the `useCache` setting that controls to a large extent the amount of memory that Banjo uses, above a minimum amount for running a very basic search.

Banjo uses 2 different types of cache: a basic cache that stores a limited number of (node) scores in a variable container, and what we call “fastCache”, which stores (node) scores for all variables of a given maximum parent count in an array for fast retrieval. The fast cache can handle a maximum parent count of up to 2, and can be used in addition to the basic cache. By

default, all cache settings are turned “on”, i.e., *useCache* is set to *fastLevel2*. However, for problems with several hundreds or several thousands of variables the cache use generally has to get dialed down to keep Banjo from running out of memory.

For additional info about tuning a search, please consult the *Setting up your Search* section.

Accessing Additional Options via Internal Code Changes

Banjo 2 has replaced most of the internal configuration constants with settings options. However, there are a few settings that are somewhat in conflict with regular Banjo behavior, but that can be useful especially to developers.

A prime example is the use of the *variableCount* setting with values that differ from the number of data entries supplied by the observations file. When Banjo operates “normally”, such a difference will be flagged as an error – as a user would come to expect:

```
[ERROR: Banjo 2, 10/10/06 12:55:35 PM]
The info below has been gathered by the application:

(ObservationsAsMatrix.LoadData) Observation #1 in observations file
'static.data.txt' contains 33 data points instead of the expected 31.
```

On the other hand, being able to run different search scenarios from the same data set is very convenient for developers. So we provide a “magic” switch in form of the *DEBUG* constant in the *BANJO* file. When set to *true*, validation checks such as for the variable count are skipped in the code.

In addition to altering the Banjo behavior, the *DEBUG* switch also provides additional internal trace feedback in several core parts of the code. This makes it easy, e.g., to monitor the memory use by individual components during setup of a search.

Unique Output File Names

Banjo 2 allows the use of time stamps in various file names for its output, by inserting the *@timestamp@* token anywhere in the file name. The use of time-stamped output files comes in handy when you need to make repeated searches with the same or similar settings, since the result for each search can be stored in an individual file. If the main result file happens to have the same name as an earlier result file, Banjo proceeds as in version 1, and appends the new results to the existing file.

Note: If you don’t use timestamps, but use the automatic generation of graphics files via the dot program, you want to be aware that your graphics output will be overwritten with the new file in case you forget to change the graphics file name(s).

The default time stamp format programmed into Banjo is of the form “*yyyy.MM.dd.HH.mm.ss*”, where *yyyy*=year, *MM*=month (numeric value), *dd*=day of the month, *HH*=hours in 24h clock, *mm*=minutes, *ss*=seconds. You can specify your own time stamp format by specifying the optional *timeStampFormat* value in the settings file.

In addition to using time stamps in organizing your output, it is also possible to include (relative) path info as part of a file name (Use forward slash as directory indicator). When used in this way one can route all graphics files into a separate subdirectory of the input directory,

and the text files into another one. Note: Any such path info cannot go to the parent directory, but has to be a proper subdirectory of the input directory. If you use this feature, you need to be careful in spelling your file name, because Banjo does not look for any directory info, and thus does not create any non-existing directories for you. Instead you may end up with an error message similar to this one:

```
-----  
(Final Checkpoint) A final check revealed the following issues that were  
encountered during Banjo's execution:  
-----  
(Post-processing) The 'dot' output could not be created. Detail info:  
'(PostProcessor.execute) When the file name 'top/top.graph.2006.10.12.15.31.13'  
for the top graph is combined with the Output directory  
'data/release2.0/static/output', the result does not form a valid path to a file.  
Please make sure that all specified directories exist!'.  
-----
```

Combining Multiple Observations Files

Even though this has been a feature in version 1, it has not been well documented. If your observations are located in multiple files, you can easily “combine them” for the purpose of a Banjo search, by simply specifying them in the settings file as a comma-separated list, or using wildcard notation. And in case you have a set of files in your input directory from which you want to exclude one or more observations files for a particular search, you can do that by listing the “to-be-excluded” file with a “-”-prefix. Of course, this means that you cannot name your observation files with names that start with a “-”-sign.

Note that when setting up a search for a dynamic network, where the data is based on possibly several independent (time based) experiments, each data file will be treated as a separate “entity” not related to the data in any other supplied data file (which is the expected behavior, of course). Consequently, all data from the same experiment for the dynamic case needs to be supplied in the same file.

This explains the separate feedback in the Banjo results file when examining dynamic networks, displaying a value for “observations in file” and for “observations used”.

Error Reporting to File

Prior to version 2, Banjo would report errors only to a special errors file, located in the directory of the Banjo executable. Starting with version 2, errors will also be reported in the standard results file.

More Flexible Structure Files

The less stringent input processing of Banjo 2 lets us skip lines in a structure file for variables that don't have any parents. Any unspecified variable is assumed to have no parent (unless a `dbnMandatoryLag` is used for a dynamic bayesnet). In addition, we don't enforce the order in which the variables and their parents are listed.

Specifying Observations in Row or Column Format

The default for the observations file is the column oriented format that was already in use in Banjo 1.0, i.e., each row in the observations file contains a single observation with values for all of the variables.

Banjo 2 adds the option to supply the observations in a “transposed” format, where each row contains the values of a variable for all observations. To input observations in this format, the optional setting *variablesAreInRows* needs to be set to “yes”

Specifying Names for the Variables

To assign names to the used variables, we can use the *variableNames* setting. Several options are available:

- Provide a list of the names in white-space delimited format in the settings file. If you need to use white-space within your variable names, then prefix your list with “commas:”, followed by the comma delimited list of variable names. [In case you need a different delimiter, you can change the choice of delimiter by editing a constant in the code].
- Or, set the value of *variableNames* to “inFile” , and provide the variable names in the actual observations file as follows:

For the standard (each observation in a row) observations format, the list of variable names needs to be supplied as the very first non-blank and non-commented text line in the file.

For the “transposed” observations format, the first entry of each line needs to be the name of the variable for that row of data.

Whenever the “inFile” option is to be used, the *observationCount* needs to be supplied in the settings file, to protect against unintended side effects from interpreting the first data row or column mistakenly as variable names.

When supplied in the observations file, the variable names can only be separated by white-space.

Note that the “inFile” option cannot be used when multiple observations files are specified.

Using a Greedy Searcher with the AllLocalMoves Proposer

When using the AllLocalMoves proposer, especially in conjunction with the Greedy searcher, one needs to be especially careful when specifying the search parameters. The values of the parameters *minProposedNetworksBeforeRestart* and *maxProposedNetworksBeforeRestart* are of particular importance, and need to be set based on the size of the underlying problem. Remember that allLocalmoves will examine in the order of N^2 networks in each individual step, so we want to allow the search to visit a meaningful number of networks before restarting. A common mistake is to choose values that lead the search to be very shallow with respect to the restart networks.

It should be obvious that Greedy and AllLocalMoves should only be used when the parameter *restartWithRandomNetwork* is set to yes, to avoid repeating a partial search over and over.

We found that in some problems the greedy search converges very fast, but it is always a good habit to check by running a simulated annealer search to make sure that there is nothing wrong with the way the greedy search is set up.

Troubleshooting

When working with a new software program it is important to know what to do when something doesn't work. In this section, we describe the main sources of problems that a user may encounter when working with Banjo and how to go about finding solutions. The problems themselves can be categorized into a number of different types, each described in detail below.

When Little Things Don't Work As Expected

Missing or Invalid Parameter in the Settings File

If a required parameter is not specified for executing a particular search strategy, Banjo's internal validation will catch this problem, and display a short report. The Banjo 2 feedback includes the Banjo header section and project data, as well as a listing of all issues with the user-supplied settings that it can safely determine. The main types of issues reported are "missing value of a required setting", "value out of accepted range", "invalid choice", "rule violation", and "wrong data type".

Examples (with error types highlighted in bold face):

```
-----  
- Banjo                      Bayesian Network Inference with Java Objects -  
- Release 2.0                  1 Apr 2007 -  
- Licensed from Duke University -  
- Copyright (c) 2005-2007 by Alexander J. Hartemink -  
- All rights reserved -  
-----  
[ERROR: Banjo 2, 8/7/06 12:51:14 PM]  
(Checkpoint) Cannot continue without a valid searcher.  
(Invalid setting choice) The supplied value 'default' is not a valid option for  
the setting 'searcherChoice'.  
-----  
End of error notification  
-----
```

In this case Banjo lets us know that the searcherChoice setting needs to be specified with a value that is the set of acceptable values (in this case, "simulated annealing", "greedy", or "skip"). While "default" is a valid choice for several of the core objects that are being used by the search, it cannot be used for specifying the searcher. Note that any setting that expects a string from a prescribed set of values is now validated in this manner.

Note that Banjo cannot continue its validation (what searcher would it use?), so it indicates that it stopped at a checkpoint, and it lists the issues that it encountered so far.

```
-----  
ERROR DETAILS  
-----  
- Banjo                      Bayesian Network Inference with Java Objects -  
- Release 2.0                  1 Mar 2007 -  
- Licensed from Duke University -  
- Copyright (c) 2005-2007 by Alexander J. Hartemink -  
- All rights reserved -  
-----
```

```

- Project:
- User:
- Dataset:
- Notes:
-----

[ERROR: Banjo 2, 8/7/06 12:46:12 PM]
(Data is of unexpected type) The value for the setting 'Min. Markov lag'
(minMarkovLag = '3.5') is not of the correct data type (expected: Integer).
(Missing value of required setting) The value for the setting 'Max. Markov lag'
(maxMarkovLag = '') needs to be supplied.
(Value out of accepted range) The value of the setting
'numberOfIntermediateProgressReports' ('-10') needs to be greater than 0.
(Value out of accepted range) The value of the setting 'maxParentCount' ('-3')
needs to be greater than 0.
-----

                        End of error notification
-----

```

First, Banjo tells us that the *minMarkovLag* needs to be an integer.

Next, Banjo is complaining about the missing value for the required setting *maxMarkovLag*. We need to make sure that the settings file contains a line with this setting name and its value. In the case of our example, we would add

```

maxMarkovLag=<value>

```

with <value> filled in with the desired value (a non-zero integer greater or equal to the *minMarkovLag*).

Finally, Banjo also flags several settings with values that cannot be used as supplied: The *numberOfIntermediateProgressReports* and the *maxParentCount* all need to be positive numbers.

Note that Banjo again stops with its validation at this point. Any errors that would be found in code that would depend on the successful validation of the searcher would not execute until the flagged issues are resolved.

```

-----
- Banjo                               Bayesian Network Inference with Java Objects -
- Release 2.0                          1 Apr 2007 -
- Licensed from Duke University         -
- Copyright (c) 2005-2007 by Alexander J. Hartemink -
- All rights reserved                   -
-----

- Project:
- User:
- Dataset:
- Notes:
-----

[ERROR: Banjo 2, 8/7/06 1:12:58 PM]
(Data is of unexpected type) The value for the setting 'Max. parent count'
(maxParentCount = 'three') is not of the correct data type (expected: Integer).
(Rule violation) The value of the setting 'minMarkovLag' ('3') needs to be less
than or equal to that of setting 'maxMarkovLag' ('1').

```

In this example Banjo complains that the numeric value for *maxParentCount* is not a number (in fact: it tells us that we need to enter an integer), and that the *minMarkovLag* is not less than or equal to the *maxMarkovLag*.

Miscellaneous Errors and Warnings

For some errors, e.g. when encountering a cycle in a structure file, you'll notice that Banjo will display a complete list of the already validated settings, as well as the loaded "raw" values. This can assist in the trouble-shooting and crosschecking of the supplied input data.

Structure files, used for example to supply an initial network structure or the `mustBePresentEdges` file, need to represent a valid network, i.e., they cannot contain cycles. In addition, the `mustBePresentEdges` and `mustNotBePresentEdges` have to be consistent with respect to each other (i.e., they cannot overlap).

Finally, for some minor issues with the input data, Banjo may only report a warning. This happens, e.g., when the number of supplied variable names does not match the number of variables. In this case Banjo will execute the search after displaying a brief message stating the issue.

Problems During Post-Processing

As the list of post-processing options grew, we wanted to keep the execution of the different post-processing code sections as independent from each other as possible. To achieve this we employ a separate error tracking mechanism for each option, with its own output "checkpoint". If Banjo encounters a problem, it will keep track of it, but try to continue with the remaining post-processing options. A sample output may look similar to this:

```
-----  
(Final Checkpoint) A final check revealed the following issues that were  
encountered during Banjo's execution:  
-----  
(Post-processing) The influence scores could not be computed.  
Detailed info: '(InfluenceScorer constructor) Error in executing the  
InfluenceScorer constructor.'  
(.dot' execution) The attempted execution of '.dot' to create the graphics file  
'data/release2.0/static/output\consensus.graph.2006.10.10.10.08.21.jpg' did not  
succeed. No output has been produced.
```

Error During Program Execution: Handled by Banjo

Here we encounter a problem that occurred during regular program execution: The Banjo code encounters (internal) data that it doesn't expect, and stops the program. A typical example could be the discovery of a cycle in a user-supplied structure: the optional initial structure file cannot contain a cycle.

```
-----  
                          ERROR DETAILS  
-----  
- Banjo                      Bayesian Network Inference with Java Objects -  
- Release 2.0                  1 Apr 2007 -  
- Licensed from Duke University -  
- Copyright (c) 2005-06 by Alexander J. Hartemink -  
- All rights reserved -  
-----  
- Project:                      banjo static example  
- User:                          demo  
- Dataset:                      33-vars-320-observations  
- Notes:                        static bayesian network inference  
-----
```

```

[ERROR: Banjo 2, 10/10/06 10:39:50 AM]
The info below has been gathered by the application:

The must-be-present edges file 'static.mandatory.with.cycle.str' contains a cycle.
-----
                        End of error notification
-----

```

Many of these errors are easily fixable by correcting the data.

Another type of error may be caused by an unexpected combination of data that is not handled properly by the Banjo code. This would be a bug in the software and has to be corrected by the developer.

```

-----
                        ERROR DETAILS
-----
- Banjo                Bayesian Network Inference with Java Objects -
- Release 2.0                1 Apr 2007 -
- Licensed from Duke University -
- Copyright (c) 2005-06 by Alexander J. Hartemink -
- All rights reserved -
-----
- Project:                banjo static example
- User:                    demo
- Dataset:                33-vars-320-observations
- Notes:                  static bayesian network inference
-----

[ERROR: Banjo 2, 10/10/06 11:00:54 AM]
[Development-related error: This message is usually generated to remind the Banjo
developer to complete or restructure a section of code]
Error details: Please notify the developer that the application provided the
following info:
(BayesNetManager.applyChange) Development issue: Can only apply a BayesNetChange
with READY status, but encountered status value = '1'.
-----
                        End of error notification
-----

```

Hopefully, such instances will be rare.

Error During Program Execution: “Insufficiently Handled” by Banjo

In this situation, an error is being handled by the Banjo code, but the error message doesn't provide enough information. Quite possibly this is due to an oversight of the developer, as the following example illustrates.

```

-----
                        ERROR DETAILS
-----
- Banjo                Bayesian Network Inference with Java Objects -
- Release 2.0                1 Apr 2007 -
- Licensed from Duke University -
- Copyright (c) 2005-06 by Alexander J. Hartemink -
- All rights reserved -
-----
- Project:                banjo static example
- User:                    demo
- Dataset:                33-vars-320-observations
- Notes:                  static bayesian network inference
-----

[ERROR: Banjo 2, 10/10/06 10:47:46 AM]

```

```
[Development-related error: This message is usually generated to remind the Banjo developer to complete or restructure a section of code]
Error details: Please notify the developer that the application provided the following info:
(BayesNetStructure constructor) Developer issue: 'bayesNetStructure' object is of unknown data type.
```

```
-----
                        End of error notification
-----
```

Note that an extended error reporting section can be displayed using the “displayDebugInfo = stackTrace” setting:

```
edu.duke.cs.banjo.utility.BanjoException: (BayesNetStructure constructor) Developer issue:
'bayesNetStructure' object is of unknown data type.
    at edu.duke.cs.banjo.bayesnet.BayesNetStructure.<init>(BayesNetStructure.java:81)
    at
edu.duke.cs.banjo.learner.SearcherSimAnneal.setupSearch(SearcherSimAnneal.java:1327)
    at edu.duke.cs.banjo.learner.SearcherSimAnneal.<init>(SearcherSimAnneal.java:722)
    at edu.duke.cs.banjo.application.Banjo.runSearch(Banjo.java:110)
    at edu.duke.cs.banjo.application.Banjo.<init>(Banjo.java:60)
    at edu.duke.cs.banjo.application.Banjo.main(Banjo.java:320)
```

Finally, a closer, in-depth examination of the faulty code section would reveal that it is missing the capability to handle the new compact parent set representations.

Running Out of Memory

Let’s say you have installed Banjo, made sure that it runs on the supplied examples, and have just set up your very first own project, with a large number of variables, and the default settings for your Java Virtual Machine (JVM). Banjo starts executing, but returns after a few seconds, with a message that it ran out of memory. This section explains a few easy steps to get you past these hurdles.

Parameters Affecting Memory Use

The internal storage requirements for Banjo can be considerable, and certain choices of settings might cause the program to request more memory from the system than is available to Banjo. The main parameters that influence memory use are:

1. The number of variables
2. The number of observations
3. The maximum Markov lag
4. The number of values or discrete states per variable
5. The maximum number of parents that any variable can have
6. The assigned level for the “fast cache” in the useCache setting
7. The value of the precomputeLogGamma setting
8. The amount of memory (possibly implicitly) assigned to the Java VM (this could be much smaller than the amount of memory that Banjo could reasonably use!).

The First Thing to Check

If your search runs out of memory, here is the first thing to do: check the amount of memory that is available on the computer that Banjo is running on. E.g., if your computer has 1 GB of physical memory, and there are no other applications running besides Banjo, then you could allow Banjo to use most of this memory (remember to leave some memory for the operating system’s processes). Considering that the default memory that the JVM sets aside for an application is only 64MB, this is the first parameter you want to adjust.

Here is an example on how to instruct the Java Virtual Machine (JVM) to use a much higher amount of memory than its default setting (which is only 64MB):

```
java -Xms256m -Xmx900m -jar banjo.jar
```

The “-Xms256m” flag indicates to the JVM to set aside a minimum of 256 MB of memory for the execution of Banjo, and “-Xmx900m” flag indicates to the JVM to set aside a maximum of 900 MB of memory. The default values are 2m and 64m respectively, so if you change “-Xms” to be greater than 64m, you must also specify “-Xmx” to be greater than 64m.

To help you find a useable value for the Xmx setting, Banjo 2 provides a new setting called *displayMemoryInfo* that provides a view at its memory use. When its value is set to “yes”, a memory “snapshot” is displayed at various intervals during a search, thus enabling some coarse monitoring, and a basis for tuning the memory parameters.

What if Banjo still Runs out of Memory?

What if increasing the amount of memory made available to Banjo by the JVM has not solved our problem, and we still get an out-of-memory message. Let’s examine the different parameters listed above more closely:

- Parameters 1 to 3 are intrinsic to the problem at hand, and likely not adjustable (except maybe for experimentation with the maximum Markov lag).
- Parameters 4 and 5 can be somewhat at the user’s discretion, and should be chosen carefully if the problem at hand poses large memory requirements.
- Parameters 6 and 7, on the other hand, are entirely “program-internal” settings that can be chosen at our discretion. There is a caveat, however: its values not only affect memory use, but they also have a huge influence on performance. In essence, the more memory we can allow for parameter 6 and 7, the faster the search runs. Consequently, it pays dividends to examine the used memory closely and adjust the fastCache level to the largest value that is still feasible without running out of memory.

To facilitate easier control over the main parameters that control its memory use, Banjo 2 enables direct access to the internal cache settings that are used to store computed results (mainly, already computed node scores for the Bde metric), via the *useCache* setting. Especially the so-called “fastCache” levels (see below) for this setting increase the amount of memory allocated by Banjo (and being worst for problems with large number of variables). On the other hand the fastCache also increases the overall performance of Banjo, in terms of the number of networks that Banjo can visit during a given time. Sometimes experimentation is the only way for finding the right balance between memory use and search performance.

The values for the *useCache* setting are

- “none”: This disables any use of the node score cache.
- “basic”: This enables the use of a basic hash-based cache, where the available amount of memory is used for storing the node score and identifying info (i.e., its parents configuration). Note that this cache is Java-system controlled (i.e., the VM uses whatever amount of free memory it has available)

The remaining *useCache* options take advantage of a special cache that is built around the use of arrays for parent configurations of a fixed parent count (currently implemented are 0, 1, and 2 parents):

- “fastCache0”: Caches all node scores when each node has no parents.
- “fastCache1”: Caches all node scores when each node has 1 or no parents.
- “fastCache2”: Caches all node scores when each node has 2, 1 or no parents.

Obviously, the higher the fastCache number, the larger is the memory requirement. In particular, the cubic order of the entire 2-parent set leads to huge memory requests, which on standard desktop machines can only be expected to succeed when the variable count is relatively small (<100 or so). When examining networks with hundreds or even thousands of variables, one will likely have to choose a lower fastCache value. Note that with the compact parent set implementations in Banjo 2, the total memory requirement of a problem with 2500 variables, 2500 observations, min. Markov lag = max. Markov lag, and fastCache1 will now fit into 500 MB.

Note that the new compact observations implementation is the default in Banjo 2; the older implementation is still available, but only selectable via constants in the code. When monitoring Banjo’s memory use, one should know that the actual memory requirements during the initial setup are about 10-20% above what will be displayed at the first feedback (i.e., the memory feedback right after the initial settings are shown), because by the time Banjo displays the initial memory use, it will already have released some large temporary storage that was used for setting up the observations. Hence, when the *nBestNetworks* setting is a small number, it will be unlikely that the application will ever run out of memory during search execution. However, when *nBestNetworks* is large, and the number of variables is large, then the amount of memory necessary for accumulating the top scoring networks can still lead to an out-of-memory error.

What if Banjo Runs out of Memory well into a Search?

In addition to displaying its memory usage, Banjo 2 implements a graceful exit when it runs out of memory: even though Banjo cannot continue the execution of a search, it prints an error message, and all the results obtained so far, then writes the obtained results to the report file.

```
[Unrecoverable runtime error: out of memory]
Banjo's memory requirements during the search execution exceeded its maximum
alloted memory of 500 mb.
Although the search cannot be continued, Banjo will try to display as much
information as possible about the obtained results.
In addition, Banjo will attempt to complete as many post-processing options as
possible.

-----
- Best 84 non-equivalent Structures
-----

Network #1, score: -8835.06, first found at iteration 11992
... (listing of best networks displayed here)

-----
- Search Statistics
-----

... (search statistics displayed here)
```

```

Statistics collected in searcher 'SearcherSimAnneal':

-----
                                ERROR DETAILS
-----
- Banjo                               Bayesian Network Inference with Java Objects -
- Release 2.0                           1 Apr 2007 -
- Licensed from Duke University          -
- Copyright (c) 2005-06 by Alexander J. Hartemink -
- All rights reserved                    -
-----
- Project:                               banjo static example
- User:                                   demo
- Dataset:                               33-vars-320-observations
- Notes:                                 static bayesian network inference
-----

[ERROR: Banjo 2, 10/10/06 9:55:21 AM]
The info below has been gathered by the application:
Out of memory in (SearcherSimAnneal$LocalSearchExecuter.executeSearch)
-----
                                End of error notification
-----

```

We expect that this feature will likely only come into play when the nBestNetworks setting is set to a (very large) number, because in this case the storage area for the top scoring networks can increase substantially as the search progresses. For all other searches, the peak memory is usually encountered during setup, while loading the observations. The memory requirements during the subsequent search are lower by more than 10%, and only fluctuate slightly.

Crash with “System” Error Message

This can be a nightmare problem if you are not familiar with modifying Java code, and Banjo quits on you unexpectedly: any error like this is what’s generally referred to as a software bug.

Here is a “typical” example (typical of bugs, but hopefully not typical for Banjo): Suppose the application attempted a division by zero somewhere in its code, the error message would be forwarded by the error handling mechanism in this way:

```

-----
                                ERROR DETAILS
-----
- Banjo                               Bayesian Network Inference with Java Objects -
- Release 2.0                           1 Apr 2007 -
- Licensed from Duke University          -
- Copyright (c) 2005-06 by Alexander J. Hartemink -
- All rights reserved                    -
-----
- Project:                               banjo static example
- User:                                   demo
- Dataset:                               33-vars-320-observations
- Notes:                                 static bayesian network inference
-----

[ERROR: Banjo 2, 10/10/06 9:42:26 AM]
Execution has stopped due to the following exception:
'java.lang.ArithmeticException: / by zero'
-----
                                End of error notification
-----

```

To get additional information about the location of the error, we can set the optional Banjo 2 setting “displayDebugInfo” to the value “stacktrace”:

```
[Banjo 2] Execution has stopped due to the following exception:  
'java.lang.ArithmeticException: / by zero'  
  
java.lang.ArithmeticException: / by zero  
  at edu.duke.cs.bayes.structures.BayesNetManager.<init>(BayesNetManager.java:121)  
  at edu.duke.cs.bayes.learner.SearcherSimAnneal.<init>(SearcherSimAnneal.java:338)  
  at edu.duke.cs.bayes.application.Banjo.runSearch(CommandLine.java:65)  
  at edu.duke.cs.bayes.application.Banjo.main(CommandLine.java:198)
```

Note that the message itself may be much more detailed, since most code sections in Banjo are designed to trap possible exceptions, and supply some informative feedback. This feedback will hopefully enable the developer to locate and fix the problem more easily.

Submitting an Error Report

Hopefully you will never see the above message, or any other error message due to an internal problem with the Banjo code. However, even the best coding practices and testing procedures sometimes let a “bug” slip through. If you encounter an error in the Banjo code, we ask you to report it, and we will try to fix it as soon as possible. However, to do so we will need your help – most importantly, we need as much information about the issue as possible:

1. We ask that you email us a copy of the **error message** along with a detailed description of the problem (see 2 below), your **settings file**, your **data file(s)**, and your **result file** to our contact address (Jürgen Sladeczek, email: hjs(at)cs.duke.edu). This information will (hopefully) enable us to recreate and quickly correct the bug.
2. Please include any additional information that you may know about the problem: is it repeatable? Does it only show up with certain data or settings, and not with others – after what change did it first occur? Did it appear after you just upgraded to a new version of Banjo, a new Java VM, etc? Did you run the jar file or within an IDE? What version of Java are you using?

Appendix A

File Formats

Example of a Minimal Settings File

Banjo 2 implements truly optional settings, in the sense that settings that are not used, or that can default to internally specified values, don't need to be part of the settings file anymore. This leads to very compact setting files. The following is a sample settings file that contains only the minimal set of settings that need to be specified for executing Banjo 2.

```
###-----
### Input parameter settings file for
###
###      BA      Bayesian
###      N      Network Inference
###      J      with Java
###      O      Objects
###
### Banjo is licensed from Duke University.
### Copyright (c) 2005-06 by Alexander J. Hartemink.
### All rights reserved.
###
### Settings file consistent with version 2.0
###-----

###-----
### Search component specifications
###-----

searcherChoice =                               SimAnneal

###-----
### Input and output locations
###-----

inputDirectory =                               data/static/input
observationsFile =                             static.data.txt
outputDirectory =                             data/static/output
reportFile =                                   static.report.@timestamp@.txt

###-----
### We require this only to validate the input
###-----

variableCount =                               33

###-----
### Network structure properties
###-----

minMarkovLag =                                0
maxMarkovLag =                                0
equivalentSampleSize =                         1.0

###-----
### Stopping criteria
###-----

maxTime =                                     10 h
```

After removing all comments, the minimal settings file becomes:

```
searcherChoice =                               SimAnneal

inputDirectory =                               data/static/input
observationsFile =                             static.data.txt
outputDirectory =                             data/static/output
reportFile =                                   static.report.txt

variableCount =                               33

minMarkovLag =                                0
maxMarkovLag =                                0
equivalentSampleSize =                        1.0

maxTime =                                     10 h
```

Example of a Comprehensive Settings File

The following setting file lists all available settings for Banjo 2. Note that when settings don't apply, they are simply ignored.

```
###-----
### Input parameter settings file for
###
###      BA      Bayesian
###      N      Network Inference
###      J      with Java
###      O      Objects
###
### Banjo is licensed from Duke University.
### Copyright (c) 2005-06 by Alexander J. Hartemink.
### All rights reserved.
###
### Settings file consistent with version 2.0
###-----

###-----
### Project information
###-----

project =                                     banjo static example
user =                                       demo
dataset =                                   33-vars-320-observations
notes =                                     static bayesian network inference

###-----
### Search component specifications
###-----

searcherChoice =                             SimAnneal
proposerChoice =                             RandomLocalMove
evaluatorChoice =                             default
deciderChoice =                               default

###-----
### Input and output locations
###-----

inputDirectory =                             data/static/input
observationsFile =                             static.data.txt
outputDirectory =                             data/static/output
reportFile =                                   static.report.txt

###-----
### We require this only to validate the input
###-----

variableCount =                               33
```

```

variablesAreInRows =
observationCount =

###-----
### Pre-processing options
###-----

discretizationPolicy = none
discretizationExceptions =
createdDiscretizationReport = withMappedValues

###-----
### Network structure properties
###-----

minMarkovLag = 0
maxMarkovLag = 0
dbnMandatoryIdentityLags = 0
equivalentSampleSize = 1.0
maxParentCount = 5
defaultMaxParentCount = 8

###-----
### Network structure properties, optional
###-----

initialStructureFile =
mustBePresentEdgesFile = static.mandatory.str
mustNotBePresentEdgesFile =

###-----
### Stopping criteria
###-----

maxTime = 1:00
maxProposedNetworks =
maxRestarts =
minNetworksBeforeChecking = 1000

###-----
### Search monitoring properties
###-----

nBestNetworks = 5
nbestNetworksAre = nonIdenticalThenPruned
numberOfIntermediateProgressReports = 10
writeToFileInterval = 0

###-----
### Parameters used by specific search methods
###-----

### For simulated annealing:
initialTemperature = 1000
coolingFactor = 0.9
reannealingTemperature = 500
maxAcceptedNetworksBeforeCooling = 1000
maxProposedNetworksBeforeCooling = 10000
minAcceptedNetworksBeforeReannealing = 200

### For greedy:
minProposedNetworksAfterHighScore = 1000
minProposedNetworksBeforeRestart = 3000
maxProposedNetworksBeforeRestart = 5000
restartWithRandomNetwork = yes
maxParentCountForRestart = 3

###-----
### Command line user interface options
###-----

askToVerifySettings = no

###-----
### Post-processing options
###-----

```

```

createDotOutput =                yes
computeInfluenceScores =        yes
computeConsensusGraph =         yes
createConsensusGraphAsHtml =    yes
dotGraphicsFormat =              jpg
dotFileExtension =              txt
htmlFileExtension =             html
fullPathToDotExecutable = C:/Program Files/ATT/Graphviz/bin/dot.exe
variableNames = (whitespace-delimited list of labels for the variables)
fileNameForTopGraph =           top.graph.@timestamp@
fileNameForConsensusGraph =     consensus.graph.@TS@
timeStampFormat =               yyyy.MM.dd.HH.mm.ss

###-----
### Memory management and performance options
###-----

precomputeLogGamma =            no
useCache =                      fastLevel2

###-----
### Misc. options
###-----

displayMemoryInfo =             yes
displayStructures =             yes
displayDebugInfo =              stackTrace

```

Observations File Example

The following is a sample file of observations, based on 20 variables, and values of 0, 1, 2, and 3 for each variable. Individual values are separated by tabs.

```

#
# this line describes the variable mapping:
# var 1 = X, var 2 = Y, ...
#
1 3 2 2 3 3 3 0 1 1 3 2 3 3 2 3 2 1 2 2
0 3 1 1 2 1 1 2 3 2 3 2 2 1 3 3 2 3 3 3
3 0 0 3 0 1 3 2 2 2 1 2 2 1 3 1 2 2 1 2
2 3 3 3 3 0 1 3 2 2 3 3 3 1 0 0 3 2 3 3

(... more observations omitted)

3 3 1 2 3 2 1 3 2 3 2 1 2 2 3 3 3 3 3 1
3 3 3 3 3 2 0 1 3 1 3 3 3 3 1 2 3 2 1 2

```

If a variable is not to be discretized, then it is important that its values are integers and lie in the range between 0 and `CONFIG_MAXVALUECOUNT-1`; since the latter constant is set to 5 by default, this means that you should use values between 0 and 4, inclusive, if not discretizing.

Banjo 2 relaxes the data format to arbitrary white space separation between data entries, as well as optional comments: any text after the #-symbol is being treated as a comment and ignored by the data loader.

Structure File: Static Bayesian Network

The following is a sample structure file for a static Bayesian network that illustrates the format currently being used by Banjo.

```

#
# Header info about a structure file for a static network
# (Banjo 2 format)
#
33      # number of variables in data set
0 0
1 1 7
2 1 20
3 4 5 9 17 22  # "Expert knowledge": node has several parents
4 1 26
5 1 27
6 0
7 1 5
8 1 9
9 1 4
10 1 30
11 1 9
12 1 5
13 1 5
14 1 4
15 1 17
16 1 25
17 2 1 2
18 1 30
19 1 22
20 1 8
21 1 9
22 0
23 1 17
24 1 5
25 1 20
26 1 28
27 0      # Maybe another comment for variable 27?
28 1 7
29 1 13
30 1 0
31 1 29
32 0

```

Banjo 2 supports an expanded format, allowing comments and white space to be embedded in the text. Comments are indicated by the # character; all text that follows the # character is being ignored when the structure is parsed.

The first (non-comment) line specifies the number of variables, 33. Each of the remaining lines specifies first the id of the variable (starting with 0), then its number of parents, and finally the id's of its parents. For example, "3 4 5 9 17 22" on line 5 means that variable with id = 3 has 4 parents, namely variables with id = 5, 9, 17, and 22.

Structure File: Dynamic Bayesian Network

The following is a sample structure file for a dynamic Bayesian network that illustrates the format currently being used by Banjo.

```

#
# Header info about a structure file for a static network
# (Banjo 2 format)
#
20      # number of variables in data set

0 0: 0      1: 2 0 7
1 0: 0      1: 1 1
2 0: 0      1: 3 0 1 2      # comment 1
3 0: 0      1: 2 2 3
4 0: 0      1: 2 1 4

```

```

5  0:  0          1:  2 4 5
6  0:  0          1:  1 6
7  0:  0          1:  2 3 7
8  0:  0          1:  2 3 8
9  0:  0          1:  3 5 6 9
10 0:  0          1:  3 8 9 10    # comment 2
11 0:  0          1:  2 10 11
12 0:  0          1:  1 12
13 0:  0          1:  1 13
14 0:  0          1:  1 14
15 0:  0          1:  1 15
16 0:  0          1:  1 16
17 0:  0          1:  1 17
18 0:  0          1:  1 18
19 0:  0          1:  1 19

```

Again, the first (non-comment) line specifies the number of variables (here: 20). Each of the remaining lines specifies first the id of the variable (starting with 0), then a block of data for each of the possible Markov lags of the underlying problem. In our case, the maximum Markov lag is 1, so there are 2 “blocks” of data, one for Markov lag 0 (indicated by “0:”), and one for Markov lag 1 (indicated by “1:”). Each of the blocks follows the convention already described for static Bayesian networks. For example, “2 0: 0 1: 3 0 1 2”, on line 4 means that the variable with id = 2 has 0 parents at Markov lag 0, and 3 parents at Markov lag 1, namely, variables with id = 0, 1, and 2.

Note: by examining the network representation, we see that no variable has a parent of Markov lag 0. Indeed, the underlying problem specifies the minimum Markov lag to be 1.

Results Output in XML Format

The following is a sample output file in the machine-readable XML format, as introduced in version 2.2.

```

<BanjoData>
  <BanjoXMLformatVersion>
    1.0
  </BanjoXMLformatVersion>
  <BanjoSettings>
    <askToVerifySettings>no</askToVerifySettings>
    <bestNetworksAre>nonidentical</bestNetworksAre>
    <computeConsensusGraph>no</computeConsensusGraph>
    <computeInfluenceScores>no</computeInfluenceScores>
    <coolingFactor>0.7</coolingFactor>

    ...

    <xmlReportFile>static.xml</xmlReportFile>
    <xmlSettingsToExport>all</xmlSettingsToExport>

  </BanjoSettings>

  <nBestNetworks>

    <network>

      <networkScore>
        -8456.2484
      </networkScore>
      <networkStructure>
        33
        0 2 5 25
        1 1 14

      ...
    </network>
  </nBestNetworks>
</BanjoData>

```

```
        30 1 0
        31 1 13
        32 1 13
        </networkStructure>
</network>

... (remaining n-best networks)

<network>

    <networkScore>
    -8466.0800
    </networkScore>
    <networkStructure>
    33
    0 2 8 21
    1 1 14

    ...

    30 1 8
    31 1 5
    32 1 13
    </networkStructure>
</network>

    </nBestNetworks>
</BanjoData>
```

Note that the variable count, minimum and maximum Markov lags, and the seed value for the random number sequence are always exported as part of the BanjoSettings listing.

Appendix B

Settings File: Parameter Names and Values

The handling of the setting parameters has changed substantially in version 2. Optional settings can now be omitted entirely from the settings file, or can have their values omitted. The spelling of setting names and values in the settings file is now case-independent. In addition, a number of optional “settings” that used to be controlled via internal constants can now be configured via the settings file.

Note that for any settings parameter that accepts a set of string values, and that has an associated default value, the string “default” can be specified in the settings file. The output by the Banjo application will record the actual value that was being used for program execution, prefixed by “defaulted to”.

Setting Name	Allowed Values	Explanation
Project information		
project	Any string	Provided to users for organizing their data. No restriction on user choice
dataset	Any string	Provided to users for organizing their data. No restriction on user choice
user	Any string	Provided to users for organizing their data. No restriction on user choice
notes	Any string	Provided to users for organizing their data. No restriction on user choice
Search component specifications		
searcherChoice	<ul style="list-style-type: none">• SimAnneal• Greedy• Skip• default = Skip	Valid string specifying any of the available searchers.
proposerChoice	<ul style="list-style-type: none">• RandomLocalMove• AllLocalMoves• default = RandomLocalMove	Valid string specifying any of the available proposers that is compatible with the selected searcher.
evaluatorChoice	<ul style="list-style-type: none">• BDe• default = BDe	Valid string specifying any of the available evaluators. Note that the version 2 BDe evaluator encompasses all the functionality of the original BDe (and literally all the original code is executed within a new inner class).

deciderChoice	<ul style="list-style-type: none"> • Metropolis • Greedy • default = Metropolis for SimAnneal, Greedy for Greedy search 	Valid string specifying any of the available deciders that is compatible with the selected searcher.
Input and output locations		
inputDirectory	Any valid directory path; can be specified as (valid) absolute or relative path	Specifies the directory where the input files for the current data run are located.
observationsFile	<ul style="list-style-type: none"> • Any comma-delimited list of valid file names • Optionally, can be specified using *-wildcard notation; When using wildcards, files that are prefixed with a “-”-sign will be excluded from the (set of) listed files. 	Specifies the name(s) of the file(s) that contain(s) the observations data for the underlying problem.
outputDirectory	Any valid directory path; can be specified as (valid) absolute or relative path	Specifies the directory where the output files for the current data run will be placed by Banjo.
ReportFile	Any valid file name; <ul style="list-style-type: none"> • May be specified using a time stamp token as part of the name. 	The name of the (detailed) report file.
We require this only to validate the input		
variableCount	Integer greater than 1	Specifies the number of variables of the underlying problem.
variablesAreInRows	<p>(Optional)</p> <p>If specified as “Yes”, observations are expected to be in column-oriented format. (in Banjo 1.0 each row in the observation file was a single observation, thus containing N entries, where N is the number of variables).</p> <p>Defaults to the original Banjo 1.0 (i.e., row-oriented) format, when unspecified.</p>	Specifies whether observations are in the transposed format compared to the original Banjo 1.0 format.
Pre-processing options		
discretizationPolicy	String, of the form qX, iX, or “none”, where X is a positive integer less than the max. number of values that a variable can have. “q” indicates “quantile”, and “i” indicates “interval” discretization.	Specifies the default type of discretization to be applied to all variables.
discretizationExceptions	A list of exceptions to the discretization policy, in the form “variable index”: “d”, where d is a valid discretization specification.	Specifies the discretization to be applied to a particular variable, when we want that discretization to be different from the overall discretizationPolicy.

createDiscretizationReport	<p>Report options:</p> <ul style="list-style-type: none"> • no • standard (provides an overview of the mapping) • withMappedValues (adds the actual value counts for each mapped value) • withMappedAndOriginalValues (also adds a list of the original values and their mapping - This seems only useful for a small set of values) 	Specifies whether to create a report on the applied discretization, as well as the extend of the report..
Network structure properties		
minMarkovLag	Integer greater or equal to 0	Sets the minimum Markov lag given by the underlying problem.
maxMarkovLag	Integer greater or equal to 0, and greater or equal to minMarkovLag	Sets the maximum Markov lag given by the underlying problem.
bdnMandatoryIdentityLags	Comma-separated list of integer values, each between (or equal) to minMarkovLag and maxMarkovLag	For specifying entire sets of parents with Markov lags that need to be included in all networks. This is always based on a-priori information about the underlying problem (applicable to dynamic Bayesian networks only). Note: if you use an initialStructure or mustBePresentParents file, you cannot list any parents that are derived from the bdnMandatoryIdentityLags property (else Banjo will display an error).
equivalentSampleSize	Real number greater than 0	The equivalent sample size.
maxParentCount	<p>Integer greater than 0 and less than the defaultMaxParentCount setting.</p> <p>Note: for maxParentCount>7, the preComputeLogGamma should likely be set to “no”, due to the extensive memory requirements.</p>	The maximum number of parents that a node is allowed to have. Note that any number greater than 4 or 5 probably won’t make much sense for the underlying problem. Generally, numbers greater than 7 will raise the memory requirements for Banjo substantially.
defaultMaxParentCount	Integer greater than 0. If not specified, defaults to the internal constant DEFAULT_MAXPARENTCOUNT (currently set to 5)	Convenient access to vary the default setting for the max. parent count. Note: due to the large memory requirements for larger values of the max. parent count, an internal limit of 12 is imposed (as a constant; edit at your own risk).
Network structure properties, optional		
initialStructureFile	Valid structure file	Specifies edges in initially prescribed network structure.
mustBePresentEdgesFile	Valid structure file	Specifies edges that any proposed network must contain.

mustNotBePresentEdgesFile	Valid structure file	Specifies edges that any proposed network must not contain.
Stopping criteria		
maxTime	<p>A valid “Banjo” time format:</p> <ul style="list-style-type: none"> • Integer greater than or equal to 0 for the time in seconds • Time in format days:hr:min:sec, where days, hr, min, and sec are integers greater than or equal to 0; prefixes can be omitted, so 1:00 means 1 minute. • A number with a qualifier (d=days, h=hours, m=minutes, s=seconds), e.g. 2.5 h for running a search for the duration of 2.5 hours. 	The maximum time that a search is scheduled to run.
maxProposedNetworks	<p>Integer greater than or equal to 0.</p> <p>If set to 0, the score for the initially specified network will be computed, and Banjo exits.</p>	The maximum number of search iterations that a search will run before Banjo will terminate.
maxRestarts	<p>Integer greater than or equal to 0.</p> <p>If set to 0, the score for the initially specified network will be computed, and Banjo exits.</p>	The maximum number of restarts that a search will run before Banjo will terminate.
<p>Note: At least 1 of the 3 search termination criteria (time, networks, restarts) needs to be specified. In Banjo 2.0, the search will run until the first specified stop criteria is reached.</p>		
minNetworksBeforeChecking	Integer greater than 0	The number of networks that Banjo will “propose” without checking for any of the “stop” conditions.
Search monitoring properties		
nBestNetworks	Integer greater than 0	The number of highest scoring networks to be tracked during a search.
nBestNetworksAre	<ul style="list-style-type: none"> • “nonIdentical” (by default the only choice for n-best=1) • “nonIdenticalThenPruned” • “nonEquivalent” 	<p>For “nonIdentical”, the tracked networks are only checked for identity, when they are to be added to the set of n-Best networks.</p> <p>For n-Best>1, “nonIdenticalThenPruned” compares using the identity comparison to track the highest scoring networks, then runs an equivalence check after completing the search to prune away any networks that are equivalent to</p>

		<p>others in the set. Note that generally the resulting set of n-Best non-equivalent networks after pruning may contain anywhere from 1 to n-Best members. To get exactly n-Best networks in the result set (assuming that the search runs long enough, and the problem parameters are consistent with the n-Best value), the “nonEquivalent” setting is available, which compares any potentially n-Best network for equivalence against the current set. Due to computational efforts for equivalence checking, searches using the “nonIdenticalThenPruned” setting are much faster than settings using the “nonEquivalent” setting.</p>
fileReportingInterval	<p>A valid time format</p> <p>Defaults to DEFAULT_FILEREPORTINGINTERVAL if invalid or not specified (currently set to 60 seconds)</p>	Determines the interval that Banjo executes a search between writing out intermediate reports (which only includes the n best networks found) to file.
screenReportingInterval	<p>A valid time format</p> <p>Defaults to DEFAULT_SCREENREPORTINGINTERVAL if invalid or not specified (currently set to 10 seconds)</p>	Determines the interval that Banjo executes between writing out feedback to screen.
Parameters used by specific search methods		
For simulated annealing:		
initialTemperature	Real greater than 0.	Sets the initial temperature on starting a simulated annealing search.
coolingFactor	Real greater than 0	Sets the cooling factor for a simulated annealing search.
reannealingTemperature	Real greater than 0.	Sets the reannealing temperature for “restarting” a simulated annealing search.
maxAcceptedNetworksBeforeCooling	Integer greater than 0	The maximum number of networks that Banjo will propose before adjusting the cooling factor.
maxProposedNetworksBeforeCooling	Integer greater than 0	The maximum number of search iterations that Banjo will propose before adjusting the cooling factor.
minAcceptedNetworksBeforeReannealing	Integer greater than 0	The minimum number of search iterations that Banjo will propose before reannealing.
For greedy:		
minProposedNetworksAfterHighScore	Integer greater than 0	The minimum number of search iterations that Banjo will execute

		after it has found a new high score, before a restart is initiated.
minProposedNetworksBeforeRestart	Integer greater than 0	The minimum number of search iterations that Banjo will execute after a restart, before the next restart is initiated.
maxProposedNetworksBeforeRestart	Integer greater than 0, and greater than minProposedNetworksBeforeRestart	The maximum number of search iterations that Banjo will execute after a restart, before the next restart is initiated.
restartWithRandomNetwork	“yes” or “no” (defaults to “yes”)	If set to “no”, Banjo will either use the initial structure file (if specified) or the empty network as the starting point for each restart. If set to “yes”, Banjo will compute a random network as the starting point for each restart. Note that maxParentCountForRestart limits the number of parents for each node for the restart network.
maxParentCountForRestart	Integer greater than 0 and less than the internal constant MAXPARENTCOUNTFORRESTART (currently set to 5)	A constraint on the number of parents in any network that Banjo computes as a restart network (when restartWithRandomNetwork is set to “yes”)
Command line user interface options		
askToVerifySettings	“yes” or “no”	If “yes” is chosen, Banjo will echo print the selected settings values, and pause for user confirmation before executing the search.
Post-processing options		
createDotOutput	“yes” or “no” (defaults to “no”)	When set to “yes”, uses the dot executable (if specified) to automatically create a graphics file of the top scoring and/or the consensus graph.
fullPathToDotExecutable	The valid (absolute) path to the dot executable. Example: “C:/Program Files /ATT/Graphviz/bin/dot.exe” Windows users should note the forward slashes in the path.	The location of the dot executable as part of the GraphViz package. Used by Banjo to create graphics output of the top scoring network.
dotGraphicsFormat	Supported are the <i>format</i> options for dot’s <i>-Tformat</i> as listed in the “Drawing graphs with dot” manual, Feb. 4 2002, most notably: <ul style="list-style-type: none"> • jpg • png • gif • ps Defaults to png.	For specifying the graphics format to the dot graphics program.
dotFileExtension	Customarily a 3-letter short cut for the file type (don’t	The dot commands are in simple text format, so the dot output file

	specify the period – if you do, you'll end up with 2). Defaults to "txt".	is a basic text file.
fileNameForTopGraph	Any valid file name that Banjo will use for the top graph. The file name will be appended to the output directory, after any time stamp token has been replaced with an actual time stamp.	Used as the file name, with the appropriate extension added, for the top graph's graphics and dot file output.
computeConsensusGraph	"yes" or "no" Only relevant when Banjo computes the top N best networks, N>1. (defaults to "no")	When set to "yes", Banjo computes the consensus graph from the top N highest scoring networks.
fileNameForConsensusGraph	Any valid file name that Banjo will use for the consensus graph. The file name will be appended to the output directory, after any time stamp token has been replaced with an actual time stamp.	Used as the file name, with the appropriate extension added, for the consensus graph's graphics and dot file output.
createConsensusGraphAsHtml	"yes" or "no" Only relevant when Banjo computes the top N best networks, N>1, and when the computeConsensusGraph flag is set to "yes". (defaults to "no")	When set to "yes", Banjo creates an html table of the consensus graph and the top N best networks.
htmlFileExtension	The extension of the file for the consensus graph and the N best networks. Typically "htm" or "html".	Appended to the consensus file name, when the html table is created.
computeInfluenceScores	"yes" or "no" (defaults to "no")	When set to "yes", Banjo computes the influence scores for the nodes of the top-scoring network.
variableNames	(Optional) Variable names can be listed in the settings file in whitespace-delimited format. Alternatively, the list can be prefixed with the token "commas:", after which the variable names can be listed in comma-separated format (thus allowing the use of whitespace in the names) If variablesAreInRows is NOT set to "yes", the variableNames setting can also be specified as "inFile", and the actual variable names can then be listed in the observations file instead, as the first non-comment text line, in whitespace-delimited format.	Used for specifying (optional) names for the variables. Note that the names for all the variables must be listed, in numerical order of the variable indexes.

XML-processing options (first introduced in version 2.2)		
XMLreportFile	(Optional, unless XMLinputFiles are specified) String representing a valid file name on the system that is executing Banjo.	For specifying the name of the XML output file. When left unspecified, no XML output will be written.
XMLoutputDirectory	(Optional) The valid (absolute) path to the XML-formatted input files.	For specifying the path of the XML output file(s).
XMLinputFiles	(Optional) List of XML-formatted files that contain results, e.g., from a search execution on a cluster environment. Any file with a “-“ prefix will be excluded from the processing.	Used to combine the results of all listed (XML-formatted) files into a single result file. When specified, no search will be performed. Instead Banjo will combine the n-best networks in the listed XML input files into a single set of n-best networks, and write the results into a single XML result file.
XMLinputDirectory	(Optional) The valid (absolute) path to the XML-formatted input files.	For specifying the path of the XML input file(s).
XMLsettingsToExport	(Optional) <ul style="list-style-type: none"> • Comma-delimited list of setting names • “All” for including all settings known to the current version of banjo • Setting names with a single “-“ prefix, to indicate the exclusion of any such marked setting • Defaults to “no setting being listed” 	Used to add any number of settings and their values to the XML output file. Note: the order in which the settings are listed is not important. First all settings “to be listed” and “to omitted” are collected into a set each, then the set of settings to be omitted is removed from the set of settings to be listed. Unknown settings are simply ignored, so spelling is important (no feedback is provided when invalid setting names are encountered). Extra spaces are ignored at beginning and end of settings names. Capitalization is not important (as usual with Banjo settings).
Miscellaneous options		

timeStampFormat	<p>Any valid time stamp format. Please consult the Java documentation for more details. Banjo simply takes the provided string and validates it as a time stamp, then uses the format when replacing its time stamp token(s) in various files. Defaults to the format “yyyy.MM.dd.HH.mm.ss”.</p>	<p>For specifying a time format for use in the time stamp token replacement. Banjo will replace any time stamp token with the actual time stamp obtained at the start of the search. Useful for keeping output data organized. The valid time stamp tokens are any of the following: “@timestamp@”, “@time stamp@”, “@time_stamp@”, “@TS@”, and “@ts@”, which can be inserted at any location into a file name.</p>
precomputeLogGamma	<p>“yes” or “no” (defaults to “yes”)</p> <p>Note that this option uses a lot of memory when the maxParentCount is above 7 parents.</p>	<p>The precomputeLogGamma setting is a tuning option that affects the memory requirements of Banjo.</p>
useCache	<ul style="list-style-type: none"> • None • Basic • FastLevel0 • FastLevel1 • FastLevel2 	<p>The useCache setting is the main tuning option for the runtime memory requirements of Banjo. This is especially an issue for very large networks: for several thousands of nodes, it is usually best to set the cache to “none”.</p>
cycleCheckerChoice	<ul style="list-style-type: none"> • dfs • dfsWithShmueli • dfsOrig (to be deprecated) • default = dfsWithShmueli • (Developer note: when the old “parent set as matrix” classes are used, then (only) the old bfs and dfs implementations are available; both are scheduled to be deprecated in a maintenance release) 	<p>Valid string specifying any of the available cycle checkers. Note: dfsOrig is only provided for testing, and generally slower than the other choices. For most problems, dfsWithShmueli can be expected to be the fastest, with increasing performance further into a search (due to the kept “history”).</p>
displayMemoryInfo	<p>“yes” or “no” (defaults to “yes”)</p>	<p>Adds the amount of memory currently used by Banjo to the output. Useful for fine-tuning the useCache setting.</p>
displayStructures	<p>“yes” or “no” (defaults to “no”)</p>	<p>For selecting whether to display the initial structure, the must-be-present-parents and the must-not-be-present-parents, as loaded from their respective files.</p>
displayDebugInfo	<ul style="list-style-type: none"> • no • stacktrace 	<p>Useful in case of a Banjo bug, to find the location of the offending code, to pass on to a developer.</p>
threads	<p>Integer greater than 0 and less than the internal constant Defaults to 1.</p>	<p>Useful when running Banjo on a multiprocessor machine: executes the specified number of threads in parallel.</p>

<p>fileNamePrefixForThreads</p>	<p>String that is appended to the beginning of the results file name. The tokens @threadID@ or @thID@ can be used to identify the individual threads. Defaults to the string "thread=i_", where i is the number (ID) of the thread.</p>	<p>Used to place the search results obtained by the individual threads (esp. for multi-processor hardware) into separate result files. Note that the combined n-Best set of networks is placed in the standard results file.</p>
<p>seedForStartingSearch</p>	<p>(Optional) Integer greater or equal to 0. Defaults to "no value", thus resulting in a randomly selected seed.</p>	<p>(Development settings) Used to put Banjo into a special "test" mode with (as of version 2.2) the following properties: When a valid value for seedForStartingSearch is provided, then the random number sequence used by Banjo will be seeded with the supplied seed, resulting in repeatable results. Note: the default seed value for the random number sequence is based on the system time. (Developers: refer to the BanjoRandomNumber class for details)</p>

Appendix C

References

Papers

- G.F. Cooper and E. Herskovits (1992): A Bayesian Method for the Induction of Probabilistic Networks from Data. *Machine Learning*, **9**, pp. 309- 347.
- Hartemink, A. (2001) [“Principled Computational Methods for the Validation and Discovery of Genetic Regulatory Networks.”](#) Massachusetts Institute of Technology, Ph.D. dissertation.
Chapter 3 has a detailed introduction to discretization techniques. Later chapters describe the use of simulated annealing for network inference.
- Hartemink, A., Gifford, D., Jaakkola, T., & Young, R. (2002) [“Combining Location and Expression Data for Principled Discovery of Genetic Regulatory Networks.”](#) In *Pacific Symposium on Biocomputing 2002 (PSB02)*, Altman, R., Dunker, A.K., Hunter, L., Lauderdale, K., & Klein, T., eds. World Scientific: New Jersey. pp. 437–449.
Describes the background for the concept of consensus graphs (model averaging).
- Yu, J., Smith, V., Wang, P., Hartemink, A., & Jarvis, E. (2002) [“Using Bayesian Network Inference Algorithms to Recover Molecular Genetic Regulatory Networks.”](#) International Conference on Systems Biology 2002 (ICSB02), December 2002.
Introduces the concept of influence scores.
- Yu, J., Smith, V., Wang, P., Hartemink, A., & Jarvis, E. (2004) [“Advances to Bayesian Network Inference for Generating Causal Networks from Observational Biological Data.”](#) *Bioinformatics*, **20**, December 2004. pp. 3594–3603.
- David Maxwell Chickering (2002) [“Learning Equivalence Classes of Bayesian Network Structures.”](#) *The Journal of Machine Learning*, **2**, March 2002. pp. 445-498.
- Oded Shmueli (1983) “Dynamic Cycle Detection.” *Information Processing Letters*, **17**, 8 November 1983. pp. 185-188.

Books and Manuals

- “Drawing graphs with *dot*”, Emden Gansner et al., 2002. Online document, from the AT&T Lab’s GraphViz library. Can be found, together with several other useful guides, at <http://www.research.att.com/sw/tools/graphviz/dotguide.pdf>.
- Hang T. Lau, “A numerical library in Java for scientists and engineers”, 2004, Chapman & Hall/CRC, book and CD. A numeric library in Java that we used for the computation of the Gamma function. Details at www.crcpress.com.

Appendix D

Project Background and Acknowledgements

Dear User,

Banjo owes much to the work of others. I am obviously grateful to the many who went before in developing the foundations for Bayesian networks, the BDe scoring metric, structure inference, and heuristic search algorithms. So many have contributed mightily to the field that I could not possibly name them all, but I would like to offer special thanks to Judea Pearl, Wray Buntine, Peter Sprites, Greg Cooper, Ed Herskovits, David Heckerman, Max Chickering, Christopher Meek, Nir Friedman, Daphne Koller, Kevin Murphy, and especially Tommi Jaakkola for their contributions to my own learning.

Regarding network inference code development, things probably first began when Tommi Jaakkola shared a small snippet of C code with me that scored a discrete network using the BDe metric, which I parlayed into a Matlab application for scoring and searching for Bayesian networks in the summer of 1999. After using this for a while, I wanted to make the application speedier, so I began to reimplement it carefully in C. In the fall of 2000, Tomi Silander was kind enough to share with me some C and Perl code from the B-Course web site, which I merged with my own code to produce an application that served as the basis of parts of my dissertation in the spring of 2001. Upon moving to Duke University in the fall of 2001, I shared this C code base with a graduate student, Allister Bernard, and a postdoc in Erich Jarvis's lab, Dr. Anne Smith. Together, we used this C code to do research leading to a number of papers, but since the code was neither modular nor well-documented, I started to imagine a new code base that would be significantly easier for others to use, but also more efficient than the existing C code. A Duke undergraduate, Daniel Greenblatt, made the first attempt by porting the code to C++ one semester, but after he graduated that effort was abandoned. Meanwhile, a student, Jing Yu, used my C code as the basis for her own C++ application, which she and I went on to improve in a number of interesting directions, including the development of influence scores.

All this background sets the stage for the emergence of Banjo, which was created from scratch in Java, and designed from the ground up to be modular and efficient, based on all of these prior experiences. I conceived a plan for a new Java code base, and concocted a number of schemes for improving the efficiency of network inference. I then hired Jürgen Sladeczek to produce the implementation, and he and I consulted with Allister Bernard and Jing Yu during the specification period to ensure that the design was sound. I owe a lot to Jürgen who worked hard to ensure that the code incorporated good software engineering principles related to things like interfaces and abstraction, error handling through exceptions, validation of values, proper file I/O, the use of global constants, and the like. In recent days, Joshua Robinson and I tried to break the code wherever possible, squashing a number of bugs in the process, but perhaps some remain—please do contact us if you find any and we'll do our best to resolve them in later versions. Jürgen created the initial drafts of the User and Developer Guides, and these were expanded and improved by Jürgen, Josh, and me. Josh created the website for Banjo and the non-commercial download mechanism. Finally, I'd like to thank Henry Berger in the Office of Science and Technology who is helping to coordinate commercial use license agreements. I hope Banjo serves you well: enjoy!

Regards,
Alex Hartemink

Index

Banjo	
Core components	5
Core objects	24
Executing	7
Explanation of components	23
Using Compute Cluster	9
Using in Matlab	47
Using Multiple Threads	8
Banjo Files	
banjo.jar	7
Download	6
LICENSE	7
README	6
Testing the installed files	7
Bayesian Network	
Dynamic	16
Searching for static	11
Bayesian Networks	
Finding non-equivalent	45
Cache	
Basic	77
FastLevel0	77
FastLevel1	77
FastLevel2	77
None	77
Compute cluster	9
Consensus Graph	43
Cycle checker	
default	77
dfs	77
dfsOrig	77
dfsWithShmueli	77
Cycle Checker	25
Data File	
Using variableNames	10
Data Formats	
Observations	10
Using variablesAreInRows	10
Decider	
Metropolis	70
Decider	26
default settings	70
Greedy	70
Discretization	30
Settings	13
Discretization Policy	
i (interval)	70
q (quantile)	70
Discretization Report	
no	71
standard	71
withMappedAndOriginalValues	71
withMappedValues	71
Distributed search	9
Equivalence Checker	26
Performance considerations	45
Error	
‘System’ crash	60
Cycle in mandatory edges file	56
Data of unexpected type	54
Developer-related	56
Display debug info	57
Insufficient memory	57
Invalid setting choice	53
Missing value of required setting	54
parameters affecting memory use	57
Rule violation	54
Submitting a report	61
Value out of accepted range	54
When ‘expected’	55
When ‘unexpected’	56
Error Reporting	51
Errors	
and warnings	55
During post-processing	55
Evaluator	
default setting	69
Evaluator	25
BDe	69
Experimenting with	
Cache	33
Comparing searchers	36
Cycle checking	37
Discretization	32
Equivalence checking	37
MaxParentCount	33
precomputeLogGamma	33
File Formats	
Observations file	65
Settings file	62
Getting Started	6
Graph	
dot output	41
Generating with dot	39
GraphViz library	39
Influence Scores	42

Memory		createDiscretizationReport	71
First things to check.....	57	createDotOutput	74
Parameters affecting	57	cycleCheckerChoice	77
Running out of.....	57	dataset	69
N-best networks		dbnMandatoryIdentityLags	71
nonEquivalent.....	72	deciderChoice	70
nonIdenticalThenPruned.....	72	defaultMaxParentCount.....	71
N-identical networks		discretizationExceptions	70
nonIdentical	72	discretizationPolicy	70
Observations		displayDebugInfo	77
Combining multiple files	51	displayMemoryInfo	77
File Format	65	displayStructures	77
Row vs column format	51	dotFileExtension.....	74
Using variable names.....	52	dotGraphicsFormat	74
Options		equivalentSampleSize.....	71
Accessing via code changes	50	evaluatorChoice	69
Post-processing.....	39	fileNameForConsensusGraph.....	75
Output		fileNameForTopGraph	75
Discretization report	13	fileNamePrefixForThreads	78
Explanation.....	15	fileReportingInterval	73
Explanation for dynamic network	21	fullPathToDotExecutable	74
Graph representation	16	htmlFileExtension.....	75
Graphic representation.....	22	initialStructureFile	71
Memory info.....	49	initialTemperature	73
Results for dynamic network.....	18	inputDirectory.....	70
Sample for static network.....	12	maxAcceptedNetworksBeforeCooling	73
Search results.....	14	maxMarkovLag	71
Unique file names.....	50	maxParentCount	71
Using time stamps	49	maxParentCountForRestart	74
Proposer.....	25	maxProposedNetworks	72
AllLocalMoves	69	maxProposedNetworksBeforeCooling	73
default setting	69	maxProposedNetworksBeforeRestart.....	74
RandomLocalMove	69	maxRestarts	72
Requirements.....	6	maxTime	72
Results		minAcceptedNetworksBeforeReanneal.....	73
Explanation.....	15	minMarkovLag	71
Explanation for dynamic network	21	minNetworksBeforeChecking	72
file output in XML format.....	67	minProposedNetworksAfterHighScore	73
Search		minProposedNetworksBeforeRestart	74
Setting up.....	28	mustBePresentEdgesFile	71
Tuning memory use	28	mustNotBePresentEdgesFile	72
Tuning performance	29	nBestNetworks	72
Searcher	24	nBestNetworksAre	72
default setting	69	notes.....	69
Greedy	69	observationsFile.....	70
SimAnneal	69	outputDirectory.....	70
Skip.....	69	precomputeLogGamma	77
Searching		project.....	69
Feedback.....	11	proposerChoice.....	69
Setting		reannealingTemperature	73
askToVerifySettings	74	reportFile	70
computeConsensusGraph	75	restartWithRandomNetwork.....	74
computeInfluenceScores.....	75	screenReportingInterval	73
coolingFactor.....	73	searcherChoice	69
createConsensusGraphAsHtml	75	testMode	78

threads	77	None	77
timeStampFormat	77	nonEquivalent.....	72
useCache.....	77	nonIdentical	72
user	69	nonIdenticalThenPruned.....	72
variableCount	70	q (quantile)	70
variableNames	75	RandomLocalMove	69
variablesAreInRows	70	SimAnneal	69
XMLinputDirectory	76	Skip.....	69
XMLinputFiles	76	stackTrace.....	57
XMLoutputDirectory.....	76	time format	72
XMLreportFile	76	Settings	
XMLsettingsToExport.....	76	Component options.....	27
Setting value		Discretization.....	30
AllLocalMoves	69	Display debug info	57
Basic	77	File format	62
BDe.....	69	Missing or invalid parameter	53
default for cycle checker.....	77	Names and values	69
default for evaluator	69	Using variableNames.....	10
default for proposer	69	Using variableNames.....	52
default for searcher	69	Using variablesAreInRows.....	10
defaults for decider	70	Using XMLinputDirectory	10
dfs	77	using XMLinputFiles.....	10
dfsOrig.....	77	Using XMLoutputDirectory	9
dfsWithShmueli	77	Using XMLreportFile	9
FastLevel0	77	Using XMLsettingsToExport	9
FastLevel1	77	Structure File	
FastLevel2	77	File format, dynamic.....	66
Greedy (decider).....	70	File format, static	65
Greedy (searcher)	69	Threads	8
i (interval)	70	Time Stamps	49
inFile (specifying variableNames).....	11	XML format	
infile (variableNames)	52	sample output file	67
Metropolis.....	70		