Banjo

Bayesian
Network Inference
with
Java
Objects, Version 2.2

Developer Guide

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

==================
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 MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE, NON-INFRINGING, 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
Banjo Architecture and Design 6
  Development Philosophy 6
  Reviewing the Banjo Application 6
  The Banjo Components 7
    The Searchers 7
    The Proposers 8
    The CycleCheckers 8
    The Evaluators 9
    The Deciders 9
    The Equivalence Checkers 10
  The Packages beneath the Main Components 10
    The Bayesnet Package 10
    The Observations Package 11
    The Settings Package 12
    The Utility Package 13
  Using Multiple Threads 13
Taking Full Advantage of Banjo 16
  Modifying Banjo Components 16
    The Existing Searchers 16
    Access a Banjo Searcher from Your Own Code 17
    How Does a Searcher Work? 18
    Obtaining the Results of the Search 19
    The Proposer 20
    The CycleChecker 21
    The Evaluator 21
    The Decider 22
    The Equivalence Checker 22
  Validating Inputs 23
    Preparing a Setting for Validation 23
  Exchanging Data Internally 26
  Error Handling 27
  Hidden Treasures 29
    Input Parsing 29
    Changing the Output Display Format 29
    Customizing your Report 30
Appendix A 31
  Modifying Banjo in Eclipse 31
    Setting up Eclipse 31
Appendix B 35
  Useful Tools 35
    Software 35
    Books 35
Index 36

What is New in Version 2

Note that the first part of this section is identical with the corresponding section in the Banjo user guide. However, we expand the developer details to address several issues that are of particular interest to Banjo customers that have made modifications to the code.

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 we 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. And should Banjo Version 2 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.

Development details:

- We removed the getMatrix() method from the EdgesI interface. While it is not an easy decision to change an already-published interface, we felt that it was a clear mistake (that any developer would have caught when trying to use EdgesI) to have included this method in the first place.
- We renamed the EdgesAsMatrixWithCachedStatisticsI to EdgesWithCachedStatisticsI. While this decision may cause some inconvenience to developers that have modified the "plumbing" section of Banjo, we felt that the slight pain was outweighed by the greater accuracy of the new name.
- The BayesNetManagerI interface has some minor, but important, changes to make use of the EdgesWithCachedStatisticsI interface.
- A Preprocessor class has been created, mirroring the Postprocessor class, with corresponding code stubs in the main application class.
- The entire handling of the observation data has been streamlined with the introduction of a number of new classes, including PreprocessedObservations, and a hierarchy of Discretizer and DataPreparer classes.
- The cycle checker classes have been renamed to CycleCheckerCheckThenApply and CycleCheckerApplyThanCheck, because that more accurately reflects their function. The actual cycle checking methods are implemented at the lowest level of the affected data structures, namely the implementation classes for EdgesWithCachedStatisticsI. Note the older "asMatrix" implementation will be deprecated in an upcoming maintenance release, and thus has not been retrofitted with the latest cycle checking algorithms.

  If you have made modifications to the Banjo 1.0 code, and have been adversely affected by our changes to the interfaces above, or by any of the other refactoring that the code went through, we would very much like to hear from you. Please send us an email about the extent of the changes to get your code to work again with Banjo 2.

Finally, and obviously, Banjo Version 2 contains the corrections for the (four) bugs that lead to the Version 1.0.x maintenance releases.

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.

As an aide to testing and error tracking, we added a new (optional) setting called s*eedForStartingSearch*, 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 includes 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 Developer Guide** provides additional details on the internal structure of the application. Its target audience is users who want to modify or extend Banjo's functionality.

- The *Banjo Architecture and Design* section describes the core components of Banjo from a developer's point of view.
- The *Taking Full Advantage of Banjo* section explains how Banjo puts features such as input validation and error handling at the developer's fingertips.

Banjo Architecture and Design

Development Philosophy

Efficiency: code designed with performance in mind, and continually profiled and optimized to provide maximum performance.

Clarity: code clearly written and well commented to ensure maximum understandability.

Maintainability: code designed in a modular framework to improve maintainability, as well as provide simple but powerful extensibility and code reuse.

Trade-offs: code designed to be as extensible as possible, only sacrificing complete modularity in a few areas to increase performance.

Reviewing the Banjo Application

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

As described in the **Banjo User Guide**, the highest-level Banjo objects are the Searchers. Within a Searcher, the CycleChecker, Proposer, Evaluator, and Decider objects are used for modeling the search process. Since these objects form the core of the Banjo framework, we frequently refer to them as the “core objects” or “core components”.

All core objects have a similar inheritance hierarchy, namely an interface (e.g., SearcherI) that describes the expected functionality from that core object, an abstract base class (e.g., Searcher) that is used for data and methods to be shared by its subclasses, and the actual implementation subclasses (e.g., SearcherGreedy or SearcherSimAnneal).

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.

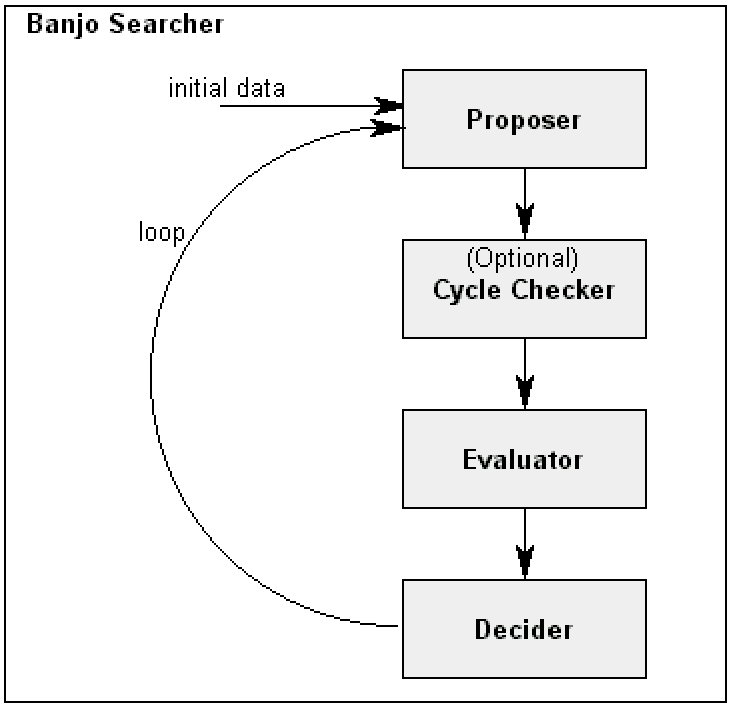
4. Decide, possibly stochastically, whether to accept the proposed network (as the new current network).

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.

Note: 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)

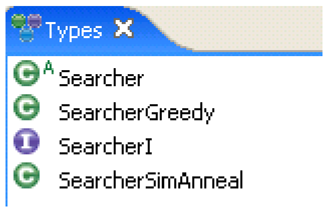
The Searchers

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 Searcher class structure is shown below.

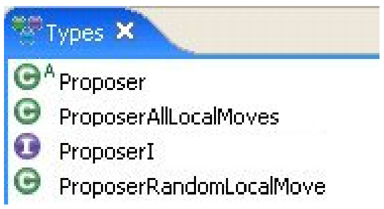

Types
Searcher
SearcherGreedy
SearcherI
SearcherSimAnneal

Banjo 2 adds the SearcherSkip searcher class, which simply initializes the necessary core objects, but skips any search iterations and proceeds straight to the post-processing section.

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.

The Proposer class structure is shown below.


Types
Proposer
ProposerAllLocalMoves
ProposerI
ProposerRandomLocalMove

The CycleCheckers

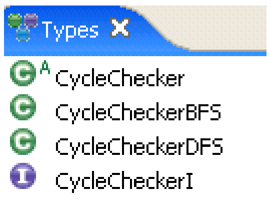
The task of the cycle checker is to examine whether each proposed network contains a cycle. Trivially, if it does, then the proposed change is discarded, and the search goes back to the Proposer to request another possible network change.

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

The options for cycle checking include depth first search (DFS) and breadth first search (BFS) strategies. The structure of the underlying problem will affect the performance of both algorithms. However, in our experience, DFS-based cycle checking seems to exhibit better performance, which is why it is the default choice for the parameter cycleCheckerChoice.

Note that the cycle checker choice is independent of the choices of the other core objects.

The CycleChecker class structure is shown below.



CycleChecker
CycleCheckerBFS
CycleCheckerDFS
CycleCheckerI

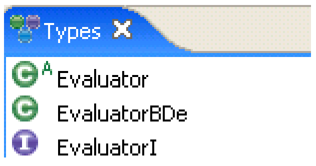
The Evaluators

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 Evaluator class structure is shown below.

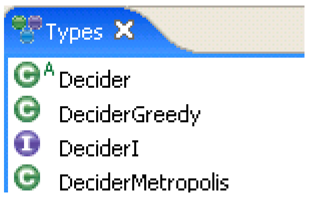


Evaluator
EvaluatorBDe
EvaluatorI

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 Decider class structure is shown below.

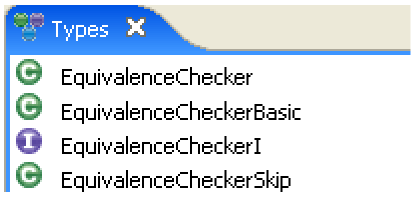


Decider
DeciderGreedy
DeciderI
DeciderMetropolis

The Equivalence Checkers

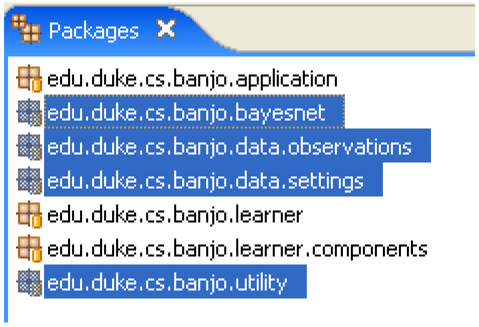
The equivalence checker class, introduced in Banjo 2, controls the equivalence property of the top-scoring networks. For collecting a set of networks that consists of non-equivalent networks at any time during the search, the EquivalenceCheckerBasic is used. Note that checking for equivalence is a time-consuming undertaking, which is the reason for the implementation of the EquivalenceCheckerSkip class, which allows us to perform identity checks only while conducting the search, then prune away any equivalent networks from the obtained set after the search is completed.

The equivalence checker class structure is shown below.


Types
EquivalenceChecker
EquivalenceCheckerBasic
EquivalenceCheckerI
EquivalenceCheckerSkip

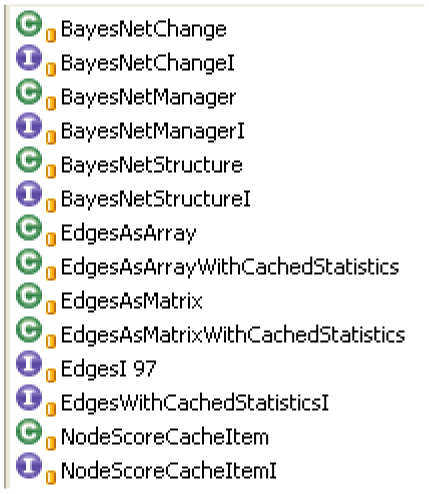
The Packages beneath the Main Components

We expect that most users that actually venture into modifying the Banjo code will want to add new search functionality in form of variations on the proposer or searcher components, etc, and thus will have little need to dig beneath the component layer into Banjo's "plumbing". However, due to the extensive changes to the "plumbing" in Version 2 we've added a brief section to review some of the code that does the work for us beneath the main component layer. This code is located in the four packages bayesnet, data.observations, data.settings, and utility:


Packages
edu.duke.cs.banjo.application
edu.duke.cs.banjo.bayesnet
edu.duke.cs.banjo.data.observations
edu.duke.cs.banjo.data.settings
edu.duke.cs.banjo.learner
edu.duke.cs.banjo.learner.components
edu.duke.cs.banjo.utility

The Bayesnet Package

The main classes in the bayesnet package are the BayesNetManager class, and the entire "Edges" class hierarchy.


BayesNetChange
BayesNetChangeI
BayesNetManager
BayesNetManagerI
BayesNetStructure
BayesNetStructureI
EdgesAsArray
EdgesAsArrayWithCachedStatistics
EdgesAsMatrix
EdgesAsMatrixWithCachedStatistics
EdgesI 97
EdgesWithCachedStatisticsI
NodeScoreCacheItem
NodeScoreCacheItemI

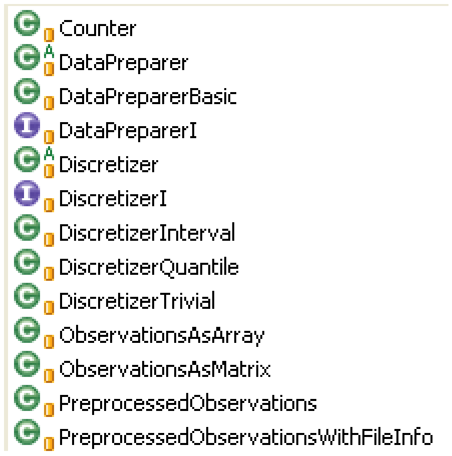
The BayesNetManager tracks the shape of the network, in all details, as a searcher moves around the search space. In particular, it is the BayesNetManager that "knows" about the edges that must or must not be present, the edges that can be added or deleted, and the current set of edges, at any given point during the iteration. Any change in the network during an iteration step is passed along between different (component) classes using the BayesNetChange. The BayesNetStructure class is used for tracking the top scoring networks, so it knows how to compare two high scoring networks.

The hierarchy built around the EdgesI and EdgesWithCachedStatisticsI interfaces contains the core data structures for actually storing a network. The "Edges" classes provide the storage structures and the basic methods for manipulating these structures, basic (identity) comparisons, and obtaining the list of parents for a node. The "EdgesWithCachedStatistics" subclasses add several derived properties to the basic edges structures, handle loading from file, and handle checking for cycles with a structure.

In Banjo 1.0 we used the EdgesAsMatrix and EdgesAsMatrixWithCachedStatistics classes for a basic network implementation based on multi-dimensional arrays. Due to the way Java implements such arrays it turned out that Banjo required much more memory for storing some structures than expected. This issue has been addressed in Banjo 2 with the addition of the EdgesAsArray and EdgesAsArrayWithCachedStatistics classes, which now map the multi-dimensional network structure into a single dimensional array.

The Observations Package

The classes in the observations package are dedicated to the loading and processing of the data set on which Banjo executes a search. In Banjo 1.0, all of this functionality had aggregated over time in the Observations class, making it a prime candidate to be refactored into a cleaner way of handling the preprocessing and discretization of the data.


Counter
DataPreparer
DataPreparerBasic
DataPreparerI
Discretizer
DiscretizerI
DiscretizerInterval
DiscretizerQuantile
DiscretizerTrivial
ObservationsAsArray
ObservationsAsMatrix
PreprocessedObservations
PreprocessedObservationsWithFileInfo

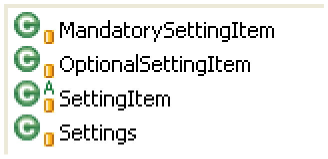
In Banjo 2 the ObservationsAsMatrix class has replaced the Observations class, using the same internal data structures for storing the actual observations in a multi-dimensional array. Due to the Java memory inefficiencies of this type of structure, we created a more compact class ObservationsAsArray, which maps the multi-dimensional array into a single dimensional one. In addition, both new Observations classes have the discretization and data preparation functionality pulled out into a separate hierarchy of its own.

The workflow of the observations data in Banjo 2 is as follows:

$$\begin{tikzcd}
\text{Raw data (strings)} \arrow[r, "\textit{tokenize}"] & \text{Symbols} \\
 \arrow[r, "\textit{validate}"] & \text{Valid symbols} \\
 \arrow[r, "\textit{discretize}"] & \text{Discrete integers} \\
 \arrow[r, "\textit{create subset}"] & \text{Subset of discrete integers} \\
 \arrow[r, "\textit{map}"] & \text{higher-dimensional representation (for use in evaluator)}
\end{tikzcd}$$

The Settings Package

The settings package contains the main settings class, which holds the data structures for storing the user input (i.e., settings), various (non-critical) error and warning collections, and containers for the data exchange between different Banjo components.

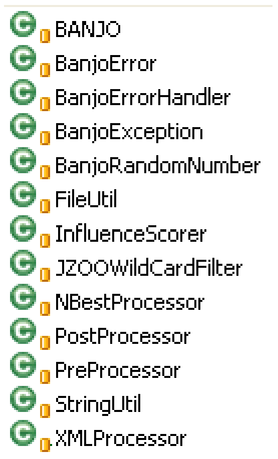

MandatorySettingItem
OptionalSettingItem
SettingItem
Settings

The SettingsItem hierarchy handles the nuances of user input validation, separated into optional and mandatory settings.

The Utility Package

The classes in the utility package consist of the error handling, the pre- and post-processing, and the string and file related routines. The one special class that stands out is the BANJO class, which is our collection of “global” constants. This includes internal settings and default values.

In version 2.2 we moved the wildcard-processing code into a separate main class (JZOOWildCardFilter), and added special classes for processing the n-best networks and XML formatted text. Finally, the small BanjoRandomNumber class works in conjunction with the *testMode* setting: it enables us to easily toggle the seeds for our random number generation.


BANJO
BanjoError
BanjoErrorHandler
BanjoException
BanjoRandomNumber
FileUtil
InfluenceScorer
JZOOWildCardFilter
NBestProcessor
PostProcessor
PreProcessor
StringUtil
XMLProcessor

Using Multiple Threads

In Banjo 2.1, the main *Banjo* class was modified to serve as a controller for launching multiple search threads, as specified by the (optional) *threads* user setting. For the user there is little difference in using Banjo compared to previous versions, except that when multiple threads are being used the output does look different, and there are multiple result files.

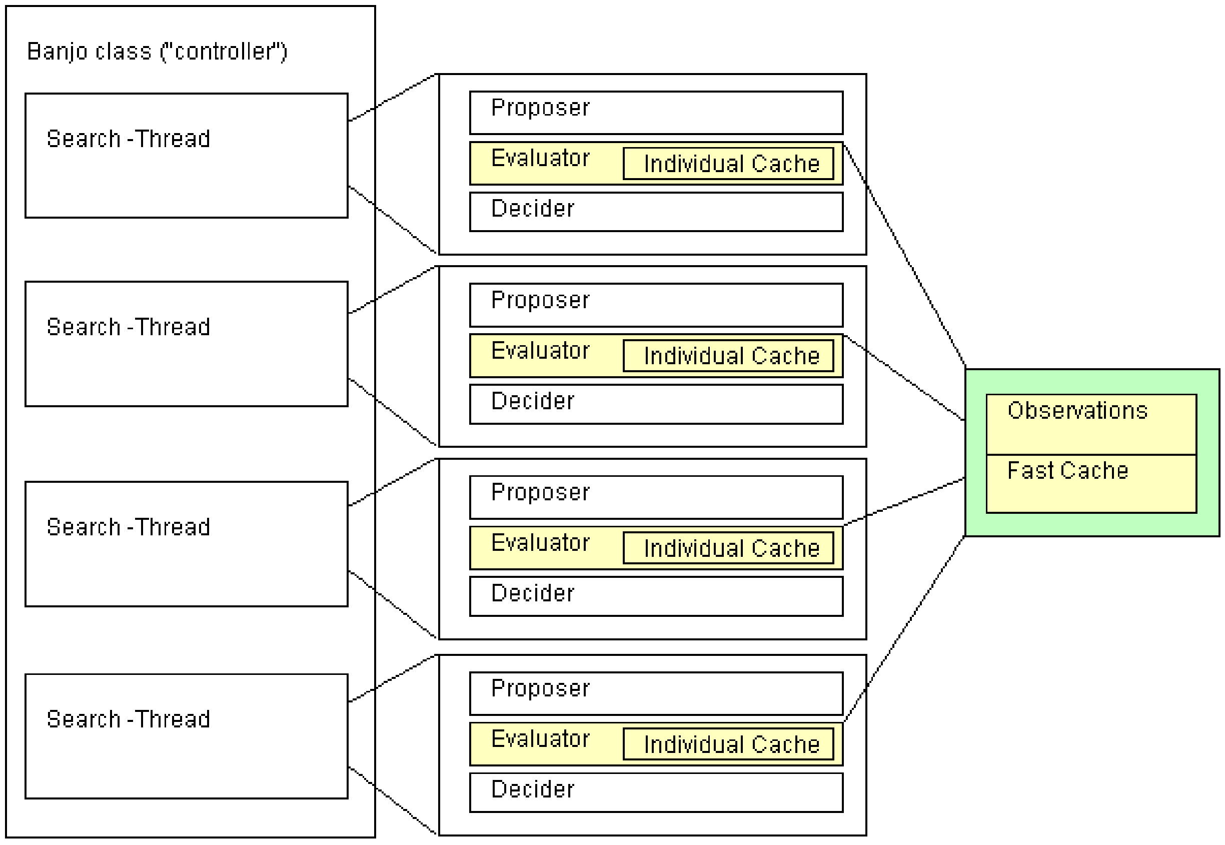
From a developer’s point, however, there are a few necessary changes that possibly impact existing code:

First we had to make the code thread safe. In several places where the threads need to write to shared data we had to wrap code segments in “synchronized” wrappers to limit access to a single thread at a time. It turns out that there are only a small number of these places, so we didn't incur a large performance penalty.

We then examined the algorithms closely and realized that we should share some additional data between the threads: the “fastCache”, the pre-computed log-Gamma values, and the observations. The memory savings due to using a single copy for each of these can be substantial for large problems. Best of all, since each individual value within each of them is only computed once, we don’t have a read-write locking issues.

For the pre-computing of the log-Gamma values and the loading of the observations we changed the order of their code execution. For example, we now load the observations before we even set up our threads.

Here is a simplified figure illustrating how the data structures relate in the new, multi-threaded design:



Currently we create multiple search threads by setting up separate Searcher instances, which in turn own their individual set of core components. It turns out that the shared data structures are all “located” within the Evaluator class – its instances share the observations and the cache (as well as the pre-computed log-gamma values).

Note that in version 2.1 all searches share the same settings as specified in the settings file. However, it would not be very difficult to use different parameters for different threads, by adding code to the section where the threads are being set up.

The results of each search are reported in individual results files, which include the collected statistics. The main Banjo class then combines the n-best networks from each of the individual searches into a single n-best results set. This set is written out to a separate results file.

Finally, on a basic coding level, the FileUtil class had to be modified so that it could be used easily for writing to multiple result files. Starting in version 2.1, FileUtil is now implemented as a “regular” instead of a static class. We tried to minimize the impact of changing FileUtil on the existing code by moving all access to file I/O into the Settings class, and using wrapper functions from Settings to the FileUtil class.

Taking Full Advantage of Banjo

Banjo tries to make your life easy when you create your own algorithms, whether it is a full-blown Searcher or a modification of an existing Proposer (or other core object). Two important tasks in developing software that are tedious but essential are properly handling the user's input and handling of all those situations when things go wrong. We hope that you will find Banjo's approach to both problems easy to use and flexible for your purposes.

Modifying Banjo Components

The Existing Searchers

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	Skips the search part, and proceeds to the post-processing part of the Banjo application.

The main changes for Banjo 2 to the Searcher classes are

- The introduction of inner classes for efficient checking on any combination of the three termination criteria.
- The integration of the equivalence-checker functionality.
- Handle out-of-memory errors gracefully by manually releasing memory, and saving any already obtained results.
- Validation of data related to the searcher classes is performed within the classes.

Access a Banjo Searcher from Your Own Code

Whether you want to add custom pre- and post-processing code of your own, or integrate Banjo into a GUI front-end program, there may be instances where you want to run a Banjo Search from your own code. This is accomplished with just a few lines of code:

```
        settings = new Settings( args );
        searcher = new SearcherGreedy( settings );
        searcher.executeSearch();
```

In the first line, we create a Settings object that we use to pass information to and from the Searcher as well as between other internal components. In line 2, we create the actual Searcher instance: this sets up the “subordinate” code, including the Proposer, Decider, etc., that is to be used for the search. Finally, in line 3, we call the executeSearch method that all Searchers need to implement to satisfy the interface requirements specified in SearcherI.

Because we need to catch exceptions, things are slightly more complicated, but not much more so. A complete “Hello Banjo” example is shown below:

```
import edu.duke.cs.bayes.learner.*;
import edu.duke.cs.bayes.utility.*;

public class RunBasicSearchExample {

        public static void main( String[] args ) {

                SearcherI searcher;
         PostProcessor postProcessor;
                Settings settings;
         FileUtilMT fileUtilMT;
         ObservationsI observations;
                BanjoErrorHandler errorHandler = new BanjoErrorHandler();

         try {

             // Load and validate the parameters for running the application
             settings = new Settings( args );

             // Setup the error handler so it knows about the settings
             errorHandler = new BanjoErrorHandler( settings );

             // Load the observations
             settings.loadObservations();

             // Set up the search(er) and run the search
             searcher = new SearcherGreedy( settings );
                    searcher.executeSearch();
```

```
            }
                catch ( final BanjoException e ) {

                    errorHandler.handleApplicationException( e );
                }
                catch ( final Exception e ) {

                    errorHandler.handleGeneralException( e );
                }
        }
}
```

Note:

- The argument passed to the constructor of the Settings object is simply the argument string that the user supplied when running the RunBasicSearchExample program from the command line. Recall from the user guide that any settings value specified this way overrides the corresponding settings value from the settings file.
- We trap any error that occurs within Banjo right where it occurs, then handle it at the top level as demonstrated in the code. You can easily extend the Banjo error handling mechanism by defining your own error messages in the Banjo class and by adding a code stub in the handleApplicationException() method of the BanjoErrorHandler class. We cover this in more detail in the section on error handling.
- In Banjo 2.1 we added support for executing multiple search threads. To be able to share the observations between thread instances, we load the observations via the settings class before we start the search. Note that the *RunBasicSearchExample* class does not implement the multiple threads, but represents the original single-threaded Banjo code.

How Does a Searcher Work?

Let's take a look under the lid of a Searcher. In the following code fragments, we will omit all "secondary" code, such as code for recording results. When you examine the executeSearch method in SearcherGreedy, you'll notice that it consists of a single line, namely,

```
searchExecuter.executeSearch();
```

The variable searchExecuter is a SearchExecuter object. SearchExecuter is an inner class, defined in the Searcher base class, that allows us to efficiently select the search characteristics (e.g., random local move versus all local moves) at the time when we define our search (i.e., in the various Searcher constructors). The main method of each SearchExecuter implementation is called executeSearch(). Here is an example of a local SearchExecuter, again with some code removed for the sake of illustrating the concepts:

```
protected void executeSearch() throws Exception {

       do {
              // Execute the inner search loop a specified number of times
              for (int i=0; i<innerSearchCounter; i++) {

                     // Get the next change from the proposer
                     suggestedBayesNetChange = proposer.suggestBayesNetChange(
                                                        bayesNetManager );
                     … cycleChecker.isValidChange( bayesNetManager,
                            suggestedBayesNetChange );
                     … bayesNetScore = evaluator.updateNetworkScore(
                            bayesNetManager, suggestedBayesNetChange);

                     if ( decider.decisionToKeep( bayesNetScore,
                            suggestedBayesNetChange )) {
                                highScoreSetUpdater.
                                     updateHighScoreStructureData( bayesNetScore );
                        }
                        else {  … }
                } // done with inner loop

                searchTerminator.generateFeedback();
        }
        while ( searchTerminator.checkTerminationCondition() );

        finalCleanup();
}
```

In the simplified code above we omitted the various logic constructs, such as how to handle the situations where we encounter a cycle (in which case there is no need to evaluate and decide).

The structure of all executeSearch methods is very similar in that within the main loop, the appropriate core objects spring into action. There are some methods that require a little explanation:

searchTerminator.generateFeedback() and
searchTerminator.checkTerminationCondition().

Both use a second inner class of the Searcher, named SearchTerminator, which provides user feedback based on the progress of the search and checks the termination condition of the main loop. Again, we use an inner class for flexibility, allowing us to cover conditions based on a time limit, max. number of proposed networks, or max. number of restarts, all of which are specified by the user. Since every condition would call for slightly different code, we would otherwise end up with a large number of condition statements that would be executed for every check of the loop condition. In Banjo 2 we cleaned up the search termination code so that it more elegantly handles any number and combination of supplied termination criteria.

The method

highScoreSetUpdater.updateHighScoreStructureData(bayesNetScore)

uses an inner class for tracking a set of N high-scoring networks. Trivially, for N=1 there is no need to maintain a set, and the logic for tracking the single best network is simpler than tracking a set.

The suggestedBayesNetChange object (of class BayesNetChange) is defined by three items: a current node, a parent node of the current node, and the type of edge change between the two nodes (which in the current version can be the addition, deletion, or reversal of the edge).

The bayesNetManager (of class BayesNetManager) is a structure that we use at the lower level for representing a network and other attributes of a network. For example, the BayesNetManager keeps track of the prescribed edges that a network needs to always have (or can never have). We will not go into details of the implementation of the BayesNetManager class, or any of the other classes at its level.

Obtaining the Results of the Search

The statistics that the searchers collect are at this time only available as plain text, either from the command-line output or the results file. However, the set of high-scoring networks is maintained by the searchers via a Collection and is stored in the Settings class. These results can be retrieved, and thus easily processed further, as follows:

```
BayesNetStructureI bayesNetStructure;
Iterator highScoreSetIterator =
        settings.getHighScoreStructureSet().iterator();

if ( highScoreSetIterator.hasNext() ) {

        bayesNetStructure = (BayesNetStructureI) highScoreSetIterator.next();
}
```

The Proposer

The Proposer, together with the Searcher, is likely the component that you will be most interested in modifying. When examining its interface, you will notice two similar functions: one provides a single change and one provides a list of changes.

```
        public abstract BayesNetChangeI suggestBayesNetChange(
                BayesNetManagerI bayesNetManager) throws Exception;

        public abstract List suggestBayesNetChanges(
                BayesNetManagerI bayesNetManager) throws Exception;
```

It is up to the Proposer whether to implement either one or both of the methods: in the class ProposerRandomLocalMove, we implement the single change method then include a wrapper around the single change method as an implementation of the list method. If a searcher were implemented by expecting a list of changes, the Proposer could comply (although with a slight performance penalty). In the ProposerAllLocalMoves class, however, we only implement the list of changes method, and throw an exception should a client ever request a single change, because this Proposer doesn't know how to provide just a single change. This means that the ultimate responsibility of putting a valid search together lies with the Searcher code.

Internally, a Proposer generally selects a change (or list of changes) based on whether to add, delete, or reverse an edge in the current network. However, for a dynamic Bayesian network with minimum Markov lag greater than 0, we don't need to consider reversals. Instead of checking this condition at every proposal step, we use a simple inner class construction that determines what type of edge "operation" is applicable:

```
private abstract class EdgeSelector {
        int changeTypeCount;

        /**
         * @return Returns the changeTypeCount.
         */

         public int getChangeTypeCount() {
             return changeTypeCount;
        }
}

private class OmitEdgeReversalSelector extends EdgeSelector {

        OmitEdgeReversalSelector() {

                // Special rule for dynamic Bayesnets with min Markov lag > 0:
                // Can't select reversals, so restrict the changeTypeCount range
                 changeTypeCount = BANJO.CHANGETYPE_COUNT - 1;
        }
}

private class AllowEdgeReversalSelector extends EdgeSelector {

        AllowEdgeReversalSelector() {

                // For static bayesnets, allow all changeType values
                changeTypeCount = BANJO.CHANGETYPE_COUNT;
        }
}
```

The main change for Banjo 2 in the implementation of the proposer classes is the use of the "edgeSelector" inner classes, which adapt the suggestBayesNetChange() methods to whatever parent set representation that we want to use. We need to use this approach because we are breaking the object-oriented encapsulation rules for performance reasons, and the inner classes provide a convenient mechanism for isolating each parent set implementation.

Finally, the validation code for all data that "belongs" to the proposer functionality has been updated in Banjo 2 to be within the proposer classes.

The CycleChecker

The CycleChecker component is generally called from within a Searcher to check whether a given BayesNetChange would create a cycle if applied to the current network managed by the BayesNetManager data structure.

```
        cycleChecker.isValidChange( bayesNetManager,
                                    suggestedBayesNetChange );
```

Internally, the CycleChecker classes have undergone a major overhaul for version 2. The actual checking for cycles is done in the implementation classes for the "parent sets", namely EdgesAsArrayWithCachedStatistics and EdgesAsMatrixWithCachedStatistics. Note that the new, more efficient (in terms of representing a parent set) EdgesAsArrayWithCachedStatistics also has a new, faster implementation of the cycle checking based on the paper by Shmueli. It turns out that the new cycle checking algorithms did not require that we apply a suggested bayesNetChange to the network first. In addition, we were able to streamline the access to the cycle checking functions, so that it made sense to separate two distinct cycle checker classes: one that applies a suggested bayesnetChange before the check for cycles is initiated (and which requires the change to be undone in case of a cycle), and a second one that checks for cycles first and only applies the change when no cycle is found.

While the Banjo end user selects the cycle checking method using the cycleCheckerChoice option in the settings file, the actual cycleChecker class that is required by the searcher is determined after the user choice is validated, and matched with the appropriate "check-then-apply" or "apply-check-undo" approach.

The Evaluator

The Evaluator component computes the score of the network using the BDe metric. The most important method of an Evaluator is:

```
        public abstract double updateNetworkScore(
              BayesNetManagerI currentBayesNetManager,
              BayesNetChangeI currentBayesNetChange) throws Exception;
```

It computes the score of the network represented by currentBayesNetManager given the change currentBayesNetChange.

The main changes for Banjo 2 to the EvaluatorBDe class are

- The introduction of inner classes that allow us to select between different implementations for observations.
- A more compact implementation for the fastCache arrays.
- Validation of data that belongs to the evaluator classes is now performed within those classes.

The Decider

Once the score of the network has been computed, the searcher calls its decider to determine whether to accept or reject the change.

```
public abstract boolean isChangeAccepted(final double newScore,
        BayesNetChangeI bayesNetChange);
```

For a simple Decider such as the greedy Decider, this is the only method that is necessary for the decision process. However, for a process such as simulated annealing, one or more parameters that the Decider uses in its decision making (e.g., the prevailing temperature) may change within the Searcher's code. To facilitate outside input, Deciders can be notified by the search process by calling the updateProcessData method to update its dynamic parameters.

```
public void updateProcessData(final Settings processData) {
                this.currentTemperature = Double.parseDouble(
        processData.getDynamicProcessParameter("currentTemperature"));
}
```

In general, parameters can easily be exchanged between any of the core components via the dynamicProcessParameter object in the Settings class.

The main change for Banjo 2 in the implementation of the Decider classes is the validation code for all data that "belongs" to the Decider functionality.

The Equivalence Checker

This is a class newly introduced in Banjo 2. The equivalence checker class comes into play when we track a set of N high-scoring networks. In Banjo 1.0, the high-scoring set was composed of N networks that were only compared to each other with respect to identity, thus allowing multiple equivalence networks to be part of the final result set. Banjo 2 provides the choice to obtain a set of non-equivalent networks.

The actual equivalence checking is performed within the updateHighScoreStructureData method of the highScoreSetUpdater inner class. As part of the highScoreSetUpdater we instantiate the type of equivalenceChecker that we need to employ: for N=1, i.e., when only the single best network is tracked, or, for N>1, when the user chooses the option "nBestnetworksAre=nonIdenticalThenPruned", we use the EquivalenceCheckerSkip object to skip any equivalence comparisons during the search. On the other hand, when N>1 and "nBestnetworksAre=nonEquivalent", we implement the full equivalence checking via the isEquivalent method of the EquivalenceCheckerBasic class. Note that the entire equivalence checking is wrapped in the highScoreUpdater inner class, which is called on when the decider keeps a network:

```
        if ( decider.decisionToKeep( bayesNetScore,
               suggestedBayesNetChange )) {

               highScoreSetUpdater.
                      updateHighScoreStructureData( bayesNetScore );
        }
```

Validating Inputs

The validation of user inputs has structurally changed from Banjo 1.0 to 1.1, from a single central section of code in the Settings class, to validation sections that are now located within each object's own validation method, named "validateRequiredData". The motivation for this change stems from the increasing amount of data that the application needs to validate, and the fact that it only makes sense to take full advantage of the object-oriented structure: by letting each object handle whichever validation it needs to apply to a particular data setting, we almost completely eliminate the need for logic constructs relative to different data scenarios. Only when we encounter closely related data within an objects validation method do we need to add special logic in comparing values, or checking their existence.

In a nutshell, the data now flows into Banjo as text-based settings that are initially loaded into a Property. During the settings' validation within the various objects we call a single method from the Settings class. This method in turn creates SettingItem objects, which we use to collect various pieces of information about the setting, such as the initial value as it was loaded, the validated value, Boolean values to indicate whether the setting is valid and whether a default value was used, as well as any problem that was encountered, such as a wrong data type, value out of required range, etc.

It may be useful to know that as part of the internal validation code, the SettingItem contains "Validator" inner classes for validating practically all of the data types that are used by Banjo, such as times, time stamps, various numeric formats, sets of strings, etc. This makes it convenient for new classes to validate their own data. And, if new code requires the validation of a new data type, it can be easily added to the SettingItem class to make it accessible to other classes as well.

Preparing a Setting for Validation

In Banjo 2 each setting is described by a set of constants defined in BANJO.java. Let's examine the settings parameter restartWithRandomNetwork, for which we accept the values "yes" and "no" from the user. The following declarations in BANJO.java will set up the use of this setting:

```
public static final String SETTING_RESTARTWITHRANDOMNETWORK = "restartWithRandomNetwork";
public static final String SETTING_RESTARTWITHRANDOMNETWORK_DESCR = "Restart with random network";
public static final String SETTING_RESTARTWITHRANDOMNETWORK_DISP = "Restart method:";
public static final String DATA_RESTARTWITHRANDOMNETWORK_DISP = "use random network";
public static final String DATA_RESTARTWITHINITALNETWORK_DISP = "use initial network";
public static final String UI_RESTARTWITHRANDOMNETWORK_YES = "yes";
public static final String UI_RESTARTWITHRANDOMNETWORK_NO = "no";
public static final String DEFAULT_RESTARTWITHRANDOMNETWORK = UI_RESTARTWITHRANDOMNETWORK_YES;
```

Note that the value of SETTING_RESTARTWITHRANDOMNETWORK, i.e., the name of the setting, defines the interface to the user of Banjo, via the settings file. Any setting name needs to be a string without a space (preferably composed only of letters a..z/A..Z and numbers 0..9), and without the #-symbol. The #-symbol is used by Java as a comment indicator, and all text that follows it will be ignored when the setting and its value are being loaded.

When a user wants to specify the "*restartWithRandomNetwork*" setting, then the constant SETTING_RESTARTWITHRANDOMNETWORK defines the string that needs to be listed in the settings file (or on the command line). The constant SETTING_RESTARTWITHRANDOM NETWORK_DESCR contains a more descriptive string for the setting (generally used in an error situation related to this particular setting), and the constant SETTING_ RESTARTWITHRANDOMNETWORK_DISP contains the string that we use in the feedback display for the search. In most cases, the _DESCR and _DISP are similar or even identical, but there are exceptions that made us decide to separate the two strings. The acceptable values for the setting are defined by the constants UI_RESTARTWITH RANDOMNETWORK_YES and UI_RESTARTWITHRANDOMNETWORK_NO, and the default behavior is defined via DEFAULT_RESTARTWITHRANDOMNETWORK.

```
// Validate the 'restart with random network'
settingNameCanonical = BANJO.SETTING_RESTARTWITHRANDOMNETWORK;
settingNameDescriptive = BANJO.SETTING_RESTARTWITHRANDOMNETWORK_DESCR;
settingNameForDisplay = BANJO.SETTING_RESTARTWITHRANDOMNETWORK_DISP;
settingDataType = BANJO.VALIDATION_DATATYPE_STRING;
validationType = BANJO.VALIDATIONTYPE_OPTIONAL;
validValues.clear();
validValues.add( BANJO.UI_RESTARTWITHRANDOMNETWORK_YES );
validValues.add( BANJO.UI_RESTARTWITHRANDOMNETWORK_NO );

settingItem = settings.processSetting( settingNameCanonical,
        settingNameDescriptive,
        settingNameForDisplay,
        settingDataType,
        validationType,
        validValues,
        BANJO.DEFAULT_RESTARTWITHRANDOMNETWORK );
```

Here *settings* is an object of the main Settings class that is originally created in the main Banjo class, and then passed along as a parameter to the various objects that need to perform validation (via their *validateRequiredData* method). Objects that have validation requirements include many of our core objects such as searchers and evaluators, post-processor and influence scorer, but also selected second-tier and bottom-level objects such as the BayesNetManager, Observations, DataPreparer, FileUtil, and BanjoErrorHandler.

The validation of a setting always includes setup code as described above, which uses the *processSetting* method of *settings* to create a SettingItem object based on the provided data, and a check of the indicated data type. This settingItem is tracked internally (within a container that is part of the settings class), so that we can request various data about a particular settings object at a later time.

For setting items of data type '*string*' the validation either needs to specify the set of acceptable values ("yes" and "no" for restartWithRandomNetwork), or set the validValues argument to BANJO. BANJO_FREEFORMINPUT).

Here is an example of validating the basic options for a searcher in the main application class. Note that we absolutely require a valid value for Banjo to continue, which we find by looking at the settingItem's isValidSetting() method. We throw an error to stop execution immediately if the user-supplied value is not acceptable.

```
    // Validate the searcher
    settingNameCanonical = BANJO.SETTING_SEARCHERCHOICE;
    settingNameDescriptive = BANJO.SETTING_SEARCHERCHOICE_DESCR;
    settingNameForDisplay = BANJO.SETTING_SEARCHERCHOICE_DISP;
    settingDataType = BANJO.VALIDATION_DATATYPE_STRING;
    validationType = BANJO.VALIDATIONTYPE_MANDATORY;
    settingItem = settings.processSetting( settingNameCanonical,
            settingNameDescriptive,
            settingNameForDisplay,
            settingDataType,
            validationType,
            validChoices(),
            BANJO.DATA_SETTINGNODEFAULTVALUE );

    //
    if ( !settingItem.isValidSetting() ) {

        throw new BanjoException( BANJO.ERROR_CHECKPOINTTRIGGER,
                    "(Checkpoint) Cannot continue without a valid searcher.\n" +
                    settings.compileErrorMessages().toString() );
    }
```

The valid choices are returned as a set from the validChoices() method:

```
     public Object validChoices() {

          Set validValues = new HashSet();

          // Implemented searchers
          validValues.add( BANJO.UI_SEARCHER_GREEDY );
          validValues.add( BANJO.UI_SEARCHER_SIMANNEAL );
          validValues.add( BANJO.UI_SEARCHER_SKIP );

          return validValues;
     }
```

Banjo 2 uses additional wrapper methods to provide the actual validation. Internally, we create an object that can either be a MandatorySettingItem or an OptionalSettingItem, depending on the *validationType*. While either mandatory or optional settings can have default values specified in case the user doesn't specify a value (or doesn't specify the setting altogether), we simply continue when an optional item is not specified, while noting an error when a mandatory item does not have a value.

Any critical error encountered during validation, and defined as an issue that prevents us from continuing, even with further validation, results in the throwing of an exception. Examples for such errors are invalid file or path names supplied by the user.

Non-critical errors, as far as immediate program execution is concerned, such as some numerical data being out of range, are handled by adding an item to the collectedErrors collection in the Settings class. For the least critical issue, we can instead add a warning item to the collectedWarnings collection.

Note: if we know that an error during validation may mean that another object could fail because it relies on the valid value, we want to return a Boolean flag from the respective validateRequiredData() method, and check for it in the calling code. If the flag is raised, we throw an error, which would get Banjo to display all errors and warnings encountered up to that point.

Finally, to obtain access to the settings once they have been validated we use the following method call:

```
IntMyInput = Integer.parseInt(
       processData.getValidatedProcessParameter( BANJO.USERINPUT_yourInput ) );
```

The getValidatedProcessParameter method of the settings object returns a string with the appropriate value. If we don't know whether the parameter has already been validated, we need to check whether the returned string value matches the constant BANJO.DATA_SETTINGNOTFOUND and act accordingly. If we need to check whether the setting is valid (or if it has already been validated), we use the isSettingValueValid() method by passing in the canonical constant for the setting name.

Exchanging Data Internally

In the process of implementing search processes, you will likely encounter situations where one of your components needs to know about a piece of data from another component. Instead of exposing data as public, we provide the Settings class with a simple "data exchange" mechanism, via three methods to access the dynamicProcessParameters object:

```
       public String getDynamicProcessParameter(String parameterName)

       public void setDynamicProcessParameter(String parameterName,
              String parameterValue )

       public Properties getDynamicProcessParameters()
```

The getDynamicProcessParameter method returns the current value of the provided parameter, the setDynamicProcessParameter method lets you update a parameter's value, and the getDynamicProcessParameters method provides the entire set of parameters in the form of a Properties object. Note that a conversion to String is necessary for getting and setting parameter values.

By convention, we suggest that all parameters that are to be used for exchanging data between various Banjo components, are declared in the BANJO class, using a DATA_ prefix.

```
       public static final String DATA_CURRENTTEMPERATURE = "currentTemperature";
```

Error Handling

Banjo implements this error handling approach: *Trap errors locally, handle them globally.* In any portion of your code where an error can occur, use the standard try-catch construct and throw a BanjoError in your catch clause:

```
     public BanjoException( final int exceptionType, final String customMessage )
```

In an effort to simplify the error handling, we have eliminated (in Banjo 2) many of the special purpose Banjo exceptions. Instead, we have combined many of them into several categories, indicated by the constants such as ERROR_BANJO_DEV and ERROR_BANJO_INPUT.

To assist future development efforts, we have added code that flags an error to the developer in many situations where Banjo encounters an unexpected value in the code.

The BanjoErrorHandler class is to be used at the highest level of the Banjo code, thus handling all errors in a global way. Its three core methods are:

```
  public void handleApplicationException( BanjoException e )
  public void handleGeneralException( Exception e Object _additionalInfo )
  public void handleOutOfMemoryError( OutOfMemoryError e, Object _additionalInfo )
```

The use of these methods is straightforward; they simply wrap the main code section in the application class, and thus take care of all errors that may occur:

```
        try {

           … (this is the place for the main application code)
        }
        catch (final BanjoException e) {

            errorHandler.handleApplicationException( e );
        }
         catch ( OutOfMemoryError e ) {

             errorHandler.handleOutOfMemoryError( e, settings );
         }
        catch (Exception e) {

            errorHandler.handleGeneralException( e );
        }
```

While our error handling philosophy has not changed in Banjo 2, we now distinguish between different categories of errors, and handle them accordingly. At the lowest level, we group together all problems that we encounter in validating various input values. As long as the rest of the application can continue (even if only to complete validating the current section of user input), we don't immediately throw an exception, but collect these errors, so that we can display them to the user all at once. Of course, if the validation problem is serious, we may only continue to the next checkpoint, at which time we then throw an exception (e.g., file- and path-related validation errors generally are of the more serious type).

Within the code we have placed “checkpoints”: if any errors have been encountered, we throw an exception to stop the further execution of the code. It is up to the developer to decide which errors are non-critical, and where to place a checkpoint to keep the program from continuing without the proper data.

A special type of error that we now handle very carefully is out-of-memory errors. When they occur during search execution, a special try-catch section in the main searcher classes traps these errors, so that we can display the results that so far have been obtained by the search, such as the top-scoring networks, and the search statistics. Compared to Banjo 1.0 we are now able to exit gracefully when encountering most out-of-memory errors by releasing memory that had been allocated for the core objects, but – due to the encountered error – would not be useful to keep around any longer anyway. Here is a (staged) example of an out-of-memory error recovery:

```
[Unrecoverable runtime error: out of memory]
Banjo's memory requirements during the search execution exceeded its maximum
alloted memory of 12 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
33
 0 2 9 30
…
```

Note that when using the compact observations implementation (which is the default in Banjo 2, via constants that can only be modified in the code) the actual memory requirements during the initial setup are about 10-20% above what will be displayed at the first user feedback (the one after the initial settings are shown), because we will have manually 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.

Hidden Treasures

This section describes how to unlock a few "hidden features" of the Banjo code that are waiting to be used, but have not been fully tested and thus not yet been fully incorporated into the release code. Some of these features may be supported in a later version of Banjo, but if you wish to explore them now, feel free to do so (with no guarantees that they work perfectly).

Input Parsing

In Banjo 2 most data handling has been extensively refactored to use regular expressions, which makes our handling of user input data substantially more flexible than it was before. Prime examples include the use of generic "white-space" delimiters in place of the exact "space" or "tab" delimiters.

The input data parsing is also set up to allow arbitrary blank lines and commented lines (i.e., lines that start with a # character), as well as line-ending comments, for convenient annotation of the data by the user. Note that the comment-indicator is the internal Java comment indicator character, so the comment character cannot be changed in Banjo 2.

Changing the Output Display Format

There are several options for changing the way the Banjo output is formatted by setting the values of some constants in the BANJO.java file.

Here is a basic example: Suppose you want to change the way that Banjo displays the elapsed time during a search. You can change the unit descriptors for the different time units, as well as the thresholds after which the display changes from, say "elapsed time in seconds" to "elapsed time in minutes".

```
        // String constansts used for time display
        public static final String BANJO_TIMEQUALIFIER_DAYS = "d";
        public static final String BANJO_TIMEQUALIFIER_HOURS = "h";
        public static final String BANJO_TIMEQUALIFIER_MINUTES = "m";
        public static final String BANJO_TIMEQUALIFIER_SECONDS = "s";
        public static final String BANJO_TIMEQUALIFIER_MILLISECS = "ms";


        // Limits where we change the display from milliseconds to seconds
        public static final int FEEDBACK_DEFAULT_CUTOFF_TO_SECONDS = 1;
        // .. from seconds to minutes
        public static final int FEEDBACK_DEFAULT_CUTOFF_TO_MINUTES = 1;
        // .. minutes to hours
        public static final int FEEDBACK_DEFAULT_CUTOFF_TO_HOURS = 1;
```

The second part of the above code indicates that the elapsed time is displayed in milliseconds until 30 seconds are reached, in seconds until 2 minutes have elapsed, and in minutes until 5 hours have elapsed, after which time, elapsed time is reported in hours. Note that the display is prettied up with

```
public static final int FEEDBACK_TIMEPADDINGLENGTH = 8;
```

to match the current time settings. This adjusts the amount of white space so that the time entries are properly justified when displayed in a mono-spaced font.

Customizing your Report

In Banjo 1.0 the output of various components of a Banjo search was determined via internal flags located in the global constants file BANJO.java. Practically all these constants have been replaced by settings that let the user control the output without having to make changes to the code.

The most important settings are:

- createDiscretizationReport: Determines whether the discretization report is created and what details about the discretization data are displayed.
- computeInfluenceScores: Determines whether the influence scores are computed and displayed, one for every edge in the highest-scoring network.
- createDotOutput: Determines whether dot output is created and displayed for visualizing the highest-scoring network with *dot*.
- displayStatistics: Determines whether the statistics from the search and its components are displayed in the output report.
- useCache: Sets the level of the cache used in the reuse of previously computed (node) scores.
- cycleCheckingMethod: Selects the method to be used in the cycle checking.

In addition, there are a number of new settings related to Banjo's output:

- displayMemoryInfo: This displays the amount of memory used by Banjo at various stages of execution, displayed at the designated output intervals.
- displayStructures: This adds the initial structure, the must-be-present and the must-not-be-present parent listings to the output (any not specified structures is skipped).
- displayDebugInfo: This can provide additional information in case of a problem or bug in the code, by displaying the stack trace without having to execute Banjo in a development environment.
- computeConsensusGraph: This controls the computation of the consensus graph from the set of top-scoring networks
- createConsensusGraphAsHtml: This creates a basic html table with the parents listing for the N top-scoring networks, as well as the derived consensus graph.

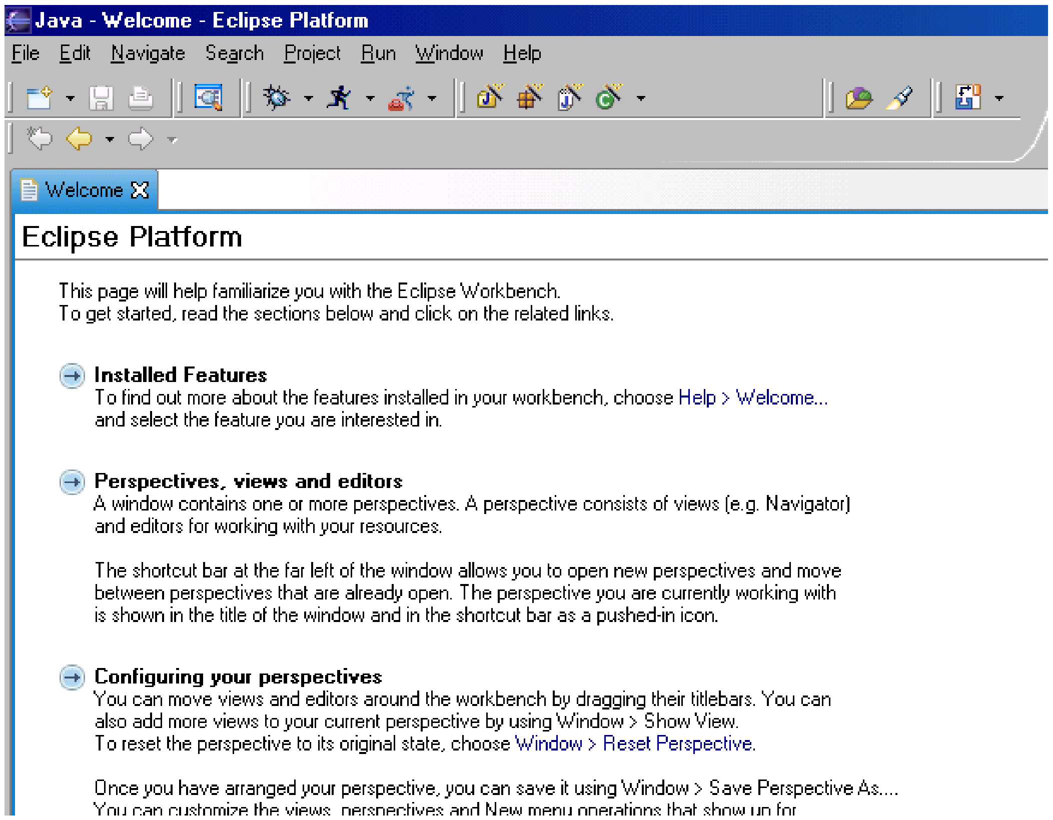
Appendix A

Modifying Banjo in Eclipse

Setting up Eclipse

The Eclipse IDE is available as a free download from http://www.eclipse.org. Make sure that you have the Java SDK 1.4x (or higher) installed on your machine, since the Banjo code is developed under Java 1.4x. The code for Banjo version 2 was developed using Eclipse 3.1. The screenshots in the brief intro to Eclipse are based on Eclipse 3.0, but have remained largely unchanged in Eclipse 3.1.

When you launch Eclipse, it looks similar to this screenshot:


Java - Welcome - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Welcome

Eclipse Platform

This page will help familiarize you with the Eclipse Workbench.
To get started, read the sections below and click on the related links.

Installed Features
To find out more about the features installed in your workbench, choose Help > Welcome...
and select the feature you are interested in.

Perspectives, views and editors
A window contains one or more perspectives. A perspective consists of views (e.g. Navigator)
and editors for working with your resources.

The shortcut bar at the far left of the window allows you to open new perspectives and move
between perspectives that are already open. The perspective you are currently working with
is shown in the title of the window and in the shortcut bar as a pushed-in icon.

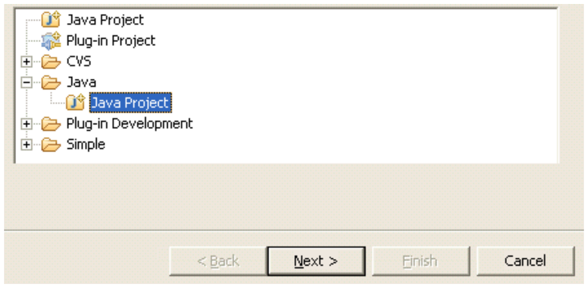
Configuring your perspectives
You can move views and editors around the workbench by dragging their titlebars. You can
also add more views to your current perspective by using Window > Show View.
To reset the perspective to its original state, choose Window > Reset Perspective.

Once you have arranged your perspective, you can save it using Window > Save Perspective As....
You can customize the views, perspectives and New menu operations that show up for

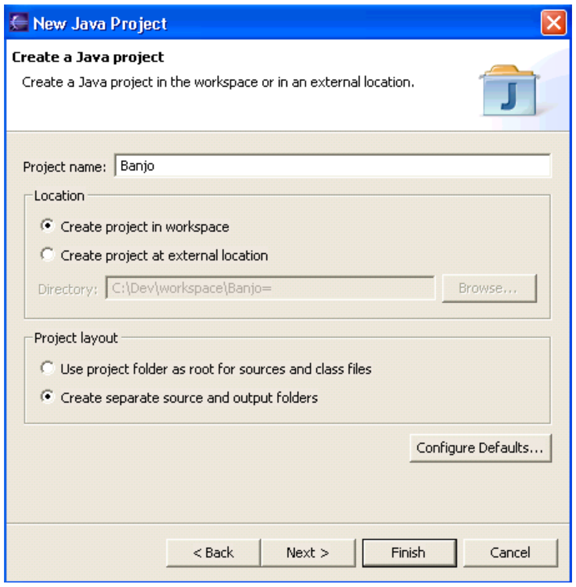
Due to the rising popularity of Eclipse, there are now quite a number of books available that describe how to use it effectively. We have listed several of our personal favorites in the Appendix.

Here is a quick way to load the Banjo code into Eclipse. Create a new project by selecting

File → New → Project. This opens the *New Project* dialog. In the dialog, expand the *Java* item, and click on *Java Project*:



Then click on *Next*. Enter "Banjo" in the *Project Name* field, and select "Create separate source and output folders". Then click on *Next* again.


New Java Project

Create a Java project

Create a Java project in the workspace or in an external location.

Project name: Banjo

Location

(●) Create project in workspace

( ) Create project at external location

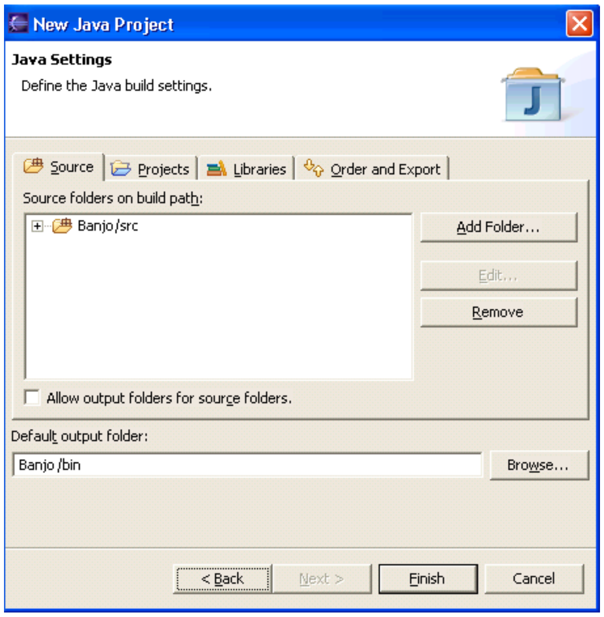
Directory: C:\Dev\workspace\Banjo=

Project layout

( ) Use project folder as root for sources and class files

(●) Create separate source and output folders

The final step of the new project wizard now displays the selected settings:


New Java Project

Java Settings

Define the Java build settings.

Source | Projects | Libraries | Order and Export

Source folders on build path:

Banjo/src

Add Folder... | Edit... | Remove

☐ Allow output folders for source folders.

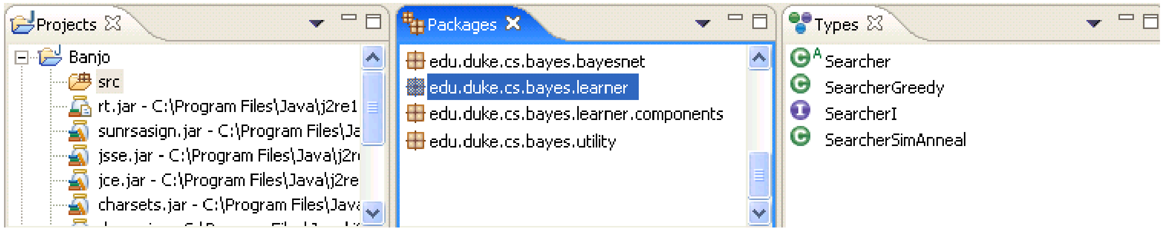
Default output folder: Banjo/bin

Browse...

< Back | Next > | Finish | Cancel

Click on *Finish* to create the “Banjo” project. This will create a Banjo folder in Eclipse’s Workspace directory (which is located where you selected it during your Eclipse installation). You will notice that a Banjo entry has been added to the Eclipse’s Project pane.

To load the Banjo source files into Eclipse, copy the `src` directory that was extracted from `banjo.zip` into the Banjo folder in Eclipse’s Workspace directory. Then return to Eclipse, and right-click on the Banjo entry in the Projects pane. Select “Refresh”. This loads the Banjo source files.


Projects | Packages | Types

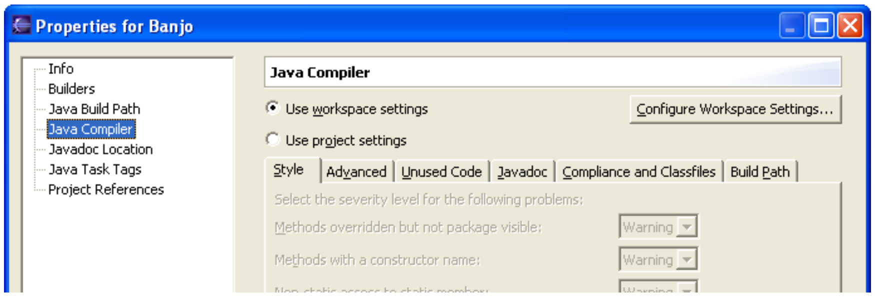
```
Banjo
  src
  rt.jar - C:\Program Files\Java\j2re1
  sunrsasign.jar - C:\Program Files\Ja
  jsse.jar - C:\Program Files\Java\j2r
  jce.jar - C:\Program Files\Java\j2re
  charsets.jar - C:\Program Files\Java
```

```
edu.duke.cs.bayes.bayesnet
edu.duke.cs.bayes.learner
edu.duke.cs.bayes.learner.components
edu.duke.cs.bayes.utility
```

```
Searcher
SearcherGreedy
SearcherI
SearcherSimAnneal
```

Feel free to explore the source files in the various panes.

Now open the Properties dialog from the Project menu, and click on “Java Compiler” in the list on the left hand side of the window:


Properties for Banjo

Info
Builders
Java Build Path
Java Compiler
Javadoc Location
Java Task Tags
Project References

Java Compiler

(●) Use workspace settings | Configure Workspace Settings...

( ) Use project settings

Style | Advanced | Unused Code | Javadoc | Compliance and Classfiles | Build Path

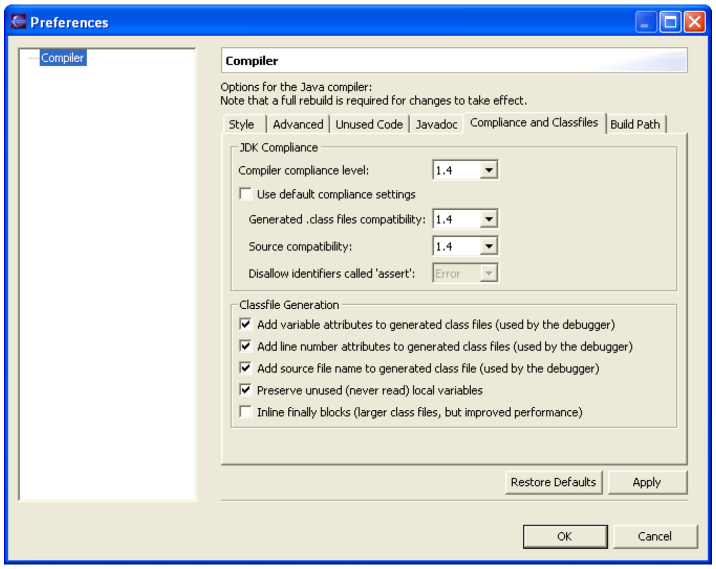
Select the severity level for the following problems:

Methods overridden but not package visible: Warning

Methods with a constructor name: Warning

Now click on the “Configure Workspace Settings…” button.

In the Preferences dialog that opens, you want to select the “Compliance and Classfiles” tab. There you need to select “1.4” as the compliance levels on all dropdowns:



Compiler

Options for the Java compiler:
Note that a full rebuild is required for changes to take effect.

JDK Compliance

Compiler compliance level: 1.4

☐ Use default compliance settings

Generated .class files compatibility: 1.4

Source compatibility: 1.4

Disallow identifiers called 'assert': Error

Classfile Generation

☑ Add variable attributes to generated class files (used by the debugger)

☑ Add line number attributes to generated class files (used by the debugger)

☑ Add source file name to generated class file (used by the debugger)

☑ Preserve unused (never read) local variables

☐ Inline finally blocks (larger class files, but improved performance)

If you forget this step, you will see a long list of errors in your problem pane in Eclipse, because Banjo relies on some Java 1.4 language features.

Finally, there is only one step left so you can run a Banjo search within Eclipse. You need to copy a settings file into the workspace’s Banjo directory (naming it `banjo.txt` unless you change the default command line arguments appropriately within Eclipse).

Note: Once you have your Banjo files set up, if you want to replace the entire directory tree with a new set of files, simply delete your existing `src` directory (maybe after you back it up to another location), and then copy the new `src` directory in its place. Then, again, right-click on the Banjo project and select “Refresh”. The same “trick” also applies if you replace some of your source files in the file system (i.e., not within the Eclipse environment): to “load” your changes into Eclipse, you need to refresh your source tree.

Appendix B

Useful Tools

Software

- The Eclipse environment. Available as a free download from http://www.eclipse.org. Simply an amazing IDE. The refactoring features alone are worth the money (it's free!).
- Jprofiler by ej-technologies (http://www.ej-technologies.com). A great tool for finding performance bottlenecks.
- The AT&T Lab's GraphViz library. Open source library for graphical visualization. Official website at http://www.graphviz.org.
- Hang T. Lau, "A numerical library in Java for scientists and engineers", 2004, Chapman & Hall/CRC, book and CD. A high-quality Java translation of the NUMAL numeric library. Details at http://www.crcpress.com.
- Kevan Stannard's JZOOWildCardFilter.java class for handling wildcards came in very handy. At http://www.jzoo.com/java/wildcardfilter.

Books

- Robert Simmons, Hardcore Java, O'Reilly, 2004.
- Bruce Eckel, Thinking in Java, 2nd ed., Prentice-Hall/PTR, 2000.
- David Flanagan, Java in a Nutshell, (various editions), O'Reilly.
- David Flanagan, Java Examples, 2nd ed., O'Reilly, 2000.
- Douglas Dunn, Java Rules, Addison-Wesley, 2002.
- Steve Holzner, Eclipse, O'Reilly, 2004.
- Doug Lea, Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley, 1997.

Index

Banjo
  Architecture and design 6
  Contacts 2
  Core components 5
Bayesnet Package 10
BayesNetManager 11
CycleChecker
  Description 8
  Inner workings.................................... 21
Data
  Exchanging internally 26
Decider
  Description 9
  Inner workings.................................... 22
Eclipse
  Configure workspace 33
  Load Banjo code 31
  Properties 33
  Setting up... 31
Equivalence Checker
  Description 10
  Inner workings 22
Error Handling 27
Evaluator
  Description... 9
  Inner workings 21
Input Validation 23
Observations Package............................ 11
  Processing of 12
Proposer
  Description... 8
  Inner workings 20
Report
  Customizing 30
Searcher
  Description... 7
  Inner workings 18
  Tasks 6
Settings Package................................... 12
Threads
  Using multiple 13
Utility Package...................................... 13