openModeller

A framework for species modeling

Fapesp process: 04/11012-0
Partial Report #2 (April 2006 – March 2007)
Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>Objectives</td>
<td>3</td>
</tr>
<tr>
<td>Activities</td>
<td>3</td>
</tr>
<tr>
<td>General framework activities/studies</td>
<td>4</td>
</tr>
<tr>
<td>Evaluation of parallelization techniques and analysis of parallel</td>
<td>4</td>
</tr>
<tr>
<td>algorithms for biodiversity data analysis</td>
<td></td>
</tr>
<tr>
<td>Initial implementation using Services Architecture</td>
<td>5</td>
</tr>
<tr>
<td>Locality data component</td>
<td>5</td>
</tr>
<tr>
<td>Study of data-cleaning techniques</td>
<td>6</td>
</tr>
<tr>
<td>Environmental data component</td>
<td>6</td>
</tr>
<tr>
<td>Pre-analysis component</td>
<td>7</td>
</tr>
<tr>
<td>Modeling component</td>
<td>8</td>
</tr>
<tr>
<td>Post-analysis component</td>
<td>9</td>
</tr>
<tr>
<td>Desktop Interface</td>
<td>9</td>
</tr>
<tr>
<td>Study Cases</td>
<td>10</td>
</tr>
<tr>
<td>Other activities</td>
<td>11</td>
</tr>
<tr>
<td>Publications and Presentations</td>
<td>12</td>
</tr>
<tr>
<td>Published or with a new status</td>
<td>12</td>
</tr>
<tr>
<td>In preparation</td>
<td>13</td>
</tr>
<tr>
<td>General Comments</td>
<td>13</td>
</tr>
</tbody>
</table>
Introduction

openModeller’s project goal is to develop a framework to facilitate the work of scientists in predictive modeling of species distribution. The four year project funded by Fapesp involves three institutions: CRIA (Centro de Referência em Informação Ambiental), Poli (Escola Politécnica da USP), and INPE (Instituto Nacional de Pesquisas Espaciais). This report summarizes the activities carried out during the second year of the project, from April 2006 to March 2007.

Objectives

This project’s original aim is to develop a flexible and robust modeling framework to predict species distribution. Main objectives are:

- develop a component-based modeling framework with reusable modules compliant with web services technology
- enable use and comparison of different modeling algorithms through the same infrastructure
- enable modeling pre-analysis and pos-analysis using specific components
- enable the existence of multiple interfaces (web, desktop, command line, web service)
- develop a modeling desktop interface fully integrated with GIS functionalities
- develop a modeling web interface
- facilitate access to distributed biological and environmental data networks
- allow usage of high performance computing with species distribution modeling.
- carry out use cases to test and validate the complete framework.

Activities

Most objectives planned for the second year of the project involved additional progress over the activities presented in the last report. During the second year, two releases of the openModeller library were made (the complete change log can be found in annex 1). The first one was a major release (0.4) including the integration with TerraLib and the complete refactoring of the Web Services interface. The other release (0.4.1) included a new algorithm called EnvironmentalDistance and improvements in the locality component. Two other algorithms (AquaMaps and AdaptGARP) were implemented but have not been released yet.

The Desktop interface was completely rewritten into a new multi-platform application that supports modeling experiments involving multiple species and multiple algorithms. When using this interface one can run models using resources from a local machine or using remote modeling servers.

A new modeling plugin for TerraView was developed, completing the bridge between TerraLib and openModeller.

A computer cluster with 11 nodes (each with 2 quad-core servers) was purchased and shall be installed by the end of April. It will provide high processing power to develop and run models.
Below is the list of objectives for the project, followed by a summary of activities and the main achievements during the second year.

General framework activities/studies

Evaluation of parallelization techniques and analysis of parallel algorithms for biodiversity data analysis

One of openModeller’s known bottlenecks is projecting the model into a distribution map, especially when the experiment involves many layers with high resolution and the region of interest has a large extent. Map projection was therefore considered a suitable target for applying parallelization techniques. During the last year, the corresponding part of openModeller code was studied in detail before starting implementation of parallelized versions. The initial implementation is being developed using OPENMP\(^1\) for multiprocessor computers with shared memory. This version is still being tested to achieve maximum performance. During this year a new version using another parallelization technique called MPI\(^2\) (Message Passing Interface) will be developed so that it can take full advantage of computer clusters. More information about this work can be found in annex 16.

The purchase of a computer cluster, which was supposed to be completed by December 2006, was delayed due to bureaucratic problems with importation. The cluster is expected to be installed by the end of April at Escola Politécnica da Universidade de São Paulo. Its configuration was updated to incorporate a new quad-core processor which was recently launched. The final configuration of the SGI Altix xE 1300 machine totalizes 11 nodes, or 88 Cores of Intel Xeon 5335, with 2.0GHz/8MB cache, 1333MHz FSB and 88 GB RAM DDR 667MHz.

After analyzing the information about tools for cluster management, two of them were identified as potential candidates to be used: Condor\(^3\) and OpenPBS\(^4\). A set of virtual machines was set up to test them. So far Condor seems to meet all requirements, including access control, job checkpoints, remote system calls, job schedulers and additional scalability through flocking. A web portal will be implemented to facilitate job submissions to the cluster.

A study about parallelization techniques has been carried out, and after analyzing the existing implementation of the GARP algorithm in the openModeller framework, a new algorithm called P-GARP was developed (still not available in the public code repository). This algorithm was designed based on the Coarse Grained Model (CGM) technique. It provides a parallel execution of the GARP iterate method, combining the results of each iteration to generate a model. It also uses synchronization barrier and communication round concepts, so that it can be used in a cluster environment. Both algorithms, GARP and P-GARP, were tested in a simulated cluster environment (see annex 11 for results). The MPI standard was also studied and tested with lam-mpi version 7.1.2 for Linux. All requirements to run P-GARP with MPI have been defined. Next steps include the implementation of P-GARP using MPI and then testing it in a

\(^1\) [http://www.openmp.org](http://www.openmp.org)

\(^2\) [http://www.netlib.org/mpi/](http://www.netlib.org/mpi/)

\(^3\) [http://www.cs.wisc.edu/condor/](http://www.cs.wisc.edu/condor/)

\(^4\) [http://www.openpbs.org/](http://www.openpbs.org/)
real cluster. Other algorithms available in openModeller will also be evaluated to see if similar parallelization techniques can be applied.

**Initial implementation using Services Architecture**

Considering that openModeller may require intensive use of computational resources, it is very important to optimize performance for each component and make all necessary adjustments in the architecture. A detailed study was conducted for each individual class using the typical execution flow. The framework was divided into several components and hosted in a distributed architecture that implements inter-process communication between the main process (the engine) and all components. To find bottlenecks, a performance evaluation methodology was devised (see annex 4). The existing code was instrumented using AOP (Aspect Oriented Programming) techniques to allow profiling the code during execution. This approach, together with asynchronous messaging programming, gathered data about the time elapsed in each component and even in each method call. This information was stored into a database and is accessible through a web interface. This work resulted in a performance simulation model that can be used to study the efficiency of different possible architectures for openModeller and to identify parts of the code that could benefit from parallelization. The prototype and infrastructure that were built are also accessible as a remote service that can process openModeller requests and collect information about performance.

A technical report is being prepared to present a detailed specification of a new architecture for openModeller considering the domain definition and specific requirements for the framework. This work was based on the definition of the modeling process which was already mentioned in the last report. A preliminary version is being evaluated by developers and users.

The latest version of openModeller includes a full server implementation for the new Web Services API based on SOAP (Simple Object Access Protocol). This API was developed in a partnership with the BiodiversityWorld project\(^5\). It enables the use of remote modeling servers that can be set up on top of a cluster. A formal definition of the Web Services API is available\(^6\), including the following operations: ping, getAlgorithms, getLayers, createModel, getModel, projectModel, getProgress, getLog, getLayerAsAttachment, getLayerAsUrl and getProjectionMetadata. This API uses the Document/Literal SOAP style and references external elements and types from an XML Schema\(^7\) that also validates openModeller serialized objects.

**Locality data component**

The last release of openModeller includes a few changes in locality data structures. Each locality provided to openModeller must now contain a unique identifier so that serialized models and system messages can explicitly reference back to it.

All necessary adjustments were made so that absence data can be provided to the library and used by the algorithms. The text file format for the corresponding occurrences reader (used mainly by the console interface) was changed accordingly. It now expects 4 mandatory columns (unique identifier, label, longitude and latitude) and one optional column for abundance. All values must now be separated by TABs.

\(^5\) [http://www.bdworld.org/](http://www.bdworld.org/)

\(^6\) [http://openmodeller.cria.org.br/ws/1.0/openModeller.wsd](http://openmodeller.cria.org.br/ws/1.0/openModeller.wsdl)

\(^7\) [http://openmodeller.cria.org.br/xml/1.0/openModeller.xsd](http://openmodeller.cria.org.br/xml/1.0/openModeller.xsd)
Absence points are indicated when “abundance” is equal to zero. When abundance is not specified, “1” is assumed (presence data).

New drivers were developed to retrieve locality data from the GBIF\(^8\) and speciesLink\(^9\) networks as part of the Desktop interface. Since there was no official Web Services API on top of GBIF’s database, the GBIF driver actually interacts with the search interface of the existing GBIF web portal. During the next year, this driver will be updated to make use of one of the Web Services APIs that will become available from GBIF (either based on WFS\(^{10}\) or based on the new custom GBIF REST-style API). The speciesLink driver is based on a custom HTTP/XML protocol. This network is developing a harvesting system that will feed a centralized database from where occurrence point data will be provided in a much faster way to openModeller. Both drivers should soon move to the openModeller locality component.

The driver to access a TerraLib database containing locality data was improved and now allows access to TerraLib databases built on top of other Database Management Systems like Oracle, Oracle Spatial, PostgreSQL, PostGIS and SQLServer, besides the initial MySQL server.

Broadening the data structure for locality data to make use of data cleaning tools, implementing a Web Services API for data cleaning, effectively making this component interact with a data cleaning service, and defining a final API for this component are some of the next features to be developed.

**Study of data-cleaning techniques**

All data cleaning tests mentioned in the previous report were implemented as part of the DataTester framework. Namely: geographic error detection, geographic outlier detection, itinerary error detection and elevation error detection. The following tests were additionally implemented:

- **Environmental outlier detection.** Detects outliers on environmental values using a statistical method based on reverse-jackknifing procedure.

- **Ecological outlier detection.** Detects outliers based on fundamental niche models (where an outlier is a presence point that has an associated probability of presence lower than a specified value according to a fundamental niche model). Note: this test may not be used by the locality component, but other external tools can make use of it, based on models produced by openModeller.

A new version of DataTester was developed\(^{11}\) (0.2) and a complete documentation for the API\(^{12}\) was made available.

**Environmental data component**

openModeller was originally designed to receive references to environmental layers as parameters for both model creation and model projection. These references are usually

---

\(^8\) [http://www.gbif.org](http://www.gbif.org)  
\(^9\) [http://splink.cria.org.br](http://splink.cria.org.br)  
\(^10\) [http://www.opengeospatial.org/standards/wfs](http://www.opengeospatial.org/standards/wfs)  
\(^12\) [http://gbif.sourceforge.net/datatester/javadoc/](http://gbif.sourceforge.net/datatester/javadoc/)
an address in the local file system pointing to a raster file\textsuperscript{13}. Therefore, the notion about "what layers are available for a modeling experiment" always depended on the users' knowledge about the raster files located in the file system. However, the new remote interaction scenario introduced by the Web Services interface required an additional functionality on the server side to present available layers to users. This was covered by the "getLayers" method. In the existing implementation of the modeling server this method returns all valid rasters that are found in a specific directory (which is determined as part of the configuration) and all sub-directories, recursively.

During the last years, more than 150,000 environmental layers have been used by CRIA in modeling experiments. Of these, approximately 4,000 are in the public domain and were organized in specific directories to be made available through this new modeling service. These layers basically consist of climate data from IPCC (past, present and future), Worldclim data (temperature, precipitation and altitude), topographic data from Hidro1K and vegetation index measurements derived from the Advanced Very High Resolution Radiometer\textsuperscript{14} (AVHRR). Annex 13 contains more information about all available layers.

Access to remote raster data from openModeller is also possible through the TerraLib driver, which can open connections to remote TerraLib databases.

A recent version of GDAL\textsuperscript{15} (1.4.0) adds support to remote rasters that are accessible through the Web Coverage Service\textsuperscript{16} (WCS). This version of GDAL will be tested with openModeller during this year.

\section*{Pre-analysis component}

Sub-sampling locality data into test and training datasets has been implemented to allow extrinsic model validation. This takes place in the “SamplerImpl” class as a global function called “splitSampler” that can also keep the original proportion of presences and absences in the new partitions. “SamplerImpl” is the main class to be used by most pre-processing techniques (documentation about this class can be found in annex 7).

openModeller also provides ways to visualize occurrence points from both the geographical and environmental viewpoints to check if the current samples are widely or poorly distributed in the region of interest from the users' perspective. The console interface includes two programs called “om_viewer” and “om_niche” for these purposes, respectively. The Desktop interface has now basic GIS capabilities to display the original points on top of the generated distribution maps. Conceiving a way to measure how spread out or concentrated are the occurrence points in the environmental dimensions is another feature to be developed.

A new study case is being carried out to compare two algorithms (GARP Best Subsets and Support Vector Machines) which will include two experiments: the first one using a large number of layers, and the other one using only the most important layers after running several pre-analysis techniques like k-means, single-link, average-link, shared nearest neighbors and clustering ensembles (multi-objective, Strehl and Ghosh). The

\textsuperscript{13} For TerraLib rasters, the address has a specific format to point to a table inside a TerraLib database.

\textsuperscript{14} \url{http://edc.usgs.gov/products/satellite/avhrr.html}

\textsuperscript{15} \url{http://www.gdal.org/}

\textsuperscript{16} \url{http://www.opengeospatial.org/standards/wcs}
results of this study case will be used to determine which techniques should be implemented as part of the pre-analysis component. It is also expected that this will provide enough information to check if it is possible to specify a standard pre-analysis interface or if it will be necessary to have multiple interfaces according to each type of pre-analysis technique.

**Modeling component**

A new algorithm called *Environmental Distance* (see annex 15) unified and generalized the existing distance-based algorithms in openModeller: *Minimum Distance* and *Distance to Average*. The new algorithm uses a linear scale of probability between a reference point and a limit specified by the maximum distance parameter. Distances can be calculated using three different metrics: Euclidean, Gower (Manhattan) and Mahalanobis. The reference point is determined by the mean of the “n” nearest points, where “n” is another parameter. When used with Gower and maximum distance of 1, this algorithm is equivalent to the algorithm known as DOMAIN\(^{17}\). Chebyshev is being implemented as a new metric option and will be released in the next versions.

Another algorithm specifically designed to model the distribution of marine organisms was developed in openModeller through a partnership with the Incofish project\(^{18}\). AquaMaps is based on the idea of environmental envelopes, where each variable has an associated preferred range and a broader accepted range. This algorithm differs from other traditional ones since it requires a fixed set of layers to work: maximum depth, minimum depth, sea ice concentration, distance to land, primary production, salinity and sea surface temperature. It also makes use of expert information for the lower and upper limits of each variable and it reads depth ranges from a local database provided by FishBase\(^{19}\) containing data for approximately 30 thousand species. AquaMaps is already available in openModeller’s code repository, but since it’s still being tested by the Incofish experts and also awaiting publishing it has not been publicly released.

An adaptive variation of the GARP algorithm, called AdaptGARP, was implemented but is not yet available in the code repository. AdaptGARP makes use of adaptive decision tables, including an adaptive crossover operator and an adaptive mutation operator. The two algorithms, GARP and AdaptGARP, were tested and compared using two different data sets. For each data set, 14 experiments were performed and the performance was almost identical for both algorithms (see annex 10 for results). The comparison metric used the number of iterations as a function of the fitness value. This work also showed that Adaptive Decision Tables can simulate any genetic algorithm without performance loss, and that previous developments on the subject are a special case for a particular genetic algorithm, e.g. the GARP algorithm.

Furthermore, a significant conceptual gain is obtained since the simulation of genetic algorithms using adaptive devices allows both genetic representation and operators to be unified.

---


19 [http://www.fishbase.net/](http://www.fishbase.net/)
A new document was created to help algorithm developers to contribute to the project. The text contains detailed explanation of the main data structures and methods, including a step-by-step algorithm creation example telling what each line of code does (see annex 5). A presentation called “High level vision to algorithms, data structures and algorithm sequence” was also prepared. It contains an informal description of the data structures and main concepts that need to be understood, and it shows a brief step-by-step sequence of the methods that are invoked during algorithm execution (see annex 6).

Post-analysis component

Model validation is considered one of the priorities for the post-analysis component. openModeller currently offers basic model validation by calculating accuracy, omission and commission errors. This functionality is available from the “ConfusionMatrix” class (see annex 8 for class documentation).

A new command-line tool (om_testsamplesample) was developed and is available in version 0.4.1 to run extrinsic tests. It loads a serialized model, tests each point in a separate file and generates the confusion matrix. Different validation tests shall be included in the future.

Another type of validation test known as ROC Analysis is being developed and should be available in the next versions. ROC Analysis provides a global quality measure based on the “area under the curve” (AUC) which is now being used by several experiments because it does not depend on a cutoff limit that is normally used by other techniques, such as confusion matrix.

Basic statistics about distribution maps can be generated by using the “AreaStats” class (see annex 9 for class documentation). This includes the total number of cells and the number of cells where presence was predicted.

Hotspot analysis was developed as part of the new Desktop interface. This technique can be used to aggregate multiple distribution maps creating a single hotspot map. The same functionality can now be migrated to the framework.

Desktop Interface

During this year, releases 1.0.0 to 1.0.4 of openModeller Desktop were produced. openModeller Desktop is a complete rewrite of the previous omgui 0.3.4 application. The rewrite was prompted by the need to create an application that meets the needs of ‘power users’ and that employs a modular and extensible architecture (see annex 3 for the complete technical training report related to this work).

openModeller Desktop progressed to the point of being a useable, user friendly front end to openModeller library with a rich feature set. Experiments involving multiple species and multiple algorithms can easily be carried out. Plugins were written, allowing modeling tasks to be run on the local machine using libopenmodeller, or on a remote machine via the openModeller Web Service API. Plugins were also written that facilitate automated occurrence data retrieval from speciesLink and GBIF online databases. This new version also sports a new embedded GIS component that allows

---

20 http://splink.cria.org.br
21 http://www.gbif.net
users to interactively explore resulting distribution maps. The latest release includes installation packages for Windows and GNU/Linux. Annex 12 contains a brief analysis of the Windows version. Within the next months another installation package will be made available for Mac OSX.

With the basic functionality for carrying out modeling in place, the plan for the next year is to further extend openModeller Desktop to allow projection into multiple climate scenarios as part of the same experiment, automate model testing and provide tools for visualizing models in the environmental space.

A new TerraView modeling plugin was developed as a separate application. TerraView is a GIS that can be used to build a TerraLib database, as well as to query and visualize its contents in several types of output (such as graphs, maps or charts). It also includes a set of spatial analysis tools for knowledge discovering. This plugin is an independent interface that can be plugged to the currently distribution of TerraView for openModeller users, providing a bridge between openModeller components and the TerraView database. This plugin is available from openModeller’s Subversion repository. A more detailed description can be found in annex 2.

Study Cases

Three study cases were carried out at CRIA to assess species distribution models:

- **Comparison of two different algorithms (GARP and MaxEnt) in modeling the potential habitat of maned wolf (Chrysocyon brachyurus).** The main objective was to know the consequences of actual habitat fragmentation and impacts of future climate changes for this species.

- **Performance test of two different algorithms (GARP and SVM – Support Vector Machine) in modeling the Cerrado tree species (Stryphnodendron obovatum).** The main objective is to compare the accuracy of these algorithms and test the effect of using a high number of environmental layers in the process. This study is not finished yet.

- **Application of different algorithms (Bioclim, GARP and MaxEnt) to compare the known and potential geographic distribution of Hennecartia omphalandra (Monimiaceae).**

Two study cases were carried out at INPE to assess species distribution models:

- **Fabio Iwashita’s master thesis (to be concluded in March 2007) assesses the sensibility of species distribution models according to the precision of locality data.** Two algorithms implemented in openModeller (GARP and BIOCLIM) and Maxent were evaluated (Iwashita, 2007).

- **Different environmental data were tested to achieve better results and optimize modeling procedures.** In particular, Normalized Difference Vegetation Index (NDVI) was used to model the genus Coccocypselum (Rubiacea) (Amaral et al., 2007).

In the last report there was a plan to assess phylogenetic diversity and spatial distribution of Bignoniaceae in the Amazon. Unfortunately, the scholarship was denied. Another study was carried out using a Rubiaceae database, applying knowledge derived from modeling to techniques that help in the conservation of species and

---

22 [http://www.dpi.inpe.br/terraview](http://www.dpi.inpe.br/terraview)
recuperation of degraded areas. Species of Rubiaceae, from the genus Coccocypselum P. Br., were used to test the effect of environmental data on the distribution models. This work also discussed the conservation aspect, contrasting the genus’ predicted distribution and diversity with the conservation units already established in the Brazilian territory (Costa et al., 2006).

A specific TerraLib database with raster data for niche modeling was prepared at INPE. Aside from the environmental data usually considered for this purpose, climate data from CPTEC-INPE, remote sensing imagery, and other maps describing the biophysical environment were used. This activity was carried out during the modeling processes of Rubiaceae, and was also used to successfully test the interface between openModeller and TerraLib.

Other activities

During the year it was felt that a greater proximity of the project teams of the 3 participating institutions was necessary to promote greater interaction. As of June, 2006 to March 2007, 8 seminars were held to present new developments and to advance other discussions about the project. A summary of all seminars can be found in annex 14.

As an additional result from the openModeller project, a study group on Biodiversity Modeling was structured at INPE. Several research initiatives, as papers and thesis, are focused in questions related to biodiversity modeling theory and computational tools. Students and researchers involved promote monthly meetings, called “Referata Biodiversa”23 where ecological and computational aspects of the biodiversity modeling process are presented and discussed. The main objective is to promote the debate about biodiversity and modeling, integrating this multidisciplinary team.

CRIA began offering in house training and remote counseling mostly to graduate students that, in return, contribute their findings to the project and help to improve the tools. The following students were trained in the period, being supervised by Dr Marinez Ferreira de Siqueira:

- Renata Sayuri Kawashima - Instituto Nacional de Pesquisas Espaciais (INPE). Modeling of the potential geographic distribution of the manded wolf (Chrysocyon brachyurus) and future climate implications. Date: July/August 2006
- Sonia Mireya Sua Tunjano - Instituto Amazônico de Investigaciones Científicas (SINCHI), Bogotá, Colômbia. Modeling of potential geographic distribution of woody Amazonian species. Date: October/2006

Dr Marinez Ferreira de Siqueira also conducted the course “Biodiversity Modeling” at the Instituto de Pesquisas Ecológicas (IPE), Nazaré Paulista, SP. Date: Mach, 23-25/2007.

From February 13th to 15th Tim Sutton and Alexandre C. Jardim met at CRIA to discuss and plan the integration of TerraLib in the new Desktop interface.

23 http://www.dpi.inpe.br/referata/index.html
The aRT package was finished, documented and is publicly available. aRT is an R package integrating R with TerraLib GIS classes. It allows spatial data to be analyzed with R.

**Publications and Presentations**

**Published or with a new status**


Kawashima, R.S., Siqueira, M.F. and Mantovani, E. (submitted) Dados do monitoramento da cobertura vegetal por NDVI na modelagem da distribuição

---

24 [http://leg.ufpr.br/aRT/](http://leg.ufpr.br/aRT/)
25 [http://www.r-project.org/](http://www.r-project.org/)
Santana, F.S., Bravo, C. and Saraiva, A.M. (to be submitted) P-GARP implementation requirements for the openModeller framework using the MPI standard. To be submitted.


In preparation


General Comments

During the second year, there was a general expectation from all participant institutions to bring more people to work on the project by means of scholarships and trainingships. Ten proposals were submitted but only four were accepted (one of them began only in 2007). It is important to mention that in all cases the reasons for denial were not related to the merit of work plans or to the quality of candidates, but to the limited number of scholarships available. This had a great impact on various activities, mainly those related to the pre and post processing components. We are trying to find new ways of bringing more people to the project, by establishing new partnerships with other institutions, and are hoping to overcome this situation soon.
Access to biological data from the Amazon region has proved to be more difficult than originally expected. The focus on the Marajó-Tocantins area proposed in the last report is no longer feasible since the necessary partnership that would enable access to data has failed. We would still like to keep the Amazon region as one of the main targets for the next studies, but if the difficulty persists to gather reliable and accurate species occurrence data as well as taxonomic information, the study area will be changed to another region in Brazil.

No further progress has been made in the Web Interface that was prototyped during the first year. A technical trainingship with this specific target is being submitted together with this report.

A new algorithm using Neural Networks was expected for the second year of the project but could not be implemented. We were actually expecting three new algorithms (Neural Networks, Decision Trees and Support Vector Machine) as part of a Postdoctoral scholarship which unfortunately was denied. On the other hand, three other algorithms were implemented (EnvironmentalDistance, AquaMaps and AdaptGARP). A proposal for a technical trainingship specifically related to Neural Networks is being submitted with this report.

Launching of a more stable desktop version and offering training to biologists and ecologists is stimulating the use of the application which we expect will result in a more detailed analysis and a greater demand for new features. The implementation of parallelization techniques in various parts of the code and the availability of a computer cluster dedicated to modelling tasks should soon enable large experiments to be conducted in shorter time.
Annex 01
Release 0.4.1 (2007-03-12)
------------------------

* New algorithm "Environmental Distance".
* Changed OccurrencesReader to handle presence and absence points separately.
* Changed occurrences file format (now TAB-delimited with each point having a unique identifier).
* Initial support to the cmake build system (should work for GNU/Linux and Mac OSX).

Release 0.4 (2006-12-04)
------------------------

* Integration with TerraLib.
* Removed the algorithm Bioclim distance.
* Changed the default output map file type to ERDAS Imagine ".img".
* Refactored SOAP interface which now covers all modelling functionality.
* Re-enabled the new version of GARP (v3).
* Several fixes in the SWIG-Python binding.
* Fixed installation issue with the RPM packages.
Annex 02
APPENDIX 1.

O Plugin do openModeller para o TerraView permite gerar modelos e projeções acessando dados diretamente de uma base TerraLib, sem ser necessário o uso de outro software externo ao TerraView, para isso é necessário selecionar:

- Os algoritmos e seus parâmetros - Figura 1;
- O sistema de coordenadas da projeção (output) - Figura 2;
- Os dados de localidade de uma ou mais espécies (layer, tabela, coluna e espécies) - Figura 3;
- Os dados ambientais para gerar o modelo e a projeção - Figura 4;
- E a mascara e o formato de saida do modelo e da projeção - Figura 5;
- O nome do layer de saida;
- O diretório onde serão gerados os arquivos de log.

Através do botão “Last Model” é possível carregar a ultima configuração usada para gerar um modelo e projeção.

Depois de selecionados todos os parâmetros ao clicar no botão “Finish” o modelo e a projeção começam a ser gerados. Uma janela mostra o progresso da construção do modelo e da projeção, o nome da espécie sobre a qual esta gerando o modelo e a projeção no momento, o numero de espécies que já foram analisadas até o momento, e o numero total de espécies - Figura 6.

Depois de prontos o plugin se fecha automaticamente e a projeção fica disponível para se visualizada através do TerraView - Figura 7.

---

Figura 1 - Algoritmos e Parâmetros.
Figura 2 - Sistema de coordenadas.

Figura 3 - Espécies.
Figura 4 - Camadas Ambientais.

Figura 5 - Mascara e o formato de saida.
Figura 6 - Janela de Progresso.

Figura 7 - Projeção visualizada no TerraView com legenda.
Annex 03

openModeller
Graphical User Interface Progress Report

Tim Sutton
tim@linfiniti.com
March 2007
1 Work Completed

1.1 Migration to a desktop application paradigm

The openModeller Wizard (which is no longer under active development) was based on a wizard paradigm, leading the user through a series of sequential steps to culminate in the generation of an ecological niche model. This wizard paradigm has limitations however. In particular it requires repetitive data entry and is not able to ‘remember’ anything other than the choices made in the previous session. This means that when the user wishes to experiment with different algorithm parameters or environmental layer combinations, they are required to hand enter this information each time.

The move to a desktop application paradigm has increased the learning curve for users new to the application, but it increases the power and flexibility of the application. The new openModeller Desktop incarnation follows a more traditional approach of accessing functionality via menus, icons and a variety of dialogs. This allows significantly more functionality to be provided within the application than could otherwise be offered within the linear workflow of a wizard application.

Statement of work: openModeller Wizard has been rewritten as a desktop application

1.2 Migration from Qt3 to Qt4

The Qt3 C++ toolkit by Trolltech (http://trolltech.com/) provides a framework for building cross platform Graphical User Interfaces (GUIs). Building a cross platform GUI application is a non trivial task. Fortunately the Qt toolkit makes this process easier by providing a uniform API on all major operating systems. It provides an abstraction layer for the operating system’s native GUI tools, so a cross platform application written in Qt still has a native ‘look and feel’. Certain issues with Qt3 prompted the migration to Qt4, including:

- no support for network access via http proxy;
- the windows version of Qt3 is not available under the GPL;
- Qt3 is being superseded by Qt4.

Various other technical reasons prompted the move to Qt4, not least of which was the idea that openModeller Desktop as a ‘green fields’ new application should make use of the very best of available current technologies. With this rationale in mind the codebase from the openModeller Wizard interface that was a likely candidate for re-use in openModeller Desktop was ported to Qt4. This involved manually updating all the deprecated Qt3 API calls to their Qt4 replacements. All new code was written to be compliant with the Qt4 toolkit.

Statement of work: Existing Qt3 code was rewritten to be Qt4 compliant. All newly written code is Qt4 compliant.

1.3 Modular Architecture

One of the goals of the migration to Qt4 and from a wizard style to a desktop style application was to improve the architecture of the application. The wizard based version was written as a proof of concept and employed a monolithic architecture. In addition the application logic was tightly coupled to the user interface which did not facilitate code re-use. Because of this the new openModeller Desktop adopts a more modular architecture to maximise code re-use and to better embrace the principles of Object Orientated Design. Core application logic has been implemented in a separate library (libomg core) and provides functionality that can be used effectively from various parts of the graphical user interface. As part of the modular design, a plugin approach (using the Qt4 plugin helper infrastructure) was developed where applicable to promote application extensibility with clearly defined Application Program Interfaces (APIs). Currently two plugin APIs have been implemented: ‘modeller’ plugins and ‘scraper’ plugins. These are discussed in more detail in Section 1.3.1 and Section 1.3.2.
In addition to modularising via plugins, the code was also written with a discrete non-GUI support library that provides all the backend functionality for the application. This includes classes to represent the various entities within the application such as an experiment class, a model class, classes for representing algorithms, algorithm parameters, layer sets, layers and so forth. This library (libomg core) is used by the GUI to carry out the running of models, the representation of various data objects and so forth. Four core classes were implemented in the GUI part of the code: the main window, the layer set manager, the algorithm profile manager and the experiment designer. A number of other gui components were also implemented in order to provide functionality for user preferences, pre and post processing of data and so on.

These three discrete areas (GUI, omg core, and plugins) provide for a modular three tiered architecture as illustrated in Figure 3. This modularity opens up interesting future possibilities, for example using lib core in other applications to provide access to its high level, non-GUI modelling workflow logic. It is also planned to use some of the desktop application classes to implement new versions of QGIS and TerraView plugins (which will embed openModeller functionality inside these applications). Finally the separation between the application logic in omg core and the user interface will allow us in the future to re-implement a Wizard interface within the openModeller Desktop application while reusing much of the code that already exists. This will allow us, then, to meet the needs of novice and advanced users alike with minimal code and effort duplication.

Statement of work: Much work was carried out in the design of openModeller Desktop so that it properly reflects Object Orientated design principles.

1.3.1 Modelling Engine Plugins

While the openModeller Wizard implementation of the graphical user interface was tightly coupled to the openModeller library, in the design of openModeller Desktop a loosely coupled architecture was preferred. This was mainly preempted by the development of the openModeller Web Service specification which provides an API specification for remote execution of openModeller jobs. Having two invocation methods for openModeller (via local library or remote web service) mandated re-architecting the original openModeller Wizard code base into something more flexible. Special care was taken in the specification of the openModeller Desktop modelling plugin API to ensure that the user would be able to switch between plugins in a seamless manner. As a result of this work, the modeller plugins, as implemented, allow the user to continue to use the same workflows in the application regardless of which plugin is active. Currently two plugins have been implemented (for local and remote web service execution) and both have been tested and are usable. Further work is being carried out to improve their robustness and performance. The modeller plugin API is shown in Figure 3. The plugin approach taken paves the way for addition plugins to be written for any new modelling engine that may be developed in the future.

Statement of work: A pluggable architecture was designed for interfacing with the modeller backend. Two modeller plugins have been written to allow local and web service based models to be run.

1.3.2 ‘Scraper’ Plugins

One of the tasks a user of openModeller Desktop regularly needs to perform is the retrieval of species occurrence data. Various online databases exist (e.g. speciesLink, GBIF Portal) that provide easy browser-based or programmatic access to occurrence data. Typically the data will be returned to the user in response to a web query sent to the server that includes a scientific name of the species for which the data is required. Since these pages return data in HTML, ‘screen scrapers’ were written to programatically retrieve data from online databases and then parse the results into a format that is usable by openModeller Desktop. From this original concept, a plugin API for automating occurrence data retrieval was written. The purpose of the plugin API is to provide a generalised interface for connecting to more such datasources in the future. The scraper plugin API is shown in Figure 3. More recently these services have been
moving to providing programmatic access to data. The plugins, as written, work but need to be updated in the future to make use of these new programmatic interfaces. At that point they will no longer make use of 'screen scraping' techniques, but rather simple XML DOM parsing operations. For legacy reasons this class of plugins are still referred to as 'scraper' plugins however.

Statement of work: A pluggable architecture was designed for interfacing with and retrieving data from online databases. Two plugin implementations were written for retrieving data from speciesLink and the GBIF portal.

1.4 Embedded GIS capabilities for browsing model outputs

The modular architecture of openModeller Desktop supports deployment of core application logic into other applications such as Desktop GIS. However, providing access to basic GIS functions within the openModeller Desktop environment streamlines the process of viewing the models produced during an experiment, whilst still giving access to all the rich modelling centric functionality of the openModeller Desktop application. Quantum GIS (QGIS) libraries were used to provide embedded GIS functionality within openModeller Desktop. As a result of this work, the user is able to zoom in, zoom out, pan and zoom to full extents for any output raster produced by the modelling adapter in use (e.g. the openModeller library, the openModeller Web Service). openModeller Desktop with embedded GIS functionality is illustrated in Figure 3. This is a great improvement over the original openModeller Wizard implementation as it was only able to display model results as static images with no navigation and overlay facilities. Providing the option of embedding openModeller Desktop functionality into QGIS (http://qgis.org) and TerraLib (http://www.dpi.inpe.br/terralib/) (as was available in the openModeller Wizard implementation) is planned for the future.

Statement of work: GIS facilities within openModeller Desktop were made available by creating a map viewing component based on the QGIS libraries.

1.5 Reporting and Provenance

One of the deficiencies of the previous openModeller Wizard GUI was the lack of a proper logging, reporting and provenance. The development of openModeller Desktop has included a new emphasis on reporting. For each model generated, full details of the input datasets, algorithm and algorithm parameters, model run duration, extracted environmental values, modeller engine logs etc are compiled into a report which can easily be printed out. In addition, when a suite of models have been executed, a report for all models can also be generated.

Statement of work: Application logic was written to provide a detailed report for each model that has been run.

1.6 Environmental Values Extraction

In addition to the ability to gather and report on all the provenance for a model, the facility was added to display and export the extracted environmental values for each occurrence point into a comma separated values (CSV) output format. A typical output from this work is shown as Table 1. This provides a useful means to take the data that was used by openModeller library to generate a model and export it into statistical applications or excel for further off-line analysis.

Statement of work: Code was implemented to parse the xml returned by the modeller plugin and render the environmental values for each occurrence point. Additional code was written to enable exporting this to a delimited text file.
Table 1: Environmental Value Extraction: openModeller Desktop can export extracted environmental values into a delimited text file. An example of the typical contents of such a file is shown here.

<table>
<thead>
<tr>
<th>Num</th>
<th>Lat</th>
<th>Long</th>
<th>rain_coolest</th>
<th>rain_hottest</th>
<th>min_tot</th>
<th>temp_avg</th>
<th>temp_cool</th>
<th>temp_dry</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-68.85</td>
<td>-11.15</td>
<td>90</td>
<td>451</td>
<td>1730</td>
<td>2439.4</td>
<td>2256.67</td>
<td>2256.67</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-67.38</td>
<td>-14.32</td>
<td>301</td>
<td>491</td>
<td>1780</td>
<td>2565.01</td>
<td>2330</td>
<td>2466.67</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-67.55</td>
<td>-14.33</td>
<td>305</td>
<td>488</td>
<td>1781</td>
<td>2565.01</td>
<td>2330</td>
<td>2466.67</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-67.58</td>
<td>-14.5</td>
<td>300</td>
<td>486</td>
<td>1769</td>
<td>2397.56</td>
<td>2170</td>
<td>2293.33</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-68.17</td>
<td>-15.25</td>
<td>112</td>
<td>505</td>
<td>1232</td>
<td>2427.53</td>
<td>2240</td>
<td>2256.67</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-67.75</td>
<td>-15.27</td>
<td>145</td>
<td>610</td>
<td>1509</td>
<td>2285.86</td>
<td>2086.67</td>
<td>2100</td>
<td></td>
</tr>
</tbody>
</table>

1.7 Set based layer management

Another limitation of the openModeller Wizard was the fact that working with different and discrete environmental datasets required frequent recapture of the appropriate filenames to be used. In cases where the user wishes to create models based on numerous different environmental scenarios and geographic region, this process can be extremely time consuming and repetitive. openModeller Desktop adopts a ‘set based’ approach to layer management. The user friendly layer manager (see Figure 3) allows the user to build an assemblage of layers and ascribe some metadata to it (name and description). This assemblage is called a layer set. Once the user has created a layer set it can be simply selected by name in other parts of the user interface. With multiple layersets defined, the user can quickly and effortlessly switch between layer sets when running models.

Statement of work: A layerset manager GUI was implemented. The application architecture was implemented to allow the user to deal with layers as conceptual collections rather than individual entities.

1.8 Profile based algorithm management

The openModeller Wizard provides the user with the ability to define specialised parameters for the algorithm being used. However when trying a variety of different parameter combinations over successive model runs, the user is required to remember the parameters used and the application has no way of storing these parameters persistently. With openModeller Desktop this shortcoming is addressed through the implementation of the Algorithm Profile Manager (see Figure 3). The algorithm manager provides the end user a simple and efficient method for carrying out large numbers of models using algorithms with a variety of different settings. For example the user can specify a number of variants of the Bioclim algorithm with slightly different parameters (e.g. 0.4, 0.6 and 0.8 cutoff). When constructing an experiment in the openModeller Desktop interface, the user can use the Algorithm Profile Manager to select 1 or more algorithm profiles and in one operation repeat the model for each species one time per algorithm profile. This is a key enabling technology in meeting the goals of the new desktop user interface - it allows for very large experiments using many algorithm profiles to be conducted with relative ease.

Statement of work: A ‘profile’ based approach to managing model parameters was developed to obviate the need for repetitive data entry and to support the conduction of large scale experiments.

1.9 Cross-platform application deployment and compilation

In addition to the above goals much attention was placed on creating an application architecture that could be compiled on the three major desktop platforms (Windows / Mac OSX / GNU/Linux) and on developing strategies for packaging and deployment on the aforementioned platforms. Currently openModeller Desktop is available for the Windows and GNU/Linux platforms and work is ongoing in porting it to the Mac OSX environment.
Under windows a build environment was developed using Mingw (http://www.mingw.org/) and the gcc (http://gcc.gnu.org/) compiler. A user installer tool was developed using NSIS. The installer can be run in both Portuguese and English. The development version of the installer tool that has been written also provides automated downloading and installation of sample environmental data. The windows installer makes it very easy for users to install openModeller on their desktops.

From an application deployment point of view, GNU/Linux presents some problems in that there are a number of competing installer package formats. Catering for the various formats (rpm, deb, ebuild etc) requires a significant investment of time and effort. As a further complication, different distributions and versions of the same distribution provide different application dependencies, making package deployment a potentially onerous task. Some time and effort was devoted to identifying potential solutions and eventually it was decided to use BitRock (http://bitrock.com) installer. BitRock is non free software, but the choice of using it was pragmatic - since there is a lack of suitable free software equivalents. Bitrock was contacted by the author to ask for a free license since openModeller Desktop is an open source project. Bitrock generously donated a copy for use in the openModeller project. From the users point of view, BitRock provides a similar user experience to installing an application under windows. Building the installer took a fair amount of effort since it required compiling all application dependencies and writing various scripts to ensure when the bitrock installed application is executed it uses the correct library search path to fine libraries and data that the application depends on.

Cross platform compilation was achieved using the Qt4 qmake build system. This build system does have some limitations however, in particular:

• on Mac OS X the build will not work when QGIS integration is enabled due to multiple symbol import issues. This has prevented the creation of Mac OS X application bundles for general distribution to date

• qmake does not automatically find the build dependencies on the system

• within the project we are using multiple build systems: qmake, automake and Visual Studio Solutions

To address these issues, work has started on the creation of a unified build system using cmake (http://www.cmake.org). This will allow the openModeller Library and openModeller Desktop to be built for all operating system with both gcc and msvc compilers using a single build system and will significantly reduce the overhead (both in terms of user support and in terms of developer time) that currently goes into maintaining three different build systems. Using cmake the author is now able to compile openModeller and openModeller Desktop on the Mac OS X platform. A generally available Mac OS X application bundle is expected to be available in the near future.

Statement of work: openModeller Desktop has been developed to be able to run on all three major desktop operating systems. Currently Windows and GNU/Linux installers are available, with a redistributable Max OS X application bundle to be available in the near future.

1.10 Tracking changes to libopenmodeller

The openModeller library and the openModeller Web Services interface have been under ongoing development during the course of the reporting period. In order for the modelling plugins to continue working against the most current versions of libopenmodeller and the openModeller Web Service, regular code revisions were made. This has enabled new features in the library to be available within openModeller Desktop within a short period of them being implemented.

Statement of work: openModeller Desktop was kept current with ongoing developments in the openModeller Library and openModeller Web Services.
1.11 Pre and post processing tools

A set of classes were written for pre processing environmental data to produce aggregate layers. For example from average monthly temperature and average monthly rainfall, we can compute new layers such as rainfall in month with highest temperature. The backend logic for this has been implemented but further work is required to expose this via the user interface.

Post processing tools were incorporated into openModeller Desktop to produce false colour representations of the grayscale model outputs, and to produce a GIS shapefile from the original occurrence points. A hotspot tool was also produced and is undergoing some minor bug fixing before being made generally available via an openModeller Desktop end user release.

Statement of work: Initial pre and post processing tools have been implemented. Some of these tools are already included in the public version of openModeller Desktop, while others are in the final stages of development and quality control and will be made available in a future release.

1.12 Internationalisation

The Qt4 provides a framework for internationalisation. This framework was used to ensure that text openModeller Desktop can be translated into different languages. Work was done to make the build system automatically generate updated translation files, and to automatically detect the current system locale and use an appropriate language. Marneza Ferreira de Siqueira assisted by performing a Portuguese translation. Through this work the openModeller Desktop interface can be used in English or Portuguese, and the possibility now exists to easily add additional language translations in the future.

Statement of work: openModeller Desktop is internationalised, with ability to use it in English or Portuguese.

2 Impacts of work on the greater project

2.1 Project Publicity

A software project cannot exist in isolation from its users. During the reporting period, much effort was made to set up an infrastructure for disseminating information relating to openModeller Desktop and the openModeller Project as a whole. This included a revision of the openModeller home page (http://openmodeller.sf.net). The revision had three main aims:

1. To improve general clarity and effectiveness with which the openModeller project is described;
2. To make navigation and finding relevant information easier for the visitor;
3. To update the resources relating the the graphical user interface to reflect the new developments of openModeller Desktop.

The revised version of the openModeller home page is illustrated in Figure 3.

Early releases of openModeller Desktop were not widely advertised or distributed due to the fact they were considered snapshot release and not ready for general usage. Sourceforge provides a monitoring infrastructure that allows the amount of traffic to the web site to be visualised. When the first releases of openModeller desktop were made at then end of December 2006, the project was advertised on several high profile mailing lists, blogs and software directory listings. The effect was evidenced in the increase in the number of downloads (see Figure 3 ). Since openModeller Desktop has been released it has been downloaded over 500 times.
2.2 Publications and Presentations

The author participated in the writing of the following scientific papers, one of which has been submitted for publication, and the other which is in preparation for being submitted:


The author also gave a presentation at the Taxonomic Databases Working Group (TDWG) 2006 conference in Missouri (http://tdwg2006.tdwg.org/). The openModeller project was described and an overview of the openModeller Web Service API was given. The abstract for this presentation is provided below. The presentation was very successfully at publicising the openModeller project to the broader scientific community. Directly as a result of this, TDWG seed funding was applied for and granted to hold a workshop at CRIA (Campinas, Brazil) in April. The purpose of the workshop is to evaluate possibilities for interoperability between various online services, with openModeller to play a central role in this.

2.3 Other Contributions

Various other contributions to the project were made over the course of the reporting period including:

• supporting the work of students (Danilo Bellini and Alexandre Jardim) in using the openModeller library and the openModeller project management tools;

• enhancements to the openModeller Library including working on the removal of all compiler warnings, implementing support for writing to various new raster formats and various other improvements;

• carrying out the migration of the openModeller source code management from CVS to SVN;

• documenting build procedures and other aspects of the project on the project wiki;

• collaborating with Renato De Giovanni on the specification of the openModeller Web Service and writing a test client for this service;

• giving various presentations to monthly openModeller project seminars;

• dealing with openModeller related queries on the openModeller users list;

• managing the bug and feature trackers;

• a training video was created that introduces users to the openModeller Desktop user interface;

• promotion of openModeller to third parties.
3 Figures

Figure 1: High level view of the three tiered architecture of openModeller Desktop. Only key classes for the openModeller Desktop application are shown here.
Figure 2: Inheritance diagram for OmgModellerPluginInterface

```
OmgModellerPluginInterface

# mMessenger
+ OmgModellerPluginInterface()
+ createModel()
+ projectModel()
+ getAlgorithmList()
+ getAlgorithmSet()
+ getLayers()
+ getName()
+ getMessenger()
+ getModelDefinition()
# OmgModellerPluginInterface()
```

```
OmgModellerLocalPlugin

- myOpenModeller
- mPlugInServices
- mTextStream
- mModelProgress
- mModelGuid
- mModelLog
- mApplicationStartedFlag

+ OmgModellerLocalPlugin()
+ OmgModellerPluginInterface()
+ createModel()
+ projectModel()
+ getAlgorithmList()
+ getAlgorithmSet()
+ getAlgorithm()
+ setCreationProgress()
+ setProjectProgress()
+ getLayers()
+ getName()
- Q_INTERFACE(S)
- initialize()
- setModelProgress()
- appendToLog
```

```
OmgModellerWebServicesPlugin

- mModelProgress
- mMapProgress

+ OmgModellerWebServicesPlugin()
+ OmgModellerPluginInterface()
+ createModel()
+ projectModel()
+ getAlgorithmList()
+ getAlgorithmSet()
+ getLayers()
+ getName()
+ convertToWorldChar()
+ getSoapConnection()
- Q_INTERFACE(S)
- initialize()
```

Figure 3: Inheritance diagram for OmgScraperPluginInterface

```
OmgScraperPluginInterface

# mLocationVector
# mTaxonName
# mFileImage

+ OmgScraperPluginInterface()
+ OmgScraperPluginInterface()
+ search()
+ getName()
+ createShapefile()
+ createMapfile()
+ getstatus
```

```
OmgScraperSbf

+ OmgScraperSbf()
+ OmgScraperPluginInterface()
+ search()
+ getName()
- Q_INTERFACE(S)
- taxonRequestDone
- localiseRequestDone
```

```
OmgScraperSlink

+ OmgScraperSlink()
+ OmgScraperPluginInterface()
+ search()
+ getName()
- Q_INTERFACE(S)
```
Figure 4: GIS browsing is now embedded in openModeller Desktop. Occurrence records for this model are show overlaid as black dots with white borders.

Figure 5: The algorithm profile manager allows the user to name the parameter choices used for an algorithm. This makes it easy to follow an experimental approach and still have be able to easily run a model with a given set of parameters used in a past model run.

Figure 6: The layer set manager provides the facility to create named collections of layers, and removes the need for repetitive data entry.
Figure 7: A screenshot showing how the openModeller website looks after being revised to improve usability and better publicise openModeller Desktop.

Figure 8: Downloads of openModeller software over the 12 month period April 2006 - March 2006. Note this graph was produced at the beginning of March 2007, so the complete statistics for March are not represented here.
Annex 04
FUNDAÇÃO DE AMPARO A PESQUISA DO ESTADO DE SÃO PAULO

Relatório de Iniciação Científica

Arcabouço para Distribuição e Modelagem de Espécies – Uma Análise de Desempenho

Bolsista: Mariana Ramos Franco
Graduanda em Engenharia de Computação,
Escola Politécnica da Universidade de São Paulo

Orientador: Prof. Dr. Pedro Luiz Pizzigatti Corrêa

Departamento de Engenharia de Computação e Sistemas Digitais,
Escola Politécnica da Universidade de São Paulo

Número do Processo: 2006/03616-9

Resumo

Balancear desenvolvimento social e econômico com conservação ambiental é um grande desafio. Existe uma grande demanda por ferramentas de software que determinem o nicho ecológico fundamental dos organismos. Tais ferramentas podem ajudar a compreender a ocorrência e distribuição das espécies biológicas, tal como aquelas em risco de extinção. O arcabouço openModeller foi desenvolvido para fornecer o ambiente de software necessário para conduzir tais análises.

O openModeller é um sistema (open source) de modelagem de distribuição espacial de espécies. Os modelos são gerados através de um algoritmo que recebe como entrada um conjunto de pontos de ocorrência (latitude/longitude) e um conjunto de variáveis ambientais. Um dos diversos algoritmos pode ser usado para gerar o modelo e a projeção das probabilidades. Os algoritmos avaliam as circunstâncias ambientais dos locais de ocorrência conhecidas da espécie, e computam o nicho ecológico perfeito.

Para otimizar o desempenho do openModeller foi conduzido um estudo cuidadoso das classes individuais e do fluxo de uma execução típica. Neste estudo a estrutura foi dividida em diversos componentes e uma carga típica foi definida com a ajuda dos usuários do sistema. Os resultados preliminares produzidos serão úteis para identificar os componentes que poderão ser processados de maneira paralela e distribuída.

Estes resultados foram também usados para propor um modelo de análise de desempenho. O objetivo desse modelo é ajudar o desenvolvimento de uma infra-estrutura considerando desempenho, disponibilidade do sistema, interoperabilidade e custo financeiro.

Palavras chaves: modelagem de espécies, análise de desempenho, arquitetura distribuída de alto desempenho, programação orientada a aspectos.

Abstract

Balancing social and economic development with environmental conservation is a major challenger. There is a strong demand for software applications to determine the fundamental ecological niche of organisms. Such tools can help us to understand the occurrence and distribution of biological species, such as invasive or endangered species. The openModeller framework was developed to provide a software environment for conducting such analyses.

openModeller is an open source static spatial distribution modeling framework. Models are generated by an algorithm that receives as input a set of occurrence points (latitude/longitude) and a set of environmental layer files. One
of several algorithms can be used to produce a model and generate a probability surface. The algorithms evaluate the environmental conditions at known occurrence sites and then compute the preferred ecological niche.

For optimizing the performance of openModeller it was conducted a careful study of individual classes and the typical execution flow chart. In this study the framework was divided into several components. A typical workload was defined with the help of end users. Preliminary results were produced that will be useful in identifying the components that can be processed in alternative parallel and distributed ways.

These results were also used to propose a performance analysis model. The aim of the performance analysis model is to aid the development of an infrastructure that will deliver optimal performance, maximum system availability, interoperability and that minimizes financial costs.

Keywords: Species distribution modeling, Performance evaluation, high performance and distributed application, Aspect Oriented Programming.
Sumário

1. Introdução 2
2. Objetivos 3
2.1. Objetivos Gerais 3
2.2. Objetivos Específicos 3
3. Revisão Bibliográfica 4
3.1. Linguagens de Programação: C++/C# 4
3.2. Framework .Net 5
3.3. Paradigmas de Programação: OOP/ AOP 5
3.4. Mecanismos para comunicação entre processos 7
3.5. Visual Studio/ SQLServer 8
3.6. Modelagem de Distribuição de Espécies 9
3.7. Metodologias de Análise de Desempenho 11
3.8. Técnicas de Análise de Desempenho: Teoria das Filas e Simulação 13
3.9. Arena 13
4. openModeller: organização do código e suas classes 14
5. Atividades Realizadas e Resultados Obtidos 18
5.1. Uso de AOP para coleta de parâmetros de desempenho. 18
5.2. Levantamento dos componentes e de suas interfaces. 19
5.3. Wrapper em C# e portabilidade para o COM+. 23
5.4. Arquitetura Implementada 25
5.5. Modelo preliminar de simulação. 28
5.6. Publicação. 30
6. Conclusão 31
7. Instituições Envolvidas 32
8. Equipe do Projeto 33
9. Bibliografia 34
Anexo 36
Assinaturas 37
Introdução

Vários estudos têm sido realizados para identificar as regiões com maior risco de perda de biodiversidade no planeta. A identificação dessas áreas é importante para direcionar investimentos e esforços na conservação e proteção desses Biomas ameaçados. Para tanto, é necessário o desenvolvimento de processos que auxiliem o trabalho de produzir, trabalhar e usar a informação disponível (SIQUEIRA, 2005).

Atualmente vêm sendo desenvolvidas e utilizadas várias técnicas de modelagem de distribuição geográfica de espécies que através de dados climáticos e espaciais, conseguem prever as áreas propícias ao desenvolvimento de determinadas espécies, utilizando algoritmos complexos como o GARP e o BIOCLIM (CANHOS et al., 2005). Através desses resultados é possível avaliar o impacto das mudanças climáticas na biodiversidade, identificar espécies que poderiam ser utilizadas em trabalhos de recuperação ambiental, avaliar o potencial de ameaça de espécies invasoras e auxiliar na determinação de áreas prioritárias para conservação (SIQUEIRA, 2005).

Como essas técnicas de modelagem necessitam de uma grande quantidade de informações e de cálculos complexos, há uma demanda por ferramentas computacionais capazes de apresentar de forma simples e rápida a análise e visualização espacial desses dados, assim como para a construção de cenários de impacto e vulnerabilidade ambiental. Entre essas ferramentas computacionais podemos encontrar o sistema openModeller (SOURCE FORGE, 2006).

openModeller é um sistema (open source) de modelagem de distribuição espacial de espécies desenvolvido inicialmente pelo CRIA (Centro de Referência em Informação Ambiental). Atualmente o CRIA, o INPE (Instituto de Pesquisas Espaciais) e a Escola Politécnica da USP vem desenvolvendo, com o apoio da Fapesp, um arcabouço para a modelagem e predição da distribuição de espécies biológicas, a partir da versão inicialmente disponível do openModeller (CANHOS et al., 2005).
O sistema openModeller faz uso intensivo de recursos computacionais para avaliar e apresentar o conhecimento presente em conjuntos de dados geográficos e de localização de espécies. Assim, tornou-se importante um estudo detalhado sobre a melhor maneira de configurar a distribuição de recursos computacionais de alto desempenho, tendo como principais parâmetros a disponibilidade do sistema, interoperabilidade e o custo financeiro.

Através deste trabalho de iniciação científica foi possível realizar um estudo das funcionalidades de cada classe do sistema e do fluxo de uma execução típica, podendo-se assim dividir esse arcabouço em diversos componentes. Diversas cargas de trabalho foram utilizadas tornando possível levantar resultados que permitiam identificar os componentes passíveis de serem processados de forma paralela e distribuída e, deste modo, foi proposto um modelo de análise de desempenho que permite direcionar a preparação da arquitetura e infra-estrutura que suportará a solução.

Objetivos

Objetivos Gerais

Analizar o desempenho da versão atual do openModeller e propor uma configuração da arquitetura e infra-estrutura de alto desempenho do sistema, aderente aos requisitos do projeto, através da análise de desempenho de cada componente do arcabouço e da demandas de utilização de recursos computacionais.

Objetivos Específicos

- Geração de um ambiente dedicado a estudos e provas de conceitos, estudo de tecnologias, execução e instrumentação do arcabouço openModeller;
- Identificação e documentação dos componentes e relacionamentos existentes entre eles, encontrados no sistema atual do openModeller;

- Levantamento de métricas de desempenho de cada componente tais como: volume e freqüência de dados manipulados pelos componentes, além do tempo de processamento dos componentes;

- Definição de um modelo baseado na teoria de filas para simulação dos componentes (JAIN, 1999);

- Estudo de tecnologias de desenvolvimento de software baseado em “aspectos”, assim como do ambiente de desenvolvimento de sistemas voltado para componentes (NATHAN, 2002).

**Revisão Bibliográfica**

A seguir, estão descritos os conceitos, as técnicas e ferramentas discutindo sua aplicação na pesquisa.

**Linguagens de Programação: C++/C#**

O código fonte do sistema em desenvolvimento do openModeller utiliza a linguagem de programação C++ (ECKEL, 2003), além disso neste trabalho foi adotado C# (LIBERTY, 2005) na implementação dos wrappers, para facilitar a aquisição de métricas de desempenho. Assim neste item será feita uma breve apresentação para referenciar parte dos estudos realizados no aprendizado destas linguagens.

C++ é uma linguagem de programação de alto nível (longe da linguagem de máquina, mais próxima da linguagem humana) com facilidades para o uso em baixo nível (mais próxima ao código de máquina), multiparadigma (estruturada ou orientada a objetos) e de uso geral. Deste os anos 90 é uma das linguagens de programação mais populares (ECKEL, 2003).

Como ambiente de desenvolvimento foi utilizado o Microsoft .NET que é uma iniciativa da Microsoft que visa uma plataforma única para desenvolvimento e execução de sistemas e aplicações. Todo e qualquer código gerado para .NET, pode ser被执行ado em qualquer dispositivo ou plataforma que possua um framework: a "Plataforma .NET" (.NET Framework). Com idéia semelhante à plataforma Java, o programador deixa de escrever código para um sistema ou dispositivo específico, e passa a escrever para a plataforma .NET (NATHAN, 2002).

Paradigmas de Programação: OOP/ AOP

A Programação Orientada a Objetos (Object–Oriented Programing: OOP) (ECKEL, 2000) é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

A análise e projeto orientado a objetos têm como meta identificar o melhor conjunto de objetos para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definidos nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

O sistema estudado foi todo desenvolvido procurando seguir o paradigma de orientação a objetos.

Como citado no Item 2.1, parte dos objetivos específicos dessa pesquisa envolveu a geração de um ambiente dedicado a estudos e provas de conceitos, execução e instrumentação do arcabouço

A Programação Orientada a Aspectos (Aspect-Oriented Programming: AOP) é um paradigma que permite aos desenvolvedores de software separar e organizar o código de acordo com sua importância para a aplicação (separation of concerns). Todo programa escrito no paradigma orientado a objetos possui código que é alheio a implementação do comportamento do objeto. Este código é todo aquele utilizado para implementar funcionalidades secundárias e que encontra-se espalhado por toda a aplicação (crosscutting concern). A AOP permite que esse código seja encapsulado e modularizado (RODKIN, 2004).

Os paradigmas de programação mais antigos, como a programação procedural e programação orientada a objeto, implementam a separação do código, através de entidades únicas. Por exemplo, a funcionalidade de log de dados, numa linguagem orientada a objetos, é implementada em uma única classe, que é referenciada em todos os pontos onde é necessário rastrear a execução. Como praticamente todo método necessita que alguns dados sejam registrados em log, as chamadas a essa classe são espalhadas por toda a aplicação.

Tipicamente uma implementação da AOP busca encapsular essas chamadas através de uma nova construção chamada de "aspecto".

Existem três conceitos importantes definidos na AOP:

- **Joinpoints**: são pontos bem definidos ao longo da execução do programa, podem ser, por exemplo: chamadas de métodos, acessos a membros de uma classe, criação de objetos.
- **Pointcuts**: construção de linguagem que junta um conjunto de joinpoints baseando-se em um critério pré-definido.
• **Advice**: é o trecho de código que é executado antes, depois ou simultaneamente a um *joinpoint*.

Um aspecto pode alterar o comportamento de um código (a parte do programa não orientada a aspecto) pela aplicação de comportamento adicional, *advice*, sobre um "ponto de execução", ou *join point*. A descrição lógica de um conjunto de *join points* é chamada de *pointcut*.

A utilização da AOP foi a forma encontrada para fazer a coleta das métricas de desempenho sem que isso interferisse nos dados coletados.

**Mecanismos para comunicação entre processos**

Neste item são apresentados mecanismos que permitem a comunicação entre processos (programas em execução) considerados nesta pesquisa: RPC, DCOM/COM+ e MSMQ.

• **RPC**

A Chamada de Procedimento Remoto ou RPC (*Remote Procedure Call*) é um *protocolo* para chamada remota de procedimentos para a transferência de controle de parte de um processo para outro. O *protocolo RPC* pode ser implementado sobre diferentes *protocolos de transporte* (TANENBAUM, 2002).

• **DCOM**

DCOM (*Distributed component object model*) é uma tecnologia proprietária da *Microsoft* para criação de *componentes de software* distribuídos em computadores interligados em rede. O DCOM é uma extensão do *COM* (também da Microsoft) para a comunicação entre objetos em *sistemas distribuídos* (TANENBAUM, 2002).

DCOM substitui a comunicação local entre processos por um protocolo de rede (semelhante ao RPC). Tratando detalhes de baixo nível de
protocolos de rede enquanto o desenvolvedor preocupa-se somente em implementar aspectos funcionais da aplicação.

- **COM+**

Criado pela Microsoft, o COM+ (Component Object Model) é uma plataforma independente, distribuída e orientada a objeto para a criação de componentes de software que possam interagir num sistema distribuído, especificando um padrão binário que define como os objetos podem interagir (NATHAN, 2002).

Através do COM+ os componentes do framework foram distribuídos para a realização da análise de desempenho do mesmo.

- **MSMQ**

O MSMQ (Microsoft Message Queuing) é essencialmente um protocolo de dados que permite aplicativos rodando em diferentes servidores se comunicarem. Assegura a entrega de confiança colocando as mensagens que não alcançam seu destino pretendido em uma fila e então as reenviando quando o destino estiver disponível. Isto permite comunicação entre diferentes redes e computadores que nem sempre podem ser conectados (MESSAGE QUEUING, 2007).

Suporta também transações e permite operações múltiplas em filas múltiplas, co todas as operações envolvidas em uma única transação.

**Visual Studio/ SQLServer**

O Microsoft Visual Studio (DEVELOPER CENTER, 2007) é a plataforma de desenvolvimento adotada nesta pesquisa como ambiente de desenvolvimento para estudo do openModeller, pois permite a utilização dos mecanismos de comunicação entre processos utilizado: DCOM/COM+, empregados no desenvolvimento dos wrappers. O Microsoft Visual Studio é um conjunto de ferramentas integradas para
desenvolvimento de software, especialmente dedicado ao framework .Net e as linguagens de programação Visual Basic (VB), C, C++, C# e J#.


Por esses motivos e pela facilidade de utilização junto com o Visual Studio, foi o sistema de banco de dados escolhido para armazenar as métricas de desempenho obtidas.

Modelagem de Distribuição de Espécies

Para entender o funcionamento do sistema openModeller foi preciso aprender sobre os procedimentos realizados na modelagem de distribuição de espécies e sua importância. Isso foi possível através da leitura de artigos acadêmicos sobre o assunto (Siqueira, 2005) e também através do contado com pesquisadores do CRIA que trabalham com esse tipo de modelagem.

Foi realizado também um estudo sobre os algoritmos de modelagem disponíveis no openModeller:
- **Bioclim:**

Para cada variável ambiental dada, o algoritmo acha o desvio médio e padrão (presumida distribuição normal) associado ao ponto de ocorrência. Neste modelo, os pontos podem ser classificados como: Apropriados: se todos os valores ambientais associados caírem dentro do intervalo calculado; Marginal: se um ou mais valores ambientais caírem fora do intervalo, mas ainda dentro dos limites superiores e inferiores. Inadequado: se um ou mais valores ambientais caírem fora dos limites superiores e inferiores. A saída determinada pelo *Bioclim* é mapeada para as probabilidades de 1.0, 0.5 e de 0.0 respectivamente.

- **Bioclim Distance:**

A especificação do *Bioclim* original define três regiões: apropriado, marginal e inadequado. A região apropriada relaciona-se ao intervalo mencionado antes, e é a única região considerada nesta execução (isto é, pontos com queda dentro de marginal ou inadequado têm probabilidade 0 aqui). A probabilidade de presença para pontos dentro da região apropriada é inversamente proporcional à distância euclidiana normalizada entre um ponto e o ponto médio.

- **Climate Space Model:**

O *Climate Space Model* (CSM) é a princípio baseado em componentes. O processo de seleção dos componentes neste algoritmo considera que todo componente abaixo do valor médio (média de uma amostra randômica) é rejeitado.

- **Distance to Average:**

Normaliza os valores ambientais e os parâmetros (de acordo com o número de variáveis ambientais). Calcula o ponto médio no espaço ambiental que considera todos os pontos da presença. Ao projetar o resultado, calcula a distância euclidiana entre o ponto
médio e cada ponto no espaço ambiental. Se a distância ‘dist’ estiver em [0, MAXDIST] então a probabilidade da ocorrência estará em [1, 0]. Se ‘dist’ > MAXDIST então a probabilidade será zero, onde MAXDIST é um dos parâmetros a serem passados para o algoritmo.

- **Genetic Algorithm for Rule-set Production (GARP):**

  GARP é um algoritmo genético que cria modelos de nicho ecológicos para espécies. Os modelos descrevem as circunstâncias ambientais em que a espécie deve poder manter populações. Como entrada, GARP usa um conjunto de pontos de localização onde é conhecida a ocorrência da espécie e um conjunto de camadas geográficas que representam os parâmetros ambientais que podem limitar as potencialidades de sobrevivência da espécie.

- **Minimum Distance:**

  Normaliza os valores ambientais das variáveis e o parâmetro (de acordo com o número de variáveis ambientais). Calcula a distância entre as circunstâncias ambientais dadas a cada ponto de ocorrência e seleciona a distância mais próxima. Se a distância ‘dist’ estiver dentro de [0, MaxDist] então a probabilidade estará dentro de [0,1]. Se ‘o dist’ > MaxDist então a probabilidade será zero.

Metodologias de Análise de Desempenho

Para a realização da análise de desempenho foi estudada uma metodologia (BRESSAN, 2003), que contribuiu para que não ocorressem enganos durante o processo de análise e para guiar o andamento dos trabalhos.

Os passos da metodologia adotada foram os seguintes:

- **Definição dos objetivos e planejamento da análise:** definição clara dos objetivos e dimensionamento do sistema, configuração do sistema a ser
analisado, cronograma para realização do estudo e os recursos necessários.

- **Definição do sistema**: determinar as características a serem analisadas e as fronteiras que delimitam o sistema.

- **Enumeração dos serviços e resultados do sistema**: identificar objetivos, serviços realizados e resultados esperados.

- **Seleção de métricas de desempenho**: determinar as formas de medir o desempenho do sistema.

- **Enumeração dos parâmetros da análise**: coleta dos dados para determinação dos parâmetros.

- **Seleção dos fatores a serem estudados**: entre os parâmetros identificados, serão selecionados aqueles que tem maior impacto no desempenho e que serão variados na análise.

- **Seleção da carga de trabalho**: projeção da carga a ser submetida ao sistema, considerando a carga típica e em momentos de pico.

- **Definição do modelo conceitual de sistema**: especificar a concepção que se tem do sistema e tudo o que foi assumido.

- **Seleção da técnica de modelagem**: escolha das ferramentas a serem utilizadas na análise.

- **Realização da modelagem**: planejamento e execução dos experimentos.

- **Definição e projeto dos experimentos**: escolher os diversos níveis (ou valores) dos fatores que afetam o desempenho do sistema, entre os quais a carga a ser submetida e aqueles que determinam características do sistema.

- **Validação do modelo**: revisão do modelo para verificar se os resultados estão corretos e dentro do esperado.
- **Execução dos experimentos**: realização dos experimentos com o modelo desenvolvido e validado utilizando os níveis de carga escolhidos.

- **Análise e interpretação dos resultados**: verificação estatística para a correta interpretação dos resultados.

- **Documentação e apresentação dos resultados**: deverá ser documentado o modelo conceitual do sistema e tudo o que foi assumido em termos de parâmetros e fatores, assim como os resultados obtidos e suas análises.

**Técnicas de Análise de Desempenho: Teoria das Filas e Simulação**

Também foi necessário o estudo de modelos de simulação de sistemas computacionais para que fosse possível determinar um modelo adequado para a realização da análise de desempenho do sistema, de forma que se obtivessem resultados condizentes com a realidade.

A Teoria das Filas e a Teoria da Simulação de Sistemas (JAIN, 1999) são técnicas utilizadas para planejamento que permitem a modelagem e a simulação de um sistema computacional. Elas constituem a base teórica de modelos de sistemas computacionais, relacionados com simulação e foram utilizadas para a construção do modelo de componentes do openModeller.

A Teoria das Filas é um método analítico de que aborda o assunto por meio de fórmulas matemáticas. Já a Simulação é uma técnica que, usando o computador digital, procura montar um modelo que melhor represente o sistema em estudo.

**Arena**

O Arena é um software de simulação que utiliza um conjunto de blocos (ou módulos) para descrever uma aplicação real. Estes blocos funcionam como comandos de uma linguagem (else, if, while ...) facilitando em muito a programação (PRADO, 2004).
Através do Arena é possível simular o funcionamento do modelo de filas criado para o *framework openModeller*.

**openModeller: organização do código e suas classes**


Pela falta de documentação e pela complexidade do sistema não é de fácil percepção o seu funcionamento, mas através de um estudo detalhado foi possível entender a organização do código e o seu funcionamento apresentado a seguir.

O funcionamento do *openModeller* segue os mesmos passos que uma modelagem de nicho ecológico comum que para realizar a distribuição de espécies, utiliza pontos de ocorrência de uma espécie em uma determinada região e a soma das variáveis ambientais desses pontos (como temperatura ou índice pluviométrico) para gerar um modelo de distribuição. Em uma segunda etapa, esse modelo é projetado em qualquer região geográfica contendo os mesmos tipos de variáveis ambientais, com o objetivo de se obter a probabilidade de ocorrência da espécie na região.

Para realizar a modelagem de nicho, o pesquisador deve seguir os seguintes passos:

- Ter um arquivo com pontos de ocorrência da espécie e mapas georeferenciados com as variáveis ambientais relevantes (definido a critério do pesquisador);
- Escolher um algoritmo adequado para o cálculo do modelo e para a projeção (sempre deve ser o mesmo algoritmo) e os parâmetros de ajuste desse algoritmo;
Figura 1. Passos para a modelagem.

O resultado da projeção é uma imagem em formato TIF em escala de cinza georreferenciada contendo a probabilidade de distribuição da espécie e a impressão de estatísticas relacionadas, como acuidade do modelo e porcentagem de ocupação da espécie na área.

Levando em conta a arquitetura e organização das classes do openModeller, foi possível descrever os seguintes itens:

- Arquivos de entrada: arquivo texto de ocorrência de espécie, mapas georreferenciados utilizados para configurar as variáveis ambientais, mapa para a projeção do modelo e arquivo de configuração do sistema;
- Biblioteca openModeller: biblioteca para processamento de entrada e saída, geração de modelo e projeção de distribuição de espécie;
- Algoritmos: cada algoritmo é constituído de um componente próprio que pode ser removido ou adicionado ao sistema, basta implementar uma estrutura de classes pré-definida. Ex: GARP, BIOCLIM, Minimum Distance.
- Arquivos de saída: Arquivo TIF em escala de cinza e georreferenciado representando a probabilidade de ocorrência da espécie naquela área;
- Bibliotecas auxiliares: bibliotecas open-source utilizadas como componentes, elas são:
  - GDAL: Manipulação de imagens rasterizadas e georreferenciadas;
  - EXPAT: Manipulação de arquivos XML
  - GSL: Biblioteca matemática utilizada no algoritmo BioClim.
O processo do trabalho do openModeller pode ser entendido também através da organização das entidades que compõe o sistema, conforme descrito a baixo:

- **Entrada de dados:** os pontos de ocorrência de uma espécie estão representados através das classes Occurrence e Occurrences, as variáveis ambientais são representadas pela classe Environment, que reúne uma colecção de mapas georeferenciados. Os parâmetros de configuração das classes Algorithm e AlgorithmMetaData servem para instanciar o algoritmo escolhido com os parâmetros necessários;

- **Geração de modelo:** A classe openModeller é responsável por configurar a classe Algorithm com as ocorrências e as varáveis ambientais e por controlar a execução do algoritmo até que o modelo

- **Projeção da distribuição de espécie:** A classe Projector recebe o modelo através de um objeto Model, o mapa de saída no objeto Map e as variáveis ambientais através do objeto Environment. Com isso, para
cada valor de latitude e longitude do mapa de saída, ela chama o modelo para calcular a probabilidade de ocorrência da espécie naquele ponto.

- Cálculo estatístico: Cada probabilidade calculada durante a projeção é armazenada na classe AreaStats, que juntamente com a classe MatrixConfusion, calcula as consolidadas do processo de projeção.

Portanto o funcionamento do openModeller apresenta dois processos principais: modelagem e projeção.

a. Processo de modelagem

A criação do modelo de distribuição se dá através da função “CreateModel” da classe Algorithm.

Como já mencionado, o openModeller suporta vários algoritmos. A função CreateModel na verdade chama várias funções genéricas que são implementadas em uma DLL (Dynamic Link Library) para cada um dos algoritmos suportados. Um único algoritmo é utilizado a cada execução: ele é selecionado a partir da leitura do arquivo “request.txt” que é responsável pela configuração da modelagem.

b. Processo de projeção

Toda a projeção é feita pelo método createMap da classe Projector que recebe como entrada o modelo e as variáveis ambientais de saída e gera a projeção em um mapa de saída, guardando também a estatísticas pertinentes. Assim podemos dividir os parâmetros do método em duas partes:

- Parâmetros de entrada:
  - Environment: Classe que contém as variáveis que representam o ambiente onde a espécie irá ser projetada, cada variável é representada pela classe Layer (Camada) e cada camada têm uma classe Map representando um mapa rasterizado e georeferenciado;
- **Model**: Classe que representa o modelo matemático de distribuição de espécies, esse modelo contém uma média para cada ponto de ocorrência da espécie e será utilizado para calcular a probabilidade de ocorrência da espécie;

- **Parâmetros de saída**:
  - **Map**: Classe que representa o mapa onde a espécie será projetada, ele encapsula uma imagem rasterizada e georeferenciada, representada por classes da API GDAL;
  - **AreaStats**: Guarda as probabilidades de ocorrência da espécie e gera uma estatística unificada de pontos e probabilidade de ocorrência;

A projeção do *openModeller* é basicamente um laço, onde para cada pixel, uma probabilidade é calculada baseada nas variáveis de projeção, e escrita em um mapa de saída. É importante ressaltar aqui, que durante a projeção, o *openModeller* ainda referência o algoritmo, isso faz com que o desempenho da projeção esteja diretamente ligada ao desempenho do algoritmo, implicando que as técnicas de paralelismo aplicadas na projeção devem levar em conta esse fator.

**Atividades Realizadas e Resultados Obtidos**

Uso de AOP para coleta de parâmetros de desempenho.

Através da AOP é possível interceptar um método de uma instância, e assim conseguir o seu tempo total de execução. Além disso, é possível obter outras métricas de utilização de recursos (como uso de CPU e memória) através da API WMI encontrada no Framework .NET.

A AOP foi desenvolvida para capturar os interesses comuns do sistema (*concerns*) que não são capturados pelas técnicas de programação tradicionais, como a OOP (Programação Orientada a Objetos). Um *concern* é um objeto, um conceito ou uma área de interesse.
Quando afeta múltiplos módulos de execução de um programa, chamamos de *crosscut*. AOP apresenta uma maneira de colocar o código destes *crosscuts* em um único lugar (por exemplo, uma única classe especializada).

Nesta pesquisa, foi usado um *advice* após cada método de cada componente. Desta maneira, pudemos ajustar um contador à zero antes da execução de algum método, iniciar o contador e parar a contagem com o retorno do método, atribuindo o resultado a uma variável e chamando um método específico no *aspect*.

Um *aspect* é um recipiente para todos os *pointcuts* e *advices* que executam um único *crosscut*; pode geralmente ser definido como uma classe, podendo ter métodos e campos além dos *pointcuts* e *advice*. Deste modo, para colher as métricas de desempenho, seria necessário apenas a criação de uma classe em AOP para interceptar os vários métodos presentes nas diversas classes do *openModeller*.

Em adição as métricas de tempo, a técnica de AOP usa a API WMI (*Windows Management Instrumentation*) para capturar outras métricas e estatísticas, como utilização de memória e processador, já que esta fornece acesso consistente às informações dentro de uma aplicação, possibilitando a descoberta da relação entre os componentes.

Levantamento dos componentes e de suas interfaces.

Através da organização do código do *openModeller* podemos verificar a seguinte separação em componentes:

- *libopenmodeller.dll*: biblioteca com as classes utilizadas para obtenção dos parâmetros de entrada, geração do modelo e projeção;
- *om_create*: responsável por iniciar a criação do modelo de distribuição;
- *om_project*: através do modelo cria a projeção em uma mapa em escala de cinza;
• *algoritmos*: implementam formas diferentes de cálculo estatístico para criação do modelo e da projeção (ex: Boclim, GARP, Minimum Distance).

![Solution Explorer](image)

**Figura 3. openModeller e seus componentes**

Como cada algoritmo consiste em um componente próprio, podendo ser removido ou adicionado ao sistema bastando implementar uma estrutura de classes pré-definida. Com isso o sistema permite que diferentes algoritmos usem os mesmos dados de entrada e o mesmo ambiente de funcionamento, possibilitando comparação entre resultados.
A classe que implementa o algoritmo deve apresentar em sua estrutura os seguintes requisitos para poder fazer parte do framework:

- Especificação dos metadados: através das constantes `parameters` e `metadata` que mantêm as informações (metadados) sobre os parâmetros a serem utilizados e sobre o próprio algoritmo;

- Método para inicialização: `initialize()`, responsável por verificar a validade dos parâmetros contidos nas classes `Algorithm` e `AlgorithmMetaData`, e por carregar os pontos de ocorrência e as variáveis ambientais armazenados nas classes `Occurrence` e `Environment` respectivamente;

- Métodos para geração do modelo;

- Método para cálculo da probabilidade.

Na construção do modelo o `openModeller` passa para o algoritmo os valores das variáveis ambientais para cada ponto de ocorrência da espécie à ser analisada (ex: 20º e 115mm de chuva – ver Figura 5). O modelo é gerado e apresentado através da classe `Model` e um arquivo .XML é criado com os resultados do modelo.
Figura 5. Construção do modelo

Na projeção o openModeller pergunta ao algoritmo a probabilidade de ocorrência da espécie para cada variável ambiente do mapa de projeção. Essas probabilidades são então armazenadas na classe AreaStats.

Figura 6. Geração da projeção

Cada algoritmo é implementado em uma DLL. O openModeller busca todas as DLLs disponibilizadas e as carrega ao sistema. A comunicação com o algoritmo selecionado através do arquivo de entrada é, então, realizada através de dois métodos públicos do algoritmo, são eles:

- `algorithmFactory()`: retorna um objeto da classe que implementa o algoritmo;
- `algorithmMetadata()`: devolve os metadados sobre o algoritmo;

Esses dois métodos servem de interface entre a biblioteca libopenmodeller_dll, responsável pela lógica de negócio do sistema, e os algoritmos.
Wrapper em C# e portabilidade para o COM+.

Um dos principais problemas em avaliar o desempenho de uma aplicação monolítica como o openModeller é predizer o desempenho de cada método à fim de identificar os gargalos. Para tentar resolver esse problema decidiu-se dividir a aplicação em componentes funcionais diferentes (já citados no item anterior) e hospedar então estes componentes em uma arquitetura distribuída que execute uma comunicação inter processos entre o processo principal e todos os componentes.

Assim escolheu-se hospedar os componentes usando a arquitetura de middleware do Microsoft COM+ (Component Object Model). O objetivo do COM+ é suportar o desenvolvimento de componentes que possam ser ativados dinamicamente e interagir entre si (TANENBAUM, 2002). O próprio COM+ é oferecido na forma de uma biblioteca que é ligada a um processo especializado para aceitar um volume elevado de chamadas inter-processos. Assim, do próprio programa principal, todos os componentes são chamados de uma maneira distribuída, independentemente de sua localização. Esta tecnologia, além de ser uma plataforma de middleware orientada a objetos para comunicação inter-processos, fornece funcionalidades ricas. Por exemplo, o COM+ apresenta o mecanismo de interception que é essencial para implementar o paradigma da AOP necessário para instrumentar o código dos componentes do openModeller.

Para transformar em componentes COM+ foi necessário criar um wrapper (casca) em C# para cada componentes já que o COM+ não suporta DLLs geradas a partir de código em C++ puro, como ocorre no caso do openModeller. Assim foi criada uma classe em C# para cada componente, que chama os métodos públicos do componente.
public class OM_BIOCLIM:ServicedComponent
{

    [DllImport ("om_bioclim.dll")]
    static extern System.IntPtr algorithmMetadata( );

    [DllImport ("om_bioclim.dll")]
    static extern System.IntPtr algorithmFactory( );

    public System.IntPtr algorithmMetadataNet( ){
        return algorithmMetadata( );
    }

    public System.IntPtr algorithmFactoryNet( ){
        return algorithmFactory( );
    }
}

Figura 7. Exemplo de wrapper em C#

Na Figura 7 é apresentado um exemplo de wrapper implementado para referenciar o componente om_biloclim. Como este componente é um dos algoritmos “plugáveis” (responsáveis pelos métodos de calculo estatístico para geração do modelo e projeção) existem apenas dois métodos públicos algorithmMetadata() e algorithmFactory(), assim foi necessário criar apenas dois métodos em C# para referenciá-los: algorithmMetadataNet() e algorithmFactoryNet().

Essas classes em C# foram então encapsuladas em COM+ para que se pudesse realizar através da AOP o interception de cada método de cada componente, ficando assim o sistema de maneira distribuída.
Figura 8. Componentes distribuídos no COM+

Arquitetura Implementada

Para coletar as métricas de desempenho do framework openModeller foi especificada uma arquitetura, procurando minimizar o impacto da instrumentação do código sobre as métricas coletadas. Deste modo, foi criado uma classe utilizando AOP para interceptar os métodos dos componentes do openModeller portados para o COM+.

Na classe de AOP, um método foi criado para gerar uma mensagem com a informação sobre o tempo de execução do método pertencente a algum componente COM+. Este método também instancia o WMI que monitora o consumo dos recursos computacionais. Outro método da classe de AOP é responsável por colocar a mensagem gerada numa fila de mensagens assíncronas através do Microsoft Message Queue (MSMQ). Outro processo é responsável por monitorar a fila do MSMQ,
capturar a mensagens e as persistir no banco de dados. Em paralelo com essa execução, o *openModeller* continua processando normalmente, já que as execuções da classe de AOP são chamadas num processo separado.

Um aplicativo chamado de *LogService* foi desenvolvido para rastrear as mensagens armazenadas em um a fila do MSMQ e para persistir numa base de dados para análise. O serviço processa continuamente a fila de mensagens, recolhendo-as em intervalos de 5 minutos. Esse intervalo pode ser configurado num arquivo XML. Assim, as mensagens geradas pela instrumentação de AOP são armazenadas na fila e permanecem lá enquanto o serviço é processado e as mensagens persistidas. Para evitar a disputa por recursos computacionais entre o processo do *openModeller* e o processo de coleta de métricas de desempenho, este último foi configurado para funcionar com menor prioridade do que o sistema *openModeller*.

A ilustração seguinte mostra o fluxo de trabalho da arquitetura implementada para permitir o recolhimento das evidências relacionadas ao processo de avaliação de desempenho. Para uma compreensão melhor do processo inteiro, uma breve explicação de cada estágio é fornecida abaixo.
Figura 9. A arquitetura e infra-estrutura projetados para o processo da avaliação de desempenho do *openModeller*.

O processo de avaliação de desempenho consiste de 5 estágios, descritos a seguir:

1. O usuário submete os dados de entrada ao componente principal do *openModeller*.
2. Esse componente principal, uma aplicação do console, invoca os componentes instalados no COM+, para construir o modelo e projeção. Como os componentes processam os dados de entrada fornecidos pelo usuário, estes componentes retornam o status do progresso da computação. O *status* é apresentado ao usuário em sua tela.
3. Cada método dos componentes hospedados no COM+ é interceptado e cada crosscut é criado pelo mecanismo de AOP em um processo diferente. As execuções da classe de AOP coletam as métricas e sincronizam-se com o WMI. Uma
mensagem é gerada com as evidências e emitida ao MSMQ para ser armazenada em uma fila.

4. O LogService, processa a fila de acordo com a configuração especificada e recupera as mensagens que foram armazenadas. Esse processamento é executado sempre na prioridade baixa, para evitar a interferência do processo de avaliação e para evitar o uso de recursos computacionais em excesso.

5. O processo continua executando e coletando as mensagens por tempo especificado na configuração. Depois desse tempo terminar, o LogService envia essas mensagens para o sistema de banco de dados para serem persistidas numa tabela apropriada. Depois de todas as mensagens salvas no banco, o LogService volta ao estado inativo.

Modelo preliminar de simulação.

A ideia principal deste modelo é permitir a simulação do sistema e analisar o seu desempenho através de uma infra-estrutura diferente de componentes; sendo uma importante ferramenta para a decisão de uma melhor distribuição dos componentes do openModeller e para identificar os gargalos possíveis do sistema.

Analisando os dados coletados e a implementação atual do openModeller foi possível pensar numa arquitetura distribuída que fizesse o sistema mais eficiente. Nessa arquitetura, o usuário não precisa instalar todo o openModeller, com a lib_openmodeller e bibliotecas auxiliares, a fim de construir e projetar o modelo. Somente a relação de usuário e um componente simples analisariam e verificariam a consistência dos argumentos necessários. Este componente simples também submeteria os dados de entrada necessários ao openModeller, que é agora um serviço remoto, hospedado em uma infra-estrutura dedicada. Esta infra-estrutura forneceria recursos computacionais mais poderosos do que os encontrados num computador pessoal comum.
Nessa arquitetura remota, são propostas duas camadas. A primeira camada consiste em um Web Server e um componente responsável por decidir qual componente do sistema deve ser invocado. Nesta camada ficariam também os serviços de segurança como autenticação e autorização, dependendo do escopo.

A segunda camada consistiria do núcleo do openModeller (lib_openmodeller) e dos componentes responsáveis por implementar os diferentes algoritmos de modelagem.

Figura 10. Arquitetura sugerida.

Baseada nessa arquitetura, foi criado na ferramenta de simulação Arena um modelo de filas para que junto com os dados coletados de desempenho da versão atual do openModeller seja possível simular e validar a arquitetura proposta.
Figura 11. Representação do modelo de simulação do sistema *openModeller*.

Também foi submetido e publicado um artigo na *Conferência 5th International Conference on Ecological Informatics* ISEI (www.isei5-conference.elsevier.com), organizado pela Elsevier em associação com a *International Society for Ecological Informatics* (ISEI), na cidade de Santa Bárbara – CA, EUA.

A ISEI tem por objetivo promover os conceitos relacionados à Informática Ecológica, organizando conferências internacionais bianuais, divulgando informações através de site na WEB e publicações no *Journal Ecological Informatics*. A ISEI pretende facilitar a formação de uma rede especializada de grupos de pesquisa, interdisciplinar em processamento de informação em ecossistemas, computação inspirada em biologia, visualização e gerenciamento de dados, bem como educação e treinamento em informática orientada em ecologia.

A participação na *5th International Conference on Ecological Informatics* ISEI, envolveu a apresentação oral do trabalho do aluno de Doutorado Jéferson Araújo, citado a seguir:

Conclusão

Em resumo o trabalho de pesquisa teve o objetivo de analisar o sistema openModeller atual e sugerir uma arquitetura distribuída de alto desempenho para o mesmo. Para isso foi criada uma infra-estrutura para coleta de métrica de desempenho, utilizando então AOP para que não houvesse impacto da infra-estrutura sobre os dados coletados.

Uma arquitetura distribuída foi proposta para a implementação do Framework openModeller, onde este estaria disponível através de Web Service e seus componentes em servidores de alto desempenho.

É sugerido como continuidade desse trabalho uma análise das métricas coletadas para identificação dos gargalos e dos componentes passíveis de paralelização. Além disso, com a análise dessas evidências será possível levantar a curva de distribuição do desempenho de cada componente pela carga aplicada ao sistema e assim realizar as simulações através do modelo de filas para verificar o comportamento da arquitetura proposta.

A apresentação do trabalho numa conferência sobre informática ecológica mostra a importância dessa pesquisa não só para o projeto openModeller, mas também para outros projetos ligados à ecologia, que muitas vezes fazem uso de uma grande quantidade de recursos computacionais, como o projeto financiado pela NSF (National Science Foundation – EUA) denominado SEEK (Science Environment for Ecological Knowledge) (PARTNERSHIP FOR BIODIVERSITY INFORMATICS, 2004). Esse mesmo trabalho realizado para o openModeller poderia ser realizado para outros sistemas, afim de obter um ganho de desempenho dessas aplicações.
Instituições Envoltidas

As instituições envolvidas no desenvolvimento do arcabouço openModeller são:

• CRIA (Centro de Referência em Informação Ambiental).

Trabalha com sistemas de informação de biodiversidade e está responsável pelo desenvolvimento e por disponibilizar o arcabouço na Internet, além de integrar os dados sobre as espécies com a ferramenta.

• INPE (Instituto de Pesquisas Espaciais)

Trabalha com pesquisa científica e aplicações tecnológicas em ciências atmosféricas. Neste projeto é responsável por integrar o openModeller a um Sistema de Informação Geográfico (SIG), denominado Terralib.

• Escola Politécnica da USP

Trabalha com pesquisa e ensino e está responsável por executar estudos que serão importantes para a implementação e aperfeiçoamento do openModeller, tais como a sua distribuição computacional através da web, o processamento computacional paralelo, desenvolvimento de novos algoritmos para modelagem, além da definição dos padrões e protocolos a serem utilizados.

• FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo)

Instituição financiadora do projeto.
Equipe do Projeto

- Mariana Ramos Franco
  Aluna de Graduação da Escola Politécnica da USP em Engenharia de Computação.

- Prof. Dr. Pedro Luiz Pizzigatti Corrêa
  Orientador do projeto.

- Prof. Dr. Antonio Mauro Saraiva
  Coordenador do grupo do openModeller na Escola Politécnica da USP e co-orientador do projeto.

- Profa. Dra. Liria Sato
  Responsável pela infra-estrutura de grid/cluster do projeto.

- Jeferson Martin de Araújo
  Pesquisador do Laboratório de Automação Agrícola da Escola Politécnica da USP, na área de pesquisa relacionada à Modelagem e Análise de Desempenho de Sistemas Computacionais.
Bibliografia

BRESSAN, G. Abordagem Sistemática de Modelagem e Análise de Desempenho. LARC-PCS/EPUSP, 2003


SIQUEIRA, M. F. Uso de modelagem de nicho fundamental na avaliação do padrão de distribuição geográfica de espécies vegetais. Tese de doutorado. Departamento de Engenharia Ambiental da Universidade de São Carlos. São Carlos, São Paulo, 2005.


LOWY, J. COM+ e .NET. Editora Campus, 2002.

PRADO, D. Usando o Arena em Simulação, Vol 3. INDG Tecnologia e Serviços Ltda. 2004


Anexo

ARTIGO

A Framework for Species Distribution Modelling: A performance evaluation approach
Assinaturas

Bolsista: Mariana Ramos Franco

Orientador: Prof. Dr. Pedro Luiz Pizzigatti Corrêa

São Paulo, 09 de março de 2007.
Annex 05

OpenModeller

How to insert an algorithm

by Danilo J. S. Bellini

Thursday, 22 / September / 2006
Working environment

This study was done in a Acer Aspire 3004WLCi laptop running Kubuntu Linux 3.03 (Kernel 2.6.1526386). This manual is made for developers who want to create a new algorithm in openModeller, and suppose that you already have installed it from scratch.

This is the source dir for all openModeller parts (except the GUI):

~/dev/cpp/om/src

For the algorithms, all files are in:

~/dev/cpp/om/src/algorithms

All openModeller (oM) main data structures classes are in:

~/dev/cpp/om/src/openmodeller

The root directory for oM, where you have “autogen.sh” configurator and the root Makefile:

~/dev/cpp/om/

How to compile with oM

Makefile changes

If we want to add a new algorithm, so we need to understand what happens in the algorithms' directory. A file named “Makefile.am” have a source to a makefile generator, and inside of it we can see something like:

\[ \text{SUBDIRS} = \text{garp dg_garp best_subsets} \]
\[ \text{OMSRCDIR} = \text{@top_srcdir@/src} \]
\[ \text{OMLIB} = $(OMSRCDIR)/openmodeller/libopenmodeller.la \]
\[ \text{plugindir} = $(libdir)/openmodeller \]

SUBDIRS have the name of all subdirectories inside algorithms directory we have another Makefile to be opened. It's useful when we have huge algorithms. The next 2 lines shouldn't be changed since they are a link between main oM sources and all the algorithms sources. The last line also shouldn't be changed, but it tells us that inside our default lib dir there is a subdir called “openmodeller” where all algorithms are in (/usr/local/lib/openmodeller, for example). So before learning how to create or install a new algorithm, we learnt how to delete one (just remove the files that have the mnemonic of the algorithm you want to remove).
Next lines are specific for each algorithm. To add a new one all we need is to forget everything that is written there and add the following lines at the end of the file (XXX is any mnemonic you want to put in your algorithm, with any size):

\[
\text{SUBDIRS} += \text{<sources subdir> <sources subdir> ...} \\
\text{plugin\_LTLIBRARIES} += \text{libomXXX.la} \\
\text{libomXXX\_la\_SOURCES} = \text{<source file> <source file> ...} \\
\text{libomXXX\_la\_LDFLAGS} = \text{noundefined} \\
\text{libomXXX\_la\_LIBADD} = $(OMLIB) \\
\text{libomXXX\_la\_CPPFLAGS} = I$(OMSRCDIR)
\]

If you see the ignored lines, you will see that they are just the same thing you see here, but not with blocks fully individualized to one algorithm. The first line (SUBDIRS) is only needed when you have subdirs to make separately (with “+=” we broke its definition to more than one line). The second one tells the lib file that your algorithm will be. The third one tells which are the source files of your algorithm (discussed ahead), including the headers. For simple codes, you probably won’t need to change last lines. They are: LDFLAGS line, that have flags to the linker; LIBADD one, that have the libs it need; and CPPFLAGS, with flags to the C++ compiler.

Now we know that there are only one file we have to change to add an algorithm: the “Makefile.am”. The other files now are algorithmspecific so we’ll have to create them. Let’s make an example: first of all we’ll edit the discussed file and add the following lines:

\[
\text{plugin\_LTLIBRARIES} += \text{libomdummy.la} \\
\text{libomdummy\_la\_SOURCES} = \text{dummy.cpp dummy.h} \\
\text{libomdummy\_la\_LDFLAGS} = \text{noundefined} \\
\text{libomdummy\_la\_LIBADD} = $(OMLIB) \\
\text{libomdummy\_la\_CPPFLAGS} = I$(OMSRCDIR)
\]

**First algorithm files**

Let’s create our “dummy.cpp” and it’s header “dummy.h”. In this moment, it’s better to use the ones below even we don’t know what they do. This is the first file, dummy.h:

```c
//
// Dummy algorithm
//
// Description: An algorithm made to teach how an algorithm is made.
//
// Author: Danilo J. S. Bellini <danilo.estagio@gmail.com>
// Copyright: See COPYING file that comes with this distribution
// Date: 20060818
//
#ifdef _DUMMYH_
#define _DUMMYH_
#endif

#include <openmodeller/om.hh>
```
class DummyAlgorithm : public AlgorithmImpl{
public:
    DummyAlgorithm();
    int initialize();
    Scalar getValue(const Sample& x) const;
};
#endif

This is the smallest code you can make to an clean algorithm header source file. Now the dummy.cpp file:

//
// Dummy algorithm
//
// Description: An algorithm made to teach how an algorithm is made.
//
// Author: Danilo J. S. Bellini <danilo.estagio@gmail.com>
// Copyright: See COPYING file that comes with this distribution
// Date: 2006-08-18
//
#include "dummy.h"

//
// METADATA
//
#define NUM_PARAM 1
#define TESTPAR "TestParameter"
#define TPARMIN 0.0
#define TPARMAX 5.0

static AlgParamMetadata parameters[NUM_PARAM] = { // Parameters
    { // First parameter
        TESTPAR, // Id
        "Dummy", // Name
        "Real", // Type
        "XXX", // Overview
        "YYY", // Description
        1, // Not zero if the parameter has lower limit
        TPARMIN, // Parameter's lower limit
        1, // Not zero if the parameter has upper limit
        TPARMAX, // Parameter's upper limit
        0.5 // Parameter's typical (default) value
    },
};

static AlgMetadata metadata = { // General metadata
    "DummyAlgorithm", // Id
    "Dummy Algorithm", // Name
    "1", // Version
    "Dummy.", // Overview
    "Algorithm made to teach how an algorithm is made.", // Description
    "Danilo J. S. Bellini", // Algorithm author
    "None", // Bibliography
    "Danilo J. S. Bellini", // Code author
    "danilo.estagio@gmail.com", // Code author's contact
    0, // Does not accept categorical data
    0, // Does not need (pseudo)absence points
NUM_PARAM, parameters // Algorithm's parameters

// LINK TO OM

// Needed code to link this algorithm to oM
OM_ALG_DLL_EXPORT
AlgorithmImpl *algorithmFactory(){
    return new DummyAlgorithm(); // Create an instance of your algorithm's class
}

OM_ALG_DLL_EXPORT
AlgMetadata const *algorithmMetadata(){
    return &metadata;
}

// ALGORITHM CONSTRUCTOR/DESTRUCTOR

// Constructor for the algorithm class
DummyAlgorithm::DummyAlgorithm() : AlgorithmImpl(&metadata){
}

// ALGORITHM NEEDED METHODS

// Initialize the model
int DummyAlgorithm::initialize(){
    return 0;
}

// Returns the occurrence probability at the environment conditions x
Scalar DummyAlgorithm::getValue(const Sample& x)const{
    return 1.0;
}

This second file isn't the smallest .cpp file we can create, but it's near to.

**Compiling**

To test it, we'll have to write these 2 files inside oM algorithm directory and compile them, so go to oM root directory (in my case ~/dev/cpp/om/) and type something like:

```
./autogen.sh --enable-python--enable-soap--prefix="/usr/local/"
makemake
```

The first line is needed, but it's only needed in first time when you compile oM with anything modified in a "Makefile.am". This call update the makefiles in all directories, because we modified the "Makefile.am" in the algorithms directory, but the parameters to autogen.sh may be different to you. The second line
compile oM (and our new algorithm). The last line installs oM (and our algorithm).

**Testing**

Now you can test it with oM (we'll use “om_console” for now...you can use any other interface if you want). Actually, the algorithm doesn't do anything (so dummy is a good name for it), but at least we can see it in oM and call:

```
danilo@hdllaptop:~/dev/cpp/om/examples$ om_console request.txt
```

openModeller version 0.4

Choose an algorithm between:

0] Bioclim
1] Bioclim distance
2] Climate Space Model
3] Climate Space Model KaiserGutman
4] GARP with best subsets DesktopGARP implementation
5] Distance to average
6] Dummy Algorithm
7] Minimum distance
8] GARP DesktopGARP implementation
9] Quit

Option: 6

> Algorithm used: Dummy Algorithm

Algorithm made to teach how an algorithm is made.

* Parameter: Dummy

XXX:

Dummy $\geq 0.00000$

Dummy $\leq 5.00000$

Enter with value [0.5]:
Creating the model
Exception occurred
Message is Algorithm could not be initialized.

As you can see, our algorithm didn't do anything, because it can't be initialized (we'll see that below), but the first step is done: we added the algorithm there!

**Algorithm link with oM**

**Metadata**

Now, let's understand what we made. An algorithm receive parameters from oM (inputs, besides the environmental layers), such as the size of something
or how many times something have to be done. They must be explained in a static/global vector of AlgParamMetadata and it's called “parameters” (see dummy.cpp), but you can rename it (I don't suggest you to do that, even for the other names we'll see). NUM_PARAM was defined with the amount of parameters we have and TESTPAR is the name I had chosen to the parameter's identification for oM (This one should be changed in your code, because your parameter probably won't have a mnemonic “test”). As these values are used more than one time in the program, it's better to keep them defined that way. All parameters are specified that way, and the comments are helpful because they say which are the metadata we need to put in each position. Remember: DON'T remove any of these, even the value is somehow null, because the order and amount of them are important. Each parameter is separated by commas from the others, so the block called “First parameter” could be copied below itself to create the second, third, etc. ones.

If you want an algorithm without parameters, you should do this (but don't forget to define NUM_PARAM as 0):

```cpp
static AlgParamMetadata *parameters = 0;
```

After that, we specified the “unuseful” details of our algorithm, things such as its name and author: this is put in a static/global “AlgMetadata” data type, called “metadata”. The rules here are almost the same of the rules for each parameter. Don't change last line inside metadata's block and things you don't know what they are.

**Dynamic link**

Next 8 lines are needed to make the connection between this algorithm and oM dynamically. The only thing you can change in there is the name “DummyAlgorithm” that is the name of the algorithm class as we'll see.

**Smallest header explanation**

Now let's see dummy.h (never forget to include algorithm's header in the .cpp file). It's really small, with only 10 lines containing code. A real algorithm must have a bigger header file, this one only have needed data to compile. All its code are inside an “#ifndef” C++ directive, that avoid including this file more than once. For your algorithm you will need to change the name _DUMMYH_ to a name you want (in #ifdef and #define lines), but I suggest to use the real file name uppercase without the dot and between underscores symbols.

Your algorithm header file must include om.hh to work. Also, your algorithm class, for now called DummyAlgorithm, must be inherited from the class AlgorithmImpl (included with om.hh). The constructor must be created explicitly and the methods initialize and getValue are needed to compilation.
Basic algorithm initialization

Now you know how to insert an algorithm in oM and how to specify its parameters metadata, so the next step is to make this dummy algorithm work (remember it can't be initialized?)

Why it can't be initialized? That happens because our initialize method returns 0 (false) always, let's change some things there, for example let's make a new initialize method in dummy.cpp, that can return 1 (true). To add some code there, let's get the parameter from the user (oM do that for us, just use the method “getParameter” from the class we inherited) and test if it was set properly (for example, 1.3 isn't an integer value, so an integer parameter can't have this value):

```cpp
int DummyAlgorithm::initialize(){
    if(!getParameter(TESTPAR,&testPar)){
        g_log.error(1, "Parameter " TESTPAR " not set properly.\n");
        return 0;
    }
    return 1;
}
```

To be fully convinced that everything is right, you should add some code to verify if your parameter obeys the limit values (maximum and minimum); for example you can use the limit value instead of the wrong value. Our parameter “TESTPAR” will be written in “testPar” variable, that you should declare in the DummyAlgorithm class, better as private, so add these lines just before the “};” line in dummy.h:

```cpp
private:
    Scalar testPar;
```

Scalar datatype

Why data type is “Scalar”? Scalar is the defined somewhere in oM as double, and it's the distance unit to oM, so that's why our “Real” type will be written in a Scalar data type. Don't redefine it, this can be found somewhere in code (if you want to see, try to type in oM main source files directory, for example ~/dev/cpp/om/src/openmodeller/, this: cat * | grep “Scalar”):

```cpp
typedef double Scalar;
```

Testing the first working algorithm

Now we don't need to worry about parameters and metadata, they all are finished, and our “black empty box” algorithm get the parameters as inputs (don't worry, we'll talk about the maps later), some constant outputs (metadata) and the link with oM is done. Let’s test our modified dummy
algorithm, so go to the oM algorithms source directory to recompile and reinstall it:

```
make
sudo make install
```

Well, we don't need to call “autogen.sh” because we didn't modify anything that needs that, so everything is done, all we need is to recompile because we modified our algorithm. Also, we don't need to recompile and reinstall everything, only what we changed (that's why we came to the algorithms directory, and “make” only compile what was changed).

Let's see what happens now:

```
Enter with value [0.5]:
Creating the model
Model creation: 100.0000%
Finished Creating Model
Map creation: 100.0000%
Model statistics
Accuracy: 100.00%
Omission error: 0.00%
Percentage of cells predicted present: 100.00%
Total number of cells: 261822
```

That's all oM needs to run your algorithm, so let's study what it says: see our getValue method, it always return 1.0, that means that it ignore the maps and always says “100% of chance to have a specie here”, that's why our result map is fully white. We'll talk about getValue's parameter after we'll see what is the Sample class.

### A more sophisticated dummy algorithm

#### New header

Let's modify our dummy.h to have this class:

```
class DummyAlgorithm : public AlgorithmImpl{

  public: // All methods here are inherited from AlgorithmImpl
    DummyAlgorithm(); // Constructor, don't have init algorithm routines
    ~DummyAlgorithm(); // Destructor

    int initialize(); // Called by oM to initialize the algorithm
    int done() const { return _done; } // Tell oM when the algorithm finished its work
    int needNormalization(Scalar *min, Scalar *max) const; // Normalize all data to [min,max]
```
Scalar getValue(const Sample& x) const; // Returns the occurrence probability

private:

// Common use attributes
bool _initialized; // Flag to indicate that the algorithm was initialized.
bool _done;     // Flag to indicate when the work is finished;
int _layerCount; // Amount of layers used (dimension of environmental space)
int _presenceCount; // Amount of presence points
int _absenceCount; // Amount of absence points
std::vector<Sample> _presencePoints_env;
std::vector<Coord> _presencePoints_x, _presencePoints_y;

// Parameters
Scalar testPar;
};

What's new here?

The “done” method always returns the attribute done, for an iterative algorithm it's really useful since the iterative algorithm can say when its work has finished, that's when oM finishes the algorithm iteration. As we don't have an iterative algorithm, _done will be assigned to true in the initialize method (it doesn't have to be assigned to false in the constructor, but must be assigned to false in the initialize method of an iterative algorithm).

The needNormalization method isn't needed. Each layer (input) gives a value to each point of the map, this value can be normalized to be in [min,max] using this method. By default, it must return 1 (or it doen't exist, because you don't want to normalize the data). We'll talk more about it after you get knowledge about what the value of the map means.

Next lines shows some common attributes we have. _initialized and _done should exist in all algorithms, since they are useful flags for the destructor and for the “done” method. The _initialized is set to true only at the end of the “initialize” method, and set to false in constructor.

The attribute _layerCount will have the amount of environmental layers the algorithm receive, _presenceCount will have the amount of presence points and _absenceCount will have the amount of absence points. The 3 vectors after these attributes will have some data from the presence points.

The other new lines will be discussed later.
**New dummy.cpp**

For example, put this in dummy.cpp, changing everything after the “link with oM” section:

```cpp
// // ALGORITHM CONSTRUCTOR/DESTRUCTOR //

// Constructor for the algorithm class
DummyAlgorithm::DummyAlgorithm() : AlgorithmImpl(&metadata){
    _initialized = false;
}

// Destructor for the algorithm class
DummyAlgorithm::~DummyAlgorithm()
{
    if(_initialized){
        /**********************************/
        /* Call all needed destructors here */
        /**********************************/
    }
}

// // ALGORITHM GENERIC METHODS (virtual AlgorithmImpl methods) //

// Initialize the algorithm
int DummyAlgorithm::initialize(){
    // Test the parameters’ data types
    if(!getParameter(TESTPAR,&testPar)){
        g_log.error(1, "Parameter " TESTPAR " wasn’t set properly.\n");
        return 0;
    }

    // Impose limits to the parameters, if somehow the user don’t obey
    if (testPar>TPARMAX) testPar = TPARMAX;
    else if(testPar<TPARMIN) testPar = TPARMIN;

    // Initialize some commonuse attributes
    _layerCount = _samp->numIndependent();
    _presenceCount = _samp->numPresence();
    _absenceCount = _samp->numAbsence();

    // Load all environmental data of presence points and put into
    // _presencePoints_env, and the same for X and Y coordinates
    if(_presenceCount == 0){
        g_log.error(1, "There is no presence point.\n");
        return 0;
    }

    OccurrencesPtr presences = _samp->getPresences();
    for (int i = 0 ; i < _presenceCount ; i++){
        _presencePoints_env.push_back((*presences)[i]->environment());
        _presencePoints_x.push_back ((*presences)[i]->x());
        _presencePoints_y.push_back ((*presences)[i]->y());
    }
```

```cpp
```
/* Call needed constructors and other initialization operations here */

// _done = true; // Needed for not iterative algorithms
// _done = false; // Needed for iterative algorithms

_initialized = true;

return 1; // There was no problem in initialization

// Normalize all data to [min, max]
int DummyAlgorithm::needNormalization(Scalar *min, Scalar *max) const{
    *min = TPARMIN;
    *max = TPARMAX;
    return 1;
}

// Returns the occurrence probability
Scalar DummyAlgorithm::getValue(const Sample& x) const{
    return 1.0;
}

New initialization “ritual”

The constructor have only what was said, and the destructor tests this value because somehow the algorithm can be destroyed before it calls the “initialize” method, so we can't destroy not created objects (the reason for the existence of this attribute).

All methods above are inherit from AlgorithmImpl. Our initialize method now imposes limits to testPar, so we can use it even if the oM user interface allows bad inputs. The 3 attributes “_layerCount”, “_presenceCount” and “_absenceCount” are the result of 3 methods from a “sampler object” called “_samp”, that we can use here. I suggest you to do everything you want with this object in this method (initialize) and forget that it exists after that (of course, for a better code, you should create private methods to get the data from _samp if you see that the code using it will be huge, and call them in this initialization method).

What's _samp object?

What's _samp object? Better, what's the SamplerPtr / SamplerImpl class? What do we need to know about them?

Sample class

We can't know what they are without the data structures, so I'll tell you about the “Sample” data, that is basic for environmental data. A Sample is a “customized version of a std::vector<Scalar>”. For one point (with coordinates X and Y) we'll have a Scalar value for each layer, so, for each point, we can
have a Sample with all the data from the layers. Example in a 3x3 map with 2 layers:

**Layer 0**

<table>
<thead>
<tr>
<th>Y/X</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>14</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>

**Layer 1**

<table>
<thead>
<tr>
<th>Y/X</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>125</td>
<td>114</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>113</td>
<td>85</td>
<td>52</td>
</tr>
<tr>
<td>3</td>
<td>67</td>
<td>45</td>
<td>18</td>
</tr>
</tbody>
</table>

We'll have a Sample object (don't confuse Sample with Sample!) for each one of the 9 points, they will be:

- \( Y=1 ; X=1 ; \text{Sample} = (11;123) \)
- \( Y=1 ; X=2 ; \text{Sample} = (14;114) \)
- \( Y=1 ; X=3 ; \text{Sample} = (56;70) \)

... 

The samples can be assigned to other as “Sample newSample = x;” if your algorithm need a backup. To acess randomly any data from a sample you can use the "[" simbols, that they'll return a Scalar. There are some pointwise operations allowed ("+=" , "-=" , "+=" , "/=" , "&=" and "|="): a pointwise operation is an operation that is done for each pair of data, that means that a += will sum the 2 vectors as we see in maths, but something strange happens with multiplication and division. "+=" returns the minimum value between the 2 vectors and "+=" returns the maximum value between the 2 vectors, for example:

- \( \text{Vector1} := (1 ; 15 ; 14 ; 17) \)
- \( \text{Vector2} := (18 ; 14 ; 12 ; 20) \)

**After Vector1 & Vector2:**

\( \text{Vector1} := (1 ; 14 ; 12 ; 17) \)

**After Vector1 | Vector2 (The 2 first Vectors of the example!):**

\( \text{Vector1} := (18 ; 15 ; 14 ; 20) \)

The operators "+=" , "-=" , "+=" and "+=" are also defined to have a Scalar “argument” as it were a Sample with the same value in all the vector. There
are some methods you can use from the Sample class (iterator is a Scalar*
typedef internal to this class, and const_iterator too, but as a constant):

- **Resize the vector to the new size passed as the parameter**
  ```cpp
  void resize(std::size_t size);
  ```

- **Returns the current size of the vector**
  ```cpp
  std::size_t size() const;
  ```

- **Iterative form of using the vector**
  ```cpp
  // Example (implementation of sqr method, explained below):
  //
  // for(iterator it = begin(); it != end() ; it++)
  //   *it = (*it) * (*it);
  //
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  ```

- **Compare 2 Sample instances: “this” and the constant parameter. Return**
  true only if they have the same size and the same values in the same order.
  ```cpp
  bool equals(const Sample&) const;
  ```

- **Calling “sqr(X);” have the same effect of “X *= X”;**
  ```cpp
  Sample& sqr();
  ```

- **Calcs the square root for each value in the vector and write it in itself, then**
  return
  ```cpp
  Sample& sqrt();
  ```

- **Return the norm of the vector “this”. By maths, the norm is the square root of**
  the dot product of a vector by itself
  ```cpp
  Scalar norm() const;
  ```

- **Calcs the dot product of this with rhs, that means the sum of the values of a**
  pointwise multiplication between “this” and “rhs”
  ```cpp
  Scalar dotProduct(const Sample& rhs) const;
  ```

A Sample is an point in a multidimensional world (one dimension for each layer). For the example given above, we have a bidimensional world (2 layers). For one dimension we have 1 layer (e.g. Layer 0), so imagine it as our axis, in that example we have points from “4” to “56” in this axis. Adding another dimension (e.g. Layer 1), we'll have a new axis, creating an angle of 90 degrees with all the other ones we've seen before. This time it's a 2D plan, where the points are given by environmental data and not spacial relative distances. Algorithms for oM can't use X and Y coordinates, the reason will be explained later (in the Environmental classes section).
**Sampler classes**

Think that the data from oM inputs arrive to the algorithms as a wizard summons a magical being: we don't need to worry about what is done with original data before we get them, all we have to know is how to get them. The _samp, as any "sampler object", have 3 kinds of data: environmental data (all layers), presence points (all) and absence points. As _samp is a SamplerPtr instance and not a SamplerImpl one, we'll use “>” instead of “.” to call the methods (that's not hard to understand, and happens to all “XXXPtr” classes that are “pointers” to “XXXImpl” classes), and the SamplerImpl methods we can use are:

- To get all data of one of the 3 parts of an Sampler class. Explanation about data types is done below
  - EnvironmentPtr getEnvironment();
  - ConstEnvironmentPtr getEnvironment() const;
  - OccurrencesPtr getPresences();
  - ConstOccurrencesPtr getPresences() const;
  - OccurrencesPtr getAbsences();

- Small info this class can give
  - Amount of independent variables, easier, amount of layers used
    - dimension of environmental space)
  - int numIndependent() const;
  - Amount of dependent variables
  - int numDependent() const;
  - Amount of presence points / occurrence points
  - int numPresence() const;
  - Amount of absence points / localities with no occurrence
  - int numAbsence() const;

- Returns a random occurrence, in the first case that's an absence or a presence point (probability = 0.5) and if there's any real absence point then it doesn't return an pseudoabsence point (a point that that isn't a presence point). The other ones are easier.
  - ConstOccurrencePtr getOneSample( ) const;
  - ConstOccurrencePtr getPresence() const;
  - ConstOccurrencePtr getAbsence() const;
  - ConstOccurrencePtr getPseudoAbsence() const;

- It's a test for categorial layers. If the layer “i” is categorical it returns notzero. Don't worry with categorical layers for now.
  - int isCategorical( int i );
**Occurrence and Occurrences classes**

Probably you can't understand the Sampler classes without understanding what is an OccurrencePtr ("pointer class" to OccurrenceImpl class), because the presences and absences are OccurrencePtr instances. Warning: OccurrencesImpl and OccurrenceImpl are different things! OccurrencesImpl is about the same of Sample: it's an OccurrencePtr vector (not a Scalar one). OccurrenceImpl doesn't have many methods, and all we need to know is that an occurrence have 2 coordinates (x and y) and the environmental Sample for this point. The other data it have is the uncertainty of this point and attributes. You can access the locality information with the methods:

```cpp
Coord x() const; // Probably you won't need these ones
Coord y() const;
Scalar error() const;
Scalar abundance() const;
Sample const & attributes() const;
Sample const & environment() const; // This one you certainly will use
```

The coordinate unit “Coord” is defined as double somewhere in code (do the same you did with Scalar to find it). These are only “get” methods for X and Y coordinates, for the uncertainty (error method), the attribute “abundance”, others attributes you want and the environment Sample. There's only one more method you might want to use is the “bool hasEnvironment() const;”, that verify if the environment Sample exist (probably you won't need this too). The other 3 methods it have are used to manually normalize that Sample, what we really doesn't need to know.

For your algorithm probably all you want to know is the “environment” method, since it gives the Sample of the occurrence. An example of an algorithm that uses coordinates is one that receives the coordinates of some points as parameters to give different weights to them, manually.

Now, for the “customized std::vector<OccurrencePtr>” called OccurrencesImpl, we have some more info: the occurrences' name (commonly the species name) and the coordinate system found in the OccurrenceImpl points. It also have the random access by "[]" and the iterative access by “begin” and “end” methods, but this time the iterator is a std::vector<OccurrencePtr>, and we also can use const_iterator. This class have the methods:

- Returns a copy of a OccurrencesImpl
  ```cpp
  OccurrencesImpl* clone() const;
  ```

- Calls std::vector>::reserve()
  ```cpp
  void reserve(int estimate);
  ```

- Gets the name and coordinate system strings (almost unuseful for us)
  ```cpp
  char const * name() const;
  char const * coordSystem() const;
  ```
• Returns true if there are environmental data
  bool hasEnvironment() const;

• Returns the amount of environmental layers
  int dimension() const;

• Add a new occurrence (useful for iterative algorithms that supposes new
  occurrences where have high probability of occurrence). The first one is
  deprecated.
  void createOccurrence(Coord longitude, Coord latitude,
                        Scalar error, Scalar abundance,
                        int num_attributes = 0, Scalar *attributes = 0,
                        int num_env = 0, Scalar *env = 0);
  void createOccurrence(Coord longitude, Coord latitude,
                        Scalar error, Scalar abundance,
                        std::vector<double> attributes,
                        std::vector<double> env);
  void insert(const OccurrencePtr&);

• Erase the occurrence chosen by the iterator
  iterator erase(const iterator& it);

• Return the amount of occurrences exist in this object (size of the vector)
  int numOccurrences() const;

• Return a OccurrenceImpl chosen at random
  ConstOccurrencePtr getRandom() const;

• Calls std::vector<>::empty()
  bool isEmpty() const;

• “Push back” the entire “source” vector (calls the std::vector<>::pushback
  method)
  void appendFrom(const OccurrencesPtr& source);

• Print verbosely all contents of “this”.
  void print(char *msg="") const;

Environment classes

Commonly you won’t need to use the info directly from EnvironmentImpl, the
most important text here is this paragraph, ignore the methods, unless you
have a really strange algorithm that needs to call them. Here lies the maps, so
maximum and minimum values of the coordinates X and Y are here,
information about a mask is here too, and other useful information. When the
OpenModeller object calls the AlgorithmImpl::getValue, it sends the return of
the EnvironmentImpl::get(x,y) method, so that’s why we don’t have the X and
Y coordinates of the point that called OpenModeller::getValue(x,y).
Information about normalization is also here, but for normalization of
environmental data you only have to use the AlgorithmImpl::needNormalization
method, don’t forget, because you don’t know if there are several places that
have to be normalized. The methods of the EnvironmentImpl class are:
int numLayers() const;
int isCategorical(int i);
Sample get(Coord x, Coord y) const;
Sample getRandom( Coord *x = 0, Coord *y = 0 ) const;
int check(Coord x, Coord y) const;
int getRegion(Coord *xmin, Coord *ymin, Coord *xmax, Coord *ymax) const;
int getExtremes(Sample* min, Sample* max) const;

This class have several methods we don't have to know, these ones are enough. The first one is obvious, the second was discussed before but we're not concerned about categorical data for now, the third is the get method seen in last paragraph. The method getRandom is the same of the get one, but the values x and y are get randomly inside the mask. The method check returns false (zero) when the point given isn't inside the mask. The method getRegion tells all limits of the coordinates as a rectangle (remember the mask don't need to be a rectangle).

The method getExtremes gives us 2 Samples with the minimum and maximum value of each layer. It can be used by algorithms that don't normalize data, for example.

Sophisticated dummy algorithm explanation

At this time you are able to understood by yourself what the dummy algorithm does, but it's hard to understand without examples, so let's understood what it says

int _layerCount; // Amount of layers used (dimension of environmental space)
int _presenceCount; // Amount of presence points
int _absenceCount; // Amount of absence points
std::vector<Sample> _presencePoints_env;
std::vector<Coord> _presencePoints_x, _presencePoints_y;

The amount values are got from _samp object:

// Initialize some common-use attributes
_layerCount = _samp->numIndependent();
_presenceCount = _samp->numPresence();
_absenceCount = _samp->numAbsence();

The vectors of _presencePoints can be get using the OccurrencesPtr object from _samp->getPresences():

// Load all environmental data of presence points and put into _presencePoints_env, and the same for X and Y coordinates
if(_presenceCount == 0){
    g_log.error(1, "There is no presence point.
return 0;
OccurrencesPtr presences = _samp->getPresences();
for(int i = 0 ; i < _presenceCount ; i++){
    _presencePoints_env.push_back((*presences)[i]->environment());
    _presencePoints_x.push_back ((*presences)[i]->x());
    _presencePoints_y.push_back ((*presences)[i]->y());
}

The X and Y values are almost unuseful, so you can ignore them and create only one vector with all environmental data from presence points, called _presencePoints (without the _env). When you see “(*presences)[i] environment()”, doesn't get mad, the 2 “contents” done is because we have a pointer of pointer (remember that the operator “[]” to be applied in OccurrencesPtr need to have the “*”, since it’s a pointer). The ith element of the OccurrencesImpl (*presences) is an OccurrencePtr, so we'll use the “>” operator to call its methods. The methods used get the environment Sample and the coordinates and push into the _presencePoint as a stack. See that now we can use _presencePoints anywhere in the algorithm, which is easier do understand (it's a vector). To access the ith Sample you can use _presencePoints[i] and the jth Scalar of this Sample, you can use _presencePoints[i][j], as used in common C matrix implementation. The same can be done with absence points.

The needNormalization can be easily understood now:

// Normalize all data to [min,max]
int DummyAlgorithm::needNormalization(Scalar *min, Scalar *max) const{
    *min = TPARMIN;
    *max = TPARMAX;
    return 1;
}

The values in the layers will now be limited to a minimum of TPARMIN and a maximum of TPARMAX. The offsets and scales to do that are done automatically by oM after you return 1.

AlgorithmImpl, daddy of my algorithm

All we need to know about AlgorithmImpl is what we can inherit (no consideration with its daddy...what a bad algorithm!). From now we know the methods:

int initialize();
int done() const;
int needNormalization(Scalar *min, Scalar *max) const;
Scalar getValue(const Sample& x) const;
But for an iterative algorithm, we'll need more methods. Here's all other methods you can inherit from this class:

```cpp
int supportsModelProjection() const
int iterate();
int finalize();
int getConvergence(Scalar const *val) const;
float getProgress() const;
Model getModel() const;
void _getConfiguration(ConfigurationPtr&) const;
void _setConfiguration(const ConstConfigurationPtr&);
```

To create an iterative algorithm, the first thing you should do is to inherit the “iterate” method and change the _done attribute to false in the “initialize” method. The iterate method should return 1, and it considers that you are training stepbystep until _done goes to true or when the iterate method returns 0 (telling that something wrong happened). After that the “finalize” method is called, so you can finish the training phase in other place to avoid things like a “if(done())” inside the iterate method; this method should return 1 too and doesn't need to exist, even in an iterative algorithm (to be more complete, it's executed even without the “iterate” method, after it sees that _done is true).

The method getProgress is useful to give to the user information about how much is done. By default, it returns 0 while _done is false and 1 when _done becomes true. All you have to do is create an attribute (why not “float _progress;” ?), initialize it and increment in your iterate method (that can be complicated for some algorithms), and add this method inline, returning the new attribute.

Other method is getConvergence that returns the convergence value at the moment. By default it always return zero, but if you have this information in your algorithm you can put there.

The other 3 methods...
openModeller

High level vision to algorithms, data structures and algorithm sequence

Danilo J. S. Bellini

openModeller - what is?

openModeller is an sophisticated image processor for ecological data

It's capable to read these data from several images (different files) and text files, containing the configuration data and the points of presence/absence
openModeller - inputs

So we have 3 kind of data as inputs to openModeller:

- configuration
  - occurrence points
    - maps

openModeller – Sample

All the data from maps are passed to the algorithm by instances of the class “Sample”

A “Sample” is a simple vector of “Scalar” values.

Each point (with coordinates X and Y) have one Sample assigned to it, with one value from each layer (map)
openModeller - layers

All environmental layers (maps) have an information that somehow openModeller sends in a numeric scale and can be automatically normalized to a certain interval.

For our example, each layer gives values from 0 to 50. Let's see what they are...

Data representation

Imagine we have 2 layers: each layer can be imagined as a matrix of values, one for each possible point

<table>
<thead>
<tr>
<th>Y=1</th>
<th>X=1</th>
<th>X=2</th>
<th>X=3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>35</td>
<td>17</td>
</tr>
<tr>
<td>Y=2</td>
<td>15</td>
<td>42</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y=1</th>
<th>X=1</th>
<th>X=2</th>
<th>X=3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12</td>
<td>5</td>
<td>27</td>
</tr>
<tr>
<td>Y=2</td>
<td>9</td>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>
An algorithm can't see all coordinates. That happens because an algorithm doesn't need that information. Algorithms are based in environmental data.

That's why we have to understand better what environmental data means to the users of oM, with precision.

When we have spatial geographic data, we can put the points in a graphic easily. The axis here may have longitude and latitude values, or X and Y converted values.
Data representation

But for each coordinate we have values from the layers

Data representation

From this data, we can imagine other axis! These are the axis we are going to use in our algorithms
Data representation

Now we can define exactly what “environmental distance” means: it's the distance in this new space! As we can have much more layers than 3, visualize this can be really difficult.

Sample

Resuming the Sample topic, a Sample is simply the “new coordinates vector” we created, then in the images showed before, we have (1,12) and (42,9) as examples of occurrence Samples. A sample exist in all places, so (15,9) is the sample to the place (X=1,Y=2), that isn't an occurrence point.
**Sampler**

A SamplerImpl object contains 3 data:

- An OccurrencesPtr for presences
- An OccurrencesPtr for absences
- An EnvironmentalPtr for all layers

---

**Occurrences**

An OccurrencesImpl object is a vector like Sample, but to another data type: OccurrenceImpl

An OccurrenceImpl have data from his coordinates (almost useless) and the Sample of that place
Environment

An EnvironmentImpl have all data from maps. These data can be accessed by coordinates X and Y, but is internal to openmodeller when calling the algorithm.

To code!

getValue method!

This method from the algorithm is called by om.getValue (that have the coordinates X and Y) and send to the algorithm only the Sample assigned to the point the “om” object knows. So there's no way to use X and Y coordinates in algorithm.
Order of execution

1 – The algorithm is constructed;
2 – all data is normalized if “needNormalization” method exists;
3 – oM calls “initialize” method;
4 – oM calls “iterate” method until the “done” method returns true;
5 – oM calls “finalize” method;
6 - method getValue is called for each place in the output map.

PS: “getProgress” can be used to return the iterative progress, if you want to show.
Annex 07

openModeller Class Documentation

SamplerImpl Class Reference

#include <Sampler.hh>

Detailed Description

Base class to create samplers of environmental variables and occurrence data. Each derived class can implement different behaviors, such as proportional sampling according to the distribution of occurrence data, disproportional sampling regarding presence and absence data, etc.

Public Member Functions

- SamplerImpl ()
- SamplerImpl (const EnvironmentPtr &env, const OccurrencesPtr &presences, const OccurrencesPtr &absence)
- ~SamplerImpl ()
- EnvironmentPtr getEnvironment ()
- ConstEnvironmentPtr getEnvironment () const
- OccurrencesPtr getPresences ()
- ConstOccurrencesPtr getPresences () const
- OccurrencesPtr getAbsences ()
- ConstOccurrencesPtr getAbsences () const
- void getMinMax (Sample *min, Sample *max) const
- void normalize (bool use_norm, const Sample &offsets, const Sample &scales)
- int numIndependent () const
- int numDependent () const
- int numPresence () const
- int numAbsence () const
- ConstOccurrencePtr getOneSample () const
- ConstOccurrencePtr getPresence () const
- ConstOccurrencePtr getAbsence () const
- ConstOccurrencePtr getPseudoAbsence () const
- int isCategorical (int i)
- void environmentallyUnique ()
- void spatiallyUnique ()
- ConfigurationPtr getConfiguration () const
- void setConfiguration (const ConstConfigurationPtr &)

Private Member Functions

- ConstOccurrencePtr getRandomOccurrence (const OccurrencesPtr &occur) const
- void setEnvironmentInOccurrences ()
- void environmentallyUnique (OccurrencesPtr &occurrencesPtr, const char *type)
- void spatiallyUnique (OccurrencesPtr &occurrencesPtr, const char *type)
Private Attributes

- OccurrencesPtr _presence
- OccurrencesPtr _absence
- EnvironmentPtr _env

Friends

- class ReferenceCountedPointer<SamplerImpl>
- class ReferenceCountedPointer<const SamplerImpl>
- SamplerPtr createSampler (const ConstConfigurationPtr &)

Constructor & Destructor Documentation

SamplerImpl::SamplerImpl ()

SamplerImpl::SamplerImpl (const EnvironmentPtr & env, const OccurrencesPtr & presences, const OccurrencesPtr & absence)

SamplerImpl::~SamplerImpl ()

Definition at line 99 of file Sampler.cpp.

Member Function Documentation

void SamplerImpl::environmentallyUnique (Occurrence sPtr & occurrencesPtr, const char * type) [private]

Definition at line 385 of file Sampler.cpp.

References _env.void SamplerImpl::environmentallyUnique ()

Remove sample duplicates accross the environment (presences and absences are treated separately). After erasing a point, the remaining one increases the abundance by one.

Definition at line 372 of file Sampler.cpp.

References _absence, and _presence.ConstOccurrencePtr SamplerImpl::getAbsence () const [inline]

Get one Absence point

Definition at line 146 of file Sampler.hh.

Referenced by getOneSample().ConstOccurrencesPtr SamplerImpl::getAbsences () const [inline]
Definition at line 95 of file Sampler.hh.
SamplerImpl::getAbsences () [inline]

Definition at line 94 of file Sampler.hh.
SamplerImpl::getConfiguration () const

Definition at line 89 of file Sampler.hh.
SamplerImpl::getEnvironment () const [inline]

Definition at line 88 of file Sampler.hh.
SamplerImpl::getMinMax (Sample * min, Sample * max) const

Indicate that all non categorical variable layers must be normalized according to the interval [min, max]. Returns through offsets, scales the normalization parameters.

Definition at line 212 of file Sampler.cpp.

References _absence, _env, and _presence.
SamplerImpl::getEnvironment () const [inline]

Definition at line 140 of file Sampler.hh.
ConstOccurrencePtr SamplerImpl::getPresence () const

Get one Presence point

Definition at line 140 of file Sampler.hh.

References _absence, _presence, getAbsence(), getPresence(), and getPseudoAbsence().

Parameters:
    independent Filled in with values of the independent variables of the sample.

Returns:
    Zero if got an absence or pseudo-absence point and not zero if got a presence point.

Definition at line 304 of file Sampler.cpp.

References _absence, _presence, getAbsence(), getPresence(), and getPseudoAbsence().

ConstOccurrencePtr SamplerImpl::getPresence () const [inline]
Referenced by `getOneSample().ConstOccurrencesPtr SamplerImpl::getPresences () const` [inline]

Definition at line 92 of file Sampler.hh.
`OccurrencesPtr SamplerImpl::getPresences ()` [inline]

Definition at line 91 of file Sampler.hh.
`ConstOccurrencePtr SamplerImpl::getPseudoAbsence () const`

Get one pseudoAbsence point

Definition at line 330 of file Sampler.cpp.
References _env, and numDependent().

Referenced by `getOneSample()`.

`ConstOccurrencePtr SamplerImpl::getRandomOccurrence (const OccurrencesPtr & occur)` const [private]

Definition at line 508 of file Sampler.cpp.
`int SamplerImpl::isCategorical (int i)`

Returns 1 if i-th variable is categorical, otherwise returns 0.

Definition at line 353 of file Sampler.cpp.
References _env.
`void SamplerImpl::normalize (bool use_norm, const Sample & offsets, const Sample & scales)`

Set specific normalization parameters

Definition at line 236 of file Sampler.cpp.
References _absence, _env, _presence, and setEnvironmentInOccurrences().

`int SamplerImpl::numAbsence () const`

Number of absences (localities with no occurrence).

Definition at line 297 of file Sampler.cpp.
References _absence.

`int SamplerImpl::numDependent () const`

Number of dependent variables (attributes of the occurred thing). These are the variables to be modelled.

Definition at line 279 of file Sampler.cpp.
References _presence.
SamplerImpl::numIndependent () const
Number of independent variables (environmental variables).

Definition at line 260 of file Sampler.cpp.

SamplerImpl::numPresence () const
Number of presences (occurrence points).

Definition at line 288 of file Sampler.cpp.

SamplerImpl::setConfiguration (const ConstConfigurationPtr &)

Definition at line 155 of file Sampler.cpp.

SamplerImpl::setEnvironmentInOccurrences ()

Definition at line 107 of file Sampler.cpp.

SamplerImpl::spatiallyUnique (OccurrencesPtr & occurrencesPtr, const char * type) [private]

Definition at line 447 of file Sampler.cpp.

SamplerImpl::spatiallyUnique ()
Remove sample duplicates across geographic space (presences and absences are treated separately). After erasing a point, the remaining one increases the abundance by one. Uniqueness is considered for row/col pairs defined in the input mask. If mask is undefined, use first layer as a mask.

Definition at line 434 of file Sampler.cpp.

createSampler (const ConstConfigurationPtr &) [friend]
Definition at line 61 of file Sampler.cpp. friend class ReferenceCountedPointer<const SamplerImpl> [friend]

Definition at line 77 of file Sampler.hh. friend class ReferenceCountedPointer<SamplerImpl> [friend]

Definition at line 76 of file Sampler.hh.

**Member Data Documentation**

**OccurrencesPtr** SamplerImpl::*_absence** [private]

Definition at line 191 of file Sampler.hh.

Referenced by environmentallyUnique(), getConfiguration(), getMinMax(), getOneSample(), normalize(), numAbsence(), setConfiguration(), setEnvironmentInOccurrences(), and spatiallyUnique().

**EnvironmentPtr** SamplerImpl::*_env** [private]

Definition at line 192 of file Sampler.hh.

Referenced by environmentallyUnique(), getConfiguration(), getMinMax(), getPseudoAbsence(), isCategorical(), normalize(), numIndependent(), setConfiguration(), setEnvironmentInOccurrences(), and spatiallyUnique().

**OccurrencesPtr** SamplerImpl::*_presence** [private]

Definition at line 190 of file Sampler.hh.

Referenced by environmentallyUnique(), getConfiguration(), getMinMax(), getOneSample(), normalize(), numDependent(), numIndependent(), numPresence(), setConfiguration(), setEnvironmentInOccurrences(), and spatiallyUnique().

**Functions**

- **SamplerPtr createSampler** (const EnvironmentPtr &env, const OccurrencesPtr &presence, const OccurrencesPtr &absence)
- **SamplerPtr createSampler** (const ConstConfigurationPtr &config)
- static void **splitOccurrences** (OccurrencesPtr &occurrences, OccurrencesPtr &trainOccurrences, OccurrencesPtr &testOccurrences, double propTrain)
- void **splitSampler** (const SamplerPtr &orig, SamplerPtr *train, SamplerPtr *test, double propTrain)
Function Documentation

**SamplerPtr createSampler (const ConstConfigurationPtr & config)**

Definition at line 61 of file Sampler.cpp.

References SamplerPtr.SamplerPtr createSampler (const EnvironmentPtr & env, const OccurrencesPtr & presence, const OccurrencesPtr & absence = OccurrencesPtr())

Factory method that creates a Sampler based on Environment and Occurrences objects

Definition at line 54 of file Sampler.cpp.

Referenced by splitSampler().

**void splitSampler (const SamplerPtr & orig, SamplerPtr * train, SamplerPtr * test, double propTrain)**

Splits data points into 2 new samplers

Parameters:
- *orig* Original sampler to split
- *train* Pointer to train sampler to be returned
- *test* Pointer to test sampler to be returned
- *propTrain* Percentage of points to go to train sampler

Definition at line 575 of file Sampler.cpp.

References splitOccurrences().

**Typedefs**

- typedef ReferenceCountedPointer< const SamplerImpl > ConstSamplerPtr

**Functions**

- dllexp void splitSampler (const SamplerPtr & orig, SamplerPtr * train, SamplerPtr * test, double propTrain)
- SamplerPtr createSampler (const EnvironmentPtr & env, const OccurrencesPtr & presence, const OccurrencesPtr & absence = OccurrencesPtr())
- SamplerPtr createSampler (const ConstConfigurationPtr & config)

**Variables**

- typedef ReferenceCountedPointer< SamplerImpl > SamplerPtr
Typedef Documentation

typedef ReferenceCountedPointer<const SamplerImpl> ConstSamplerPtr

Definition at line 47 of file Sampler.hh.

Function Documentation

SamplerPtr createSampler (const ConstConfigurationPtr & config)

Definition at line 61 of file Sampler.cpp.

References SamplerPtr SamplerPtr createSampler (const EnvironmentPtr & env, const OccurrencesPtr & presence, const OccurrencesPtr & absence = OccurrencesPtr())

Factory method that creates a Sampler based on Environment and Occurrences objects

Definition at line 54 of file Sampler.cpp.

References SamplerPtr dllexp void splitSampler (const SamplerPtr & orig, SamplerPtr * train, SamplerPtr * test, double propTrain)

Splits data points into 2 new samplers

Parameters:
  orig Original sampler to split
  train Pointer to train sampler to be returned
  test Pointer to test sampler to be returned
  propTrain Percentage of points to go to train sampler

Definition at line 575 of file Sampler.cpp.

References splitOccurrences().

Variable Documentation

class typedef ReferenceCountedPointer<SamplerImpl> SamplerPtr

Definition at line 45 of file Sampler.hh.

Referenced by createSampler()
Annex 08

openModeller Class Documentation

ConfusionMatrix Class Reference

#include <ConfusionMatrix.hh>

Detailed Description

Class ConfusionMatrix tabulates the proportions of types of successes and errors generated by the model. The matrix is represented as following:

```
+---------------------------------+
|           |  Actual Values      |
+ Predicted |---------------------+
| Values    | Presence | Absence |
+---------------------------------+
| Presence  |     a     |    b    |
+---------------------------------+
| Absence   |     c     |    d    |
```

Values a and d are correct predictions, while b and c are Type I and Type II errors.

Public Member Functions

- `ConfusionMatrix` (Scalar predictionThreshold=0.5)
- `~ConfusionMatrix` ()
- void `reset` (Scalar predictionThreshold=0.5)
- void `calculate` (const EnvironmentPtr &env, const Model &model, const OccurrencesPtr &presences, const OccurrencesPtr &absences=OccurrencesPtr())
- void `calculate` (const Model &model, const SamplerPtr &sampler)
- int `getValue` (Scalar predictionValue, Scalar actualValue)
- double `getAccuracy` () const
- double `getOmissionError` () const
- double `getCommissionError` () const
- bool `ready` ()
- ConfigurationPtr `getConfiguration` () const

Private Attributes

- int _confMatrix [2][2]
- Scalar _predictionThreshold
- bool _ready
Constructor & Destructor Documentation

ConfusionMatrix::ConfusionMatrix (Scalar predictionThreshold = 0.5)

Default constructor.

**Parameters:**

`predictionThreshold` Threshold for prediction values, above which a prediction value means a predicted presence.

Definition at line 41 of file ConfusionMatrix.cpp.

References reset().

ConfusionMatrix::~ConfusionMatrix ()

Destructor.

Definition at line 47 of file ConfusionMatrix.cpp.

Member Function Documentation

void ConfusionMatrix::calculate (const Model & model, const SamplerPtr & sampler)

Calculate confusion matrix based on an abstract Sampler object

**Parameters:**

`model` Model object to be evaluated

`Sampler` Pointer to a Sampler object that will provide data for evaluation

Definition at line 164 of file ConfusionMatrix.cpp.

References calculate().

void ConfusionMatrix::calculate (const EnvironmentPtr & env, const Model & model, const OccurrencesPtr & presences, const OccurrencesPtr & absences = OccurrencesPtr())

Calculate confusion matrix based on model and sampled data from environment and occurrences objects.

**Parameters:**

`env` Pointer to Environment object containing the environmental data to be used in calculation

`model` Model object to be evaluated

`presences` Pointer to an Occurrences object storing the presence points being evaluated

`absences` Pointer to an Occurrences object storing the absence points being evaluated

Definition at line 67 of file ConfusionMatrix.cpp.

References _confMatrix, _predictionThreshold, _ready, and reset().
double ConfusionMatrix::getAccuracy () const
Returns the accuracy of the model, which corresponds to the sum of successes (points predicted correctly) divided by all available points. I.e., accuracy = (a + d) / (a + b + c + d)

Definition at line 183 of file ConfusionMatrix.cpp.
References _confMatrix, and _ready.

Referenced by getConfiguration() double ConfusionMatrix::getCommissionError () const
Returns the commission error of the model, which corresponds to the sum of absence points predicted present divided by all available absence points. I.e., omission = b / (b + d)

Definition at line 196 of file ConfusionMatrix.cpp.
References _confMatrix, and _ready.

Referenced by getConfiguration() ConfigurationPtr ConfusionMatrix::getConfiguration () const
Serialize the confusion matrix

Definition at line 216 of file ConfusionMatrix.cpp.
References getAccuracy(), getCommissionError(), and getOmissionError() double ConfusionMatrix::getOmissionError () const
Returns the omission error of the model, which corresponds to the sum of presence points predicted absent divided by all available presence points. I.e., omission = c / (a + c)

Definition at line 206 of file ConfusionMatrix.cpp.
References _confMatrix, and _ready.

Referenced by getConfiguration() int ConfusionMatrix::getValue (Scalar predictionValue, Scalar actualValue)
Returns a value from the confusion matrix.

Parameters:

  predictionValue Prediction value corresponding to the row selected in the matrix
  actualValue Actual value corresponding to the column selected in the matrix

Definition at line 171 of file ConfusionMatrix.cpp.

References _confMatrix, and _predictionThreshold bool ConfusionMatrix::ready () [inline]
Check whether the confusion matrix has been calculated already
Definition at line 136 of file ConfusionMatrix.hh.

void ConfusionMatrix::reset (Scalar predictionThreshold = 0.5)

Resets area counters.

**Parameters:**

*predictionThreshold* New threshold for prediction values, above which a prediction value means a predicted presence.

Definition at line 52 of file ConfusionMatrix.cpp.
References _confMatrix, _predictionThreshold, and _ready.
Referenced by calculate(), and ConfusionMatrix().

---

**Member Data Documentation**

int ConfusionMatrix::_confMatrix[2][2] [private]

Definition at line 156 of file ConfusionMatrix.hh.

Referenced by calculate(), getAccuracy(), getCommissionError(), getOmissionError(), getValue(), and reset().

Scalar ConfusionMatrix::_predictionThreshold [private]

Definition at line 158 of file ConfusionMatrix.hh.

Referenced by calculate(), getValue(), and reset().

bool ConfusionMatrix::_ready [private]

Definition at line 160 of file ConfusionMatrix.hh.

Referenced by calculate(), getAccuracy(), getCommissionError(), getOmissionError(), and reset().
Annex 09

openModeller Class Documentation

AreaStats Class Reference

```
#include <AreaStats.hh>
```

Public Member Functions

- `AreaStats` (Scalar predictionThreshold=0.5)
- `AreaStats` (const `AreaStats` *areaStats)
- `~AreaStats`()
- `void reset (Scalar predictionThreshold=0.5)`
- `void addPrediction (Scalar predictionValue)`
- `void addNonPrediction ()`
- `int getTotalArea () const`
- `int getAreaPredictedPresent () const`
- `int getAreaPredictedAbsent () const`
- `int getAreaNotPredicted () const`
- `Scalar getPredictionThreshold () const`
- `ConfigurationPtr getConfiguration () const`

Private Attributes

- `int _areaTotal`
- `int _areaPredPresent`
- `int _areaPredAbsent`
- `int _areaNotPredicted`
- `Scalar _predictionThreshold`

Constructor & Destructor Documentation

`AreaStats::AreaStats (Scalar predictionThreshold = 0.5)`

Default constructor.

**Parameters:**

`predictionThreshold` Threshold for prediction values, above which a prediction value means a predicted presence.

Definition at line 33 of file AreaStats.cpp.

References reset().

`AreaStats::AreaStats (const AreaStats * areaStats)`

Copy constructor.

**Parameters:**

`areaStats` Pointer to an AreaStats object to be copied.
Definition at line 38 of file AreaStats.cpp.
AreaStats::~AreaStats ()

Destructor.

Definition at line 46 of file AreaStats.cpp.

Member Function Documentation

void AreaStats::addNonPrediction ()
Count a cell where the prediction doesn't apply.

Definition at line 67 of file AreaStats.cpp.

References _areaNotPredicted, and _areaTotal.

void AreaStats::addPrediction (Scalar predictionValue)
Count another cell based on its prediction value.

Parameters:

predictionValue Value of the prediction in the current cell.

Definition at line 57 of file AreaStats.cpp.

References _areaPredAbsent, _areaPredPresent, _areaTotal, and _predictionThreshold.

int AreaStats::getAreaNotPredicted () const [inline]
Returns total number of cells where prediction doesn't apply.

Definition at line 94 of file AreaStats.hh.

int AreaStats::getAreaPredictedAbsent () const [inline]
Returns total number of cell predicted absent.

Definition at line 89 of file AreaStats.hh.

int AreaStats::getAreaPredictedPresent () const [inline]
Returns total number of cell predicted present.

Definition at line 84 of file AreaStats.hh.

ConfigurationPtr AreaStats::getConfiguration () const
Serialize the area stats

Definition at line 75 of file AreaStats.cpp.
References _areaPredPresent, _areaTotal, and _predictionThreshold.

```cpp
Scalar AreaStats::getPredictionThreshold () const [inline]
Returns the prediction threshold.
```

**Definition at line 99 of file AreaStats.hh.**

```cpp
int AreaStats::getTotalArea () const [inline]
Returns total number of cells counted.
```

**Definition at line 79 of file AreaStats.hh.**

```cpp
void AreaStats::reset (Scalar predictionThreshold = 0.5)
Resets area counters.

**Parameters:**
- `predictionThreshold` New threshold for prediction values, above which a prediction value means a predicted presence.
```

Definition at line 50 of file AreaStats.cpp.
References _areaNotPredicted, _areaPredAbsent, _areaPredPresent, _areaTotal, and _predictionThreshold.
Referenced by AreaStats().

---

**Member Data Documentation**

```cpp
int AreaStats::_areaNotPredicted [private]
Definition at line 110 of file AreaStats.hh.
```

Referenced by addNonPrediction(), and reset().

```cpp
int AreaStats::_areaPredAbsent [private]
Definition at line 109 of file AreaStats.hh.
```

Referenced by addPrediction(), and reset().

```cpp
int AreaStats::_areaPredPresent [private]
Definition at line 108 of file AreaStats.hh.
```

Referenced by addPrediction(), getConfiguration(), and reset().

```cpp
int AreaStats::_areaTotal [private]
Definition at line 107 of file AreaStats.hh.
```
Referenced by addNonPrediction(), addPrediction(), getConfiguration(), and reset().

Scalar AreaStats::_predictionThreshold [private]

Definition at line 112 of file AreaStats.hh.

Referenced by addPrediction(), getConfiguration(), and reset()
Annex 10

Towards and Adaptive implementation of Genetic Algorithm

César Bravo, João José Neto, Fabiana Soares Santana
Escola Politécnica da Universidade de São Paulo
lapus@usp.br; joao.jose@poli.usp.br; fabiana.santana@gmail.com

Abstract

In this paper, a tutorial on how to execute genetic algorithms using adaptive decision tables is presented. An adaptive decision table is a decision table able to modify the rules defining it in execution time. As an concept proof, we applied the proposed method to the GARP (Genetic Algorithm for Rule Production). GARP is a genetic algorithm that allows inferring the presence of a species according to the values of environmental variables and species presence points in a given geographic area; it proceeds correlating the presence data of the species with environmental variables in the sampled area until a model were identified. In this context, usually known as Ecological Niche Modelling, one of the main problems is to identify which systems parameters are able to result in a better performance and to generate an adequate model for the species geographic distribution. In this case, the adaptive techniques may imply in a better evaluation of the context and, in the experiments executed during this research, it was possible to show that the new implementation does not affect the performance of the simulated genetic algorithms. This approach may result in a methodology for adoption and software implementation of adaptive decision tables in order to solve problems related to the use of genetic algorithms, since the new defined operations, adaptCrossover and adaptMutation, are common to all genetic algorithms. The referred GARP algorithm is implemented and available at the openModeller framework, and future improvements may include the adaptGARP implementation, the GARP algorithm that implements adaptCrossover and adaptMutation for application in biodiversity researches.

1. Introduction

In this paper two very strong recent trends - adaptivity and evolutionary programming - are put together in an application related to environmental modeling. This approach is suitable in the practice of programming complex systems featuring artificial intelligence capabilities, such as grammatical inference, computer learning and autonomous dynamically changeable behavior.

The starting point for this experience is the modeling system openModeler, http://openmodeller.sourceforge.net/, intended to infer a probability distribution of a species, in a given geographical area, from data about the species occurrence and environmental variables in that area.

The openModeller system includes several alternative algorithms to do the inference and, among them, we choose GARP genetic algorithm [6] to ours experiments. GARP is a genetic algorithm which takes as parameters information about the presence of a species in a geographical area and also information about the values of several environmental variables in the same geographical area and, with this data, produce a set of rules which allow to infer the presence of the given specie in another geographical area according to the values of the considered environmental variables. We claim any genetic algorithm can be executed by an adaptive decision table [1]; an adaptive decision table is a decision table provided with a set of adaptive functions able to add or remove the rules on defining the subjacent decision table. To show a concept proof of our claim, we had substitute the original implementation of the GARP algorithm with the AdaptGARP proposed in this paper and some experiments had been made in order to check impact on the performance. Our final intent is providing OpenModeller with a new version of GARP, with adaptive implementation of the genetic Crossover and Mutation operators.

2. Adaptive devices

A In this work we have implemented adaptive models that follow slightly modified version of the formulation defined in [2]. We briefly present the essentials of this formulation, and the reader is addressed to the reference for details.

Non-adaptive devices are formal devices whose behavior is defined as a static set of rules. This kind of devices may have their operation enhanced by adding to them an
adaptive layer that associates each device's rule to a set of adaptive actions.

Adaptive actions specify the changes to be applied to the device's rule set by using primitive editing operators, which allow applying inspection, deletions and additions to the set of rules defining the device.

In this way, the adaptive layer performs all operations that are needed for dynamically modifying the set of rules defining device's operation.

Such improved devices are called adaptive devices and the original devices from which they are obtained are said to be their corresponding subjacent non-adaptive devices.

There is no restriction to the nature of subjacent devices, so one may obtain adaptive devices from virtually any kind of abstraction defined as set of rules, e.g. automata, grammars, decision tables, etc.

3. Mapping Genetic Algorithms into Adaptive Decision Tables

In this work, the subjacent abstraction to be used is the decision table, whose corresponding adaptive devices are called adaptive decision tables.

A decision table ([1], [3], [4] and [5]) is a table encoding a set of rules in columns other than the first, letting the first column to designate a number of Condition Rows (on top of the table) and a set of Actions Rows on the bottom of the table. Starting at an initial configuration, the decision table operates checking the valid conditions against the values defined in the column rules. When a condition is found to be true for a given rule, then all the marked action for this rule are executed.

<table>
<thead>
<tr>
<th>Table 1 Decision Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition rows</td>
</tr>
<tr>
<td>c_1</td>
</tr>
<tr>
<td>c_2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>c_n</td>
</tr>
<tr>
<td>Actions</td>
</tr>
<tr>
<td>a_1</td>
</tr>
<tr>
<td>a_2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>z_m</td>
</tr>
</tbody>
</table>

An adaptive version for this abstraction is obtained by adding to an existing non-adaptive table a number of further lines which encode, for each column representing a single rule, the (parametric) calls to adaptive functions associated to the execution of that particular rule.

Whenever a rule in the table is applied, the associated adaptive functions are invoked, and their corresponding collateral effects change the current set of rules according to the adaptive operators performed by the adaptive function.

Our strategy in this work consist of encoding decision-taking rules as rules of an adaptive decision table, and modeling the changes in behavior as adaptive actions over the rules encoded in the decision table.

In order to be useful, the resulting adaptive decision table must replicate the behavior of the original genetic algorithm.

Once the programming of the mapping law is determined, any genetic algorithm-based program may be converted to an equivalent adaptive transition table-driven version.

In this work, both version were compared regarding the number of iterations as a function of the fitness function precision.

3.1. Adaptive Decision Table Format

As we already pointed out, to create an adaptive decision table from a decision table, we need to add some adaptive functions to the definition of the decision table. As defined in [2], these adaptive functions are placed in a set of rows under the Actions Rows of the decision table. The Table 2 shows the general setup of an adaptive decision table.

<table>
<thead>
<tr>
<th>Table 2 Adaptive Decision Table Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 k+1 R_1 R_2 ... R_m</td>
</tr>
<tr>
<td>Condition rows</td>
</tr>
<tr>
<td>c_1</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>c_n</td>
</tr>
<tr>
<td>Actions</td>
</tr>
<tr>
<td>a_1</td>
</tr>
<tr>
<td>a_2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>z_m</td>
</tr>
</tbody>
</table>

3.2. Adaptive Decision Table operation

To operate an adaptive decision table, first the status of the system is checked against the combinations of conditions stated in each of the rules encoded in the table. If no rule matches the current status, then no action is
executed at all. Otherwise, if a single rule matches the current status, then we have a deterministic choice, so the matching rule is selected to be applied. Finally, if more than one rule matches the current status, then we face a non-deterministic situation. Consequently, all those rules are to be applied in parallel. In practice, parallelism may be simulated, e.g. by some exhaustive backtracking strategy. The selected rule is then applied by executing the set of all actions indicated with a Boolean value True in the columns of the rule corresponding to action rows. Once the selected rule has been applied, the decision table gets ready to be used again.

4. Mapping Genetic Algorithms into Adaptive Decision Tables

This section is dedicated to describe an Adaptive implementation for the GARP algorithm [6] and its Crossover and Mutation operators, as implemented in the OmenModeller system http://omenmodeller.sourceforge.net.

In order to do this, the rule set will describe an individual, at a specific step of the genetic algorithm, in a format suitable for specifying those adaptive versions of the classical genetic operators.

Adaptive Crossover and Adaptive Mutation will act as rules manipulators over other rules stated in this new format, and they are invoked by the main algorithm—indeed, the genetic schema—as subroutines, replacing the non-adaptive versions. After describing suitable data structure for use with adaptive genetic operators, Adaptive Crossover and Adaptive Mutation are described. In the sequence, these operators are called AdaptCrossover and AdaptMutation.

4.1. Data structures

First of all, data structures must be designed in order to apply the general technique described in [2], which allow obtaining an adaptive device from any rule-driven non-adaptive device. The Crossover genetic operator takes two individuals and interchanges genetic information between them at the genetic level, producing a new individual; in the rules of the GARP algorithm, this handling of information operates on the limits of the intervals, defined within the rules.

Such technique requires that the rules of the non-adaptive device be encoded as a decision table. Then it is necessary to design suitable adaptive functions for manipulating the rules encoded in the decision table. These adaptive functions are encoded as part of the adaptive decision table, this way resulting in a compact representation of the genetic operators as part of the adaptive decision table and representing the dynamic change of the rule set.

In order to implement the GARP algorithm’s rules, we follow the format defined in [2]. As an example, consider $h$ interval rules, associated to $k$ environmental variables, as depicted in Table 3.

<table>
<thead>
<tr>
<th>$R_i$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>$A_{11}$</td>
<td>$A_{21}$</td>
<td>$A_{31}$</td>
</tr>
<tr>
<td>$x_2$</td>
<td>$B_{12}$</td>
<td>$B_{22}$</td>
<td>$B_{32}$</td>
</tr>
<tr>
<td>$x_3$</td>
<td>$C_{13}$</td>
<td>$C_{23}$</td>
<td>$C_{33}$</td>
</tr>
<tr>
<td>$x_4$</td>
<td>$D_{14}$</td>
<td>$D_{24}$</td>
<td>$D_{34}$</td>
</tr>
<tr>
<td>$x_5$</td>
<td>$E_{15}$</td>
<td>$E_{25}$</td>
<td>$E_{35}$</td>
</tr>
</tbody>
</table>

Assume that interval rules are PRESENCE rules. In GARP usual codification they are read as:

If $X_1 \in \{A_{11}, B_{11}\}$ AND $X_2 \in \{A_{12}, B_{12}\}$ AND ... $X_k \in \{A_{1k}, B_{1k}\}$ THEN PRESENT

The meaning of a rule $R_j$, where $1 \leq j \leq h$, is that if each environmental variable $X_p$, where $1 \leq i \leq k$, then the species is present in the associated localization.

That is the main data structure for the adaptive genetic operators described in the following subsections; the number of columns of the Table 3 will change as a side-effect of AdaptCrossover’s operator execution. The AdaptMutation operator will operate over some rule (column) in the table by changing the limits of its intervals.

We need to take some decisions on how to handle the rules; indeed, we refer to any rule by its index on the table, e.g. its column’s index. Since in the rule-based adaptive device the number of rules may change, we will consider that set of rules as a dynamic array which maintain the sequential numbering of its columns, so avoiding to renumber the columns in the table. To specify the adaptive functions we use some auxiliary functions described below

$$U(x,y) = \begin{cases} 0, & \text{if } x < y \\ 1, & \text{if } x \geq y \end{cases}$$

The $rand(a,b)$ function produces a pseudo random integer number between 1 and $n$, while $rand(a,b)$ produces a pseudo random integer number between $a$ and $b$, when they are integers and produces a pseudo random float number between $a$ and $b$ when $a$ and $b$ are float.

4.2 Adaptive implementation of Crossover operator

The Crossover genetic operator takes two individuals and interchanges genetic information between them, at the genetic level, producing a new individual; in the rules of the
The GARP algorithm, this handling of information operates on the limits of the intervals, defined within the rules.

The Table 4 specifies the adaptive function AdapCrossover in the format of the decision tables as described in [2].

<table>
<thead>
<tr>
<th>Table 4 Adaptive Decision Table for AdapCrossover</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
</tr>
<tr>
<td>x_{12}</td>
</tr>
<tr>
<td>x_{13}</td>
</tr>
<tr>
<td>x_{22}</td>
</tr>
<tr>
<td>x_{23}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>x_{k2}</td>
</tr>
<tr>
<td>x_{k3}</td>
</tr>
</tbody>
</table>

The cell on the first row of the first column is used to identify the function's name, leaving unused any other cells on that row; the second column of rows from second to fifth are used to specify the parameters of AdapCrossover; the parameters i and j are integer numbers representing the indexes of the rules to be crossed; parameters p and q are integer numbers that specify the sections of the intervals being inherited by the new individual resulting from the crossover between rule i and rule j. The remaining of the rows specify conditions the rule must follow.

This adaptive function adds a new row to the table of the rules; this addition of new rules is expressed by the symbol "+" at the header of the third column, encoding one of the three elementary actions that may operate on the rules defining the adaptive device. Another elementary action is the "-" which removes rules. We use it in the specification of the adaptive version of the Mutation operator, in section 4.3. As specified in Table 4, the AdapCrossover operator creates a new rule for which all intervals with index less than p or greater or equal than q, will inherit from rule R_k, and the intervals with index greater or equal than p and less than q, will be inherited from the rule R_i. This can be resumed as:

\[
x_{i} \geq \begin{cases} 
    A_{i}, & \text{if } h < p \\
    A_{k}, & \text{if } p \leq h < q \\
    A_{g}, & \text{if } h \geq q 
\end{cases}
\]

and

\[
x_{k} \geq \begin{cases} 
    A_{i}, & \text{if } r < g \\
    A_{k}, & \text{if } p \leq h < q \\
    A_{k}, & \text{if } h \geq q 
\end{cases}
\]

for all \( i \leq h \leq k \).

4.3 Adaptive implementation of Mutation operator

The Mutation genetic operator takes one individual and modifies its genetic information, at the genetic level, producing a mutation of the original individual, in the case of the rules of the GARP algorithm, this operation alters the limits of the intervals defined within the rules.

Table 5 specifies the adaptive function AdapMutation as a decision table following [2]; the cell in the first row of the first column is used to identify the function's name only, leaving unused any other cells in that row; the second column in rows from 2-5 specify the parameters of AdapMutation; parameters j is an integer representing the index associated to the rule; parameters k and g are pseudo random integer numbers; k specifies a gene to be changed; g specifies whether the k-th gene is to be changed or not; parameters a and b, are pseudo random integer numbers, representing new limits for the interval chosen to be changed. The remaining rows specify which conditions the rule must follow.

<table>
<thead>
<tr>
<th>Table 5 Adaptive Decision Table for AdapMutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>j</td>
</tr>
<tr>
<td>x_{13}</td>
</tr>
<tr>
<td>x_{14}</td>
</tr>
<tr>
<td>x_{23}</td>
</tr>
<tr>
<td>x_{24}</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>x_{k3}</td>
</tr>
<tr>
<td>x_{k4}</td>
</tr>
</tbody>
</table>

and
4.4 Putting all together: AdaptGARP

The Table 6 shows the general setup to encode the GARP genetic algorithm in a corresponding Adaptive Decision Table. The second column of the first five rows labeled as "Condition rows", defines the conditions controlling the operation of the Adaptive Decision Table.

To the column marked as 0, corresponds the first three rows marked as "Actions", and them are used to set up initial values to the Adaptive Decision Table operation.

The columns marked as 1, and 2, controls the operation of the Adaptive Decision Table, corresponding to them the four Action rows. The columns marked as 3F, and 4F, indicates halt conditions for the operation of the Adaptive Decision Table.

<table>
<thead>
<tr>
<th>Condition rows</th>
<th>AdaptCrossover specification</th>
<th>AdaptMutation specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterations = 0</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>fitness &lt; 75%</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>iterations &lt; IterMax</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>fitness ≥ 75%</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>iterations ≥ 2</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>IterMax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>n = PopSize</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptive Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AdaptCrossover</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>AdaptMutation</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>q</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>n</td>
<td>b</td>
</tr>
</tbody>
</table>

![Table 6 AdaptGARP](image-url)
Also, in last seven condition rows we encode, in the columns beyond column 5, the sample data available (columns marked as $D_1$, $D_2$, $D_3$, ..., $D_k$) and reserve room for the rules the Adaptive Decision will produce (columns marked as $R_1$, $R_2$, $R_3$, ..., $R_k$).

As each rule must specify one interval for each of the environmental variables, there are $2^k$ rows for each rule to be produced; this way, the pair $A_i$, $B_i$ represents the extreme values of the interval for the $i$-th environmental variable of the rule $R_j$, where $h-j \leq i \leq k-j+r$ and $1 \leq j \leq k$.

As the available data sample has only punctual data, we required that $A_i = B_i$ when $5 \leq i \leq k+r$ and $1 \leq j \leq k$.

5. EXPERIMENTS

We run fourteen experiments with two data with both version of the algorithms. This section describes the data, used, the metrics used and the comparison of the results.

5.1 The data for the experiments

We use as data for our experiments the usual test set of openModeler system referent to the species “Furcata Boliviensis” and another data set of a species under study by the biological researchers of the project.

5.1.1 Furcata boliviensis

This data set contains 96 presence points with coordinates of longitude, latitude and altitude.

5.1.2 Species B

This data set contains 34 presence points with coordinates of longitude and latitude.

5.2 Performance metrics

We choose as our metric the grow of the number of iterations as a function of the required precision of the fitness function. In fact, the fitness function is based on a-priori probability calculated over the presence points—usually, half the sample used to generate the model compared with the posterior probability calculated taking into account all the available data in the sample.

5.3 Comparison of results

We run fourteen experiments with each data sample, requiring in each experiment a more accurate value of the fitness function as halt criterion; only for the “Furcata boliviensis” we obtained a different value of iterations for a required precision of 0.7%, as is showed in the Table 7.

<table>
<thead>
<tr>
<th>Fitness precision</th>
<th>GARP</th>
<th>AdaptGARP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.010</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>0.020</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>0.009</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>0.008</td>
<td>127</td>
<td>127</td>
</tr>
<tr>
<td>0.007</td>
<td>144</td>
<td>145</td>
</tr>
<tr>
<td>0.006</td>
<td>168</td>
<td>168</td>
</tr>
<tr>
<td>0.005</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>0.004</td>
<td>252</td>
<td>252</td>
</tr>
<tr>
<td>0.003</td>
<td>335</td>
<td>335</td>
</tr>
<tr>
<td>0.002</td>
<td>502</td>
<td>502</td>
</tr>
<tr>
<td>0.001</td>
<td>1002</td>
<td>1002</td>
</tr>
</tbody>
</table>

6. Conclusions

In this paper we describe how to implement genetic algorithm using Adaptive Decision Tables and, as concept proof, we made such an implementation of the GARP genetic algorithm.

To compare the performance of the proposed implementation, we ran several experiments with both implementation using the same two data samples and the same parameters. For one sample data the performance of our implementation was the same as the original implementation and, for the other data sample, there was only a minimal loss of performance in one of the experiments.

Future work includes exercising the new implementation with other, and more massive, data samples; made available our implementation for the biological researchers community and to show another genetic algorithm implemented using an Adaptive Decision Tables to validate this process as an alternative implementation without loss of performance.

7. Acknowledgments

Our thanks to FAPESP funding under the process 2004/11012-0.
The first author wants to thanks to FAPESP funding under the process 2006/05797-0.

8. References


Annex 11

P-GARP (Parallel Genetic Algorithm for Rule-set Production) for clusters applications

F. S. Santana\textsuperscript{1,2}, C. Bravo\textsuperscript{3}, A. M. Saraiva\textsuperscript{1}, P. L. P. Correia\textsuperscript{1}

\textsuperscript{1}Escola Politécnica da Universidade de São Paulo, Brazil; \textsuperscript{2}FEI/UNIFIEO, Brazil; fabiomalar@gmail.com, annsaraviv@usp.br

Abstract

Ecological niche models may predict the potential geographical distribution of biological species and may be applied for conservation purposes, sustainable use and environmental preservation. The purpose is to obtain areas environmentally similar to those where the species occurs and, using suitable algorithms, generate a geographic map that shows the probability of occurrence of the study species in the selected area. In spite of all important applications of that, the model generation process may be highly computationally demanding, depending on the chosen algorithm and on the amount of environmental layers and species points, therefore alternatives must be considered to improve the time that takes to generate a model – at least in critical tasks. The most common solutions are related to the use of clusters, with many sequential executions of the algorithms in parallel, resulting in the generation of more models in the same period of time, which is desirable and efficient for some purposes, e.g. for generating several models, in order to compare them or to obtain geographical models for different species. Nevertheless, the performance of each particular algorithm does not change, taking the same time of the sequential solution to generate a model (of course, discarding machine differences). Thus, in order to obtain a better performance and scalability in general, such improvements may also require the design of new parallel algorithms, in order to break the so-called “parallel software barrier” and to reduce the final time of each model generation. Considering the available algorithms for the modeling process, the Genetic Algorithm for Rule-set Production (GARP) is currently one of the most important available worldwide and may be considered a critical point, so this paper presents the P-GARP – a parallel version of the GARP algorithm. Moreover, for better parallel algorithm design, the chosen of an appropriate parallel computational model is required so as to allow the efficient use of clusters, exploiting the large-scale scientific computation potential and resulting in better practical results; the chosen model was already successfully used in other parallel bioinformatics solutions, like DNA and protein sequence analysis and, perhaps, may inspire the design of other parallel solutions. The P-GARP algorithm was implemented and tested as part of the openModeller framework, an open-source software tool available for free use, and probably will integrate the framework in the cluster future version. At last, alternatives and future related work are discussed.

1. Introduction

The purpose of predictive niche models is to identify areas environmentally similar to those where a species is known to occur, encouraging the conservation and preservation of the environment and of the natural resources, in order to answer a set of very complex questions related to current, future and past geographic distribution of genes, organisms and ecosystems from an ecological and evolutionary perspective. Generated models can be used to propose scenarios for environmental preservation and sustainable use, potential invasive species valuation and climatic changes impacts on biodiversity.

The process for generating a model combines occurrence species data, usually named species presence and absence points, with ecological and environmental data distributed in environmental layers, to generate a model that predicts the potential geographic distribution of a species, projected onto a map of the study region. Species points are in the form of geo-referenced coordinates of latitude and longitude for confirmed localities, and the ecological and environmental variables are selected between biotic and abiotic factors, like temperature, precipitation, elevation, geology, soil and vegetation. Even considering recent progress and computational tools, the study and conservation of biodiversity are not easy tasks and smart technologies need to be employed.

Nowadays, one of the most related problems of the available tools is the time that a model takes to be generated, when many environmental layers and species points are incorporated. In Santana et al. (2006), the process for generating a model were described, as part of the openModeller project [http://openmodeller.sourceforge.net], an integrated open source software framework for species distribution modeling, available at that website for download and free use. The analysis of the presented process and the practical performance tests realized at Escola Politécnica da Universidade de São Paulo [http://www.poli.usp.br]
and at CRIA [http://www.cria.org.br] lead to the conclusion that one of the most time consuming step is the algorithm execution and projection. Consequently, to improve the performance of the modeling process is necessary to improve the performance of the modeling algorithms.

However, by considering an algorithm as a sequence of steps and by supposing that all steps are required so as to solve the problem, the best alternative for performance improvements is to use supercomputers or parallel machines, like the clusters – multiple processor machines that allow the execution of tasks in parallel (or at the same time), and each machine/processor in a cluster is usually called a node. Other alternative would be to design faster algorithms, but even this is succeed, the cluster solution continues to be useful.

Cluster solutions has been used to execute, e.g., several instances of the same algorithm in different nodes, preferably with different data sets or under different conditions, resulting in more models generated in the same period of time. This improvement is welcome and important results may derive from that, in fact, more than one institution or researchers have projects into this direction (Caulios et al., 2005; Stockwell et al., 2006).

That may lead to a higher throughput but the performance of the algorithm will not be improved. In other words, if an algorithm in a regular non-parallel machine takes hours or days to get a result, this situation will remain the same in a cluster. Such a real improvement only will be achieved by breaking the “parallel software barrier” (Valiant, 1990), therefore, the design of specific solutions that explore the resources available in parallel machines is demanding, in order to obtain maximum advantage of the hardware available. That are usually known as parallel algorithms, and have the ability to combine the results obtained in distinct nodes to solve the same problem, instead of only executing different non-parallel algorithms in each node of the cluster.

For designing a parallel algorithm, a computer parallel model is required (the same occurs in the design of non-parallel algorithms, that use the von Neumann computer architecture approach) and the comprehension of the model is required for the comprehension of the parallel algorithms.

All the algorithms available at the openModeller framework or related tools are candidates to be redesigned, in order to obtain the maximum benefits of the clusters, depending on a previous analysis – algorithms

2. Materials and methods

The openModeller framework was the environment used for the P-GARP implementation and tests. The implementation of the modeling process, data cleaning routines, data format conversions, model presentation and other features that may simplify the research job can be found at the framework version 0.3.4. Besides, a good environment for the developer is also available, since new algorithms and tools may be included for tests and evaluation. Moreover, a version of the tool integrated to a cluster is foreseen, probably for the next year (Caulios et al., 2005).

The P-GARP was designed based on the CGM parallel model for algorithm design, and will be presented in next subsection. The same model was successfully used for designing parallel algorithms for solving several theoretical and practical computational problems (Delne, 2006, Delne, 1999), e.g. involving graph theory (Chan et al, 2003, Delne et al., 2002) and computational geometry (Delne, Fabri and Rau-Chaplin, 1996), among other parallel bioinformatics research, like DNA and protein sequence analysis (Pitre et al. 2006).

After the model presentation, the GARP and P-GARP algorithms will presented, followed by some biodiversity spatial distribution models, generated with the same data for both algorithms. As the openModeller was used to implement P-GARP, the GARP version considered in this paper was also the implemented at the openModeller, so as to maintain coherence in the models generations and comparisons.

2.1. The CGM Model
Synchronous Parallel — named as BSP — bridging model for parallel computation was proposed, as a candidate for this role. The BSP approach requires that the ratio “local computation speed / router bandwidth” is low or fixed, even for increasing number of processors, and it also introduces the concept that parallel computation should be modeled as a series of supersteps, rather than individual message passing steps or sharing memory accesses (Deline, Fabri and Rau-Chaplin, 1996). Each superstep consists of various message passing operations and are separated by barrier synchronization, which means that the BSP model made parallel computation coarse-grained, reducing the synchronization overhead. Using the BSP model as a starting point, the Coarse-Grained Multicomputer — named as CGM — was proposed, adding coarse-grained communication and coarse-grained memories (Deline, Fabri and Rau-Chaplin, 1996). Efficient CGM algorithm depends on a fixed small number of large messages and one single thread and memory space per processor, reducing the communication overhead and the allowing large-scale samplings, besides reducing context-switching overheads.

The CGM model is represented by a set of \( p \) processors with at least \( O(n/p) \) local memory each — where \( n \) represents the size of the input, connected to some arbitrary interconnection network (Deline, Fabri and Rau-Chaplin, 1996) that may deliver messages in an all-to-all broadcasting between all processors. A CGM algorithm is an alternate sequence of supersteps rounds — as defined in the BSP model — composed by a local computation round and a communication round. The communication round allows each processor to exchange data with all other processors whereas, for local computations, the best available sequential algorithm must be used, considering also data locally available. Both local computation time and communication time must be considered to evaluate the algorithm performance, but the improvements may be observed only by counting the supersteps. The local computation round, when performed in parallel, will represent the natural gain of the parallel solution — so the result will be obtained in a time proportional to \( O(n/p) \) steps — while the number of supersteps represents the lost, because the communication will require extra time.

Note that the CGM model is an adequate and simple model sufficiently close to a cluster architecture, where a set of rack mounted standard machines are connected by a fast switch — for more details, see [www.deline.net]. It also gives a reasonable prediction of the performance when parallel algorithms designed based on this model are implemented, even when the theoretical complexity is considered.

2.2. The GARP - Genetic Algorithm for Rule-set Production

GARP is a genetic algorithm projected to predict spatial distribution of animal and plant species. In order to do that, the occurrence or absence of the given species at specified locations is correlated with the environmental conditions in order to identify suitable geographical and climatic conditions for the presence or absence of the specie. Each of these conditions can be viewed as an interval of values for which the presence or absence of the species were observed and the rules the algorithm produces become the Cartesian product of these intervals in \( \mathbb{R}^n \), where \( n \) is the number of environmental variables considered [Stockwel-Noble 1999]. The implementation focused on three kinds of rules, namely, RangeRule, NegateRange and LogicRule which we describe in the following paragraphs.

RangeRule: this kind of rules are similar to intervals in the real line; in fact, a RangeRule \( R \) can be read as "If the value of the environmental variable \( X \) is between the values \( A \) and \( B \), where \( A \) is less than \( B \), then the species is present".

NegateRange: this kind of rules captures the presence of a species for values of a environmental variable out of a given interval: "If the value of the environmental variable \( X \) is less than the value \( A \) or greater than the value \( B \), where \( A \) is less than \( B \), then the species is present".

LogicRule: reflects the presence of a species as a probability calculated, in a logistic fashioned way, in terms of the different environmental variables: "Lets assume that the variables \( X_1, X_2, \ldots, X_k \) determines the presence of a species in a geographical area. Then the probability of the presence of the species in a specific point \( p_s \) of this area is given by: Prob( the species is present at \( p_s \) ) = \( 1 - e^{a_1 x_1 + a_2 x_2 + \ldots + a_k x_k} \).

In order to validate the produced rules, GARP first operates on a part of the presented sample — typically, half of the sample— in a preprocessing stage similar to that for training a neural network and, after the rules have been produced for this part of the sample, GARP enters a second stage to validate the produced rules on the sample data points that were not used to produce the rules. This scheme enables GARP to determine the adequacy of the produced rules as a tool for predicting the presence or absence of the studied species [Stockwel-Peters 1999].
Below, the openModeler implementation of the initialization and iterate procedures of the GARP algorithm is available – implementation details were omitted for the sake of comprehension. For that, consider that _offspring and _fittest are rule-sets, _popsize is the size of the population (in this case, the number of rule-sets), defaultPerfIndex, perfBest, perfWorst and perfAvg corresponding to the default, best, worse and average performance evaluation of the specified rule-set.

```java
void initializeGARP ()
{
    _offspring = new GarpRuleSet ( 2 * _popsize );
    _fittest = new GarpRuleSet ( 2 * _popsize );
    colonize (_offspring, _popsize );
}

void iterateGARP () {
    colonize (_offspring, _popsize );
    _offspring = trim (_offspring, _popsize );
    _fittest = trim (_offspring, _fittest, defaultPerfIndex );
    _fittest = performanceSummary ( defaultPerfIndex, perfBest, perfWorst, perfAvg );
    Select (_fittest, _offspring, _popsize );
    Crossover (_offspring );
    Evaluate (_offspring );
    _fittest = filter ( defaultPerfIndex, _significance );
    gen++;  
}
```

The initializeGARP only will be performed once, in order to create the initial rule-sets. After that, the number of iterations can be determined by two different criteria: if the maximum number of predefined iterations or if the minimum value for convergence were reached, which happens first.

The colonize method receives a rule-set and a prefixed size for the population and, if necessary, generates the number of rules necessary to keep the size of the population at the prefixed value, so, in the first iteration of the while loop, colonize generates the adequate number of rules for the first population. On the other hand, the trim method cuts off the current population _offspring, if necessary, to a number of rules enough to reduce the size of the population to the same prefixed size.

The select method is an implementation of the Stochastic Universal Sampling SUS, algorithm by James E. Baker [Baker, 1987]. The first parameter is the current population and the second will store the result of the SUS execution; the third parameter gives the proportion contribution to the new population from an historical archive set. The crossover method takes a prefixed crossover rate of the _offspring population and choose at random a pair of chromosomes indexes and exchanges all the chromosomes between these indexes of both individuals. The mutate method exchange at random the values of two chromosomes in each rule of the _offspring population.

The evaluate method determine the performance of each rule in terms of the number of points that the rule applies to (sum of strength) pVs, the number of points with the same conclusion as the rule (sum of certainty) pVs, the sum of points correctly predicted by the rule pXs, and the number of points selected by the rule.

The keepFittest method, tries to find a similar rule in the _fittest rule set and, if it finds such a rule, replaces it if the first has better performance; in case that there is no similar rule, the rule is inserted in the _fittest rule set. That is performed for each rule in the _offspring rule set.

The filter cuts off from the fittest rules with a poor fitness value. Also it decrements the number of rules in the _fittest attribute, according to the quantity of deleted rules. The performanceSummary method updates, for the _fittest current population the values of the better, worst and average performance of its rules.

Finally, the number of iterations gen is incremented and, once an iteration is finished, the stop conditions are tested to terminate the loop execution or to do a repeated execution of the same iteration.
2.3. The P-GARP – parallel version of the GARP Algorithm

Accordingly to the CGM model, the P-GARP algorithm must be defined by establishing a superstep, containing a local iteration step and a communication round. In order to maintain the same implementation of GARP algorithm, the previous implementation will be used in the P-GARP description. This is a good approach because of different implementations of the GARP algorithm will be able to do the same.

The initializePGARP only will perform the same initializeGARP. In such a case, the most important is to observe that, according to the CGM requirements, all necessary data must be available for each processor before the execution starts, otherwise the extra communication requirements may invalidate the performance improvements.

In order to understand the iterationPGARP, suppose that $p$ is the number of nodes (processors) that will be performing the parallel execution – this number may be variable for each execution, depending on cluster configuration. _offspring[i] stores the _offspring rule-set for each processor, where $i$ is the number of the processor. This data will be received after a communication round. Basically, this procedure combines the results of the all $p$ parallel and independent previous iterateGARP() executions, by using the keepFittest method, the colonize() and trim() methods are only applied to control the size of the rule-set.

```c
void initializePGARP ()
{
    initializeGARP();
}

void iteratePGARP () {
    for ( int i=0; i<p; i++ ) // Para todos os processadores
    {
        keepFittest(_offspring[i], _localFittest, defaultPerfIndex);
        colonize(_localFittest, _popsize);
        _localFittest->trim(_popsize);
    }
    _offspring = _localFittest;
    iterateGARP();
}
```

The communication round, performed after initializePGARP() and after each execution of iteratePGARP(), unless one of the finalization criteria be reached, consists of the sending, by each processor to all the others, the last rule-set obtained, stored in the _offspring rule-set. Each processor must receive this data and store, perhaps in a vector of rule-sets, likewise the code shown above. The software utilized for controlling the cluster is necessary for performing this task, but most of them offers easy ways for sending and receiving all-to-all broadcasting messages.

2.4. P-GARP analysis

i. Correctness

The keepFittest method does not generate a worse rule-set than the previous, by definition, and each iterateGARP() is performed with the rules produced by a previous iteration of the same iterateGARP(). Thus, both facts indicate that the P-GARP algorithm can be considered correct, despite the differences from the traditional non-parallel GARP solution.

ii. Performance

First, suppose that neither GARP nor P-GARP stop by the convergence criteria, so only after the predefined maximum number of generations the execution will stop. In this case, suppose that $p$ is the number of processors and $n$ is the number of iterations that will be performed. The expected number of iterations that will be performed by each processor will be $O(n/p)$. All processors works in parallel, hence $p$ iterations are realized at the same period of time. As the number of iterations in GARP and P-GARP are the same, and that number is increased by all processors before iterateGARP – not iteratePGARP – finishes, the expected total time for performing the $n$ iterations also will be $O(n/p)$.
If the number of processors is small, the extra operations introduced by P-GARP for the generation of the new rule sets with the result of previous $p$ iterate GARP($p$) will be $O(p)$. In this case, as $p << n/p < n$, some times it may even be disregarded. Otherwise, when $p$ is high and this step leads to loss in the performance, an alternative is to recombine the results in pairs, using a recursive method – e.g. the same used by the MergeSort algorithm – this will reduce the complexity from $O(p)$ to $O(\log p)$ and probably will solve the problem.

At last, the time taken by a communication round must be evaluated but, since the rule-set is usually a small vector compared to the cluster bandwidth, this tends not to be a problem, but it is dependant on the machine configuration and may be solved by establishing a machine requirement for the cluster communication speed, because nothing else can be done to solve the problem in P-GARP.

On the other hand, if GARP or P-GARP stop by the convergence criteria, it would be reasonable to expect that, with the same data and conditions, both would have similar behavior, but so far there are no proofs of that fact. As the number of iterations is limited by the maximum established number, this unsolved problem is minimized.

2.5. Tests and results

Using openModeller and forcasta boliviana species, both software and description available at the openModeller web site, the following executions of GARP and P-GARP were realized. Figure 1 shows (A) GARP and (B) P-GARP, performed with 100 iterations; figure 2 shows (C) GARP and (D) P-GARP, performed with 400 iterations. Figure 3 shows (E) GARP and (F) P-GARP executions with a different species (training species), performed with 20 iterations, and Figure 4 shows (G) GARP and (H) P-GARP executions with the same species, performed with 100 iterations.

Observing the models above, it is possible to conclude that some proximity exists between the models generated by GARP and P-GARP, perhaps compatible with different executions only of the same GARP algorithm. It also seems that, as the number of iterations increase, the results get closer. But this point still need to be better investigated and, so far, just can be used as an indication that the algorithm is correct. Other tests were performed but not included in this paper. They were performed with different species and environmental layers, and the results were similar to the presented in Fig. 1 and may be supplied as required.

3. Discussions and future works

Only the P-GARP design is not enough for establishing the algorithm as a modeling solution whose relevance may be compatible with the original GARP algorithm. The main objective in this paper was to present a solution faster than the sequential GARP for clusters implementations, in order to solve a parallel problem, but more
effort is needed before concluding the research about P-GARP. Therefore, other theoretical analysis and practical tests must be performed so as to attribute to the P-GARP the same status that the GARP algorithm has in modeling generation.

At first, a version must be submitted to a stress test at a cluster, in order to verify if the expected performance will be achieved. Performance evaluation tools or automatic tests, performed by testers – programs implemented so as to test programs – may be used for that.

Other relevant discovery will be to evaluate, if possible, the distance between solutions derived from GARP and P-GARP. Moreover showing the correctness of the P-GARP, that also would give the researcher a parameter for deciding which algorithm to choose between both. Maybe P-GARP gains in performance but looses in efficacy, maybe gains in both, maybe the gain in performance is not enough to justify the algorithm adoption, but this a very important point to be investigated.

Likewise, the convergence of the parallel algorithm must be deeply studied, in order to proof, on one hand, that P-GARP is correct when this is the stopping criteria and, on the other hand, to verify relationships between GARP and P-GARP convergence.

At last, implementation details must be carefully analyzed for achieving the maximum performance when a cluster solution will be delivered. In any way, P-GARP relevance only will be really evaluated when the cluster version of the openModeller – or other compatible tool for modeling generation – is available with the P-GARP algorithm implementation and be submitted to modeling experts approval and this tends to happen soon, so the relevance of this work will be tested.

Acknowledgments

The authors are grateful to FAPESP – Fundação de Amparo à Pesquisa do Estado de São Paulo – Brazil, for the support to the OpenModeller project, under grant 04/11012-0.

References


Annex 12

Análise da ferramenta openModeller do ponto de vista do usuário de modelagem de biodiversidade

Projeto: OpenModeller - A framework for species distribution modeling. Processo FAPESP - 04/11012-0

Período: novembro de 2006 a janeiro de 2007

Análise da interface desktop do OM – versão 1.0.4

De um modo geral a versão 1.0.4 está muito boa, bem apresentada e bastante rica em detalhes para facilitar/agilizar o usuário de modelagem de biodiversidade. A interface lidou com uma grande quantidade de layers (teste realizado com 64 camadas ambientais) sem problemas de finalização. Foi incluída nesta interface uma janela para conversão de formatos, o que é bastante oportuna e útil, uma vez que libera o usuário de ter que usar um Sistema de Informação Geográfica para isso. A inclusão de uma janela de conversão de estrutura de arquivo texto também é muito útil pois a maior parte dos dados (pontos de ocorrências) vêm em formato não compatível com o OM (tipo campo contendo a latitude antes da longitude, com a primeira linha não comentada, etc). Por último, a novidade mais importante dessa nova interface é a geração de uma tabela contendo os pontos de ocorrência e seus respectivos valores referentes às camadas ambientais utilizadas. Esta tabela será muito útil para o usuário, pois pode ser diretamente utilizada para análises de pré-modelagem.

A seguir são apresentadas algumas sugestões com a intenção de contribuir para melhorar ainda mais a interface desenvolvida. Essas sugestões visam tornar a utilização da interface mais simples (mais intuitiva) para o usuário e também visam reportar alguns problemas de funcionamento.

- Durante a realização de um experimento de modelagem, ao se trocar o conector de local para remoto, são perdidas todas as configuração feitas no gerenciador de camadas ambientais. Isso dificulta a realização de testes de grande porte, entre performance local e via web, para o usuário.
- A forma de mudar os parâmetros dos algoritmos, no gerenciador de algoritmos, por clonagem do algoritmo não é intuitiva para o usuário. Primeiro ele tem que clonar o algoritmo para depois mexer nos parâmetros. Minha sugestão é permitir que o usuário mexa nos parâmetros diretamente e permitir que ele salve essas mudanças (para uso futuro) no final do processo.
- No seletor de camadas ambientais, para escolher/mudar a máscara de entrada, primeiro é preciso adicionar a camada máscara e depois é preciso selecionar a mesma camada no combo masklayer. A segunda etapa é facilmente esquecida pelo usuário. Uma vez escolhida a máscara, é de se esperar que a interface carregue essa camada como máscara diretamente. Esse é um detalhe importante, pois como não é intuitivo para o usuário, é comum gerar o modelo pensando que o mesmo está sendo rodado com a máscara escolhida mas na verdade, como o usuário não selecionou no combo masklayer, o modelo está rodando com a máscara utilizado no último procedimento. E isso só é percebido no final do processo de modelagem.
• Outro procedimento que não é muito intuitivo para o usuário é ter que criar dois conjuntos de camadas ambientais apenas para escolher outra máscara. Este caso é bastante frequente quando o usuário quer criar o modelo em uma determinada região e projetar em outra. Isso não é muito produtivo pois a utilização de máscaras diferentes para criação e projeção de modelos é muito comum (prática bastante utilizada para modelagem de espécies invasoras).

• Uma limitação da ferramenta, em termos de uso de dados ambientais é o fato do openModeller não ler arquivos shape. Essa é uma deficiência da biblioteca utilizada, o GDAL, que só lê arquivos raster. Mas é bom deixar esse fato registrado porque, além desse formato ser bem mais "enxuto" que grid ou asc, vários layers ambientais estão originalmente nesse formato e é uma pena o openModeller não tratar disso.

• Quando estamos rodando um experimento com o algoritmo GARP, devido ao tempo de processamento ser grande, a janela do OM (interface Windows) fica praticamente inacessível (a informação contida na janela acaba não sendo carregada na tela, ou demora muito pra ser carregada). O que acontece na prática, é que não dá para ver a progressão da geração do modelo. O usuário fica sem saber se a modelagem continua sendo gerada ou se o experimento "capotou". Como é comum um experimento rodado com GARP demorar horas ou dias para ser completado, é bastante provável que o usuário acabe matando um experimento que estava sendo gerado.

• Na plataforma Windows, a janela de conversão de formatos não está funcionando, toda a interface “fecha” ao tentarmos utilizar esta tela.

• Na plataforma Windows, a janela de conversão de estrutura de dados texto não está funcionando, toda a interface “fecha” ao tentarmos utilizar esta tela.

• A tabela gerada com os dados registros de ocorrência das espécies e os valores correspondentes das camadas ambientais é gerada em formato .csv, essa tabela seria mais facilmente (mais diretamente) utilizada pelo usuário se fosse gerada em formato .txt, separado por tabulações.
Annex 13

Public environmental data available at CRIA
(20GB of environmental information, more than 4 thousand folders in EsriGrid format)

Climate Data:

IPCC – Intergovernmental Panel on Climate Change
(http://www.ipcc-data.org/)
Climate data (present and some future projections, models CCC e HadCm2, and 2 past projections, pleistocene and holocene) 50Km resolution.

Data description: the files containing the observed climate data are in ASCII format with one file per year per variable. The first and second lines of the file contain information on the grid size, for example: grd_sz xmin ymin xmax ymax n_cols n_rows n_months missing 0.5 0.25 -89.75 359.75 89.75 720 360 12 -9999. This is followed by 12 monthly grids that are n_cols by n_rows in size. Each record contains n_cols (=720) columns longitude values, format 720i 5, starting at xmin (= 0.25 deg East) and ending at xmax (=0.25 deg West) The first record starts at January, ymax (=89.75 deg North) and the last record is the row for December, ymin (= -89.75 deg South). Co-ordinates represent the centre of each grid cell. Missing values (Antarctica, oceans and major inland water bodies) are assigned integer values of -9999.

- Diurnal Temperature Range
- Ground-frost Frequency
- Precipitation
- Radiation
- Wet Day Frequency
- Mean Temperature
- Maximum Temperature
- Minimum Temperature
- Vapour Pressure
- WindPercent

Worldclim (http://www.worldclim.org/)
Climate data of temperature (maximum, mean and minimum), precipitation and altitude in 4 different resolutions (10’, 5’, 2.5’ e 30”).

Data description: these layers (grid data) cover the global land areas except Antarctica. They are in geodetic coordinate system (not projected, i.e., ‘GEOGRAPHIC’ or ‘LATLONG’ system). The datum is WGS84. They are available at 4 different spatial resolutions: from 30 seconds (0.93 x 0.93 = 0.86 km2 at the equator) to 2.5, 5 and 10 minutes (18.6 x 18.6 = 344 km2 at the equator). The original data were at a 30 second resolution, the other data have been derived through aggregation, by calculating the mean of groups of cells. Cells with 'no data' were ignored. In other words, if some of the original cells were on land, and some cells were on sea, the aggregate cells have data. Only if all original cells have 'no data' then the aggregate cell has 'no data'. Aggregation was done for monthly
precipitation, minimum, mean and maximum temperature. The Bioclimatic variables were calculated from these aggregated data.

**Bioclimatic:** bioclimatic variables are derived from the monthly temperature and rainfall values in order to generate more biologically meaningful variables. These are often used in ecological niche modeling (e.g., BIOCLIM, GARP). The bioclimatic variables represent annual trends (e.g., mean annual temperature, annual precipitation) seasonality (e.g., annual range in temperature and precipitation) and extreme or limiting environmental factors (e.g., temperature of the coldest and warmest month, and precipitation of the wet and dry quarters). A quarter is a period of three months (1/4 of the year). They are coded as follows:

- **BIO1 = Annual Mean Temperature**
- **BIO2 = Mean Diurnal Range (Mean of monthly (max temp - min temp))**
- **BIO3 = Isothermality (P2/P7) (* 100)**
- **BIO4 = Temperature Seasonality (standard deviation *100)**
- **BIO5 = Max Temperature of Warmest Month**
- **BIO6 = Min Temperature of Coldest Month**
- **BIO7 = Temperature Annual Range (P5-P6)**
- **BIO8 = Mean Temperature of Wettest Quarter**
- **BIO9 = Mean Temperature of Driest Quarter**
- **BIO10 = Mean Temperature of Warmest Quarter**
- **BIO11 = Mean Temperature of Coldest Quarter**
- **BIO12 = Annual Precipitation**
- **BIO13 = Precipitation of Wettest Month**
- **BIO14 = Precipitation of Driest Month**
- **BIO15 = Precipitation Seasonality (Coefficient of Variation)**
- **BIO16 = Precipitation of Wettest Quarter**
- **BIO17 = Precipitation of Driest Quarter**
- **BIO18 = Precipitation of Warmest Quarter**
- **BIO19 = Precipitation of Coldest Quarter**

This scheme follows that of ANUCLIM, except that for temperature seasonality the standard deviation was used because a coefficient of variation does not make sense with temperatures between -1 and 1). This AML (Arc-Info script) was used to generate these layers.

**Temperature:** tmean = average monthly mean temperature (°C * 10), tmin = average monthly minimum temperature (°C * 10), tmax = average monthly maximum temperature (°C * 10)

**Precipitation:** prec = average monthly precipitation (mm)

**Altitude:** alt = altitude (elevation above sea level) (m) (from SRTM)

**Topographic data**


Data description: HYDRO1k is a geographic database developed to provide comprehensive and consistent global coverage of topographically derived data sets, including streams, drainage basins and ancillary layers derived from the USGS’ 30 arc-second digital elevation model of the world (GTOPO30). HYDRO1k provides a suite of geo-referenced data sets, both raster and vector, which will be of value for all users who need to organize, evaluate, or process hydrologic information on a
continental scale. Developed at the U.S. Geological Survey’s Center for Earth Resources Observation and Science (EROS), the HYDRO1k project's goal is to provide to users, on a continent by continent basis, hydrologically correct DEMs along with ancillary data sets for use in continental and regional scale modeling and analyses. Detailed descriptions of the processing steps involved in development of the HYDRO1k data sets can be found in the Readme file. This work was conducted by the U.S. Geological Survey in cooperation with UNEP/GRID Sioux Falls. Additional funding was provided by the Brazilian Water Resources Secretariat and the Food and Agriculture Organization/Inland Water Resources and Aquaculture Service.

- Aspect
- Slope
- Elevation
- Topographic index
- Flow direction
- Flow accumulation

Vegetation index NDVI (Normalized Difference Vegetation Index)


Data description: measurements derived from the Advanced Very High Resolution Radiometer (AVHRR) Sensor and that are well-known as sources of detailed, fine-scale information regarding photosynthetic mass, or ‘greenness’ of the landscape; the multitemporal nature of these data sets effectively summarizes a combination of land use/land cover and vegetation phenology. Original sources: [http://edc.usgs.gov/products/satellite/avhrr.html](http://edc.usgs.gov/products/satellite/avhrr.html); [http://www.whrc.org/southamerica/LBADATA/Amaz_8K_AVHRR.htm](http://www.whrc.org/southamerica/LBADATA/Amaz_8K_AVHRR.htm); [http://www.whrc.org/southamerica/LBADATA/Amaz_1K_NDVI.htm](http://www.whrc.org/southamerica/LBADATA/Amaz_1K_NDVI.htm)
Annex 14

Seminars

Date: 2nd June, 2006
Location: Poli
Presentations: Metadata standards (Rubens R. Fonseca, Poli), Work plan for developing a services architecture for openModeller (Anna Catarina B. Tavella, Poli), Work plan for performance analysis of openModeller (Mariana R. Franco, Poli), Work plan for implementation of new algorithms (Danilo J. S. Bellini, Poli), General presentation about the openModeller architecture group (Fabiana S. Santana, Poli).

Date: 29th June, 2006
Location: Poli
Presentations: Discussion about openModeller’s architecture (Fabiana S. Santana, Poli), Profiling strategy for openModeller (Jeferson Martin, Poli), Plans for the cluster and paralelization of code (Prof. Liria Sato, Poli).

Date: 10th August, 2006
Location: Poli
Presentations: New desktop interface (Tim Sutton, CRIA), Performance analysis (Jeferson Martin, Poli).

Date: 21st September, 2006
Location: Poli
Presentations: Cluster acquisition (Mario Donato Marino, Poli), Using Condor (Nilton Cézar de Paula, Poli), Paralelization with MPI (Elaine Machtans, Poli), P-GARP (Fabiana S. Santana, Poli), New algorithm Environmental Distance (Danilo J. S. Bellini, Poli).

Date: 26th October, 2006
Location: Poli
Presentations: AquaMaps (Renato De Giovanni, CRIA), Web Services API (Tim Sutton, CRIA), Report about INPE activities in the project (Silvana Amaral, INPE), Web Services architecture and model repository (Karla Fook, INPE), Integration between openModeller and TerraLib (Alexandre C. Jardim, INPE), Sudy case – modeling Coccocypselum P. Br (Cristina Bestetti Costa, INPE).

Date: 28th November, 2006
Location: INPE
Presentations: Report about CRIA activities in the project (Renato De Giovanni, CRIA), Demonstration of the new Desktop interface (Tim Sutton, CRIA), Report about Poli activities in the project (Prof. A. M. Saraiva, Prof. João José Neto, Prof. Liria Sato, Prof. Pedro L. Pizzigatti Correa), Performance analysis (Jeferson Martin, Poli), Web Service for cooperation in biodiversity modeling (Karla Fook, INPE).

Date: 8th February, 2007
Location: Poli
Presentations: Report about INPE activities in the project (Lúbia Vinhas and Silvana Amaral), Report about Poli activities in the project (Prof. A. M. Saraiva), Paralelization
and Cluster (Prof. Líria Sato, Poli), P-GARP/ AdaptiveGARP / Cluster (César Bravo, Poli).
Annex 15

openModeller algorithm's documentation

Environmental Distance

High level algorithm's general documentation

Danilo J. S. Bellini
Index

Table of Contents

Algorithm definition..............................................................................................................3
Black box model – inputs and outputs definition..............................................................3
Mathematical issues...............................................................................................................4
Algorithm explanation..........................................................................................................4
Distance equations..............................................................................................................7
  Euclidean distance (2-norm Minkowski distance; Generalization of Pythagorean theorem)........7
  Manhattan/Gower distance (1-norm Minkowski distance)..............................................8
  Mahalanobis distance.......................................................................................................9
  Chebyshev distance (infinite-norm Minkowski distance; Chessboard distance)...............12
Internal architecture..........................................................................................................13
  Overloaded methods.......................................................................................................13
    Method “initialize”.......................................................................................................13
    Method “getValue”......................................................................................................13
Specific methods..............................................................................................................14
  Method “InitDistanceType” ...........................................................................................14
  Method “Distance”........................................................................................................14
Execution flow....................................................................................................................15
**Algorithm definition**

The algorithm Environmental Distance is a collection of fully deterministic distance based prediction algorithms. Notice that the algorithm is deterministic but its results are probability values.

It's needed to imagine an ndimensional world where each axis is an environmental possibility, that means that a sample point will be seen as a value in an Euclideanlike ndimensional geometrical space.

In the space used for an experiment, it's possible to calculate a distance between 2 points using an equation (commonly the norm of the difference vector). There are several possibilities of equations for that, but the biologists only want the ones that are useful for their data.

All distances are calculated between 2 points and it's needed to find a probability for each point given by openModeller to the algorithm. That means that we need to find one point that represents the place where the distance should be calculated for each point oM gives. For that, the algorithm uses the presence points given by the user.

**Black box model – inputs and outputs definition**

The algorithm only use the presence points given by the user, and ignores all absence ones. All layers are used as new dimensions to calculate the distance. With that layers we can find one sample point for each presence geographical point. To avoid confusion, presence points are for now the sample points found with presence geographical points. The output is a probability value given for each point oM gives to the algorithm.

There are 3 parameters used by the algorithm: one tells the metric that should be used (Euclidean, for example), the other one is an amount of points that are used to find a mean point (explained later) and the last one is a maximum distance saturation, to avoid null models as outputs.

**Mathematical issues**

**Algorithm explanation**

For each point given by oM, the algorithm finds a new point to calculate a distance and use this distance as a basis for reckoning a probability value. This new point is always the mean point of the “n” nearest points of the given point, for example:
The light points are presence points in a bidimensional world (for example the X axis means “temperature” and the Y axis means “air humidity”), and the dark point is a point given by oM.

The algorithm can calculate the distances between the given point and each presence point. With that it can put all presence point in order from the nearest point to a far away one. In this example we can use the order:

Near
  c
  a
  b
  d

Far
  e

Now it uses the parameter that gives the amount “n” of nearest points that should be used. If we use “n” as 3, we will have to find the mean of the 3 nearest points and use the distance between it and the given point. Below is shown the mean point lighter, and the distance that is used to obtain the output probability value:
It's easy to understand that “n” should be an integer value in \([1;p]\) where “p” is the amount of presence points used as input.

The distance range starts from zero (when the point given by oM is exactly the mean point calculated) but its maximum should be calculated using an equation (or an algorithm). To help the calculations, the range of each dimension can be normalized to be always \([k1;k2]\), so all possible points are inside an hypercube. For Euclidean distance, the maximum distance is the distance between 2 opposite edges of the hypercube (square in this case).

The probability is calculated linearly, using a distance range. Assuming that the user hadn't put a saturation limit for the maximum distance, the distance range used by the probability evaluation is exactly the distance range seen above:

![Probability vs Distance](image)

Actually the line is a stairlike curve, since oM is a software that works with digital data and somehow the data have to be quantized and it can be a source of errors. In some cases the distance may be always in a small range near the probability of 1, or even a range that doesn't change the probability, giving models without information. For that, what can be done is:
The probability saturates in zero here because it's using $X$ as the maximum distance, not $\text{Max}$. With this, the probability range can be controlled to give values with significant representation, then the user don't need to worry with the quantization error of the model as before. Also, this process can be seen in an output grayscale map as a change in the brightness of the output.

The value of $X$ is always a fraction of $\text{Max}$. This fraction is exactly the value given by the user as a parameter (the maximum distance one), that will be called here as “$m$” for now.

Using $D$ as the distance found, the equation of the curve above in $[0;X]$ is:

\[
\text{Probability} = 1 - \frac{D}{X}
\]

or

\[
\text{Probability} = 1 - \frac{D}{m \cdot \text{Max}}
\]

Obviously “$m$” must be in $(0;1)$. 
Distance equations

Each distance seen before was calculated using the same equation, chosen by the user as a parameter. For the distances, doesn’t matter how the algorithm calculates the probabilities, it only needs the 2 points which will be used. So the distance equations have 2 sample points as inputs and a scalar value as output. The possibilities of equations in the algorithm are:

Euclidean distance (2-norm Minkowski distance; Generalization of Pythagorean theorem)

The usual distance used in geometry, physics, maths, etc. This distance can be calculated using the usual norm of the distance (difference) vector. This vector can be easily calculated subtracting one sample point from another. The norm is the square root of the sum of squares of each vector’s coordinate (already implemented in oM Sample class), or simply the square root of the dot product of the vector with itself.

The maximum distance in a hypercube is needed to calculate the probabilities. For an onedimensional world, we impose a size limit equals to L (normalizing the data). For a 2D world, we have a square, with size limit equals to \( L \cdot \sqrt{2} \), since each side has limits equals to L. In a 3D world, size limit is \( L \cdot \sqrt{3} \), because we have a cube. For a ndimensional world, we have the maximum distance equals to \( L \cdot \sqrt{n} \), because \( \sqrt{(L \cdot \sqrt{n-1})^2 + L^2} = L \cdot \sqrt{n} \) (induction using Pythagoras).

In 2D, the geometrical shape that represents the points with the same distance in Euclidean distance is a circle. In 3D it’s a sphere and in ndimension it’s a hypersphere.

Manhattan/Gower distance (1norm Minkowski distance)

The equation written originally as the “Gower” distance used in another predictive software needs a new probability equation where X must be 1, but
it's possible to change it to use the probability equation seen above (using the Manhattan formula, that doesn't use the range of the dimensions). The original “Gower” equation is:

\[
Distance = \frac{\sum_{i=1}^{n} \frac{|X_i - Y_i|}{\text{range}(\text{dimension } i)}}{n}
\]

Here “n” means the amount of dimensions that exists, XiYi is a coordinate of the difference vector and the range of the dimension “i” is always the same in M, defined when the data is normalized, so we can put it outside the sum. Since the probability equation already divides the distance by its range, this value should be forgotten here to be divided in the probability equation, making it generic enough to use the original definition of the X. So the equation used is:

\[
Distance = \frac{\sum_{i=1}^{n} |X_i - Y_i|}{n}
\]

The maximum value in this distance is always the range size of one dimension (all dimensions have the same range), because the maximum XiYi possible is always the range of the dimension i, and doesn't matter how many dimensions we have due to the division by “n”. This division doesn't exist in the original 1norm Minkowski distance, but it doesn't affect the results since the fraction of the maximum distance that will appear to calculate the probabilities won't change.

In this distance, for 2D data, the set of geometrical points that have a fixed distance to a particular point is a square rotated to have all its edges in the axis (regular rhombus, diamond or lozenge). For 3D it's a rotated cube and in ndimension it's a rotated hypercube, also with all edges in the axis.
Mahalanobis distance

This is the hardest distance used by oM. This one isn’t generic enough to satisfy all possible inputs. Its equation is:

\[ \text{Distance} = P^t \cdot \Sigma^{-1} \cdot P \]

Where the sigma inside the equation represents the covariance matrix and \( P \) is the difference vector (the one with coordinates equals to the \( XiYi \) seen above) written as a column matrix, as usually the linear algebra uses vectors in linear transforms.

To obtain the covariance matrix we need a set of important points. The algorithm uses the presence points as important points to find the matrix. It’s a square matrix with size “\( n \)” where “\( n \)” is the amount of dimensions we have. In its diagonal it have the variance value of each layer (the set used to find the variance and the covariances is the set of presence points), and in each other place it have the covariance between the values in each dimension, ordered. For a 3D world we have a matrix like this:

\[
\Sigma = \begin{bmatrix}
\sigma_1^2 & \sigma_{12} & \sigma_{13} \\
\sigma_{12} & \sigma_2^2 & \sigma_{23} \\
\sigma_{13} & \sigma_{23} & \sigma_3^2
\end{bmatrix}
\]

That means that for each place of the matrix, we have the value of a covariance:

\[
\sigma_{xy} = \frac{\sum_{i=1}^{N} (x_i - \bar{x})(y_i - \bar{y})}{N}
\]

or the value of a variance:

\[
\sigma_x^2 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N}
\]

The number “\( N \)” here means the amount of presence points given, so \( x_i \) is the value of the iest sample point in the dimension “\( x \)”. The \( \bar{x} \) is the mean of all \( x_i \) values:

\[
\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N}
\]
The geometrical set of points that have the same distance in relation to a particular point here is an hyperellipsoid. For 3D it's an ellipsoid and for 2D it's an ellipse. It's rotated when there's any covariance value in the matrix that isn't zero.

![Image]

That's easy to understand that this distance equation isn't generic enough for all input possibilities, since if any variance is zero for a dimension (that means that all presence points have exactly the same value in the dimension, and that all covariances calculated with that dimension is also zero), the determinant of the covariance matrix is null and it can't be inverted.

The maximum value of this distance in the hypercube is the distance between 2 opposite edges of it, but all possibilities should be verified since the distance is commonly different for each pair of opposite edges. An informal proof for that can be this: imagine that we have an hyperellipsoid centered in an edge of an hypercube (imagine in 2D and 3D first), now make it bigger without leaving the hypercube and stop exactly when it's getting outside the hypercube. For 2D that's something like:

![Image]

Rotation isn't allowed, and always the picture will have at least one edge of intersection with the hyperellipsoid. The point A is the center of the ellipse and the point B is the intersection point of the 2 shapes. Notice that A and B are opposite edges, but if we use another shape with the ellipse, it can be different:
If you try to use more dimensions, the same occurs with some edge. If you center the ellipse in another place inside the square (or do the same with the ndimensional world), the maximum distance will be in at least one edge too, so the maximum distance is always between 2 edges (not necessarily opposite for now).

Now it's possible to think in the distance as a linear transform to the input data, and after that it's possible to use the Euclidean distance to think. This transform in 2D creates a parallelogram, so the distance must be between 2 opposite edges. For 3D it's a generic parallelepiped (linear transformation of a cube result) and there goes. These shapes have always parallelograms as 2D sections that cuts 4 points of the ndimensional hypercube transformed, so the maximum distance must be between 2 opposite edges of the original hypercube.

Chebyshev distance (infinitenorm Minkowski distance; Chessboard distance)

All Minkowski distance equations have the format below, for any value of “k”, but here “k” tends to infinite:

\[
Distance = \lim_{k \to \infty} \left( \sum_{i=1}^{n} |p_i - q_i|^k \right)^{1/k}
\]

Here “i” represents a dimension counter. This equation can be easily reduced to:

\[
Distance = \max (|p_i - q_i|)
\]

Which can be calculated easily. Also, the maximum distance in the ndimensional world is always the range used in the data normalization.

For 2D data, the set of geometrical points that have a fixed distance to a particular point is a square with sides parallel to the axis. For 3D it's cube and in ndimension it's an hypercube.
Internal architecture

Overloaded methods

The algorithm extends from the algorithm implementation class directly, so there's nothing special to worry in the inheritance. The overloaded methods are explained here, trying to obey the order of execution that is used by openModeller. Only relevant contents for the architecture are exposed here, for more details it's possible to see the implementation that have several details that shouldn't be here.

The algorithm isn't iterative, there’s only 2 important methods that should be detailed for now. Before it starts, the data is normalized to be from DATA_MIN to DATA_MAX.

Method “initialize”

The data is verified and converted to be in a default structure. Here the layerCount (amount of dimensions), presenceCount (amount of presence points), the presence points and the average point (mean of all points) are initialized. There must exist at least one presence point in order to make the algorithm work.

After initializing the basic data, there's a method called InitDistanceType that is called. It's specific and will be seen later.

Method “getValue”

Here the parameter “n” (amount of nearest presence points to be used) is used. It's written in the parameter ParPointQnt. It calculates the distance of the first “n” points and writes it in a list with its index. After that each next value is verified with the following ones, and the biggest value is put away. At the end, it has all the “n” nearest points to calculate the mean. With the mean the distance between it and the getValue's parameter is calculated with the method Distance.
For \( n = 1 \) an optimized version of \( \text{getValue} \) is run, since all it needs to do is to find the nearest point and uses its distance to the parameter point. If \( n \) is equals to the amount of presence points given or it's an invalid value, another optimized version of \( \text{getValue} \) is run, since the distance is between the given point and the average point, already calculated. The value "0" (zero) is interesting here because it will always represent all points, no matter how many exists.

With the distance \( D \) (variable "dist"), the method calculates the probability using the equation already discussed, since \( X \) is written in "ParDist". The equation chosen has a number identification given by the parameter \( \text{ParDistType} \). The numbers can be seen during execution.

**Specific methods**

There are 3 specific methods in this algorithm, but one of them is specific for Mahalanobis distance and shouldn't be a concern. It's called "CalcCovarianceMatrix" and all it does is to initialize the variables "covMatrix" and "covMatrixInv". For this distance it was needed to use matrices, so a free C++ header with a full personalized matrix class was found and used. Don't worry about that, the relevant here are the 2 methods below:

**Method “InitDistanceType”**

It only continues what “initialize” started, but it's specific for the distance type. For Mahalanobis distance here is the place where the "CalcCovarianceMatrix" is called.

The maximum distance using the range as \([\text{DATA_MIN};\text{DATA_MAX}]\) is calculated here, for the chosen distance equation. With this information, the original parameter \( \text{ParDist} \), that was the “m” that's in \((\text{PARDISTMIN};\text{PARDISTMAX})\) goes to the interval \((0,1]\) as seen before as default values and then go to the interval \((0,\text{distMax}]\). In this last interval \( \text{ParDist} \) becomes exactly the \( X \) in the original equation.

**Method “Distance”**

Here 2 points are given and it's solicited to result a scalar number with the value of the distance calculated using the equation chosen in the parameter. This method must not be called before the “CalcCovarianceMatrix” when using the Mahalanobis distance. The equations of each distance were explained before and there's nothing besides that done here.

**Execution flow**

Here the flow of the main methods are explained. The algorithm class always gives the control back to the openModeller, so there's nothing special in this algorithm about the flow. A graphical model can show better the execution flow than words:
Steps 4m1 and 4m2 are called only when Mahalanobis distance is used; After step 7 openModeller call again step 6, for a new point, until it finishes the projection.
Annex 16
RELATÓRIO DE INICIAÇÃO CIENTÍFICA

TÍTULO: FERRAMENTA PARA ANÁLISE DE DADOS SOBRE DISTRIBUIÇÃO DE ESPÉCIES UTILIZANDO PROCESSAMENTO PARALELO

Orientando: Marcos Paulo Pereira Cabral dos Santos
Orientador: Profa. Dra. Liria Matsumoto Sato
Data: 09/02/2007

Instituição: Universidade de São Paulo
Escola Politécnica
Departamento de Engenharia de Computação e Sistemas Digitais
Índice

1- Introdução

2- Plano Inicial

3- Resumo das atividades realizadas
   3.1- Etapas cumpridas

4- Detalhamento das atividades realizadas
   4.1- Estudos de Programação paralela
   4.2- OpenMP
   4.3- O Openmodeller
   4.4- A estratégia de paralelização do OpenModeller
      4.4.1- Localização do trecho a ser paralelizado
      4.4.2- Paralelização
      4.4.3- Compilação e Depuração
      4.4.4- Testes e análise de desempenho

5- Etapas a serem cumpridas

6- Conclusão
1- Introdução

O presente documento consiste no primeiro relatório das atividades realizadas no projeto de Iniciação Científica no período compreendido entre 01 de setembro de 2006 e 07 de fevereiro de 2007. Durante este período foram realizadas as seguintes atividades: estudo do paralelismo e suas formas de implementação, estudo da programação orientada a objetos, familiarização com a ferramenta Openmodeller e análise e paralelização do código fonte do OpenModeller.

2- Plano Inicial

Os métodos mais usados para a predição de distribuição de espécies são baseados no conceito de nichos ecológicos [Peterson,2001; Anderson et al. 2002a,b] que combinam dados de ocorrência da espécie com as condições ambientais em cada ponto. Modelos de nichos podem ser projetados em dimensões geográficas para predizer onde a espécie em análise está presente. Contudo, tais métodos podem demandar um tempo excessivo de execução. Buscando reduzir o tempo de execução e consequentemente viabilizar a análise de distribuições que requerem tempo de processamento inviável, como também, permitir análises mais complexas com a aplicação de diversos algoritmos e considerações diversas sobre os dados ambientais, este projeto de iniciação científica pretende aplicar o conceito de paralelismo no módulo de projeção da versão corrente de uma ferramenta já disponível denominada Openmodeller. Serão apresentadas duas versões de paralelização do módulo de projeção do Openmodeller:

- Versão paralela utilizando o sistema OpenMP (http://www.openmp.org/drupal/mpdocuments/spec25.pdf) para um computador multiprocessador, podendo ser executado em um dos nós multiprocessadores do cluster. OpenMP é uma interface padrão de diretrizes para os compiladores da linguagem C que provê os recursos necessários para a paralelização de um programa para computadores multiprocessadores os quais contêm vários processadores e memória compartilhada. Implementações deste padrão encontram-se disponíveis gratuitamente.

- Versão paralela utilizando a biblioteca MPI [Snir,2006] para ser executado utilizando vários nós do cluster. MPI (Message Passing Interface) é um padrão de interface para comunicação entre processos distribuídos por passagem de mensagem.

O projeto será realizado em três etapas:

**Etapa 1: estudo e familiarização com o sistema Openmodeller**
A. Estudo e uso do sistema corrente
B. Estudo e análise do código do sistema openmodeller corrente
Etapa 2: desenvolvimento da versão paralela utilizando a interface OpenMP.
A. Definição da Estratégia de Paralelização
B. Implementação no código do Openmodeller
C. Depuração e testes

Etapa 3: desenvolvimento da versão paralela utilizando o sistema MPI
A. Estudo de programação e familiarização com o sistema MPI
B. Implementação usando a interface MPI, para cluster de computadores, tomando como base a estratégia de paralelismo definida na segunda etapa e utilizando o código fonte do Openmodeller
C. Depuração e Testes

Cronograma
Início: 1 de Setembro de 2006
Duração: 12 meses

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etapa 1</td>
<td></td>
<td></td>
<td>A</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etapa 2</td>
<td></td>
<td></td>
<td>A</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etapa 3</td>
<td></td>
<td></td>
<td>A</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Cabe a até o presente momento, apresentar a primeira versão de paralelização do Openmodeller.
3- Resumo das atividades realizadas

No início do projeto foram estudados os conceitos básicos de programação paralela e de programação orientada a objetos em C++ [Stroustrup's, 2000]. Depois de aprendidos os fundamentos do paralelismo, deuse início a um processo de familiarização com a biblioteca OpenMP e com a ferramenta de análise de distribuição de espécies, OpenModeler.

Na etapa seguinte, definiu-se a estratégia de paralelização do código, seguida de sua implementação e depuração.

3.1 Etapas cumpridas

Etapa 1: estudo e familiarização com o sistema Openmodeller
A. Estudo dos princípios de programação paralela, dos fundamentos de programação orientada a objetos em C++, familiarização com o OpenMP e uso do sistema corrente.
B. Estudo e análise do código do sistema openmodeller corrente.

Etapa 2: desenvolvimento da versão paralela utilizando a interface OpenMP.
A. Definição da Estratégia de Paralelização.
B. Implementação no código do Openmodeller.
C. Depuração e testes.

Cronograma
Início: 1 de Setembro de 2006
Duração: 12 meses

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etapa 1</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etapa 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

4- Detalhamento das atividades realizadas

Foram realizadas atividades de estudo e a implementação paralela do Openmodeller utilizando o sistema OpenMP.

4.1 Estudos de Programação paralela

Os primeiros conceitos de programação paralela a serem estudados foram os de Macrotasking, Microtasking e laços paralelos. Como método de aprendizagem, utilizou-se a resolução de pequenos problemas, como
multiplicação de matrizes e soma de seus elementos utilizando processamento paralelo. As soluções eram implementadas em linguagem C, com uso da biblioteca CPAR [Sato,1995], desenvolvida por gerações anteriores do Laboratório de Arquitetura e Programação de Alto Desempenho, LAHPCUSP. Foi possível de se implementar em CPAR: macrotarefas e microtarefas, laços paralelos e semáforos.

Depois de ganhar familiaridade com os conceitos de paralelismo, iniciou-se o estudo do OpenMP, biblioteca para multiprocessadores com memória compartilhada. Os mesmos problemas que foram resolvidos com CPAR foram resolvidos com OpenMP, sempre observando quais meios de implementação mudavam de uma biblioteca para outra.

Foram tópicos relevantes da aprendizagem do OpenMP: região paralela, laços paralelos, variáveis locais e compartilhadas, seção crítica e barreira.

Para que fosse possível a modificação do código do OpenModeller, foi necessário aprender programação orientada a objetos em C++, bem como a junção desta linha de programação com a de paralelismo.

Os conceitos fundamentais do paralelismo empregado estão brevemente descritos na próxima seção.

**Conceitos básicos de programação paralela**

Listam-se aqui os conceitos básicos de programação paralela para máquinas de memória compartilhada, isto é, máquinas cuja arquitetura permite que seus processadores possam escrever sobre uma região de memória comum [Sato, 1996].

- **Região paralela**: região do código que é executada simultaneamente por um conjunto de *threads*.
- **Macrotarefas**: módulos independentes de computação que podem ser executados paralelamente.
- **Microtarefas**: partes de código sequencial existente dentro de um laço, que podem ser executadas paralelamente. É possível encontrar microtarefas dentro de macrotarefas.
- **Laços paralelos**: laços cujas iterações podem ser divididas entre os *threads*.
- **Privado ou local**: atributo que cada processador cria em um espaço de memória exclusivo, isto é, onde os demais processadores não têm acesso.
- **Compartilhado**: atributo que todo processador tem acesso. As alterações feitas por um determinado *thread* podem ser vistas pelos demais.
- **Seção Crítica**: porção de código dentro da região paralela onde os processos realizam suas tarefas, um de cada vez.

Semáforo é uma variável inteira que só pode ter valores não negativos. São definidas exatamente duas operações com um semáforo S:
Wait (S): Se \( S > 0 \) então \( S := S - 1 \), senão suspenda a execução deste processo. O processo, então, é dito estar suspenso no semáforo \( S \).

Signal (S): Se há processos que foram suspensos neste semáforo, continue a execução de um deles e, senão \( S := S + 1 \).

- Um semáforo que só pode assumir valores não negativos é chamado de semáforo geral, o que só pode assumir valores 0 e 1 é chamado de semáforo binário e, neste caso Signal (S) é definido por: se ..., senão \( S := 1 \). [BenAri, 1948]
- Barreira: ponto do código que impede o avanço de um ou mais processos enquanto todos os demais processos não chegarem a ele.

### 4.2- OpenMP

Serão brevemente descritas as diretivas do OpenMP que se utilizam na paralelização do Openmodeller.

O OpenMP é uma biblioteca que suporta a programação paralela de memória compartilhada em todas as arquiteturas [www.openmp.org]. Está disponível para as linguagens C/C++ e Fortran, em plataformas do Unix e do Windows NT. A implementação do paralelismo em OpenMP se faz da seguinte forma:

A região paralela é implementada pela diretiva `#pragma omp parallel { região paralela}`.

Variáveis declaradas e objetos criados dentro da região paralela são tidos como locais. Variáveis declaradas anteriormente à região paralela devem ser especificadas como privadas ou compartilhadas logo em seguida da chamada da região paralela: `#pragma omp parallel private(variáveis locais separadas por vírgula) shared (variáveis compartilhadas separadas por vírgula)`. Objetos criados anteriormente à região paralela são compartilhados, enquanto aqueles criados dentro dela, são locais.

Laços paralelos são implementados através da diretiva `#pragma omp for` seguida do `for` a ser paralelizado. O total de iterações passa a ser dividido entre os processos.

Para se implementar uma seção crítica usa-se `#pragma omp critica { região critica}`, e para se impor uma barreira basta usar a diretiva `#pragma omp barrier`.

Quando se deseja definir o número de threads que estará presente numa determinada região paralela, chama-se a função `omp_set_num_threads` (`número de processos`). Caso não se especifique o número de threads, ele passa a ser o que está definido numa variável de ambiente. Caso contrário, o número de threads passa a ser o número de processadores da máquina. Para sabermos qual thread corrente está executando o código, usamos a função `omp_get_thread_num`. 


4.3- O Openmodeller

O Openmodeller é uma implementação de métodos para a predição de distribuição de espécies baseados no conceito de nichos ecológicos [SOURCEFORGE,2006]. Tais métodos combinam dados de ocorrência de uma determinada espécie com as condições ambientais em cada ponto. Eles tentam identificar, através de algoritmos existentes, quais pontos no espaço ambiental têm condições similares entre. Agrupados, estes pontos representam um modelo de nicho ecológico, dadas as dimensões ambientais consideradas. Desta forma pode-se se predizer onde a espécie poderá ou não manter populações, através das projeções destes nichos em dimensões geográficas.

Esta ferramenta de modelagem, escrita em linguagem C++ ANSI, recebe como parâmetros um conjunto de pontos de ocorrência (latitude e longitude) de uma determinada espécie e um conjunto de mapas de variáveis ambientais. Os algoritmos utilizados na versão corrente do Openmodeller são: Bioclim, Climate Space Model, GARP, Environmental Distance e Minimum Distance.

O funcionamento do software se dá em duas etapas: modelagem e projeção. Na primeira etapa combinam-se os dados de ocorrência da espécie com as condições ambientais de cada ponto para se obter, através dos algoritmos já citados, um modelo que representa a viabilidade da espécie sob determinadas condições ambientais. Na segunda, o modelo é projetado em dimensões geográficas para predizer onde a espécie poderá ou não manter populações.

Estão envolvidos no desenvolvimento do Openmodeller a Escola Politécnica da Universidade de São Paulo (EPUSP), o Instituto Nacional de Pesquisas Espaciais (INPE) e o Centro de Referência em Formação Ambiental (CRIA).

4.4- A estratégia de paralelização do OpenModeller

Foi paralelizado o módulo de projeção do Openmodeller, assim como estava previsto no plano inicial. O processo de paralelização se deu em etapas: localização do trecho a ser paralelizado, paralelização, compilação e depuração.

4.4.1- Localização do trecho a ser paralelizado

Inicialmente, fez-se uma visão panorâmica do código referente à etapa de projeção, que está estruturada em módulos e é compilado por partes automaticamente através de um comando makefile. Deuso maior atenção às regiões que continham laços longos, que são considerados como regiões que demorariam mais tempo de computação devido à sua complexidade e que seriam possíveis de serem paralelizadas.
Não foi encontrado algum laço longo com o número de iterações predeterminado (comando for). Todavia foi encontrado um laço — do tipo while — que tem como critério de parada uma comparação entre objetos, através da sobrecarga do operador binário de diferença, algo característico do polimorfismo existente na linguagem C++. Este laço está presente no arquivo Projector.cpp.

O trecho está apresentado a seguir:

```
MapIterator it = map->begin();
MapIterator fin;
    while( it != fin )
    {
        (...)
        ++it;
    }
```

Nota-se que, além da sobrecarga do operador diferença, foi usada, nesse trecho, a sobrecarga do operador de incremento préfixado.

### 4.4.2- Paralelização

A paralelização do while apresentada na seção 4.4.1 requisitou a proposta de uma solução não trivial, uma vez que a expressão de condição envolvia uma comparação entre dois objetos, sendo um deles iterado através do operador “++” préfixado. Além disso, era necessário garantir que cada thread fizesse uma operação com um objeto distinto.

Para resolver estes problemas primeiramente todo o laço foi aninhado por uma região paralela. Depois, foram criadas variáveis e objetos auxiliares e compartilhados entre as threads. A variável e o objeto que eram iterados continuaram sendo elementos locais. Assim, dentro de uma seção crítica, iteravase a variável e o objeto compartilhados e depois atribuía-se os valores atualizados aos componentes locais. Tais componentes continuavam na comparação do while como critério de parada. Desta forma, fez-se com que cada thread realizasse uma tarefa diferente e cada uma parasse quando seus componentes locais atendiam o critério de parada. Além disso, este procedimento permitiu que, se porventura uma thread fosse mais lenta que os demais, o número de operações realizado por ela seria menor. Também vale o dual dessa solução: se alguma thread fosse mais rápida ela realizaria mais operações.

Em continuidade da resolução do problema, para garantir o processamento de objetos distintos por cada thread, a variável e o objeto foram inicializados dentro de uma seção crítica. Foi necessário utilizar duas variáveis e objeto auxiliares do tipo compartilhado. Esta inicialização foi feita da seguinte maneira:

1- a variável auxiliar 1 é inicializada anteriormente à região paralela com o valor zero;
2- dentro da região paralela as threads executam segundo a seguinte sequência:

*Início da seção crítica:

se a variável auxiliar 1 tem o valor zero, a variável auxiliar 2 e a variável auxiliar 1 recebe 1, o objeto auxiliar tem os seus atributos atualizados com valores iniciais; senão, o valor da variável auxiliar 2 é incrementado assim como os atributos do objeto auxiliar variável e o objeto locais recebem respectivamente o valor da variável auxiliar 2 e os atributos do objeto auxiliar.

*Instala-se uma barreira.

*Fim da seção crítica.

O efeito desta lógica é fazer com que somente um processo, aquele que “chegar” primeiro, inicialize variável, os demais apenas teriam acesso ao valor através das variáveis compartilhadas.

Tanto a inicialização como a iteração são feitas em seções críticas e ocorre um certo aumento no número de operações, variáveis e objetos. Apesar disso, ganha-se tempo na computação global do laço, posto que quase todo seu restante é feito na forma paralela.

Mostra-se abaixo como ficaram as partes mais relevantes deste trecho. Observa-se que em (*) todo o corpo do laço é realizado paralelamente:

(...) Criação das variáveis auxiliares, inicialização de algumas delas e criação de um objeto auxiliar:

```c
int temp_it = 0 ;
MapIterator controle_it ;
int controle_pixels;
int temp_pixels = 0;
```

(...) Definição o número de processos e início da região paralela

```c
omp_set_num_threads(2);
#pragma omp parallel shared(temp_contador,temp_pixels,controle_contador, controle_pixels,pixelcount,pixelstep)
{
Início da primeira seção crítica
```
else
{
++controle_it;
controle_pixels++;
}

it=controle_it;
pixels=controle_pixels;

} Fim da seção crítica e imposição da barreira 
#pragma omp barrier

while( it != fin ) {
(*)

Início da segunda seção crítica

#pragma omp critical
{
controle_pixels++; 
pixels=controle_pixels;

++controle_it;

it=controle_it;
}
Fim da seção crítica e do laço.

} Um trecho do código no corpo do while (*) referente à escrita do resultado da projeção do modelo do objeto foi encerrado numa seção crítica.

Sample amb;
#pragma omp critical
{
Sample const &amb1 = env->get( lg, lt );
amb=amb1;
}
(...)
#pragma omp critical
{
if( amb.size() == 0 )
map->put(lg,lt);
else
map->put(lg,lt,val);
}

4.4.3-Compilação e Depuração

Na compilação foi utilizado o compilador icc da Intel que oferece a linguagem C++ e o OpenMP. Na depuração foram detectados e solucionados alguns
problemas, em particular aqueles referentes à necessidade de incluir parte do código em seção crítica.

4.4.4- Testes e análise de desempenho

A implementação e os testes foram realizados em um computador com um processador dual core com a finalidade de ser verificada a funcionalidade da implementação e de realizar uma análise preliminar do desempenho.

Em um primeiro teste, utilizando uma massa de dados pequena, verificou-se a funcionalidade da implementação com o paralelismo.

Contudo, em um segundo teste, realizado com a finalidade de se ter uma visão preliminar de desempenho obteve-se um tempo de execução maior do que o obtido sem a paralelização. Detectou-se que o ponto de gargalo é o trecho de código referente à gravação do resultado da projeção do modelo no objeto.

Uma solução, ainda em fase de estudo e proposição, se baseia em uma estratégia de escrita inicialmente em um buffer local em cada thread.

5- Etapas a serem cumpridas

Segue-se, nos próximos meses, como previsto, a continuidade da implementação utilizando OpenMP e a realização da terceira e última etapa do projeto de pesquisa: a implementação da paralelização do Openmodeler com a utilização de cluster, através dos recursos da biblioteca MPI, Message Passage Interface. Tal etapa será realizada em três passos:

A. Otimização, Depuração e Testes da implementação com o OpenMP;
B. Estudo de programação e familiarização com o sistema MPI
C. Implementação usando a interface MPI, para cluster de computadores, tomando como base a estratégia de paralelismo definida na segunda etapa e com o uso do código fonte do Openmodeller
D. Depuração e Testes

Cronograma

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6- Conclusão

Os estudos realizados sobre C++ e programação paralela proveram a fundamentação necessária para o desenvolvimento da implementação. Desenvolveu-se uma versão ainda preliminar utilizando o OpenMP, que
demonstrou a sua funcionalidade mas não apresentou um ganho de desempenho em relação a versão sequencial.

Uma versão apresentando maior ganho de desempenho deverá ser implementada visando a uma melhoria da performance do sistema.

Referências Bibliográficas


