Moebius: An interface to web map services

David Procházka*, Jana Procházková**
* Dep. of Informatics, Faculty of Business and Economics, Mendel University in Brno
** Dep. of Mathematics, Faculty of Mechanical Engineering, Brno University of Technology
`prochazka@pef.mendelu.cz`

Keywords: Indexing, Searching, Web Map Service

Abstract

Our article presents a concept of a geospatial search engine based on a Web Map Service (WMS) compliant virtual mapserver. This virtual mapserver is able to index mapservers based on the WMS standard and create an unified interface to all shared map layers. Our presented approach also allows to search the map layers within the virtual mapserver and process the results directly in GIS tools.

Introduction

We could recognize two basic approaches for retrieving some files or more generally a piece of information: searching and classification. Searching is a widely used method and is replacing the classification approach in many applications (for instance retrieval of a relevant web page). In geoinformatics however, ontological classification is dominantly used: metadata catalogs (http://mis.cenia.cz), semantic rules (see [1], [2]), etc. Although these methods have some benefits, they also have many drawbacks: 1. Catalogues and other ontologically based approaches require manual administration (delays in actualization, limited range, etc.). 2. It is hard to classify geodata into a fix set of categories because on every layer is possible to look from many aspects (origin, resolution, coverage, content). For an overview of currently used approaches see [3] or [4].

Generally, these approaches are not solving the basic problem: geodata is spread across the Internet on many mapservers and it is usually a preliminary problem to find these mapservers, for this reason, there is a need for a geospatial version of a search service such as Google (http://www.google.com), Jyxo (http://www.jyxo.cz) or similar engines. Nowadays geodata is usually published through different map services, therefore we have focused on them. The presented search engine is using OpenGIS standards for communication, especially the *Web Map Service* (see [5]).

Geospatial search service

The presented solution has three basic parts: First an indexing engine to find as many mapservers as possible. Indexing must be as autonomous as possible because manual administration would quickly become the bottleneck of the engine. Secondly it is necessary to create a unified interface to them. For this purpose we have designed a virtualization engine. The third part is a search engine working with given indices. Such engine must be very simple and intuitive (e.g. like Google). The following sections present the basic structure of these components.

Indexing tool

Indexing tool (called *Indexer*) is a web service written in Python. To start the *Indexer* it must be given the address of the indexed mapserver. It then sends a *GetCapabilities* request to the given mapserver and decomposes the resulting XML file. Pieces of information connected directly to the map layer (bounding box, name, title, ...) are integrated with information valid for more layers. For instance the *Abstract* of the mapset (or the mapserver itself) is valid for all layers in the mapset. Therefore the index of each layer must contain this information. Result of this process is an index with following structure.

- **NickName** – unique identification string, composition of the name of the map layer (unique within the mapserver) and unique identification of a mapserver (it is chosen during indexing, usually it is part of URL – result is e. g. *lakes@mapserver.water.gov*),
- **Name** – name of the layer (content of name element),
- **WMS** – version of WMS supported by the mapserver, taken from the head of the *GetCapabilities* file,
- **Address** – URL of the mapserver where the layer is stored, hence it is also the URL used for the *GetMap* requests, again taken from the head of the *GetCapabilities* file,
- **Access** – access mode to the layer, there are three options: all (everyone is able to access this layer), black (everyone except for users from IP addresses on a blacklist), white (users from IP addresses on a whitelist only), used for security reasons,
- **Descriptions** – contain the content of *Title* elements in *Layer* elements (description of the layer or mapset) and usually also from the head of the *GetCapabilities* file,
- **Abstracts** – list of *Abstracts* taken from the head of the *GetCapabilities* file and instances of *Layer* element,
- **SRSs** – list of supported coordinate systems,
- **BoundingBoxes** and **LatLonBoundingBox** – define the bounding box of the layer,
- **MinScale** and **MaxScale** – maximal and minimal scale of the layer, taken from the lowest instance of *Layer* element (could be replaced or extended by *ScaleHint* element),
- **Formats** – list of supported output formats, taken from the head of the *GetCapabilities* file,

- **Opaque** and **Queriable** – same meaning as in *GetCapabilities* file, values are taken from the lowest instance of *Layer* element,
- **Styles** – list of *Styles* – names of *NamedStyles* known to the WMS and appliable to this layer.

All described pieces of metadata are given by the WMS itself. As there exists no unified metadata system for geospatial data, it is not possible to rely on information stored in metadata files using different formats. But there are enough different elements in *GetCapabilities* documents to provide complex information about a map layer. The basic problem is that these elements are frequently not used. Abstracts and descriptions are very brief, information about supported resolutions, accuracy, etc. is usually completely missing. From our point of view the situation is slowly getting better, but there is still enough place for improvements.

Currently it is necessary to pass the URL of some mapserver to the *Indexer*. Appropriate indices are created automatically. For higher performance the indexing tool should be accompanied by some kind of a web crawler for automatic mapserver discovery as described in [6].

It is necessary to emphasize that the contents of the indices have to be checked periodically. There are two possible control approaches. The first one checks just the existence of the layer. This could frequently be done by a *GetMap* request on some small part of the layer. The second approach could be called ”reindexing”: If a newly created index entry matches to an old one, the contents must be the same or otherwise it is necessary to replace the entry.

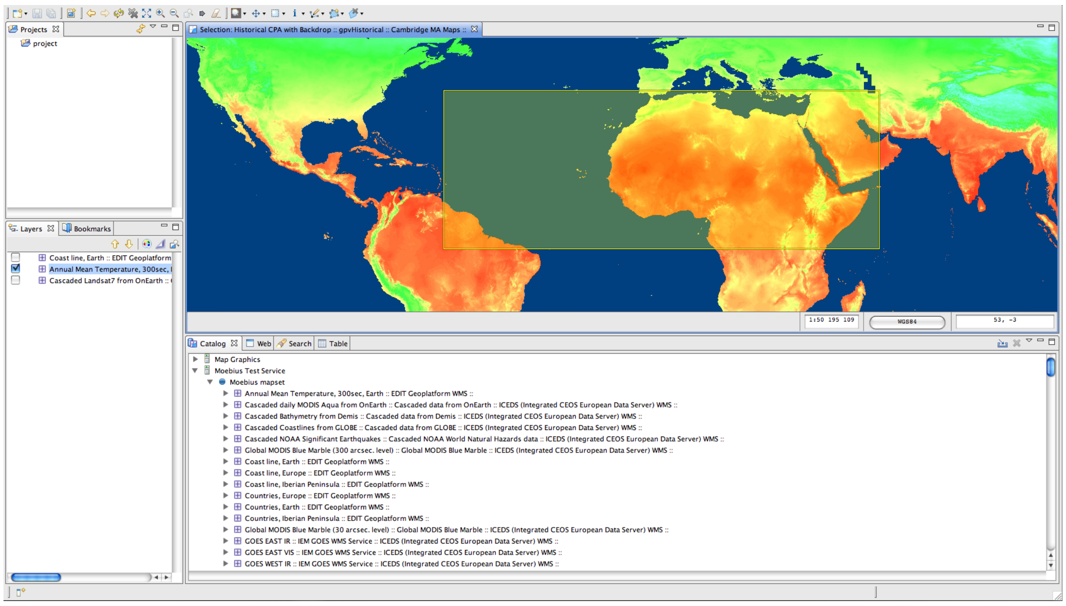
Virtualization tool

There are many approaches in virtualization (or rather aggregation) of web services. Probably the most successful projects are GIDB [6] and *GeoBrain* [7], [8]. In our project a different approach is used. The concept is described in more detail in [4]. The standard “old-style” approach is to create lists of mapserver URL or create a WMS interface to them (GIDB). But still we have a number of different mapservers. In our approach we are merging layers from all indexed layers together into one huge virtual mapserver. Such a mapserver contains no data, the virtual layers are generated from the indices stored in the database.

It is obvious, that every WMS compliant mapserver must be able to respond on *GetMap* and *GetCapabilities* requests. Following section describes the implementation of these requests in our virtual mapserver called *Moebius*.

GetCapabilies

The implementation of the *GetCapabilities* request is straight forward: The *Moebius* has the indices that contain all information necessary for generation of the *GetCapabilities* (GC) documents. Therefore the response is in fact a translation of the indices into a GC file. The first part of the GC file contains information about the *Moebius* (supported formats, address of the service, contact information, etc.). These information is stored in a configuration file. The second part is generated by the translation method. Indices are stored in current implementation in an XML, therefore an parser generates just slightly different XML tree according to the GC DTD.


Projects
project
Layers
Bookmarks
Coast line, Earth :: EDIT Geoplatform
Annual Mean Temperature, 300sec,
Cascaded Landsat7 from OnEarth ::
Selection: Historical CPA with Backdrop :: gpvHistorical :: Cambridge MA Maps ::
1:50 195 109
WGS84
53, -3
Catalog
Web
Search
Table

Map Graphics
Moebius Test Service
Moebius mapset
Annual Mean Temperature, 300sec, Earth :: EDIT Geoplatform WMS ::
Cascaded daily MODIS Aqua from OnEarth :: Cascaded data from OnEarth :: ICEDS (Integrated CEOS European Data Server) WMS ::
Cascaded Bathymetry from Demis :: Cascaded data from Demis :: ICEDS (Integrated CEOS European Data Server) WMS ::
Cascaded Coastlines from GLOBE :: Cascaded data from GLOBE :: ICEDS (Integrated CEOS European Data Server) WMS ::
Cascaded NOAA Significant Earthquakes :: Cascaded NOAA World Natural Hazards data :: ICEDS (Integrated CEOS European Data Server) WMS ::
Global MODIS Blue Marble (300 arcsec. level) :: Global MODIS Blue Marble :: ICEDS (Integrated CEOS European Data Server) WMS ::
Coast line, Earth :: EDIT Geoplatform WMS ::
Coast line, Europe :: EDIT Geoplatform WMS ::
Coast line, Iberian Peninsula :: EDIT Geoplatform WMS ::
Countries, Europe :: EDIT Geoplatform WMS ::
Countries, Earth :: EDIT Geoplatform WMS ::
Countries, Iberian Peninsula :: EDIT Geoplatform WMS ::
Global MODIS Blue Marble (30 arcsec. level) :: Global MODIS Blue Marble :: ICEDS (Integrated CEOS European Data Server) WMS ::
GOES EAST IR :: IEM GOES WMS Service :: ICEDS (Integrated CEOS European Data Server) WMS ::
GOES EAST VIS :: IEM GOES WMS Service :: ICEDS (Integrated CEOS European Data Server) WMS ::
GOES WEST IR :: IEM GOES WMS Service :: ICEDS (Integrated CEOS European Data Server) WMS ::

Figure 1: uDig application with our virtual mapserver Moebius. In the bottom window is displayed the content of the Moebius map service.

GetMap

Every *GetMap* request must be decomposed according to the number of requested layers. For every requested layer is a new *GetMap* request executed. This request is sent to the real mapserver. The response – an image – is stored by the *Moebius*. After the mapservers have returned all requested images, the *Moebius* merges them into one. This image is returned to the client. It is obvious that the client does not know that it is in fact receiving data originating from different mapservers (see Fig. 2).

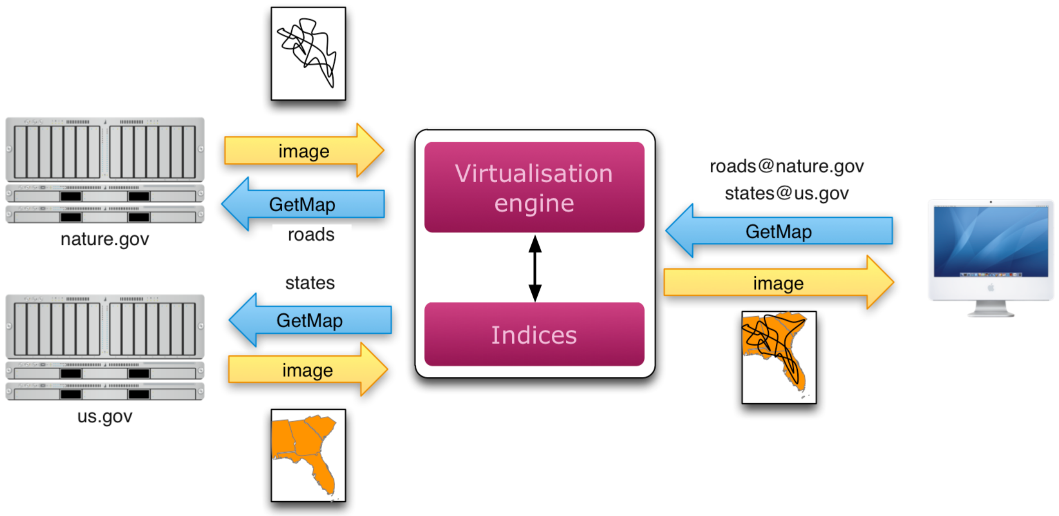


Figure 2: Scheme of the GetMap request implementation in the Moebius

An example using layers from two mapservers:

example URL[1]

and the result is the following single image (Fig. 3).

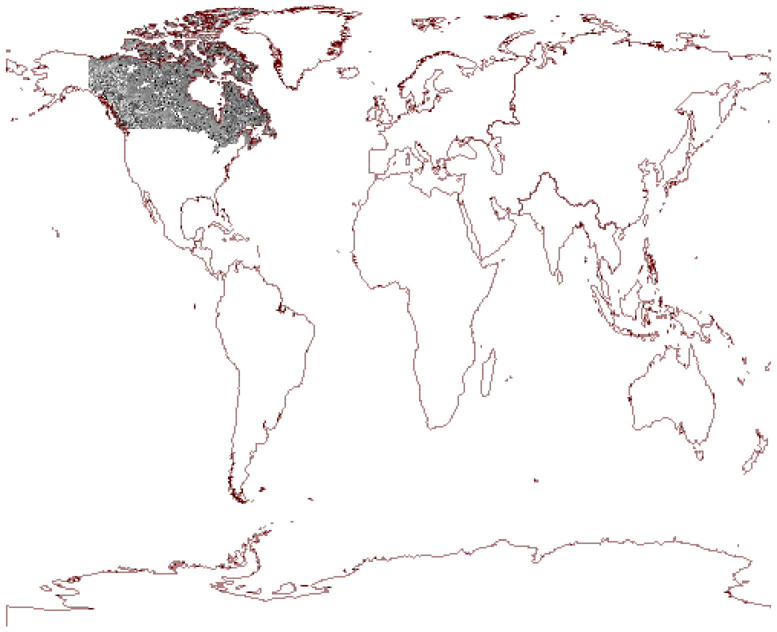


Figure 3: Example of a GetMap request with layers from two different mapservers.

Search engine

A web page is a common approach for searching the web. This approach, however usually effective, is inconvenient in this situation. Let us suppose that a user formulates a question and receives an answer in form of some list of links. There is an significant disadvantage in such a response: In case the user wants to add some layers into his project in a GIS, it will be necessary to copy the addresses of mapservers, names of the layers, etc. Therefore we have designed a completely different solution.

The basic idea of our approach is: if *GetCapabilities* means "return all available layers", there should be an another request *FindMap* which means "return me layers which fulfil given criteria". **The response on such a request should be again a *GetCapabilities* file**, just with limited amount of layers. This approach allows to process the response directly in a GIS application because every response is in fact from the GIS application point of view an independent WMS mapserver.

Structure of FindMap request

[1]`http://echo.mendelu.cz/cgi-bin/moebius/moebius.py?service=wms&version=1.1.1&request=g \`
`etmap&layers=topp:tdwg_level_1@edit3.csic.es,Radarsat_1000@cgkn.net&srs=EPS \`
`G:4326&bbox=-180,-90,180,90&styles=&format=image/png&width=500&height=400&`

The *FindMap* request is similar to other WMS requests. Parameters allow the user to formulate a question for what he is searching for and where it should be. This can be done using the following attributes:

- **request=FindMap** – identification of the request, should be mandatory or optional (depend on implementation of the service),
- **words=keyword,keyword,...** – list of keywords which are searched in the indices, mandatory,
- **bbox=minx,miny,maxx,maxy** – bounding box for searching, mandatory,
- **operator=and,or** – defines relation between keywords, optional (default value is "or"),
- **version=1.0.0** – version of request, currently not used, just for the future development,
- **exceptions=exception_format** – defines format of exceptions, optional,
- **abstract=0..n** – number from 0 to n which represents the significance of instances of keywords in this part of the index (0 – abstract is not used in the calculation, n – abstract has the highest significance),

Example of such a *FindMap* request is:

example URL[2]

The response is an appropriate part of the *GetCapabilities* document of the *Moebius* with layers for a given part of China.

From the request (especially the *bounding box* part) it is obvious that user assumes that there exists only one place called "Three Gores" and that he does't know where this place is. Therefore he is searching the whole Earth.

More usual is the second application of this service, where the user is searching on some specific part of the Earth. For example: If a user is searching for the coast of Iberian peninsula, it is possible to search for keywords "coast" and "Iberian"/"Iberian peninsula" on the whole Earth or for "coast" just above the appropriate peninsula.

1. example URL[3]
2. example URL[4]

The second approach is obviously much more effective. On a mapserver in Spain or Portugal there will be probably a layer called "coast", but it is much less probable, that this layer will be called "Iberian coast". Moreover the coast looked for could be part of a greater layer – e. g. a European coast layer. It is obvious that in this case searching for "Iberian coast" is ineffective.

Search method and calculation of the relevance

[2] `http://echo.mendelu.cz/cgi-bin/moebius/search.py?words=Three,Gorges&operator=and&bbox \ =-180,-90,180,90`

[3] `http://echo.mendelu.cz/cgi-bin/moebius/search.py?words=iberian,coast&operator=and&bbo \ x=-180,-90,180,90`

[4] `http://echo.mendelu.cz/cgi-bin/moebius/search.py?words=coast&bbox=-16,42,10,36`

Probably the widest spread method used for searching is *Inverted Index* (see [9], [10] and many others). A great advantage of this approach is its simplicity. *Inverted Index* method is used by *Google* and many other search engines. The gist is building records which contain touples – a keyword and its instances in documents (e.g. golf is in documents 1, 7 and 9). Usually the records also contain information about position or positions of the keyword in the document (golf appears in document 1 on positions 7, 25 and 78). This method is frequently extended with a thesaurus, dictionary and other improvements. Important for implementation is, that these improvements can be added independently.

Inverted Index based methods do not reflect the semantic meaning of documents. If a user is searching for the word "Golf", they are only able to find documents containing this word. Usually they are not able to recognize the difference between golf (sport) and Volkswagen Golf (car). Some engines (such as Google) are using the history to guess the semantic meaning of a question (e.g. a user is usually asking for information about cars). But what if there is a page about Tiger Woods which is in fact about the golf sport, but does not contain the word golf itself? *Inverted Index* methods are usually not able to recognize it. Therefore there is a need for a more complex method which is able to work with semantic relations.

Latent Semantic Analysis

Important method based on analysis of the semantic meaning is *Latent Semantic Analysis* (LSA), also called *Latent Semantic Indexing.* LSA is a technique in natural language processing for analyzing relationships between a set of documents by producing a set of concepts related to the documents and terms they contain. LSA is based only on mathematical principles and does not use any indices or keywords. Important advantage is that a similar document must not contain a given keyword (see [11]) and can still be found.

The input of the algorithm is a set of different documents and one document which contains the keywords. LSA will find the documents which are close to given keywords.

LSA can use a term-document matrix which describes the occurrences of terms in documents. It is a sparse matrix where the rows correspond to terms (typically stemmed words that appear in the documents) and the columns correspond to documents.

$$X = \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix} \quad (1)$$

A typical example of the weighting of the elements of the matrix is *Inverse Document Frequency* (IDF). The element of the matrix is proportional to the number of times the terms appear in each document, where rare terms are upweighted to reflect their relative importance. The next step is applying mathematical algorithm *Singular Value Decomposition* – SVD (for mathematical background see [12] or [13]) The output is the product of three special matrices:

$$X = K \ \cdot S \ \cdot D^T \ (2)$$

Matrix K contains the eigenvectors $\mathbf{u}_i$ of XX^T (in columns), D^T is the matrix of the eigenvectors $\mathbf{v}_i$ of X^TX (in rows). Matrix S is composed of square root of singular values, which are written in descending order on the main diagonal.

It turns out that when you select the s largest singular values, and their corresponding singular vectors from K and D^T, you get the rank s approximation to X with the smallest error (Frobenius norm). This approximation translates the term and document vectors into a concept space. Equation (2) can be rewritten as:

$$M_s = K_s \cdot S_s \cdot D_s^T, \ (3)$$

The last step is to find, which documents are close to the given query (view this as a mini document). To do the latter, we must first translate our query $\mathbf{q}$ into the concept space – $\mathbf{q}^*$. It is obvious that we must use the same transformation that we use on our documents (SVD transformation). Then we compare it to our documents (vectors $\mathbf{v}_i$) using cosine similarity.

$$\cos(\mathbf{q}^*, \mathbf{v}_i) = \frac{\mathbf{q}^* \cdot (D_s^T)_i}{|\mathbf{q}^*| \cdot |(D_s^T)_i|} \ (4)$$

The result of the described equation always lies in the interval $\langle 0, 1 \rangle$ – property of cosine function. The result near zero shows that there is no similarity between query and the document. A value near one shows that there is a high similarity, hence we have probably found a relevant result.

This approach is very promising. Although it is necessary to comprehend and implement many mathematical algorithms, the results outweigh the difficulties. For the presented approach to create accurate results at least few larger sentences which describe the content are needed. Nowadays descriptions of layers contain usually only a few words, and hence it is not possible to use LSA efficiently right now. It is necessary to wait until owners of mapservers are publishing more complete and precise meta-data. For the time being it is necessary to use an algorithm which is able to work with less information.

Implemented method

Our approach is a combination between *Inverted Index* and LSA principles. The implemented search engine is using indices created by the indexing tool. Relevance is based on number of instances of searched keywords and their position in the indices.

Let for all elements of index e_i, $i = 0, 1, \ldots, n-1$ (*Abstracts*, *Titles*, ...) exists a coefficient of the importance of the element w_i. Coefficient w_i starts with value zero and is increased by one with every keyword k_s found in the element. This calculation of the importance is done for all elements e_i.

Furthermore for every element e_i is defined coefficient v_i. This is the weight of of the element. The weight is designed to emphasize the important elements such as *Keywords*. For instance if a keyword appears in the element *Keywords*, it is more important than its appearance in element *Abstract*. Hence every element has its weight given by default settings of the engine or by the user as a parameter of the FindMap request. These two coefficients are used for calculation of the **importance of the layer**:

$$W = \prod_{i=0, w_i \neq 0}^{n-1} w_i . v_i$$

where

$$bin(k_s, e_i) = \begin{cases} 1, & \text{keyword } k_s \text{ is in element } e_i \\ 0, & \text{keyword } k_s \text{ is not in element } e_i \end{cases}$$

and coefficient of the importance w_i for element e_i is given by:

$$w_i = \sum_{\forall s} bin(k_s, e_i)$$

The calculation is based on following assumption: If in some element more keywords appears, it is more probable that this layer is relevant. If there are more elements containing more keywords, relevance is much higher. From this reason values of the non-zero coefficients are multiplied.

The second important coefficient which is used for calculation of the relevance is the **coefficient of instances**. The number of instances of searched keyword k_s in element e_i is called a_i. We calculate the sum m_s of these instances for every keyword k_s:

$$m_s = \sum_{i=0}^{n-1} a_i$$

In case there is an operator "and" between keywords and there exists at least one $m_s = 0$, is coefficient of instances for that layer set to zero. In all other cases is the coefficient given by equation

$$M = \sum_{\forall s} m_s$$

The value which represents the **relevance of a layer** – R – is calculated by multiplying the coefficients W and M presented above.

$$R = W \cdot M$$

This formula reflects the thought that important is not only the number of instances of the keywords, but also their position in the index and their proximity.

Conclusion and further development

The key innovations of the presented approach are the virtualization of multiple Web Map Servers and the method of searching. It is necesary to empathize that the virtualization engine creates a single WMS compliant interface to all mapservers. Hence the virtual mapserver *Moebius* can be opened in every GIS tool that connects to OGC WMS. The *FindMap* request which is embedded into the *Moebius* allows to process search results directly in GIS applications. This was done by selecting the *GetCapabilities* document language as the output format of the search results. Moreover, because this *GetCapabilities* document is an ordinary XML document, it could be transformed into any other XML based format – XHTML, KML, etc. This allows to process the results in many more ways.

Currently we are developing an extension to Moebius which transforms the *GetCapabilities* files into *Keyhole Markup Language* (KML, for description see [14]). Therefore it is possible to load virtually any WMS mapserver (or the search result) in Google Earth. An example of such a translated search result follows. The KML document itself contains no data. All geodata is loaded on demand using the Moebius WMS. The generated KML supports the *Super-Overlay* technology (see [14]).

Great challenge which is before us is the optimization of the ranking algorithm. The currently used approach is very simple. We are working on development of a rank for every layer that could present its reliability (similar meaning as *pagerank* has [15]). It will be based on observing the usage of different layers (frequently used layers are probably more relevant). Extension of our algorithm with such a rank could significantly improve the search results.

Although we have a working proof of our concept, there must be done a lot of work before this application can be used for everyday work. Currently we are experimenting with new solution for storage of the indices and we are trying to remove the performance bottlenecks. Source codes of our solution written in Python and further information are available on http://echo.mendelu.cz, where you can also find more examples and further information. If you are interested in this project, do not hesitate to contact us.

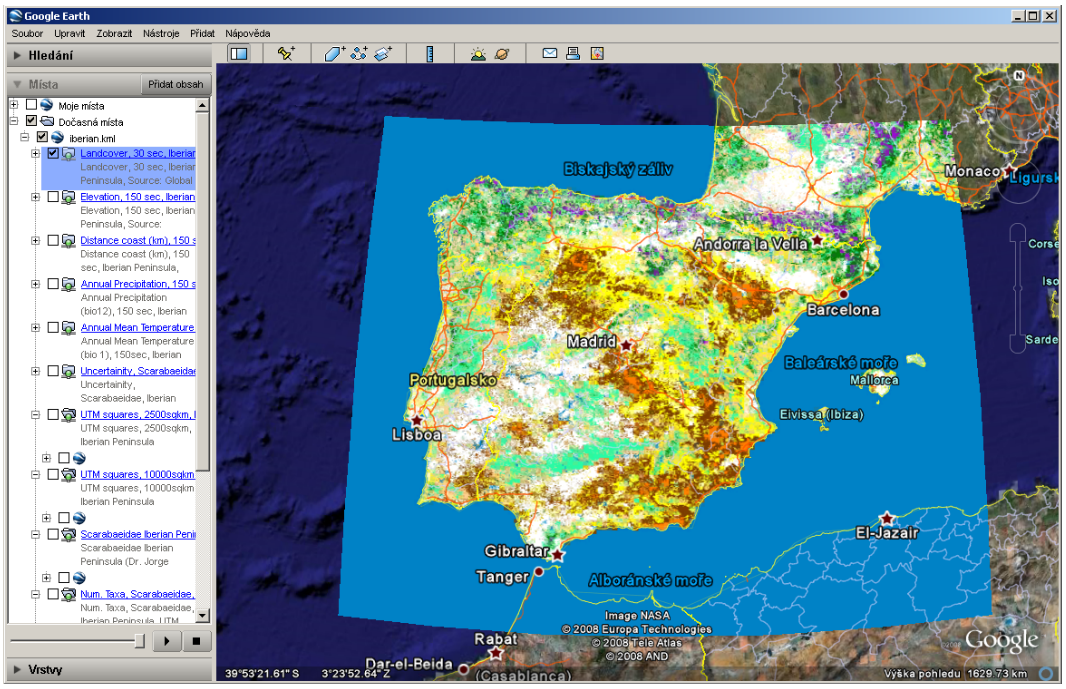

Google Earth
Soubor Upravit Zobrazit Nástroje Přidat Nápověda
Hledání
Místa
Přidat obsah
Moje místa
Dočasná místa
iberian.kml
Landcover, 30 sec, Iberian Peninsula, Source: Global
Elevation, 150 sec, Iberian Peninsula, Source:
Distance coast (km), 150 sec, Iberian Peninsula,
Annual Precipitation (bio12), 150 sec, Iberian
Annual Mean Temperature (bio 1), 150sec, Iberian
Uncertainty, Scarabaeidae, Iberian
UTM squares, 2500sqkm, Iberian Peninsula
UTM squares, 10000sqkm, Iberian Peninsula
Scarabaeidae Iberian Peninsula (Dr. Jorge
Num. Taxa, Scarabaeidae, Iberian Peninsula, UTM
Vrstvy
Biskajský záliv
Monaco
Andorra la Vella
Barcelona
Madrid
Portugalsko
Baleárské moře
Mallorca
Eivissa (Ibiza)
Lisboa
El-Jazair
Gibraltar
Tanger
Alboránské moře
Rabat
Dar-el-Beida (Casablanca)
Image NASA
© 2008 Europa Technologies
© 2008 Tele Atlas
© 2008 AND
Google
39°53'21.61" S 3°23'52.64" Z
Výška pohledu 1629.73 km

Figure 4: Google Earth application with opened KML file with the results of the search

References

1. Cruz, I. F. et al. Handling semantic heterogenities using declarative agreements. In *GIS '02: Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pp. 168–174, ACM Press, New York, NY, USA, 2002.

2. Wiegand, N. et al. A web query system for heterogeneous government data. In *Proceedings of the 2004 annual national conference on Digital government research*. Digital Government Research Center, 2004.

3. Procházka, D. Modelování a vizualizace vymezeného geografického prostoru (Ph.D. Thesis). MUAF in Brno, Brno, 2008, online[5].

4. Procházka, D. Motyčka, A. Geospatial Search Service. In *Collaboration, software and services in information society*, Ljubljana, Slovenija, 2008.

[5] http://echo.mendelu.cz/disertace.pdf

5. De La Beaujardiere, J. OpenGIS Web Map Server Specification Implementation, 2007, online[6].

6. Sample, J. et al. Enhancing the US Navy's GIDB Portal with Web Services. In *Internet Computing*, IEEE. Sept.-Oct. 2006, 10, 5, pp. 53–60.

7. Zhao, P. – Di, L. Semantic Web Service Based Geospatial Knowledge Discovery. In *IEEE International Conference on Geoscience and Remote Sensing Symposium 2006.* 2006, pp. 3490–3493.

8. Yue, P. et all Semantic Augmentations for Geospatial Catalogue Service. In *IEEE International Conference on Geoscience and Remote Sensing Symposium 2006.* 2006, pp. 3486-3489.

9. Manning, Ch. D. Raghavan, P. Schütze, H. Introduction to Information Retrieval. Cambridge University Press, Cambridge, MA, 2008, online[7].

10. Black, P. E. Inverted index. In Dictionary of Algorithms and Data Structures, U.S. National Institute of Standards and Technology, 2008, online[8].

11. Yu, C. Cuadrado, J. Ceglowski, M. Payne, J. S. Patterns in Unstructured Data – Discovery, Aggregation, and Visualization. National Institute for Technology and Liberal Education (NITLE), 2008, online[9].

12. Aggarwal, C.C. Yu, P.S. On effective conceptual indexing and similarity search in text data. In *Proceedings of the IEEE International Conference on Data Mining*, 2007, pp. 3-10.

13. Wall, M. E. Rechtsteiner, A. and Rocha, L. M. A Practical Approach to Microarray Data Analysis. Kluwel, Norwell, MA, 2003.

14. Google, Inc. Keyhole Markup Language Introduction. Mountain View, CA, 2008, online[10].

15. Page, L. Brin, S. Motwani, R. Winograd, T. The PageRank Citation Ranking: Bringing Order to the Web. Stanford Univeristy, 1999, online[11].

Acknowledgement

This article was written in context of project **VZ MSM 6215648904/03/03/01** – Ministry of Education, Youth and Sports of the Czech Republic.

[6] `http://www.opengeospatial.org/standards/wms`
[7] `http://nlp.stanford.edu/IR-book/html/htmledition/irbook.html`
[8] `http://www.nist.gov/dads/HTML/invertedIndex.html`
[9] `http://www.knowledgesearch.org/lsi/cover_page.htm`
[10] `http://code.google.com/apis/kml/documentation/`
[11] `http://dbpubs.stanford.edu:8090/pub/1999-66`