THE ACTIVE GUIDEBOOK

Information retrieval by keyword and location

Trevor Boyd & Peter Robinson
University of Cambridge, Computer Laboratory

Abstract The active guide book is a context-aware information management system that uses a combination of spatial and keyword indexing to retrieve data. The system has three principal components:

- A new document description language extends HTML to include facilities for tagging with spatial locations.
- Retrieval uses two separate indexes – a segment tree is used for spatial indexing and an inverted file is used for keyword indexing.
- A user interface allows queries involving keywords and location data to be expressed, and presents their results.

The system has been evaluated with the implementation of an interactive guidebook. The test data was drawn from existing Web pages describing the City of Cambridge in England, which were augmented with spatial information. A GPS system is used to provide the default location information for retrieval, but can be overridden with explicit coordinates.

Keywords: Context-aware applications, mobility, location.

1. INTRODUCTION

Improvements in communications bring the benefits of distributed computing to mobile users. Moreover, cellular wireless systems can identify the user's location with considerable precision. This paper considers the use of position as part of an information retrieval system. An experimental system has been built to investigate the combined use of spatial and keyword indexing to retrieve data.

Traditional guidebooks present text and pictures with an alphabetical index of keywords and using maps as the sole source of spatial information. It would be better to allow more flexible indexing of material that included audio and video content.

The *active guidebook* presents tourists with a system that guides them around a new city and provides useful information on local landmarks and places of interest. This guidebook uses a positioning system to enable it to display the user's current location and tailor information accordingly. For example, relevant information can be automatically displayed as the user walks past an historic building.

The system runs on a portable unit, equipped with a Global Positioning System (GPS) or equivalent module that identifies the location of the user. This allows the information presented to be keyed to the location of the user. Content can be taken from any suitable source, such as a conventional tourist guide or continuous media from CD-ROM or the Internet. This is indexed and annotated with positional information and presented to the user.

The information to be stored and presented will contain both spatial and context information and it must be possible to retrieve information in a variety of ways. Most importantly, it is necessary to enable the efficient retrieval of information by its spatial reference to exploit the system's knowledge of the user's current position.

Spatial data differs in important ways from normal keyword data. This project demonstrates how the two can be combined in a retrieval system providing a uniform interface to the data.

Others have considered the problem of content and presentation for electronic tourist guides [1, 5]. We focus on the problems of indexing and retrieval.

2. DESIGN

The system draws on the work of Professor Peter Brown and others at the University of Kent on context-sensitive information retrieval [2, 3, 7]. His *Stick-e notes* associate a context with pieces of information and retrieval is triggered when the user's current context matches that associated with the information. The data files in this system are also referred to as *notes.*

The focus of this work is on retrieval rather than content generation, so notes are stored simply as Hypertext Markup Language (HTML). This allows the inclusion of different media types in the notes, and also allows them to be displayed easily using a conventional web browser. It also allows existing publications to be reused for content and to write new notes without any specialist skills. Guidebooks consist of a collection of HTML notes and several extra index files, which can be generated quickly within the system.

Each note's context consists of two separate components: spatial location and keywords. Following Brown's suggestion [2], two separate structures are used to index the data.

Different data structures were needed to allow efficient searching and retrieval of notes in these two indices. The spatial index uses a *segment tree* for

fast and efficient retrieval of notes covering a given point. The keyword index uses an inverted file structure. These are described in more detail in below.

A tree of filters is used to sift the notes, with each filter implementing either a restriction on keyword or location, or combining two streams through a Boolean operation.

2.1. Document Structure

The HTML notes are augmented with meta tags to store keywords and values, along with a type field for the value. The work from the University of Kent proposes using Standard Generalised Mark-up Language (SGML) to represent the context associated with a given document, but this was considered unnecessarily complex for this application and HTML is sufficient.

Brown also proposes designing an object-oriented model for the values and information stored in the documents [3]. While this is a very elegant system, it is also unnecessarily complicated for this application.

The context information is stored as `meta` tags in the head section of the HTML. These take the form:

```
<meta name = "..." value = "..." type = "...">
```

We refer to the contents of the name field as the *keyword* and those of the `value` field as the *value*. Together, these make up a (keyword, value) pair.

A variety of data types are provided for information in the notes. These are integers, strings, point locations and lines. The `meta` tags include an explicit `type` field for each keyword. This was designed to allow easy reading of the files by the system, as types do not need to be inferred.

By convention, spatial information uses the keyword `location` as this presents a consistent naming convention for the whole system.

The process of creating notes is simplified by the provision of a special editor to facilitate the addition of the `meta` tags to existing HTML documents.

2.2. Spatial Index – Segment Tree

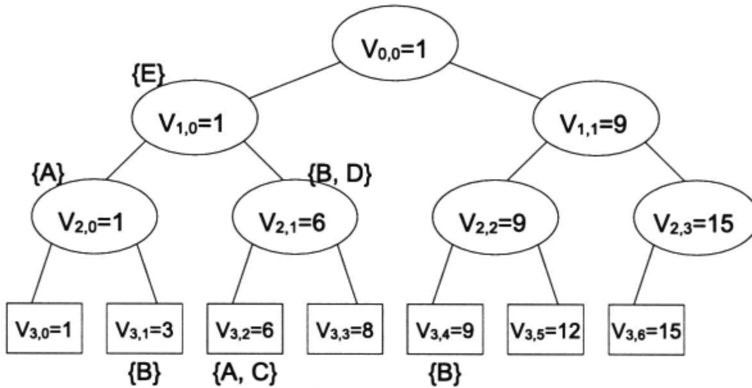
Each location has an associated *area of interest*, an arbitrarily shaped zone surrounding the object. A similar, circular zone surrounds the user's current position. The retrieval system must identify any objects whose areas of interest overlap that of the user.

Areas of interest are stored as sets of disjoint rectangles. Each rectangle is stored in a pair of data structures, one recording horizontal extents and one vertical. *Segment trees* are used to store the intervals corresponding to these extents.

A segment tree [8, 10] is a binary tree used to store sets of intervals. Each leaf node represents an interval. For example, consider the following set of intervals:

```
A. 1 - 8
B. 3 - 12
C. 6 - 8
D. 6 - 9
E. 1 - 9
```

The segment tree corresponding to this set would be as follows:



In this structure, node $n_{l,\,x}$ is at level l (where level 0 is the top of the tree and level *leaf* is the leaf node level) and horizontal position x (where x runs from 0 to $2l - 1$). The two children (if they exist) of node $n_{l,\,x}$ are the nodes $n_{l+1,\,2x}$ and $n_{l+1,\,2x+1}$. $v_{l,\,x}$ is the value stored in node $n_{l,\,x}$. Each leaf node $n_{leaf,\,x}$ is marked with an integer and the node is deemed to span the interval $[v_{leaf,x}, v_{leaf,x+1})$, where $[a, b)$ is the half-open interval $\{ z \mid a \leq z < b \}$. In the diagram above, *leaf* = 3 and node $n_{3,2}$ spans the interval [6, 8).

Note that the tree need not be perfectly balanced in the sense that the leaf node level need not be complete if there are insufficient values. Some nodes are simply missing from the right hand side.

Nodes higher in the tree are deemed to span the intervals spanned by their children. Node $n_{l,x}$ where $l \neq leaf$ spans the interval $[v_{l+1,2x}, V_{l+1,2x})$. Thus, we can mark higher nodes with the same value as their left child and the same rule applies to nodes at the same level in the tree.

This data structure allows us to store sets of rectangles easily. We simply have two segment trees, one each for the latitude and longitude information. To build the latitude tree, we create a sorted list of all the unique latitude values at which edges of rectangles occur. These values are then used to mark the

leaf nodes in the tree. The values are then propagated up the tree in the way discussed above. A similar process creates the longitude tree.

Each node can then be marked with a set of rectangles that cover the interval denoted by the node. The intervals should mark nodes as high in the tree as possible. The sets of intervals are shown next to the nodes in the tree above. A given interval can mark several nodes at different levels in the tree, but never two adjacent nodes at the same level.

Extracting the list of rectangles that cover a given point in such a structure is clearly easy and can be accomplished with a simple scan down the tree.

Starting with $l = 0$ and $x = 0$, and searching for the rectangles covering point p we perform the following function recursively:

Compare p with $v_{l,x}$

- If $p < v_{l,x}$ then point p is not covered by any rectangles in the tree and we exit.

- If $v_{l,x} \leq p < v_{l+1,2x+1}$ set $l = l + 1$ and $x = 2x$ and continue.

- If $p \geq v_{l+1,2x+1}$, set $l = l + 1$ and $x = 2x + 1$ and continue.

At each stage we add any rectangles in the current node's list to our result list and then return this as the result when we reach $l = leaf$.

2.3. Keyword Index – Inverted File

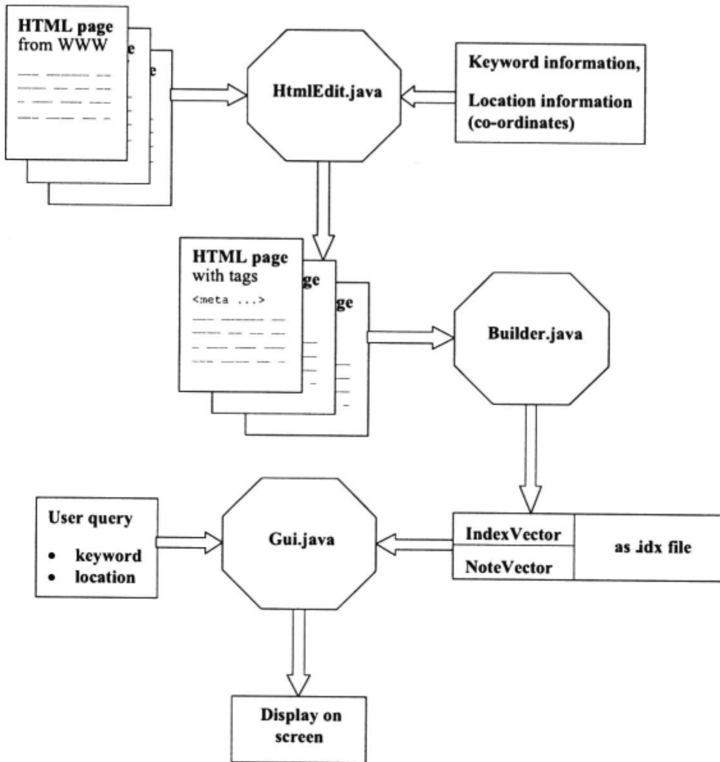
The (keyword, value) pairs stored in the HTML files are used to build an inverted file structure to allow retrieval of the notes based on keyword searches.

An inverted file is a structure that allows the retrieval of documents by reference to the keywords contained in them. This is clearly the inverse of simply retrieving all the keywords in a given document. The design of the file structure and means of reading it is based on the system designed by Mike Burrows for the index in the AltaVista internet search engine [4].

The retrieval of notes from the index is based on a data-driven model. Information from retrieved notes 'flows' from readers through a tree of filters and is made available to the application. These readers and filters could be implemented as separate threads using buffers with synchronized methods to pass values between them, but explicit transfer of the information proved more efficient.

3. IMPLEMENTATION

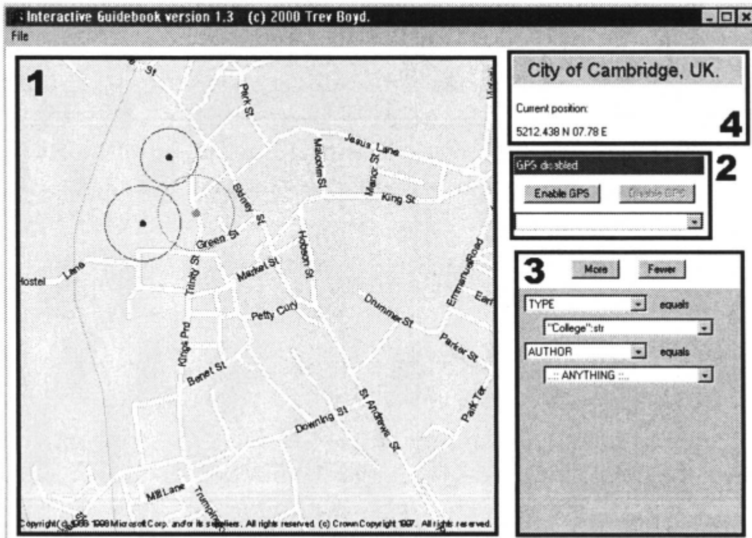
The system is implemented in Java and has the following overall structure:



HtmlEdit is a simple utility that allows the quick and easy addition of meta tags of the correct form to HTML documents. Its GUI contains text fields to allow the entry of the keyword and value fields of the tag and a drop-down menu to allow selection of a type for the value. These are then combined into a suitable `meta` tag in the format described above and this is inserted into the `<head>` section of the HTML.

Builder is responsible for generating the inverted file structure used in the indexing and retrieval of notes by context in the main GUI of the system. The program takes a list of HTML files and directories and scans each file in turn, recursing down through directories, extracting tags. For each file containing at least one such tag, a `Note` object is generated. This contains a list of all the keywords and their values in the HTML. The `Note` object also contains a pointer to the HTML file from which it was generated to allow the file to be retrieved later for display.

Gui provides the map display, GPS controls and the query interface.



The user's current position is displayed on the map (Area 1 in the screen shot above) as a green dot surrounded by a circle. This circle represents the user's area of interest and only notes overlapping this circle will be displayed. As notes are retrieved by the search system, their locations are shown on the map as red dots and circles to show the locations and (spatial) sizes of the notes. The figure below shows the display after a query has returned two notes of interest.

The user's location can be updated as either the result of a mouse click in the map window, or on receipt of an NMEA location message from the GPS unit if serial communication has been enabled using the controls in Area 2.

Queries are constructed in Area 3. Terms are added to the query through two pull-down menus, one for a keyword and the other for its value. The value can be set to 'Anything', which allows the retrieval of all notes with the given keyword specified, independently of its value. The terms are then ANDed together with the position information to retrieve all notes of interest.

Finally, Area 4 gives some general information, including the name of the current guidebook and the user's current position.

A separate window displays the list of notes recovered by the query. Clicking on a note's name opens the relevant document in a Web browser.

3.1. Segment Tree

The system's performance depends on fast spatial indexing using the segment tree. Two trees are used (one each for the latitude and longitude values), so two lists are returned, which must be intersected to find the list of rectangles covered by the given point. This is achieved by sorting the two lists into two arrays, which can then be scanned in parallel.

Rectangles are inserted into the tree using the following algorithm. Suppose x_1 and x_2 are the two boundary values for the rectangle, with $x_1 < x_2$. Nodes are named $n_{l,x}$ as before with the index marking a node being $v_{l,x}$ The rectangles marking a node are represented by $R_{l,x}$. We write $R^*_{l,x}$ to represent the new list created by adding the current rectangle to the rectangle list $R_{l,x}$.

At each node $n_{l,x}$, there are four options to consider.

1 The current node is a leaf node (l = *leaf*).
In this case, we simply mark it with the rectangle ($R_{l,x} = R^*_{l,x}$ and return.

2 Both boundary values are less than the index of the right child ($x_1 < x_2 \leq v_{l+1,2x+1}$).
We only need to mark nodes in the left sub-tree, so we simply proceed down that branch ($l = l + 1; x = 2x$).

3 Both boundary values are greater than the index of the right child ($v_{l+1,2x+1} \geq x_1 > x_2$).
The rectangle only covers intervals in the right sub-tree, so we can proceed down that branch only ($l = l + 1; x = 2x + 1$).

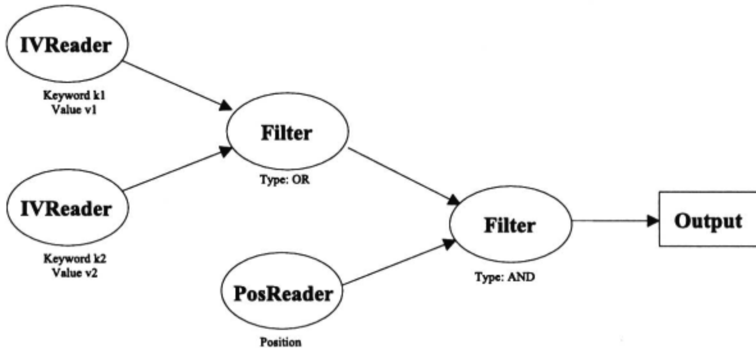
4 The index of the right child lies between the two boundary values ($x_1 < v_{l+1,2x+1} < x_2$).
In this case, the rectangle covers intervals in both sub-trees. We proceed down both branches and then check on our return to see if both children are marked. If so, we unmark both of them and mark the current node instead.

An ordering is imposed on the rectangles to allow the easy intersection of two lists to be calculated. This ordering is the numerical order of the notes from which the rectangles were generated. If two rectangles are from the same note, the co-ordinates of their corners distinguish them. This allows the production of an ordered output compatible with the remainder of the search system.

3.2. Query Processing

All classes in the retrieval subsystem are derived from an abstract class, `streamer`. This provides the basis for a class that will produce a *stream*, or lazily evaluated list, of notes matching given criteria. These criteria might be containing a given (keyword, value) pair, matching a given location or a logical combination of two inputs.

These derived classes can be combined to form tree structures for retrieving notes matching complex search criteria:



Each class has a selection of inputs (either one or two, although more could easily be supported) and a single output, which is then connected to the input of another class. Notes are presented on the inputs and only those matching the criteria set in that object will be passed to the output. Thus, by constructing a tree of such objects, we can build a complex query with many inputs and a single output at the top of the tree.

Three concrete instantiations of streamer are used:

- An `IVReader` yields a sequence of notes containing a given (keyword, value) pair.
- A `PosReader` yields a sequence of notes that include an area of interest intersecting the user's area of interest.
- A `Filter` combines two streams using AND or OR.

Each note carries a unique identifier and data in each stream is sorted by this identifier. `Streamers` are initialised with their output as `Integer.MIN_VALUE`. When a `Streamer` has reached the end of its input, it returns a result of `Integer.MAX_VALUE` to signal the end of the data stream.

If either of the inputs are `Integer.MIN_VALUE` (i.e. this child has not yet been asked for a result), we call `getFile()` on the child to start it searching. Obviously, this may result in the child making calls to its children if it is not a leaf node in the search tree.

Once the inputs are both valid, we compare them and the subsequent behaviour depends on the type of `Filter`.

If it is an AND node:

- If either input is `Integer.MAX_VALUE`, the node has finished and can set its output to `Integer.MAX_VALUE`.
- Otherwise, if the two inputs are equal, one input is copied to the output and both input streams are advanced.

– If they are not equal, the output is not changed and the input with the lowest value is advanced. The lower valued input continues to advance until either the two inputs are equal or one of them finishes.

If the current node is an OR node:

– If both inputs are `Integer.MAX_VALUE`, the node has finished and can set its output to `Integer.MAX_VALUE`.

– Otherwise, if the two inputs are equal, one input is copied to the output and both the input streams are advanced.

– If they are different, the smaller input is copied to the output and its input stream is advanced.

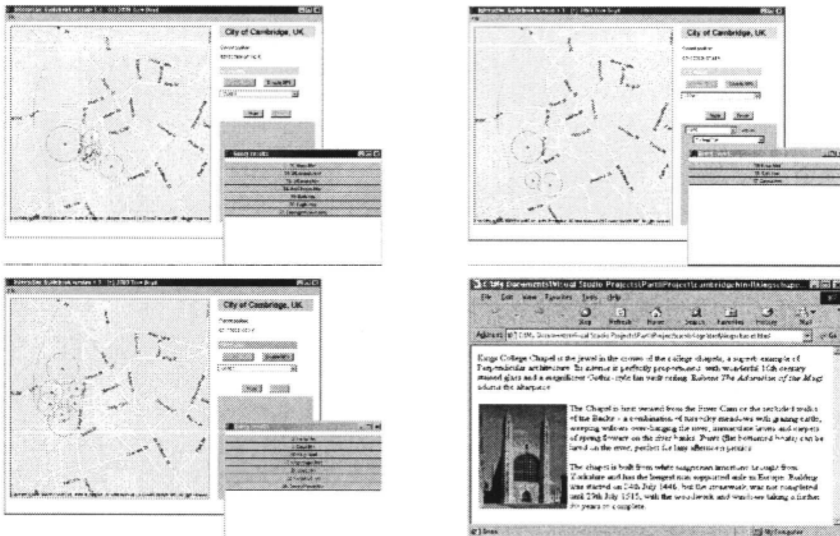
4. RESULTS

This section shows the system in use in its intended environment, i.e. in the hands of a tourist visiting Cambridge and looking for directions to nearby points of interest. Performance figures for the segment tree are also presented.

4.1. Using the Guidebook

The guidebook in use in the example is that compiled during development. It contains notes attached to most of the major landmarks, including colleges, churches and museums.

The following screen shots show the changing display as the user moves, refines the search criteria and finally opens a note:

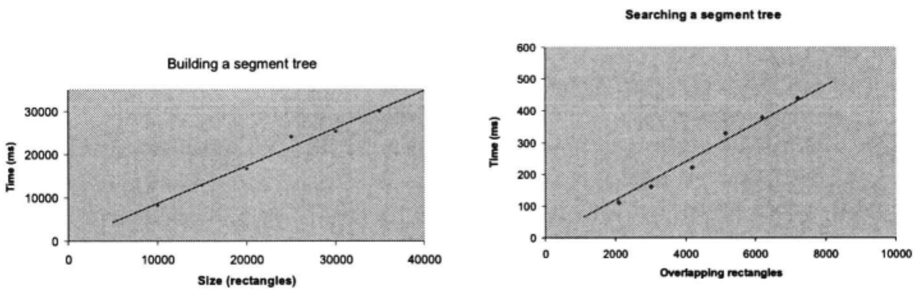


4.2. Segment Tree Performance

Rapid spatial indexing is vital to the system's performance, and the segment tree achieves this. Tests with randomly generated data yield the following results on a modest (133MHz Pentium) lap-top computer.

Two main tests were conducted:

- time taken to construct a segment tree as a function of the number of rectangles,
- time taken to retrieve a list of rectangles covering a fixed point as a function of both segment tree size and the length of the retrieved list.



These test results show a clear linear (order O(n)) relationship between the number of rectangles in a tree and the time taken to build it. The search time also shows a linear relationship to the number of rectangles in the result list. The number of enclosing rectangles is also related by a linear relationship to the tree size. Thus, the search time is also O(n) with tree size. This is extremely good and shows that the system is very efficient in the spatial retrieval of information. We can run many searches a second, even on very large trees, providing us with a low-latency retrieval system.

These tests are using large search trees, several orders of magnitude larger than any expected to be used in the system in practice.

5. CONCLUSIONS

The main aim of this project was to investigate techniques for spatial indexing and the possibility of combining it with keyword indexing of information. The resulting system shows that the combination is not only possible but can be implemented efficiently by a search system based on segment trees and `Streamers`. The two indices complement each other well and combine to provide considerable power and flexibility.

One possible extension of the system is to make use of the position information available from GSM mobile phones, as provided by *Cambridge Positioning*

Systems' Cursor system [6]. This provides an accuracy of around 50 metres from a standard GSM digital phone, which would be much more convenient and cheaper than the use of a dedicated GPS receiver. It has the added advantage over GPS of working indoors.

The recently announced Cambridge Open Mobile System [9] will be used to develop this system in further experiments with information retrieval based on location.

References

[1] ABOWD, G.D. et al. Cyberguide: A Mobile Context-Aware Tour Guide. *Wireless Networks, 3(5), pg 421-433,* October 1997.

[2] BROWN, P.J. (1998). Triggering Information by Context. *Personal Technologies, 2(1),* pg. 1-9, September 1998.

[3] BROWN, P.J. (1996). The Stick-e Document: a framework for creating context-aware applications. *Proceedings of EP '96,* Palo Alto, pg. 259-272.

[4] BURROWS, M. (1999). A Library for Indexing and Querying Text. *Computer Laboratory seminar,* Michaelmas 1999.

[5] DAVIES, N. et al. (1998). Developing a Context Sensitive Tourist Guide. Proceedings of the *First Workshop on Human Computer Interaction for Mobile Devices,* University of Glasgow, pg 64-68, May 1998.

[6] DUFFET-SMITH, P. (1996). High precision CURSOR and digital CURSOR: the real alternatives to GPS. *Proceedings of EURONAV 96 Conference on Vehicle Navigation and Control.*

[7] PASCOE, J. (1997). The Stick-e Note Architecture: Extending the Interface beyond the User. Proceedings of the *1997 International Conference on Intelligent User Interfaces,* pg. 261-264.

[8] SAMET, H. (1990). *The Design and Analysis of Spatial Data Structures.* Addison – Wesley Publishing Company, Inc., USA.

[9] UNIVERSITY OF CAMBRIDGE (2000). *Vodafone and University Announce Experimental Network in Cambridge.* Press release, 11 October 2000. http://www.admin.cam.ac.uk/news/pr/2000101103.html.

[10] VAN LEEUWEN, J. (ed) (1990). *Handbook of Theoretical Computer Science Volume A: Algorithms and Complexity.* Elesevier Science Pulishers B.V., Amsterdam.