PAL - A Cartographic Labelling Library

Olivier Ertz[1], Maxence Laurent[2], Daniel Rappo[1], Abson Sae-Tang[1], Eric Taillard[2]

[1]IICT-SYSIN, University of Applied Sciences, Switzerland
[2]MIS-TIC, University of Applied Sciences, Switzerland
{olivier.ertz, maxence.laurent, daniel.rappo, abson.sae-tang, eric.taillard}@heig-vd.ch

Abstract

PAL is a ready-to-use library for cartographic label placement under free software license. Developed at the University of Applied Sciences of Western Switzerland (HES-SO), it is the result of combining experiences of three teams, in optimization algorithms, in GIS development and in geomatic science and cartography.

Designed for multi-layers and real-time labelling of maps, it provides impressive results, both in terms of execution time and solution quality, using combinational optimization approaches. Options can be set for each layer to customize the labelling process. Options include: priority, in order to decide which of two conflicting labels from different layers to display, the concept of obstacle in order to avoid labels to be displayed above other features, the orientation preference to display labels (free, horizontal, line, centroid,...).

All functionalities are embedded in a C++ library. A Java library is also available within a JNI wrap to have the advantage of PAL in Java applications. This wrap has been implemented with gvSIG through an extension, extJPAL. It demonstrates that it is possible to integrate the library into any desktop or web application, written in Java or C. In the near future, PAL could be a good alternative to be part of what is known as OSGeo Cartographic Library.

1. Introduction

Thanks to standards, geographical information systems are more and more interoperable, helping the user to access easily to all data needed. Quantity of data to visualize in GIS applications is growing, and it is thus more than ever important to have smart tools which can address the problematic of label placement on documents like a map, avoiding overlapping problems and maximizing the number of displayed labels.

PAL is the French acronym for "Automated Placement of Labels". The project aims at providing effective and configurable meta-heuristic algorithms for real time labelling of maps. It is developed at the University of Applied Sciences of Western Switzerland (HES-SO) combining experiences of three teams. MIS-TIC, a team specialized in optimization algorithms for very large problems is in charge of the computational part of the project. IICT-SYSIN, active in Web and GIS development, is focusing on the integration of PAL algorithms into GIS softwares. G2C team, specialized in geomatic science and cartography, provides the knowhow for labelling rules.

2. PAL Features and Functionalities

PAL is designed for multi-layers labelling. It handles layers of points, lines or polygons. Each layer has properties that influence the labelling process: (I) a scale range for which the layer will be labelled, (II) a priority, in order to decide which of two conflicting labels from different layers to display, (III) a concept of obstacle in order to avoid labels to be displayed above other features, (IV) an activity status (is the layer currently displayed ?), (V) a “toLabel” status (“is the layer to be labelled ?”), (VI) the orientation preference to display labels (free, horizontal, line, centroid, ...). All functionalities are embedded into a C++ library. A Java library is also available within a JNI wrap.

PAL has only to know the size of each features' label. Two kinds of units are handled for label size: pixel (label has the same size on screen whatever the scale is) or map unit (the label size varies while zooming in or out). For PAL, labels are abstract bounding box, so everything which has to be placed can be handled.

2.1 Labelling example with PAL

Figures 1 and 2 below clearly illustrate the common situation of cartographers wanting to display a maximum of labels on a map so to identify objects at a glance. These figures are representations of a small part of Blo03 data (see computational section below) with polygons representing buildings and parcels, and lines representing fresh and used water conducts (scale : 1/1000).

Figure 2. Blo02-3 Polygons

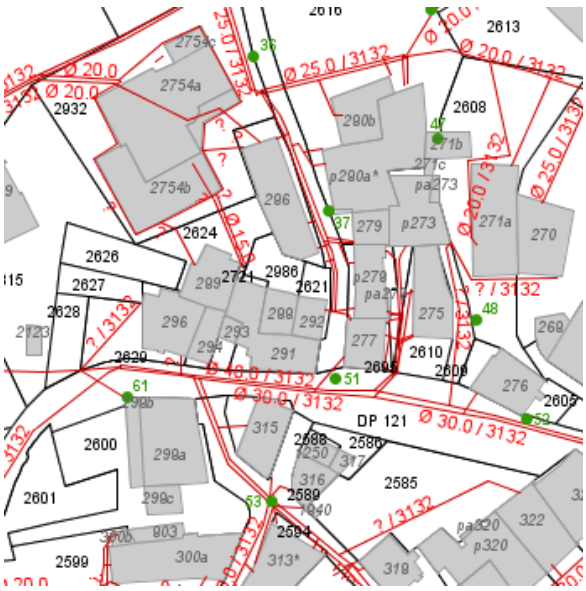

Figure 1. Blo03-4 Polygons, lines, points

3. How Does PAL Work ?

3.1 Algorithms

The labelling process has two phases: problem generation and optimization. Problem generation answers the question : Where are the best candidate positions to put a label for a given geometry (either a point or a line or a polygon feature) ? This phase returns a set of candidate label positions for each object and a conflict graph. The nodes of this graph are the label candidate and an edge links two candidates if they cannot be displayed simultaneously.

The optimization phase is to choose which of the candidate label should actually be displayed on the map. The choice of retaining a candidate takes into account position quality, layer priority and conflict graph. Thus, the optimization phase can be done independently of feature type.

3.2 Problem Generation Phase

The problem generation phase is composed of 2 main steps: Candidate generation and candidate filtering. Candidate generation takes into account the list of objects to label (practically stored into layers), a list of obstacles, a scale and a map extent. Three conditions must be satisfied for an object to be part of the problem. (I) its layer is active (meaning that it has to be displayed), (II) the scale is in the layer's scale range, (III) the object is completely or partially on the map extent.

For each objects to label, a large number of candidate label positions are first generated and a cost is associated to each candidate. The cost of a candidate depends on its cartographic preference and on the fact that it covers an obstacle or not. Then, candidate filtering selects only few of the candidates with smallest costs. The number of candidates retained for each object depends on its geometry.

Candidates Generation

Point feature : exactly *p* candidates are generated, where *p* is a parameter. The best candidate is located upper-right of the point and has a cost of 0.0001. Others candidates are uniformly arranged around the point, the worst is located bottom-left and its cost is 0.0021. The distance *distlabel* between candidates and the point can be specified. It is generally equal to the symbol radius.

Line feature : two different ways are provided to arrange labels. The first, *P_LINE*, puts candidates above the line, the second, *P_LINE_AROUND*, puts labels on both sides of the line. With the latter, the distance *distlabel* between candidates and the line can be defined. The number of candidates depends on the line's length. If the line is shorter than the label, only one candidate (or two with *P_LINE_AROUND*) centred on the line is generated. Otherwise, candidates are regularly spread along the line. The number of candidates generated depends on the line length.

The orientation of a candidate is parallel to a straight segment linking two points of the line, with the length of the line portion connecting these point being equal to the length of the label. If the ratio between the label length and the distance from one point to the other is higher than 0.98, then the candidate cost is 0.0001. Otherwise, the candidate cost is set to $10^{-2} \cdot (1 - \textit{label length / line length})$.

Polygon feature : four ways are provided to arrange candidates: *P_POINT* uses polygon's centroid as a point (see the point feature section); *P_LINE* uses polygon's perimeter as a line (see the line feature section); *P_FREE* arranges candidates at best inside the polygon, rotations are allowed; *P_HORIZ* is the same as *P_FREE*, but force all candidates to be horizontal.

First of all, it is checked whether the polygon is convex[1] or not. A non-convex polygon is split into several convex polygons. The candidate positions for a convex polygon are defined as follows. First, a rectangle of minimal area embedding the polygon is determined. Then, this rectangle is filled with candidates. In *P_FREE* mode, candidates are parallel to rectangle borders, in *P_HORIZ* mode, candidates are horizontal. The cost of a candidate is inversely proportional to its distance to

1 . Actually, a polygon is considered as convex if the area of its convex hull is less than twice the label area.

the polygon's border or hole, or to the distance to an obstacle.

Candidates Filtering

Candidates generation generally produces too much candidate labels. So, only candidates having potential to lead to good solution are preserved. A first filter stage discourages the use of candidates locating over obstacles. A second filter stage selects for each object the *p* candidates having the lowest cost. A third filter stage, in the spirit of rule *L*1 (Wagner et al, 2001), removes all candidates of a given object that are worse than a candidate of this object having no overlap with candidates of any other objects. This filter stage is recursively applied until no improvements can be obtained.

3.3 Optimization Phase

The problem generation phase provides a list of *n* objects to label and, for each object, a list of candidate label positions. Each candidate has a geographical cost (stored in the vector *costs*), and a list of other conflicting candidates. Two conflicting candidates cannot be displayed at the same time on the map. Each object $^{i}, i \square \{0 \ldots n\}$ has a special cost *inactiveCosts*[*i*] used when the object is not labelled; this cost depends on layer's priority and is between 1 and 10. A solution to the problem is a list of conflict free labels to display. A solution can be represented by a vector *sol*, of size *n*, with he *i*th component indicating which candidate is displayed for object *i*. Value –1 indicates that the object is not labelled. The labelling problem can be stated as finding a solution *sol* without overlaps minimizing the expression given by Formula 1.

$$\sum_{i=0}^{n} \begin{cases} costs[sol[i]] & \text{if} \quad sol[i] \geq 0 \\ inactiveCosts[i] & \text{else} \end{cases} \qquad [1]$$

An initial solution is first built along the lines of the first step of FALP algorithm (Yamamoto et al, 2005): (I) Put every candidates into a set *s*, (II) select the candidate *c* from *s* which has the smallest number of overlaps and put it into the solution, (III) remove from *s* all candidates which overlap with *c*, as well as other candidates of *c*'s object and update number of overlaps for remaining candidates in *s*. (IV) Go back to (II) until *s* is empty.

Solution Improvement Techniques

Three techniques *chain*, *pop_chain* and *pop_tabu_chain* are available in the software for improving the initial solution. Moreover, *pop_tabu* (Alvim et al, 2008), one of the best technique available today for point feature label placement is also considered in the numerical experiments that follow. This technique is based on the generic POPMUSIC optimization frame (Taillard et al, 2001) and works on a slightly different version of the problem where all objects must be labelled, but with a cost (comprised between 1 and 10 depending on layer's priority) for each label overlap. It embeds a tabu search method (Glover et al, 1997) that works with a very simple neighbourhood consisting of iteratively changing the label candidate retained for one object at a time.

Chained neighbourhood (chain)

The chained neighbourhood works as follows. An object is selected and its label is modified in the current solution. If the object was not labelled, then the best candidate for this object is selected (even if it creates one or more overlaps); if the object was already labelled, then the best among all

other candidates for this object is selected. At this point, several possibilities may occur:

1) The solution obtained is better than the current solution, then it becomes the current solution and the process is repeated from there

2) The solution obtained has exactly one overlap (that involves the label just modified and the label of another object). In this case, the label of the other object is modified (if possible) and the process recursively continues from there

3) The solution obtained has more than one overlap or exactly one overlap but there is no other possibility to label the other object. In this case, the overlapping labels are not displayed any more (thus leading to a solution with inactive costs) and the chain of modifications stops.

The chain modifications is also stopped if the number of modifications is higher than a given limit or if the label to modify is those of an object already modified in the chain. Once the chain is stopped, the best solution visited along the chain becomes the current solution and the process is repeated while the solution is improved.

POPMUSIC with chain (pop_chain)

The basic idea of POPMUSIC is to locally optimize sub-parts of a solution, once a valid solution to the problem is available. When these local optimizations are made with the chain method presented above, the technique is called *pop_chain*. Here, a chain is initiated by modifying the label of one object after the other. If a chain contains a valid and improving solution, the last is retained and the process is iterated. The process is stopped when all objects have initiated a rejected chain.

POPMUSIC with tabu search and chained neighbourhood (pop_tabu_chain)

The idea behind tabu search frame is to iteratively perform local modifications to the solution, even if the solution so obtained is worse than the starting solution. In order to avoid to visit cyclically the same subset of solutions, the reverse of a modification is forbidden (made tabu) for few iterations. At each tabu search iteration, several modifications are evaluated and the best non-tabu is selected and performed. The new technique *pop_tabu_chain* is a POPMUSIC that embeds a tabu search with a neighbourhood based on chained modifications as optimization process.

4. Computational Experiments

The techniques presented above are programmed in C++ and run on Gentoo GNU/Linux 2.6.23 with 2.2 Ghz Intel Core2 Duo processor (only one core used at a time by the algorithm). They have been tested on academic problem instances (Wagner et al, 2001) (HardGrid and RandomRect: objects are points and have exactly four candidates per point) and real data containing two layers of polygons, one layer of lines and one layer of points (Blo01, Blo02, Blo03). Table 1 provides the geographical characteristics of the real instances while Table 2 provides the characteristics of the instances obtained by the problem generation phase, depending on the obstacle layers chosen by the user. These characteristics are the number of label candidates, the number of conflicts between candidates and the computational time required by this phase.

Figure 3 compares the performances (computational time and percentage of objects labelled) of the four techniques on HardGrid and RandomRect instances. It can be seen that *chain* provides moderately good solutions very rapidly. The solutions produced by this technique are sufficiently

good and look pretty for interactive mappings. The new methods *pop_chain* and *pop_tabu_chain* are able to find better or significantly better solutions than *pop_tabu*, at the expense of a higher computational effort. Figure 3 shows that computational effort grows quasi linearly with the number of objects to label. The last columns of Table 2 provides the computational times and percentage of objects labelled for the new optimization techniques presented in this article.

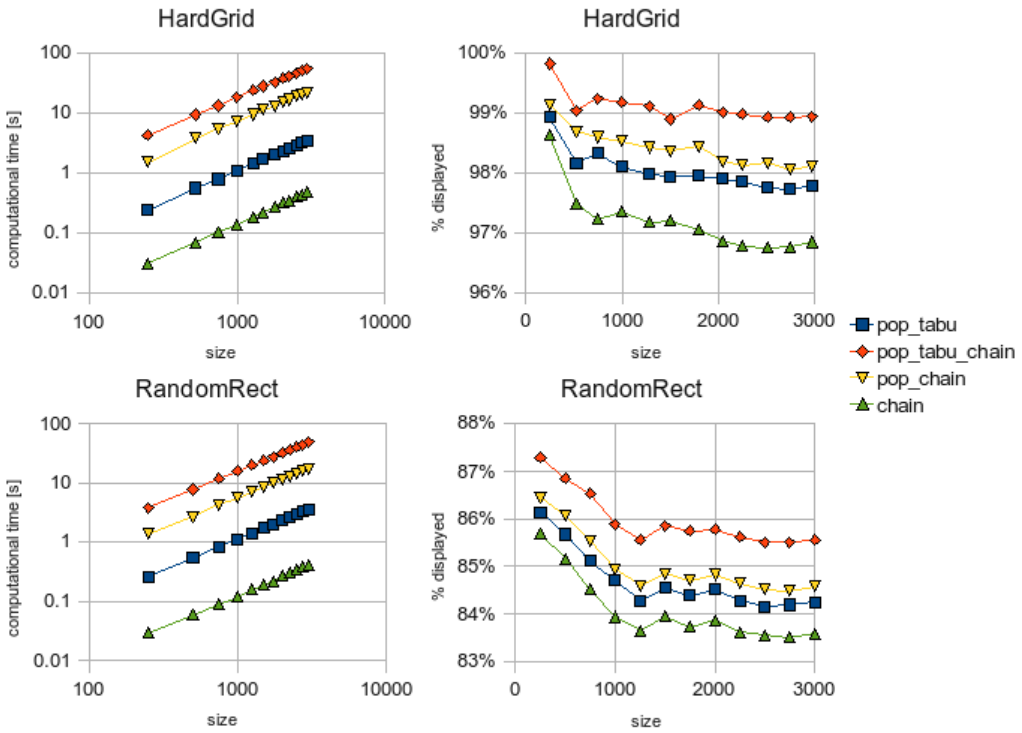

Figure 3. HardGrid and RandomRect problems: time and percent displayed

Name	*Scale*	*Area [km²]*	*Map size [cm]*	*# points*	*# lines*	*# polygons*	*# objects*
Blo01	1:6000	0.478380	15.7x8.5	149	398	513	1060
Blo02	1:1000	0.055952	26.9x20.8	90	264	276	630
Blo03	1:5000	45.813726	171.8x106.7	1,182	2901	5,171	9254

Table 1. Main characteristics of real problems

Problem	*# obstacle layers*	*# candidates*	*# conflicts*	*Time gen. [s]*	*Time chain*	*% chain*	*Time pop_chain*	*% pop_chain*	*Time pop_tabu_chain*	*% pop_tabu_chain*
Blo01	0	5,490	236,387	0.31	0.23	30.28	10.42	30.85	17.64	31.04
Blo01	1	3,275	76,189	0.25	0.08	28.21	5.07	28.77	10.01	28.68
Blo01	3	2,611	49,301	0.23	0.06	25.75	3.20	26.32	7.11	26.04
Blo01	4	2,529	47,544	0.23	0.05	26.04	3.04	26.04	7.19	26.42
Blo02	0	5,028	48,458	0.39	0.18	67.62	5.64	67.62	6.77	68.41
Blo02	1	2,289	10,773	0.62	0.03	63.81	1.44	63.65	2.61	64.13
Blo02	3	2,129	8,317	0.60	0.03	62.22	1.23	62.70	2.15	63.65
Blo02	4	2,007	7,809	0.60	0.02	62.86	1.11	63.17	1.82	63.65
Blo03	0	54,558	1,466,745	12.43	2.77	46.71	79.13	47.62	110.19	47.24
Blo03	1	37,562	576,017	13.83	1.4	45.70	51.23	46.57	74.91	46.27
Blo03	3	30,666	359,779	13.83	0.92	43.32	34.15	44.28	58.07	44.16
Blo03	4	29,945	345,764	13.80	0.92	43.18	32.73	44.05	54.86	43.79

Table 2. Instance characteristics and efficiency of techniques proposed in this article

5. Integration of PAL Into GIS Desktop

Currently, default labelling functionalities provided by GIS softwares, FOSS and non-FOSS, are generally basic without a candidates generation. So, labels can only be placed on polygon centroid (sometimes label is even displayed outside the polygon to label), on line end or middle of line.

Thus, labelling possibilities are significantly restricted and so is the map legibility. A main drawback is that labels are greedily placed. They are just displayed one after the other if there is no overlapping, so that the first one has more chance to appear than the last one. To perform advanced labelling, one has often to acquire a specific extension.

Even if labelling is less a GIS functionality than a cartographic tool, there are two important dimensions that any labelling module has to take into account : the algorithmic aspects and the GUI (Graphical User Interface). The first one has to conciliate efficiency and solution quality to place intelligently in near real-time a maximum of labels, so as to produce a legible map. The second one must provide good interactivity for the user to complete his work at best. Providing a saving time rich user interface to customize label placement is an important aspect, but the PAL project does not focus on this second dimension.

For 40 years, many research projects have been lead on automated labels placement and describe algorithms providing better results than the greedy approaches. Best of them provide impressive results, both in terms of execution time and solution quality, using combinational optimization approaches (Edmonson et al, 1997), as PAL does.

Translating the power of these algorithms can be discouraging because of their complexity. Therefore, by combining three specialized teams, this project was able to provide a ready-to-use C++ library for automated placement of labels.

5.1 Towards PAL As A Labelling Library for FOSS4G Applications

To prove usability of the PAL library, it is necessary to build a graphical user interface on this "brick of intelligence" to drive it. It appeared clear that an integration into a well-known GIS application would be better for the long term life of the project. As FOSS has priority, among many existing applications (OpenJUMP, gvSIG, uDig, ...), it was decided to use gvSIG (Generalitat Valenciana 2008) to create an extension for label placement based on PAL library due to experience acquired through other previous projects. Notice that gvSIG is written in Java, so it was first necessary to create a C-Java bridge to be able to use PAL from Java source code.

5.2 JPAL, A Java Bridge for PAL

With JNI (Java Native Interface) framework, it is possible to call from a Java program some non-Java native source code and reversely. For PAL, the native source code is written in C/C++. Therefore, PAL library (pal.dll, pal.so) has been wrapped with the addition of some classes that assume the gateway when calling a « cross-language » method. Following this guideline, JPAL has been created. It is a Java library encapsulating the PAL library and these JNI specific classes, so that it is quite easy for a Java developer to use PAL functionnalities.

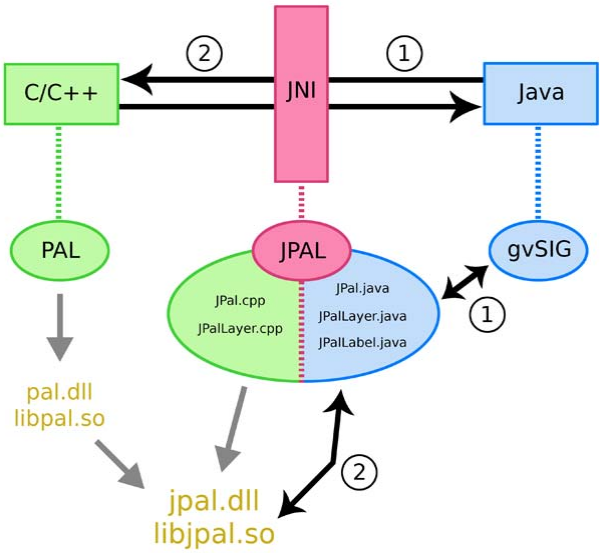

Figure 4. JPAL=JNI+PAL

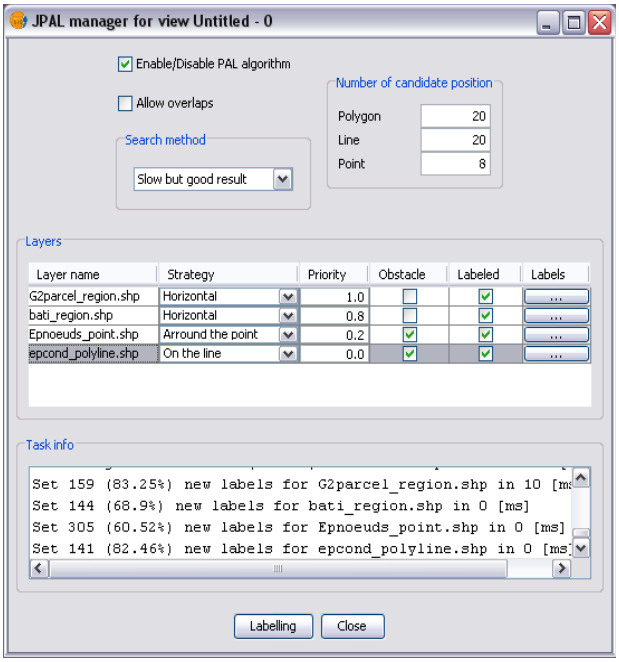

Figure 5. extJPAL control panel

5.3 extJPAL: A PAL Extension for gvSIG

gvSIG is built on a framework providing a plugin system so that it is possible to add new functionalities without modifying core source code. Following this architecture, extJPAL is an extension for gvSIG 1.1.x enabling some new functionalities of labelling based on PAL. Yet, the extension has reached the beta testing phase. It exposes almost all functionalities from PAL library and fill the initial purpose to demonstrate its use in real situations of use.

Finally, in term of execution time, it is important to take into account time due to the JNI wrapping, but also the efficiency of the GIS software itself, for rendering and event management. Moreover, to connect gvSIG with JPAL, the system must also calculate the bounding box of each

label by considering the chosen font name and size. The extension receives back placement results to create and render the text layer. This chain of actions takes only few seconds to label a map with four important layers at the same time. It demonstrates that the extension fits for use during navigation (zoom and pan).

6. Conclusion

More than just a description of new algorithms, PAL is a ready-to-use library for cartographic label placement under free software license. Source code is available through a SVN repository. The same goes for the extJPAL extension available through the gvSIG extension community repository (gvSIG team 2008). One can get there installation instructions, project status and links to source code and binaries. Moreover, the project provides a bug reporting tool.

Many improvements could still be lead : considering line width in obstacle detection, allowing on-the-fly projection changes, directly accessing and releasing spatial entities (with callbacks registering), forcing label positions, improving candidate generation for polygon, etc.

Implementing PAL into a gvSIG extension demonstrates that it is possible to integrate it into any other desktop or web application. Also, in the near future, one can imagine that PAL could be a good alternative to be part of what is known as OSGeo Cartographic Library (Neteler 2008).

For more informations, go to http://geosysin.iict.ch (IICT-SYSIN 2008).

7. Acknowledgements

The authors wish to thank Adriana C. F. Alvim who provided code for the point feature label placement, Patrick Bailly who has developed functions for measuring the quality of candidate label positions, Francis Grin and Sébastien Roh who provided real datasets for computational tests and shared their experience and comments in map production.

8. References

Alvim, A.C.F., Taillard, E., (2008), *POPMUSIC for the point feature label placement*, European J. Oper. Res., To appear.

Edmondson, S., Christensen, J., Marks, J., & Shieber, S, 1997, 'A General Cartographic Labeling Algorithm', *Cartographica*, vol. 33, no. 4, pp. 13-23.

Generalitat Valenciana 2008, gvSIG project, Valencia, Spain, viewed 15 August 2008, <http://www.gvsig.gva.es>.

Glover, F., Laguna, M. (1997), *Tabu Search, Kluwer Academic Publishers, Dordrecht, The* Netherlands

gvSIG team 2008, gvSIG extensions repository, Valencia, Spain, viewed 15 August 2008, <https://gvsig.org/plugins/downloads>.

IICT-SYSIN 2008, PAL Project, Yverdon-les-Bains, Switzerland, viewed 15 August 2008, <http://geosysin.iict.ch/PAL>.

Neteler, M., *OSGeo Cartographic Library*, Open Source Geospatial Foundation Wiki, viewed 15 August 2008, <http://wiki.osgeo.org/wiki/OSGeo_Cartographic_Library>.

Taillard, E., Voss, S. (2001), 'POPMUSIC: Partial Optimization Metaheuristic Under Special Intensification Conditions', in Ribeiro, C. and Hansen, P. (eds.), *Essays and surveys in metaheuristics*,

Kluwer Academic Publishers, Boston, USA, pp. 613–629.

Wagner, F., Wolff, A., Kapoor, V., Strijk, T. (2001), 'Three rules suffice for good label placement', *Algorithmica*, vol. 30, pp. 334–349.

Yamamoto, M., Camara, G. and Lorena, L.A.N. (2005), *Fast Point-Feature Label Placement Algorithm for Real Time Screen Maps*, VII Brazilian Symposium on GeoInformatics (GeoInfo 2005).