Partitioning a Call Graph

Rob H. Bisseling[*] *Jarosław Byrka*[†] *Selin Cerav-Erbas*[‡]
Nebojša Gvozdenović[†] *Mathias Lorenz*[‡] *Rudi Pendavingh*[§]
Colin Reeves[¶] *Matthias Röger*[§] *Arie Verhoeven*[§]

Abstract

Splitting a large software system into smaller and more manageable units has become an important problem for many organizations. The basic structure of a software system is given by a directed graph with vertices representing the programs of the system and arcs representing calls from one program to another. Generating a good partitioning into smaller modules becomes a minimization problem for the number of programs being called by external programs. First, we formulate an equivalent integer linear programming problem with 0–1 variables. Theoretically, with this approach the problem can be solved to optimality, but this becomes very costly with increasing size of the software system. Second, we formulate the problem as a hypergraph partitioning problem. This is a heuristic method using a multilevel strategy, but it turns out to be very fast and to deliver solutions that are close to optimal.

1 Introduction

In recent years, the capabilities of information technology have increased tremendously. At the same time, large software systems in today's organizations such as banks, health care providers, or government agencies, have become costly to maintain. To reduce the maintenance costs, the systems need to be split into smaller, more manageable modules (typically 5–10 modules). Each module can then be assigned to a separate team of maintainers. A partitioned system needs interfaces for the communication between modules; the number of interfaces is the main cost factor. In a good partitioning, the size of each module is restricted and the total size of the interface is minimized. To find such a partitioning is a job for a trained expert, but when the system is large an automated suggestion for a partitioning into modules becomes useful.

A software system can be described by a *call graph*. A call graph is a directed graph, where vertices represent programs, classes, or similar program units, and where an arc (v, w), i.e. $v \to w$, means that program v calls program w. Figure 1 shows a complete call graph of a Java software system named `Java1` and Figure 2 shows part of this graph in detail. Each vertex may have a weight, such as the number of lines

[*]Universiteit Utrecht
[†]Centrum voor Wiskunde en Informatica
[‡]Université Catholique de Louvain
[§]Technische Universiteit Eindhoven
[¶]Coventry University

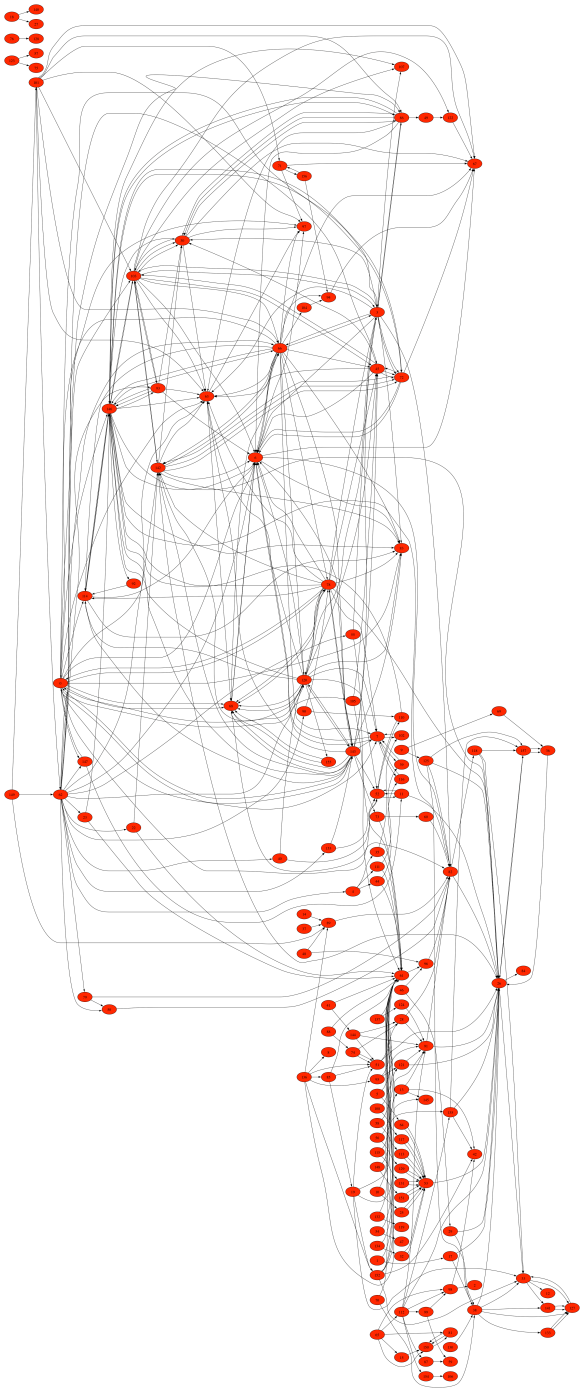

Figure 1: Call graph of the software system `Java1`, which was provided by SIG. The number of vertices (programs) is $N = 158$.

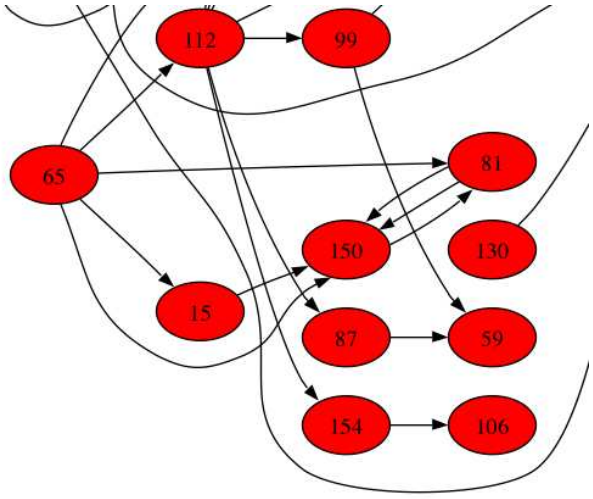

Figure 2: Detailed view of the bottom part of the call graph from Figure 1 showing the arcs between the programs. Note for instance that program 65 calls program 81. Program 81 calls 150 and vice versa. Duplicate calls may occur; these can be removed without affecting the problem.

of code of the corresponding program. Here, we assume that less detail is required so that all programs are equally costly to maintain, and hence all vertices have weight 1. A module contains a subset of the vertices, representing a subset of the programs. The size of a module equals the sum of the vertex weights in the corresponding subset; the size of its interface equals the number of vertices which have an incoming arc from a different module. Thus, the software splitting problem can be formulated as a partitioning problem of a call graph.

The Software Improvement Group (SIG, `http://www.sig.nl`), located in Diemen (the Netherlands), provides tools which help organizations to understand their software better. They are interested in partitioning algorithms which solely exploit the structure of the call graph to split the software and which are useful for various graph sizes, from hundreds of vertices to over a million.

The call-graph partitioning problem was posed by SIG at the opening day of the Study Group Mathematics with Industry 2005 in Amsterdam. We have studied the problem and in this report we propose our solutions. The following sections present different formulations of the same problem and different solution methods. In Section 2, the problem is mathematically formulated as a graph partitioning problem, and then translated into an integer linear programming (ILP) problem with variables taking values only in $\{0,1\}$, which can be solved by standard commercial software such as CPLEX [4]. Because the ILP problem is NP-hard, it cannot be solved to optimality for very large call graphs (more than 1500 vertices). Therefore, Section 3 describes a very fast heuristic method which is based on a multilevel approach to hypergraph partitioning. Section 4 compares the different methods for some real call graphs of software systems written in Java and COBOL, which were provided by SIG. Finally, Section 5 presents our conclusions and recommendations.

2 Solution by integer linear programming

We will first formulate the problem as a graph partitioning problem and then translate it into an ILP problem with 0–1 variables.

2.1 Graph partitioning problem

A call graph is a directed graph $D = (V, A)$, where the vertex set V is the set of programs of the software system and the arc set $A := \{(u, v) \in V \times V \mid u \text{ calls } v\}$. Given a subset of programs $U \subseteq V$, the *interface* of U in D is the set of all programs in U called by programs not in U:

$$I_D(U) := \{u \in U \mid (v, u) \in A \text{ for some } v \in V \setminus U\}. \tag{1}$$

We call $u \in I_D(U)$ an *interface vertex* of U. A *partition* of a set V is a collection of nonempty, pairwise disjoint subsets of V, such that the union of these subsets is V.

With these definitions, a mathematical formulation of the graph partitioning problem is:

Given: A directed graph $D = (V, A)$, $K \in \mathbb{N}$, $L \in \mathbb{N}$
Find: A partition $V_1, \ldots, V_L$ of V such that $|V_l| \leq K$ for each l, and such that $\sum_{l=1}^{L} |I_D(V_l)|$ is as small as possible.

2.2 Integer linear programming problem

Consider the following ILP problem:

$$\begin{array}{lll}
\text{minimize} & \sum_{l=1}^{L} \sum_{v \in V} x_{vl} & \\
\text{subject to} & \sum_{l=1}^{L} y_{vl} = 1 & \text{for all } v \in V \\
& \sum_{v \in V} y_{vl} \leq K & \text{for } l = 1, ..., L \\
& x_{vl} \leq y_{vl} & \text{for } l = 1, ..., L \text{ and for all } v \in V \\
& y_{vl} \leq y_{ul} + x_{vl} & \text{for } l = 1, ..., L \text{ and for all } (u, v) \in A \\
& x_{vl}, y_{vl} \in \{0, 1\} & \text{for } l = 1, ..., L \text{ and for all } v \in V
\end{array}$$

It is not difficult to see that if $V_1, \ldots, V_L$ is a proper solution to the graph partitioning problem, then by setting

$$y_{vl} = 1 \text{ if } v \in V_l, \text{ and } 0 \text{ otherwise,} \tag{2}$$

$$x_{vl} = 1 \text{ if } v \text{ is an interface vertex of } V_l, \text{ and } 0 \text{ otherwise,} \tag{3}$$

we obtain a feasible solution of the above ILP problem. Conversely, given an optimal solution to the ILP problem, taking

$$V_l := \{v \in V \mid y_{vl} = 1\}, \tag{4}$$

$$I := \{v \mid x_{vl} = 1 \text{ for some } l\}, \tag{5}$$

will yield an optimal solution $V_1, \ldots, V_L$ to the call-graph partitioning problem with a set of interface vertices I. (It is straightforward to adapt this formulation to the variant

of the call-graph partitioning problem where each program has a certain weight and the total weight of each module is bounded.)

The general ILP problem, which is to solve

$$\min\{c^{\mathrm{T}}x \mid Ax \leq b, x \in \mathbb{Z}^n\} \tag{6}$$

for a given matrix A and vectors b, c, is NP-hard, see [6]. The standard solution methods are efficient in practice when the polyhedron $P := \{x \in \mathbb{R}^n \mid Ax \leq b\}$ is close to the convex hull of $P \cap \mathbb{Z}^n$. We have attempted to create a formulation of our problem with this property — which is why we chose this formulation over others with less variables/constraints. For detailed information on integer programming theory and methods, see [7]. A textbook is [9].

In our formulation of the call-graph problem, each feasible partition $V_1, \ldots, V_L$ is represented at least $L!$ times, since each permutation of the subsets of the partition yields a different binary vector y. This decreases the efficiency of the standard solution methods, so some form of symmetry breaking is desired. To eliminate the abundance of representations of essentially the same feasible solution, we added, for certain vertices $s_1, \ldots, s_{L-1}$, the set of constraints

$$\sum_{i=1}^{l} y_{s_l i} = 1 \text{ for } l = 1, ..., L-1 \tag{7}$$

to our model. Thus, the first vertex is fixed in V_1, the second in $V_1 \cup V_2$, and so on. These constraints will allow at least one representation y of each feasible partition $V_1, \ldots, V_L$ in the feasible set (and exactly one for feasible partitions with all fixed vertices in different subsets). We chose $s_1, \ldots, s_{L-1}$ to be the set of $L-1$ vertices of largest *outdegree* in D (i.e., with the largest number of outgoing arcs). The choice of $s_1, ..., s_{L-1}$ and our method of symmetry breaking is still not optimal. Solving the symmetry problem properly, however, seems the key to solving the call-graph problem through an integer programming formulation. Further improvement of our method is up to future research.

3 Solution by multilevel hypergraph partitioning

We will reformulate the graph partitioning problem as a hypergraph partitioning problem and then present a heuristic solution method based on a multilevel approach.

3.1 Hypergraph partitioning problem

A hypergraph $H = (V, \mathcal{N})$ consists of a set of vertices $V = \{v_1, \ldots, v_N\}$ and a set $\mathcal{N}$ of *hyperedges*, or *nets*, which are subsets of V. A hypergraph is a generalization of an undirected graph: a hyperedge connects an arbitrary number of vertices, whereas an edge in a graph connects two vertices; an edge can be viewed as a subset $\{v_i, v_j\}$ of size two.

As before, let the structure of a software system be given by a directed graph $D = (V, A)$, where $V = \{v_1, ..., v_N\}$ represents the set of programs and $(v_i, v_j) \in A$

indicates that program i calls program j. We consider the hypergraph $H = (V, \mathcal{N})$ and choose the set of nets as

$$\mathcal{N} = \bigcup_{j=1}^{N} n_j, \tag{8}$$

where net n_j consists of program j and all programs that call j,

$$n_j \quad := \quad \{v_j\} \cup \{v_i \mid 1 \le i \le N \text{ and } (v_i, v_j) \in A\}. \tag{9}$$

A net is *broken* by a given partition $V_1, \ldots, V_L$ if its vertices are in different subsets of the partition, i.e., in different modules. A net n_j is broken if and only if program j is an interface program. This is because for a broken net n_j, at least one calling program i must be in a module different than the module of j. The software splitting problem has become a hypergraph partitioning problem where we are looking for a partition that minimizes the total interface size,

$$|I| \quad := \quad |\{j \mid 1 \le j \le N \text{ and } n_j \text{ is broken}\}|.$$

A convenient way of looking at a graph is to consider its *adjacency matrix*. We describe the calls between the N programs by the $N \times N$ adjacency matrix $A = (a_{ij})_{i,j=1,\ldots,N}$, with $a_{ij} \in \{0, 1\}$, defined by

$$a_{ij} \quad = \quad \begin{cases} 1 & \text{if program } i \text{ calls program } j, \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

Thus, $a_{ij} = 1$ if and only if $(v_i, v_j) \in A$. Note that we identify the matrix A with the arc set A. We add a unit diagonal to the adjacency matrix. This way, the vertices of net n_j correspond exactly to the positions of nonzeros in column j of the matrix. Figure 3 shows an extended adjacency matrix.

A traditional application area of hypergraph partitioning is the design of electronic circuits. MLpart [2] is a hypergraph partitioner specifically developed for this purpose. Çatalyürek and Aykanat [3] introduced hypergraph partitioning for the purpose of distributing the work in multiplying a sparse matrix and a vector on a parallel computer, which is the core computation of iterative linear system solvers. They implemented the partitioning in software called PaToH. The package Mondriaan, recently developed at Utrecht University [8], is a two-dimensional sparse matrix partitioner which cuts the matrix recursively into smaller rectangular shapes, similar to the paintings of the Dutch painter Piet Mondriaan (1872–1944). Each cut is based on hypergraph *bi*partitioning, which is explained in the following.

Multilevel methods for graph or hypergraph partitioning reduce the size of the problem repeatedly by merging vertices with similar connectivity until the remaining problem is sufficiently small (a few hundred vertices), then solve the smaller problem for instance by a local heuristic such as the Kernighan–Lin [5] algorithm, and finally unmerge the merged vertices at the different levels, each time refining the solution by a simpler method such as trying to move interface vertices to the other subset of the partition. Typically, each level of merging halves the problem size. A good similarity criterion for merging, used in both PaToH and Mondriaan, is the inner product of the corresponding rows in the adjacency matrix. A large inner product for rows i and i'

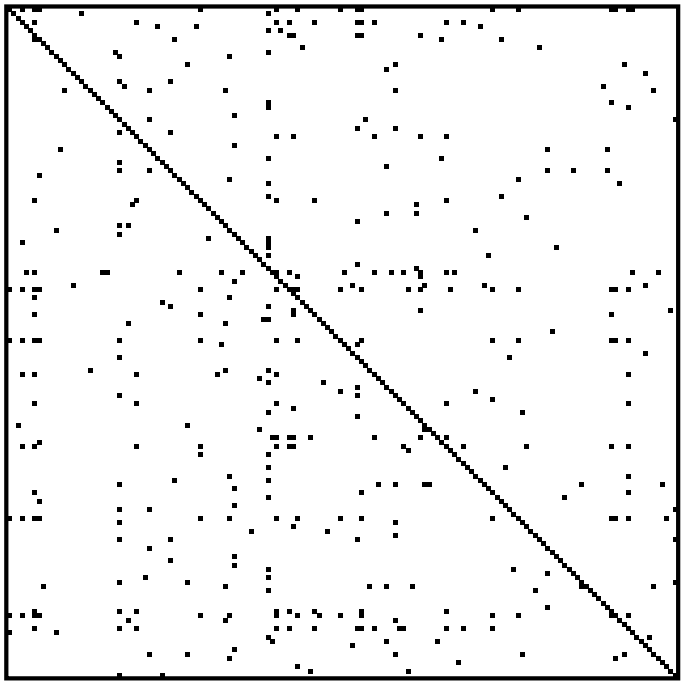

Figure 3: 158×158 adjacency matrix of the problem `Java1`, provided by SIG, which has 158 programs and 422 calls from programs to other programs. The matrix, extended by a unit diagonal, has 580 nonzero elements. It is *sparse*, since the vast majority of its elements is zero.

means that many nonzeros from those rows occur in the same positions, and hence programs i and i' often call the same program j. Multilevel methods, first proposed by Bui and Jones [1], have been very successful in graph and hypergraph partitioning. The Kernighan–Lin algorithm works by trying moves of a vertex to the other subset of the partition, each time accepting the move with the largest gain, i.e., reduction in number of broken nets. A temporary increase is also accepted if this leads to a reduction later on. Several passes are made through the whole set of vertices. This algorithm is local in nature and hence only works for a limited number of vertices; for larger problem sizes, the algorithm easily gets stuck in a local minimum.

To apply Mondriaan (version 1.01) to our problem, we had to make the following adjustments. First, we want to use the matrix partitioner Mondriaan only in one-dimensional (1D) mode, because the partitioning of the vertices we are looking for corresponds to a partitioning of the rows of the adjacency matrix. Our aim is to partition the rows such that as few columns (i.e. nets) as possible are broken. Fortunately, 1D partitioning is a standard option of Mondriaan. Second, Mondriaan penalizes every additional cut of a broken net. This is because each additional column part corresponds to an extra processor involved in the handling of the matrix column in a parallel computation. In the present application, however, the situation is different: once a program becomes an interface program, and serves at least one outside mod-

ule, it does not matter how many such modules it serves. Therefore, we can further break the already broken nets for free. Of course, this will have large effects on our minimization procedure. We modified Mondriaan in various places to take this into account. Third, the work load assumed by Mondriaan is just the number of nonzeros in the corresponding matrix part. In the present application, the workload is 1 for all programs, i.e., for all matrix rows. This was a relatively easy change, since internally Mondriaan already uses arbitrary weights (to enable merging vertices).

4 Comparison of the partitioning methods

4.1 Integer linear programming

We used CPLEX 6.3 (most recent version is 9.0, see [4]) on a 2.8 GHz Intel Xeon processor to solve the ILP model of the call-graph problem. To improve the running times, we provided CPLEX with a *branching order*, specifying that it should branch on x_{vl} before x_{wl} and on y_{vl} before y_{wl} if the outdegree of v in D is larger than the outdegree of w in D, and on y-variables before x-variables.

Table 1 shows the results. The numbers $|V|$ given are less than those for the original call graph, because vertices without outgoing arcs (corresponding to empty rows in the adjacency matrix) were removed beforehand. These programs can be assigned to any module without affecting the interface size $|I|$. The value $L = 8$ was chosen, because typically L is in the range 5–10, and because the current version of Mondriaan requires L to be a power of 2. The value of K was chosen such that no module would have more than 20% extra work compared to the average work of a module: $K = \lfloor 1.2|V|/L \rfloor$, where $|V|$ refers to the original call graph. We removed the small problem `Java2` with $V = 19$ and $|A| = 47$ from our test set, because it is infeasible for the parameter of 20% we chose; it would lead to $K = 2$, so that $KL = 16 < V$. (Of course, we can still find a solution if we are willing to accept more than 20% extra work.) The table gives the best result $|I|$ found, a lower bound on the best result possible, and the fraction $|I|/|V|$ of programs that are actually interface programs in the best solution, where $|V|$ refers to the original call graph.

We terminated problems `Java3` and `Cobol4` after about 4 days of CPU time; we expected that the gap between the best solution and the lower bound was going to be closed only at an extremely slow rate. Problem `Cobol2` aborted due to lack of memory; here, our current strategy for breaking symmetry apparently failed. But inspection of the last lower bound revealed that there could not be a solution better

Problem	$\|V\|$	$\|A\|$	K	L	best $\|I\|$	lower bound	best $\|I\|/\|V\|$(%)	running time (s)	remarks
`Java1`	144	422	23	8	26	26	16.5	2.04×10^2	
`Java3`	837	5252	127	8	251	230	29.5	3.59×10^5	terminated
`Java4`	15	39	2	8	11	11	68.8	0.22	
`Cobol1`	947	1900	209	8	13	13	0.9	1.06×10^3	
`Cobol2`	449	659	81	8	6	6	1.1	7.51×10^4	aborted
`Cobol3`	1145	2686	203	8	51	51	3.8	3.78×10^5	
`Cobol4`	1100	2951	167	8	32	28	2.9	3.60×10^5	terminated

Table 1: Integer linear programming results

than the one found anyway. Considering intermediate results, the best solution found for the larger problems after a day of CPU time was 251 for `Java3`, 57 for `Cobol3`, and 47 for `Cobol4`.

4.2 Multilevel hypergraph partitioning

Table 2 presents the results for 10 runs of the program Mondriaan, version 1.01, modified for this purpose. Mondriaan was run on an 867 MHz Apple PowerBook G4 computer running MacOS 10.2. Since Mondriaan uses a random number generator, we can run it with different random number seeds and get different solutions. The table shows the best result obtained in 10 runs, the average result, and also the average running time. The default settings of Mondriaan were used, except that the program was run in 1D mode and with random seed. An important default is that the multilevel algorithm moves over to the Kernighan–Lin algorithm when the number of vertices is 200 or less. This means that problems `Java1` and `Java4` were in fact solved by pure Kernighan–Lin. The number $|V|$ is the number of the original call graph; $|A|$ is the number of nonzeros of the matrix extended by a unit diagonal. The K and L values were chosen identical to those for the ILP solution. Therefore, the number of interface programs $|I|$ obtained by the two methods can be compared. (For the timings, the difference between the computers used in our experiments must be taken into account.)

Comparing Tables 1 and 2, we note that 5 out of the 7 feasible problems were solved to optimality by the ILP method and 1 by hypergraph partitioning (HP). The results of the ILP method are always better than those of HP, but never by more than a factor 1.63. The HP method on the other hand is much faster than the ILP method; the solution is almost instantaneous. The HP results can be improved by fine-tuning the Mondriaan parameters for the application at hand, instead of using the defaults which were chosen to obtain good performance for a wide range of applications. A quick trial of a few different parameter settings reduced $|I|$ for `Cobol4` from 52 to 47; further reduction should be possible.

Figure 4 compares the number of interface programs obtained by the ILP and HP methods. All ILP solutions are within a factor 1.14 from optimal; all HP solutions within a factor 1.86. Note that in fact they may even be closer, since the comparison is with a lower bound, not necessarily a known minimum. Figure 5 shows a solution obtained by the HP method for the problem `Java1`.

Problem	$\|V\|$	$\|A\|$	best $\|I\|$	avg $\|I\|$	best $\|I\|/\|V\|$ (%)	running time (s)
`Java1`	158	580	30	30.7	19.0	0.06
`Java3`	851	6103	275	283.2	32.3	0.54
`Java4`	16	55	11	11.2	68.8	0.001
`Cobol1`	1398	3298	17	22.4	1.2	0.33
`Cobol2`	545	1204	10	11.5	1.8	0.12
`Cobol3`	1357	4043	69	74.6	5.1	0.34
`Cobol4`	1116	4067	52	56.5	4.7	0.41

Table 2: Hypergraph partitioning results

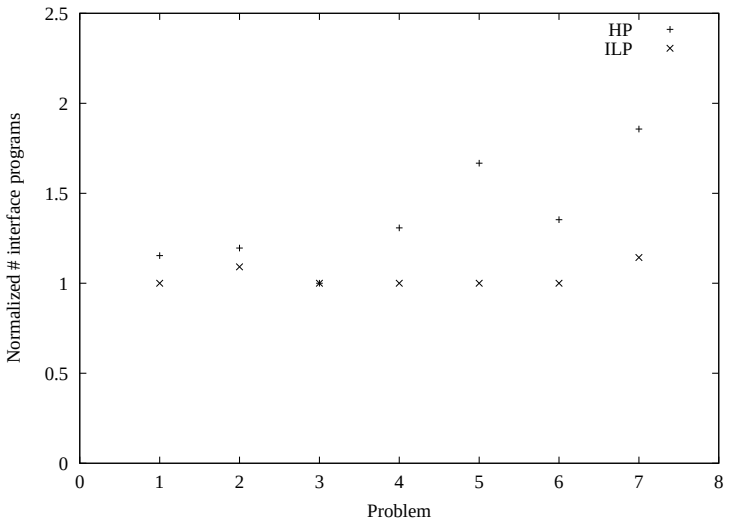

Figure 4: Number of interface programs obtained by the integer linear programming (ILP) method and the hypergraph partitioning (HP) method. The results are normalized by the lower bound provided by the ILP method. A value 1.0 means that the solution is guaranteed to be optimal. The problems are numbered 1–7, which corresponds to `Java1`, `Java3`, `Java4`, `Cobol1`–`Cobol4`.

5 Conclusions

Splitting a large software system into smaller and more manageable units has become an important problem and a challenging task for many organizations. We were introduced to this problem during the Study Group Mathematics with Industry 2005, where it was presented by the Software Improvement Group (SIG). Apart from the information specific to the application, the basic structure of a software system is given as a directed graph with vertices representing the programs of the system and arcs representing calls from one program to another. The question of generating a good partitioning into smaller modules becomes a minimization problem for the number of programs being called by external programs.

During the one week of the study group and in the six weeks of continuing investigations afterwards, we were able to give a clear mathematical description of the problem and to bring in some fresh ideas and new methods. We have presented two different solution strategies, which both seem to be a suitable and valuable tool for the intended applications. The formulations we gave reduce the problem to standard problems in discrete optimization; this makes it possible to apply some state-of-the-art software packages and to deal successfully with the real-world examples which were provided by SIG.

Two different approaches turned out to be promising to tackle the problem. First, we gave an equivalent formulation as an integer linear programming problem with 0–1 variables and used the software package CPLEX to implement this solution strategy. Theoretically, with this approach the problem can be solved to optimality, but this

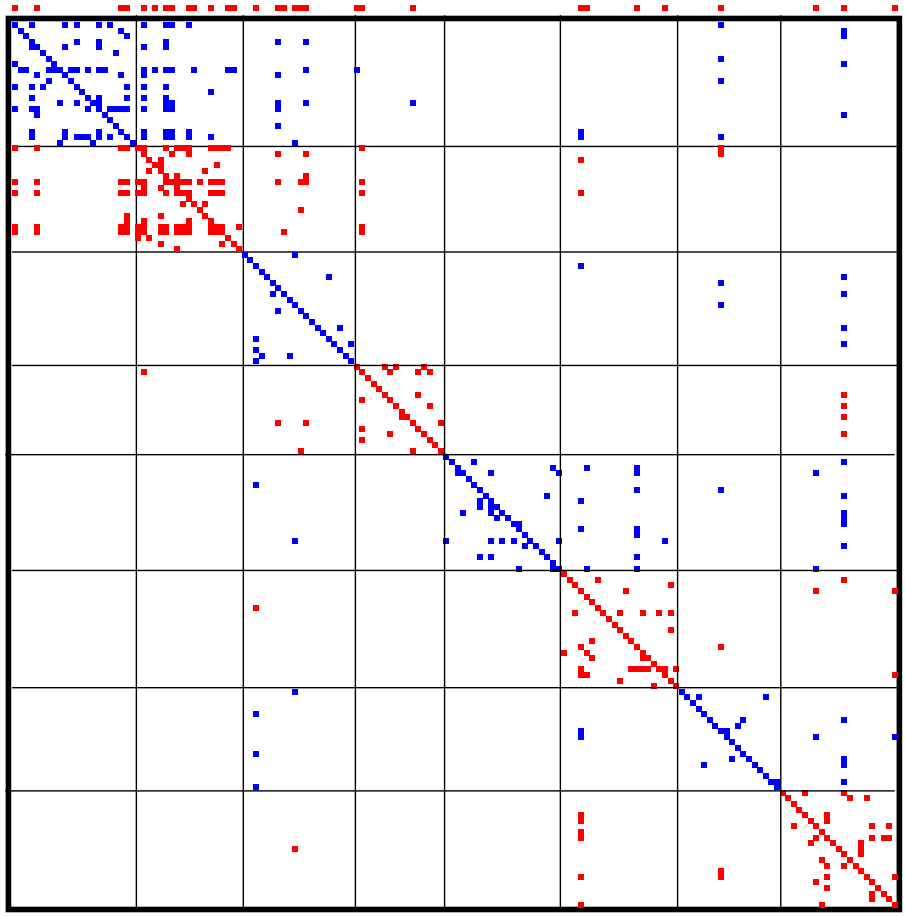

Figure 5: Permuted 158×158 adjacency matrix of the problem `Java1`. The rows were permuted such that rows (programs) belonging to the same module were brought together. Columns were permuted by the same permutation as the rows. Each block of rows corresponds to a subset of the vertices, and hence to a module. The number of modules is $L = 8$; the modules are shown by alternating coloring in red (light) and blue (dark). The number of programs in each module is 22, 19, 20, 16, 21, 21, 18, 21, respectively. The permutation corresponds to a solution with $|I| = 30$ interface programs produced by Mondriaan. Whether a program is an interface program or not can be read from the columns. Columns corresponding to interface programs are marked (above the matrix) and have at least one nonzero outside their diagonal block. Note that the solution method tries to confine all nonzeros to the diagonal blocks. Where this fails, the method does not attempt to limit the number of blocks involved; this explains the spread of the nonzeros over the different blocks.

becomes very costly with increasing size of the software system; obtaining efficient reformulations and a clever implementation becomes an important task. We succeeded to make this method work reasonably well for software systems of the size of the real-world examples.

Second, we have formulated the problem as a hypergraph partitioning problem. We have modified the package Mondriaan, recently developed at Utrecht University, and applied this to the examples given by SIG. This second approach is a heuristic method using a multilevel strategy, but it turns out to be very fast and to deliver solutions that are close to optimal.

Our two methods can drastically reduce the fraction of interface programs, in particular for Cobol systems, where the resulting fraction is at most 5.1%. For small problems, we recommend using the integer linear programming method, perhaps speeding up the solution process by starting with a heuristic solution produced by Mondriaan. For large problems, with thousands of vertices in the call graph, multilevel hypergraph partitioning such as done by Mondriaan is the only realistic option. Here, the performance can be improved by fine-tuning, perhaps aided by experience with smaller problems. Having knowledge of lower bounds or optimal solutions such as provided by the ILP method for smaller problems can be of tremendous help in the fine-tuning.

The problem presenter SIG apparently appreciated the solutions proposed in this report as well as the insight which they could gain from our investigations. Conversely, we have profited from this well-prepared problem which led us to new interesting questions such as for example the comparison between an exact and a heuristic method in a realistic situation. We would be pleased if the strategies presented here would have some impact on the applications and we hope that this collaboration stimulates further joint work between mathematics and industry.

References

[1] T. N. Bui and C. Jones, "A heuristic for reducing fill-in in sparse matrix factorization", in *Proceedings Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 445–452, SIAM, Philadelphia, 1993.

[2] A. E. Caldwell, A. B. Kahng, and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning", in: *Proceedings Asia and South Pacific Design Automation Conference*, pp. 661–666, ACM Press, New York, 2000.

[3] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication", *IEEE Transactions on Parallel and Distributed Systems*, **10** (7), pp. 673–693 (1999).

[4] `http://www.ilog.com/products/cplex/`

[5] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell System Technical Journal*, **49**, pp. 291–307, (1970).

[6] C. H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs NJ, 1982

[7] A. Schrijver, Theory of Linear and Integer Programming, Wiley, New York, 1986.

[8] B. Vastenhouw and R. H. Bisseling: “A two-dimensional data distribution method for parallel sparse matrix–vector multiplication”, *SIAM Review*, **47** (1), pp. 67–95 (2005).

[9] L. A. Wolsey, Integer Programming, Wiley, New York, 1998.