Adaptive Memory Search for Boolean Optimization Problems

Lars M. Hvattum
Molde College, 6411 Molde, Norway.
Lars.M.Hvattum@himolde.no

Arne Løkketangen
Molde College, 6411 Molde, Norway.
Arne.Lokketangen@himolde.no

Fred Glover
Leeds School of Business, UCB 419, University of Colorado, Boulder, CO 80309, USA
Fred.Glover@Colorado.edu

Abstract

We describe a simple adaptive memory search method for Boolean Optimization Problems. The search balances the level of infeasibility against the quality of the solution, and uses a simple dynamic tabu search mechanism. Computational results on a portfolio of test problems taken from the literature are reported, showing very favorable results, both in terms of search speed and solution quality.

1 Introduction

Boolean Optimization Problems (BOOP) represent a large class of binary optimization models, including weighted versions of Set Covering, Graph Stability, Set Partitioning and Maximum Satisfiability problems. These problems are NP-hard, and the use of heuristic search methods are highly competitive for even moderately sized instances.

We describe a reasonably simple iterative search procedure for this class of problems, using adaptive memory and learning principles derived from tabu search. Guidance for the search is based on strategic oscillation around the feasibility boundary, coordinating the interplay between changes in objective function values and changes in primal feasibility. This is then modified by short term tabu criteria, together with the use of periodic restarting to provide a rudimentary diversification process.

Previous heuristic work on this problem is mainly by Davoine, Hammer and Vizvári (2001). They use a greedy heuristic based on pseudo-boolean functions, with cutting off of local optima solutions reached. Their approach is similar to Lagrangean relaxation, using a DNF (*disjunctive normal form*) representation. We base our computational testing on their test case portfolio, and our computation results are compared with theirs, as well as with XPRESS/MP (http://www.dash.co.uk/) and CPLEX (http://www.ilog.com/products/cplex/).

This introduction is followed in Section 2 by BOOP problem formulations. Section 3 describes our approach and preliminary testing for search parameter settings, while the computational results are in Section 4. The conclusions are summarized in Section 5, together with some avenues for further work.

2 Problem Formulation

The Boolean Optimization Problem (BOOP), first formulated in Davoine, Hammer and Vizvári (2001), is based on logical expressions in prepositional, first-order logic, with an extra cost (or profit) associated with the variables having a *true* (or *false*) value. One formulation can be (assuming maximization)

$$Max\, z = \sum_{i=1}^{N} (c_i \,|\, x_i = true \,/\, false)$$

such that

$$\Phi(x) = \Phi(x_1, ..., x_N) = \begin{cases} true \\ false \end{cases}$$

where $\Phi(x)$ is the logical expression, and N the number of variables. The solution to this problem is then the set of truth value assignments to the x_i that yields the highest objective function value z, while satisfying the logical expression. The logical expression can in general be arbitrary, but we restrict ourselves to formulations in *conjunctive normal form, CNF* (the *disjunctive normal form* can be obtained by a simple transformation). To be informal, a BOOP can be regarded as a *satisfiability problem* (SAT) with an extra objective function added on. (For more info on SAT, see e.g. Cook, 1997 and Du et al., 1997.)

To be able to treat this as a more traditional optimization problem, using numbers instead of truth values, we let the logical value *true* be represented by 1, and the value false be represented by 0, giving us the following objective function.

$$Max\, z = \sum_{i=1}^{N} c_i x_i$$

The logical function $\Phi(x)$, in CNF, consists of a set of conjunctions of clauses $\Phi = c_1 \wedge c_2 \cdots \wedge c_M$, where each clause is a disjunction of complemented and uncomplemented variables, with M being the number of clauses. As a simple example, let

$$\Phi = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_3)$$

Replacing true/false with 1/0, disjunction with +, representing each conjunction as a separate constraint row, and splitting each variable into its complemented and uncomplemented occurrences, we get the following constraint set for the example, where the variable pair y_i and $y_{i\#}$ represents x_i.

$$y_1 + y_2 \geq 1$$
$$y_1 + y_{3\#} \geq 1$$

$$y_i + y_{i\#} = 1$$

Our final model is then

$$Max\, z = \sum_{i=1}^{N} c_i x_i \qquad (1.1)$$

s.t.

$$Dy \geq 1 \qquad (1.2)$$

$$y_i + y_{i\#} = 1 \qquad (1.3)$$

where D is the 0-1 matrix obtained by substuting the y's for the x_i's. The last constraint (1.3) is handled implicitly in the search heuristics we describe.

3 Adaptive Memory (Tabu) Search

The search we have implemented is based on an elementary form of tabu tenure, and a simple self-adapting move evaluation function. This move evaluation function tries to keep the search focus around the infeasibility boundary, while at the same time maintaining a good objective function value.

3.1 Search Implementation

Our implementation of the search process has the following basic components. The emphasis has been to have a simple implementation, and incorporate more sophisticated mechanisms in future work. Thus, for example, we use random starting solutions and random re-starts, both of which can be improved in the tabu search setting along the lines indicated in Glover and Laguna (1997).

1. The *starting solution* (or starting point) is based on a random assignment to the variables. As this solution might be primally infeasible, the search must be able to move in infeasible space. (Davoine, Hammer and Vizvári, 2001, used a quite complex Lagrangean based linear approximation constructive heuristic to obtain feasibility).
2. A *move* is the flip of a variable. A flip means assigning the opposite value to a variable. (i.e. change $1 \rightarrow 0$ or $0 \rightarrow 1$).
3. The *search neighborhood* is the full set of possible flips, with a *neighborhood size* of $|N|$, the number of variables.
4. *Move evaluation* is based on both the change in objective function value, and the change in amount of infeasibility.
5. The *move selection* is greedy (i.e. take the best move according to the move evaluation).
6. Simple *tabu* and *aspiration criterion* are enabled.
7. A random restart is applied after a certain number of moves, to *diversify* the search

8. The *stopping criterion* is a simple time limit or a cutoff on the number of allowable flips.

3.2 Tabu and Aspiration criteria

As moves consist of flipping variables, the change in the value of the objective function, Δz, changes sign almost every move. This causes very many local optima to be visited by the search, and using a tabu criterion is thus highly beneficial. There are many ways to apply tabu criteria to a search. Our choice of tabu criterion is an elementary one of not flipping a variable that has recently been flipped. Our key interest is to keep the tabu mechanisms simple, while obtaining good search guidance. It is important to find an efficient range for the tabu tenure (TT), and to change this TT dynamically, since a static TT might be too limiting. Suitable values for the tabu tenure are identified in 3.4. For a treatment of these issues in tabu search generally, see Glover and Laguna (1997).

Our aspiration criterion operates by permitting an otherwise tabu move leading to a new best solution. In section 3.4 we illustrate the benefit of using this simple aspiration.

3.3 Adaptive move evaluation function

The move evaluation function for each possible move, F_{Mi}, has two components. One is the change in objective function value. The cost coefficients, c_i, are initially normalized to lie in the range (0,1). This means that the change in objective function value per move, Δz_i, is in the range (-1, +1).

The other component is the change in the number of violated clauses (or constraint rows), for the flipping of each variable. This number, ΔV_i will usually be a small positive or negative integer, and can be found from the change in a standard surrogate constraint function. (See e.g. Løkketangen and Glover, 1996.)

These two components are combined so as to give a balanced view to maintaining primal feasibility and a good objective function value. The emphasis between the two components is changed dynamically to keep the search around the feasibility boundary.

This gives the following move evaluation function:

$$F_{Mj} = \Delta V_i + w * \Delta z_i$$

The value of w, the adaptive component, is initially set to 1. It is adjusted after each move as follows:

- If the current solution is feasible: $w = w + \Delta w_{inc}$
- If the current solution is not feasible, and $w > 1$: $w = w - \Delta w_{dec}$

Separate values are used for the increment and decrement. Suitable values for the weight modifiers Δw_{inc} and Δw_{dec} are found in 3.4. The effect of the adaptation is to induce a strategic oscillation around the feasibility boundary. A different approach appears in Glover and Kochenberger (1996), where the oscillation is coupled with the use of a critical event memory, forcing the search into new areas.

3.4 Preliminary testing for setting of search parameters

Even though our implemented search is quite simple, there are quite a few choices to be made regarding search parameter values. Doing a full search on the full set of test cases (5485 in all,

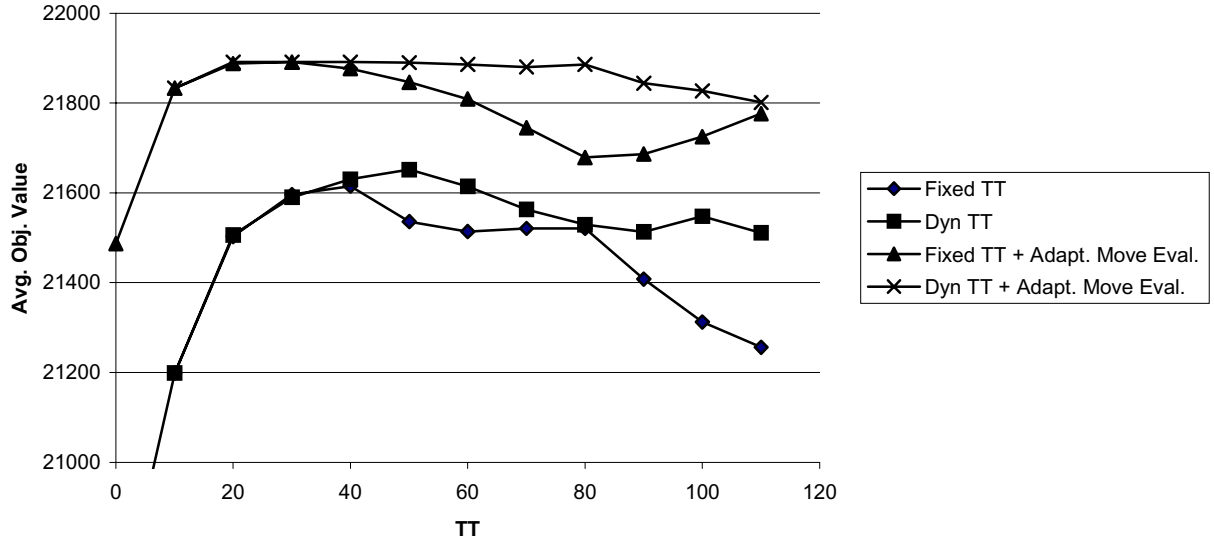

Figure 1. Tabu tenure

see Section 4) for all the possible parameter values and levels of search effort is prohibitive. We have therefore chosen a small subset of test cases to tune our search parameter values on, and subsequently used these values for the full test set. The three test cases were chosen (rather arbitrarily) to be *small* (from class 4 - rn50m200t10s0c0num0, 50 variables, 200 clauses), *medium* (from class 38 - rn200m400t10s0c50num0, 200 variables, 400 clauses) and large (from class 38 - rn500m1000t25s0c50num0, 500 variables, 1000 clauses).

It should also be noted that the effects of, and values for, the different parameters are not independent, and hence we should ideally do a full search in the parameter space. As this also seems quite prohibitive, we have opted for a greedy approach, selecting good values for one search parameter at a time. The values for the other parameters are kept either at reasonable values, or at the best values found if the parameter already has been subjected to this search. The sequencing of testing is thus important, but we have not undertaken to account for this.

Not all results in this chapter are reported in full, but rather are summarized by describing relative performance.

Search for tabu tenure

To find good values for the tabu tenure, and the effect of adding dynamism to the TT, we ran a set of tests for each of the three test cases. Each test was run 20 times with different random seeds. Aspiration was included, but initially no other mechanisms. Figure 1 shows the average results for running with fixed TT on the selected *medium* test case, with TT ranging from 0 to 110. The optimum is at 21891. As can be seen, the best value is around 40. Also in the same figure is the average result when using a dynamic tabu tenure, dynamic move evaluation weight,

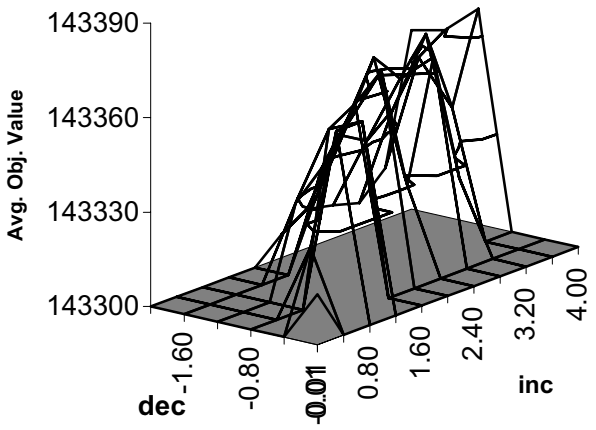

Figure 2. Relationship between w_{inc} and w_{dec}.

w, and both. In the case shown in figure 1, the dynamic TT is shown for ranges of the TT value between 10 and the number on the TT axis. At each new TT assignment, a random TT in this range is chosen. The Adaptive Move Evaluation tests used $\Delta w_{inc} = 0.1$, and $\Delta w_{dec} = 0.05$.

Evidently, the effects of these mechanisms are not independent, as a much shorter TT is needed when using the self-adapting move evaluation weight. The figure suggests that the search becomes quite insensitive to the actual TT range, when both dynamic TT and adaptive move evaluation weights are used. Graphs like those in figure 1 will of course be different for each instance. The tests for the other preliminary cases showed similar results, and a dynamic TT in the range [10-15] was used for further tests.

Search for adaptive move evaluation weights

To recap, the move evaluation function used is $F_{Mj} = \Delta V_i + w * \Delta z_i$, where the relative emphasis of the objective function value vs. the primal infeasibility level is controlled by the parameter *w*.This parameter changes value dynamically as explained in section 3.3. Of importance here is to find proper choices for incrementing and decrementing *w*, i.e. values for Δw_{inc} and Δw_{dec}. What turned out to be relevant was not so much the sizes of these adjustments, but rather the ratio between them, $\Delta w_{inc} / \Delta w_{dec}$. This is illustrated in Figure 2, where the average objective function value is shown for different combinations of adjustments for the selected *large* test case. Very similar pictures could be drawn for the other test cases. The best ratio is around 2.5, and for the computational testing we used $\Delta w_{inc} = 0.90$ and $\Delta w_{dec} = 0.35$.

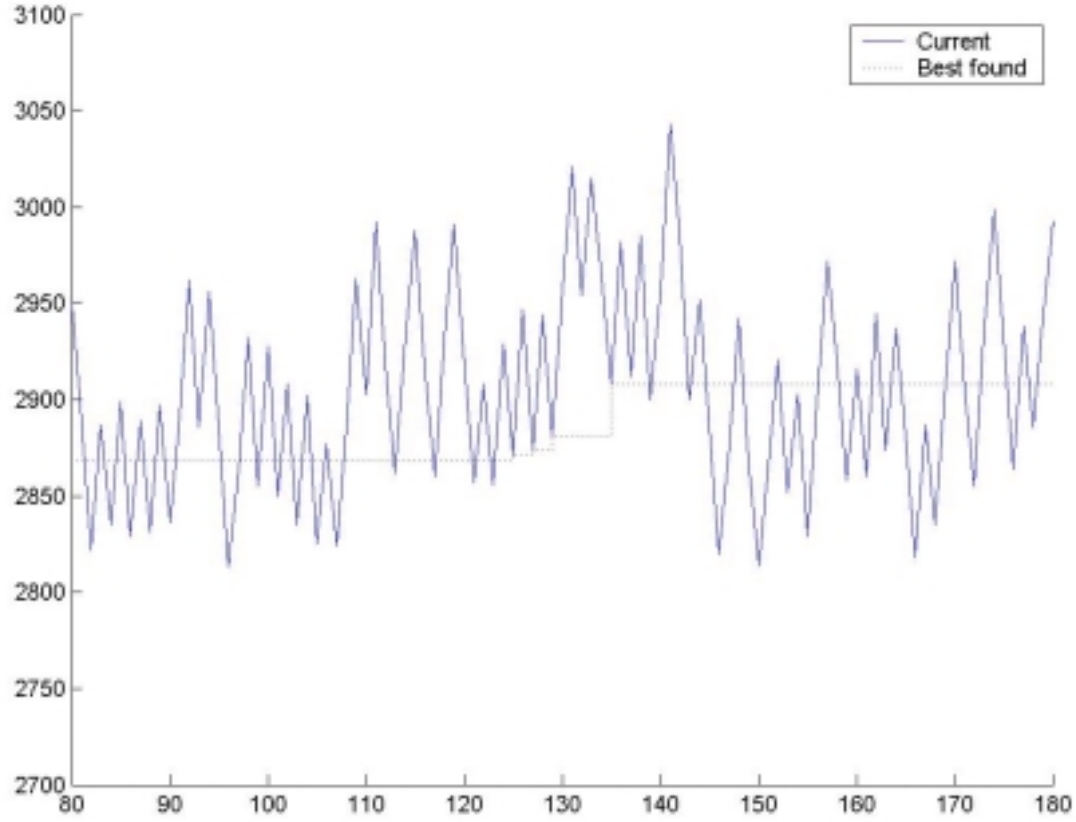

Figure 3. Objective function value and best value found, per iteration

Figure 3 shows the development of the objective function value for the selected *small* test case, together with the best objective function value found so far, for a part of the search. The search spends a large part of the time in infeasible space, finding new best solutions at points where it *enters* the feasible region. In a way, the search meanders around the feasibility boundary. This is also illustrated in Figure 4, showing the development of the adaptive component w of the move evaluation function, and the infeasibility level. In this case, the search is only feasible for one iteration before going back to infeasibility, and the ratio between feasible and infeasible iterations is about the same as the ratio between the chosen values for Δw_{inc} and Δw_{dec}.

The adaptive weight, w, is not reset when the search is restarted (see below). As it is self adjusting it has no discernible effect.

The effect of aspiration

The use of aspiration criteria is deemed to be very important in tabu search, as otherwise the tabu criteria restricts the search too much. This claim is seldom documented in the literature. The

effect of our choice of aspiration criterion (new best solution found), is shown in Table I for the first 13 classes of test cases (see Section 4 for details about the test cases). The search was for 5 seconds per test case, with restart as outlined below. These are among the smaller test cases, where optimality is easily reached for most instances. Even though the results without the use of aspiration are better than than those reported by Davoine, Hammer and Vizvári (2001), still better results, both in terms of quality and time to find the best solution, are obtained when using the aspiration. (Other forms of aspiration criteria may of course prove superior to the one we implemented.)

When to restart

Without the use of specially designed diversification mechanisms, the search is likely to become less effective after a while, remaining in the same general area of the search space. Some diversification process is therefore usually warranted. Our choice of diversification is to simply restart from a randomly generated starting point. Tabu search normally counsels the use of more strategic forms of diversification, but in our testing we have elected to employ this rudimentary mechanism and focus on other issues. Thus the primary question in this instance reduces to deciding how long to search before restarting. In *reactive* tabu search (see e.g. Battiti 1996), the search keeps track of the solution space it is in, and diversification measures are instantiated when there indicators of stagnation, or of being trapped in a particular region. We ran a series of tests with different triggers for restarting the search in the three selected test cases, noting the

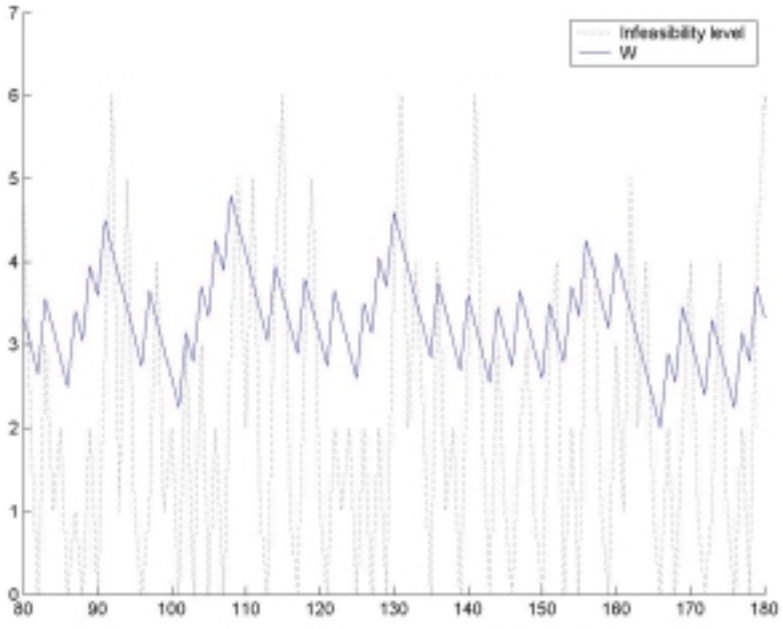

Figure 4. Development of *w* and infeasibility level

Table I. Effect of Aspiration

	No Asp as % of w/ Asp	No Asp Time to best	Asp Time to best
Class 1	99.985	0.05	0.01
Class 2	100.000	0.01	0.00
Class 3	100.000	0.02	0.00
Class 4	100.000	0.05	0.00
Class 5	99.992	0.02	0.01
Class 6	100.000	0.02	0.00
Class 7	100.000	0.04	0.00
Class 8	99.985	0.03	0.03
Class 9	100.000	0.02	0.00
Class 10	100.000	0.03	0.00
Class 11	99.955	0.11	0.08
Class 12	99.980	0.06	0.01
Class 13	99.998	0.03	0.00

time taken to find the best solution. Restarting was clearly better than not restarting. Again we elected for simplicity, basing the trigger for restarting on the numbers of iterations R_I since initiating the last restart (or the first start). The best value, R_I, in terms of iterations for restart was found to be correlated with N, the number of variables, and the average number of non-zero elements in each problem class, CL_{avg}.

The value used in our computational tests were thus

$$R_I = N^* CL_{avg}$$

4 Computational Results

To test our methods, we used the same set of 5485 test cases as Davoine, Hammer and Vizvári (2001). These can be obtained by anonymous ftp from *rutcor.rutgers.edu* in directory */pub/BOP*. We report our results in the same framework they used, to make comparisons easier.

There are three general classes of test cases, all randomly generated, in the following general classification:

- Random problems Class 01 to 49
- Graph Stability problems Class 50 to 54
- Set covering problems Class 55 to 63

The 49 *random* test cases can grouped into 3 sets, and within each set there are four sub-groups with 0%, 25%, 50% and 75% of clauses with negated variables.

- Class 01 – Class 13, 50 variables, 30 - 200 clauses, 240 instances per class

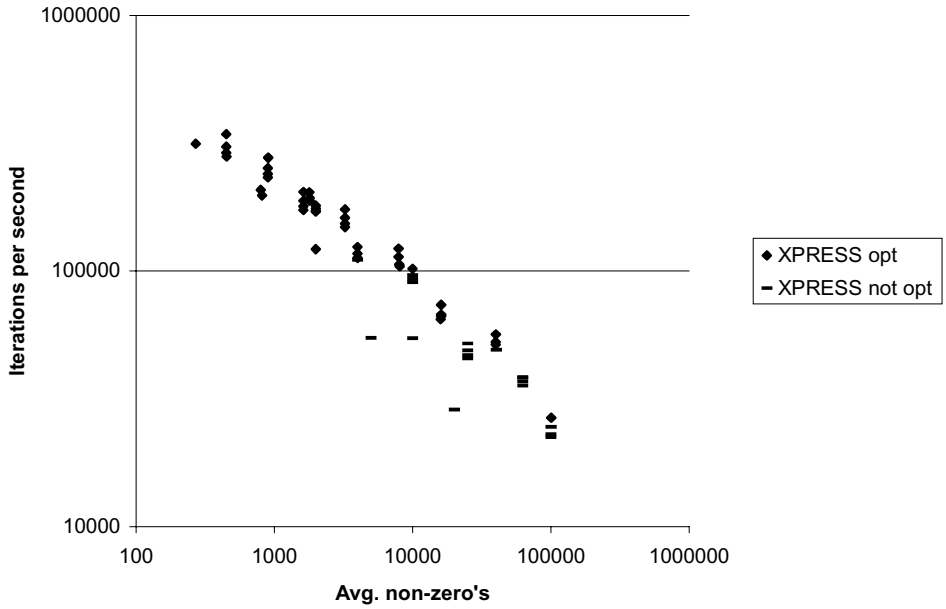

Figure 5. Search speed vs. problem size

- Class 14 – Class 22, 100 variables, 50 – 200 clauses, 240 instances per class
- Class 23 – Class 49, 100 – 500 variables, 400 – 2500 clauses, varying clause length, 5 instances per class

The *Graph Stability* problems are in Class 50 – Class 54, 100 – 1000 variables, 400 – 10000 clauses, 5 instances per class.

The *Set Covering* problems are in Class 55 – Class 63, 100 – 500 variables, 400 – 2500 clauses, 5 instances per class. Our findings suggest feasibility is easily obtained for all the instances.

We compare our results to those obtained by Davoine, Hammer and Vizvári (2001), as well as with XPRESS/MP v.12 (http://www.dash.co.uk/) and CPLEX v 6.5 (http://www.ilog.com/products/cplex/). Our code is implemented in Visual C++ 6.0, running on a standard 1 GHz Pentium 3 PC with Microsoft Windows 2000. Our CPLEX tests were on the same machine, while XPRESS/MP was run on a 400 MHz Sun UltraSparc. (A simple whetstone test deemed the PC to be about 5 times faster than the Sun). Davoine, Hammer and Vizvári (2001) ran their experiments on a 50 MHz Sun Sparcstation 5, and used CPLEX 6.0 for comparisons. We have unfortunately not been able to run their code on our machine. Precise comparison is therefore rather difficult, but as we report both solution time and quality, reasonable conclusions can be made.

We ran the following series of tests on all the test cases (the time is the maximum allotted for each instance):

- Adaptive local search, 5 seconds
- Adaptive local search, 60 seconds
- XPRESS/MP v. 12 for 4 hours
- CPLEX v. 6.5 for 4 hours

For the search we used dynamic TT (10-15), $R_I = N^*CL_{avg}$, $\Delta w_{inc} = 0.90$ and $\Delta w_{dec} = 0.35$. The 5 second tests were run 10 times, and the average is reported. CPLEX produced very similar results.

Figure 5 shows the number of flips per second for the different test case classes. As can be seen, even for the larger instances we manage more than 20000 flips per second. Also shown in the graph are the test cases where XPRESS/MP finds the optimum in reasonable time (less than 4 hours). For CPLEX we got very similar results.

Overall outcomes are shown in Table II. Our method is under the heading ALS. The percentages are expressed as a % of the CPLEX results reported by Davoine, Hammer and Vizvári (2001), even though our XPRESS and CPLEX runs produced better results. This is done to enable easier comparisons. The rows represent the small and large random instances, the graph stability and set covering instances, and finally all test cases.

More detailed results are given in Tables III to IX. Explanation of the column headings is given at the start of the appendix. For both the 5 second and 60 second searches we show the results compared to CPLEX as reported by Davoine, Hammer and Vizvári (2001), and also show the average time taken to find the best. For the small test cases it is evident that virtually no time is used to find the best solution, while for the larger test cases a large fraction of the allotted time is spent before finding such a solution. Consequently more search time might be beneficial for the larger test cases. In 6 of the 5280 easy test cases (where XPRESS used less than 1 second), our method did not find the globally optimal solution. Although the best solutions found by our method in these 6 cases were obtained very quickly and were very close to global optimality, even allotting a 60 second run time did no permit us to find the global optimum. This provides a clear indication that a better diversification mechanism than random restart is needed. Somewhat

Table II. Overall computational results

	Davoine et al.	**ALS – 60 sec**	**XPRESS**
Class 1 – 22	99.161	100.001	100.002
Class 23 – 49	100.440	101.215	101.127
Class 50 – 54	102.806	106.982	85.357
Class 55 – 63	101.238	102.465	102.237
Class 1 – 63	100.295	101.450	99.641

surprisingly, given our emphasis on a simple implementation, the 5 second search limit produces very competitive results even for the larger test cases.

Proven optimal solutions are shown in **bold** (meaning that all instances in the class are solved to optimality). Our method finds better results for *all* test classes where XPRESS/MP does not find the optimum, even when limiting the search to 5 seconds. In Table VIII are shown the *graph stability* instances. The larger of these (classes 52, 53, and 54) give our best comparative results. As in the case of the smaller problems, we believe that better results on these instances can be found with better diversification methods.

Overall our search is better than Davoine, Hammer and Vizvári (2001), both in terms of solution quality and in terms of search speed. For the larger test cases, we are also clearly better (and faster) than XPRESS and CPLEX. (XPRESS usually spends a significant fraction of its allotted 4 hours before finding its best results).

5 Conclusions and future work

Boolean Optimization Problems represent a large class of binary optimization problems, and consequently it is important to be able to solve reasonably large instances quickly and efficiently. We have described an adaptive memory (tabu search based) metaheuristic to solve these kinds of problems, designed to incorporate a strategic oscillation around the infeasibility boundary that coordinates tradeoffs between feasibility and the objective function value. Our method clearly outperforms the specialized procedure previously developed for these problems, both in terms of solution quality and solution time, and also beats the commercial solvers XPRESS/MP and CPLEX.

Our approach does not yet incorporate some of the more advanced components of the tabu search framework, notably lacking efficient diversification processes. In general, the computational results suggest that better outcomes can be found with more sophisticated long term strategies.

We anticipate that diversification methods based on learning and adaptive memory, specifically relying on the use of surrogate constraint evaluations and frequency based mechanisms, will provide significant performance gains. Constructive solvers founded on the same principles also provide an interesting avenue to pursue.

References

S. A. Cook. (1971). “The complexity of theorem-proving procedures”. Proceedings of the Third ACM Symposium on Theory of Computing, pp 151-158.

Davoine, Thomas, Peter L. Hammer and Béla Vizvári. (2001). “A Heuristic for Boolean optimization problems”. Forthcoming in *Journal of Heuristics*.

Glover, Fred and Gary Kochenberger.(1996). “Critical Event tabu Search for Multidimensional Knapsack Problems”, In I.H. Osman and J.P. Kelly, editors, *Meta Heuristics: Theory and Applications*, Kluwer Academic Publishers, pp 407 – 427.

Glover, Fred and Manuel Laguna. (1997). **Tabu Search**. Kluwer Academic Publishers.

Løkketangen, Arne and Fred Glover. (1997). “Surrogate Constraint Analysis - New Heuristics and Learning Schemes for Satisfiability Problems”. In: *Satisfiability Problem: Theory and*

Applications. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol 35.

Du, Dingzhu, Jun Gu and Panos Pardalos (Eds.). (1997). *Satisfiability Problem: Theory and Applications.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol 35.

Battiti, Roberto.(1996). "Reactive search: Toward self-tuning heuristics**".** In V. J. Rayward-Smith, editor, **Modern Heuristic Search Methods**, chapter 4, pages 61--83. John Wiley and Sons Ltd, 1996.

Appendix – Computational result tables

Due to lack of space in the column headers, the following is an explanation of some of the acronyms used in the minor column names.

Terms	Number of terms in CNF form. Same as *rows.*
%NL	% of terms with negated literals (DNF).
%	Average results of each test case class compared to the CPLEX runs by Davoine et al.
Secs	Average overall time used in seconds (Davoine et al).
Minutes	Average overall time used in minutes (Davoine et al).
Sec-B	Average seconds to best value found (ALS and XPRESS/MP).

Numbers in **bold** signifies optimal results.

Table III. Random instances, 50 variables, 240 instances per class

			Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Terms	% NL	%	Secs	%	Sec-B	%	Sec-B	%	Sec-B
Class 01	30	0	99.92	29	100.001	0.01	**100.002**	0.08	**100.002**	0.00
Class 02	50	0	99.84	36	**100.003**	0.00	**100.003**	0.00	**100.003**	0.00
Class 03	100	0	99.66	49	**100.004**	0.00	**100.004**	0.00	**100.004**	0.36
Class 04	200	0	99.43	74	**100.005**	0.00	**100.005**	0.00	**100.005**	6.36
Class 05	50	25	99.87	36	100.000	0.01	**100.002**	0.20	**100.002**	0.00
Class 06	100	25	99.75	29	**100.003**	0.00	**100.003**	0.00	**100.003**	0.13
Class 07	200	25	99.43	74	**100.005**	0.00	**100.005**	0.00	**100.005**	2.59
Class 08	50	50	99.93	36	99.996	0.01	**100.001**	0.63	**100.001**	0.00
Class 09	100	50	99.80	52	**100.004**	0.00	**100.004**	0.00	**100.004**	0.00
Class 10	200	50	99.65	75	**100.005**	0.00	**100.005**	0.00	**100.005**	0.66
Class 11	50	75	99.96	26	99.991	0.06	99.993	0.11	**100.000**	0.00
Class 12	100	75	99.91	45	99.992	0.01	99.993	0.01	**100.002**	0.00
Class 13	200	75	99.79	67	**100.004**	0.00	**100.004**	0.00	**100.004**	0.02

Table IV. Random instances, 100 variables, 240 instances per class

			Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Terms	% NL	%	Secs	%	Sec-B	%	Sec-B	%	Sec-B
Class 14	50	0	98.85	25	**100.000**	0.00	**100.000**	0.00	**100.000**	0.00
Class 15	100	0	97.82	36	**100.001**	0.00	**100.001**	0.00	**100.001**	0.10
Class 16	200	0	96.92	58	**100.002**	0.01	**100.002**	0.01	**100.002**	5.95
Class 17	100	25	98.43	41	**100.001**	0.01	**100.001**	0.00	**100.001**	0.01
Class 18	200	25	97.34	54	**100.001**	0.00	**100.001**	0.00	**100.001**	1.08
Class 19	100	50	98.80	30	**100.001**	0.01	**100.001**	0.01	**100.001**	0.00
Class 20	200	50	98.04	48	**100.001**	0.00	**100.001**	0.00	**100.001**	0.26
Class 21	100	75	99.42	25	**100.000**	0.00	**100.000**	0.00	**100.000**	0.00
Class 22	200	75	98.99	42	**100.001**	0.00	**100.001**	0.00	**100.001**	0.00

Table V. Larger random instances, 5 instances per class,
25% Terms with negated literals (DNF)

				Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Vars	Terms	TermLn	%	Minutes	%	Sec-B	%	Sec-B	%	Sec-B
Class 23	100	400	[5,5]	100.78	1.4	**101.933**	0.03	**101.933**	0.03	**101.933**	225.80
Class 24	100	400	[10,30]	99.54	2.1	**100.004**	0.01	**100.004**	0.00	**100.004**	26.60
Class 25	200	400	[10,10]	101.07	1.9	**102.610**	0.27	**102.610**	0.13	**102.610**	3013.60
Class 26	200	400	[20,60]	99.33	5.0	**100.034**	0.20	**100.034**	0.25	**100.034**	214.40
Class 27	200	1000	[10,10]	109.71	5.7	111.189	0.26	111.196	11.85	110.938	6595.40
Class 28	200	1000	[20,60]	100.52	11.0	**101.077**	0.06	**101.077**	0.04	**101.077**	5379.60
Class 29	500	1000	[25,25]	100.49	17.0	101.281	2.03	101.290	6.41	101.065	3821.20
Class 30	500	1000	[50,150]	100.11	45.0	100.391	1.25	100.391	3.20	100.368	9581.79
Class 31	500	2500	[25,25]	100.32	54.0	101.437	1.73	101.459	13.01	100.974	5634.60

Table VI. Larger random instances, 5 instances per class,
50% Terms with negated literals (DNF)

				Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Vars	Terms	TermLn	%	Minutes	%	Sec-B	%	Sec-B	%	Sec-B
Class 32	100	400	[5,5]	98.35	1.5	**100.359**	0.02	**100.359**	0.02	**100.359**	54.40
Class 33	100	400	[10,30]	99.31	1.9	**100.000**	0.00	**100.000**	0.00	**100.000**	0.20
Class 34	200	400	[10,10]	98.98	2.9	**100.031**	0.06	**100.031**	0.10	**100.031**	60.20
Class 35	200	400	[20,60]	99.49	4.8	**100.000**	0.03	**100.000**	0.04	**100.000**	24.80
Class 36	200	1000	[10,10]	103.84	6.5	105.074	0.38	105.074	0.48	104.904	5124.60
Class 37	200	1000	[20,60]	99.99	12.0	**100.543**	0.15	**100.543**	0.04	**100.543**	2456.19
Class 38	500	1000	[25,25]	100.10	17.0	100.982	0.84	100.987	3.50	100.868	7597.20
Class 39	500	1000	[50,150]	100.01	34.0	100.299	0.93	100.299	1.17	100.267	10052.60
Class 40	500	2500	[25,25]	100.48	50.0	101.613	1.46	101.621	6.67	101.056	5864.20

Table VII. Larger random instances, 5 instances per class,
75% Terms with negated literals (DNF)

				Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Vars	Terms	TermLn	%	Minutes	%	Sec-B	%	Sec-B	%	Sec-B
Class 41	100	400	[5,5]	98.72	1.4	**100.012**	0.02	**100.012**	0.01	**100.012**	2.20
Class 42	100	400	[10,30]	99.69	1.7	**100.010**	0.02	**100.010**	0.02	**100.010**	0.60
Class 43	200	400	[10,10]	99.31	2.8	**100.000**	0.07	**100.000**	0.05	**100.000**	0.60
Class 44	200	400	[20,60]	99.65	4	**100.000**	0.01	**100.000**	0.02	**100.000**	2.20
Class 45	200	1000	[10,10]	100.82	5.6	**101.752**	0.12	**101.752**	0.04	**101.752**	1920.80
Class 46	200	1000	[20,60]	99.43	8.5	**100.000**	0.06	**100.000**	0.15	**100.000**	141.80
Class 47	500	1000	[25,25]	99.65	16	100.483	0.75	100.484	0.54	100.477	9128.60
Class 48	500	1000	[50,150]	99.80	28	**100.115**	0.56	**100.115**	0.23	**100.115**	3477.80
Class 49	500	2500	[25,25]	100.38	50	101.512	1.47	101.524	17.76	101.027	6357.60

Table VIII. Graph Stability Instances, 5 instances per class

			Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Vars	Terms	%	Minutes	%	Sec-B	%	Sec-B	%	Sec-B
Class 50	100	400	99.52	0.75	**100.000**	0.00	**100.000**	0.00	**100.000**	0.20
Class 51	200	1000	98.68	2.8	**101.502**	0.34	**101.502**	0.14	**101.502**	708.00
Class 52	500	2500	105.72	16	108.302	3.36	109.299	27.33	79.565	5602.20
Class 53	500	5000	105.04	32	111.917	3.12	113.215	34.71	79.487	7401.20
Class 54	1000	10000	105.07	152	108.440	3.91	110.895	31.36	66.231	8333.00

Table IX. Set Covering instances, 5 instances per class

				Davoine et. al.		**ALS – 5 sec**		**ALS – 60 sec**		**XPRESS-MP**	
Name	Vars	Terms	TermLn	%	Minutes	%	Sec-B	%	Sec-B	%	Sec-B
Class 55	100	400	[5,5]	101.49	2.2	**103.280**	0.02	**103.280**	0.03	**103.280**	522.00
Class 56	100	400	[10,30]	99.21	3.2	**100.000**	0.00	**100.000**	0.00	**100.000**	29.60
Class 57	200	400	[10,10]	102.61	4.5	104.162	0.43	104.162	0.77	104.038	3648.60
Class 58	200	400	[20,60]	99.34	9	**100.049**	0.07	**100.049**	0.10	**100.049**	411.60
Class 59	200	1000	[10,10]	104.28	9	107.362	0.58	107.363	0.74	106.508	6062.60
Class 60	200	1000	[20,60]	100.62	20	101.204	0.29	101.204	0.18	101.180	5084.20
Class 61	500	1000	[25,25]	100.38	27	101.545	1.81	101.559	12.16	101.235	9884.79
Class 62	500	1000	[50,150]	100.61	75	100.976	1.86	100.982	1.84	100.949	4269.80
Class 63	500	2500	[25,25]	102.60	62	103.571	1.48	103.585	28.58	102.891	8315.60