A Local Search Approach for Binary Programming : Feasibility Search

Samuel Souza Brito, Haroldo Gambini Santos
and Bruno Henrique Miranda Santos

Computing Department,
Universidade Federal de Ouro Preto (UFOP)
Ouro Preto – Minas Gerais - Brasil
samuel.ufop@gmail.com
haroldo@iceb.ufop.br
bruno_h_m_s@hotmail.com

Abstract In this paper we propose a Local Search approach for NP-Hard problems expressed as binary programs. Our search method focuses on the fast production of feasible solutions. The method explicitly considers the structure of the problem as a conflict graph and uses a systematic neighbor generation procedure to jump from one feasible solution to another using chains of movements. Computational experiments comparing with two open source integer programming solvers, CBC and GLPK, in MIPLIB 2010 instances, showed that our approach is more reliable for the production of feasible solutions in restricted amounts of time.

Keywords: Binary Programming, Heuristics, Local Search

1 Introduction

In this work we consider Binary Linear Programs, or Binary Programs (BP), which can be expressed as:

$$\begin{aligned} min.: & \\ & c^T x \qquad (1) \\ s.t.: & \\ & l \leq Ax \leq u \qquad (2) \\ & x_j \in \{0,1\} \ \ \forall j \in J \qquad (3) \end{aligned}$$

Where x is a vector of n binary variables with its associated cost vector c to be minimized (1). A is a matrix with dimension $m \times n$ expressing the constraint system where each constraint has a lower and upper bound expressed in vectors l and u respectively.

In spite of its simplicity, Binary Programming , is one of the most important techniques in Operations Research (OR). Some notable applications include

The Traveling Salesman Problem [1], Project Scheduling [2] and Computational Biology [3]. The availability of constantly improving optimization packages [4], some of which are open source [5], has made Binary Programming a great choice for OR practitioners.

Linear Programming based methods for Binary Programming typically work by solving series of linear programs using some variation of the Branch-and-Bound method [6]. This method works with fractional solutions but the systematic exploration of a tree of progressively restricted subproblems eventually produces an Integer Feasible solution. Since the fractional solution is usually useless for practical purposes, solvers are also being evaluated [7] considering their ability to quickly produce an Integer Feasible solution.

In order to obtain feasible solutions in acceptable computational time, this paper presents a hybrid heuristic. This approach is characterized by two phases: a constructive phase, which involves solving the maximum independent set problem and a local search phase. Both phases work with information provided by a conflict graph, created from the analysis of the constraints imposed by the problem input. They do not require a black-box linear solver or branch-and-bound family methods. Experiments with binary problems of the Mixed Integer Programming Library (MIPLIB) 2010 show that the proposed approach is able to produce more feasible solutions than the GNU Linear Program Kit (GLPK) and COIN-OR [8] CBC [9] solvers in restricted time intervals.

The paper is organized as follows: Section 2 presents related works, Section 3 presents a description of the proposed approach. In Section 4 computational experiments with MIPLIB are presented. Finally, Section 5 discusses future works and conclusions.

2 Related Works

One common approach to the development of heuristics for Integer Programming is to use the information from the linear programming relaxation alone or combined with some black-box integer programming solver. These methods solve series of linear programs iteratively [10–12]. One of them, called Feasibility Pump, is a smart and simple heuristic, proposed by Fischetti et al. [10]. The purpose of this heuristic is to find an initial feasible solution, even in difficult problems. The basic idea is to start with a relaxed linear solution and then make changes in the objective function to try to minimize the infeasibility related to the integrality constraints. So, this method is designed to pump the feasibility of a relaxed solution for an integer solution. Performed tests show that this method is able to find feasible solutions quickly and can be used in other methods to accelerate the search process.

Based in structure of Pure Integer Programming problems, in [13] is proposed a approach that uses a genetic algorithm. In this approach, a gene corresponds to a decision variable of the problem, represented by a bit array. Therefore, a chromosome is defined by a decision variable set, which the fitness is the objective function. The initial population is generated randomly, respecting the variables domains defined by constraints. Then, the method attempts to remove the infeasibilities exchanging the variable values by rounded values of the linear relaxation of the problem. The basic steps of a genetic algorithm are performed: crossover, mutation and selection. Crossover uses an one point operator. Mutation is made inverting a bit value selected randomly of a chromosome. Lastly, the selection phase is defined by a roulette based operation. Experiments were performed to investigate the behavior of the proposed method with example instances of Lingo 8.0 and the results were compared with this solver. The genetic algorithm was able to obtain better solutions only in 2 of 9 instances.

LocalSolver [14] is local search based heuristic commercial solver that uses local search to optimize linear and nonlinear binary problems. By default, LocalSolver performs a descent method as search strategy using autonomous movements. A Simulated Annealing based algorithm is also included. Both of these strategies are implemented in multithreading. Initial solution is found by a basic randomized greedy algorithm. *Autonomous movements*, which are k-flips movements, are used, generating movements similar to Ejection Chains. These movements preserve the feasibility of a solution. In its latest version, LocalSolver reaches better solutions than Gurobi and CPLEX solvers in some MIPLIB 2010 instances classified as hard. Both local search and the constructive algorithm are only superficially described by the authors, probably because of the commercial nature of the product.

Vassilev et al. [15] presents a hybrid heuristic algorithm for Mixed Integer Programming were each iteration has polynomial-time computing complexity. This algorithm searches for feasible integer directions and uses a linear solver for the continuous part. Three solutions are provided and the best of them is chosen. The method proceeds, iteratively generating subproblems which the feasible region is defined by all problem constraints satisfied at the current iteration. In these subproblems the objective function is one of the constraints which are not been satisfied yet. When a feasible solution was found, the objective function of the original problem is inserted on the next subproblems. Any solution found from this stage leads to a gradual improvement. Tests compare two versions of the method, one that uses feasible integer directions with one non-zero component and other that uses two of this components. The first variant shows solutions with better quality and execution time, besides occupying a smaller portion of memory.

3 The BP Local Search Solver

Before proceeding to a detailed description of our solver it is important to comment about the diversity of constraint types which can appear in BP problems and how it determines the hardness of finding a first feasible solution. Some quite common constraint types are:

Set Covering: $\sum x_i \geq 1$, x_i is binary
Set Packing: $\sum x_i \leq 1$, x_i is binary
Set Partition: $\sum x_i = 1$, x_i is binary

While for some BPs a feasible solution is trivial, e.g. Set Covering or Set Packing, different constraints can significantly complicate this initial step. The satisfaction of just one constraint can be a NP-Complete problem if it represents, for instance the Number Partitioning Problem. Problems where only few, *hidden*, subsets of all possible incidence vectors are feasible tend to be much harder. Set Partitioning problems are typical examples of this type.

Our solver discovers and explores relationships between variables both in the constructive phase and in the local search phase by means of conflict graphs, which will be explained in the next subsection.

3.1 Conflict Graphs

A fundamental information from BP used by Linear Programming (LP) based solvers to generate cuts and to strengthen the LP relaxation is the conflict graph [16]. In ou work the conflict graph is always used in the primal search space, both in our constructive approach and in the local search approach.

We construct a conflict graph by detecting pairs of variables which cannot be activated at the same time. Since the construction of a full conflict graph may require the execution expensive techniques such as probing [17], we opted for a simpler procedure: conflicts are detect by processing each constraint individually, checking for pairs of variables in this constraint whose activation cannot occur at the same time without violating it.

To illustrate relationships between variables and conflict graph consider the binary program $\mathcal{P}$:

$$\begin{aligned}
&min.: \\
&\quad 10x_1 + 12x_2 + 4x_3 + 7x_4 + 5x_5 \\
&s.t.: \\
&\quad x_1 + x_2 \leq 1 && (4) \\
&\quad x_1 + x_3 + x_5 = 1 && (5) \\
&\quad x_2 + x_4 \geq 1 && (6) \\
&\quad x_2 + x_4 + x_5 \leq 1 && (7) \\
&\quad x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}
\end{aligned}$$

In $\mathcal{P}$, Set Partitioning (5) and Set Packing constraints (4 and 7) are the constraints which provide obvious sets of conflicting variables. More generally, Generalized Upper Bounds (GUB) constraints are rich sources of conflicts. The conflict graph for $\mathcal{P}$ can be seen in Figure 3.1. Connected nodes represent conflicting variables. Besides conflicts this graph also shows a different relationship between variables: dashed lines are drawn in groups of variables where at least one variable must be activated.

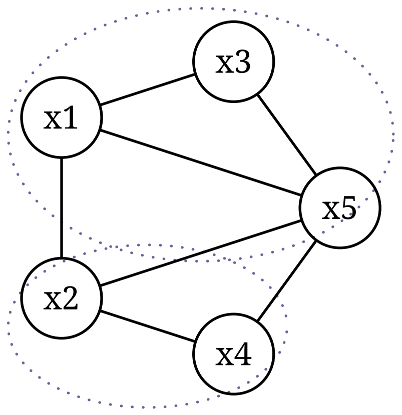

Figure 1. Conflict Graph for $\mathcal{P}$

3.2 Constructive Approach

An initial solution is built by solving a subproblem considering only the structure of set partition, packing and covering constraints. Solving this subproblem corresponds to find an independent set in the conflict graph, i.e. finding a set of variables that have no conflict with each other.

Finding and independent set corresponds to finding a clique in the complimentary graph. Thus, the subproblem is modeled as a graph which is complementary to the original conflict graph. The weight of each vertex is related to the number of Packing and Covering constraints that it satisfies when activated. A Tabu Search based algorithm [18] then searches for cliques with weight above a threshold given as input. In this case, the threshold is the number of set partition and set covering constraints contained in the problem. When the algorithm finishes, a solution is created for the original problem, activating only the variables returned by Tabu Search.

This phase was developed to obtain an initial set of variables that can be activated without generating infeasibilities among them. In the case of instances which contain only these three types of constraints, the result of the constructive phase is a feasible solution for the original problem. Otherwise, the returned solution can be infeasible, so that infeasible constraints are relaxed and sent to be fixed in the local search phase. In our experiments we observed that even when the BP has many other constraints, the initial satisfaction of these three constraint types speeds up a lot the search process, since remaining constraints are quite often easier to satisfy.

3.3 Local Search : Chains of Flips

As the authors of [14] noted, the major obstacle when performing local search with binary programming is to automatically detect relationships between decision variables. Problem specific metaheuristics usually have a compact solution representation where few changes are required to jump from one solution to another. In contrast, Binary Programs are usually modeled a much larger number of decision and related auxiliary variables, so that it is very likely that by flipping one bit at time only unfeasible solutions will be produced.

As an example, consider the BP stated in page 4. One feasible solution is to activate variables x_1 and x_4, with cost 17. Once in this feasible solution, a method whose local search only flips one bit at time would be trapped in a local optimum surrounded by infeasible solutions. A smarter solver, when trying to flip the then inactive variable x_2, for instance, would have to automatically detect that x_1 should be flipped too, to remove the conflict caused by constraint 4, and that another variable should now flipped to satisfy constraint 5, say x_3, which would be infeasible. Thus, when trying to flip the then inactive variable x_2 the solver would detect a *chain* of movements: $x_2 \rightarrow x1 \rightarrow x3$, where every subsequent move would fix an infeasibility caused by a previous move. This specific chain would produce a better solution with cost 16.

A fast algorithm to search for these chains of movements is the key component to every local search based method for BP. In [14], even though authors comment about the importance of this step, no details are given about how these chains are generated, probably because the referred paper describes a commercial product.

Algorithm 1.1 describes our implementation of an algorithm to detect a chain of movements which lead from one feasible solution to another. The algorithm performs a backtracking with limited depth $\overline{d}$ and limited breadth $\overline{f}$. At each recursion a set $\hat{J}$ of variables are flipped: the current variable j and all conflicting variables, if j will become an active variable. These variables are put in a *frozen* state (set S) in this and in deeper levels of the recursion. Subsequent variables to be flipped are chosen from a set $\tilde{J}$ of variables. To fix new infeasibilities, only variables which appear on the constraints set C can help. Candidate variables $\tilde{j}$ are evaluated with respect to how many conflicts it decreases in constraints of C, this evaluation is stored in $e_{\tilde{j}}$. The most promising variables flips are further evaluated recursively in lines 21 to 26 and if the final effect is positive then the recommended chain of movements J^* is augmented. As one can observe, a smart computation of $\tilde{j}_j$ is a key point to the success of the method, since large values of $\overline{d}$ and $\overline{f}$ would result in prohibitive computing times. In this sense, we observed that besides prioritizing variables which decrease the largest amount of infeasibilities we also should include in this evaluation a larger priority to variables which decrease infeasibilities in constraints with less options to resolve these infeasibilities.

The current implementation of local search performs iterated calls to `chainFlip` made from different, randomly selected variables. If the solution is still unfeasible, the search concentrates in variables which appear in constraints which are still not satisfied. In this case, we first randomly select one of the unfeasible constraints and then randomly select one of its variables. Movements are accepted according to a RNA (Random Non Ascendent) rule. As it can be seen in the next section, this simple approach, combined with our constructive algorithm was enough to produce very encouraging results.

Algorithm 1.1: `chainFlip`

```
Input :
    $x$: current solution;
    $j$: variable to be flipped;
    $J$: variables already flipped;
    $C$: constraints to check;
    $S$: frozen variables;
    $d$: current depth;
    $\overline{d}$: maximum depth;
    $\overline{f}$: maximum number of flips per recursive call;
Output:
    $(z^*, J^*)$: cost and variables of the best chain found
1  if $d \geq \overline{d}$ then return ;
2  $\hat{J} = \{j\}$;
3  if $x_j = 0$ then
4      $S \leftarrow S \cup \{j'\} : conflict(j, j')$;
5      $\hat{J} \leftarrow \hat{J} \cup \{j'\} : conflict(j, j') \wedge x_{j'} = 1$;
6  end if
7  $x' = x$;
8  for $j' \in \hat{J}$ do
9      $x^{'}_{j} = 1 - x^{'}_{j}$
10 end for
11 $J \leftarrow J \cup \hat{J}$;
12 $z^* \leftarrow f(x')$;
13 $J^* \leftarrow J$;
14 $C \leftarrow C \cup \{i\} : i$ is a constraint where one or more variables of $\hat{J}$ appear;
15 $\tilde{J} \leftarrow$ all $j$ which appear in some constraint of $C$ and is not in $S$;
16 $e_{\tilde{j}} = 0,\ \forall \tilde{j} \in \tilde{J}$;
17 for $\tilde{j} \in \tilde{J}$ do
18     compute the impact $e_{\tilde{j}}$ of flipping $\tilde{j}$ considering constraints $C$;
19 end for
20 for $k = 1$ to $min(\overline{f}, |\tilde{J}|)$ do
21     $\tilde{j} \leftarrow$ the $k-$th element from $\tilde{J}$ with best $e_{\tilde{j}}$;
22     $(z', J') \leftarrow$ chainFlip$(x', \tilde{j}, J, C, S, d+1, \overline{d}, \overline{f})$;
23     if $z' < z^*$ then
24         $z^* \leftarrow z'$;
25         $J^* \leftarrow J$;
26     end if
27 end for
28 return $(z^*, J^*)$;
```

4 Computational Experiments

Our code was written in C++ using the open source COIN-OR libraries to read instances. The code was compiled on GCC/G++ version 4.6.3. We ran all the experiments on an Intel Core i7-3770 ® 3.4GHz computer with 16Gb of RAM running the openSUSE Linux 12.3 operating system.

Computational experiments were made using all 32 binary problems of MIPLIB 2010 benchmark set [19] which have a feasible solution. Since its introduction in 1992, the MIPLIB became a standard library of tests used to compare the performance of integer programming solvers. It contains a collection of real problems, most of them based on industrial applications. The details of the used problems can be seen in Table 1. Columns Rows and Cols indicate the number of constraints and decision variables of the problems, respectively. Column Objective presents the optimal objective value for each instance. The remaining columns COV, PAC and PAR indicates the number of set covering, set packing and set partition constraints for each instance, respectively.

These experiments compare our approach with two of the best open source integer programming solvers: CBC[1] and GLPK[2]. Table 2 shows the obtained results with execution time limit set to 60 and 300 seconds. In this table, columns GLPK and CBC indicate, respectively, tests performed using GLPK and CBC solvers with default parameters. The Last column, BPLS, corresponds to results obtained by our approach. In both of these columns, a check mark is used to indicate the method has found a feasible solution for a instance in the restricted time limit.

Results show that our approach was able to find feasible solutions to a greater number of instances in a 60 seconds timeout when comparing with CBC and GLPK. Relaxing this time limit to 300 seconds, all methods were able to find more feasible solutions. While our approach found feasible solutions for 21 instances, GLPK and CBC found 19 and 23 feasible solutions, respectively.

5 Conclusions

In this work we proposed and evaluated computationally a hybrid, local search based solver to search for feasible solutions for Binary Programming problems. Computational experiments performed in the MIPLIB 2010 instance set showed that our approach is more reliable to find feasible solutions in very restricted amounts of time than two of the best open source integer programming solvers available: CBC and GLPK.

This feature is fundamental for those interested in the application of Binary Programming where time is a limiting factor. It is also worth to note that the production of the first feasible solution can also speed up the production of high quality solutions: once a feasible solution is available methods like RINS or Local Branching can me immediately applied to improve the incumbent solution.

[1] https://projects.coin-or.org/Cbc

[2] http://www.gnu.org/software/glpk/

Instance	**Rows**	**Cols**	**Objective**	**COV**	**PAC**	**PAR**
acc-tight5	3052	1339	0.00	11	288	244
air04	823	8904	56137.00	0	0	823
bab5	4964	21600	-106412.00	0	88	21
bley_xl1	175620	5831	190.00	14	5133	169
bnatt350	4923	3150	0.00	183	0	0
cov1075	637	120	20.00	252	0	0
eil33.2	32	4516	934.01	0	0	32
eilB101	100	2818	1216.92	0	0	100
ex9	40962	10404	81.00	0	0	162
iis-100-0-cov	3831	100	29.00	3831	0	0
iis-bupa-cov	4803	345	36.00	4803	0	0
iis-pima-cov	7201	768	33.00	7201	0	0
m100n500k4r1	100	500	-25.00	0	100	0
macrophage	3164	2260	374.00	609	0	0
mine-166-5	8429	830	-5.66E+08	0	0	0
mine-90-10	6270	900	-7.84E+08	0	0	0
mspp16	561657	29280	363.00	15	1695	31
n3div36	4484	22120	130800.00	2	4424	0
n3seq24	6044	119856	52200.00	120	4484	0
neos-1109824	28979	1520	378.00	0	1520	23
neos-1337307	5687	2840	-202319.00	0	0	126
neos18	11402	3312	16.00	2809	0	2262
neos-849702	1041	1737	0.00	0	540	270
netdiversion	119589	129180	242.00	103	49799	1
ns1688347	4191	2685	27.00	0	382	88
opm2-z7-s2	31798	2023	-10280.00	0	0	0
reblock67	2523	670	-3.46E+07	0	0	0
rmine6	7078	1096	-457.19	0	0	0
sp98ic	825	10894	4.49E+08	6	627	0
tanglegram1	68342	34759	5182.00	7843	0	0
tanglegram2	8980	4714	443.00	2160	0	0
vpphard	47280	51471	5.00	0	0	320

Table 1. Details of instances.

Instance	**60 seconds**			**300 seconds**		
	GLPK	**CBC**	**BPLS**	**GLPK**	**CBC**	**BPLS**
acc-tight5						
air04	✓	✓	✓	✓	✓	✓
bab5					✓	
bley_xl1						
bnatt350						
cov1075	✓	✓	✓	✓	✓	✓
eil33-2	✓	✓	✓	✓	✓	✓
eilB101	✓	✓	✓	✓	✓	✓
ex9						
iis-100-0-cov	✓	✓	✓	✓	✓	✓
iis-bupa-cov	✓	✓	✓	✓	✓	✓
iis-pima-cov	✓	✓	✓	✓	✓	✓
m100n500k4r1	✓	✓	✓	✓	✓	✓
macrophage	✓	✓	✓	✓	✓	✓
mine-166-5	✓	✓	✓	✓	✓	✓
mine-90-10			✓		✓	✓
mspp16						
n3div36		✓	✓	✓	✓	✓
n3seq24			✓		✓	✓
neos-1109824	✓	✓		✓	✓	✓
neos-1337307	✓	✓		✓	✓	
neos18	✓	✓	✓	✓	✓	✓
neos-849702						
netdiversion						
ns1688347						
opm2-z7-s2	✓	✓	✓	✓	✓	✓
reblock67		✓	✓	✓	✓	✓
rmine6	✓	✓	✓	✓	✓	✓
sp98ic	✓	✓	✓	✓	✓	✓
tanglegram1			✓			✓
tanglegram2	✓	✓	✓	✓	✓	✓
vpphard					✓	
Total	**17**	**19**	**20**	**19**	**23**	**21**

Table 2. Production of feasible solutions in 60 and 300 seconds.

References

1. G. Dantzig, R.F., Johnson, S.: Solution of a Large-Scale Traveling-Salesman Problem. Journal of the Operations Research Society of America **2**(4) (1954) 393–410
2. A. Alan B. Pritsker, L.J.W., Wolfe, P.M.: Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach. Management Science **16**(1) (1969) 93–108
3. Lancia, G.: Integer programming models for computational biology problems. Journal of Computer Science and Technology **19**(1) (2004) 60–77
4. Johnson, E., Nemhauser, G., Savelsbergh, W.: Progress in Linear Programming-Based Algorithms for Integer Programming: An Exposition. INFORMS Journal on Computing **12** (2000)
5. Linderoth, J.T., Ralphs, T.K.: Noncommercial Software for Mixed-Integer Linear Programming. In Karlof, J.K., ed.: Integer Programming: Theory and Practice. CRC Press Operations Research Series (2005) 253–303
6. Doig, A.H.L., G., A.: An Automatic Method of Solving Discrete Programming Problems. Econometrica **28**(3) (1960) 497–520
7. Mittelman, H.: Feasibility benchmark. `http://plato.asu.edu/ftp/feas_bench.html` (February 2014)
8. Lougee-Heimer, R.: The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. IBM Journal of Research and Development **47**(1) (2003) 57–66
9. Forrest, J., Lougee-Heimer, R.: CBC User Guide. INFORMS Tutorials in Operations Research. (2005) 257–277
10. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Mathematical Programming **104** (2005) 2005
11. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming **98**(1-3) (2003) 23–47
12. Danna, E., Rothberg, E., Pape, C.L.: Exploring relaxation induced neighborhoods to improve mip solutions. Mathematical Programming **102**(1) (2005) 71–90
13. Huy, P.N.A., San, C.T.B., Triantaphyllou, E.: Solving integer programming problems using genetic algorithms. In: ICEIC: International Conference on Electronics, Informations and Commumications. (2004) 400–404
14. Benoist, T., Estellon, B., Gardi, F., Megel, R., Nouioua, K.: Localsolver 1.x: a black-box local-search solver for 0-1 programming. 4OR **9**(3) (2011) 299–316
15. Vassilev, V., Genova, K.: An algorithm of internal feasible directions for linear integer programming. European Journal of Operational Research **52**(2) (1991) 203 – 214
16. Atamtürk, A., Nemhauser, G.L., Savelsbergh, M.W.P.: Conflict graphs in solving integer programming problems. European Journal of Operational Research **121** (2000) 40–55
17. Borndorfer, R.: Aspects of Set Packing, Partitioning, and Covering. PhD thesis (1998)
18. Wu, Q., Hao, J.K., Glover, F.: Multi-neighborhood tabu search for the maximum weight clique problem. Annals of Operations Research **196**(1) (2012) 611–634
19. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R., Danna, E., Gamrath, G., Gleixner, A., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D., Wolter, K.: Miplib 2010. Mathematical Programming Computation **3**(2) (2011) 103–163