

Constructive vs Perturbative Local Search for General Integer Linear Programming^{*}

Stefania Verachi and Steven Prestwich

Cork Constraint Computation Centre
Department of Computer Science, University College, Cork, Ireland
{s.verachi,s.prestwich}@cs.ucc.ie

Abstract. Most local search algorithms are “perturbative”, incrementally moving from a search state to a neighbouring state while performing noisy hill-climbing. An alternative form of local search is “constructive”, repeatedly building partial solutions using greedy or other heuristics. Both forms have been combined with constraint propagation, and they can be hybridised with each other by perturbing partial solutions. We design a new hybrid constructive local search algorithm for general (non-binary) integer linear programs, combining techniques from constraint programming, boolean satisfiability, numerical optimisation and scheduling. On a hard design problem it scales better to large instances than both a perturbative algorithm and a Benders decomposition algorithm.

1 Introduction

A currently active area of research is the hybridisation of constraint programming (CP) techniques with those of artificial intelligence (AI) and operations research. By combining their different strengths we may solve problems that are considered otherwise practically unsolvable. One branch of this research aims to combine the space-pruning ability of constraint propagation with the scalability of local search. For example replacing systematic backtracking by a non-systematic form called Incomplete Dynamic Backtracking (IDB) boosts scalability to equal that of local search [19], and in fact is claimed to be local search in a different search space. IDB has been implemented mainly using variations on forward checking (FC). Another example is Decision-Repair [11], designed for constraint satisfaction problems (CSPs) and applied to open shop problems. It uses learning and has heuristics such as clause weighting, a TABU list and greedy hill climbing.

However, there are drawbacks with these hybrids. Like most local search algorithms, they are *perturbative*: they incrementally move from one search state to another, while combining hill-climbing with noise and other heuristics. Unfortunately, incrementally maintaining consistency can be quite complex and require

^{*} This is a revised version of a paper published in the 18th Irish Conference on Artificial Intelligence and Cognitive Science, 2007. The material is based upon works supported by the Science Foundation Ireland under Grant Nos. 04/BR/CS0355 and 00/PL.1/C075.

expensive data structures, especially for higher forms of consistency. An alternative *constructive* form of local search has received relatively little attention [9] but is a promising approach to hybridisation. (For the rest of this paper we shall abbreviate *constructive local search* to *CoLS* and *perturbative local search* to *PeLS*.) CoLS algorithms behave like backtracking algorithms until the first backtrack would normally occur, at which point they restart the search. This repeated construction of partial solutions may be combined with a variety of techniques including greedy algorithms, constraint propagation and relaxation. An advantage of CoLS over PeLS is that it is much easier to hybridise with constraint propagation techniques, because exactly the same implementation techniques can be used. A disadvantage is that frequent restarting incurs significant overheads, but despite this CoLS has been shown to pay off on several problems. For example Squeaky Wheel Optimization (SWO) beat TABU search on a set of scheduling problems, and it beat TABU and three specialised local search algorithms on graph colouring problems [10]; a version of SWO also beat a version of IDB on some generalised graph colouring instances though IDB won on other instances [21]; and UnitWalk [8] beat several perturbative algorithms on industrial benchmarks in SAT solver competitions.

Besides the obvious algorithmic differences, there is another important difference between local search and CP algorithms, leading to a design decision regarding hybrids. With most forms of propagation we must represent entire variable domains in order to keep track of pruned values. In contrast, most local search algorithms do not maintain domains but simply remember the current value of each variable. Their memory requirements are therefore independent of the domain sizes, which is a significant advantage for problems with large domains. In order to combine the power of constraint propagation with the low memory requirements of local search, we use only a restricted form of consistency: *bounds consistency* (BC). This restriction has the advantage that for each integer variable we maintain only an upper and a lower bound denoting the values currently in the domain, making the memory requirement independent of domain size. BC also has the advantage of cheap propagation algorithms [27].

Another design decision is the choice of constraint language, and we choose general (non-binary) Integer Linear Programming (ILP). Many theoretical and real-world combinatorial problems have elegant ILP models, so local search algorithms for ILP immediately have potential applications. ILP also fits very well with the choice of BC: though BC is generally a rather weak form of consistency, in the special case of *linear inequalities* (and for a more general class of *monotonic constraints*) it is equivalent to *generalised arc* (or *hyper-arc*) *consistency* (GAC) [31]. GAC is a strong form of consistency that is much used in CP, so for this restricted class of problems we can design a hybrid local search algorithm that maintains a high level of consistency. Any equality constraints in an ILP can be reduced to inequalities: this weakens the propagation but it is impractical to achieve BC on equalities, which is an NP-hard problem [31].

Thus the proposed algorithm has a combination of features that should make it a useful tool for a range of large structured problems: low memory require-

ments, powerful but cheap constraint propagation, local search scalability, no novel implementation techniques, and a well-known and reasonably expressive modelling language. The paper is structured as follows: Section 2 describes our algorithm, Section 3 reports the results of experiments, Section 4 describes related work, and Section 5 concludes the paper and discusses future work.

2 The Algorithm

Our algorithm, which we call BOLOS (Bounds-Oriented Local Search), is similar in structure to Adaptive Iterated Construction Search [9]. Its original inspiration was the UnitWalk algorithm for SAT [8] but it also takes ideas from other algorithms including SWO [10]. An outline of the core of BOLOS, which finds feasible solutions to a problem π , is shown in Fig. 1 (we discuss solution quality below).

```

procedure BOLOS-feasible( $\pi$ ):
   $s \leftarrow \emptyset$ ,  $w \leftarrow 0$ 
  make  $\pi$  locally consistent
  if inconsistent then return “no solution”
  while not feasible( $\pi$ ,  $s$ )
     $s \leftarrow \text{construct}(\pi, w, s)$ 
     $s \leftarrow \text{perturb}(\pi, s)$ 
     $w \leftarrow \text{prioritise}(\pi, s, w)$ 
  return  $s$ 

```

Fig. 1. BOLOS core for finding feasible solutions

The initial partial solution s is empty. With each problem variable is associated a weight, and the weights are initialised to zero. Constraint propagation is applied before search begins to enforce local consistency, in this case Bounds Consistency (BC). If BC establishes unsolvability then the algorithm terminates. Each iteration has three phases: firstly a partial candidate solution s is constructed, guided by the weights and the previous partial solution; secondly perturbative local search is applied to improve s ; thirdly the weights w are adjusted. Termination occurs when all constraints are satisfied. Section 2.1 describes partial solution construction, Section 2.2 perturbative local search, Section 2.3 variable prioritisation by weight adjustment, and Section 2.4 the extension to optimisation problems.

2.1 Construction

A partial solution is constructed by selecting a variable, assigning a value to it, propagating the result to the other variable domains, and repeating until either

no unassigned variables remain or *domain wipeout* occurs (constraint propagation reduces the domain of at least one variable to the empty set, implying that the partial solution cannot be extended to a full solution). If domain wipeout occurs then the variable whose assignment led to the wipeout, and the variables whose domains were wiped out, are labelled as *troublemakers*, and this information is used during prioritisation (see Section 2.3). Apart from the troublemaker heuristic, this is identical to the behaviour of a standard constraint solver prior to its first backtrack.

Variables are selected first by smallest domain size (a common CP heuristic), ties are broken by *prioritisation* using the dynamic weights (see Section 2.3), and further ties are broken randomly. The value selected for each variable is, where possible, the same value used in the previous iteration after perturbation (random values are assumed before the first iteration). If this value is not currently in the variable domain then another value is chosen as follows: if the current lower bound is greater than the previous value then the lower bound is used; otherwise, if the current upper bound is smaller than the previous value then the upper bound is used.

The form of constraint propagation used is BC, which can be applied to many types of constraint problem. BC has low computational overhead, since each domain is represented only by its lower and upper bound. There is more than one BC algorithm even for the case of linear inequalities on integer variables [7, 31], and it is also possible to perform a weaker form of consistency called Bounds Propagation (BP) which is a form of forward checking using bounds reasoning. The difference in the implementation between BC and BP is that BC requires a queue for arc revision, to maintain the violated constraints that must be revisited whenever a bounds update will be made. BP only propagates to future (currently unassigned) variables and does not require a queue, so it sacrifices some propagation but reduces runtime overhead. It is known that stronger forms of consistency pay off in some cases [25] but not others [12], and we have found that BC is sometimes worthwhile and sometimes not; in this paper we actually use BP.

2.2 Perturbation

To improve the quality of a partial solution we apply a simple perturbative local search algorithm to try to reduce the number of constraint violations. Perturbative local moves are usually cheap compared to constructive iterations so this adds relatively little runtime overhead. We also execute the perturbation phase once at the start of the algorithm, to provide a reasonable initial state.

In local search for CP and SAT, in which we try to remove violations, perturbative moves are usually applied to complete variable assignments. In BOLOS we only have partial assignments, but we overcome this problem by constructing a total assignment consisting of the current assignments plus the most recent assignments of the unassigned variables. The perturbation works by randomly selecting a constraint that is violated under this total assignment, randomly selecting a variable in the constraint, and reassigning that variable so that the

constraint is just satisfied. If no domain value satisfies the constraint then either the upper or lower bound (whichever is best) is used to reduce the degree of violation. The resulting values are used as the default values in the next iteration. This Gauss-Seidel-like technique was taken from numerical optimisation (see standard textbooks in this area, for example [17]). A design decision concerns how long to run the perturbation phase. We apply it in a limited way, visiting each constraint at most once, and ignoring any new violations generated during perturbation.

We also apply a form of *noise* by randomising the assignment of a randomly-chosen variable — in this paper we do this every 10 iterations (UnitWalk only does this on detecting that no variable was reassigned during an iteration). We also apply an extreme form of perturbation: *random restarts*. New random values for all variables are generated after k iterations, where k is a runtime parameter tuned by the user for each problem. In this paper we set $k = 2000$.

2.3 Prioritisation

As mentioned in Section 2.1 some variables are labelled as “troublemakers” during the construction phase, and we hope that by focusing attention on these variables we can more quickly move toward a solution. This kind of dynamic prioritization is used in several AI algorithms, including some scheduling algorithms [5] and the VSIDS branching rule in some complete SAT algorithms [15]. Each variable has an associated weight that changes dynamically during search. Large weights denote more troublesome variables, and during construction these variables are preferred.

We use a simple scheme to update the weights: they are all initially 0, and on defining a new set of troublemakers their weights are assigned to n (the number of variables in the problem), and all other weights are decremented by 1 (unless they are already 0). However, for the problem described in this paper, we found that a variant in which the weights are not decremented worked better. This has the effect of prioritising variables during the first few iterations, but not later when all variables have been assigned top priority. For most of the search the variables are therefore ordered using only the smallest-domain branching rule.

2.4 Optimisation

If (as is usual) the ILP has a linear objective function then BOLOS as described above must be extended. Assuming without loss of generality that the function is to be minimised, we use a simple approach: start with a very high (or infinite) upper bound on the objective, expressed as another linear constraint; solve the feasibility problem using BOLOS; then tighten the constraint and repeat, until reaching either a known optimum cost, or timing out, or the initial application of BC detects infeasibility. Variable assignments and priorities are retained between iterations so that the search for a new solution starts in the neighbourhood of the last solution, in a similar way to [2]. We also use a form of intensification

described in [2, 9]: every f iterations (for some user-defined integer f) we return to the last feasible solution found. In this paper we set $f = 130$.

3 Application to Stochastic Template Design

The original template design problem was first described by Proll & Smith [23] who observed it at a local colour printing firm producing a variety of products from thin board, including cartons for human and animal food and magazine inserts. The problem is described as follows. Given a set of variations of a design, with a common shape and size and such that the number of required *pressings* of each variation is known. The problem is to design a set of templates, with a common capacity to which each must be filled, by assigning one or more instances of a variation to each template. A design should be chosen that minimises the total number of *runs* of the templates required to satisfy the number of pressings required for each variation. As an example, the variations might be for cartons for different flavours of cat food, such as fish or chicken, where ten thousand fish cartons and twenty thousand chicken cartons need to be printed. The problem would then be to design a set of templates by assigning a number of fish and/or chicken designs to each template such that a minimal number of runs of the templates is required to print all thirty thousand cartons. Proll & Smith [23] address this problem by fixing the number of templates and minimising the total number of pressings.

The problem was extended to demand uncertainty via scenarios by Tarim & Miguel [28]. They used a probabilistic model in which demands are random variables, and added *scrap cost* and *shortage cost*. They proposed a certainty-equivalent, non-linear model for this generalised problem, in which they minimise the expected total cost. They solved the problem using a Benders decomposition algorithm (with single cuts) implemented in CPLEX that was shown to be superior to two other versions (with complete and multiple cuts) and to a stochastic constraint programming method. Prestwich, Tarim & Hnich [22] later linearised this model and solved it using a local search algorithm (VWILP) for ILP models, obtaining better results as the allowed number of templates increased.

We applied BOLOS to the same benchmark set as [22, 28] using the same ILP model as [22] (details omitted for space reasons) and under the same experimental conditions: for each problem instance the algorithm was run once with a cutoff time of 1 hour, and the cost of the best solution found in that time recorded along with the actual time taken to find it. The results from [22] used a 2 GHz Intel Centrino, 1 GB RAM machine, and we normalised our runtimes to that machine by comparing runtimes for our algorithm on that machine and ours (a 2.8 GHz Pentium (R) 4 with 512 RAM), which was almost exactly 3 times faster. We compare the complete Benders algorithm with VWILP and BOLOS. The results are shown in Tables 1 and 2 for two, three and four templates. Lowest costs in each case are shown in **bold**.

Clear patterns emerge over the 60 instances: for 2 templates the winner is Benders, for 3 templates VWILP, and for 4 templates BOLOS. As noted in

[22], as the number of templates increases Benders finds poorer solutions within the cutoff time, though the optimal solution can only get better and the local search algorithms do find better (optimal or suboptimal) solutions. Comparing VWILP and BOLOS only: with 2 templates VWILP wins in 15 cases and BOLOS in 1 case; with 3 templates VWILP wins in 10 cases and BOLOS in 7 cases; with 4 templates VWILP wins in 5 cases and BOLOS in 14 cases. In summary, despite BOLOS’s greater runtime overhead than VWILP, it scales better to more templates, and beats 5 other known algorithms applied to these benchmarks. We do not yet know why this is, but we conjecture that problems with more templates have a more rugged search space with deep local optima, requiring frequent restarts as in CoLS.

no	Benders		VWILP		BOLOS		no	Benders		VWILP		BOLOS	
	cost	time	cost	time	cost	time		cost	time	cost	time	cost	time
1	285.00	10	285.00	230	285.00	25	1	295.50	68	285.00	160	285.50	1914
2	285.00	8	285.00	990	285.00	1107	2	305.50	110	285.00	440	285.50	3150
3	480.00	62	480.00	4	562.50	303	3	309.00	3600	307.50	2100	307.50	681
4	332.50	41	332.50	6	412.50	195	4	462.00	110	310.00	99	332.50	423
5	322.50	190	322.50	196	332.50	513	5	465.00	440	310.00	690	309.50	2406
6	401.00	3400	365.00	140	365.00	114	6	481.00	180	365.00	310	365.00	378
7	308.00	670	315.00	570	327.50	180	7	805.00	1100	307.50	1600	314.00	1683
8	310.00	820	312.50	170	317.50	102	8	464.50	1100	312.50	230	305.50	2151
9	308.00	1800	310.00	1500	319.00	3582	9	471.00	1600	312.50	390	306.00	1578
10	326.00	2900	310.00	1680	316.00	216	10	775.00	1500	310.00	3100	307.50	1203
11	333.00	480	339.00	3500	342.00	1998	11	770.00	620	333.00	87	342.00	3042
12	354.00	1100	362.00	47	362.00	705	12	747.25	630	361.75	3300	362.25	348
13	374.00	190	379.50	2	384.50	696	13	557.00	290	372.00	2200	370.75	549
14	393.50	1300	396.00	420	399.50	783	14	522.25	420	393.50	1300	393.50	540
15	407.00	800	414.50	3000	423.75	795	15	531.25	410	407.00	570	408.00	3282
16	432.50	3600	433.25	44	474.00	1662	16	544.25	820	428.25	2900	447.00	498
17	398.25	540	400.75	220	435.75	267	17	526.50	300	399.50	2200	400.75	2283
18	377.75	290	380.75	740	392.50	390	18	512.75	220	375.75	3000	378.75	1737
19	396.75	1200	414.25	1700	413.25	2484	19	530.75	620	399.25	880	398.00	2868
20	406.75	400	409.25	110	425.50	2682	20	547.25	660	408.00	2900	407.75	1197

Table 1. Results with two (left) and three (right) templates

4 Related Work

A simple form of CoLS is Iterative Sampling [13] which simply restarts a back-track-style algorithm at every dead end. Allowing a number of backtracks is known as *random restarting* [6]. This can be made complete and works well on many problems, but does not scale in a similar way to local search. UnitWalk [8] is a CoLS algorithm for SAT that uses unit propagation. It starts in the same

no	Benders cost time		VWILP cost time		BOLOS cost time	
1	295.50	860	285.00	470	285.50	279
2	545.00	82	286.00	3100	292.00	3162
3	2128.00	2100	310.00	35	305.00	2976
4	468.00	1100	307.50	2300	305.00	783
5	3478.00	0.0	310.00	210	309.50	423
6	481.00	93	365.00	430	365.00	1413
7	855.50	650	307.50	950	305.50	1308
8	2563.00	140	312.50	1700	307.50	1458
9	3478.00	0.0	312.50	320	306.50	909
10	3476.00	0.0	310.00	2200	306.00	2358
11	770.00	3300	335.00	1700	333.50	3255
12	1569.50	380	357.00	1900	351.00	819
13	557.00	1500	372.00	1200	370.75	3138
14	522.25	2200	393.50	1100	400.50	612
15	531.25	2200	407.00	2700	406.75	1713
16	2003.75	220	433.25	2400	428.25	786
17	526.50	1500	397.00	3200	402.00	2232
18	512.75	1100	379.50	1500	377.50	2535
19	530.75	3600	399.25	780	403.75	2397
20	1747.50	170	411.75	650	410.50	777

Table 2. Results with four templates

way as a backtracker, selecting a variable, assigning a value to it, propagating where possible, and repeating, until a dead end is reached. It then restarts using a slightly different variable ordering. It has also been hybridised with perturbative local search to improve its performance on some benchmarks. SWO [10] is a CoLS algorithm that operates in two search spaces: a solution space and a prioritisation space. Both searches influence each other: each solution is analysed and used to change the prioritisation, which guides the search strategy used to find the next solution, found by restarting the search. Other examples of CoLS cited in [9] are a stochastic tree search algorithm [3] and Adaptive Probing [24], and it can also be viewed as a special case of Ant Colony Optimisation.

A few (mainly perturbative) local search algorithms for forms of integer program have been reported. General Purpose SIMulated ANnealing (GPSIMAN) [4] is an early algorithm for 0/1 problems using Simulated Annealing (SA). From a feasible assignment a variable is selected and its value flipped (reassigned from 0 to 1 or vice-versa), then SA is used to restore feasibility. This approach was generalised to integer variables in [1]. Pedroso [18] describes an evolutionary algorithm for MIP that searches on the integer variables, then uses linear relaxation to fix the continuous variables. A related approach used a version of the Greedy Randomised Adaptive Search Procedure (GRASP) [16] for MIP, which is the closest algorithm to BOLOS is GRASP. It is a meta-heuristic that alternates constructive phases to find good solutions with local search phases to

find locally optimum solutions. In fact BOLOS could be viewed as a form of GRASP for ILP with some novel features (for example bounds consistency). TABU search has also been applied to MIP, for example see [14]. *Filled function* methods are similar to perturbative local search and have also been used to solve ILPs [26]. Some SAT local search algorithms have been adapted to integer programs. WSAT(PB) [29] is a generalisation of the WalkSAT algorithm that applies to 0/1 problems, and WSAT(OIP) [30] is a further generalisation to integer variables. Saturn [20] is a hybrid of local search and constraint propagation for 0/1 problems, generalised from an earlier SAT algorithm.

5 Conclusion

We presented a hybrid constructive/perturbative local search algorithm for ILP and showed that, on a design problem, it scales up better than the best known complete algorithm and a recently published perturbative algorithm. The algorithm contains a novel combination of techniques from constraint programming, SAT, numerical optimisation and scheduling.

We envisage several directions for future work. Firstly, like most local search algorithms, BOLOS has several runtime parameters and optional heuristics, and judging from experiments so far all its components seem important. In this paper we chose values for a given class of problems after some experimentation, but in future work we hope to freeze the choice of heuristics and eliminate at least some of the parameters. Secondly, we aim to improve BOLOS's heuristics, perhaps by extension to a population-based search such as an evolutionary or ant colony algorithm. Thirdly, we plan to make BOLOS applicable to a wider range of problems by extending its use of bounds consistency to other constraints besides linear inequalities, for example to more general monotonic constraints, or to global constraints such as `alldifferent`. Fourthly, we could extend BOLOS to MIP by applying the Simplex method to the continuous variables after fixing the integer variables, as in [16, 18].

References

1. D. Abramson, M. Randall. A Simulated Annealing Code for General Integer Linear Programs. *Annals of Operations Research* 86:3–24, 1999.
2. J. C. Beck. Solution-Guided, Multi-Point Constructive Search. *Journal of Artificial Intelligence Research*, 2007 (to appear).
3. J. L. Bresina. Heuristic-Biased Stochastic Sampling. *13th National Conference on Artificial Intelligence*, AAAI Press / The MIT Press, 1996, pp. 271–278.
4. D. Connolly. General Purpose Simulated Annealing. *Journal of the Operational Research Society* 43:495–505, 1992.
5. J. M. Crawford. An Approach to Resource Constrained Project Scheduling. *Artificial Intelligence and Manufacturing Research Planning Workshop*, 1996, pp. 35–39.
6. C. P. Gomes, B. Selman, K. McAloon, C. Tret. Randomization in Backtrack Search: Exploiting Heavy-Tailed Profiles for Solving Hard Scheduling Problems. *4th International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, PA, 1998.

7. W. Harvey, J. Schimpf. Bounds Consistency Techniques for Long Linear Constraints. *Workshop on Techniques for Implementing Constraint Programming Systems*, 2002.
8. E. A. Hirsch, A. Kojevnikov. UnitWalk: A New SAT Solver That Uses Local Search Guided by Unit Clause Elimination. *Annals of Mathematics and Artificial Intelligence* 43(1-4):91-111, 2005.
9. H. H. Hoos, T. Stützle. Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco, CA, USA, 2004.
10. D. E. Joslin, D. P. Clements. Squeaky Wheel Optimization. *Journal of Artificial Intelligence Research* 10:353-373, 1999.
11. N. Jussien and O. Lhomme. Local Search With Constraint Propagation and Conflict-Based Heuristics. *Artificial Intelligence* 139(1):21-45, 2002.
12. V. Kumar. Algorithms for Constraint Satisfaction Problems: a Survey. *AI Magazine* 13(1):32-44, 1992.
13. P. Langley. Systematic and Nonsystematic Search Strategies. *1st International Conference on Artificial Intelligence Planning Systems*, 1992.
14. A. Løkketangen, F. Glover. Tabu Search for Zero-One Mixed Integer Programming with Advanced Level Strategies and Learning. *International Journal of Operations and Quantitative Management* 1(2):89-108, 1995.
15. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik. Chaff: Engineering an Efficient SAT Solver. *39th Design Automation Conference*, Las Vegas, 2001.
16. T. Neto, J. P. Pedroso. GRASP for Linear Integer Programming. Metaheuristics: Computer Decision-Making. Kluwer Academic Publishers, 2004, pp. 545-574.
17. J. Nocedal, S. J. Wright. Numerical Optimization. Springer-Verlag, 1999.
18. J. P. Pedroso. An Evolutionary Solver for Linear Integer Programming. BSIS Technical Report 98-7, Riken Brain Science Institute, Wako-shi, Saitama, Japan, 1998.
19. S. D. Prestwich. Local Search and Backtracking vs Non-Systematic Backtracking. *AAAI Fall Symposium on Using Uncertainty within Computation*, Technical Report FS-01-04, AAAI Press, 2001, pp. 109-115.
20. S. D. Prestwich. Incomplete Dynamic Backtracking for Linear Pseudo-Boolean Problems. *Annals of Operations Research* 130:57-73, 2004.
21. S. D. Prestwich. Generalized Graph Colouring by a Hybrid of Local Search and Constraint Programming. *Discrete Applied Mathematics* (to appear).
22. S. D. Prestwich, S. A. Tarim, B. Hnich. Template Design under Demand Uncertainty by Integer Linear Local Search. *International Journal of Production Research* 44(22/15):4915-4928, 2006, Special Issue on Advances in Evolutionary Computation for Design and Manufacturing Problems.
23. L. Proll, B. M. Smith. Integer Linear Programming and Constraint Programming Approaches to a Template Design Problem. *INFORMS Journal of Computing* 10:265-275, 1998.
24. W. Ruml. Incomplete Tree Search Using Adaptive Probing. *17th National Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 2001, pp. 235-241.
25. D. Sabin, G. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. *11th European Conference on Artificial Intelligence*, 1994, pp. 125-129.
26. Y.-L. Shang, L.-S. Zhang. A Filled Function Method for Finding a Global Minimizer on Global Integer Optimization. *Journal of Computational and Applied Mathematics* 181(1):200-210, 2005.
27. C. Schulte, P.J. Stuckey. When do Bounds and Domain Propagation Lead to the Same Search Space. *ACM Transactions on Programming Languages and Systems* 27(3):388-425, 2005.

28. S. A. Tarim, I. Miguel. A Hybrid Benders' Decomposition Method for Solving Stochastic Constraint Programs with Linear Recourse. *Lecture Notes in Artificial Intelligence* 3978:133–148, Springer-Verlag, 2006.
29. J. P. Walser. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. *14th National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference*, AAAI Press / MIT Press, 1997, pp. 269–274.
30. J. P. Walser, R. Iyer, N. Venkatasubramanian. An Integer Local Search Method with Application to Capacitated Production Planning. *15th National Conference on Artificial Intelligence*, AAAI Press, 1998, pp. 373–379.
31. Y. Zhang, R. H. C. Yap. Arc Consistency on n-ary Monotonic and Linear Constraints. *6th International Conference on Principles and Practice of Constraint Programming, Lecture Notes In Computer Science* 1894:470–483, 2000.