A Local Search Algorithm for Balanced Incomplete Block Designs

Steven Prestwich

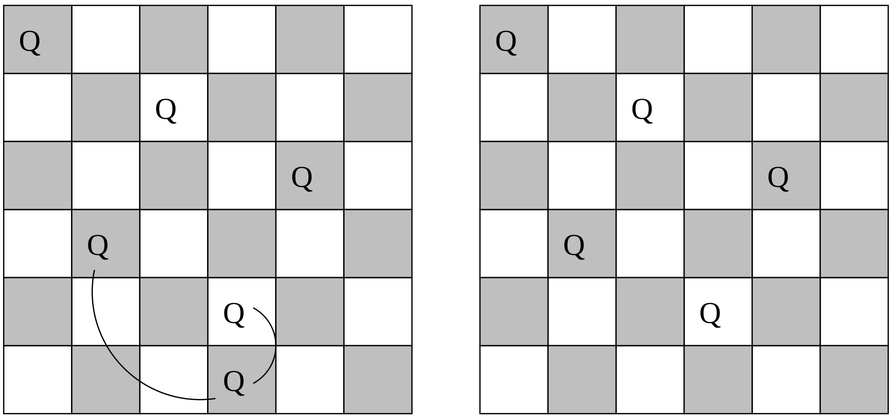
Cork Constraint Computation Centre
Department of Computer Science
University College, Cork, Ireland
s.prestwich@cs.ucc.ie

Abstract. Local search is often able to solve much larger problems than systematic backtracking. The natural way to apply local search to optimisation problems is to explore the legal solutions and minimise the given cost function. This is impractical for problems in which legal solutions are very hard to find, so local search is usually applied to a space of total variable assignments that may violate constraints. Though often successful, this approach is less suited to structured problems with few solutions. An alternative is to explore the consistent partial assignments, while minimising the number of unassigned variables. A local search algorithm of this form violates no constraints and can exploit cost and propagation techniques. This paper describes such an algorithm for balanced incomplete block design generation, which exploits forward checking on some of the constraints and does not violate the others. On a large set of instances it out-performs several backtrackers and a neural network with simulated annealing.

1 Introduction

Local search has proved very successful on many combinatorial problems. It is incomplete and (usually) does not exploit cost and constraint reasoning, but its good scalability sometimes makes it indispensable. The natural way to apply local search to optimisation problems is to explore the legal solutions and minimise the given cost function. This is impractical for constraint satisfaction problems in which legal solutions are hard to find. For such problems local search is therefore usually applied to the space of total assignments, to minimise the number of constraint violations; if this is minimised to zero then a solution has been found. In some algorithms the two functions are combined, for example as a weighted sum, and local search tries to minimise solution cost and infeasibility at the same time.

As an example of a pure constraint satisfaction problem, consider the N-queens problem with a popular model: N variables each taking a value from the integer domain $\{1, \dots, N\}$, each queen corresponding to a row and each value to a column. The constraints for this problem are that no two queens may attack each other (a queen attacks another if they lie on the same row, column or diagonal). The usual local search approach is to explore a space of total assignments, that is with all N queens placed somewhere on the board. Figure 1(i) shows a total assignment containing two constraint violations: the last queen can attack two others and vice-versa (attack is symmetrical). We may try to remove these violations, at the risk of introducing new ones, by repositioning a queen involved in a conflict — that is by reassigning a variable to another value. This is the idea behind most local search algorithms for constraint problems, which has been highly successful for many problems; for example Min-Conflicts Hill Climbing [6] for binary CSPs, GSAT [15] for Boolean satisfiability, and many more recent algorithms.

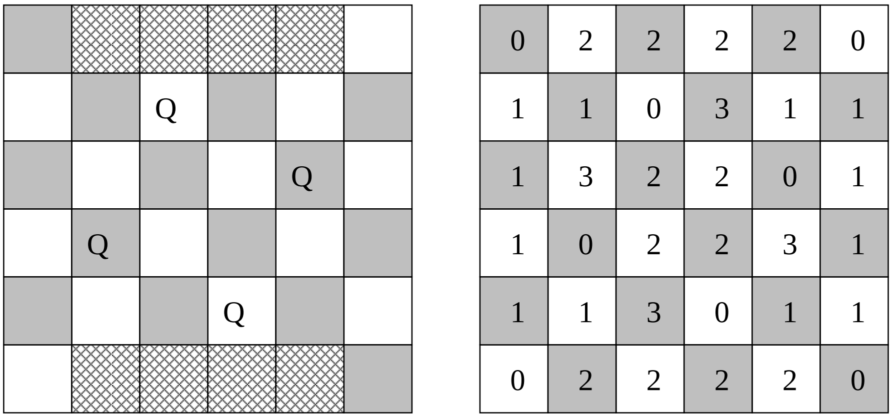


(i) total assignment (ii) consistent partial assignment

Fig. 1. Local search spaces for N-queens

However, a non-computer scientist (or at least a non-constraint programmer) might design a very different form of local search for N-queens: place queens randomly in non-attacked positions; when a queen cannot be placed, randomly remove one or more placed queens; continue until all queens are placed. The states of this algorithm correspond to consistent partial assignments in our model, as shown in Figure 1(ii). We may add and remove queens randomly, or bias the choice using heuristics. This algorithm explores a different space but is still local search. We may view the number of unassigned variables (unplaced queens) as the cost function to be minimised. No queens can be added to the state in Figure 1(ii), which is therefore a local minimum under this function. This approach is taken manually by dispatch schedulers [4]. Another use is in the IMPASSE class of graph colouring algorithms [7], where a consistent partial assignment is called a *coloration neighborhood*.

An advantage of this type of local search is that constraints are never violated, avoiding the need for weighted sums of cost and infeasibility functions: local



(i) FC-consistent partial assignment (ii) corresponding conflict counts

Fig. 2. Forward checking in local search

search can concentrate its efforts on feasibility. For example in [8] local search was performed on partial assignments consistent under cost constraints obtained from a relaxation of an optimisation problem. Upper and lower cost bounds were maintained in exactly the same way as in branch-and-bound but scalability was improved. Furthermore, it found optimal solutions while local search on the legal solutions (minimising the given cost function) failed to do so.

Another advantage is that this form of local search can be integrated with (at least some forms of) constraint propagation in the tightest possible way: propagation prunes states from the search space, as in a backtrack-based constraint system. In [10] coloration neighbourhoods were restricted to those consistent under forward checking, with improved results over IMPASSE colouring. Notice that though the state in Figure 1(ii) is consistent, under constraint propagation it is inconsistent: the domain of the variable corresponding to the last row is empty. Local search enhanced with constraint propagation prunes such states. Removing the queen on row 1 makes it FC-consistent (consistent under forward checking) as shown in Figure 2(i); squares on free rows that are under attack by at least one queen are shaded, and both free rows contain empty squares. However, this state cannot be extended while maintaining FC-consistency (placing a queen in column 1 or 6 on row 1 or 6 causes all squares to be under attack on the other row) so it is a local minimum.

To update variable domains while applying local search, we maintain a *conflict count* for each variable-value pair. The conflict counts for the FC-consistent state are shown in Figure 2(ii). A conflict count denotes the number of constraints that would be violated if the corresponding assignment were made. They are computed incrementally on [un]assigning a variable. Notice that the number of attacking queens *on other rows* are counted (each potential attack corresponds to a binary constraint that would be violated), and that the queens are placed on squares with zero conflict counts (corresponding to values that have not been deleted from domains by forward checking). Standard variable ordering heuristics can be used, for example selecting the variable with smallest domain. An additional benefit of conflict counts is that we can obtain a domain size for assigned variables as well as unassigned ones. In the example of Figure 2 the variables for rows 2–5 each have domain size 1, but in general these may be different. This can be used to guide the selection of variables for unassignment, for example the variable with *greatest* domain size. These techniques are explored in [12]. Conflict counts can be generalised to non-binary constraints; they were used for unit propagation in a local search algorithm for SAT in [8], and for similar propagation on linear constraints over 0/1 variables in [11].

Because this approach performs local search in a constrained space, we call it Constrained Local Search (CLS). Alternatively it can be viewed as a non-systematic form of backtracking, which we call Incomplete Dynamic Backtracking. The relationships between CLS and other search algorithms are discussed in previous papers [8, 10–12]. In the next section we describe a CLS implementation for BIBD generation.

2 Local search for BIBD generation

BIBD generation is a standard combinatorial problem, originally used in the statistical design of experiments but since finding other applications such as cryptography. A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Another way of defining a BIBD is in terms of its *incidence matrix*, which is a binary matrix with v rows, b columns, r ones per row, k ones per column, and scalar product λ between any pair of distinct rows. A BIBD is therefore specified by its parameters (v, b, r, k, λ). An example is shown in Figure 3. It can be proved that for a BIBD to exist its parameters must satisfy the conditions $rv = bk$, $\lambda(v-1) = r(k-1)$ and $b \geq v$, but these are not sufficient conditions. Constructive methods can be used to design BIBDs of special forms, but the general case is very challenging and there are surprisingly small open problems, the smallest being (22,33,12,8,4). One source of intractability is the large number of symmetries: given any solution, any two rows or columns may be exchanged to obtain another solution. A survey of known results is given in [2] and some references and instances are given in CSPLib (problem 28). [1]

The most direct CSP model for BIBD generation represents each matrix element by a binary variable $m_{ij} \in \{0, 1\}$. There are three types of constraint: (i) v b-ary constraints for the r ones per row, (ii) b v-ary constraints for the k ones per column, and (iii) $v(v-1)/2$ $2b$-ary constraints for the λ matching ones in each pair of rows. The new algorithm maintains several integer counts to avoid violating these constraints:

[1] http://www.csplib.org

$$\begin{bmatrix} 1\,0\,1\,1\,1\,1\,0\,0\,0\,0\,1 \\ 0\,0\,1\,1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,1\,0\,0\,0\,1\,1\,0 \\ 0\,0\,0\,0\,1\,0\,1\,1\,1\,1 \\ 0\,1\,1\,0\,0\,1\,0\,1\,0\,1 \\ 1\,1\,0\,0\,1\,1\,1\,0\,0\,0 \end{bmatrix}$$

Fig. 3. A solution to the BIBD instance $(6, 10, 5, 3, 2)$

- for each row there are two variables

$$\rho_i^0 = card\{j \in \{1 \dots b\} |\, m_{ij} = 0\} \quad (1 \le i \le v)$$
$$\rho_i^1 = card\{j \in \{1 \dots b\} |\, m_{ij} = 1\} \quad (1 \le i \le v)$$

- similarly for each column there are two variables

$$\chi_j^0 = card\{i \in \{1 \dots v\} |\, m_{ij} = 0\} \quad (1 \le j \le b)$$
$$\chi_j^1 = card\{i \in \{1 \dots v\} |\, m_{ij} = 1\} \quad (1 \le j \le b)$$

- to handle the scalar products there are two variables for each pair of rows

$$\pi_{ii'}^1 = card\{j \in \{1 \dots b\} |\, m_{ij} = 1 \,\wedge\, m_{i'j} = 1\} \quad (1 \le i < i' \le v)$$
$$\pi_{ii'}^0 = card\{j \in \{1 \dots b\} |\, m_{ij} = 0 \,\vee\, m_{i'j} = 0\} \quad (1 \le i < i' \le v)$$

The constraints are expressed on these counts:

$$\begin{array}{lll} \rho_i^1 \le r & \rho_i^0 \le b - r & (1 \le i \le v) \\ \chi_j^1 \le k & \chi_j^0 \le v - k & (1 \le j \le b) \\ \pi_{ii'}^1 \le \lambda & \pi_{ii'}^0 \le b - \lambda & (1 \le i < i' \le v) \end{array}$$

Initially all the m_{ij} are unassigned so all counts are zero. It can easily be verified that a total variable assignment satisfying these constraints is a BIBD with the required properties.

To this model we add forward checking, but only on the row and column constraints (it may be added to the scalar product constraints in future work). For each matrix entry m_{ij} and domain value $b \in \{0, 1\}$ a conflict count κ_{ijb} is maintained, which counts the number of constraints that would be violated under the assignment $m_{ij} = b$.

All the counts are updated incrementally. For example on assigning $m_{ij} = 0$, ρ_i^0 and χ_j^0 are incremented. If $\rho_i^0 = b - r$ then $\kappa_{ij'0}$ is incremented for all $m_{ij'}$ (where $j \ne j'$) not currently assigned to 0; and if $\chi_i^0 = v - k$ then $\kappa_{i'j0}$ is incremented for all $m_{i'j}$ (where $i \ne i'$) not currently assigned to 0. On incrementing a conflict count from 0 to 1, the corresponding value is deleted from its domain, and its domain size updated. Counts are similarly incremented on assigning $m_{ij} = 1$ and decremented on unassignment. Note that if incrementing any count would violate a row or column constraint, or if a domain size becomes zero, then the assignment is disallowed and its effects are removed.

By maintaining this information during search we can unassign any assigned variable at any point (or assign any unassigned variable, subject to constraints), allowing flexible search strategies. It remains only to choose heuristics to guide the search, and we use heuristics similar to those in previous papers on CLS. Variables are selected for smallest domain size, breaking ties randomly. If the selected variable can be assigned a value then assign it, otherwise backtrack. To backtrack we unassign one or more assigned variables, in fact $B \geq 1$ variables where B is an integer *noise parameter* specified by the user at runtime. For unassignment the reverse heuristic is used: select variables for greatest domain size, breaking ties randomly. Values are chosen with a preference for the last value used (initialised randomly), but to avoid stagnation a small random factor is added.

In the next section we evaluate this algorithm on a large number of BIBD instances, and compare its performance with published results.

3 Results

We compare results for CLS with those available for several other algorithms. From [5] we obtain results for (i) forward checking with conflict-directed backjumping and variable ordering based on domain size and degree (FC-CBJ-DG) [13]; (ii) the same algorithm enhanced with heuristics for exploiting problem symmetries (FC-CBJ-SVP-VM). Both were executed on a Sun Ultra 60 with 360MHz. From [1] we obtain results for a neural network with simulated annealing (NN-SA). It was executed on a Connection Machine CM-200 with 2048 processors, and because of the parallel platform no execution times were given. CLS was executed on a 300 MHz DEC Alphaserver 1000A 5/300 under Unix.

First we consider the solvable instances with $vb < 1400$ and $k = 3$. Results are shown in Figures 4 and 5, where mean CPU times are given in milliseconds, the mean number of search tree nodes is shown for the backtrackers, and the mean number of backtracks shown for CLS. All algorithms except NN-SA were executed 50 times per instance and column *solns* shows in how many runs a solution was found in the time limit (hence the long execution times in cases where few or no solutions were found). CLS is the most successful algorithm, solving all instances (unlike the neural network and the non-symmetry-breaking backtracker), and taking a fraction of the time of the enhanced backtracker (assuming similar speeds for the two machines).

Next we consider the solvable instances with $vb < 1000$ and $k \neq 3$. Results are shown for CLS and NN-SA in Figures 6, 7 and 8. These problems are considerably harder, so CLS was executed three times per instance with noise levels 1, 2 and 3. The best result is reported in each case, "no" denoting that no solution was found after 2,000,000 backtracks. All instances solved by NN-SA were also solved by CLS. NN-SA solves 18/86 instances while CLS solves 54/86. Including the

39 instances with $k = 3$ and $vb < 1000$ from the earlier tables, NN-SA solves 54/125 instances (43.2%) while CLS solves 93/125 (74.4%).

In a previous paper [11] the experiments on the easier instances ($vb < 1400$, $k = 3$) were performed using a version of CLS for 0/1 integer programs. A 0/1 model of BIBD generation was used to encode the problems that introduced an auxiliary variable for each pair of matrix rows. This algorithm performed similarly to the enhanced backtracker, and considerably more slowly than the specialised implementation described here. Several other recent papers mention with BIBD generation. In [9] five instances are considered: (7,7,3,3,1), (6,10,5,3,2), (7,14,6,3,2), (9,12,4,3,1) and (8,14,7,4,3). These are SAT-encoded and passed to three SAT algorithms using backtracking, local search and a hybrid (a SAT implementation of CLS). Computational results were unexpectedly poor, the fifth instance being unsolved after several hundred seconds by any SAT algorithm. In [14] the same five instances are considered, with better results. Using six different encodings in the Choco constraint system, all are solved in 10–3670 ms. In [3] a matrix model is used to break symmetries, and (8,14,7,4,3) is solved in 238 seconds.

4 Conclusion

This paper described and tested a Constrained Local Search (CLS) algorithm for the problem of BIBD generation. Previous papers have argued that this form of local search is well-suited to large, structured combinatorial problems with few solutions. It combines the scalability of local search with the space-pruning ability (but not the completeness) of constraint programming. The BIBD results support this view: CLS outperformed several backtrack-based approaches, and solved more instances than another local search algorithm (a neural network with simulated annealing).

This work can be developed in at least two ways. Forward checking can be applied to the scalar product constraints, which may enable the algorithm to solve more instances (ideally one of those whose solvability is unsettled). This has not been done yet, mainly because of lack of time (the implementation was already becoming quite complex). Alternatively the CLS implementation for 0/1 integer programs [11] can be extended to handle non-linear constraints, so that no auxiliary variables are necessary; this should enable it to match or exceed the performance of the specialised BIBD algorithm, as well as making it applicable to further combinatorial problems. However, both would need $O(v^2 b)$ memory whereas this algorithm needs only $O(vb)$, making it more suitable for very large instances.

The addition of symmetry breaking constraints should be tried next. In a previous paper [9] it was found to be counter-productive when combined with local search, on three problems including BIBD generation (but using a different model). There was insufficient time to modify the algorithm described in this paper, but simple symmetry-breaking unary constraints were added to a 0/1 ILP model to fix the values of the first matrix row. On instance (8,14,7,4,3) these constraints halved the mean execution time of the CLS algorithm for 0/1 programs, showing that symmetry breaking *can* improve stochastic search. Perhaps unary constraints are an exception; this issue requires further investigation.

Acknowledgement The Cork Constraint Computation Centre is supported by Science Foundation Ireland.

References

1. P. Bofill, C. Torras. Neural Cost Functions and Search Strategies for the Generation of Block Designs: an Experimental Evaluation. *International Journal of Neural Systems* vol. 11 no. 2, World Scientific Publishing Company, 2001, pp. 187–202.
2. C. J. Colbourn, J. H. Dinitz (eds.). The CRC Handbook of Combinatorial Designs. CRC Press, 1996.
3. P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Matrix Modelling. *CP'01 Workshop on Modelling and Problem Formulation.*
4. M. Y. Loenko. A Non-Return Search Algorithm. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems* pp. 251–259.
5. P. Meseguer, C. Torras. Exploiting Symmetries Within Constraint Satisfaction Search. *Artificial Intelligence* vol. 129 no. 1–2, 2001, pp. 133–163.
6. S. Minton, M. D. Johnston, A. B. Philips, P. Laird. Minimizing Conflicts: A Heuristic Repair Method For Constraint Satisfaction and Scheduling Problems. *Constraint-Based Reasoning*, Freuder & Mackworth (Eds.), 1994.
7. C. Morgenstern. Distributed Coloration Neighborhood Search. D. S. Johnson and M. A. Trick (eds.), *Cliques, coloring and satisfiability: second DIMACS implementation challenge, DIMACS series in discrete mathematics and theoretical computer science* vol. 26, American Mathematical Society 1996, pp. 335–357.
8. S. Prestwich. A Hybrid Search Architecture Applied to Hard Random 3-SAT and Low-Autocorrelation Binary Sequences. *The Sixth International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science vol. 1894, Springer-Verlag 2000, pp. 337-352.
9. S. Prestwich. First-Solution Search with Symmetry Breaking and Implied Constraints. *CP'01 Workshop on Modelling and Problem Formulation.*
10. S. Prestwich. Coloration Neighbourhood Search With Forward Checking. *Annals of Mathematics and Artificial Intelligence* vol. 34 no. 4, 2002, Kluwer Academic Publishers, pp. 327–340.
11. S. Prestwich. Randomised Backtracking for Linear Pseudo-Boolean Constraint Problems. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, 2002, pp. 7–19.
12. S. Prestwich. Combining the Scalability of Local Search with the Pruning Techniques of Systematic Search. *Annals of Operations Research*, special issue (to appear).
13. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* vol. 9 no. 3, 1993, pp. 268–299.
14. P. Prosser, E. Selensky. On the Encoding of Constraint Satisfaction Problems with 0/1 Variables. *CP'01 Workshop on Modelling and Problem Formulation.*

15. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, MIT Press 1992, pp. 440–446.

		FC-CBJ-DG			FC-CBJ-SVP-VM		NN-SA	CLS	
parameters	vb	solns	nodes	ms	nodes	ms	solved	back	ms
7 7 3 3 1	49	50	21	1.4	21	4	yes	61	0
6 10 5 3 2	60	50	60	3.6	30	6	yes	141	1
7 14 6 3 2	98	50	2152	130	44	12	yes	293	3
9 12 4 3 1	108	50	40	1.8	48	10	yes	386	7
6 20 10 3 4	120	18	435	3700	62	35	yes	284	3
7 21 9 3 3	147	16	2877	4300	69	44	yes	364	6
6 30 15 3 6	180	6	196	9900	95	140	yes	430	9
7 28 12 3 4	196	11	195	7600	86	120	yes	502	13
9 24 8 3 2	216	44	763	1200	77	84	yes	536	23
6 40 20 3 8	240	3	156	1700	126	390	yes	457	214
7 35 15 3 5	245	6	230	1500	109	270	yes	444	18
7 42 18 3 6	294	6	141	1900	133	480	yes	619	32
10 30 9 3 2	300	38	181	3400	123	200	yes	1518	55
6 50 25 3 10	300	1	1057	31000	156	830	yes	517	32
9 36 12 3 3	324	29	478	6800	173	380	yes	1623	47
13 26 6 3 1	338	50	1076	350	145	170	yes	4548	154
7 49 21 3 7	343	2	151	33000	163	790	yes	578	38
6 60 30 3 12	360	2	139	46000	185	1500	yes	487	29
7 56 24 3 8	392	1	36401	46000	173	1200	yes	1604	50
6 70 35 3 14	420	0	0	54000	217	2300	yes	1602	56
9 48 16 3 4	432	19	685	16000	152	730	yes	1631	81
7 63 27 3 9	441	0	0	6000	193	1700	yes	1606	63
8 56 21 3 6	448	5	285	37000	323	2000	yes	1645	80
6 80 40 3 16	480	0	0	72000	246	3600	yes	1569	69

Fig. 4. Results with $vb < 1400$ and $k = 3$ (1 of 2)

parameters	vb	FC-CBJ-DG			FC-CBJ-SVP-VM		NN-SA	CLS	
		solns	nodes	ms	nodes	ms	solved	back	ms
7 70 30 3 10	490	1	235	67000	216	2400	no	1621	81
15 35 7 3 1	525	48	395	980	193	390	yes	12586	548
12 44 11 3 2	528	41	591	5100	204	670	yes	6639	285
7 77 33 3 11	539	0	0	93000	240	3300	no	2593	106
9 60 20 3 5	540	12	386	29000	238	1600	yes	2603	133
7 84 36 3 12	588	1	1027	92000	254	4200	yes	2585	112
10 60 18 3 4	600	12	613	26000	188	1500	no	4594	204
11 55 15 3 3	605	33	680	12000	229	1300	yes	5559	234
7 91 39 3 13	637	0	0	130000	277	5400	no	3564	165
9 72 24 3 6	648	8	671	42000	309	3000	no	4628	201
13 52 12 3 2	676	43	298	4600	1008	3400	yes	8541	317
9 84 28 3 7	756	8	2054	54000	257	4200	no	4594	229
9 96 32 3 8	864	9	3997	66000	295	6400	no	5549	326
10 90 27 3 6	900	8	3131	56000	286	5300	no	8419	502
9 108 36 3 9	972	3	1193	96000	341	40000	no	8507	501
13 78 18 3 3	1014	37	1392	16000	280	3500		18454	1099
15 70 14 3 2	1050	36	1647	23000	1058	5700		25472	1501
12 88 22 3 4	1056	33	1271	28000	296	5000		15486	945
9 120 40 3 10	1080	6	10429	110000	461	14000		11471	658
19 57 9 3 1	1083	46	778	4800	745	6900		64528	3912
10 120 36 3 8	1200	4	9927	110000	377	13000		13581	896
11 110 30 3 6	1210	24	2491	49000	366	36000		13440	914
16 80 15 3 2	1280	40	2275	23000	490	4700		39486	2812
13 104 24 3 4	1352	30	1076	49000	344	8500		24507	1815

Fig. 5. Results with $vb < 1400$ and $k = 3$ (2 of 2)

		NN-SA	CLS		
parameters	vb	solved	noise	backtracks	sec
8 14 7 4 3	112	yes	2	282	0.0
11 11 5 5 2	121	yes	2	2544	0.04
10 15 6 4 2	150	yes	2	2852	0.04
9 18 8 4 3	162	yes	2	2714	0.05
13 13 4 4 1	169	yes	2	674	0.01
10 18 9 5 4	180	yes	2	5172	0.12
8 28 14 4 6	224	yes	2	2408	0.06
15 15 7 7 3	225	yes	2	30488	0.61
11 22 10 5 4	242	yes	2	47868	1.2
16 16 6 6 2	256	yes	2	25662	0.54
12 22 11 6 5	264	no	2	48516	1.23
10 30 12 4 4	300	yes	2	3632	0.12
16 20 5 4 1	320	yes	3	6771	0.16
9 36 16 4 6	324	yes	2	7316	0.21
8 42 21 4 9	336	no	2	8004	0.25
13 26 8 4 2	338	yes	2	13552	0.5
13 26 12 6 5	338	no	2	283418	7.93
10 36 18 5 8	360	no	2	30432	1.08
19 19 9 9 4	361	no	1	320158	9.73
11 33 15 5 6	363	no	2	51228	1.79
14 26 13 7 6	364	no	—	no	—
16 24 9 6 3	384	no	1	320052	9.8
12 33 11 4 3	396	yes	3	11706	0.33
21 21 5 5 1	441	yes	3	9576	0.22
8 56 28 4 12	448	no	2	3072	0.12
10 45 18 4 6	450	no	2	2314	0.08
15 30 14 7 6	450	no	—	no	—
16 30 15 8 7	480	no	—	no	—
11 44 20 5 8	484	no	2	31076	1.13

Fig. 6. Results with $vb < 1000$ and $k \neq 3$ (1 of 3)

		NN-SA	CLS		
parameters	vb	solved	noise	backtracks	sec
9 54 24 4 9	486	no	3	5739	0.21
13 39 12 4 3	507	no	3	33426	1.22
13 39 15 5 5	507	no	2	257178	11.2
16 32 12 6 4	512	no	—	no	—
15 35 14 6 5	525	no	—	no	—
12 44 22 6 10	528	no	2	647678	27.9
23 23 11 11 5	529	no	—	no	—
10 54 27 5 12	540	no	2	24100	0.98
8 70 35 4 15	560	no	3	3750	0.13
17 34 16 8 7	578	no	—	no	—
10 60 24 4 8	600	no	2	14418	0.75
11 55 20 4 6	605	no	3	17787	0.8
11 55 25 5 10	605	no	2	86634	4.45
18 34 17 9 8	612	no	—	no	—
25 25 9 9 3	625	no	—	no	—
15 42 14 5 4	630	no	1	690050	42.9
21 30 10 7 3	630	no	—	no	—
16 40 10 4 2	640	no	2	89090	4.22
16 40 15 6 5	640	no	—	no	—
9 72 32 4 12	648	no	2	10262	0.54
15 45 21 7 9	675	no	—	no	—
13 52 16 4 4	676	no	3	89193	3.76
13 52 24 6 10	676	no	2	1133166	49.7
10 72 36 5 16	720	no	3	28620	1.33
19 38 18 9 8	722	no	—	no	—
11 66 30 5 12	726	no	2	28148	1.52
22 33 12 8 4	726	no	—	no	—
15 52 26 7 12	780	no	—	no	—

Fig. 7. Results with $vb < 1000$ and $k \neq 3$ (2 of 3)

		NN-SA	CLS		
parameters	vb	solved	noise	backtracks	sec
27 27 13 13 6	729	no	—	no	—
21 35 15 9 6	735	no	—	no	—
10 75 30 4 10	750	no	3	21420	0.92
25 30 6 5 1	750	yes	2	361682	14.3
20 38 19 10 9	760	no	—	no	—
16 48 15 5 4	768	no	1	694455	43.2
16 48 18 6 6	768	no	—	no	—
12 66 22 4 6	792	no	1	22626	1.38
12 66 33 6 15	792	no	1	199475	12.8
9 90 40 4 15	810	no	2	12862	0.69
13 65 20 4 5	845	no	2	23760	1.4
11 77 35 5 14	847	no	1	97508	5.11
21 42 10 5 2	882	no	—	no	—
21 42 12 6 3	882	no	—	no	—
21 42 20 10 9	882	no	—	no	—
16 56 21 6 7	896	no	—	no	—
10 90 36 4 12	900	no	2	20348	0.99
15 60 28 7 12	900	no	—	no	—
18 51 17 6 5	918	no	—	no	—
22 42 21 11 10	924	no	—	no	—
15 63 21 5 6	945	no	1	423463	30.7
16 60 15 4 3	960	no	1	187646	6.31
16 60 30 8 14	960	no	—	no	—
31 31 6 6 1	961	yes	2	121008	2.04
31 31 10 10 3	961	no	—	no	—
31 31 15 15 7	961	no		no	
11 88 40 5 16	968	no	1	61386	4.07
22 44 14 7 4	968	no	—	no	—
25 40 16 10 6	1000	no	—	no	—

Fig. 8. Results with $vb < 1000$ and $k \neq 3$ (3 of 3)