A Constraint Model and a Reduction Operator for the Minimising Open Stacks Problem

Alice Miller and Patrick Prosser and Chris Unsworth
Department of Computing Science
University of Glasgow, Scotland
{alice/pat/chrisu}@dcs.gla.ac.uk

Abstract

We present two constraint models for the Minimising Open Stacks Problem (MOSP). Our first model is based on that reported in [1], and our second is a refinement and is more space efficient. We also present two reduction operators for the MOSP. One reduction operator is applied as a pre-process to remove elements of the problem that are provably redundant, the second dynamically reduces the problem during search. We also introduce conditional lower bounds, which are lower bounds associated with a partial assignment. Experiments are then performed using the two reduction operators, and our best constraint model using the lower bounds reported in [2] and the conditional lower bounds.

1 Introduction

The minimising open stacks problem (MOSP) is in principal very similar to the rehearsal problem, as described in [3]. The MOSP is essentially a permutation problem. We are given m products and n customers. A customer may demand a number of different products. Therefore we can think of a customer as a 0/1 vector (or row) of length m and a product as a 0/1 column, where the column has n elements corresponding to customers. When the first product for a customer is produced, a stack (or a pallet) is opened for that customer, and that stack is closed when we have made the last product for that customer. The products can be made in any order, i.e. we can permute the columns in $m!$ ways. By permuting the columns we can then control when customer orders are open and closed. The goal is then to find a permutation that minimises the number of stacks/pallets open at any time.

Below we present two constraint encodings for the MOSP. The first encoding is based on that in [1] and was implemented in JChoco. Our second encoding is more compact, using less variables and less constraints, and allows us to model larger problems. This model was then encoded in ILOG's JSolver.

We also present a reduction operator. This is a pre-processing step in problem solving, i.e. the problem is processed to produce a smaller representative problem that can then be modelled and solved. The solution to this problem can then be inflated, in linear time, to give a solution to the original problem.

2 A Constraint Programming Encodings

Below we introduce the variables and the constraints (in *italics*). The main aspects of the model are to *1-fill* rows of a 0/1 array, such that for a given row, say i, we locate the position of the first 1 in that row and also the last 1 in that row, and then fill the intermediate elements with 1's. This corresponds to a customer order being open from the first product demanded up to and including the last product demanded.

- M a two dimensional array. M[i][j] = 1 if and only if customer i requires product j. The array M is essentially an array of constants, i.e. M is the data read in initially and does not change.
- S a one dimensional array. If S[j] = k then product j will be produced in time slot (column) k. That is, S gives us the permutation of the columns. Each variable in S has a domain 1 to m, and all the variables in S must take different values.
- P a one dimensional array. If P[k] = j then in time slot (column) k, product j will be produced. In order to force S and P to maintain a permutation we use the channelling constraints $S[j] = k \leftrightarrow P[k] = j$.
- T is the timetable, and is a two dimensional array of 0/1 variables. $S[j] = k \rightarrow T[i][k] = M[i][j]$. That is, if product j is made in time slot k (i.e. S[j] = k) and customer i demands product j (i.e. M[i][j]) then product j is made for customer i in time k (i.e. T[i][k] = M[i][j]).
- open is a two dimensional array of 0/1 variables. If open[i][k] = 1 then something is made for customer i in time k or earlier. Consequently, a stack is open for that customer at time k and if a stack is open for customer i at time k the stack is also open at time k+1. Therefore we have a right-rippling constraint such that $open[i][k] = 1 \rightarrow open[i][k+1] = 1$. This right-ripple is initiated when T[i][k] = 1. We then have the following constraints, $T[i][k] = 1 \rightarrow open[i][k] = 1$.
- nc is also a two dimensional array of 0/1 variables. This array is symmetrical to the array above, stating when a stack is not closed, and has a left-rippling constraint. If nc[i][k] = 1 then something is made for customer i

in time k or earlier, consequently the stack for this customer cannot be closed at time k, and neither can it be closed at time k-1. Therefore we have the left-rippling constraint $nc[i][k] = 1 \rightarrow nc[i][k-1] = 1$. Again, the left-ripple is kicked off when T[i][k] = 1. The constraint is then $T[i][k] = 1 \rightarrow nc[i][k] = 1$.

Stacked is a two dimensional array, such that Stacked[i][k] is 1 if and only if the ith customer's order is stacked at time k and the stack has not been closed at time k. Therefore we have the constraint $Stacked[i][k] = 1 \leftrightarrow open[i][k] = 1 \wedge nc[i][k] = 1$.

soat is a one dimensional array of m variables with domains 0 to n, such that soat[k] is the number of stacks open at time k. Therefore soat[k] is the sum of the variables in the kth column of array Stacked. Therefore we have the constraint $soat[k] = \sum_{i=0}^{i=n-1} Stacked[i][k]$.

cost is the objective variable to be minimised, and is the maximum of the values in the vector soat (stacks open at time). That is, we want to minimise the maximum number of stacks open at any time, consequently we have the constraint $maximum(cost, \{soat[k] : 0 \leq k < m\})$. The maximum constraint works as follows. If the lower bound of some variable soat[k] increases then $lwb(cost) = max(lwb(cost), lwb(soat[k]))$. If $lwb(cost)$ increases then there is no effect. If the upper bound of some variable soat[k] decreases then we find the variable soat[l] that has the largest upper bound and set $upb(cost) = min(upb(cost), upb(soat[l]))$. If $upb(cost)$ decreases then for all values of k ($0 \leq k < m$) $upb(soat[k]) = min(upb(soat[k]), upb(cost))$.

The $maximum$ constraint can also be realised using primitive constraints. Consider the case where we have 3 variables A, B, C and we constrain variable X to be the maximum of A, B, and C. This can be done as follows: $X \geq A \wedge X \geq B \wedge X \geq C \wedge (X = A \vee X = B \vee X = C)$.

The above model was coded in JChoco using the S variables as the decisions. Variables were instantiated in the static order S[0], S[1], S[2], ... , S[m-1]. A symmetry breaking constraint was applied, similar to that in [1], such that S[0] < S[m-1].

A second model was also coded up in JSolver. The P variables were used as decisions, and the arrays open, nc and T were not used (and consequently the ripple constraints in those arrays were not used). To replace the 2-dimensional arrays T, open and nc, we introduced the following one dimensional arrays.

start a one dimensional integer array. If start[i] = k then the first product for customer i is made in time slot k. This is maintained by the constraint $minimum(start[i], \{S[j] : 0 \leq j < m \wedge M[i][j] = 1\})$. The $minimum$ constraint can be realised either as a n-ary constraint, similar to $maximise$ above, or by using primitives (also as above).

end a one dimensional integer array. If end[i] = k then the last product for customer i is made in time slot k. This is maintained by the constraint $maximum(end[i], \{S[j] : 0 \leq j < m \wedge M[i][j] = 1\})$.

The variables in the Stacked array are then maintained by the constraint, $Stacked[i][k] = 1 \leftrightarrow start[i] \leq k \leq end[i]$. This (JSolver) model is significantly more compact than the one first described (coded in JChoco) using less variables and less constraints. This allows us to model larger problems.

3 A Reduction Operator

We now describe a reduction operator, and present a proof that this reduction is sound. Essentially this operator is a preprocess step where we input the data for the problem and remove from it products that have no effect on the optimal solution. This delivers a new representative problem that has less columns/products than the original. This reduced problem can then be solved to optimality and an optimal solution to the original problem constructed in linear time.

3.1 Removing Subsumed Columns

Our reduction operator removes a product p_i from the problem if there exists some other product p_j such that all the customers that demand product p_i also demand product p_j, i.e. we say that p_i is a subset of p_j. A practical example of this might be two products, the first a dustbin and the second a dustbin lid. No customer demands a dustbin lid if they have not already demanded a dustbin. We can ignore the production of the dustbin lid in constructing our schedule. We can then re-insert the production of the dustbin lid into the schedule immediately after the production of the dustbin without altering the cost of that schedule.

Theorem 1 *Let P_n be a problem (instance) consisting of n products $p_1, p_2, \ldots, p_n$. If, for some i and j, $p_i \subseteq p_j$, P^i_{n-1} is the problem obtained from P_n by removing p_i and s^i_{n-1} an optimal sequence for P^i_{n-1}, then $s^i_{n-1}(i)$, the sequence formed by inserting p_i after p_j in s^i_{n-1}, is an optimal sequence for P_n.*

To prove the theorem, we use two lemmas:

Lemma 1 *If s_{n-1} is a sequence corresponding to any problem with $n-1$ products, and s_n a sequence formed by adding a new product p_i, then if R and R' are the maximum number of open stacks for s_{n-1} and s_n respectively, $R' \geq R$.*

Proof Suppose that the maximum number of stacks (R) for s_{n-1} is achieved when product p_k is placed. All of the customers that have stacks open at that point have ordered products before (or at) p_k. Regardless of where p_i is placed, this is still true, and so the number of stacks open when p_k is placed is still at least R.

Lemma 2 *If s_{n-1} is a sequence corresponding to any problem with $n-1$ products, and s_n a sequence formed by adding a new product p_i, which is a subset of some existing product p_j, directly after p_j, then if R and R' are the maximum open stacks for s_{n-1} and s_n respectively, $R' = R$.*

Proof We show that when p_i is inserted after p_j, the number of stacks open when any product p_k is placed is unchanged.

We also show that the number of open stacks when p_i is placed will be less than or equal to that of p_j.

The number of open stacks open when p_k is placed equals the number of orders opened on or before p_k minus the number or orders closed before p_k. If p_k is positioned before p_i then clearly these totals will be unaffected, thus the number of open stacks will remain the same.

If p_k is positioned after p_i then the number of open orders will remain the same as all orders that require p_i will already have been opened by p_j. The number of closed orders will also remain the same as any orders that where previously closed by p_j that require p_i will now be closed by p_i. Therefore the number of open stacks will remain the same.

Because p_i will not open any new orders, the number of open stacks when p_i is placed will be the same as p_j minus the number of orders closed at p_j.

Proof of Theorem 1 Suppose that s^i_{n-1} is the optimal solution of P^i_{n-1} and that R is the maximum number of open stacks associated with s^i_{n-1}. By Lemma 2, the maximum number of open stack for $s^i_{n-1}(i)$, with p_i placed directly after p_j is R.

If there is no solution to P_n with maximum number of open stacks less than R then we are done. Suppose then, that there is some solution, s_n with maximum number of open stacks equal to R' say, where $R' \leq R$. Removing p_i from s_{n-1} will result in a sequence for P^i_{n-1} which must have maximum number of open stacks $R'' \geq R$ since we know that R is the optimal value for P^i_{n-1}. But by replacing p_i we must obtain a sequence with maximum number of open stacks $\geq R''$, by Lemma 1. Thus $R' \geq R''$. So we must have $R' = R''$, which is a contradiction as $R'' \geq R$.

3.2 Dynamic Reduction

A variant of the reduction operator is used inside the search process. This operator forces a product to be produced next, based on the state of the partial solution. Assume the search process is free to select the next product to produce, and we have computed the current number of open stacks. If there is an unselected product such that if selected next the number of open stacks does not increase, then that decision is forced. Furthermore, this selection will not increase the number of open stacks in any subsequent time slots.

3.3 Lower Bounds

A set of lower bounds is calculated for each problem, during a pre-processing stage. The first, $bound_0$, is simply the maximum number of 1s contained in any given column of the input data, i.e. the maximum demand for a product. The other bounds, together with the proof of their validity, are described in full in [2]. We summarise them below. Note that, for any customer i, $deg(i)$ denotes the *degree* of i, defined as the number of customers that order at least one of the same products as i (i.e. the number of *neighbours* of i). Also, if n is the number of customers, $d_0, d_1, \ldots, d_{n-1}$ is a list of the degrees of the customers in non-decreasing order. For any pair of customers i and j, $i \neq j$, we define $R_{i,j}$ to be the set of neighbours of i and j, not including i or j. Similarly we define $R_{i,j,k}$ for any set of (distinct) customers i, j and k.

$$
\begin{aligned}
bound_1 &= d_0 + 1 \\
bound_2 &= \max\{d_i + 1 - i : 0 \leq i < n\} \\
bound_3 &= \begin{cases} d_0 + 2 & \text{if } d_1 - d_0 > 0 \\ 0 & \text{otherwise} \end{cases} \\
bound_4 &= \max\{|R_{x,y}| + 1 : x \neq y\} \\
bound_5 &= \max\{|R_{x,y,z}| + 1 : x \neq y \neq z\}
\end{aligned}
$$

From this set of lower bounds we choose the maximum, G. If we find a solution with cost G, we know that this must be an optimal solution.

3.4 Conditional Lower Bounds

An additional set of lower bounds are calculated which take into account a partial assignment. We define D_a to be the set of customers that require product a. We define R_i to be the set of neighbours of i. $bound_a$ is the lower bound for a partial solution where a product a has been assigned to the first time slot. $bound^{a,b}$ is the lower bound when products a and b have been assigned to the first two time slots.

$$
\begin{aligned}
bound^a_1 &= \max\{|R_x \cup D_a| + 1 : x\} \\
bound^a_2 &= \max\{|R_{x,y} \cup D_a| + 1 : x \neq y\} \\
bound^a_3 &= \max\{|R_{x,y,z} \cup D_a| + 1 : x \neq y \neq z\} \\
bound^a &= \max\{bound^a_1, bound^a_2, bound^a_3\} \\
bound^{a,b}_1 &= \max\{|R_x \cup D_a \cup D_b| + 1 : x\} \\
bound^{a,b}_2 &= \max\{|R_{x,y} \cup D_a \cup D_b| + 1 : x \neq y\} \\
bound^{a,b}_3 &= \max\{|R_{x,y,z} \cup D_a \cup D_b| + 1 : x \neq y \neq z\} \\
bound^{a,b} &= \max\{bound^{a,b}_1, bound^{a,b}_2, bound^{a,b}_3\}
\end{aligned}
$$

These bounds are used by adding the following constraints.

$$
\begin{aligned}
\{s[1] = a &\rightarrow cost \geq bound^a : 1 \leq a \leq m\} \\
\{cost < bound^a &\rightarrow s[1] \neq a : 1 \leq a \leq m\} \\
\{s[1] = a \wedge s[2] = b &\rightarrow \\
cost \geq bound^{a,b} &: 1 \leq a, b \leq m \wedge a \neq b\} \\
\{cost < bound^{a,b} &\rightarrow \\
s[1] \neq a \vee s[2] = b &\rightarrow : 1 \leq a, b \leq m \wedge a \neq b\}
\end{aligned}
$$

Our current implementation has a very high cost to calculate these bounds. The worst of which being $bound^{a,b}$, which takes $O(n^4)$ time. We calculate this for all a and b therefore the total time for this is $O(m^2n^4)$. Because of the limited time available for development our implementation is very naive. As a result we have incorporated these bounds into the model, but the cost to calculate the bounds has not been included in the results. We do this to show the potential benefits of the technique without the disadvantage of our naive time restricted code. We have not measured the time to calculate

these bounds but we estimate the cost for a problem of size $n = 100$ $m = 100$ to be a few minutes. For the problem $GP5$ adding these bounds reduced the total time from one and a half hours to 26 minutes, and reduced the number of fails from 12324 to 3706.

4 The Experimental Model

Our experiments were performed using our second model, implemented in ILOG's JSolver. Problems are read in and reduced, using the operator derived from Theorem 1 above. The search process uses the dynamic reduction method described in section 3.2. The optimal solution was then found for this representative problem and proved to be optimal (either using the greatest lower bound G discussed above, or by showing that no solution with smaller cost exists). The actual optimal solution was then reconstructed by re-inserting the subsumed products, and this was done in linear time.

5 Work Not Done

We have not fully explored the effectiveness of the reduction operator. It would be interesting to measure the amount of reduction achieved on each of the challenge problems. We have only done an ad-hoc study, which demonstrated that the use of the operator invariably reduced the time taken to obtain a solution. Indeed in some cases the improvement was quite dramatic.

We have not investigated the relative performances of the various lower bounds, i.e. to measure which one is typically best. Indeed, for each of the bounds there were example instances where the bound gave the greatest lower bound. It would also be possible to increase the number of lower bounds checked. For example, extending the notation used in bounds 4 and 5, $max\{|R_{x,y,z,w}| + 1\}$ also gives an upper bound, but is expensive to compute.

We have implemented two upper-bounds but we have not incorporated these into our model. The first upper-bound reads in the problem and takes the identity permutation as a solution and then costs that. The second upper bound greedily produces a solution in a "furthest-insertion" style. It was considered that if the furthest-insertion upper-bound was reasonably good then we could use this not only as an upper-bound but also use that permutation as a static variable ordering heuristic.

We have not systematically investigated variable or value ordering heuristics.

Acknowledgements

We are grateful to ILOG SA for providing us with the JSolver toolkit via an Academic Grant licence. We would like to thank Barbara and Ian for producing such an exciting challenge.

References

[1] P. Gregory, A. Miller, and P. Prosser. Solving the rehearsal problem with planning and with model checking. In Brahim Hnich and Toby Walsh, editors, *Proceedings of the Workshop on modelling and solving problems with constraints. Held in conjunction with the 16th European Conference on Artificial Intelligence (ECAI 2004).*, pages 157–171, Valencia, Spain, August 2004.

[2] Alice Miller. Improved lower bounds for solving the open stacks problem. In Ian Gent and Barbara Smith, editors, *Proceedings of the 5th Workshop on modelling and solving problems with constraints. Held in conjunction with IJCAI-05.*, 2005.

[3] B.M. Smith. Constraint Programming in Practice: Scheduling a Rehearsal. Technical report, APES, 2003.

Appendix of results

Our experiments were performed on a Pentium4 2.8Ghz processor with 512 Mbytes of RAM running Microsoft Windows XP Professional and Java2 SDK 1.4.2.6 with an increased heap size of 512 Mbytes. Our model was coded up in ILog's JSolver. For the experiments in Tables 1 and 2 we allowed 60 seconds per instance and for Table 3 7200 seconds (2 hours) per instance. These times include model creation but do not include the time to compute the lower bounds. In Tables 1 and 2 we give the average and max results for only those instances that we found and proved optimality.

File	% solved	mean value	time (sec)			number of fails		
			mean	median	max	mean	median	max
problem_10_10.dat	100	8.07	0.14	0.14	0.24	2	0	81
problem_10_20.dat	100	8.92	0.16	0.17	0.41	5	0	679
problem_15_15.dat	100	12.87	0.21	0.19	1.14	42	1	3440
problem_15_30.dat	92.73	14.02	0.65	0.22	35.25	446	0	40820
problem_20_10.dat	100	15.88	0.21	0.2	0.66	40	9	1422
problem_20_20.dat	85	17.98	0.85	0.23	36.47	1167	1	79327
problem_30_10.dat	100	23.95	0.27	0.24	2.89	128	27	10208
problem_30_15.dat	85.91	25.98	1.24	0.28	37.95	2583	26	125372
problem_30_30.dat	66.36	28.55	3.01	0.3	49.74	898	0	17141
problem_40_20.dat	62.73	36.49	7.03	0.42	59.77	6948	56	67796
ShawInstances.txt	68	13.68	0.87	0.33	9.64	1006	77	16015
wbo_10_10	100	5.95	0.18	0.19	0.23	11	6	60
wbo_10_20	100	7.35	0.24	0.22	0.56	45	5	1005
wbo_10_30	97.5	8.2	0.29	0.25	0.77	20	3	244
wbo_15_15	100	9.35	0.26	0.24	0.44	72	25	618
wbo_15_30	60	11.62	2.07	0.36	48.88	773	24	25680
wbo_20_10	100	12.9	0.23	0.22	0.33	49	28	342
wbo_20_20	81.11	13.7	1.58	0.39	34.75	2392	119	71762
wbo_30_10	100	20.05	0.29	0.27	1.06	119	66	2345
wbo_30_15	98.33	20.96	0.65	0.42	5.52	811	257	10234
wbo_30_30	32.14	22.99	4.09	0.63	39.92	1921	148	32462
wbop_10_10	100	6.75	0.17	0.17	0.2	6	6	21
wbop_10_20	100	8.07	0.26	0.22	1.72	48	0	1515
wbop_10_30	95	8.55	0.69	0.25	12.55	212	1	4813
wbop_15_15	100	10.37	0.24	0.25	0.39	54	31	494
wbop_15_30	75	12.27	1.85	0.33	42.33	608	11	14609
wbop_20_10	100	14.28	0.21	0.22	0.3	40	27	196
wbop_20_20	85.56	14.87	2.09	0.38	47.45	2811	113	97374
wbop_30_10	100	22.48	0.26	0.27	0.45	70	60	506
wbop_30_15	95	22.38	0.57	0.41	2.33	625	207	4578
wbop_30_30	47.86	24.37	4.32	0.49	53	2256	30	42519
wbp_10_10	100	7.33	0.15	0.16	0.2	5	2	73
wbp_10_20	100	8.71	0.17	0.17	0.28	6	0	106
wbp_10_30	100	9.31	0.19	0.19	0.45	13	0	545
wbp_15_15	100	11.05	0.22	0.2	0.42	44	14	365
wbp_15_30	87.5	13.1	0.91	0.25	27.59	605	1	20633
wbp_20_10	100	15.13	0.21	0.2	0.28	35	26	236
wbp_20_20	86.67	15.42	2.15	0.3	42.33	4660	60	209328
wbp_30_10	100	23.18	0.3	0.25	1.14	187	47	2794
wbp_30_15	75	22.98	1.65	0.42	23.59	3864	254	68342
wbp_30_30	47.86	24.96	3.64	0.36	59.2	2955	12	103704

Table 1: Times and number of fails to find the optimal solution for each data set

File	% solved	mean value	time (sec)			number of fails		
			mean	median	max	mean	median	max
problem_10_10.dat	100	8.07	0.15	0.14	0.53	15	4	1665
problem_10_20.dat	100	8.92	0.21	0.17	12.52	159	4	35527
problem_15_15.dat	100	12.87	0.57	0.2	52.19	1234	9	160625
problem_15_30.dat	92.73	14.02	1.26	0.22	43.47	1803	4	29657
problem_20_10.dat	100	15.88	0.31	0.22	1.98	416	29	7988
problem_20_20.dat	85	17.98	3.66	0.22	54.13	9030	5	86966
problem_30_10.dat	100	23.95	0.57	0.33	6.45	1014	188	22108
problem_30_15.dat	85.91	25.98	6.54	0.28	54.72	15474	46	117260
problem_30_30.dat	66.36	28.55	1.68	0.3	49.74	265	5	6880
problem_40_20.dat	62.73	36.49	5.58	0.28	59.77	4268	5	26289
ShawInstances.txt	68	13.68	19	14.02	58.78	46529	35835	120177
wbo_10_10	100	5.95	0.19	0.19	0.31	26	17	262
wbo_10_20	100	7.35	0.78	0.22	17.41	624	21	17438
wbo_10_30	97.5	8.2	1.45	0.25	30.94	1260	10	24299
wbo_15_15	100	9.35	1.13	0.38	7.53	2793	252	26457
wbo_15_30	60	11.62	1.77	0.38	22.59	541	32	10375
wbo_20_10	100	12.9	0.3	0.28	0.77	284	176	2385
wbo_20_20	81.11	13.7	9.92	0.55	54.55	20594	283	94726
wbo_30_10	100	20.05	0.55	0.47	1.89	812	542	3755
wbo_30_15	98.33	20.96	8.06	2.64	51.77	21915	4539	142875
wbo_30_30	32.14	22.99	1.72	0.31	11.98	430	9	4409
wbop_10_10	100	6.75	0.18	0.19	0.28	21	14	234
wbop_10_20	100	8.07	1.4	0.22	24.11	2219	6	43075
wbop_10_30	95	8.55	0.72	0.25	12.55	228	6	4821
wbop_15_15	100	10.37	0.54	0.27	6.98	1034	63	16378
wbop_15_30	75	12.27	2.78	0.27	42.33	1439	8	15190
wbop_20_10	100	14.28	0.27	0.25	0.69	219	110	2020
wbop_20_20	85.56	14.87	5.12	0.39	57.14	9828	154	86323
wbop_30_10	100	22.48	0.37	0.36	0.84	358	242	1764
wbop_30_15	95	22.38	7	1.98	43.05	18200	4115	131457
wbop_30_30	47.86	24.37	1.56	0.3	36.69	280	7	6207
wbp_10_10	100	7.33	0.16	0.16	0.33	23	7	474
wbp_10_20	100	8.71	0.24	0.17	2.72	196	6	5901
wbp_10_30	100	9.31	0.3	0.19	3.75	310	4	7527
wbp_15_15	100	11.05	1.29	0.22	44.02	4457	31	183345
wbp_15_30	87.5	13.1	2.28	0.25	42.92	3678	7	36740
wbp_20_10	100	15.13	0.32	0.3	0.89	458	210	3095
wbp_20_20	86.67	15.42	7.23	0.34	58.73	21671	121	131545
wbp_30_10	100	23.18	0.73	0.5	2.64	1535	689	8030
wbp_30_15	75	22.98	8.34	0.89	58.75	24118	774	121667
wbp_30_30	47.86	24.96	1.56	0.3	33.42	841	6	7876

Table 2: Times and number of fails to prove the optimal solution for each data set

File	solved	best value	time to best solution	fails to best solution	time to prove	fails to prove
Miller19	Yes	13	0.312	0	0.328	17
GP1	Yes	45	0.484	0	0.5	6
GP2	Yes	40	7.375	448	7.375	458
GP3	Yes	40	11.78	399	11.78	411
GP4	Yes	30	47.89	7523	47.89	7542
GP5	Yes	95	1329.71	3700	1329.73	3706
GP6	Yes	75	6812.98	81443	6813	81469
GP7	Yes	75	2020.32	428	20.32	453
GP8	No	84	2.985	22	7200	1105756
NWRS1	Yes	3	0.172	12	0.172	15
NWRS2	Yes	4	0.203	11	0.203	17
NWRS3	Yes	7	0.25	28	0.375	579
NWRS4	Yes	7	0.219	21	0.219	31
NWRS5	Yes	12	0.437	155	26.5	77122
NWRS6	Yes	12	1.015	354	203.01	615419
NWRS7	No	10	23.15	38377	7200	16551474
NWRS8	No	16	681.46	154359	7200	4554999
SP1	Yes	9	0.391	282	287.58	1134916
SP2	No	22	11.51	3415	7200	3009686
SP3	No	40	97.7	8115	7200	236553
SP4	No	65	31.72	1148	7200	219177

Table 3: Times and number of fails to prove the optimal solution for each problem