Impact-Based Search Strategies for Constraint Programming

Philippe Refalo

ILOG, Les Taissounieres, 1681, route des Dolines
06560 Sophia Antipolis, France
refalo@ilog.fr

Abstract. A key feature of constraint programming is the ability to design specific search strategies to solve problems. On the contrary, integer programming solvers have used efficient general-purpose strategies since their earliest implementations. We present a new general purpose search strategy for constraint programming inspired from integer programming techniques and based on the concept of the *impact* of a variable. The impact measures the importance of a variable for the reduction of the search space. Impacts are learned from the observation of domain reduction during search and we show how restarting search can dramatically improve performance. Using impacts for solving multiknapsack, magic square, and Latin square completion problems shows that this new criteria for choosing variables and values can outperform classical general-purpose strategies.

1 Introduction

One of the key features of constraint programming is the ability to exploit the structure of a problem to design an adapted search strategy to solve it. Since the earliest constraint logic programming systems based on Prolog, facilities were available to design sophisticated search procedures. As a consequence, not much use is made of general-purpose strategies in constraint programming. The most popular ones are based on first selecting variables having the minimum domain size [11]. Ties can be broken with the dynamic degree of variables [6]. Other variations include the ratio between the size of the domain and the degree of the variable [4] or looking at the neighborhood structure of the variable [15, 3].

On the other hand, in integer programming solvers, it is considered that designing a search strategy is complex. It is true that the underlying concepts of integer programming that need to be understood (i.e., relaxed optimal solution, dual values, reduced costs) are not very intuitive. In comparison, the constraint programming concept of domain reduction is easier to understand and to use for the design of a search strategy.

As a consequence, a class of techniques for efficient general-purpose strategies in integer programming has emerged. The emphasis of integer programming being optimization, these techniques are based on estimating the importance of a variable with respect to the variation of the objective function value. The

M. Wallace (Ed.): CP 2004, LNCS 3258, pp. 557–571, 2004.

criteria used is called a *pseudo-cost* [7]. These techniques are so efficient that it has become rare that someone can compete with them by designing his or her own strategy.

In the following we propose new general-purpose strategies for constraint programming inspired by the notion of pseudo-cost. These strategies are based on the concept of *impact*. An impact measures the importance of an assignment $x = a$ for the search space reduction. Impacts are obtained from the domain reduction involved by assignments made during search.

In this context, probing on some variables at a node to get the true impacts can improve performance. This is analogous to *strong branching* [5] which is an efficient technique for solving hard integer programming problems. Furthermore, restarting search permits the solver to use the impacts learned during search in order to start a new, and hopefully smaller, search tree by making more relevant choices at the beginning.

Our goal is to provide better performance than the general strategies based on the domain size or on the degree. Experiments illustrate the benefits of this new approach on multiknapsack, magic square and Latin square completion problems. Before going into details, let us review basic facts about search strategies.

2 Search Strategies

The problem considered is to rapidly find a solution of a set of constraints $S = \{c_1, \ldots, c_m\}$ over a set of variables $V = \{x_1, \ldots, x_n\}$ having domains $D_{x_1}, \ldots, D_{x_n}$ or to prove that none exists. A solution is a set of assignments $\{x_1 = a_1, \ldots, x_n = a_n\}$ such that $a_i \in D_{x_i}$ and such that each constraint is satisfied when each variable x_i is replaced by the value a_i. Optimization problems are not considered here but only satisfaction problems.

To solve this problem, a common approach is *backtrack search*. At each step (or node), backtrack search chooses a non-assigned variable x_i and a subset $E \subset D_{x_i}$ and states the choice point

$$x_i \in E \text{ or } x_i \in D_{x_i} - E$$

In general, the subset E contains a single value. The constraint $x_i \in E$ is added to the current problem (the node problem) and variable domains are reduced with constraint propagation. When a constraint is violated or when a domain is empty, the search procedure backtracks to the last choice point by undoing constraint additions and tries the first unexplored branch encountered. When the domain of all variables contains a single value and no constraint is violated, a solution is found.

The performance of backtrack search (or the number of nodes it traverses) varies dramatically depending on the strategy used to choose a variable and the value for this variable. Searching for a solution to a problem involves facing two objectives that can be contradictory:

1. If the problem has a solution, find one quickly.
2. If the problem has no solution, create a small search tree to prove it.

In Case 1., using the structure of the problem can be very helpful. However, general-purpose strategies ignore the overall problem structure. These strategies assume that the problem is Case 2., with the hope that reducing the size of the whole search tree will also reduce the size of the search tree needed to reach a first solution.

In this context, there are principles for reducing the search effort that are worth mentioning here:

First principle. Since all variables have to be instantiated, first choosing the variable that maximally constrains the rest of the search space reduces the search effort. This principle is popular and widely applied. It is often implemented by choosing the variable having the smallest domain first or the one that participates in the largest number of constraints (the variable degree) first or a combination of both. Updating the domains and degrees at each node permits the solver to make better choices [11].

Second principle. This concerns the choice of a value. It is not that useful if the problem has no solution. However, if the problem has solutions, a solution can be reached more quickly if one chooses a value that maximizes the number of possibilities for future assignments.

Third principle. Make good choices at the top of the search tree. Choices made at the top of the search tree have a huge impact on its size and a bad choice can have disastrous effects. Applying this principle means that some effort must be made *before* starting the search to make good choices from the beginning.

The two first principles have been used in constraint programming for a long time but the last one is not always applied.

It is interesting to see that the integer programming community has applied these three principles since the early seventies but in a different context (see next section). In integer programming, the emphasis is on finding optimal solutions. An estimation of the objective function improvement is associated to a variable. According to these principles, the variable chosen first is the one involving the largest improvement; then the branch chosen first is the one that involves the least improvement.

Inspired by the integer programming strategies, we introduce the concept of *impact* in constraint programming. An impact associates a value with each assignment $x_i = a_j$, $a_j \in D_{x_i}$ to measure the importance of this assignment for the search space reduction. Here also, to reduce the search tree, a variable with the greatest impact should be chosen first and a value with the smallest impact should be used first.

3 Pseudo-costs in Integer Programming

Pseudo-costs are widely used in integer programming solvers such as CPLEX [1]. They have been introduced briefly in [2] and fully described in [7]. They have been successfully used for the default search strategies of modern integer programming solvers. Even if it is possible to redesign the search strategy in

these solvers, pseudo-cost based strategies are so effective that it has become rare for someone to obtain a speedup by designing a dedicated strategy.

In integer programming problems, constraints are linear, and the domain of a variable is an interval of real or integer values. In addition, there is a linear cost function to be minimized (the maximization case is ignored, as it is similar). Integer programming solvers maintain a relaxed optimal solution at each node. This solution is given by a linear solver (such as the simplex method) applied to the node problem where integrality conditions on integer variables have been removed. This relaxed solution on a variable x is noted x^*. The value of the objective function in this solution is z^*. It is a lower bound on the optimal value of the problem. The value z^* increases as variables are instantiated during search. It is the amount of that increase that is used to select variables.

3.1 Variable and Value Selection

Consider a variable x whose non-integer value is $x^* = \lfloor x^* \rfloor + f$ where $0 < f < 1$. Forcing x to be an integer is done by creating the choice point

$$x \leq \lfloor x^* \rfloor \text{ or } x \geq \lceil x^* \rceil$$

Let z^* be the objective function value before the choice point. Let Δ_{down} be the increase of the objective value when adding the constraint $x \leq \lfloor x^* \rfloor$ and Δ_{up} be the increase of the objective value when adding the constraint $x \geq \lceil x^* \rceil$.

The values Δ_{down} and Δ_{up} can be computed for each variable having a non-integer value by solving two linear programs. From the first principle above, the variable to be chosen is the one having a maximum impact on the objective. In practice, it is the one that maximizes the weighted sum

$$v(x) = \alpha \min(\Delta_{down}, \Delta_{up}) + \beta \max(\Delta_{down}, \Delta_{up})$$

Usually more importance is given to the maximum of Δ_{down} and Δ_{up}. For $\alpha = 1$, choosing β greater than 3 gives good results (see [12]).

From the second principle above, the first branch to explore is the one that creates the smallest improvement in the objective function value with the hope of getting solutions with a low objective value earlier.

3.2 Pseudo-costs

The pseudo-costs of a variable x measure the increase of the objective function value per unit of change of x when adding the constraint $x \leq \lfloor x^* \rfloor$ or $x \geq \lceil x^* \rceil$. The *down pseudo cost* $PC_{down}(x) = \Delta_{down} \ / \ f$ corresponds to the decrease of x^* and the *up pseudo-cost* $PC_{up}(x) = \Delta_{up} \ / \ (1-f)$ corresponds to the increase of x^*.

A fundamental observation about pseudo-costs is that experiments reveal that the pseudo-costs of a variable tend to be the same from one node to another (see [7] and [12]). As a consequence we can avoid solving two linear programs

(one for Δ_{down} and one for Δ_{up}) for each non-integer variable at each node, which is computationally very heavy. We can estimate that the up and down pseudo-costs of a variable x at a node are likely to be the average of the ones observed so far when choices are made on x. This averaged pseudo-cost (PC^e) is used to compute an estimation of $v(x)$ from an estimation of the objective function variation:

$$\Delta^e_{down} = f * PC^e_{down}(x) \quad \text{if } x \leq \lfloor x^* \rfloor \text{ is added}$$
$$\Delta^e_{up} = (1-f) * PC^e_{up}(x) \quad \text{if } x \geq \lceil x^* \rceil \text{ is added}$$

3.3 Strong Branching

On some problems the pseudo-costs may vary between nodes and it may be worth computing the estimation by solving a linear program. As it is costly to perform this operation on every non-integer variable, this is done only for some of them. In particular, this is done for breaking ties between variables that have similar estimations. This process has been introduced in [5] and is called *strong branching*.

More precisely, strong branching consists of performing a limited number of dual simplex iterations on a subset of variables having a non-integer value. Dual simplex tries to recover optimality that was destroyed by the addition of the constraint $x \leq \lfloor x^* \rfloor$ or $x \geq \lceil x^* \rceil$. After a certain number of iterations, the simplex iterations are stopped. The current (maybe still non-integer) value x^* and the current objective value z^* are used to compute an approximated pseudo-cost. It is used to differentiate variables instead of the averaged pseudo-cost. Although strong branching can create a significant overhead at each node, it can pay off on hard integer programs by dramatically reducing the number of nodes [1].

3.4 Pseudo-costs Initialization

At the beginning of search, pseudo-costs are unknown. Having pseudo-costs at this time is extremely important since choices made high in the tree are crucial. Computing explicit pseudo-costs by bounding up and down variables that have a non-integer value can degrade performance significantly [2]. As for strong branching, a trade-off consists of performing a limited number of dual simplex iterations [12]. The approximated pseudo-cost computed in this way is usually replaced by the first observed one.

4 Impacts in Constraint Programming

As with pseudo-costs, the basic idea of impacts is intuitive. In constraint programming, when a value is assigned to a variable, constraint propagation reduces the domains of other variables. We consider that the number of all possible combinations of values for the variables (the Cartesian product) is an estimation

of the search size. Therefore, an estimation of the size of the search tree is the product of every variable domain size:

$$P = |D_{x_1}| \times \ldots \times |D_{x_n}|$$

4.1 Impact of an Assignment

If we look at this product before (P_{before}) and after (P_{after}) an assignment $x_i = a$ we have an estimation of the importance of this assignment for reducing the search space. This reduction rate is called the *impact* of the assignment

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}}$$

The higher the impact, the greater the search space reduction. From this definition, an assignment that fails has an impact of 1.

The impact of assignments can be computed for every value of all non-instantiated variables, but this can create a huge overhead. From the experiments we have made, impacts, like pseudo-costs, do not vary much from node to node. The impact value distribution of a given assignment almost always presents a sharp and unique peak centered on the average value. An important consequence is that the impact of an assignment at a given node can be the average of the observed impacts of this assignment up to this point. If K is the index set of impacts observed so far for assignment $x_i = a$, $\bar{I}$ is the averaged impact:

$$\bar{I}(x_i = a) = \frac{\sum_{k \in K} I^k(x_i = a)}{|K|}$$

A nice consequence is that impacts do not need to be computed explicitly at each node but are available almost for free.

4.2 Impact of a Variable

The impact of a variable x_i can be the average of impacts $\bar{I}(x_i = a)$ for $a \in D_{x_i}$. A more accurate measure would use only the values remaining in its domain. Thus, if the current domain of x_i at a node is D'_{x_i} we have

$$\tilde{I}(x_i) = \frac{\sum_{a \in D'_{x_i}} \bar{I}(x_i = a)}{|D'_{x_i}|}$$

This approach is not accurate enough. The goal is to choose a variable having the largest impact when assigning to it one of the values remaining in its domain. Since it is assumed that each value will be tried (the hypothesis is that the problem is infeasible as mentioned above) we need to consider the search reduction if *every* value of the domain is tried. Let P be the product of the

domain sizes at a node and consider a variable x_i. The estimation of the size of the search space when trying $x_i = a$ with $a \in D'_{x_i}$ is

$$P \times (1 - \bar{I}(x_i = a))$$

This is an estimation of the size of the search tree for $x_i = a_j$. If we were to try every value, an estimation of the search tree is the *sum* of the estimation for each value remaining in the domain:

$$\sum_{a \in D'_{x_i}} P \times (1 - \bar{I}(x_i = a))$$

The value P is a constant at a node and it is not relevant to compare the impact of several variables at the same node. Finally, the impact of a variable that depends on its current domain is defined as

$$\mathcal{I}(x_i) = \sum_{a \in D'_{x_i}} 1 - \bar{I}(x_i = a)$$

Experiments we have made comparing the use of the average impact $\tilde{I}(x)$ and the use of the sum of impacts on values $\mathcal{I}(x)$ show that using $\mathcal{I}(x)$ is much more efficient over all the problems we have tested.

4.3 Initialization of Impacts

As we said previously, some effort must be made before starting search to compute impacts. This helps to differentiate variables at the root node and to make better choices at the beginning of the search where this is crucial. One approach is to try every value for every variable. However on problems where domains are large this can be more costly than solving the problem itself. Impacts can be approximated before search in order to be efficiently computed.

The basic idea is to divide the domain D_{x_i} of a variable x_i into distinct subdomains $D_{x_i} = D^1_{x_i} \cup \ldots \cup D^k_{x_i}$ and compute the impact of $x_i \in D^w_{x_i}$. The impact of a value of a subdomain is then

$$I(x_i = a) = 1 - \frac{1 - I(x_i \in D^w_{x_i})}{\left|D^w_{x_i}\right|} \text{ for } a \in D^w_{x_i}$$

This formula assumes that the overall impact of x_i is shared equally between its values. The advantage is that when two domain reductions $x_i \in D^u_{x_i}$ and $x_i \in D^v_{x_i}$ have the same impact, the values in the largest domain have an impact greater than the one of the smallest domain since in one case there are more values to achieve the same reduction.

It is not realistic to decide to split a domain by imposing a maximal size on the $D^j_{x_i}$ because variables with a small domain, such as binary variables, will never be considered in an approximation and large domains will require more computation

than smaller ones. Instead we recursively split a domain into two parts a certain number of times s (the splitting value) in order to ensure that small domains will be approximated and to bound the effort for large ones. Domains are then divided into at most 2^s subdomains. Here also, an approximated impact is replaced by the first truly observed impact.

4.4 Node Impacts

To get accurate impact values, one can compute the impact of a set of variables at a node by trying all possible assignments for each variable from the set. This is a costly technique similar to strong branching. It should be used to break ties in a subset of variables that have equivalent impacts. However, it can pay off if there are many ties to break. It can also be relevant when impacts appear to vary more than expected from one node to another.

Additionally, removing those of the tried values that fail from the domains leads to a weaker form of singleton arc-consistency [13] that helps to further reduce the search tree.

In order to reduce the time spent on computing node impacts, they can be approximated the same way that initial impacts can.

4.5 Restarts

Restarting the search (i.e., stopping the search process and restarting it from scratch) is an effective technique when using randomized strategies [8]. Restart has been used to solve Boolean satisfaction problems. In particular it is used in the SAT solver `Chaff` [14] where variable and value choices at the top of the tree are made randomly until a certain depth.

The basic idea behind the association of restart and randomization is to give equal chances to all parts of the search space to be explored at the beginning of the search.

Concerning impacts, as the search progresses we get more and more accurate impacts. We learn in this way what are the important assignments and variables. Consequently, we may discover that variables assigned high in the search tree are not as important as they seemed at the beginning. Therefore, as with any strategy that learns during search, it is useful to restart search from time to time in order to use the most recent information with the hope that the new search tree will be smaller. Section 5.5 give more details about the automatic restart procedure and some experimental results.

5 Experiments

The experiments in this section aim to show the relevance of impacts with regards to classical strategies. We also demonstrate the importance of impact initialization, of computing node impacts and of restart by comparing these approaches all together. For this purpose we have studied multiknapsack and magic square problems. Additional results on Latin square completion problems are also given in the last subsection as we tried to solve the instance in many different ways.

5.1 Problems and Hardware

Two sets of problems are considered. The first one is a set of multiknapsack problems where only a solution needs to be found (there is no cost function). These problems are modeled by a set of linear constraints over binary variables. They are usually hard for constraint programming solvers. Five problems come from the smaller set of the operations research library[1] where the cost function is constrained to take its optimal value (problems `mknap1-*`). These problems have between 6 and 11 constraints and from 15 to 50 variables. Four other problems are isolated subproblems of a real-life configuration problem (problems `mc*`). These subproblems have 2 constraints and 31 variables.

The second set of problems is magic square problems. The problem consists of filling in a matrix $n \times n$ by a set of all different values such that the sum of values in each row, in each column, and in the two diagonals is the same. The model we use contains one *alldifferent* constraint over all variables and $2n + 2$ linear constraints to constrain the sum of values.

All experiments in this section were made on a Pentium 4 / 2.8Gz machine using ILOG Solver 6.0. The time is given in seconds and the time limit is 1500s. The number of choice points is shown as a measure of the size of the search tree.

5.2 Classical Strategies

We compare our strategy with the minimum domain strategy, which is the classical strategy. We have tried variations around this strategy. In particular the Brelaz method to break ties has been tested [6], as well as the ratio between domain size and variable degree presented in [4]. None of the approaches improved the minimum size domain significantly on the problems considered. Therefore only the minimum size domain results are given in the comparisons. We also compare with a random strategy (i.e. it chooses variables and values randomly). This permits us to avoid biasing the comparison with the exceptionally bad results of the minimum size domain strategy due to its dependence on the order of variables.

5.3 Impacts and Initialization

To show the relevance of impact-based strategies and of impact initialization, this test compares

- a random search strategy (variables and values are chosen randomly);
- the minimum domain size strategy, that chooses first the variable with the minimum domain size and the smallest available value for this variable;
- a strategy using impacts where the best variable x_i is the one that maximizes $\mathcal{I}(x_i)$ and the best value a is the one that minimizes $\bar{I}(x_i = a)$;
- the same strategy with initialization of impacts before search.

[1] Problems are available at
`http://mscmga.ms.ic.ac.uk/jeb/orlib/mdmkpinfo.html`

	Random		Min. domain size		Impact w/o init.		Impact + init.	
Problems	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.
mknap1-0	0.02	2	0.01	2	0.02	2	0.03	2
mknap1-2	0.02	10	0.02	37	0.05	15	0.03	26
mknap1-3	0.03	408	0.03	384	0.06	304	0.05	186
mknap1-4	0.55	11485	0.66	16946	0.24	3230	0.05	434
mknap1-5	48.91	1031516	3.33	99002	1.7	29418	0.22	4247
mknap1-6	>1500		716.75	21532775	>1500		50.46	902319
mc3923	14.67	491445	867.24	42328413	197.65	5862508	0.38	11768
mc3800	2.28	75270	131.22	6396644	248.81	7348618	0.06	1769
mc3888	53.40	1784812	722.73	35242940	33.93	1007735	1.36	44682
mc3914	26.91	899114	895.10	43631272	305.56	9084462	0.44	14390

Fig. 1. Initialization of impacts on multiknapsack problems.

	Random		Min. domain size		Impact w/o init.		Impact + init.	
Size	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.
5	0.08	258	0.02	632	0.05	148	0.13	3486
6	12.20	458023	1.78	125789	52.62	1067734	0.16	3458
7	301.17	9756428	>1500		2.30	40310	0.11	1594
8	>1500		>1500		48.68	748758	9.70	134031
9	>1500		>1500		>1500		1.47	15244
10	>1500		>1500		>1500		10.13	67581
11	>1500		>1500		>1500		>1500	

Fig. 2. Initialization of impacts on magic square problems.

The initialization of impacts before search is not approximated. The overhead of this full initialization is negligible for both types of problem. In impact strategies, remaining ties are broken with a random variable and value choice.

The results are shown in Figure 1 and 2.

Concerning multiknapsack problems, the minimum domain size strategy does not perform well except for smaller instances. This is due to the binary variables. A free variable always has a domain size of 2. As a consequence, this strategy only depends on the order of variables. It chooses the first free variable and then the value 0 before the value 1. Choosing randomly can improve performance on the configuration multiknapsack problems.

Using impacts outperforms both minimum domain size and random strategies. It is interesting to note that without initialization the results are much worse (even worse than the random strategy sometimes) and the instance `mknap1-6` cannot be solved. This shows the high importance of impact initialization for this problem. Moreover the cost of computing these initial impacts (that are not approximated for this test) is negligible here.

Magic square problems also benefit from impacts. Without initialization, problems of size 8 can be solved but initialization of impacts permits the solver to go up to size 10. Here again the cost of computing initial impacts before search is negligible with respect to the solving time.

	Impacts		Node Impacts	
Problems	Time	Ch.pts.	Time	Ch.pts.
mknap1-0	0.03	2	0.03	0
mknap1-2	0.03	26	0.03	29
mknap1-3	0.05	186	0.05	79
mknap1-4	0.05	434	0.13	1321
mknap1-5	0.22	4247	0.17	2154
mknap1-6	50.46	902319	20.34	98805
mc3923	0.38	11768	3.03	24890
mc3800	0.06	1769	0.22	2897
mc3888	1.36	44682	1.31	12571
mc3914	0.44	14390	0.59	6415

Fig. 3. Node impacts on multiknapsack problems.

	Impacts		Node Impacts	
Size	Time	Ch.pts.	Time	Ch.pts.
5	0.13	3486	0.08	439
6	0.16	3458	0.19	2353
7	0.11	1594	0.25	259
8	9.70	134031	2.16	17535
9	1.47	15244	2.00	1844
10	10.13	67581	>1500	
11	>1500		38.01	90483
12	>1500		55.76	92650
13	>1500		42.50	7471
14	>1500		187.83	106258
15	>1500		>1500	
16	>1500		587.23	189884
17	>1500		>1500	

Fig. 4. Node impacts on magic square problems.

5.4 Node Impacts

Note that node impacts can be approximated the same way that initial impacts can. However this does not greatly improve performance on the magic square and multiknapsack problems. Therefore full impacts are computed at a node.

The experiments done with the computation of node impacts compare

- an impact strategy with initialization;
- and impact strategy with initialization where ties are broken with node impacts and remaining ties are broken randomly.

The results go beyond expectations (see Figure 3 and 4). For most problems, the use of node impacts reduces the size of the search tree up to a factor of 10.

For the multiknapsack problem it does not increase the overall running time in general (except for `mc3923` which is significantly slower) and even helps to reduce the solving time for the hardest instance (`mknap1-6`).

For the magic square problem node impacts permit the solver to solve much larger instances, although the problem of size 10 cannot be solved within the time limit. However, instances 11, 12, 13, 14 and 16 can be solved within the time limit.

5.5 Restart

We have tried various strategies for doing automatic restarts. A very simple approach proved to be quite effective. Before the first run we imposed a maximum number of failures of $3n$ where n is the number of variables. This value is called the *cutoff*. Then at each restart we increase this value by multiplying it by a value $e > 1$. The increase of the cutoff permits us to guarantee that the search process

is complete. The cutoff will always reach a value that is sufficiently large enough to explore the whole search tree in a single run. Setting e to 2 theoretically increases the size of the search tree by a factor of 2 (if we consider that restart is useless). However this increases the cutoff too quickly and better results were obtained with $e = \sqrt{2}$ which theoretically increases the cumulative size by a factor of $2\sqrt{2}$.

	Impacts		Random + MDS		Impacts + restart	
Size	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.
5	0.13	3486	0.03	306	0.05	840
6	0.16	3458	0.05	324	0.03	454
7	0.11	1594	0.06	572	0.05	320
8	9.70	134031	1.69	21604	1.31	29107
9	1.47	15244	294.49	3413789	0.25	3206
10	10.13	67581	187.14	2111256	2.19	35796
11	>1500		>1500		0.67	5227
12	>1500		>1500		14.89	182392
13	>1500		>1500		13.22	108501
14	>1500		>1500		21.25	83047
15	>1500		>1500		38.64	95769
16	>1500		>1500		1354.64	975655
17	>1500		>1500		>1500	

Fig. 5. Restart on magic square problems.

The experiments using restarts compare

- an impact strategy with initialization;
- random choices until depth 5 and minimum domain size (MDS) strategy below using restart;
- an impact strategy with initialization using restart.

The second strategy permits us to measure the importance of restart.

Results on the magic square problem are presented in Figure 5. Restarting search using random choices until depth 5 improves the minimum domain size strategy (see figure 2) but it does not give better results than using impacts without restart. When using impacts, restart improves the performance of the instances solved by using the impacts alone and solves all instances within the time limit up to size 16.

The results for the multiknapsack problems are not presented here. Using restarts is ineffective on these problems. The time to find a solution increases by a factor between 2 and 3 on average (close to $2\sqrt{2}$). This means that initial impacts are good enough to avoid bad choices at the beginning on these instances.

5.6 Latin Square Completion Problems

The third set of problems tested is the Latin square completion problem. The Latin square problem consists of filling an $n \times n$ matrix with values from the set $\{1, \ldots, n\}$ such that values on each line and on each column are all different. The problem is simply formulated by one *alldifferent* constraint for each row and each column. The Latin square completion problem has, in addition, some variables already instantiated in the matrix. The set of problems considered is a selection of hard instances generated randomly [10].

The experiments using restarts compare

- the minimum domain size strategy;
- random choices until depth 5 and minimum domain size (MDS) strategy below using restarts;
- an impact strategy with initialization;
- and impact strategy with initialization using restarts.

In these problems, variables have a large domain and initialization without approximation can be costly (up to 300s on some problems) and therefore we have used approximations with a splitting value $s = 4$. The domains are then divided in at most 16 subdomains and the maximum time for initialization goes down to 7 seconds.

		min. domain size		Random + MDS		Impacts		Impacts + restart	
order	holes	Time	Ch. pts	Time	Ch. pts	Time	Ch.pts	Time	Ch.pts
18	120	0.06	2	0.05	4	0.05	1	0.05	2
30	316	8.81	18627	15.89	23545	0.07	30	0.06	31
30	320	0.47	856	0.52	552	0.17	277	0.15	278
33	381	>1500		>1500		>1500		>1500	
35	405	>1500		>1500		9.84	16367	453.04	752779
40	528	>1500		>1500		>1500		>1500	
40	544	>1500		>1500		>1500		>1500	
40	560	>1500		>1500		>1500		225.34	289686
50	2000	2.95	225	8.42	1728	13.52	1734	13.53	1735
50	825	>1500		>1500		>1500		>1500	
60	1440	>1500		>1500		>1500		>1500	
60	1620	>1500		>1500		>1500		115.91	56050
60	1692	410.63	193687	>1500		41.40	21746	299.23	164048
60	1728	>1500		>1500		7.95	2332	8.02	2333
60	1764	849.78	486639	>1500		639.82	334699	99.32	48485
60	1800	>1500		865.86	197549	7.81	1933	7.78	1934
70	2450	>1500		>1500		>1500	382745	175.94	43831
70	2940	10.17	1463	138.53	22585	32.39	3731	32.51	3732
70	3430	9.92	1252	53.67	7659	51.39	3072	51.03	3073

Fig. 6. Results on Latin square completion problems.

The results are presented in Figure 6. These instances are hard for the minimum domain size strategy which cannot solve 11 over 19 instances. The addition of restart with random choices until depth 5 does not improve it. Using impacts permits the solver to solve 3 more instances. Using restart in addition to impacts permits the solver to solve another 3 instances. Thus 5 instances out of 19 remain unsolved with the new strategy with restarts.

It is worth mentioning that using node impacts to break ties without restart as presented in Section 4.4 solves the instance 60/1440 in 14.5 seconds and 2583 choice points. However the instance 70/2450 cannot be solved with node impacts.

Additionally, we have tried a stronger propagation by using the alldifferent constraint on a matrix [9]. It improves computation times in several cases but not all of them. However, it permits the solver to solve the instance 33/381 which can only be solved by SAT solvers (see [10]). It is solved in 1357s and 1128390 choice points. Thus 3 instances remain unsolved for our search strategy.

6 Conclusion

The new general-purpose strategy based on impacts presented in this paper is inspired from integer programming techniques where default search strategies are extensively used. Impacts permit us to benefit from the search effort made up to a certain node by storing the observed importance of variables. As we have seen, impact initialization, search restart and computation of node impacts are central techniques for improving performance. These techniques permit us to solve instances that remain unsolved with standard strategies. The significant improvement over the minimum size domain strategy on multiknapsack, magic square and Latin square completion problems and the negligible overhead created by the update of impacts make us believe that this class of strategies is a candidate for becoming the basis of a default search goal for constraint programming.

Acknowledgments

The author thanks Oliver Lhomme, Jean-Charles Régin and Paul Shaw for animated discussions about search strategies.

References

1. ILOG CPLEX 9.0. User Manual. ILOG, S.A., Gentilly, France, September 2003.
2. M. Benichou, J.M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, and O. Vincent. Experiments in mixed-integer linear programming. Mathematical Programming, (1):76–94, 1971.
3. C. Bessiere, A. Chmeiss, and L. Sais. Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In Proceedings of CP 2001, pages 565–569, 2001.

4. C. Bessière and J-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In CP96, Second International Conference on Principles and Practice of Constraint Programming, pages 61–75, Cambridge, MA, USA, 1996.
5. R.E. Bixby, W. Cook, A. Cox, and E.K. Lee. Parallel mixed integer programming. Technical Report Research Monograph CRPC-TR95554, Center for Research on Parallel Computation, 1995.
6. D. Brélaz. New methods to color the vertices of a graph. Communication of the ACM, (22):251–256, 1979.
7. J.-M. Gauthier and G. Ribiere. Experiments in mixed-integer linear programming using pseudo-costs. Mathematical Programming, (12):26–47, 1977.
8. C. Gomes. Complete randomized backtrack search (survey). In M. Milano, editor, Constraint and Integer Programming: Toward a Unified Methodology, pages 233–283. Kluwer, 2003.
9. C. Gomes and J.-C. Regin. Modelling alldi. matrix models in constraint programming. In Optimization days, Montreal, Canada, 2003.
10. C. Gomes and D. Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In Proceedings of the Computational Symposium on Graph Coloring and Extensions, 2002.
11. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, (14):263–313, 1980.
12. J. Linderoth and M. Savelsberg. A computational study of search strategies for mixed integer programming. INFORMS Journal on Computing, 11(2):173–187, 1999.
13. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. Information Science, 19:229-250, 1979.
14. M.W. Moskewicz, C.F. Madigan, and S. Malik. Chaff: Engineering an efficient SAT solver. In Design Automation Conference, 2001.
15. B. Smith. The Brelaz heuristic and optimal static ordering. In Proceedings of CP'99, pages 405–418, (Alexandria, VA), 1999.