Constraint Programming Lessons Learned from Crossword Puzzles

Adam Beacham, Xinguang Chen, Jonathan Sillito, and Peter van Beek

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1
{abeacham,xinguang,sillito,vanbeek}@cs.ualberta.ca

Abstract Constraint programming is a methodology for solving difficult combinatorial problems. In the methodology, one makes three design decisions: the constraint model, the search algorithm for solving the model, and the heuristic for guiding the search. Previous work has shown that the three design decisions can greatly influence the efficiency of a constraint programming approach. However, what has not been explicitly addressed in previous work is to what level, if any, the three design decisions can be made independently. In this paper we use crossword puzzle generation as a case study to examine this question. We draw the following general lessons from our study. First, that the three design decisions—model, algorithm, and heuristic—are mutually dependent. As a consequence, in order to solve a problem using constraint programming most efficiently, one must exhaustively explore the space of possible models, algorithms, and heuristics. Second, that if we do assume some form of independence when making our decisions, the resulting decisions can be sub-optimal by orders of magnitude.

1 Introduction

Constraint programming is a methodology for solving difficult combinatorial problems. A problem is modeled by specifying constraints on an acceptable solution, where a constraint is simply a relation among several unknowns or variables, each taking a value in a given domain. Such a model is often referred to as a constraint satisfaction problem or CSP model. A CSP model is solved by choosing or designing an algorithm that will search for an instantiation of the variables that satisfies the constraints, and choosing or designing a heuristic that will help guide the search for a solution.

As previous work has shown, the three design decisions—model, algorithm, and heuristic—can greatly influence the efficiency of a constraint programming approach. For example, Nadel [13] uses the n-queens problem to show that there are always alternative CSP models of a problem and that, given the naive chronological backtracking algorithm, these models can result in different problem solving performances. Ginsberg et al. [9] use one possible model of crossword puzzle generation (model m_2 in our notation) to study the effect on performance of various backtracking algorithms and variable ordering heuristics. Smith et al. [17]

use Golomb rulers to study the effect on performance of alternative models, algorithms and heuristics. Finally, a variety of studies have shown how the relative performance of various search methods and heuristics vary with properties of a random binary CSP instance such as domain sizes, tightness of the constraints, and sparseness of the underlying constraint graph (e.g., [2,6,8,10,18]).

However, what has not been explicitly addressed in previous work is to what level, if any, the three design decisions can be made independently. There are three possible levels of independence in the design decisions: complete independence, one-factor independence, and conditional independence (see, e.g., [4]). Suppose that there are four choices each for the model, the algorithm, and the heuristic. Complete independence would mean that choosing the best model, the best algorithm, and the best heuristic could all be done independently and a total of $4+4+4=12$ tests would need to be performed to choose the best overall combination. One-factor independence would mean that, while two of the design decisions might depend on each other, the third could be made independently and a total of $4+(4\times 4)=20$ tests would need to be performed to choose the best overall combination. Conditional independence would mean that two of the design decisions could be made independently, given the third design decision and a total of $4\times(4+4)=32$ tests would need to be performed to choose the best overall combination. Finally, if none of the independence conditions hold and the design decisions are mutually dependent a total of $4\times 4\times 4=64$ tests would need to be performed to choose the best overall combination. Thus, it is clear that the level of independence of the design decisions can have a large impact if we seek the best overall combination of decisions.

In this paper we use crossword puzzle generation as a case study to examine the interdependence of the choice of model, backtracking search algorithm, and variable ordering heuristic on the efficiency of a constraint programming approach. We perform an extensive empirical study, using seven models, eight backtracking algorithms, and three variable ordering heuristics for a total of 34 different combinations (not all algorithms can be applied on all models). The goal of our study is to examine to what extent the design decisions can be made independently. We draw the following general lessons from our study. First, that the three design decisions—model, algorithm, and heuristic—are mutually dependent. In other words, neither complete independence, one-factor independence, nor conditional independence hold. As a consequence, in order to solve a problem using constraint programming most efficiently, one must exhaustively explore the space of possible models, algorithms, and heuristics. Second, that if we do assume some form of independence when making our decisions, the resulting decisions can be sub-optimal. As one example, if the model is chosen independently of the algorithm and the heuristic, the result can be orders of magnitude less effective as the best algorithm and heuristic cannot overcome a poor choice of model.

2 CSP Models for Crossword Puzzles

In crossword puzzle generation, one is required to fill in a crossword puzzle grid with words from a pre-defined dictionary such that no word appears more than once. An example grid is shown in Figure 1. In this section, we present the seven different CSP models of the crossword puzzle problem that we used in our experiments. A constraint satisfaction problem (CSP) consists of a set of variables, a domain of possible values for each variable, and a collection of constraints. Each constraint is over some subset of the variables called the scheme of the constraint. The size of this set is known as the arity of the constraint.

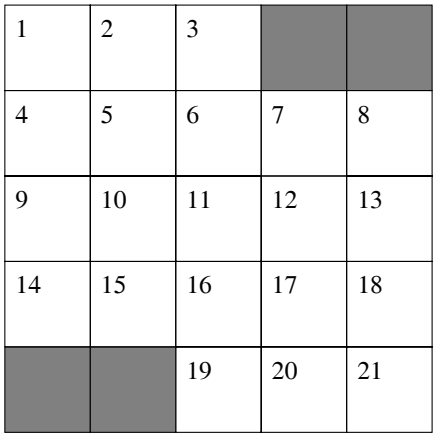

Figure 1. A crossword puzzle grid.

Model m_1. In model m_1 there is a variable for each unknown letter in the grid. Each variable takes a value from the domain $\{a, \ldots, z\}$. The constraints are of two types: word constraints and not-equals constraints. There is a word constraint over each maximally contiguous sequence of letters. A word constraint ensures that the sequence of letters forms a word that is in the dictionary. The arity of a word constraint reflects the length of the word that the constraint represents. For example, the word at "1 Across" in Figure 1 has three letters and will result in a 3-ary constraint over those letter variables. The tuples in the word constraints represent the words that are of the same length as the arity of the constraint in the pre-defined dictionary. There is a not-equals constraint over pairs of maximally contiguous sequences of letters of the same length. A not-equals constraint ensures that no word appears more than once in the puzzle. The arity of a not-equals constraint depends on whether the corresponding words intersect. For example, the words at "9 Across" and "3 Down" will result in a 9-ary constraint over those letter variables.

Model m_1^+. Model m_1^+ is model m_1 with additional, redundant constraints. One technique to improve the performance of the forward checking algorithm is to add redundant constraints in the form of projections of the existing constraints. (Projection constraints are not as effective for algorithms that maintain generalized arc consistency.) In model m_1^+, for each word constraint C and for each proper subset S of the variables in the scheme of C in which the variables are consecutive in the word, we add a redundant constraint which is the projection of C onto S. For example, for the constraint over the three letter variables x_1, x_2, and x_3 which form the word at "1 Across" in Figure 1, projection constraints would be added over x_1 and x_2, and over x_2 and x_3. The tuples in these constraints would represent the valid prefixes and valid suffixes, respectively, of the three-letter words in the dictionary.

Model m_2. In model m_2 there is a variable for each unknown word in the grid. Each variable takes a value from the set of words in the dictionary that are of the right length. The constraints are of two types: intersection constraints and not-equals constraints. There is an intersection constraint over a pair of distinct variables if their corresponding words intersect. An intersection constraint ensures that two words which intersect agree on their intersecting letter. There is a not-equals constraint over a pair of distinct variables if their corresponding words are of the same length. A not-equals constraint ensures that no word appears more than once in the puzzle. All of the constraints in model m_2 are binary. Although a natural model, model m_2 can be viewed as a transformation of model m_1 in which the constraints in m_1 become the variables in m_2. The transformation, known as the dual transformation in the literature, is general and can convert any non-binary model into a binary model [16].

Model m_3. In model m_3 there is a variable for each unknown letter in the grid and a variable for each unknown word in the grid. Each letter variable takes a value from the domain $\{a, \ldots, z\}$ and each word variable takes a value from the set of words in the dictionary that are of the right length. The constraints are of two types: intersection constraints and not-equals constraints. There is an intersection constraint over a letter variable and a word variable if the letter variable is part of the word. An intersection constraint ensures that the letter variable agrees with the corresponding character in the word variable. There is a not-equals constraint over a pair of distinct word variables if their corresponding words are of the same length. All of the constraints in model m_3 are binary. Model m_3 can be viewed as a transformation of model m_1 which retains the variables in the original problem plus a new set of variables which represent the constraints. The transformation, known as the hidden transformation in the literature, is general and can convert any non-binary model into a binary model [16].

A CSP problem can be encoded as a satisfiability (SAT) problem (see, e.g., [19]). To illustrate the encoding we use in this paper, consider the CSP with three variables x, y, and z, all with domains {a, b, c}, and constraints $x \neq y$, $x \neq z$, and $y \neq z$. In the SAT encoding a proposition is introduced for every variable in the CSP and every value in the domain of that variable: x_a, x_b, x_c, y_a,

y_b, y_c, and z_a, z_b, z_c. The intended meaning of the proposition x_a, for example, is that variable x is assigned the value a. Clauses (or constraints) are introduced to enforce that each variable must be assigned a *unique* value. For example, for the variable x, the following clauses are introduced: $x_a \vee x_b \vee x_c$, $x_a \Rightarrow \neg x_b$, $x_a \Rightarrow \neg x_c$, and $x_b \Rightarrow \neg x_c$. Clauses are introduced to specify the illegal tuples in the constraints. For example, for the constraint $x \neq y$, the following clauses are introduced: $x_a \Rightarrow \neg y_a$, $x_b \Rightarrow \neg y_b$, and $x_c \Rightarrow \neg y_c$.

Model s_1. Model s_1 is the SAT encoding of model m_1 with the following improvements designed to reduce the amount of space required. In the generic encoding, clauses are introduced to rule out the tuples that are not allowed by a constraint. For the word constraints in m_1, the illegal tuples represent sequences of letters that are not words in the dictionary. In s_1, not all illegal words are translated to clauses. Instead, we translate all invalid prefixes. For example, "aa" is not a valid prefix for words of length 4. Instead of recording all of the illegal tuples "aaaa" ... "aazz", just "aa" is recorded. For not-equals constraints in m_1, only the tuples that form a word are translated into clauses in s_1. For example, we do not say that "aaaa" $\neq$ "aaaa" because "aaaa" is not a valid word.

Model s_2. Model s_2 is the SAT encoding of model m_2. In particular, for each pair of propositions that represents intersecting words in the puzzle and words from the dictionary that do not agree, we introduce a negative binary clause ruling out the pair.

Model s_3. Model s_3 is the SAT encoding of model m_3 with the following improvements. The clauses for the domains of the hidden variables that ensure that each word must get a unique value are dropped as they are redundant (the clauses for the domains of the letter variables and the clauses for the intersection constraints together entail that a hidden variable can take only one value).

3 Backtracking Algorithms

In this section, we present the eight different backtracking algorithms we used in our experiments. At every stage of backtracking search, there is some current partial solution which the algorithm attempts to extend to a full solution by assigning a value to an uninstantiated variable. The idea behind some of the most successful backtracking algorithms is to look forward to the variables that have not yet been given values to determine whether a current partial solution can be extended towards a full solution. In this forward looking phase, the domains of the uninstantiated variables are filtered based on the values of the instantiated variables. The filtering, often called constraint propagation, is usually based on a consistency technique called arc consistency or on a truncated form of arc consistency called forward checking (e.g., [7,10]). If the domain of some variable is empty as a result of constraint propagation, the partial solution cannot be part of a full solution, and backtracking is initiated. A further improvement can be made to backtracking algorithms by improving the backward looking phase when the

algorithm reaches a dead-end and must backtrack or uninstantiate some of the variables. The idea is to analyze the reasons for the dead-end and to backtrack or backjump enough to prevent the conflict from reoccurring (e.g., [7,14]). All of the algorithms we implemented used some form of constraint propagation and all were augmented with conflict-directed backjumping [14].

Algorithm FC. Algorithm FC performs forward checking [10].

Algorithm GAC. Algorithm GAC performs generalized arc consistency propagation in the manner described in [12].

Algorithm EAC. Algorithm EAC performs generalized arc consistency propagation in the manner described in [3]. In the implementation of EAC used in the experiments the constraints were stored extensionally and advantage was taken of this fact to improve overall performance.

Algorithms PAC(m_1), PAC(m_2), and PAC(m_3). A technique for improving the efficiency of generic constraint propagation is to design special purpose propagators where constraints have methods attached to them for propagating the constraint if the domain of one of its variables changes (see, e.g., [1]). Propagators provide a principled way to integrate a model and an algorithm [15]. We designed and implemented propagators which enforce arc consistency for each of the constraints in models m_1, m_2, and m_3.

Algorithms ntab_back, ntab_back2. These algorithms are the implementations of the TABLEAU algorithm described in [5]. Algorithm ntab_back uses backjumping and algorithm ntab_back2 uses relevance bounded learning[1].

4 Experimental Results

We tested a total of 34 different combinations of seven models, eight backtracking algorithms, and three variable ordering heuristics (not all algorithms can be applied on all models; e.g., the algorithms based on TABLEAU are applicable only to SAT problems and each propagator-based algorithm is designed for a particular model). The three dynamic variable orderings heuristics used were the popular *dom+deg* heuristic [10] which chooses the next variable with the minimal domain size and breaks ties by choosing the variable with the maximum degree (the number of the constraints that constrain that variable, excluding the not-equals constraints), the *dom/deg* heuristic proposed by Bessière and Régin [2] which chooses the next variable with the minimal value of the domain size divided by its degree, and a variant of the MOM heuristic [5] which is geared to SAT problems and chooses the next variable with the Maximum Occurrences in clauses of Minimum size.

In the experiments we used a test suite of 50 crossword puzzle grids and two dictionaries for a total of 100 instances of the problem that ranged from easy to hard. For the grids, we used 10 instances at each of the following sizes: 5×5,

[1] Available at: http://www.cirl.uoregon.edu/crawford

Table 1. Effect of model, algorithm, and heuristic on number of instances (out of a total of 100) that could be solved given a limit of 228 Mb of memory and ten hours of CPU time per instance; (a) dom+deg heuristic; (b) dom/deg heuristic; (c) variant of the MOM heuristic. The absence of an entry means the combination was not tested.

	model			
algorithm	m_1	m_1^+	m_2	m_3
FC	20	59	61	48
GAC	20	10	50	83
EAC	89		0	0
PAC	88		80	84

(a)

	model			
algorithm	m_1	m_1^+	m_2	m_3
FC	20	50	63	55
GAC	20	10	50	81
EAC	92		0	0
PAC	91		85	84

(b)

	model		
algorithm	s_1	s_2	s_3
ntab_back	10	0	20
ntab_back2	11	0	20

(c)

15×15, 19×19, 21×21, and 23×23[2]. For the dictionaries, we used Version 1.5 of the UK cryptics dictionary[3], which collects about 220,000 words and in which the largest domain for a word variable contains about 30,000 values, and the Linux /usr/dict/words dictionary, which collects 45,000 words and in which the largest domain for a word variable has about 5,000 values. Although use of a smaller dictionary decreases the size of search space, the number of solutions also decreases and, in this case, made the problems harder to solve.

All the experiments except those for EAC were run on a 300 MHz Pentium II with 228 Megabytes of memory. The experiments on EAC were run on a 450 MHz Pentium II and the CPU times were converted to get approximately equivalent timings. A combination of model, algorithm, and variable ordering heuristic was applied to each of the 100 instances in the test suite. A limit of ten hours of CPU time and 228 Megabytes of memory was given in which to solve an instance. If a solution was not found within the resource limits, the execution of the backtracking algorithm was terminated.

Table 1 summarizes the number of instances solved by each combination. The low numbers of instances solved by the SAT-based models are due to both the time and the space resource limits being exceeded (as can be seen in Table 2, the SAT models are large even for small instances and storing them requires a lot of memory). The EAC algorithm also consumes large amounts of memory for its data structures and ran out of this resource before solving any instances from models m_2 and m_3. In all other cases, if an instance was not solved it was because the CPU time limit was exceeded.

Because the number of instances solved is a coarse measure, Figure 2 shows approximate cumulative frequency curves for some of the timing results. We can read from the curves the $0, \ldots, 100$ percentiles of the data sets (where the value of the median is the $50th$ percentile or the value of the $50th$ test). The curves

[2] The ten 5×5 puzzles are all of the *legal* puzzles of that size; the other puzzles were taken from the Herald Tribune Crosswords, Spring and Summer editions, 1999.

[3] Available at: http://www.bryson.demon.co.uk/wordlist.html

Table 2. Size of an instance of a model given a dictionary and the grid shown in Figure 1, where n is the number of variables, d is the maximum domain size, r is the maximum constraint arity, and m is the number of constraints.

model	dictionary	n	d	r	m
m_1	UK	21	26	10	23
m_1^+	UK	21	26	10	83
m_2	UK	10	10,935	2	34
m_3	UK	31	10,935	2	55
s_1	UK	546	2	26	1,336,044
s_2	UK	65,901	2	10,935	$\approx 8 \times 10^8$
s_3	UK	66,447	2	10,935	408,302
m_1	words	21	26	10	23
m_1^+	words	21	26	10	83
m_2	words	10	4,174	2	34
m_3	words	31	4,174	2	55
s_1	words	546	2	26	684,464
s_2	words	26,715	2	4,174	$\approx 2 \times 10^8$
s_3	words	27,261	2	4,174	168,339

are truncated at time = 36000 seconds (ten hours), as a backtracking search was aborted when this time limit was exceeded.

5 Analysis

In this section, we use the experimental results to show that the design decisions are not completely independent, one-factor independent, nor conditionally independent. Hence, the design decisions are mutually dependent.

Complete independence. For the choice of the best model, algorithm, and heuristic to be completely independent decisions *all* of the following must hold: (i) the ordering of the models, such as by number of problems solved, must be roughly invariant for all algorithms and heuristics, (ii) the ordering of the algorithms must be roughly invariant for all models and heuristics, and (iii) the ordering of the heuristics must be roughly invariant for all models and algorithms. However, none of these conditions hold. For (i), consider the different orderings of the models given by FC and GAC using the dom+deg heuristic in Table 1; for (ii), consider the relative orderings of the FC and GAC algorithms given by the different models; and for (iii), consider the reversed orderings of the heuristics given by FC on m_1^+ and m_3.

One-factor independence. One-factor independence can occur in three ways, corresponding to the three conditions given under complete independence. As shown there, none of these conditions hold.

Conditional independence. Conditional independence of the decisions can occur in three ways: (i) the choice of the best algorithm and heuristic can be

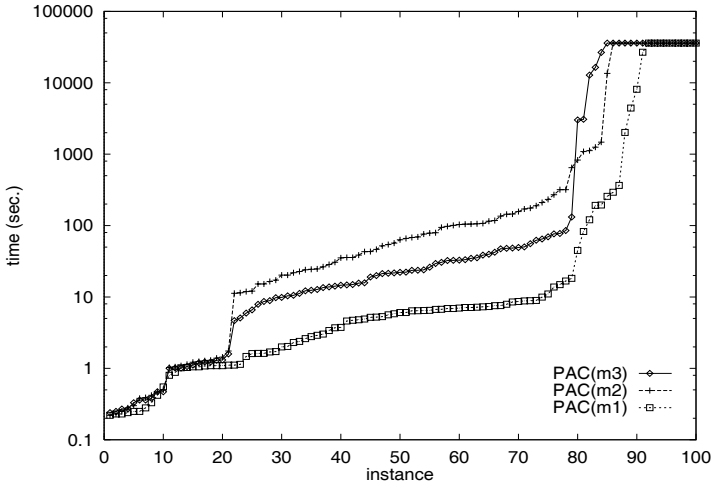

Figure 2. Effect of model on time (sec.) of backtracking algorithms, given the dom/deg dynamic variable ordering heuristic. Each curve represents the result of applying the given backtracking algorithm to the 100 instances in the test suite, where the instances are ordered by time taken to solve it (or to timeout at 36,000 seconds).

independent decisions, given a choice of model (i.e., given a model, the ordering of the algorithms is roughly invariant for all heuristics, and, by symmetry, the ordering of the heuristics is roughly invariant for all algorithms); (ii) the choice of the best model and heuristic can be independent decisions, given a choice of algorithm (i.e., given an algorithm, the ordering of the models is roughly invariant for all heuristics, and the ordering of the heuristics is roughly invariant for all models); and (iii) the choice of the best model and algorithm can be independent decisions, given a choice of heuristic (i.e., given a heuristic, the ordering of the models is roughly invariant for all algorithms, and the ordering of the algorithms is roughly invariant for all models). For (i), suppose the model given is m_3 and consider the reversed orderings of the heuristics given by FC and GAC; for (ii), suppose the algorithm given is FC and consider the reversed orderings of the heuristics given by m_1^+ and m_3; and for (iii), suppose the heuristic given is dom+deg and consider the different orderings of the models given by FC and GAC.

We can also see from the data the importance of choosing or formulating a good model of a problem. In Table 1 we see that the best algorithm or set of algorithms cannot overcome a poor model and compiling a CSP to an instance of SAT in order to take advantage of progress in algorithm design (cf. [11]) can be a disastrous approach. In Figure 2 we see that even when our models are all relatively good models (such as m_1, m_2, and m_3), and much effort is put into correspondingly good algorithms, the form of the model can have a large effect—ranging from one order of magnitude on the instances of intermediate difficulty to two and three orders of magnitude on harder instances.

6 Conclusions

We draw the following three general lessons from our study: (i) the form of the CSP model is important; (ii) the choices of model, algorithm, and heuristic are interdependent and making these choices sequentially or assuming a level of independence can lead to non-optimal choices; and (iii) to solve a problem using constraint programming most efficiently, one must simultaneously explore the space of models, algorithms, and heuristics.

References

1. P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proc. of IJCAI-95*, pp. 600–606.
2. C. Bessière and J.-C. Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. of CP-96*, pp. 61–75.
3. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proc. of IJCAI-97*, pp. 398–404.
4. P. R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, 1995.
5. J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artif. Intell.*, 81:31–57, 1996.
6. D. Frost and R. Dechter. In search of the best search: An empirical evaluation. In *Proc. of AAAI-94*, pp. 301–306.
7. J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. of the 2nd Canadian Conf. on AI*, pp. 268–277, 1978.
8. I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. of CP-96*, pp. 179–193.
9. M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search lessons learned from crossword puzzles. In *Proc. of AAAI-90*, pp. 210–215.
10. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14:263–313, 1980.
11. H. Kautz and B. Selman. Planning as satisfiability. In *Proc. of ECAI-92*, pp. 359–363.
12. A. K. Mackworth. On reading sketch maps. In *Proc. of IJCAI-77*, pp. 598–606.
13. B. A. Nadel. Representation selection for constraint satisfaction: A case study using n-queens. *IEEE Expert*, 5:16–23, 1990.
14. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intell.*, 9:268–299, 1993.
15. J.-F. Puget and M. Leconte. Beyond the glass box: Constraints as objects. In *Proc. of ISLP-95*, pp. 513–527.
16. F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. of ECAI-90*, pp. 550-556.
17. B. Smith, K. Stergiou, and T. Walsh. Using auxilliary variables and implied constraints to model non-binary problems. In *Proc. of AAAI-00*.
18. E. P. K. Tsang, J. E. Borrett, and A. C. M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Proc. of the AI and Simulated Behaviour Conf.*, pp. 203–216, 1995.
19. T. Walsh. SAT v CSP. In *Proc. of CP-00*.