CS 227 Programming Assignment 2: Constraint Satisfaction Problems

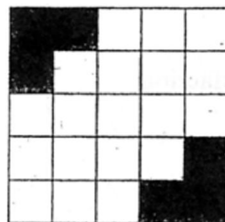
Todd Sullivan
todd.sullivan@cs.stanford.edu

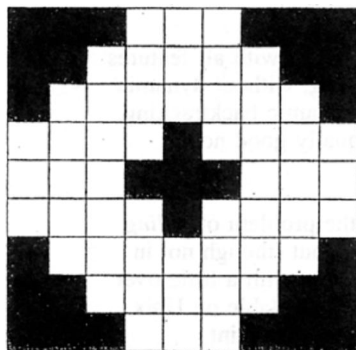
Harry Robertson
harry.robertson@gmail.com

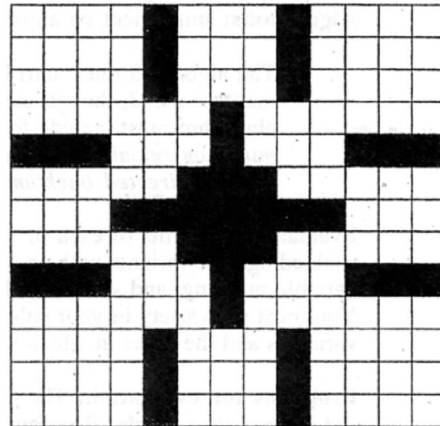
Pavani Vantimitta
pavani@stanford.edu

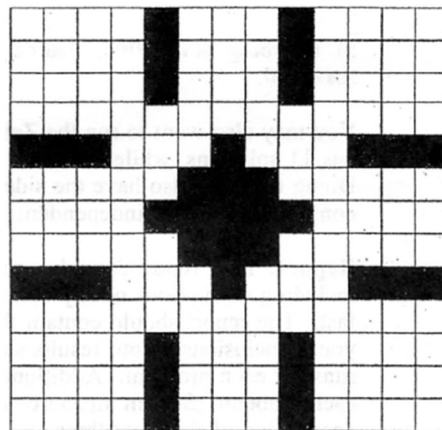
1 Introduction

Crossword puzzles, in addition to being a hobby of potentially millions of people around the world, are an excellent test bed for evaluation of constraint satisfaction algorithms. A solution to a crossword puzzle is an assignment of words to a grid such that each word meets a number of constraints: a semantic constraint provided by the clue (which we do not take into account here), a length constraint provided by the grid, and constraints on its letters provided by the words which it overlaps in the grid. Below are the four crossword puzzles that we used as test problems in this assignment for our algorithms.


Puzzle 1


Puzzle 2


Puzzle 3


Puzzle 4

There are two approaches one can take to model these crossword problems as CSPs: a word based approach and a character based approach. In the character based approach, we consider the set of white squares to be our set of variables. Thus the domain of each square is the set of permissible characters (94 ASCI characters in this case). The requirement that each vertical or horizontal alignment of squares must constitute a dictionary word forms the constraint set (thus we have non-binary constraints).

In the word-based approach, we consider each word (i.e. a horizontal or vertical alignment of white squares) in the crossword puzzle to be a variable. Thus the domain for each variable is the set of dictionary words of the corresponding size. The constraints are the overlap conditions imposed by the grid: whenever two words overlap, they must share the same letter in the overlap position (thus the constraints are all binary, although one can also consider the requirement that each word value be in the dictionary to form a unary constraint).

There is clearly a trade-off between the two approaches: with the character-based approach, the variables have very limited domains (94 possible values, as opposed to tens of thousands of potential words in the word-based approach), while the word-based approach has relatively simple binary constraints.

However, the primary reason against choosing the word-based approach, the large domain size, can be mitigated by developing fast and efficient word-lookup methods, which are provided by the bit-set dictionary system explained in Section 3. We were further swayed towards the word-based approach by Ginsberg's paper on the subject [1].

In this paper, we begin by describing the algorithms we used to solve the CSPs, then the optimizations we use to take advantage of the structure of crossword puzzles. We then describe our results, the conclusions we drew and the reasons. Finally we conclude and give ideas for future work.

2 Algorithms

All the algorithms we implemented have three main common functions – label, unlabel, chooseNextVariable.

2.1 Backtracking

These algorithms check backwards, from the current variable against the past variable. It is a depth-first search technique that chooses values for one variable at a time, and backtracks when a variable has no legal values in its domain left to assign.

2.1.1 Chronological Backtracking

This technique withdraws the most recently made choice, the consequences of the recent choice, selects a different alternative at that point and moves ahead again. If the point backtracked to has been explored previously, it withdraws further until an unexplored alternative comes up. In label we loop through the current domain of the variable i being assigned a value and check if it is consistent with the other variables with which i has constraints. In the case that the value is not consistent it just picks the next value in the domain of i and runs checks again. In unlabel we backtrack to the previous level and remove the assigned value for i from its domain and also unassign i. chooseNextVariable chooses the next variable by a static ordering, chronologically.

The problem that is apparent with this technique is that many of the points it backtracks to have nothing to do with the dead-end it had encountered and thus is inefficient.

2.2 Back jumping

One way of doing informed backtracking is called back-jumping. This approach goes back directly to the point (or variable) which caused the dead-end or failure to the most recent assignment.

2.2.1 Conflict directed back jumping

This is a simple variant of back jumping which uses a set of variables, called conflict set, which consists of all the variables that have constraints with the variable being assigned a value. This approach backtracks to the most recent variable in this set. This is implemented by accumulating this set while checking for domain values that satisfy a constraint for a variable and when a domain wipe out happens, the conflict set is used. In label

we try to assign a value to variable i from its domain that is consistent with all the other variables it has constraints with. In case there is a conflict with any variable, we add it to the conflict set and we remove that value from i's domain and move through the other values in the domain. In unlabel we pick the deepest level from the conflict set and unassign the variables whose values have been set from that deepest level to the current level and also reset their domains. chooseNextVariable goes chronologically.

2.3 Forward Checking

Forward checking makes additional use of constraints by looking at each unassigned variable connected to a variable by a constraint when assigning it a value, in order to predict future domain wipe-outs. In forward checking, for each variable i we keep three stacks, reductions_i, pastfc_i, and future-fc_i. The reductions stack for a variable i, reductions_i, keeps elements that are lists of values in the domain of i disallowed by previously instantiated variables. That is, a variable instantiated earlier in the search than i removed some set of values from i's domain, and these are kept on the reductions_i stack. Past-fc_i is a stack of the levels in the search that have checked against variable i, while future-fc_i is the stack of future variables against which i has checked.

In forward checking's labeling step, we iterate over all the variables that have not yet been assigned, checking the currently instantiated variable against them. After attempting to instantiate a variable i, for every variable j that remains unassigned we remove the possible values for j that do not meet the constraints between i and j. For every j, we add to reductions_j all values from j's domain that has been eliminated by constraints with xi, and we push variable j onto future-fc_i, since j comes after xi in the search. We also push the current level onto pastfc_j. In the unlabelling step, we move to the previous level update the domains back to undo the changes, undo the reductions, and unassign the variables previously assigned

If any of the unassigned variable's domains has a value that is inconsistent with the assignment it deletes the value from that variables domain. We can thus see that this procedure makes it easy for collecting the conflict set elements. Whenever forward checking based on an assignment for variable i deletes a value from another variable j, it should just add i to j's conflict set. During this process if variable j's domain empties out, then j goes into the conflict set of i. It can also be said that forward checking is better than back jumping because it draws the final conclusion faster than back jumping, saving some redundant iterations.

2.4 Dynamic Variable Ordering

The order in which variables are assigned values can be of two types: static and dynamic. Static ordering is when we always know that variable i will be assigned a value before variable j. In Dynamic Variable Ordering (DVO) the search process determines which variable is to be assigned based on the state of the search process. Static variable ordering is what we use when we do not use dynamic ordering. This ordering is said to have a very high role in the efficiency of constraint satisfaction problem solutions. The heuristic we use for DVO is called minimum remaining values (MRV).

2.4.1 Minimum Remaining Values

The first variable is assigned without DVO and then onwards the MRV function is called. This technique checks each value in the domain of each unassigned variable to see if that value is consistent with the set of assignments made so far. It then keeps a count of the remaining values in each variable's domain that is compatible with the current set of assignments and returns the variable with the minimum of this count.

2.5 Arc Consistency-3

One common technique used to increase the efficiency of these algorithms is to prune the search space prior to processing these techniques. This is done by a technique called Constraint propagation, propagating the implications of a constraint on one variable onto other variables. This is to reduce the amount of search time afterwards. The term Arc refers to the directed constraint from one variable to another variable and thus shrinks the search space by removing the un-supporting values from variable's domains, which corresponds to an early detecting of inconsistencies. A variable x's value is unsupported if there is a constraint on the variable such that there are no instantiations of other variables that satisfy the constraint when x is set to v.

AC-3 is one such arc consistency algorithms and works in two steps. It first iterates over every variable x, checking whether its values are supported in each constraint over x and simultaneously it queues the unsupported values it has removed. It then dequeues the previously removed unsupported values, checking whether the removals have caused more values to become unsupported. Any newly unsupported values it places on the queue and continues dequeueing until it gets to the point where the queue is empty. This happens when no more values become unsupported, and we are guaranteed termination because there only finitely many values that can be removed.

3 Optimizations

Caching and other optimizations are essential for creating usable CSP solvers. Like our previous project's SAT solvers, we developed our CSP algorithms with efficiency in mind. Our optimizations include managing our own memory for most data structures (no STL Vectors), not allowing duplicate arcs in our AC3 queue, indexing character occurrence in domains using bitsets, and several other memory allocation tweaks.

3.1 Memory Management

Keeping with the spirit of our efficient SAT solvers, we again do not use the C++ Standard Template Library (STL) Vector class to store our data. Instead, we use our own data structure called superArray, which is a templated class that simply contains an integer called length and a pointer to an array of whichever data type the superArray stores. The class contains two functions, allocate for allocating a set size and wipe for deleting the dynamically allocated memory. Items in the array are accessed directly using the pointer.

We do not use the Vector class because of the overhead associated with its resizable array properties and its out-of-bounds checking. While these features are important in many settings, they are only barriers in our quest for speed and efficiency. In our previous report about our SAT solvers, we presented a simple experiment that shows how our data structure is much faster than using the Vector class when performing reads and writes. One lacking element of our experiment was that it involved repeatedly reading and writing to the arrays and did not include any attempt to measure the speedup in our application domain. We expand upon our previous experiment here and present the effects of using our own data structure on crossword CSP solving.

The Vector class contains two methods for accessing elements in the internal array. The first method, which we will refer to as Op, is through the overloaded [ ] operator. Op does not perform out-of-bounds checking. The second method, which we will refer to as At, is through the at() function. As one would guess, At is different from Op in that it performs out-of-bounds checking.

First, we ran a modified version of our original experiment, which is located in TestVector.h within our source code, on the machines described in Section 4 of this report (Dell Precision 390 with a 2.4 GHz Core 2 Duo and 2 GB of RAM running Ubuntu). The experiment performed 4 billion reads and 4

billion writes using each method (Op, At, and our superArray), with all memory allocated before beginning the test. We executed the test ten times and measured CPU processing time instead of real time. Table 3.1.a shows, as expected, that our superArray direct access method was about 1.5 times faster than Op on reads/writes and 5.5 times faster than At.

Table 3.1.a: TestVector.h Computation Time Relative to superArray

	Reads	Writes
superArray	1.0	1.0
Vector Op	1.4	1.7
Vector At	5.5	5.5

In an effort to prove that using our superArray has a measureable impact on the performance of our CSP algorithms, we included a preprocessor define statement VECTOR_TEST in data_structures.h. When VECTOR_TEST is set to false, all of our algorithms use our superArray direct access method. When VECTOR_TEST is set to true, our algorithms use a Vector class in place of the pointer within superArray and access elements using the Op method. Thus we have a compile-time switch that allows us to specify which data structure our algorithms use.

Regardless of which data structure is used, we still perform all of the memory allocation tweaks and other optimizations described later in this section. We ran our CSP solver with all features turned on (AC3, forward checking, dynamic variable ordering with the minimum remaining values heuristic, and conflict-directed backjumping). We measured CPU time and executed the experiment on the same machines as before, averaging over ten executions. We timed the two versions of our solver on the puzzles contained in the file "puzzlesWithHardStuff", which is included in our source. This puzzle file contains the four puzzles described in Section 1, as well as a 26-by-26 cell puzzle that was created by taking four copies of Puzzle 3 and arranging them to form a square and a 37-by-37 cell puzzle that was created by taking nine copies of Puzzle 3, arranging them in a similar manner, and removing the right two columns and bottom two rows.

Table 3.1.b presents the results of this real-world experiment. Our superArray method was 1.02 times faster than the Vector Op method. While this is certainly not a drastic improvement, it is a measurable improvement nonetheless. Additionally, while Vector provides automatic resizing functionality that superArray does not include, both methods do not perform out-of-bounds checking. Since we do not require array resizing, and in fact we pre-allocate all memory at the beginning as described in Seciton 3.4, there is no reason to use the Vector Op method when our superArray gives a slight performance boost.

Table 3.1.b: Crossword CSP Computation Time Relative to superArray

superArray	1.0
Vector Op	1.02
Vector At	1.18*

*Estimated using relative read / write times from Table 3.1.a.

Since we do not require array resizing, the only advantage of using Vector is the out-of-bounds checking in At. Using the relative read/write computation times from Table 3.1.a and the result of our real-world experiment, we estimate that our superArray method is 1.18 times faster than using the Vector At method. While out-of-bounds checking can help decrease the amount of hard to find bugs, we do not believe the potential development time benefit makes up for the performance decrease. (We just love memory corruption bugs.) We unfortunately cannot use the VECTOR_TEST switch to obtain exact results of using Vector At because the switch took advantage of the fact that the superArray and Vector Op methods both use brackets [ ] to access the array members.

3.2 Faster AC3

Traditional AC3 uses a queue to maintain the arcs that need to be processed. When running the algorithm on a small testing puzzle as it is traditionally defined, we found instances where the algorithm would process the same arc back-to-back or have the same arc in the queue multiple times (but separated by other arcs). Both of these cases resulted in redundant processing that reduced the speed of the algorithm.

To solve this problem, we created a second version of AC3 that uses a set, which does not allow duplicate arcs, instead of a queue. This modification was relatively easy and only required changing four lines of code. Our hope was that the performance gains from not processing arcs multiple times would be greater than the computation time required to maintain that set's unique item property (no duplicate arcs). Our experiments running both versions of AC3 on multiple puzzles shows that this was indeed the case.

We ran both versions of AC3 on three different puzzle collections. Collection 1, contained in the file "puzzlesAC3," consisted of 80 copies of Puzzle 4. Collection 2, contained in the file "puzzlesAC3_blank," consisted of 80 copies a blank 13-by-13 puzzle (no black squares). Collection 3, contained in the file "puzzlesAC3_hard," consisted of 40 copies of the 37-by-37 puzzle that is described in the previous section. We conducted the experiments on the same machines as our other tests. Aside from measuring CPU time, we also included checks at the end to make sure that the result of both versions were identical.

Table 3.2 shows the results of these experiments. In all cases, the output of AC3 was identical with both versions. Using a set is 1.5 to 2.2 times faster than using a queue. We initially believed that we would find greater speedup as we increased the number of constraints within the puzzle because the set version would eliminate the processing more duplicate arcs, which was our motivation for creating Collections 2 and 3 after running the experiment on Collection 1.

Our results show that this is not necessarily the case. While moving from Collection 1 to Collection 2 follows our hypothesized trend of greater speedup, moving from Collection 2 with 338 constraints to Collection 3 with 1,972 constraints actually reduces the speedup. Despite this result, we still believe that the relative trend of greater speedup as the constraint count increases.

Table 3.2: AC3 with a Set vs. AC3 with a Queue

Collection	Constraints Per Puzzle	Set CPU Time Per Puzzle	Set Speedup
1	264	0.06 seconds	1.54
2	338	0.16 seconds	2.23
3	1972	0.53 seconds	1.65

3.2.1 A Revised Hypothesis

We believe that the difference in speedup between Collections 2 and 3 is because of the different distribution of constraints that each puzzle contains. Collection 2 contains no black squares, so every word has a constraint on every letter. Collection 3 has many black squares that are arranged in a similar manner to Collection 1. Collections 1 and 3, which have similar distributions for the constraints on each variable, show the trend of the AC3 set version's speedup increasing as the amount of constraints increases.

We created additional puzzle collections to test our revised hypothesis that, when comparing processing time on puzzles that share the same distribution of constraints on variables, an increase in the number of constraints will result in a greater speedup from using AC3 with sets over AC3 with queues. We ran the previous experiment on 6 different puzzle sets,

which we will ball Blank 13 through Blank 18. Each puzzle set contained no black squares and thus every word has a constraint on every letter. Each set contains 80 puzzles. The number in each set's name indicates the size of the puzzles. Therefore Blank 13 contains 80 13-by-13 blank puzzles. Note that Blank 13 is the same as Collection 3 in the previous experiment. Blank 14 through Blank 18 are contained in "puzzlesAC3_blank14" through "puzzlesAC3_blank18" respectively.

Table 3.2.1 shows the results of running our new experiment on the same machines as before. Our hypothesized trend appears to hold true for Blank 15 through 18, but the set speedup for Blank 14 is more than twice the speedup of any other puzzle set. While the results do not prove our hypothesis, we still have an inkling that our revised hypothesis is true.

The primary problem with this latest experiment is that we are not accounting for the fact that the domains of the variables completely change from one puzzle size to the next. In Blank 13, all variables are words of length 13 while in Blank 17 all variables are words of length 17. Thus the variable domains in the Blank 13 puzzle are not of the same size as the domains in the Blank 17 puzzle. Additionally, the actual distribution of possible characters at each location in each word is different across the puzzle sets. To solve these problems one would need to create a normalized dictionary that, for example, contains the same probability of an 'A' occurring as the first letter of a word regardless of word length. Unfortunately, we did not have enough time to pursue this extension.

3.2.2 Faster AC3: Does It Matter?

While using a set instead of a queue can make AC3 up to 4.8 times faster, the fact remains that in our crossword puzzle setting AC3 takes less than one second to complete. Indeed, the pre-processing arc-consistency step is generally very fast relative to over-all algorithm run-time. Thus, by Amdahl's law, any speed improvements to AC3 are limited to this small fraction of over-all system performance.

However, as we discuss in our experimental results below, our fastest solvers frequently solve crossword puzzles in less than one second. Therefore in this setting, our reduction in AC3 computation time is significant in that it allows AC3 to actually be useful in some cases.

Table 3.2.1: AC3 with a Set vs. AC3 with a Queue

Blank	Constraints Per Puzzle	Set CPU Time Per Puzzle (sec)	Set Speedup
13	338	0.16	2.23
14	392	0.06	4.83
15	450	0.03	1.76
16	512	0.03	1.88
17	578	0.04	1.89
18	648	0.04	1.89

3.3 Managing Domains

Our most important optimization is our managing of each variable's domain. In all of our experiments, we use a dictionary of roughly 20,000 words. Each word in our dictionary can contain ASCII characters ranging from 33 (!) to 126 (~), with all lowercase letters mapped to their corresponding uppercase letters. This results in 94 possible characters. Despite uppercasing all letters, we left our management system treating lowercase letters as a possibility so that we could turn off the uppercasing policy if we desired. Including lowercase letters as a possibility without actually using them only slightly increases memory usage and does not increase processing time except for when reading in the dictionary. Due to the amount of words that could possibly be in a variable's domain, if we were to maintain each variable's domain as an array of strings, we would end up spending much of our time managing these arrays rather than solving the problem.

3.3.1 A Motivating Example

Suppose we have a variable V that is a 5-letter word with constraints on its first and fourth letters. Our dictionary contains 3,169 5-letter words, so V's initial domain size is a little over 3,000. Now suppose need to apply V's two constraints, with 'A' as the first letter and 'E' as the fourth. Without considering how we will store this restricted domain, to apply the constraints we must loop through all 3,169 words and perform a check against their first and fourth letters.

Now suppose V's domain was reduced to zero, so we back track and then return to choosing a value for V given the new constraints that the first letter is 'A' and the fourth letter is 'L'. Again, to calculate the restricted domain we must loop through all 3,169 words and perform a check against their first and fourth letters.

As one might notice, this management scheme is incredibly slow because we are performing checks against every 5-letter word during every restriction. The commonality between the previous two restrictions is that in both cases we required the first letter to be 'A'. If we stored the restricted domain for the first letter being 'A', then to apply the second constraint on the fourth letter we could simply loop through the stored restricted domain and perform one letter check instead of looping over all 3,169 words and performing two letter checks.

3.3.2 Our Solution

This storing of restricted domains, which is proposed in the Ginsberg 1990 paper mentioned in the project handout, is the idea behind our domain management scheme. When reading in the dictionary, for each word length L we create a 94-by-L array of bitsets with indexing starting at zero. Each bitset contains one bit for each word of length L. If a word's bit is equal to one then the word exists in the domain that the bitset represents.

The index of the 94 rows indicate which character the row of bitsets belongs to, with '!' being at index 0 and '~' being at index 93. The index into the L columns indicates the position in the L-letter word. The bitset at row A and column B represents the domain for an L-letter word that contains the sole restriction of having the character indicated by A at location B in the word. In our original implementation all variables of the same word length share the same array of bitsets, but, as we will discuss in Section 4, to satisfy the project's requirements of randomization our current implementation maintains an separate array of bitsets for each variable.

Thus for our variable V, we have a 94-by-5 array of bitsets with each bitset containing 3,169 bits. In our example where V requires the first letter to be 'A' and the fourth to be 'E', we simply have to perform a bitwise AND operation on the bitset at row 32 and column 0 and the bitset at row 36 and column 4. The first bitset contains ones for all 5-letter words that contain 'A' as the first letter. The second bitset contains ones for all 5-letter words that contain 'E' as the fourth letter. Thus the bitwise AND operation will result in an intersection of the two sets and will contain ones for all 5-letter words that contain 'A' as the first letter and 'E' as the second letter.

3.3.3 Empirical Comparison

These operations on sets of bits are significantly faster than looping through all of the words of the correct length and performing checks on specific characters. To show this, we conducted a simple test of repeatedly computing the restrictions in our previous example with variable V. We were not able to compare the relative performance of both methods on the actual crossword puzzle solving problem because, as we will discuss in Section 3.3.4, the bitset method requires changing the structure of the CSP algorithms to specifically take advantage of the bitwise operations.

In this simple experiment, which is the Dictionary::timeTest function in our code, we repeatedly restricted the domain of a 5-letter word V by the constraints that the first letter is

'A' and the fourth letter is 'E'. Instead of actually creating the list of words that satisfy the restrictions, we only computed the amount of words that satisfy the restriction. Thus for the non-bitset method we loop through all 3,169 words, perform two equality checks (one for 'A' on the first letter, the other for 'E' on the fourth letter), and increment a counter if both checks return true. For the bitset method we perform a bitwise AND operation on the two bitsets as described in the previous section.

We performed the restriction 5 million times and calculated per restriction processing times by dividing the total time by 5 million. We measured CPU time and used the same machines as the previous experiments. We excluded the time required to generate the arrays of bitsets for each word length from the bitset method's recorded time, but this computation time is included in the analysis.

Table 3.3.3 shows the results of our experiment. The bitset method is an order of magnitude faster than the non-bitset method, but it incurs a preprocessing penalty of 0.03 seconds to create all of the bitsets when reading in the dictionary. Since the bitset method is an order of magnitude faster, one only needs to perform 671 restrictions in order to break even due to the 0.03 second preprocessing cost. As our results in Section 5 will show, we perform far more than 671 restrictions (in fact, our algorithms perform millions or billions of restrictions). Thus the bitset method is essential to fast crossword puzzle CSP solving. The bitset method also incurs an additional cost of space, but when variables whose words are of equal length share the same bitset arrays the space cost is only 2.7 MB.

Another important positive point of our simple experiment is that in the bitset method, in order to calculate the amount of words that satisfy the restriction, we actually compute the list containing the restricted domain and then count the amount of bits set to one. Thus in our experiment the bitset method is both computing the restricted domain and the size of the restricted domain, while the non-bitset method is only computing the size and would have to populate some other data structure to generate the restricted domain.

Table 3.3.3: Comparison of Domain Management Methods

Bitset CPU Time Per Restriction	5.3×10^{-6} sec.
Non-bitset CPU Time Per Restriction	5.0×10^{-5} sec.
CPU Time to Generate All Bitsets	0.03 sec.
Restrictions Required to Break Even	**671 restrictions**
Memory Required to Hold Bitsets	2.7 MB

3.3.4 CSP Algorithm Adaptation

Due to our domain-specific approach for managing domains, we had to restructure the standard vanilla CSP algorithms presented in Section 2. In this section we describe how we modified several of the standard algorithms to use our domain management scheme.

3.3.4.1 Chronological Backtracking

The standard routine for labeling a variable V is to loop through each value in V's current domain and check if the value violates a constraint with any of the previously assigned variables. If the value does not violate any constraint then that value is chosen as V's assigned value and the search moves on trying to assign values to other variables.

Unfortunately, we cannot follow this procedure when using our bitsets because we use the AND operation to compute the entire restriction on a domain in one swoop. Thus our standard label function, which is CSPSolver::label() in our source, is slightly different. First, we apply all restrictions that exist due to any of the previously assigned variables having a constraint with V. We then pick the word that corresponds to the first one bit in the resulting bitset. If we for some rea-

son or another return to labeling V because the first value did not work, then we find the next one bit in the bitset, pick its corresponding word, and do not apply the restrictions again. If all of V's values failed to work, then we restore V's domain in the unlabel function and the next time we try to label V we compute the new restricted domain and start over.

3.3.4.2 Conflict-Directed Backjumping

The standard label routine of conflict-directed backjumping contains a slight modification to the chronological backtracking routine. Whenever one of V's values violates a constraint with some previously assigned variable, we add the previously assigned variable to V's conflict set. We handle this slight modification, which is CSPSolver_CBJ::label() in our source, by calculating the restricted domain with one constraint restriction at a time. After applying the restriction induced by a constraint, we check whether the size of the domain has changed. If the size decreased then that means that there were values in V's domain that violated the constraint, so we add the previously assigned variable that corresponds to the constraint to V's conflict set.

3.3.4.3 Forward Checking

Forward checking involves computing and storing reductions, which are lists of values that were removed from a domain. The bitset method for computing reductions is quite elegant. We simply store the current domain's bitset, restrict the current domain by the given restriction imposed by a constraint, and then perform a bitwise XOR operation on the old and new current domains to generate the list of values that were removed. We can then restore the reduction by performing a bitwise OR operation between the current domain and the reduction.

3.3.4.4 AC3

AC3 includes a function that takes in two variables V and W and removes inconsistent values from the domain of V based on the constraint between V and W. The routine loops through all values in V's domain and checks if W's domain contains a value that allows the pair of values to satisfy the constraint between V and W. The function removes the value from V's domain if no such pair exists.

In our routine, we start by storing the location of the constraint for each variable with loc_V holding the restricted location in V's word and loc_W holding the restricted location in W's word. Then for each character C of the 94 valid ASCII characters, we restrict W's domain with the restriction that the letter at loc_W is C. If the resulting domain is empty then we need to remove all words from V's domain that contain C at loc_V. To do this we store V's domain in a temporary bitset Dtemp, restrict V's domain by C at loc_V, perform a NOT operation on the restricted bitset, and then perform an AND operation on Dtemp and the result of the NOT operation, storing the result as V's base domain. The only other key detail is that one must remember to restore the domains to their base domains before proceeding to the next character C.

3.4 Memory Allocation and Quick Insertions

As we noticed when creating our SAT solvers, memory allocation can be costly. To reduce the amount of memory allocation required during the search, we allocate all data structures before starting the label/unlabel loop. While this step is obviously required for the domain caching structure, it is less obvious for structures such as the list of variables with the best score in our dynamic variable ordering method. We allocate a superArray for this list at the beginning of the label/unlabel loop, making its size the same as the number of variables in the CSP. Instead of using the superArray's length attribute to indicate the

amount of space allocated in the array, we use it to indicate the amount of variables that are actually in the list. Thus whenever we choose the next variable to label we do not need to allocate memory for the list and instead set the length to zero and start filling the list. We also pre-allocate our confSets and other structures to their potential maximum size before entering the label/unlabel loop so that we never have to reallocate memory due to resizing requirements. Additionally, we make every attempt to use pointers and not invoke a copy constructor anywhere in each solver's code.

3.5 Restart Timeouts

We found that many of our algorithms were susceptible to bad luck on their initial seed values for the random number generator. As discussed in Section 4.3, we randomize our choices for equally good variables. We also randomize the order that values in each variable's domain are considered by shuffling the words in each variable's domain when reading the dictionary and puzzles from their respective files, as described in Section 4.3. We found that the ordering of the domains had significant implications on runtime. For example, with all of our features turned on (AC3, conflict directed back jumping, forward checking, and dynamic variable ordering with the minimum remaining values heuristic), our solver could often solve Puzzle 4 in around 0.28 seconds, but with unlucky seed values the solver would require at least several minutes to find a solution.

An obvious insight from these observations is that periodically restarting with a reshuffled ordering for each domain could drastically improve overall performance. To fix this problem, we implemented an increasing restart timer that allows the algorithm additional time between restarts as the number of restarts increases. The initial execution length and the multiplier that increases the length of time allowed for the next run can be specified on a per solver basis, but we did not have sufficient time to run experiments with the various algorithms to find optimal values for each solver.

For our solvers using forward checking, we set the execution length of the first run to 0.75 seconds and the multiplier to 1.1. Thus the first run is allowed to execute for 0.75 seconds, the second run for 0.825, and the third run for 0.91 seconds. For our other solves, we set the execution length of the first run to 60 seconds and the multiplier to 2. Additional discussion on randomization and how we ensure deterministic behavior (for repeatability of experiments) given the same initial seed value is included in Section 4.3.

4 Experimental Method

We conducted all of our experiments on the Pod cluster. The Pod cluster contains Dell Precision 390s, each with a 2.4 GHz Core 2 Duo and 2 GB of RAM running Ubuntu 7.04. All results in Section 5 are from running each solver on the four crossword puzzles using 15 different seeds for the random number generator. We decided on the fixed 15 specific seed values to use by seeding the random number generator with the clock's time and then printing out 15 random numbers. The readme included with the source describes how to specify your own seed or which of our 15 seeds to use.

4.1 Naming Conventions

In all results we refer to forward checking as FC, dynamic variable ordering with the minimum remaining values heuristic as DVO, conflict directed backjumping as CBJ, and chronological backtracking as chronBT. The naming convention for solvers is AC3?-FC?-DVO?-CBJ?. With the exception of CBJ/chronBT, if the particular feature is turned off then it has the word "no" in front of it. Thus the solver that uses AC3, has FC off, uses DVO, and uses CBJ has the name "AC3-noFC-DVO-CBJ".

Through all of our experiments we gathered results for all combinations of AC3 on/off, FC on/off, DVO on/off and CBJ on/off. Several of these combinations produced solvers that could not solve the problems, so we imposed a timeout limit of 10 minutes. The solvers of primary important to this study (the solver with all features turned on and all of the solvers with only one feature turned off) were all able to find solutions with each seed in less than 10 minutes.

4.2 Our Performance Metrics

We track six metrics: Restrict Domain Counter, Jump Counter, Label Counter, Unlabel Counter, CPU Time, Real Time and Restart Counter. Restrict Domain Counter is the number of times that the solver restricts a domain by ANDing the current domain of a variable with some bitset. The Jump Counter is the number of levels that are jumped over when backtracking (this value is zero when CBJ is turned off). Label Counter is the amount of times we attempt to label a variable and Unlabel Counter is the amount of times we unlabel a variable. Restart Counter is the number of times the solver restarts the search and reshuffles the ordering of each domain.

The CPU Time and Real Time are both in seconds. Since our CPU timer less reliable for runtimes close to zero, we record the Real Time for both CPU Time and Real Time when the reported CPU Time is less than one. CPU Time and Real Time do not include the processing time for reading in the crossword puzzles/dictionary or creating the arrays of bitsets. The timing starts when the function CSPSolver::solveCSP is called.

4.3 Randomization

The random number generation is invariant to the amount of puzzles in the puzzles file, the order that the puzzles are solved, and the specific puzzles in the puzzles file. Thus if the puzzles file has only the four standard puzzles in it, given a specific seed the output of the smallest puzzle when it is the first puzzle in the file will be the same as the output when it is the last puzzle in the file. We achieve this by seeding the random number generator with the specified seed before each crucial section of code. Our seed points are: when we read in the dictionary, when we start to read a new puzzle from the puzzle file, and just before we call CSPSolver::solveCSP.

To find a variety of solutions we randomly pick variables when choosing the next variable to label if multiple variables exist that are equally good. When not using dynamic variable ordering, this involves randomly picking a variable in the variablesNotLabeled array. When using dynamic variable ordering, this involves randomly picking a variable from the array of variables with the smallest domain size.

We also randomly shuffle the domains of each variable. When creating the domain for each variable, we take the array of words for the given word length, loop through each index randomly swapping the word at the given index with another word in the array, and then create the array of bitsets. Thus all variables of the same word length loop through their domains in a different order.

On must take special care to ensure deterministic behavior given a seed when using our restart optimization in Section 3.5. In a traditional timeout scenario, one would set a timeout of say 30 seconds and force an automatic restart once the elapsed CPU time is greater than 30 seconds. The problem with this method is that the timeout due to the elapsed CPU time check is not guaranteed to occur on the same iteration of the search loop during each execution. If the timeout does not occur on the same iteration during each execution then across different executions of the program a different number of random numbers will be generated for a given run. This will result in different outputs because the restarts will be effectively starting the search at differ-

ent points of the pseudorandom number generation based on the initial seed.

We devised a simple method to ensure deterministic behavior given a seed when restarting. We allow each value to define a window value in which all timeouts that occur during the same window are guaranteed to force a timeout at the same time (the end of the window). We implemented this guarantee by adding an additional constraint to the timeout check that the Label Counter must be a multiple of the window value in order for a timeout to occur.

As an example, the window value for our algorithms that use forward checking is 5,000. As we discussed in Section 3.5, the first run for our forward checking solvers is allowed to last for 0.75 seconds. After 0.75 seconds has elapsed, the solver will be forced to restart on the next iteration where the Label Counter is a multiple of 5,000. As long as our timer does not vary in its readings more than the amount of time the algorithm takes to perform 5,000 label operations, our program will be completely deterministic given an initial seed value. We set the window value for all non-forward checking algorithms to 30,000, and unfortunately did not have sufficient time to conduct experiments to find an optimal window value for each algorithm.

5 Progressive results analysis

5.1 Initial naïve implementation

When we began the task of developing a CSP solver for crossword puzzles, we obviously had no experimental data from which to judge how much optimization would be required. We initially developed a generic CSP solver, for the following reasons:

- to avoid premature optimization
- to gauge the level of optimization and specialization required to handle the task
- to have a program capable of handling a much more general class of CSP problems than just crosswords.

Obviously, we were immediately confronted with the fundamental decision of how to represent the crossword problem in the CSP formalism. In particular, there is the key choice of what to consider as the variables: individual character slots, or whole words. As we saw in Section 1, there is clearly a trade-off between the small domain size of the characters (a few dozen possible values, as opposed to tens of thousands of potential words) and the simple binary constraints over word variables.

Ideally we would have liked to develop both approaches in order to compare their performance and decide conclusively which is more effective. However, we estimated that our time would be better spent by focusing on one approach, and going into greater depth. For the reasons explained in Section 1, we decided to us a word-variable CSP representation of the problem.

Our initial results quickly showed, however, that this generic approach was overwhelmed by even simple 5*5 crosswords. It therefore became clear that a much more specialized approach would be required, taking into account the nature of the variables and constraints. In particular we hypothesized that tremendous gains in performance could be reaped by exploiting the very specific nature of the constraints (i.e. the fact that every constraint is simply the equality between the characters at two specified positions in two words). To validate this hypothesis, and hopefully improve our crossword results, we redeveloped our base CSP solving platform using the bit-set dictionary-tree look-up method described in Section 3.

5.2 A problem-specific approach

We implemented this bit-set dictionary representation based primarily on Ginsberg's 1990 paper. As we have already seen in part 3, it provides efficient data structures for restricting the current domain of a word-variable given a constraint of fixing one of its characters. We hypothesized that naively running the

standard CSP solving algorithms (and in particular iterating over the potential values of a variable to evaluate constraint) using these data structures would be very inefficient. Our results confirmed this: we were still unable to solve simple 5*5 crosswords. During this time we were primarily using a set of extremely simple crosswords (around 4*4) to evaluate our performance and debug.

We therefore modified the basic CSP algorithm to exploit the strength of the bit-set dictionary data structure, by simultaneously pruning all impossible values of a variable based on a constraint with a previously assigned algorithm. This reduction of the usual iteration over potential values to a single operation dramatically improved performance, and allowed us to finally tackle the simpler crosswords. However, our algorithm was still unable to solve more complex crosswords in a reasonable amount of time. We therefore went on to implement several algorithmic extensions. The implementation details of these extensions can be found in Section 3; here we shall examine the key results and our progressive analytical process.

5.3 Extensions

5.3.1 Forward checking

Based on the quantitative results in the CSP literature, we hypothesized that the most effective extension would be Forward Checking. However, this also proved to be relatively tricky to implement. In particular, the standard pseudo-code for FC had to be adapted to appropriately use our bit-set dictionary system (see Section 3). In order to debug our implementation, we used a set of specially chosen crosswords designed to test specific components of the algorithm (e.g. a crossword on which no unlabeling is ever required, an extremely constrained crossword with no black squares, etc.).

Once implemented, the results are extremely positive, as evidenced by the comparison of the run-times on Puzzle 1 in this table (note that the CPU timer is inaccurate for run-times below one second).

	Averages		
	Label Counter	Unlabel Counter	CPU Time (seconds)
noAC-FC-noDVO-chronBT	34	15	**0.03**
noAC-noFC-noDVO-chronBT	8,084,841	4,049,466	19.54

Forward checking reduces the time taken to treat Puzzle 1 by several orders of magnitude (note that we cannot compare for the other puzzles, as the base algorithm could not solve them before time-out). To understand why the effect is so strong, we evaluated certain meta-statistics: the number of calls to label and unlabel in particular. The origin of the speed-up is clear: the Forward Checking algorithm completely avoids labeling most “dead-end” variables, which would otherwise lead us down very large useless parts of the search tree. This effect is also true for harder problems, but is a lot less pronounced when DVO is activated.

5.3.2 Dynamic Variable Ordering

In parallel, we implemented Dynamic Variable Ordering, which was extremely easy to add given the architecture of our system. We had hoped that this would provide a good performance boost, and this immediately proved to be the case. Results for Puzzle 1 are below (with AC3 and CBJ on to avoid time-outs).

	Averages		
	Label Counter	Unlabel Counter	CPU Time (seconds)
noAC-noFC-noDVO-chronBT	8,084,841	4,049,466	19.54
noAC-noFC-DVO-chronBT	133	71	**0.00061**
AC3-FC-noDVO-CBJ	33	15	0.033
AC3-FC-DVO-CBJ	37	16	**0.0086**

With FC off, this DVO clearly has a tremendous impact, decreasing run-time by around 4 orders of magnitude. This is due to the tremendous decrease in the number of calls to label and unlabel, thanks to the "intelligent" ordering of variables assured by DVO. Indeed, in the absence of FC, our results show that DVO is the best of our three other extensions to have (see tables in annex).

In the presence of FC however, the results above for Puzzle 1 do not show a great improvement with DVO. The figures for harder problems tell a different story. Here are the results for Puzzle 4:

	Averages	
	CPU Time (seconds)	Real Time (seconds)
AC3-FC-noDVO-CBJ	21.31	23.97
AC3-FC-DVO-CBJ	4.87	4.87

Here DVO has a substantial impact (roughly one order of magnitude) on run-time, even in the presence of FC. We can conclude that DVO is extremely important for an algorithm which must scale well with problem difficulty. It is also interesting to note that DVO is essentially "free" in terms of run-time when couple with FC, as the smallest-current-domain variable heuristic is sufficient.

Indeed, we evaluated multiple heuristics for choosing the next variable in DVO. In particular we attempted various degrees of "look-ahead", for example by evaluating the number of values which do not conflict with any current assignment, rather than just considering the current domain size (of course, with Forward Checking there is no difference). This method turned out to be the most successful in our experiments, and was retained subsequently for all our results whenever FC was turned off.

5.3.3 Arc consistency

We next decided to implement a pre-processing step, using arc-consistency. Indeed, the CSP literature shows that methods such as AC3 and AC5 can be very effective at simplifying complex CSP's. These methods also have the advantage from an engineering perspective of being decoupled from the rest of the algorithms, as they run once as a pre-processing step, before the main solver. This greatly facilitates debugging and good system design. Our results on the base set of problems were not clear-cut, as evidenced by these figures:

	Averages		
	Label Counter	Unlabel Counter	CPU Time (seconds)
Puzzle 2			
noAC-FC-DVO-CBJ	732	436	**0.019**
AC3-FC-DVO-CBJ	1472	864	0.074
Puzzle 4			
noAC-FC-DVO-CBJ	107,677	64,766	5.51
AC3-FC-DVO-CBJ	97,904	58,819	4.86

Analyzing the results showed that is actually bad for over-all run-time performance on the easier problems. However, on harder problems (i.e. puzzle 4 and harder) it has a noticeable decreasing effect. We can once again explain this thanks to the label and unlabel counts: AC3 reduces them even in the presence of FC and DVO. This is clearly worth the time-cost of pre-processing in harder problems.

We implemented AC3 here, and had been planning on implementing AC5 to make the algorithm more efficient. However, profiling our system's runtime performance revealed that the arc-consistency pre-processing step is extremely fast relative to over-all runtime on hard problems. AC5 would therefore have provided a negligible performance boost, and we thus decided not to pursue it.

5.3.4 Backjumping

The final algorithmic extension was to implement a form of back-tracking. We implemented conflict-directed backjumping (CBJ), but our performance results did not indicate a great performance boost (especially with FC or DBO running), as evidenced by the run-time results here for Puzzle 1:

	Averages			
	Jump Count	Label Counter	Unlabel Counter	CPU Time (sec)
AC3-FC-DVO-chronBT	0	37	16	0.0088
AC3-FC-DVO-CBJ	0	37	16	0.0088

It is clear that while CBJ has some impact in the absence of FC and DVO, it has no impact whatsoever with them on. We hypothesized that this was because we were rarely "backjumping" back several levels (which is the case in which CBJ has an advantage over basic chronological backtracking). To evaluate this hypothesis, we put in place "jump" counters in our code. These show that the number of jumps is always relatively low, and that with Forward-Checking on we basically never backjump multiple levels (see table above and the Annex).

There are however some notable cases (mainly with DVO on but FC off) in which CBJ still has a strong run-time impact, despite the small number of backjumps. This is the case in the following results on Puzzle 4 for example:

	Averages			
	Jump Counter	Label Counter	Unlabel Counter	CPU Time (sec)
noAC-noFC-DVO-CBJ	3,310	**49,539**	27,820	1.21
noAC-noFC-DVO-chronBT	0	1,636,240	1,026,021	30.14

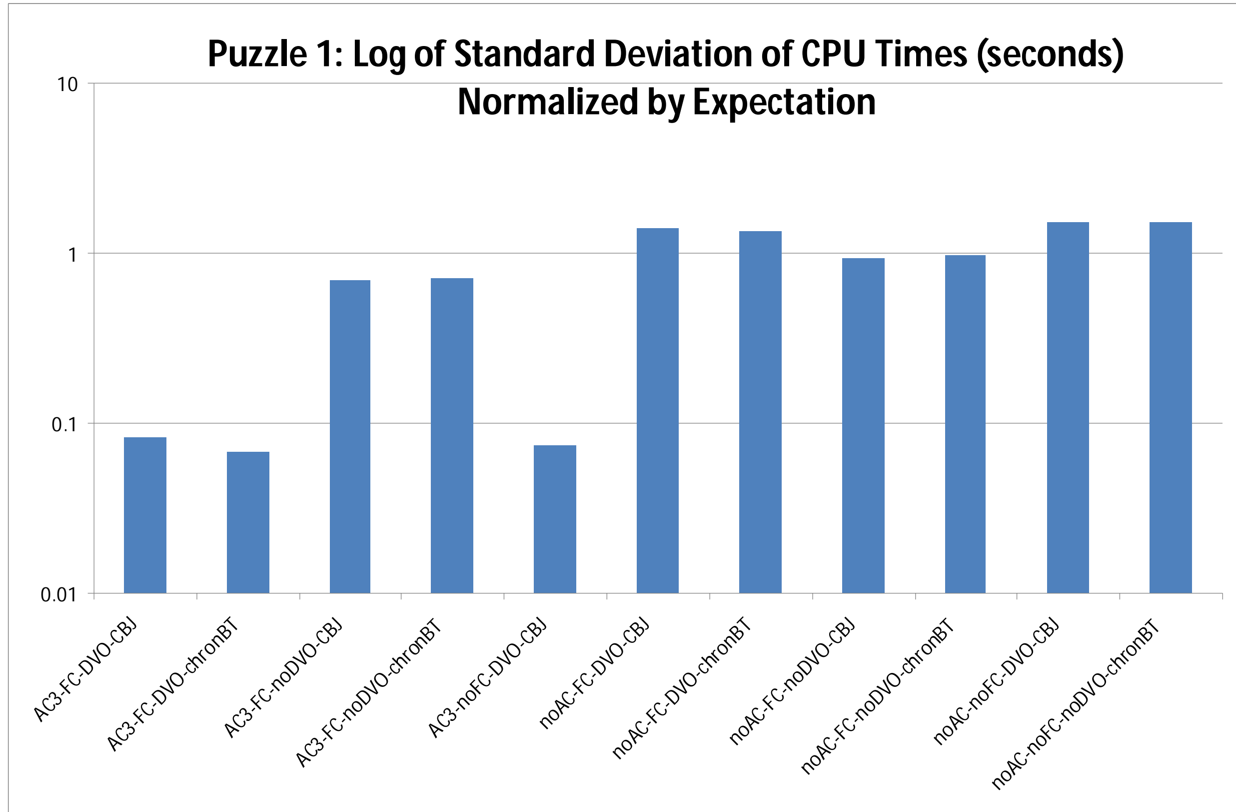
Here CBJ decreases run-time by more than one order of magnitude; the label and unlabel counters show that this is due to a huge decrease in the number of these function calls. Yet CBJ only jumped over 3310 levels (tiny compared to the number of saved labels)! We hypothesize that this is because some of the variables skipped by these jumps would have led to large search tree explorations, but were unable to corroborate this hypothesis. Had we had more time, we would have implemented Dynamic Backtracking to see if it has more of a performance impact than CBJ.

5.4 Algorithm variance analysis

Until this point we had been primarily considering aggregate average results in order to get the "big picture", to understand the general effectiveness of our algorithms. However, averages do not capture the variations in the re-

sults and performance of a given algorithm on a given problem.

To evaluate this, we considered the standard deviation of the CPU run-time of a given algorithm on a given problem for a set of 15 different random seeds. To account for the fact that the different algorithms have very different expected run-times, we normalized the standard deviation relative to the expected value, obtaining the following results.



These results reveal that the four algorithmic extensions we implemented have very different effects on the variance of the performance results:

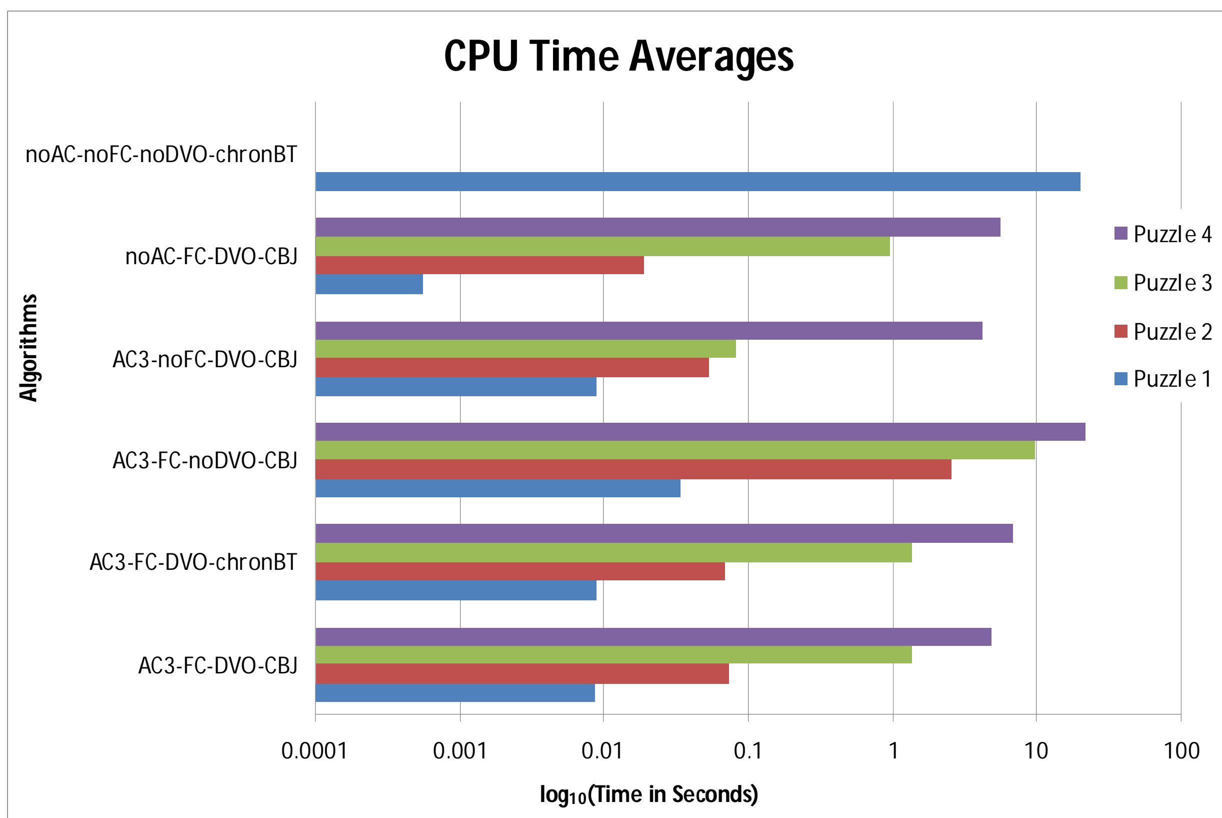
- DVO greatly decreases the normalized standard deviation (by roughly an order of magnitude). A likely explanation for this behavior is that without DVO a "bad" choice of random seed can lead to a very inefficient ordering of variables (with extremely large current-domain variables first for example), whereas a "lucky" random seed can make the same problem very easy; DVO levels the field in this respect, as it enforces a good ordering of variables. One interesting anomaly is that the solver with DVO on and everything else off has higher normalized standard deviation than any other solver.
- AC3 also has a very strong decreasing effect on variance. This is probably due to the fact that arc-consistency prunes out some obviously bad variable values, which may otherwise have slowed down the algorithm for "unlucky" random seeds.
- FC has a slightly decreasing effect, but primarily in the absence of DVO. The explanation is similar to that of DVO.
- CBJ has little effect either way; this is no doubt because of the general lack of effectiveness of this extension.

These variations in variance are theoretically interesting, but can also have practical importance. Indeed, in a time-critical environment, it is desirable to have algorithms with minimal run-time variance (even at the expense of some expected run-time), to be able to more accurately plan time usage and avoid risking running out of time.

5.5 Final results

5.5.1 Raw run-time performance

To take a step back and see the full picture of our algorithm's performance, we ran the full stable of algorithms on the set of four problems, averaging over 15 different seeds. In particular, we performed an ablation study by trying removing each of the algorithm extensions individually. The CPU time results are below.



These results appear to be largely homogenous. We suspect that this is due to our restart optimization method. While by and large our restart optimization method improved performance, we believe that it eliminated many of the large distinctions between the solvers and made many of our results fairly random.

6 Future work

The results analysis we carried out throughout our experimental phases revealed several development and research paths to explore. In particular we would like to extend the use of heuristics from variable choosing (as in DVO) to other parts of the CSP solver, notably the choice of the next value to explore. There are obvious heuristics for this (such as the least constraining value heuristic), but one could also imagine more complex approaches, with a certain degree of look-ahead and preprocessing for instance.

It would also be interesting to further explore backtracking/backjumping techniques.

Indeed, we had only limited success with these approaches, and it would be instructive to try to improve on this. In particular we would evaluate Dynamic Backtracking relative to our current algorithms.

Additionally, an area which could be of great practical importance is studying how to do intelligent restarts and time-outs of our algorithms. Indeed, performance can vary quite widely depending on random factors (summarized in our code by the random seeds), and thus intelligent restarting strategies could prove to be just as useful for CSP solving as in hill-climbing SAT solvers, for example. Our success with restarts that reshuffled the order of each domain also suggests that developing a preprocessing step that orders each domain in a "good way" could be useful.

Finally, although all the problems we considered here were crosswords, we did not develop any domain-specific heuristics or strategies based on the nature of the puzzles (beyond our bit-set dictionary domain representation). There could potentially be huge performance advantages to be had by taking into account the very specialized nature of crossword problems.

7 References

[1] M. Ginsberg, et al, 1990. *Search lessons learned from crossword puzzles.* In Proc. of AAAI-90.