Crossword Puzzles and Constraint Satisfaction

James Connor, John Duchi, and Bruce Lo
jconnor@stanford.edu, jduchi@stanford.edu, brucelo@cs.stanford.edu

Stanford University
CS 227: Assignment 2
May 6, 2005

Introduction

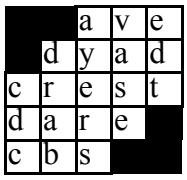
Crossword puzzles provide an interesting test bed for evaluation of constraint satisfaction algorithms. Constraint satisfaction problems are among the most difficult problems for a computer to solve, but methods exist that make solving them relatively easy and quick to run.

In this report, we will be investigating different constraint satisfaction problem (CSP) algorithms, including forward checking, dynamic variable ordering, conflict-directed backjumping, and arc consistency. Past research has suggested these algorithms make for significantly more efficient CSP solvers. Although CSPs are in the class of NP-Complete problems, even fairly large instances of constraint satisfaction problems can be solved quickly.

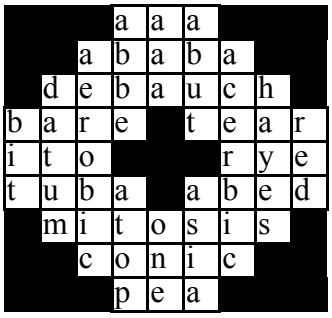
We describe first the algorithms and techniques we use to solve the CSPs, along with the optimizations we make to our CSP solver that take advantage of the structure of crossword puzzles. Next, we will describe the results and the reasons we have gotten those results with our algorithms. The final section is a concluding discussion and a few ideas for further work.

Crosswords to Solve

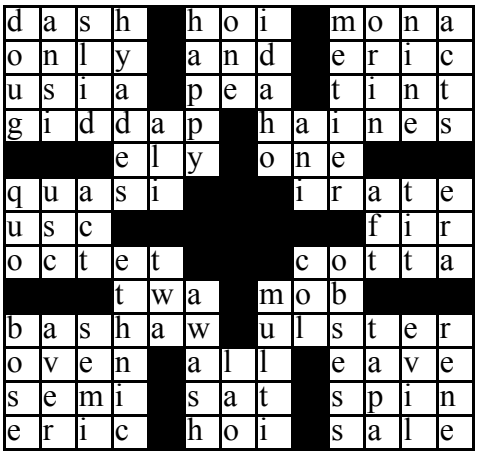
Below are examples of solutions to the crosswords we consider in this paper, labeled A, B, C, and D. We will maintain this labeling throughout.



Crossword A



Crossword B



Crossword C Crossword D

Figure 1 (Crosswords Used)

Algorithms and Techniques Used

Because crosswords have properties that can be exploited to make solving significantly quicker, some of our techniques are not directly applicable to general CSP solving, but they are quite effective in crosswords. Nevertheless, as Ginsberg et al. suggested, crossword solving problems can be transformed into arbitrary declarative queries, so the results are not too narrow [2].

The basic framework we use is that provided by Prosser [5]; our algorithms match his backtracking, forward checking, and conflict directed backjumping algorithms with a few modifications. They, however, use the basic backtracking strategy of attempting to label each variable, un-labeling and retrying if labeling fails, and returning true once all variables have been assigned values or false once all variables have failed.

The pseudo-code for basic backtracking search is given here; our methods only modify the labeling and un-labeling steps.

```
Function BacktrackSearch (variables V) {
        consistent = TRUE
        level = 1;
        Variable x_i = initialize()
        Loop
                If (consistent) then (x_i, consistent) = label (x_i)
                Else (xi, consistent) = unlabel (x_i)
                If (No more unassigned variables) return "Solution Found"
                Else if (All variables have been tried) return "No Solution"
        End loop
} end BacktrackSearch
```

Here, label (x_i) returns the next variable to attempt to instantiate if it finds a consistent step, or the variable x_i it is given if it cannot find a consistent solution at this point in the

search. In our implementation, if we are not dynamically selecting variables, selecting the next variable to instantiate is done randomly rather than simply by incrementing the level, as Prosser did.

Variables

To represent the crossword as a constraint satisfaction problem, we followed Ginsberg et al.'s approach. We treat words in the crossword as variables in the CSP with constraints among and on themselves. That is, a word is a variable constrained to be a string of a certain length whose letters, wherever the word intersects another in the crossword, must be the same as those of the intersected word. A variable *x* keeps a list of the other variables with which it intersects and the positions in which it intersects them, constraining *x* such that *x*'s letters must match other variables'.

The domain of each variable is the set of words in a 20,000 word dictionary, though each variable's domain can be made significantly smaller by constraining the words that the variable might become by the length of the word the variable represents, which brings the size of the domains of variable to about 2,000. Each letter in the words is one of the normal 26 along with three extra characters, & (the ampersand symbol), ‘ (the single quote), and . (a period).

Lexicon

Implementing a fast and space efficient lexicon to contain the words in our dictionary was of paramount importance. Variables must be able both to keep track of their domains—the words they might take on as values—and to make quick queries of the lexicon to find words whose letters match the constraints on a variable. To that end, we represented the lexicon as a series of packed bit arrays whose indexes corresponded to words. Bitwise calculations are very quick in C and C++, and they provide a compact representation of the dictionary, so it seemed reasonable to pursue such an approach.

The lexicon is structured as layers of 29 bit arrays for each word length, each index into a bit array indicating whether or not the word stored at that index has a specific letter at the bit array's layer. To make this clearer, a bit array *B* in the fifth layer of bit arrays for words of length six would correspond to a letter, say ‘b’, and any positions in the bit array with a 1 rather than a 0 would indicate that the six letter word indexed to that position had the letter ‘b’ as its fifth letter. Because of this representation, if we quickly want to find words of length six that have an ‘a’ in their first position and a ‘b’ in their fifth, we would simply bitwise intersect the bit array corresponding to words of length six with a's in their first position with the bit array corresponding to words of length six with b's in their fifth position. This intersection calculation runs very fast and will tell us which words have a's in their first and b's in their fifth positions. Because we wish to be able to keep track of which words have specific letters in their positions, we impose a strict ordering on the words in the lexicon (although we can and do randomize this ordering for purposes of finding multiple solutions to the crossword problems).

Because of the relative smallness of our bit array representation, we can store all 20,000 words of our dictionary in 2.4 Megabytes of space, but we are able to have intersection and union operations that run in $O(d/32)$ time, which is linear but fast, since it is a bit calculation and d, the domain size, is usually under 2000. Each variable also stores a bit array for its current domain that it updates as search through the problem space executes. This storage of bit arrays allows for fast domain updates in forward

checking and conflict directed backjumping methods, because intersections and unions of domains based on the constraints variables have with one another can be calculated very quickly. The lexicon allows us to in constant time access the bit array corresponding to all words that have a letter c at a specific position in words of length l. Once we have the bit array, intersecting its domain with the domain of a variable will immediately give us the remaining values possible for the variable.

Forward Checking

Forward checking algorithms use current instantiations of variables to prune domains of variables that have not yet been instantiated. They allow one, in effect, to look ahead in the search tree of a CSP and check the status of domains of variables, and if one of these domains has been annihilated—all of its possible values have been eliminated—to begin backtracking earlier. Forward checking has proven to be one of the most effective methods of speeding up solxing CSPs, and our results supported this.

In forward checking, for each variable x_j we keep three stacks, *reductions*$_j$, *past-fc*$_j$, and *future-fc*$_j$. The reductions stack for a variable x_j, *reductions*$_j$, keeps elements that are lists of values in the domain of x_j disallowed by previously instantiated variables. That is, a variable instantiated earlier in the search than was j removed some set of values from x_j's domain, and these are kept on the *reductions*$_j$ stack. *Past-fc*$_j$ is a stack of the levels in the search that have checked against variable x_j, while *future-fc*$_j$ is the stack of future variables against which x_j has checked. We use the algorithm presented in lecture and in Prosser's 1993 paper on CSPs with a few small modifications.

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 (we do not use Prosser's indexing scheme, though this difference is relatively insignificant). After attempting to instantiate a variable x_i, for every variable x_j that remains unassigned we remove the possible values for x_j that do not meet the constraints between x_i and x_j via a method called *check-forward*. For every x_j, we add to *reductions*$_j$ all values from x_j's domain that have been eliminated by constraints with x_i, and we push variable x_j onto *future-fc*$_i$, since x_j comes after x_i in the search. We also push the current level onto *past-fc*$_j$.

The difference between Prosser's and our algorithm comes if we annihilate the domain of possible values for some x_j. Rather than just eliminating the value w to which x_i has been instantiated from x_i's domain, we take advantage of the structure of the crossword puzzle to eliminate a whole set of values. We eliminate from x_i's domain all those values that also break the constraint that w breaks. We know that x_i intersected with x_j at a specific position p, and as such, whatever letter a_p was at position p in w causes x_i to annihilate x_j's domain. Thus, we can eliminate all words whose p^{th} letter is a_p, and we can get these values very quickly from our lexicon's bit arrays. This allows us to more quickly eliminate possible values from x_i's domain, thus speeding up search.

Dynamic Variable Ordering

Dynamic variable ordering (DVO) is a usually heuristic strategy that attempts to select the best variables to explore at every point in the search. In our implementations, we used the minimum remaining values (MRV) heuristic, which selects variables to instantiate whose remaining domains of possible values are smallest. Past research has demonstrated that DVO with the MRV heuristic is very effective when used in

conjunction with forward checking [1].

DVO is not as effective when used with backjumping alone. In backjumping without forward checking, if we want to choose the variable with the minimum remaining values consistent with all previous values, we must do a backward check for each variable, which would be computationally expensive. If we avoid the computation and simply choose the variable with the smallest current domain, we lose most of the benefits of the heuristic because every variable domain would be mostly full (except for AC-3 and a few previous backjumping steps) and would not reflect how constrained a variable actually is. Past research has proved that FCCBJ with DVO using MRV is at least as good (in the sense that it performs fewer consistency checks on variables) as CBJ with DVO using the MRV heuristic [1]. In our implementation, when we select the next variable to instantiate in the search, we randomly select from the set of unassigned variables whose domains of possible values are the smallest. Another approach that might be investigated is selection among these, but favoring those variables that most constrain or least constrain further search, determined by the length of the word we consider.

Conflict-Directed Backjumping

In conflict-directed backjumping (CBJ), we maintain a list of the levels in the search with which every variable conflicts. That is, we have a set $conf\text{-}set_i$, which is a set of the past levels with which x_i conflicted, and the level l is in $conf\text{-}set_i$ if the variable assigned at level l had a constraint with x_i that ruled out a possible value for x_i. The idea is that we keep track of problem variables in the search space, and rather than instantiating them and continuing on, checking what becomes an exponential number of variables further down the search tree that do not work, we immediately jump back to the problem variables and switch their values.

With CBJ, we again use the same algorithms as Prosser, but make a modification similar to the modification we did with forward checking and annihilation of domains. In CBJ's labeling method, we may find that some previously assigned level r is inconsistent with the variable x_i we are trying to assign at level l. We keep the level r in $conf\text{-}set_i$, but rather than simply removing the value w from x_i's domain, we remove the set of values that also break the constraint that w broke.

When combining CBJ with forward checking, and when running CBJ on its own, un-labeling variables is also slightly different from Prosser's approach, though not significantly. Normally, upon un-labeling, we begin at the level r, which is the most recently visited level in $conf\text{-}set_i$, and reset the domains of the conflict sets for every variable assigned up to the current level. In the case of FCCBJ, we also undo all domain reductions (and, intuitively, variable assignments of levels over which we skip) that resulted from forward checks to our current level.

Arc Consistency

Arc consistency shrinks the search space for a CSP by removing unsupported values from variables' domains. A variable x's value v_k 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_k. AC-3 works in two steps. It first iterates over every variable x_i, checking whether its values are supported in each constraint over x_i. While it does this, AC-3 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. This algorithm is described in more detail in Hentenryck et al.'s treatment of arc-consistency [3].

We implemented AC-3 in two ways. First, we implemented a generic version of the algorithm. In this, for every variable we check whether every value is supported, removing values that are unsupported. Secondly, we implemented AC-3 but optimized it using the structure of the crossword problem. Instead of checking each value's support, we check each character that could appear in a particular position *p*; as there are only 29 characters (including non-letters), this runs much more quickly than checking all 2000 or so values to which a variable might be instantiated. If a particular character has no support, we remove all values from the variable's domain that contain that character in position *p*, and this runs quickly using our bit array based variable domain implementation.

Miscellaneous Optimizations

The bit arrays were the most significantly optimized piece of our crossword solver. Finding the number of elements stored in a bit array would not be the quickest operation were it not for the pre-computing we do on the bit arrays. We store a length 2^{16} character array, call it *bits_in*, in which each index is the number of 1 bits in the 16-bit integer that corresponds to that index. For example, *bits_in*[2] would have value 1, since there is one on bit in the number 2. This allows for extremely quick computation of the number of elements stored in a bit array, because we can calculate the number of on bits in any 16 bits in constant time [4].

Results

In this section, we will examine the results of running our CSP solver on crosswords of different sizes and with different options turned on. We ran experiments with combinations of forward checking, dynamic variable ordering using minimum remaining values, conflict-directed backjumping, and AC-3 enabled. Our results suggest that the most effective methods for crossword puzzles—and CSPs generally whose variable domains are very large and thus not constrained enough to be effectively made smaller by arc consistency algorithms—are those that combine forward checking, dynamic variable ordering, and conflict-directed backjumping, which is orthogonal to the first two extensions of backtracking search. Arc consistency did not actually speed search in any significant way.

Dynamic variable ordering in conjunction with forward checking was the strategy that allowed us to solve the largest problems—without it, crossword D became largely unsolvable, and solving crossword C took much longer. Running DVO with forward checking and conflict-directed backjumping (FCCBJ-DVO) gave the fastest CSP solver, and it solved the crosswords most consistently (See Figure 2). Interestingly, FCCBJ-DVO more quickly solved crossword D than crossword C, which can be explained by the backjumping—crossword D's four thirteen letter words are difficult, and if they are solved, they constrain further search enough that it becomes quicker. We also kept track

of the number of bit arrays representing domains we allocate during search; these were allocated during attempts to label a variable and are also used to keep track of the values we eliminate through forward checking to a level. Thus, the number of bit arrays is directly related, if loosely, to the number of nodes we expand in a search.[1] One can see that FCCBJ with DVO allocates fewer bit arrays than other search strategies even though forward checking allocates extra bit arrays to do its forward checks, and so FCCBJ-DVO is the most space-efficient search we considered as well as being the fastest.

Averages	**Time**	**Bit Array**	**Success**
a	0.01	141.8	1
b	0.08	2991	1
c	0.2	13570.7	1
d	0.36	32674.7	1
Medians			
a	0.01	103	1
b	0.06	2124	1
c	0.21	10984.5	1
d	0.16	15012.5	1

Figure 2: Average and median run-times for forward checking, conflict-directed backjumping and dynamic variable ordering.

Dynamic variable ordering was most effective when it was used with forward checking, as Bacchus and van Run suggested. In fact, running the CSP solver with forward checking and DVO was, aside from DVO with forward checking and conflict-directed backjumping, the only way to solve crossword D. FC-DVO did solve crossword D, only not so consistently as FCCBJ-DVO; FC-DVO solved D on 64% of its attempts and crossword C on 80% of its attempts. FCCBJ-DVO had a 100% success rate on these (see Figure 3). The standard deviation for times solving crosswords C and D were very large, both above 200, which can be explained as a result of randomness in variable selection and the dictionary. In Figure 4, one can see results for FCCBJ without dynamic variable ordering. Notice that not once did the solver solve crossword D, and crosswords B and C took significantly longer than they did with dynamic variable ordering.

Comparing these results with those for FCCBJ-DVO, it becomes apparent that conflict directed backjumping is a method that helps to significantly constrain search. This is especially true in more difficult problems. In smaller problems, such as crossword A, there are so few constraints and so few domain annihilations from selecting a specific value for a variable (as long as we have forward checking), we barely need to backtrack, so CBJ becomes moot. In larger problems, though, especially D, we have far more constraints that are more difficult to satisfy. The four length thirteen words in D provide many constraints that must be satisfied, and so CBJ offers a way to jump back to them as problem variables, which means that much of the search can be spent looking for

1Note that 6907 bit arrays are allocated for the dictionary, and we do not include these in the bit array counts.

Averages	AC-3 Time	Calc Time	Total Time	Bit Array	Success
a	0.17	0	0.18	10685.6	1
b	1.05	0.05	1.1	27135.2	1
c	1.54	153.03	154.57	10623236.7	0.8
d	7.53	285.8	293.33	23062559.64	0.64
Medians					
a	0.17	0	0.18	10643.5	1
b	1.05	0.04	1.08	25085.5	1
c	1.52	45.66	47.18	2829301.5	1
d	7.51	243.93	251.44	21477401	1

Figure 3: Forward checking with dynamic variable ordering statistics.

Averages	**AC-3 Time**	**Calc Time**	**Total Time**	**Bit Array**	**Success**
a	0.63	0.03	0.66	4739.5	1
b	3.82	40.46	44.28	2322790.5	1
c	4.68	485.81	490.48	56118964.5	0.2
d	21.96	578.04	600	105059561.8	0
Medians					
a	0.63	0.02	0.65	4302	1
b	3.83	7.52	11.33	487296.5	1
c	4.67	595.33	600	68222696	0
d	21.87	578.13	600	104274670.5	0

Figure 4: Forward checking with AC-3 and conflict-directed backjumping.

assignments that meet the constraints of those problem variables. We do not waste exponential amounts of time trying to assign easier variables later in the search when the longer words were actually the problem. As such, the variance of solving times and the actual solving times when running FCCBJ-DVO are much smaller than they are when running simple FC-DVO.

Because AC-3 runs in time $O(ed^3)$, where e is the number of constraint edges between variables, d is the size of the largest variable domain, and the domains are on the order of 2000 values, AC-3 runs fairly slowly. Also, in a problem such as the crossword puzzle, finding unsupported values to eliminate from variable domains proves very difficult. There are 20,000 words in the dictionary, and the crossword problem is loosely constrained relative to the sizes of domains of its variables, and so it is often easy to find values that will support any—or almost any—instantiation of some variable. Our results supported the inefficacy of arc consistency checking.

When we ran experiments with FCCBJ-DVO, as stated above, we had very fast results—our solver solved even the most difficult problem in our test bed, on average, well under a second. Pre-processing the crossword with AC-3, did not make FCCBJ-DVO run any more quickly (see Figure 4). In fact, the median and mean times (Calc Time in the table) for solving a crossword with FCCBJ-DVO were slower after having AC-3 run on them than they were without AC-3. When the total time (time of AC-3 plus the time of solving with FCCBJ-DVO) is considered, we find that AC-3 gives no

performance advantage. Both versions can solve all the provided crossword puzzles at a success rate of 100%, but AC-3's processing slows down overall solution time.

Averages	AC-3 Time	Calc Time	Total Time	Bit Array	Success
a	0.61	0.01	0.62	3833.3	1
b	3.7	0.13	3.83	18542.3	1
c	4.53	0.33	4.86	43885.6	1
d	21.83	1.22	23.05	208697	1
Medians					
a	0.61	0.01	0.62	3703.5	1
b	3.69	0.1	3.81	17876	1
c	4.54	0.19	4.82	33841	1
d	21.76	0.18	22.01	141262	1

Figure 5: Forward checking with dynamic variable ordering, conflict directed backjumping, and AC-3.

We did find significant speedups of AC-3, however, using our optimized version. Because we did not have to check all the values to which a variable might be instantiated, AC-3 ran much more quickly. The results can be seen in Figure 6.

	Run Times in Seconds			
Crossword	a	b	c	d
AC-3 Non-optimized	0.21	1.26	1.54	7.53
AC-3 Optimized	0.06	0.15	0.36	0.58

Figure 6: AC-3 Optimization Testing

The last modification we made to standard backtracking was to use forward checking to propagate the constraints we have established farther forward in our search than the current variable we instantiate. Forward checking proved to be very important to solving CSPs, especially, as mentioned above, in conjunction with DVO. In Figure 6, one can see that even with DVO, a conflict-directed backjumping algorithm with AC-3 runs incredibly slowly relative to our other methods. A CSP solving algorithm without forward checking, even with other bells and whistles enabled, fails to solve all the problems within 10 minutes except the most simple, crossword A. In a domain like crossword puzzle solving, forward checking is very effective, because as soon as one variable x is instantiated, it eliminates on the order of 25/26ths of the domain for other variables intersecting it, constraining the letters at their intersection to be the letter in x. Without this sort of paring down of domains, dynamic variable ordering becomes totally ineffective, as very few variables have small enough domains to justify choosing them, and the search has a much larger branching factor.

Averages	AC-3 Time	Calc Time	Total Time	Bit Array	Success
a	0.64	3.86	4.49	172878.3	1
b	3.85	596.15	600	23009098	0
c	4.68	595.32	600	26532393.1	0
d	22	578	600	25562252.3	0
Medians					
a	0.64	0.07	0.71	6522.5	1
b	3.84	596.17	600	22425872.5	0
c	4.68	595.33	600	26534807	0
d	21.96	578.04	600	25552188	0

Figure 7: Conflict directed backjumping with AC-3 and dynamic variable ordering.

It is interesting to note that the different algorithms have break points—points at which the time it takes them to solve a crossword grows very quickly and beyond our 10 minute time limit—at very different points. Forward checking with dynamic variable ordering exhibits what looks like slow, or even polynomial growth with the size and difficulty of the crosswords. But forward checking with conflict-directed backjumping and conflict directed backjumping with AC-3 and dynamic variable ordering both exhibit "blow-up," becoming unable to solve problems of the difficulty of crossword C and crossword B, respectively (see Figure 8). These blow-ups are a result of the increase in difficulty of doing larger crosswords, which is significantly offset by being able to effectively choose the best variables to instantiate at different levels in the search.

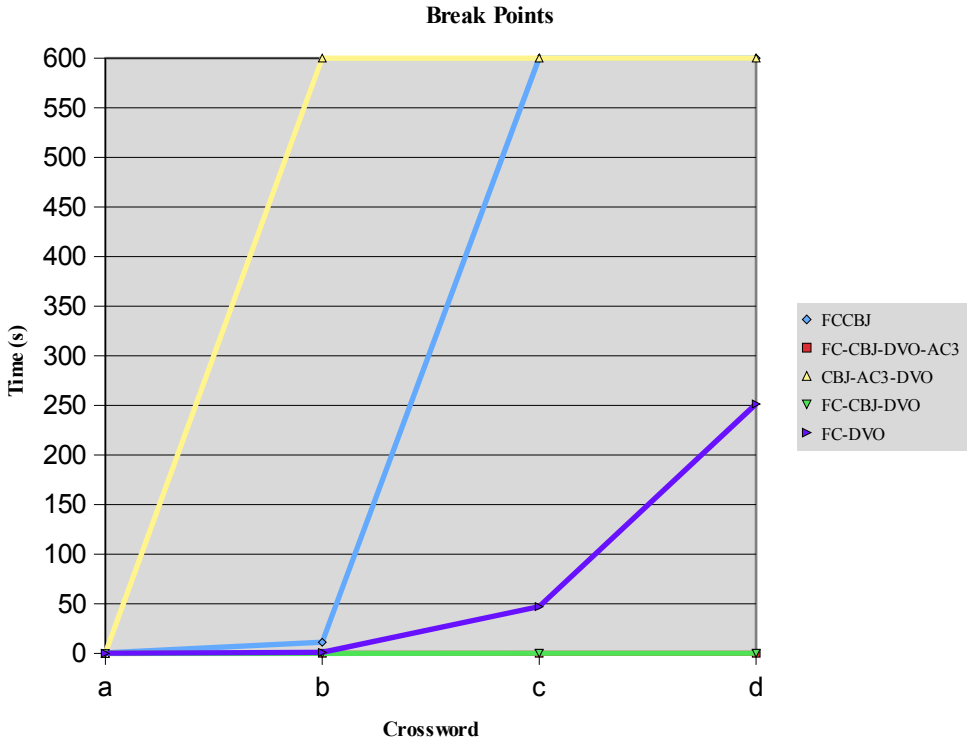


Figure 8: Cutoff points for solutions with different methods.

Discussion and Future Directions

In this report, we considered different strategies for solving constraint satisfaction problems. We found that in the crossword puzzle domain, algorithms running conflict-directed backjumping with forward checking and dynamic variable ordering proved very effective. Though CSPs are in the NP-Complete class of problems, fairly large instances can be solved through effective heuristics and strategies. There is, however, a break point for many of the algorithms after which they no longer are efficient CSP solvers.

Some algorithmic techniques we did not consider were backmarking and dynamic backtracking, which might be worthwhile. Dynamic backtracking might allow us to better do local search, because we would not have to unassign variables that we have assigned but need to backtrack over. In the crossword domain, this might prove effective, because different parts of crosswords are often separated by black squares from one another and do not effect each other much except for one or two connections.

Crossword puzzles provide a good and somewhat (some argue ridiculously or immensely) fun test bed to demonstrate the strengths of different CSP solving techniques.

References

[1] F. Bacchus and P. van Run. Dynamic Variable Ordering in CSPs. *Principles and Practice of Constraint Programming*, 258-275. 1995.

[2] M. Ginsberg, et al. Search Lessons Learned From Crossword Puzzles. *Proceedings of AAAI-90*. 1990.

[3] P. Hentenryck, Y. Deville, C. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*. Elsevier Science Publishers Ltd. Essex, 1992.

[4] G. Manku. Fast Bit Counting Routines. Available online at http://www-db.stanford.edu/~manku/bitcount/bitcount.html.

[5] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, Volume 9, Number 3. 1993.