Michael Gould

CompSci 275

Professor Dechter

December 7, 2010

Comparison of Value-Ordering Heuristics on Generating Diverse Solutions for Constraint Satisfaction Problems

I. Introduction

This project is based on the research paper entitled "Value-Ordering Heuristics: Search Performance vs. Solution Diversity" by Yevgeny Schreiber that appeared at CP 2010 and was published in LNCS 6308, pp. 429-444 by Springer, Heidelberg (2010).

For a given constraint satisfaction problem (CSP) we may want to find several solutions to the problem. We would also like that the solutions that we find be as different from one another as possible. Thus, we can explore using various value-ordering heuristics during search to decrease the amount of time it takes to find a solution as well as maintain a desired level of solution diversity.

II. Summary of Paper

We are familiar with how a *Constraint Satisfaction Problem (CSP)* is defined. There is a set of variables, a set of domains for the variables, and a set of constraints that define the allowed value combinations of the variables. A *solution* is an assignment of values to variables so that all of the constraints are satisfied. The *general solution search method* consists of (a) repeatedly selecting an unassigned variable x, (b) assigning to x one of the remaining values in its domain, and (c) propagating every constraint that involves x by removing conflicting values in the domains of other variables. If a variable domain becomes empty, then we backtrack and try again. A *value-ordering heuristic*

determines which value is selected by the backtracking search algorithm at step (b). On average, a solution can be found more quickly if the selected value maximizes the number of possibilities to assign variables that are still unassigned (i.e. has as little chance as possible of leading to a conflict).

The paper mentions several various existing value-ordering heuristics. For example, one class of such heuristics is called *survivors-first* where the heuristics use simple statistics accumulated during the search to select the value that has been involved in less conflicts than others. These heuristics are easy to implement, have low run-time and memory overhead, and can be quite helpful.

As mentioned in the introduction there are times when we want to find several solutions to a given problem and those solutions should be as different from each other as possible. It is important to note the difference between solution diversity and *solution distribution*. Consider a solution space composed of a small subset S1 of solutions where each variable is assigned a different value, and a much larger subset of solutions S2 where only a single variable is assigned a different value in each solution. A uniform sample of solutions would mostly contain solutions from S2, but we would like to find solutions from S1. Our problem can compare to the MaxDiverse*k*Set problem of computing k maximally diverse solutions of a given CSP. We need to define how two solutions are "different" from one another. One metric of difference is the *Hamming distance*. In this metric, given solution $s = \langle s_1,...,s_n \rangle$ and solution $s' = \langle s'_1,...,s'_n \rangle$ we define $H_i(s, s') = 1$ if $s_i \neq s_i'$ and $H_i(s, s') = 0$ otherwise, for $1 \leq i \leq n$. Then the Hamming distance is the sum of the H_i values.

A real-world example of a CSP where the solution diversity requirement is very

important is the *Automatic Test Generation Problem (ATGP)*. This problem is to automatically generate a valid test for a hardware specification, which is a sequence of a large number of instructions. These instructions have to be diverse in order to trigger as many different hardware events as possible. Computational time to find a solution to the ATGP problem is important because thousands of tests have to be generated for even a small subset of a hardware specification. There is a *tradeoff* between finding solutions quickly and finding solutions that are of high-quality (diverse). It is too computationally intensive to find the optimal solution to the MaxDiverse*k*Set problem so we can employ value-ordering heuristics to help us find a good solution.

The most basic value-ordering heuristic is the RANDOM heuristic, which selects a uniformly random value from the domain of the variable to be instantiated. Not only is the procedure itself of randomly selecting a value from a domain very fast, but the heuristic can lead to a solution rather quickly. It also often achieves relatively high solution diversity in problems that have many solutions because there are usually a large number of values in every domain whose selection does not lead to a conflict.

The paper proposes using six additional heuristics: (1) LeastFails, which assigns a variable x a value v with the lowest recorded number of attempts to assign v to x that have led to a conflict (and the removal of v from the domain of x); (2) BestSuccessRatio, which assigns x a value v with the lowest ratio of (number of attempts to assign v to x that have led to a conflict)/(the total recorded number of attempts to assign v to x); (3) ProbLeastFails, which is a probabilistic version of LeastFails; (4) ProbBestSuccessRatio, which is a probabilistic version of BestSuccessRatio; (5) ProbMostFails, which assigns x a value v with a probability that is proportional to the number of attempts to assign v to x

that have led to a conflict; and (6) ProbWorstSuccessRatio, which is the opposite of ProbBestSuccessRatio.

The paper discusses three parameters that are used to fine-tune the above *learning-reuse* heuristics. The first one, alpha, is used to give an "initial score" to an unordered value v since initially no information is available at the beginning of the search about any of the values. The beginning of the first search is equivalent to RANDOM. The second parameter, beta, is used with the probabilistic heuristics to effect the non-uniformness of the probabilities of selecting each value. The third parameter, gamma, is used in the first two heuristics as a *tie-range* parameter that can be used to smooth differences between sufficiently close values. For each of the six heuristics the paper also defines an *adaptive* version that does more computation during the search. The goal of the adaptive versions is to find a subset of variables that form a sub-problem that is difficult to satisfy and make the search more performance-oriented for the variables in the subset and more diversity-oriented for the rest of the variables.

Experiments were run on randomly generated problems and Automatic Test Generation problems. Each of the six heuristics and their adaptive versions are compared with the RANDOM heuristic. The results are presented in several tables in the paper. The results show that it is hard to achieve a better solution diversity than that of the RANDOM heuristic. Only a few heuristic configurations produce better solution diversity and not by much. On the other hand, there were many heuristic configurations whose solution diversity was much worse than that of RANDOM. When looking at the *runtime* of the different heuristics to that of RANDOM, there are many heuristic configurations that run much faster than RANDOM. However, when the speed-up is

very high, the loss of solution diversity is also very significant. There are several entries of heuristic configurations in each table that achieve significant speed-up and only a very slight loss in solution diversity.

III. Critique of Paper

This research paper raises several important questions. First, do the experimental results hold for larger problems? The randomly generated problems that were tested had only 50 variables and a maximum domain size of 10. When there are hundreds of variables or larger domains it would be interesting to know if the same results are achieved. Second, do other real-world problems besides the Automatic Test Generation problem achieve similar results? Since each CSP problem is unique in its own way, one heuristic may work better for certain problems while another heuristic works better for others. Third, since all of the problems were solved using the variable ordering heuristic that always picked the variable with the currently minimal domain we may ask if other variable orderings improve search performance and solution diversity. We may also want to use other variable selection heuristics in combination with the value selection heuristics. Does selecting variables at random or using a fixed variable ordering alter performance in any manner? Fourth, instead of using backtracking search would it be beneficial to use look-ahead mechanisms or apply some level of consistency like arc-consistency beforehand? The running time of finding multiple solutions may increase but solution diversity may improve. Finally, is there a consensus on which value-ordering heuristic is the best? The simple RANDOM heuristic may be good enough for some problems. Very few heuristic test cases produced a solution diversity that was greater than that produced by RANDOM. Thus we can ask if a performance speed-up is worth a

slight decrease in solution diversity. Since heuristic performance most likely depends on the particular CSP problem being solved we should consider which heuristics work best for which problem classes.

IV. Further Analysis

The research paper does not mention how the six value-ordering heuristics and their adaptive versions were implemented. It does not specify what programming language was used nor the specific technical details about what hardware and software was used to perform the tests. I was unable to find out these details despite searching for more information on the Internet.

As a result, I looked at a couple of currently available constraint satisfaction problem solvers to see if they had any built-in value selection heuristics. The first one that I looked at was Numberjack. Numberjack is a package in Python for constraint programming. Its standard CSP solver has a parameter tuning method called setHeuristic that sets the variable and value heuristics. However, I could not find adequate documentation about how to use that particular method. Therefore I next looked at JaCoP, a Java library for solving constraint satisfaction problems. The JaCoP solver has several simple variable and value selection methods built-in for finite domain variables (FDVs). They are the following:

- **value selection methods**

Indomain method	Description
`IndomainMin`	selects a minimal value from the current domain of FDV
`IndomainMax`	selects a maximal value from the current domain of FDV
`IndomainMiddle`	selects a middle value from the current domain of FDV and then left and right values
`IndomainRandom`	selects a random value from the current domain of FDV
`IndomainSimpleRandom`	faster than `IndomainRandom` but does not achieve uniform probability
`IndomainList`	uses values in an order provided by a programmer if values not specified uses default indomain method
`IndomainHierarchical`	uses indomain method based provided variable-indomain mapping

- **variable selection methods**

Comparator	Description
`SmallestDomain`	selects FDV which has the smallest domain size
`MostConstrainedStatic`	selects FDV which has most constraints assign to it
`MostConstrainedDynamic`	selects FDV which has the most pending constraints assign to it
`SmallestMin`	selects FDV with the smallest value in its domain
`LargestDomain`	selects FDV with the largest domain size
`LargestMin`	selects FDV with the largest value in its domain
`SmallestMax`	selects FDV with the smallest maximal value in its domain
`MaxRegret`	selects FDV with the largest difference between the smallest

The JaCoP solver uses a standard backtracking depth-first search using the specified variable and value selection methods. The solver can easily be asked to find one solution or all solutions to the problem. I attempted to implement a Java program using the JaCoP library to solve instances of the Magic Square and N-Queens problems that used various value selection methods. My plan was to compare the time required to find a solution to instances of various sizes of these problems and see which value selection heuristic found the quickest solution to the problem. However, I found that coding the constraints for both the Magic Square and N-Queens problems was rather difficult. The constraint formulations in JaCoP can be rather cumbersome to use. Thus, I did not finish implementing the program. It may be worth the effort to go back and complete the program.

I can conclude from my experiences that the implementation of various advanced heuristics to speed up the search for diverse solutions to a CSP can be difficult in practice. Thus for some problems it may not be worth the effort to implement more

complex heuristics for only a slight improvement in performance when a random value selection heuristic may work good enough. As various CSP solvers are improved and achieve better performance we should expect to see more use of these heuristics.

V. Conclusion

The value-ordering heuristics that were presented in the paper can be categorized according to their behavior: (1) LeastFails and BestSuccessRatio, which attempt to find a solution as fast as possible, usually achieve very high speed-up over random but are often accompanied by a significant loss of solution diversity. Their adaptive versions show similar behavior. Despite this, in the randomly generated problems the solution diversity loss was not as great as the speed-up gain. (2) ProbMostFails and ProbWorstSuccessRatio and their adaptive versions usually do not achieve a better solution diversity and can be much slower than Random. (3) ProbLeastFails and ProbBestSuccessRatio and their adaptive versions often achieve a moderate speed-up without sacrificing much solution diversity. The adaptive versions usually achieve somewhat higher solution diversity but are slower than the non-adaptive versions.

More testing is needed on different types of CSPs of varying sizes. There is no one value-ordering heuristic that works best for all problem instances.