Optimizing the Method for Finding an Optimal Solution of a Constraint Satisfaction Problem

Andrew Pagan
Computer Engineering
Ph.D. Jeffery Duffany
Computer Engineering Department
Polytechnic University of Puerto Rico

***Abstract* — Constraint satisfaction problems are the subject of intense research in both artificial intelligence and operations research. They are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. Often, they exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Most optimal solutions for a constraint satisfaction problem are found by a standard substitution and elimination technique, similar to the procedure used to solve systems of equations, with the decision function f(A)=max(A2). However, this method involves squaring the A matrix after each substitution, making it time consuming for larger matrices. This can be remedied using a different method that only requires A to be squared once, and subsequently updated thereafter.**

***Key Terms* — Algorithm, Complexity, Constraint, Matrix.**

INTRODUCTION

Constraint satisfaction problems represent the entities in a problem as a homogeneous collection of finite constraints over variables, which are solved by constraint satisfaction methods. Their regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. Examples of problems that can be modeled as a constraint satisfaction problem:

- Eight queens puzzle
- Map coloring problem
- Boolean satisfiability

Graph coloring will be used to demonstrate the effectiveness of the proposed method. In graph theory, graph coloring is a special case of graph labeling; it is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color; this is called a vertex coloring. Similarly, an edge coloring assigns a color to each edge so that no two adjacent edges share the same color, and a face coloring of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.

Vertex coloring is the starting point of the subject, and other coloring problems can be transformed into a vertex version. For example, an edge coloring of a graph is just a vertex coloring of its line graph, and a face coloring of a planar graph is just a vertex coloring of its planar dual. However, non-vertex coloring problems are often stated and studied as is. That is partly for perspective, and partly because some problems are best studied in non-vertex form, as for instance is edge coloring. Graph coloring enjoys many practical applications as well as theoretical challenges. Beside the classical types of problems, different limitations can also be set on the graph, or on the way a color is assigned, or even on the color itself. It has even reached popularity with the general public in the form of the popular number puzzle Sudoku. Graph coloring is still a very active field of research.

CONSTRAINT SATISFACTION

Constraint satisfaction problems (CSP)s are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time[1].

Examples of problems that can be modeled as a constraint satisfaction problem:

- Eight queens puzzle
- Map coloring problem
- Sudoku
- Boolean satisfiability

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exists. Backmarking improves the efficiency of checking consistency. Backjumping allows saving part of the search by backtracking "more than one variable" in some cases[2]. Constraint learning infers and saves new constraints that can be later used to avoid part of the search. Look-ahead is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value, thus sometimes determining in advance when a subproblem is satisfiable or unsatisfiable.

Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraints propagation has various uses. First, they turn a problem into one that is equivalent but is usually simpler to solve. Second, they may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and/or for some certain kinds of problems. The most known and used form of local consistency are arc consistency, hyper-arc consistency, and path consistency. The most popular constraint propagation method is the AC-3 algorithm[2], which enforces arc consistency. Local search methods are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem is satisfiable. They work by iteratively improving a complete assignment over the variables. At each step, a small number of variables are changed value[1], with the overall aim of increasing the number of constraints satisfied by this assignment. The min-conflicts algorithm is a local search algorithm specific for CSPs and based in that principle. In practice, local search appears to work well when these changes are also affected by random choices. Integration of search with local search have been developed, leading to hybrid algorithms.

GRAPH COLORING

In the field of distributed algorithms, graph coloring is closely related to the problem of symmetry breaking. The current state-of-the-art randomized algorithms are faster for sufficiently large maximum degree Δ than deterministic algorithms. A deterministic distributed algorithm cannot find a proper vertex coloring in a symmetric graph. Some auxiliary information is needed in order to break symmetry. A standard assumption is that initially each node has a unique identifier, for example, from the set $\{1, 2, ..., n\}$. Put otherwise, we assume that we are given an n-coloring. The challenge is to reduce the number of colors from n to, e.g., $\Delta + 1$. The more colors are employed, e.g. $O(\Delta)$ instead of $\Delta + 1$, the fewer communication rounds are required[3].

A straightforward distributed version of the greedy algorithm for $(\Delta + 1)$-coloring requires $\Theta(n)$ communication rounds in the worst case[4] − information may need to be propagated from one side of the network to another side. The simplest interesting case is an n-cycle. By iterating the same procedure, it is possible to obtain a 3-coloring of an n-cycle in $O(\log^* n)$ communication steps (assuming that we have unique node identifiers).

Graph coloring is computationally hard. It is NP-complete to decide if a given graph admits a k-coloring for a given k except for the cases k = 1 and k = 2. It is especially NP-hard to compute the chromatic number[4]. The 3-coloring problem remains NP-complete even on planar graphs of degree. The best known approximation algorithm computes a coloring of size at most within a factor O(n(log n)−3(log n)2) of the chromatic number. For all ε > 0, approximating the chromatic number within n1−ε is NP-hard. It is also NP-hard to color a 3-colorable graph with 4 colors and a k-colorable graph with k(log k) / 25 colors[3] for sufficiently large constant k.

Computational Complexity

In computational complexity theory, the complexity class NP-complete (abbreviated NP-C or NPC) is a class of decision problems. A decision problem “L” is NP-complete if it is in the set of NP problems so that any given solution to the decision problem can be verified in polynomial time, and also in the set of NP-hard problems so that any NP problem can be converted into L by a transformation of the inputs in polynomial time. Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today.

As a consequence, determining whether or not it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems in computer science today. While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using approximation algorithms. NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine[5]. NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if any single problem in NP-complete can be solved quickly, then every problem in NP can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete[4] (that is, it can be reduced in polynomial time).

Because of this, it is often said that the NP-complete problems are harder or more difficult than NP problems in general. A decision problem *C* is NP-complete if *C* is in NP, and every problem in NP is reducible to *C* in polynomial time[4]. *C* can be shown to be in NP by demonstrating that a candidate solution to *C* can be verified in polynomial time. A problem *K* is reducible to *C* if there is a polynomial-time many-one reduction, a deterministic algorithm which transforms any instance ***k€K*** into an instance ***c€C***[5], such that the answer to *c* is yes if and only if the answer to *k* is yes. To prove that an NP problem *C* is in fact an NP-complete problem it is sufficient to show that an already known NP-complete problem reduces to *C*. Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1. A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for *C*, we could solve all problems in NP in polynomial time. At present, all known algorithms for NP-complete problems require time that is super-polynomial in the input size, and it is unknown whether there are any faster algorithms.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution, search for an "almost" optimal one.
- Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. See Monte Carlo method.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.
- Heuristic: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and produces good results. Meta-heuristic approaches are often used.

A substitution method also offers a significantly more efficient method for finding an optimal solution for computational problems. We can demonstrate this by simply incrementally updating the matrix.

BIG-O

In mathematics, computer science, and related fields, big-O notation describes the limiting behavior of the function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

Although developed as a part of pure mathematics, this notation is now frequently also used in the analysis of algorithms to describe an algorithm's usage of computational resources: the worst case or average case running time or memory usage of an algorithm is often expressed as a function of the length of its input using big O notation. This allows algorithm designers to predict the behavior of their algorithms and to determine which of multiple algorithms to use, in a way that is independent of computer architecture or clock rate.

Because big O notation discards multiplicative constants on the running time, and ignores efficiency for low input sizes, it does not always reveal the fastest algorithm in practice or for practically-sized data sets[6], but the approach is still very effective for comparing the scalability of various algorithms as input sizes become large. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols o, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates[6]. Big O notation is also used in many other fields to provide similar estimates.

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function f(x) is derived by the following simplification rules:

- If f(x) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If f(x) is a product of several factors, any constants (terms in the product that do not depend on x) are omitted.

PROPOSED METHOD

Using a pre-defined graph coloring R type language package, we will develop a more optimized way of returning a solution vector for a system of inequations. As explained by Duffany[1], “The ineq algorithm starts with a solution vector s which has an initial value of s = (1,2,3,...n). The algorithm squares the adjacency matrix A and finds the maximum value of A2 [i,j] for pairs of variables that can be combined (i.e., A[i,j]=0). It then combines variables xi and xj by taking the constraints that are in xj but not in xi and adding them to xi (xi=xi|xj) where | = logical OR. Then it updates the solution vector s[j]=s[i] and eliminates variable xj (i.e., remove row and column j from A as represented by the line A=A[-j,-j]). The matrix A is reduced by one in dimension each time a variable is eliminated.”

We know that squaring the adjacency matrix is $O(n^3)$. Having to do this on each iteration is (O(n)). This means that the entire algorithm has a notation of $O(n^4)$. This is not very efficient and can become time consuming with large matrices. Furthermore, each elimination only adds relatively few constraints to another variable, and in turn, becomes a very small change for the A matrix. We propose a different solution, one that, in theory, would make the ineq algorithm be $O(n^3)$[7]. Instead of calculating A^2, we can simply *update* it. That is, we calculate A^2 once, and then add it to an incremental matrix. With this method, we see some aspects of parameterization. The equation for the proposed method is as follows:

$$A^2 + A*\Delta A + \Delta A*A + (\Delta A)^2 \qquad (1)$$

The logic for (1) comes from the original equation itself:

$$A'^2 = (A + \Delta A)^2 \qquad (2)$$

From (1) and (2), we see that each of these operations can be calculated in $O(n^2)$, thus making this entire equation $O(n^2)$, $O(n^3)$ at worst.

R LANGUAGE

We used the RGUI environment (freely distributed online) to run the standard operations for calculating a solution vector for a system of inequations, along with our updated algorithm, and to perform runtime tests on both algorithms to determine if our proposed method is faster. R is a programming language and software environment for statistical computing and graphics. The R language has become a standard among statisticians for developing statistical software, and is widely used for statistical software development and data analysis. R is an implementation of the S programming language combined with lexical scoping semantics inspired by Scheme. R is part of the GNU project. Its source code is freely available under the GNU General Public License[8], and pre-compiled binary versions are provided for various operating systems. R uses a command line interface; however, several graphical user interfaces are available for use with R.

R provides a wide variety of statistical and graphical techniques, including linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, and others. R is easily extensible through functions and extensions, and the R community is noted for its active contributions in terms of packages. There are some important differences, but much code written for S runs unaltered. Many of R's standard functions are written in R itself, which makes it easy for users to follow the algorithmic choices made. For computationally intensive tasks, C, C++, and Fortran code can be linked and called at run time. Advanced users can write C or Java code to manipulate R objects directly.

R is highly extensible through the use of user-submitted packages for specific functions or specific areas of study. Due to its S heritage, R has stronger object-oriented programming facilities than most statistical computing languages. Extending R is also eased by its permissive lexical scoping rules. Another strength of R is static graphics, which can produce publication-quality graphs, including mathematical symbols. Dynamic and interactive graphics are available through additional packages such as RGL. R has its own LaTeX-like documentation format[8], which is used to supply comprehensive documentation, both on-line in a number of formats and in hard copy.

As a programming language, R is a command line interpreter similar to BASIC or Python. If one types "2+2" at the command prompt and presses enter, the computer replies with "4". This example is deceptively simple because, like APL, R implements matrices, so from the command line R can add or even invert matrices without explicit loops[8]. R's data structures include scalars, vectors, matrices, data frames (similar to tables in a relational database) and lists. The R object system has been extended by package authors to define objects for regression models, time-series and geo-spatial coordinates. The capabilities of R are extended through user-created packages, which allow specialized statistical techniques, graphical devices, import/export capabilities, reporting tools, etc. These packages are developed primarily in R, and sometimes in Java, C and Fortran. Reproducible research and automated report generation can be accomplished with packages that support execution of R code embedded within LaTeX, OpenDocument, format and other markups.

R supports procedural programming with functions and object-oriented programming with generic functions. A generic function acts differently depending on the type of arguments it is passed. In other words, the generic function recognizes the type of object and selects (dispatches) the function (method) specific to that type of object. For example, R has a generic print() function that can print almost every type of object in R with a simple "print(objectname)" syntax.

Although R is mostly used by statisticians and other practitioners requiring an environment for statistical computation and software development, it can also be used as a general matrix calculation toolbox with performance benchmarks comparable to GNU Octave or MATLAB.

COMPUTATIONS

For these calculations, we used a variety of sample sizes. In this case, "N" refers to the size of a matrix NxN. For example, a N=500 chart would refer to a 500x500 matrix. We used three different sample sizes, each with a great difference in size, in order to have a clear picture of the differences between the methods. We also ran each sample ten times for the sake of certainty, using a different randomly generated matrix for each attempt (Note: the matrix varied across attempts, not methods. For instance, the matrix used in the first attempt in the first sample size is the same matrix used in the first attempt for the second method). Accuracy is also important, meaning we also verified that the resulting vectors were the same. In Figure 1, we see the results from using a 1000x1000 matrix:

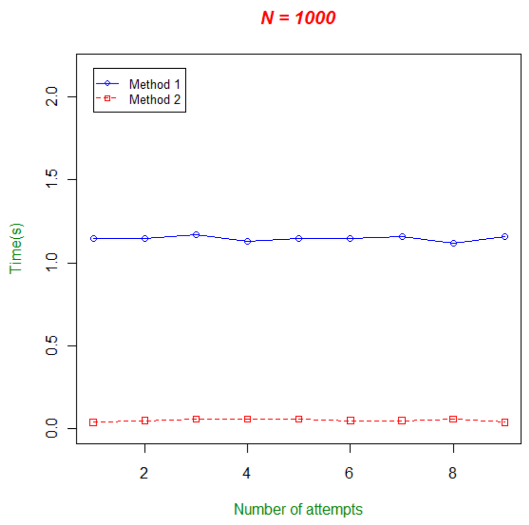

Figure 1
1000x1000 Matrix

The first method (represented in blue), in each attempt, is about one second slower than the second (represented in red), with the second method yielding almost instant results. Even with such a small sample, this already shows a considerable difference between the speeds of each method. Figure 2, using a matrix of 2500x2500, further demonstrates the power of our proposed method.

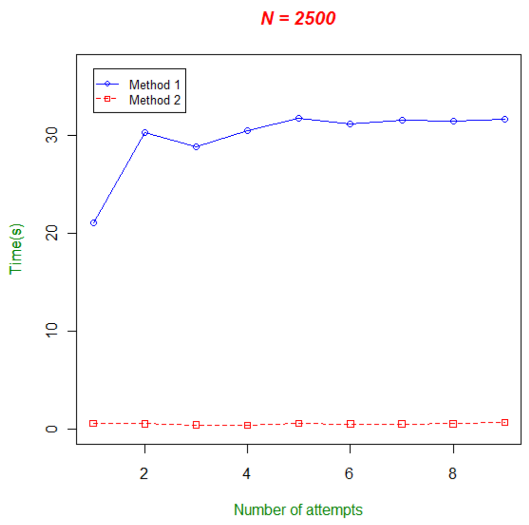

Figure 2
2500x2500 Matrix

Whereas the first method increases exponentially in time taken to complete the calculations, taking around half a minute to complete, the second method remains significantly low, barely breaking the one second mark. Noticeably, the first attempt using the second method was faster than the rest of the attempts. It is unknown what the reasoning behind this may be. A possibility may have been the randomly generated matrix that was used, or simply human error. In Figure 3, we see a fairly large matrix size and what is, arguably, the real demonstration of the substitution method's efficiency.

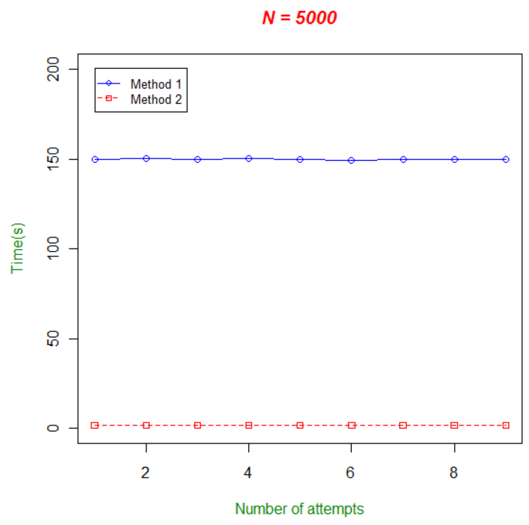

Figure 3
5000x5000 Matrix

Once again, we see an exponential increase in the time taken to calculate the solution vector with the multiplication method, and a very miniscule increase with the substitution method. In fact, the time to calculate increases by almost five times the amount. It is safe to assume that this pattern will continue with larger matrices. Thus, no further testing is required. We must note that, after each attempt, we verified to make sure that the solution matrices given for both methods were, in fact, correct and equal to each other.

CONCLUSION

From the results, we can see that a viable solution for a number of constraint satisfaction problems can be found using substitution and elimination of variables in place of the standard multiplication technique. The simplicity and effectiveness of the algorithm is incredible, as it is able to find an optimal solution with little effort. Ideally, we would be able to see the algorithm perform at a much larger sample size, but the tests were limited to the technology that was readily available. Even so, it is easy to see that even for a massive matrix, this method would perform exceptionally well.

The substitution and elimination method trumps the original equation, being that the original needs to square the updated A matrix after each variable elimination. This is significantly different from our proposed method, in that the matrix A in the decision function only needs to be squared once at the beginning of the algorithm. Afterwards, for all subsequent steps, it only needs to be incrementally updated, which, as we've clearly seen, results in significant improvements, and reduces the overall algorithm complexity to O(n^3). Needless to say, this idea of transforming these types of problems in a system of inequations is a substantial change, and may provide a different viewpoint on other mathematical problems.

REFERENCES

[1] Duffany, J.L., "Optimal Solution of Constraint Satisfaction Problems", *International Conference on Applied Computer Science,* January 2009.

[2] Chen, H. "A Rendezvous of Logic, Complexity, and Algebra". *ACM Computing Surveys (ACM,).* 42, December 2009, pp 1–32.

[3] Duffy, K.; O'Connell, N.; Sapozhnikov, A., "Complexity analysis of a decentralised graph colouring algorithm", *Information Processing Letters*, 107, 2008, pp 60–63.

[4] Duffany, J.L., "Statistical Characterization of NP-Complete Problems", *Foundations of Computer Science Conference*, July 14 2008.

[5] Hopcroft, J.E., Motwani, R. and Ullman, J.D, "Introduction to Automata Theory, Languages, and Computation", *Addison Wesley*, 2007, p 368.

[6] Michael, S., “Introduction to the Theory of Computation”, *PWS Publishing,* 1997, pp 226–228 of section 7.1: Measuring complexity.

[7] Duffany, J.L. “Systems of Inequations”, *4th LACCEI Conference,* June 21 2006.

[8] Homik, K., "R FAQ." The R FAQ. 2011, Retrieved from http://cran.r-project.org/doc/FAQ/R-FAQ.html