Constraints: An International Journal, 2, 75–82 (1997)

Towards a Constraint-Based Engineering Framework for Algorithm Design and Application

SIMON KASIF kasif@eecs.uic.edu

Department of Electrical Engineering and Computer Science, University of Illinois, Chicago, Illinois 60607

Abstract. The search for programming frameworks that provide the formalism that can help organize literally thousands of algorithmic ideas and facilitate their applications to new problems is an important research direction in computer science. That is, we are seeking to develop a practical programming environment that allows the user to state a family of problems, and subsequently systematically apply a collection of established algorithmic techniques to individual problems correctly. In this note we informally discuss constraint networks as a vehicle that provides a general framework for the synthesis, abstraction and application of problem-specific algorithms.

Keywords: constraints, program synthesis, efficient algorithms

1. Introduction

Design and analysis of new efficient algorithms and the development of new programming languages are two central research topics in computer science. The area of programming languages focuses on important issues such as universality (the ability of a programmer to express any algorithm), semantic correctness (e.g., the verifiability of a program with respect to a logical specification), and efficiency of (typically low-level) programming constructs.

Algorithms research emphasizes the design of sophisticated data structures and algorithms for very specific problems. Algorithms have been developed over the years for combinatorial problems, graph problems, statistics, sequence comparison and analysis, computational geometry, optimization, and the list goes on. In particular, general algorithmic schemas such as dynamic programming, binary search, divide-and-conquer, local search and many others can be found in standard textbooks. For a non-expert in algorithm design, however, the area of algorithms appears as a collection of very clever, however mysterious tricks. Thus, while most specific algorithms are often inspired by a general technique, it is typically difficult (especially for non-experts in algorithm design) to precisely formalize the abstraction of an algorithmic technique to a family of problems or extend a general algorithmic principle to a new problem. Novices typically find it difficult to develop the right algorithmic solution for a new problem. What is even more troubling is the fact that even coding a well-known algorithmic paradigm for a given new problem is surprisingly challenging for most. E.g., we do not have programming environments that provide tools to apply a specific algorithmic paradigm such as divide-and-conquer technique to a family of problems without the need to explicitly code it for each member of the family.

The problem of abstracting an algorithmic technique to a family of problems (rather than just one problem) and devising a framework that allows the correct application of a

technique to every problem in the family is an open research problem that we address in this note [1].

That is, we are seeking to develop a practical programming environment that allows the user to state a family of problems, and subsequently systematically apply a collection of established techniques (abstract algorithmic schemas) to individual problems correctly. Such a framework would have to develop a programming language methodology to devise a programming notation for a family of problems, as well as developing abstract algorithmic techniques that can applied to individual specifications.

In this note we (very informally) discuss constraint networks as a vehicle that provides a general framework for the synthesis, abstraction and application of problem-specific algorithms. [2]. The ultimate goal of this framework is to develop a programming environment that will allow us to develop general algorithmic techniques that subsequently can be called on for application to a broad family of problems stated in a convenient programming notation.

One of the important practical consequences of a framework of this type would of course be the capability of developing a collection of software libraries that automatically (using minor variants of the same algorithm) can be used to solve different problems. While much of the work in declarative programming has been dedicated to searching for such a programming notation, this goal has generally been elusive.

The area of design and analysis of domain specific software libraries that are applicable only to a limited domain is relatively dormant in basic computer science research (with some exceptions such as linear or symbolic algebra packages). However, as computer science matures one expects more and more specialized software packages to be developed. One of the more interesting research aspects of this suggestion is that it allows us to marry principles obtained in programming language research with ideas and techniques developed in algorithms research. The search for programming frameworks that provide the formalism that can help organize literally thousands of algorithmic ideas and facilitate their applications to new problems (algorithm synthesis) is an important research direction that we must address.

2. Constraint Networks

The field of constraint networks is focused on producing a library of efficient procedures to solve general constraints over abstract data types. Research in constraint networks integrates programming language methodology to develop a programming language notation for expressing general constraint expressions, and algorithmic ideas to compile these expression into efficiently executable specifications. Once a particular engineering problem has been coded as a constraint network, the application of a particular algorithmic technique usually results in an efficient algorithm for the given problem.

Constraint networks research has been generally influenced by the methodologies of applied mathematics and traditional engineering disciplines such as control theory and operations research (OR). It is therefore useful to introduce constraint networks in this perspective, shedding light on the similarities and differences of the fields.

Traditional engineering and applied mathematics disciplines often follow a common prescription for a research methodology. For instance, the field of differential equations provides a formal notation that can be used to express a large family of problems. Then researchers define various restrictions of the general notation that allow them to develop either closed form solutions or efficient iterative algorithms to solve the equations. In many cases the form of the notation is dictated by an important application in physics, biology or engineering. In other cases the form is dictated by properties that allow an efficient solution.

In Operations Research (OR), a specification often starts from specifying a finite collection of variables $X = \{X_i, X_2, \ldots, X_n\}$. The variables are typically real-valued. Then we are provided with formal syntax for expressing a set of constraints over these variables. These constraints typically take the form of numerical inequalities of the form $f(X) < 0$ where f is a multi-variate function. We typically strive to find an assignment to variables that maximizes some function $g(X)$ subject to the constraint $f(X) < 0$. The most popular form of this methodology is the area of linear programming where both f and g are linear functions.

The area of constraint networks follows a similar prescription, however it strives to define a family of general algorithms that will be applicable to a broad family of constraint expressions over general data types.

The formalism of constraint networks typically provides a programming notation that allows the engineer to specify a wide range of problems such as scheduling, resource allocation, planning, computer vision, intelligent interfaces to computer graphics, system design and diagnosis, etc. As in OR we typically start by specifying a set of variables. However, here we take the first detour from traditional applied mathematics. We often allow substantial flexibility in defining the domain of each variable. The most popular class of constraint networks deals with variables over discrete domains, namely variables that can be assigned to integers. That is with each variable X_i we associate a domain D_i which is a set of possible integer-valued assignments to X_i. However, this scenario is a special case. We often deal with variables that are defined over finite sets, sequences, vectors, temporal intervals, rectangles, polygons, general graphical objects, or even abstract data types such as functions or programs.

To specify constraints we again generalize the methodology of OR and allow somewhat general constraint expressions over the variables. Naturally, the notation used to specify the constraints must be consistent with the data types of the variables. A common notation of constraints is given by tables that provide a short notation for listing the allowable assignments. However in many cases we simply rely on user-defined parameterized procedures to specify the allowable assignments. For instance, for graphical interface systems we may define constraints such as "the distance between two rectangles is at least two centimeters."

An important aspect of constraint networks that we would like to put forth in this note is that the framework allows us to apply abstract algorithmic schemas to a constraint network, irrespective of the particular data type of the variables or the nature of the constraints. Thus, once we code a particular problem into the constraint network we obtain a problem-specific algorithm automatically without the need to code this algorithm explicitly for a specific problem domain.

To illustrate the utility of this approach we shall review a few commonly used constraint solving paradigms and their application in generating specific problem-dependent algorithms. Before we "jump" into this topic we want to comment on the historical evolution of constraints.

It should be rather clear from the observations above, that the modern view of constraints is a natural generalization of mathematical equations. In particular, many ideas that are currently used in symbolic constraint solving have deep roots in traditional equation solving or related areas such as operations research. Mathematical equations were originally developed in response to the need to model complex physical, economic or engineering systems. Because computers were not available (or even before they became extremely efficient and storage rich), the work on mathematical equation solving naturally focused on linear systems, primarily motivated by either mathematical elegance or the search for closed-form solutions that can be derived by hand by following simple procedures. However, since the advent of high-performance computers that can manipulate complex expressions, it became evident that we can expand and generalize mathematical equations to very general constraints. This holds as long as we have effective means to find solutions to systems of constraints. As a result, many procedures used in constraint solving often have counterparts in traditional mathematics. E.g., local search is certainly reminiscent of gradient descent and partial constraint solving is inspired by Gauss-Seidel solutions of linear equations or Penrose's inverse. This note is actually trying to encourage researchers to significantly broaden the family of algorithmic procedures used in constraint solving beyond very general procedures (see below).

Finally, we comment on the reason the Artificial Intelligence (AI) community was the first to discover the utility of symbolic constraints. AI researchers were the first implementors of models of complex systems using qualitative methods that were not based on traditional approaches such as differential equations or linear programming. As a result, AI papers were the first to explicitly state the problem of manipulating relational constraints over symbolic domains. This resulted in ground breaking work such as Waltz's filtering algorithms in early computer vision and later application of constraints in qualitative physics. However, today researchers across many disciplines of computer science and engineering (e.g, robotics, graphics, architecture, control) perform some form of mixed qualitative/mathematical modeling of hybrid complex systems and processes and as a result the use of general constraints became more widely spread.

3. Local Search in Constraint Networks

Local search in constraint networks is simply the following iterative technique.

- Order the variables.
- For each variable X_i, $i = 1, 2, \ldots$, we dynamically designate C_i to be the set of possible assignments that currently are being considered for X_i. Note that for standard local search C_i is typically the set of all possible assignments to a variable.

- Define the "happiness" of a constraint network to be some easily computable function of the current assignment and the set of constraints.

- For each variable X_i, iterate over all possible assignments (elements of C_i) and find an assignment that maximizes the happiness of the network.

Local search is of course inspired by traditional applied mathematics techniques and can be seen as a discrete version of gradient descent. A well known variant of local search for problems involving permutations is an approach where we exchange the values of X_i and X_{i+1} if the local constraint on these variables is not satisfied.

4. Applications of Local Search

4.1. Sorting

Consider the problem of sorting a set S of N unique integers. One standard algorithm for the problem is **bubble-sort**. We start from a random permutation of the input numbers. Bubble-sort repeats N phases of "bubbling", namely we compare element X_i to element X_{i+1} and exchange the two values if the order is wrong. We can easily state a variant of this algorithm as an application of "bubble-local search" to the appropriately defined constraint problem.

We define n variables, where the domain of each is the set of integers to be sorted.

- Order the variables.

- Let the constraint $R(X_i, X_{i+1})$ be the relation $X_i < X_{i+1}$ for each $i < N$.

- Let the domain D_i of variable X_i be S for each i.

- Define "happiness" of a constraint network to be number of "satisfied" constraints.

- Iterate over the variables in increasing order exchanging X_i and X_{i+1} if the constraint R is not satisfied.

It is easy to see that a complete pass improves the "happiness" of the network, i.e., the total number of satisfied constraints is increased by at least one. Since the happiness is bounded above, the algorithm must converge. When the elements are unique there is only one local minimum, which happens to be the global minimum, thus the we reach the desired solution. However, note that we have defined a general version of "bubbling" in constraint networks that can be applied to any chain constraint network seeking a permutation that satisfies a set of given constraints. Thus if a user can formulate a problem as a chain constraint network, it would be possible now to apply the "bubble-satisfy" algorithm find a permutation of the input that satisfies a set of given constraints. We can now seek to establish the conditions on R that will guarantee the convergence of the "bubble-satisfy" algorithm to a global minima.

5. Divide and Conquer in Constraint Networks

Divide and conquer is another well established idea in algorithm design. One possible special case of the divide-and-conquer paradigm can be abstracted by the following general technique:

- Pick a set of variables (we call them a separator).
- This separator set usually partitions the remaining variables into two or more subsets of variables V_1 and V_2 that do not share constraints anymore.
- Instantiate the separator set to a set of assignments C_i.
- For each assignment in C_i, recursively solve the two constraint problems on the variables in V_1 and V_2.

5.1. Applications of Divide and Conquer

To illustrate the abstraction of divide and conquer in a sorting context we first sketch a somewhat inefficient abstraction of a divide and conquer. Again, we start with N variables. The set of possible assignments for each variable, i.e., the domain of each variable is the set of unique integers to be sorted.

We apply the general version of divide-and conquer for chain networks. That is, we iterate over all possible assignments to variable $X_{k/2}$. For each assignment we recursively solve two constraint problems, one on variables X_i, $i < k/2$ and one for X_j, $j > k/2$. One possibility to improve this is to find the correct assignment for variable $X_{k/2}$ (the median) which is not immediately obvious.

The algorithm above is not a particularly efficient algorithm. We therefore suggest a different algorithm which is inspired by quicksort and abstracted to general chain constraint networks. Let R be the constraint on variables X_i and X_{i-1}. For the sorting problem, there is an induced constraint R on X_i and X_j, $j < i$.

- Randomly guess an element of the domain x.
- Compute the set of elements S_1 such that y is in S_1 iff $R(y, x)$ (e.g, $y < x$).
- Compute the set of elements S_2 such that y does not satisfy the constraint above.
- Let K be the size of S_1. Then we assign x to X_{K+1} and solve the problem recursively on the subchains on the left and right of X_{K+1} with domains S_1 and S_2 respectively.

As before, we managed to abstract a specific algorithm (quicksort) to chain networks. This algorithm would clearly be applicable to find a consistent solution (permutation) to chain networks provided the induced constraint on X_i and X_j, $j < i$ is equal to R.

6. Combining Techniques

We can also define new algorithmic schemas by combining previously defined techniques. As an example, consider the following algorithm constructed by integrating divide-and-conquer with local search.

- Pick a set of variables (we call them a separator).
- This separator set V_s partitions the remaining variables into two or more subsets of variables V_1 and V_2 that do not share constraints anymore.
- Iteratively run a local search algorithm only on the constraint network defined by V_s.
- For each assignment to variables in V_s we recursively solve the two constraint problems on the variables in V_1 and V_2.

Thus instead of iterating over all possible assignments for the separator set V_s in some blind order, we iterate over these assignments using local search. Thus we create a tree where each node corresponds to a local search process that "feeds" assignments to the local searches recursively generated by this process.

7. Summary

In this note we sketched an approach for synthesis, design and abstraction of algorithms in the constraint network framework. We showed that we can abstract certain problem-specific algorithms to systematic techniques that subsequently can be applied more generally to a family of problems provided these problems can be stated in the appropriate form. This note did not focus on automatic synthesis of algorithms such as quick-sort or merge-sort. Instead we outlined a framework where the algorithm designer can abstract algorithmic techniques and make them readily applicable to a family of problems. We discussed constraint networks as providing one possible programming notation that supports this algorithm desgn paradigm.

However, this approach may lead to an environment where we maintain a collection of software libraries and the programmer can "simply" call a particular technique (e.g, dynamic programming) and apply it generally to a family of problems. The programmer also can write simple scripts combining applications of two or more previously defined procedures (as in the previous section). Our approach generally suggests a greater focus in basic computer science research on software libraries. Our vision of such research will include:

- Developing a somewhat restricted programming notation (such as constraint networks) that can be used to specify a family of problems.
- Developing general algorithmic techniques stored as preprogrammed and parameterized software routines (such as dynamic programming, divide and conquer, local search, greedy assignment algorithms).

- A simple programming notation that would allow programmers to write simple scripts that apply these library routines to the specification.

One obvious problem we have not discussed is the process of transforming a specific input to an appropriate constraint network. This problem exists for most software libraries. For instance, linear programming packages expect input in a very particular format. One often needs to write small scripts that perform these transformations. This process is usually less tedious and time-intensive than the process of coding a specific algorithm. Even if one does not subscribe to the methodology advocated in this note, the constraint network formalism provides a convenient framework to state several well known algorithms from a unified perspective and is therefore useful as an instructional device in the study of algorithm design.

Notes

1. In software engineering there is a similar concern, namely software reuse. We are primarily concerned with the derivation of relatively short but still intellectually challenging algorithms, rather than very long programs where the complexity often stems from the intricate interaction of the components (e.g, functions, modules)
2. We attempt to keep the discussion in this position paper on a very high non-technical level and therefore omit references. The reader is encouraged to refer to the survey paper in this issue for a thorough literature review