A Lazy Divide & Conquer Approach to Constraint Solving

Saswat Anand Wei-Ngan Chin Siau-Cheng Khoo
School of Computing
National University of Singapore
E-mail: {saswat,chinwn,khoosc}@comp.nus.edu.sg

Abstract

Divide and conquer strategy enables a problem to be divided into subproblems, which are solved independently and later combined to form the solutions of the original problem. In solving constraint satisfaction problems, however, divide and conquer technique has not been shown to be effective. Because, it is not possible to cleanly divide a problem into independent subproblems in the presence of constraints that involve variables belonging to different subproblems. Consequently, solutions of one subproblem may prune some solutions of another subproblem, making those solutions of the latter subproblem redundant. In this paper, we propose a divide and conquer approach to constraint solving in a lazy evaluation framework. In this framework, a subproblem is solved on demand, which eliminates redundant consistency checks. Moreover, once solved, the solutions of a subproblem can be reused in the satisfaction of various global constraints connecting this subproblem with others, thus reducing the search space. We also demonstrate the effectiveness of our algorithm in solving a practical problem: finding all instances of a user-defined pattern in stock market price charts.

1 Introduction

Constraint Satisfaction Problems (CSP) have drawn a great deal of attention because of their simple and general definitions and wide application in diverse fields. For the past two decades, there have been numerous improvements of the naive backtracking algorithm, which was proposed a century before. Some of the popular ones are, backjumping, backmarking, forward checking, arc consistency, and of course their various combinations and variants.

However, none of the algorithms described has adopted divide and conquer (**d&c**) paradigm. Although there are schemes for identifying constraint networks that are decomposable [3, 8], these works restrict the manner in which a problem can be decomposed.

The main reason for *not* expressing constraint solving algorithms in d&c paradigm is the difficulty in cleanly dividing a constraint satisfaction problem into independent subproblems, with no constraint inter-connecting these subproblems. If global constraints that inter-connect subproblems exist, a naive solution, in the spirit of d&c, is to temporarily ignore these constraints, solve the subproblems independently, and later check for consistency of those global constraints. This method might cause many unnecessary consistency checks during solving of subproblems. Specifically, the existence of global constraints inter-connecting two subproblems might make their solutions incompatible.

On the other hand, such "over-checking" of constraints on a subproblem can be eliminated by working on the subproblems *only on demand*. Programming-wise, this amount to solving problems under *lazy evaluation*.

Lazy evaluation is a notion that comes from functional programming language like Haskell [10], where a computation is performed if and only if its result is required for further computations. For example, consider the following definition of a function `f` with parameter `x` and function body that always returns `2`.

```
fun f x = 2
```

When the function `f` is called, its actual argument – which might be a complicated expression – is never evaluated because it is not needed for calculating the result of `f` (which is `2`.)

In this paper, we describe a lazy d&c strategy for constraint solving, and show that this strategy can significantly reduce the number of consistency checks. Furthermore, we demonstrate the use of this framework in a practical problem: finding all instances of a user-defined pattern in stock market time series.

2 Background

We formulate constraint satisfaction problems as constraint networks. A *constraint network* consists of a set of *variables* $X = \{x_1, \ldots, x_n\}$, *domains* $D = \{D_1, \ldots, D_n\}$,

$D_i = \{v_1, \ldots, v_m\}$ and *constraints* $R_{S_1}, \ldots, R_{S_k}$, where $S_i \subseteq X$, $1 \leq i \leq k$. For any subset of variables $S = \{x_1, \ldots, x_r\}$, the *domain* of S is $D_1 \times \ldots \times D_r$. A *constraint* is a pair (R, S) where S is a subset of variables $S \subseteq X$, also called its *scope* and R is a relation defined over the domain of S, whose tuples denote the legal combination of domain values. The pair (R, S) is also denoted R_S. The *arity* of a constraint R_S is defined as the cardinality of S.

Given a set of variables $Y \subseteq X$, we define $\mathcal{K}_Y$ as follows:

$$\mathcal{K}_Y = \bigcup \{R_{S_i} \mid S_i \subseteq Y, 1 \leq i \leq k\}.$$

As an example, an n-queen problem can be formulated as a constraint network consisting of a set of variables $X = \{q_1, \ldots, q_4\}$, domains $D = \{D_1, \ldots, D_4\}$ such that $D_i = \{1, 2, 3, 4\}$, for $1 \leq i \leq 4$, and constraints $\bigcup_{1 \leq i < j \leq 4} R_{\{q_i, q_j\}}$, where $R_{\{q_i, q_j\}}$ specifies that a queen at row i cannot attack another queen at row j. Furthermore, $\mathcal{K}_{\{q_1, q_2, q_3\}} = R_{\{q_1, q_2\}} \cup R_{\{q_1, q_3\}} \cup R_{\{q_2, q_3\}}$.

3 Motivation for Lazy D&C

In this section, we give the motivation for using lazy d&c strategy through simple examples. Consider Figure 1, which shows the partial search space for 4-queen problem. There are 4 variables in the problem: q_1, q_2, q_3 and q_4, each represents a level in the search space, and the first level (the root) contains a dummy node. A node in the tree at level i with value v represents the assignment $q_i = v$.

A solution is represented by a complete path in the corresponding solution tree, joined by solid edges. A complete path is one that contains assignment to all variables in the subproblem. The only complete path shown in the figure contains the assignment $q_1 = 3$, $q_2 = 1$, $q_3 = 4$, and $q_4 = 2$. Moreover, a dashed edge joined a node at level i to another at level $i + 1$ represents an assignment of a value to the variable at level $i + 1$ that fails a consistency check. Consequently, no further search is carried out starting from the node of each dashed edge.

Two nodes labeled a and b in the tree have value 1, indicating that the 2nd queen (q_2) is positioned at column 1. The search spaces rooted at a and b are identical, as both of them are associated with three constraints namely, $R_{\{q_2, q_3\}}$, $R_{\{q_2, q_4\}}$, and $R_{\{q_3, q_4\}}$, with q_2 being assigned the value 1. Comparing the subtrees rooted at a and b, we see that two dashed edges, joining pair of assignments ($q_2 = 1, q_3 = 1$) and ($q_2 = 1, q_3 = 2$) respectively, are explored and discarded at both subtrees. This re-computation can be avoided if we can save our exploration result at a and reuse it at b.

However, backtracking algorithm, or any of its intelligent siblings like backjumping or backmarking [6], do not achieve the desired saving. Backmarking comes close to meeting our expectation; it does not redundantly check the constraints involving a node living in the subtree and the common ancestors of nodes a and b. However, in this example, backmarking does not eliminate redundant checks, as the common ancestor of nodes a and b is just the dummy root node.

In general, caching a solution subtree for future reuse can reduce the number of consistency checks drastically, since the number of identical nodes(nodes with same assignments) occurring at a level can be potentially exponential, resulting in exponential number of identical subtrees. Every time a solver comes to such a node, like a in level q_2 of Figure 1, it has to check the constraints involving variables that lie in or below level 2 of the tree. In the worst case, the complete subtree might get re-evaluated.

We can view the subtree rooted at a as representing the search space for a variant of 3-queen problem (with domain values ranged from 1 to 4). In the spirit of divide-and-conquer strategy, we wish to solve this subproblem and combine its result with those obtained from other subproblems. In many traditional problems, such as sorting, we can intuitively divide a problem into independent subproblems, and merge the individual results to form the final solution. For constraint satisfaction problems, however, it is generally not possible to cleanly divide a problem into subproblems that can be solved independently. For instance, if we divide the 4-queen problem such that q_1 and q_3 live in different subproblems, there is a constraint, $R_{\{q_1, q_3\}}$, connecting these two subproblems. So when should such constraint be checked?

One simple solution, as mentioned in Section 1, is to first ignore those constraints that involve variables living in different subproblems, solve each of the subproblems and later use the ignored constraints to filter their combined solution. But this forceful division of problem may cause redundant consistency checks; some instantiations of the solution of one subproblem, Π_1 say, may prune part of the search tree of another subproblem Π_2, and thus make the computation of that part of Π_2's solution redundant.

Figure 2 shows a division of 4-queen problem into two subproblems: The first subproblem contains variables $\{q_1, q_2\}$, and the second contains variables $\{q_2, q_3, q_4\}$. (The astute readers will notice the repetition of variable q_2 in both subproblems. This arrangement will facilitate merging of solutions from two subproblems (*cf.*, Section 4.2).)

When solving the second subproblem independently, two subtrees rooted at level q_3, with assigned values 3 and 4 respectively, are explored. However, as we have already seen in Figure 1, if we were to choose only one solution for the entire problem, we would only need to explore the path passing through node a, and explore the subtree rooted at the node with assignment $q_3 = 4$. In this case, the computation performed at second subproblem on subtree rooted at

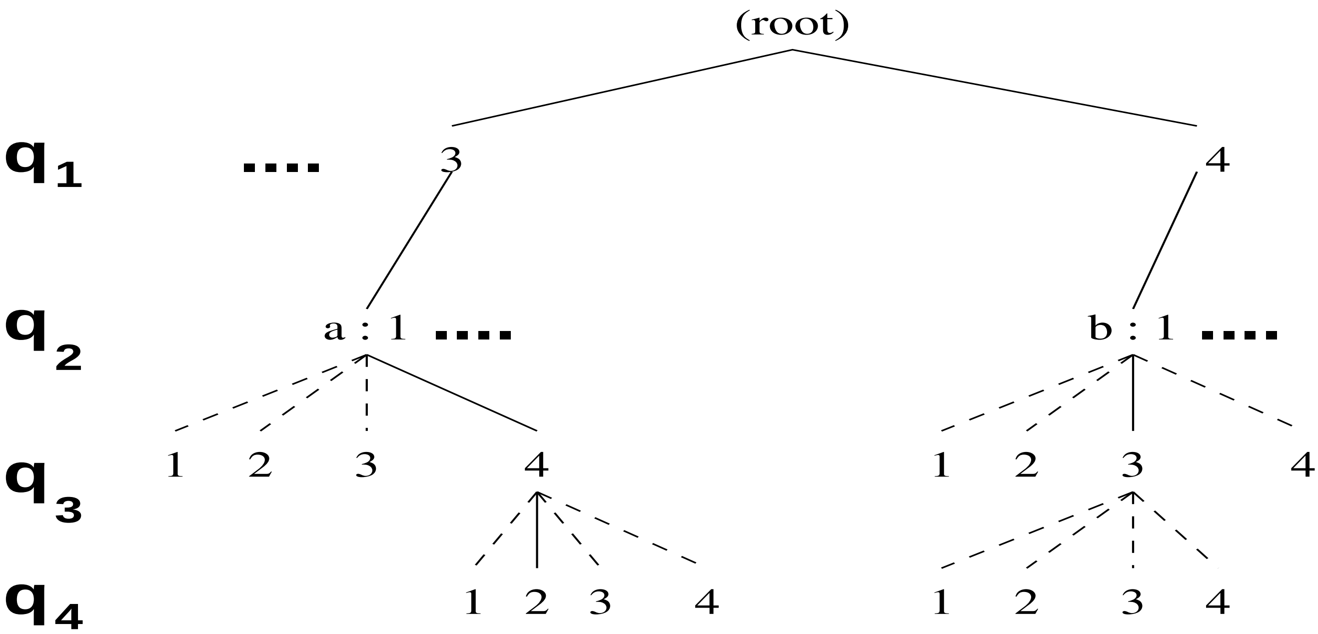

Figure 1. Partial Solution Trees of 4-queen problem

$q_3 = 3$ becomes redundant.

In general, consistency checks done at subproblem can become redundant even when we are finding all solutions to a CSP. This problem can be addressed conveniently by a lazy computation framework like that of Haskell [10]. Subproblems are solved lazily, whenever they are demanded. The demand arises when constraints connecting subproblems are checked to produce final solutions for the entire problem. This demand-driven nature of computation allows some parts of the search trees of subproblems to remain unevaluated, thus eliminating the risk of redundant consistency checks.

4 Algorithm

We identify a constraint satisfaction problem with a constraint network, and a subproblem with a subnetwork. The lazy divide-&-conquer algorithm is defined by function $DivCon_S$, as shown in Figure 3. The function is subscripted by a solver S, and takes in two arguments: a constraint network N; and a map p describing how to divide N into subnetworks. If the network is not to be divided (*ie.*, p is primitive), the solver S is called to find all solutions for N. Otherwise, function $divide$ is called to partition the network into a list of subnetworks; it returns two components: nms is the list of subnetworks and their associated "sub-maps" describing how each subnetwork should be further divided. C is the set of global constraints (taken from N) inter-connecting these subnetworks. Each subnetwork is solved recursively. The results are combined, and the global constraints are solved (with the same solver S) by calling the function $join_S$.

function $DivCon_S$ (N : network, p : map) : solnSet
  local σ : solnSetSet
    nms : network-mapSet
  if $primitiveMap$ (p) **then return** $S(N)$;
  $(C, nms) \leftarrow divide$ (N, p) ;
  for each $(N', p') \in nms$ **lazy-do**
    add $DivCon_S$ (N', p') to σ ;
  return $join_S(C, \sigma)$

Figure 3. Lazy Divide-&-Conquer Algorithm

The algorithm is executed *lazily*. Firstly, every function call invocation is performed lazily; when a function is applied to a tuple of arguments, the call invocation does *not* begin with the evaluation of its arguments. Rather, the function body is executed first. When the value of an argument is required to advance the execution, that argument is evaluated up to the point when the required value is obtained. For example, if an argument will evaluate to a linear list of values, and only the first two elements of the list are needed to complete the function-body execution, then the argument will be evaluated to produce the first two elements; it will not produce the other list elements. In programming terminology, this suspension of an expression evaluation is achieved by building a *thunk* to encapsulate the computa-

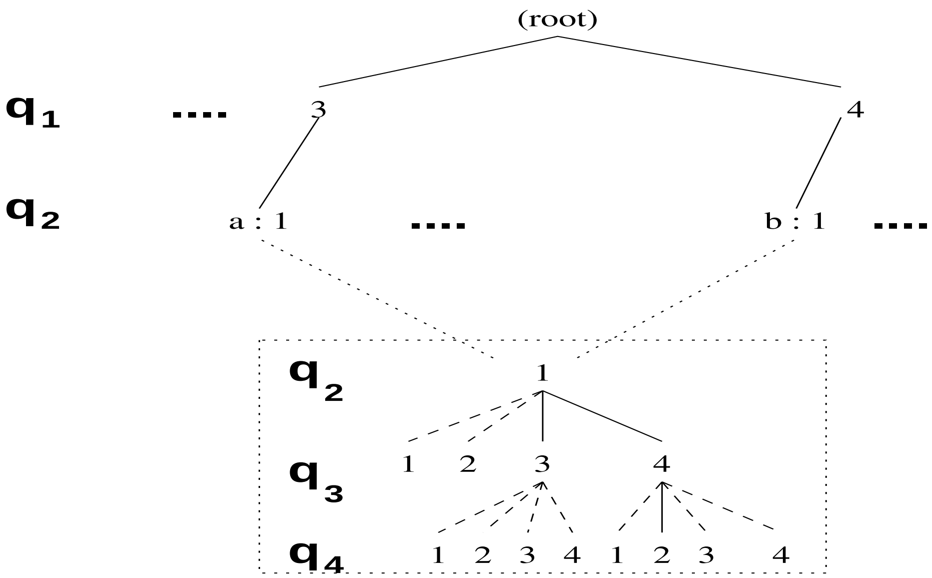

Figure 2. Partial Solution Tree of 4-queens problem divided into two sub-problems with variables q_1, q_2 **and** q_2, q_3, q_4**.**

tion of the expression. When the value of an expression (in this case, the argument) is required, its corresponding thunk will be *forced*.[1]

In addition to lazy function calls, a lazy-loop command, **for each** . . . **lazy-do**, executes its loop-body *lazily* too. Instead of eagerly adding solution set of subproblem (N', p') to σ at each iteration, a thunk corresponding to the execution of this addition (of the solution set) is created. This thunk will be forced when the corresponding solution set is needed (by the $join_S$ operation). Consequently, during runtime, σ does not really contain all solution sets. For those subproblems that have been (partially) solved, σ keeps their (partial) solutions else σ maintains the thunks representing the pending computations.

4.1 Map

A map describes how a network can be subdivided. Our strategy assumes some variable ordering. When dividing a network, we assert that subnetworks preserve this variable ordering; *ie.*, variables occurring in a subnetwork must obey the same variable ordering as the original network.

Network division is done such that two consecutive subnetworks share exactly one common variable. This facilitates the merging of solution sets of subnetworks at later stage. Hence, given a network with variable ordering

$$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$$

The division

$$(x_1, x_2, x_3, x_4) \texttt{ and } (x_4, x_5, x_6, x_7, x_8)$$

is valid, but the division

$$(x_1, x_2, x_3, x_4) \texttt{ and } (x_5, x_6, x_7, x_8)$$

is invalid because the first two subnetworks do not share a common variable.

Without loss of generality, we assume that division of a map always produce two sub-maps, and division operations can be nested.

In defining a constraint-satisfaction problem, we first order the variables, and index them, beginning from 1. Hence, a division of a map can be succinctly described by specifying the index of the common variable. The syntax of a map can be expressed by the following grammar:

$$\text{map} \ ::= \ \text{map} :: int :: \ \text{map} \ | \ \text{None}$$

The earlier example of dividing a network of eight variables can now be expressed as:

$$p \ = \ \text{None} \ :: \ 4 \ :: \ \text{None}$$

[1]The thunk must contain the information required to execute the expression. The process of executing the expression in a thunk is called *forcing*.

A possible nested map is:

$$p' = \text{None} :: 4 :: (\text{None} :: 7 :: \text{None})$$

Given a network N of eight variables $Z = x_1, ..., x_8$, and the above map p'. The call $divide\ (N, p')$ will return

$$(C_Z, \{(N_1, \text{None}), (N_2, (\text{None} :: 7 : \text{None}))\})$$

where N_1 consists of the set of variables $A = \{x_1, x_2, x_3, x_4\}$, and constraints $\mathcal{K}_A$; N_2 consists of the set of variables $B = \{x_4, x_5, x_6, x_7, x_8\}$, and constraint $\mathcal{K}_B \setminus R_{\{x_4\}}$. The decision to put the unary constraint (like $R_{\{x_4\}}$) involving the common variable into left or right subproblem depends on the order in which they are solved, which is controlled by $join_S$ function (described in Section 4.2). The unary constraints should be included into that subproblem(left or right) which is solved first. Lastly, $C_Z = \mathcal{K}_{A \cup B} \setminus (\mathcal{K}_A \cup \mathcal{K}_B)$ is the set of constraints between variables in (A - $\{x_4\}$) and (B - $\{x_4\}$); *ie.*, the inter-connecting constraints between N_1 and N_2.

To proceed further, $divide(N_2, (\text{None} :: 7 :: \text{None}))$ yields

$$(C_B, \{(N_{21}, \text{None}), (N_{22}, \text{None})\})$$

where N_{21} consists of variables $D = \{x_5, x_6, x_7\}$, and constraints $\mathcal{K}_D$; N_{22} has variables $E = \{x_7, x_8\}$, and constraints $\mathcal{K}_E \setminus R_{\{x_7\}}$. Lastly, $C_B = \mathcal{K}_{D \cup E} \setminus (\mathcal{K}_D \cup \mathcal{K}_E)$.

4.2 Function $join_S$

Function $join_S$ combines the solution sets of all subproblems created earlier by the $divide$ function, and subjects them to global constraint (C) satisfaction. Because of the existence of common variable between two subproblems, combining two solution sets is equivalent to "equi-join" operation on two tables in database, where two tables are joined on the condition of equality of their common attribute (the common variable, in our case.)

Under lazy evaluation, when the solution set of a subproblem is needed for the first time, its corresponding thunk is forced, resulting in the thunk being replaced by the (progressively constructed) solution set.

Demand for the solution set of a subnetwork arises from the need to solve global constraints C (which is needed in order to construct a solution for the entire constraint network.) In solving global constraints, we can employ techniques different from that provided by the solver S, although it is simpler to use the same solving technique throughout the entire process.

The result of $join_S$ will be one (possibly empty) solution set that satisfies *all* constraints on the variables of the combined subproblems.

4.3 Solving $S(N)$

In the degenerate case (with the presence of a primitive map $None$), the call $S(N)$ is also executed lazily. Note that $S(N)$ yields solutions that only satisfy constraints local to the subproblem, without considering the interconnecting constraints. The possible output of $S(N)$ are as follows:

1. $S(N)$ may not be executed at all, if there is no need for its solution set. This happens when the algorithm aborts because one subproblem (not the current N) turns out to be unsatisfiable.

2. $S(N)$ may be executed to produce all solutions. This happens when none of its solution is redundant due to global constraints and the algorithm is used to generate all possible solutions to the constraint satisfaction problem.

3. $S(N)$ may be executed to produce some solutions. This happens when some of its solutions are redundant or the algorithm is used to generate just one solution, and the execution of $S(N)$ will halt when the first solution that satisfies the global constraints connecting the subproblems is found.

It is important to understand that, due to lazy evaluation, all functions are executed in an interweaving manner. Thus, it is not the case that $S(N)$ runs to its completion and passes its result to $join_S$. Rather, we can imagine both $S(N)$ and $join_S$ as running in a fine-scale co-routine manner, with $join_S$ pausing to request for some intermediate result from $S(N)$, and continuing when the requested result is available.

5 Experimentation

We have implemented our lazy D&C strategy in Haskell – the *de-facto* lazy functional language. In our implementation, the subproblems corresponding to primitive map are solved using backtracking. Given two subproblems sharing a common variable, the $join_S$ operation solves left subproblem first and then the second subproblem (lazily) and checks the global constraints in backtracking manner on the combined solution.

To illustrate the effectiveness of our approach, we apply our program to solve n-queen problems. We benchmark it against the standard search algorithms developed by Nordin *etc.* in Haskell [9]. A widely used metric for evaluating constraint solving algorithms is the number of consistency checks performed by the algorithms. Adopting this metric, we obtain very encouraging results, as shown in Table 1. Notice that in all cases, our d&c strategy performs the least number of consistency checks.

Queens	9	10	11	12
`bt`	243009	1297558	7416541	45396914
`bjbt`	214510	1099796	6129447	36890689
`bjff1`	47375	191776	868066	4280093
$\texttt{DivCon}_{\texttt{bt}}$	40548	172198	790634	3976570
Solutions	352	724	2680	14200

Legend:

`bt`	backtracking algorithm [9]
`bjbt`	backjumping algorithm [9]
`bjff1`	backjumping with forward checking (The fastest algorithm reported in [9])
$\texttt{DivCon}_{\texttt{bt}}$	d&c framework with backtracking solver

Table 1. Number of consistency checks performed by various algorithms on the all solutions n-queen problem.

Test case	Consistency checks
`test [ ]`	45,396,914
`test [2]`	28,398,804
`test [2,3]`	18,087,264
`test [2,3,4,5]`	7,978,112
`test [2,3,4,5,6,7,8,9,10,11]`	3,976,579

Table 2. Performance Impact of Different Granularity

Table 2 shows the effect of different partitioning on the number of consistency checks. Here, the program `test` takes in a map and runs $\texttt{DivCon}_{\texttt{bt}}$ on 12-queen problem using the map. For simplicity, we express the map in list format: a list `[a,b,c]` is equivalent to the map `None :: a :: (None :: b :: (None :: c :: None))`.From the table, we can see that the number of consistency checks decreases with finer granularity of partitioning.

Our experiment shows that $\texttt{DivCon}_{\texttt{bt}}$ solves the n-queen problem the best when the map is right-skewed, and has finest granularity. We contribute this behavior to the following two reasons: 1. The n-queen problem has a uniform distribution of constraints over all variables; 2. The solver `bt` and the operation $join_{\texttt{bt}}$ find solutions in depth-first manner, traversing the list of variables from left to right. Thus, maximal reuse of solutions can be attained by grouping the subproblem in right-associative fashion. But in general, the topology of constraint network , the solver used to solve the subproblems and the method of combining the solutions of subproblems would decide the goodness of a particular division of the problem.

6 Application

In this section we demonstrate the usefulness of our d-&-c framework for efficient pattern matching in *Chart Pattern Language* (CPL)[1]. CPL is an embedded language for specifying patterns that may occur in price histories of stock market. A pattern consists of a series of points, which are referred to as *landmarks*. Two important primitive patterns are *up* and *down* patterns. An up pattern describes price movement between two landmarks to reflect an upward trend in stock prices. A down pattern describes a downward trend in stock prices between two landmarks. We call the two end-points of a pattern *the landmarks of the pattern*. A general *pattern* is composed by joining a sequence of primitive patterns together and subjecting this composition to some constraints. These constraints are specified in terms of landmarks of the primitive patterns. The landmarks of a general pattern are the collection of landmarks of its primitive patterns, without duplication.

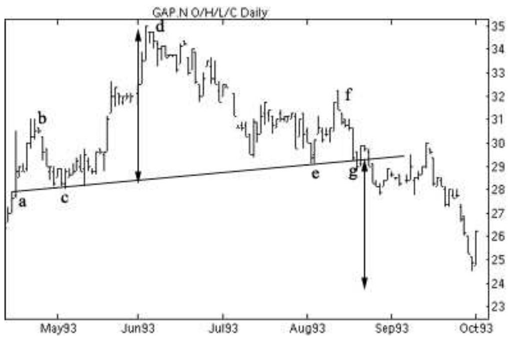

Figure 4. Head and Shoulder

A pattern that is commonly used in technical analysis[14] (of stock performance) is called *head-and-shoulder*. It is a bearish signal that bodes of decline in price. The pattern is shown in Figure 4. It is composed by six primitive patterns. Its corresponding seven landmarks are labeled from `a` to `g` in the figure.

One important task of CPL is to find all occurrences (which are called *instances*) of a pattern, such as head-and-shoulder, in price databases. To do so, CPL finds all possible assignments to the landmarks of a pattern, such that these assignments satisfy the pattern constraints.

We provide below three different definitions of the head-and-shoulder pattern. These definitions differs in the way the pattern is divided into sub-patterns. The definitions use simple Haskell syntax. The symbol >< is a join operation defined informally as follows: if the end-points of a pattern p_1 are a and b (with $a < b$), and the end-points of a pattern p_2 are c and d, then p_1><p_2 is a pattern formed by joining p_1 and p_2 together, such that $b = c$. Furthermore, the symbol ?? is a binary operation that takes a pattern and a set (a list, in fact) of constraints, and returns a new pattern which has the same composition as the pattern argument, but also satisfies the set of constraints. Thus, an instance of $(p\texttt{??}C)$ looks similar to p, and also satisfies the constraints specified in C.

```
global = (up >< down >< up >< down
               >< up >< down) ?? [..]
conc   = let b2d = (down >< up) ?? [..]
             b2e = (b2d >< down) ?? [..]
             b2f = (b2e >< up) ?? [..]
             b2g = (b2f >< down) ?? [..]
         in (up >< b2g) ?? [..]
right  = let e2g = (up >< down) ?? [..]
             d2g = (down >< e2g) ?? [..]
             c2g = (up >< d2g) ?? [..]
             b2g = (down >< c2g) ?? [..]
         in  (up >< b2g) ?? [..]
```

In the definition of `global`, all constraints are globally specified; the pattern thus has the coarsest granularity. In `conc`, the sub-patterns are composed in left-associative manner. In the equation `b2e = (b2d >< down) ?? [..]`, the list of constraints (`[..]`) contains global constraints connecting sub-patterns `b2d` and `down`. Lastly, in `right`, sub-patterns are composed in right-associative manner. Both `conc` and `right` have the finest granularity.

Division	Max. Length of a pattern (days)			
	250	450	650	850
`global`	16.5	24.4	29.8	33.8
`conc`	14.0	17.2	20.6	23.0
`right`	12.3	15.2	17.0	19.0
No. of instances found	21	26	27	27

Table 3. Running time (in seconds) for searching all head-&-shoulder pattern instances

Table 3 shows the time taken to search all instances of head-and-shoulder pattern over 10 years of daily data of 40 companies, under different decompositions and maximum lengths of patterns. In practice, patterns that occur over a very long period of time are not useful for forecasting. So the program only searches for patterns of length less than a particular value given by the user.

Matching a pattern defined by `global` is similar to solving a constraint network using backtracking algorithm. While backtracking algorithm is still used at subproblems, we see that pattern matching can be done efficiently using lazy d&c strategy with finer granularity. The motivation behind decomposing a pattern into `conc` format is that it is efficient to have subgraphs with more local constraints, so that there will be more saving obtained from reusing its solutions. For the current pattern, we find that there are most constraints associated with landmarks `b` to `f`. Thus, we chose to make that a `b2f` a subproblem, and provide further division of this subproblem. As one can see, there can be many way to partition a problem, and the topology of the constraint graph can be exploited during partitioning. However, unlike other works, where decomposition is possible only with a topology that satisfy some properties, our algorithm works for any arbitrary decomposition; though performance may be different.

It is interesting to see that the essence of our lazy d&c strategy has been encoded in the operators >< and ?? of CPL. As such, a pattern defined using these operators reflects actual division of a pattern. Such encoding thus opens up the opportunity for a CPL system to transform and optimize the execution of the lazy d&c strategy, through transformation of pattern definition at compile time.

7 Related Works

The prevalent method used in decomposing constraint network is the tree clustering method [4, 8]. The objective has been to identify constraint networks that can be compiled into an equivalent tree of subnetworks whose respective solutions can be combined into a complete solution efficiently. Some of these tree-decomposition schemes are join-tree clustering [4, 8], junction-tree decompositions, and hyper-tree decomposition [7]. In contrast, what we propose in this paper is a scheme for programmer to freely divide the network, through introduction of maps, in a manner that fits the properties of the network. To support general division of network, we show that lazy evaluation is crucial to the elimination of redundant consistency checks. It will be interesting to determine if the introduction of lazy evaluation can help broaden the range of constraint networks handled by the existing tree-decomposition schemes.

Lazy evaluation is not new to constraint satisfaction algorithms. A flavor of this technique was first exploited by Zweben and Eskey in [13] and later in [5] and [11] to improve forward checking and arc consistency algorithms. Although these algorithms delay some constraint checks, they do not really require a lazy-evaluation framework. In a stroke, these algorithms cleverly infer the amount of "laziness" required to avoid redundant checks[2], so it is more apt to describe these algorithms as "minimal" rather than "lazy".

8 Conclusion and Future Work

In this paper, we address the redundant computation problem associated with application of divide-and-conquer approach to constraint satisfaction problem by employing lazy evaluation. We show that, by performing backtracking algorithm in a lazy d&c framework, we can eliminate redundant computations, leading to a reduction in consistency checks.

But this improvement comes with a price; the space complexity of the algorithm is too high for the algorithm to scale up to large problems. One solution is related to "learning" algorithm[15] for constraint solving, where as the algorithm explores different frontiers of the search space, it learns from it mistakes, that is, when it reaches a node in the solution tree that can not be further extended, it records some information(called "nogoods") which will prevent it to come to the same dead end again. The nogoods recorded during learning represents an approximation of the already explored search space. In our current algorithm, we store a complete copy of the search space explored so far, thus increasing the space requirement. But we can instead store nogoods as an approximation of the search space like learning algorithms.

We have incorporated our algorithm in a domain specific language called Chart Pattern Language for searching patterns in stock market time series data.

9 Acknowledgments

This work was supported by the research grant R-252-000-092-112. We thank the members of programming language and systems lab for their comments on the work.

References

[1] S. Anand, W.N. Chin and S.C. Khoo: Charting Patterns on Price History. ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Sep 2001: 134–145.

[2] F. Bacchus and A. Grove: On the Forward Checking Algorithm. Proceedings of CP 1995, 292-308

[2] Bacchus and Grove remarked that "we rediscovered this (minimal forward checking) not by thinking about lazy evaluation, but instead.." in [2].

[3] R. Dechter and J. Pearl: Network-based heuristics for constraint satisfaction problems. Artificial Intelligence, 34(1), December 1987: 1–38.

[4] R. Dechter and J. Pearl: Tree clustering for constraint networks. Artificial Intelligence, Vol 38, 1989: 353–366.

[5] M.J. Dent and R.E. Mercer: Minimal Forward Checking. 6th IEEE International conference on Tools with Artificial Intelligence, 1994: 432–438

[6] J. Gaschnig: A general backtrack algorithm that eliminates most redundant tests. In IJCAI77, Cambridge, MA, 1977.

[7] G. Gottlob, N. Leone, and F. Scarcello: A Comparison of Structural CSP Decomposition Methods. Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Jul 1999: 394–399.

[8] S.L. Lauritzen and D.J. Spiegelhalter: Local computations with probabilities on graphical structures and their applications to expert systems. Proceedings of the Royal Statistical Society, Series B, No 50, 1988: 154–227.

[9] T. Nordin and A. Tolmach: Modular Lazy Search for Constraint Satisfaction Problems. Journal of Functional Programming, Vol. 11(5), 2001: 557–587

[10] S.L. Peyton Jones and R.J.M. Hughes (eds): Report on the Programming Language Haskell 98: A Non-Strict, Purely Functional Language, Available at `http://www.haskell.og/definition/`

[11] T. Schiex, J-C. Rgin, C. Gaspin and G. Verfaillie: Lazy Arc Consistency. AAAI/IAAI, Vol. 1, 1996: 216–221

[12] R. Seidel: A new method for solving constraint solving problems. Proceedings of IJCAI, 1981, 338–342

[13] M. Zweben and M. Eskey: Constraint satisfaction with delayed evaluation. Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), 1989: 875–880

[14] N. Bulkowski: Encyclopedia of chart patterns. John Wiley and Sons, 2000.

[15] Daniel Frost, Rina Dechter: Dead-End Driven Learning. AAAI 1994: 294-300