A DEEP DIVE INTO BACKTRACKING: SOLVING CSPS WITH EFFICIENCY AND ACCURACY

Nguyen Hoang Tan
Department of Computer Science
University of Information Technology
Linh Trung, Thu Duc
21521313@gm.uit.edu.vn

ABSTRACT

Over the past twenty five years many backtracking algorithms have been developed for constraint satisfaction problems. This article describes the basic backtrack search within the search space framework and then presents a number of improvements developed in the past two decades, branching strategies, constraint propagation, nogood recording, backjumping, heuristics for variable and value ordering, randomization and restart strategies, and alternatives to depth-first search.

1 Introduction

There are three main algorithmic techniques for solving constraint satisfaction problems: backtracking search, local search, and dynamic programming. In this article, I survey backtracking search algorithms.

An algorithm for solving a constraint satisfaction problem (CSP) can be either complete or incomplete. Complete, or systematic algorithms, come with a guarantee that a solution will be found if one exists, and can be used to show that a CSP does not have a solution and to find a provably optimal solution. Backtracking search algorithms and dynamic programming algorithms are, in general, examples of complete algorithms. Incomplete, or non-systematic algorithms, cannot be used to show a CSP does not have a solution or to find a provably optimal solution. However, such algorithms are often effective at finding a solution if one exists and can be used to find an approximation to an optimal solution. Local or stochastic search algorithms are examples of incomplete algorithms.

Of the two classes of algorithms that are complete—backtracking search and dynamic programming—backtracking search algorithms are currently the most important in practice. The drawbacks of dynamic programming approaches are that they often require an exponential amount of time and space, and they do unnecessary work by finding, or making it possible to easily generate, all solutions to a CSP. However, one rarely wishes to find all solutions to a CSP in practice. In contrast, backtracking search algorithms work on only one solution at a time and thus need only a polynomial amount of space.

Since the first formal statements of backtracking algorithms over 40 years ago, many techniques for improving the efficiency of a backtracking search algorithm have been suggested and evaluated. In this article, I survey some of the most important techniques including branching strategies, constraint propagation, nogood recording, backjumping, heuristics for variable and value ordering, randomization and restart strategies, and alter- natives to depth-first search. The techniques are not always orthogonal and sometimes combining two or more techniques into one algorithm has a multiplicative effect (such as combining restarts with nogood recording) and sometimes it has a degradation effect (such as increased constraint propagation versus backjumping). Given the many possible ways that these techniques can be combined together into one algorithm, I also survey work on comparing backtracking algorithms. The best combinations of these techniques result in robust backtracking algorithms that can now routinely solve large, hard instances that are of practical importance.

2 Preliminaries

In this section, I first define the constraint satisfaction problem followed by a brief review of the needed background on backtracking search.

2.1 The constraint framework

Definition 2.1 (CSP)**.** *A constraint satisfaction problem (CSP) consists of a set of variables, $X = \{x_1, ..., x_n\}$; a set of values, $D = \{a_1, ..., a_d\}$, where each variable $x_i \in X$ has an associated finite domain $dom(x_i) \subseteq D$ of possible values; and a collection of constraints.*

As a running example in this survey, I will use the 6-queens problem: how can we place 6 queens on a 6×6 chess board so that no two queens attack each other. As one possible CSP model, let there be a variable for each column of the board $\{x_1, ..., x_6\}$, each with domain $dom(x_i) = \{1, ..., 6\}$. Assigning a value j to a variable x_i means placing a queen in row j, column i. Between each pair of variables x_i and x_j , $1 \leq i < j \leq 6$, there is a constraint $C(x_i, x_j)$, given by $(x_i \neq x_j) \wedge (|i-j| \neq |x_i - x_j|)$. One possible solution is given by $\{x_1 = 4, x_2 = 1, x_3 = 5, x_4 = 2, x_5 = 6, x_6 = 3\}$.

2.2 Search - backtracking

The term *search* is used to represent a large category of algorithms which solve problems by guessing an operation to perform or an action to take, possibly with the aid of a heuristic. A good guess results in a new state that is nearer to a goal. If the operation does not result in progress towards the goal (which may not be apparent until later in the search), then the operation can be retracted and another guess made.

For CSPs, search is exemplified by the backtracking algorithm. Backtracking search uses the operation of assigning a value to a variable, so that the current partial solution is extended. When no acceptable value can be found, the previous assignment is retracted, which is called a backtrack. In the worst case the backtracking algorithm requires exponential time in the number of variables, but only linear space. The algorithm was rst described more than a century ago, and since then has been reintroduced several times.

3 Backtracking

A simple algorithm for solving constraint satisfaction problems is backtracking, which traverses the search graph in a depth- first manner. The order of the variables can be xed in advance or determined at run time. The backtracking algorithm maintains a partial solution that denotes a state in the algorithm's search space. Backtracking has three phases: a forward phase in which the next variable in the ordering is selected; a phase in which the current partial solution is extended by assigning a consistent value, if one exists, to the next variable; and a backward phase in which, when no consistent value exists for the current variable, focus returns to the variable prior to the current variable.

Figure 1 describes a basic backtracking algorithm. As presented in Figure 1, the backtracking algorithm returns at most a single solution, but it can easily be modified to return all solutions, or a desired number. The algorithm employs a series of mutable value domains D'_i such that each $D'_i \in D_i.D'_i$ holds the subset of D_i that has not yet been examined under the current partial instantiation. The D' sets are not needed if the values can be mapped to a contiguous set of integers that are always considered in ascending order; in this case a single integer can be used as a marker to divide values that have been considered from those that have not. We use the D' sets to describe backtracking for increased generality and to be consistent with the portrayal of more complex algorithms later in the paper.

The $selectValue$ subprocedure is separated from the main backtracking routine for clarity. It has access to the current value of all variables in the main procedure. The complexity of $selectValue$ depends on how the constraints are represented internally. When the CSP is binary, it may be practical to store the constraints in a table in contiguous computer memory; a one bit or one byte flag denotes whether each possible pair of variables with corresponding values is compatible. Accessing an entry in the table is an $O(1)$ operation. Since the compatibility of the current candidate assignment $(x_i; a)$ must be checked with at most n earlier variable-value pairs, the complexity of selectValue for binary CSPs can be $O(n)$. With non-binary CSPs, checking compatibility is more expensive for two reasons. First, the number of possible constraints is exponential in the number of variables, and so the actual constraints are most likely stored in a list structure. Checking the list will be $O(\log c)$, where c is the number of constraints. Another possibility that bears consideration is that a constraint can be represented not as data but as a procedure. In this case the complexity of $selectValue$ is of course dependent on the complexity of the procedures it invokes.

```
procedure BACKTRACKING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    i ← 1                                   (initialize variable counter)
    D'_i ← D_i                              (copy domain)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE
        if x_i is null                      (no value was returned)
            i ← i − 1                       (backtrack)
        else
            i ← i + 1                       (step forward)
            D'_i ← D_i
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        if (x_i, a) is consistent with ā_{i−1}
            return a
    end while
    return null                             (no consistent value)
end procedure
```

Figure 1: The Backtracking algorithm

4 Improvements to backtracking

Much of the work in constraint satisfaction during the last decade has been devoted to improving the performance of backtracking search. Backtracking usually suffers from thrashing, namely, rediscovering the same inconsistencies and same partial successes during search. Efficient cures for such behavior in all cases are unlikely, since the problem is NP-hard.

The performance of backtracking can be improved by reducing the size of its expanded search space, which is determined both by the size of the underlying search space, and by the algorithm's control strategy. The size of the underlying search space depends on the way the constraints are represented (e.g. on the level of local consistency), the order of variable instantiation, and, when one solution succes, the order in which values are assigned to each variable. Using these factors, researchers have developed procedures of two types: those employed before performing the search, thus bounding the size of the underlying search space; and those used dynamically during the search and that decide which parts of the search space will not be visited. Commonly used preprocessing techniques are arc- and path-consistency algorithms, and heuristic approaches for determining the variable ordering.

The procedures for dynamically improving the pruning power of backtracking can be conveniently classified as *look-ahead schemes* and *look-back schemes*, in accordance with backtracking's two main phases of going forward to assemble a solution and going back in case of a dead-end. *Look-ahead schemes* can be invoked whenever the algorithm is preparing to assign a value to the next variable. The essence of these schemes is to discover from a restricted amount of constraint propagation how the current decisions about variable and value selection will restrict future search. Once a certain amount of forward constraint propagation is complete the algorithm can use the results to:

1. Decide which variable to instantiate next, if the order is not predetermined. Generally, it is advantageous to first instantiate variables that maximally constrain the rest of the search space. Therefore, the most highly constrained variable having the least number of values, is usually selected.

2. Decide which value to assign to the next variable when there is more than one candidate. Generally, when searching for a single solution an attempt is made to assign a value that maximizes the number of options available for future assignments.

Look-back schemes are invoked when the algorithm is preparing the backtracking step after encountering a dead-end. These schemes perform two functions:

1. Deciding how far to backtrack. By analyzing the reasons for the dead-end, irrelevant backtrack points can often be avoided so that the algorithm goes back directly to the source of failure, instead of just to the immediately preceding variable in the ordering. This procedure is often referred to as backjumping.

2. Recording the reasons for the dead-end in the form of new constraints, so that the same con icts will not arise again later in the search. The terms used to describe this function are constraint recording and learning.

In sections 5 and 6 we will describe in detail several principle look-back schemes, while section 7 will focus on look-ahead methods.

5 Backjumping

Backjumping schemes are one of the primary tools for reducing backtracking's unfortunate tendency to rediscover the same dead-ends. A dead-end occurs if x_i has no consistent values left, in which case the backtracking algorithm will go back to x_{i-1}. Suppose a new value for x_{i-1} exists but there is no constraint between x_i and x_{i-1}. A dead-end will be reached at xi for each value of x_{i-1} until all values of $xi - 1$ have been exhausted. We can finesse this situation by identifying the *culprit variable* responsible for the dead-end and then jumping back immediately to reinstantiate the *culprit variable*, instead of repeatedly instantiating the chronologically previous variable. Identification of a *culprit variable* in backtracking is based on the notion of *conflict sets*.

Definition 5.1 (conflict set)**.** *Let $\vec{a} = (a_1; ...; a_i)$ be a consistent instantiation, and let x be a variable not yet instantiated. If no value in the domain of x is consistent with $\vec{a}$, we say that $\vec{a}$ is a conflict set of x, or that $\vec{a}$ conflicts with variable x. If, in addition, $\vec{a}$ does not contain a subtuple that is in conflict with x, $\vec{a}$ is called a minimal conf;ict set of x.*

Definition 5.2 (i-leaf dead-ends)**.** *Given an ordering $d = x_1; ...; x_n$, then a tuple $\vec{a}_i = (a_1; ...a_i)$ that is consistent but is in conflict with x_{i+1} is called an i-leaf dead-end state.*

Definition 5.3 (no-good)**.** *Any partial instantiation $veca$ that does not appear in any solution is called a no-good. Minimal no-goods have no no-good subtuples.*

A conflict set is clearly a no-good, but there are no-goods that are not conflict sets of any single variable. Namely, they may conflict with two or more variables.

Whenever backjumping discovers a dead-end, it tries to jump as far back as possible without skipping potential solutions. These two issues of safety in jumping and optimality in the magnitude of a jump need to be de ned relative to the information status of a given algorithm. What is safe and optimal for one style of backjumping may not be safe and optimal for another, especially if they are engaged in different levels of information gathering.

6 Learning Algorithms

The earliest minimal conflict set of Definition 6.1 is a no-good explicated by search and is used to focus backjumping. However, this same no-good may be rediscovered again and again while the algorithm explores different paths in the search space. By making this no-good explicit, in the form of a new constraint, we can make sure that the algorithm will not rediscover it and, moreover, that it may be used for pruning the search space. This technique, called *constraint recording*, is behind the learning algorithms described in this section.

Definition 6.1 (earliest minimal con ict set)**.** *Let $\vec{a}_i$ be a dead-end tuple whose dead-end variable is x_{i+1}. We denote by $emc(\vec{a}_i)$ the earliest minimal conflict set of $\vec{a}_i$ and by $par(\vec{a}_i)$ the set of variables appearing in $emc(\vec{a}_i)$. Formally, the emc is generated by selecting its members from $\vec{a}_i$ in increasing order. Assume that $(a_{i_1}; ...; a_{i_j})$ were the first j members selected. Then, the first value appearing after a_i in $\vec{a}_i$ that is inconsistent with a value of x_{i+1} that was not ruled out by $(a_{i_1}; ...; a_{i_j})$, will be included in emc.*

By learning we mean recording potentially useful information as that information becomes known to the problem-solver during the solution process. The information recorded is deduced from the input and involves neither generalization nor errors. An opportunity to learn (or infer) new constraints is presented whenever the backtracking algorithm encounters a dead-end, namely, when the current instantiation ai = (a1 ; :::; ai) is a con ict set of xi+1. Had the problem included an explicit constraint prohibiting this con ict set, the dead-end would never have been reached. The learning procedure records a new constraint that makes explicit an incompatibility that already existed implicitly in a given set of variable assignments. There is no point, however, in recording at this stage the conflict set $\vec{a}_i$ itself as a constraint, because under the backtracking control strategy the current state will not recur. Yet, when $\vec{a}_i$ contains one or more subsets that are in conflict with x_{i+1} , recording these smaller conflict sets as constraints may prove useful in the continued search; future states may contain these conflict sets, and they exclude larger conflict sets as well.

With the goal of speeding up search, the target of learning is to identify con ict sets that are as small as possible, namely, minimal. As noted above, one obvious candidate is the earliest minimal conflict set, which is identified anyway for conflict-directed backjumping. Alternatively, if only graph information is used, the graph-based conflict set could be identified and recorded. Another (extreme) option is to learn and record *all* the minimal conflict sets associated with the current dead-end.

In learning algorithms, the savings from possibly reducing the amount of search by finding out earlier that a given path cannot lead to a solution must be balanced against the costs of processing at each node generation a more extensive database of constraints.

Learning algorithms may be characterized by the way they identify smaller conflict sets. Learning can be deep or shallow. Deep learning records only the minimal conflict sets. Shallow learning allows recording of nonminimal conflict sets as well. Learning algorithms may also be characterized by how they bound the arity of the constraints recorded. Constraints involving many variables are less frequently applicable, require more space to store, and are more expensive to consult than constraints having fewer variables. The algorithm may record a single no-good or multiple no-goods per dead-end, and it may allow learning at leaf dead-ends only or at internal dead-ends as well.

7 Look-ahead Strategies

7.1 Combining backtracking and constraint propagation

CSP search algorithms can combine backtracking and local constraint propagation, by applying a consistency enforcing procedure to the uninstantiated variables. This combination is known as "looking ahead." Extending a partial instantiation may induce constraints on the remaining variables, and making these constraints explicit may reduce the amount of backtracking search required. Of course, actions conditioned on a partial instantiation will have to be undone if the partial instantiation becomes no longer current due to backtracking.

While look-ahead strategies incur an extra cost after each instantiation, they can provide several benefits. First, by removing from each future variable's domain all values that are not consistent with the partial instantiation, they eliminate the need to test values of the current variable for consistency with previous variables. A corollary benefit is that if *all* values of an uninstantiated variable are removed by the look-ahead procedure, then the current instantiation cannot be part of a solution and the algorithm can backtrack. Consequently, dead-ends occur earlier in the search, and users of CSP algorithms often note much smaller search spaces when look-ahead is employed. In general, the stronger the level of constraint propagation, the smaller the search space explored and the higher the computational overhead. Another benefit of look-ahead is that the sizes of the uninstantiated variable domains can be used to guide the selection of the variable and value to choose next.

7.2 Look-ahead algorithms

Look-ahead strategies that employ local consistency procedures have the same exponential worst-case time bounds as backtracking. If the look-ahead procedure is based on arc-consistency or a weaker form of consistency, then the space requirements are no more than maintaining the D' sets. The key issue is determining experimentally a cost-effective balance between look-ahead's accuracy and its overhead. We define four levels of look-ahead. Each employs the revise subprocedure for enforcing the arc-consistency of the constraint between two variables.

- *Forward checking*. This approach, which is described in Figure 2, does the most limited form of constraint propagation. Forward checking propagates separately the effect of a tentative value selection to each of the future variables. If, as a result of calling revise the domain of one of these future variables becomes empty, the value is not selected and the next candidate value is tried.

```
procedure FORWARD-CHECKING
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n          (copy all domains)
    i ← 1                              (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-FORWARD-CHECKING
        if x_i is null                 (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                  (backtrack)
        else
            i ← i + 1                  (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE-FORWARD-CHECKING
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        empty-domain ← false
        for all k, i < k ≤ n
            REVISE(k, i)
            if D'_k is empty           (x_i = a leads to a dead-end)
                empty-domain ← true
        if empty-domain                (don't select a)
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
    end while
    return null                        (no consistent value)
end procedure
```

Figure 2: The forward checking algorithm

- *Arc-consistency look ahead*. This category includes backtracking based algorithms which enforce arc-consistency on the uninstantiated variables after each assignment of a value to the current variable. If the uninstantiated variables are not arc-consistent, namely, during the process a variable's domain becomes empty, then the value is rejected. Several arcconsistency enforcing algorithms have been developed; we do not specify which version should be used in Figure 3.

```
procedure ARC-CONSISTENCY-LOOKING-AHEAD
Input: A constraint network with variables {x_1, ..., x_n} and domains
{D_1, ..., D_n}.
Output: Either a solution, or notification that the network is inconsistent.
    D'_i ← D_i for 1 ≤ i ≤ n              (copy all domains)
    i ← 1                                  (initialize variable counter)
    while 1 ≤ i ≤ n
        instantiate x_i ← SELECTVALUE-ARC-CONSISTENCY
        if x_i is null                     (no value was returned)
            reset each D' set to its value before x_i was last instantiated
            i ← i − 1                      (backtrack)
        else
            i ← i + 1                      (step forward)
    end while
    if i = 0
        return "inconsistent"
    else
        return instantiated values of {x_1, ..., x_n}
end procedure

procedure SELECTVALUE-ARC-CONSISTENCY
    while D'_i is not empty
        select an arbitrary element a ∈ D'_i, and remove a from D'_i
        apply ARC-CONSISTENCY to all uninstantiated variables
        if any future domain is empty(don't select a)
            reset each D'_k, i < k ≤ n to value before a was selected
        else
            return a
    end while
    return null                            (no consistent value)
end procedure
```

Figure 3: The foward checking algorithm with arc-consistency enforcing after each instantiation

Our list of look-ahead techniques is not meant to be exhaustive. In particular, a search algorithm could enforce a higher degree of consistency than arc-consistency after each instantiation. Although applying arc-consistency was highly successful on a class of vision instances, the more extensive varieties of look-ahead have received less attention. This may be due, in part, to the negative conclusions about full looking ahead reached in: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required." More recent work has shown that as larger and more difficult problems are experimented with, higher levels of look-ahead become more useful.

8 References

1. S. Arnborg. Effcient algorithms for combinatorial problems on graphs with bounded decomposability
2. A. B. Baker. The hazards of fancy backtracking. In Proceedings of National Conference of Arti cial Intelligence.
3. A. B. Baker. Intelligent Backtracking on constraint satisfaction problems: experimental and theoretical results. PhD thesis, Graduate
4. Rina Dechter. Backtracking Algorithms for Constraint satisfaction problems - a tutorial survey.
5. F. Rossi, P. van Beek. Handbook of Constraint Programing - Chapter 4: Backtracking Search Algorithms.