DEPENDENCY DIRECTED BACKTRACKING IN GENERALIZED SATISFICING ASSIGNMENT PROBLEMS

by

Albert Croker
Leonard N. Stern School of Business
Department of Information Systems
New York University
90 Trinity Place
New York, New York 10006

Vasant Dhar
Department of Information Systems
New York University
90 Trinity Place
New York, New York 10006

and

David McAllester
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Massachusetts 02139

August 1990

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-90-10

ABSTRACT

Many authors have described search techniques for the *satisficing assignment problem*: the problem of finding an interpretation for a set of discrete variables that satisfies a given set of constraints. In this paper we present a formal specification of *dependency directed backtracking* as applied to this problem. We also generalize the satisficing assignment problem to include limited resource constraints that arise in operations research and industrial engineering. We discuss several new search heuristics that can be applied to this generalized problem, and give some empirical results on the performance of these heuristics.

1. Introduction

In this paper, we provide a description of a general purpose dependency directed backtracking algorithm that is applicable to a type of problem referred to as the *satisficing assignment problem* (SAP), (Knuth, 1975; Gashnig, 1979). This problem has also been referred to as the *consistent labeling problem* (Haralick and Shapiro, 1980) and the *constraint satisfaction problem* (Fikes, 1970). Many problems in Artificial Intelligence and Operations Research can be viewed as instances or special cases of such a problem. Broadly, a satisficing assignment problem involves determining an interpretation for a discrete set of variables, that is, assigning a value to each variable, such that the interpretation satisfies a given set of constraints.

Several search algorithms have been proposed for solving the satisficing assignment problem. These include tree search techniques such as *backtracking* (Knuth, 1975) and its variants (Gashnig 1974, 1979), and filtering techniques such as Waltz's (1975) *arc-consistency* and Montanari's (1979) *path-consistency* algorithms. As Nudel (1983) notes, filtering can reduce the number of assignments of values to variables that must be explored, but is not guaranteed to find a solution. Tree search alone guarantees to find all solutions but suffers from *thrashing* (Bobrow and Raphael, 1974; Mackworth, 1977). An analysis of several tree search, filtering, and hybrid techniques can be found in Haralick and Elliot (1980) and Nudel (1983).

Over the last few years, *dependency directed backtracking*, first proposed by Stallman and Sussman (1977), has been receiving increasing attention as a method for reducing the thrashing behavior associated with tree search programs. This method associates *justification* information with each assignment, and uses this information for adjusting beliefs in assignments when constraints are violated (de Kleer et. al, 1977; Doyle, 1979; McAllester, 1982; Goodwin, 1985; de Kleer, 1986). While there are differences in the motivations and reasoning methods underlying the various dependency directed reasoning formalisms, a common feature is their ability to eliminate parts of the search space by "localizing" the assignments responsible for the violation, and avoiding future maneuvers that lead to inclusion of these *nogood* sets of assignments. In this paper, we show that dependency directed backtracking can be used to reduce search in satisficing assignment problems. Specifically, we show how justifications associated with the assignment of values to variables can be used to identify specific assignments that are responsible for the violation of constraints. By avoiding assignments that include these nogood assignments, large parts of the search space can be excluded from consideration.

One of our objectives is to provide a formal specification of dependency directed backtracking as applied to the satisficing assignment problem. We also generalize this problem by including a new type of constraint, one that arises frequently in operations research and industrial engineering. Specifically, each assignment of a value to a variable has associated with it a set of values, each of which specifies the amount of a specific resource consumed by that assignment. An acceptable solution is one where the total of each resource consumed by the assignments does not exceed the available amount of that resource. Resource constraints provide a way of guiding the search process by avoiding assignments that lead to consumption of large amounts of resources that are in short supply (Dhar and Quayle, 1985).

Preliminary empirical results that we have obtained indicate some of the savings in search that are achievable using knowledge about resource constraints.

The remainder of this paper is organized as follows: a description and formal definition of the SAP is presented in Section 2. Section 3 provides a description of three functionally equivalent backtracking algorithms, two from the literature, and one that we propose. Included with the presentation of our backtracking algorithm is a discussion of possible heuristics that can be encoded into it. We conclude with a brief discussion of certain extensions to our backtracking algorithm that we are currently pursuing.

2. Problem Definition

The research presented in this paper has been motivated by an attempt at partially automating the generation of business models to support decision-making in business organizations. Approaches generally adopted for this purpose by researchers in Operations Research and Industrial Engineering have involved the design of optimization models, typically linear programming models, formulated in terms of continuous variables relevant to the problem domain. If the problem requires the introduction of discrete boolean variables, an integer programming formulation becomes necessary. However, if a significant number of discrete boolean variables are involved, the formulation can become cumbersome to specify and difficult to solve. Further, if no objective function can be specified, as is the case with satisficing problems (Simon, 1947), even an integer programming formulation is impossible. For such problems, optimization techniques become difficult to apply.

A *satisficing assignment problem*, SAP, is characterized by a decomposition of the overall problem into discrete sets of competing alternatives and making choices from each of these sets in the presence of a set of constraints. A *satisficing solution* is one where a selection has been made from each set of competing alternatives such that no constraint is violated. Formally, a SAP can be defined as follows:
Let $X = \{x_1, x_2,...,x_n\}$ be a set of variables. Associated with each x_i is a set

$$D_i = \{v_{i,1}, v_{i,2},...,v_{i,n_i}\}$$

of n_i alternatives (values), one of which must be assigned to the variable x_i.

A *constraint* over the set of variables X is a restriction on the set of values that can be simultaneously assigned to a specified subset of these variables. More specifically, we define a *discrete constraint* $\mathbf{P}(x_{i_1}, x_{i_2}, ..., x_{i_m})$ to be a subset of the tuples in the Cartesian product

$$D_{i_1} \times D_{i_2} \times ... \times D_{i_m}$$

and is the set of all permissible simultaneous assignments to the variables $x_{i_1}, x_{i_2}, ..., x_{i_m}$. An *assignment* ρ is a function over the variables $x_1, x_2, ..., x_n$ satisfying $\rho(x_i) \in D_i$. A *partial assignment* over a set of variables is an assignment over some subset of those variables. A variable over which an assignment or partial assignment is defined is called an *assigned variable*. An assignment ρ is said to

satisfy the discrete constraint **P** over the variables $x_{i_1}, x_{i_2}, ..., x_{i_m}$ if

$$<\rho(x_{i_1}), \rho(x_{i_2}), ..., \rho(x_{i_m})> \in \mathbf{P}(x_{i_1}, x_{i_2}, ..., x_{i_m})$$

For example, consider a flexible manufacturing scenario involving four machines, denoted ***C***, ***M***, ***B***, and ***F***, one for each of the operations of cutting, milling, buffing, and finishing, respectively. Each machine can be configured in one of three ways with each configuration being characterized by a processing time and a processing cost. We also associate an identifier with each configuration. Figure 2-1 shows the configurations associated with each of the four machines.

cutting (C)

config.	time	cost
c1	20	5
c2	5	20
c3	10	10

(A)

milling (M)

config.	time	cost
m1	10	10
m2	12	8
m3	15	6

(B)

buffing (B)

config.	time	cost
b1	8	10
b2	10	8
b3	5	12

(C)

finishing (F)

config.	time	cost
f1	6	8
f2	8	6
f3	10	4

(D)

Figure 2-1: Manufacturing Machine Configurations

We assume that only certain combinations of configurations are possible for manufacturing a given product. Specifically, consider the following two constraints:

1. *The time required in finishing the product must be greater than one-third the sum of the cutting and milling times.*
2. *The time required for buffing must not exceed the cutting time.*

Each of these constraints limits the permissible combination of machine configurations. For example, the choice of configurations *c1* and *m1* for the cutting and milling machines, respectively, will require configuration *f3* for the finishing machine. With the second constraint, any configuration of the cutting machine can be used with configuration *b3* of the buffing machine, whereas configuration *c2* is disallowed with configurations *b1* and *b2*.

In addition to the discrete constraints defined above, we define a second type of constraint called a *resource constraint*. Each variable x_i has associated with it a set of *cost tables*, each of which is a mapping from D_i to real numbers. A *resource constraint* is expressed as an inequality of the form

$$T_{j,1}(x_1) + T_{j,2}(x_2) + ...+ T_{j,n}(x_n) \leq \mathbf{c_j}$$

where for a given resource, R_j, $T_{j,i}$ is the cost table associated with the variable x_i. An assignment ρ satisfies the above resource constraint if the sum of the costs assigned to the values of the variables is less than the given bound $\mathbf{c_j}$ of that resource.

As an example of a resource constraint, suppose that in the manufacturing scenario described above the total time allowed for the processing of a product on the four machines must not exceed 27 time units (say minutes). We express this constraint as:

$$time(\mathbf{C})+time(\mathbf{M})+time(\mathbf{B})+time(\mathbf{F}) \leq 27$$

The generalized SAP defined in this paper consists of a set of discrete variables and a set of associated alternatives for each variable, a set of discrete constraints, and a set of resource constraints.

Since it is possible to determine the set of tuples that satisfy a given resource constraint, each resource constraint could be transformed into a discrete constraint. However, a resource constraint typically involves a large number of variables, and explicitly storing the set of tuples can be prohibitively expensive. Furthermore, a resource constraint contains useful information that would be lost in transforming it into a discrete constraint. Specifically, the resource constraint gives the cost of individual variable assignments, information that is not explicitly represented in the corresponding discrete constraint. This information can be useful in backtracking when a constraint violation occurs when some resource bounds are exceeded.

3. Search and Backtracking

In this section, we present three backtracking algorithms that utilize varying degrees of knowledge in determining a backtracking point. The simplest of these, *chronological* backtracking, is completely *uninformed*. This is followed by a description of the *dependency directed* backtracking formalism that makes use of dependency information for backtracking. We conclude the section with our description of a *heuristic dependency directed* backtracking algorithm, which in addition to making use of discrete constraints, utilizes resource constraints to determine the most appropriate backtracking maneuver.

3.1. Chronological Backtracking

Following Knuth (1975), we view the chronological backtracking approach as one where all attempts are made to extend a given partial assignment into a satisficing assignment before an alternative partial assignment is explored. This approach, which we present present below as Procedure SA-SEARCH, extends a partial assignment (including the empty assignment) into a satisficing assignment if such an extension exists. When invoked with a partial assignment ρ of the problem variables Procedure SA-SEARCH will either return a satisficing assignment that is an extension of ρ, or the value *fail*. The value *fail*, which we use to denote the *empty* assignment to the problem variables, indicates that no such extension exists, that is, every attempt to extend the partial assignment represented by ρ leads to the

violation of some problem constraint.

Initially SA-SEARCH (Step SA-1) checks to see if its argument, the partial assignment ρ, violates any constraints. If a violation is detected, then SA-SEARCH returns the value *fail* (i.e., an empty assignment). If no constraints are violated, SA-SEARCH next selects an unassigned variable x_i (Step SA-3) to which it assigns a value that is used in an attempt to recursively extend the partial assignment ρ (Steps SA-4 and SA-5). If this attempt fails, successively assigns values to x_i, each of which is used in an attempt to recursively extend the partial assignment ρ (Step SA-6). This successive assignment of values terminates when a satisficing assignment is found, or when there remains no additional values that can be assigned to x_i. If found, the satisficing assignment is returned by SA-SEARCH, otherwise the value *fail* is returned.

Procedure SA-SEARCH (ρ)

```
SA-1. if some constraint P is violated by ρ then return (fail).
SA-2. if all variables x_i have been assigned a value
          then return (ρ).
SA-3. Select x_i, a variable not assigned a value by ρ.
SA-4. let j=1.
SA-5. let ρ' = SA-SEARCH (ρ[x_i = v_i,j]).[1]
SA-6. while fail(ρ') & j < n_i do
          begin
          j=j+1
          ρ' = SA-SEARCH (ρ[x_i = v_i,j])
          end
SA-7. return (ρ')
```

Figure 3-1 illustrates the behavior of SA-SEARCH on the problem of determining a satisficing set of configurations for the four machines introduced as part of the manufacturing scenario in the last section. In this figure we use the symbols "*" to indicate the occurrence of a constraint violation, and "⊥" to indicate a satisficing solution.

As Figure 3-1 indicates, chronological backtracking results in a lot of wasted work since the reasons for failure are not recorded. If a bad choice is made early in the search, unfruitful parts of the search space are explored. In the following subsection we illustrate how dependency directed backtracking uses information about failures to focus search to more relevant parts of the search space.

3.2. Dependency Directed Backtracking

In the context of a set of discrete constraints a partial assignment can be extended in one of two ways. It can be extended by making an "assumption", that is, selecting some unassigned variable and assigning it a value from the set of available choices. It can also be extended by making a "deduction", that is, assigning values to unassigned variables by propagating forward the consequences of the currently made set of assumptions. The assignment of a value to a variable through a deduction, thus, is always dependent on the assumption-based assignment of values to other variables.

---

[1] Given a function f, $f[x=y]$ is the function f' that agrees with f on all variables other than x to which it assigns the value y.

```
1.  C = c1
2.  C = c1,  M = m1
3.  C = c1,  M = m1,  B = b1
4.  C = c1,  M = m1,  B = b1,  F = f1 *
5.  C = c1,  M = m1,  B = b1,  F = f2 *
6.  C = c1,  M = m1,  B = b1,  F = f3 *
7.  C = c1,  M = m1,  B = b2
8.  C = c1,  M = m1,  B = b2,  F = f1 *
9.  C = c1,  M = m1,  B = b2,  F = f2 *
10. C = c1,  M = m1,  B = b2,  F = f3 *
11. C = c1,  M = m1,  B = b3
12. C = c1,  M = m1,  B = b3,  F = f1 *
13. C = c1,  M = m1,  B = b3,  F = f2 *
14. C = c1,  M = m1,  B = b3,  F = f3 *
15. C = c1,  M = m2
16. C = c1,  M = m2,  B = b1
17. C = c1,  M = m2,  B = b1,  F = f1 *
18. C = c1,  M = m2,  B = b1,  F = f2 *
20. C = c1,  M = m2,  B = b1,  F = f3 *
21. C = c1,  M = m2,  B = b2
22. C = c1,  M = m2,  B = b2,  F = f1 *
23. C = c1,  M = m2,  B = b2,  F = f2 *
24. C = c1,  M = m2,  B = b2,  F = f3 *
25. C = c1,  M = m2,  B = b3
26. C = c1,  M = m2,  B = b3,  F = f1 *
27. C = c1,  M = m2,  B = b3,  F = f2 *
28. C = c1,  M = m2,  B = b3,  F = f3 *
29. C = c1,  M = m3
30. C = c1,  M = m3,  B = b1
31. C = c1,  M = m3,  B = b1,  F = f1 *
32. C = c1,  M = m3,  B = b1,  F = f2 *
33. C = c1,  M = m3,  B = b1,  F = f3 *
34. C = c1,  M = m3,  B = b2
35. C = c1,  M = m3,  B = b2,  F = f1 *
36. C = c1,  M = m3,  B = b2,  F = f2 *
37. C = c1,  M = m3,  B = b2,  F = f3 *
38. C = c1,  M = m3,  B = b3
39. C = c1,  M = m3,  B = b3,  F = f1 *
40. C = c1,  M = m3,  B = b3,  F = f2 *
41. C = c1,  M = m3,  B = b3,  F = f3 *
42. C = c2
43. C = c2,  M = m1
44. C = c2,  M = m1,  B = b1
45. C = c2,  M = m1,  B = b1,  F = f1 *
46. C = c2,  M = m1,  B = b1,  F = f2 *
47. C = c2,  M = m1,  B = b1,  F = f3 *
48. C = c2,  M = m1,  B = b2
49. C = c2,  M = m1,  B = b2,  F = f1 *
50. C = c2,  M = m1,  B = b2,  F = f2 *
51. C = c2,  M = m1,  B = b2,  F = f3 *
52. C = c2,  M = m1,  B = b3
53. C = c2,  M = m1,  B = b3,  F = f1 ⊥
```

Figure 3-1: Trace of SA-SEARCH on Machine Configuration Problem

The underlying reason for which a specific value is assigned to a variable in a partial assignment function ρ is referred to as a *justification*. Formally, a *justification assignment* **S** for ρ is a function that associates with each assigned variable x_i of ρ a non-empty set of *assignment terms*. An *assignment term* is a statement of the form x_k=v where x_k is an assigned variable of ρ, and $v \in D_k$. The intent of a

justification assignment is to specify the set of assumptions underlying each assigned variable. A value v assigned to the variable x_i is said to be *assumed* (it justifies itself) if $\mathbf{S}(x_i) = \{x_i=v\}$, otherwise it is said to be *deduced*.

We characterize dependency directed backtracking with the Procedure DDSA-SEARCH shown below. In contrast to SA-SEARCH, this procedure makes use of a justification assignment **S** that associates with each assigned variable the set of assumptions underlying its current value assignment. When a constraint violation occurs, this information is used to determine the set of assumptions that led to the violation, enabling a more informed decision about what value assignment to retract.

Initially DDSA-SEARCH, in Step DDSA-1, invokes Procedure DDSA-CLOSURE. This procedure (described below) performs the "deduced" extension of the partial assignment specified by its first argument, and checks for constraint violations. When in Steps DDSA-6 and DDSA-7 the assignment of a value v to a variable x is assumed, its justification $(x=v)$ is recorded as part of the justification assignment **S**. Once a variable has been assigned a value an attempt is made to extend the resulting partial assignment into a satisficing assignment through a recursive call to DDSA-SEARCH.

Procedure DDSA-SEARCH (ρ, S)

```
DDSA-1. let <ρ', S'> := DDSA-CLOSURE (ρ, S)
DDSA-2. if fail (ρ')
            then return (fail)
DDSA-3. if all variables x_i have been assigned a value
            then return (ρ').
DDSA-4. Select x_i, a variable not assigned a value.
DDSA-5. let j = 1
DDSA-6. let <ρ", S"> = DDSA-SEARCH (ρ' [x_i = v_i,j] , S' [x_i = {x_i = v_i,j}])
DDSA-7. while fail (ρ") & j < n_i do
            begin
            j = j + 1
            let <ρ", S"> =
                DDSA-SEARCH (ρ' [x_i = v_i,j] , S' [x_i = {x_i = v_i,j}])
            end
DDSA-8. return (ρ", S")
```

In carrying out its task of constraint propagation of a partial assignment ρ, Procedure DDSA-CLOSURE makes use of the notion of an *open discrete constraint.* A discrete constraint P is called *open* under a partial assignment ρ if all variables in P but one are assigned values by ρ, and there is only one possible value v for the remaining variable x such that $x=v$ satisfies P. In this case we call the assignment of v to x the derivable assignment for P under ρ. For each open discrete constraint under the partial assignment denoted by its argument ρ, DDSA-CLOSURE attempts to successively extend ρ by the assignment derivable from that constraint.

The violation of a discrete constraint P occurs as a result of the collective assignment of specific values to the variables over which the constraint is defined. When a discrete constraint is violated, the collective set of justifications for the assigned variables of the constraint is taken to be the cause of the untenability since for each variable its value assignment was either assumption-based, and thus is contained within

the set of justifications, or deduced from assumptions represented by the set of justifications.

The collective set of justifications is constructed using the function **ASSUMPT**. When presented with a set of assigned variables, its first argument, this function extracts for each variable in the set the justification assignment specified by the second argument. It then returns the union of these results.

The set of justifications returned by **ASSUMPT** denotes a partial assignment that cannot be extended into a satisficing assignment. This set of justifications is placed in a set labeled **NOGOOD**. The set **NOGOOD** serves the role of a "memory" that is used to avoid making the same mistake twice; that is, repeating a set of assignments that is known to lead to the eventual violation of a constraint. The set **NOGOOD** is said to be *violated* by a partial assignment ρ if ρ is an extension of any partial assignment in **NOGOOD**.

In Step DDSAC-5 of Procedure DDSA-CLOSURE, the assignment of a value to a variable is deduced from an *open discrete constraint* ***d***. In this case the justification recorded consists of the union of the set of justifications associated with each of the assigned variables in the open discrete constraint ***d***.

Procedure DDSA-CLOSURE (ρ, S)

```
DDSAC-1. if NOGOOD is violated by ρ
            then return (fail)
DDSAC-2. if some constraint P is violated by ρ
            then
              begin
              let φ = the set of assigned variables in P
              let NOGOOD = NOGOOD ⊕ ASSUMPT (φ, S)[2]
              return (fail)
              end
DDSAC-3. if there are no open discrete constraints
            then return (<ρ, S>)
DDSAC-4. let d be an open discrete constraint
          let x_i = v_ij be the derivable assignment, &
          let W be the set of justifications of the assignments in d
DDSAC-5. return (DDSA-CLOSURE (ρ[x_i=v_ij], S[x_i=W]))
```

We illustrate in Figure 3-2 the behavior of the dependency directed backtracking algorithms using once again the manufacturing example. In this figure we use the symbol "===>" to indicate an application of DDSA-CLOSURE. (The line following this symbol shows the status of assignments to the variables in ρ after application of DDSA-CLOSURE.)

The advantages of dependency directed backtracking over standard backtracking can be summarized as follows. Chronological backtracking, shown in SA-SEARCH, is a depth-first tree search. In contrast, the algorithms DDSA-SEARCH and DDSA-CLOSURE use dependency information to reduce search. Specifically, in DDSA-CLOSURE either constraint propagation occurs -- in which case justifications are recorded for each assignment, or a violation is detected during constraint propagation -- in which case

[2]The operator ⊕ inserts its second operand, a set, into its first operand, also a set. For example, $\{a, b\} + \{c, d\} = \{a, b\} \cup \{\{c, d\}\}$.

```
1.  C = c1
2.  C = c1, M = m1 ===>
3.  C = c1, M = m1, F = f3 *          NOGOOD = {{c1, m1}}
4.  C = c1, M = m2 ===>
5.  C = c1, M = m2, F = f3 *          NOGOOD = {{c1, m1}, {c1, m2}}
6.  C = c1, M = m3 ===>
7.  C = c1, M = m3, F = f3 *          NOGOOD = {{c1, m1}, {c1, m2}, {c1, m3}}
8.  C = c2 ===>
9.  C = c2, M = m3
10. C = c2, M = m3, B = b1
11. C = c2, M = m3, B = b1, F = f1 ⊥
```

Figure 3-2: Trace of DDSA-SEARCH on Machine Configuration Problem

backtracking occurs. In effect, constraint violations can be detected earlier than in the tree search algorithm SA-SEARCH. Further, when a violation is detected, dependency directed backtracking focuses on the assumptions culpable for the violation. By recording sets of assumptions responsible for constraint violations as nogoods, and ensuring that no extension of such sets is attempted, parts of the search space that are guaranteed not to contain a solution are eliminated from further consideration.

3.3. Heuristic Dependency Directed Backtracking

In this section, we extend the dependency directed backtracking algorithm discussed in the last section by providing for the ability to incorporate heuristics that make use of information gleaned from the consumption of resources associated with assigned values. Incorporating heuristics into the dependency directed backtracking algorithm makes it sensitive toward discriminating among the assumed values assigned to variables in untenable situations.

A feature of the dependency directed backtracking technique is that when an untenable situation arises, the underlying set of assumptions that led to that situation are treated uniformly; no attempt is made to assess their relative culpabilities. Often there is some discretion as to which of the assumptions should be retracted. However, in order to select, and then retract, the most appropriate culprit a metric is needed to determine the relative degrees of culpability of the various underlying assumptions.

When assessing the culpability of assumptions in untenable situations, it is useful to distinguish between the two types of constraints that we have defined earlier: discrete constraints and resource constraints. In particular, if it is known that the untenability of a partial assignment resulted from the violation of a resource constraint, it becomes possible to contrast different backtracking maneuvers according to the extent to which they alleviate the violated resource constraint. The Procedure HDDSA-SEARCH presented below extends DDSA-SEARCH to allow the incorporation of different backtracking maneuvers that are sensitive to the degree of assumption culpability for the violation of a resource constraint.

Before attempting an assumption-based extension to the partial assignment denoted by it first argument, Procedure HDDSA-SEARCH first invokes Procedure HDDSA-CLOSURE. This procedure, similar to its dependency directed backtracking counterpart performs constraint propagation and checks for any constraint violation that occurs. However, unlike DDSA-CLOSURE, in the event of a constraint violation, HDDSA-CLOSURE returns one of two values that identify the type of constraint that was violated. The value $fail_{ND}$ is returned when the violation of a NOGOOD or a discrete constraint occurs. When the violation of a resource constraint occurs, the value $fail_{R}$ is returned.

In order to determine if $fail_{ND}$ or $fail_{R}$ has been returned, HDDSA-SEARCH uses the two Boolean functions $\mathbf{fail}_{ND}$ and $\mathbf{fail}_{R}$, respectively. If the current partial assignment causes the value $fail_{ND}$ to be returned, then HDDSA-SEARCH returns the value *fail*. This action is the same as that that would be taken by the dependency directed backtracking algorithm in the previous section.

If the current partial assignment leads to the violation of a resource constraint, then HDDSA-SEARCH takes several actions to remedy the violation. First using the function ASSUMPT, it determines the set of assumptions underlying the partial assignment. Using a heuristic implemented in the function CHOOSE, it next determines the assignment having the "greatest" culpability for the constraint violation. The function CHOOSE is passed four arguments that it can use in making its determination: the current partial assignment ρ, the corresponding justification assignment S, the underlying set of assumptions derived from the partial assignment A, and an identifier of the violated resource constraint. (We discuss some possible heuristics later in this section.)

Next HDDSA-SEARCH invokes the function RETRACT. This function is used to retract values assigned to a set of variables. This set of variables consists of the variable specified in the third parameter of the call to RETRACT (i.e., the value returned by CHOOSE), and those variables having values that are deduced as a result of the value assigned to this specified variable. The function call **RETRACT**(ρ, S, x), where ρ is a partial assignment with justification assignment S, and x is an assigned variable with $x=v$:

- retracts the value and justification assigned to x, and
- for each assigned variable x_i where $x=v \in S(x_i)$
  retracts the value and justification assigned to x_i

After retraction the search process continues since there still remain unassigned variables.

Procedure HDDSA-SEARCH (ρ, S)

```
HDDSA-1. let <ρ', S'> := HDDSA-CLOSURE (ρ, S)
HDDSA-2. if fail_ND (ρ')
             then return (fail)
HDDSA-3. if fail_R (ρ')
             then
               begin
               let j be identifier of violated resource constraint
               let ϕ = the set of assigned variables in ρ'
               let A = ASSUMPT(ϕ, S)
               let x = CHOOSE(ρ, S, A, j)
               let <ρ', S'> = RETRACT(ρ', S', x)
               end
HDDSA-4. if there are no unassigned variables x_i
             then return (ρ')
HDDSA-5. Select x_i, a variable not assigned a value.
HDDSA-6. let j = 1
HDDSA-7. let <ρ", S"> =
             HDDSA-SEARCH (ρ' [x_i = v_i,j] , S' [x_i = {x_i = v_i,j}] )
HDDSA-8. while fail (ρ") & j < n_i do
            begin
            j = j + 1
            let <ρ", S"> =
                 HDDSA-SEARCH (ρ' [x_i = v_i,j] , S' [x_i = {x_i = v_i,j}] )
            end
HDDSA-9. return (ρ", S")
```

One can show that the recursion in Procedure HDDSA-SEARCH always terminates. More specifically, suppose that the recursion did not terminate. In this case the procedure HDDSA-SEARCH would call itself to an infinite recursive depth. But every time the procedure calls itself recursively it either calls itself on a larger partial assignment or the Procedure HDDSA-CLOSURE returned $fail_R$. Partial assignments can be no larger than the number of variables in the problem. Therefore, in order for Procedure HDDSA-SEARCH to call itself to an infinite recursive depth Procedure HDDSA-CLOSURE must return $fail_R$ an infinite number of times. But this cannot happen because each time HDDSA-CLOSURE returns $fail_R$ it creates a *new* nogood. Since there are only finitely many possible nogoods Procedure HDDSA-CLOSURE cannot return $fail_R$ an infinite number of times.

If Procedure HDDSA-SEARCH returns a non-failing variable assignment then this assignment is a solution to the constraint problem. On the other hand, one can show that for any given variable assignment ρ and justification assignment S, if **HDDSA-SEARCH**(ρ, S) returns *fail* then no solution of the constraints satisfies the assignments in **ASSUMPT**(ρ, S). More specifically, note that if HDDSA-SEARCH returns *fail*, then it must have returned from either Step HDDSA-2 or Step HDDSA-9. If the procedure returns from HDDSA-2 then the Procedure HDDSA-CLOSURE has deduced that no solution satisfies the assignments in **ASSUMPT**(ρ, S). If the Procedure HDDSA-SEARCH returns *fail* from Step HDDSA-9 then every recursive call in steps HDDSA-7 and HDDSA-8 must return *fail*. One can assume that the recursive calls to HDDSA-SEARCH behave as specified. If all of these recursive calls return *fail* then no solution of the constraints satisfies the assignments in **ASSUMPT**(ρ', S'). But **ASSUMPT**(ρ' S') is a subset of **ASSUMPT**(ρ S).

Procedure HDDSA-CLOSURE (ρ, S)

```
HDDSAC-1. if some discrete constraint P is violated by ρ
              then
                begin
                let ϕ = the set of assigned variables in P
                let NOGOOD = NOGOOD ⊕ ASSUMPT (ϕ, S)
                return (fail_ND)
                end
          if NOGOOD is violated by ρ
              then
                return (fail_ND)
          if some resource constraint R_j is violated by ρ
              then
                begin
                let ϕ = the set of assigned variables in ρ
                let NOGOOD = NOGOOD ⊕ ASSUMPT (ϕ, S)
                return (fail_R)
                end
HDDSAC-2. if there are no open discrete constraints
              then return (<ρ, S>)
HDDSAC-3. let d be an open discrete constraint
           let x_i = v_ij be the derivable assignment, &
           let W be the set of justifications of the assignments in d
HDDSAC-4. return (HDDSA-CLOSURE (ρ[x_i=v_ij], S[x_i=W])
```

Next we identify three heuristics that could be used to select an assignment to be retracted, and the corresponding versions of **CHOOSE** used to implement them. The function **cost** invoked by **CHOOSE** returns the amount of a resource, identified by its second argument, that is consumed by the partial assignment specified by its first argument.

HEURISTIC 1: Compare all competitors of an assumed assignment and its dependents with all other assumed assignments and their dependents on the extent to which they free up the resource whose consumption has been exceeded.

Procedure CHOOSE (ρ,S,A,R)

```
C-1. for each x_i in the set of assumptions A do
          begin
          let ρ' = RETRACT(ρ, S, x_i)
          let min(i) = cost(ρ'[x_i=v_i,1], R)
          for k = 2 to n_i do
              if cost(ρ'[x_i=v_i,k], R) < min(i)
                  then min(i) = cost(ρ'[x_i=v_i,k], R)
          end
C-2. bestmin=i /*where x_i is assigned a value by ρ */
C-3. for each x_i assigned a value by ρ do
          if min(i) < min(bestmin)
              then bestmin=i
C-4. return (x_bestmin)
```

HEURISTIC 2: Compare only the assumed assignments and their dependents on the extent to which they free up the resource whose consumption has been exceeded.

Procedure CHOOSE (ρ,S,A,R)

C-1. for each x_i in the set of assumptions **A** do
begin
let ρ' = **RETRACT**(ρ, S, x_i)
let *min*(i) = **cost**(ρ' [x_i=$v_{i,j}$], R)
end
C-2. let *bestmin*=i /*where x_i is assigned a value by ρ */
C-3. for each x_i assigned a value by ρ do
if *min*(i) < *min*(*bestmin*)
then *bestmin*=i
C-4. **return** ($x_{bestmin}$)

HEURISTIC 3: Compare only the assumed assignments, ignoring dependencies, on the extent to which they free up the resource whose consumption has been exceeded.

Procedure CHOOSE (ρ,S,A,R)

C-1. let *bestmin*=i /* where x_i is assigned a value by ρ */
C-2. for each x_i in the set of assumptions **A** do
if $T_{R,i}(x_i)$ < $T_{R,bestmin}(x_{bestmin})$
then *bestmin*=*i*
C-3. **return** ($x_{bestmin}$)

The ordering of the three heuristics reflects a decreasing amount of computation required in backtracking situations. The first requires an extensive enumeration of all possible backtracking maneuvers and computation of resource requirements associated with each maneuver, whereas the last one uses a simple scheme requiring little computation.

The behavior of the heuristic algorithms with the manufacturing example is illustrated in Figure 3-3. In this simple case, the three heuristics for CHOOSE result in the same behavior.

```
1. C = c1
2. C = c1, M = m1 ===>
3. C = c1, M = m1, F = f3 *              NOGOOD = {{c1, m1}}
4. C = c2
5. C = c2, M = m1
6. C = c2, M = m1, B = b1
7. C = c2, M = m1, B = b1, F = f1 ⊥
```

Figure 3-3: Trace of HDDSA-SEARCH on Machine Configuration Problem

As shown by this figure, C (i.e., selecting a different configuration for the cutting machine) is preferred to M as the backtracking point since it "better" addresses the particular constraint violated, the constraint involving the scarce resource. With large problems, we have found the savings in search resulting from the application of such heuristic knowledge substantial.

4. Conclusion

The importance of effective heuristics for reducing search has been long recognized in the AI and Management Science literature. In this paper, we have provided a formal specification for an important search technique, namely, dependency directed backtracking, as it applies to satisficing assignment problems. Many search problems in various problem domains can be viewed as instances of the SAP.

We have also generalized the classical SAP and described several heuristics that can be used to limit the amount of backtracking that is associated with search. Traditionally, satisficing assignment problems have been defined in terms of what we have called discrete constraints; we have extended this problem definition by providing for resource consumption constraints. In this extended problem definition, a satisficing solution is one that satisfies the discrete and resource constraints. With respect to the resource constraints, it is necessary that the assignment not result in the use of more of a resource than is available.

The inclusion of the resource consumption constraint allows for a further extension of the problem. Specifically, an objective function can be formulated to specify that the solution must be "optimal" in its use of the resources. For example, an objective function could state that minimum amounts of resources be consumed, subject to the resource and discrete constraints. In this way, the satisficing problem becomes transformed into an optimization problem. This transformed problem can be solved using the *branch and bound* technique augmented with knowledge about the NOGOOD set. Specifically, by using the NOGOOD set, the branch and bound procedure can rule out paths that are known to be untenable. However, this efficiency is achieved at the expense of an additional structure that grows with each backtracking maneuver. As a next step in this research, we intend to study these tradeoffs in more detail.

REFERENCES

Bobrow, D.G., and Raphael, B., New Programming Languages for AI Research, *Computing Surveys*, 6, 1974, pp. 153-174.

de Kleer, J., An Assumption-based TMS, *Artificial Intelligence*, vol 28, no.2, March 1986.

de Kleer, J., Doyle, J., Steele, G. and Sussman, G., AMORD : Explicit Control of Reasoning, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, 1977.

Dhar, V., and Quayle C., An Approach to Dependency Directed Backtracking Using Domain Specific Knowledge, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

Doyle, J., A Truth Maintenance System, *Artificial Intelligence*, June, 1979.

Fikes, R.E., REF-ARF: A System for Solving Problems Stated as Procedures, *Artificial Intelligence*, volume 1, number 1, 1970, pp. 27-120.

Gashnig, J.A., A Constraint Satisfaction Method for Inference Making, *Proceedings of the 12th Annual Conference on Circuit System Theory*, University of Illinois, Urbana-Champaign, 1974.

Gashnig, J., Performance Measurement and Analysis of Certain Search Algorithms, Ph.D Dissertation, Technical Report CMU-CS-79-124, Department of Computer Science, Carnegie-Mellon University, 1979.

Goodwin, J.W., A Process Theory of Non-Monotonic Inference, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, 1985.

Haralick, R.M., and Shapiro, L., The Consistent Labeling Problem: Part I, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 2, number 3, 1980.

Haralick, R.M., and Elliot, G.L., Increasing Tree Search Efficiency for Constraint Satisfaction, *Artificial Intelligence*, volume 14, number 1, August 1980, pp. 263-313.

Knuth, D., Estimating the Efficiency of Backtrack Programs, *Mathematics of Computation* 24 (129), 1975, pp. 121-136.

Mackworth, A., Consistency in Networks of Relations, *Artificial Intelligence*, 8 (1), 1977, pp. 99-118.

McAllester, D., Reasoning Utility Package, AI Laboratory Memo 667, April 1982.

Montanari, U., Optimization Methods in Image Processing, *Proceedings IFIP Congress*, North-Holland, 1974, pp. 727-732.

Nudel, B., Consistent Labeling Problems and Their Algorithms: Expected-Complexities and Theory-Based Heuristics, *Artificial Intelligence*, **21**, 1983, pp. 135-178.

Simon, H., *Administrative Behavior*, Free Press, 1947.

Stallman, R. and Sussman, G., Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, volume 9, No.2, October 1977, pp 135-196.

Waltz, D., Understanding Line Drawings of Scenes With Shadows, in: P.H. Winston (Ed.), *The Psychology of Computer Vision*, McGraw-Hill, New York, 1975, pp. 19-91.