Lecture Notes for 6.824, Artificial Intelligence
Rev. October, 1992

Constraint Satisfaction Search

For the past thirty five years artificial intelligence researchers have been studying heuristic search techniques. People writing AI programs have had strong intuitions about what it means for a program to search. Intuitively they classified programs according to the kinds of search performed. Certain programs, such as matrix multiplication routines, did no search whatsoever. Other programs searched for solutions to a fixed finite set of constraints. Others searched for paths in graphs, or for strategies in game trees, or for proofs in formal inference systems. The intuitive classification of search programs seems to roughly correspond to complexity classes. The no-search procedures correspond to the complexity class P. Constraint satisfaction search procedures correspond to the complexity class NP, graph search and game search procedures to the class PSPACE, and search in theorem proving to the class of recursively enumerable functions. This chapter discusses heuristic techniques for solving search problems of the simplest type — constraint satisfaction problems. Heuristic techniques for other forms of search are discussed in later chapters.

1 Constraint Satisfaction Problems

Intuitively, a search problem involves finding an object that satisfies a given specification. A classic example is the eight queens problem. This is the problem of placing eight queens on a chess board so that no two queens attack each other, i.e., so that no single row, column, or diagonal contains more than one queen. This problem can be solved (inefficiently) by a "generate and test" program that first generates a list of all the possible ways eight queens can be placed a chess board and then tests each possibility to see if it is a solution to the problem.

Another example is the **SAT** problem. This is the problem of determining whether a set of Boolean clauses is satisfiable. A Boolean variable is a variable that can have one of two values, **True** or **False**. We often use the letters P and Q to represent Boolean variables. A *literal* is either a Boolean variable of the negation of a Boolean variable. A *Boolean clause* is a disjunction of Boolean literals. For example, $\neg P \vee Q$ is a clause containing the two literals $\neg P$ and Q. The clause $\neg P \vee Q$ is satisfied if either P is **False** or Q is **True**. A **SAT** problem consists of a set of Boolean clauses. The problem is to determine if there exists an interpretation of the Boolean variables that satisfies all of the given clauses. This is a search problem — it can be solved by a generate and test search process. One generates all possible assignments of truth values to Boolean variables and tests each assignment to see if it satisfies every clause.

Search can be very expensive, i.e., it can require a large amount of computation time. Note that if there are n Boolean variables in a **SAT** problem then there are 2^n possible assignments of truth values to these variables. Searching all these truth assignments will take time proportional to 2^n — the time taken by a simple search program is exponential in the number of Boolean variables. It turns out that all the known algorithms for determining Boolean satisfiability require exponential time. Intuitively, this means that every known technique for solving **SAT** problems essentially searches the space of truth assignments.

To develop a more refined understanding of search problems some more precise terminology will be useful. First, we can give a formal definition of what we mean by procedures that do not search. Formally a "no-search procedure" is one that only requires polynomial time. An procedure is said to require polynomial time if there exists a polynomial $f(n)$ (like $cn^3 + bn^2$) such that, for inputs of size n, the algorithm always terminates in less than $f(n)$ time. Note that a generate and test procedure for solving **SAT** is not polynomial time — it requires time proportional to 2^n where n is the number of the Boolean variables and the function 2^n grows faster than any polynomial. A procedure that terminates in time less than $2^{f(n)}$ for some polynomial $f(n)$ is said to execute in *exponential* time. By definition, search procedures require at least exponential time. It is important to note that this classification of procedures into search and no-search procedures concerns the worst case time. If a procedure terminates in linear time on 99.99% of all inputs, but requires exponential time on the remaining .01%, it is considered to be a search procedure.

It would be a mistake to assume that a problem, like **SAT**, is really intractable in practice just because all known algorithms for solving the problem take exponential time in the worst case. Many such problems only take polynomial expected time. In fact, for many natural probability distributions the **SAT** problem only requires polynomial average time [Goldberg, 1979], [Purdom, 1983]. Other problems that require search, such as finding good moves in a chess game, can be solved in practice by using the most efficient known search procedures combined with powerful high speed computation. The need to search is not fatal! This chapter, and those that follow, discuss general purpose techniques for improving the efficiency of search procedures.

There is a significant body of AI literature on constraint satisfaction problems; a survey can be found in [Pearl and Korf, 1987]. To more precisely define the notion of a constraint satisfaction search problem we first define the concept of a domain variable.

> **Definition:** A *domain variable* is a pair $<x, D>$ where x is a variable (a symbol) and D is a finite set called the domain of the variable.

By abuse of notation we use the letters x, y and z to denote domain variables. It should be remembered that a domain variable x is actually a pair of a token and a set (a domain). In the eight queens problem one can define eight domain variables $Q_1, \cdots, Q_8$, one variable for each column of the chess board. The variable Q_i represents the position of the i'th queen, i.e., Q_i will represent the row on which the queen of the i'th column is to be placed. The queen on the i'th column can be placed in any one of rows 1 through 8 so the domain of Q_i is the set of numbers $\{1, 2, \cdots, 8\}$.

> **Definition:** A *variable interpretation* for a given set of domain variables is a mapping ρ from the variables to values such that $\rho(x)$ is always an element of the domain of the variable x.

A constraint satisfaction problem consists of a set of domain variables and a ste of constraints on those variables. In the 8-queens problem there is one constraint for each pair of variables — the interpretation of Q_i and Q_j must be such that the two queens do not attack. In a **SAT** problem the variables are Boolean and each clause is a constraint on the variables contained in that clause. A constraint involving the domain variables $x_1, x_2, \cdots, x_n$ will often be written as $\Phi(x_1, x_2, \cdots, x_n)$.

A constraint can be represented by a predicate and domain variables. The constraint that the queen Q_i does not attack the queen Q_j can be represented as the triple $<\neg\texttt{Attacks}_{i,j}, Q_1, Q_2>$ where $\neg\texttt{Attacks}_{i,j}$ is the predicate which is true of two numbers n and m if a queen on row i and column n does not attack a queen on row j and column m. In general, a constraint can be represented as a tuple $<P, x_1, x_2, \cdots, x_k>$ where P is a predicate of k arguments. This constraint holds under a particular variable interpretation if the predicate P is true of the values of the variables $x_1, \ , x_2, \ \cdots, \ x_k$. Formally, we will assume that the predicates used in representing constraints are defined by an explicit table which states, for each possible tuple of argument values, whether the predicate is true or false on those arguments. In practice the predicate is usually specified with a computer program such as a Lisp procedure. Note that individual constraints usually only involve a small subset of the variables involved in the overall constraint satisfaction problem.

Definition: Let $D_1, \cdots, D_k$ be finite sets (variable domains). A *tabular predicate* on $D_1, \cdots, D_k$ is a k-dimensional table which specifies a truth value for each tuple $<v_1, \cdots, v_k>$ where v_i is an element of D_i.

Definition: A *constraint* is a tuple $<P, x_1, \cdots, x_k>$ where each x_i is a domain variable and P is a tabular predicate on the domains of $x_1, \cdots, x_k$.

Definition: A variable interpretation ρ *satisfies* a constraint $<P, x_1, \cdots, x_k>$ if the tabular predicate P is true of the tuple $<\rho(x_1), \cdots, \rho(x_k)>$.

Definition: A *constraint satisfaction problem* (CSP) is a set of constraints. A variable interpretation satisfies a CSP, and is called a solution of that CSP, if it satisfies each constraint in the CSP.

It is possible to show that the problems of determining if a given CSP has a solution has the technical property of being NP complete.[1][2] This is strong evidence that any procedure for solving an arbitrary CSP must search (must take exponential time in the worst case). However, as we shall see in the next section, some search procedures are more efficient than others.

2 Value Propagation

Consider a constraint satisfaction problem involving domain variables $x_1, \cdots, x_n$. To find a solution to the constraint satisfaction problem one can search the set of possible assignments of values to variables. The set of possible variable assignments can be organized into a tree. At the root of the tree no commitment has been made about the values of the variables. Each branch from the root of the tree corresponds to a particular value for the first variable — if there are five possible values for x_1 then there are five branches from the root node. The second level of branching in the tree corresponds to the possible values of x_2 and so on. The leaves of the tree correspond to complete variable assignments.

Given values for some, but not all, variables it may be possible to use inference techniques to determine that no solution is consistent with the given values. In other words, inference can be used to do more effective consistency testing at intermediate nodes of the search tree [Mackworth, 1977], [Freuder, 1985]. To formalize the notion of inference we first formalize the information

[1]For a thorough presentation of NP completeness see [Garey and Johnson, 1979].

[2]Because 3-**SAT** is NP-complete, constraint satisfaction is NP-complete even in the case where each domain variable has at most two values and each constraint involves at most three variables. If every constraint involves only two variables and every variable has a domain of just two values then the satisfiability problem is solvable in polynomial time (by reduction to 2-**SAT**). If every constraint involves two variables but variables can have three values then determining the existence of a solution is NP-complete (be reduction of 3-**SAT** where clauses are mapped to variables.)

that is present at an intermediate node of the search tree. Each intermediate node of the search tree represents a partial assignment of values to variables.

> **Definition:** A *partial assignment* is a set of equations of the form $\{x_1 = v_1, \cdots, x_k = v_k\}$ where v_i is an element of the domain of x_i. A *complete assignment* for a given set of constraints is a partial assignment that contains an assignment for every variable appearing in the constraints. If ρ is a partial assignment, then a *completion* of ρ is a complete assignment that contains ρ.

For technical reasons it is convenient to allow partial assignments to contain more than one value for the same variable. However, an assignment that contains more than one value for the same variable will be called *inconsistent*.

> **Definition:** A partial assignment will be called *inconsistent* if it contains more than one value for the same variable. A partial assignment which is not inconsistent will be called *consistent*.

A consistent complete assignment is just a representation for a variable interpretation as defined in the previous section. If ρ is a partial assignment that represents the information present at an internal node of the search tree, then a consistent completion of ρ represents the information that is present at some leaf node under that internal node.

Any constraint satisfaction problem can be viewed a "network" of variables and constraints. In this network each variable is connected to the constraints that involve it, and each constraint is connected to the variables it involves. A network representation for a constraint satisfaction problem is shown in figure 1. In figure 1 variables are represented by circles and constraints are represented by boxes. Figure 1 represents both a constraint network and a partial assignment of values to variables. A variable that has not yet been assigned a value is labeled with "??".

Given a partial assignment to the variables in a constraint network one can often infer values that are "forced" for other variables. For example, consider

Figure 1: A Constraint Network

a **SAT** problem that contains the clause $\neg P \vee \neg Q \vee W$ and consider a partial assignment that assigns both P and Q **False** but provides no assignment for W. Any extension of this partial assignment to a complete solution must assign W the value **True**. In this case one can extend the assignment so that W is assigned the value **True** without fear of omitting any solutions. This kind of inference appears to be essential to performing constraint satisfaction search efficiently.

> **Definition:** Let ρ be a partial assignment and let Φ be a constraint. We say ρ and Φ *entail* a (new) equation $x = v$ if the equation $x = v$ is contained in every consistent completion of ρ that satisfies Φ.

The inference of new equations naturally leads to a propagation process in which the inference of one equation can justify the inference of a second equation and so on. For example, if we have the Boolean constraints $P_1 \rightarrow P_2$, $P_2 \rightarrow P_3$, $\cdots$ $P_{k-1} \rightarrow P_k$, and we have a partial assignment in which P_1 is assigned **True**, then we can infer that P_2 must be assigned **True**, and hence P_3 must be assigned **True** and so on up to P_k. When searching for a solution to a constraint satisfaction problem the partial assignments at each node of the search tree can be closed under constraint propagation inference of this type.

> **Definition:** Let $\mathcal{C}$ be a set of constraints and let ρ be a partial assignment. We say that an equation $y = v$ is *derivable by value propagation* from ρ and $\mathcal{C}$ there is a single constraint Φ in $\mathcal{C}$ such that ρ and Φ entail $y = v$.
>
> **Definition:** The *value propagation closure* of a partial assignment ρ with respect to a constraint set $\mathcal{C}$ is the least partial assignment ρ' such that every equation derivable by value propagation from ρ' and $\mathcal{C}$ is already contained in ρ'.

Value propagation adds labels that can be derived from existing labels and a single constraint. For example, in Figure 1 value propagation may be be able to derive an equation of the form $x_3 = v_3$ from the equations $x_1 = v_1$, $x_2 = v_2$, and the constraint C_1. Given an equation of the form $x_3 = v_3$, one may then be able to use the equations $x_3 = v_3$, $x_4 = v_4$, and the constraint C_2 to derive a new equation of the form $x_5 = v_5$. This process continues until no new equations can be derived (or until an inconsistency is discovered as described below).

It is possible for value propagation to discover an inconsistency. For example, suppose that the constraint set $\mathcal{C}$ contains the constraints $x = a \rightarrow y = b$ and $x = a \rightarrow y = c$ where b and c are distinct values. Now suppose that ρ contains the equation $x = a$. In this case the value propagation closure of ρ contains both the equation $y = b$ and the equation $y = c$ and hence is inconsistent. In practice value propagation can be terminated whenever an inconsistency is discovered.

Each variable in a CSP can be represented by a data structure that contains a field for the value, if any, assigned to that variable. Each variable data structure can also contain a list of all of the constraints in the network that mention that variable. Given this representation of variables, and the obvious representation of constraints, the following procedure can be used to compute the value propagation closure of a partial assignment. Actually, the procedure only returns the closure if the closure is consistent. If the closure is inconsistent then the procedure returns the token "inconsistent".

Value Propagation Procedure:

1. Initialize `Queue` to be a list of all the constraints in the network.
2. If the current partial assignment is inconsistent the terminate and return "inconsistent".
3. If `Queue` is empty then return the current partial assignment.
4. Remove a constraint Φ from `QUEUE`. For each equation that can be derived from Φ and existing equations, update the variable data structures to incorporate the new equations.
5. For each updated variable in step 4 (for each variable where the derived

equation was not already present) add all constraints involving that variable to the list `QUEUE`.

6. Goto 2.

If there is a bound on the size of variable domains and a bound on the number of variables in a single constraint, for example every variable ranges over at most three values and every constraint involves at most four variables, then the above procedure runs to completion in time linear in the number of constraints in the network. To see this note that, given upper bounds on the domain size of variables and the number of variables in a constraint, there exists an upper bound on the amount of time taken by step 4 of the procedure. Given that an individual execution of step 4 takes constant time, it is not difficult to verify that the time taken by the overall procedure is proportional to the total number of times a constraint is added to the queue. But the upper bound on the number of variables in a constraint places an upper bound on the number of times a given constraint can be added to the queue. Thus the total number of times a constraint is added to the queue is bounded by a constant times the number of constraints.

Constraint propagation is an inference process — it infers new equations from constraints and existing equations. Furthermore, constraint propagation is very efficient — for bounded constraint size it can be run to completion in time linear in the number of constraints. Unfortunately, constraint propagation is not complete. To understand completeness we need the following definition.

> **Definition:** Let ρ be a partial assignment and let $\mathcal{C}$ be a constraint set. We say that ρ and $\mathcal{C}$ *entail* an equation $x = v$ if every consistent completion of ρ that satisfies $\mathcal{C}$ contains $x = v$.

Constraint propagation is incomplete. This means that it is possible that ρ and $\mathcal{C}$ entail $x = v$ but that the value propagation closure of ρ does not contain $x = v$. For example, let $\mathcal{C}$ consist of the two constraints $x = a \rightarrow y = c$ and $x = a \rightarrow y = d$. Suppose that the domain of x is the set $\{a, b\}$ and that the domain of y is the set $\{c, d\}$. In this case, any variable interpretation that satisfies these two constraints must interpret x as b, because if x is interpreted as a then any particular interpretation of y violates one of the given constraints. However, if no equations have yet been given, then constraint propagation will not derive any new equations because no *single* constraint entails a new equation. Thus, although $x = b$ is entailed by the constraints, constraint propagation can not derive it.

There is a good reason for the incompleteness of constraint propagation. For bounded constraint size, constraint propagation terminates in linear time in the size of the constraint network. However, even for bounded constraint size, determining the existence of a solution to a constraint satisfaction problem is NP-complete. If constraint propagation were complete then it is not difficult to show that we would have a polynomial time procedure for an NP-complete problem. Assuming P$\neq$NP, there can not exist any such procedure.

Even with value propagation, solving a CSP requires search. The search can be done in such a way that it forms a tree where each branch in the tree corresponds to the possible values of some variable. Each node in the tree contains a partial assignment. Value propagation is used to close the partial assignment present at each node. Because many values can be assigned by value propagation rather than by branching, the search tree based on value propagation is usually much smaller than than the naive search tree which explores all possible assignments.

3 Arc Consistency

There are various polynomial time (and hence semantically incomplete) constraint propagation inference procedures for CSPs. The procedures vary in cost and strength — there are cheap weak procedures and expensive strong procedures. Strong procedures draw more conclusions but take more time to do it. Weak procedures draw fewer conclusion but find those conclusions more quickly. The value propagation procedure defined in the previous section is cheap and weak. In this section we define a stronger but more expensive procedure — arc consistency propagation. Arc consistency involves maintaining more information than just a simple partial assignment. In particular, arc consistency keeps track of values that have been ruled out as well as values that have been determined.

Definition: A *disequation* is a negation of an equation, e.g., an expression of the form $x \neq v$. A *CSP knowledge state* is a set of equations and disequations. A CSP knowledge state is called *inconsistent* if it either contains two different equations for the same variable or if it contains both $y = v$ and $y \neq v$ for some variable y and value v.

I will use the symbol Σ to denote a CSP knowledge state.

Definition: A *completion* of a CSP knowledge state Σ is a variable interpretation that satisfies the equations and disequations in Σ.

Definition: A CSP knowledge state Σ and a constraint Φ *entail* an equation $x = v$ if every completion of Σ that satisfies Φ assigns x the value v. Σ and Φ entail a disequation $x \neq v$ if every completion of Σ that satisfies Φ assigns x some value other than v.

Definition: A knowledge state Σ entails an equation $y = v$ by *exhaustion of alternatives* if for every value w other than v in the domain of y the knowledge state Σ contains the disequation $y \neq w$.

Definition: Let $\mathcal{C}$ be a set of constraints and let Σ be a CSP knowledge state. We define the *Arc consistency closure* of Σ with respect to constraint set $\mathcal{C}$ to be the least CSP knowledge state Σ' of equations and disequations satisfying the following conditions.[3]

[3]The term "arc consistency" comes from the case where every constraint involves only two variables. In this case the constraint network defines a graph where the nodes are variables and there is an arc between any two nodes that are involved in the same constraint. Arc consistency is the property that the CSP knowledge state is consistent with each individual arc in the graph — for each individual arc there exists a completion of the knowledge state that satisfies that arc.

- Σ' contains Σ.
- If Σ' together with *some single constraint* in $\mathcal{C}$ entail a disequation $x \neq v$, then Σ' also contains the disequation $x \neq v$.
- If Σ' entails $y = v$ by exhaustion of alternatives, then Σ' contains $y = v$.

Arc consistency is strictly stronger than value propagation. Although the specification of when one can derive an equation appears weaker — one can only derive an equation when all other values have been ruled out — if a new value does semantically follows from the knowledge state and a single constraint then all other values will be ruled out and the new remaining value will be derived. Actually, arc consistency can be implemented as an algorithm that only derives disequations — equations are implicitly present when all but one value has been eliminated. To see that arc consistency is in fact stronger than value propagation one can simply examine the 8-queens problem. In the 8-queens problem no single constraint between two queens can force a value of one of the queens and so value propagation can never derive a new equation. However it is easy to construct a case where arc consistency derives new values.

The procedure for value propagation given above can be modified to perform arc consistency propagation. An analysis similar to that given above can be used to show that if an upper bound is placed on the size of the variable domains, and an upper bound is placed on the number of variables in each constraint, then the procedure runs in time linear in the number of constraints. However, arc consistency propagation is more expensive than value propagation. To see this we can include the number of domain values as an explicit parameter in the analysis of the running time. For now we assume that each constraint involves at most two variables. A CSP in which every constraint involves at most two variables is often called a *binary* CSP. Let e be the number of constraints in a binary CSP (e is the number of edges in the graph representation of the binary CSP). Let d be an upper bound on the number of values in the domain of each variable. The value propagation closure of a partial assignment can be computed in time proportional to de. The best known algorithm for computing the arc consistency closure of a CSP knowledge state has a worst case running time proportional to d^2e

[Mohr and Henderson, 1986]. Note that for a bounded value of d both of these running times are linear in e (the number of constraints). However, for nontrivial values of d value propagation is considerably faster (but weaker) than arc consistency propagation.

4 Generalized Forward Checking (GFC)

A third propagation algorithm, known as generalized forward checking (GFC), is often better in solving CSPs than either value propagation or arc consistency. GFC is intermediate in strength between value propagation and arc consistency.

> **Definition:** Let $\mathcal{C}$ be a set of constraints and let Σ be a CSP knowledge state. We define the *generalized forward checking closure* of Σ with respect to constraint set $\mathcal{C}$ to be the least set Σ' of equations and disequations satisfying the following conditions.[4]
>
> 1. Σ' contains Σ.
> 2. If the equations in Σ' (ignoring the disequations) together with a single constraint in $\mathcal{C}$ entail a disequation $x \neq v$, then Σ' also contains the disequation $x \neq v$.
> 3. If Σ' entails $y = v$ by exhaustion of alternatives, then Σ' contains $y = v$.

Condition 2 in the above definition is weaker than the corresponding condition in the definition of arc consistency. Arc consistency is based on the

[4]The term "generalized forward checking" comes from viewing this procedure as a generalization of a much simpler technique called "forward checking". Forward checking is a consistency test rather than a propagation procedure. It simply checks that for each unassigned variable y, and for each constraint Φ involving y, there exists a value in the domain y which is consistent with Φ and the existing equations for other variables in the constraint Φ. Generalized forward checking converts this consistency test into a propagation procedure which generates new equations.

derivation of disequations from disequations. Disequations carry more information than equations — disequations can specify partial information about a variable even when no equation is known. GFC keeps track of disequations, but only derives disequations from the known equations. GFC is strictly stronger than value propagation. If an equation follows from other equations and a single constraint then GFC will derive that equation. Unlike value propagation, GFC can derive new equations in the 8-queens problem. There are also examples that show that arc consistency is strictly stronger than GFC, although these examples are somewhat more difficult to find (see the exercises at the end of this section).

Recall that a binary CSP is one in which each constraint involves at most two variables. In a binary CSP it is possible to compute the GFC closure of a knowledge state in de time where d is an upper bound on the size of the domain of each variable and e is the number of constraints. This is the same bound as for value propagation. Since GFC is considerably more powerful than value propagation, and not significantly more costly, GFC is generally preferable to value propagation.[5] For most problems GFC is nearly as powerful as arc consistency, and its running time of de is considerably better than arc consistency's time of d^2e.

5 Restricted GFC

Consider nonbinary CSPs, i.e., ones in which the number of variables per constraint can be larger than two. Let e be the number of constraints, d an upper bound on the number of values in variable domains, and a (for arity) be an upper bound on the number of variables in each constraint. Value propagation and GFC can be implemented to run in time $d^{a-1}e$ while the best known algorithm for arc consistency can be run in time proportional to

[5]We are assuming that constraints are represented by tables. Some constraints, such as numerical constraints, are not represented as tables. For numerical constraints the domain sizes are either infinite or finite but very large. For numerical constraints value propagation is preferable to GFC. However, numerical constraints are usually best handled with bounds propagation rather than any of techniques described here. Bounds propagation is described in the chapter on nondeterministic lisp.

$d^a e$. When a is greater than two all of these procedures are nonlinear in d. It is possible to construct another inference procedure which is linear in d even for a greater than two. This new procedure will be called *restricted GFC*.

> **Definition:** A constraint Φ will be called *active* relative to a knowledge state Σ if all but one of the variables in Φ have been assigned a value in Σ.
>
> **Definition:** Let $\mathcal{C}$ be a set of constraints and let Σ be a CSP knowledge state. We define the *Restricted GFC closure* of Σ with respect to constraint set $\mathcal{C}$ to be the least set Σ' of equations and disequations satisfying the following conditions.
>
> - Σ' contains Σ.
> - If the equations in Σ' (ignoring the disequations) together with some single *active* constraint in $\mathcal{C}$ entail a disequation $x \neq v$, then Σ' also contains the disequation $x \neq v$.
> - If Σ' entails $y = v$ by exhaustion of alternatives, then Σ' contains $y = v$.

For binary CSPs restricted GFC is essentially identical to unrestricted GFC.[6] Restricted GFC can be implemented in time proportional to $(d + a)e$. Note that in the case where a is greater than two this is a considerable improvement over $d^{a-1}e$, the running time of GFC. This improvement in running time is gained at the cost of a loss of some inferential power. Restricted GFC is strictly weaker than GFC and hence strictly weaker than arc consistency. Restricted GFC is incomparable with value propagation — there are inferences that will be made by value propagation that will not be made by restricted GFC and, more commonly, inferences made by restricted GFC that will not be made by value propagation.

[6]The one exception is the case where a constraint can be used to rule out a value even in an empty knowledge state — this does not generally arise in practice.

6 Nonlinear Propagation Algorithms

All of the propagation algorithms discussed above run in time linear in the number of constraints (assuming bounded variable domain size and a bounded number of variables per constraint). There are other procedures which are inferentially more powerful but which require superlinear time. A hierarchy of propagation procedures based on the notion of k-consistency is defined by Mackworth [Mackworth, 1977]. The larger the value of k the more powerful, and the more costly, the propagation procedure. All of the propagation procedures run in polynomial time but the order of the polynomial increases with k.

Other hierarchies of propagation procedures are possible. Let C be any propagation algorithm (such as value propagation or GFC). If Σ is a knowledge state then we let $C(\Sigma)$ is the knowledge state that results from applying the propagation algorithm C to the knowledge state Σ. Recall that all the propagation procedures discussed in this chapter run in polynomial time and are semantically incomplete. To find a solution to a CSP one must still search. Let Σ be the knowledge state at a node in the search tree. Since the search might fail, it must be possible that extending Σ with an equation $y = v$ leads to an inconsistent state, i.e., that $C(\Sigma \cup \{y = v\})$ is inconsistent. At the node with knowledge state Σ we could "look ahead" in the search tree and see that $y = v$ leads to a contradiction. This should allow one to add the disequation $y \neq v$ to the knowledge state at that node. We can define a stronger propagation procedure that infers all disequations that can be inferred by this kind of lookahead. First we define the general notion of a closure operator on knowledge states. Each constraint propagation algorithm defines such a closure operator.

Definition: A closure operator on knowledge states is a mapping C form knowledge states to knowledge states such that $C(\Sigma)$ contains Σ, $C(C(\Sigma))$ equals $C(\Sigma)$ and if Σ is a subset of Γ then $C(\Sigma)$ is a subset of $C(\Gamma)$.

Definition: Given a closure operator C on knowledge states we define $L(C)$ to be the closure operator such that $L(C)(\Sigma)$ is the least set Σ' satisfying the following conditions.

- Σ' contains Σ.
- Σ' is closed under C, i.e., Σ' contains $C(\Sigma')$.
- If $C(\Sigma' \cup \{y = v\})$ is inconsistent then Σ' contains the disequation $y \neq v$.
- If Σ' entails $y = v$ by exhaustion of alternatives, then Σ' contains $y = v$.

The closure operator $L(C)$ adds a level of "look ahead" to the closure operator C. If C runs in amortized time T then $L(C)$ can be made to run in amortized time ndT where n is the number of variables and d is an upper bound on variable domain size. So if C is polynomial then so is $L(C)$, but the order of $L(C)$ is larger than the order of C. We define $L^k(C)$ to be $L(L(\cdots L(C)\cdots))$ with k applications of L. $L^k(C)$ is the k-fold look ahead operator, i.e., the operator which looks ahead k steps in the search tree. As k increases the k level look ahead propagator becomes more powerful. However, the cost of running the propagator also increases.

To the author's knowledge, nonlinear propagation procedures have never been shown to be useful in solving CSPs efficiently. For most applications restricted GFC appears to be the most effective propagation procedure.

7 Applications to Vision (Line Labeling)

Constraint propagation was introduced by Waltz to help in the visual interpretation of line drawings [Waltz, 1975]. Consider the line drawing shown in figure 2. This line drawing represents a two dimensional projection of a set of three dimensional objects. Waltz considered making the following assumptions.

- The three dimensional objects are polyhedra, i.e., objects with flat faces.
- Each line in the drawing represents an edge of one of the polyhedra (we will ignore shadows, cracks, and changes of color).

Figure 2: A simple line drawing

Figure 3: The four vertex types

Given a particular three dimensional interpretation of the image each line can be given one of four labels. If the line represents a convex edge of a polyhedra, and the two faces of the polyhedra which meet at the line are visible in the image, the the line is labeled $+$. If it represents a concave edge, and the two meeting planes are visible, then it is labeled $-$. If the line represents an edge such that only one of the two meeting faces is visible in the image then the line is labeled with an arrow pointing along the line. The arrow is oriented so that the visible face is clockwise from the head of the arrow. The drawing in figure 2 has been labeled according to the natural physical interpretation.

Figure 4: The constraint imposed by each vertex type

Waltz also made an additional assumption that the objects are in general position relative to the viewer, i.e., there are no coincidental alignments of vertices or edges of different objects, although objects are allowed to rest on one another in a face-to-face manner. Under the general position assumption we need only consider line drawings in which at most three lines meet at any give vertex. We can classify all vertices into three kinds, Ls, arrows, Forks and Ts as shown in figure 3. Furthermore, each kind of vertex has a limited number of possible three dimensional labelings. The allowed labelings for each kind of vertex is shown in figure 4.

The assumptions Waltz made about the physical objects and their projection onto the image allowed him to convert the line drawing into a constraint satisfaction problem. The line drawing can be viewed as a CSP in which the variables are the lines of the drawing. Each variable (line) has one of four possible values, $+$, $-$, or an arrow in one of two possible directions. Figure 4 gives a way of interpreting each vertex in a line drawing as a constraint on the labels assigned to the lines meeting at that vertex. We will also assume that the lines on the periphery of the drawing are labeled with arrows in a clockwise direction around the image (the physical objects occur only inside the periphery of the image).

Figure 5 shows the result of value propagation closure on the initial partial assignment of one image. In this case, and for most line drawings, value propagation is complete, it derives all labels that are implied by the vertex

Figure 5: An example of propagation

constraints. In general, however, value propagation, and even arc consistency, is not complete for line labeling CSPs. In fact it has been shown that the problem of determining the existence of a solution to a line labeling CSP is NP-complete [Kirousis and Papadimitriou, 1988]. So in general finding consistent line labelings requires exponential search.

8 Problems

1. Give an partial placement of queens on a 6x6 chessboard where GFC derives an equation that value propagation fails to derive. Place an x on each square ruled out by disequations derivable by GFC.

2. Give a partial placement of queens on a 6x6 chessboard where arc consistency propagation derives an equation not derived by GFC. Draw the configuration twice. In one drawing show all the information derived by GFC including squares ruled out and derived placements. In the other drawing show all the information derived by arc consistency propagation.

3. If all solutions to a constraint satisfaction problem assign the variable X the value x then we say that $X = x$ is *entailed by* the constraints. A constraint propagation algorithm is called *complete* if it can derive all assignments that follow from the constraints. This is the same notion of completeness that is used for inference rules in general — inference rules are complete if, given any set of premises, they can be used to derive all assertions semantically entialed by the premises. Give an example of a Waltz line drawing where value propagation is incomplete, i.e., there is an assignment that is entailed by the constraints but value propagation will not derive that assignment.

4. Give an example of a Waltz line drawing where arc consistency propagation is incomplete. (This is difficult.)

5. Give a proof that in Boolean CSPs, i.e., in CSPs in which every variable has only two possible values, the procedures value propagation, arc consistency propagation, and GFC are all equivalent.

6. Show that arc consistency can be run on **SAT** problems in linear time

(time proportional to the sum over all clauses of the number of literals in the clause). Arc consistency on **SAT** problems is more commonly known as unit resolution.

7. Give a method of translating an arbitrary CSP into a **SAT** problem such that the **SAT** problem is satisfiable if and only if the CSP has a solution.

8. Give a reduction from CSPs to **SAT** problems as in 7 which has the additional property that running arc consistency on the **SAT** problem corresponds to running GFC on the original CSP. More formally, running arc consistency on the **SAT** problem should generate an inconsistency if and only if running GFC on the original CSP generates an inconsistency. Give a proof that your reduction satisfies these criterion.

9. Repeat problem 8 but give a reduction where running arc consistency on the **SAT** problem corresponds to running arc consistency on the original CSP.

10. Use the results from problems 6 and 9 to show that in CSPs where variable domains have no more than d elements, and constraints involve no more than a variables, arc consistency closure can be computed in time proportional to $d^a e$ where e is the number of constraints.

11. A Horn clause is a clause with at most one positive literal. Show that arc consistency is complete for Horn clauses, i.e., if arc consistency propagation fails to derive an inconsistency then the given set of Horn clauses is satisfiable. Combined with the results from problem 6, this shows that Horn clause satisfiability is linear time decidable. A different proof of this fact can be found in [Downing and Gallier, 1984].

References

[Downing and Gallier, 1984] William Downing and Jean H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.

[Freuder, 1985] E. C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, 1985.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability, A guide to the Theory of NP-completeness*. Freeman Press, 1979.

[Goldberg, 1979] A. Goldberg. Average case complexity of the satisfiability problem. In *Fourth Workshop on Automated Deduction*, pages 1–6, 1979.

[Kirousis and Papadimitriou, 1988] L. M. Kirousis and C. H. Papadimitriou. The complexity of recognizing polyhedral scenes. *Journal of Computer and Systems Science*, 37(1):14–38, 1988.

[Mackworth, 1977] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–181, 1977.

[Mohr and Henderson, 1986] R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2):225–233, 1986.

[Pearl and Korf, 1987] Judea Pearl and Richard Korf. Search techniques. *Ann. Rev. Comput. Sci.*, 2:451–467, 1987.

[Purdom, 1983] Paul Walton Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[Waltz, 1975] David Waltz. Understanding line drawings of scenes with shadows. In Patrick H. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975.